EDR Psychedelia

Pushing Boundaries of Perception for Reshaping EDR's Truths

SECFORCE

BEACON

Who Am I?

Dimitri (GlenX) Di Cristofaro
@d_glenx

Security consultant and researcher @ SECFORCE LTD

- Red & Purple Teaming
- Malware development
- OS Internals

What Is This Talk About?

1. The Source of Truth - Telemetry Sources
2. Do not Take Sweets from Strangers – RT vs EDR
3. I'm in A Fairy World - Unhooking
4. Playing with Perception – Thread Call Stack Spoofing
5. Brain Confusion - Stealthier Syscalls & APIs
6. Brain Manipulation - Entering Kernel

Information Sources

- Malware database for static analysis
- Telemetry from injected AV DLL
- Telemetry from the OS
- Telemetry from memory scanners
- Telemetry from local network events
- Telemetry from gateways network traffic

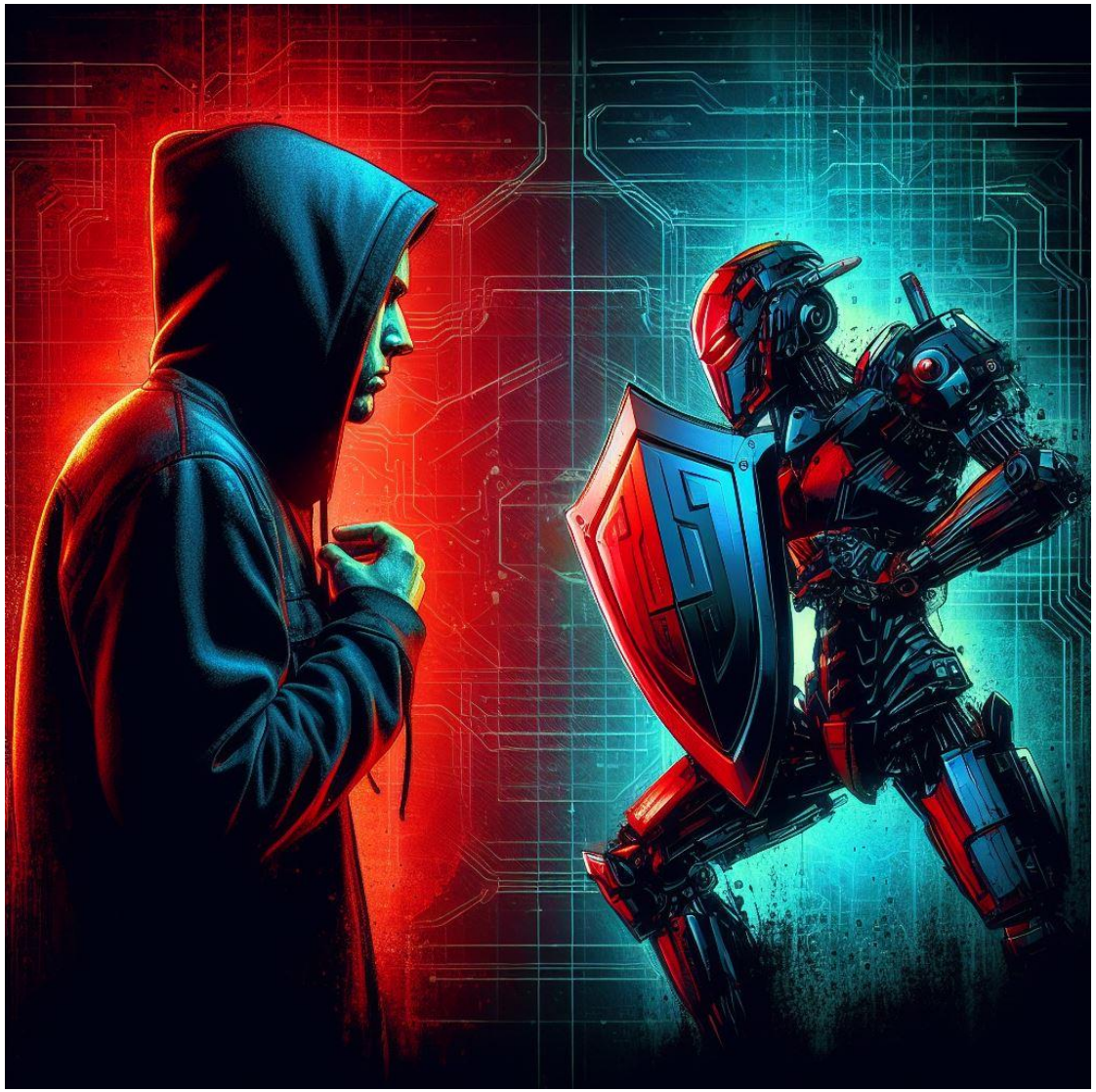
RT vs EDR

ANALYSIS

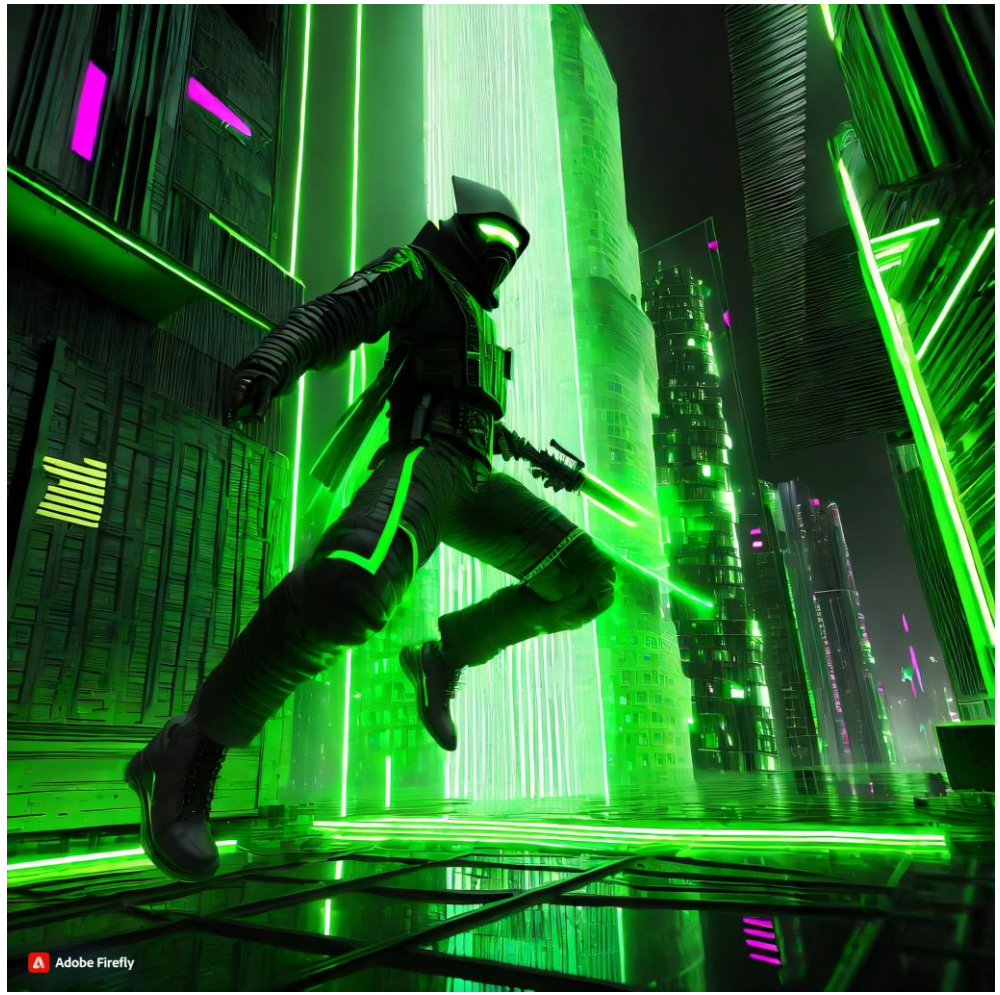
- Identify telemetry sources
- Identify patterns used to trigger a detection event

IMPLEMENTATION

- Operate under the detection threshold using legitimate tools/TTPs/APIs to hide in the noise
- Block telemetry
- Poison telemetry



RT vs EDR



Operate under the detection threshold

A score is assigned for every monitored event. If a given threshold is reached, an alert is triggered

- Domain Fronting , HTTPS traffic shaping
- LOLBins , Trusted binaries
- DLL Sideloading , DLL Proxying
- Syscalls , Less common APIs , Dynamic API resolution

RT vs EDR

Block telemetry

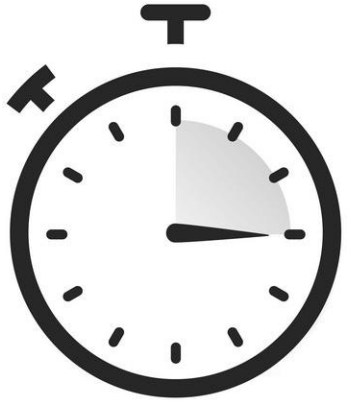
- Remove hooks
- Patch logging functions (e.g. AMSI, ETW)
- Kill sensors

RT vs EDR

Poison telemetry

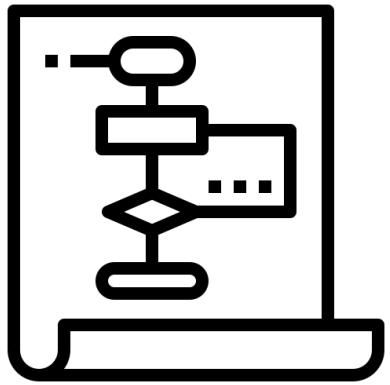
- Identify what information the malware controls
- Modify the information to conceal malicious actions
- Drug the telemetry sources to confuse the security brain

Some EDR Strategies



Threads may be periodically scanned. Yara rules may be applied to memory dumps and the stack of the threads may be analysed

Hooked Windows APIs allow process execution monitoring. Calls to dangerous APIs may be traced and logged



Condition-based analysis

- RWX memory regions
- RX memory regions not backed by a file
- Sleeping Threads
- Image loaded but module structure not linked to PEB
- ...

Dynamic Analysis – Behavioural Detection

- Suspicious sequence of operations
  - CreateFile(ntdll) -> ReadFile() (unhook NTDLL)
  - VirtualAlloc -> WriteProcessMemory -> CreateRemoteThread (process injection)

- Windows events correlation
  - Image Mapped -> Thread Created
  - Get Handle to lsass.exe

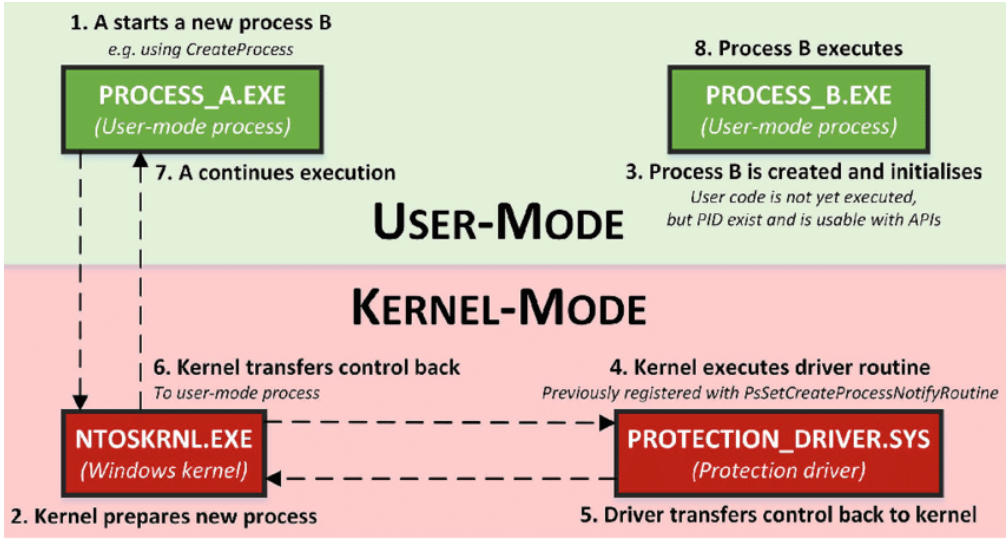
- Memory Dumps
  - Pattern matching (e.g. YARA)
  - Strings (e.g. IPs, domains, function names, etc.)

- Dotnet
  - AMSI
  - ETW

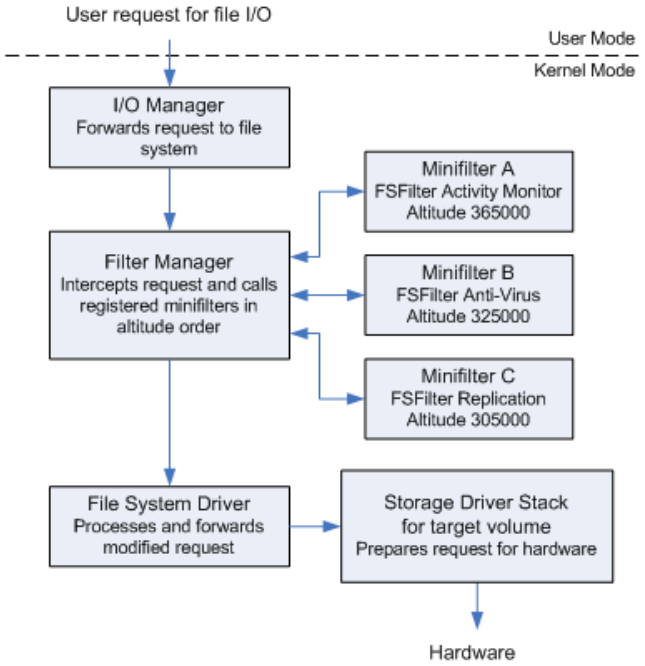
Dynamic Analysis – Behavioural Detection

Kernel Callbacks

- PsSetCreateProcessNotifyRoutine
- PsSetCreateThreadNotifyRoutine
- PsSetLoadImageNotifyRoutine



Src: Fast and Furious: Outrunning Windows Kernel Notification Routines from User-Mode http://dx.doi.org/10.1007/978-3-030-52683-2_4



Src: https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/filter-manager-concepts

Minifilters

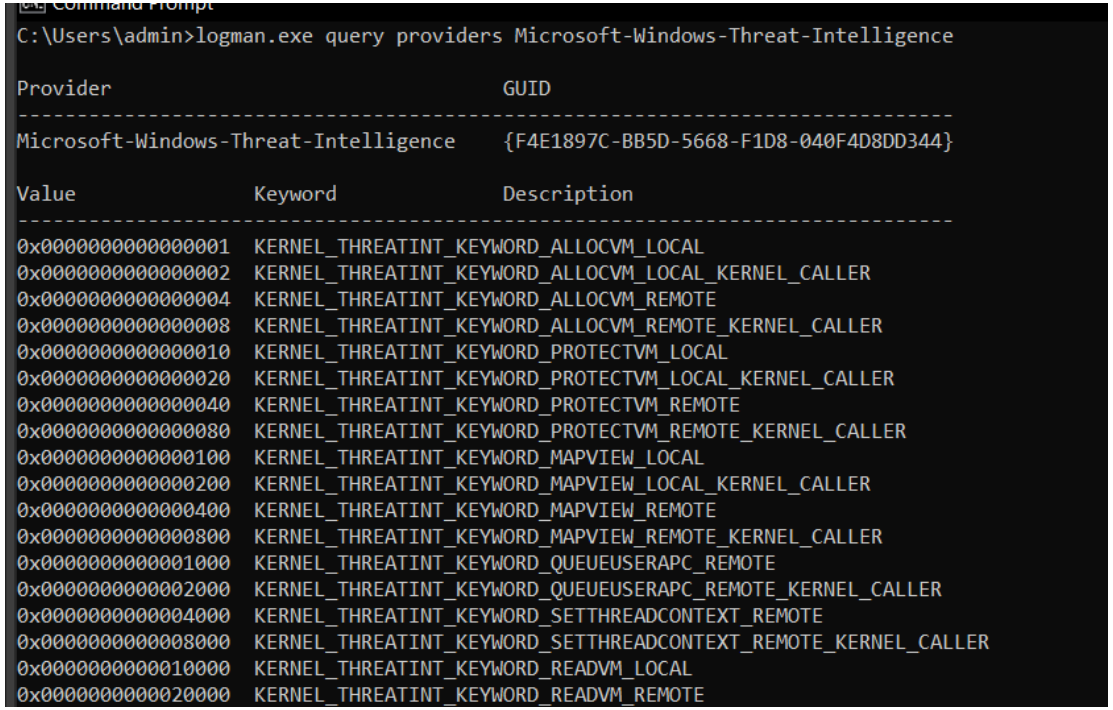
- I/O Activity
- Execution of queued minifilters based on Altitude value

Dynamic Analysis – ETW Ti

`Microsoft-Windows-Threat-Intelligence` ETW provider (or `EtwTi` in short)

Kernel instrumented to log calls to some routines (Memory Management, APC, Threads, etc.)

Clients registered to this provider will get notified when a number of potentially malicious actions are executed (e.g. Memory Allocation, Memory Read, APC Queued, etc.).



Get more Information about the `EtwTi` provider:

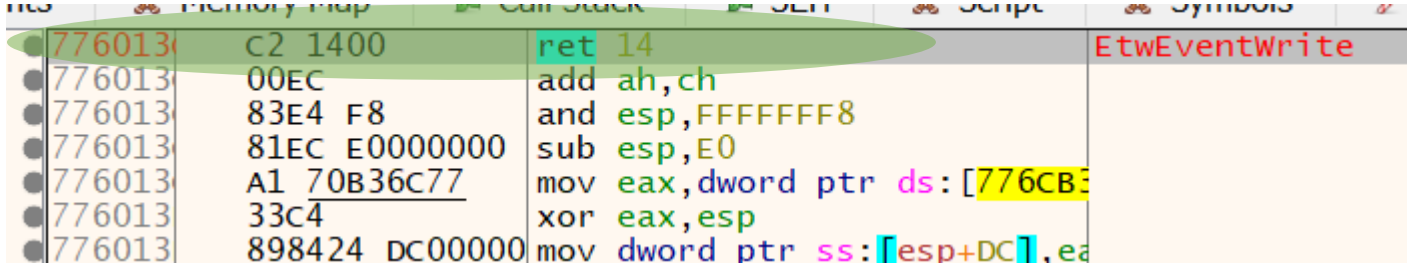
```
logman.exe query providers Microsoft-Windows-Threat-Intelligence
```

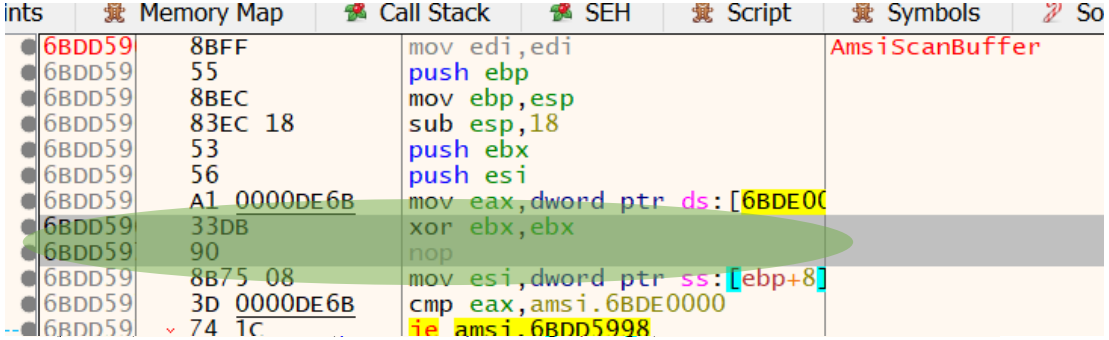
Behavioural Analysis – AMSI / ETW

Different ways of disabling monitoring

- Memory patching
  - Force the function to always return SUCCESS
  - Tamper function parameters

- Hooking
  - Using hooking engine (e.g. Detours)
  - Using HW Breakpoints





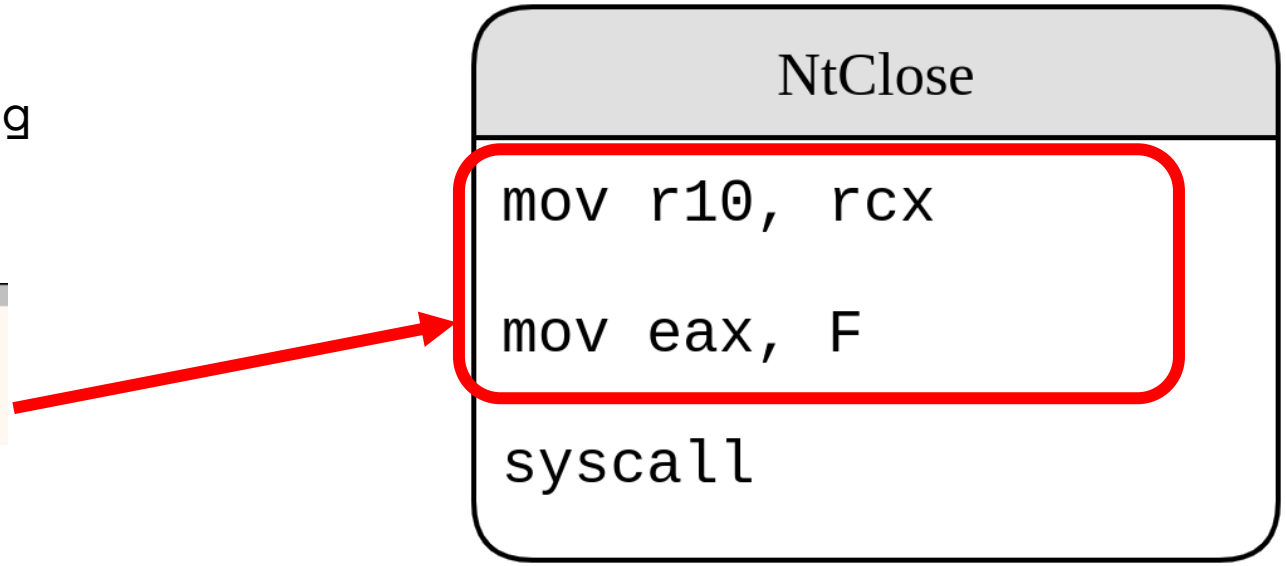
Hooking

- AVs use hooks to monitor processes
- AVs **MUST hook** because the kernel does not provide kernel callbacks for every possible syscall.
  - It actually does but it's not documented - `PsAltSystemCallHandlers`

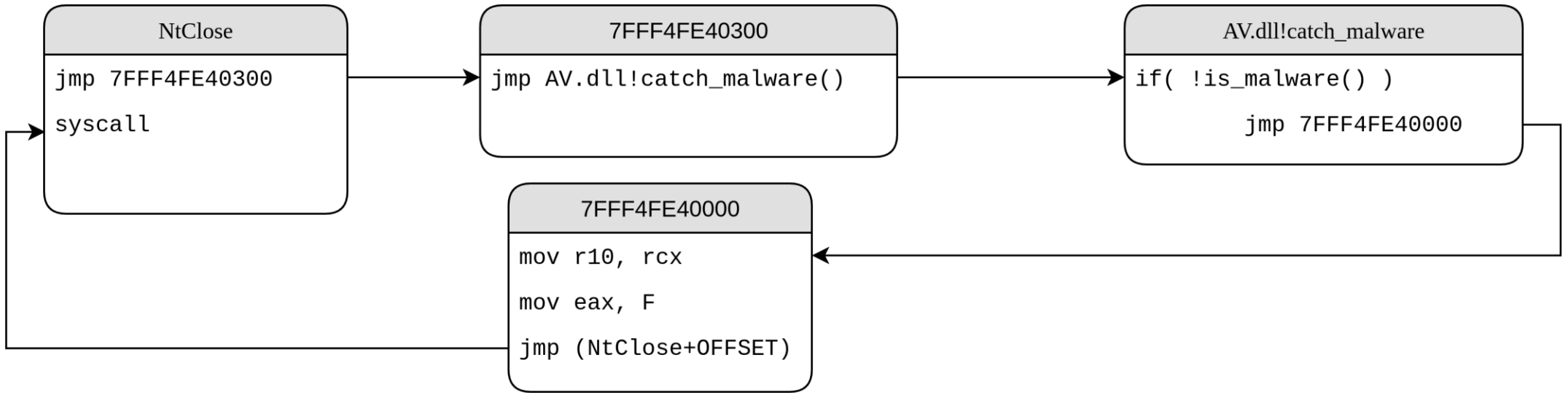
Hijack the execution flow to allow monitoring

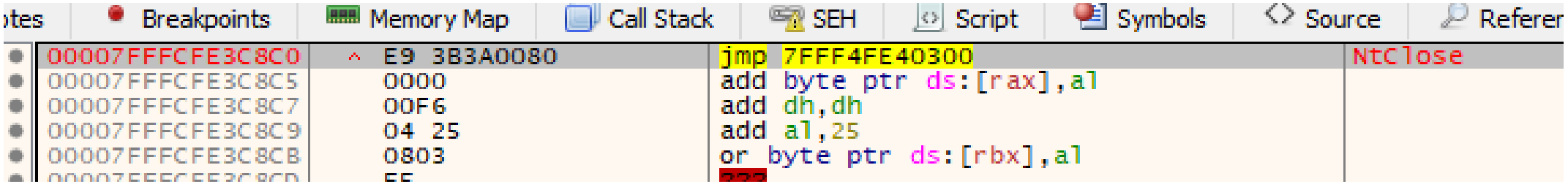
- Patch prologue of the function to hook

```
4C:8BD1          mov r10,rcx                                  NtClose
B8 0F000000      mov eax,F
F60425 0803FE7F 01 test byte ptr ds:[7FFE0308],1
75 03            jne ntdll.7FFFCFE3C8D5
0F05             syscall
C3               ret
CD 2E            int 2E
```



Hooking


NtClose
jmp 7FFF4FE40300
syscall
7FFF4FE40300
jmp AV.dll!catch_malware()
AV.dll!catch_malware
if(!is_malware())
jmp 7FFF4FE40000
7FFF4FE40000
mov r10, rcx
mov eax, F
jmp (NtClose+OFFSET)


Breakpoints
Memory Map
Call Stack
SEH
Script
Symbols
Source
Referer
00007FFFCFE3C8C0 E9 3B3A0080 jmp 7FFF4FE40300 NtClose
00007FFFCFE3C8C5 0000 add byte ptr ds:[rax],al
00007FFFCFE3C8C7 00F6 add dh,dh
00007FFFCFE3C8C9 04 25 add al,25
00007FFFCFE3C8CB 0803 or byte ptr ds:[rbx],al

Unhooking

Removing user-space hooks to make AV “blind”

Many different techniques:

- Shellycoat
Upayan's (@slaeryan) ~~implementation~~
the repository does not exist anymore :(

- Perun's Fart – Sektor7 blog post

- Whisper2Shout – our blog post :)

Unhooking – Shellycoat

1. Map clean NTDLL from disk (NtCreateFile, NtCreateSection, NtMapViewOfSection)
2. Overwrite .text section of the hooked NTDLL (NtProtectVirtualMemory, memcpy)
3. Unmap "second" NTDLL (NtUnmapViewOfSection)

Pitfalls

- DLL (e.g. NTDLL)mapped twice in memory (temporary)
- Syscalls / API needed to map/unmap images. EDRs may use kernel callbacks to detect image loading events
- Some AVs may check hook integrity

Perun's Fart

- Create a sacrificial process in suspended state
  - NTDLL has not been hooked yet because the callback to inject AV DLL has not been called yet.
- Copy clean NTDLL from the suspended process
- Terminate sacrificial process
- Overwrite the hooked NTDLL with the clean one

Pitfalls

- Need to create a new process
- Some AVs may check hook integrity
- May need VirtualProtect on NTDLL to make it temporarily RWX

Unhooking – Whisper2Shout

1. Walk the process address space to find the original prologue stub (math , NtQueryVirtualMemory)

2. Patch memory to skip the hook (NtProtectVirtualMemory, memcpy)

Blog Post: https://www.secforce.com/blog/whisper2shout-unhooking-technique/

Where Are The Stubs?

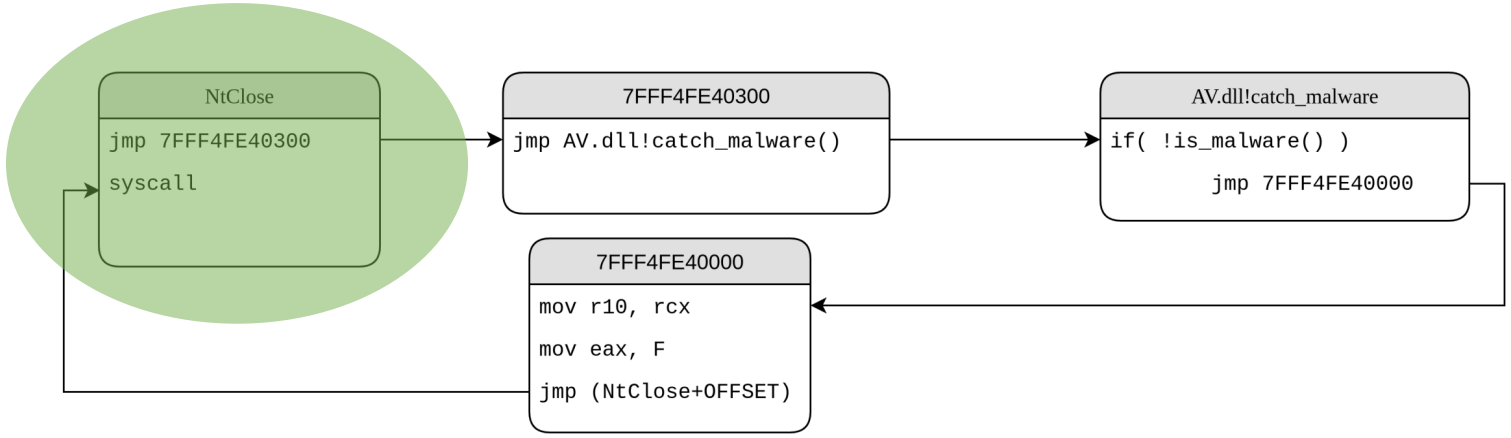
github.com/microsoft/Detours/blob/master/src/detours.cpp

```
1241 static PVOID detour_alloc_region_from_lo(PBYTE pbLo, PBYTE pbHi)
1242 {
1243     PBYTE pbTry = detour_alloc_round_up_to_region(pbLo);
1244
1245     DETOUR_TRACE((" Looking for free region in %p..%p from %p:\n", pbLo, pbHi, pbTry)
1246
1247     for (; pbTry < pbHi;) {
1248         MEMORY_BASIC_INFORMATION mbi;
1249
1250         if (pbTry >= s_pSystemRegionLowerBound && pbTry <= s_pSystemRegionUpperBound)
1251             // Skip region reserved for system DLLs, but preserve address space entro
1252             pbTry += 0x08000000;
1253             continue;
1254         }
1255
1256         ZeroMemory(&mbi, sizeof(mbi));
1257         if (!VirtualQuery(pbTry, &mbi, sizeof(mbi))) {
1258             break;
1259         }
1260
1261         DETOUR_TRACE((" Try %p => %p..%p %6lx\n",
1262                       pbTry,
1263                       mbi.BaseAddress,
1264                       (PBYTE)mbi.BaseAddress + mbi.RegionSize - 1,
1265                       mbi.State));
1266
1267         if (mbi.State == MEM_FREE && mbi.RegionSize >= DETOUR_REGION_SIZE) {
1268
1269             PVOID pv = VirtualAlloc(pbTry,
1270                                     DETOUR_REGION_SIZE,
1271                                     MEM_COMMIT|MEM_RESERVE,
1272                                     PAGE_EXECUTE_READWRITE);
1273             if (pv != NULL) {
1274                 return pv;
1275             }
1276             else if (GetLastError() == ERROR_DYNAMIC_CODE_BLOCKED) {
```

Where Are The Stubs?

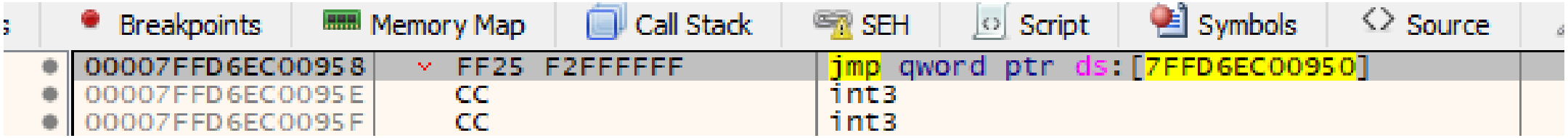
`Ntdll!LdrLoadDll` hooked by AVG

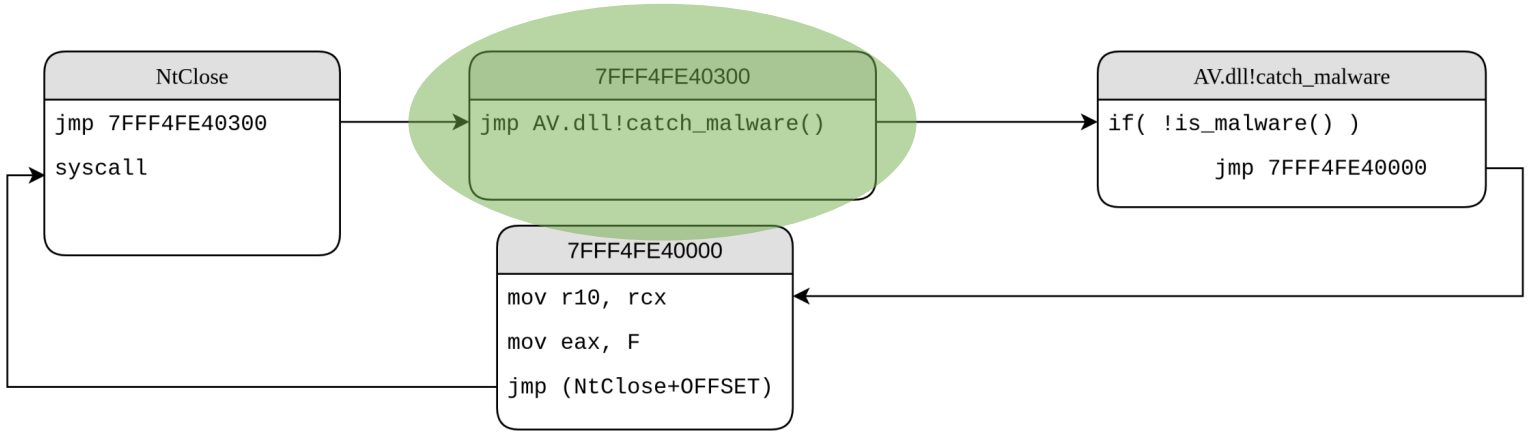
```
00007FFDAEBA1600  ^ E9 53F305C0          jmp 7FFD6EC00958          LdrLoadDll
00007FFDAEBA1605    CC                   int3
00007FFDAEBA1606    57                   push rdi
00007FFDAEBA1607    41:56                push r14
00007FFDAEBA1609    48:81EC D0000000     sub rsp,D0
00007FFDAEBA1610    48:8B05 E9BE1500     mov rax,qword ptr ds:[7FFDAECFD500]
00007FFDAEBA1617    48:33C4              xor rax,rsp
```



Where Are The Stubs?

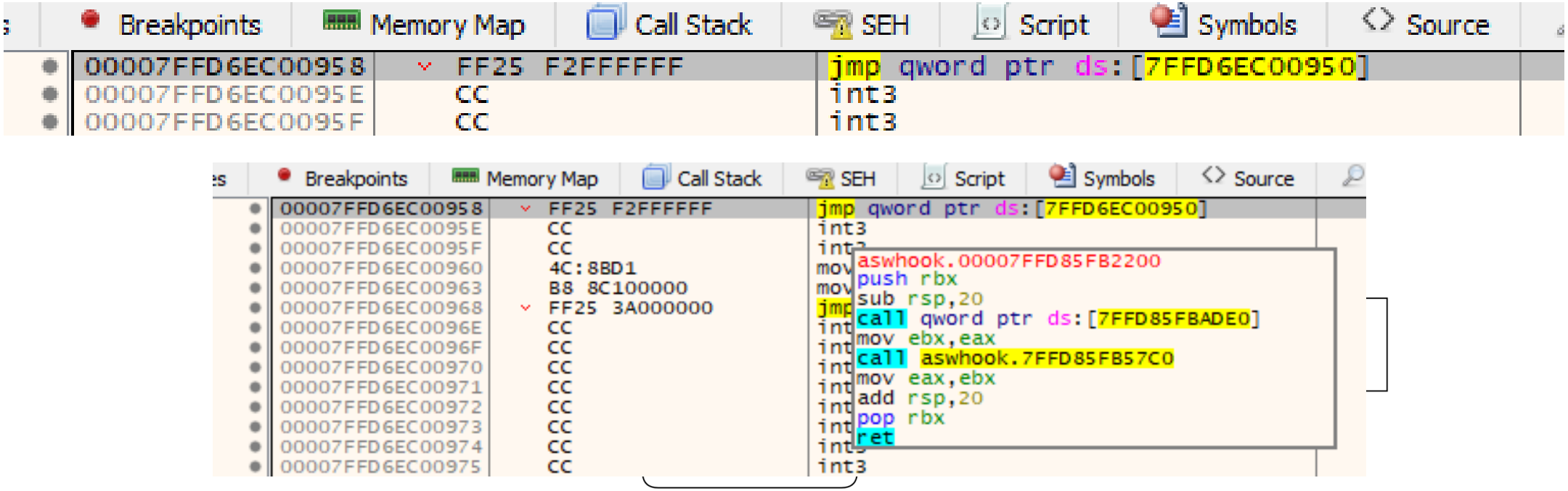
"Hooking" stub





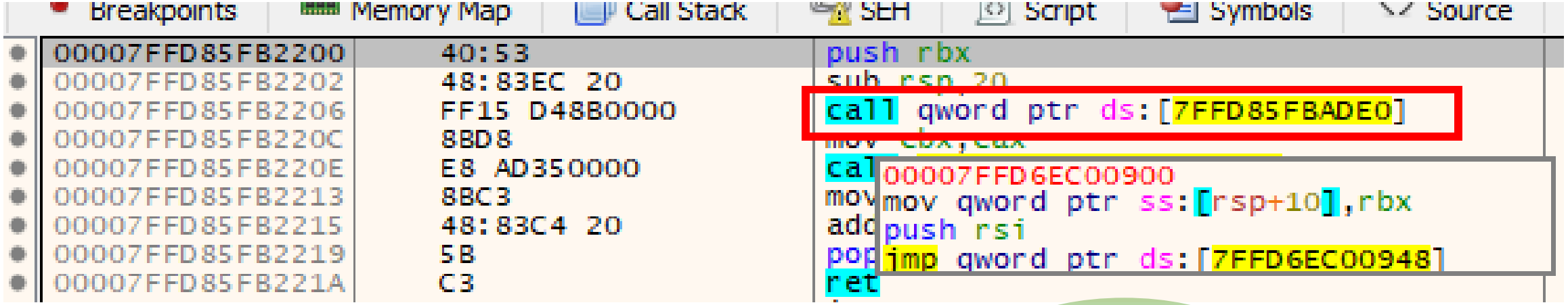
Where Are The Stubs?

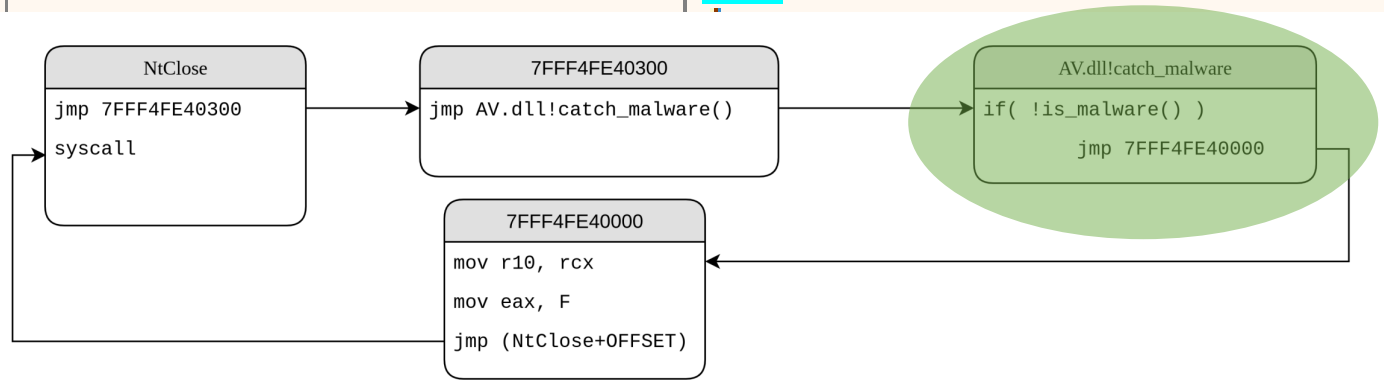
"Hooking" stub



Where Are The Stubs?

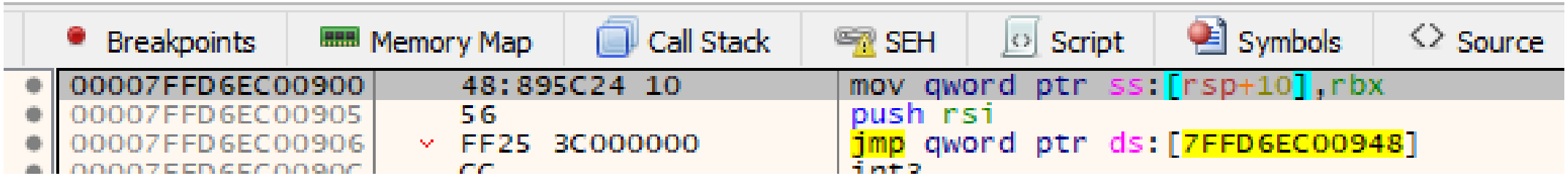
AVG DLL

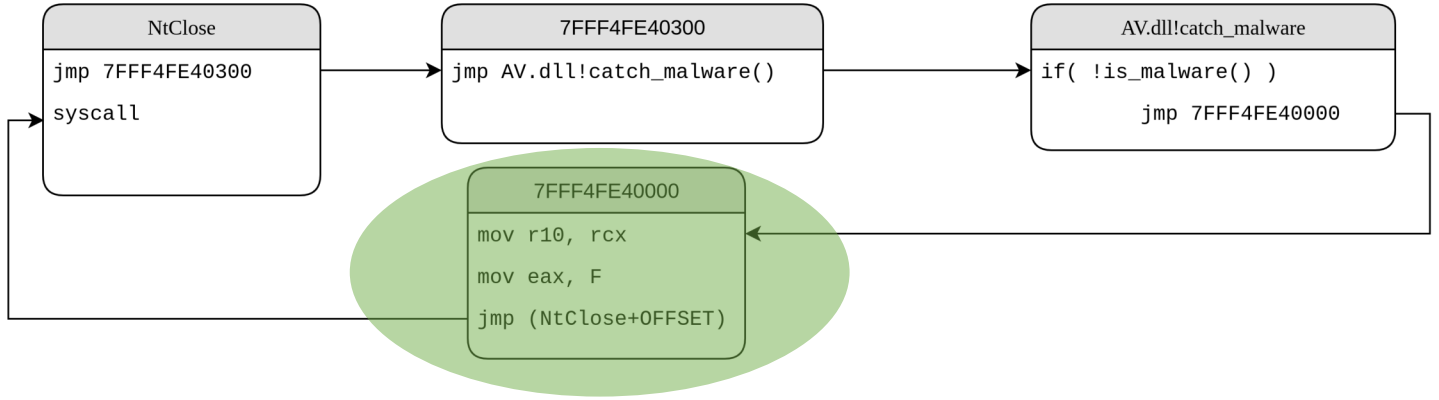




Where Are The Stubs?

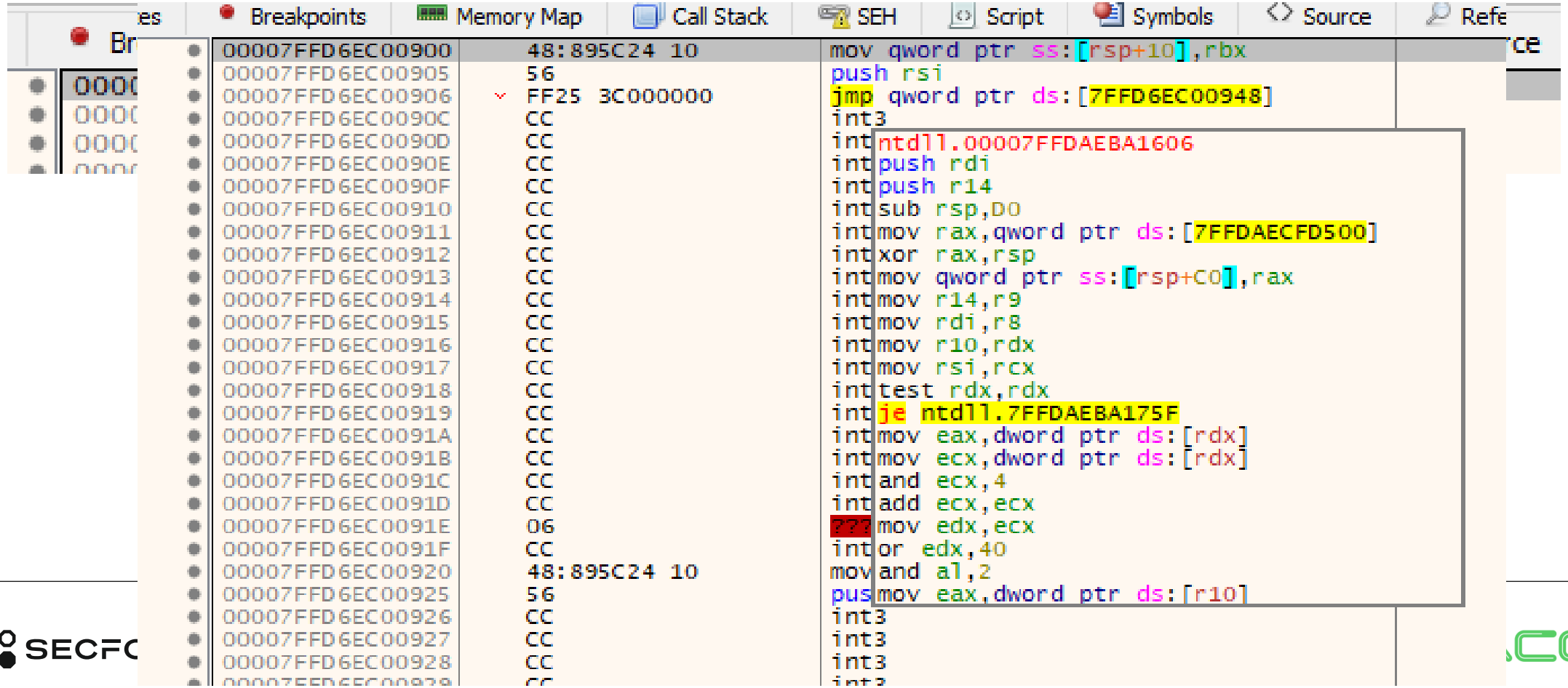
Trampoline to come back to the original function (AVG)





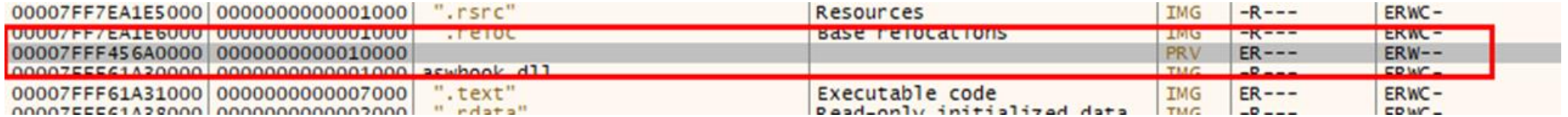
Where Are The Stubs?

Trampoline to come back to the original function (AVG)

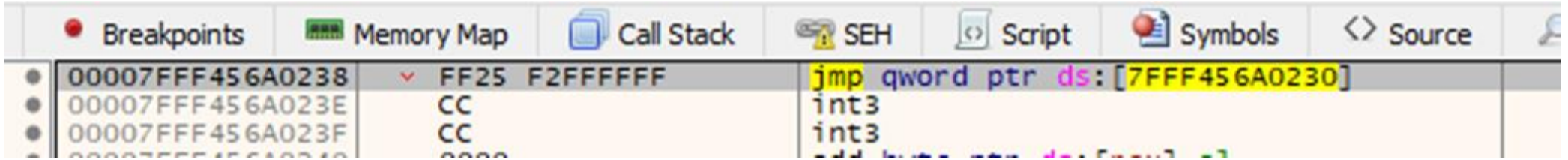


Where Are The Stubs?

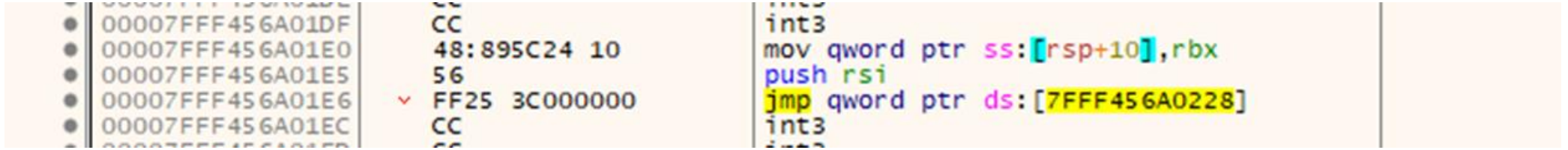
Memory area hosting the stubs (AVG)



"Hooking" stub (AVG)



Trampoline to come back to the original function (AVG)



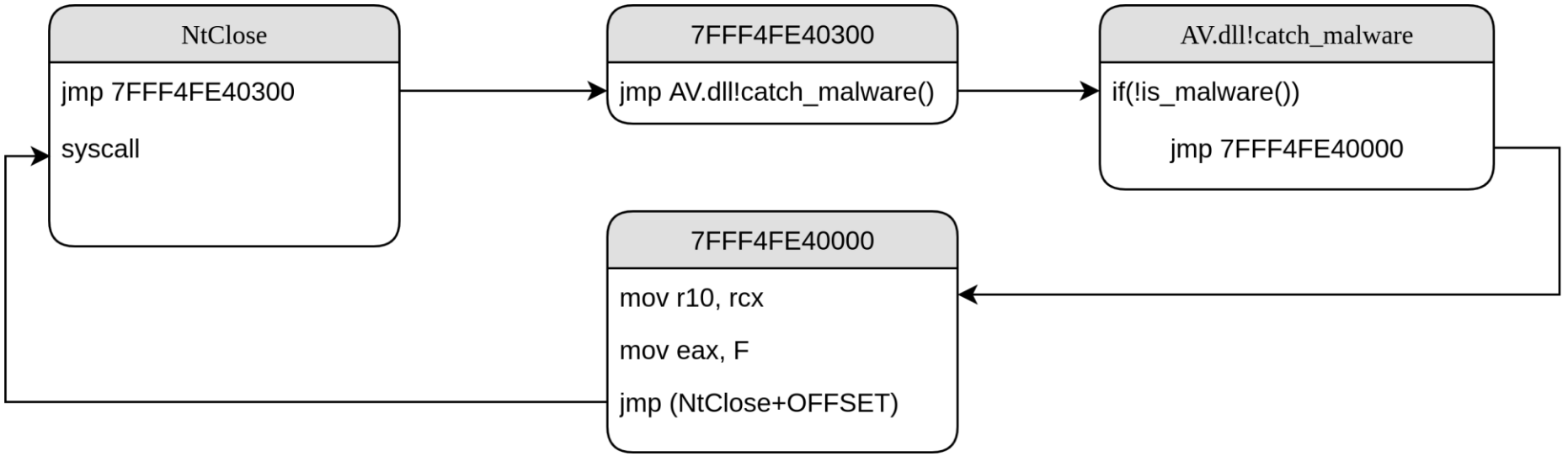
Where Are The Stubs?

- The "hooking" stub is stored in a memory area allocated with NtAllocateVirtualMemory
  - There is a reference to this memory area in the first bytes of the hooked function

- The original prologue is stored in a memory area allocated with NtAllocateVirtualMemory
  - Contains a jump to an address near the function we are unhooking

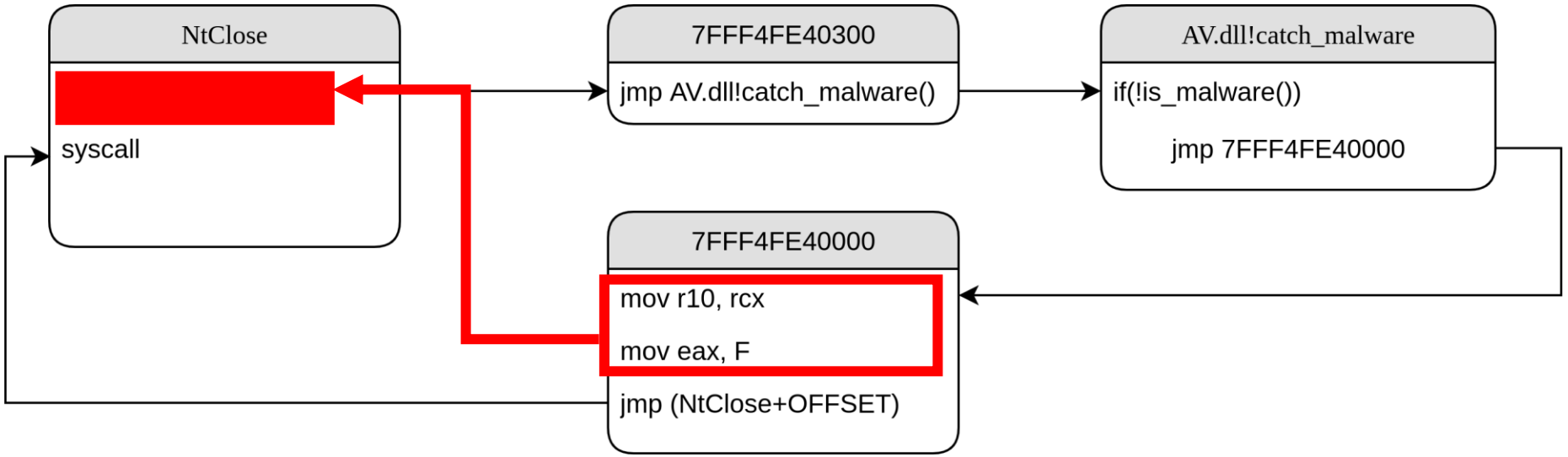
UNHOOK IDEA

Walk the pointers to retrieve the original prologue



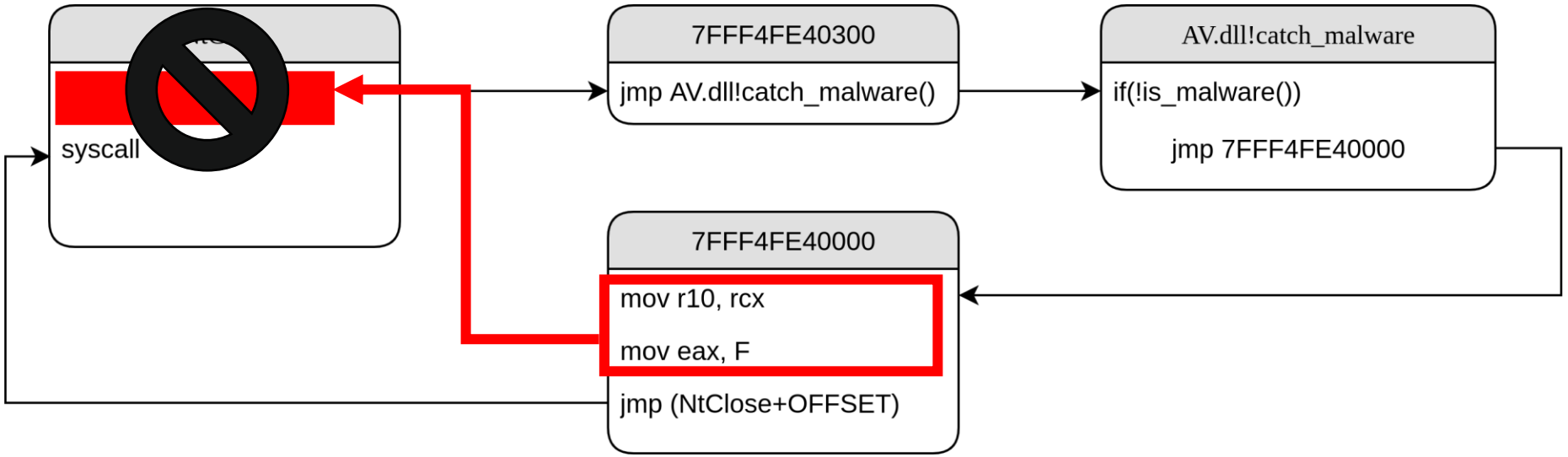
UNHOOK IDEA

Overwrite the prologue with the original instructions



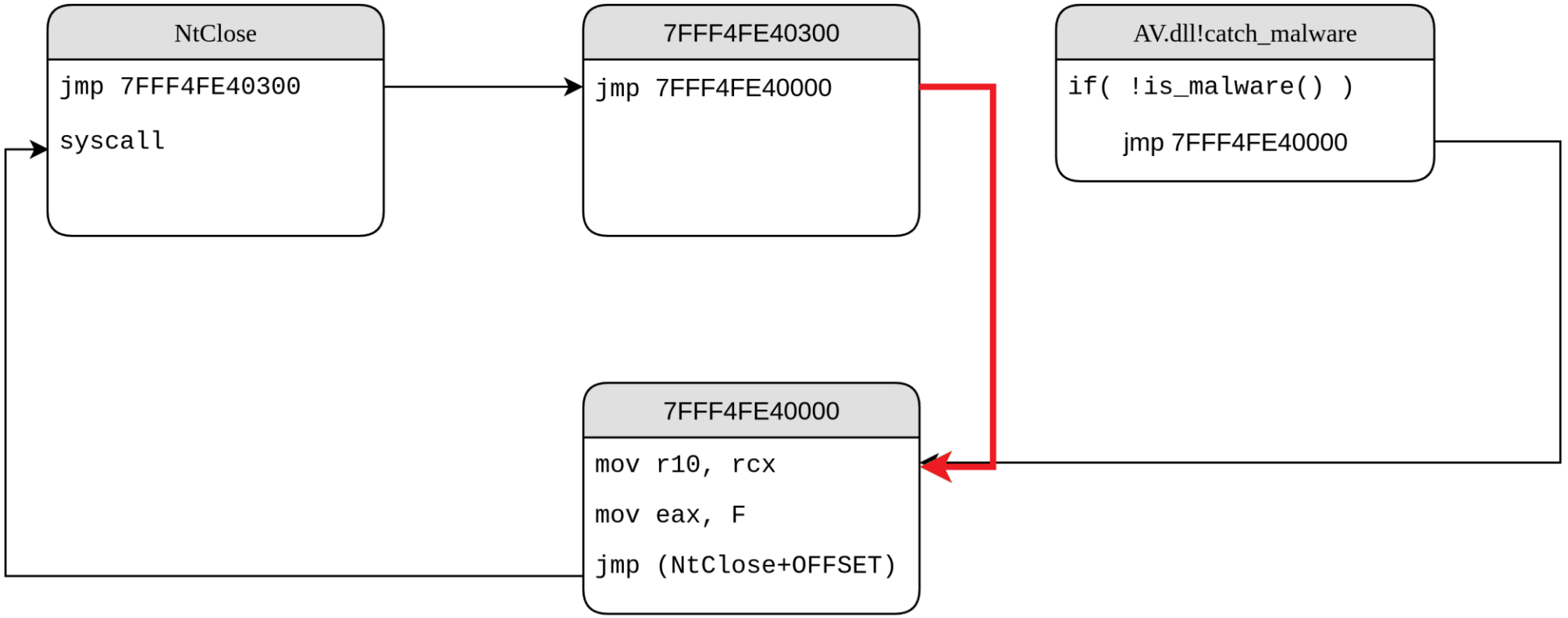
UNHOOK IDEA

Some AVs periodically check if the hooks are in place



UNHOOK IDEA

So... Patch the "hooking" stub :)



UNHOOK IDEA

Selectively unhook only when needed

Pre-define at compile time the functions to unhook

OR

Wrap all APIs in your malware with an helper function that checks for hooks before calling them

Unhooking Tips

- When enumerating hooked functions consider API forwarders in your code

- If your malware (or anything your malware loads) calls `LoadLibrary` (or any similar function), the loaded library may be hooked.
  - Every time a library is loaded check if it has been hooked and unhook it if necessary.

- Do extra checks when enumerating loaded DLLs : Some EDRs overwrite the `DllBase` attribute in `LDR_DATA_TABLE_ENTRY` .
  - When dereferencing the pointer the process will crash. `OriginalBase` attribute can be used instead
  - `NTDLL` and `Kernel32` should always have `DllBase` == `OriginalBase`
  - **CAUTION:** It is not 100% guaranteed that the `OriginalBase` value contains the right value. Some DLLs (e.g. `rpcrt4.dll`) make use of this value and the address may not be in the process address space. This may result in false positives when checking `DllBase` != `OriginalBase`

Unhooking – Caveats

- At least one vendor does **not** use Private Memory regions to store the trampolines to support the hooks.
  - Code caves inside NTDLL are used instead

Am I chasing my own tail?

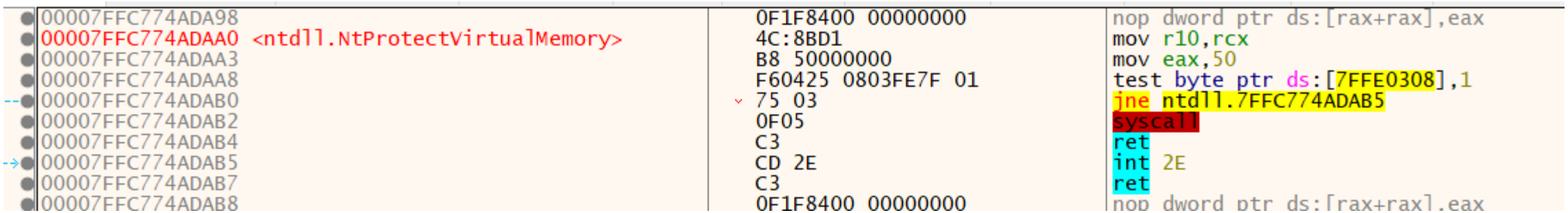
How do I unhook myself if the APIs I need are hooked?

Am I chasing my own tail?

How do I unhook myself if the APIs I need are hooked?

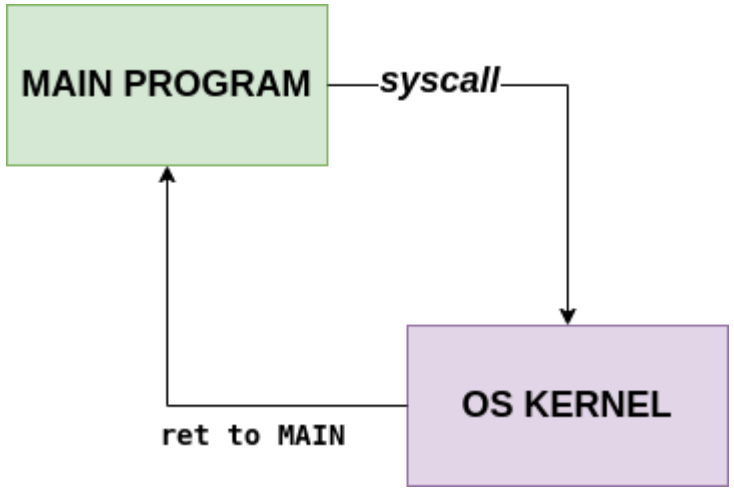
We don't need any Windows Library.
OS "assistance" is needed only for:

- **(self) Memory information query (`NtQueryVirtualMemory`)**
- **(self) Memory protection change (`NtProtectVirtualMemory`)**



Caveat: ETW Ti can be used to detect memory permission changes from the kernel.
https://github.com/jdu2600/EtwTi-FluctuationMonitor

Direct System Calls



Detection Strategies:

- Static Analysis
- Thread Call Stack

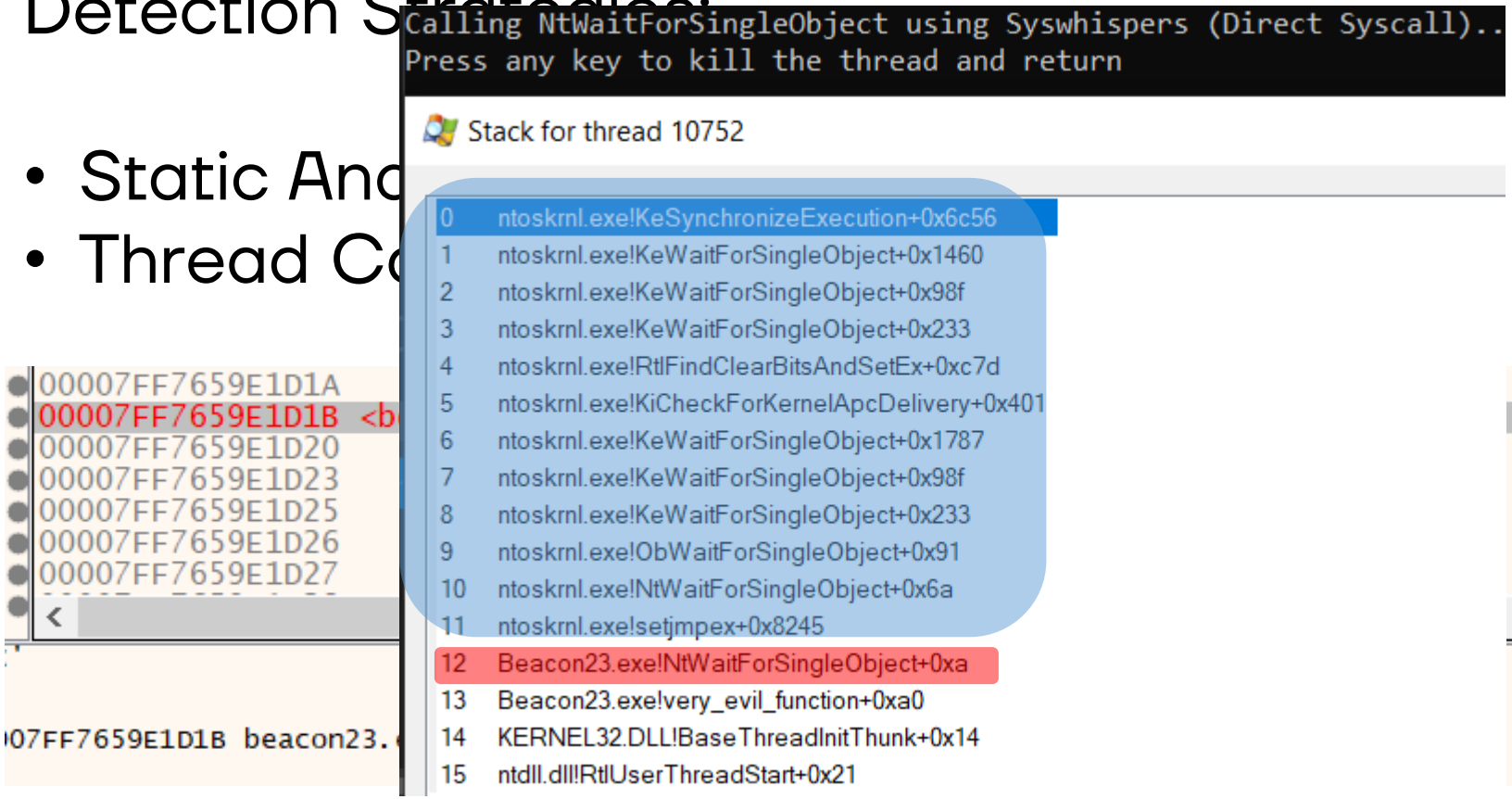
```
00007FF7659E1D1A                C3              ret
00007FF7659E1D1B <beacon23.NtWaitForSingleObject>  B8 04000000  mov eax,4
00007FF7659E1D20                4C:8BD1         mov r10,rcx
00007FF7659E1D23                0F05            syscall
00007FF7659E1D25                C3              ret
00007FF7659E1D26                CC              int3
00007FF7659E1D27                CC              int3
```

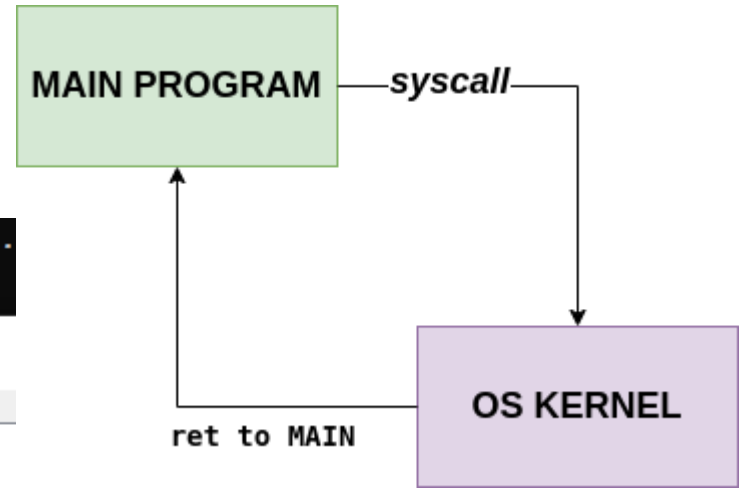
```
07FF7659E1D1B beacon23.exe:$11D1B #111B <NtWaitForSingleObject>
```

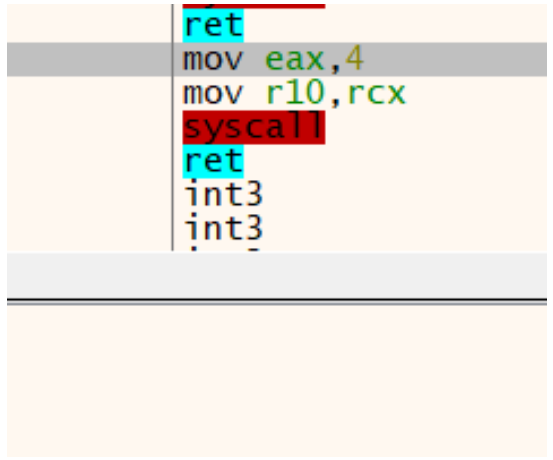
Direct System Calls

Detection Strategies:

- Static And
- Thread Co

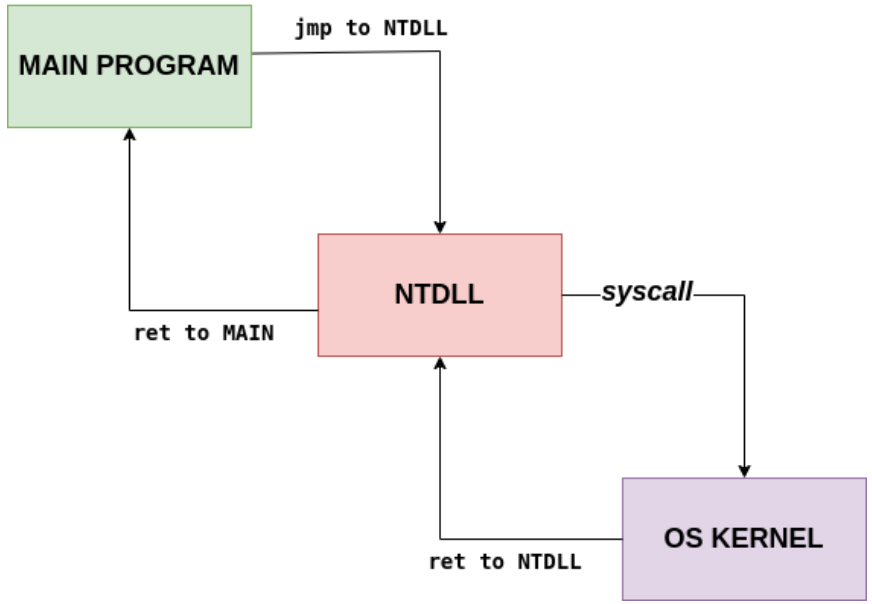


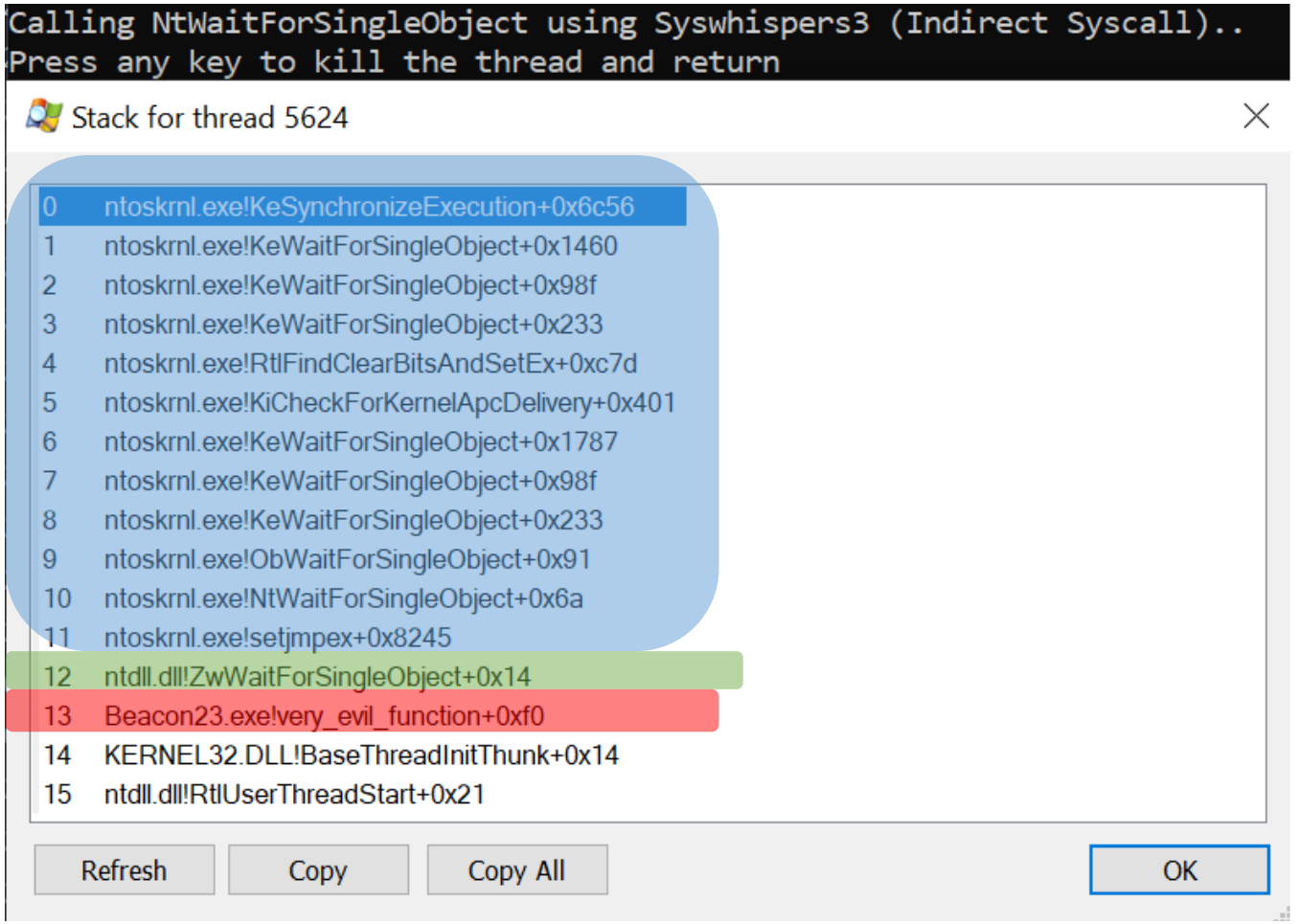




Indirect System Calls

- No Syscall Instruction
- Nicer Thread Call Stack





Indirect System Calls (Randomized)

- The source of the syscall is ntdll
- The real target syscall is hidden in userspace

Stack - thread 10804

	Name
0	ntdll.dll!ZwAccessCheckAndAuditAlarm+0x14
1	Beacon23.exe!very_evil_function+0xf0
2	kernel32.dll!BaseThreadInitThunk+0x14
3	ntdll.dll!RtlUserThreadStart+0x21

User space View

Stack for thread 10804

0	ntoskrnl.exe!KeSynchronizeExecution+0x6c56
1	ntoskrnl.exe!KeWaitForSingleObject+0x1460
2	ntoskrnl.exe!KeWaitForSingleObject+0x98f
3	ntoskrnl.exe!KeWaitForSingleObject+0x233
4	ntoskrnl.exe!RtlFindClearBitsAndSetEx+0xc7d
5	ntoskrnl.exe!KiCheckForKernelApcDelivery+0x401
6	ntoskrnl.exe!KeWaitForSingleObject+0x1787
7	ntoskrnl.exe!KeWaitForSingleObject+0x98f
8	ntoskrnl.exe!KeWaitForSingleObject+0x233
9	ntoskrnl.exe!ObWaitForSingleObject+0x91
10	ntoskrnl.exe!NtWaitForSingleObject+0x6a
11	ntoskrnl.exe!setjmpex+0x8245
12	ntdll.dll!ZwAccessCheckAndAuditAlarm+0x14
13	Beacon23.exe!very_evil_function+0xf0
14	KERNEL32.DLL!BaseThreadInitThunk+0x14
15	ntdll.dll!RtlUserThreadStart+0x21

Call Stack Analysis

Thread	Address	To	From	Size	Comment	Party
8548						
	000000CA004FF668	00007FFC74C94268	00007FFC77[illegible]D3B2	4[illegible]	ntdll.00007FFC774AD3B2	System
	000000CA004FF6A8	00007FF7812C19F7	00007FFC[illegible]94268	170	kernelbase.00007FFC74C94268	User
	000000CA004FF818	00007FF7812C1890	00007FF7[illegible]2C19F7	100	beacon23.very_evil_function+57	User
	000000CA004FF918	00007FF7812C23F9	00007FF7[illegible]2C1890	50	beacon23.main+20	User
	000000CA004FF968	00007FF7812C229E	00007FF7[illegible]2C23F9	70	beacon23.invoke_main+39	User
	000000CA004FF9D8	00007FF7812C215E	00007FF7[illegible]2C229E	30	beacon23.__scrt_common_main_seh+12E	User
	000000CA004FFA08	00007FF7812C248E	00007FF7[illegible]2C215E	30	beacon23.__scrt_common_main+E	User
	000000CA004FFA38	00007FFC75A37614	00007FF78[illegible]248E	30	beacon23.mainCRTStartup+E	System
	000000CA004FFA68	00007FFC774626F1	00007FFC75A[illegible]614		kernel32.00007FFC75A37614	System
	000000CA004FFAE8	0000000000000000	00007FFC77462[illegible]		ntdll.00007FFC774626F1	User

- Leak our intentions – Which function is being called?
- Leak the source of the call – Which module (and which offset) called the function?
- Suspicious if malformed (i.e. not unwindable) – ☺

Call Stack Analysis - Target

Leak our intentions – Which function is being called?

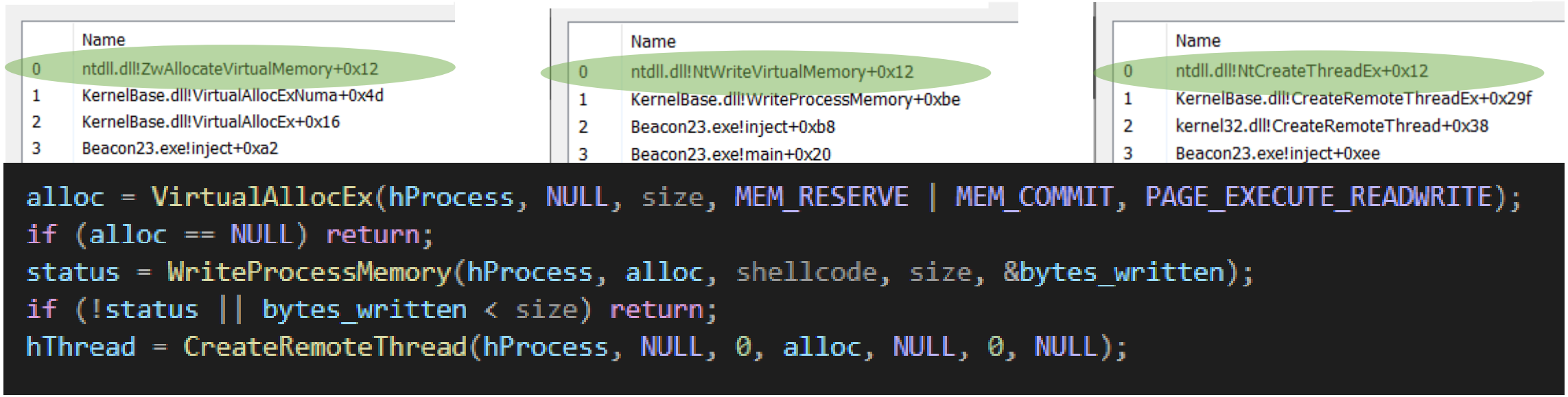
	Name
0	ntdll.dll!ZwAllocateVirtualMemory+0x12
1	KernelBase.dll!VirtualAllocExNuma+0x4d
2	KernelBase.dll!VirtualAllocEx+0x16
3	Beacon23.exe!inject+0xa2
4	Beacon23.exe!main+0x828
5	Beacon23.exe!invoke_main+0x39
6	Beacon23.exe!__scrt_common_main_seh+0x12e
7	Beacon23.exe!__scrt_common_main+0xe
8	Beacon23.exe!mainCRTStartup+0xe
9	kernel32.dll!BaseThreadInitThunk+0x14
10	ntdll.dll!RtlUserThreadStart+0x21

	Name
0	ntdll.dll!NtWriteVirtualMemory+0x12
1	KernelBase.dll!WriteProcessMemory+0xbe
2	Beacon23.exe!inject+0xb8
3	Beacon23.exe!main+0x20
4	Beacon23.exe!invoke_main+0x39
5	Beacon23.exe!__scrt_common_main_seh+0x12e
6	Beacon23.exe!__scrt_common_main+0xe
7	Beacon23.exe!mainCRTStartup+0xe
8	kernel32.dll!BaseThreadInitThunk+0x14
9	ntdll.dll!RtlUserThreadStart+0x21

	Name
0	ntdll.dll!NtCreateThreadEx+0x12
1	KernelBase.dll!CreateRemoteThreadEx+0x29f
2	kernel32.dll!CreateRemoteThread+0x38
3	Beacon23.exe!inject+0xee
4	Beacon23.exe!main+0x20
5	Beacon23.exe!invoke_main+0x39
6	Beacon23.exe!__scrt_common_main_seh+0x12e
7	Beacon23.exe!__scrt_common_main+0xe
8	Beacon23.exe!mainCRTStartup+0xe
9	kernel32.dll!BaseThreadInitThunk+0x14
10	ntdll.dll!RtlUserThreadStart+0x21

Call Stack Analysis - Target

Leak our intentions – Which function is being called?



	Name
0	ntdll.dll!ZwAllocateVirtualMemory+0x12
1	KernelBase.dll!VirtualAllocExNuma+0x4d
2	KernelBase.dll!VirtualAllocEx+0x16
3	Beacon23.exe!inject+0xa2

	Name
0	ntdll.dll!NtWriteVirtualMemory+0x12
1	KernelBase.dll!WriteProcessMemory+0xbe
2	Beacon23.exe!inject+0xb8
3	Beacon23.exe!main+0x20

	Name
0	ntdll.dll!NtCreateThreadEx+0x12
1	KernelBase.dll!CreateRemoteThreadEx+0x29f
2	kernel32.dll!CreateRemoteThread+0x38
3	Beacon23.exe!inject+0xee

```
alloc = VirtualAllocEx(hProcess, NULL, size, MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
if (alloc == NULL) return;
status = WriteProcessMemory(hProcess, alloc, shellcode, size, &bytes_written);
if (!status || bytes_written < size) return;
hThread = CreateRemoteThread(hProcess, NULL, 0, alloc, NULL, 0, NULL);
```

Call Stack Analysis - Source

Leak the source of the call – Which module (and at which offset) called the function?

Stack - thread 10708

	Name
0	ntdll.dll!ZwWaitForSingleObject+0x14
1	KernelBase.dll!WaitForSingleObjectEx+0x8e
2	Beacon23.exe!main+0x6a
3	Beacon23.exe!__scrt_common_main_seh+0x10c
4	kernel32.dll!BaseThreadInitThunk+0x14
5	ntdll.dll!RtlUserThreadStart+0x21

Main Thread calling `WaitForSingleObject`

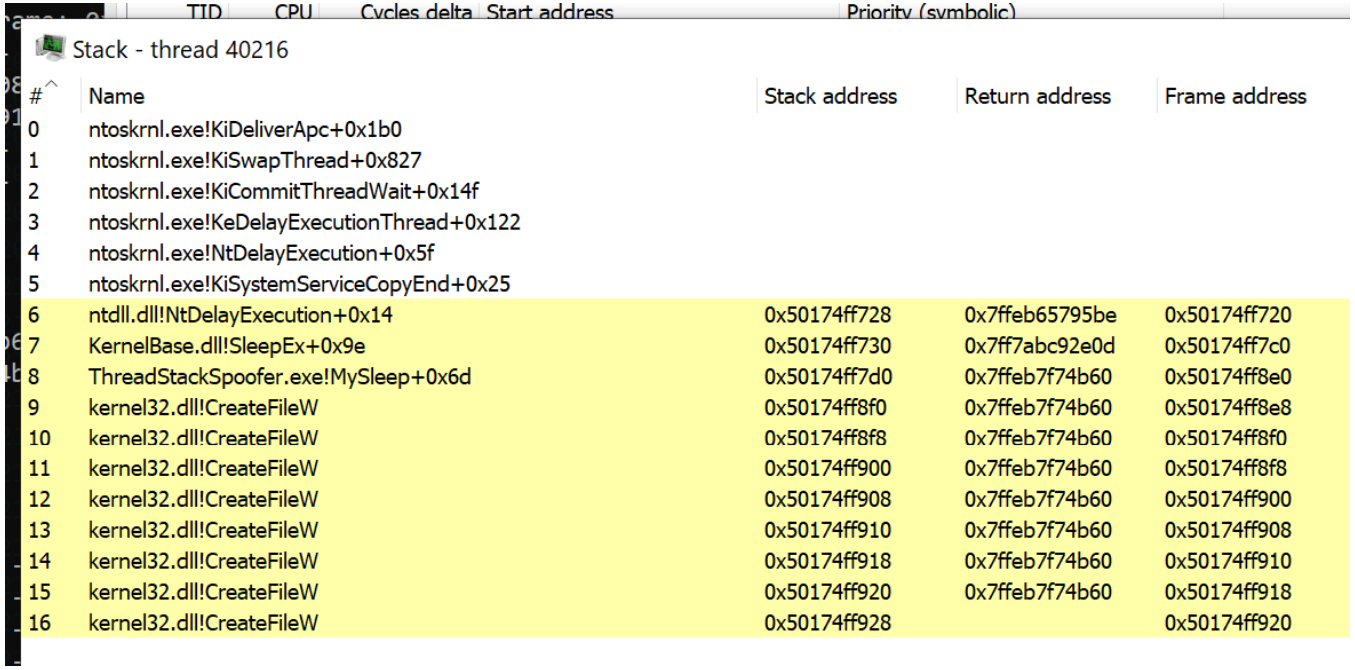
Stack - thread 5348

	Name
0	ntdll.dll!NtReadFile+0x14
1	KernelBase.dll!ReadFile+0x73
2	ucrtbase.dll!read+0x209
3	ucrtbase.dll!read+0xa2
4	ucrtbase.dll!fread_nolock_s+0x210
5	ucrtbase.dll!fgetc+0xce
6	Beacon23.exe!time_waste_func+0x16
7	kernel32.dll!BaseThreadInitThunk+0x14
8	ntdll.dll!RtlUserThreadStart+0x21

Created Thread calling `getchar`

Call Stack Analysis – Invalid Unwind

- Hide the real caller correctly (i.e. Forging a ‘valid’ call stack)
- Non Unwindable stack is suspicious and may trigger further inspection



#	Name	Stack address	Return address	Frame address
0	ntoskrnl.exe!KiDeliverApc+0x1b0			
1	ntoskrnl.exe!KiSwapThread+0x827			
2	ntoskrnl.exe!KiCommitThreadWait+0x14f			
3	ntoskrnl.exe!KeDelayExecutionThread+0x122			
4	ntoskrnl.exe!NtDelayExecution+0x5f			
5	ntoskrnl.exe!KiSystemServiceCopyEnd+0x25			
6	ntdll.dll!NtDelayExecution+0x14	0x50174ff728	0x7ffeb65795be	0x50174ff720
7	KernelBase.dll!SleepEx+0x9e	0x50174ff730	0x7ff7abc92e0d	0x50174ff7c0
8	ThreadStackSpoofer.exe!MySleep+0x6d	0x50174ff7d0	0x7ffeb7f74b60	0x50174ff8e0
9	kernel32.dll!CreateFileW	0x50174ff8f0	0x7ffeb7f74b60	0x50174ff8e8
10	kernel32.dll!CreateFileW	0x50174ff8f8	0x7ffeb7f74b60	0x50174ff8f0
11	kernel32.dll!CreateFileW	0x50174ff900	0x7ffeb7f74b60	0x50174ff8f8
12	kernel32.dll!CreateFileW	0x50174ff908	0x7ffeb7f74b60	0x50174ff900
13	kernel32.dll!CreateFileW	0x50174ff910	0x7ffeb7f74b60	0x50174ff908
14	kernel32.dll!CreateFileW	0x50174ff918	0x7ffeb7f74b60	0x50174ff910
15	kernel32.dll!CreateFileW	0x50174ff920	0x7ffeb7f74b60	0x50174ff918
16	kernel32.dll!CreateFileW	0x50174ff928		0x50174ff920

Call Stack Analysis - Invalid Unwind

- Hide the real caller correctly (i.e. Forging a 'valid' call stack)
- Non Unwindable stack is suspicious and may trigger further inspection

Call Stack Analysis – Direct Syscalls

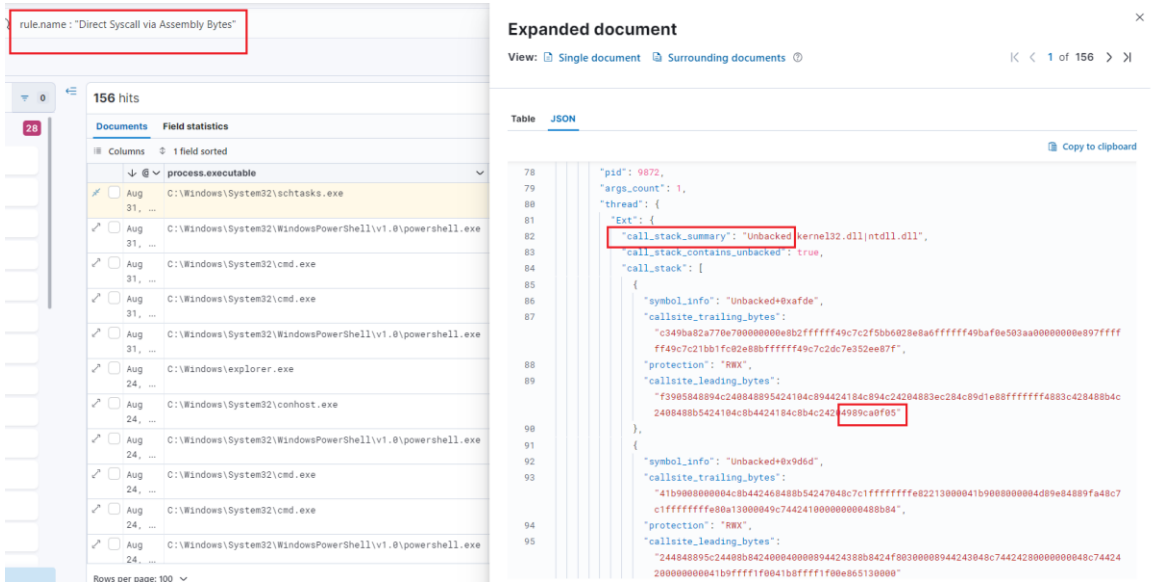
elastic.co/security-labs/peeling-back-the-curtain-with-call-stacks

stic security labs

Direct Syscall via Assembly Bytes

The second and final example for process events is process creation via direct syscall. This directly uses the syscall instruction instead of calling the **NtCreateProcess** API. Adversaries may use **this method** to avoid security products that are reliant on usermode API hooking (which Elastic Defend is not):

```
process where event.action : "start" and

// EQL detecting a call stack not ending with ntdll.dll
not process.parent.thread.Ext.call_stack_summary : "ntdll.dll*" and

/* last call in the call stack contains bytes that execute a syscall
 manually using assembly <mov r10,rcx, mov eax,ssn, syscall> */

_arraysearch(process.parent.thread.Ext.call_stack, $entry,
 ($entry.callsite_leading_bytes : ("*4c8bd1b8??????000f05",
 "*4989cab8??????000f05", "*4c8bd10f05", "*4989ca0f05")))
```



Reference:
https://www.elastic.co/security-labs/peeling-back-the-curtain-with-call-stacks

Call Stack Analysis – Injected Code

Stack - thread 6548

	Name
0	win32u.dll!NtUserWaitMessage+0x14
1	user32.dll!DialogBoxIndirectParamAorW+0x431
2	user32.dll!DialogBoxIndirectParamAorW+0x1a1
3	user32.dll!SoftModalMessageBox+0x85b
4	user32.dll!DrawStateA+0x1e25
5	user32.dll!MessageBoxTimeoutW+0x1a7
6	user32.dll!MessageBoxTimeoutA+0x108
7	user32.dll!MessageBoxA+0x4e
8	0x27624080100
9	0x27624080127
10	0x27624080142

Injected Shellcode calling `MessageBoxA`

Stack - thread 11428

	Name
0	win32u.dll!NtUserWaitMessage+0x14
1	user32.dll!DialogBoxIndirectParamAorW+0x431
2	user32.dll!DialogBoxIndirectParamAorW+0x1a1
3	user32.dll!SoftModalMessageBox+0x85b
4	user32.dll!DrawStateA+0x1e25
5	user32.dll!MessageBoxTimeoutW+0x1a7
6	user32.dll!MessageBoxTimeoutA+0x108
7	user32.dll!MessageBoxA+0x4e
8	Beacon23.exe!time_waste_func+0x1d
9	kernel32.dll!BaseThreadInitThunk+0x14
10	ntdll.dll!RtlUserThreadStart+0x21

Legit call to `MessageBoxA`

Call Stack Spoofing

Image src:
https://www.archdaily.com/942109/mirrors-in-architecture-possibilities-of-reflected-space/5ef05946b3576529f5000120-mirrors-in-architecture-possibilities-of-reflected-space-photo

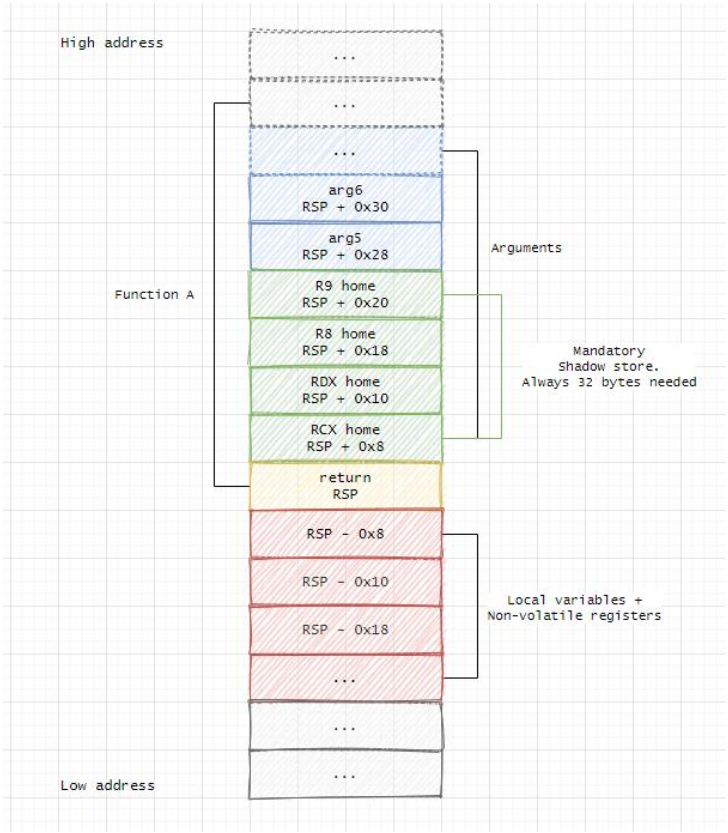
2 Main reasons to Spoof the Call Stack:

- Conceal origin of the call
- Make the call look legit

Windows x64 calling convention

- The first four arguments to a function are passed in registers.
- Remaining arguments get pushed on the stack in right-to-left order.

First	Second	Third	Fourth	Fifth and higher
RCX	RDX	R8	R9	stack



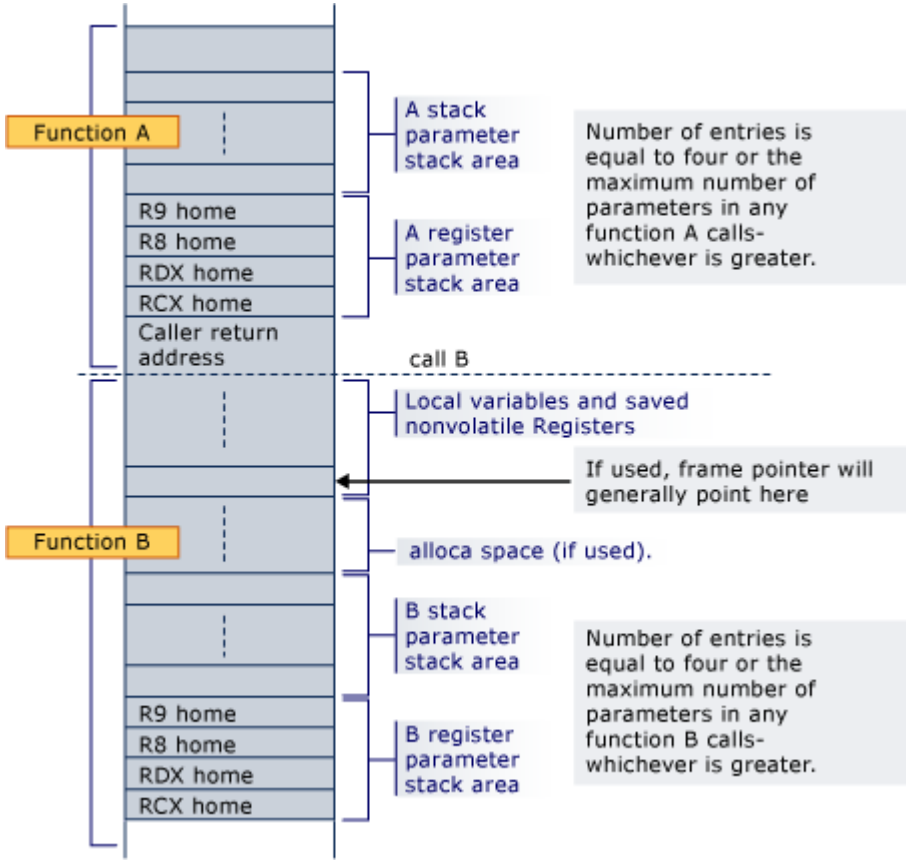
Source: https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/windows-x64-calling-convention-stack-frame

MSDN Documentation: https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170

Windows x64 calling convention

- Usually stack operations happen only in function prologue and epilogue.
  - Standardization allows to unwind the stack without using a frame pointer

- If the function body changes the stack pointer, the frame pointer must be used to save the base of the stack
  - RBP = RSP
  - Frame pointer = RBP



https://learn.microsoft.com/en-us/cpp/build/stack-usage?view=msvc-170

Windows x64 calling convention

- Volatile registers – must be **caller preserved**
- Nonvolatile registers – must be **callee preserved**

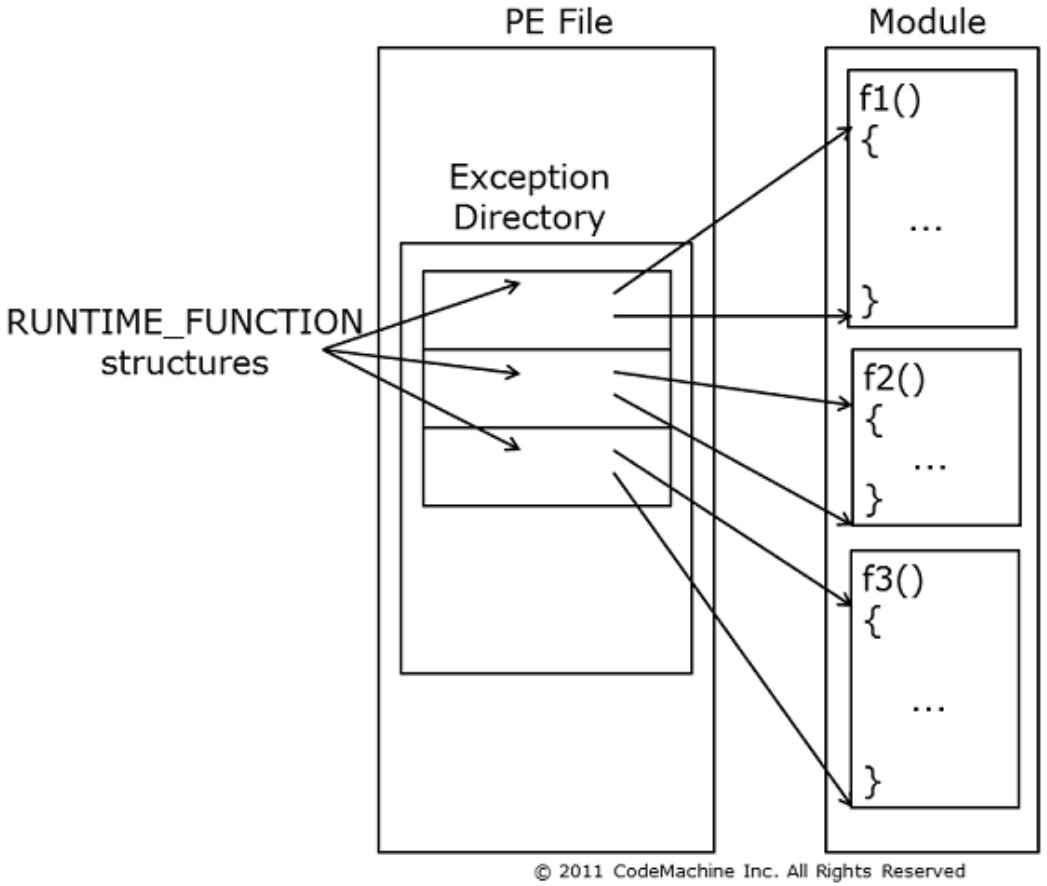
The following table describes how each register is used across function calls:

Register	Status	Use
RAX	Volatile	Return value register
RCX	Volatile	First integer argument
RDX	Volatile	Second integer argument
R8	Volatile	Third integer argument
R9	Volatile	Fourth integer argument
R10:R11	Volatile	Must be preserved as needed by caller; used in syscall/sysret instructions
R12:R15	Nonvolatile	Must be preserved by callee
RDI	Nonvolatile	Must be preserved by callee
RSI	Nonvolatile	Must be preserved by callee
RBX	Nonvolatile	Must be preserved by callee
RBP	Nonvolatile	May be used as a frame pointer; must be preserved by callee
RSP	Nonvolatile	Stack pointer

https://learn.microsoft.com/en-us/cpp/build/x64-software-conventions?view=msvc-170

Windows x64 Exceptions

- All stack operations performed by non-leaf functions (i.e. functions that calls other functions) are saved at compile time inside a PE section (`.pdata`) called Exception Directory
- PE Exception Directory contains `RUNTIME_FUNCTION` structs.
- When an exception is thrown, the system can unwind the stack using the information in the Exception Directory

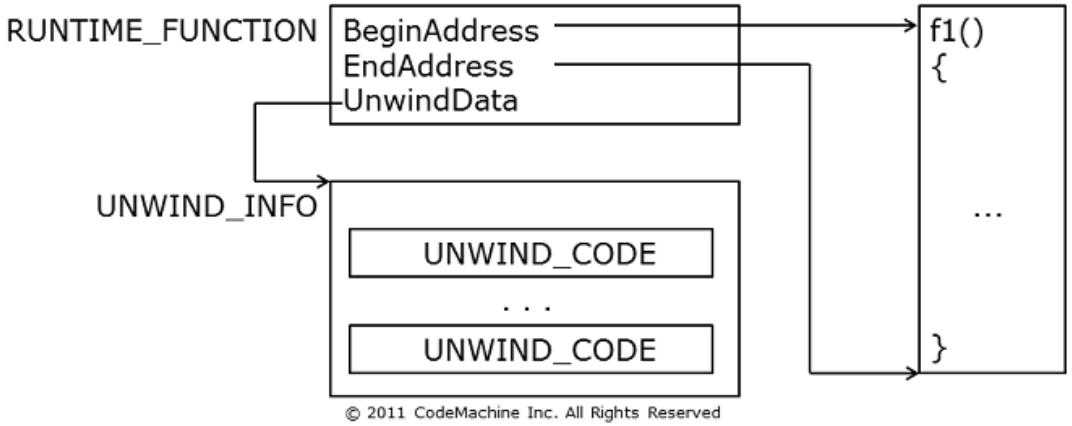


More details:
https://codemachine.com/articles/x64_deep_dive.html

Stack Unwinding 101

- `RUNTIME_FUNCTION` contains
  - References to function address (offsets from module base)
  - Reference to `UNWIND_INFO` stack containing stack operations

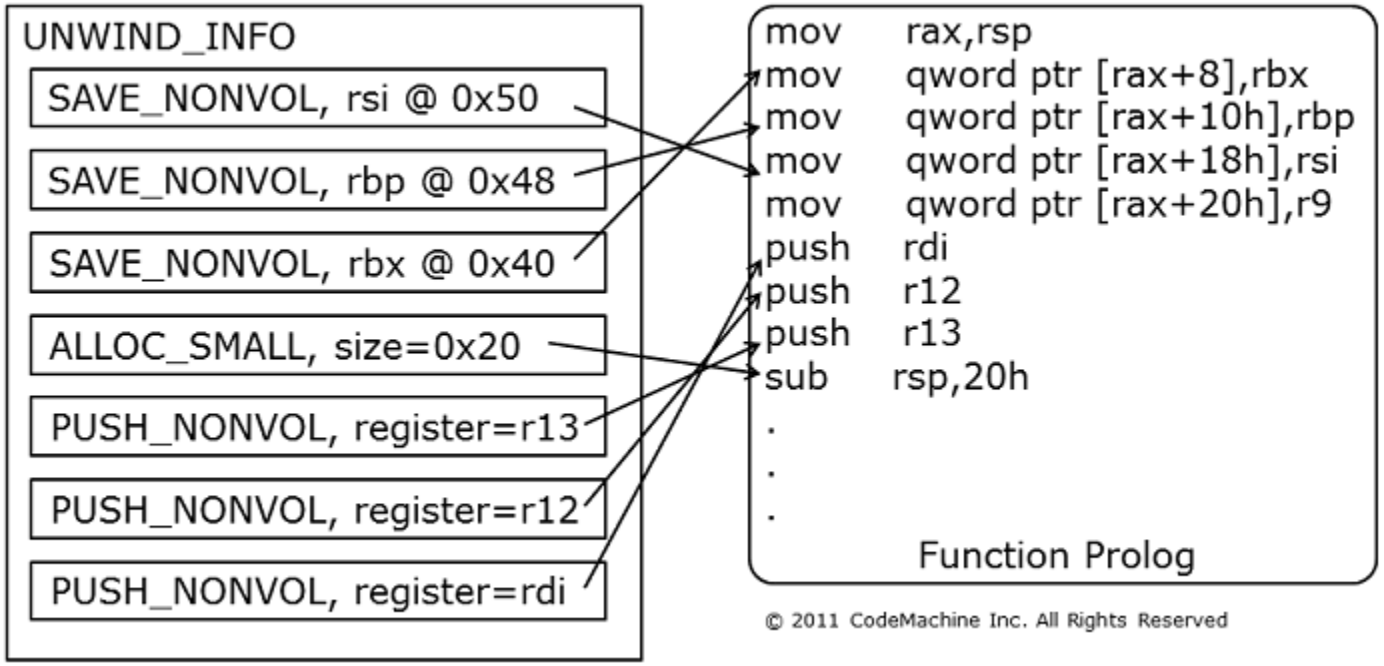
- `UNWIND_INFO` structure contains `UNWIND_CODE` structures, each one of which reverses the effect of a single stack related operation



More details:
https://codemachine.com/articles/x64_deep_dive.html

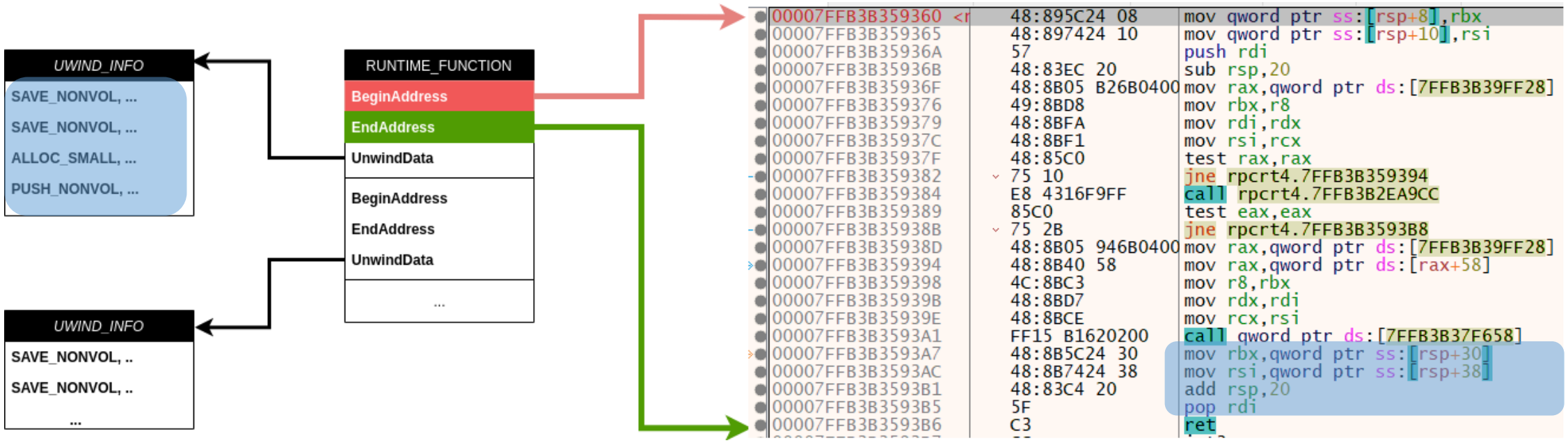
Stack Unwinding 101

- `SAVE_NONVOL` - Save a non-volatile register on the stack.
- `PUSH_NONVOL` - Push a non-volatile register on the stack.
- `ALLOC_SMALL` - Allocate space (up to 128 bytes) on the stack.
- `ALLOC_LARGE` - Allocate space (up to 4GB) on the stack.



More details:
https://codemachine.com/articles/x64_deep_dive.html

Stack Unwinding 101



It’s all about perception…

- Retrieving the call stack is an indrect operation
- Interpration of the stack content relies on a data structure that contains the requied information
- The call stack observed by the unwinding algorithm is an interpretation of the stack content under the guide of the Exception Table
  - starting from the return address at the top of the stack (i.e. return address of the function that is running), find the next return address using the Exception Table

It’s all about perception…

- Retrieving the call stack is an indrect operation
- Interpration of the stack content relies on a data structure that contains the requied information
- The call stack observed by the unwinding algorithm is an interpretation of the stack content under the guide of the Exception Table
  - starting from the return address at the top of the stack (i.e. return address of the function that is running), find the next return address using the Exception Table

Having an address written at a certain place in the stack (i.e. the return address that points to the caller function) does not necessarily imply that a given operation has been executed in the stack

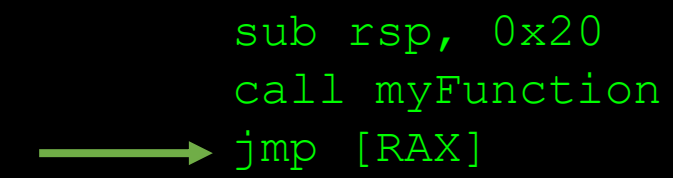
It’s all about perception...

```
sub rsp, 0x20
call myFunction
jmp [RAX]
```

This code does the following operations

- 0x20 bytes are allocated in the stack
- `myFunction` is called
  - The address of the next instruction (`address_of(jmp [RAX])`) gets pushed in the stack. This address is `RIP + 8`
  - `RIP` is set to the address of `myFunction` and the function gets executed
- when `myFunction` returns, it will pop the return address in RIP (`pop RIP` -> `RIP = address_of(jmp [RAX])`) and the `jmp` instruction will be executed

It’s all about perception… What If..



If the stack of our thread contains a return address pointing to `jmp [RAX]`, the unwinding algorithm will assume that our caller function was the snippet above.

It will also assume that we have allocated 0x20 bytes in the stack AND it will interpret the content of the stack accordingly.

Luckily enough, we can write whatever we want in our stack

How Can We Abuse This? - Synth

Craft synthetic (i.e. artifical) stack inside the spoofer function stack

- Write the 'real' return address (address of the real control flow) inside a Nonvolatile register (e.g. `RBX`)
  - address of a restore function that will restore the registers and the stack pointer to the state before the spoofed call happened
- Allocate memory in our stack to store the fake stack
- Place specially crafted stack frames 'pointing' to the functions we want to add to our spoofed call stack
- Setup a frame containing a ROP gadget that will restore the real execution by jumping to the address stored in the Nonvolatile register picked in the first step (e.g. jmp [RBX]).
- Setup a stack pivot frame to conceal the caller of the function

How Can We Abuse This? - Desync

- Write the 'real' return address (address of the real control flow) inside a Nonvolatile register (e.g. `RBX`)
  - address of a restore function that will restore the registers and the stack pointer to the state before the spoofed call happened

- Find desync frame that contains a jump targeting the Nonvolatile register picked in the previous step (e.g. `jmp [RBX]`)
  - used to restore the original execution flow when the called function returns

- Find frames that involve usage of stack frame, to allow `RSP` manipulation
  - allow to put an arbitrary pointer in `RSP` when the stack is inspected by the unwinding algorithm (more on this later…)

- Find a stack pivot gadget that will move the stack pointer
  - this frame will conceal the source address of the call

How Can We Abuse This? - Desync

- Write the 'real' return address (address of the real control flow) inside a Nonvolatile register (e.g. `RBX`)
  - address of a restore function that will restore the registers and the stack pointer to the state before the spoofed call happened

- Find desync frame that contains a jump targeting the Nonvolatile register picked in the previous step (e.g. `jmp [RBX]`)
  - used to restore the original execution flow when the called function returns

- **Find frames that involve usage of stack frame, to allow `RSP` manipulation**
  - **allow to put an arbitrary pointer in `RSP` when the stack is inspected by the unwinding algorithm (more on this later…)**

- Find a stack pivot gadget that will move the stack pointer
  - this frame will conceal the source address of the call

How Can We Abuse This? - Desync

- **Find frames that involve usage of stack frame, to allow `RSP` manipulation**
  - **allow to put an arbitrary pointer in `RSP` when the stack is inspected by the unwinding algorithm (more on this later…)**

If we write the address we want to spoof 'at the right place' inside the stack, the unwinding algorithm will use that value to `POP RBP` and consequently, RSP will be set to that value.
We can now control which is the parent frame, which can be BaseThreadInitThunk ;)

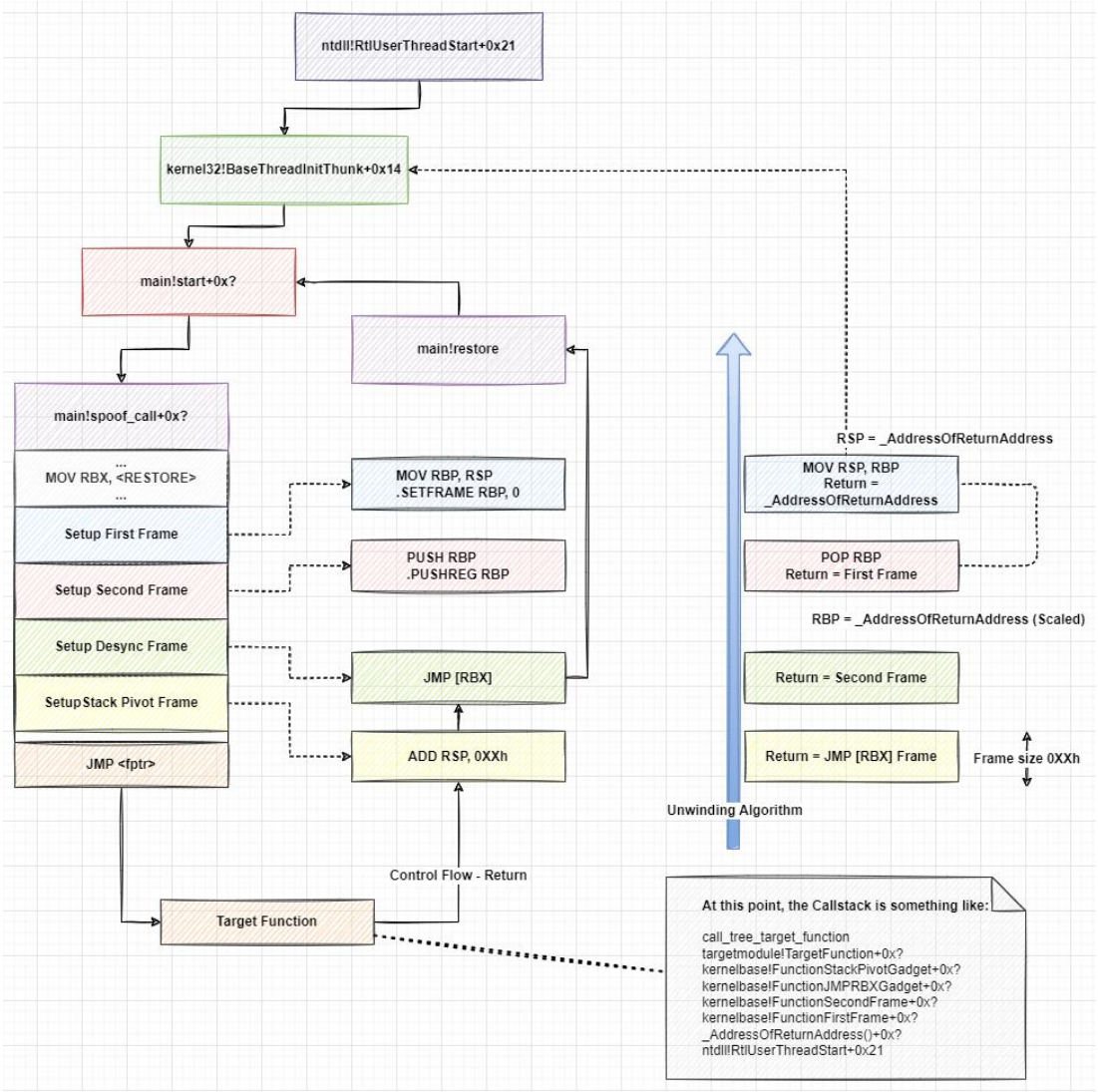
Desync Spoofing

1. Frame that performs `UWOP_SET_FPREG` operation (i.e. set frame pointer to RBP -> RBP = RSP + X)

2. Frame that pushes RBP to the stack
   - combined with the first frame allows to put an arbitrary pointer on the stack, which will be used as the simulated Stack Pointer by the unwinding algorithm.

3. Desync frame which contains a ROP gadget that will execute `JMP [RBX]` to jump back to the original control flow

4. Frame that contains a stack pivot gadget to conceal the original RIP

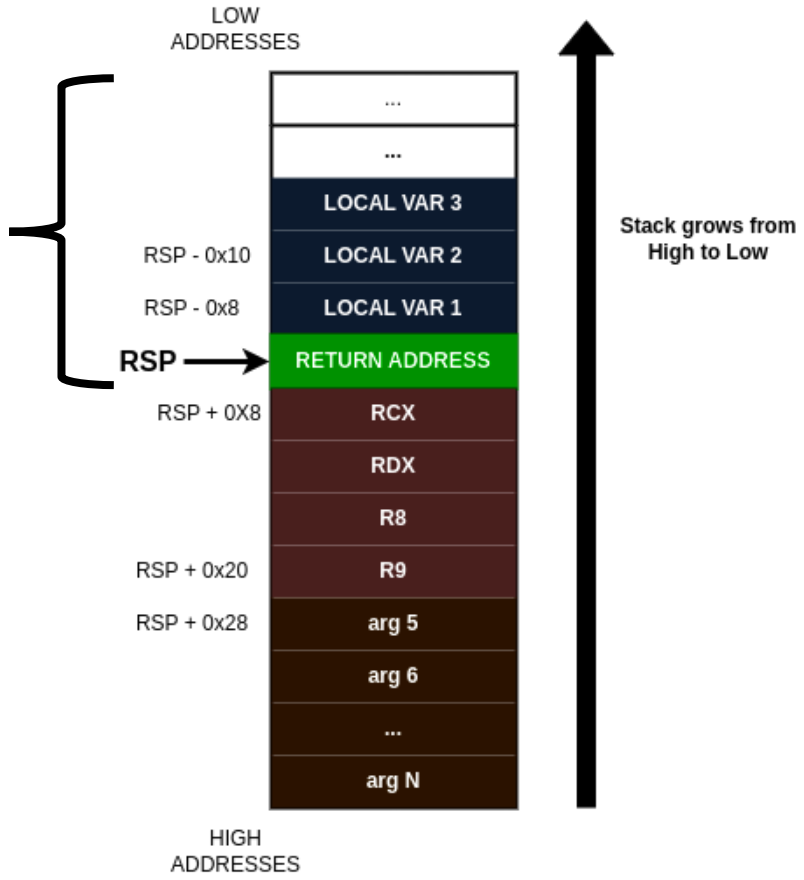


Source: https://klezvirus.github.io/RedTeaming/AV_Evasion/StackSpoofing/

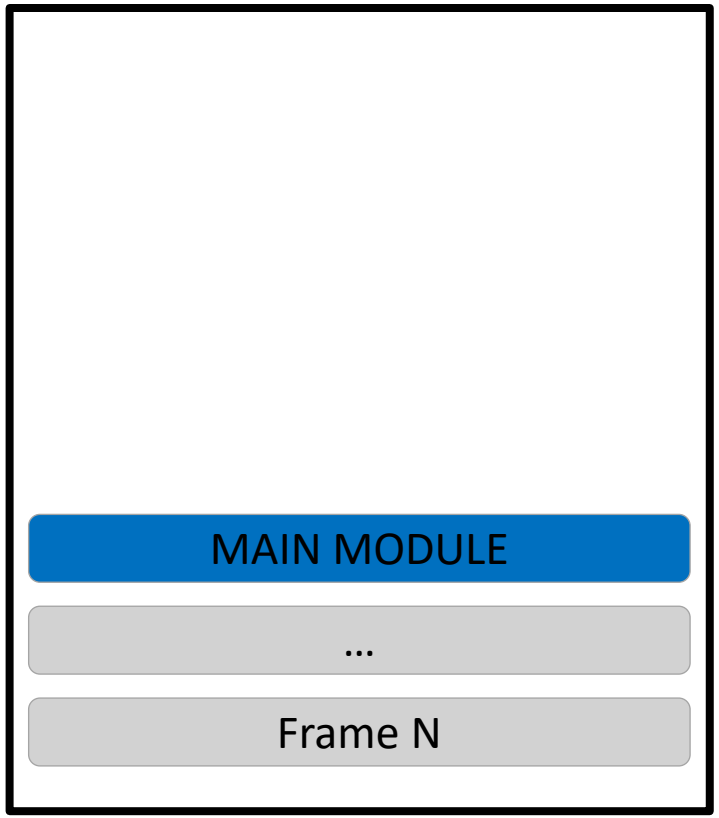
Construct the Spoofed Stack

The concpet of creating a stack frame can be summarized with the following steps

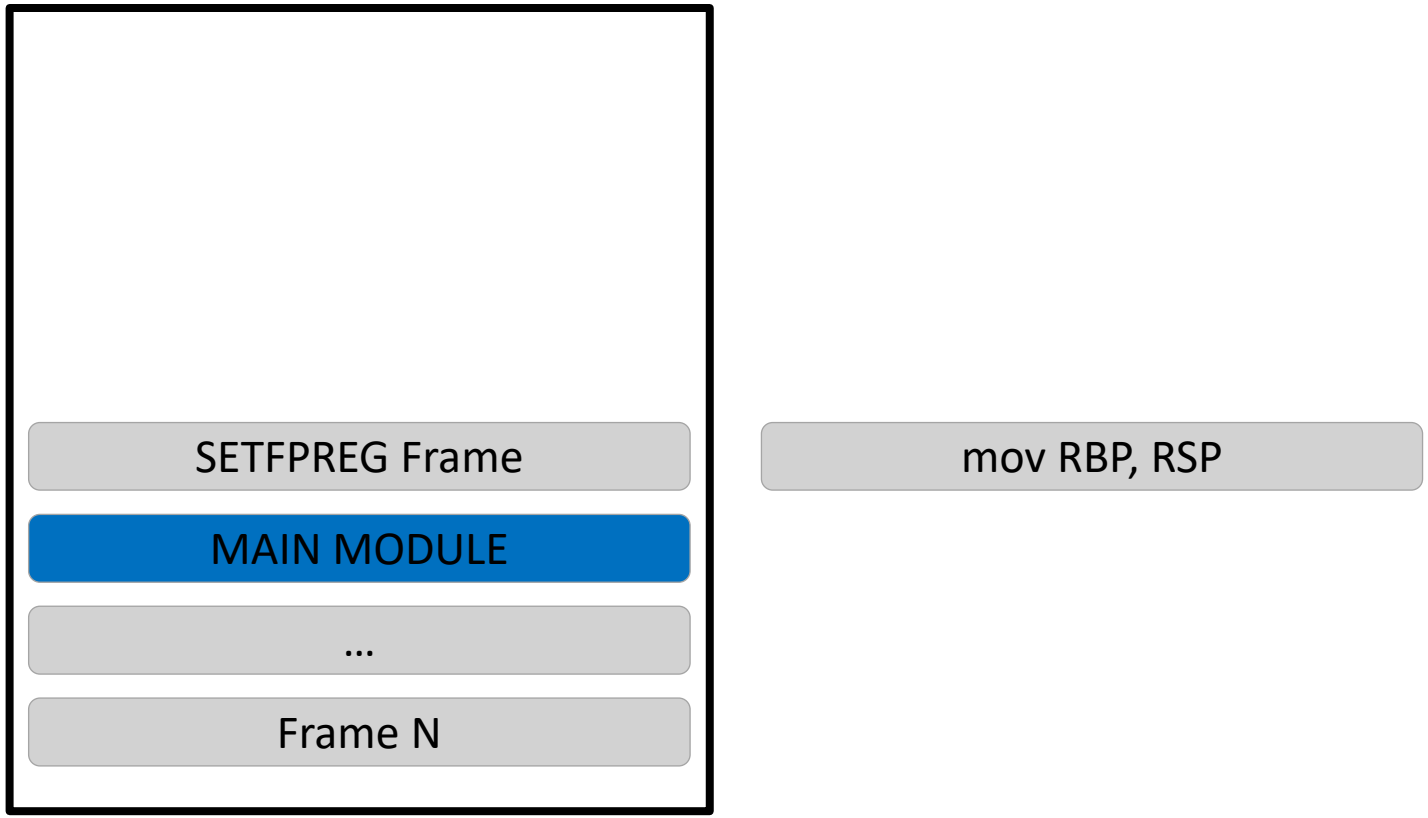
- Push parameters
- Push return address
- Allocate stack space (i.e. `sub rsp , X`)



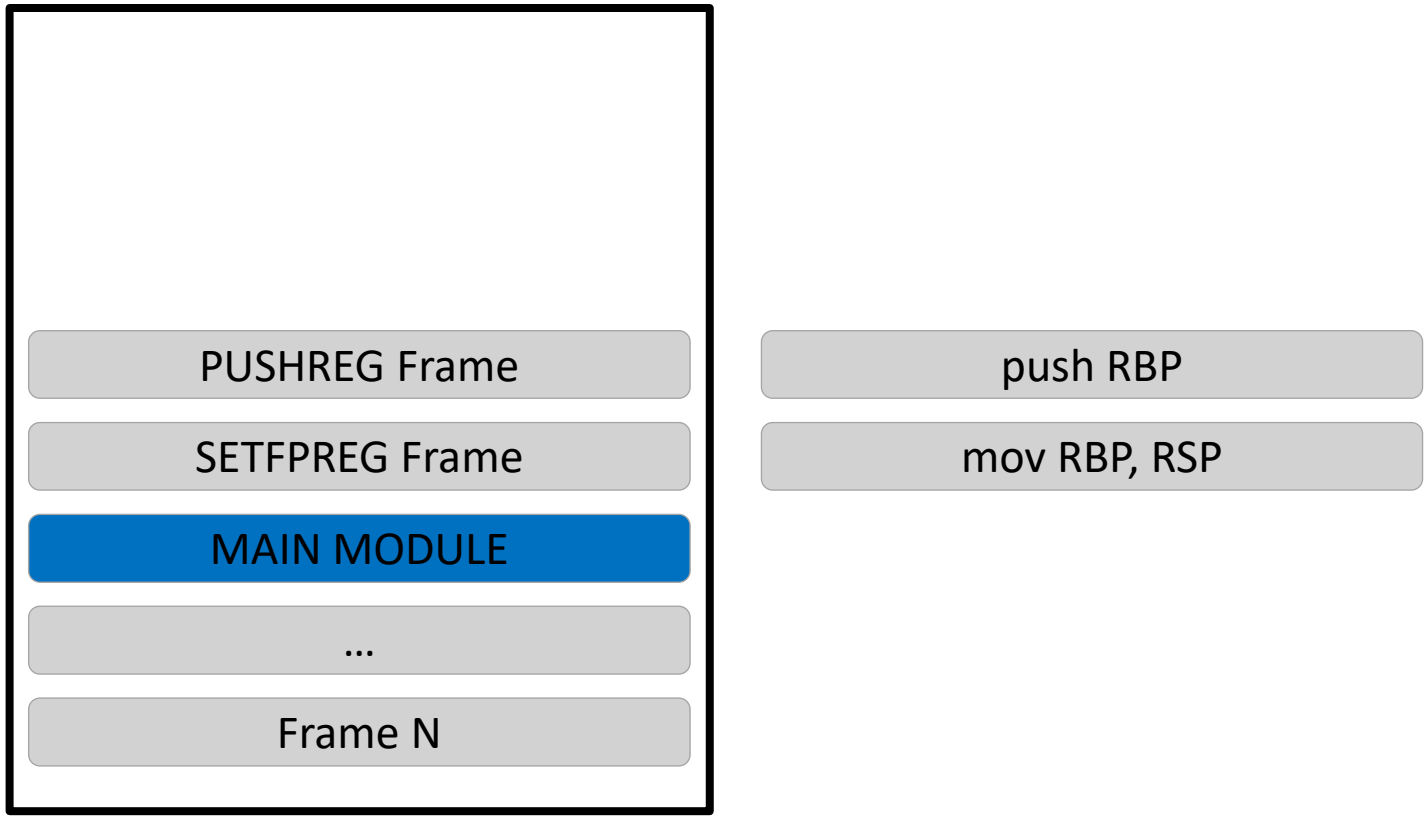
Construct the Spoofed Stack



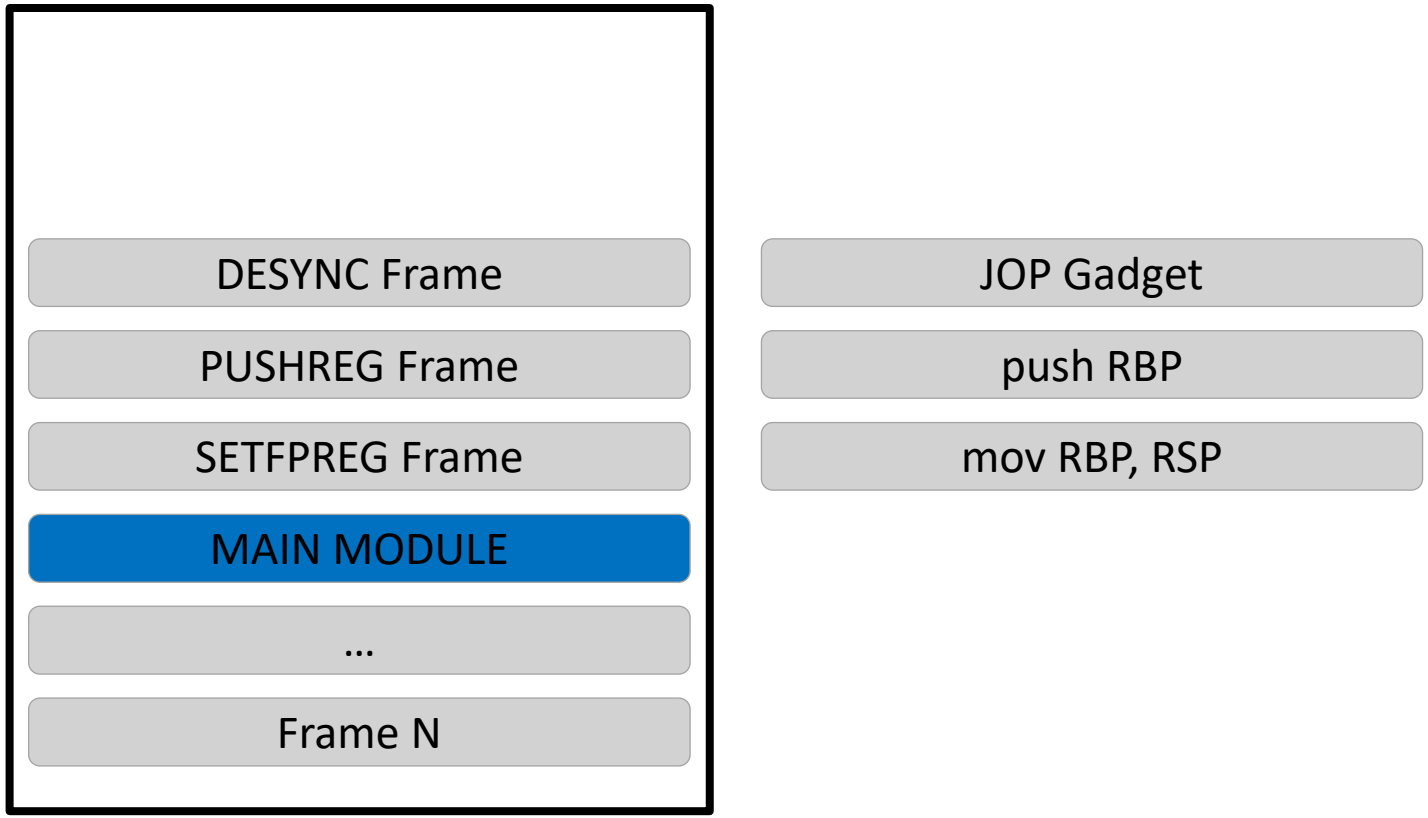
Construct the Spoofed Stack



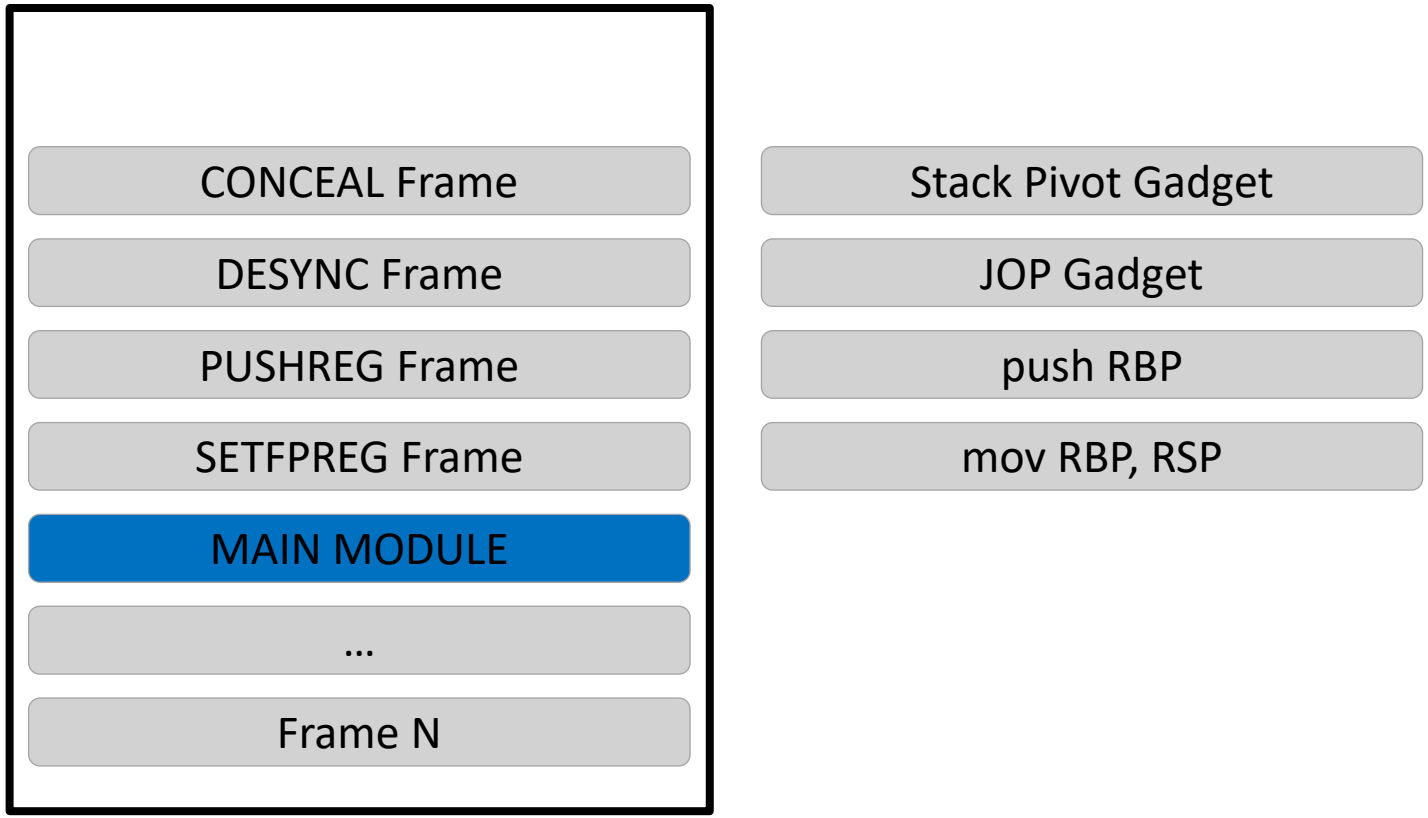
Construct the Spoofed Stack



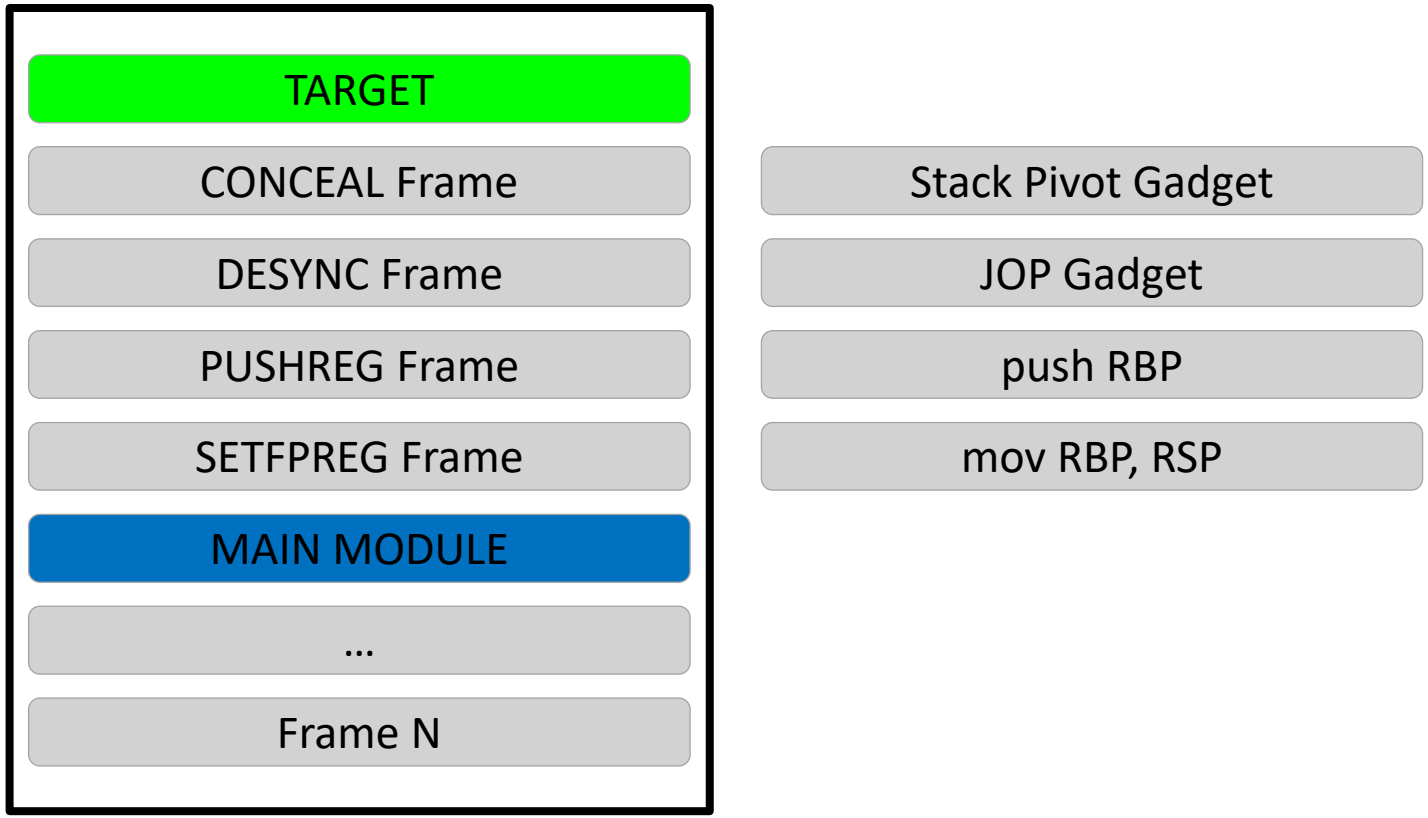
Construct the Spoofed Stack



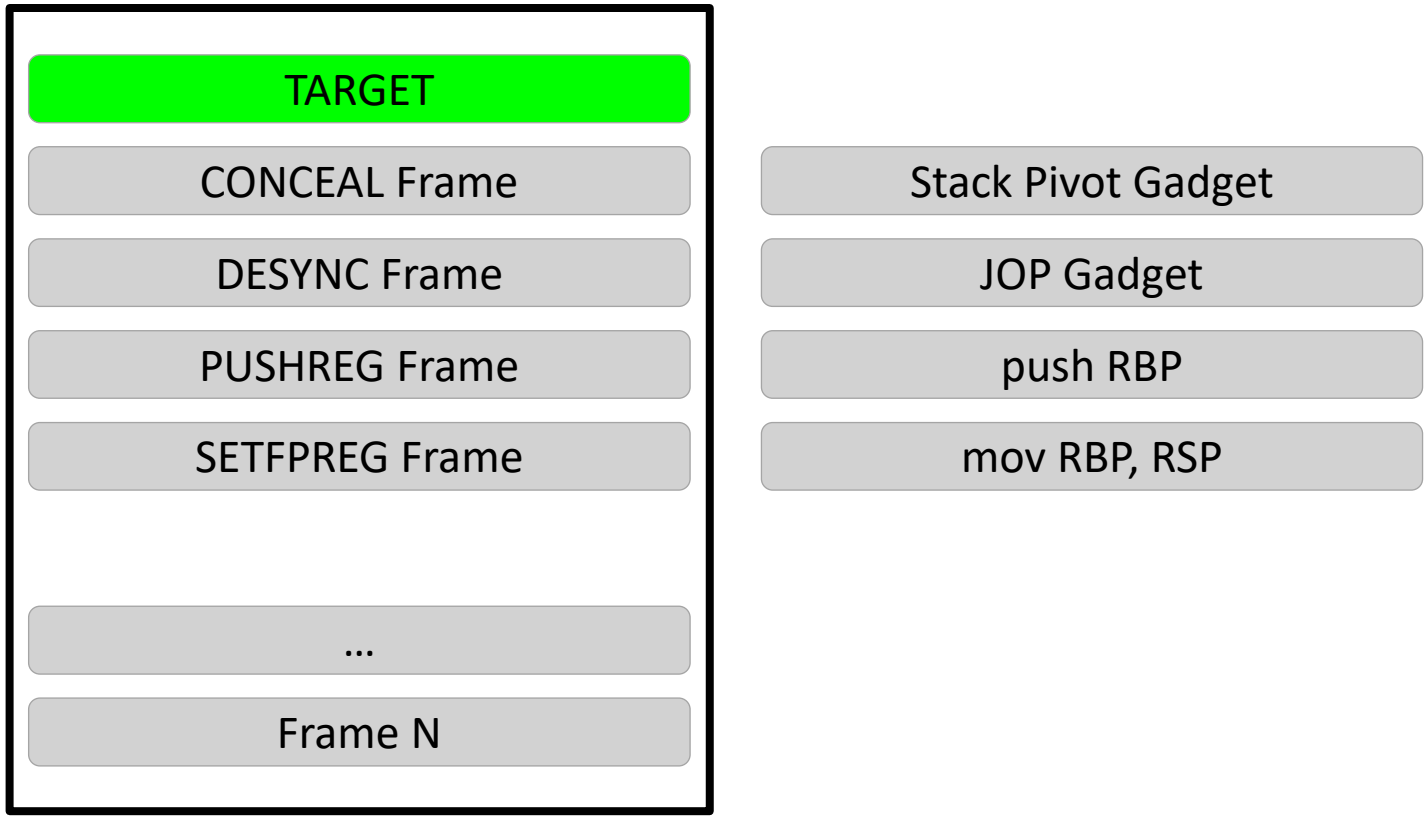
Construct the Spoofed Stack



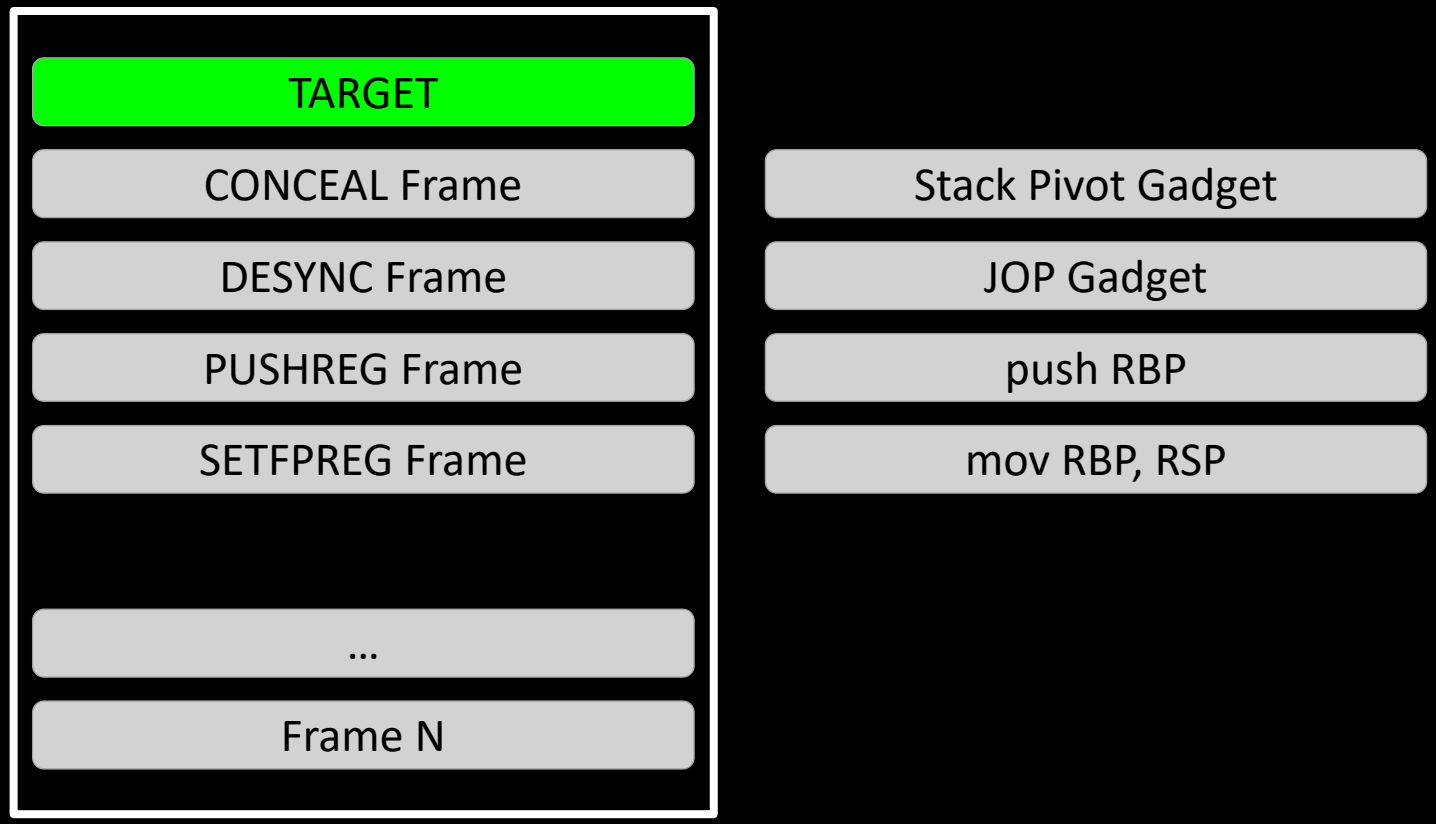
Construct the Spoofed Stack



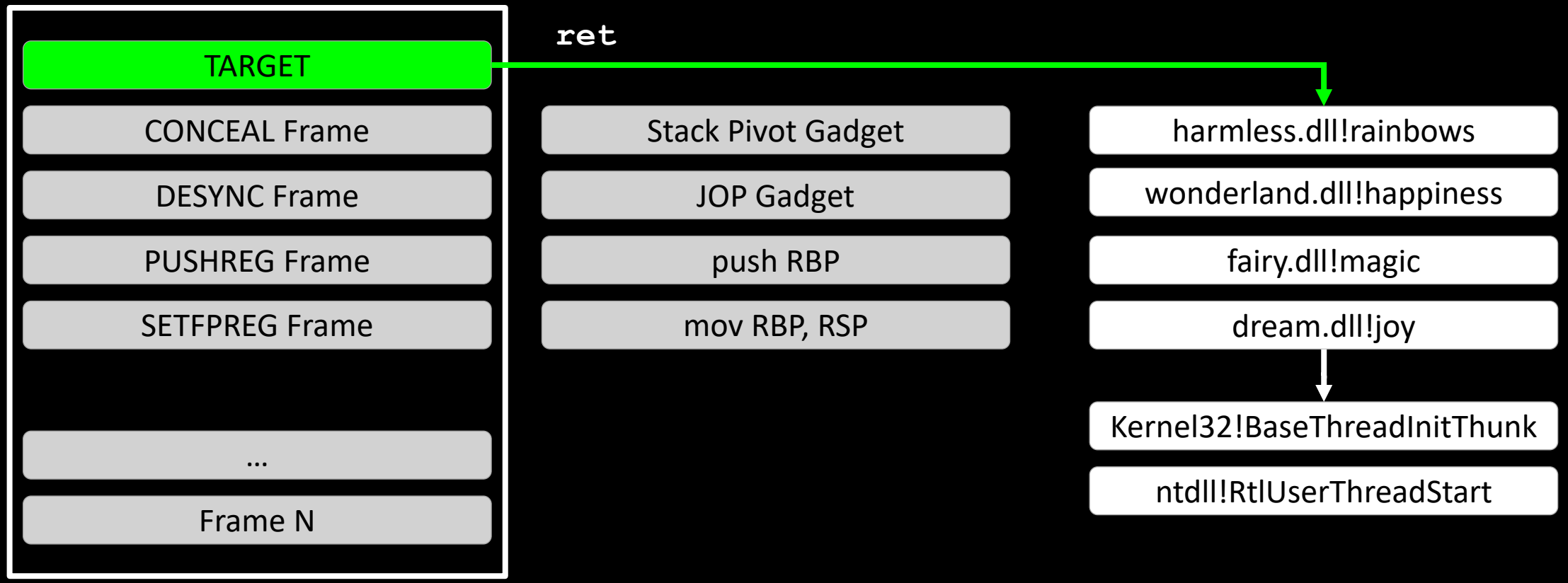
Construct the Spoofed Stack



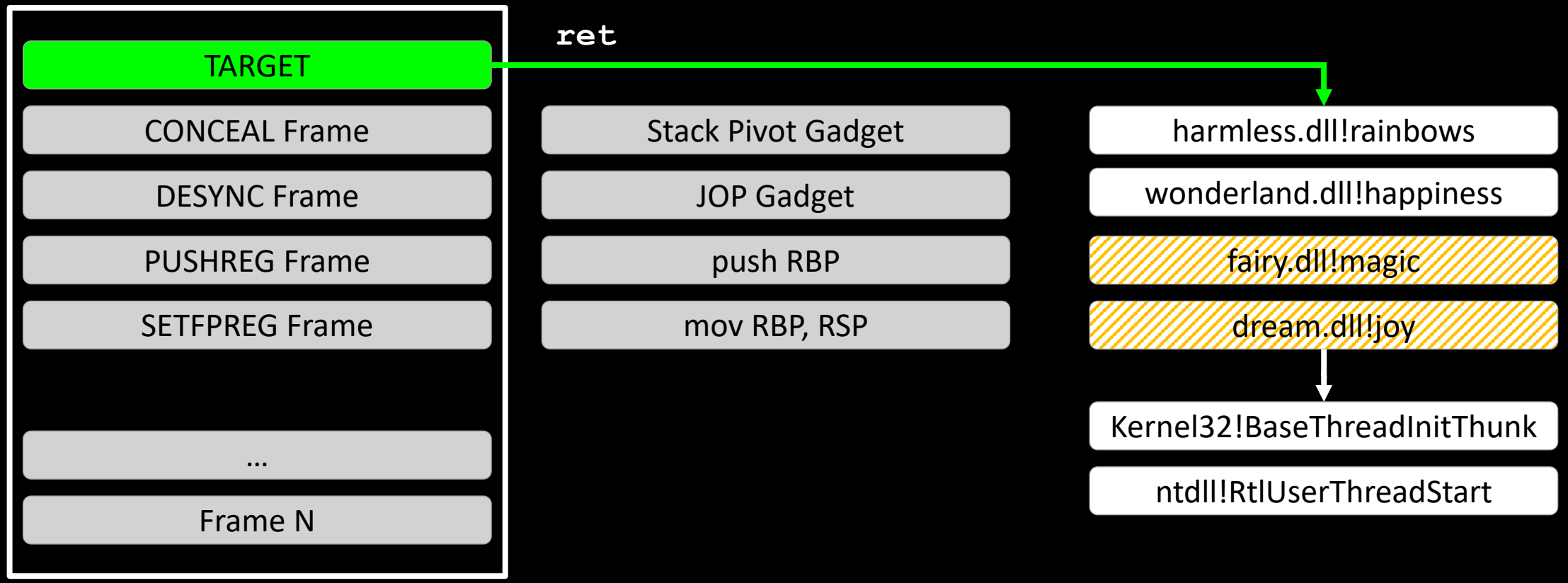
It’s all about perception…



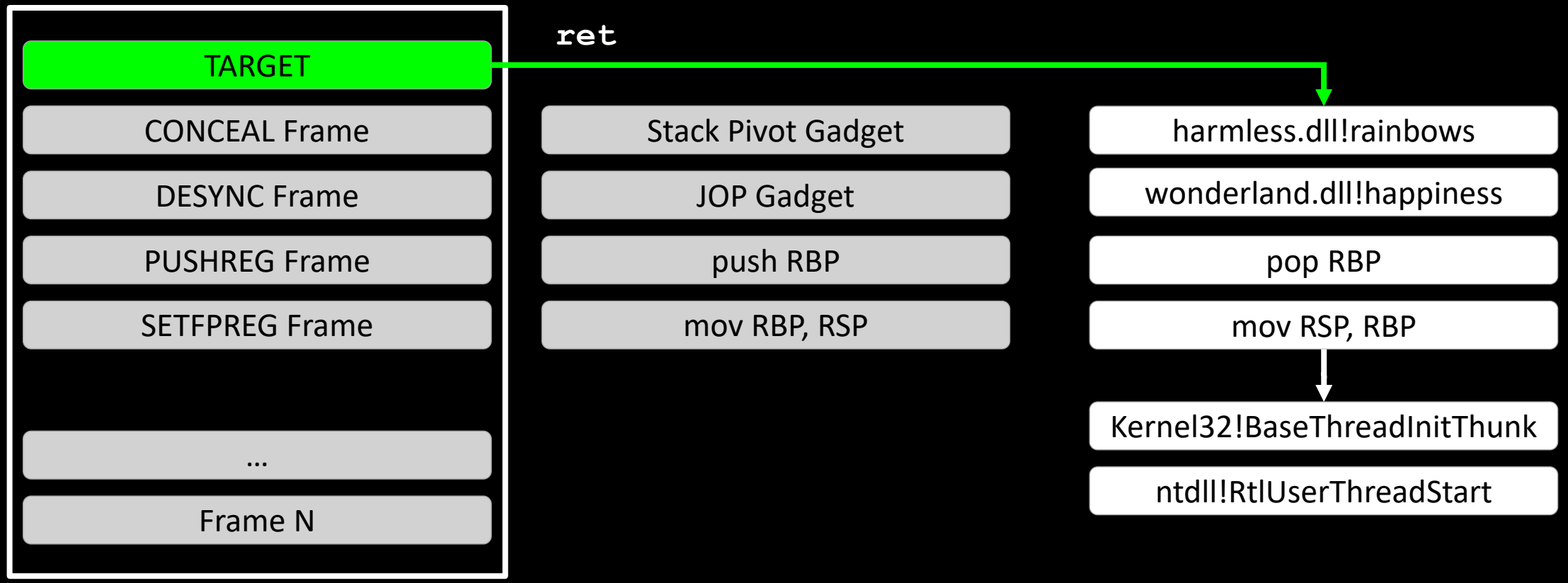
It’s all about perception...



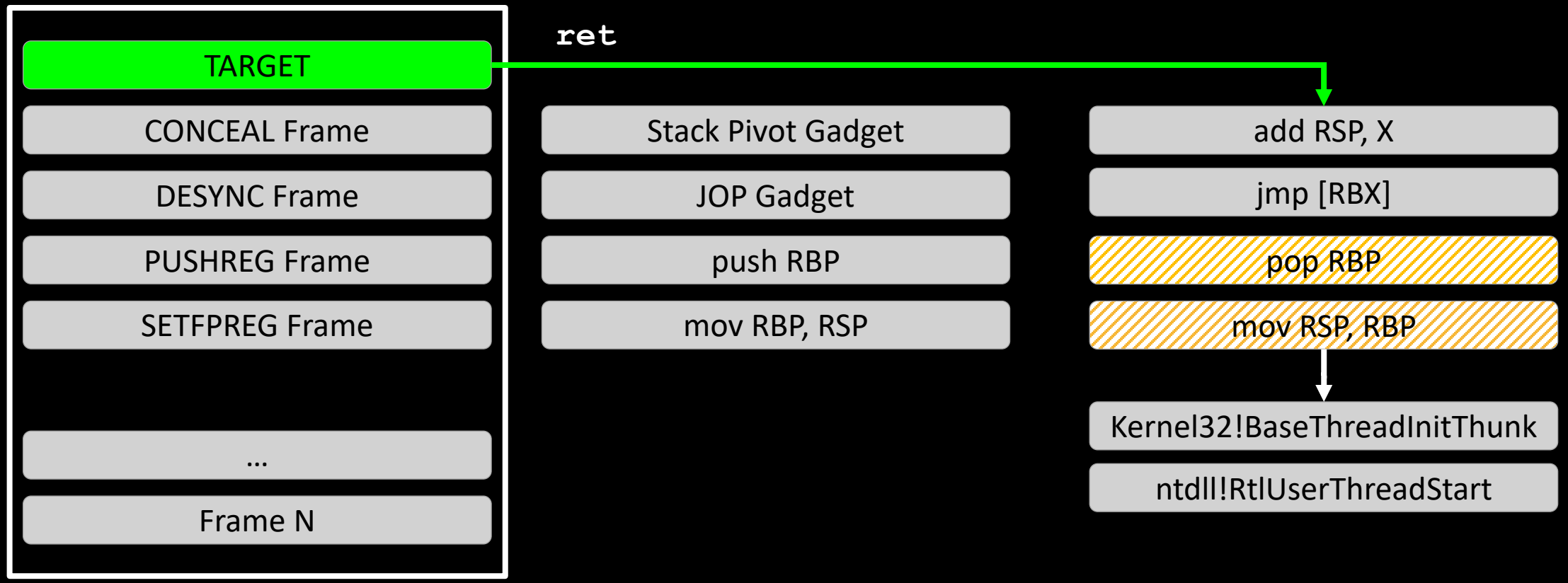
It’s all about perception...



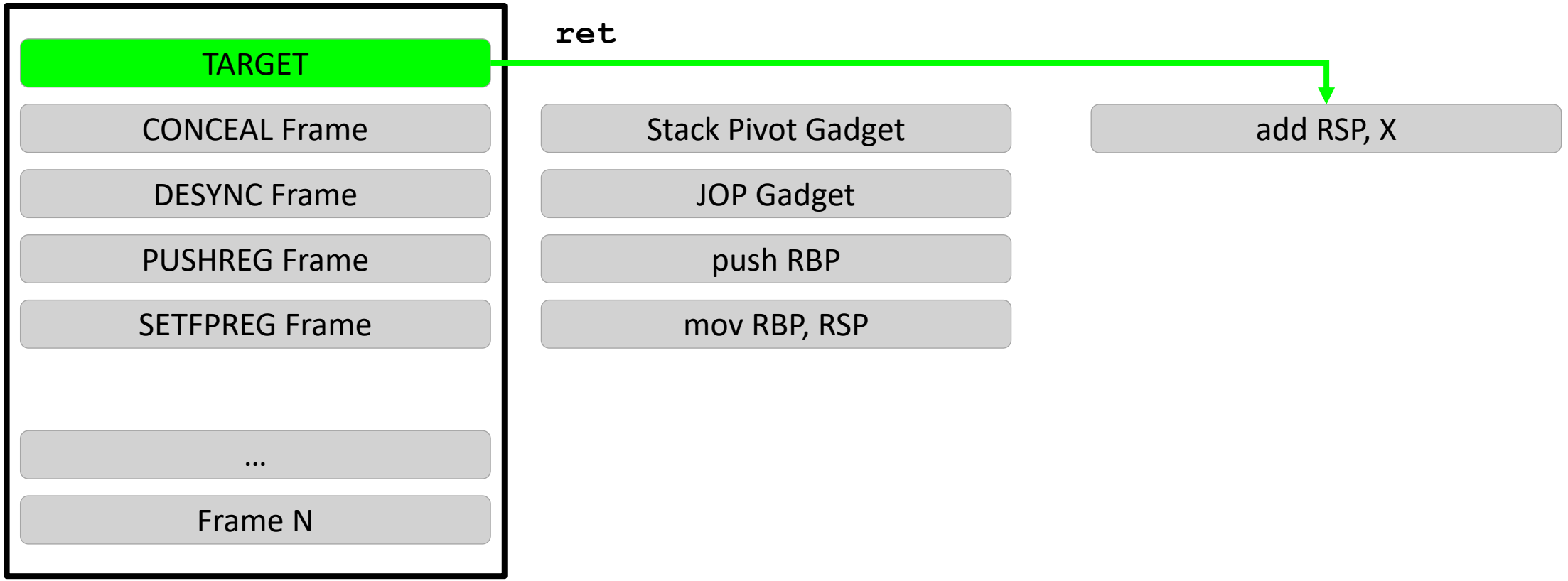
It’s all about perception...



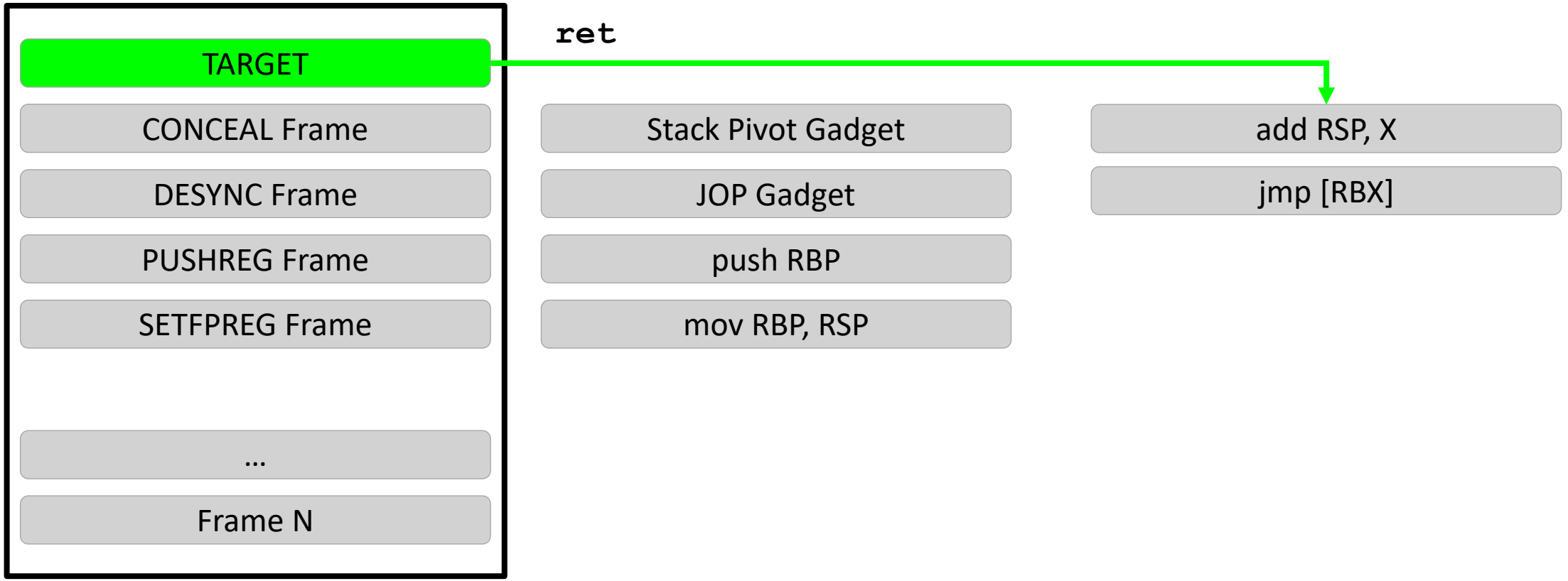
It’s all about perception...



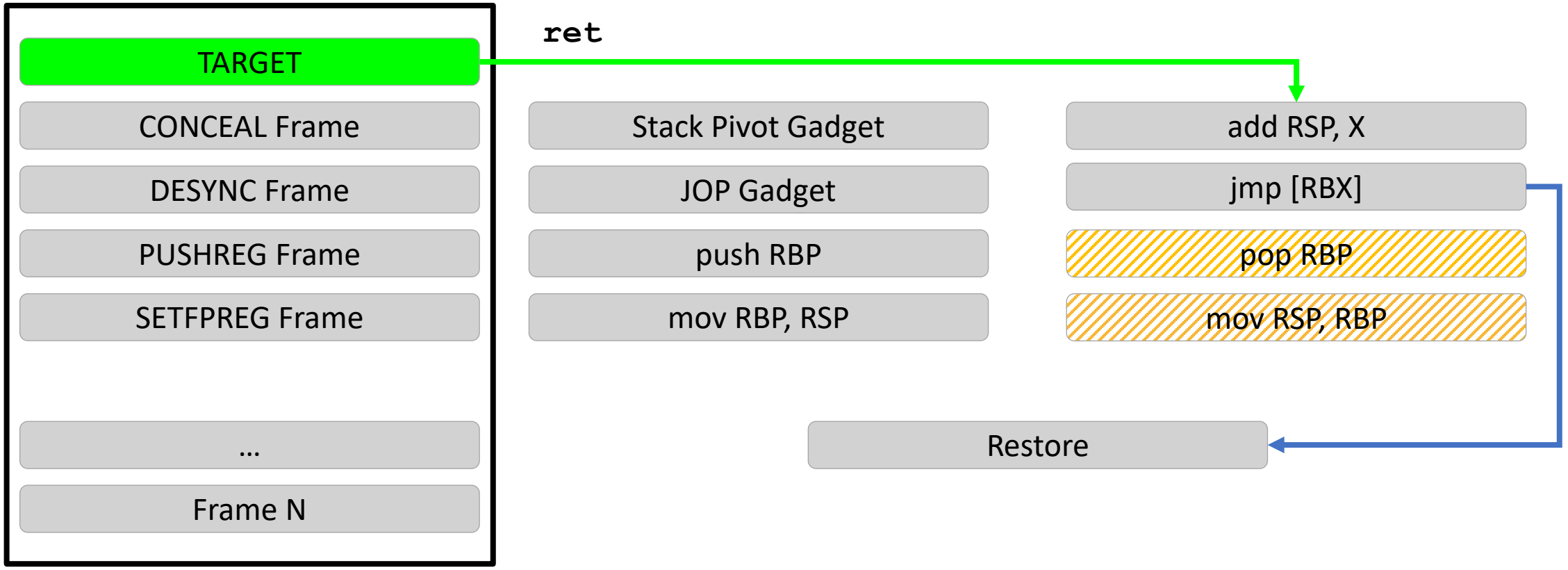
What Actually Happens



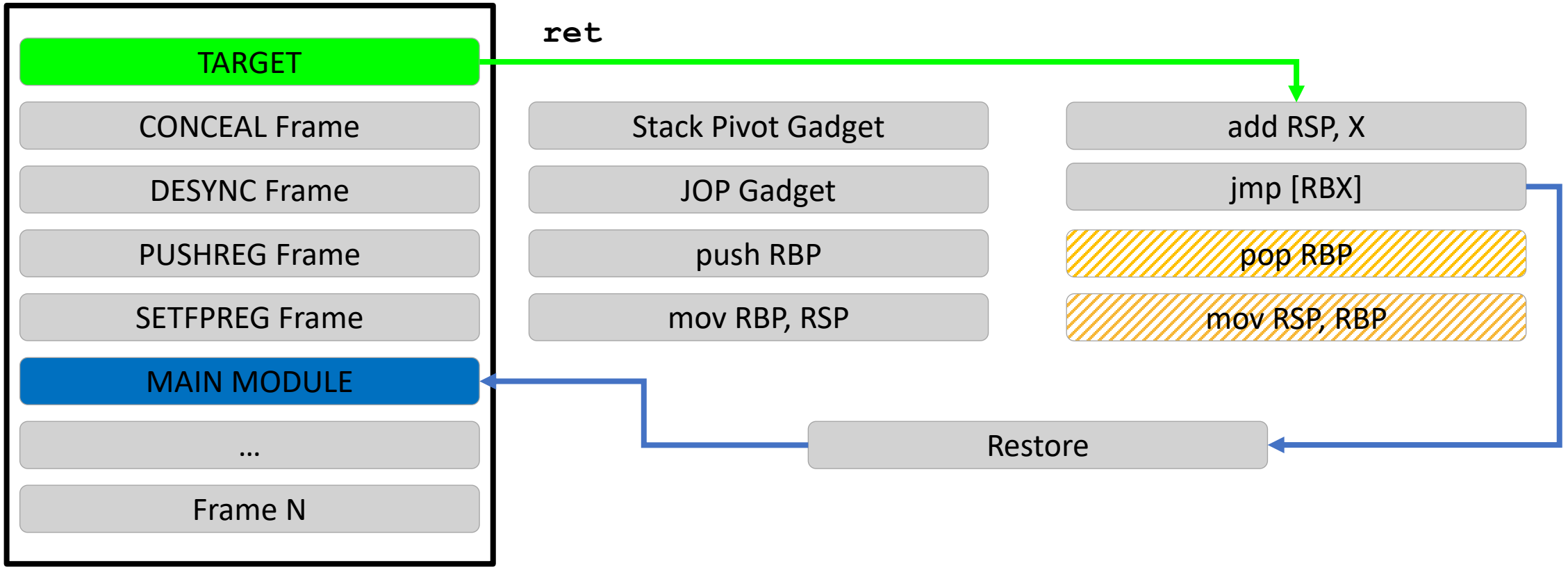
What Actually Happens



What Actually Happens



What Actually Happens



Credits and References

https://codemachine.com/articles/x64_deep_dive.html

@namazso
https://www.unknowncheats.me/forum/anti-cheat-bypass/268039-x64-return-address-spoofing-source-explanation.html

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

https://github.com/JLospinoso/gargoyle

https://github.com/waldo-irc/YouMayPasser

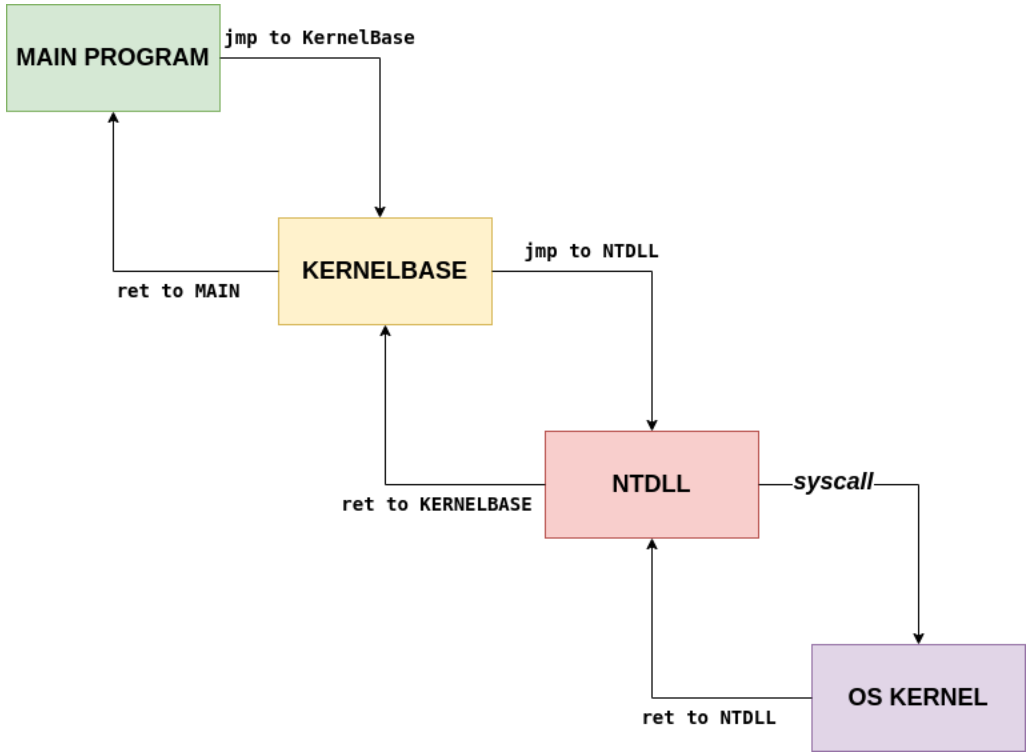
https://github.com/kyleavery/AceLdr

Indirect System Calls with Call Stack Spoofing

- Resolve System Call Number
- Prepare spoofed stack
- Setup parameters
- Setup “syscall” registers

```
mov r10, rcx
mov eax, syscalls_no
```

- Jump to any NTDLL `syscall` instruction



Legit Syscall Call Stack Trace

Stack - thread 10804

	Name
0	ntdll.dll!ZwAccessCheckAndAuditAlarm+0x14
1	Beacon23.exe!very_evil_function+0xf0
2	kernel32.dll!BaseThreadInitThunk+0x14
3	ntdll.dll!RtlUserThreadStart+0x21

Call to NtWaitForSingleObject using **(randomized) indirect system calls**

Stack - thread 10752

	Name
0	ntdll.dll!ZwWaitForSingleObject+0x14
1	KernelBase.dll!WaitForSingleObjectEx+0x8e
2	Beacon23.exe!very_evil_function+0x5e
3	kernel32.dll!BaseThreadInitThunk+0x14
4	ntdll.dll!RtlUserThreadStart+0x21

Call to WaitForSingleObject **API**

Call Stack Trace

Call to Vir

```
nop dword ptr ds:[rax+rax],eax
mov r10,rcx
mov eax,18
test byte ptr ds:[7FFE0308],1
jne ntdll.7FFC774AD3B5
syscall
ret
```

Stack - thread 5396

	Name
0	ntdll.dll!ZwAllocateVirtualMemory+0x12
1	KernelBase.dll!VirtualAlloc+0x48
2	Beacon23.exe!very_evil_function+0x57
3	Beacon23.exe!main+0x20
4	Beacon23.exe!invoke_main+0x39
5	Beacon23.exe!__scrt_common_main_seh+0x12e
6	Beacon23.exe!__scrt_common_main+0xe
7	Beacon23.exe!mainCRTStartup+0xe
8	kernel32.dll!BaseThreadInitThunk+0x14
9	ntdll.dll!RtlUserThreadStart+0x21

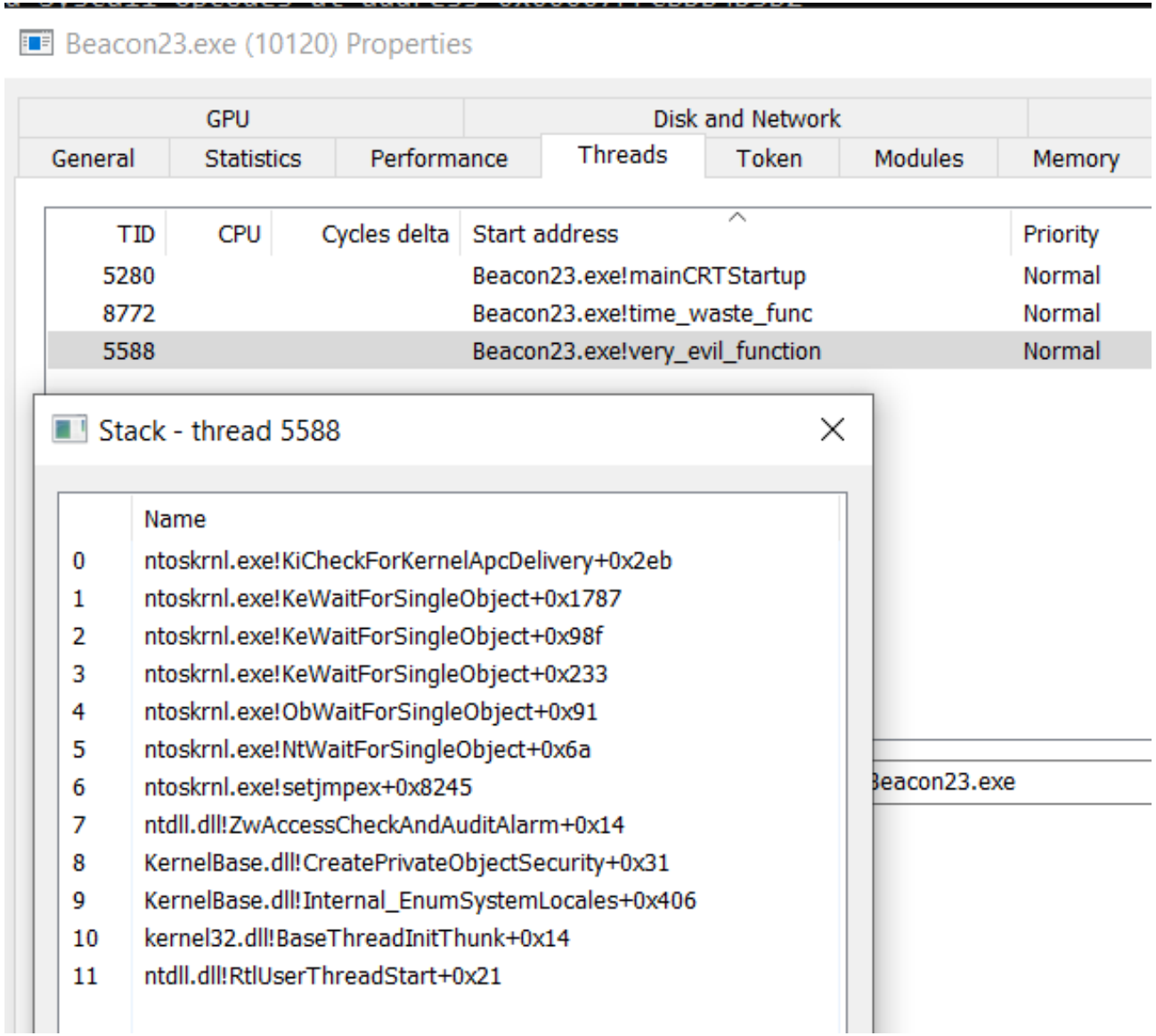
```
void very_evil_function(void)
{
    NTSTATUS status = 0;
    PVOID allocation_ptr = NULL;
    DWORD PAGE_SIZE = 0x1000;

    printf("Calling VirtualAlloc..\n");
#ifdef _DEBUG
    __debugbreak();
#endif
    allocation_ptr = VirtualAlloc(NULL, PAGE_SIZE, MEM_RESERVE | MEM_COMMIT, PAGE_EX
```

Indirect System Calls with Call Stack Spoofing (randomized)

- Unwindable stack
- Correct calling tree

`Kernelbase > NTDLL > Kernel`

- Caller module hidden
- Target kernel function hidden in userspace

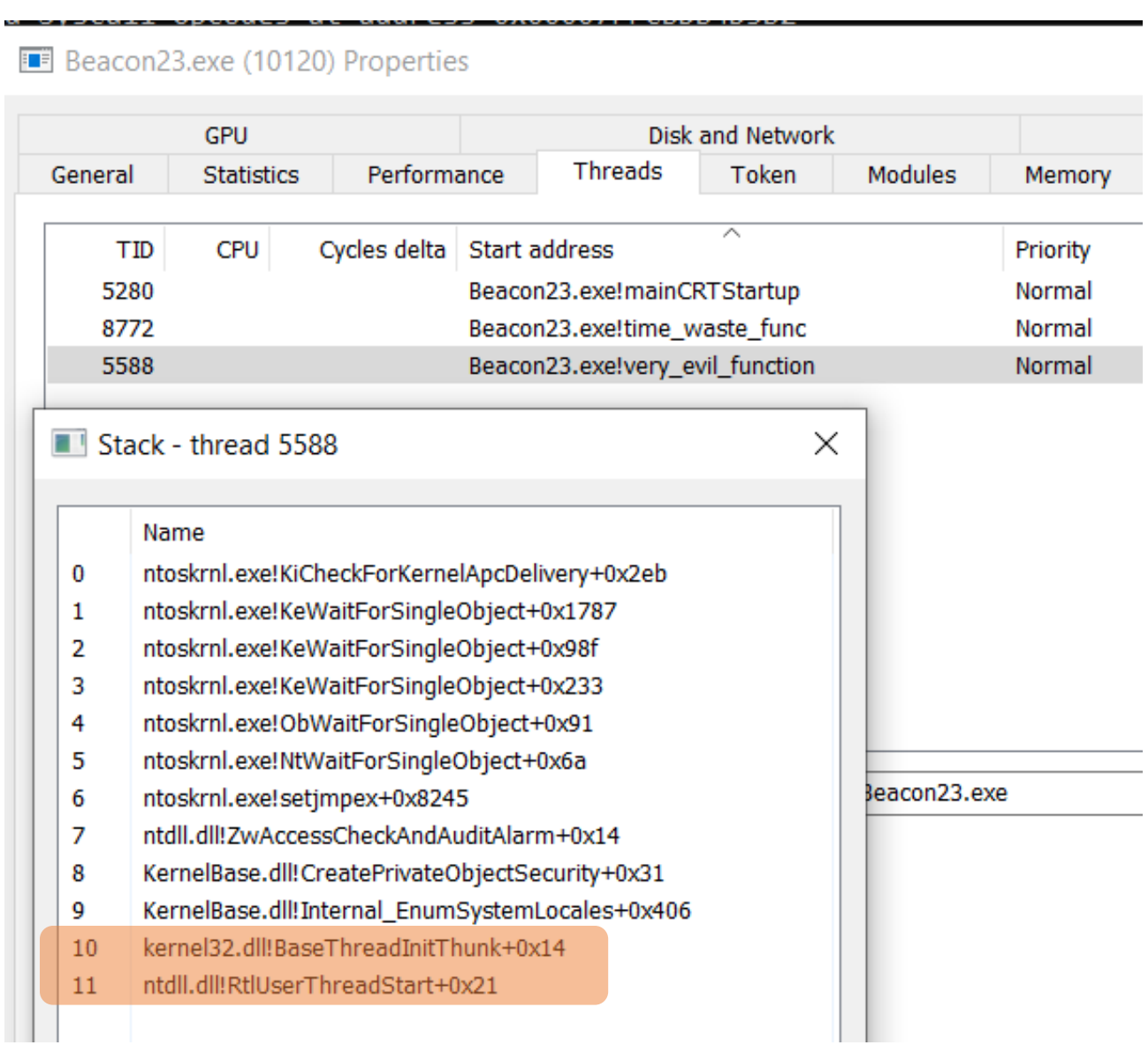


Indirect System Calls with Call Stack Spoofing (randomized)

- **Unwindable stack**
- Correct calling tree

`Kernelbase > NTDLL > Kernel`

- Caller module hidden
- Target kernel function hidden in userspace

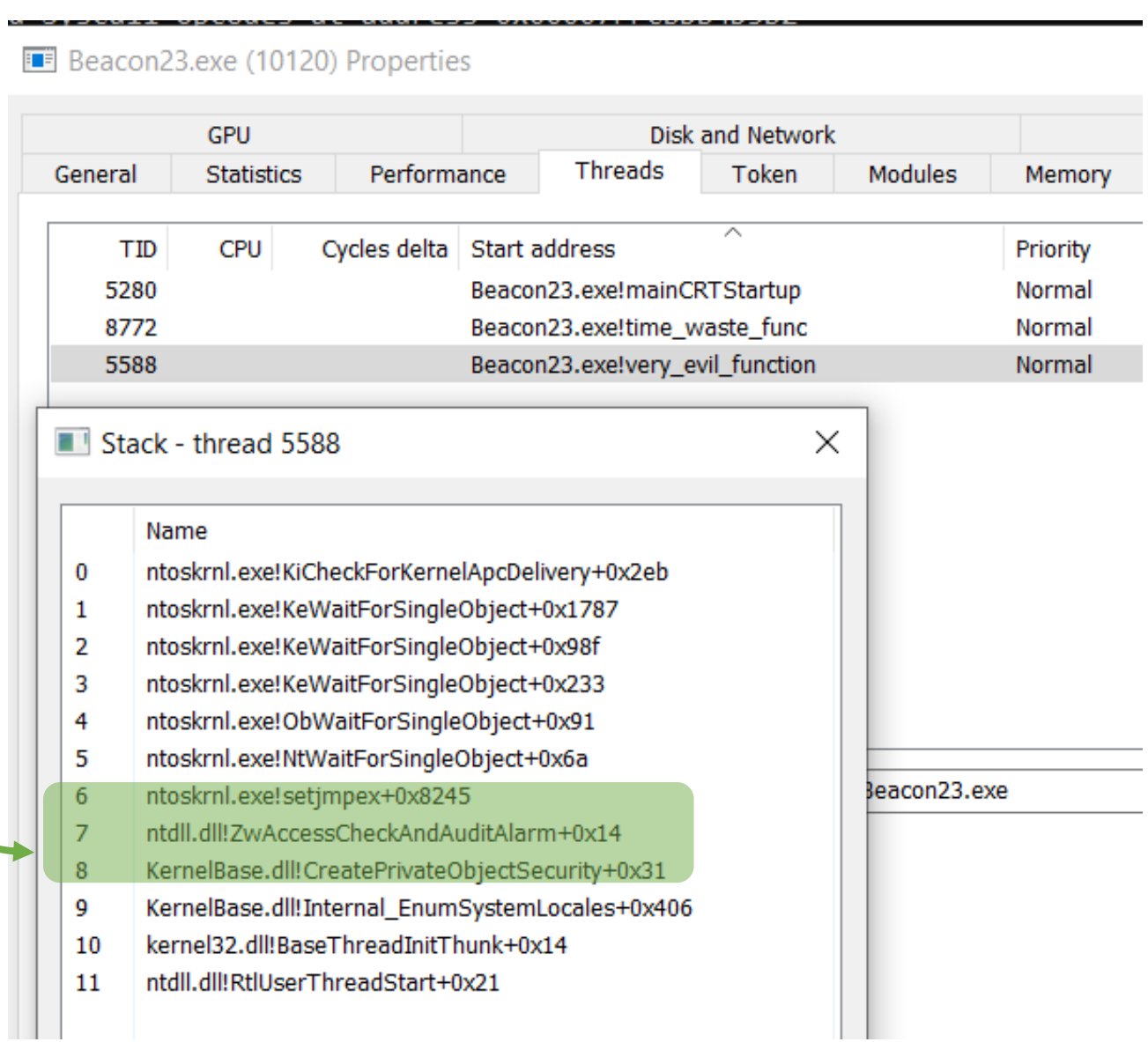


Indirect System Calls with Call Stack Spoofing (randomized)

- Unwindable stack
- **Correct calling tree**

`Kernelbase > NTDLL > Kernel`

- Caller module hidden
- Target kernel function hidden in userspace

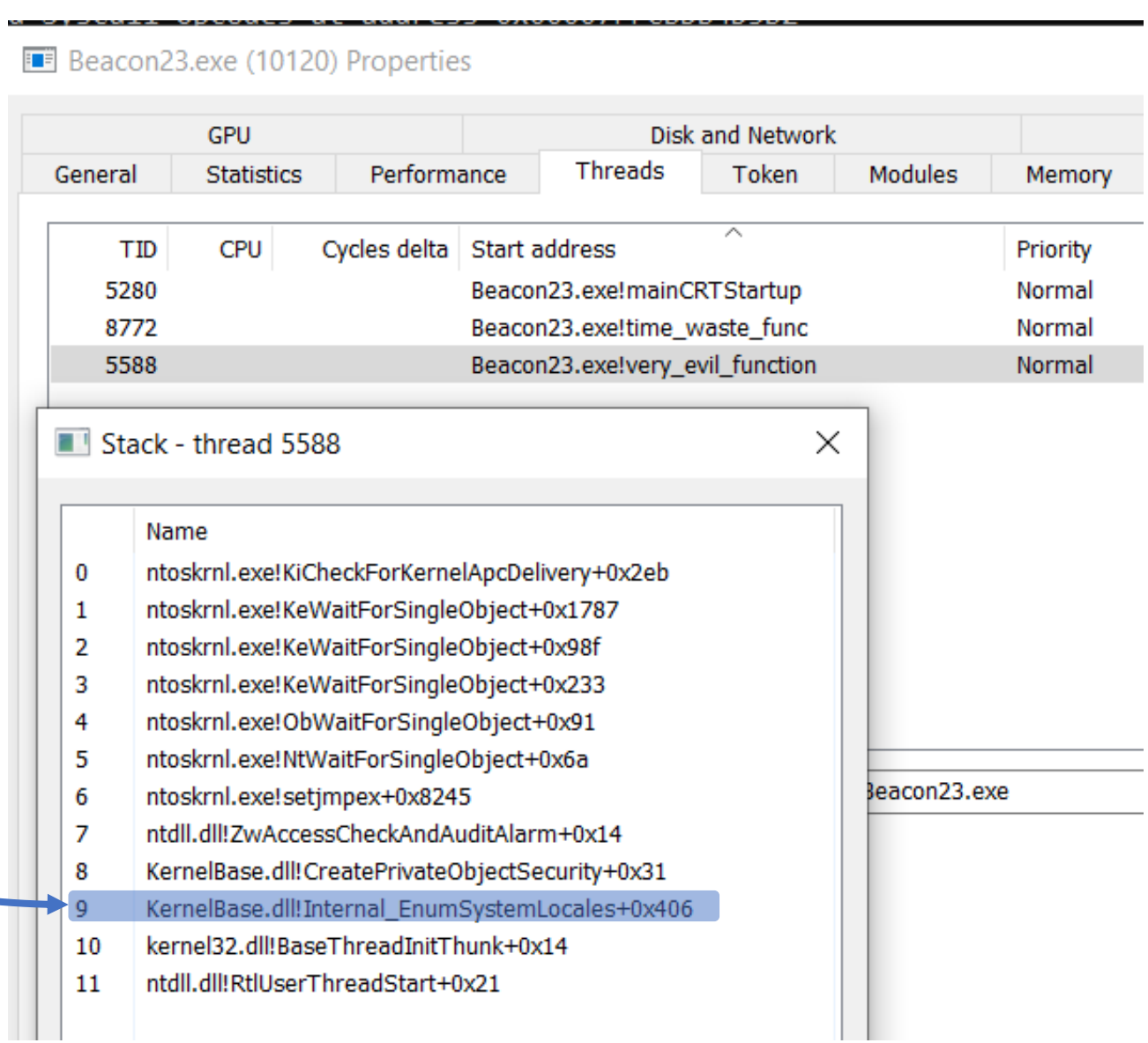


Indirect System Calls with Call Stack Spoofing (randomized)

- Unwindable stack
- Correct calling tree

`Kernelbase > NTDLL > Kernel`

- **Caller module hidden**
- Target kernel function hidden in userspace

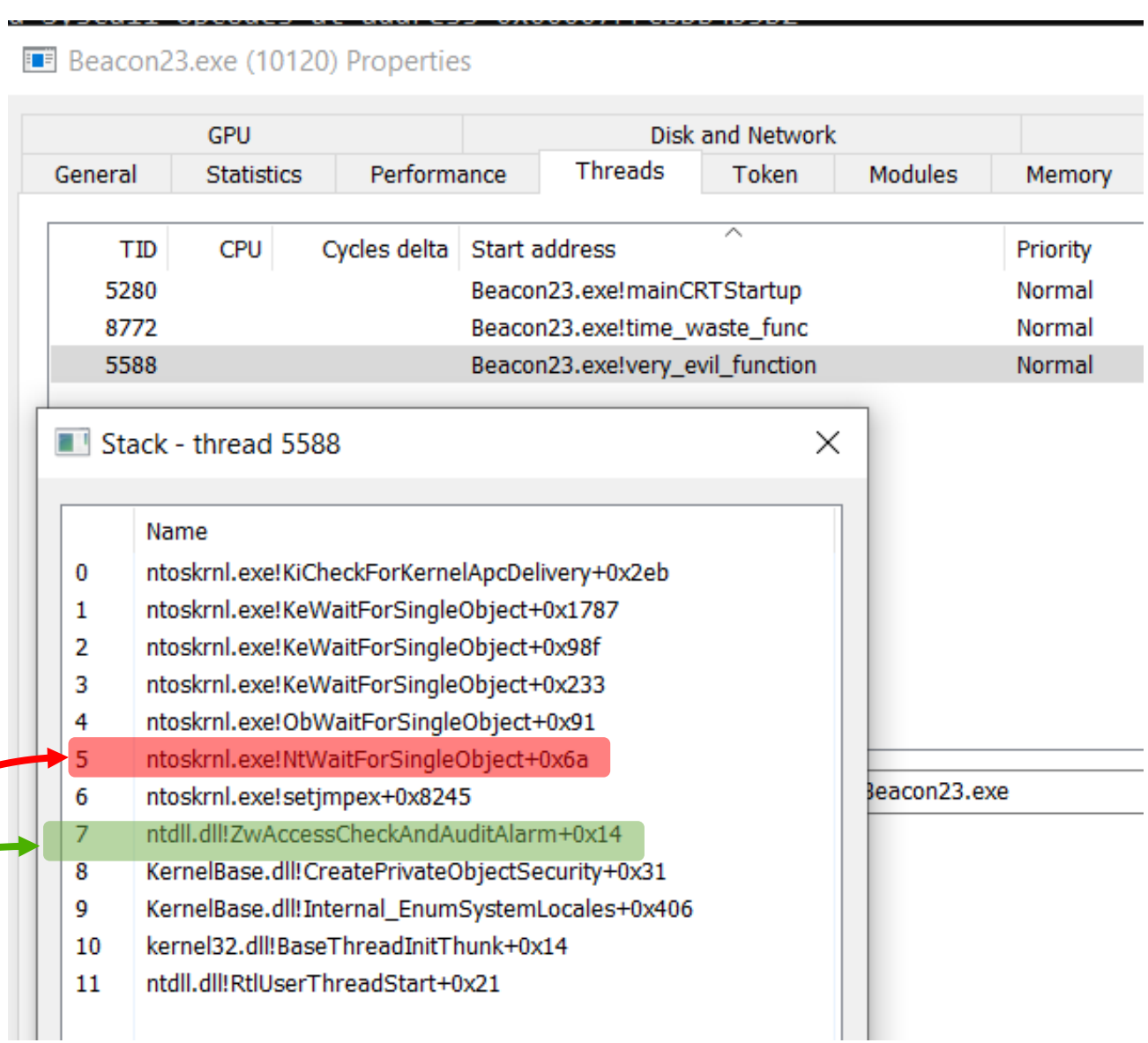


Indirect System Calls with Call Stack Spoofing (randomized)

- Unwindable stack
- Correct calling tree

`Kernelbase > NTDLL > Kernel`

- Caller module hidden
- **Target kernel function hidden in userspace**

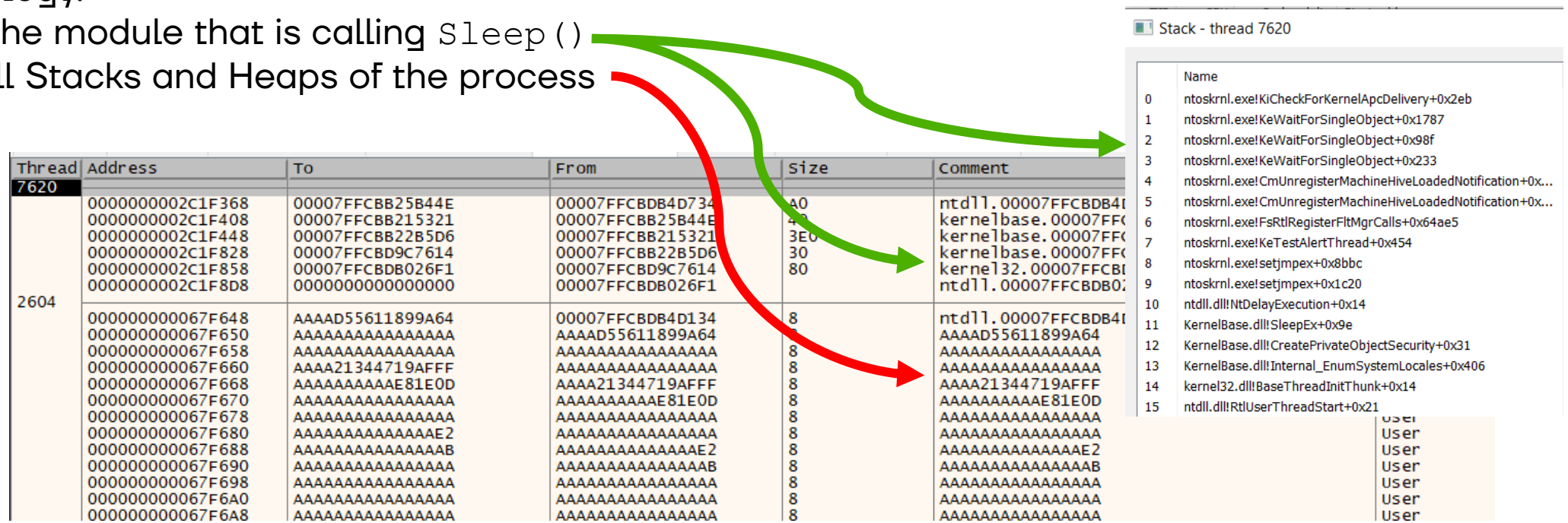


Sleep with Call Stack Spoofing

EDRs may analyse Threads in `WAIT` state looking for sleeping malwares

Evasion Strategy:

- Conceal the module that is calling `Sleep()`
- Encrypt all Stacks and Heaps of the process



Thread	Address	To	From	Size	Comment
7620					
	0000000002C1F368	00007FFCBB25B44E	00007FFCBDB4D734	A0	ntdll.00007FFCBDB4D
	0000000002C1F408	00007FFCBB215321	00007FFCBB25B44E	4	kernelbase.00007FFC
	0000000002C1F448	00007FFCBB22B5D6	00007FFCBB215321	3E0	kernelbase.00007FFC
	0000000002C1F828	00007FFCBD9C7614	00007FFCBB22B5D6	30	kernelbase.00007FFC
	0000000002C1F858	00007FFCBDB026F1	00007FFCBD9C7614	80	kernel32.00007FFCBD
	0000000002C1F8D8	0000000000000000	00007FFCBDB026F1		ntdll.00007FFCBDB02
2604					
	000000000067F648	AAAAD55611899A64	00007FFCBDB4D134	8	ntdll.00007FFCBDB4D
	000000000067F650	AAAAAAAAAAAAAAAA	AAAAD55611899A64	8	AAAAD55611899A64
	000000000067F658	AAAAAAAAAAAAAAAA	AAAAAAAAAAAAAAAA	8	AAAAAAAAAAAAAAAA
	000000000067F660	AAAA21344719AFFF	AAAAAAAAAAAAAAAA	8	AAAAAAAAAAAAAAAA
	000000000067F668	AAAAAAAAAAE81E0D	AAAA21344719AFFF	8	AAAA21344719AFFF
	000000000067F670	AAAAAAAAAAAAAAAA	AAAAAAAAAAE81E0D	8	AAAAAAAAAAE81E0D
	000000000067F678	AAAAAAAAAAAAAAAA	AAAAAAAAAAAAAAAA	8	AAAAAAAAAAAAAAAA
	000000000067F680	AAAAAAAAAAAAAAE2	AAAAAAAAAAAAAAAA	8	AAAAAAAAAAAAAAAA
	000000000067F688	AAAAAAAAAAAAAAAB	AAAAAAAAAAAAAAE2	8	AAAAAAAAAAAAAAE2
	000000000067F690	AAAAAAAAAAAAAAAA	AAAAAAAAAAAAAAAB	8	AAAAAAAAAAAAAAAB
	000000000067F698	AAAAAAAAAAAAAAAA	AAAAAAAAAAAAAAAA	8	AAAAAAAAAAAAAAAA
	000000000067F6A0	AAAAAAAAAAAAAAAA	AAAAAAAAAAAAAAAA	8	AAAAAAAAAAAAAAAA
	000000000067F6A8	AAAAAAAAAAAAAAAA	AAAAAAAAAAAAAAAA	8	AAAAAAAAAAAAAAAA

Stack - thread 7620

	Name	
0	ntoskrnl.exe!KiCheckForKernelApcDelivery+0x2eb	
1	ntoskrnl.exe!KeWaitForSingleObject+0x1787	
2	ntoskrnl.exe!KeWaitForSingleObject+0x98f	
3	ntoskrnl.exe!KeWaitForSingleObject+0x233	
4	ntoskrnl.exe!CmUnregisterMachineHiveLoadedNotification+0x...	
5	ntoskrnl.exe!CmUnregisterMachineHiveLoadedNotification+0x...	
6	ntoskrnl.exe!FsRtlRegisterFltMgrCalls+0x64ae5	
7	ntoskrnl.exe!KeTestAlertThread+0x454	
8	ntoskrnl.exe!setjmpex+0x8bbc	
9	ntoskrnl.exe!setjmpex+0x1c20	
10	ntdll.dll!NtDelayExecution+0x14	
11	KernelBase.dll!SleepEx+0x9e	
12	KernelBase.dll!CreatePrivateObjectSecurity+0x31	
13	KernelBase.dll!Internal_EnumSystemLocales+0x406	
14	kernel32.dll!BaseThreadInitThunk+0x14	
15	ntdll.dll!RtlUserThreadStart+0x21	

BYOVD

(Ab)use a signed driver to execute malicious actions

- WWW primitives
- Process killer
- Handle Leaks
- Be Creative ☺

LOLDrivers project : https://www.loldrivers.io

Example - Kernel Cactus
https://github.com/SpikySabra/Kernel-Cactus
(dbutil_2_3.sys weaponization)

Microsoft implemented a mitigation using a block list of vulnerable drivers

Agent Killer

1. Get handle to the target process from killer process using `NtOpenProcess()`
2. Find in kernel `EPROCESS` structure associated to the killer process
3. Get the Handle Table `HANDLE_TABLE *ObjectTable`
4. Search the Handle associated to the target process
5. Make the handle privileged by editing `HANDLE_TABLE_ENTRY->GrantedAccess`
6. Call `TerminateProcess()`

References: Kernel Cactus

Disable Kernel Callbacks

- Identify the callback array (e.g. `PspCreateProcessNotifyRoutine`)
  - Symbol NOT EXPORTED - We need to find an offset from an exported symbol (Windows version dependent) by analysing the routine that sets the values in the array (e.g. `nt!PsSetCreateProcessNotifyRoutine`)
- Identify which entry in the corresponding callback array is pointing to the EDR driver.
  - The callback handler is likely in the address space of the EDR driver
- Zero the entry ☺

Tamper
~~Disable~~ Kernel Callbacks

- Identify the callback array (e.g. `PspCreateProcessNotifyRoutine`)
  - Symbol NOT EXPORTED - We need to find an offset from an exported symbol (Windows version dependent) by analysing the routine that sets the values in the array (e.g. `nt!PsSetCreateProcessNotifyRoutine`)

- Identify which entry in the corresponding callback array is pointing to the EDR driver.
  - The callback handler is likely in the address space of the EDR driver

- ~~Zero the entry~~ 😀 **Overwrite pointer with our function**

Disable ETW Ti

- ntoskrnl exports `EtwRegister` function to register a ETW provider
- During the execution of `nt!EtwRegister` a _ETW_REG_ENTRY structure is created
- _ETW_REG_ENTRY contains the information associated to an ETW provider
- Every ETW provider is identified by a GUID
- ETW Ti GUID : **f4e1897c-bb5d-5668-f1d8-040f4d8dd344**

Reference: https://securityintelligence.com/x-force/direct-kernel-object-manipulation-attacks-etw-providers/

Disable ETW Ti

- NULL the _ETW_REG_ENTRY pointer. Any functions referencing the registration handle would then assume that the provider had not been initialized.
- NULL the `_ETW_REG_ENTRY->GuidEntry.ProviderEnableInfo` pointer. This should effectively disable the provider's collection capabilities as ProviderEnableInfo is a pointer to a _TRACE_ENABLE_INFO structure which outlines how the provider is supposed to operate.
- `_ETW_REG_ENTRY->GuidEntry.ProviderEnableInfo.IsEnabled = 0`

Reference: https://securityintelligence.com/x-force/direct-kernel-object-manipulation-attacks-etw-providers/

Disable ETW Ti

https://securityintelligence.com/x-force/direct-kernel-object-manipulation-attacks-etw-providers/

https://web.archive.org/web/20220703151911/https://public.cnotools.studio/bring-your-own-vulnerable-kernel-driver-byovkd/exploits/data-only-attack-neutralizing-etwti-provider#neutralizing-etwti-provider

https://github.com/wavestone-cdt/EDRSandblast/blob/master/EDRSandblast/KernellandBypass/ETWThreatIntel.c

https://github.com/SpikySabra/Kernel-Cactus/blob/main/KernelCactus/KernelOps.cpp

Install your Unsigned Driver

Once we can write in Kernel memory, we can disable the signature check.

`ci.dll` is responsible for Windows Driver Signing Enforcement (DSE) management.

Setting `Ci!g_CiOptions` to `0` will allow us to load our driver

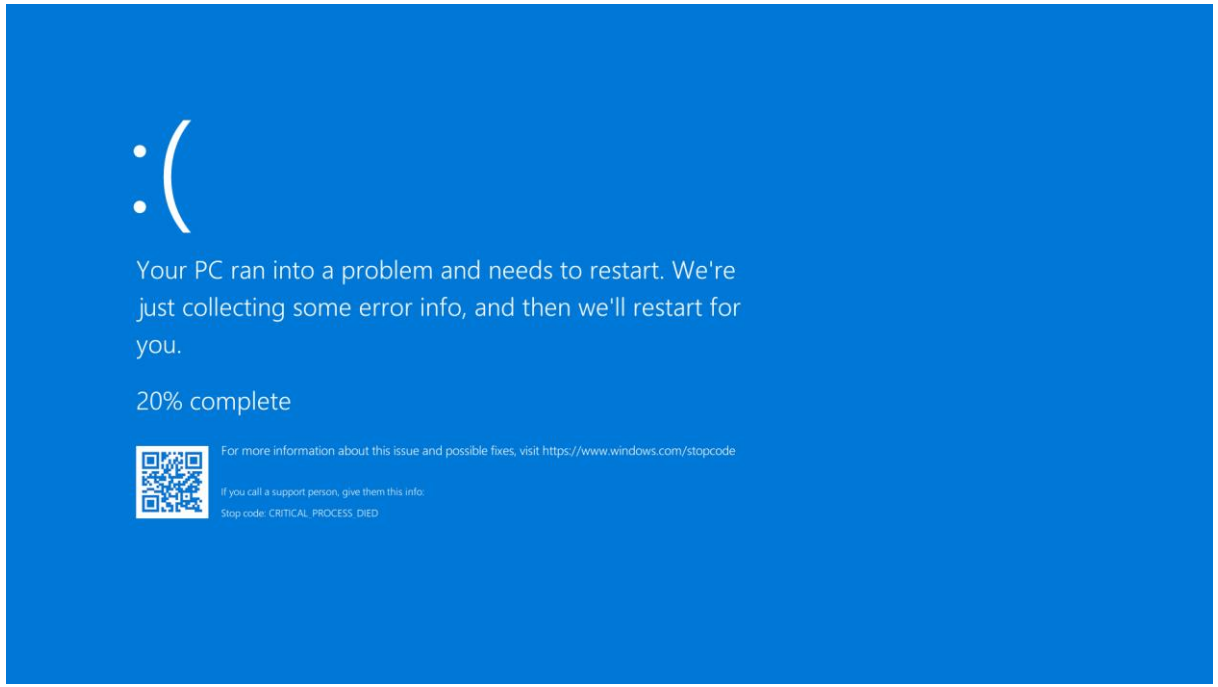
Windows KPP (Parch Guard) periodically checks for tampering of Kernel memory

With great power comes great responsibility

Install your Unsigned Driver

- Find `Ci!g_CiOptions`
- Set `Ci!g_CiOptions` to `0`
- Install Driver
- Set `Ci!g_CiOptions` value back

With great power comes great responsibility



If KPP check happens at the wrong time, GAME OVER

References:
[Loading unsigned Windows drivers without reboot]

Install your Unsigned Driver

- Find `Ci!g_CiOptions`
- Set `Ci!g_CiOptions` to `0`
- Install Driver
- Set `Ci!g_CiOptions` value back

The Symbol `Ci!g_CiOptions` is not exported :(

We need to find the address using an offset from an exported symbol (Windows version dependent)

References:
[Loading unsigned Windows drivers without reboot]

Rootkits

- Unlimited Privileges
- Telemetry tampering is trivial
- Wide surface to be creative

Maybe a topic for another talk..

Rootkits examples ITW

https://www.cisa.gov/sites/default/files/2023-05/aa23-129a_snake_malware_2.pdf
https://www.trendmicro.com/en_ph/research/23/q/hunting-for-a-new-stealthy-universal-rootkit-loader.html

THANKS!

proud sponsor of