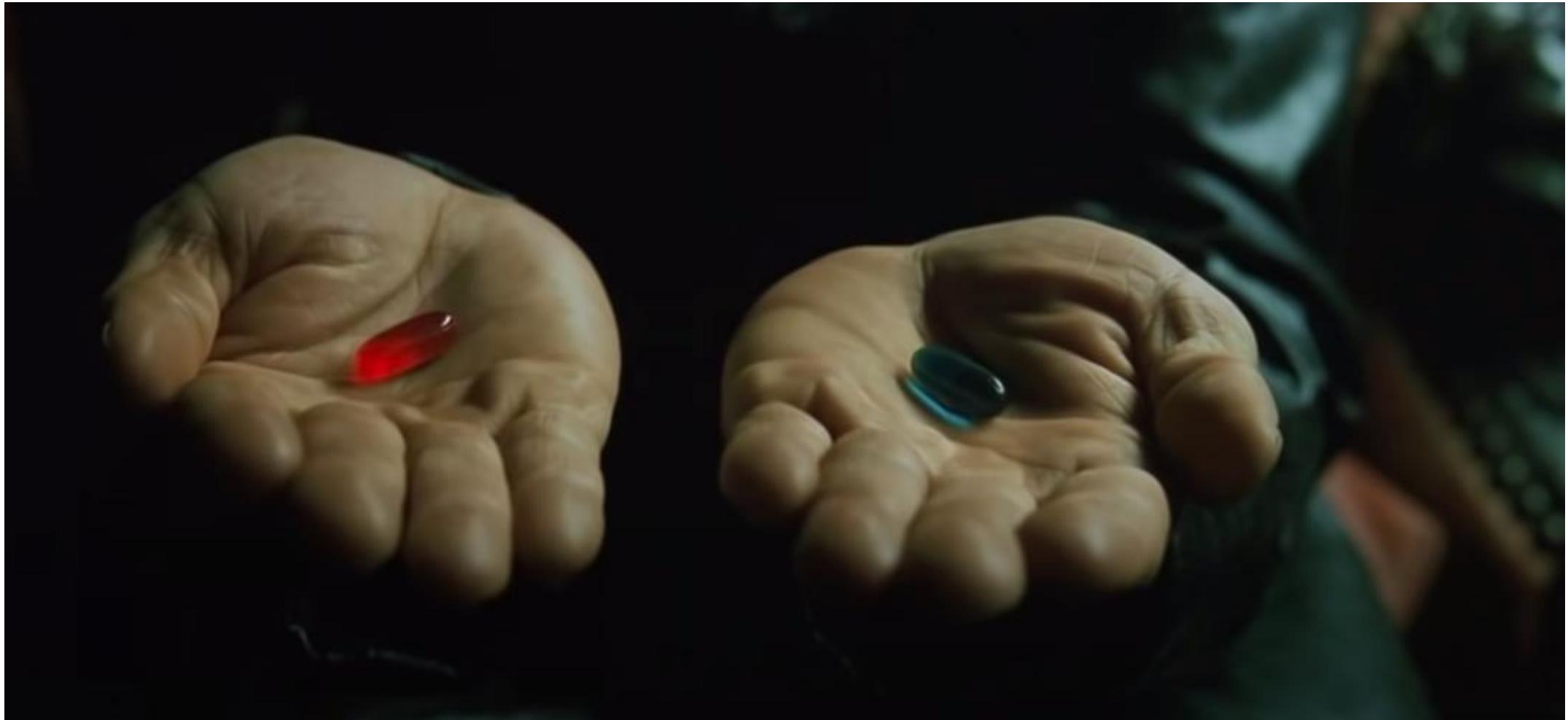


# EDR Psychedelia

Pushing Boundaries of Perception  
for Reshaping EDR's Truths



# 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

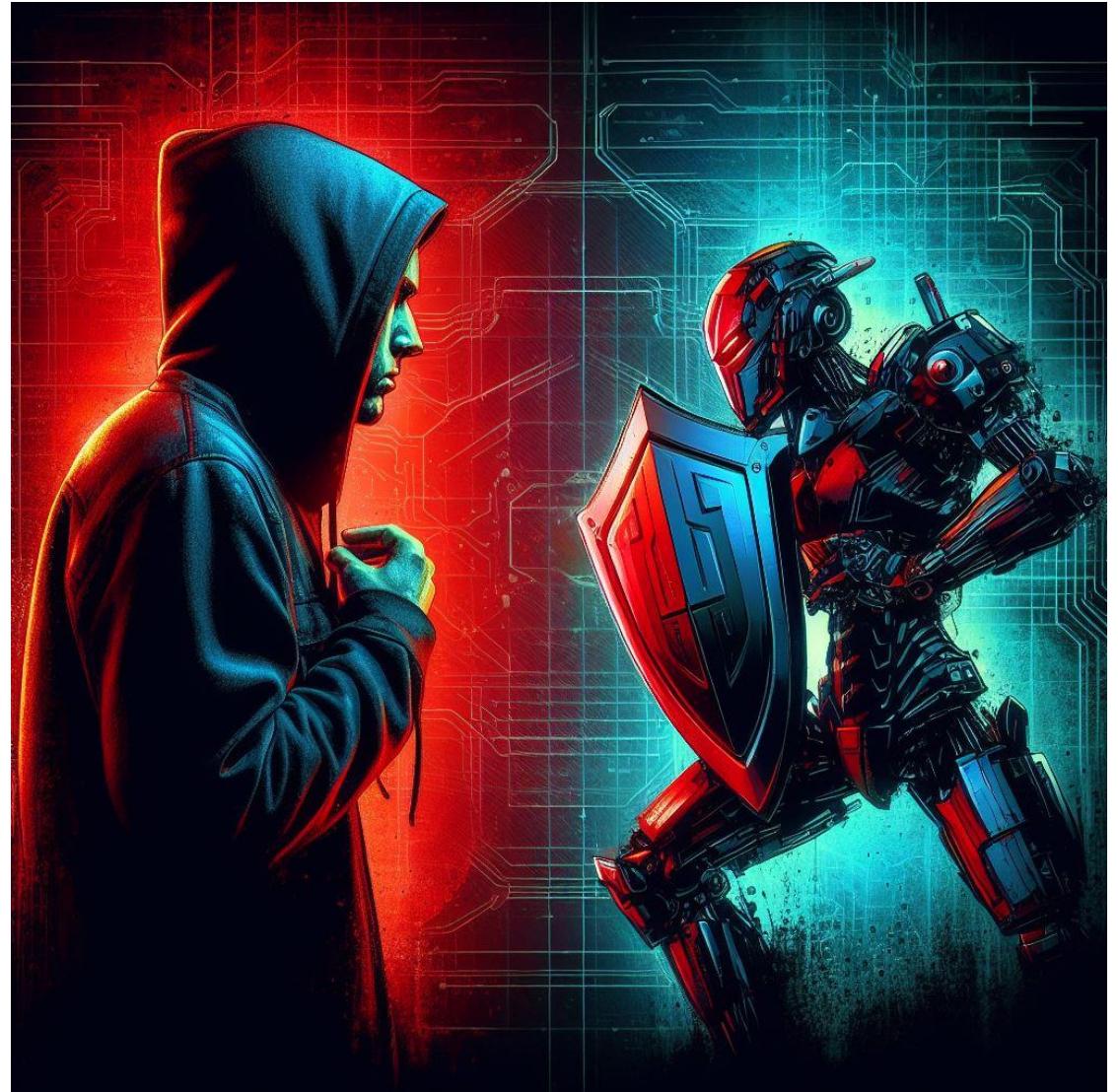
# RT vs EDR

## ANALYSIS

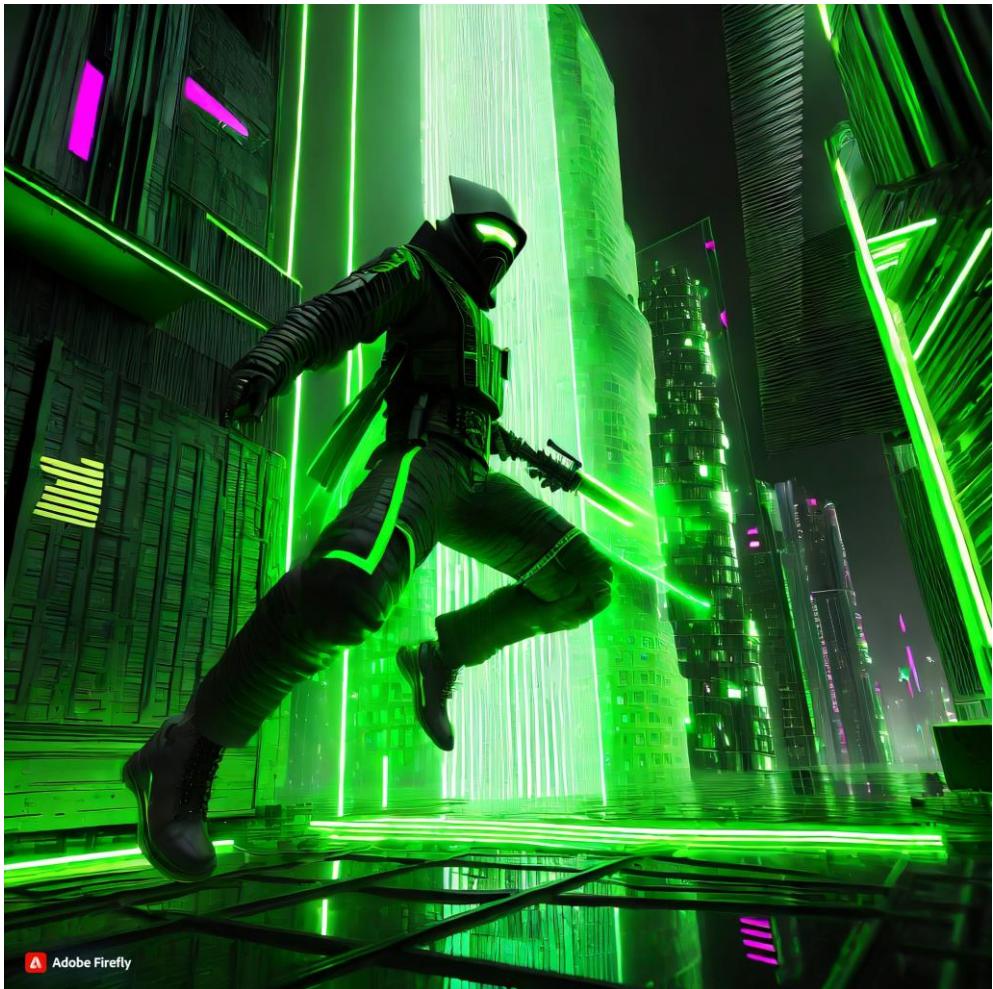
- Identify telemetry sources
- Identify patterns used to trigger a detection event

## 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 Sideloaded , 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



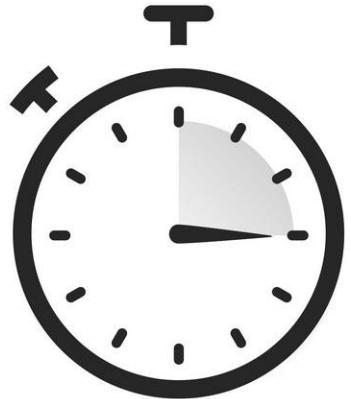
# RT vs EDR



## 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

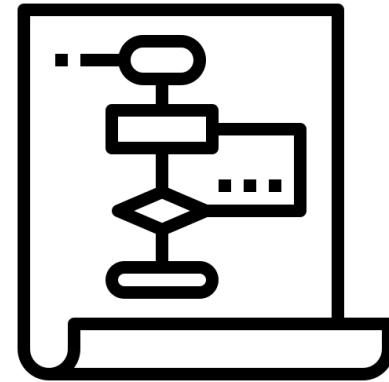


Threads may be periodically scanned.

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
- ...

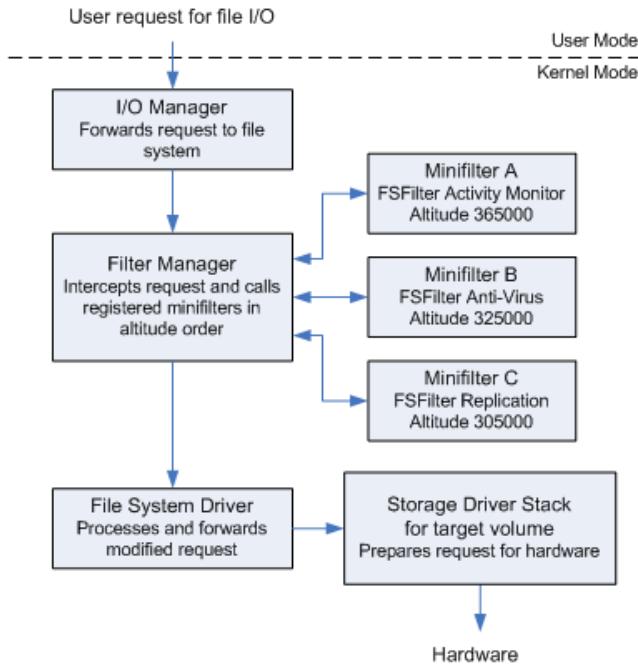
# Dynamic Analysis – Behavioural Detection

- 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

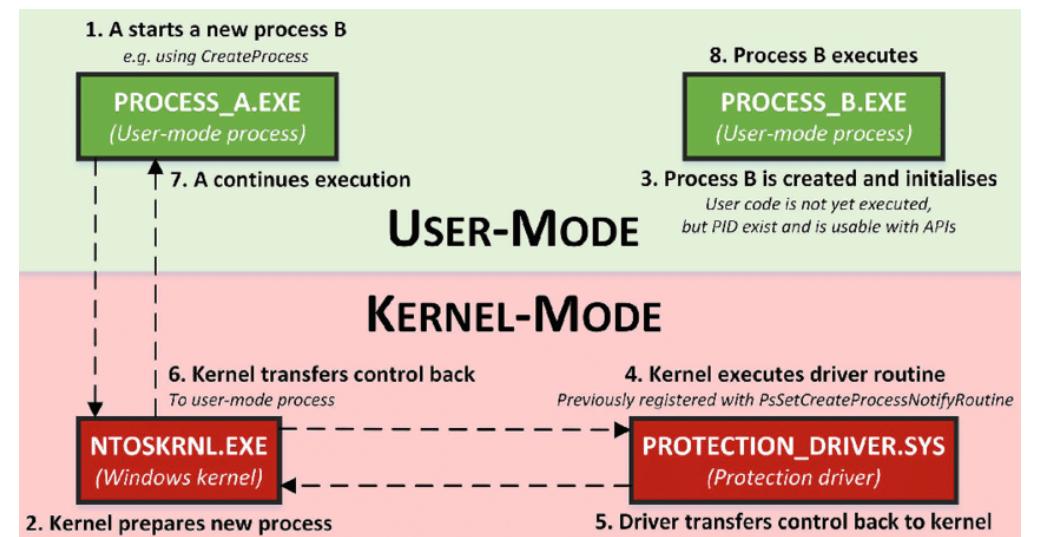
# Dynamic Analysis – Behavioural Detection

## Kernel Callbacks

- PsSetCreateProcessNotifyRoutine
- PsSetCreateThreadNotifyRoutine
- PsSetLoadImageNotifyRoutine



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



Src: Fast and Furious: outrunning Windows Kernel Notification Routines from User-Mode  
[http://dx.doi.org/10.1007/978-3-030-52683-2\\_4](http://dx.doi.org/10.1007/978-3-030-52683-2_4)

## 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
```

Provider	GUID	
Microsoft-Windows-Threat-Intelligence	{F4E1897C-BB5D-5668-F1D8-040F4D8DD344}	
Value	Keyword	Description
0x0000000000000001	KERNEL_THREATINT_KEYWORD_ALLOCVM_LOCAL	
0x0000000000000002	KERNEL_THREATINT_KEYWORD_ALLOCVM_LOCAL_KERNEL_CALLER	
0x0000000000000004	KERNEL_THREATINT_KEYWORD_ALLOCVM_REMOTE	
0x0000000000000008	KERNEL_THREATINT_KEYWORD_ALLOCVM_REMOTE_KERNEL_CALLER	
0x0000000000000010	KERNEL_THREATINT_KEYWORD_PROTECTVM_LOCAL	
0x0000000000000020	KERNEL_THREATINT_KEYWORD_PROTECTVM_LOCAL_KERNEL_CALLER	
0x0000000000000040	KERNEL_THREATINT_KEYWORD_PROTECTVM_REMOTE	
0x0000000000000080	KERNEL_THREATINT_KEYWORD_PROTECTVM_REMOTE_KERNEL_CALLER	
0x0000000000000100	KERNEL_THREATINT_KEYWORD_MAPVIEW_LOCAL	
0x0000000000000200	KERNEL_THREATINT_KEYWORD_MAPVIEW_LOCAL_KERNEL_CALLER	
0x0000000000000400	KERNEL_THREATINT_KEYWORD_MAPVIEW_REMOTE	
0x0000000000000800	KERNEL_THREATINT_KEYWORD_MAPVIEW_REMOTE_KERNEL_CALLER	
0x0000000000001000	KERNEL_THREATINT_KEYWORD_QUEUEUSERAPC_REMOTE	
0x0000000000002000	KERNEL_THREATINT_KEYWORD_QUEUEUSERAPC_REMOTE_KERNEL_CALLER	
0x0000000000004000	KERNEL_THREATINT_KEYWORD_SETTHREADCONTEXT_REMOTE	
0x0000000000008000	KERNEL_THREATINT_KEYWORD_SETTHREADCONTEXT_REMOTE_KERNEL_CALLER	
0x0000000000010000	KERNEL_THREATINT_KEYWORD_READVM_LOCAL	
0x0000000000020000	KERNEL_THREATINT_KEYWORD_READVM_REMOTE	

# 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

The screenshot shows the Immunity Debugger interface with two windows side-by-side. The left window displays assembly code for the `EtwEventWrite` function, which is part of the `AmsiScanBuffer` module. The right window shows the memory map for the same module, listing various memory regions and their addresses.

ints	Memory Map	Call Stack	SEH	Script	Symbols	So
6BDD59	8BFF	mov edi,edi				AmsiScanBuffer
6BDD59	55	push ebp				
6BDD59	8BEC	mov ebp,esp				
6BDD59	83EC 18	sub esp,18				
6BDD59	53	push ebx				
6BDD59	56	push esi				
6BDD59	A1 0000DE6B	mov eax,dword ptr ds:[6BDE00]				
6BDD59	33DB	xor ebx,ebx				
6BDD59	90	nop				
6BDD59	8B75 08	mov esi,dword ptr ss:[ebp+8]				
6BDD59	3D 0000DE6B	cmp eax,amsi.6BDE0000				
6BDD59	v 74 1C	je amsi.6BDD5998				

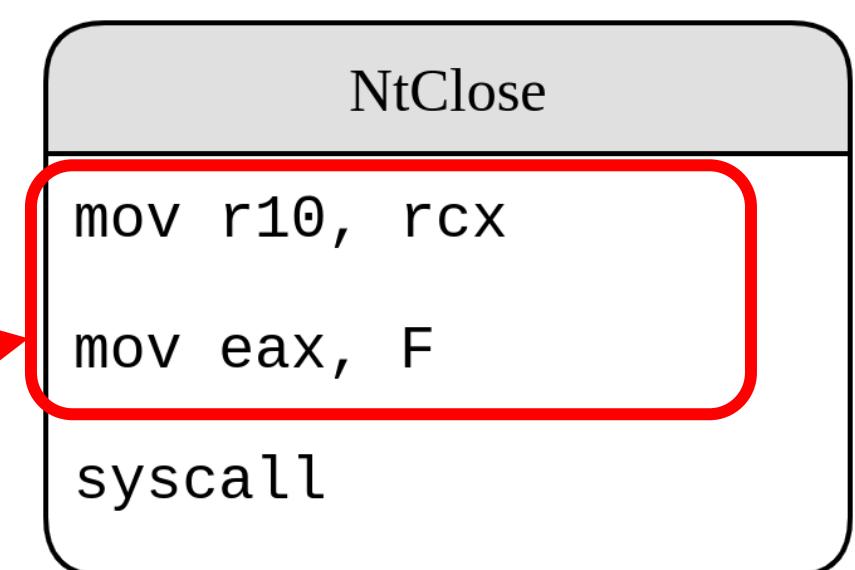
# Hooking

- 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

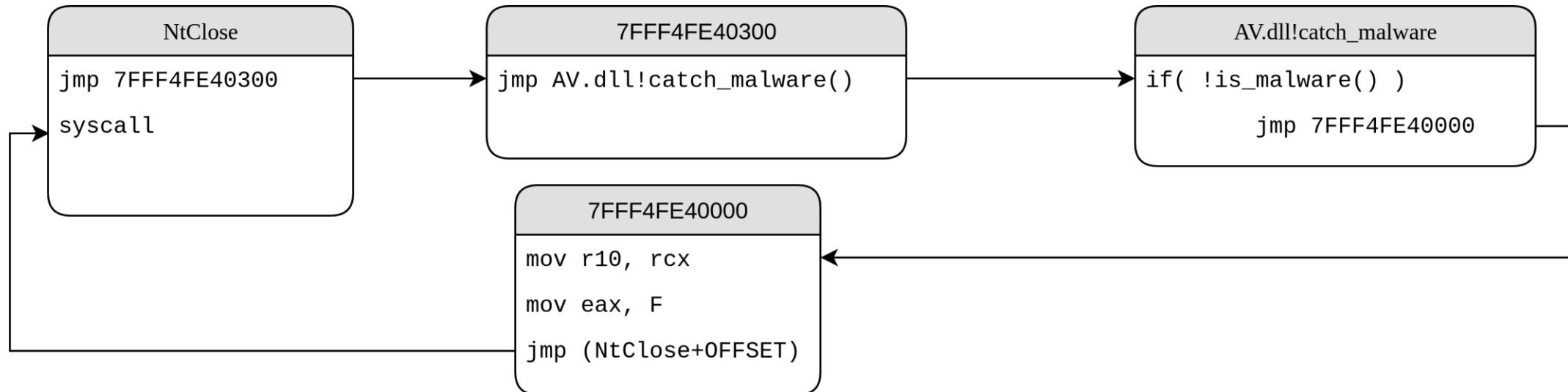
Hijack the execution flow to allow monitoring

- 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!1.7FFFCE3C8D5	
0F05	syscall	
C3	ret	
CD 2E	int 2E	



# Hooking



Notes	Breakpoints	Memory Map	Call Stack	SEH	Script	Symbols	Source	Referer
00007FFFCE3C8C0	^ E9 3B3A0080	jmp 7FFF4FE40300					NtClose	
00007FFFCE3C8C5	0000	add byte ptr ds:[rax],a1						
00007FFFCE3C8C7	00F6	add dh,dh						
00007FFFCE3C8C9	04 25	add a1,25						
00007FFFCE3C8CB	0803	or byte ptr ds:[rbx],a1						
00007FFFCE3C8CC	EE	....						

# 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/>

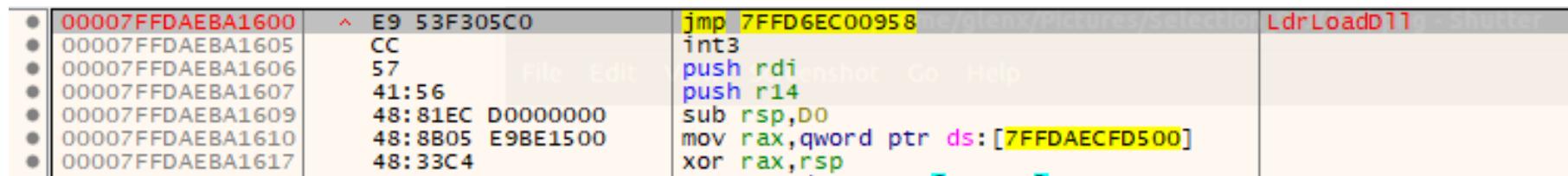
# Where Are The Stubs?

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

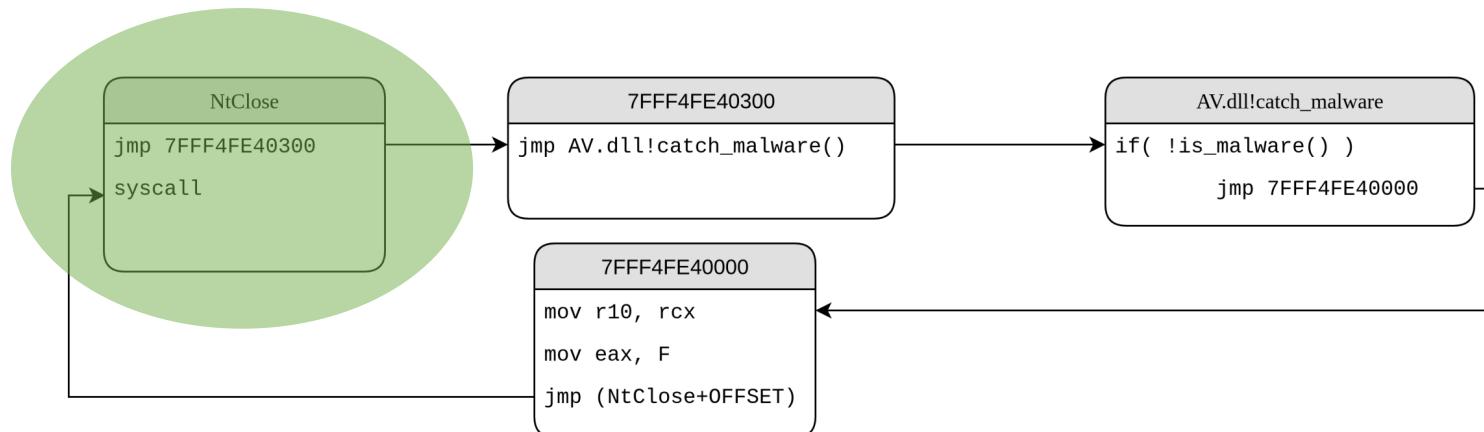
```
1241 static PVOID detour_alloc_region_from_lo(PBYTE pbLo, PBYTE pbHi)
1242 {
1243     PBYTE pbTry = detour_alloc_round_up_to_region(pbLo);
1244
1245     DETOUR_TRACE((" Looking for free region in %p..%p from %p:\n", pbLo, pbHi, pbTry))
1246
1247     for (; pbTry < pbHi;) {
1248         MEMORY_BASIC_INFORMATION mbi;
1249
1250         if (pbTry >= s_pSystemRegionLowerBound && pbTry <= s_pSystemRegionUpperBound)
1251             // Skip region reserved for system DLLs, but preserve address space entropy
1252             pbTry += 0x80000000;
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
1274         if (pv != NULL) {
1275             return pv;
1276         }
1277     }
1278     else if (GetLastError() == ERROR_DYNAMIC_CODE_BLOCKED) {
```

# Where Are The Stubs?

Ntdll!LdrLoadDll hooked by AVG



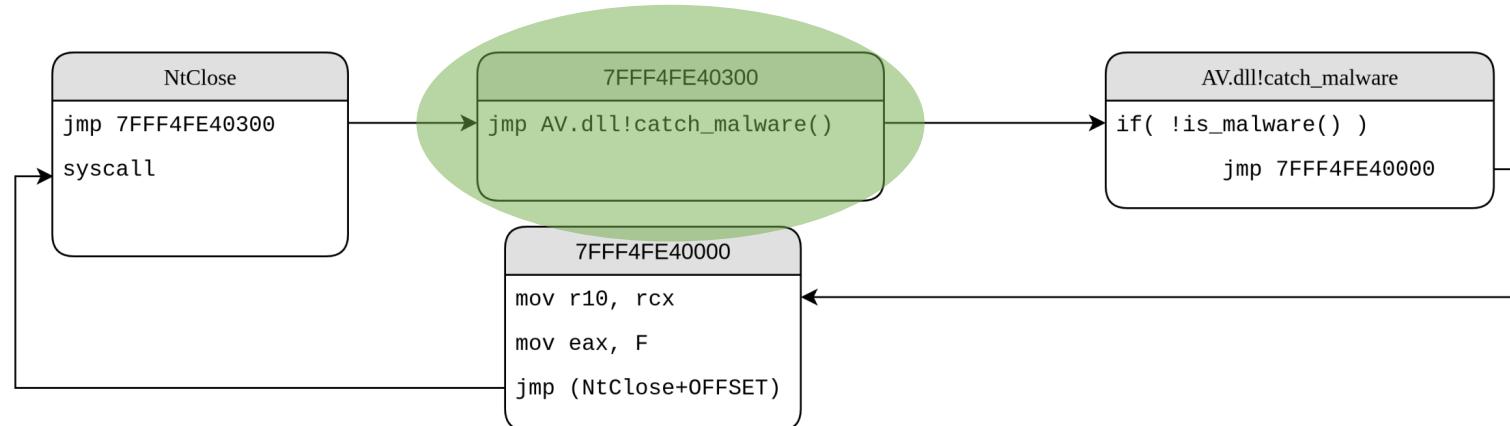
00007FFDAEBA1600	^ E9 53F305C0	jmp 7FFD6EC00958 int3 push rdi push r14 sub rsp,00 mov rax,qword ptr ds:[7FFDAECFD500] xor rax,rspl	LdrLoadDll - SHADER
00007FFDAEBA1605	CC		
00007FFDAEBA1606	57	File Edit	
00007FFDAEBA1607	41:56		
00007FFDAEBA1609	48:81EC D0000000		
00007FFDAEBA1610	48:8B05 E9BE1500		
00007FFDAEBA1617	48:33C4		



# Where Are The Stubs?

“Hooking” stub

Breakpoints				Memory Map	Call Stack	SEH	Script	Symbols	Source
●	00007FFD6EC00958	FF25	F2FFFFFF		jmp qword ptr ds:[7FFD6EC00950]				
●	00007FFD6EC0095E	CC			int3				
●	00007FFD6EC0095F	CC			int3				



# Where Are The Stubs?

“Hooking” stub

The screenshot shows two windows of a debugger interface. Both windows have tabs for Breakpoints, Memory Map, Call Stack, SEH, Script, Symbols, and Source. The top window displays assembly code:

```
00007FFD6EC00958 FF25 F2FFFFFF jmp qword ptr ds:[7FFD6EC00950]
00007FFD6EC0095E CC int3
00007FFD6EC0095F CC int3
```

The bottom window shows a more detailed assembly dump for the first instruction:

```
00007FFD6EC00958 FF25 F2FFFFFF jmp qword ptr ds:[7FFD6EC00950]
00007FFD6EC0095E CC int3
00007FFD6EC0095F CC int3
00007FFD6EC00960 4C:8BD1 mov rbp,rbx
00007FFD6EC00963 B8 8C100000 push rbp
00007FFD6EC00968 00007FFD85FB2200 aswhook.00007FFD85FB2200
00007FFD6EC0096E CC int3
00007FFD6EC0096F CC int3
00007FFD6EC00970 CC int3
00007FFD6EC00971 CC int3
00007FFD6EC00972 CC int3
00007FFD6EC00973 CC int3
00007FFD6EC00974 CC int3
00007FFD6EC00975 CC int3
```

A tooltip or callout box highlights the assembly code starting at address 00007FFD6EC00958, which contains a jump to memory location 7FFD6EC00950. The tooltip content is:

```
jmp qword ptr ds:[7FFD6EC00950]
int3
int3
aswhook.00007FFD85FB2200
push rbx
sub rsp,20
jmp call qword ptr ds:[7FFD85FBAD0]
int3
mov ebx,eax
int3
call aswhook.7FFD85FB57C0
int3
mov eax,ebx
int3
add rsp,20
int3
pop rbx
int3
ret
int3
```

# Where Are The Stubs?

## AVG DLL

Screenshot of a debugger interface showing assembly code and a control flow graph.

The assembly code pane shows the following sequence:

```
push rbx  
sub rsp, 20  
call qword ptr ds:[7FFD85FBADEO]  
mov rcx, rax  
call 00007FFD6EC00900  
mov qword ptr ss:[rsp+10], rbx  
add rsi, 1  
push rsi  
pop qword ptr ds:[7FFD6EC00948]  
ret
```

The instruction `call qword ptr ds:[7FFD85FBADEO]` is highlighted with a red box.

The control flow graph below illustrates the flow of control:

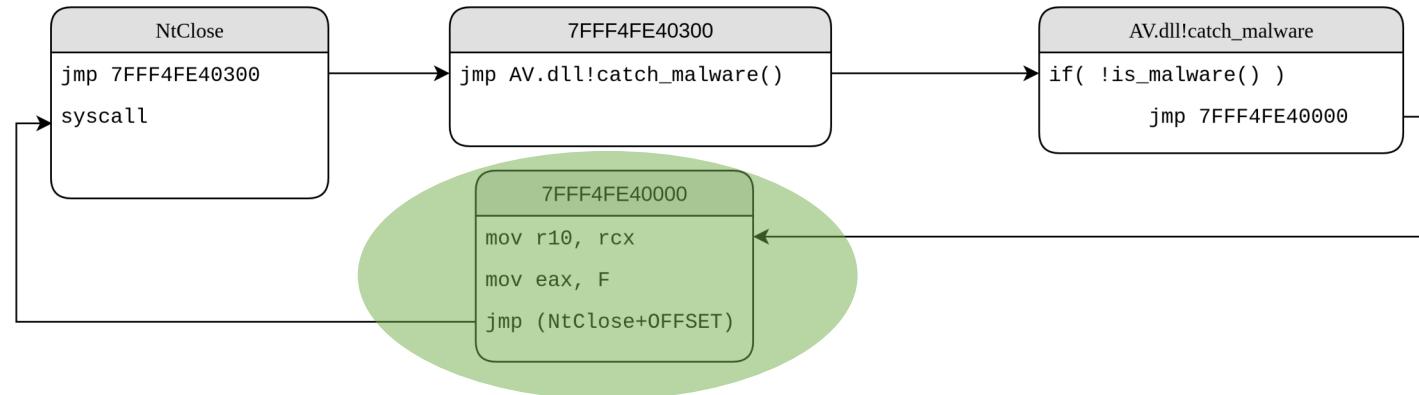
```
graph LR; NtClose[NtClose] --> 7FFF4FE40300[7FFF4FE40300]; 7FFF4FE40300 --> AVdllCatchMalware[AV.dll!catch_malware]; AVdllCatchMalware --> 7FFF4FE40000[7FFF4FE40000]; 7FFF4FE40000 --> NtClose
```

A green oval highlights the `AV.dll!catch_malware` function, which contains the assembly code shown above. The `if( !is_malware() )` condition is also highlighted.

# Where Are The Stubs?

Trampoline to come back to the original function (AVG)

Breakpoints	Memory Map	Call Stack	SEH	Script	Symbols	Source
00007FFD6EC00900	48:895C24 10	mov qword ptr ss:[rsp+10],rbx				
00007FFD6EC00905	56	push rsi				
00007FFD6EC00906	FF25 3C000000	jmp qword ptr ds:[7FFD6EC00948]				
00007FFD6EC00907	CC	int3				



# Where Are The Stubs?

Trampoline to come back to the original function (AVG)

The screenshot shows a debugger interface with multiple tabs at the top: Breakpoints, Memory Map, Call Stack, SEH, Script, Symbols, Source, and Reference. The assembly window displays the following code:

```
00007FFD6EC00900 48:895C24 10    mov qword ptr ss:[rsp+10],rbx
00007FFD6EC00905 56                push rsi
00007FFD6EC00906  FF25 3C000000    jmp qword ptr ds:[7FFD6EC00948]
00007FFD6EC0090C  CC                int3
00007FFD6EC0090D  CC                int nt!1.00007FFDAEBA1606
00007FFD6EC0090E  CC                int push rdi
00007FFD6EC0090F  CC                int push r14
00007FFD6EC00910  CC                int sub rsp,DO
00007FFD6EC00911  CC                int mov rax,qword ptr ds:[7FFDAECD500]
00007FFD6EC00912  CC                int xor rax,rsp
00007FFD6EC00913  CC                int mov qword ptr ss:[rsp+C0],rax
00007FFD6EC00914  CC                int mov r14,r9
00007FFD6EC00915  CC                int mov rdi,r8
00007FFD6EC00916  CC                int mov r10,rdx
00007FFD6EC00917  CC                int mov rsi,rcx
00007FFD6EC00918  CC                int test rdx,rdx
00007FFD6EC00919  CC                int je nt!1.7FFDAEBA175F
00007FFD6EC0091A  CC                int mov eax,dword ptr ds:[rdx]
00007FFD6EC0091B  CC                int mov ecx,dword ptr ds:[rdx]
00007FFD6EC0091C  CC                int and ecx,4
00007FFD6EC0091D  CC                int add ecx,ecx
00007FFD6EC0091E  06                ?? mov edx,ecx
00007FFD6EC0091F  CC                int or edx,40
00007FFD6EC00920  48:895C24 10    mov and al,2
00007FFD6EC00925  56                push mov eax,dword ptr ds:[r10]
00007FFD6EC00926  CC                int3
00007FFD6EC00927  CC                int3
00007FFD6EC00928  CC                int3
00007FFD6EC00929  CC                int3
```

The assembly code is a trampoline stub. It pushes the current stack pointer onto the stack, jumps to a memory location (7FFD6EC00948), and then performs a series of operations involving registers rsi, r14, r9, r8, r10, rdx, rcx, and eax, before finally jumping to a specific address (nt!1.7FFDAEBA175F) and performing additional operations like mov, and, or, add, and push.

# Where Are The Stubs?

## Memory area hosting the stubs (AVG)

00007FF7EA1E5000	0000000000001000	".rsrc"	Resources	IMG	-R---	ERWC-
00007FFF/EAE60000	0000000000001000	.rsrc	Base relocations	IMG	-R---	ERWC-
00007FFF456A0000	00000000000010000			PRV	ER---	ERW--
0000755561A20000	0000000000001000	aswbook.dll		TMG	-P---	FRWC-
00007FFF61A31000	0000000000007000	".text"	Executable code	IMG	ER---	ERWC-
0000755561A20000	0000000000002000	".rdata"	Read-only initialized data	TMG	-P---	FRWC-

“Hooking” stub (AVG)

Breakpoints	Memory Map	Call Stack	SEH	Script	Symbols	Source
	00007FFF456A0238	FF25 F2FFFFFF	jmp qword ptr ds:[7FFF456A0230]			
	00007FFF456A023E	CC	int3			
	00007FFF456A023F	CC	int3			
	00007FFF456A0240	0000	add byte [rsp+10], al			

Trampoline to come back to the original function (AVG)

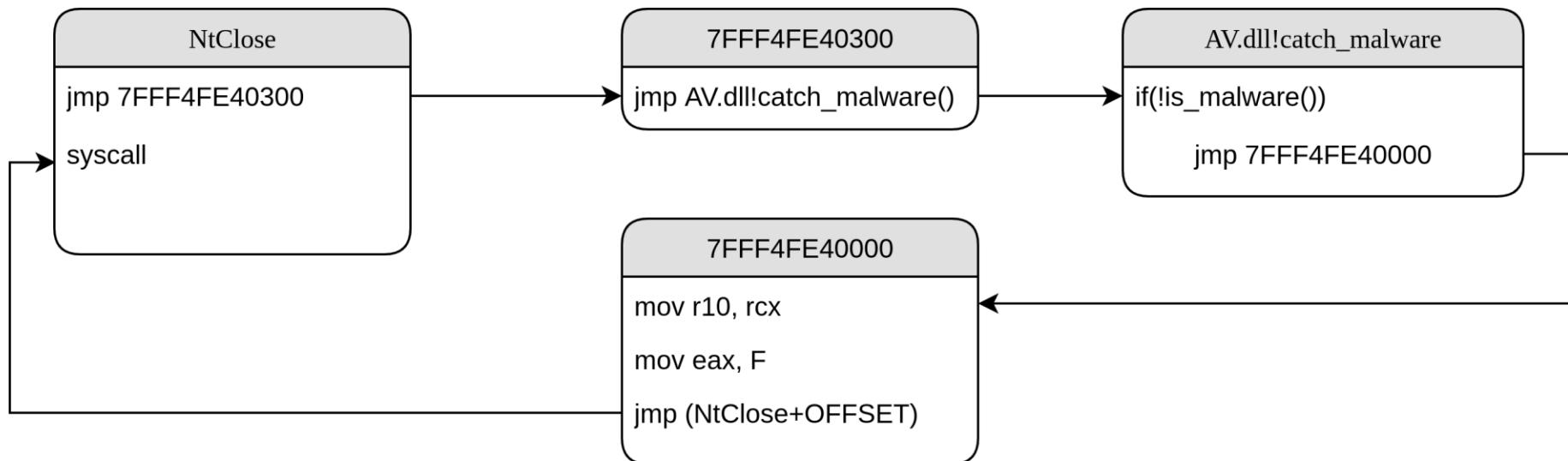
00007FFF456A01DF	CC	int3
00007FFF456A01E0	48:895C24 10	mov qword ptr ss:[rsp+10],rbx
00007FFF456A01E5	56	push rsi
00007FFF456A01E6	FF25 3C000000	jmp qword ptr ds:[7FFF456A0228]
00007FFF456A01EC	CC	int3

# 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

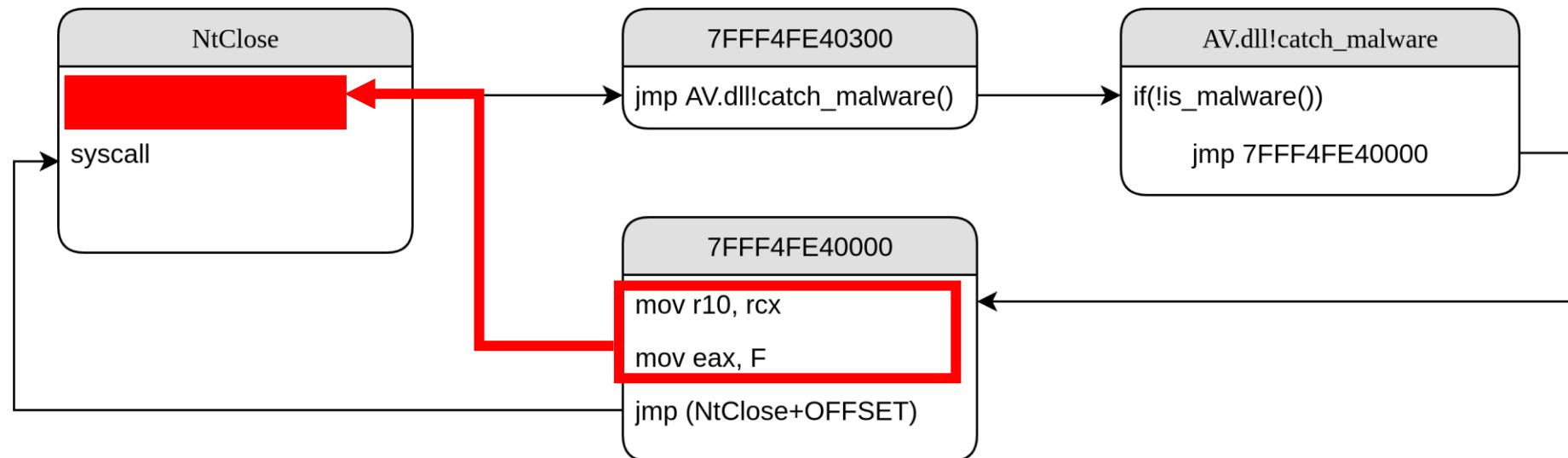
# UNHOOK IDEA

Walk the pointers to retrieve the original prologue



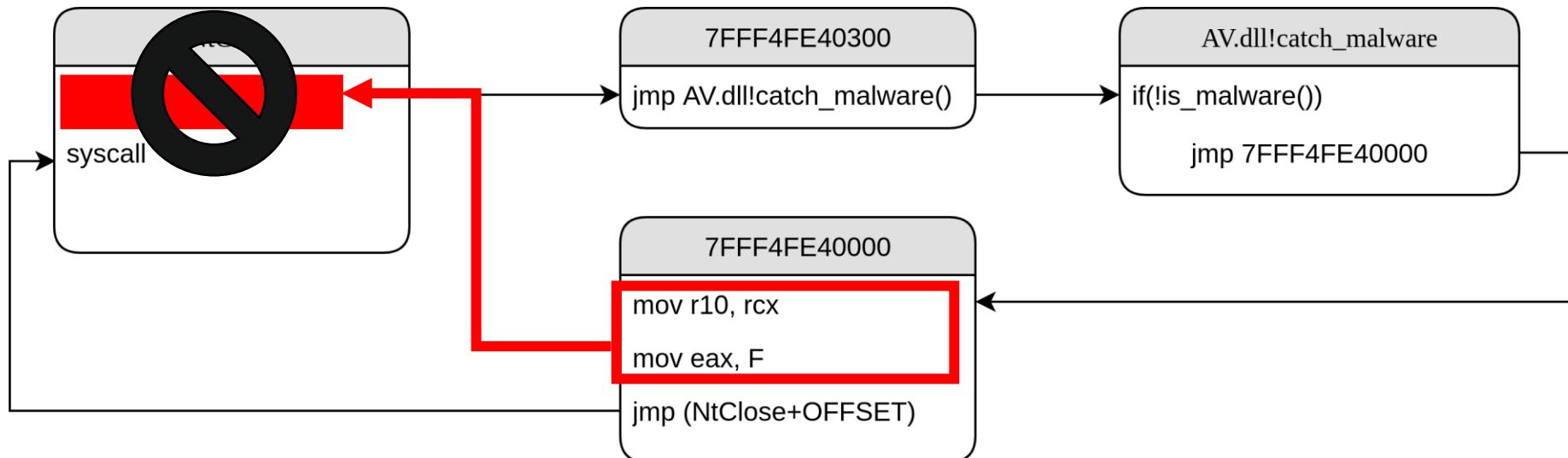
# UNHOOK IDEA

Overwrite the prologue with the original instructions



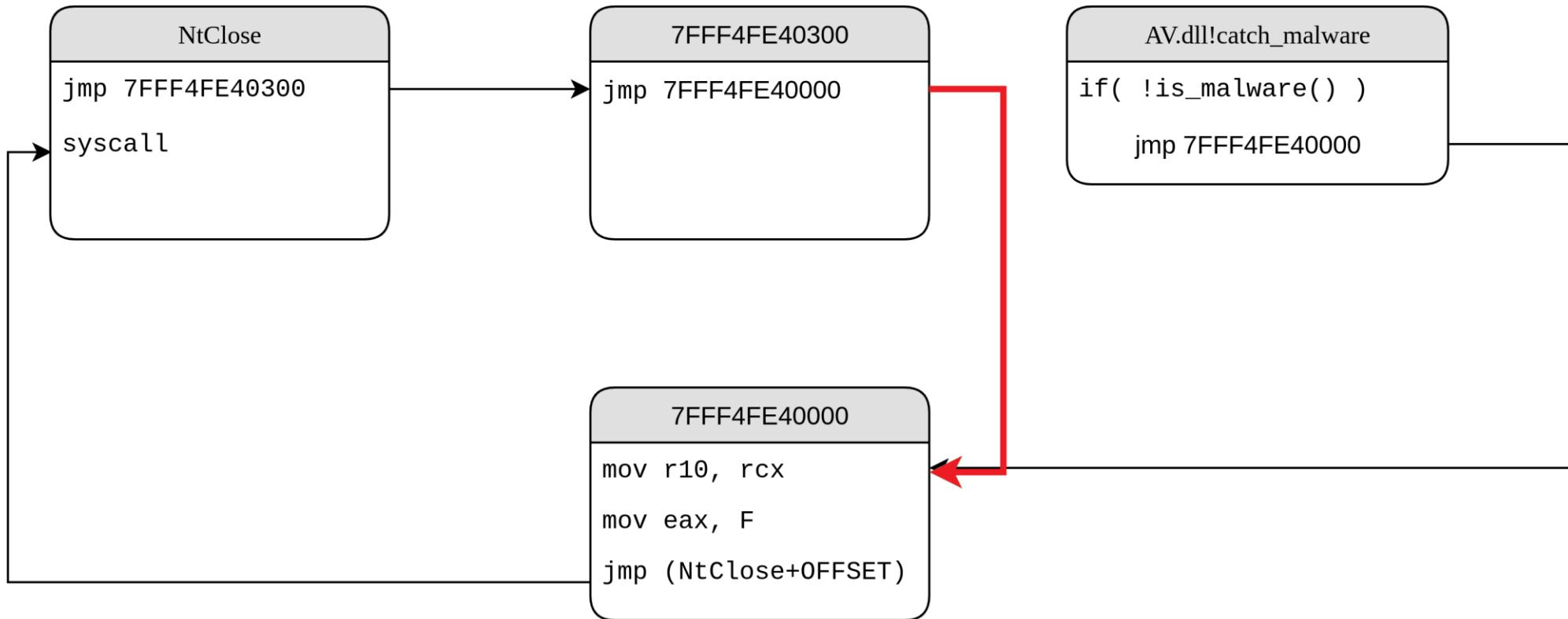
# UNHOOK IDEA

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

The screenshot shows three code snippets from a debugger, each with a downward arrow indicating flow:

- Top Snippet:** ntdll:00007FFDEEFAD3B0 ; FUNCTION CHUNK AT umppc16207:0000025726B69090 SIZE 00000016 BYTES  
ntdll:00007FFDEEFAD3B0 ; FUNCTION CHUNK AT ntdll:00007FFDEF02AC7B SIZE 0000000C BYTES  
ntdll:00007FFDEEFAD3B0  
ntdll:00007FFDEEFAD3B0 mov r10, rcx  
ntdll:00007FFDEEFAD3B3 jmp loc\_7FFDEF02AC7B  
ntdll:00007FFDEEFAD3B3 ntdll\_ZwAllocateVirtualMemory endp ; sp-analysis failed  
ntdll:00007FFDEEFAD3B3
- Middle Snippet:** ntdll:00007FFDEF02AC7B ; START OF FUNCTION CHUNK FOR ntdll\_ZwAllocateVirtualMemory  
ntdll:00007FFDEF02AC7B  
ntdll:00007FFDEF02AC7B loc\_7FFDEF02AC7B:  
ntdll:00007FFDEF02AC7B push rcx  
ntdll:00007FFDEF02AC7C push rcx  
ntdll:00007FFDEF02AC7D push rcx  
ntdll:00007FFDEF02AC7E push rcx  
ntdll:00007FFDEF02AC7F push rcx  
ntdll:00007FFDEF02AC80 push rcx  
ntdll:00007FFDEF02AC81 jmp cs:off\_7FFDEF02AC87  
ntdll:00007FFDEF02AC81 ; END OF FUNCTION CHUNK FOR ntdll\_ZwAllocateVirtualMemory
- Bottom Snippet:** umppc16207:0000025726B69090 ; START OF FUNCTION CHUNK FOR ntdll\_ZwAllocateVirtualMemory  
umppc16207:0000025726B69090  
umppc16207:0000025726B69090 loc\_25726B69090:  
umppc16207:0000025726B69090 mov r10, cs:qword\_25726B71BA8  
umppc16207:0000025726B69097 mov eax, cs:dword 25726B71DC8  
:F02AC7B: ntdll ZwAllocateVirtualMemory:loc (Synchronized with

# 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)**

```
00007FFC774ADA98 0F1F8400 00000000
00007FFC774ADAA0 <ntdll.NtProtectVirtualMemory> 4C:8BD1
00007FFC774ADAA3 B8 50000000
00007FFC774ADAA8 F60425 0803FE7F 01
--00007FFC774ADAB0 75 03
00007FFC774ADAB2 0F05
00007FFC774ADAB4 C3
--00007FFC774ADAB5 CD 2E
00007FFC774ADAB7 C3
00007FFC774ADAB8 0F1F8400 00000000

nop dword ptr ds:[rax+rax],eax
mov r10,rcx
mov eax,50
test byte ptr ds:[7FFE0308],1
jne ntdll.7FFC774ADAB5
syscall
ret
int 2E
ret
nop dword ptr ds:[rax+rax].eax
```

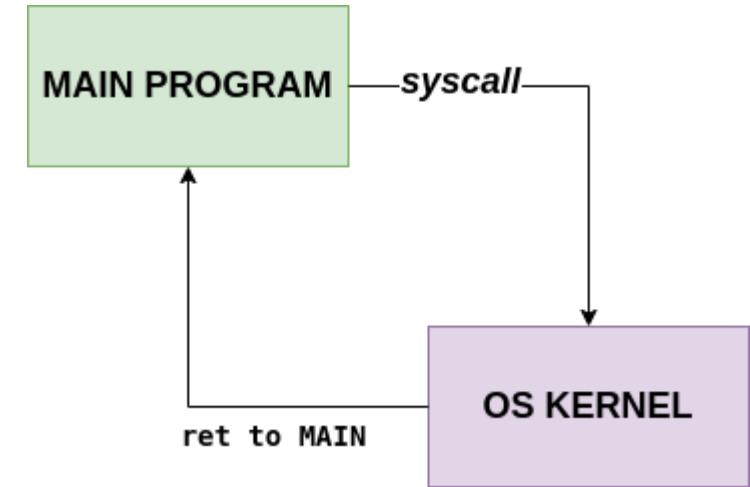
**Caveat:** ETW Ti can be used to detect memory permission changes from the kernel.

<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

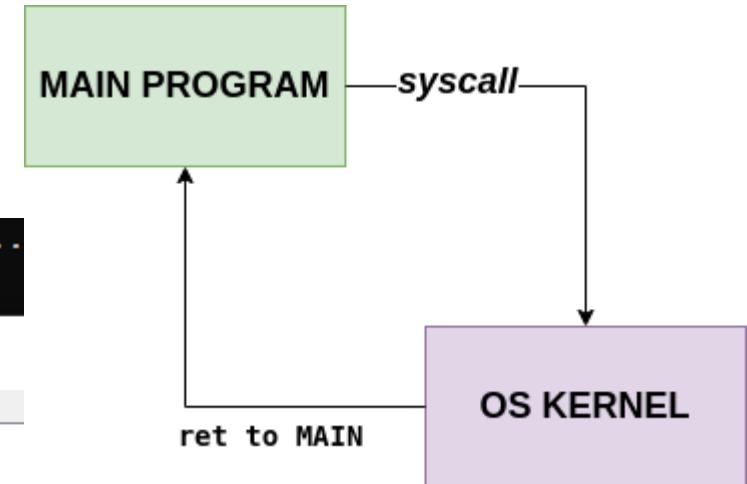
## Detection Strategies:

- Static Analysis
- Thread Correlation

```
00007FF7659E1D1A  
00007FF7659E1D1B <b  
00007FF7659E1D20  
00007FF7659E1D23  
00007FF7659E1D25  
00007FF7659E1D26  
00007FF7659E1D27  
<  
107FF7659E1D1B beacon23.  
07FF7659E1D1B beacon23.
```

```
Stack for thread 10752  
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 Beacon23.exe!NtWaitForSingleObject+0xa  
13 Beacon23.exe!every_evil_function+0xa0  
14 KERNEL32.DLL!BaseThreadInitThunk+0x14  
15 ntdll.dll!RtlUserThreadStart+0x21
```

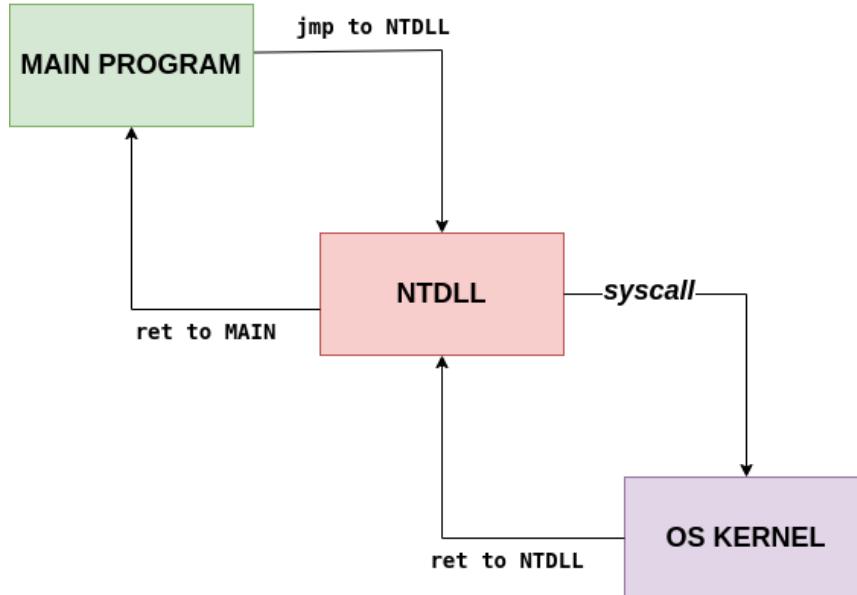
Calling NtWaitForSingleObject using Syswhispers (Direct Syscall)..  
Press any key to kill the thread and return



```
ret  
mov eax,4  
mov r10,rcx  
syscall  
ret  
int3  
int3
```

# Indirect System Calls

- No Syscall Instruction
- Nicer Thread Call Stack



Calling NtWaitForSingleObject using Syswhispers3 (Indirect Syscall)..  
Press any key to kill the thread and return

Stack for thread 5624 X

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!ZwWaitForSingleObject+0x14 (highlighted)  
13 Beacon23.exe!very\_evil\_function+0xf0 (highlighted)  
14 KERNEL32.DLL!BaseThreadInitThunk+0x14  
15 ntdll.dll!RtlUserThreadStart+0x21

Refresh Copy Copy All OK

# 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

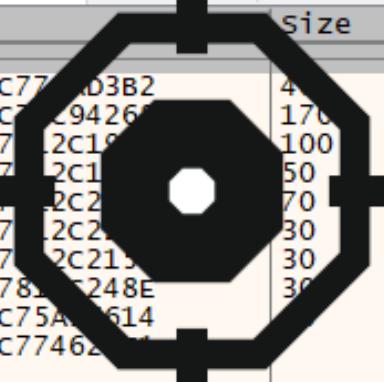
## User space View

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	00007FFC774AD3B2	4	ntdll.00007FFC774AD3B2	System
	000000CA004FF6A8	00007FF7812C19F7	00007FFC774C94268	170	kernelbase.00007FFC74C94268	User
	000000CA004FF818	00007FF7812C1890	00007FF7812C19F7	100	beacon23.very_evil_function+57	User
	000000CA004FF918	00007FF7812C23F9	00007FF7812C1890	50	beacon23.main+20	User
	000000CA004FF968	00007FF7812C229E	00007FF7812C23F9	70	beacon23.invoke_main+39	User
	000000CA004FF9D8	00007FF7812C215E	00007FF7812C229E	30	beacon23.__scrt_common_main_seh+12E	User
	000000CA004FFA08	00007FF7812C248E	00007FF7812C215E	30	beacon23.__scrt_common_main+E	User
	000000CA004FFA38	00007FFC75A37614	00007FF7812C248E	30	beacon23.mainCRTStartup+E	System
	000000CA004FFA68	00007FFC774626F1	00007FFC75A37614	1	kernel32.00007FFC75A37614	System
	000000CA004FFAE8	00000000000000000000	00007FFC774626F1	1	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

	TID	CPU	Cycles delta	Start address	Priority (symbolic)	
#	Name			Stack address	Return address	Frame address
0	Stack - thread 40216					
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	0x7ffb65795be	0x50174ff720
7	KernelBase.dll!SleepEx+0x9e			0x50174ff730	0x7ff7abc92e0d	0x50174ff7c0
8	ThreadStackSpoof.exe!MySleep+0x6d			0x50174ff7d0	0x7ffb7f74b60	0x50174ff8e0
9	kernel32.dll!CreateFileW			0x50174ff8f0	0x7ffb7f74b60	0x50174ff8e8
10	kernel32.dll!CreateFileW			0x50174ff8f8	0x7ffb7f74b60	0x50174ff8f0
11	kernel32.dll!CreateFileW			0x50174ff900	0x7ffb7f74b60	0x50174ff8f8
12	kernel32.dll!CreateFileW			0x50174ff908	0x7ffb7f74b60	0x50174ff900
13	kernel32.dll!CreateFileW			0x50174ff910	0x7ffb7f74b60	0x50174ff908
14	kernel32.dll!CreateFileW			0x50174ff918	0x7ffb7f74b60	0x50174ff910
15	kernel32.dll!CreateFileW			0x50174ff920	0x7ffb7f74b60	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

The screenshot shows a tweet from the user 'namazso' (@namazso). The tweet content is:

that's no stack spoofing whatsoever, that's an invalid unwind. you can achieve the same with a tweetable piece of code: `\*  
(void\*\*)\_AddressOfReturnAddress() = CreateFileW;`

The tweet was posted at 3:22 am · 27 Sep 2021.

Below the tweet, there is a table showing call stack information:

	TID	CPU	Cycles delta	Start address	Priority (symbolic)
Stack - thread 40216					
15	kernel32.dll!CreateFileW			0x50174ff920	0x7ffeb7f74b60
16	kernel32.dll!CreateFileW			0x50174ff928	0x50174ff918 0x50174ff920

# Call Stack Analysis – Direct Syscalls

The screenshot shows a search results page from elastic.co. The URL in the address bar is [elastic.co/security-labs/peeling-back-the-curtain-with-call-stacks](https://www.elastic.co/security-labs/peeling-back-the-curtain-with-call-stacks). The search query in the search bar is "rule.name : "Direct Syscall via Assembly Bytes"".

**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")))
```

The screenshot shows the Elastic Stack interface displaying search results for the query "rule.name : "Direct Syscall via Assembly Bytes"".

**Expanded document**

View: Single document Surround documents

Table JSON Copy to clipboard

156 hits

Document Field statistics

Columns 1 field sorted

rule.name : "Direct Syscall via Assembly Bytes"

pid: 9872, args.count: 1, thread: { Ext: { call\_stack\_summary: "Unbacked\kernel32.dll|ntdll.dll", call\_stack\_contains\_unbacked: true, call\_stack: [ { symbol\_info: "Unbacked+0xafde", callsite\_trailing\_bytes: "\x349ba82770e70000000b2fffff49c7c2f5bb6020e0a6fffff49bafe0503aa00000000e097ffff", protection: "RWX", callsite\_leading\_bytes: "\x73905484894c240848894542104cb94424184cb894c24204889ec284c89d1e88fffff4883c428488b4c", 2408488b55424104cb4424184cb4c24084889ca0f05" }, { symbol\_info: "Unbacked+0x9dd6", callsite\_trailing\_bytes: "\x41b9080000004c08442468488b054247048c7c1ffffffffe82213000041b9000000004d89e84889fa48c7", protection: "RWX", callsite\_leading\_bytes: "\x244848895c24488b84240804000089442388b8424f803000089442388c7442428000000088c74424", 20000000041bfffff1f041b0ffff1f00e86513000" } ] }, protection: "RWX", callsite\_leading\_bytes: "\x73905484894c240848894542104cb94424184cb894c24204889ec284c89d1e88fffff4883c428488b4c", 2408488b55424104cb4424184cb4c24084889ca0f05" }, { symbol\_info: "Unbacked+0x9dd6", callsite\_trailing\_bytes: "\x41b9080000004c08442468488b054247048c7c1ffffffffe82213000041b9000000004d89e84889fa48c7", protection: "RWX", callsite\_leading\_bytes: "\x244848895c24488b84240804000089442388b8424f803000089442388c7442428000000088c74424", 20000000041bfffff1f041b0ffff1f00e86513000" } ] }

## 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:

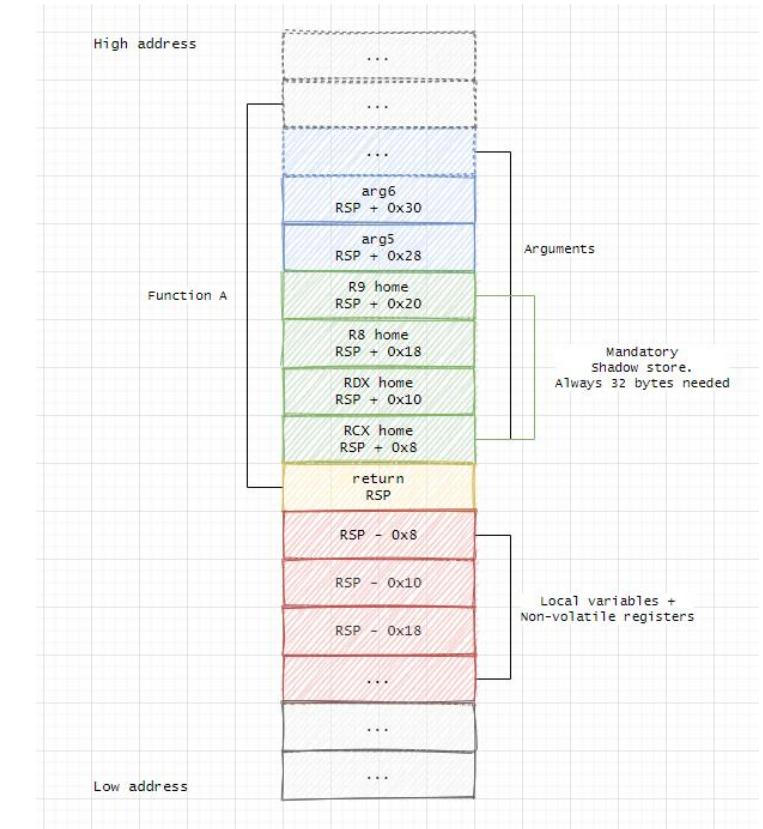
- Conceal origin of the call
- Make the call look legit

# 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

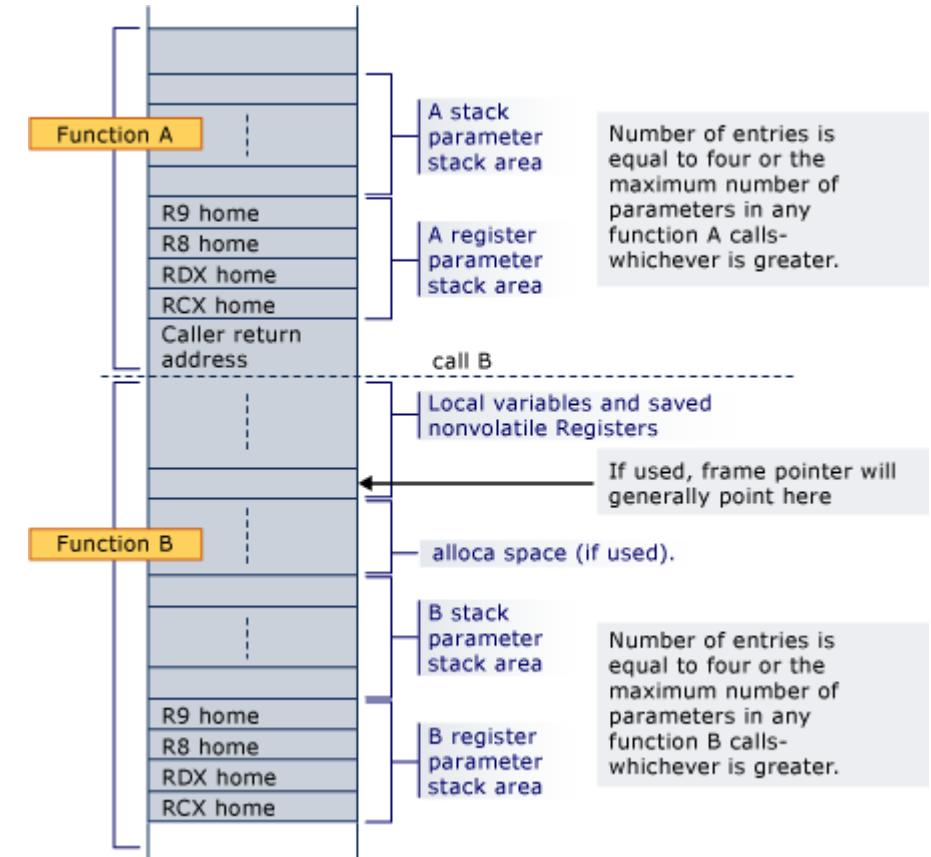
MSDN Documentation: <https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170>



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

# 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**

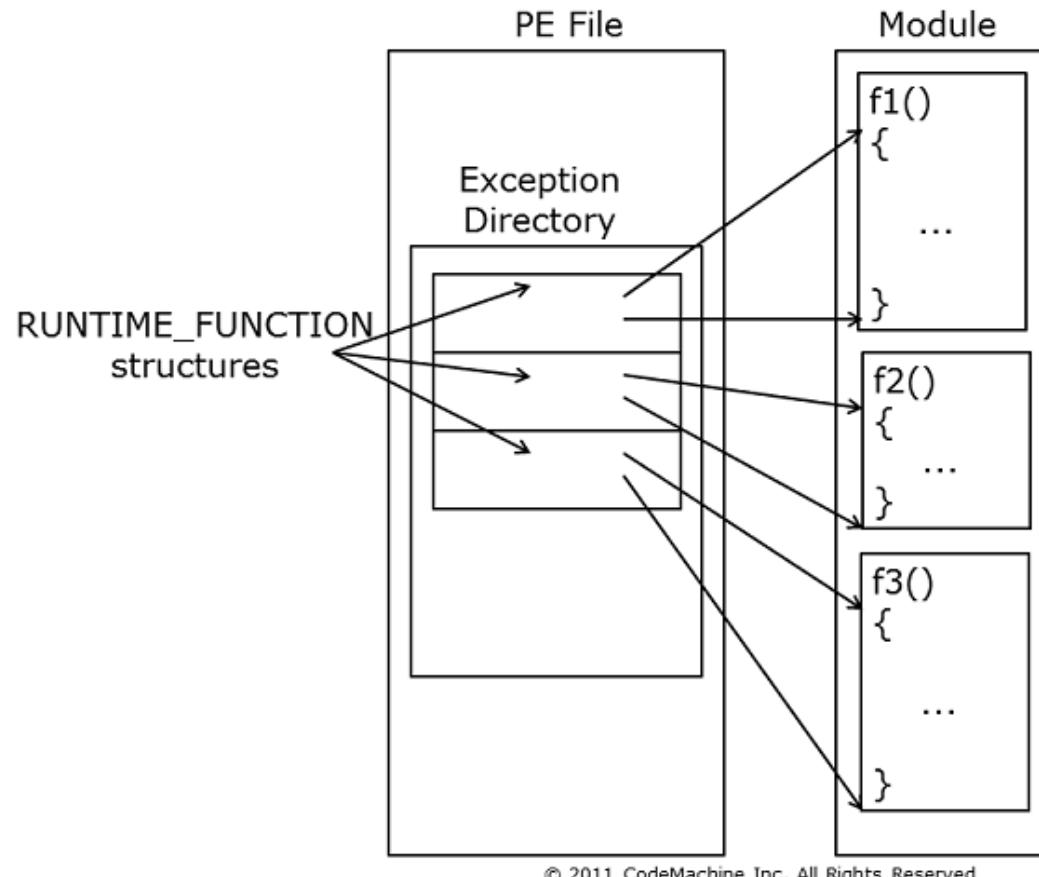
The following table describes how each register is used across function calls:

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



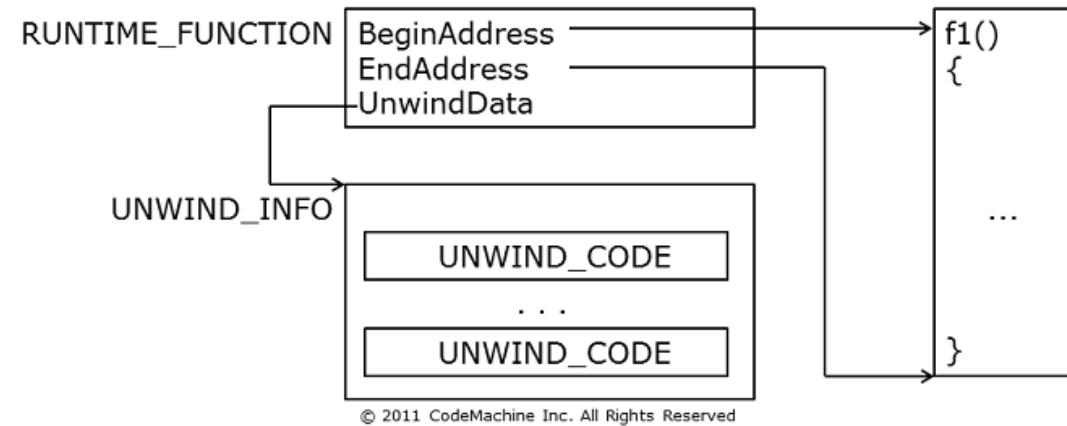
© 2011 CodeMachine Inc. All Rights Reserved

More details:

[https://codemachine.com/articles/x64\\_deep\\_dive.html](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

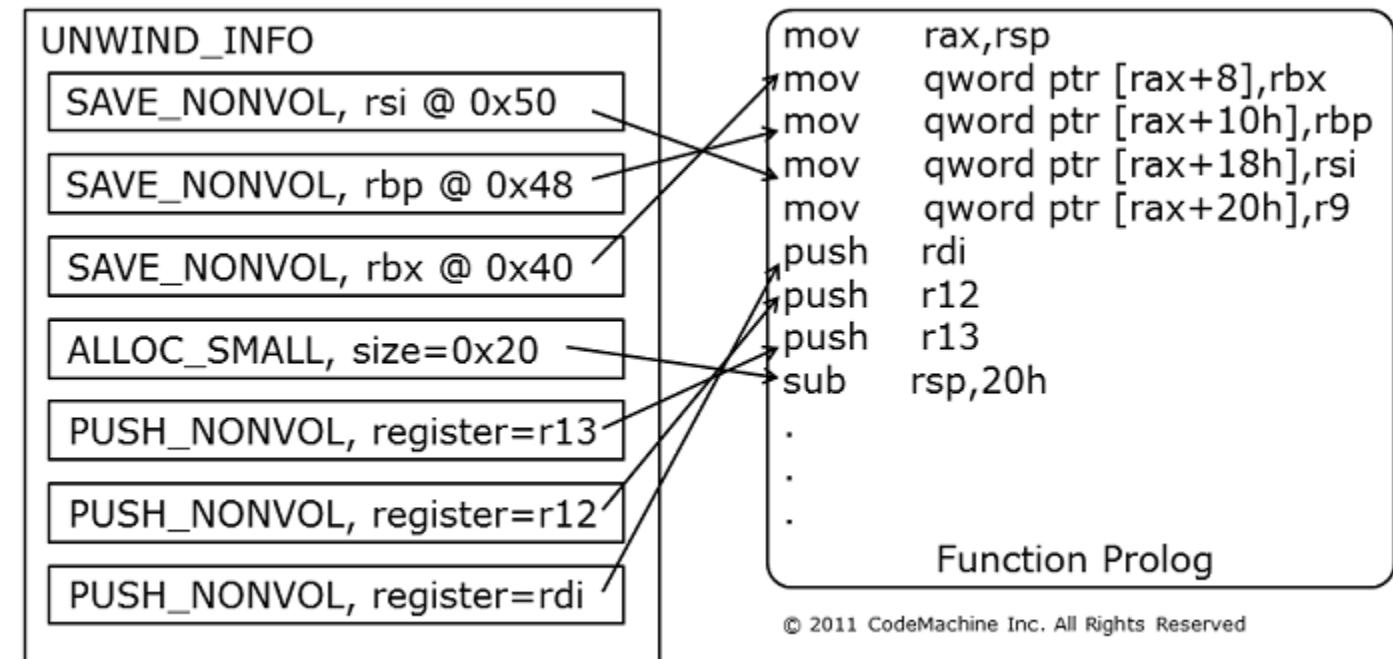


More details:

[https://codemachine.com/articles/x64\\_deep\\_dive.html](https://codemachine.com/articles/x64_deep_dive.html)

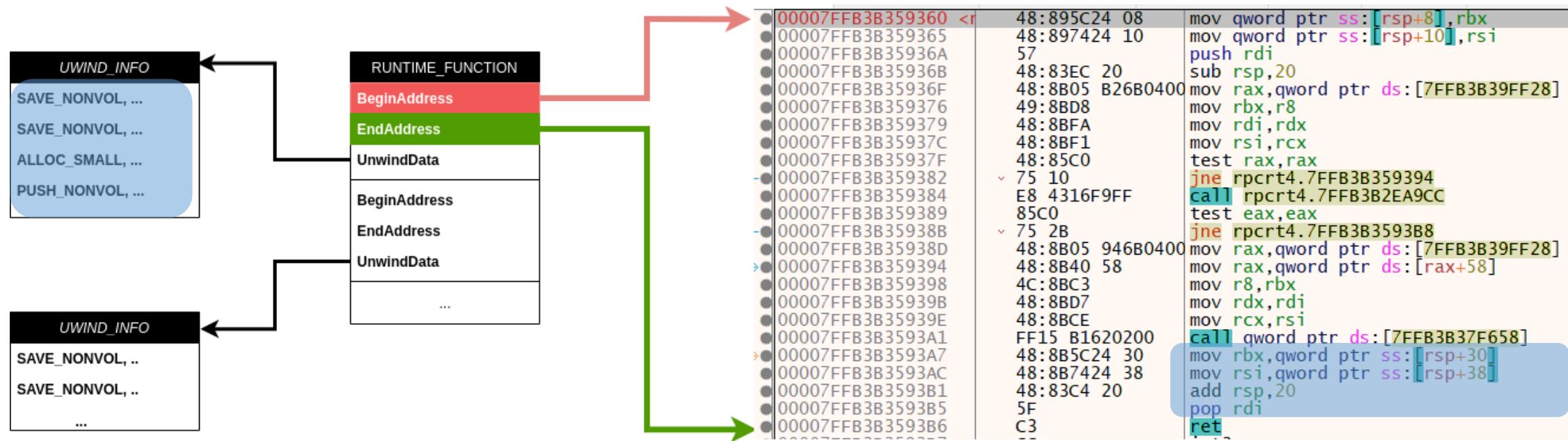
# Stack Unwinding 101

- `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](https://codemachine.com/articles/x64_deep_dive.html)

# Stack Unwinding 101



# It's all about perception...

- Retrieving the call stack is an indirect operation
- Interpretation of the stack content relies on a data structure that contains the required 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 indirect operation
- Interpretation of the stack content relies on a data structure that contains the required 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..

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

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. artificial) 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



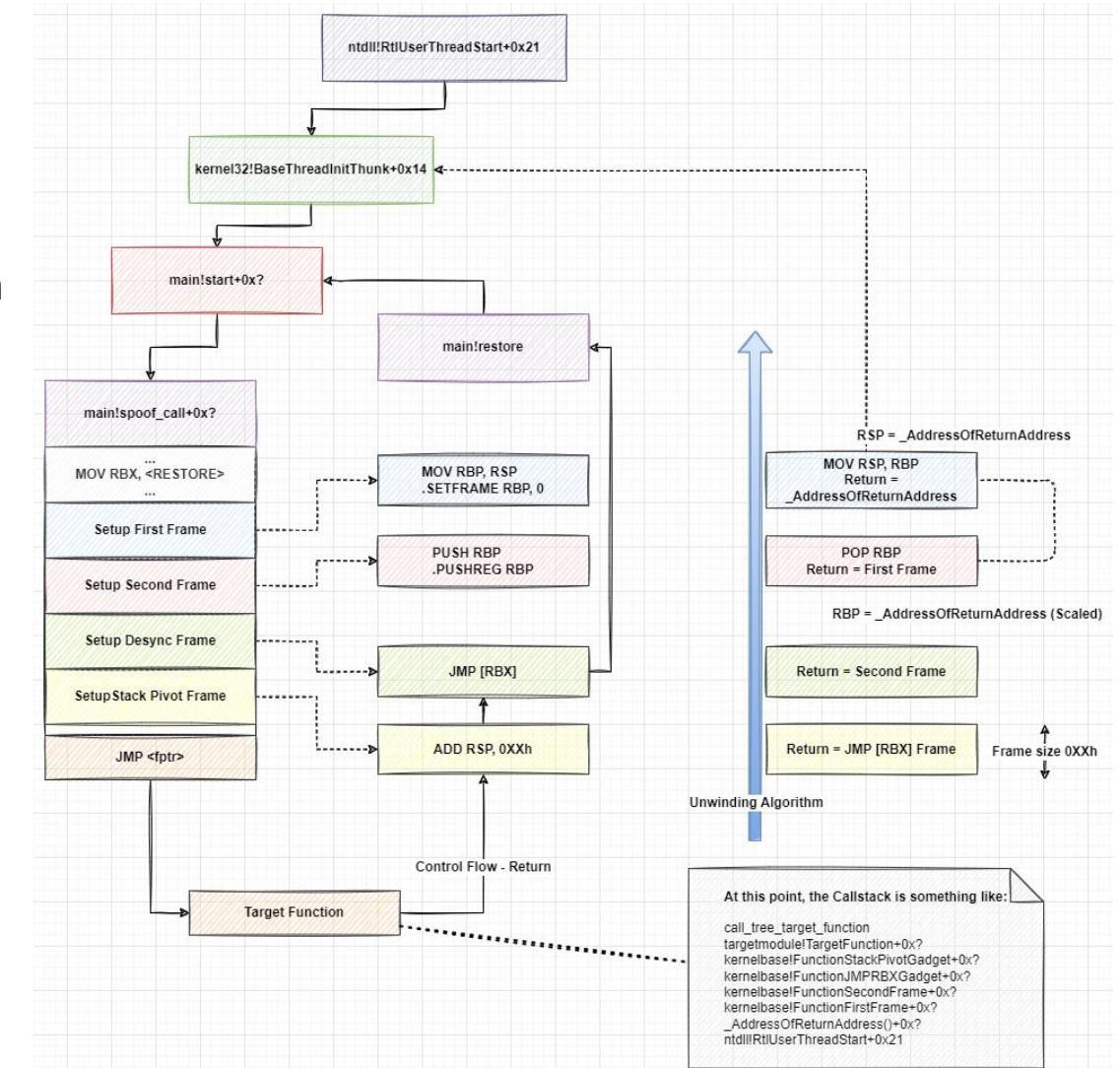
- Find frames that involve usage of stack frame, to allow RSP manipulation
  - allow to put an arbitrary pointer in RSP when the stack is inspected by the unwinding algorithm (more on this later...)

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

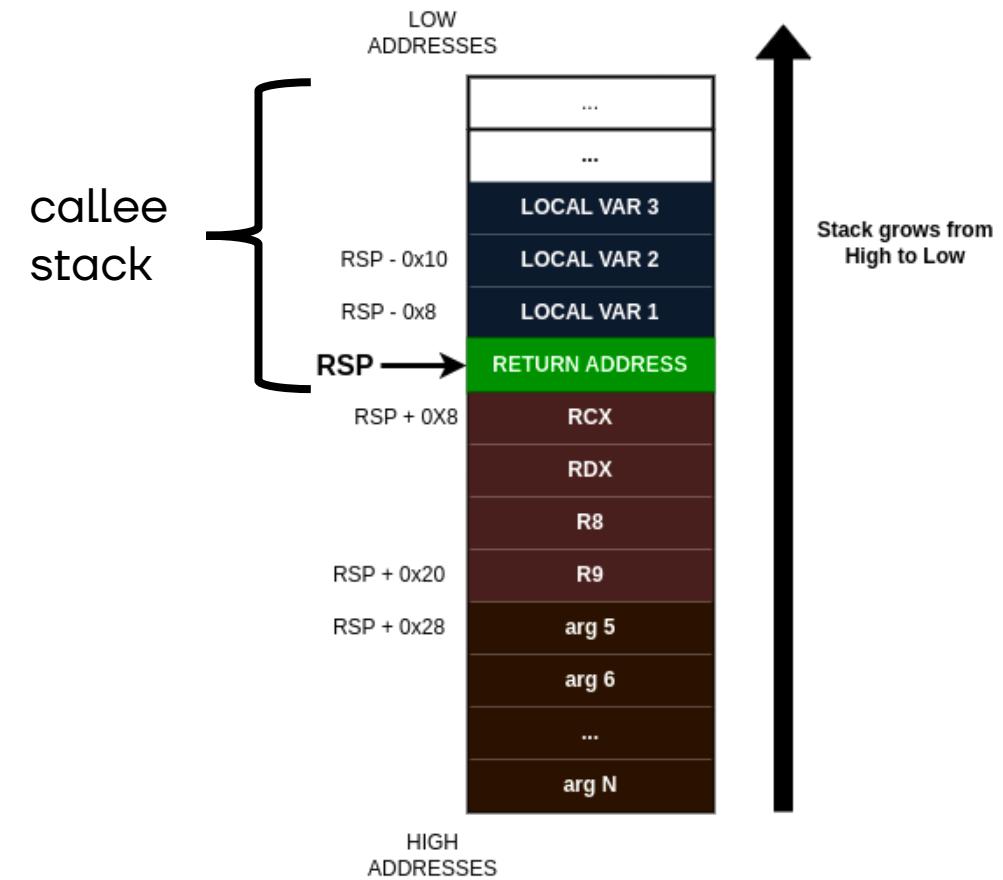


Source: [https://klezvirus.github.io/RedTeaming/AV\\_Evasion/StackSpoofing/](https://klezvirus.github.io/RedTeaming/AV_Evasion/StackSpoofing/)

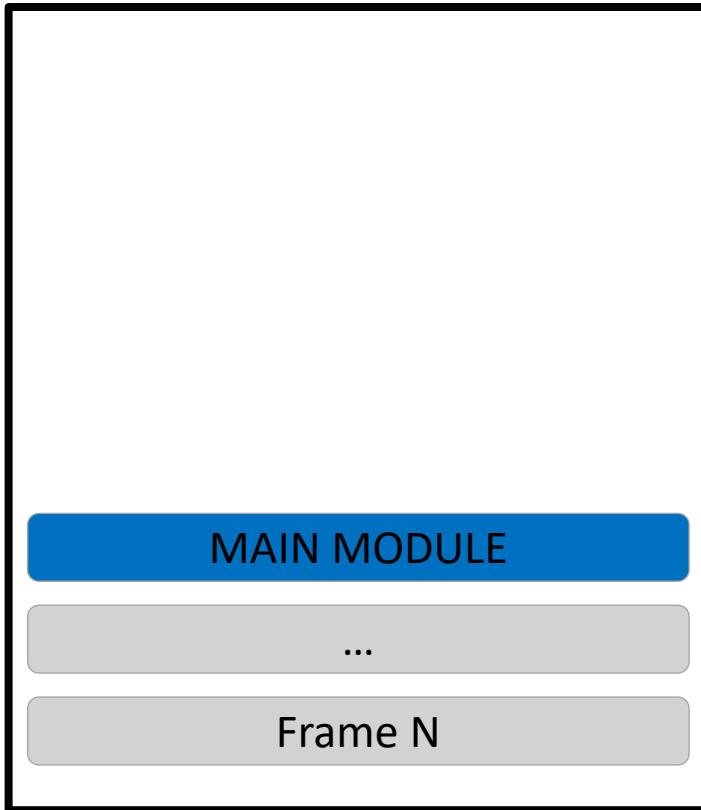
# Construct the Spoofed Stack

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

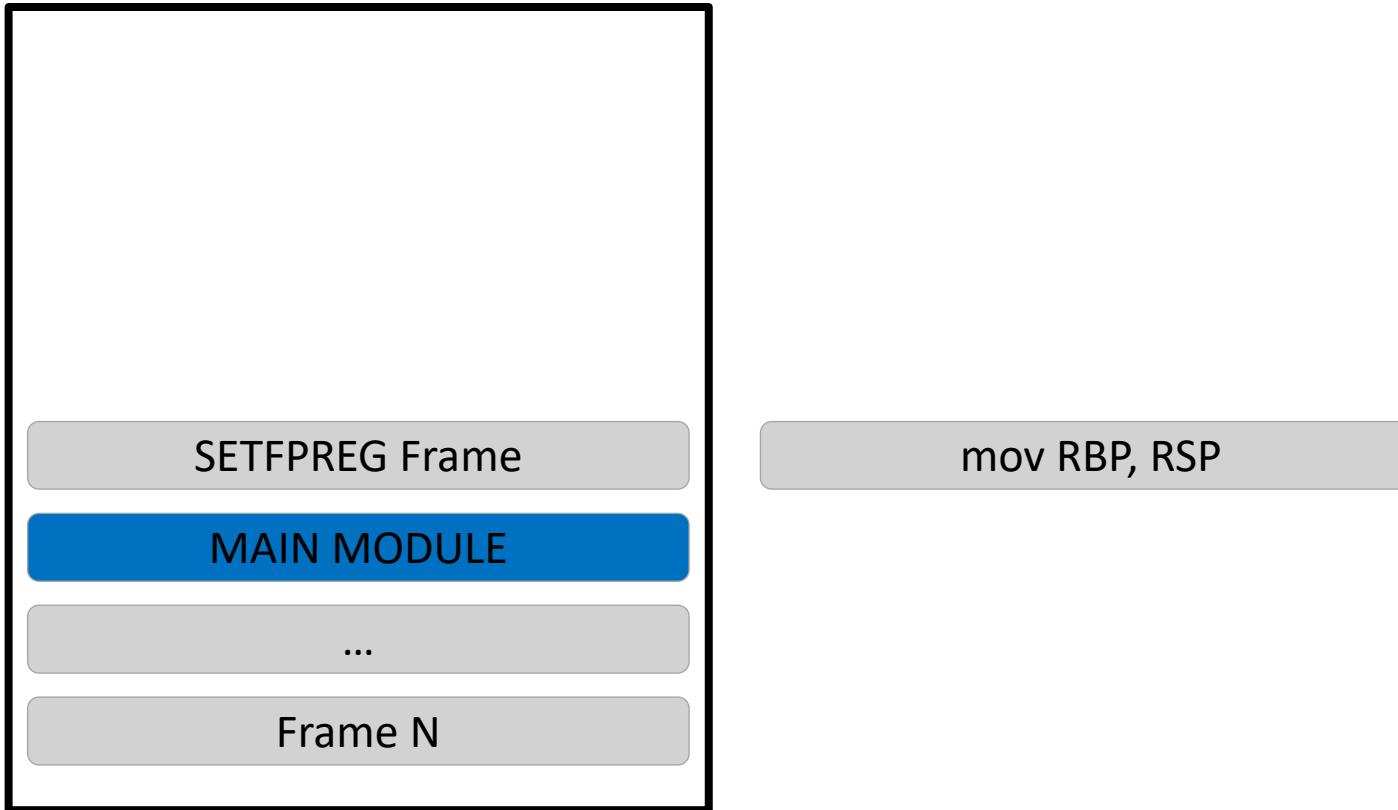
- Push parameters
- Push return address
- Allocate stack space (i.e. `sub rsp , X`)



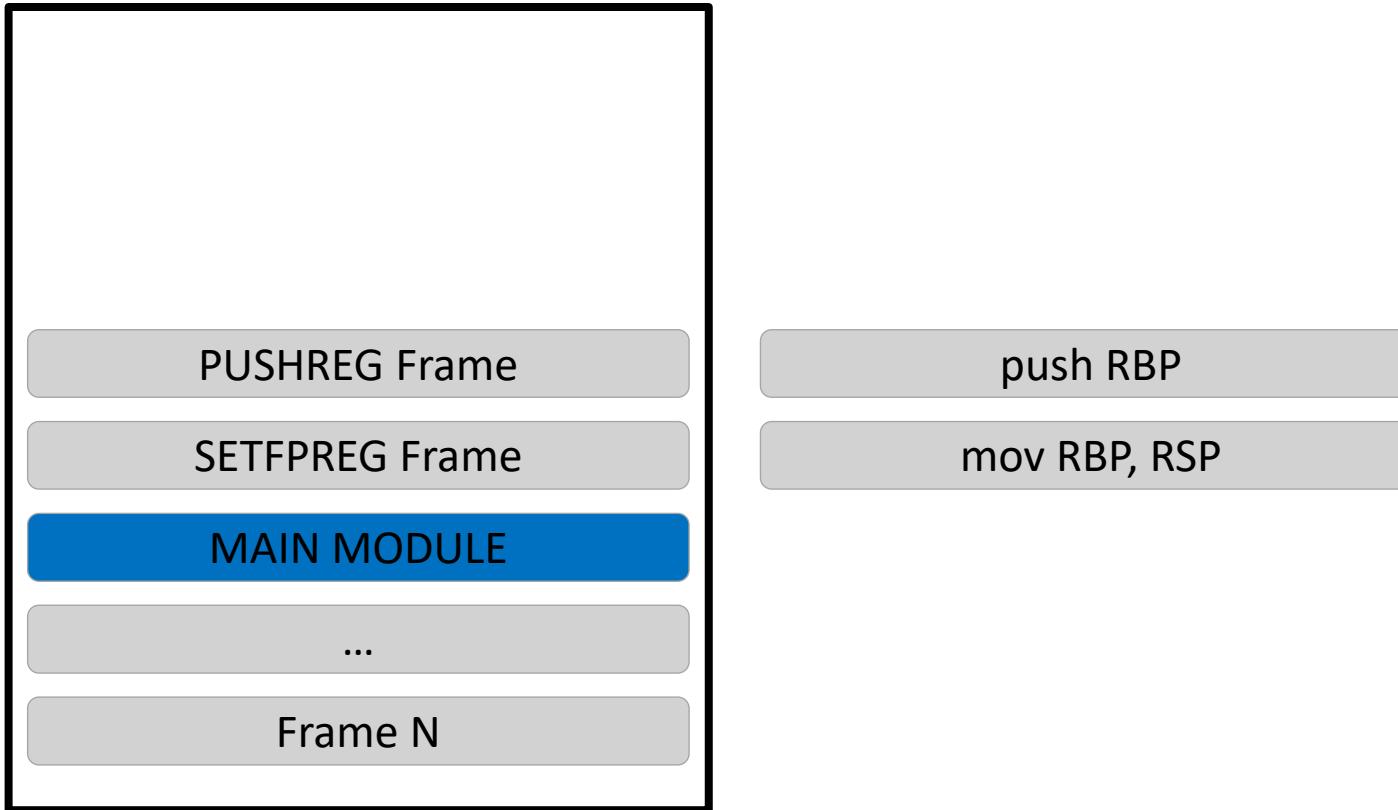
# Construct the Spoofed Stack



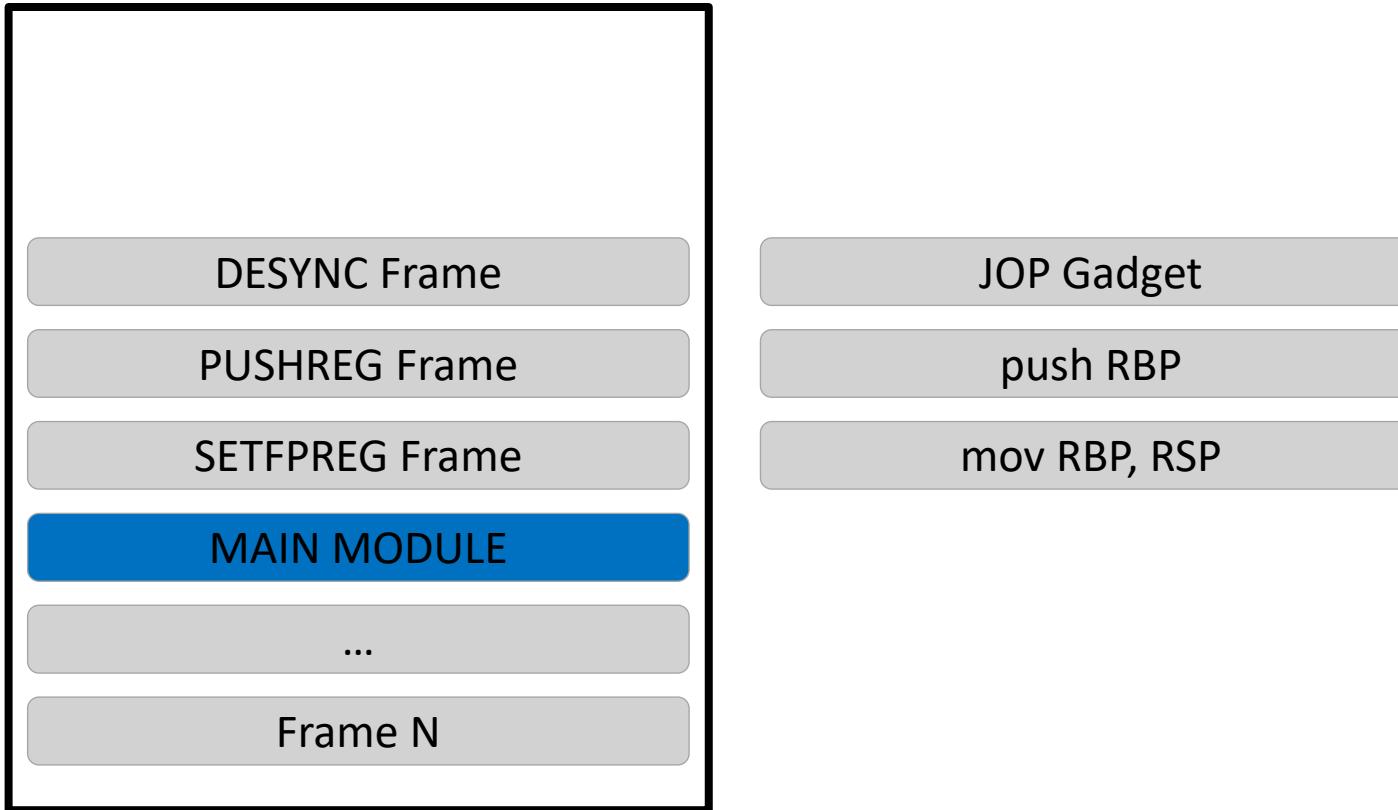
# Construct the Spoofed Stack



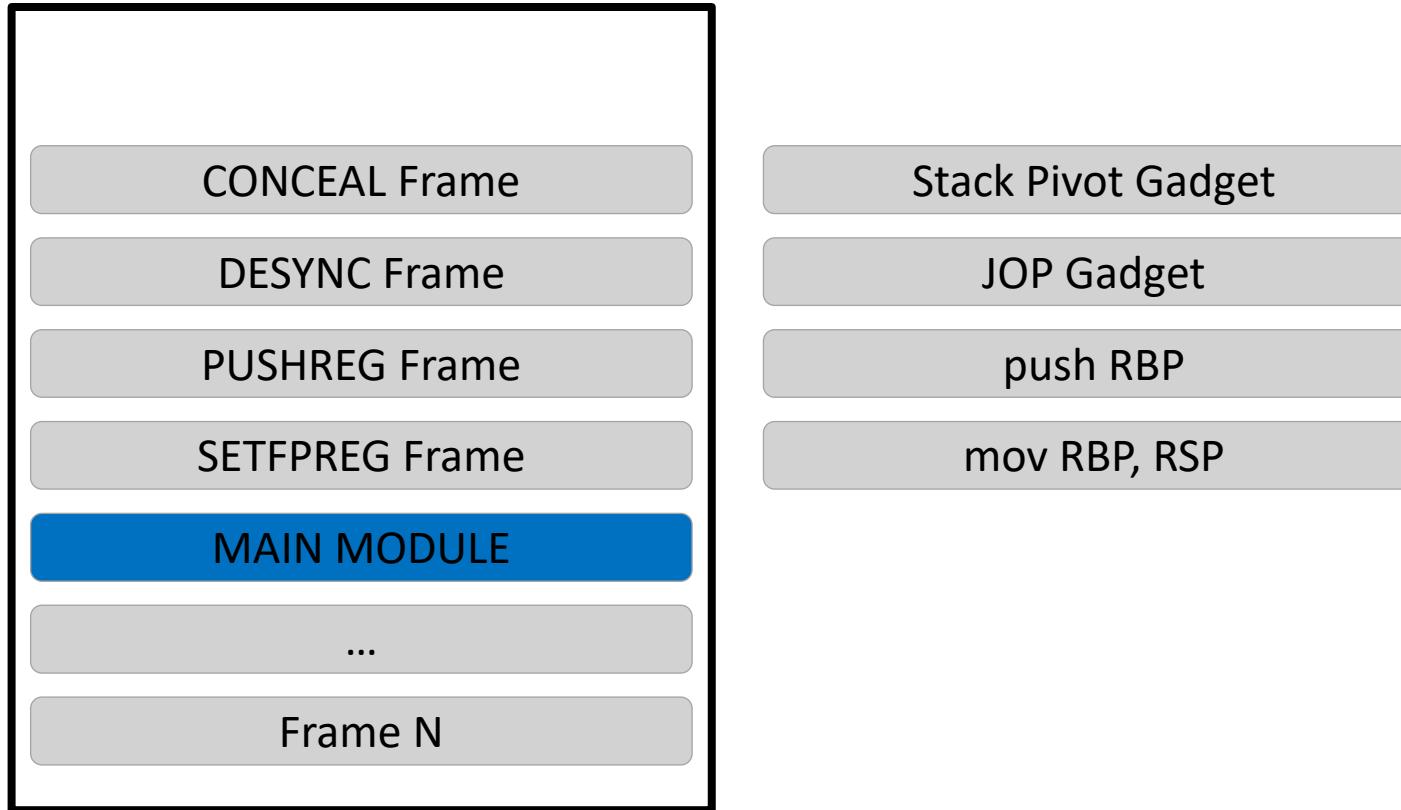
# Construct the Spoofed Stack



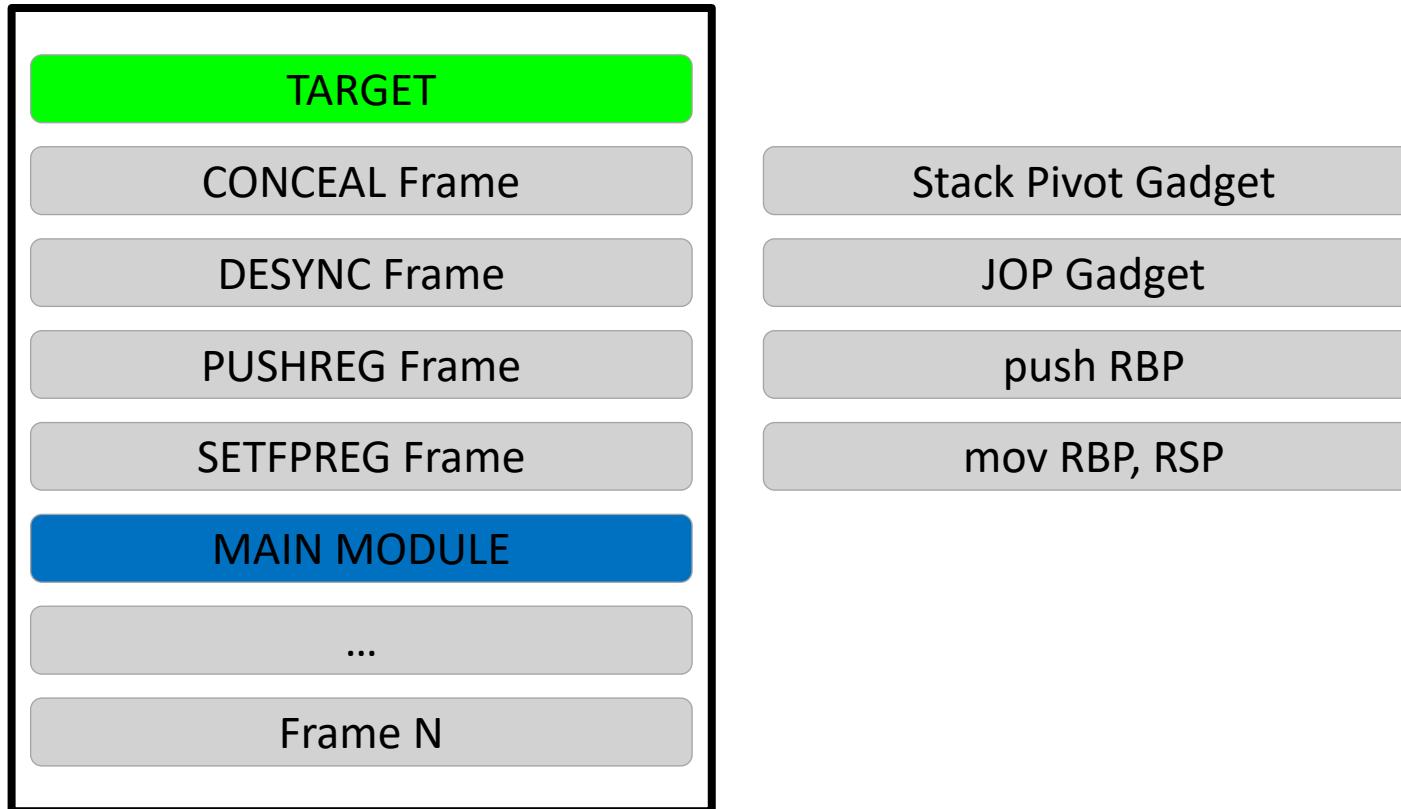
# Construct the Spoofed Stack



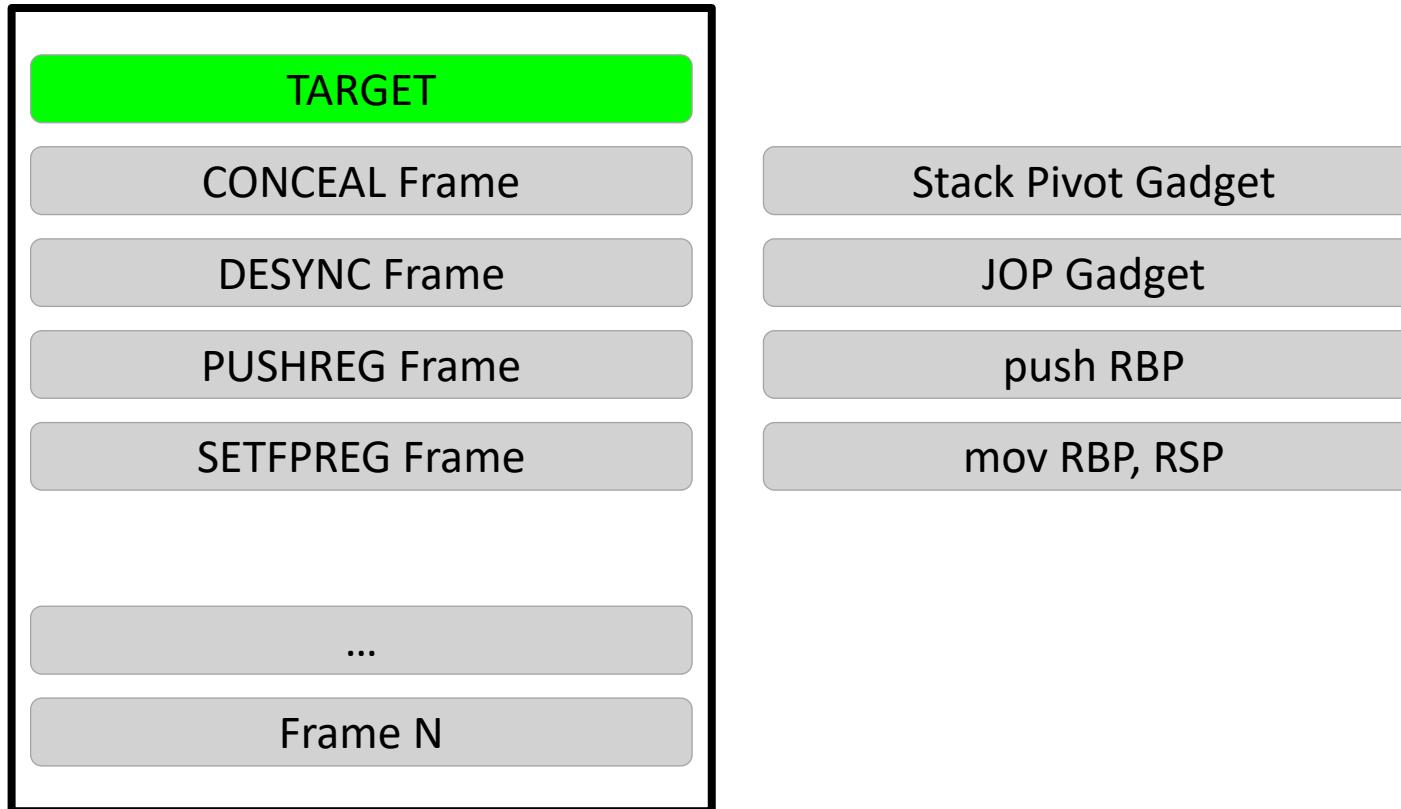
# Construct the Spoofed Stack



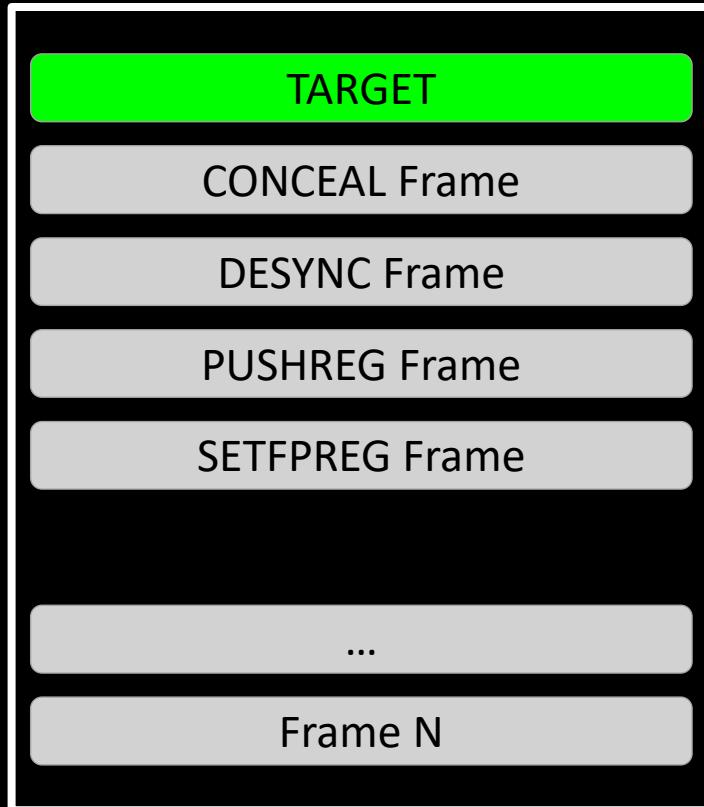
# Construct the Spoofed Stack



# Construct the Spoofed Stack

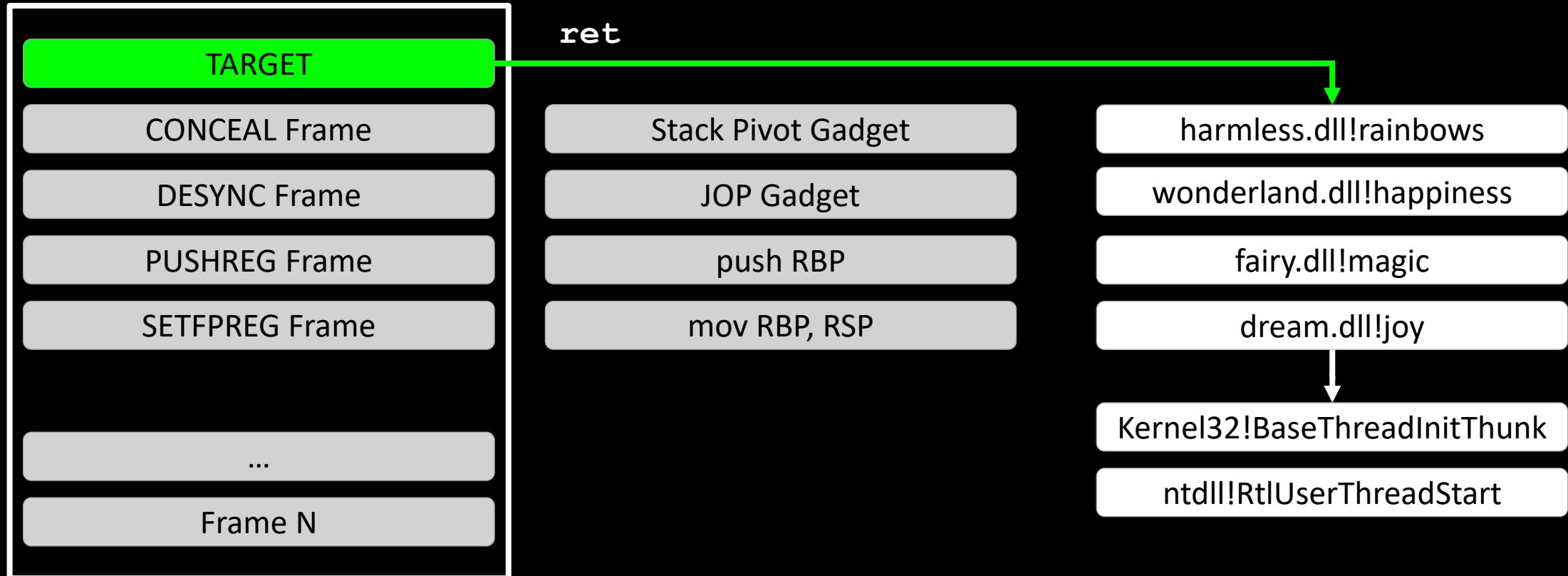


# It's all about perception...

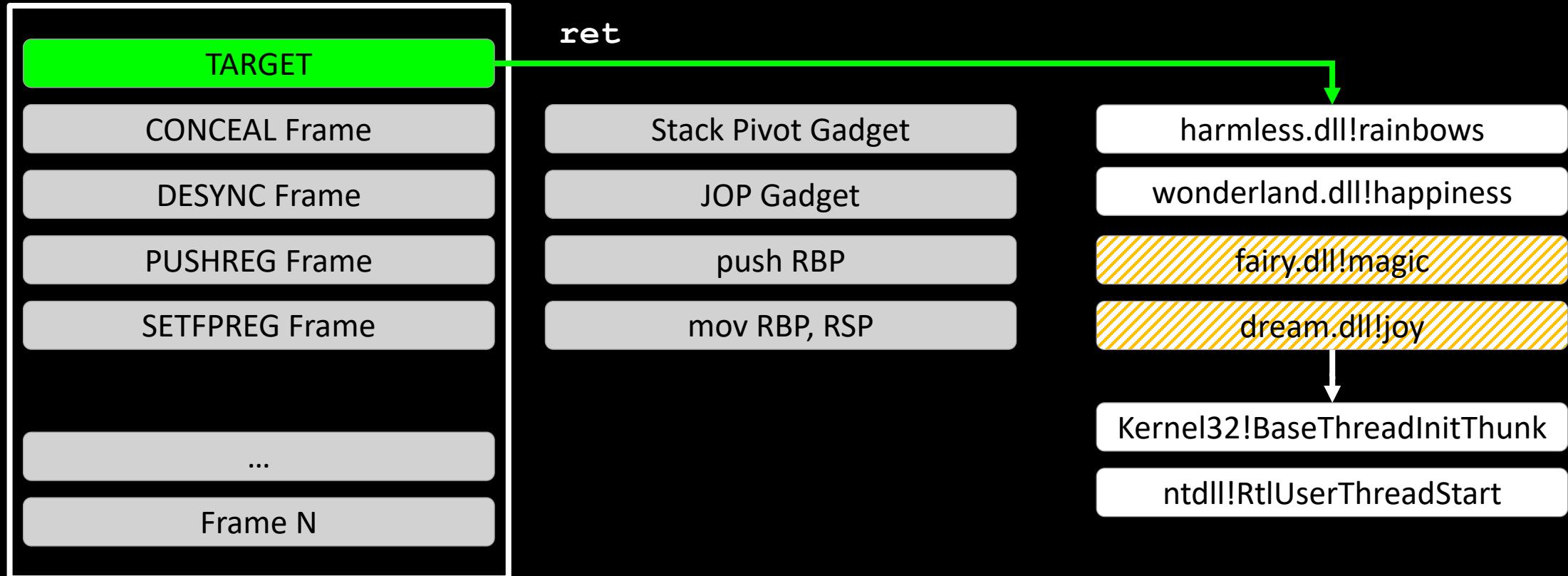


Stack Pivot Gadget  
JOP Gadget  
push RBP  
mov RBP, RSP

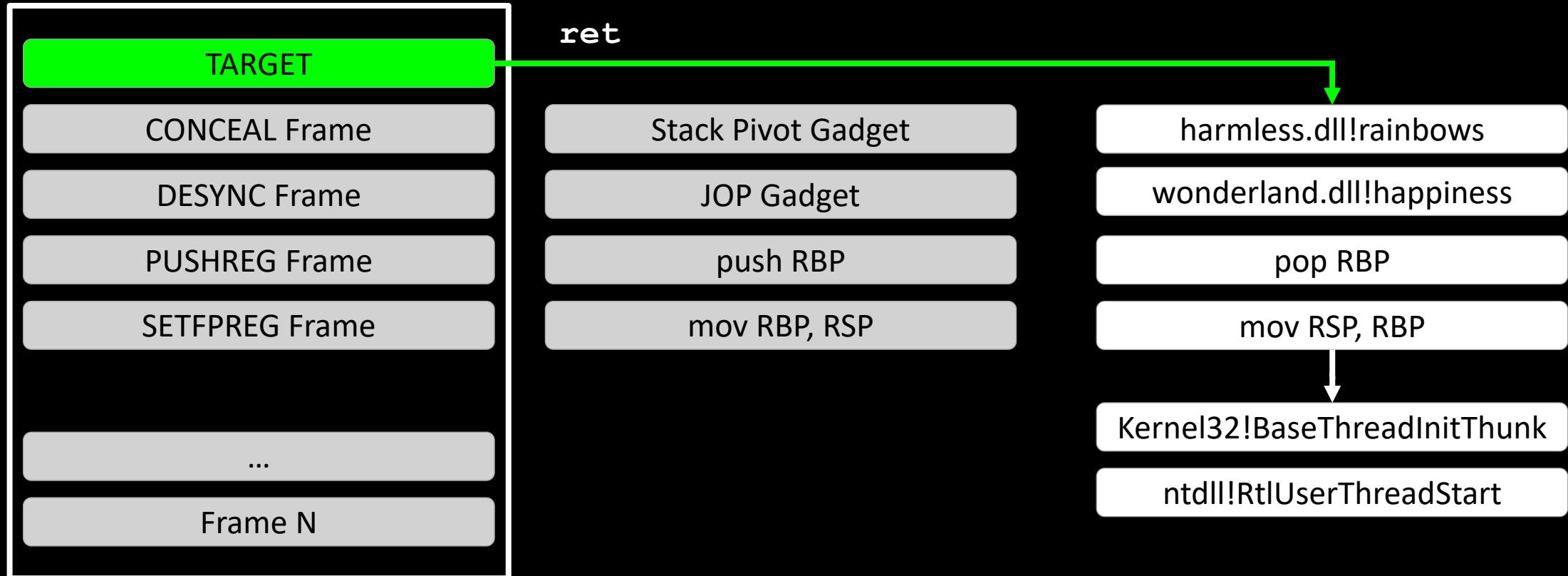
# It's all about perception...



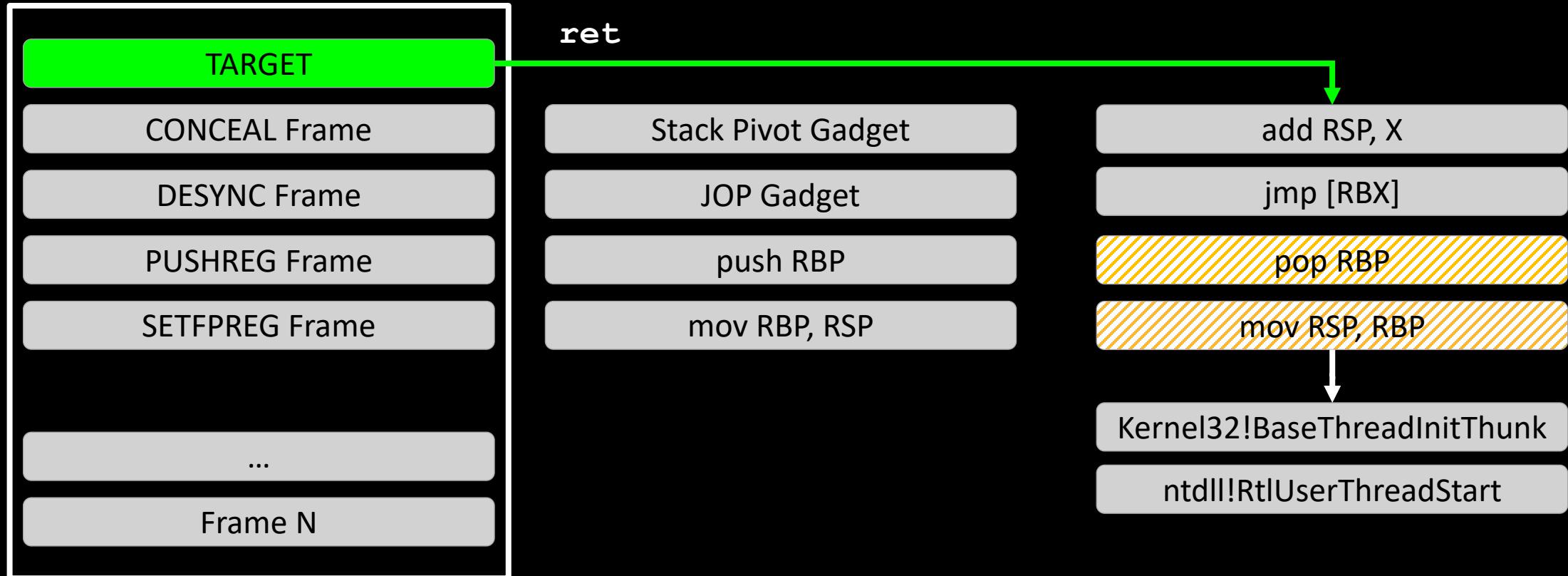
# It's all about perception...



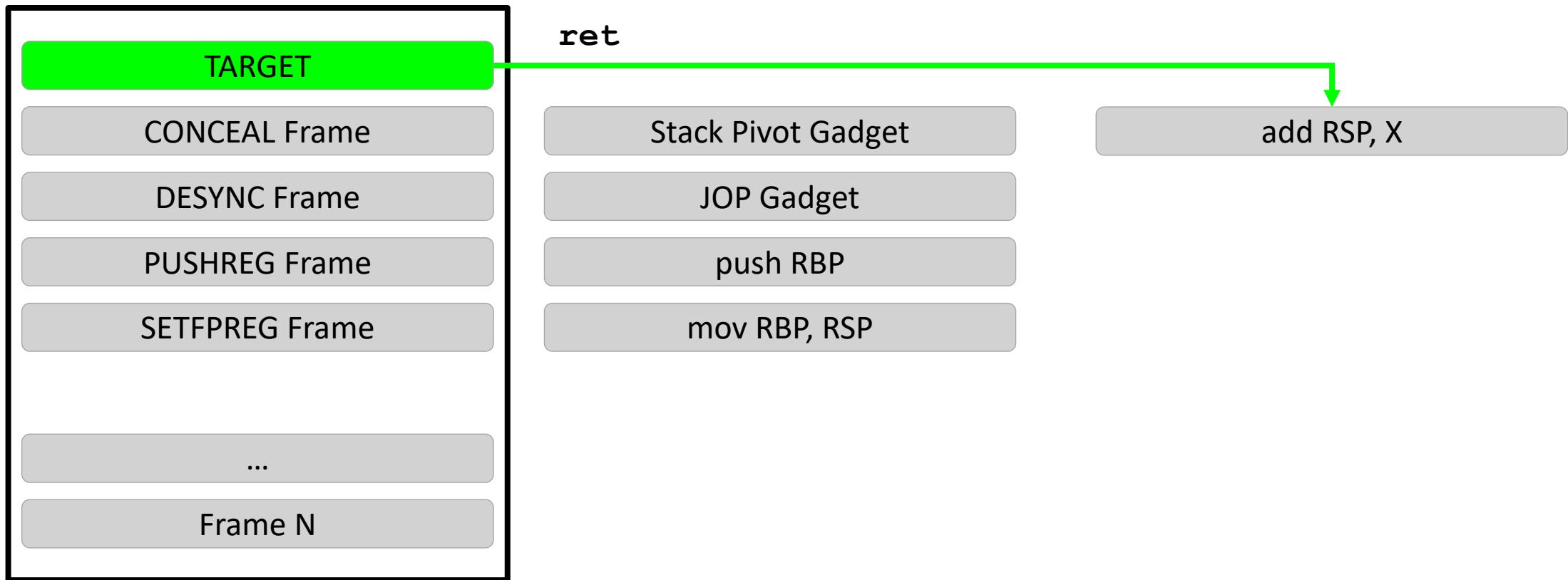
# It's all about perception...



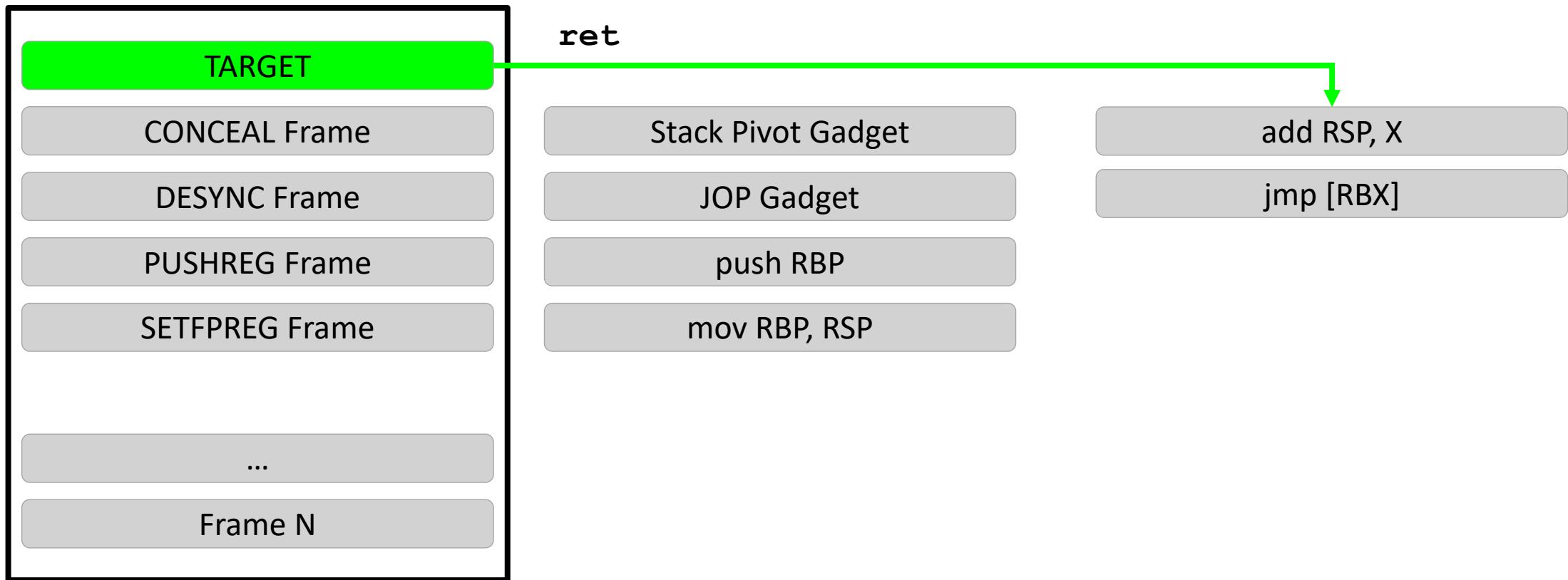
# It's all about perception...



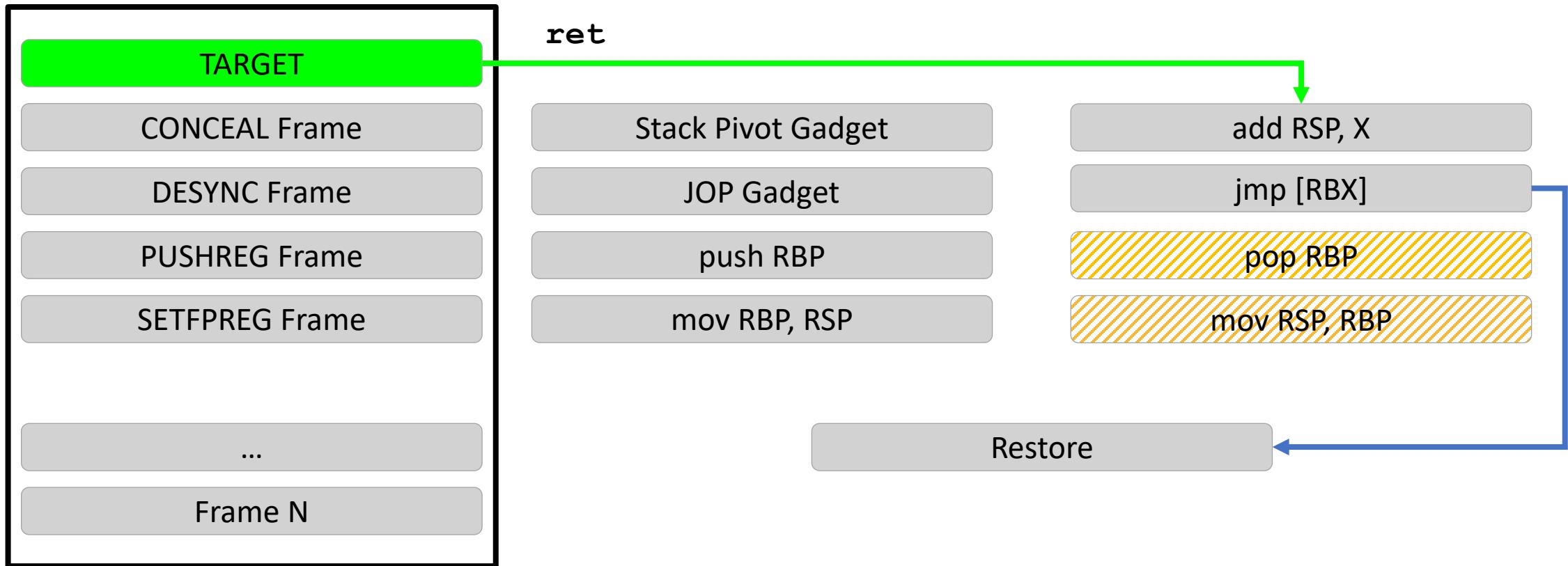
# What Actually Happens



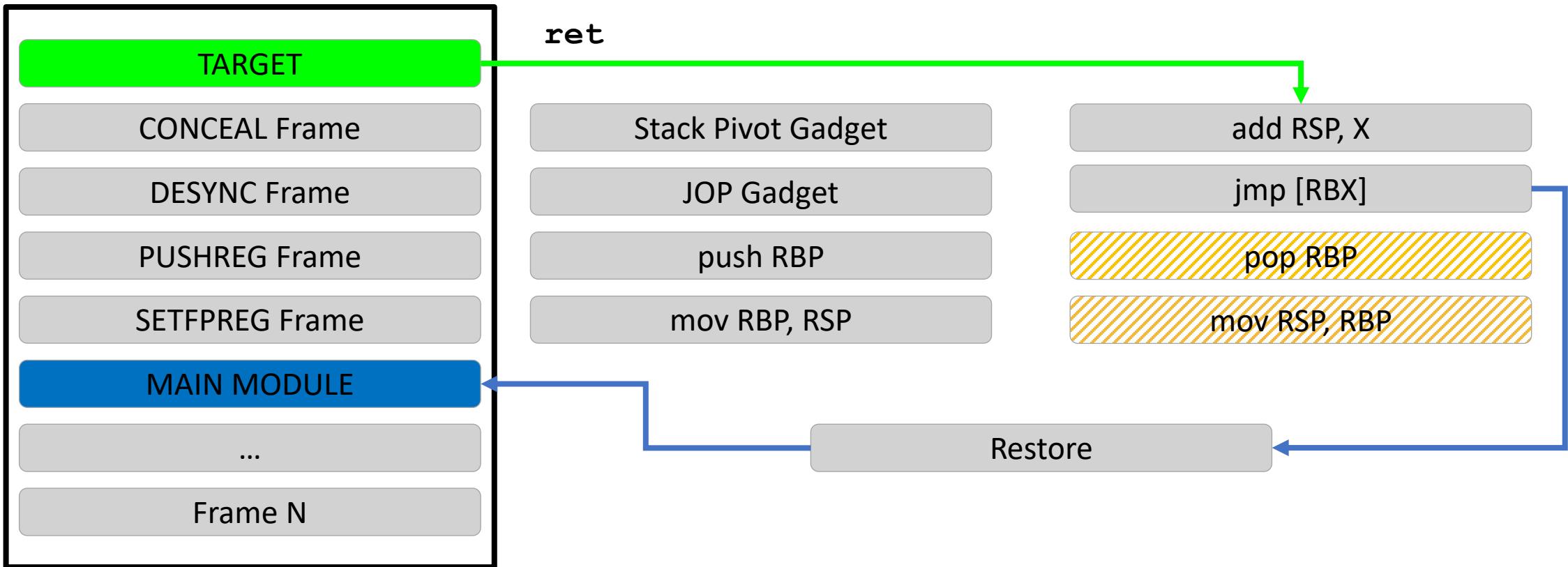
# What Actually Happens



# What Actually Happens



# What Actually Happens



# Credits and References

[https://codemachine.com/articles/x64\\_deep\\_dive.html](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://klezvirus.github.io/RedTeaming/AV_Evasion/StackSpoofing)

<https://github.com/JLospinoso/gargoyle>

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

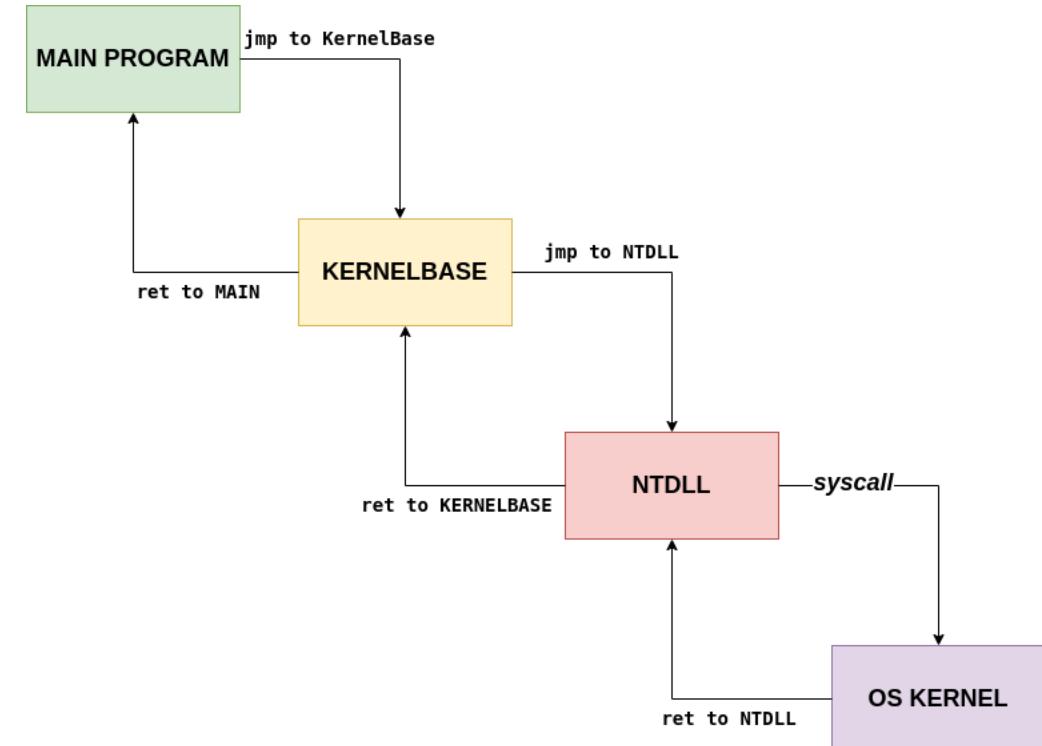
<https://github.com/kyleavery/AceLdr>

# 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

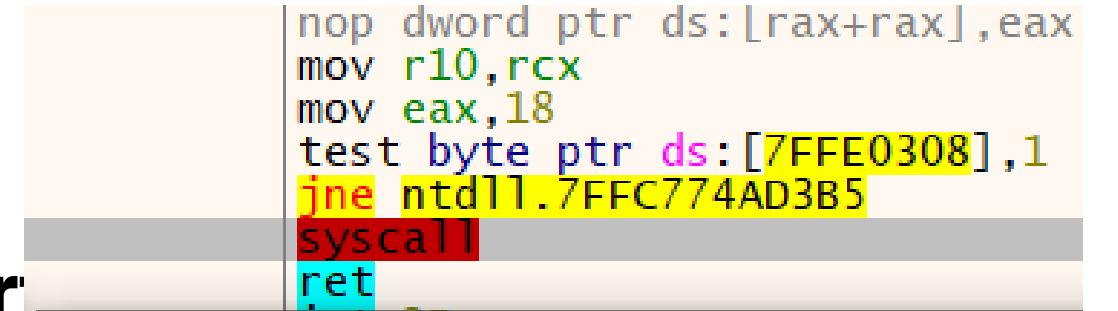
Call to WaitForSingleObject **API**

# Call Stack Trace

## Call to Vir

```
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
```



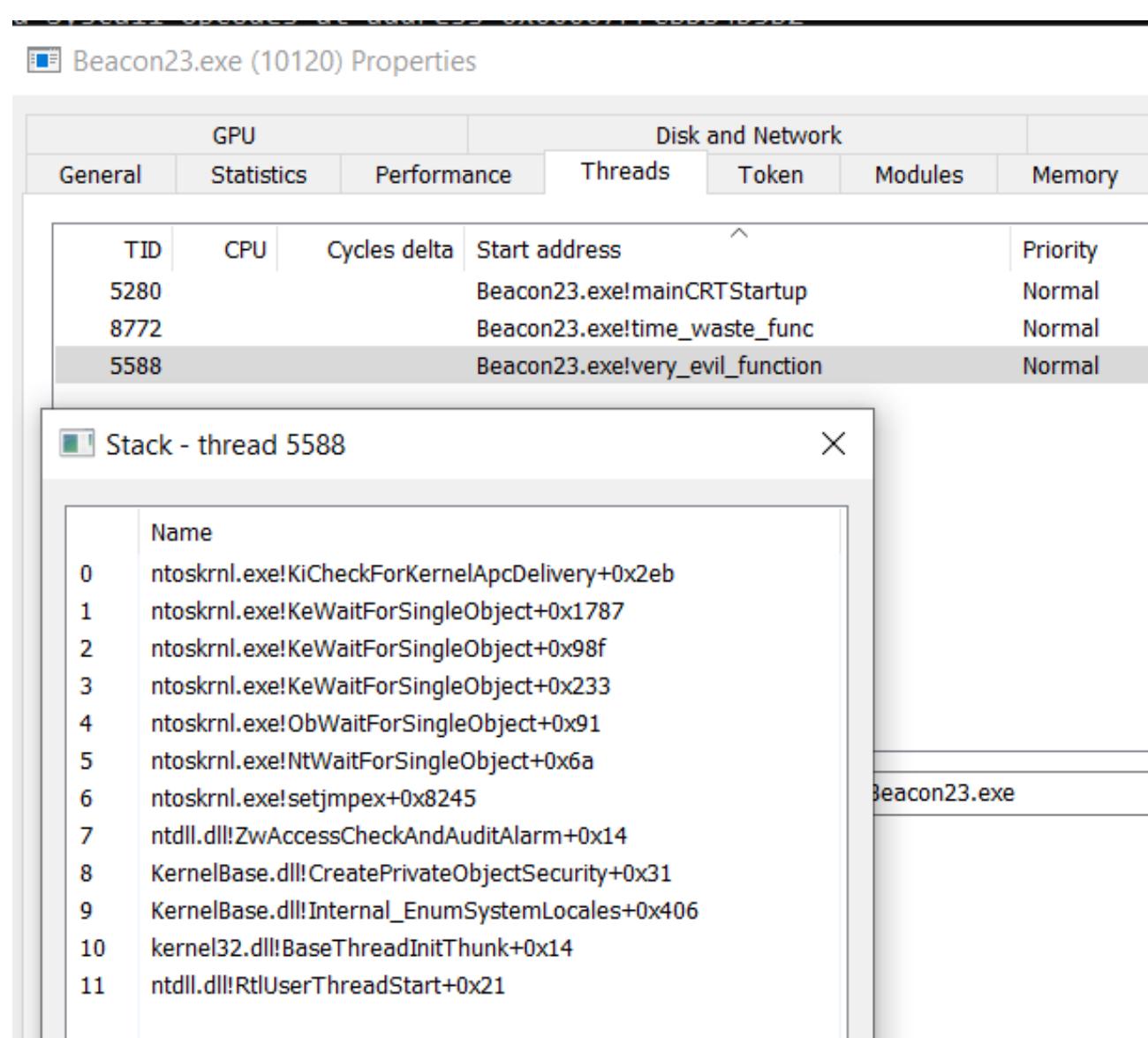
The screenshot shows assembly code in a debugger. The code includes instructions like `nop dword ptr ds:[rax+rax],eax`, `mov r10,rcx`, `mov eax,18`, `test byte ptr ds:[7FFE0308],1`, `jne ntdll!7FFC774AD3B5` (which is highlighted in yellow), `syscall` (highlighted in red), and `ret` (highlighted in cyan).

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

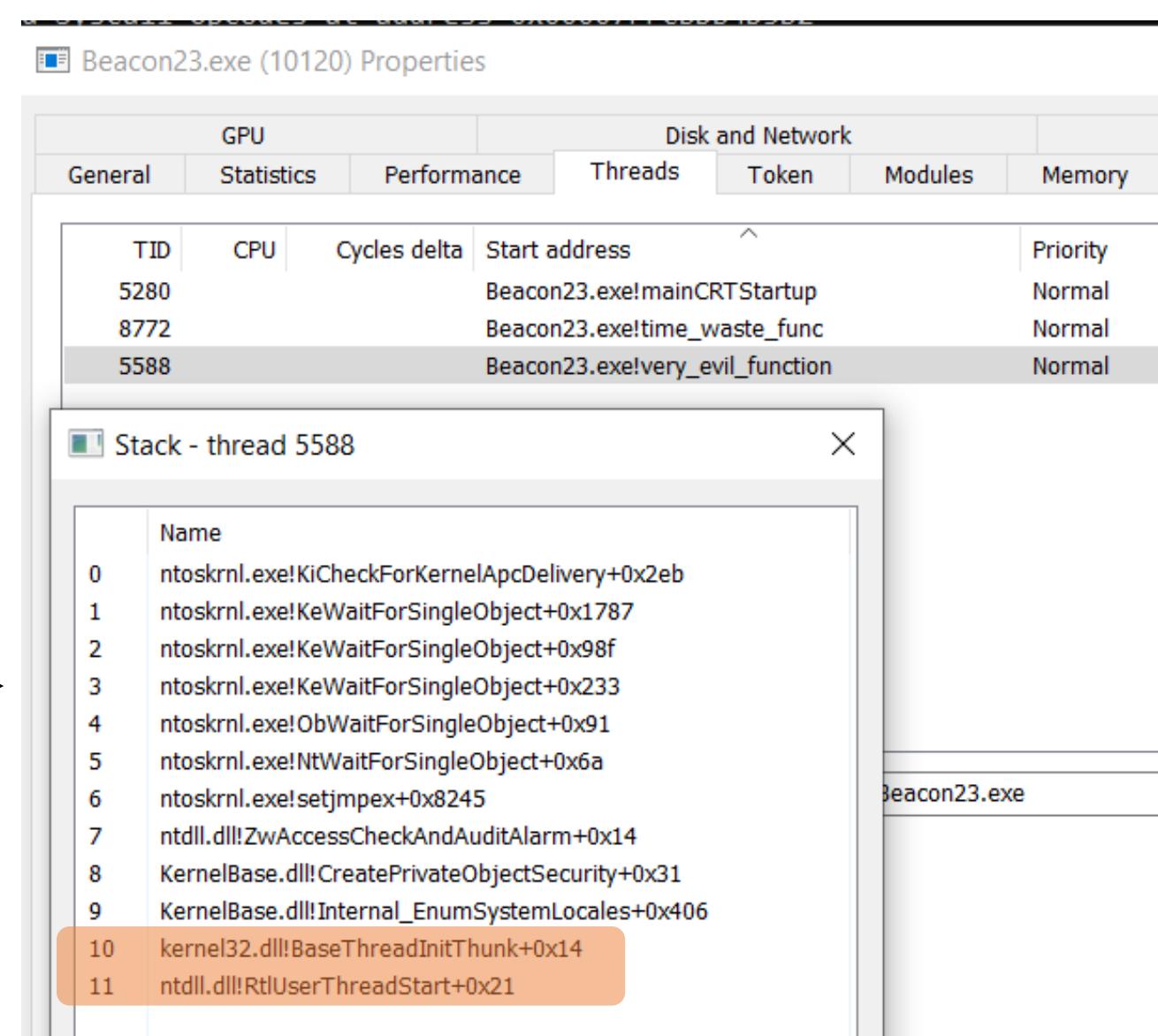
# Indirect System Calls with Call Stack Spoofing (randomized)

- Unwindable stack
- Correct calling tree  
Kernelbase > NTDLL > Kernel
- Caller module hidden
- Target kernel function hidden  
in userspace



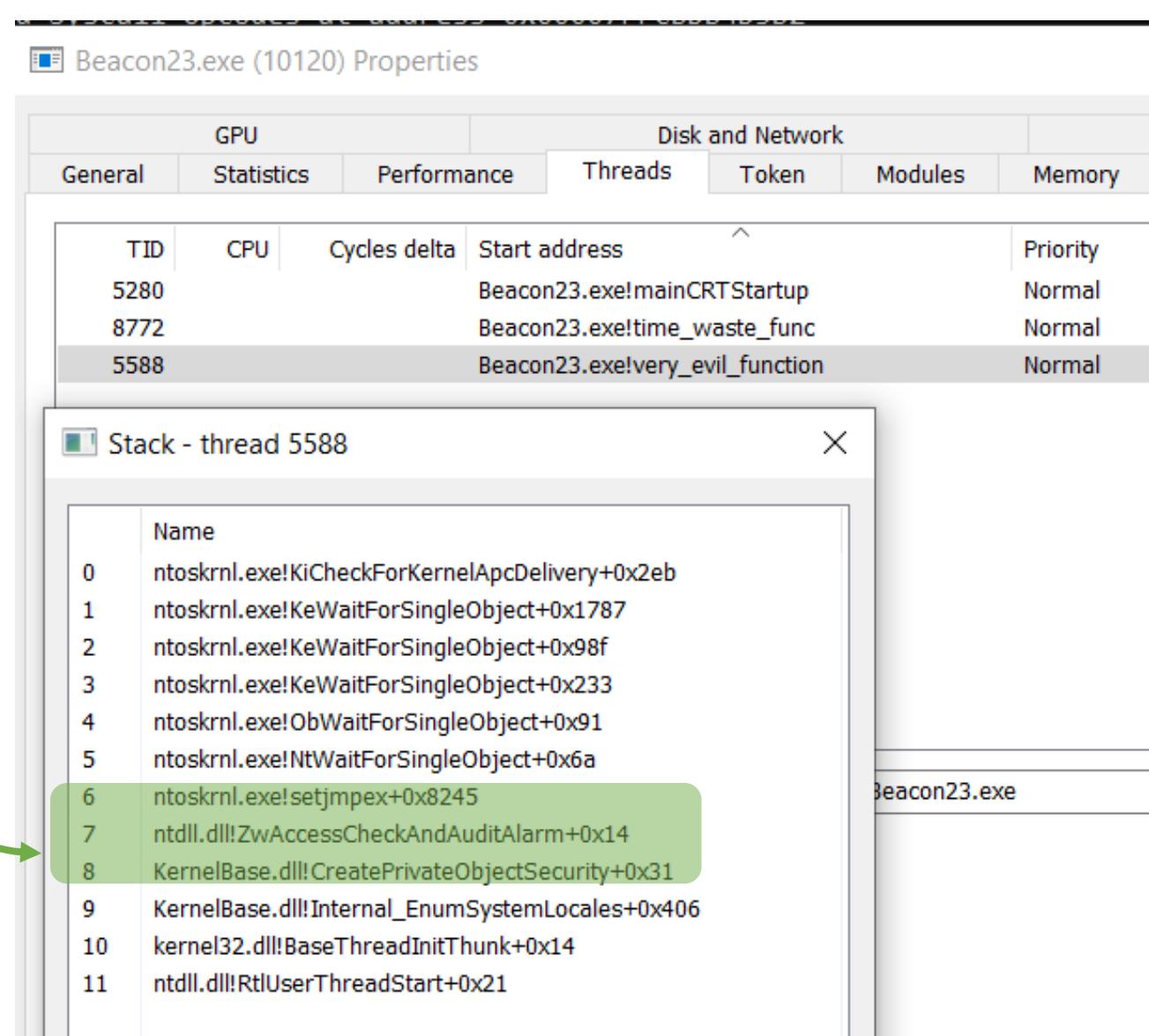
# Indirect System Calls with Call Stack Spoofing (randomized)

- **Unwindable stack**
- Correct calling tree  
Kernelbase > NTDLL > Kernel
- Caller module hidden
- Target kernel function hidden in userspace



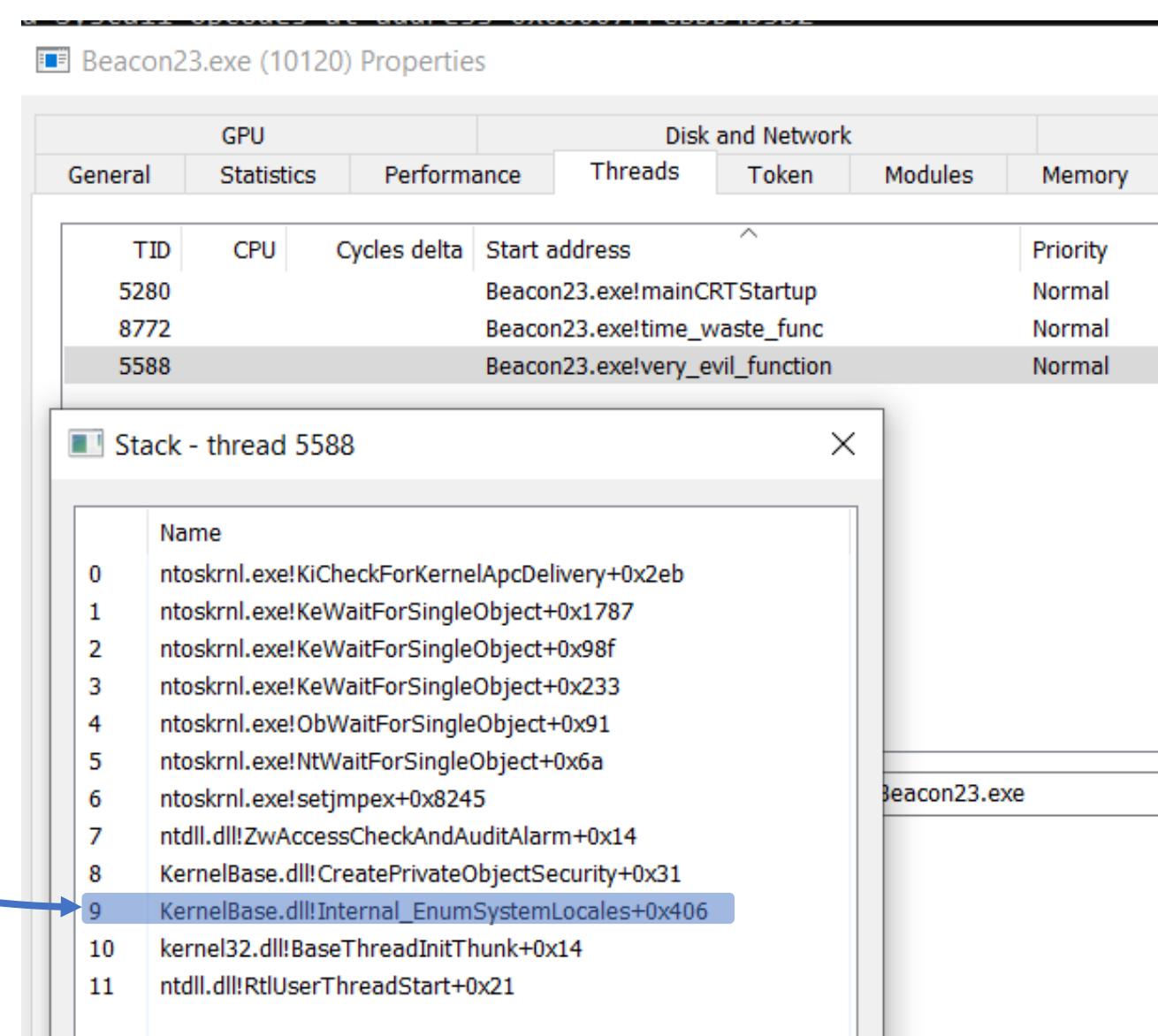
# Indirect System Calls with Call Stack Spoofing (randomized)

- 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)

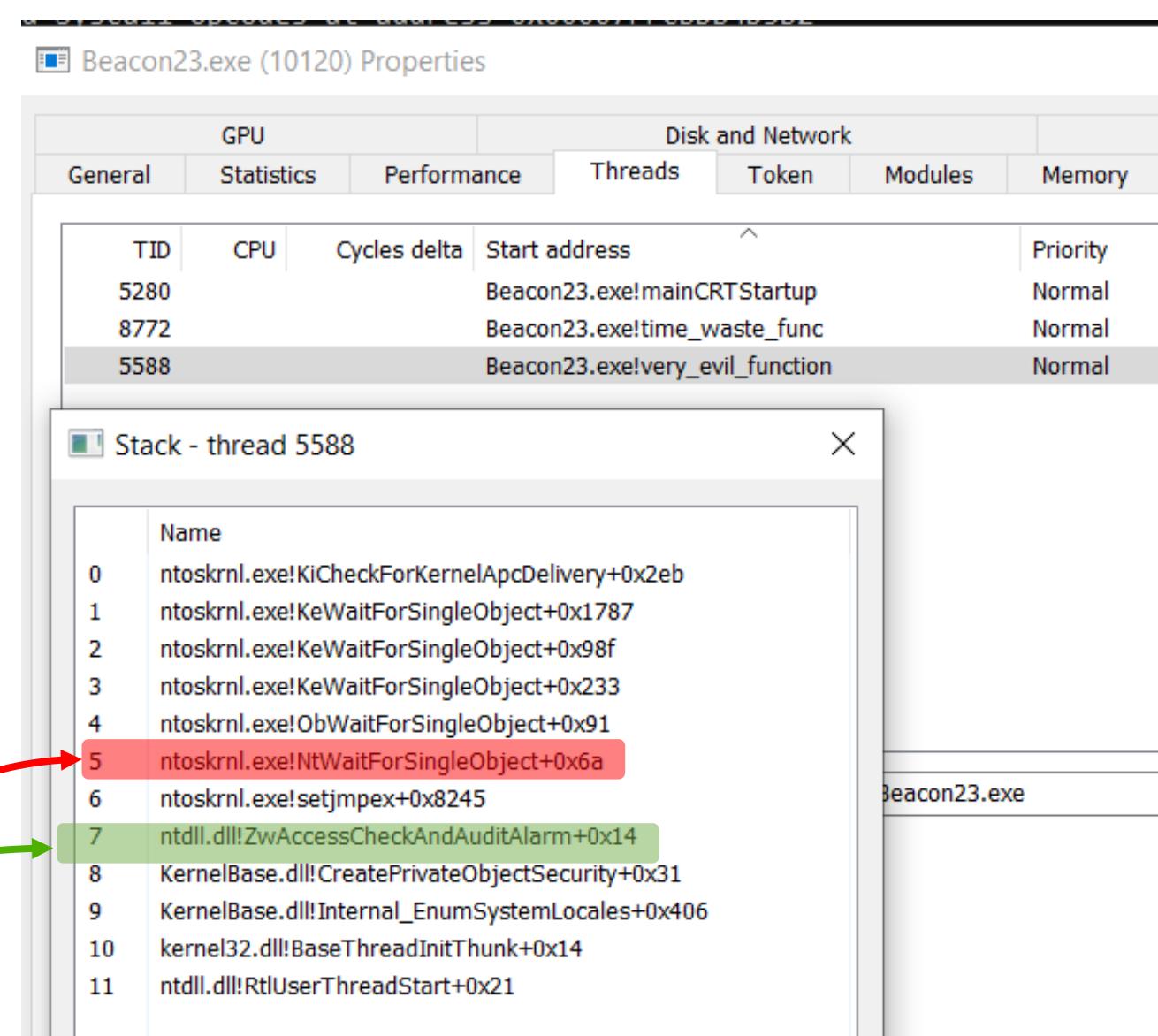
- Unwindable stack

- 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	00007FFCB25B44E	00007FFCBDB4D734	40	ntdll.00007FFCBDB41
	0000000002C1F408	00007FFCB215321	00007FFCB25B44E	40	kernelbase.00007FFC
	0000000002C1F448	00007FFCB22B5D6	00007FFCB215321	3E0	kernelbase.00007FFC
	0000000002C1F828	00007FFCB9C7614	00007FFCB22B5D6	30	kernelbase.00007FFC
	0000000002C1F858	00007FFCBDB026F1	00007FFCB9C7614	80	kernel32.00007FFCB1
	0000000002C1F8D8	0000000000000000	00007FFCBDB026F1		ntdll.00007FFCBDB0:
2604	000000000067F648	AAAAD5611899A64	00007FFCBDB4D134	8	ntdll.00007FFCBDB41
	000000000067F650	AAAAAAAAAAAAAA	AAAAD5611899A64	8	AAAAD5611899A64
	000000000067F658	AAAAAAAAAAAAAA	AAAAAAAAAAAAAA	8	AAAAAAAAAAAAAA
	000000000067F660	AAAAA21344719AFFF	AAAAAAAAAAAAAA	8	AAAAAAAAAAAAAA
	000000000067F668	AAAAAAAAAE81E0D	AAA21344719AFFF	8	AAA21344719AFFF
	000000000067F670	AAAAAAAAAAAAAA	AAAAAAAAAE81E0D	8	AAAAAAAAAE81E0D
	000000000067F678	AAAAAAAAAAAAAA	AAAAAAAAAAAAAA	8	AAAAAAAAAAAAAA
	000000000067F680	AAAAAAAAAAAAAAE2	AAAAAAAAAAAAAA	8	AAAAAAAAAAAAAAE2
	000000000067F688	AAAAAAAAAAAAAAAB	AAAAAAAAAAAAAAE2	8	AAAAAAAAAAAAAAAB
	000000000067F690	AAAAAAAAAAAAAA	AAAAAAAAAAAAAAAB	8	AAAAAAAAAAAAAAAB
	000000000067F698	AAAAAAAAAAAAAA	AAAAAAAAAAAAAA	8	AAAAAAAAAAAAAA
	000000000067F6A0	AAAAAAAAAAAAAA	AAAAAAAAAAAAAA	8	AAAAAAAAAAAAAA
	000000000067F6A8	AAAAAAAAAAAAAA	AAAAAAAAAAAAAA	8	AAAAAAAAAAAAAA

Stack - thread 7620

Name
0 ntoskrnl.exe!KiCheckForKernelApcDelivery+0x2eb
1 ntoskrnl.exe!KeWaitForSingleObject+0x1787
2 ntoskrnl.exe!KeWaitForSingleObject+0x233
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
User

# BYOVD

(Ab)use a signed driver to execute malicious actions

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

LOLDivers project : <https://www.loldivers.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

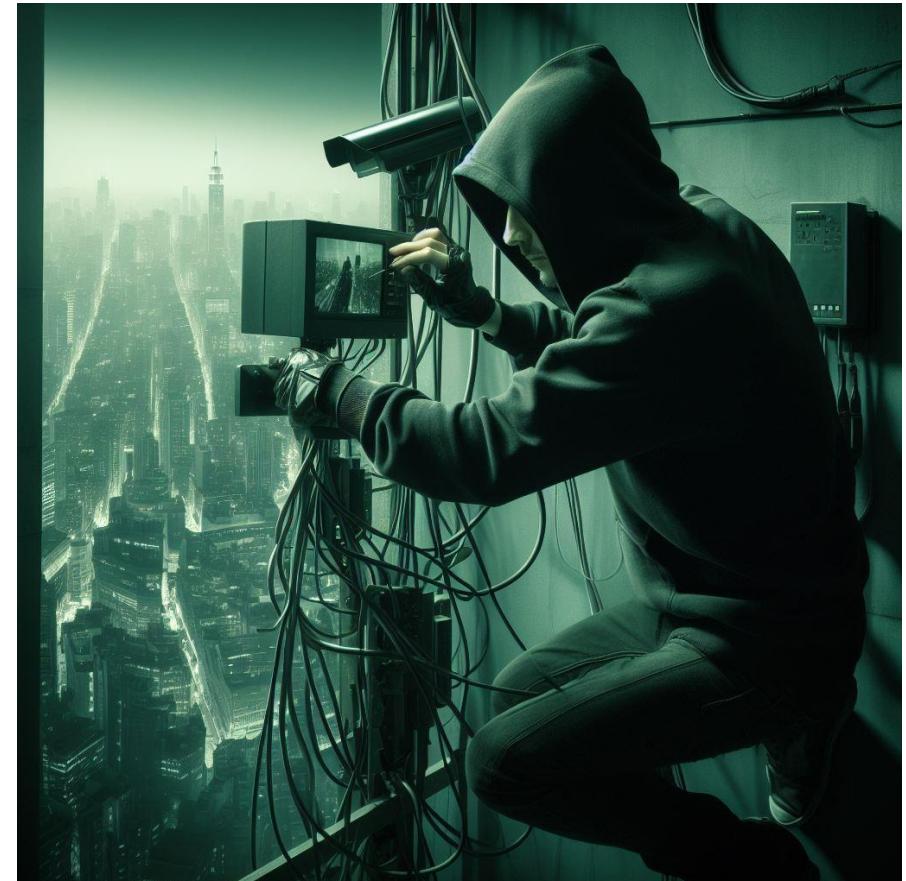
```
HANDLE_TABLE *ObjectTable
```
4. Search the Handle associated to the **target process**
5. Make the handle privileged by editing `HANDLE_TABLE_ENTRY->GrantedAccess`
6. Call `TerminateProcess()`



References: [Kernel Cactus](#)

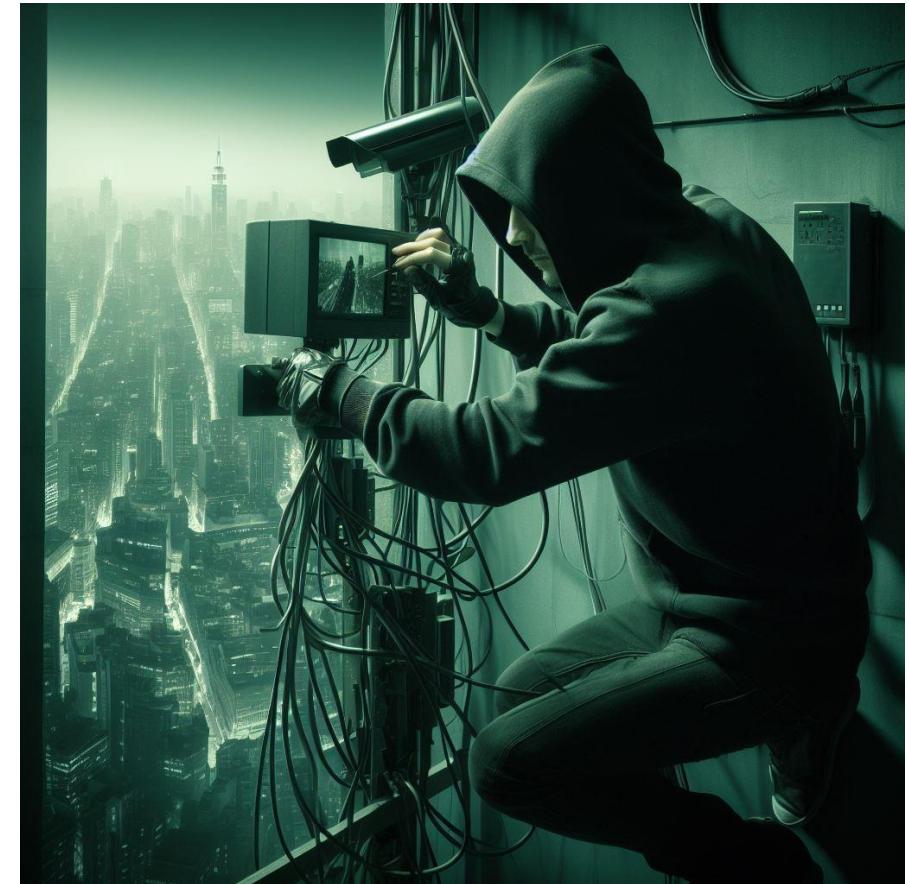
# Disable Kernel Callbacks

- 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 (Patch Guard) periodically checks for tampering of Kernel memory

**With great power comes great responsibility**



# Install your Unsigned Driver

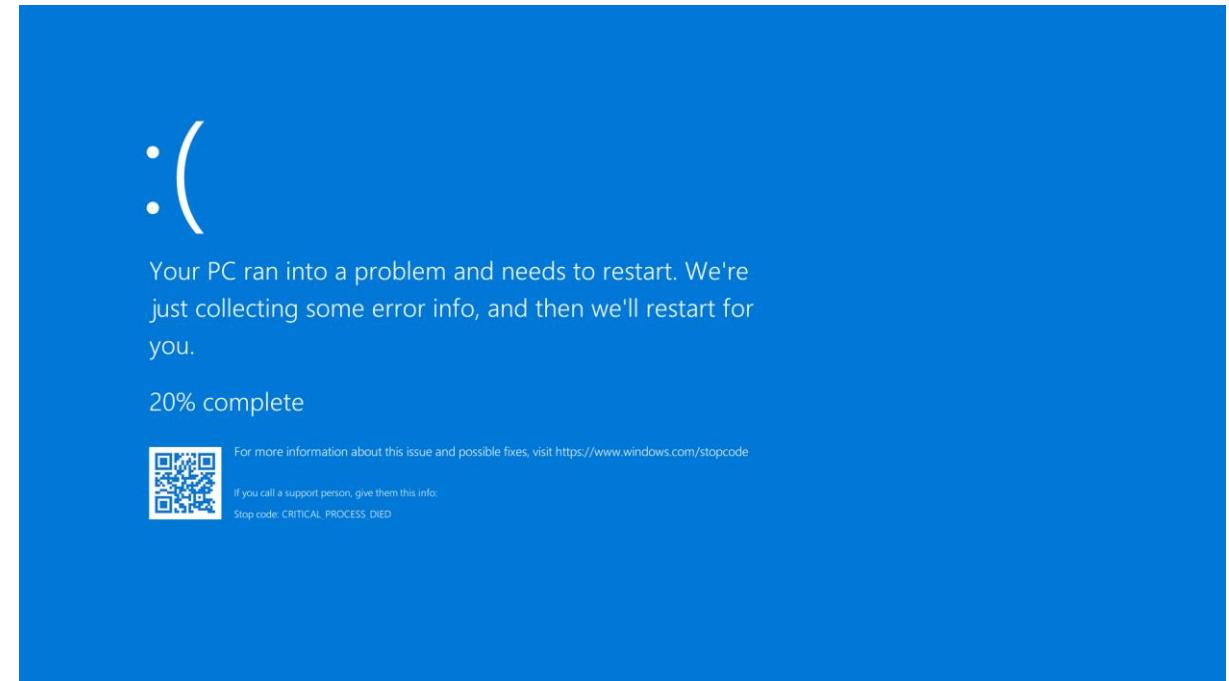
- Find Ci!g\_CiOptions
- Set Ci!g\_CiOptions to 0
- Install Driver
- Set Ci!g\_CiOptions value back

**If KPP check happens at the wrong time, GAME OVER**

References:

[Loading unsigned Windows drivers without reboot](#)

**With great power comes great responsibility**



# Install your Unsigned Driver

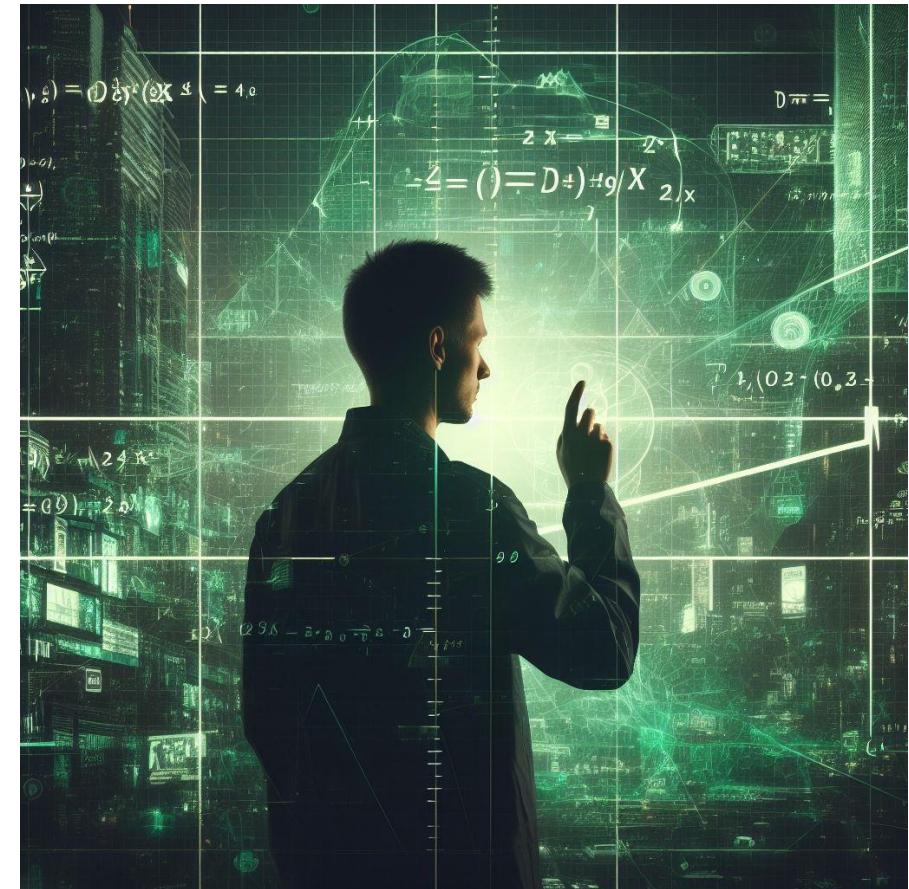
- Find `Ci!g_CiOptions`
- Set `Ci!g_CiOptions` to 0
- Install Driver
- Set `Ci!g_CiOptions` value back

**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.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](https://www.trendmicro.com/en_ph/research/23/q/hunting-for-a-new-stealthy-universal-rootkit-loader.html)



# THANKS!



proud sponsor of

BEACON

