Fantastic unwind information and where to find them


klezvirus.github.io/posts/Byoud

March 16, 2026

Foreword

This is the third and final installment in a series of posts on stack spoofing research that I presented at Black Hat Europe 2025. The first two posts covered Stack Moonwalking++ and Callback Hell, which explored techniques for spoofing call stacks in pre-CET environments.

However, Intel CET (Control-flow Enforcement Technology) fundamentally breaks those approaches. The shadow stack maintained by CET catches any attempt to modify return addresses, making traditional stack spoofing techniques obsolete on modern systems.

This post presents BYOUD (Bring Your Own Unwinding Data), a new framework that works within CET's constraints by targeting a different layer entirely: Windows unwind metadata. The techniques described here were developed to answer a simple question: can we spoof call stacks without touching return addresses at all?

Overview

Intel CET and Hardware Stack Protection (HSP) have fundamentally altered the landscape of stack spoofing techniques. Traditional approaches such as Stack Moonwalking that rely on modifying return addresses are rendered ineffective. CET's shadow stack detects these modifications immediately and raises a Control Protection exception.

CET enforces return flow integrity, while Windows stack unwinding operates independently through exception metadata. This separation presents a new attack surface. This post introduces **BYOUD** (Bring Your Own Unwinding Data), a CET-compliant stack spoofing framework that manipulates Windows unwind information without modifying return addresses.

In this blog, I'll present three main variants plus a JIT-specific approach, demonstrating how malware can present fully unwindable, legitimate-looking call stacks while remaining compliant with Intel CET, even when executing from memory.

TL;DR (aka I Don't Want to Read 8000 Words About Pointer Arithmetic)

I was told at Black Hat that this topic was "too complicated" and I was "unable to make it simpler." I have chosen to interpret this as a compliment. Here is my attempt at simple.

Explain everything and the meaning drowns in detail; simplify too much and the meaning disappears.

Every running program can be described as a call tree. Each function call is a node, and the path from the root to any currently executing function is exactly what you see in a call stack. If you have any intuition at all for how a debugger shows you “who called what,” you already understand the structure:

```
main()
  └─ do_the_needful()
       └─ maybe_do_maybe_dont()
            └─ ReadFile()   <- you are here
```

What makes this tree reconstructable at runtime is that each node stores its **return address** (the location to jump back to when it finishes) on the stack, at a fixed offset from the current stack pointer. That offset is the **frame size**: the amount of stack space the function allocated in its prologue. If you know the frame size of any node, you can find its parent. If you know the frame size of every node along a path, you can walk the entire tree from any leaf back to the root.

```
RSP ──► [ current frame     ]  frame size = 0x80
        [ return addr       ]  ← points to parent
        [ parent frame      ]  frame size = 0x200
        [ return addr       ]  ← points to grandparent
        [ grandparent frame ]  ...and so on
```

This is precisely what a stack walker does. It reads frame sizes from `.pdata` unwind metadata, hops from frame to frame, and reconstructs the call tree one node at a time.

BYOUD is a set of techniques that exploits this exact arithmetic. Once you know the cumulative frame size from the root of the tree down to some node N (i.e., the total stack distance from the thread’s entry point to N) then for any child of N, the distance is simply:

```
distance(child) = distance(parent) + child_frame_size
```

This means you can craft a fake subtree. Pick any chunk in the real call tree that you want to hide and an API you want to call. Manipulate the frame size of the API (or forward frame) to equal exactly the distance from the `BaseThreadInitThunk` to the frame calling the API + the frame size of the API. The hidden portion of the tree simply does not exist from an unwinding perspective.

```
Real tree:                              What the walker sees:

RtlUserThreadStart                      RtlUserThreadStart
  └─ BaseThreadInitThunk                  └─ BaseThreadInitThunk
       └─ [CRT main]                         - [HIDDEN]
            └─ shellcode()                   - [HIDDEN]
                 └─ TargetWindowsApi()       └─ TargetWindowsApi()
```

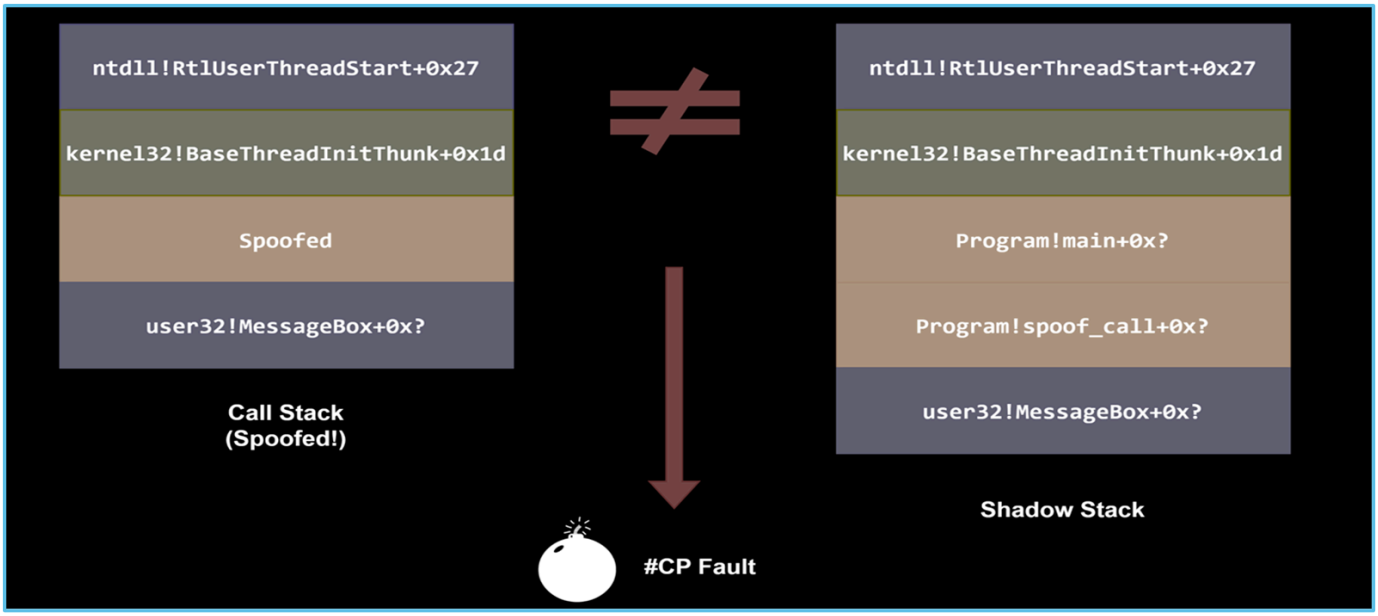
The remaining 8000 words explain how to make the Windows unwinder agree with this fiction manipulating a few data structures, and why this is CET compatible. You were warned.

The Dragon in the Room: Hardware Stack Protection

Microsoft Corporation initiated the deployment of user-mode Hardware Stack Protection (HSP) with Windows 10 20H1. HSP represents an exploit mitigation technology aimed at preventing the corruption of return addresses on the stack. Underpinned by silicon-based architecture, HSP leverages Intel's Control Flow Enforcement Technology (CET) and AMD's Shadow Stack in conjunction with software support. The terminologies HSP and CET are often utilized interchangeably.

The HSP mechanism establishes a shadow stack, distinct from the conventional stack. This shadow stack is rendered read-only in user mode and is exclusively composed of return addresses. This stands in contrast to the regular stack, which contains data and return addresses and necessitates write access for the proper functioning of applications.

Whenever a CALL instruction is executed, the current instruction pointer is pushed onto both the regular and shadow stacks. When RET instructions execute, the return address is popped from both stacks. Any mismatch generates an exception.



Return-Oriented Programming (ROP) attacks are mitigated because adversaries cannot write arbitrary values to the read-only shadow stack. The modification of the Shadow Stack Pointer (SSP) constitutes a privileged operation, rendering pivots unattainable.

In addition to Shadow Stack, CET includes Indirect Branch Tracking (IBT). This feature ensures that indirect branches only target valid instruction locations marked with a special `ENDBR` instruction. However, Microsoft decided to not support this feature, instead relying on their own Control Flow integrity check implementation (CFG/XFG).

Empirical evidence demonstrated that the implementation of Hardware Stack Protection would disrupt the execution of all variations of the Stack Moonwalking approach, including the frame swapping primitive. This confirmed theoretical expectations.

CET Operating Mechanism

CET operates as follows:

- Every CALL instruction pushes the return address onto both the regular stack and the shadow stack
- When a RET instruction executes, CET checks that the return addresses match on both stacks
- Any mismatch raises an exception (#CP, Control Protection Exception, interrupt 21)

An attacker who tampers with the main stack cannot spoof a valid return path without also compromising the shadow stack, which is hardware-protected and inaccessible from software.

Rethinking Stack Spoofing at the Design Level

Stack spoofing attacks do not aim to hijack control flow in the traditional sense. They seek to simulate a plausible forward control flow to mislead post-execution inspection or static analysis. Stack spoofing generally disregards the actual return path taken following an API call, provided that execution does not result in a crash.

CET is not designed to address in-memory execution or unresolvable call stacks. It does not inherently flag anomalous forward flows. Its enforcement is limited to ensuring that call and return sequences are consistent with the shadow stack maintained for each thread.

Most current stack spoofing techniques rely on forging stack frames by modifying return addresses directly, frequently producing return flows in the form of ad hoc ROP chains. These modified return paths violate CET's strict return integrity checks, triggering detection or termination.

Separation of Concerns

The shadow stack maintained by CET is entirely decoupled from the traditional Windows stack unwinding mechanism. This separation allows us to design techniques that address these two constraints independently: one targeting stack unwinding logic, the other focused on shadow stack consistency.

Two questions arise:

- Is it possible to spoof a call stack without touching the return addresses placed on the stack?

- Is it possible to dynamically create forward patterns that hide the caller address without compromising return flow integrity?

Bring Your Own Unwind Data

I designed and developed a technique to manipulate Windows stack unwinding information while leaving CET mechanisms and the shadow stack unaffected.

As discussed in Microsoft's documentation on x64 exception handling, stack walking on Windows is implemented through structured exception handling (SEH) and unwind metadata embedded in the PE format. Windows x64 relies on unwind codes and runtime function tables to traverse and reconstruct call stacks. Each function that supports unwinding must provide an `UNWIND_INFO` structure, which describes how to restore the previous frame.

During exception dispatching, profiling, or security checks, the system walks the stack using these unwind descriptors rather than relying on traditional frame pointers. If we can tamper with the information contained in the `UNWIND_INFO` structure of a specific function, we can trick the Windows unwinding algorithm into reconstructing a call stack that diverges from the one actually executed.

Feasibility of Tampering

The Windows unwinding mechanism is process-based. For every DLL loaded into a process, including the main executable image, the operating system retrieves the `UNWIND_INFO` metadata from the memory-mapped representation of structures within the `.pdata` and `.rdata` (or .xdata) sections associated with each module.

These structures are resident in the process's virtual memory. They are accessible and, under the right conditions, modifiable. Once we identify the memory addresses of these structures, we can alter the information they contain.

Design Objectives

From this realization, I've designed BYOUD (Bring Your Own Unwinding Data). I focused on developing three different variants with two primary objectives:

- Conceal arbitrary portions of the stack, building on principles from the stack moonwalking techniques to render segments of the call stack invisible to user-mode inspection and unwinding tools
- Construct execution sequences that remain valid under direct shadow stack analysis, maintaining the forward chain fully visible while hiding the caller from inspection

BYOUD: Direct Tampering

In my first experiments, I've selected the RPC Server Call `NdrServerCall2` as the main proxy function. This call can be used for arbitrary call invocations using one structure passed in RCX as a pointer (`PRPC_MESSAGE`). I extensively worked on abusing this function in past research. As I will reiterate often in the reminder of this blog post, any other proxy function would do just fine.

The Windows OS unwinding mechanism does not validate the exact return address during stack traversal. It ensures that the address falls within the bounds of a function associated with a matching unwind descriptor. Return addresses can often be substituted freely if the replacement corresponds to a function with an identical stack frame size and compatible unwind metadata.

In most cases we can swap return addresses if the functions they point to have the same frame size. We can replace multiple stack frames with one that is equal in size to the sum of the sizes of the replaced frames.

Addressing CET Constraints

During real-world execution, finding a stack frame that meets these constraints isn't always practical. CET's return flow integrity will block most forms of return address swapping, making traditional frame spoofing ineffective. To achieve the same deceptive effect, we don't need to modify the stack at all. We can tamper with the unwind information of a function we can safely use as a proxy.

The technique works as follows:

1. Identify the proxy function `UNWIND_DATA`

Locate the `RUNTIME_FUNCTION` structure within the EXCEPTION directory (`.pdata` section) and recover the address of the unwind metadata (`.rdata` section).

2. Calculate the size of the area to conceal

This is done dynamically by identifying the return address of `BaseThreadInitThunk`, then scanning the stack downward from the thread stack base (retrievable via the TEB) until this address is found. This emulates a custom version of the `_AddressOfReturnAddress` intrinsic. The distance from this address to the current RSP provides the exact stack region that needs to be concealed.

The simplest implementation is a stack search for the return value:

```
.code

; RCX - QWORD value to search for
; Returns:
;   RAX = offset in bytes from RSP to found address, or 0 if not found

StackSearch PROC
    mov r11, rsp             ; Save current RSP
    mov r10, gs:[08h]        ; Get StackBase from TEB

    mov rdx, r11             ; RDX = search pointer (start from RSP)

search_loop:
    cmp rdx, r10             ; Have we reached StackBase?
    jae not_found            ; If yes, stop

    cmp qword ptr [rdx], rcx ; Compare memory at [RDX] with search value
    je found

    add rdx, 8               ; Move to next QWORD
    jmp search_loop

found:
    mov rax, rdx
    sub rax, r11             ; RAX = found address - RSP
    ret

not_found:
    xor rax, rax
    ret

StackSearch ENDP

END
```

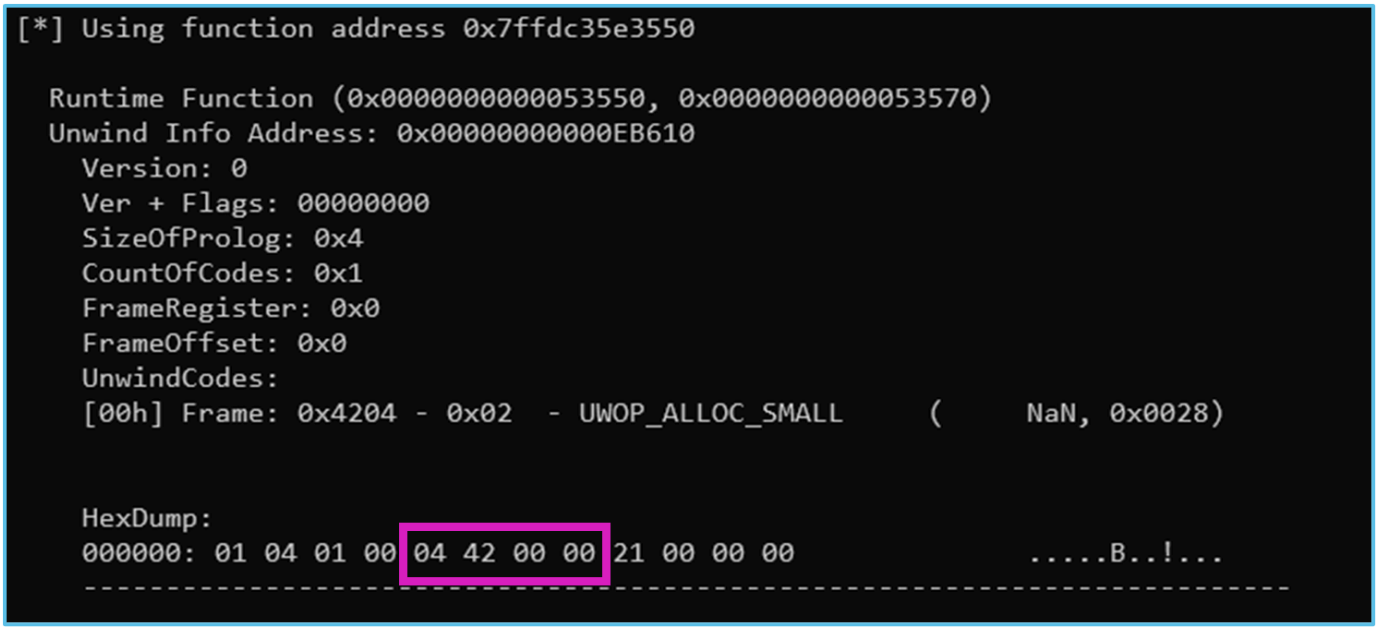
3. Tamper the proxy unwind information

We can now tamper the Proxy function unwind information to match the allocation size of the conceal data in addition to its own frame size. To not disrupt the unwinding information, we need to understand what metadata is contained in the function. For the sake of giving a real example, I've opted for a fixed Proxy call (`NdrServerCall2` in the images, `InitOnceRunOnce` in the released framework); however, many other proxy calls could be used on its behalf (check my previous post).

Properties of NdrServerCall2

`NdrServerCall2` has properties that make it suitable for this kind of tampering:

1. The function has one unwind code and no linked exception handler. The `UNWIND_CODE` contains two null bytes after the first one (used for alignment, as unwind codes are 4-bytes aligned), which we can use to add an additional Unwind code to this structure.



1. The single unwind code is a small allocation, which immediately gives away the full size of the frame without requiring additional calculation.

Supporting Arbitrary Frame Sizes

To support stack frame sizes of almost any value, we must work within the constraints of `UNWIND_INFO` and the available unwind codes.

```
typedef union _UNWIND_CODE {
    struct {
        UBYTE CodeOffset;          // 0xff00
        UBYTE UnwindOp : 4;        // 0x000f OPCODE
        UBYTE OpInfo : 4;          // 0x00f0
    };
    USHORT FrameOffset;
} UNWIND_CODE, * PUNWIND_CODE;

typedef struct _UNWIND_INFO {
    UBYTE Version : 3;
    UBYTE Flags : 5;               // 4 bytes
    UBYTE SizeOfProlog;            // 4 bytes
    UBYTE CountOfCodes;            // 4 bytes
    UBYTE FrameRegister : 4;
    UBYTE FrameOffset : 4;         // 4 bytes
    UNWIND_CODE UnwindCode[1];
    union {
        OPTIONAL ULONG ExceptionHandler;
        OPTIONAL ULONG FunctionEntry;
    };
    OPTIONAL ULONG ExceptionData[];
} UNWIND_INFO, * PUNWIND_INFO;
```

Some unwinding opcodes require two `UNWIND_CODE` entries for full representation. This is the case with `UWOP_ALLOC_LARGE`, which is necessary when expressing larger stack allocations.

In the case of `NdrServerCall2`, we can abuse the lingering two null bytes (using for alignment) to introduce a new `UNWIND_CODE`. In Windows, the information after the `UNWIND_CODE` array is optional.

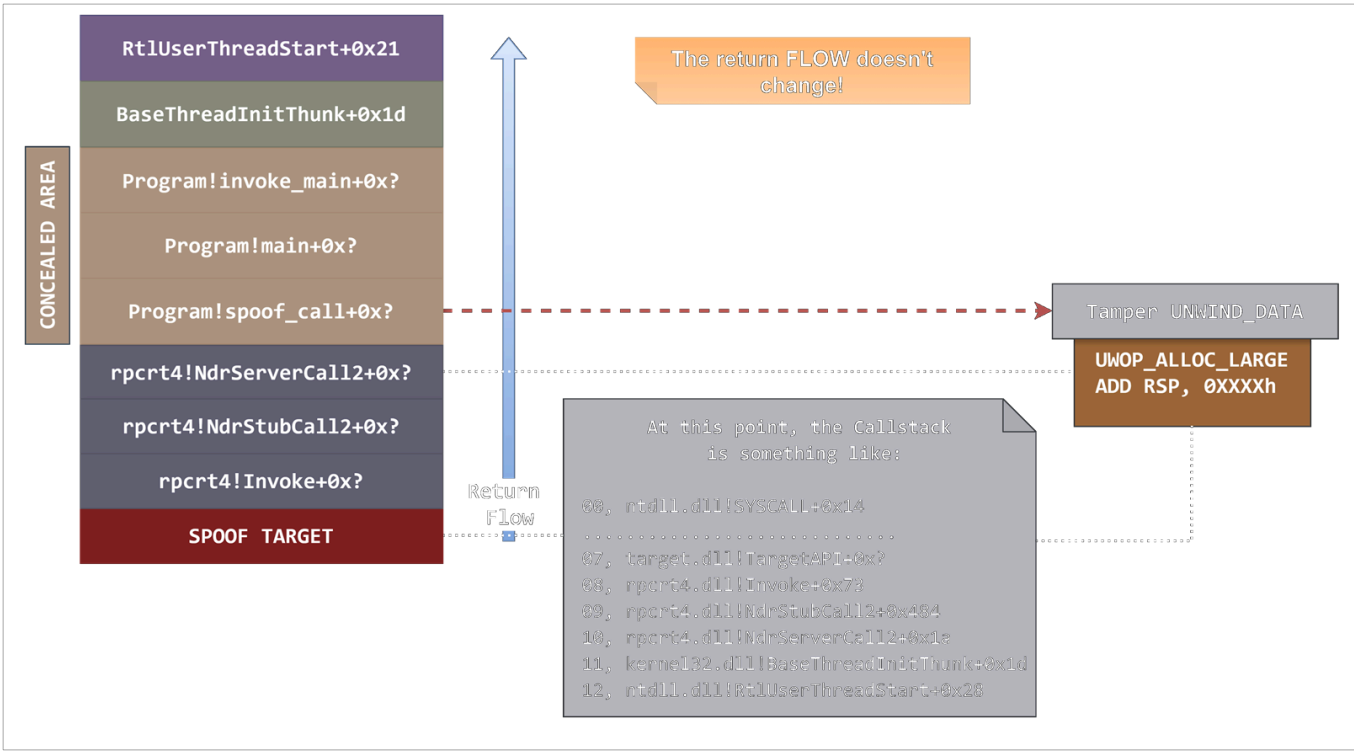
To properly modify the `UNWIND_CODE` array, we make the following adjustments:

- **CountOfCode**: Increase by 1 as we are adding a code
- **SizeOfProlog**: add rsp, `[00h-7fh]` is 4 bytes, but `add rsp, [80h-7fffffffh]` is 7, so we set this to 7
- **UnwindCode**: Fully overwritten. To change from `UWOP_ALLOC_SMALL` to `UWOP_ALLOC_LARGE`:
  - First `UNWIND_CODE` changes from `04 42` to `04 01`
  - Second changes from `00 00` to `XX YY`, representing the size of the allocation scaled by 8, reversed (example: for an allocation of 0x618 bytes, this code will be C3 00, as 0x618 / 0x8 = 0x00C3)

- **Flags**: To make the unwinder properly ignore the missing presence of the optional fields, we set up the frame as an unwind termination handler. The flags are populated then with `UNW_FLAG_UHANDLER`. This may somehow be counterintuitive for someone as logically the correct flag for “ignoring” the exception handler should be `UNW_FLAG_NHANDLER`. The reason why this is necessary is that we need the handler to UNWIND the frame as per UNWIND data we are setting it to use.

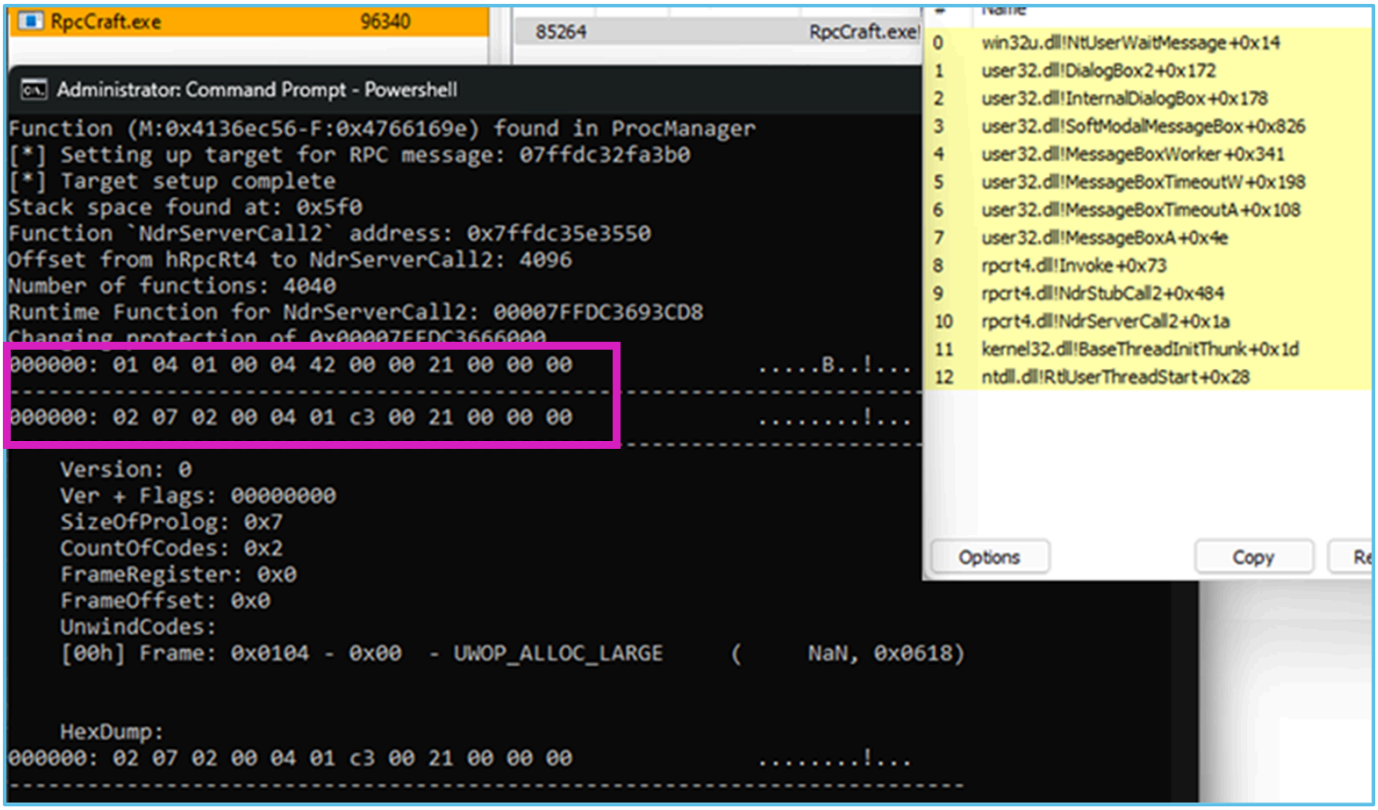
Execution Results

With the new `UNWIND_CODE` array and `UNWIND_INFO`, on execution the stack appears as follows:



Once the setup is complete, we can validate this stack spoofing approach by executing the function directly within the program’s main thread. The technique successfully masks the presence of the main module from the reconstructed call stack. This proof-of-concept is implemented as a standard PE file, but the resulting stack trace demonstrates the technique will work if injected as shellcode.

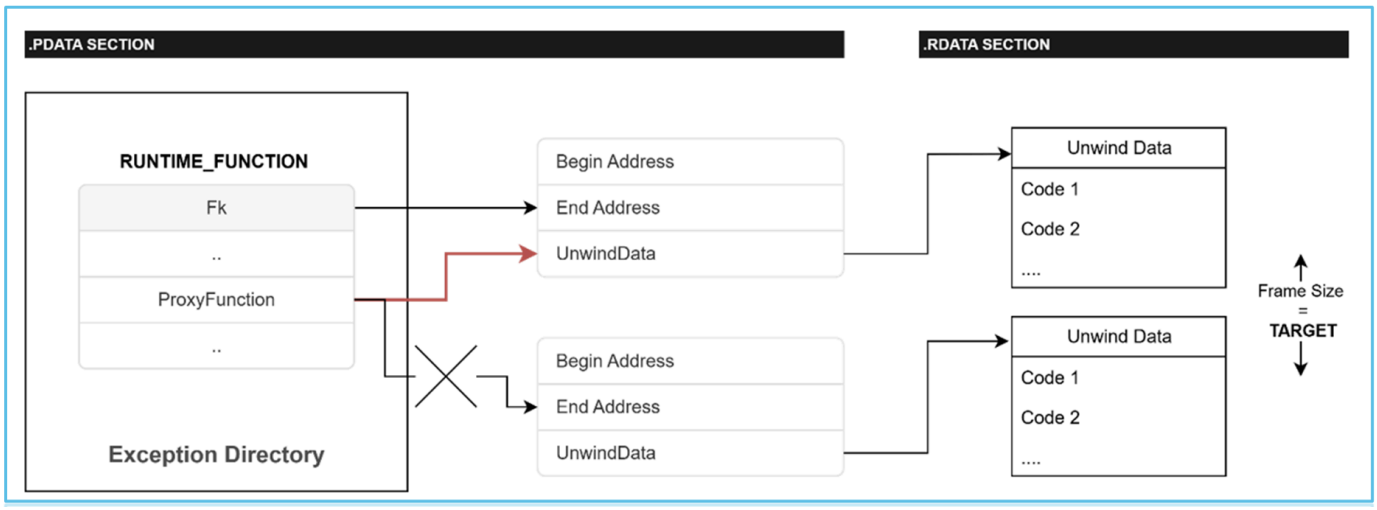
The technique produces an identical call stack on systems with or without CET enabled. To test on a CET enabled system, compile with the `/CETCOMPAT:Yes` flag.



BYOUD: Offset Hijacking

A variation of this technique involves locating another `RUNTIME_FUNCTION` entry whose associated stack frame size matches or exceeds the size of the data we intend to conceal. Once such a function is identified, we can use its unwind metadata to simulate a valid stack frame, effectively covering the concealed region without triggering stack unwinding inconsistencies.

This variation targets the runtime function table itself rather than modifying the data referenced by the unwind data offset. The attack rewrites the offset of the Unwind data referenced by the Runtime Function, causing a redirection of the unwinding information resolution. The manipulation occurs within the `.pdata` section (where the runtime function table typically resides) rather than the `.rdata` section where the actual unwinding data resides.



For this attack to work, the `UNWIND_DATA` address to replace should be in the same runtime function table as the one that will be used as a replacement.

BYOUD: Runtime Function Injection

This technique builds on the previously described methods, with a focus on targeting the process image base rather than external library modules. While still under active development, its core operating mechanism can be summarized as follows:

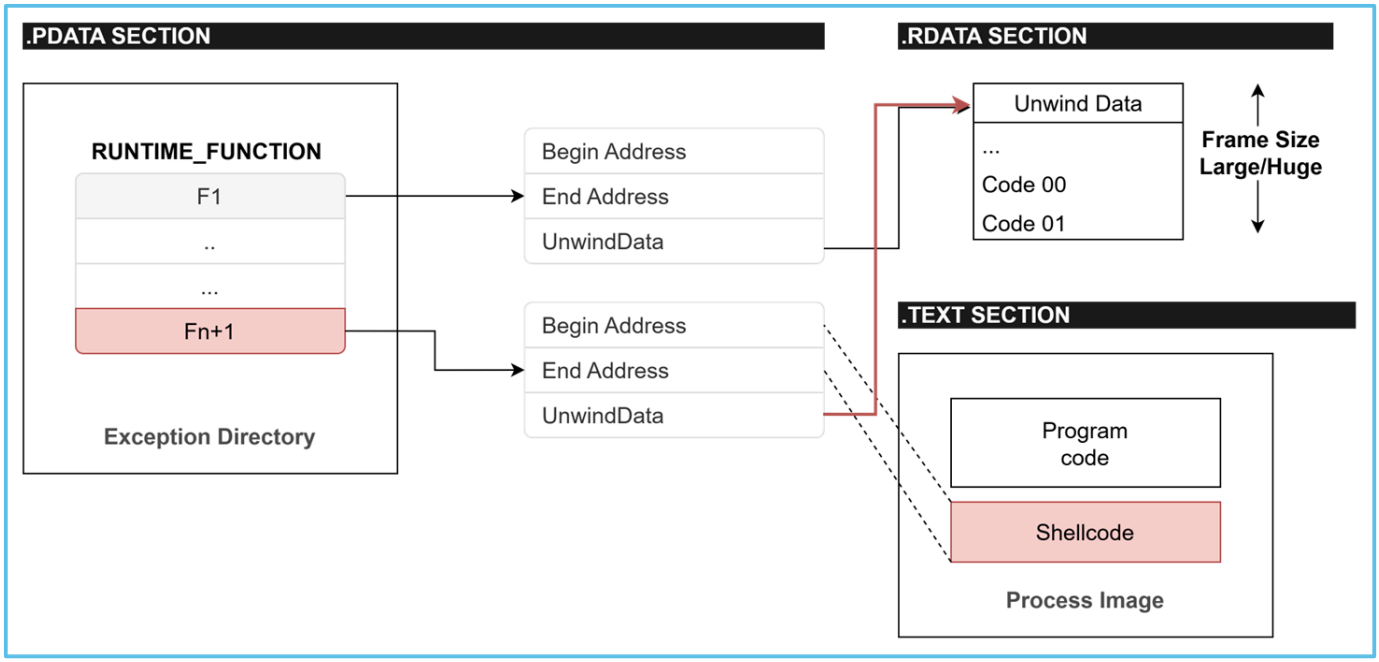
1. By exploiting the padding introduced for page alignment, shellcode is appended directly to the process's image

2. A `RUNTIME_FUNCTION` structure is dynamically constructed to enable the custom code region to participate in structured exception handling and unwinding. Two approaches exist:

   a. Append the structure to the process's exception table. The Exception Directory Entry is modified with an increased size (+0Ch bytes)

   b. Overwrite or hijack an existing `RUNTIME_FUNCTION` entry. Instead of creating a new structure, malware takes control of an existing one by changing its Function Begin Address and End Address to match the newly inserted code

3. Using tampering or hijacking, an `UNWIND_INFO` structure is created and associated with the newly constructed `RUNTIME_FUNCTION`. The unwind data is crafted to define an artificially large stack frame for the target function

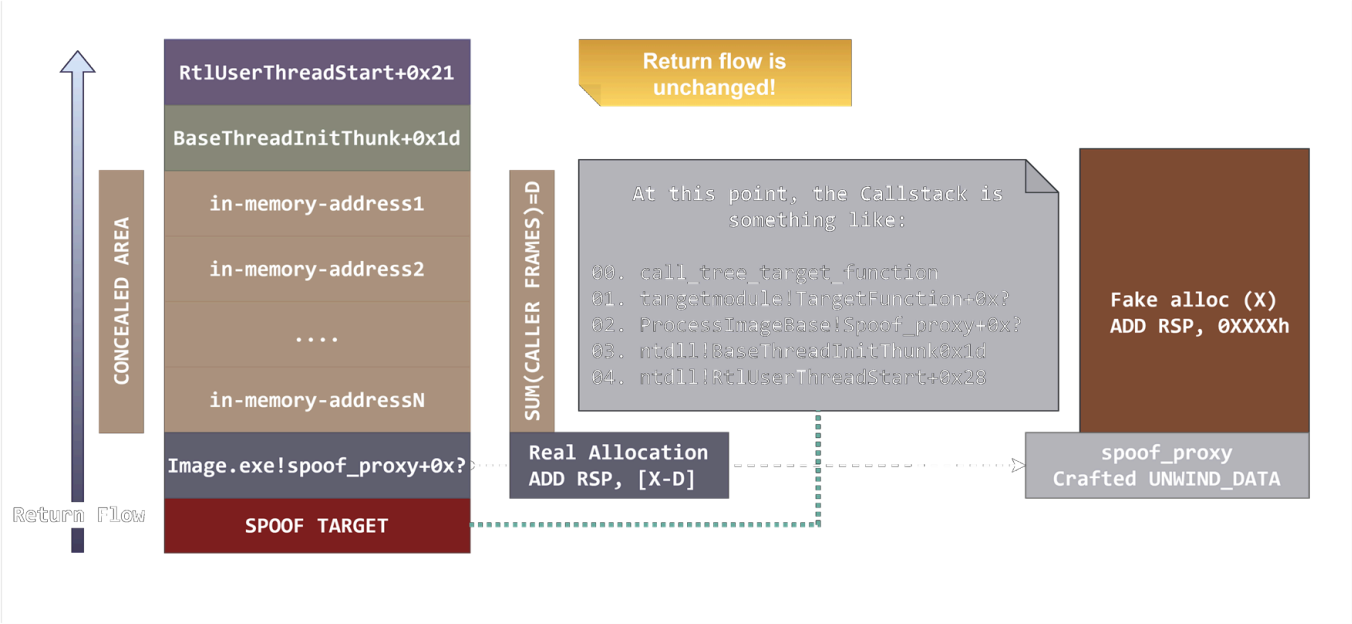


1. The shellcode appended to the .text section includes a dynamic proxy function, which performs the following sequence:

   a. Calculate the distance between the current RSP and the `BaseThreadInitThunk` function return address (call this D)

   b. Perform a stack allocation of exactly `|X - D|`, prepare the parameters, and call a provided function pointer

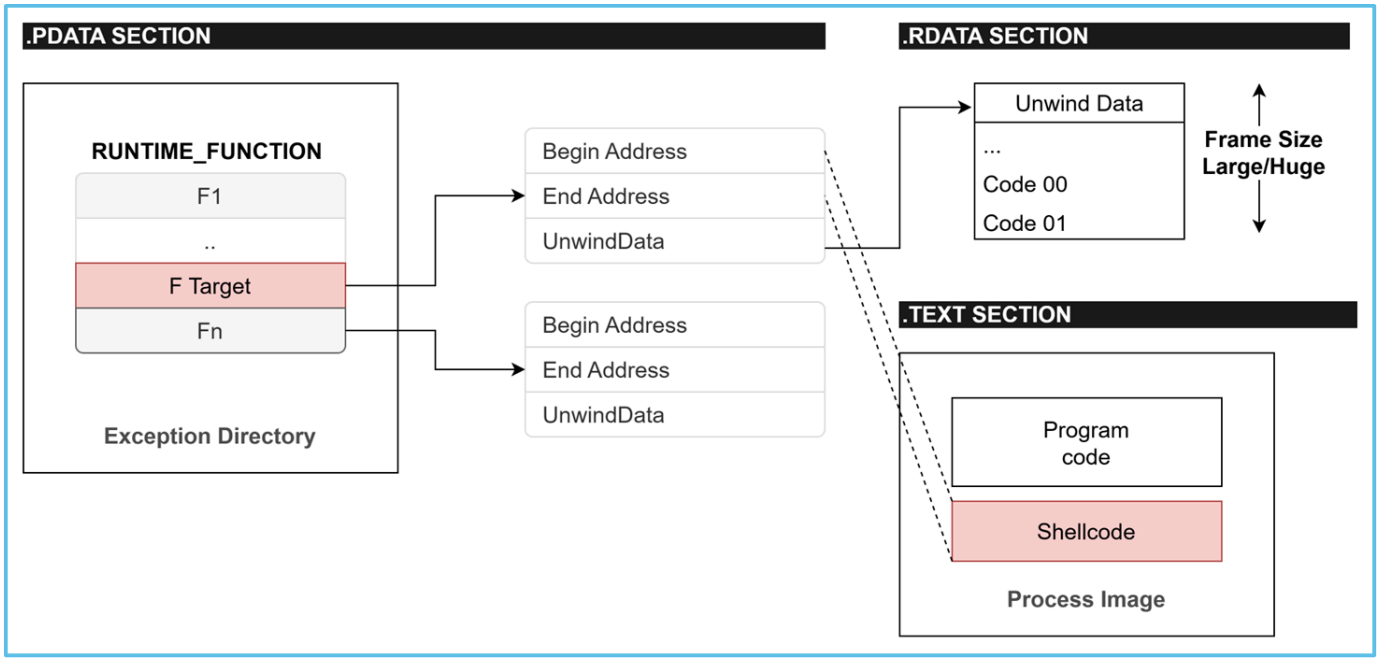
This architecture enables the dynamic concealment of all caller frames whenever the designated proxy function is used. Like the previously discussed techniques, it is fully CET-compatible. Execution remains compliant with shadow stack enforcement. This approach can be reliably used in any target process, regardless of whether CET is active.



BYOUD: Runtime Function Reuse

A slight modification of the above works by enumerating `RUNTIME_FUNCTION` structures until we find one which has an unwind data suitable for our shellcode.

In this case, we don't need to add a `RUNTIME_FUNCTION` to the table, but only to modify the existing `RUNTIME_FUNCTION` Begin and End addresses to point to our shellcode.



BYOUD: JIT Edition

JIT processes have been a prime target for abuse due to their ability to generate and execute code at runtime. These processes often expose memory regions with RWX (read-write-execute) permissions, which deviate from standard memory protection policies. Such characteristics make JIT engines attractive to threat actors looking to inject, hide, or execute malicious code under the guise of legitimate runtime behavior.

Just-In-Time (JIT) compilation dynamically translates intermediate representations into machine code during runtime. While this enables performance optimization and greater portability, it introduces a challenge on Windows in maintaining accurate exception handling and stack unwinding metadata.

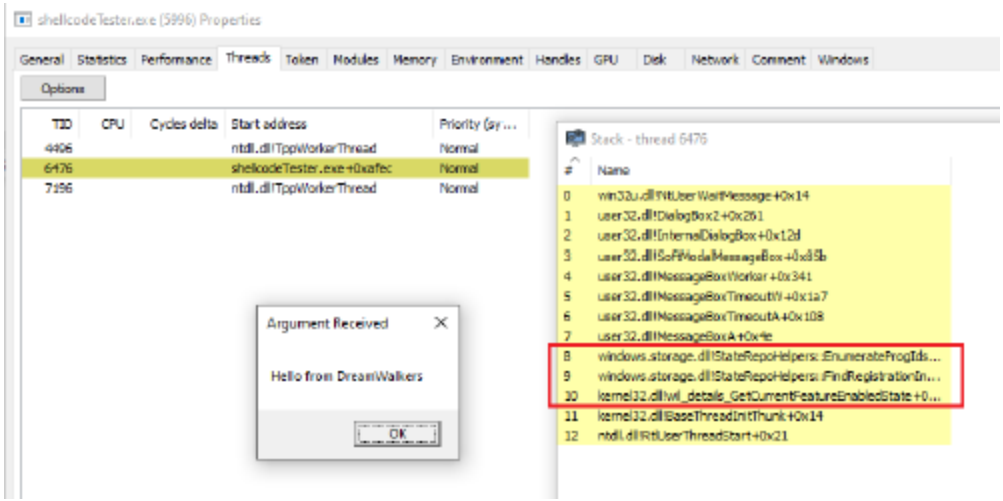
Dynamic Function Tables

To support stack unwinding and exception handling, Windows offers APIs to JIT compilers to register and manage their own unwinding metadata. The Windows API provides `RtlAddFunctionTable` and `RtlDeleteFunctionTable` to register and unregister dynamic unwind metadata. These functions allow a JIT runtime to allocate executable memory, generate code, and then associate it with a corresponding array of `RUNTIME_FUNCTION` entries.

These dynamically registered entries are stored in the process's internal dynamic function table, maintained by the OS. When an exception occurs or a stack walk is triggered, the Windows unwinder consults both static exception directories (from loaded modules) and dynamic function tables (populated via `RtlAddFunctionTable`).

Limitations of Existing Approaches

Some novel approaches have been presented that exploit this functionality to register dynamic function tables for dynamically allocated code. While these approaches are a step in the right direction, they fail to understand the real issue that stack spoofing addresses. When using stack spoofing, we don't just want to make the stack unwindable, rather to conceal the caller in a way that is not easy to identify.



The project however tried to explore avenues still untouched by the security community (dynamic exception entries, (e.g., `RtlAddFunctionTable`), and was a good step in the right direction to encourage people to move on from simply copying and renaming `StackMoonwalk`.

BYOUD Adaptation for JIT

An adaptation of the third variation of the BYOUD technique performs a full exception setup using a dynamic exception table instead of adding an entry to the PE `.pdata` section. The reason why I'm not fond of these "variants" is that in my tests, although I managed to get a fully unwindable stack "locally" (in WinDbg, or stack trace via hook), I couldn't replicate the same stack frame resolution using `StackWalk64` from an external tool.

The inner working of the stack spoofing is virtually the same. The difference is the allocation of the relevant structures for the Exception setup. Instead of overwriting the PE memory-mapped section, we construct the `RUNTIME_FUNCTION` on the heap of the process, then call `RtlAddFunctionTable` to register it as a dynamic exception table. For this technique, there are 2 sub-strategies:

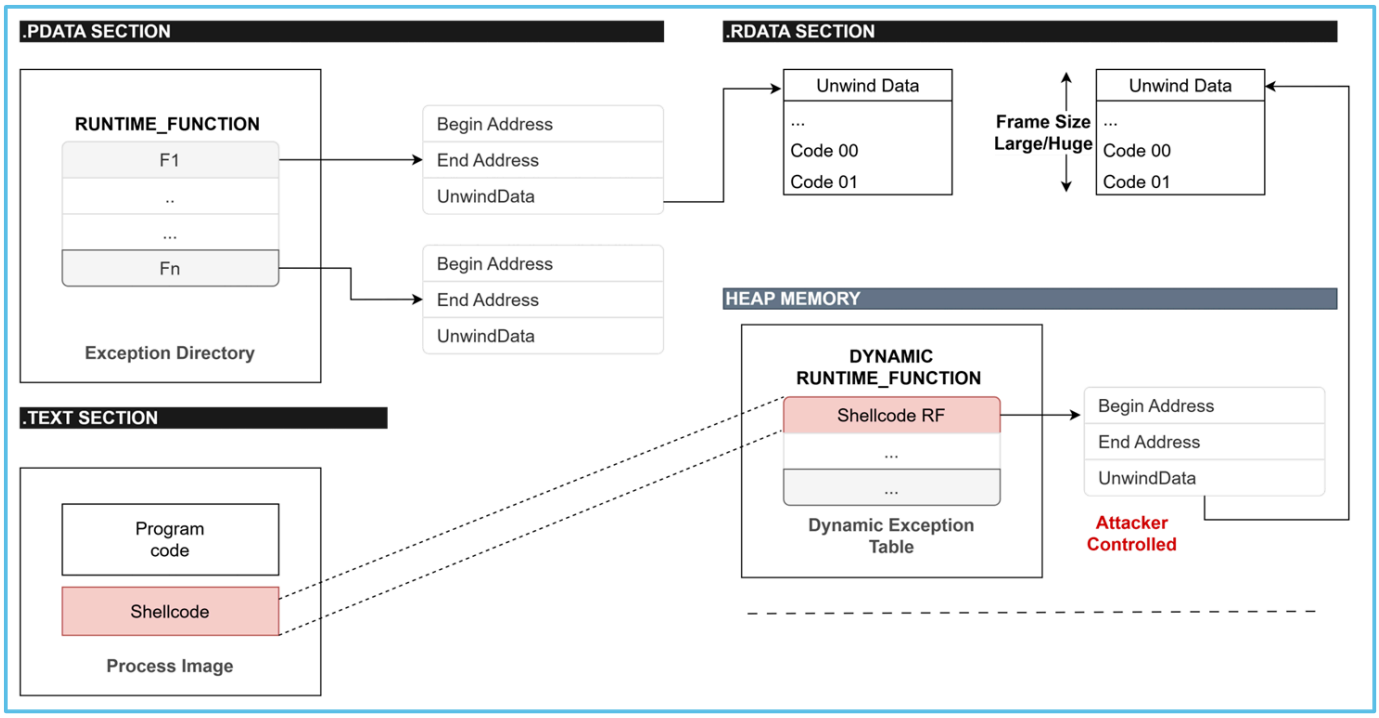
1. Reuse the UNWIND_DATA of a suitable function in .pdata
2. Just fully construct the UNWIND_DATA on the heap

Probably no need to say it, but both come with a bunch of drawbacks.

There is another slightly difference in approach in regard to allocate the new memory:

1. Allocate in a random virtual address
2. Allocate in the module itself (code caves or end of module)
3. Allocate near to the module, exploiting gaps introduced by relocations or just using an adjacent VA

JIT BYOUD + UNWIND_DATA Reuse (Shellcode Residing in DLL Module)



> I'm pretty sure somebody will be wondering why we are REUSING an existing UNWIND_INFO address instead of creating everything from scratch. The answer is simple, even though it might not be immediate. The reason is that the shellcode resides in the .text section of the main process. This means the UNWIND_INFO address still needs to be located within a DWORD offset from the image base address.
>
> I've considered potential edge cases here, where the bytes that we need (UNWIND_DATA) occur naturally in other sections, or even in adjacent modules' memory, but didn't actively hunt for those or try to leverage them directly.
>
> In theory, it could have been possible to just allocate another fake, adjacent section (image-base+[0x0, 0xFFFFFFFF]) with all the UNWIND_DATA sequences we need, or use an existing RW section to store the UNWIND_DATA needed for the tampering. However, in this case the symbol resolution would likely fail for the module and show the spurious address in the stack as it was normally VirtualAlloc'd in a random memory range, so there is no advantage in using this strategy.

Problems

The issues that I have personally observed and are why I don't personally like JIT-like techs are the following:

- They are not compatible with stack cookies (`/GS`). You may be good enough to find your solution to this, though

- From my test, there is an… issue (?) with `StackWalk64` which will make this technique possibly always detectable (false-positives apart)

StackWalk64 Problem

When a function is registered dynamically via `RtlAddFunctionTable`, the OS correctly inserts the corresponding `RUNTIME_FUNCTION` into a linked list of `_DYNAMIC_FUNCTION_TABLE` structures maintained by ntdll, each node also inserted into an AVL tree for efficient lookup. The OS unwinder, `RtlVirtualUnwind`, reaches these entries through `RtlLookupFunctionEntry`, which, after exhausting its fast-path cache and invoking `RtlpxLookupFunctionTable` to find the owning module, will call `RtlpLookupDynamicFunctionEntry` if and only if no static module claims ownership of the target address. This is the critical condition, and it is enforced explicitly in `RtlLookupFunctionEntry`:

```
// RtlLookupFunctionEntry (reconstructed)
if ((ControlPc < CachedImageBase) ||
    (CachedImageSize + CachedImageBase <= ControlPc)) {
    // Address outside cached module range - query full module list
    tablePtr = RtlpxLookupFunctionTable(ControlPc, &local);
} else {
    // Fast path: address is inside cached module - use cached .pdata
directly
    tablePtr = CachedTablePtr;  // never checks dynamic table
}

if (tablePtr == 0) {
    // No module owns this address - check dynamic registrations
    return RtlpLookupDynamicFunctionEntry(ControlPc, ImageBase);
} else {
    // Module found - binary search its static .pdata only
    return BinarySearchExceptionDirectory(tablePtr, ControlPc);
}
```

When shellcode is placed inside the virtual address range of an existing loaded module `RtlpxLookupFunctionTable` will always find the host module first and return its exception directory pointer. The binary search over the static `.pdata` returns NULL for the shellcode address, but crucially `RtlpLookupDynamicFunctionEntry` is never reached. Since `RtlVirtualUnwind` is called with the `RUNTIME_FUNCTION` pointer obtained beforehand via a direct lookup, it still operates correctly and the unwinder itself never re-invokes the lookup, it just applies the unwind codes it was handed.

The failure of `StackWalk64` and by extension any tool relying on `SymFunctionTableAccess64` is caused by a different resolution path. Contrary to what I was expecting, `SymFunctionTableAccess64` does not seem to delegate to `RtlLookupFunctionEntry`. Instead, as can be observed by reversing `SymFunctionTableAccess64AccessRoutines` inside `dbghelp.dll`, it maintains its own internal module database populated during `SymInitialize`. When asked to resolve a function table entry for a given address, it first queries this internal database to identify the owning module:

```
// SymFunctionTableAccess64AccessRoutines (reconstructed)
moduleEntry = FindModuleInDbghelpDatabase(processHandle, address);

if (moduleEntry != NULL &&
    address >= moduleEntry->imageBase &&
    address < moduleEntry->imageBase + moduleEntry->imageSize) {

    // Module found in dbghelp's own database
    // Perform direct binary search into cached .pdata - never calls
RtlLookupFunctionEntry
    return BinarySearchCachedExceptionDirectory(
        address - moduleEntry->imageBase,   // RVA
        moduleEntry->exceptionDirectory,    // cached .pdata pointer
        moduleEntry->exceptionDirectorySize);
}

// Only reaches here if dbghelp has no module record for this address
// This is the path that eventually calls RtlLookupFunctionEntry
return FallbackDiaOrRtlLookup(processHandle, moduleEntry, address);
```

Since the shellcode lives inside an injected module's address range, `dbghelp` correctly identifies the module as the owner, retrieves its cached exception directory, performs its own binary search, finds no entry covering the shellcode address, and returns NULL. The dynamic registration via `RtlAddFunctionTable` is never consulted. This might be an intentional design where the symbol engine owns its resolution pipeline independently of the OS unwinder, however I couldn't find clear references about this behaviour in the official documentation.

The practical consequence is that any stack walk initiated through the `dbghelp` APIs will fail to unwind frame 0, producing a broken call stack display, even though the exact same scenario handled natively by the OS exception dispatcher and `RtlVirtualUnwind` works perfectly. This can be visualized as two completely parallel resolution pipelines that never communicate:

```
OS Exception Dispatcher / RtlVirtualUnwind
    └── RtlLookupFunctionEntry
            └── RtlpxLookupFunctionTable  (checks loaded modules)
                    └── [miss] RtlpLookupDynamicFunctionEntry  ← finds the dynamic entry
                    └── [hit]  BinarySearch(.pdata)            ← misses the dynamic entry

StackWalk64 / SymFunctionTableAccess64 / WinDbg k / SystemInformer
    └── dbghelp internal module database
            └── [hit]  BinarySearch(cached .pdata)             ← always misses the dynamic entry
            └── [miss] FallbackDiaOrRtlLookup                  ← would find the dynamic entry
                                                                    but never reached
```

This means, trivially, that any external tool (i.e., Eclipse), executing `StackWalk64` in its default fashion will always see a cut stack and be in a perfect situation to detect a malicious behaviour.

```
C:\>Eclipse /pid 27064 /trace

  [Thread 44432] Stack trace:
    #0    0x00007FF97ADB15F4  NtUserWaitMessage  (+0x14)
    #1    0x00007FF97BC3170E  DialogBoxIndirectParamAorW  (+0x37E)
    #2    0x00007FF97BC3157C  DialogBoxIndirectParamAorW  (+0x1EC)
    #3    0x00007FF97BCAB2A6  SoftModalMessageBox  (+0x826)
    #4    0x00007FF97BCA9C09  DrawStateW  (+0x25D9)
    #5    0x00007FF97BCAA9E8  MessageBoxTimeoutW  (+0x198)
    #6    0x00007FF97BCAA7E8  MessageBoxTimeoutA  (+0x108)
    #7    0x00007FF97BCAA3FE  MessageBoxA  (+0x4E)
    #8    0x00007FF97D3428BA  RtlRunOnceExecuteOnce  (+0x9A)
    #9    0x00007FF97A929EE1  InitOnceExecuteOnce  (+0x21)
    #10   0x00007FF97895BC74  SHOpenFolderAndSelectItems  (+0x9D1A4)
[!][udinject.exe][TID 44432] Cut stack detected: RtlUserThreadStart was never reached. The stack has likely been spoofed
 or truncated.
[!][udinject.exe] Suspicious return address. Alerts: (1)
```

The Stack Cookies

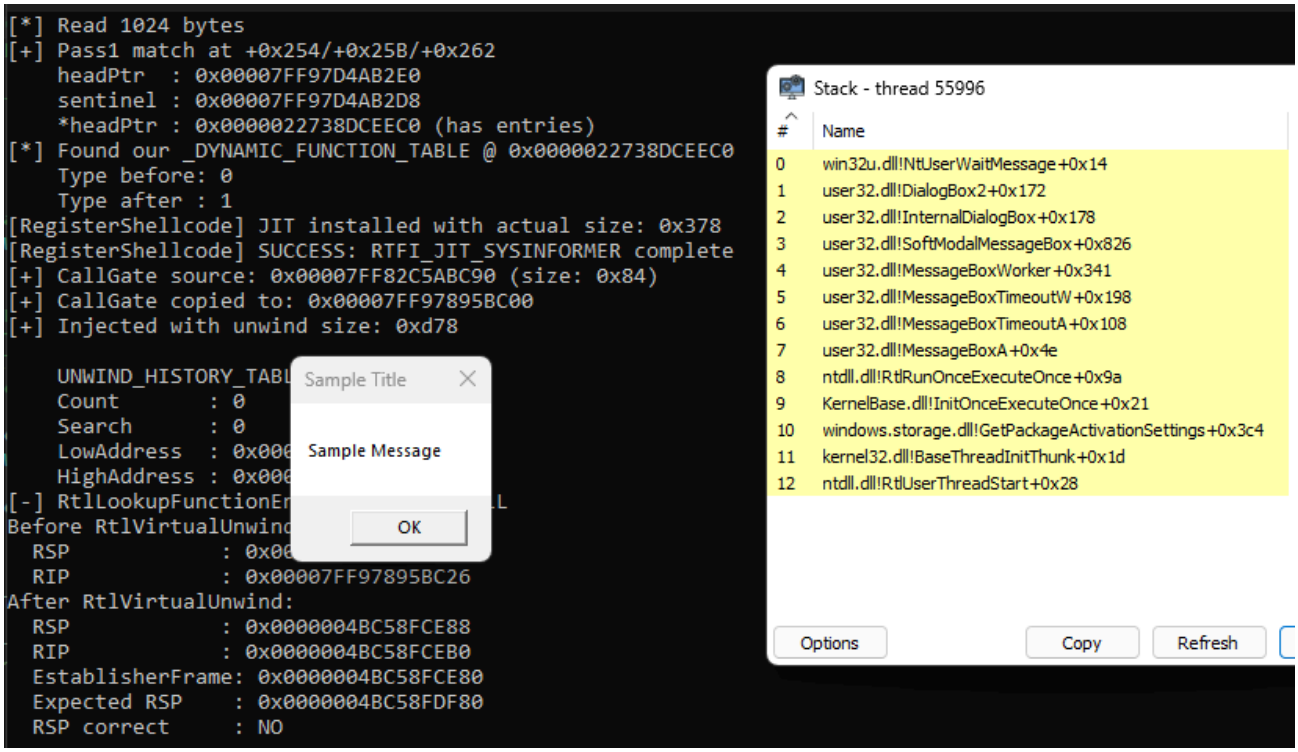
Before Reading: You can, for your own sake, just compile `BYOUD` with `/GS-` and the following techniques will work independently against System Informer and WinDbg.

After a bit of struggling reversing `RtlpxLookupFunctionTable`, I found that, even with Stack Cookies enabled, there were some cases where I could, potentially, make the stack resolve correctly in System Informer XOR WinDbg (this is not a joke, I never managed to get the stack resolution right in both simultaneously).

```
if (LdrInitState == 3) {
    RtlAcquireSRWLockShared(&LdrpInvertedFunctionTableSRWLock);
    /*
      binary search LdrpInvertedFunctionTable for param_1
    */
    if (found) {
        RtlReleaseSRWLockShared(...);
        return result;           // found in static table, return
immediately
    }
    RtlReleaseSRWLockShared(...);
    if (DAT_18019b52c == 0)
        return 0;                // frankly not super sure yet as to what
this is
    // fall through to VirtualQuery path
}
// VirtualQuery fallback (early boot or overflow)
lVar6 = RtlpGetImageBaseViaQueryVirtualMemory(param_1, ...);
if (lVar6 != 0) {
    RtlCaptureImageExceptionValues(lVar6, ...);
    return result;
}
return 0;
```

To obtain a correctly resolved stack in SystemInformer, I had to invalidate the fast-path cache. Before calling `RtlpxLookupFunctionTable`, `RtlLookupFunctionEntry` checks a small cache storing the last successful lookup's module base and exception directory size. If the size is non-zero and the requested RIP falls within the cached range, `RtlpxLookupFunctionTable` is never called at all. Zeroing the cached size forces the full lookup path.

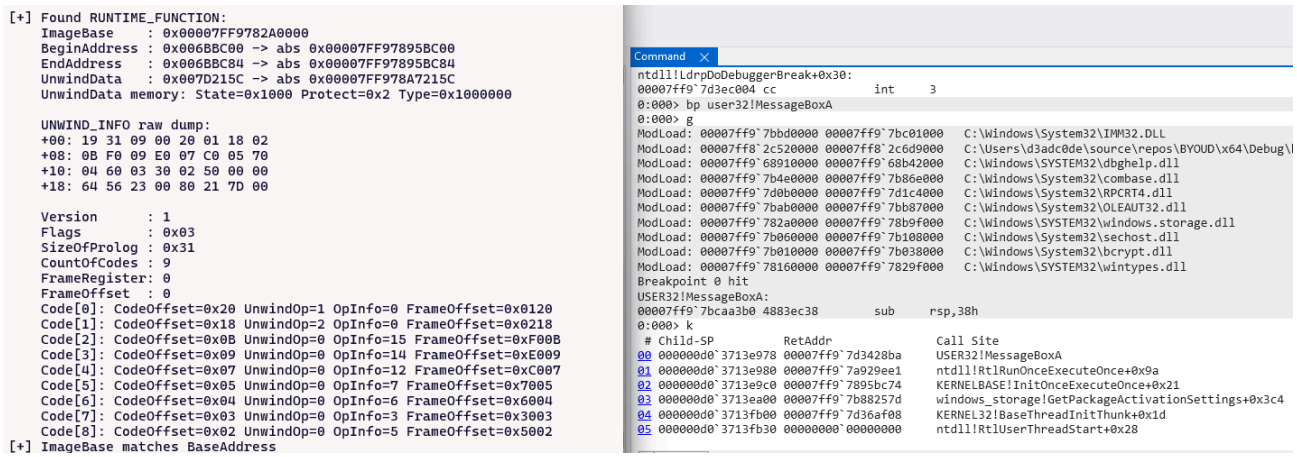
> You can test this with BYOUD using RTFI_JIT_SYSINFORMER, the result once MessageBoxA has been triggered should be the following:



However, this will not work from a WinDbg point of view, which needs another two patches in order to resolve the stack correctly (if the `BYOUD` dll is compiled with `/GS`, otherwise, the cache trick is enough):

- First, I had to shrink the module's recorded `ImageSize` in `LdrpInvertedFunctionTable`. The binary search compares the target RIP against each entry's ImageBase and ImageBase + ImageSize. If the shellcode's address falls beyond the shrunken size, the module is not matched, the binary search misses, the overflow flag is zero, and the function returns 0. Again, this should force the lookup to the dynamic table.
- But this is still not enough. Another step needed (well, empirically, this was the only one working) was to change the protection of the full `.pdata` section to `PAGE_NOACCESS`. To my surprise, making `.pdata` inaccessible causes that read to fault and return 0, restoring the miss condition and keeping the dynamic table reachable.

> You can test this with BYOUD using RTFI_JIT_WINDBG, start the process directly from WinDbg, set a breakpoint on
>
> ```
> bp user32!MessageBoxA
> ```
>
> The stack should look like this:



Of course, other reliable solutions (apart from disabling stack cookies) are either to place shellcode in a `VirtualAlloc`'d region outside all loaded module ranges, causing the module lookup to miss and fall through to the fallback path that does invoke `RtlLookupFunctionEntry`, or to patch the host module's static `.pdata` directly, making the binary search find the entry organically. This, however, will make the dynamic registration unnecessary, as we've seen in the previous attack techniques.

JIT BYOUD + UNWIND_DATA Create (Shellcode Residing on the HEAP)

When shellcode is placed in a `VirtualAlloc`'d region outside the address range of any loaded module, the dynamic function table registration via `RtlAddFunctionTable` works correctly end to end. The reason should be that `dbghelp` internal module lookup fails to find any owner for the address, causing `SymFunctionTableAccess64` to fall through to its fallback path which eventually invokes `RtlLookupFunctionEntry`. At that point the OS dynamic table is correctly consulted, the registered `RUNTIME_FUNCTION` is found, and `RtlVirtualUnwind` successfully unwinds the frame. `StackWalk64` produces a complete and accurate call stack with correct `Child-SP` and `RetAddr` values at every frame. The unwinding is correct. However, the shellcode frame will appear as a bare hexadecimal address rather than a named symbol:

```
# Child-SP              RetAddr               Call Site
00 0x000000812F0FDF90   0x00007FFB78425971    0x00007FFB90D46C26        <- no name
01 0x000000812F0FE910   0x00007FFB70A1C3D2    kernel32!BaseThreadInitThunk+0x14
02 0x000000812F0FE940   0x0000000000000000    ntdll!RtlUserThreadStart+0x21
```

This is purely a symbol resolution issue, not an unwinding issue. `SymFromAddr` is called for the shellcode's `RIP` value, finds no matching symbol in any loaded PDB or export table, and returns the raw address. The stack itself is structurally valid, every frame is correctly computed, every

return address is valid, and the full chain back to `BaseThreadInitThunk` is intact. The distinction matters: a broken stack means frames are missing or return addresses are garbage; an unnamed frame means the stack is correct but the address has no associated debug information. This is the expected and acceptable behavior for dynamically generated code, and is identical to what you see with JIT compilers: the V8 engine, the CLR JIT, and similar runtimes all produce correctly unwinding but unnamed frames in stack traces until explicit symbol registration via `SymLoadModuleEx` or `SymAddSymbol` is performed to associate a name with the address range.

Important Considerations

JIT compiled code is not image backed by default. When registering exceptions dynamically using `RtlAddFunctionTable` (or similar APIs, as there are a few offering similar functionalities) for code segments that are image-backed, we create indicators of compromise that could be used for detection.

When using `RtlAddFunctionTable` (even in combination with BYOUD), at least one of the caller frames will remain observable in the call stack. By resolving any of the executing frames, a defender could easily recover the address of the runtime function table in memory.

This technique involves mimicking JIT compilation. It could provide a useful primitive when used in combination with RWX hunting techniques. RWX memory areas are often offered by JIT-enabled processes and will provide natural memory regions where both dynamic exception registration and BYOUD can be used together, minimizing the noise generated.

All these techniques offer chances for detection.

BYOUD: Full CET Compliance via State-Driven Architecture

In response to research by Gabriel Landau at Elastic Security Lab on shadow stack analysis, based on previous research by A. Ionescu and Y. Shafir on CET internals, I've decided to take on one final task. Our goal was to answer this question: Is it possible to design an architecture that complies with CET and also produces a call stack that perfectly mirrors the shadow stack?

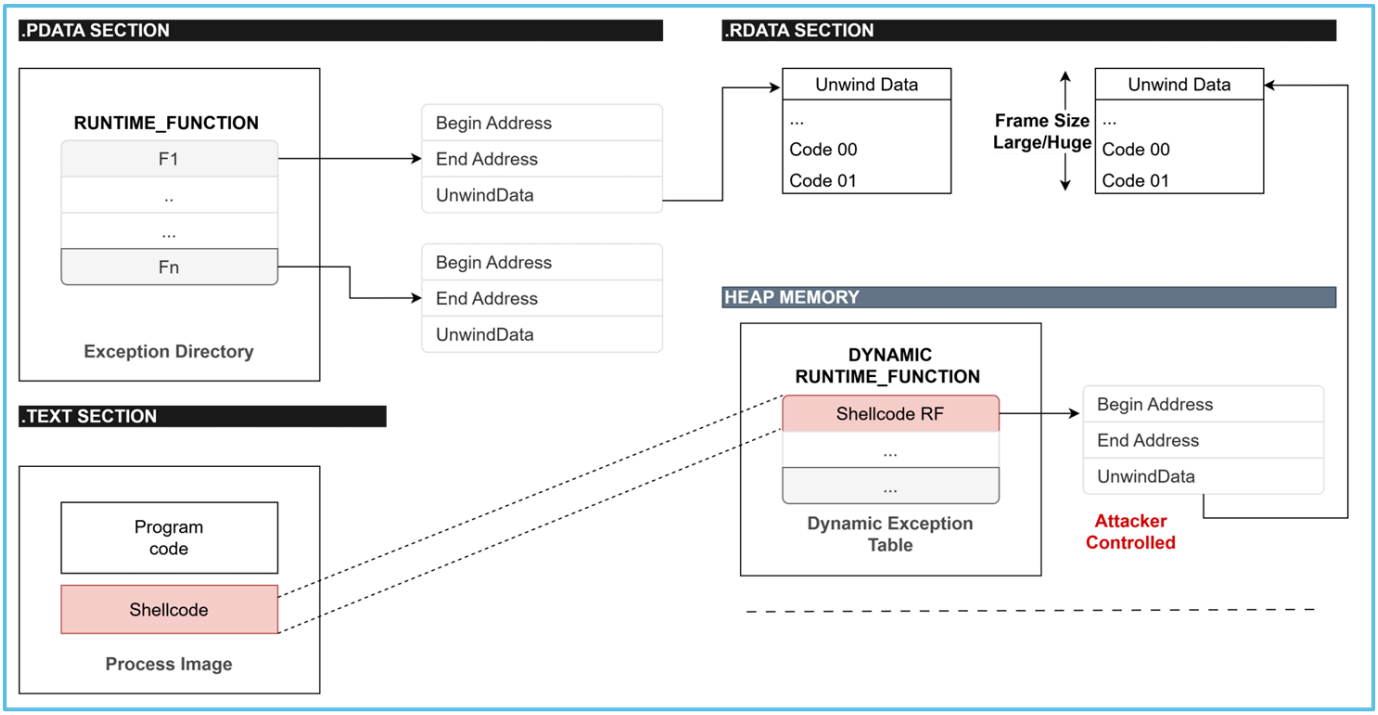
After extensive work on the proxy architecture behind BYOUD, I'd say yes, it is feasible, though far from straightforward. I personally consider this approach to be somewhat of a cheat workaround.

Architecture Overview

The architecture follows what we did for the Full Exception Setup, with one major difference. In the proxy architecture, our main shellcode was completely free from constraints and could perform arbitrary calls, establishing the required intermediate function frames. For any API call, the shellcode would use the proxy allocator to equalize the stack allocation to the registered frame size, concealing all the intermediate frames.

Even under CET, this architecture is usable and shows a spoofed stack. However, this creates inconsistencies between the regular call stack and the shadow stack.

We can adapt the proxy architecture to handle this case, though it requires rethinking how we execute our shellcode.



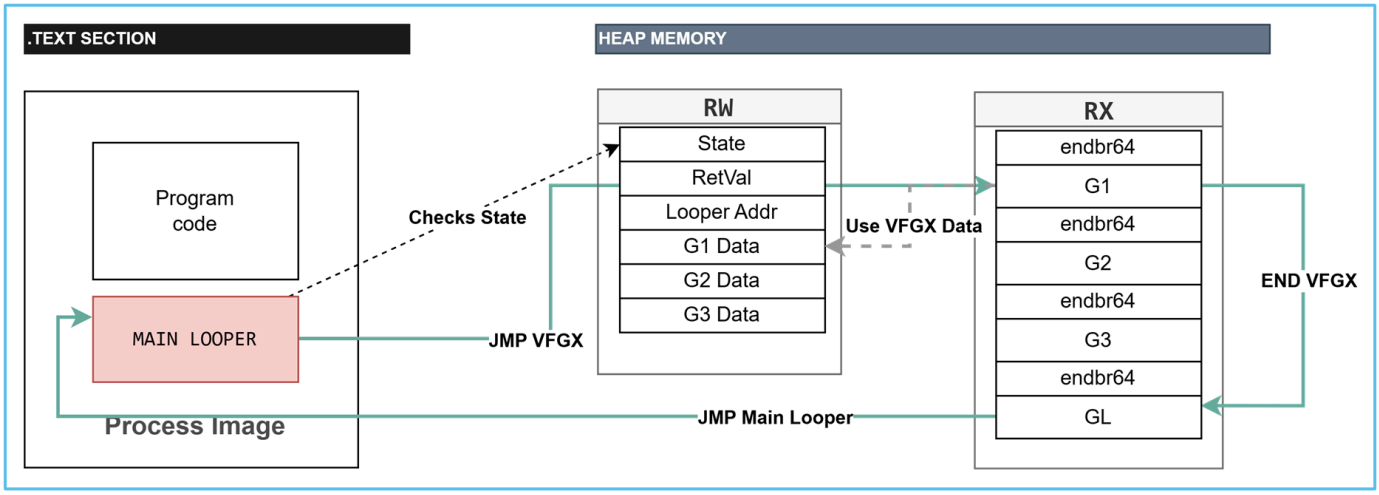
State-Driven Indirect Execution Architecture (SDIE)

The core idea is to transform the proxy function into a proxy/dispatcher, while any other shellcode or implant injected into memory consists of a sequence of virtual gadgets. A third component (virtually allocated RW memory) holds the state and operation descriptors (SOPD).

These three components form a stateful architecture: the proxy-dispatcher reads the current state, dispatches to the appropriate virtual gadget, updates the state, and repeats until the terminate state restores context and returns. We provisionally named this State-Driven Indirect Execution Architecture (SDIE).

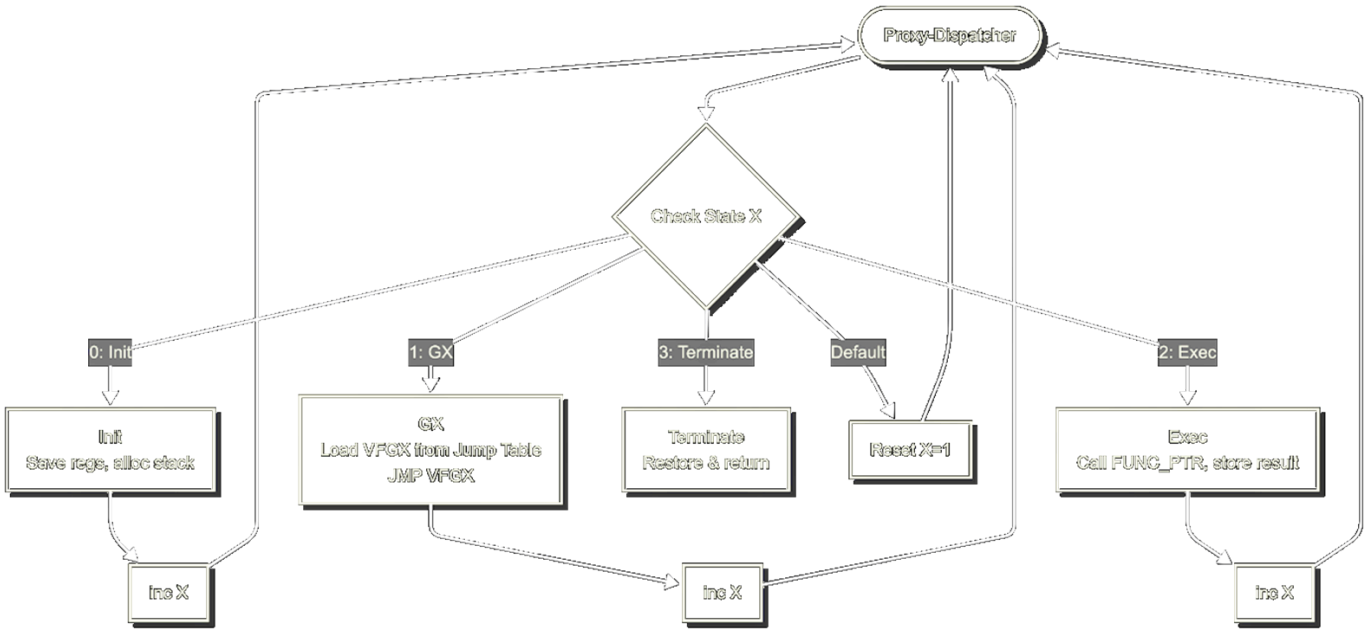
The gadgets consist of a sequence of one or more valid branches, starting with an `endbr64*` instruction. Each gadget then performs a short jump to a default gadget, which in turn executes an indirect jump to the beginning of the main looper function.

> endbr64* = selected only because it's basically treated as a NOP in Windows, completely irrelevant from a SS/IBT perspective, but it allows us to identify a specific chunk by index, instead of by address

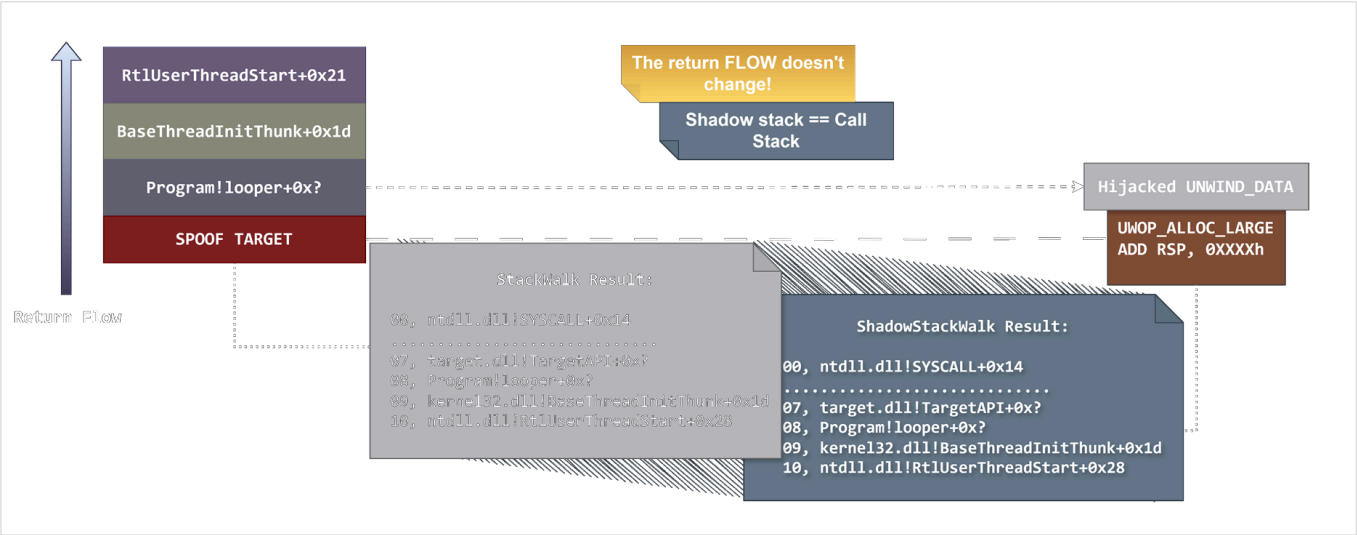


In our proof-of-concept, the resulting state machine resembles a simple dispatcher loop, cycling through a set of well-defined states that drive execution forward. Each state corresponds to a specific operation:

1. Initialization: Allocate the frame stack to match the registered `RUNTIME_FUNCTION` metadata
2. Gadget select and dispatch: Based on data recovered from the SOPD Structure, select a gadget and perform an indirect JMP to it
3. Execution: Call an API. After the call, recover the pointer to the SOPD Structure, save the return value, update the state, and loop
4. Termination: Deallocate the frame and return



The resulting call stack when an Exec state is reached:



DISCLAIMER: I have decided not to release this, as I frankly think the implementation was lacking the level of detail needed to be actually a useful learning experience. I might release in the future if I have some more time to dedicate to it.

Reflections on Complexity

This approach resembles a COOP-style exploit. The proxy-dispatcher plays the role of a LOP "main gadget," driving execution by repeatedly selecting the next action. The jump table serves as a collection of virtual function gadgets (vfgadgets), much like in the original COOP paradigm, where control is steered through a predefined chain of gadget calls to achieve the desired behavior.

As both hardware and software mitigations become widely adopted, creating effective evasion techniques will get more difficult. Malware authors will probably find it safer to implement more natural execution flows that resemble legitimate software rather than traditional exploits.

Detection Landscape

In the following paragraphs, we will discuss potential detection strategies for these techniques.

Detection Mechanisms

The detection landscape for this technique can be addressed through integrity monitoring of the `.pdata/.xdata` and `.rdata` sections in memory. These sections, typically marked as read-only, are not expected to change throughout the lifetime of a process. Active integrity checks or snapshot comparisons during runtime would be sufficient to detect tampering attempts.

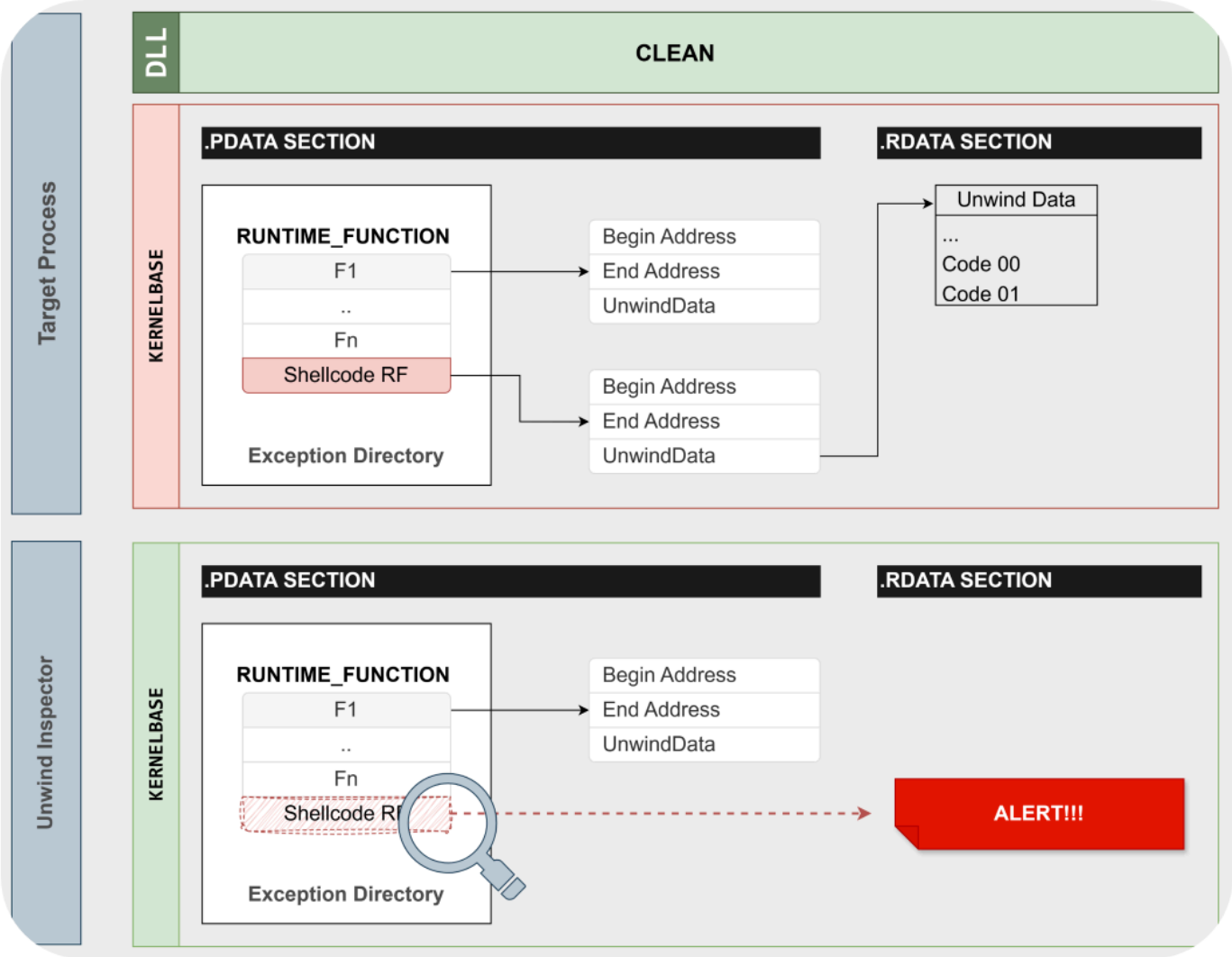
Active monitoring of memory protection changes involving these sections should be considered. Preventing the remapping of `.rdata` or `.pdata` to writable permissions would significantly hinder the feasibility of overwriting the critical runtime structures that this attack relies on.

UnwindInspector: Detection Tool

As a detection proof-of-concept, we developed an updated version of UnwindInspector, extending its functionality beyond offensive research. The tool can now be used to inspect and analyze modifications in the Exception Directory, making it a practical resource for detecting tampering and validating the integrity of `RUNTIME_FUNCTION` data in loaded modules.

The inner workings of the tool are straightforward. It opens a handle to the target process and performs a diff of the `RUNTIME_FUNCTION` entries for each loaded module, comparing them against a clean reference copy of the same module loaded independently. The tool triggers an alert under the following conditions:

1. `RUNTIME_FUNCTION` entry count mismatch: suggesting a rogue RF has been added or removed
2. `RUNTIME_FUNCTION` entry `UNWIND_DATA` address mismatch: suggesting the hijacking of an existing dataframe
3. `RUNTIME_FUNCTION` entry begin/end function offset mismatch: suggesting an existing entry has been assigned to another code segment
4. `UNWIND_DATA` content mismatch: suggesting tampering has taken place



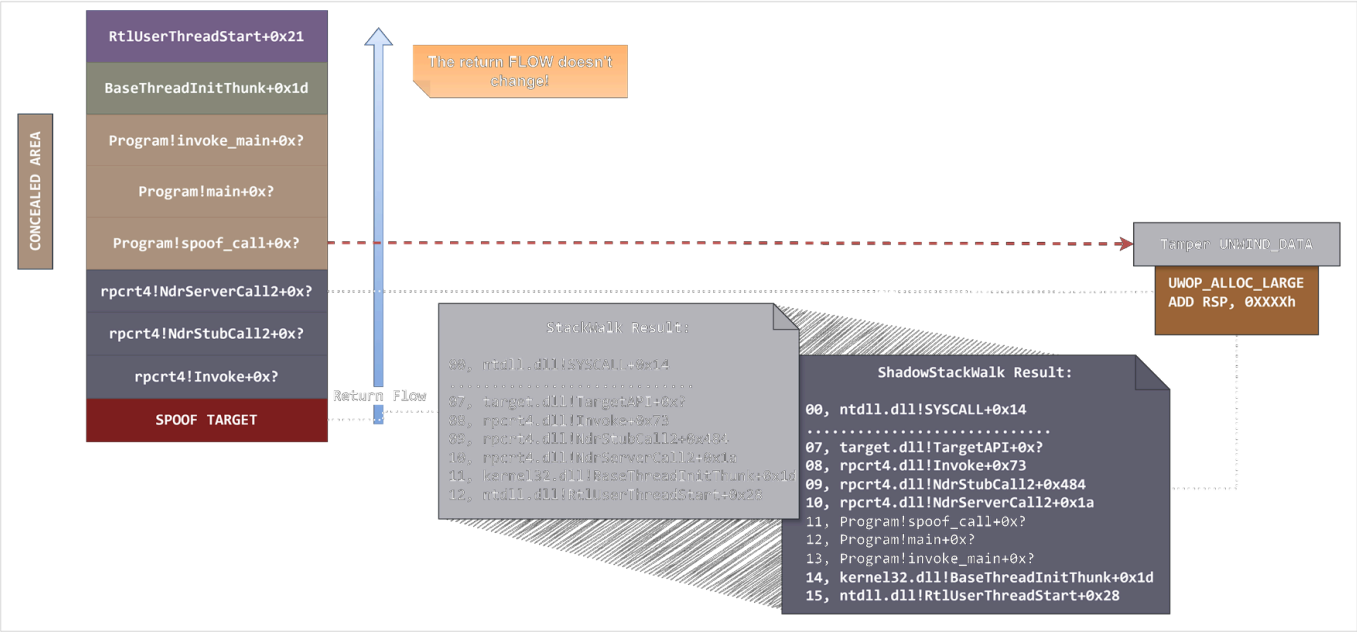
Detection for the JIT Case

For the variant of the technique that abuses Dynamic Exception Registration, we provide an extension to our previously released Eclipse tool. The tool adds an additional check to validate the address of the `RUNTIME_FUNCTION` table. The checked conditions:

1. **Rogue Table Entry**: The runtime function pointer is on the heap, but the function address points to an image-backed memory region
2. **Possible JIT malware**: The runtime function pointer and the base address for the table are both on the heap

Shadow Stack Enhanced Detection

On systems with CET enabled, while the technique is CET-compliant, this does not imply invisibility under CET enforcement. The shadow stack will still contain the actual return addresses pushed during execution, even if the primary stack is spoofed with fake unwinding data. The inconsistency between the standard call stack and the shadow stack remains a viable detection vector.



Security solutions should adopt a hybrid inspection model, combining traditional stack walking with shadow stack verification to accurately reconstruct the execution path and detect anomalies introduced by stack spoofing techniques. This approach has been implemented by Elastic Security Labs in their research on shadow stack analysis, with accompanying tooling available on GitHub.

Final Remarks & Open Questions

The aim of this research was to redesign stack spoofing by addressing the emerging constraints introduced by Hardware-enforced Stack Protection (HSP) and Intel CET. The techniques I presented in the series of posts, particularly the BYOUD framework, demonstrate that stack spoofing can shift from return address manipulation and operate within CET-compliant environments as well.

BYOUD variants enable the construction of spoofed call stacks that don't trigger CP faults while still tricking the Windows Unwinding mechanics. Through manual, dynamic code injection and exception information generation, it's possible to create seemingly legitimate, unwindable code paths without violating execution integrity.

Open Questions

Can we create a unified spoofing architecture that is simultaneously CET-resilient, unwind-valid, and allows for return value preservation while maintaining modularity and general applicability across execution contexts such as thread pools, shellcode, or proxy calls? While this work provides strong groundwork, answering that question definitively requires further exploration.

Another question is whether pursuing such techniques remains worth the effort. The strategic value of stack spoofing as a primary evasion mechanism appears to be in decline. As defenders grow more adept and architectural safeguards like CET become ubiquitous, the broader security research community is increasingly shifting toward models that bypass the need for stack manipulation entirely. While edge-case exploration and complexity remain intellectually compelling, the direction forward may lie in adopting evasive paradigms that are inherently less dependent on the stack as an observable attack surface.

Future Directions

Although HSP and CET adoption is increasing, legacy systems without hardware support will remain prevalent for years. Meanwhile, techniques that align with CET design offer fertile ground for research. Continued exploration into runtime exception handling, unwind metadata semantics, and indirect invocation primitives may yield the next generation of stealthy, post-CET stack manipulation strategies.

If you managed to arrive here, thanks for reading so far.

POC||GTFO

In case you want to play around with the techniques, a POC is available on GitHub:

> BYOUD

References

- x64 return address spoofing by namazso
- ShadowStackWalk by Gabriel Landau
- R.I.P ROP: CET Internals in Windows 20H1
- Microsoft: x64 prolog and epilog
- Microsoft: x64 exception handling
- The Stack Series: Return Address Spoofing on x64
- Finding Truth in the Shadows by Gabriel Landau
- DreamWalkers by Maxime Dcb
- SilentMoonwalk: Implementing a dynamic Call Stack Spoofer
- Unraveling an RPC Thread - RpcProxyInvoke

Acknowledgments

I would like to acknowledge the contributions of all researchers whose work has informed this research, particularly Gabriel Landau at Elastic Security Labs for the shadow stack analysis research, Alex Ionescu and Yarden Shafir for their CET internals work.

Last but not least, I would like to thank namazso, because his original work on stack spoofing has laid the groundwork for all current research on the topic.