Callback hell: abusing callbacks, tail-calls, and proxy frames to obfuscate the stack

klezvirus.github.io/posts/Callback-Hell

December 21, 2025

```
0:004> k
 # Child-SP          RetAddr               Call Site
00 000000d4`d7dff378 00007ff7`03921dab     KERNEL32!LoadLibraryAStub
01 000000d4`d7dff380 00007ffd`c391287a     ThreadPoolExec!ThreadPoolCallback2+0x7b
02 000000d4`d7dff500 00007ffd`c38e5e46     ntdll!TppWorkpExecuteCallback+0x13a
03 000000d4`d7dff550 00007ffd`c24f257d     ntdll!TppWorkerThread+0x8f6
04 000000d4`d7dff830 00007ffd`c390af08     KERNEL32!BaseThreadInitThunk+0x1d
05 000000d4`d7dff860 00000000`00000000     ntdll!RtlUserThreadStart+0x28
```

Foreword

Once upon a time, my friend **Athanasios Tserpelis**, aka trickster0, decided to give me a call with a great problem on his hands:

> I'm using TpAllocWork + TpPostWork to execute an arbitrary function, but I'm not fully sure how to recover the return value. Any ideas?

That question reminded me of some experiments I was working on previously, but had set aside out of laziness. I decided to revisit it and give it another shot. What you see in this blog post is the result of that renewed effort, an attempt to solve the problem, which I later integrated into my stack spoofing research.

This specific work did not make it into the final talk for several reasons, mainly because I don't believe it materially advances the research or provides meaningful additional utility.

Still, I'm publishing the results here for completeness.

Definitions

Before "jumping" into the topic itself, let's give some definitions to ease readability.

Tail calls

A tail call is a subroutine call performed as the final action in a function. If a function calls another function as its last operation (i.e., it returns the result of that call directly), this is a tail call.

At the assembly level, a tail call is typically implemented using a jmp instruction, not a call.

In a normal function call, the `call` instruction pushes the return address onto the stack nad transfers control to the target function.

But in a tail call, since the current function is done and its return address is no longer needed, we can reuse the current stack frame. A `jmp` instruction achieves that, as it does not push a new return address and transfers control directly to the callee.

This eliminates the stack growth that would otherwise occur with deeply recursive calls.

Callbacks

A callback is a function passed as an argument to another function, intended to be called at a later time. Typically, callbacks are executed after a specific event or operation completes.

- Callbacks are fundamental to asynchronous programming, event-driven systems, and APIs.
- They decouple the caller from the callee, allowing for extensibility and inversion of control.

ROP Gadget

A ROP gadget is a short sequence of existing machine instructions ending in a ret (or similar control-transfer) that attackers chain together to perform arbitrary operations without injecting code, thereby bypassing protections like DEP/NX.

JOP/COP Gadget

A JOP gadget (Jump-Oriented Programming) or COP gadget (Call-Oriented Programming) is a short sequence of legitimate instructions ending in an indirect `jmp` or `call` (instead of `ret` as in ROP), which attackers chain using controlled pointers to achieve arbitrary execution while bypassing return-based protections like shadow stacks or control-flow integrity.

Ad-hoc Definitions

The following definitions are “arbitrary”, meaning are used in an ad hoc manner for the purposes of this post and may not reflect a broadly accepted consensus.

Forward Proxy Frame

A Forward Proxy Frame is essentially a JOP or COP gadget that is executed using its own crafted stack frame, where the control flow transition to the gadget occurs via a forward edge. The gadget is invoked directly through an indirect jump or call, not via a return instruction.

A valid example of a frame like this would be:

```
call REG <-- we place RIP here via CONTEXT or reach it via JMP REG
pop rbx
add rsp, 20
ret
```

Frames like this are not really easy to find in general, which means that we will extend the definition to other functions that can serve as a proxy (e.g., `NdrClientCall3`, `NdrServerCall2`, etc.).

Backward Proxy Frame

A Backward Proxy Frame is a ROP gadget executed using its own dedicated stack frame, where the control flow reaches the gadget via a backward edge—meaning, the gadget is triggered through a return instruction, consistent with traditional return-oriented programming semantics.

In a nutshell a ROP gadget where we allocate the corresponding frame, and we let the original epilog unwind it.

> For readers who followed any of the Moonwalk blogposts, it should be easy to see that any CONCEAL gadget is indeed a valid backward proxy frame.

A valid example of a frame like this would be:

```
call 0xaddress
pop rbx <-- we push the address of this instruction as return address and let it execute on return
add rsp, 20
ret
```

It is very clear that while forward frames are CET compliant (the CALL performs the setup of the return address), backward frames are not, as the return address is created artificially.

How callbacks appear in real life

A persistent limitation in call stack detection and general user-defined callbacks is that the memory region executed by the thread worker must reside in memory, either embedded in a module (via stomping) or within a dedicated RX/RWX private memory area. As a result, the callback address inevitably appears in the call stack, making it susceptible to inspection and detection.

We can see below that the callback address is perfectly visible in the call stack.

```
0:004> k
 # Child-SP          RetAddr               Call Site
00 000000d4`d7dff378 00007ff7`03921dab     KERNEL32!LoadLibraryAStub
01 000000d4`d7dff380 00007ffd`c391287a     ThreadPoolExec!ThreadPoolCallback2+0x7b
02 000000d4`d7dff500 00007ffd`c38e5e46     ntdll!TppWorkpExecuteCallback+0x13a
03 000000d4`d7dff550 00007ffd`c24f257d     ntdll!TppWorkerThread+0x8f6
04 000000d4`d7dff830 00007ffd`c390af08     KERNEL32!BaseThreadInitThunk+0x1d
05 000000d4`d7dff860 00000000`00000000     ntdll!RtlUserThreadStart+0x28
```

One potential mitigation is to convert the callback into a no-frame function, using a pattern similar to tail call optimization. We can do so by implementing the callback as pure MASM. By doing so, we could pop the immediate return address, prepare the parameters to accept more than the standard 3 parameters usually accepted by a Thread Worker Callback, and by replacing the `CALL` instruction with a `JMP` to the actual target function.

> Abusing callbacks and tail calls to hide the original caller from the stack is not a new concept at all, and there is quite a bit of public research on the topic already. Most notably, Chetan Nayak, the creator of Brute Ratel C2, published a blog post about it: Hiding In PlainSight - Indirect Syscall is Dead! Long Live Custom Call Stacks
>
> In a nutshell, it is known to be possible to use a frameless callback, built in a way where the real function to invoke is invoked as a tail call, in this way:

```
section .text

global WorkCallback

WorkCallback:
   mov rbx, rdx                ; backing up the struct as we are going to stomp rdx
   mov rax, [rbx]              ; NtAllocateVirtualMemory
   mov rcx, [rbx + 0x8]        ; HANDLE ProcessHandle
   mov rdx, [rbx + 0x10]       ; PVOID *BaseAddress
   xor r8, r8                  ; ULONG_PTR ZeroBits
   mov r9, [rbx + 0x18]        ; PSIZE_T RegionSize
   mov r10, [rbx + 0x20]       ; ULONG Protect
   mov [rsp+0x30], r10         ; stack pointer for 6th arg
   mov r10, 0x3000             ; ULONG AllocationType
   mov [rsp+0x28], r10         ; stack pointer for 5th arg
   jmp rax
```

```
 # Child-SP          RetAddr               Call Site
00 000000ad`aa3ff618 00007ffd`c391287a     KERNEL32!LoadLibraryAStub
01 000000ad`aa3ff620 00007ffd`c38e5e46     ntdll!TppWorkpExecuteCallback+0x13a
02 000000ad`aa3ff670 00007ffd`c24f257d     ntdll!TppWorkerThread+0x8f6
03 000000ad`aa3ff950 00007ffd`c390af08     KERNEL32!BaseThreadInitThunk+0x1d
04 000000ad`aa3ff980 00000000`00000000     ntdll!RtlUserThreadStart+0x28
```

While this effectively removes the callback frame from the call stack, it comes at the cost of losing the return value from the invoked function. And in many contexts this is an unacceptable trade-off, particularly given that callbacks (in this case via thread pools) do not natively support return value retrieval after a worker's execution completes.

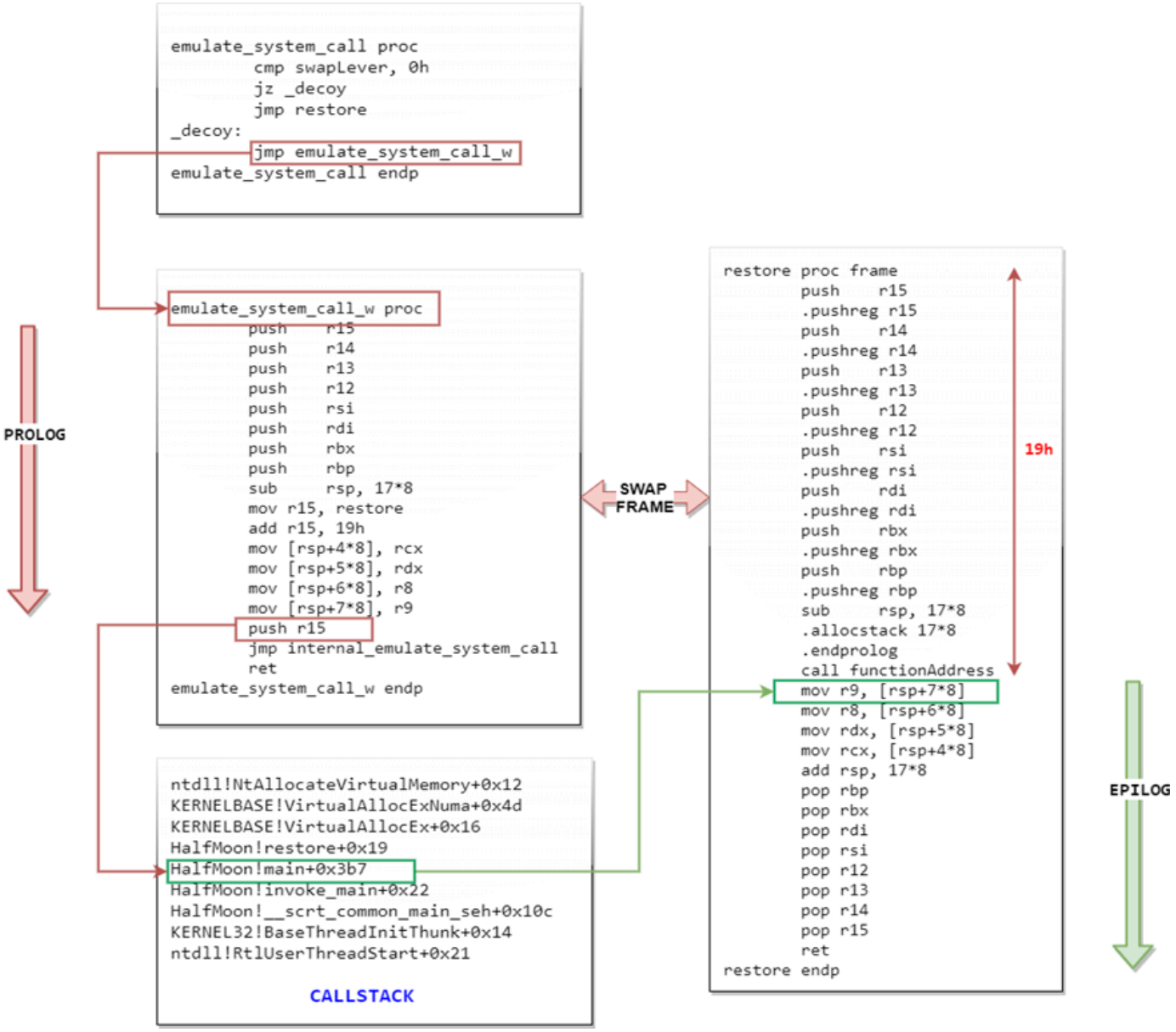
Initial Frame Swapping Design

During the development of the stack moonwalking techniques, we considered the algorithm's resilience to manual inspection. To strengthen the technique to human inspection, we first designed an opaque architecture for the calls inside the main executable, consisting of a conditional trampoline and an arbitrary function invoker. The conditional trampoline is set up to

“trick” the call stack inspector about which branch was taken when the function was called. Since it is impossible to know the value of a variable at a specific past-time T, the branch can only be inferred by examining the subsequent frames in the call stack, which can be controlled.

The arbitrary function invoker, on the other hand, uses a function pointer variable to “pretend” that the function called by the main executable was indeed the High-Level API. In the implementation, this is achieved by using two functions: an illegal frameless function and a standard framed function.

The frameless function is the one executed by the program and prepares the stack to hold the legitimate frame function. The framed function is never executed in its entirety. Its main task is to validate the stack space created by the illegal function, validate the return address, validate the call flow to the high-level API, and execute a restore routine before returning to the main executable code.



In the described scheme, `emulate_system_call` serves as the opaque trampoline, or dispatcher, while `emulate_system_call_w` functions as the frameless function. The restore function is replaced by the restore function in the call stack, with the return address set to the instruction `restore+19h`, which is the instruction following the call functionAddress. The variable functionAddress, ultimately, holds the fake function pointer to the High-Level API, as in the original architecture.

The final call stack can be observed in the figure, along with the program execution flow, which should help clarify the roles of both the frameless and framed functions. The frameless function essentially functions as the frame function prologue, creating its frame and positioning the correct return address. The restore function, on the other hand, executes only its epilogue, restoring all saved registers and deallocating the stack space that was previously allocated.

> Not really important in our artificial scenario, but mirroring the prolog should have the added benefit of avoiding strange side effects if an exception in raised in the function.

The new architecture indeed adds a layer of obfuscation to the program code, making it harder to analyze and detect the stack spoofing techniques being used. The dispatcher can be extended to include multiple conditional jumps, which can further complicate the execution flow and make it more challenging to determine the actual path taken. Additionally, the frameless function can be fragmented and obfuscated without affecting the framed function, providing even more flexibility in hiding the true nature of the program.

However, since the half moonwalk technique involved only a partial stack spoof, the added complexity it introduced was ultimately deemed disproportionate to the limited benefits it provided.

Frame Swapping Proxy Using a Thread Pool

While the overall architecture may be considered overly complex relative to its immediate benefits, it introduces a primitive with broader applicability: frame swapping (aka “let’s stick another lame name to something we’ve been doing since ret address spoofing was invented”).

This technique proves highly versatile in scenarios where a developer seeks to proxy arbitrary function calls using a callback-style mechanism. A comparable paradigm can be observed in thread pool execution models, such as those first explored by SafeBreach Labs here.

Our little experiment found that frame swapping resolves this dilemma quite effectively. It enables the concealment of the callback frame within the call stack while still preserving the ability to retrieve the return value from the proxied function, offering a powerful evasion and control flow primitive for callback-based execution models.

For this technique to function as intended (i.e., hiding the callback frame while preserving the return value) we must identify function patterns that store the return value (typically in `RAX`) into a memory location via another register (used as a pointer), just before the function epilog.

Multiple examples of this pattern can be found in several system DLLs. However, to maximize the legitimacy of the call stack, we would need to select a frame directly from the image base of the program where we are injecting code into.

For the sake of giving an example, this is the EPILOG of the function `GlobalGetUserAndPassW` in `wininet`:

```
180152087 48 89 03        MOV       qword ptr [RBX],RAX
18015208a 48 83 c4 20     ADD       RSP,0x20
18015208e 5b              POP       RBX
18015208f c3              RET
```

From this snippet, we observe that the function expects a reference to a 64-bit variable, which it uses to store the return value produced by a preceding `CALL`. The total frame size is straightforward to compute based on the epilogue: `0x20` bytes released by the `ADD RSP, 0x20` instruction, plus an additional 8 bytes accounted for by the `POP` instruction, for a total of 0x28 bytes.

Following the pattern established in the previous section, we only need to replicate the prologue behavior within our custom callback. Specifically, we prepare the stack such that execution returns directly to the address of the `MOV` instruction responsible for storing the return value.

```
do_call:
    mov r10, [r10 + 08h]   ; addressToPush

    ;; Pretending we are in a frame proc
    ;; Note: compile this as a frame proc is absolutely not necessary
    ;;       in this case, it is just to show we are mirroring the prolog of GlobalGetUserAndPassW
    .pushreg rbx
    push rbx
    .allocstack 20h
    sub rsp, 20h           ; this is for the spoofed/swapped frame
    .endprolog

    ; This is the address after the call in GlobalGetUserAndPassW
    push r10
    ; GCONTEXT is the address of a user-controlled Work Item structure
    mov rbx, GCONTEXT      ; we need this structure in RBX
    add rbx, 8             ; we use the address of the return value, unused for generic callbacks

    ; Finally we just to the target function
    jmp rax
```

The resulting call stack appears as the following:

```
Breakpoint 1 hit
KERNELBASE!LoadLibraryA:
00007ffa`5cf51e20 48895c2408      mov     qword ptr [rsp+8],rbx ss:000000f7`750ff970=0000000000000000
0:003> k
 # Child-SP          RetAddr               Call Site
00 000000f7`750ff968 00007ffa`4e702347     KERNELBASE!LoadLibraryA
01 000000f7`750ff970 00007ffa`5f7d287a     wininet!GlobalGetUserAndPassW+0x3f
02 000000f7`750ff9a0 00007ffa`5f7a5e46     ntdll!TppWorkpExecuteCallback+0x13a
03 000000f7`750ff9f0 00007ffa`5d94257d     ntdll!TppWorkerThread+0x8f6
04 000000f7`750ffcd0 00007ffa`5f7caf08     KERNEL32!BaseThreadInitThunk+0x1d
05 000000f7`750ffd00 00000000`00000000     ntdll!RtlUserThreadStart+0x28
0:003> da rcx
000002da`7ae6f000  "urlmon.dll"
```

And this will allow us to recover the callback return value as well. Also consider that this kind of gadget would also be compliant with Eclipse-based inspection, as the full pattern in the function allow to place the return exactly after the `CALL` instruction, as visible below:

```
180152075 e8 ee af        CALL       internal_function
          00 00
18015207a 48 89 03        MOV        qword ptr [RBX],RAX   <--- We can set the return address here
18015207d b8 01 00        MOV        EAX,0x1
          00 00
180152082 eb 06           JMP        PROCEDURE_END
                      SAVE_RBX_RDX
180152084 48 89 02        MOV        qword ptr [RDX],RAX
180152087 48 89 03        MOV        qword ptr [RBX],RAX
                      PROCEDURE_END
18015208a 48 83 c4 20     ADD        RSP,0x20
18015208e 5b              POP        RBX
18015208f c3              RET
180152090 cc              ??         CCh
```

Limitations

The technique does not have many inherent limitations, but of course, depending on the save gadget, you can achieve different results.

Return value[s]

In general, with a `MOV [REG], RAX`, it's possible to implement the above behavior, to save a normal call return value.

For function passing a return value as a parameter, you won't need any of this, of course.

Number of arguments

The last gadget is very important to define how many parameters you can pass onto the stack. The function `GlobalGetUserAndPassW` allocates only the shadow stack and save RBX before calling the internal function, this means that the return address of the previous frame is just 0x28 bytes far. Recalling how parameters passing works in 64-bit, the reader should have clear that in

this case, even clobbering the original value of RBX pushed onto the stack (illegal operation), we have space for maximum one stack parameter before overwriting the return address (which will lead to a crash on return).

For this reason, in case you want to allow for more parameters, you'd need to find proxy frames similar to this one, but with bigger allocations (to support `nargs` parameters you'd need at least `0x20 + (nargs*8)`).

Speaking with Alex Reid, we agreed that alternatively, you can use the same pattern implemented in Moonwalk++, and add an additional proxy frame (conceal-gadget) big enough to permit as many argument as needed.

One, None and a Hundred Thousand

I remember coming across a blog post not long ago that described loading a strangely named DLL to "add a frame" between the ThreadPool worker callback using one of its functions as a forward proxy frame. ~~I wanted to reference it, but I didn't manage to find it.~~ The post I'm referring to is available at this LINK

However, building on this pattern, we can explore several ways to not only add a frame, but literally construct almost fully "custom" call stacks without relying on unusual DLL loading. Naturally, each of these approaches comes with its own set of trade-offs.

1. One example would be to create chains of "simple" backward proxy frames ending with the final one that saves the return address value from the target call.
    - Main advantage: callstacks can be fully customized
    - Disadvantages:
        - valid caller-callee relationships might not be possible to construct consistently
        - not all epilogs executes immediately after a call instruction, making it susceptible to detection
2. Another solution would be to use natural occurring forward proxy frames (i.e., callbacks)
    - Main advantage: forward edges are all perfectly valid and CET compliant
    - Disadvantage:
        - I don't think these chains would ever be used by anybody with a sane mind
        - We are stacking callbacks over… callbacks?
3. … didn't really think about a third one … or did I?

Now, the first solution would require a similar setup we used for stack moonwalk, to identify frame sizes and collect valid backward proxy frames, then just push them on the stack setting the valid return address. Personally, I think the code and examples provided in my previous posts and projects should be more than enough to reproduce it without further explanations.

The second solution, instead, unleashes what I friendly name the “callback hell”, where we can chain multiple callback mechanisms together to obfuscate the callstack. The strategy is quite simple, even more than the previous one, and there’s no shortage of functions that can serve as callback invokers. If you don’t want to research them yourself, the project AlternativeShellcodeExec covers most of what you need.

In order to implement the chaining logic, we will define a general array of structures (Work Items), where each work item would define the pointer to the next call in the chain, its parameters, and the optional return argument.

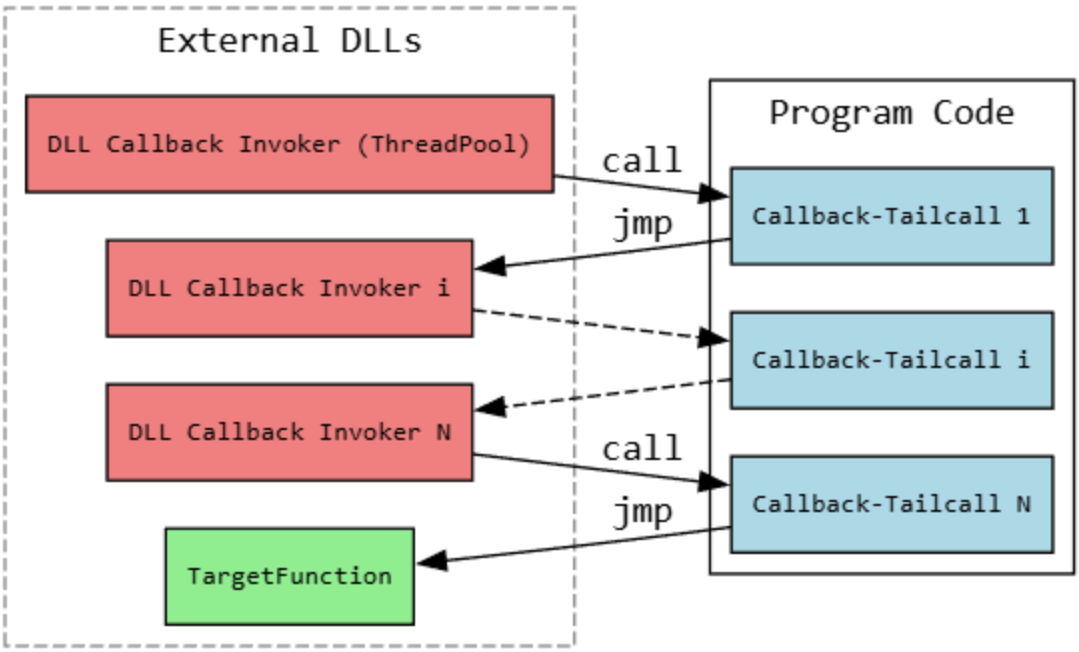
```
typedef struct WorkItemContext {
    FARPROC func;           // Function pointer
    void* retAddress;       // Return address to simulate stack return
    uint64_t argc;          // Argument count
    void* args[MAX_ARGC];   // Arguments (up to MAX_ARGC)
} WorkItemContext;
```

Now, for the actual logic, we will construct three main assembly functions*:

- **FirstCallback**: Initialize certain variables used across the callback chains
- **GenericCallback**: Perform the tail call to the next generic callback invoker
- **LastCallback**: Sets up the save to RBX_SAVE backward proxy frame and performs the actual call to the

> *Note: This kind of implementation is purely arbitrary

The resulting program flow would be similar to the following:



Ok, but why?

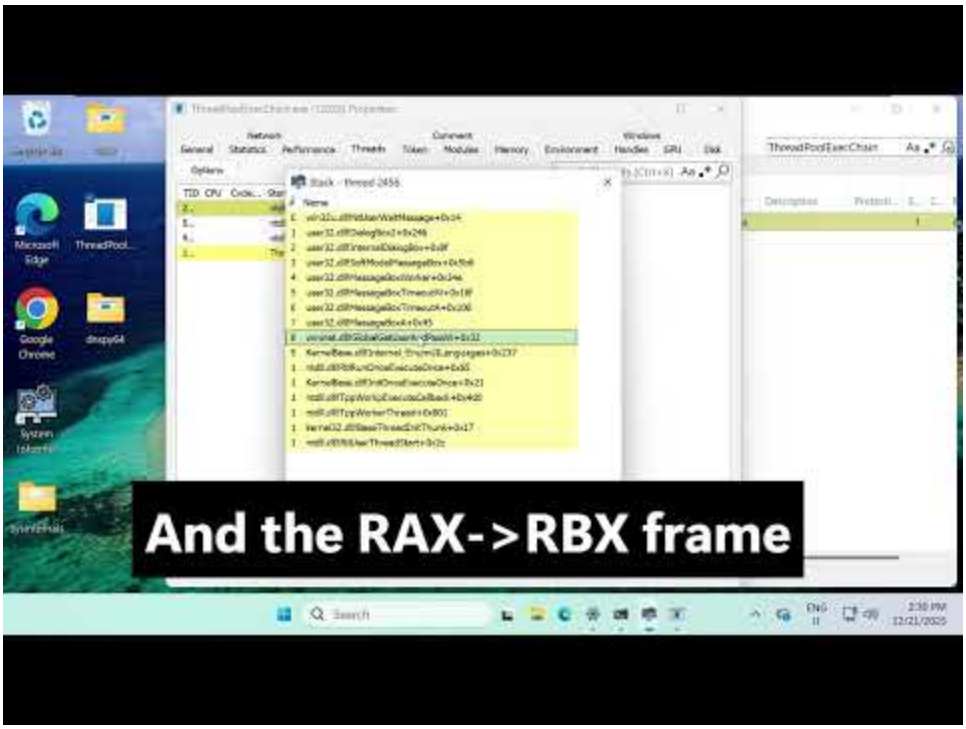
Apparently, as weird as it sounds, this strange call stack construction bypasses current callstack detection logic.

In theory, it could be possible to chain `n` callbacks in `n!` ways, which makes this technique quite interesting for artificial stack construction. On the other side, most of these callbacks are known, so it might also be possible to create signatures for them (well, at least the documented ones).

POC||GTFO

The final POC can be found at ThreadPoolExecChain. As always, the important piece of the puzzle is the assembly. It's nothing complex, and should serve as a basis for more complex designs.

Following, a small video of the technique in action. If the embedded version does not work, here's the LINK.



Watch Video At: https://youtu.be/8oF-33is-Ic

Detection Landscape

> Note: Detection is only required for processes where Intel-CET is disabled. Backward proxy frames are not compatible with CET.

For Frame Swapping using backward proxy frames, the detection landscape largely mirrors that of the original stack moonwalking technique. The most reliable detection strategy remains monitoring the call stack for anomalies. Specifically, identifying return addresses that are not preceded by a legitimate `CALL` instruction.

In fact, this technique imposes quite strict constraints for successful execution: we must locate an instruction that not only stores `RAX` into a pointer but also returns without clobbering critical registers. This combination of conditions significantly reduces the availability of suitable gadgets. However, it is far from impossible to find suitable gadgets, as proven by the one we used in this blog post.

What about the chain of callbacks? Well, frankly I don't know why this behaviour does not have a signature yet, as there are zero legitimate reasons to implement something like this.

Other detection ideas:

- Inspect process IAT to detect dynamic calls
- Validate called function using CFG/XFG bitmap (if CFG enabled)
- CALL address validation
- Code emulation

Closing Remarks

All of this trouble was ultimately the result of chasing a very contingent problem raised by a friend; ironically, one I have not even informed him about yet.

Something to consider is that while chaining callbacks works perfectly fine even with CET enabled, the return-value recovery trick does not, meaning we will have to figure out a different approach in the future. Until then.

References

- Hiding In PlainSight - Indirect Syscall is Dead! Long Live Custom Call Stacks
- Windows x64 Calling Convention
- AlternativeShellcodeExec
- PoolParty
- Evading Elastic EDR's call stack signatures with call gadgets