Sleeping Beauty II: CFG, CET, and Stack Spoofing

maorsabag.github.io/posts/adaptix-stealthpalace/sleeping-beauty-ii

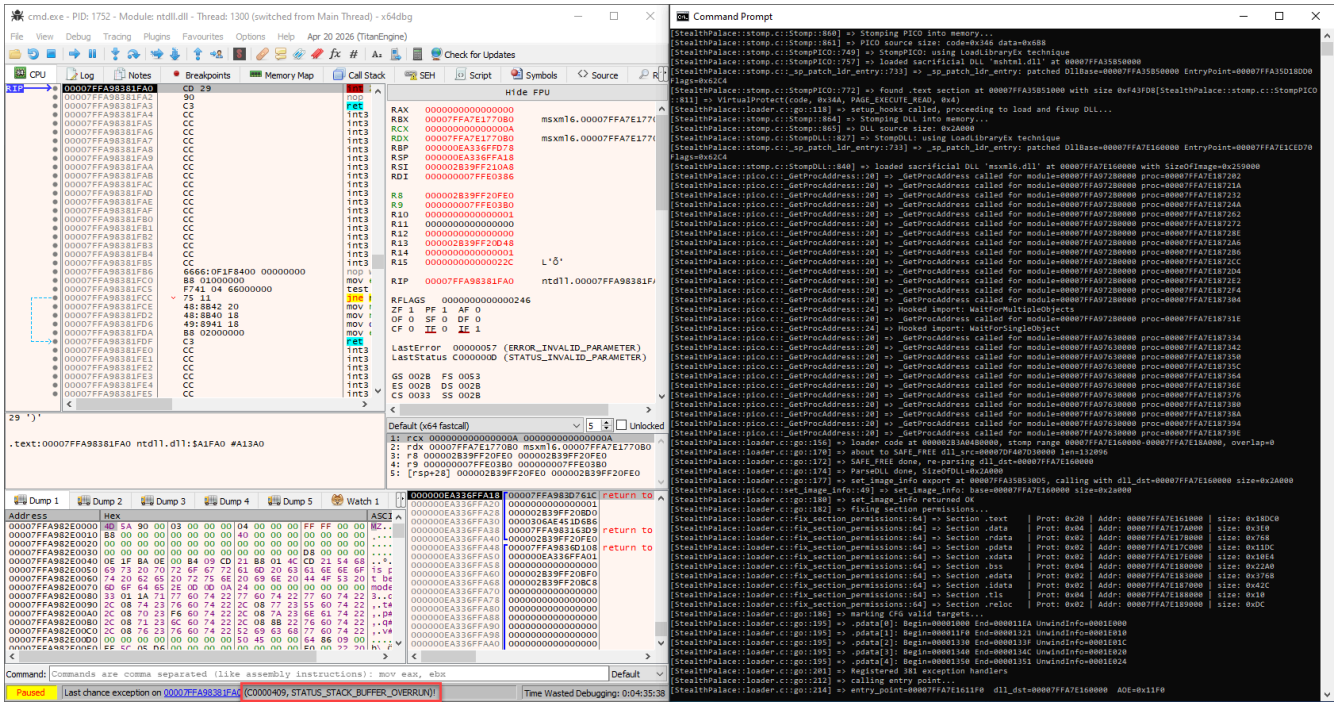
Maor Sabag June 6, 2026

Contents

3349 words 16 minutes

> *A tale of CFG bitmaps, shadow stacks, and teaching an implant to sleep in places it was never meant to survive.*

In Part I, we built StealthPalace: a Crystal Palace RDLL wrapper for Adaptix with IAT hooking and Ekko-style sleep obfuscation. It worked - on most targets. But inject that payload into a process with **Control Flow Guard** enabled, and the first indirect call in the ROP chain triggers a CFG violation. The process dies. No callback, no fallback, just a silent `STATUS_STACK_BUFFER_OVERRUN` and a corpse in Event Viewer.



The first sign something went wrong - STATUS_STACK_BUFFER_OVERRUN after injecting into a CFG-enabled process.



First time injecting into a CFG process?

This post covers the three changes that fix that - and what it took to make stack spoofing survive alongside CFG and CET/Shadow Stack enforcement.

Chapter 1: The CFG Problem - Why ROP Chains Crash on Hardened Processes

What Is Control Flow Guard?

Before diving into what broke, let's take a step back. **Control Flow Guard (CFG)** is a Microsoft exploit mitigation that's been shipping since Windows 8.1 Update 3 and Windows 10. Its job is simple: prevent attackers from hijacking indirect calls (calls through function pointers, vtable dispatches, callback invocations) to redirect execution to arbitrary code.

If you've ever seen a binary compiled with `/guard:cf` in Visual Studio - that's CFG. The compiler analyzes the program at build time, identifies every function that could legitimately be the target of an indirect call, and embeds that list into the PE's load config directory. At runtime, the OS builds a **bitmap** from that list. Before every indirect call, a compiler-inserted

check validates the target against the bitmap. Invalid target? The process terminates immediately - no exception handler, no second chance.

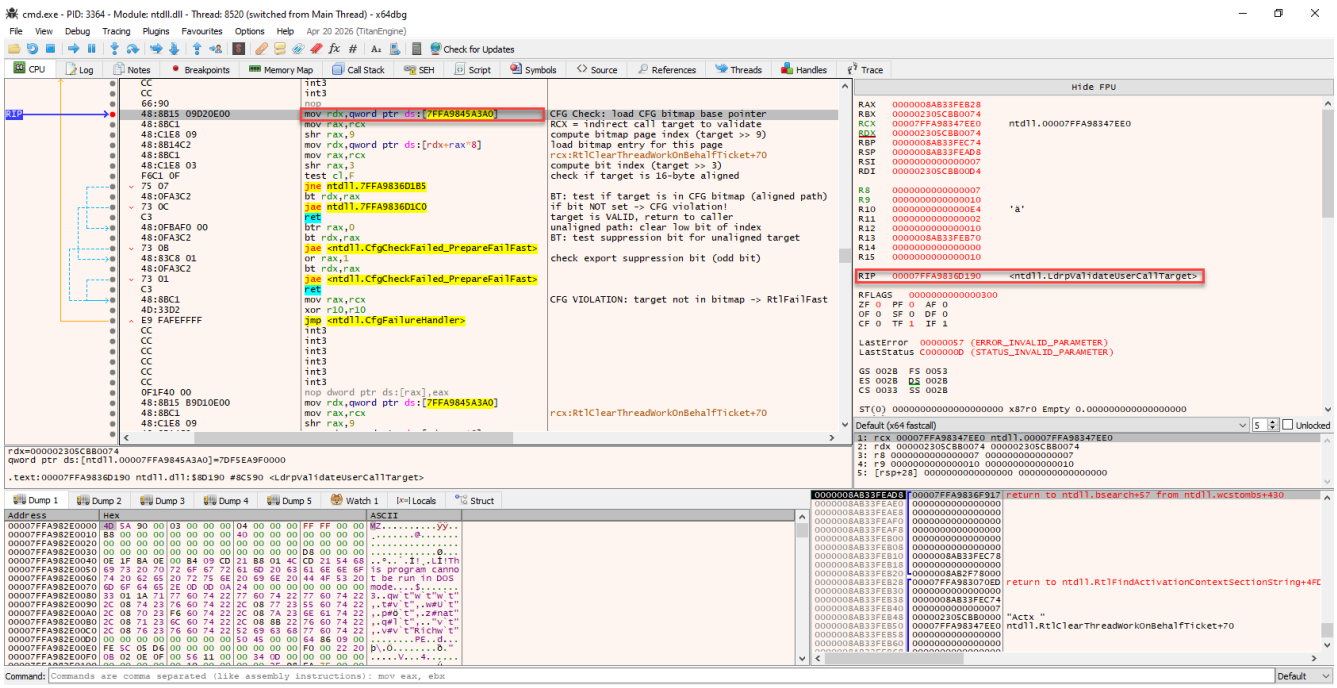
For a deeper look at CFG internals, see:

- Microsoft's official documentation: Control Flow Guard
- MJ0011's "Windows 10 Control Flow Guard Internals" - one of the earliest and most detailed reverse-engineering walkthroughs of the CFG bitmap structure

> **Why should you care?** CFG is enabled by default on most system processes (explorer.exe, svchost.exe, msedge.exe) and is routinely enforced via enterprise Group Policy. If your implant can only survive in processes *without* CFG, you've lost most of the interesting injection targets.

What CFG Actually Enforces

Control Flow Guard maintains a **bitmap** of valid indirect call targets for every executable page in a process. When a module is loaded, the linker-emitted CFG metadata (`IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG` -> `GuardCFCheckFunctionPointer`) populates that bitmap. Every indirect `call` instruction is preceded by a check against the bitmap. If the target address isn't marked valid, `ntdll!RtlFailFast2` terminates the process.

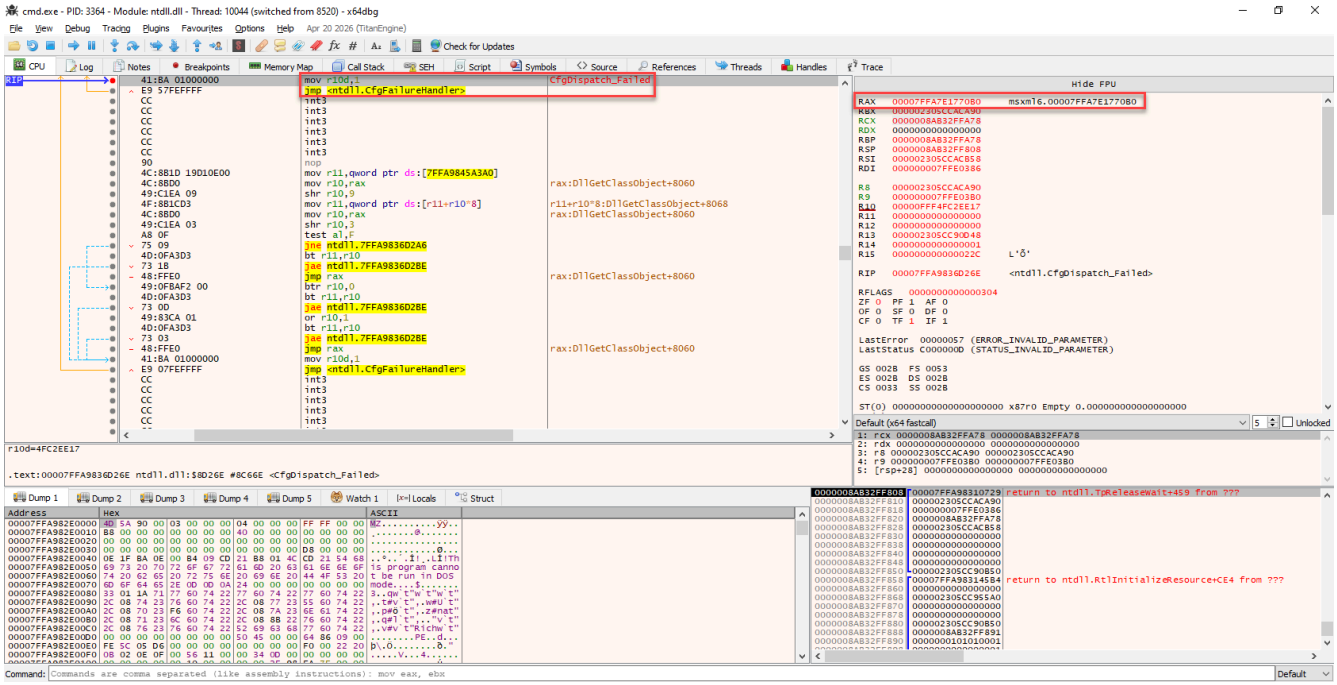


Every indirect call is checked against the CFG bitmap before execution.

Our sleep obfuscation works by building a ROP chain of `CONTEXT` structures and firing them through `NtContinue` via timer callbacks (Ekko) or queued APCs (Kraken Mask). Each frame's `Rip` points to a **gadget** - a `jmp [rbx]` or `jmp rdi` instruction found by scanning ntdll's `.text`

section. These are mid-function addresses. They were never in any module's CFG metadata. On a CFG-enabled process, the first `NtContinue` -> gadget transition is an invalid indirect call.

> **Why this matters:** Modern EDRs and enterprise builds routinely enable CFG via `/guard:cf`. Any serious C2 implant that uses ROP-based sleep obfuscation needs to register its gadgets, or it's limited to unprotected processes.



Invalid call target from our stomped module - CFG rejects the indirect call.

The Fix: NtSetInformationVirtualMemory

Windows exposes `NtSetInformationVirtualMemory` with information class `VmCfgCallTargetInformation` (value 2). This lets a caller register arbitrary addresses as valid CFG call targets at runtime - no linker metadata required.

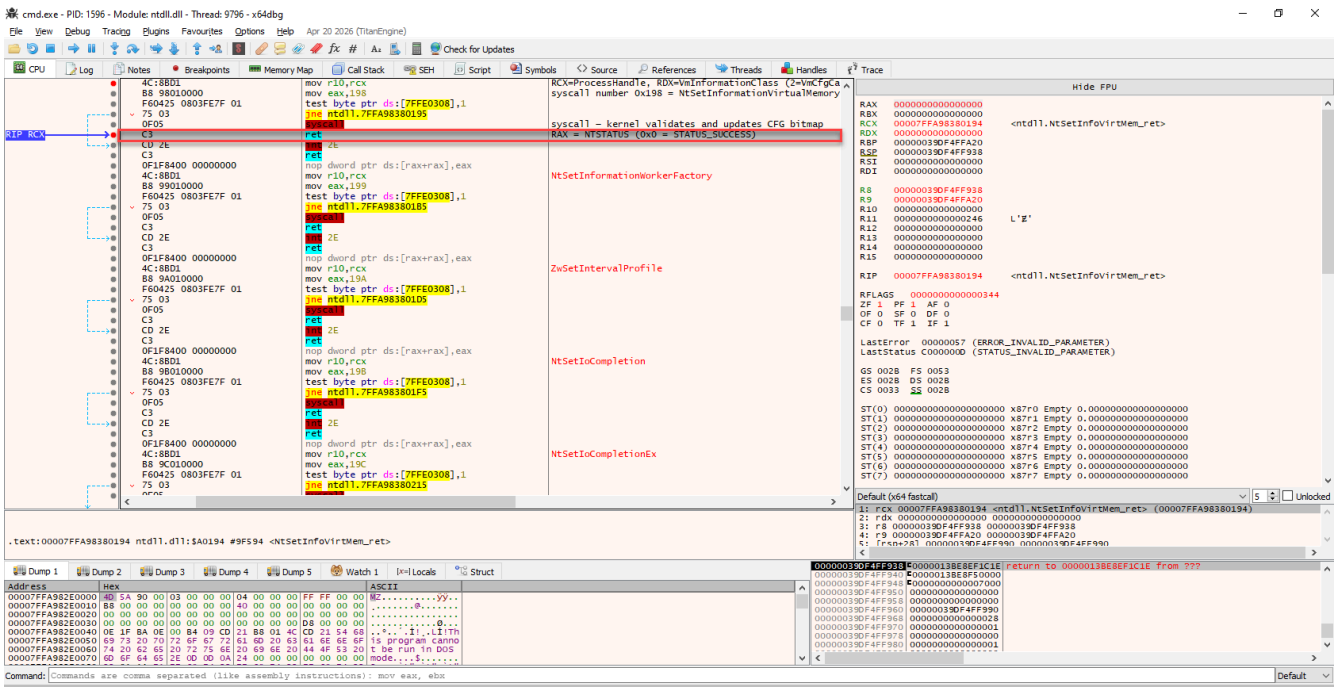
The API takes a `MEMORY_RANGE_ENTRY` describing the page range and an array of `CFG_CALL_TARGET_INFO` structures, each specifying an offset within that range and a `CFG_CALL_TARGET_VALID` flag:

```
typedef struct {
    ULONG_PTR Offset;
    ULONG_PTR Flags;    // CFG_CALL_TARGET_VALID = 0x1
} MY_CFG_CALL_TARGET_INFO;

typedef struct {
    ULONG                    dwNumberOfOffsets;
    PULONG                   plOutput;
    MY_CFG_CALL_TARGET_INFO *ptOffsets;
```

```
    PVOID                   pMustBeZero;
    PVOID                   pMoarZero;
} MY_VM_INFORMATION;

NTSTATUS status = NtSetInformationVirtualMemory(
    NtCurrentProcess(),
    VmCfgCallTargetInformation,   // 2
    1, &range, &vmInfo, sizeof(vmInfo)
);
```



After registration - our gadgets are now valid CFG call targets.

There's a subtlety: for addresses inside a loaded image (ntdll, kernel32, advapi32), the `MEMORY_RANGE_ENTRY.VirtualAddress` must be the **image base**, and `NumberOfBytes` must cover the full image size. For addresses in manually mapped memory (our PICO code, our stomped DLL), the range is the allocation base. Getting this wrong produces `STATUS_INVALID_PAGE_PROTECTION` or silently applies zero targets.

This led to three primitives:

- `_cfg_mark_single_image(ImageBase, Function)` - for gadgets inside loaded system DLLs
- `_cfg_mark_single(addr)` - for gadgets in our own PIC code
- `_cfg_mark_region(base, size)` - for marking entire executable sections (every 16-byte-aligned offset)

Can be found here: https://github.com/MaorSabag/Adaptix-StealthPalace/blob/main/src/cfg.c

What Gets Registered

Every indirect call target in the sleep obfuscation chain needs to be marked. For **Ekko** (timer-based):

```
jmp [rbx] gadget              (ntdll .text)     _cfg_mark_single
NtContinue                    (ntdll export)    _cfg_mark_single_image
SystemFunction032             (advapi32)        _cfg_mark_single_image
WaitForSingleObjectEx         (kernel32)        _cfg_mark_single_image
VirtualProtect                (kernel32)        _cfg_mark_single_image
GetThreadContext              (kernel32)        _cfg_mark_single_image
SetThreadContext              (kernel32)        _cfg_mark_single_image
SetEvent                      (kernel32)        _cfg_mark_single_image
RtlMoveMemory                 (ntdll)           _cfg_mark_single_image
restore_section_permissions   (our PIC)         _cfg_mark_single
```

For **Kraken Mask** (APC-based), the list is longer - it adds `NtTestAlert`, `NtWaitForSingleObject`, `NtSetEvent`, `NtSignalAndWaitForSingleObject`, `NtAlertResumeThread`, `RtlExitUserThread`, `NtSuspendThread`, `NtResumeThread`, plus the `jmp rdi` gadget and heap functions.

```
jmp rdi gadget                    (kernel32 .text)  _cfg_mark_single
NtTestAlert                       (ntdll export)    _cfg_mark_single_image
NtWaitForSingleObject             (ntdll export)    _cfg_mark_single_image
NtSetEvent                        (ntdll export)    _cfg_mark_single_image
NtSignalAndWaitForSingleObject    (ntdll export)    _cfg_mark_single_image
NtAlertResumeThread               (ntdll export)    _cfg_mark_single_image
RtlExitUserThread                 (ntdll export)    _cfg_mark_single_image
NtSuspendThread                   (ntdll export)    _cfg_mark_single_image
NtResumeThread                    (ntdll export)    _cfg_mark_single_image
VirtualProtect                    (kernel32)        _cfg_mark_single_image
SystemFunction032                 (advapi32)        _cfg_mark_single_image
GetThreadContext                  (kernel32)        _cfg_mark_single_image
SetThreadContext                  (kernel32)        _cfg_mark_single_image
RtlMoveMemory                     (ntdll)           _cfg_mark_single_image
restore_section_permissions       (our PIC)         _cfg_mark_single
```

When Registration Happens

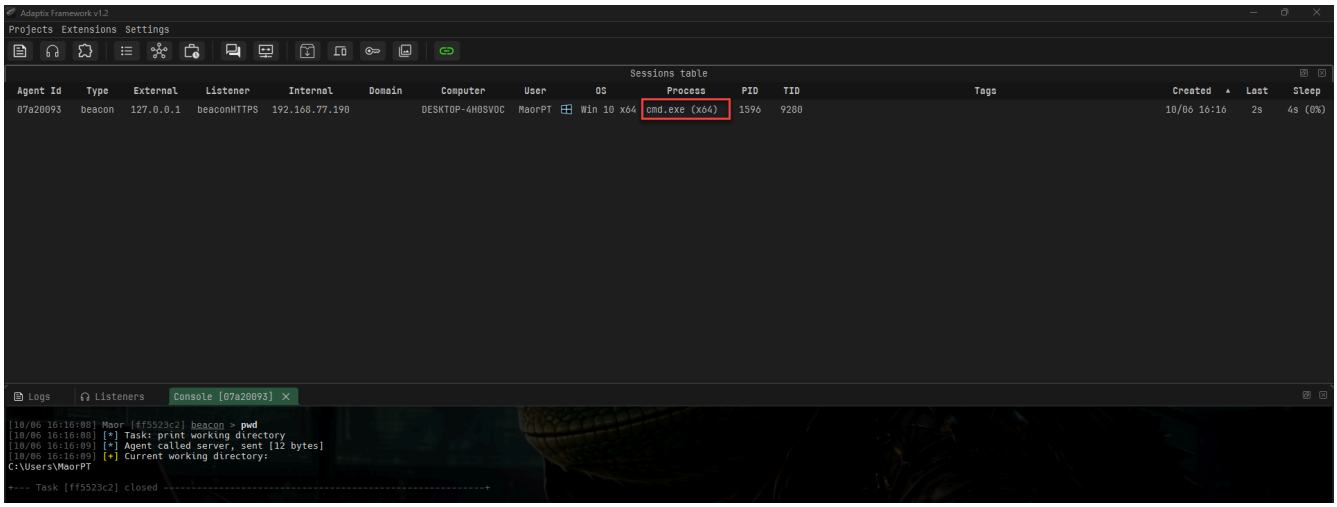
CFG marking runs at three points in the loader lifecycle:

1. **`EnableCFGForPICO()`** - called from `go()` right after the PICO code is stomped into the sacrificial DLL. Marks the entire PICO code region.
2. **`EnableCFGForGadgets()`** - called from `set_image_info()` after the agent DLL is mapped and `ResolveHookFunctions()` has populated the global function pointers. Marks every ROP gadget.

3. **`EnableCFG()`** - called from `go()` after `fix_section_permissions()`. Walks the agent DLL's section table and marks every executable section.

```
┌──────────────────────────────────────────────────┐
│                  loader PIC (go)                 │
│                                                  │
│  1. Stomp PICO into sacrificial DLL              │
│  2. EnableCFGForPICO()        ← mark PICO code   │
│  3. setup_hooks()                                │
│  4. Map + fixup agent DLL                        │
│  5. set_image_info()                             │
│     ├─ ResolveHookFunctions()                    │
│     └─ EnableCFGForGadgets()  ← mark ROP gadgets │
│  6. fix_section_permissions()                    │
│  7. EnableCFG()               ← mark agent .text │
│  8. DllMain(DLL_PROCESS_ATTACH)                  │
└──────────────────────────────────────────────────┘
```

The result: the payload can now be injected into any CFG-enabled process - including those spawned with `ProcessMitigationPolicy` flags like `ProcessControlFlowGuardPolicy` - and the sleep obfuscation chain runs cleanly.



Same process, same payload. CFG registration is the difference - agent running on a CF-Guard-enabled target.



The payload after surviving CFG enforcement.

Chapter 2: Stack Spoofing During Sleep - TIB Swap and CET Compatibility

The Problem With Sleeping Threads

Encrypting the agent's memory during sleep (as we did in Part I) solves the static memory scanning problem - but it doesn't address **stack analysis**. When the implant sleeps, the main thread blocks on a wait function (`WaitForSingleObjectEx`, `WaitForMultipleObjects`, or `ConnectNamedPipe`). During that window, the thread's call stack is frozen and visible to any scanner. A stack walk reveals:

```
ntdll!NtWaitForSingleObject
KERNELBASE!WaitForSingleObjectEx
    ↓
<our ROP gadget address>     ← suspicious: not in any known module
    ↓
<our hooked function>        ← inside stomped DLL: looks wrong
```

This is the bread and butter of modern EDR thread inspection. Products like Elastic's call-stack-based thread analysis, and Microsoft Defender's stack walking heuristics all inspect sleeping threads' call stacks. If the return addresses point into memory that doesn't belong to a legitimately loaded module - or into a region whose backing file doesn't match the expected image - the thread gets flagged.

> **The takeaway:** memory encryption hides what's *in* the pages; stack spoofing hides *who's sleeping on them*. You need both.

The Part I Approach: Full Context Swap via GetThreadContext / SetThreadContext

In Part I, we already had a form of stack spoofing. The Ekko ROP chain included two frames that captured a random thread's full `CONTEXT` and applied it wholesale to the sleeping main thread:

```
Step 1: GetThreadContext(MainThread, &CtxBkp)    ← save real context
Step 2: SetThreadContext(MainThread, &CtxSpf)     ← overwrite with spoof thread's conte
         ...sleep...
Step 8: SetThreadContext(MainThread, &CtxBkp)     ← restore real context
```

This worked - on most targets. The spoof context replaced **everything**: `Rip`, `Rsp`, `Rbp`, all general-purpose registers. A stack walker inspecting the sleeping thread saw the spoof thread's instruction pointer and stack pointer, so the call stack looked completely legitimate.

Then we tried it on a machine with CET.

Intel **Control-flow Enforcement Technology (CET)**, specifically its **Shadow Stack** feature, maintains a hardware-backed copy of return addresses. Every `call` pushes the return address to both the regular stack and the shadow stack; every `ret` compares the two. But CET doesn't just protect `ret` - it also validates `SetThreadContext`. When you call `SetThreadContext` to change a thread's `Rip`, the kernel checks whether the new `Rip` is consistent with the shadow stack. If it isn't - and a spoofed `Rip` from a completely different thread won't be - the call fails silently or the thread faults on resume.

The result on CET-enabled hardware: `SetThreadContext` with the spoofed context either fails outright or produces an inconsistent state. The thread sleeps with its **original, unspoofed stack** fully visible to memory scanners. Every return address in the call stack still points into our stomped DLL - exactly what we were trying to hide.

> **In short:** the Part I approach treated the entire `CONTEXT` as one opaque blob to swap. CET made that impossible by tying `Rip` to hardware state we can't forge. We needed a

> more surgical approach - one that spoofs what stack *walkers* look at (the stack bounds and RSP) while leaving what *CET* looks at (the instruction pointer) untouched.

For background on CET and Shadow Stacks, see:

- Intel's CET specification (PDF)
- Alex Ionescu's "Battle Of The SKM And IUM" for how Windows integrates CET with kernel-mode protections
- Microsoft's documentation on Hardware-enforced Stack Protection

TIB Swap: Spoofing the Stack Bounds

The fix requires understanding what stack walkers actually look at. The Thread Information Block (TIB), accessible at `gs:0x30` (x64) or `fs:0x18` (x86), contains `StackBase` and `StackLimit` - the range the OS considers the thread's valid stack. Stack walkers - including EDR stack-walking engines - use these bounds to decide when to stop unwinding and to validate whether return addresses fall within a "legitimate" stack range.

Instead of replacing the entire context (which CET blocks), we **swap only the TIB's stack bounds** with a legitimate thread's bounds. This makes stack walkers believe our thread's stack is somewhere else entirely - somewhere clean - while leaving `Rip` untouched so CET stays happy.

```
/* Pick a random thread in the same process */
ULONG SpoofTid = RndThreadId(CurrentTid);

/* Read the spoof thread's TIB via NtQueryInformationThread */
THREAD_BASIC_INFORMATION tbi = { 0 };
NtQueryInformationThread(DupThreadHandle, 0, &tbi, sizeof(tbi), NULL);
memcpy(&TibSpoof, tbi.TebBaseAddress, sizeof(NT_TIB));

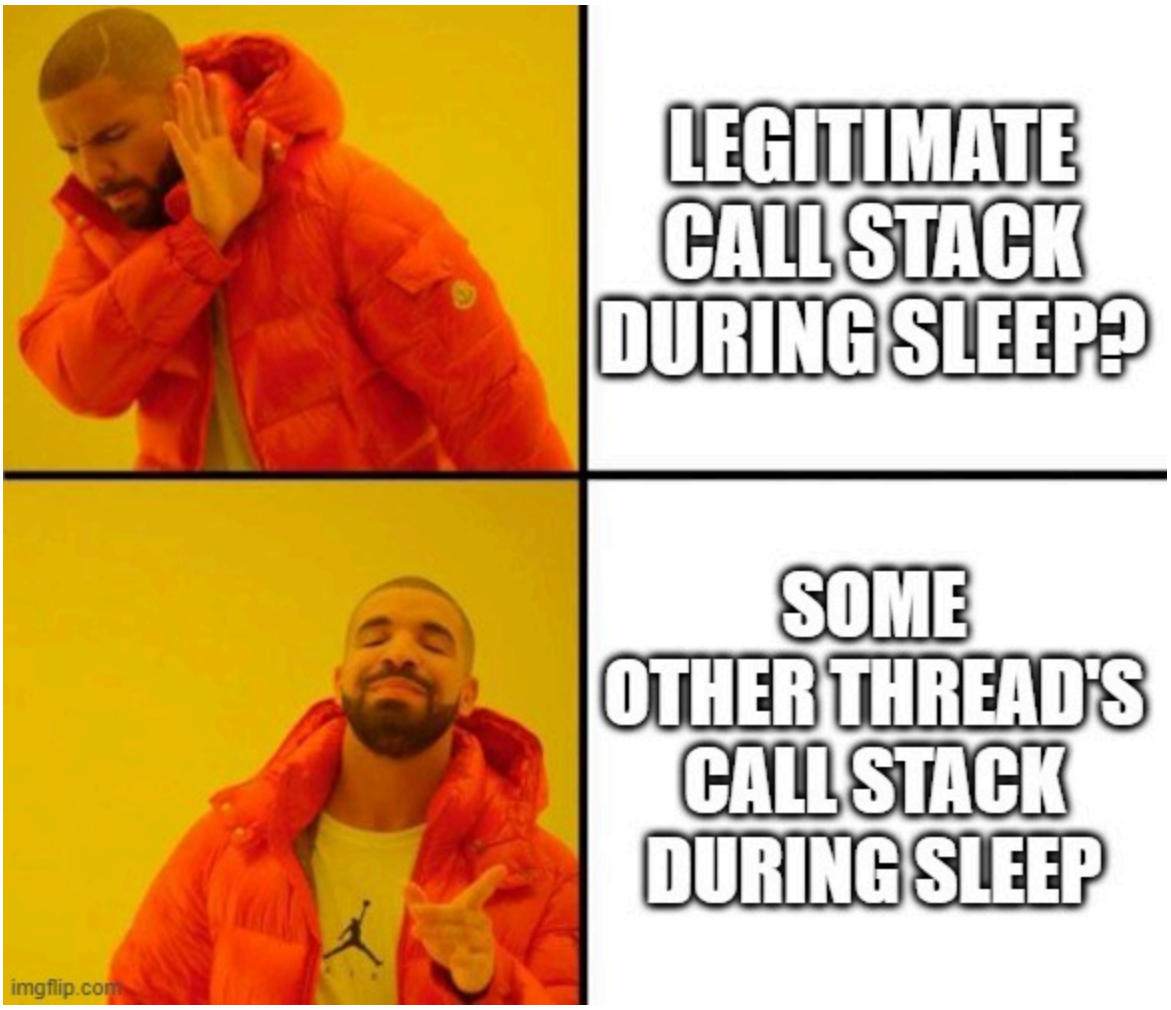
/* Backup our real TIB */
PVOID pTeb;
__asm__ volatile ("movq %%gs:0x30, %0" : "=r" (pTeb));
memcpy(&TibBackup, pTeb, sizeof(NT_TIB));

/* ROP frame: swap TIB to spoof's stack bounds */
// RtlCopyMemory(pTeb, &TibSpoof, sizeof(NT_TIB))

/* ... sleep ... */

/* ROP frame: restore original TIB */
// RtlCopyMemory(pTeb, &TibBackup, sizeof(NT_TIB))
```

Now when an EDR walks the sleeping thread's stack, it sees `StackBase` and `StackLimit` pointing to a legitimate thread's stack range. The unwinder follows that range and finds clean return addresses from a real thread - not our implant's frames.



Why have your own call stack when you can borrow someone else's?

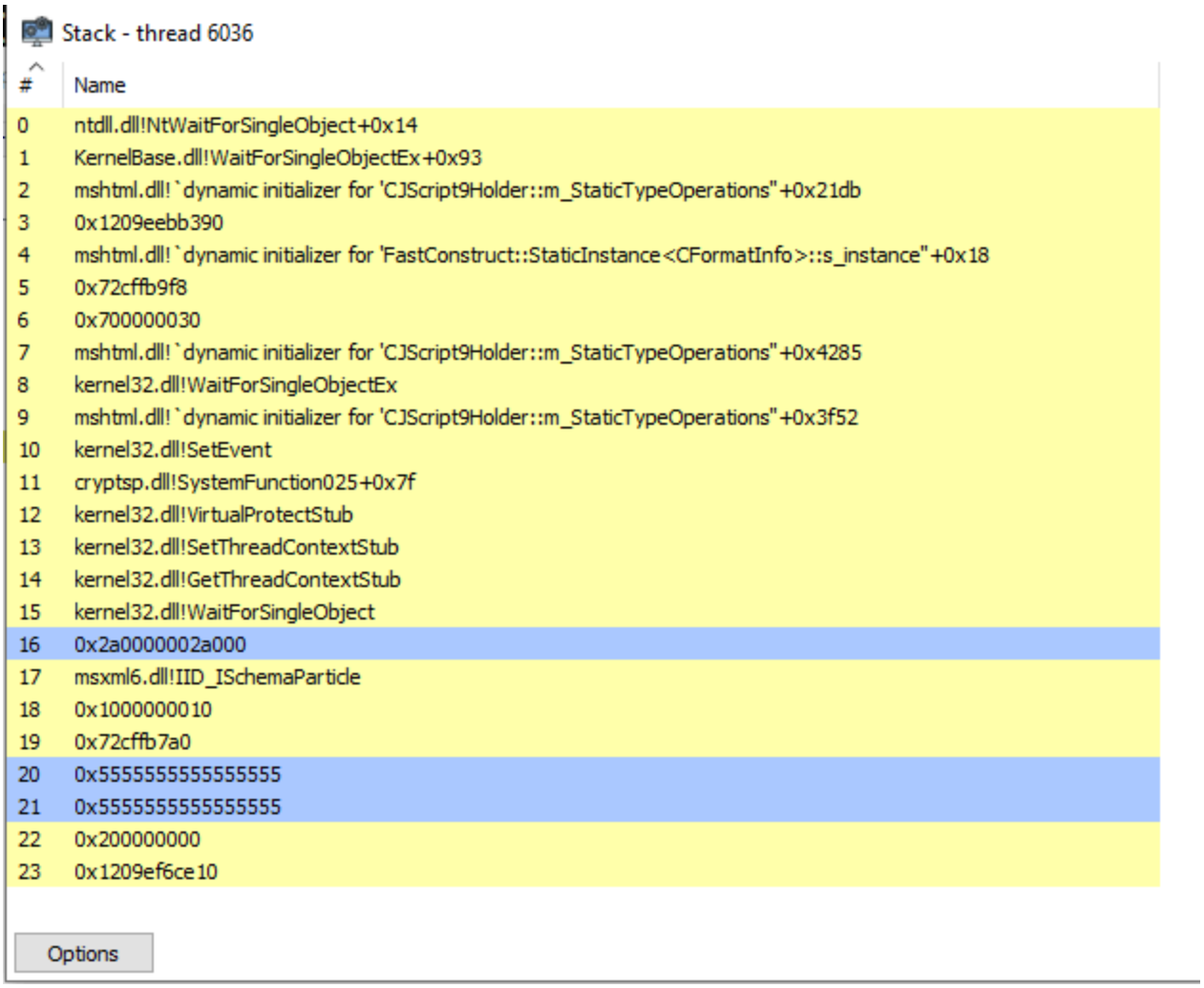
CET / Shadow Stack Compatibility

We've already seen in the Part I section above why the full context swap fails on CET. Let's look at the mechanism in more detail.

Intel CET (Control-flow Enforcement Technology), shipping on 12th-gen+ Intel and AMD Zen 3+ processors, introduces a **shadow stack**: a separate, hardware-maintained stack that the CPU manages in parallel with the regular stack. On every `call`, the CPU pushes the return address to *both* stacks. On every `ret`, it compares the two - if they don't match, a `#CP` (Control Protection) exception fires. This is primarily an anti-ROP mitigation, but it has a side effect that matters to us: **`SetThreadContext` also validates `Rip` against the shadow stack.** If you try to set `Rip` to an address that wasn't pushed by a `call`, the operation fails.

For more context on how CET works at the hardware level, see:

- Intel's CET specification - the authoritative hardware reference
- Connor McGarr's "A Technical Analysis of CET" - excellent hands-on walkthrough with WinDbg examples showing exactly how shadow stack validation works during `SetThreadContext`



Without CET compatibility - SetThreadContext overwrites RIP, which CET rejects.

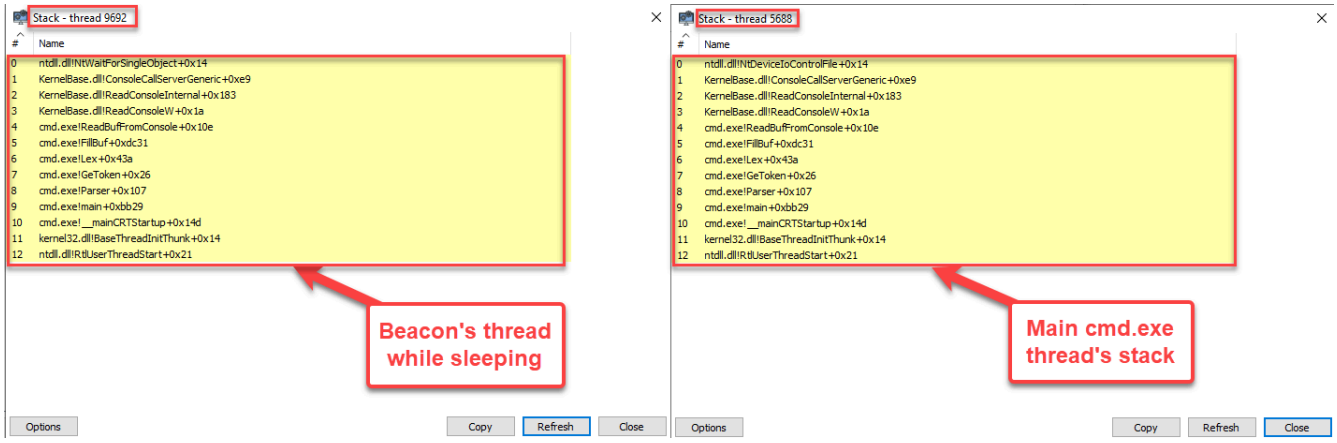
As we discussed, the naive approach (Part I's method) of capturing a spoof thread's context and applying it wholesale to the sleeping thread would change `Rip`. CET catches that. The fix is surgical: **copy only the spoof thread's RSP (and stack bounds via TIB), but preserve the sleeping thread's real RIP.**

```
/* ROP frame: capture the main thread's real sleeping context */
// GetThreadContext(MainThread, &CtxCap)

/* ROP frame: copy the real RIP into the spoof context */
// RtlCopyMemory(&CtxSpf.Rip, &CtxCap.Rip, sizeof(DWORD64))
```

```
/* ROP frame: apply spoof context (RSP is spoofed, RIP is real) */
// SetThreadContext(MainThread, &CtxSpf)
```

The `GetThreadContext` -> `memcpy(Rip)` -> `SetThreadContext` sequence ensures that `CtxSpf.Rip` matches the shadow stack's expected return address. CET sees a consistent instruction pointer. The RSP and TIB now point to the spoof thread's stack range, but the RIP is genuine. Stack scanners are satisfied, and CET doesn't fault.



CET-safe stack spoofing - RSP and TIB point to the spoof thread, but RIP stays genuine.



Identity theft is not a joke, Jim!

> **Why this matters:** CET Shadow Stacks are enabled by default on 12th-gen+ Intel (Alder Lake and later) and AMD Zen 3+ CPUs running Windows 11 22H2+. Microsoft has been progressively expanding CET enforcement across system processes and applications - Edge, Chrome, and most system services now run with shadow stacks

> active. If your sleep obfuscation breaks on CET, you're locked out of the majority of modern enterprise endpoints.

Chapter 3: Ekko v2 vs Kraken Mask - Two Approaches to the Same Problem

Both techniques solve the same problem - encrypt the image, spoof the stack, sleep, restore - but they differ in how the ROP chain is dispatched and how the worker thread is managed.

Ekko v2: Timer-Based Dispatch

Ekko uses `CreateTimerQueueTimer` to schedule each ROP frame. The timer thread pool calls `NtContinue` with each `CONTEXT` structure in sequence, gated by 100ms intervals:

```
┌──────────────────────────────────────────────────────┐
│  Ekko v2 - 14 ROP frames (timer-queue callbacks)     │
│                                                      │
│  0:  WaitForSingleObject(hEvtStart)     gate         │
│  1:  VirtualProtect(image -> RW)                     │
│  2:  SystemFunction032(encrypt)         RC4          │
│  3:  GetThreadContext(Main, &CtxCap)    capture RIP  │
│  4:  memcpy(&CtxSpf.Rip, &CtxCap.Rip)  CET safe      │
│  5:  memcpy(TEB, &TibSpoof)            swap TIB      │
│  6:  SetThreadContext(Main, &CtxSpf)    spoof RSP    │
│  7:  <sleep / hook dispatch>                         │
│  8:  memcpy(TEB, &TibBackup)           restore TIB   │
│  9:  SetThreadContext(Main, &CtxCap)    restore ctx  │
│  10: SystemFunction032(decrypt)         RC4          │
│  11: restore_section_permissions                     │
│  12: SetEvent(hEvtEnd)                  signal done  │
└──────────────────────────────────────────────────────┘
```

The main thread fires the chain via `SetEvent(hEvtStart)` and blocks on `WaitForSingleObject(hEvtEnd)`. When the timer thread signals completion, the main thread wakes with the image decrypted, permissions restored, and its original context back in place.

Ekko's gadget: `jmp [rbx]` (`FF 23`) found in ntdll's `.text` section. Each ROP frame sets `Rbx` to point to the target function pointer, achieving an indirect call through a legitimate ntdll instruction.

Kraken Mask: APC-Based Dispatch

Kraken Mask takes a different approach. Instead of timer callbacks, it creates a **suspended helper thread** with entry point `NtTestAlert` and queues the entire ROP chain as APCs via

NtQueueApcThread:

```
┌──────────────────────────────────────────────────────┐
│  Kraken Mask - 16 ROP frames (queued APCs)           │
│                                                      │
│  0:  NtWaitForSingleObject(hEvtStart)   gate         │
│  1:  GetThreadContext(Main, &CtxCap)    capture RIP  │
│  2:  RtlCopyMemory(&CtxSpf.Rip, ...)   CET safe      │
│  3:  RtlCopyMemory(TEB, &TibSpoof)     swap TIB      │
│  4:  SetThreadContext(Main, &CtxSpf)    spoof RSP    │
│  5:  VirtualProtect(image -> RW)                     │
│  6:  SystemFunction032(encrypt)         RC4          │
│  7:  <sleep / hook dispatch>                         │
│  8:  SystemFunction032(decrypt)         RC4          │
│  9:  restore_section_permissions                     │
│  10: NtSuspendThread(Main)             safe restore  │
│  11: RtlCopyMemory(TEB, &TibBackup)   restore TIB    │
│  12: SetThreadContext(Main, &CtxCap)   restore ctx   │
│  13: NtSetEvent(hEvtEnd)               signal done   │
│  14: NtResumeThread(Main)                            │
│  15: RtlExitUserThread(0)             cleanup helper │
└──────────────────────────────────────────────────────┘
```

Key differences:

Dispatch mechanism. Ekko fires `NtContinue` from the timer thread pool - these are shared OS threads. Kraken Mask uses a dedicated helper thread with `NtTestAlert` as its entry point, alert-resumed via `NtAlertResumeThread`. The APC queue drains sequentially through that single thread.

State allocation. Ekko keeps all 14 `CONTEXT` structures and TIB copies on the stack. Kraken Mask heap-allocates a `KRAKEN_STATE` structure containing all 16 contexts, the TIB copies, and the RC4 descriptors. This avoids stack overflow on deep call chains - the helper thread's stack is only 64KB (`0x10000`).

TIB restore safety. Ekko restores the TIB via a ROP frame while the main thread is already awake (blocked on `WaitForSingleObject(hEvtEnd)`). Kraken Mask does it more carefully: frame 10 calls `NtSuspendThread(Main)` to freeze the main thread, frames 11-12 restore the TIB and context while the thread is suspended, then frame 14 calls `NtResumeThread(Main)`. This eliminates a race window where a stack scanner could catch the thread with a partially restored TIB.

Gadget choice. Ekko uses `jmp [rbx]` (`FF 23` in ntdll) and sets each frame's `Rbx`. Kraken Mask uses `jmp rdi` (`FF E7` in kernel32) and sets each frame's `Rdi`. Different register, different

module, different byte signature. Having two gadget families means a signature targeting one doesn't catch the other.

Firing mechanism. Ekko: `SetEvent(hEvtStart)` + `WaitForSingleObject(hEvtEnd)`. Kraken Mask: `NtSignalAndWaitForSingleObject(hEvtStart, hEvtEnd)` - a single atomic syscall that signals the start event and blocks on the end event simultaneously, reducing the timing window between signal and sleep.

Choosing Between Them

	Ekko v2	Kraken Mask
Dispatch	Timer queue callbacks	Queued APCs
ROP frames	14	16
Gadget	`jmp [rbx]` (ntdll)	`jmp rdi` (kernel32)
State storage	Stack	Heap
TIB restore	While main is awake	While main is suspended
Fire mechanism	Two calls	Single atomic syscall
Thread footprint	Uses OS timer pool	Dedicated helper thread

Ekko is simpler and doesn't create a new thread. Kraken Mask has a tighter restore sequence and no timer pool dependency. Both produce a clean call stack during sleep.

Chapter 4: Putting It All Together

The full pipeline for a StealthPalace payload on a CFG+CET target now looks like this:

```
┌──────────────────────────────────────────────────────────┐
│  Injection into CFG-enabled process                      │
│                                                          │
│  1. Loader PIC executes go()                             │
│  2. PICO stomped into sacrificial DLL                    │
│  3. EnableCFGForPICO()          mark PICO as CFG-valid   │
│  4. setup_hooks() installs IAT hooks                     │
│  5. Agent DLL mapped + imports resolved                  │
│  6. ResolveHookFunctions()      resolve all gadgets + APIs  │
│  7. EnableCFGForGadgets()       mark every ROP target    │
│  8. fix_section_permissions()   per-section protections  │
```

```
│  9. EnableCFG()                 mark agent .text sections   │
│  10. DllMain(DLL_PROCESS_ATTACH)                            │
│                                                             │
│  ── Agent running ──                                        │
│                                                             │
│  On sleep (WaitForSingleObjectEx > 1000ms):                 │
│    a. Pick spoof thread, read its TIB + context             │
│    b. Build ROP chain (Ekko or Kraken Mask)                 │
│    c. VirtualProtect(RW) -> RC4 encrypt image               │
│    d. Copy real RIP into spoof context (CET safe)           │
│    e. Swap TIB to spoof's stack bounds                      │
│    f. SetThreadContext with spoofed RSP, real RIP           │
│    g. Sleep (actual blocking wait)                          │
│    h. RC4 decrypt -> restore permissions                    │
│    i. Restore TIB + original context                        │
│    j. Continue execution                                    │
└─────────────────────────────────────────────────────────────┘
```

During step (g), an EDR scanning the thread sees:

- **Call stack**: return addresses pointing into the spoof thread's legitimate stack range
- **TIB**: `StackBase`/`StackLimit` consistent with a real thread
- **RIP**: the genuine sleep location (CET shadow stack agrees)
- **Memory**: the agent image is RC4-encrypted, permissions set to `PAGE_READWRITE`
- **CFG bitmap**: all indirect call targets registered as valid

The implant is asleep, encrypted, with a spoofed stack, in a CFG-enforced process, on CET hardware. When the wait completes, the ROP chain restores everything and execution continues.

Sleeping Beauty indeed - again.

> **Disclaimer:** This research is for educational and authorized red team purposes only. Always obtain proper authorization before using these techniques.