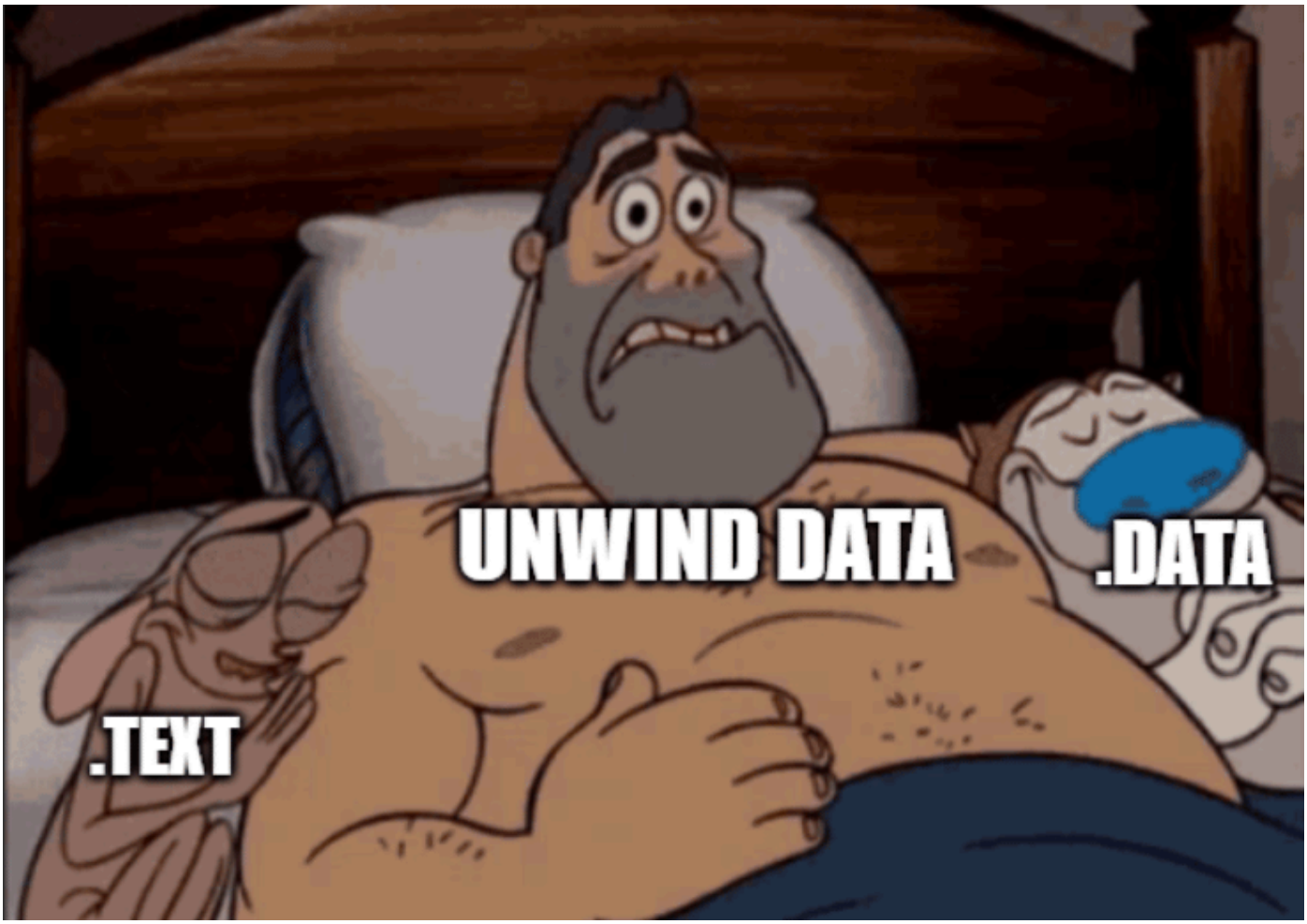
Unwind Data Can't Sleep - Introducing InsomniacUnwinding

lorenzomeacci.com/unwind-data-cant-sleep-introducing-insomniacunwinding

Lorenzo Meacci March 29, 2026

Hi all, in this blog we will discuss sleep masking in detail, the default assumptions that come with it, and how we are going to break those assumptions with a novel approach called InsomniacUnwinding.



Prerequisites

This is a continuation of my previous blog ***Bypassing EDR in a Crystal Clear Way***, so I recommend reading that first to fully understand what we are building on today.

The Sleepmask source code is here -> ***InsomniacUnwinding***

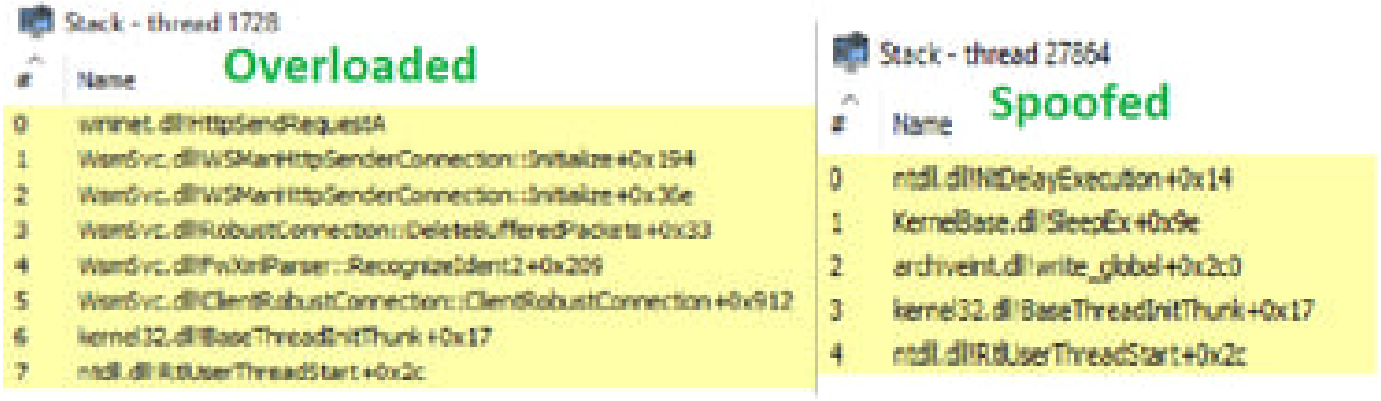
The source code for the cross-process POC is here -> ***InsomniacUnsindingCrossProcess***

Quick Recap

In the previous blog, we created a reflective loader using Crystal Palace that worked with Cobalt Strike beacon. If you remember, sleep masking was implemented in a separate PICO (Position Independent Code Object) that lived in unbacked RX memory. Because of its nature, code originating from unbacked memory will not have a legitimate looking call stack. The *why* was discussed in detail in that blog, and for this reason call stack spoofing techniques like Draugr needed to be implemented.

You might also remember that beacon was overloaded into a legitimate DLL and its .pdata section was registered, allowing Windows to properly unwind beacon's API calls and functions. But during the sleep cycle, the Sleep() API was hooked via the IAT and redirected to the PICO, which then encrypted the beacon sections before calling the real Sleep API via spoofing. The flow looked something like this:

The fact that spoofing was needed for the entire duration of the beacon's life frustrated me, and I felt this was a "waste" of potential for that fantastic overloading technique. So that's when I started playing around with memory masking and exploring what we could do to preserve the unwind info at sleep time. But that's not the whole story, as we will see in a bit. Where the sleepmask code lives is just as important as the unwind preservation technique we will discuss.



Unwind deep dive

To understand what we will do next, we first need to know what Windows does when unwinding and resolving the stack of a running application.

x64 Unwinding Basics

On x86, stack walking was straightforward. Functions typically set up a frame pointer using push ebp; mov ebp, esp, creating a linked list of frame pointers that debuggers and exception handlers could follow. You could walk the stack by just chasing EBP values.

x64 changed everything. For performance reasons, most functions no longer use frame pointers. RBP is often used as a general purpose register instead. So instead of using frame pointers Windows uses metadata that exists in the PE itself.

The Chain of References

When Windows needs to unwind the stack (for debugging, exception handling etc), it follows a chain of references baked into the executable.

- The **PE headers** contain a DataDirectory array. Entry index 3 (IMAGE_DIRECTORY_ENTRY_EXCEPTION) holds the RVA and size of the exception data. This points to the .pdata section.
- the .pdata section is an array of RUNTIME_FUNCTION structures, one per function in the PE

  The **UNWIND_INFO structures** live in .rdata and describe exactly how to reverse each function's prolog (asm instructions added by the compiler for setting up the stack and registers before a function starts):

How the Unwinder Works

When RtlVirtualUnwind needs to find the caller of a function, it:

- Takes the current instruction pointer (RIP)
- Binary searches .pdata to find which RUNTIME_FUNCTION contains that address
- Follows UnwindInfoAddress to locate the UNWIND_INFO structure in .rdata
- Reads the UNWIND_CODE array and "undoes" the prolog in reverse: if a register was pushed, pop it. If stack space was allocated, add it back to RSP.
- After reversing the prolog, RSP points to the return address
- Pops the return address into RIP and repeats

This continues until RIP is NULL or lands in kernel space, at which point the stack has been fully walked.

Why do we care?

When we perform sleepmasking we are encrypting all beacon memory starting from the base of the image base address, this way we encrypt everything including:

- The PE headers (Windows cannot find the exception directory)

- The .pdata section (Windows cannot find RUNTIME_FUNCTION entries)
- The .rdata section (Windows cannot read UNWIND_INFO structures)

If we encrypt any of these RtlVirtualUnwind will fail and the stack walker cannot determine frame sizes, cannot figure out where return addresses are, and cannot walk backwards through the call chain. Your call stack turns into garbage.

This is why naive encryption breaks stack walking entirely, and why we need to be surgical about what we preserve.

The architecture - My first POC

The first POC is here - ***InsomniacUnwindingCrossProcess***

I put my hands forward and I want to specify that this POC is only for demonstration purposes, the actual architecture for C2 frameworks will be discussed later in the blog. This POC was created to have a simple implementation of the technique without the need to modify other projects that would overcomplicate the explanation.

The Goal

We want to encrypt beacon memory during sleep while preserving valid stack unwinding. The cross-process architecture is the simplest way to demonstrate this.

The Solution: Cross Process Sleep Masking

The sleepmask lives in a completely separate process. The beacon does not encrypt itself. Instead, it asks another process to encrypt it from the outside.

The beacon connects to a named pipe, sends a request with its PID, image base, and sleep duration, then blocks on ReadFile waiting for a response. The sleepmask process receives the request, opens a handle to the beacon process, encrypts its memory, sleeps, decrypts, and sends a response back. The beacon wakes up and continues.

Why Spoofing Is Not Needed

In this POC both processes are regular executables, but imagine both were stomped into legitimate DLLs. The beacon lives in backed memory with registered .pdata. The sleepmask also lives in backed memory. When the sleepmask calls Sleep, ReadProcessMemory, WriteProcessMemory, those calls originate from backed addresses. No spoofing needed on either side.

The beacon's thread is just blocked on a pipe read. It never leaves its original context. If we keep the unwind data readable, the call stack will resolve cleanly through the stomped module, down to BaseThreadInitThunk and RtlUserThreadStart.

Preserving Unwind Data

From the sleepmask's perspective, the process is straightforward:

1. Receive the sleep request containing the beacon's PID, image base, image size, and sleep duration
2. Open a handle to the beacon process
3. Read the beacon's entire image into a local buffer
4. Save copies of the regions we need to preserve
5. Encrypt the local buffer
6. Patch back the preserved regions over the encrypted bytes
7. Write the modified buffer back to the beacon process
8. Sleep for the requested duration
9. Repeat the process in reverse to decrypt

The IPC protocol is simple. We use the following structures to communicate and pass data between the two processes.

First Implementation: Preserving Full Sections

Based on what we know about unwinding, we need to preserve:

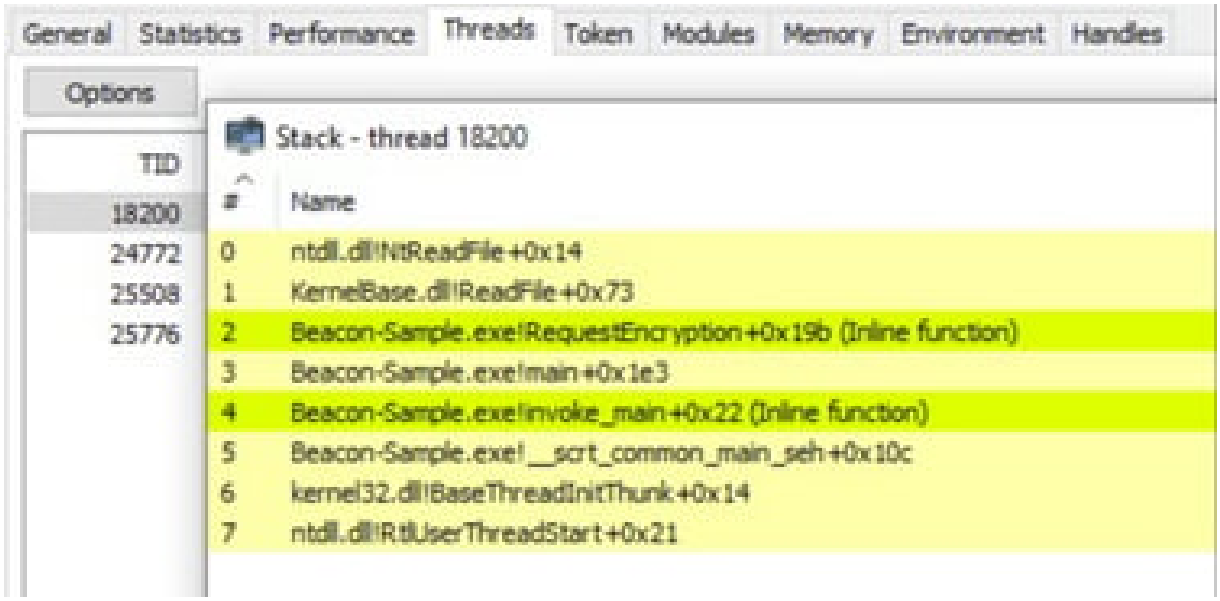
- PE headers (contains DataDirectory[IMAGE_DIRECTORY_ENTRY_EXCEPTION])
- .pdata section (the RUNTIME_FUNCTION array)
- .rdata section (contains UNWIND_INFO structures)

Now I know what you are thinking: ".rdata can contain signatures! And we are patching it back?!" At the moment yes, but don't worry, a surgical approach is showcased later (and is actually what is inside the POC on GitHub).

The preservation logic finds each section and saves a plaintext copy before encryption:

After encryption, we patch back the preserved regions:

The call stack now resolves correctly during sleep:



Every frame resolves through the beacon module down to BaseThreadInitThunk and RtlUserThreadStart. No spoofing required.

The Problem: YARA Still Hits

The call stack works, but we have a different problem. The .rdata section contains more than just UNWIND_INFO structures. It contains string literals, const arrays, import names, vtables, and other signaturable artifacts.

I added some signature bytes to the beacon to test this:

And created this YARA rule:

Running YARA during sleep:

The .data signature is encrypted and gone. But the .rdata signature is still there because we preserved the entire section. Any signatures living in .rdata remain visible to memory scanners during sleep.

Preserving full sections fixes stack walking but defeats the purpose of sleep masking. We need to preserve less.

Surgical UNWIND_INFO Extraction

We do not need the entire .rdata section. The stack unwinder only reads the specific UNWIND_INFO structures that .pdata references. Everything else in .rdata (strings, const data, vtables, import names) is irrelevant to stack walking.

The approach is simple: parse .pdata, follow each UnwindInfoAddress, calculate the exact size of each UNWIND_INFO structure, and preserve only those bytes.

Finding the UNWIND_INFO Structures

As we saw earlier each RUNTIME_FUNCTION in .pdata contains an UnwindInfoAddress field pointing to an UNWIND_INFO structure in .rdata:

We iterate through every RUNTIME_FUNCTION, extract the UnwindInfoAddress, and record that location. Multiple functions can share the same UNWIND_INFO if they have identical prologs, so we track unique addresses to avoid duplicates.

Calculating UNWIND_INFO Size

UNWIND_INFO is a variable length structure. The size depends on the number of unwind codes and optional trailing data:

The size calculation:

The Extraction Logic

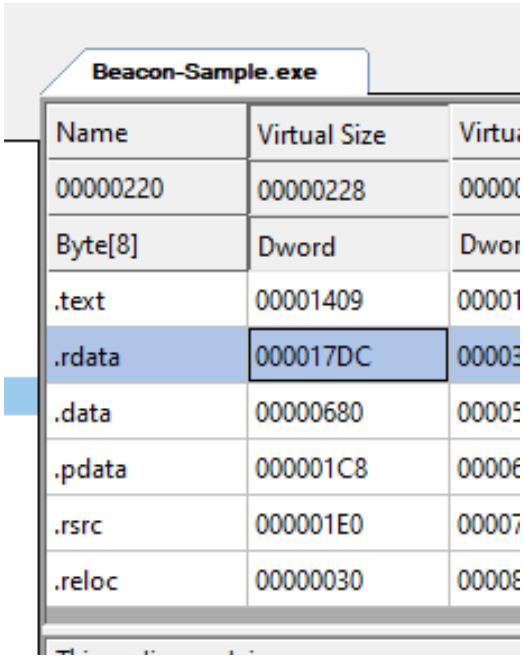
We build a list of regions to preserve: PE headers, .pdata, and each individual UNWIND_INFO:

Result

Sleepmask Output:

Yara scan:

for comparison the size of .rdata is 6108 bytes:



and we only patched back 252 bytes!!! That's about **4%** of the section. The other 96% (strings, const data, signatures) gets encrypted.

Real World Architecture - Modifying Ekko

How Ekko Works

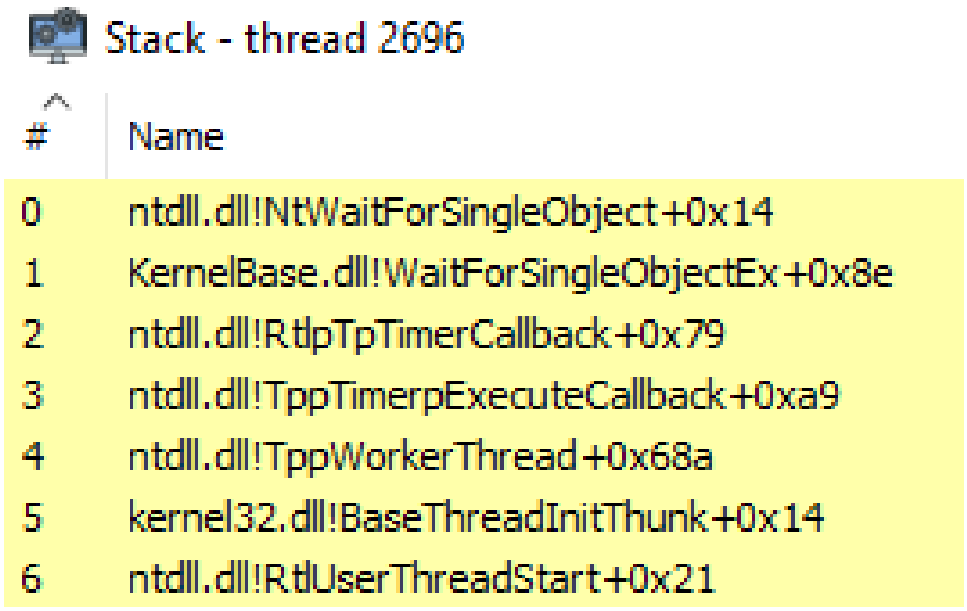
Ekko is a timer-based sleep masking technique. Instead of encrypting memory directly (which would encrypt the code that needs to do the decryption), it queues a chain of timers that the OS executes on your behalf.

Timers in Windows are callbacks scheduled to fire after a specified delay. When you call CreateTimerQueueTimer with the WT_EXECUTEINTIMERTHREAD flag, Windows executes your callback on a dedicated timer thread from the thread pool. The trick Ekko uses is that NtContinue can be used as a callback. NtContinue takes a CONTEXT structure and switches execution to whatever state that context describes. By crafting CONTEXT structures, you can make Windows call VirtualProtect, SystemFunction032 (RC4 encryption), WaitForSingleObject, and so on, all without your own code executing.

This is how the entire image can be encrypted while sleeping. The timer thread does all the work. Your main thread just waits on an event until the chain completes.

The Timer Thread

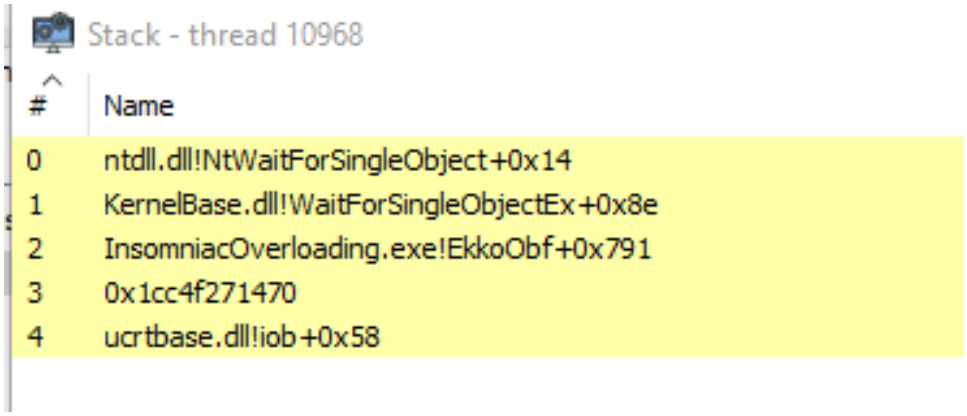
Here is what the timer thread's call stack looks like during the sleep:



The timer thread has a perfectly clean stack with BaseThreadInitThunk and RtlUserThreadStart at the bottom. Thread pool workers are legitimate threads initialized through normal paths. No spoofing needed for the timer thread itself.

The Main Thread Problem

This is what the main thread call stack would look like with the default Ekko sleepmask:



In-Process Timer-Based Implementation

The cross-process POC is clean and easy to understand, but it is not the only way to apply surgical UNWIND_INFO preservation. The same technique works perfectly with timer-based approaches like Ekko.

The core concept is identical to the cross-process POC: any thread that has your code on its call stack needs readable unwind data during sleep. When you call WaitForSingleObject from EkkoObf, the main thread's stack contains frames like main -> EkkoObf -> WaitForSingleObject. If a debugger or EDR walks that stack, it needs to read the UNWIND_INFO for EkkoObf to calculate frame sizes and find return addresses.

As explained earlier, encrypt the unwind data and the stack turns to garbage. Preserve it and the stack resolves correctly.

The Implementation

I modified the Ekko sleepmask to preserve PE headers, .pdata, and surgically extracted UNWIND_INFO regions, exactly like the cross-process POC does it. The challenge is that each region needs its own RtlCopyMemory call, which means its own CONTEXT structure and timer.

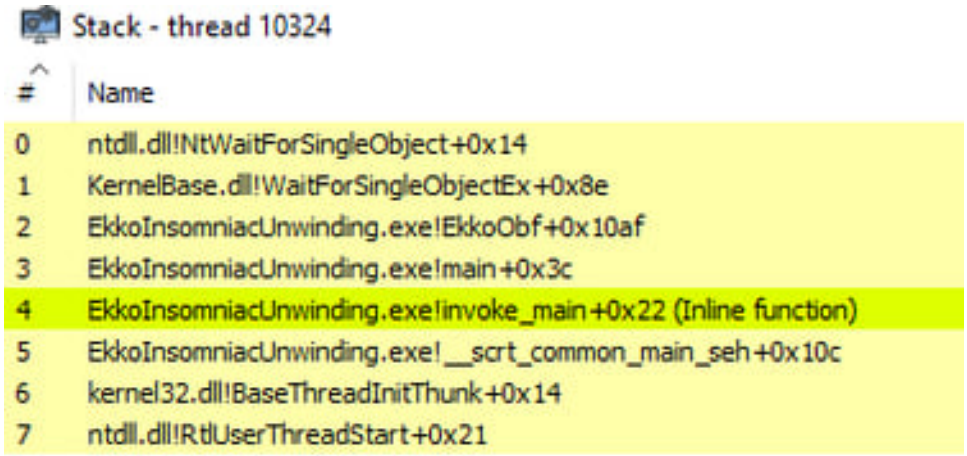
First we find all UNWIND_INFO regions using the same extraction logic:

Then we create CONTEXT structures for each patch. We need two sets: one for patching before sleep, one for patching after decryption (because decryption actually turns into garbage the plain text regions):

The timer chain is built dynamically based on how many regions we need to patch:

The Result

The main thread's call stack resolves correctly during sleep:



The Critical Point

The InsomniacUnwinding technique works regardless of architecture. The important constraint is where the sleepmask executes from.

If the sleepmask runs from unbacked memory, the call stack will show unbacked return addresses during the Sleep call. You are back to needing spoofing.

If the sleepmask and beacon run from backed memory (stomped module), the call stack is clean. No spoofing needed.

The unwind preservation handles the beacon side. The sleepmask location handles the sleep call side. Both must be in backed memory for a fully clean call stack without spoofing.

Conclusions and acknowledgements

This blog post and the initial POC were modified 1 day after being published because Alex Reid **@Octoberfest73** spotted a mistake I made about timer threads. This was actually huge because I initially assumed the InsomniacUnwinding technique was limited to architectures that required the sleepmask code to live in a separate memory region from the beacon, when in reality, that's far from the truth! InsomniacUnwinding can be performed by self-encrypting sleepmasks, too. This was showcased exactly in the Ekko modification.

This research was done out of pure fun and curiosity. I also like to challenge myself when some techniques are considered mandatory to achieve the desired output.

I hope to see this technique implemented in the future by someone less lazy than myself. For now, this is all. Happy hacking!

I know this screenshot demonstrates nothing. I just find it funny xD