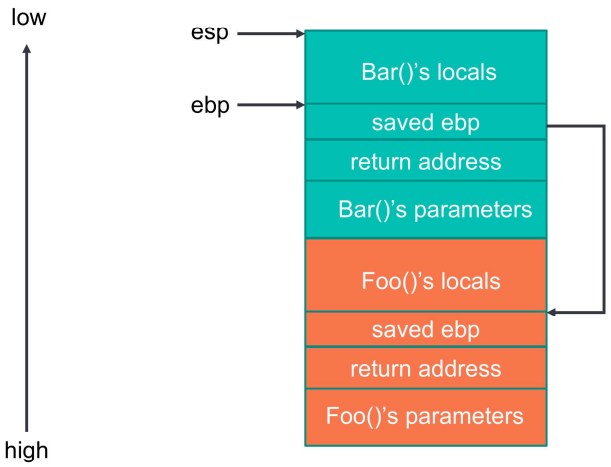
This talk covers the differences between x86 and x64 stack walking on Windows – and the implications for security folks.

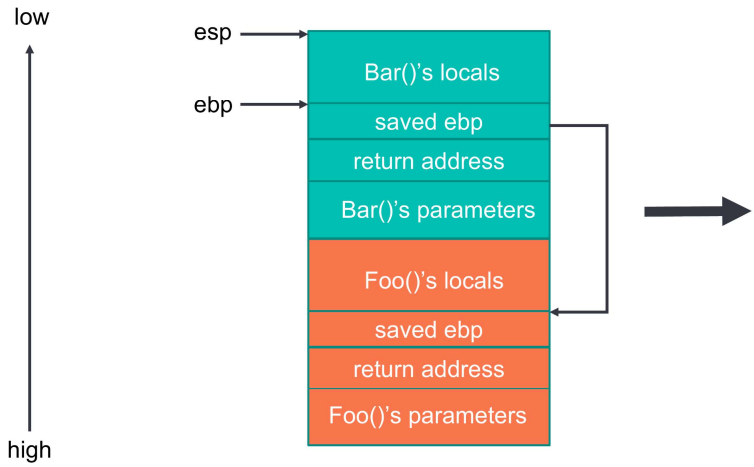
x86 stack refresher

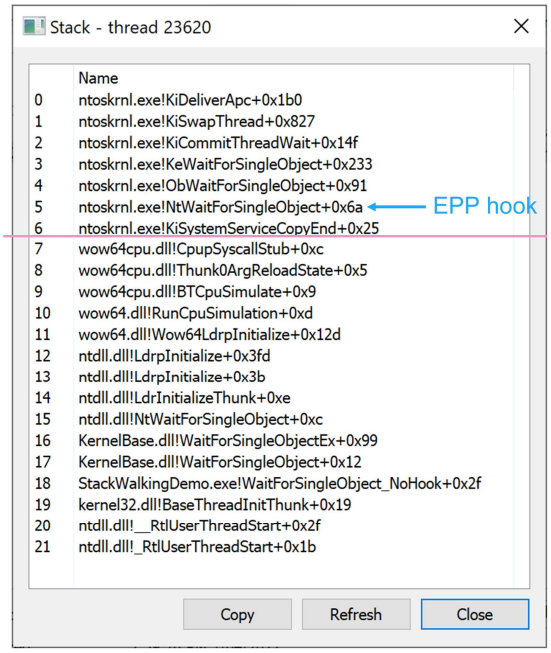


elastic

Stacks grow down.
Except when you view them upside down in a debugger memory window.

x86 stack refresher



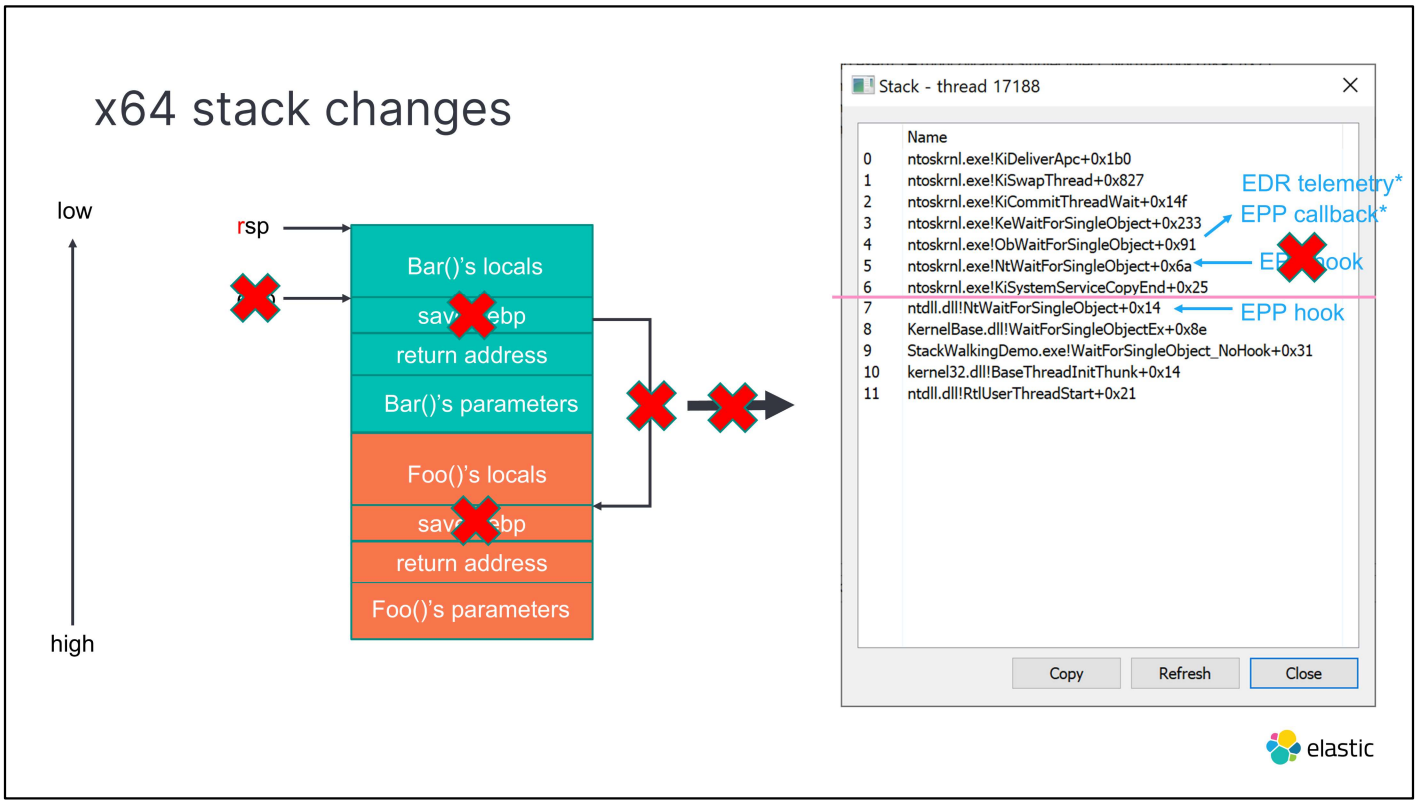


elastic

Call stacks can be reconstructed by following the saved frame pointers. Originally this was done for debugging purposes – but determining a call's provenance is highly useful for security products too.

Edge case - compilers could choose to omit the base pointer for performance.

Aside – on x86 kernels hooks were ubiquitous in EPP products.



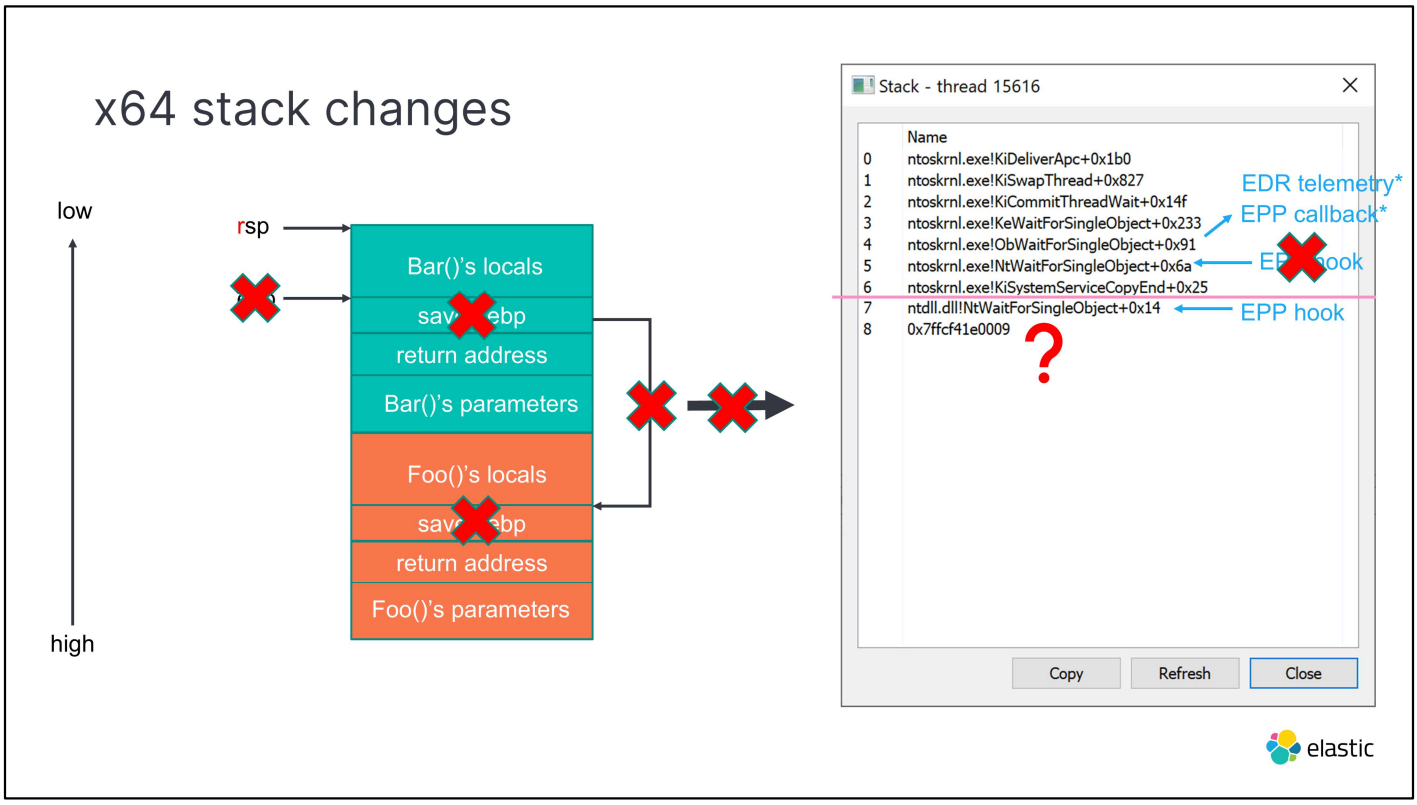
x64 did away with the default base pointer – this saved a whole register and improved performance.

x64 also did away with arbitrary kernel hooks. (PatchGuard says no). Instead, security vendors were offered a handful of kernel callbacks, and later some additional kernel telemetry (ETW).

And so usermode hooks became prevalent.
For the older vendors, this was the quickest path to reuse existing detection logic.
And, for some newer vendors, it was the quickest path to market.

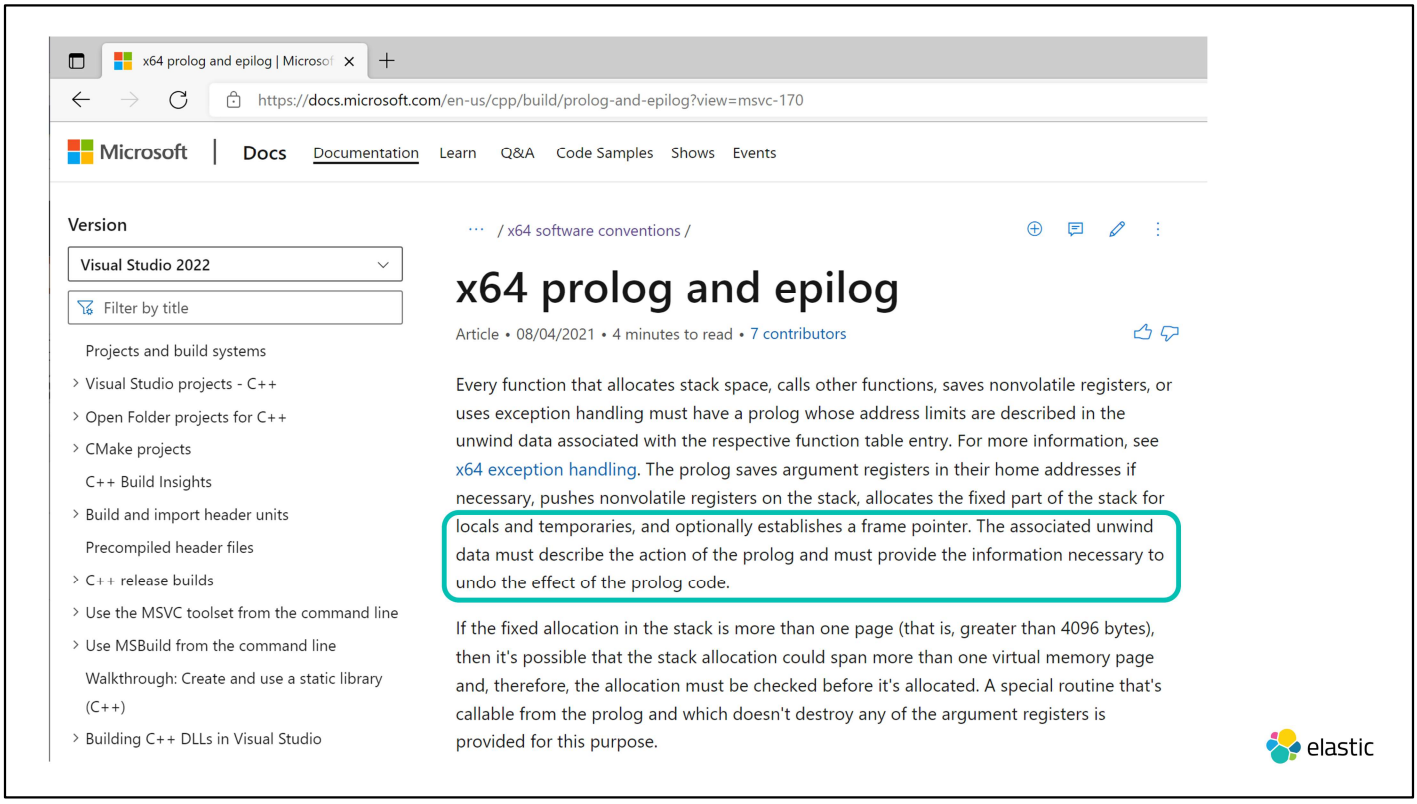
The majority of usermode hooks, especially system call hooks, are imho technical debt. Unless you're enforcing a sandbox for a constrained runtime environment its typically not a defensible boundary.

Meanwhile, without the base pointer… how where call stacks being generated?
It was working so I didn't give it much thought…



…but then I noticed that sometimes it didn't work.
And particularly if there was an unbacked return address.

So, it was time to learn more about how x64 managed its stack walking magic.



So, I read the manual.

When an exception occurs, you need to be able to leave a function without explicitly reaching the function epilog. For x64 Microsoft achieved this by creating unwind metadata during compilation and storing it in the exception directory in the PE file.

The purpose of this UNWIND_INFO is “undo the effect of the prolog code”.
So, stack walking is just looking up your stack frame size in the UNWIND_INFO in the image’s exception directory.

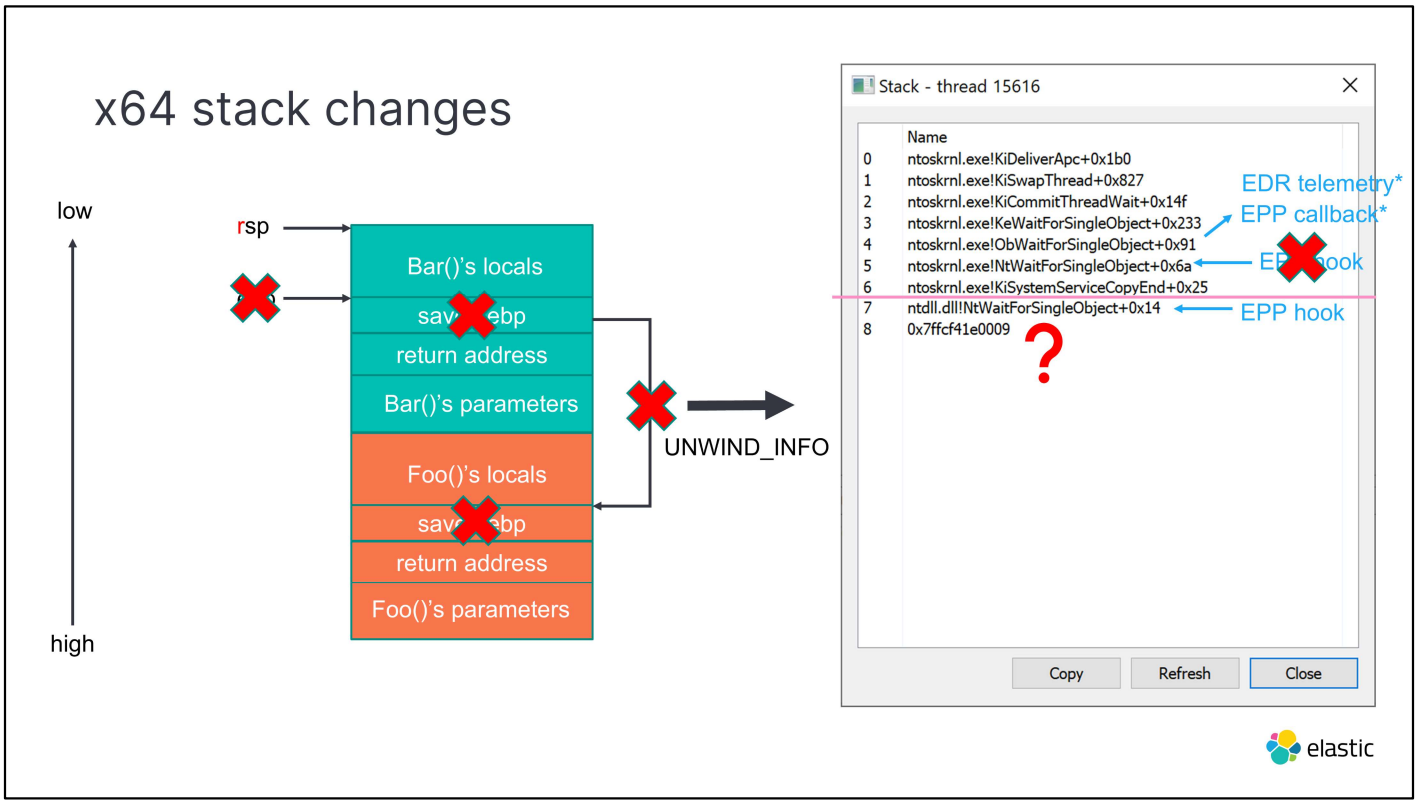
Which explains why it (almost) never works for unbacked code.

x64 stack walking

- On x64 stack walking relies on registered UNWIND_INFO in the RuntimeFunctionTable.
  https://docs.microsoft.com/en-us/cpp/build/exception-handling-x64

- For PE files, this information is stored in the Exception Directory. However, if the function doesn't have an exception handler, then this information seems to be in the PDB.
  https://social.msdn.microsoft.com/Forums/windowsdesktop/en-US/97a969b2-de80-42d4-b1a0-a7298e3695cb/stackwalk64-on-x64-stack-and-the-pdb-symbols

- JIT code should dynamically register its unwind information but may choose to install a dynamic function table callback instead. **Some JIT engines do neither.**
  https://docs.microsoft.com/en-us/windows/win32/api/winnt/nf-winnt-rtlinstallfunctiontablecallback
  .NET uses the callback approach and StackWalk64 does not query this by default.
  ~~Process Hacker~~ System Informer handles this. Process Explorer does not.



But it was still annoying. I wanted a proper call stack.

In this instance, I was in a kernel callback to a create thread notification routine. And I noticed that there was always a one of a handful of other EPP vendors also installed whenever I encountered a corrupt stack.

So, I had a hypothesis that the usermode hooks of some EPP vendors were corrupting the call stack. (Though why any EPP vendor ever really needs to hook CreateThread I'd like to know…)

I did some hooking myself – but I couldn't reproduce what was going on here…
My hooks didn't break call stacks.
And unfortunately, I didn't have copies of any of the EPP products.

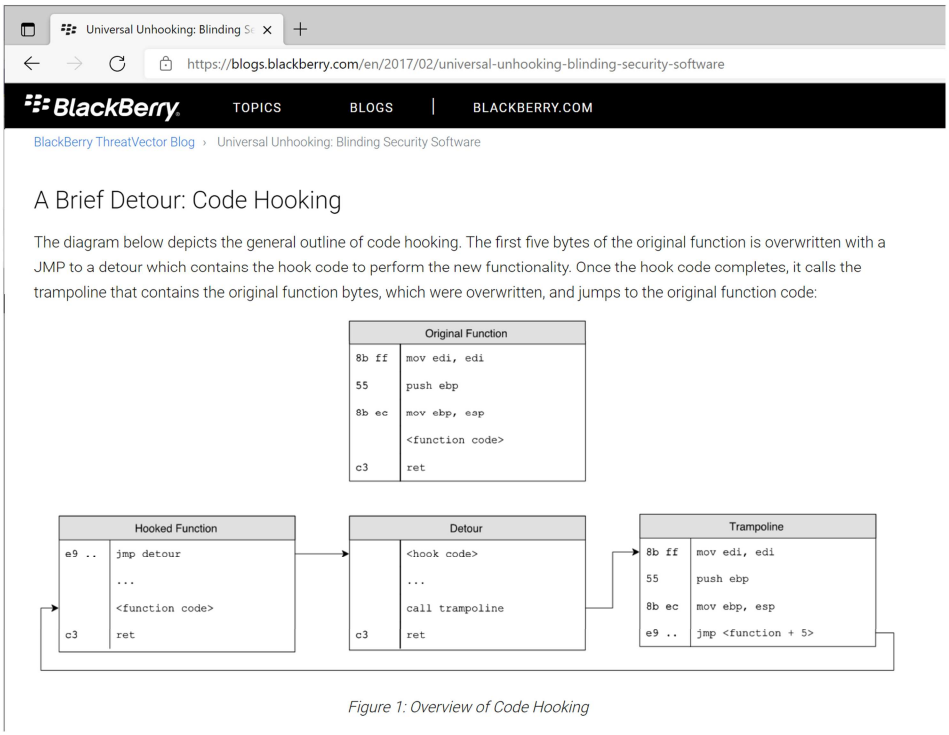
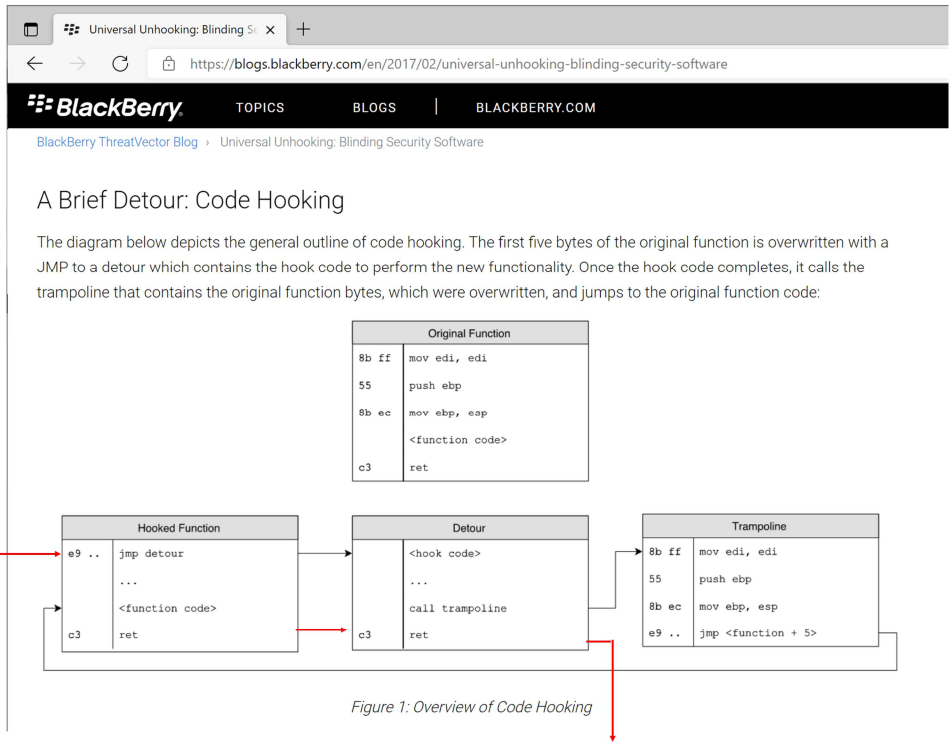


Figure 1: Overview of Code Hooking

elastic

But one of those vendors was Cylance. And they write some helpful blogs.
And one included an overview of Code Hooking.

So… they jump to a Detour which finishes by **calling** a Trampoline – that contains the overwritten instructions and a jump back to the original function.
The original function that returns into the Detour which then returns to the caller.



elastic

This convoluted flow means that the detour return address is on the call stack whenever the hooked function makes a syscall. ☹

And, the lack of registered UNWIND_INFO, blinds the kernel to the true provenance of the call.

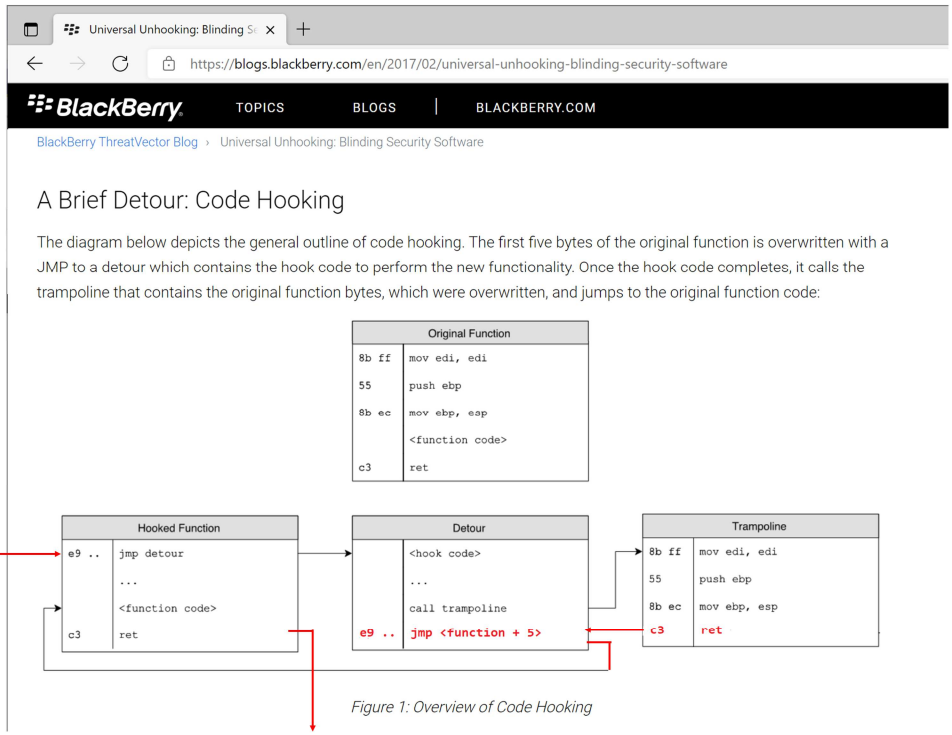
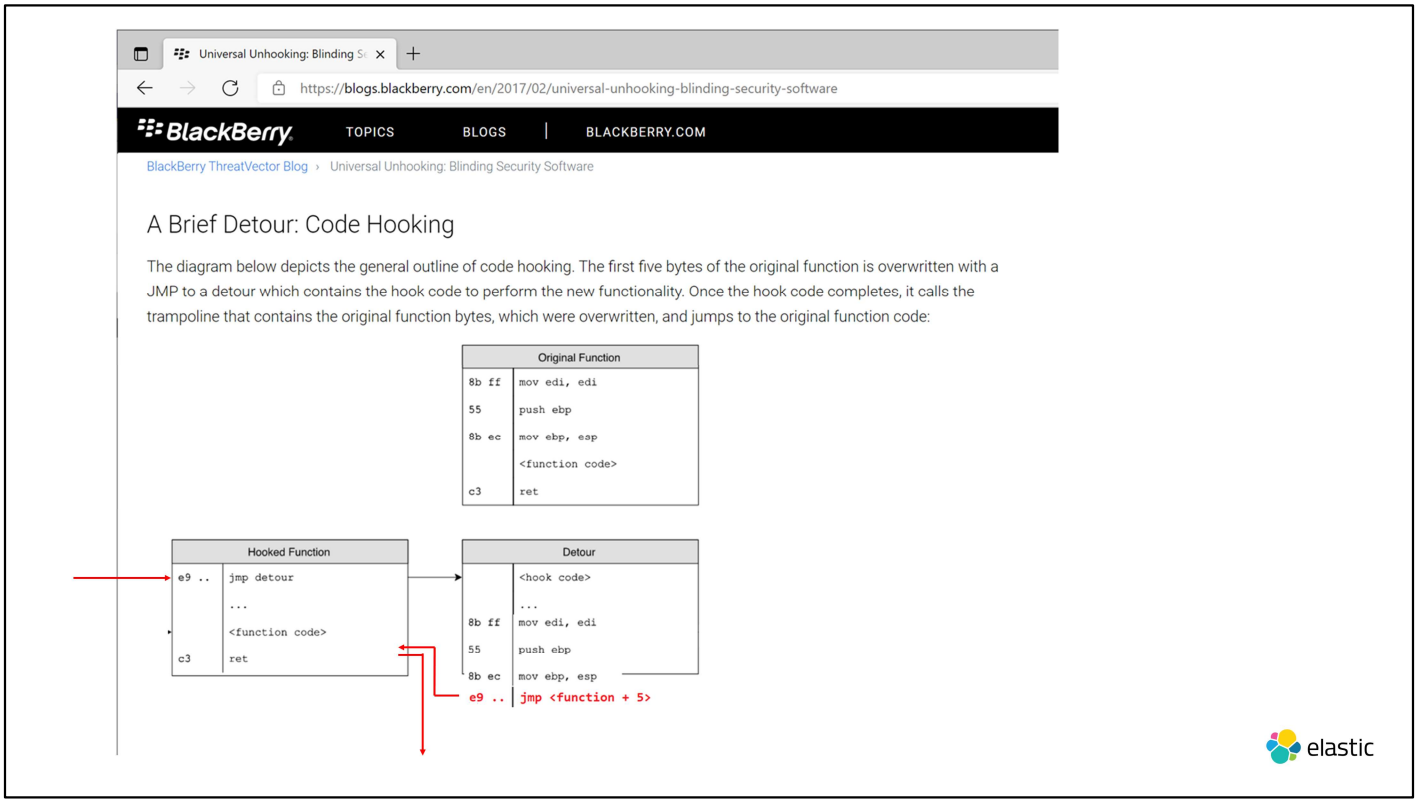


Figure 1: Overview of Code Hooking

elastic

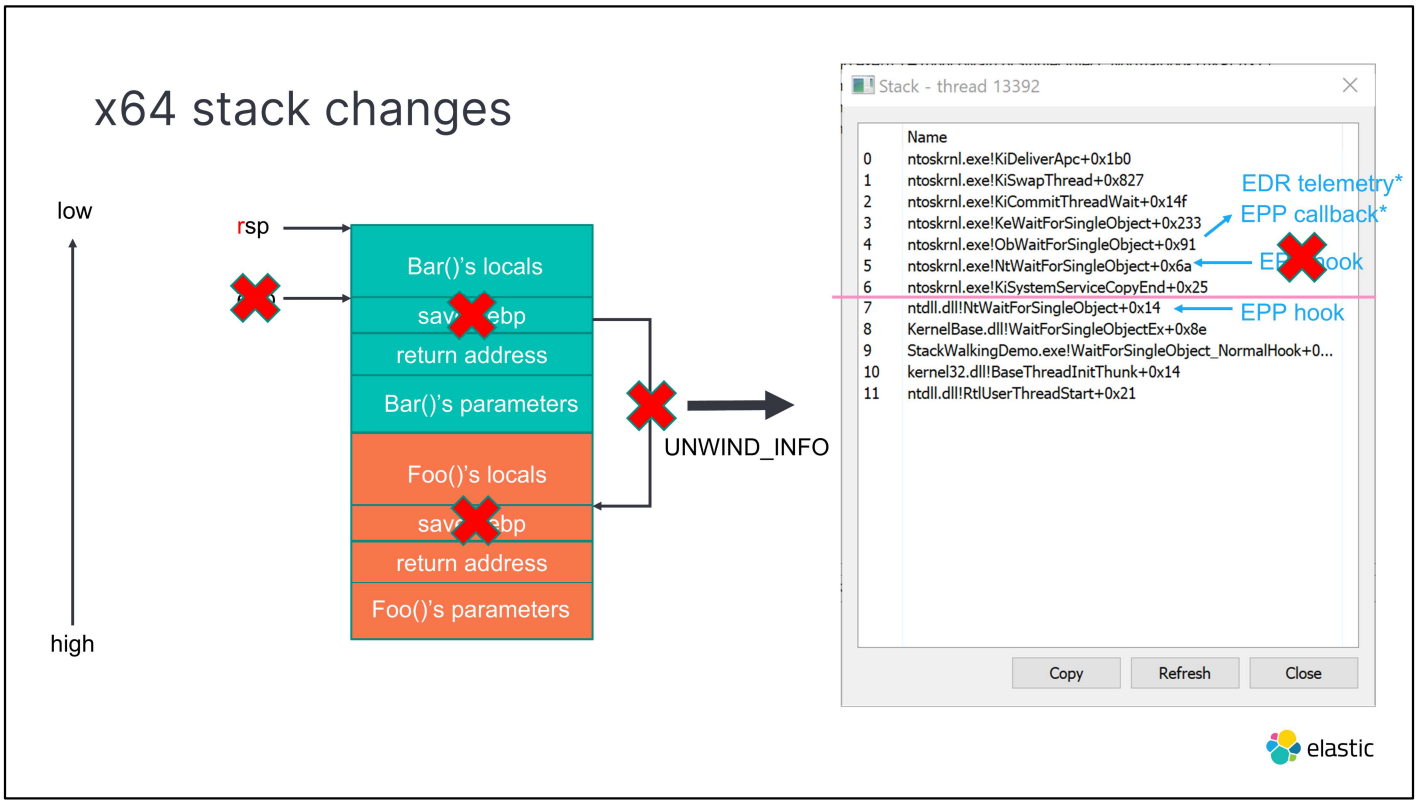
Maybe it's just the offensive researcher in me, but that unbacked return address on the stack is totally unnecessary in this tail call scenario.

A saner approach would be to return to the detour let it jump back to the original function.
No dangling unbacked return addresses.

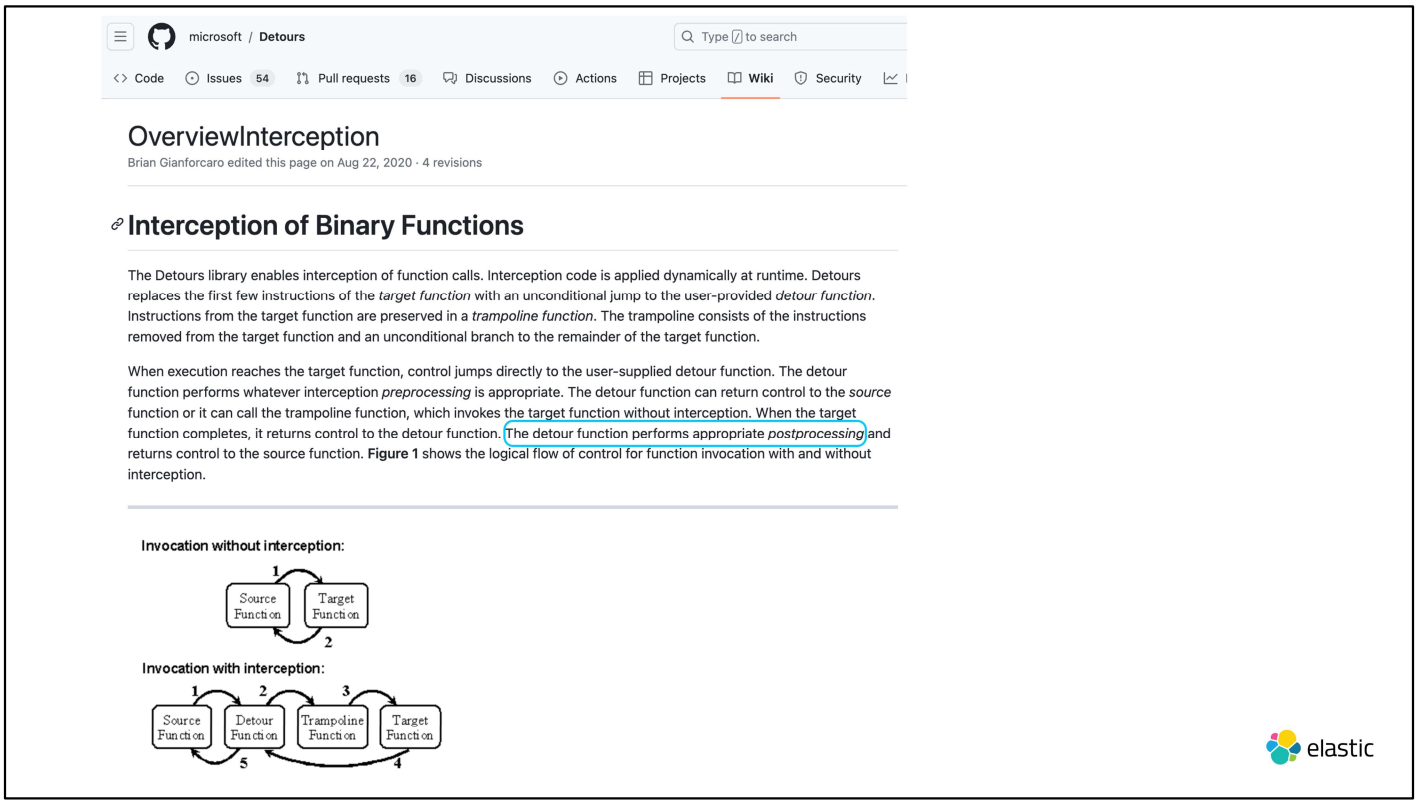


In fact, the whole detour + trampoline approach just feels weird when **both are in private memory**.
Just put the trampoline at the end of the detour…
(The trampoline needs to be on an instruction boundary – so needs to be determined at runtime. But it is at most 19 bytes. Our hook is 5 bytes, and there is a 15 byte limit on x64 instruction size.)

Cylance's approach is 100% not how I would implement native API hooking in a security product.



As mentioned earlier, with my sane hooking approach – calls stacks are possible!



But it wasn't actually completely insane.
My next clue was Comodo – another product that was messing with my call stacks. Comodo gave us OpenEDR. OpenEDR uses Microsoft Detours (unless the commercial madCodeHook library is available). And the Microsoft Detours interception pattern looks **exactly like the one that Cylance describes**.

These hooking libraries just have way more flexibility than what is required by the native API pre-call hooking scenario. But they **assume that your Detour function is in a DLL**.
In fact, it is actually 98% how I would design an instrumentation hooking API where I wanted to be able to flexibly call the target function at any point during my Detour – or even never call it at all.

But maybe… hear me out here… **maybe a security product shouldn't just write unnecessary JIT code into every process**.
The Detour function should always be in MEM_IMAGE – meaning that it'll have UNWIND_INFO
Only the Trampoline is dynamic. And only barely so in this case.

PS I opened PRs on Microsoft Detours to address the last 2% -

- https://github.com/microsoft/Detours/pull/307 - update system DLL range logic for 64-bit Oses
- https://github.com/microsoft/Detours/pull/308 - don't break stack walking

x64 JIT stack walking – dynamic function tables

Legacy APIs

- `RtlAddFunctionTable`
- `RtlInstallFunctionTableCallback` - .NET Framework

elastic

How is JIT (aka private code regions) function stack walking supposed to work?

Microsoft initially provided JIT engines with 2 mechanisms to register a function table at runtime.

- The ability to register a fixed function table for a given region of JIT code.
- The ability to register a callback DLL that will generate a function table just-in-time – this is what the original .NET framework uses.

The fixed approach wasn't suitable for most dynamic code scenarios. The table also didn't need to be sorted – so lookups could be slow.
And the callback approach had multiple drawbacks – lots of developer effort to implement plus the debugger had to load and call into a 3rd party dll.
And notably neither was supported by ETW call stacks – Microsoft's native tracing solution.

x64 JIT stack walking – dynamic function tables

Legacy APIs

- `RtlAddFunctionTable`
- `RtlInstallFunctionTableCallback` - .NET Framework

Modern API

- `RtlAddGrowableFunctionTable` - .NET Core

elastic

So, a third approach was added in Windows 8.

The ability to register a growable function table for a given region of JIT code. And this time it was required to be pre-sorted.

This is what modern .NET Core uses.

x64 JIT stack walking – dynamic function tables

Legacy APIs

- `RtlAddFunctionTable`
- `RtlInstallFunctionTableCallback` - .NET Framework

Modern API

- `RtlAddGrowableFunctionTable` - .NET Core

`StackWalk64` (user-mode) does not query any dynamic function tables by default. 😨

`RtlWalkFrameChain` (kernel-mode) queries the modern (sorted) table. 😀

elastic

However, for reasons unknown, the user-mode StackWalk64 API does not query dynamic function tables by default – only static ones.
It does, however, let you extend it with custom function table lookup functions.
We couldn't find any Microsoft documentation on how to do this though – but thankfully there is System Informer source code.

The kernel-mode RtlWalkFrameChain API will query the modern growable tables only.
This exported function isn't officially documented by Microsoft – but you can find it mentioned on some OSR driver forums.

x64 stack walking – manually

- Disassemble until epilog

```
add     RSP, fixed-allocation-size
pop     R13
pop     R14
pop     R15
ret
```

elastic

But knowing how to solve the problem for code that I write didn't help my immediate use case – how to walk calls stack for code that didn't adhere to these conventions?

Could I manually walk the stack?

Functions have one prolog, and one or more epilogs. If I can find one of these epilogs then I can calculate the frame size.
Finding an epilog is easy. Code naturally goes forward.

x64 stack walking – manually

- But frame pointers…

```
lea     RSP, fixed-allocation-size - 128[R13]
pop     R13
pop     R14
pop     R15
ret
```

- "Disassemble backwards" until prolog

```
mov     [RSP + 8], RCX
push    R15
push    R14
push    R13
sub     RSP, fixed-allocation-size
lea     R13, 128[RSP]
...
```

elastic

Remember frame pointers? Well, they still exist in x64 code. Sometimes it improves performance to use a register to point *somewhere* within the frame and then use efficient rel8 instructions to reference locations on the stack.

Unfortunately, it also makes sense for the compiler to coalesce trimming the stack relative to the frame pointer and trimming the fixed allocation into a single instruction.
This means that we can't determine the fixed-allocation-size from the epilog without knowing the value in the frame pointer register.

But… we can determine it from the prolog.
Unfortunately finding a prolog is not trivial.
Reverse decompilation is not a thing. And absolute jumps are a thing.
Things in our favour –

- Compilers like locality.
- Prologs have stricts rules on form.
- Prologs must match epilogs – same order of registers.
- Return addresses are preceded by call instructions.

So, we can write an okay heuristic.

x64 stack walking – manually

- But...

_malloca

Article • 02/09/2022 • 3 minutes to read • 11 contributors

Allocates memory on the stack. This is a version of _alloca with security enhancements as described in Security Features in the CRT.

Syntax

C Copy

```
void *_malloca(
   size_t size
);
```

elastic

Even with all that, the C Runtime allows dynamic stack allocations.
And that size might be determined at runtime and unable to be calculated by code disassembly.

So, if everything else has failed then we could crawl the stack itself looking for return address candidates.
Things in our favour –

- Return addresses are preceded by call instructions.
- We may have identified the prolog.

Things not in our favour –

- Indirect calls.
- Tail call optimisations.

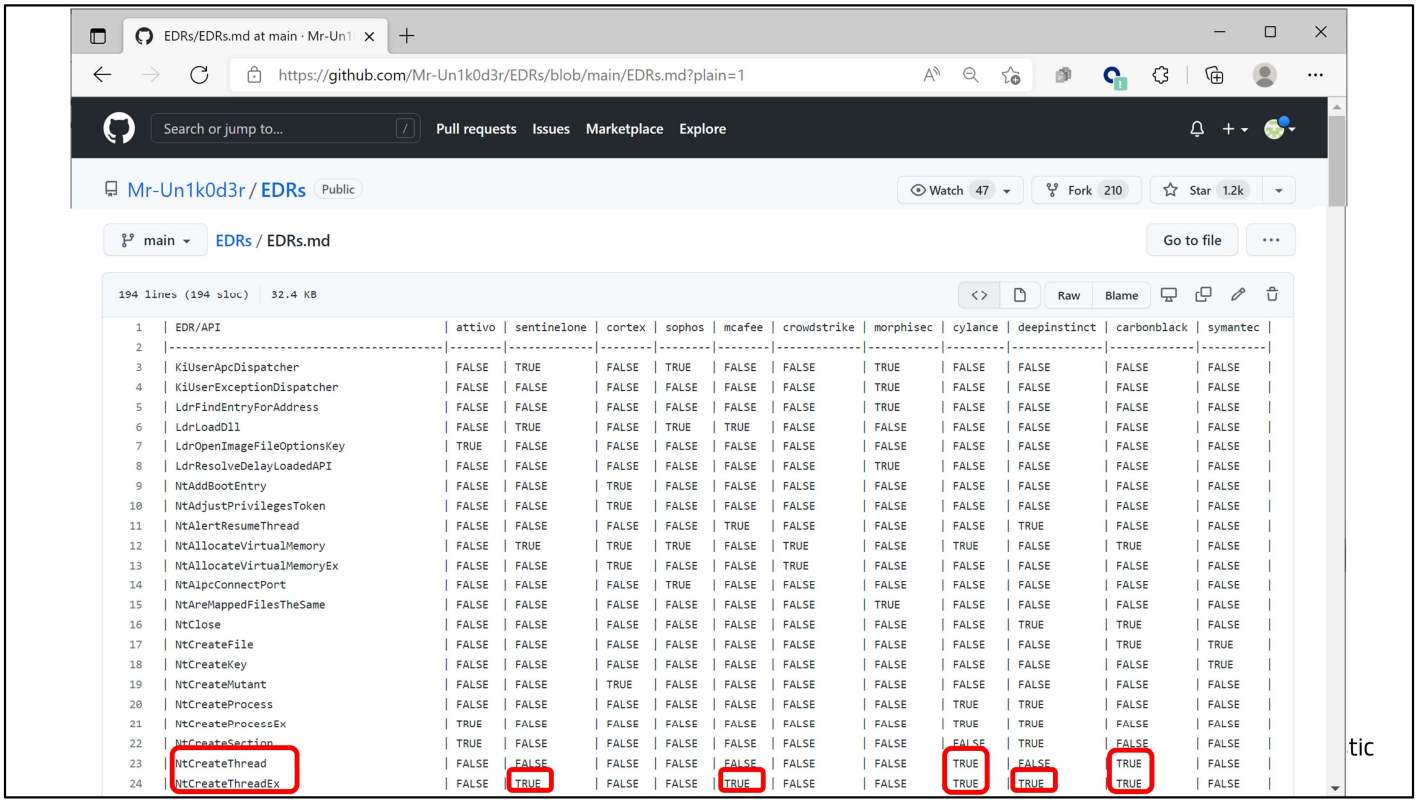
x64 stack walking – manually

- But atypical / out-of-spec compilers??
- But malware??

elastic

Manual stack walking is not perfect – and, being reliant on adherence to specification and compiler conventions, it is vulnerable to adversarial manipulation.

But it's good enough to distinguish between normal and weird.



EDR/API	attivo	sentinelone	cortex	sophos	mcafee	crowdstrike	morphisec	cylance	deepinstinct	carbonblack	symantec
KiUserApcDispatcher	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
KiUserExceptionDispatcher	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
LdrFindEntryForAddress	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
LdrLoadDll	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
LdrOpenImageFileOptionsKey	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
LdrResolveDelayLoadedAPI	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NtAddBootEntry	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
NtAdjustPrivilegesToken	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
NtAlertResumeThread	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE
NtAllocateVirtualMemory	FALSE	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE
NtAllocateVirtualMemoryEx	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
NtAlpcConnectPort	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
NtAreMappedFilesTheSame	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NtClose	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE
NtCreateFile	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE
NtCreateKey	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE
NtCreateMutant	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
NtCreateProcess	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
NtCreateProcessEx	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
NtCreateSection	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE
NtCreateThread	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE
NtCreateThreadEx	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	TRUE	TRUE	FALSE

tic

PS - Cylance is not alone.

I've seen corrupted call stacks likely caused by SentinelOne, BitDefender, Trend Micro, Comodo and Windows Defender Payload Restrictions* as well. (*part of the free WDEG/ASR - not Windows Defender for Endpoint).

Also – in my opinion the Tier 1 EPP products use thread creation *kernel* callbacks that aren't easily unhooked by usermode malware.
SentinelOne, McAfee, Cylance, Deep Instinct, Carbon Black and more all have *usermode* hooks on NtCreateThreadEx. I find this strange and would be keen to hear the reasons.

Further Reading

- https://codemachine.com/articles/x64_deep_dive.html
- https://labs.withsecure.com/publications/spoofing-call-stacks-to-confuse-edrs
- https://www.elastic.co/security-labs/finding-truth-in-the-shadows
- https://klezvirus.github.io/RedTeaming/AV_Evasion/StackSpoofing/
  - https://media.defcon.org/DEF CON 31/DEF CON 31 presentations/Alessandro klezVirus Magnosi Arash waldoirc Parsa Athanasios trickster0 Tserpelis - StackMoonwalk A Novel approach to stack spoofing on Windows x64.pdf
- https://www.cobaltstrike.com/blog/behind-the-mask-spoofing-call-stacks-dynamically-with-timers

elastic

Manual stack walking is not perfect and, being reliant on adherence to specification and compiler conventions, it is vulnerable to adversarial manipulation.

But it's usually good enough to distinguish between normal and weird.