An Introduction into Stack Spoofing

dtsec.us/2023-09-15-StackSpoofin/

Stack spoofing is a really cool malware technique that isn't new, but has been receving some more attention recently. The goal of this post is to introduce readers to the concept and dive into two implementations. This post will focus only on call stack spoofing in x64 Windows with "active" spoofing techniques.

Preface

None of the work here is novel. The majority of the stack spoofing research that I will be describing was done by Klezvirus, trickster012, waldo-irc, and namaszo. Massive thanks to them for providing the foundation for me to dig into such a cool topic.

During my internship at X-Force Red I got to work with 0xboku in researching the work above. Some of that work got to be integrated with BokuLoader, which was really awesome. Without Bobby's help I wouldn't have been able to understand these techniques as much, so massive thanks to him for sticking through several hour calls with my dumb questions.

Lastly, I am not an expert on this subject, this post is meant to just share what I understand with others trying to cook funny software.

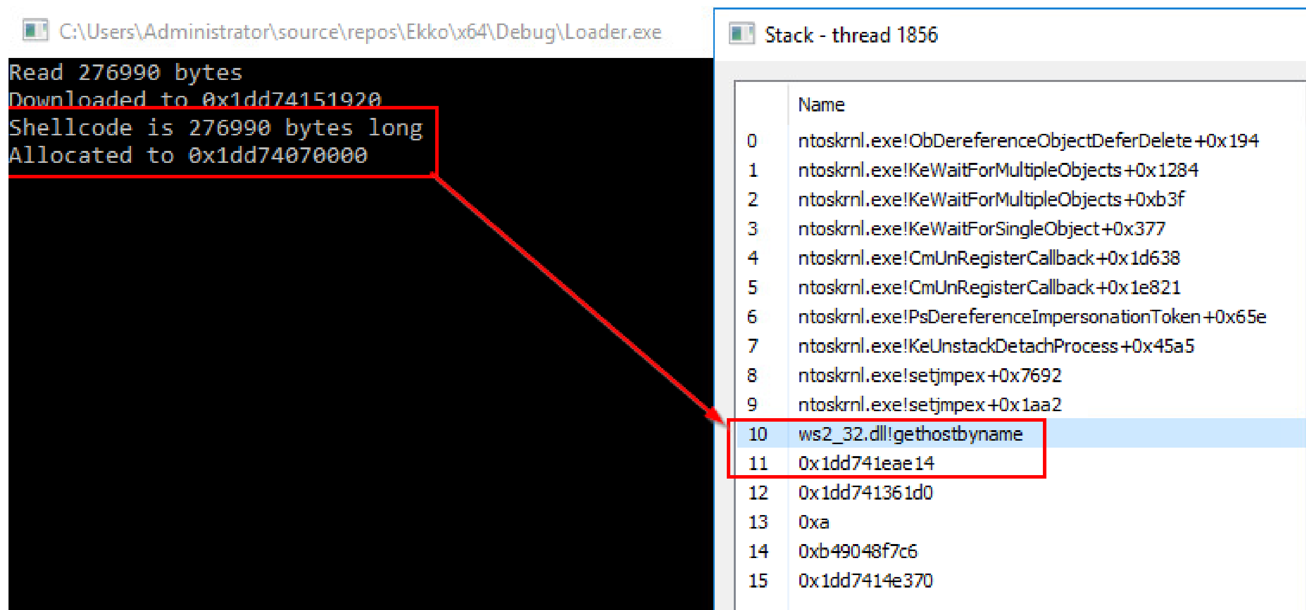
Introduction

EDR have more capabilities beyond inline hooking. It is possible for them to utilize the call stack of a function call to determine whether a function is malicious or not. Elastic has demonstrated this capability and I am guessing this call stack telemetry is obtained via

kernel callbacks registered for process and thread creation events, and probably more.

▶ Call stack?

Using a debugger, we can place breakpoints on some function calls and view our call stack. When our implant (in my case, meterpreter), makes a function call (`gethostbyname` in the screenshot below), we can see that the call stack points back to its location in memory.



A little note: the most recent caller is higher in the stack (lower number), each entry you go down is the above entry's caller. The additional addresses beyond the meterpreter return address are “leaked” stack values; when unbacked memory is encountered by a walk, it assumes the stack frame is just of size 8, causing it to just dump stack values until it encounters a 0.

▶ The size and the allocation don't line up!

From here, EDR could:

- Kill our implant because that address is RX (Read-Executable) memory unbacked by a file on disk
- Run a memory scan and kill our implant because it will probably get flagged
- Flag our process to warrant more investigation by an IR operator
- Some other voodoo

This is not ideal. So how can we address this? We can try spoofing our callstack; making it less suspicious by attempting to hide the source of our call. There are multiple ways to achieve this; this blog post will not cover all of them.

Active Spoofing vs Passive Spoofing

The techniques I will introducing fall under the “active” category; that is, they allow for the spoofing of any function being called. See the following projects:

Passive spoofing is something I sort of went over in my previous blog post and it is a type of spoofing that can only occur when the implant is sleeping. See the following projects:

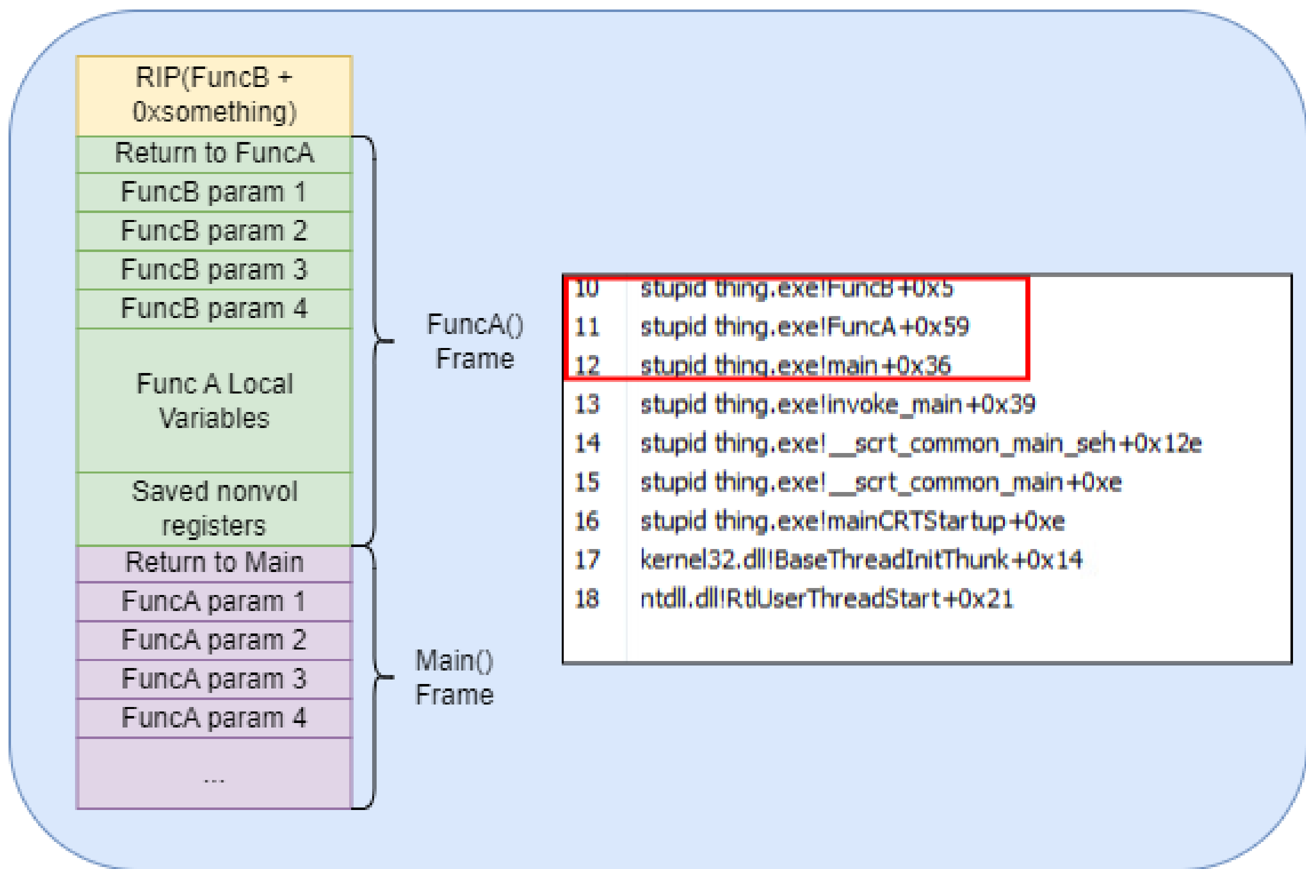
The active spoofing techniques I am covering have been done by the SilentMoonwalk project.

x64 Call Stack Primer

Unlike x86, only the RSP is necessary for keeping track of stack frames. Take the following code

```
void FuncC()
{
    return;
}
void FuncB(int a, int b, int c, int d)
{
    FuncC();
    return;
}
int FuncA(int a, int b, int br, int uh)
{
    int c = a;
    int d = b;
    FuncB(1, 1, 1, 1);
    return 5;
}
void main()
{
    FuncA(1, 2, 3, 4);
}
```

The contents of the call stack when FuncB() is called is below on the left. The Process Hacker stack walk view is on the right.



The space between the return addresses (the “Return to X”) is what I will refer to as the “stack size” of a function. Understanding how to calculate this stack size will be important for trying to spoof stacks as it is a common attribute utilized when unwinding the stack.

The .pdata section and unwind codes

The .pdata section is a PE section which includes information to assist with exception handling. More specifically, it is an array of `RUNTIME_FUNCTION`s.

```
typedef struct _IMAGE_RUNTIME_FUNCTION_ENTRY {
  DWORD BeginAddress;
  DWORD EndAddress;
  union {
    DWORD UnwindInfoAddress;
    DWORD UnwindData;
  } DUMMYUNIONNAME;
} RUNTIME_FUNCTION
```

The `BeginAddress` and `EndAddress` are offsets to the base of the PE. The actual code for the function this `RUNTIME_FUNCTION` belongs to is between those addresses. The union contains the interesting bit; an offset to the `UNWIND_INFO` struct.

```
typedef struct _UNWIND_INFO {
    BYTE Version : 3;
    BYTE Flags : 5;
    BYTE SizeOfProlog;
    BYTE CountOfCodes;
    BYTE FrameRegister : 4;
    BYTE FrameOffset : 4;
    UNWIND_CODE UnwindCode[1];
} UNWIND_INFO, * PUNWIND_INFO;
```

This `UNWIND_INFO` contains an array of `UNWIND_CODE`s. We can iterate through these to calculate the stack size of the function.

```
typedef union _UNWIND_CODE {
    struct {
        BYTE CodeOffset;
        BYTE UnwindOp : 4;
        BYTE OpInfo : 4;
    };
    USHORT FrameOffset;
} UNWIND_CODE, * PUNWIND_CODE;
```

There are different code types, with some affecting the stack size of a function by varying amounts.

Creating our own frames

With the information we have so far, we can try to create “synthetic” frames via the following steps.

1. Decrement `rsp` by the stack size
2. Set `[rsp]` to the a return address
3. Repeat at if we want to create another frame

Active Spoofing: Synthetic Frames

Note: This method was done by both VulcanRaven and SilentMoonwalk and is essentially an extension of namaszo’s return address spoofing, implemented by AceLdr by Kyle Avery.

▶ More details about the technique here

Putting everything together, this is what we need to accomplish:

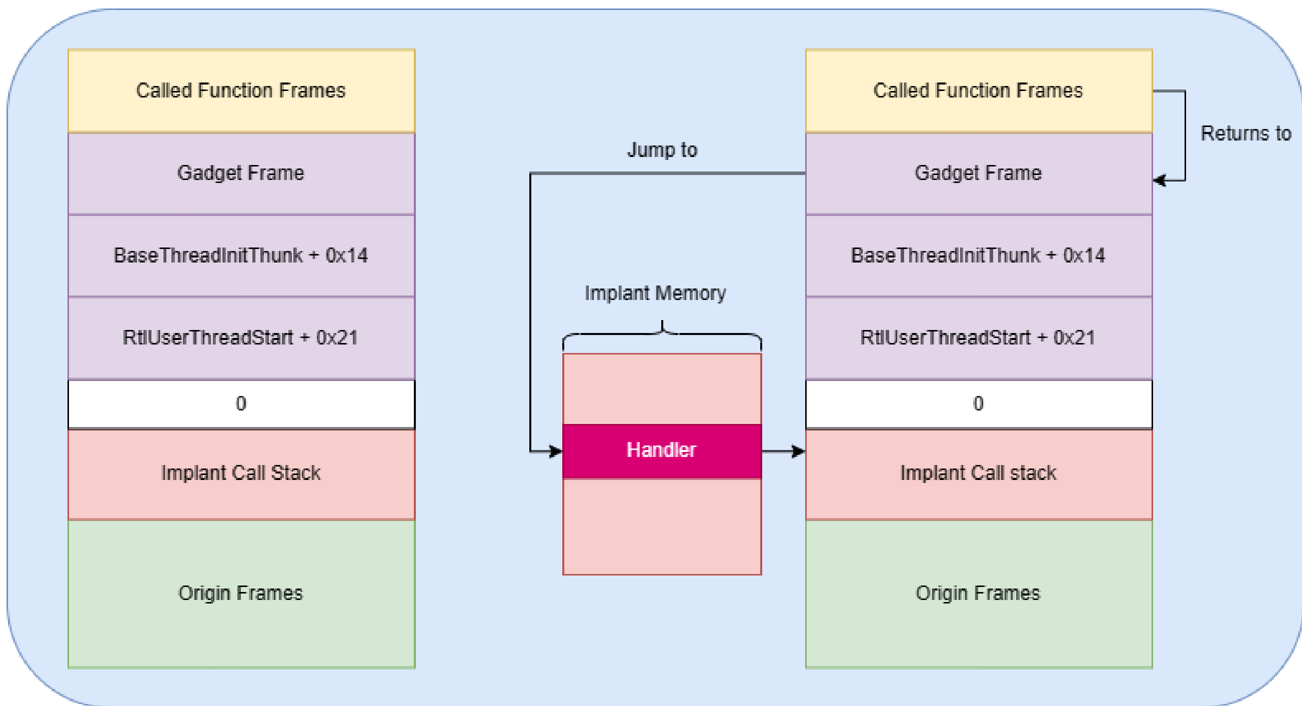
1. Locate the `RUNTIME_FUNCTION` of the frames we wish to craft within their modules’ .pdata section
2. Parse the `UNWIND_CODE`s to calculate their stack sizes
3. Push a 0
4. Create our fake frames

5. Locate a `jmp [rbx]` gadget and create a frame for that
6. Jump to the function we wish to call

The strategy I implemented in my first POC pushes a 0 and creates two fake "origin" frames; these were `ntdll.RtlUserThreadStart + 0x21` and `kernel32.BaseThreadInitThunk+0x14` as they were the most common start frames on the system I was using. The initial 0 being pushed will cause stack walks to prematurely end their walks, so only our synthetic frames and the frames generated after our function call are shown.

▶ More details about LoudSunRun here

On the left is the stack we're trying to create, and on the right is the execution flow.



Spoofing `NtAllocateVirtualMemory` causes the stack to appear like below.

Stack - thread 12088

	Name
0	ntoskrnl.exe!KiCheckForKernelApcDelivery+0x2eb
1	ntoskrnl.exe!KeWaitForSingleObject+0x1787
2	ntoskrnl.exe!KeWaitForSingleObject+0x98f
3	ntoskrnl.exe!KeWaitForSingleObject+0x233
4	ntoskrnl.exe!CmUnregisterMachineHiveLoadedNotification+0x1be88
5	ntoskrnl.exe!CmUnregisterMachineHiveLoadedNotification+0x1d928
6	ntoskrnl.exe!FsRtlRegisterFltMgrCalls+0x64ae5
7	ntoskrnl.exe!KeTestAlertThread+0x454
8	ntoskrnl.exe!setjmpex+0x8bbc
9	ntoskrnl.exe!setjmpex+0x144a
10	ntdll.dll!ZwAllocateVirtualMemory+0x12
11	kernel32.dll!SetDefaultCommConfigW+0xed9
12	kernel32.dll!BaseThreadInitThunk+0x14
13	ntdll.dll!RtlUserThreadStart+0x21

JMP RBX gadget

Fake frames added

We can see the initial 0 pushed caused the walk to be stopped at our synthetic `RtlUserThreadStart` frame; our function call no longer traces back to meterpreter.

The code to the butchered version of KaynLdr I used to replace meterpreter's reflective loader is available here. This was so I could integrate this implementation of stack spoofing into meterpreter via IAT hooking. It does not bypass defender lol.

- ▶ More details about the reflective loader here
- ▶ Can I see the relevant code?

Synthetic Frames vs Elastic + Bitdefender

I tested this successfully (so far as I can tell) against Elastic and Bitdefender. Call stack spoofing isn't a silver bullet; if we're doing a sacrificial process + creating a remote thread, there's much more going on than just a suspicious unbacked call site. Something less loud in nature and still popular (I think) is unhooking. Though Elastic does not use userland hooks, BitDefender does, so I tested a POC which performed unhooking with indirect syscalls, one with spoofing, and one without. The video below has more details.

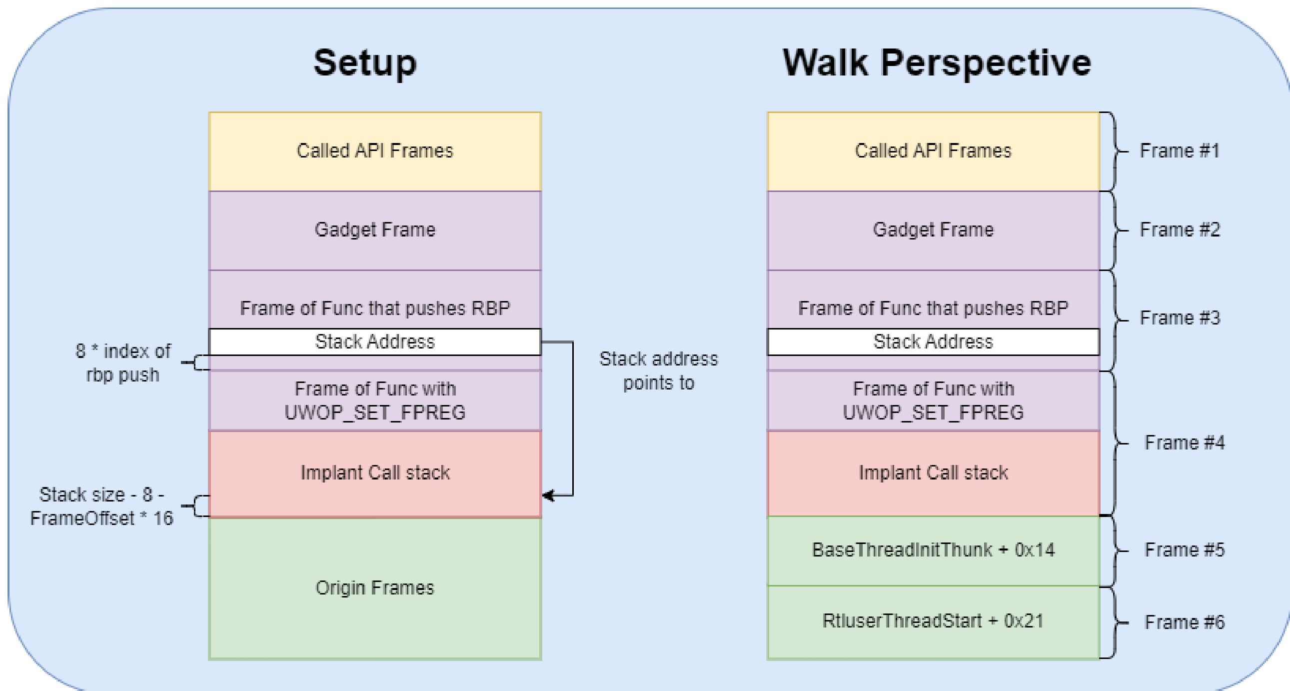
The Cooler Spoof: Moonwalking

With the synthetic approach, the stack never truly unwinds to its base; the zero causes a premature cutoff for the unwinding process. The SilentMoonWalk project has the capability to create frames of dynamic size. Klezvirus called it moonwalking, but I like to call it

“stitching” because of the way the dynamic stack size would “stitch” multiple frames together.

Earlier I stated that stack size was a common attribute used during stack unwinding, but it is not the only one. Interestingly, if a frame that utilizes the `RBP` as a frame pointer is followed by a frame which pushes the `RBP`, then it is possible to create a frame of dynamic size.

Below the frame setup that we’re going for is on the left, and the walk perspective is on the right.



It’s a complicated setup, hence I didn’t integrate it into my meterpreter reflective loader. But it’s really cool because the stack walk actually unwinds to the base of the stack. The `UWOP_SET_FPREG` frame being dynamic in size makes it seem like the implant’s frames get stitched into it frame, hence I like to call it stitching. It still includes the gadget frame so we can regain execution flow in a similar manner to the synthetic frame approach.

Let’s actually break down the necessary components to perform this, though.

- A frame for a function that uses the RBP as a frame pointer (Frame #1)
  This can be determined via the operation code within the `UNWIND_CODE` of the `RUNTIME_FUNCTION` of the frame’s function. If this code is 3 (`UWOP_SET_FPREG`), then this function utilizes the `RBP` as a frame pointer.
- A frame for a function that pushes the RBP (Frame #2)
  This can be determined via the operation code being an `UWOP_PUSH_NONVOL`, with the operation info in the `UNWIND_CODE` being 5 (`RBP_OP_INFO`)

- The index of the RBP being pushed (was it the first register pushed, or the second, etc.)

  We just iterate through the `UNWIND_CODE`s of Frame #2 and count how many `UWOP_PUSH_NONVOL`s there were until we hit one with `RBP_OP_INFO`
- The offset from the origin frame (ie: `BaseThreadInitThunk + 0x14`)

  This offset is calculated via the stack address of `BaseThreadInitThunk + 0x14` - (Stack Size of Frame #1 - (`UNWIND_CODE.FrameOffset` of Frame #1 * 16) - 8)

With this information, we then build our frames like so:

1. Create Frame #1
2. Create Frame #2
3. Insert the stack address at an offset above Frame #1. This stack address is equal to the origin frame address, minus the previously mentioned offset. The stack address placement offset is equal to the index of `RBP` being pushed, times 8.
4. Create gadget frame

While the spoofed call output may seem similar to the synthetic approach, it actually unwinds to the base of the stack rather than the stack being truncated by a 0.

Spoofed via Moonwalk

	Name
0	KernelBase.dll!Sleep
1	kernel32.dll!SetDefaultCommConfigW+0xed9
2	KernelBase.dll+0x11cd
3	ntdll.dll!DbgPrint+0x72
4	kernel32.dll!BaseThreadInitThunk+0x14
5	ntdll.dll!RtlUserThreadStart+0x21

Normal call

	Name
0	KernelBase.dll!Sleep
1	LoudSunRun.exe!LeMoonWalk+0x172
2	LoudSunRun.exe!main+0x28
3	LoudSunRun.exe!invoke_main+0x39
4	LoudSunRun.exe!__scrt_common_main_seh+0x12e
5	LoudSunRun.exe!__scrt_common_main+0xe
6	LoudSunRun.exe!mainCRTStartup+0xe
7	kernel32.dll!BaseThreadInitThunk+0x14
8	ntdll.dll!RtlUserThreadStart+0x21

Note that this was a standalone POC and not implemented into the meterpreter reflective loader, hence all memory addresses are backed by modules. The `LoudSunRun.exe` backed frames would be the implant frames had this been implemented.

Defensive Considerations

The spoofed stack from these techniques can be identified manually by looking weird. For example, in the previously shown spoofed `NtAllocateVirtualMemory` call, `NtAllocateVirtualMemory` usually won't return to `SetDefaultCommConfigW`, if at all.

These implementations relay on a gadget that jumps to a nonvolatile registers to return execution flow to the implant. This is because nonvolatile registers have their values preserved over function calls; they are a safe place to store the address of a fixup handler and/or a struct with information. A frame with a return address to a `jmp nonvolatile` or `jmp [nonvolatile]` register should be seen as suspicious. This seems to be something game cheats have actually picked upon, based on this article by the Secret Club.

Additionally, Klezvirus introduced a novel detection in his Defcon Talk. His tool, Eclipse, checks if the instruction before the return address isn't a `call`; if there isn't then a call has been spoofed. If we check our `jmp [rbx]` gadget, we can see that the instruction before it definitely isn't a call.

Even if the instruction before the return address is a `call`, Eclipse checks to make sure that the `call` is calling the function in the next frame. There is a ton of other interesting and novel techniques in that talk that I can't cover because I don't know enough, so please read it if you can.

Bonus: Function Proxying

This technique isn't stack spoofing but it is intended to address the same defense detection, so I thought it's worth covering. Essentially, some functions can run callbacks and pass arguments to them, all within a separate thread. This causes a target function to be executed with normal looking call stack without the need for spoofing. See the following posts/projects:

While those stacks are completely clean, there's the possiblity of EDR hooking those WinAPIs needed to perform function proxying. And to unhook them it would require executing other WinAPIs/syscalls which could be detected via call stack again. A chicken and egg problem, sort of. There's probably a solution to this, I'm just not there yet lol.

Regarding detection opportunities, proxy calling is in a similar case as spoofing stuff, except rather than having a gadget frame, if the "lower api" call is proxied (e.g: to bypass hooks), then the stack will still be missing the typical frames for that function.

Conclusion

This topic is super cool, and there's more to dig into it. It took a few months, but I think it was worth it. I hope this helped to introduce and explain this (in my opinion) beast of a technique.

Credits and References

A big list of people who helped me or projects/blogs that I referenced during my research.

- KlezVirus - SilentMoonWalk is amazing, and his Defcon talk even more so. Seriously, check the slides and all the Moonwalk talks
- 0xboku - For a ton of help during my internship and guiding me through this topic
- William Burgess - CallStackMasker, VulcanRaven, and this article
- Namaszo - Return address spoofing
- 5pider - Havoc Framework, Kaynldr, ShellcodeTemplate, and for being epic
- Kudaes - Unwinder
- mgeeky - ThreadStackSpoofer
- Paranoid Ninja - Hiding in PlainSight
- rad9800 - Proxy Loading
- Kyle Avery - AceLdr
- realoriginal - TitanLdr
- spotheplanet - Unhooking code

Tags: Pentesting Evasion Windows

- ← Previous Post
- Next Post →