Reflective call stack detections and evasions

October 6, 2023

In a blog published this March, we explored reflective loading through the lens of an offensive security tool developer, highlighting detection and evasion opportunities along the way. This time we are diving into call stack detections and evasions, and how BokuLoader reflectively loads call stack spoofing capabilities into beacon.

We created this blog and public release of BokuLoader during Dylan's summer 2023 internship with IBM X-Force Red. While researching call stack spoofing for our in-house C2, this was one of the techniques identified — the X-Force Adversary Services team's private C2 has much stealthier techniques implemented. The call stack evasion techniques that have been integrated into BokuLoader are not novel and are publicly disclosed. While these techniques are available to security vendors, detections for these evasions may not be present.

In this post, we have created hypothetical detections that could be implemented to detect publicly disclosed active call stack evasion techniques. Each detection has exceptions. Significant tuning would be required before they are deployed into customer environments.

While sharing call stack evasions from the perspective of an offensive operator, our intention is to equip threat hunters and detection engineers with knowledge on how to detect advanced malware from call stack traces.

Call stack detections and evasions

Call stack detection #1 — Return address hunting

When analyzing a call stack, verify that the caller's return address resolves to a loaded module. Unresolved addresses could hint to shellcode and be flagged as malicious. This detection exists in some form within most modern security solutions.

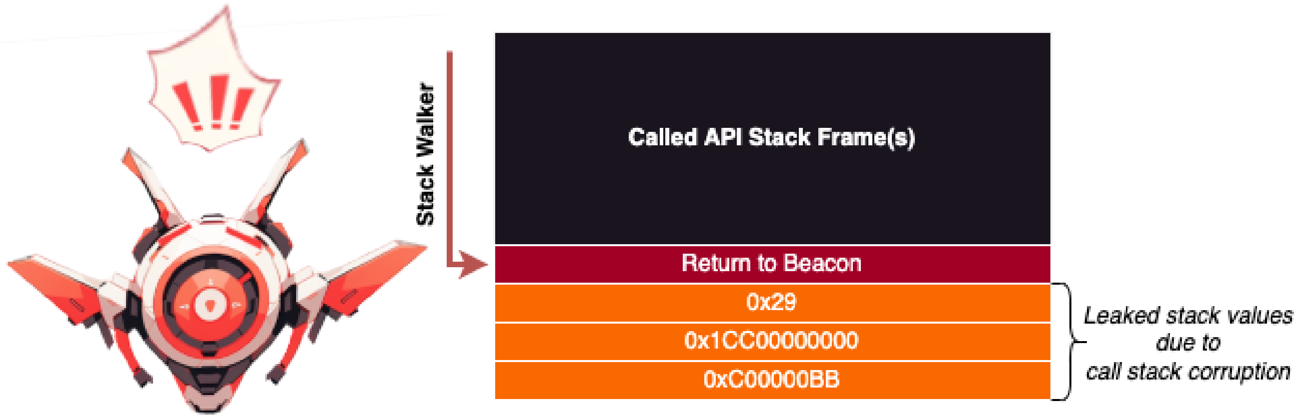


Diagram of detecting shellcode by checking caller’s return address

In this call stack below, a sleeping beacon’s return address is exposed.

Size	Comment	Party
	ntdll.ZwDelayExecution+14	System
A0	kernelbase.SleepEx+9E	User
8	000001CC8DDE362D Beacon	User
8	0000000000000029 Return Address	User
8	000001CC00000000	User
8	00000000C00000BB	User

Default beacon with return address exposed in call stack while executing Sleep API

Evading call stack detection #1 — Return address spoofing

One way to evade this detection is saving beacon’s return address in a non-volatile register before calling an API, then returning to beacon through a ROP gadget in a trusted DLL. Since non-volatile registers are restored by the called API, we can rely on their values persisting through the call.

While checking the return address of the caller, security solutions may be fooled into thinking our stack frame belongs to a trusted DLL API. When our thread finishes executing the called API, it takes a different path than the stack walker. Our thread returns to the gadget, and the gadget jumps our thread back home to beacon.

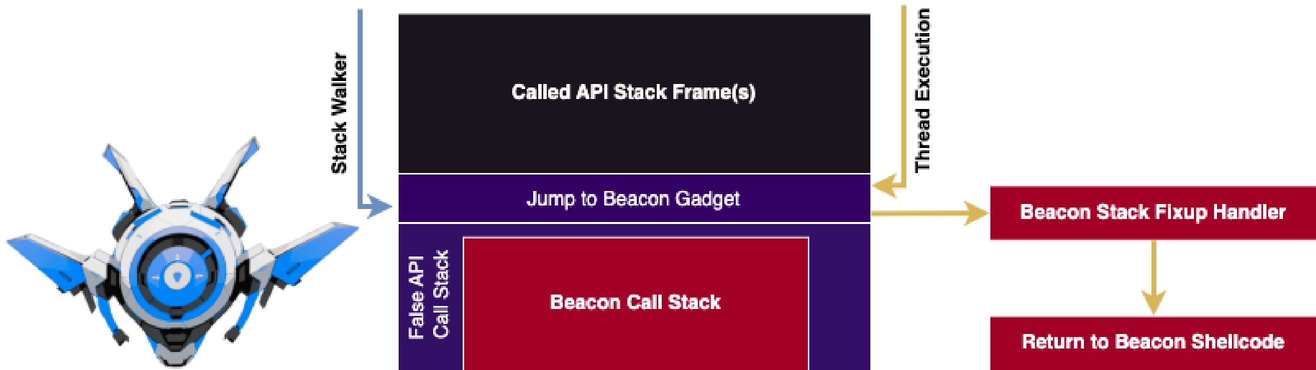
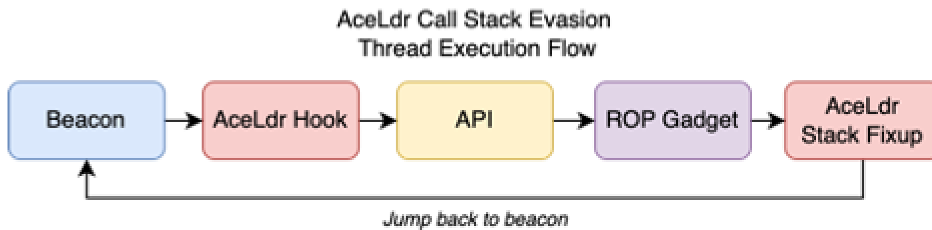


Diagram of evading caller return address detection using return address spoofing

This clever trick was popularized by Namazso in a game hacking forum and then by Kyle Avery in his AceLdr reflective loader project, which he presented at DEFCON 2022.

Like BokuLoader, AceLdr adds call stack spoofing capabilities to beacon by reflectively hooking imported APIs. Before the hook passes execution to the API, beacon's return address is stored on the stack. This stack address is saved into the nonvolatile `RBX` register and a `jmp [rbx]` gadget replaces the caller's return address. To avoid a return address to the hook being pushed onto the stack, the AceLdr hook jumps to the API instead of calling it.



In the call stack below, we see the gadget's return address which resolves to the `WININET` module, instead of a return address to our beacon shellcode.

From	Size	Comment
00007FFBF37773E0	90	wininet.InternetConnectA
00007FFBF37D50A8		wininet.public: unsigned l
00007FFC090C1		
00007FFC09C31		
00007FF6234526B6	40	onenotem.00(

wininet.00007FFBF37D50A8
jmp qword ptr ds:[rbx]
Gadget

AceLdr beacon call stack with return address spoofing of InternetConnectA API

While examining the AceLdr call stack above, we see it ends prematurely after the gadget stack frame. This is because the return address of the next stack frame is 0. Stack walkers typically walk down the call stack until the next stack frame has a return address of 0.

Call stack detection #2 — Truncated call stack

When analyzing a call stack, verify the call stack traces down to common thread start APIs such as `BaseThreadInitThunk` & `RtlUserThreadStart`. Call stack traces that end prematurely are suspicious and could be flagged for further investigation.

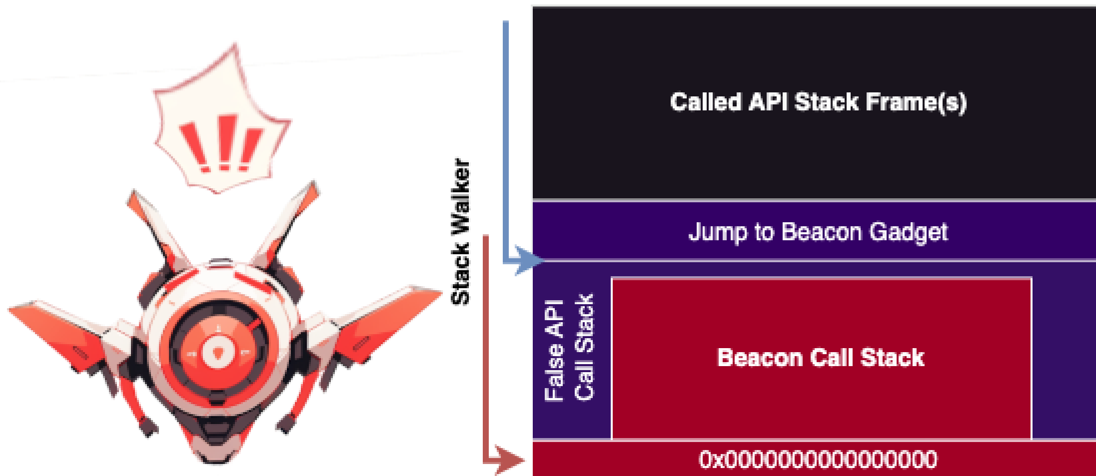


Diagram of a suspicious call stack that uses the call stack truncation evasion.

Evading call stack detection #2 — Synthetic frames

To evade this call stack tracing detection we can extend the first evasion by pushing fake thread start API frames to the stack. As long as we allocate memory on the stack for our fake frames, we can continue to add frames and avoid overwriting beacon's real call stack.

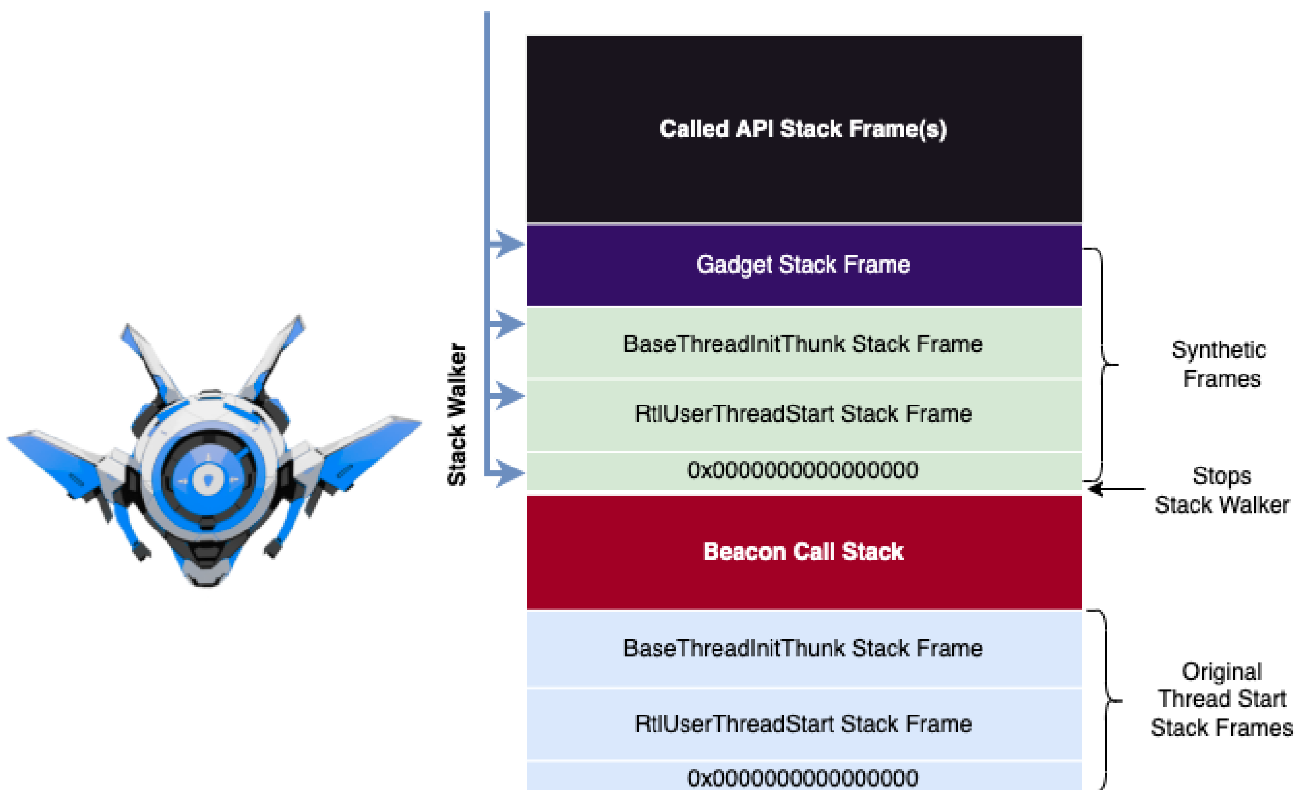
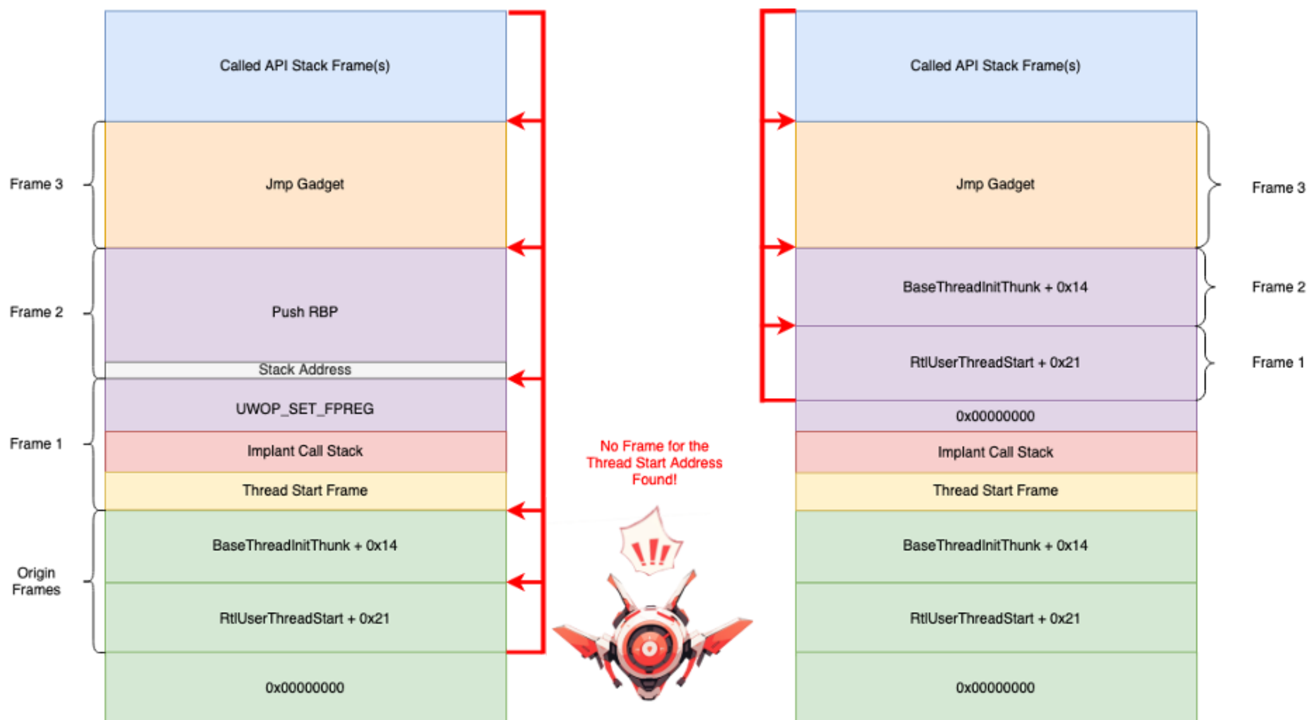


Diagram of evading call stack tracing detections using synthetic frames

Synthetic frames are used in the SilentMoonwalk project as the non-default option and are also used in the Vulcan Raven project.

Call stack detection #3 — Hunting for the thread start frame

When analyzing a call stack, verify that the third stack frame from the bottom is a return address to the thread's start address function. If the call stack does not align with the thread's start function, this may be suspicious and warrant further investigation. This detection technique could be leveraged to detect both modes of the SilentMoonwalk project and the new release of BokuLoader.



Detecting suspicious call stack due to thread start address frame missing (Desynchronization technique — left, Synthetic Frame technique — right)

In the below screenshot we see a thread of `OneDrive.exe` has a start address, which aligns with the third lowest frame in the thread's call stack.

Stack - thread 8628

	Name
0	ntoskrnl.exe!KiCheckForKernelApcDelivery+0x2eb
1	ntoskrnl.exe!KeWaitForSingleObject+0x1787
2	ntoskrnl.exe!KeWaitForSingleObject+0x98f
3	ntoskrnl.exe!IoRetrievePriorityInfo+0x1524
4	ntoskrnl.exe!MmProbeAndLockProcessPages+0x23c0
5	ntoskrnl.exe!setjmpex+0x82b5
6	ntdll.dll!NtWaitForAlertByThreadId+0x14
7	ntdll.dll!RtlSleepConditionVariableSRW+0x131
8	KernelBase.dll!SleepConditionVariableSRW+0x29
9	msvcp140.dll!std::_Winerror_message+0x139
10	msvcp140.dll!Cnd_wait+0x3a
11	FileSyncFALWB.dll!UploadTelemetry+0xd4ad
12	FileSyncFALWB.dll!UploadTelemetry+0xf8f6
13	ucrtbase.dll!configthreadlocale+0x92
14	kernel32.dll!BaseThreadInitThunk+0x14
15	ntdll.dll!RtlUserThreadStart+0x21

Copy

OneDrive.exe (8580) Properties

General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles | GPU | Disk and Netwo

TID	CPU	Cycles delta	Start address	Priority
8732			ucrtbase.dll!configthreadlocale+0x50	Normal
8712			ucrtbase.dll!configthreadlocale+0x50	Normal
8708			ucrtbase.dll!configthreadlocale+0x50	Normal
8704			ucrtbase.dll!configthreadlocale+0x50	Normal
8692			Windows.UI.Xaml.dll!DllCanUnloadNow+0xe0d0	Time critical
8684			ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
8664			ucrtbase.dll!configthreadlocale+0x50	Normal
8656			ucrtbase.dll!configthreadlocale+0x50	Normal
8652			ucrtbase.dll!configthreadlocale+0x50	Lowest
8648			ucrtbase.dll!configthreadlocale+0x50	Normal
8644			ucrtbase.dll!configthreadlocale+0x50	Normal
8640			ucrtbase.dll!configthreadlocale+0x50	Normal
8636			SyncEngine.dll!CreateDirectoryListing+0x28770	Normal
8632			SyncEngine.dll!CreateDirectoryListing+0x28770	Normal
8628			ucrtbase.dll!configthreadlocale+0x50	Normal
8624			ucrtbase.dll!configthreadlocale+0x50	Normal

Start module: C:\Windows\System32\ucrtbase.dll

Started: 2:15:12 PM 7/12/2023

The symbol from a thread's Start Address is normally present on its call stack

This detection will likely result in a lot of false positives if implemented without proper tuning. While typical user programs launched from the desktop follow this pattern, there are many service and background processes that do not. Regardless of the noise, threat hunters and detection engineers may find this detection technique useful while investigating call stack traces.

Call stack detection #4 — Hunting for gadgets

All the evasion techniques we have described depend on a `jmp [rbx]` gadget to restore execution to a handler that returns to beacon. However, it is possible to also use other nonvolatile registers such as `rsi` and `rdi`.

Bokuloader integration

For the public BokuLoader project, we have incorporated the call stack spoofing capabilities of the LoudSunRun project by Dylan Tran. Dylan's project is derived from the SilentMoonwalk and the Vulcan Raven projects. In this release of BokuLoader synthetic stack frames are used to evade call stack detections. The detection methods described in this post should be able to detect a BokuLoader beacon.

Active call stack spoofing has been applied to all `WININET` APIs imported by the Cobalt Strike HTTPS beacon. A hashing method has been included in the BokuLoader project and replaces the previously used string encryption method. This was done to save memory on the stack and avoid potential stack overflows which would crash beacon.

```
LPVOID InternetOpenA_Hook(char* lpszAgent, DWORD dwAccessType, char* lpszProxy, char* lpszProxyBypass, DWORD  dwFlags)
{
    Spoof_Struct spoof_struct = { 0 };
    wininet_apis wininet = { 0 };
    resolve_wininet_apis(&wininet);
    setup_callstack_spoof(&spoof_struct);
    return Spoof(
        lpszAgent,              // Argument # 1
        dwAccessType,           // Argument # 2
        lpszProxy,              // Argument # 3
        lpszProxyBypass,        // Argument # 4
        &spoof_struct,          // Pointer to Spoof Struct
        wininet.InternetOpenA,  // Pointer to API Call
        (void *)1,              // Number of Arguments on Stack (Args 5+)
        dwFlags
    );
```

Hook call stack spoof implementation for InternetOpenA

Some direct syscalls in BokuLoader have been replaced by spoofed indirect syscalls. To determine the syscall number, the HalosGate technique is still used. A simple gadget hunter has been added which searches for a syscall gadget within the executable section of the NTDLL module.

```
    spoof_struct->ssn = getSyscallNumber(api->pNtProtectVirtualMemory);
    base = Spoof(
        NtCurrentProcess(),                     // Argument # 1
        &base,                                  // Argument # 2
        &size,                                  // Argument # 3
        raw_beacon_dll->BeaconMemoryProtection, // Argument # 4
        spoof_struct,                           // Pointer to Spoof Struct
        syscall_gadget,                         // Pointer to API Call
        (void *)1,                              // Number of Arguments on Stack (Args 5+)
        &oldprotect                             // Argument ++
    );
    // HellsGate(getSyscallNumber(api->pNtProtectVirtualMemory));
    // ((tNtProt)HellDescent)(NtCurrentProcess(), &base, &size, raw_beacon_dll->BeaconMemoryProtection, &oldprotect);
}
```

Executing NtProtectVirtualMemory NTAPI via spoofed indirect syscall

Closing thoughts

We hope you found this post helpful and learned something new about call stack detections and evasions. If you are interested in learning more, check out Dylan's recently released blog post "An Introduction into Stack Spoofing." Stay tuned for our next post!

References and resources

- BokuLoader: A proof-of-concept Cobalt Strike Reflective Loader that aims to recreate, integrate, and enhance Cobalt Strike's evasion features — https://github.com/boku7/BokuLoader
- Behind the Mask: Spoofing Call Stacks Dynamically with Timers — https://www.cobaltstrike.com/blog/behind-the-mask-spoofing-call-stacks-dynamically-with-timers/

- AceLdr: Cobalt Strike UDRL for memory scanner evasion — https://github.com/kyleavery/AceLdr
- SilentMoonwalk: PoC Implementation of a fully dynamic call stack spoofer — https://github.com/klezVirus/SilentMoonwalk
- Vulcan Raven: PoC implementation for spoofing arbitrary call stacks when making syscalls — https://github.com/WithSecureLabs/CallStackSpoofer
- CallStackMasker: A PoC implementation for dynamically masking call stacks with timers — https://github.com/Cobalt-Strike/CallStackMasker
- LoudSunRun: Call stack synthetic frame spoofer with indirect syscalls — https://github.com/susMdT/LoudSunRun
- ThreadStackSpoofer: Thread Stack Spoofing PoC — https://github.com/mgeeky/ThreadStackSpoofer

IBM X-Force Research | Offensive security | red team | threat hunting | X-Force

Bobby Cooke
Red Team Operator, Adversary Services, IBM X-Force Red

Dylan Tran
Student and Offensive Security Professional