Spoofing Call Stacks To Confuse EDRs

labs.withsecure.com/blog/spoofing-call-stacks-to-confuse-edrs

Call stacks are an understated yet often important source of telemetry for EDR products. They can provide vital context to an event and be an extremely powerful tool in determining false positives from true positives (especially for credential theft events such as handle access to lsass). An example of this is that attackers will typically reside in-memory via injected code. This unbacked, or floating memory, will show up in call stacks when making API calls and appear highly anomalous.

There has been some public research on spoofing call stacks (most notably https://github.com/mgeeky/ThreadStackSpoofer and https://github.com/Cracked5pider/Ekko), however these seem largely focused on obscuring the call stack for sleeping threads from AV/EDR detection (i.e. for the Cobalt Strike sleep mask).

This contrasts with actively tricking an EDR (or ETW provider) to record a fake call stack from a kernel driver for a specific TTP, say opening a handle to lsass in preparation for dumping credentials. This blog post will demonstrate a PoC technique that will enable NtOpenProcess to be called with an arbitrary call stack (i.e. a genuine call stack spoofer).

Technical Walkthrough

The Windows kernel provides a number of callbacks for AV/EDR drivers to subscribe to in order to receive notifications about system events. For example, this includes process creation/deletion events (PsSetCreateProcessNotifyRoutineEx), thread creation/deletion events (PsSetCreateThreadNotifyRoutine), and object access (ObRegisterCallbacks) etc.

Many of these callbacks run in the context of the thread that triggered the action. Hence, when a kernel driver's process notify routine is called, it is running in the context of the process that triggered the callback (e.g. via calling CreateProcess) and interprets user mode virtual addresses in the context of that user process. Furthermore, the callback will run inline; the operating system is waiting on the callback to return before it can complete the target action of say creating a process or new thread.

This is demonstrated in the contrived kernel call stack below (obtained from windbg via kernel debugging). This shows a breakpoint set on a custom ObRegisterCallback routine (in this case a process handle operation) which was triggered via Outflank's dumpert tool:

```
1: kd> k
00 ffff9387`368011f0 fffff806`2e0a78cc     exampleAVDriver!ObjectCallback+0x50
01 ffff9387`36801b70 fffff806`2e0a7a3a     nt!ObpCallPreOperationCallbacks+0x10c
02 ffff9387`36801bf0 fffff806`2e015e13     nt!ObpPreInterceptHandleCreate+0xaa
03 ffff9387`36801c60 fffff806`2e086ca9     nt!ObpCreateHandle+0xce3
04 ffff9387`36801e70 fffff806`2e09a60f     nt!ObOpenObjectByPointer+0x1b9
05 ffff9387`368020f0 fffff806`2e0f27b3     nt!PsOpenProcess+0x3af
06 ffff9387`36802480 fffff806`2de272b5     nt!NtOpenProcess+0x23
07 ffff9387`368024c0 00007ff7`ef821d42     nt!KiSystemServiceCopyEnd+0x25
08 0000000f`f4aff1e8 00007ff7`ef8219b2     Outflank_Dumpert+0x1d42
09 0000000f`f4aff1f0 00007ff7`ef821fb0     Outflank_Dumpert+0x19b2
0a 0000000f`f4aff890 00007ffd`6c317034     Outflank_Dumpert+0x1fb0
0b 0000000f`f4aff8d0 00007ffd`6d862651     KERNEL32!BaseThreadInitThunk+0x14
0c 0000000f`f4aff900 00000000`00000000     ntdll!RtlUserThreadStart+0x21
```

From this callback, an AV/EDR driver can inspect the object access request and take direct action, such as stripping permission bits from the requested handle if required. Similarly, from a process or thread callback perspective, an AV/EDR can inspect the new process/thread and take preventative action such as blocking it from executing based on some kind of detection logic/heuristic (does the thread point to sketchy memory? etc.).

Additionally, to support the argument of just how useful call stack collection can be, the example above clearly demonstrates the use of direct system calls as there is no ntdll listed in the call stack prior to nt!KiSystemServiceCopyEnd.

As a word of caution, the ObjectCallback is not actually guaranteed to run in the context of the thread that triggered the action; it runs in what is called an arbitrary thread context (hence the current context may not be the actual process which triggered the callback). However, it appears you can reliably assume this is the case the majority of the time.

What should be clear from the example above is that one action an AV/EDR can perform inline from a kernel callback is the act of walking the call stack. In fact, this is exactly what SysMon does for process access events (Event id 10: Process access).

In the screenshot below we can see a Process access event generated by SysMon, which shows svchost obtaining a handle to lsass:

Figure 1: An example SysMon event for process access where lsass is the target image.

We can see the event contains a 'CallTrace' field; this shows the user mode call stack and essentially reveals the chain of events within the process that led to the handle request (albeit without full symbol resolution). This particular event was generated minutes after installing SysMon and occurred at a regular frequency afterwards. Given the call stack does not contain any anomalous memory regions it should be clear this is demonstrably a false positive.

If we load up the SysMon driver (SysmonDrv.sys) into IDA we can determine how exactly SysMon collects the call stack. The key function to look for is RtlWalkFrameChain and cross reference from there. SysMonDrv registers a callback (ObjectHandleCallback below) for process handle operations and on each invocation will call RtlWalkFrameChain to collect the user mode call stack (via the StackWalkWrapper function):

```
void __fastcall ObjectHandleCallback(__int64 a1, __int64 a2)
{
  HANDLE CurrentThreadId; // rax
  struct _KPROCESS *v4; // rcx
  HANDLE ProcessId; // rax
  int *v6; // rcx
  _QWORD *v7; // rax
  _QWORD *v8; // rbx
  _QWORD *v9; // rdx
  char *v10; // rcx
  __int64 v11; // rax
  __int128 v12; // xmm0
  int v13; // eax
  _QWORD *v14; // rcx
  char v15[16]; // [rsp+30h] [rbp-108h] BYREF
  HANDLE CurrentProcessId; // [rsp+40h] [rbp-F8h]
  HANDLE v17; // [rsp+48h] [rbp-F0h]
  HANDLE v18; // [rsp+50h] [rbp-E8h]
  int v19; // [rsp+58h] [rbp-E0h]
  __int64 v20; // [rsp+60h] [rbp-D8h]
  char v21[192]; // [rsp+68h] [rbp-D0h] BYREF
  int v22; // [rsp+128h] [rbp-10h]
  union _LARGE_INTEGER Timeout; // [rsp+150h] [rbp+18h] BYREF

  if ( byte_1800234FA )
  {
    if ( byte_180023550 )
    {
      if ( (*(_DWORD *)(a2 + 4) & 1) == 0 && PsProcessType == *(POBJECT_TYPE **)(a2 + 16) && *(_DWORD *)a2 == 1 )
      {
        CurrentProcessId = PsGetCurrentProcessId();
        CurrentThreadId = PsGetCurrentThreadId();
        v4 = *(struct _KPROCESS **)(a2 + 8);
        v17 = CurrentThreadId;
        ProcessId = PsGetProcessId(v4);
        v6 = *(int **)(a2 + 40);
        v18 = ProcessId;
        v19 = *v6;
        if ( ProcessId != CurrentProcessId )
        {
          v22 = StackWalkWrapper(v21, 24i64);
          v20 = MEMORY[0xFFFFF78000000014];
          if ( !byte_1800234F8 )
          {
            sub_180009BE0(v15);
            return;
          }
```

```
; ULONG __fastcall StackWalkWrapper(PVOID *, ULONG)
StackWalkWrapper proc near
push    rbx
sub     rsp, 20h
mov     r8d, 1          ; Flags
mov     ebx, edx
call    cs:RtlWalkFrameChain
cmp     eax, ebx
cmova   eax, ebx
add     rsp, 20h
pop     rbx
retn
StackWalkWrapper endp
```

Figure 2: The decompilation generated by IDA for SysMonDrv's object call back.

Note that SysMon uses a flag of 1 ('mov r8d, 1') in the call to RtlWalkFrameChain to indicate it wants to collect a user mode call stack only.

RtlWalkFrameChain is exported by ntoskrnl and (at a very high level) works as follows:

- Calls RtlCaptureContext to capture a ContextRecord / CONTEXT structure of the current thread
- Calls RtlpxVirtualUnwind which will take the CONTEXT structure and start unwinding the stack (based on the current state of execution recorded in the CONTEXT structure e.g. via Rip/Rsp etc.)

An example of the implementation of RtlVirtualUnwind can be found here: https://github.com/hzqst/unicorn_pe/blob/master/unicorn_pe/except.cpp#L773 and here: https://doxygen.reactos.org/d8/d2f/unwind_8c.html#a03c91b6c437066272ebc2c2fff051a4c.

Furthermore, ETW can also be configured to collect a stack trace (see: https://github.com/microsoft/krabsetw/pull/191). This again can be very useful for determining anomalous activity for a number of providers (For example, when applied to the Microsoft TI feed or looking for unbacked wininet calls). As a note, ETW collects the call stack in a slightly different way to the typical inline kernel callback approach demonstrated above. It queues an APC to the target thread first and then calls RtlWalkFrameChain. This is probably due to the fact that some ETW providers execute in an arbitrary thread context.

A quick look at the implementation of RtlVirtualUnwind reveals the (rather complicated looking) parsing of X64 unwind codes. Therefore, in order to understand how the call stack is walked via RtlVirtualUnwind, it is first necessary to understand a little bit about how code generation/execution works on X64. A full overview is beyond the scope of this blog post, but this excellent CodeMachine blog post contains everything that is needed to fully understand the techniques used in this blog post: https://codemachine.com/articles/x64_deep_dive.html.

As a brief recap, the CPU has no concept of a function, rather it is a higher level language abstraction. In x86, functions are implemented at the CPU level via using the frame pointer register (Ebp), which can be used as a reference to access local variables and arguments passed via the stack. Through following this chain of Ebp pointers (or function frames) it is possible to find the next stack frame up and hence walk the x86 stack.

In X64, things are more complicated as Rbp is no longer used as a frame pointer and so, alas, the method used above will not work. The key difference to understand is that X64 executables contain a new section called ".pdata". This section is essentially a database containing every function in the executable and instructions (known as UNWIND_CODEs) as to how to "unwind" a given function in the event of an exception. "Unwind" here essentially means to reverse any operation it performed in its function prologue which modified the stack in some way (e.g. made space for local variables, pushed any non-volatile registers to the stack etc.). On X64, once a function has finished its prologue (and hence stack modifications), it does not modify the stack pointer until its epilogue reverses them, hence Rsp is static throughout the function body.

Some typical UNWIND_CODEs are:

- ALLOC_SMALL/LARGE (allocates a small/large amount of memory for local args e.g. sub rsp, 80h)
- PUSH_NONVOL (pushes a non-volatile register to the stack e.g. push rdi)

In windbg, the '.fnent' command will parse this information for a specified function and display its unwind info, as demonstrated for kernelbase!OpenProcess below:

```
0:000> .fnent kernelbase!OpenProcess
Debugger function entry 000001e2`92241720 for:
(00007ff8`7a3bc0f0)   KERNELBASE!OpenProcess   |  (00007ff8`7a3bc170)
KERNELBASE!SetWaitableTimer
Exact matches:
BeginAddress      = 00000000`0002c0f0
EndAddress        = 00000000`0002c160
UnwindInfoAddress = 00000000`00266838

Unwind info at 00007ff8`7a5f6838, 6 bytes
  version 1, flags 0, prolog 7, codes 1
  00: offs 7, unwind op 2, op info c    UWOP_ALLOC_SMALL.
```

This shows that OpenProcess only has one unwind code, which is that it allocates a small area of memory on the stack. The total size of 'UWOP_ALLOC_SMALL' is calculated by multiplying the op info value by 8 and adding 8 (0xc * 8 + 8 =0x68). This can be confirmed by disassembling the first few bytes of kernelbase!OpenProcess (sub rsp, 68h):

```
0:000> uf kernelbase!OpenProcess
KERNELBASE!OpenProcess:
00007ff8`7a3bc0f0 4c8bdc          mov     r11,rsp
00007ff8`7a3bc0f3 4883ec68        sub     rsp,68h
```

A documented list of all the available UNWIND_CODES (and how to parse them) can be found here: https://docs.microsoft.com/en-us/cpp/build/exception-handling-x64?view=msvc-170

In order to walk the stack, windbg will calculate the total stack size for each function found by adding up the size of:

- any local variables
- any stack based parameters
- the return address (8 bytes)
- homing space
- stack space taken up by non-volatile registers

If we take a call to OpenProcess as an example:

```
0:000> knf
 #   Memory  Child-SP          RetAddr               Call Site
00           000000df`7d8fef88 00007ffd`b1bdc13e     ntdll!NtOpenProcess
01         8 000000df`7d8fef90 00007ff7`f10c087d     KERNELBASE!OpenProcess+0x4e
02        70 000000df`7d8ff000 00007ff7`f10c24b9     VulcanRaven!main+0x5d
[C:\Users\wb\source\repos\VulcanRaven\VulcanRaven\VulcanRaven.cpp @ 641]
03       9e0 000000df`7d8ff9e0 00007ff7`f10c239e     VulcanRaven!invoke_main+0x39
[d:\a01\_work\43\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl @ 79]
04        50 000000df`7d8ffa30 00007ff7`f10c225e
VulcanRaven!__scrt_common_main_seh+0x12e
[d:\a01\_work\43\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl @ 288]
05        70 000000df`7d8ffaa0 00007ff7`f10c254e
VulcanRaven!__scrt_common_main+0xe
[d:\a01\_work\43\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl @ 331]
06        30 000000df`7d8ffad0 00007ffd`b2237034     VulcanRaven!mainCRTStartup+0xe
[d:\a01\_work\43\s\src\vctools\crt\vcstartup\src\startup\exe_main.cpp @ 17]
07        30 000000df`7d8ffb00 00007ffd`b3e82651
KERNEL32!BaseThreadInitThunk+0x14
08        30 000000df`7d8ffb30 00000000`00000000     ntdll!RtlUserThreadStart+0x21
```

The top entry, ntdll!NtOpenProcess (#00), is the current stack frame. The Child-SP value of 000000df`7d8fef88 is the value of Rsp after NtOpenProcess has finished its function prologue (i.e. the value of the stack pointer after any stack modifications NtOpenProcess needs to make have been performed). The value of 8 in the 'Memory' column in the row below the current frame is the total stack size used by NtOpenProcess. Therefore, in order to calculate the Child-SP of the next frame, we can add the total stack size of the current frame (8) to the current Child-SP:

```
0:000> ? 000000df`7d8fef88 + 8
Evaluate expression: 959884291984 = 000000df`7d8fef90
```

Note that NtOpenProcess has no unwind op codes (it doesn't modify the stack) so the next Child-SP is simply found by skipping the return address pushed by the previous caller (KERNELBASE!OpenProcess). This is why its total stack size is listed as 8 bytes (e.g. the return address only).

This new Child-SP (000000df`7d8fef90) is the value of Rsp after KERNELBASE!OpenProcess has finished its function prologue. When KERNELBASE!OpenProcess calls ntdll!NtOpenProcess it will push its ret address on to the stack; therefore this return address will be the next value on the stack immediately after where its Child-SP points, as shown by Child-SP 01 in Figure 3.

This process can be repeated again for the next frame. Kernelbase!OpenProcess has a Child-SP of 000000df`7d8fef90 and its total stack utilisation is 0x70 bytes. Once again if we add them together we can get the next Child-SP for VulcanRaven!main:

```
0:000> ? 000000df`7d8fef90 + 70
Evaluate expression: 959884292096 = 000000df`7d8ff000
```

This process is repeated until the debugger has completely walked the stack. Therefore, at a high level the stack walking process looks like this:

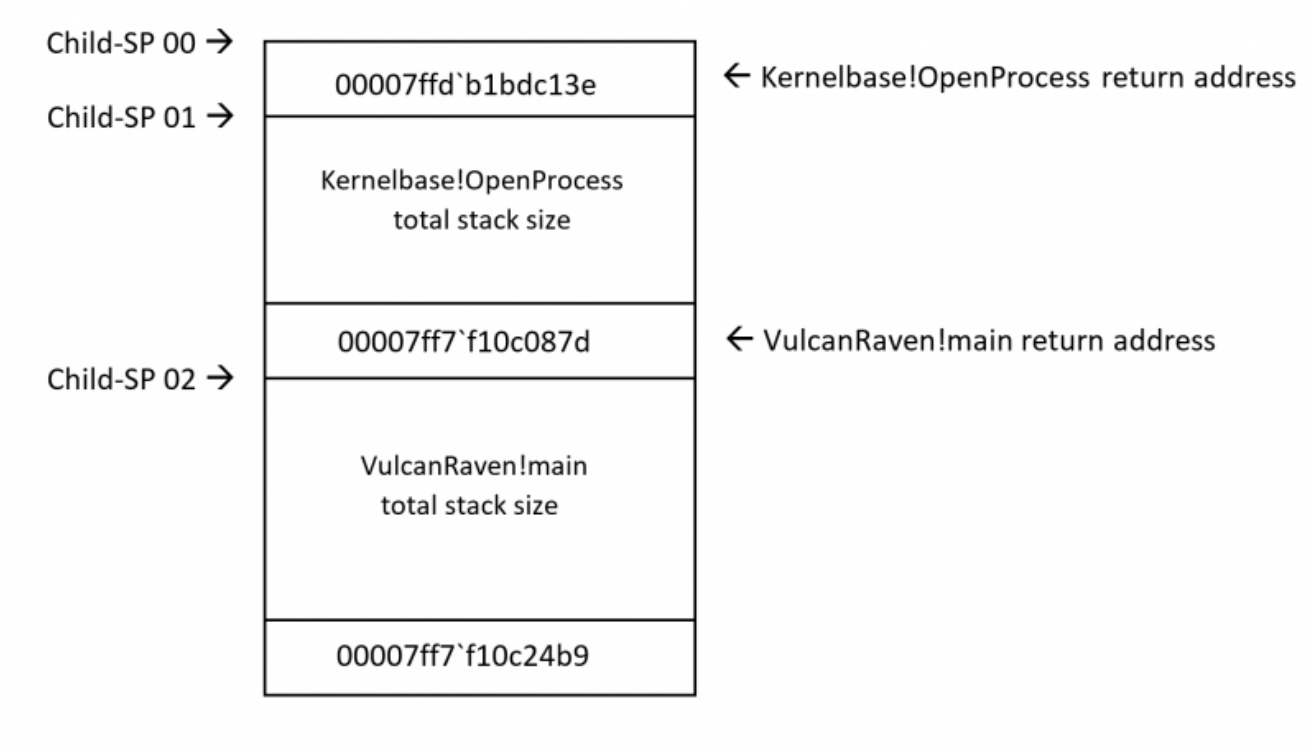

Figure 3: A diagram showing the stack walking process for X64.

The key take away relevant to this blog post is that by knowing the total stack size of a function, it is possible (without symbols) to follow this chain of child stack pointers and walk a call stack. This process will be mimicked in reverse when it comes to spoofing call stacks.

Having discussed the usefulness of call stack telemetry and given a brief overview of how unwinding call stacks works on x64, we can now pivot to the question this blog post addresses: is it possible to spoof a call stack so that when this collection takes place inline (say from within a kernel driver callback routine) a fake call stack is recorded?

Approach

The PoC in this blog post takes the following approach:

1. Identify a target call stack to spoof. For this SysMon was used and an arbitrary entry for event type 10 that opened a handle to lsass was chosen e.g. the example below:

```
CallTrace:
 C:\Windows\SYSTEM32\ntdll.dll + 9d204 (ntdll!NtOpenProcess)
 C:\Windows\System32\KERNELBASE.dll + 32ea6  (KERNELBASE!OpenProcess)
 C:\Windows\System32\lsm.dll + e959
 C:\Windows\System32\RPCRT4.dll + 79633
 C:\Windows\System32\RPCRT4.dll + 13711
 C:\Windows\System32\RPCRT4.dll + dd77b
 C:\Windows\System32\RPCRT4.dll + 5d2ac
 C:\Windows\System32\RPCRT4.dll + 5a408
 C:\Windows\System32\RPCRT4.dll + 3a266
 C:\Windows\System32\RPCRT4.dll + 39bb8
 C:\Windows\System32\RPCRT4.dll + 48a0f
 C:\Windows\System32\RPCRT4.dll + 47e18
 C:\Windows\System32\RPCRT4.dll + 47401
 C:\Windows\System32\RPCRT4.dll + 46e6e
 C:\Windows\System32\RPCRT4.dll + 4b542
 C:\Windows\SYSTEM32\ntdll.dll + 20330
 C:\Windows\SYSTEM32\ntdll.dll + 52f26
 C:\Windows\System32\KERNEL32.DLL + 17034
 C:\Windows\SYSTEM32\ntdll.dll + 52651
```

2. For each return address in the target call stack above, parse its unwind codes and calculate the total stack space required so we can locate the next childSP frame.

3. Create a suspended thread and modify the CONTEXT structure so that the stack/rsp fits the **exact** outline of the target call stack to spoof (without any of the actual data being there). Hence we are initialising the thread's state to "fit the profile" of another thread by pushing fake return addresses and subtracting correct child-SP offsets (e.g. stack unwinding in reverse). Care needs to be taken when handling certain unwind codes (UWOP_SET_FPREG) as this will reset rsp == rbp.

4. Modify the CONTEXT structure so that Rip points to our target function (ntdll!NtOpenProcess in this case) and set the appropriate arguments required by the x64 calling convention (e.g. via setting Rcx/Rdx/R8/R9).

5. Resume the thread and handle the inevitable error once the sys call returns (as it is returning up a fake call stack) via a vectored exception handler. From this exception handler, we can re-direct the thread to RtlExitUserThread (via re-setting Rip) and let it gracefully exit.

For 5), this approach is an obvious limitation; a better approach would be to use VEH exception handling with either hardware or software breakpoints in a similar manner to this patchless AMSI bypass: https://gist.github.com/CCob/fe3b63d80890fafeca982f76c8a3efdf.

With this approach, we could place a breakpoint on the ret immediately after the NtOpenProcess sys call (00007ff8`7ca6d204 below) has returned:

```
0:000> uf ntdll!NtOpenProcess
ntdll!NtOpenProcess:
00007ff8`7ca6d1f0 4c8bd1          mov     r10,rcx
00007ff8`7ca6d1f3 b826000000      mov     eax,26h
00007ff8`7ca6d1f8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308
(00000000`7ffe0308)],1
00007ff8`7ca6d200 7503            jne     ntdll!NtOpenProcess+0x15
(00007ff8`7ca6d205)  Branch
ntdll!NtOpenProcess+0x12:
00007ff8`7ca6d202 0f05            syscall
00007ff8`7ca6d204 c3              ret
```

Once a breakpoint exception is generated (and before the thread returns and crashes) we could handle the error in the same way as discussed previously. Additionally, recovering the state of the fake thread and being able to re-use it would be an improvement and stop the need to repeatedly create “sacrificial threads”.

Furthermore, this approach could also potentially be applied to the sleeping obfuscation problem; a fake thread with a legitimate call stack could be initialised to call ntdll!NtDelayExecution (or WaitForSingleObject etc.) and VEH exceptions used to re-direct flow to a main beacon function on return of the sleep duration.

PoC || GTFO

The PoC code can be found here: https://github.com/countercept/CallStackSpoofer

The PoC comes with three example call stacks (wmi/rpc/svchost) to mimic, all of which were taken arbitrarily from SysMon logs via observing process handle access to lsass. These call stack profiles can be selected via the '–wmi', '--rpc', and '–svchost' flags, as demonstrated below:

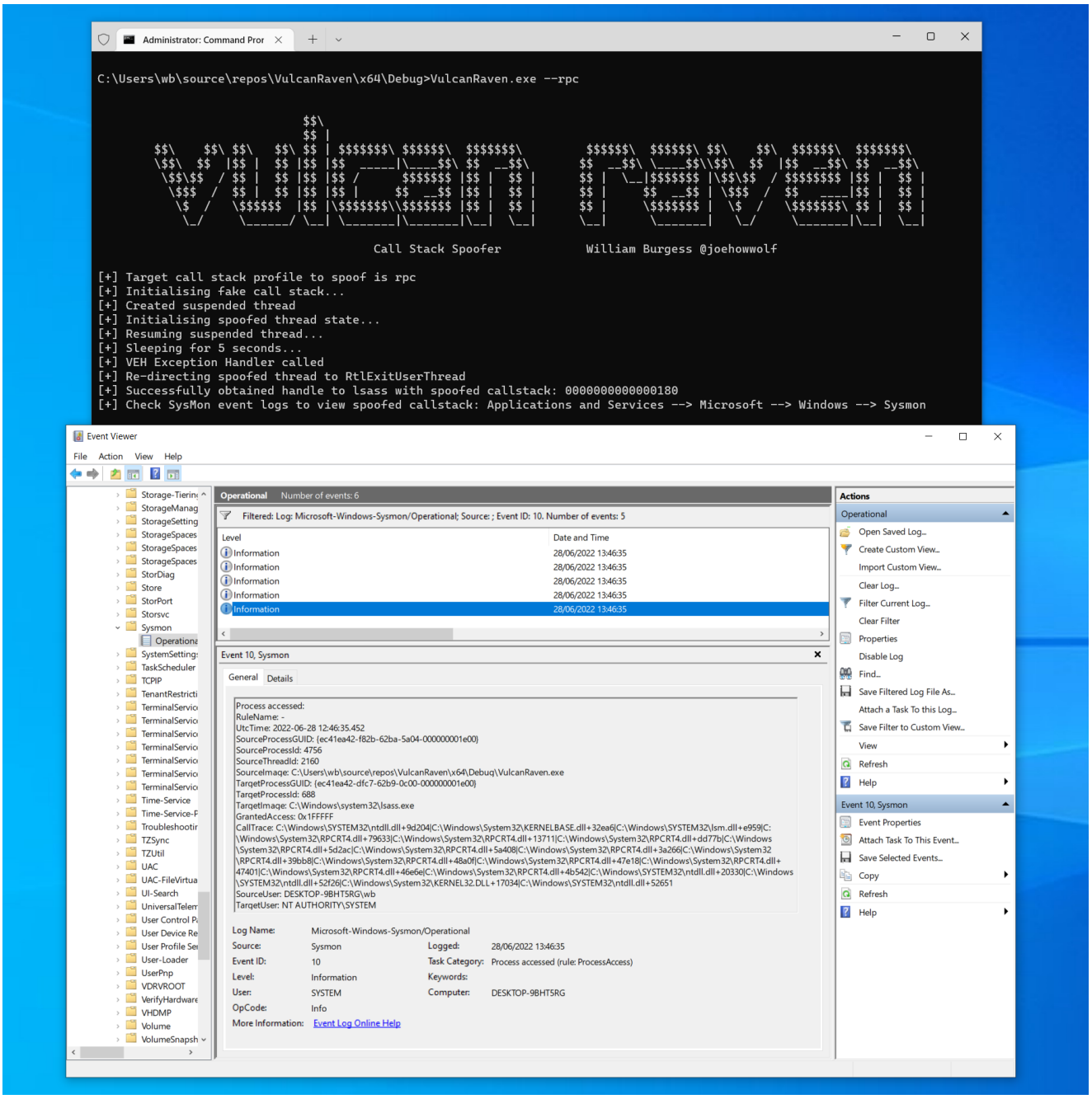

Figure 4: A screenshot showing VulcanRaven grabbing a handle to lsass and spoofing the call stack to look like RPC activity.

The screenshot above demonstrates a fake call stack being recorded by SysMon (contrast this to the expected use of OpenProcess which would have a call stack of: VulcanRaven.exe -> kernelbase!OpenProcess -> ntdll!NtOpenProcess). Just to stress, the examples in this PoC were chosen to mimic events found via SysMon but the call stack does not have to make sense and can be anything, as demonstrated below:

```
0:000> eb 00007ffd`24b2d202 0xcc
0:000> g
(1980.19f4): Break instruction exception - code 80000003 (first chance)
ntdll!NtOpenProcess+0x12:
00007ffd`24b2d202 cc              int     3
0:003> knf
 #   Memory  Child-SP          RetAddr               Call Site
00           000000a2`51cfeef0 00007ffd`221e96de     ntdll!NtOpenProcess+0x12
01         8 000000a2`51cfeef8 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
02        a0 000000a2`51cfef98 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
03        a0 000000a2`51cff038 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
04        a0 000000a2`51cff0d8 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
05        a0 000000a2`51cff178 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
06        a0 000000a2`51cff218 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
07        a0 000000a2`51cff2b8 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
08        a0 000000a2`51cff358 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
09        a0 000000a2`51cff3f8 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
0a        a0 000000a2`51cff498 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
0b        a0 000000a2`51cff538 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
0c        a0 000000a2`51cff5d8 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
0d        a0 000000a2`51cff678 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
0e        a0 000000a2`51cff718 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
0f        a0 000000a2`51cff7b8 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
10        a0 000000a2`51cff858 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
11        a0 000000a2`51cff8f8 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
12        a0 000000a2`51cff998 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
13        a0 000000a2`51cffa38 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
14        a0 000000a2`51cffad8 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
15        a0 000000a2`51cffb78 00007ffd`221e96de     KERNELBASE!SleepEx+0x9e
16        a0 000000a2`51cffc18 00000000`00000000     KERNELBASE!SleepEx+0x9e
```

Figure 5: A screenshot from WinDbg showing a completely nonsensical call stack being spoofed when calling NtOpenProcess.

The most obvious example of why this technique would be of interest to an attacker is that most remote access trojans (beacon etc.) still tend to operate off floating / unbacked memory. Thus, when an attacker injects mimikatz directly into memory, the call stack for the subsequent handle access from this injected code will look highly anomalous.

As an example, a SysMon event is shown below for unbacked memory calling OpenProcess:

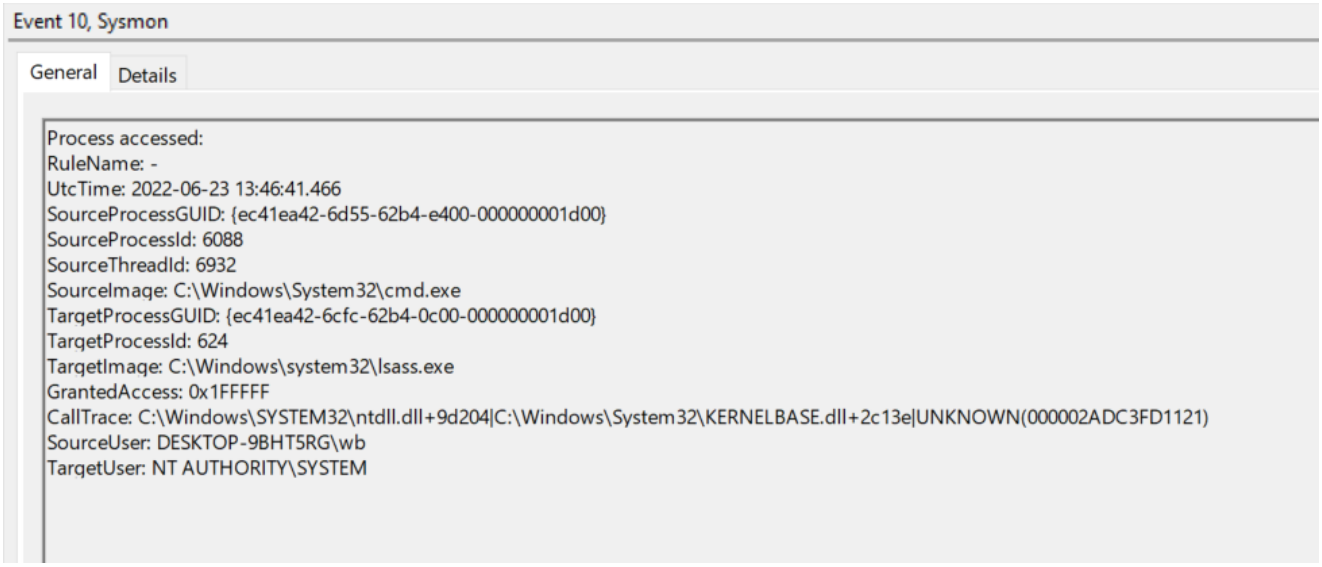

Figure 6: A SysMon event showing handle access to lsass originating from unbacked memory.

This was generated via using a modified version of Stephen Fewer's ReflectiveDLLInjection https://github.com/stephenfewer/ReflectiveDLLInjection codebase.

In this example, a reflective DLL has been injected into cmd.exe and subsequently obtained a handle to lsass with PROCESS_ALL_ACCESS access. As the call originated from unbacked memory, SysMon records the last entry in the call stack as "UNKNOWN" (e.g. the last return address in the stack walk belonged to floating/unbacked code and not a legitimately loaded module) and hence is obviously suspicious.

However, if we modify the VulcanRaven PoC above to run as a reflective DLL, we generate the following event:

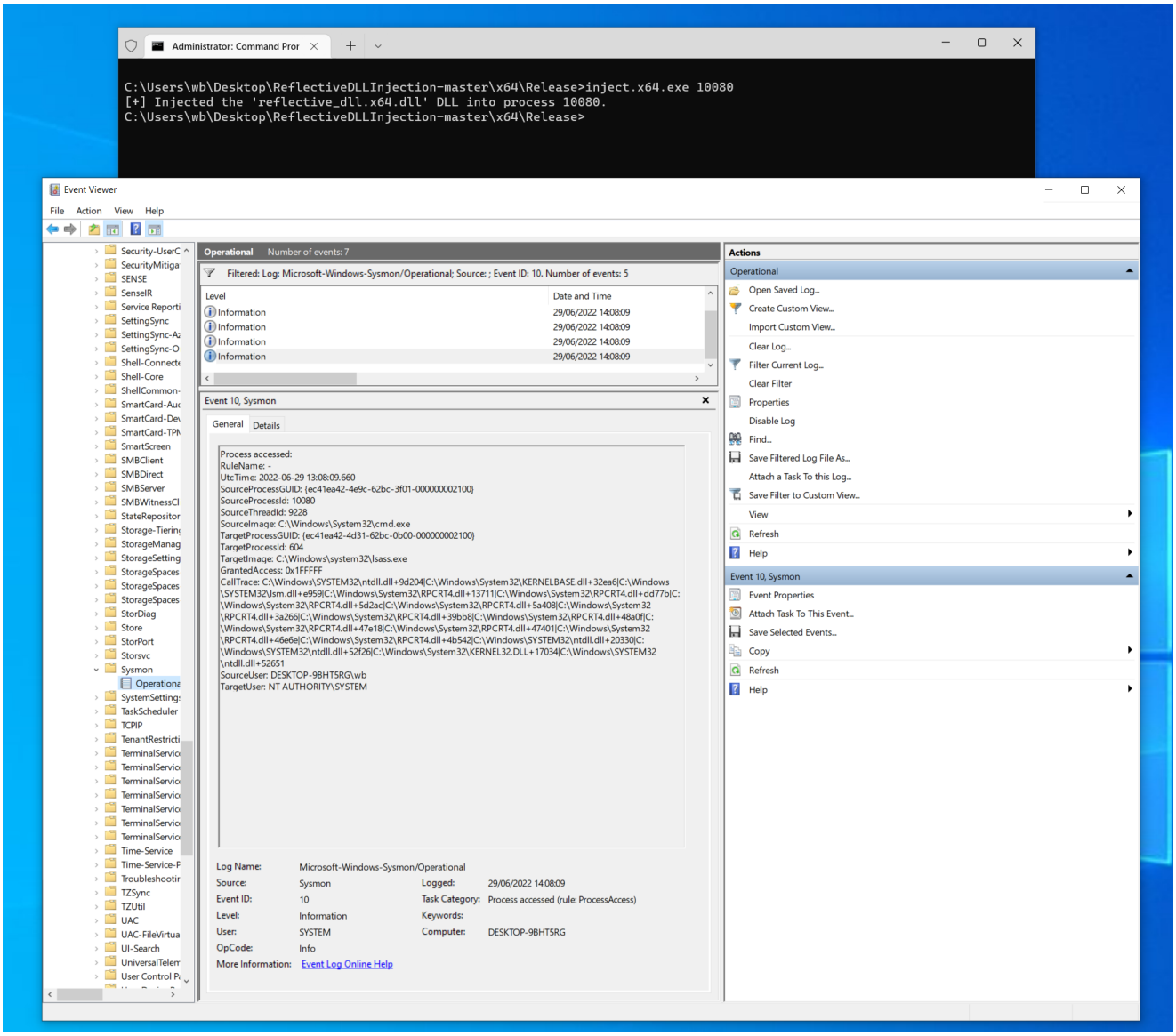

Figure 7: A screenshot showing Vulcan Raven modified to run as reflective DLL. Even though it is running from unbacked memory, the call stack for handle access to lsass is still spoofed to look legimate.

The call stack ("CallTrace") is spoofed to an arbitrary value pulled from SysMon as expected; it is impossible to tell from this call stack that the call to NtOpenProcess/OpenProcess actually originated from code running from unbacked memory, and the thread on the surface looks genuine (although the cmd.exe is contrived and obviously suspicious). Also note the different GrantedAccess to Figure 1, which in this case is PROCESS_ALL_ACCESS/0x1FFFFF.

It is clear though that an attacker could profile their chosen call stack to blend in with their chosen injected process (wmi, procexp, svchost etc. all regularly grab handles to lsass).