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/106e3dd5d9c49326d55ce17c919bffe68ead1467/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
        
        "), UserBuffer);
        DbgPrint("[+] UserBuffer Size: 0x%X
        
        "), Size);
        DbgPrint("[+] KernelBuffer: 0x%p
        
        "), &KernelBuffer);
        DbgPrint("[+] KernelBuffer Size: 0x%X
        
        "), 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%\n        
        ", Status);
    }

    return Status;
}

TriggerStackOverflow is called via StackOverflowIoctlHandler
(https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/106e3dd5d9c49326d55ce17c919bffe68ead1467/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
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:

```
k> k
# ChildEBP RetAddr
 00 8c812d0c 8292f1ce 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+0x4dc
 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 8c813a44 41414141 0x41414141
 08 8c813aa8 41414141 0x41414141
 09 8c813aa8 41414141 0x41414141
 10 8c813ab8 41414141 0x41414141
 11 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.
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.

We now know that the return address is stored \textbf{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 \texttt{StackOverflowIoctlHandler} at offset \texttt{0x1e}.

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

\texttt{9176b6718 pop ebp}
\texttt{9176b6719 ret 8}

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

\texttt{xor eax, eax ; Set NTSTATUS SUCCESSE
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).

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 StackOverflow_ioctl::Handler before returning.

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

- Abatchy

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