hasherezade's 1001 nights

projects and tasks that I do in my free time

Starting with Windows Kernel Exploitation – part 2 – getting familiar with HackSys Extreme Vulnerable Driver

Posted on June 5, 2017

Recently I started learning Windows Kernel Exploitation, so I decided to share some of my notes in form of a blog.

The previous part was about setting up the lab. Now, we will play a bit with HackSysExtremeVulnerableDriver by Ashfaq Ansari in order to get comfortable with it. In the next parts I am planning to walk through the demonstrated vulnerabilities and exploitation techniques.

What I use for this part:

- The lab described in the previous part
- HackSys Extreme Vulnerable Driver (HEVD) – prebuild version + the source code
- OSR Driver Loader
- DebugView (from SysInternals Suite)
- Visual Studio 2012 (you can use any version you like)

Installing and testing HEVD

First, I will show how to install HEVD. We will and configure Debugee and the Debugger in order to see the Debug Strings and HEVD's symbols. We will also play a bit with dedicated exploits. You can see the video and read the explanations below:

Watching the DebugStrings

HEVD and the dedicated exploits prints a lot of information as DebugStrings. We can watch them from the Debugger machine (using WinDbg) as well as from Debugee machine (using DebugView).

Before installing HEVD, we will set up everything in order to see the strings that are being printed during driver's initialization.

On the Debugger:

We need to break the execution of the Debugee in order to get the *kd* prompt (in WinDbg: Debug -> Break). Then, we enable printing Debug Strings via command:

```
ed nt!Kd_Default_Mask 8
```

After that, we can let the Debugee run further by executing the command:

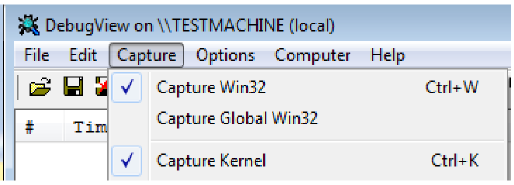
```
g
```

Warning: Enabling this slows down the Debugee. So, whenever possible, try to watch DebugStrings locally (on the Debugee only).

On the Debugee:

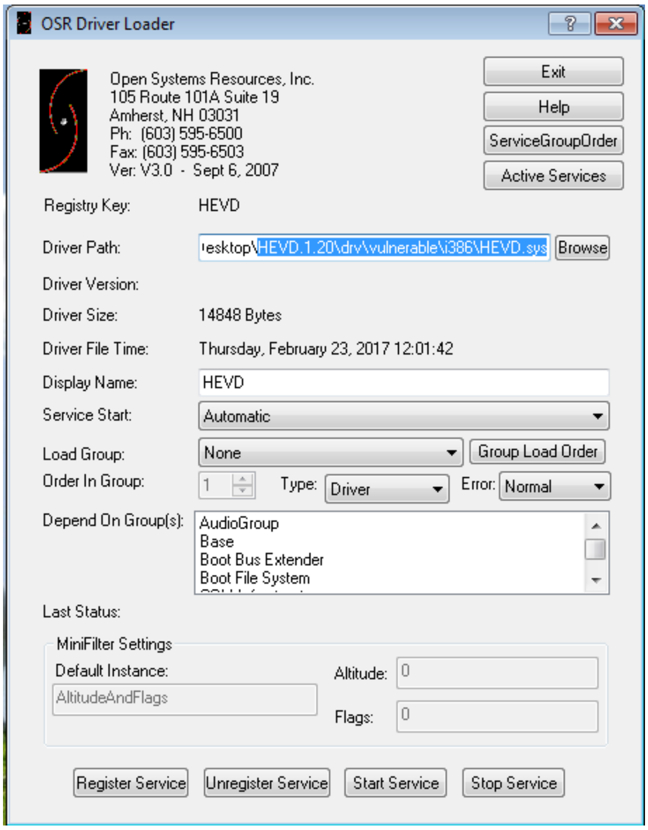
We need to run DebugView as Administrator. Then we choose from the menu:

Capture -> Capture Kernel



Installing the driver

First, we will download the pre-build package (driver+exploit) on the Debugee (the victim machine), install them and test. We can find it on the github of HackSysTeam, in section releases (https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/releases). The package contains two version of driver – vulnerable and not. We will pick the vulnerable one, built for 32 bit (*i386*).



Open Systems Resources, Inc.
105 Route 101A Suite 19
Amherst, NH 03031
Ph: (603) 595-6500
Fax: (603) 595-6503
Ver: V3.0 - Sept 6, 2007

Registry Key: HEVD

Driver Path: esktop\HEVD.1.20\drv\vulnerable\i386\HEVD.sys

Driver Version:

Driver Size: 14848 Bytes

Driver File Time: Thursday, February 23, 2017 12:01:42

Display Name: HEVD

Service Start: Automatic

Load Group: None

Order In Group: 1 Type: Driver Error: Normal

Depend On Group(s): AudioGroup
Base
Boot Bus Extender
Boot File System

Last Status:

MiniFilter Settings

Default Instance: AltitudeAndFlags

Altitude: 0

Flags: 0

We choose *Service Start* as *Automatic*. Then we click: *[Register Service]* and when it succeeded: *[Start Service]*.

We should see the HEVD banner printed on WinDbg (on the Debugger machine) as well as on DbgView on Debugee Machine.

Adding symbols

The precompiled package of HEVD comes with symbols (*sdb* file) that we can also add to our Debugger. First, let's stop the Debugee by sending it a break signal, and have a look at all the loaded modules.

```
lm
```

To find the HEVD module, we can set a filter:

```
lm m H*
```

We will see, that it does not have any symbols attached. Well, it can be easily fixed. First, turn on:

```
!sym_noisy
```

– in order to print all the information about the paths to which WinDbg referred in search for the symbol. Then, try to reload the symbols:

```
.reload
```

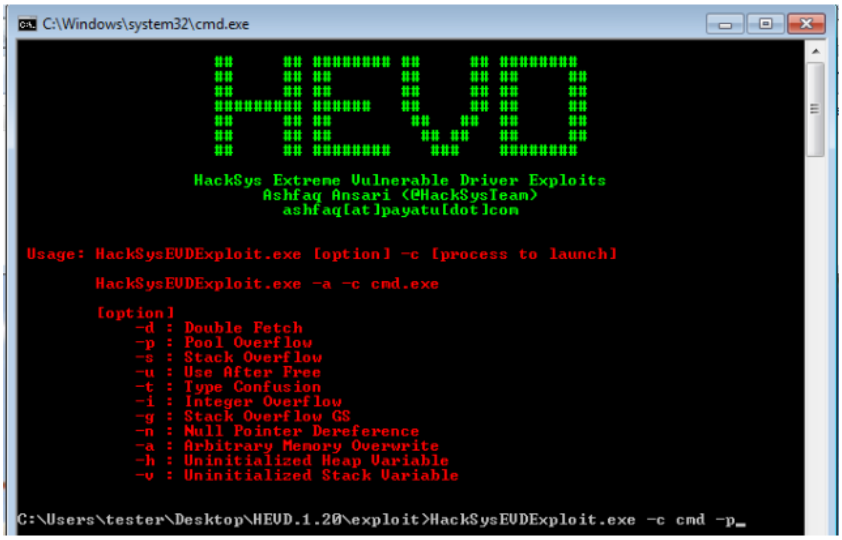
…and try to refer to them again. You will see the path, where we can copy the *pdb* file. After moving the *pdb* file to the appropriate location on the Debugger machine, reload the symbols again. You can test them by trying to print all the functions from HEVD:

```
x HEVD!*
```

(See the details on the Video#1)

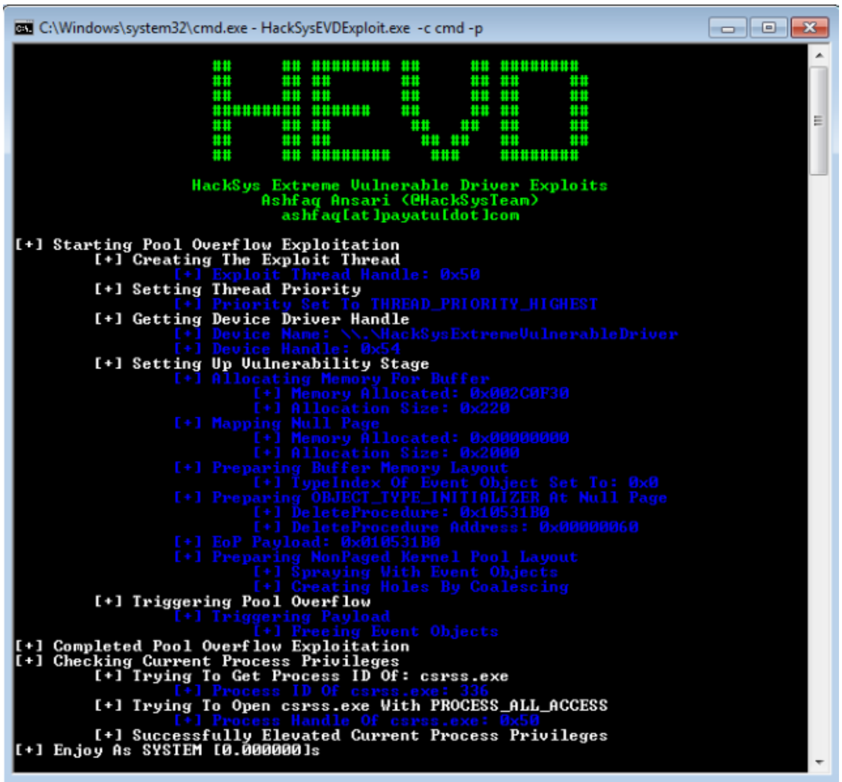
Testing the exploits

The same package contains also a set of the dedicated exploits. We can run each of them by executing an appropriate command. Let's try to deploy some of them and set *cmd.exe* as a program to be executed.



```
                    ##      ## ######### ##      ## #########
                    ##      ## ##        ##      ## ##      ##
                    ##      ## ##        ##      ## ##      ##
                    ########## #######   ##      ## ##      ##
                    ##      ## ##         ##    ##  ##      ##
                    ##      ## ##          ## ##    ##      ##
                    ##      ## #########     ###    #########

                  HackSys Extreme Vulnerable Driver Exploits
                         Ashfaq Ansari (@HackSysTeam)
                           ashfaq[at]payatu[dot]com

 Usage: HackSysEVDExploit.exe [option] -c [process to launch]

        HackSysEVDExploit.exe -a -c cmd.exe

        [option]
            -d : Double Fetch
            -p : Pool Overflow
            -s : Stack Overflow
            -u : Use After Free
            -t : Type Confusion
            -i : Integer Overflow
            -g : Stack Overflow GS
            -n : Null Pointer Dereference
            -a : Arbitrary Memory Overwrite
            -h : Uninitialized Heap Variable
            -v : Uninitialized Stack Variable

C:\Users\tester\Desktop\HEVD.1.20\exploit>HackSysEVDExploit.exe -c cmd -p_
```

Pool Overflow Exploit deployed:



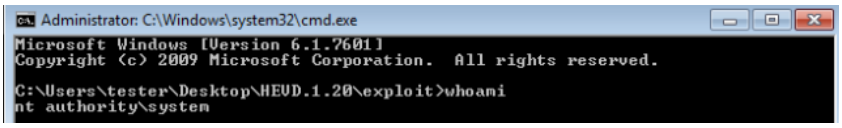
```
                    ##      ## ######### ##      ## #########
                    ##      ## ##        ##      ## ##      ##
                    ##      ## ##        ##      ## ##      ##
                    ########## #######   ##      ## ##      ##
                    ##      ## ##         ##    ##  ##      ##
                    ##      ## ##          ## ##    ##      ##
                    ##      ## #########     ###    #########

                  HackSys Extreme Vulnerable Driver Exploits
                         Ashfaq Ansari (@HackSysTeam)
                           ashfaq[at]payatu[dot]com

[+] Starting Pool Overflow Exploitation
        [+] Creating The Exploit Thread
                [+] Exploit Thread Handle: 0x50
        [+] Setting Thread Priority
                [+] Priority Set To THREAD_PRIORITY_HIGHEST
        [+] Getting Device Driver Handle
                [+] Device Name: \\.\HackSysExtremeVulnerableDriver
                [+] Device Handle: 0x54
        [+] Setting Up Vulnerability Stage
                [+] Allocating Memory For Buffer
                        [+] Memory Allocated: 0x002C0F30
                        [+] Allocation Size: 0x220
                [+] Mapping Null Page
                        [+] Memory Allocated: 0x00000000
                        [+] Allocation Size: 0x2000
                [+] Preparing Buffer Memory Layout
                        [+] TypeIndex Of Event Object Set To: 0x0
                [+] Preparing OBJECT_TYPE_INITIALIZER At Null Page
                        [+] DeleteProcedure: 0x10531B0
                        [+] DeleteProcedure Address: 0x00000060
                [+] EoP Payload: 0x010531B0
                [+] Preparing NonPaged Kernel Pool Layout
                        [+] Spraying With Event Objects
                        [+] Creating Holes By Coalescing
        [+] Triggering Pool Overflow
                [+] Triggering Payload
                        [+] Freeing Event Objects
[+] Completed Pool Overflow Exploitation
[+] Checking Current Process Privileges
        [+] Trying To Get Process ID Of: csrss.exe
                [+] Process ID Of csrss.exe: 336
        [+] Trying To Open csrss.exe With PROCESS_ALL_ACCESS
                [+] Process Handle Of csrss.exe: 0x50
        [+] Successfully Elevated Current Process Privileges
[+] Enjoy As SYSTEM [0.000000]s
```

If the exploitation went successful, the requested application (cmd.exe) will be deployed with elevated privileges.

By the command

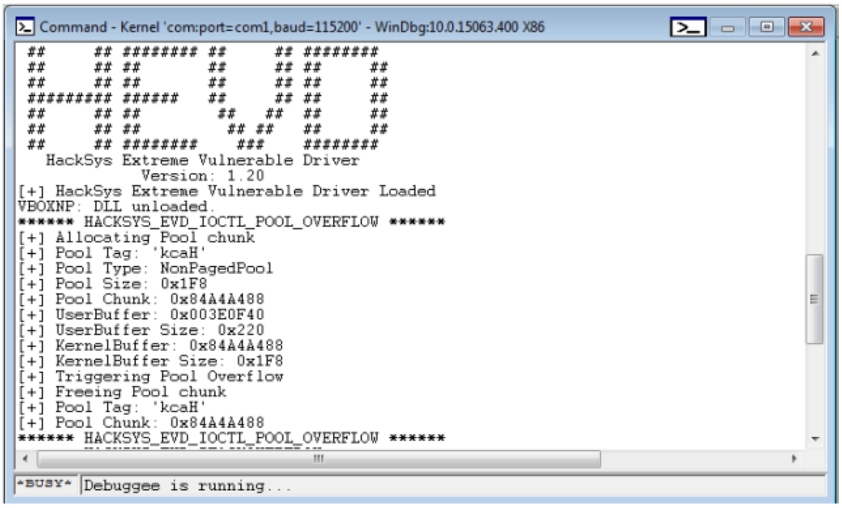
```
whoami
```

we can confirm, that it is really run elevated:



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

C:\Users\tester\Desktop\HEVD.1.20\exploit>whoami
nt authority\system
```

At the same time, we can see on our Debugger machine the Debug Strings printed by the exploit:



```
 ##     ## ######## ##     ## ########
 ##     ## ##       ##     ## ##     ##
 ##     ## ##       ##     ## ##     ##
 ######### ######   ##     ## ##     ##
 ##     ## ##        ##   ##  ##     ##
 ##     ## ##         ## ##   ##     ##
 ##     ## ########    ###    ########
   HackSys Extreme Vulnerable Driver
             Version: 1.20
[+] HackSys Extreme Vulnerable Driver Loaded
VBOXNP: DLL unloaded.
****** HACKSYS_EVD_IOCTL_POOL_OVERFLOW ******
[+] Allocating Pool chunk
[+] Pool Tag: 'kcaH'
[+] Pool Type: NonPagedPool
[+] Pool Size: 0x1F8
[+] Pool Chunk: 0x84A4A488
[+] UserBuffer: 0x003E0F40
[+] UserBuffer Size: 0x220
[+] KernelBuffer: 0x84A4A488
[+] KernelBuffer Size: 0x1F8
[+] Triggering Pool Overflow
[+] Freeing Pool chunk
[+] Pool Tag: 'kcaH'
[+] Pool Chunk: 0x84A4A488
****** HACKSYS_EVD_IOCTL_POOL_OVERFLOW ******
```

```
*BUSY* Debuggee is running...
```

All of the exploits, except the double fetch should run well on one core. If we want this exploit to work, we need to enable two cores on the Debugee machine.

WARNING: Some of the exploits are not 100% reliable and we can encounter a system crash after deploying them. Don't worry, this is normal.

Hi driver, let's talk!

Just like in case of the user land, in the kernel land exploitation begins from finding the points, where we can supply an input to the program. Then, we need to find the input that can corrupt the execution (in contrary to the user land – in kernel land a crash will directly result in having a blue screen!). Finally, we will be trying to craft the input in a way that let us control the execution of the vulnerable program.

In order to communicate with a driver from user mode we will be sending it IOCTLs – Input-Output controls. The IOCTL allows us to send from the user land some input buffer to the driver. This is the point from which we can attempt the exploitation.

HEVD contains demos of various classes of vulnerabilities. Each of them can be triggered using a different IOCTL and exploited by the supplied buffer. Some (but not all) will cause our system to crash when triggered.

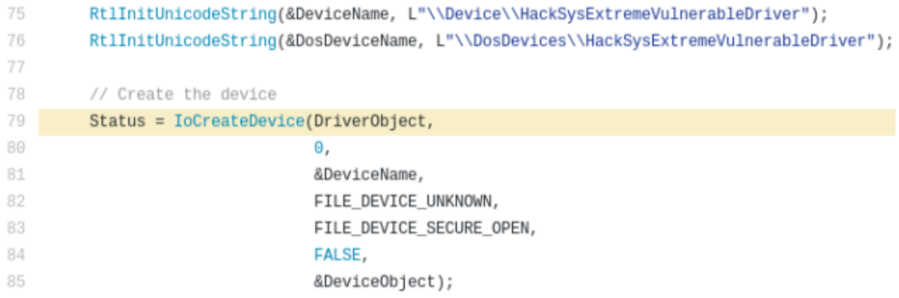
Finding Device name & IOCTLs

Before we try to communicate with a driver, we need to know two things:

1. the device that the driver creates (if it doesn't create any, we will not be able to communicate)
2. list of IOCTLs (Input-Output Controls) that the driver accepts

HEVD is open-source, so we can read all the necessary data directly from the source code. In real life, most of the time we will have to reverse the driver in order to get it.

Let's have a look at the fragment of code where HEVD creates a device.
https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/master/Driver/HackSysExtremeVulnerableDrive

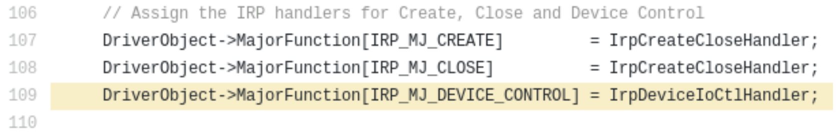


```
    RtlInitUnicodeString(&DeviceName, L"\\Device\\HackSysExtremeVulnerableDriver");
    RtlInitUnicodeString(&DosDeviceName, L"\\DosDevices\\HackSysExtremeVulnerableDriver");

    // Create the device
    Status = IoCreateDevice(DriverObject,
                            0,
                            &DeviceName,
                            FILE_DEVICE_UNKNOWN,
                            FILE_DEVICE_SECURE_OPEN,
                            FALSE,
                            &DeviceObject);
```

The name of the device is mentioned above.

Now, let's see find the list of IOCTLs. We will start from looking at the array of IRPs:

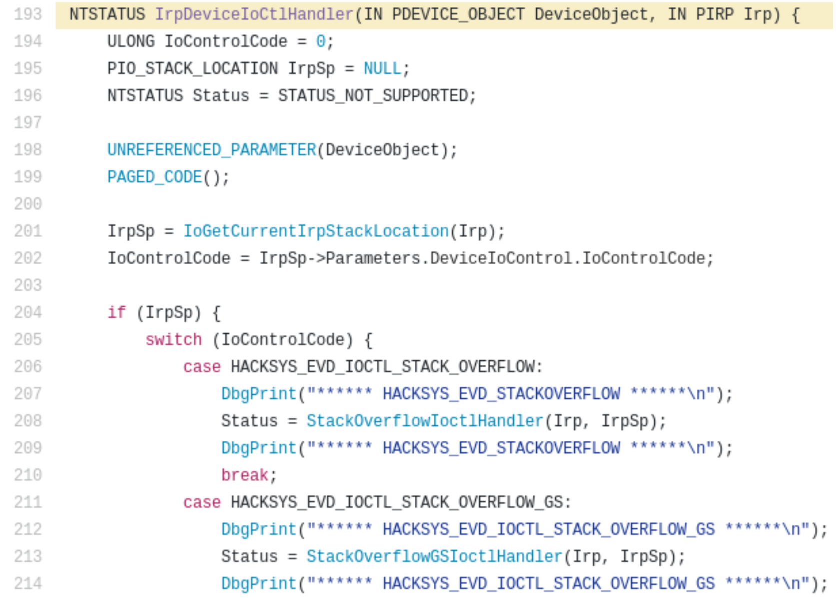
https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/master/Driver/HackSysExtremeVulnerableDrive



```
    // Assign the IRP handlers for Create, Close and Device Control
    DriverObject->MajorFunction[IRP_MJ_CREATE]         = IrpCreateCloseHandler;
    DriverObject->MajorFunction[IRP_MJ_CLOSE]          = IrpCreateCloseHandler;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IrpDeviceIoCtlHandler;

```

The function linked to IRP_MJ_DEVICE_CONTOL will be dispatching IOCTLs sent to the driver. So, we need to take a look inside this function.

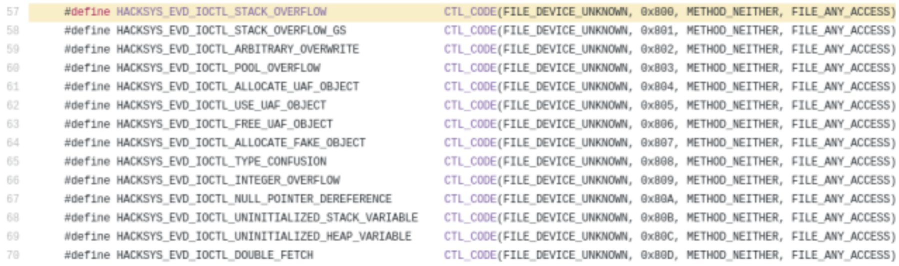
https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/master/Driver/HackSysExtremeVulnerableDrive



```
NTSTATUS IrpDeviceIoCtlHandler(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp) {
    ULONG IoControlCode = 0;
    PIO_STACK_LOCATION IrpSp = NULL;
    NTSTATUS Status = STATUS_NOT_SUPPORTED;

    UNREFERENCED_PARAMETER(DeviceObject);
    PAGED_CODE();

    IrpSp = IoGetCurrentIrpStackLocation(Irp);
    IoControlCode = IrpSp->Parameters.DeviceIoControl.IoControlCode;

    if (IrpSp) {
        switch (IoControlCode) {
            case HACKSYS_EVD_IOCTL_STACK_OVERFLOW:
                DbgPrint("****** HACKSYS_EVD_STACKOVERFLOW ******\n");
                Status = StackOverflowIoctlHandler(Irp, IrpSp);
                DbgPrint("****** HACKSYS_EVD_STACKOVERFLOW ******\n");
                break;
            case HACKSYS_EVD_IOCTL_STACK_OVERFLOW_GS:
                DbgPrint("****** HACKSYS_EVD_IOCTL_STACK_OVERFLOW_GS ******\n");
                Status = StackOverflowGSIoctlHandler(Irp, IrpSp);
                DbgPrint("****** HACKSYS_EVD_IOCTL_STACK_OVERFLOW_GS ******\n");
```

It contains a switch, that calls a handler function appropriate to handle a particular IOCTL. We can grab our list of IOCTLs by coping the switch cases. The values of the constants are defied in a header:

https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/master/Driver/HackSysExtremeVulnerableDrive



```
#define HACKSYS_EVD_IOCTL_STACK_OVERFLOW                 CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_NEITHER, FILE_ANY_ACCESS)
#define HACKSYS_EVD_IOCTL_STACK_OVERFLOW_GS              CTL_CODE(FILE_DEVICE_UNKNOWN, 0x801, METHOD_NEITHER, FILE_ANY_ACCESS)
#define HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE            CTL_CODE(FILE_DEVICE_UNKNOWN, 0x802, METHOD_NEITHER, FILE_ANY_ACCESS)
#define HACKSYS_EVD_IOCTL_POOL_OVERFLOW                  CTL_CODE(FILE_DEVICE_UNKNOWN, 0x803, METHOD_NEITHER, FILE_ANY_ACCESS)
#define HACKSYS_EVD_IOCTL_ALLOCATE_UAF_OBJECT            CTL_CODE(FILE_DEVICE_UNKNOWN, 0x804, METHOD_NEITHER, FILE_ANY_ACCESS)
#define HACKSYS_EVD_IOCTL_USE_UAF_OBJECT                 CTL_CODE(FILE_DEVICE_UNKNOWN, 0x805, METHOD_NEITHER, FILE_ANY_ACCESS)
#define HACKSYS_EVD_IOCTL_FREE_UAF_OBJECT                CTL_CODE(FILE_DEVICE_UNKNOWN, 0x806, METHOD_NEITHER, FILE_ANY_ACCESS)
#define HACKSYS_EVD_IOCTL_ALLOCATE_FAKE_OBJECT           CTL_CODE(FILE_DEVICE_UNKNOWN, 0x807, METHOD_NEITHER, FILE_ANY_ACCESS)
#define HACKSYS_EVD_IOCTL_TYPE_CONFUSION                 CTL_CODE(FILE_DEVICE_UNKNOWN, 0x808, METHOD_NEITHER, FILE_ANY_ACCESS)
#define HACKSYS_EVD_IOCTL_INTEGER_OVERFLOW               CTL_CODE(FILE_DEVICE_UNKNOWN, 0x809, METHOD_NEITHER, FILE_ANY_ACCESS)
#define HACKSYS_EVD_IOCTL_NULL_POINTER_DEREFERENCE       CTL_CODE(FILE_DEVICE_UNKNOWN, 0x80A, METHOD_NEITHER, FILE_ANY_ACCESS)
#define HACKSYS_EVD_IOCTL_UNINITIALIZED_STACK_VARIABLE   CTL_CODE(FILE_DEVICE_UNKNOWN, 0x80B, METHOD_NEITHER, FILE_ANY_ACCESS)
#define HACKSYS_EVD_IOCTL_UNINITIALIZED_HEAP_VARIABLE    CTL_CODE(FILE_DEVICE_UNKNOWN, 0x80C, METHOD_NEITHER, FILE_ANY_ACCESS)
#define HACKSYS_EVD_IOCTL_DOUBLE_FETCH                   CTL_CODE(FILE_DEVICE_UNKNOWN, 0x80D, METHOD_NEITHER, FILE_ANY_ACCESS)
```

Writing a client application

Ok, we got all the necessary data that we can use to communicate with the driver by our own program. We can put it all together in a header file, i.e.: hevd_constants.h

```
#pragma once
#include <windows.h>

const char kDevName[] = "\\\\.\\HackSysExtremeVulnerableDriver";

#define HACKSYS_EVD_IOCTL_STACK_OVERFLOW                  CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_NEITHER, FILE_ANY_ACCE
#define HACKSYS_EVD_IOCTL_STACK_OVERFLOW_GS               CTL_CODE(FILE_DEVICE_UNKNOWN, 0x801, METHOD_NEITHER, FILE_ANY_ACCE
#define HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE             CTL_CODE(FILE_DEVICE_UNKNOWN, 0x802, METHOD_NEITHER, FILE_ANY_ACCE
#define HACKSYS_EVD_IOCTL_POOL_OVERFLOW                   CTL_CODE(FILE_DEVICE_UNKNOWN, 0x803, METHOD_NEITHER, FILE_ANY_ACCE
#define HACKSYS_EVD_IOCTL_ALLOCATE_UAF_OBJECT             CTL_CODE(FILE_DEVICE_UNKNOWN, 0x804, METHOD_NEITHER, FILE_ANY_ACCE
#define HACKSYS_EVD_IOCTL_USE_UAF_OBJECT                  CTL_CODE(FILE_DEVICE_UNKNOWN, 0x805, METHOD_NEITHER, FILE_ANY_ACCE
#define HACKSYS_EVD_IOCTL_FREE_UAF_OBJECT                 CTL_CODE(FILE_DEVICE_UNKNOWN, 0x806, METHOD_NEITHER, FILE_ANY_ACCE
#define HACKSYS_EVD_IOCTL_ALLOCATE_FAKE_OBJECT            CTL_CODE(FILE_DEVICE_UNKNOWN, 0x807, METHOD_NEITHER, FILE_ANY_ACCE
#define HACKSYS_EVD_IOCTL_TYPE_CONFUSION                  CTL_CODE(FILE_DEVICE_UNKNOWN, 0x808, METHOD_NEITHER, FILE_ANY_ACCE
#define HACKSYS_EVD_IOCTL_INTEGER_OVERFLOW                CTL_CODE(FILE_DEVICE_UNKNOWN, 0x809, METHOD_NEITHER, FILE_ANY_ACCE
#define HACKSYS_EVD_IOCTL_NULL_POINTER_DEREFERENCE        CTL_CODE(FILE_DEVICE_UNKNOWN, 0x80A, METHOD_NEITHER, FILE_ANY_ACCE
#define HACKSYS_EVD_IOCTL_UNINITIALIZED_STACK_VARIABLE    CTL_CODE(FILE_DEVICE_UNKNOWN, 0x80B, METHOD_NEITHER, FILE_ANY_ACCE
#define HACKSYS_EVD_IOCTL_UNINITIALIZED_HEAP_VARIABLE     CTL_CODE(FILE_DEVICE_UNKNOWN, 0x80C, METHOD_NEITHER, FILE_ANY_ACCE
#define HACKSYS_EVD_IOCTL_DOUBLE_FETCH                    CTL_CODE(FILE_DEVICE_UNKNOWN, 0x80D, METHOD_NEITHER, FILE_ANY_ACCE
```

Number of each IOCTL is created by a macro defined in a standard windows header winioctl.h:

```
#define CTL_CODE(DeviceType, Function, Method, Access) \
        (((DeviceType) << 16) | ((Access) << 14) |     \
        ((Function) << 2) | (Method))
```

If you include *windows.h* header, the above macro will be added automatically. For now, we not need to bother about meaning of the particular constants – we will just use the defined elements as they are.

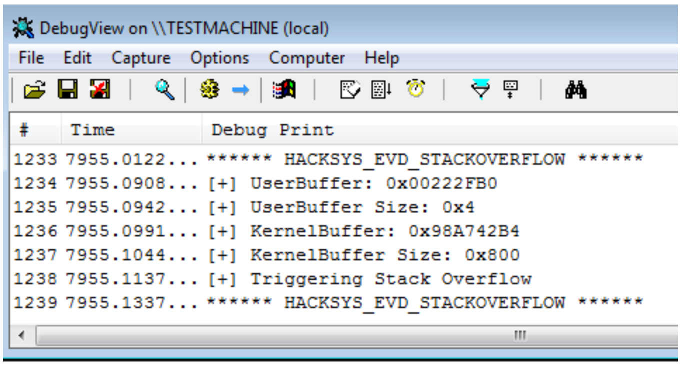
So, we are ready to write a simple user land application that will talk to the driver. First, we open the device using function CreateFile. Then, we can send the IOCTL using DeviceIoControl.

Below you can see a tiny example. This application sends the STACK_OVERFLOW IOCTL to the driver: send_ioctl.cpp

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

#define HACKSYS_EVD_IOCTL_STACK_OVERFLOW    CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_NEITHER, FILE_ANY_ACCESS)

const char kDevName[] = "\\\\.\\HackSysExtremeVulnerableDriver";

HANDLE open_device(const char* device_name)
{
    HANDLE device = CreateFileA(device_name,
        GENERIC_READ | GENERIC_WRITE,
        NULL,
        NULL,
        OPEN_EXISTING,
        NULL,
        NULL
    );
    return device;
}

void close_device(HANDLE device)
{
    CloseHandle(device);
```

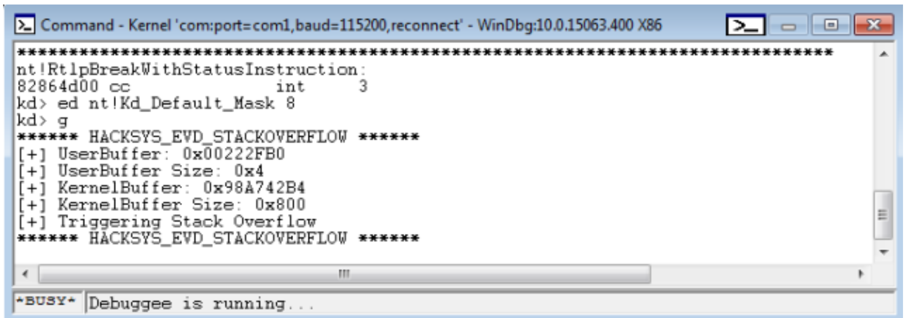
```
}

BOOL send_ioctl(HANDLE device, DWORD ioctl_code)
{
    //prepare input buffer:
    DWORD bufSize = 0x4;
    BYTE* inBuffer = (BYTE*) HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, bufSize);

    //fill the buffer with some content:
    RtlFillMemory(inBuffer, bufSize, 'A');

    DWORD size_returned = 0;
    BOOL is_ok = DeviceIoControl(device,
        ioctl_code,
        inBuffer,
        bufSize,
        NULL, //outBuffer -> None
        0, //outBuffer size -> 0
        &size_returned,
        NULL
    );
    //release the input bufffer:
    HeapFree(GetProcessHeap(), 0, (LPVOID)inBuffer);
    return is_ok;
}

int main()
{
    HANDLE dev = open_device(kDevName);
    if (dev == INVALID_HANDLE_VALUE) {
        printf("Failed!\n");
        system("pause");
        return -1;
    }

    send_ioctl(dev, HACKSYS_EVD_IOCTL_STACK_OVERFLOW);

    close_device(dev);
    system("pause");
    return 0;
}
```

Try to compile this program and deploy it on the Debugee machine. Start the DebugView and observe DebugStrings printed by the driver.



```
1233 7955.0122... ****** HACKSYS_EVD_STACKOVERFLOW ******
1234 7955.0908... [+] UserBuffer: 0x00222FB0
1235 7955.0942... [+] UserBuffer Size: 0x4
1236 7955.0991... [+] KernelBuffer: 0x98A742B4
1237 7955.1044... [+] KernelBuffer Size: 0x800
1238 7955.1137... [+] Triggering Stack Overflow
1239 7955.1337... ****** HACKSYS_EVD_STACKOVERFLOW ******
```

If you enabled printing DebugStrings on the Debugger machine, you should see similar output:



```
*******************************************************************************
nt!RtlpBreakWithStatusInstruction:
82864d00 cc              int     3
kd> ed nt!Kd_Default_Mask 8
kd> g
****** HACKSYS_EVD_STACKOVERFLOW ******
[+] UserBuffer: 0x00222FB0
[+] UserBuffer Size: 0x4
[+] KernelBuffer: 0x98A742B4
[+] KernelBuffer Size: 0x800
[+] Triggering Stack Overflow
****** HACKSYS_EVD_STACKOVERFLOW ******
```

```
*BUSY* Debuggee is running...
```

As we can see, the driver got our input and reported about it.

Exercise: let's have a crash!

As an exercise, I created a small client for HEVD, that allows to send it various IOCTLs with the input buffer of the requested length. You can find the source code here:

https://github.com/hasherezade/wke_exercises/tree/master/task1

..and the compiled 32 bit binary here.

Try to play with various IOCTLs, till you get the crash. Because the Debugee runs under the control of the Debugger, you should not get a blue screen – instead, WinDbg will get triggered. Try to make a brief crash analysis for every case. Start from printing the information by:

```
!analyze -v
```

Some other helpful commands:

```
k - stack trace
kb - stack trace with parameters
r - registers
dd [address]- display data as DWORD starting from the address
```

For more, check the WinDbg help file:

```
.hh
```

In our sample application, the user buffer is filled with "A" -> ASCII 0x41 (https://github.com/hasherezade/wke_exercises/blob/master/task1/src/main.cpp#L34):

```
RtlFillMemory(inBuffer, bufSize, 'A');
```

So, wherever we see it in the crash analysis, it means the particular data can be filled by the user.

Example #1

Example #2

Mind the fact, that triggering the same vulnerability can give you a different output, depending on the immediate source of the crash, that is related to i.e. size of the overflow, current layout of the memory, etc.

Part 3: https://hshrzd.wordpress.com/2017/06/22/starting-with-windows-kernel-exploitation-part-3-stealing-the-access-token/

Appendix

- http://expdev-kiuhnm.rhcloud.com/2015/05/17/windbg/ – introduction to WinDbg (by Massimiliano Tomassoli)
- https://github.com/mwrlabs/win_driver_plugin – An IDA Pro plugin to help when working with IOCTL codes or reversing Windows drivers (by Sam Brown)