[hfiref0x/CVE-2015-1701: Win32k LPE vulnerability used in APT attack (github.com)](https://github.com/hfiref0x/CVE-2015-1701)

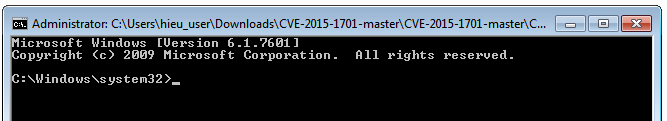
I) Thực thi CVE:

[WINDOWS EXPLOITATION CVE 2015 1701 WINDOWS PRIVILEGES ESCLATION BYPASS - YouTube](https://www.youtube.com/watch?v=zTjgWWrYGJQ&ab_channel=CYBERHUNT)

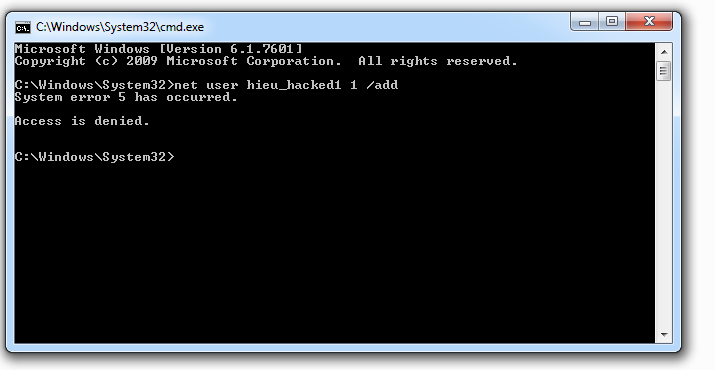
1) Truy cập Window 7 bằng tài khoản User:



2) Khi chạy file Taihou64.exe, ta mở ra được cửa sổ:



Chương trình đã mở được Command Prompt dưới quyền Administrator mặc dù chúng ta đang chạy Windows dưới quyền User, tức là đã leo quyền thành công. Tuy nhiên không phải cứ mở được cmd ở system32 đã là leo quyền thành công, cái này cần phải kiểm tra, ví dụ khi mở cmd ở hẳn file system32 dưới quyền User:



3) Trên máy local window 7 ta thực hiện các lệnh:

+) Tạo một user mới là “hieu\_hacked”, mật khẩu là “1”:

net user hieu\_hacked 1 /add

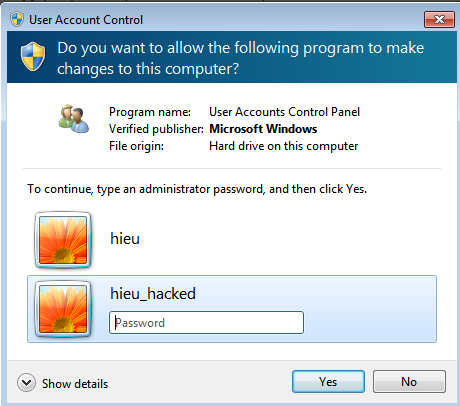


+) Đặt user hieu\_hacked làm admin:

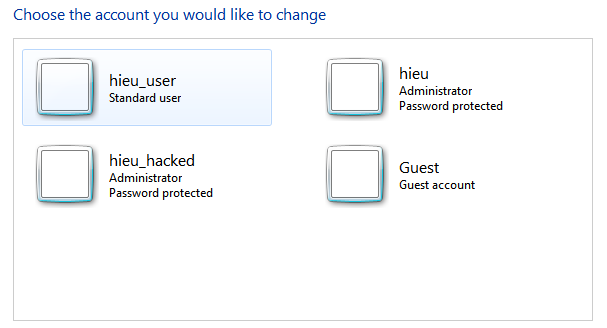
net localgroup administrators hieu\_hacked /add



+) User hieu\_hacked đã xuất hiện:



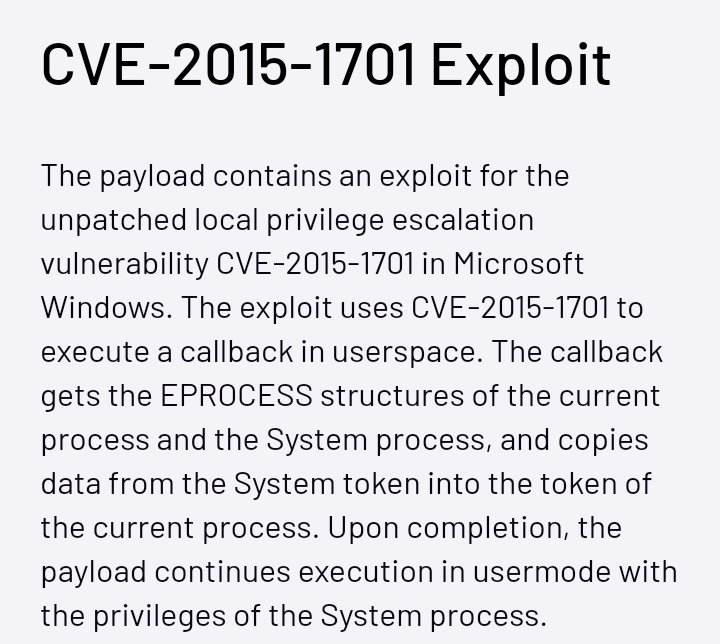
Ta thử nhập mật khẩu và check vai trò của user hieu\_hacked:



Thành công, ta đã tạo một tài khoản Administrator thông qua lỗ hổng của CVE-2015-1701.

II) Tìm kiếm tài liệu:

Tìm kiếm về CVE-2015-1701, ta có:



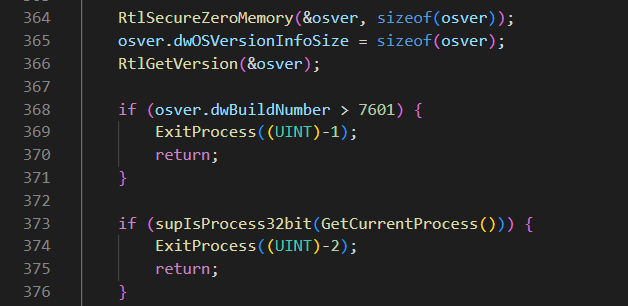
Ta có thể hiểu được rằng exploit sẽ sử dụng CVE-2015-1701 để tạo một callback ở trong userspace (phân biệt với kernel space). Callback sẽ lấy cấu trúc EPROCESS của tiến trình (process) hiện tại và của tiến trình system , **sau đó sao chép token của System để ghi vào token của tiến trình hiện tại.** Nhờ đó, tiến trình hiện tại sẽ được máy hiểu là tiến trình của System, có đầy đủ các quyền của System.

III) Phân tích code exploit:

**CHÚ Ý: PHÂN TÍCH CÓ THỂ SẼ CÓ NHIỀU SAI SÓT, CẦN ĐƯỢC PHÁT HIỆN VÀ CHỈNH SỬA TRONG QUÁ TRÌNH DEBUG.**

**Phân tích code exploit khi compile ra Taihou64.exe:**

Nhảy thẳng tới hàm main, ta có:



Dòng 368 sẽ kiểm tra xem hệ điều hành có phải Window 7 hay không, nếu KHÔNG thì thoát.

Dòng 373 sẽ kiểm tra xem file .exe này có phải WOW64 hay không, nếu ĐÚNG thì thoát. Hiểu một cách cơ bản, WOW64 là cấu trúc khác với cấu trúc của máy tính (nếu cấu trúc máy tính là 64-bit thì WOW64 là 32-bit). Mà ở đây ta có Taihou64 tức là cấu trúc 64-bit, nếu check cấu trúc của file mà ra WOW64 là 32-bit tức là không tương thích -> Trả về kết quả là ĐÚNG -> Thoát chương trình.

**Debug sáng ngày 19/01/2023:**

Tiếp đến chúng ta có:

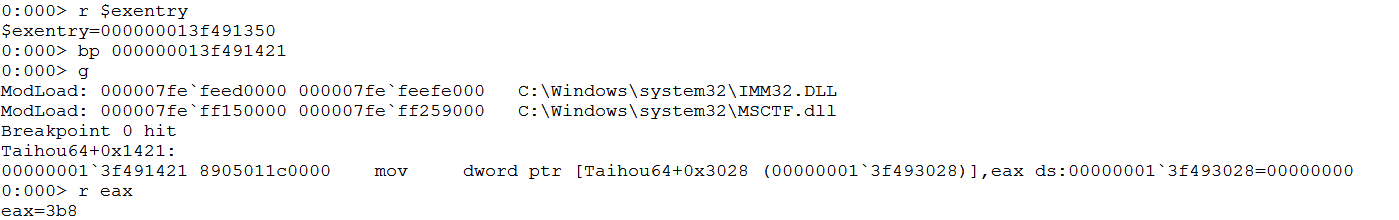


Ở dòng 377, chúng ta sẽ lấy ID của current process (tiến trình hiện tại).

Đặt break point tại $exentry + 0xD1 (winDBG command: bp $exentry + 0xd1):

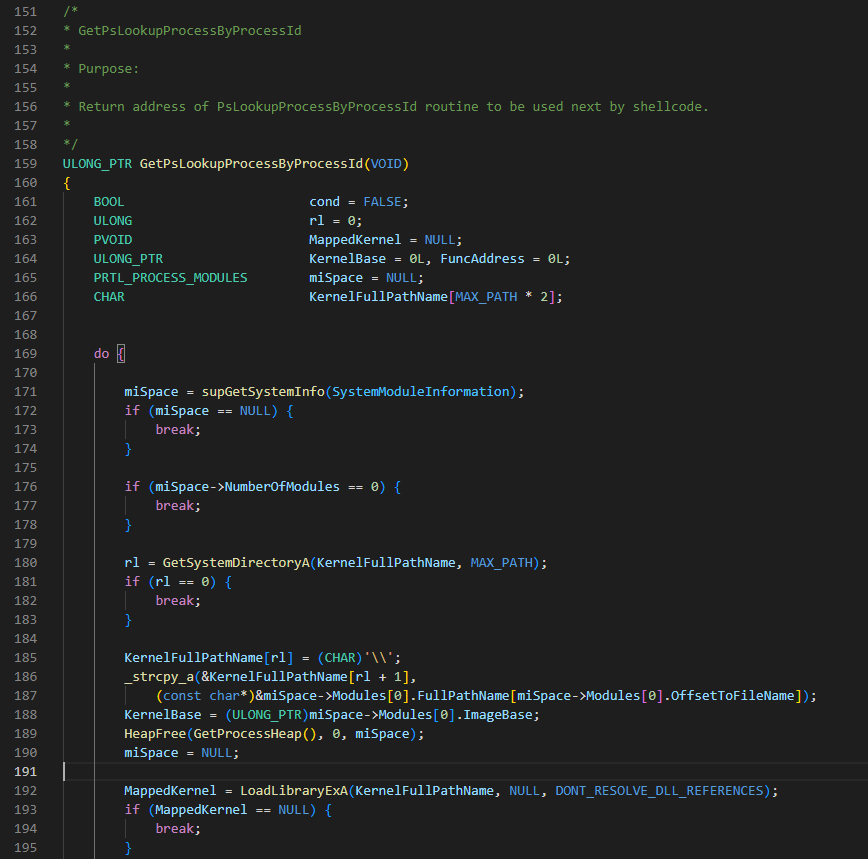


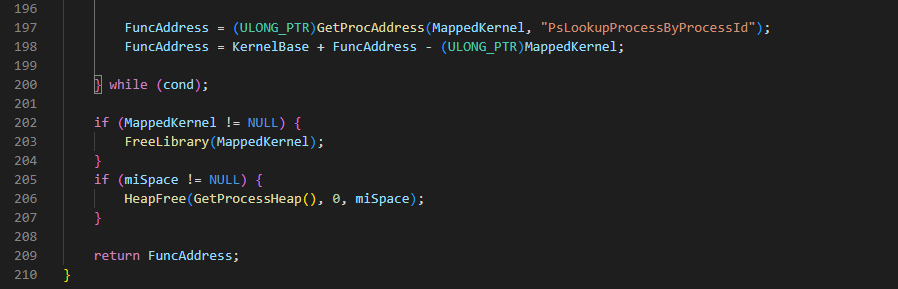
Ta có ID của process hiện tại là 0x3b8:



Ở dòng 378, chúng ta sẽ xem hàm GetPsLookupProcessByProcessId() làm gì:

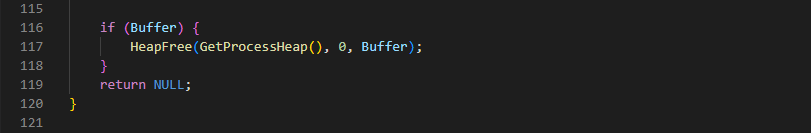
Ở hàm này chúng ta nhìn thấy một vòng lặp do-while nhưng trong IDA chúng ta không nhìn thấy được vòng lặp nào -> bỏ qua vòng lặp do-while, tập trung vào chủ đề chính:





Tại dòng 171, chúng ta thấy exploit gọi hàm supGetSystemInfo(SystemModuleInformation) :

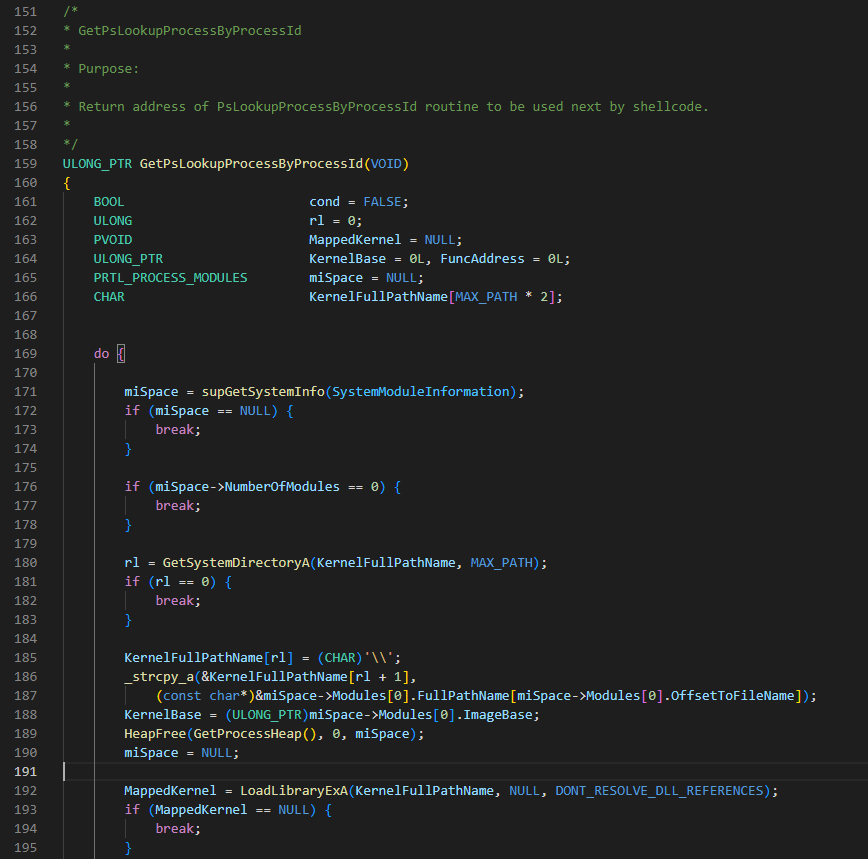


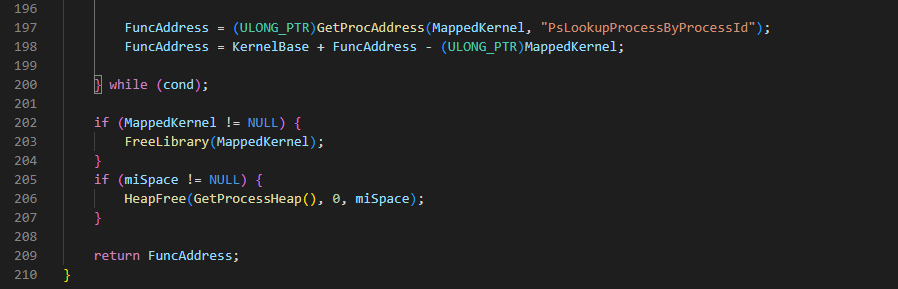


Hàm này sẽ trả về thông tin của SystemModuleInfomation thông qua hàm NtQuerySystemInformation(): [NtQuerySystemInformation function (winternl.h) - Win32 apps | Microsoft Learn](https://learn.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntquerysysteminformation)

Tham số là SystemModuleInfomation sẽ trả về cái gì? [List loaded drivers with NtQuerySystemInformation - Source Codes - rohitab.com - Forums](http://www.rohitab.com/discuss/topic/40696-list-loaded-drivers-with-ntquerysysteminformation/)

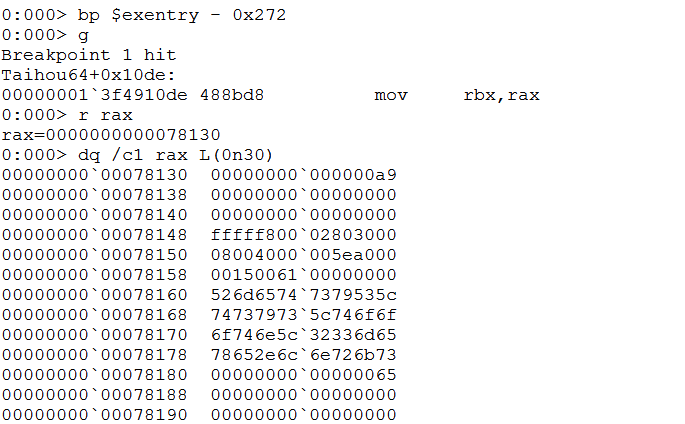
Quay trở lại hàm GetPsLookupProcessByProcessId():



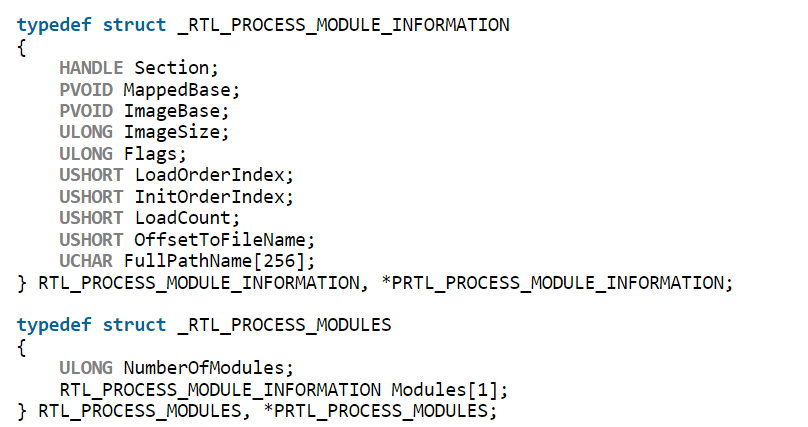


Sau khi gọi xong xong hàm supGetSystemInfo(SystemModuleInformation), ta sẽ xem dữ liệu trong miSpace:

Đặt breakpoint tại 0x10DE (WinDBG cmd: bp $exentry – 0x272):



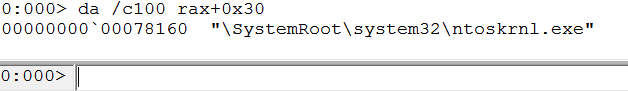
Cấu trúc của miSpace:



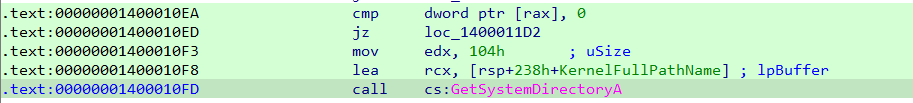
miSpace->NumberOfModules = 0xA9.

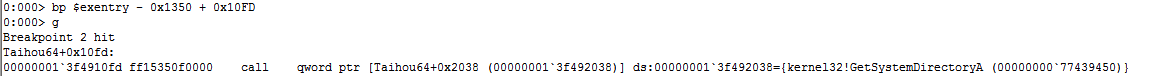
KernelBase = (ULONG\_PTR)miSpace->Modules[0].ImageBase = 0xFFFFF800’02803000

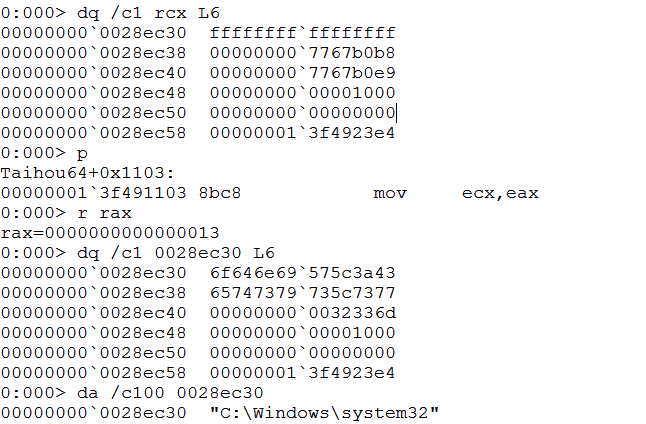
miSpace->Modules[0].FullPathName[miSpace->Modules[0].OffsetToFileName] = “\SystemRoot\system32\ntoskrnl.exe”



Đặt breakpoint ở 0x10FD xem GetSystemDirectoryA(KernelFullPathName, MAX\_PATH) sẽ làm gì với KernelFullPathName:





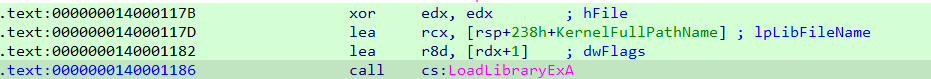


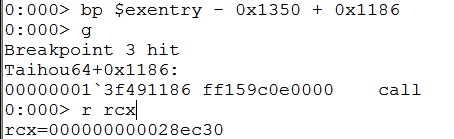
Nó sẽ trả về địa chỉ của SystemRoot ( ở đây là “C:\Windows\system32” ).

Sau đó ta quan sát tiếp tại:



Để xem cuối cùng KernelFullPathName là gì?



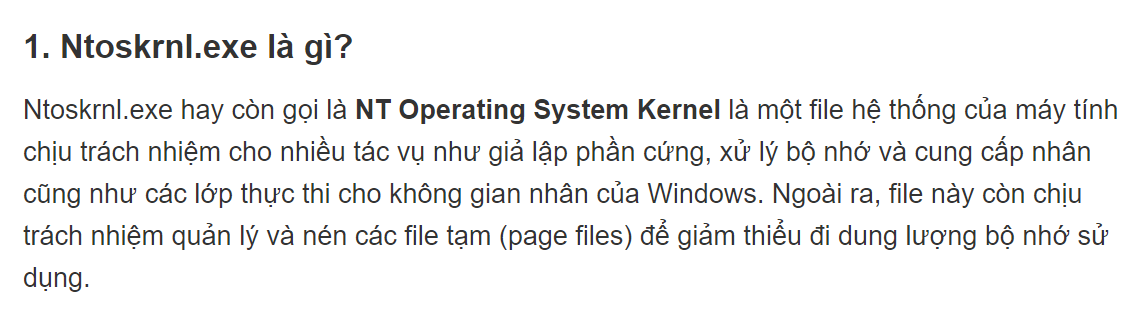
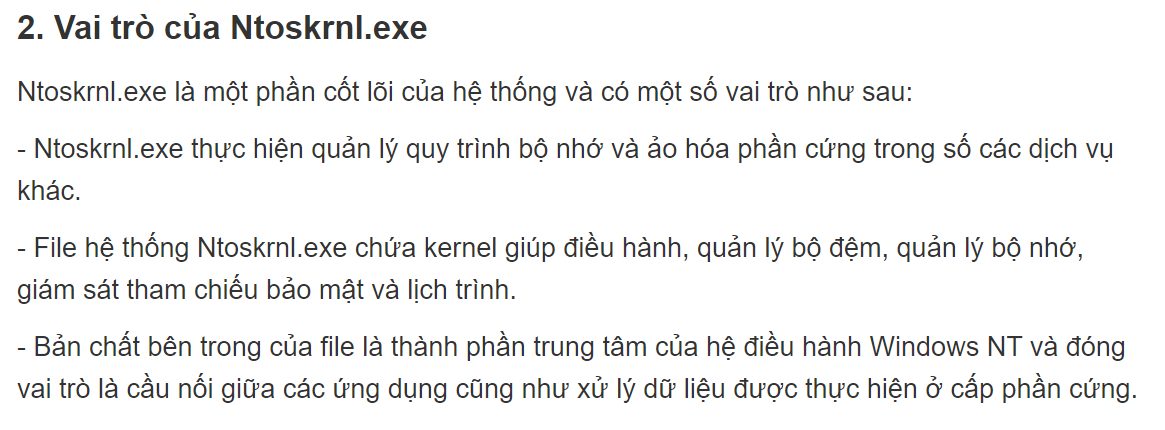




KernelFullPathName = “C:\Windows\system32\ntoskrnl.exe”

ntoskrnl.exe là gì?

[ntoskrnl.exe - Wikipedia](https://en.wikipedia.org/wiki/Ntoskrnl.exe)

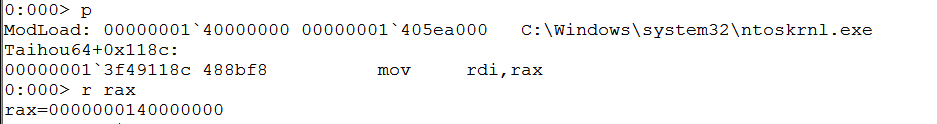
 

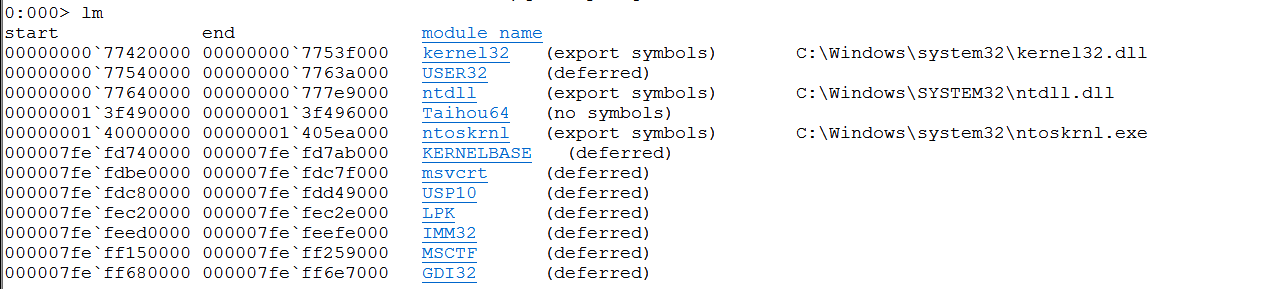
Hiểu nôm na, ntoskrnl.exe quản lí tất cả các thư viện được import.

Tiếp tục, ta sẽ xem MappedKernel sẽ có giá trị là gì:



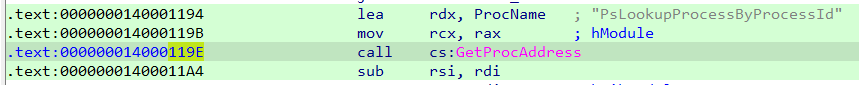
Chúng ta vừa break point ở 0x1186 nên chỉ cần step over là được:



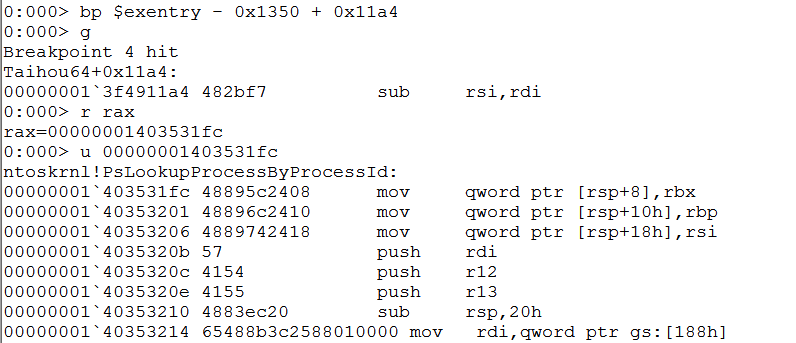


MappedKernel = 0x00000001’40000000 chính là base của ntoskrnl.exe.

Tiếp đó, ta xét giá trị của FuncAddress bằng cách đặt breakpoint tại 0x11a4:

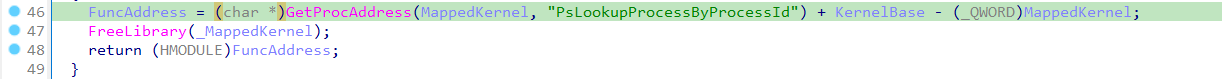




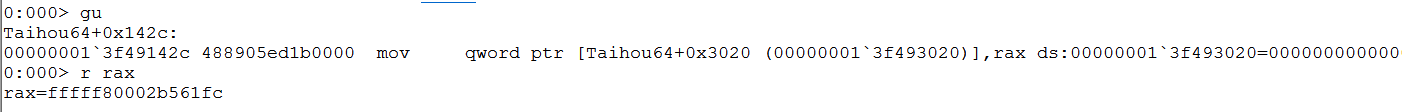


Sau đó:



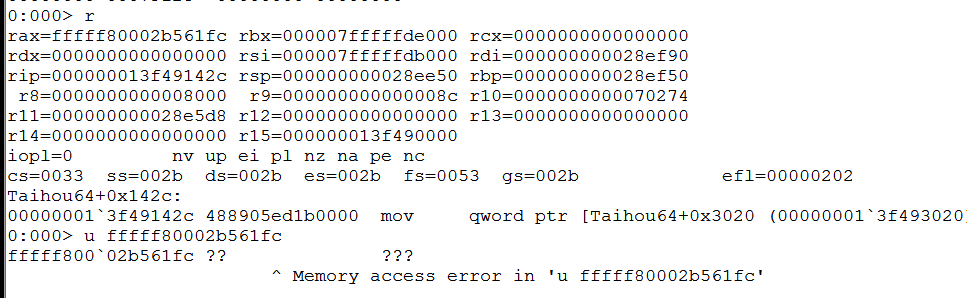


Đoạn này ta chỉ cần cho finish function đang chạy (GetPsLookupProcessByProcessId()) là được:





Câu hỏi: tại sao không lấy luôn GetPsLookupProcessByProcessId() để chạy mà lại phải lấy hàm GetPsLookupProcessByProcessId() tại KernelBase?



Mà cũng không disassem tại địa chỉ đó được?

Kết thúc debug này vào ngày 19/01/2023 tại dòng 378 trên code, ở debug mới các địa chỉ sẽ thay đổi, tuy nhiên 3-4 byte cuối vẫn được giữ nguyên.

Ngày 22/01/2023, cài đặt VirtualKD để debug các hàm trong kernel.

[(137) CVE-2015-1701 Windows ClientCopyImage Win32k Exploit - YouTube](https://www.youtube.com/watch?v=zYwHWBxEGZo&t=20s&ab_channel=EricRomang)

Ngày 23/01/2023, cài đặt thành công Debugee để debug -> Đã break được tại chương trình.

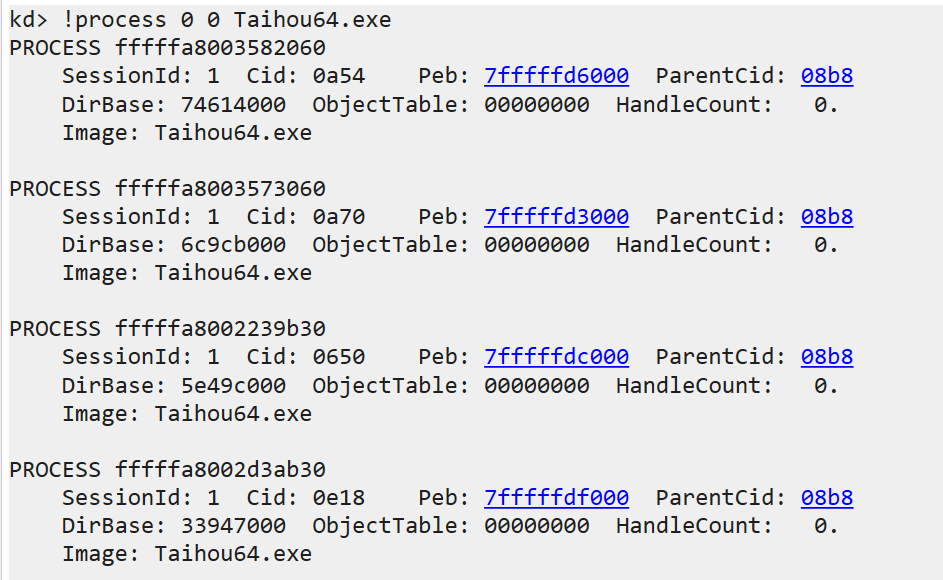
Ngày 24/01/2023:

Khởi động vmmon64.exe trong folder của VirtualKD-Redux trên máy Debugger (máy thật).

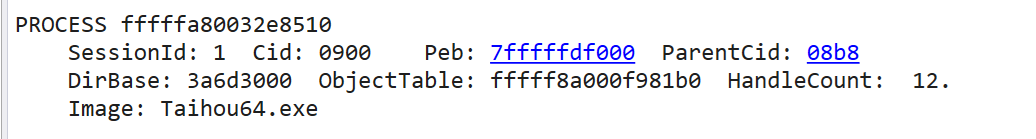
Chạy file Taihou64 trên máy ảo.

Giờ chúng ta sẽ tìm địa chỉ của Taihou64 để đặt các breakpoint.

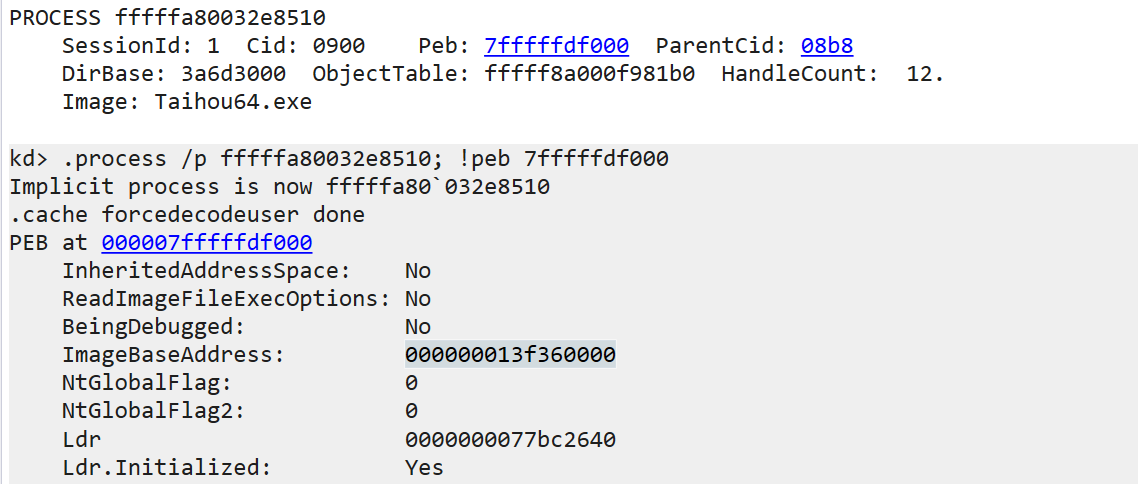
Vì ta có cài 1 hàm DebugBreak() trên file exploit build lại nên chúng ta có thể tìm process:



Tìm đến cái có HandleCount > 0:



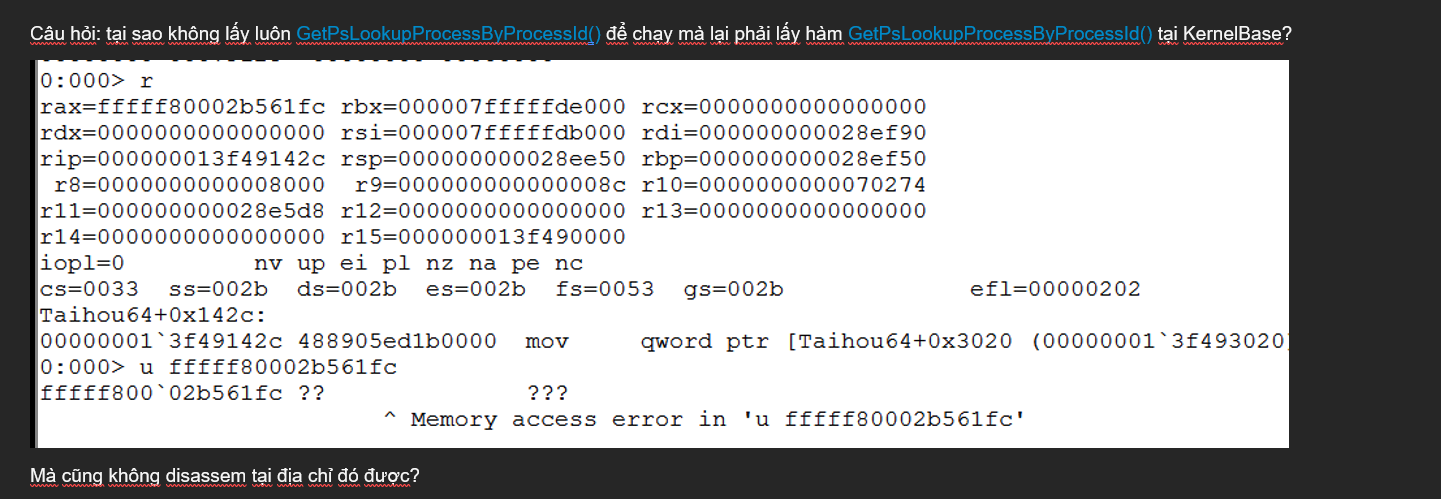
Ta có CID của process đó là 0x0900, tiến hành tìm base address bằng cách ấn vào cái Peb màu xanh xanh ở trên:



Ta có base address là: 0x000000013f360000. Tiến hành đặt 1 số breakpoint.

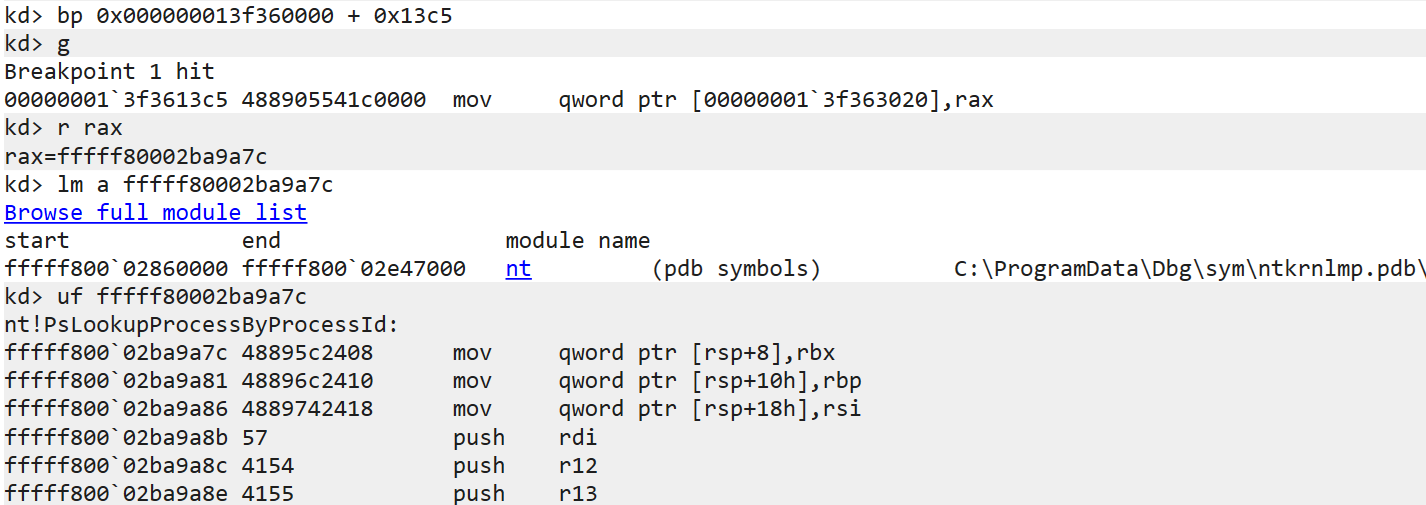
Ta sẽ đi trả lời cho câu hỏi hôm trước:

Câu hỏi hôm nọ:



Trả lời: Đã debug được, vì chương trình kia chạy ở User-Mode cho nên nó chỉ load các địa chỉ và process được gọi vào trong chương trình đó, không thể vượt quyền ra ngoài để truy cập đến các chỗ khác.

Ngoài ra, đó là hàm nt!PsLookupProcessByProcessId của module nt:

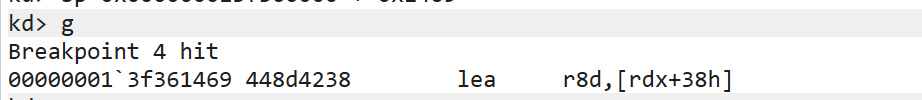


Tiếp tục với chương trình

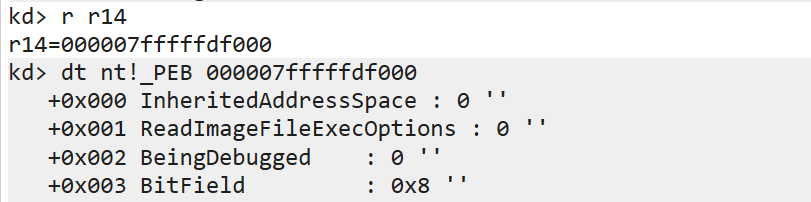
****

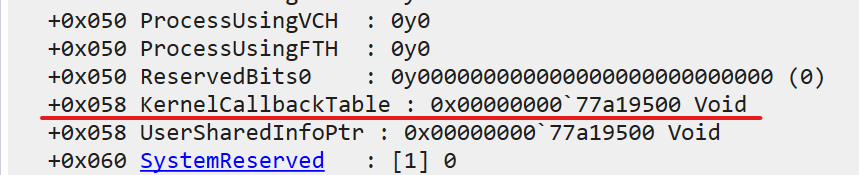
Ở dòng 402, ta tạo một biến g\_w32theadinfo = teb->Win32ThreadInfo, biến này không có tác dụng gì đặc biệt ngoài việc tí nữa sẽ dùng để kiểm tra xem có đúng là cái chương trình mình đang chạy không ( dùng cho hàm để lấy Handle của thread hiện tại )

Ta tìm hiểu tiếp biến g\_ppCCI:

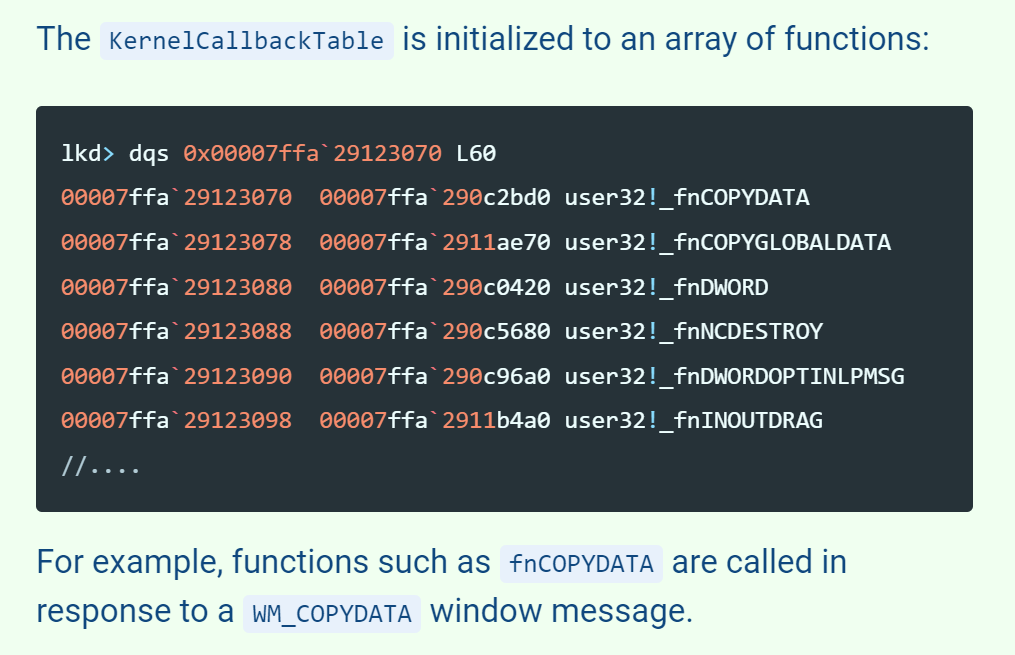


Ta lấy địa chỉ của PEB của tiến trình hiện tại, nằm ở thanh ghi r14 và kiểm tra địa chỉ của KernelCallbackTable:





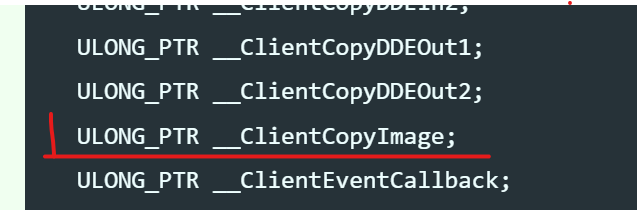
Vậy KernelCallbackTable là gì?



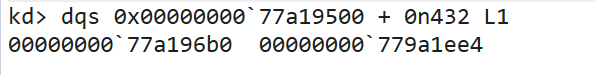
Đây là các hàm đọc và xử lí các thông tin trong một cửa sổ của Window. Tên gọi của các hàm tại link này:

[Process injection via KernelCallbackTable. Simple C++ malware example. - cocomelonc](https://cocomelonc.github.io/tutorial/2022/01/24/malware-injection-15.html)

Trong đó, ta có địa chỉ của \_\_ClientCopyImage là hàm số 0x36, địa chỉ 0x36 \* 8 (64-bit) = 432.

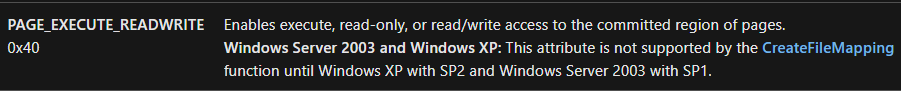






Tiếp đó, ta sẽ kiểm tra xem địa chỉ tại g\_ppCCI có quyền thực thi, đọc/ghi hay không, nếu không thì thoát. Cái này để đảm bảo ta có thể ghi giá trị vào g\_ppCCI khi thực hiện hàm InterlockedExchangePointer()





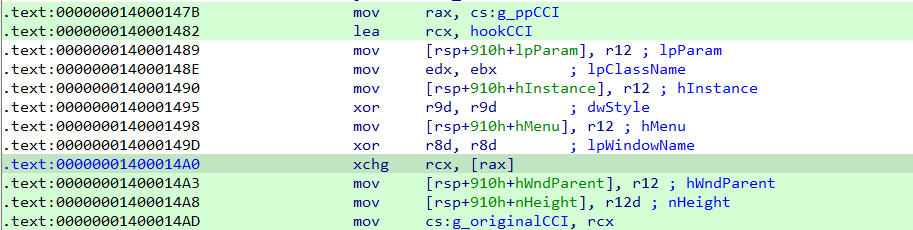


Tiếp đến dòng 409:



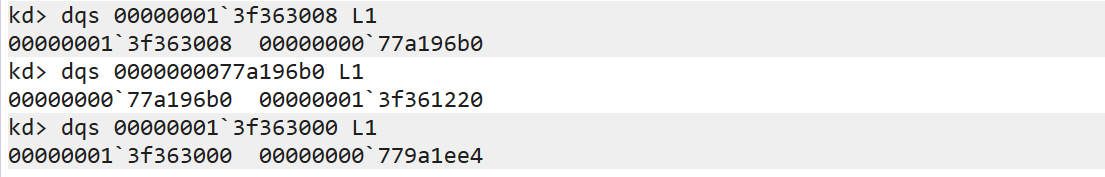
[InterlockedExchange64 function (winnt.h) - Win32 apps | Microsoft Learn](https://learn.microsoft.com/en-us/windows/win32/api/winnt/nf-winnt-interlockedexchange64)

Nhìn vào IDA:

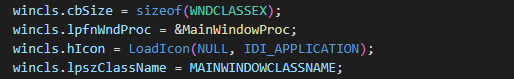


Thực ra chỉ cần quan tâm đến các dòng 147B, 1482, 14A0, 14AD, các dòng còn lại là do lỗi hiển thị của IDA.

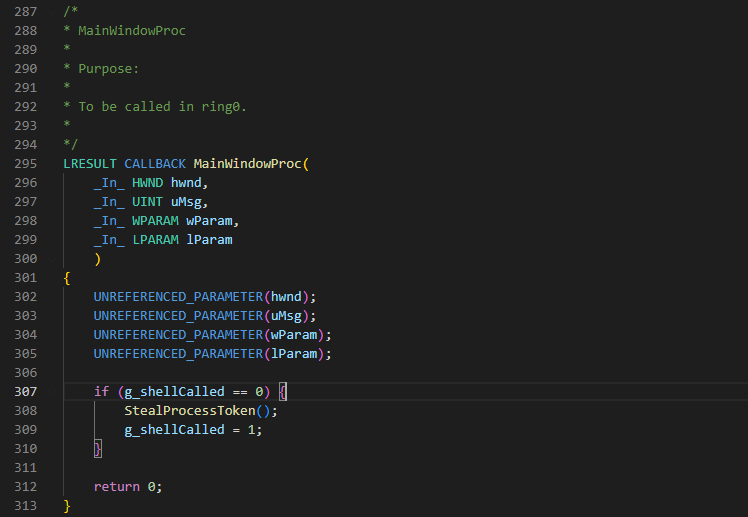
Hàm này gán g\_originalCCI = [g\_ppCCI] và [g\_ppCCI] = &hookCCI, sau khi thực hiện xong ta có (363008 là địa chỉ của g\_ppCCI, 196b0 là giá trị của g\_ppCCI, 363000 là địa chỉ của g\_originalCCI):



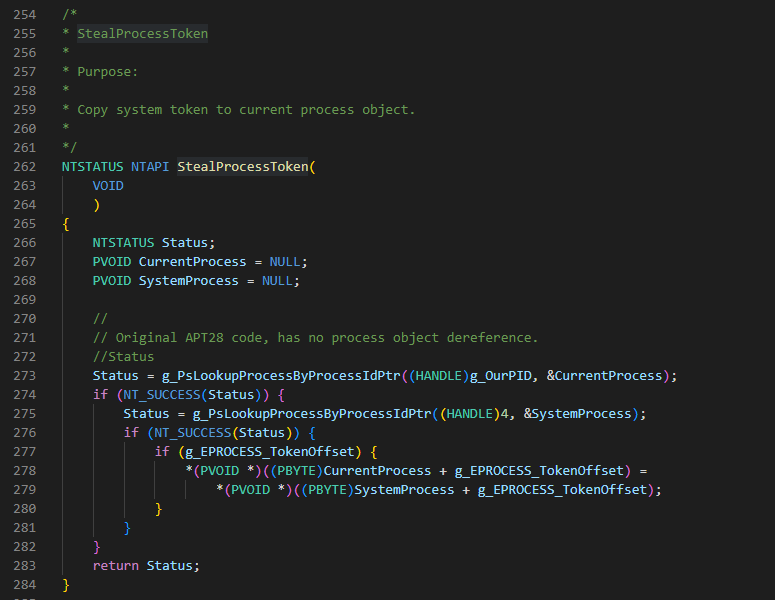
Tiếp đó ta sẽ xét đến đoạn code này:







Về UNREFERENCED\_PARAMETER: [windows - what's the purpose of the following about UNREFERENCED\_PARAMETER? - Stack Overflow](https://stackoverflow.com/questions/11229180/whats-the-purpose-of-the-following-about-unreferenced-parameter)



Chính là nó, đoạn này thay đổi TokenOffset của tiến trình hiện tại để khiến máy tính tưởng tiến trình hiện tại của chúng ta là Admin -> User nhưng được cấp quyền như Admin.

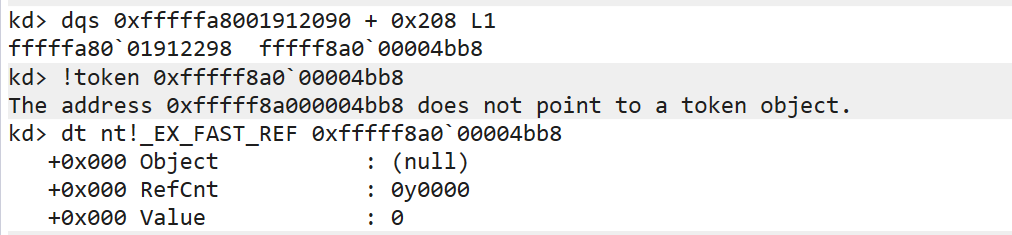
Tham khảo tại đây:

[Token Abuse for Privilege Escalation in Kernel - Red Team Notes (ired.team)](https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/how-kernel-exploits-abuse-tokens-for-privilege-escalation)

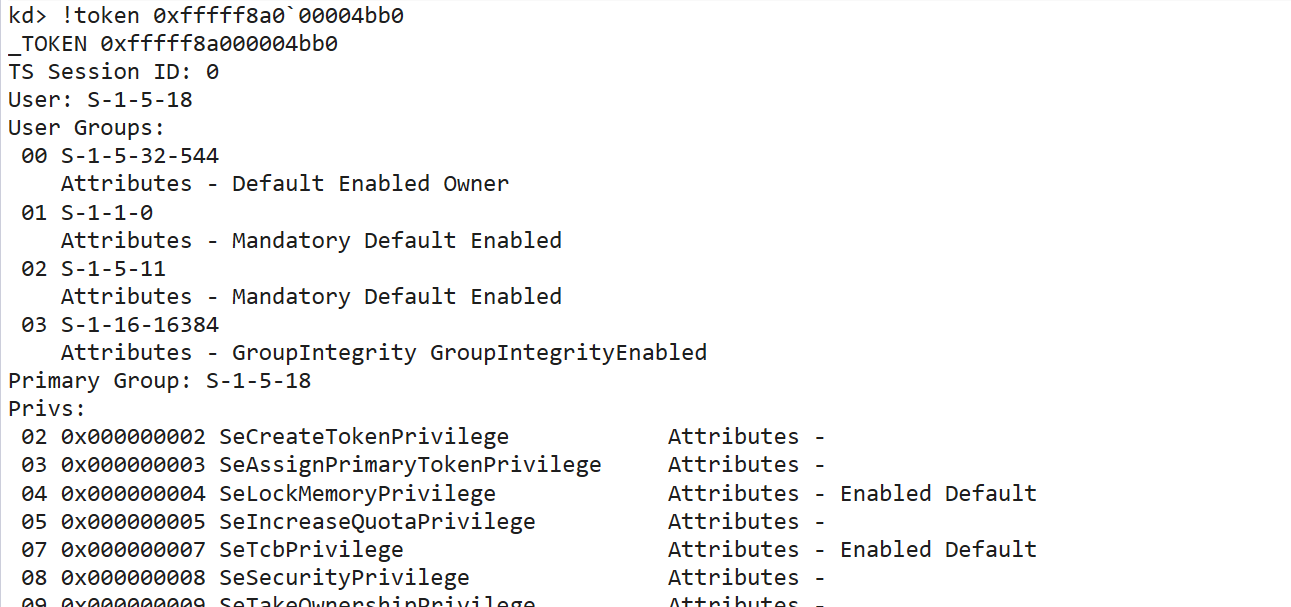
Ta kiểm tra tiến trình của System:



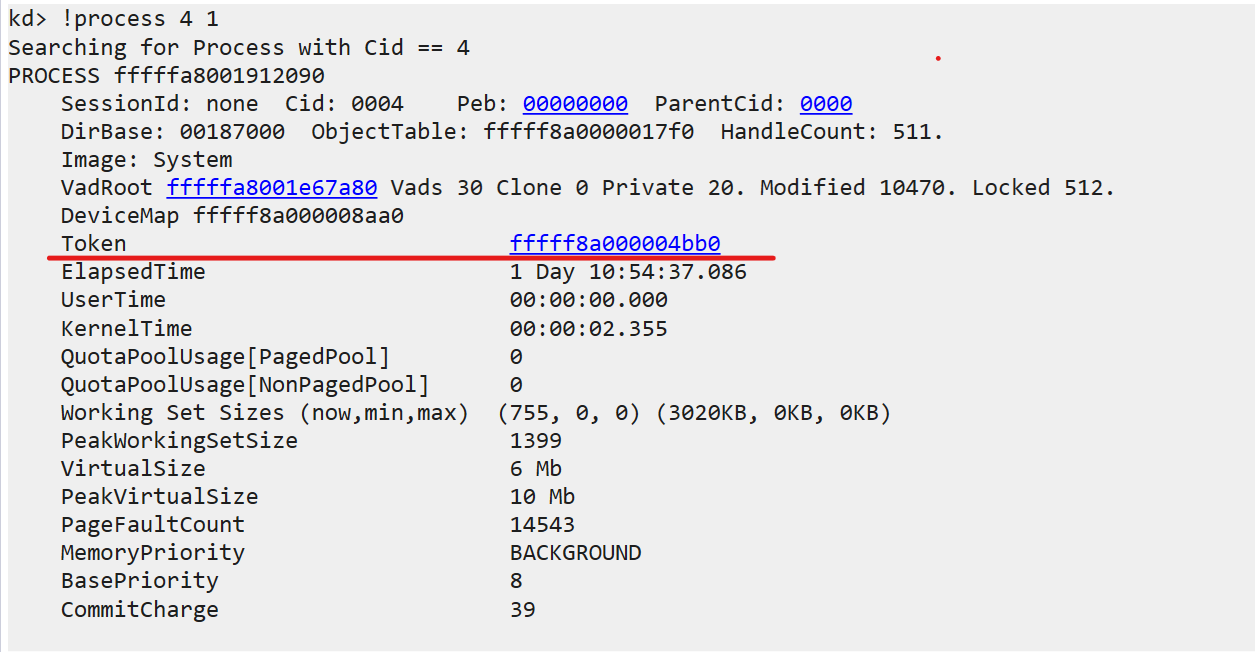


’

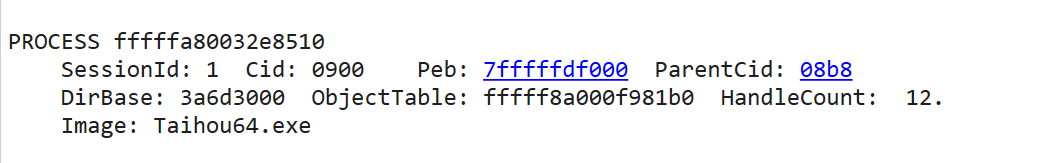
Ta thấy tại đó lại bảo phần token của chúng ta không phải là một token, đọc link trên thì ta chỉ cần đặt byte cuối chúng ta bằng 0 là được:

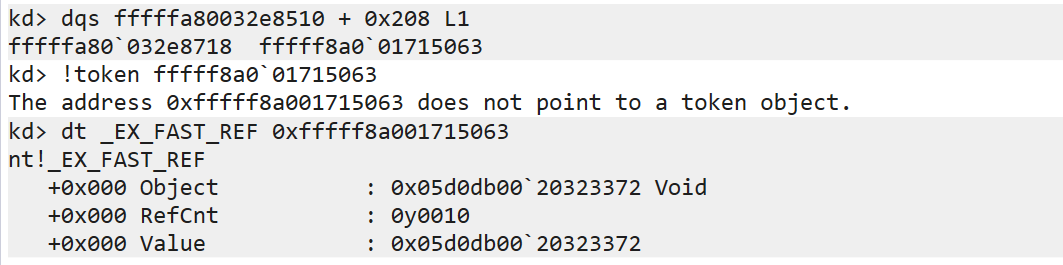


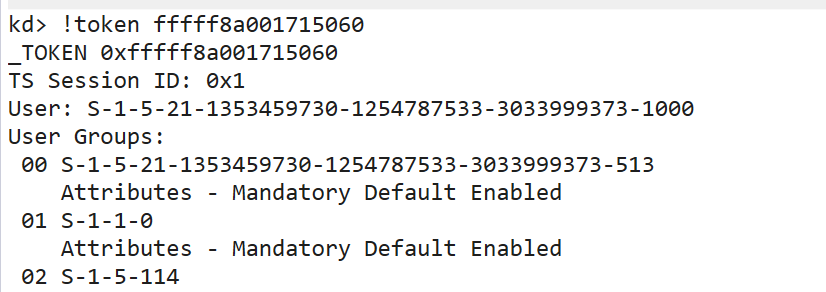
Hoặc chúng ta chỉ cần:



Ta kiểm tra tiến trình của Taihou64:

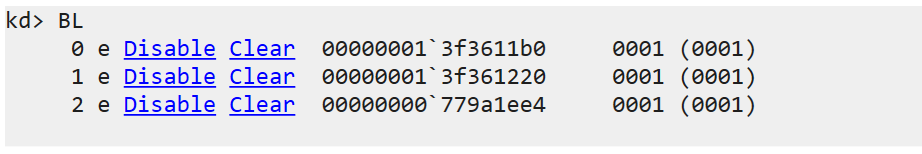




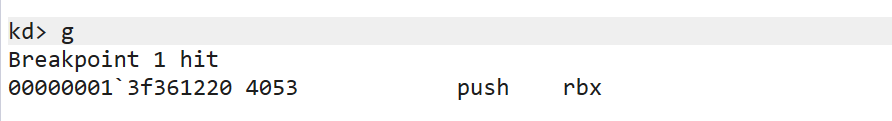


Vì thế chúng ta có thể thay đổi giá trị Token của tiến trình hiện tại (Taihou64.exe) (ở địa chỉ 0xfffffa80`032e8510 + 0x208) từ 0xfffff8a0`01715063 thành 0xfffff8a0`00004bb8 (giống như ở link trên).

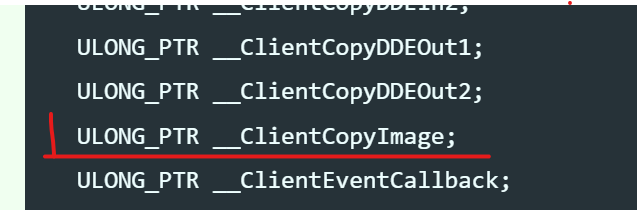
Ý tưởng là vậy, giờ ta đặt breakpoint ở 3 hàm là HookCCI , user32!\_\_ClientCopyImage, MainWindowProc để xem hệ thống sẽ chạy đến đâu trước.



Ta chạy:



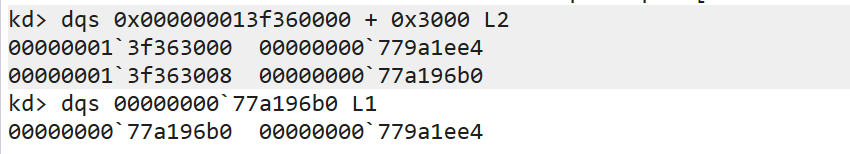
Ta thấy chạy đến HookCCI trước, chứng tỏ hàm CreateWindowEx() sẽ gọi \_\_ClientCopyImage khi khởi tạo cửa sổ, sau đó mới gọi MainWindowProc() (ta biết được vì ta vừa thay đổi giá trị tại nơi chứa địa chỉ hàm \_\_ClientCopyImage thành địa chỉ của hàm HookCCI() )



Ta sẽ khảo sát xem HookCCI làm gì:



Trả lại giá trị cho hàm \_\_ClientCopyImage():



25/01/2023: Lỗi xảy ra khi khởi động lại máy, ta sẽ chạy lại file Taihou64.exe với Base Address khác.

SetWindowLongPtr(handle, -4, value): nhận vào handle của thread đang chạy rồi thay đổi địa chỉ hàm WndProc của nó bằng value, giá trị trả về là địa chỉ của hàm WndProc cũ.

[SetWindowLongPtrA function (winuser.h) - Win32 apps | Microsoft Learn](https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowlongptra)

Chính là chỗ này:



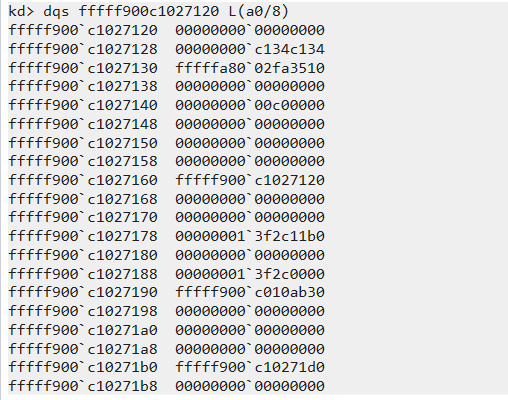
31/01/2023:

Các câu hỏi:

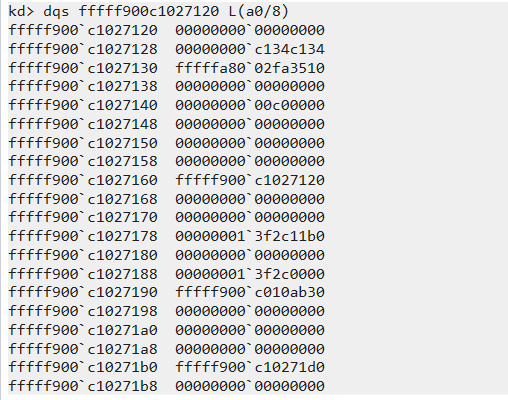
WndProc được lưu ở đâu:







Class hoàn chỉnh, atom là 0xc134:



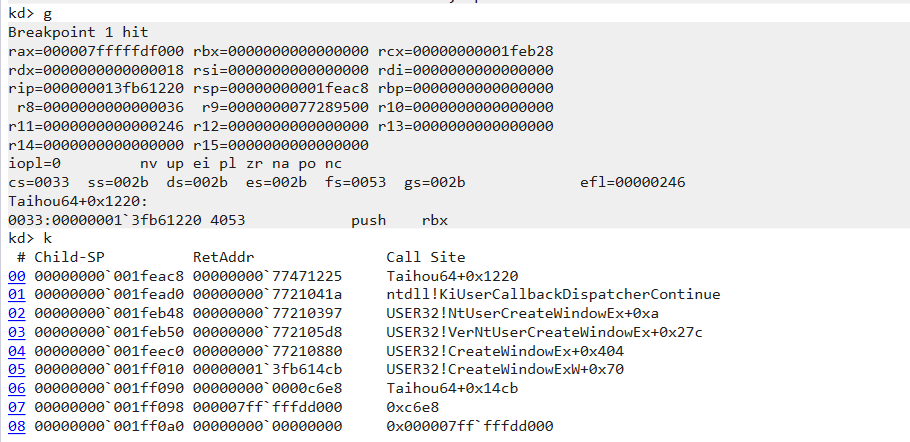
Hàm SetWindowLongPtr được sử dụng như thế nào mà sau khi set xong vẫn gọi hàm cũ lên? -> Trả lời: Đây chính là bug, giải thích ở phần kết luận.

Tại sao mà hàm PsLookupProcessByProcessID() chạy được trong khi đang gọi hàm CreateWindowEx() mà để bên ngoài lại crash? -> Trả lời: Vì PsLookupProcessByProcessID() trong hàm Taihou64+0x11b0 chỉ chạy được ở trong Kernel Mode. Đây chính là bug, giải thích ở phần kết luận.

Hàm MainWindowProc được gọi khi nào?

[List Of Windows Messages - WineHQ Wiki](https://wiki.winehq.org/List_Of_Windows_Messages) danh sách các hàm được gọi thông qua WndProc: 

Đầu tiên ta đi đến chỗ của \_ClientCopyImage:

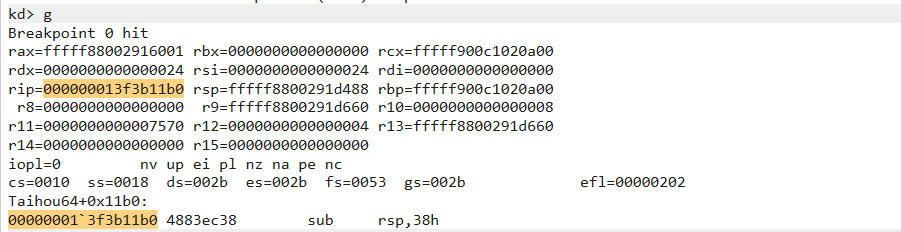


Gọi đến hàm MainWindowProc

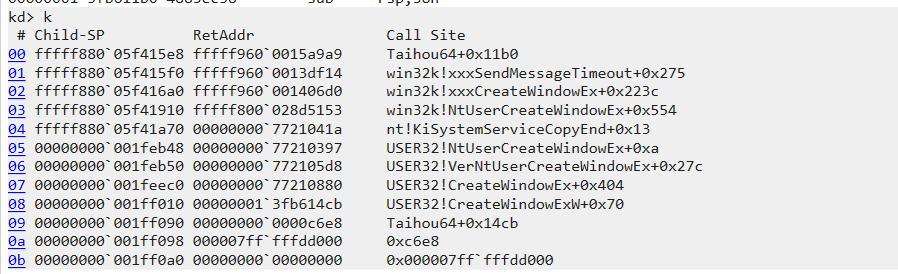
Lần đầu tiên:



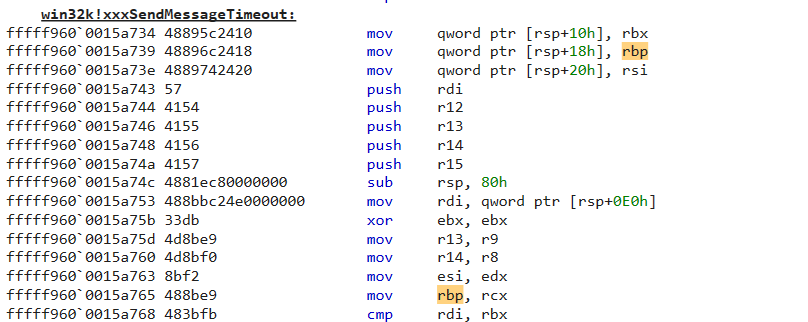
Gọi với uMsg 0x24:

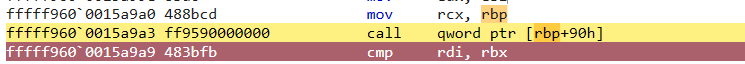


Lần thứ 2:



rbp là một parameter của hàm xxxSendMessageTimeout, tức là nó đã được định nghĩa từ trước:

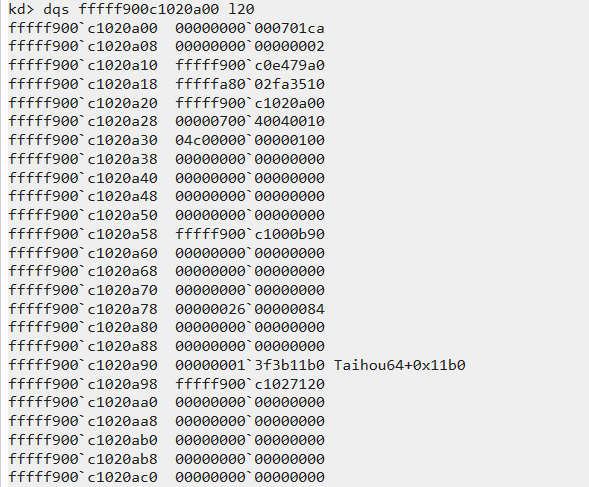




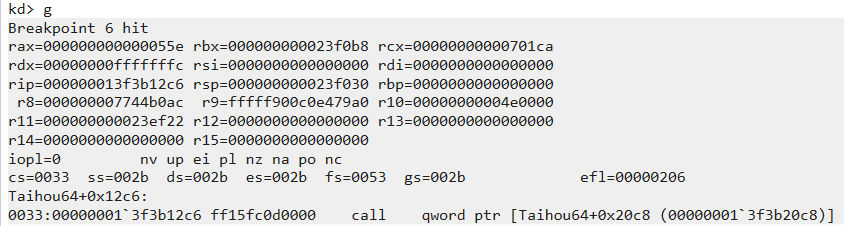
Ta để ý thấy fffff900c1020a00 được lưu rất lâu tại r12, tức là nó đã được define từ trước, giờ cần xem nó define tại đâu.

Ta thấy nó lấy thông tin từ địa chỉ fffff900c1020a00 trong đó 0x701ca là ThreadHandle của Taihou64.exe -> fffff900c1020a00 là một struct có chứa

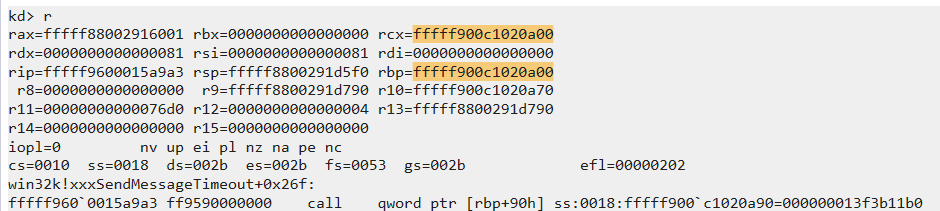
thông tin gì đó của thread:



ThreadHandle của Taihou64.exe nằm tại rcx:



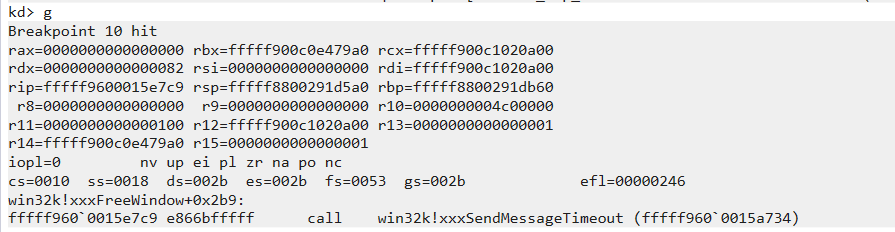
Về uMsg 0x81: [WM\_NCCREATE message (Winuser.h) - Win32 apps | Microsoft Learn](https://learn.microsoft.com/en-us/windows/win32/winmsg/wm-nccreate)



Lần thứ 3:



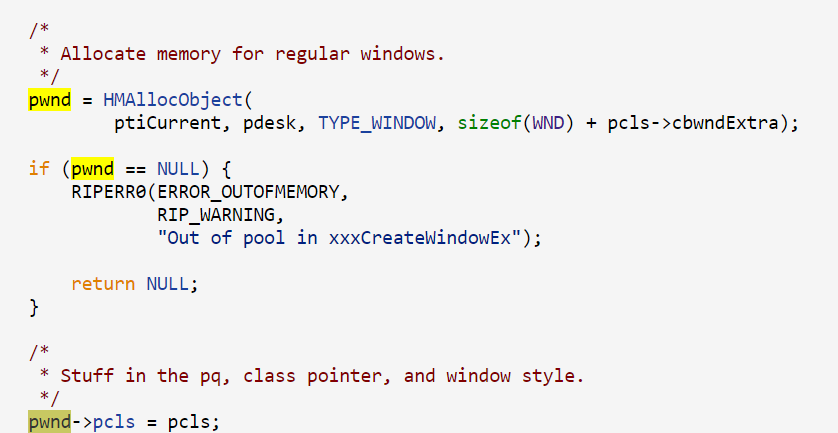
Gọi với uMsg 0x82:



Câu hỏi: sự xuất hiện của fffff900c1020a00, một struct có chứa thông tin về Thread của Handle, nó được khởi tạo như thế nào?

Khởi tạo tại xxxCreateWindowEx:

[test: createw.c Source File (gitee.io)](https://systemroot.gitee.io/pages/apiexplorer/d5/d7/createw_8c-source.html#l00033)

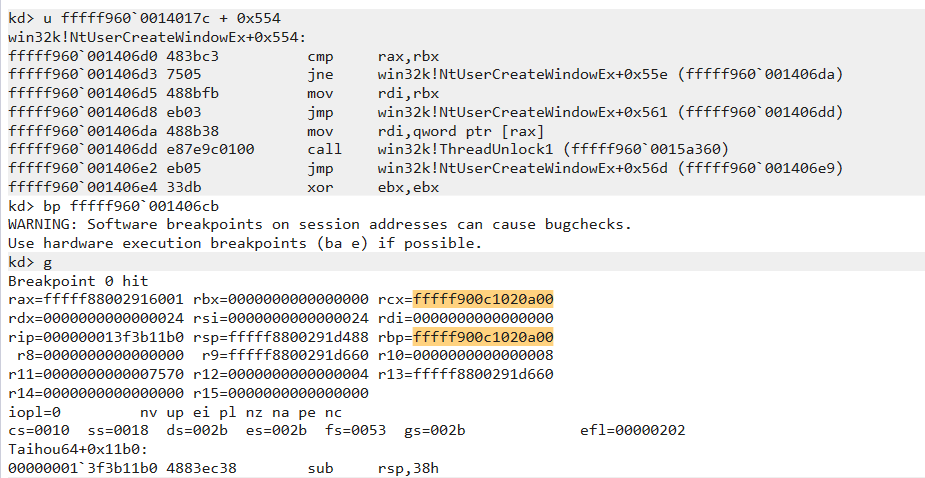
****

Trong đó pcls là class ta tạo ra ở hàm RegisterClassEx().

Struct đó là struct PWND.

[test: tagWND struct Reference (gitee.io)](https://systemroot.gitee.io/pages/apiexplorer/d6/d9/structtagWND.html)

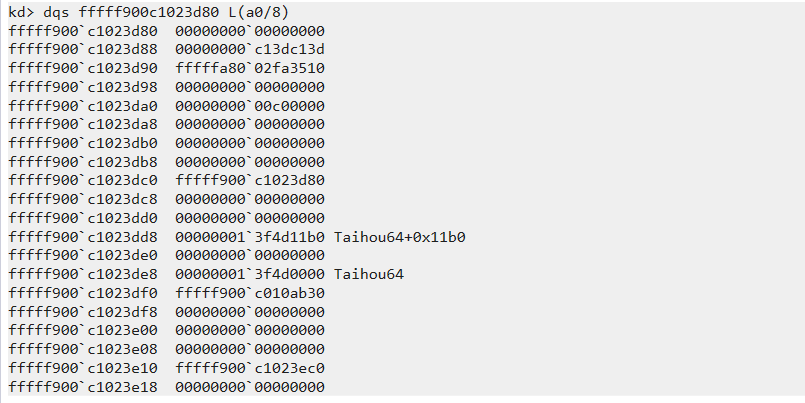
Trong hàm hookCCI có một hàm để kiếm ThreadHandle của Thread hiện tại, thử kiểm tra xem sao?



Trong hàm hookCCI có một hàm để kiếm ThreadHandle của Thread hiện tại, thử kiểm tra xem sao?

Ngày 31/01/2023, thử kiểm tra hookCCI:

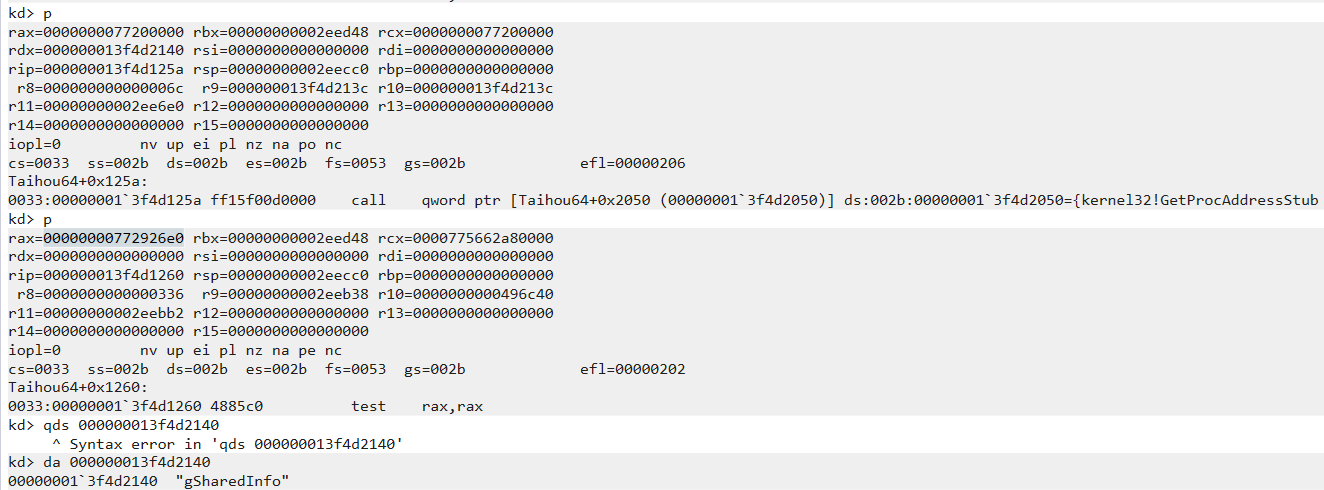
WndProc được lưu tại:

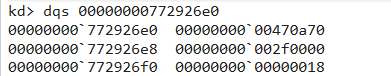


AtomID là 0xc13d.

Process Handle được lấy từ đâu? (31/01/2023)

Lấy địa chỉ của User32!gSharedInfo:

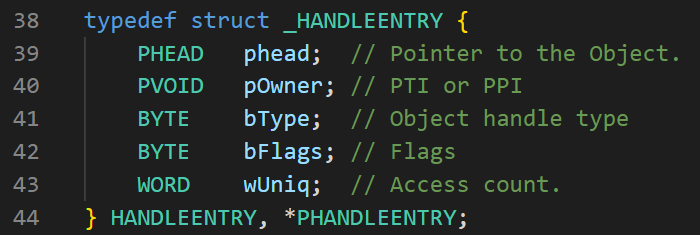


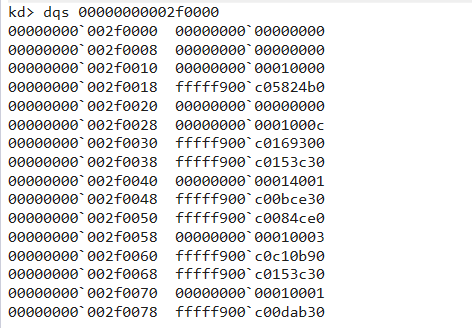


Trong đó psi là QWORD tại 6e0, aheList tại 6e8 và size tại 6f0:

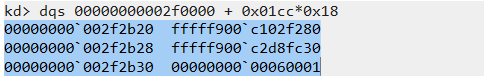


Tại aheList ta có size là 0x18, QWORD pOwner là Win32Thread của Thread đó:

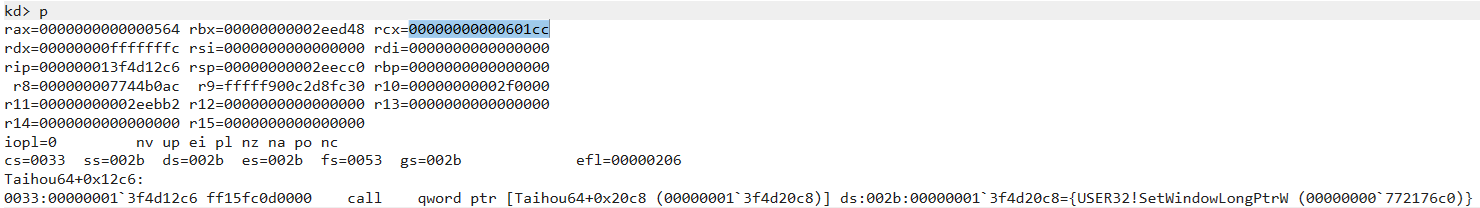




Ta tìm được Win32Thread của Thread Taihou64.exe là HandleEntry thứ 0x01cc:

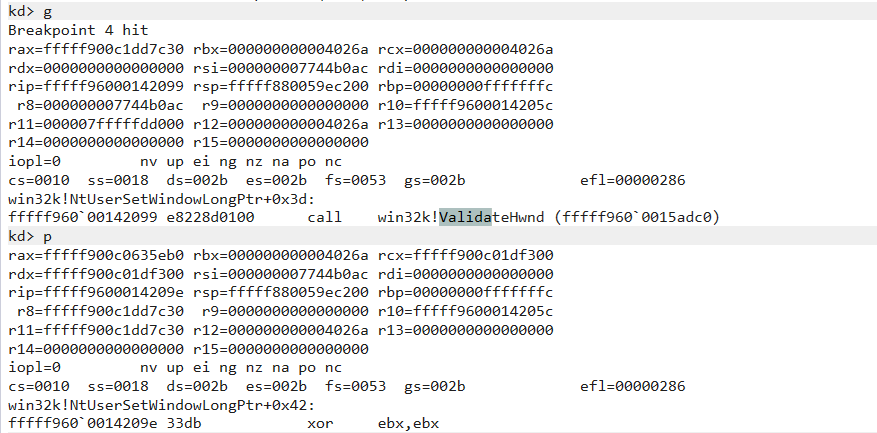


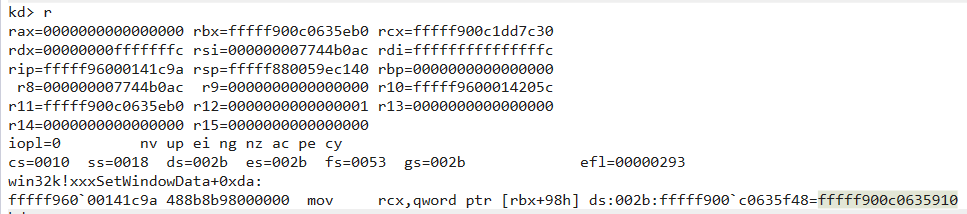
Sau đó ta sẽ lấy wUniq\*(2^16(DWORD)) + 0x01cc là ra Handle:



Tại sao SetWindowLongPtr() lại có được Taihou64+0x11b0 chỉ nhờ ThreadHandle?

Trả lời:





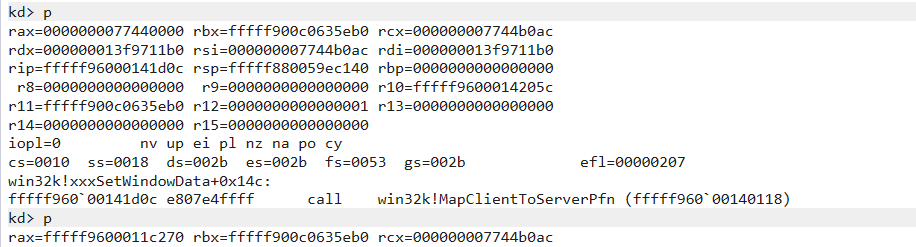


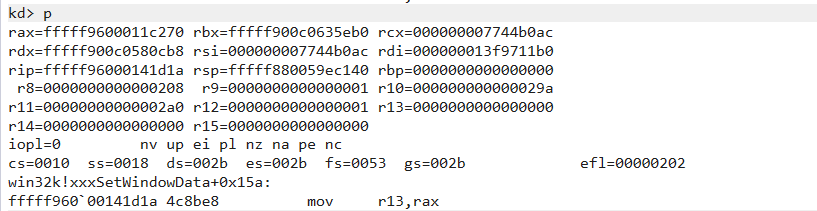
Địa chỉ fffff900`c0635910 chính là cái struct pcls ta khởi tạo ở hàm RegisterClassEx(), địa chỉ fffff900`c0635eb0 thì là cái pwnd trả về sau khi gọi hàm win32k!ValidateHwnd với parameter là ThreadHandle ở gSharedInfo tại User32.dll.

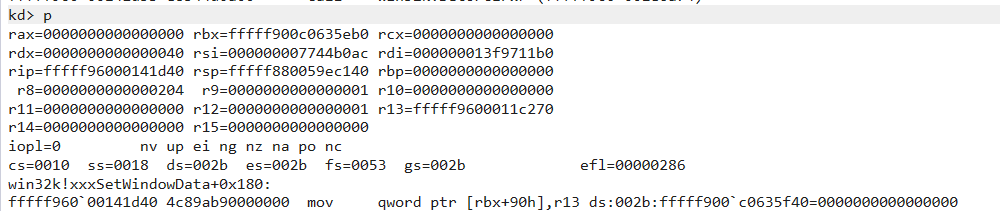
[0xfffff900`c0635eb0 + 0x98] = fffff900`c0635910, [fffff900`c0635910 + 0x58] = Taihou64+0x11b0

Thử check xem cái hàm SetWindowLongPtr() có thay đổi cái gì không?

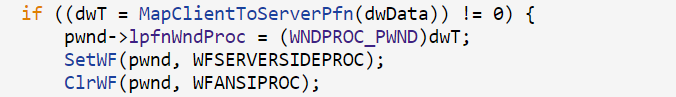
Có,hàm SetWindowLongPtr(ThreadHandle, WND\_PROC, &DefWindowProc) thay đổi giá trị của pwnd-> lpfnWndProc thành &win32k!xxxDefWindowProc:





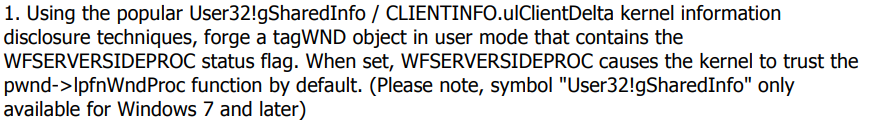






Đặt WFlag của pwnd thành WFSEVRVERSIDEPROC, thông báo rằng giờ hàm sau khi ta thay đổi là hàm của SERVERSIDE.

[Microsoft Word - FireEye\_letterhead\_v1.docx](https://www.fireeye.com/content/dam/fireeye-www/blog/pdfs/twoforonefinal.pdf)



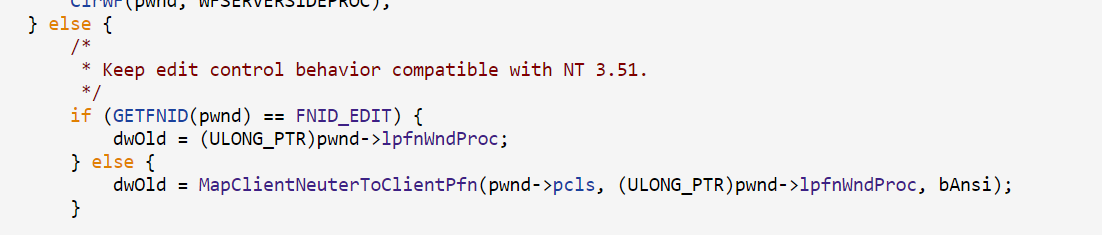
Một số thông tin:

+) Trong hàm

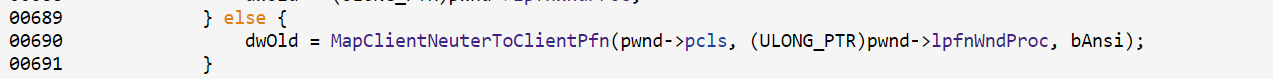
Trước đó:



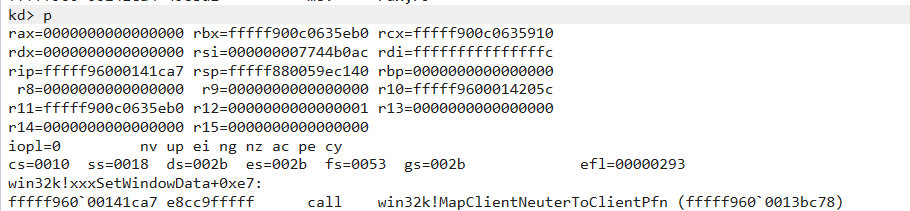
Hiện tại pwnd với WndProc = Taihou64+0x11b0 không phải là 1 hàm ở Server Side -> Flag WFSERVERSIDEPROC trả ra false:



Không có FNID\_EDIT tức là chưa edit pwnd->lpWndProc, tức pwnd đang rỗng, nhảy sang else:



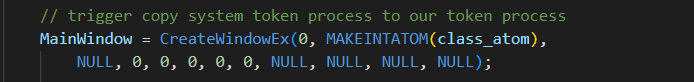
[test: userk.h File Reference (gitee.io)](https://systemroot.gitee.io/pages/apiexplorer/d4/d1/userk_8h.html#a1433)



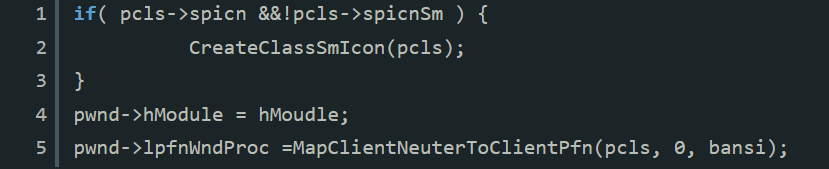
Khi gọi đến đoạn này, pwnd->lpfnWndProc chưa tồn tại. dwOld trả về là pwnd->pcls->lpfnWndProc = Taihou64+0x11b0.

Tóm gọn lại flow của chương trình:

Ta sẽ đi đến hàm này:

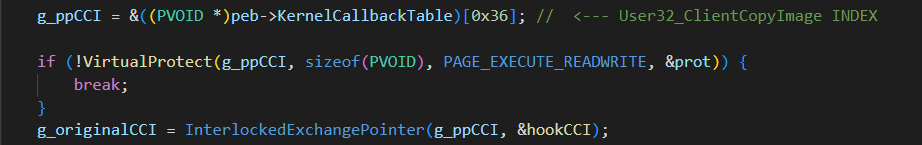


Dòng code này tạo ra một cửa sổ dựa vào thuộc tính của class\_atom vừa khởi tạo, và Win32k!xxxCreateWindows sau khi phân tích các thuộc tính của các tham số truyền vào, sẽ dử dụng HMAllocateObject để khởi tạo một cửa sổ win32, sau đó điền vào các tham số ví dụ như Windows Class Object:



Ở đây, mục đích của CreateClassSmIcon là để tạo ra một cache icon cho icon của cửa sổ đó (pcls), và tiếp đến, system sẽ đặt WindowProc. Hàm CreateClassSmIcon được gọi từ KeUserModeCallBack, lấy từ PEB->KernelCallBack, gọi lần lượt từ trên xuống dưới và execute nó ở User-Mode, trong đó \_ClientCopyImage là hàm cuối cùng được gọi.

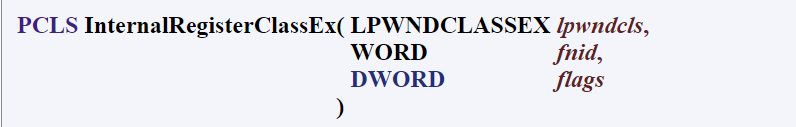
Ta thay thế hàm USER32!\_ClientCopyImage bằng hàm hookCCI:

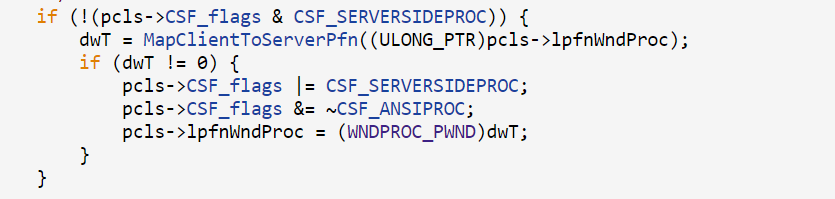


Ta USER32!\_ClientCopyImage có là hàm cuối cùng được gọi trong KernelCallbackTable

Sau đó ta để ý:

Trong hàm RegisterClassEx:

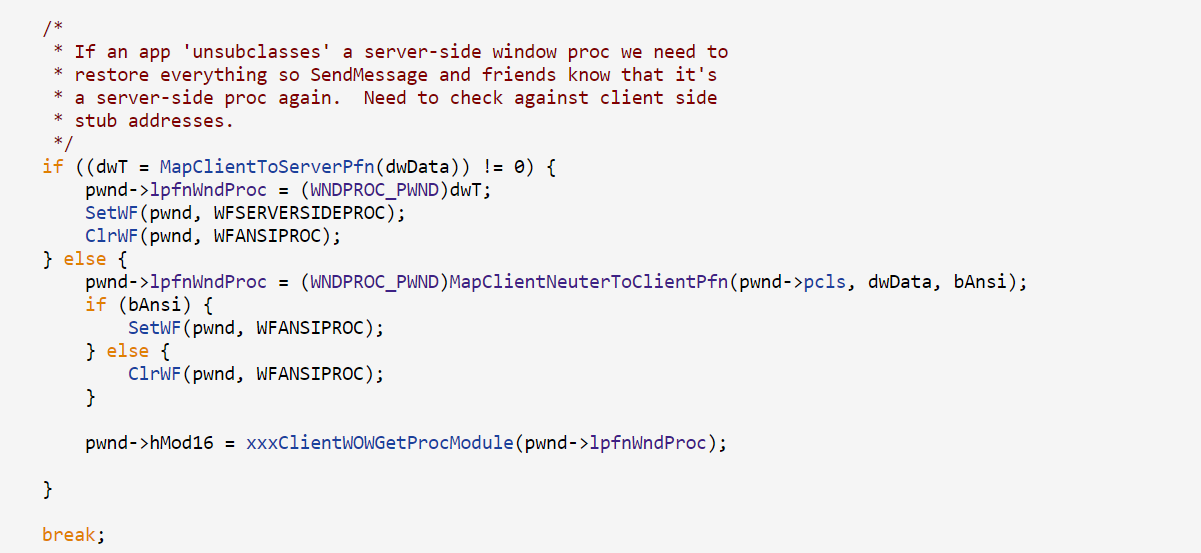




Trong hàm xxxSetWindowData:

[test: getset.c Source File (gitee.io)](https://systemroot.gitee.io/pages/apiexplorer/d1/d4/ntuser_2kernel_2getset_8c-source.html#l00260) ([xxxSetWindowLongPtr](https://systemroot.gitee.io/pages/apiexplorer/d4/d1/userk_8h.html#a586))

[test: userk.h File Reference (gitee.io)](https://systemroot.gitee.io/pages/apiexplorer/d4/d1/userk_8h.html#a1433) (**xxxSetWindowData)**



Logic ở đây là check xem có phải GWLP\_WNDPROC là một hàm được định nghĩa sẵn (trong thư viện, ví dụ User32!DefWindowProc), và nếu đúng, hàm kiểm soát cửa sổ sẽ được đặt vào kernel, và đặt Flag phía Server Side Proc cho cửa sổ (WFSERVERSIDEPROC), có nghĩa rằng từ giờ hàm thủ tục cửa sổ sẽ được gọi trong kernel mode. CÙng lúc đó, hàm thủ tục cửa sổ (pwnd->lpfnWndProc) được sửa đổi tới hàm xử lí của kernel tương ứng với User32!DefWindowProc (Win32k!xxxDefWindowProc).

Kết luận:

Flow đúng là: Gọi hàm MapClientNeuterToClientPfn() để hoán cải WindowProc mặc định của cửa sổ hiện tại, và sau đó đưa WindowProc (trong đó pwnd->pcls->lpfnWndProc = Taihou64+0x11b0) tới đối tượng cửa sổ được kiểm soát bởi user, bởi vì Taihou64+0x11b0 không phải làm 1 hàm của kernel, cho nên chỉ có thể được chạy trong usermode.

Tuy nhiên, tiến trình bị chen ngang tại \_ClientCopyImage() ngay trước khi CreateWindowProc() cài WindowProc cho cửa sổ -> việc (SetWindowData chỉnh sửa WindowProc (pwnd)) này là không hợp lệ. Từ đó, windowProc (pwnd) vẫn được là WindowProc đặt bởi usermode (với pwnd->pcls->lpfnWndProc = Taihou64+0x11b0), nghĩa là hàm xử lí tiến trình cửa sổ cũng được đặt là MainWindowProc (Taihou64+0x11b0), trong khi WFSERVERSIDEPROC đã được set. Đây chính là bug -> bởi vì tại thời điểm này, cờ của cửa sổ đã được đặt là cần phải chạy WindowProc ở kernel mode (WFSERVERSIDEPROC flag), cho nên khi mà SendMessage() và các hàm khác gửi messages tới cửa sổ này, nó sẽ nhảy ngay tới kernel mode và gọi cái hàm usermode để xử lí (dù đúng ra cái hàm usermode Taihou64+0x11b0 này phải chạy ở usermode) -> ta có thể thực thi kernel code thoải mái.

