Vulnérabilité Microsoft Windows GDI (MS07-017)
De l’analyse à l’exploitation

Introduction

Un process devant utiliser des objets ou des fonctions liés à l'affichage a à sa disposition une bibliothèque appelée GDI (Graphics Device Interface). La GDI permet de faire la jonction entre les applications et les pilotes graphiques. Au niveau utilisateur, ces API sont exportées dans leur majorité par les modules user32.dll et gdi32.dll. Mais rares sont les fonctions qui sont exécutées en mode utilisateur. En effet, les systèmes basés sur l'architecture NT, ont la particularité d'avoir tout le traitement de l'affichage qui s'effectue en mode noyau. Ceci a été implanté à l'origine afin d'optimiser le traitement général de l'affichage. Mais il y a une contrepartie de taille à cette architecture. En effet, dans cet environnement là, les failles qui peuvent exister dans la partie GDI du système ouvrent inévitablement des portes entre les applications utilisateurs et le noyau.
Sous les systèmes Windows 2000 et XP, il existe une petite erreur de programmation du module win32k.sys qui peut paraître anodine à première vue. Mais s'approchant de plus prêt, on peut sentir comme un courant d'air... Finalement, en fouillant un peu plus profondément, c'est une porte grande ouverte vers le Kernel's Wonderland que l'on découvre.

Etat des lieux

Lors de l'initialisation du système, le module win32k.sys crée une Section partagée (shared) et la mappe en mémoire système. Cette section va servir à stocker toutes sortes de data devant être accessible en lecture à toutes les applications utilisant des services du GDI.
Cette partie est traitée dans la fonction HmgCreate(). Elle est elle-même appelée dans la routine d'initialisation de win32k.sys (DriverEntry()).

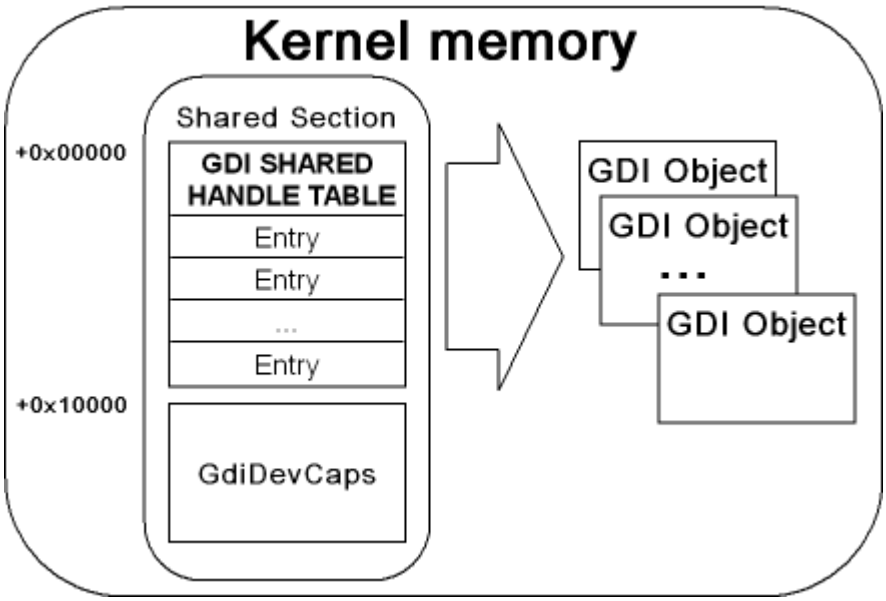
win32k.sys : DriverEntry() -> InitializeGre() -> HmgCreate()

```
BF890FC0 ; __stdcall HmgCreate()
BF890FC0 _HmgCreate@0 proc near
...
BF8910DB push ebx
BF8910DC push ebx
BF8910DD push 8400000h                  ; (SEC_COMMIT | SEC_NO_CHANGE) Attributes
BF8910E2 push edi                       ; edi=4 (PAGE_READWRITE) SectionPageProtection
BF8910E3 lea eax, [ebp+var_C]
BF8910E6 push eax
BF8910E7 push ebx
BF8910E8 push 6                         ; (SECTION_MAP_WRITE | SECTION_MAP_READ) DesiredAccess
BF8910EA push offset _gpHmgrSharedHandleSection
BF8910EF mov [ebp+var_8], ebx
BF8910F2 mov [ebp+var_C], 102ADCh          ; Size of Section
BF8910F9 call ds:__imp__MmCreateSection@32 ; MmCreateSection(x,x,x,x,x,x,x,x)
BF8910FF test eax, eax
BF891101 jl loc_BF89083A

BF891107 lea eax, [ebp+ViewSize]
BF89110A push eax                                  ; ViewSize
BF89110B push offset gpGdiSharedMemory             ; MappedBase
BF891110 push _gpHmgrSharedHandleSection           ; Section
BF891116 mov [ebp+ViewSize], ebx
BF891119 call ds:__imp__MmMapViewInSessionSpace@12 ; MmMapViewInSessionSpace(x,x,x)
BF89111F test eax, eax
BF891121 jl loc_BF89083A

BF891127 mov eax, gpGdiSharedMemory
BF89112C cmp eax, ebx
BF89112E lea ecx, [eax+100000h]
BF891134 mov _gpentHmgr, eax
BF891139 mov _gpGdiDevCaps, ecx
BF89113F jz loc_BF89083A
etc...
```

A ce stade là, nous avons donc une jolie section shared mappée en mémoire système. Et cette section est constituée de 2 parties distinctes. Nous allons nous intéresser à la première partie (0x0 à 0x100000). Cette zone est une grande table qui permet au système de gérer les objets du GDI :



Abordons maintenant la partie application utilisateur.
Lorsqu'une application devant utiliser des services graphiques est créée, le système mappe automatiquement une vue de la section dans l'espace mémoire de celle-ci. Pour des raisons évidentes de stabilité du système, cette vue est mappée en lecture seulement (READ_ONLY).

Cette opération se déroule dans la fonction win32k!GdiProcessCallout.

ntoskrnl.exe -> win32k!W32pProcessCallout -> GdiProcessCallout

```
BF8BFABB ; int __stdcall GdiProcessCallout(PVOID BaseAddress,HANDLE SectionHandle)
BF8BFABB _GdiProcessCallout@8 proc near
...
(ebx=0)
...
BF8BFB26 xor eax, eax
BF8BFB28 lea edi, [esi+PEB.GdiHandleBuffer]
BF8BFB2E rep stosd

BF8BFB30 lea eax, [ebp+SectionHandle]
BF8BFB33 push eax
BF8BFB34 push ebx
BF8BFB35 push ebx
BF8BFB36 push SECTION_ALL_ACCESS         ; DesiredAccess
BF8BFB3B push ebx
BF8BFB3C push ebx                        ; (0 -> no handle attribute) HandleAttributes
BF8BFB3D push _gpHmgrSharedHandleSection
BF8BFB43 mov [ebp+BaseAddress], ebx
BF8BFB46 mov [ebp+ViewSize], ebx
BF8BFB49 mov [ebp+SectionHandle], ebx
BF8BFB4C call ds:__imp__ObOpenObjectByPointer@28 ; ObOpenObjectByPointer(x,x,x,x,x,x,x)
BF8BFB52 test eax, eax
BF8BFB54 jl loc_BF8BFAAC

BF8BFB5A push PAGE_READONLY             ; Protect
BF8BFB5C push ebx                       ; AllocationType
BF8BFB5D push ViewUnmap                 ; InheritDisposition
BF8BFB5F lea eax, [ebp+ViewSize]
BF8BFB62 push eax                       ; ViewSize
BF8BFB63 push ebx                            ; SectionOffset
BF8BFB64 push ebx                            ; CommitSize
BF8BFB65 push ebx                            ; ZeroBits
BF8BFB66 lea eax, [ebp+BaseAddress]
BF8BFB69 push eax                            ; BaseAddress
BF8BFB6A push 0FFFFFFFFh                     ; ProcessHandle
BF8BFB6C push [ebp+SectionHandle]            ; SectionHandle
BF8BFB6F call ds:__imp__ZwMapViewOfSection@40 ; ZwMapViewOfSection(x,x,x,x,x,x,x,x,x,x)
BF8BFB75 mov edi, eax
BF8BFB77 cmp edi, ebx
BF8BFB79 jl loc_BF8BFAAC
```

```
BF8BFB7F mov eax, [ebp+BaseAddress]
BF8BFB82 mov [esi+PEB.GdiSharedHandleTable], eax
BF8BFB88
BF8BFB88 loc_BF8BFB88:                          ; CODE XREF: GdiProcessCallout(x,x)-A j
BF8BFB88                                        ; GdiProcessCallout(x,x)+5B j
BF8BFB88 mov eax, edi
BF8BFB8A
BF8BFB8A loc_BF8BFB8A:                          ; CODE XREF: GdiProcessCallout(x,x)+13A j
BF8BFB8A pop edi
BF8BFB8B pop esi
BF8BFB8C pop ebx
BF8BFB8D leave
BF8BFB8E retn 8
Forget to close section handle opened by ObOpenObjectByPointer !!!
```

Après cela, la vue est accessible en lecture seule par l'application et le pointeur de celle-ci se trouve placé dans le PEB (**PEB.GdiSharedHandleTable**).
Mais c'est gdi32.dll qui utilisera principalement cette vue. Ces informations sont directement stockées dans les datas de gdi32 :

gdi32.dll!pGdiSharedMemory
gdi32.dll!pGdiSharedHandleTable
gdi32.dll!pGdiDevCaps

Le code suivant montre comment gdi32.dll récupère ces 2 pointeurs :

```
gdi32.dll : GdiDllInitialize() -> GdiProcessSetup ()

77EF663F ; __stdcall GdiProcessSetup()
77EF663F public _GdiProcessSetup@0
77EF663F _GdiProcessSetup@0 proc near
...
77EF6773 mov eax, large fs:18h   ; TEB
77EF6779 mov eax, [eax+TEB.Peb]
77EF677C mov eax, [eax+PEB.GdiSharedHandleTable]
77EF6782 mov _pGdiSharedMemory, eax
77EF6787 mov _pGdiSharedHandleTable, eax
77EF678C add eax, 100000h
...
77EF6796 mov _pGdiDevCaps, eax
```

Vulnérabilité

Nous arrivons maintenant au vif du sujet. En regardant attentivement, nous pouvons constater que la routine win32k!GdiProcessCallout comporte 2 erreurs.

En effet, pour pouvoir obtenir l'handle de l'objet section à partir de son adresse, l'API ObOpenObjectByPointer est utilisé. L'handle obtenu n'a pas d'attribut. Ce qui donne un handle avec aucune restriction d'utilisation. Cette handle peut donc être utilisé par une application utilisateur. De plus, il a été créé pour représenter une Section sans restriction (SECTION_ALL_ACCESS).

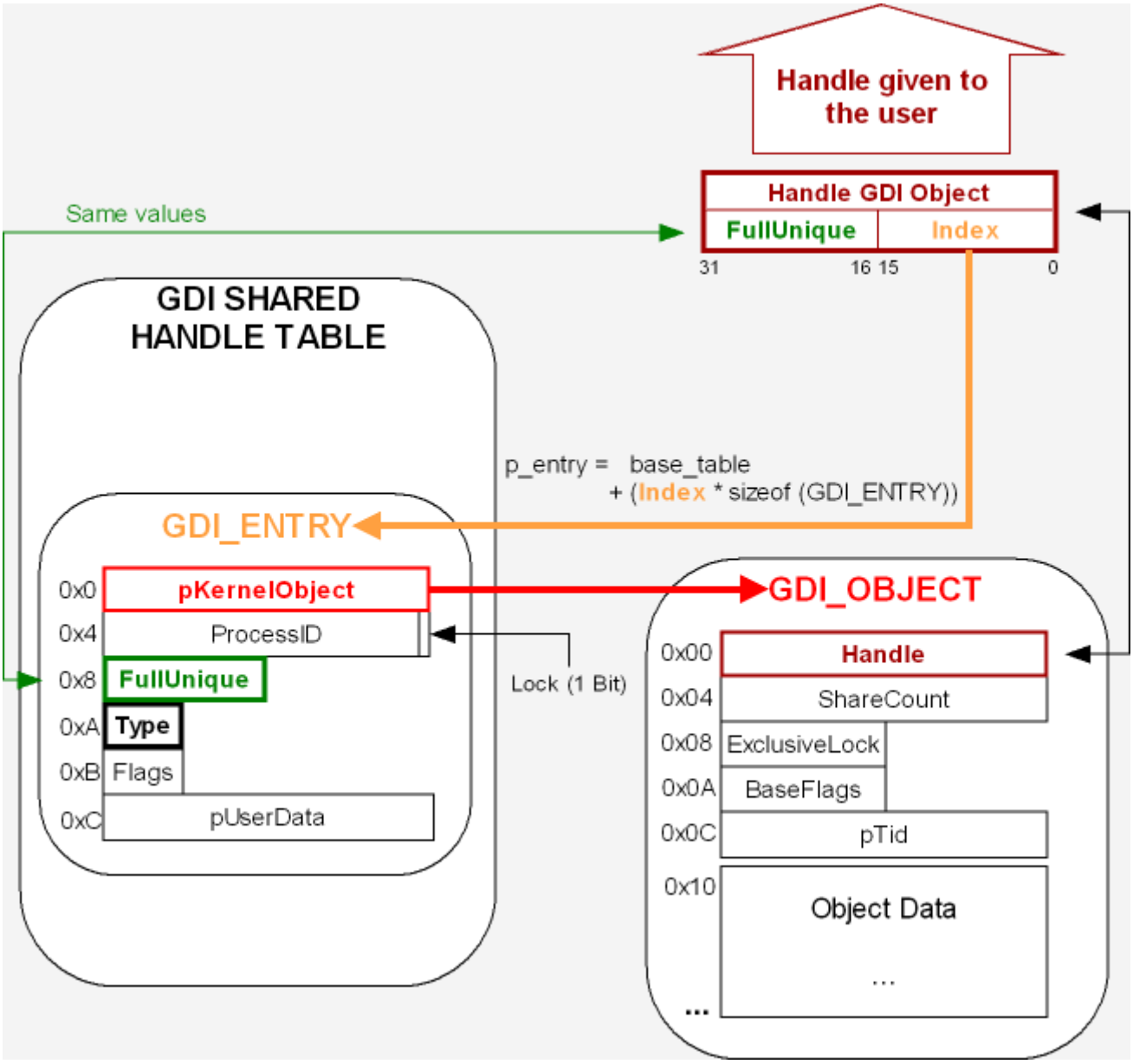
A ce stade là, nous voyons déjà une erreur flagrante. L'handle est créé pour être utilisé seulement par le noyau. Il devrait donc être créé avec comme attribut OBJ_KERNEL_HANDLE afin de ne pas pouvoir être utilisé par une application utilisateur.
Dans l'absolu, ceci n'est pas bien grave. Encore faut-il que cet handle soit correctement fermé à la fin de son utilisation.
Et.... ce n'est pas le cas !!!

Cette deuxième erreur, quand à elle, rend les choses critiques puisqu'elle laisse cet handle à disposition de l'application utilisateur.
En arrivant à déduire cet handle, l'application peut alors se re-mapper toute la section en lecture et en écriture (READWRITE) puisque l'handle permet un accès sans restriction à la section (SECTION_ALL_ACCESS).

Exploitation

Maintenant que nous avons un environnement de travail, il nous reste à répondre à une question de taille : Est-il possible d'exploiter cet handle afin de contrôler des actions du système ?
Pour répondre tout de suite à cette question, je dirais que nous pouvons faire le pire avec cette vulnérabilité. C'est à dire que nous pouvons faire en sorte d'exécuter du code arbitraire en Ring0.

Pour comprendre comment exploiter cette faille, rien de tel qu'un petit schéma :



Ce schéma montre les relations entre la table en mémoire partagée et les objets GDI du noyau.

Détails de la structure interne d'un handle GDI :

```
          typedef struct _GDIHandleBitFields // 5 elements, 0x4 bytes (sizeof)
          {
/*0x000*/     ULONG32      Index : 16;       //  0 BitPosition
/*0x000*/     ULONG32      Type : 5;         // 16 BitPosition
/*0x000*/     ULONG32      AltType : 2;      // 21 BitPosition
/*0x000*/     ULONG32      Stock : 1;        // 23 BitPosition
/*0x000*/     ULONG32      Unique : 8;       // 24 BitPosition
          }GDIHandleBitFields, *PGDIHandleBitFields, **PPGDIHandleBitFields;
```

❖ GDI_ENTRY

```
          typedef struct _GDI_ENTRY              // 6 elements, 0x10 bytes (sizeof)
          {
/*0x000*/     struct _BASEOBJECT* pobj;
              struct                             // 2 elements, 0x4 bytes (sizeof)
              {
/*0x004*/         ULONG32      Lock : 1;         // 0 BitPosition
/*0x004*/         ULONG32      Pid_Shifted : 31; // 1 BitPosition
              }Share;
/*0x008*/     WORD         FullUnique;
/*0x00A*/     BYTE         Objt;
/*0x00B*/     BYTE         Flags;
/*0x00C*/     PVOID        pUser;
          }GDI_ENTRY, *PGDI_ENTRY, **PPGDI_ENTRY;
```

Pour accéder à un objet GDI_ENTRY, il faut deux informations : L'adresse de base de la table et l'handle de l'objet GDI recherché. Cet handle correspond à celui renvoyé lors de la création d'un objet par une application user. Le Word le plus bas de l'handle correspond à un index et le word le plus haut à un identifiant unique de l'objet GDI.

Pour accéder à une entrée à partir d'un handle, il suffit donc d'appliquer ceci :

pointer_gdi_entry = address_table + (((handle_obj_gdi) & 0xFFFF) * sizeof (GDI_ENTRY));

❖ GDI_OBJECT

Un objet GDI commence toujours par la même structure de base suivie de ses propres informations.

```
          typedef struct _BASEOBJECT       // 5 elements, 0x10 bytes (sizeof)
          {
/*0x000*/     PVOID               hHmgr;
/*0x004*/     ULONG32             ulShareCount;
/*0x008*/     WORD                cExclusiveLock;
/*0x00A*/     WORD                BaseFlags;
/*0x00C*/     struct _W32THREAD*  Tid;
          }BASEOBJECT, *PBASEOBJECT, **PPBASEOBJECT;
```

Un GDI_ENTRY contient, entre autre, 2 pointeurs ; un pointeur vers l'objet GDI dans la mémoire du noyau et un pointeur vers des données dans la mémoire de l'application propriétaire.
On accède à un objet GDI simplement comme ceci :

pointer_obj_gdi = ((PGDI_ENTRY)gdi_entry)->pobj;

Pour l'instant, nous savons que nous avons la possibilité de mapper la table des GDI_ENTRY en mémoire READWRITE. Ce qui nous donne le pouvoir de modifier n'importe quelle donnée contenue dans la table. Mais nous ne pouvons pas modifier d'objet GDI puisqu'ils se trouvent en mémoire système. Ils sont donc inaccessibles à une simple application utilisateur.
En toute logique, je me suis concentré, dans mes recherches, sur l'étude des objets GDI pouvant être pointés par le champ **GDI_ENTRY.pobj**. Le but étant bien sûr de trouver un objet pouvant être usurpé et utilisé à partir d'une application utilisateur afin d'exécuter du code en mode noyau.
Le système de script de windbg m'a permis de créer des fonctions qui formatent un minimum cette environnement non documenté.

Les objets GDI peuvent être de types complètement différents. En voici la liste :

```
lkd> dt win32k!GDIObjType
GDIObjType_DEF_TYPE = 0
GDIObjType_DC_TYPE = 1
GDIObjType_UNUSED1_TYPE = 2
GDIObjType_UNUSED2_TYPE = 3
GDIObjType_RGN_TYPE = 4
GDIObjType_SURF_TYPE = 5
GDIObjType_CLIENTOBJ_TYPE = 6
GDIObjType_PATH_TYPE = 7
GDIObjType_PAL_TYPE = 8
GDIObjType_ICMLCS_TYPE = 9
GDIObjType_LFONT_TYPE = 10
GDIObjType_RFONT_TYPE = 11
GDIObjType_PFE_TYPE = 12
GDIObjType_PFT_TYPE = 13
GDIObjType_ICMCXF_TYPE = 14
GDIObjType_SPRITE_TYPE = 15
GDIObjType_BRUSH_TYPE = 16
GDIObjType_UMPD_TYPE = 17
GDIObjType_UNUSED4_TYPE = 18
GDIObjType_SPACE_TYPE = 19
GDIObjType_UNUSED5_TYPE = 20
GDIObjType_META_TYPE = 21
GDIObjType_EFSTATE_TYPE = 22
GDIObjType_BMFD_TYPE = 23
GDIObjType_VTFD_TYPE = 24
GDIObjType_TTFD_TYPE = 25
GDIObjType_RC_TYPE = 26
GDIObjType_TEMP_TYPE = 27
GDIObjType_DRVOBJ_TYPE = 28
GDIObjType_DCIOBJ_TYPE = 29
GDIObjType_SPOOL_TYPE = 30
GDIObjType_MAX_TYPE = 30
GDIObjTypeTotal = 31
```

Il y a sûrement beaucoup de choses à faire avec ces objets que je n'ai, bien sûr, pas tous étudiés. Pour ma part, je me suis très vite focalisé sur l'objet Palette. Cet objet est un objet très intéressant puisqu'il contient une interface avec 2 pointeurs de fonctions statiques de win32k.sys.

```
        typedef struct PAL_OBJECT
        {
/*0x00*/    BASEOBJECT baseobj;
/*0x10*/    ...
/*0x14*/    WORD palNumEntries;
/*0x16*/    ...
/*0x3C*/    PVOID Methode_1;
/*0x40*/    PVOID Methode_2;
/*0x44*/    ...
        } PAL_OBJECT, *PPAL_OBJECT;
```

Les objets Palettes que l'on rencontre peuvent utiliser plusieurs paires de routines en fonction de leur nature. Mais si le champ palNumEntries a une valeur supérieure à 0 l'interface sera toujours ainsi :

(palNumEntries > 0)

Methode_1 : win32k!ulIndexedGetNearestFromPalentry
Methode_2 : win32k!ulIndexedGetMatchFromPalentry

Cette Interface fournit des services statiques pour la gestion de la Palette qui permettent aux applications de traduire des valeurs de couleur. Ce point est vraiment très intéressant puisque ces routines sont directement appelées à partir des API utilisateur !

Par exemple si l'on crée une simple Palette avec une seule entrée (LOGPALETTE.palNumEntries = 1).
HANDLE hPal = CreatePalette(pLogPal);

Le système crée en interne un Objet GDI avec l'interface d'écrite ci-dessus.

Ensuite, si l'on appelle la routine utilisateur **gd32!GetNearestPaletteIndex** afin d'obtenir une correspondance de couleur, le système ira chercher dans l'interface de la Palette correspondante la routine **win32k!ulIndexedGetNearestFromPalentry**.

dgi32!GetNearestPaletteIndex (hPal, 0) -> appel de win32k!ulIndexedGetNearestFromPalentry

Avec une Palette fictive contenant une Interface donnant des routines choisies arbitrairement, il serait donc possible d'exécuter du code en Ring0.

Pour s'assurer de la validité de cette hypothèse, il nous faut regarder ce qui se passe entre l'appel de **dgi32!GetNearestPaletteIndex** et **win32k!ulIndexedGetNearestFromPalentry**. Seulement après ces vérifications, nous pourrons déduire comment construire une Palette fictive qui sera acceptée par le système.
Côté gdi32.dll, aucune vérification n'est faite. La routine **dgi32!GetNearestPaletteIndex** appelle directement sa grande sœur chez **win32k.sys** par l'intermédiaire des services système.

-> dgi32!GetNearestPaletteIndex()
-> win32k!NtGdiGetNearestPaletteIndex()
-> Pal_Obj.Methode_1()

```
BF94AE9A ; __stdcall NtGdiGetNearestPaletteIndex(x, x)
BF94AE9A _NtGdiGetNearestPaletteIndex@8 proc near

BF94AE9A mov edi, edi
BF94AE9C push ebp
BF94AE9D mov ebp, esp
BF94AE9F push esi
BF94AEA0 push edi
BF94AEA1 push [ebp+p_GDI_Pal_Object]
BF94AEA4 lea ecx, [ebp+p_GDI_Pal_Object]
BF94AEA7 call ??0EPALOBJ@@QAE@PAUHPALETTE__@@@Z ; ShareCheckLock (bit Lock of ProcessID)
BF94AEAC mov esi, [ebp+p_GDI_Pal_Object]
BF94AEAF test esi, esi
BF94AEB1 jz short loc_BF94AEE4

BF94AEB3 mov eax, [esi+GDI_PAL_OBJECT.palNumEntries]
BF94AEB6 test eax, eax
BF94AEB8 mov edi, [ebp+Color_Ref]
BF94AEBB jz short loc_BF94AEEE   ; GDI_PAL_OBJECT.palNumEntries must be > 0

BF94AEBD test edi, 1000000h      ;
BF94AEC3 jz short loc_BF94AED3   ; Color_Ref must be 24Bits color.

BF94AEC5 and edi, 0FFFFh
BF94AECB cmp edi, eax
BF94AECD jb short loc_BF94AEEE

BF94AECF xor edi, edi
BF94AED1 jmp short loc_BF94AEEE

BF94AED3 ; ---------------------------------------------------------------------------
loc_BF94AED3:
BF94AED3 mov edx, edi
BF94AED5 and edx, 0FFFFFFh
BF94AEDB mov ecx, esi
BF94AEDD call [esi+GDI_PAL_OBJECT.Methode_1] ; <--- possible hook !!!
BF94AEE0 mov edi, eax
BF94AEE2 jmp short loc_BF94AEEE
BF94AEE4 ; ---------------------------------------------------------------------------

loc_BF94AEE4:
BF94AEE4 push 6
BF94AEE6 call _EngSetLastError@4 ; EngSetLastError(x)
BF94AEEB or edi, 0FFFFFFFFh
BF94AEEE
BF94AEEE loc_BF94AEEE:
BF94AEEE test esi, esi
BF94AEF0 jz short loc_BF94AEF9
BF94AEF2 mov ecx, esi
BF94AEF4 call @HmgDecrementShareReferenceCount@4 ; HmgDecrementShareReferenceCount(x)
BF94AEF9
BF94AEF9 loc_BF94AEF9:    ; CODE XREF: NtGdiGetNearestPaletteIndex(x,x)+56 j
BF94AEF9 mov eax, edi
BF94AEFB pop edi
BF94AEFC pop esi
BF94AEFD pop ebp
BF94AEFE retn 8
BF94AEFE _NtGdiGetNearestPaletteIndex@8 endp
```

Nous pouvons constater (avec joie) qu'il ne faut vraiment pas grand chose pour mettre en place un objet Palette fictif valide afin de forcer le système à appeler la Methode_1.
Avec un GDI_PAL_OBJECT.palNumEntries>0 et une Color_Ref au format 24Bits, nous obtenons ce que nous voulons. :)

Nous avons maintenant tout pour construire cet exploit avec succès.

Les étapes à suivre sont donc les suivantes :

- Retrouver l'handle de la Section Partagée
- Mapper une vue de celle-ci en READWRITE
- Créer une Palette (API CreatePalette)
- Créer un GDI_PAL_OBJECT fictif en y plaçant l'handle de la palette originale, un palNumEntries>0 et un pointeur d'interface qui correspond à une fonction que l'on veut exécuter en Ring0.
- Changer dans le GDI_ENTRY correspondant le champ pobj afin qu'il pointe vers notre GDI_PAL_OBJECT fictif.
- Appeler gdi32!GetNearestPaletteIndex -> Exécute du code arbitraire en Ring0

Cette dernière action aura pour effet d'appeler notre routine personnelle en Ring0.
Pour finaliser l'action, il ne reste plus qu'à restaurer le champ **pobj** du GDI_ENTRY correspondant et tout redeviendra normal.

Cependant, il reste en suspend une question importante : Y a-t-il un risque que la méthode de l'interface soit appelée par une autre application ? Car dans ce cas, le système s'effondrerait puisque le GDI_PAL_OBJECT fictif n'est mappé que dans la mémoire virtuelle de notre application.

Et bien, il y a un risque effectivement car les Palettes sont des objets qui peuvent agir sur l'apparence de toutes les fenêtres présentes. Cependant, une Palette simplement créée, n'a aucun effet et reste un simple objet. Pour la rendre active, il faut avant tout la sélectionner (API SelectPalette) et la "réaliser" à un contexte de périphérique (API RealizePalette). Il est donc peu probable qu'une autre application manipule notre Palette fictive.

Pour finir, il est à noter que cette vulnérabilité ne touche que les systèmes Windows 2000 et XP ; Windows 2003 Server n'étant pas affectée.

Je me suis focalisé sur l'objet Palette mais bien d'autres objets restent mystérieux et doivent permettre pas mal de fantaisies.

Voici une petite illustration du concept à télécharger :

Exploit MS07-017 (développé en C) : http://www.laboskopia.com/download/MS07-017_Exploit.zip

note : Cet exploit a été réalisé dans un but exclusivement pédagogique.

Informations complémentaires :

http://research.eeye.com/html/alerts/zeroday/20061106.html
http://projects.info-pull.com/mokb/MOKB-06-11-2006.html
http://www.blackhat.com/presentations/bh-europe-07/Eriksson-Janmar/Whitepaper/bh-eu-07-eriksson-WP.pdf
http://www.ivanlef0u.tuxfamily.org/?p=41

Microsoft Security Advisory (935423) : http://www.microsoft.com/technet/security/advisory/935423.mspx
Microsoft Security Bulletin MS07-017 : http://www.microsoft.com/technet/security/bulletin/MS07-017.mspx

Merci d'avoir lu ce papier jusqu'au bout. :)

That's all folks !
22 avril 2007

Lionel d'Hauenens - www.laboskopia.com -