Heap de Windows : structure, fonctionnement et exploitation

Introduction

Force est de remarquer que les vulnérabilités des systèmes d'exploitation Microsoft sont nombreuses (et de plus en plus recherchées) et leur exploitation a souvent des conséquences désastreuses pour la sécurité d'un réseau ou la santé d'Internet en général. Il est tout de même nécessaire de mitiger ce constat par une remarque simple : la très grande majorité des vers ayant ciblé dernièrement certaines versions de Windows reposait sur une vulnérabilité de type débordement de buffer dans la pile ("stack overflow"), citons par exemple Blaster (MS03-026), Slammer (MS02-061), Sasser (MS04-011), Witty.

Qu'en est-il de la seconde vulnérabilite RPC/DCOM (MS03-039), de la vulnérabilité du Windows Messenger Service (MS03-043), la vulnérabilité ASN.1 (MS04-007) ? Certes, des programmes permettant de compromettre des systèmes non mis à jour existent, mais leur fiabilité est douteuse : ils nécessitent en général une connaissance au service pack près (voire au patch) des systèmes à attaquer, dépendent de la régionalisation de l'OS et par conséquent peuvent difficilement être utilisés dans le cadre d'outils automatisés de compromission ou de virus. La raison ? Ces vulnérabilités sont de type débordement de buffer dans le tas ("heap overflow"), un domaine largement parcouru dans le monde UNIX, mais qui reste encore très obscur en ce qui concerne Windows.

Quelques papiers ont été publiés couvrant les techniques de base utilisables lorsqu'est impliqué le heap de Windows **[LITCHFIELD] [AITEL] [HALVAR]** et les connaissances à en tirer sont somme toute limitées. Cependant, durant la conférence CanSecWest 2004 **[RUIU]**, une analyse poussée du fonctionnement du heap de Windows a été présentée par Oded Horovitz et Matthew Conover **[HOROVITZ]**, ainsi que des façons possibles d'exploiter de manière fiable des "heap overflows" sous Windows de façon indépendante du Service Pack ou de la régionalisation (elle justifiait à elle seule d'être présent à la conférence...).

Les implications sont nombreuses, puisque s'ouvre aux plus motivés un pan encore peu exploré de l'exploitation de certains types de débordement de tampon sous Windows. Cet article, très largement fondé sur le travail de Horovitz et Conover, développera des aspects de la gestion des tas de Windows [**NOTE** : les plateformes concernées sont les Windows à noyau NT : Windows NT 4.0, Windows 2000, Windows XP (hors SP2), de légères différences pouvant apparaître d'un environnement à un autre - Windows XP SP2 et Windows 2003 ont des protections supplémentaires], et des moyens de les détourner pour aboutir à des exploitations fiables de "heap overflows".

La structure des tas sous Windows

ATTENTION : en mode debug, les tas ont une structure différente ; pour appliquer correctement ce qui est présenté dans cet article, il vous faudra attacher des processus en cours d'exécution, et non les exécuter directement sous un debugger. Je n'aborderai pas les différences en question, libre au lecteur de creuser ces aspects.

Tas et processus

Tout processus voit coexister dans son espace mémoire un ou plusieurs tas qui accueilleront les données allouées dynamiquement au cours de son exécution. Deux fonctions exportées par la bibliothèque `kernel32.dll` permettent d'obtenir les handles (correspondant aussi à l'adresse de base) des différents tas du processus :

- `GetProcessHeap()` renvoie le handle du tas par défaut ;
- `GetProcessHeaps()` renvoie un tableau de handles de tas dont le premier sera toujours le tas par défaut du processus.

En regardant un peu plus en détail leur contenu et celui de `RtlGetProcessHeaps()` dans `ntdll.dll`, on notera que les dites fonctions se contentent de lire des pointeurs et entiers situés dans le Process Environnement Block. Le PEB est une structure spécifique au processus courant (voir encart ou **[NTDLL]**), gérée par le système d'exploitation, et contenant des variables nécessaires à son exécution. Parmi les éléments pertinents, le handle du tas par défaut situé à

l'offset 0x18 du PEB, le nombre de tas à l'offset 0x88, l'adresse du tableau des tas existants à l'offset 0x90.

Ceci peut être illustré par les quelques lignes de C suivantes :

```
int i;
PDWORD PPEB = (PDWORD)0x7ffdf000; // Adresse statique du PEB
HANDLE ProcessHeap = (HANDLE)(*(PPEB + 0x18/4)); // Tas par défaut du processus
DWORD NumberOfHeaps = *(PPEB + 0x88/4); // Nombre de tas
PHANDLE ProcessHeapsListBuffer = (PHANDLE)(*(PPEB + 0x90/4)); // Tableau des tas du processus
fprintf(stdout, "ProcessHeap :%08x\nNumberOfHeaps : %d\nProcessHeapsListBuffer : %08x\nHeaps : ",
    ProcessHeap,NumberOfHeaps, ProcessHeapsListBuffer);
for (i = 0; i < NumberOfHeaps; i++)
    fprintf(stdout, "%08x ", *(ProcessHeapsListBuffer + i));
fprintf(stdout, "\n");
```

Exemple d'exécution :

```
ProcessHeap :00140000
NumberOfHeaps : 3
ProcessHeapsListBuffer : 77fb5a80
Heaps : 00140000 00240000 00250000
```

A titre de rappel, l'adresse du PEB, fixée à 0x7ffdf000 dans les versions de Windows étudiées, se retrouve via les instructions assembleur :

```
mov     eax, large fs:18h
mov     eax, [eax+30h]
```

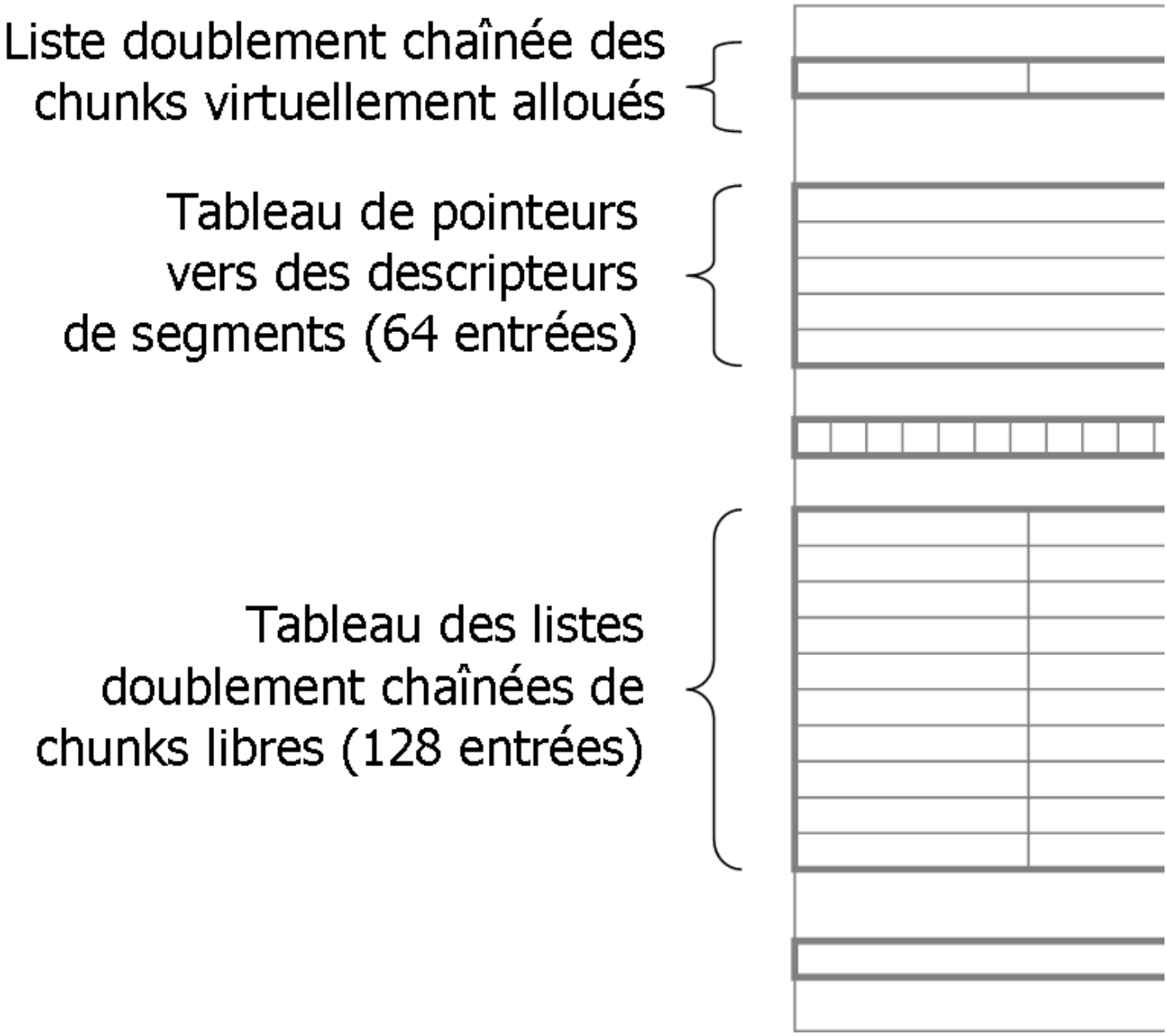
ou bien par exemple grâce à la fonction `RtlGetCurrentPeb()` exportée par `ntdll.dll` sous Windows XP et ultérieur.

Il est possible de créer de nouveaux tas dans le contexte du processus courant via l'API `HeapCreate()` de `kernel32.dll`, les variables du PEB étant alors modifiées en conséquence, et bien évidemment de détruire un tas via l'API `HeapDestroy()`.

Structure du tas

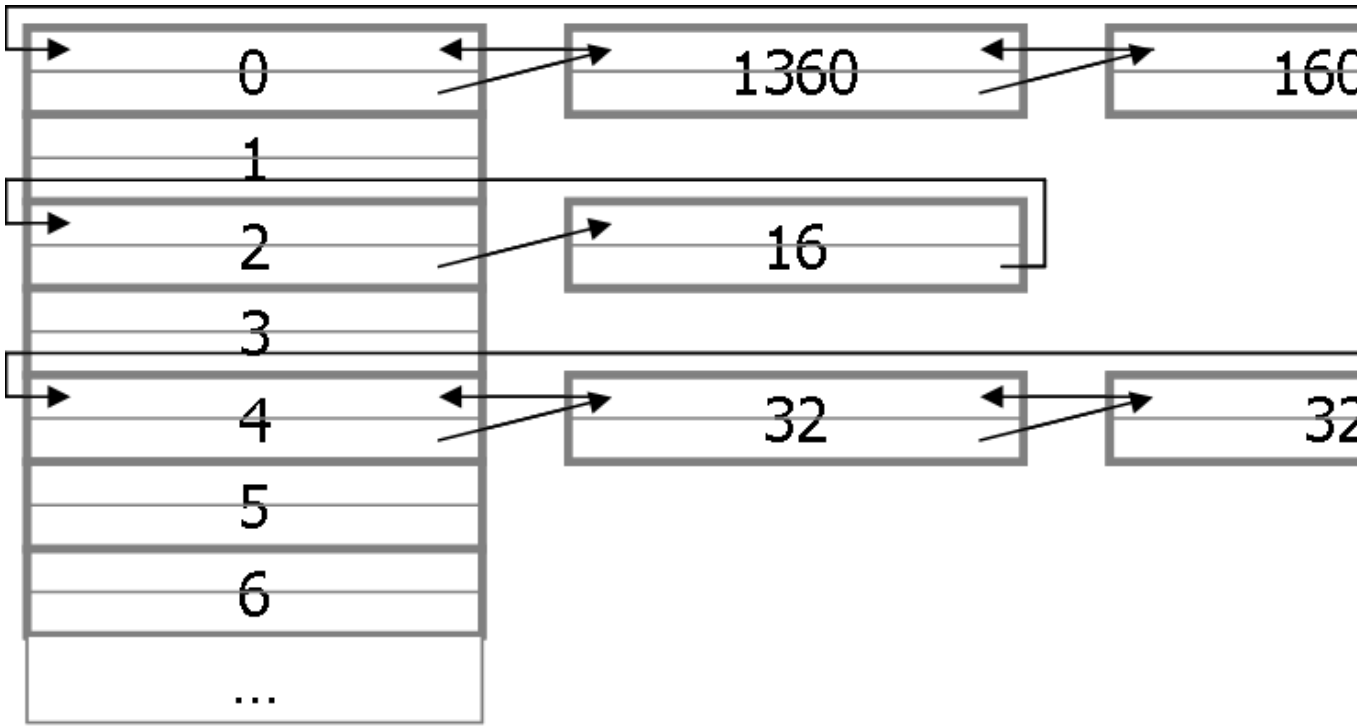
Lors de la création d'un tas, un gros segment (64 * 4096 octets par défaut) [**NOTE** : comprenez des zones de mémoire virtuelle contigues destinées à être utilisées par le tas] est tout d'abord réservé via l'API `ZwAllocateVirtualMemory()` et seulement une petite partie (4096 octets par défaut) est allouée. C'est dans cette dernière que sera stockée la structure de gestion du tas, comprenant de nombreuses variables nécessaires aux allocations, accès et libérations de mémoire. L'ensemble n'étant pas documenté, il est assez complexe d'en établir un tableau exhaustif suite à une analyse superficielle des bibliothèques dynamiques de Windows, néanmoins, certains de ces éléments (dont l'organisation est présentée en Figure 1) méritent que l'on s'attarde sur eux.

Figure 1 : Structure de gestion du tas



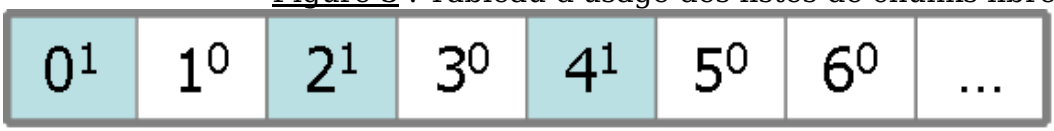
- la liste des chunks virtuellement alloués : lorsque la taille d'allocation demandée est supérieure à un seuil déterminé (d'autres conditions peuvent ou doivent être respectées mais je ne m'attarderai pas là-dessus), un nouvel espace mémoire est virtuellement alloué afin de satisfaire la demande, celui-ci étant ajouté par la suite à la liste en question ; il s'agit d'une liste dynamique doublement chainée.
- le tableau des segments du tas : d'une taille fixe (64 entrées), il regroupe les informations nécessaires à la gestion de chaque segment : adresse de base, nombre de pages, première entrée dans le tas, dernière entrée dans le tas, etc.
- le tableau des listes de chunks libres : 128 listes doublement chainées référençant les chunks libres disponibles ; la taille des chunks en octets dans chaque liste correspond à l'indice de la liste dans le tableau multiplié par 8, à l'exception de l'indice 0 qui contient une liste de chunks libres dont la taille est supérieure ou égale à 1024, et strictement inférieure au seuil d'allocation virtuelle, triée du plus petit chunk au plus gros (Figure 2).

Figure 2 : Tableau des listes doublement chaînées de chunks libres



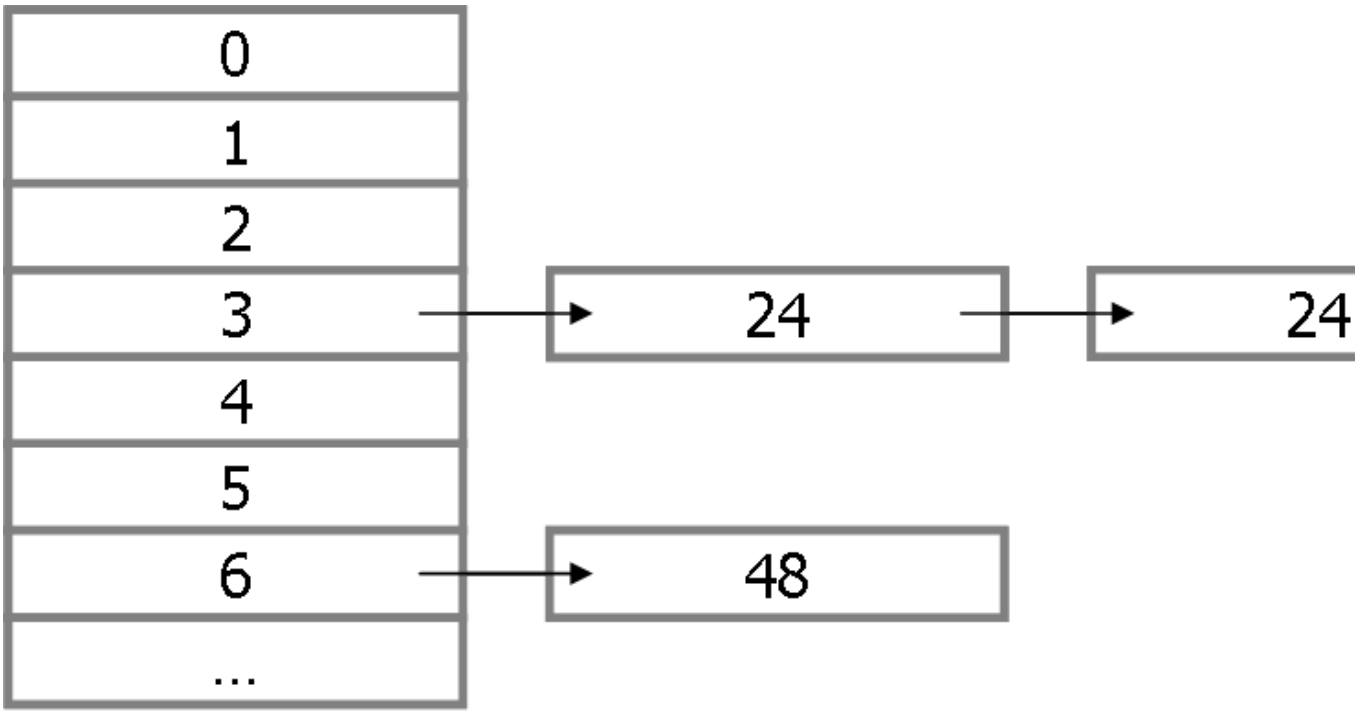
- le tableau d'indication d'usage des listes de chunks libres : d'une taille de 128 bits, il implémente un moyen rapide de déterminer si des chunks libres sont disponibles dans les listes précédemment explicitées pour la taille désirée (Figure 3) ; lorsqu'un bit du tableau est positionné à 1, cela signifie que la liste correspondante est non vide.

Figure 3 : Tableau d'usage des listes de chunks libres



- la table de "lookaside" : élément des plus intéressants (qui présente l'inconvénient de ne pas être créé dans tous les cas), elle contient des listes simplement chainées de chunks "occupés", à savoir récemment libérés et pouvant être réalloués immédiatement. Le tout est optimisé pour un accès rapide (la profondeur est limitée) et représente la méthode privilégiée de libération et d'allocation mémoire comme sera détaillé par la suite (Figure 4).

Figure 4 : Tableau de "lookaside"

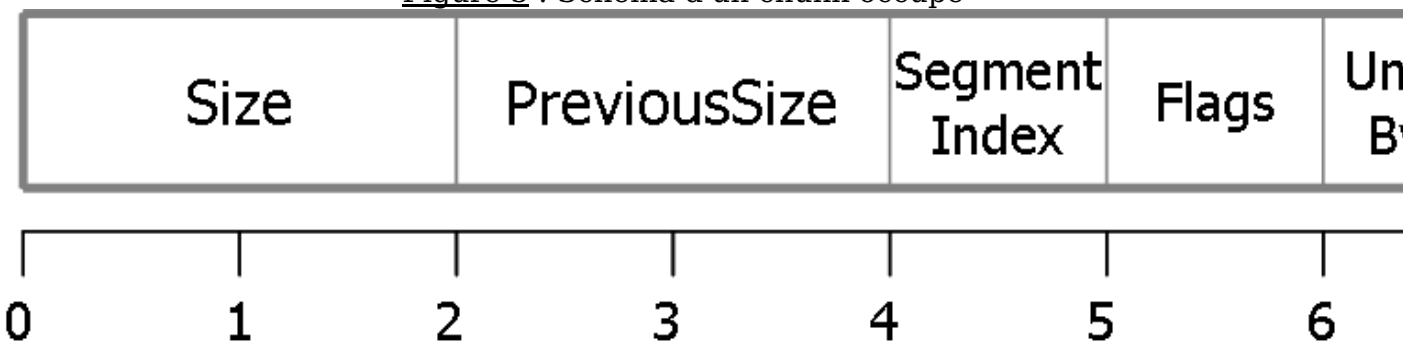


Structure d'un chunk

Les éléments essentiels de gestion du tas maintenant connu, il convient de regarder plus en détail ce qui est à la base des opérations d'allocation et de libération de mémoire : le "chunk". Il se présente sous deux formes (on met de côté le cas des chunks alloués virtuellement), l'une correspondant à un un morceau de mémoire alloué, l'autre à un morceau de mémoire libéré, la différence résidant dans la présence de deux pointeurs dans la seconde forme, pour des tailles respectives de 8 et 16 octets.

Chunk alloué

Figure 5 : Schéma d'un chunk occupé



Descriptif des champs :

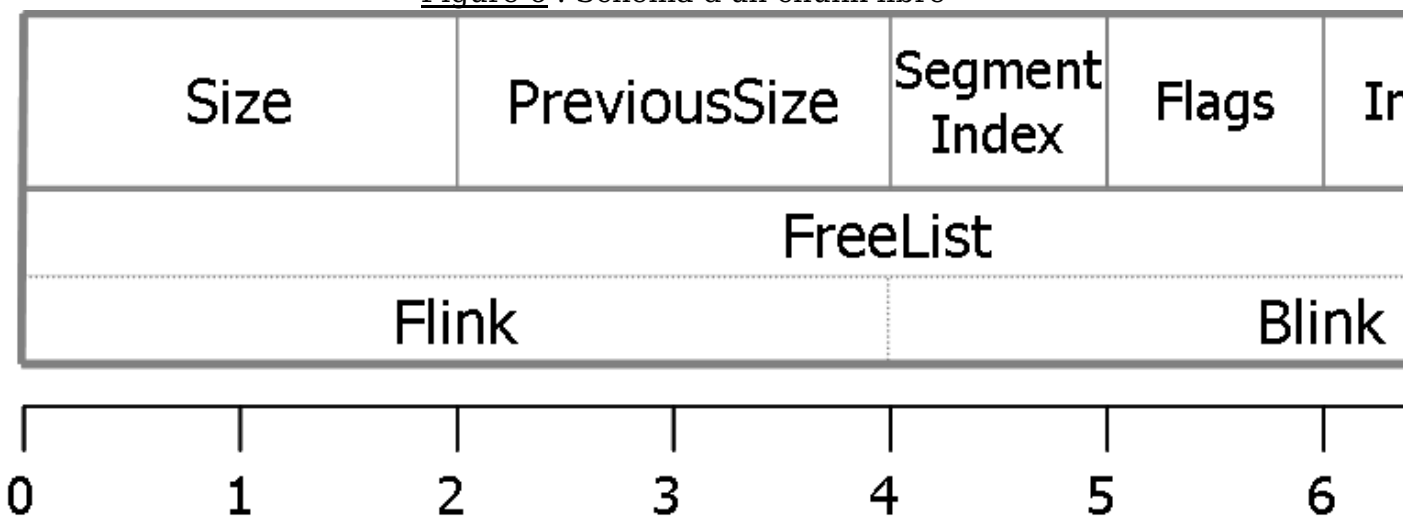
- "Size" - USHORT, 2 octets : taille du chunk courant. Elle correspond à un nombre de cellules de 8 octets occupées par le chunk. Lors de la demande d'allocation, la taille demandée est arrondie à un multiple de 8 (supérieur ou égal), auquel s'ajoutent les 8 octets necessaires au stockage des champs du chunk. La taille stockée ici répond donc à l'opération `(tailledemandee + 7) >> 3 + 1`, soit 65 pour 512 octets, 18 pour 129, etc ;
- "PreviousSize" - USHORT, 2 octets : taille du chunk précédant, possède les mêmes caractéristiques que "Self size" ;
- "SegmentIndex" - UCHAR, 1 octet : indice du segment auquel appartient le chunk, en référence au tableau des segments du tas ;
- "Flags" - UCHAR, 1 octet : indicateurs des propriétés du chunk, chaque bit positionné à 1 de cet octet faisant état d'une caractéristique particulière du chunk, ils peuvent être trouvés sur le web (les plus utiles seront détaillés par la suite) :

    - 0x01, HEAP_ENTRY_BUSY
    - 0x02, HEAP_ENTRY_EXTRA_PRESENT
    - 0x04, HEAP_ENTRY_FILL_PATTERN
    - 0x08, HEAP_ENTRY_VIRTUAL_ALLOC
    - 0x10, HEAP_ENTRY_LAST_ENTRY
    - 0x20, HEAP_ENTRY_SETTABLE_FLAG1
    - 0x40, HEAP_ENTRY_SETTABLE_FLAG2
    - 0x80, HEAP_ENTRY_SETTABLE_FLAG3
- "UnusedBytes" - UCHAR, 1 octet : nombre d'octets non utilisés dans le buffer alloué, égal à la taille allouée à laquelle est soustraite la taille demandée - information d'une utilité très relative ;
- "SmallTagIndex" - UCHAR, 1 octet : uniquement utilisé en mode debug.

Chunk libre

Dans le cas d'un chunk libre (Figure 6), le début de la structure est rigoureusement identique à celle d'un chunk occupé ; vient s'ajouter à celle-ci un couple de pointeurs vers le chunk libre suivant et un pointeur vers le chunk libre précédant.
Ce couple est utilisé dans le cadre des listes doublement chainées précédemment évoquées.

Figure 6 : Schéma d'un chunk libre



Pratiquement

L'extrait de programme suivant vous permettra de visualiser les champs présentés dans la Figure 5 et 6 (vous pouvez vous faire les structures qui vont bien) :

```
HANDLE hHeap;
unsigned char *buffer;
hHeap = HeapCreate(0, 4096, 65536); // Création d'un tas
fprintf(stdout, "*** HeapAlloc ***\n");
buffer = HeapAlloc(hHeap, HEAP_ZERO_MEMORY, 512); // Allocation de 512 octets sur ce tas
fprintf(stdout, "Self size : %d\nPrevious size : %d\nSegment index : %d\nFlags : %d\nUnused bytes : %d\nTag ind
    *(unsigned short int *)(&buffer[-8]), *(unsigned short int *)(&buffer[-6]),
    buffer[-4], buffer[-3], buffer[-2], buffer[-1]);
fprintf(stdout, "*** HeapFree ***\n");
HeapFree(hHeap, 0, buffer); // Libération du tampon alloué
fprintf(stdout, "Self size : %d\nPrevious size : %d\nSegment index : %d\nFlags : %d\nUnused bytes : %d\nTag ind
    "Next chunk : 0x%08x\nPrevious chunk : 0x%08x\n",
    *(unsigned short int *)(&buffer[-8]), *(unsigned short int *)(&buffer[-6]),
    buffer[-4], buffer[-3], buffer[-2], buffer[-1],
    *(unsigned long int *)(&buffer[0]), *(unsigned long int *)(&buffer[4]));
HeapDestroy(hHeap); // Destruction du tas
```

Exemple d'exécution :

```
*** HeapAlloc ***
Self size : 65
Previous size : 8
Segment index : 0
Flags : 1
Unused bytes : 8
Tag index : 0
*** HeapFree ***
```

```
Self size : 304
Previous size : 8
Segment index : 0
Flags : 16
Unused bytes : 8
Tag index : 0
Next chunk : 0x00320178
Previous chunk : 0x00320178
```

Algorithmes d'allocation et de libération mémoire

Les quelques structures nécessaires à la compréhension du fonctionnement du tas de Windows ayant été introduites, nous allons maintenant nous attaquer aux algorithmes d'allocation et de libération de la mémoire. Ces algorithmes sont bien évidemment loin d'être simples (surtout lorsque étudiés en assembleur...) et il serait très long de les développer en détail ici. Néanmoins, une compréhension minutieuse de ces derniers est absolument nécessaire afin d'appréhender les nouvelles techniques d'exploitation de "heap overflows" sous Windows.

Algorithme d'allocation mémoire

L'allocation d'un espace mémoire sur le tas de Windows se déroule en suivant les étapes ci-après :

1. La taille du buffer demandée est ajustée suivant l'opération décrite plus haut.
2. Si le tas comporte un tableau de "lookaside", que ce dernier n'est pas verrouillé, et que la taille demandée est inférieure à 1024 octets (afin de ne pas dépasser les limites du tableau), alors on regarde dans ce tableau si un buffer de taille exacte est disponible. Si tel est le cas, on le retire du tableau et on le retourne au demandeur, sinon, on poursuit.
3. Si la taille demandée est inférieure à 1024, on peut regarder dans le tableau des listes de chunks libres (sans tenir compte de l'entrée 0) :
    1. Si la liste correspondant à la taille exacte du buffer n'est pas vide, on enlève le premier chunk disponible de cette liste et on le retourne à l'utilisateur ;
    2. Sinon, on utilise le tableau d'usage des listes de chunks libres afin de trouver la première liste de chunks libres (de taille supérieure à celle demandée bien entendu) non vide. Si une telle liste est trouvée, on en extrait le premier élément pour le retourner au demandeur. Si la taille du chunk retourné dépasse de plus de 8 octets (strictement) la taille demandée - ce qui est des plus probables, on est amené à casser ce chunk en deux morceau, l'un de la taille demandée, le morceau de chunk restant étant retourné dans la liste de chunks libres correspondant à sa taille.
4. Si la taille demandée est inférieure au seuil d'allocation virtuelle, on peut alors chercher dans l'entrée 0 du tableau des listes des chunks libre. Dans le cas où aucun chunk ne satisfait à notre requête, le tas est étendu (je ne vais pas m'étendre ici sur la méthode) et un chunk fraichement créé est renvoyé à l'utilisateur.
5. Sinon, une allocation est effectuée (si le tas est indiqué comme étant extensible), le chunk résultant est ajouté à la liste des chunks virtuellement alloués

Algorithme de libération mémoire

La libération d'un espace mémoire alloué sur le tas de Windows observe le cheminement suivant :

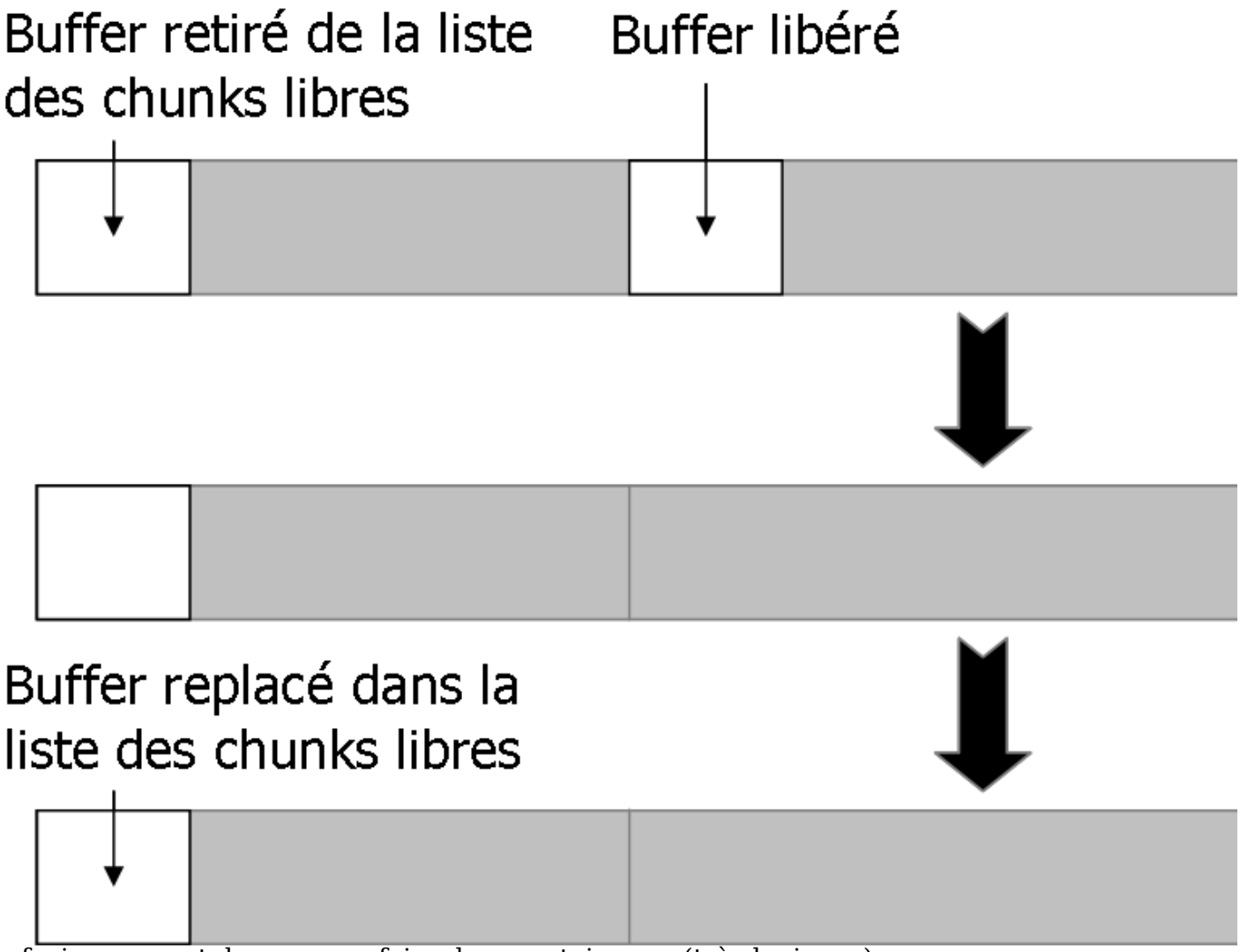
1. Si le chunk n'est pas indiqué comme étant occupé (flag "Busy"), que l'adresse du buffer n'est pas un multiple de 8, ou que le "Segment index" du chunk est supérieur ou égal à 64 (le maximum autorisé), on sort en indiquant une erreur dans les paramètres fournis.
2. Si le tas comporte un tableau de "lookaside", qu'il n'est pas verrouillé, que le chunk n'est pas indiqué comme étant virtuellement alloué (flag "Vitrual alloc"), que la taille du chunk est inférieure à 1024, alors on tente de libérer le chunk vers le tableau de "lookaside". Cela consiste à laisser le chunk marqué occupé, et à rajouter une entrée dans le tableau si ce dernier n'est pas plein.
3. Si le chunk n'est pas virtuellement alloué, on le fusionne éventuellement avec des chunks libres situés avant et/ou après (procédé développé après), et le chunk résultant ira s'ajouter à la liste des chunks libres correspondant à sa taille.
4. Sinon, on retire le chunk en question de la liste des chunks virtuellement alloués, et on libère la mémoire anciennement occupée vers le système d'exploitation.

Fusion de chunks libres

Le principe de fusion des chunks libres est assez intuitif. Puisque les chunks libres sont régulièrement séparés, le tas pourrait être amené à se retrouver dans un état très fragmenté rendant son utilisation peu efficace. Ainsi, lorsqu'un chunk est libéré, Windows va vérifier la présence de chunks libres avant et après celui-ci afin de les fusionner en un seul et même chunk, rétablissant par là même une certaine homogénéité dans le tas. La Figure 7 montre comment la libération d'un buffer situé entre deux chunks libres va entraîner la fusion des trois buffers entre eux pour qu'il n'en reste qu'un au final (procédé illustré sur la Figure 7) :

1. On détermine l'emplacement du chunk précédant (grâce au champ PreviousSize), si ce chunk existe, qu'il n'est pas signalé comme étant occupé (flag "Busy"), et que la somme des tailles des deux chunks ne dépasse pas la taille maximale d'un chunk (0xfe00 octets), alors on peut les fusionner :
    1. retrait du chunk précédant de sa liste de chunks disponibles ;
    2. retrait du chunk courant de sa liste de chunks disponibles ;
    3. mise à jour de l'information de dernière entrée du chunk précédant si le chunk courant en était doté (flag "Last entry") ;
    4. mise à jour de la taille du chunk précédant pour refléter la fusion, et éventuellement le champ PreviousSize du chunk suivant s'il existe (absence du flag "Last entry").
2. On détermine ensuite l'emplacement du chunk suivant le chunk fusioné (grâce au champ Size), si ce chunk existe, n'est pas signalé comme étant occupé, et que la somme des tailles des deux chunks ne dépasse pas la taille maximale d'un chunk, alors on peut les fusionner :
    1. si ce n'a pas déjà été fait, on retire le chunk courant de sa liste de chunks disponibles ;
    2. retrait du chunk suivant de sa liste de chunks disponibles ;
    3. éventuelle mise à jour l'information de dernière entrée ;
    4. mise à jour de la taille du chunk courant, et éventuellement du champ PreviousSize du chunk situé après le chunk fusionné s'il existe.
3. Le chunk résultant de la fusion est alors retourné.

Figure 7 : Exemple de fusion



La fusion ne peut donc pas se faire dans certains cas (très logiques) :

- lorsque le chunk est indiqué comme ne pouvant fusionner (flag "Don't coalesce", 0x80) ;
- lorsque le chunk est le premier, il ne peut fusionner avec un chunk situé avant ;
- lorsqu'il est le dernier, il ne peut fusionner avec un chunk situé après ;
- les chunks précédant ou suivant sont indiqués comme étant occupés (par exemple s'ils ont été libérés vers le tableau de "lookaside") ;
- lorsque la taille du chunk résultant de la fusion dépasse le seuil d'allocation virtuelle.

Si la quantité de mémoire libérée dépasse un certain seuil, elle pourra être directement libérée vers le système d'exploitation plutôt que d'être introduite dans les listes de chunks libres.

Pour résumer...

Les structures du tas particulièrement intéressantes sont le tableau de "lookaside", les listes de chunks libres, la liste de chunks libres d'indice 0, le tableau des segments, le tableau d'usage des listes de chunks libres.
Les structures impliquées dans l'allocation et la libération de mémoire le sont toujours dans le même ordre : le tableau de "lookaside", les listes de chunks libres, la liste de chunks libres d'indice 0.
Enfin, la mémoire du tas est entièrement recyclable : les chunks sont cassés pour satisfaire aux demandes d'allocation, et fusionnés suite à leur libération.

Les Heap Overflows

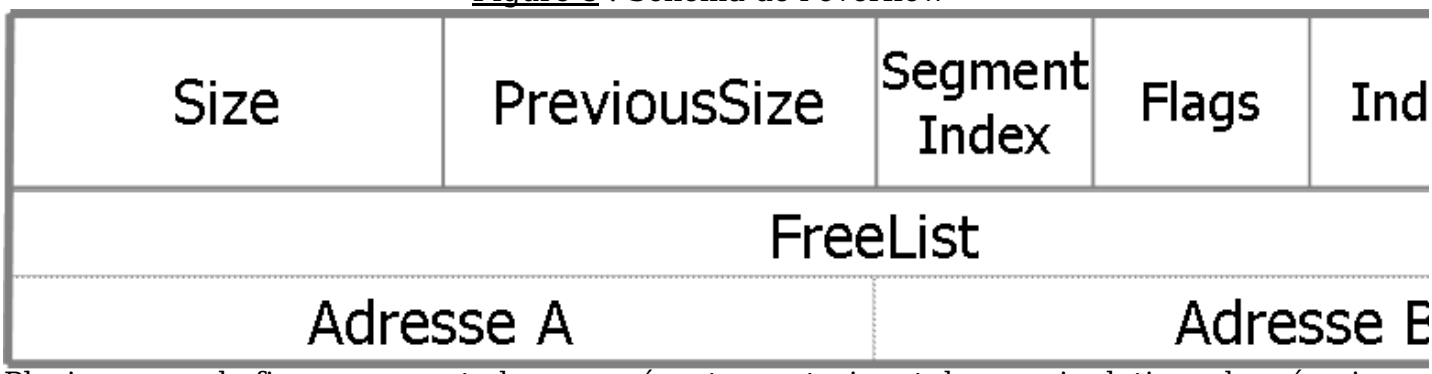
Le principe

Contrairement aux stack overflows qui donnent - en général - un contrôle immédiat du registre EIP et donc du flux d'exécution, les possibilités offertes par un débordement de tampon dans le tas de Windows sont plus "limitées". Les outils d'exploitation de "heap overflows" et articles circulant sur le sujet aboutissent tous à un résultat similaire dit **4 byte overwrite**, qui correspond au fait d'écrire 4 octets à n'importe quel emplacement mémoire.Cela nous permet d'écraser une adresse par une autre et potentiellement aboutir à de l'exécution de code lorsqu'on sait ce que l'on fait.

Comment est-ce possible ?

Vous l'aurez certainement compris, la réponse se situe au niveau des deux pointeurs situés dans la structure d'un chunk libre. Lors d'un débordement de tampon dans le tas, nous allons dans un premier temps remplir l'espace normalement alloué pour le buffer - jusqu'ici tout va bien - puis nous allons écraser la structure du chunk situé immédiatement après le notre dans le tas, quel que soit son état. En conséquence, nous prenons le contrôle de sa taille, de la taille du chunk précédant (à savoir le notre), de l'indice de son segment, de ses options, de l'index et du masque, et des adresses A et B des chunks libres suivant et précédant (voir Figure 8) : même si le chunk sur lequel nous débordons était un chunk alloué, peu nous importe, puisque nous pouvons en faire un chunk libre :)

Figure 8 : Schéma de l'overflow



Plusieurs cas de figure peuvent alors se présenter, entrainant des manipulations de mémoire au niveau des adresses A et B. Je vais maintenant entrer dans les détails pour l'un de ces cas.

Le couple "Adresse A", "Adresse B" est en fait une structure de type LIST_ENTRY qui s'inscrit dans le contexte de gestion des listes doublement chaînées par Windows :

```
typedef struct _LIST_ENTRY {
    struct _LIST_ENTRY *Flink;
    struct _LIST_ENTRY *Blink;
} LIST_ENTRY,*PLIST_ENTRY;
```

Lorsque l'on désire enlever un élément d'une liste doublement chaînée, il est nécessaire de mettre à jour le pointeur Flink (élément suivant) de l'élément qui le précède dans la liste, ainsi que le pointeur Blink (élément précédant) de l'élément qui lui succède. Cela ressemble grosso modo aux quelques lignes de codes ci-dessous, en supposant lorsque l'on désire retirer l'élément 'e' de la liste :

```
PLIST_ENTRY b;
PLIST_ENTRY f;
f = e->Flink;
b = e->Blink;
b->Flink = f;
f->Blink = b;
```

Puisque `e->Flink = AdresseA` et que `e->Blink = AdresseB`, on en déduit que le retrait de 'e' correspond aux opérations :

```
*(AdresseB) = AdresseA
*(AdresseA + 4) = AdresseB
```

Cela s'applique relativement bien au cas présent : lorsque notre chunk reconstruit devra être retiré d'une liste, les 4 octets de `AdresseA` seront écrits à l'emplacement pointé par `AdresseB`, et les 4 octets de `AdresseB` seront écrits à l'emplacement pointé par (`AdresseA + 4`), et nous avons un des cas menant à un double **4 byte overwrite** ! Rappelez-vous des opérations pouvant

mener au retrait de notre faux chunk d'une liste :

- allocation de notre chunk s'il est référencé comme libre ;
- fusion de notre chunk avec un chunk en train d'être libéré qui le suit ou le précéde immédiatement ;
- libération de notre faux chunk si ce dernier est indiqué comm virtuellement alloué...

Que de possibilités à exploiter. Mais attention, en fonction du cas de figure que vous désirez exploiter, il faudra bien évidemment adapter le contenu des différents champs du chunk que vous construisez via le débordement. Certes le couple d'adresse constituant le champ FreeList est d'importance, mais les 4 premiers champs de la structure le sont tout autant pour aboutir à une exploitation correcte. Il est notamment indispensable que les champs de taille de l'enregistrement soient corrects lors de la fusion de blocs, que le champ d'indice de segment soit inférieur au nombre maximal possible de segments (64), et que les options du chunk soient en concordance avec la situation (présence du flag "Last entry" on non, présence du flag "Busy" on non, présence du flag "Virtually allocated" ou non).

Les couples populaires

Nous sommes maintenant plus ou moins en mesure d'écrire 4 octets à un emplacement mémoire déterminé, il s'en suit une question évidente : comment choisir ce couple de nombre de 32 bits pour prendre le contrôle du flux d'exécution du processus vulnérable ? Ce problème n'est pas des moindres puisqu'il a empêché jusqu'à présent toute exploitation fiable et portable de "heap overflows" sous Windows, un mauvais choix du couple entrainera au choix une violation d'accès mémoire, une corruption de données, un crash du programme.
Plusieurs couples d'adresses sont déjà connus et utilisés couramment dans des programmes publics d'exploitation de "heap overflows" sous Windows :

Adresse B	Adresse A	Commentaire
Unhandled Exception Filter	call [edi + XX] ou équivalent	Bon taux de succès mais dépendant du SP
Vectored Exception Handler	Emplacement sur la pile pointant vers notre buffer	Bon taux de succès mais dépendant du SP
Fonctions de verrouillage du PEB	Adresse devinée ou spécifique à l'application	Taux de réussite moyen mais indépendant du SP

Développons cela :

- Le "Unhandled Exception Filter" correspond à l'adresse d'un jeu d'instructions exécutées par le système d'exploitation lorsqu'une exception se déclenche et n'est pas gérée. Il est défini grâce à la fonction `SetUnhandledExceptionFilter()` exportée par la bibliothèque `kernel32.dll` :

```
.text:77E5E5A1                mov     ecx, [esp+arg_0]
.text:77E5E5A5                mov     eax, dword_77EB73B4
.text:77E5E5AA                mov     dword_77EB73B4, ecx
```

Ici, l'adresse du UEF est 0x77EB73B4. Lorsque le noyau prend en charge une exception non gérée, il exécute la portion de code suivante, située dans `UnhandledExceptionFilter()` :

```
.text:77E73114                mov     eax, dword_77EB73B4 ; POINTEUR VERS L'UEF
.text:77E73119                cmp     eax, esi
.text:77E7311B                jz      short loc_77E73132
.text:77E7311D                push    edi
.text:77E7311E                call    eax ; APPEL A NOTRE CODE
```

Le registre EDI est poussé sur la pile, et à 0x78 octets de EDI se situe une adresse pointant vers le tas - à un endroit que l'on peut contrôler dans le cas d'un débordement. Le principe est donc de remplacer le UEF par l'adresse d'un `call [edi+0x78]` ou équivalent et de provoquer une exception afin de mener à l'exécution de notre code. L'exception est en général assez facile à déclencher, il suffit de choisir Adresse A dans une page mémoire n'offrant pas les droits en écriture, le second **4 byte overwrite** provoquant alors une violation d'accès. On peut aboutir à des résultats similaires avec des adresses relatives aux registres EBP ou ESI dans certaines versions de Windows.
L'inconvénient ? Le UEF et l'adresse du call dépendent étroitement de la version de l'OS, de son SP et de sa régionalisation. A moins de trouver la version exacte (quasiment au

patch près) du système vulnérable, l'exploitation a de grandes chances d'échouer.
- Le "Vectored Exception Handler" est une des nouveautés de XP. Contrairement aux structures usuelles d'administration des gestionnaires d'exception situées sur la pile, le VEH se trouve dans le tas. Un pointeur vers le premier VEH est situé à une adresse fixe dans la bibliothèque `ntdll.dll` :

```
.text:77F5CD0E                mov     esi, dword_77FB4880 ; POINTEUR VERS LE VEH
.text:77F5CD14                jmp     short loc_77F5CD24
.text:77F5CD16
.text:77F5CD16 loc_77F5CD16:                              ; CODE XREF: sub_77F5CCDE+48
.text:77F5CD16                lea     eax, [ebp+var_8]
.text:77F5CD19                push    eax
.text:77F5CD1A                call    dword ptr [esi+8] ; APPEL A NOTRE CODE
.text:77F5CD1D                cmp     eax, 0FFFFFFFFh
.text:77F5CD20                jz      short loc_77F5CD3C
.text:77F5CD22                mov     esi, [esi]
.text:77F5CD24
.text:77F5CD24 loc_77F5CD24:                              ; CODE XREF: sub_77F5CCDE+36
.text:77F5CD24                cmp     esi, edi
.text:77F5CD26                jnz     short loc_77F5CD16
```

  La technique consiste à remplacer ce dernier par un pointeur vers une fausse structure de VEH que nous avons construite sur la pile. Lorsqu'une exception se produira, nous gagnerons le contrôle du flux d'exécution via le pointeur situé en `[esi+8]`.
  L'inconvénient ? L'adresse du VEH est spécifique aux versions de Windows - encore une fois au SP près, l'adresse sur la pile n'est pas forcément fiable.
- Il existe deux pointeurs, situés dans le PEB, qui référencent deux fonctions. Ces deux fonctions sont appelées dans `RtlAcquirePebLock()` et `RtlReleasePebLock()` au sein de `ntdll.dll`, elles même susceptible d'être appelées à différentes reprises. Dans la mesure où l'adresse du PEB est fixe dans les versions de Windows considérées, les deux pointeurs vers les fonctions sont aussi situées à des emplacements fixes (PEB+0x20 pour le premier, PEB+0x24 pour le second, se référer à l'encart ou à **[NTDLL]**), et représentent donc une cible de choix à écraser par une adresse pouvant aboutir à notre code.
  L'inconvénient ? Le saut vers le pointeur écrasé se fera à un moment où il n'existera plus de référence à notre code aisément atteignable (à savoir pas moyen d'accéder à notre buffer grâce à une adresse relative à un registre), il faudra donc utiliser des adresses devinées ou spécifiques à l'application - ce qui nuit fortement à la fiabilité de l'exercice.

Au final, pas de méthode miracle, et des débordements assez complexes à exploiter de façon convenable. Y a-t-il un moyen d'améliorer cela ?

Le tableau de "lookaside" : le saint graal de l'exploitation des "heap overflows" sous Windows

Nous avons vu précédemment que le tableau de "lookaside", lorsqu'il existe, est la première option retenue pour le choix de chunks lors de l'allocation et la libération d'espace mémoire de taille réduite (à savoir inférieure à 1024 octets). Approfondissons cela.

Ses caractéristiques

Le tableau de "lookaside" est créé à la fin de la fonction `RtlCreateHeap()` de `ntdll.dll` si les conditions suivantes sont réunies :

1. le tas supporte les accès sérialisés (absence de l'indicateur HEAP_NO_SERIALIZE dans les propriétés du tas) ;
2. le tas est extensible (présence de l'indicateur HEAP_GROWABLE dans les propriétés du tas) ;
3. l'utilisation de tableau de "lookaside" n'est pas désactivée ;

Le tableau est la première structure a être allouée sur le tas, d'une taille de 6144 octets : 128 fois la taille d'une entrée de "lookaside" soit 0x30 octets, et par conséquent commence toujours au même emplacement relativement à l'adresse de base du tas (à 0x688 octets de la base). Bien que l'adresse soit fixe relativement au tas, le tableau de "lookaside" est référencé par un pointeur dans la structure de gestion du tas à l'offset 0x580 : si ce pointeur est nul, le tas ne possède pas de tableau de "lookaside", s'il n'est pas nul, il pointe vers la première entrée du tableau. Un autre moyen de déterminer si un tas possède un tableau de "lookaside" est l'utilisation de la fonction `HeapQueryInformation()` apparue avec Windows XP qui renvoie 1 si le tas le supporte.

Attention, La fonction `RtlCreateHeap()` est très rarement utilisée directement pour la création d'un tas, au profit de la fonction `HeapCreate()` de `kernel32.dll`. Or cette fonction filtre l'indicateur HEAP_GROWABLE dans le champ flOptions : le seul moyen d'obtenir un tas extensible et possédant donc un tableau de "lookaside" est de spécifier une taille minimale et maximale égale à 0, la fonction s'occupera du reste. Notez que le tas par défaut d'un processus est extensible, et que le "lookaside" commence vierge de toute entrée.

Illustrons tout cela grâce au programme suivant :

```
typedef BOOL (*MYPROC)(HANDLE, HEAP_INFORMATION_CLASS, PVOID,
SIZE_T, PSIZE_T);
...
HANDLE hHeap;
MYPROC ProcAdd;
HINSTANCE hInstance;
hHeap = HeapCreate(0, 4096, 65536);
fprintf(stdout, "%08x %08x\n", hHeap, *(unsigned int *)((unsigned int)hHeap + 0x580));
hHeap = HeapCreate(0, 0, 0);
fprintf(stdout, "%08x %08x\n", hHeap, *(unsigned int *)((unsigned int)hHeap + 0x580));
hHeap = GetProcessHeap();
fprintf(stdout, "%08x %08x\n", hHeap, *(unsigned int *)((unsigned int)hHeap + 0x580));
if ((hInstance = LoadLibrary("kernel32")) == NULL)
    goto Fin;
if ((ProcAdd = (MYPROC)GetProcAddress(hInstance, "HeapQueryInformation")) == NULL) // Seulement sous XP
    goto Fin;
(ProcAdd)(hHeap, HeapCompatibilityInformation, &ulHeapInformation, sizeof(ulHeapInformation), &ReturnLength);
fprintf(stdout, "Heap base address: 0x%x\nHeap type: %u\n\n", hHeap, ulHeapInformation);
Fin:
```

Exemple d'exécution :

```
00320000 00000000
00330000 00330688
00140000 00140688
Heap base address: 0x140000
Heap type: 1
```

Dans le premier tas créé : pas de "lookaside", dans le second il y en a un, dans le tas par défaut du processus aussi. Notez que la présence de "lookaside" ou non est rarement mise en avant dans les exemples d'exploitation de "heap overflows" publiés jusqu'à présent (comme par exemple dans **[LITCHFIELD]**) alors qu'il s'agit d'un élément déterminant dans le déroulement des procédés étudiés.

Contrôle du "lookaside"

Puisque le "lookaside" est généralement présent dans le tas et que son emplacement reste fixe avec les versions de Windows, il est intéressant d'étudier les possibilités offertes par un **4 byte overwrite** dans ce tableau, ou en rapport avec ce tableau.

Si on provoque l'allocation d'un buffer de taille inférieure à 1024 octets, celui ci va être libéré vers le "lookaside" : le chunk va rester occupé et intact (sauf les 4 premiers octets mis à 0), et son adresse placée dans l'entrée correspondante du tableau de "lookaside". Puisque nous connaissons 1) l'adresse de base du tableau, 2) la taille d'une entrée et que 3) le numéro de l'entrée est déductible de la taille d'allocation demandée, nous connaissons un emplacement fixe pointant sur notre buffer !

En pratique, on calcule le numéro de l'entrée à partir de la taille ajustée du buffer à allouer i = (taille + 7) / 8 + 1, on retrouve l'adresse grâce à l'opération base + i * 0x30. En l'occurence, pour une adresse de base du tas de 0x140000 (et donc du "lookaside" de 0x140688) et une taille à allouer de 922, on retrouve le pointeur vers notre buffer après libération à l'adresse 0x140000 + 0x688 + ((922 + 7) / 8 + 1) * 0x30 = 0x140688 + 0x75 * 0x30 = 0x141C78.

En C, on illustre ça avec :

```
HANDLE hHeap;
unsigned char *buffer;
hHeap = GetProcessHeap();
fprintf(stdout, "*** HeapAlloc ***\n");
buffer = HeapAlloc(hHeap, HEAP_ZERO_MEMORY, 922); // Allocation de 922 octets
memset(buffer, 0x42, 922); // On remplit de buffer de 0x42
fprintf(stdout, "Buffer address : %08x\nBuffer content : %08x\nLookaside pointer : %08x\n", buffer, *(unsigned
fprintf(stdout, "*** HeapFree ***\n");
HeapFree(hHeap, 0, buffer); // Libération du buffer
fprintf(stdout, "Buffer address : %08x\nBuffer content : %08x\nLookaside pointer : %08x\n", buffer, *(unsigned
```

Exemple d'exécution :

```
*** HeapAlloc ***
Buffer address : 00145620
Buffer content : 42424242
Lookaside pointer : 00000000
*** HeapFree ***
Buffer address : 00145620
Buffer content : 00000000
Lookaside pointer : 00145620
```

Une fois le buffer libéré, ses 4 premiers octets sont mis à 0, et la valeur du pointeur dans le "lookaside" correspondant à la taille de 922 octets est mise à l'adresse de notre buffer. Maintenant que se passe-t-il si on redemande l'allocation d'un espace mémoire de 922 octets ? Windows va tout simplement renvoyer le pointeur se trouvant dans l'entrée correspondante du "lookaside". Imaginons que ce pointeur ait été modifié entre temps (par un **4 byte overwrite** par exemple :), l'adresse retournée par `HeapAlloc()` sera en notre contrôle et nos 922 octets seront écrits à un emplacement arbitraire ! Notre **4 byte overwrite** devient un **922 byte overwrite**, les possibilités ouvertes sont énormes...

```
HANDLE hHeap;
unsigned char *buffer;
hHeap = GetProcessHeap();
fprintf(stdout, "*** HeapAlloc ***\n");
buffer = HeapAlloc(hHeap, HEAP_ZERO_MEMORY, 922);
memset(buffer, 0x42, 922);
fprintf(stdout, "*** HeapFree ***\n");
HeapFree(hHeap, 0, buffer);
/* Simulation du 4 byte overwrite : l'adresse 0x7ffdf130 remplace le pointeur vers notre buffer dans le "lookas
   C'est équivalent à Adresse B = 0x141c78 et Adresse A = 0x7ffdf130 (ça tombe bien pour Adresse B, on a le dro
   un 0x00 terminal, et les deux adresses sont dans des pages avec droit en écriture, donc pas d'exception) */
*(unsigned int *)((unsigned int)hHeap + 0x1c78) = 0x7ffdf130;
fprintf(stdout, "*** HeapAlloc ***\n");
buffer = HeapAlloc(hHeap, HEAP_ZERO_MEMORY, 922);
memset(buffer, 0x43, 922);
fprintf(stdout, "Buffer address : %08x\nBuffer content : %08x\nLookaside pointer : %08x\n", buffer, *(unsigned
```

Exemple d'exécution :

```
*** HeapAlloc ***
*** HeapFree ***
*** HeapAlloc ***
Buffer address : 7ffdf130
Buffer content : 43434343
Lookaside pointer : 00000000
```

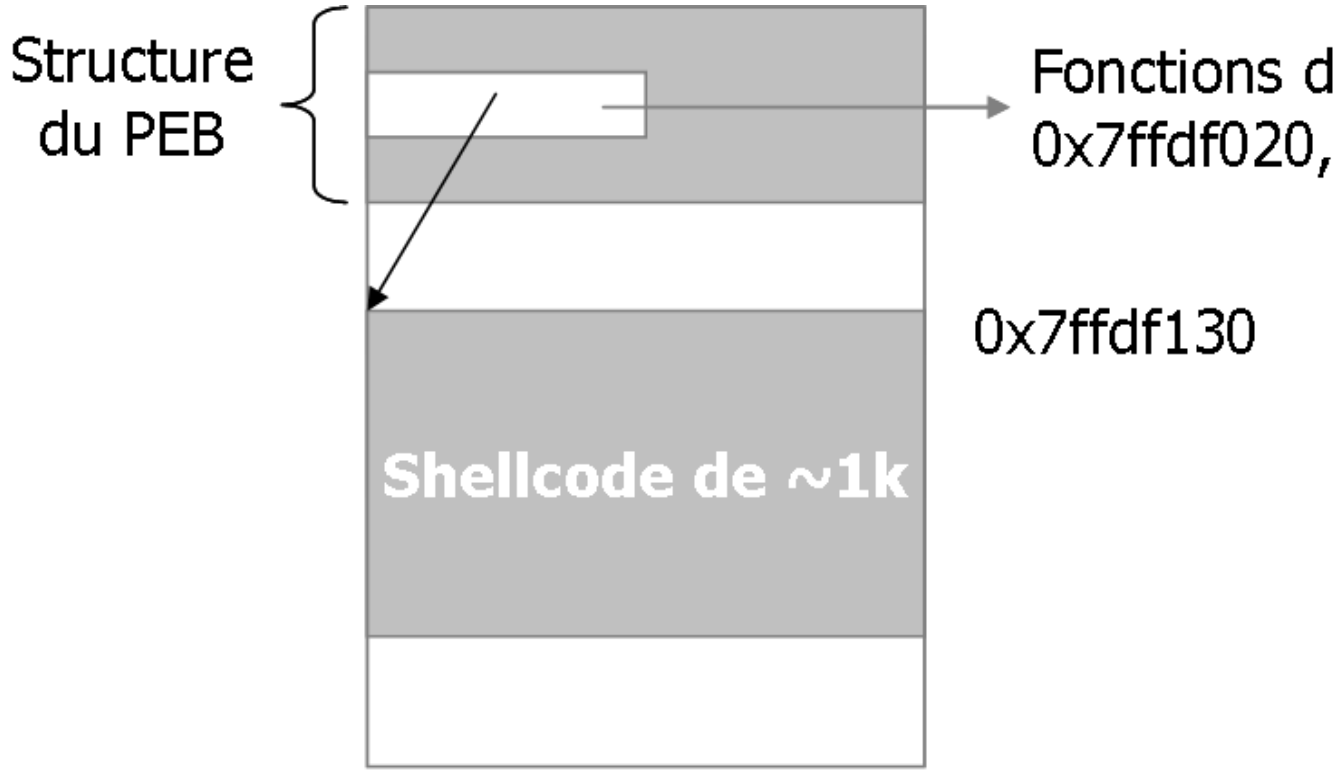
Nous venons d'écrire 922 octets dans le PEB à l'adresse 0x7ffdf130 !

Application à l'exécution de code

En se fondant sur les constats précédemment énoncés, on peut en déduire plusieures méthodes utilisant le "lookaside" sous une forme ou une autre pour provoquer l'exécution de code arbitraire de façon fiable, je m'attarderai sur quelques unes seulement :

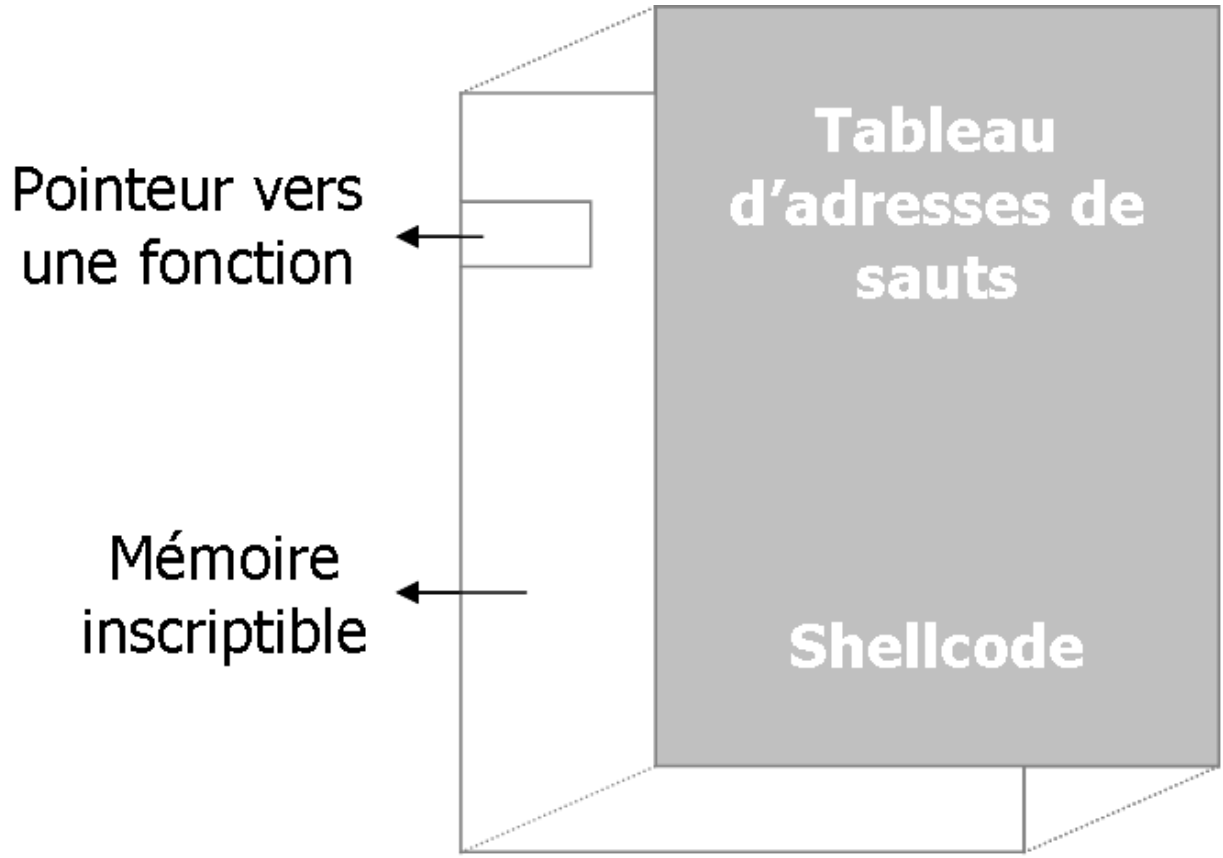
1. Le grand désavantage de l'utilisation du couple faisant intervenir les fonctions de verrouillages du PEB est l'absence quasi totale de visibilité quant à l'emplacement du shellcode en mémoire. On peut grandement améliorer cette technique grâce à l'utilisation du "lookaside", en suivant les étapes ci-après :
    1. On provoque l'allocation d'un buffer d'une taille < 1024 octets et sa libération afin de peupler l'entrée du "lookaside" correspondante ;
    2. On provoque un **4 byte overwrite** afin d'écraser le pointeur de l'entrée en question (calculable bien évidemment) par une adresse fixe (pourquoi pas 0x7ffdf130 dans le PEB ? :) ;
    3. On provoque l'allocation d'un buffer de taille identique à celui de 1. afin que notre shellcode soit écrit à l'adresse fixée précédemment ;
    4. On provoque un **4 byte overwrite** afin d'écraser l'une des deux fonctions de verrouillage du PEB (connue) par l'adresse de notre shellcode (aussi connue);
    5. On attend que soit appelée la fonction en question !

Figure 9 : Illustration de la première méthode



2. La méthode précédente est séduisante et assez intuitive, mais necéssite deux overflows à la suite, ce qui se révèle la plupart du temps hasardeux dans la mesure où le tas peut se retrouver dans des états quelque peu instables après ce type d'opération. Pour la seconde méthode il nous faudra trouver un emplacement mémoire fixe contenant une ou plusieurs adresses de fonctions susceptibles d'être appelées dans la suite du programme. Puisque nous pouvons maintenant écrire un peu moins de 1024 octets de données n'importe où, nous pouvons écraser le pointeur vers la dite fonction par une adresse pointant sur notre shellcode : si nous connaissons l'adresse du pointeur en question, notre shellcode sera quelques octets plus loin en mémoire.
    1. On provoque l'allocation d'un buffer d'une taille < 1024 octets et sa libération afin de peupler l'entrée du "lookaside" correspondante ;
    2. On provoque un **4 byte overwrite** afin d'écraser le pointeur de l'entrée en question par l'emplacement mémoire contenant le pointeur vers la fonction qui sera exécutée - cette zone doit bien sûr être inscriptible ;
    3. On provoque l'allocation d'un buffer de taille identique à celui de 1. afin que notre buffer contenant l'adresse du shellcode et notre shellcode (le tout minutieusement arrangé) soit écrit à l'adresse précédemment déterminée ;
    4. On attend que soit appelée la fonction en question !

Figure 10 : Illustration de la seconde méthode



3. La troisième méthode consiste simplement à trouver une variable globale (un chaine de caractères par exemple) que nous pourrons remplacer, par exemple celle utilisée par l'API `GetSystemDirectory()`. Cette dernière, qui renvoie normalement la chaine "c:\winnt\system32\", pourrait maintenant renvoyer "\\1.2.3.4\backdoors" :)
   Dès lors, il vous suffira de mettre à la dite adresse une bibliothèque backdoorée (`kernel32.dll`, `user32.dll`, peu importe), elle sera automatiquement chargée par les exécutables sur la machine compromise si ces derniers en importent des fonctions.

Personnellement, mon coup de coeur va à une autre technique qui consiste à déplacer le pointeur d'une "Dispatch Table" vers un endroit du "lookaside" afin que la prochaine fonction à être requise pointe en fait sur notre buffer : c'est très efficace et portable entre les SP d'une même version de Windows. La méthode diffère un peu des précédentes dans la mesure où cette fois-ci on n'écrase pas une adresse du "lookaside", mais on modifie l'adresse d'une zone mémoire vers ce dernier afin de bénéficier de l'adresse connue d'un buffer libéré contenant notre shellcode (ses 4 premiers octets ayant été mis à 0).

1. On provoque l'allocation d'un buffer d'une taille < 1024 octets (contenant notre shellcode) et sa libération. Les 4 premiers octets de ce buffer seront mis à 0 et son adresse ira dans le "lookaside" dans l'entrée associée à sa taille ;
2. On provoque un **4 byte overwrite** afin d'écraser le contenu de la User32Dispatch située à l'offset 0x2c dans le PEB (ce tableau est utilisé dans le cadre d'opération graphiques). La nouvelle valeur de cette variable pointera dans le tas de sorte que User32Dispatch + (4 * Numéro de la fonction) = Adresse du buffer dans le "lookaside". Le numéro de la fonction se trouve en analysant le flux d'exécution du programme afin de déterminer quelle sera la prochaine appelée ;
3. Il ne reste plus qu'à attendre que la fonction soit appelée, ou bien forcer le destin en provoquant son appel :)

Et cette liste ne se veut pas exhaustive...

Conclusion

Un exploit pour le Messenger Service de Windows qui fonctionne convenablement sans avoir à identifier de façon exacte la version du système d'exploitation vulnérable, vous en réviez, c'est maintenant possible (et fait). Les compromissions de systèmes Microsoft ont

encore de beaux jours devant elles, même si les techniques exposées ici ne semblent pas encore avoir fait leur chemin jusqu'aux exploits publics. Il est à mettre au crédit de Microsoft des efforts conséquents pour tenter d'arranger un peu les choses avec le Service Pack 2 de Windows XP et avec Windows 2003 Server : l'adresse du PEB est rendue plus aléatoire, un cookie est introduit dans les chunks des tas afin de se prémunir contre tout overflow, le retrait d'un élément d'une liste doublement chainée est plus sûr.

Un point important n'a pas été abordé ici, faute de temps et d'espace, il s'agit bien évidemment de la stabilisation de l'environnement d'exécution du programme. Il n'est pas rare que l'exploitation d'un "heap overflow" provoque des dysfonctionnements au niveau du processus, ou pire, du système d'exploitation ; il est nécessaire de s'assurer qu'une telle chose ne surviendra pas, notamment en reconstruisant les structures modifiées, ce qui inclut par exemple le PEB et ses fonctions de verrouillage, ou bien le tas lui même qu'il faudra réparer. Gageons que ces sujets seront abordés dans un prochain numéro de MISC !

Encore une fois, j'adresse mes salutations et mes sincères remerciements à Oded Horovitz et Matthew Conover pour leur travail sur le tas de Windows.

ENCART Structure du PEB

Voici la structure définissant le PEB telle qu'elle peut apparaître après étude des bibliothèques dynamiques de Windows. N'étant pas documentée, certains champs sont d'une exactitude toute relative (voire non définis).

```
/*
 * Process Environment Block, situé à l'adresse 0x7FFDF000.
 */
typedef struct _PEB {
/*000*/    BOOLEAN InheritedAddressSpace;
/*001*/    BOOLEAN ReadImageFileExecOptions;
/*002*/    BOOLEAN BeingDebugged;
    BYTE b003;
    DWORD d004;
/*008*/    PVOID SectionBaseAddress;
/*00C*/    PPROCESS_MODULE_INFO ProcessModuleInfo;
/*010*/    PPROCESS_PARAMETERS ProcessParameters;
/*014*/    DWORD SubSystemData;
/*018*/    HANDLE ProcessHeap;
/*01C*/    PCRITICAL_SECTION FastPebLock;
/*020*/    VOID (*AcquireFastPebLock)(PCRITICAL_SECTION);
/*024*/    VOID (*ReleaseFastPebLock)(PCRITICAL_SECTION);
    DWORD d028;
/*02C*/    PPVOID User32Dispatch;
    DWORD d030;
    DWORD d034;
    DWORD d038;
/*03C*/    DWORD TlsBitMapSize;
/*040*/    PRTL_BITMAP TlsBitMap;
/*044*/    DWORD TlsBitMapData[2];
    PVOID p04C;
    PVOID p050;
/*054*/    PTEXT_INFO TextInfo;
/*058*/    PWCHAR InitAnsiCodePageData;
/*05C*/    PWCHAR InitOemCodePageData;
/*060*/    PSHORT InitUnicodeCaseTableData;
/*064*/    DWORD KeNumberProcessors;
/*068*/    DWORD NtGlobalFlag;
    DWORD d06C;
/*070*/    LARGE_INTEGER MmCriticalSectionTimeout;
/*078*/    DWORD MmHeapSegmentReserve;
/*07C*/    DWORD MmHeapSegmentCommit;
/*080*/    DWORD MmHeapDeCommitTotalFreeThreshold;
/*084*/    DWORD MmHeapDeCommitFreeBlockThreshold;
/*088*/    DWORD NumberOfHeaps;
/*08C*/    DWORD AvailableHeaps;
/*090*/    PHANDLE ProcessHeapsListBuffer;
    DWORD d094;
    DWORD d098;
    DWORD d09C;
/*0A0*/    PCRITICAL_SECTION LoaderLock;
/*0A4*/    DWORD NtMajorVersion;
/*0A8*/    DWORD NtMinorVersion;
/*0AC*/    WORD NtBuildNumber;
/*0AE*/    WORD CmNtCSDVersion;
/*0B0*/    DWORD PlatformId;
/*0B4*/    DWORD Subsystem;
/*0B8*/    DWORD MajorSubsystemVersion;
/*0BC*/    DWORD MinorSubsystemVersion;
/*0C0*/    KAFFINITY AffinityMask;
/*0C4*/    DWORD ad0C4[35];
    PVOID p150;
    DWORD ad154[32];
```

```
/*1D4*/    HANDLE Win32WindowStation;
    DWORD d1D8;
    DWORD d1DC;
/*1E0*/    PWORD CSDVersion;
    DWORD d1E4;
} PEB, *PPEB, **PPPEB;
```

Auteur

Kostya Kortchinsky
Responsable du CERT RENATER
kostya.kortchinsky@renater.fr

Références

[LITCHFIELD] Windows Heap Overflows : http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt
[AITEL] Microsoft Heap Overflows : http://www.immunitysec.com/downloads/msrpcheap.pdf & http://www.immunitysec.com/downloads/msrpcheap2.pdf
[HALVAR] Third Generation Exploitation : http://www.blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt
[RUIU] cansecwest/core04 conference : http://www.cansecwest.com/
[HOROWITZ] Reliably Exploiting Windows Heap Overflows : http://cansecwest.com/csw04/csw04-Oded+Connover.ppt
[NTDLL] `ntdll.h` : http://cvs.kldp.net/cgi-bin/cvsweb.cgi/moguasrc/include/ntdll.h?cvsroot=mogua&rev=1.3