

Bypassing Windows Heap Protections

Nicolas Falliere
nicolas.falliere@gmail.com
Version française:
Jerome Athias
<http://www.athias.fr>

Historique

Les buffer overflows (débordements de tampon) Windows basés sur le tas (heap) peuvent être classés en deux catégories.

- 1) La première couvre les débordements pour les plateformes Windows 2000, Windows XP et Windows XP SP1.

Le code de gestion du tas pour ces systèmes, localisé dans ntdll.dll, n'effectue aucune vérification sur les chunks du tas. Quand un débordement survient, le prochain chunk adjacent peut être sur-écrit, et si de bonnes valeurs sont soumises, une opération sur le tas (alloc, free...) peut résulter en une sur-écriture de 4 octets arbitraires en mémoire. De nouvelles techniques ont vu le jour récemment, mais le principe reste le même : sur-écrire une portion spécifique de la mémoire avec des valeurs spécifiques pour obtenir le contrôle et exécuter un payload plus tard.

- 2) La seconde catégorie inclue Windows XP SP2 et Windows 2003.

Microsoft a modifié les structures du tas et les fonctions de manipulation du tas; deux vérifications sur les chunks ont été ajoutées.

La première vérification consiste à vérifier l'intégrité d'un cookie de sécurité dans l'entête du chunk, pour s'assurer qu'aucun débordement n'est survenu quand ce même chunk est alloué.

La seconde vérification, extrêmement efficace, vérifie les pointeurs de lien suivants et précédents d'un chunk libre étant délié, et ce pour n'importe quelle raison (allocation, coalescence). La même vérification est réalisée pour les blocs alloués virtuellement. D'autres protections ont été introduites, comme le PEB aléatoire, et l'encodage des pointeurs d'exception. Ces protections ont été mises en place pour diminuer le nombre de pointeurs de fonctions fixes et connus utilisés globalement par le processus.

Le premier document public détaillant une méthode pour contourner les protections du tas a été publié au début de 2005 par Alexander Anisimov.

Sa méthode consiste à exploiter les vérifications inexistantes sur la liste lookaside.

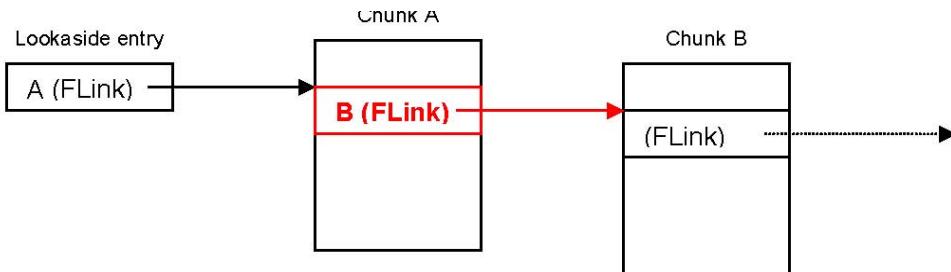
Le premier dword d'une entrée lookaside est le début d'une liste simplement chaînée de chunks, marqués comme occupés, mais prêts à être alloués. Lors d'une allocation, le premier bloc d'une liste lookaside correspondante peut être retourné : il est simplement retiré de la liste en remplaçant le pointeur de lien suivant (Flink) dans l'entrée lookaside par le pointeur Flink du nouveau bloc alloué. Ce processus est illustré en Figure1.

Cette nouvelle technique est bonne en théorie, mais semble très difficile en pratique. Les opérations de tas suivantes peuvent survenir, en fournissant de bonnes valeurs d'entrée, si nous voulons sur-écrire l'octet N :

- 1– Allocation d'un bloc de taille N (<0x3F8 octets)
- 2– Libération de ce bloc: le bloc devient référencé par la table lookaside
- 3– Le débordement survient dans un bloc adjacent précédent: nous pouvons manipuler le pointeur Flink du bloc libéré précédent
- 4– Un bloc de taille N est alloué: notre faux pointeur est écrit dans la table lookaside
- 5– Un second bloc de taille N est alloué: notre faux pointeur est retourné
- 6– Une opération de copie d'une entrée contrôlée vers ce buffer survient: ces octets sont écrits vers l'emplacement de notre choix

Comme vous pouvez le constater, ces conditions sont difficiles à mettre en pratique, surtout dans des programmes complexes. Le tas doit également avoir une table lookaside active et non bloquée pour que l'opération réussisse.

Avant l'allocation:



Après l'allocation:

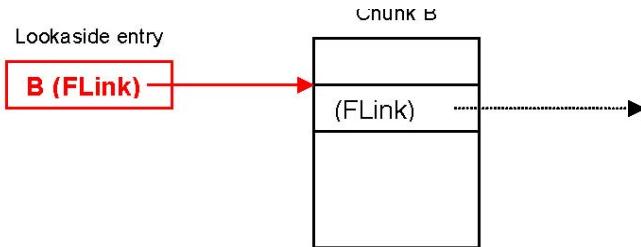


Figure 1: Allocation d'un bloc A depuis la table lookaside

Une nouvelle manière de contourner les protections du tas

La méthode que j'introduis ici n'utilise pas la sur-écriture des structures de gestion du tas pour sur-écrire 4 octets.

Le tas par défaut d'un processus, comme les autres tas créés par le système, est utilisé par beaucoup d'APIs pour enregistrer des informations concernant le processus et son environnement. Quand une DLL est chargée, sa fonction principale est exécutée (DLLMain par exemple) et souvent, les données peuvent être stockées sur le tas du processus. Que se passe-t-il si ces parties de données sont sur-écrites ?

Prenons l'exemple d'un programme basique, comme le Bloc Notes Windows (notepad).

Nous pouvons remarquer que même ce programme requiert un grand nombre de librairies dynamiques pour s'exécuter. Si nous examinons le tas par défaut, avant que le thread principal commence de s'exécuter, nous verrons qu'un bon nombre de « heap chunks » ont été alloués par ces DLLs.

Beaucoup de ces chunks ont une taille de 40 octets (incluant 8 octets pour l'entête) et possèdent la structure décrite en Figure 2:

Chunk header	
0	X
A	B
0	0
?	?

Figure 2: Un “heap chunk” de 40 octets, trouvé dans le tas par défaut d'un processus

A: Adresse de la prochaine “structure de 40 octets”

B: Adresse la précédente “ structure de 40 octets”

Il arrive que la structure pointée par X soit en fait une section critique. Quand une section critique est initialisée, une « structure associée de 40 octets », nous l'appellerons « structure de liaison », est également créée pour garder une trace de la section critique. Peu de ces structures sont localisées dans la section de données (data section) de ntdll.dll ; quand elles sont toutes utilisées, les structures de liaison sont créées dans le tas par défaut. La Figure 3 illustre la relation entre les structures de liaison et les sections critiques.

Cette liste doublement chainée nous rappelle la manière dont les chunks libres sont gérés par les routines de gestion du tas. Pendant la destruction d'une section critique, la structure de liaison associée sera retirée de sa liste. Si nous remplaçons A et B, nous devrions alors pouvoir sur-écrire une portion de 4 octets de la mémoire :

From RtlDeleteCriticalSection (ntdll.dll version 5.1.2600.2180):

```
...
mov [eax], ecx      ; eax=B
mov [ecx+4], eax   ; ecx=A
...
```

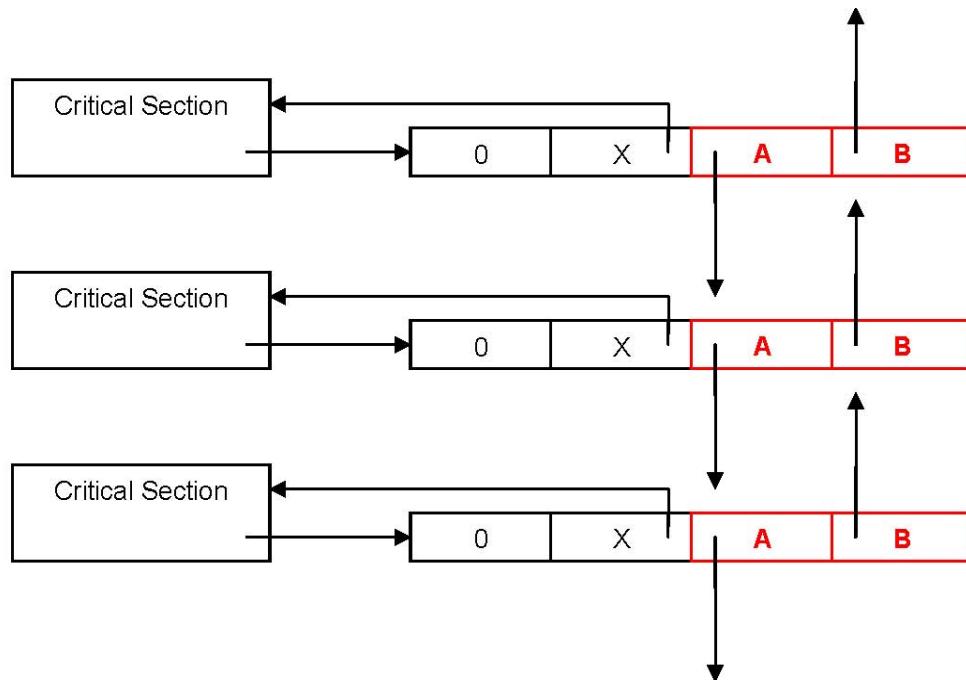


Figure 3: Sections critiques et structures de liaison

La technique fonctionne car:

- Aucune vérification n'est effectuée sur ces pointeurs précédent et suivant spécifiques.
- Les sections critiques sont détruites pendant la terminaison du processus; cela assure la réalisation de la sur-écriture.
- Les structures de liaison peuvent facilement être trouvées dans le tas par défaut; si nous contrôlons la taille du chunk dans lequel nous sur-écrivons, nous pouvons l'ajuster de manière à ce qu'une structure de liaison réside quelques octets après.

Vous trouverez un code de démonstration (PoC) de cette technique dans l'Annexe1.

Conclusion

Cette technique a été utilisée pour exploiter avec succès un heap overflow non corrigé localisé dans un utilitaire standard de Windows XP SP2.

Néanmoins, plusieurs problèmes restent partiellement résolus : alors que nous avons la possibilité de sur-écrire 4 octets de mémoire, nous devons choisir de bonnes valeurs de pointeurs précédent et suivant. L'utilisation classique de pointeurs sur les handlers d'exception (SEH) est compromise, de même que les pointeurs de fonctions globaux localisés dans le PEB.

L'exploitation de heap overflows sur les derniers systèmes Windows est donc possible, mais les problèmes sont d'améliorer leur portabilité et leur fiabilité.

Annexe 1: Proof of Concept code

```
//-----
// This code demonstrates how an overflow in critical section related
// structures stored in heap chunks can be used to produce an
// arbitrary memory overwrite
// (c) 2005 Nicolas Falliere
// nicolas.falliere gmail.com
//-----

#include <windows.h>
#include <tlhelp32.h>
#include <stdio.h>

VOID GetChunkList(DWORD *pChunks, INT *nbChunks)
{
    DWORD pid;
    HANDLE snapshot;
    HEAPLIST32 list;
    HEAPENTRY32 entry;
    BOOLEAN bNext;
    INT cnt=0;

    pid = GetCurrentProcessId();
    snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPHEAPLIST, pid);
    if(snapshot == INVALID_HANDLE_VALUE)
    {
        printf("[Error] Cannot take a heap snapshot\n");
    }
    else
    {
        ZeroMemory(&list, sizeof(list));
        list.dwSize = sizeof(HEAPLIST32);
        bNext = Heap32ListFirst(snapshot, &list);

        while(bNext)
        {
            ZeroMemory(&entry, sizeof(entry));
            entry.dwSize = sizeof(HEAPENTRY32);
            bNext = Heap32First(&entry, list.th32ProcessID, list.th32HeapID);

            while(bNext)
            {
                pChunks[cnt] = entry.dwAddress;
                cnt++;
                ZeroMemory(&entry, sizeof(entry));
                entry.dwSize = sizeof(HEAPENTRY32);
                bNext = Heap32Next(&entry);
            }
        }
    }
}
```

```

        }

        ZeroMemory(&list, sizeof(list));
        list.dwSize = sizeof(HEAPLIST32);
        bNext = Heap32ListNext(snapshot, &list);
    }

    CloseHandle(snapshot);
    *nbChunks = cnt;
}

int main(void)
{
    HANDLE hHeap;
    DWORD pChunks[500];
    INT nbChunks;
    INT i;
    HMODULE hLib;
    DWORD *p;

    hHeap = GetProcessHeap();
    printf("Default heap: %X\n", hHeap);
    hLib = LoadLibrary("oleaut32.dll");
    printf("LoadLibrary : oleaut32.dll\n");

    GetChunkList(pChunks, &nbChunks);

    for(i = 0; i < nbChunks; i++)
    {
        // Chunk size is 40 bytes
        if(*(WORD *)(pChunks[i] - 8) == 5)
        {
            p = (DWORD *)(pChunks[i]);
            // Check if FLink and BLink are there
            if(p[2] && p[3])
            {
                printf("Structure found at address: %8X\n", p);
                printf("Before modification : A=%8X B=%8X\n", p[2], p[3]);
                memcpy(p + 2, "AAAAABBBB", 8);
                printf("After modification : A=%8X B=%8X\n", p[2], p[3]);
                break;
            }
        }
    }

    printf("Press Enter to terminate the program and trigger the access violation\n");
    getchar();

    return 0;
}

```