Les Buffers Overflow

Matieu Beretti
Rodolphe Lacroix

Mars 2003

Table des matières

INTRODUCTION

En 1988 est apparue une nouvelle forme d'attaque avec le "Morris Worm" ou "vers de l'Internet", programmé par un étudiant de Cornell. Infectant des milliers de machines en utilisant deux bugs important sur les programmes "sendmail" et "fingerd" sous UNIX, les buffers overflow exploités lui permettaient de lancer un shell à distance sur les machines attaquées.

Un buffer overflow est une attaque très efficace et assez compliquée à réaliser. Elle vise à exploiter une faille, une faiblesse dans une application (type browser, logiciel de mail) pour executer un code arbitraire qui compromettra la cible (acquisition des droits administrateur par exemple).

Le fonctionnement général d'un buffer overflow consiste à faire crasher un programme en écrivant dans un buffer plus de données qu'il ne peut en contenir (un buffer est un zone mémoire temporaire utilisée par une application), dans le but d'écraser des parties du code de l'application et d'injecter des données utiles pour exploiter le crash de l'application.

Cela permet donc d'exécuter du code arbitraire sur la machine où tourne l'application vulnérable.

L'intérêt de ce type d'attaque est qu'il ne nécessite pas -le plus souvent- d'accès au système, ou dans le cas contraire, un accès restreint simplement.

Il s'agit donc d'une attaque redoutable. D'un autre côté, il reste difficile à mettre en oeuvre car il requiert des connaissances avancées en programmation ; de plus, bien que les nouvelles failles soient largement publiées sur le web, les codes ne sont pas ou peu portables. Une attaque par buffer overflow signifie en général que l'on a affaire à des attaquants doués.

Chapitre 1

Mémoire et Processus

1.1 Organisation globale

Quand un programme s'excécute, ses différents éléments (instructions, variables ...) sont gérés en mémoire de manière structurée.
Chaque programme se décompose en 3 sections principales contenant
- le code
- les datas : données initialisées et non initialisées (par exemple, une allocation avec malloc)
- la pile

Un programme se sert de la pile pour sauvegarder au cours de son exécution les variables locales ou les paramètres. La pile croit vers les adresses basses. Le registre ESP contient l'adresse de la dernière valeur empilée.

Nous disposons de registres 8, 16 et 32 bits. Les registres de 32 bits sont dénotés : EAX, EBX, ECX, EDX, EBP, ESP, EDI, ESI, EIP et EFLAGS.

Sans entrer dans les détails, les registres EAX, EBX, ECX et EDX servent aux manipulations de données, ESI et EDI sont utilisés dans les opérations sur les chaînes. ESP est le pointeur de pile et EIP pointe sur l'instruction courante. EBP permet d'adresser les variables locales et les paramètres dans les procédures. EFLAGS contient des informations sur l'état du processeur.

1.2 Appels de fonction

Les paramètres d'entrée des appels de procédures sont poussées dans la pile en commençant par le dernier jusqu'au premier. C'est à la procédure appelante de supprimer les paramètres de la pile après l'appel. Dans le cas d'une fonction, lorsqu'il n'y a qu'une seule variable en retour, la valeur de celle-ci est contenu dans EAX.

La procédures appelée sauvegarde sur la pile EBP puis affecte ESP à EBP. Elle prend ensuite la place sur la pile nécessaire à ses variables locales puis éventuellement sauve tous les registres banalisés ou au moins ceux qu'elle va manipuler. Lorsqu'elle se termine elle restaure les registres, l'ancien EBP et l'affecte à ESP et enfin effectue un ret.

Le code suivant illustre comment tout cela marche et permet de mieux comprendre les techniques invoquées pour l'utilisation de buffers overflow.

```
int toto(int a, int b, int c)
{
int i=4;
return (a+i);
}

int main(int argc, char **argv)
{
toto(0, 1, 2);
return 0;
}
```

On désassemble le binaire obtenu à l'aide de gdb.
La fonction main donne :

0x80483e4 -main- : push %ebp
0x80483e5 -main+1- : mov %esp,%ebp
0x80483e7 -main+3- : sub $0x8,%esp
C'est le prologue de la fonction main.
0x80483ea -main+6- : add $0xfffffffc,%esp
0x80483ed -main+9- : push $0x2
0x80483ef -main+11- : push $0x1
0x80483f1 -main+13- : push $0x0
0x80483f3 -main+15- : call 0x80483c0 -toto-

L'appel à la fonction toto() est faite par ces quatres instructions : ses paramètres sont empilés et la fonction est invoquée.

0x80483f8 -main+20- : add $0x10,%esp

Cette instruction représente la fonction de retour de toto() dans le main : le pointeur sur la stack pointe sur l'adresse de retour.

0x80483fb -main+23- : xor %eax,%eax
0x80483fd -main+25- : jmp
0x8048400 -main+28-
0x80483ff -main+27- : nop
0x8048400 -main+28- : leave
0x8048401 -main+29- : ret

Les deux dernières instructions sont l'étape de retour de la fonction main().

Si l'on regarde maintenant le code désassemblé de toto() :
0x80483c0 -toto- : push %ebp
0x80483c1 -toto+1- : mov %esp,%ebp
0x80483c3 -toto+3- : sub $0x18,%esp

C'est le prologue de la fonction : EBP pointe initialement sur l'environnement ; il est empilé (pour sauvegarder l'environnement courant), et la seconde instruction fait pointer EBP sur le sommet de la pile, qui contient l'adresse initiale de l'environnement. La troisième instruction alloue la mémoire nécessaire à la fonction.

0x80483c6 -toto+6- : movl $0x4,0xfffffffc(%ebp)
0x80483cd -toto+13- : mov 0x8(%ebp),%eax
0x80483d0 -toto+16- : mov 0xfffffffc(%ebp),%ecx
0x80483d3 -toto+19- : lea (%ecx,%eax,1),%edx
0x80483d6 -toto+22- : mov %edx,%eax
0x80483d8 -toto+24- : jmp 0x80483e0 -toto+32-

0x80483da -toto+26- : lea 0x0(%esi),%esi

Ce sont les instructions de la fonctions...
0x80483e0 -toto+32- : leave
0x80483e1 -toto+33- : ret

L'étape de retour est faite avec ces deux instructions. La première réinitialise les pointeurs EBP et ESP à la valeur qu'ils avaient avant le prologue. La seconde instruction via l'instruction registre sait qu'elle instruction doit être excecutée.

Cet exemple montre l'organisation de la stack à l'appel de fonctions. Si la section mémoire n'est pas manipulée avec précaution, des opportunités s'ouvrent à l'attaquant pour modifier l'organisation de la stack et excécuter du code corrompu. Cela est possible car quand une fonction retourne, l'adresse de l'instruction suivante est copiée de la stack au pointeur EIP. Si la valeur de cette adresse est donc modifiée, elle pourra alors pointer vers une zone mémoire contenant du code malveillant.

1.3 Les buffers et leur vulnérabilité

En langage C, les chaînes de caractères, ou buffers, sont représentés par un pointeur sur l'adresse de leur premier byte. La fin de chaîne est avérée lorsque l'on atteint un byte NULL. Cela signifie qu'il n'y a pas de moyen de définir précisément la mémoire allouée pour le buffer. Tout dépend du nombre de caractères qu'il contient.

Le programme ci-dessous illustre la vulnérabilité des buffers.

```
#include ¡stdio.h¿

int main(int argc, char **argv)
{
char jayce[4]="Oum" ;
char herc[8]="Gillian" ;
strcpy(herc, "BrookFlora") ;
printf("%s", jayce ) ;
return 0 ;
}
```

Deux buffers sont stockés dans la stack. Quand dix caractères sont copiés dans un buffer qui est supposé de taille huit bytes, le premier buffer est modifié. Cette copie cause un buffer overflow.

Chapitre 2

Le Stack Overflow

2.1 Son principe

Nous avons vu précédement le role du registre EIP contenant l'adresse de la prochaine instruction stockée. On peut modifier sa valeur pour le faire pointée sur du code malveillant. Cependant il n'est pas aisé de savoir où l'information est stockée. Il est plus facile d'écrire sur la totalité de la section mémoire en initialisant chaque bloc d'instruction sur l'adresse d'instructions de notre choix. Trouver l'adresse du shellcode en mémoire n'est pas facile. On veut trouver la distance entre le pointer sur la stack et le buffer en sachant approximativement où le buffer commence le programme vulnérable en mémoire. En mettant du shellcode au milieu du buffer dont le début qvec du NOP opcode (NOP est un byte d'opcode qui ne fait rien), on sait que le pointeur sur la stack contiendra approximativement le début du buffer et effectuera un jump dessus pour excecuter NOP opcode jusqu'à la fin du shellcode.

2.2 Un exemple

Considérons le programme C dont la tache toute simple consiste à lire le fichier dont le nom est passé en paramètre et à en afficher le contenu à l'écran. Ce fichier texte se compose d'une chaîne terminé par le caractère de valeur ASCII 0.

```
// vulnerable.c
#include ¡stdio.h¿

int main(int argc , char * argv[])
{
// Lit une chaine dans le fichier texte passe en 1er parametre et l'affiche à
l'écran char buf[512] , *c ;
FILE *fichier ;
if (argc¡=1)
{
printf("Veuillez preciser le nom du fichier à afficher") ;
exit() ;
}
fichier = fopen(argv[1],"r") ;
if (fichier==NULL)
{
printf("Erreur d'ouverture du fichier") ;
exit() ;
}
c = &buf[0] ;
```

```
// printf("adresse : %p",c);
// Lecture du fichier
while((*c = fgetc(fichier)) != 0)
{
c++;
}
c = 0;
//Caractère de fin de chaine fclose(fichier);
// Affichage du texte l'écran
printf("%s",buf);
}
```

Regardons ce que donne le listing de ce programme en assembleur. Il peut être obtenu par :
- gcc -s vulnerable.c ; lecture du fichier résultat.s
- as -a vulnerable.r ; lecture du fichier résultat vulnerable.l
La deuxième commande donne également les codes hexadécimaux correspondant à chaque ligne de programme assembleur.

```
.file "vulnerable.c"
.version "01.01"
gcc2_compiled. : .section.rodata.align32
.LC0 : .string"Veuillezpreciserlenomdufichieraafficher."
.LC1 : .string"r".align32
.LC2 : .string"Erreurd'ouverturedufichier"
.LC3 : .string"%s".text.align16
```

$.globlmain.typemain, function$

$main : pushl\%ebp$

$movl\%esp, \%ebp$

$subl\$536, \%esp$

$cmpl\$1, 8(\%ebp)$

$jg.L3$

$subl\$12, \%esp$

$pushl\$.LC0callprintf$

$addl\$16, \%esp$

$callexit.p2align4, , 7$

$.L3 :$

$subl\$8, \%esp$

$pushl\$.LC1$

$movl12(\%ebp), \%eax$

$addl\$4, \%eax$

$pushl(\%eax)$

$callfopen$

$addl\$16, \%esp$

$movl\%eax, \%eax$

$movl\%eax, -528(\%ebp)$

$cmpl\$0, -528(\%ebp)$

$jne.L4$

$subl\$12, \%esp$

$pushl\$.LC2callprintf$

$addl\$16, \%esp$

$callexit.p2align4, , 7$

$.L4:$

$leal -520(\%ebp), \%eax$

$movl\%eax, -524(\%ebp)$

$.p2align4, , 7$

$.L5:$

$subl\$12, \%esp$

$pushl -528(\%ebp)$

$callfgetc$

$addl\$16, \%esp$

$movl\%eax, \%edx$

$movl -524(\%ebp), \%eax$

$movb\%dl, (\%eax)$

$movb(\%eax), \%al$

$cmpb\$ -1, \%al$

$jne.L7jmp.L6.p2align4, , 7$

$.L7:$

$leal -524(\%ebp), \%eax$

$incl(\%eax)$

$jmp.L5.p2align4, , 7$

$.L6:$

$movl -524(\%ebp), \%eax$

$movb\$0, (\%eax)$

$subl\$12, \%esp$

$pushl -528(\%ebp)$

$callfclose$

$addl\$16, \%esp$

$subl\$8, \%esp$

$leal -520(\%ebp), \%eax$

$pushl\%eax$

$pushl\$.LC3callprintf$

$addl\$16, \%esp$

$movl\%ebp, \%esp$

$popl\%ebp$

$ret.Lfe1 : .sizemain, .Lfe1 - main.ident"GCC : (GNU)"$

La troisième ligne qui suit l'étiquette main nous donne la place que va réserver le compilateur dans la pile pour les variables locales de la main :

```
pushl %ebp
movl %esp, %ebp
subl $536, %esp
```

Il y a ainsi 536 octets alloués. Ils correspondent au tableau de caractères buf qui fait 512 octets, à la structure FILE de 16 octets et à 8 octets supplémentaires réservés automatiquement par gcc lorsqu'il traite la procédure main mais qui n'ont pas vraiment d'importance.
La vulnérabilité du programme apparaît aux lignes suivantes :

```
c = &buf[0] ;
while((*c = fgetc(fichier)) != 0)
{
c++ ;
}
```

Dans le cas ou le fichier fait plus de 512 caractères, le programme va continuer à lire les données et à les écrire en mémoire. Les données supplémentaires seront placées dans la pile au dessus de buf[511] et vont écraser l'EBP sauvé et l'adresse de retour. Il suffit alors de mettre en forme le fichier judicieusement de telle sorte à remplacer l'adresse de retour du main par une autre adresse. Lorsque la procédure main va effectuer son ret, le programme sera redirigé vers l'emplacement de notre choix. Si nous plaçons en outre du code exécutable dans le fichier et si l'adresse de retour pointe vers le début des données soit buf[0], nous sommes en mesure de prendre la main et d'exécuter notre propre code.

CONCLUSION

Les attaques de type stack overflow résultent d'erreur de conception et de bugs dans des applications, programmées essentiellement en C/C++. L'utilisateur et le gestionnaire de serveur devront donc, pour se protéger, consulter régulièrement les sites internet et les newsletters portant sur la sécurité. Ils devront appliquer immédiatement les mises à jour bouchant les trous de sécurités. Il faut garder à l'esprit que bien souvent, les pirates exploitent des failles connues depuis très longtemps. Au niveau du développement le meilleur conseil serait de toujours programmer des vérifications de taille de buffer.

Malheureusement, il peut toujours y avoir un oubli. Pour y faire face, il existe un compilateur nommé StackGuard ou un patch pour gcc appelé StackShield qui interdisent l'écrasement de l'adresse de retour à l'exécution du programme.