TLS-SEC

TP Sécurité du logiciel: Débordement de tampon mémoire par la pratique

---

Objectifs : Comprendre les rudiments du débordement de tampon dans la pile. Pour cela, nous allons procéder en deux étapes. La première étape (section 1) consiste simplement à modifier dans la pile l'adresse de retour d'une fonction. Cet exercice, même s'il ne constitue pas une attaque à proprement parler, est fondamental pour bien comprendre le principe du débordement de tampon. Ensuite, nous étudierons un réel débordement de tampon sur un programme vulnérable (section 2).

---

1 Identification et modification de l'adresse de retour

Nous allons dans un premier temps, modifier l'adresse de retour empilée lors de l'appel de fonction dans un programme C, et ceci directement dans le programme C lui-même. Dans un premier temps, nous présentons le programme de test. Ensuite, nous présenterons deux techniques permettant de localiser l'adresse de retour. Pour finir, nous exécuterons le programme de test avec la modification de cette adresse.

1.1 Le programme de test

Soit le programme `tp1.c` :

```
void f(int a, int b, int c)
{
  char buffer1[4]="aaa";
  char buffer2[8]="bbbbbbb";
}
```

```
int main()
{
  int x;

  x=0;
  f(1,2,3);
  x=1;
  printf("%d\n",x);
  return(0);
}
```

Pour compiler ce programme, nous utiliserons la commande suivante :

```
$ gcc -Wall -g -fno-stack-protector -z execstack tp1.c -o tp1
```

Comme nous l'avons vu en cours, l'adresse de retour empilée lors de l'appel de la fonction `f` se situe non loin de `buffer1` et `buffer2` dans la pile. Pour connaître sa position exacte, on peut analyser l'assembleur ou alors utiliser les petites astuces présentées en cours. Nous utiliserons ici les deux techniques.

1.2 Analyser l'assembleur

La traduction en assembleur du programme `tp1` peut être obtenue avec la commande `gcc` :

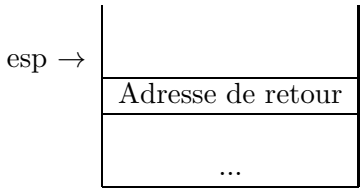
```
$ gcc -Wall -S -fno-stack-protector -z execstack tp1.c -o tp1.s
```

L'option `-g` a volontairement été omise pour ne pas surcharger l'affichage avec les options de debuggage. Voici un extrait du fichier `tp1.s` ainsi obtenu :

```
$ cat -n tp1.s
...
    10  f:
    11          pushl   %ebp
    12          movl    %esp, %ebp
    13          subl    $16, %esp
    14          movl    .LC0, %eax
    15          movl    %eax, -4(%ebp)
    16          movl    .LC1, %eax
```

```
    17          movl    .LC1+4, %edx
    18          movl    %eax, -12(%ebp)
    19          movl    %edx, -8(%ebp)
    20          leave
    21          ret
...
```

L'état de la pile, juste avant l'exécution de l'instruction de la ligne 11, est le suivant (l'instruction `pushl` n'a pas encore été exécutée) :



1. Déterminez la nature des éléments `.LC0` et `.LC1`.
2. Déduisez les adresses de `buffer1` et `buffer2`.
3. Tracez l'évolution de la pile, au cours de l'exécution de la fonction `f`.
4. Déduisez l'adresse de retour relativement au registre `ebp`, à la ligne 20 (et pas conséquent par rapport à `buffer1`).

1.3 Utiliser `gdb`

Nous allons réaliser toutes les manipulations à l'aide du debugger `gdb`. Il est au préalable nécessaire de compiler le programme `tp1.c` avec l'option `-g`. Nous devons déterminer la valeur de l'adresse de retour empilée lors de l'appel de la fonction `f` et déterminer la position de cette adresse de retour par rapport à `buffer1` ou `buffer2`. Nous utiliserons les fonctionnalités suivantes de `gdb` :

```
$ gdb tp1                    (<- desassemblage de main)
(gdb) disas main
Dump of assembler code for function main:
0x080483d1 <main+0>:    lea    0x4(%esp),%ecx
...
(gdb) list                   (<- lister le programme source)
2       {
3         char buffer1[4]="aaaa";
4         char buffer2[8]="bbbbbbbb";
```

```
5       }
...
(gdb) b 5                    (<- point d'arret a la ligne 5)
Breakpoint 1 at 0x80483be: file tp1.c, line 5
(gdb) run                    (<- execution du programme)
(gdb) x /20x buffer1         (<- examiner 20 octets de memoire a
                                partir de buffer1)
(gdb) info frame             (<- information sur le contexte
                                 d'execution courant)
 eip = 0x80483ee in function (tp1.c:10); saved eip 0x804844b
 called by frame at 0xbffff7e0
 source language c.
 Arglist at 0xbffff7a8, args: a=1, b=2, c=3
 Locals at 0xbffff7a8, Previous frame's sp is 0xbffff7b0
 Saved registers:
  ebp at 0xbffff7a8, eip at 0xbffff7ac
```

1. Déterminez l'adresse de retour en cherchant l'instruction qui suit l'appel de la fonction `f` dans le main. Notez l'adresse de cette instruction.
2. Tentez de repérer cette adresse dans la pile. Pour cela, exécutez le programme et interrompez-le juste après les initialisations de `buffer1` et `buffer2`. Profitez-en pour vérifier que l'adresse de retour que vous avez trouvée est correcte à l'aide de `info frame`.
3. Identifiez la distance entre cette adresse et celle de `buffer1`.

1.4 Modifier l'adresse de retour

Nous avons maintenant tout ce dont nous avons besoin pour modifier l'adresse de retour de la fonction. Nous allons simplement faire en sorte que ce programme C saute l'instruction `x=1` après l'appel à la fonction `f`. Il faut pour cela déterminer l'adresse dans le `main` de l'instruction qui suit `x=1` de façon à sauter à cette adresse et modifier l'adresse de retour (nous savons maintenant où elle dans la pile) de façon à sauter l'instruction.

1. A l'aide de `gdb`, déterminez la nouvelle adresse de retour à laquelle nous voulons sauter (dans le `main`).
2. Ajoutez dans la fonction `f` du code C qui permet de modifier l'adresse de retour (Note : comme le code que vous allez ajouter va probablement nécessiter l'utilisation d'une nouvelle variable, il faut probablement recalculer la distance entre l'adresse de retour et `buffer1`).
3. Vérifiez, en exécutant le programme, que la modification fonctionne.

2 Analyse d'un buffer overflow

Nous allons à présent analyser une "vraie" attaque d'un programme vulnérable. Pour cela, nous allons utiliser le programme C vulnérable suivant :

```
void copie(char * ch)
{
  char str[512];

  strcpy(str,ch);
}

int main(int argc, char * argv[])
{
  copie(argv[1]);
  return(0);
}
```

2.1 Compilation du programme vulnérable

Comme vous pouvez le constater, ce programme utilise `argv[1]`, paramètre fourni par l'utilisateur, sans l'assainir ni le tester avant de l'utiliser. Ce paramètre est copié dans une variable locale de la fonction `copie` à l'aide de la fonction `strcpy`. Le paramètre fourni va donc bien être recopié dans la pile, à l'adresse `str` et ceci à l'aide d'une fonction qui ne vérifie pas que la taille est correcte avant la copie. Si nous fournissons donc en entrée du programme, une chaîne de caractères trop grande, nous pouvons donc écraser `str` et les octets suivants, et par conséquent l'adresse de retour de la fonction `copie`.

La compilation du programme vulnérable se fait comme dans la section 1 :

```
gcc -g -fno-stack-protector -z execstack vulnerable.c -o vulnerable
```

Ensuite, de façon à désactiver la randomization de l'espace d'adressage des processus (ASLR), exécutez la commande :

```
$ echo 0 > /proc/sys/kernel/randomize_va_space
```

2.2 Le shellcode

Un shellcode est en général utilisé par les attaquants pour être exécuté lors du détournement de la fonction (cela leur permet d'obtenir un invité de commandes sur la machine attaquée). La compréhension de ce shellcode n'entre pas dans le cadre de notre formation. L'enseignant vous fournira un exemple que vous pourrez utiliser pour la suite du TP.

2.3 Fabrication de l'argument `argv[1]`

Il vous reste maintenant à fabriquer une chaîne de caractères, qui sera fournie en paramètre du programme, et qui soit : 1) d'une longueur suffisante pour espérer écraser l'adresse de retour dans la pile et 2) qui écrase cette adresse de retour avec l'adresse où sera copié cette chaîne de caractères, c'est-à-dire à l'adresse `str`. Il faut donc deviner cette adresse. Comme il est très difficile d'estimer précisèment cette adresse, nous allons dans notre chaînes de caractères, inclure beaucoup de NOP au début de façon à pouvoir s'autoriser une imprécision dans la recherche de l'adresse `str`.

La chaîne que nous allons fabriquer est donc ainsi formée :

```
NNNNNNNNNNSSSSSSSSSSSSSSSSSSAAAAAAAAAA
```

où :
- `N` est l'instruction NOP ;
- `SSSSSSSSSSSSSSS` est le shellcode ;
- `A` est l'adresse *supposée* de `str`.

Pour déterminer l'adresse `A`, comme on a desactivé ASLR, on peut se servir de la fonction suivante, qui permet en C de connaître la valeur du pointeur de pile :

```
unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}
long l=get_sp();
```

L'algorithme du programme est le suivant :
- Calcul de l'adresse de base de la pile (utilisation de la fonction `get_sp`).
- Calcul de l'adresse du tableau `str` : `get_sp - offset`. L'objectif est d'arriver à remplacer l'adresse de retour par cette valeur, de manière à sauter dans [et exécuter] le code contenu dans le tableau `str`.
- Calcul de la taille de la chaîne à fabriquer (multiple de 4).

— Allocation dynamique de cette taille.
— Remplissage de cette zone allouée par l'adresse du tableau (attention, l'adresse du tableau est codée sur 4 octets).
— Remplissage de la première moitié de cette zone par des `NOP`.
— Copie du `shellcode` dans la zone en commençant par le milieu de la zone.

La technique consiste ensuite à tester des adresses qui sont "non loin" de celle du pointeur de pile et ceci jusqu'à ce que l'exploit fonctionne.

1. Réalisez un programme C qui génère une telle chaîne de caractères. Ce programme doit accepter des paramètres lui indiquant la taille totale de la chaîne et un offset permettant de faire varier l'adresse A. Cette offset représente un déplacement par rapport à l'adresse du pointeur de pile, récupérée par la fonction `get_sp()`
2. Faites en sorte d'exporter la chaîne de caractères obtenue dans une variable d'environnement (utilisation de `putenv`) et terminez votre programme C en lançant un shell fils (`system("/bin/bash")`)
3. Lancez ensuite le programme vulnérable en lui donnant en paramètre la variable d'environnement créée (faites beaucoup de tests en faisant varier la longueur de la chaîne et l'offset de l'adresse). Lorsque vous obtenez un invité de commandes indiquant le lancement d'un shell, l'exploit est réussi. On pourra utiliser 612 comme longueur de chaîne et tester les offets en partant de 0 et en incrémentant de 100 en 100 par exemple.

3 Les protections du noyau

Pour réaliser ce TP, nous avons désactivé plusieurs mécanismes de protections. Nous allons les passer en revue et identifier leur utilité.

3.1 Option de compilation de `gcc` : `-z execstack`

Compilez le programme vulnérable, sans l'option `-z execstack` et essayez à nouveau d'exploiter la vulnérabilité.

```
$ gcc -g -fno-stack-protector vulnerable.c -o vulnerable
```

Que se passe-t-il ? En déduire l'utilité de cette option de compilation.

3.2 Option de compilation de `gcc` : `-fno-stack-protector`

Compilez le programme vulnérable sans l'option `-fno-stack-protector` :

```
$ gcc -g vulnerable.c -z execstack -o vulnerable
```

Essayez à nouveau l'exploit. Que voyez-vous ?

Traduisez le programme vulnérable, sans l'option `-fno-stack-protector`, en assembleur, et comparez le fichier assembleur généré à la version précédente.

```
$ gcc -Wall -S -z execstack vulnerable.c -o vulnerable_stack-protector.s
```

Identifiez les zones différentes à l'aide de la commande `diff` *par exemple et en déduire l'utilité de cette option.*

3.3 Randomization de l'espace d'adressage

Réactivez la randomization de l'espace d'adressage :

```
$ echo 1 > /proc/sys/kernel/randomize_va_space
```

Créez un programme de test qui invoque la fonction `get_sp` et affiche la valeur retournée. Exécutez plusieurs fois ce programme de test et analysez les retours.

Que permet cette protection ?