

Hepia

Haute École du Paysage, d'Ingénierie et d'Architecture

Ingénierie et Systèmes de Communication

Année académique 2020/2021

Hacking et pentesting

Série 2 – Randomness

Genève, 14 Mars 2021

Étudiant :

Sergio Guarino

Professeur :

Stéphane Küng

Table des matières

Introduction	2
1. Nuclear Warhead code generator	3
2. CardGame	3
Note sur l'optimisation du temps d'exécution	4
3. EuroMillion.....	6
Conclusion.....	9
Annexes.....	9

Introduction

Le but de cette série d'exercices est d'analyser parties de codes et sorties d'un terminal pour comprendre le principe de fonctionnement de la fonction rand() et quels sont les moyens pour la contourner. En effet, il n'est pas possible de générer des données aléatoires à l'aide d'un système déterministe, tel qu'un ordinateur. Le nombre de variables qui peuvent être prises en compte étant fini, il sera toujours possible de déterminer la séquence de nombres générés. Dans ce cas on parle de générateurs pseudo-aléatoires. Toutefois, si on a un nombre de variables assez important, trouver la séquence exacte peut devenir vite très complexe. Néanmoins, ils existent également des générateurs très efficaces, tels que le Mersenne Twister¹, dont la séquence générée se répète après $2^{19937}-1$ nombres générés, soit 10 suivi de 6000 zéros !

Les exercices ont été effectués sur une machine Linux Ubuntu 20.04.

1. Source https://fr.wikipedia.org/wiki/Mersenne_Twister

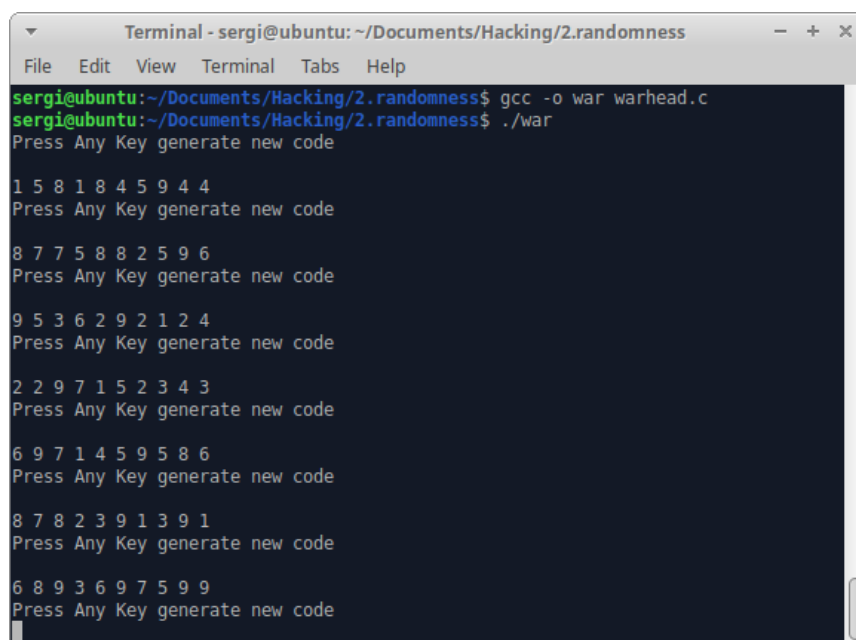
1. Nuclear Warhead code generator

Dans ce premier exercice on a à disposition un code utilisé pour générer une séquence de 10 nombres ainsi que le résultat affiché à la console lorsque ce code a été exécuté la dernière fois.

En lisant le code, on retrouve 2 fonctions qui peuvent nous aider à retrouver la séquence : **rand** et **srand**. La fonction *rand()* retourne un nombre pseudo-aléatoire et *srand()* permet de fournir un *seed* à la fonction *rand*, soit un sel qui permet de rendre la sortie un peu plus aléatoire. La particularité de ce sel, est que, s'il reste le même, la séquence de nombres générés par *rand* sera toujours la même. Ceci veut dire que, si on retrouve le sel utilisé lorsque le code a été lancé, on obtiendra la même suite de nombres que celle affichée à la console de l'exercice.

Depuis la capture d'écran qui nous est fournie, on sait que le code a été lancé le 1^{er} Mars 2021 à 23h20 et 17 secondes. Le sel est donc le nombre de secondes écoulées entre cette date et le minuit du 1^{er} Janvier 1970. On peut utiliser une structure de la librairie *time.h* pour retrouver ce sel ou bien utiliser une calculatrice en ligne pour connaître le nombre de secondes écoulées.

Une fois rentrée la bonne date, si on exécute le code, on retrouve la même séquence que celle récupéré sur la console. On peut donc retrouver la suite de la séquence :



```
Terminal - sergi@ubuntu: ~/Documents/Hacking/2.randomness
sergi@ubuntu:~/Documents/Hacking/2.randomness$ gcc -o war warhead.c
sergi@ubuntu:~/Documents/Hacking/2.randomness$ ./war
Press Any Key generate new code

1 5 8 1 8 4 5 9 4 4
Press Any Key generate new code

8 7 7 5 8 8 2 5 9 6
Press Any Key generate new code

9 5 3 6 2 9 2 1 2 4
Press Any Key generate new code

2 2 9 7 1 5 2 3 4 3
Press Any Key generate new code

6 9 7 1 4 5 9 5 8 6
Press Any Key generate new code

8 7 8 2 3 9 1 3 9 1
Press Any Key generate new code

6 8 9 3 6 9 7 5 9 9
Press Any Key generate new code
```

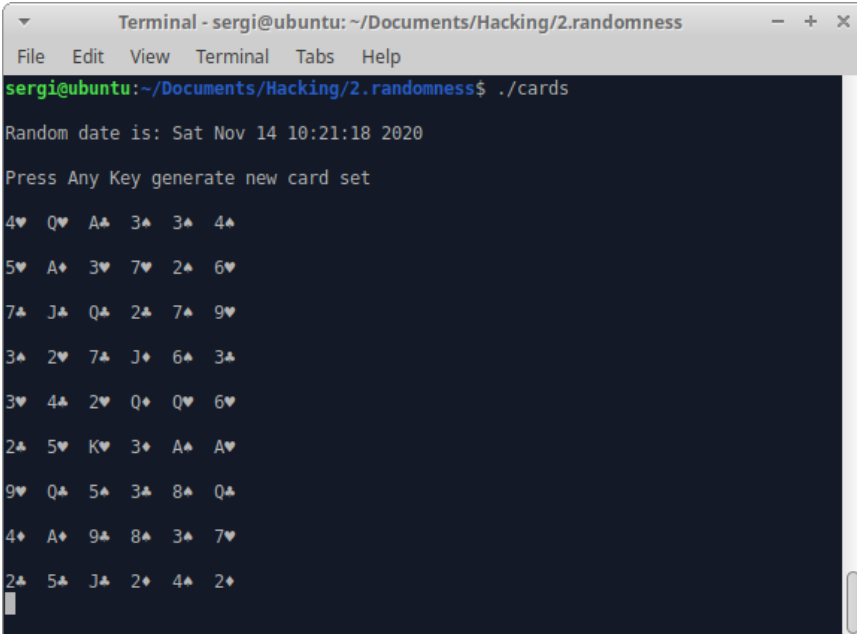
2. CardGame

Dans ce deuxième exercice, on doit prédire les futurs tirages d'un jeu de cartes. La différence avec l'exercice précédent, est que cette fois on a une double séquence d'éléments à deviner et qu'on n'a pas une date précise à laquelle le code a été exécuté. En revanche on sait que l'année d'exécution est le 2020 et que les couleurs des cartes ont été choisi avant leurs valeurs.

On a décidé d'écrire un code en 2 parties : une partie pour trouver la date et l'autre pour retrouver la bonne séquence.

Pour la première partie, il a fallu comparer chaque carte trouvée avec celle de la suite à disposition pour l'exercice. Pour simplifier, la comparaison a été faite seulement avec les 5 premières cartes. Dans tous les cas, le code a été écrit de façon à chercher tous les sels pour lesquels la séquence était juste. Cette comparaison a été faite pour chaque seconde de l'année, que pour l'année 2020 correspond à 31'622'400 secondes, 86400 plus que d'habitudes puisque bissextile.

Après exécution de la première partie du code, on a trouvé que la date à laquelle le programme avait été lancé, était Samedi 14 Novembre 2020 à 10h21 et 18 secondes. En rentrant ce sel comme paramètre de la fonction `srand`, avec la deuxième partie du code on a réussi à récupérer la suite de la séquence de cartes :



```
Terminal - sergi@ubuntu: ~/Documents/Hacking/2.randomness
sergi@ubuntu:~/Documents/Hacking/2.randomness$ ./cards
Random date is: Sat Nov 14 10:21:18 2020
Press Any Key generate new card set
4♥ Q♥ A♠ 3♠ 3♠ 4♠
5♥ A♠ 3♥ 7♥ 2♠ 6♥
7♠ J♠ Q♠ 2♠ 7♠ 9♥
3♠ 2♥ 7♠ J♠ 6♠ 3♠
3♥ 4♠ 2♥ Q♠ Q♥ 6♥
2♠ 5♥ K♥ 3♠ A♠ A♥
9♥ Q♠ 5♠ 3♠ 8♠ Q♠
4♠ A♠ 9♠ 8♠ 3♠ 7♥
2♠ 5♠ J♠ 2♠ 4♠ 2♠
```

Remarque intéressante : cette date, convertie en secondes, donne le chiffre : 1605345678. Le derniers 6 chiffres sont en séquence. Une coïncidence ?

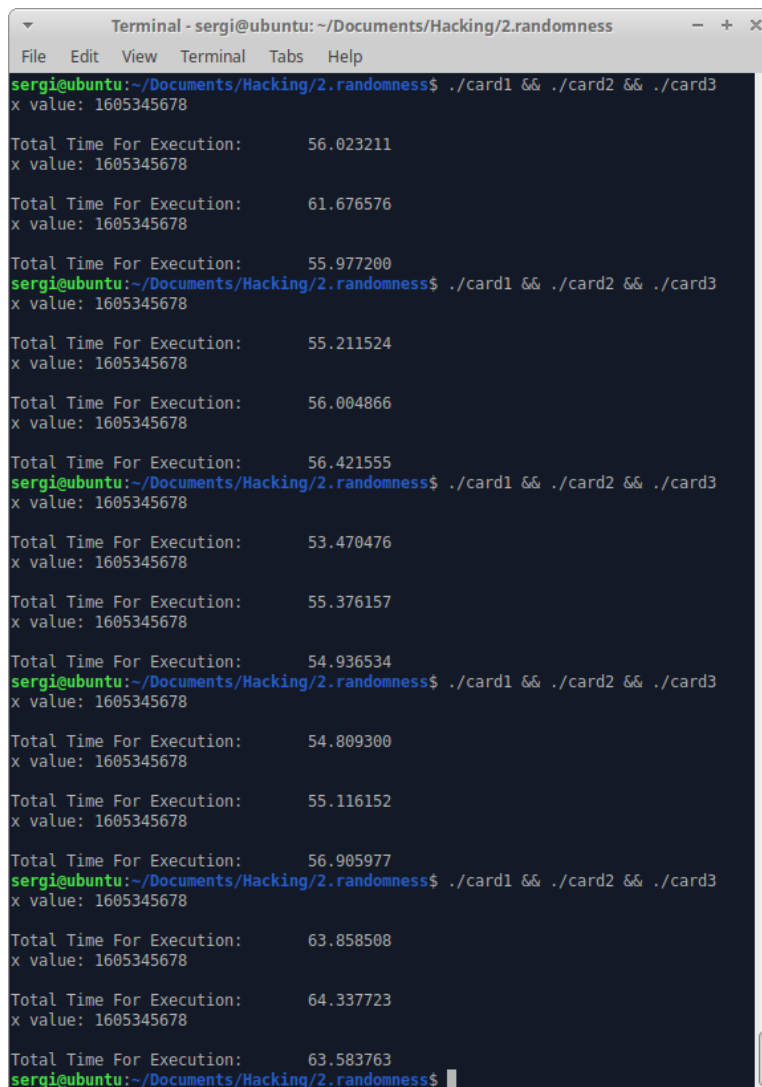
Note sur l'optimisation du temps d'exécution

Comme 31 millions de calculs est un nombre relativement petit pour un ordinateur moderne, la comparaison a été faite pour une carte à la fois et si cette condition était vraie, on aurait vérifié la 2^{ème} carte et ainsi de suite. Mais on aurait pu également tester une seule valeur ou couleur à la fois pour rendre l'opération théoriquement plus rapide. Ou encore comparer les 5 cartes au même temps en écrivant une seule fonction `if`.

Un test a été effectué pour déterminer laquelle de ces méthodes était la plus rapide. Grâce au type `clock_t` et à la fonction `clock` de la librairie `time.h` on peut connaître le temps d'exécution d'un programme.

Trois programmes différents ont été créés :

- `card1` qui teste une carte à la fois (5 fonctions `if`)
- `card2` qui teste 5 cartes à la fois (1 fonction `if`)
- `card3` qui teste une couleur et une valeur à la fois (10 fonctions `if`).

A terminal window titled 'Terminal - sergi@ubuntu: ~/Documents/Hacking/2.randomness'. It shows a series of commands and their outputs. The commands are: `./card1 && ./card2 && ./card3` and `./card1 && ./card2`. The outputs show 'x value: 1605345678' and 'Total Time For Execution: [time]'. The times for card1, card2, and card3 are consistently close to each other, around 55-65 seconds. The terminal text is as follows:

```
sergi@ubuntu:~/Documents/Hacking/2.randomness$ ./card1 && ./card2 && ./card3
x value: 1605345678
Total Time For Execution: 56.023211
x value: 1605345678
Total Time For Execution: 61.676576
x value: 1605345678
Total Time For Execution: 55.977200
sergi@ubuntu:~/Documents/Hacking/2.randomness$ ./card1 && ./card2 && ./card3
x value: 1605345678
Total Time For Execution: 55.211524
x value: 1605345678
Total Time For Execution: 56.004866
x value: 1605345678
Total Time For Execution: 56.421555
sergi@ubuntu:~/Documents/Hacking/2.randomness$ ./card1 && ./card2 && ./card3
x value: 1605345678
Total Time For Execution: 53.470476
x value: 1605345678
Total Time For Execution: 55.376157
x value: 1605345678
Total Time For Execution: 54.936534
sergi@ubuntu:~/Documents/Hacking/2.randomness$ ./card1 && ./card2 && ./card3
x value: 1605345678
Total Time For Execution: 54.809300
x value: 1605345678
Total Time For Execution: 55.116152
x value: 1605345678
Total Time For Execution: 56.905977
sergi@ubuntu:~/Documents/Hacking/2.randomness$ ./card1 && ./card2 && ./card3
x value: 1605345678
Total Time For Execution: 63.858508
x value: 1605345678
Total Time For Execution: 64.337723
x value: 1605345678
Total Time For Execution: 63.583763
sergi@ubuntu:~/Documents/Hacking/2.randomness$
```

On peut voir dans la capture ci-dessus que le temps d'exécution pour les 3 fonctions est pratiquement le même. Les 3 programmes ont été exécuté plusieurs fois pour avoir des valeurs moyennes. On voit que card2 prends quelques dixièmes de secondes de plus par rapport à card1 (cas 2-3-4-5), ça pourrait être une indication qu'il est plus efficace par rapport à card2. Quant à card3, il n'est pas beaucoup plus rapide des autres.

Il faut aussi tenir en compte le fait que le temps d'exécution ne dépend pas seulement du programme, mais aussi de l'utilisation du processeur au moment que les différents programmes sont exécutés. Et en effet, lors de la dernière exécution, une vidéo a été ouverte sur le navigateur au même temps et on peut voir comme le processeur, en ayant dû allouer des ressources pour le navigateur et la vidéo, a pris presque 10 secondes supplémentaires pour exécuter le code.

Même si une optimisation du temps d'exécution n'est pas nécessaire pour un cas comme celui de cet exercice où on essaye de trouver un sel par force brute, il faut considérer que pour des codes beaucoup plus complexes, réussir à réduire le temps d'exécution même d'une seule microseconde, peut faire la différence. Par exemple, si on a une boucle for à exécuter 1 milliard de fois, gagner une microseconde par tour de boucle correspond à gagner plus de 15 minutes !

3. EuroMillion

Pour le dernier exercice on nous demande de trouver deux séquences de nombres (5 plus 2 nombres) choisis dans deux groupes de 50 et 12 éléments. Pour cet aspect, il peut être vu comme une variante du deuxième exercice, sauf avec plus de combinaisons possibles. Comme pour l'exercice précédent, on ne connaît pas la date exacte à laquelle le code a été exécuté, mais on sait qu'il l'a été au premier semestre 2020, donc il y a la moitié de dates à vérifier.

Toutefois, l'exercice ne peut pas être abordé comme le deuxième. Une des difficultés est de trouver l'ordre de génération des nombres. Par exemple, on ne sait pas si on a choisi d'abord 5 nombres du premier groupe et puis les 2 du deuxième groupe ou si on a choisi d'abord 3 nombres dans le premier, puis 1 dans le deuxième et ainsi de suite. Les combinaisons possibles sont assez nombreuses et on se rend compte que ça devient vite très complexe de trouver la solution en essayant toutes les combinaisons possibles.

Heureusement, on a à disposition l'exécutable utilisé pour l'exercice. Une analyse du code avec Ghidra nous permet de comprendre tout de suite comment les nombres ont été générés.

```
Decompile: main - (euromillion)
1
2 void main(void)
3
4 {
5     time_t tVar1;
6     void *pvVar2;
7     void *pvVar3;
8     int local_28;
9     int local_24;
10    int local_20;
11    int local_1c;
12
13    puts("Draw program for EuroMillion");
14    tVar1 = time((time_t *)0x0);
15    srand((uint)tVar1);
16    pvVar2 = malloc(200);
17    pvVar3 = malloc(0x30);
18    do {
19        puts("Press Any Key to launch draw lots");
20        getchar();
21        local_28 = 0;
22        while (local_28 < 0x32) {
23            *(int *)((long)pvVar2 + (long)local_28 * 4) = local_28 + 1;
24            local_28 = local_28 + 1;
25        }
26        local_24 = 0;
27        while (local_24 < 0xc) {
28            *(int *)((long)pvVar3 + (long)local_24 * 4) = local_24 + 1;
29            local_24 = local_24 + 1;
30        }
31        secure_shuffle(pvVar2, 0x32);
32        secure_shuffle(pvVar3, 0xc);
33        printf("Draw : ");
34        local_20 = 0;
35        while (local_20 < 5) {
36            printf("%d ", (ulong)*(uint *)((long)pvVar2 + (long)local_20 * 4));
37            local_20 = local_20 + 1;
38        }
39        printf(" - ");
40        local_1c = 0;
41        while (local_1c < 2) {
42            printf("%d ", (ulong)*(uint *)((long)pvVar3 + (long)local_1c * 4));
43            local_1c = local_1c + 1;
44        }
45        putchar(10);
46    } while( true );
47 }
48
```

Ci-dessus on retrouve la fonction main de l'exécutable. On remarque les choses suivantes :

- Le sel de la fonction srand correspond à la date et l'heure à laquelle le code a été exécuté
- 2 tableaux sont créés pour stocker les nombres
- Une fonction **secure_shuffle** prend en paramètre le tableau et sa taille. On remarque qu'on appelle la fonction d'abord avec le tableau de 50 nombres et puis avec le tableau de 12 nombres
- Des boucles pour l'affichage des combinaisons

Si on regarde la fonction secure_shuffle, on arrive à trouver la façon exacte dont les nombres ont été calculés. On se rend compte aussi qu'il aurait été presque impossible retrouver la séquence exacte des opérations. Notamment, la **ligne 13** nous dit que les nombres ont été recalculés un nombre « aléatoire » de fois avant de terminer la fonction, ce qui rend encore plus difficile une estimation à l'aveugle du code.

```
C: Decompiler: secure_shuffle - (euromillion)
1 |
2 void secure_shuffle(long param_1,int param_2)
3 |
4 {
5     undefined4 uVar1;
6     int iVar2;
7     int iVar3;
8     int local_1c;
9     int local_18;
10 |
11     iVar2 = rand();
12     local_1c = 0;
13     while (local_1c < iVar2 % 5 + 2) {
14         local_18 = 0;
15         while (local_18 < param_2) {
16             iVar3 = rand();
17             uVar1 = *(undefined4 *) (param_1 + (long) local_18 * 4);
18             *(undefined4 *) (param_1 + (long) local_18 * 4) =
19                 *(undefined4 *) (param_1 + (long) (iVar3 % param_2) * 4);
20             *(undefined4 *) ((long) (iVar3 % param_2) * 4 + param_1) = uVar1;
21             local_18 = local_18 + 1;
22         }
23         local_1c = local_1c + 1;
24     }
25     return;
26 }
27 |
```

Grâce au code source, pour trouver la solution, on applique la même méthode que pour le deuxième exercice et on exécute le code autant de fois que de secondes dans les premiers 6 mois du 2020.

Pour ce faire il suffit d'adapter le code pour qu'il puisse être exécuté : par exemple remplacer les *undefined4* dans la fonction secure_shuffle par des entiers. Quand on le compile on obtient des Warnings car on passe en paramètre à secure_shuffle l'adresse du tableau sans lui dire explicitement.

```
Terminal - sergi@ubuntu: ~/Documents/Hacking/2.randomness
File Edit View Terminal Tabs Help

sergi@ubuntu:~/Documents/Hacking/2.randomness$ gcc -o euroM euroMillion.c
euroMillion.c: In function 'main':
euroMillion.c:78:20: warning: passing argument 1 of 'secure_shuffle' makes integ
er from pointer without a cast [-Wint-conversion]
   78 |     secure_shuffle(pvVar2,0x32);
      |                   ^~~~~~
      |                   |
      |                   int *
euroMillion.c:21:26: note: expected 'long int' but argument is of type 'int *'
   21 | void secure_shuffle(long param_1,int param_2){
      |                   ^~~~~~
euroMillion.c:79:20: warning: passing argument 1 of 'secure_shuffle' makes integ
er from pointer without a cast [-Wint-conversion]
   79 |     secure_shuffle(pvVar3,0xc);
      |                   ^~~~~~
      |                   |
      |                   int *
euroMillion.c:21:26: note: expected 'long int' but argument is of type 'int *'
   21 | void secure_shuffle(long param_1,int param_2){
      |                   ^~~~~~
sergi@ubuntu:~/Documents/Hacking/2.randomness$
```

On a pu ainsi retrouver la date d'exécution du code et obtenir les extractions suivantes du jeu.

```
Terminal - sergi@ubuntu: ~/Documents/Hacking/2.randomness
File Edit View Terminal Tabs Help

sergi@ubuntu:~/Documents/Hacking/2.randomness$ ./euroM
Draw program for EuroMillion

Random date is: Sat May  2 22:22:22 2020

Press Any Key to launch draw lots

Draw : 28 44 30 35 10 - 1 10
Press Any Key to launch draw lots

Draw : 14 46 47 28 8 - 4 12
Press Any Key to launch draw lots

Draw : 44 12 4 42 28 - 6 11
Press Any Key to launch draw lots

Draw : 28 21 9 39 25 - 4 5
Press Any Key to launch draw lots

Draw : 33 48 46 28 43 - 8 10
Press Any Key to launch draw lots

Draw : 22 29 35 4 49 - 9 8
Press Any Key to launch draw lots

Draw : 29 12 20 27 6 - 10 6
Press Any Key to launch draw lots
```

Bizarrement, la date d'exécution est le 2 Mai 2020 à 22h22 et 22 secondes. Une autre coïncidence ?

Conclusion

On a pu voir comme des séquences de nombres à l'apparence aléatoires ne sont en réalité que facilement prévisibles une fois que l'on sait comment elles ont été générées.

Quelle est donc la solution pour pouvoir avoir des nombres plus aléatoires ? Dans l'introduction on a parlé de Mersenne Twister, mais ils existent également d'autres manières. Une première méthode pour rendre le code plus sûr est d'utiliser, à la place de *rand*, la fonction *arc4random*, qui se base sur */dev/urandom* pour avoir une sortie plus aléatoire. */dev/urandom* est un processus Linux qui récupère les valeurs de variables de l'environnement de travail pour générer des valeurs aléatoires. Il reste néanmoins un générateur pseudo-aléatoire et */dev/random* est à préférer si on souhaite rendre une application la plus aléatoire possible.

Autres méthodes incluent se servir d'événements atmosphériques, tel que la mesure de la radiation², mais on pourrait imaginer plein d'autres événements qui rendraient des nombres aléatoires. Une bonne pratique est d'utiliser des nombres avec un nombre élevé d'octets, qui rendrait une attaque de force brute assez coûteuse à mettre en place.

2. Source : <https://www.sparkfun.com/tutorials/132>

Annexes

Ce dossier inclut en annexe les codes sources utilisés pour effectuer les exercices. La partie de code utilisée pour trouver le sel a été mise en commentaire.