

Hepia

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

Ingénierie et Systèmes de Communication

Année académique 2020/2021

Hacking et pentesting

Série 1 – Hardcoded passwords

Genève, 7 Mars 2021

Étudiant :

Sergio Guarino

Professeur :

Stéphane Küng

Table des matières

Introduction	2
1. CrackMe 1	3
2. CrackMe 2	3
3. CrackMe 3	10
Conclusions	14

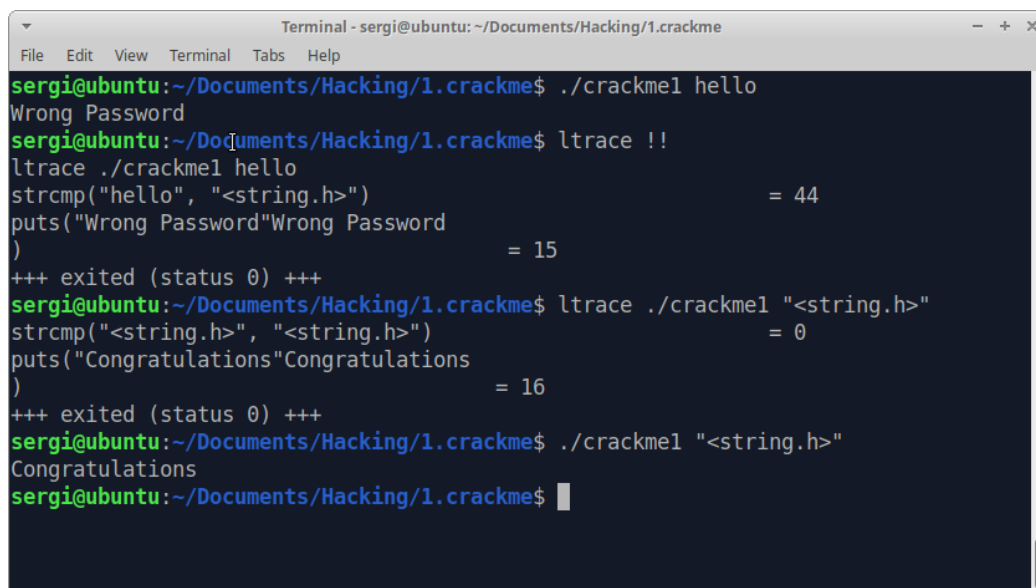
Introduction

Le but de cette série d'exercices est d'analyser 3 fichiers exécutables pour essayer de retrouver des mots de passes qui ont été écrit en dur dans le code.

Les fichiers à disposition sont en format ELF pour une machine Linux à 64 bits avec architecture x86. Cette série d'exercices a été effectuée sur une machine Linux Ubuntu 20.04.

1. CrackMe 1

Pour le premier exécutable fourni on nous a indiqué que le mot de passe était écrit en dur dans le code. Cette information nous permet de retrouver facilement le mot de passe en utilisant la commande **ltrace**, qui affiche à l'écran les appels aux bibliothèques utilisées lors de l'exécution du code.



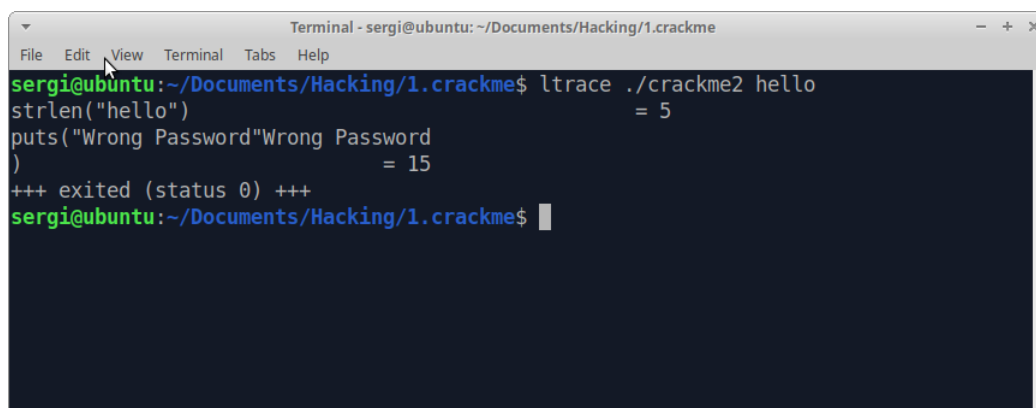
```
Terminal - sergi@ubuntu: ~/Documents/Hacking/1.crackme
sergi@ubuntu:~/Documents/Hacking/1.crackme$ ./crackme1 hello
Wrong Password
sergi@ubuntu:~/Documents/Hacking/1.crackme$ ltrace !!
ltrace ./crackme1 hello
strcmp("hello", "<string.h>") = 44
puts("Wrong Password"Wrong Password) = 15
+++ exited (status 0) +++
sergi@ubuntu:~/Documents/Hacking/1.crackme$ ltrace ./crackme1 "<string.h>"
strcmp("<string.h>", "<string.h>") = 0
puts("Congratulations"Congratulations) = 16
+++ exited (status 0) +++
sergi@ubuntu:~/Documents/Hacking/1.crackme$ ./crackme1 "<string.h>"
Congratulations
sergi@ubuntu:~/Documents/Hacking/1.crackme$
```

Quand on exécute crackme1, tout paramètre différent du mot de passe correct fera afficher à l'écran un message de mot de passe incorrect (1^{ère} commande dans l'image ci-dessus).

Si on exécute à nouveau le code avec **ltrace**, on peut voir que ce qui est passé en paramètre est comparé avec le mot **<string.h>** (2^{ème} commande). Si on exécute à nouveau le programme avec ce mot en paramètre, on peut voir que c'est le bon mot de passe (3^{ème} et 4^{ème} commandes).

2. CrackMe 2

Dans le deuxième cas, on nous dit que le mot de passe est modifié dynamiquement lors de l'exécution. Ceci nous fait comprendre que **ltrace** ne sera pas utile cette fois et en effet, si on exécute le code avec **ltrace**, on obtient le résultat suivant :



```
Terminal - sergi@ubuntu: ~/Documents/Hacking/1.crackme
sergi@ubuntu:~/Documents/Hacking/1.crackme$ ltrace ./crackme2 hello
strlen("hello") = 5
puts("Wrong Password"Wrong Password) = 15
+++ exited (status 0) +++
sergi@ubuntu:~/Documents/Hacking/1.crackme$
```

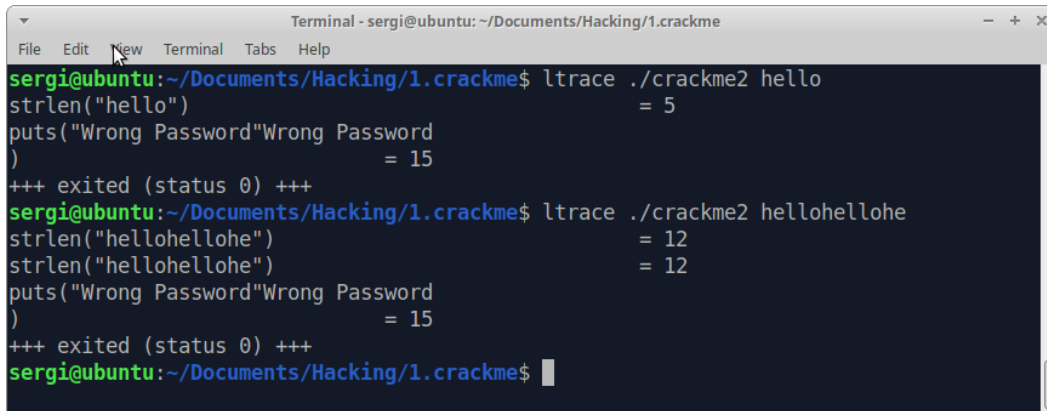
On voit qu'il y a un appel à **strlen**, ça pourrait être un indice que le mot de passe doit être d'une longueur spécifique, mais on n'a pas d'informations supplémentaires.

On peut alors analyser le code en lisant l'assembleur de l'exécutable. Il y a plusieurs programmes pour cela, dans ce cas on va utiliser **Ghidra**, qui a une interface utilisateur assez user-friendly. Ce programme a été développé par la NSA et mis à disposition gratuitement à l'adresse <https://ghidra-sre.org>.

En important et en analysant le code dans **Ghidra**, on a les informations suivantes en décompilant la section **main** du programme :

```
Decompile: main - (crackme2)
1
2 undefined8 main(int param_1,undefined8 *param_2)
3
4 {
5     size_t sVar1;
6     long in_FS_OFFSET;
7     int local_34;
8     undefined8 local_2d;
9     undefined4 local_25;
10    undefined local_21;
11    long local_20;
12
13    local_20 = *(long *)(in_FS_OFFSET + 0x28);
14    if (param_1 == 2) {
15        sVar1 = strlen((char *)param_2[1]);
16        if (sVar1 == 0xc) {
17            local_2d = 0x716c616b706a6370;
18            local_25 = 0x34373537;
19            local_21 = 0;
20            local_34 = 0;
21            while( true ) {
22                sVar1 = strlen((char *)param_2[1]);
23                if (sVar1 <= (ulong)(long)local_34) break;
24                if ((*byte *)((long)local_34 + param_2[1]) ^ 5) !=
25                    *(byte *)((long)&local_2d + (long)local_34) {
26                    puts("Wrong Password");
27                    goto LAB_0010127a;
28                }
29                local_34 = local_34 + 1;
30            }
31            puts("Congratulations");
32        }
33        else {
34            puts("Wrong Password");
35        }
36    }
37    else {
38        printf("Usage : %s password\n",*param_2);
39    }
40    LAB_0010127a:
41    if (local_20 != *(long *)(in_FS_OFFSET + 0x28)) {
42        /* WARNING: Subroutine does not return */
43        __stack_chk_fail();
44    }
45    return 0;
46 }
47
```

Le mot de passe n'est toujours pas visible, mais on retrouve des indices qui nous permettent de le déduire. Par exemple, aux **lignes 15 et 16** on voit que la longueur du mot que l'on rentre en paramètre est comparée avec la valeur **0xc**, qui en décimal est 12. On peut essayer de voir ce que **ltrace** nous affiche si on rentre un mot de passe de 12 caractères.



```
Terminal - sergi@ubuntu: ~/Documents/Hacking/1.crackme
File Edit New Terminal Tabs Help
sergi@ubuntu:~/Documents/Hacking/1.crackme$ ltrace ./crackme2 hello
strlen("hello") = 5
puts("Wrong Password") = 15
+++ exited (status 0) +++
sergi@ubuntu:~/Documents/Hacking/1.crackme$ ltrace ./crackme2 hellohellohe
strlen("hellohellohe") = 12
strlen("hellohellohe") = 12
puts("Wrong Password") = 15
+++ exited (status 0) +++
sergi@ubuntu:~/Documents/Hacking/1.crackme$
```

Malheureusement, ça nous n'aide pas. Il faut alors analyser le code un peu plus en profondeur.

Dans la capture prise avec **Ghidra**, on peut voir aux **lignes 24 et 25** une comparaison qui se fait et que tant qu'elle n'est pas satisfaite, ça nous affichera toujours le message **Wrong password**.

La ligne `*(byte *)((long)local_34 + param_2[1]) ^ 5` signifie que le mot passé en paramètre est parcouru caractère par caractère (`local_34` est incrémenté de 1 à chaque itération) et comparé avec la valeur contenue dans la variable `local_2d` `*(byte *)((long)&local_2d + (long)local_34)`. Comme l'on peut le voir à la **ligne 17**, `local_2d` ne contient pas un mot mais une adresse d'un registre, que l'on ne peut pas voir avec Ghidra.

La **ligne 24** nous dit aussi qu'il y a une opération qui est effectuée sur notre mot, pour chaque caractère le programme fait un **XOR 5**. Ceci veut dire que pour trouver le mot de passe il va falloir faire l'opération inverse.

On peut alors exécuter notre programme avec **GDB** (GNU Debugger) qui permet d'effectuer des opérations sur le code pendant qu'il est exécuté.

Il est lancé avec la commande `gdb ./crackme2` qui nous fait accéder à son interface. Depuis là, avec la commande `disass main` on retrouve le code assembleur du programme. Voici le code complet :

```
Terminal - sergi@ubuntu: ~/Documents/Hacking/1.crackme
File Edit View Terminal Tabs Help
(gdb) disass main
Dump of assembler code for function main:
0x000055555555169 <+0>: push %rbp
0x00005555555516a <+1>: mov %rsp,%rbp
0x00005555555516d <+4>: push %rbx
0x00005555555516e <+5>: sub $0x38,%rsp
0x000055555555172 <+9>: mov %edi,-0x34(%rbp)
0x000055555555175 <+12>: mov %rsi,-0x40(%rbp)
0x000055555555179 <+16>: mov %fs:0x28,%rax
0x000055555555182 <+25>: mov %rax,-0x18(%rbp)
0x000055555555186 <+29>: xor %eax,%eax
0x000055555555188 <+31>: cmpl $0x2,-0x34(%rbp)
0x00005555555518c <+35>: je 0x555555551b3 <main+74>
0x00005555555518e <+37>: mov -0x40(%rbp),%rax
0x000055555555192 <+41>: mov (%rax),%rax
0x000055555555195 <+44>: mov %rax,%rsi
0x000055555555198 <+47>: lea 0xe65(%rip),%rdi # 0x555555556004
0x00005555555519f <+54>: mov $0x0,%eax
0x0000555555551a4 <+59>: callq 0x55555555060 <printf@plt>
0x0000555555551a9 <+64>: mov $0x0,%eax
0x0000555555551ae <+69>: jmpq 0x5555555527a <main+273>
0x0000555555551b3 <+74>: mov -0x40(%rbp),%rax
0x0000555555551b7 <+78>: add $0x8,%rax
0x0000555555551bb <+82>: mov (%rax),%rax
0x0000555555551be <+85>: mov %rax,%rdi
0x0000555555551c1 <+88>: callq 0x55555555040 <strlen@plt>
0x0000555555551c6 <+93>: cmp $0xc,%rax
0x0000555555551ca <+97>: je 0x555555551e2 <main+121>
0x0000555555551cc <+99>: lea 0xe46(%rip),%rdi # 0x555555556019
0x0000555555551d3 <+106>: callq 0x55555555030 <puts@plt>
0x0000555555551d8 <+111>: mov $0x0,%eax
0x0000555555551dd <+116>: jmpq 0x5555555527a <main+273>
0x0000555555551e2 <+121>: movl $0x0,-0x2c(%rbp)
0x0000555555551e9 <+128>: movabs $0x716c616b706a6370,%rax
0x0000555555551f3 <+138>: mov %rax,-0x25(%rbp)
0x0000555555551f7 <+142>: movl $0x34373537,-0x1d(%rbp)
0x0000555555551fe <+149>: movb $0x0,-0x19(%rbp)
0x000055555555202 <+153>: movl $0x0,-0x2c(%rbp)
0x000055555555209 <+160>: jmp 0x5555555524b <main+226>
0x00005555555520b <+162>: mov -0x40(%rbp),%rax
0x00005555555520f <+166>: add $0x8,%rax
0x000055555555213 <+170>: mov (%rax),%rdx
0x000055555555216 <+173>: mov -0x2c(%rbp),%eax
0x000055555555219 <+176>: cltq
0x00005555555521b <+178>: add %rdx,%rax
0x00005555555521e <+181>: movzbl (%rax),%eax
0x000055555555221 <+184>: xor $0x5,%eax
0x000055555555224 <+187>: mov %eax,%edx
0x000055555555226 <+189>: mov -0x2c(%rbp),%eax
0x000055555555229 <+192>: cltq
0x00005555555522b <+194>: movzbl -0x25(%rbp,%rax,1),%eax
0x000055555555230 <+199>: cmp %al,%dl
0x000055555555232 <+201>: je 0x55555555247 <main+222>
0x000055555555234 <+203>: lea 0xdde(%rip),%rdi # 0x555555556019
0x00005555555523b <+210>: callq 0x55555555030 <puts@plt>
0x000055555555240 <+215>: mov $0x0,%eax
0x000055555555245 <+220>: jmp 0x5555555527a <main+273>
0x000055555555247 <+222>: addl $0x1,-0x2c(%rbp)
0x00005555555524b <+226>: mov -0x2c(%rbp),%eax
0x00005555555524e <+229>: movslq %eax,%rbx
0x000055555555251 <+232>: mov -0x40(%rbp),%rax
0x000055555555255 <+236>: add $0x8,%rax
0x000055555555259 <+240>: mov (%rax),%rax
0x00005555555525c <+243>: mov %rax,%rdi
0x00005555555525f <+246>: callq 0x55555555040 <strlen@plt>
0x000055555555264 <+251>: cmp %rax,%rbx
0x000055555555267 <+254>: jb 0x5555555520b <main+162>
0x000055555555269 <+256>: lea 0xdb8(%rip),%rdi # 0x555555556028
0x000055555555270 <+263>: callq 0x55555555030 <puts@plt>
0x000055555555275 <+268>: mov $0x0,%eax
0x00005555555527a <+273>: mov -0x18(%rbp),%rcx
0x00005555555527e <+277>: sub %fs:0x28,%rcx
0x000055555555287 <+286>: je 0x5555555528e <main+293>
0x000055555555289 <+288>: callq 0x55555555050 <__stack_chk_fail@plt>
0x00005555555528e <+293>: mov -0x8(%rbp),%rbx
0x000055555555292 <+297>: leaveq
0x000055555555293 <+298>: retq
End of assembler dump.
(gdb) █
```

On remarque de suite la **ligne +184**, la fonction XOR, ce qui nous fait comprendre qu'on est très proche de la fonction if de comparaison. Et en effet, à la **ligne +199** on retrouve la commande **CMP**, qui est la comparaison de notre mot de passe avec la valeur stockée dans le registre. On peut alors mettre un breakpoint à cette adresse pour visualiser la valeur des registres quand le programme arrive à ce point. La commande pour ajouter un breakpoint à ce point est **b *0x000055555555230**.

Note importante : il faut lancer au moins une fois le programme avant d'utiliser la commande disass main, sinon on obtiendra des adresses différentes, comme ceci :

```

Terminal - sergi@ubuntu: ~/Documents/Hacking/1.crackme
File Edit View Terminal Tabs Help
0x0000000000001216 <+173>: mov -0x2c(%rbp),%eax
0x0000000000001219 <+176>: cltq
0x000000000000121b <+178>: add %rdx,%rax
0x000000000000121e <+181>: movzbl (%rax),%eax
0x0000000000001221 <+184>: xor $0x5,%eax
0x0000000000001224 <+187>: mov %eax,%edx
0x0000000000001226 <+189>: mov -0x2c(%rbp),%eax
0x0000000000001229 <+192>: cltq
0x000000000000122b <+194>: movzbl -0x25(%rbp,%rax,1),%eax
0x0000000000001230 <+199>: cmp %al,%dl
0x0000000000001232 <+201>: je 0x1247 <main+222>
0x0000000000001234 <+203>: lea 0xdde(%rip),%rdi # 0x2019
0x000000000000123b <+210>: callq 0x1030 <puts@plt>
--Type <RET> for more, q to quit, c to continue without paging--c
0x0000000000001240 <+215>: mov $0x0,%eax
0x0000000000001245 <+220>: jmp 0x127a <main+273>
0x0000000000001247 <+222>: addl $0x1,-0x2c(%rbp)
0x000000000000124b <+226>: mov -0x2c(%rbp),%eax

```

On peut maintenant lancer le programme avec la commande **r** (ou **run**) suivi du mot de passe. Pour rappel, il faut qu'il s'agisse d'un mot de passe de 12 caractères.

```

Terminal - sergi@ubuntu: ~/Documents/Hacking/1.crackme
File Edit View Terminal Tabs Help
(gdb) b *0x000055555555230
Breakpoint 1 at 0x55555555230
(gdb) run hello12hello
Starting program: /home/sergi/Documents/Hacking/1.crackme/crackme2 hello12hello

Breakpoint 1, 0x000055555555230 in main ()
(gdb) i r
rax            0x70             112
rbx            0x0             0
rcx            0xa             10
rdx            0x6d             109
rsi            0x7fffffff118    140737488347416
rdi            0x7fffffff44a    140737488348234
rbp            0x7fffffff020    0x7fffffff020
rsp            0x7fffffffdf0    0x7fffffffdf0
r8             0x0             0
r9             0x7ffff7fe0d50    140737354009936
r10            0x555555544ee     93824992232686
r11            0x7ffff7f50660    140737353418336
r12            0x55555555070     93824992235632
r13            0x7fffffff110    140737488347408
r14            0x0             0
r15            0x0             0
rip            0x55555555230    0x55555555230 <main+199>
eflags        0x202             [ IF ]
cs            0x33             51
ss            0x2b             43
ds            0x0             0
es            0x0             0
fs            0x0             0
gs            0x0             0
(gdb)
(gdb)

```

Le programme c'est bien arrêté au breakpoint choisi. On peut maintenant voir la valeur contenue dans les registres avec la commande **i r** (ou **info registers**). On remarque que les registres désirés, soit **AL** et **DL**, ne sont pas dans cette liste. Heureusement, gdb nous permet de choisir quel registre on souhaite afficher. La commande **i r al dl** nous affiche les 2 registres recherchés et on remarque qu'ils correspondent à **rax** et **rdx**.

```

(gdb) i r
rax            0x70            112
rbx            0x0             0
rcx            0xa            10
rdx            0x6d            109
rsi            0x7fffffff118    140737488347416
rdi            0x7fffffff44a    140737488348234
rbp            0x7fffffff020    0x7fffffff020
rsp            0x7fffffffdf0    0x7fffffffdf0
r8             0x0             0
r9             0x7ffff7fe0d50    140737354009936
r10            0x5555555544ee    93824992232686
r11            0x7ffff7f50660    140737353418336
r12            0x55555555070    93824992235632
r13            0x7fffffff110    140737488347408
r14            0x0             0
r15            0x0             0
rip            0x555555555230    0x555555555230 <main+199>
eflags        0x202            [ IF ]
cs             0x33            51
ss             0x2b            43
ds             0x0             0
es             0x0             0
fs             0x0             0
gs             0x0             0
(gdb) i r al dl
al            0x70            112
dl            0x6d            109
(gdb)

```

À ce point on sait que notre mot de passe a été soumis à l'opération XOR 5 et que le mot qu'on a rentré commence par **h**, qui dans la table ASCII correspond à 68 en hexadécimal. **0x68 XOR 0x5** nous donne **0x6d**, qui est la valeur contenue dans **dl**. On en déduit que **al** est le registre contenant le mot de passe et que **0x70** est la première lettre du mot de passe à laquelle il faut appliquer le XOR, donc la première lettre est dans ce cas **0x75**, soit **u**.

Après avoir exécuté à nouveau le programme avec un mot commençant par **u**, on vérifie à nouveau la valeur des registres et on peut voir comme la lettre **u** c'est bien celle correcte. Avec la commande **c** (ou **continue**) on peut aller à l'étape suivante de la boucle **while** dans laquelle on se trouve et on remarque que la prochaine lettre demandée est 0x63, soit **f** dans la table ASCII après y avoir appliqué le XOR.


```

Terminal - sergi@ubuntu: ~/Documents/Hacking/1.crackme
File Edit View Terminal Tabs Help

(gdb) run uhellohello
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/sergi/Documents/Hacking/1.crackme/crackme2 uhellohello

Breakpoint 1, 0x0000555555555230 in main ()
(gdb) i r al dl
al      0x70      112
dl      0x70      112
(gdb) c
Continuing.

Breakpoint 1, 0x0000555555555230 in main ()
(gdb) i r al dl
al      0x63      99
dl      0x6d      109
(gdb)

```

On peut également utiliser la fonction **set** pour changer la valeur des registres et n'avoir pas à exécuter le code dès le début à chaque étape. De cette manière, on obtient le résultat suivant :

```

Terminal - sergi@ubuntu: ~/Documents/Hacking/1.crackme
File Edit View Terminal Tabs Help

(gdb) r ufhellohello
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/sergi/Documents/Hacking/1.crackme/crackme2 ufhellohello

Breakpoint 1, 0x0000555555555230 in main ()
(gdb) i r al dl
al      0x70      112
dl      0x70      112
(gdb) c
Continuing.

Breakpoint 1, 0x0000555555555230 in main ()
(gdb) i r al dl
al      0x63      99
dl      0x63      99
(gdb) c
Continuing.

Breakpoint 1, 0x0000555555555230 in main ()
(gdb) i r al dl
al      0x6a      106
dl      0x6d      109
(gdb) set $dl = 0x6a
(gdb) c
Continuing.

Breakpoint 1, 0x0000555555555230 in main ()
(gdb) i r al dl
al      0x70      112
dl      0x60      96
(gdb) set $dl = 0x70
(gdb) c
Continuing.

Breakpoint 1, 0x0000555555555230 in main ()
(gdb) i r al dl
al      0x6b      107
dl      0x69      105
(gdb) set $dl = 0x6b
(gdb) c
Continuing.

Breakpoint 1, 0x0000555555555230 in main ()
(gdb) i r al dl
al      0x61      97
dl      0x69      105
(gdb) set $dl = 0x61
(gdb) c
Continuing.

Breakpoint 1, 0x0000555555555230 in main ()
(gdb) i r al dl
al      0x6c      108
dl      0x6a      106
(gdb) set $dl = 0x6c
(gdb) c
Continuing.

Breakpoint 1, 0x0000555555555230 in main ()
(gdb) i r al dl
al      0x71      113
dl      0x6d      109
(gdb) set $dl = 0x71
(gdb) c
Continuing.

Breakpoint 1, 0x0000555555555230 in main ()
(gdb) i r al dl
al      0x37      55
dl      0x60      96
(gdb) set $dl = 0x37
(gdb) c
Continuing.

Breakpoint 1, 0x0000555555555230 in main ()
(gdb) i r al dl
al      0x35      53
dl      0x69      105
(gdb) set $dl = 0x35
(gdb) c
Continuing.

Breakpoint 1, 0x0000555555555230 in main ()
(gdb) i r al dl
al      0x37      55
dl      0x69      105
(gdb) set $dl = 0x37
(gdb) c
Continuing.

Breakpoint 1, 0x0000555555555230 in main ()
(gdb) i r al dl
al      0x34      52
dl      0x6a      106
(gdb) set $dl = 0x34
(gdb) c
Continuing.
Congratulations
[Inferior 1 (process 4696) exited normally]
(gdb)

```

On a donc nos 12 lettres qui en hexadécimal sont 0x70, 0x63, 0x6a, 0x70, 0x6b, 0x61, 0x6c, 0x71, 0x37, 0x35, 0x37, 0x34 et selon la table ASCII, elles correspondent à **pcjpkalq7574**. Après avoir appliqué le XOR, on obtient le mot **ufoundit2021**.

Essayons :

```
Terminal - sergi@ubuntu: ~/Documents/Hacking/1.crackme
File Edit View Terminal Tabs Help
sergi@ubuntu:~/Documents/Hacking/1.crackme$ ./crackme2 ufoundit2021
Congratulations
sergi@ubuntu:~/Documents/Hacking/1.crackme$
```

Le mot de passe de crackme2 est bien **ufoundit2021**.

3. CrackMe 3

Pour le troisième fichier on sait que le mot de passe est généré lors de l'exécution. On va donc commencer par analyser l'exécutable avec Ghidra :

```
Decompile: main - (crackme3)
1
2 undefined8 main(int param_1,undefined8 *param_2)
3
4 {
5     char cVar1;
6     undefined8 uVar2;
7     size_t sVar3;
8     long in_FS_OFFSET;
9     int local_5c;
10    undefined8 local_58;
11    undefined8 local_50;
12    undefined8 local_48;
13    undefined8 local_40;
14    undefined8 local_38;
15    undefined8 local_30;
16    undefined4 local_28;
17    undefined2 local_24;
18    long local_20;
19
20    local_20 = *(long *) (in_FS_OFFSET + 0x28);
21    if (param_1 == 2) {
22        local_58 = 0x6867666564636261;
23        local_50 = 0x737271706e6d6b6a;
24        local_48 = 0x417a797877767574;
25        local_40 = 0x4a48474645444342;
26        local_38 = 0x54535251504e4d4b;
27        local_30 = 0x33325a5958575655;
28        local_28 = 0x37363534;
29        local_24 = 0x38;
30        local_5c = 0;
31        while (local_5c < 10) {
32            cVar1 = *(char *) ((long) local_5c + param_2[1]);
33            sVar3 = strlen((char *) &local_58);
34            if (cVar1 != *(char *) ((long) &local_58 + (ulong)(long)(local_5c * 0x4d + 3) % sVar3)) {
35                puts("Wrong Password");
36                uVar2 = 1;
37                goto LAB_0010129a;
38            }
39            local_5c = local_5c + 1;
40        }
41        puts("Congratulations");
42        uVar2 = 0;
43    }
44    else {
45        printf("Usage : %s password\n", param_2);
46        uVar2 = 0;
47    }
48    LAB_0010129a:
49    if (local_20 != *(long *) (in_FS_OFFSET + 0x28)) {
50        /* WARNING: Subroutine does not return */
51        __stack_chk_fail();
52    }
53    return uVar2;
54 }
```

Le code est similaire à celui d'avant :

- Ligne 31 : on déduit que le mot de passe doit être d'au moins 10 caractères ;
- Ligne 34 : chaque caractère du mot passé en paramètre (**cVar1**) est comparé avec des valeurs calculés sur la base des variables **local_58**, **local_5c**, **sVar3**.

On remarque que le mot que l'on passe en paramètre reste intacte, il n'est pas modifié en cours d'exécution comme c'était le cas avant.

On peut lancer **GDB** et voir le code en assembleur :

```
Terminal - sergi@ubuntu: ~/Documents/Hacking/1.crackme
File Edit View Terminal Tabs Help
0x00005555555516a <+1>: mov    %rsp,%rbp
0x00005555555516d <+4>: push  %r12
0x00005555555516f <+6>: push  %rbx
0x000055555555170 <+7>: sub    $0x60,%rsp
0x000055555555174 <+11>: mov    %edi,-0x64(%rbp)
0x000055555555177 <+14>: mov    %rsi,-0x70(%rbp)
0x00005555555517b <+18>: mov    %fs:0x28,%rax
0x000055555555184 <+27>: mov    %rax,-0x18(%rbp)
0x000055555555188 <+31>: xor    %eax,%eax
0x00005555555518a <+33>: cmpl   $0x2,-0x64(%rbp)
0x00005555555518e <+37>: je     0x555555551b5 <main+76>
0x000055555555190 <+39>: mov    -0x70(%rbp),%rax
0x000055555555194 <+43>: mov    (%rax),%rax
0x000055555555197 <+46>: mov    %rax,%rsi
0x00005555555519a <+49>: lea    0xe63(%rip),%rdi    # 0x555555556004
0x0000555555551a1 <+56>: mov    $0x0,%eax
0x0000555555551a6 <+61>: callq  0x55555555000 <printf@plt>
0x0000555555551ab <+66>: mov    $0x0,%eax
0x0000555555551b0 <+71>: jmpq   0x5555555529a <main+305>
0x0000555555551b5 <+76>: movabs $0x6867666564636261,%rax
0x0000555555551bf <+86>: movabs $0x737271706e6d6b6a,%rdx
0x0000555555551c9 <+96>: mov    %rax,-0x50(%rbp)
0x0000555555551cd <+100>: mov    %rdx,-0x48(%rbp)
0x0000555555551d1 <+104>: movabs $0x417a797877767574,%rax
0x0000555555551db <+114>: movabs $0x4a48474645444342,%rdx
0x0000555555551e5 <+124>: mov    %rax,-0x40(%rbp)
0x0000555555551e9 <+128>: mov    %rdx,-0x38(%rbp)
0x0000555555551ed <+132>: movabs $0x54535251504e4d4b,%rax
0x0000555555551f7 <+142>: movabs $0x33325a5958575655,%rdx
0x000055555555201 <+152>: mov    %rax,-0x30(%rbp)
0x000055555555205 <+156>: mov    %rdx,-0x28(%rbp)
0x000055555555209 <+160>: movl   $0x37363534,-0x20(%rbp)
0x000055555555210 <+167>: movw   $0x38,-0x1c(%rbp)
0x000055555555216 <+173>: movl   $0x0,-0x54(%rbp)
0x00005555555521d <+180>: jmp    0x55555555203 <main+202>
0x00005555555521f <+182>: mov    -0x70(%rbp),%rax
0x000055555555223 <+186>: add    $0x0,%rax
0x000055555555227 <+190>: mov    (%rax),%rdx
0x00005555555522a <+193>: mov    -0x54(%rbp),%eax
0x00005555555522d <+196>: cltq   %rdx,%rax
0x00005555555522f <+198>: add    %rdx,%rax
0x000055555555232 <+201>: movzbl (%rax),%r12d
0x000055555555236 <+205>: mov    -0x54(%rbp),%eax
0x000055555555239 <+208>: imul   $0x4d,%eax,%eax
0x00005555555523c <+211>: add    $0x3,%eax
0x00005555555523f <+214>: movslq %eax,%rbx
0x000055555555242 <+217>: lea    -0x50(%rbp),%rax
0x000055555555246 <+221>: mov    %rax,%rdi
0x000055555555249 <+224>: callq  0x55555555040 <strlen@plt>
0x00005555555524e <+229>: mov    %rax,%rsi
0x000055555555251 <+232>: mov    %rbx,%rax
0x000055555555254 <+235>: mov    $0x0,%edx
0x000055555555259 <+240>: div    %rsi
0x00005555555525c <+243>: mov    %rdx,%rcx
0x00005555555525f <+246>: mov    %rcx,%rax
0x000055555555262 <+249>: movzbl -0x50(%rbp,%rax,1),%eax
0x000055555555267 <+254>: cmp    %al,%r12b
0x00005555555526a <+257>: je     0x5555555527f <main+278>
0x00005555555526c <+259>: lea    0xda6(%rip),%rdi    # 0x555555556019
0x000055555555273 <+266>: callq  0x55555555030 <puts@plt>
0x000055555555278 <+271>: mov    $0x1,%eax
0x00005555555527d <+276>: jmp    0x5555555529a <main+305>
0x00005555555527f <+278>: addl   $0x1,-0x54(%rbp)
0x000055555555283 <+282>: cmpl   $0x9,-0x54(%rbp)
0x000055555555287 <+286>: jle    0x5555555521f <main+182>
0x000055555555289 <+288>: lea    0xd98(%rip),%rdi    # 0x555555556028
0x000055555555290 <+295>: callq  0x55555555030 <puts@plt>
0x000055555555295 <+300>: mov    $0x0,%eax
0x00005555555529a <+305>: mov    -0x18(%rbp),%rcx
0x00005555555529e <+309>: sub    %fs:0x28,%rcx
0x0000555555552a7 <+318>: je     0x555555552ae <main+325>
0x0000555555552a9 <+320>: callq  0x55555555050 <__stack_chk_fail@plt>
0x0000555555552ae <+325>: add    $0x60,%rsp
0x0000555555552b2 <+329>: pop    %rbx
0x0000555555552b3 <+330>: pop    %r12
0x0000555555552b5 <+332>: pop    %rbp
0x0000555555552b6 <+333>: retq
--Type <RET> for more, q to quit, c to continue without paging--c
End of assembler dump.
(gdb)
```

Pour nous aider à trouver plus rapidement la ligne de code sur laquelle on va mettre un breakpoint, on peut se servir de Ghidra, qui permet d'afficher le correspondant assembleur du code à page 10.

En cliquant sur la variable **cVar1**, Ghidra met en évidence le code assembleur **CMP R12B, AL** :

```

25  local_40 = 0x4a48474645444342;
26  local_38 = 0x54535251504e4d4b;
27  local_30 = 0x33325a5958575655;
28  local_28 = 0x37363534;
29  local_24 = 0x38;
30  local_5c = 0;
31  while (local_5c < 10) {
32      cVar1 = *(char *)((long)local_5c + param_2[1]);
33      sVar3 = strlen((char *)&local_58);
34      if (cVar1 != *(char *)((long)&local_58 + (ulong)(long)(local_5c * 0x4d + 3) % sVar3)) {
35          puts("Wrong Password");
36          uVar2 = 1;
37          goto LAB_0010129a;
38      }
39      local_5c = local_5c + 1;
40  }

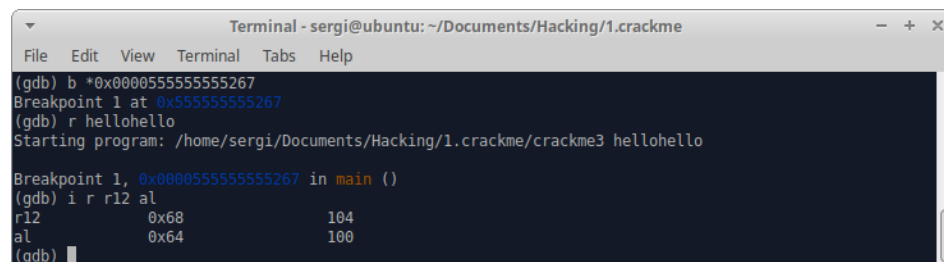
```


00 00			
00101259	48 f7 f6	DIV	RSI
0010125c	48 89 d1	MOV	RCX, RDX
0010125f	48 89 c8	MOV	RAX, RCX
00101262	0f b6 44	MOVZX	EAX, byte ptr [RBP + RAX*0x1 + -0x50]
05 b0			
00101267	41 38 c4	CMP	R12B, AL
0010126a	74 13	JZ	LAB_0010127f
0010126c	48 8d 3d	LEA	RDI, [s_Wrong_Password_00102019]
a6 0d 00 00			= "Wrong Password"
00101273	e8 b8 fd	CALL	puts
ff ff			int puts(char * __s)
00101278	b8 01 00	MOV	EAX, 0x1
00 00			
0010127d	eb 1b	JMP	LAB_0010129a

Avec cette information, on va chercher cette ligne dans **gdb** et on y met un breakpoint :

b *0x05555555555267

On exécute le code avec **r** et puis on affiche les registres **R12** et **AL** :



```

Terminal - sergi@ubuntu: ~/Documents/Hacking/1.crackme
File Edit View Terminal Tabs Help
(gdb) b *0x0000555555555267
Breakpoint 1 at 0x555555555267
(gdb) r hellohello
Starting program: /home/sergi/Documents/Hacking/1.crackme/crackme3 hellohello

Breakpoint 1, 0x0000555555555267 in main ()
(gdb) i r r12 al
r12      0x68      104
al       0x64      100
(gdb)

```

La lettre recherchée est **0x64** puisque on sait que **0x68** correspond à **h** en code ASCII. On répète l'opération tant qu'on arrive au message de félicitations.

```

Terminal - sergi@ubuntu: ~/Documents/Hacking/1.crackme
File Edit View Terminal Tabs Help

0x000055555552a9 <+320>: callq 0x55555555050 <__stack_chk_fail@plt>
0x000055555552ae <+325>: add $0x60,%rsp
0x000055555552b2 <+329>: pop %rbx
0x000055555552b3 <+330>: pop %r12
0x000055555552b5 <+332>: pop %rbp
0x000055555552b6 <+333>: retq
--Type <RET> for more, q to quit, c to continue without paging--c
End of assembler dump.
(gdb) b *0x000055555555267
Breakpoint 1 at 0x55555555267
(gdb) r hellohello
Starting program: /home/sergi/Documents/Hacking/1.crackme/crackme3 hellohello

Breakpoint 1, 0x000055555555267 in main ()
(gdb) i r r12 al
r12      0x68      104
al       0x64      100
(gdb) set $r12=0x64
(gdb) c
Continuing.

Breakpoint 1, 0x000055555555267 in main ()
(gdb) i r r12 al
r12      0x65      101
al       0x45      69
(gdb) set $r12=0x45
(gdb) c
Continuing.

Breakpoint 1, 0x000055555555267 in main ()
(gdb) i r r12 al
r12      0x6c      108
al       0x37      55
(gdb) set $r12=0x37
(gdb) c
Continuing.

Breakpoint 1, 0x000055555555267 in main ()
(gdb) i r r12 al
r12      0x6c      108
al       0x7a      122
(gdb) set $r12=0x7a
(gdb) c
Continuing.

Breakpoint 1, 0x000055555555267 in main ()
(gdb) i r r12 al
r12      0x6f      111
al       0x32      50
(gdb) set $r12=0x32
(gdb) c
Continuing.

(gdb) c
Continuing.

Breakpoint 1, 0x000055555555267 in main ()
(gdb) i r r12 al
r12      0x68      104
al       0x75      117
(gdb) set $r12=0x75
(gdb) c
Continuing.

Breakpoint 1, 0x000055555555267 in main ()
(gdb) i r r12 al
r12      0x65      101
al       0x56      86
(gdb) set $r12=0x56
(gdb) c
Continuing.

Breakpoint 1, 0x000055555555267 in main ()
(gdb) i r r12 al
r12      0x6c      108
al       0x70      112
(gdb) set $r12=0x70
(gdb) c
Continuing.

Breakpoint 1, 0x000055555555267 in main ()
(gdb) i r r12 al
r12      0x6c      108
al       0x51      81
(gdb) set $r12=0x51
(gdb) c
Continuing.

Breakpoint 1, 0x000055555555267 in main ()
(gdb) i r r12 al
r12      0x6f      111
al       0x68      104
(gdb) set $r12=0x68
(gdb) c
Continuing.
Congratulations
[Inferior 1 (process 5208) exited normally]
(gdb)

```

Les lettres rentrées sont 0x64, 0x45, 0x37, 0x7a, 0x32, 0x75, 0x56, 0x70, 0x51, 0x68, soit **dE7z2uVpQh**.

On teste :

```

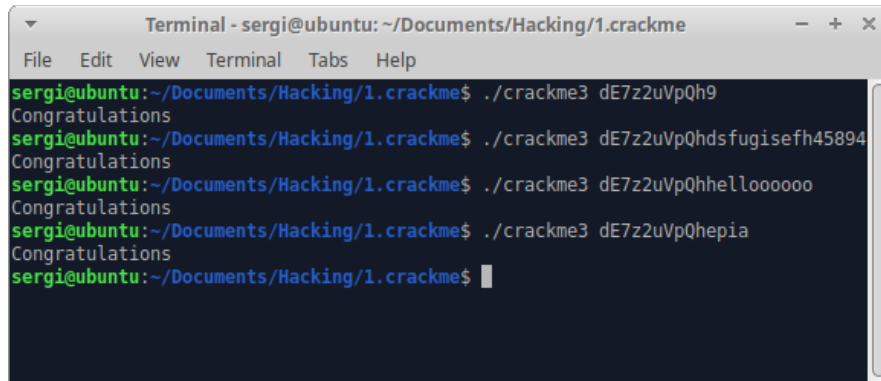
Terminal - sergi@ubuntu: ~/Documents/Hacking/1.crackme
File Edit View Terminal Tabs Help

sergi@ubuntu:~/Documents/Hacking/1.crackme$ ./crackme3 dE7z2uVpQh
Congratulations
sergi@ubuntu:~/Documents/Hacking/1.crackme$

```

Le mot de passe de crackme3 est bien **dE7z2uVpQh**.

Au début on a dit que le mot de passe devait être d'au moins 10 caractères, donc que se passe t'il s'il fait 11 caractères par exemple ? Voyons :



```
Terminal - sergi@ubuntu: ~/Documents/Hacking/1.crackme
File Edit View Terminal Tabs Help
sergi@ubuntu:~/Documents/Hacking/1.crackme$ ./crackme3 dE7z2uVpQh9
Congratulations
sergi@ubuntu:~/Documents/Hacking/1.crackme$ ./crackme3 dE7z2uVpQhdsfugisefh45894
Congratulations
sergi@ubuntu:~/Documents/Hacking/1.crackme$ ./crackme3 dE7z2uVpQhhelloooooo
Congratulations
sergi@ubuntu:~/Documents/Hacking/1.crackme$ ./crackme3 dE7z2uVpQhepia
Congratulations
sergi@ubuntu:~/Documents/Hacking/1.crackme$
```

On remarque quelque chose d'intéressant : on peut écrire tout ce que l'on veut après le mot de passe et il sera toujours accepté.

Conclusions

On a pu voir comment découvrir des mots de passes écrits dans des fichiers en utilisant des outils assez simples et accessibles à tous. Ceci nous fait comprendre que stocker des mots de passes à l'intérieur des programmes n'est pas une bonne pratique de sécurité, puisqu'ils peuvent être trouvés assez facilement.

Une possible solution est de hasher le mot de passe et le stocker dans un fichier séparé. Ajouter une authentification à 2 facteurs rendrait aussi l'accès au programme plus difficile à outrepasser. Un mot de passe jetable (utilisable qu'une seule fois) peut aussi être une autre alternative.