

Rapport Exercices Sécurité des Applications : Série 1

Thomas Dagier

September, 16, 2020

1 CrackMe1 (CrM1)

Pour le premier fichier, on peut observer une indication nous permettant de cracker le fichier : le mot de passe est "hardcodé" dans le programme. Je me doute donc qu'il faudra d'une manière ou d'une autre afficher le contenu du fichier. J'ai donc essayé la commande : `nano crackme1`

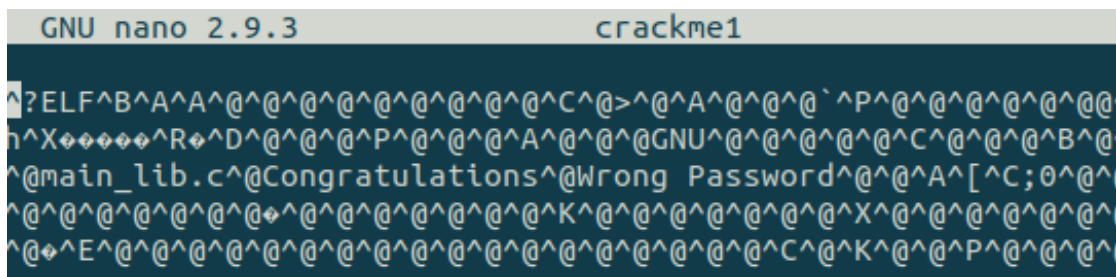


Figure 1: utilisation de la commande "nano" sur le fichier crackme1

Ici, on voit déjà les informations qui sont "hardcodées" dans le fichier. On a donc une première intuition sur le mot de passe que l'on va pouvoir confirmer.

En lisant la documentation, j'ai remarqué que l'on pouvait utiliser "ltrace". Cet outil permet d'identifier les appels aux bibliothèques externes faits depuis le programme comme `<string.h>`. J'essaye donc la commande : `ltrace ./crackme1 password`.

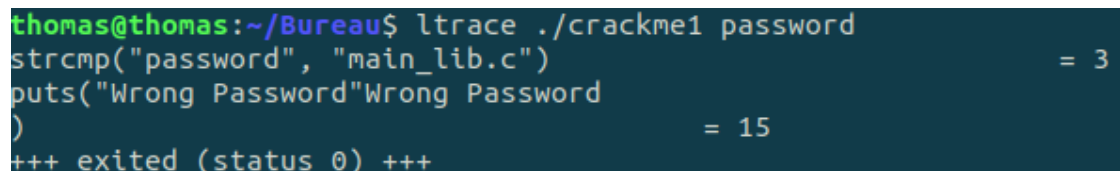


Figure 2: utilisation de la commande "ltrace" sur le fichier crackme1

On observe alors que le mot de passe donné en paramètre est comparé avec une chaîne de caractère qui est : "main_lib.c".

On essaye donc d'exécuter le fichier avec le potentiel mot de passe et on observe que tout marche:

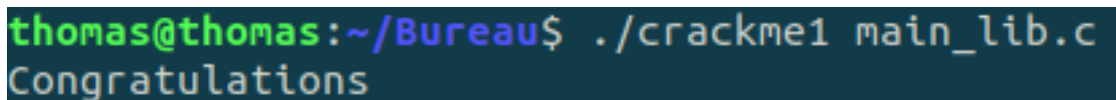


Figure 3: exécution du fichier avec le mot de passe trouvé

Ici, le mot de passe est écrit ainsi, de sorte que l'on ne puisse pas aisément détecter le mot de passe en appliquant la commande : `strings crackme1`.

2 CrackMe2 (CrM2)

Cette fois-ci, si on essaye de faire la commande "nano" sur le fichier, on voit que rien n'est hardcodé dans le programme.

On doit donc utiliser un outil différent : "gdb". Pour cela, après avoir installé les packets gdb, on utilise la commande : `gdb crackme2` . En suivant le tutoriel disponible, on apprend que la commande : `disass main` permet de désassembler le binaire du fichier et comprendre quelle est sa structure.

```
0x00000000000001169 <+0>:    push    %rbp
0x0000000000000116a <+1>:    mov     %rsp,%rbp
0x0000000000000116d <+4>:    push    %rbx
0x0000000000000116e <+5>:    sub     $0x38,%rsp
0x00000000000001172 <+9>:    mov     %edi,-0x34(%rbp)
0x00000000000001175 <+12>:   mov     %rsi,-0x40(%rbp)
0x00000000000001179 <+16>:   mov     %fs:0x28,%rax
0x00000000000001182 <+25>:   mov     %rax,-0x18(%rbp)
0x00000000000001186 <+29>:   xor     %eax,%eax
0x00000000000001188 <+31>:   cmpl    $0x2,-0x34(%rbp)
0x0000000000000118c <+35>:   je      0x11b3 <main+74>
0x0000000000000118e <+37>:   mov     -0x40(%rbp),%rax
0x00000000000001192 <+41>:   mov     (%rax),%rax
0x00000000000001195 <+44>:   mov     %rax,%rsi
0x00000000000001198 <+47>:   lea     0xe65(%rip),%rdi      # 0x2004
0x0000000000000119f <+54>:   mov     $0x0,%eax
0x000000000000011a4 <+59>:   callq   0x1060 <printf@plt>
0x000000000000011a9 <+64>:   mov     $0x0,%eax
0x000000000000011ae <+69>:   jmpq    0x127c <main+275>
0x000000000000011b3 <+74>:   mov     -0x40(%rbp),%rax
0x000000000000011b7 <+78>:   add     $0x8,%rax
0x000000000000011bb <+82>:   mov     (%rax),%rax
0x000000000000011be <+85>:   mov     %rax,%rdi
0x000000000000011c1 <+88>:   callq   0x1040 <strlen@plt>
0x000000000000011c6 <+93>:   cmp     $0xd,%rax
```

Figure 4: exécution de la commande `disass main`

La première chose que j'ai fait est de regarder les fonctions assembleur pour voir ce qu'il se passait dans les grandes lignes. La fonction "cmp" (dernière ligne de la figure 4) semble comparer la valeur 0x13 avec un registre. En cherchant un peu, j'ai remarqué que le contenu de \$rax n'était pas le même si je testais un autre mot de passe. La conclusion de ceci est que \$rax stocke la taille du mot de passe donné au programme. Si la taille est bonne (donc 13 caractères), on peut accéder à la suite du programme, sinon, le programme s'arrête avec le message :

```
(gdb) run password
Starting program: /home/thomas/Bureau/crackme2 password
Wrong Password
[Inferior 1 (process 4027) exited normally]
```

Figure 5: exécution du programme avec un mauvais mot de passe

Sachant que la fonction "cmp" m'avait mis sur la bonne piste, j'ai continué de chercher les autres fonctions "cmp" en y mettant des breakpoints aux adresses qui correspondent :

```
0x000055555555232 <+201>:    cmp    %al,%dl
```

Figure 6: adresse qui semble intéressante pour appliquer un breakpoint

```
(gdb) b *0x000055555555232
Breakpoint 1 at 0x55555555232
```

Figure 7: application d'un breakpoint

On peut ensuite utiliser la commande : `run password12345` . Il ne faut pas oublier que le mot de passe doit être composé de 13 caractères pour atteindre au moins une fois le breakpoint. En appuyant sur "enter", on peut passer à l'instruction suivante et on continue tant que le breakpoint n'est pas atteint. Une fois atteint, on peut utiliser la commande : `info registers` ce qui permet de voir les valeurs qui sont actuellement dans les registres.

```
Breakpoint 1, 0x000055555555232 in main ()
(gdb) info registers
rax                0x64          100
rbx                0x1           1
rcx                0x11          17
rdx                0x64          100
rsi                0x7fffffffdec8 140737488346824
rdi                0x7fffffffef260 140737488347744
rbp                0x7fffffffddd0 0x7fffffffddd0
rsp                0x7fffffffdd90 0x7fffffffdd90
r8                 0x0           0
r9                 0x7ffff7fdffa0 140737354006432
r10                0x7ffff7fef460 140737354069088
r11                0x7ffff7f4b170 140737353396592
r12                0x55555555070 93824992235632
r13                0x0           0
r14                0x0           0
r15                0x0           0
rip                0x55555555232 0x55555555232 <main+201>
eflags             0x202      [ IF ]
cs                 0x33          51
ss                 0x2b          43
ds                 0x0           0
es                 0x0           0
fs                 0x0           0
gs                 0x0           0
```

Figure 8: affichage du contenu des registres

En cherchant un moment, j'ai remarqué que les registres `rax` et `rdx` correspondent au `%al` et `%dl` qui sont ceux comparés à la ligne 201 du `main` (endroit où le breakpoint a été mis). Pour remarquer cela, j'ai fait tourner le programme en changeant le mot de passe et en affichant très régulièrement les registres.

Ainsi, j'ai vu que les données du registre `rax` changent lorsque le mot de passe donné change alors que les informations du registre `rdx` ne changent pas. On peut donc en conclure que le mot de passe donné est comparé caractère par caractère au mot de passe attendu. Si la comparaison du premier caractère est bonne, on passe à la comparaison du caractère suivant, sinon, on quitte en affichant que le mot de passe donné n'est pas bon.

J'avais donc la certitude que pour trouver le mot de passe, il faut faire correspondre chaque valeur du registre rax avec celle du registre rdx . La seule solution que j'ai trouvé est de tester un peu tout les caractères jusqu'à trouver une correspondance. De cette manière, une fois qu'on a trouvé le premier caractère, on peut chercher une correspondance pour le second et ainsi de suite pour les 13 caractères.

Il m'a fallut une vingtaine de minutes pour déterminer les 4 premiers caractères : San<space> Et en testant quelques idées, je suis tombé sur le bon mot de passe qui est : San Francisco . Relativement évident car peu de mots commencent par San ...

```
thomas@thomas:~/Bureau$ ./crackme2 San\ Francisco
Congratulations
```

Figure 9: test du programme avec le bon mot de passe

3 CrackMe3 (CrM3)

Pour ce troisième exercice, j'ai fait le choix d'utiliser ghidra de sorte à découvrir une nouvelle méthode pour cracker le mot de passe (et aussi car mes collègues qui ont utilisé cet outil pour le crackme2 avaient l'air de trouver cela beaucoup plus simple avec ghidra).

La première étape a donc été de décompiler le "main" en passant par l'interface de ghidra :

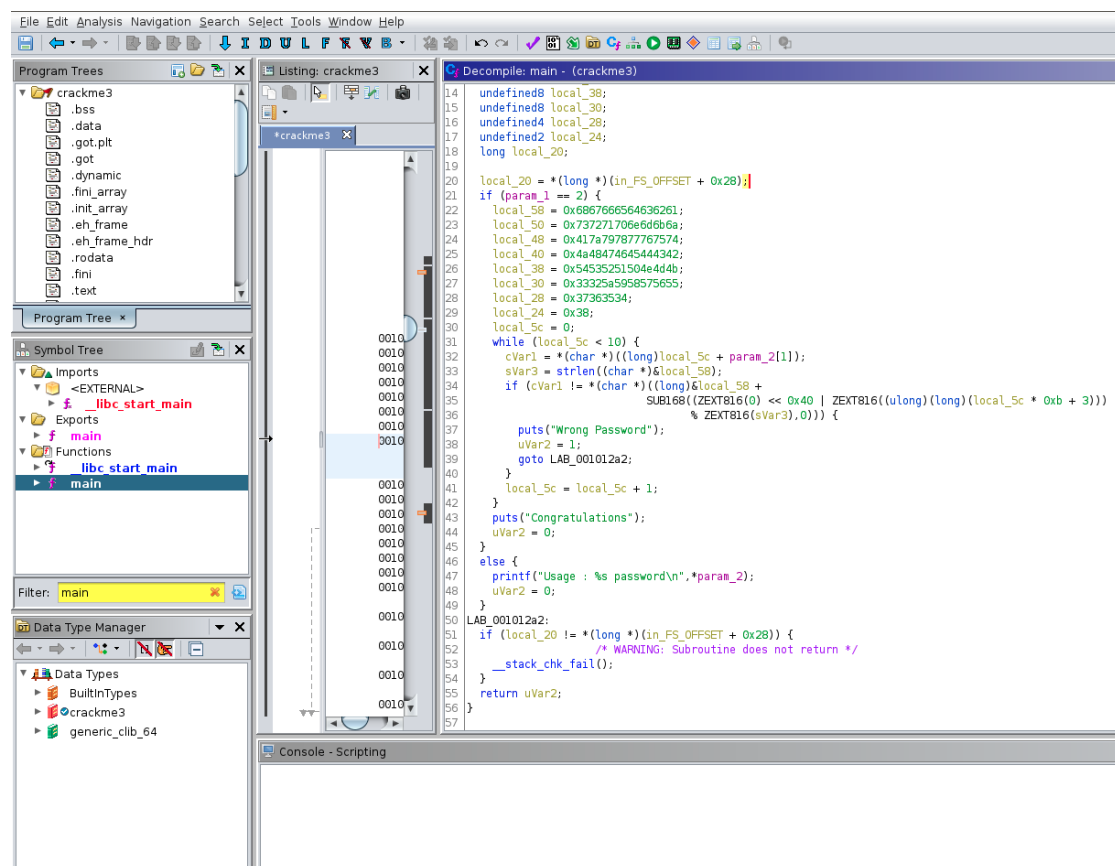


Figure 10: affichage du main avec ghidra

On remarque que le binaire qui a été donné est très bien convertit de sorte que l'on puisse comprendre ce qui se passe lors de l'exécution.

En essayant de comprendre un peu le code, on arrive à deviner que notre mot de passe doit faire 10 caractères (condition de la boucle while). On sait aussi que `cVar1` correspond au caractère courant du mot de passe que l'on test. Ce dernier sera comparé à la valeur ASCII d'un autre caractère qui est contenu dans la variable `local58` ou suivante.

Mais alors pour trouver quelle valeur ASCII est sélectionnée parmi toutes celles disponibles, on doit résoudre un calcul mathématique assez simple. La valeur ASCII du caractère à comparer (donc un des caractères du mot de passe à trouver) se situe à la position :

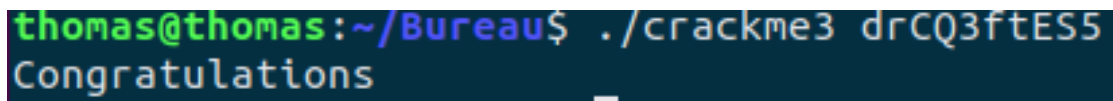
$$\text{value} = (\text{local_5c} * 11 + 3) \bmod \text{sVar3}.$$

Avec `value`, le décalage à faire depuis le premier byte de `local_58`, `local_5c` l'indice du caractère qui est testé (de 0 à 9) et `sVar3` le nombre de bytes contenu entre `local_58` et `local_24` soit 53 bytes.

Pour le premier caractère, `local_5c` = 0 donc `value` = $(0 * 11 + 3) \bmod 53 = 3$. La valeur ASCII testée se situe à l'indice 3 soit 0x64 puisque l'on commence toujours à l'indice 0. En ASCII 0x64 correspond à la lettre `d` que l'on estime être la première lettre du mot de passe.

Pour le second, `local_5c` = 1 donc `value` = $(1 * 11 + 3) \bmod 53 = 14$. On prend le byte relatif au 14eme indice soit 0x72 qui correspond à la lettre `r`.

Et on continue ainsi de suite pour les 8 autres caractères jusqu'à obtenir le mot de passe :

A terminal window with a dark blue background. The prompt is 'thomas@thomas:~/Bureau\$' in green and blue. The command './crackme3 drCQ3ftES5' is entered in yellow. The output 'Congratulations' is shown in yellow on the next line. A white cursor is visible at the end of the command line.

```
thomas@thomas:~/Bureau$ ./crackme3 drCQ3ftES5
Congratulations
```

Figure 11: test du crackme3 avec le mot de passe trouvé