

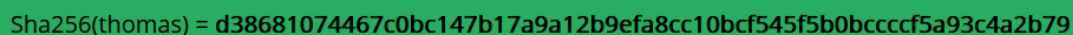
Rapport Exercices Sécurité des Applications : Série 4 - suite

Thomas Dagier

October, 31, 2020

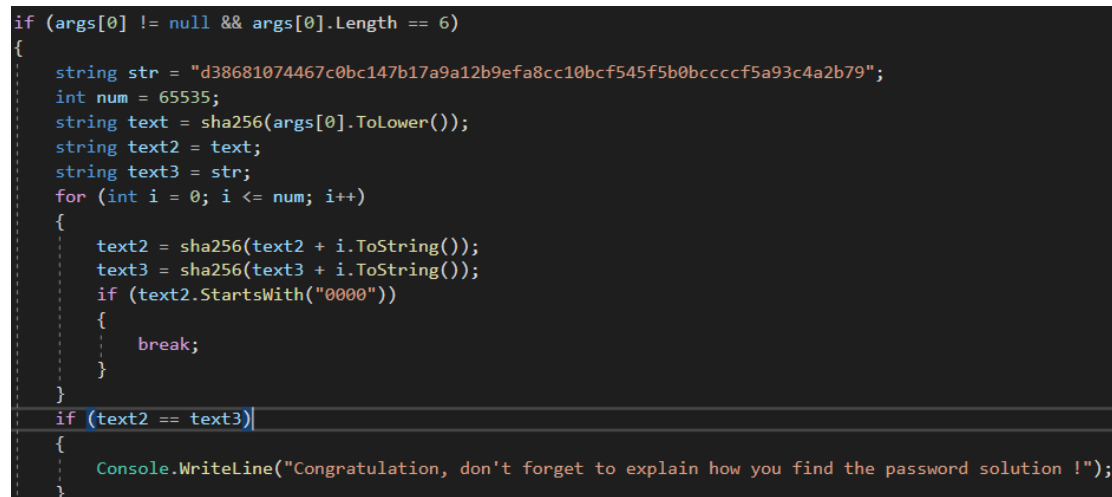
1 Signature 1 (SIG1)

Dans cet exercice, nous devons signer le CrackMe4 avec un certificat. La première étape est de modifier le binaire de sorte que le mot de passe à rentrer soit: `thomas`. Pour cela, on refait la manipulation avec ILSpy puis Visual Studio et on modifie le hash du mot de passe attendu.



Sha256(thomas) = d38681074467c0bc147b17a9a12b9efa8cc10bcf545f5b0bccccf5a93c4a2b79

Figure 1: récupération du hash de `thomas`

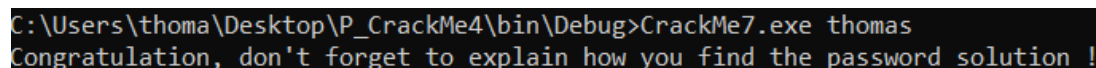


```
if (args[0] != null && args[0].Length == 6)
{
    string str = "d38681074467c0bc147b17a9a12b9efa8cc10bcf545f5b0bccccf5a93c4a2b79";
    int num = 65535;
    string text = sha256(args[0].ToLower());
    string text2 = text;
    string text3 = str;
    for (int i = 0; i <= num; i++)
    {
        text2 = sha256(text2 + i.ToString());
        text3 = sha256(text3 + i.ToString());
        if (text2.StartsWith("0000"))
        {
            break;
        }
    }
    if (text2 == text3)
    {
        Console.WriteLine("Congratulation, don't forget to explain how you find the password solution !");
    }
}
```

Figure 2: modification du code

La variable `text3` qui correspond au mot de passe attendu, n'est plus une concaténation d'un hash hardcodé et d'une description mais seulement du hash obtenu sur internet pour le mot `thomas`. Il ne faut pas non plus oublier de modifier la longueur du mot de passe attendu qui est cette fois-ci de 6.

Une fois ceci fait, on teste le binaire pour vérifier que le code fonctionne bien :



```
C:\Users\thoma\Desktop\P_CrackMe4\bin\Debug>CrackMe7.exe thomas
Congratulation, don't forget to explain how you find the password solution !
```

Figure 3: vérification de l'exécution

Puisque tout fonctionne, on peut continuer en signant le binaire. Pour cela, on utilise l'outil SignTool de Microsoft et XCA pour la création de certificat.

On commence par créer le certificat avec XCA. Une fois la base de données créée, on peut ajouter notre nouveau certificat. Comme c'est le premier certificat que l'on fait, on doit générer une AC.

Figure 4: Création de l'AC puis du certificat

Figure 5: Ajout des informations et de la nouvelle clé

Une fois toutes les informations ajoutées avec succès, on peut cliquer sur OK et retourner dans la page d'accueil de XCA. On observe alors que le nouveau certificat a été ajouté correctement :

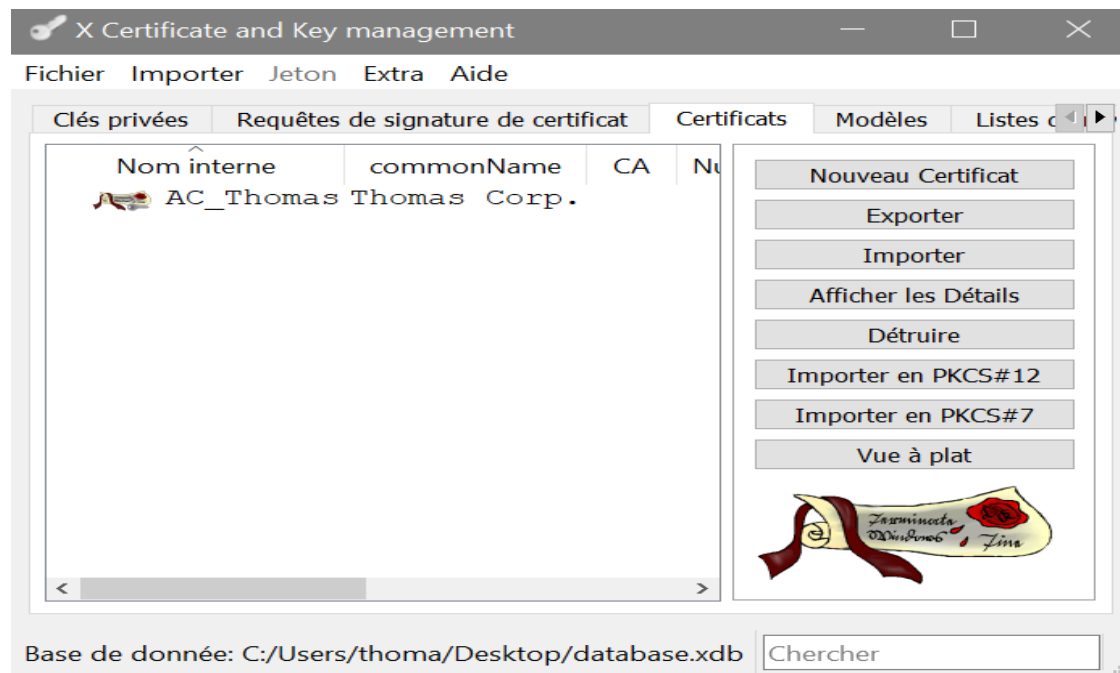


Figure 6: Affichage du nouveau certificat

Le certificat étant créé on peut désormais l'exporter :

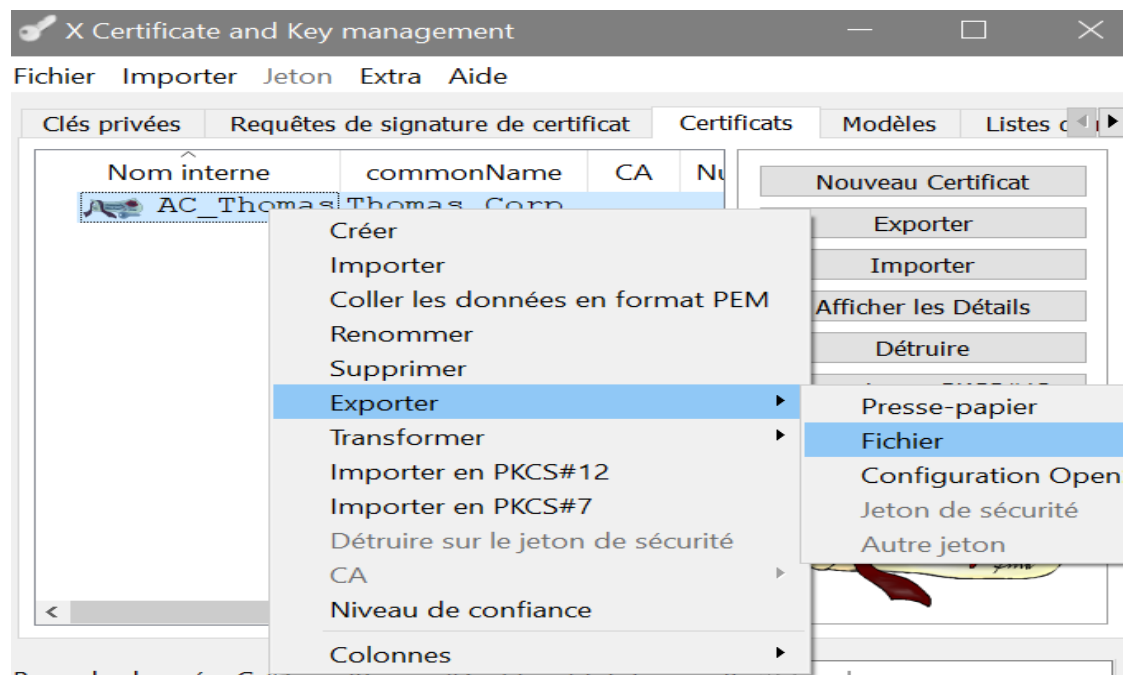


Figure 7: Exportation du nouveau certificat

Le fichier .crt peut être ouvert et observe qu'il n'est pas reconnu car il n'appartient pas à la banque de données de Windows. Il faut donc l'ajouter avec:



Figure 8: Appercu certificat

Une fois le certificat validé, on retourne dans XCA pour créer le certificat dédié à la signature de l'exécutable. Il suffit donc de refaire la même manipulation à la différence que cette fois-ci le certificat n'est pas auto-signé. Il est signé par notre certificat. On ajoute alors les informations sans oublier de spécifier que ce certificat servira pour des signatures digitales.

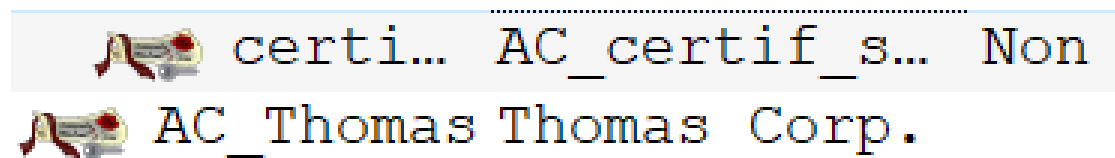
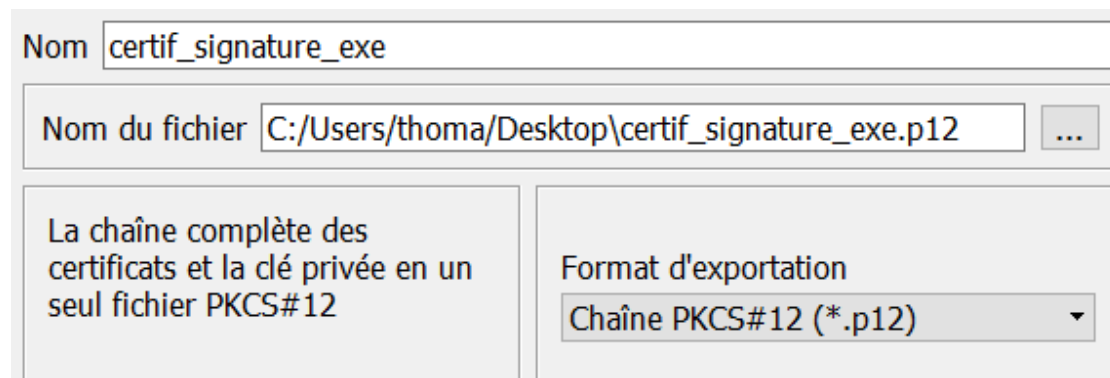


Figure 9: Appercu du nouveau certificat dédié à la signature des exécutables

Il faut une nouvelle fois exporter ce certificat mais dans un nouveau format qui contient aussi la clef privée :



Nom

Nom du fichier ...

La chaîne complète des certificats et la clé privée en un seul fichier PKCS#12

Format d'exportation

Figure 10: Exportation du nouveau Certificat pour la signature des exécutables

Une fois tout cela fait, on peut ouvrir un terminal windows dans lequel on va signer notre exécutable.

AC_Thomas	31.10.2020 14:24	Certificat de sécurité	2 Ko
certif_signature_exe	31.10.2020 15:10	Échange d'informatio...	3 Ko
CrackMe7	31.10.2020 15:15	Application	8 Ko
database	31.10.2020 15:10	XCA database	10 Ko
signtool	06.12.2019 22:09	Application	414 Ko

Figure 11: Contenu du dossier qui contient l'exécutable et les certificats

Dans ce dossier, on trouve l'AC qui est utilisée pour signer le certif_signature_exe qui va lui même signer le CrackMe7 modifié au tout début. Il y a aussi la base de donnée qui contient nos certificats et l'exécutable signtool qui a été installé au début et copié dans ce dossier.

```
C:\Users\thoma\Desktop\test>signtool.exe sign /t http://timestamp.digicert.com/ /f certif_signature_exe.p12 -p CrackMe7.exe
Done Adding Additional Store
Successfully signed: CrackMe7.exe
```

Figure 12: Signature du CrackMe7 depuis le terminal

Note : après le -p et avant CrackMe7.exe se trouve le mot de passe pour la clef privée qui est ici caché.

On peut donc regarder les propriétés de l'exécutable et observer que le programme est bien signé par nous-même avec la date de signature :

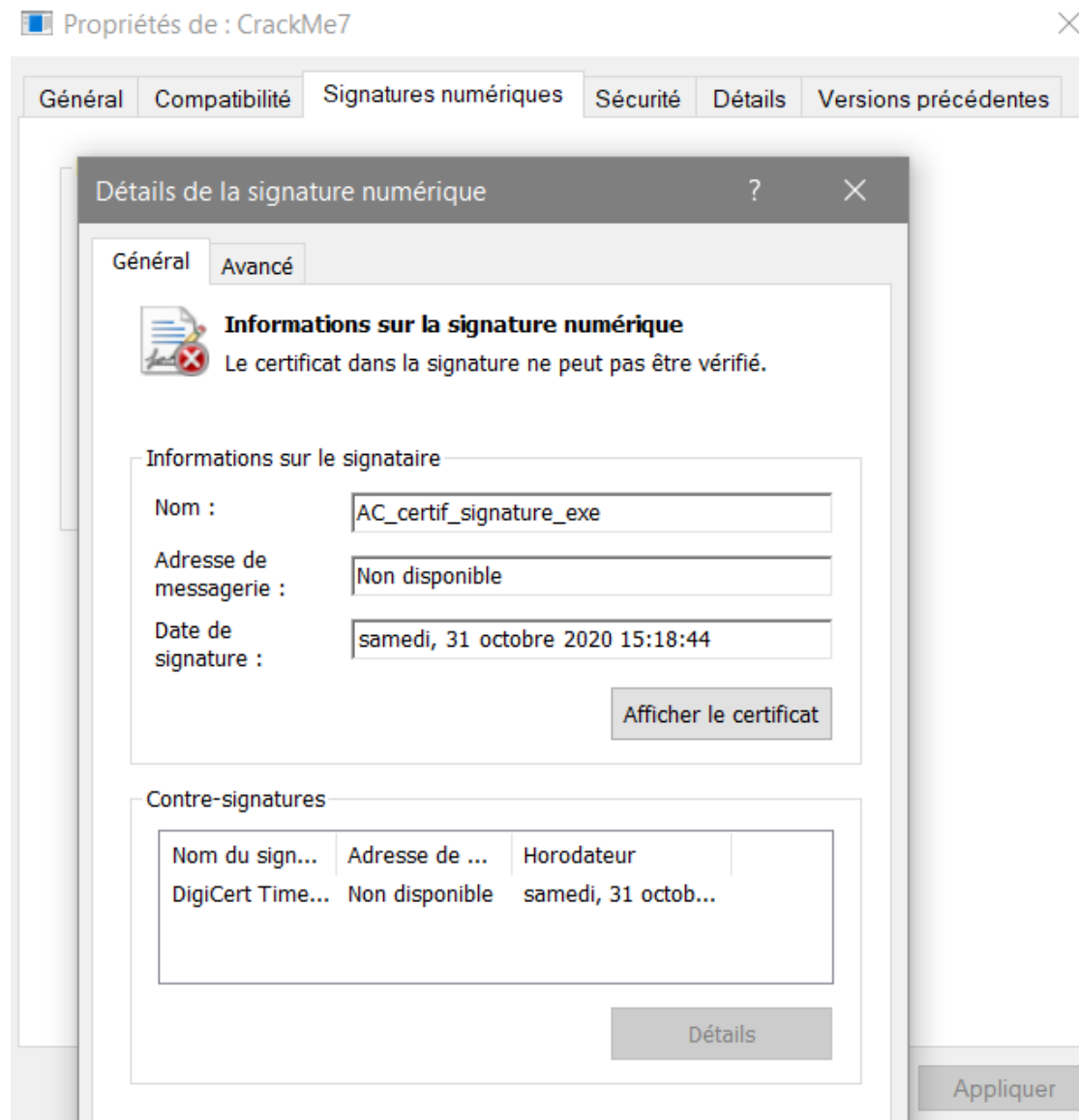
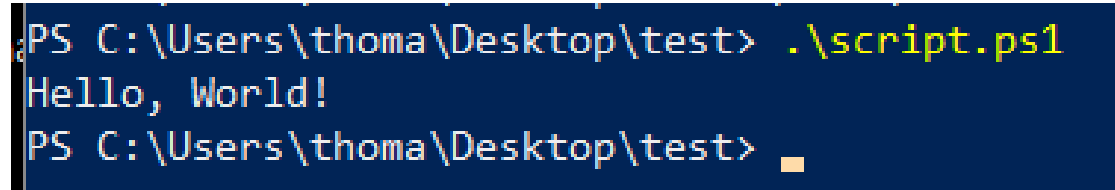


Figure 13: Affichage des propriétés de l'exécutable avec la nouvelle signature

2 Signature 2 (SIG2)

Pour le second exercice, nous devons créer un script HelloWorld en PowerShell. Ce dernier sera signé avec le même certificat que pour l'exercice précédent. On se place donc dans le dossier où se trouve déjà signtool.exe et nos certificats. Avec Notepad, on peut créer un fichier que l'on nomme script.ps1. Ce fichier contient uniquement la commande : Write-Host "Hello, World!".

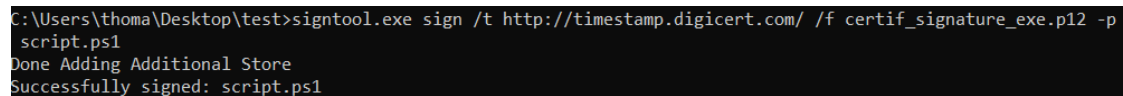
On peut donc ouvrir le fichier et exécuter le script depuis PowerShell:



```
PS C:\Users\thoma\Desktop\test> .\script.ps1
Hello, World!
PS C:\Users\thoma\Desktop\test> █
```

Figure 14: execution du script dans PowerShell

On voit que le script fonctionne bien, on peut donc le signer de la même manière avec notre certificat :



```
C:\Users\thoma\Desktop\test>signtool.exe sign /t http://timestamp.digicert.com/ /f certif_signature_exe.p12 -p
script.ps1
Done Adding Additional Store
Successfully signed: script.ps1
```

Figure 15: Signature du script

Le script est alors signé correctement et si on le réouvre, on voit que sa structure a changée:

```
Write-Host "Hello, World!"
# SIG # Begin signature block
# MIIT/AYJKoZIhvcNAQcCoIIT7TCCE+kCAQExCzAJBgUrDgMCGGUAMGkGCisGAQQB
# gjcCAQSGWzBZMDQGCisGAQQBgjccAR4wJgIDAQAABBAfzDtgWUsITrck0sYpfvNR
# AgEAAgEAAgEAAgEAAgEAMCEwCQYFKw4DAhoFAAQUyny3BBtdRvf+dinAQdzNwDJg
# +r2gghAeMIIC2zCCAcOgAwIBAgIBAJANBgkqhkiG9w0BAQsFADAYMRYwFAYDVQQD
# Ew10aG9tYXMgY29yb3AuMB4XDTEwMTAzMTEzNTQwMFoXDTEwMTAzMTEzNTQwMFo
```

Figure 16: Affichage du Script signé

Note : je n'ai pris qu'une partie de la signature simplement pour afficher que le fichier a bien été modifié.

On peut aussi voir la valeur du timestamp pour valider l'exercice:

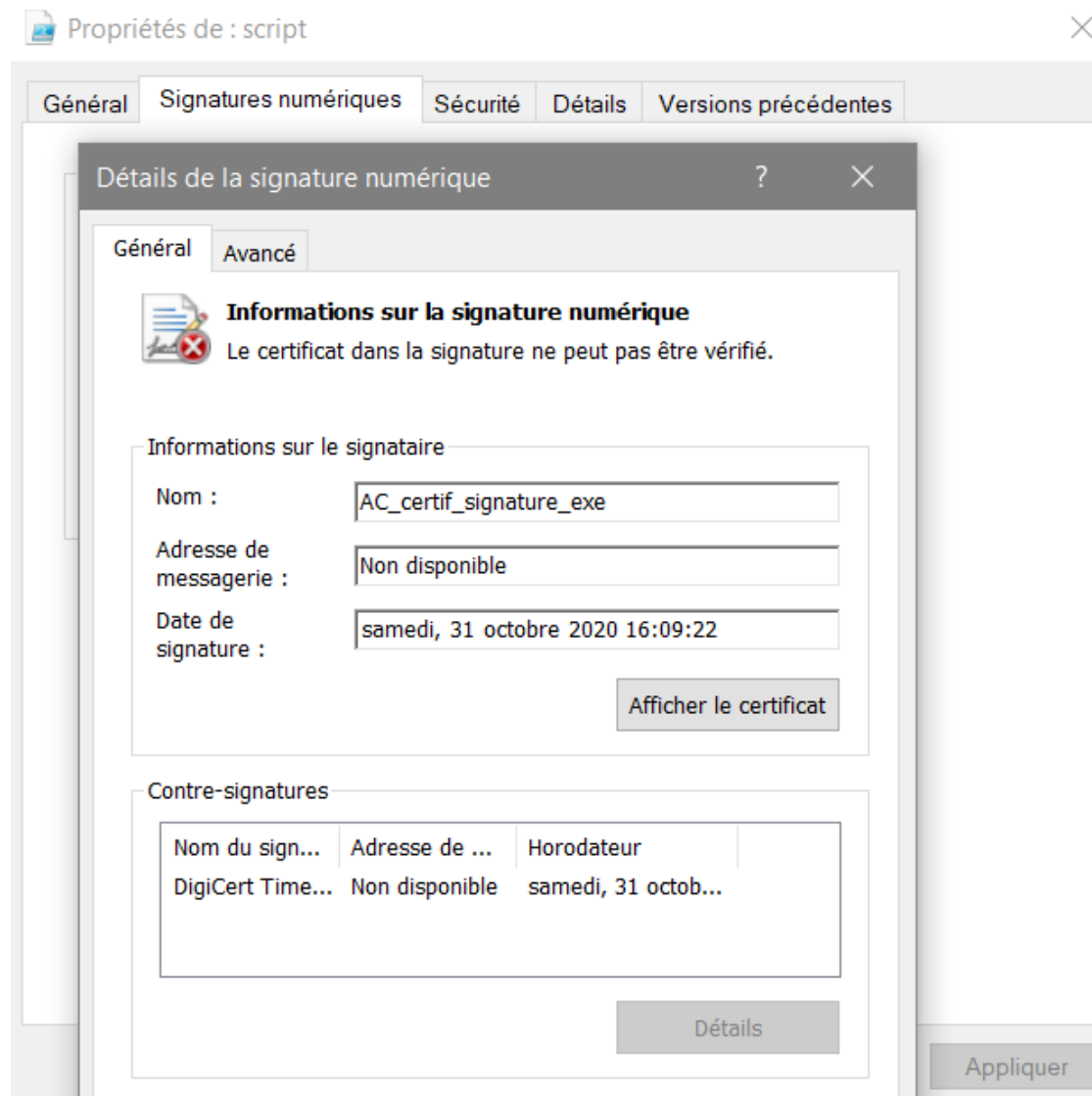


Figure 17: Affichage du timestamp pour le Script

3 Obfuscation de binaire (OBF1)

Pour cet exercice, nous avons plusieurs choix pour faire une obfuscation sur un code en C. D'après le commentaire sur le scoreboard, nous pouvons choisir entre la compression du code avec UPX, la modification du flow d'exécution du binaire ou alors de faire les 2. J'ai choisi de faire la compression avec UPX.

Il faut alors installer l'outil avec la commande : `sudo apt-get install upx`.

Ultimate Packer for eXecutables permet de compresser des fichiers exécutables. Il faut donc compiler notre fichier C avec la commande : `gcc -static -o output CodeGenerator.c`. Ceci va avoir pour effet de générer un fichier exécutable qui portera le nom "output". On peut alors appliquer la commande : `upx -o output_compressed output`. On va alors générer un exécutable compressé depuis l'exécutable qui avait été généré auparavant.

```
thomas@thomas:~/Documents/GIT/securite_applications/Serie2$ gcc -static -o output CodeGenerator.c
thomas@thomas:~/Documents/GIT/securite_applications/Serie2$ upx -o output_compressed output
Ultimate Packer for eXecutables
Copyright (C) 1996 - 2018
UPX 3.95      Markus Oberhumer, Laszlo Molnar & John Reiser   Aug 26th 2018

   File size      Ratio      Format      Name
   -----
1035560 ->   393904   38.04%   linux/amd64   output_compressed

Packed 1 file.
```

Figure 18: Compression du binaire avec UPX

Au moment d'exécuter, tout fonctionne exactement de la même manière. Il n'y a donc pas de problèmes avec la compilation.

```
thomas@thomas:~/Documents/GIT/securite_applications/Serie2$ ./output_compressed
Nuclear Warhead code generator
Fri Jul 31 12:08:05 2020
Press Any Key generate new code

2 7 2 5 9 9 5 3 3 8
Press Any Key generate new code
```

Figure 19: Execution du binaire compressé avec UPX

En appliquant cette commande, upx va compresser le fichier sans pertes. Une fois lancé, il se décompresse lui-même. Cela signifie que l'analyse du programme est beaucoup plus compliquée. Si on ouvrait tout les binaires avec ghidra pour chercher la fonction main et le code qui est exécuté, cette fois-ci c'est impossible :

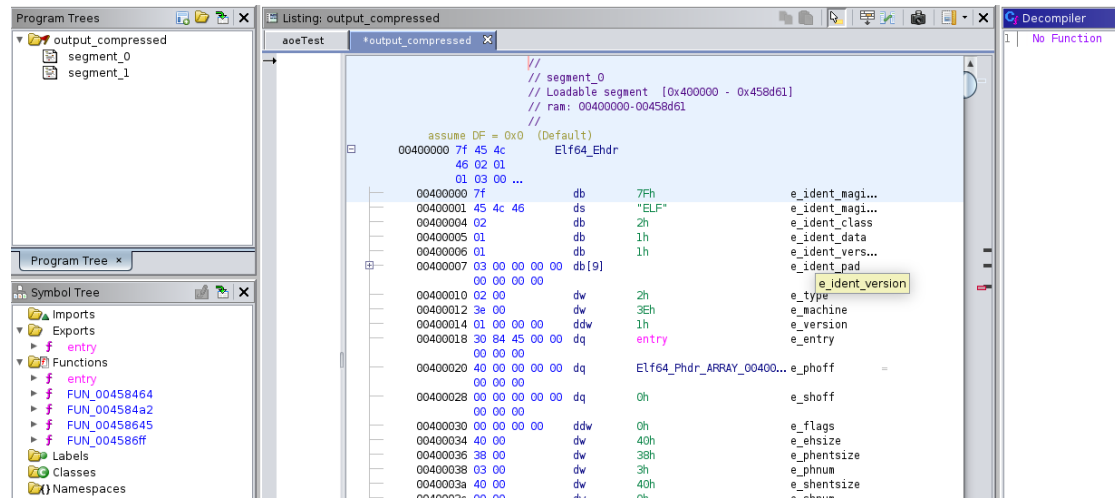


Figure 20: Ouverture du binaire avec ghidra

On voit que dans la catégorie Symbol Tree (à gauche), il y a très peu de fonction est les données ne sont pas intéressantes pour comprendre comment s'exécute le binaire. On a accès à très peu d'informations dans la mémoire. Les données sont cachées ou alors incompréhensibles. On en déduit que l'obfuscation par la compression UPX a bien fonctionnée.

4 TimeAttack on CrackMe 5 (TIME1)

Dans cet exercice, nous devons trouver le bon mot de passe du binaire. Pour cela, nous savons seulement que le mot de passe est une chaîne dont chaque caractère est testé un à un par le programme. Nous savons que le bon caractère est celui pour lequel l'exécution du binaire prend le plus de temps. Par exemple si le premier caractère est un "m" (au hasard), et que l'on rentre un "K" (toujours au hasard), et bien on aura plus vite l'information que le mot de passe est erroné que si l'on rentre vraiment un "m". La commande : `time ./crackme5` est donc très utile pour trouver le temps d'exécution.

```
thomas@thomas:~/Documents/GIT/securite_applications/Serie4$ time ./crackme5 K.....
Wrong password

real    0m0.002s
user    0m0.002s
sys     0m0.000s
thomas@thomas:~/Documents/GIT/securite_applications/Serie4$ time ./crackme5 m.....
Wrong password

real    0m0.332s
user    0m0.327s
sys     0m0.000s
```

Figure 21: Utilisation de la commande time sur le crackme5 pour trouver le premier caractère

Ce qu'il reste néanmoins à déterminer, c'est la longueur du mot de passe sinon on ne saura jamais

si le caractère est bon ou pas. Pour cela, on peut ouvrir le binaire avec ghidra (quelle surprise) et on voit que la ligne 59 fait une comparaison entre sVar4 qui est la longueur du mot de passe rentré en paramètre et sVar5 qui semble être la longueur attendu. Dans ce cas, on peut regarder l'adresse qui correspond à cette comparaison.

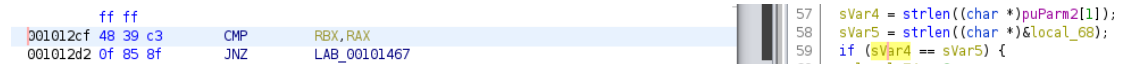


Figure 22: Utilisation de la commande time sur le crackme5 pour trouver le premier caractère

Une fois trouvée, on peut lancer le binaire avec gdb pour placer un breakpoint à l'adresse trouvée et connaître la longueur du mot de passe qui est attendu. On commence donc par faire la commande : `disass main`:

```
(gdb) disass main
Dump of assembler code for function main:
0x00000000000001199 <+0>:      push    %rbp
0x0000000000000119a <+1>:      mov     %rsp,%rbp
0x0000000000000119d <+4>:      push    %rbx
0x0000000000000119e <+5>:      sub     $0x78,%rsp
0x000000000000011a2 <+9>:      mov     %edi,-0x74(%rbp)
0x000000000000011a5 <+12>:     mov     %rsi,-0x80(%rbp)
0x000000000000011a9 <+16>:     mov     %fs:0x28,%rax
0x000000000000011b2 <+25>:     mov     %rax,-0x18(%rbp)
0x000000000000011b6 <+29>:     xor     %eax,%eax
0x000000000000011b8 <+31>:     cmpl    $0x2,-0x74(%rbp)
0x000000000000011bc <+35>:     je      0x11f6 <main+93>
0x000000000000011be <+37>:     mov     -0x80(%rbp),%rax
```

Figure 23: disass main pour trouver l'adresse que l'on cherche

```
0x000000000000012cf <+310>:    cmp     %rax,%rbx
```

Figure 24: adresse à laquelle on veut mettre un breakpoint

On peut donc appliquer la commande : `run password` puis à nouveau `disass main` pour sélectionner l'adresse à l'index 310 et y mettre un breakpoint :

```
0x0000555555552cf <+310>:    cmp     %rax,%rbx
```

Figure 25: adresse à laquelle on veut mettre un breakpoint

On peut alors placer un breakpoint et relancer le programme, toujours avec le mot de passe "password" qui fait 8 caractères :

```
End of assembler dump.
(gdb) b *0x00005555555552cf
Breakpoint 1 at 0x5555555552cf
(gdb) run password
Starting program: /home/thomas/Documents/GIT
Breakpoint 1, 0x00005555555552cf in main ()
```

Figure 26:

Avec la commande : `info registers`, on observe les registres lorsque l'on atteint le breakpoint. Ici les informations qui nous intéressent sont `rax` et `rbx`:

```
(gdb) info registers
rax                                0xf                                15
rbx                                0x8                                8
```

Figure 27: a

La valeur de `rbx` est la longueur du mot de passe que l'on rentre et la valeur de `rax` est la longueur du mot de passe attendu. On en déduit donc que le mot de passe fait 15 caractères. Il reste alors à faire un code qui va tester toutes les combinaisons possibles jusqu'à trouver le bon code.

Pour cela, j'ai réalisé le code suivant en C :

```
20     for(int i = 0; i < 15; i++){
21         for (unsigned char k = 0; k < 255; k++){
22             if (isalpha(k)) {
23                 password[i] = k;
24                 strcpy(input, command);
25                 strcat(input, password);
26                 strcat(input, " > /dev/null");
27                 for(int j = 0; j < 4; j++){
28                     gettimeofday(&start, NULL);
29                     status = system(input);
30                     gettimeofday(&stop, NULL);
31                     secs += (double)(stop.tv_usec - start.tv_usec) / 1000000 + (double)(stop.tv_sec - start.tv_sec);
32                 }
33                 secs = secs / 4;
34                 if(secs > max_time){
35                     max_time = secs;
36                     theGoodOne = k;
37                 }
38                 secs = 0.0;
39                 strcpy(input, "");
40             }
41         }
42         max_time = 0.0;
43         password[i] = theGoodOne;
44         strcpy(input, "");
45     }
46     printf("%s\n", password);
47     return 0;
```

Figure 28: aperçu du code

Pour itérer sur chaque caractère, je suis parti de la supposition que le mot de passe est uniquement composé de lettres majuscules ou minuscules (comme pour les autres CrackMe). De ce fait, on va tester chaque lettre sur le premier caractère. Les lignes 27 à 32 permettent de lancer la commande avec le nouveau mot de passe. On n'utilise pas la commande `time` mais la librairie `time.h` pour connaître le temps qui s'est écoulé entre le lancement de la commande et la fin. On regarde alors quel est le caractère qui a pris le plus de temps et on considère que ce caractère est le bon.

Note : Il faut lancer la commande plusieurs fois avec le même caractère et déduire le temps moyen d'exécution. En effet, comme des processus tournent en parallèle ce n'est pas toujours le bon caractère qui prend le plus de temps si on ne fait le test qu'une fois. Il semble que 4 fois soient suffisantes puisque c'est toujours le bon caractère qui sera retourné.

Une fois le bon caractère trouvé, on itère sur le suivant et ainsi de suite jusqu'au dernier. Le résultat est alors affiché dans le terminal :

```
thomas@thomas:~/Documents/GIT/securite_applications/Serie4$ gcc PasswordFinder.c
thomas@thomas:~/Documents/GIT/securite_applications/Serie4$ ./a.out
mKjBeDaYpjvraet
thomas@thomas:~/Documents/GIT/securite_applications/Serie4$ █
```

Figure 29: résultat du code

```
thomas@thomas:~/Documents/GIT/securite_applications/Serie4$ ./crackme5 mKjBeDaYpjvraet
Congratulations
thomas@thomas:~/Documents/GIT/securite_applications/Serie4$ █
```

Figure 30: lancement du binaire avec le bon mot de passe