QUELQUES ÉPREUVES DU CHALLENGE SÉCURITÉ « TRUST THE FUTURE »

Florian Lugou – florian.lugou@telecom-paristech.fr
Ludovic Apvrille – ludovic.apvrille@telecom-paristech.fr

mots-clés : ??????????????????

Airbus Defense and Space a récemment organisé un challenge sécurité intitulé « Trust the Future », à destination des étudiants de certaines écoles et universités. Nous nous sommes intéressés à ces épreuves – une seule nous a résisté avant la date limite. Nous décrivons ici comment nous avons réussi ces épreuves, en essayant pour certaines de donner des moyens un peu atypiques pour arriver à la solution (pas de python, etc.)...

1 Introduction

Les challenges sécurité ou CTF – *Capture the Flag* – consistent à obtenir des secrets grâce à la réalisation d'épreuves orientées sécurité. Les qualifications du challenge « Trust the Future » d'Airbus Defense and Space **[1]** ont eu lieu mi-novembre 2014 et consistaient à trouver des « tokens » répartis dans 16 épreuves (plus une épreuve « cachée »). Les épreuves se répartissaient comme suit : deux « brute-force », trois épreuves crypto, une épreuve réseau, cinq épreuves de rétro-ingénierie, trois épreuves de stéganographie, et trois épreuves « web ».

Nous avons terminé seconds de ce challenge, et nous vous proposons ici de parcourir certaines de ces différentes épreuves, en décrivant nos solutions de façon rapide pour certains challenges simples, et de façon plus approfondie pour certains challenges plus difficiles, en particulier pour des challenges en cryptographie et en rétro-ingénierie.

2 « Brute force »

2.1 Fichier ZIP

Un premier brute force concerne un fichier ZIP, dont l'ouverture provoque la demande d'un mot de passe.

```
$ unzip SECRET.zip
Archive:  SECRET.zip
[SECRET.zip] secr3t password:
```

Voilà une excellente occasion de ressortir *John the ripper* !

```
$ zip2john SECRET.zip > bruteforce/zip.hashes
```

Puis

```
$ john --show  ./zip.hashes
./bruteforce/SECRET.zip:st.ANTI#
1 password hash cracked, 0 left
```

On obtient le fichier **secr3t** :

```
$ more secr3t
The token is : A_BIG_secr3t
```

2.2 Fichier « shadow »

Un fichier de type « shadow password » est fourni :

```
$ more shadow
$1$gNdVWdxu$ihd.dBdcC49pWkwhr/xYt0
```

Il contient un mot de passe hashé avec un salt. Pas simple à attaquer, donc, l'on peut présupposer que le mot de passe est alphanumérique sans majuscule.

Nous avons essayé différentes longueurs, jusqu'à 6, en utilisant **hashcat** :

```
$ hashcat-cli64.app -a 3 -m 500 ../shadow ?l?l?l?l?l?l -o found.txt
```

Nous obtenons le mot de passe (et token) « loutre » ... Nous aurions donc pu aussi tenter une attaque en dictionnaire, sûrement plus rapide. Il s'agissait d'ailleurs du seul token purement alphanumérique des épreuves.

3 Cryptographie

Là commencent des choses bien plus sérieuses. Les trois épreuves consistent globalement à récupérer une clé cryptographique, par différentes techniques (reste chinois, factorisation avec l'approche de Fermat, etc.). Dans un premier temps, un seul fichier **crypto.tgz** est fourni. Il comporte quatre certificats **alice.crt**, **bob.crt**, **charly.crt** et **ca.crt**, et un fichier « challenge1 ». Ce dernier est un fichier de type « MIME entity text ». Une fois décodé de la base 64, le contenu de l'e-mail est un binaire que l'on peut analyser avec **dumpasn1** :

```
$ dumpasn1 -a toto|more
```

Il contient le fait que cet e-mail a été envoyé à Alice, Bob et Charly par ca@example.com, et que le contenu est chiffré avec du DES « des-EDE3-CBC ». Il contient aussi trois chiffrés, qui correspondent probablement à la clé DES chiffrée avec les clés publiques d'Alice, Bob et Charly respectivement... Mais comment retrouver les clés privées afin de connaître cette clé DES ?

Analysons à présent les certificats fournis.

```
$ openssl x509 -in alice.crt -text
Certificate:
    Data:
        Version: 4 (0x3)
        Serial Number:
            a6:d4:ef:4d:d3:8b:1b:b0:16:d2:50:c1:6a:68:04:70
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=FR, L=Paris, CN=ca@example.com
        Validity
            Not Before: Oct  7 07:22:30 2014 GMT
            Not After : Oct  6 07:22:30 2019 GMT
        Subject: C=FR, L=Paris, CN=charly@example.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:cf:a7:85:43:52:ff:9d:f5:e8:4a:b1:0a:b8:f0:
                 ...
                    ea:ce:64:94:f2:89:18:4d:ad:16:d2:d9:95:ed:1a:
                    dc:13
                Exponent: 3 (0x3)
        X509v3 extensions:
            ...

    Signature Algorithm: sha256WithRSAEncryption
...
```

Rien de vraiment anormal : une clé RSA 2048 bits ... Ah mais si, son exposant est de 3, voilà qui est bien faible. En regardant les certificats de Bob et Charly, nous constatons le même problème : des exposants de 3. Après un passage par Wikipédia, puis par un article expliquant comment récupérer le texte chiffré lorsque nous avons trois messages chiffrés avec trois clés à faible exposant (le théorème du reste chinois) **[2]**, nous savons que nous pouvons récupérer la clé de session. En quelques mots, supposons que **Ka**, **Kb** et **Kc** soient les clés publiques de Alice, Bob et Charly. Alors, « ca » a généré trois messages chiffrés **Ca**, **Cb**, **Cc** comme suit (**m** étant le message initial) :

Ca = m^3 mod Ka Cb = m^3 mod Kb Cc = m^3 mod Kc

Un attaquant peut alors calculer **c = m^3 mod Ka Kb Kc**. Comme **Ka.Kb.Kc** est plus grand que **m^3** (hypothèse très probable), alors **c = m^3**. On peut donc en déduire **m**.

Pour calculer cela, il est assez usuel d'utiliser la librairie grand nombre de python. Faisons cela avec la librairie « BigInteger » de Java pour montrer que cela se fait facilement avec d'autres langages. Nous avons aussi extrait, à l'aide de **dumpasn1**, les clés publiques d'Alice, Bob et Charly de leurs certificats (**alice.pub**, etc.) et aussi les clés chiffrées par « ca » : **Ca**, **Cb**, et **Cc** (**alice.key**, etc.).

```
import java.math.BigInteger; import ...

public class AnalyzeKeys  {
    public static final BigInteger THREE = new BigInteger("3");

    public static BigInteger cubeRoot(BigInteger n) {
    // Based on Newton's method
    BigInteger x = BigInteger.ZERO.setBit(n.bitLength() / 3 + 1);
    while (true) {
        BigInteger y = x.shiftLeft(1).add(n.divide(x.multiply(x))).
divide(THREE);
        if (y.compareTo(x) >= 0) {break;}
        x = y;}
    return x;}

    public static void load(BigInteger tab[], int index, String fileName)
{...}

    public static void printInfoOnKeys(BigInteger tab[], String name) {...}

    public static BigInteger computeC(BigInteger pubs[], BigInteger keys[]) {
  BigInteger nb = pubs[0]; BigInteger nc = pubs[1]; BigInteger nd = pubs[2];
  BigInteger cb = keys[0]; BigInteger cc = keys[1]; BigInteger cd = keys[2];

  BigInteger mult1 = nc.abs().multiply(nd);
  BigInteger mult2 = nb.abs().multiply(nd);
  BigInteger mult3 = nb.abs().multiply(nc);

  BigInteger mod1 = mult1.abs().modInverse(nb);
  BigInteger mod2 = mult2.abs().modInverse(nc);
  BigInteger mod3 = mult3.abs().modInverse(nd);

  BigInteger final1 = mult1.multiply(mod1).multiply(cb);
  BigInteger final2 = mult2.multiply(mod2).multiply(cc);
  BigInteger final3 = mult3.multiply(mod3).multiply(cd);

  BigInteger finalA = final1.add(final2).add(final3);

  BigInteger allMods = nb.abs().multiply(nc).multiply(nd);
  BigInteger finalB = finalA.abs().mod(allMods);
```

```
  BigInteger finalC = cubeRoot(finalB);

  return finalC;
    }

    public static void main(String[] args) {
        if (args.length < 6) {
            System.out.println("Must provide 6 files to analyze: 3 pubs keys,
3 encrypted keys");return;}

  BigInteger pubs[] = new BigInteger[3]; BigInteger keys[] = new
BigInteger[3];
  String alicePub, bobPub, charlyPub, aliceKey, bobKey, charlyKey;

       load(pubs,0,"alice.pub");load(pubs,1,"bob.pub");load(pubs,2,"charly.
pub");
  load(keys, 0, "alice.key");load(keys, 1, "bob.key"); load(keys, 2, "charly.
key");

  BigInteger result = computeC(pubs, keys);

  System.out.println("result:\nlength=" +  result.bitLength() + "\nvalue="  +
result.toString(16));
    }
}
```

Avec :

```
$ java AnalyzeKeys alice.pub bob.pub charly.pub alice.key bob.key charly.key
```

Nous obtenons une clé (**4f8957408f0ea202c785b95e206b3ba8da3dba7aea08dca1**), que l'on peut appliquer sur les données de l'e-mail chiffré (« cypher »):

```
$ openssl enc -d -des-ede3-cbc -K
4f8957408f0ea202c785b95e206b3ba8da3dba7aea08dca1 -iv
01d4ce3af4d17abb -in cipher > decrypt
```

Nous y sommes presque. « decrypt » est encodé en base64 et est un tgz, qui contient un fichier « challenge2 » (pour crypto 2) et le token...

4 Rétro-ingénierie

Parmi les cinq épreuves de rétro-ingénierie, trois d'entre elles peuvent être résolues assez aisément avec l'aide d'un désassembleur performant (tel IDA Pro ou radare2), une présente la particularité d'être une image d'un noyau pour processeur Intel 80386 et la dernière utilise diverses techniques d'offuscation et demande une analyse plus approfondie. C'est celle-ci que nous avons choisi de présenter ici.

Le fichier fourni « crackme3 » est un exécutable ELF pour une architecture AMD64, lié statiquement et sans symboles. Dans un premier temps, il est intéressant d'utiliser des outils d'analyse statique de haut niveau (**file**, **readelf**, **strings**, etc.) afin de mieux cerner à quel type d'épreuve nous nous attaquons. Ici en particulier, **readelf** nous permet de voir qu'aucune section n'est définie :

```
$ readelf -S crackme3
Il n'y a pas de section dans ce fichier.
```

Le fichier n'étant pas excessivement léger, on peut supposer que le programme a été modifié, *packé* et que les instructions réellement exécutées sont décodées à chaque lancement. Cette hypothèse est confirmée par la mention d'un packer dans les chaînes de caractère extraites avec **strings** et par une entropie anormalement élevée (7,72 bits par octet contre un peu moins de 6 habituellement).

Afin d'estimer à quel moment le code utile est présent en mémoire, on peut commencer par observer le comportement global de l'exécutable en regardant les *syscall* qu'il émet en utilisant l'outil **strace** :

```
$ strace ./crackme3 password
...
ptrace(PTRACE_TRACEME, 0, 0x1, 0)       = -1 EPERM (Operation not permitted)
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0} ---
+++ killed by SIGSEGV (core dumped) +++
Erreur de segmentation (core dumped)
```

Contrairement à une exécution directe, le programme se termine avec une erreur de segmentation. L'explication se trouve juste avant dans la liste des syscall : **crackme3** appelle **ptrace** afin de se tracer lui-même, ce qui est refusé par le système d'exploitation, car **strace** lui est déjà attaché. Cette technique courante dans les malwares permet d'en compliquer l'analyse.

Cet appel arrive cependant assez tard et on peut espérer que le code utile est alors en mémoire. On va donc générer un dump du processus au moment de l'appel à **ptrace** :

```
$ gdb crackme3
...
(gdb) catch syscall ptrace
Catchpoint 1 (syscall 'ptrace' [101])
(gdb) run password
Starting program: /home/misc/crackme3 password

Catchpoint 1 (call to syscall ptrace), 0x00007ffff7add2c9 in ?? ()
(gdb) generate-core-file crackme3.dump
```

En utilisant à nouveau « strings » sur **crackme3.dump**, on peut confirmer que le programme *réel* est bien présent en mémoire en recherchant la chaîne de caractères qui est affichée quand on ne fournit pas de paramètre :

```
$ strings --radix=x crackme3.dump | grep usage:
   2abe   usage: %s <password>
```

On utilise alors un désassembleur (IDA Pro pour notre part) afin d'analyser **crackme3.dump**. On peut se servir de l'offset de la chaîne **usage:%s <password>** pour retrouver la fonction dans laquelle elle est utilisée. Celle-ci semble correspondre à la fonction main initiale et la fin de la fonction semble relativement simple : deux nombres de 64 bits chacun sont xor-és ensemble et le résultat est comparé à un troisième nombre à l'adresse **0x4013c2**. Pour nous assurer que nous sommes sur la bonne voie, nous retournons sous **gdb** :

```
$gdb crackme3
...
(gdb) hbreak *0x00007ffff7add2c9
```

```
Hardware assisted breakpoint 1 at 0x7ffff7add2c9
(gdb) commands
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>set $rax=0
>continue
>end
(gdb) hbreak *0x4013c2
Hardware assisted breakpoint 2 at 0x4013c2
(gdb) run 0123456789abcdef
Starting program: /home/aishuu/shared/crackme3 0123456789abcdef

Breakpoint 1, 0x00007ffff7add2c9 in ?? ()

Breakpoint 2, 0x00000000004013c2 in ?? ()
(gdb) set $rax = 0
(gdb) c
Continuing.
Good boy :)
```

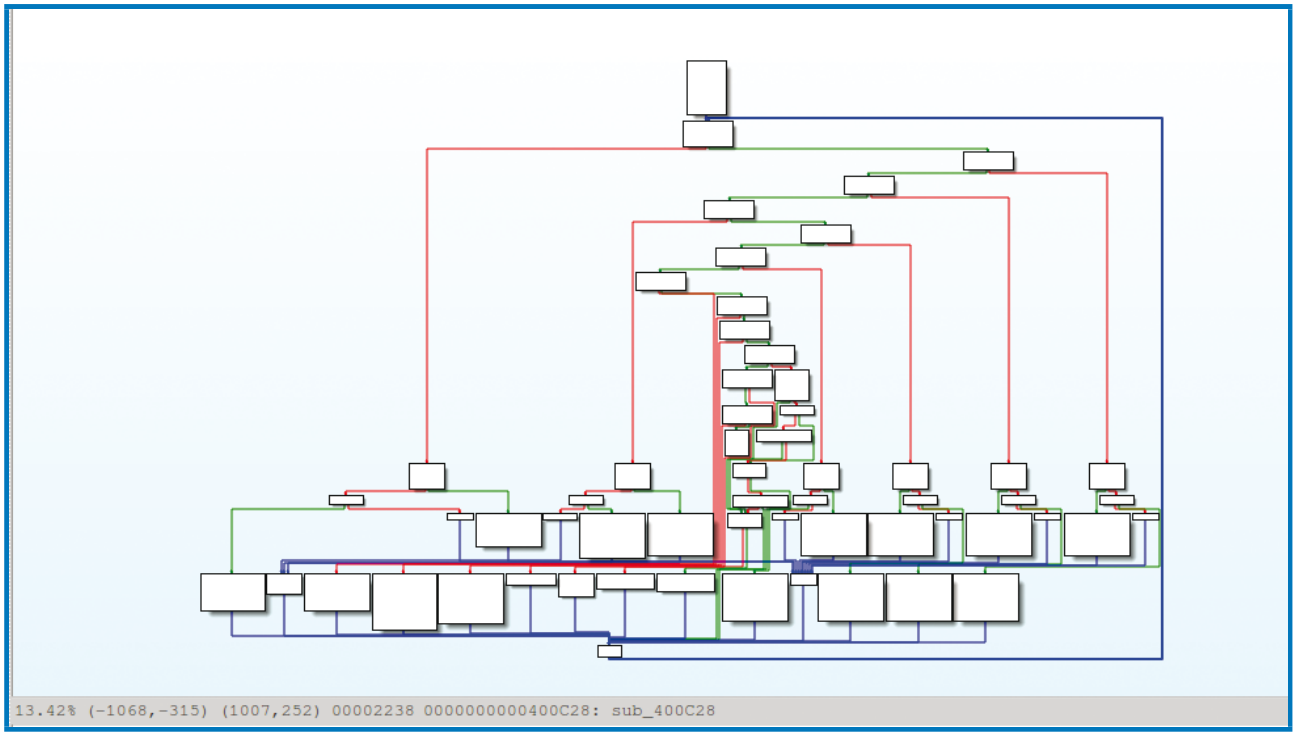
Ici, le premier breakpoint nous permet de contourner la protection utilisant **ptrace** et lorsque le second est atteint, nous changeons le résultat de la comparaison afin de faire croire au programme que les deux nombres étaient identiques. Celui-ci nous répond « Good boy:) », ce qui est bon signe. Il ne nous manque plus qu'une chose : le mot de passe. Pour le retrouver, nous devons nous intéresser aux trois nombres comparés. Deux d'entre eux ne dépendent visiblement pas du mot de passe entré et le troisième est initialisé avec les seize premiers caractères du mot de passe et est transformé par une fonction débutant à l'adresse **0x400C28**.

Celle-ci peut paraître compliquée, mais on s'aperçoit qu'il s'agit en réalité d'une boucle (que l'on reconnaît dans le diagramme ci-dessus au trait bleu sur la droite représentant un jump) à l'intérieur de laquelle se trouve un switch ne contenant que quelques instructions réellement exécutées pour chaque valeur possible. Cela correspond à une autre technique couramment employée dans des malwares : le code utile est exprimé à l'aide d'un jeu d'instructions personnalisé et interprété par un processeur abstrait.

Ce processeur dispose de registres et d'une zone mémoire dans laquelle se trouve notre mot de passe et le jeu d'instruction est similaire à MIPS. Une instruction inhabituelle a tout de même été rajoutée. Celle-ci (que nous avons ici nommée MANG) complique l'analyse du programme selon trois modes en mélangeant les instructions ou en modifiant les registres et la mémoire.

En réimplémentant cette machine virtuelle en python, l'on peut mimer l'exécution de crackme3 pour une valeur de mot de passe particulière :

```
$ ./vm_crackme3.py 0123456789abcdef
   0:    LOAD      r0          *0x2180
   5:    LOAD      r1          *0x2182
   a:    MV        r2          $0x1337
  10:    ADD       r0          r2
  14:    SUB       r1          r2
  18:    STOR      *0x2180     r0
  1d:    STOR      *0x2182     r1
  22:    MV        r0          $0xbeef
  28:    MANG      $3
  2a:    LOAD      r3          *0x2184
  2f:    LOAD      r4          *0x2186
  34:    NOT       r3
  36:    XOR       r4          $0xde4d
  3c:    ADD       r3          r1
  40:    SUB       r4          r1
  44:    STOR      *0x2186     r3
  49:    STOR      *0x2184     r4
  4e:    MANG      $1
  50:    LOAD      r1          *0x2188
  55:    LOAD      r2          *0x218a
```



```
5a:    MV       r0          $0x4
60:    MV       r3          r1
64:    ADD      r3          r2
68:    XOR      r3          $0xcafe
6e:    ADD      r2          $0xbad
74:    XOR      r2          r4
78:    MV       r1          r3
7c:    SUB      r0          $0x1
82:    JNZ      60
87:    STOR     *0x2188     r2
8c:    STOR     *0x218a     r1
91:    MANG     $2
93:    LOAD     r6          *0x218c
98:    LOAD     r5          *0x218e
9d:    LOAD     r1          *0x2180
a2:    LOAD     r2          *0x2182
a7:    XOR      r5          r1
ab:    XOR      r5          r2
af:    ADD      r6          r1
b3:    SUB      r6          r5
b7:    STOR     *0x218e     r5
bc:    STOR     *0x218c     r6
c1:    JMP      db
db:    END


Bad password
Expect : 0x77B6B1AAEB5D9A20B86F33DBD4CF6CB7
Got:     0xD21664FEFD7F4C5E451FE699F014AB88
```

On remarque dans cette trace que l'exécution ne dépend pas du mot de passe qui se situe à l'adresse (virtuelle) **0x2180**. Cela signifie qu'il nous suffit d'inverser la procédure pour retrouver le bon mot de passe. On obtient alors :

```
$ ./crackme3 '!pWN_mY_vm_FFS! '
Good boy :)
```

5 Stéganographie

Trois épreuves de stéganographie sont proposées : deux images, et un fichier de son au format WAV.

La première image est le logo d'Airbus. Rien de particulier a priori, sauf que le fond blanc incite à faire un remplissage avec du noir, avec un seuil faible... Il apparaît alors un texte caché.

La deuxième image est le logo de Google. Là encore, rien de visible directement. Des zooms nous permettent finalement de constater une petite dizaine de pixels noirs sur le fond blanc, à certains endroits. Nous réalisons de plus que ces pixels sont situés dans la partie supérieure gauche du logo, ce qui veut dire que leurs coordonnées (x,y) ont des valeurs comprises entre 60 et 120... Nous pensons alors à une représentation graphique de caractères ascii, ce qui nous permet finalement d'obtenir le token.

La troisième stégano est plus surprenante : un fichier appelé **noise.wav**, que nous jouons avec audacity. Le fichier porte bien son nom. Audacity nous permet de constater que le même motif se répète régulièrement.

Cela peut faire penser à du morse, avec les pics « courts » qui pourraient être considérés comme des points, et les plateaux plus longs comme des traits. Mais cela ne donne rien.

Notre deuxième idée est de considérer que cela représente un flux de caractères codés sur 7 bits. Selon cette idée, nous obtenons le string : **theAi_WK]'OT 4qroe7q**.

La solution n'est pas encore là ... mais le « the » du début fait indéniablement penser que la solution est proche. Nous allons alors effectuer des décalages de 0 à 6 bits sur le flux binaire considéré, ce qui nous donne donc 7 strings :

1. **theAi_WK]'OT 4qroe7q**
2. **iQKS?/:O(ice_Job**
3. **S#&~^.t> P%SGK?_D**
4. **&F,M}<]h<} K'~+?**
5. **:MXy;P4yzAN-|V~**
6. **7urw isu-<[y-|**
7. **tokenASgjZ8y7r[x@**

En utilisant les strings dans l'ordre 1 – 7 – 6 – 5 – 2, l'on découvre « The token is Nice_Job ».

Conclusion

Cet article a présenté la résolution de certaines épreuves des qualifications du CTF « Trust The Future » organisé par Airbus Defense and Space en novembre 2014. Nous avons terminé deuxièmes des qualifications, puis 4ème lors de la finale, qui consistait en des épreuves différentes (découverte d'un réseau, attaques de serveurs). Les épreuves étaient très intéressantes, elles nous ont tenus en haleine pendant les 2 semaines de la compétition : nous ne pouvons que demander à Airbus Defense and Space de recommencer l'an prochain ! ■

Références

[1] https://challenge.trustthefuture.eu/index.html

[2] http://en.wikipedia.org/wiki/RSA_%28cryptosystem%29

[3] http://www.cims.nyu.edu/~regev/teaching/lattices_fall_2004/ln/rsa.pdf