

Techniques d'attaque - stéganographie

Tristan BILOT, Nora DELFAU, Enzar SALEMI, Madushan THAMBITHURAI

31 Mai 2021

Abstract

Vous êtes Elliot Hackerman, un expert dans le domaine de l'informatique forensique. NotThatEvilCorp, l'un des leaders mondiaux du marché des logiciels, avait besoin de vos services. Apparemment, ils soupçonnent l'un de leurs employés (un développeur Java) de vendre des informations à la concurrence. Il nous est mis à disposition 3 emails, dont 1 contenant une image et un autre en contenant 2 (voir Fig. 1). L'objectif va être de découvrir des informations cachées dans ces images afin de déduire si oui ou non la personne transmet des secrets au concurrent.

1 Notes

1.1 Outils fréquemment utilisés

Elpserk est un outil de social engineering qui à partir du profil d'une personne, va générer un dictionnaire de mots de passe potentiellement utilisés par une personne cible. Hydra est un des outils les plus utilisés et complet pour effectuer des attaques par bruteforce. D'autres outils existents comme Brutus, Cain et Abel ou John the Ripper.

1.2 Rainbow tables

Dans `/etc/shadow`, nous trouvons les hashes des mots de passe système. Grâce aux rainbow tables, on peut parvenir à trouver le hash correspondant au mot de passe en utilisant des dictionnaires de hash. Les rainbow tables permettent une recherche plus rapide dans ces dictionnaires très longs à parcourir. Quand on cible un système, on fait en sorte de connaître à l'avance la méthode de hashage pour avoir le dictionnaire de hashes qui utilise les hashes correspondants à la méthode. Stegseek permet de faire de la recherche sur les mots de passe utilisés pour la stéganographie.

1.3 Diffie-Hellman

L'algorithme Diffie-Hellman est utilisé lors de la génération de clés secrètes afin d'établir une connexion sécurisée entre 2 machines. C'est grâce à cela qu'il est possible à Alice et Bob de communiquer de façon sécurisée grâce à leur clé privée sans qu'une troisième personne ne puisse intercepter les messages. On l'utilise notamment lors de l'ouverture d'une connexion à un site sécurisé via le protocole SSL/TLS. De plus, il est omniprésent sur Internet et sur nos smartphones, ce qui en fait un algorithme important à connaître.



Figure 1: Les 3 images à analyser

2 Attaque par dictionnaire sur images

Dans le cas d'informations cachées dans des images par mot de passe, il peut être intéressant d'utiliser une attaque bruteforce (recherche exhaustive) ou une attaque par dictionnaire (utiliser des mots de passe souvent utilisés afin de maximiser nos chances de trouver le bon mot de passe, souvent créé par un humain et utilisant des inspirations de la vie courante). Une méthode pourrait être de créer un dictionnaire contenant tous les mots du mail puis d'utiliser ce dictionnaire afin d'essayer d'attaquer l'image grâce à l'outil stegseek. Dans un premier temps, construisons donc un dictionnaire contenant l'ensemble des mots contenus dans l'email. A présent, utilisons l'outil stegseek afin d'utiliser un dictionnaire de mots

```
1 Bonjour
2 James
3 Je
4 suis
5 de
6 retour
7 des
8 USA
9 New York
10 J'ai
11 reçu
12 votre
13 message
14 merci
15 Je
16 vous
17 envoie
18 deux
19 petites
20 photos
21 que
22 j'ai
23 prises
24 sur
25 le
26 pont
27 de
28 Brooklyn
```

Figure 2: Dictionnaire contenant la liste des mots du 1er email

de passe contre une image (Fig. 3). Pour utiliser l'outil stegseek sur MacOS, il est possible d'utiliser

```
o docker run --rm -it -v "${PWD}:/steg" rickdejager/stegseek Eiffel.jpeg dict
StegSeek 0.6 - https://github.com/RickdeJager/StegSeek

[!] Found passphrase: "Eiffel"
[!] Original filename: "Diffi-Hellman1.txt".
[!] Extracting to "Eiffel.jpeg.out".
```

Figure 3: StegSeek utilisé sur l'image "Tour Eiffel"

docker, sinon il est possible de l'installer directement sur Linux. L'outil retourne bien qu'un mot de passe parmi le dictionnaire a été découvert, dans le cas de l'image de la Tour Eiffel il s'agit du mot de passe "Eiffel". De la même façon, nous pouvons trouver les 2 mots de passes des deux autres images: "Brooklyn" (Fig. 4) et "Statue".

```
o docker run --rm -it -v "$(pwd):/steg" rickdejager/stegseek Brooklyn.jpg dic
StegSeek 0.6 - https://github.com/RickdeJager/StegSeek

[i] Found passphrase: "Brooklyn"
[i] Original filename: "Diffi-Hellman2.txt".
[i] Extracting to "Brooklyn.jpg.out".
```

Figure 4: StegSeek utilisé sur l'image "Brooklyn"

3 Exfiltration des clés de chiffrement

Maintenant que nous avons le mot de passe associé à chaque image, nous pouvons poursuivre l'analyse avec l'outil steghide, qui va nous permettre de récupérer les différentes clés utilisées pour le chiffrement des images (Fig. 5).

```
nora@nora-GL62-7RD:~/Documents/Ing2/TATT$ steghide extract -sf Elffel.jpeg
Enter passphrase:
Wrote extracted data to "Diffi-Hellman1.txt".
nora@nora-GL62-7RD:~/Documents/Ing2/TATT$ cat Diffi-Hellman1.txt
p=7254984186419407889972683711689258549695575260843302046062234899404431141511
g=58759714624708089702199189818787533437405834872094654299502596364496752316901
A=71559523660568269188776671270040587101782920281998883402616283230525149718057
nora@nora-GL62-7RD:~/Documents/Ing2/TATT$ steghide extract -sf Statue.jpg
Enter passphrase:
Wrote extracted data to "cipher.txt".
nora@nora-GL62-7RD:~/Documents/Ing2/TATT$ cat cipher.txt
02Xj00<X[0]Eu00R<C8[0]0005([0]00Bai[0]0000Z[0]
[0]0060fqu000-0'!jd0NB0L 0F0A00!0'00b0c00004d00q 4040[0]0[0]0X0Yeg[d0/0]L[0]000000
l 0000I00000[0]0y0000h>0[0]0
0000[0]0[0]0X0<Ch0'-w[0]0[0]300L0[0]0[0]0y0V00"00
30000 -0J00m0R?_F00Y0[0]0[0]B000K000 D000000[0]0000[0]0[0]; m%000d
W0 0q000"0
00k0[0]0[0]00[0]0>0[0]0000
0000D00[0]0[0]0D0 00[0]00000860[0]nora@nora-GL62-7RD:~/Documents/Ing2/TATT$
nora@nora-GL62-7RD:~/Documents/Ing2/TATT$ steghide extract -sf Brooklyn.jpg
Enter passphrase:
Wrote extracted data to "Diffi-Hellman2.txt".
nora@nora-GL62-7RD:~/Documents/Ing2/TATT$ cat Diffi-Hellman2.txt
B=6771936423796414792844827144091002869644758288336922881400421691431779605772
AES-256-CBC no salt
vi 123456789
nora@nora-GL62-7RD:~/Documents/Ing2/TATT$
```

Figure 5: StegHide permettant de récupérer les clés de chiffrement

Après utilisation de l'outil, plusieurs fichiers sont générés: 1 fichier cipher.txt contenant le message chiffré en aes-256-cbc et 2 fichiers Diffie-Hellman qui contiennent les clés utilisées pour l'échange de clé lors du chiffrement. Grâce à ces différentes informations, nous sommes presque capable de déchiffrer le message car nous savons que l'algorithme utilisé pour la génération des clés est Diffie-Hellman, et nous

avons la plupart des clés. Il ne nous manque plus que la clé privée b afin d'appliquer le déchiffrement. Or, nous savons que la personne à l'origine de ces images est un développeur Java qui a certainement utilisé une génération pseudo-aléatoire pour l'obtention de la clé b . Cette génération aléatoire utilise le timestamp actuel de la machine donc nous pouvons essayer par bruteforce tous les timestamps possibles dans l'intervalle de temps où la génération de l'image est la plus probable (entre les dates d'envoi des deux emails). Ce qui nous invite à créer un algorithme Java utilisant de la même façon une génération de nombre aléatoire afin de compléter l'équation:

$$B = G^b \text{ mod } P$$

L'algorithme consiste à essayer de résoudre l'équation pour toutes les millisecondes entre l'envoi des 2 messages (Fig. 6). Avec cette première méthode, nous nous rendons compte qu'il y a 2 milliards de millisecondes entre les deux dates, ce qui mettrait un temps conséquent à résoudre. Nous avons donc pensé à regarder la date de création des images afin de réduire fortement cet intervalle de temps. Grâce à cette amélioration de l'algorithme, nous avons réussi à trouver la clé manquante en seulement quelques secondes.

```
public static void main (String[] args){
    BigInteger key, b;
    var p = new BigInteger( val: "72549841864194078899726837116892585496955575260843302046062234899404431414511");
    var g = new BigInteger( val: "58759714624708089702199189818787533437405834872094654299502596364496752316901");
    var A = new BigInteger( val: "71559523660568269188776671270040587101782920281998883402616283230525149718057");

    var B = new BigInteger( val: "67719364237964147928448271440910028696447582883369228814004216914317779605772");

    var c1 = new GregorianCalendar( year: 2015, month: 11, dayOfMonth: 13, hourOfDay: 14, minute: 40, second: 22);
    var c2 = new GregorianCalendar( year: 2015, month: 12, dayOfMonth: 6, hourOfDay: 15, minute: 16, second: 36);
    c1.setTimeZone(TimeZone.getTimeZone("GMT"));
    c2.setTimeZone(TimeZone.getTimeZone("GMT"));

    long beg = c1.getTimeInMillis();
    long end = c2.getTimeInMillis();

    while (beg <= end) {
        b = new BigInteger( numBits: 256, new Random(beg));
        if (B.compareTo(g.modPow(b, p)) == 0) {
            System.out.println("seed found: " + beg);
            key = A.modPow(b, p);
            System.out.println("key decimal : " + key);
            var res = key.toString( radix: 16);
            System.out.println("key hexa: " + res);
            break;
        }
        beg ++;
    }
}
```

Figure 6: Algorithme de bruteforce de la seed

Résultats de l'algorithme:

- random seed: 1450018437892
- decimal b key: 30899383377856444262432941452181130060109265952665886885841266221802454368037
- hexa b key: 44506e64c6a3de6416c6c3ce567a418371dbbebd917fc5df61201770365bf25

4 Déchiffrement du message

Il est désormais possible de déchiffrer le message cipher.txt. Pour cela nous utiliserons openssl avec les options -d pour déchiffrer, -aes-256-cbc -nosalt car c'est le mode de chiffrement utilisé initialement, ainsi que l'option -iv définissant le vecteur d'initialisation à utiliser. Pour rappel, dans un mode de chiffrement cbc, le vecteur d'initialisation va servir de source aléatoire pour le 1er bloc à chiffrer, qui lui-même sera utilisé pour le chiffrement des blocs suivants via des XOR par exemple. C'est grâce à cela que l'algorithme peut retourner à chaque chiffrement un résultat différent, et donc être beaucoup plus robuste et fiable. A noter qu'avant d'utiliser la clé b directement avec l'option -K d'openssl, il est nécessaire de convertir la clé en base hexadécimal, ce qui est effectué directement par l'algorithme Java. Enfin, nous arrivons à déchiffrer le message secret initialement caché dans l'image, grâce aux clés.

```
nora@nora-GL62-7RD:~/Documents/ing2/TATT$ openssl enc -d -aes-256-cbc -nosalt -iv 123456789 -K 44506e64c6a3de6416c6c3ce567a418371dbbcbdc917fc5df61201770365bf25 -in cipher.txt
hex string is too short, padding with zero bytes to length
Bonjour James,

Je vous envoie ci-après la clé de chiffrement utilisée par le serveur principal de notre société. Il utilise le chiffrement symétrique Blowfish en mode CCM:

clé: 765C4425c443AAF909862430BDA6C2C45FA

Veuillez effectuer le virement sur le compte bancaire suivant: 9898 322208 32.

N'oubliez pas de supprimer tous les fichiers compromettants.
Cordialement,
nora@nora-GL62-7RD:~/Documents/ing2/TATT$
```

Figure 7: Déchiffrement final du message