Rapport préliminaire – Audit de clefs cryptographiques

Date	5 février 2014
Rédigé par	Claire Smets, William Boisseleau, Pascal Edouard, Mathieu Latimier, Julien Legras
À l'attention de	Ayoub Otmani

1 Introduction

Cette première partie du projet consiste à faire une étude des clefs cryptographiques RSA circulant sur Internet. Nous avons donc dans un premier temps récupéré une liste d'adresses IP dont le port 443 (HTTPS) était ouvert.

Ensuite, nous avons tenté de récupérer les certificats de toutes les adresses listées. Cette récupération s'occupait également d'extraire les moduli et les clefs publiques des certificats, mais aussi de stocker une liste de certificats doublons.

Enfin, avec tous les moduli que nous avons extraits, nous avons pu construire deux arbres permettant de définir si des moduli contiennent des facteurs premiers communs.

2 ZMAP

ZMAP est un outil permettant de faire des scans réseau. Il est capable de scanner toutes les adresses IPv4 possibles, avec une moyenne de 1.4 millions de paquets envoyés par secondes sur les connexions à très haut débit.

Nous l'avons utilisé pour scanner l'ensemble des adresses IPv4, en envoyant simplement des paquets SYN aux adresses visées sur le port 443 pour SSL.

Pour l'installer sur nos machines, voici la procédure à suivre :

1. Installation des dépendances :

   ```
   ~ $ sudo apt-get install cmake libgmp3-dev libpcap-dev gengetopt byacc flex git
   ```

2. Récupération des sources de ZMap :

   ```
   ~ $ git clone git://github.com/zmap/zmap.git
   ```

3. Compilation :

   ```
   ~ $ cd zmap
   ~/zmap $ mkdir build
   ~/zmap $ cd build
   ~/zmap/build $ cmake .. -DENABLE_HARDENING=ON
   ~/zmap/build $ make
   ~/zmap/build $sudo make install
   ```

Il y a plusieurs options disponibles, nous avons utilisé la commande suivante pour effectuer notre scan :

```
# zmap -p 443 -o scan_output_zmap
```

3 Application RC

Une fois que l'on avait la liste fournie par ZMAP, il nous restait à récupérer les certificats sur les serveurs dont le port SSL était ouvert, en établissant une connexion sur chacun d'eux.

Pour cela nous avons codé deux scripts perl, l'un pour la récupération de certificat SSH (que nous n'avons pas testé à cause de l'avertissement reçu), et un autre pour la récupération de certificats SSL (ssl_collector.pl).

Commandes OpenSSL utilisées :

– pour l'établissement de la connexion (avec s_client) :

```
echo  \" \" | timeout 20 openssl s_client -connect $addr:443 -sess_out tmp_$addr.pem
-ignore_critical -showcerts -CApath /etc/ssl/certs > /dev/null 2> tmp_$addr.log
```

– pour la capture de session (avec sess_id) :

```
openssl sess_id -in tmp_$addr.pem -cert > certs/$addr.pem 2> tmp_$addr.log
```

– pour la récupération du certificat et de la clé privée :

```
openssl x509 -in certs/$addr.pem -noout -pubkey > keys/$addr.pem 2> tmp_$addr.log
```

Pour ce qui est de la gestion des doublons, nous avons également créé un autre script perl (clean.pl).

La chaîne d'utilisation est donc la suivante :

```
~ $ ./ssl_collector ips.txt
~ $ ./clean.pl
```

4 Factorisation

L'objectif final de cette première partie était de déterminer s'il existait des facteurs premiers communs parmi les clefs récupérées. Pour cela nous avons développé deux arbres de factorisation. Lors du scan nous avions récupéré 85.000 certificats uniques, mais nous avons également souhaité faire la factorisation avec les données du projet Sonar de Rapid7 Labs avec 523 000 certificats.

Le premier arbre se contente de multiplier l'ensemble des moduli par groupe de deux. Le second arbre calcule deux modulos du résultat produit, selon le carré des résulats stockés par l'arbre des produits (aux étapes intermédiaires) :

Une fois le second arbre calculé, les dernières feuilles des arbres indiquent soit un PGCD(res_final, N_i) = 1 (donc avec des entiers premiers uniques), ou un PGCD(res_final, N_i) = p (entier premier non unique).

Voici les algorithmes de ces deux arbres :

Entrées : Tableau des moduli des clefs publiques : T
Sorties : Hauteur de l'arbre, produits des moduli des clefs publiques
Données : Tableaux v, tmp ; Entier i, level
$v \leftarrow T$;
$level \leftarrow 0$;
tant que $|v| > 1$ **faire**
  $tmp \leftarrow \emptyset$;
  pour chaque $i \in \{0, .., |v|/2\}$ **faire**
    $tmp[i] \leftarrow v[i \times 2] \times v[i \times 2 + 1]$;
  fin
  $storeProductLevel(v, level)$;
  $v \leftarrow tmp$;
  $level \leftarrow level + 1$;
fin
retourner *level*

Algorithme 1 : Construction de l'arbre des produits

Entrées : Hauteur de l'arbre des produits : level
Sorties : PGCDs des moduli des clefs publiques
Données : Tableaux P, v, w ; Entier i
$P \leftarrow getProductLevel(level)$;
tant que $level > 0$ **faire**
  $v \leftarrow getProductLevel(level - 1)$;
  pour chaque $i \in \{0, .., |v|\}$ **faire**
    $v[i] \leftarrow P[i/2] \pmod{v[i]^2}$;
  fin
  $level \leftarrow 1$;
  $storeRemainderLevel(v, level)$;
  $v \leftarrow tmp$;
  $level \leftarrow level + 1$;
fin
$w \leftarrow \emptyset$;
pour chaque $i \in \{0, .., |v|\}$ **faire**
  $w[i] \leftarrow P[i/2] \pmod{v[i]^2}$;
  $w[i] \leftarrow w[i]/v[i]$;
  $w[i] \leftarrow pgcd(w[i], v[i])$;
fin
retourner w

Algorithme 2 : Construction de l'arbre des restes

Lors de nos tâches d'optimisation, nous avons réalisé qu'il était possible de construire l'arbre entier (product tree + remainder tree), en mémoire vive (nous avons donc créé une option fullRAM à notre algo).

L'algorithme est le même, seul le choix de la structure change. Dans la première méthode, nous utilisions des fichiers binaires GMP à chaque étape de construction, dans la seconde nous stockons l'intégralité de l'arbre des produits en mémoire.

Pour accélérer la construction de l'arbre, nous utilisons des *threads*, qui parallélisent le traitement de chaque niveau d'arbre (sur la largeur de l'arbre). Nous avons tenté d'améliorer encore le procédé, en parallélisant le calcul sur la hauteur de l'arbre. Il s'est avéré que ce n'était pas aussi efficace, nous avons donc abandonné cette partie.

Pour compiler le programme :

```
~ $ sudo apt-get install libgmp-dev cmake git
~ $ git clone https://github.com/RandomGuys/fact.git
~ $ cd fact
~/fact $ mkdir build
~/fact $ cd build
~/fact/build $ cmake ..
~/fact/build $ make
~/fact/build $ sudo make install
```

Il y a également un *man* qui s'installe, vous pouvez donc consutler l'aide avec la commande `man fact`.

Pour utiliser ce programme, la ligne de commande est la suivante :

```
$ fact -m mes_moduli --fullram|--files [--format hexa|decimal]
```

5 Conclusion des résultats produits

5.1 Fichier de résultats

Les résultats se trouvent dans le fichier `moduli_p_q` construit de la façon suivante :

```
modulus, facteur commun, facteur déduit
...
```

5.2 Premiers chiffres

Voici les premiers chiffres obtenus des scans :

	Scan 1	**Scan Sonar**
moduli	85 636	523 017
vulnérables	104 (0,12%)	2 382 (0,46%)
facteurs communs	11	757
autres facteurs	104	2 382
total facteurs	115	3 139

En comparant avec les résultats du projet factorable de l'Université de Californie et de l'Université du Michigan, nous obtenons un pourcentage de clefs vulnérables assez proche. En effet, ils ont obtenu 0,50% de clefs vulnérables et nous en avons trouvées 0,46%

Pour la suite de l'analyse des résultats, nous nous sommes focalisés sur le scan Sonar.

5.3 Entropie

Le graphe suivant représente le pourcentage d'entropie des 757 facteurs premiers trouvés. Ce niveau d'entropie a été calculé selon l'entropie de Shannon (représentation binaire des clefs). Nous nous sommes basés sur la bibliothèque `libdisorder` (`http://libdisorder.freshdefense.net/`) qui fournit un programme de tests. Ce dernier a été modifié pour nos besoins et pour pouvoir injecter les résultats dans gnuplot pour obtenir le graphe suivant :

Entropie moyenne des facteurs communs : 0.998518
Entropie moyenne des second facteurs : 0.998551

5.4 Tailles des clefs

Dans nos résultats, nous pouvons constater que seules deux tailles de clefs apparaissent : 512 et 1024 bits. Voici les chiffres :

	512 bits	**1024 bits**
Total	2 345	318 558
Vulnérables	6 (0,26%)	2 376 (0,75%)

Nous avions également d'autres tailles de clefs en entrée mais leur nombre devait être trop petit par rapport à l'espace des clefs de cette taille pour obtenir des facteurs communs.