Etude et analyse de la génération de clés: Butterfly keys

Abdessalam Boulahdid et Selim Ben Ammar

Le 28 Juin 2017

1 Introduction

Le cryptage a été utilisé comme méthode de protection de la vie privée informatique depuis 1977. La communication joue un rôle croissant dans les organisations, en particulier les banques, l'industrie, le commerce, les télécommunications, etc. Ces organismes peuvent échanger des secrets grâce à la cryptographie.
Le cryptage consiste à modifier le trafic réseau pour qu'un message transmis ne peut être reconstruit que par un destinataire légal. Chaque jour, de nouvelles organisations sont soumises à une variété d'attaques informatiques basées sur la cryptographie telle que le dernier ransomware WannaCry. Quelle est la place des Butterfly Keys dans cet écosystème?

Qu'est ce qu'une Butterfly key?

Butterfly Keys est une nouvelle méthode cryptographique qui permet à un appareil (une voiture par exemple) de demander un grand nombre de certificats à partir d'un simple seed, une paire de clés ECC et une paire de clés AES pour la signature et le cryptage. Tous les calculs coûteux ne se feront pas au niveau de l'appareil. Grâce aux Butterfly Keys, nous réduisons la durée du chargement. Ainsi, nous pouvons faire des requêtes lorsqu'il n'y a qu'une connectivité limitée.

À quoi servent les Butterfly keys?

Vous pouvez avoir recours aux Butterfly keys lorsque vous désirez:

- Éviter toute répudiation : Personne ne connaît la clé privée de l'appareil, inclus l'authorité de certification (CA).
- Minimiser les calculs au niveau de l'appareil : Une voiture n'est utilisée que 5% du temps. Donc, nous ne voulons pas utiliser le CPU et la batterie pour générer des clés qui ne seront pas utilisés par la suite.
- Minimiser la taille du téléchargement : la génération des certificats peut prendre du temps. Grâce aux téléchargements de petite taille, l'appareil peut générer et soumettre une demande sur une connexion de données à faible bande passante juste avant de collecter les certificats sur une connexion à large bande passante.

Notre projet

Notre projet peut se décomposer en trois grandes parties. D'abord, nous avons fait un travail de recherche afin de comprendre le mécanisme des Butterfly Keys. Après avoir bien assimilé le fonctionnement de l'algorithme, nous sommes passés à l'étape de l'implémentation en Java. Enfin, nous avons utilisé notre implémentation pour tester la performance de l'algorithme des Butterfly Keys.

2 L'algorithme implémenté

Notre projet est basé sur l'algorithme décrit dans : `https://wiki.campllc.org/display/SCP/SCP1%3A+Butterfly+Keys`
Le schéma suivant résume **les échanges** qui ont lieu entre les 3 composants de l'écosystème : Le Device (une voiture connectée par exemple), Le RA (registration authority) et le CA (certification authority).

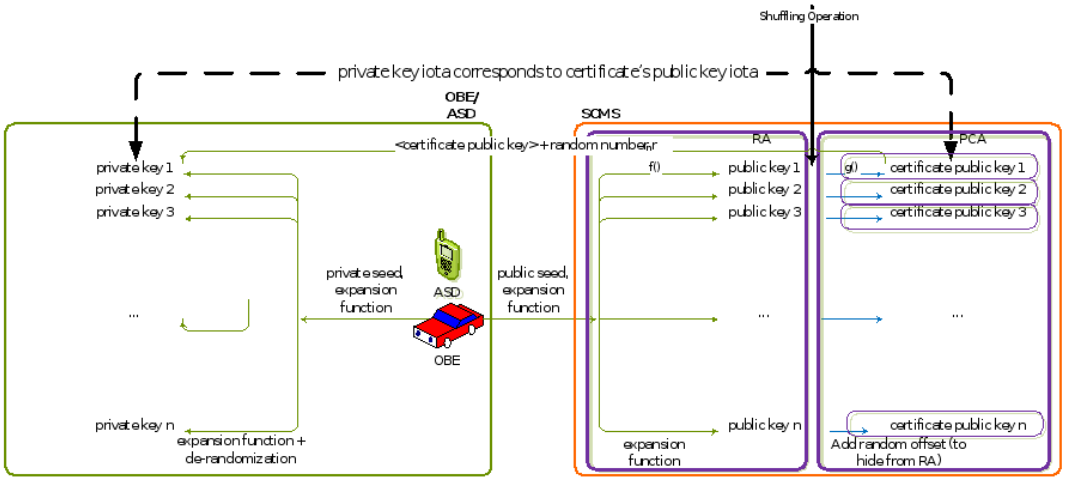

Figure 1: Les étapes de l'algorithme Butterfley Key

Présentation de l'algorithme

Il existe un "point de base" convenu appelé **G** (c'est une pratique courante pour la cryptographie à courbe elliptique). L'appareil, une voiture connectée par exemple, génère deux paires de clés ECC **(a, A = aG)** (seed pour les clés de signature) et **(p, P = pG)** (seed pour les clés de chiffrement des certificats) et les descriptions des deux fonctions d'extension f1 **(clé de signature ck)** et f2 **(la clé de cryptage ek)**. Le véhicule stocke a, p, et les descriptions de f1 et f2, et envoie A, P, et la description de f1 et f2 cryptées sur le SCMS.
Les fonctions f1 et f2 sont conçues pour être cryptographiquement sécurisées, ce qui signifie à peu près que la sortie semble aléatoire, de sorte que, compte tenu de deux valeurs de f1 (l), l (ou, f2 (l), l), un tiers ne peut pas indiquer si les deux valeurs ont été générées par la même version de f1 (ou, f2), ou par différentes versions. Le calcul des fonctions d'expansion se fait ainsi:

- $f1(k,l) = f1_int(k,l) mod I$: Afin de calculer f1, nous avons donc besoin de trouver l'ordre de la courbe elliptic I ainsi le time period l.
- $f1_int(k,l)$ est la représentation d'un entier big-endian de $(AES(k,x+1)XOR(x+1))||(AES(k,x+2)XOR(x+2))||(AES(k,x+3)XOR(x+3))$:

Le calcul de $f1_int$ est une triple concaténation d'XOR. $(x+1), (x+2) et (x+3)$ sont obtenus en incrémentant simplement x par 1 à chaque fois.

- x est un entier de 128 bits dérivé de manière du time period I tel que: $x1 = (0^32||i||j||0^32)$ et $x2 = (1^32||i||j||0^32)$.

Arrivé à ce stade de l'algorithme, le device envoie au SCMS a,p,ck et ek. Le SCMS est formé par une authorité d'enregistrement RA et de certification CA. Maintenant, le RA (registration authority) a la capacité de générer un grand nombre de points dérivés grâce aux fonctions d'expansion.

Bi = A + f1 (ck, l) * G, avec A = aG (clés de signature)
Qi = P + f2 (ek, l) * G, avec P = pG (clés de cryptage)
Les clés privées correspondantes seront :
bi = a + f1 (ck, l) (clés de signature)
qi = p + f2 (ek, l) (clés de cryptage)

Étant donné que le SCMS ne connaît pas la valeur d'origine de a (ou, p), il ne peut pas connaître les valeurs bi (ou qi) ainsi, seul le véhicule connaît les correspondants clés privées. De plus, comme les fonctions d'extension sont cryptographiquement sécurisées, toute personne qui ne connaît pas la description de f1 ne peut pas indiquer si deux clés publiques de signature différentes appartiennent à la même série Bi ou à différentes séries. Cela signifie que la RA, décrite en détail plus loin, peut utiliser en toute sécurité f1 pour créer une liste élargie de clés publiques de signature à envoyer à l'Autorité de certification et l'autorité de certification ne pourra pas indiquer que les clés appartiennent au même véhicule.

Le périphérique génère deux clés AES 128 bits ck et ek, pour les fonctions d'extension des clés de signature et de cryptage, respectivement, et deux paires de clés "cocon":
(A, A = aG) utilisé pour la signature, c'est-à-dire A pour être placé dans le certificat
(P, P = pG) utilisé pour le cryptage du certificat

La série de clés publiques de cryptage pour chiffrer la réponse du certificat, associe chaque Bi avec le correspondant Qi, et envoie les paires Bi, Qi à l'AC. CA ne sait pas quelles clés publiques proviennent du même périphérique, mais RA sait quelles clés publiques sont dans les requêtes, donc CA doit aléatoirement modifier les clés publiques pour les masquer de RA. Pour chaque demande, CA génère un entier aléatoire unique c et définit la clé publique dans chaque certificat à la valeur du Batterfly (Bi + cG). CA utilise ensuite Qi pour chiffrer la réponse, qui contient:

- Certificat contenant la clé publique (Bi + cG)
- Contribution de CA à la clé privée, c

Le RA envoie le message chiffré au au device avec le l correspondant. L'appareil utilise ek, p, l pour calculer qi. Il utilise qi pour décrypter la réponse et récupérer le certificat contenant la clé publique (Bi + cG) et la contribution de CA à la clé privée, c. Il utilise ensuite ck, a, l pour calculer bi. La clé privée pour le certificat est alors: Clé privée du Batterfly key = bi (calculé ci-dessus) + c (fourni par CA) Le périphérique devrait à ce stade vérifier que la clé privée récupérée correspond à la clé publique certifiée par le certificat pour s'assurer qu'il a été envoyé le certificat correct. Cela signifie que le périphérique a obtenu un ensemble de certificats tels que: Seul l'appareil connaît les clés privées.

Les difficultés rencontrées

Tout d'abord, on a eu du mal à installer la bibliothèque Bouncy Castle. En effet la documentation relative à cette bibliothèque est souvent imprécise voire même inexistante. On était donc contraint à travailler avec une ancienne version contenant des fonctions mentionnées comme obsolètes, mais cela ne nous a pas empêché d'implémenter l'algorithme. Toutefois, on a pas réussi à faire l'étape consistant à chiffrer les certificats par les clés de cryptage Qi car cela requiert l'utilisation du ECIES "Elliptic Curve Integrated Encryption Scheme" dont l'implémentation n'était pas claire sur Bouncy Castle.

Une autre difficulté est que la bonne implémentation de l'algorithme nécessite trois entités distinctes : Le device (la voiture), le RA et le CA. De plus le device ne possède normalement pas une puissance de calcul et de stockage comparable à un ordinateur. Or pour notre cas, tout est fait sur un même ordinateur, l'algorithme perd donc tout son intérêt : La comparaison de la génération directe de clés au niveau du device est bien sûr plus rapide que l'utilisation du Butterfly key, cela est dû au fait que les deux méthodes sont faites par le même processeur et sur la même machine.

Notre implémentation

On a implémenté l'algorithme en java en utilisant la bibliothèque Bouncy Castle. Les trois entités (device, RA et CA) sont représentées par des classes d'un même projet java et les échanges entre ces classes se font de manière directe, c'est à dire sans utiliser un réseau pour la transmission de données.

Le CA est identifié par une paire de clé asymétrique et est utilisé pour signer les certificats, le device ne génère que deux paires de clés asymétrique A et P. Toutes les clés sont générées en utilisant la courbe elliptique "prime256v1", mais le code supporte aussi les autres courbes prises en charge par Bouncy Castle.

Pour le calcule du time period I qui sert au calcul des fonctions d'expansion, on a utilisé une formule qui produit une chaîne de 64bits en prenant comme entrée le date d'exécution du code. En ce qui est du c ajouté aux clés publique par le CA, on a préféré utilisé la classe Random de java.

3 Conclusion

Notre code permet de tester le bon fonctionnement de l'algorithme. Cependant, pour évaluer les performances du Butterfly Key, il faut avoir à disposition Un réseau véhiculaire (V2I : Vehicle to Infrastructure) ainsi que plusieurs véhicules équipés d'OBU (la partie véhicule du système V2I) afin de tester son efficacité, sa rapidité et sa fiabilité.