RAPPORT

Etude d'une vulnérabilité dans une implémentation de ECDSA dans Java : CVE 2022 – 21449

Zinoni Alexander

Tuteur : Talbot Jean-Marc

Université Aix-Marseille

Stage réalisé sur juin juillet 2022

Table des matières

Introduction

La cryptographie symétrique nous a longtemps permis d'échanger des messages en secret, mais avec la globalisation et le besoin de communiquer à distance, on a été obligé d'adapter nos techniques de communication.

En effet, c'est seulement dans les 50 dernières années, grâce au chiffrage asymétrique que nous avons pu pour la première fois échanger de l'information à distance de façon entièrement sécurisée. Par les mêmes techniques on a aussi réussi à déduire des algorithmes pour signer de façon digitale un message. C'est un des ces algorithmes de signature, nommé le Elliptic Curve Digital Signature Algorithm (ECDSA), auquel on s'intéresse dans ce document.

Ce rapport vise à détailler au lecteur toutes les notions nécessaires pour comprendre le fonctionnement d'une signature digitale grâce à ECDSA sans prérequis nécessaires. On étudiera pourquoi certaines versions passées de Java implémentent cet algorithme incorrectement.

1. Préface : Notions importantes

Avant de commencer, il est important d'introduire la notion de « difficulté » d'un calcul. De manière simplifiée, on considère qu'un calcul est « difficile » si avec tous les ordinateurs de la planète rassemblés il faut encore des milliers d'années pour le résoudre.
Plus généralement, si un problème de calcul protège un message important, ce problème est « difficile » si on estime que sa résolution avec les outils et connaissances actuelles est tellement longue que le contenu du message ne sera plus important ou d'actualité.

Voici les concepts nécessaires pour bien comprendre sur quels principes cryptographiques se base l'ECDSA :

1.1 La cryptographie symétrique vs asymétrique

1.1.1 La cryptographie symétrique

La cryptographie symétrique est la technique de chiffrage la plus ancienne, elle nécessite la même clé pour chiffrer et déchiffrer un message.
Par exemple, Jules César l'a utilisée pour créer le *Caesar Shift*. Ce chiffrage revient à décaler toutes les lettres du message d'un certain nombre *k* de fois dans l'alphabet. Par example, avec une clé k = 1, le message 'abc' devient 'bcd', on décale toutes les lettres de k = 1 position dans l'alphabet. Pour le déchiffrer, on redécale simplement toutes les lettres de k= 1 position dans l'autre sens pour retrouver 'abc'. Au fil de l'histoire, d'autres techniques de chiffrage symétrique ont été inventes, les plus populaires sont le Vigenère Cipher, Enigma, ou plus récemment, l'AES (Advanced Encryption Standard).

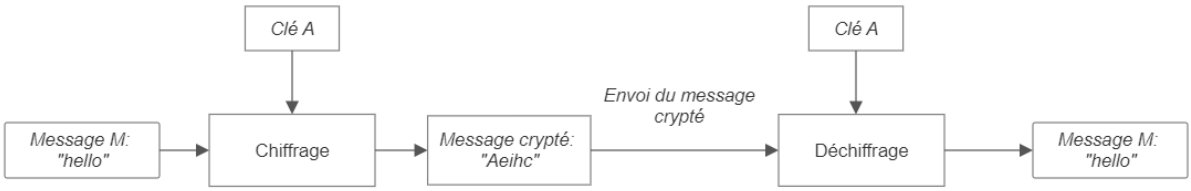

Figure 1

L'avantage du chiffrage symétrique est que le chiffrage et le déchiffrage des messages peut se faire plus efficacement par rapport au chiffrage asymétrique. De plus, si un ennemi ne possède pas la clé, il sera très difficile pour lui de pouvoir déchiffrer le message. En revanche, si on veut envoyer un message à une personne à distance, il faut trouver un moyen de pouvoir lui transmettre la clé sans que personne ne puisse la lire. Ce problème se résout par la cryptographie asymétrique.

1.1.2 La cryptographie asymétrique

La cryptographie asymétrique, contrairement à celle symétrique, nécessite deux clés, l'une privée et l'autre publique. Ces deux clés sont l'inverse l'une de l'autre : lorsque l'une chiffre un message, seule l'autre peut le déchiffrer.

La construction d'une paire de clés se fait à partir d'une fonction trappe (facile à réaliser dans un sens, difficile à calculer dans l'autre). Un exemple de telle fonction est la factorisation, en prenant deux entiers premiers p et q, il est facile de calculer leur produit e, mais difficile de retrouver p et q en partant de e avec p et q suffisamment grands. Dans cet exemple simplifié, e serait notre clé publique et la paire (p, q) notre clé privée.

La factorisation de nombres premiers est le principe sur lequel se fonde l'algorithme de cryptage RSA [1](Rivest – Shamir – Adleman). En pratique, si je voulais envoyer un message à un ami Pierre sans que personne d'autre puisse le lire, je devrais prendre la clé publique de Pierre, (qui est accessible par tout le monde) chiffrer mon message avec et lui envoyer. A son tour, Pierre utilise sa clé privée (qu'il est le seul à connaitre) et déchiffre le message envoyé.
Il est très important que Pierre garde sa clé privée secrète, il doit être le seul à la connaître.

De façon plus importante, le chiffrage asymétrique nous permet de communiquer à distance, sans avoir à faire un échange de clé de chiffrage au préalable (contrairement au chiffrage symétrique). En observent ce diagramme (Figure 1), on observe que tout le monde peut crypter un message avec la clé publique du récipient, mais personne ne peut le décrypter à par le récipient lui-même car il est le seul possesseur de la clé privée.

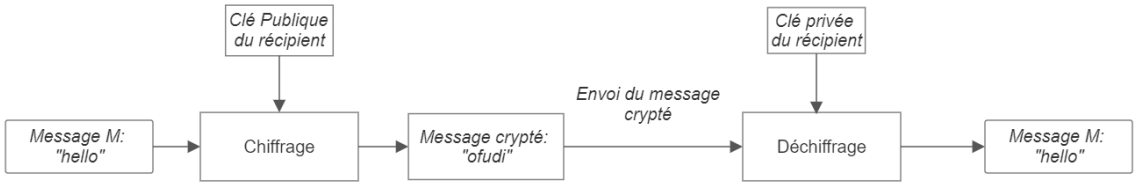

Figure 2

1.2 Les fonctions de hachage

1.2.1 Propriétés

Une fonction de hachage prend en argument un ensemble de données de taille arbitraire et retourne un ensemble de données de taille fixe.

Ses propriétés :

- Déterministes. Une même entrée aura toujours la même sortie.

- Rapide. Elle peut se calculer "facilement" par un ordinateur
- Irréversible. Sauf par « brute force », ce qui est très difficile, car ça revient à essayer toutes les possibilités.
- Résistant aux collisions. Il y a une très faible probabilité que deux entrées différentes aient jamais la même sortie.
- La modification même très légère du message d'entré modifiera considérablement la valeur de sortie : l'effet d'avalanche.

Ce sont donc des fonctions à sens unique, dont il est impossible d'inverser autrement que par une approche force brute.

De plus, ces fonctions sont:

- Résistantes à la préimage: pour une image donnée (*h*), il est très difficile de retrouver le message *m* tel que *hachage(m) = h*.
- Résistantes à la seconde préimage: pour un message *m1* donné, il est très difficile de trouver un autre message *m2* tel que *hachage(m1) = hachage(m2)*.

Par exemple, la fonction SHA2-256 [2] (Secure Hash Algorithm), prend tout texte, est retourne un hash de 256 bits. Voici un exemple implémenté en Python 3:

```
message = "Salut !"
message2 = "salut !"
print(message, 'devient', hashlib.sha256(message.encode('utf-8')).hexdigest())
print(message2, 'devient', hashlib.sha256(message2.encode('utf-8')).hexdigest())
```

Affiche la valeur :

```
Salut ! devient 212febfe999494533c29f6bf73cd2ed8d1343f023b8e6a04a3421be28da29b89
salut ! devient 9052306a326f9c6ff7695413af85854be5bd931810b25fdc9a984e218d872bf1
```

1.2.2 Comment fonctionnent les fonctions hachage

L'algorithme SHA-1 est une fonction de hachage sur laquelle s'est bâtie la famille de fonctions SHA-2, ayant un principe de fonctionnement très similaire à SHA-1.

Pour fonctionner, SHA-1 prends en argument une donnée, et la divise en morceau de 512 bits (si la taille de la donnée n'est pas un multiple de 512, on ajoute des bits à la fin pour qu'elle le devienne, an anglais « *padding* »). (*1* sur la figure 1)

La fonction a un état initial, d'une longueur de 160 bits, divisé en 5 morceaux de 32 bits. Cet état initial va être modifié un certain nombre de fois et deviendra notre sortie finale de taille 160 bits. (*2*)

Pour chaque morceau de 512 bits, SHA-1 prends un "sous-morceau" (*m1*) des 512 bits avec un bout d'état initial et les passe dans une fonction de compression (*3*). Cette fonction de compression est au cœur de l'algorithme et détermine la sécurité de la fonction de hachage.
Après avoir effectué 80 itérations de fonction de compression, on arrive à un résultat d'une longueur de 160 bits. Pour finir de hacher ce morceau de 512 bits, on finit par faire une opération de somme entre notre résultat et l'état initial tout en prenant le modulo pour ne pas dépasser 160 bits (la prise du modulo est une des raisons pour laquelle SHA1 est irréversible, car on perdrait de l'information sinon) (4).

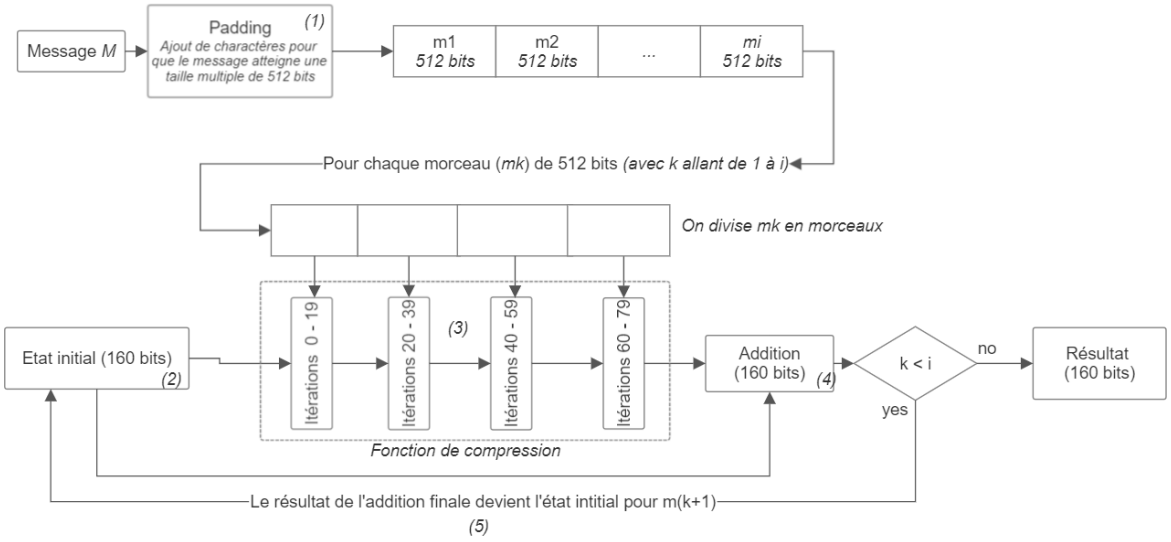

Figure 3

Si le message original était exactement de longueur 512 bits, notre travail est fini et le résultat de la somme précédente est bien la valeur du hash de notre message. Sinon, on répète l'opération en prenant cette fois comme état initial le résultat obtenu (*5*).
Cet algorithme de SHA-1 fonctionne de la même manière que SHA-2, la sécurité de SHA-2 est augmentée par l'utilisation de clés plus grandes et est donc plus difficile à attaquer par force brute.

Pour donner une idée, les chances de trouver deux messages ayant le même hash sont de 2^{80} pour SHA1, pouvant être réduites jusqu'à 2^{60} avec certaines attaques. En revanche, pour SHA-2, ces chances sont de 2^{128}. On n'utilise donc plus SHA-1 aujourd'hui car elle n’est pas assez sûre. En effet, Gaëtan Leurent et Thomas Peyrin ont réussi à trouver deux messages différents avec la même sortie par SHA-1. [3]

1.3 Les signatures digitales

En reprenant l'exemple du paragraphe 1.1.2, pour que Pierre soit sûr que le message que je lui ai envoyé vienne bien de moi, je dois le signer. Pour cela, en annexe avec mon message chiffré par la clé publique de Pierre, j'ajoute une copie hachée de celui-ci que je chiffre ensuite avec ma clé privée. Pour vérifier l'authenticité et l'intégrité du message, Pierre doit décrypter l'annexe avec ma clé publique, et comparer ce résultat avec le message haché par le même algorithme que j'ai utilisé. Si ces deux derniers sont identiques, alors Pierre peut être certain que le message provient bien de moi. Voici un schéma qui résume tout [4]:

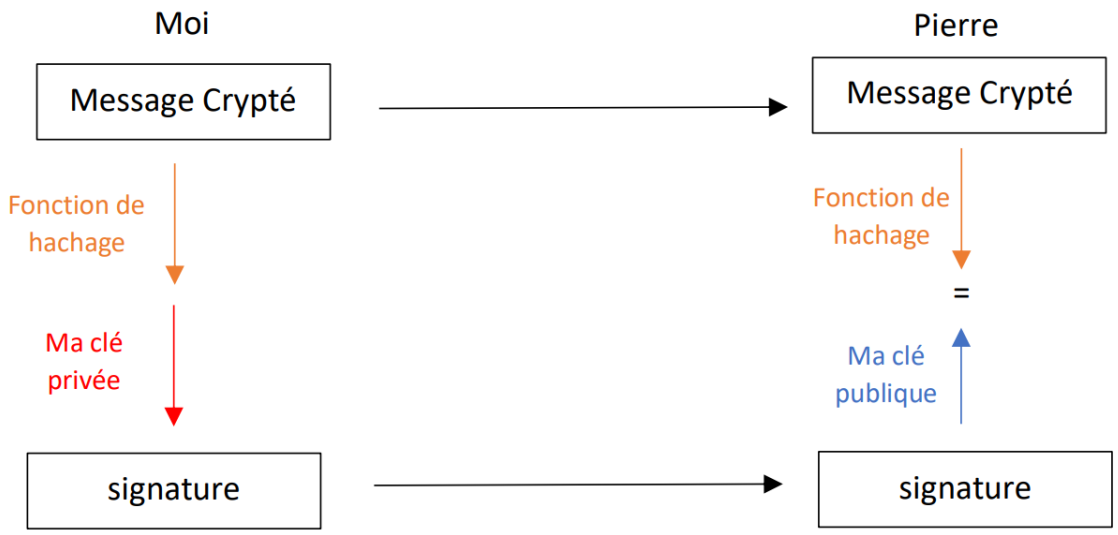

Figure 4

2. Cryptographie sur des Courbes Elliptiques

2.1 Qu'est-ce qu'un courbe elliptique ?

Une courbe elliptique *C* est modélisée par l'équation suivante (avec a et b deux paramètres):

$$C: y^2 = x^3 + ax + b$$

En prenant comme paramètres a = -1 et b = 1, on obtient la figure suivante :

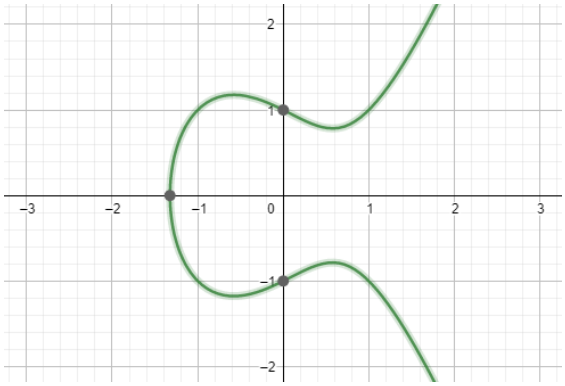

Figure 5 - géogebra

Définissons l'addition de deux points G et G1 appartenant à la courbe elliptique C. En traçant la droite (G ; G1), celle-ci recoupe la courbe en un troisième point G'2 .En prenant G2 symétrique de G'2 par rapport à l'axe des abscisses, on définit l'addition : G2 = G + G1. *Voir figure 6.*

Si on cherche G + G, la tangente à la courbe en G recoupe la courbe en un deuxième point G'1. G1 = G + G est le point symétrique à G'1 par rapport à l'axe des abscisses. *Voir figure 7.* [6]

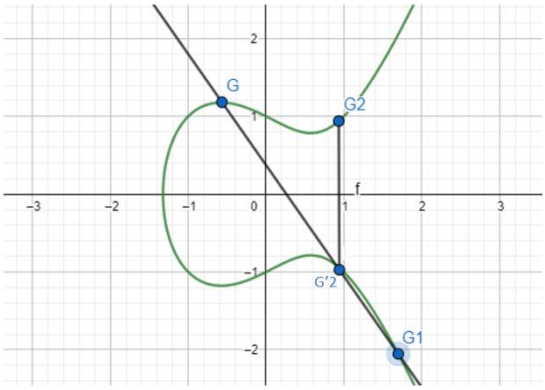

Figure 6 : Ici, on cherche G2 = G + G1

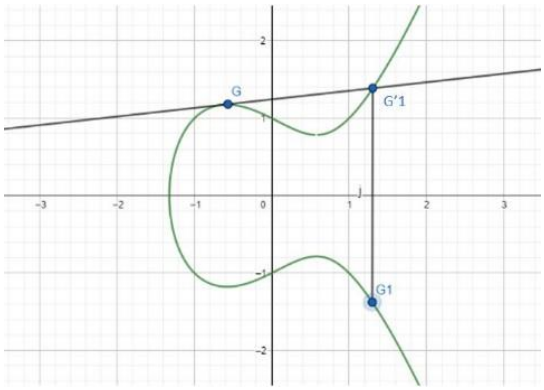

Figure 7 : Ici, on cherche G1 = G + G

On peut aussi définir la multiplication d'un point G par un scalaire n, tel que n·G = G + G + … + G *(n fois)*.

2.2 Courbes elliptiques sur un corps fini

Pour représenter des courbes elliptiques de manière finie, on choisit un corps fini $\mathbb{Z}/p\mathbb{Z}$, avec p premier. L'équation d'une courbe elliptique reste la même :

$$C: y^2 = x^3 + ax + b$$

Voici une représentation graphique de la courbe sur $\mathbb{Z}/p\mathbb{Z}$ au triplet de paramètres, a = 26, b = 3, p = 31 (26, 3, 31) :

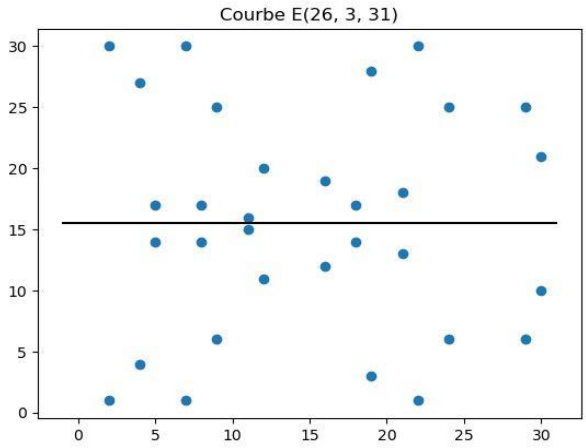

Figure 8

La règle d'addition et le point à l'infini maintiennent les mêmes propriétés que sur un courbe sur $\mathbb{R}$.

2.3 Point à l'infini

Le point à l'infini *O* est une convention utilisée lorsque l'addition de deux points est « impossible », c'est-à-dire lorsqu'ils ont la même ordonnée.

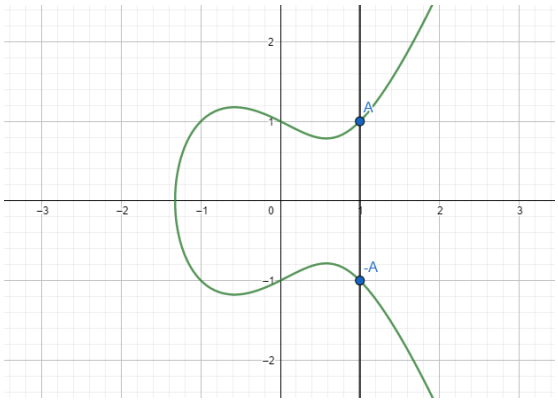

Figure 9

En essayant d'additionner les deux points A et -A (*Figure 2*) on remarque qu'il est impossible de trouver une troisième intersection entre la droite (A(-A)) et la courbe. Par convention, on note que A + (-A) = *O*.

On obtient aussi le point à l'infini en multipliant tout point par 0 :

$$0 \cdot A = O$$

Le point à l'infini est l'élément d'identité d'addition de deux points :

$$A + O = A$$

$$O + O = O$$

2.3.1.1 Ordre d'une courbe elliptique

Avant d'introduire l'algorithme ECDSA, il est très important de comprendre ce qu'est l'ordre d'une courbe elliptique. L'ordre d'une courbe elliptique sur $\mathbb{Z}/p\mathbb{Z}$ est simplement le nombre de points qui la compose (une courbe elliptique avec 23 points est donc d'ordre 23). Or, il est très inefficace - surtout pour une grande courbe - de compter son nombre de points, on utilise donc l'algorithme de Schoof[1] pour éviter de les énumérer.

2.3.1.2 Ordre d'un sous-groupe induit par un point de la courbe

En ajoutant un point à lui-même un certain nombre de fois, on commence à apercevoir un cycle. Par exemple [8], les multiples de P(3, 6) sur la courbe E = (2, 3, 97) donne les résultats suivants:

0P = *O*	**5P = *O***	**10P = *O***
1P = (3, 6)	6P = (3, 6)	11P = (3, 6)
2P = (80, 10)	7P = (80, 10)	...
3P = (80, 87)	8P = (80, 87)	
4P = (3, 91)	9P = (3, 91)	

En additionnant P(3, 6) et 4P(3, 91), on obtient bien le point à l'infini *O* car P et 4P ont la même abscisse.

On peut alors écrire pour tout entier k:

(5k)P = O
(5k + 1)P = (3, 6)
(5k + 2)P = (80, 10)
(5k + 3)P = (80, 87)
(5k + 4)P = (3, 91)

[1] **L'algorithme de Schoof** est un algorithme efficace pour compter les points de courbes elliptiques sur les corps finis. L'algorithme a été publié par René Schoof en 1985, ce qui a constitué une avancée majeure, s'agissant du **premier algorithme déterministe polynomial pour le comptage de points.** [5]

D'où kP = (k mod 5) P.

On peut donc prouver que les multiples de P sont un sous-groupe cyclique[2] des points de la courbe (2, 3, 97), contenant les points (3, 6), (80, 10), (80, 87), (3, 91). P est le générateur de ce sous-groupe.

L'ordre d'un sous-groupe induit par un point P est le plus petit entier naturel n (sauf 0) tel que n·P = O (O étant le point à l'infini). [9]

2.3.2 Intérêt des courbes elliptiques

En pratique, on peut générer à partir d'une courbe elliptique une paire de clés. En effet, l'addition répétée d'un point à lui-même sur une courbe est une fonction trappe.

Pour un point G sur une courbe elliptique C et un entier n, les coordonnées du point nG (= n·G) sont faciles à calculer. En revanche, si on connait seulement les paramètres de la courbe C, ainsi que les coordonnées du point nG, il est très difficile de retrouver le nombre n ; d'où la fonction trappe.

On définit n comme notre clé privée, et les coordonnées (x, y) de nG comme notre clé publique. Afin de comprimer la clé publique, d'après la symétrie d'une courbe elliptique, qu'il suffit de conserver la coordonnée *x* de nG en ajoutant un bit pour indiquer si *y* est positif ou négatif pour maintenir toute l'information du point. [7]

2.3.3 Conventions sur les courbes elliptiques

Par convention, une courbe elliptique *C* est définie par les paramètres suivants :

- a et b : les paramètres de l'équation de la courbe ($y^2 = x^3 + ax + b$)
- p : un nombre premier qui définit le corps fini : $\mathbb{Z}/\mathrm{p}\mathbb{Z}$
- G : un point sur la courbe qui sert de générateur pour calculer une paire de clé
- n : l'ordre du groupe des points engendrés par le générateur G

Une courbe elliptique *C* s'écrira donc de cette manière :

$$C = (a, b, p, G, n)$$

[2] **Définition d'un groupe cyclique** — Un groupe G est dit cyclique si :
- G est d'ordre fini
- $\exists\, x \in G, \forall\, y \in G, \exists\, n \in \mathbb{Z}, y = x^n$

3. ECIES – Elliptic Curve Integrated Encryption Scheme

L'ECIES est un algorithme de chiffrage asymétrique qui permet grâce à secret commun de pouvoir échanger un message chiffré de manière symétrique. Imaginons que Alice veuille envoyer un message chiffré par ECIES à Bob. Pour s'y prendre, il faut d'abord initialiser quelques valeurs :

- Une courbe elliptique E au paramètres publiques (comme NIST P-256[10]) avec les paramètres (a, b, p, G, n).
- Une clé publique de Bob : $K_b = k_bG$ avec k_b comme clé privée.

Pour que Alice chiffre son message *m*, il faut qu'elle :

- Sélectionne un nombre aléatoire *r* et calcule le point R = rG.
- Calcule S (appelé le secret commun) : $S = P_x$, avec $P = (P_x, P_y) = rK_b$.
- Utilise une fonction de dérivation de clé (KDF[11]) pour trouver une clé α à partir du secret commun S.
- Chiffre le message grâce à la clé α précédente. Pour chiffrer le message, on utilise un algorithme de chiffrage symétrique comme l'AES (Advanced Encryption Standard).
- Alice peut envoyer le message chiffré à Bob !

Une fois le message reçu, pour déchiffrer le message, Bob doit :

- Retrouver le même secret S qu'Alice en calculant cette fois $S = P_x$, avec $P = (P_x, P_y) = k_bR$. On retrouve bien le même nombre S car :

$$P = k_bR = k_brG = rk_bG = rK_b = P$$

- Utiliser la même fonction KDF pour retrouver la clé à partir de S.
- Déchiffrer le message.

4. ECDSA – Elliptic Curve Digital Signature Algorithm

Le principe de l'ECDSA est de pouvoir attribuer à tout document, une paire de valeurs (r, s) comme signature de ce document. Cette signature témoigne:

- De l'intégrité du document, signifiant qu'il n'a pas été modifié après son envoi.
- De l'authenticité du document, prouvant qu'il provient bien d'une certaine personne.

Si Alice veut signer un message et l'envoyer à Bob, elle procède par calculer deux valeurs, r et s dépendant de sa clé privée et du hash de son document.
En recevant ces valeurs, Bob pourra vérifier grâce à la clé publique d'Alice que c'est bien elle qui a signé le document, et grâce au hash qu'il n'a pas été modifié depuis son envoi.

4.1 Calcul de la signature

Pour que Alice calcule la signature de son message *m*, elle doit d'abord avoir une paire de clés générés sur une courbe standard (a, b, p, G, n), avec Q_a sa clé publique, et d_a sa clé privée. Elle procède alors par :

1. Calculer un hash de son message m : e = hash(m).
2. Choisir un nombre aléatoire *k*, dans l'intervalle [1, n-1]
3. Calculer le point $P(x_1, y_1) = k \cdot G$. (avec G le générateur de la courbe). On note alors $r \equiv x_1 \ mod\, n$.
4. Calcule $s = k^{-1}(e + rd_a) \ mod\, n$. Si s = 0, il sélectionne une autre valeur de k et on répète les deux calculs précédents.
5. La signature est donc la paire (r, s). Alice peut donc envoyer son message *m* à Bob, accompagné de la paire de valeurs (r, s).

4.2 Vérification de la signature

Une fois le message *m* et la signature reçus, Bob vérifie la signature en :

1. Vérifiant que r et s sont bien des entiers dans l'intervalle [1, n-1]. Si ce n'est pas le cas, la signature est invalide.
2. Calcule le hash du message *m* de la même manière que Alice : e = hash(m).
3. Calcule $u_1 = es^{-1} \bmod n$ et $u_2 = rs^{-1} \bmod n$.
4. Calcule le point $P(x_1, y_1) = u_1 \cdot G + u_2 \cdot Q_a$. Si P est le point à l'infini, alors la signature est invalide.
5. Si et seulement si $r \equiv x_1 \bmod n$, alors la signature est valide.

4.3 Exemple pratique

Montrons en pratique comment se calcule la signature du message m = 'Bonjour :)' sur une courbe simplifiée aux paramètres (26, 3, 31, (21, 18), 11)[3].

- La première étape est de calculer un hash du message 'Bonjour :)'. Pour cela, on utilise la fonction SHA3 qui retourne un hash de 256 bits. Dans cet exemple simplifié, on se contente de rogner ce nombre en gardant uniquement les 5 premiers chiffres. On obtient e = SHA3_256(m) = 10125.
- Choisissons pour k une valeur aléatoire dans l'intervalle $[1, n-1] = [1, 10]$. On a k = 10.
- Calculons le point $P = k \cdot G = 10 \cdot (21, 18) = (21, 13)$. On a donc $x_1 = 21$ et $r \equiv 21 \bmod 11$ donc r = 10.
- Calculons $s = k^{-1}(e + rd_a)\ mod\ n$. On a d'abord que l'inverse de k mod 11 est égal à 10. D'où : $s = 10(10125 + 10 * 3)\ mod\ 11 = 9$.

La signature du message m est la paire de valeurs : $(r, s) = (10, 9)$. On peut finalement envoyer le message accompagné de la signature.

Une fois reçu, il faut vérifier la signature :

- On s'assure d'abord que les valeurs de la signature sont dans l'intervalle $[1, n-1] = [1, 10]$; ce qui est bien le cas vu que $r = 10$ et $s = 9$.
- On recalcule le hash du message 'Bonjour :)' avec la même fonction, on obtient $e = \text{SHA3\_256(m)} = 10125$.
- Calculons l'inverse mod n de s : $s^{-1} \bmod n = 9^{-1} \bmod 11 = 5$
- Calculons $u1 = e * s\text{-}1 \bmod n = 10125 * 5 \bmod n = 3$ et $u2 = r * s\text{-}1 = 10 * 5 \bmod 11 = 6$.
- Il nous faut maintenant calculer le point P d'équation : $P(x_1, y_1) = u_1 \cdot G + u_2 \cdot Q_a = 3 \cdot (21, 18) + 6 \cdot (22, 30) = (21, 13)$. Il s'agit là d'addition de deux points sur une courbe elliptique et de multiplication de points par des scalaires. On a alors $x1 = 21$.
- Il ne reste plus qu'à vérifier que $r = x_1\ mod\ n$:

$21\ mod\ 11 = 10 = r$. La signature est bien valide.

[3] a = 26, b = 3, p (corps fini) = 31, G (générateur) = (21, 18), n (ordre) = 11

5. Vulnérabilité – CVE 2022 21449

En avril 2022, Oracle annonce une vulnérabilité (identifiée par CVE-2022-21449[12]) dans la librairie « Security » et plus particulièrement dans l'algorithme de signature digitale ECDSA.
Cette vulnérabilité ne provient pas dans l'ECDSA lui-même, mais plutôt dans une mauvaise implémentation.
L'exploitation de cette faille compromet toute intégrité d'une signature, étant donné que l'algorithme acceptait une signature nulle (r = s = 0) comme une signature valide quel que soit le document.
Comment cette erreur survient-elle ?

1. ~~Vérifier que r et s sont bien des entiers dans l'intervalle [1, n-1]. Si ce n'est pas le cas, la signature est invalide.~~

> ***Java ne vérifie pas si r = s = 0***
>
> *La première raison, et la plus grave, est qu'il a été oublié d'introduire un contrôle des valeurs de r et s avant de de réaliser la suite de l'algorithme. Dans la version originale de l'algorithme, on remarque que la première étape et de s'assurer que r et s sont bien dans l'intervalle [1, n], ce qui aurait permis de réparer l'erreur.*
> *Dans les versions précédentes à Java 15, ce contrôle était présent, mais c'est dans la réécriture du code des courbes elliptiques de C++ vers Java que ce contrôle a été oublié.*

3. ~~Calcule de $u_1 = es^{-1} \bmod n$ et $u_2 = rs^{-1} \bmod n$.~~

> ***5.2 L'inverse modulo de s est calculé par le Théorème de Fermat***
>
> *Si les maths sont votre point fort, vous avez peut-être dis: « Mais comment c'est possible de calculer l'inverse modulo de s quand s = 0 ? C'est impossible… ».*
> *L'implémentation de Java utilise le Petit de Fermat pour calculer l'inverse s modulo n :*
>
> $$s^{n-2} = s^{-1} \ (mod\ n)$$
>
> *Comme il a été oublié de vérifier que s est différent de 0, on trouve* $s^{-1} = s^{n-2} = 0^{n-2} = 0$*, ce qui est faux car c'est un cas qu'il faut exclure lors de l'utilisation de ce théorème.*

4. ~~Calcul du point $P(x_1, y_1) = u_1 \cdot G + u_2 \cdot Q_a$. Si P est le point à l'infini, alors la signature est invalide.~~

5.3 Le point P

Lors de l'étape ***4.*** *Dans la vérification de la signature, on calcule le point*

$$P(x_1, y_1) = u_1 \cdot G + u_2 \cdot Q_a$$

$$P = (es^{-1}) \cdot G + (rs^{-1}) \cdot Q_a$$

Mais comme $s^{-1} = 0$*...*

$$P = 0 \cdot G + 0 \cdot Q_a$$

Tout point sur une courbe elliptique multiplié par 0 donne le point à l'infini, et comme un point O +O =O

$$P = O + O$$

$$P = O$$

Dans cette même étape, il est aussi détaillé dans l'algorithme original que si le point P est égal au point à l'infini, alors la signature est refusée. Malheureusement, Java a oublié cette vérification aussi, ce qui nous mène au problème suivant.

5. ~~Si et seulement si $r \equiv x_1 \bmod n$, alors la signature est valide.~~

5.4 Le point à l'infini

Un problème en informatique est qu'il n'est pas facile d'implémenter le point à l'infini étudié précédemment. Etant donné que ce point est une convention, Java a décidé d'implémenter le point à l'infini par le point (0, 0). Ce n'est pas une mauvaise implémentation en soit car sur de grandes courbes, on est presque sûrs que le point (0, 0) n'en fera pas partie. En revanche, étant donné des oublis de vérification, cette implémentation devient un problème.

Lors du dernier contrôle (étape ***5****), la signature est dite valide si la valeur en abscisse du point P nommée* x_1 *est égale r (mod n). Comme P est le point à l'infini, est que le point à l'infini est le point (0, 0), on a* $x_1 = 0 = r \pmod n$*. La signature est valide !*

Références

[1] RSA: Rivest – Shamir – Adleman. *RSA (cryptosystem)*, wikipedia. 30/06/2022

[2] SHA: *Secure Hash Algorithms*, wikipedia. 30/06/2022

[3] Gaëtan Leurent, Thomas Peyrin, *"SHA-1 is Shambles"*. 30/06/2022

[4] Dr Mike Pound, *What are Digital Signatures? Computerphile* 01/07/2022

[5] Algorithme de René Schoof, wikipedia 01/07/2022

[6] Livre, *Initiation à la cryptographie*, Gilles Dubertret 01/07/2022

[7] Dr.Mike Pound, *Elliptic Curves - Computerphile* 01/07/2022

[8] A. Corbellini, *Elliptic Curve Cryptography: finite fields and discrete logarithms* 30/06/2022

[9] N. Sullivan, *A (Relatively Easy To Understand) Primer on Elliptic Curve Cryptography* 01/07/2022

[10] Mehmet Adalier, Antara Teknik, Efficient and Secure Elliptic Curve Cryptography Implementation of Curve P-256 30/06/2022

[11] KDF: Key Derivation Function, wikipedia 01/07/2022

[12] CVE-2022-21449