Rapport de laboratoire #3

CRY - Chiffrement asymétrique

Alexis Martins

10 juin 2023

Table des matières

1 RSA avec exposant 3

1.1 Décrypter le message

L'attaque est assez simple, il suffit de calculer la racine cubique du message chiffré.

```
1 print("First message :", cubeRoot(c1))
```

Le message obtenu est le suivant : Message de test pour voir si tout fonctionne. Le code est phantom

Cette attaque fonctionne, car le message est plus petit que $\sqrt[3]{n}$ où n est le produit des deux nombres premiers p et q que l'on utilise dans l'algorithme RSA lors du chiffrement.

$$c = m^e \mod n$$

Si le message était plus grand que cette valeur, lors de l'application du chiffrement RSA, le message aurait le modulo appliqué et on ne pourrait pas le retrouver aussi facilement.

1.2 Application du Théorème des Restes Chinois

Ici, le message est beaucoup trop grand pour être déchiffré avec la méthode précédente (plus grand que la valeur citée précédemment). Il faut donc utiliser le théorème des restes chinois.

```
1 x = crt([c2,c3,c4],[pk12.n,pk13.n,pk14.n])
2 print("Second message (CRT) :", cubeRoot(x))
```

Le message obtenu correspond à : This top secret message contains the to secret nuclear bomb code. It should never fall in enemy hands. In particular, be careful to whom you send it. The password is placated.

Lors du chiffrement de ces messages, on retrouve le système suivant :

$$\begin{cases} c_2 = & m^3 & \mod n_2 \\ c_3 = & m^3 & \mod n_3 \\ c_4 = & m^3 & \mod n_4 \end{cases}. \tag{1}$$

On peut donc appliquer le théorème des restes chinois pour trouver la valeur de m^3.

$$\begin{cases} x = & c_2 & \mod n_2 \\ x = & c_3 & \mod n_3 \\ x = & c_4 & \mod n_4 \end{cases}. \tag{2}$$

Ici, la valeur x correspond à m^3. Il suffit donc de calculer la racine cubique de x pour trouver le message. Cela fonctionne, car chaque n_i est copremier avec les autres, sinon il serait possible de calculer leur facteur commun en réalisant leur PGCD (soit p, soit q).
Ensuite, la variable répond à cette égalité $m < n_2, n_3, n_4$, donc $m^3 < n_2 \cdot n_3 \cdot n_4$. On peut ainsi résoudre facilement le système avec le théorème des restes chinois.

Référence de l'attaque généralisée : Håstad's broadcast attack

2 RSA-PSS

2.1 Implémentation de la signature

```python
def PSSverify(pubkey, message, signature):
    pubkey =  RSA.import_key(pubkey)
    h = SHA256.new(message)
    verifier = pss.new(pubkey)
    try:
        verifier.verify(h, signature)
        print("The signature is authentic.")
    except (ValueError, TypeError):
        print("The signature is not authentic.")
```

2.2 Exploitation de la vulnérabilité

Dans un premier temps, j'ai compris que l'objectif de cette attaque était de récupérer la clé privée afin de signer ce que l'on souhaite. J'ai déterminé que je devais donc récupérer soit p, soit q pour pouvoir calculer la clé privée.

En s'aidant de l'indice et des exercices faits en cours, j'ai pensé qu'il me fallait une valeur qui était multiple d'une de ces deux valuers. Cela me permettrait de réaliser le PGCD entre cette valeur et n afin de trouver p ou q. Il ne faut cependant pas qu'il soit multiple des deux, sinon on ne pourrait pas isoler une des deux valeurs.

Pour trouver cette valeur, j'ai imaginé à quoi pouvait ressembler un multiple de p dans le monde $\mathbb{Z}_p \times \mathbb{Z}_q$ (on aurait pu faire la même chose avec un multiple de q). Soit un multiple de p de la forme $k \cdot p$. Celui-ci donne dans $\mathbb{Z}_{pq}$ la valeur $k \cdot p \mod p \cdot q$. Mais si on le passe dans $\mathbb{Z}_p \times \mathbb{Z}_q$, on obtient la valeur $(0 \mod p, k \cdot p \mod q)$.

Maintenant, si on représente les 4 racines du nombre R que l'on possède dans $\mathbb{Z}_p \times \mathbb{Z}_q$, on obtient les valeurs suivantes :

- $R_1 = (a, -b)$
- $R_2 = (a, b)$
- $R_3 = (-a, -b)$
- $R_4 = (-a, b)$

Ce que j'ai remarqué, c'est que l'addition de R_1 et R_4 donne $(0, 0)$ qui est un multiple de p et q. Mais le plus important, c'est que l'addition de R_1 et R_3 donne $(0, 2b)$ qui est un multiple de p uniquement.

J'ai donc commencé à additionner les racines entre elles afin de trouver quelle combinaison était différente de 0. Cela indiquerait que la combinaison de racines différentes de zéro est un multiple de p ou q. Une fois cela trouvé, j'ai pu appliquer le calcul du PGCD entre cette valeur et n pour trouver p (ou q selon le combo).

Lorsque l'on trouve p, on peut trouver q en faisant $q = \frac{n}{p}$. Ensuite, il suffit de calculer la clé privée avec $d = e^{-1} \mod (p-1)(q-1)$. Lorsque l'on possède la clé privée, on peut signer ce que l'on souhaite. Il suffit de construire la clé via la librairie RSA et de signer notre message.

Voici ci-dessous le code que j'ai utilisé pour réaliser cette attaque.

```
public_key =  RSA.import_key(pk2)

# Calcul du multiple de p ou q.
multiple = r2[1] + r2[3]

# Calcul du pgcd entre le multiple et n, ce qui va nous donner p ou q.
gcd = gcd(multiple, public_key.n)

# Calcul de la seconde composante de n. Si on avait p, on a q et inversement
    .
other_component= public_key.n // gcd

# Calcul de phi(n)
phi = (other_component - 1) * (gcd - 1)

# Calcul de d
d = inverse_mod(public_key.e, phi)

# Création de la clé privée
key = RSA.construct((public_key.n, public_key.e, int(d)))

# Message à signer pour tester la clé privée
message_to_sign = "Atterrissage à Yverdon plage, il est 14h et le temps est
    magnifique.".encode('utf-8')

# Signature du message
sign = pss.new(key).sign(SHA256.new(message_to_sign))

# Vérification de la signature
PSSverify(pk2, message_to_sign, sign)
```

3 ECDSA

3.1 Fonction de signature

```sage
def sign(M, a, G, n):
    r = s = 0
    while r == 0 or s == 0:
        k = (Integers(n).random_element(n)).lift()
        R = k*G
        r = R[0].lift() % n
        s = (fastInverse(k, n)*(H(M, n) + a*r)) % n
    return (r, s)
```

3.2 Inverse modulaire rapide

Cela fonctionne, car on peut travailler $\mod n$ et que n est premier. Si on applique le théorème de Fermat-Euler on obtient :

$$k^{n-2} \mod n \equiv k^{(n-2) \mod (n-1)} \mod n \equiv k^{-1} \mod n$$

Le seul avantage que je peux voir, c'est que mathématiquement c'est intéressant et facile à comprendre. Sinon, ce n'est pas plus rapide que la fonction native inverse_mod (comme le montre le test ci-dessous). Après vérification, cela s'explique car derrière les décors, la fonction inverse_mod est dévellopée en C directement.

```
Fast inverse is faster : 0 times with average time   : 1.090427614607901e-05
    seconds
Inverse is faster      : 106 times with average time : 1.947834806622199e-06
    seconds
```

3.3 Exploitation de la vulnérabilité

La vulnérabilité trouvée vient du fait que l'on ne vérifie pas si la signature composée de (r, s) a bien des composantes dans l'intervalle $[1, n-1]$ où n est l'ordre du point G. Ainsi pour n'importe quel message, on a toujours une signature valide qui est (0, 0). Cela est aussi possible à cause de la fonction d'inverse modulaire qui fourni un résultat de 0 si on essaye de calculer l'inverse de 0, alors que celui-ci ne devrait pas exister.

Ces deux problèmes combinés font que lors de la vérification, on a une valeur de r qui est égale à 0 et une valeur calculée avec s qui permet d'obtenir une valeur de 0 pour l'autre côté de l'équation. Voici le calcul avec les valeurs remplacées :

$$0^{-1} \cdot H(m) \cdot g + 0^{-1} \cdot r \cdot A = 0 \mod n$$

```sage
m = "Je dois 10000CHF à Alexandre Duc".encode('utf-8')
print(verify(m, (0,0), A, G, n))
```

Remarque : Le message signé est à titre d'exemple. Malgré le fait qu'une signature garantisse la non-répudiation et qu'elle soit ici valide, je ne dois pas réellement 10000CHF à Alexandre Duc.

3.4 Correction de la vulnérabilité

Pour corriger la vulnérabilité, il faut vérifier que r et s sont bien dans l'intervalle $[1, n-1]$. Je conseillerais aussi d'utiliser la fonction d'inverse modulaire classique ou du moins de retirer la possibilité d'avoir un inverse de 0. Il manquait aussi les vérifications sur la clé publique (mais cela n'est pas directement lié à la vulnérabilité).

Sinon le mieux restera toujours de ne pas implémenter soi-même un système de ce style, mais d'utiliser une librairie qui a déjà été testée et qui est sûre. Cela évitera les erreurs de ce type qui peuvent s'avérer très graves.

3.5 Vulnérabilité similaire sur DSA

Sur DSA, on retrouve donc des élévations à la puissance plutôt que des multiplications. Si on souhaite avoir une signature valide pour tous les messages, il faut que $r = 1$ et $s = 0$. Ainsi on aura les paramètres g et A élevés à la puissance 0 à cause du s, ce qui fera que la partie gauche de l'équation sera égale à 1. Puis, on aura la partie droite de l'équation qui est composée de r qui est égal à 1. La vérification sera ainsi vraie à tous les coups.

4 DSA sur $\mathbb{Z}_p$

4.1 Explication mathématique

Comme pour ECDSA sur les courbes elliptiques, il faut d'abord choisir les paramètres des paramètres globaux au système, ici p et g.

Premièrement, on peut commencer par définir la génération des clés.

- Clé privée : $a \in \{1, \dots, p-1\}$
- Clé publique : $A = g \cdot a \mod p$

Pour la signature, on retrouve les opérations suivantes :

- Tirage du paramètre k aléatoire : $k \in \{1, \dots, p-1\}$
- Calcul de la première partie de la signature (r) : $r = g \cdot k \mod p$
- Hachage du message : $h = H(m)$ où H une fonction de hachage tel que $H : \{0,1\}^* \to \{1, \dots, p-1\}$
- Calcul de la seconde partie de la signature (s) : $s = \frac{h + a \cdot r}{k} \mod p$
- La signature est donc (r, s)

Pour la vérification, on a les opérations suivantes :

- Calcul du hash du message : $h = H(m)$
- Calcul de $u_1 = \frac{h}{s} \mod p$
- Calcul de $u_2 = \frac{r}{s} \mod p$
- Calcul de $x = u_1 \cdot g + u_2 \cdot A \mod p$
- La signature est valide si $r = x$

4.2 Implémentation

```
1  def sign(m, a, p, g):
2
3      r = s = 0
4      F = Integers(p)
5      while r == 0 or s == 0:
6          k = F.random_element()
7          r = F(k * g)
8          h = hash_to_Zp(m, p)
9          s = F((h + a * r)/k)
10     return (r, s)
11
12
13 def verify(m, s, r, A, p, g):
14
15     F = Integers(p)
16     h = hash_to_Zp(m, p)
17     u1 = F(h/s)
18     u2 = F(r/s)
19     u1 = F(u1 * g)
20     u2 = F(u2 * A)
21     x = F(u1 + u2)
22     return r == x
```

4.3 Exploitation de la vulnérabilité

La faille vient de la manière dont les opérations sont transformées dans un groupe additif. On a à notre disposition les paramètres $A = g \cdot a \mod p$ et g. On remarque qu'il est donc très simple d'obtenir la clé privée a à partir de ces deux paramètres.

$$a = \frac{A}{g} \mod p = \frac{g \cdot a}{g} \mod p$$

Voici le code permettant de réaliser cette manipulation :

```
# Récupération de la clé privée
private_key = F(A/g)

# Signature d'un message en utilisant la clé récupérée
m2 = "Ceci est un message authentique, intègre et non-répudiable".encode('utf-8')
(r,s) = sign(m2,private_key,p,g)
print("Attack working :", verify(m2,s,r,A,p,g))
```