Laboratoire 3

Cryptographie

ANNEN Rayane

HEIG VD

11.06.2023

Table des matières

RSA avec exposant $e = 3$

Dans un premier temps, le message m1 est chiffré avec textbook RSA et exposant public 3. Ce message fait moins de 600 bits. Vous trouverez dans votre fichier de paramètres la clef publique pk11 ainsi que le texte chiffré c1. Decryptez ce message. Expliquez dans votre rapport pourquoi cela fonctionne.

Pour récupérer m on peut calculer $\sqrt[3]{c}$, c est initialement obtenu de la manière suivante : $c = m^e \mod \varphi(n)$, Comme m est relativement petit, précisément $m < \varphi(n)$, il est très facile de retrouver m, en effet il n'est pas réduit modulo $\varphi(n)$ on peut donc simplement prendre la racine e-ième dans les réels de m.

Le code pour obtenir le message est le suivant :

```python
e = 3
m = pow(c1, 1/e)
print(f"Ex1.1 message: {long_to_bytes(int(m))}")
```

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

Nous allons maintenant complexifier l'attaque précédente. Cette fois, le texte clair est beaucoup plus grand. Il fait plus de 1300 bits. Par contre, le même message a été envoyé à trois destinataires différents. Ces destinataires ont les clefs publiques pk12, pk13 et pk14 et les textes chiffrés obtenus sont c2, c3, c4. Decryptez ce message et expliquez dans votre rapport pourquoi cela fonctionne.

Nous avons à notre disposition trois messages chiffrés c_1, c_2 et c_3 qui correspondent à un seul même texte m. L'exposant est toujours le même mais les clefs sont différentes, on peut donc écrire le système de congruence suivant :

$$\begin{cases} x & \equiv c_1 \pmod{n_1} \\ x & \equiv c_2 \pmod{n_2} \\ x & \equiv c_3 \pmod{n_3} \end{cases}$$

Pour que le système soit valide, on doit supposer que les n_i sont premiers entre eux, autrement l'attaque est terminée car on peut extraire un facteur d'un des n_i en calculant leur pgcd et ainsi retrouver la clef privée.

En utilisant le théorème des restes chinois, on peut retrouver $x \in \mathbb{Z}_{n_1 n_2 n_3}$, cependant pour que l'attaque fonctionne comme dans le premier exercice il faut que $x < n_1 n_2 n_3$, ainsi on pourra calculer sur les entiers la racine e-ième de x pour retrouver m comme dans l'exercice précédent.

Ainsi on trouve le message suivant :

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 standard

Code de l'attaque :

```
x = CRT([c2, c3, c4], [pk12.n, pk13.n, pk14.n])
print(f"Ex1.2 message: {long_to_bytes(int(pow(x, 1/e)))}")
```

RSA-PSS

Implémentez (en utilisant les fonctions déjà existantes dans pycryptodome) la vérification de signature RSA-PSS et testez cette vérification sur la signature fournie.

Code de l'implémentation :

```
def verify(message: bytes, signature: bytes, public_key: RsaKey):
    verifier = pss.new(public_key)
    h = SHA256.new(message)
    try:
        verifier.verify(h, signature)
        print("The signature is authentic")
    except (ValueError, TypeError):
        print("The signature is not authentic")
```

Forgez la signature d'un message vous autorisant à atterrir partout (le contenu exact du message importe peu). Expliquez votre attaque dans votre rapport.

On connait 4 racines qu'on appelle r_1, r_2, r_3, r_4. On utilise le théorème des restes chinois et particulièrement l'isomorphisme d'anneau $\mathbb{Z}_{pq} \cong \mathbb{Z}_p \times \mathbb{Z}_q$.

Le but est de retrouver un nombre qui est un multiple de p mais pas un multiple de q (indice donné) dans $\mathbb{Z}_{pq}$ ce nombre est de la forme kp avec $k = 1, \ldots, q-1$. On va chercher à se représenter ce multiple dans $\mathbb{Z}_p \times \mathbb{Z}_q$, On sait qu'un nombre a représenté dans $\mathbb{Z}_{pq}$ sera représenté sous forme de couple tel que

$$(a \mod p, a \mod q) \in \mathbb{Z}_p \times \mathbb{Z}_q$$

En cherchant à trouver kp dans $\mathbb{Z}_p \times \mathbb{Z}_q$ on a que :

$$(kp \mod p, kp \mod q) = (0, kp \mod q) \in \mathbb{Z}_p \times \mathbb{Z}_q$$

Il faut donc chercher un nombre dans $\mathbb{Z}_{pq}$ tel qu'on ait le couple $(0, kp \mod q)$ dans $\mathbb{Z}_p \times \mathbb{Z}_q$. On peut alors se représenter les racines qui ont été extraites dans le monde $\mathbb{Z}_p \times \mathbb{Z}_q$, on sait également que les racines viennent par paires opposées :

$$\begin{cases} r_1 &= (a \mod p, b \mod q) \\ r_2 &= (-a \mod p, b \mod q) \\ r_3 &= (a \mod p, -b \mod q) \\ r_4 &= (-a \mod p, -b \mod q) \end{cases}$$

Ainsi dans ce cas : $r_1 + r_4 = 0, r_2 + r_3 = 0$, ce qu'on cherche c'est un nombre tel que $a \mod p = 0$, on trouve cela en calculant par exemple $r_1 + r_2 = (0, 2b \mod q)$, cela fonctionne car par le théorème des restes chinois : $\mathbb{Z}_{pq} \cong \mathbb{Z}_p \times \mathbb{Z}_q$, ainsi les propriétés qui nous intéressent (à savoir avoir un multiple de p mais pas de q) restent vraies dans les deux anneaux avec cette addition.

Maintenant que nous avons un multiple de p mais pas de q l'attaque est presque terminée, on peut retrouver le nombre premier p en calculant le pgcd entre le multiple de p mais pas de q et n : $p = \gcd(kp, n)$, connaissant p on retrouve q et nous pouvons calculer d :

$$\begin{cases} p, n & \text{connus} \\ q = \frac{n}{p} \\ d = e^{-1} \mod \varphi(pq) = e^{-1} \mod (p-1)(q-1) \end{cases}$$

Nous avons retrouvé la clef privée (n, d). On peut ainsi créer une signature valide comme on le souhaite.

Voici l'attaque avec python :

```python
Kp = Fn(r2[0] + r2[2]) # r1 + r3 = kp
n = pk2.n
p = gcd(Kp, n).lift()
q = n / p
print(p * q  == n) # true
d = inverse_mod(pk2.e, (p-1) * (q-1))
k = construct((n, pk2.e, int(d))) # rsa private key: N, e, d

m = b"You are allowed to land at Zurich airport on 02.02.2022 at 13h55"
h = SHA256.new(m)
sig = pss.new(k).sign(h)
verify(m, sig, pk2) # The signature is authentic
```

ECDSA

Il manque la fonction permettant de signer. Codez la !

```sage
def sign(M: bytes, a: int):
    (G, E, n) = params()
    F = Integers(n)
    r = 0
    s = 0
    while r == 0 or s == 0:
        k = ZZ.random_element(n)
        Q = k * G
        x_1 = Q[0]
        r = F(x_1)
        k_inv = fastInverse(k, n)
        s = F((H(M, n) + a * r) * k_inv)
    return (r, s)
```

L'inverse modulaire est calculé d'une manière différente que celle vue en cours. Pourquoi cela fonctionne ? Quel est l'avantage de cette méthode ?

Soit k et n deux entiers, on calcule $k^{-1} \mod n$ de la manière suivante : $k^{-1} \equiv k^{n-2} \pmod{n}$.

Par le théorème d'Euler si a est premier avec n, on a que $a^{\varphi(n)} \equiv 1 \pmod{n}$. En multipliant les deux côtés par a^{-1} on trouve que $a^{-1} \equiv a^{\varphi(n)-1} \pmod{n}$, étant donné que n est premier dans notre cas, on a que $\varphi(n) = n - 1$, ainsi on trouve la relation :

$$a^{-1} \equiv a^{n-2} \pmod{n}$$

Malgrés ce que laisse entendre le nom de la fonction, cette méthode n'est pas forcément plus rapide, en effet, même si théoriquement cette méthode devrait être plus rapide (bien qu'asymptotiquement les deux méthodes sont équivalentes) ce n'est pas le cas ici, la fonction `inverse_mod` étant implémentée en C (via la CFFI on utilise en fait la fonction `mpz_invert` [1] de la librairie GMP, qui elle utilise l'algorithme d'Euclide étendue avec `mpz_gcdext`) et de manière optimisée elle est plus rapide.

Dans un certains sens elle donne un avantage en tant qu'attaquant car elle nous simplifie l'attaque sur ECDSA car quand on cherche l'inverse de 0 modulo n, elle ne fait pas de vérifications et on tombe sur 0 au lieu d'avoir une erreur comme avec `inverse_mod` (division par zéro).

Cette fonction permettant de vérifier les signatures souffre d'une vulnérabilité très grave (vue récemment dans un cas réel). Exploitez la pour obtenir une signature valide du message "Je dois 10000 CHF à Alexandre Duc". Cette signature doit être valide pour la clef publique A donnée dans vos paramètres. Vous y trouverez aussi une signature sig du message "Hello World !"

[1] Code source: https://github.com/alisw/GMP/blob/2bbd52703e5af82509773264bfbd20ff8464804f/mpz/invert.c

On voit rapidement qu'il n'y pas de vérifications que r ou s sont dans l'intervalle $1, \dots, n-1$.

On peut simplement envoyer comme couple $(r, s) = (0, 0)$, ainsi on peut vérifier et valider n'importe quel message.

Ci-dessous le code de l'attaque :

```python
# Attack with (0, 0)
(G, e, n) = params()
A = e(10339..., 11799...)
m = "Je dois 10000 CHF à Alexandre Duc".encode('utf8')
pair = (0, 0)
print(verify(m, pair, A, G, n))  # returns true
```

L'attaque fonctionne car si $s = 0$ et $r = 0$ on a que $s^{-1} = 0^{n-2} \mod n = 0$ et donc

$$s^{-1} \cdot H(m) \cdot G + s^{-1} \cdot r \cdot A = 0 \cdot H(m) \cdot G + 0 \cdot r \cdot A = r = 0$$

On peut donc passer n'importe quelle message et le calcul ci-dessus sera toujours vrai, donc le message sera toujours valide.

Comment faudrait-il corriger cette fonction de vérification de signatures afin d'empêcher l'attaque précédente ?

On vérifie que r et s sont dans l'intervalle $1, \dots, n-1$: on peut le faire en ajoutant la condition au début de la fonction `verify` :

```python
if not (1 <= r <= n - 1) or not (1 <= s <= n - 1):
        return False
```

Il faudrait également vérifier que la clef publique $A \neq \mathcal{O}$ et que A est bien sur la courbe elliptique telle que $nA = \mathcal{O}$

Idéalement on devrait utiliser les fonctions permettant de signer un message venant directement d'une librairie telle que pycryptodome au lieu d'essayer de réimplementer ces algorithmes nous même, il est trop facile de faire des erreurs.

Imaginez que la même vulnérabilité existe dans une fonction de vérification de signature DSA. A quoi ressemblerait votre attaque dans ce cas ?

Il faut envoyer le couple $(r, s) = (1, 0)$, similairement à avant : $s^{-1} = 0$ (fastInverse) et quelque soit le message m à vérifier on aura l'identité :

$$\left(g^{H(m)s^{-1}} A^{rs^{-1}} \mod p\right) \mod q = \left(g^{H(m)\cdot 0} A^{r\cdot 0}\right) = \left(g^0 A^0\right) = r = 1$$

DSA sur $\mathbb{Z}_p$

Expliquez mathématiquement comment fonctionne DSA sur $\mathbb{Z}_p$. Plus précisément, décrivez la signature et la vérification.

On se rapproche de quelque chose qui ressemble aux algorithmes sur les courbes elliptiques, évidemment sans toutes les propriétés qui font que DSA est viable sur celles-ci. Sur $\mathbb{Z}_p$ les opérations d'exponentiations sont remplacées par des multiplications et les opérations de multiplications sont remplacées par des additions.

On fixe les paramètres publics g et p avec g qui a ordre p.

Les clefs sont générées de la manière suivante :

- On tire une valeur secrète $a \in \{1, \dots, p-1\}$, ce sera notre clef privée.
- La clef publique est obtenue de la manière suivante : $A = a \cdot g \mod p$

Concernant la signature :

- Toutes les opérations se font modulo p
- On utilise une fonction de hachage cryptographiquement sûre : $H : \{0,1\}^* \to \{1, \dots, p-1\}$

1. On initialise s et r à 0.
2. On génère un élement uniformément aléatoire $k \in \{1, \dots, p-1\}$
3. On calcule $r = k \cdot g$
4. On calcule $s = \frac{H(m)+ar}{k}$
5. On retourne à l'étape 2 si r ou s sont égales à 0.

La signature du message m est le couple (r, s)

Concernant la vérification :

- Toutes les opérations se font modulo p.
- On vérifie la signature (r, s) attachée au message m
- On prend en entrée le message m, r, s et la clef publique A.

1. On calcule $u_1 = \frac{H(m)}{s} \cdot g$
2. On calcule $u_2 = \frac{r}{s} \cdot A$
3. On additionne u_1 et u_2 :
$$u_1 + u_2 = \frac{H(m)}{s} \cdot g + \frac{r}{s} \cdot A$$
4. On vérifie que $u_1 + u_2 = r$, si oui, la signature est authentique et elle ne l'est pas sinon.

La vérification fonctionne car :

$$\begin{aligned} u_1 + u_2 = \frac{H(m)}{s} \cdot g + \frac{r}{s} \cdot A &= \frac{H(m)}{\frac{H(m)+ar}{k}} \cdot g + \frac{r}{\frac{H(m)+ar}{k}} \cdot ag \\ &= H(m)\frac{k}{H(m)+ar} \cdot g + r\frac{k}{H(m)+ar} \cdot ag \\ &= g\frac{k}{\cancel{H(m)+ar}}\left(\cancel{H(m)+ar}\right) = gk = r \end{aligned}$$

Implémentez cette construction (la signature et la vérification). Vous trouverez dans votre fichier de paramètres les valeurs de g et p. A noter que g a ordre p. Vous aurez besoin d'une fonction de hashage mappant un message vers un entier modulo p qui vous est fournie dans le fichier dsa.sage

Signature :

```
def sign(m, a, p, g):
    s = 0
    r = 0
    F = Integers(p)
    while s == 0 or r == 0:
        k = F.random_element()
        r = F(k * g)
        h = hash_to_Zp(m, p)
        s = F((h + a * r) * inverse_mod(k, p))
    return (r, s)
```

Vérification :

```
def verify(m, s, r, A, p, g):
    F = Integers(p)
    h = hash_to_Zp(m, p)
    s_inv = inverse_mod(s, p)
    u1 = F(h * s_inv * g)
    u2 = F(r * s_inv * A)
    return F(u1 + u2) == r
```

Cette construction est très vulnérable. Vous trouverez dans votre fichier de paramètres une clef publique A, un message m et sa signature (r, s). Cassez la construction pour signer un autre message (qui a du sens).

On veut avec notre clef A signer un message m', on sait que $A = ag$, on peut simplement calculer $a = \frac{A}{g}$ pour retrouver la clef privée a et ainsi signer tous les messages que l'on souhaite.

Code de l'attaque :

```
m = "Please send 100000CHF to Rayane Annen".encode('utf8')
```

```
a = Fp(A/g)
(r,s) = sign(m, a, p, g)
print(f"Authentic signature with retrieved private key: {verify(m, s, r, A, p, g)}")
```