Chiffrement RSA

HackademINT

23 novembre 2021

Table des matières

1 Principe du chiffrement

Le chiffrement RSA est un système fonctionnant sur une dualité entre clé publique et clé privée. La clé publique est utilisée pour chiffrer un message, et la clé privée pour le déchiffrer. Comme leur nom l'indique, la clé privée est en accès libre, contrairement à la clé privée qui doit à tout prix rester secrète. Ce type de chiffrement est dit *asymétrique*, en opposition aux types de chiffrement symétriques où il y a un même mot de passe connu par l'émetteur et le destinataire.

1.1 Clé publique

On commence par créer une clé publique. Pour cela, on sélectionne tout d'abord deux nombres premiers quelconques distincts p et q. On pose ensuite n telle que $n = p \times q$. On appelle n le *module de chiffrement*.

On utilise la fonction indicatrice d'Euler φ sur n. Pour rappel, cette fonction nous donne le nombre d'entiers premiers avec n, 1 et n inclus. Ici, p et q étant premiers, on a :

$$\varphi(n) = (p-1) \times (q-1)$$

Enfin, on choisit un nombre e, appelé exposant de chiffrement, qui respecte les deux conditions suivantes :

1. $1 < e < \varphi(n)$
2. e et $\varphi(n)$ sont premiers entre eux.

On a donc créé un couple (n, e) qui constitue ce qu'on appelle une *clé publique*. Notons que e n'est pas unique, on peut donc avoir plusieurs clés publiques.
Cette clé est utilisée pour chiffrer le message.

1.2 Clé privée

Il faut maintenant créer la clé privée.

On créé l'exposant de déchiffrement d. Cet entier est défini comme l'inverse modulaire de e par rapport à $\varphi(n)$, c'est-à-dire tel que

$$d \times e \equiv 1\,[\varphi(n)]$$

Autrement dit, on cherche un couple (d, k) tel que

$$de + k\varphi(n) = 1$$

Ici, seul d nous intéresse. Un tel couple peut se calculer facilement en utilisant l'algorithme d'Euclide étendu.
Le couple (n, d) est appelé *clé privée*.

On remarque que l'exposant de chiffrement e et l'exposant de déchiffrement d sont en fait interchangeables, c'est à dire qu'un fichier chiffré avec e peut être déchiffré avec d, et un fichier chiffré avec d peut être déchiffré avec e.

1.3 Chiffrement et déchiffrement

Ces procédés sont assez simples. On découpe le fichier à coder en morceaux auxquels on associe une valeur entière m strictement inférieure à n. Par exemple pour chiffrer un fichier texte, on pourrait associer à chaque lettre sa valeur ascii pour n assez grand et concaténer les valeurs obtenues en un nombre m. Pour le chiffrement, on défini la variable c ainsi :

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

Le déchiffrement de manière similaire :

$$m = c^d \bmod n$$

La valeur c est donc le message chiffré ! Notons que les nombres manipulés sont ici extrêmement grands, il peut être nécessaire d'utiliser des techniques d'exponentiation modulaire pour optimiser les calculs.

Il est nécessaire que m soit inférieur à n !!!

Dans la pratique, il est très courant de choisir des exposants de chiffrement e ayant pour valeur 3, 5, 17, 257 ou 65537. En effet, ces nombre sont des *Nombres de Fermat*, c'est à dire qu'ils s'écrivent sous la forme $2^{2^k} + 1$. Ce choix s'explique par des soucis d'optimisation, le chiffrement s'effectuant alors par ce calcul : $m^e = m \times (...((m\underbrace{^2)^2)...)^2}_{\text{k fois}}$.

2 Implémentation

La théorie c'est bien beau, mais dans la pratique, comment on fait ? Il n'existe pas un unique format pour le RSA. Les différences d'implémentation reposent sur des tailles de clé différentes et suivent une certaine convention de nommage, RSA-m, avec m un entier : de RSA-100 à RSA-500, le nombre correspond au nombre de chiffres dans le module de chiffrement n. La convention a ensuite changé à partir de RSA-576, où le nombre correspond désormais au nombre de bits dans le module de chiffrement à l'exception de RSA-617 qui fut créé avant le changement.

Le binaire du fichier est convertit en un nombre entier. En fait, on se contente de concaténer les octets du fichier en un seul nombre, auquel on applique le chiffrement décrit ci-dessus. Ensuite, le résultat du chiffrement est convertit en base64 est est écrit dans le fichier. Lorsque vous partagez un fichier chiffré avec OpenSSL, c'est en fait la base64 de ce fichier que vous envoyez.

2.1 Format d'une clé

Le chiffrement RSA est implémenté par de nombreux programmes, on peut notamment citer OpenSSL et OpenSSH pour les plus utilisés. Il existe

des différences entre chaque implémentation, mais le principe général reste le même.

Voici une clé publique générée par OpenSSL :

```
-----BEGIN PUBLIC KEY-----
MGQwDQYJKoZIhvcNAQEBBQADUwAwUAJJAMLLsk/b+SO2Emjj8Ro4lt5FdLO6WHMM
vWUpOIZOIiPu63BKF8/QjRa0aJGmFHR1mTnG5Jqv5/JZVUjHTB1/uNJM0VyyO0zQ
owIDAQAB
-----END PUBLIC KEY-----
```

Et voici la clé privée correspondante :

```
-----BEGIN PRIVATE KEY-----
MIIBeQIBADANBgkqhkiG9w0BAQEFAASCAWMwggFfAgEAAkkAwsuyT9v5I7YSaOPx
GjiW3kV0s7pYcwy9ZSk4hk4iI+7rcEoXz9CNFrRokaYUdHWZOcbkmq/n8llVSMdM
HX+40kzRXLI7TNCjAgMBAAECSDIDDkLGnU533n4jl7E9ui5S8sV6IF9Zc/7W+HYy
9Tz4iGYJ/2P6EU4rXfHbYkn46vC/X6Jq1bjki3+rBQ0y/FdLRG4i0IunsQIlAMzp
VFfxJ8SVRuV4QQKaxq9wYExk6CgQGKy7U4Izu1fzem5olQIlAPNcuCXwAzlvEfFJ
pZoP5z20MWLLMFuOv+2lyoQL4kxTa2quVwIlAKMa3z4ZAPSjdglVzTmLhbdEFgL5
Fyo8lcd4XKJ5FxiM/9k+eQIlANVRQsFOXyUq7MlZyuT+Z624k0mL3bmDqUzAdIHG
gPhybse2RQIkWpZM7MIg4PDt6jKlwliObY7+iESxKMReyDoGHyZjcu6cFqqI
-----END PRIVATE KEY-----
```

On voit ici que les clés commence avec le texte "BEGIN — KEY" et finissent avec "END — KEY". Ces chaînes de caractères sont spécifiques à OpenSSL. Selon l'implémentation, ces chaînes vont varier. Par exemple, avec OpenSSH, elles sont de la forme "BEGIN OPENSSH PUBLIC KEY". Voir "PKCS" pour plus d'informations. OpenSSl utilise le format PKCS1.

Il existe plusieurs moyens de récupérer les information de ce fichier. Pour la clé privée :

```
$ openssl rsa -in privatekeyfile -text -noout >
decoded_private_key
```

Sinon, on peut décoder le base64 que l'on écrit dans le fichier *base64_décodé*, et interpréter le résultat en quelque chose de lisible :

```
$ cat decoded_base64 | openssl asn1parse -inform DER
```

On obtient donc un résultat de cette forme pour la clé privée :

```
RSA Private-Key: (576 bit, 2 primes)
modulus:
    00:c2:cb:b2:4f:db:f9:23:b6:12:68:e3:f1:1a:38:
    96:de:45:74:b3:ba:58:73:0c:bd:65:29:38:86:4e:
    22:23:ee:eb:70:4a:17:cf:d0:8d:16:b4:68:91:a6:
```

```
    14:74:75:99:39:c6:e4:9a:af:e7:f2:59:55:48:c7:
    4c:1d:7f:b8:d2:4c:d1:5c:b2:3b:4c:d0:a3
publicExponent: 65537 (0x10001)
privateExponent:
    32:03:0e:42:c6:9d:4e:77:de:7e:23:97:b1:3d:ba:
    2e:52:f2:c5:7a:20:5f:59:73:fe:d6:f8:76:32:f5:
    3c:f8:88:66:09:ff:63:fa:11:4e:2b:5d:f1:db:62:
    49:f8:ea:f0:bf:5f:a2:6a:d5:b8:e4:8b:7f:ab:05:
    0d:32:fc:57:4b:44:6e:22:d0:8b:a7:b1
prime1:
    00:cc:e9:54:57:f1:27:c4:95:46:e5:78:41:02:9a:
    c6:af:70:60:4c:64:e8:28:10:18:ac:bb:53:82:33:
    bb:57:f3:7a:6e:68:95
prime2:
    00:f3:5c:b8:25:f0:03:39:6f:11:f1:49:a5:9a:0f:
    e7:3d:b4:31:62:cb:30:5b:8e:bf:ed:a5:ca:84:0b:
    e2:4c:53:6b:6a:ae:57
exponent1:
    00:a3:1a:df:3e:19:00:f4:a3:76:09:55:cd:39:8b:
    85:b7:44:16:02:f9:17:2a:3c:95:c7:78:5c:a2:79:
    17:18:8c:ff:d9:3e:79
exponent2:
    00:d5:51:42:c1:4e:5f:25:2a:ec:c9:59:ca:e4:fe:
    67:ad:b8:93:49:8b:dd:b9:83:a9:4c:c0:74:81:c6:
    80:f8:72:6e:c7:b6:45
coefficient:
    5a:96:4c:ec:c2:20:e0:f0:ed:ea:32:a5:c2:58:8e:
    6d:8e:fe:88:44:b1:28:c4:5e:c8:3a:06:1f:26:63:
    72:ee:9c:16:aa:88
```

On a donc le module n (modulus), l'exposant privé d (publicExponent) qui vaut ici 65537 (0x10001 est sa valeur hexadécimale), p et q (prime1 et prime2). Les autre valeurs servent juste à optimiser la vitesse de déchiffrement lors de l'utilisation du théorème des restes chinois.

2.2 Format binaire d'une clé

Attention : les détails ci-dessous ne concernent que le format de stockage ASN.1, que OpenSSL utilise. Cela varie selon l'implémentation, par exemple OpenSSH ne l'utilise pas.

Les base64 contiennent des données binaires, encodées sous le format DER (Distinguished Encoding Rules). Pour faire simple, les données sont identifiées par leur type, par exemple `0x02h` pour un entier, suivi de leur de

leur taille en octet, puis des données. Par exemple, pour l'entier

$$16442014250840310102418670872522 2$$

soit

$$081B46223237580BD439A06B0DE6$$

en hexadécimal qui occupe 14 octets (0x0Eh), la séquence sera

$$020E081B46223237580BD439A06B0DE6$$

En réalité, la taille est définie de manière un peu plus complexe, elle peut en effet utiliser plusieurs octets. Pour que ce soit le cas, il faut que le bit le plus significatif de la taille soit à 1. Quand c'est le cas, le premier octet de la taille donne *la taille de la taille*. Par exemple, si vous voulez définir un entier d'une taille de 1234 octets, vous devez définir la taille sur deux octets. 1234 vaut 0100 1101 0010 en binaire un seul octet n'est pas suffisant. La taille sera donc encodée de la manière suivante sur *trois octets* en binaire :

$$10000010 \quad 00000100 \quad 11010010$$

Donc de cette manière en hexadécimal :

$$8204d2$$

On procède également de cette manière si la taille ne nécessite qu'un octet mais que son bit le plus significatif vaut 1. Dans ce cas, le premier octet sera 10000001.

Vous pouvez visualiser interactivement vos fichier sur ce site https://lapo.it/asn1js, et trouver la spécification complète ici : https://ldap.com/ldapv3-wire-protocol-reference-asn1-ber/

2.3 Signature

Le protocole RSA basique permet de créer des messages que seul le destinataire pourra lire, mais par nature ne permet pas de vérifier la légitimité d'un message, étant donné que tout le monde utilise la même clé publique pour un destinataire donnée. Alors, s'assurer que c'est l'expéditeur est bien celui qu'il prétend ? C'est là l'intérêt de la signature des messages.

Le principe est simple : l'expéditeur créé le message puis calcule le hash du fichier. Ensuite, le hash est chiffré avec une clé privée de l'expéditeur, le résultat est ajouté au message, et enfin le message est chiffré avec la clé publique du destinataire. Le destinataire pourra vérifier la légitimité du message avec la clé publique de l'expéditeur.t Il est tout à fait possible d'utiliser des clés différentes pour gérer le chiffrement des messages et leur signature.

L'implémentation d'une signature peut varier selon les logiciels, mais les concepts restent similaires.

2.4 Protection par mot de passe

La sécurité du RSA reposant intégralement sur le secret de la clé privée, il est important de la protéger de tout accès extérieur. C'est pourquoi certaines implémentations proposent de protéger la clé privée par un mot de passe, utilisant un algorithme symétrique tel que AES. Les implémentations peuvent varier. Prenons l'exemple du format PKCS#8 (voir PKCS en annexe).

Le format PKCS#8 permet de chiffrer la clé privée, avec un algorithme de votre choix (AES par exemple). Qand la clé est chiffrée, le format ASN.1 est alors :

```
PrivateKeyInfo ::= SEQUENCE {
  version         Version,
  algorithm       AlgorithmIdentifier,
  PrivateKey      OCTET STRING
}

AlgorithmIdentifier ::= SEQUENCE {
  algorithm       OBJECT IDENTIFIER,
  parameters      ANY DEFINED BY algorithm OPTIONAL
}
```

3 Conclusion

Le chiffrement RSA est un outil à la fois utile et utilisé, mais présente un lourd défaut, la puissance de calcul demandée. Si cela ne pose pas de problèmes pour de petits messages, chiffrer des fichiers lourds n'est pas envisageable. À l'inverse, le chiffrement symétrique AES n'a pas ce défaut, mais la symétrie de la clé est un problème. C'est pourquoi on associe les forces de ces deux algorithmes : le RSA chiffre la clé utilisée par l'AES, et l'AES chiffre les données. De cette manière, on peut assurer la sécurité des données avec une puissance de calcul raisonnable. C'est sous un principe similaire que fonctionne le protocole TLS, au cœur du protocole https utilisé le réseau internet.

Le chiffrement RSA bien utilisé est extrêmement solide. Casser une clé de 2048 bits nécessiterait des *milliards* d'années avec les moyens actuels. Cependant, l'informatique quantique pourrait changer la donne, puisque ce type d'informatique est très puissant dans la factorisation en nombres premiers, et des ordinateurs quantiques de 20 millions de bits quantiques pourraient briser un chiffrement RSA de 2048 bits en quelques heures. À l'heure actuelle (2020), les ordinateurs quantiques ont une capacité de l'ordre de 100 qubits, mais leur évolution est à surveiller de près.

La solidité du RSA n'est cependant pas absolue. Le principe est simple, mais il est aisé de faire de lourdes erreurs sans même s'en rendre compte. Ainsi, utiliser un exposant de chiffrement d trop faible, partager un même module de chiffrement n entre plusieurs utilisateurs, ne pas choisir judicieusement ses entiers, ou tout simplement utiliser un module n factorisé publiquement compromet lourdement la sécurité du RSA. Les erreurs d'implémentations sont légions, sachez les repérer.

A Annexe

A.1 Vocabulaire

— PEM (Privacy-Enhanced Mail) : Il s'agit d'un format de fichier servant à stocker les clés. Ces fichiers commence par un en-tête d'une ligne commençant par "—–BEGIN ", suivit ou non d'une étiquette et finissant par "—–", et d'un bas de page similaire d'une ligne commençant par "—–END ", suivit d'une étiquette et finissant par "—–". Entre ces deux lignes, il y a des données encodées en base64.

— ASN.1 : Il s'agit d'un système de notation standard servant à décrire des structures de données. Ce système ressemble à ceci :

```
  0:d=0  hl=4 l= 377 cons: SEQUENCE
  4:d=1  hl=2 l=   1 prim:  INTEGER           :00
  7:d=1  hl=2 l=  13 cons:  SEQUENCE
  9:d=2  hl=2 l=   9 prim:   OBJECT
 :rsaEncryption
20:d=2  hl=2 l=   0 prim:   NULL
22:d=1  hl=4 l= 355 prim:  OCTET STRING      [HEX D
UMP]:3082015F020100024900C2CBB24FDBF923B61268E3F11A
3896DE4574B3BA58730CBD652938864E2223EEEB704A17CFD08
D16B46891A61474759939C6E49AAFE7F2595548C74C1D7FB8D2
4CD15CB23B4CD0A3020301000102483203 0E42C69D4E77DE7E2
397B13DBA2E52F2C57A205F5973FED6F87632F53CF8886609FF
63FA114E2B5DF1DB6249F8EAF0BF5FA26AD5B8E48B7FAB050D3
2FC574B446E22D08BA7B1022500CCE95457F127C49546E57841
029AC6AF70604C64E8281018ACBB538233BB57F37A6E6895022
500F35CB825F003396F11F149A59A0FE73DB43162CB305B8EBF
EDA5CA840BE24C536B6AAE57022500A31ADF3E1900F4A376095
5CD398B85B7441602F9172A3C95C7785CA27917188CFFD93E79
022500D55142C14E5F252AECC959CAE4FE67ADB893498BDDB98
3A94CC07481C680F8726EC7B64502245A964CECC220E0F0EDEA
32A5C2588E6D8EFE8844B128C45EC83A061F266372EE9C16AA88
```

— DER (Distinguished Encoding Rules) : C'est un standard d'encodage pour les données sous format ASN.1.

A.2 Formats PKCS

Les formats PKCS (Public-Key Cryptography Standards), ou *standards de cryptographie à clé publique* sont un ensemble de spécifications de formats cryptographiques conçues par les laboratoires RSA en Californie. Ces formats sont numérotés de 1 à 15, on peut notamment en retenir deux concernant le RSA, les formats PKCS#1 et PKCS#8. Le PKCS#1 est le standard pour le RSA, il permet le stockage simple des clés publiques et privées, il se présente ainsi :

```
-----BEGIN PUBLIC KEY-----
MIIBIDANBgkqhkiG9w0BAQEFAAOCAQ0AMIIBCAKCAQECB6ffnRc/WWmtFtwxhJaz
a+Of5YEgfm6jGNO/viLItIVgC6mBGnjezG1aq3mhwsSR621POYIGV7ZoY5G4VHQX
KuUE9I8C9+46KrMfzhz5wi9A6RmWXH9nqKy/oR7k5+LzIXvJoFRYdQBCTQgGwOdZ
CBZR9uQGqaZC3m6OExy2RKEuRlc72CRtxeBn0qTxdv727sRFv6nbiIo1JXN25nEJ
+qvjmwz4r+LKEj2oMU0J8kBJIvxBFtaCpL2u7Lc/WcSdt/oSp/xcmBRUklyU4LVH
LgLZJNrWLCYAZuB8fTsQidVHXCwGa3+UVTx16FbjoqdzxsJNW6ZAVeuP6j5XsGsE
owIBfw==
-----END PUBLIC KEY-----
```

Les clés privées sont similaires, mais elles sont délimitées par :

```
-----BEGIN RSA PRIVATE KEY-----
Insérer base 64...
-----END RSA PRIVATE KEY-----
```

Le format PKCS#8 quant à lui sert uniquement aux clés privées. L'avantage de ce format est qu'il permet de chiffrer la clé privée à l'aide d'un mot de passe, au moyen de l'algorithme de votre choix. Il est délimité par

```
-----BEGIN PUBLIC KEY-----
Insérer base 64...
-----END PUBLIC KEY-----
```

Si la clé privée est chiffré, alors le format est :

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
Insérer base 64...
-----END ENCRYPTED PRIVATE KEY-----
```

A.3 Outils utiles

— FactorDB : base de donnée de factorisation en nombres premiers, vous pouvez trouver la factorisation de nombre très larges. Cette base n'est évidemment pas exhaustive, mais peut être très utile. http://factordb.com/

— Récupérer la clé publique et le module du fichier *public_key.pem*

```
$ openssl rsa -pubin -in public_key.pem -modulus -text
```

— Récupérer les informations du fichier *private_key.pem*

```
$ openssl rsa -in public_key.pem -modulus -text
```

De manière générale, vous pouvez interchanger les commandes d'OpenSSL pour les clés publiques/privées en ajoutant ou non l'option `-pubin`

— Alternative pour récupérer la clé publique du fichier *public_key.pem*

```
$ PUBKEY=`grep -v -- ----- public_key.pem | tr -d '\n'`
$ echo $PUBKEY | base64 -d | openssl asn1parse -inform
DER -i
```

— Commande OpenSSL pour déchiffrer un fichier en utilisant la clé privée :

```
$ openssl rsautl -decrypt -in /path/to/your/encrypted
-out /path/where/you/want/your/decrypted.txt -inkey
/path/to/your/private_key.pem
```

— Commande OpenSSL pour chiffrer un fichier en utilisant la clé publique :

```
$ openssl rsautl -encrypt -in /path/to/your/file -out
/path/to/your/encrypted -pubin -inkey
/path/to/your/public_key.pem
```

— Python possède nombreuses bibliothèques extrêmement utiles concernant le RSA. On peut notamment citer la bibliothèque *Crypto* qui dispose de fonctions assez pratiques. Par exemple, vous pouvez créer une clé publique ou privée à partir du module et des exposants :

```
from Crypto.PublicKey import RSA
```

```
# Clé privée :
with open('privatekeyfile','wb') as pub:
    pub.write(RSA.construct((n, e, d, p,
    q)).exportKey(pkcs=1))

# Clé publique :
with open('publickeyfile','wb') as priv:
    priv.write(RSA.construct((n, e)).exportKey(pkcs=1))
```

Elle est également utile pour convertir un fichier en nombre et appliquer directement les opérations arithmétiques nécessaires au chiffrement ou au déchiffrement :

```
from Crypto.Util.number import long_to_bytes,
bytes_to_long
with open("fichier_chiffré", 'r') as file:
    code = bytes_to_long(file.read())
    ...
    print("Message caché : ", long_to_bytes(message))
```