Niveau : 2 **Orange**

Chiffrer avec RSA

1 Introduction

Tout d'abord disons encore une fois qu'à cause de la lenteur des systèmes à clé publique, il n'est pas envisageable de chiffrer de grosses masses de données avec ces systèmes, surtout si ceci doit être fait en temps réel (penser à une conversation téléphonique par exemple). Le chiffrement RSA sert donc à chiffrer des messages relativement courts, et même parfois très courts (des clés secrètes de systèmes à clé secrète par exemple).

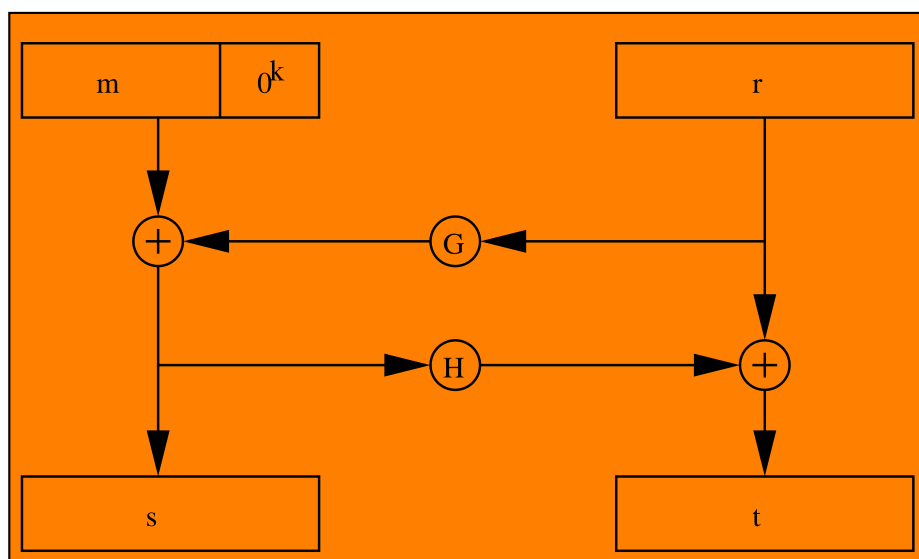
Pour chiffrer proprement avec RSA, il ne suffit pas de prendre le message et de lui appliquer la fonction de chiffrement RSA. D'une part, ceci conduirait à un chiffrement déterministe, c'est-à-dire que si on chiffre deux fois le même message on obtient deux fois le même chiffré, ce qui peut avoir des inconvénients, d'autre part, dans le cas de messages très courts on risque des attaques spécifiques, puisqu'on ne bénéficie plus de la totale diversité fournie par la taille de l'espace des messages proposée en théorie par RSA. Il convient donc de préparer le message à chiffrer par une phase dite "phase de padding (bourrage en français)".

En 1994, M. Bellare et P. Rogaway ont introduit le système OAEP (Optimal Asymetric Encryption Padding) qui permet cette phase de padding.

Nous présenterons ici le système RSAES-OAEP (RSA Encryption Scheme) basé sur RSA et OAEP, qui permet de chiffrer avec RSA.

2 Rappel sur OAEP

OAEP est un moyen de préparer le message à chiffrer ("padding" du message) qui n'est pas spécifique à l'encapsulation d'une clé, mais qui peut être employé pour un chiffrement asymétrique général.



Le message m est rallongé en un message M (sur le dessin on a rajouté des 0, mais ça peut être autre chose). Un nombre aléatoire r est tiré au sort. On dispose par ailleurs d'une fonction publique de hachage H et d'un générateur de masque (ou KDF) public G. La fonction H peut bien entendu être remplacée sans inconvénient par un générateur de masque. On calcule alors :

$$s = G(r) \oplus M,$$
$$t = H(s) \oplus r.$$

Le message encodé est alors la concaténation $x = s||t$. Le chiffré est

$$y = RSA_e(x).$$

Le déchiffrement s'effectue à partir de y en calculant d'abord x par déchiffrement RSA :

$$x = RSA_d(y).$$

Ensuite les valeurs s et t dont on connaît les longueurs respectives sont extraites de x. On calcule alors :

$$r = H(s) \oplus t,$$
$$M = G(r) \oplus s.$$

Le message m est extrait de M.

3 RSAES-OAEP

Ce système (RSA Encryption Scheme, Optimal asymmetric Padding) est recommandé dans la directive PKCS# 1v2.1 de la Société RSA et repris dans la norme **iso 18033-2**. Ce sont les notations de cette dernière norme que nous utiliserons dans la suite du texte.

Les entrées et les sorties de données, sont en général des suites d'octets. Les traitements intermédiaires font parfois intervenir leur traduction en nombres entiers. Nous utiliserons les notions définies dans la "fichecrypto_ 109" pour les divers formats de données ainsi que les traductions d'un format à un autre.

RSAES est formé de plusieurs parties :
- une partie de génération des clés du système RSA qui va être utilisé : RSAKeyGen ;
- une partie qui concerne un mécanisme d'encodage RSA (padding) : REM. Cet encodage suit le principe OAEP décrit dans le paragraphe précédent.
- À partir de ces deux parties, RSAES assure une fonction de chiffrement RSAES.Encrypt et une fonction de déchiffrement RSAES.Decrypt.

La seule contrainte qui lie les ces parties est que la longueur en octets $\mathcal{L}(n)$ du module n de la clé RSA générée par la fonction $KeyGen()$ de la première partie soit supérieur ou égal à la borne REM.bound définie dans la partie REM.

3.1 La partie RSAKeyGen

La génération d'une paire de clés RSA comporte un algorithme probabiliste $RSAKeyGen()$ sans entrée, et qui renvoie un triplet (n, e, d) où n est un module RSA, (n, e) la clé publique et d la clé privée du système RSA (cf. la "fichecrypto_ 110").

Remarque 1 : la fonction $RSAKeyGen()$ ne comporte pas en entrée la taille du système. On peut en effet considérer que la fonction $RSAKeyGen()$ est elle même obtenue à partir d'une fonction qui à un paramètre de sécurité k (la longueur du module) associe la fonction $RSAKeyGen()$.

Remarque 2 : dans la norme, l'algorithme $RSAKeyGen()$ est autorisé à sortir des valeurs erronées pourvu que la probabilité d'une telle sortie soit négligeable, ceci pour tenir compte par exemple des tests probabilistes sur les nombres premiers qui pourraient déclarer premier un nombre qui ne l'est pas.

3.2 La partie RSA Encoding Mechanism : REM

Cette partie spécifie une borne $REM.Bound$ ainsi que deux algorithmes :

- un **algorithme probabiliste** $REM.Encode(M, L, ELen)$ dont l'entrée est constituée d'un texte clair M (suite d'octets), d'une étiquette L (suite d'octets) d'un entier représentant une longueur $ELen$ (et d'un aléa r non noté), et qui renvoie une suite d'octets de longueur $ELen$ octets. Les entrées sont soumises à la contrainte suivante sur la longueur $|M|$ de M :

  $$|M| \leq ELen - REM.Bound.$$

  Si tel n'est pas le cas l'algorithme doit partir en erreur.
- un **algorithme déterministe** $REM.Decode(E, L)$ dont l'entrée est constituée de deux suites d'octets E et L et qui doit renvoyer le texte clair M s'il existe un aléa r pour lequel

  $$E = REM.Encode(M, L, |E|)$$

  et partir en erreur sinon.

Remarque : Pour une raison technique qui tient essentiellement au fait que la primitive RSA doit s'appliquer à un entier stictement plus petit que le module on impose à la sortie de $REM.Encode$ de commencer par l'octet $Oct(0)$ constitué de huit bits 0.

Spécifions maintenant un "RSA Encoding Mechanism" nommé REM1 et qui est le seul actuellement dans la norme.

3.2.1 Les paramètres utilisés

On utilisera dans la suite une fonction de hachage publique notée $Hash$ et dont la longueur de sortie en octets est $Hash.len$. On utilisera aussi un générateur de masque (ou Key Derivation Function) noté KDF.

3.2.2 La borne REM.bound

On définit :

$$REM1.Bound = 2Hash.len + 2.$$

3.2.3 La fonction d'encodage

L'algorithme $REM1.Encode(M, L, ELen)$ fonctionne de la façon suivante :

1) On vérifie que $|M| \leq ELen - 2Hash.len - 2$, sinon erreur.

2) On note pad la suite d'octets formée par $Elen - 2Hash.len - 2 - |M|$ octets nuls.

3) On génère un aléa $seed$ qui est une suite d'octets de longueur $Hash.len$.

4) On calcule $check = Hash.eval(L)$, l'empreinte de L calculée avec la fonction de hachage $Hash$.

5) On calcule :

$$DataBlock = check||pad||\langle Oct(1)\rangle||M.$$

6) On calcule $DataBlockMask = KDF(seed, ELen - Hash.len - 1)$ qui à partir de $seed$ construit un masque de longueur $ELen - Hash.len - 1$ (c'est-à-dire exactement la longueur de $DataBlock$) grâce au générateur de masque.

7) On calcule $MaskedDataBlock = DataBlockMask \oplus DataBlock$.

8) On calcule $SeedMask = KDF(MaskedDataBlock, Hash.len)$.

9) On calcule $MaskedSeed = SeedMask \oplus seed$.

10) On renvoie :

$$E = \langle Oct(0)\rangle||MaskedSeed||MaskedDataBlock.$$

On reconnaît là, l'application de OAEP.

3.2.4 La fonction de décodage

L'algorithme $REM1.Decode(E, L)$ fonctionne de la façon suivante :

1) Soit $ELen = |E|$.

2) On vérifie que $ELen \geq 2Hash.len + 2$ sinon erreur.

3) On calcule $check = Hash.eval(L)$.

4) On analyse E comme la concaténation d'un octet X avec une suite d'octets $MaskedSeed$ de longueur Hash.len et avec une suite d'octets $MaskedDataBlock$ de longueur $Elen - Hash.len - 1$.

5) On calcule $SeedMask = KDF(MaskedDataBlock, Hash.len)$.

6) On obtient $seed = MaskedSeed \oplus SeedMask$.

7) On calcule $DataBlockMask = KDF(seed, ELen - Hash.len - 1)$.

8) On obtient $DataBlock = MaskedDataBlock \oplus DataBlockMask$.

9) On analyse $DataBlock$ comme la concaténation de $check'$ avec M' où $check'$ est constitué de $Hash.len$ octets.

10) On analyse M' comme un certain nombre m (qui peut être nul) d'octets $Oct(0)$ suivis d'un octet T (qui doit être $Oct(1)$), suivi d'une suite d'octets M.

11) Si $check' \neq check$ ou si X n'est pas l'octet $Oct(0)$ ou si T n'est pas l'octet $Oct(1)$, alors erreur.

12) Sortir M.

3.3 Synthèse assurant le fonctionnement de RSAES

On peut maintenant décrire le fonctionnement complet de RSAES-OAEP.

3.3.1 Le chiffrement

La fonction de chiffrement $RSAES.Encrypt((n, e), L, M)$ a pour entrées la clé publique RSA (n, e) de la primitive RSA utilisée, une étiquette L chargée d'identifier la transaction, un texte clair M de longueur au plus $\mathcal{L}(n) - REM.Bound$ où $\mathcal{L}(n)$ est la longueur en octets de l'entier n.

On calcule alors $E = REM.Encode(M, L, \mathcal{L}(n))$. Puis on utilise la primitive de chiffrement RSA appliquée à l'entier u représenté par la suite d'octets E ($u = OS2IP(E)$) pour obtenir tout d'abord un entier $v = RSA_{(n,e)}(u)$ puis la suite d'octets de longueur $\mathcal{L}(n) correspondante C = I2OSP(v, \mathcal{L}(n))$ qui constitue le texte chiffré.

3.3.2 Le déchiffrement

La fonction de déchiffrement $RSAES.Decrypt(d, L, C)$ a pour entrées la clé privée d de la primitive RSA utilisée, une étiquette L et un texte chiffré C. On calcule $v = OS2IP(C)$, puis $u = RSA_{(n,d)}(v)$, puis $E = I2OSP(v, \mathcal{L}(n))$. On applique alors à E la fonction de décodage pour obtenir $M = REM.Decode(E, L)$. Cet algorithme de déchiffrement peut se terminer en erreur.

4 Remarques

L'étiquette L dont il est question doit être connue des deux parties puisqu'elle intervient dans le chiffrement et dans le déchiffrement. Elle peut servir à diverses choses : identification d'une transaction, horodatage, etc. On ne dit pas comment est échangée cette étiquette, ça peut être en clair, ou par tout autre moyen.

Si une fonction part en erreur, il est prudent de ne jamais renvoyer la cause de l'erreur. Si possible, la transaction s'interrompt sans préciser quoique ce soit.

Comme E commence par un octet nul et que $|E| = \mathcal{L}(n)$, alors $u = OS2IP(E) < n$ ce qui permet d'appliquer la primitive de chiffrement RSA à u.