Encryption pour les technologies web

Michaël Mathieu

25 mars 2024

Résumé

Dans ce document, nous allons voir des exemples d'encryption et de signature avec PHP (OpenSSL) et JavaScript (CryptoJS, JSEncrypt et jsrsasign).

Table des matières

1 Avant–propos

Afin de vous assurez de votre compréhension des mécanismes d'encryption et de signature, nous recommandons l'utilisation de GPG4USB [1].

Si pour PHP, la bibliothèque OpenSSL est installée d'office, pour JavaScript, il faut référencer les bibliothèques nécessaires. Par précaution, nous vous conseillons de faire une copie locale de ces bibliothèques dont le chemin d'accès peut changer.

CryptoJS nous utilisons cette bibliothèque pour ses fonctions de hachage (par exemple SHA512), ainsi que pour l'encryption symétrique.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/4.2.0/crypto-js.min.js">
</script>
```

JSEncrypt nous utilisons cette bibliothèque pour l'encryption asymétrique.

```
<script src="http://travistidwell.com/jsencrypt/bin/jsencrypt.min.js">
</script>
```

jsrsasign nous utilisons cette bibliothèque dans le cadre de l'encryption asymétrique uniquement pour décrypter la clé privée si celle–ci est cryptée (la bibliothèque `JSEncrypt` ne le permet pas !).

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jsrsasign/11.1.0/jsrsasign-all-min.js">
</script>
```

1.1 Persistance

Pour stocker en base de données une clé publique/privée, il faut prévoir un champ avec suffisamment d'espace afin que la donnée ne soit pas tronquée (et donc qu'elle ne soit pas exploitable).
Par exemple pour une clé privée sur 2048 bits encryptée, il faut plus de 1800 caractères !

La taille des messages n'étant pas connue à l'avance, il est recommandé s'assurer que le type de données en base est suffisant pour stocker tout le message.
Avec MySQL/MariaDB, les types pour les textes sont :
— `TINYTEXT` (maximum 255 caractères)
— `TEXT` (maximum 65 535 caractères)
— `MEDIUMTEXT` (maximum 16 777 215 caractères)
— `LONGTEXT` (maximum 4 294 967 295 caractères)

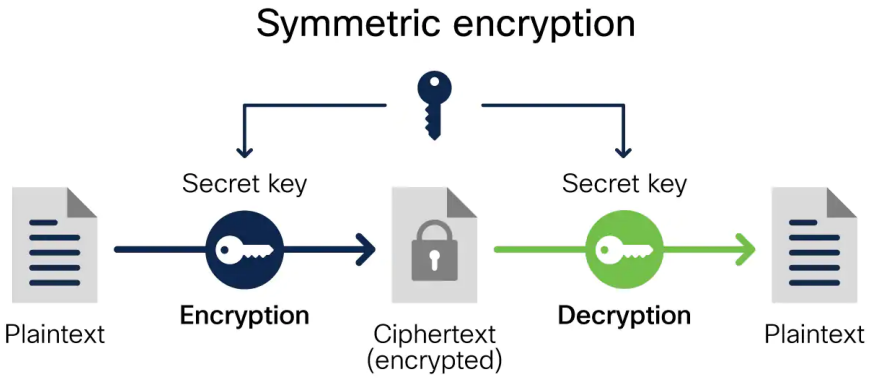
1.2 Sérialisation

Lors de l'encryption d'un message (PHP : `openssl_encrypt`, `openssl_public_encrypt`, JavaScript(CryptoJS) : `AES.encrypt`) ou de sa signature (PHP : `openssl_sign`), le résultat généré contient des caractères qui ne sont pas tous sérlialisables ce qui a comme conséquence que les données sont altérées lorsqu'elles sont persistées en base de données ou transmises par HTTP au format texte. Pour éviter cette altération des données, il faut soit les persister en `BLOB` / les transmettre avec le header `Content-Type: application/octet-stream`. Mais dans certains cas, on souhaite les persister/transférer au format texte qui permet de mettre ces données dans un JSON ou de les stocker dans le `localStorage`. Dans ce cas, il faut les encoder en *Base 64* avant de les sérialiser (PHP : `base64_encode`, JavaScript : `CryptoJS.enc.Base64.stringify`) et les décoder après la déserialisation (PHP : `base64_decode`, JavaScript : `atob`).

⚠lors de l'encryption/décryption asymétrique ou de la signature/vérification de signature en JavaScript, JSEncrypt encode/décode automatiquement en Base 64 ⇒ il ne faut donc pas, par exemple, faire du `atob` avant de vérifier une signature envoyée par du PHP, en revanche, en PHP, il faudra utiliser `base64_decode` sur la signature reçue par un client JavaScript avant d'appeler `openssl_verify`.

1. `https://www.gpg4usb.org/`

2 Encryption symétrique



Pour l'encryption symétrique, nous utilisons l'algorithme AES (Advanced Encryption Standard).
Cet algorithme nécessite un mot de passe (passphrase en anglais) de 128, 192 ou 256 bits. Lorsque nous choisissons un mot de passe, il est rare d'avoir exactement le nombre de bits nécessaires (ni plus ni moins).
Dans la pratique, nous utilisons l'algorithme PBKDF2 qui permet d'obtenir une clé qui a la bonne longueur et qui ajoute aussi de la difficulté à trouver le mot de passe.

Lorsqu'une clé secrète saisie pour décrypter des informations est très proche de la bonne clé –mais pas identique et donc fausse–, la décryption peut ne pas retourner `null`, mais une donnée mal déchiffrée. Cela engendre par exemple des exceptions comme *Malformed UTF-8 data*. Ainsi, plutôt que de mettre un `try-catch` qui pourrait masquer un réel problème d'encodage alors que la clé est correcte, nous comparons la signature du message encrypté avant de tenter de le décrypter.

2.1 PHP

2.1.1 Encrypter un message

```
$secretKey = "1234";
$message = "secret message";

$salt = openssl_random_pseudo_bytes(256);
$iv = openssl_random_pseudo_bytes(16);

$iterations = 999;
$key = hash_pbkdf2("sha512", $secretKey, $salt, $iterations, 64);

$encryptedData = openssl_encrypt($message, "aes-256-cbc", hex2bin($key),
                                 OPENSSL_RAW_DATA, $iv);
if (!$encryptedData) {
  error_log("openssl_encrypt: " . openssl_error_string());
}

$encryptedData = base64_encode($encryptedData);
$signature = hash_hmac("sha256", $encryptedData, $key);
$encryptedMessage = ["ciphertext" => $encryptedData,
                     "salt" => bin2hex($salt),
                     "iv" => bin2hex($iv),
                     "signature" => $signature];
```

La variable `$encryptedMessage` contient les données encryptées et encodées en base64 (afin de pouvoir les sérialiser sans soucis d'encodage), ainsi que le vecteur d'initialisation, le `salt` et la signature permettant de décrypter le message.

2.1.2 Décrypter un message

```
$secretKey = "1234";

$salt = hex2bin($encryptedMessage["salt"]);
$iv = hex2bin($encryptedMessage["iv"]);

$iterations = 999;
$key = hash_pbkdf2("sha512", $secretKey, $salt, $iterations, 64);

$signature = hash_hmac("sha256", $encryptedMessage["ciphertext"], $key);
if ($signature != $encryptedMessage["signature"]) {
  error_log("signature don't match");
}

$message = openssl_decrypt(base64_decode($encryptedMessage["ciphertext"]), "aes-256-cbc",
                           hex2bin($key), OPENSSL_RAW_DATA, $iv);
if (!$message) {
  error_log("openssl_decrypt: " . openssl_error_string());
}
```

La variable `$message` contient le message decrypté.

2.2 JavaScript

2.2.1 Encrypter un message

```
let secretKey = "1234";
let message = "secret message";

let salt = CryptoJS.lib.WordArray.random(256);
let iv = CryptoJS.lib.WordArray.random(16);

let key = CryptoJS.PBKDF2(secretKey, salt, { hasher: CryptoJS.algo.SHA512,
                                             keySize: 64/8,
                                             iterations: 999 });

let encryptedData = CryptoJS.enc.Base64.stringify(
                        CryptoJS.AES.encrypt(message, key, {iv: iv}).ciphertext);
let signature = CryptoJS.HmacSHA256(encryptedData, key.toString());

let encryptedMessage = {
    ciphertext: encryptedData,
    salt: CryptoJS.enc.Hex.stringify(salt),
    iv: CryptoJS.enc.Hex.stringify(iv),
    signature: signature.toString()
}
```

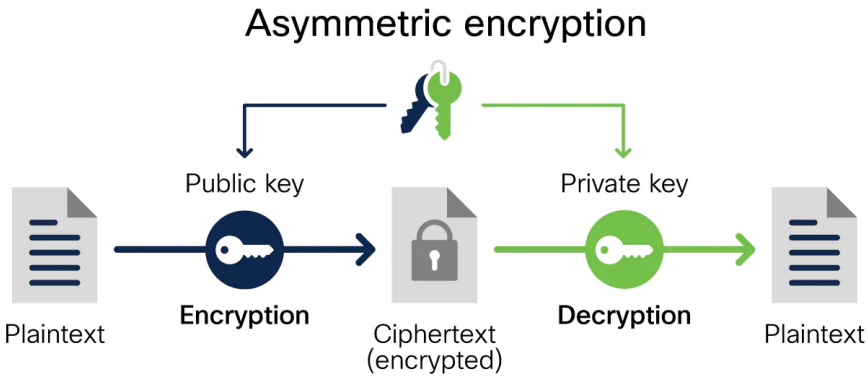
La variable `encryptedMessage` contient les données encryptées et encodées en base64 (afin de pouvoir les sérialiser sans soucis d'encodage), ainsi que le vecteur d'initialisation, le `salt` et la signature permettant de décrypter le message.

2.2.2 Décrypter un message

```
let secretKey = "1234";

let salt = CryptoJS.enc.Hex.parse(encryptedMessage.salt);
let iv = CryptoJS.enc.Hex.parse(encryptedMessage.iv);

let key = CryptoJS.PBKDF2(secretKey, salt, { hasher: CryptoJS.algo.SHA512,
                                             keySize: 64/8,
                                             iterations: 999});

let signature = CryptoJS.HmacSHA256(encryptedMessage.ciphertext, key.toString()).toString();
if (signature != encryptedMessage.signature) {
  console.error("signature don't match");
}

let decryptedData = CryptoJS.AES.decrypt(encryptedMessage.ciphertext, key, { iv: iv});
let message = decryptedData.toString(CryptoJS.enc.Utf8);
```

La variable `message` contient le message décrypté.

3 Encryption asymétrique



Pour l'encryption asymétrique, nous utilisons l'algorithme RSA (Rivest–Shamir–Adleman).

⚠L'algorithme RSA n'est pas prévu pour encrypter des données supérieurs à la taille de la clé. Voir la section suivante *Encryption asymétrique d'un message volumineux.*

3.1 PHP

3.1.1 Générer une paire de clé publique–privée

```
$config = array(
  "digest_alg" => "sha512",
  "private_key_bits" => 2048,
  "private_key_type" => OPENSSL_KEYTYPE_RSA);

$keypair = openssl_pkey_new($config);
if (!$keypair) {
  error_log("openssl_pkey_new: " . openssl_error_string());
}
```

3.1.2 Récupérer la clé publique

```
$publicKeyDetails = openssl_pkey_get_details($keypair);
if (!$publicKeyDetails) {
  error_log("openssl_pkey_get_details: " . openssl_error_string());
}
$publicKey = $publicKeyDetails["key"];
```

La variable `$publicKey` contient :

```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQE etc.
-----END PUBLIC KEY-----
```

3.1.3 Récupérer la clé privée

Clé privée non–cryptée

```
openssl_pkey_export($keypair, $privateKey, null, []);
if (!$privateKey) {
  error_log("openssl_pkey_export: " . openssl_error_string());
}
```

La variable `$privateKey` contient :

```
-----BEGIN PRIVATE KEY-----
u46UB2OC1wMhiIoc61R etc.
-----END PRIVATE KEY-----
```

Clé privée cryptée

```
$passphrase = "1234";
openssl_pkey_export($keypair, $encryptedPrivateKey, $passphrase, []);
if (!$encryptedPrivateKey) {
  error_log("openssl_pkey_export: " . openssl_error_string());
}
```

La variable `$encryptedPrivateKey` contient :

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
MIICxjBABgkqhkiG9w0 etc.
-----END ENCRYPTED PRIVATE KEY-----
```

3.1.4 Encrypter un message

```
$message = "secret message";
if (!openssl_public_encrypt($message, $encrypted, $publicKey)) {
  error_log("openssl_public_encrypt: " . openssl_error_string());
}
```

La variable `$encrypted` contient le message crypté.

3.1.5 Décrypter un message

Clé privée non–cryptée

```
if (!openssl_private_decrypt($encrypted, $message, $privateKey)) {
  error_log("openssl_private_decrypt: " . openssl_error_string());
}
```

La variable `$message` contient le message décrypté.

Clé privée cryptée

```
$passphrase = "1234";
$privateKey = openssl_pkey_get_private($encryptedPrivateKey, $passphrase);
if (!$privateKey) {
  error_log("openssl_pkey_get_private: " . openssl_error_string());
}
if (!openssl_private_decrypt($encrypted, $message, $privateKey)) {
  error_log("openssl_private_decrypt: " . openssl_error_string());
}
```

La variable `$message` contient le message décrypté.

3.2 JavaScript

3.2.1 Générer une paire de clé publique–privée

```
let keypair = new JSEncrypt({default_key_size: 2048});
keypair.getKey();
```

3.2.2 Récupérer la clé publique

```
let publicKey = keypair.getPublicKey();
```

La variable `publicKey` contient :

```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQE etc.
-----END PUBLIC KEY-----
```

3.2.3 Récupérer la clé privée

Clé privée non–cryptée

```
let privateKey = KEYUTIL.getPEM(KEYUTIL.getKey(keypair.getPrivateKey()), "PKCS8PRV");
```

La variable `privateKey` contient :

```
-----BEGIN PRIVATE KEY-----
u46UB2OC1wMhiIoc61R etc.
-----END PRIVATE KEY-----
```

Clé privée cryptée Malgré de nombreuses tentatives, je n'ai pas réussi à crypter/décrypter une clé privée ayant été générée avec *JSEncrypt* : l'encryption de la clé privée fonctionne, mais je n'arrive pas à décrypter cette clé ensuite.
Ce problème n'est pas forcément bloquant, car dans la plupart des cas, lorsque nous avons besoin de crypter une clé privée, nous le faisons sur le serveur (en PHP) et non sur le client en (JavaScript).

3.2.4 Encrypter un message

```
let message = "secret message";
let crypter = new JSEncrypt();
crypter.setPublicKey(publicKey);
let encrypted = crypter.encrypt(message);
```

La variable `encrypted` contient le message crypté.

3.2.5 Décrypter un message

Clé privée non–cryptée

```
let decrypter = new JSEncrypt();
decrypter.setPrivateKey(privateKey);
let message = decrypter.decrypt(encrypted);
```

La variable `message` contient le message décrypté.

Clé privée cryptée

```
let passphrase = "1234";
let decrypter = new JSEncrypt();
let privateKey = KEYUTIL.getKey(encryptedPrivateKey, passphrase);
decrypter.setPrivateKey(privateKey);
let message = decrypter.decrypt(encrypted);
```

La variable `message` contient le message décrypté.

4 Encryption asymétrique d'un message volumineux

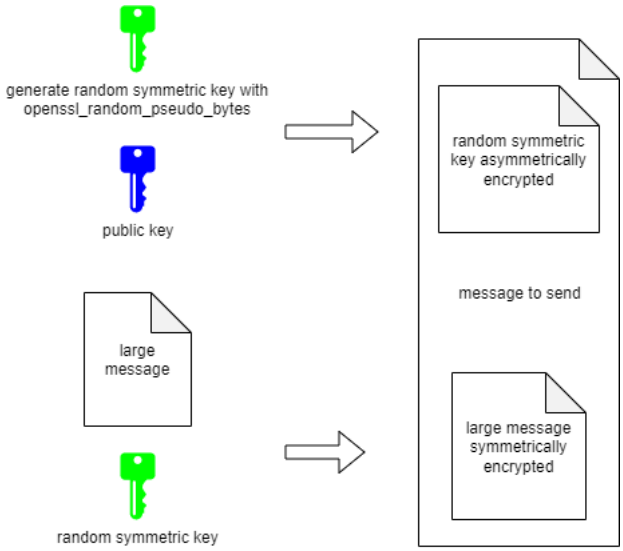
L'algorithme RSA n'est pas prévu pour encrypter des données supérieurs à la taille de la clé [2]. Avec une clé de 2048 bits (256 octets), moins le padding et les entêtes (11 octets pour PKCS#1 v1.5 padding) cela nous limite donc à 245 octets.

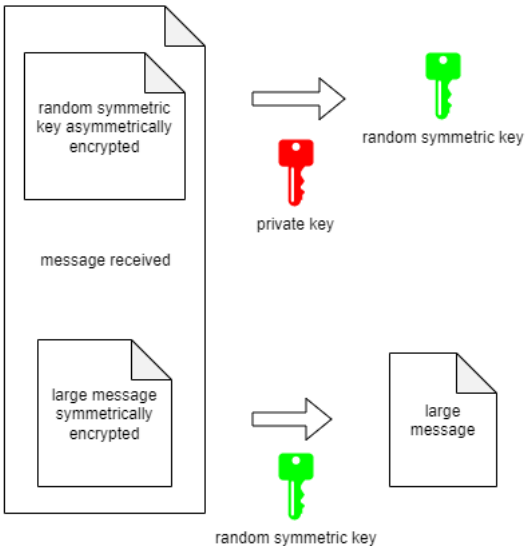
La plupart du temps, nous ne pouvons pas encrypter directement avec RSA des fichiers (RSA n'a pas été conçu pour cela). Si nous souhaitons encrypter plus de données, nous pouvons en revanche :

1. Générer une clé aléatoire (cf. `openssl_random_pseudo_bytes(245, $strongResult = true)`) (clé verte)
2. Encrypter le message avec AES en utilisant la clé aléatoire
3. Encrypter la clé aléatoire avec RSA en utilisant la clé publique du destinataire (clé bleue)
4. Envoyer les données encryptées avec AES et la clé encryptée avec RSA au destinataire

Pour décrypter le message, nous devons :

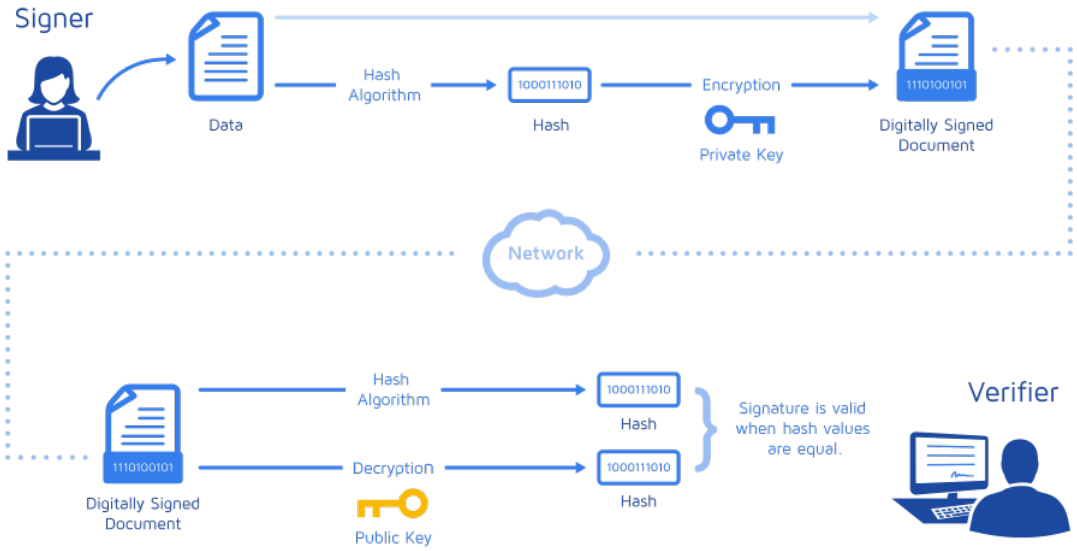
1. Décrypter la clé aléatoire (clé verte) en utilisant la clé privée (clé rouge)
2. Décrypter le message en utilisant la clé aléatoire





2. `https://tls.mbed.org/kb/cryptography/rsa-encryption-maximum-data-size`

5 Signature



Pour la signature, nous utilisons les algorithmes RSA et SHA512.

5.1 PHP

5.1.1 Signer un message

La liste des algorithmes de signatures sont disponibles ici : `https://www.php.net/manual/fr/openssl.signature-algos.php`.

```
$message = "secret message";
if (!openssl_sign($message, $signature, $privateKey, OPENSSL_ALGO_SHA512)) {
  error_log("openssl_sign: " . openssl_error_string());
}
```

La variable `$signature` contient la signature du message.

5.1.2 Vérifier la signature d'un message

```
$message = "secret message";
if (openssl_verify($message, $signature, $publicKey, OPENSSL_ALGO_SHA512)) {
  // signature matches message
} else {
  // signature DOES NOT match message
}
```

5.2 JavaScript

5.2.1 Signer un message

La liste des algorithmes de signatures sont disponibles ici : `https://cryptojs.gitbook.io/docs/#hashing`.

```
let message = "secret message";
let signer = new JSEncrypt();
signer.setPrivateKey(privateKey);
let signature = signer.sign(message, CryptoJS.SHA512, "sha512");
```

La variable `signature` contient la signature du message.

5.2.2 Vérifier la signature d'un message

```
let message = "secret message";
let verifier = new JSEncrypt();
verifier.setPublicKey(publicKey);
if (verifier.verify(message, signature, CryptoJS.SHA512)) {
  // signature matches message
} else {
  // signature DOES NOT match message
}
```