ICR - Labo #2 : *Conception et implémentation d'un container sécurisé pour des données médicales*

G.Burri

15 décembre 2014

1 Introduction

Le but de ce laboratoire est de définir les algorithmes cryptographiques et leurs paramètres afin de sécuriser des données médicales. Une donnée médicale est représentée par un fichier qui devra être sécurisé au sein d'un container dont le format sera défini par nos soins. Une implémentation sera ensuite proposée.

2 Niveaux de sécurité

2.1 Quel est le niveau de sécurité que l'on souhaite atteindre ?

Le niveau souhaité est de 128 bits. Cela implique l'utilisation d'une clef *AES* de 128 bits et de clefs *RSA* de 3072 bits d'après *Wikipedia* [4].

Les éléments de sécurité suivants sont requis :

- Confidentialité : les données chiffrées ne doivent pas pouvoir être décryptées par un attaquant.
- Authenticité : un attaquant ne doit pas pouvoir forger un container. Une signature est réalisée à l'aide d'une paire de clefs *RSA* publique-privée.
- Intégrité : il ne faut pas que les données chiffrées aient pu être altérées par un attaquant.

2.2 Comment s'assure-t-on que les données sont stockées de manière confidentielle ? En particulier en ce qui concerne les méta-données ?

Les méta-données ainsi que les données sont chiffrées ensemble. Voir le format du container décrit à la section 4.

2.3 Comment s'assure-t-on que les données stockées sont authentiques ? Quels sont les risques à prendre en compte ?

L'empreinte des données chiffrées est signée à l'aide d'une clef privée donnée en paramètre de l'*API* : ceci représente la signature qui est placée dans le container. Lors du déchiffrement, la clef publique correspondante est fournie puis utilisée pour vérifier la signature avec l'empreinte des données chiffrées.

2.4 Comment s'assure-t-on que les données stockées sont intègres ?

Cela est réalisé avec un *MAC*, dans notre cas nous utilisons *HMAC-SHA256* sur l'ensemble des données chiffrées (*Encrypt-then-MAC*).

2.5 Quels sont les clefs cryptographiques requises qu'il est nécessaire de gérer ?

2.5.1 Clefs externes

Concerne les clefs externes à l'*API*.
— Une paire de clefs *RSA-3072* pour la signature.
— Une paire de clefs *RSA-3072* pour le chiffrement des clefs *AES* et *HMAC*.

2.5.2 Clefs internes

Concerne les clefs gérées à l'intérieur du container.
— Une clef de 128 bits pour *AES*.
— Une clef de 256 bits pour *HMAC*.
Ces clefs sont générées aléatoirement à chaque création d'un container.

3 Choix des algorithmes et des paramètres

— *RSA-3072* pour la signature ainsi que pour le chiffrage des clefs *AES* et *HMAC*. Le bourrage *OAEP* (*PKCS#1 v2*) est utilisé ;
— *HMAC-SHA256* pour la vérification de l'intégrité ;
— *AES-CBC128* pour le chiffrement symétrique du contenu du fichier et des méta-données associées. Le bourrage *PKCS7* est utilisé.

D'après *Wikipedia* [4], la société *RSA Security*[1] annonce qu'une taille de clefs *RSA* de 3072 bits est suffisante pour une utilisation au delà de 2030. Cela dépend également du niveau d'importance des documents que l'on souhaite chiffrer dans la mesure où une attaque demande énormément de moyens.

Toujours d'après *Wikipedia* [4], une taille de clef *AES* de 128 bits reste, actuellement, hors de portée de toutes attaques.

1. `http://en.wikipedia.org/wiki/RSA_Security`

4 Format du container

Le format est défini comme suit en *EBNF*. Les valeurs entre crochets correspondent soit à une taille en bits soit à un type.

```
container = header, ciphertext ;
header = mac[256], signature[3072], keys[3072] ;
ciphertext = AES(plaintext) ;
plaintext = meta-data, file-content ;
meta-data = nb-meta-data[byte], { key-value-pair } ;
key-value-pair = key[string], value[string] ;
string = size[vint], content-utf8 ;
```

`nb-meta-data` est le nombre de paires clef-valeur des méta-données.

`keys` correspond aux clefs k_c et k_a ainsi qu'à l'*IV*, le tout chiffré avec *RSA-3072*. La taille des données à chiffrer est égale à $k_c + k_a + iv = 128 + 256 + 128 = 512\,bits$.

Les méta-données (`meta-data`) peuvent contenir, par exemple, le nom du fichier, sa date de création, ses droits, ou toutes autres données associées.

Le type `vint` correspond à un entier de taille variable, initialement occupant un octet.

Comme les clefs (*AES* et *HMAC-SHA256*) sont différentes à chaque chiffrement, que le *MAC* dépend de sa clef et des données chiffrées et que la signature dépend du *MAC*, alors l'ensemble des octets des différentes parties du fichier résultat va être fortement différent d'un chiffrement à l'autre pour le même fichier en entrée.

5 Processus

5.1 Chiffrement

Entrées :
— f : fichier
— k_{pub} : clef publique RSA
— $k_{signpriv}$: clef privée de signature RSA
Sortie :
— c : container chiffré.
Processus :

1. Génération d'une clef 128 bits pour $AES \rightarrow k_c$.
2. Génération d'une clef 256 bits pour $MAC \rightarrow k_a$.
3. Génération d'un IV 128 bits pour le mode $CBC \rightarrow iv$.
4. Construction du *plaintext* à partir de f, voir format décrit à la section 4.
5. Chiffrement du *plaintext* avec *AES-CBC128*, k_c et $iv \rightarrow ciphertext$.
6. Calcul du *HMAC-SHA256* de $ciphertext \rightarrow mac$.
7. Signature de mac avec $k_{signpriv} \rightarrow sig$.
8. Chiffrement de $k_c + k_a + iv$ avec $k_{pub} \rightarrow keys$.
9. $mac + sig + keys + ciphertext \rightarrow c$.

Où + dénote la concaténation.

5.2 Déchiffrement

Entrée :
— c : container chiffrés
— k_{priv} : clef privée RSA
— $k_{signpub}$: la clef publique de signature RSA
Sortie :
— f : fichier original
Processus :

1. Lecture de mac, calcul de mac' sur c, comparaison des deux valeurs afin de vérifier l'intégrité.
2. Vérification de la signature avec $k_{signpub}$.
3. Déchiffrement de $k_c + k_a + iv$ avec k_{priv}.
4. Déchiffrement du reste des données $(ciphertext) \rightarrow f$.

Ce processus nécessite deux cycles de lecture des données, le premier pour le calcul de mac' et le deuxième pour le déchiffrement. Le deuxième cycle n'est effectué que si l'intégrité et l'authenticité ont été validées.

6 Implémentation

Nous utilisons ici la plate-forme *.NET* ainsi que le langage *F#*, un dialecte de *ML*[2]. L'ensemble des éléments cryptographiques requis sont fournis par *.NET*[3].
Deux *assemblies* sont créées :
— *CryptoFile* : *Library* mettant à disposition l'*API* de chiffrement de fichier et de déchiffrement de container.
— *CryptoFileTests* : Exécutable utilisant la *library CryptoFile* et permettant d'utiliser l'*API* à l'aide d'arguments fournis par la ligne de commande.

6.1 Utilisation

Il est possible de compiler la solution à l'aide de *MonoDevelop*[4] ou de *Visual Studio 2012*. Le script *Bash* `labo2-fsharp/run_tests.sh` permet de compiler la solution puis d'exécuter un certain nombre de tests.

À partir du dossier `labo2-fsharp` et après avoir compilé en *release* la solution, voici ce qu'il est possible d'effectuer :
— `CryptoFileTests/bin/Release/CryptoFileTests.exe tests` : Réalise une série de tests.
— `CryptoFileTests/bin/Release/CryptoFileTests.exe encrypt <file> <container>` : Chiffre le fichier `<file>` vers le container `<container>`.
— `CryptoFileTests/bin/Release/CryptoFileTests.exe decrypt <container> <output directory>` : Déchiffre le container `<container>` dans le dossier `<output directory>`.

2. http://en.wikipedia.org/wiki/ML_%28programming_language%29
3. http://msdn.microsoft.com/en-us/library/System.Security.Cryptography%28v=vs.110%29.aspx
4. http://www.monodevelop.com/

Les clefs publiques et privées pour le chiffrement ainsi que pour la réalisation de la signature se trouvent dans les fichiers `keys-crypt.priv`, `keys-crypt.pub`, `keys-sign.priv` et `keys-sign.pub`. Ceux-ci sont automatiquement générés dans le cas où ils sont introuvables.

6.2 Organisation du code

La *library CryptoFile* est composée de trois fichiers :
— *Types.fs* : Quelques types publics.
— *Crypto.fs* : Toutes les primitives cryptographiques nécessaires.
— *UnitTests.fs* : Quelques tests unitaires du module *Crypto*.
— *API.fs* : L'interface publique de la *library*. Elle est détaillée à la section 6.2.1.

6.2.1 API

Voici la partie publique de la *library CryptoFile*.

```
module API =
    (* Generates a pair of keys (public * private)
       to be used in the following two functions.
       You have the reponsability of keeping
       the private part secret. *)
    let generatKeysPair : Key * Key

    let encryptFile (inputFilePath : string)
                    (outputFilePath : string)
                    (signaturePrivKey: Key)
                    (cryptPubKey : Key)

    let decryptFile (sourceFilePath : string)
                    (targetDirPath : string)
                    (signaturePubKey: Key)
                    (decryptPrivKey : Key)
```

Les formats des clefs, publique et privée, sont décrits sur cette page [5].

6.3 Mesures de performance

Quelques mesures sur un fichier de 871 MiB ont été effectuées sous *Linux* avec *Mono* 3.10.0 ainsi que sous *Windows 8.1* avec *Visual Studio 2012*. Il est à noter que l'implémentation *AES* de *Mono* est en *C#* et n'utilise évidemment pas l'accélération matérielle d'*Intel* présente sur la machine : *AES-NI*.

Les tests sous *Windows 8* ont été fais sur une machine ne possédant pas *AES-NI*. Cet ensemble d'instructions est normalement supporté par l'implémentation du *runtime .NET* de *Microsoft*.

	Chiffrement		Déchiffrement	
	Mono	*MS .NET*	*Mono*	*MS .NET*
Temps	39 s	20 s	48 s	20 s
Mémoire utilisée	7.0 MiB	14 MiB	15.2 MiB	13.9 MiB
Taux *CPU*	1 x 100 %	1 x 100 %	1 x 100 %	1 x 100 %

5. `http://msdn.microsoft.com/en-us/library/system.security.cryptography.rsa.toxmlstring%28v=vs.110%29.aspx`

6.3.1 Génération de paire de clefs *RSA*

La génération de clefs *RSA* est très lente sous *Mono*. Pour une taille de 2048 bits cela prend environ une seconde, pour une taille de 3072 bits cela prend environ dix secondes. Sous *Windows*, la version *.NET* de *Microsoft* est environ dix fois plus rapide.

7 Analyse de la sécurité de l'implémentation

7.1 Quelles sont les parties critiques du code et comment s'assure-t-on que ces parties sont correctement implémentées ?

Le choix des algorithmes, de leurs paramètres et de leur implémentation est une partie critique. Il est possible de se référer aux recommandations de certains organismes comme par exemple le *NIST*[6].

La génération des clefs *AES* doit être faite avec un générateur cryptographique. Dans notre cas nous utilisons *RNGCryptoServiceProvider*[7].

7.2 Quels sont les points faibles restants et quelles sont les possibilités de les corriger ?

Les deux clefs privées *RSA* doivent absolument rester secrètes. Pour ce faire, il faudrait chiffrer les fichiers contenant ces clefs à l'aide d'une *passphrase* robuste et garder celle-ci en sécurité.

8 Conclusion

Ce laboratoire a permis de mettre en évidence la problématique de la sécurisation de fichiers ainsi que de leurs méta-données associées. Le choix de bons algorithmes et des bons paramètres associés est capital pour garantir la sécurité des fichiers.

Références

[1] Stephen Haunts. Cryptography in .net : Advanced encryption standard (aes). `http://stephenhaunts.com/2013/03/04/cryptography-in-net-advanced-encryption-standard-aes/`, 2013.

[2] Stephen Haunts. Cryptography in .net : Rsa. `http://stephenhaunts.com/2013/03/26/cryptography-in-net-rsa/`, 2013.

[3] Wikipedia. Digital signature — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Digital_signature`, 2014.

[4] Wikipedia. Key size — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Key_size`, 2014.

6. *National Institute of Standards and Technology*

7. `http://msdn.microsoft.com/en-us/library/system.security.cryptography.rngcryptoserviceprovider%28v=vs.110%29.aspx`