ICR - Practical Work #2

Designing and Implementing a Secure Container for Medical Data

Gugger Joël [1]

May 17, 2016

[1] joel.gugger@master.hes-so.ch

Abstract

A university hospital has mandated you to design and implement a secure archival software system for documents containing medical data. Legal regulations require that those data are kept in a highly secure way. In the following, we will focus on confidentiality, authenticity and integrity of the archived data, and we will assume that their availability is handled by separate mechanisms.

In order to simplify the scenario, a further assumption that can be taken is that these documents containing sensitive medical data are stored in flat (binary) files, that can however have an arbitrary bytes length. Furthermore, one can assume that the unprotected files have been compressed.

To implement this secure archival system, you can freely select the cryptographic library (e.g., Botan, OpenSSL, PolarSSL, etc.) and use either the C, C++, Java or Python programming language.

Contents

Chapter 1

Security Requirements

1.1 Questions

1.1.1 What is the overall security level that you are targeting?

Le but de ce container est de permettre l'archivage des données. Le context d'utilisation de l'utilitaire serait sur un serveur de confiance de l'université médicale ou se trouve une clé privée.

Cette clé est utilisée pour chiffrer une partie du contenu et signer les containers. Il peut être envisagé d'utiliser plusieurs paire de clés pour différents service d'archive. Si cette clé fuite, le système est compromis. Si le serveur sur lequel les containers sont créés n'est de confiance, le système est compromis.

Les exigences définies sont de :

1. pouvoir detecter si l'archive a été alterée après génération (intégrité des données)
2. pouvoir garantir la provenance du container (autheticité)
3. ne pas être capable de savoir combien de fichiers sont archivés
4. ne pas savoir le nom des fichiers archivés

1.1.2 How can you ensure that the stored data will be kept in a confidential way? What about the file metadata?

Pour être sur d'obtenir une confidentialité élevée, un les métadonnées sont chiffrées avec le contenu. Pour y arriver, tout les fichiers devant être archivés sont compresser pour obtenir un seul gros fichier, contenant les entêtes et les contenus.

Le format d'archive zip est utilisé. Malheureusement je me suis rendu compte trop tardivement que celui-ci ne conservait pas les permissions sur les fichiers ainsi que les dates de création/modification. Une amélioration nécessaire à apporter serait d'utiliser le format d'archive tar, qui elle conserve ces métadonnées de fichiers.

Une fois compresser, le contenu de l'archive et chiffré symétriquement avec AES256 en mode CBC avec un vecteur d'initialisation de 128 bits. Cette méthode permet de cacher toutes les métadonnées sauf celle de la taille globale des fichiers. On ne peut pas savoir combien de fichiers sont stocker dans le container, donc on ne peut pas retrouver la taille de chaque fichier. Cependant le poids global peut donner une première indication sur le type de fichiers archivés.

Cette méthode de chiffrement symétrique permet de chiffrer un grand nombre de données, et convient donc pour les gros volume de fichiers. Les données sont confidentiel car il est nécessaire d'avoir la clé privée du serveur pour déchiffrer la clé utilisée pour le chiffrement AES256.

1.1.3 How can you ensure that the stored data are authentic? What is the risk addressed?

L'authenticité des données est garantie grâce à une signature effectuée à la fin du processus. Cette signature est faite grâce à la clé privée du serveur, permettant l'authentification de l'émetteur du container. La signature est basée sur du SHA256.

Le principal risque est la perte de la clé privée ou son vol. Dans le premier cas il ne serait plus possible de vérifier la provenance des container, ni même de les relire. Dans le deuxième cas, il ne serait plus possible de garantir la provenance des containers. Il serait alors possible pour l'attaquant de créer de fausse données médicales et de les archivées.

1.1.4 How can you ensure that the stored data keep their integrity?

L'intégrité des données est vérifiée en premier lieu lors de la lecture d'un container. Celle-ci se fait grâce à la signature imposée sur tout le container. Le principe utilisé est celui de *encrypt-then-mac*.

1.1.5 What are the cryptographic keys required to be managed in your system?

La seul exigence du système est une clé privée sous format .PEM stockée directement sur le serveur utilisé pour générer les containers. Cette clé privée ne doit pas contenir de mot de passe. La clé et le vecteur d'initialisation de chaque container est générée par le système.

1.2 Data format of a secure container

Le format d'un container contient quatre informations distinctes : le vecteur d'initialisation, la clé symétrique chiffrée, le ciphertext et la signature. Les quatre éléments sont placés de la manière suivante (l'opérateur | fait office de concaténation) :

$$IV|K|C...C|M$$

Le vecteur d'initialisation, la clé de chiffrement symétrique et la signature ont les longueurs définie comme tel :

Element	longueur (bytes)
IV	16
Key	256
Mac	256

La longueur de la clé est celle une fois chiffrée par la clé public donnée en paramètre à l'utilitaire. La clé de celle-ci non-chiffrée est évidemment de 256 bits (AES256).

Le ciphertext $C...C$ contient toutes les informations des fichiers du container. Il représente le contenu binaire des fichiers sous format d'archive zip avec un padding de type *PKCS7* pour correspondre au critère de chiffrement de l'AES256 en mode CBC.

Pour retrouver le contenu de l'archive, les étapes suivantes sont effectuées :

1. Récupération de la clé privée asymétrique
2. Vérification de la signature sur $IV|K|C...C$
3. Si celle-ci est valide, déchiffrement de la clé symétrique
4. Récupération du vecteur d'initialisation
5. Déchiffrement du ciphertext avec la clé symétrique et l'IV
6. Suppression du padding *PKCS7*
7. Reconstitution de l'archive et desarchivage dans le dossier cible

Le processus de génération du container est similaire, dans le sens inverse. A cela s'ajoute la génération aléatoire du vecteur et de la clé via le device *urandom* des systèmes UNIX-like. Le processus est le suivant :

1. Génération de l'archive zip
2. Récupération de la clé privée asymétrique
3. Génération du vecteur IV et de la clé K
4. Padding du binaire de l'archive avec *PKCS7*
5. Chiffrement du binaire de l'archive
6. Chiffrement de la clé symétrique avec la clé public
7. Signature sur $IV|K|C...C$ avec la clé privée
8. Sauvegarde du binaire concaténé dans le fichier *.medivac*

Chapter 2

Implementation and Tests

L'implémentation a été faite en Python sous le nom de **Medivac**. Medivac est un utilitaire en ligne de commande permettant de chiffrer et déchiffrer des containers. Pour créer un container avec N fichiers :

```
medivac ~/.ssh/private_key.pem file1.txt file2.txt
```

Le premier argument doit être le chemin vers une clé privée RSA au format PEM. Cette clé permet le chiffrement de la clé symétrique et la signature du container. Les arguments suivant sont un liste de fichiers à inclure dans le container.

Pour déchiffrer un container, le premier argument doit être égal à −d, le second doit être le chemin vers la clé RSA utilisée pour créer le container, troisième argument doit être le chemin vers le fichier .medivac et le dernier, optionnel, le dossier de sortie :

```
medivac -d ~/.ssh/private.pem medfile.medivac ./out/
```

Le dossier de sortie par défaut et le dossier courant.

2.1 Encryption implementation

La première étape de processus de chiffrement permet de récupérer la clé privée au format PEM.

```
67    key = None
68    if os.path.isfile( private_key ) :
69      with open( private_key, "rb" ) as key_file:
70        private_key = serialization.load_pem_private_key(
71          key_file.read(),
72          password=None,
73          backend=default_backend()
74        )
```

Si le fichier zip temporaire est déjà présent, il est supprimé. Ce cas ne devrais pas arrivé car les fichiers temporaires sont systématiquement supprimés lors de la création des containers.

```
76    if os.path.isfile( ZIP_FILENAME ) :
77      print ( '  ' + ZIP_FILENAME + ' already exist!' )
78      print ( '  Removing file ' + ZIP_FILENAME )
79      os.remove( ZIP_FILENAME )
80
81    print ( """\n  Start compresion...\n""" )
82    with zipfile.ZipFile(ZIP_FILENAME, 'x') as medzip :
83      for medFile in args:
84        if os.path.isfile( medFile ) :
85          print ( '   * Adding ' + str ( medFile)  )
86          medzip.write( str ( medFile) )
87      medzip.close()
88    print ( """\n\n  Start encryption:\n""" )
```

Une fois l'archive temporaire créée, on récupère son contenu pour le chiffrer. L'archive créée doit être impérativement supprimée à la fin du processus.

```
90    fh = open( ZIP_FILENAME, 'rb' )
91    try :
92      plaintext = fh.read()
```

On récupère la clé publique asymétrique, on pad le contenu binaire de l'archive et on génère aléatoirement la clé symétrique et le vecteur d'initialisation.

```
94      public_key = private_key.public_key()
95
96      padder = padding.PKCS7(128).padder()
97      padded_plaintext = padder.update( plaintext )
98      padded_plaintext += padder.finalize()
99
100     print ( """  - Generating key..."""  )
101     key = os.urandom( 32 )     # 256 bits
102     iv = os.urandom( 16 )      # 128 bits
```

Une fois le binaire paddé avec *PKCS7* et les clés générées, on peut chiffrer l'archive avec de l'*AES256* en mode *CBC*. La clé symétrique est ensuite chiffrée à sont tour par la clé publique pour être stockée dans le container.

```
104     print ( """  - Encrypt..."""  )
105     cipher = Cipher(
106       algorithms.AES( key ),
107       modes.CBC( iv ),
108       backend=backend
109     )
110     encryptor = cipher.encryptor()
```

```
111     ciphertext = encryptor.update( padded_plaintext ) +
        ↪  encryptor.finalize()
112
113     encrypted_key = public_key.encrypt(
114       key,
115       cryptography.hazmat.primitives.asymmetric.padding.OAEP(
116         mgf=... .primitives.asymmetric.padding.MGF1(
117           algorithm=hashes.SHA1()
118         ),
119         algorithm=hashes.SHA1(),
120         label=None
121       )
122     )
```

L'IV, la clé symétrique et le ciphertext sont concaténé et ensuite signé par la clé privée pour garantir l'intégrité et l'authenticité du container. La clé est concaténée à son tour pour donner le binaire final.

```
124     print ( """  - Signing..""" )
125     for_signature = iv + encrypted_key + ciphertext
126
127     signer = private_key.signer(
128       cryptography.hazmat.primitives.asymmetric.padding.PSS(
129         mgf=... .primitives.asymmetric.padding.MGF1(
            ↪  hashes.SHA256() ),
130         salt_length=... .asymmetric.padding.PSS.MAX_LENGTH
131       ),
132       hashes.SHA256()
133     )
134     signer.update( for_signature )
135     signature = signer.finalize()
136
137     medivac = for_signature + signature
```

Le binaire final est ensuite sauvegardé dans le fichier medfile.medivac. L'archive temporaire est ensuite supprimée.

```
139     print ( """  - Saving file...\n""" )
140     with open( MED_FILENAME, 'wb' ) as f :
141       f.write( medivac )
142       f.close()
143
144   finally :
145     fh.close()
146
147   print ( """\n  Removing temp files...\n""" )
148   if os.path.isfile( ZIP_FILENAME ) :
```

```
149     os.remove( ZIP_FILENAME )
150     print ( '   * ' + ZIP_FILENAME + ' removed' )
```

2.2 Decryption implementation

La première étape de l'implémentation du processus de déchiffrement est de récupérer la clé privée au format PEM ainsi que de vérifier les paramètres d'entrées.

```
169   backend = default_backend()
170   key = None
171   if os.path.isfile( private_key ) :
172     with open( private_key, "rb" ) as key_file:
173       private_key = serialization.load_pem_private_key(
174         key_file.read(),
175         password=None,
176         backend=default_backend()
177       )
178
179   if len( args ) < 1 or len( args ) > 2 :
180     print ( """  > Number of args incorect, exit ... \n""" )
181     sys.exit( 0 )
182
183   medfile = str( args[0] )
184
185   try:
186     output = str( args[1] )
187     if not os.path.isdir( output ) :
188       output = '.'
189   except IndexError:
190     output = '.'
```

Le contenu binaire du fichier medivac passé en paramètre est récupéré après avoir testé sont existance.

```
192   if ( os.path.isfile( medfile ) ) :
193     print ( '  Reading file ' + medfile )
194     print ( '  Output result in ' + output + '\n' )
195
196     print ( """  Start decryption:\n""" )
197     fh = open( medfile, 'rb' )
198     try :
199       b = fh.read()
```

Le binaire récupéré est découpé en deux blocs afin de récupérer le contenu et la signature. Celle-ci est vérifiée grâce à la clé passée en argument lors de l'appel de l'utilitaire.

```
201       for_signature = b[:-256]
202       signature = b[-256:]
203
204       print ( """  - Check signature..."""  )
205       public_key = private_key.public_key()
206       verifier = public_key.verifier(
207         signature,
208         cryptography.hazmat.primitives.asymmetric.padding.PSS(
209           mgf=... .asymmetric.padding.MGF1(
210             hashes.SHA256()
211           ),
212           salt_length=... .padding.PSS.MAX_LENGTH
213         ),
214         hashes.SHA256()
215       )
216       verifier.update( for_signature )
217       try :
218         verifier.verify()
219       except cryptography.exceptions.InvalidSignature :
220         error_then_quit()
```

Si la clé est vérifiée, on récupère le vecteur d'initialisation, la clé symétrique chiffrée et le ciphertext. La clé est ensuite déchiffrée par la clé privée.

```
222       iv = for_signature[:16]
223       encrypted_key = for_signature[16:272]
224       ciphertext = for_signature[272:]
225
226       print ( """  - Retreive key..."""  )
227       key = private_key.decrypt(
228         encrypted_key,
229         cryptography.hazmat.primitives.asymmetric.padding.OAEP(
230           mgf=... .primitives.asymmetric.padding.MGF1(
231             algorithm=hashes.SHA1()
232           ),
233           algorithm=hashes.SHA1(),
234           label=None
235         )
236       )
```

On récupère le plaintext grâce à l'*IV* et à la clé, puis on enlève le padding *PKCS7*.

```
238       print ( """  - Decrypt file(s)..."""  )
239       cipher = Cipher(
240         algorithms.AES( key ),
```

```
241           modes.CBC( iv ),
242           backend=backend
243       )
244       decryptor = cipher.decryptor()
245       padded_plaintext = decryptor.update( ciphertext ) +
          ↪  decryptor.finalize()
246
247       unpadder = padding.PKCS7(128).unpadder()
248       plaintext = unpadder.update( padded_plaintext )
249       plaintext += unpadder.finalize()
```

On fini par recréer l'archive et desarchiver son contenu dans le dossier spécifié. On supprime le fichier temporaire de l'archive avant de quitter.

```
251       print ( """  - Saving file...\n""" )
252       if os.path.isfile( ZIP_FILENAME ) :
253         print ( '  ' + ZIP_FILENAME + ' already exist!' )
254         print ( '  Removing file ' + ZIP_FILENAME )
255         os.remove( ZIP_FILENAME )
256
257       with open( ZIP_FILENAME, 'wb' ) as f :
258         f.write( plaintext )
259         f.close()
260
261       with zipfile.ZipFile( ZIP_FILENAME ) as medzip:
262         if medzip.testzip() == None :
263           medzip.extractall( output )
264         medzip.close()
265
266       print ( """\n  Removing temp files...\n""" )
267       os.remove( ZIP_FILENAME )
268       print ( '   * ' + ZIP_FILENAME + ' removed' )
```

2.3 Questions

2.3.1 What are the critical parts in your code in terms of security?

Je vois trois points majeurs dans cette implémentation :

Premier point critique, celui de la masse de données à chiffrer. Le but étant de faire de l'archivage il doit donc être possible de chiffrer des To de données sans avoir de fuite d'informations.

Second point critique, celui des fichiers temporaires. Si les fichiers intermédiaires ne sont pas supprimés correctement il est facile de récupérer toutes les données.

Troisième point, celui des algorithmes. Les algorithmes de chiffrement doivent être choisi de manière à être le plus efficace et sûre.

2.3.2 How did you ensure that these critical parts are properly implemented?

Le point faible de la quantité de données à chiffrer se trouve plus précisément dans dans la clé symétrique chiffrée K du container. Les bytes 17 à 273 sont ceux de la clé symétrique utilisée pour le container.

$$IV|K|C...C|M$$

Ces bytes peuvent être lu par n'importe quel personne ayant accès au serveur. Pour tout les containers, cette clé symétrique est chiffrée avec la clé privée RSA. Si la même clé RSA est utilisée pour chiffrer des milliers de containers, il peut y avoir des fuites.

Cependant l'implémentation permet une certaine souplesse concernant cette clé. Si à l'usage plusieurs clés sont utilisées, sur plusieurs serveur par exemple, cela répartit le nombre de chiffrement de chacune d'elles. On évite ainsi les fuites.

L'IV n'est pas un nonce, cependant vu que la clé symétrique K et le vecteur IV sont tiré aléatoirement, il est peut probable que la combinaison de ceux-ci soit tiré deux fois ou plus. Ce qui règle le problème.

Les fichiers temporaires sont effacé, cependant il reste des failles de sécurité expliquées dans la question suivante.

Les algorithmes ont été choisi selon leurs caractéristiques première. Une paire de clés privée et publique pour l'intégrité et l'authenticité. Les paramètre de signature utilise un HMAC SHA256 et le padding se fait avec PKCS7. Le chiffrement de l'archive se fait grâce à une clé de 256 bits en AES mode CBC avec un vecteur de 128 bits. La politique de signature est encrypt-then-mac, ce qui permet de vérifier avant toute chose que le container n'a pas été altéré et provient bien du service d'archivage.

2.3.3 What are the remaining weak points in your implementation?

Les fichiers temporaires sont supprimés via un librairie Python. Le fichier à de grande chance de ne pas être correctement éffacé sur le disque. Celui-ci ayant toujours le même nom, cela comprend un risque. L'implémentation actuelle ne prend pas en compte cette faille.

Les containers peuvent être renommés et déplacé sur le serveur. Ce qui peut engendrer des fuites au niveau des métadonnées. L'utilitaire devrait vérifier la date de modification du container et figé son emplacement au moment de la création.

2.3.4 How do you propose to address the remaining weak points?

Avant suppression des fichiers temporaire, il serait nécessaire de réécrire dans le fichier des byte à \x00 pour effacer toute trace de l'archive.

La date de création/modification du container et son emplacement pourrait être rajoutée aux données signées. Ce qui permettrait la vérification que le container n'a pas été déplacé ou modifié.

Abstract

The sources of the project are available on GitHub at the following address:
https://github.com/GuggerJoel/Crypto-ICR-lab002