Revue de Code - TP2

Valentin PORTAIL

20 décembre 2024

Table des matières

1 Introduction

Dans ce TP, nous avons à notre disposition une bibliothèque de cryptographie qui permet de protéger et d'encapsuler l'accès à une *keystore*. Cette dernière peut servir à garder en mémoire des certificats d'authentification ou des clés de chiffrement.

Cependant, cette bibliothèque est déjà compilée et réunie dans une archive JAR. Notre objectif est donc d'analyser l'archive JAR afin de retrouver la clé secrète. Une fois cette dernière trouvée, notre objectif sera de comprendre davantage comment fonctionne la bibliothèque.

Pour cela, en plus de l'archive JAR `com.michelin.ACO.crypto-2.0.0.jar`, nous avons à notre disposition un environnement d'exécution Java IBM (Java JRE) et une keystore `Secret.ks` pour nous permettre de valider notre clé.

2 Recherche de la clé

2.1 Extraction des fichiers

De manière générale, une archive JAR (Java Archive) est composée de plusieurs fichiers d'extension `.class` rangés au sein d'une arborescence. Nous aurons besoin de ces fichiers pour tenter de comprendre

le fonctionnement du programme et pouvoir retrouver la clé. Pour avoir accès à ces fichiers, il suffit ainsi d'extraire le fichier `com.michelin.ACO.crypto-2.0.0.jar`. Pour ce TP, nous avons directement utilisé l'explorateur de fichiers *Nemo*, intégré à l'environnement Linux installé sur les machines de l'ISIMA, pour l'extraire l'archive.

On se retrouve ensuite avec un dossier dans lequel se retrouvent nos fichiers .class. On retrouve deux grands paquets : `com.michelin.ACO.crypto` et `com.michelin.xnet.crypto`. Dans le premier paquet, les noms des fichiers sont des combinaisons aléatoires d'une ou deux lettres. Le nom des fichiers a donc été obfusqué et nous ne pourrons pas directement comprendre leur utilité simplement en regardant leur nom. Cependant, le deuxième paquet contient deux fichiers `.class` avec un nom lisible et compréhensible : `CryptoConverter.class` et `SimpleCryptoConverter.class`, qui peuvent contenir des informations intéressantes.

2.2 Décompilation du code

À présent, notre objectif est de décompiler ces fichiers pour obtenir du code Java que nous pourrons exploiter pour retrouver la clé. Le paquet *Extension Pack for Java* de l'éditeur de code *Visual Studio Code* inclut par défaut le décompilateur *FernFlower*. À partir d'un fichier `.class`, cet outil va pouvoir reconstruire un fichier `.java`. Cependant, certaines informations comme le nom des variables locales va être perdu. Nous utilisons donc FernFlower pour pouvoir retrouver le code des classes `SimpleCryptoHandler` et `CryptoConverter`. Nous enregistrerons le code de ces classes dans des fichiers `.java` pour pouvoir l'étudier et le modifier plus facilement.

2.3 Analyse des dépendances

Une immense partie du code de ces fichiers a été obfusqué. On retrouve ainsi, par exemple, de nombreux appels de fonctions appelant le constructeur de `mE` en passant en paramètres de tableaux contenant de grands entiers. Cependant, plusieurs informations sont encore présentes, comme les importations de dépendances ou le type des variables passées en paramètres des différentes méthodes des classes. Certains noms d'attributs et de méthodes ont également été conservés. On peut ainsi comprendre que la classe `SimpleCryptoHandler` sert ainsi d'interface pour interagir avec la keystore et chiffrer et déchiffrer des objets Java. Cette dernière fait appel à d'autres classes présentes dans le paquet `com.michelin.ACO.crypto`.

Nous pourrions utiliser la même technique que précédemment pour obtenir du code Java à partir des fichiers `.class` présents dans ce paquet et ainsi mieux comprendre son fonctionnement. Cependant, il y a plus de 700 fichiers `.class` présents dans le paquet et une immense partie d'entre eux semblent avoir un code très similaire. Il faut donc trouver une autre technique pour pouvoir retrouver les fichiers `.class` réellement utiles pour le programme et les analyser. Pour cela, nous allons parcourir les différentes dépendances de nos fichiers `.class` en commençant par les classes `SimpleCryptoHandler` et `CryptoConverter`. Nous savons déjà que ces classes sont utiles puisque ce sont elles qui vont servir d'interface.

Pour la suite de l'analyse, nous allons donc nous concentrer sur les fichiers suivants (tous paquets confondus) :

- mz
- mA
- mC
- mD
- mE
- CryptoConverter
- SimpleCryptoHander

2.4 Appel de différentes méthodes

À présent, notre objectif est de comprendre plus en détail comment fonctionne notre programme. Pour cela, nous allons tenter d'exécuter plusieurs méthodes présentes dans les fichiers étudiés en tentant de faire varier les paramètres et nous allons observer les résultats obtenus. Comme il s'agit de notre principale interface avec le programme, nous allons commencer par `SimpleCryptoHandler` et nous allons tenter d'appeler son constructeur.

Pour cela, nous créons un nouveau projet Java dans lequel nous importerons notre archive JAR. Nous allons en profiter pour changer d'outil et utiliser l'éditeur de code intégré *NetBeans* qui nous permettra de gérer plus facilement le projet. N'ayant pas réussi à faire fonctionner le IBM JRE inclus avec le TP, nous retéléchargeons le IBM Java SDK 8 depuis le site d'IBM et nous l'ajoutons à NetBeans.

En appelant le constructeur de `SimpleCryptoHandler` et en lui passant une valeur `null` en paramètres, on obtient l'avertissement suivant :

```
[Date + Heure] com.michelin.ACO.crypto.mC a
WARNING: No keystore specified; Using the default location: data/key.ks
```

Cet avertissement est renvoyé par la méthode a de la classe `mC`. On peut en déduire que la classe `mC` joue un rôle clé dans l'accès à la keystore. En lisant l'avertissement, on comprend que le paramètre à passer dans le constructeur de `SimpleCryptoHandler` correspond au chemin de la keystore.

En passant en paramètre du constructeur de `SimpleCryptoHandler` le chemin vers notre keystore `Secret.ks`, on obtient le message suivant :

```
[Date + Heure] com.michelin.ACO.crypto.mD a
SEVERE: Can't open or read the Secret.ks keystore; Exception is:
com.sun.crypto.provider.SealedObjectForKeyProtector
```

La keystore `Secret.ks` ne semble donc pas pouvoir être ouverte pour le moment. Il faudra donc utiliser une autre méthode pour retrouver la clé.

2.5 Instrumentation du constructeur de SimpleCryptoHandler

Intéressons nous maintenant au constructeur de la classe `SimpleCryptoHandler`. Son code décompilé est le suivant :

```
public SimpleCryptoHandler(String var1) {
  try {
     Class var4 = Class.forName((new mE(new long[]{-8986732235973050073L,
         -64977297075237667 62L, -8288613700403925373L, 1365972465865545358L,
         569133294553221 1299L})).toString(), true,
         Thread.currentThread().getContextClassLoader());
     Method var2 = var4.getDeclaredMethod((new mE(new long[]{656445270904799409L,
        239297309973187611L})).toString());
     Object var5 = var2.invoke((Object)null);
     Method var3 = var5.getClass().getDeclaredMethod((new mE(new long[]
        {-6704157949232290634L, 4033087034327257981L})).toString());
     String var6 = (String)var3.invoke(var5);
     this.facade = mC.a(var1, var6);
  } catch (Exception var7) {
     LOGGER.log(Level.SEVERE, MessageFormat.format("Unable to intialize Xnet
```

```
            cryptography services; The application will surely crash: {0}",
            var7.getMessage()));
    }
}
```

Grâce aux expérimentations précédentes, on sait que la chaîne de caractères `var1` correspond au chemin de la keystore. On remarque aussi que ce constructeur appelle la méthode `a` de la classe `mC` qui prend également en paramètre une chaîne de caractères `var6`. D'après le message d'avertissement précédemment obtenu, c'est cette méthode qui se charge d'accéder à la keystore.

En regardant attentivement les codes décompilés des différentes classes, on peut voir que sont appelées successivement les méthodes :

— Constructeur de `SimpleCryptoHandler` avec les paramètres `var1`
— `a` de `mC` avec les paramètres `var1` et `var6`
— Constructeur de `mC` avec les paramètres `var1` et `var6`
— `a` de `mD` avec les paramètres `var1` et `var6`

En regardant plus en détail la signature de cette dernière méthode, on peut voir que la chaîne de caractères `var6` est utilisée comme une clé. Il suffirait donc d'afficher le contenu de la variable `var6` pour obtenir la clé.

On copie donc dans notre programme principal le code du constructeur de la classe `SimpleCryptoHandler` afin de pouvoir l'instrumenter et retrouver la clé. En affichant le contenu de la chaîne de caractères `var6`, on retrouve la clé suivante : `aM#2uT)@e1*A`.

Nous avons donc réussi à retrouver la clé secrète présente dans le programme.

3 Fonctionnement global de la bibliothèque

3.1 Utilité des différentes classes

En étudiant en détail le code décompilé de certaines classes appelées par le programme, il est possible de deviner leur utilité dans le programme :

mz Pour un tableau de bits, associe les différents types des données encodées afin de pouvoir plus facilement les décoder.

mA Structure de données, garde en mémoire une chaîne de caractères et une `Map`.

mC Interface d'accès à la keystore.

mD Ouverture de la keystore, chiffrement et déchiffrement d'objets passés en paramètres des différentes méthodes.

mE Utilisée pour appeler de manière obfusquée certaines méthodes ou accéder à certaines variables.

CryptoConverter Interface servant d'intermédiaire entre la façade encapsulant la keystore et l'utilisateur. L'accès à la keystore se fait par un système de sessions.

SimpleCryptoHander Version simplifiée de l'interface présentée ci-dessus. Le chemin d'accès à la keystore est spécifié par l'utilisateur.

3.2 Algorithmes utilisés pour le chiffrement, le déchiffrement et l'aléatoire

Dans ce programme, plusieurs algorithmes sont utilisés pour chiffrer un objet Java à l'aide de la clé et le déchiffrer. Ces derniers sont explicités dans les différents messages d'erreur renvoyés par le programme.

Le principal algorithme mentionné pour le chiffrement et le déchiffrement des données est l'algorithme *AES (Advanced Encryption Standard)*, un algorithme de chiffrement symétrique très communément utilisé. Autrement dit, la clé utilisée pour chiffrer les données est la même que celle utilisée pour les déchiffrer.

Pour obtenir des valeurs aléatoires, deux algorithmes sont mentionnés dans les messages d'erreur : *IBMSecureRandom*, qui sera utilisé en priorité, et *SHA1PRNG*.