Java security- praca z kluczami i certyfikatami

Wstęp

Istnieje wiele przypadków użycia, w których konieczne jest wykorzystanie kluczy lub certyfikatów. Choć sama technologia może nie jest skomplikowana, to jednak jej użycie może okazać się kłopotliwe, szczególnie wtedy, gdy nie posiada się wiedzy o stojącej za nią wzorcach i rozwiązaniach. Poniższy materiał powstał celem wyjaśnienia wybranych zagadnień z tym związanych. Dużo więcej informacji można znaleźć w tutorialach opublikowanych w Internecie, np. `http://edu.pjwstk.edu.pl/wyklady/tbo/scb/lecture-10/lecture-10.html`.

Na początek warto wyjaśnić, że obsługa kluczy i certyfikatów na platformie Java ma swoją własną specyfikę. Ta specyfika zaczyna być widoczna już w samym sposobie przechowywania kluczy i certyfikatów. Otóż w Javie stosuje się do tego magazyny kluczy (ang. Java KeyStore, JKS). Magazyny te przechowują **razem** certyfikaty oraz klucze prywatne i publiczne.

Jeśli ktoś pamięta narzędzie `ssh-keygen` (stosowane do generowania kluczy ssh prywatnych i publicznych) to jego odpowiednikiem na platformie Java jest `keytool` (narzędzie znajdujące się w dystrybucji jdk). Jednak nie są to takie same narzędzia.

`keytool` operuje na magazynie kluczy i certyfikatów (do magazynu można dodać wiele kluczy i certyfikatów), natomiast `ssh-keygen` obsługuje tylko pojedynczą parę kluczy (składającą się z klucza prywatnego i publicznego).

Można o tym poczytać na stronach:

`https://www.ibm.com/support/knowledgecenter/en/SSS9FA_12.0.0/com.ibm.hod.doc/tutorials/ssh/ssh-pk03.html`

`https://commandlinefanatic.com/cgi-bin/showarticle.cgi?article=art049`

`keytool` uruchamia się z linii komend przekazując mu opcje (a może ich być wiele, co opisano na stronie:

`https://docs.oracle.com/javase/7/docs/technotes/tools/solaris/keytool.html`)

Niektóre z opcji, jeśli nie zostaną zdefiniowane, to przyjmują wartości domyślne. Poniżej wypisano ich przykłady:

`-alias "mykey"`

`-keyalg`
- `"DSA"` gdy użyto `-genkeypair`
- `"DES"` gdy użyto `-genseckey`

`-keysize`
- `2048` (gdy użyto `-genkeypair` oraz gdy `-keyalg` jest `"RSA"`)
- `1024` (gdy użyto `-genkeypair` oraz gdy `-keyalg` jest `"DSA"`)
- `256` (gdy użyto `-genkeypair` oraz gdy `-keyalg` jest `"EC"`)
- `56` (gdy użyto `-genseckey` oraz gdy `-keyalg` jest `"DES"`)
- `168` (gdy użyto `-genseckey` oraz gdy `-keyalg` jest `"DESede"`)

`-validity 90` (ważność certyfikatu wyrażona w dniach)

`-keystore .keystore` (plik o takiej nazwie w katalogu domowym użytkownika)

`-storetype "keystore.type"` (wartość tej właściowości pobrana z pliku z właściwościami bezpieczeństwa, która jest zwracana z metody statycznej `getDefaultType` dostępnej w klasie `java.security.KeyStore`)

`-file stdin` (czyli standardowe wejście w przypadku czytania), `stdout` (czyli standardowe wyjście w przypadku pisania)

`-protected false`

Podpisywanie plików jar oraz ich użycie

Scenariusze na stronach Oracle

Na stronach internetowych Oracle opisano scenariusze:

- podpisywania powstałego oprogramowania:
  `https://docs.oracle.com/javase/tutorial/security/toolsign/signer.html`
- wykorzystania podpisanego oprogramowania:
  `https://docs.oracle.com/javase/tutorial/security/toolsign/receiver.html`

Poniżej zaś przedstawiono scenariusz pozwalający na uruchomienie aplikacji korzystającej z zewnętrznej biblioteki (pliku jar) podpisanej wygenerowanym certyfikatem. Oparto go na przykładzie eclipsowych projektów `Biblioteka01` oraz `Aplikacja01`.

1. Generowanie kluczy

Pierwszym etapem opisywanego scenariusza jest wygenerowanie kluczy (i certyfikatu) za pomocą komendy:

```
$ keytool -genkeypair -keyalg rsa -keysize 2048 -validity 731 -alias tkubik -keystore tkubikkeys.jks -dname "CN=Tomasz Kubik"
```

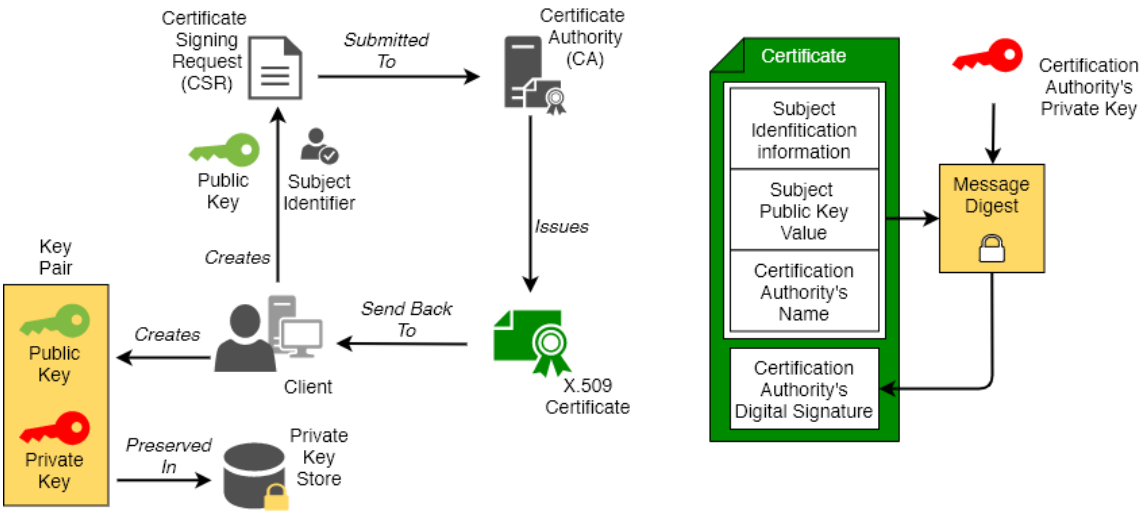
Powyższe wygeneruje klucze (prywatny i publiczny) oraz samodzielnie podpisany certyfikat (ang. *self-signed-certificate*) ważny przez 2 lata. W ogólności wystarcza on do przeprowadzania testów podczas realizacji ćwiczenia.

Oczywiście można podać nieco więcej informacji o właścicielu certyfikatu, np:

CN= Common Name (nazwa występująca w certyfikacie)
OU= Organizational Unit Name (nazwa jednostki organizacyjnej używającej certyfikatu, np.: IT department, sales)
O= Organization Name (pełna nazwa organizacji/firmy)
L= Locality Name (nazwa miejscowości, w której zlokalizowana jest organizacja/firma)
S= State/Province Name (pełna nazwa stanu/prowincji)
C= Country Name (dwuliterowy kod państwa)
E= Email address (adres poczty, nie jest konieczny do wygenerowania certyfikatu)

2. Podpisanie certyfikatu (krok opcjonalny)

Certyfikaty używane w środowiskach produkcyjnych powinny być podpisane przez odpowiednią instytucję. Poniższy schemat wyjaśnia procedurę uzyskania podpisanego certyfikatu: po wygenerowaniu kluczy i samodzielnie podpisanego certyfikatu należy wygenerować żądania podpisania certyfikatu. W żądaniu tym zamieszcza się klucz publiczny oraz dane instytucji bądź osoby, której certyfikat ma być podpisany. Instytucja certyfikująca po sprawdzeniu żądania przesyła zwrotnie podpisany certyfikat (pojawia się w nim informacja o tej instytucji oraz podpis cyfrowy wiadomości). Aby zweryfikować tak podpisany certyfikat należy wyliczyć funkcję skrótu wiadomości i porównać ją z tym, co jest w podpisie, odkodowując to publicznym kluczem instytucji certyfikującej. Taką weryfikację wykonują np. przeglądarki internetowe (gdy łączą się z serwisami posiadającymi wystawione certyfikaty).



W celu uzyskania podpisanego certyfikatu należy najpierw wygenerować żądanie podpisania certyfikatu (ang. *certificate signing request*). W przykładzie wywołania odpowiedniej komendy żądanie to ma zostać zapisane w pliku `tkubikcert.csr`:

```
$ keytool -certreq -alias tkubik -keystore tkubikkeys.jks > tkubikcert.csr
```

Po przesłaniu żądania do instytucji certyfikującej i otrzymaniu odpowiedzi z podpisanym certyfikatem należałoby go zaimportować do repozytorium kluczy. W przykładzie odpowiedniej komendy podpisany certyfikat znajduje się w pliku `tkubikcert.pem`:

```
$ keytool -import -alias tkubik -file tkubikcert.pem -keystore tkubikkeys.jks
```

Wykonanie powyższej komendy spowoduje nadpisanie niepodpisanego certyfikatu znajdującego się w magazynie kluczy.

Wystawianie (i utrzymywanie) certyfikatów odbywa się za opłatą. Do testowania nie potrzeba zabiegać o podpisanie samodzielnie wygenerowanego certyfikatu.

4. Wyeksportowanie certyfikatu z kluczem publicznym

Podpisana bibliteka (plik jar) powinna być dystrybuowana razem z certyfikatem zawierającym klucz publiczny. Bez niego klienci korzystający z biblioteki nie będą w stanie zweryfikować poprawności jej podpisu.

Aby wyeksportować klucz publiczny należy wydać następującą komendę:

```
$ keytool -export -keystore tkubikkeys.jks -alias tkubik -file tkubik_pub.cer
```

Klucz publiczny zostanie wyeksportowany do pliku `tkubik_pub.cer`, który powinien towarzyszyć dystrybuowanemu plikowi jar biblioteki. Klienci korzystający z biblioteki będą importować ten certyfikat do własnego magazynu kluczy.

W szczególności można zaimportować ten certyfikat do `cacerts` - domyślnego magazynu certyfikatów platfomy Java:

```
keytool -import -trustcacerts -keystore  <ścieżka do jdk>\lib\security\cacerts -alias tkubik -file tkubik_pub.cer
```

Certyfikat zaimportowany można też z magazynu usunąć:

```
keytool -delete -alias tkubik -keystore cacerts
```

5. Podpisanie pliku jar

Podpisanie biblioteki może odbyć się za pomocą narzędzia `jarsigner`:

```
$ jarsigner -keystore tkubikkeys.jks -storepass <hasło magazynu> -keypass <hasło klucza prywatnego> -signedjar sbilioteka01.jar biblioteka01.jar tkubik
```

gdzie:

`biblioteka01.jar` - oryginalna biblioteka (niepodpisana), `sbiblioteka01.jar` - wynikowa biblioteka (podpisana), `tkubik` - alias wpisu w magazynie kluczy (by było wiadomo, jakim certyfikatem podpisać bibliotekę)

Przy okazji można zweryfikować, czy faktycznie plik jar został podpisany komendą:

```
$ jarsigner -verify -certs -verbose sbiblioteka01.jar
```

W rozważanym przykładzie wynikiem wykonania komendy będzie:

```
s       196 Wed Apr 22 03:16:32 CEST 2020 META-INF/MANIFEST.MF

      >>> Signer
      X.509, CN=Tomasz Kubik
      [trusted certificate]

        332 Wed Apr 22 03:16:32 CEST 2020 META-INF/TKUBIK.SF
       1115 Wed Apr 22 03:16:32 CEST 2020 META-INF/TKUBIK.RSA
          0 Wed Apr 22 01:37:16 CEST 2020 pl/
          0 Wed Apr 22 01:37:16 CEST 2020 pl/edu/
          0 Wed Apr 22 01:37:16 CEST 2020 pl/edu/pwr/
          0 Wed Apr 22 01:37:16 CEST 2020 pl/edu/pwr/example/
sm     3229 Wed Apr 22 02:48:30 CEST 2020 pl/edu/pwr/example/Encoder.class

      >>> Signer
      X.509, CN=Tomasz Kubik
      [trusted certificate]
```

```
  s = signature was verified
  m = entry is listed in manifest
  k = at least one certificate was found in keystore

- Signed by "CN=Tomasz Kubik"
    Digest algorithm: SHA-256
    Signature algorithm: SHA256withRSA, 2048-bit key

jar verified.

Warning:
This jar contains entries whose signer certificate is self-signed.
```

6. Wykorzystanie podpisanej biblioteki

Ktoś, kto chciałby skorzystać z podpisanej biblioteki powinien dołączyć ją do projektu. Samo dołączenie do projektu pozwala kompilować kod. Dopóki nie ma zdefiniowanego menadżera bezpieczeństwa, to ten kod daje się również wykonywać. Ale nie zawsze. Jeśli dana biblioteka (podpisany jar) ma takie same pakiety jak aplikacja, która z niej korzysta, to przy próbie uruchomienia pojawia się wyjątek bezpieczeństwa:

```
Exception in thread "main" java.lang.SecurityException: class
"pl.edu.pwr.example.Encoder"'s signer information does not match signer information of
other classes in the same package
      at java.base/java.lang.ClassLoader.checkCerts(ClassLoader.java:1150)
      at java.base/java.lang.ClassLoader.preDefineClass(ClassLoader.java:905)
      at java.base/java.lang.ClassLoader.defineClass(ClassLoader.java:1014)
      at
java.base/java.security.SecureClassLoader.defineClass(SecureClassLoader.java:174)
      at
java.base/jdk.internal.loader.BuiltinClassLoader.defineClass(BuiltinClassLoader.java:80
2)
      at
java.base/jdk.internal.loader.BuiltinClassLoader.findClassOnClassPathOrNull(BuiltinClas
sLoader.java:700)
      at
java.base/jdk.internal.loader.BuiltinClassLoader.loadClassOrNull(BuiltinClassLoader.jav
a:623)
      at
java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.java:581)
      at
java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoaders.java:1
78)
      at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:521)
      at pl.edu.pwr.example.Main.main(Main.java:30)
```

W przypadku różnych nazw pakietów w bibliotece i aplikacji żaden wyjątek się nie pojawi - dopóki nie zostanie zainstalowany menadżer bezpieczeństwa.

Jeśli aplikacja korzysta z menadżera bezpieczeństwa, to by wszystko zadziałało należy podjąć kolejne kroki.

Na początek należy zaimportować certyfikat towarzyszący podpisanej bibliotece do własnego magazynu kluczy. Wcześniej pokazano już, jak taki certyfikat zaimportować do domyślnego magazynu platformy Java. Jeśli podpisana biblioteka jest testowana na komputerze jej autora, to wtedy, oczywiście, nie trzeba importować certyfikatu do magazynu - w tym magazynie jest już przecież certyfikat (z niego był eksportowany).

W szczególności certyfikat można zaimportować do magazynu zaufanych certyfikatów:

```
$ keytool -import -trustcacerts -keystore [keystore_filename] -alias [alias_name] -file
[cert_file]
```

Jeśli ktoś zapomniał aliasu do certyfikatu, to można go odszukać przeglądając magazyn:

```
$ keytool -list -v -keystore [keystore name]
```

Następnie należy zdefiniować pozwolenia w pliku polityki (więcej informacji na temat pozwoleń można znaleźć pod adresem: `https://docs.oracle.com/en/java/javase/11/security/permissions-jdk1.html`).

Przykładowo plik taki może mieć postać jak niżej:

```
/* AUTOMATICALLY GENERATED ON Wed Apr 22 03:11:22 CEST 2020*/
/* DO NOT EDIT */

keystore "file:/E:/Dydaktyka/JavaWyk2017/Programy/tkubikkeys.jks", "JKS", "SUN";

keystorePasswordURL "file:/E:/Dydaktyka/JavaWyk2017/Programy/keystore.pass";

grant signedBy "tkubik",  codeBase
"file:/E:/Dydaktyka/JavaWyk2017/Programy/sbiblioteka01.jar" {
  permission java.security.AllPermission;
  permission java.io.FilePermission "<<ALL FILES>>", "read, write, delete, execute";
};

grant codeBase "file:/E:/Dydaktyka/JavaWyk2017/Programy/Aplikacja01/bin/-"{
  permission java.security.AllPermission;
  permission java.io.FilePermission "<<ALL FILES>>", "read, write, delete, execute";
};
```

Proszę zwrócić uwagę na składnię kolejnych sekcji grant. Wskazano w nich, jakie źródła kodu mają jakie zezwolenia. Jak widać uwzględniono położenie skompilowanych klas aplikacji oraz położenie podpisanej biblioteki. Gdyby aplikacja została wyeksportowana do pliku jar, plik polityki należałoby zmienić.

W pliku polityki wskazano na położenie pliku `keystore.pass` z hasłem do magazynu kluczy użytkownika, do którego zaimportowano certyfikat. Plik ten można nazwać dowolnie i umieścić w katalogu dostępnym tylko dla użytkownika. Jego zawartością ma być jedna linijka zawierająca wspomniane hasło.

Pliki polityki można generować ręcznie, ale czasem dobrze jest użyć narzędzia `policytool`. Narzędzie to było dystrybuowanej w jdk1.8, w nowszych wersjach jdk już się, niestety, nie pojawia.

W plikach polityki można zamieścić sporo informacji. Proszę spojrzeć na kolejne przykłady, w których posłużono się zmiennymi systemowymi (przykłady pochodzą ze strony: `http://edu.pjwstk.edu.pl/wyklady/tbo/scb/lecture-02/lecture-02-content.html`)

```
keystore "${user.home}${/}.keystore";

grant codeBase "http://pjwstk.edu.pl/" {
   permission java.io.FilePermission "/tmp", "read";
   permission java.lang.RuntimePermission "queuePrintJob";
};

grant signedBy "edek", codeBase "file:${java.home}/lib/ext/." {
   permission java.security.AllPermission;
};

grant signedBy "edek", codeBase "file:${java.class.path}/." {
   permission java.net.SocketPermission "*:1024-",
      "accept, connect, listen, resolve";
};

grant {
   permission java.util.Permission "java.version", "read";
};
```

Po tym wszystkim można już uruchomić program z odpowiednimi parametrami, najlepiej w katalogu projektu aplikacji (w przykładzie jest to główny katalog projektu `Aplikacja01`). W parametrach przekazuje się menadżera bezpieczeństwa, położenie pliku polityki oraz wskazuje `classpath`:

```
$ java -Djava.security.manager -Djava.security.policy=tkubik.policy -classpath
"./bin;../sbilioteka01.jar" pl.edu.pwr.example1.Main
```

Oczywiście parametry te można wpisać w konfigurację uruchomieniową wybranego IDE.

Próba uruchomienia aplikacji z menadżerem bezpieczeństwa bez wskazania położenia pliku polityki zakończy się wyrzuceniem wyjątku:

```
Exception in thread "main" java.security.AccessControlException: access denied
("java.io.FilePermission" "logFileIn.txt" "write")
      at
java.base/java.security.AccessControlContext.checkPermission(AccessControlContext.java:
472)
      at
java.base/java.security.AccessController.checkPermission(AccessController.java:895)
      at java.base/java.lang.SecurityManager.checkPermission(SecurityManager.java:322)
      at java.base/java.lang.SecurityManager.checkWrite(SecurityManager.java:752)
      at
java.base/sun.nio.fs.WindowsChannelFactory.open(WindowsChannelFactory.java:301)
      at
java.base/sun.nio.fs.WindowsChannelFactory.newFileChannel(WindowsChannelFactory.java:16
8)
      at
java.base/sun.nio.fs.WindowsFileSystemProvider.newByteChannel(WindowsFileSystemProvider
.java:226)
      at
java.base/java.nio.file.spi.FileSystemProvider.newOutputStream(FileSystemProvider.java:
478)
      at java.base/java.nio.file.Files.newOutputStream(Files.java:219)
      at pl.edu.pwr.example1.Main.main(Main.java:26)
```

Ciekawostki:

symlink (chodzi o to, że przy kilku wersjach JDK zainstalowanych na jednym komputerze wydawanie komend java z konsoli może być kłopotliwe ze względu na istnienie symbolicznych linków w ścieżce systemowej: C:\ProgramData\Oracle\Java\javapath)

```
https://stackoverflow.com/questions/26864662/the-system-cannot-find-the-file-c-
programdata-oracle-java-javapath-java-exe
```

```
https://www.howtogeek.com/howto/16226/complete-guide-to-symbolic-links-symlinks-on-
windows-or-linux/
```

```
https://stackoverflow.com/questions/10188485/windows-symbolic-link-target
```

JShell (shell do testowania fragmentów kodu java)

```
https://docs.oracle.com/en/java/javase/12/jshell/introduction-jshell.html#GUID-
DA9FA090-7015-4F30-BBD0-5F6ED0EBDF91
```

Materiały:

```
https://www.sslshopper.com/article-how-to-create-a-self-signed-certificate-using-java-
keytool.html
```

```
https://www.sslshopper.com/article-most-common-java-keytool-keystore-commands.html
```

```
keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore.jks -storepass
password -validity 360 -keysize 2048
```

```
https://docs.oracle.com/javase/1.5.0/docs/tooldocs/solaris/keytool.html#importCmd
```

```
https://www.certum.pl/pl/upload_module/wysiwyg/certum/instrukcje/Instrukcja_podpisywani
a_kodu_przy_uzyciu_narzedzi_signtool_i_jarsigner.pdf
```

```
https://dzone.com/articles/signing-and-verifying-a-standalone-jar
```

Co należy zrobić, gdy zostanie wyświetlony monit zabezpieczeń oprogramowania Java?

https://www.java.com/pl/download/help/appsecuritydialogs.xml

Java SE Security

https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html

Java Platform Standard Edition 8 Documentation

https://docs.oracle.com/javase/8/docs/index.html

https://stackoverflow.com/questions/17187520/signing-jar-file

https://docs.oracle.com/en/java/javase/12/

https://docs.oracle.com/javase/1.5.0/docs/tooldocs/solaris/keytool.html#importCmd

Permission policy file

https://docs.oracle.com/en/java/javase/11/security/java-se-platform-security-architecture.html#GUID-F5270083-CCAC-49E0-ACAC-50176FBFDD97

Jar signing in Maven

https://blog.frankel.ch/jvm-security/2/

https://www.digicert.com/code-signing/java-code-signing-guide.htm

Podpisywanie kodu Java: Generowanie żądania CSR

https://pl.godaddy.com/help/podpisywanie-kodu-java-generowanie-zadania-csr-4780

https://docs.oracle.com/en/java/javase/11/tools/jarsigner.html

https://docs.oracle.com/javase/tutorial/security/sigcert/index.html

https://commandlinefanatic.com/cgi-bin/showarticle.cgi?article=art049