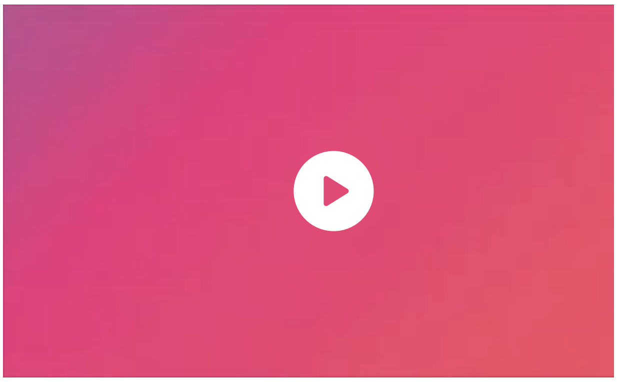
(/)

How to Read PEM File to Get Public and Private Keys



Last updated: June 3, 2022

> Written by: Catalin Burcea (https://www.baeldung.com/author/catalin-burcea)

Java (https://www.baeldung.com/category/java) •

Security (https://www.baeldung.com/category/security)

1. Overview

In public-key cryptography, also known as asymmetric cryptography (/cs/symmetric-vs-asymmetric-cryptography), the encryption mechanism relies upon two related keys, a public key and a private key. The public key is used to encrypt the message, while only the owner of the private key can decrypt the message.

In this tutorial, we'll learn how to read public and private keys from a PEM file.

First, we'll study some important concepts around public-key cryptography. Then we'll learn how to read PEM files using pure Java.

Finally, we'll explore the BouncyCastle (/java-bouncy-castle) library as an alternate approach.

2. Concepts

Before we start, let's discuss some key concepts.

X.509 is a standard defining the format of public-key certificates. So this format describes a public key, among other information.

DER is the most popular encoding format to store data, like X.509 certificates, and PKCS8 private keys in files. It's a binary encoding, and the resulting content can't be viewed with a text editor.

PKCS8 is a standard syntax for storing private key information. The private key can be optionally encrypted using a symmetric algorithm.

Not only can RSA private keys be handled by this standard, but also other algorithms. The PKCS8 private keys are typically exchanged through the PEM encoding format.

PEM is a base-64 encoding mechanism of a DER certificate. PEM can also encode other kinds of data, such as public/private keys and certificate requests.

A PEM file also contains a header and footer describing the type of encoded data:

```
-----BEGIN PUBLIC KEY-----
...Base64 encoding of the DER encoded certificate...
-----END PUBLIC KEY-----
```

3. Using Pure Java

3.1. Read PEM Data From a File

Let's start by reading the PEM file, and storing its content into a string:

```
String key = new String(Files.readAllBytes(file.toPath()), Charset.defaultCharset());
```

3.2. Get Public Key From PEM String

Now we'll build a utility method that gets the public key from the PEM encoded string:

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAsjtGIk8SxD+OEiBpP2/T
JUAF0upwuKGMk6wH8Rwov88VvzJrVm2NCticTk5FUg+UG5r8JArrV4tJPRHQyvqK
wF4NiksuvOjv3HyIf4oaOhZjT8hDne1Bfv+cFqZJ61Gk0MjANh/T5q9vxER/7TdU
NHKpoRV+NVlKN5bEU/NQ5FQjVXicfswxh6Y6fl2PIFqT2CfjD+FkBPU1iT9qyJYH
A38IRvwNtcitFgCeZwdGPoxiPPh1WHY8VxpUVBv/2JsUtrB/rAIbGqZoxAIWvijJ
Pe9o1TY3VlOzk9ASZ1AeatvOir+iDVJ5OpKmLnzc46QgGPUsjIyo6Sje9dxpGtoG
QQIDAQAB
-----END PUBLIC KEY-----
```

Let's suppose we receive a *File* as a parameter:

```
public static RSAPublicKey readPublicKey(File file) throws
Exception {
    String key = new String(Files.readAllBytes(file.toPath()),
Charset.defaultCharset());

    String publicKeyPEM = key
      .replace("-----BEGIN PUBLIC KEY-----", "")
      .replaceAll(System.lineSeparator(), "")
      .replace("-----END PUBLIC KEY-----", "");

    byte[] encoded = Base64.decodeBase64(publicKeyPEM);

    KeyFactory keyFactory = KeyFactory.getInstance("RSA");
    X509EncodedKeySpec keySpec = new X509EncodedKeySpec(encoded);
    return (RSAPublicKey) keyFactory.generatePublic(keySpec);
}
```

As we can see, first we need to remove the header, the footer, and the new lines as well. Then we need to decode the Base64-encoded string into its corresponding binary format.

Next, we need to load the result into a key specification class able to handle public key material. In this case, we'll use the *X509EncodedKeySpec* class.

Finally, we can generate a public key object from the specification using the *KeyFactory* class.

3.3. Get Private Key From PEM String

Now that we know how to read a public key, the algorithm to read a private key is very similar.

We'll use a PEM encoded private key in PKCS8 format. Let's see what the header and footer look like:

```
-----BEGIN PRIVATE KEY-----
...Base64 encoded key...
-----END PRIVATE KEY-----
```

As we learned previously, we need a class able to handle PKCS8 key material. The *PKCS8EncodedKeySpec* class fills that role.

So let’s see the algorithm:

```
public RSAPrivateKey readPrivateKey(File file) throws Exception {
    String key = new String(Files.readAllBytes(file.toPath()),
Charset.defaultCharset());

    String privateKeyPEM = key
      .replace("-----BEGIN PRIVATE KEY-----", "")
      .replaceAll(System.lineSeparator(), "")
      .replace("-----END PRIVATE KEY-----", "");

    byte[] encoded = Base64.decodeBase64(privateKeyPEM);

    KeyFactory keyFactory = KeyFactory.getInstance("RSA");
    PKCS8EncodedKeySpec keySpec = new PKCS8EncodedKeySpec(encoded);
    return (RSAPrivateKey) keyFactory.generatePrivate(keySpec);
}
```

4. Using BouncyCastle Library

4.1. Read Public Key

We'll explore the BouncyCastle library, and see how we can use it as an alternative to the pure Java implementation.

Let's get the public key:

```
public RSAPublicKey readPublicKey(File file) throws Exception {
    KeyFactory factory = KeyFactory.getInstance("RSA");

    try (FileReader keyReader = new FileReader(file);
      PemReader pemReader = new PemReader(keyReader)) {

        PemObject pemObject = pemReader.readPemObject();
        byte[] content = pemObject.getContent();
        X509EncodedKeySpec pubKeySpec = new
X509EncodedKeySpec(content);
        return (RSAPublicKey) factory.generatePublic(pubKeySpec);
    }
}
```

There are a few important classes that we need to be aware of when using BouncyCastle:

- *PemReader* – takes a *Reader* as a parameter and parses its content. **It removes the unnecessary headers** and **decodes the underlying Base64 PEM data into a binary format.**
- *PemObject* – **stores the result generated by the *PemReader***

Let's see another approach that wraps Java's classes (*X509EncodedKeySpec, KeyFactory*) into BouncyCastle's own class (*JcaPEMKeyConverter*):

```
public RSAPublicKey readPublicKeySecondApproach(File file) throws
IOException {
    try (FileReader keyReader = new FileReader(file)) {
        PEMParser pemParser = new PEMParser(keyReader);
        JcaPEMKeyConverter converter = new JcaPEMKeyConverter();
        SubjectPublicKeyInfo publicKeyInfo =
SubjectPublicKeyInfo.getInstance(pemParser.readObject());
        return (RSAPublicKey)
converter.getPublicKey(publicKeyInfo);
    }
}
```

4.2. Read Private Key

Now we'll see two examples that are very similar to the ones shown above.

In the first example, we just need to replace the *X509EncodedKeySpec* class with the *PKCS8EncodedKeySpec* class, and return a *RSAPrivateKey* object instead of a *RSAPublicKey*:

```
public RSAPrivateKey readPrivateKey(File file) throws Exception {
    KeyFactory factory = KeyFactory.getInstance("RSA");

    try (FileReader keyReader = new FileReader(file);
      PemReader pemReader = new PemReader(keyReader)) {

        PemObject pemObject = pemReader.readPemObject();
        byte[] content = pemObject.getContent();
        PKCS8EncodedKeySpec privKeySpec = new
PKCS8EncodedKeySpec(content);
        return (RSAPrivateKey)
factory.generatePrivate(privKeySpec);
    }
}
```

Now let's rework the second approach from the previous section a bit in order to read a private key:

```
public RSAPrivateKey readPrivateKeySecondApproach(File file) throws
IOException {
    try (FileReader keyReader = new FileReader(file)) {

        PEMParser pemParser = new PEMParser(keyReader);
        JcaPEMKeyConverter converter = new JcaPEMKeyConverter();
        PrivateKeyInfo privateKeyInfo =
PrivateKeyInfo.getInstance(pemParser.readObject());

        return (RSAPrivateKey)
converter.getPrivateKey(privateKeyInfo);
    }
}
```

As we can see, we just replaced *SubjectPublicKeyInfo* with *PrivateKeyInfo* and *RSAPublicKey* with *RSAPrivateKey*.

4.3. Advantages

There are a couple of advantages provided by the BouncyCastle library.

One advantage is that **we don’t need to manually skip or remove the header and footer.** Another is that **we’re not responsible for the Base64 decoding,** either. Therefore, we can write less error-prone code with BouncyCastle.

Moreover, the **BouncyCastle library supports the PKCS1 format** as well. Despite the fact that PKCS1 is also a popular format used to store cryptographic keys (only RSA keys), Java doesn't support it on its own.

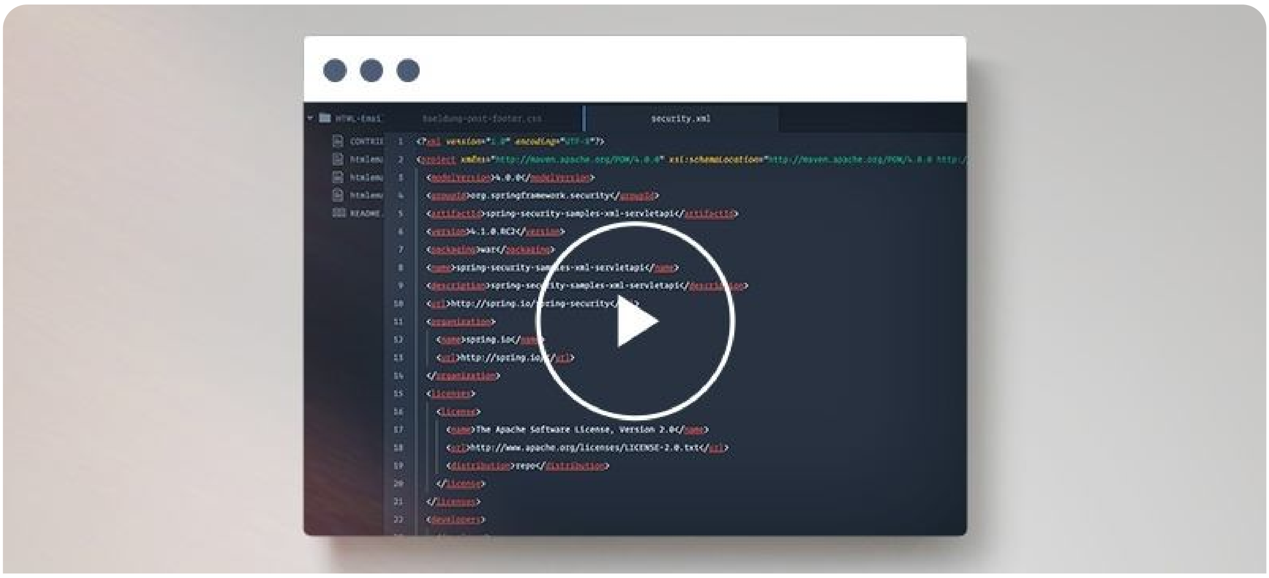
5. Conclusion

In this article, we learned how to read public and private keys from PEM files.

First, we studied a few key concepts around public-key cryptography. Then we saw how to read public and private keys using pure Java.

Finally, we explored the BouncyCastle library and discovered it's a good alternative, since it provides a few advantages compared to the pure Java implementation.

The full source code for both the Java (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-security-2) and BouncyCastle (https://github.com/eugenp/tutorials/tree/master/libraries-security) approaches is available over on GitHub.



```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:/
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-samples-xml-servletapi</artifactId>
  <version>4.1.0.RC2</version>
  <packaging>war</packaging>
  <name>spring-security-samples-xml-servletapi</name>
  <description>spring-security-samples-xml-servletapi</description>
  <url>http://spring.io/spring-security</url>
  <organization>
    <name>spring.io</name>
    <url>http://spring.io/</url>
  </organization>
  <licenses>
    <license>
      <name>The Apache Software License, Version 2.0</name>
      <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
      <distribution>repo</distribution>
    </license>
  </licenses>
  <developers>
```

Comments are closed on this article!

COURSES

ALL COURSES (/ALL-COURSES)

ALL BULK COURSES (/ALL-BULK-COURSES)

ALL BULK TEAM COURSES (/ALL-BULK-TEAM-COURSES)

THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)

JACKSON JSON TUTORIAL (/JACKSON)

APACHE HTTPCLIENT TUTORIAL (/HTTPCLIENT-GUIDE)

REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)

SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)

SECURITY WITH SPRING (/SECURITY-SPRING)

SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)

ABOUT

ABOUT BAELDUNG (/ABOUT)

THE FULL ARCHIVE (/FULL_ARCHIVE)

EDITORS (/EDITORS)

JOBS (/TAG/ACTIVE-JOB/)

OUR PARTNERS (/PARTNERS)
PARTNER WITH BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)