Javamex Home
SecureRandom
Java performance topics
Synchronization and concurrency
Java data compression

Java tutorials home ▶ Java cryptography ▶ Encryption intro ▶ Keys ▶ Symmetric encryption ▶ AES/block ciphers ▶ Block modes (ECB, CTR, OFB) ▶ Asymmetric encryption ▶ RSA in Java ▶ Comparison of algorithms ▶ Key sizes ▶ Hash functions

Search this site: Search

Like 31 Tweet G+1 5 111

RSA encryption in Java

Popular topics:
- What RSA key length?
- RSA encryption in Java
- RSA algorithm and security
- Symmetric encryption
- AES and block ciphers
- Encryption key lengths
- Reading encrypted ZIP files in Java
- Java cryptography

We introduced the notion of asymmetric encryption, in which a key needed to *encrypt* data is made public, but the corresponding key needed to *decrypt* it is kept private, for example in a file on the server to which clients connect. In principle, such a system solves the problem of how to send a temporary encryption key securely to the server when opening a secure connection*. A very common asymmetric encryption system is **RSA**, named after inventors Rivest, Shamir & Adleman.

*If we just used a symmetric encryption system such as AES on its own, we'd have the problem of how the two parties agree on the key in advance. This is sometimes possible, but usually, we want to transmit a temporary key at the moment of connecting. Generally, we use an asymmetric encryption system such as RSA to transmit the secret key, then for the rest of the conversation, use a regular symmetric encryption system using that secret key.

Basic algorithm and terminology

RSA encryption and decryption are essentially mathematical operations. They are what are termed *exponentiation*, *modulo* a particular number. Because of this, **RSA keys** actually consist of numbers involved in this calculation, as follows:

- the **public key** consists of the **modulus** and a **public exponent**;
- the **private key** consists of that same **modulus** plus a **private exponent**.

Those interested in more details of the RSA algorithm should see this separate page.

Creating an RSA key pair in Java

As a one-off process, we need to generate an RSA key pair that from then on, we'll use for all conversations between our clients and server.

Creating an RSA key pair essentially consists of picking a modulus, which is based on two random primes intended to be unique to that key pair, picking a public exponent (in practice, the common exponent 65537 is often used), then calculating the corresponding private exponent given the modulus and public exponent. Java provides the `KeyPairGenerator` class for performing this task. The essential idea is as follows:

- we create an **instance of `KeyPairGenerator`** suitable for generating RSA keys;
- we **initialise** the generator, telling it the **bit length of the modulus** that we require (see below);
- we call `genKeyPair()`, which eventually returns a `KeyPair` object;
- we call `getPublic()` and `getPrivate()` on the latter to pull out the public and private keys.

The code looks as follows:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
kpg.initialize(2048);
KeyPair kp = kpg.genKeyPair();
Key publicKey = kp.getPublic();
Key privateKey = kp.getPrivate();
```

Notice that we specify a **key length** of 2048 bits. Common values are 1024 or 2048. Choosing an RSA key length is a tradeoff between security and performance.

We now have two `Key` objects. Whilst we could immediately use these to start encrypting and decrypting, in most practical uses, we want to *store* these two keys for later use. For that, we use a `KeyFactory` to pull out the components (modulus and exponents) of the keys.

Saving the public and private key

In practice, we need to store the public and private keys somewhere. Typically, the private key will be placed on our server, and the public key distributed to clients. To store the key, we simply need to pull out the modulus and the public and private exponents, then write these numbers to some file (or put in whatever convenient place).

The `Key` interface allows us to pretend for a second that we don't need to worry about the algorithm-specific details of keys. But unfortunately, in practice we do. So there also exist "key specification" classes— `RSAPublicKeySpec` and `RSAPrivateKeySpec` in this case— with transparent methods for pulling out the parameters that make up the key. Then, a `KeyFactory` allows us to translate between `Key`s and their corresponding specification. It's a bit clumsy, but the code ends up as follows:

```
KeyFactory fact = KeyFactory.getInstance("RSA");
RSAPublicKeySpec pub = fact.getKeySpec(kp.getPublic(),
  RSAPublicKeySpec.class);
RSAPrivateKeySpec priv = fact.getKeySpec(kp.getPrivate(),
  RSAPrivateKeySpec.class);

saveToFile("public.key", pub.getModulus(),
  pub.getPublicExponent());
saveToFile("private.key", priv.getModulus(),
  priv.getPrivateExponent());
```

To save the moduli and exponents to file, we can just use boring old serialisation, since the modulus and exponents are just `BigInteger` objects:

```
public void saveToFile(String fileName,
  BigInteger mod, BigInteger exp) throws IOException {
  ObjectOutputStream oout = new ObjectOutputStream(
    new BufferedOutputStream(new FileOutputStream(fileName)));
  try {
    oout.writeObject(mod);
    oout.writeObject(exp);
  } catch (Exception e) {
    throw new IOException("Unexpected error", e);
  } finally {
    oout.close();
  }
}
```

In our example, we end up with two files: `public.key`, which is distributed with out clients (it can be packed into the jar, or whatever); meanwhile, `private.key`, is kept secret on our server. Of course, we needn't even bother saving the keys to a file: they're just numbers, and we could embed them as constant strings in our code, then pass those strings to the constructor of `BigInteger`. Saving the keys to file simply makes it a bit easier to manage keys in some cases (for example, we might change keys periodically and distribute new key files to the clients; distributing a few bytes is often more practical than distributing the entire code base).

Now we've got a mechanism to generate a key pair and save those keys for the future, we're ready to consider how to actually perform RSA encryption/decryption, reading in the key files we generated.

Like 31 Tweet 111

3 Comments **Javamex Programming Tutorials** 1 **Login**

Recommend Share Sort by Best

Join the discussion…

Markus Schurtenberger • 3 years ago
Thank you for the very good tutorial. What bothers me a little is the described mode of saving to file. Could you also show how go generate PEM-style key files that can be used on other systems, too. There, the keys are somehof base64-encoded and bordered with a "----- Begin of RSA private key-----" or similar.
13 • Reply • Share ›

Sharat • 3 years ago
Where is the jar file present for this?
11 • Reply • Share ›

Leonardo Hernández • a year ago
Nice, this works fine.
• Reply • Share ›