Commons Security

Encryption done easy

Avec112

Version 1.0-SNAPSHOT, 2025-10-21

Table of Contents

Overview

Commons Security is a Java library with several encryption features for `Java 11` and higher build on top of `JCA/JCE` with `BouncyCastle` as provider. The library hides the implementation behind a simple and easy to use API. Typically a simplified API requires that some decisions has been made already, like which algorithm to use, encryption strength, etc. However do not worry, look behind the simplified API and you will find more configuration options.

Features

- **Asymmetric encryption** and decryption with *public key* and *private key*. `RSA` is supported.
- **Symmetric encryption** and decryption with one (same) key. `AES/CTR` and `AES/GCM` is supported.
- **Hybrid encryption** combines RSA and AES for secure and efficient encryption of large data. RSA encrypts the AES key, while AES encrypts the actual data.
- **Shamir's Secret Sharing** is a way to split a secret into shares. The secret can be recreated by putting the minimum required amount of shares back together.
- **Password Encoder** uses proven algorithms for hashing passwords in a safe and secure way.
- **Digest** or message digest is the result of hashing and is a way to integrity check our data. A typical use case could be to compare content (hashed) with a stored hash to verify if the data has been altered.
- Digital **Signature** and **verification** using RSA keypair and modern `RSASSA-PSS` padding. The user signs data with their *private key* and others verify the signature with the originator's *public key*.

You may use the simple delegate class `CryptoUtils` or you can access classes like `RsaCipher`, `AesCipher`, `Shamir`, etc. directly. The latter might give you more options. Implementation alternatives are shown in examples below.

The `CryptoUtils` facade provides convenient methods for:

- **AES encryption/decryption** - `aesEncrypt()` / `aesDecrypt()`
- **RSA encryption/decryption** - `rsaEncrypt()` / `rsaDecrypt()`
- **Hybrid encryption/decryption** - `hybridEncrypt()` / `hybridDecrypt()`
- **Shamir's Secret Sharing** - `getShamirShares()` / `getShamirSecret()`
- **Digest (hashing)** - `digest()` / `base64Digest()` / `hexDigest()`
- **Digital signatures** - `sign()` / `verify()`

Some common classes

This is some common classes used when working with *Symmetric* and *Asymmetric encryption* but also *Password Encoding*.

- `PlainText` is a placeholder for unencrypted text.
  `PlainText plainText = new PlainText("My secret plaintext");`
- `CipherText` is a placeholder for encrypted text.

`CipherText cipherText = new CipherText("ymEIVhbBPhWAIzDx7MalbeLoccwnw=");`

- `Password` is a placeholder for a string (the password) used for either encryption or decryption.
  `Password password = new Password("SecretPassword123!");`
- `PublicKey` holds the *public key* derived from `java.security.KeyPair` and can be used for encryption or signature verification.
  `PublicKey publicKey = keyPair.getPublicKey();`
- `PrivateKey` holds the *private key* derived from `java.security.KeyPair` and can be used for decryption or for creating a signature.
  `PrivateKey privateKey = keyPair.getPrivateKey();`

RSA asymmetric encryption

RSA[1] (Rivest–Shamir–Adleman) is a *public-key* cryptosystem that is widely used for secure data transmission. In a *public-key* cryptosystem, the encryption key is public and distinct from the decryption key, which is kept secret (private).

> It involves the use of two mathematically related keys. The public key (the one that's known to everybody) and the private key (which is only known by you) are required for encrypting and decrypting the message. The public key can be derived from the private key but not the other way.

Asymmetric encryption use a *public key* for encryption and a *private* key for decryption. Useful when you want to share secrets with others. You may encrypt a file with someone's *public key*, so they will be able to decrypt with their matching *private key*.

Used to share small files or text with other users. For large content (more than 117 bytes) use AES symmetric encryption. The AES secret key can later be shared by using *Public Key Encryption* like RSA encryption or *Diffie-Hellman*[2].

RSA KeyPair

RSA requires a *private key* and a matching *public key*. Supported key sizes (encryption strength) are `2048`, `3072` (default) and `4096`.

A key size of `3072` bit or higher is recommended for strong security. Key size `1024` is not supported.

Any secret encrypted with a *public key* can only be decrypted with the associated *private key*.

Using `KeyUtils` *to generate a* `4096` *bit* `KeyPair`

```
KeyPair keyPair = KeyUtils.generateKeyPair4096();
```

Alternative

```
RsaCipher rsaCipher = new RsaCipher();
KeyPair keyPair = rsaCipher.generateKeyPair(KeySize.BIT_4096);
```

RSA Encryption and Decryption

RSA encryption and decryption using `CryptoUtils`

```
KeyPair keyPair = CryptoUtils.generateKeyPair4096();
CipherText cipherText = CryptoUtils.rsaEncrypt("My secret", keyPair.getPublicKey());
PlainText plainText = CryptoUtils.rsaDecrypt(cipherText, keyPair.getPrivateKey());
```

Alternative

```
RsaCipher rsaCipher = new RsaCipher();
KeyPair keyPair = rsaCipher.generateKeyPair(KeySize.BIT_4096);
CipherText cipherText = rsaCipher.encrypt("My secret", keyPair.getPublicKey());
PlainText plainText = rsaCipher.decrypt(cipherText, keyPair.getPrivateKey());
```

AES symmetric encryption

The Advanced Encryption Standard (AES)[3], also known by its original name Rijndael is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001.

> There's a single shared key that's used for encryption and decryption.

Useful for encryption larger files compared to asymmetric encryption. A symmetric key can be shared (distributed) by using *RSA public key encryption*, *Diffie-Hellman* or even *Sharmir's Secret Sharing* if needed.

Modes supported: `GCM`, `CTR`
Strength supported: `128`, `192`, `256`

AES Encryption and Decryption

AES encryption and decryption using `CryptoUtils` *with* `GCM`@`256`*-bit*

```
Password password = new Password("SecretPassword123!");
CipherText cipherText = CryptoUtils.aesEncrypt("Secret text", password);
PlainText plainText = CryptoUtils.aesDecrypt(cipherText, password);
```

Alternative (here as `CTR`@`192`-bit)

```
Password password = new Password("SecretPassword123!");
AesCipher aesCipher = new AesCipher(EncryptionMode.CTR, EncryptionStrength.BIT_192);
h cipherText = aesCipher.encrypt("Secret text", password);
PlainText plainText = aesCipher.decrypt(cipherText, password);
```

Hybrid Encryption

Hybrid encryption combines the strengths of both asymmetric (RSA) and symmetric (AES) encryption. RSA is used to securely encrypt a randomly generated AES key, while AES encrypts the actual data payload. This approach provides the security of RSA with the performance and capacity of AES.

> RSA can only encrypt small amounts of data (typically < 200 bytes depending on key size and padding). Hybrid encryption solves this by using RSA to encrypt a symmetric key, which then encrypts arbitrarily large data.

Hybrid encryption is the recommended approach for encrypting large files or data. It's the same technique used in protocols like TLS/SSL and PGP.

Hybrid Encryption and Decryption

Hybrid encryption and decryption using `CryptoUtils`

```
KeyPair keyPair = KeyUtils.generateKeyPair3072();
PlainText plainText = new PlainText("Large amount of sensitive data...");

// Encrypt
HybridEncryptionResult result = CryptoUtils.hybridEncrypt(plainText,
keyPair.getPublic());

// Decrypt
PlainText decrypted = CryptoUtils.hybridDecrypt(result, keyPair.getPrivate());
```

Alternative using builders with custom AES mode and strength

```
KeyPair keyPair = KeyUtils.generateKeyPair3072();
PlainText plainText = new PlainText("Large amount of sensitive data...");

// Encrypt with custom settings
HybridEncryptionResult result = EncryptBuilder.encryptionBuilder()
    .plainText(plainText)
    .key(keyPair.getPublic())
    .withMode(EncryptionMode.GCM)
    .withStrength(EncryptionStrength.BIT_256)
```

```
    .build();

// Decrypt
PlainText decrypted = DecryptBuilder.decryptionBuilder()
    .key(keyPair.getPrivate())
    .cipherText(result.getCipherText())
    .encryptedSymmetricalKey(result.getEncryptedSymmetricalKey())
    .withMode(result.getAesEncryptionMode())
    .withStrength(result.getAesEncryptionStrength())
    .build();
```

The `HybridEncryptionResult` contains the AES-encrypted data, the RSA-encrypted AES key, and metadata about the encryption mode and strength used.

Shamir's Secret Sharing

Shamir's Secret Sharing[4] is used to share a secret in a distributed way, most often to secure other encryption keys. These shares are used to reconstruct the original secret.

To unlock the secret via Shamir's Secret Sharing, a minimum number of shares are needed. This is called the threshold, and is used to denote the minimum number of shares needed to unlock the secret.

This implementation is based on the work of *Coda Hale*[5] and his project *Shamir*[6].

Shamir's Secret Sharing using `CryptoUtils`

```
int totalShares = 5;
int minShares = 2; // threshold
Secret secret = new Secret("Secret text");

// Split the secret into 5 shares
Shares shares = CryptoUtils.getShamirShares(secret, totalShares, minShares);
Share bob = shares.get(0);
Share alice = shares.get(1);

// Reconstruct the secret using any combination of at least `minShares` shares
Secret reconstructed = CryptoUtils.getShamirSecret(bob, alice);
```

The API validates invalid configurations such as `threshold < 2` or `threshold > totalShares`.

Alternative

```
int totalShares = 5;
int minShares = 2; // threshold
Secret secret = new Secret("Secret text");
```

```
// Split the secret into 5 shares
Shares shares = Shamir.getShares(secret, totalShares, minShares);
Share bob = shares.get(0);
Share alice = shares.get(1);

// Reconstruct the secret using any combination of at least `minShares` shares
Secret reconstructed = Shamir.getSecret(bob, alice);
```

Shamir example

A bank has a vault full of money. The bank's policy requires that nobody should be able to open the vault alone. Five employees are selected to have access to the vault and there must be at least two (2) employees at any time when opening the vault.

- **Split phase:** Five (5) keys are being distributed. `Bob`, `Alice`, `Eve`, `Tom` and `Lisa` all get one *share* each using *Shamir's Secret Sharing* to split the secret into five shares.
- **Join phase:** It's time to open the safe. The requirement is two (2) shares to open the vault. `Bob` and `Alice` bring their shares. By using *Shamir's Secret Sharing* the shares from both will be joined and the secret recreated.

Digest (hashing)

Digest or *message digest* is the result of *hashing*[7] data or content. The hashing is a one-way compression function to convert inputs of different lengths into a fixed-length output (hash value). The default used by this library is `SHA-2@SHA-512/256`. It provides 256-bit output with the internal 64-bit strength of SHA-512, combining high security with good performance on modern CPUs. Up to `SHA-2@SHA-512` is possible with `JDK 8`. Stronger and newer hashing like `SHA3` is not supported out of the box with `JDK 8`. That would require `JDK 9` or higher. However *BouncyCastle*[8] has been added as main provider and `SHA-3` is therefore available.

SHA-512/256 offers the same collision resistance as SHA-512, but produces a shorter digest (256 bits) and is faster on 64-bit systems. Other algorithms such as SHA3-256 and SHA3-512 are also supported when using the BouncyCastle provider.

Do not use Message Digest for password storage. For that you should use KDF[9] algorithms like `ARGON2`, `BCRYPT`, `SCRYPT` or `PBKDF2`. Se *Password Encoder* below.

Non supported hashing algorithms

The following hashing algorithms are **not** recommended or supported: `MD4`, `MD5`, `SHA-0` and `SHA-1`

Supported hashing algorithms

```
    SHA_256("SHA-256"),
    SHA_512_256("SHA-512/256"), // default
    SHA3_256("SHA3-256"),
```

```
    SHA3_512("SHA3-512");
```

Worth reading

- Read more about *Secure Hash Algorithms*[10]
- Nice article about *hashing security*[11]

Example

Using `CryptoUtils` *(recommended for simple use cases)*

```
// Raw bytes (SHA-512/256)
final byte[] digest = CryptoUtils.digest("My data");

// Base64 encoded
final String base64Digest = CryptoUtils.base64Digest("My data");

// Hex encoded
final String hexDigest = CryptoUtils.hexDigest("My data");
```

Alternative using `DigestUtils` *directly*

```
// Raw bytes
final byte[] digest = DigestUtils.digest(data);

// Base64 encoded
final String digest = DigestUtils.base64Digest(data);

// Hex encoded
final String digest = DigestUtils.hexDigest(data);

// Other hashing algorithms (example SHA3-256)
final byte[] digest = DigestUtils.digest(data, DigestAlgorithm.SHA3_256);
```

Password Encoder

Key Derivation Functions (KDF)[9] from a password must be able to stand attacks like **brute-forcing**, **dictionary attacks**, **rainbow attacks** and more. Attempts to reverse hashed password values is common.

`ARGON2`, `BCRYPT`, `SCRYPT` and `PBKDF2` are common algorithms used for password hashing since they are much more robust when attacked.

`ARGON2` is recommended as the most secure hash algorithm for passwords and is the default implementation for this API.

Encoding

A plaintext password should always be encoded in case of a breach. After a password is encoded it may be stored for future matching.

Supported password encoders

```
    ARGON2("argon2"), // default
    BCRYPT("bcrypt"),
    SCRYPT("scrypt"),
    PBKDF2("pbkdf2");
```

Password encode (default `ARGON2`)

```
final String encodedPassword = PasswordEncoderUtils.encode(password);
```

Alternative encoding with use of `enum PasswordEncoderType`

```
final String encodedPassword = PasswordEncoderUtils.encode(password,
PasswordEncoderType.BCRYPT);
```

Matching

When a user is authenticated they must input their password. This plaintext password will be matched against the stored encoded password.

Password matching (default `ARGON2`)

```
final boolean isMatching = PasswordEncoderUtils.matches(rawPassword, encodedPassword);
```

Alternative matching with use of `enum PasswordEncoderType`

```
final boolean isMatching = PasswordEncoderUtils.matches(rawPassword, encodedPassword,
PasswordEncoderType.BCRYPT);
```

Signature and verification

A *digital signature*[12] is a mathematical scheme for verifying the authenticity of digital messages or documents. A valid digital signature, where the prerequisites are satisfied, gives a recipient very strong reason to believe that the message was created by a known sender (authentication), and that the message was not altered in transit.

Commons Security uses `RSASSA-PSS` (SHA-256 with MGF1)[13] — the modern and recommended signature scheme replacing legacy `SHA256withRSA`.

Using `CryptoUtils` (recommended for simple use cases)

```
KeyPair keyPair = KeyUtils.generateKeyPair2048();

// Sign data
byte[] signature = CryptoUtils.sign("My data", keyPair.getPrivate());

// Verify signature
boolean verified = CryptoUtils.verify(signature, "My data", keyPair.getPublic());
```

Alternative using `SignatureUtils` directly

```
KeyPair keyPair = KeyUtils.generateKeyPair2048();
byte[] signature = SignatureUtils.sign("My data", keyPair.getPrivate());
boolean verified = SignatureUtils.verify(signature, "My data", keyPair.getPublic());
```

RSASSA-PSS provides probabilistic padding, making it more secure than the traditional PKCS#1 v1.5 (`SHA256withRSA`).

Documentation

PDF Version

A PDF version of this documentation is available:

- **Latest version:** Download PDF from GitHub Pages
- **Release versions:** Available as assets on the Releases page

Generate PDF locally

To generate a PDF version locally, you can use your IDE's AsciiDoc plugin or install `asciidoctor-pdf`:

```
# Install asciidoctor-pdf
gem install asciidoctor-pdf

# Generate PDF
asciidoctor-pdf README.adoc -o README.pdf
```

Recommended reading

- Practical Cryptography for Developers[14] by Svetlin Nakov
- Password1 - Why we moved to 256-bit AES keys[15] by Jeffrey Goldberg in 2013

TODO

- Make API fluently as much as possible
- Replacing MultipleMissingArgumentsError with own class without dependency to opentest4j and only throwing/supporting RuntimeException
- Support more algorithms for hashing digests[10] including password hashing
- Implement support for ECC as option over RSA[16]. ECC is faster and stronger than RSA.
- Look into Diffie-Hellman and/or other better key exchange alternatives
- Consider adding support for generating and validation passwords with help of *Passay*[17] library
- Consider making this project a library on *Maven Central*[18]

[1] https://en.wikipedia.org/wiki/RSA_(cryptosystem)

[2] https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange

[3] https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[4] https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing

[5] https://github.com/codahale

[6] https://github.com/codahale/shamir

[7] https://en.wikipedia.org/wiki/Hash_function

[8] https://www.bouncycastle.org/java.html

[9] Key Derivation Functions. https://cryptobook.nakov.com/mac-and-key-derivation/kdf-deriving-key-from-password

[10] https://en.wikipedia.org/wiki/Secure_Hash_Algorithms

[11] https://crackstation.net/hashing-security.htm

[12] https://en.wikipedia.org/wiki/Digital_signature

[13] https://www.encryptionconsulting.com/overview-of-rsassa-pss

[14] https://cryptobook.nakov.com/

[15] https://blog.1password.com/why-we-moved-to-256-bit-aes-keys/

[16] https://www.globalsign.com/en/blog/elliptic-curve-cryptography

[17] http://www.passay.org

[18] https://central.sonatype.org/publish/