PGPainless

Release 1.3.6

Paul Schaub

Sep 13, 2022

CONTENTS

OpenPGP (RFC 4480[1]) is an Internet Standard mostly used for email encryption. It provides mechanisms to ensure *confidentiality*, *integrity* and *authenticity* of messages. However, OpenPGP can also be used for other purposes, such as secure messaging or as a signature mechanism for software distribution.

PGPainless strives to improve the (currently pretty dire) state of the ecosystem of Java libraries and tooling for OpenPGP.

The library focuses on being easy and intuitive to use without getting into your way. Common functions such as creating keys, encrypting data, and so on are implemented using a builder structure that guides you through the necessary steps.

Internally, it is based on Bouncy Castles[2] mighty, but low-level `bcpg` OpenPGP API. PGPainless' goal is to empower you to use OpenPGP without needing to write all the boilerplate code required by Bouncy Castle. It aims to be secure by default while allowing customization if required.

From its inception in 2018 as part of a Google Summer of Code project[3], the library was steadily advanced. Since 2020, FlowCrypt is the primary sponsor of its development. In 2022, PGPainless received a grant from NLnet for creating a Web-of-Trust implementation[4] as part of NGI Assure.

[1] https://datatracker.ietf.org/doc/rfc4880/
[2] https://www.bouncycastle.org/java.html
[3] https://summerofcode.withgoogle.com/archive/2018/projects/6037508810866688
[4] https://nlnet.nl/project/PGPainless/

CHAPTER

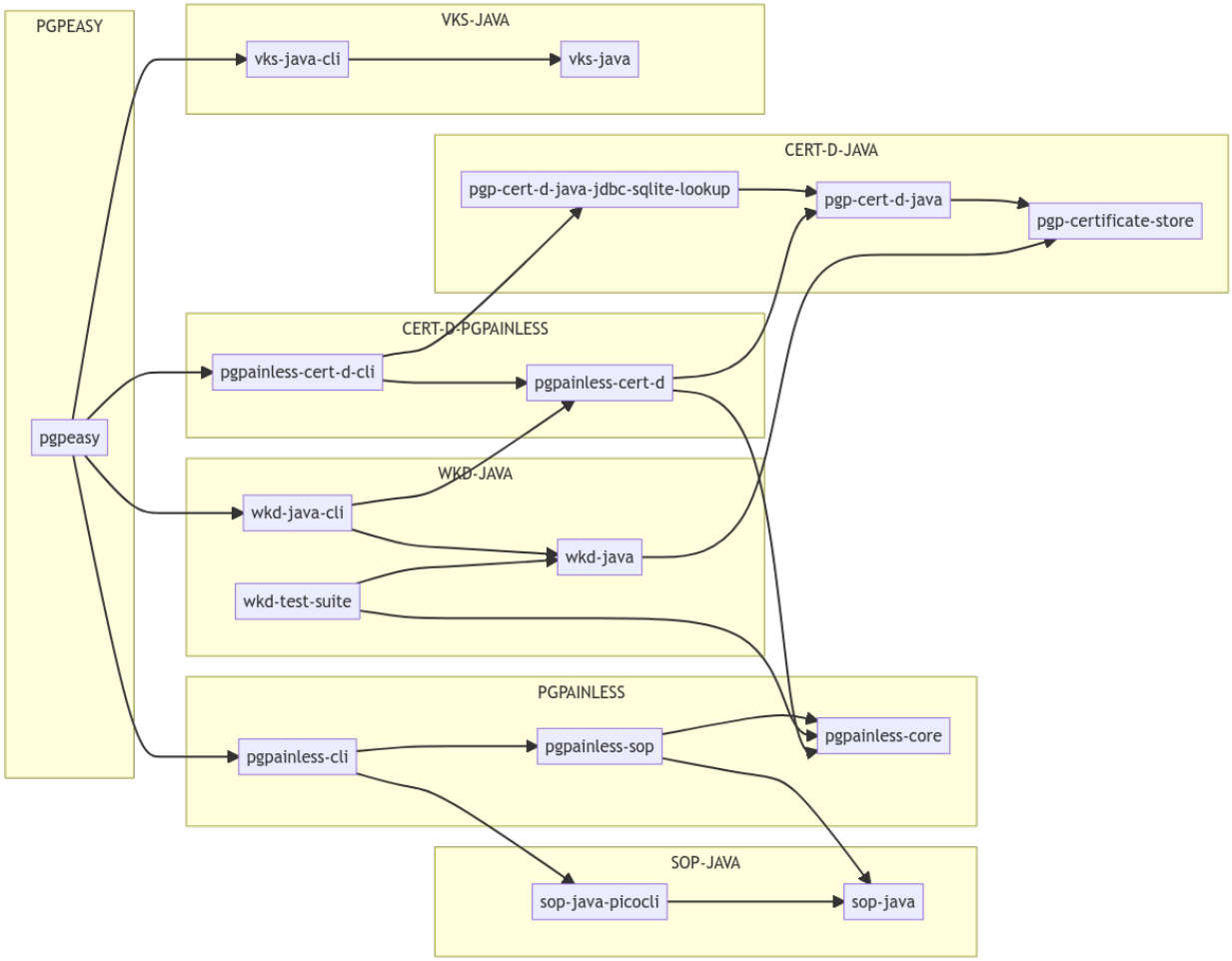
ONE

CONTENTS

1.1 The PGPainless Ecosystem

PGPainless consists of an ecosystem of different libraries and projects.

The diagram below shows, how the different projects relate to one another.



1.1.1 Libraries and Tools

- PGPainless[5]

  The main repository contains the following components:

  - `pgpainless-core` - core implementation - powerful, yet easy to use OpenPGP API
  - `pgpainless-sop` - super simple OpenPGP implementation. Drop-in for `sop-java`
  - `pgpainless-cli` - SOP CLI implementation using PGPainless

- SOP-Java[6]

  An API definition and CLI implementation of the Stateless OpenPGP Protocol[7] (SOP). Consumers of the SOP API can simply depend on `sop-java` and then switch out the backend as they wish. Read more about the *SOP* protocol here.

  - `sop-java` - generic OpenPGP API definition
  - `sop-java-picocli` - CLI frontend for `sop-java`

- WKD-Java[8]

  Implementation of the Web Key Directory[9].

  - `wkd-java` - generic WKD discovery implementation
  - `wkd-java-cli` - CLI frontend for WKD discovery using PGPainless
  - `wkd-test-suite` - Generator for test vectors for testing WKD implementations

- VKS-Java[10]

  Client-side API for communicating with Verifying Key Servers, such as https://keys.openpgp.org/.

  - `vks-java` - VKS client implementation
  - `vks-java-cli` - CLI frontend for `vks-java`

- Cert-D-Java[11]

  Implementations of the Shared OpenPGP Certificate Directory specification[12].

  - `pgp-certificate-store` - abstract definitions of OpenPGP certificate stores
  - `pgp-cert-d-java` - implementation of `pgp-certificate-store` following the PGP-CERT-D spec
  - `pgp-cert-d-java-jdbc-sqlite-lookup` - subkey lookup using sqlite database

- Cert-D-PGPainless[13]

  Implementation of the Shared OpenPGP Certificate Directory specification[14] using PGPainless.

  - `pgpainless-cert-d` - PGPainless-based implementation of `pgp-cert-d-java`
  - `pgpainless-cert-d-cli` - CLI frontend for `pgpainless-cert-d`

[5] https://codeberg.org/pgpainless/pgpainless
[6] https://codeberg.org/pgpainless/sop-java
[7] https://datatracker.ietf.org/doc/draft-dkg-openpgp-stateless-cli/
[8] https://codeberg.org/pgpainless/wkd-java
[9] https://www.ietf.org/archive/id/draft-koch-openpgp-webkey-service-13.html
[10] https://codeberg.org/pgpainless/vks-java
[11] https://codeberg.org/pgpainless/cert-d-java
[12] https://sequoia-pgp.gitlab.io/pgp-cert-d/
[13] https://codeberg.org/pgpainless/cert-d-pgpainless
[14] https://sequoia-pgp.gitlab.io/pgp-cert-d/

- PGPeasy[15]

  Prototypical, comprehensive OpenPGP CLI application

  - `pgpeasy` - CLI application

1.2 Quickstart Guide

In this guide, we will get you started with OpenPGP using PGPainless as quickly as possible.

At first though, you need to decide which API you want to use;

- PGPainless' core API is powerful and heavily customizable
- The SOP API is a bit less powerful, but *dead* simple to use

The SOP API is the recommended way to go if you just want to get started already.

In case you need more technical documentation, Javadoc can be found in the following places:

- For the core API: pgpainless-core[16]
- For the SOP API: pgpainless-sop[17]

1.2.1 SOP API with pgpainless-sop

The Stateless OpenPGP Protocol (SOP) defines a simplistic interface for the most important OpenPGP operations. It allows you to encrypt, decrypt, sign and verify messages, generate keys and add/remove ASCII armor from data. However, it does not yet provide tools for key management. Furthermore, the implementation is deciding for you, which (secure) algorithms to use, and it doesn't let you change those.

If you want to read more about the background of the SOP protocol, there is a *whole chapter* dedicated to it.

Setup

PGPainless' releases are published to and can be fetched from Maven Central. To get started, you first need to include `pgpainless-sop` in your projects build script.

```
// If you use Gradle
...
dependencies {
    ...
    implementation "org.pgpainless:pgpainless-sop:XYZ"
    ...
}

// If you use Maven
...
<dependencies>
    ...
    <dependency>
        <groupId>org.pgpainless</groupId>
```

(continues on next page)

[15] https://codeberg.org/pgpainless/pgpeasy
[16] https://javadoc.io/doc/org.pgpainless/pgpainless-core/1.3.6/index.html
[17] https://javadoc.io/doc/org.pgpainless/pgpainless-sop/1.3.6/index.html

(continued from previous page)

```
        <artifactId>pgpainless-sop</artifactId>
        <version>XYZ</version>
    </dependency>
    ...
</dependencies>
```

Important: Replace `XYZ` with the current version, in this case 1.3.6!

The entry point to the API is the `SOP` interface, for which `pgpainless-sop` provides a concrete implementation `SOPImpl`.

```
// Instantiate the API
SOP sop = new SOPImpl();
```

Now you are ready to go!

Generate a Key

To generate a new OpenPGP key, the method `SOP.generateKey()` is your friend:

```
// generate key
byte[] keyBytes = sop.generateKey()
        .userId("John Doe <john.doe@pgpainless.org>")
        .withKeyPassword("f00b4r")
        .generate()
        .getBytes();
```

The call `userId(String userId)` can be called multiple times to add multiple user-ids to the key, but it MUST be called at least once. The argument given in the first invocation will become the keys primary user-id.

Optionally, the key can be protected with a password by calling `withKeyPassword(String password)`. If this method is not called, the key will be unprotected.

The `generate()` method call generates the key and returns a `Ready` object. This in turn can be used to write the result to a stream via `writeTo(OutputStream out)`, or to get the result as bytes via `getBytes()`. In both cases, the resulting output will be the UTF8 encoded, ASCII armored OpenPGP secret key.

To disable ASCII armoring, call `noArmor()` before calling `generate()`.

At the time of writing, the resulting OpenPGP secret key will consist of a certification-capable 256-bits ed25519 EdDSA primary key, a 256-bits ed25519 EdDSA subkey used for signing, as well as a 256-bits X25519 ECDH subkey for encryption.

The whole key does not have an expiration date set.

Extract a Certificate

Now that you generated your secret key, you probably want to share the public key with your contacts. To extract the OpenPGP public key (which we will call *certificate* from now on) from the secret key, use the `SOP.extractCert()` method call:

```
// extract certificate
byte[] certificateBytes = sop.extractCert()
        .key(keyBytes)
        .getBytes();
```

The `key(_)` method either takes a byte array (like in the example), or an `InputStream`. In both cases it returns another `Ready` object from which the certificate can be accessed, either via `writeTo(OutputStream out)` or `getBytes()`.

By default, the resulting certificate will be ASCII armored, regardless of whether the input key was armored or not. To disable ASCII armoring, call `noArmor()` before calling `key(_)`.

In our example, `certificateBytes` can now safely be shared with anyone.

Apply / Remove ASCII Armor

Perhaps you want to print your secret key onto a piece of paper for backup purposes, but you accidentally called `noArmor()` when generating the key.

To add ASCII armor to some binary OpenPGP data, the `armor()` API can be used:

```
// wrap data in ASCII armor
byte[] armoredData = sop.armor()
        .data(binaryData)
        .getBytes();
```

The `data(_)` method can either be called by providing a byte array, or an `InputStream`.

Note: There is a `label(ArmorLabel label)` method, which could theoretically be used to define the label used in the ASCII armor header. However, this method is not (yet?) supported by `pgpainless-sop` and will currently throw an `UnsupportedOption` exception. Instead, the implementation will figure out the data type and set the respective label on its own.

To remove ASCII armor from armored data, simply use the `dearmor()` API:

```
// remove ASCII armor
byte[] binaryData = sop.unarmor()
        .data(armoredData)
        .getBytes();
```

Once again, the `data(_)` method can be called either with a byte array or an `InputStream` as argument.

If the input data is not validly armored OpenPGP data, the `data(_)` method call will throw a `BadData` exception.

Encrypt a Message

Now lets get to the juicy part and finally encrypt a message! In this example, we will assume that Alice is the sender that wants to send a message to Bob. Beforehand, Alice acquired Bobs certificate, e.g. by fetching it from a key server.

To encrypt a message, you can make use of the `encrypt()` API:

```
// encrypt and sign a message
byte[] aliceKey = ...;  // Alice' secret key
byte[] aliceCert = ...; // Alice' certificate (e.g. via extractCert())
byte[] bobCert = ...;   // Bobs certificate

byte[] plaintext = "Hello, World!\n".getBytes(); // plaintext

byte[] ciphertext = sop.encrypt()
        // encrypt for each recipient
        .withCert(bobCert)
        .withCert(aliceCert)
        // Optionally: Sign the message
        .signWith(aliceKey)
        .withKeyPassword("sw0rdf1sh") // if signing key is protected
        // provide the plaintext
        .plaintext(plaintext)
        .getBytes();
```

Here you encrypt the message for each recipient (Alice probably wants to be able to decrypt the message too!) by calling `withCert(_)` with the recipients certificate as argument. It does not matter, if the certificate is ASCII armored or not, and the method can either be called with a byte array or an `InputStream` as argument.

The API not only supports asymmetric encryption via OpenPGP certificates, but it can also encrypt messages symmetrically using one or more passwords. Both mechanisms can even be used together in the same message! To (additionally or exclusively) encrypt the message for a password, simply call `withPassword(String password)` before the `plaintext(_)` method call.

It is recommended (but not required) to sign encrypted messages. In order to sign the message before encryption is applied, call `signWith(_)` with the signing key as argument. This method call can be repeated multiple times to sign the message with multiple signing keys.

If any keys used for signing are password protected, you need to provide the signing key password via `withKeyPassword(_)`. It does not matter in which order signing keys and key passwords are provided, the implementation will figure out matches on its own. If different key passwords are used, the `withKeyPassword(_)` method can be called multiple times.

By default, the encrypted message will be ASCII armored. To disable ASCII armor, call `noArmor()` before the `plaintext(_)` method call.

Lastly, you need to provide the plaintext by calling `plaintext(_)` with either a byte array or an `InputStream` as argument. The ciphertext can then be accessed from the resulting `Ready` object as usual.

Decrypt a Message

Now let's switch perspective and help Bob decrypt the message from Alice.

Decrypting encrypted messages is done in a similar fashion using the `decrypt()` API:

```
// decrypt a message and verify its signature(s)
byte[] aliceCert = ...; // Alice' certificate
byte[] bobKey = ...;    // Bobs secret key
byte[] bobCert = ...;   // Bobs certificate

byte[] ciphertext = ...; // the encrypted message

ReadyWithResult<DecryptionResult> readyWithResult = sop.decrypt()
        .withKey(bobKey)
        .verifyWith(aliceCert)
        .withKeyPassword("password123") // if decryption key is protected
        .ciphertext(ciphertext);
```

The `ReadyWithResult<DecryptionResult>` can now be processed in two different ways, depending on whether you want the plaintext as bytes or simply write it out to an `OutputStream`.

To get the plaintext bytes directly, you shall proceed as follows:

```
ByteArrayAndResult<DecryptionResult> bytesAndResult = readyWithResult.
↪toByteArrayAndResult();
DecryptionResult result = bytesAndResult.getResult();
byte[] plaintext = bytesAndResult.getBytes();
```

If you instead want to write the plaintext out to an `OutputStream`, the following code can be used:

```
OutputStream out = ...;
DecryptionResult result = readyWithResult.writeTo(out);
```

Note, that in both cases you acquire a `DecryptionResult` object. This contains information about the message, such as which signatures could successfully be verified.

If you provided the senders certificate for the purpose of signature verification via `verifyWith(_)`, you now probably want to check, if the message was actually signed by the sender by checking `result.getVerifications()`.

Note: Signature verification will be discussed in more detail in section "Verifications".

If the message was encrypted symmetrically using a password, you can also decrypt is symmetrically by calling `withPassword(String password)` before the `ciphertext(_)` method call. This method call can be repeated multiple times. The implementation will try different passwords until it finds a matching one.

Sign a Message

There are three different main ways of signing a message:

- Inline Signatures
- Cleartext Signatures
- Detached Signatures

An inline-signature will be part of the message itself (e.g. like with messages that are encrypted *and* signed). Inline-signed messages are not human-readable without prior processing.

A cleartext signature makes use of the cleartext signature framework[18]. Messages signed in this way do have an ASCII armor header and footer, yet the content of the message is still human-readable without special software.

Lastly, a detached signature can be distributed as an extra file alongside the message without altering it. This is useful if the plaintext itself cannot be modified (e.g. if a binary file is signed).

The SOP API can generate all of those signature types.

Inline-Signatures

Let's start with an inline signature:

```
byte[] signingKey = ...;
byte[] message = ...;

byte[] inlineSignedMessage = sop.inlineSign()
        .mode(InlineSignAs.Text) // or 'Binary'
        .key(signingKey)
        .withKeyPassword("fnord")
        .data(message)
        .getBytes();
```

You can choose between two different signature formats which can be set using `mode(InlineSignAs mode)`. The default value is `Binary`. You can also set it to `Text` which signals to the receiver that the data is UTF8 text.

Note: For inline signatures, do NOT set the `mode()` to `CleartextSigned`, as that will create message which uses the cleartext signature framework (see further below).

You must provide at least one signing key using `key(_)` in order to be able to sign the message.

If any key is password protected, you need to provide its password using `withKeyPassword(_)` which can be called multiple times to provide multiple passwords.

Once you provide the plaintext using `data(_)` with either a byte array or an `InputStream` as argument, you will get a `Ready` object back, from which the signed message can be retrieved as usual.

By default, the signed message will be ASCII armored. This can be disabled by calling `noArmor()` before the `data(_)` method call.

[18] https://datatracker.ietf.org/doc/html/rfc4880#section-7

Cleartext Signatures

A cleartext-signed message can be generated in a similar way to an inline-signed message, however, there are is one subtle difference:

```
byte[] signingKey = ...;
byte[] message = ...;

byte[] cleartextSignedMessage = sop.inlineSign()
        .mode(InlineSignAs.CleartextSigned) // This MUST be set
        .key(signingKey)
        .withKeyPassword("fnord")
        .data(message)
        .getBytes();
```

Important: In order to produce a cleartext-signed message, the signature mode MUST be set to `CleartextSigned` by calling `mode(InlineSignAs.CleartextSigned)`.

Note: Calling `noArmor()` will have no effect for cleartext-signed messages, so such method call will be ignored.

Detached Signatures

As the name suggests, detached signatures are detached from the message itself and can be distributed separately.

To produce a detached signature, the `detachedSign()` API is used:

```
byte[] signingKey = ...;
byte[] message = ...;

ReadyWithResult<SigningResult> readyWithResult = sop.detachedSign()
        .key(signingKey)
        .withKeyPassword("fnord")
        .data(message);
```

Here you have the choice, how you want to write out the signature. If you want to write the signature to an `OutputStream`, you can do the following:

```
OutputStream out = ...;
SigningResult result = readyWithResult.writeTo(out);
```

If instead you want to get the signature as a byte array, do this instead:

```
ByteArrayAndResult<SigningResult> bytesAndResult = readyWithResult.
↪toByteArrayAndResult();
SigningResult result = bytesAndResult.getResult();
byte[] detachedSignature = bytesAndResult.getBytes();
```

In any case, the detached signature can now be distributed alongside the original message.

By default, the resulting detached signature will be ASCII armored. This can be disabled by calling `noArmor()` prior to calling `data(_)`.

The `SigningResult` object you got back in both cases contains information about the signature.

Verify a Signature

In order to verify signed messages, there are two API endpoints available.

Inline and Cleartext Signatures

To verify inline-signed messages, or messages that make use of the cleartext signature framework, use the `inlineVerify()` API:

```
byte[] signingCert = ...;
byte[] signedMessage = ...;

ReadyWithResult<List<Verification>> readyWithResult = sop.inlineVerify()
        .cert(signingCert)
        .data(signedMessage);
```

The `cert(_)` method MUST be called at least once. It takes either a byte array or an `InputStream` containing an OpenPGP certificate. If you are not sure, which certificate was used to sign the message, you can provide multiple certificates.

It is also possible to reject signatures that were not made within a certain time window by calling `notBefore(Date timestamp)` and/or `notAfter(Date timestamp)`. Signatures made before the `notBefore(_)` or after the `notAfter(_)` constraints will be rejected.

You can now either write out the plaintext message to an `OutputStream`...

```
OutputStream out = ...;
List<Verifications> verifications = readyWithResult.writeTo(out);
```

... or you can acquire the plaintext message as a byte array directly:

```
ByteArrayAndResult<List<Verifications>> bytesAndResult = readyWithResult.
↪toByteArrayAndResult();
byte[] plaintextMessage = bytesAndResult.getBytes();
List<Verifications> verifications = bytesAndResult.getResult();
```

In both cases, the plaintext message will have the signatures stripped.

Detached Signatures

To verify detached signatures (signatures that come separate from the message itself), you can use the `detachedVerify()` API:

```
byte[] signingCert = ...;
byte[] message = ...;
byte[] detachedSignature = ...;

List<Verification> verifications = sop.detachedVerify()
        .cert(signingCert)
        .signatures(detachedSignature)
        .data(signedMessage);
```

You can provide one or more OpenPGP certificates using `cert(_)`, providing either a byte array or an `InputStream`.

The detached signatures need to be provided separately using the `signatures(_)` method call. You can provide as many detached signatures as you like, and those can be binary or ASCII armored.

Like with Inline Signatures, you can constrain the time window for signature validity using `notAfter(_)` and `notBefore(_)`.

Verifications

In all above cases, the `verifications` list will contain `Verification` objects for each verifiable, valid signature. Those objects contain information about the signatures: `verification.getSigningCertFingerprint()` will return the fingerprint of the certificate that created the signature. `verification.getSigningKeyFingerprint()` will return the fingerprint of the used signing subkey within that certificate.

Detach Signatures from Messages

It is also possible, to detach inline or cleartext signatures from signed messages to transform them into detached signatures. The same way you can turn inline or cleartext signed messages into plaintext messages.

To detach signatures from messages, use the `inlineDetach()` API:

```
byte[] signedMessage = ...;

ReadyWithResult<Signatures> readyWithResult = sop.inlineDetach()
        .message(signedMessage);
ByteArrayAndResult<Signatures> bytesAndResult = readyWithResult.toByteArrayAndResult();

byte[] plaintext = bytesAndResult.getBytes();
Signatures signatures = bytesAndResult.getResult();
byte[] encodedSignatures = signatures.getBytes();
```

By default, the signatures output will be ASCII armored. This can be disabled by calling `noArmor()` prior to `message(_)`.

The detached signatures can now be verified like in the section above.

1.2.2 PGPainless API with pgpainless-core

The `pgpainless-core` module contains the bulk of the actual OpenPGP implementation.

This is a quickstart guide. For more in-depth exploration of the API, checkout .

Note: This chapter is work in progress.

Setup

PGPainless' releases are published to and can be fetched from Maven Central. To get started, you first need to include `pgpainless-core` in your projects build script:

```
// If you use Gradle
...
dependencies {
    ...
    implementation "org.pgpainless:pgpainless-core:XYZ"
    ...
}

// If you use Maven
...
<dependencies>
    ...
    <dependency>
        <groupId>org.pgpainless</groupId>
        <artifactId>pgpainless-core</artifactId>
        <version>XYZ</version>
    </dependency>
    ...
</dependencies>
```

This will automatically pull in PGPainless' dependencies, such as Bouncy Castle.

Important: Replace `XYZ` with the current version, in this case 1.3.6!

The entry point to the API is the `PGPainless` class. For many common use-cases, examples can be found in the examples package[19]. There is a very good chance that you can find code examples there that fit your needs.

Read and Write Keys

Reading keys from ASCII armored strings or from binary files is easy:

```
String key = "-----BEGIN PGP PRIVATE KEY BLOCK-----\n"...;
PGPSecretKeyRing secretKey = PGPainless.readKeyRing()
        .secretKeyRing(key);
```

Similarly, keys or certificates can quickly be exported:

```
// ASCII armored key
PGPSecretKeyRing secretKey = ...;
String armored = PGPainless.asciiArmor(secretKey);

// binary (unarmored) key
byte[] binary = secretKey.getEncoded();
```

[19] https://codeberg.org/pgpainless/pgpainless/src/branch/main/pgpainless-core/src/test/java/org/pgpainless/example

Generate a Key

PGPainless comes with a method to quickly generate modern OpenPGP keys. There are some predefined key archetypes, but it is possible to fully customize the key generation to fit your needs.

```
// EdDSA primary key with EdDSA signing- and XDH encryption subkeys
PGPSecretKeyRing secretKeys = PGPainless.generateKeyRing()
        .modernKeyRing("Romeo <romeo@montague.lit>", "thisIsAPassword");

// RSA key without additional subkeys
PGPSecretKeyRing secretKeys = PGPainless.generateKeyRing()
        .simpleRsaKeyRing("Juliet <juliet@montague.lit>", RsaLength._4096);
```

As you can see, it is possible to generate all kinds of different keys.

Extract a Certificate

If you have a secret key, you might want to extract a public key certificate from it:

```
PGPSecretKeyRing secretKey = ...;
PGPPublicKeyRing certificate = PGPainless.extractCertificate(secretKey);
```

Apply / Remove ASCII Armor

ASCII armor is a layer of radix64 encoding that can be used to wrap binary OpenPGP data in order to make it save to transport via text-based channels (e.g. email bodies).

The way in which ASCII armor can be applied depends on the type of data that you want to protect. The easies way to ASCII armor an OpenPGP key or certificate is by using PGPainless' `asciiArmor()` method:

```
PGPPublicKey certificate = ...;
String asciiArmored = PGPainless.asciiArmor(certificate);
```

If you want to ASCII armor ciphertext, you can enable ASCII armoring during encrypting/signing by requesting PGPainless to armor the result:

```
ProducerOptions producerOptions = ...; // prepare as usual (see next section)

producerOptions.setAsciiArmor(true); // enable armoring

EncryptionStream encryptionStream = PGPainless.encryptAndOrSign()
        .onOutputStream(out)
        .withOptions(producerOptions);

...
```

If you have an already encrypted / signed binary message and want to add ASCII armoring retrospectively, you need to make use of BouncyCastle's `ArmoredOutputStream` as follows:

```
InputStream binaryOpenPgpIn = ...; // e.g. new ByteArrayInputStream(binaryMessage);

OutputStream output = ...; // e.g. new ByteArrayOutputStream();
ArmoredOutputStream armorOut = ArmoredOutputStreamFactory.get(output);
```

(continues on next page)

(continued from previous page)

```
Streams.pipeAll(binaryOpenPgpIn, armorOut);
armorOut.close(); // important!
```

The output stream will now contain the ASCII armored representation of the binary data.

If the data you want to wrap in ASCII armor is non-OpenPGP data (e.g. the String "Hello World!"), you need to use the following code:

```
InputStream inputStream = ...;
OutputStream output = ...;

EncryptionStream armorStream = PGPainless.encryptAndOrSign()
        .onOutputStream(output)
        .withOptions(ProducerOptions.noEncryptionNoSigning()
                .setAsciiArmor(true));

Streams.pipeAll(inputStream, armorStream);
armorStream.close();
```

To remove ASCII armor, you can make use of BouncyCastle's `ArmoredInputStream` as follows:

```
InputStream input = ...; // e.g. new ByteArrayInputStream(armoredString.
↪getBytes(StandardCharsets.UTF8));
OutputStream output = ...;

ArmoredInputStream armorIn = new ArmoredInputStream(input);
Streams.pipeAll(armorIn, output);
armorIn.close();
```

The output stream will now contain the binary OpenPGP data.

Encrypt and/or Sign a Message

Encrypting and signing messages is done using the same API in PGPainless. The type of action depends on the configuration of the `ProducerOptions` class, which in term accepts `SigningOptions` and `EncryptionOptions` objects:

```
// Encrypt only
ProducerOptions options = ProducerOptions.encrypt(encryptionOptions);

// Sign only
ProducerOptions options = ProducerOptions.sign(signingOptions);

// Sign and encrypt
ProducerOptions options = ProducerOptions.signAndEncrypt(signingOptions,␣
↪encryptionOptions);
```

The `ProducerOptions` object can then be passed into the `encryptAndOrSign()` API:

```
InputStream plaintext = ...; // The data that shall be encrypted and/or signed
OutputStream ciphertext = ...; // Destination for the ciphertext
```

(continues on next page)

(continued from previous page)

```
EncryptionStream encryptionStream = PGPainless.encryptAndOrSign()
        .onOutputStream(ciphertext)
        .withOptions(options); // pass in the options object

Streams.pipeAll(plaintext, encryptionStream); // pipe the data through
encryptionStream.close(); // important! Close the stream to finish encryption/signing

EncryptionResult result = encryptionStream.getResult(); // metadata
```

The `ciphertext` output stream now contains the encrypted and/or signed data.

Now lets take a look at the configuration of the `SigningOptions` object and how to instruct PGPainless to add a simple signature to the message:

```
PGPSecretKeyRing signingKey = ...; // Key used for signing
SecretKeyRingProtector protector = ...; // Protector to unlock the signing key

SigningOptions signOptions = SigningOptions.get()
        .addSignature(protector, signingKey);
```

This will add an inline signature to the message.

It is possible to add multiple signatures from different keys by repeating the `addSignature()` method call.

If instead of an inline signature, you want to create a detached signature instead (e.g. because you do not want to alter the data you are signing), you can add the signature as follows:

```
signOptions.addDetachedSignature(protector, signingKey);
```

Passing in the `SigningOptions` object like this will result in the signature not being added to the message itself. Instead, the signature can later be acquired from the `EncryptionResult` object via `EncryptionResult.getDetachedSignatures()`. That way, it can be distributed independent of the message.

The `EncryptionOptions` object can be configured in a similar way:

```
PGPPublicKey certificate = ...;

EncryptionOptions encOptions = EncryptionOptions.get()
        .addRecipient(certificate);
```

Once again, it is possible to add multiple recipients by repeating the `addRecipient()` method call.

You can also encrypt a message to a password like this:

```
encOptions.addPassphrase(Passphrase.fromPassword("sw0rdf1sh"));
```

Both methods can be used in combination to create a message which can be decrypted with either a recipients secret key or the passphrase.

Decrypt and/or Verify a Message

Decryption and verification of a message is both done using the same API. Whether a message was actually signed / encrypted can be determined after the message has been processed by checking the `OpenPgpMetadata` object which can be obtained from the `DecryptionStream`.

To configure the decryption / verification process, the `ConsumerOptions` object is used:

```
PGPPublicKeyRing verificationCert = ...; // optional, signers certificate for signature␣
↪verification
PGPSecretKeyRing decryptionKey = ...; // optional, decryption key

ConsumerOptions options = ConsumerOptions.get()
        .addVerificationCert(verificationCert) // add a verification cert for signature␣
↪verification
        .addDecryptionKey(decryptionKey); // add a secret key for message decryption
```

Both verification certificates and decryption keys are optional. If you know the message is signed, but not encrypted you can omit providing a decryption key. Same goes for if you know that the message is encrypted, but not signed. In this case you can omit the verification certificate.

On the other hand, providing these parameters does not hurt. PGPainless will ignore unused keys / certificates, so if you provide a decryption key and the message is not encrypted, nothing bad will happen.

It is possible to provide multiple verification certs and decryption keys. PGPainless will pick suitable ones on the fly. If the message is signed with key `0xAAAA` and you provide certificates `0xAAAA` and `0xBBBB`, it will verify with cert `0xAAAA` and ignore `0xBBBB`.

To do the actual decryption / verification of the message, do the following:

```
InputStream ciphertext = ...; // encrypted and/or signed message
OutputStream plaintext = ...; // destination for the plaintext

ConsumerOptions options = ...; // see above
DecryptionStream consumerStream = PGPainless.decryptAndOrVerify()
        .onInputStream(ciphertext)
        .withOptions(options);

Streams.pipeAll(consumerStream, plaintext);
consumerStream.close(); // important!

// The result will contain metadata of the message
OpenPgpMetadata result = consumerStream.getResult();
```

After the message has been processed, you can consult the `OpenPgpMetadata` object to determine the nature of the message:

```
boolean wasEncrypted = result.isEncrypted();
SubkeyIdentifier decryptionKey = result.getDecryptionKey();
Map<SubkeyIdentifier, PGPSignature> validSignatures = result.getVerifiedSignatures();
boolean wasSignedByCert = result.containsVerifiedSignatureFrom(certificate);

// For files:
String fileName = result.getFileName();
Date modificationData = result.getModificationDate();
```

Verify a Signature

In some cases, detached signatures are distributed alongside the message. This is the case for example with Debians `Release` and `Release.gpg` files. Here, `Release` is the plaintext message, which is unaltered by the signing process while `Release.gpg` contains the detached OpenPGP signature.

To verify a detached signature, you need to call the PGPainless API like this:

```
InputStream plaintext = ...; // e.g. new FileInputStream(releaseFile);
InputStream detachedSignature = ...; // e.g. new FileInputStream(releaseGpgFile);
PGPPublicKeyRing certificate = ...; // e.g. debians public signing key

ConsumerOptions options = ConsumerOptions.get()
        .addVerificationCert(certificate) // provide certificate for verification
        .addVerificationOfDetachedSignatures(detachedSignature) // provide detached␣
↪signature

DecryptionStream verificationStream = PGPainless.decryptAndOrVerify()
        .onInputStream(plaintext)
        .withOptions(options);

Streams.drain(verificationStream); // push all the data through the stream
verificationStream.close(); // finish verification

OpenPgpMetadata result = verificationStream.getResult(); // get metadata of signed␣
↪message
assertTrue(result.containsVerifiedSignatureFrom(certificate)); // check if message was␣
↪in fact signed
```

1.3 User Guide PGPainless-CLI

The module `pgpainless-cli` contains a command line application which conforms to the Stateless OpenPGP Command Line Interface[20].

You can use it to generate keys, encrypt, sign and decrypt messages, as well as verify signatures.

1.3.1 Implementation

Essentially, `pgpainless-cli` is just a very small composing module, which injects `pgpainless-sop` as a concrete implementation of `sop-java` into `sop-java-picocli`.

[20] https://datatracker.ietf.org/doc/draft-dkg-openpgp-stateless-cli/

1.3.2 Install

The `pgpainless-cli` command line application is available in Debian unstable / Ubuntu 22.10 and can be installed via APT:

```
$ sudo apt install pgpainless-cli
```

This method comes with man-pages:

```
$ man pgpainless-cli
```

1.3.3 Build

To build a standalone *fat*-jar:

```
$ cd pgpainless-cli/
$ gradle shadowJar
```

The fat-jar can afterwards be found in `build/libs/`.

To build a distributable[21]:

```
$ cd pgpainless-cli/
$ gradle installDist
```

Afterwards, an uncompressed distributable is installed in `build/install/`. To execute the application, you can call `build/install/bin/pgpainless-cli{.bat}`

1.3.4 Usage

Hereafter, the program will be referred to as `pgpainless-cli`.

```
$ pgpainless-cli help
Stateless OpenPGP Protocol
Usage: pgpainless-cli [COMMAND]

Commands:
  help           Display usage information for the specified subcommand
  armor          Add ASCII Armor to standard input
  dearmor        Remove ASCII Armor from standard input
  decrypt        Decrypt a message from standard input
  inline-detach  Split signatures from a clearsigned message
  encrypt        Encrypt a message from standard input
  extract-cert   Extract a public key certificate from a secret key from
                   standard input
  generate-key   Generate a secret key
  sign           Create a detached signature on the data from standard input
  verify         Verify a detached signature over the data from standard input
  inline-sign    Create an inline-signed message from data on standard input
  inline-verify  Verify inline-signed data from standard input
  version        Display version information about the tool
```

(continues on next page)

[21] https://docs.gradle.org/current/userguide/distribution_plugin.html

(continued from previous page)

```
Exit Codes:
   0   Successful program execution.
   1   Generic program error
   3   Verification requested but no verifiable signature found
  13   Unsupported asymmetric algorithm
  17   Certificate is not encryption capable
  19   Usage error: Missing argument
  23   Incomplete verification instructions
  29   Unable to decrypt
  31   Password is not human-readable
  37   Unsupported Option
  41   Invalid data or data of wrong type encountered
  53   Non-text input received where text was expected
  59   Output file already exists
  61   Input file does not exist
  67   Cannot unlock password protected secret key
  69   Unsupported subcommand
  71   Unsupported special prefix (e.g. "@env/@fd") of indirect parameter
  73   Ambiguous input (a filename matching the designator already exists)
  79   Key is not signing capable
Powered by picocli
```

1.4 Stateless OpenPGP Protocol (SOP)

The Stateless OpenPGP Protocol[22] (short *SOP*) is a specification of a standardized command line interface for a limited set of OpenPGP operations.

By standardizing the interface, users are able to choose between different, compatible implementations.

Note: This chapter is work in progress.

[22] https://datatracker.ietf.org/doc/draft-dkg-openpgp-stateless-cli/

1.5 In-Depth Guide to pgpainless-core

This is an in-depth introduction to OpenPGP using PGPainless. If you are looking for a quickstart introduction instead, check out [](quickstart.md).

1.5.1 Contents

PGPainless In-Depth: Generate Keys

There are two API endpoints for generating OpenPGP keys using `pgpainless-core`:

`PGPainless.generateKeyRing()` presents a selection of pre-configured OpenPGP key archetypes:

```
// Modern, EC-based OpenPGP key with dedicated primary certification key
// This method is recommended by the authors
PGPSecretKeyRing secretKey = PGPainless.generateKeyRing()
        .modernKeyRing(
                "Alice <alice@pgpainless.org>",
                Passphrase.fromPassword("sw0rdf1sh"));

// Simple, EC-based OpenPGP key with combined certification and signing key
// plus encryption subkey
PGPSecretKeyRing secretKey = PGPainless.generateKeyRing()
        .simpleEcKeyRing(
                "Alice <alice@pgpainless.org>",
                Passphrase.fromPassword("0r4ng3"));

// Simple, RSA OpenPGP key made of a single RSA key used for all operations
PGPSecretKeyRing secretKey = PGPainless.generateKeyRing()
        .simpleRsaKeyRing(
                "Alice <alice@pgpainless.org>",
                RsaLength._4096, Passphrase.fromPassword("m0nk3y")):
```

If you have special requirements on algorithms you can use `PGPainless.buildKeyRing()` instead, which offers more control over parameters:

```
// Customized key

// Specification for primary key
KeySpecBuilder primaryKeySpec = KeySpec.getBuilder(
        KeyType.RSA(RsaLength._8192),                                   // 8192 bits RSA key
        KeyFlag.CERTIFY_OTHER)                                          // used for␣
↪certification
        // optionally override algorithm preferences
        .overridePreferredCompressionAlgorithms(CompressionAlgorithm.ZLIB)
        .overridePreferredHashAlgorithms(HashAlgorithm.SHA512, HashAlgorithm.SHA384)
        .overridePreferredSymmetricKeyAlgorithms(SymmetricKeyAlgorithm.AES256);

// Specification for a signing subkey
KeySpecBuilder signingSubKeySpec = KeySpec.getBuilder(
        KeyType.ECDSA(EllipticCurve._P256),                             // P-256 ECDSA key
```

(continues on next page)

(continued from previous page)

```
        KeyFlag.SIGN_DATA);                                          // Used for signing

// Specification for an encryption subkey
KeySpecBuilder encryptionSubKeySpec = KeySpec.getBuilder(
        KeyType.ECDH(EllipticCurve._P256),
        KeyFlag.ENCRYPT_COMMS, KeyFlag.ENCRYPT_STORAGE);

// Build the key itself
PGPSecretKeyRing secretKey = PGPainless.buildKeyRing()
        .setPrimaryKey(primaryKeySpec)
        .addSubkey(signingSubKeySpec)
        .addSubkey(encryptionSubKeySpec)
        .addUserId("Juliet <juliet@montague.lit>")                   // Primary User-ID
        .addUserId("xmpp:juliet@capulet.lit")                        // Additional User-ID
        .setPassphrase(Passphrase.fromPassword("romeo_oh_Romeo<3")) // passphrase␣
↪protection
        .build();
```

To specify, which algorithm to use for a single (sub) key, `KeySpec.getBuilder(_)` can be used, passing a `KeyType`, as well as some `KeyFlag`s as argument.

`KeyType` defines an algorithm and its parameters, e.g. RSA with a certain key size, or ECDH over a certain elliptic curve. Currently, PGPainless supports the following `KeyType`s:

- `KeyType.RSA(_)`: Signing, Certification, Encryption
- `KeyType.ECDH(_)`: Encryption
- `KeyType.ECDSA(_)`: Signing, Certification
- `KeyType.EDDSA(_)`: Signing, Certification
- `KeyType.XDH(_)`: Encryption

The `KeyFlag`s are used to specify, how the key will be used later on. A signing key can only be used for signing, if it carries the `KeyFlag.SIGN_DATA`. A key can carry multiple key flags.

It is possible to override the default algorithm preferences used by PGPainless with custom preferences. An algorithm preference list contains algorithms from most to least preferred.

Every OpenPGP key MUST have a primary key. The primary key MUST be capable of certification, so you MUST use an algorithm that can be used to generate signatures. The primary key can be set by calling `setPrimaryKey(primaryKeySpec)`.

Furthermore, an OpenPGP key can contain zero or more subkeys. Those can be set by repeatedly calling `addSubkey(subkeySpec)`.

OpenPGP keys are usually bound to User-IDs like names and/or email addresses. There can be multiple user-ids bound to a key, in which case the very first User-ID will be marked as primary. To add a User-ID to the key, call `addUserId(userId)`.

By default, keys do not have an expiration date. This can be changed by setting an expiration date using `setExpirationDate(date)`.

To enable password protection for the OpenPGP key, you can call `setPassphrase(passphrase)`. If this method is not called, or if the passed in `Passphrase` is empty, the key will be unprotected.

Finally, calling `build()` will generate a fresh OpenPGP key according to the specifications given.

Edit Keys

User-IDs

User-IDs are identities that users go by. A User-ID might be a name, an email address or both. User-IDs can also contain both and even have a comment.

In general, the format of a User-ID is not fixed, so it can contain arbitrary strings. However, it is agreed upon to use the Below is a selection of possible User-IDs:

```
Firstname Lastname (Comment) <email@address.tld>
Firstname Lastname
Firstname Lastname (Comment)
<email@address.tld>
```

PGPainless comes with a builder class `UserId`, which can be used to safely construct User-IDs:

```
UserId nameAndEMail = UserId.nameAndEmail("Jane Doe", "jane@pgpainless.org");
assertEquals("Jane Doe <jane@pgpainless.org>", nameAndEmail.toString()):

UserId onlyEmail = UserId.onlyEmail("john@pgpainless.org");
assertEquals("<john@pgpainless.org>", onlyEmail.toString());

UserId full = UserId.newBuilder()
        .withName("Peter Pattern")
        .withEmail("peter@pgpainless.org")
        .withComment("Work Address")
        .build();
assertEquals("Peter Pattern (Work Address) <peter@pgpainless.org>", full.toString());
```

Passwords

In Java based applications, passing passwords as `String` objects has the disadvantage[23] that you have to rely on garbage collection to clean up once they are no longer used. For that reason, `char[]` is the preferred method for dealing with passwords. Once a password is no longer used, the character array can simply be overwritten to remove the sensitive data from memory.

Passphrase

PGPainless uses a wrapper class `Passphrase`, which takes care for the wiping of unused passwords:

```
Passphrase passphrase = new Passphrase(new char[] {'h', 'e', 'l', 'l', 'o'});
assertTrue(passphrase.isValid());

assertArrayEquals(new char[] {'h', 'e', 'l', 'l', 'o'}, passphrase.getChars()):

// Once we are done, we can clean the data
passphrase.clear();

assertFalse(passphrase.isValid());
assertNull(passphrase.getChars());
```

[23] https://stackoverflow.com/a/8881376/11150851

Furthermore, `Passphrase` can also wrap empty passphrases, which increases null-safety of the API:

```
Passphrase empty = Passphrase.emptyPassphrase();
assertTrue(empty.isValid());
assertTrue(empty.isEmpty());
assertNull(empty.getChars());

empty.clear();

assertFalse(empty.isValid());
```

SecretKeyRingProtector

There are certain operations that require you to provide the passphrase for a key. Examples are decryption of messages, or creating signatures / certifications.

The primary way of telling PGPainless, which password to use for a certain key is the `SecretKeyRingProtector` interface which maps `Passphrases` to (sub-)keys. There are multiple implementations of this interface, which may or may not suite your needs:

```
// If your key is not password protected, this implementation is for you:
SecretKeyRingProtector unprotected = SecretKeyRingProtector
        .unprotectedKeys();

// If you use a single passphrase for all (sub-) keys, take this:
SecretKeyRingProtector singlePassphrase = SecretKeyRingProtector
        .unlockAnyKeyWith(passphrase);

// If you want to be flexible, use this:
CachingSecretKeyRingProtector flexible = SecretKeyRingProtector
        .defaultSecretKeyRingProtector(passphraseCallback);
```

`SecretKeyRingProtector.unprotectedKeys()` will return an empty passphrase for any key. It is best used when dealing with unencrypted secret keys.

`SecretKeyRingProtector.unlockAnyKeyWith(passphrase)` will return the same exact passphrase for any given key. You should use this if you have a single key with a static passphrase.

The last example shows how to instantiate the `CachingSecretKeyRingProtector` with a `SecretKeyPassphraseProvider` as argument. As the name suggests, the `CachingSecretKeyRingProtector` caches passphrases it knows about in a map. That way, you only have to provide the passphrase for a certain key only once, after which it will be remembered. If you try to unlock a protected secret key for which no passphrase is cached, the `getPassphraseFor()` method of the `SecretKeyPassphraseProvider` callback will be called to interactively ask for the missing passphrase. Afterwards, the acquired passphrase will be cached for future use.

Note: While especially the `CachingSecretKeyRingProtector` can handle multiple keys without problems, it is advised to use individual `SecretKeyRingProtector` objects per key. The reason for this is, that internally the 64bit key-id is used to resolve `Passphrase` objects and collisions are not unlikely in this key-space. Furthermore, multiple OpenPGP keys could contain the same subkey, but with different passphrases set. If the same `SecretKeyRingProtector` is used for two OpenPGP keys with the same subkey, but different passwords, the key-id collision will cause the password to be overwritten for one of the keys, which might result in issues. See `FLO-04-004 WP2` of the 2021 security audit[24] for more details.

---

[24] https://cure53.de/pentest-report_pgpainless.pdf

Most `SecretKeyRingProtector` implementations can be instantiated with custom `KeyRingProtectionSettings`. By default, most implementations use `KeyRingProtectionSettings.secureDefaultSettings()` which corresponds to iterated and salted S2K using AES256 and SHA256 with an iteration count of 65536.