Atbash Octopus JWT Support

Rudy De Busscher

Version 1.1.0, 20/08/2022

Table of Contents

User manual for Atbash JWT Support.

Release Notes

1.1.0

1. Support for Reading PEM in PKCS8 unencrypted format.
2. Support custom claims of type number array in JWT payload.
3. More alternative interpretations of claims (like comma separated string for aud interpreted as array)
4. Support for parsing Strings in the different Readers.
5. Convert 'Java Objects' to JsonValue instances and vice versa.
6. The *exp*, *iat*, and *nbf* claims can be set as `LocalDateTime` value using the `JWTClaimsSet.Builder`
7. Support for setting 'crit' header values that are handled by applicaiton (understood by `JWSVerifier` implementations). Possibility to pass on these header clams to `DefaultJWSVerifierFactory` ad `JWTDecoder`.
8. New method within the builder to define the expiration time as a duration from current time. `Builder.expirationTime(Duration)`.
9. Identify the `SecretKey` through the class `KeyFamilyUtil`.
10. The reason for a Failed JWT token verification (signature, verifier, ...) can be found within MDC of SLF4J with key `JWTValidationConstant.JWT_VERIFICATION_FAIL_REASON`.
11. Possibility to recreate the `SecureRandom` or developer can provide (implement the interface `SecureRandomProvider`) the instance that is retrieved by `JCASupportConfiguration.getSecureRandom`.
12. Do not allow un-encoded payload for JWT (RFC-7797, Section 7)
13. Define constants for Spec based values; see `HeaderParameterNames`, `JWKIdentifiers`, and `JWTClaimNames` (defined as PublicAPI)
14. Support for the flattened JWS JSON Serialization syntax (besides the compact String serialisation). Added method to `JWTEncoder` and `JWTDecoder` to support this.
15. Validations on Signatures using EC Curves to capture the CVE-2022-21449 (https://nvd.nist.gov/vuln/detail/cve-2022-21449)
16. Important changes / new features from Nimbus Jose library (since JWT Support classes are based on the Nimbus Jose classes)
    - Limit size of parsed headers (plain, JWS, ...) to 20 000 characters. (JWTDecoder throws an InvalidJWTException)
    - Support for RSA-OAEP-384 and RSA-OAEP-512.
    - EC Curve *P-256K* renamed to *SECP256K1*
    - `X509CertChainUtils.parse` method for File (and String)
    - `X509CertChainUtils.store` method to add certificates to Java KeyStore.

- *b64* header parameter is now supported (no longer use header.parameter to specify this value)
- Improved detection off missing values
- Updates KeyUse.from(X509Certificate) to return null when the certificate public key use doesn't map to a single JWK use value.

Breaking changes

There are many small changes, but the majority of the code should still compile without any change.

- `RSA_OAEP_256` is removed and replaced by `RSA_OAEP_2`
- Many classes no longer implement `Serializable`.
- Removed `getDeferredCriticalHeaderParams()` from `JWEDecrypter` and `JWTVerifier` implementations.
- The method `asJsonArray` is removed from `JSONObjectUtils`, and you should use `toJsonArray` method.

1.0.0

1. RemoteKeyManager
2. Support for Reading and Writing OCT JWK.
3. Support OCT keys in various places
4. PlainJWT support by JWTEncode and JWTDecoder.
5. Support for storing EC Key in KeyStore format
6. Support for reading public key from certificate of KeyStore format.
7. Updated BouncyCastle to the latest version.
8. Various small fixes and updates.
9. Configuration of SecureRandom (JVM default or BouncyCastle defined one)

Breaking changes

- be.atbash.ee.security.octopus.jwt.decoder.JWTDecoder.decode(java.lang.String, java.lang.Class<T>) has a changed return type.
- A subset of Nimbus JOSE is integrated in this JWT support module.

0.9.1

1. Improve usability KeyManager.
2. Support for creating JWE (Encrypted JWT from Java Object and vice versa)
3. Writing and Reading EC keys in JWK format.
4. Support for AtbashKey by JWTEncoder and JWTDecoder.

5. Support for jku header (JSON Key URL)
6. *discriminator* object for SelectorCriteria
7. Reading public key from certificate from KeyStore format.

Breaking changes

1. KeyManager.retrieveKeys has now **SelectorCriteria** as parameter.
2. KeyUse references are removed leading to some removal of methods, constructors or changed number of parameters.

Known issues

1. There are several issues and limitations when creating a KeyStore (only single private or public key, OCT/AES key only in PKCS12, etc ...)
2. Writing an EC Private Key to JWK will only be possible when private key is Bouncy Castle Based.

0.9.0

1. Support for reading and writing multiple formats (PEM, KeyStore, JWK and JWKSet)
2. Concept of KeySelector and KeyManager (with SPI)
3. Generating Cryptographic keys.
4. Key for HMAC uses now standards (SecretKey and OCT JWK)

Breaking changes (although 0.5.0 was an alfa release and thus changes are expected)

1. HMacSecret removed, use now *HmacSecretUtil.generateSecretKey()*.

0.5.0

1. First separate release from Octopus repository.

Requirements

Runs on JDK 8 and JDK 11 ClassPath.

??Unit testing fails on JDK 11 ClassPath.

Introduction

Code around handling all types of Cryptographic Keys (RSA, EC, OCT / AES secret Keys, Diffie Hellman), reading and writing it in different formats (PEM with PKCS1, PKCS8 and unencoded, JWK, JWKSet and KeyStore (JKS)).

It allows creating signed or encrypted JWT objects from a Java Object.

Standard usage JWT Serialization

Convert the Object *data* to JSON and Base64 encoded format.

```
    @Inject
    private JWTEncoder jwtEncoder;

    JWTParameters parameters = JWTParametersBuilder.newBuilderFor(JWTEncoding.NONE)
                .build();

    String encoded = jwtEncoder.encode(data, parameters);
```

Use the Object *data* as JWT payload, signed with a HMAC

```
    @Inject
    private JWTEncoder jwtEncoder;

    JWTParameters parameters = JWTParametersBuilder.newBuilderFor(JWTEncoding.JWS)
                .withHeader("Octopus Offline", "v0.4")

.withSecretKeyForSigning(HmacSecretUtil.generateSecretKey(LOCAL_SECRET_KEY_ID,
localSecret))
                .build();

    String encoded = encoder.encode(data, parameters);
```

In the above example, the JWT has a custom header.

Instead of creating a String, you can also create a JSON object using (flattened JWS JSON Serialization syntax https://datatracker.ietf.org/doc/html/rfc7515#section-7.2.2)

```
    JsonObject json = jwtEncoder.encodeAsJson(data, parameters);
```

Instead of injecting the Encoder, it is also possible to instantiate the encoder directly.

```
   JWTEncoder jwtEncoder = new JWTEncoder();
```

Use the Object *data* as payload within a JWE (first signed and then encrypted)

This examples uses an RSA for signing (private key) and encryption (public key).

```
    JWTParameters parameters = JWTParametersBuilder.newBuilderFor(JWTEncoding.JWE)
            .withSecretKeyForSigning(keyForSigning)
            .withSecretKeyForEncryption(keyForEncryption)
            .build();

    String encoded = new JWTEncoder().encode(data, parameters);
```

Converting the JWT (or Base64 encoded JSON) to an Object instance.

The following example converts a signed JWT or a JWE.

```
    @Inject
    private JWTDecoder jwtDecoder;

    @Inject
    private KeySelector keySelector;

    @Inject
    private MPBearerTokenVerifier verifier;

    JWTData<MPJWTToken> data = jwtDecoder.decode(token, MPJWTToken.class, keySelector,
verifier);
    MPJWTToken token = data.getData();
```

KeySelector → Selector of the key based on the id found in the header using a default (but can be configured) keyManager which looks for all keys defined based on some location (see *keys.location* and *key.manager.class* configuration parameters)

See also https://github.com/rdebusscher/atbash-key-server for a Key management server implementation.

MPBearerTokenVerifier → Optional verifier for validating the JWT.

You can now also provide a JSONObject with the flattened JWS JSON Serialization syntax to use the 'token'.

TODO : Describe the default verifications.

Additional verification

As indicated in the above example, an optional verifier, of type `be.atbash.ee.security.octopus.jwt.decoder.JWTVerifier` can be passed to the decode() method. The interface has the following verification method:

```
     boolean verify(CommonJWTHeader header, JWTClaimsSet jwtClaimsSet);
```

The header parameter has the header, either the header for the JWS (signed JWT) or the JWE (encrypted JWT). The jwtClaimsSet parameter contains the payload of the JWT.The claims value are still in basic format, so if there is an converter defined, it is not yet applied.

Validate jku

The `RemoteKeyManager` retrieves the JWKSet from the JKU values defined in the header of a signed/encrypted JWT.However, allowing any URI in this header is a serious security issue as anyone can create a JWT and point to an endpoint he controls. There, these URIs needs to be approved.

Create an instance of `RemoteJWKSetURIValidator` and implement the is valid method. The instance can be defined through the ServiceLoader mechanism or as CDI bean (when used in a CDI environment) Only those URIs which are denoted as valid will be used.

When no instances of `RemoteJWKSetURIValidator` are found, no URI will be valid.

Verify 'crit' header content

When you are processing a JWT containing a value for the `crit` header, the Application must indicate the header names it understands/process from the JWT before the token can be accepted.

The developer can define the header names that are accepted by the `crit` header:

1. When using the low-level method of creating a `JWSVerifier` through the `JWSVerifierFactory.createJWSVerifier()` method by specifying the names as the var-arg parameter.
2. When using the high-level `JWTDecoder.decode()` method also specifying the names as the var-arg parameter.
3. When using `JWTDecoder` you, you can also specifying the accepted critical header names through the `JWTVerifier` by overriding the default `getSupportedCritHeaderValues` method.

Creating JWE

Creating a JWE, the JWT variant which uses encryption, is very similar in creating a signed version. This is done on purpose so that creating a signed JWT or an encrypted JWT is similar and developers don’t need to learn different patterns.

```
String encoded = new JWTEncoder().encode(payload, parameters);
```

The `JWTEncoder` class can also be used for creating a JWE. The difference is the parameter we supply, when encoding. The following example shows the minimal required parameters.

```
JWTParameters parameters = JWTParametersBuilder.newBuilderFor(JWTEncoding.JWE)
        .withSecretKeyForSigning(signKey)
        .withSecretKeyForEncryption(encryptKey)
        .build();
```

When creating a JWE, first a signed JWT is created and then a encryption is done.

The signing key can be of type RSA, EC, or AES. The only requirement is that when decoding the corresponding key is present in the KeyManager.

Keys

Retrieving a certain Cryptographic key can be performed by the class *KeySelector* through the methods *selectAtbashKey* and *selectSecretKey*. You give it a few criteria (like key id, key type like RSA, EC, etc ..) and it tries to find the key which correspond to these values.

If it finds no key or multiple keys which match these criteria, you will see a warning in the log and the method returns null. Which will obviously result in a error in the calling method because it probably needs a key.

Most of the time, you give it the key id (and if you like the private of public part, see also further on why this is important) but you could also decide that the library can take the only available private key it knows of for creating the signature for instance.

You supply the criteria to match, through a *SelectorCriteria* when can be created by a Builder pattern.

The filtering is performed in this order - id - secret key type (combination of type like RSA and part like private part) - key type (RSA, EC, ...) - part (is possible)

The *KeySelector* however, is only responsible for selecting the matching key from the 'pool'. Managing (reading) the 'pool' is the responsibility of the *KeyManager*.

Providing the keys

The *KeyManager* is responsible for supplying the requested key to the *KeySelector*. (when verifying signature, when decrypting and so on)

There is a default *KeyManager* available called **LocalKeyManager** which is active when no specific configuration value is set (see further on or the configuration section in this manual). It is capable of reading a specific file with keys, or directory with several key files.

If you need another implementation, like reading it from a central managed storage or reading keys in a more dynamic way, you can implement the interface *be.atbash.ee.security.octopus.keys.KeyManager*.

In a CDI environment, you can define your custom *KeyManager* as a CDI bean and it will be picked up. An other option is to define the class as parameter value of configuration parameter

key.manager.class. It must contain the fully qualified class name. It is just instantiated, so no CDI features (unless you use programmatic retrieval of beans) will be available.

The keys are read by the *KeyReader* which has support for PEM, JKS (Java KeyStore), JWK, and JWKSet. (see Supported Key Formats)

Define Key type

By default, based on the file extension, the type is determined and how it should be read. There is a default implementation which makes the following mapping

- .jwk → JWK
- .jwke → JWK (Atbash encrypted JWK)
- .jwks → JWKSet
- .jwkset → JWKSet
- .jwksete → JWKSet (Atbash encrypted JWKSet)
- .pem → PEM
- .der → PEM
- .jks → KeyStore
- .p12 → KeyStore
- .pfx → KeyStore

When you want to use other suffixes, implement the interface *be.atbash.ee.security.octopus.keys.reader.KeyResourceTypeProvider* and define the class name as parameter value of key *key.resourcetype.provider.class*.

The return value of the interface method *determineKeyResourceType* will determine how the resource will be read. Returning null means that the type is unknown

Supply passwords

Various types have encrypted storage of private keys (as they have a sensitive nature).

TODO Specify how *ConfigKeyResourcePasswordLookup* reads the password from the configuration.

Discriminator in SelectorCriteria

Some *KeyManager* support some kind of separation of the keys (like tenants). The KeyManager implemented in the [Key Manager Server](https://github.com/atbashEE/atbash-key-server) for example has such a separation.

Therefor we can pass some kind of *discriminator* object to the SelectorCriteria. This discriminator (a string, a POJO, whatever will be understood by the KeyManager) will then be used by it to distinct keys.

Reading any format

As described in above sections, the suffix of the file which is read, determines the format of the file and determines how the file is read by the code.

However, there are situations where the format is not know, like in the case when you read some keys from an URL. For that use case, the *keyReader* has the *readKeyResource* method. it takes an *InputStream* and tries to read the contents using different formats until one of them returns something valid. The method has 2 overloaded versions, one taken the *InputStream* and the other one an *URI*.

The order in which the formats are tried on the input can be configured. With the configuration parameter `jwt.reader.order`, one can define the order in which the formats are tried.

The *tryToReadKeyResource* method also tries to read the resource but the order is determined by the MicroProfile JWT specification; PEM, JWK, KeyStore and as last format the JWKSet.

The *tryToReadKeyContent* method takes a String as parameter but performs the same logic as the *tryToReadKeyResource*.

Customization

Password are by default read from configuration (parameter *lookup.password.class* define the class)

KeyResourceTypeProvider → defines mapping between file extension and type of key (PEM, JKS, ...)

Filter Keys

The preferred way to retrieve a Key from some source is the use of the *KeyManager* as described above. However, there are situations where you want to select a certain key from a set of keys.

The classic use case is that you have generated a key pair (using the *KeyGenerator* class) but want to retrieve the public key from this pair. since the generated key pair is not available within the KeyManager, another way must be available to retrieve the key (which is more high level then just checking which key implements the PrivateKey interface)

The **ListKeyManager** is created specially for this purpose. It is also a *KeyManager* implementation but you supply the set of keys it own through the constructor (and thus the KeyManager doesn't retrieve his keys from the environment in some way.

The following snippet retrieves the public key.

```
ListKeyManager keyManager = new ListKeyManager(keys);
SelectorCriteria criteria =
SelectorCriteria.newBuilder().withAsymmetricPart(AsymmetricPart.PUBLIC).build();
List<AtbashKey> keyList = keyManager.retrieveKeys(criteria);
```

Supported Key Formats

```
PEM
   PKCS8
   PKCS1
JWK
JWKSet
Java KeyStore
```

Generating Cryptographic key

With the class **be.atbash.ee.security.octopus.keys.generator.KeyGenerator** you are able to generate cryptographic keys.

It is a CDI bean and can be injected into other CDI controlled classes. But you can also create a new instance of this class when you need to generation capabilities in other environments.

By calling the method

```
generateKeys(GenerationParameters);
```

It will give you one or more keys, depending on the type you requested. For asymmetric keys, like RSA keys, you get the corresponding public and private keys.

The kind of keys which are generated, depend on the parameter you supply, which can be created using a builder pattern. The following example gives you the code for generating an RSA key.

```
RSAGenerationParameters generationParameters = new
RSAGenerationParameters.RSAGenerationParametersBuilder()
        .withKeyId("the-kid")
        .build();
List<AtbashKey> atbashKeys = generator.generateKeys(generationParameters);
```

The default size of the generated key is 2048, but you can define it using the method *.withKeySize* on the builder.

There are also builders for Elliptic Curve (EC) and Octet sequence (used in MAC and symmetric encryption) available.

The key id is always required and for EC keys, the curve name is also required.

The generation is performed by the JVM classes itself in case of the RSA and OCT keys, and BouncyCastle in the case of EC keys (*ECDSA* algorithm)

Supported EC Curve names

This is the list of the supported curves (through BouncyCastle)

- prime192v1
- prime192v2
- prime192v3
- prime239v1
- prime239v2
- prime239v3
- prime256v1
- secp192k1
- secp192r1
- secp224k1
- secp224r1 / P-224
- secp256k1
- secp256r1 / P-256
- secp384r1 / P-384
- secp521r1 / P-521

Writing Cryptographic key

With the class **be.atbash.ee.security.octopus.keys.writer.KeyWriter** you can convert an *AtbashKey* into one of the supported formats. See ??? for a list of the formats.

This class can also be used as CDI bean, and injected into other CDI artifacts, or used in other environments by instantiating it yourself.

With the method *writeKeyResource*, you can convert the *AtbashKey*, containing the Cryptographic key, into one of the formats as defined by the parameter **KeyResourceType**.

There are 2 variants for the method. One contains a String value defining the location where it key will be stored, the other one just returns the result as a byte array.

When the target file already exists, and the format supports multiple formats (like Java Key Store - JKS and JWK Set) the file is updated with the key you want to write.

The 2 last parameters are used depending on the type of the format.

- keyPassword: Used for encryption of the key, when this is used in the format or requested through a config parameter (for PEM format for example)
- filePassword: Used for the encryption of the file when file is encrypted as a whole (like with Java Key store). This password is also used for reading it first if the file already exists.

When you write out a private Key in the JWK format, a key password is required. All sensitive JSON values are encrypted and written as 'enc' to the JWK. Reading such a format requires the same password to retrieve the keys from it. This is an alternative for the signing and encryption of the JWK, but ony supported by Atbash.

If you do not want to create this encrypted format of Atbash, set the parameter `jwt.jwk.encrypted` to false.

When you convert a JWK with the `JWTEncoder`to JSON, the conversion is not performed as there is no possibility to supply a password.

Define Serializer

When a custom JSON serializer is required

- Create a class implementing `javax.json.bind.serializer.JsonbSerializer`
- Define the class (FQCN) through the ServiceLoader mechanism (`META-INF/service/javax.json.bind.serializer.JsonbSerializer`)

When the class has Jackson JSonProperty annotation on the fields, the `AbstractJacksonJsonSerializer` can be used to create a serializer without the need for coding.

Define a Class as this and define the FQCN in the service loader file.

```
public class TestAbstractJacksonJsonSerializer extends
AbstractJacksonJsonSerializer<MainClass> implements JsonbSerializer<MainClass> {
}
```

Define Deserializer

When a custom JSON deserializer is required

- Create a class implementing `javax.json.bind.serializer.JsonbDeserializer`
- Define the class (FQCN) through the ServiceLoader mechanism (`META-INF/service/javax.json.bind.serializer.JsonbDeserializer`)

Conversion from and to JsonValue (Since 1.1)

With the help of the `JSONObjectUtils` class, you can convert a String to the JsonString instance, JsonNumber to a Long etc. So you can convert the Java objects to JsonValue instances and vice versa.

The code was already partially available in previous releases but now finalized and usable by the developers (marked as PublicAPI).

- `JSONObjectUtils.getJsonValueAsObject` extracts the basic Java value from any `JSONValue` instance.

An array is converted to a List.

- `JSONObjectUtils.getAsJsonValue` wraps the Java value in a `JSONValue` instance. Any Collection is converted to an array, and Map to a JsonObject.

SecureRandom

The class `JCASupportConfiguration` has the option to customize a `SecureRandom` instance centrally that is used by the Key Generators, code that perform signing, etc..., but also that you can use within your application. (use `JCASupportConfiguration.getInstance().getSecureRandom()`).

As of version *1.1*, you can

- Recreate the SecureRandom instance after some time (see configuration parameter *jwt.jca.securerandom.recreate*)
- Provide an instance through the Java Service Loader mechanism if you want to fine tune the seeding for example. Implement the `SecureRandomProvider` interface and define a Service Loader configuration file with as content the class name.

Already available in older versions of the library, configure the algorithm that is used by specifying the configuration parameter *jwt.jca.securerandom.algo*.

Configuration

key.manager.class

default : **be.atbash.ee.security.octopus.keys.LocalKeyManager**

The *KeyManager* that supplies the request key to the KeySelector. Other implementations which are supported by default are **be.atbash.ee.security.octopus.keys.RemoteKeyManager** (to support the jku header claim) and **be.atbash.ee.security.octopus.keys.CombinedKeyManager** which combines the functionality of LocalKeyManager and RemoteKeyManager.

keys.location

default : **none**

The location of the Cryptographic keys.The value must start with **classpath:**, **file:** or **url:**.When the value points to a directory, all files (which are identified as containing a key, see *KeyResourceTypeProvider*) are read.

key.resourcetype.provider.class

default : **be.atbash.ee.security.octopus.keys.reader.DefaultKeyResourceTypeProvider**

Defines the class which determines the key format of the resources.

lookup.password.class

default : **be.atbash.ee.security.octopus.keys.reader.password.ConfigKeyResourcePasswordLookup**

Defines the class which supplies the passwords for encrypted storage of keys

key.pem.encryption

default : **PKCS8**

Defines the encryption of the (RSA only?) private key when written to a PEM formatted file. Valid values ora PKCS1, PKCS8 and NONE (meaning no encryption of the key at all and just BASE64 encoded)

key.pem.pkcs1.encryption

default : **DES-EDE3-CBC**

Defines the default PKCS#1 encryption used. Valid values are defined according the following rules.

```
algorithm names composed from 3 parts glued with hyphen.
The first part determines algorithm, one of AES, DES, BF and RC2.
The second part determines key bits and is used for AES and optionally for RC2.
   For AES it is possible to use values 128, 192 and 256.
   For RC2 64, 40 can be used or nothing - then value 128 is used.
The last part determines the block mode: CFB, ECB, OFB, EDE and CBC.
Additionally EDE3 can be used in combination with DES to use DES3 with EDE.
```

```
Examples:
  AES-192-ECB
  DES-EDE3.
```

key.store.certificate.x500name

default : **CN=localhost**

Defines the x500 name for the generated certificate when storing a public key into a Key store file.

key.store.signature.algo.RSA

default : **SHA1WithRSA**

Defines the algorithm used for singing the certificate which is generated when storing a RSA public key into a Key store file. Please note that there will be always an RSA key generated for this, so one is only able to change the hashing part of the signature.

key.store.signature.algo.EC

default : **SHA384withECDSA**

Defines the algorithm used for singing the certificate which is generated when storing a EC public key into a Key store file. Please note that there will be always an EC key generated for this, so one is only able to change the hashing part of the signature.

key.store.type

default : **PKCS12**

The default type of the created keyStores. This overwrites the default which is set by the JRE config.

jwt.sign.rsa.algo

default : **RS256**

Defines the algorithm used for signing the JWT in case we use RSA keys. Valid values are *RS256*, *RS384*, *RS512*, *PS256*, *PS384*, and *PS512*.

jwt.clock.skew.secs (since 1.0.0)

default : **60**

Defines the clock skew value for verifying expiration dates of JWT tokens.

jwt.jwe.algorithm.default.EC (since 1.0.0)

default : **ECDH-ES+A256KW**

Defines the default encryption method when a JWE is created without explicitly defining the method. For valid values, see `be.atbash.ee.security.octopus.nimbus.jwt.jwe.JWEAlgorithm.Family.ECDH_ES`

jwt.jwe.algorithm.default.OCT (since 1.0.0)

default : **A256KW**

Defines the default encryption method when a JWE is created without explicitly defining the method. For valid values, see `be.atbash.ee.security.octopus.nimbus.jwt.jwe.JWEAlgorithm.Family.AES_KW`

jwt.remote.jwk.cache.period

default : **24h**

Defines the expiration period of the remote JWKSet data read from a `jku header claim. After that

period, the data is discarded and needs to be reread from the URI.

The value has the following format

<v><unit>

- v : A positive integral number
- unit : s (seconds), m (minutes) or h (hours)

jwt.jca.securerandom.algo

default : **none**

Algorithm name for the SecureRandom implementation.

When no value is set, JVM configured one will be taken.

jwt.jca.securerandom.recreate

default : **0**

The time in seconds to recreate the SecureRandom instance. 0 means the instance is never recreated. Don't use a value that is too low as it can have an impact on performance when you recreate the instance every second for example.

jwt.jwk.encrypted

default : **true**

Are private Keys written by the `KeyWriter` 'encrypted' using the Custom Atbash format?

jwt.reader.order

dafault : **JWKSET, JWK, PEM, KEYSTORE**

Defines the order in whioch the Key formats are tried to read a certain 'input stream'. Values are the enum names of `KeyResourceType` (case insensitive)