

Atbash Octopus JWT Support

Rudy De Busscher

Version 0.9.1, ??/??/2018

Table of Contents

Release Notes	1
0.9.1	1
0.9.0	1
0.5.0	1
Introduction	1
Standard usage JWT Serialization	2
Creating JWE	3
Keys	4
Providing the keys	4
Define Key type	5
Supply passwords	5
Discriminator in SelectorCriteria	5
Customization	5
Filter Keys	6
Supported Key Formats	6
Generating Cryptographic key	6
Supported EC Curve names	7
Writing Cryptographic key	8
Configuration	8
key.manager.class	8
keys.location	8
key.resourcetype.provider.class	8
lookup.password.class	9
key.pem.encryption	9
key.pem.pkcs1.encryption	9
key.store.certificate.x500name	9
key.store.signature.algo	9
jwt.sign.rsa.algo	10

User manual for Atbash configuration.

Release Notes

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

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.

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.

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 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)

The default KeyManager now also supports the *jku* header. When available, the URL is called to retrieve the keys (in the format of a JWK Set as defined in the specification). In this scenario, the parameter *keys.location* is not required. The current version is a draft implementation and will be improved in the future (so that rotating keys are better supported and caching of those keys).

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

MPBearerTokenVerifier → Optional verifier for validating the JWT.

TODO : Describe the default verifications.

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. It can even read from an URL resource, but will be used most of the times to read it from a local (from the point of the server process) location like the classpath or file and directory.

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.

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 snippets 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 Octec 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.

Configuration

key.manager.class

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

The *KeyManager* that supplies the request key to the KeySelector.

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 are 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 to 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

default : **SHA1WithRSA**

Defines the algorithm used for signing the certificate which is generated when storing a 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.

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*.