
*Secure Provisioning of Cloud Services*
*based on SLA Management*

## SPECS Project - Deliverable 4.4.2

# The Credentials Service Components – Finalized

Version no. 1.1
31 January 2016


The activities reported in this deliverable are partially supported
by the European Community's Seventh Framework Programme under grant agreement no. 610795.

# Deliverable information

| | |
|---|---|
| Deliverable no.: | D4.4.2 |
| Deliverable title: | The Credentials Service Components – Finalized |
| | |
| Deliverable nature: | Prototype |
| Dissemination level: | Public |
| | |
| Contractual delivery: | 31 October 2015 |
| Actual delivery date: | 31 January 2016 |
| | |
| Author(s): | Silviu Panica (IeAT), Massimiliano Rak (CeRICT), Valentina Casola (CeRICT) |
| Contributors: | Jolanda Modic (XLAB), Damjan Murn (XLAB), Alessandra De Benedictis (CeRICT) |
| Reviewers: | Umberto Villano (EMC), Dana Petcu (IeAT) |
| Task contributing to the deliverable: | T4.4 |
| Total number of pages: | 34 |

## Executive summary

This deliverable is a second version of the deliverable reporting the status of implementation of two of the SPECS vertical layers devoted to secure interactions among internal and external SPECS components. As already reported in the previous version, in SPECS the secure interaction mechanisms specifically tackle two research challenges encountered when dealing with cloud application security, namely service-to-service authentication (and in future possibly also authorization) and credential data management. These challenges, although not visible to the End-user, are cornerstones of the overall SPECS solution, and they are integral part of the Vertical Layer services offered by the SLA Platform, as described in D1.1.3 and D1.4.1.

In particular, the service-to-service authentication is covered by the SPECS Security Tokens mechanism, and provides a way to embed in the requests (usually into HTTP requests via headers) the data needed to assess the identity of the calling service, and also required to apply access policies by the called service. The Security Tokens service component was already available in the previous version of this deliverable. In this second iteration of the document we will mainly report documentation about the final API offered, the involved data model, testing results and new usage examples.

The credential data management is covered by the SPECS Credential Service mechanism, which provides a REST API that enables the execution of the required security protocol operations over the credential data, without exposing the raw data to the requester. In the first year, we implemented a preliminary service and discussed an incremental design to be developed in the second year, in order to cover all security requirements. In this second iteration of the document, we will present the improved architecture and all implementation details that are based on open source solutions provided by the *Vault project*, whose primary goal is to provide secure secret storage and API to manage credentials, keys, revocations, and so on. On the basis of the current implementation, we will also provide a revised validation scenario for credential management, which substitutes the one reported in D5.1.2 and which will be used for integration.

# Table of contents

# Index of figures

# Index of tables

# 1. Introduction

As stated in the previous version of this deliverable, this document focuses on secure interaction mechanisms provided by the Vertical Layer and devoted to protecting the internal communications among SPECS components and the access to external providers. In particular, the following mechanisms have been introduced:

- the **Security Tokens mechanism**, implemented by the Security Tokens component of the Vertical Layer and securing all internal communications among SPECS components by providing authentication and authorization features, and
- the **Credential Service mechanism**, implemented by the Credential Service component of the Vertical Layer and protecting the interactions among SPECS components and external CSPs by providing a means to securely store and share the needed access credentials.

The Security Tokens mechanism has been designed and implemented with the aim of authenticating and authorizing all the HTTP requests issued by SPECS components and targeted to the invocation of the REST APIs offered by other SPECS components (referred to as the *target* components).

The Credential Service mechanism targets the communications between SPECS and the external CSPs that host the target services by managing the set of credentials needed to access related resources. The mechanism has been designed to be able to cope with existing security protocols and, in particular, with existing CSP authentication schemes.

Scenarios, requirements and design of both secure interaction mechanisms are discussed in deliverables D5.1.2, D4.1.2 and D4.2.2, respectively. In the previous version of this deliverable, we focused on implementation, installation, and usage of the Security Tokens mechanism and on the implementation of a preliminary architecture of the Credential Service. In particular, we provided a complete implementation of the Security Tokens mechanism able to cover the full list of associated requirements, and we presented the prototype of the initial Credential Service mechanism. Furthermore, we discussed how to enhance the architecture and the implementation of the Credential Service, in order to fulfil the uncovered requirements.

Before going into the details of the secure interaction mechanisms, Section 2 discusses the relationships among this deliverable and the others. Then, we discuss for both mechanisms the overall status of development (including design validation), the organization of repositories and the final prototypes in terms of design, implementation, installation, usage, and testing. This information can be found in Sections 3 and Section 4 for the Security Tokens and the Credential Service mechanisms, respectively. A complete description of the APIs that the mechanisms offer, namely the *Security Tokens API* and the *Credential API*, is provided in Annex A and B, respectively. Moreover, Annex B also reports a revised version of the validation scenario for credential management, which is based on the current implementation and substitutes the one reported in D5.1.2. This scenario will be used for integration tasks.

## 2. Relationship with other deliverables

This deliverable reports on the final design and implementation of the secure interaction mechanisms, namely Security Tokens and Credential Service, whose previous version was described in D4.4.1. The requirements for these mechanisms were originally discussed in D4.1.2, while D4.2.2 provided the related preliminary design. Scenarios related to such mechanisms have been discussed in D5.1.2. As anticipated, we further refined the scenario related to Credentials Management, included in Annex B. This scenario will be used by integration Task T1.5.

Figure 1 presents the above-mentioned dependencies.

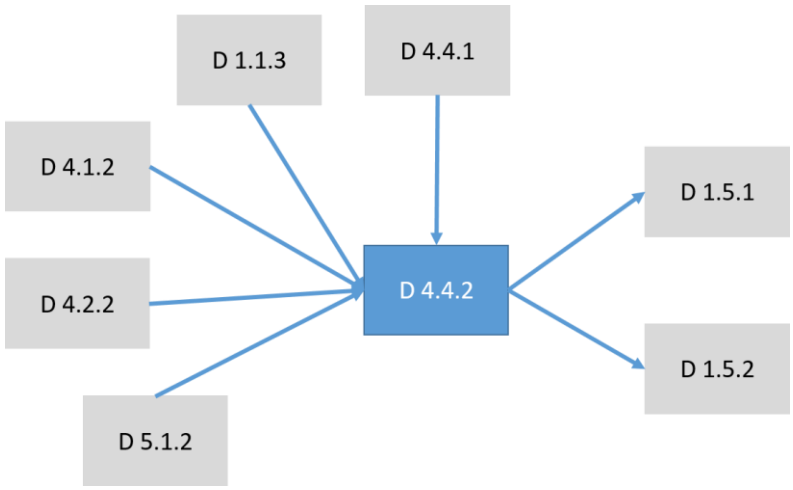


**Figure 1. Relationships with other deliverables**

# 3. The Security Tokens mechanism

The Security Tokens mechanism, a solution fully developed within the SPECS project, is responsible for protection of internal interactions among SPECS components. It provides authentication and authorization of all REST API requests among components of the SPECS framework, as already described in Section 1.
As defined in deliverables D4.1.2 and D4.2.2, and further elaborated in D4.4.1, the Security Tokens mechanism provides infrastructure for exchanging authentication/authorization data and arbitrary attributes in service-to-service interaction. The mechanism is implemented by the Security Tokens component of the SPECS Vertical Layer and includes the following two components:

- **Security Tokens Service**: a RESTful Web service that issues security tokens and maintains token revocation list;
- **Security Tokens Client**: a Java library that provides support for obtaining, parsing, and validating security tokens.

Note that the design of the mechanism has not changed during the second year of the project.

As discussed in D4.4.1, the structure of the Security Tokens mechanism is based on the following technologies: JSON Web Token [1], JSON Web Encryption [2] and JSON Web Signature [3]. Further details are provided on the SPECS Security Tokens mechanism's Bitbucket Wiki [4].

In the next subsection, we briefly summarize the behaviour of the mechanism, report the status of development activities, discuss APIs, present organization of repositories for source code, provide with guidelines for installation and usage, and present the tests executed for verifying the quality of the code.

## *3.1. Behaviour*

All REST API calls among components of the SPECS framework are authenticated and authorised with Security Tokens mechanism, as described below.

The SLA Enabling platform deploys a client certificate required for establishing HTTPS connection on each node (i.e., for each component of the SPECS framework). When setting up the SPECS framework, a security token is obtained for each SPECS component by sending a request to the Security Tokens Service component. The request contains the ID of the target SPECS component. The Security Tokens Service component generates the security token for the SPECS component, which contains the following information:

- component ID;
- component name;
- component IP address;
- list of services the token is eligible to access.

The token is signed by the SPECS certificate authority. Then the Security Tokens Service component returns the token to the SPECS component, which stores it in the local token vault.

When one SPECS component makes calls to another component of the SPECS framework, it attaches the security token to the request. When making REST API calls, the security token is

put in the HTTP header named X-AUTH-TOKEN. All communication among components is encrypted by using secure HTTPS protocol.

The target SPECS component that receives a request with a security token, validates the token and decodes it. Using the information from the token, the authorization engine makes the access decision based on the predefined policy.

## *3.2. Status of development activities*

The development of the mechanism was concluded at M12, as we already reported in D4.4.1. The following table presents the coverage of requirements associated to Security Tokens mechanism.

| Requirements for Security Tokens mechanism | SPECS Component | |
|---|---|---|
| | Security Tokens Client | Security Tokens Service |
| *ENF_TOK_R1* | X | |
| *ENF_TOK_R2* | X | |
| *ENF_TOK_R3* | | X |
| *ENF_TOK_R4* | X | |
| *ENF_TOK_R5* | | X |
| *ENF_TOK_R6* | | X |
| *ENF_TOK_R7* | | X |
| *ENF_TOK_R8* | X | |

**Table 1. Security Tokens components and related requirements**

There are 8 requirements associated to Security Tokens mechanism and the final prototype covers all of them. The following table presents a validation of the mechanism's design in terms of explaining how each requirement associated to the mechanism has been covered.

| REQ_ID | Requirement | Description |
|---|---|---|
| *ENF_TOK_R1* | ***Support offline token validation*** | Security tokens are digitally signed with the Security Tokens Service private key. The digital signature can be validated offline using the Security Tokens Service certificate and thus ensuring the token's authenticity and integrity. |
| *ENF_TOK_R2* | ***Send tokens in HTTP header*** | Security tokens are designed to be as small as possible. The token's payload is compressed using the GZIP algorithm. Typical size of an encoded token is 500 - 1000 bytes, depending on the payload size. The HTTP protocol does not define any size limit for HTTP headers, but the typical size limit on common web and application servers is 8192 bytes. |
| *ENF_TOK_R3* | ***Obtain security tokens issued by a centralized service*** | The Security Tokens Service is a centralized service that provides REST API for issuing security tokens, revoking tokens and retrieving token revocation list. The security token contains set of claims about the specified subject. Security-tokens-client is a Java library, which provides Java API for obtaining security tokens from the Security Tokens Service. |
| *ENF_TOK_R4* | ***Request, parse and validate tokens*** | The Java library security-tokens-client provides Java API for requesting, decoding and validating security |

| | | |
|---|---|---|
| | | tokens. Tokens are validated offline by verifying their digital signature using the Security Tokens Service certificate. The security-tokens-client maintains a local token revocation list and periodically updates it by downloading the token revocation list deltas from the Security Tokens Service. |
| *ENF_TOK_R5* | ***Revoke tokens*** | The Security Tokens Service provides REST API for revoking security tokens. A token is marked as revoked in the database, and is added to the token revocation list from where it is propagated to the clients. The revoked token remains in the database till the token expiration date extended by some safety time interval. |
| *ENF_TOK_R6* | ***Generate token revocation lists*** | The Security Tokens Service periodically generates token revocation list, which contains a list of revoked but not expired tokens. There are two types of revocation lists: full revocation list and delta revocation list. The full revocation list contains all revoked tokens that are not yet expired at the time of revocation list creation. The delta revocation list contains only the tokens that have been revoked since the last delta list generation. |
| *ENF_TOK_R7* | ***Sign tokens*** | The Security Tokens Service digitally signs tokens with its private key. The digital signature ensures the token's authenticity and integrity. The digital signature can be verified using the Security Tokens Service certificate (public key). |
| *ENF_TOK_R8* | ***Decode tokens*** | The Java library security-tokens-client provides Java API for decoding and validating security tokens. Tokens are decoded by splitting them into three parts: header, payload and signature. After the token is validated, the payload is decoded from the Base64 encoding scheme and decompressed. |

**Table 2. Requirements for the Security Tokens mechanism**

The current development status is summarized in Table 3.

| Mechanism | Artifacts under development | Status |
|---|---|---|
| Security Tokens | component: security-tokens-service | Completed |
| | component: security-tokens-client | Completed |
| | component: security-tokens-core | Completed |

**Table 3. Security Tokens implementation status**

The prototype of the mechanism is available on the project's Bitbucket repository [4]. Its organization is presented in the following subsection.

### *3.3. Repository*

The Security Tokens mechanism is implemented as a Maven-based Java project with three modules:

- security-tokens-core
- security-tokens-service
- security-tokens-client

The module security-tokens-core provides common functionality and class definitions that are shared among all modules. The module security-tokens-service is a Web application that exposes the *Security Tokens API,* whose complete documentation is provided in Annex A. The module security-tokens-client is a Java library that provides functionality for obtaining, validating and decoding security tokens. The source code of the Security Tokens mechanism can be found on the project's BitBucket repository at [5].

## *3.4. Installation*

Since the implementation of the mechanism has not changed in the second year of the project, the following installation guides are the same as reported in the first iteration of this deliverable.

The Security Tokens Service is a Java Web application and has to be deployed on the servlet container. The Security Tokens Client is a Java libray and is intended to be used by other applications / SPECS components as a dependency, and does not need any installation.

Prerequisites:

- Servlet container or J2EE application server (e.g., Apache Tomcat [6]).
- Relational database (e.g., MySQL Server [7]).

The installation guide is tailored for the Apache Tomcat and MySQL server.

### 3.4.1. Apache Tomcat configuration

The Security Tokens Service should be accessible only through secure (HTTPS) connection with client certificate-based authentication. In order to enable it, the Tomcat configuration file server.xml[1] must be edited to set up SSL enabled connector's property `clientAuth` to `true`:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
   maxThreads="150" scheme="https" secure="true"
   clientAuth="true" sslProtocol="TLS"
   keystoreFile="/etc/specs/security-tokens-service/sts-server.jks"
   keystorePass="***"
   keyAlias="sts"
   truststoreFile="/etc/specs/security-tokens-service/cacerts.jks"
   truststorePass="***"
   ciphers="SSL_RSA_WITH_RC4_128_SHA" />
```

The key store file sts-server.jks contains the Security Tokens Service (STS) server certificate. The trust store file cacerts.jks contains the CA certificate used to validate client certificates. Both files must be stored into the /etc/specs/security-tokens-service directory.

The certificate can be imported into the key store by using the following commands:

```
openssl pkcs12 -export -name sts-server -in sts-server.crt -inkey sts-server.key
-out sts-server.p12
```

[1] The default path for Tomcat 7 in Ubuntu is /etc/tomcat7/server.xml.

```
keytool -importkeystore -destkeystore sts-server.jks -srckeystore sts-server.p12
-srcstoretype pkcs12 -alias sts-server
```

### 3.4.2. Installing the Security Tokens Service

In order to install the Security Tokens service, the package security-tokens-service.tar.gz must be downloaded from the SPECS maven repository[2] and extracted in a local folder:

```
tar xzvf security-token-service.tar.gz
```

The package contains:

- a configuration file: sts-config.xml
- a database schema: sts-schema.sql
- a WAR file: security-tokens-service.war

The security-tokens-service.war Web application archive must be deployed in the Tomcat webapps directory (/var/lib/tomcat7/webapps). Then, the configuration file sts-config.xml must be copied to the /etc/specs/security-tokens-service directory and a database for the Security tokens service must be created by using the file sts-schema.sql. This last operation is illustrated in the following section.

### 3.4.3. Setting up the database

To create the database `sts` and the user with appropriate privileges on that database, the following commands must be run in the MySQL shell:

```
SOURCE sts-schema.sql
CREATE USER 'sts'@'localhost' IDENTIFIED BY 'somepass';
GRANT SELECT, INSERT, UPDATE, DELETE ON sts.* TO 'sts'@'localhost';
```

The database connection settings are located in the file /var/lib/tomcat7/webapps/WEB-INF/classes/META-INF/persistence.xml:

```
<property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
<property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost/sts"/>
<property name="javax.persistence.jdbc.user" value="sts"/>
<property name="javax.persistence.jdbc.password" value="somepass"/>
```

### 3.4.4. Configuring the Security Tokens Service

As said, the application configuration file must be copied to /etc/specs/security-tokens-service/. The file is structured as follows:

```
<config>
  <signing>
    <signerName>specs-demo</signerName>
    <signingKeyStore>
      <keyStoreFile>/etc/specs/security-tokens-service/sts-
signing.p12</keyStoreFile>
      <keyStorePass>password</keyStorePass>
      <signingCertFingerprint>01:18:BD:FE:5A:AF:DC:64:21:F5:07:93:7C:87:
        50:F6:5E:4C:75:B0</signingCertFingerprint>
```

[2] https://nexus.services.ieat.ro/nexus/content/repositories/specs-snapshots/eu/specs-project/utility/security-tokens/security-tokens-service

```
      <signingPrivateKeyPass>password</signingPrivateKeyPass>
    </signingKeyStore>
  </signing>
</config>
```

The section `signingKeyStore` specifies the signing certificate used to sign the security tokens. To obtain the certificate and to store it into a PKCS12 keystore, the following command must be executed:

```
openssl pkcs12 -export -name sts-signing -in sts-signing.crt -inkey sts-signing.key -out sts-signing.p12
```

The settings `keyStoreFile` and `keyStorePass` in the configuration file contain the file path and password of the key store file. The setting `signingCertFingerprint` contains the SHA1 fingerprint of the signing certificate, which can be determined by running the following command:

```
keytool -list -keystore sts-signing.p12 -storepass somepass -storetype PKCS12 -v
```

## *3.5. Usage*

The SPECS Security Tokens mechanism contains a library *security-tokens-client* that provides support to SPECS components for using security tokens. There are two use cases:

- Obtaining a security token.
- Decoding and validating a security token and retrieving required information from it.

### 3.5.1. Obtaining a Security Token

In the first use case, a SPECS component has to obtain a security token to call some other SPECS service that requires it (to authorize the request, or to get some information from the token needed to fulfil the request). For example, the Planning component calls the Implementation component using security tokens. For this use case, the *security-tokens-client* library provides class `SecurityTokensRetriever` with the following API:

```
public SecurityTokensRetriever(
   String stsAddress,
   String trustStoreFile, String trustStorePass,
   String keyStoreFile, String keyStorePass);

public Token obtainToken(String subject);
```

The `SecurityTokensRetriever` is built by using the following parameters:

- `stsAddress`: address of the STS.
- `trustStoreFile`, `trustStorePass`: trust store file path and password. The trust store contains the Certification Authority (CA) certificate chain (the issuer of the STS server certificate) or the STS server certificate. The trust store is needed to validate the STS certificate when establishing the secure connection with the server.
- `keyStoreFile`, `keyStorePass`: key store file path and password. The key store contains the client certificate and private key which are needed to authenticate to the STS.

The method `obtainToken` accepts the parameter `subject`. It calls the STS and requests a security token for the specified subject. The method decodes a received token and returns the `Token` object.

### 3.5.2. Decoding and Validating a Security Token

The second use case happens when a SPECS service receives a security token attached to a HTTP request (presumably in a HTTP header). The SPECS service has to decode and validate the token, retrieve some information from the claims, and make an authorization decision. For this use case, the *security-tokens-client* library provides the class `SecurityTokensValidator` with the following API:

```
public SecurityTokensValidator(String stsAddress,
     String trustStoreFile, String trustStorePass)
public SecurityTokensValidator(
   VerificationCertProvider verifCertProvider,
   RevocationVerifier revocationVerifier)
public Token validate(String encodedToken)
```

The `SecurityTokensValidator` provides two constructors. The first one creates a `SecurityTokensValidator` instance, which uses default `VerificationCertProvider` and `RevocationVerifier`, that is `VerificationCertProviderWS` and `TRLCache`. It accepts three parameters:

- `stsAddress`: address of the STS.
- `trustStoreFile`, `trustStorePass`: trust store file path and password. The trust store contains the CA certificate chain (the issuer of the STS server certificate) or the STS server certificate. The trust store is needed to validate the STS certificate when establishing the secure connection with the server.

The second constructor creates a `SecurityTokensValidator` instance using the provided `VerificationCertProvider` and `RevocationVerifier`. It accepts two parameters:

- `verifCertProvider`: an instance of `VerificationCertProvider` which is needed to obtain STS's signing certificate for verifying tokens signature.
- `revocationVerifier`: an instance of `RevocationVerifier` which is needed to check the token revocation status.

The method `validate` decodes and validates the given encoded token and returns the `Token` object. From the `Token` object the service can get the token's payload.

#### *3.5.2.1. VerificationCertProvider Implementations*

The *security-tokens-client* module provides two implementations of the `VerificationCertProvider` interface:

- `VerificationCertProviderP12`: retrieves the requested signing certificate from a local trust store. The trust store is maintained manually by the administrator, who obtains the signing certificate(s) and imports it into the trust store. No connection with the STS is required.
- `VerificationCertProviderWS`: retrieves the requested signing certificate from the STS by calling its RESTful API, and caches it locally. Afterwards, the certificates are retrieved from the local cache.

#### *3.5.2.2. RevocationVerifier Implementations*

The *security-tokens-client* module provides two implementations of `RevocationVerifier` interface:

- `TRLCache`: maintains a local TLR (Token Revocation List) cache. During the initialization, it downloads the full TRL, then it periodically pulls delta TRLs with the recent changes (tokens revoked since the last update). Moreover, it periodically cleans up the TRL cache (i.e., it removes all expired tokens from the list). The tokens revocation status is checked against the local TRL copy. This approach is fast - the validation can be accomplished locally. However, the disadvantages are time lag between token revocation and client local cache, and complexity of the token revocation list synchronization.
- `OnlineRevocationVerifier`: checks token revocation status by calling the STS RESTful API for each token. This approach is most accurate - there is no time lag between token revocation and client local cache. The disadvantage is that it requires a call to the STS for each validation that takes some time, causes a lot of network traffic and load on the STS.

### 3.5.3. Security Tokens CLI Shell

The *security-tokens-client* provides a command-line application, which can be used for manually obtaining security tokens, decoding and validating them, checking if a specific token is revoked, printing the TRL. The application can be started from the command line (as shown below) with the following parameters: `sts-address`, `truststore-file`, `truststore-pass`, `keystore-file`, and `keystore-pass`. The last two parameters are optional, and are needed for obtaining security tokens.

```
java -cp libs/* org.specs.pkitokens.client.ConsoleClient \
   --sts-address=https://localhost:8443/security-tokens-service \
   --truststore-file=sts-truststore.jks --truststore-pass=password \
   --keystore-file=security-tokens-client-cli.jks --keystore-pass=password
```

When started, the application provides the following commands:

```
obtain <username> <password> <sla-id>
validate <token>
isRevoked <token-id>
printTRL
exit
```

### 3.5.4. Security Tokens Servlet Filter

The *security-tokens-client* library provides a Java servlet filter called `SecurityTokensFilter`, which can be plugged into any Java Web application or Java RESTful web service. The filter intercepts every HTTP request before it is processed, checks if a security token is present, decodes and validates the token and puts the `Token` object into `HttpServletRequest` object from where the `Token` is available to servlets processing the request, which are able to get information stored in the token.

The filter can also be used for authorization, using an XACML authorization engine. The filter extracts a set of claims from the token, creates a XAML access control decision request, sends it to the Policy Decision Point (PDP) and enforces the access decision received from the PDP - allows or denies the request.

The filter can be deployed in a web application deployment descriptor file (`web.xml`). First, the filter is declared as shown below:

```
<filter>
   <filter-name> SecurityTokensFilter </filter-name>
   <filter-class>org.specs.pkitokens.client.SecurityTokensFilter</filter-class>
   <init-param>
      <param-name>configFile</param-name>
      <param-value>security-tokens-filter-config.xml</param-value>
   </init-param>
</filter>
```

Then the filter is mapped to a specific servlet or URL pattern, as follows:

```
<filter-mapping>
   <filter-name>SecurityTokensFilter</filter-name>
   <servlet-name>Jersey REST Services</servlet-name>
   <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

### *3.6. Testing*

The following tables present several unit tests executed for the Security Tokens mechanism.

| Test ID | security-tokens-core:CertUtilsTest |
|---|---|
| Test objective | Test CertUtils utility methods for reading PrivateKey object from a PEM format, reading X509Certificate object from a PEM format, converting X509Certificate object to a PEM format. |
| Verified requirements | / |
| Inputs | PEM file containing test private key and certificate. |
| Expected results | All operations execute successfully. |
| Outputs | None. |
| Comments | All operations executed successfully. |

| Test ID | security-tokens-core:CompressUtilsTest |
|---|---|
| Test objective | Test CompressUtils utility methods for compressing and decompressing data. |
| Verified requirements | *ENF_TOK_R4, ENF_TOK_R8* |
| Inputs | String object with test data. |
| Expected results | Test data is first compressed and then decompressed. The decompressed data must be identical to the source data. |
| Outputs | None. |
| Comments | Everything as expected. |

| Test ID | security-tokens-core:JacksonSerializer |
|---|---|
| Test objective | Test JacksonSerializer functionality for serialization/deserialization to/from JSON |
| Verified requirements | *ENF_TOK_R3, ENF_TOK_R8* |
| Inputs | A test Java object |
| Expected results | Test object is serialized to JSON. The object deserialized from that JSON must be equal to the source object. |
| Outputs | None. |

| | |
|---|---|
| Comments | Everything as expected. |

| Test ID | security-tokens-core:TokenSignerTest |
|---|---|
| Test objective | Create TokenSigner object, load signing key and certificate from a Java key store. |
| Verified requirements | *ENF_TOK_R7* |
| Inputs | Key store with test signing key and certificate. |
| Expected results | All operations execute successfully. |
| Outputs | None. |
| Comments | All operations executed successfully. |

| Test ID | security-tokens-core:TokenTest |
|---|---|
| Test objective | Create a security token (Token object) with some test claims, sign it with TokenSigner using test signing key and get the encoded token. Decode and validate the encoded token and compare decoded Token object with source Token object. |
| Verified requirements | *ENF_TOK_R7, ENF_TOK_R8* |
| Inputs | Key store with test signing key and certificate. |
| Expected results | The token's signature is valid, decoded Token object is identical to the source Token object. |
| Outputs | None. |
| Comments | All operations executed successfully. |

| Test ID | security-tokens-service:TokensResourceTest |
|---|---|
| Test objective | Test REST API for issuing tokens and retrieving list of issued tokens. |
| Verified requirements | *ENF_TOK_R3* |
| Inputs | Key store with test signing key and certificate. |
| Expected results | Issued token is valid and contains correct claims. |
| Outputs | None. |
| Comments | All operations executed successfully. |

| Test ID | security-tokens-service:RevocationListResourceTest |
|---|---|
| Test objective | Test REST API for revoking tokens and generating token revocation list. |
| Verified requirements | *ENF_TOK_R5, ENF_TOK_R6* |
| Inputs | Key store with test signing key and certificate. |
| Expected results | The token is successfully revoked, the generated token revocation list contains the revoked token. |
| Outputs | None. |
| Comments | All operations executed successfully. |

| Test ID | security-tokens-service:CertificatesResourceTest |
|---|---|
| Test objective | Test REST API for retrieving signing certificate. |
| Verified requirements | *ENF_TOK_R1, ENF_TOK_R8* |
| Inputs | Key store with test signing key and certificate. |
| Expected results | The certificate returned by the service is valid and matches the test |

| | |
|---|---|
| | signing certificate. |
| Outputs | None. |
| Comments | All operations executed successfully. |

| Test ID | security-tokens-client:VerificationCertProviderP12Test |
|---|---|
| Test objective | Create a VerificationCertProviderP12 object and test retrieving signing certificate. |
| Verified requirements | *ENF_TOK_R4* |
| Inputs | Key store with test signing key and certificate. |
| Expected results | The certificate returned by the VerificationCertProviderP12 matches the certificate in the test key store. |
| Outputs | None. |
| Comments | All operations executed successfully. |

| Test ID | security-tokens-client:VerificationCertProviderWSTest |
|---|---|
| Test objective | Create a VerificationCertProviderWS object and test retrieving signing certificate. |
| Verified requirements | *ENF_TOK_R4* |
| Inputs | None. |
| Expected results | The certificate returned by the VerificationCertProviderWS matches the test-signing certificate. |
| Outputs | None. |
| Comments | Requires mock Security Tokens Service. |

| Test ID | security-tokens-client:TRLCacheTest |
|---|---|
| Test objective | Create a TRLCache object and test maintaining local token revocation list cache functionality. Request a new token, revoke it and test if the token appears in the token revocation list cache. |
| Verified requirements | *ENF_TOK_R4* |
| Inputs | None. |
| Expected results | The revoked token appears in the token revocation list cache. |
| Outputs | None. |
| Comments | Requires mock Security Tokens Service. |

| Test ID | security-tokens-client:TokenRetrieverTest |
|---|---|
| Test objective | Create a TokenRetriever object and obtain security token for test subject. |
| Verified requirements | *ENF_TOK_R3, ENF_TOK_R4, ENF_TOK_R7* |
| Inputs | Trust store with Security Tokens Service certificate, key store with the client certificate and private key. |
| Expected results | The obtained token is valid and contains correct set of claims for specified subject. |
| Outputs | None. |
| Comments | Requires mock Security Tokens Service. |

| Test ID | security-tokens-client:TokenValidatorTest |
|---|---|
| Test objective | Create a TokenValidator object and validate an encoded test token. |
| Verified | *ENF_TOK_R4, ENF_TOK_R8* |

| | |
|---|---|
| requirements | |
| Inputs | Trust store with Security Tokens Service certificate, test token in an encoded form. |
| Expected results | The encoded token is successfully decoded and validated. |
| Outputs | None. |
| Comments | Requires mock Security Tokens Service. |

# 4. The Credential Service mechanism

The Credential Service mechanism is devoted to storing and managing credentials belonging to the SPECS Owner and needed to access the resources offered by external CSPs. The mechanism, moreover, enables to share securely these credentials with the set of SPECS Platform's core components that need them to automatically access those resources (e.g., the Broker, which needs credentials to acquire resources from Amazon).

The Credential Service mechanism is offered by the Credential Service component of the SPECS Vertical Layer, which includes the following components:

- the **Credential Manager**: stores and manages the credentials provided by the SPECS Owner;
- the **Credential Management Application**: enables the SPECS Owner to manage credentials and to assign them to components;
- the **Credential Client**: enables the SPECS core components that need credentials to acquire and use them.

In order to use the Credential Service mechanism, a SPECS component must integrate the Credential Client and implement a predefined interface. The Credential Management Application component offers the *Credentials API*, discussed in Annex B, which provides the functionalities needed to manage credentials and associate existing credentials with components.
The source code of the Credential Service mechanism can be found on the project's Bitbucket repository at [8][9].
In the next subsections, the behaviour of the mechanism is dealt with. Moreover, we report the status of development activities, present the organization of repositories of source code, provide with guidelines for installation and usage, and present tests executed for verifying the quality of the code.

## *4.1. Behaviour*

In this section, we briefly describe the behaviour of the Credential Service mechanism and present the updated architecture, motivating the changes that have been made as compared to the previous version. The integration of the mechanism into the SPECS framework will be presented in the documents of the integration task T1.5, at the end of the project.

The current version of the Credential Service mechanism relies upon the **Vault project** [10], a tool for securely accessing *secrets.* According to the Vault definition, a secret is defined as "anything that you want to tightly control access to, such as API keys, passwords, certificates, and more". Vault provides features such as secure secret storage, dynamic secret generation, and secret leasing and revocation, in addition to enabling the encryption of data before they are stored elsewhere.

From the architecture point of view, Vault is based on a Server (the *Vault Server*) that offers an API (the *Vault API*), which clients interact with. Vault's internal components (responsible for auditing, authentication, policies management etc.) are protected by a *Barrier*, which ensures that only encrypted data is sent outward, and that data is verified and decrypted on the way in. Inside the Barrier, secrets are managed by a *Secret Backend*, while a *Credential Backend* is used to authenticate users that are connecting to Vault: after authentication, a

*client token* is returned, to be used for future requests. The token identifies users; ACL policies are associated with a token when the token is generated and are used for authorization.

Located outside the Barrier, an untrusted *Storage Backend* is used to store encrypted secrets (so that they are available across restart). When the Vault Server is started, the data in the Storage Backend is first decrypted, and then all the configured audit, credential and secret backend are loaded (Vault *unsealing* process).

Given the features of the Vault solution, we decided to reuse its tools for the implementation of the Credential Service mechanism. As previously mentioned, the SPECS platform needs to store secrets (i.e., credentials) and to share them with some of the core components, in order, for example, to enable the access to External CSPs for the acquisition of new cloud services/resources. Hence, the SPECS Owner, who is the holder of the credentials needed to access external resources, should be able to communicate such credentials to the components that actually perform the access, in order to automate the acquisition and use of the resources.

In order to enable this behaviour, we integrated the Vault solution into the SPECS platform. In particular, we integrated the Vault Server and the Vault Storage Backend in the Credential Manager component. The configuration and deployment of the Vault Server and of the Vault Storage Backend within the Credential Manager component is automated by a suitable Chef recipe, whose execution is triggered by the SPECS Owner at the start-up of the SPECS platform. In this phase, the SPECS Owner is also responsible for the initialization and unsealing of the Vault Server. In fact, when a Vault Server is launched, it starts in a *sealed* state, in that it is configured to know where and how to access the physical storage, but does not know how to decrypt its content. Unsealing is the process of constructing the *master key* necessary to read the decryption key to decrypt the data, thus allowing access to the Vault.
The master key is obtained by the SPECS Owner during the initialization phase, together with a *root token,* needed to generate the authentication tokens used to communicate with the Vault Server. Note that this confidential information is displayed to the SPECS Owner only once, through a web interface, and it is neither stored by SPECS or transmitted anywhere else. It is the SPECS Owner's responsibility to store it securely.

After the Vault initialization and unsealing, the Credential Service can be started. This is accomplished through the installation and configuration of the *Credential Management Application*, which offers a Web interface to the SPECS Owner for the management of credentials and the communication with Vault.

The Credential Service architecture is presented in Figure 2, which shows the main components.

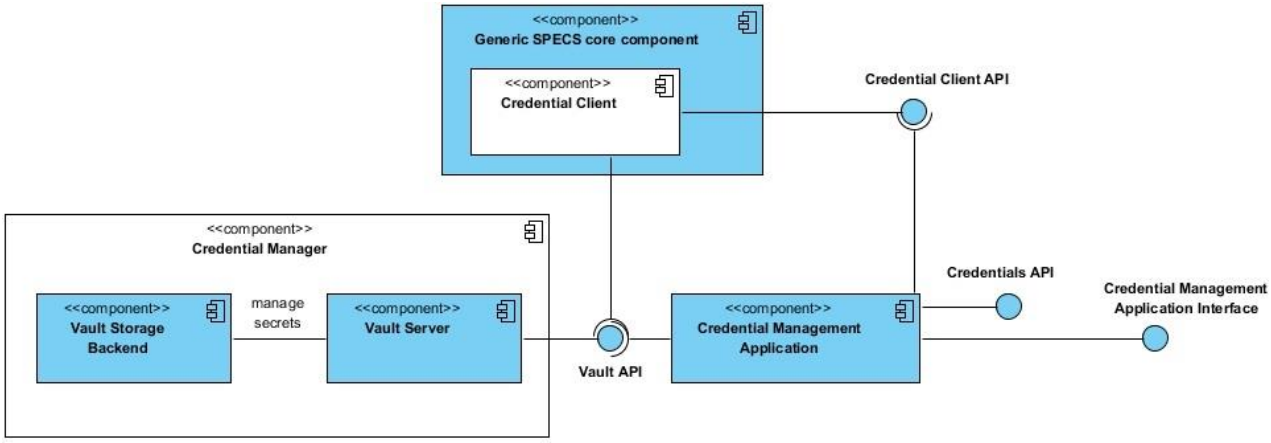


**Figure 2: Credential Service Architecture**

The figure shows (i) the Credential Management Application, exposing the Credentials API and used by the SPECS Owner to manage credentials, (ii) the Credential Manager component, including the Vault Server and the Vault Storage Backend, and (iii) the generic SPECS core component, which needs credentials to carry out its tasks (e.g., the Secure Provisioning component). Such component, has shown, includes a Credential Client component, which must be installed on it in order to enable the access to the Vault API.

The Credential Client component exposes an API (the Credential Client API) used by the Credential Management Application to notify the component about the assignment of credentials, and to provide it with the authentication token for the communication with the Credential Manager. In fact, as previously mentioned, in order to communicate with the Credential Manager (i.e., the Vault Server) a token is needed; this is generated by the SPECS Owner via the Credential Management Application when configuring the component for credentials usage. Note that even the Credential Management Application needs an authentication token to invoke the Vault API; this is generated at the set-up of the Credential Service mechanism, after the initialization of Vault.

By means of the Credential Management Application, the SPECS Owner can:

- Add new credentials;
- Configure a component for credentials usage;
- Assign existing credentials to a component;
- Initialize Credential Service.

These represent the main use cases related to the Credential Service mechanism (see Figure 3) and are the basic operations that can be done by the SPECS Owner, who administers the platform and is the only one devoted to assigning and distributing credentials to components.

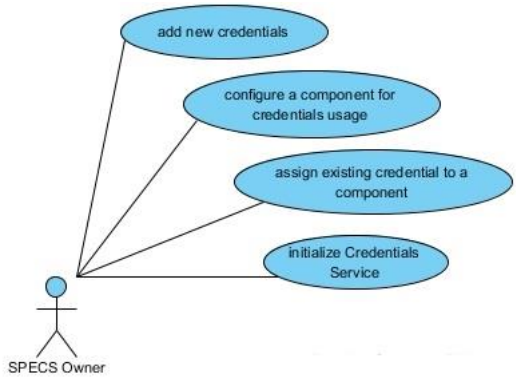


**Figure 3: Credential Service mechanism use cases**

The above mentioned functionalities rely upon the Credentials API, which enables to:

- Initialize and unseal the Vault Server;
- Add new credentials to the Credential Manager at the SPECS Owner's path;
- Generate tokens for components, so that they can authenticate and access the Credential Manager (i.e., the Vault Server);
- Assign credentials to components by making them available at the component's path in Vault.

In order to store new credentials, the SPECS Owner interacts with the Credential Management Application, which in turn invokes the Vault API exposed by the Credential Manager to add new credentials. Then the credentials are stored at the SPECS Owner's path.

The SPECS core components that need the Credential Service mechanism must be configured properly. At the start-up of the platform, the SPECS Owner triggers the execution of a recipe stored in the Chef Server that installs the component along with the Credential Client. Later, when credentials must be assigned to the component, to complete its set-up, the SPECS Owner generates an authentication token for the component through the Credential Management Application interface. Then such token is provided to the component via the Credential Client API exposed by the running Credential Client, and will be used for subsequent communications with the Credential Manager.

When the SPECS Owner has to assign credentials to a component that has been previously configured in the above-discussed way, the following process is carried out. First, the credentials to assign are copied to the component's path. This is handled by the Credential Management Application, which accesses the Credential Manager to retrieve the credentials and to copy them to the component's path. Then, the Credential Management Application notifies the assignment of such credentials to the Credential Client of the interested core component via the exposed Credential Client API. Once retrieved, the credentials are stored in the internal memory of the component and used for resources acquisition.

It is worth pointing out that components receive the credentials through the Credentials API and there is no means to directly share such credentials with other components (there is no API enabled to output the obtained credentials).

## *4.2. Status of development activities*

In Table 4, we synthetically report the requirements covered by the Credential Service components.

| Credential Service Mechanism | SPECS Components | | |
|---|---|---|---|
| Requirements | Credential Client | Credential Manager | Credential Management Application |
| *ENF_CRED_R1* | - | - | - |
| *ENF_CRED_R2* | X | X | X |
| *ENF_CRED_R3* | X | X | X |
| *ENF_CRED_R4* | X | X | X |
| *ENF_CRED_R5* | X | X | X |

**Table 4. SPECS Components related to the Credential Service mechanism and related requirements**

There are 5 requirements associated to Credential Service mechanism, one of which has been deprecated since it is already covered by the Secure Provisioning component. Remaining requirements are all covered by the final prototype. The following table presents a validation of the mechanism's design in terms of explaining how each requirement associated to the mechanism has been covered.

| **REQ_ID** | **Requirement** | Description |
|---|---|---|
| *ENF_CRED_R1* | ***Target service authentication schemes support*** | *Deprecated* – The authentication procedures required by supported CSPs are carried out by the Secure provisioning component. |
| *ENF_CRED_R2* | ***Access control policies to the credentials usage*** | In the developed solution, credentials are assigned to components by the SPECS Owner and copied to a specific Vault path that can be accessed only with an authentication token. |
| *ENF_CRED_R3* | ***Multiple credentials for the same target service*** | The Credential Service mechanism allows to assign credentials to a component associated with a certain account. Hence, a component devoted to the acquisition of a target service can be invoked with different credentials belonging to different accounts. |
| *ENF_CRED_R4* | ***Credentials usage auditing*** | The SPECS Credential Service mechanism envisions the invocation of the Auditing component whenever credentials are assigned to a component. Moreover, when a component (like the Secure Provisioning component) requests the access to a target service with assigned credentials, this will be logged by the Auditing component as well. |
| *ENF_CRED_R5* | ***Disjoint credentials data management and storage*** | In the SPECS Credential Service mechanisms, credentials are stored encrypted in the Vault Credential Backend. When assigned to a component, credentials are copied to its internal memory thus reducing the risk of compromise. |

**Table 5. Requirements for the Security Tokens mechanism**

In Table 6, we report the current status of development activities of all SPECS components associated to the Credential Service mechanism. Note that the *Credential Client* component and the *Credential Management Application* substitute the original *Credential Service* component (thought as independent component), now deprecated. Moreover, the *Credential Store* component designed in D4.4.1 is currently integrated in the Credential Manager (it is implemented by the Vault Secret Backend). All these components are currently available.

| Mechanism | Artifacts under development | Status |
|---|---|---|
| Credential Service | component: credential-manager | Completed |
| | component: credential-client | Completed |
| | component: credential-application | Completed |

**Table 6. Credential Service implementation status**

## *4.3. Repository*

The Credential Service mechanism implementation is made up of three modules, representing respectively the Credential Manager, the Credential Management Application and the Credential Client. The final prototype of the mechanism is available on the project's Bitbucket repository [8]. Its organization is presented in the following subsection.

## *4.4. Installation*

In this section, we describe how to install and configure the Credential Service mechanism. The installation requires the following steps:

- Installation and configuration of the Credential Manager;
- Installation of the Credential Management Application;
- Configuration of the generic component that needs credentials with a Credential Client for communication with the Credential Manager.

The details of each step are reported in the next subsections.

### 4.4.1. Installing the Credential Manager

Installing the Credential Manager means instantiating and configuring both the Vault Server and the Vault Storage Backend. This procedure is accomplished by using a Chef Recipe that allows to automate the whole flow of operations, and to store in a Chef Databag all the data needed by the other components to call the Vault API (e.g. the IP address of the machine hosting the Credential Manager). Once the execution of the recipe has completed its operations, the component gets running and offers its functionalities to the other components.

Prerequisites:

- Java 7

### 4.4.2. Installing the Credential Management Application

The installation of the Credential Management Application, which provides the Credentials API and is able to contact any component that has been configured to use credentials stored into the Credential Manager, is accomplished by running a Chef Recipe that allows to automate the whole flow of operations. Once the execution of the recipe has completed, the component gets properly running.

Prerequisites:

- Java 7

### 4.4.3. Installing a Credential Client on a SPECS component

As said in Section 4.1, in order to enable a SPECS component to use credentials, it is necessary to install and configure a Credential Client on it. More precisely, the Credential Client component must be installed as a dependency of the existing Java project related to the SPECS component.
Actually, this is not the only action needed. Indeed, credentials are represented by different sets of key-value pairs depending on the specific provider, and components must be allowed to manage correctly the supported sets of credentials (i.e., the supported providers). To this aim, in the current Credential Service mechanism implementation, a *DataModel* project has been created. This project must be edited to add support for a new set of credentials to store in and retrieve from the Credential Manager, and must be included as a dependency by the core component project, along with the Credential Client project.
The steps needed to install both these projects are illustrated in the following paragraph.

Prerequisites:

- Git client;
- Maven;
- Java 7;
- Java web container;
- Java code of the existing core component.

In order to import the DataModel project, it is necessary to:

- clone the DataModel git repository [11];
- convert it into a Maven project;
- execute the 'maven install' command in order to execute tests and to generate the artifact;

When using the Eclipse IDE, these steps are detailed as follows:

- Import the project from git as a "general project";
- right click on the project, click on "configure", then click on "Convert to Maven Project";
- right click on the project, click on "Run as", then click on "Maven install".

Once the project has been properly configured in the IDE, it has to be included as dependency by the core component project. This can be achieved by adding the following lines into the `<dependencies>` tag of the core component *pom* file:

```
<dependency>
      <groupId>eu.specs-project.utility</groupId>
      <artifactId>data-model</artifactId>
      <version>0.1-SNAPSHOT</version>
</dependency>
```

To download and generate the artifact of the *Credential Client*, here are the general steps:

- clone the git repository [8];
- convert it into a Maven project;
- execute the 'maven install' command in order to execute tests and to generate the artifact;

When using the Eclipse IDE, these steps are detailed as previously described for the DataModel project.

Once the project has been properly configured in the IDE, even this project must be included as a dependency of the core component project. This can be achieved by adding the following lines into the `<dependencies>` tag of the core component *pom* file:

```
<dependency>
        <groupId>specs-utility-credential-client</groupId>
        <artifactId>specs-utility-credential-client</artifactId>
        <version>0.0.1-SNAPSHOT</version>
</dependency>
```

### *4.5. Usage*

In the previous section, we showed how to install the *Credential Client* and the *DataModel* projects as dependencies of the generic core component project. In this section, we provide some details on the implementation of the functionalities needed to enable the component to (i) receive the token for the communication with the Vault Server and to (ii) be notified about credentials assignment.

The steps that are mandatory to make the component able to access credentials stored into the Vault Server are the following:

- Inside the DataModel project, it is necessary to:
  - add a new class inside the package *eu.specs.credentials.<component_name>*, which contains a property for each piece of credential needed by the component itself, and the related *get* and *set* methods. The class has to be a Java Bean. This procedure has to be repeated for each set of credentials needed by the component.

- Inside the component project, it is necessary to:
  - add a servlet mapped at the URL:
    <ip_address_of_core_component>/**credentialClientToken**,
    able to handle HTTP POST requests. The servlet must invoke the static method *setToken* provided by the class *ComponentServletManagment*, whose signature is defined as:

```
add public static HttpServletResponse setToken(HttpServletRequest req,
HttpServletResponse resp);
```

  - add a servlet mapped at the URL:
    <ip_address_of_core_component>/**credentialClientCredentialIdentifier**,
    able to handle HTTP POST requests. The servlet must invoke the static method *setCredentials()* provided by the class ComponentServletManagment, whose signature is defined as:

```
public static HttpServletResponse setCredentials(HttpServletRequest req,
HttpServletResponse resp)
```

  - implement the interface defined inside the CredentialClient project that defines the method “storeCredentials”, whose signature is defined as:

```
public void setCredential(HashMap<String, String> hashmap);
```

This method receives as input parameter a HashMap containing all the values (stored in Vault Server) associated to the keys useful to make the generic component full working. Here the developer has to write the code useful to store the credentials, so that its component can use them when needed.

- create an instance of the implemented interface, get an instance of the class ComponentManagerSingleton and call its `setInterface` method, that receives the implemented interface as parameter.
  It is necessary to remark the fact that this interface and the singleton enable the component to get the credentials stored into the Vault Server; in fact, the credentials are retrieved from the VaultServer by the CredentialClient and passed to the component using the implementation of the provided interface.

### *4.6. Testing*

The following tables present several unit tests executed for the Credential Service mechanism.

| Test ID | testSecretEntity |
|---|---|
| Test objective | The goal is to verify that the entity Secret is correctly created and that the setting and getting methods works fine. |
| Verified requirements | *ENF_CRED_R2* |
| Inputs | Id and String of Secret |
| Expected results | Secret object with id and secret setted |
| Outputs | none |
| Comments | All operations executed successfully |

| Test ID | testSetComponentManager |
|---|---|
| Test objective | The goal is to verify that the object ComponentManager is correctly implemented and the constructor works fine. |
| Verified requirements | *ENF_CRED_R2* |
| Inputs | |
| Expected results | A not null ComponentManager |
| Outputs | none |
| Comments | All operations executed successfully |

| Test ID | testComponentActionsReceiveToken |
|---|---|
| Test objective | The goal is to verify that the method receiveToken of the class ComponentActions is called correctly and works fine.<br>To test this method an HttpServletRequest is created and the key and token parameters are setted on it. |
| Verified requirements | *ENF_CRED_R2* |
| Inputs | Key and Token parameters |
| Expected results | Returns response code 200 OK |
| Outputs | none |

| Comments | All operations executed successfully |
|---|---|

| Test ID | testComponentActionsReceiveEucalyptusCredential |
|---|---|
| Test objective | The goal is to verify that the method receiveCredentials of the class ComponentActions is called correctly and works fine.<br>To test this method an HttpServletRequest is created and the Type parameter equals to "eucalyptus" is setted.<br>Morover a Virtual Interface is created at the path ("/v1/secret/owner/component_name/secret_name") to emulate the Vault server. |
| Verified requirements | *ENF_CRED_R2* |
| Inputs | Type parameter |
| Expected results | Returns response code 200 OK<br>Number of read credentials equals to five |
| Outputs | none |
| Comments | All operations executed successfully |

| Test ID | testComponentActionsReceiveAmazonCredential |
|---|---|
| Test objective | The goal is to verify that the method receiveCredentials of the class ComponentActions is called correctly and works fine.<br>To test this method an HttpServletRequest is created and the Type parameter equals to "amazon" is setted.<br>Moreover a Virtual Interface is created at the path "/v1/secret/owner/component_name/secret_name" to emulate the Vault server. |
| Verified requirements | *ENF_CRED_R2* |
| Inputs | Type parameter |
| Expected results | Returns response code 200 OK<br>Number of read credentials equals to three |
| Outputs | none |
| Comments | All operations executed successfully |

| Test ID | testInit |
|---|---|
| Test objective | The goal is to verify that the Vault server is initialized correctly. |
| Verified requirements | *ENF_CRED_R5* |
| Inputs | None |
| Expected results | It returns both the key and the root token |
| Outputs | |
| Comments | All operations executed succesfully |

| Test ID | testUnseal |
|---|---|
| Test objective | The goal is to verify that the Vault server is correctly unsealed. |
| Verified requirements | *ENF_CRED_R5* |
| Inputs | Unseal key parameter |
| Expected results | Returns the response code 200 ok |

| | |
|---|---|
| Outputs | none |
| Comments | All operations executed successfully |

| Test ID | testConfigureOwnerComponent |
|---|---|
| Test objective | The goal is to verify that the action correctly updates a new owner policy, creates a new owner token and sends it to the application |
| Verified requirements | *ENF_CRED_R2* |
| Inputs | Owner Component parameter, Owner path parameter, Root token parameter |
| Expected results | Returns the response code 200 ok |
| Outputs | none |
| Comments | All operations executed successfully |

| Test ID | testConfigureGenericComponent |
|---|---|
| Test objective | The goal is to verify that the action correctly updates a new generic component policy, creates a new generic component token and sends it to the application |
| Verified requirements | *ENF_CRED_R2* |
| Inputs | Generic Component parameter, Generic Component path parameter, Root token parameter |
| Expected results | Returns the response code 200 ok |
| Outputs | none |
| Comments | All operations executed succesfully |

| Test ID | testWriteCredentials |
|---|---|
| Test objective | The goal is to verify that the action correctly store credentials in the vault server at the owner path |
| Verified requirements | *ENF_CRED_R5* |
| Inputs | Provider parameter, Username Parameter, keys parameter |
| Expected results | Returns the response code 200 ok |
| Outputs | |
| Comments | All operations executed succesfully |

| Test ID | testAddToken |
|---|---|
| Test objective | The goal is to verify that the method receiveToken of the class ComponentActions is called correctly and works fine. |
| Verified requirements | *ENF_CRED_R2* |
| Inputs | Key and Token parameters |
| Expected results | Returns the response code 200 ok |
| Outputs | |
| Comments | All operations executed succesfully |

| Test ID | testAssignSecret |
|---|---|
| Test objective | The goal is to verify that the Assign action correctly copy the credentials from the owner path to relative component path |

| | |
|---|---|
| Verified requirements | *ENF_CRED_R2* |
| Inputs | Providers parameter, Username parameter, Component parameter |
| Expected results | Returns the response code 200 ok |
| Outputs | |
| Comments | All operations executed succesfully |

| | |
|---|---|
| Test ID | testNotify |
| Test objective | The goal is to verify that the action correctly notify the component credential client |
| Verified requirements | *ENF_CRED_R2* |
| Inputs | Component parameter, Username parameter, Provider parameter |
| Expected results | Returns the response code 200 ok |
| Outputs | |
| Comments | All operations executed succesfully |

# 5. Conclusions

This document presents the final implementation of the secure interactions mechanisms, namely Security Tokens and Credential Service. As already reported in D4.4.1, in SPECS the secure interaction mechanisms specifically tackle two different aspects encountered when dealing with cloud application security, namely service-to-service authentication and credential data management. It is worth to highlight that the authorization aspects where taken in consideration by both components and, in fact, the security token mechanism implemented an identity-based authorization mechanism, implicitly associated to the issue of security tokens while the credential manager allows the configuration of security policies, based on the definition of Access Control List, that can be easily set up at installation time.

While the Security Tokens mechanism was already fully developed and implemented at month 12, the implementation of the Credential Service mechanism has been substantially revised to include powerful existing tools for credential management. The Credential Service is used by the Broker component only once and off-line, during the set-up, when a new instance of resources need to be provisioned; for this reason, this component is not critical from a performance point of view and there was no need to benchmark it, as done for almost all SPECS components.

In conclusion, this document reports on the current version and presents all implementation and usage details, in addition to an updated validation scenario, which refines the one presented in Deliverable 5.1.2 and that will be used for integration by Task 1.5. The two components are available on the SPECS bitbucket repository.

# 6. Bibliography

[1] M. Jones, J. Bradley, N. Sakimura, "*JSON Web Token (JWT) draft-ietf-oauth-json-web-token-25*", 2014. [Online]. Available: https://tools.ietf.org/html/draft-ietf-oauth-json-web-token-25.

[2] M. Jones, J. Hildebrand, "*JSON Web Encryption (JWE) draft-ietf-jose-json-web-encryption-31*", 2014. [Online]. Available: https://tools.ietf.org/html/draft-ietf-jose-json-web-encryption-31.

[3] M. Jones, J. Bradley, N. Sakimura, "*JSON Web Signature (JWS) draft-ietf-jose-json-web-signature-31*", 2014. [Online]. Available: https://tools.ietf.org/html/draft-ietf-jose-json-web-signature-31.

[4] SPECS, "*SPECS Utility Security Tokens Wiki*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-utility-security-tokens/wiki/Home.

[5] SPECS, "*SPECS Utility Security Tokens*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-utility-security-tokens

[6] "*Apache Tomcat*", 2014. [Online]. Available: http://tomcat.apache.org/.

[7] MySQL, "*Download MySQL Community Server*", Oracle Corporation, 2014. [Online]. Available: http://dev.mysql.com/downloads/mysql/.

[8] SPECS, "*SPECS Credentials Service*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-utility-credential-client

[9] SPECS, "*SPECS Credentials Service*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-utility-credential-management-application

[10] *"Vault Project"*, HashiCorp. [Online]. Available: https://vaultproject.io

[11] SPECS, "*SPECS Utility Data Model*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-utility-data-model