Spring Security SAML Extension

Reference Documentation

Vladimír Schäfer

Spring Security SAML Extension: Reference Documentation

by Vladimír Schäfer

3.0-RC2

Copyright © 2009-2013 Vladimír Schäfer [mailto:vladimir.schafer@gmail.com]

Table of Contents

Part I. Getting Started

This chapter provides essential information needed to enable your application to act as a service provider and interact with identity providers using SAML 2.0 protocol. Later in this guide you can find information about detailed configuration options and additional use-cases enabled by this component.

1. Introduction

1.1 What this manual covers

This manual describes Spring Security SAML Extension component, it's uses, installation, configuration, design and tested environments.

1.2 When to use Spring Security SAML Extension

Component enables both new and existing applications to act as a Service Provider in federations based on SAML 2.0 protocol and enable Web Single Sign-On. Spring Security Extension allows seamless combination of SAML 2.0 and other authentication and federation mechanisms in a single application. All products supporting SAML 2.0 in Identity Provider mode (e.g. ADFS 2.0, Shibboleth, OpenAM/OpenSSO, RM5 IdM or Ping Federate) can be used to connect with Spring Security SAML Extension.

Extension be used in applications which are not primarily secured using Spring Security. It can be adapted for both single and multi-tenant environments.

Spring Security SAML Extension can be either embedded inside application and work along other authentication or single sign-on mechanisms or it can be deployed separately and convey authentication information to applications using a custom mechanism.

Spring Security Extension is probably the most complete open-source SAML 2.0 SP implementation with the widest feature-set and configuration possibilities. Other Java open-source alternatives are e.g. native SAML service providers integrating with IIS or Apache from Shibboleth (SAML processing is done on the web server and not on the application level) or OpenAM Fedlet.

1.3 Features and supported profiles

Current implementation should be conformant to SAML SP Lite and SAML eGovernment profile. The following profiles, bindings and features are supported as part of the product:

- Web single sign-on profile
- Web single sign-on holder-of-key profile
- IDP and SP initialized single sign-on
- Single logout profile
- Enhanced client/proxy profile
- Identity provider discovery profile and IDP selection
- Metadata interoperability and PKIX trust management
- Automatic service provider metadata generation
- Metadata loading from files, URLs, file-backed URLs
- Processing and automatic reloading of metadata with many identity providers
- Support for authentication contexts
- Logging for authentication events
- Customization of both SP and IDP metadata
- Processing of SAML attributes and user data using UserDetails interface

- Support for HTTP-POST, HTTP-Redirect, SOAP, PAOS and Artifact bindings
- Easy integration with applications using Spring Security
- Sample application with an user interface for quick configuration

Internal processing of SAML messages, marshalling and unmarshalling is handled by OpenSAML [https://wiki.shibboleth.net/confluence/display/OpenSAML/Home].

You can use the following supported standards as a reference:

SAML 2.0 basic profiles
- http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf
- http://docs.oasis-open.org/security/saml/v2.0/saml-metadata-2.0-os.pdf
- http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf
- http://docs.oasis-open.org/security/saml/v2.0/saml-authn-context-2.0-os.pdf
- http://docs.oasis-open.org/security/saml/v2.0/saml-bindings-2.0-os.pdf
- http://docs.oasis-open.org/security/saml/v2.0/saml-conformance-2.0-os.pdf

SAML 2.0 additional profiles
- http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-holder-of-key-browser-sso.pdf
- http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-idp-discovery.pdf
- http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml2-holder-of-key.pdf
- http://docs.oasis-open.org/security/saml/Post2.0/sstc-metadata-iop.pdf

eGovernment profile
- http://kantarainitiative.org/confluence/download/attachments/42139782/kantara-egov-saml2-profile-2.0.pdf

1.4 Requirements

Spring Security SAML Extension requires as minimum Java 1.6.

TODO Apache Tomcat, Jetty, Oracle Weblogic, ...

1.5 Source code

Source code for the project is maintained on Github [https://github.com/SpringSource/spring-security-saml].

1.6 License

Source code of the module is licensed under the Apache License, Version 2.0. You may obtain copy of the license at http://www.apache.org/licenses/LICENSE-2.0.

1.7 Support

Issue tracking for the module can be found at Spring Security Extensions Jira [https://jira.springsource.org/browse/SES/component/10711]. Feel free to submit bugs, patches and feature requests.

For community support please use Spring Security forum [http://forum.springsource.org/forumdisplay.php?86-SAML]. For additional support you can reach me at vladimir.schafer at gmail.com.

2. Glossary

Table 2.1. Definitions of terms used within this manual.

Term	Definition
Assertion	A part of SAML message (an XML document) which provides facts about subject of the assertion (typically about the authenticated user). Assertions can contain information about authentication, associated attributes or authorization decisions.
Artifact	Identifier which can be used to retrieve a complete SAML message from identity or service provider using a back-channel binding.
Binding	Mechanism used to deliver SAML message. Bindings are divided to front-channel bindings which use web-browser of the user for message delivery (e.g. HTTP-POST or HTTP-Redirect) and back-channel bindings where identity provider and service provider communicate directly (e.g. using SOAP calls in Artifact binding).
Discovery	Mechanism used to determine which identity provider should be used to authenticate user currently interacting with the service provider.
Metadata	Document describing one or multiple identity and service providers. Metadata typically includes entity identifier, public keys, endopoint URLs, supported bindings and profiles, and other capabilities or requirements. Exchange of metadata between identity and service providers is typically the first step for establishment of federation.
Profile	Standardized combination of protocols, assertions, bindings and processing instructions used to achieve a particular use-case such as single sign-on, single logout, discovery, artifact resolution.
Protocol	Definition of format (schema) for SAML messages used to achieve particular functionality such as requesting authentication from IDP, performing single logout or requesting attributes from IDP.
Identity provider (IDP)	Entity which knows how to authenticate users and provides information about their identity to service providers/relaying parties using federation protocols.
Service provider (SP)	Your application which communicates with the identity provider in order to obtain information about the user it interacts with. User information such as authentication state and user attributes is provided in form of security assertions.
Single Sign-On (SSO)	Process enabling access to multiple web sites without need to repeatedly present credentials necessary for authentication. Various federation protocols such as SAML, WS-Federation, OpenID or OAuth can be used to

Term	Definition
	achieve SSO use-cases. Information such as means of authentication, user attributes, authorization decisions or security tokens are typically provided to the service provider as part of single sign-on.
Single Logout (SLO)	Process terminating authenticated sessions at all resources which were accessed using single sign-on. Techniques such as redirecting user to each of the SSO participants or sending a logout SOAP messages are typically used.

3. Quick start guide

This chapter will guide you through steps required to easily integrate Spring Security SAML Extension with SSO Circle IDP service using SAML 2.0 protocol. When done you will have a working example of Web SSO against a single Identity Provider. The steps will guide you through deployment of the module, configuration of IDP metadata (XML document describing how to connect to the IDP server using SAML 2.0 protocol) and SP metadata (XML document describing your own service) and testing of web single sign-on and single logout.

3.1 Pre-requisites

Please make sure the following items are available before starting the installation:

- Supported application server or container
- Spring Security SAML Extension
- Java 1.6+ SDK
- Maven

SAML Extension relies on XML processing capabilities of JAXP. Some older versions of JRE might require updating of the embedded JAXP libraries. In case you encounter XML processing exceptions please create folder *jdk/jre/lib/endorsed* in your JDK installation and include files in *lib/endorsed* from the latest OpenSAML archive available at http://shibboleth.net/downloads/java-opensaml/. The location of the endorsed folder may differ based on your application server or container.

3.2 Installation steps

Compilation of the module

After downloading the Spring Security SAML Extension module and unziping compile the whole project using:

```
mvn package
```

Command will create file *spring-security-saml2-sample.war* in directory *saml2-sample/target*. We will be customizing content of the application in the following steps.

Configuration of IDP metadata

Modify file *WEB-INF/classes/security/securityContext.xml* inside the war archive and replace *metadata* bean as follows:

```
<bean id="metadata" class="org.springframework.security.saml.metadata.CachingMetadataManager">
    <constructor-arg>
        <list>
            <bean class="org.opensaml.saml2.metadata.provider.HTTPMetadataProvider">
                <constructor-arg>
                    <value type="java.lang.String">http://idp.ssocircle.com/idp-meta.xml</value>
                </constructor-arg>
                <constructor-arg>
                    <value type="int">5000</value>
                </constructor-arg>
                <property name="parserPool" ref="parserPool"/>
```

```
            </bean>
        </list>
    </constructor-arg>
</bean>
```

The settings tell system to download IDP metadata from the given URL with timeout of 5 seconds. In production system metadata should be either stored as a local file or be downloaded from a source using SSL/TLS with configured trust or which provides digitally signed metadata.

Generation of SP metadata

Modify file *WEB-INF/classes/security/securityContext.xml* inside the war archive and replace *metadataGeneratorFilter* bean as follows and make sure to replace the entityId value with a string which is unique within the SSO Circle service (e.g. urn:test:yourname:yourcity):

```
<bean id="metadataGeneratorFilter" class="org.springframework.security.saml.metadata.MetadataGeneratorFilter">
    <constructor-arg>
        <bean class="org.springframework.security.saml.metadata.MetadataGenerator">
           <property name="entityId" value="replaceWithUniqueIdentifier"/>
           <property name="signMetadata" value="false"/>
        </bean>
    </constructor-arg>
</bean>
```

Deployment

Deploy the updated war archive to your application server or container. After deployment the SP module will be available at e.g. *http://localhost:8080/spring-security-saml2-sample*

Uploading of SP metadata to the IDP

Download current SP metadata:

- Open web browser at the URL of the deployed application.
- Select *Metadata information*.
- Select first item from category *Service providers*, e.g. *http://localhost:8080/spring-security-saml2-sample/saml/metadata/alias/defaultAlias*
- Copy content of the Metadata textarea to your clipboard.

Upload SP metadata to the IDP:

- Register yourself at www.ssocircle.com and login to the service.
- Select Metadata manager and click Add new Service Provider.
- Enter *entityId* configured in the section called “Generation of SP metadata” to the FQDN field.
- Paste content of clipboard to the metadata information textarea.
- Store metadata by pressing the Submit button.
- Logout from the SSOCircle service.

3.3 Testing single sign-on and single logout

Open the front page of your SP application, select *http://idp.ssocircle.com* IDP and press login. System will generate new authentication request using SAML 2.0 protocol, digitally sign it and send it to the IDP. After

authentication at IDP with your account you will be redirected back to your application and automatically signed-in.

Pressing local logout will destroy local session and logout the user. Session is still active at the IDP and an attempt to reauthenticate will proceed without need to enter credentials.

Pressing global logout will destroy both local session and session at IDP.

Part II. Configuring SSO with SAML

This chapter provides information about configuration and customization options of the SAML extension. It will guide you throught typical scenarios including problems you migth encounter during integration with identity providers.

4. Configuration and integration

This chapter will discuss aspects of configuring and using the library in target applications.

4.1 Overview

Spring Security SAML 2.0 library comprises three modules:

- *saml2-core* contains implementation of the WebSSO profiles of the SAML 2.0 protocol and is required for integration to target systems.
- *saml2-sample* contains example of Spring configuration used for integration to target systems. It also contains user interface for generation and management of metadata.
- *saml2-doc* contains this documentation.

Configuration of library is done using Spring context XML. An example of configuration can be found under*saml2-sample/src/main/resources/security/securityContext.xml*. Setting up of the library typically involves these steps:

- integration to application using Spring XML configuration
- configuration of signature, encryption and trust keys
- configuration of security profiles
- import, generation and customization of SP and IDP metadata
- configuration of IDP selection
- configuration of single sign-on process
- configuration of logout process
- configuration of authentication object

Additional steps such as configuration of authentication logging, customization of SAML 2.0 bindings, configuration of artifact resolution or configuration of time skews might be needed.

4.2 Integration to applications

SAML module can be directly embedded into new of existing Spring applications. In this case application itself includes the SAML library in WEB-INF/lib directory of the war archive and processes all SAML interactions. The other option of using the SAML library is deploying it as a stand-alone module which transfers information about the authenticated user to the target application using a custom mechanism. This chapter only discusses the first option.

Maven dependency

In order to include the library and all it's dependencies add the following dependency to your pom.xml file:

```
<dependency>
    <groupId>org.springframework.security.extensions</groupId>
    <artifactId>spring-security-saml2-core</artifactId>
    <version>3.1</version>
</dependency>
```

The current version of SAML Extension has been tested to work with Spring 3.1.2, Spring Security 3.1.2 and OpenSAML 2.5.3. Later versions of these libraries are likely to be compatible without modifications.

Bean definitions

Configuration of the SAML library requires beans definitions included in the *saml2-sample/src/main/resources/security/securityContext.xml* configuration file. Include copy of the file in your own Spring application, either directly or with an inclusion. Configuration steps in the following chapters will be customizing beans included in the default context.

Beans of the SAML library are using auto-wiring and annotation-based configuration by default. Make sure that your Spring configuration contains e.g. the following settings in order to enable support for these features:

```
<context:annotation-config/>
<context:component-scan base-package="org.springframework.security.saml"/>
```

Spring Security integration

Filters of the SAML module need to be enabled as part of the Spring Security settings. In case SAML authentication should be the default authentication mechanism of the application set bean *samlEntryPoint* as the default entry point. Make sure that filter *samlFilter* is included as one of the custom filters. In case SP metadata should be generated automatically during first request to the application include also filter *metadataGeneratorFilter*. The configuration directive may for example look as follows:

```
<security:http entry-point-ref="samlEntryPoint">
    <security:custom-filter before="FIRST" ref="metadataGeneratorFilter"/>
    <security:custom-filter after="BASIC_AUTH_FILTER" ref="samlFilter"/>
</security:http>
```

4.3 Metadata configuration

SAML metadata is an XML document which contains information necessary for interaction with SAML-enabled identity or service providers. Document contains e.g. URLs of endpoints, information about supported bindings, identifiers and public keys. Typically one metadata document will be generated for your own service provider and sent to all identity providers you want to enable single sign-on with. Similarly, each identity provider will make it's own metadata available for you to import into your service provider application.

Each metadata document can contain definition for one or many identity or service providers and optionally can be digitally signed. Metadata can be customized either by direct modifications to the XML document, or using extended metadata. Extended metadata is added directly to the Spring configuration file and can contain additional options which are unavailable in the basic metadata document.

Service provider metadata

Service provider metadata contains keys, services and URLs defining SAML endpoints of your application. Metadata can be either generated automatically upon first request to the service, or it can be pre-created (see Chapter 5, *Administration user interface*). Once created metadata needs to be provided to the identity providers with whom we want to establish trust.

Automatic metadata generation

Automatic metadata generation is enabled by including the following filter in the Spring Security configuration:

```
<security:custom-filter before="FIRST" ref="metadataGeneratorFilter"/>
```

Filter is automatically invoked as part of the first request to a URL processed by Spring Security. In case there is no service provider metadata already specified (meaning property *hostedSPName* of the *metadata* bean is empty) filter will generate a new one.

By default metadata will be generated with the following values which can be customized by setting properties of the *metadataGeneratorFilter* bean:

Table 4.1. Metadata generator settings

Property	Description	Default value
`entityBaseURL`	Base URL to construct SAML endpoints from, needs to be a URL with protocol, server, port and context path.	Values from the request in format: *scheme://server:port/contextPath*
`entityAlias`	Local alias for the entityId which can be part of a simple URL path and contains only alphanum characters. See Section 4.4, “Entity alias”.	*defaultAlias*
`entityId`	Unique identifier of the service provider.	<entityBaseUrl>/saml/ metadata/ alias/<entityAlias>
`requestSigned`	Flag indicating whether this service signs authentication requests.	true
`wantAssertionSigned`	Flag indicating whether this service requires signed assertions.	true
`signingKey`	Key to include with usage "signing" in the metadata. Value will be set in ExtendedMetadata as `signingKey`.	Default private key from the KeyManager
`encryptionKey`	Key to include with usage "encryption" in the metadata. Value will be set in ExtendedMetadata as `encryptionKey`.	Default private key from the KeyManager
`tlsKey`	Key to include with usage "unspecified" in the metadata. Value will be set in ExtendedMetadata as `tlsKey`.	By default not included. Key is only included in metadata when it's different from signing and encryption keys.
`signMetadata`	When true generated metadata will be signed using XML Signature	true

Property	Description	Default value
	using certificate with alias of `signingKey`.	
`bindingsSSO`	Bindings to be included in the metadata for WebSSO profile. Supported values are: POST, Artifact and PAOS. Order of bindings in the property determines order of endpoints in the generated metadata.	Artifact, POST, PAOS
`bindingsHoKSSO`	Bindings to be included in the metadata for WebSSO Holder-of-Key profile. Supported values are: POST and Artifact. Order of bindings in the property determines order of endpoints in the generated metadata.	Artifact, POST
`bindingsSLO`	Bindings to be included in the metadata for Single Logout profile. Supported values are: POST and Redirect. Order of bindings in the property determines order of endpoints in the generated metadata.	POST, Redirect
`assertionConsumerIndex`	Index of assertion consumer point to be marked as default.	0
`includeDiscovery`	When true system will initialize discovery process during attempt to initialize single sign-on without pre-selected IDP.	true
`customDiscoveryURL`	When `includeDiscovery` is true value overrides default discovery request URL.	generated value for internal discovery service
`customDiscoveryResponseURL`	When `includeDiscoveryExtension` is true value overrides default discovery response URL.	generated value for entry point response URL
`includeDiscoveryExtension`	When true generated metadata will contain extension indicating that it's able to consume response from an IDP Discovery service.	false

Property	Description	Default value
`nameID`	Name identifiers to be included in the metadata. Supported values are: EMAIL, TRANSIENT, PERSISTENT, UNSPECIFIED and X509_SUBJECT. Order of NameIDs in the property determines order of NameIDs in the generated metadata.	EMAIL, TRANSIENT, PERSISTENT, UNSPECIFIED, X509_SUBJECT

Providing an empty collection or null value to properties *bindingsSSO*, *bindingsHoKSSO* and *bindingsSLO* will disable and remove the given profile. For example the following setting removes the holder-of-key profile from the generated metadata, forces artifact binding for single sign-on and redirect binding for single logout:

```
<bean class="org.springframework.security.saml.metadata.MetadataGenerator">
    <property name="bindingsSSO"><list><value>artifact</value></list></property>
    <property name="bindingsSLO"><list><value>redirect</value></list></property>
    <property name="bindingsHoKSSO"><list/></property>
</bean>
```

By default generated metadata will be digitally signed using the default credential specified in KeyManager (see Section 4.5, “Key management” for details). Digital signature can be disabled using property `signMetadata` of the *metadataGeneratorFilter* bean.

In case application is deployed behind a reverse-proxy or other mechanism which makes the URL at the application server different from the URL seen by client at least property `entityBaseURL` should be set to a value e.g. https://www.server.com:8080 For details about load balancing see Section 4.13, “Load balancing”.

Pre-configured metadata

In some situations it is beneficial to provide static version of the metadata document instead of the automatic generation. Need for manual changes in the metadata or fixing of production settings are some of those. Custom metadata document describing local SP application can be added by updating the *metadata* bean with correct *ExtendedMetadata*. Please follow these steps in order to do so:

- Generate and download metadata, e.g. using the *Metadata information -> Generate new service provider metadata* option in the sample application's administration UI or using instructions in the section called “Automatic metadata generation”.
- Store the metadata file as part of your project classpath, e.g. in *WEB-INF/classes/security/localhost_sp.xml*.
- Disable the automatic metadata generator by removing the following custom filter from the *securityContext.xml*:

  ```
  <security:custom-filter before="FIRST" ref="metadataGeneratorFilter"/>
  ```

- Include the SP metadata in the *metadata* bean and mark the entity as *local* in the extended metadata. Make sure to specify the *alias* property in case it was used during metadata generation.

  It is recommended to use the administration UI which also generates all the Spring declarations ready for inclusion in your *securityContext.xml*.

```
<bean class="org.springframework.security.saml.metadata.ExtendedMetadataDelegate">
    <constructor-arg>
        <bean class="org.opensaml.saml2.metadata.provider.FilesystemMetadataProvider">
            <constructor-arg>
                <value type="java.io.File">classpath:security/localhost_sp.xml</value>
            </constructor-arg>
            <property name="parserPool" ref="parserPool"/>
        </bean>
    </constructor-arg>
    <constructor-arg>
        <bean class="org.springframework.security.saml.metadata.ExtendedMetadata">
           <property name="local" value="true"/>
           <property name="alias" value="default"/>
           <property name="securityProfile" value="metaiop"/>
           <property name="sslSecurityProfile" value="pkix"/>
           <property name="signingKey" value="apollo"/>
           <property name="encryptionKey" value="apollo"/>
           <property name="requireArtifactResolveSigned" value="false"/>
           <property name="requireLogoutRequestSigned" value="false"/>
           <property name="requireLogoutResponseSigned" value="false"/>
           <property name="idpDiscoveryEnabled" value="true"/>
           <property name="idpDiscoveryURL"
              value="https://www.server.com:8080/context/saml/discovery/alias/default"/>
           <property name="idpDiscoveryResponseURL"
              value="https://www.server.com:8080/context/saml/login/alias/default?disco=true"/>
        </bean>
    </constructor-arg>
</bean>
```

Same instance of your application can include multiple statically declared local service providers each differentiated with it's own unique alias and entity ID. Each service provider can e.g. process a different domain or have different security key settings. This feature makes it possible to create multi-tenant applications with individual SAML settings for each of the tenants. In case multiple local SPs are declared, property *hostedSPName* of the *metadata* bean should be set to the entity ID of the default one.

For details about available settings of the ExtendedMetadata see Section A.1, “Extended metadata”.

Downloading metadata

Metadata describing the default local application can be downloaded from URL:

```
https://www.server.com:8080/context/saml/metadata
```

In case application is configured to contain multiple service providers metadata for each can be loaded by adding the alias:

```
https://www.server.com:8080/context/saml/login/alias/defaultAlias
```

URL for metadata download can be disabled by removing filter *metadataDisplayFilter* from the *securityContext.xml*.

Metadata is also available in the sample application's administration UI under *Metadata information -> selected SP*.

Identity provider metadata

Metadata for identity providers is imported to the *metadataManager* in a similar way as pre-configured SP metadata. Metadata containing one or many identity providers can be added by providing an URL or a file. Processing of metadata and processing of SAML messages can be customized using properties of *ExtendedMetadataDelegate* and *ExtendedMetadata*.

File-based metadata provider

File-based provider loads metadata from a file available in the filesystem or classpath.

```
<bean class="org.springframework.security.saml.metadata.ExtendedMetadataDelegate">
    <constructor-arg>
        <bean class="org.opensaml.saml2.metadata.provider.FilesystemMetadataProvider">
            <constructor-arg>
                <value type="java.io.File">classpath:security/idp.xml</value>
            </constructor-arg>
            <property name="parserPool" ref="parserPool"/>
        </bean>
    </constructor-arg>
    <constructor-arg>
        <bean class="org.springframework.security.saml.metadata.ExtendedMetadata"/>
    </constructor-arg>
</bean>
```

Metadata is automatically refreshed in intervals specified by properties *minRefreshDelay* and *maxRefreshDelay* of the *MetadataProvider* bean.

HTTP-based metadata provider

HTTP-based provider loads metadata from an URL.

```
<bean class="org.springframework.security.saml.metadata.ExtendedMetadataDelegate">
    <constructor-arg>
        <bean class="org.opensaml.saml2.metadata.provider.HTTPMetadataProvider">
            <constructor-arg>
                <value type="java.lang.String">http://idp.ssocircle.com/idp-meta.xml</value>
            </constructor-arg>
            <constructor-arg>
                <!-- Timeout for metadata loading in ms -->
                <value type="int">5000</value>
            </constructor-arg>
            <property name="parserPool" ref="parserPool"/>
        </bean>
    </constructor-arg>
    <constructor-arg>
        <bean class="org.springframework.security.saml.metadata.ExtendedMetadata"/>
    </constructor-arg>
</bean>
```

Metadata is automatically refreshed in intervals specified by properties *minRefreshDelay* and *maxRefreshDelay* of the *MetadataProvider* bean.

Alternatively class *org.opensaml.saml2.metadata.provider.FileBackedHTTPMetadataProvider* can be used to provide a backup in case URL is temporarily unavailable. File to use as backup is specified as third argument in the *MetadataProvider* bean constructor.

Signature verification

Importing of digitally signed metadata requires verification of signature's validity and trust. Metadata is not required to be signed by default. When present, signature is verified with PKIX algorithm and uses all public keys present in the configured *keyManager* as trust anchors. Make sure to include root CA certificate and intermediary CA certificates of the signature in your *keyStore*. For details see the section called “Importing public keys”.

You can limit certificates used to peform the verification by setting property *metadataTrustedKeys* of the *ExtendedMetadataDelegate* bean. The provided collection should contain aliases of keys to be used as trust anchors.

Signature verification can be disabled by setting property *metadataTrustCheck* to false in the *ExtendedMetadataDelegate* bean. Setting *metadataRequireSignature* to true will reject metadata unless it's digitally signed.

Extended metadata

Additional processing instructions related to SAML exchanges with the IDP can be defined in ExtendedMetadata bean. In case your metadata document contains multiple identity providers (in multiple EntityDescriptor elements) extended metadata can be set separately for each of them using a map with entity ids as keys, e.g.:

```
<bean class="org.springframework.security.saml.metadata.ExtendedMetadataDelegate">
    <constructor-arg>
        metadata provider bean
    </constructor-arg>
    <constructor-arg>
        <!-- Default extended metadata for entities not specified in the map -->
        <bean class="org.springframework.security.saml.metadata.ExtendedMetadata"/>
    </constructor-arg>
    <constructor-arg>
        <!-- Extended metadata for specific IDPs -->
        <map>
            <entry key="http://idp.ssocircle.com">
                <bean class="org.springframework.security.saml.metadata.ExtendedMetadata"/>
            </entry>
        </map>
    </constructor-arg>
</bean>
```

For details about available settings of the ExtendedMetadata see Section A.1, “Extended metadata”.

4.4 Entity alias

TODO

4.5 Key management

SAML exchanges involve usage of cryptography for signing and encryption of data. All interaction with cryptographic keys is done through interface *org.springframework.security.saml.key.KeyManager*. Default

implementation relies on a single JKS key store which contains all private and public keys. KeyManager must contain at least one private key which should be marked as default by using the alias of the private key as part of the KeyManager constructor.

Make sure that your configuration of SAML module contains bean keyManager with your custom key store and passwords.

Sample keystore

Sample application contains a default key store with a sample private certificate usable for test purposes. The key store is defined as:

```
<bean id="keyManager" class="org.springframework.security.saml.key.JKSKeyManager">
    <constructor-arg value="classpath:security/samlKeystore.jks"/>
    <constructor-arg type="java.lang.String" value="nalle123"/>
    <constructor-arg>
        <map>
            <entry key="apollo" value="nalle123"/>
        </map>
    </constructor-arg>
    <constructor-arg type="java.lang.String" value="apollo"/>
</bean>
```

The first argument points to the used key store file, second contains password for the keystore, third then map with passwords for private keys with alias-password value pairs. Alias of the default certificate is the last parameter.

Generating and importing private keys

Private keys (with either self-signed or CA signed certificates) are used to digitally sign SAML messages, encrypt their content and in some cases for SSL/TLS Client authentication of your service provider application. SAML Extension ships with a default private key in the *samlKeystore.jks* with alias *apollo* which can be used for initial testing, but for security reason should be replaced with your own key in early development stages.

In case your IDP doesn't require keys signed by a specific certification authority you can generate your own self-signed key using the Java utility *keytool*, e.g. with:

```
keytool -genkeypair -alias some-alias -keypass changeit -keystore samlKeystore.jks
```

The keystore will now contain additional PrivateKeyEntry with alias mykey which can be imported to the *keyManager* in your *securityContext.xml*.

Keys signed by certification authorities are typically provided in .p12/.pfx format (or can be converted to such using OpenSSL) and imported to Java keystore with, e.g.:

```
keytool -importkeystore -srckeystore key.p12 -srcstoretype PKCS12 -srcstorepass password \
   -alias some-alias -destkeystore samlKeystore.jks -destalias some-alias -destkeypass changeit
```

The following command can be used to determine available alias in the p12 file:

```
keytool -list -keystore key.p12 -storetype pkcs12
```

Importing public keys

Cryptographic material used to decrypt incoming data and verify trust of signatures in SAML messages and metadata is stored either in metadata of remote entities or in the *keyManager*. In order to import additional trusted key to the keystore run, e.g.:

```
keytool -importcert -alias some-alias -file key.cer -keystore samlKeystore.jks
```

Imported keys can be referenced in *ExtendedMetadataDelegate* and *ExtenedMetadata* beans, for details see the section called “Signature verification” and Section 4.6, “Security profiles”.

Loading SSL/TLS certificates

Direct SSL/TLS connections (used with HTTP-Artifact binding) require verification of the public key presented by the server. The SSL Extractor utility [https://github.com/vschafer/ssl-extractor] can be used to extract certificates presented by an SSL/TLS endpoint, e.g. with:

```
java -jar sslextractor-0.9.jar www.google.com 443
```

The certificates are stored as .cer files and can be imported to the keystore as a usual public key. For details about configuring of trust for SSL/TLS connections see Section 4.6, “Security profiles”.

4.6 Security profiles

Exchanges of messages between identity providers and service providers with SAML protocol involves usage of digital signatures. Signatures are typically constructed using means of asymetric cryptography and public key infrastructure with public and private keys signed by trusted certification authorities. Signatures are either applied directly to parts of XML representation of SAML messages using XML Signature or are part of the transport layer used to deliver the message like SSL/TLS.

Verification of signatures is executed in two phases. Signature is first checked for validity by comparing digital hash included as part of the signature with value calculated from the content. Subsequently it is verified whether party who created the signature is trusted by the recipient. Module provides two mechanisms for defining which signatures should be accepted - metadata interoperability mode and PKIX mode.

Security profiles are defined in Extended Metadata of your local SP. Profile can be defined separately for XML Signatures using property *securityProfile* and for SSL/TLS Signatures using property*sslSecurityProfile*. Value of both properties can be either *metaiop* or *pkix*. For details about using Extended Metadata see Section 4.3, “Metadata configuration”, for reference of allowed values see Section A.1, “Extended metadata”.

Metadata interoperability profile (MetaIOP)

With MetaIOP mode certificates are not checked for expiration or revocation and certificate paths are not verified. This means that it does not matter which certification authority issued the certificate, as the fact whether the certificate is trusted or not is conveyed using other mechanisms (e.g. by secure metadata exchange or digital signature of metadata itself).

Signature is deemed trusted when the certificate used to create it is included in one of the following places:

- Key with usage of signing or unspecified in entity metadata of a remote entity
- Signing key specified in property *signingKey* of extended metadata of a remote entity

MetaIOP is the default profile for verification of XML signatures. For details about this profile see the specification [http://docs.oasis-open.org/security/saml/Post2.0/sstc-metadata-iop.pdf].

PKIX profile

With PKIX profile trust of signature certificates is verified based on a certificate path between trusted CA certificates and the certificate in question. Certificate is trusted when it's possible to construct path from a trusted certificate to the validated one. With this profile certificate expiration and revocation can be checked.

Trusted keys (anchors) for PKIX verification of signatures are combined from the following places:

- Key with usage of signing or unspecified in entity metadata of a remote entity
- Signing key specified in property *signingKey* of extended metadata of a remote entity
- All keys specified in *trustedKeys* set of extended metadata of a remote entity

Custom profile

Engine used to verify trust of signatures for given combination of SP/IDP is created in methods *populateTrustEngine* and *populateSSLTrustEngine* of interface *org.springframework.security.saml.context.SAMLContextProvider* and can be overriden with custom implementation. See Section 4.12, “Context provider” for details on context customization.

4.7 Single sign-on process

SP initialized SSO process can be started in two ways:

- User accesses a resource protected by Spring Security which initializes SAMLEntryPoint
- User is redirected to the SSO page at e.g. https://www.server.com/context/saml/login/alias/defaultAlias

After identification of IDP to use (for details see Section 4.8, “IDP selection”) SAML Extension creates an AuthnRequest SAML message and sends it to the selected IDP. Both construction of the AuthnRequest and binding used to send it can be customized using *WebSSOProfileOptions* object. SAMLEntryPoint determines *WebSSOProfileOptions* configuration to use by calling method *getProfileOptions*. Default implementation returns value specified in property *defaultOptions*. Method can be overriden to provide custom logic for SSO initialization.

The following SSO parameters can be customized using the *WebSSOProfileOptions* object:

Table 4.2. org.springframework.security.saml.websso.WebSSOProfileOptions parameters

Property	Value

Default settings for *WebSSOProfileOptions* can be specified in bean *samlEntryPoint* of your securityContext.xml, e.g.:

```
<bean id="samlEntryPoint" class="org.springframework.security.saml.SAMLEntryPoint">
    <property name="defaultProfileOptions">
```

```
        <bean class="org.springframework.security.saml.websso.WebSSOProfileOptions">
            <property name="binding" value="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"/>
            <property name="includeScoping" value="false"/>
        </bean>
    </property>
</bean>
```

4.8 IDP selection

TODO

4.9 Logout process

TODO

4.10 Authentication object

Successful authentication using SAML token results in creation of an *Authentication* object by the *SAMLAuthenticationProvider*. By default instance of *org.springframework.security.providers.ExpiringUsernameAuthenticationToken* is created. Content of the resulting object can be customized by setting properties of the *samlAuthenticationProvider* bean in the *securityContext.xml*. Instance of *org.springframework.security.saml.userdetails.SAMLUserDetailsService* can be provided to supply application specific information about the authenticated user. Property *forcePrincipalAsString* can be used to force String value of the *principal* property. The Authentication object is available in pages secured with Spring Security using *SecurityContextHolder.getContext().getAuthentication()* and is populated with the following values:

Table 4.3. ExpiringUsernameAuthenticationToken values.

Property	**Value**
Principal	When forcePrincipalAsString = false AND userDetail = null (default) - *NameID* object included in the SAML Assertion (*credential.getNameID()* of type *org.opensaml.saml2.core.NameID*)
Principal	When forcePrincipalAsString = true - *String* value of *NameID* included in the SAML Assertion (*credential.getNameID().getValue()* of type java.lang.String)
Principal	When forcePrincipalAsString = false AND userDetail != null - *UserDetail* object returned from the *SAMLUserDetailsService*
Credentials	SAML authentication object including entity ID of local and remote entity, name ID, assertion and relay state (*org.springframework.security.saml.SAMLCredential*)
Authorities	Result of *getAuthorities()* call on the *UserDetails* object returned from *SAMLUserDetailsService*, empty list when there's no *UserDetail* object available.
Expiration	Value of *SessionNotOnOrAfter* in the SAML Assertion when avaialble, null otherwise. *Authentication* object will start returning false on the *isAuthenticated()* after the expiration time.

Custom implementation of the *SAMLUserDetailsService* can be provided as property *userDetails* of the *SAMLAuthenticationProvider*. Implementation can perform operation such as parsing of attributes present in the SAML Assertion, e.g.:

```
package fi.schafer.test.saml;

import org.opensaml.saml2.core.Attribute;
import org.opensaml.xml.XMLObject;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.saml.SAMLCredential;
import org.springframework.security.saml.userdetails.SAMLUserDetailsService;

public class TestUserDetails implements SAMLUserDetailsService {

    @Override
    public Object loadUserBySAML(SAMLCredential credential) throws UsernameNotFoundException {
        Attribute accountID = credential.getAttributeByName("accountID");
        if (accountID == null || accountID.getAttributeValues() == null
                    || accountID.getAttributeValues().size() == 0) {
            return null;
        }
        XMLObject attributeValue = accountID.getAttributeValues().iterator().next();
        return attributeValue.getDOM().getTextContent();
    }

}
```

Population of the authentication object can be further customized by overriding of the *getUserDetails*, *getPrincipal*, *getEntitlements* and *getExpirationDate* methods in the *SAMLAuthenticationProvider*.

4.11 Authentication log

Key events such as single sign-on and single logout initialization, success or failure can be logged for creation of an audit trail. A custom logger can be created by implementing interface *org.springframework.security.saml.log.SAMLLogger* and including it's bean in the securityContext.xml, e.g.:

```
<bean id="samlLogger" class="org.springframework.security.saml.log.SAMLDefaultLogger"/>
```

Two basic implementations are provided by default:

- org.springframework.security.saml.log.SAMLEmptyLogger

  Doesn't perform any logging, simply ignores all events.

- org.springframework.security.saml.log.SAMLDefaultLogger

  Logs events as INFO level messages to the standard log configurable as described in Section 7.1, “Logging”. Setting property *logMessages* to *true* will include content of the SAML messages as part of the log.

4.12 Context provider

Context provider populates information about the local service provider (your application) such as entityId, role, metadata, security keys, SSL credetials and trust engines for verification of

signatures and SSL/TLS connections. Once populated context is made available to all components participating in processing of the incoming or outgoing SAML messages. ContextProvider can customized to alter behavior of the SAML Extension. The default implementation *org.springframework.security.saml.context.SAMLContextProviderImpl* relies on information available in the ExtendedMetadata and performs the followign steps for creation of the context:

- Locate entityId of the local SP by parsing part of the URL after */alias/* (e.g. myAlias in https://www.myserver.com/saml_extension/saml/sso/alias/myAlias?idp=myIdp) and match it with property *alias* specified in the ExtendedMetadata. In case URL doesn't contain any alias part the default service provider configured with property *hostedSPName* on the *metadata* bean is used.
- Populate credential used to decrypt data sent to this service provider. In case ExtendedMetadata specifies property *encryptionKey* it will be used as an alias to lookup a private key from the *keyManager* bean. Otherwise defaultKey of the *keyManager* bean will be used.
- Populate credential used for SSL/TLS client authentication. In case ExtendedMetadata specifies property *tlsKey* it will be used as an alias to lookup key from *keyManager* bean. Otherwise no credential will be provided for client authentication.
- Populate trust engine for verification of signatures. Depending on *securityProfile* setting in the ExtendedMetadata trust engine based on either the section called “Metadata interoperability profile (MetaIOP)” or the section called “PKIX profile” is created.
- Populate trust engine for verification of SSL/TLS connections. Depending on *sslSecurityProfile* setting in the ExtendedMetadata trust engine based on either the section called “Metadata interoperability profile (MetaIOP)” or the section called “PKIX profile” is created.

During initialization of SSO ContextProvider is also requested to provide metadata of the peer IDP. System performs these steps to locate peer IDP to use:

- Load parameter *idp* of the HttpRequest object and try to locate peer IDP by the entityId. When there's no *idp* parameter provided system will either start IDP discovery process (when enabled in the ExtendedMetadata of the local SP) or use the default IDP specified in the *metadata* bean.

4.13 Load balancing

SAML Extension can be deployed in scenarios where mutliple back-end servers process SAML requests forwarded by a reverse-proxy or a load balancer. SSL termination proxies which communicate using an unencrypted channel between the proxy and back-end servers are also supported. In order to confugure SAML Extension for deployment behing a load balancer or a reverse-proxy please follow these steps:

- Make sure that your reverse-proxy or load-balancer is configured to use sticky sessions. Information about e.g. sent requests is stored within user's HTTP session and sending of response to another back-end node would make the original request data unavailable and fail the validation. Sticky session are not necessary in case only IDP-initialized SSO is used.
- Provide information about front-end URL to the back-end servers by changing the *contextProvider* bean implementation in your *securityContex.xml* to class *org.springframework.security.saml.context.SAMLContextProviderLB*:

```
<bean id="contextProvider" class="org.springframework.security.saml.context.SAMLContextProviderLB">
    <property name="scheme" value="https"/>
    <property name="serverName" value="www.myserver.com"/>
    <property name="serverPort" value="443"/>
    <property name="includeServerPortInRequestURL" value="false"/>
    <property name="contextPath" value="/spring-security-saml2-sample"/>
</bean>
```

This setting enables extension to correctly form all generated URLs and verify endpoints of the incoming SAML messages.

- In case you use automatically generated metadata make sure to configure *entityBaseUrl* matching the front-end URL in your *metadataGeneratorFilter* bean:

```
<bean id="metadataGeneratorFilter"
        class="org.springframework.security.saml.metadata.MetadataGeneratorFilter">
    <constructor-arg>
        <bean class="org.springframework.security.saml.metadata.MetadataGenerator">
            <property name="entityBaseURL" value="https://www.myserver.com/spring-security-saml2-sample"/>
        </bean>
    </constructor-arg>
</bean>
```

5. Administration user interface

TODO

6. IDP integration guide

This chapter provides step-by-step guides for basic configuration of SAML Extension with specific IDP products. Integration can be further configured with settings discussed in previous chapters.

6.1 Active Directory Federation Services 2.0 (ADFS)

ADFS 2.0 supports SAML 2.0 in IDP mode and can be easily integrated with SAML Extension for both SSO and SLO. Before starting with the configuration make sure that the following pre-requisites are satisfied:

- Install AD FS 2.0 (http://www.microsoft.com/en-us/download/details.aspx?id=10909)
- Run AD FS 2.0 Federation Server Configuration Wizard in the AD FS 2.0 Management Console
- Make sure that DNS name of your Windows Server is available at your SP and vice-versa
- Install a Java container (e.g. Tomcat) for deployment of the SAML 2 Extension
- Configure your container to use HTTPS, this is required by AD FS (http://tomcat.apache.org/tomcat-6.0-doc/ssl-howto.html)

Initialize IDP metadata

- Download AD FS 2.0 metadata from *https://server/FederationMetadata/2007-06/FederationMetadata.xml*
- Store the downloaded content to saml2-sample/WEB-INF/src/main/resources/security/FederationMetadata.xml
- Modify bean *metadata* in *securityContext.xml* and replace *classpath:security/idp.xml* with *classpath:security/FederationMetadata.xml* and add property *metadataTrustCheck* to *false* to skip signature validation:

```
<bean class="org.springframework.security.saml.metadata.ExtendedMetadataDelegate">
    <constructor-arg>
        <bean class="org.opensaml.saml2.metadata.provider.FilesystemMetadataProvider">
            <constructor-arg>
                <value type="java.io.File">classpath:security/FederationMetadata.xml</value>
            </constructor-arg>
            <property name="parserPool" ref="parserPool"/>
        </bean>
    </constructor-arg>
    <constructor-arg>
        <bean class="org.springframework.security.saml.metadata.ExtendedMetadata"/>
    </constructor-arg>
    <property name="metadataTrustCheck" value="false"/>
</bean>
```

Initialize SP metadata

- Deploy SAML 2 Extension war archive from *saml2-sample/target/spring-security-saml2-sample.war*
- Open browser at e.g. https://server:port/spring-security-saml2-sample, make sure to use HTTPS protocol, system will automatically generate metadata document
- Click Metadata information, select item with your server name in the Service providers list
- Store content of the Metadata field to a document metadata.xml and upload it to the AD FS server
- In AD FS 2.0 Management Console select "Add Relying Party Trust"

- Select "Import data about the relying party from a file" and select file created earlier, select Next
- System may complain that some content of metadata is not supported, you can safely ignore this warning
- Continue with the wizard, on the "Ready to Add Trust" make sure that tab endpoints contains multiple endpoing values, if not verify that your metadata was generated with https protocol in their URLs
- Leave "Open the Edit Claim Rules dialog" checkbox checked and finish the wizard
- Select "Add Rule", choose "Send LDAP Attributes as Claims" and press Next
- Add NameID as "Claim rule name", choose "Active Directory" as Attribute store, choose "SAM-Account-Name" as LDAP Attribute and "Name ID" as "Outgoing claim type", finish the wizard and confirm the claim rules window
- Open the provider by double-clicking it, select tab Advanced and change "Secure hash algorithm" to SHA-1

Test SSO

Open SAML Extension at https://localhost:8443/spring-security-saml2-sample, select your AD FS server and press login. In case Artifact binding is used and SSL/TLS certificate of your AD FS is not already trusted you have to import it to your samlKeystore.jks by following instructions in the error report.

7. Troubleshooting

7.1 Logging

SAML Extension uses SLF4J framework [http://www.slf4j.org/] for logging. The same applies to the underlaying OpenSAML library. The sample application by default uses log4j version 1.2 binding for SLF4J.

You can enable debug logging by modifying file *saml2-sample/src/main/resources/log4j.properties* and adding:

```
log4j.logger.org.springframework.security.saml=DEBUG
log4j.logger.org.opensaml=DEBUG
```

For details about using other logging frameworks please consult the SLF4J manual [http://www.slf4j.org/manual.html].

7.2 Common problems

Appendix A. Configuration reference

This chapter provides reference for settings available in configuration beans of the SAML module.

A.1 Extended metadata

Extended metadata provides additional settings for customization of IDP and SP behavior. Bean can be found in package *org.springframework.security.saml.metadata.ExtendedMetadata*. For details on setting up metadata please consultSection 4.3, “Metadata configuration”.

Table A.1. Extended metadata settings

Property	Default	Entities	Description
local	false	both	
alias		local	
idpDiscoveryEnabled	false	local	
idpDiscoveryURL		local	
idpDiscoveryResponseURL		local	
ecpEnabled	false	local	
securityProfile	metaiop	local	
sslSecurityProfile	pkix	local	
signingKey		both	
encryptionKey		both	
tlsKey		both	
trustedKeys		both	
requireLogoutRequestSigned		both	
requireLogoutResponseSigned		both	
requireArtifactResolveSigned		both	