SecurityConfiguration

File(s): conf/credentials.xml, conf/relying-party.xml, conf/idp.properties

Format: Native Spring / Deprecated Custom Schema

Legacy V2 File(s): conf/relying-party.xml

- Overview
- Credentials
- Signing and Encryption Enablement
- Signing and Encryption Configuration
- Trust Engines
- Security Rules
- Reference
  - Beans
  - Properties
- V2 Compatibility
  - Legacy relying-party.xml and global security defaults
- Notes

Overview

There are multiple aspects of configuration that fall under the heading of security:

- credentials (keys and certificates)
- configuration and control of algorithms to use for signing or encryption (as well as verifying signatures and decryption)
- "TrustEngines" used to evaluate signatures and certificates during message processing
- other steps that run during profile execution to evaluate messages

All of these settings are associated with profile or relying party behavior, so they are configured as part of the more general RelyingPartyConfiguration service, and as such are reloaded when that service is reloaded.

When using a legacy V2 relying-party.xml configuration file, much of the content below does not apply, as noted in the compatibility section near the end of this topic.

Credentials

In most cases, it's not necessary to modify your credential configuration except to ensure the idp.properties settings that identify keys and certificates are set correctly, which is true after an installation when using the automatically generated credentials. The actual beans that load these files are in credentials.xml and have the following required names:

- shibboleth.DefaultSigningCredential
- shibboleth.DefaultClientTLSCredential
- shibboleth.DefaultEncryptionCredentials

The first two are instances of the Credential class and specify the default keypair to use for signing and client TLS respectively. Client TLS isn't common, but may occur in rare instances such as inbound HTTP-Artifact binding messages or in future features such as outbound SOAP logout messages.

The third is a list that contains one or more instances of that same class, used for encryption (actually for decryption; this is the public key used to encrypt data for the IdP and the private key to decrypt it with).

If all you need is a single keypair for each function, then nothing needs to be done to this file. This is the strongly recommended approach. If you need to create additional signing credentials to use as overrides to behavior for specific relying parties, you may want to copy the existing bean and assign it a different id to refer to later (see below under Signing and Encryption).

In the event that you want to perform a controlled rollover of an encryption keypair, that involves the configuration of multiple keypairs (old and new) into the IdP, and this is accommodated with pre-existing definitions you can use by uncommenting the second keypair bean in the list and setting a second set of properties to point to the keypair. Support for decrypting with either key will be handled automatically when both are configured.

Mechanisms exist for defining advanced Credentials and are discussed here.

Signing and Encryption Enablement

For most situations, the default behavior of each ProfileConfiguration is appropriate when it comes to what to sign or encrypt. A common use case for creating overrides for particular relying parties is to either turn off XML Encryption, or possibly alter which part of a SAML message is to be signed. This is controlled with a set of properties on all SAMLProfileConfiguration and SAML2ProfileConfiguration objects, each of which is a condition that evaluates to true or false to determine what to sign or encrypt.

In most cases, you can set the properties to "true" or "false" to turn them on or off. An example follows:
Per-Profile Signing or Encryption Options

<!-- excerpt of relying-party.xml -->

```xml
<bean parent="RelyingPartyByName" c:relyingPartyIds="https://sp.example.org">
  <property name="profileConfigurations">
    <list>
      <bean parent="SAML2.SSO" p:signAssertions="true" p:encryptAssertions="false" />
    </list>
  </property>
</bean>
```

For more advanced scenarios, you can configure an arbitrary condition function to control the decision. Most commonly this would be used in conjunction with a built-in condition that works by deciding whether to sign or encrypt based on the security of the underlying communication channel. It is equivalent to the V2 setting of "conditional" in the legacy configuration format. This isn't typically needed, but an example follows:

Conditional Signing or Encryption

<!-- excerpt of relying-party.xml -->

```xml
<bean id="SignNoIntegrity" class="org.opensaml.profile.logic.NoIntegrityMessageChannelPredicate"/>
<bean id="EncryptNoConfidentiality" class="org.opensaml.profile.logic.NoConfidentialityMessageChannelPredicate"/>

<bean parent="RelyingPartyByName" c:relyingPartyIds="https://sp.example.org">
  <property name="profileConfigurations">
    <list>
      <bean parent="SAML2.SSO" p:signAssertions-ref="SignNoIntegrity" p:encryptAssertions-ref="EncryptNoConfidentiality" />
    </list>
  </property>
</bean>
```

Finally, a new feature exists to do encryption "opportunistically", that is, to encrypt whenever possible (meaning a compatible key is found in the peer's metadata to encrypt with) but to skip encryption otherwise. For those wishing to avoid having to constantly adapt their system to deal with new services that don't support encryption, this is an option you can use, understanding that this means a loss of explicit control over whether XML Encryption takes place. This feature is enabled with the `idp.encryption.optional` property.

Signing and Encryption Configuration

There are several levels of SecurityConfiguration objects to control the signing, validation, encryption, and decryption processes. These configurations include things like the keypairs to use, preferred/default algorithms, and algorithm whitelists or blacklists to enforce. The configurations can be derived from the underlying library defaults, a global IdP default, or a per-profile override.

In most cases you will leave the defaults in place, or very occasionally define a special configuration to use for specific profiles and/or relying parties.

The default objects that make up these configurations are defined in `system/conf/relying-party-system.xml` and are defined to match the defaults in the OpenSAML libraries. You won't need to (and should not) modify these beans, but you might need to copy them into your own files to make changes to them or create an override in particularly advanced cases. More commonly, you can `inherit` from some of them in order to override specific bits and leave the other defaults in place, which is simpler than creating an entire `SecurityConfiguration` object from scratch.

You can override the default bean name that is used as a global default by setting the `idp.security.config` property. That effectively ignores the system-defined objects in favor of something else. You can also override this on a per-profile, per-relying-party basis by adding a `p:securityConfiguration-ref` attribute to a profile bean.

The special/common case of controlling the use of SHA-1 and SHA-256 digest algorithms while signing is simplified because we have predefined beans that toggle between those two algorithms at a global level, with a property to switch between them (idp.signing.config).

Similarly, V3.4 adds predefined beans for toggling between two data encryption families, AES-CBC and AES-GCM. The former is widespread and well-supported, and vulnerable to attacks that can reveal the data. The latter is not widely supported but is the only practical algorithm that remains secure. A new property will switch between them globally (idp.encryption.config) but in practice that won't work all that well because so many SPs will not support GCM (though that should be considered broken).

The following is an example (specific to V3.4) to follow if you need to specify the use of SHA-1 instead of SHA-256 for a specific relying party.
Per-Profile Signing Algorithm

<!-- excerpt of relying-party.xml -->
<util:list id="shibboleth.RelyingPartyOverrides">
    <bean parent="RelyingPartyByName" c:relyingPartyIds="https://shalonly.example.org">
        <property name="profileConfigurations">
            <list>
                <bean parent="Shibboleth.SSO" p:securityConfiguration-ref="shibboleth.SecurityConfiguration.SHA1" />
                <bean parent="SAML1.AttributeQuery" p:securityConfiguration-ref="shibboleth.SecurityConfiguration.SHA1" />
                <bean parent="SAML1.ArtifactResolution" p:securityConfiguration-ref="shibboleth.SecurityConfiguration.SHA1" />
                <bean parent="SAML2.SSO" p:securityConfiguration-ref="shibboleth.SecurityConfiguration.SHA1" />
                <bean parent="SAML2.ECP" p:securityConfiguration-ref="shibboleth.SecurityConfiguration.SHA1" />
                <bean parent="SAML2.Logout" p:securityConfiguration-ref="shibboleth.SecurityConfiguration.SHA1" />
            </list>
        </property>
    </bean>
</util:list>

Similarly, you can do the same with encryption for GCM:

Per-Profile Encryption Algorithm

<!-- excerpt of relying-party.xml -->
<util:list id="shibboleth.RelyingPartyOverrides">
    <bean parent="RelyingPartyByName" c:relyingPartyIds="https://shalonly.example.org">
        <property name="profileConfigurations">
            <list>
                <bean parent="SAML2.SSO" p:securityConfiguration-ref="shibboleth.SecurityConfiguration.GCM" />
                <bean parent="SAML2.ECP" p:securityConfiguration-ref="shibboleth.SecurityConfiguration.GCM" />
            </list>
        </property>
    </bean>
</util:list>

A more advanced case: to specify a custom keypair for signing for a particular relying party, you might create a bean in credentials.xml called "ObnoxiousVendorCredential", and then do the following (there are many ways to lay things out, this is just an example of what's possible):
Per-Profile Credential

<!-- excerpt of relying-party.xml -->

```xml
<bean id="ObnoxiousSecurityConfig" parent="shibboleth.DefaultSecurityConfiguration">
    <property name="signatureSigningConfiguration">
        <bean parent="shibboleth.SigningConfiguration.SHA256" p:signingCredentials-ref="ObnoxiousVendorCredential" />
    </property>
</bean>

<util:list id="shibboleth.RelyingPartyOverrides">
    <bean parent="RelyingPartyByName" c:relyingPartyIds="https://obnoxious.vendor.com/sp">
        <property name="profileConfigurations">
            <list>
                <bean parent="Shibboleth.SSO" p:securityConfiguration-ref="ObnoxiousSecurityConfig" />
                <bean parent="SAML1.AttributeQuery" p:securityConfiguration-ref="ObnoxiousSecurityConfig" />
                <bean parent="SAML1.ArtifactResolution" p:securityConfiguration-ref="ObnoxiousSecurityConfig" />
                <bean parent="SAML2.SSO" p:securityConfiguration-ref="ObnoxiousSecurityConfig" />
                <bean parent="SAML2.ECP" p:securityConfiguration-ref="ObnoxiousSecurityConfig" />
                <bean parent="SAML2.Logout" p:securityConfiguration-ref="ObnoxiousSecurityConfig" />
                <bean parent="SAML2.AttributeQuery" p:securityConfiguration-ref="ObnoxiousSecurityConfig" />
                <bean parent="SAML2.ArtifactResolution" p:securityConfiguration-ref="ObnoxiousSecurityConfig" />
            </list>
        </property>
    </bean>
</util:list>
```

Typically you want to think about what you need to customize, and then consolidate definitions of beans as much as possible to simplify the XML.

Trust Engines

TrustEngines are a type of object used to evaluate signatures and TLS certificates and make up the security backbone of the IdP. The V3 configuration, as V2 did, defaults to a "chain" of two strategies:

- looking for a matching public key inside SAML metadata corresponding to a message issuer
- verifying a certificate indirectly by performing PKIX path validation, generally against trust rules supplied in SAML metadata extensions defined by the Shibboleth Project many years ago

The recommended approach is the first one. The second is provided solely for compatibility and while it is used by some deployers, we expect to eventually disable it by default in a future release. It is not deprecated, but it is not a recommended choice.

In the event that you wish to replace the "chain" with a reference to one of the TrustEngines alone, you can set one or both of the `idp.trust.signatures` and `idp.trust.certificates` properties in `idp.properties` to appropriate bean names (noted below and in the property file's comments).

Security Rules

The last area of security configuration involves the steps to follow after a message is received to check the content and perform steps like replay detection, freshness detection, signature checking, evaluation of client certificate authentication for server to server communication, etc. This is also where profiles that require authentication of the request, such as Attribute Queries, enforce that requirement.

Internally, these rules are run as part of something called the inboundInterceptorFlows collection, which is a list of "interceptor" webflows to run before accepting a request for further processing. While this is in principle an extensible feature, in most cases you don't need to even think about this mechanism, and it's configured for you for all the built-in profile beans to do the "right" thing, provided you inherit any custom profile beans you want to use for overrides from the built-in beans listed in the `RelyingPartyConfiguration` (e.g., "Shibboleth.SSO", "SAML2.AttributeQuery").

Normally you need never customize these flows directly, other than via a few properties defined for that purpose, such as `idp.policy.messageLifetime`.

Reference

Beans

| shibboleth.DefaultSigningCredential | Credential | Keypair used for signing, usually including a certificate |

```
<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Default</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>idp.cookie.secure</td>
<td>Boolean</td>
<td>false</td>
<td>If true, all cookies issued by the IdP (not including the container) will be limited to TLS</td>
</tr>
<tr>
<td>idp.cookie.httpOnly</td>
<td>Boolean</td>
<td>true</td>
<td>If true, all cookies issued by the IdP (not including the container) will contain the HttpOnly property</td>
</tr>
<tr>
<td>idp.cookie.domain</td>
<td>String</td>
<td></td>
<td>Overrides the domain of any cookies issued by the IdP, not including the container</td>
</tr>
<tr>
<td>idp.cookie.path</td>
<td>String</td>
<td></td>
<td>Overrides the path of any cookies issued by the IdP, not including the container</td>
</tr>
<tr>
<td>idp.cookie.maxAge</td>
<td>Integer</td>
<td>31536000</td>
<td>Lifetime in seconds of cookies issued by the IdP that are meant to span sessions (365 days)</td>
</tr>
<tr>
<td>idp.sealer.storeType</td>
<td>String</td>
<td>JCEKS</td>
<td>Type of Java keystore used for IdP's internal AES encryption key</td>
</tr>
<tr>
<td>idp.sealer.updateInterval</td>
<td>Duration</td>
<td>PT15M</td>
<td>Time between checks for a new AES key version</td>
</tr>
<tr>
<td>idp.sealer.aliasBase</td>
<td>String</td>
<td>secret</td>
<td>Case insensitive name of keystore alias prefix used in AES keystore (the entries will be suffixed by the key version number)</td>
</tr>
<tr>
<td>idp.sealer.storeResource</td>
<td>Resource path</td>
<td></td>
<td>Keystore resource containing AES encryption key, usually a file path</td>
</tr>
</tbody>
</table>
idp.sealer.versionResource Resource path Resource that tracks the "active" AES encryption key version, usually a file path

idp.sealer.storePassword String Keystore password unlocking AES encryption keystore, typically set during installation

idp.sealer.keyPassword String Key password unlocking AES encryption key, typically set to the same as the previous property and set during installation

idp.signing.key Resource path Resource containing private key for signing, typically a file in the credentials directory

idp.signing.cert Resource path Resource containing the public key certificate inserted into signed messages, typically a file in the credentials directory

idp.encryption.key Resource path Resource containing a private key for decryption, typically a file in the credentials directory

idp.encryption.cert Resource path Resource containing a public key certificate given to others needing to encrypt data for the IdP, typically a file in the credentials directory

idp.encryption.key.2 Resource path Resource containing an alternate private key for decryption, generally unused except while changing decryption keys

idp.encryption.cert.2 Resource path Resource containing an alternate public key certificate, generally unused except while changing decryption keys

idp.security.config Bean ID DefaultSecurityConfiguration Name of Spring bean supplying the default SecurityConfiguration

idp.signing.config Bean ID SigningConfiguration.SHA256 Name of Spring bean supplying the default SignatureSigningConfiguration

idp.encryption.config 3.4 Bean ID EncryptionConfiguration.CBC Name of Spring bean supplying the default EncryptionConfiguration

idp.trust.signatures Bean ID ChainingSignatureTrustEngine Name of Spring bean for the trust engine used to verify signatures

idp.trust.certificates Bean ID ChainingX509TrustEngine Name of Spring bean for the trust engine used to verify TLS certificates

idp.encryption.optional Boolean false If true, failure to locate an encryption key to use, when enabled, won't result in request failure

idp.errors.detailed Boolean false If true, more detailed error information may be returned in profile responses, which could leak useful information in rare cases

idp.errors.signed Boolean true When message signing is enabled, controls whether to sign responses that signal errors as opposed to successful outcomes

idp.policy.messageLifetime Duration PT3M Default freshness window for accepting timestamped messages

idp.policy.clockSkew Duration PT3M Default allowance for clock differences between systems

idp.artifact.secureChannel Boolean true If true, skips signing/encryption when the message will be passed by reference (via artifact in SAML terms)

V2 Compatibility

The V3 IdP supports a small subset of the deprecated V2 relying-party and security schemas for configuring the keys and certificates (credentials) to be used for signing using the defaultSigningCredentialRef and signingCredentialRef attributes along with the associated <security:Credential> element(s).

V3 continues to honor the legacy signing and encryption enumeration (always, conditional, never) used for controlling what to sign and encrypt when using the legacy configuration syntax. The enumeration is not explicitly supported when using the new native Spring syntax.

Note that when a legacy configuration is used, the content of the credentials.xml file, and the associated properties in idp.properties are ignored in favor of the credential settings in the legacy file. You can see this explicitly if you examine the content of services.xml and note that the shibboleth. LegacyRelyingPartyResolverResources bean does not include the credentials.xml file at all.

The rest of the security-related content of a legacy relying-party.xml file is not supported and is replaced by default configuration settings. This includes all the security policy "rules" at the bottom of the file. Changing other default behaviors and most advanced options require the use of native Spring syntax and the retirement of the legacy configuration.

This notably includes controlling signing or encryption algorithms on a per-relying-party basis, although it is in many cases possible to control this by adding extensions to an SP's metadata to get the result you need.

The new support for XML decryption, typically encountered in SAML only when processing <LogoutRequest> messages that contain an encrypted subject ID, is also not active when using a legacy configuration.

Legacy relying-party.xml and global security defaults
As mentioned above, when using a legacy relying-party.xml, advanced per-relying-party security configuration is not supported. However, via the use of some additional Spring wiring, one can customize the global security configuration defaults. This will affect all relying parties.

A common requirement is to revert the IdP back to the use of SHA-1 algorithms for signing, which was the default in V2. The Spring configuration below would be placed into conf/global.xml.

### Spring wiring for globally defaulting SHA-1 signing for legacy relying-party.xml

```xml
<!-- Place in conf/global.xml -->

<bean class="org.springframework.beans.factory.config.MethodInvokingBean" depends-on="shibboleth.OpenSAMLConfig">
  <property name="targetObject">
    <bean factory-method="getGlobalSignatureSigningConfiguration" class="org.opensaml.xmlsec.SecurityConfigurationSupport" />
  </property>
  <property name="targetMethod" value="setSignatureAlgorithms" />
  <property name="arguments">
    <list>
      <list>
        <util:constant static-field="org.opensaml.xmlsec.signature.support.SignatureConstants.ALGO_ID_SIGNATURE_RSA_SHA1" />
        <util:constant static-field="org.opensaml.xmlsec.signature.support.SignatureConstants.ALGO_ID_SIGNATURE_ECDSA_SHA1" />
        <util:constant static-field="org.opensaml.xmlsec.signature.support.SignatureConstants.ALGO_ID_SIGNATURE_DSA_SHA1" />
        <util:constant static-field="org.opensaml.xmlsec.signature.support.SignatureConstants.ALGO_ID_MAC_HMAC_SHA1" />
      </list>
    </list>
  </property>
</bean>

<bean class="org.springframework.beans.factory.config.MethodInvokingBean" depends-on="shibboleth.OpenSAMLConfig">
  <property name="targetObject">
    <bean factory-method="getGlobalSignatureSigningConfiguration" class="org.opensaml.xmlsec.SecurityConfigurationSupport" />
  </property>
  <property name="targetMethod" value="setSignatureReferenceDigestMethods" />
  <property name="arguments">
    <list>
      <list>
        <util:constant static-field="org.opensaml.xmlsec.signature.support.SignatureConstants.ALGO_ID_DIGEST_SHA1" />
      </list>
    </list>
  </property>
</bean>

```

Other global security configuration changes are possible using the same general pattern. Consult the Javadocs for the relevant classes to determine what methods and arguments may be used.

### Notes

V3 defaults to the use of separate keypairs for signing and decryption, the latter being a new capability that is rarely used at present and primarily comes into play only when processing `<LogoutRequest>` messages that contain an encrypted subject ID.

The default algorithms used have been updated to:

- RSA or ECDSA (depending on key type) with SHA-256 digests
- AES-CBC-128 with RSA-OAEP-MGF1 key transport

The IdP also supports a standardized SAML metadata extension that influences the algorithms it will use, based on the capabilities of the SP. For metadata under your control, using the extension to specify that SHA-1 is required is generally a better strategy than specifying it in a RelyingParty override, but often this isn't possible. We encourage deployers to ask their metadata sources (i.e., federations) about this capability, as it is critically important for dealing with future algorithm weaknesses that may emerge over time.