ProfileHandling

Error rendering macro 'toc'

[com.ctc.wstx.exc.WstxLazyException] com.ctc.wstx.exc.WstxParsingException: Undeclared namespace prefix "xlink" (for attribute "href")

at [row,col {unknown-source}]: [7,167]

Overview

The primary "API" of the IdP is the support for various standard (or standardized) federation protocols like SAML and CAS. Most interactions with the IdP are in the form of requests using those protocols or request to a few internal administrative functions. A summary of the supported protocol interfaces can be found here.

The profile workflows are built using Spring Web Flow, so it's important to understand that technology to make sense out of most of the code. All of the "top level" flows in the system implement one of the supported protocol interfaces, which are referred to as "profiles", a term used in SAML for a unit of interoperability that supports a particular function. Example profiles would be SAML 1 attribute queries, SAML 2 SSO requests, etc.

Within the top level profile flows, various subflows are run that carry out reusable sequences of tasks like authentication. These subflows are generally described in their own sections of the top level design and are designed to be composable with any profile flows that need to use them, by defining their own conventions for inputs and outputs.

Relying Party and Profile Configuration

As described in the GeneralArchitecture topic, the root of the context tree for any of the profile flows is the **ProfileRequestContext** class. Beyond that commonality, all of the "typical" federation profiles have a common approach to their configuration and how its exposed in the context tree.

For consistency with V2, and also because it works reasonably well as a configuration model, V3 maintains the idea of a "relying party" configuration as a container for one or more "profile configurations".

A **RelyingPartyConfiguration** (RPC) is a set of configuration options that apply to a given relying party. Every RPC contains, at least:

- a unique identifier that is used mostly for logging purposes; it doesn't necessarily correspond to any actual "name" of the RP
- a predicate that determines if the RPC applies for a given request
- the identity the IdP should assume in its communication with the RP
- a collection of of profile configurations implementing the **ProfileConfiguration** interface

The profile configurations indicate whether a particular protocol/profile is enabled for use by the relying party, and any special configuration options for that profile. Profile configurations also carry a **SecurityConfiguration** that supplies relevant security settings such as algorithms, credentials, and so forth, though these are rarely used directly, but rather as input to a more complex derivation process (docs TBD).

By convention, the resolution of the appropriate RPC to use for a request is represented by attaching a **net.shibboleth.idp.profile.context.RelyingPartyContext** child context to the **ProfileRequestContext**. Both the RPC and the specific profile configuration in effect are captured by that context, along with information about the identity of the relying party and whether that identity was verified or not.

Relying Party Configuration Resolver

The IdP contains a service responsible for selecting the appropriate RPC for a given request, the **RelyingPartyConfigurationResolver**. Because it's a service, the underlying implementation is fully abstracted from the system and is pluggable.

The default implementation relies on a reloadable configuration resource and just iterates through an ordered list of registered RPC objects, evaluating the current **ProfileRequestContext** against the RPC's predicate. The first RPC with a predicate to return an affirmative result is the RPC that's used for the request. In addition, the resolver stores two special RPCs: a default "verified" one to apply in the absence of a more specific rule, and an "unverified" one to apply in the event that the identity of the relying party can't be verified, in a fashion that depends on the profile used (in the case of SAML, a RP can only be verified if metadata is provided for it).

Profile Design Documentation

The following describes the flow for each profile including the steps that make it up and what they do. Most of it is empty or out of date at this point as it's mostly internal documentation and is not required to extend the IdP in most cases.

- SAML 1.1 Browser SSO
- SAML 1 Attribute Query
- SAML 1 Artifact Resolution
- SAML 2 SP-Initiated SSO
- SAML 2 IdP-Initiated SSO
- SAML 2 Attribute Query
- SAML 2 Artifact Resolution

- Constrained Web Service SSO

Programming Guide to Extending Profiles

One of the challenges in implementing the profile behavior as webflows was in establishing a mechanism for extension that didn't require copying or editing the internal flow definition files, making upgrades a difficult proposition for anybody that had extended them. Any given change becomes rather simple and small in terms of effort, but splicing in the changes can be complex.

While not every change can be easily accomodated, a large number of use cases are enabled by providing a set of well-defined "hooks" for branching control to user-supplied subflows that can alter the state of the request, or prevent a request from completing outright. The hooking subflow is called an "interceptor", and there are three main hooks during which any number of interceptors can be configured:

- Inbound message processing
- Post-authentication / attribute lookup processing
- Outbound message processing

When feasible, the first and third hooks are provided in any given profile flow, but the middle hook only applies to more "traditional" SSO-oriented profiles that perform user authentication. Most often, it's that middle hook that is of interest to a deployer, and a number of useful example interceptors are provided, such as attribute release consent and usage terms, coarse-grained authorization, and expiring password notification.

Authoring and Enabling Intercepts

An interceptor is an arbitrary subflow that is assigned a flow ID that starts with "intercept/" and is further defined to the system with a bean of type **net.shibboleth.idp.profile.interceptor.ProfileInterceptorFlowDescriptor** in a list in *conf/intercept/profile-intercept.xml*.

While you may deliver a custom flow in a relatively "drop-in", self-contained jar, you **MAY NOT** manipulate the state of the IdP at runtime to install the necessary descriptor bean because it is impossible to guarantee that your modification will take place early enough to be seen by other objects in the system. There is no publically supported mechanism to extend any of the beans defined inside the "root" web app context, and so you **MUST** rely on the deployer making the necessary adjustments to define custom flows to the system via the associated type of FlowDescriptor.

Interceptor flows must inherit from an abstract flow named "intercept.abstract" by specifying it in the `<flow>` element's `parent` attribute. Failure to do so will prevent the system's built-in event and error handling from functioning correctly.

To enable an interceptor, the flow ID suffix (the name after the "intercept/" prefix) must be added into the appropriate list property on the relevant **ProfileConfiguration** object in *conf/relying-party.xml*. The properties contain an ordered list of flow IDs representing the flows to execute in a sequence until one fails (results in a non-"proceed" event) or all have run.

Note that interceptors may also have ActivationConditions attached that further control whether they run.

See ProfileInterceptConfiguration for more information on this "deployer-facing" aspect of the system.

General Event Behavior

The system as a whole follows a fairly standard pattern: flows signal the "proceed" event to allow execution to continue, and any other event will propagate into the error handling machinery of a profile (this can have different effects depending on the profile, e.g. a SOAP fault, an error page, a SAML response, etc.).

It's possible to create your own events that can be handled in custom fashion, but doing so requires that the system be taught about the event by adding an appropriate `<end-state>` element to *conf/intercept/intercept-events-flow.xml*.

Inbound Interceptor Contract

There is no specific input contract, since profiles vary in their function, but as a general rule, inbound interceptors can assume that the **ProfileRequestContext**'s inbound **MessageContext** tree is fully populated, and when possible a **RelyingPartyContext** child context will have been created, along with any contexts needed to drive functions like transport or message authentication.

Similarly, there is no specific output contract. Typically inbound interceptors will examine the tree to determine whether to continue with processing or not, but may also be used to add information to the tree. Developers are cautioned that mutating the state of the tree is likely to directly influence subsequent processing. That is, any information already in the tree is likely to be assumed to remain valid if it's changed by the interceptor, which provides wide latitude to supersede the information placed there originally.

The specific content of the **MessageContext** tree will depend on the profile involved (documentation on which is very limited at present). Inbound interceptors are generally configured by the system and serve to implement various security policy checks against the inbound message context. Often, they are made up of one or more **MessageHandler** objects, which are similar to ProfileActions but are more constrained in purpose.

Since the system comes pre-configured with inbound interceptors attached that perform important security functions, adding a new interceptor generally requires that the developer examine the pre-defined set of interceptors and populate a new list of intercepts that includes existing ones plus the new one, to avoid creating a security hole. Be very careful in using this hook, as it's very easy to compromise the system by doing so indiscriminately.

Post-Authentication Intercept Contract

The post-authentication hook is both more limited than the others (in that it is supported only for a subset of profiles) and more powerful (in that it runs in the middle of processing, allowing manipulation of the context tree before outgoing message content is actually constructed by the system. Changes ranging from manipulation of the authenticated identity to addition or removal of attributes can be effected.

As with all interceptors, the exact input state of the context tree is profile-dependent, but generally speaking you will find some important child contexts in the following arrangement:

Graphviz output could not be displayed here.

On the output side, interceptors can, by design, mutate these contexts to affect profile processing. If you plan to signal an event to abort processing or cause an error response, and would like to prevent the remotest possibility that a message containing any user data might be issued, removing the SubjectContext and AttributeContext objects from the tree above is an effective, though unnecessary, insurance policy.

Outbound Intercept Contract

The inverse of the inbound case, the outbound interceptors run just before the final stage of message processing that will issue a response to the client. As such, the input contract is essentially "the entire state of the tree produced by the requested profile execution", whatever that happens to be. That will typically include a populated outbound **MessageContext** tree, although the message may not be in its absolutely final form (e.g., signing may be yet to happen).

As with inbound interceptors, information may be added or changed in the tree, and the eventual output message can be dliberately changed, or processing aborted.