AuthenticationFlowSelection

The authentication "engine" includes support for a range of use cases from the very simple to the very complex, along with a lot of extension points for handling even more complex cases that it doesn't support out of the box.

The core of this engine is a set of steps that evaluate a request for authentication and local policy to winnow a set of possible login flows down to a smaller set that it determines are suitable. This process might be extremely simple or extremely not-simple, depending on configuration and on the number of login methods in use. In the vast majority of cases, only a single method may be in play, but for those with more complex requirements, the process is described below, along with the options available to influence the process without writing code. Of course, understanding the process is important for those writing code as well.

Inputs

At a very high level, the following runtime information is captured as input to the process:

- any active session for the client along with any still-valid previous AuthenticationResult objects for reuse
- the AuthenticationFlowDescriptor objects for any login flows enabled for use via the **idp.authn.flows** property and the `authenticationFlows` property attached to the profile configuration in effect; any flows not included in both sets are considered unavailable
- the request to the IdP (the specifics of which depend on the protocol in use)
- the `defaultAuthenticationMethods` property attached to the profile configuration in effect

Internally, the AuthenticationFlowDescriptor objects are maintained in the order they're declared in the list of beans in *authn/general-authn.xml*. This means that absent other influences, the general order flows will be tried can be controlled with that list.

With respect to the request, the chief piece of information extracted is anything the request specifies to limit or control the login method used. At present, this is only supported for SAML 2.0 requests (the `<RequestedAuthnContext>` element), but the information is extracted in a portable form ahead of this step, so any protocol with this feature can be supported equally and uniformly.

In the event that nothing is specified in the request, the `defaultAuthenticationMethods` property is essentially a pseudo-requirement that will be imposed automatically, as if the request specified that an exact match to one of the specified custom Principals is required. In SAML terms, a profile configuration such as this:

```
<bean parent="SAML2.SSO">
        <property name="defaultAuthenticationMethods">
                <list>
                        <bean parent="shibboleth.SAML2AuthnContextClassRef"
                                c:classRef="urn:oasis:names:tc:SAML:2.0:ac:classes:TimeSyncToken" />
                </list>
        </property>
</bean>
```

is equivalent to an SP for which the configuration applies requesting this:

```
<samlp:RequestedAuthnContext operator="exact">
        <saml:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:TimeSyncToken</saml:
AuthnContextClassRef>
</samlp:RequestedAuthnContext>
```

Thus, the property is proscriptive, not advisory.

Flow Filtering

The simplest steps performed during this pipeline are a straightforward filtering of the possible login flows based on high-level requirements. This fiiltering includes:

- eliminating flows that don't support "passive" use if the request requires this (corresponding to the IsPassive flag in SAML 2)
- eliminating flows that don't support "forced authentication" if the request requires this (corresponding to the ForceAuthn flag in SAML 2)
- eliminating flows that require a browser client if the request is from a non-browser client (e.g., the ECP profile in SAML 2)
- eliminating flows that aren't supported are permitted for the active subject (see below)

The last is a feature that allows an attribute to be configured for resolution that contains values corresponding to the login flows to permit the use of. This only happens in the case that an active session already exists with the client so that the subject of the session can be used as input to the attribute resolution process. A common use for this feature is to track users that have multi-factor authentication tokens or certificates and so are able to use stronger login methods than the population as a whole.

Flow Selection

The heart of the process is a step that picks either a pre-existing AuthenticationResult or a login subflow to attempt to fulfull the request. This is where a lot of complex logic is embedded, and in principle could be replaced with an entirely different algorithm without impacting the rest of the system.

One of the things this step does is prevent the same methods from being attempted multiple times in a loop by tracking the ones that have been tried already. This can be cirumvented in exceptional cases, but when it's allowed to just "do its thing", the selection bean will eventually try each flow in arbitrary order until one succeeds or all have failed.

Apart from that, one of three cases applies:

- a login subflow explicitly asked that a particular login subflow be tried (inter-flow signaling, mentioned under AuthenticationConfiguration, Advanced Features)
- the request contained no requirements as to which subflow to use
- the request contained specific requirements as to which subflow to use

In the first case, a specific flow is being selected, so in essence it just "promotes" a flow into first position for the other two cases to handle. In other words, if the request doesn't dictate any constraints, the signaled flow is attempted, whereas if the request does contain constraints, those constraints have to be met by the signaled flow or it won't be tried.

The other two general cases are described below.

Handling Requests With No Requirements

When the request to the IdP doesn't explicitly have method requirements, the selection process proceeds as follows:

1. If the request doesn't call for forced authentication, then one of the active AuthenticationResults from a session will be reused if the subflow that produced it is enabled for the request (SSO).
2. Otherwise, one of the enabled and unattempted subflows is chosen and attempted, based on the order of declaration in *authn/general-authn.xml*.
3. If no unattempted flows remain, authentication fails.

This is the basic/simple case and requires little in the way of special care.

Handling Requests With Requirements

When the request to the IdP does have explicit method requirements, or a `defaultAuthenticationMethods` property is set on the applicable profile configuration, then the selection process is as follows:

1. If the request calls for forced authentication, or if there are no active AuthenticationResults from a session, go to step 3.
2. If the **idp.authn.favorSSO** property is true, then the collection of active AuthenticationResults is searched for a result that matches the request's requirements (SSO).
3. The request's requirements are examined in order so that its precedence rules are applied, and the collection of enabled and unattempted flows is searched (in the order they're declared) for a match to the requirement being examined. Assuming a match is found, one of the following applies:
    a. The collection of active AuthenticationResults from a session is searched for a result from that flow, and the result is further checked to determine if it matches the request requirement. If so, it is reused (SSO).
    b. The matching flow is chosen.
4. If no unattempted flows that match any of the requirements remain, authentication fails.

A subtle point: login flows may claim to support a variety of requirements but an AuthenticationResult is only reused if that result itself supports a particular requirement. As a concrete example, a single flow might handle both password and multi-factor authentication but generate results specific to one type or the other. Alternatively, separate flows might be used for the two. The selection process handles either case.

Comparison Configuration

The *authn/authn-comparison.xml* file is an advanced configuration file used to support the matching process described above. In V2, only "exact" matching was supported, and each login handler typically supported only a single hardwired "method" value for comparison purposes. In V3, this has been generalized to support "inexact" matching. This file defines the relationships between authentication types in order to support this inexact matching.

The support for this is not limited to SAML in the design, but since SAML is the only implemented case, there are classes and beans defined to manage the relationships and support the operators specific to that case.

Exact matching is automatic; this applies when a SAML 2 request specifies "exact" matching in an `<RequestedAuthnContext>` element, or if nothing is requested but the profile configuration includes the `defaultAuthenticationMethod` property.

To make inexact matching work, you have to define the matching rules. These rules apply only to SAML 2 requests at the moment. While you can define matching rules for AuthnContextDeclRef constants, these are rarely used in practice, and you will generally only need to define rules for AuthnContextClassRef constants (if anything). There are beans predefined and registered for each of the inexact matching types possible:

- **shibboleth.BetterClassRefMatchFactory**
- **shibboleth.MinimumClassRefMatchFactory**

- **shibboleth.MaximumClassRefMatchFactory**

The latter two, if not modified, still allow degenerate support by treating the request as equivalent to "exact" (because they're inclusive of the value supplied in the request), but "better" matching won't succeed without comparison rules added since it's non-inclusive.

The file includes an example of how to define such rules. Each bean contains a property to set with a map of values that might be requested and a corresponding list of values that should satisfy the request. Those values are attached to AuthenticationFlowDescriptor beans in *authn/general-authn.xml* via the `supportedPrincipals` property.

(V3.2+ only) Lastly, a bean called **shibboleth.IgnoredContexts** can be defined to identify specific AuthnContextClassRef or AuthnContextDeclRef values to ignore if found in a SAML 2 `<RequestedAuthnContext>` element. By default this consists of a single value, `urn:oasis:names:tc:SAML:2.0:ac:classes:unspecified`, which was ignored in V2.