PasswordAuthnConfiguration

Current File(s): *conf/authn/password-authn-config.xml, views/login.vm*

Format: Native Spring

Legacy V2 File(s): *conf/handler.xml, conf/login.config, login.jsp*

Overview

The authn/Password login flow supports an extensible set of back-ends for password-based authentication, normally collected using a web form, and is the flow used by most deployments.

It is compatible with non-browser clients by virtue of supporting HTTP Basic authentication if credentials are provided without prompting, and knows not to present a form when a non-browser profile like ECP is used.

General Configuration

Use *authn/password-authn-config.xml* to configure this flow.

The most critical part is at the top; you must choose a back-end implementation to use by importing *one and only one* Spring bean file that contains the back-end-specific configuration needed. The default is native LDAP; JAAS and Kerberos are also implemented. You can also implement your own and import a bean file describing it as long as you comment out or remove the others.

For detailed information on configuring the supplied back-ends, see:

- LDAPAuthnConfiguration
- JAASAuthnConfiguration
- KerberosAuthnConfiguration

Aside from the back-end configuration, beans are provided for some general configuration that is independent of the back-end chosen. They are described in the reference section below.

User Interface

The first user interface layer of the flow is actually HTTP Basic authentication; if a header with credentials is supplied, the credentials are tested immediately with no prompting. If that step fails, or no credentials are supplied, then a view is rendered unless the request is for passive authentication or part of a non-browser profile.

Views are handled by Spring Web Flow, so support various technologies, but Velocity and JSP are supported by default.

Be aware that the default view template and configuration of classified messages (see next section) results in a system that will report context-sensitive errors to the user, such as whether the password was invalid or the username was mis-entered. This is appropriate for most organizations to reduce help desk calls caused by simple user error, but some organizations keep usernames a secret and may wish to adjust the configuration to collapse all error reporting and avoid leaking information about whether usernames are valid or not.

Example Velocity User Interface

The example Velocity templates *views/login.vm* and *views/login-error.vm* illustrate generally how to populate the form, and how to detect and respond to error conditions. Internationalization is performed through the use of Spring message properties (see the files in the **messages** folder). Information on Spring internationalization is near the end of this section of the Spring documentation.

When rendering Velocity views, several properties are available to aid per-relying party customization.

Spring form generation macros such as *#springMessage and #springMessageText* are available to Velocity templates.

You can freely comment out or remove the "Do Not Cache" support of course, or use Javascript to automate it for certain address ranges.

JSP User Interface

To use JSP, *views/login.vm* must be removed or renamed and the IdP restarted, or it will take precedence. Views in JSP should be created in **edit-webapp /WEB-INF/jsp** (and the warfile should be recreated with *bin/build.sh* or *bin/build.bat* and the container restarted). The V2 Taglibs are supported for jsp.

Errors and Warnings

A special feature of this flow is the ability to inject your own behavior in response to particular error or warning conditions that occur. These conditions are set via the **shibboleth.authn.Password.ClassifiedMessageMap** bean in *authn/password-authn-config.xml*. That is, if you map a particular string label to one or more exception/error messages in the map, that string label becomes a flow "event" that you can program the flow to respond to.

In the simplest case, it's possible to map messages to labels like "InvalidPassword" or "AccountLocked", and the *views/login-error.vm* file includes example behavior that handles these kinds of error labels. That's one form of customization.

If your goal is to terminate processing after an error, then you should take a look at the discussion in AuthenticationConfiguration under "Custom Events", which describes how to authorize the system to treat your custom events as allowable "end" states for the request and handle them as errors.

If your goal is rather to interrupt but then resume the login flow, then this is somewhat accomodated with a predefined set of flow defintions designed to be user-editable in **flows/authn/conditions/**

The *conditions-flows.xml* file controls what events are detected and handled, and what to do. The example included responds to a number of events by calling predefined subflows that are themselves just empty examples that return immediately, and then control passes back to the view state that renders the login form.

An obvious use for this feature is responding to an imminently expiring password with some kind of warning view; another is detection of a locked account, and the use of a dedicated view to help the user with that condition.

In the simplest case, maybe you just want to display a page, in which case you can insert your own `<view-state>` that displays a template of your own creation. Views generally have to contain a form whose action is set to "$flowExecutionUrl" and that contain a button to submit the form with the value "_eventId_proceed". Your view then transitions on "proceed" to the corresponding `<end-state>` and ends up back on the login form.

In more complex cases, you may need to actually redirect the user out to other functionality implemented outside the IdP, which is something the Spring Web Flow documentation refers to as an external redirect, and it's possible to resume the flow in that scenario as well. If you don't, the user's session is suspeneded and will consume resources until it times out, and that doesn't work well at scale, so you have to be careful with that approach.

Custom Back-Ends

Due to the design of this login flow, it's possible to plug in alternative validation strategies by commenting out all of the existing bean imports in *password-authn-config.xml* and supplying an alternative. The only real requirement is that you supply a bean "alias" of this form:

```
<alias name="ValidateUsernamePasswordAgainstXXXX" alias="ValidateUsernamePassword"/>
```

The `name` portion is arbitrary and refers to a Java bean that you must supply. The `alias` portion is fixed and allows your bean to be used during webflow execution as the validation step once the username/password are obtained.

You will need to learn at least a bit about Spring, Spring WebFlow, and the IdP's action design to do this properly, but writing a single action is much less work than defining an entire custom flow yourself. In many cases, you can avoid a lot of the work by inheriting your bean from net.shibboleth.idp.authn.AbstractUsernamePasswordValidationAction, a public API class that implements a lot of the housekeeping required. You can also refer to the existing beans used in the currently supplied back-ends to get additional insight.

In principle, you can implement a lot of advanced behavior this way. If you wanted to support, for example, multiple back-ends at the same time, you could embed that sequencing or selection logic in this action to decide what to do. It's Java code, so in general you can do anything you want to do.

Calling Extended Flows $^{3.2}$

A feature included with the Password flow is the ability to present additional login options to the user and call them directly as subflows in response to some form of user input or client-side script. This is useful if you want most users to have the option to use a password, but make more advanced options available at the same time without requiring extra pages or clicks to reach the password option.

This feature has been superseded by the MFA login flow implemented in V3.3 and is much more flexible and usually simpler to use, even though it's more complex to understand at first. There are no explicit plans to deprecate or remove this feature yet, but migrating away from it would be a good decision.

Basic Setup

Some preconditions for using this feature:

- The "extended" flows you want to make available must be active. That is, they must be enabled via the **idp.authn.flows** property and/or any profile-specific configuration you create.
- The Password flow must be listed earlier in the **shibboleth.AvailableAuthenticationFlows** bean in *authn/general-authn.xml* than the extended flows (or at least, you must be cognizant that any flows listed earlier may be run earlier).
- The `supportedPrincipals` property of the Password flow must typically be a union of its own list of custom Principals and any supported by the extended flows. Otherwise the IdP may not invoke the Password flow if a request is made that requires one of the extended flows to be used.

Assuming the above changes are made, to configure this feature you must supply definitions for the following beans in *authn/password-authn.xml* (there are example definitions at the bottom of the file in a comment):

- **shibboleth.authn.Password.ExtendedFlows**
    - Similar to the **idp.authn.flows** property, this is a String containing a regular expression matching the names of the candidate extended login flows to offer.
- **shibboleth.authn.Password.PrincipalOverride**
    - This list bean overrides the values supplied in the Password flow's `supportedPrincipals` property and reduces the list back down to a smaller set, just the custom Principals to include in any results produced by the Password mechanism itself. In other words, if the Password flow has to be enabled for stronger authentication types, this is where you reduce the list back down to just the values associated with password authentication. Typically you would set this to the same values the Password flow's `supportedPrincipals` property would have originally been configured with.

User Interface

The above changes trigger the example logic in the login view template (*views/login.vm*) that iterates over the extended flows, determines which ones are "allowable" for the request, and displays a button to run them. This is obviously a crude UI that you are expected to tailor for your use case, usually by taking advantage of knowledge of the specific methods you're trying to make available and how you want to describe them to users. The UI also includes support for **not** rendering the Password login form if that method itself isn't allowed.

The example buttons will run the corresponding extended flow. If the flow succeeds, the authentication process will complete in the usual manner. If not, the login view will be re-displayed, and the "event" triggered will reflect whatever was returned by the failed flow. In addition, the AuthenticationFlowDescriptor returned by `$authenticationContext.getAttemptedFlow()` will return the flow that failed, rather than the Password flow. With this information, you can customize the login-error.vm template to respond to the error appropriately.

For obvious reasons, it's hard to document exactly how to do all this "correctly" because the UI will be very site-dependent (and quite often will be complex to display). The goal is to supply the template with sufficient information to allow you to script the UI without having to make coding changes.

Advanced Features

Some login flows may be implemented to take advantage of the use of this feature in a couple of different ways. A custom login flow can distinguish between being called directly by the IdP and as an extended flow called by another login flow by the presence of a flow variable called "calledAsExtendedFlow". This is automated for the External login flow by making a request attribute available called "extended" when this flow variable is set.

It's also possible to pass information across the flow boundary. Any form parameters you include in the form that submits the event to call the extended flow can be captured by adding their names to a bean called **shibboleth.authn.Password.ExtendedFlowParameters**. Any values for parameters that match a name in that bean will be saved to a map returned by the AuthenticationContext's getAuthenticationStateMap() method. They are **not** typically preserved on the actual query string that a called flow will see.

Account Lockout $^{3.3}$

A pluggable implementation of account lockout can be enabled by defining a bean called **shibboleth.authn.Password.AccountLockoutManager** that implements the AccountLockoutManager interface. A default implementation of this interface is available using a parent bean named **shibboleth.StorageBackedAccountLockoutManager** (commented out by default in *conf/authn/password-authn-config.xml*).

The default implementation has the following behavior:

- Tracks lockout state via an injected StorageService, by default the in-memory variant preconfigured with the IdP. This results in per-node lockout tracking, usually sufficient considering the overhead of other options.
- Provides a simple algorithm that tracks invalid attempts that occur within a specified interval, and once a limit is reached, blocks subsequent attempts for a specified duration.
- Scopes lockouts to a given username and client IP address (but this is configurable via an injectable function).

Note that **each** attempt can be separated in time by the specified interval, so a 5 minute interval does not mean all the attempts must occur within 5 minutes, but could occur over a period as long as 5 times the number of attempts.

Once an account locks, an "AccountLocked" event is signaled, and the new default configuration maps this event to the "AccountLocked" classified error. If your installation pre-dates this feature, you will likely want to map that event to whichever classified error reporting condition you want to expose to the user.

Managing Lockout

V3.4 adds an official mechanism for querying the state of an account, incrementing the lockout count (which could be used to lockout an account administratively), and clearing a lockout. This is exposed via a simple REST API via an administrative flow located at the path **/idp/profile/admin/lockout** which is restricted to localhost by default. Like all administrative features, you have the ability to customize authentication and access control.

To the base path you must append a slash, the name of a bean implementing the AccountLockoutManager interface, another slash, and finally the "key" that identifies the lockout record to access. By default this will be a username, a bang (!), and an IP address (unless you customize the way the lockout is scoped).

The default location of the lockout manager bean that is commented out by default is inside the *password-authn-config.xml* file. If you need to make use of this management API, move that bean (or copy it if you prefer) into *global.xml* so that it will be accessible to this feature.

Three HTTP methods are supported:

1. GET – query an account to see if it's locked or not
2. POST – increment an account's lockout counter artificially
3. DELETE – clear an account's lockout state

The POST/DELETE operations return a 204 on success, while the GET operation returns a JSON response describing the object queried and the lockout status. An example trace follows (much of the response header dump is elided, this just shows the basics).

Example lockout operations

```
$ curl -ik "https://localhost/idp/profile/admin/lockout/shibboleth.authn.Password.AccountLockoutManager/jdoe%
21192.168.1.1"

HTTP/1.1 200 OK
Content-Type: application/json;charset=utf-8

{
  "data" : {
    "type" : "lockout-statuses",
    "id" : "shibboleth.authn.Password.AccountLockoutManager/jdoe!192.168.1.1",
    "attributes" : {
      "lockout" : true
    }
  }
}

$ curl -X DELETE -ik "https://localhost/idp/profile/admin/lockout/shibboleth.authn.Password.
AccountLockoutManager/jdoe%21192.168.1.1"
HTTP/1.1 204 No Content
```

Reference

Beans

The beans defined in *authn/password-authn-config.xml* follow:

Bean ID	Type	Default	Function
shibboleth.authn.Password.UsernameFieldName	String	j_username	Name of the username field in the form
shibboleth.authn.Password.PasswordFieldName	String	j_password	Name of the password field in the form
shibboleth.authn.Password.SSOBypassFieldName	String	donotcache	Name of the form field signaling to avoid caching the authentication result for SSO, defaulting to donotcache
shibboleth.authn.Password.Lowercase	Boolean	false	Whether to lowercase the username before validating it
shibboleth.authn.Password.Uppercase	Boolean	false	Whether to uppercase the username before validating it
shibboleth.authn.Password.Trim	Boolean	true	Whether to trim leading and trailing whitespace from the username before validating it
shibboleth.authn.Password.Transforms	Pair<String,String>		Pairs of regular expressions and replacement expressions to apply to the username before validating it
shibboleth.authn.Password.matchExpression 3.3	Pattern		Regular expression to check username against
shibboleth.authn.Password.RetainAsPrivateCredential 3.2	Boolean	false	Whether to keep the password around as a private "credential" in the Java Subject for use in later stages such as attribute resolution
shibboleth.authn.Password.	Boolean	true	Whether to remove the object holding the password from the request's active state after

RemoveAfterValidation $^{3.3}$			validating it (to avoid it being preserved in the session any longer than needed)
shibboleth. StorageBackedAccountLockoutManager $^{3.3}$	StorageBackedAccountLockoutManager		Parent bean for declaring storage-backed account lockout with appropriate defaults
shibboleth.authn.Password. AccountLockoutManager $^{3.3}$	AccountLockoutManager		A lockout manager that, if defined, will enable account lockout feature
shibboleth.authn.Password. ClassifiedMessageMap	Map<String, List<String>>	various	A map between defined error/warning conditions and events and implementation-specific message fragments to map to them.
shibboleth.authn.Password. resultCachingPredicate	Predicate<ProfileRequestContext>		An optional bean that can be defined to control whether to preserve the authentication result in an IdP session
The beans below are primarily used in conjunction with the Extended Flow Calling feature added in V3.2.0:			
shibboleth.authn.Password. ExtendedFlows $^{3.2}$	String		Regular expression identifying the login flows to present as additional options to a user in the view
shibboleth.authn.Password. ExtendedFlowParameters $^{3.2}$	List<String>		A list of form parameters to preserve for access by extended flows
shibboleth.authn.Password. PrincipalOverride $^{3.2}$	List<Principal>		Custom principals to include in the results produced by the Password flow, if different from the set attached to the flow's `supportedPrincipals` property
shibboleth.authn.Password. addDefaultPrincipals $^{3.2}$	Boolean	true	Whether to add the content of the `supportedPrincipals` property of the underlying flow descriptor to the resulting Subject

V2 Compatibility

This flow is designed to be integration-compatible with the V2 UsernamePassword handler when it comes to the JAAS configuration used (if in fact the JAAS back-end is used), but is not fully compatible with the old *login.jsp* template, though there are broad similarities, obviously. The form action is set differently, and the interface to error information when a failed login occurs is different. In return, more detailed error state is available through the use of a message mapping facility that can be taught how to classify arbitrary errors from back-end systems and libraries.

Though it is possible to use a JSP view as a login form, the default view template uses Velocity (and as a consequence can be changed at any time without unpacking or repacking the warfile).

Notes

The shibboleth.authn.Password.RetainAsPrivateCredential flag added to V3.2 should be used with caution, as it retains the password and makes it available in plaintext form within server memory at various stages. When the session is written to storage, the password is encrypted with the secret key used by the IdP for other encryption of data to itself, but it will be decrypted and back in memory at various times when the session is accessed or updated.