ContextCheckInterceptConfiguration

Current File(s): *conf/intercept/context-check-intercept-config.xml*

Format: Native Spring

- Overview
- General Configuration (V3.4 and above)
- General Configuration (V3.3 and below)
    - Multiple Checkers
- Examples
    - Enforcing Authentication Policy
- Reference
    - Beans
- Notes

Overview

The "context-check" interceptor flow is an example of how to use an interceptor to interrupt processing and either continue or halt processing based on the state of the "context tree" that makes up the state of the request. A common use case for this feature is to impose authorization rules at the IdP to work around the limitations of a service that either does not implement any authorization or does not provide an adequate user experience in the event of failure. A frequently cited example of the latter is Google Apps for Education. Must be their limited budget.

Everything about a request is tracked in the tree of context objects (or can be injected into the condition bean applied), so there is no "hidden" information. Anything is on the table for examination, including:

- information about the relying party
- information about the user, the user's session, the user's authentication state, or attributes
- environmental information from the client request
- any configuration/rules you define and inject with Spring

All interceptors are enabled or disabled on a per-relying-party basis using properties in the profile bean(s) you want to enable the flow for. See the ProfileInterceptConfiguration topic for an example.

General Configuration (V3.4 and above)

As of V3.4, this flow can operate in the limited "single condition" mode offered originally or a more flexible functional mode.

For the single condition mode, refer to the original documentation below for older versions.

Previously, it was common to need to create copies of the flow to accommodate different needs, as shown in the section labeled "Multiple Checkers". While this is still possible, it can be avoided in many cases by using the newer support for applying a Function instead of a condition/predicate. While the condition mode applies a single condition and returns a fixed error Event, the Function mode is more general in that it directly returns a string to use as the follow up event.

The bean named **shibboleth.context-check.Function** in *intercept/context-check-intercept-config.xml* must be defined by you with the function you want to apply. The bean must be of type Function<ProfileRequestContext,String>. The return value contains the event that the interceptor should signal, either "proceed" to indicate that processing should continue successfully or any other value as a custom Event. For example, returning "ContextCheckDenied" will match the existing behavior of the original condition mode.

If you want to support one or more custom events, you'll need to add the event(s) to *conf/intercept/intercept-events-flow.xml*. The default file includes a commented example for an event called "MyCustomEvent". Then you'll need to add that event in *conf/errors.xml* if you want it handled with a local error page.

As a primitive example, consider a map indexed by relying party name, allowing a condition to be uniquely defined for each applicable relying party separately.

Example function using a map of conditions

```
<util:map id="ConditionMap">
        <entry key="https://sp.example.org/sp">
                <ref bean="Condition1"/>
        </entry>
        <entry key="https://another.example.org/sp">
                <ref bean="Condition2"/>
        </entry>
</util:map>

<bean id="shibboleth.context-check.Function" parent="shibboleth.ContextFunctions.Scripted" factory-method="
inlineScript"
                p:customObject-ref="ConditionMap">
        <constructor-arg>
                <value>
                <![CDATA[
                var event = "proceed";
                var rpid = input.getSubcontext(
                                "net.shibboleth.idp.profile.context.RelyingPartyContext").getRelyingPartyId();
                var condition = custom.get(rpid);
                if (condition != null && !condition.apply(input)) {
                        event = "ContextCheckDenied";
                }
                event;
                ]]>
                </value>
        </constructor-arg>
</bean>
```

General Configuration (V3.3 and below)

The only configuration involved with this flow is to define the condition you want it to evaluate, and possibly adjust the user interface result in the event of failure.

The bean named **shibboleth.context-check.Condition** in *intercept/context-check-intercept-config.xml* must be defined by you with the condition you want to apply. The bean must be of type Predicate<ProfileRequestContext>, but beyond that, it can do anything. Note that this is the same type signature as the conditions discussed in the ActivationConditions topic, so the examples there may help you. Non-programmers may be particularly interested in the scripted examples.

The common authorization usage for this flow is reflected in the example condition you will find in the file. It demonstrates the use of a built-in condition called a SimpleAttributePredicate, which evaluates the request for the presence of particular attribute(s) and optionally value(s) through a simple map. Each map entry is the ID of an attribute, and the map values are a list of values of that attribute to check for (or you can use an asterisk as a wildcard to indicate that any value is acceptable). Refer to the Javadoc for additional details.

The other half of the configuration of this flow is the result. Obviously success simply causes things to proceed in the usual fashion, but failure will produce an event called "ContextCheckDenied". The IdP's built-in ErrorHandlingConfiguration treats that event as a "local error", meaning that processing is halted and an error page displayed.

The event is mapped to a default error message using the standard machinery, which you can adjust, but producing the right response for a lot of different (possibly unrelated) scenarios will quickly become hard to manage. You will probably want to define your own events, and to do that, you need to create your own copy of this flow.

Multiple Checkers

While you absolutely can use this flow directly, it will often become unwieldy to try and combine every possible use for this feature into a single condition to evaluate. One reason is the user interface problem discussed above. So you may find it more fruitful to actually copy it into your own version in "user-space" and create multiple versions of it for different purposes. That way each one is simple, self-contained, and easier to maintain.

To create a new version of this flow called "intercept/mycheck", do the following (commands shown are basic Linux/Unix, idp.home is your installation directory):

Creating the new flow from the original

```
$ cd idp.home
$ mkdir -p flows/intercept/mycheck
$ cp system/flows/intercept/context-check-flow.xml flows/intercept/mycheck/mycheck-flow.xml
$ cp system/flows/intercept/context-check-beans.xml flows/intercept/mycheck/mycheck-beans.xml
```

Next, edit the new *mycheck-flow.xml* file and change the location of the `<bean-import>` to "mycheck-beans.xml".

Then edit the new *mycheck-beans.xml* file and remove the `<import>` and `<alias>` elements (the last two) and replace them with your own condition bean. The condition bean must be called "ContextCheckPredicate", but is otherwise the same as what you would define the **shibboleth.context-check.Condition** bean to be if you were using the built-in example.

Finally edit conf/intercept/profile-intercept.xml and add the bean `<bean id="intercept/mycheck" parent="shibboleth.InterceptFlow"/>` to the existing list of beans to make the system aware of your new intercept flow.

That's all you have to do, but you may want to change the event returned by your copy so that you can map it to a different error message. To change the event, edit your new *mycheck-flow.xml* file and replace the reference to "ContextCheckDenied" with the name of your event. It can be anything, more or less. Then you'll need to add that event to a file named *conf/intercept/intercept-events-flow.xml*. The default file includes a commented example for an event called "MyCustomEvent". Then you'll need to add that event in *conf/errors.xml* to the action needed.

You can create as many copies like this as you want. Each one will have its own name, and you must individually enable these flows using the `postAuthenticationFlows` profile configuration bean property as described in the ProfileInterceptConfiguration topic.

As an example, you might create a copy of this flow for use with Google and enable it in a relying party override for that SP like so:

Example enabling an intercept for an SP in relying-party.xml

```
<util:list id="shibboleth.RelyingPartyOverrides">
        <bean parent="RelyingPartyByName" c:relyingPartyIds="google.com">
                <property name="profileConfigurations">
                        <list>
                                <bean parent="SAML2.SSO" p:encryptAssertions="false" p:postAuthenticationFlows="
googlecheck" />
                        </list>
                </property>
        </bean>
</util:list>
```

Examples

See below for additional example scenarios for this feature.

Enforcing Authentication Policy

In most cases, it is strongly advisable that the authentication "strength" required for a given transaction be managed by expressing requirements through requested principal types associated with the request, either via a SAML 2.0 `<RequestedAuthnContext>` element, or when necessary via the `defaultAuthenticationMethods` Profile Configuration property. When properly configured, the IdP will automatically prevent an inadequate authentication result from being used for a request.

In some cases, this may not be sufficient, such as when the user's identity or associated attributes dictate whether authentication is sufficient, regardless of the service. If the MFA login flow is being used, the suggested approach is to manipulate the RequestedPrincipalContext object in the context tree as part of the MFA ruleset, so that the IdP will enforce the appropriate policy for you.

If the MFA login flow is not being used, another possible technique is to perform this enforcement check yourself by means of this interceptor flow. As a simple example, consider a case where the presence of an attribute is a signal to require a particular principal be present in the result. The example demonstrates use of an InCommon-defined MFA profile in which the values represent "use of MFA" or "non-use of MFA".

Example enforcing use of MFA based on user policy

```
        <!--
        Returns true if a user's directory entity authorizes use of the "basic" profile or
        if the active results include the "mfa" profile constant.
        -->
    <bean id="shibboleth.context-check.Condition" parent="shibboleth.Conditions.OR">
        <constructor-arg>
            <list>
                <bean class="net.shibboleth.idp.profile.logic.SimpleAttributePredicate"
                        p:useUnfilteredAttributes="true">
                    <property name="attributeValueMap">
                        <map>
                            <entry key="eduPersonAssurance">
                                <list>
                                    <value>http://id.incommon.org/assurance/basic</value>
                                </list>
                            </entry>
                        </map>
                    </property>
                </bean>
                <ref bean="CheckForMFA" />
            </list>
        </constructor-arg>
    </bean>

        <!-- Checks all the active authentication results for the appropriate AuthnContextClassRefPrincipal. -->
        <bean id="CheckForMFA" parent="shibboleth.Conditions.Scripted" factory-method="inlineScript">
                <constructor-arg>
                        <value>
<![CDATA[
                        value = false;

                        principalType = Java.type("net.shibboleth.idp.saml.authn.principal.
AuthnContextClassRefPrincipal");

                        subjectCtx = profileContext.getSubcontext("net.shibboleth.idp.authn.context.
SubjectContext");
                        if (subjectCtx != null) {
                                var subjectIter = subjectCtx.getSubjects().iterator();
                                while (!value && subjectIter.hasNext()) {
                                        var princIter = subjectIter.next().getPrincipals(principalType.class).
iterator();
                                        while (!value && princIter.hasNext()) {
                                                if (princIter.next().getName() == "http://id.incommon.org
/assurance/mfa") {
                                                        value = true;
                                                }
                                        }
                                }
                        }

                        value;
]]>
                        </value>
                </constructor-arg>
        </bean>
```

Reference

Beans

Bean ID	Type	Function
shibboleth.context-check.Condition	Predicate<ProfileRequestContext>	Condition evaluated by the interceptor flow to decide whether to continue

shibboleth.context-check.Function [3.4]	Function<ProfileRequestContext, String>	Function evaluated by the interceptor flow to produce the event to signal

Notes

TBD