Implementing Security Check with Attempts

Overview

This abstract class extends `SecurityCheckWithExternalization` and implements most of its methods to simplify usage. The only two methods required to implement are `validateCredentials` and `createChallenge`. This class is good for simple flows that just need to validate some arbitrary credentials to grant access.

This class also provides built-in capabilities to block access after a set number of attempts.

This tutorial uses the example of a hard-coded PIN code to protect resources, and gives the client 3 attempts (after which the client is blocked for 60 seconds).

PinCodeAttempts

In a Java adapter, add a Java class named `PinCodeAttempts` that extends `SecurityCheckWithAttempts`.

```
public class PinCodeAttempts extends SecurityCheckWithAttempts {

    @Override
    protected boolean validateCredentials(Map<String, Object> credentials) {
        return false;
    }

    @Override
    protected Map<String, Object> createChallenge() {
        return null;
    }
}
```

Creating the challenge

When the SecurityCheck is triggered, it sends a challenge to the client. Returning `null` will creating an empty challenge which may be enough in some cases. Optionally, you can some data with the challenge, such as an error message to display, or any other data that can be used by the client.

For example, `PinCodeAttempts` sends a predefined error message and the number of remaining attempts.

```
@Override
protected Map<String, Object> createChallenge() {
    HashMap challenge = new HashMap();
    challenge.put("errorMsg",errorMsg);
    challenge.put("remainingAttempts",remainingAttempts);
    return challenge;
}
```

`remainingAttempts` is inherited from `SecurityCheckWithAttempts`.

Validating the credentials

When the client sends the challenge's answer, the answer is passed to `validateCredentials` as a `Map`. This method should implement your logic and return `true` if the credentials are valid.

```
@Override
protected boolean validateCredentials(Map<String, Object> credentials) {
    if(credentials!=null && credentials.containsKey("pin")){
        String pinCode = credentials.get("pin").toString();

        if(pinCode.equals("1234")){
            return true;
        }
        else {
            errorMsg = "Pin code is not valid.";
        }

    }
    else{
        errorMsg = "Pin code was not provided";
    }

    //In any other case, credentials are not valid
    return false;

}
```

Configuration class

Instead of hardcoding the valid PIN code, let's allow it to be configured in the adapter.xml and the MobileFirst Console.

Create a new Java class that extends `SecurityCheckWithAttemptsConfig`. It is important to extend a class that matches the parent SecurityCheck in order to inherit the default configuration.

```
public class PinCodeConfig extends SecurityCheckWithAttemptsConfig {

    public String pinCode;

    public PinCodeConfig(Properties properties) {
        super(properties);
        pinCode = getStringProperty("pinCode", properties, "1234");
    }

}
```

The only required method in this class is a constructor that can handle a `Properties` instance. Use the `get[Type]Property` methods to retrieve a specific property from the adapter.xml. If no value is found, the third parameter defines a default value (`1234`).

You can also add error handling in this constructor, using the `addMessage` method:

```
//Check that the PIN code is at least 4 characters long. Triggers an error.
if(pinCode.length() < 4){
   addMessage(errors,"pinCode","pinCode needs to be at least 4 characters");
}

//Check that the PIN code is numeric. Triggers warning.
try
{ int i = Integer.parseInt(pinCode); }
catch(NumberFormatException nfe)
{ addMessage(warnings,"pinCode","PIN code contains non-numeric characters"); }
```

In your main class (`PinCodeAttempts`), add the following two methods to be able to load the configuration:

```
@Override
public SecurityCheckConfiguration createConfiguration(Properties properties) {
   return new PinCodeConfig(properties);
}
@Override
protected PinCodeConfig getConfig() {
   return (PinCodeConfig) super.getConfig();
}
```

Now, you can use `getConfig().pinCode` to retrieve the default PIN code.

`validateCredentials` can be modified to use the PIN code from the configuration instead of the hardcoded value.

```
@Override
protected boolean validateCredentials(Map<String, Object> credentials) {
   if(credentials!=null && credentials.containsKey(PINCODE_FIELD)){
      String pinCode = credentials.get(PINCODE_FIELD).toString();

      if(pinCode.equals(getConfig().pinCode)){
         return true;
      }
      else {
         errorMsg = "Pin code is not valid. Hint: " + getConfig().pinCode;
      }

   }
   else{
      errorMsg = "Pin code was not provided";
   }

   //In any other case, credentials are not valid
   return false;

}
```

Configuring the SecurityCheck

In your adapter.xml, add a `<securityCheckDefinition>` element:

```
<securityCheckDefinition name="PinCodeAttempts" class="com.sample.PinCodeAttempts">
  <property name="pinCode" defaultValue="1234" displayName="The valid PIN code"/>
  <property name="maxAttempts" defaultValue="3" displayName="How many attempts are allowed"/>
  <property name="failureExpirationSec" defaultValue="60" displayName="How long before the client can tr
y again (seconds)"/>
  <property name="successExpirationSec" defaultValue="60" displayName="How long is a successful state
valid for (seconds)"/>
</securityCheckDefinition>
```

The `name` attribute will be the name of your SecurityCheck, the `class` should be set to the class created previously.

A `securityCheckDefinition` can contain zero or more `property` elements. The `pinCode` property is the one defined in the `PinCodeConfig` configuration class. The other properties are inherited from the `SecurityCheckWithAttemptsConfig` configuration class.

By default, if you do not specify those properties in the adapter.xml you received the defaults set by `SecurityCheckWithAttemptsConfig`:

```
public SecurityCheckWithAttemptsConfig(Properties properties) {
    super(properties);
    maxAttempts = getIntProperty("maxAttempts", properties, 1);
    attemptIntervalSec = getIntProperty("attemptIntervalSec", properties, 120);
    successExpirationSec = getIntProperty("successExpirationSec", properties, 3600);
    failureExpirationSec = getIntProperty("failureExpirationSec", properties, 0);
}
```

Note that the default for `failureExpirationSec` is set to `0`, which means if the client sends invalid credentials, it can try again "after 0 seconds". This means that by default the "attempts" feature is disabled.