

Implementing the CredentialsValidation Security Check

fork and edit tutorial (<https://github.ibm.com/MFPSamples/DevCenter/tree/master/tutorials/en/foundation/8.0/authentication-and-security/credentials-validation/security-check.md>) | report issue (<https://github.ibm.com/MFPSamples/DevCenter/issues/new>)

Overview

This abstract class extends `ExternalizableSecurityCheck` and implements most of its methods to simplify usage. Two methods are required to be implemented: `validateCredentials` and `createChallenge`.

The `CredentialsValidationSecurityCheck` class is meant for simple flows to need to validate arbitrary credentials in order to grant access to a resource. Also provided is a built-in capability to block access after a set number of attempts.

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

Prerequisites: Make sure to read the Authentication concepts ([../authentication-concepts/](https://github.ibm.com/MFPSamples/DevCenter/tree/master/tutorials/en/foundation/8.0/authentication-and-security/authentication-concepts)) and Creating a Security Check ([../creating-a-security-check](https://github.ibm.com/MFPSamples/DevCenter/tree/master/tutorials/en/foundation/8.0/authentication-and-security/creating-a-security-check)) tutorials.

Jump to:

- Creating the Security Check
- Creating the Challenge
- Validating the user credentials
- Configuring the SecurityCheck

PinCodeAttempts

Create a Java adapter ([../adapters/creating-adapters](https://github.ibm.com/MFPSamples/DevCenter/tree/master/tutorials/en/foundation/8.0/authentication-and-security/creating-adapters)) and add a Java class named `PinCodeAttempts` that extends `CredentialsValidationSecurityCheck`.

```
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 return 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 `CredentialsValidationSecurityCheck`.

Validating the user 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, it can also be configured using the adapter.xml file and the MobileFirst Operations 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` method to retrieve a specific property from the adapter.xml file. 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();
}

```

`getConfig().pinCode` can now be used 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
try again (seconds)"/>
    <property name="successExpirationSec" defaultValue="60" displayName="How long is a successful stat
e valid for (seconds)"/>
</securityCheckDefinition>

```

The `name` attribute should be the name of the 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 file 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.

