Enrollment

Overview

This sample demonstrates a custom enrollment process and step-up authorization. During this one-time enrollment process, the user is required to enter his user name and password, and to define a PIN code.

Prerequisites: Make sure to read the ExternalizableSecurityCheck (../externalizable-security-check/) and Step-up (../step-up/) tutorials.

Jump to:

Application Flow

- When the application starts for the first time (before enrollment), it shows the UI with two buttons: **Get public data** and **Enroll**.
- When the user taps on the **Enroll** button to start enrollment, he is prompted with a log-in form and is then requested to set a PIN code.
- After the user has enrolled successfully, the UI includes four buttons: **Get public data**, **Get balance**, **Get transactions**, and **Logout**. The user can access all four buttons without entering the PIN code.
- When the application is launched for a second time (after enrollment), the UI still includes all four buttons. However, when the user clicks the **Get transactions*** button, he is required to enter the PIN code.

After three failing attempts at entering the PIN code, the user is prompted to authenticate again with a user name and password, and to reset a PIN code.

Storing Data in Persistent Attributes

You can choose to save protected data in the `PersistentAttributes` object which is a container for custom attributes of a registered client. The object can be accessed either from a security check class or from an adapter resource class.

In the provided sample application the `PersistentAttributes` object is used in the adapter resource class to store the PIN code:

- The **setPinCode** resource adds the **pinCode** attribute and calls the `AdapterSecurityContext.storeClientRegistrationData()` method to store the changes.

```
@POST
@OAuthSecurity(scope = "setPinCode")
@Path("/setPinCode/{pinCode}")

public Response setPinCode(@PathParam("pinCode") String pinCode){
  ClientData clientData = adapterSecurityContext.getClientRegistrationData()
;
  clientData.getProtectedAttributes().put("pinCode", pinCode);
  adapterSecurityContext.storeClientRegistrationData(clientData);
  return Response.ok().build();
}
```

Here, `users` has a key called `EnrollmentUserLogin` which itself contains the `AuthenticatedUser` object.

- The **unenroll** resource deletes the **pinCode** attribute and calls the `AdapterSecurityContext.storeClientRegistrationData()` method to store the changes.

```
@DELETE
@OAuthSecurity(scope = "unenroll")
@Path("/unenroll")

public Response unenroll(){
  ClientData clientData = adapterSecurityContext.getClientRegistrationData()
;
  if (clientData.getProtectedAttributes().get("pinCode") != null){
   clientData.getProtectedAttributes().delete("pinCode");
   adapterSecurityContext.storeClientRegistrationData(clientData);
  }
  return Response.ok().build();
}
```

Security Checks

The Enrollment sample contains three security checks:

EnrollmentUserLogin

The `EnrollmentUserLogin` security check protects the **setPinCode** resource so that only authenticated users can set a PIN code. This security check is meant to expire quickly and to hold only for the duration of the “first time experience”. It is identical to the `UserLogin` security check explained in the Implementing the UserAuthenticationSecurityCheck (../user-authentication/security-check) tutorial? except for the extra `isLoggedIn` and `getRegisteredUser` methods.
The `isLoggedIn` method returns `true` if the security check state equals SUCCESS and `false` otherwise.
The `getRegisteredUser` method returns the authenticated user.

```
public boolean isLoggedIn(){
    return getState().equals(STATE_SUCCESS);
}
```

```
public AuthenticatedUser getRegisteredUser() {
    return registrationContext.getRegisteredUser();
}
```

EnrollmentPinCode

The `EnrollmentPinCode` security check protects the **Get transactions** resource and is similar to the `PinCodeAttempts` security check explained in the Implementing the CredentialsValidationSecurityCheck (../credentials-validation/security-check) tutorial, except for a few changes.

In this tutorial's example, `EnrollmentPinCode` **depends on** `EnrollmentUserLogin`. After a successfully login to `EnrollmentUserLogin`, the user is only asked to enter a PIN code.

```
@SecurityCheckReference
private transient EnrollmentUserLogin userLogin;
```

When the application starts **for the first time** and the user is successfully enrolled, the user must able to access the **Get transactions** resource without having to enter the PIN code that he just set. For this purpose, the `authorize` method uses the `EnrollmentUserLogin.isLoggedIn` method to check whether the user is logged in. This means that as long as `EnrollmentUserLogin` is not expired, the user can access **Get transactions**.

```
@Override

public void authorize(Set<String> scope, Map<String, Object> credentials, HttpSer
vletRequest request, AuthorizationResponse response) {
    if (userLogin.isLoggedIn()){
        setState(STATE_SUCCESS);
        response.addSuccess(scope, userLogin.getExpiresAt(), getName());
    }
}
```

When the user fails to enter the PIN code after three attempts, the tutorial is designed so that the **pinCode** attribute is deleted before the user is prompted to authenticate by using the user name and password and resetting a PIN code.

```
@Override

public void authorize(Set<String> scope, Map<String, Object> credentials, HttpSer
vletRequest request, AuthorizationResponse response) {
    PersistentAttributes attributes = registrationContext.getRegisteredProtectedA
ttributes();
    if (userLogin.isLoggedIn()){
        setState(STATE_SUCCESS);
        response.addSuccess(scope, userLogin.getExpiresAt(), getName());
    } else {
        super.authorize(scope, credentials, request, response);
        if (getState().equals(STATE_BLOCKED)){
            attributes.delete("pinCode");
        }
    }
}
```

The `validateCredentials` method is the same as in the `PinCodeAttempts` security check, except that here the credentials are compared to the stored **pinCode** attribute.

```
@Override

protected boolean validateCredentials(Map<String, Object> credentials) {
    PersistentAttributes attributes = registrationContext.getRegisteredProtectedA
ttributes();
    if(credentials!=null && credentials.containsKey("pin")){
        String pinCode = credentials.get("pin").toString();

        if(pinCode.equals(attributes.get("pinCode"))){
            errorMsg = null;
            return true;
        }
        else {
            errorMsg = "The pin code is not valid. Hint: " + attributes.get("pinC
ode");
        }
    }
    else{
        errorMsg = "The pin code was not provided.";
    }
    //In any other case, credentials are not valid
    return false;
}
```

IsEnrolled

The `IsEnrolled` security check protects:

- The **getBalance** resource so that only enrolled users can see the balance.
- The **transactions** resource so that only enrolled users can get the transactions.
- The **unenroll** resource so that deleting the **pinCode** is possible only if it has been set before.

Creating the Security Check

Create a Java adapter (../../adapters/creating-adapters/) and add a Java class named `IsEnrolled` that extends `ExternalizableSecurityCheck`.

```
public class IsEnrolled  extends ExternalizableSecurityCheck{
    protected void initStateDurations(Map<String, Integer> durations) {}

    public void authorize(Set<String> scope, Map<String, Object> credentials, Htt
pServletRequest request, AuthorizationResponse response) {}

    public void introspect(Set<String> scope, IntrospectionResponse response) {}
}
```

The IsEnrolledConfig Configuration Class

Create an `IsEnrolledConfig` configuration class that extends `ExternalizableSecurityCheckConfig`:

```
public class IsEnrolledConfig extends ExternalizableSecurityCheckConfig {

    public int successStateExpirationSec;

    public IsEnrolledConfig(Properties properties) {
        super(properties);
        successStateExpirationSec = getIntProperty("expirationInSec", properties,
8000);
    }
}
```

Add the `createConfiguration` method to the `IsEnrolled` class:

```
public class IsEnrolled  extends ExternalizableSecurityCheck{
    @Override
    public SecurityCheckConfiguration createConfiguration(Properties properties)
{
        return new IsEnrolledConfig(properties);
    }
}
```

The initStateDurations Method

Set the duration for the SUCCESS state to `successStateExpirationSec`:

```
@Override
protected void initStateDurations(Map<String, Integer> durations) {
    durations.put (SUCCESS_STATE, ((IsEnrolledConfig) config).successStateExpirat
ionSec);
}
```

The authorize Method

The code sample simply checks whether the user is enrolled and returns success or failure accordingly:

```
public void authorize(Set<String> scope, Map<String, Object> credentials, Http
ServletRequest request, AuthorizationResponse response) {
    PersistentAttributes attributes = registrationContext.getRegisteredProtectedA
ttributes();
    if (attributes.get("pinCode") != null){
        setState(SUCCESS_STATE);
        response.addSuccess(scope, getExpiresAt(), this.getName());
    } else  {
        setState(STATE_EXPIRED);
        Map <String, Object> failure = new HashMap<String, Object>();
        failure.put("failure", "User is not enrolled");
        response.addFailure(getName(), failure);
    }
}
```

- In case the `pinCode` attribute exists:
- Set the state to SUCCESS by using the `setState` method.
- Add success to the response object by using the `addSuccess` method.
- In case the `pinCode` attribute doesn't exist:

- Set the state to EXPIRED by using the `setState` method.
- Add failure to the response object by using the `addFailure` method.

The `IsEnrolled` security check **depends on** `EnrollmentUserLogin`:

```
@SecurityCheckReference
private transient EnrollmentUserLogin userLogin;
```

Set the active user by adding the following code:

```
public void authorize(Set<String> scope, Map<String, Object> credentials, Http
ServletRequest request, AuthorizationResponse response) {
    PersistentAttributes attributes = registrationContext.getRegisteredProtectedA
ttributes();
    if (attributes.get("pinCode") != null){
        // Is there a user currently active?
        if (!userLogin.isLoggedIn()){
            // If not, set one here.
            authorizationContext.setActiveUser(userLogin.getRegisteredUser());
        }
        setState(SUCCESS_STATE);
        response.addSuccess(scope, getExpiresAt(), this.getName());
    } else  {
        setState(STATE_EXPIRED);
        Map <String, Object> failure = new HashMap<String, Object>();
        failure.put("failure", "User is not enrolled");
        response.addFailure(getName(), failure);
    }
}
```

Then, the `transactions` resource gets the current `AuthenticatedUser` object to present the display name:

```
@GET
@Produces(MediaType.TEXT_PLAIN)
@OAuthSecurity(scope = "transactions")
@Path("/transactions")

public String getTransactions(){
  AuthenticatedUser currentUser = securityContext.getAuthenticatedUser();
  return "Transactions for " + currentUser.getDisplayName() + ":\n{'date':'12/01/
2016', 'amount':'19938.80'}";
}
```

> For more information about the `securityContext`, see the Security API (../../adapters/java-adapters/#security-api) section in the Java adapter tutorial.

Add the registered user to the response object by adding the following:

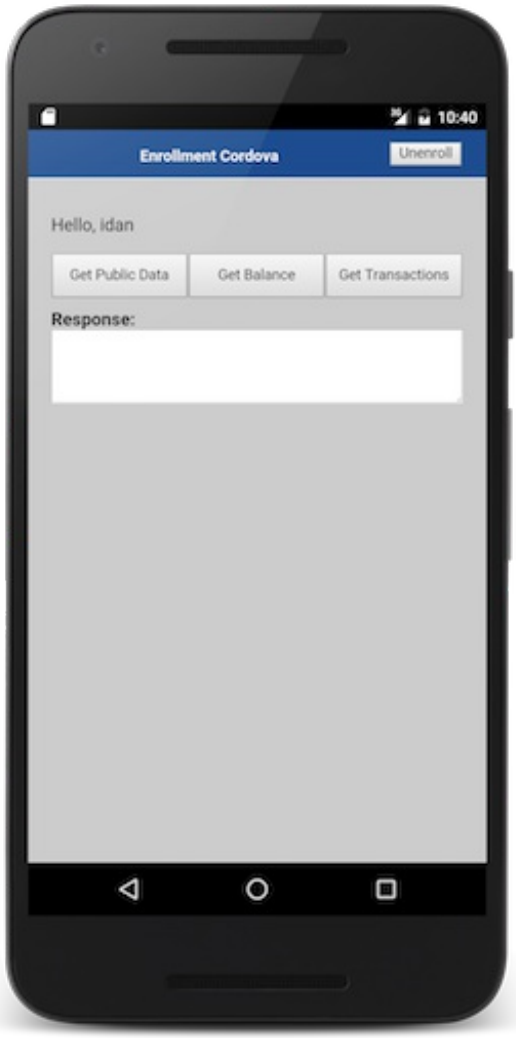
```
public void authorize(Set<String> scope, Map<String, Object> credentials, Http
ServletRequest request, AuthorizationResponse response) {
    PersistentAttributes attributes = registrationContext.getRegisteredProtectedA
ttributes();
    if (attributes.get("pinCode") != null){
        // Is there a user currently active?
        if (!userLogin.isLoggedIn()){
            // If not, set one here.
            authorizationContext.setActiveUser(userLogin.getRegisteredUser());
        }
        setState(SUCCESS_STATE);
        response.addSuccess(scope, getExpiresAt(), getName(), "user", userLogin.g
etRegisteredUser());
    } else  {
        setState(STATE_EXPIRED);
        Map <String, Object> failure = new HashMap<String, Object>();
        failure.put("failure", "User is not enrolled");
        response.addFailure(getName(), failure);
    }
}
```

In our sample code, the `IsEnrolled` challenge handler's `handleSuccess` method use the user object to present the display name.

Sample Applications

Security check

The `EnrollmentUserLogin`, `EnrollmentPinCode`, and `IsEnrolled` security checks are available in the SecurityChecks project under the Enrollment Maven project. Click to download (https://github.com/MobileFirst-Platform-Developer-Center/SecurityCheckAdapters/tree/release80) the Security Checks Maven project.



Applications

Sample applications are available for iOS (Swift), Android, Cordova, and Web.

- Click to download (https://github.com/MobileFirst-Platform-Developer-Center/EnrollmentCordova/tree/release80) the Cordova project.
- Click to download (https://github.com/MobileFirst-Platform-Developer-Center/EnrollmentSwift/tree/release80) the iOS Swift project.
- Click to download (https://github.com/MobileFirst-Platform-Developer-Center/EnrollmentAndroid/tree/release80) the Android project.
- Click to download (https://github.com/MobileFirst-Platform-Developer-Center/EnrollmentWeb/tree/release80) the Web app project.

Sample usage

Follow the sample's README.md file for instructions.

Last modified on November 17, 2016