Migrating Authentication and Security Concepts

Overview

The security framework of MobileFirst Foundation has undergone some major changes in version 8.0, to improve and simplify security development and administration tasks. In particular, the security building blocks have changed – In 8.0, OAuth security scopes and security checks replace the security tests, realms and login modules of previous versions.

This tutorial guides you through the steps required for migrating the security code of your application. We will use a sample 7.1 application as a starting point, and describe the complete process that will take us from the 7.1 sample application to a 8.0 application, with the same security protection. Both the 7.1 sample app and the migrated application are attached here.

The migration steps that we will describe below are:

- Migrating the resource adapter to 8.0, and maintaining the protection of the resources
- Migrating the client app
- Creating security checks to replace the authentication realms of the 7.1 app
- Modifying the challenge handlers on the client-side to use the new challenge handler API.

In the second part of this tutorial, we will address additional migration issues that are not demonstrated in the migration of the sample app:

- Migrating other types of authorization realms, beyond form-based authentication and adapter-based authentication that are demonstrated in the sample.
- Access token expiration
- Application-level protection (application security test)
- Security configuration settings in 7.1 that are no longer required in the simpler security model of 8.0, such as the user identity realm and the device identity realm

> Before you start the migration you are advised to go through the migration cookbook (../migration-cookbook). See also the Authentication and Security tutorial (../../authentication-and-security) to learn about the basic concepts of the new security framework.

The sample application

Our starting point is a sample 7.1 hybrid application. The application accesses a Java adapter protected with OAuth. The adapter has two methods – the `getBalance` method, which is protected with a form-based authentication realm (login by user name and password), and the `transferMoney` method, protected with an adapter-based authentication realm, requiring the user to provide a pin code. The source code of the 7.1 sample application and the source code of the same application after it has been migrated to 8.0 are available for download (https://github.com/MobileFirst-Platform-Developer-Center/MigrationSample).

Migrating the resource adapter

We will start the migration process with the resource adapter. In MobileFirst 8.0 adapters are developed as separate Maven projects, unlike in 7.1 where adapters were part of the MobileFirst project. This means that we can migrate the resource adapter, build and deploy it, independently of the client app. The same is true for the client app itself and the security checks (which are actually also deployed as adapters). This leaves us the freedom to migrate these parts in the order of our choice. In this tutorial, we will migrate the resource adapter first so that we can introduce the OAuth security scope elements by which the resources are protected.

Note that we are going to migrate the resource adapter `AccountAdapter`, but there is no need to migrate the other adapter, `PinCodeAdapter`, which is used for adapter-based authentication, because adapter-based authentication is no longer supported in 8.0. In one of the next steps we will replace that adapter with a MobileFirst 8.0 security check.

> For instructions on migrating adapters to 8.0 see the migration cookbook (../migration-cookbook).

The methods of `AccountAdpter` in the 7.1 sample are already protected with the `@OAuthSecurity` annotation. The same annotation is used in version 8.0. The only difference is that in 7.1 the scope elements `UserLoginRealm` and `PinCodeRealm` refer to security realms that are defined in the authenticationConfig.xml file. In 8.0, on the other hand, scope elements are

mapped to security checks deployed on the server. We could keep the code unchanged with the same names of the scope elements, but let's rename the scope elements to `UserLogin` and `PinCode` because the term "realm" is no longer used in MFP 8.0:

```
@OAuthSecurity(scope="UserLogin")

@OAuthSecurity(scope="PinCode")
```

Use the new API for getting the user identity

Our resource adapter uses the server-side security API to obtain the identity of the authenticated user. This API has changed in 8.0, so we need to fix it. Replace the following 7.1 code:

```
WLServerAPI api = WLServerAPIProvider.getWLServerAPI();
api.getSecurityAPI().getSecurityContext().getUserIdentity();
```

with the new API in 8.0:

```
// Inject the security context
@Context
AdapterSecurityContext securityContext;

 // Get the authenticated user name
String userName = securityContext.getAuthenticatedUser().getDisplayName();
```

Build the adapter and deploy it to the server (../../adapters/creating-adapters/#build-and-deploy-adapters), using either Maven or the MobileFirst CLI.

Migrating the client application

Next, we will migrate the client application. Refer to the migration cookbook for client application migration instructions. For the time being, comment out the code of the challenge handlers. We will fix the challenge handlers later. In the main HTML file of the app, index.html, put a comment around the lines that import the challenge handlers code.

```
<!--
    <script src="js/UserLoginChallengeHandler.js"></script>
    <script src="js/PinCodeChallengeHandler.js"></script>
 -->
```

Change to the new client API for logout

As part of the client migration, you need to handle changes in the client-side APIs of MobileFirst 8.0. For a list of client API changes, see Upgrading the WebView (../migrating-client-applications/cordova/#upgrading-the-webview). In our sample application, there is one client API change related to security – the API for logging out. The method `WL.Client.logout` of 7.1 is not supported in 8.0. Instead, use `WLAuthorizationManager.logout`, and pass the name of the security check that replaces the authorization realm of 7.1. The Logout button in our sample app logs the user out from both the `UserLogin` security check and the `PinCode` security check:

```
function logout() {
    WLAuthorizationManager.logout('UserLogin').then(
        function () {
            WLAuthorizationManager.logout('PinCode').then(function () {
                $("#ResponseDiv").html("Logged out");
            }, function (error) {
                WL.Logger.debug("failure on logout from PinCode check: " +
                    JSON.stringify(error));
            });
      },
      function (error) {
          WL.Logger.debug("failure on logout from UserLogin check: " +
              JSON.stringify(error));
      });
}
```

After finishing the steps for migrating the app, build the application, and register it on your MobileFirst server using the command `mfpdev app register`. You should now see the application listed in the MobileFirst operations console.

Migrating the form-based authentication realm

At this stage we already have the client application and the resource adapter migrated and deployed. However if we try to run the application now, it will not be able to access the resources. This is because the application is expected to present an access token that contains the scope elements required by the resource adapter methods ("UserLogin" or "PinCode"), but since we haven't created the security checks yet, the application cannot obtain an access token and the application is not authorized to access the protected resources.

We will now create an 8.0 security check named "UserLogin" in replacement for the 7.1 form-based authentication realm "UserLoginRealm". The security check will perform the same authentication steps that were previously implemented by the form-based authenticator and the custom login module – sending a challenge to the client, collecting the credential from the challenge response, validating the credentials and creating a user identity. As you will see below, creating the security check is quite straightforward, and what remain for you is to copy the code for validating the credentials from the 7.1 custom login module to the new security check.

Security checks are implemented as adapters, and therefore we start by creating a new Java adapter (../../adapters/creating-adapters) named `UserLogin`.

When creating a Java adapter, the default template assumes that the adapter will serve resources. The same adapter can be used both for serving resources and for packaging a security test, but in this case we use the new adapter just for the security check. Therefore, we will remove the default resource implementation: delete the files UserLoginApplication.java and UserLoginResource.java. Remove the element from adapter.xml, too. In the Java adapter's adapter.xml file, add an XML element called `securityCheckDefinition`. For example:

```
<securityCheckDefinition name="UserLogin" class="com.sample.UserLogin">
     <property name="successStateExpirationSec" defaultValue="3600"/>
</securityCheckDefinition>
```

- The name attribute is the name of your security check.
- The class attribute specifies the implementation Java class of the security check. We will create this class in the next step.
- The property successStateExpirationSec is equivalent to the expirationInSeconds property of 7.1 login modules. It indicates the interval in seconds during which successful login to this security check holds. The default value of these properties is 3600 seconds in both 7.1 and 8.0. If the 7.1 login module was configured with a different value, you should put the same value here.

For the purpose of this tutorial we're only defining the `successStateExpirationSec` property. There is actually much more you can do with security check configuration (../../authentication-and-security/creating-a-security-check/#security-check-configuration). In particular, you can configure your security check to use some advanced features such as blocked state expiration, multiple attempts and "remember me". You can add custom configuration properties, and modify the configuration properties in runtime from the MFP console.

Creating the security check Java class

Create a Java class named `UserLogin` that extends `UserAuthenticationSecurityCheck` and add it to the adapter. Next, we override the default implementation of the three methods, `createChallenge`, `validateCredentials` and `createUser`.

- The method `validateCredentials` is where we put the authentication logic. Copy the authentication logic code – the code that validates the username and the password – from the 7.1 login module and put it here. In this case the logic is very simple – we just test that the password is the same as the user name.
- In the method `createChallenge` we create the challenge message (hash map) to be sent to the client. In general, a security check can put here a challenge phrase or some other kind of challenge object that will be used to validate the client's response. This security check does not require a challenge phrase so all we need to put in the challenge message is the error message (if an error was found).
- The `createUser` method is the equivalent of the `createIdentity` method in the 7.1 login module.

The complete class is shown below.

```
public class UserLogin extends UserAuthenticationSecurityCheck {
    private String userId, displayName;
    private String errorMsg;

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

            // the authentication logic, copied from the 7.1 login module
            if (!username.isEmpty() && !password.isEmpty() && username.equals(password)) {
                userId = username;
                displayName = username;

                errorMsg = null;
                return true;
            } else {
                errorMsg = "Wrong Credentials";
            }
        } else {
            errorMsg = "Credentials not set properly";
        }
        return false;
    }

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

    @Override
    protected AuthenticatedUser createUser() {
        return new AuthenticatedUser(userId, displayName, this.getName());
    }
}
```

Build the adapter and deploy it to the server (../../adapters/creating-adapters/#build-and-deploy-adapters), using either Maven or the MobileFirst CLI. In the MobileFirst console you should see the new adapter UserLogin in the list of adapters

Migrating the pin code realm

The pin code realm in our sample is implemented with adapter-based authentication, which is no longer supported in 8.0. We will replace this realm with a new security check.

Create a new Java adapter named `PinCode`. Create a Java class named `PinCode` that extends `CredentialsValidationSecurityCheck` and add it to the adapter. Note that this time we use `CredentialsValidationSecurityCheck` as the base class, and not `UserAuthenticationSecurityCheck`, which we used for the UserLogin security check. This is because the pin code security check only needs to validate the credentials (the pin code) but it doesn't assign a user identity.

To create a security check that extends `CredentialsValidationSecurityCheck` we need to implement two methods: `createChallenge` and `validateCredentials`.

Similar to the `UserLogin` security check, the `PinCode` security check does not have any special information to send to the client as part of the challenge. The `createChallenge` method only puts the error message (if exists) inside the challenge message.

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

The `validateCredentials` method validates the pin code. In this case, the validation code consists of one line of code, but in general you can copy the validation code from the 7.1 authentication adapter into the `validateCredentials` method.

```
@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";
    }
    return false;
}
```

Build the adapter and deploy it to the server (../../adapters/creating-adapters/#build-and-deploy-adapters).

Migrating the challenge handlers

We're almost there – we have the client app migrated, the resource adapter and the security checks to protect the resources. The only part missing are the challenge handlers on the client side that allow the client to respond to the challenges and send the credentials to the security check. Remember that when we migrated the client app, we commented out the lines that include the challenge handlers. Now is the time to uncomment these lines, and migrate the challenge handlers to 8.0.

We will start with the user login challenge handler. This challenge handler performs the same functions in 8.0 as it does in 7.1 – it's responsible for presenting the login form to the user upon receiving a challenge, and sending the user name and password to the server. However, the client API for challenge handlers has been changed and simplified, so we need to make the following changes:

- Replace the call for creating challenge handler with:

```
var userLoginChallengeHandler = WL.Client.createSecurityCheckChallengeHandler('UserLogin');
```

The method `createSecurityCheckChallengeHandler` creates a challenge handler that handles challenges sent by a MobileFirst security check. In most cases, you should use this method in replacement for either the method `createWLChallengeHandler` or the method `createChallengeHandler` of the 7.1 client API. The only exception is challenge handlers that are designed to handle challenges sent by a third party gateway. This type of challenge handlers, called gateway challenge handlers in 8.0, are created with the method `WL.Client.createGatewayChallengeHandler(). For example, if your resource is protected by a reverse proxy such as DataPower, which sends a custom login form to the client, you should use a gateway challenge handler to handle the challenge. For more information on gateway challenge handlers see the article Quick Review of Challenge Handlers (https://mobilefirstplatform.ibmcloud.com/blog/2016/06/22/challenge-handlers/).

- Remove the method `isCustomResponse`. It is no longer needed for security check challenge handlers.
- Replace the method `handleChallenge` with the three methods that a challenge handler must implement – `handleChallenge()`, `handleSuccess()` and `handleFailure`. In 8.0 the challenge handler no longer has to check the response in order to find if the response carries a challenge, success or error. The framework takes care for that, and calls the appropriate method.
- Remove the call to `submitSuccess`. The framework handles the success response automatically.
- Replace the call to `submitFailure` with `userLoginChallengeHandler.cancel`.
- Replace the call to `submitLoginForm` with:

  ```
  userLoginChallengeHandler.submitChallengeAnswer({'username':username, 'password':password}
  )
  ```

The complete code of the challenge handler after applying these changes is shown below.

```
function createUserLoginChallengeHandler() {
    var userLoginChallengeHandler = WL.Client.createSecurityCheckChallengeHandler('UserLogin');

    userLoginChallengeHandler.handleChallenge = function(challenge) {
        showLoginDiv();
        var statusMsg = (challenge.errorMsg !== null) ? challenge.errorMsg : "";
        $("#loginErrorMessage").html(statusMsg);
    };

    userLoginChallengeHandler.handleSuccess = function(data) {
        hideLoginDiv();
    };

    userLoginChallengeHandler.handleFailure = function(error) {
        if (error.failure !== null) {
            alert(error.failure);
        } else {
            alert("Failed to login.");
        }
    };

    $('#AuthSubmitButton').bind('click', function () {
        var username = $('#AuthUsername').val();
        var password = $('#AuthPassword').val();
        if (username === "" || password === "") {
            alert("Username and password are required");
            return;
        }

        userLoginChallengeHandler.submitChallengeAnswer(
            {'username':username, 'password':password});});

    $('#AuthCancelButton').bind('click', function () {
        userLoginChallengeHandler.cancel();
        hideLoginDiv();
    });

    return userLoginChallengeHandler;
 }
```

The migration of the pin code challenge handler is very similar to the migration of the user login challenge handler, and therefore we will not show the details here. See the code of the migrated challenge handler in the attached 8.0 sample. This completes the migration of the app. You can now rebuild the application, deploy it to the server, test that it works, and test that access to the resources is protected as expected.

Migrating other types of authentication realms

In the sections above we described the process for migrating form-based authentication realms and adapter-based authentication realms. Your 7.1 application might include other types of realms, including realms that were explicitly added to the application security test, or included by default in a `mobileSecurityTest` or a `webSecurityTest`. See below guidelines for migrating other types of realms to 8.0.

Application authenticity

Application authenticity is provided as a predefined security check in 8.0. By default, this security check is run during the application's runtime registration with MobileFirst Server, which occurs the first time an instance of the application attempts to connect to the server. However, as with any MobileFirst security check, you can also include this predefined check in custom security scopes.

LTPA realm

Use the predefined 8.0 security check, `LtpaBasedSSO`. For more information, see the tutorial Protecting IBM MobileFirst Foundation 8.0 application traffic using IBM DataPower (https://mobilefirstplatform.ibmcloud.com/blog/2016/06/17/datapower-integration/).

Device provisioning

The client registration process in 8.0 replaces device provisioning of 7.1. In MobileFirst 8.0, a client (an instance of an application) registers itself with MobileFirst server on the first attempt to access the server. As part of the registration, the client provides a public key that will be used for authenticating its identity. This protection mechanism is always enabled, and there is no need for you to migrate the device-provisioning realm to 8.0.

Anti-cross site request forgery (anti-XSRF) realm

Anti-XSRF is no longer relevant in the OAuth-based security framework of 8.0.

Direct Update realm

There is no need to migrate the Direct Update realm to 8.0. The Direct Update feature is supported in MobileFirst 8.0, but it does not require a security check, such as the Direct Update realm that was required in previous versions. Note however that the steps to deliver updates by using the Direct Update feature have changed. For more information see the Migrating Direct Update (http://www.ibm.com/support/knowledgecenter/SSHS8R_8.0.0/com.ibm.worklight.upgrade.doc/dev/c_upgrade_direct_update.html) user documentation topic.

Remote disable realm

There is no need to migrate the Remote Disable realm to 8.0. The remote disable feature in MobileFirst 8.0 does not require a security check.

Custom authenticators and login modules

Create a new security check as described above. Use either of the base classes `UserAuthenticationSecurityCheck` or `CredentialsValidationSecurityCheck`. Although you cannot migrate the authenticator class or the login module class directly, you may copy relevant code pieces into the security check, such as code for generating the challenge, extracting credentials from the response, and validating the credentials.

Migrating additional security configurations of 7.1

The application security test

In addition to the OAuth scopes that are used to protect the resource adapter, our 7.1 sample application is also protected by an application-level security test. This sample does not have the application security test defined in the application-descriptor.xml file, and hence it is protected with the default security test. The default security test for mobile applications in 7.1 consists of realms that either irrelevant in 8.0 (anti-XSRF) or do not require explicit migration (Direct Update, remote disable). Therefore in this case no migration is required for the application security test.

If your application has an application security test that includes checks (realms) that you still want to keep at the application level after the migration to 8.0, you can configure a mandatory scope for the application. When an application tries to access a protected resource, it has to pass the security checks that are mapped to the mandatory scope in addition to the checks mapped to the scope protecting the resource.

To define a mandatory scope for an application, select the application version in the MobileFirst console, select the Security tab and click the Add to Scope button. You can include in the scope any predefined or custom security checks, or mapped scope elements.

Access token expiration

Check the value of the Access Token Expiration property in application-descriptor.xml file. The default value in both version 7.1 and version 8.0 is 3600 seconds, so unless your application has a different value defined in the application descriptor file, you don’t have to change anything. To set the expiration value in 8.0, navigate to the application version page in the MobileFirst console, select the security tab, and enter the value in the Maximum Token-expiration Period field.

User identity realm

In MobileFirst 7.1, authentication realms may be configured as user identity realms. Applications that use OAuth authentication flow use the `userIdentityRealms` property in the application descriptor file to define an ordered list of user identity realms. In applications that use the classic Worklight authentication flow (non OAuth), the attribute `isInternalUserId` indicates whether the realm is a user identity realm. These configurations are no longer needed in MobileFirst 8.0. In MobileFirst 8.0, the active user identity is set by the last security check that called the `setActiveUser` method. If your security check extends the abstract base class `UserAuthenticationSecurityCheck`, like the UserLogin security check in our sample application, the base class takes care for setting the active user.

Device identity realm

In 7.1 applications there must be a realm that is defined as a device identity realm. No migration is needed for this configuration in 8.0. In MobileFirst 8.0, the device identity isn't associated with a security check. The device information is registered as part of the client registration flow, which occurs on the first time the client attempts to access a protected resource.

Summary

In this tutorial we covered only the basic steps required to migrate the security artifacts of existing applications from previous versions. We encourage you to learn more about the new security framework (https://mobilefirstplatform.ibmcloud.com/tutorials/en/foundation/8.0/authentication-and-security/) and take advantage of additional features that were not covered here.

Last modified on November 24, 2016