Client X.509 Certificate Authentication and User Enrollment

Topics covered in this tutorial:

- Overview
- Prerequisites
- Understanding how user certificate authentication works
- X.509 certificate and certificate authorities (CAs)
    - Create root CA, signing CA, and certificates
    - Create root CA
    - Create signing CA
    - Create server certificate
    - Create certificate chain for the server certificate
    - Export a PKCS12 file for the signing CA
    - Export a PKCS12 file for the server certificate
- Configure IBM WebSphere Application Server Liberty profile (Liberty)
- Configure authenticationConfig.xml
- Configure application-descriptor.xml
- Install root CA on iOS and Android
- Install application and test

Overview

The X.509 User Certificate Authentication feature is a user realm that establishes user identity with an X.509 client certificate.

The user identity is established for a particular user on a specific device and application.

This feature provides SSL client-side certificate authentication and user enrollment capabilities.

SSL client-side certificate authentication consists of establishing a two-way SSL handshake between MobileFirst Platform client and server, which in turn, enables the client and server both to present their identities and therefore establish mutual trust through the SSL/TLS protocol.

You can enroll new users to the MobileFirst Platform Mobile Application Management system and your PKI of choice with the user enrollment capabilities.

A basic embedded PKI is provided with this feature that is meant to get you started quickly for educational and non-production environments only.

For production environments, this feature makes it easy to integrate with your existing PKI

- You can use either the PKI Bridge Java™ interface or built-in MobileFirst Platform adapters to delegate certificate management functions down to an external PKI system

In this module, you learn how to enable and configure the User Certificate Authentication user realm.

- How to use the embedded PKI that is provided by MobileFirst Platform

For production environments, this feature makes it easy to integrate with your existing PKI.

- You can use either the PKI Bridge Java™ interface or built in MobileFirst Platform adapters to delegate certificate management functions down to an external PKI system

Prerequisites

You must have a general understanding of MobileFirst Platform user realms and adapters.

It is assumed that you follow these instructions by using an application that currently supports form-based authentication.

- The form-based authentication module uses non-validating login modules. These login modules are not recommended for production environments
- Use other user authentication realms, like WASLTPA in production.

X.509 certificate and certificate authorities (CAs)

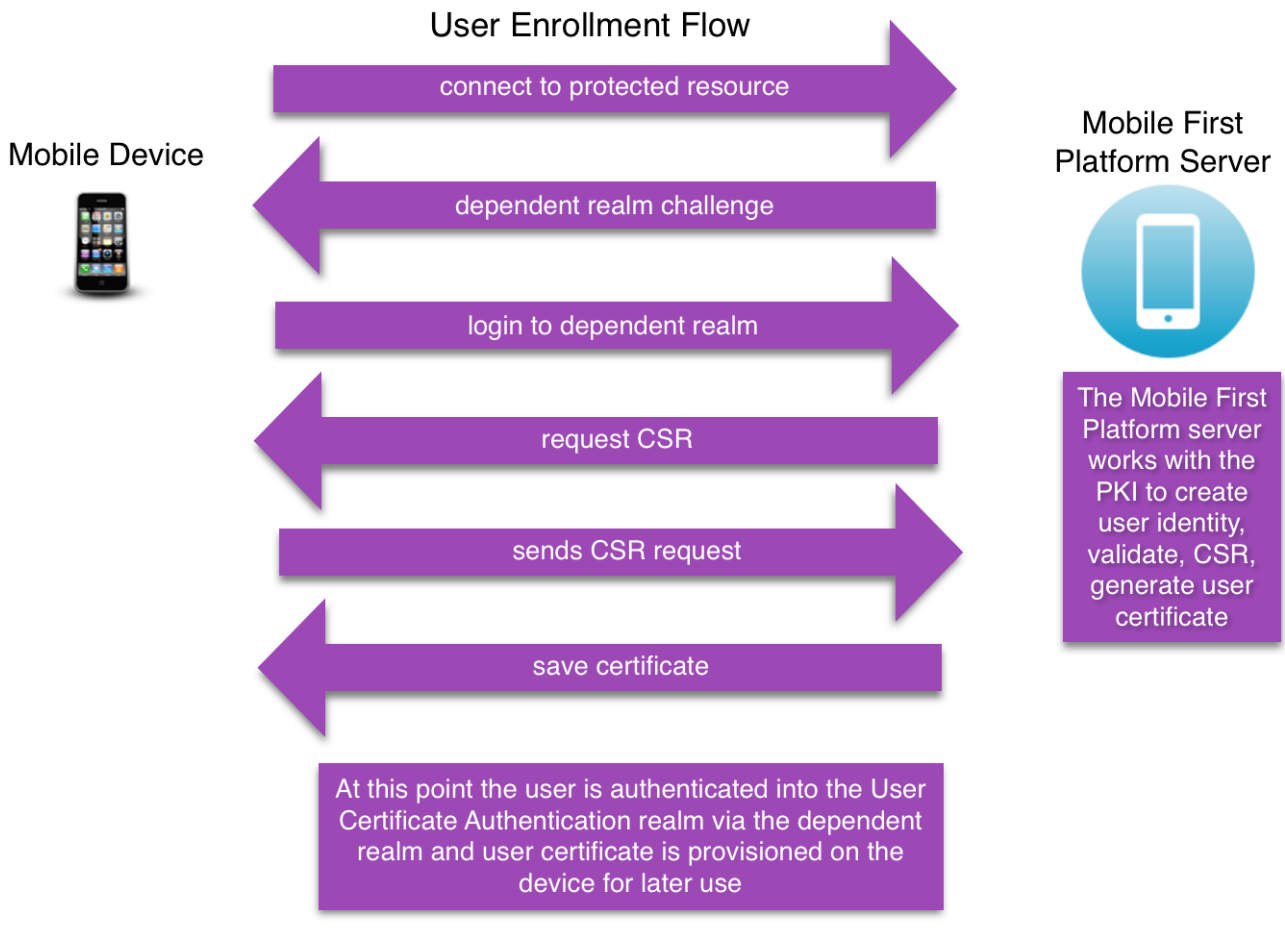
User certificate authentication is the process in which an X.509 certificate is issued by an existing PKI through the MobileFirst Platform server to a specific user on a specific application and device.

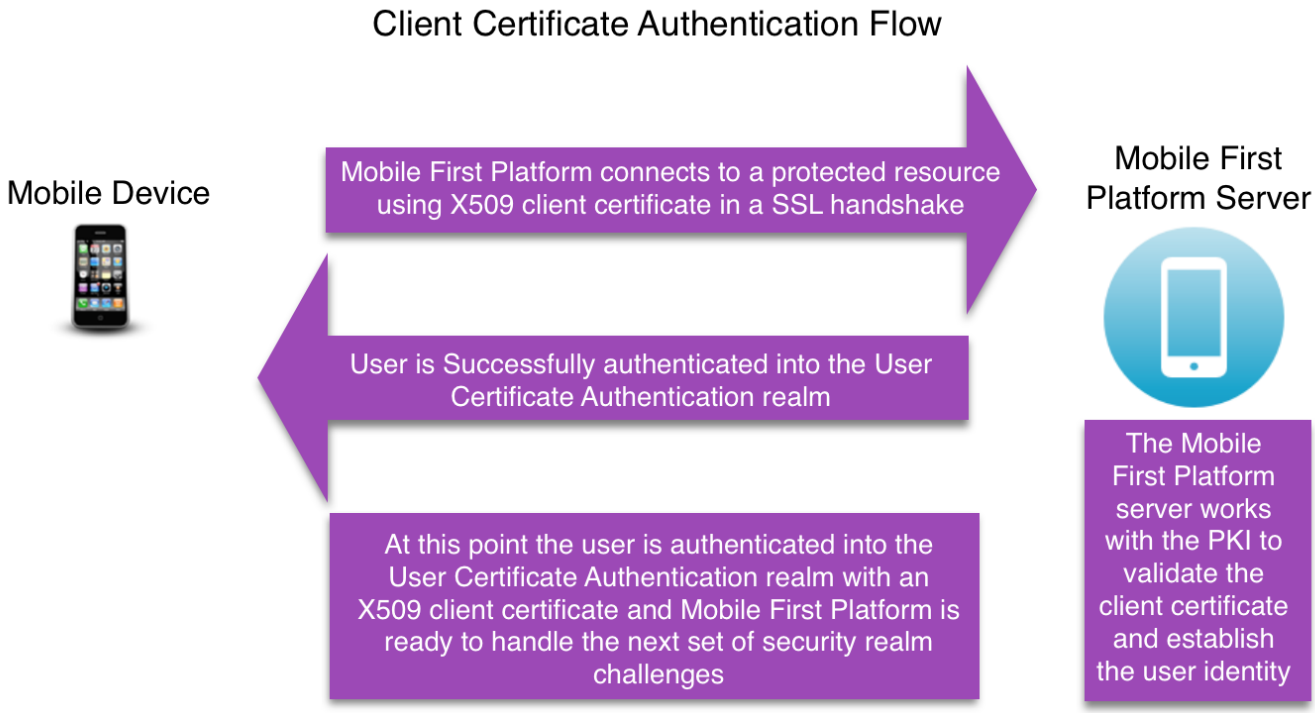
The relevant user information is obtained during the user enrollment process with the specified dependent user realm.

The user enrollment process relies on a dependent user realm to help it establish the initial user identity to which the X.509 certificate is issued.

MobileFirst then provisions the device with the X.509 client certificate for use in subsequent connections to the server.

The first time a user connects to the MobileFirst Platform server, the user must authenticate through the dependent realm to initiate the enrollment process. After a user is enrolled into the User Certificate Authentication realm, subsequent connections to the server occur through the two way SSL/TLS handshake, where the client certificate is presented as the SSL client entity.





X.509 certificate and certificate authorities (CAs)

For security reasons, during testing, it is not recommended to use an established CA that uses an embedded PKI in your infrastructure.

It is possible to create a self-signed CA that can sign both a server certificate and user certificates.

This module uses the OpenSSL command-line utility.

OpenSSL is included in most Linux distributions and in Mac OS X. Windows users can obtain an OpenSSL binary from the OpenSSL website.

The commands that are shown in this module work on Linux and Mac OS X. For Windows, use the equivalent MS-DOS commands.

Create root CA, signing CA, and certificates

Create an empty directory and navigate to that path in your system's terminal.

Create a basic OpenSSL configuration file that is named `openSSL.cnf`. Move this file to the directory that you created.

If you want different policy requirements, see the OpenSSL configuration documentation for instructions on how to configure the various options.

`openSSL.cnf` sample file:

```
[ req ]
    default_bits          = 2048                  # size of keys
    default_keyfile       = key.pem               # name of generated keys
    default_md            = sha1                  # message digest algorithm
    string_mask           = nombstr               # permitted characters
    distinguished_name    = req_distringuished_name

[ req_distinguished_name ]
0.organizationName
organizationUnitName
emailAddress
emailAddress_max
localityName
stateOrProvinceName
countryName
countryName_min
countryName_max
commonName
commonName_max

[ policy_match ]
countryName                = optional
stateOfProvinceName        = optional
localityName               = optional
organizationName           = optional
orgainizationalUnitName    = supplied
commonName                 = optional
emailAddress               = optional
```

Create root CA

Append the following section to the `openSSL.cnf` configuration file to set up the root CA requirements.

```
[ root_authority ]
basicConstraints        = CA:TRUE
subjectKeyIdentifier    = hash

[ root_authority_ca_config ]
dir              = ./rootca
certs            = $dir/certs
new_certs_dir    = $dir/newcerts
database         = $dir/index.txt
certificate      = $dir/root_ca.crt
private_key      = $dir/root_ca_key.pem
serial           = $dir/serial
RANDFILE         = $dir/.rand
policy           = $dir/policy_match
```

Using a terminal, create the folder structure and requirements for the root CA

Create a root CA certificate directory structure

```
mkdir rootca
mkdir rootca/certs rootca/crl rootca/newcerts
touch rootca/serial

export HEXOUT=0123456789ABCDEF
```

Create a serial list of random numbers

```
for y in {1..2048}
do
export output="";
for i in {1..16}
do
   export randomnum=$RANDOM%16;
   export output=$output${HEXOUT:$randomnum:1};
done
echo "$output" >> rootca/serial
done

touch rootca/index.txt
```

For Windows, create the folder structure and requirements for the root CA.
Create a root CA certificate directory structure

```
MKDIR rootca
MKDIR rootca\certs
MKDIR rootca\crl
MKDIR rootca\newcerts
```

Create a serial list of random numbers for the root CA

```
openssl rand -hex -out rootca\serial 8
```

Create index for root CA

```
COPY NUL rootca\index.txt
```

Using a terminal, generate an RSA key pair and then self sign a root CA certificate. The password must remain secure, even for a test environment. For the following example, the password is `passRoot`.

```
export ROOT_CA_SUBJECT="Development Root CA"
```

Create the RSA key pair.
The parameter, 2048, represents the key length.

```
openssl genrsa -des3 -out rootca/root_ca_key.pem -passout pass:passRoot 2048
```

Sign a certificate with the key pair

```
openssl req -new -x509 -nodes -sha1 -days 365 -key rootca/root_ca_key.pem -out rootca/root_ca.crt -config PATH_TO_/ope
nssl.conf -subj "/CN=$ROOT_CA_SUBJECT" -extensions root_authority -passin pass:passRoot
```

For Windows, generate an RSA key pair and then self-sign a root CA certificate. The password must remain secure, even for a test environment. For the following example, the password is `passRoot`.

Create the RSA key pair
The parameter, 2048, represents the key length

```
openssl genrsa -des3 -out rootca\root_ca_key.pem -passout pass:passRoot 2048
```

Sign a certificate with the key pair

```
openssl req -new -x509 -nodes -sha1 -days 365 -key rootca\root_ca_key.pem -out rootca\root_ca.cert -config openssl.cnf -su
bj "/CN=Development Root CA" -extensions root_authority -passin pass:passRoot
```

Create signing CA

Edit the `openssl.conf` file that you created earlier and append the following configuration to set up the signing CA configuration options:

```
[ signing_authority ]
basicConstraints            = CA:TRUE,pathlen:0
subjectKeyIdentifier        = hash
authorityKeyIdentifier      = keyid:always, issuer:always

[ signing_authority_ca_config ]
dir             = ./signingca
certs           = $dir/certs
new_certs_dir   = $dir/newcerts
database        = $dir/index.txt
certificate     = $dir/signing_ca.crt
private_key     = $dir/signing_ca_key.pem
serial          = $dir/serial
RANDFILE        = $dir/.rand
policy          = policy_match
email_in_dn     = false
```

Using the terminal, create the folder structure and requirements for the signing CA. Run these commands from the base directory.

Create a signing CA certificate directory structure

```
mkdir signingca
mkdir signingca/certs signingca/crl signingca/newcerts
touch signingca/serial
export HEXOUT=0123456789ABCDEF
```

Create a serial list of random numbers

```
for y in {1..2048}
do
export output="";
for i in {1..16}
do
    export randomnum=$RANDOM%16;
    export output=$output${HEXOUT:$randomnum:1};
done
echo "$output" >> signingca/serial
done

touch signingca/index.txt
```

For Windows, create the folder structure and requirements for the signing CA. Run these commands from the base directory.
Create a signing CA certificate directory structure

```
MKDIR signingca
MKDIR signingca\certs
MKDIR signingca\crl
MKDIR signingca\newcerts
```

Create a serial list of random numbers for the signing CA

```
openssl rand -hex -out signingca\serial 8
```

Create index for signing CA

```
COPY NUL signingca\index.txt
```

Using the terminal, generate an RSA key pair, and then sign a signing CA CSR with the root CA. For this example, the password is `passSigning`. Run these commands from the base directory.

```
export SIGNING_CA_SUBJECT="Development Signing CA"

openssl genrsa -des3 -out signingca/signing_ca_key.pem -passout pass:passSigning 2048

openssl req -new -key signingca/signing_ca.csa -out signingca/signing_ca.crt -keyfile rootca/root_ca_key.pem -cert rootca/ro
ot_ca.crt -config openssl.cnf -name root_authority_ca_config -extensions signing_authority -md sha512 -days 365 -passin pa
ss:passRoot
```

For Windows, generate an RSA key pair,and then sign a signing CA CSR with the root CA. For this example the password is `passSigning`. Run these commands from the base directory.

```
openssl genrsa -des3 -out signingca\signing_ca_key.pem -passout pass:passSigning 2048

openssl req -new -key signingca\signing_ca_key.pem -out signingca\signing_ca.csr -config openssl.cnf -subj "/CN=Developm
ent Signing CA" -passin pass:passSigning

openssl ca -in signing_ca.csr -out signingca\signing_ca.crt -keyfile rootca\root_ca_key.pem -cert rootca\root_ca.crt -config op
enssl.cnf -name root_authority_ca_config -extensions signing_authority -md sha512 -days 365 -passin pass:passRoot
```

Create server certificate

Edit the `openssl.cnf` file that you create earlier and append the following configuration to set up the server certificate configuration options.

```
[ server_identity ]
basicConstraints       = CA:TRUE
subjectKeyIdentifier   = hash
authorityKeyIdentifier = keyid:always;issuer:always
```

using the terminal, generate an RSA key pair and sign the new certificate with the signing CA. This certificate is your server identity certificate. The example uses passServer as the password. Run these commands from the base directory.

Use the full hostname of your MobileFirst Platform server. SSL will break if the full hostname is not provided, or if an IP address is used as the hostname.

```
export SERVER_FULL_HOSTNAME=dev.yourcompany.com
mkdir server
```

Create the RSA key pair and generate a CSR

```
openssl genrsa -des3 -out server/server_key.pem -passout pass:passServer 2048
openssl req -new -key server/server_key.pem -out server/server.csr -config openssl.cnf -subj "/CN=$SERVER_FULL_HOST
NAME" -passin pass:passServer
```

Sign the CSR with the signing CA

```
openssl ca -in server/server.csr -out server/server.crt -keyfile signingca/signing_ca_key.pem -cert signignca/signing_ca.crt -c
onfig openssl.cnf -name signing_authority_ca_config -extensions sever_identity -md sha512 -days 365 -passin pass:passSig
ning
```

For windows, generate an RSA key pair and sign the new certificate with the signing CA. This certificate is your server identity certificate. The example uses `passServer` as the password. Run these commands from the base directory.

Use the full hostname of your MobileFirst Platform server. SSL will break if the full hostname is not provided or if an IP address is used as the hostname.

```
REM mkdir server
```

Create the RSA key pair and generate a CSR

```
REM openssl genrsa -des3 -out server\server_key.pem -passout pass:passServer 2048
openssl req -new -key server\server_key.pem -out server\server.csr -config openssl.cnf -subj "/CN=%HOSTNAME%" -passin pass:passServer
```

Sign the CSR with the signing CA

```
REM openssl ca -in server\server.csr -out server\server.crt -keyfile signingca\signing_ca_key.pem -cert signingca\signing_ca.crt -config openssl.cnf -name signign_authority_ca_config -extensions server_identity -md sha512 -days 365 -passin pass:passSigning
```

Create certificate chain for the server certificate

Using the terminal, send a full certificate chain all the way up to the trust anchor (root CA) for iOS and Android environments. You can concatenate the certificate files to the trust anchor (root CA).

Create a chain for the signing CA

```
cat signingca/signing_ca.crt rootca/root_ca.crt > signing_ca_chain.crt
```

Create a chain for the server certificate

```
cat server/server.crt signingca/signing_ca.crt rootca/root_ca.crt > server_chain.crt
```

For Windows, send a full certificate chain all the way up to the trust anchor (root CA) for Windows environments. You can concatenate the certificate files to the trust anchor (root CA).

Create a chain for the signing CA

```
copy rootca\root_ca.crt+signingca\signing_ca.crt signing_ca_chain.crt
```

Create a chain for the server certificate

```
copy rootca\root_ca.crt+signingca\signing_ca.crt+server\server.crt server_chain.crt
```

Export a PKCS12 file for the signing CA

Export the private key and certificate for the signing CA into a `.p12` keystore file so that the embedded PKI can sign the user certificates with the signing CA.

```
openssl pkcs12 -export -in signingca/signing_ca.crt -inkey signingca/signing_ca_key.pem -out signingca/signing_ca.p12 -passin pass:passSigning -passout pass:passSigningP12
```

Export a PKCS12 file for the server certificate

Export the private key and certificate for the server into a `.p12` keystore file so that the server can send the client a valid server certificate.

```
openssle pkcs12 -export -in server_chain.crt -inkey server/server_key.pem -out server/server.p12 -passout pass:passServerP12 -passin pass:passServer
```

Configure IBM WebSphere Application Server Liberty profile (Liberty)

Enable the `ssl-1.0` and `appSecurity-2.0` features in the `server.xml` file:

```
<featureManager>
  <feature>ssl-1.0</feature>
  <feature>appSecurity-2.0</feature
>
</featureManager>
```

Liberty requires setting up the keystore and truststore to establish trust for the generated client certificates. Click here (http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/index.jsp?topic=/com.i bm.websphere.wlp.nd.doc/ae/rwlp_ssl.html) for more information.

- Set up your server's keystore to use the `.p12` file that was generated earlier (`server.p12`).
- Set up your truststore to use the `.p12` file that was generated earlier (`signing_ca.p12`).
- Configure your server's HTTP endpoint and allow (but not require) client-side certificates. This configuration is available by using the `clientAuthenticationSupported="true"` property in the Liberty SSL element.

The following example shows the updated SSL configuration:

```
<!-- default SSL configuration is defaultSSLSettings -->
<sslDefault sslRef="defaultSSLSettings"/>
<ssl clientAuthenticationSupported="true" id="defaultSSLSettings" keyStoreRef="defaultKeyStore" trustStoreRef="
defaultTrustStore"/>
<keyStore id="defaultKeyStore" location="server.p12" password="passServerP12" type="PKCS12" />
<keyStore id="defaultTrustStore" location="signing_ca.p12" password="passSigningP12" type="PKCS12"/>
```

Configure authenticationConfig.xml

Uncomment the UserCertificate Login Module section of the `authenticationConfig.xml` file, as shown below.

```
<!-- Login Module for User Certificate Authentication -->
<loginModule name="WLUSERCertificateLoginModule">
  <className>com.worklight.core.auth.ext.UserCertificateLoginModule</className
>
</loginModule>
```

Uncomment the `wl_userCertificateAuthRealm` section, as shown below.

```
<!-- Login Module for User Certificate Authentication -->
<realm name="wl_userCertificateAuthRealm" loginModule="WLUserCertificateLoginModule">
  <className>com.worklight.core.auth.ext.UserCertificateAuthenticator</className>
  <parameter name="dependent-user-auth-realm" value="SampleAppRealm"/>
  <parameter name="pki-bridge-class" value="com.worklight.core.auth.ext.UserCertificateEmbeddedPKI"/
>
</realm>
```

Update the value of the `embedded-pki-bridge-ca-p12-file-path` element to the full path of your signing CA `.p12` file.

Update the value of the `embedded-pki-bridge-ca-p12-password` element to the password (`passSigningP12`) that was used to create the `.p12` file.

Update the value of the `dependent-user-auth-realm` to the dependent realm you want to use (`SampleAppRealm`).

The realm name(`wl_userCertificateAuthRealm`) cannot be changed.

The following examples shows the updates made above.

```
<!--For User Certificate Authentication -->
<realm name="wl_userCertificateAuthRealm" loginModule="WLUserCertificateLoginModule">
  <className>com.worklight.core.auth.ext.UserCertificateAuthenticator</className>
  <parameter name="dependent-user-auth-realm" value="SampleAppRealm"/>
  <parameter name="pki-bridge-class" value="com.worklight.core.auth.ext.UserCertificateEmbeddedPKI"/>
  <parameter name="embedded-pki-bridge-ca-p12-file-path" value="YOUR_BASE_DIRECTORY/signing_ca.p12"/
>
  <parameter name="embedded-pki-bridge-ca-p12-password" calue="passSigningP12"/>
</realm>
```

Define a security test that uses `wl_userCertificateAuthRealm`.

```
<!--For User Certificate Authentication -->
<customSecurityTest name="customx509Tests">
  <test realm="wl_antiXSRFRealm" step="1"/>
  <test realm="wl_authenticityRealm" step="1"/>
  <test realm="wl_directUpdateRealm" mode="perSession" step="1"/>
  <test realm="wl_userCertificateAuthRealm" isInternalUserID="true" step="1"/>
  <test realm="wl_deviceNoProvisioningRealm" isInternalUserID="true" step="2"/
>
</customSecurityTest>
```

Configure application-descriptor.xml

Ensure that you added iOS or Android environments to your MobileFirst Platform application

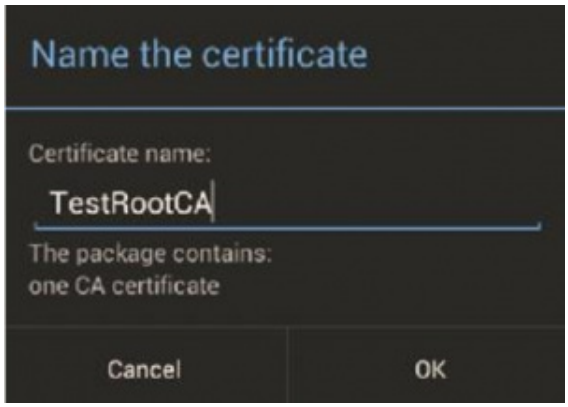
Protect your application or environment with your custom security test.

```
<!--For User Certificate Authentication -->
<android securityTest="customx509Tests" version="1.0">
<iPhone bundleId="com.SampleApp" securityTest="customx509Tests" version="1.0">
```

Build and deploy your application and adapters to the MobileFirst Platform server.

Install root CA on iOS and Android

You must install the root CA that you generated in the previous steps onto your client devices for your devices to trust your MobileFirst Platform server over SSL. Email or host the `root_ca.crt` file, and then open the file on your device. The iOS and Android devices ask for approval when you manually attempt to install certificates.



Install application and test

1. Deploy your application to the MobileFirst Platform server
    - Run as > Run on
2. Update the deploy target for HTTPS.
    - Run as > Build Settings and Deploy Target...
    - Mark the check box "Build the application to work with a different MobileFirst Platform Server."
    - Enter the server HTTPS address. `https://:`
    - Enter the context path. `/`;
3. Build the application with the updated deploy target.
    - Run as > Build All Environments
4. Run the application on the specified environments.

Sample application

Click to download (http://public.dhe.ibm.com/software/products/en/MobileFirstPlatform/docs/v700/UserCertificateAuthenticationProject.zip) the Studio project.

To confirm a successful configuration, ensure that you see a log-in form the first time that you try to access a protected resource. If `WL.Client.connect()` is uncommented in the `main.js` file, the log-in form is displayed when the application starts. Otherwise, `WL.Client.connect()` must be invoked before you call an adapter procedure to see a log-in form after the adapter is called.

After you log in through the dependent realm, a successful response from the adapter invocation indicates that the user was successfully enrolled.

On subsequent connections to the server, you are no longer asked to log in and the adapter calls continue to return successfully.

> For more information review the User certificate authentication (http://www-01.ibm.com/support/knowledgecenter/SSHS8R_6.3.0/com.ibm.worklight.monitor.doc/monitor/c_user_CA.html?lang=en) topic in the user documentation