Oracle Banking APIs

Security Guide
Release 19.1.0.0.0

Part No. F18559-01

May 2019

ORACLE®

Security Guide

May 2019

Oracle Financial Services Software Limited

Oracle Park

Off Western Express Highway

Goregaon (East)

Mumbai, Maharashtra 400 063

India

Worldwide Inquiries:

Phone: +91 22 6718 3000

Fax:+91 22 6718 3001

www.oracle.com/financialservices/

Table of Contents

1. Preface

This document provides a comprehensive overview of security for Oracle Banking API. It includes conceptual information about security principles, descriptions of the product's security features, and procedural information that explains how to use those features to secure Oracle Banking API.

This preface contains the following topics:

- Audience
- Documentation Accessibility
- Access to Oracle Support
- Related Documents

1.1 Audience

This Security Guide is intended for Bank IT Staff responsible for application installation and security configuration as well as Product Implementation teams.

1.2 Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

1.3 Access to OFSS Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit

http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit

http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if your hearing is impaired.

1.4 Related Information Sources

For more information, see the following documentation:

Hardening Tips for Default Installation of Oracle Enterprise Linux 6 at
https://docs.oracle.com/cd/E37670_01/E36387/E36387.pdf

Oracle® Fusion Middleware Installation Guide for Oracle WebLogic Server at
http://docs.oracle.com/cd/E17904_01/doc.1111/e14142/toc.htm

Oracle® Fusion Middleware Application Security Guide - Configuring and Managing Auditing at
http://docs.oracle.com/cd/E23943_01/core.1111/e10043/audpolicy.htm

For installation and configuration information, see the Oracle Banking APIs Installation Guide

For the complete list of Oracle Banking licensed products and the Third Party licenses included with the license, see the Oracle Banking Licensing Guide.

2. General Security Principles

The following principles are fundamental for using any application securely.

2.1 Restrict Network Access to Critical Services

Keep both the Oracle Banking API middle-tier and the database behind a firewall. In addition, place a firewall between the middle-tier and the database. The firewalls provide assurance that access to these systems is restricted to a known network route, which can be monitored and restricted, if necessary. As an alternative, a firewall router substitutes for multiple, independent firewalls.

If firewalls cannot be used, be certain to configure the TNS Listener Valid Node Checking feature which restricts access based upon IP address. Restricting database access by IP address often causes application client or server programs to fail for DHCP clients. To resolve this, consider using static IP addresses, a software or a hardware VPN or Windows Terminal Services or its equivalent.

2.2 Follow the Principle of Least Privilege

The principle of least privilege states that users should be given the least amount of privilege to perform their jobs. User privileges should be reviewed periodically to determine relevance to current job responsibilities.

2.3 Monitor System Activity

System security largely depends on the following practices:

- Good security protocols
- Proper system configuration
- System monitoring

The system needs to be constantly monitored from a monitoring tool.

2.4 Keep Up To Date on Latest Security Information

Oracle continually improves its software and documentation. It is recommended to keep your software updated.

Home

3. Secure Installation and Configuration

This chapter provides an overview of the architecture of the deployment and describes the installation and configuration procedure for Oracle Banking API.

Please note that this is only a guide to securing the Oracle Banking API application and does not replace periodic reviews of the security architecture of the entire ecosystem of multiple applications maintained by the customer. The guidance provided in this document must always be augmented by specific understanding of the security considerations of the specific deployment architecture.

3.1 Architecture Diagram
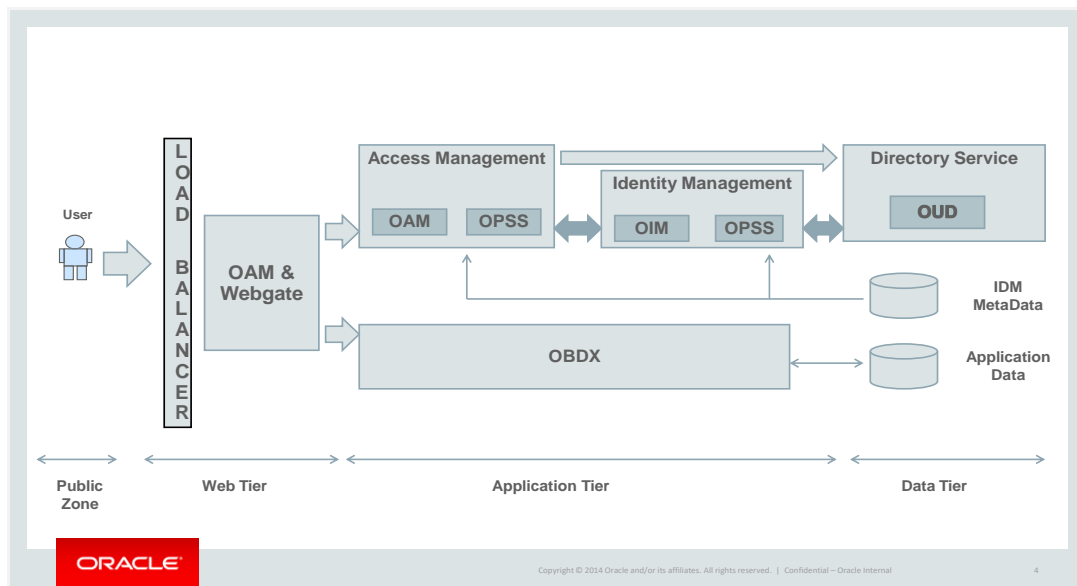
3.2 Installing WebLogic

Installation of WebLogic Server can be done by referring to the documentation published at https://docs.oracle.com/cd/E24329_01/doc.1211/e24492/toc.htm.

3.3 Configuring SSL

One way SSL between the presentation tier and the application on WebLogic server is supported. The detailed configuration is explained below:

Note: Procure an external CA signed certificate before proceeding further. Follow the instructions below to install the certificate once the certificate is available

1. Import the Certificate into a Java Trust Keystore

   Execute the following command:

   ```
   keytool -import -trustcacerts -alias sampletrustself -keystore
   SampleTrust.jks -file SampleSelfCA.cer.der -keyalg RSA
   keytool -import -alias `hostname -f` -file `hostname -f`.cer -keystore
   <JAVA_HOME>/jre/lib/security/cacerts -storepass changeit -noprompt
   ```

2. Configure Application Domain's WebLogic with Custom Identity and Trust Keystores

   a. Open the WebLogic admin console and navigate to
      Home --> Summary of Servers --> AdminServer.

   b. Click the **Keystores** tab.
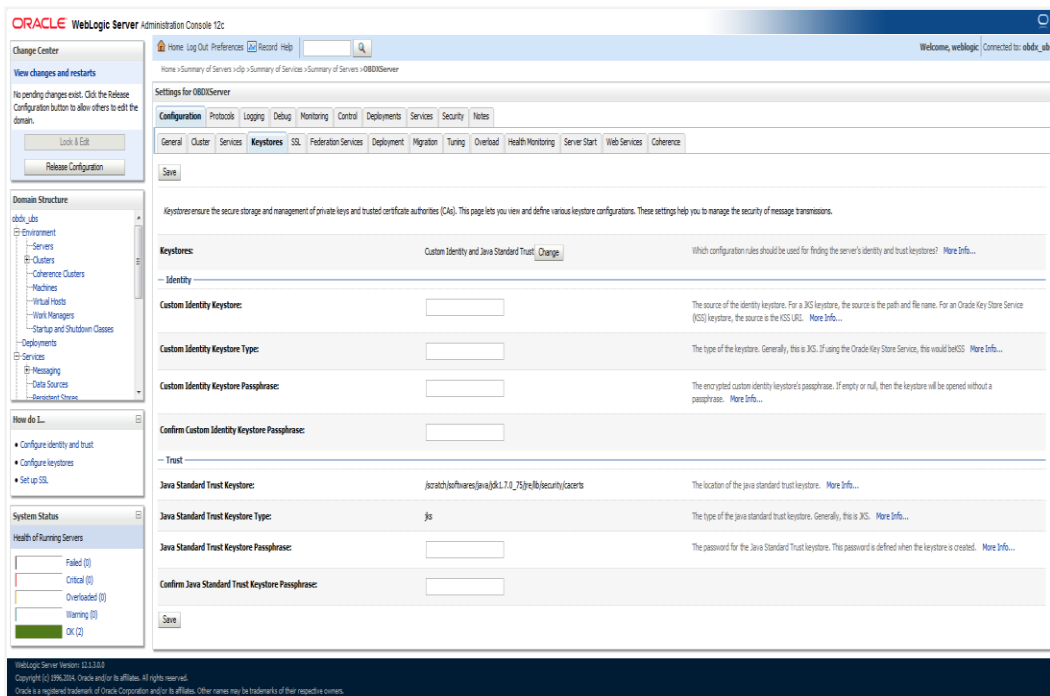
- Click the **Change** button.
- Select Custom Identity and Java Standard Trust option from the list.
- Click the **Save** button.
- Enter the following details in the **Identity** and **Trust** sections:

Field	Value
Custom Identity Keystore	Absolute path of the custom keystore
Custom Identity KeyStore Type	JCEKS
Custom Identity KeyStore Passphrase	<Passphrase>

Field	Value
Confirm Custom Identity KeyStore Passphrase	<Re-enter the same Passphrase>

Enter the passphrases that were used while creating the custom Identity Keystore and certificate.

    c.   Click the **Save** button.

    d.   Click the **SSL** Tab.

Enter the following details in the **Identity** section:

Field	Value
Private Key Alias	<Alias>
Private Key Passphrase	<Passphrase>
Confirm Private Key Passphrase	<Re-enter passphrase>

    i.   Enter the passphrases that were used while creating the certificate.

    ii.   Click the **Save** button.

    iii.   Click the **Advanced** link.

    iv.   Ensure that **Two Way Client Cert Behavior** is set to **Client Certs Not Requested**.

    e.   Click the **General** tab.

    f.   Select the **SSL Listen Port Enabled** check box.

> g. Click the **Save** button.

3.4 Disable SSLv3

By default, SSLv3 should be disabled.

Specifying the `weblogic.security.SSL.protocolVersion` system property in a command-line argument that starts the WebLogic Server lets you specify the protocol that is used for SSL connections.

The following command-line arguments can be specified so that WebLogic Server supports only TLS connections:

```
-Dweblogic.security.SSL.protocolVersion=TLS1
```

Note: If you don't specify the above property, WebLogic assumes SSLv3 by default.

3.5   HTTP Response Header Configurations

The following are some HTTP Response Headers that mitigate certain vulnerabilities.

Vulnerability	HTTP Response Header
Clickjacking	`X-Frame-Options`
XSS	`Content-Security-Policy`
	`X-XSS-Protection`
Cookie hijacking Protocol Downgrade attacks	`Strict-Transport-Security`
Retrieving Sensitive data from browser cache	`Cache-Control`

The sections below specify how to configure these response headers in the `httpd.conf` file of the web server.

3.5.1   X-Frame-Options

```
Header always append X-Frame-Options SAMEORIGIN
```

3.5.2   Content-Security-Policy

```
Header set Content-Security-Policy "default-src 'none'; img-src 'self';
script-src 'self' 'unsafe-inline' 'unsafe-eval'; style-src 'self'
https://fonts.googleapis.com 'unsafe-inline'; object-src 'none'; frame-src
'none'; font-src 'self' https://fonts.gstatic.com; connect-src 'self'
http://<OAM Server>:<OAM Port>/; child-src 'self'"
```

Please note that the policy mentioned here is for the base product. If the product gets customized and content from different URLs needs to be allowed to be executed by the browser, then this policy will have to be modified accordingly.

3.5.3   X-XSS-Protection

```
Header set X-XSS-Protection "1; mode=block"
```

3.5.4   Strict-Transport-Security

Set this for your top level domain. The header directive needs to be included inside the VirtualHost directive

```
<VirtualHost *:443>
   Header always set Strict-Transport-Security "max-
age=31540000; includeSubDomains"
</VirtualHost>
```

Consider submitting your website to be included in the HSTS preload list of websites maintained by Google Chrome at https://hstspreload.appspot.com/.
Other browsers like MS IE 11, MS Edge, Firefox and Opera also refer to this list maintained by Google and therefore the security offered by this mechanism will extend to other browsers too.

3.5.5 Cache-Control

```
Header set Cache-Control "max-age=0, no-cache, no-store, must-revalidate"
Header set Pragma "no-cache"
Header set Expires 0
```

3.6 Password Policy Guidelines

Our recommendations for setting a password policy are in line with the latest recommendations from NIST as of June 2018.

1. The minimum length of a password must be at least 8 characters. You can choose to increase this number to 10 or 12.

2. The maximum length of a password must be at least 64 characters. You can choose to increase this number to 80 or 100.

3. Do not cause passwords to expire without reason. A password must be expired only when the user has forgotten it and has requested a reset.

4. Allow all printable ASCII characters, including spaces, and accept all UNICODE characters too.

5. Do not force the user to use a combination of upper case characters, lower case characters, numbers and special characters.
   Instead recommend to him that he uses "passphrases" instead of passwords, and that's the reason why the recommended minimum length must be at least 8 and the maximum length must be at least 64.
   Passphrases are sentences like "*Wow, I like the freedom to choose this password!!*" (yes, with spaces, a comma and exclamation marks in it)

3.7 Configuring 2FA for login

Oracle Banking API supports a 2nd factor of authentication during login.
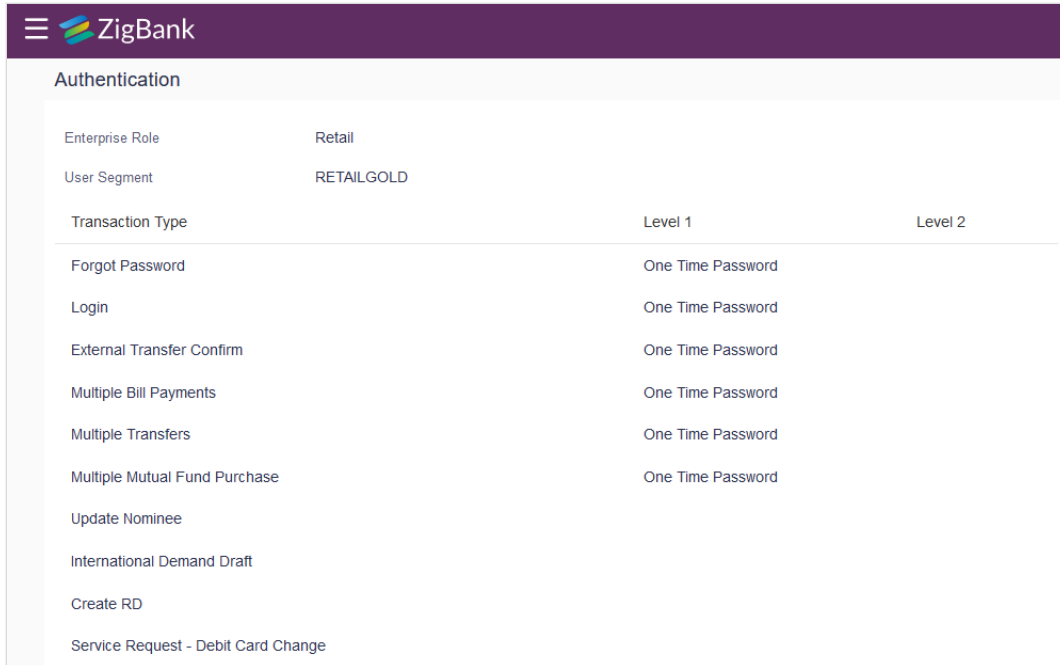
1. Login as the Admin user

2. Click on "Authentication"
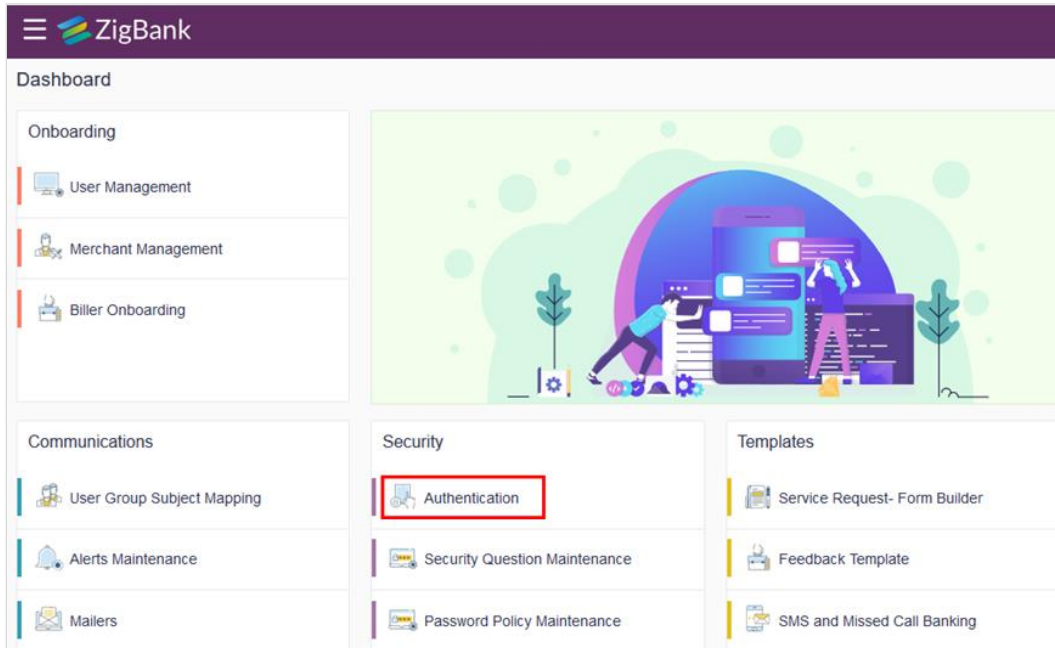
3. Choose the user segment for which you want to configure 2FA for login and click on "View".
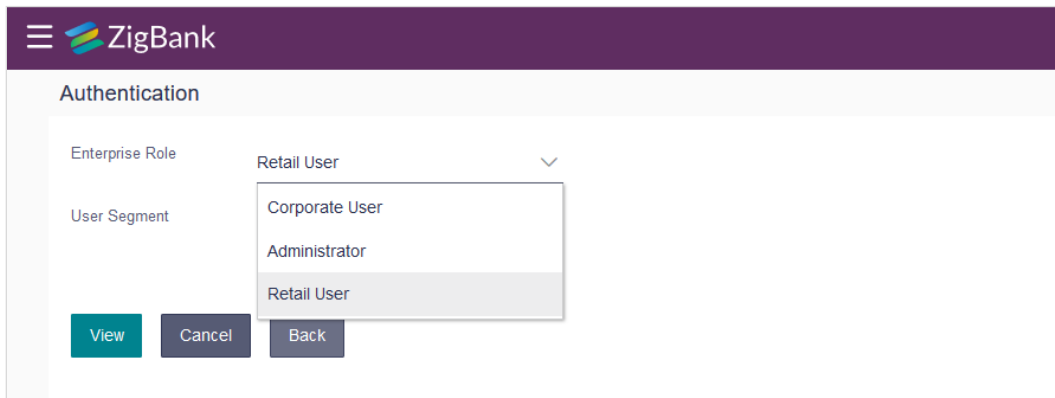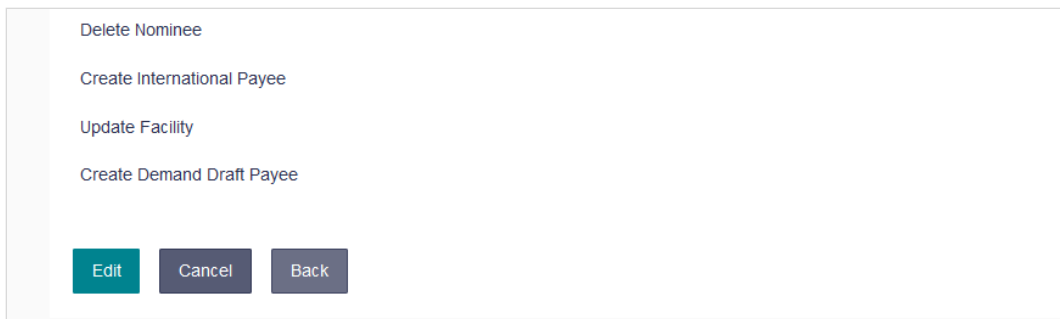
4. You will see the following screen where you can configure 2FA for virtually every transaction, including Login.

5. Click on the "Edit" button at the bottom of the screen.

6. You can now configure up to 2 factors (levels) of authentication / re-authorization. However please note that the system will not let you set "Security Questions" as a factor of authentication / re-authorization for the Login transaction. You will have to choose either OTP or Soft Token.

7. Click on the "Save" button at the bottom of the screen, followed by the "Confirm" button seen in the subsequent verification screen.

3.8 Configuring 2FA attributes

This section covers some key attributes of the 2nd factor of authentication (re-authorization). Attributes like the maximum number of times a user is allowed to hit the "Resend" button after an OTP is generated, the pool of security questions etc are a couple of examples of 2FA attributes.

These attributes are seen in the database in the PROP_ID column of the table DIGX_FW_CONFIG_ALL_B (CATEGORY_ID = 'authenticationConfig'). The following table lists down all possible attributes and their significance. Their values must be set in the column PROP_VALUE.

PROP_ID	SIGNIFICANCE
OTP.EXPIRATION_TIME	Time in milliseconds after which an OTP will expire.
T_SOFT_TOKEN.EXPIRATION_TIME	Time in milliseconds after which a time based soft token will expire.
R_SOFT_TOKEN.EXPIRATION_TIME	Time in milliseconds after which a random soft token will expire.
SEC_QUE.EXPIRATION_TIME	Time in milliseconds after which answers to the security questions presented to the user will no longer be considered for re-authorization.

PROP_ID	SIGNIFICANCE
EXPIRATION_TIME	Time in milliseconds after which the re-authorization factor will expire. This is the default property that will be looked up in case factor specific expiration times are not maintained.
OTP.MAX_NO_ATTEMPTS	Max number of unsuccessful attempts of entering a valid OTP after which all 2FA enabled transactions for the user will be locked for a "cooling period" amount of time.
T_SOFT_TOKEN.MAX_NO_ATTEMPTS	Max number of unsuccessful attempts of entering a valid time based soft token after which all 2FA enabled transactions for the user will be locked for a "cooling period" amount of time.
R_SOFT_TOKEN.MAX_NO_ATTEMPTS	Max number of unsuccessful attempts of entering a valid random soft token after which all 2FA enabled transactions for the user will be locked for a "cooling period" amount of time.
SEC_QUE.MAX_NO_ATTEMPTS	Max number of unsuccessful attempts of entering valid answers to security questions after which all 2FA enabled transactions for the user will be locked for a "cooling period" amount of time.
MAX_NO_ATTEMPTS	Max number of unsuccessful attempts of entering valid 2FA after which all 2FA enabled transactions for the user will be locked for a "cooling period" amount of time. This is the default property that will be looked up in case factor specific Max Attempts are not maintained.
TFA_LOCK_COOLING_PERIOD	This is the cooling period in milliseconds after which 2FA transactions which were locked out because of exceeding MAX_NO_ATTEMPTS, are enabled once again.

PROP_ID	SIGNIFICANCE
OTP.MAX_ACTIVE_REF_NO	Max number of attempts to generate 2FA reference numbers for a transaction after which no more attempts can be made for EXPIRATION_TIME units of time for that factor of authentication. This one is specific to OTPs. This property is in place as a basic mechanism to protect the application against DOS attacks where the end user can keep generating OTPs by initiating transactions and making the system generate the 2nd factor of authentication, but not going through and completing the transaction.
T_SOFT_TOKEN.MAX_ACTIVE_REF_NO	Max number of attempts to generate 2FA reference numbers for a transaction after which no more attempts can be made for EXPIRATION_TIME units of time for that factor of authentication. This one is specific to Time Based Soft Tokens.
R_SOFT_TOKEN.MAX_ACTIVE_REF_NO	Max number of attempts to generate 2FA reference numbers for a transaction after which no more attempts can be made for EXPIRATION_TIME units of time for that factor of authentication. This one is specific to Random Soft Tokens.
SEC_QUE.MAX_ACTIVE_REF_NO	Max number of attempts to generate 2FA reference numbers for a transaction after which no more attempts can be made for EXPIRATION_TIME units of time for that factor of authentication. This one is specific to Security Questions.
MAX_ACTIVE_REF_NO	Max number of attempts to generate 2FA reference numbers for a transaction after which no more attempts can be made for EXPIRATION_TIME units of time for that factor of authentication. This is the default property that will be looked up in case factor specific Max Active Reference Number attempts are not maintained.
OTP.RESEND_COUNT	Max number of times a user can hit the "Resend" button in case of OTPs. After exceeding this count, the user will need to re-initiate the transaction all over again.
retailuser.NO_QUE_ANS	Number of security questions that a retail user needs to setup (answer). During an actual transaction he will be asked a sub set of these questions.

PROP_ID	SIGNIFICANCE
corporateuser.NO_QUE_ANS	Number of security questions that a corporate user needs to setup (answer). During an actual transaction he will be asked a sub set of these questions.
administrator.NO_QUE_ANS	Number of security questions that an admin user needs to setup (answer). During an actual transaction he will be asked a sub set of these questions.
NO_QUE_ANS	Number of security questions that a user segment needs to setup (answer). During an actual transaction he will be asked a sub set of these questions. This is the default property that will be looked up in case factor specific NO_QUES_ANS is not maintained

3.9   Choosing a non blocking PRNG

OBAPI uses Java's random number generation capabilities internally. However the out of the box algorithm for PRNG configured in the JDK can block the thread after a certain time if there isn't enough randomness available. This is because the default configuration uses `/dev/random` on Linux for PRNG.

Therefore we recommend that you navigate to `<JDK_HOME>/jre/lib/security` and edit the `java.security` file. Comment out the old property and change its value as shown below

```
#securerandom.strongAlgorithms=NativePRNGBlocking:SUN
securerandom.strongAlgorithms=NativePRNGNonBlocking:SUN
```

This will ensure that the application uses /dev/urandom for PRNG.

Needless to say, make sure you make this change in the JDK that your WebLogic server is going to be using.

3.10 Generating Security Keys

Oracle Banking API supports generating Security Keys required for encryption of sensitive information. This is an optional step and is applicable only if a UI has been developed to interact with OBAPI directly. The presence of the keys provides an option to encrypt the login credentials on the client.

3.10.1 Generating Public and Private Key Pair

Oracle Banking API supports generating Public and Private Key pair that will be used for encryption of login password on the User Interface.

1. Login as the Admin user

2. Click on Menu item "Security" → "Security Keys"
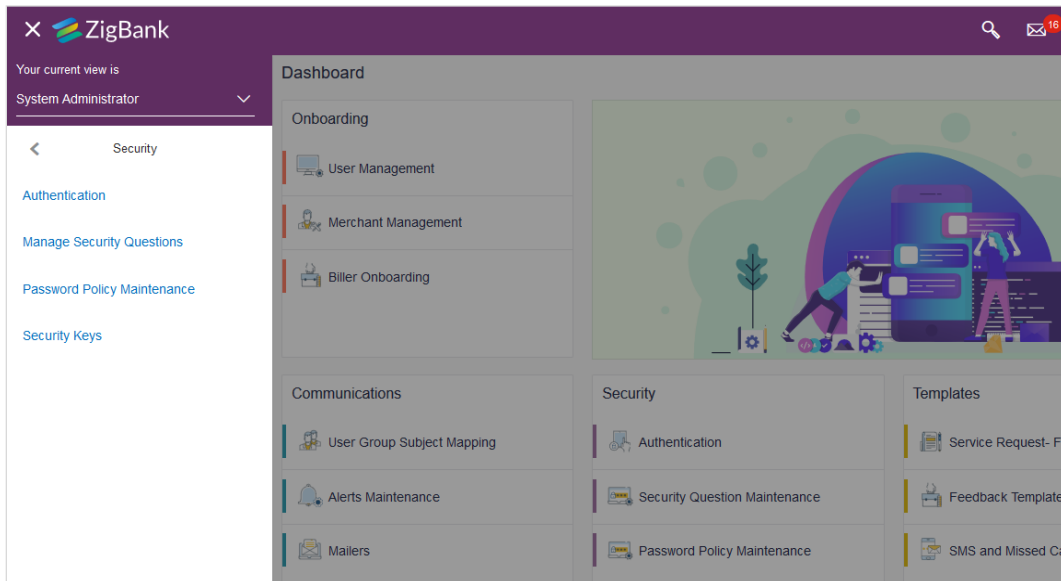
3. Click on "Generate" for new Public and Private Key Pair generation used for encryption.
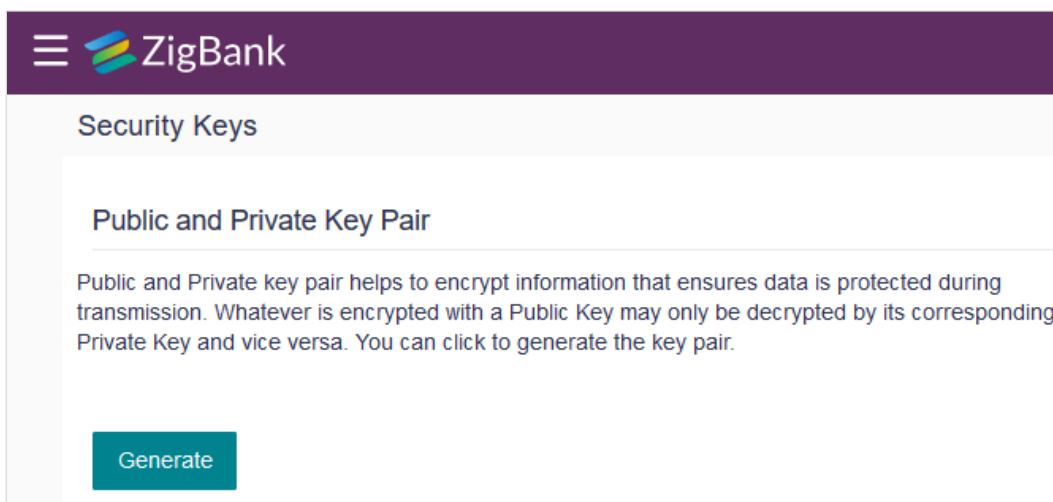
4. Restart the WebLogic server for utilizing the above generated key pair.

By default the Public and Private key pair is not generated and the password is not encrypted on the User Interface. Once the Key Pair is generated, encryption will be effective after server restart.

In case of Private key compromise, an Administrator can generate a new Key Pair to mitigate the impact of compromised key.

Home

4. Guidance for Implementation Teams

4.1 CSRF Mitigation – Generating Nonces

A nonce is a pseudo random number that may be used only once. If a nonce is sent across in every request from the client to the server and the server validates the sent nonce every single time, then it mitigates the risk of Cross Site Request Forgery (CSRF).

The product provides a REST Service to generate nonces – each nonce can be used only once to identify each request uniquely, for each session. The product also has an inbuilt framework that will validate the nonce sent in the request.

Therefore post a successful login you need to make a call to `https://<Host>:<Port>/digx/v1/session/nonce` before you make a call to any other service. This service will return back an array of nonces in the response header. You can pick up any one nonce from the array and use it to send across the nonce required in a subsequent request. A nonce can be used only once. You need to discard it after usage.

Please note that unless you send across a nonce, the services that are accessed post login will not work.

4.2 Indirect Object Reference Implementation

4.2.1 What it means

It is a good security practice to hide sensitive data objects from the end user. Although the system needs to play around with sensitive data objects, it is recommended to refer to these sensitive data objects via pointers – tokens that temporarily point to the sensitive data objects but themselves do not contain any sensitive data.

For example consider a credit card application on the web which offers the following 2 transactions:

> ➢ Credit Cards Summary – Displays a list of all credit cards the user owns.

> ➢ Credit Card Details – Displays the details of one specific Credit Card that the user selects

The Credit Cards Summary page will typically list all credit card numbers in a masked format. Let's assume that the end user holds 2 Credit Cards C1 and C2. When the end user hits the Summary link, the server returns back the following in its response:

a. Masked Credit Card Number C1 (visible to the user)

b. Masked Credit Card Number C2 (visible to the user)

c. Token T1 (not visible to the user)

d. Token T2 (not visible to the user)

T1 and T2 are random tokens – difficult to guess – which the server has generated as proxies for C1 and C2 respectively. The server has internally stored this mapping of C1-T1 and C2-T2 somewhere. Please note that T1 and T2 are tied to the current session. The moment the session expires, T1 and T2 get discarded. Next time the user logs in, the server generates different tokens T1x and T2x for C1 and C2 respectively.

Whenever the user clicks on say Credit Card Details for C1, the client sends T1 to the server instead of C1, as a request parameter. The server internally figures out that the request is actually for C1 and processes the request accordingly.

Thus we refer to sensitive data indirectly via tokens that are generated with different values for every session.

4.2.2 How OBAPI supports it

To implement the above mechanism the framework offers interception of both the request and the response. The recommendation is to apply indirect referencing to sensitive data fields like Personally Identifiable Information fields aka PII data.

For the interception to work automatically, the sensitive fields holding the PII data must be defined as a Java type, which extends the abstract class

`com.ofss.digx.datatype.complex.MaskedIndirectedObject.`

The abstract class exposes 2 data fields, namely `value` and `displayValue`. The `value` field holds the indirect reference value, which is used during data transmission from the client to the server. The `displayValue` field holds the masked value of the data.

The following data types are supported out-of-box:

1. com.ofss.digx.datatype.complex.Account - Account number

2. `com.ofss.digx.datatype.complex.Applicant` – The unique identifier to identify an applicant. Typically a party ID.

3. `com.ofss.digx.datatype.complex.ApplicationId` – The unique identifier of an application for account opening.

4. `com.ofss.digx.datatype.complex.ContentId` – The unique identifier for content such as documents for a party.

5. com.ofss.digx.datatype.complex.CreditCard – Credit Card Number

6. com.ofss.digx.datatype.complex.DebitCard – Debit Card Number

7. com.ofss.digx.datatype.complex.Email – Email ID

8. `com.ofss.digx.datatype.complex.Party` – The unique identifier for a party.

9. com.ofss.digx.datatype.complex.PhoneNumber – Phone Number

10. com.ofss.digx.datatype.complex.SSN – Social Security Number

11. `com.ofss.digx.datatype.complex.SubmissionId` – The unique identifier for a submission containing 1 or more applications for account opening.

To modify the existing/base product-masking pattern for any of the above data types, the following entries need to be copied/cloned from the table DIGX_FW_CONFIG_ALL_B to the table DIGX_FW_CONFIG_ALL_O and then modified as required in DIGX_FW_CONFIG_ALL_O.

Note: Please DO NOT MODIFY these entries IN DIGX_FW_CONFIG_ALL_B.

Data Type	Category ID / Preference Name	Property ID
Account	MaskingPattern	AccountNumberMasking
Applicant	MaskingPattern	ApplicantIdMasking
ApplicationId	MaskingPattern	ApplicationIdIdMasking
ContentId	MaskingPattern	ContentIdMaskingPattern

Data Type	Category ID / Preference Name	Property ID
CreditCard	MaskingPattern	CreditCardNumberMasking
DebitCard	MaskingPattern	DebitCardNumberMasking
Email	MaskingPattern	EmailIdMasking
Party	MaskingPattern	PartyIdMasking
PhoneNumber	MaskingPattern	PhoneNumberMasking
SSN	MaskingPattern	SSNMasking
SubmissionId	MaskingPattern	SubmissionIdMaskingPattern

The characters allowed in the making pattern are as below:

N – Keeps the character transparent. Does not mask.

Any other character – Replaces the character at the location with the character specified.

For example: XXXXXNNNN will keep the last 4 characters in clear text and mask the first 5 characters using the character 'X'.

4.3   Output Encoding

To mitigate inline Cross Site Scripting attacks, the product provides a framework to encode the data sent in the response. In OBAPI 18.3.0.0.0 this is something that is handled implicitly in the framework for all services, base as well as any custom services that you might write.

There is nothing that you need to do explicitly to achieve this.

4.4   Implementing a custom Cryptography Provider

The base product provides a symmetric key cryptography framework that enables the implementation team to implement its own custom symmetric key encryption/decryption mechanism.

The product is shipped out with an out of the box Cryptography Provider that will be invoked if no custom implementation is found.

If you wish to write your own custom Cryptography Provider, the required steps are as follows:

1. Write the custom cryptography provider class such that it implements the interface `com.ofss.digx.infra.crypto.spi.ICryptographyProvider`

   The interface defines methods as shown below:

```
package com.ofss.digx.infra.crypto.spi;

import javax.crypto.SecretKey;

/**
 * The interface to be implemented by any symmetric cryptography provider. Provides contract methods for encrypting,
 * decrypting and generating a random secret key.
 */
public interface ICryptographyProvider {
    * Fetches the algorithm supported by the implementing cryptography provider.
    public String getAlgorithm();

    * Fetches the version number of the cryptography provider. This number is used to identify the latest cryptography
    public int getVersion();

    * Encrypts the data provided with the secret key provided. Returns a HEX encoded representation of the encrypted
    public String encrypt(String data, SecretKey key) throws CryptographyException;

    * Decrypts the data provided with the secret key provided. Returns a @{code java.lang.String} representation of the
    public String decrypt(String data, SecretKey key) throws CryptographyException;

    * Generates and returns a random secret which can be used for encrypting data. Care <i><b>MUST</b></i> be taken by
    public SecretKey generateRandomSecretKey();

    * Fetches the key length used by the cipher for encrypting the data.
    public Integer getCipherKeyLength();
}
```

2. Implement the `encrypt()` and `decrypt()` methods to encrypt and decrypt the data passed to the methods, using the key passed along with the data.

3. Implement the `getVersion()` method to simply return a number greater than 1.

   For example:

```java
public int getVersion() {
    return 2;
}
```

4. Implement the method to simply return the symmetric encryption algorithm name.
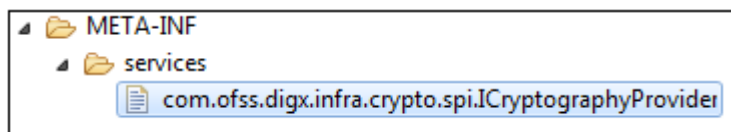
   For example:

```java
public String getAlgorithm() {
    return "AES";
}
```

5. You can leave the implementations of the other two methods blank.

6. In the custom jar file that contains the custom Cryptography Provider, create a file with the name `com.ofss.digx.infra.crypto.spi.ICryptographyProvider` under the folder structure META-INF/services

```
▲ 🗁 META-INF
   ▲ 🗁 services
        📄 com.ofss.digx.infra.crypto.spi.ICryptographyProvider
```

7. Open the file for editing.

8. Type out the fully qualified class name of your custom Cryptography Provider class. For example:
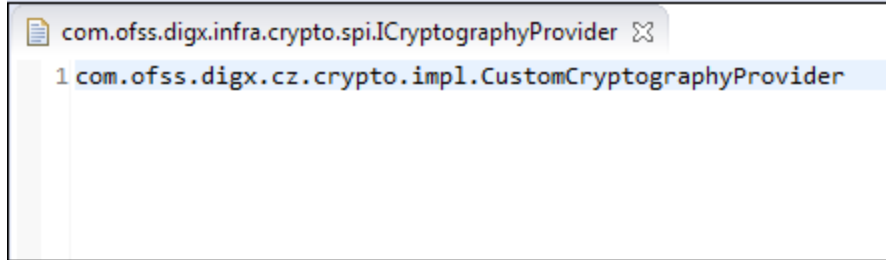
Deploy your custom jar on the WebLogic server and you should be all set.

The Cryptography Provider will be invoked when the system invokes the encryption framework for the following actions:

9. Encrypting the SMTP Server password entered by the OBAPI Admin, before storing it in the Database.

10. Encrypting the system generated password for first time users, before storing it in the Database. This is will be relevant to the function of Password Printing.

11. Encrypting the Identification Number for the selected Identification Type (Driver's License, Passport etc) during originating a loan application online.

4.5 Implementing a custom 2FA mechanism

1. You will need to write your own Java class to implement your own custom factor of authentication.

2. The class must be registered in the table DIGX_AU_AUTH_TYPE_MST. Choose a custom ID.

3. The custom class must implement the interface

```
com.ofss.digx.framework.security.authentication.provider.I2FactorAut
henticationProvider
```

4. To configure your custom authenticator as an additional option available to the admin during the 2FA configuration of transactions, set the custom ID used in Step 2 in the table DIGX_FW_CONFIG_VAR_B

5. The configuration already seen in the above image suggests that an admin will have the option of setting one of OTP, Soft Token and Security Questions as an additional factor of authentication when configuring 2FA for user segments Retail, Corporate and Administrator.

6. The PROP_ID that the system must look up in this table (DIGX_FW_CONFIG_VAR_B) is maintained in the table DIGX_FW_CONFIG_ALL_B against the PROP_ID SUPPORTED_AUTH_TYPE.

7. If ${_PROPERTY_} is the value maintained against retailuser.SUPPORTED_AUTH_TYPE in the table DIGX_FW_CONFIG_ALL_B, then for retail users the application will look up the table DIGX_FW_CONFIG_VAR_B where PROP_ID = _PROPERTY_ to check what options are available to the admin.

```
select * from DIGX_FW_CONFIG_ALL_B where CATEGORY_ID='authenticationConfig'
AND PROP_ID LIKE '%SUPPORTED_AUTH_TYPE%';
```

	PROP_ID	CATEGORY_ID	PROP_VALUE
1	PC_CM_ME.SUPPORTED_AUTH_TYPE	authenticationConfig	${PC_CM_ME.SUPPORTED_AUTH_TYPE}
2	SUPPORTED_AUTH_TYPE	authenticationConfig	${SUPPORTED_AUTH_TYPE}
3	administrator.SUPPORTED_AUTH_TYPE	authenticationConfig	${administrator.SUPPORTED_AUTH_TYPE}
4	corporateuser.SUPPORTED_AUTH_TYPE	authenticationConfig	${corporateuser.SUPPORTED_AUTH_TYPE}
5	retailuser.SUPPORTED_AUTH_TYPE	authenticationConfig	${retailuser.SUPPORTED_AUTH_TYPE}

4.6 Configuring Password Printing Securely

Banks need to provide new customers with system-generated credentials to enable them to login into the system for the first time. Some of the banks prefer to print the first time password on paper and then hand it over to the customer in person.

To enable banks to do this, OBAPI has the "Print Password" function built out of the box. However, the base OBAPI product will not provide an end-to-end solution since password printing is not something universal.

For the sake of this explanation, we are going to break up the process of Password Printing into 6 steps:

- Generate the password using a secure random number generation mechanism.

- Encrypt the password.

- Store the password in the Database.

- Retrieve the password for printing.

- Decrypt the password.

- Do the actual printing.

Steps 2 and 5 can be customized, but not mandatory. Please refer to section **Implementing a custom Cryptography Provider**.

However, it is mandatory to implement Step 6. Here is how you can plug-in your implementation of printing the password:

- Write a custom class that will implement the interface com.ofss.digx.app.sms.user.printinformation.provider.IUserInformationPrintAdapter

- The interface defines a single method as shown below

```
package com.ofss.digx.app.sms.user.printinformation.provider;

import com.ofss.digx.app.sms.dto.user.printInformation.PasswordPrintInformationDTO;

public interface IUserInformationPrintAdapter {

    /**
    
    public void print(PasswordPrintInformationDTO userprintDTO) throws Exception;

}
```

- The DTO passed to the `print()` method will contain the password that is needed for printing.

- Also, add the following entry to the file Preferences.xml
  <Preference name="UserPrintConfig'"
  PreferencesProvider="com.ofss.digx.infra.config.impl.DBBasedPropertyProvider"
  parent="jdbcpreference" propertyFileName="select prop_id, prop_value from
  digx_fw_config_all_b where category_id = 'UserPrintConfig'" syncTimeInterval="36000000"
  />

- Run the following Database script

```
Insert into DIGX_FW_CONFIG_ALL_B (PROP_ID, CATEGORY_ID, PROP_VALUE,
FACTORY_SHIPPED_FLAG, PROP_COMMENTS, SUMMARY_TEXT, CREATED_BY,
CREATION_DATE, LAST_UPDATED_BY, LAST_UPDATED_DATE, OBJECT_STATUS,
OBJECT_VERSION_NUMBER) values
('USER_INFORMATION_PRINT_PROVIDER','UserPrintConfig','com.ofss.digx.app.sms.
user.printinformation.provider.CustomUserInformationPrintAdapter','N',null,'
Custom adapter for User Password Information
Printing','ofssuser',sysdate,'ofssuser',sysdate,'A',1);
```

Home