Oracle Fusion Middleware

App Containerization Tool Guide for Oracle Mobile Security Suite

Release 3.0

E51928-01

February 2014

Oracle Mobile Security Suite enhances employee productivity by allowing secure access to corporate apps and data from mobile devices while preserving rich user experience. Its Mobile Security Container creates the enterprise workspace on any mobile device, corporate owned or personal, and for all mobile platforms. Employees get seamless access to intranet, corporate data and apps with enterprise-grade security and deep integration with Windows Authentication for true Single Sign-On.

The Oracle Mobile Security Container is an important feature of Oracle Mobile Security Suite. The container isolates your enterprise access on personal devices, enabling corporate Bring Your Own Device (BYOD).

This document contains the following sections:

1 Containerization Tool Features

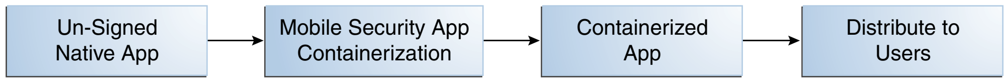
With Mobile Security App Containerization Tool, customers can add a standardized security layer to native mobile apps. The containerization process is simple and injects the following security services into your app.

- Secure data transport: An encrypted AppTunnel through Mobile Security Access Server to application back end resources behind an enterprise firewall.
- Authentication: Windows Integrated Authentication/SSO (Kerberos or NTLM) to application back end servers.
- Secure data storage: Encrypted storage of app data, including files, database, app cache and user preferences.
- Data leakage controls: The ability to restrict file sharing and copy paste to only other trusted apps. This enables you to control the sharing of data, including email, messaging, printing and saving.
- Dynamic policy engine: More than 50 detailed app controls, including authentication frequency, geo and time fencing as well as remote lock and wipe.

Customers use the Mobile Security App Containerization Tool to add enterprise security services to apps including advanced features such as multi-factor authentication and Windows integrated Authentication (Kerberos or NTLM).

2 Containerization Process

The containerization process is simple and developers do not need to change a line of code. The first step is to obtain an unsigned version of the app. The unsigned version is a fully compiled app that is intended to be signed with an enterprise distribution certificate for distribution as an internal app to company employees through an enterprise appstore. The unsigned app can be an Android `apk` file or for iOS either `ipa` or static library.



Administrators then import the unsigned app into the Mobile Security App Containerization Tool, which inspects target apps and then injects containerization code and signs with the target enterprise distribution certificate. The result of this process is a fully signed app ready for enterprise distribution that can be uploaded to the Mobile Security Administrative Console or any other Enterprise AppStore.

The injected code looks for specified system calls to the target OS and injects Mobile Security frameworks in between to handle security tasks such as data encryption, networking and authentication. The beauty of this process is that developers don't have to change code. Rather, they make the same app that is on public appstores available in a different distribution package for enterprise customers.

A containerized app does not require its own login screen. Once the app is launched, it is redirected to the Mobile Security Container, which performs Single Sign-On and hands the session back to the app. Also, the app does not require VPN to connect to an internal website or services. Instead a secure AppTunnel is established between the app and Mobile Security Access Server, which provides secure transport for accessing internal sites and services.

3 Preparing an App for Containerization

To prepare an app for containerization, proceed as described in the following subsections:

- Part 3.1, "Preparing an iOS App for Containerization"
- Part 3.2, "Preparing an Android App for Containerization"

3.1 Preparing an iOS App for Containerization

In preparation for containerization, an iOS app for containerization must either be unsigned or signed by the same cert that will be used for signing after the app is containerized. A signed iOS app cannot be signed again with a different cert.

To create an unsigned iOS app follow these instructions:

1. Open `BitzerSecureContainer.xcodeproj` in Xcode
2. Go to Build Settings and set **Provisioning Profile** to **None** and **Code Signing Identity** to **Don't Code Sign**.
3. Save the project and exit.
4. From the command line, go to the folder where you extracted enterprise distribution static library project.
5. Run the following command:

```
xcodebuild clean build -project BitzerSecureContainer.xcodeproj -target Bitzer CODE_SIGN_IDENTITY=""
CODE_SIGNING_REQUIRED=NO -configuration Release
```

This generates an unsigned application bundle with the extension `.app` under the `build/Release-iphoneos` folder. You can create an IPA for the application bundle or pass the application bundle `.app` file to the containerization tool as an input file.

To create an IPA from application bundle, follow these steps:

1. Create a folder called `Payload`.
2. Copy the app bundle (`.app` file) to the `Payload` folder.
3. Zip the `Payload` folder and rename it, changing the file extension from `.zip` to `.ipa`

3.2 Preparing an Android App for Containerization

In preparation for containerization, an Android app can be signed or unsigned. An already signed Android app must not be signed with a different cert.

4 Running the Containerization Tool

The containerization tool is a command-line utility with the file name `c14n`. You can run `c14n` from anywhere because the Oracle Mobile Security Suite installer sets the containerization tool system path during installation.

The `c14n` command takes a number of parameters. You use different parameters for iOS apps than for Android apps. The tool can determine which type of app you are using as input.

Most of the parameters can be omitted, either because they are optional or because their value can be passed through environment variable instead of on the command line. This makes it easier to containerize multiple apps without having to type common parameters every time.

4.1 Syntax for c14n

The containerization tool command `c14n` has following syntax:

```
c14n [-c command]
     [-i input_file]
     [-o output_file]
     [-conf conf_file]
     [-cert cert_name -p provisioning_profile_file]
     [-keystore keystore_file -storepass keystore_password -storealias keystore_alias]
     [-x custom_parameters]
     [-xc]
     [-v ]
     [-version ]
     [-log {log_file | off} ]
```

4.2 Parameters and Their Options for c14n

-c *command*

The command to pass to containerization tool. The command determines what action will be performed on the input IPA. Command can be one of following:

- `inject`: Injects and signs an unsigned app in a single step. The output app is ready for enterprise distribution. This is the default value if **-c** is not specified.
- `injectonly`: Injects an unsigned app without signing it. The output app is unsigned and will need to be signed using `signonly` option.

Note: For iOS, codesign should not be directly used on an uninjected IPA.

- `signonly`: Sign an injected or uninjected app. The app must be unsigned or previously signed by the same certificate.
- `info`: Display information about an injected or an uninjected app.

-i *input_file*

The input file to run containerization tool on. It can be an iOS IPA (`.ipa`), iOS app bundle (`.app`), iOS Xcode Archive (`.xcarhive`) or Android APK (`.apk`). You can specify *`input_file`* as the full path to the file or just as the filename. If you specify just the filename, the command looks for the input file in the directory that the containerization tool is being run from.

You can also specify the input file by setting the environment variable *`C14N_INPUT_FILE`*.

-o *output_file*

The output app file generated by the command. If no path is specified then the output file is generated in the same location as input file. The output file is has the extension `.ipa` for iOS and `.apk` for Android.

If you do not specify an output file, the command generates a file with the default name *`input_file`*`-c14n-`*`command-MMDDYYYY`*`.ipa` or *`input_file`*`-c14n-`*`command-MMDDYYYY`*`.apk` in the same folder as the input file.

You can also specify the output file by setting the environment variable *`C14N_OUTPUT_FILE`*.

-conf *conf_file*

The conf file to use for iOS. This parameter is optional. If you do not specify this parameter, the default configuration file is used.

You can also specify the conf file by setting the environment variable *`C14N_CONF_FILE`*.

-cert *cert_name*

The certificate to use for code signing an iOS app. It must be a valid iOS development or distribution certificate present in Keychain.

You can also specify the certificate by setting the environment variable *`C14N_CERT_NAME`*.

This parameters is only needed for containerization of iOS apps.

-p *provisioning_profile_file*

The provisioning profile file to use for an iOS app. You can specify the path to the file or just the profile name. If you specify just the profile name, the containerization tool looks for it in the folder that the containerization tool is being run from.

You can also specify the provisioning profile by setting the environment variable *`C14N_PROVISIONING_PROFILE`*.

This parameters is only needed for containerization of iOS apps.

-keystore *keystore_file*

The path of the keystore file to use for signing an Android APK.

If you do not specify the keystore file, the command uses the default keystore. You can only use the default keystore for testing. To do so, you must also have the workspace app signed by the default keystore.

You can also specify the keystore file by setting the environment variable C14N_KEYSTORE_FILE.

This parameters is only needed for containerization Android apps.

-storepass *password*

The password for the keystore file that you specified with `-keystore`.

You can also specify the keystore password by setting the environment variable C14N_KEYSTORE_PASSWORD.

This parameters is only needed for containerization Android apps.

-storealias *alias*

An alias for the keystore entry to use for the keystore specified with `-keystore`.

You can also specify the keystore alias by setting the environment variable C14N_KEYSTORE_ALIAS.

This parameters is only needed for containerization Android apps.

-x *custom_parameters*

Custom parameters to pass to the containerization tool.

You can also specify the custom parameters by setting the environment variable C14N_CUSTOM_PARAMETERS.

This parameters is only needed for containerization Android apps.

-xc

Disables validation that the input iOS app is signed by same cert as the one specified by `-cert`. This validation is enabled by default.

This parameters is only needed for containerization iOS apps.

-v

Verbose mode. If not specified, the containerization tool is run in silent mode.

-version

Prints the version of containerization tool

-log *log_file* | off

Generates log file with the path specified by `log_file`, or disables log file generation if off is specified.

By default, a log file is generated under `/opt/BitzerC14N directory` each time the containerization tool is run. By default, the log file name is *`output_ipa`*`-c14n-`*`command-MMDDYYYY`*`.log`.

The following table gives snapshot of the parameters required for iOS and Android

Table 1 Containerization Tool (c14n) Parameters

Parameter	iOS	Android	Required	Environment Variable	Default
`-c`	Yes	Yes	No		`inject`
`-i`	Yes	Yes	Yes	`C14N_INPUT_FILE`	
`-o`	Yes	Yes	No	`C14N_OUTPUT_FILE`	*`input_file`*`-c14n-`*`command-MMDDYYYY`*`.ipa` or `.apk`
`-conf`	Yes	No	No	`C14N_CONF_FILE`	Set by `c14n`
`-cert`	Yes	No	Yes	`C14N_CERT_NAME`	

Table 1 (Cont.) Containerization Tool (c14n) Parameters

Parameter	iOS	Android	Required	Environment Variable	Default
`-p`	Yes	No	Yes	`C14N_PROVISIONING_PROFILE`	
`-keystore`		Yes	Yes	`C14N_KEYSTORE_FILE`	Default provided by `c14n`
`-storepass`		Yes	Yes	`C14N_KEYSTORE_PASSWORD`	Default provided by `c14n`
`-storealias`	No	Yes	Yes	`C14N_KEYSTORE_ALIAS`	Default provided by `c14n` if `-keystore` is not specified
`-x`	No	Yes	No	`C14N_CUSTOM_PARAMETERS`	
`-xc`	Yes	No	No		
`-v`	Yes	Yes	No		
`-version`	Yes	Yes	No		
`-log`	Yes	Yes	No		*output_ipa*`-c14n-`*command-MMDDYYYY*`.log`

4.3 Examples for c14n

Example 1 Inject and Sign an Unsigned iOS App

```
c14n -c inject -i Candidate.ipa -o injected.ipa -conf c14n.conf -cert 'iPhone Distribution:
Acme Corp Inc.' -p dist.mobileprovision -v
```

Example 2 Inject Unsigned iOS app Only and Run in Silent Mode

```
c14n -c injectonly -i Candidate.ipa -o injected.ipa -conf c14n.conf
```

Example 3 Sign an iOS App Only Without Injection

```
c14n -c signonly -i ./Candidate.ipa -o ./injected.ipa -cert 'iPhone Distribution: Acme Corp
Inc.' -p dist.mobileprovision -v
```

Example 4 Inject and Sign an Android apk

```
c14n -c inject -i candidate.apk -o containerized.apk -keystore prod-key.keystore -storepass
mypass -storealias mykey -v
```

Example 5 Inject Android apk Only and Run in Silent Mode

```
c14n -c injectonly -i candidate.apk -o containerized.apk
```

Example 6 Sign Android apk Only Without Injection

```
c14n -c signonly -i candidate.apk -o containerized.apk -keystore prod-key.keystore -storepass
mypass -storealias mykey -v
```

5 App Signing Requirements

The following requirements apply when signing apps. A containerized app will not work with the Workspace app if these conditions are not met.

5.1 App Signing Requirements for iOS

Workspace app and containerized apps must be signed using provisioning profiles that have the same App ID Prefix. Containerized apps will not work with Workspace if they have a different App ID Prefix.

5.2 App Signing Requirements for Android

Workspace app and containerized apps must be signed using the same certificate. Containerized apps will not work with Workspace if they are signed with a different certificate.

For more information on signing Android apps, see "Signing Your Applications" at `http://developer.android.com`.

6 Logging

Containerization tool creates a log file every time an app is containerized. By default log files are generated with name *`output ipa`*`-c14n-`*`command-MMDDYYYY`*`.log`. The file name and location can be specified using `-log` *`log_file parameter`*. Logging can also be disabled using `-log off` *`parameter`*.

7 Dev Mode for iOS

Dev mode enables developers to containerize and run an iOS app on a device by using Xcode. Dev mode provides developers the following benefits:

- They can leverage Single Sign On and AppTunnel for the app during development.
- They can debug any issues with the containerized app through XCode.
- They can switch file system encryption or SQLite encryption on or off during development to test and debug the application.

Pre-requisites for running dev mode are:

- iOS Device must be selected as the destination. Dev mode is not supported on Simulator
- A specific Code Signing Identity must be selected in build settings. Dev mode does not work with automatic selection for Code Signing Identity.
- A Workspace app signed by the same Code Signing Identity as the one selected in build settings must already be installed on the device.

To enable Dev Mode perform the following steps in your Xcode project.

1. Open your XCode project and edit the scheme.
2. In the Edit Scheme window, expand **Build steps** and go to **Pre-actions**.
3. Add a new Run Script.
4. In the Run Script window, set the following field values and click **OK**.
   - **Shell**: `/bin/bash`
   - **Provide build settings from**: Select your target from the list
   - **Script**: `/opt/BitzerC14N/bin/c14n -c devmode_clean -v`
5. In the Edit Scheme window, expand **Build step** and go to **Post-actions**.
6. Add a new Run Script.
7. In the Run Script window, set the following field values and click **OK**:
   - **Shell**: `/bin/bash`

- **Provide build settings from**: Select your target from drop down.
- **Script**: `/opt/BitzerC14N/bin/c14n -c devmode -v`

  Select **iOS device** as the destination.

Select **iOS Device** as the destination in Xcode and then run the app. Xcode will containerize the app before deploying and running it on the device. Once the app runs on the device, you can attach it to the debugger and take full advantage of Xcode debugging features for the containerized app.

After the containerized app is run for the first time, the following two Boolean keys are added to the app's Info plist:

- C14NEncryptFilesystem
- C14NEncryptSQLite

Setting the Boolean to YES or NO will allow you to enable or disable encryption to test your app.

When running an app in Dev mode through Xcode, the containerization tool generates a log file under `/opt/BitzerC14N/logs` folder with the name `c14n-devmode-`*`MM-DD-YYYY`*. You can monitor this file for any errors generated during containerization.

8 Related Documents

For more information, see the following documents in the Oracle Mobile Security Suite documentation set:

- *Oracle Mobile Security Suite Administrative Console Guide*
- *Oracle Mobile Security Suite Customization and Branding Guide*
- *Oracle Mobile Security Suite Installation Guide*
- *Oracle Mobile Security Suite Release Notes*
- *Oracle Mobile Security Suite Troubleshooting Guide*

9 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`.

Access to Oracle 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 you are hearing impaired.

Oracle Fusion Middleware Administrative Console for Oracle Mobile Security Suite, Release 3.0
E51928-01