Android 15

Common Criteria

Administrator Guidance for Zebra Devices (Q-6690)

Version 1.0
2026/05/05

ZEBRA

Terms of Use

Product Improvements

Continuous improvement of products is a policy of Zebra Technologies. All specifications and designs are subject to change without notice.

Liability Disclaimer

Zebra Technologies takes steps to ensure that its published Engineering specifications and manuals are correct; however, errors do occur. Zebra Technologies reserves the right to correct any such errors and disclaims liability resulting therefrom.

Limitation of Liability

In no event shall Zebra Technologies or anyone else involved in the creation, production, or delivery of the accompanying product (including hardware and software) be liable for any damages whatsoever (including, without limitation, consequential damages including loss of business profits, business interruption, or loss of business information) arising out of the use of, the results of use of, or inability to use such product, even if Zebra Technologies has been advised of the possibility of such damages. Some jurisdictions do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you.

Publication Date

2026/05/05

Contents

Contents

Zebra Android 15 Administrator Guidance

1.0 Document Introduction

This guide includes procedures for configuring Zebra Devices running Android 15 into a Common Criteria evaluated configuration and additionally includes guidance to application developers wishing to write applications that leverage the Zebra Device's Common Criteria compliant APIs and features.

1.1 Evaluated Devices

The evaluated devices include the following models and versions:

Table 1 Evaluated Devices

Product Model #	Device	CPU	Kernel	Android OS Version	Security Patch Level
ET401	ET4010A	Q-6690	6.1.128	Android 15	Mar 03 2026
ET401	ET4010B	Q-6690	6.1.128	Android 15	Mar 03 2026
ET401	ET401EA	Q-6690	6.1.128	Android 15	Mar 03 2026
ET401	ET401EB	Q-6690	6.1.128	Android 15	Mar 03 2026
ET401	ET4015A	Q-6690	6.1.128	Android 15	Mar 03 2026
ET401	ET4015B	Q-6690	6.1.128	Android 15	Mar 03 2026

To verify the OS Version and Security Patch Level on your device:

1. Tap on **Settings**.
2. Tap on **About phone**.
3. Scroll down and tap on the **Android version**.

1.2 Acronyms

Acronym	Description
AE	Android Enterprise
AES	Advanced Encryption Standard
API	Application Programming Interface
BYOD	Bring Your Own Device
CA	Certificate Authority
DO	Device Owner
DPC	Device Policy Controller
EMM	Enterprise Mobility Management
FBE	File Based Encryption
FDE	Full Disk Encryption
FIPS	Federal Information Processing Standards
MDM	Mobile Device Management
MX	Mobility Extensions
PKI	Public Key Infrastructure
TOE	Target of Evaluation

2.0 Evaluated Capabilities

The Common Criteria configuration adds support for many security capabilities. Some of those capabilities include the following:

- Data Protection
- Key Management
- Device Integrity
- Device Management
- Work Profile Separation
- VPN Connectivity
- Audit Logging

2.1 Data Protection

Android uses industry-leading security features to protect user data. The platform creates an application environment that protects the confidentiality, integrity, and availability of user data.

2.1.1 File-Based Encryption

Zebra devices by default use File Based Encryption [FBE]. To make it compliant to CC state, Zebra devices should follow the steps mentioned. See step 2 in 2. Create a StageNow Profile and use it to apply the CCReadinesspackage_A15_<CPU>.zip on the device.

Encryption is the process of encoding user data on an Android device using an encryption key. With encryption, even if an unauthorized party tries to access the data, they won't be able to read it. The device utilizes File-based encryption (FBE) which allows different files to be encrypted with different keys that can be unlocked independently.

Direct Boot allows encrypted devices to boot straight to the lock screen and allows alarms to operate, accessibility services to be available and phones to receive calls before a user has provided their credential.

With file-based encryption and APIs to make apps aware of encryption, it's possible for these apps to operate within a limited context before users have provided their credentials while still protecting private user information.

On a file-based encryption-enabled device, each device user has two storage locations available to apps:

- Credential Encrypted (CE) storage, which is the default storage location and only available after the user has unlocked the device. CE keys are derived from a combination of user credentials and a hardware secret. It is available after the user has successfully unlocked the device the first time after boot and remains available for active users until the device shuts down, regardless of whether the screen is subsequently locked or not.
- Device Encrypted (DE) storage, which is a storage location available both before the user has unlocked the device (Direct Boot) and after the user has unlocked the device. DE keys are derived from a hardware secret that's only available after the device has performed a successful Verified Boot.

By default, applications do not run during Direct Boot mode. If an application needs to take action during Direct Boot mode, such as an accessibility service like Talkback or an alarm clock application, the application can register components to run during this mode.

DE and CE keys are unique and distinct - no user's CE or DE key will match another. File-based encryption allows files to be encrypted with different keys, which can be unlocked independently. All encryption is based on AES-256 in XTS mode. Due to the way XTS is defined, it needs two 256-bit keys. In effect, both CE and DE keys are 512-bit keys.

By taking advantage of CE, file-based encryption ensures that a user cannot decrypt another user's data. This is an improvement on full-disk encryption (FDE) where there's only one encryption key, so all users must know the primary user's passcode to decrypt data. Once decrypted, all data is decrypted.

2.1.2 Cryptographic Self-Tests

Zebra devices also perform a series of cryptographic self-tests to ensure the correct performance of cryptographic algorithms automatically upon start up. If the verification of algorithm is successful, control is passed and the device continues the boot process. Should any of the tests fail, the device will halt the boot process and force a reboot of the device.

2.2 Key Management

2.2.1 KeyStore

The Android KeyStore class lets you manage private keys in secure hardware to make them more difficult to extract from the device. The KeyStore enables applications to generate and store credentials used for authentication, encryption, or signing purposes.

Keystore supports symmetric cryptographic primitives such as AES (Advanced Encryption Standard) and HMAC (Keyed-Hash Message Authentication Code) and asymmetric cryptographic algorithms such as RSA and EC. Access controls are specified during key generation and enforced for the lifetime of the key. Keys can be restricted to be usable only after the user has authenticated, and only for specified purposes or with specified cryptographic parameters. For more information, see the Authorization Tags and Functions pages.

Additionally, version binding binds keys to an operating system and patch level version. This ensures that an attacker who discovers a weakness in an old version of system or TEE software cannot roll a device back to the vulnerable version and use keys created with the newer version.

On Zebra Devices, the KeyStore is implemented in secure hardware.

2.2.2 KeyStore key and ID Attestation

Zebra Devices also support Key and ID Attestation, which empowers a server to gain assurance about the properties of keys.

2.2.3 KeyChain

The KeyChain class allows applications to use the system credential storage for private keys and certificate chains. KeyChain is often used by Chrome, Virtual Private Network (VPN) applications, and many enterprise applications to access keys imported by the user or by the mobile device management application.

Whereas the KeyStore is for non-shareable application-specific keys, KeyChain is for keys that are meant to be shared across profiles. For example, your mobile device management agent can import a key that Chrome will use for an enterprise website.

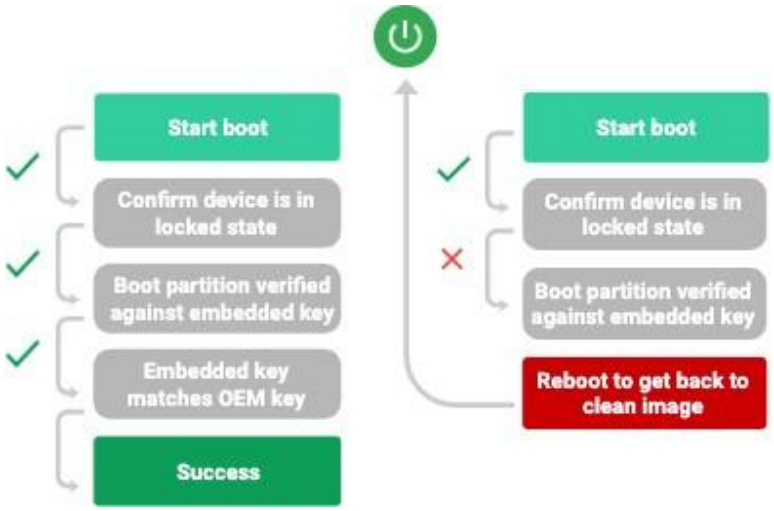
2.3 Device Integrity

Device integrity features protect the mobile device from running a tampered operating system. With companies using mobile devices for essential communication and core productivity tasks, keeping the OS secure is essential. Without device integrity, very few security properties can be assured. Android adopts several measures to guarantee device integrity at all times.

2.3.1 Verified Boot

Verified Boot is Android's secure boot process that verifies system software before running it. This makes it more difficult for software attacks to persistent across reboots and provides users with a safe state at boot time. Each Verified Boot stage is cryptographically signed. Each phase of the boot process verifies the integrity of the subsequent phase, prior to executing that code. Full boot of a compatible device with a locked bootloader proceeds only if the OS satisfies integrity checks. Verification algorithms used must be as strong as current recommendations from NIST for hashing algorithms (SHA-256) and public key sizes (RSA-2048).

Figure 1 Verified Boot Process



The Verified Boot state is used as an input in the process to derive disk encryption keys. If the Verified Boot state changes (e.g. the user unlocks the bootloader), then the secure hardware prevents access to data used to derive the disk encryption keys that were used when the bootloader was locked.

Enterprises can check the state of Verified Boot using KeyStore key attestation. This retrieves a statement signed by the secure hardware attesting to many attributes of Verified Boot along with other information about the state of the device.

Find out more about Verified Boot here.

2.3.2 Secure Boot

In addition to Google's mandated Verified. Zebra devices support additional integrity check with Secure Boot, to protect OS image's integrity. Zebra Devices (Secure Boot Enabled) ensures protection against binary manipulation of software and re-flashing attacks.

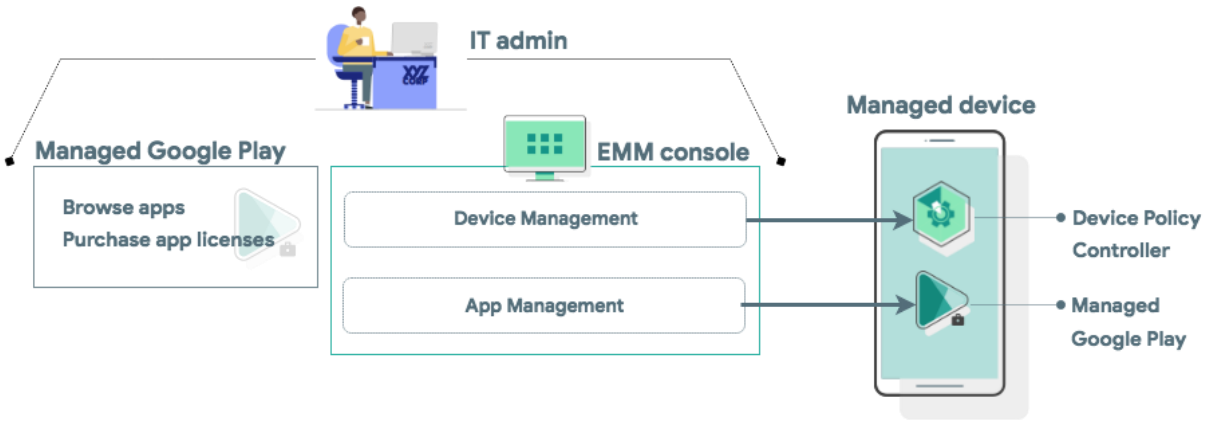
Secure boot enabled device protects itself from modification by untrusted subjects using following methods:

- First level Protection is a Secure Boot process that uses cryptographic signatures to ensure the authenticity/integrity of bootloader/Kernel. The protection is done using data fused into the device processor.
- Zebra Devices (Secure Boot Enabled protects its REK (Resource Environment Key) by limiting access to only trusted applications within the Trusted Environment (TEE).
- Zebra Devices includes a Trusted Module which utilizes the REK to protect all other key in the hierarchy.
- Bootloader Security offers users no other method of installing new software other than Zebra Secured prescribed OTA methods.

2.4 Device Management

The TOE leverages the device management capabilities that are provided through Android Enterprise which is a combination of three components: your EMM/MDM console, a device policy controller (DPC) which is your MDM Agent, and an EMM/MDM Application Catalog.

Figure 2 Components of an Android Enterprise solution.



2.4.1 EMM/MDM console

EMM solutions typically take the form of an EMM console—a web application you develop that allows IT admins to manage their organization, devices, and apps. To support these functions for Android, you integrate your console with the APIs and UI components provided by Android Enterprise.

2.4.2 DPC (MDM Agent)

All Android devices that an organization manages through your EMM console must install a DPC application during setup. A DPC is an agent that applies the management policies set in your EMM console to devices. Depending on which development option you choose, you can couple your EMM solution with the EMM solution's DPC, Android's DPC, or with a custom DPC that you develop.

End users can provision a fully managed or dedicated device using a DPC identifier (such as "afw#"), according to the implementation guidelines defined in the Play EMM API developer documentation.

- The EMM's DPC must be publicly available on Google Play, and the end user must be able to install the DPC from the device setup wizard by entering a DPC-specific identifier.
- Once installed, the EMM's DPC must guide the user through the process of provisioning a fully managed or dedicated device.

2.4.3 Managed Configuration

Managed configurations allow the organization's IT admin to remotely specify settings for apps. *Zebra OEMConfig* is Zebra's OEM-specific application that conforms to the OEMConfig model. It provides access to Zebra-specific and privileged functions via Managed Configurations that are exposed by the Zebra OEMConfig application.

Using EMM DPC enrolled as a Device Owner, you can set EMM policies or managed configuration values on a device.

IMPORTANT: You **must enable security logging** via your EMM DPC to achieve CC compliance.

To Use Zebra OemConfig through Test DPC:

1. Install Test DPC and make DPC as Device Owner.
2. Side load Zebra OEMConfig application from Google Play or from Zebra Support Central.
3. Enable security logging via Test DPC.

2.5 VPN Connectivity

IT admins can specify an Always On VPN to ensure that data from specified managed apps will always go through a configured VPN.

NOTE: This feature requires deploying a VPN client that supports both Always On and per-app VPN features.

IT admins can specify an arbitrary VPN application (specified by the application package name) to be set as an Always On VPN. IT admins can use managed configurations to specify the VPN settings for an application.

See 6.0 VPN Configuration for more information about VPN configuration options.

2.6 Audit Logging

IT admins can gather usage data from devices that can be parsed and programmatically evaluated for malicious or risky behavior. Activities logged include Android Debug Bridge (adb) activity, application launches, and screen unlocks. For Audit Logging, IT admins can do the following:

- Enable security logging for target devices, and the EMM's DPC must be able to retrieve both security logs and pre-reboot security logs automatically.
- Review enterprise security logs for a given device and configurable time window, in the EMMs console.
- IT admins can export enterprise security logs from the EMMs console.
- Capture relevant logging information from Logcat which does not require any additional configuration to be enabled.

See Table 4 for an example of a detailed audit logging table, along with information on how to view and export the different types of audit logs.

Zebra additional Security Logging

IMPORTANT: You **must use your EMM DPC enable security logging** to meet CC compliance.

3.0 Security Configuration

The Zebra Devices offer a rich built-in interface and MDM callable interface for security configuration. This section identifies the security parameters for configuring your device in Common Criteria mode and for managing its security settings.

3.1 Entering into Common Criteria State

IMPORTANT: The following 5 steps MUST be performed in order.

Critical Notice: Prior to initiating the download of product-specific packages, please consult Table 1 for detailed information regarding the product and its corresponding CPU identifier.

Software Archives

CPU	Build ID	Download	CCReadinesspackage	CCExitPackage
QCM6690	15-13-13.01-VG-U00-STD-ERS-04	https://www.zebra.com/content/servlet/supportdownload/downloadManager?dlp=/content/dam/support-dam/en/operating-system/restricted/0006/ER-FULL-UPDATE-15-13-13-01-VG-U00-STD-ERS-04.zip&c=ap&l=en&pagePath=/content/zebra1/ap/en/support-downloads/tablets/et401	15	https://www.zebra.com/us/en/support-downloads/software/mobile-computer-software/security-certification.html#accordion-9f6ede9710-item-0c3ecf4eb4

1. The Zebra Device for CC compliance should be Boring FIPS supported. Below are pre-requisites.
   a. Select the device from Table 1.
   b. If build fingerprint is greater than or equal to the build ID mentioned in software archive, then continue with step 2.
   c. Obtain and install the BSP from the software archive, ensuring it matches the product being used.
2. Create a StageNow Profile and use it to apply the CCReadinesspackage_A15_<CPU>.zip on the device.
   a. Download CCReadinesspackage_A15_<CPU>.zip as referred to in software archive.
   b. Use StageNow to deploy the package to the device. See 11.1 Creating and Applying the StageNow Profile.

NOTE: See 11.0 Annexure for more details on creating and applying the StageNow Profile.

3. Use StageNow to configure critical settings and to enroll your EMM agent as Device Owner. As required for CC compliance:
   a. Disable use of SDcard.
   b. Disable various alternate administrative functions.
   c. Enroll the Device Owner to provide administrative functions.
   d. Convert Staging method to Trusted Staging and deploy MDM Agent to enroll as Device Owner.

NOTE: See 11.2 Configuring Critical Settings Using Stage Now for more details.

4. Configure the device into Common Criteria state.

IMPORTANT: You must set the following options using your EMM after enrolling as Device Owner:
   a. Turn ON “Enable Common Criteria Mode” via TestDPC.
   b. Require a lockscreen password.
      Please review the Password Management items in 3.6 Common Criteria Related Settings.
   c. Disable Smart Lock.
      Smart Lock can be disabled using KEYGUARD_DISABLE_TRUST_AGENTS().

d. Disable Debugging Features (Developer options).
   By default Debugging features are disabled. The system administrator can prevent the user from enabling Debugging features using DISALLOW_DEBUGGING_FEATURES().
e. Disable installation of applications from unknown sources
   This can be disabled by using DISALLOW_INSTALL_UNKNOWN_SOURCES().
f. VPN Full Tunnel Configuration In order to leverage full tunnel IPSEC VPN, the VPN client must be configured to route all traffic (0.0.0.0) through the VPN application.

3.2 Successfully Achieving Common Criteria State

If all steps in 3.1 Entering into Common Criteria State are completed successfully, your device is in CC state. No additional configuration is required to ensure key generation, key sizes, hash sizes, and all other cryptographic functions meet NIAP requirements.

3.2.1 Limitations

Zebra devices in CC State will not support the following:

- Management of Work Profile
- Multi-User
- SDCard and USB external storage
- Downgrades are NOT allowed.

3.3 Exiting from Common Criteria State

1. Download CCExitPackage_A15_<CPU>.zip or relevant package from HERE.
2. Use StageNow to deploy the package to the device.

Refer 11.1 Creating and Applying the StageNow Profile for more details.

No additional configuration is required to ensure key generation, key sizes, hash sizes, and all other cryptographic functions meet NIAP requirements.

3.4 Cryptographic Module Identification

The TOE implements CAVP certified cryptographic algorithms which are provided by the following cryptographic components:

- BoringSSL Library
  - BoringCrypto version 20240805
- The TOE's LockSettings service
  - Android LockSettings service KBKDF (version 77561fc30db9aedc1f50f5b07504aa65b4268b88)
- Hardware Cryptography
  - TOE's Wi-FI Chipset provides an AES-CCMP implementation.
  - The TOE's application processor (e.g., Snapdragon 6375) provides additional cryptographic algorithms. The CAVP certificates correctly identify the specific hardware.

The use of other cryptographic components beyond those listed above was neither evaluated nor tested during the TOE's Common Criteria evaluation.

No additional configuration is needed for the cryptographic modules in order to be compliant.

Note: Some of the claimed devices are equipped with Strongbox capabilities; however, the scope of the evaluation does not cover the validation of this functionality, and its use is not supported within the evaluated configuration.

3.5 Permissions Model

Android runs all apps inside sandboxes to prevent malicious or buggy application code from compromising other apps or the rest of the system. Because the application sandbox is enforced in the kernel, this enforcement extends to the entire application regardless of the specific development environment, APIs used, or programming language. A memory corruption error in an application only allows arbitrary code execution in the context of that particular application, with the permissions enforced by the OS.

Similarly, system components run in least-privileged sandboxes in order to prevent compromises in one component from affecting others. For example, externally reachable components, like the media server and WebView, are isolated in their own restricted sandbox.

Android employs several sandboxing techniques, including Security-Enhanced Linux (SELinux), seccomp, and file-system permissions.

The purpose of a *permission* is to protect the privacy of an Android user. Android apps must request permission to access sensitive user data (such as contacts and SMS), as well as certain system features (such as camera and internet). Depending on the feature, the system might grant the permission automatically or might prompt the user to approve the request.

A central design point of the Android security architecture is that no application, by default, has permission to perform any operations that would adversely impact other apps, the operating system, or the user. This includes reading or writing the user's private data (such as contacts or emails), reading or writing another application's files, performing network access, keeping the device awake, and so on.

The DPC can pre-grant or pre-deny specific permissions using PERMISSION_GRANT_STATE APIs. In additio, the end user can revoke a specific apps permission by doing the following:

1. Tap on **Settings > Apps&notifications**.
2. Tap on the particular application and then tap **Permissions**.

From there the user can toggle off specific permissions. You can learn more about Android Permissions on developer.android.com.

3.6 Common Criteria Related Settings

The Common Criteria evaluation requires a range of security settings to be available. Those security settings are identified in the table below. In many cases, the administrator or user must have the ability to configure the setting, but no specific value is required.

Table 2 Common Criteria Settings

Security Feature	Setting	Description	Required Value	API	User Interface
Encryption	Device Encryption	Encrypts all internal storage	N/A	Encryption on by default with no way to turn off wipeData().	To wipe the device go to **Settings** > **System** > **Reset options** and select **Erase all date (factory reset)**.
	Wipe Device	Removes all data from device	No required value		
	Wipe Enterprise Data	Remove all enterprise data from device	No required value	wipeData called from secondary user.	
Password Management	Password Length	Minimum number of characters in a password	No required value	setPasswordMinimumLength()	To set a screen lock go to **Settings** > **Security & location** > **Screen lock** and tap **Password**.
	Password Complexity	Specify the type of characters required in a password	No required value	setPasswordQuality()	
	Password Expiration	Maximum length of time before a password must change	No required value	setPasswordExpirationTimeout()	
	Authentication Failures	Maximum number of authentication failures	10 or less	setMaximumFailedPasswordsForWipe()	

Table 2 Common Criteria Settings (Continued)

Security Feature	Setting	Description	Required Value	API	User Interface
Lockscreen	Inactivity to lockout	Time before lockscreen is engaged	No required value	setMaximumTimeToLock()	To set an inactivity lockout go to **Settings** > **Security & location** and tap on the gear icon next to Screen lock then tap on **Automatically lock** and select the appropriate value. To set a banner go to **Settings** > **Security & location** > **Lock screen preferences** > Lock screen message. Set a message and tap **Save**. Tap the power button to turn off the screen which locks the device.
	Banner	Banner message displayed on the lockscreen	Administrator or user defined text	setDeviceOwnerLockScreenInfo	
	Remote Lock	Looks the device remotely	Function must be available	lockNow()	
	Show Password	Disallows the displaying of the password on the screen of lock-screen password	Disable	This is disabled by default	
	Notifications	Controls whether notifications are displayed on the lockscreen	Enable/Disable are available options	KEYGUARD_DISABLE_SECURE_NOTIFICATIONS() KEYGUARD_DISABLE_UNREDACTED_NOTIFICATIONS	
	Control Biometric Face Unlock	Control the use of Biometric Face unlock	Enable/Diable are available options	KEYGUARD_DISABLED_FEATURES_SET	
Certificate Management	Import CA Certificates	Import CA Certificates into the Trust Anchor Database or the credential storage	No required value	installCaCert()	Tap on **Settings** > **Security & location** > **Advanced** > **Encryption & credentials** and select Install from storage
	Remove Certificates	Remove certificates from the Trust Anchor Database or the credential storage	No required value	uninstallCACert()	To clear all user installed certificates tap on **Settings** > **Security & location** > **Advanced** > **Encryption & credentials** and select **Clear credentials**. To remove a specific user installed certificate tap on **Settings** > **Security & location** > **Advanced** > **Encryption & credentials** > **Trusted credentials**. Switch to the **User** tab, select the certificate you want to delete and tap on **Remove**.
	Import Client Certificates	Import client certificates in to Keychain	No required value	installKeyPair()	Tap on **Settings** > **Security & location** > **Advanced** > **Encryption & credentials** and select **Install from storage**.
	Remove Client Certificates	Remove client certificates from Keychain	No required value	removeKeyPair()	To remove a specific user installed client certificate tap on **Settings** > **Security & location** > **Advanced** > **Encryption & credentials** > **User credentials**. Switch to the **User** tab, select the certificate you want to delete and tap on **Remove**.

Table 2 Common Criteria Settings (Continued)

Security Feature	Setting	Description	Required Value	API	User Interface
Radio Control	Control Wi-Fi	Control access to Wi-Fi	Enable/Disable are available options	DISALLOW_CONFIG_WIFI()	To disable Wi-Fi tap on **Settings** > **Network & internet** and toggle **Airplane mode** to On.
	Control GPS	Control access to GPS	Enable/Disable are available options	DISALLOW_SHARE_LOCATION() DISALLOW_CONFIG_LOCATION()	
	Control Cellular	Control access to Cellular	Enable/Disable are available options	DISALLOW_CONFIG_MOBILE_NETWORKS()	To disable Cellular tap on **Settings** > **Network & internet** > **Mobile network** and tap on your carrier and toggle to Off.
	Control NFC	Control access to NFC	Enable/Disable are available options	DISALLOW_OUTGOING_BEAM()	To disable NFC tap on **Settings** > **Connected devices** > **Connection preferences** and toggle **NFC** to Off.
	Control Bluetooth	Control access to Bluetooth	Enable/Disable are available options	DISALLOW_BLUETOOTH () DISALLOW_BLUETOOTH_SHARING() DISALLOW_CONFIG_BLUETOOTH()	
	Control Location Service	Control access to Location Service	Enable/Disable are available options	DISALLOW_SHARE_LOCATION() DISALLOW_CONFIG_LOCATION()	
Wi-Fi Settings	Specify Wi-Fi SSIDs	Specify SSID values for connecting to Wi-Fi. Can also create white and black lists for SSIDs.	No required value	WifiEnterpriseConfig()	
	Set WLAN CA Certificate	Select the CA Certificate for the Wi-FI connection	No required value	WifiEnterpriseConfig()	
	Specify security type	Specify the connection security (WPA2, WPA3 etc)	No required value	WifiEnterpriseConfig()	
	Select authentication protocol	Specify the EAP-TLS connection values	No required value	WifiEnterpriseConfig()	
	Select client credentials	Specify the client credentials to access a specified WLAN	No required value	WifiEnterpriseConfig()	
	Control Always-on VPN	Control access to Always-on VPN	Enable/Disable are available options	setAlwaysOnVPNPackage()	

Table 2 Common Criteria Settings (Continued)

Security Feature	Setting	Description	Required Value	API	User Interface
Hardware Control	Control Microphone (across device)	Control access to microphone across the device	Enable/Disable are available options	DISALLOW_UNMUTE_MICROPHONE()	
	Control Microphone (per-app basis)	Control access to microphone per application	Enable/Disable are available options		Tap on **Settings** > **Apps & notifications** > **App permissions** > **Microphone** and then de-select the apps to remove permissions.
	Control Camera (per-app basis)	Control access to camera per application	Enable/Disable are available options		Tap on **Settings** > **Apps & notifications** > **App permissions** > **Camera** and then de-select the apps to remove permissions.
	Control USB Mass Storage	Control access to mounting the device for storage over USB.	Enable/Disable are available options	DISALLOW_MOUNT_PHYSICAL_MEDIA()	
	Control USB Debugging	Control access to USB debugging.	Enable/Disable are available options	DISALLOW_DEBUGGING_FEATURES()	
	Control USB Tethered Connections	Control access to USB tethered connections.	Enable/Disable are available options	DISALLOW_CONFIG_TETHERING()	
	Control Bluetooth Tethered Connections	Control access to Bluetooth tethered connections.	Enable/Disable are available options	DISALLOW_CONFIG_TETHERING()	
	Control Hotspot Connections	Control access to Wi-Fi hotspot connections	Enable/Disable are available options	DISALLOW_CONFIG_TETHERING()	
	Automatic Time	Allows the device to get time from the Wi-Fi connection	Enable/Disable are available options	setAutoTimeRequired()	Tap on **Settings** > **System** > **Date & time** and toggle **Automatic date & time** to On.
Application Control	Install Application	Installs specified application	No required value	PackageInstaller.Session()	
	Uninstall Application	Uninstalls specified application	App to uninstall	uninstall()	To uninstall an application tap on **Settings** > **Applications & notifications** > **See all**. Select the application and tap on **Uninstall**.
	Application Whitelist	Specifies a list of applications that may be installed	No required value	This is done by the EMM/MDM when they setup an application catalog which leverages PackageInstaller.Session()	
	Application Blacklist	Specifies a list of applications that may not be installed	No required value	PackageInstaller.SessionInfo()	
	Application Repository	Specifies the location from which applications may be installed	No required value	DISALLOW_INSTALL_UNKNOWN_SOURCES()	

Table 2 Common Criteria Settings (Continued)

Security Feature	Setting	Description	Required Value	API	User Interface
TOE Management	Enrollment	Enroll TOE in management	No required value		During device setup scan EMM/MDM provided QR code or enter EMM/MDM DPC identifier
	Disallow Unenrollment	Prevent the user from removing the managed profile	Enable/Disable	DISALLOW_REMOVE_MANAGED_PROFILE() DISALLOW_FACTORY_RESET()	
	Unenrollment	Unenroll TOE from management	App to uninstall	uninstall() – this API can be used to uninstall the MDM Agent from the device. Uninstalling the MDM agent from an enterprise profile will delete the entire profile and all its applications.	This API can be used to uninstall enterprise apps. If an admin uninstalls the MDM agent installed on an enterprise profile, the entire profile and all enterprise applications are deleted.
	Allow Developer Mode	Controls Developer Mode access	Enable/Disable are available options	DISALLOW_DEBUGGING_FEATURES()	
	Sharing Data Between Enterprise and Personal Apps	Controls data sharing between enterprise and work apps	Enable/Disable	DISALLOW_CROSS_PROFILE_COPY_PASTE() addCrossProfileIntentFilter()	

3.7 Password Recommendations

When setting a password, you should select a password that:

- Does not use known information about yourself (e.g. pets names, your name, kids' names or any information available in the public domain);
- Is significantly different from previous passwords (adding a '1' or "!" to the end of the password is not sufficient); or
- Does not contain a complete word (such as Password!).
- Does not contain repeating or sequential numbers and/or letters.

3.8 Bug Reporting Process

Zebra supports a bug filing system for the Android OS. For more information, see zebra.com/us/en/about-zebra/contact-zebra/contact-tech-support.html.

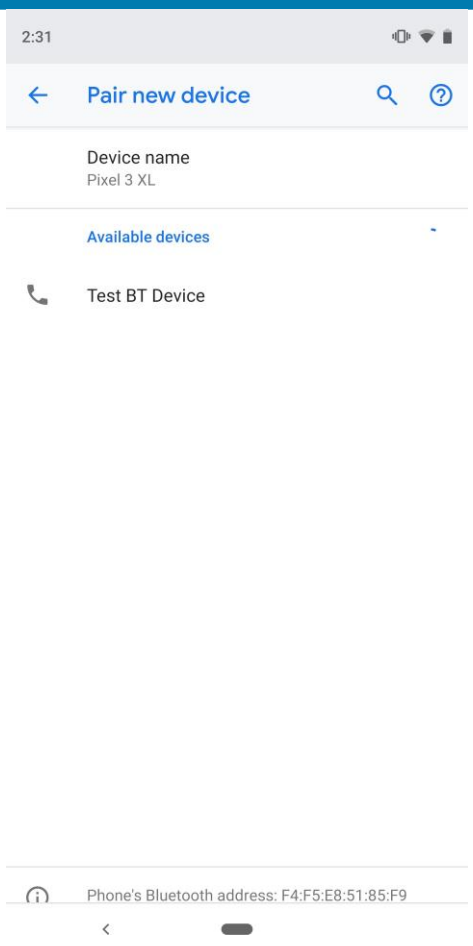
4.0 Bluetooth Configuration

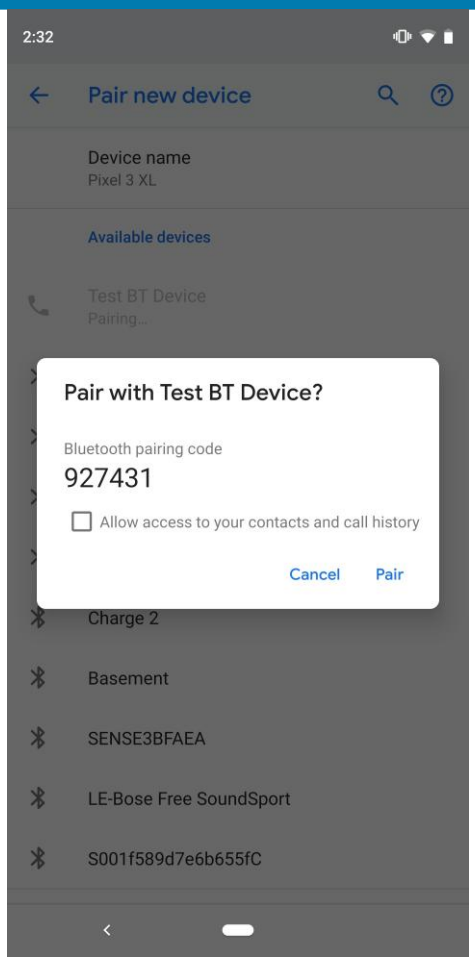
Follow the below steps to pair and connect your device using Bluetooth.

4.1 Pair

NOTE: If your device is connected to another device via Bluetooth, you will see a Bluetooth icon at the top of the screen.

1 Open your phone or tablet's Settings application.
2 Tap **Connected devices** > **Connection preferences** > **Bluetooth**. Make sure Bluetooth is turned on.
3 Tap **Pair new device**.
4 Tap the name of the Bluetooth device you want to pair with your phone or tablet.
5 Follow the on-screen steps.





4.2 Connect

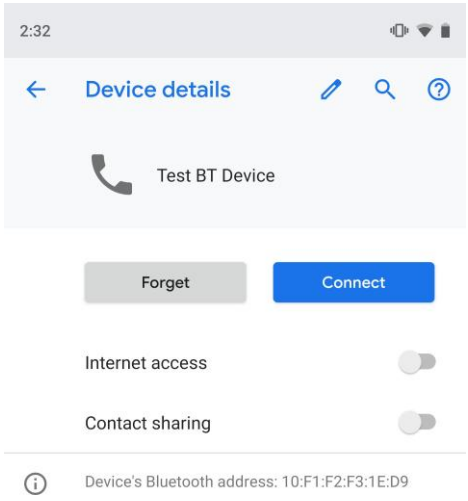
NOTE: If your device is connected to another device via Bluetooth, you will see a Bluetooth icon at the top of the screen.

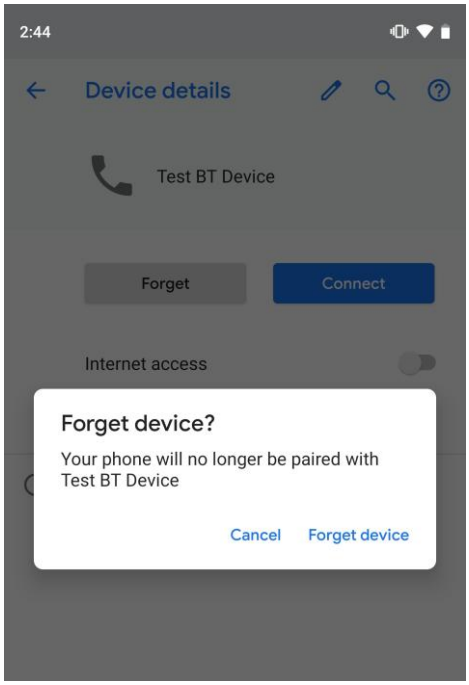
1. Open your phone or tablet's **Settings** application.
2. Tap **Connected devices** > **Connection preferences** > **Bluetooth**.
3. Make sure Bluetooth is turned on.
4. In the list of paired devices, tap a paired but unconnected device.
5. When your phone or tablet and the Bluetooth device are connected, the device shows as "Connected" in the list.

4.3 Remove Previously Paired Device

NOTE: If your device is connected to another device via Bluetooth, you will see a Bluetooth icon at the top of the screen.

6. Open your phone or tablet's **Settings** application.
7. Tap **Connected devices > Previously connected devices**.
8. Tap the gear icon to the right of the device you want to unpair.
9. Tap **Forget** and confirm in the popup window by tapping **Forget device**.



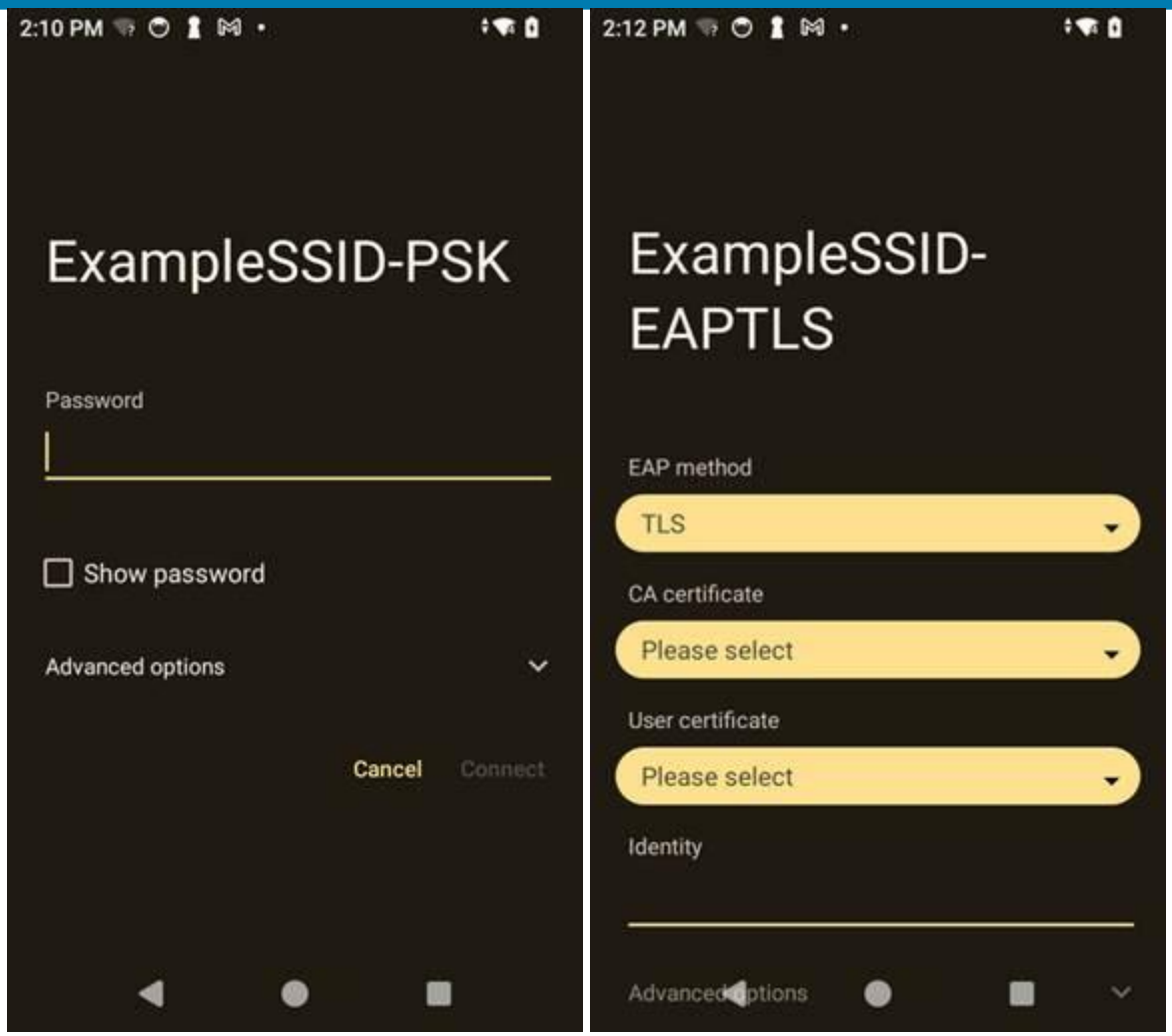


NOTES:

- For additional support information around Bluetooth, see developer.android.com/reference/android/bluetooth/package-summary.html.
- For Zebra Bluetooth-specific configuration, see the BluetoothTechDoc.

5.0 Wi-Fi Configuration

Wi-Fi can be configured either by the user through the system Settings application or through the EMM's DPC. The below screenshots show the user controls through the Settings application including configuration of the SSID, password, client certificate, and CA certificate when applicable:



Android supports the WPA2-Enterprise and WPA3-Enterprise protocol, which is specifically designed for enterprise networks and can be integrated into a broad range of Remote Authentication Dial-In User Service (RADIUS) authentication servers. Zebra devices also support WPA3-Enterprise 192-bit mode which is aligned with the CNSA (Commercial National Security Algorithm) security recommendations for high-security Wi-Fi networks.

IT admins have several abilities to control the environment for your devices. They can:

- Silently provision enterprise WiFi configurations on managed devices via the EMM's DPC, including:
  - SSID
  - Password
  - Identity
  - Certificate for clients authorization
  - CA certificate(s)
- Lock down Wi-Fi configurations on managed devices to prevent users from creating new configurations or modifying corporate configurations.
- Lock down corporate Wi-Fi configurations in either of the following configurations:
  - Users cannot modify any WiFi configurations provisioned by the EMM, but may add and modify their own user-configurable networks (for instance personal networks).
  - Users cannot add or modify any WiFi network on the device, limiting Wi-Fi connectivity to just those networks provisioned by the EMM.

When the device tries to connect to a WiFi network it performs a standard captive portal check which bypasses the full tunnel VPN configuration. If the administrator wants to turn the captive portal check off, they need to do this physically on the device before enrolling it in to the MDM by:

- Enable Developer Options by tapping on **Settings > About** phone and tapping on **Build number**

five times until they see that **Developer options** has been enabled.

- Enable Android Debug Bridge (ADB) over USB by tapping on **Settings > System > Advanced > Developer options** and scroll down to USB debugging and enable the toggle to **On**.
- Connect to the device to a workstation that has ADB installed and type in “adb shell settings put global captive_portal_mode 0” followed by pressing **Enter.**
- You can verify the change by typing **“adb shell settings get global captive_portal_mode”** and confirming that the return value is **“0”**.
- Turn off Developer options by tapping on **Settings > System > Advanced>Developer** options and toggling the **On** option to **Off** at the top.

If a WiFi connection unintentionally terminates, the end user will need to reconnect to re-establish the session.

6.0 VPN Configuration

Android supports securely connecting to an enterprise network using VPN:

- Always-on VPN—The VPN can be configured so that apps don’t have access to the network until a VPN connection is established, which prevents apps from sending data across other networks.

  Always-on VPN supports VPN clients that implement VpnService. The system automatically starts that VPN after the device boots. Device owners and profile owners can direct work apps to always connect through a specified VPN. Additionally, users can manually set Always-on VPN clients that implement VpnService methods using **Settings > More > VPN**. Always-on VPN can also be enabled manually from the **Settings** menu.

7.0 Secure Update Process

Over the Air (OTA) updates (which includes baseband processor updates) using a public key chaining will be verified against a zip file of certificates present on the device. Verification succeeds if the OTA package is signed by the private key corresponding to any public key in this file. Should this verification fail, the software update will fail and the update will not be installed. Zebra recommends using OEMConfig as the method for administrator to upgrade/downgrade the device.

Downgrade

Default zebra devices supports Downgrade, but once you have created a StageNow Profile, and used it to apply the CC Readiness package on the device as detailed in step 2 in 3.1 Entering into Common Criteria State, then it will be possible to update but it will no longer be possible to downgrade.

To enable downgrade, you must follow the steps from 3.3 Exiting from Common Criteria State.

To upgrade a Zebra device with a new Over the Air (OTA) update, you must Acquire and Transfer/Apply the suitable update file(s).

In A15, incremental patches are created as true delta packages and are applied sequentially.

7.1 Acquire

Find the build you want on the Zebra support central: https://www.zebra.com/us/en/support-downloads.html. Depending on the build you choose, you may need a delta package, a full OTA package, or both in order to get the device(s) from the current build to desired build.
Download the necessary file(s) and then perform the appropriate steps.

7.2 Transfer/Apply

7.2.1 Delta Package or Full OTA Package

1. Place the downloaded file on the https server that is reachable from the device.
2. Acquire the URL of the file location on the server. (If server requires authentication provide the credentials).
3. Use OEMConfig - File Management-Download File Source URL.

See https://techdocs.zebra.com/oemconfig/latest/mc2/and Download Destination Path and File Name to copy the file from the server to the device.

4. Use OEMConfig-Firmware Over The Air Configuration-Mode Manual Action=OS Update and OS Update/Upgrade/Downgrade File to apply the update from downloaded file.

IMPORTANT: In a situation where any future Zebra OS security patch installs successfully but fails to boot into a new installed image, On A15, the bootloader will fall back to the old OS.

- If the old OS is CC compliant:
  a. The device will be Factory Reset.
  b. User data will be erased.
- If the old OS is non-CC compliant:
  a. The device will be Factory Reset.
  b. User data will be erased.
  c. The device will be in a non-CC compatible OS image.

To recover from a non-CC compliant baseline OS image, the user must follow the steps in 3.1 Entering into Common Criteria State.

8.0 Audit Logging

8.1 Security Logs

A MDM agent acting as Device Owner can control the logging with DevicePolicyManager#setSecurityLoggingEnabled. When security logs are enabled, Device Owner apps receive periodic callbacks from DeviceAdminReceiver#onSecurityLogsAvailable, at which time new batch of logs can be collected viaDevicePolicyManager#retrieveSecurityLogs. SecurityEvent describes the type and format of security logs being collected.

Audit events from the Security Log are those where the "Keyword" field appears first in the format. Subject Identities for Security Logs can be identified individually from the “message” field in each log. For example: **<Keyword> (<Date><Timestamp>): <message>**

8.2 Logcat Logs

Logcat logs can be read by a command issued via an ADB shell running on the phone. Information about reading Logcat logs can be found at developer.android.com/studio/command-line/logcat. The command to issue a dump of the logcat logs is:

```
> adb logcat
```

Logcat logs cannot be exported from the device outside of using the above ADB command to dump to a file, then retrieving the file from the device (which requires developer settings enabled and administrative permissions).

Logcat logs can also be read by an application (for example an MDM agent) granted permission from an ADB shell:

```
> adb shell pm grant <application_package_name> android.permission.READ_LOGS
```

Audit events from the Logcat log are those where the "Keyword" field appears after the timestamp field in the format. For example: **<Date> <Time> <ID> | <Keyword> <Message>**

Table 3 shows examples of audit events:

Table 3 Audit Events

Requirement	Auditable Events	Additional Audit Record Contents	Log Events & Examples
FAU_GEN.1	Start-up and shutdown of the audit functions		Start-up: LOGGING_STARTED (<Date> <Timestamp>): Note: Subject Identity for this log is always fixed to be the TOE's logging service and as a result, is not explicitly logged. Shutdown: All logs are stored in memory. When audit functions are disabled, all memory being used by the audit functions is released by the OS, and so this log cannot be seen.
	All administrative actions	See Management Function Table	
	Start-up and shutdown of the Rich OS		Start-up: OS_STARTUP (<Date> <Timestamp>): <verified boot status color> <dm-verity status> Note: Subject Identity for this log is always fixed to be the TOE's init process and as a result, is not explicitly logged. Shutdown: All logs are stored in memory. This log is not capturable or persistent through boot, and thus isn't available to an MDM Administrator
FCS_CKM_EXT.1	[None].	No additional information.	
FCS_CKM_EXT.5	[None].	No additional information.	
FCS_CKM.1	[None].	No additional information.	
FCS_STG_EXT.1	Import or destruction of key.	Identity of key. Role and identity of requestor.	KEY_IMPORT (<Date> <Timestamp>): <success-boolean> <key name> <requesting process / role / identify>
	[No other events]		KEY_DESTRUCTION (<Date> <Timestamp>): <success-boolean> <key name> <requesting process / role / identify>
FCS_STG_EXT.3	Failure to verify integrity of stored key.	Identity of key being verified.	KEY_INTEGRITY_VIOLATION (<Date> <Timestamp>): <key name> <requesting process / role / identify>
FDP_DAR_EXT.1	[None].	No additional information.	N/A
FDP_DAR_EXT.2	[None].	No additional information.	N/A
FDP_STG_EXT.1	Addition or removal of certificate from Trust Anchor Database.	Subject name of certificate.	CERT_AUTHORITY_INSTALLED (<Date> <Timestamp>): <success-boolean> <cert authority> <user id> CERT_AUTHORITY_REMOVED (<Date> <Timestamp>): <success-boolean> <cert authority> <user id>

FIA_X509_EXT.1	Failure to validate X.509v3 certificate.	Reason for failure of validation.	<Date> <Time> <ID> System.err: java.security.cert.CertPathValidatorException [<error message>] <Date> <Time> <ID> ValidatableSSLSocket: Failed to establish a TLS connection to <IP address> ... [<error message>]

Table 3 Audit Events (Continued)

Requirement	Auditable Events	Additional Audit Record Contents	Log Events & Examples
FMT_SMF_EXT.2	[none].	[none].	
FPT_NOT_EXT.1	[None].	[No additional information].	
FPT_TST_EXT.1	Initiation of self-test.	[none]	CRYPTO_SELF_TEST_COMPLETED (<Date><Timestamp>): 1 Note: Subject Identity for this log is always fixed to be the TOE's init process and as a result, is not explicitly logged.
	Failure of self-test.		CRYPTO_SELF_TEST_COMPLETED (<Date><Timestamp>): 0 Note: Subject Identity for this log is always fixed to be the TOE's init process and as a result, is not explicitly logged.
FPT_TST_EXT.2(1) (Selection is optional)	Start-up of TOE.	No additional information.	See audits for FAU_GEN.1 - Start-up and shutdown of the Rich OS
	[none]	No additional information.	
WLAN EP Audit Logs:			
FCS_TLSC_EXT.1/WLAN	Failure to establish an EAP-TLS session.	Reason for failure	Errors: <Date> <Time> <ID> wpa_supplicant: wlan0: CTRL-EVENT-EAP-TLS-CERT-ERROR <Error Details> <Date> <Time> <ID> wpa_supplicant: wlan0: CTRL-EVENT-EAP-FAILURE EAP authentication failed <Date> <Time> <ID> wpa_supplicant: TLS - SSL error: <error message> Termination (follows after above error log): <Date> <Time> <ID> wpa_supplicant: wlan0: CTRL-EVENT-DISCONNECTED bssid=<BSSID> reason=<reason code>
	Establishment/termination of an EAP-TLS session.	Non-TOE endpoint of connection	Establishment: <Date> <Time> <ID> wpa_supplicant: wlan0: CTRL-EVENT-CONNECTED - Connection to <BSSID> completed. Termination <Date> <Time> <ID> wpa_supplicant: wlan0: CTRL-EVENT-DISCONNECTED bssid=<BSSID> reason=<reason code>

Table 3 Audit Events (Continued)

Requirement	Auditable Events	Additional Audit Record Contents	Log Events & Examples
FIA_X509_EXT.1/WLAN	Failure to validate X.509v3 certificate	Reason for failure of validation	<Date> <Time> <ID> wpa_supplicant: TLS: Certificate verification failed, <reason/details>
FIA_X509_EXT.6	Attempts to load/revoke certificates	no additional information].	See audits for FCS_STG_EXT.1 – Import and Destruction of keys
FPT_TST_EXT.1/WLAN (note: can be performed by TOE or TOE platform)	Execution of this set of TSF self-tests. [none].	[no additional information].	See the audits for MDFPP FPT_TST_EXT.1, these self-tests are included in the same audit message.
FTA_WSE_EXT.1	All attempts to connect to access points.	Identity of access point being connected to as well as success and failures (including reason for failure).	<Date> <Time> <ID> wpa_supplicant: wlan0: Trying to associate with SSID <SSID> 05-03 02:03:32.431 23406 23406 I wpa_supplicant: wlan0: Trying to associate with SSID 'fscaesdot1x <Date> <Time> <ID>wpa_supplicant: wlan0: Associated with <MAC> 05-03 02:03:32.673 23406 23406 I wpa_supplicant: wlan0: Associated with 94:64:24:89:b2:d2 See audits for FIA_X509_EXT.1/WLAN and FCS_TLSC_EXT.1/WLAN for failures to connect
FTP_ITC_EXT.1/WLAN	All attempts to establish a trusted channel.	Identification of the non-TOE endpoint of the channel.	Same as above (FTA_WSE_EXT.1)
FIA_BLT_EXT.1	Failed Authorization of Bluetooth device	User authorization decision	See audits for FIA_BLT_EXT.2 – Failure of Bluetooth Connection Status: 9 – BT_STATUS_AUTH_FAILURE 11 – BT_STATUS_AUTH_REJECTED HCI Reason: 5 = Authentication Failure 19 = Remote Request Disconnect 26 = Remote Error
FIA_BLT_EXT.1	Failed user authorization for local Bluetooth Service	[last 2 octets of the] BD_ADDR and [no other information] Bluetooth profile Identity of local service with profile name	BluetoothDatabase: getProfileConnectionPolicy: device <masked MAC> profile=<profile name> connectionPolicy=<#> CachedBluetoothDevice: onProfileStateChanged: profile <profilename>, device <masked MAC>, newProfileState=0 (0 means connection state is disconnected)
FIA_BLT_EXT.2	Initiation of Bluetooth connection	Complete BD_ADDR and no other information	<Date> <Time> <ID> I BluetoothBondStateMachine: bondStateChangeCallback: Status: 0 Address: <MAC address> newState: 2 hciReason: 0 <Date> <Time> <ID> BluetoothBondStateMachine: Bond State Change Intent:<MAC Address> BOND_BONDING => BOND_BONDED

Table 3 Audit Events (Continued)

Requirement	Auditable Events	Additional Audit Record Contents	Log Events & Examples
FIA_BLT_EXT.2	Failure of Bluetooth connection	Reason for failure	<Date> <Time> <ID> I BluetoothBondStateMachine: bondStateChangeCallback: Status: <Status> Address: <MAC address> newState: 0 hciReason: <hci reason> <Date> <Time> <ID> I BluetoothBondStateMachine: Bond State Change Intent:<MAC address> BOND_BONDING => BOND_NONE
FIA_BLT_EXT.3	Duplicate connection attempt	BD_ADDR of connection attempt	This is performed at the HCI layer and is not able to be logged.

Table 4 shows examples of sample management function audits.

Table 4 Sample Management Function Audits

REQUIREMENT	FUNCTION	Required Value	AUDIT LOG
FMT_SMF_EXT.1.1 Function 1	Configure password policy		
FMT_SMF_EXT.1.1 Function 1a	a. minimum password length	Greater than or equal to 8	PASSWORD_COMPLEXITY_SET (<Date><Timestamp>): <package> 0 0 0 65536 1 0 1 0 0 1
FMT_SMF_EXT.1.1 Function 1b	b. minimum password complexity	No required value	PASSWORD_COMPLEXITY_SET (<Date><Timestamp>): <package> 0 0 0 131072 1 0 1 0 0 1
FMT_SMF_EXT.1.1 Function 1c	c. maximum password lifetime		<Keyword> (<Date><Timestamp>): <message> PASSWORD_EXPIRATION_SET (<Date><Timestamp>): <package> 0 0 500000
FMT_SMF_EXT.1.1 Function 2	Configure session locking policy	10 minutes or less	
FMT_SMF_EXT.1.1 Function 2a	a. screen-lock enabled/disabled (By setting a password complexity, the admin enables screen-lock. Passwords are required as a part of the evaluated configuration and this prevents the user from leaving the evaluated configuration and disabling screen-lock)	Enabled	PASSWORD_COMPLEXITY_SET (<Date><Timestamp>): <package> 0 0 5 0 1 0 1 0 0 1
FMT_SMF_EXT.1.1 Function 2b	b. screen lock timeout	10 minutes or less	MAX_SCREEN_LOCK_TIMEOUT_SET (<Date><Timestamp>): <package> 0 0 100000

Table 4 Sample Management Function Audits (Continued)

REQUIREMENT	FUNCTION	Required Value	AUDIT LOG
FMT_SMF_EXT.1.1 Function 2b	b. screen lock timeout (after setting a max time, the admin can prevent any user changes with this)		USER_RESTRICTION_ADDED (<Date><Timestamp>): <package> 0 no_config_screen_timeout
FMT_SMF_EXT.1.1 Function 2c	c. number of authentication failures	10 or less	MAX_PASSWORD_ATTEMPTS_SET (<Date><Timestamp>): <package> 0 0 10
FMT_SMF_EXT.1.1 Function 8a	Configure application installation policy a. restricting the sources of applications	Disable	USER_RESTRICTION_REMOVED (<Date><Timestamp>): <package> 0 no_install_unknown_sources
FMT_SMF_EXT.1.1 Function 8a	Configure application installation policy a. restricting the sources of applications	Enable	USER_RESTRICTION_ADDED (<Date><Timestamp>): <package> 0 no_install_unknown_sources
FMT_SMF_EXT.1.1 Function 8c	Configure application installation policy c. denying installation of applications	Enable	USER_RESTRICTION_ADDED (<Date><Timestamp>): <package> 0 no_install_apps
FMT_SMF_EXT.1.1 Function 8c	Configure application installation policy c. denying installation of applications	Disable	USER_RESTRICTION_REMOVED (<Date><Timestamp>): <package> 0 no_install_apps

9.0 FDP_DAR_EXT.2 & FCS_CKM.2(2) – Sensitive Data Protection Overview

Using the NIAPSEC library, sensitive data protection including Biometric protections are enabled by default by using the Strong configuration.

To request access to the NIAPSEC library, please reach out to: niapsec@google.com.

The library provides APIs via SecureContextCompat to write files when the device is either locked or unlocked. Reading an encrypted file is only possible when the device is unlocked and authenticated biometrictrically.

Saving sensitive data files requires a key to be generated in advance. See 10.2.2 SecureKeyGenerator.

Supported Algorithms via SecureConfig.getStrongConfig():

- File Encryption Key: AES256 - AES/GCM/NoPadding
- Key Encryption Key: RSA3072 - RSA/ECB/OAEPWithSHA-256AndMGF1Padding

Writing Encrypted (Sensitive) Files:

- SecureContextCompat opens a FileOutputStream for writing and uses SecureCipher (below) to encrypt the data.
- The Key Encryption Key, which is stored in the AndroidKeystore encrypts the File Encryption Key which is encoded with the file data.

Reading Encrypted (Sensitive) Files:

- SecureContextCompat opens a FileInputStream for reading and uses SecureCipher (below) to decrypt the data.
- The Key Encryption Key, which is stored in the AndroidKeystore decrypts the File Encryption Key which is encoded with the file data.

The File encryption key material is automatically destroyed and removed from memory after each operation. See EphemeralSecretKey for more information.

9.1 SecureContextCompat

NOTE: SecureContextCompat is included in the NIAPSEC library.

SecureContextCompat is used to encrypt and decrypt files that require sensitive data protection.

Supported Algorithms

- AES256 - AES/GCM/NoPadding
- RSA3072 - RSA/ECB/OAEPWithSHA-256AndMGF1Padding

Table 5 SecureContextCompat Public Constructors

Constructor	Description
SecureContextCompat	new SecureContextCompat(Context, BiometricSupport) See BiometricSupport Constructor to create an instance of the SecureContextCompat with Biometric support.

Table 6 SecureContextCompat Public Methods

Method	Description
FileOutputStream	openEncryptedFileOutput (String name, int mode, String keyPairAlias) Gets an encrypted file output stream using the asymmetric/ephemeral algorithms specified by the default configuration, using NIAP standards. -name - The file name -mode - The file mode, usually Context.MODE_PRIVATE -keyPairAlias - Encrypt data with the AndroidKeyStore key referenced - Key Encryption Key

void	openEncryptedFileInput (String name, Executor executor, EncryptedFileInputStreamListener listener) Gets an encrypted file input stream using the asymmetric/ephemeral algorithms specified by the default configuration, using NIAP standards. -name - The file name -Executor - to handle the threading for BiometricPrompt. Usually Executors.newSingleThreadExecutor() -Listener for the resulting FileInputStream.

Code Examples

```
SecureContextCompat secureContext = new SecureContextCompat(getApplicationContext(),
SecureConfig.getStrongConfig(biometricSupport));


// Open a sensitive file for writing
FileOutputStream outputStream = secureContext.openEncryptedFileOutput(FILE_NAME,
Context.MODE_PRIVATE, KEY_PAIR_ALIAS);


// Write data to the file, where DATA is a String of sensitive information.
outputStream.write(DATA.getBytes(StandardCharsets.UTF_8));
outputStream.flush();
outputStream.close();


// Read a sensitive data file


 secureContext.openEncryptedFileInput(FILE_NAME, Executors.newSingleThreadExecutor(),
inputStream -> {
   byte[] clearText = new byte[inputStream.available()];
   inputStream.read(encodedData);
   inputStream.close();
  // do something with the decrypted data
});
```

NOTE: Built using the JCE libraries. For more information see the following resources:

- AndroidKeyStore – developer.android.com/training/articles/keystore.
- BiometricPrompt – developer.android.com/reference/android/hardware/biometrics/BiometricPrompt

10.0 API Specification

This section provides a list of the evaluated cryptographic APIs that developers can use when writing their mobile applications.

- 10.1 Cryptographic APIs - this section lists the APIs for the algorithms and random number generation.
- 10.2 Key Management - this section lists the APIs for importing, using, and destroying keys.
- 10.3 FCS_TLSC_EXT.1 - Certificate Validation, TLS, HTTPS - this section lists the following:
  - APIs used by applications for configuring the reference identifier.
  - APIs for validation checks (should match the test program provided).
  - TLS, HTTPS, Bluetooth BR/EDR, BLE (any other protocol available to applications).

10.1 Cryptographic APIs

This section includes code samples for encryption and decryption, including random number generation.

10.1.1 Code Examples

```
// Data to encrypt
byte[] clearText = "Secret Data".getBytes(StandardCharsets.UTF_8);
// Create a Biometric Support object to handle key authentication
BiometricSupport biometricSupport = new BiometricSupportImpl(activity,
getApplicationContext()) {
   …
```

```
};
SecureCipher secureCipher = SecureCipher.getDefault(biometricSupport);
secureCipher.encryptSensitiveData("niapKey", clearText, new
SecureCipher.SecureSymmetricEncryptionCallback() {
   @Override
   public void encryptionComplete(byte[] cipherText, byte[] iv) {
       // Do something with the encrypted data
   }
});
// to decrypt
secureCipher.decryptSensitiveData("niapKey", cipherText, iv, new
SecureCipher.SecureDecryptionCallback() {
   @Override

   public void decryptionComplete(byte[] clearText) {
       // do something with the encrypted data
   }
});
// Generate ephemeral key (random number generation)
int keySize = 256;
SecureRandom secureRandom = SecureRandom.getInstanceStrong();
byte[] key = new byte[keySize / 8];
secureRandom.nextBytes(key);
// Encrypt / decrypt data with the ephemeral key
EphemeralSecretKey ephemeralSecretKey = new EphemeralSecretKey(key,
SecureConfig.getStrongConfig());
Pair<byte[], byte[]> ephemeralCipherText =
secureCipher.encryptEphemeralData(ephemeralSecretKey, clearText);
byte[] ephemeralClearText = secureCipher.decryptEphemeralData(ephemeralSecretKey,
ephemeralCipherText.first, ephemeralCipherText.second);
```

10.1.2 SecureCipher

NOTE: SecureCipher is included in the NIAPSEC library.

SecureCipher handles low-level cryptographic operations including encryption and decryption. For sensitive data protection, this library is not used directly by developers.

Supported Algorithms

- AES256 - AES/GCM/NoPadding
- RSA3072 - RSA/ECB/OAEPWithSHA-256AndMGF1Padding

Table 7 SecureCipher Public Static Accessors

Accessor	Description
SecureCipher	SecureCipher.getDefault(BiometricSupport) See BiometricSupport API to get an instance of the SecureCipher with Biometric support.

Table 8 SecureCipher Public Methods

Method	Description
void	encryptSensitiveData (String keyAlias, byte[] clearData, SecureSymmetricEncryptionCallback callback) Encrypt sensitive data using the symmetric algorithm specified by the default configuration, using NIAP standards. See SecureConfig.getStrongConfig() - Default is AES256 GCM. -keyAlias - Encrypt data with the AndroidKeyStore key referenced -clearData - the data to be encrypted -callback, the callback to return the cipherText after encryption is complete.
void	encryptSensitiveDataAsymmetric (String keyAlias, byte[] clearData, SecureAsymmetricEncryptionCallback callback) Encrypt sensitive data using the asymmetric algorithm specified by the default configuration, using NIAP standards. See SecureConfig.getStrongConfig() - Default is RSA3072 with OAEP. -keyAlias - Encrypt data with the AndroidKeyStore key referenced -clearData - the data to be encrypted -callback, the callback to return the cipherText after encryption is complete.
Pair<byte[], byte[]>	encryptEphemeralData (EphemeralSecretKey ephemeralSecretKey, byte[] clearData) Encrypt data with an Ephemeral AES 256 GCM key, used for encrypting file data for SDP. -The Ephemeral key to use -clearData, the data to be encrypted Returns a Pair of the cipherText, and IV byte arrays respectively.
void	decryptSensitiveData (String keyAlias, byte[] encryptedData, byte[] initializationVector, SecureDecryptionCallback callback) Decrypt sensitive data using the symmetric algorithm specified by the default configuration, using NIAP standards. See SecureConfig.getStrongConfig() - Default is AES256 GCM. -keyAlias - Encrypt data with the AndroidKeyStore key referenced -encryptedData - the data to be decrypted -initializationVector - the IV used for encryption -callback, the callback to return the clearText after decryption is complete.
void	decryptSensitiveData (String keyAlias, byte[] encryptedData, SecureDecryptionCallback callback) Decrypt sensitive data using the asymmetric algorithm specified by the default configuration, using NIAP standards. See SecureConfig.getStrongConfig() - Default is RSA3072 with OAEP. -keyAlias - Encrypt data with the AndroidKeyStore key referenced -encryptedData - the data to be decrypted -callback, the callback to return the clearText after decryption is complete.
byte[]	decryptEphemeralData (EphemeralSecretKey ephemeralSecretKey, byte[] encryptedData, byte[] initializationVector) Decrypt data with an Ephemeral AES 256 GCM key, used for encrypting file data for SDP. -The Ephemeral key to use -encryptedData - the data to be decrypted -initializationVector - the IV used for encryption Returns a byte array of the clear text.

NOTE: Built using the JCE libraries. For more information see the following resources:

- AndroidKeyStore – developer.android.com/training/articles/keystore
- Cipher – developer.android.com/reference/javax/crypto/Cipher
- SecretKey – developer.android.com/reference/javax/crypto/SecretKey
- SecureRandom – developer.android.com/reference/java/security/SecureRandom
- BiometricPrompt – developer.android.com/reference/android/hardware/biometrics/BiometricPrompt

10.1.3 FCS_CKM.2(1) – Key Establishment (RSA)

Assume that Alice knows a private key and Bob knows Alice's public key. Bob sent a key encrypted by the public key. This example shows how Alice gets a plain key sent by Bob. Alice needs her own private key to decrypt an encrypted key.

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA", "AndroidOpenSSL");
keyGen.initialize(keySize);
KeyPair keyPair = keyGen.generateKeyPair();
RSAPublicKey pub = (RSAPublicKey) keyPair.getPublic();
RSAPrivateCrtKey priv = (RSAPrivateCrtKey) keyPair.getPrivate();


// Encrypt
Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPWithSHA-256AndMGF1Padding");
cipher.init(Cipher.ENCRYPT_MODE, publicKey, new OAEPParameterSpec("SHA-256",
       "MGF1", new MGF1ParameterSpec("SHA-1"), PSource.PSpecified.DEFAULT));
byte[] cipherText = cipher.doFinal(data.getBytes(StandardCharsets.UTF_8));


// Decrypt
Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPWithSHA-256AndMGF1Padding");
cipher.init(Cipher.DECRYPT_MODE, privateKey, new OAEPParameterSpec("SHA-256",
       "MGF1", new MGF1ParameterSpec("SHA-1"), PSource.PSpecified.DEFAULT));
Byte[] plainText = cipher.doFinal(cipherText);
```

Algorithms

```
RSA/ECB/OAEPWithSHA-256AndMGF1Padding
```

Reference

Cipher – developer.android.com/reference/javax/crypto/Cipher

10.1.4 FCS_CKM.2(1) – Key Establishment (ECDSA) & FCS_COP.1(3) – Signature Algorithms (ECDSA)

Assume that Alice knows a private key and Bob's public key. Bob knows his private key and Alice's public key. Alice and Bob can then sign and verify the contents of a message.

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("EC", "AndroidOpenSSL");
ECGenParameterSpec ecParams = new ECGenParameterSpec(spec);
keyGen.initialize(ecParams);
KeyPair keyPair = keyGen.generateKeyPair();
ECPublicKey pubKey = (ECPublicKey) keyPair.getPublic();
ECPrivateKey privKey = (ECPrivateKey) keyPair.getPrivate();


// Sign
Signature signature = Signature.getInstance(algorithm);
signature.initSign(privateKey);
signature.update(data.getBytes(StandardCharsets.UTF_8));
byte[] signature =  signature.sign();


// Verify
Signature signature = Signature.getInstance(algorithm);
signature.initVerify(publicKey);
```

```
signature.update(data.getBytes(StandardCharsets.UTF_8));
boolean verified =  signature.verify(sig);
```

Algorithms
- "SHA256withECDSA", "secp256r1"
- "SHA384withECDSA", "secp384r1"

Reference
Signature – developer.android.com/reference/java/security/Signature

10.1.5 FCS_CKM.1 – Key Generation (ECDSA)

Assume that Alice knows a private key and Bob's public key. Bob knows his private key and Alice's public key.

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("EC", "AndroidOpenSSL");
ECGenParameterSpec ecParams = new ECGenParameterSpec(spec);
keyGen.initialize(ecParams);
KeyPair keyPair = keyGen.generateKeyPair();
ECPublicKey pubKey = (ECPublicKey) keyPair.getPublic();
ECPrivateKey privKey = (ECPrivateKey) keyPair.getPrivate();
```

Algorithms
- "SHA256withECDSA", "secp256r1"
- "SHA384withECDSA", "secp384r1"

Reference
Signature – developer.android.com/reference/java/security/Signature

10.1.6 FCS_COP.1(1) – Encryption/Decryption (AES)

Cipher class encrypts or decrypts a plain text.

```
KeyGenerator keyGenerator = KeyGenerator.getInstance("AES", "AndroidOpenSSL");
keyGenerator.init(keySize);
SecretKey key = keyGenerator.generateKey();

// Encrypt
Cipher cipher = Cipher.getInstance(transformation);
cipher.init(Cipher.ENCRYPT_MODE, secretKey);
byte[] iv = cipher.getIV();
byte[] clearData = data.getBytes(UTF_8);
byte[] cipherText = cipher.doFinal(clearData);
Pair<byte[], byte[]> result =  Pair<>(cipherText, iv);

// Decrypt
Cipher cipher = Cipher.getInstance(transformation);
cipher.init(Cipher.DECRYPT_MODE, secretKey, spec);
String plainText =  new String(cipher.doFinal(cipherText), UTF_8);
```

Algorithms
- AES/CBC/NoPadding
- AES/GCM/NoPadding

Reference
Cipher – developer.android.com/reference/javax/crypto/Cipher

10.1.7 FCS_COP.1(2) – Hashing (SHA)

You can use MessageDigest class to calculate the hash of plaintext.

```
MessageDigest messageDigest = MessageDigest.getInstance(algorithm);
messageDigest.update(data.getBytes(StandardCharsets.UTF_8));
byte[] digest = messageDigest.digest();
```

Algorithms

- SHA-1
- SHA-256
- SHA-384
- SHA-512

Reference

MessageDigest – developer.android.com/reference/java/security/MessageDigest

10.1.8 FCS_COP.1(3) – RSA (Signature Algorithms)

KeyFactory class generates RSA private key and public key. Signature class signs a plaintext with private key generated above and verifies it with public key.

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA", "AndroidOpenSSL");
keyGen.initialize(keySize);
KeyPair keyPair = keyGen.generateKeyPair();
RSAPublicKey pub = (RSAPublicKey) keyPair.getPublic();
RSAPrivateCrtKey priv = (RSAPrivateCrtKey) keyPair.getPrivate();


// Sign
Signature signature = Signature.getInstance(algorithm);
signature.initSign(privateKey);
signature.update(data.getBytes(StandardCharsets.UTF_8));
byte[] sig =  signature.sign();


// Verify
Signature signature = Signature.getInstance(algorithm);
signature.initVerify(publicKey);
signature.update(data.getBytes(StandardCharsets.UTF_8));
boolean verified =  signature.verify(sig);
```

Algorithms

- SHA256withRSA
- SHA384withRSA

Reference

Signature – developer.android.com/reference/java/security/Signature

10.1.9 FCS_CKM.1 – Key Generation (RSA)

KeyFactory class generates RSA private key and public key.

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA", "AndroidOpenSSL");
keyGen.initialize(keySize);
KeyPair keyPair = keyGen.generateKeyPair();
RSAPublicKey pub = (RSAPublicKey) keyPair.getPublic();
RSAPrivateCrtKey priv = (RSAPrivateCrtKey) keyPair.getPrivate();
```

Algorithms

- SHA256withRSA
- SHA384withRSA

Reference

Signature – developer.android.com/reference/java/security/Signature

10.1.10 FCS_COP.1(4) - HMAC

Mac class calculates the hash of plaintext with key.

```
KeyGenerator keyGenerator = KeyGenerator.getInstance(
       algorithm, "AndroidOpenSSL");
keyGenerator.init(keySize);
SecretKey key = keyGenerator.generateKey();


// Mac
Mac mac = Mac.getInstance(algorithm);
mac.init(secretKey);
byte[] mac =  mac.doFinal(data.getBytes(StandardCharsets.UTF_8));
```

Algorithms

- HmacSHA1
- HmacSHA256
- HmacSHA384
- HmacSHA512

Reference

Mac – developer.android.com/reference/javax/crypto/Mac

10.2 Key Management

This section provides code samples for key management.

10.2.1 Code examples:

```
SecureKeyGenerator keyGenerator = SecureKeyGenerator.getInstance();
// Generate Keypair
keyGenerator.generateAsymmetricKeyPair(KEY_PAIR_ALIAS);
// Generate Symmetric Key


keyGenerator.generateKey(KEY_ALIAS);


// Generate ephemeral key (random number generation)
keyGenerator.generateEphemeralDataKey();


// To delete a key stored in the Android Keystore
KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
keyStore.load(null);
keyStore.deleteEntry("KEY_TO_REMOVE");
```

10.2.2 SecureKeyGenerator

NOTE: SecureKeyGenerator is included in the NIAPSEC library.

SecureKeyGenerator handles low-level key generation operations using the AndroidKeyStore. For sensitive data protection, this library is not used directly by developers.

Supported Algorithms

- AES256 - AES/GCM/NoPadding
- RSA3072 - RSA/ECB/OAEPWithSHA-256AndMGF1Padding

Table 9 SecureKeyGenerator Public Static Accessories

Accessors	Description
SecureKeyGenerator	SecureCipher.getDefault() API to get an instance of the SecureCipher with NIAP settings.

Table 10 SecureKeyGenerator Public Methods

Methods	Description
boolean	generateKey(String keyAlias) Generate an AES key with NIAP settings that is stored and protected in the AndroidKeyStore. See SecureConfig.getStrongConfig() - Default is AES256 GCM. -keyAlias - name for the key
boolean	generateKeyAsymmetricKeyPair(String keyAlias) Generate an RSA key pair with NIAP settings that is stored and protected in the AndroidKeyStore. See SecureConfig.getStrongConfig() - Default is RSA3072 OAEP. -keyAlias - name for the key pair
EphemeralSecretKey	generateEphemeralDataKey() Generate an AES key with NIAP settings. This key is not stored in the AndroidKeyStore Uses SecureRandom.getInstanceStrong() to generate a random key. See SecureConfig.getStrongConfig() - Default is AES256 GCM.

NOTE: Built using the JCE libraries. For more information see the following resources:

- AndroidKeyStore – developer.android.com/training/articles/keystore
- KeyPairGenerator – developer.android.com/reference/java/security/KeyPairGenerator
- SecretKey – developer.android.com/reference/javax/crypto/SecretKey
- SecureRandom – developer.android.com/reference/java/security/SecureRandom
- KeyGenParameterSpec – developer.android.com/reference/android/security/keystore/KeyGenParameterSpec

10.3 FCS_TLSC_EXT.1 - Certificate Validation, TLS, HTTPS

NOTE: SecureURL is included in the NIAPSEC library.

SecureURL automatically configures TLS and can perform certificate and host validation checking. At construction, SecureURL requires a reference identifier.

Code examples:

```
SecureURL url = new SecureURL(referenceIdentifier, "google_cert");
HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
conn.setRequestMethod("GET");
conn.setDoInput(true);
conn.connect();

// Manual check
SecureURL url = new SecureURL(referenceIdentifier, "google_cert");
boolean valid = url.isValid(urlConnection);
```

Table 11 SecureURL Public Constructors

Constructors	Description
SecureURL	new SecureURL(String referenceIdentifier, String clientCert) API to create an instance of the SecureURL with NIAP settings. clientCert is optional.

Table 12 SecureURL Public Methods

Methods	Description
HttpsURLConnection	openConnection Opens an HttpsUrlConnection using TLS by default and handles OCSP validation checks and does a hostname verification check on initiation of the connection.
boolean	isValid(String hostname, SSLSocket socket) A manual OCSP certificate and hostname check. Based on a hostname and underlying SSLSocket.
boolean	isValid(HttpsURLConnection conn) A manual OCSP certificate and hostname check. Based on an existing HttpsUrlConnection.
boolean	isValid(Certificate cert) A manual OCSP certificate check.
boolean	isValid(List<Certificate> certs) A manual OCSP certificates check.

NOTE: Built using the networking libraries. For more information see the following resources:

- PKIXRevocationChecker – developer.android.com/reference/java/security/cert/PKIXRevocationChecker
- SSLSocket – developer.android.com/reference/javax/net/ssl/SSLSocket

10.3.1 Cipher Suites

By default, the device is restricted to only support TLS Ciphersuites that are RFC compliant and can be claimed under MDFPP. As such, no configuration is needed to restrict or allow ciphersuites to be compliant. A list of the ciphersuites supported by Android 15 can be found below:

Table 13 TLS 1.2 Cipher Suites

Approved Cipher Suites	TLS Version
TLS_RSA_WITH_AES_128_GCM_SHA256 as defined in RFC 5288,	TLS v1.2
TLS_RSA_WITH_AES_256_GCM_SHA384 as defined in RFC 5288,	
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 as defined in RFC 5289,	
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 as defined in RFC 5289,	
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 as defined in RFC 5289,	
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 as defined in RFC 5289	

The device supports TLS versions 1.1 and 1.2 for use with EAP-TLS as part of WPA2 and WPA3. The TOE supports the following cipher suites for this:

- TLS_RSA_WITH_AES_128_CBC_SHA as defined in RFC 5246,
- TLS_RSA_WITH_AES_128_GCM_SHA256 as defined in RFC 5288,
- TLS_RSA_WITH_AES_256_GCM_SHA384 as defined in RFC 5288,
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 as defined in RFC 5289,
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 as defined in RFC 5289,
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 as defined in RFC 5289,
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 as defined in RFC 5289

10.3.2 Guidance for Bluetooth Low Energy APIs

Provides classes that manage Bluetooth functionality, such as scanning for devices, connecting with devices, and managing data transfer between devices. The Bluetooth API supports both Classic Bluetooth and Bluetooth Low Energy (BLE).

For more information about Classic Bluetooth, see the Android Bluetooth guide. For more information about Bluetooth Low Energy, see the Android Bluetooth Low Energy (BLE) guide.

The Bluetooth APIs allow applications to do the following:

- Scan for other Bluetooth devices (including BLE devices).
- Query the local Bluetooth adapter for paired Bluetooth devices.
- Establish RFCOMM channels/sockets.
- Connect to specified sockets on other devices.
- Transfer data to and from other devices.
- Communicate with BLE devices, such as proximity sensors, heart rate monitors, and fitness devices.
- Act as a GATT client or a GATT server (BLE).

To perform Bluetooth communication using these APIs, an application must declare the BLUETOOTH permission. Some additional functionality, such as requesting device discovery, also requires the BLUETOOTH_ADMIN permission.

Table 14 Bluetooth Interfaces

Interface	Description
BluetoothAdapter.LeScanCallback	Callback interface used to deliver LE scan results.
BluetoothProfile	Public APIs for the Bluetooth Profiles.
BluetoothProfile.ServiceListener	An interface for notifying BluetoothProfile IPC clients when they have been connected or disconnected to the service.

Table 15 Bluetooth Classes

Class	Description
BluetoothA2dp	This class provides the public APIs to control the Bluetooth A2DP profile.
BluetoothAdapter	Represents the local device Bluetooth adapter.
BluetoothAssignedNumbers	Bluetooth Assigned Numbers.
BluetoothClass	Represents a Bluetooth class, which describes general characteristics and capabilities of a device.
BluetoothClass.Device	Defines all device class constants.
BluetoothClass.Device.Major	Defines all major device class constants.
BluetoothClass.Service	Defines all service class constants.
BluetoothDevice	Represents a remote Bluetooth device.
BluetoothGatt	Public API for the Bluetooth GATT Profile.
BluetoothGattCallback	This abstract class is used to implement BluetoothGatt callbacks.
BluetoothGattCharacteristic	Represents a Bluetooth GATT Characteristic A GATT characteristic is a basic data element used to construct a GATT service,BluetoothGattService.
BluetoothGattDescriptor	Represents a Bluetooth GATT Descriptor GATT Descriptors contain additional information and attributes of a GATT characteristic, BluetoothGattCharacteristic.
BluetoothGattServer	Public API for the Bluetooth GATT Profile server role.
BluetoothGattServerCallback	This abstract class is used to implement BluetoothGattServer callbacks.
BluetoothGattService	Represents a Bluetooth GATT Service Gatt Service contains a collection of BluetoothGattCharacteristic, as well as referenced services.
BluetoothHeadset	Public API for controlling the Bluetooth Headset Service.

Table 16 Bluetooth Classes (Continued)

Class	Description
BluetoothHealth	This class was deprecated in API level 29. Health Device Profile (HDP) and MCAP protocol are no longer used. New apps should use Bluetooth Low Energy based solutions such as BluetoothGatt,BluetoothAdapter#listenUsingL2capChannel(), orBluetoothDevice#createL2capChannel(int)
BluetoothHealthAppConfiguration	This class was deprecated in API level 29. Health Device Profile (HDP) and MCAP protocol are no longer used. New apps should use Bluetooth Low Energy based solutions such as BluetoothGatt,BluetoothAdapter#listenUsingL2capChannel(), orBluetoothDevice#createL2capChannel(int)
BluetoothHealthCallback	This class was deprecated in API level 29. Health Device Profile (HDP) and MCAP protocol are no longer used. New apps should use Bluetooth Low Energy based solutions such as BluetoothGatt,BluetoothAdapter#listenUsingL2capChannel(), orBluetoothDevice#createL2capChannel(int)
BluetoothHearingAid	This class provides the public APIs to control the Hearing Aid profile.
BluetoothHidDevice	Provides the public APIs to control the Bluetooth HID Device profile.
BluetoothHidDevice.Callback	The template class that applications use to call callback functions on events from the HID host.
BluetoothHidDeviceAppQosSettings	Represents the Quality of Service (QoS) settings for a Bluetooth HID Device application.
BluetoothHidDeviceAppSdpSettings	Represents the Service Discovery Protocol (SDP) settings for a Bluetooth HID Device application.
BluetoothManager	High level manager used to obtain an instance of an BluetoothAdapter and to conduct overall Bluetooth Management.
BluetoothServerSocket	A listening Bluetooth socket.
BluetoothSocket	A connected or connecting Bluetooth socket.

For more information, see developer.android.com/reference/android/bluetooth/package-summary.html.

How to connect and pair with a bluetooth device:

```
// get bluetooth adapter
BluetoothAdapter bluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
if (bluetoothAdapter == null) {
    // Device doesn't support Bluetooth
}
// make sure bluetooth is enabled
if (!bluetoothAdapter.isEnabled()) {
    Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
```

```
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
}
// query for devices
Set<BluetoothDevice> pairedDevices = bluetoothAdapter.getBondedDevices();
if (pairedDevices.size() > 0) {
    // There are paired devices. Get the name and address of each paired device.
    for (BluetoothDevice device : pairedDevices) {
        String deviceName = device.getName();
        String deviceHardwareAddress = device.getAddress(); // MAC address
    }
}
// Connect to devices.
private class AcceptThread extends Thread {
    private final BluetoothServerSocket mmServerSocket;
    public AcceptThread() {
        // Use a temporary object that is later assigned to mmServerSocket
        // because mmServerSocket is final.
        BluetoothServerSocket tmp = null;
        try {
            // MY_UUID is the app's UUID string, also used by the client code.
            tmp = bluetoothAdapter.listenUsingRfcommWithServiceRecord(NAME, MY_UUID);
        } catch (IOException e) {
            Log.e(TAG, "Socket's listen() method failed", e);
        }
        mmServerSocket = tmp;
    }
    public void run() {
        BluetoothSocket socket = null;
        // Keep listening until exception occurs or a socket is returned.
        while (true) {
            try {
                socket = mmServerSocket.accept();
            } catch (IOException e) {
                Log.e(TAG, "Socket's accept() method failed", e);
                break;
            }
            if (socket != null) {
                // A connection was accepted. Perform work associated with
                // the connection in a separate thread.
                manageMyConnectedSocket(socket);
                mmServerSocket.close();
                break;
            }
        }
    }
    // Closes the connect socket and causes the thread to finish.
    public void cancel() {
        try {
            mmServerSocket.close();
```

```
        } catch (IOException e) {
            Log.e(TAG, "Could not close the connect socket", e);
        }
    }
}
```

For more information, see developer.android.com/guide/topics/connectivity/bluetooth.html#SettingUp.

Sample service to interact with a bluetooth APIs:

```
// A service that interacts with the BLE device via the Android BLE API.
public class BLEService extends Service {
   private final static String TAG = "BLEService";
   private BluetoothManager mBluetoothManager;
   private BluetoothAdapter mBluetoothAdapter;
   private String  mBluetoothDeviceAddress;
   private BluetoothGatt mBluetoothGatt;
   private int mConnectionState = STATE_DISCONNECTED;
   private static final int STATE_DISCONNECTED = 0;
   private static final int STATE_CONNECTING = 1;
   private static final int STATE_CONNECTED = 2;
   public final static String ACTION_GATT_CONNECTED =
           "com.niap.ble.ACTION_GATT_CONNECTED";
   public final static String ACTION_GATT_DISCONNECTED =
           "com.niap.ble.ACTION_GATT_DISCONNECTED";
   public final static String ACTION_GATT_SERVICES_DISCOVERED =
           "com.niap.ble.ACTION_GATT_SERVICES_DISCOVERED";
   public final static String ACTION_DATA_AVAILABLE =
           "com.niap.ble.ACTION_DATA_AVAILABLE";
   public final static String EXTRA_DATA =
           "com.niap.ble.EXTRA_DATA";
   // Various callback methods defined by the BLE API.
   private final BluetoothGattCallback mGattCallback =
           new BluetoothGattCallback() {
               @Override
               public void onConnectionStateChange(BluetoothGatt gatt, int status,
                                                   int newState) {
                   String intentAction;
                   if (newState == BluetoothProfile.STATE_CONNECTED) {
                       intentAction = ACTION_GATT_CONNECTED;
                       mConnectionState = STATE_CONNECTED;
                       broadcastUpdate(intentAction);
                       Log.i(TAG, "Connected to GATT server.");
                       Log.i(TAG, "Attempting to start service discovery:" +
                               mBluetoothGatt.discoverServices());
                   } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
                       intentAction = ACTION_GATT_DISCONNECTED;
                       mConnectionState = STATE_DISCONNECTED;
                       Log.i(TAG, "Disconnected from GATT server.");
                       broadcastUpdate(intentAction);
                   }
```

```
                }
                @Override
                // New services discovered
                public void onServicesDiscovered(BluetoothGatt gatt, int status) {
                    if (status == BluetoothGatt.GATT_SUCCESS) {
                        broadcastUpdate(ACTION_GATT_SERVICES_DISCOVERED);
                    } else {
                        Log.w(TAG, "onServicesDiscovered received: " + status);
                    }
                }
                @Override
                // Result of a characteristic read operation
                public void onCharacteristicRead(BluetoothGatt gatt,
                                                 BluetoothGattCharacteristic
characteristic,
                                                 int status) {
                    if (status == BluetoothGatt.GATT_SUCCESS) {
                        broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
```

```
                    }
                }
            };
    }
```

11.0 Annexure

11.1 Creating and Applying the StageNow Profile

11.1.1 Install StageNow

Installing StageNow is dependent on the version you are using. The instructions for each version are found on the StageNow support page. Follow the instructions for the version you are installing.

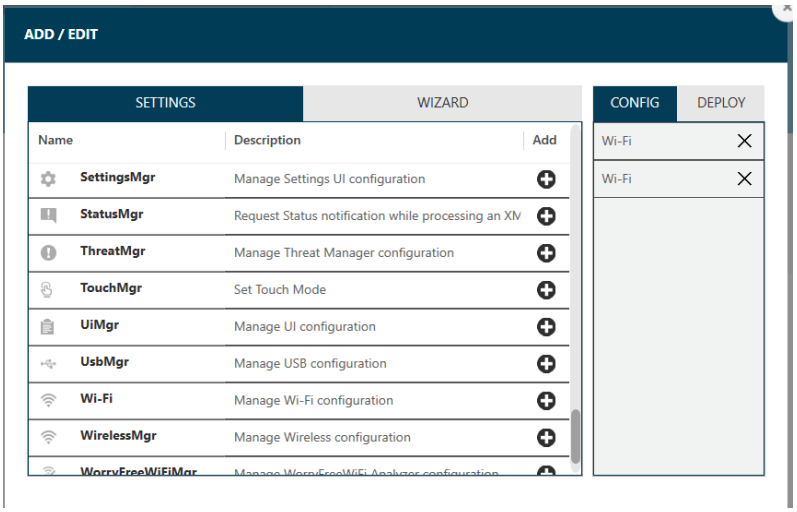
1. Install StageNow on your workstation tool from Zebra support portal at zebra.com/us/en/support-downloads/software/utilities/stagenow.html.
2. Select the version you are installing; typically you will install the newest version.
3. Click the version number to expand the options and access the Installation Guide, Release Notes, and install file. Follow the instructions in the Installation Guide to install StageNow.
4. Follow the instructions to create the StageNow Profiles. This example uses the instructions from version 5.12. https://techdocs.zebra.com/stagenow/5-12/stagingprofiles/
5. Download CCReadinesspackage_A15_<CPU>.zip from HERE.
6. Refer the steps to create Xpert Mode profiles at techdocs.zebra.com/stagenow/4-2/Profiles/xpertmode/.
7. Refer to the settings type at techdocs.zebra.com/stagenow/4-2/settingtypes/.
8. Once you are familiar with StageNow usage, continue with 11.1.2 Create the StageNow Profiles to connect the device to your network. Download CC Readiness package from StageNow workstation and update the device.

NOTE: You can use the profiles attached at the end of this document and import to StageNow and update the network and Package details.

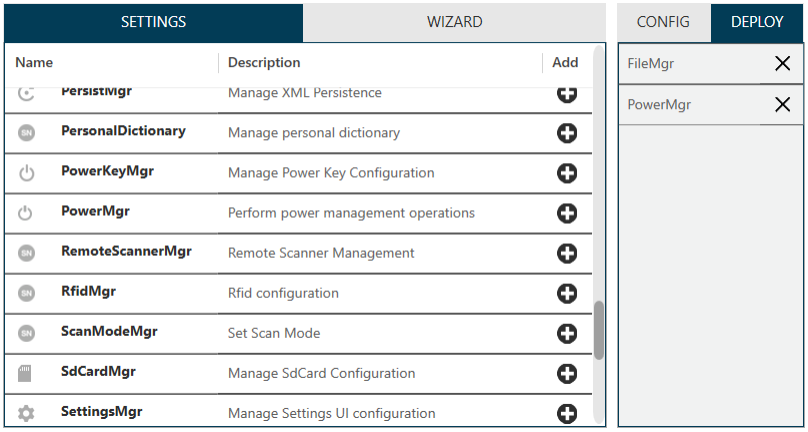
9. Navigate to the **All Profiles** page.
10. Click on **Import Profiles** to import the zip file as detailed in the following section.

11.1.2 Create the StageNow Profiles

1. Launch StageNow tool on your Workstation.
2. Login with Admin credentials.
3. From the home page, click on **Create New Profile**.
4. On the **Select a Wizard** pop up page, choose the **Please select MX version on your device** drop down, and set the value same as device MX version - example MX 14.2.Select the **Xpert Mode** wizard and then click on **Create**.
5. Enter the Profile name and then click on **Start**.
6. In the **CONFIG** section, add two Wi-Fi network settings; one for configuring the network settings and a second one to connect the device to that network.

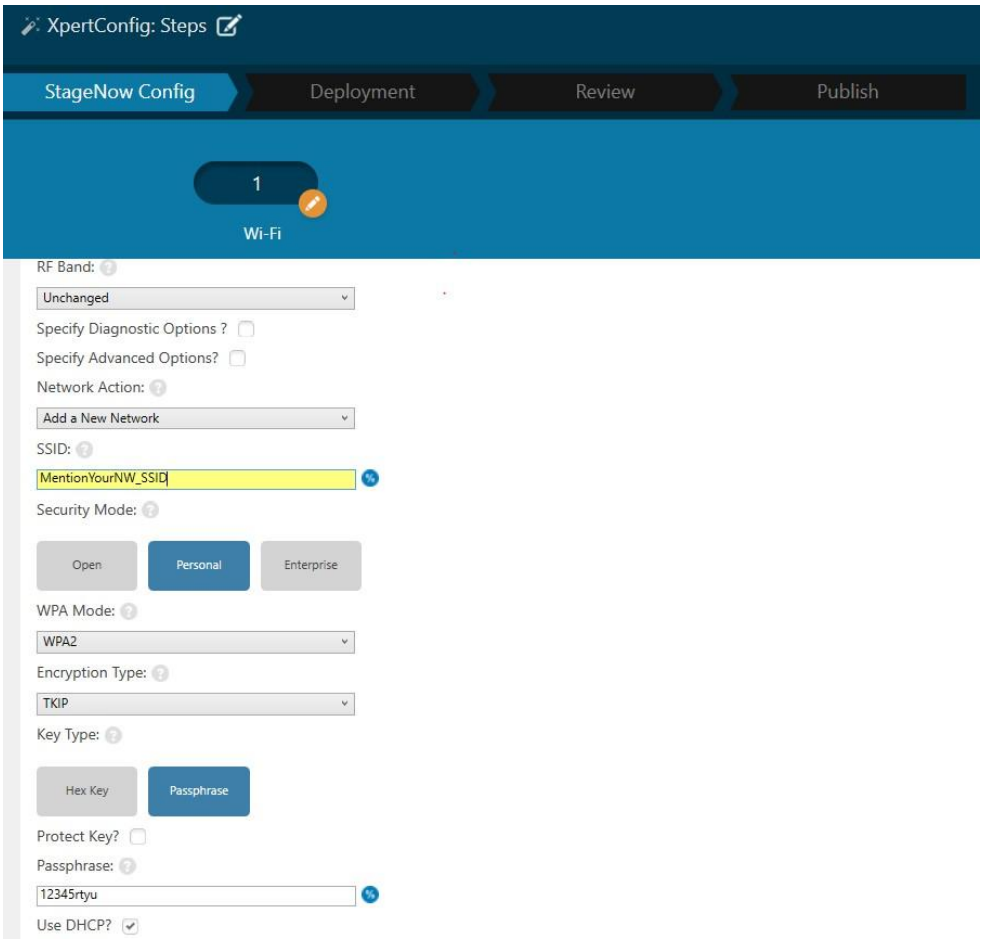


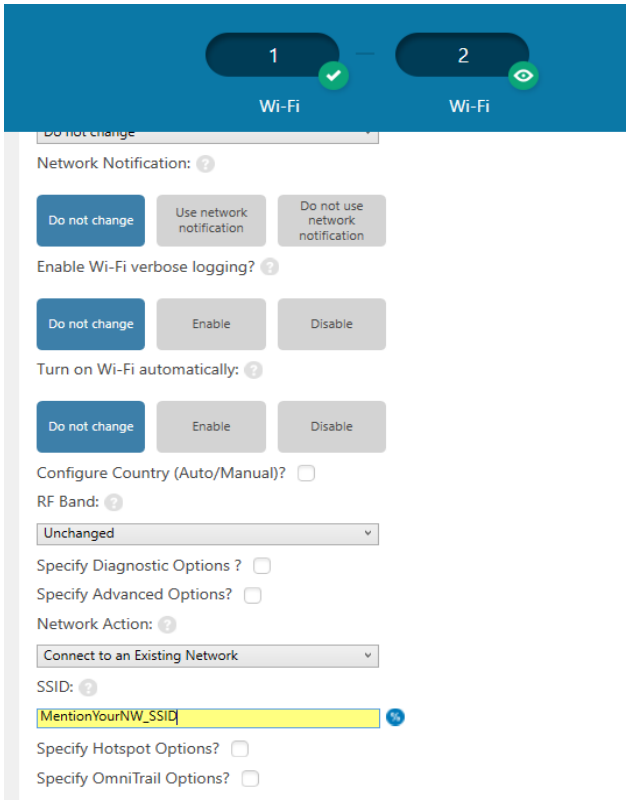
7. In the **DEPLOY** section, add FileMgr to allow settings that transfer the FBE file to the device, and PowerMgr settings to update the device OS using the FBE package.



8. Click **Update** and then enter the settings data.

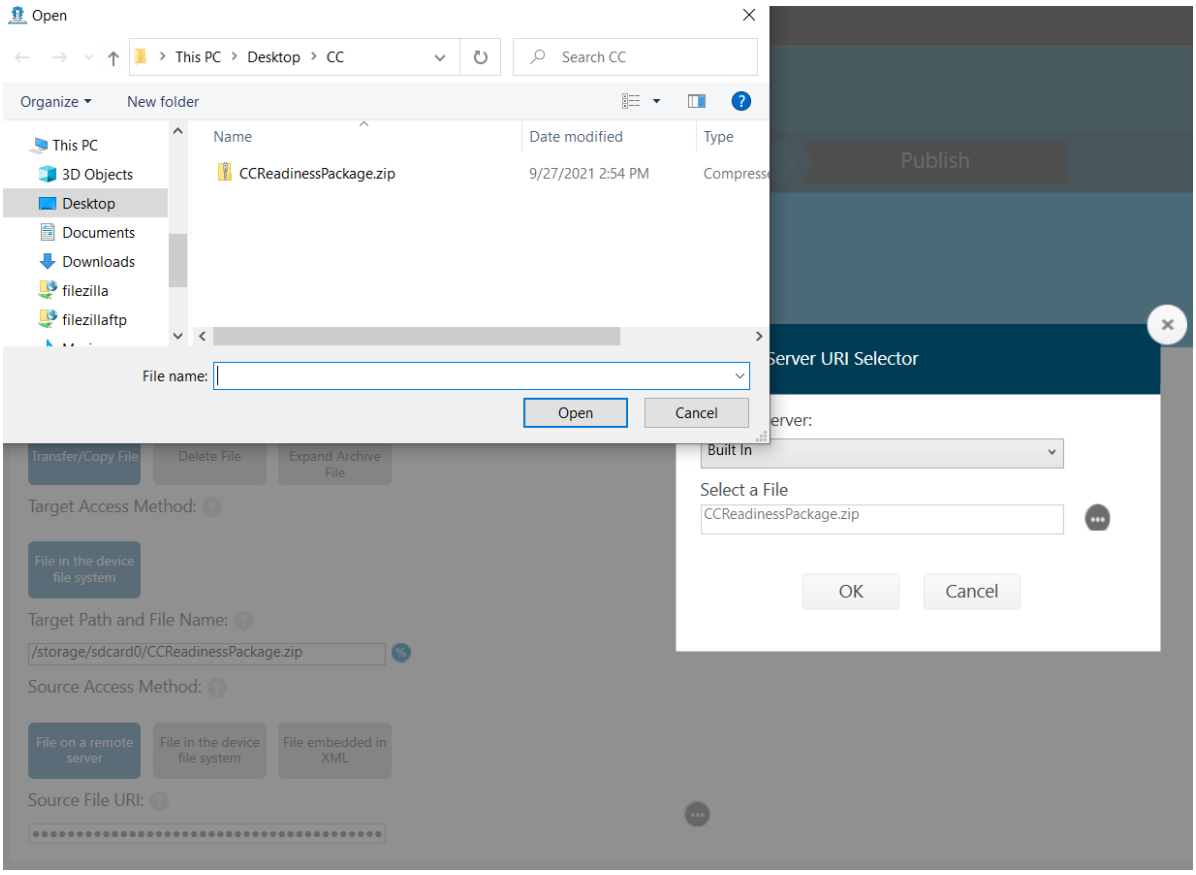
9. Enter your network details and click **Continue**.



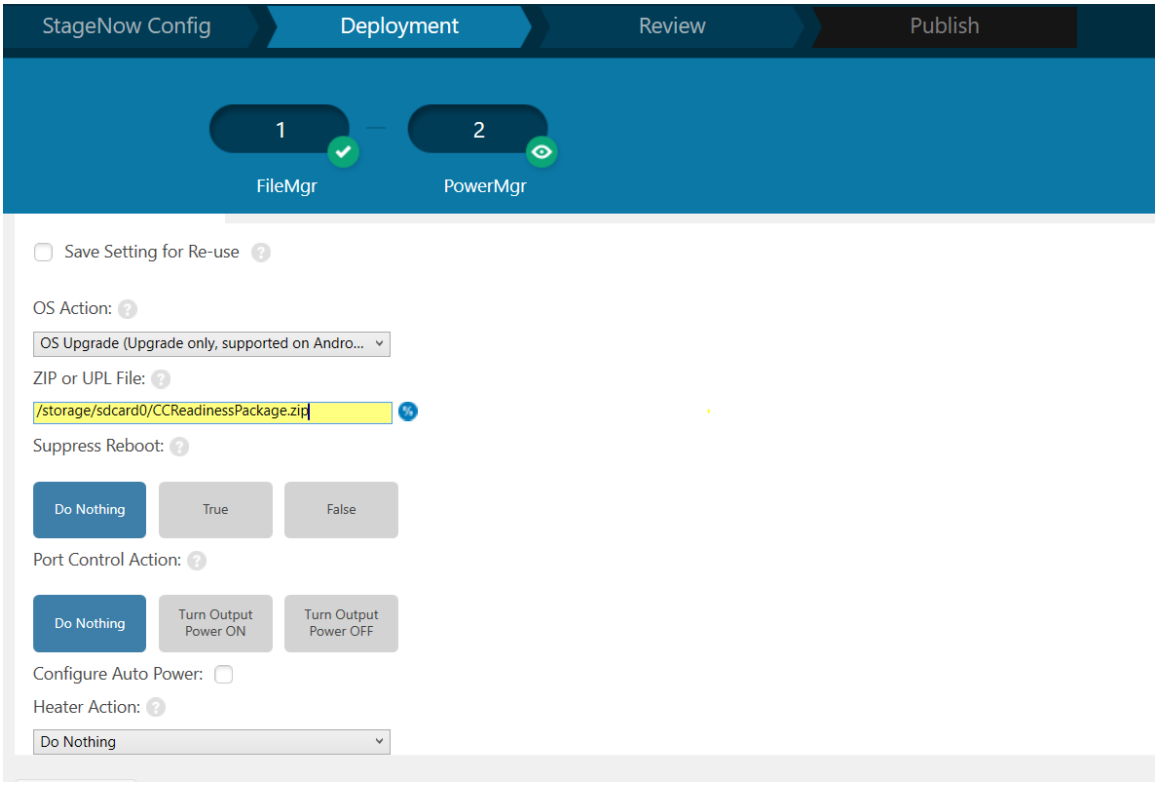


10. Enter file path details and then select the CCReadinesspackage_A15_<CPU>.zip to be downloaded to the device.

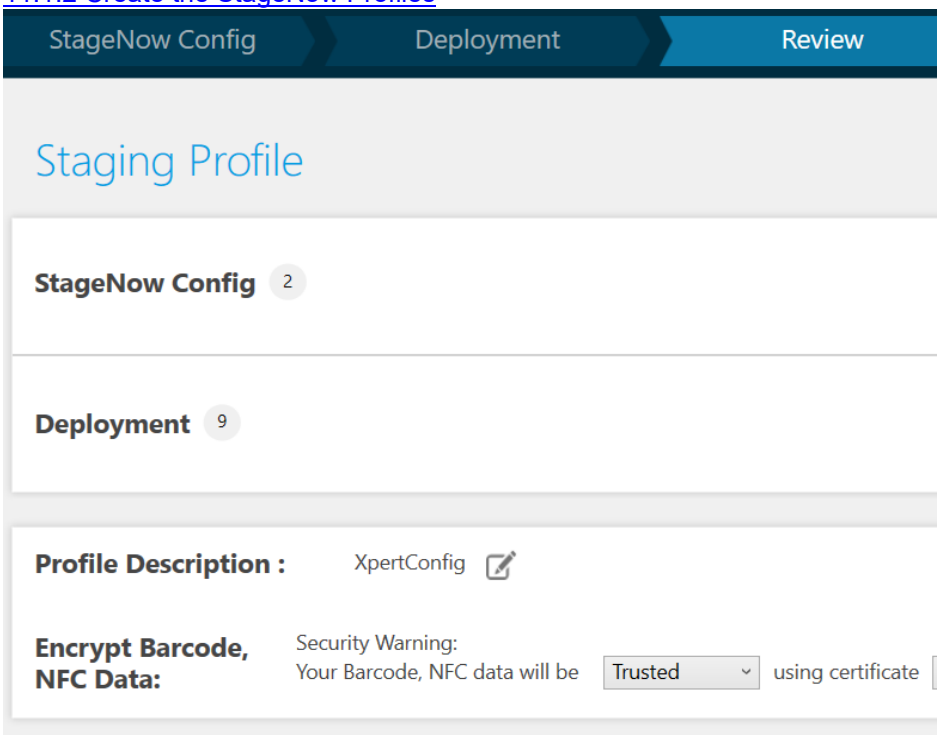
    Note:In case of Exit from CC use the file CCExitPackage_A15_<CPU>.zip or relevant package from HERE
11. Click on **Continue**.


Open
This PC
Desktop
CC
Search CC
Organize
New folder
This PC
3D Objects
Desktop
Documents
Downloads
filezilla
filezillaftp
Name
Date modified
Type
CCReadinessPackage.zip
9/27/2021 2:54 PM
File name:
Open
Cancel
Publish
Server URI Selector
Built In
Select a File
CCReadinessPackage.zip
OK
Cancel
Transfer/Copy File
Delete File
Expand Archive File
Target Access Method:
File in the device file system
Target Path and File Name:
/storage/sdcard0/CCReadinessPackage.zip
Source Access Method:
File on a remote server
File in the device file system
File embedded in XML
Source File URI:

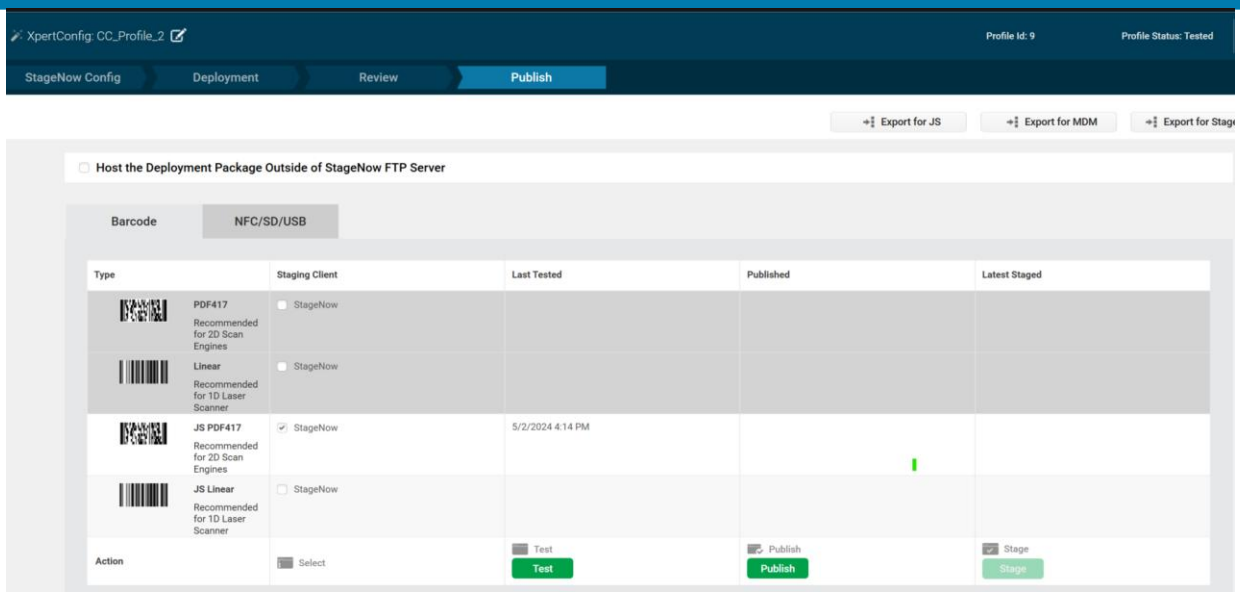
12. Enter the FBE package name in the **PowerMgr** setting to apply the patch on the device.
    - Note: Sample path for ZIP or UPL file is /sdcard/CCReadinesspackage_ A15_<CPU>.zip



13. On Review screen, Select Barcode as Trusted and opt the certificate which was imported while following [11.1.2 Create the StageNow Profiles]



14. Click on “Continue” and complete the profile
15. On the **Publish** page, select **JS PDF417** type, and then click on **Test** to generate barcodes.



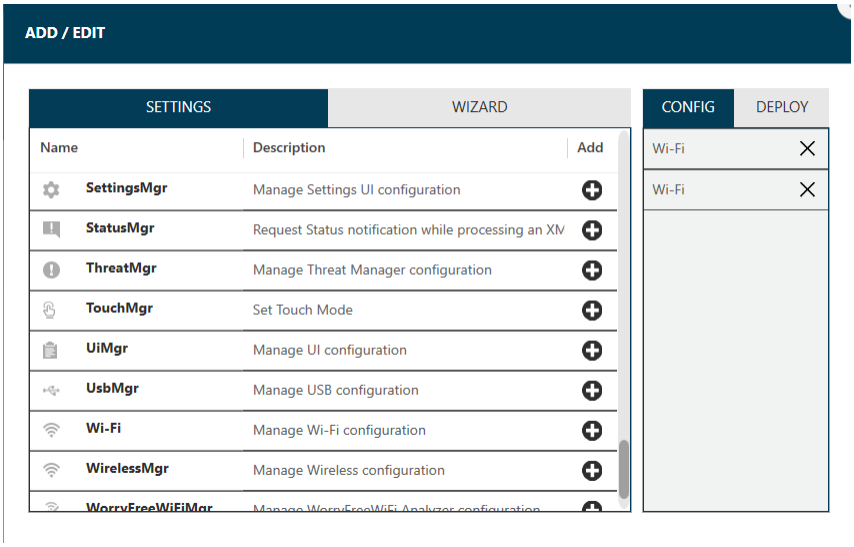
16. Do one of the following:
    - For a factory-fresh (or factory-reset) device at the Welcome screen, select FIPS-enabled device and scan the barcode(s) from the device, or
    - Launch the StageNow application from the page and then scan the barcode(s).
17. Once the device connects to the network, download the patch and then apply the patch.

11.2 Configuring Critical Settings Using Stage Now

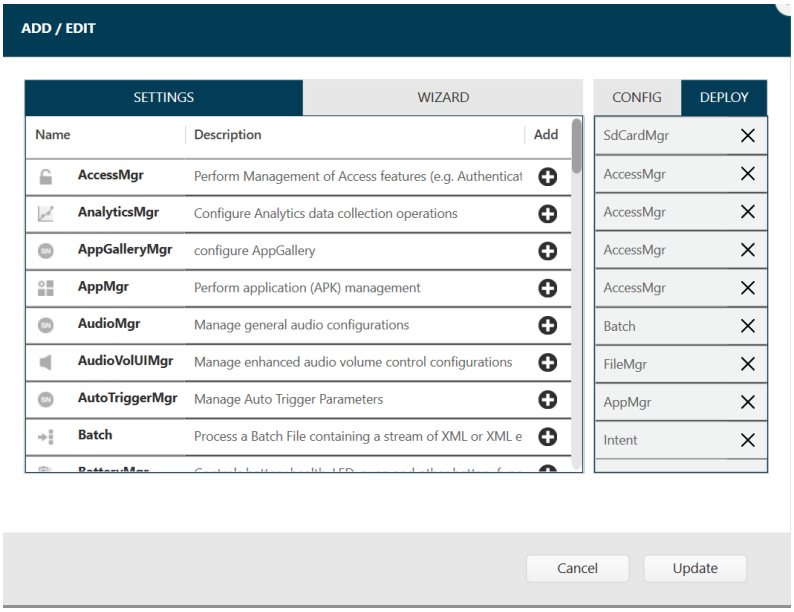
NOTES:

- For settings type details for SDCardMgr settings, see techdocs.zebra.com/stagenow/4-2/settingtypes/.
- For information on Trusted Staging, see techdocs.zebra.com/stagenow/4-2/trustedstaging/.

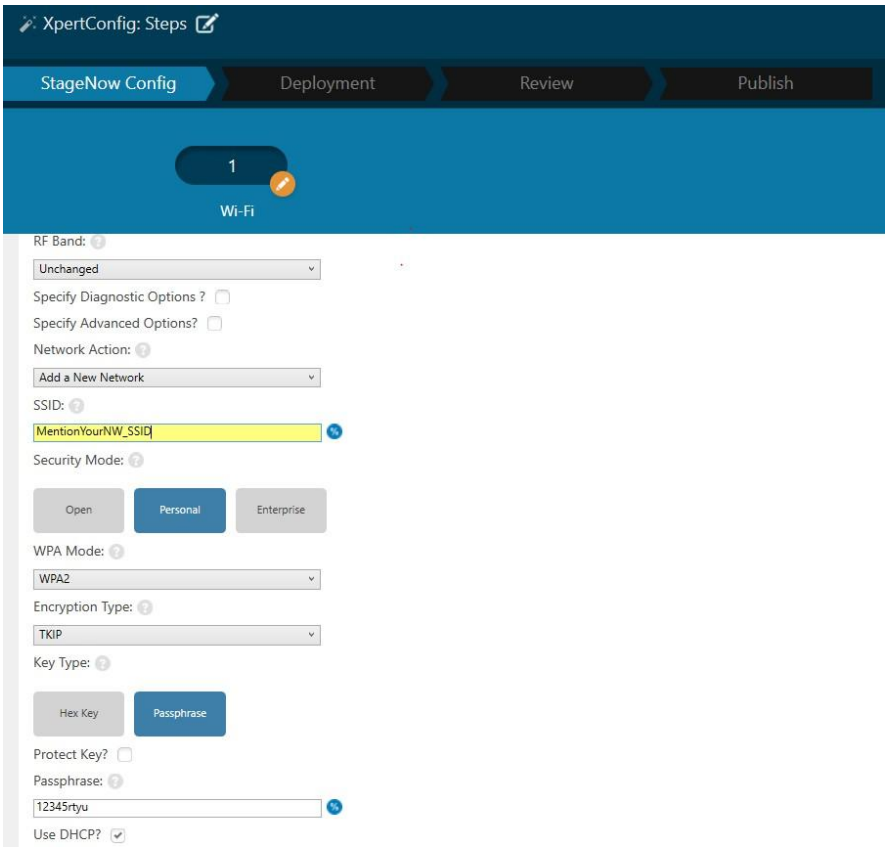
1. Refer to 11.1.2 Create the StageNow Profiles. Follow steps to create new StageNow Profile # 2 with the settings shown in the figure below. Add two Wi-Fi settings to the CONFIG section to create a network and then connect to it.

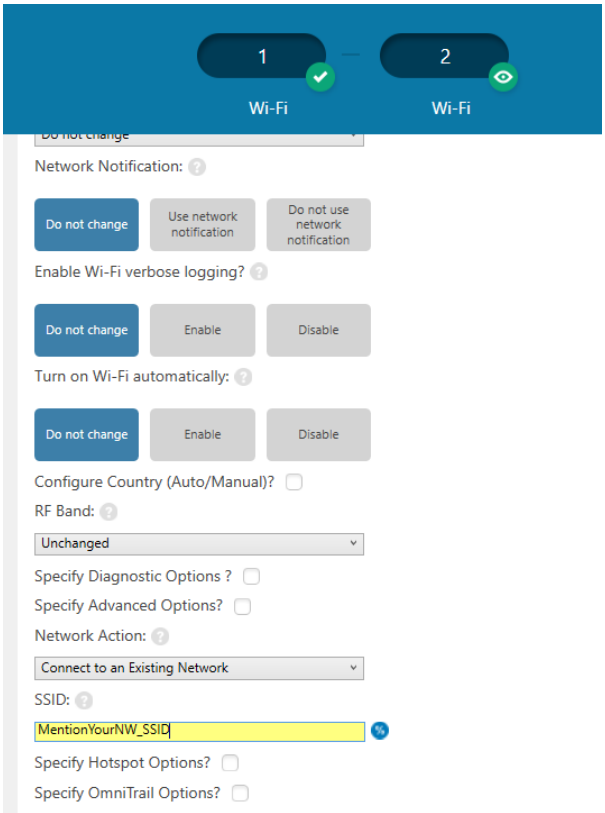


2. Add the following settings in DEPLOY Section and then click on update.
   - SDCardMgr – Disable SD card access (Unmount)
   - BatchMgr – Enable Trusted Staging
   - FileMgr – Download MDM agent to the device
   - AppMgr – Install MDM agent
   - IntentMgr – Set/enroll MDM agent as a Device Owner



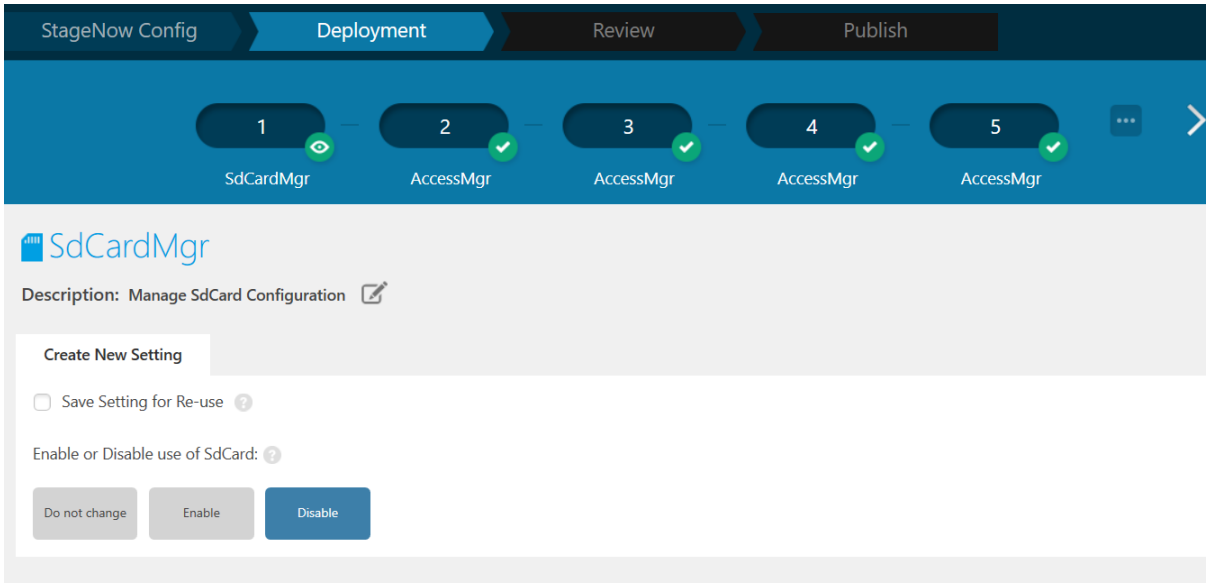
3. Enter valid data for network creation and connecting to the same network





4. Follow the steps below to configure the deployment settings, first to disable SD card.

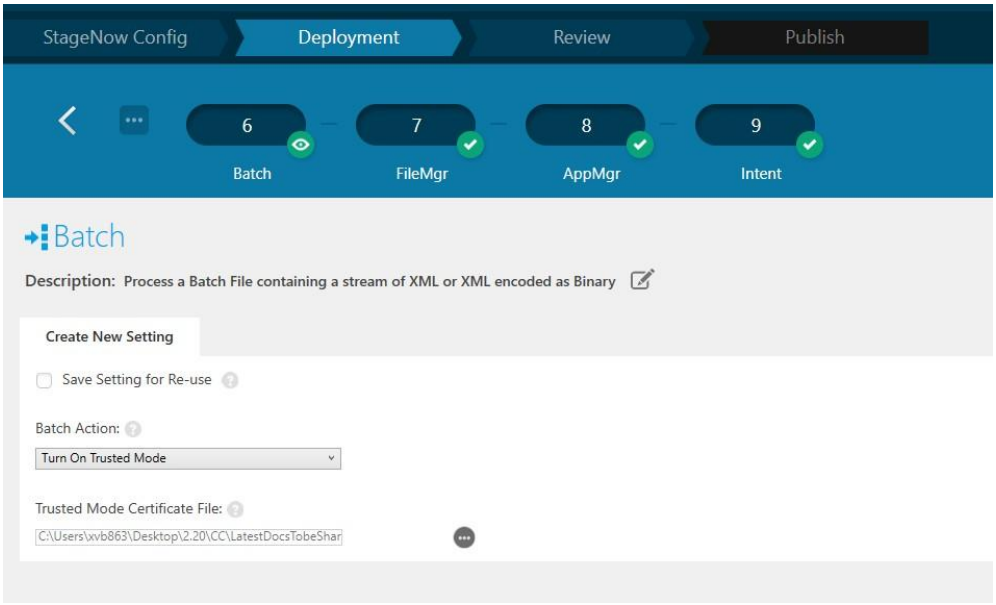
Note: Step 4 is not applicable on devices that doesn't support sdcard



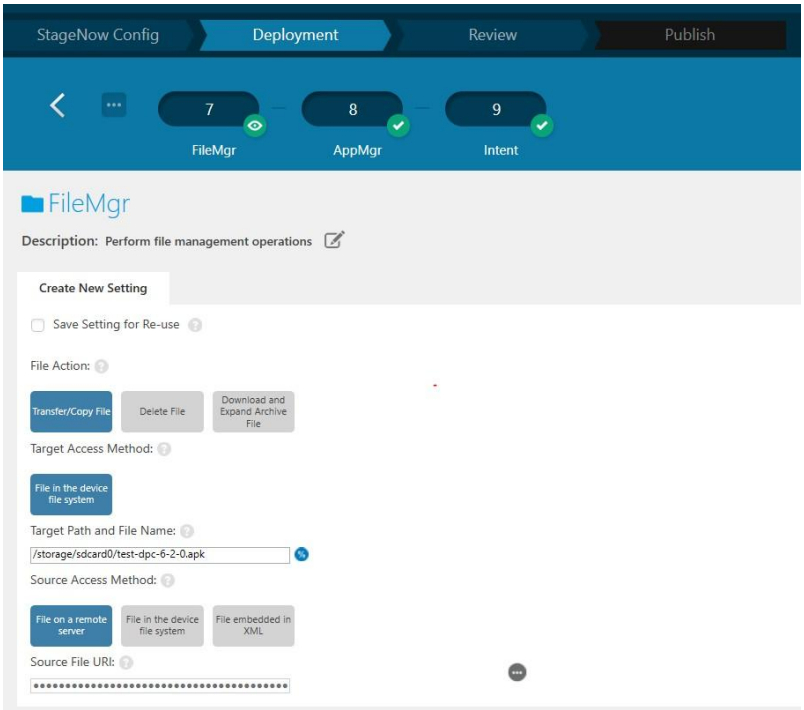
.

5. Set StageNow to Trusted Mode.

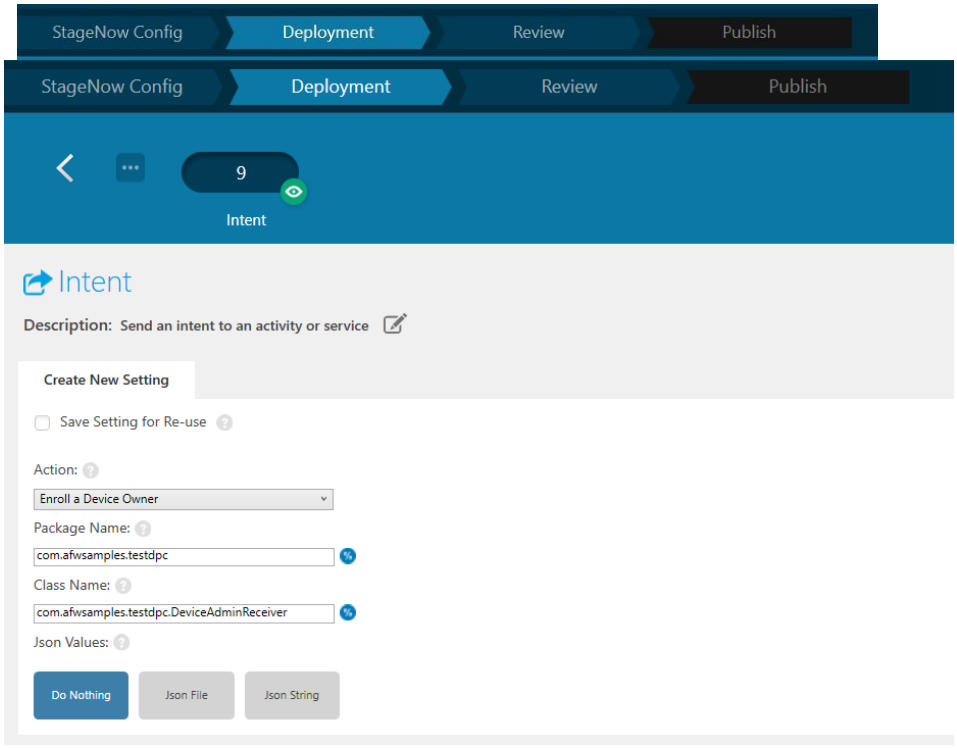
IMPORTANT: This step requires the use of Trusted certificates. See techdocs.zebra.com/stagenow/4-2/trustedstaging/ for more information.



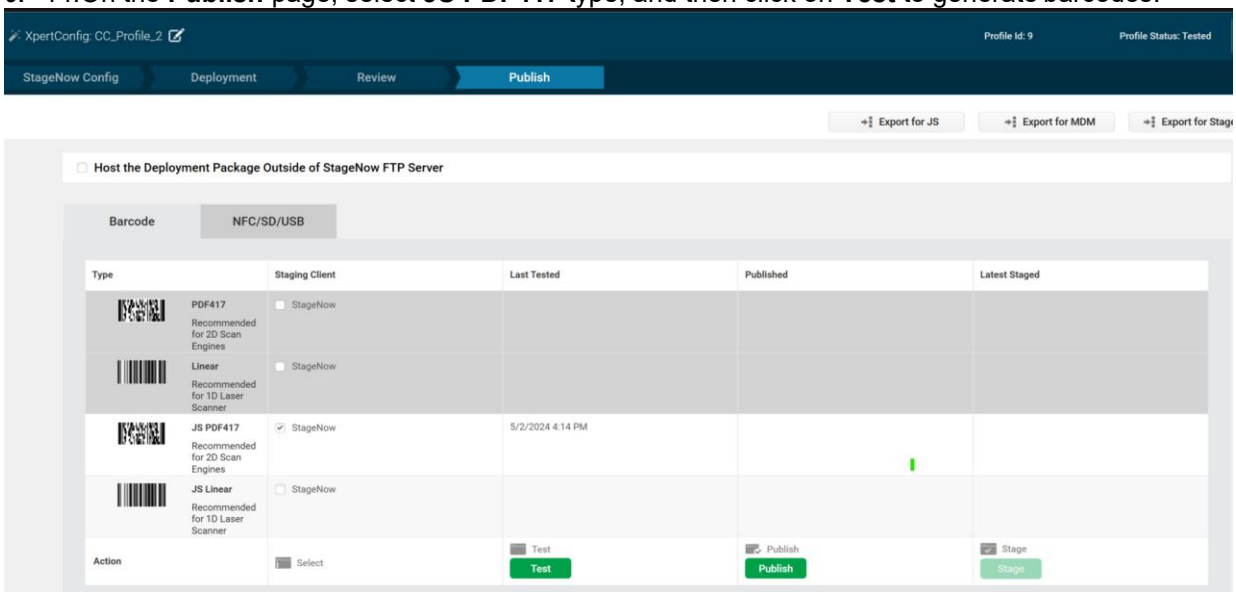
6. Download the MDM agent package to the device.



7. Install MDM agent package on the device.



8. Enroll MDM agent as Device Owner.
9. 14.On the **Publish** page, select **JS PDF417** type, and then click on **Test** to generate barcodes.



IMPORTANT: Zebra devices in CC mode will not support the following:

- Work Profile Separation
- MultiUser
- SDCard