Motorola Mobility LLC Mobile Devices on Android 15 Administrator Guidance Documentation

Version 0.2
08/06/2026

Table of Contents

List of Tables

1. Introduction

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

1.1 Evaluated Devices

Th evaluated device encompasses mobile devices that support enterprises and individual users alike and includes the following models and versions:

Product	Model Number	CPU	Kernel	Android OS Version	Security Patch Level
Motorola Edge 2024	XT2405-1 XT2405V XT2307-1 XT2307-2 XT2307-3	Qualcomm Snapdragon 7s Gen 2	5.10	Android 15	June 2026
Motorola Razr 60 Ultra	XT2551-3 XT2551-5 XT2551-6 XT2551-7	Qualcomm Snapdragon 8 Elite	6.6	Android 15	June 2026
Motorola Razr Ultra 2025	XT2551-1 XT2551-2	Qualcomm Snapdragon 8 Elite	6.6	Android 15	June 2026

Table 1 Evaluated Devices Details

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

1. Open the Settings app
2. Tap on About phone
3. Tap on Device details
4. Tap on Android version

1.2 Acronyms

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
EC – Elliptic Curve
EMM – Enterprise Mobility Management
FBE – File Based Encryption
FDE – Full Disk Encryption

FIPS – Federal Information Processing Standards
HTTPS – Hyper Text Transfer Protocol Security
IT – Information Technology
MDM – Mobile Device Management
MX – Mobility Extensions
NIST – National Institute of Standards and Technology
PKI – Public Key Infrastructure
TEE – Trusted Execution Environment
TLS – Transport Layer Security
TOE – Target of Evaluation

2. Evaluated Capabilities

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

- Data Protection
- Lock Screen
- 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

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 will not be able to read it. The device utilizes File-based encryption (FBE) by default, 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 is 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:

1. 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.
2. 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 is only available after the device has performed a successful Verified Boot.

By default, apps do not run during Direct Boot mode. If an app needs to take action during Direct Boot mode, such as an accessibility service like Talkback or an alarm clock app, the app 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 where there is 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

Motorola 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 Lock Screen

2.2.1 Biometric Authentication

Biometric authentication using fingerprints is available on all devices. While the position of the sensor varies by device, the user interactions with the system are otherwise identical.

Up to four separate fingerprints can be enrolled into the device at one time. This provides both the ability to use different hands as well as options for events that prevent a successful match, such as a cut on the finger. To enroll a fingerprint, the user must first set a password for the device.

The user can manage their fingerprints by going to Settings > Security & privacy > Device unlock > Fingerprint. The user will be prompted to enter their password. Once the correct password is entered, the user can then manage their fingerprints (fingerprints cannot be managed without entering the password). Tapping Fingerprint will let the user add new fingerprints or delete existing fingerprints from the device. Once the maximum number of fingerprints have been added, the 'Add fingerprint' option is no longer available.

When enrolling a fingerprint, the device will prompt the user to provide multiple good samples to build the template. The user will be asked to move the part of the finger touching the sensor around (to cover a wider area of the finger). If the user moves their finger too far (such as off the sensor) or not enough, the user will be prompted to move their finger again in order to ensure a proper range of locations of the finger as well as enough quality samples. Different messages will guide the user to provide the proper samples and will continue until the device confirms enough quality data has been acquired.

Biometric authentication cannot be used after a power event (such as a power-on or reboot), and so the password must be configured to act as the primary authentication method. After the password has been entered once, the user will be able to authenticate using fingerprints. The user will be required to enter the password periodically (at least once per day).

2.2.2 Authentication Rate Limiting

Both biometric template matching and passcode verification can only take place on secure hardware with rate limiting (exponentially increasing timeouts) enforced. Android's GateKeeper throttling is also used to prevent brute-force attacks. After a user enters an incorrect password, GateKeeper APIs return a value in milliseconds for which the user must wait before attempting to validate another password. Any attempts before the defined amount of time has passed are ignored because any letter pressed on the keyboard is not registered. GateKeeper also keeps count of the number of failed validation attempts since the last successful attempt. These two values together are used to prevent brute-force attacks of the TOE's password.

For biometric authentication, the user can attempt 5 failed fingerprint unlock attempts before fingerprint is locked for 30 seconds. After the 20th cumulative attempt, the device prohibits the use of biometric authentication until the password is entered.

Android offers APIs that allow apps to use biometrics for authentication and allows users to authenticate by using their fingerprint scans. These APIs are used in conjunction with the Android Keystore system.

2.3 Key Management

2.3.1 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 apps 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 Trusted Execution Environment (TEE) software cannot roll a device back to the vulnerable version and use keys created with the newer version.

2.3.2 KeyChain

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

Whereas the KeyStore is for non-sharable app-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.

2.4 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.

2.4.1 Verified Boot

Verified Boot is Android's secure boot process that verifies system software before its run. This makes it more difficult for software attacks to persist 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 the 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 National Institute of Standards and Technology (NIST) for hashing algorithms (SHA-256) and public key sizes (RSA-2048).

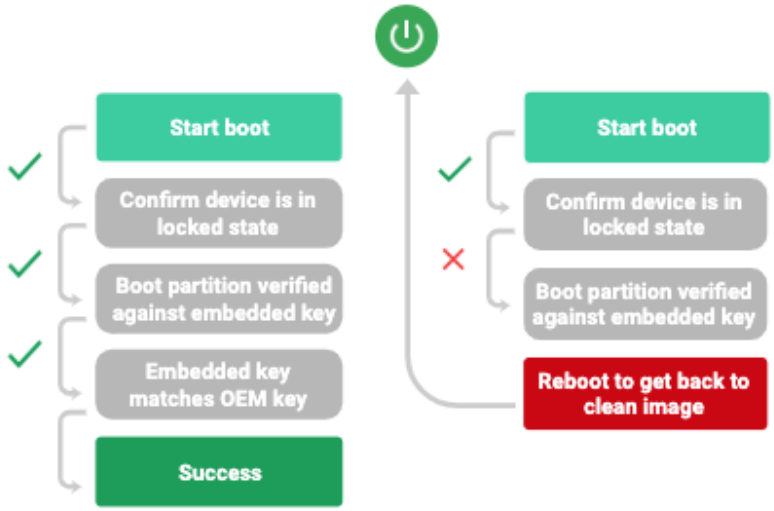


Figure 1 Verified Boot Process

The Verified Boot state is used as an input in the process to derive disk encryption key. 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.

More information about Verified Boot can be found here.

2.5 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 EEM/MDM Application Catalog.

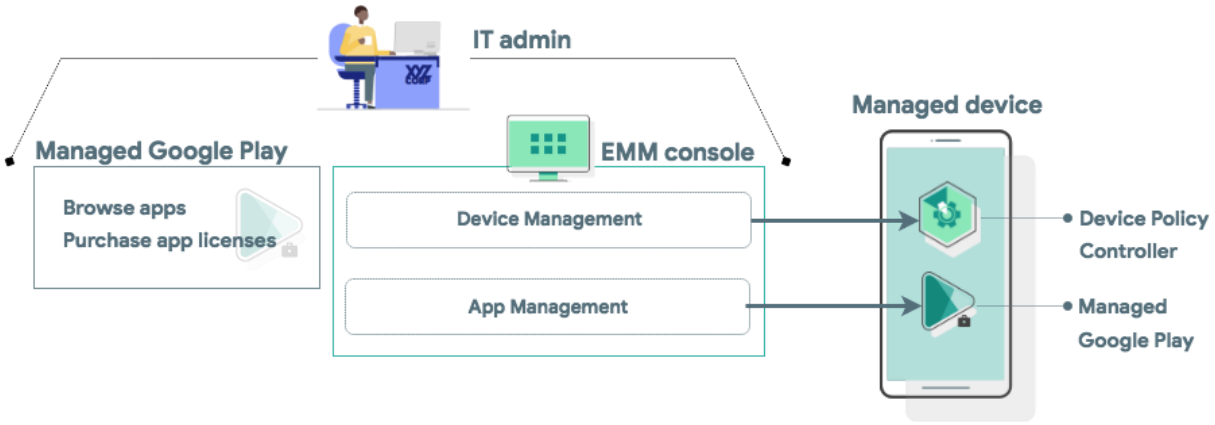


Figure 2 Components of an Android Enterprise Solution

2.5.1 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.

2.5.2 DPC (MDM Agent)

All Android devices that an organization manages through your EMM console must install a DPC app 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 (e.g. "afw#"), according to the implementation guidelines defined in the Play EEM 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.

2.6 Work Profile Separation

Fully managed devices with work profiles are for company-owned devices that are used for both work and personal purposes. The organization still manages the entire device. However, the separation of work data and apps into a work profile allows organizations to enforce two separate sets of policies. For example:

- A stronger set of policies for the work profile that applies to all work apps and data.
- A more lightweight set of policies for the personal profile that applies to the user's personal apps and data.

More information about work profile separation is provided in Section 7.

2.7 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 (using 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 app.

More information about VPN configuration options is provided in Section 6.

2.8 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, app 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 EMM's console.
- export enterprise security logs from the EMM's console.
- Capture relevant logging information from Logcat, which does not require any additional configuration to be enabled.

A detailed audit logging table is available in Section 9 along with information on how to view and export the different types of audit logs.

Note: In order to meet CC compliance, the EMM DPC must be used to enable security logging.

3. Security Configuration

The Motorola phones offer a rich built-in interface and MDM callable interface for security configuration. This section identifies the security parameters for configuring the device in Common Criteria Mode and managing its security settings.

3.1 Common Criteria Mode

To configure the device into Common Criteria Mode, the user must set the following options:

1. Require a lock screen password
    a. Please review the Password Management items in Section 3.4 (Common Criteria Related Settings)
2. Disable Smart Lock
    a. Smart Lock can be disabled using KEYGUARD_DISABLE_TRUST_AGENTS()
3. Enable encryption of Wi-Fi and Bluetooth secrets
    a. Encryption of Wi-Fi and Bluetooth Secrets can be enabled using setting niap_mode
4. Disable Debugging features (Developer options)
    a. Debugging features are disabled by default, but the system administrator can prevent the user from enabling them by using DISALLOW_DEBUGGING_FEATURES()
5. Disable installation of applications from unknown sources
    a. Installation of applications from unknown sources can be disabled using DISALLOW_INSTALL_UNKNOWN_SOURCES()
6. Turn off usage & diagnostics
    a. Open the device's Settings app
    b. Tap Google, then All services, then Usage & diagnostics
    c. Turn Usage & diagnostics off
7. Enable Audit Logging
    a. Audit logging can be enabled using setSecurityLoggingEnabled
    b. For certain items, Logcat can be used (does not require additional enablement)
8. Applications that require MDFPPv3.3 compliant Sensitive Data Protection (SDP), Hostname Checking, Revocation Checking, or Transport Layer Security (TLS) Ciphersuite restriction must implement the NIAPSEC library

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

3.2 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 b58a0134d24b27f673e8ab494d1a65dc8883d5a02b0ed68468a55cfdb2a34d23
- Hardware Cryptography
    - The TOE's Wi-Fi Chipset provides an AES-CCMP implementation
    - The TOE's application processor (e.g., Snapdragon 7s Gen 2 & Snapdragon 8 Elite) provides additional cryptographic algorithms and 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.

3.3 Permissions Model

Android runs all apps inside sandboxes to prevent malicious or buggy app 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 app regardless of the specific development environment, APIs used, or programming language. A memory corruption error in an app only allows arbitrary code execution in the context of that particular app 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 will either grant the permission automatically or prompt the user to approve the request.

A central design point of the Android security architecture is that no app, 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 app's files, performing network access, keeping the device awake, and so on.

The DPC can grant or deny specific permissions beforehand using PERMISSION_GRANT_STATE APIs. Additionally, the end user can revoke a specific app's permission by:

1. Tapping on Settings > Apps > All apps
2. Tapping on the particular app
3. Tapping on Permissions

From there the user can select any specific permission and select the level of access to be allowed.

More information about Android Permissions is available on developer.android.com.

3.4 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. The API column indicates the administrator interface used to control the setting, while the User Interface column provides steps the user can take to control the setting.

Security Feature	Setting	Description	Required Value	API	User Interface
Encryption	Device Encryption	Encrypts all internal storage	None	Encryption on by default with no way to turn off	
	Wipe Device	Removes all data from device	None	wipeData()	Settings > System > Reset options > Erase all data (factory reset)
	Wipe Enterprise Data	Remove all enterprise data from device	None	wipeData() called from secondary user	
Password Management	Min Password Length	Minimum number of characters in a password	None	setPasswordMinimumLength()	Settings > Security & privacy > Device unlock > Screen lock > Password
	Max Password Length	Maximum number of characters in a password	16	Default value, cannot be changed	
	Password Complexity	Specify the type of characters required in a password	None	setPasswordQuality()	Settings > Security & privacy > Device unlock > Screen lock > Password
	Password Expiration	Maximum length of time before a password must change	None	setPasswordExpirationTimeout()	
	Authentication Failures	Maximum number of authentication failures	10 or less	setMaximumFailedPasswordsForWipe()	
Lockscreen	Inactivity to lockout	Time before lockscreen is engaged	None	setMaximumTimeToLock()	Settings > Display > Screen Timeout
	Banner	Banner message displayed on the lockscreen	Administrator or user defined text	setDeviceOwnerLockScreenInfo()	Settings > Home & lock screen > Lock screen > Text on lock screen
	Remote Lock	Locks the device remotely	Function must be available	lockNow()	Tap the power button to turn off the screen which locks the device

	Show Password	Disallows the displaying of the password on the 'Enter password' screen	Disable	Disabled by default	Settings > Security & privacy > Privacy controls > Show passwords toggle (display characters briefly as you type)
	Notifications	Controls if notifications are displayed on the lockscreen	Enable/Disable are available options	KEYGUARD_DISABLE_SECURE_NOTIFICATIONS() KEYGUARD_DISABLE_UNREDACTED_NOTIFICATIONS()	Settings > Home & lock screen > Lock screen > Privacy Settings > Security & privacy > More security & privacy > Notifications on lock screen Settings > Notifications > Notification on lock screen
	Control Biometric Fingerprint	Control the use of Biometric Fingerprint authentication factor	Enable/Disable are available options	KEYGUARD_DISABLE_FINGERPRINT()	
Certificate Management	Import CA Certificate	Import CA Certificates into the Trust Anchor Database or the credential storage	Certificate to import	installCaCert()	Settings > Security & privacy > More security & privacy > Encryption & credentials > Install a certificate > CA certificate
	Remove Certificates	Remove certificates from the Trust Anchor Database or the credential storage	Certificate to remove	uninstallCaCert()	To clear all user-installed certificates: Settings > Security & privacy > More security & privacy > Encryption & credentials > Clear credentials To remove a specific user-installed certificate: Settings > Security & privacy > More security & privacy > Encryption & credentials > Trusted credentials > User > desired certificate > Uninstall
	Import Client Certificates	Import client certificates in to Keychain	Certificate to import	installKeyPair()	Settings > Security & privacy > More security & privacy > Encryption & credentials > Install a certificate > VPN & app user certificate or Wi-Fi certificate
	Remove Client Certificates	Remove client certificates from Keychain	Certificate to remove	removeKeyPair()	To remove a specific user-installed certificate: Settings > Security & privacy > More security & privacy > Encryption & credentials > Trusted credentials > User > desired certificate > Uninstall

Radio Control	Control Wi-Fi	Control access to Wi-Fi	Enable/Disable are available options	DISALLOW_CONFIG_WIFI()	Settings > Network & internet > Wi-Fi toggle
	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()	Settings > Network & internet > Mobile data toggle
	Control NFC	Control access to NFC	Enable/Disable are available options	DISALLOW_OUTGOING_BEAM()	Settings > Connected devices > Connection preferences > NFC > Use NFC toggle
	Control Bluetooth	Control access to Bluetooth	Enable/Disable are available options	DISALLOW_BLUETOOTH() DISALLOW_BLUETOOTH_SHARING() DISALLOW_CONFIG_BLUETOOTH()	Settings > Connected devices > Connection preferences > Bluetooth > Use Bluetooth toggle
	Control Location Service	Control access to Location Service	Enable/Disable are available options	DISALLOW_SHARE_LOCATION() DISALLOW_CONFIG_LOCATION()	Settings > Location >Use location toggle
Wi-Fi Settings	Specify Wi-Fi SSIDs	Specify SSID values for connecting to Wi-Fi. Can also create white and black lists for SSIDs	None	WifiEnterpriseConfig()	
	Set WLAN CA Certificate	Select the CA Certificate for the Wi-Fi connection	None	WifiEnterpriseConfig()	
	Specify security type	Specify the connection security (WPA3, WPA2, etc.)	None	WifiEnterpriseConfig()	
	Select authentication protocol	Specify the EAP-TLS connection values	None	WifiEnterpriseConfig()	
	Set client credentials	Specify the client credentials to access a specified WLAN	None	WifiEnterpriseConfig()	
	Control Always-on VPN	Control access to Always-on VPN	Enable/Disable are available options	setAlwaysOnVPNPackage()	
Hardware Control	Control Microphone (across device)	Control access to microphone across the device	Enable/Disable are available options	DISALLOW_UNMUTE_MICROPHONE()	Settings > Security & privacy > Privacy controls > Microphone access toggle (For apps and services. If this setting is off,

					microphone data may still be shared when you call an emergency number)
	Control Microphone (per-app basis)	Control access to microphone per application	Enable/Disable are available options		Settings > Apps > All apps > app in question > Permissions > Microphone > select the access option desired
	Control Camera (across device)	Control access to camera across the device	Enable/Disable are available options		Settings > Security & privacy > Privacy controls > Camera access toggle (For apps and services)
	Control Camera (per-app basis)	Control access to camera per application	Enable/Disable are available options		Settings > Apps > All apps > app in question > Permissions > Camera > select the access option desired
	Control USB Mass Storage	Control access to mounting the device for storage over USB	Enable/Disable are available options	DISALLOW_MOUNT_PHYSICAL_MEDIA()	Settings > System > Developer options > Default USB configuration
	Control USB Debugging	Control access to USB debugging	Enable/Disable are available options	DISALLOW_DEBUGGING_FEATURES()	Settings > System > Developer options > USB debugging toggle
	Control USB Tethered Connections	Control access to USB tethered connections	Enable/Disable are available options	DISALLOW_CONFIG_TETHERING()	Settings > System > Developer options > Default USB configuration Settings > Network & internet > Hotspot & tethering > USB tethering toggle
	Control Bluetooth Tethered Connections	Control access to Bluetooth tethered connections	Enable/Disable are available options	DISALLOW_CONFIG_TETHERING()	Settings > Network & internet > Hotspot & tethering > Bluetooth tethering toggle
	Control Hotspot Connections	Control access to Wi-Fi hotspot connections	Enable/Disable are available options	DISALLOW_CONFIG_TETHERING()	Settings > Network & internet > Hotspot & tethering > Wi-Fi hotspot toggle
	Automatic Time	Allows the device to get time from the Wi-Fi connection	Enable/Disable are available options	setAutoTimeRequired()	Settings > System > Date & time > Set time automatically toggle
Application Control	Install Application	Installs specified application	None	PackageInstaller.Session()	
	Uninstall Application	Uninstalls specified application	App to uninstall	uninstall()	Settings > Apps > See all > app in question > Uninstall
	Application Whitelist	Specifies a list of applications that may be installed	None	This is done by the EMM/MDM when an application catalog is setup, which leverages PackageInstaller.Session()	
	Application Blacklist	Specifies a list of applications	None	PackageInstaller.SessionInfo()	

		that may not be installed			
	Application Repository	Specifies the location from which applications may be installed	None	DISALLOW_INSTALL_UNKNOWN_SOURCES()	
TOE Management	Enrollment	Enroll the TOE in management	None		During device setup, scan EMM/MDM provided QR code or enter EMM/MDM DPC identifier Refer to Section 2.5.2 for more details
	Disallow Unenrollment	Prevent the user from removing the managed profile	Enable/Disable	DISALLOW_REMOVE_MANAGED_PROFILE() DISALLOW_FACTORY_RESET()	
	Unenrollment	Unenroll the 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	Settings > Apps > See all > app in question > Uninstall
	Allow Developer Mode	Controls Developer Mode access	Enable/Disable are available options	DISALLOW_DEBUGGING_FEATURES()	Settings > System > Developer options > Use developer options toggle Settings > About phone > Device identifiers > tap Build number 7 times
	Sharing Data Between Enterprise and Personal Apps	Controls data sharing between enterprise and personal apps	Enable/Disable	DISALLOW_CROSS_PROFILE_COPY_PASTE() addCrossProfileIntentFiler()	

Table 2 Common Criteria Security Settings

3.5 Password Recommendations

When setting a password, the user 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)
- Does not contain a complete word (Password!)
- Does not contain repeating or sequential numbers and/or letters

3.6 Bug Reporting Process

Google supports a bug filing system for the Android OS outlined here:

https://source.android.com/setup/contribute/report-bugs.

This allows developers or users to search for, file, and vote on bugs that need to be fixed. This helps to ensure that all bugs that affect large numbers of people get pushed up in priority to be fixed.

Security vulnerability for moto device can be reported through the following link:

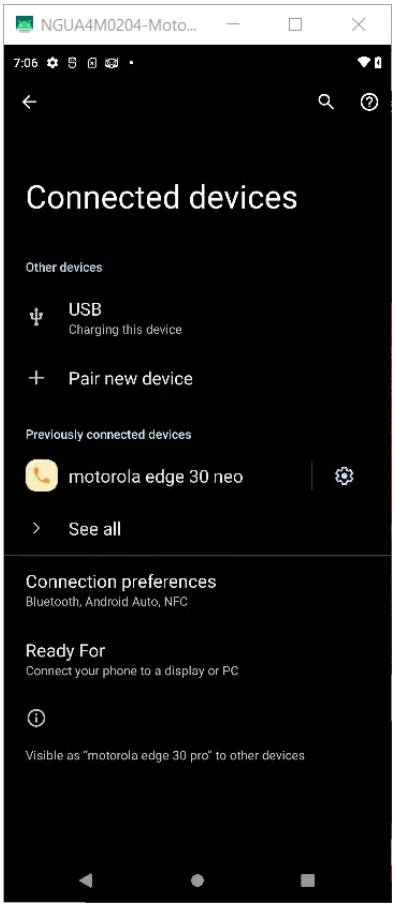
https://en-us.support.motorola.com/app/emailform.

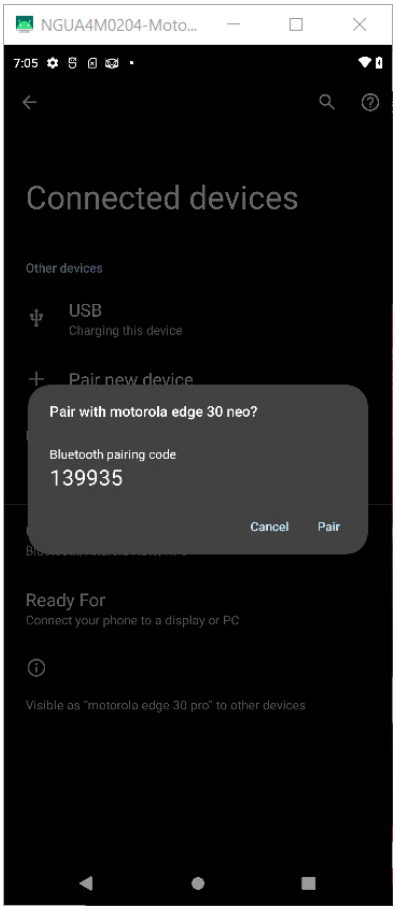
4. Bluetooth Configuration

Follow the steps below to pair and connect using Bluetooth.

Pair:

1. Open the Settings app
2. Tap Connected devices > Connection preferences > Bluetooth
3. Make sure Bluetooth is turned on by looking at the Bluetooth toggle
4. Tap Pair new device
5. Tap the name of the Bluetooth device you want to pair with
6. Follow any on-screen steps





Connect:

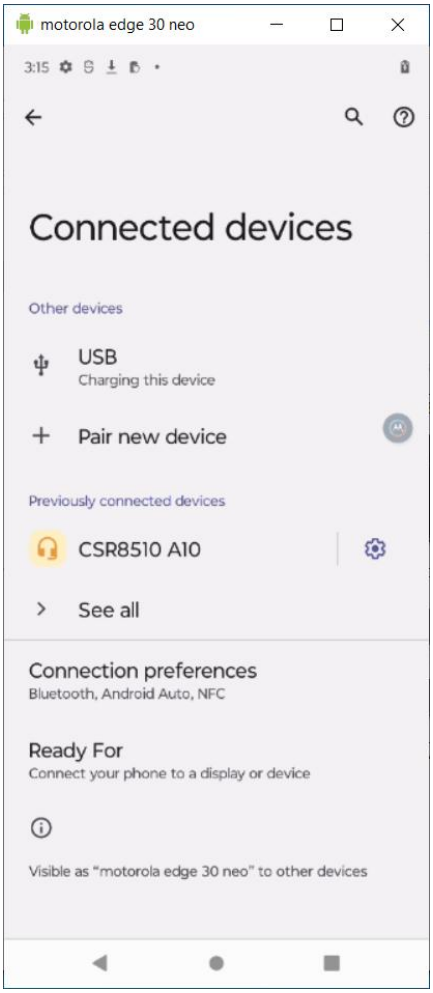
1. Open the Settings app
2. Tap Connected devices > Connection preferences > Bluetooth
3. Make sure Bluetooth is turned on by looking at the Bluetooth toggle
4. In the list of paired devices, tap a paired but unconnected device

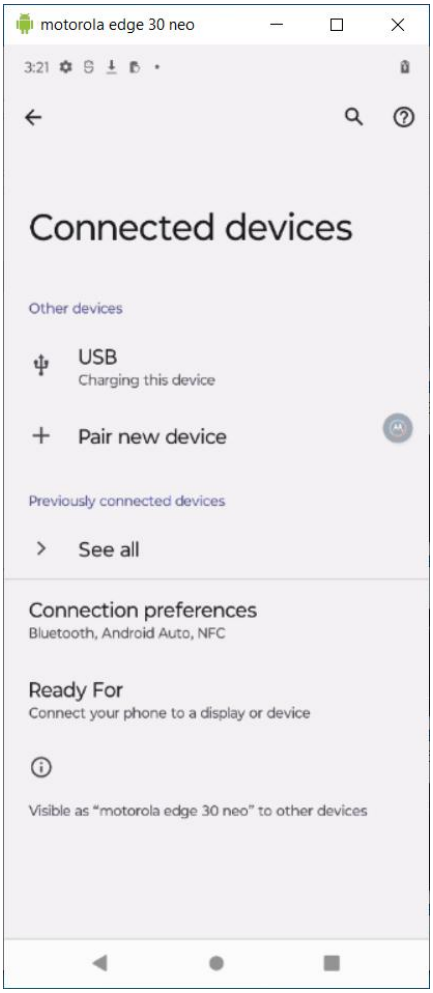
5. When your phone or tablet and the Bluetooth device are connected, the device shows as “Connected” in the list of paired devices.

Tip: If your phone is connected to something through Bluetooth, at the top of the screen, you’ll see a Bluetooth icon ( ).

Remove Previously Paired Device

1. Open the Settings app
2. Tap Connected devices > Saved devices
3. Tap the gear icon to the right of the device you want to unpair
4. Tap Forget and confirm in the popup window by tapping on Forget device





For additional support information around Bluetooth, please take a look at this support link.

5. Wi-Fi Configuration

Android supports the WPA2-Enterprise (802.11) protocol and WPA3-Enterprise protocols, which are specifically designed for enterprise networks and can be integrated into a broad range of Remote Authentication Dial-In User Service (RADIUS) authentication servers.

IT admins can silently provision enterprise Wi-Fi configurations on managed devices, including:

- SSID, via the EMM's DPC
- Password, via the EMM's DPC
- Identity, via the EMM's DPC
- Certificate for clients authorization, via the EMM's DPC
- CA certificate(s), via the EMM's DPC

IT admins can lock down Wi-Fi configurations on managed devices, to prevent users from creating new configurations or modifying corporate configurations.

IT admins can lock down corporate Wi-Fi configurations in either of the following configurations:

- Users cannot modify any Wi-Fi 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 Wi-Fi network on the device, limiting Wi-Fi connectivity to just those networks provisioned by the EMM.

When the device tries to connect to a Wi-Fi 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:

1. Enable Developer Options by tapping on Settings > About phone and tapping on Build number seven times until they see that Developer options has been enabled
2. Enable Android Debug Bridge (ADB) over USB by tapping on Settings > System > Developer options and scroll down to USB debugging and enable the toggle to on
3. Connect the device to a workstation that has ADB installed and type in "adb shell settings put global captive_portal_mode ()" and hit enter
4. You can verify the change by typing "adb shell settings get global captive_portal_mode" and the return value should be "()"
5. Turn off Developer options by tapping on Settings > System > Developer options and toggling the on option at the top to off

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

6. 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 > Network & internet > VPN. Always-on VPN can also be enabled manually from the settings menu.

7. Work Profile Separation

Work profile mode is initiated when the DPC initiates a managed provisioning flow. The work profile is based on the Android multi-user concept, where the work profile functions as a separate Android user segregated from the primary profile. The work profile shares common UI real estate with the primary profile. Apps, notifications, and widgets from the work profile show up next to their counterparts from the primary profile and are always badged so users have an indication as to what type of app it is.

With the work profile, enterprise data does not intermix with personal application data. The work profile has its own apps, its own downloads folder, its own settings, and its own KeyChain. It is encrypted using its own encryption key, and it can have its own passcode to gate access.

The work profile is provisioned upon installation, and the user can only remove it by removing the entire work profile. Administrators can also remotely instruct the device policy client to remove the work profile, for instance, when a user leaves the organization or a device is lost. Whether the user or an IT administrator removes the work profile, user data in the primary profile remains on the device.

A DPC running in profile owner mode can require users to specify a security challenge for apps running in the work profile. The system shows the security challenge when the user attempts to open any work apps. If the user successfully completes the security challenge, the system unlocks the work profile and decrypts it, if necessary.

Android also provides support for a separate work challenge to enhance security and control. The work challenge is a separate passcode that protects work apps and data. Admins managing the work profile can choose to set the password policies for the work challenge differently from the policies for other device passwords. Admins managing the work profile set the challenge policies using the usual DevicePolicyManager methods, such as setPasswordQuality() and setPasswordMinimumLength(). These admins can also configure the primary device lock, by using the DevicePolicyManager instance returned by the DevicePolicyManager.getParentProfileInstance() method.

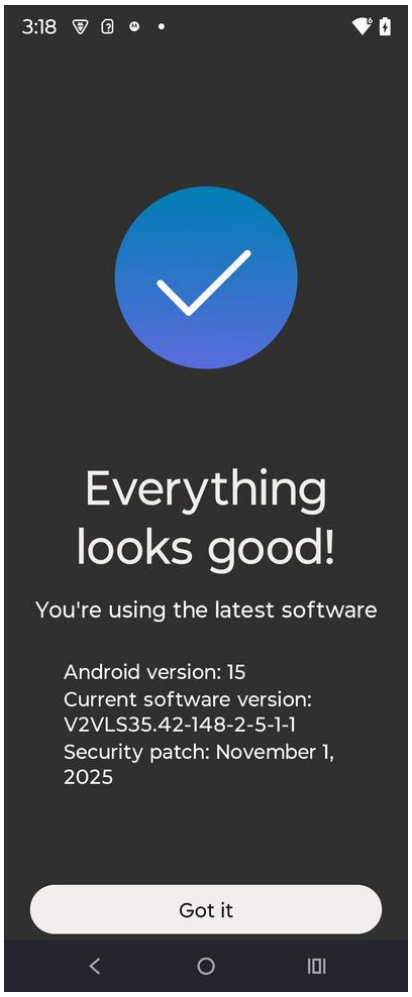
As part of setting up a separate work challenge, users may also elect to enroll fingerprints to unlock the work profile more conveniently. Fingerprints must be enrolled separately from the primary profile as they are not shared across profiles.

As with the primary profile, the work challenge is verified within secure hardware, ensuring that it is difficult to brute-force. The passcode, mixed in with a secret from the secure hardware, is used to derive the disk encryption key for the work profile, which means that an attacker cannot derive the encryption key without either knowing the passcode or breaking the secure hardware.

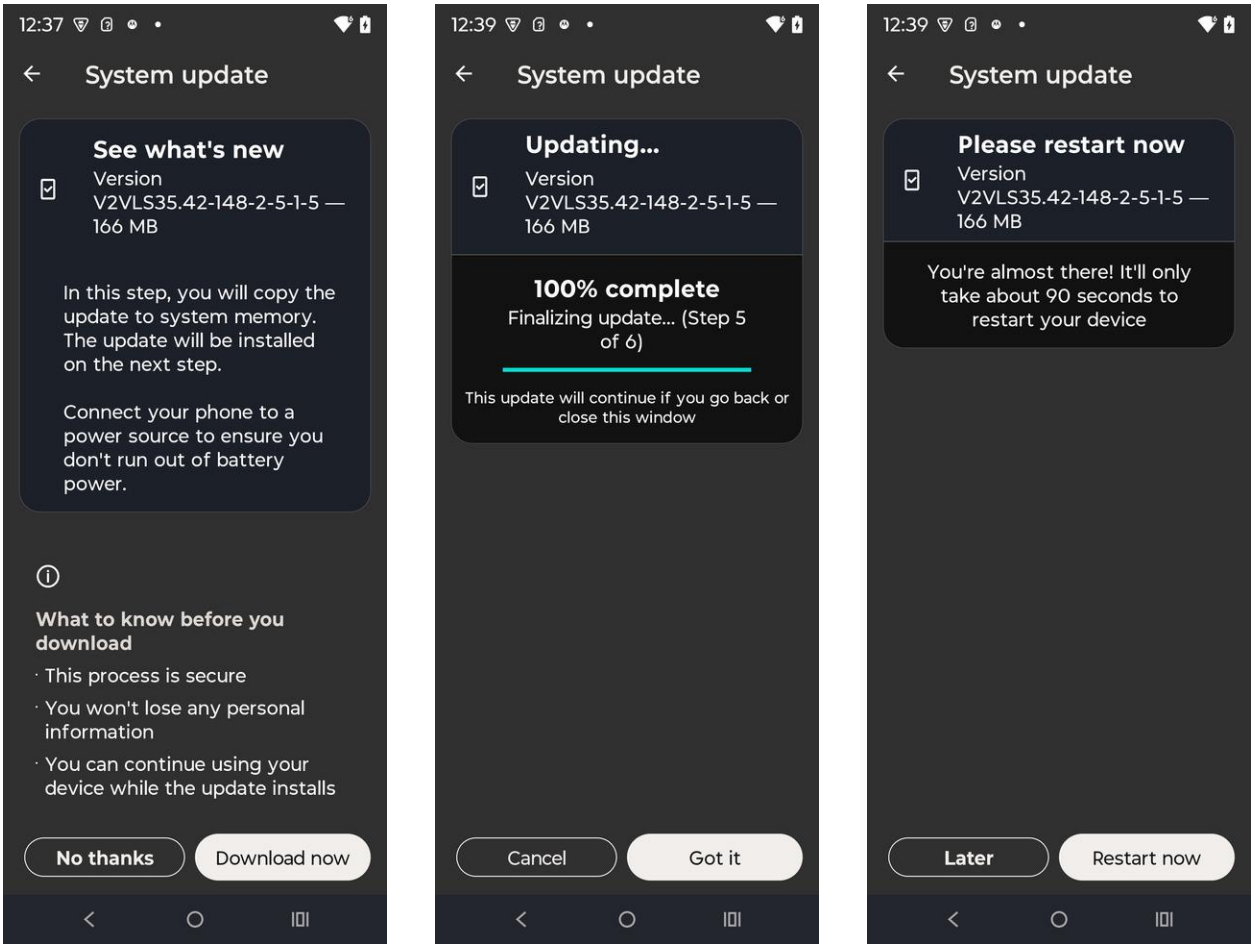
8. Secure Update Process

Motorola does not provide full Over-the-Air (OTA) packages that can be sideloaded. Instead, delta OTA files are provided for internal/testing purposes. In order to perform an SDCard update with the delta OTA file, the file need to be copied to the /sdcard/Android/data/com.motorola.ccc.ota/files location of the TOE device. This update will then be applied via Settings > System updates > Check for updates. If the internet is not available, a screen with the message “No Internet” is displayed. If the internet is available, the device will first try to install from the OTA package present in sdcard. The OTA update from sdcard will then be verified via OtaApp (built into the system settings). Should this verification fail, the software update will fail and the update will not be installed. The device will then attempt to download the OTA package via the internet. Additionally, the Motorola phones also provide roll-back protection for OTA updates to prevent a user from installing a prior/previous version of software by check via OtaApp.

If there is no OTA package in sdcard (or verification of the OTA package in sdcard failed), the device will try to download the OTA package via the internet. If the current software version is found to match the most recent version available on the internet, a screen with the message “Everything looks good!” along with the current Android version, software version, and security patch is displayed.



If the current software version is found to be later than the most recent version available on the internet, a screen displaying the software version available on the internet gives the options “Download now” and “No thanks”. If “No thanks” is selected, the OTA package is not downloaded and the current software version remains the same. If “Download now” is selected, the OTA package is downloaded and details of the system update are displayed. Once the process is complete, a screen with the message “Please restart now” is displayed with the option to restart the device at a later time, or restart the device immediately in order to apply the update.



The Motorola phones leverage A/B system updates, also known as seamless updates. This approach ensures that a workable booting system remains on the disk during an OTA update. This approach reduces the likelihood of an inactive device after an update, which means fewer device replacements and device reflashes at repair and warranty centers. Other commercial-grade operating systems such as ChromeOS also use A/B updates successfully.

The user will get a notification when an update is made available. No special configuration will be required to ensure a secure update process.

8.1 Google Play System Updates

Google Play System Updates offer a simple and fast method to deliver updates. End-user devices receive the components from the Google Play Store or through a partner-provided over-the-air (OTA) mechanism.

The components are delivered as either APK or APEX files – APEX is a new file format which loads earlier in the booting process. Important security and performance improvements that previously needed to be part of full OS updates can be downloaded and installed similarly to an app update. Updates delivered in this way are secured by being cryptographically signed.

Google Play System Updates can also deliver faster security fixes for critical security bugs by modularizing media components, which accounted for nearly 40% of recently patched vulnerabilities, and allowing updates to Conscrypt, the Java Security Provider.

9. Audits

9.1 Security Logs

An MDM agent acting as Device Owner can control the logging with setSecurityLoggingEnabled. When security logs are enabled, device owner apps receive periodic callbacks from onSecurityLogsAvailable at which time a new batch of logs can be collected via 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. For example: <Keyword> (<Date><Timestamp>): <message>

9.2 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 here. 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>

The table below provides general audit events:

Requirement	Auditable Events	Additional Audit Record Contents	Log Events & Examples
FAU_GEN.1	Start-up and shutdown of the audit functions		**SecurityLog** Start-up: LOGGING_STARTED (Thu Sep 24 10:53:19 EDT 2020): 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.

	Start-up and shutdown of the OS		**SecurityLog** Start-up: OS_STARTUP (Thu Sep 24 10:53:18 EDT 2020): orange enforcing 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.
FAU_SAR.1	No events specified		
FAU_STG.1	No events specified		
FAU_STG.4	No events specified		
FCS_CKM.1	**[None]**	No additional information.	
FCS_CKM.2/ LOCKED	No events specified		
FCS_CKM.2/ UNLOCKED	No events specified		
FCS_CKM_EXT.1	**[None]**	No additional information.	
FCS_CKM_EXT.2	No events specified		
FCS_CKM_EXT.3	No events specified		
FCS_CKM_EXT.4	No events specified		
FCS_CKM_EXT.5	**[None]**	No additional information.	
FCS_CKM_EXT.6	No events specified		
FCS_COP.1/ CONDITION	No events specified		
FCS_COP.1/ ENCRYPT	No events specified		
FCS_COP.1/ HASH	No events specified		
FCS_COP.1/ KEYHMAC	No events specified		
FCS_COP.1/SIGN	No events specified		
FCS_IV_EXT.1	No events specified		
FCS_SRV_EXT.1	No events specified		
FCS_SRV_EXT.2	No events specified		
FCS_STG_EXT.1	Import or destruction of key **[No other events]**	Identity of key. Role and identity of requestor.	**SecurityLog** Import: KEY_IMPORTED (Thu Sep 24 12:21:47 EDT 2020): 1 USRPKEY_852acf518726852acf518726278597463f75999f3e2811 0a61a9 1000 Destroy:

			KEY_DESTROYED (Thu Sep 24 12:22:38 EDT 2020): 1 USRPKEY_852acf518726852acf518726278597463f75999f3e2811 0a61a9 1000
FCS_STG_EXT.2	No events specified		
FCS_STG_EXT.3	Failure to verify integrity of stored key	Identity of key being verified.	**SecurityLog** KEY_INTEGRITY_VIOLATION (Thu Oct 29 16:20:44 EDT 2020): USRPKEY “corrupt” 1010
FDP_ACF_EXT.1	No events specified		
FDP_ACF_EXT.2	No events specified		
FDP_BLT_EXT.1	No events specified		
FDP_DAR_EXT.1	**[None]**	No additional information.	
FDP_DAR_EXT.2	**[None]**	No additional information.	
FDP_STG_EXT.1	Addition or removal of certificate from Trust Anchor Database	Subject name of certificate.	**SecurityLog** Addition: CERT_AUTHORITY_INSTALLED (Thu Sep 24 12:22:17 EDT 2020): 1 cn=rootca-rsa,1.2.840.113549.1.9.1=#161a726f6f7463612d72736140676f7373616d65727365632e636f6d,o=gss,l=catonsville,st=md,c=us 0 Removal: CERT_AUTHORITY_REMOVED (Thu Sep 24 12:22:30 EDT 2020): 1 cn=rootca-rsa,1.2.840.113549.1.9.1=#161a726f6f7463612d72736140676f7373616d65727365632e636f6d,o=gss,l=catonsville,st=md,c=us 0
FDP_UPC_EXT.1/ APPS	Application initiation of trusted channel	Name of application. Trusted channel protocol. Non-TOE endpoint of connection.	This test references a test case within the PKGTLS11 protection profile, which has no auditable events.
FIA_PMG_EXT.1	No events specified		
FIA_TRT_EXT.1	No events specified		
FIA_UAU.5	No events specified		
FIA_UAU.6/ CREDENTIAL	No events specified		
FIA_UAU.7	No events specified		
FIA_UAU_EXT.1	No events specified		

FIA_X509_EXT.1	Failure to validate X.509v3 certificate	Reason for failure of validation.	**Logcat** Revoked Certificate: 06-04 12:32:16.605 15231 15330 W System.err: java.security.cert.CertPathValidatorException: Certificate has been revoked, reason: UNSPECIFIED, revocation date: Wed Mar 29 13:12:17 EDT 2023, authority: EMAILADDRESS=server-ocsp-subsubca-rsa@gossamersec.com, CN=server-ocsp-subsubca-rsa, O=GSS, L=Catonsville, ST=MD, C=US, extension OIDs: [] 06-04 12:32:16.606 15231 15330 W System.err: Caused by: java.security.cert.CertificateRevokedException: Certificate has been revoked, reason: UNSPECIFIED, revocation date: Wed Mar 29 13:12:17 EDT 2023, authority: EMAILADDRESS=server-ocsp-subsubca-rsa@gossamersec.com, CN=server-ocsp-subsubca-rsa, O=GSS, L=Catonsville, ST=MD, C=US, extension OIDs: [] Expired Certificate: 06-04 13:58:01.020 5661 5783 I ValidatableSSLSocket: Failed to establish a TLS connection to 192.168.144.36 ... Unacceptable certificate: EMAILADDRESS=subca-expired-rsa@gossamersec.com, CN=subca-expired-rsa, O=GSS, L=Catonsville, ST=MD, C=US 06-04 13:58:01.023 5661 5783 W System.err: Caused by: java.security.cert.CertificateExpiredException: Certificate expired at Wed Mar 29 13:16:00 EDT 2023 (compared to Tue Jun 04 13:58:01 EDT 2024) No Trust Anchor: 06-04 13:51:59.939 2164 3001 I ValidatableSSLSocket: Failed to establish a TLS connection to 192.168.144.36 ... java.security.cert.CertPathValidatorException: Trust anchor for certification path not found. Unspecified Error: 06-04 14:03:02.570 7046 7083 I ValidatableSSLSocket: Failed to establish a TLS connection to 192.168.144.36 ... Read error: ssl=0xb4000075f4939098: Failure in SSL library, usually a protocol error
FIA_X509_EXT.3	No events specified		
FMT_SMF_EXT.3	No events specified		
FPT_AEX_EXT.1	No events specified		
FPT_AEX_EXT.2	No events specified		
FPT_AEX_EXT.3	No events specified		
FPT_AEX_EXT.4	No events specified		
FPT_AEX_EXT.5	No events specified		
FPT_BBD_EXT.1	No events specified		
FPT_BLT_EXT.1	No events specified		
FPT_JTA_EXT.1	No events specified		

FPT_KST_EXT.1	No events specified		
FPT_KST_EXT.2	No events specified		
FPT_KST_EXT.3	No events specified		
FPT_NOT_EXT.1	**[None]**	**[No additional information]**	
FPT_STM.1	All administrative actions		See Table 5 below which provides audit events for the administrative management functions that are mandated by the PP to be admin only.
FPT_TST_EXT.1	Initiation of self-test	No additional information.	**SecurityLog** CRYPTO_SELF_TEST_COMPLETED (Thu Sep 24 10:53:19 EDT 2020): 1
	Failure of self-test	**[none]**	**SecurityLog** CRYPTO_SELF_TEST_COMPLETED (Thu Sep 24 10:53:19 EDT 2020): 0
FPT_TST_EXT.2/ PREKERNEL	Start-up of TOE	No additional information.	**SecurityLog** OS_STARTUP (Thu Sep 24 10:53:18 EDT 2020): orange enforcing
	[None]	**[No additional information]**	
FPT_TST_EXT.2/ POSTKERNEL	**[None]**	**[No additional information]**	
FPT_TST_EXT.3	No events specified		
FPT_TUD_EXT.1	No events specified		
FPT_TUD_EXT.6	No events specified		
FTA_SSL_EXT.1	No events specified		
FTA_TAB.1	No events specified		
FTP_ITC_EXT.1	Initiation and termination of trusted channel	Trusted channel protocol. Non-TOE endpoint of connection.	This test references several other test cases: PKGTLS11:FCS_TLSC_EXT.2.1, which has no auditable events WLANC10:FCS_CKM.1/WPA, which has no auditable events WLANC10:FCS_TLSC_EXT.1/WLAN, see Table 4 below

Table 3 MDFPP33 Auditable Events

The table below provides audit events for the Wi-Fi connectivity:

Requirement	Auditable Events	Additional Audit Record Contents	Log Events & Examples
FAU_GEN.1/ WLAN	No events specified		
FCS_CKM.1/WPA	No events specified		

FCS_CKM.2/ WLAN	No events specified		
FCS_TLSC_EXT.1/ WLAN	Failure to establish an EAP-TLS session	Reason for failure	**Logcat** Certificate error: 08-27 10:40:27.290 3713 3713 I wpa_supplicant: wlan0: CTRL-EVENT-EAP-TLS-CERT-ERROR reason=1 depth=3 subject='/C=US/ST=MD/L=Catonsville/O=GSS/emailAddress=rootca-rsa@gossamersec.com/CN=rootca-rsa' err='self signed certificate in certificate chain' Authentication failure: 08-27 10:40:28.298 3713 3713 I wpa_supplicant: wlan0: CTRL-EVENT-EAP-FAILURE EAP authentication failed Unspecified TLS/SSL error: 08-27 10:40:32.418 3713 3713 I wpa_supplicant: TLS - SSL error: error:0900006e:PEM routines:OPENSSL_internal:NO_START_LINE Post-failure termination (occurs after the above 3): 08-27 10:40:28.328 3713 3713 I wpa_supplicant: wlan0: CTRL-EVENT-DISCONNECTED bssid=70:1a:b8:db:3c:c9 reason=23 locally_generated=0 disconnect_rssi=50
	Establishment/ termination of an EAP-TLS session	Non-TOE endpoint of connection	**Logcat** Establishment: 08-27 10:40:58.777 3713 3713 I wpa_supplicant: wlan0: CTRL-EVENT-CONNECTED - Connection to 70:1a:b8:db:3c:c9 completed [id=0 id_str=%7B%22configKey%22%3A%22%5C%22nanoPC2-EAP%5C%22WPA_EAP%22%2C%22creatorUid%22%3A%221000%22%7D] Termination: 08-27 10:49:51.249 3713 3713 I wpa_supplicant: wlan0: CTRL-EVENT-DISCONNECTED bssid=70:1a:b8:db:3c:c9 reason=3 locally_generated=1 disconnect_rssi=53
FCS_TLSC_EXT.2/ WLAN	No events specified		
FCS_WPA_EXT.1	No events specified		
FIA_PAE_EXT.1	No events specified		
FIA_X509_EXT.1/ WLAN	Failure to validate X.509v3 certificate	Reason for failure of validation	**Logcat** Certificate signature failure: 06-23 11:26:52.053 16553 16553 I wpa_supplicant: OpenSSL: openssl_handshake - SSL_connect error:04000069:RSA routines:OPENSSL_internal:BAD_SIGNATURE Invalid CA in Chain: 06-23 11:24:33.884 16553 16553 I wpa_supplicant: SSL: SSL3 alert: write (local SSL3 detected an error):fatal:unknown CA Expired Certificate: 06-23 11:24:00.159 16553 16553 I wpa_supplicant: SSL: SSL3 alert: write (local SSL3 detected an error):fatal:certificate expired

FIA_X509_EXT.2/ WLAN	No events specified		
FIA_X509_EXT.6	Attempts to load certificates	None	See the audits for FCS_STG_EXT.1 – Import and Destruction of keys.
	Attempts to revoke certificates	None	See the audits for FCS_STG_EXT.1 – Import and Destruction of keys.
FMT_SMF.1/ WLAN	No events specified		
FPT_TST_EXT.3/ WLAN	Execution of this set of TSF self-tests	No additional information	See the audits for FPT_TST_EXT.1, these self-tests are included in the same audit message.
	[None]	**[none]**	
FTA_WSE_EXT.1	All attempts to connect to access points	For each access point record the **[Certificate Check Message and the last [2] octets]** of the MAC Address Success and failures (including reason for failure)	**Logcat** Attempt to connect: 08-23 13:26:50.074 3713 3713 I wpa_supplicant: wlan0: Trying to associate with SSID 'nanoPC2-EAP' Success & Failure: 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	See the audits for FTA_WSE_EXT.1. Connecting to a trusted channel under this profile is equivalent to connecting to an access point.

Table 4 WLAN Auditable Events

The table below provides audit events for the management functions:

Requirement	Auditable Events	Additional Audit Record Contents	Log Events & Examples
FMT_SMF.1 Function 1	Configure password policy		
	a. minimum password length	Greater than or equal to 8	**SecurityLog** PASSWORD_COMPLEXITY_SET (Thu Jun 18 19:50:21 EDT 2020): com.afwsamples.testdpc 0 0 16 393216 0 0 0 0 0 0
	b. minimum password complexity	No value required	**SecurityLog** PASSWORD_COMPLEXITY_SET (Sun Jul 05 19:01:57 EDT 2020): com.afwsamples.testdpc 0 0 0 393216 1 0 1 0 0 1
	c. maximum password lifetime		**SecurityLog** PASSWORD_COMPLEXITY_SET (Sun Jul 05 19:03:55 EDT 2020): com.afwsamples.testdpc 0 0 600000
FMT_SMF.1 Function 2	Configure session locking policy		
	a. screen-lock enabled/disabled	Enabled	By setting a password complexity, the admin enables screen-lock: **SecurityLog**

		No value required	PASSWORD_COMPLEXITY_SET (Sun Jul 05 19:01:57 EDT 2020): com.afwsamples.testdpc 0 0 0 393216 1 0 1 0 0 1 After requiring a password, the admin can force the user to set a password: **Logcat** 09-23 13:17:18.528 1499 6482 I ActivityTaskManager: START u0 {act=android.app.action.SET_NEW_PASSWORD cmp=com.android.settings/.password.SetNewPasswordActivity} from uid 10245
	b. screen lock timeout	10 minutes or less	Admin can set max timeout, user can select more restrictive: **SecurityLog** MAX_SCREEN_LOCK_TIMEOUT_SET (Mon Jul 13 21:39:23 EDT 2020): com.afwsamples.testdpc 0 0 120000 After setting a max time, admin can prevent further configuration: **SecurityLog** USER_RESTRICTION_ADDED (Mon Jul 13 21:42:18 EDT 2020): com.afwsamples.testdpc 0 no_config_screen_timeout
	c. number of authentication failures	10 or less	**SecurityLog** MAX_PASSWORD_ATTEMPTS_SET (Wed Sep 23 13:22:53 EDT 2020): com.afwsamples.testdpc 0 0 10
FMT_SMF.1 Function 8	Configure application installation policy		
	a. restricting the sources of applications	Enable Disable	Set restriction: **SecurityLog** USER_RESTRICTION_ADDED (Thu Aug 27 13:34:17 EDT 2020): com.afwsamples.testdpc 0 no_install_unknown_sources Remove restriction: **SecurityLog** USER_RESTRICTION_REMOVED (Thu Aug 27 13:34:42 EDT 2020): com.afwsamples.testdpc 0 no_install_unknown_sources
	c. denying installation of applications	Enable Disable	Set restriction: **SecurityLog** USER_RESTRICTION_ADDED (Wed Jan 15 14:33:38 EDT 2020): com.afwsamples.testdpc 0 no_install_apps Remove restriction: **SecurityLog**

			USER_RESTRICTION_REMOVED (Wed Jan 15 14:33:53 EDT 2020): com.afwsamples.testdpc 0 no_install_apps

Table 5 Administrative Management Function Auditable Events

The table below provides audit events for the Bluetooth connectivity:

Requirement	Auditable Events	Additional Audit Record Contents	Log Events & Examples
FCS_CKM_EXT.8	None		
FIA_BLT_EXT.1	Failed user authorization of Bluetooth device.	User authorization decision (e.g., user rejected connection, incorrect pin entry). **[last [2] octets of the] BD_ADDR** and **[*no other information*]**.	**Logcat** 03-28 16:59:08.487 12828 12828 D BTPairingController: Pairing dialog canceled 03-28 16:59:08.487 12828 12828 I BluetoothDevice: cancelBondProcess() for device XX:XX:XX:XX:71:0A called by pid: 12828 tid: 12828
	Failed user authorization for local Bluetooth Service.	Bluetooth profile. Identity of local service with **[profile name]**. (TD0645 applied)	**Logcat** 08-30 13:05:03.189 3191 13451 V BluetoothDatabase: getProfileConnectionPolicy: XX:XX:XX:XX:71:0A, profile=9, connectionPolicy = -1 08-30 13:05:11.150 3191 3191 W AdapterProperties: PROFILE_CONNECTION_STATE_CHANGE: unexpected transition for profile=9, device=XX:XX:XX:XX:71:0A, 2 -> 0 08-30 13:05:11.165 3002 3086 D CachedBluetoothDevice: onProfileStateChanged: profile MAP, device XX:XX:XX:XX:71:0A, newProfileState 0 (0 means connection state is disconnected)
FIA_BLT_EXT.2	Initiation of Bluetooth connection	**[last [2] octets of the] BD_ADDR** and **[*no other information*].**	**Logcat** 08-22 13:30:54.102 17477 17477 D BTPairingController: Pairing dialog accepted 08-22 13:30:57.848 3198 3505 I BluetoothBondStateMachine: Bond State Change Intent:XX:XX:XX:XX:74:59 BOND_BONDING => BOND_BONDED
	Failure of Bluetooth connection	Reason for failure. (TD0645 applied)	**Logcat** 08-22 13:30:46.505 17477 17477 D BTPairingController: Pairing dialog canceled 08-22 13:30:46.535 3198 3505 I BluetoothBondStateMachine: Bond State Change Intent:7C:7B:1C:A2:74:59 BOND_BONDING => BOND_NONE
FIA_BLT_EXT.3 (optional)	Duplicate connection attempt	**[last [2] octets of the]** BD_ADDR of connection attempt. (TD0645 applied)	Labeled optional as it may not be feasible to log if performed at the HCI layer.
FIA_BLT_EXT.4	None		
FIA_BLT_EXT.6	None		
FIA_BLT_EXT.7	None		
FTP_BLT_EXT.1	None		

FTP_BLT_EXT.2	None		
FTP_BLT_EXT.3 /BR	None		
FTP_BLT_EXT.3 /LE	None		

Table 6 Bluetooth Auditable Events

10. 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 biometrically.

Saving sensitive data files requires a key to be generated in advance. Please see the Key Generation Section for more information.

Supported algorithms via SecureConfig.getStrongConfig():

- File Encryption Key: AES256 – AES/GCM/NoPadding
- Key Encryption Key: RSA4096 – 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. Please see EphemeralSecretKey for more information.

10.1 SecureContextCompat

Included in the NIAPSEC library.

Used to encrypt and decrypt files that require sensitive data protection.

Supported Algotihms:

- AES256 – AES/GCM/NoPadding

- RSA4096 – RSA/ECB/OAEPWithSHA-256AndMGF1Padding

Public Contstructor	
SecureContextCompat	**new SecureContextCompat**(Context, BiometricSupport) *See BiometricSupport* Constructor to create an instance of the SecureContextCompat with Biometric support.
Public Methods	
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 – listener for the resulting FileInputStream

Table 7 SecureContextCompat

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
});
```

Built using the JCE libraries for more information please see the following resources:
AndroidKeyStore - https://developer.android.com/training/articles/keystore

BiometricPrompt -
https://developer.android.com/reference/android/hardware/biometrics/BiometricPrompt

11. API Specification

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

1. Cryptographic APIs
    a. APIs for the algorithms and random number generation
2. Key Management
    a. APIs for importing, using, and destroying keys
3. Certificate Validation, TLS, Hyper Text Transfer Protocol Security (HTTPS)
    a. API used by applications for configuring the reference identifier
    b. APIs for validation checks (should match the test program provided)
    c. TLS, HTTPS, Bluetooth BR/EDR (any other protocol available to applications)

11.1 Cryptographic APIs

Code samples to do encryption and decryption, including random number generation.

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);
```

11.1.1 SECURECIPHER

Included in the NIAPSEC library.

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
- RSA4096 – RSA/ECB/OAEPWithSHA-256AndMGF1Padding

Public Static Accessors	
SecureCipher	**SecureCipher.getDefault**(BiometricSupport) *See BiometricSupport* API to get an instance of the SecureCipher with Biometric support.
Public Methods	
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 RSA4096 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.

	- ephemeralSecretKey – 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 RSA4096 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. - ephemeralSecretKey – the ephemeral key to use - encryptedData – the data to be encrypted - initializationVector – the IV used for encryption Returns a byte array of the clear text.

Table 8 SecureCipher

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

AndroidKeyStore – https://developer.android.com/training/articles/keystore

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

SecretKey – https://developer.android.com/reference/javax/crypto/SecretKey

SecureRandom – https://developer.android.com/reference/java/security/SecureRandom

BiometricPrompt – https://developer.android.com/reference/android/hardware/biometrics/BiometricPrompt

11.1.2 FCS_CKM.2/UNLOCKED – 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 – https://developer.android.com/reference/javax/crypto/Cipher

11.1.3 FCS_CKM.2/UNLOCKED – Key Establishment (ECDSA) & FCS_COP.1/SIGN – 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. Then Alice and Bob can sign/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"

“SHA512withECDSA”, “secp521r1”

Reference:

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

11.1.4 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”
“SHA512withECDSA”, “secp521r1”

Reference:

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

11.1.5 FCS_COP.1/ENCRYPT – 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 – https://developer.android.com/reference/javax/crypto/Cipher

11.1.6 FCS_COP.1/HASH – Hashing (SHA)

The MessageDigest class can be used 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 – https://developer.android.com/reference/java/security/MessageDigest

11.1.7 FCS_COP.1/SIGN – 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 – https://developer.android.com/reference/java/security/Signature

11.1.8 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 – https://developer.android.com/reference/java/security/Signature

11.1.9 FCS_COP.1/KEYHMAC – 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 – https://developer.android.com/reference/javax/crypto/Mac

11.2 KEY MANAGEMENT

Code samples to do key management.

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");
```

11.2.1 SECUREKEYGENERATOR

Included in the NIAPSEC library.

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
- RSA4096 – RSA/ECB/OAEPWithSHA-256AndMGF1Padding

Public Static Accessors	
SecureKeyGenerator	**SecureCipher.getDefault**() API to get an instance of the SecureCipher with NIAP settings.
Public Methods	
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 RSA4096 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.*

Table 9 SecureKeyGenerator

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

AndroidKeyStore – https://developer.android.com/training/articles/keystore

KeyPairGenerator – https://developer.android.com/reference/java/security/KeyPairGenerator

SecretKey – https://developer.android.com/reference/javax/crypto/SecretKey

SecureRandom – https://developer.android.com/reference/java/security/SecureRandom

KeyGenParameterSpec – https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec

11.3 FCS_TLSC_EXT.1 – Certificate Validation, TLS, HTTPS, and Bluetooth

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);
```

Public Constructors	
SecureURL	**new SecureURL**(String referenceIdentifier, String clientCert) API to create an instance of the SecureURL with NIAP settings. clientCert is optional.
Public Methods	
HttpsURLConnection	**openConnection** Opens an HttpsURLConnection using TLS by default, 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 certificate check.

Table 10 SecureURL

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

HttpsURLConnection – https://developer.android.com/reference/javax/net/ssl/HttpsURLConnection

PKIXRevocationChecker – https://developer.android.com/reference/java/security/cert/PKIXRRevocationChecker

SSLSocket – https://developer.android.com/reference/javax/net/ssl/SSLSocket

11.3.1 CIPHERSUITES

When applications utilize the NIAPSEC library, no configuration is needed to restrict or allow ciphersuites to be compliant. A list of the ciphersuites supported by Android 15 NIAPSEC can be found below.

For TLS 1.2 with mutual authentication:

Approved Ciphersuites	TLS Version
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	TLS v1.2

Table 11 Android 15 Supported TLS Ciphersuites

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 ciphersuites for this:

TLS_RSA_WITH_AES_128_CBC_SHA as defined in RFC 5246,
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

Additional ciphersuites are supported by the TOE for Wi-Fi networks but outside the specific scope of testing for the compliant configuration. It is expected that the Wi-Fi AP will be configured to utilize the proper ciphersuites to ensure compliance with the expected algorithms.

11.3.2 GUIDANCE FOR BLUETOOTH LOW ENERGY APIS

Provides classes that manage Bluetooth functionality, such as scanning 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 Bluetooth guide. For more information about Bluetooth Low Energy, see the Bluetooth Low Energy (BLE) guide.

The Bluetooth APIs let applications:

- 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, fitness devices, and so on
- 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.

Interfaces:

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 12 Bluetooth Interfaces

Classes:

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

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(), or BluetoothDevice#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(), or BluetoothDevice#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(), or BluetoothDevice#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 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 a BluetoothAdapter and to conduct overall Bluetooth Management
BluetoothServerSocket	A listening Bluetooth socket
BluetoothSocket	A connected or connecting Bluetooth socket

Table 13 Bluetooth Classes

https://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);
            }
        }
    }
```

More information here:

https://developer.android.com/guide/topics/connectivity/bluetooth.html#SettingUp

Sample service to interact with 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);
                    }
                }
            };
}
```