Administrator Guidance Documentation

2022/12/07
Version: 1.0.2

Version History

Version	Date	Changes
1.0.0	2022/10/21	First release edition
1.0.1	2022/11/21	Updated for CC comments
1.0.2	2022/12/07	Updated for CC comments

Contents

1. Document Introduction 5

  1.1 Evaluated Devices 5

  1.2 Acronyms 5

2. Evaluated Capabilities 6

  2.1 Data Protection 6

  2.2 Lock screen 7

  2.3 Key Management 7

  2.3.1 KeyStore 7

  2.4 Device Integrity 8

  2.5 Device Management 8

  2.6 Work Profile Separation 9

  2.7 VPN Connectivity 10

  2.8 Audit Logging 10

3. Security Configuration 11

  3.1 Common Criteria Mode 11

  3.2 Cryptographic Module Identification 11

  3.3 Permissions Model 11

  3.4 Common Criteria Related Settings 12

  3.5 Password Recommendations 17

  3.6 Bug Reporting Process 17

4. Bluetooth Configuration 18

5. Wi-Fi Configuration 20

6. VPN Configuration 20

7. Work Profile Separation 21

8. Secure Update Process 22

9. Audit Logging 22

10. FDP_DAR_EXT.2 & FCS_CKM.2(2) – Sensitive Data Protection Overview 33

  10.1 SecureContextCompat 33

11. API Specification 35
11.1 Cryptographic APIs 35
11.1.1 SecureCipher 36
11.1.2 FCS_CKM.2(1) – Key Establishment (RSA) 38
11.1.3 FCS_CKM.2(1) – Key Establishment (ECDSA) & FCS_COP.1(3) – Signature Algorithms (ECDSA) 38
11.1.4 FCS_CKM.1 – Key Generation (ECDSA) 39
11.1.5 FCS_COP.1(1) – Encryption/Decryption (AES) 39
11.1.6 FCS_COP.1(2) – Hashing (SHA) 40
11.1.7 FCS_COP.1(3) – RSA (Signature Algorithms) 40
11.1.8 FCS_CKM.1 –Key Generation (RSA) 40
11.1.9 FCS_COP.1(4) - HMAC 41
11.2 Key Management 41
11.2.1 SecureKeyGenerator 41
11.3 Certificate Validation, TLS, HTTPS, and Bluetooth 42
11.3.1 Cipher Suites 44
11.3.2 Bluetooth APIs 44

1. Document Introduction

This guide includes procedures for configuring Common Criteria on TOUGHBOOK devices on Android 11.

1.1 Evaluated Devices

The evaluated devices include the following models and versions:

Product	Model #	CPU	Kernel	Android OS version	Security Patch Level
TOUGHBOOK	N1	SDM660	4.19.157	Android 11.0	November 2022
TOUGHBOOK	S1	SDM660	4.19.157	Android 11.0	November 2022
TOUGHBOOK	A3	SDM660	4.19.157	Android 11.0	November 2022

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

1. Tap on Settings
2. Tap on About tablet
3. Scroll down to Android version and tap on it

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
- EMM – Enterprise Mobility Management
- MDM – Mobile Device Management
- PKI – Public Key Infrastructure
- 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.

The data protection feature is configured to be used by default, and users are not required to perform any actions beyond configuration and providing authentication credentials. In addition, users are not required to identify encryption type on a per-file basis.

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 won't be able to read it.
The device utilizes File-based encryption (FBE) which allows different files to be encrypted with different keys that can be unlocked independently.

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

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

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

By default, 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's only one encryption key, so all users must know the primary user's passcode to decrypt data. Once decrypted, all data is decrypted.

2.2 Lock screen

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 in which the caller must wait before attempting to validate another password. Any attempts before the defined amount of time has passed will be ignored by GateKeeper. Gatekeeper also keeps a count of the number of failed validation attempts since the last successful attempt. These two values together are used to prevent bruteforce attacks of the TOE's password.

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 TEE software cannot roll a device back to the vulnerable version and use keys created with the newer version.

On TOUGHBOOK, the KeyStore is implemented in secure hardware. This guarantees that even in the event of a kernel compromise, KeyStore keys are not extractable from the secure hardware.

2.3.2 KeyStore key attestation

The TOUGHBOOK also supports Key Attestation, which empowers a server to gain assurance about the properties of keys. Devices that support Google Play are provisioned at the factory with an attestation key generated by Google. The secure hardware on such devices can sign statements with the provisioned key, which attests to properties of keys protected by the secure hardware, such as the fact that the key was generated and can't leave the secure hardware. Attestation fields include purpose, padding, activate DateTime, and authTimeout. Additionally, key attestation better enables the location of important properties about the device, such as the OS version, patch level, and whether it passed Verified Boot.

Find out more information about verifying hardware-backed keys with Key Attestation.

2.3.3 KeyChain

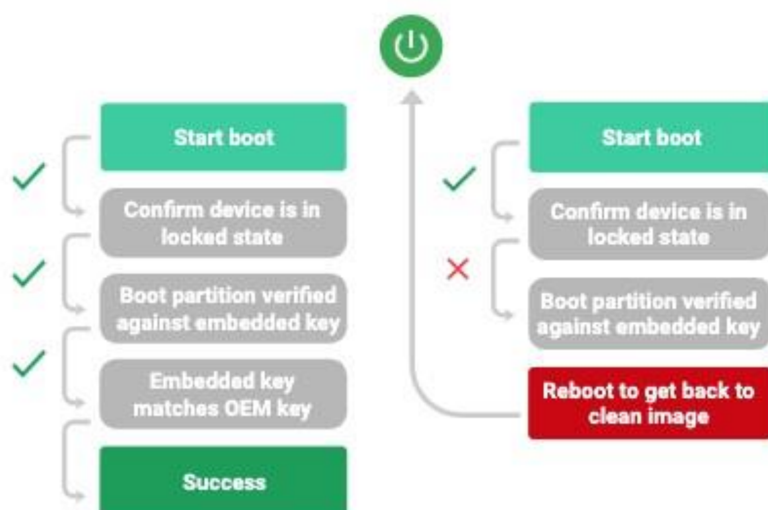
The KeyChain class allows apps to use the system credential storage for private keys and certificate chains. KeyChain is often used by Chrome, Virtual Private Network (VPN) apps, and many enterprise apps to access keys imported by the user or by the mobile device management app.
Whereas the KeyStore is for non-shareable 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 running it. 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 that code. Full boot of a compatible device with a locked bootloader proceeds only if the OS satisfies integrity checks. Verification algorithms used must be as strong as current recommendations from NIST for hashing algorithms (SHA-256) and public key sizes (RSA-2048).



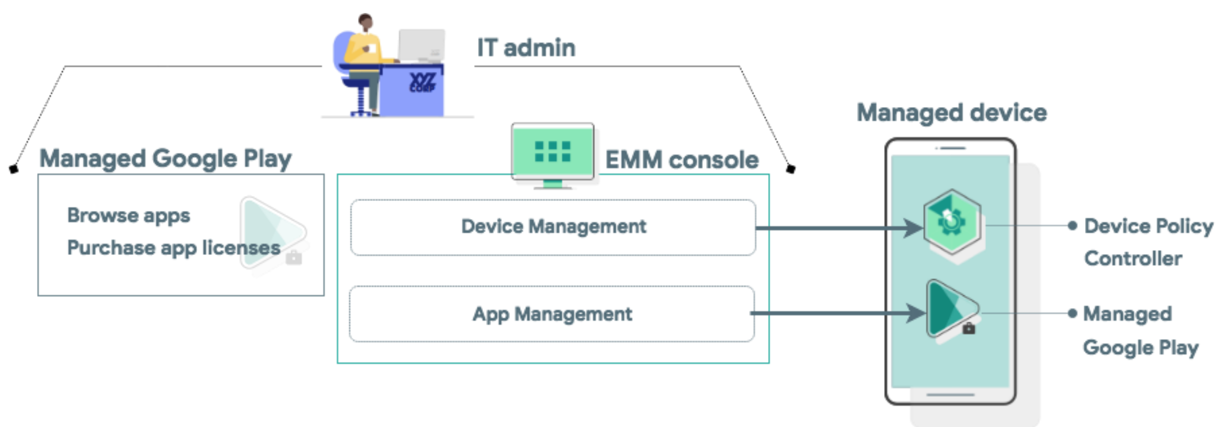
The Verified Boot state is used as an input in the process to derive disk encryption keys. If the Verified Boot state changes (e.g. the user unlocks the bootloader), then the secure hardware prevents access to data used to derive the disk encryption keys that were used when the bootloader was locked. Verified Boot on compatible devices running Android 9.0 and above require rollback protection. This means that a kernel compromise (or physical attack) cannot put an older, more vulnerable, version of the OS on your system and boot it. Additionally, rollback protection state is also stored in tamper-evident storage.

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

Find out more about Verified Boot here.

2.5 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 EMM/MDM Application Catalog.



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 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 EMM API developer documentation.

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

2.6 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

You can learn more about work profile separation 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 package to be set as an Always On VPN. IT admins can use managed configurations to specify the VPN settings for an app.

You can read more about VPN configuration options 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:

- IT admins can 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.
- IT admins can review enterprise security logs for a given device and configurable time window, in the EMMs console.
- IT admins can export enterprise security logs from the EMMs console.

IT admins can also capture relevant logging information by using Logcat which does not require any additional configuration to be enabled.

You can see a detailed audit logging table in section 9, along with information on how to view and export the different types of audit logs.

3. Security Configuration

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

3.1 Common Criteria Mode

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

1. Require a lockscreen password
   Please review the Password Management items in section 3.4 (Common Criteria Related Settings).

2. Disable Debugging Features (Developer options)
   By default Debugging features are disabled. The system administrator can prevent the user from enabling them by using DISALLOW_DEBUGGING_FEATURES().

3. Disable installation of applications from unknown sources
   This can be disabled by using DISALLOW_INSTALL_UNKNOWN_SOURCES().

4. Enable Audit Logging
   Audit Logging can be enabled using setSecurityLoggingEnabled.
   For certain items, Logcat can be used which does not require any additional enablement.

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

3.2 Cryptographic Module Identification

The following crypto modules are part of the product

- BoringSSL Library
- The TOE's Trusted Execution Environment's Kernel Software
- Hardware Cryptography which the TOE's application processor (Snapdragon 660) provides

No special configuration is required in order to use them on NIAP certified device.
During boot, each of the above cryptographic modules run a series of self-tests.

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 might grant the permission automatically or might prompt the user to approve the request.
A central design point of the Android security architecture is that no 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 pre-grant or pre-deny specific permissions using PERMISSION_GRANT_STATE API's. In addition, the end user can revoke a specific app's permission by:

1. Tapping on Settings>Apps and notifications
2. Tapping on the particular app and then tapping on Permissions
3. From there the user can toggle off any specific permission

You can learn more about Android Permissions on developer.android.com.

3.4 Common Criteria Related Settings

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

Security Feature	Setting	Description	Required Value	API	User Interface
Encryption	Device Encryption	Encrypts all internal storage	N/A	Encryption on by default with no way to turn off	
	Wipe Device	Removes all data from device	No required value	wipeData()	To wipe the device go to Settings>System>Reset options and select **Erase all data (factory reset)**
	Wipe Enterprise Data	Remove all enterprise data from device	No required value	wipeData() called from secondary user	
Password Management	Password Length	Minimum number of characters in a password	No required value	setPasswordMinimumLength()	To set a screen lock go to Settings>Security & location>Screen lock and tap on **Password**
	Password Complexity	Specify the type of characters required in a password	No required value	setPasswordQuality()	To set a screen lock go to Settings>Security & location>Screen lock and tap on **Password**
	Password Expiration	Maximum length of time before a password must change	No required value	setPasswordExpirationTimeout()	
	Authentication Failures	Maximum number of authentication failures	50 or less	setMaximumFailedPasswordsForWipe()	

Lockscreen	Inactivity to lockout	Time before lockscreen is engaged	No required value	setMaximumTimeToLock()	To set an inactivity lockout go to Settings>Security & location> and tap on the gear icon next to Screen lock then tap on Automatically lock and select the appropriate value
	Banner	Banner message displayed on the lockscreen	Administrator or user defined text	setDeviceOwnerLockScreenInfo	To set a banner go to Settings>Security & location>Lock screen preferences>Lock screen message. Set a message and tap **Save**
	Remote Lock	Locks the device remotely	Function must be available	lockNow()	
	Power Button Lock	Locks the device using the power button	Enable/Disable are available options	Tap on 'Settings>Security>Screen Lock>Power button instantly locks	Tap the power button to turn off the screen which locks the device
	Show Password	Disallows the displaying of the password on the screen of lock-screen password	Disable	This is disabled by default	
	Notifications	Controls whether notifications are displayed on the lockscreen	Enable/Disable are available options	KEYGUARD_DISABLE_SECURE_NOTIFICATIONS() KEYGUARD_DISABLE_UNREDACTED_NOTIFICATIONS	
Certificate Management	Import CA Certificates	Import CA Certificates into the Trust Anchor Database or the credential storage	No required value	installCaCert()	Tap on Settings>Security & location>Advanced> Encryption & credentials and select **Install from storage** Select a PEM-encoded CA certificate
	Remove Certificates	Remove certificates from the Trust Anchor Database or the credential storage	No required value	uninstallCACert()	To clear all user installed certificates tap on Settings>Security & location>Advanced> Encryption & credentials and select Clear credentials To remove a specific user installed certificate tap on Settings>Security & location>Advanced> Encryption & credentials>Trusted

					credentials. Switch to the User tab, select the certificate you want to delete and tap on **Remove**
	Import Client Certificates	Import client certificates in to Keychain	No required value	installKeyPair()	Tap on Settings>Security & location>Advanced> Encryption & credentials and select **Install from storage** Select a P12-encoded client certificate.
	Remove Client Certificates	Remove client certificates from Keychain	No required value	removeKeyPair()	To remove a specific user installed client certificate tap on Settings>Security & location>Advanced> Encryption & credentials>User credentials. Switch to the User tab, select the certificate you want to delete and tap on **Remove**
Radio Control	Control Wi-Fi	Control access to Wi-Fi	Enable/Disable are available options	DISALLOW_CONFIG_WIFI()	To disable Wi-Fi tap on Settings>Network & internet and toggle Airplane mode to **On**
	Control GPS	Control access to GPS	Enable/Disable are available options	DISALLOW_SHARE_LOCATION() DISALLOW_CONFIG_LOCATION()	
	Control Cellular	Control access to Cellular	Enable/Disable are available options	DISALLOW_CONFIG_MOBILE_NETWORKS()	To disable Cellular tap on Settings>Network & internet>Mobile network and tap on your carrier and toggle to **Off**
	Control NFC	Control access to NFC	Enable/Disable are available options	DISALLOW_OUTGOING_BEAM()	To disable NFC tap on Settings>Connected devices>Connection preferences and toggle NFC to **Off**
	Control Bluetooth	Control access to Bluetooth	Enable/Disable are available options	DISALLOW_BLUETOOTH () DISALLOW_BLUETOOTH_SHARING() DISALLOW_CONFIG_BLUETOOTH()	

	Control Location Service	Control access to Location Service	Enable/Disable are available options	DISALLOW_SHARE_LOCATION() DISALLOW_CONFIG_LOCATION()	
Wi-Fi Settings	Specify Wi-Fi SSIDs	Specify Wi-Fi SSID values for connecting to Wi-Fi.	No required value	WifiEnterpriseConfig()	
	Set WLAN CA Certificate	Select the CA Certificate for the Wi-FI connection	No required value	WifiEnterpriseConfig()	
	Specify security type	Specify the connection security (WEP, WPA2, etc)	No required value	WifiEnterpriseConfig()	
	Select client credentials	Specify the client credentials to access a specified WLAN	No required value	WifiEnterpriseConfig()	
Hardware Control	Control Microphone (across device)	Control access to microphone across the device	Enable/Disable are available options	DISALLOW_UNMUTE_MICROPHONE()	
	Control Microphone (per-app basis)	Control access to microphone per application	Enable/Disable are available options	Tap on 'Settings>Apps & notifications>App permissions>Microphone' then de-select the apps to remove permissions	Control Microphone (per-app basis)
	Control Camera (per-app basis)	Control access to camera per application	Enable/Disable are available options	Tap on 'Settings>Apps & notifications>App permissions>Camera' then de-select the apps to remove permissions	Control Camera (per-app basis)
	Control USB Mass Storage	Control access to mounting the device for storage over USB.	Enable/Disable are available options	DISALLOW_MOUNT_PHYSICAL_MEDIA()	
	Control USB Debugging	Control access to USB debugging.	Enable/Disable are available options	DISALLOW_DEBUGGING_FEATURES()	

	Control USB Tethered Connections	Control access to USB tethered connections.	Enable/Disable are available options	DISALLOW_CONFIG_TETHERING()	
	Control Bluetooth Tethered Connections	Control access to Bluetooth tethered connections.	Enable/Disable are available options	DISALLOW_CONFIG_TETHERING()	
	Control Hotspot Connections	Control access to Wi-Fi hotspot connections	Enable/Disable are available options	DISALLOW_CONFIG_TETHERING()	
	Automatic Time	Allows the device to get time from the Wi-Fi connection	Enable/Disable are available options	setAutoTimeEnabled()	Settings > system > Date & time > Automatic date & time
Application Control	Install Application	Installs specified application	No required value	PackageInstaller.Session()	
	Uninstall Application	Uninstalls specified application	App to uninstall	uninstall()	To uninstall an application tap on Settings>Applications & notifications>See all. Select the application and tap on Uninstall
	Application Whitelist	Specifies a list of applications that may be installed	No required value	This is done by the EMM/MDM when they setup an application catalog which leverages PackageInstaller.Session()	
	Application Blacklist	Specifies a list of applications that may not be installed	No required value	PackageInstaller.SessionInfo()	
	Application Repository	Specifies the location from which applications may be installed	No required value	DISALLOW_INSTALL_UNKNOWN_SOURCES()	

3.5 Password Recommendations

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

- Does not use known information about yourself (such as pet's names, your name, kid's 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.

An MDM application can set the number of failed unlock attempts before a device is factory reset. The administrator can set the number of unlock attempts to be anywhere from 02,147,483,648 attempts. A few warnings to the administrator:

1. Putting in a value of 0 or a negative integer results in the device never wiping.
2. While the device can be configured to have a very large number of attempted logins, it is strongly recommended to use an integer value of 20 or less. Anything more increases the chances of a brute-force password exploit attempt.

3.6 Bug Reporting Process

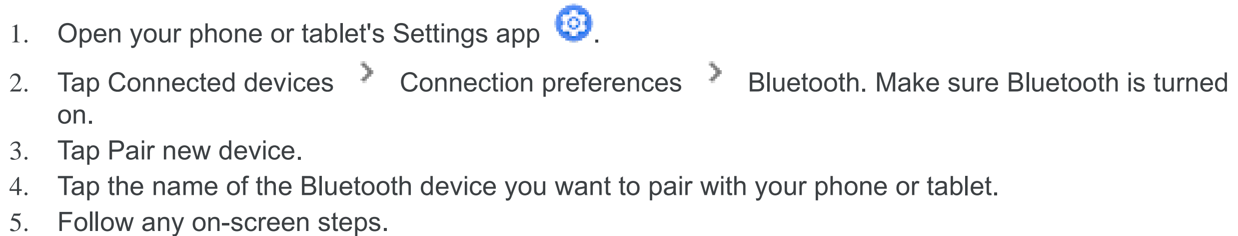
Google supports a bug filing system for the Android OS outlined here: https://source.android.com/setup/contribute/report-bugs.

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.

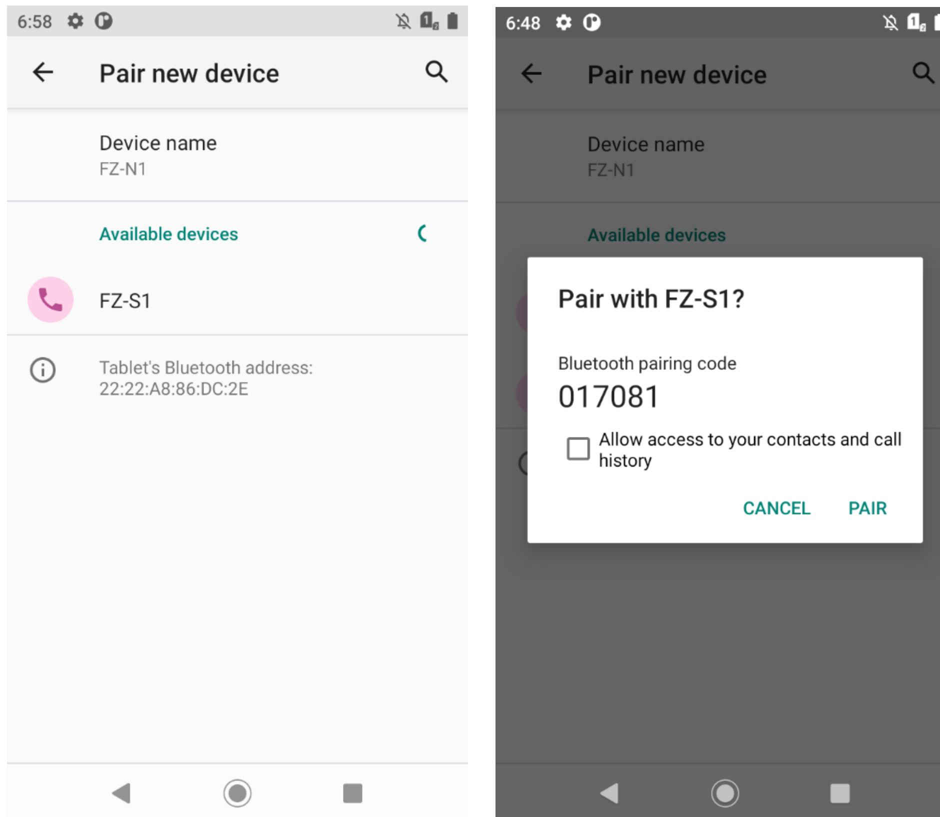
4. Bluetooth Configuration

Follow the below steps to pair and connect using Bluetooth

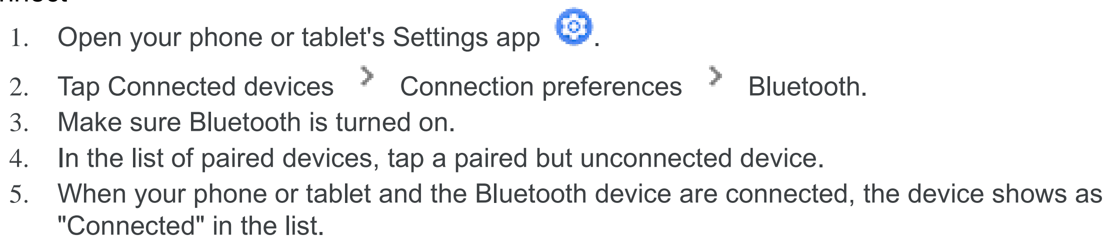
Pair



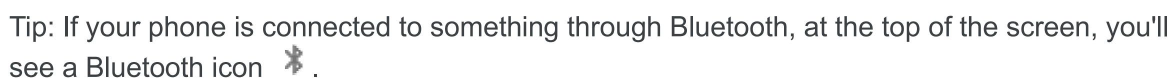
1. Open your phone or tablet's Settings app .
2. Tap Connected devices > Connection preferences > Bluetooth. Make sure Bluetooth is turned on.
3. Tap Pair new device.
4. Tap the name of the Bluetooth device you want to pair with your phone or tablet.
5. Follow any on-screen steps.



Connect

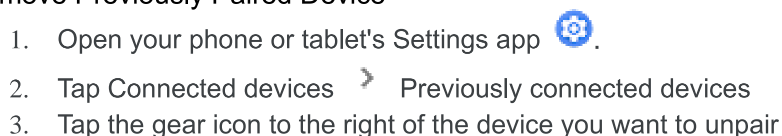


1. Open your phone or tablet's Settings app .
2. Tap Connected devices > Connection preferences > Bluetooth.
3. Make sure Bluetooth is turned on.
4. In the list of paired devices, tap a paired but unconnected device.
5. When your phone or tablet and the Bluetooth device are connected, the device shows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



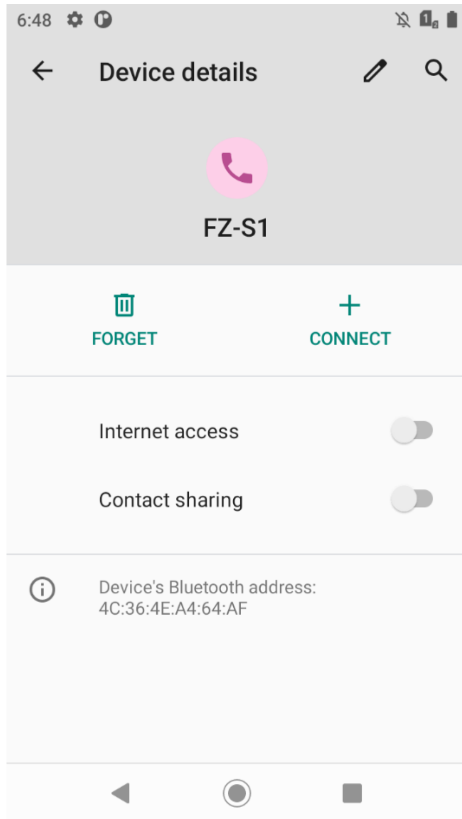
Tip: If your phone is connected to something through Bluetooth, at the top of the screen, you'll see a Bluetooth icon .

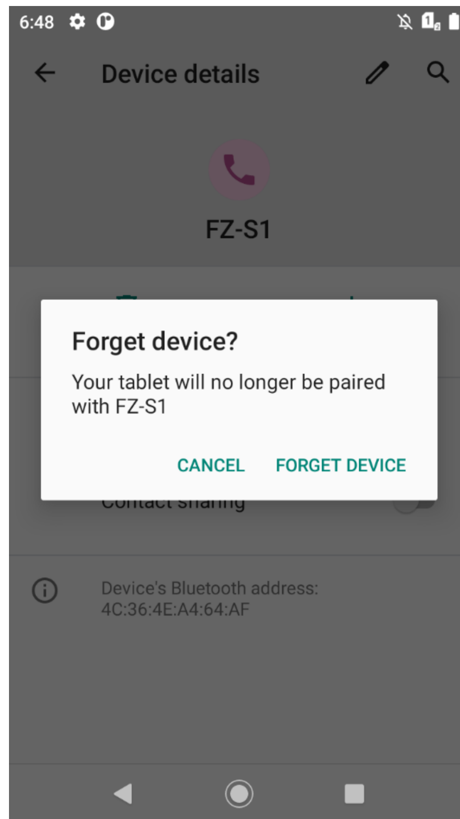
Remove Previously Paired Device



1. Open your phone or tablet's Settings app .
2. Tap Connected devices > Previously connected devices
3. Tap the gear icon to the right of the device you want to unpair

4. Tap on Forget and confirm in the popup window by tapping on Forget device





5. Wi-Fi Configuration

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

IT admins can silently provision enterprise WiFi 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

This step is applicable for connection to an access point.

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

IT admins can lock down corporate WiFi configurations in either of the following configurations:

- Users cannot modify any WiFi configurations provisioned by the EMM, but may add and modify their own user-configurable networks (for instance personal networks).
- Users cannot add or modify any WiFi network on the device, limiting WiFi connectivity to just those networks provisioned by the EMM.

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

1. Enable Developer Options by tapping on Settings>About phone and tapping on Build number five times until they see that Developer options has been enabled
2. Enable Android Debug Bridge (ADB) over USB by tapping on Settings>System>Advanced>Developer options and scroll down to USB debugging and enable the toggle to On
3. Connect the device to a workstation that has ADB installed and type in “adb shell settings put global captive_portal_mode 0” 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 “0”
5. Turn off Developer options by tapping on Settings>System>Advanced>Developer options and toggling the On option to Off at the top

If a WiFi connection unintentionally terminates, the end user will need to reconnect to reestablish 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>More>VPN**. Always-on VPN can also be enabled manually from the settings menu.

- **Per User VPN**—On multi-user devices, VPNs are applied per Android user, so all network traffic is routed through a VPN without affecting other users on the device. VPNs are applied per work profile, which allows an IT administrator to specify that only their enterprise network traffic goes through the enterprise-work profile VPN—not the user's personal network traffic.
- **Per Application VPN** — This enables VPN connections on allowed apps and to prevent VPN connections on disallowed apps.

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 with the primary profile, the work challenge is verified within secure hardware, ensuring that it's 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

System software updates are validated using a digital signature. Should this verification fail, the software update will fail and the update will not be installed. The public key which is used this verification is stored in the system image, and this system image is also validated using a digital signature, which is verified by ultimately hardware protected key.
The TOUGHBOOK will notify the user when the over-the-air update is available. Alternatively, Panasonic official website will offer the downloadable update package.

9. Audit Logging

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

Audit events from the Security Log are those where the "Keyword" field appears first in the format. For example: <Keyword> (<Date><Timestamp>): <message>

The table below provides audit events:

Requirement	Auditable Events	Additional Audit Record Requirements	Log Events & Examples
FAU_GEN.1	Start-up and shutdown of the audit functions		<Keyword> (<Date><Timestamp>): <message> Start-up: LOGGING_STARTED (Thu Sep 27 14:10:17 EDT 2018): Shutdown: All logs are stored in memory. When audit functions are disabled, all memory being used by the audit functions is released by the OS, and so this log cannot be seen.
	All administrative actions	**See Management Function Audits sheet**	
	Start-up and shutdown of the Rich OS		<Keyword> (<Date><Timestamp>): <message> Start-up: OS_STARTUP (Thu Sep 27 14:36:30 EDT 2018): 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.
FCS_CKM_EXT.1	[None]	No additional information	
FCS_CKM_EXT.5	[None]	No additional information.	
FCS_CKM.1	[None]	No additional information.	
FCS_STG_EXT.1	Import of key	Identity of key.Role and identity	<Keyword> (<Date><Timestamp>): <message> KEY_IMPORTED (Tue Mar 19 17:25:52 EDT 2019): 1

		of requestor	USRPKEY_GSS-Test-Cert 1000
	Destruction of key	Identity of key. Role and identity of requestor	<Keyword> (<Date><Timestamp>): <message> KEY_DESTROYED (Thu Sep 27 15:02:02 EDT 2018): 1 USRPKEY_android_pay_recent_unlock_key_2 10007
FCS_STG_EXT.3	Failure to verify integrity of stored key	Identity of key being verified	<Keyword> (<Date><Timestamp>): <message> KEY_INTEGRITY_VIOLATION (Fri Mar 22 20:38:19 EDT 2019): USRPKEY_f5c08a18548827f10f70006a7f0aa00b0902a75a 1000
FDP_DAR_EXT.1	[None]	No additional information.	
FDP_DAR_EXT.2	Failure to encrypt/decrypt data	No additional information.	<Date> <Time> <ID> | <Keyword> <Message> 08-22 10:51:10.471 19699 19699 W KeyStore: KeyStore exception 08-22 10:51:10.471 19699 19699 W KeyStore: android.os.ServiceSpecificException: (code 7) ... 08-22 10:51:10.471 19699 19699 W KeyStore: at com.android.certifications.niap.niapsecbio.crypto.SecureKeyStore.keyExists(SecureKeyStore.java:58) 08-22 10:51:10.471 19699 19699 W KeyStore: at com.android.certifications.niap.niapsecbio.crypto.FileCipher$EncryptedFileOutputStream.write(FileCipher.java:154) ... 08-22 10:51:10.472 19699 19699 W KeyStore: KeyStore exception 08-22 10:51:10.472 19699 19699 W KeyStore: android.os.ServiceSpecificException: (code 7) ... 08-22 10:51:10.472 19699 19699 W KeyStore: at com.android.certifications.niap.niapsecbio.crypto.SecureKeyStore.keyExists(SecureKeyStore.java:58) 08-22 10:51:10.472 19699 19699 W KeyStore: at com.android.certifications.niap.niapsecbio.crypto.FileCipher$EncryptedFileOutputStream.write(FileCipher.java:154)
FDP_STG_EXT.1	Addition or removal of certificate from Trust Anchor Database	Subject name of certificate	<Keyword> (<Date><Timestamp>): <message> CERT_AUTHORITY_INSTALLED (Thu Sep 27 15:52:36 EDT 2018): 1 cn=gossamer rsa root ca,1.2.840.113549.1.9.1=#161a726f6f7463612d72736140676f7373616d65727365632e636f6d,o=gss,l=catonsville,st=md,c=us
FIA_X509_EXT.1	Failure to validate X.509v3 certificate	Reason for failure of validation	<Date> <Time> <ID> | <Keyword> <Message> 02-21 19:12:35.488 1942 1973 W System.err: javax.net.ssl.SSLHandshakeException: java.security.cert.CertPathValidatorException: Trust anchor for certification path not found. 03-22 18:56:23.043 31552 31571 W System.err: java.security.cert.CertPathValidatorException: Certificate has been revoked, reason: UNSPECIFIED, revocation date: Tue Jan 15 12:30:07 EST 2019, authority: emailaddress=server-ocsp-subsubca-rsa@gossamersec.com, CN=server-ocsp-subsubca-rsa, O=GSS, L=Catonsville, ST=MD, C=US, extension OIDs: []
FMT_SMF_EXT.2	[None]	[None]	
FPT_NOT_EXT.1	[None]	No additional information.	
FPT_TST_EXT.1	Initiation of self-test	[None]	<Keyword> (<Date><Timestamp>): <message> CRYPTO_SELF_TEST_COMPLETED (Thu Sep 27 14:10:17 EDT 2018): 1

	Failure of self-test		<Keyword> (<Date><Timestamp>): <message> CRYPTO_SELF_TEST_COMPLETED (Wed Mar 20 23:13:45 EDT 2018): 0
FPT_TST_EXT.2(1) (Selection is optional)	Start-up of TOE	No additional information.	<Keyword> (<Date><Timestamp>): <message> OS_STARTUP (Thu Sep 27 14:36:30 EDT 2018): orange enforcing
	[None]	No additional information.	
WLAN EP Audit Logs:			
FCS_TLSC_EXT.1/WLAN	Failure to establish an EAP-TLS session	Reason for failure	<Date> <Time> <ID> | <Keyword> <Message> 03-20 18:25:46.017 10553 10553 W wpa_supplicant: TLS: Certificate verification failed, error 26 (unsupported certificate purpose) depth 0 for '/C=US/ST=MD/L=Catonsville/O=GSS/CN=tl9-16x.gss.com/emailAddress=server-no-auth-eku-rsa@gossamersec.com'
	Establishment/ termination of an EAP-TLS session	Non-TOE endpoint of connection	<Date> <Time> <ID> | <Keyword> <Message> Establishment: 09-01 09:18:52.621 3773 3773 I wpa_supplicant: wlan0: CTRL-EVENT-CONNECTED - Connection to 9c:4e:36:87:88:2c completed [id=0 id_str=%7B%22configKey%22%3A%22%5C%22nanoPC-EAP%5C%22WPA_EAP%22%2C%22creatorUid%22%3A%221000%22%7D] Termination: 03-20 18:28:56.358 10834 10834 I wpa_supplicant: wlan0: CTRL-EVENT-DISCONNECTED bssid=9c:4e:36:87:88:2c reason=1 locally_generated=1
FPT_TST_EXT.1/WLAN (note: can be performed by TOE or TOE platform)	Execution of this set of TSF self-tests. [*None*]	No additional information.	<Date> <Time> <ID> | <Keyword> <Message> Performed as part of MDFPP self-tests
FTA_WSE_EXT.1	All attempts to connect to access points		<Date> <Time> <ID> | <Keyword> <Message> 03-20 18:18:38.487 9135 9135 I wpa_supplicant: wlan0: Trying to associate with SSID 'nanoPC-EAP'
FTP_ITC_EXT.1/WLAN	All attempts to establish a trusted channel	Identification of the non-TOE endpoint of the channel.	Same as above
BT Audit logs:			
FIA_BLT_EXT.1	Failed user authorization of Bluetooth device.	User authorization decision (e.g., user rejected connection, incorrect pin entry).	<Date> <Time> <ID> | <Keyword> <Message> 10-12 18:07:40.273 2500 2620 D bt_btif : btif_dm_auth_cmpl_evt() Authentication fail reason 19
	Failed user authorization for local Bluetooth Service.	Bluetooth address and name of device. Bluetooth profile. Identity of	<Date> <Time> <ID> | <Keyword> <Message> 10-09 23:25:01.930 2522 2583 D CachedBluetoothDevice: onProfileStateChanged: profile PAN, device=BC:C3:42:9F:A7:FC, newProfileState 0 10-09 23:25:05.987 2500 3336 I bt_btm_sec: btm_sec_disconnected clearing pending flag handle:8 reason:22 10-09 23:25:05.987 2500 3336 V bt_stack:

		local service with [selection: service ID, profile name].	[VERBOSE1:btm_sec.cc(4777)] btm_sec_disconnected bd_addr: bc:c3:42:9f:a7:fc name: TestDevice state: IDLE reason: sec_req: 3080
FIA_BLT_EXT.2	Initiation of Bluetooth connection.	Bluetooth address and name of device.	<Date> <Time> <ID> | <Keyword> <Message> 10-12 21:03:45.589 2500 3336 I bt_btm : btm_sec_rmt_name_request_complete PairState: IDLE RemName: TestDevice status: 0 State:3 p_dev_rec: 0x760b7e60 10-12 21:03:45.589 2500 3336 D bt_l2cap: l2c_link_sec_comp2: status=0, p_ref_data=0x712d207448, BD_ADDR=bc:c3:42:9f:a7:fc
	Failure of Bluetooth connection.	Reason for failure.	<Date> <Time> <ID> | <Keyword> <Message> 10-12 19:58:38.827 2500 3336 V bt_stack: [VERBOSE1:btm_sec.cc(4012)] btm_sec_auth_complete: Security Manager: in state: IDLE handle:12 status:dev->sec_state: bda:bc:c3:42:9f:a7:fcRName:TestDevice 10-12 19:58:38.827 2500 3336 D bt_btm : btm_sec_auth_retry Retry for missing key sm4:x13 sec_flags:0x70a8
FIA_BLT_EXT.3	Duplicate connection attempt.	BD_ADDR of connection attempt.	[NONE] rejection is performed at the HCI layer

The table below provides sample management function audits:

REQUIREMENT	FUNCTION	Required Value	AUDIT LOG
FMT_SMF_EXT.1.1 Function 1	Configure password policy		
FMT_SMF_EXT.1.1 Function 1a	a. minimum password length	Greater than or equal to 8	<Keyword> (<Date><Timestamp>): <message> PASSWORD_COMPLEXITY_SET (Thu Mar 21 00:32:16 EDT 2019): com.afwsamples.testdpc 0 0 8 393216 1 0 1 0 0 1
FMT_SMF_EXT.1.1 Function 1b	b. minimum password complexity	No required value	<Keyword> (<Date><Timestamp>): <message> PASSWORD_COMPLEXITY_SET (Wed Oct 10 14:53:19 EDT 2018): com.afwsamples.testdpc 0 0 0 65536 1 0 1 0 0 1
FMT_SMF_EXT.1.1 Function 1c	c. maximum password lifetime		<Keyword> (<Date><Timestamp>): <message> PASSWORD_EXPIRATION_SET (Wed Oct 10 14:53:50 EDT 2018): com.afwsamples.testdpc 0 0 100000
FMT_SMF_EXT.1.1 Function 2	Configure session locking policy	10 minutes or less	
FMT_SMF_EXT.1.1 Function 2a	a. screen-lock enabled/disabled	Enabled	<Keyword> (<Date><Timestamp>): <message> PASSWORD_COMPLEXITY_SET (Wed Oct 10 14:53:19 EDT 2018): com.afwsamples.testdpc 0 0 0 65536 1 0 1 0 0 1
FMT_SMF_EXT.1.1 Function 2a	a. screen-lock enabled/disabled (after requiring a password above, admin can request the user set a password)	No required value	<Date> <Time> <ID> | <Keyword> <Message> 04-04 17:21:03.110 1173 5842 I ActivityManager: START u0 {act=android.app.action.SET_NEW_PASSWORD cmp=com.android.settings/.password.SetNewPasswordActivity} from uid 10160
FMT_SMF_EXT.1.1 Function 2a	a. screen-lock enabled/disabled (after requiring a password		#<Intent ID>: <Action> <Additional information/Message> #64: act=android.app.action.ACTION_PASSWORD_CHANGED flg=0x10 (has extras)

	above, admin can forcibly set a password)		
FMT_SMF_EXT.1.1 Function 2b	b. screen lock timeout	10 minutes or less	<Keyword> (<Date><Timestamp>): <message> MAX_SCREEN_LOCK_TIMEOUT_SET (Wed Oct 10 14:42:18 EDT 2018): com.afwsamples.testdpc 0 0 10000
FMT_SMF_EXT.1.1 Function 2b	b. screen lock timeout (after setting a max time, the admin can prevent any user changes with this)		<Keyword> (<Date><Timestamp>): <message> USER_RESTRICTION_ADDED (Wed Oct 10 14:56:10 EDT 2018): com.afwsamples.testdpc 0 no_config_screen_timeout
FMT_SMF_EXT.1.1 Function 2c	c. number of authentication failures	10 or less	<Keyword> (<Date><Timestamp>): <message> MAX_PASSWORD_ATTEMPTS_SET (Wed Oct 10 14:42:26 EDT 2018): com.afwsamples.testdpc 0 0 20
FMT_SMF_EXT.1.1 Function 3	Enable/disable the VPN protection	Enable/Disable	
FMT_SMF_EXT.1.1 Function 3a	a. across device		<Keyword> (<Date><Timestamp>): <message> USER_RESTRICTION_ADDED (Wed Oct 10 14:49:57 EDT 2018): com.afwsamples.testdpc 0 no_config_vpn
FMT_SMF_EXT.1.1 Function 4	enable/disable [assignment: list of all radios]		
FMT_SMF_EXT.1.1 Function 4a	Enable/disable [Bluetooth]	Enable/Disable	<Keyword> (<Date><Timestamp>): <message> USER_RESTRICTION_ADDED (Wed Oct 10 14:43:06 EDT 2018): com.afwsamples.testdpc 0 no_bluetooth
FMT_SMF_EXT.1.1 Function 4b	Enable/disable [GPS]	Enable/Disable	<Keyword> (<Date><Timestamp>): <message> USER_RESTRICTION_ADDED (Wed Oct 10 14:47:22 EDT 2018): com.afwsamples.testdpc 0 no_config_location USER_RESTRICTION_REMOVED (Wed Oct 10 16:33:02 EDT 2018): com.afwsamples.testdpc 0 no_config_location
FMT_SMF_EXT.1.1 Function 4c	Enable/disable [NFC]	Enable/Disable	<Keyword> (<Date><Timestamp>): <message> USER_RESTRICTION_ADDED (Wed Oct 10 14:46:52 EDT 2018): com.afwsamples.testdpc 0 no_outgoing_beam
FMT_SMF_EXT.1.1 Function 4d	Enable/disable [Wi-Fi]	Enable/Disable	User feature only
FMT_SMF_EXT.1.1 Function 4e	Enable/disable [Cellular/Mobile	Enable/Disable	User feature only

	data]		
FMT_SMF_EXT.1.1 Function 5a	Enable/disable [microphone] a. across device	Enable/Disable	<Keyword> (<Date><Timestamp>): <message> USER_RESTRICTION_ADDED (Wed Oct 10 14:55:44 EDT 2018): com.afwsamples.testdpc 0 no_unmute_microphone
FMT_SMF_EXT.1.1 Function 5b	Enable/disable [camera, microphone] c. On a per-app basis	Enable/Disable	User feature only
FMT_SMF_EXT.1.1 Function 6	Transition to the locked state		<Keyword> (<Date><Timestamp>): <message> REMOTE_LOCK (Thu Oct 11 11:34:16 EDT 2018): com.afwsamples.testdpc 0 0
FMT_SMF_EXT.1.1 Function 7	Full wipe of protected data		The act of wiping protected data resets the phone, and no audit record available. Wiping of phone available to the administrator via an MDM API and to the user using factory reset.
FMT_SMF_EXT.1.1 Function 8a	Configure application installation policy a. restricting the sources of applications	Disable	<Keyword> (<Date><Timestamp>): <message> USER_RESTRICTION_REMOVED (Wed Oct 10 14:47:00 EDT 2018): com.afwsamples.testdpc 0 no_install_unknown_sources
FMT_SMF_EXT.1.1 Function 8a	Configure application installation policy a. restricting the sources of applications	Enable	<Keyword> (<Date><Timestamp>): <message> USER_RESTRICTION_ADDED (Wed Oct 10 14:47:00 EDT 2018): com.afwsamples.testdpc 0 no_install_unknown_sources
FMT_SMF_EXT.1.1 Function 8c	Configure application installation policy c. denying installation of applications	Enable	<Keyword> (<Date><Timestamp>): <message> USER_RESTRICTION_ADDED (Wed Oct 10 14:47:03 EDT 2018): com.afwsamples.testdpc 0 no_install_apps
FMT_SMF_EXT.1.1 Function 9	Import keys/secrets to secure key store		<Keyword> (<Date><Timestamp>): <message> KEY_IMPORTED (Fri Mar 08 11:37:39 EST 2019): 1 USRPKEY_client-rsa 1000

FMT_SMF_EXT.1.1 Function 10	Destroy imported keys/secrets in secure key store		<Keyword> (<Date><Timestamp>): <message> KEY_DESTROYED (Fri Mar 08 11:37:43 EST 2019): 1 USRPKEY_client-rsa 1000
FMT_SMF_EXT.1.1 Function 11	Import X.509v3 certificates into the Trust Anchor Database		<Keyword> (<Date><Timestamp>): <message> CERT_AUTHORITY_INSTALLED (Wed Oct 10 14:45:25 EDT 2018): 1 cn=gossamer rsa root ca,1.2.840.113549.1.9.1=#161a726f6f7463612d72736140676f7373616d65727365632e636f6d,o=gss,l=catonsville,st=md,c=us
FMT_SMF_EXT.1.1 Function 12	Remove imported X.509v3 certificates and [no other certificates] from the Trust Anchor Database		<Keyword> (<Date><Timestamp>): <message> CERT_AUTHORITY_REMOVED (Wed Oct 10 14:45:31 EDT 2018): 1 cn=gossamer rsa root ca,1.2.840.113549.1.9.1=#161a726f6f7463612d72736140676f7373616d65727365632e636f6d,o=gss,l=catonsville,st=md,c=us
FMT_SMF_EXT.1.1 Function 13	Enroll TOE in management		Logs do not exist until after the MDM agent has been installed and enabled security logging.
FMT_SMF_EXT.1.1 Function 14	Remove apps		#<Intent ID>: <Action> <Additional information/Message> #126: act=android.intent.action.PACKAGE_REMOVED dat=package:org.fdroid.fdroid flg=0x4000010 (has extras) +1ms dispatch +3ms finish enq=2019-03-18 11:31:55 disp=2019-03-18 11:31:55 fin=2019-03-18 11:31:55 extras: Bundle[{android.intent.extra.REMOVED_FOR_ALL_USERS=false, android.intent.extra.DONT_KILL_APP=false, android.intent.extra.UID=10172, android.intent.extra.DATA_REMOVED=false, android.intent.extra.user_handle=0}]
FMT_SMF_EXT.1.1 Function 15	Update system software		06-14 12:56:03.444 18996 20854 I SystemUpdate: [Execution,InstallationIntentOperation] Received intent: Intent { act=com.google.android.gms.update.INSTALL_UPDATE cat=[targeted_intent_op_prefix:.update.execution.InstallationIntentOperation] cmp=com.google.android.gms/.chimera.GmsIntentOperationService }.
FMT_SMF_EXT.1.1 Function 16	Install apps		#<Intent ID>: <Action> <Additional information/Message> #110: act=android.intent.action.PACKAGE_ADDED dat=package:org.fdroid.fdroid flg=0x4000010 (has extras) +2ms dispatch +2ms finish enq=2019-03-18 11:23:59 disp=2019-03-18 11:23:59 fin=2019-03-18 11:23:59 extras: Bundle[{android.intent.extra.UID=10172, android.intent.extra.user_handle=0}]
FMT_SMF_EXT.1.1 Function 17	Remove Enterprise apps		#<Intent ID>: <Action> <Additional information/Message> #259: act=android.intent.action.PACKAGE_REMOVED dat=package:org.fdroid.fdroid flg=0x4000010 pkg=com.afwsamples.testdpc (has extras) +3ms dispatch 0 finish enq=2019-03-18 11:23:59 disp=2019-03-18 11:23:59 fin=2019-03-18 11:23:59 extras: Bundle[{android.intent.extra.REMOVED_FOR_ALL_USERS=false,

			android.intent.extra.DONT_KILL_APP=false, android.intent.extra.UID=1210172, android.intent.extra.DATA_REMOVED=false, android.intent.extra.REPLACING=true, android.intent.extra.user_handle=12}]
FMT_SMF_EXT.1.1 Function 18	Configure the Bluetooth trusted channel		
FMT_SMF_EXT.1.1 Function 18a	a. disable/enable the Discoverable mode (for BR/EDR)		User feature only
FMT_SMF_EXT.1.1 Function 18b	b. change the Bluetooth device name		User feature only
FMT_SMF_EXT.1.1 Function 19	Enable/disable notifications in locked state f. all notifications	Enable/Disable	<Keyword> (<Date><Timestamp>): <message> KEYGUARD_DISABLED_FEATURES_SET (FRI MAR 08 14:01:02 EST 2019): com.afwsamples.testdpc 0 0 <0/4/8/12> 0 = no restriction 4 = disable unredated notifications ("Show all notification content" setting selection greyed out) 8 = disable secure notifications ("Hide sensitive content" & "Show all notification content" greyed out) 12 = both 4 and 8 (but 8 supersedes 4 anyway)
FMT_SMF_EXT.1.1 Function 20	Enable DAR protection		N/A - DAR cannot be disabled
FMT_SMF_EXT.1.1 Function 22	Enable/disable location services	Enable/Disable	
FMT_SMF_EXT.1.1 Function 22a	a. across device		<Keyword> (<Date><Timestamp>): <message> USER_RESTRICTION_ADDED (Wed Oct 10 14:47:13 EDT 2018): com.afwsamples.testdpc 0 no_share_location
FMT_SMF_EXT.1.1 Function 23	Enable/disable the use of [Biometric Authentication Factor]	Enable	<Keyword> (<Date><Timestamp>): <message> KEYGUARD_DISABLED_FEATURES_SET (Wed Oct 10 14:57:53 EDT 2018): com.afwsamples.testdpc 0 0 420
FMT_SMF_EXT.1.1 Function 25	Enable/disable [Wi-Fi tethering, USB tethering, and Bluetooth tethering]	Disable	<Keyword> (<Date><Timestamp>): <message> USER_RESTRICTION_ADDED (Wed Oct 10 14:58:21 EDT 2018): com.afwsamples.testdpc 0 no_config_tethering USER_RESTRICTION_REMOVED (Wed Oct 10 16:37:47 EDT 2018): com.afwsamples.testdpc 0 no_config_tethering

FMT_SMF_EXT.1.1 Function 26	Enable/disable developer modes	Disable	<Keyword> (<Date><Timestamp>): <message> USER_RESTRICTION_ADDED (Thu Oct 11 11:48:28 EDT 2018): com.afwsamples.testdpc 0 no_debugging_features USER_RESTRICTION_REMOVED (Thu Oct 11 11:48:30 EDT 2018): com.afwsamples.testdpc 0 no_debugging_features
FMT_SMF_EXT.1.1 Function 28	Wipe Enterprise Data		#<Intent ID>: <Action> <Additional information/Message> #30: act=android.intent.action.MANAGED_PROFILE_REMOVED flg=0x50000010 (has extras)
FMT_SMF_EXT.1.1 Function 41	Enable/disable [Hotspot functionality, USB tethering]	Enable/Disable	<Keyword> (<Date><Timestamp>): <message> USER_RESTRICTION_ADDED (Wed Oct 10 14:58:21 EDT 2018): com.afwsamples.testdpc 0 no_config_tethering
FMT_SMF_EXT.1.1 Function 44	Unenroll TOE from management		#<Intent ID>: <Action> <Additional information/Message> #14: act=android.app.action.DEVICE_ADMIN_DISABLED flg=0x10
FMT_SMF_EXT.1.1 Function 45	Enable/disable the Always On VPN protection	Enable	<Keyword> (<Date><Timestamp>): <message> SettingsProvider: Notifying for 0: content://settings/secure/always_on_vpn_app 03-28 10:37:05.091 1400 1400 VSettingsProvider: Notifying for 0: content://settings/secure/always_on_vpn_lockdown
FMT_SMF_EXT.1.1 Function 48	Configure Security Policy for each wireless network:		<Date> <Time> <ID> | <Keyword> <Message> 05-28 18:57:54.859 1220 4568 I addOrUpdateNetwork: uid = 10161 SSID "GSSSSID" nid=-1 05-28 18:58:31.667 1220 1539 I addOrUpdateNetwork: uid = 10161 SSID GSSEAPTLS nid=-1
FMT_SMF_EXT.1.1 Function 48a	a. [selection: specify the CA(s) from which the TSF will accept WLAN authentication server certificate(s), specify the FQDN(s) of acceptable WLAN authentication server certificate(s)]		See Function 48
FMT_SMF_EXT.1.1 Function 48b	b. security type		See Function 48

FMT_SMF_EXT.1.1 Function 48c	c. authentication protocol		See Function 48
FMT_SMF_EXT.1.1 Function 48d	d. client credentials		See Function 48
FMT_SMF_EXT.1.1 Function 49	specify wireless networks (SSIDs) to which the TSF may connect		See Function 48

Audit error codes correspond to the following error code table
https://cs.android.com/android/platform/superproject/+/master:packages/modules/Bluetooth/system/stack/include/hci_error_code.h

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

Using the NIAPSEC library, sensitive data protection including 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.

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.
Encrypt and decrypt files that require sensitive data protection.

Supported Algorithms:
AES256 - AES/GCM/NoPadding
RSA4096 - RSA/ECB/OAEPWithSHA-256AndMGF1Padding

Public Constructor	
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 a*symmetric/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 a*symmetric/ephemeral* algorithms specified by the default configuration, using NIAP standards. -name - The file name -Executor - to handle the threading for BiometricPrompt. Usually Executors.newSingleThreadExecutor() -Listener for the resulting FileInputStream.

Code Examples:

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

// Open a sensitive file for writing
FileOutputStream outputStream = secureContext.openEncryptedFileOutput(FILE_NAME,
Context.MODE_PRIVATE, KEY_PAIR_ALIAS);
// Write data to the file, where DATA is a String of sensitive information.
outputStream.write(DATA.getBytes(StandardCharsets.UTF_8)); outputStream.flush();
outputStream.close();

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

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
   - This section lists all the APIs for the algorithms and random number generation
2. Key Management
   - APIs for importing, using, and destroying keys
3. Certificate Validation, TLS, HTTPS
   - API used by applications for configuring the reference identifier
   - APIs for validation checks (should match the test program provided)
   - 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. -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 a*symmetric* 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. -The Ephemeral key to use -encryptedData - the data to be decrypted -initializationVector - the IV used for encryption Returns a byte array of the clear text.

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(1) – Key Establishment (RSA)

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

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

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

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

Algorithms::
RSA/ECB/OAEPWithSHA-256AndMGF1Padding

Reference:
Cipher - https://developer.android.com/reference/javax/crypto/Cipher

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

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

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"

Reference:
Signature - https://developer.android.com/reference/java/security/Signature

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

Cipher class encrypts or decrypts a plain text.

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

// Encrypt
Cipher cipher = Cipher.getInstance(transformation);
cipher.init(Cipher.ENCRYPT_MODE, secretKey);

byte[] iv = cipher.getIV();
byte[] clearData = data.getBytes(UTF_8);
byte[] cipherText = cipher.doFinal(clearData);
Pair<byte[], byte[]> result = Pair<>(cipherText, iv);

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

Algorithms:
AES/CBC/NoPadding
AES/GCM/NoPadding

Reference:
Cipher - https://developer.android.com/reference/javax/crypto/Cipher

11.1.6 FCS_COP.1(2) – Hashing (SHA)

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

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

Algorithms:
SHA-1
SHA-256
SHA-384
SHA-512

Reference:
MessageDigest - https://developer.android.com/reference/java/security/MessageDigest

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

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

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

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

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

Algorithms:
SHA256withRSA
SHA384withRSA

Reference:
Signature - 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(4) - HMAC

Mac class calculates the hash of plaintext with key.

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

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

Algorithms:
HmacSHA1
HmacSHA256
HmacSHA384
HmacSHA512

Reference:
Mac - 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.*

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 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. Both code examples below complete

certificate path validation automatically.

By default, the device is restricted to only support TLS versions 1.0, 1.1, 1.2 and TLS ciphersuites that are RFC compliant and claimed under the MDFPP. As such, no configuration is needed to restrict or allow ciphersuites to be compliant.

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 and handles OCSP validation checks and does a hostname verification check on initiation of the connection.
boolean	isValid(String hostname, SSLSocket socket) A manual OCSP certificate and hostname check. Based on a hostname and underlying SSLSocket.
boolean	isValid(HttpsURLConnection conn) A manual OCSP certificate and hostname check. Based on an existing HttpsUrlConnection.
boolean	isValid(Certificate cert) A manual OCSP certificate check.

boolean	isValid(List<Certificate> certs) A manual OCSP certificates check.

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/PKIXRevocationChecker

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

11.3.1 Cipher Suites

A list of the supported cipher suites can be found below:

TLS_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

The NIST Curves P-256 and P-384 are supported. No configuration is needed to use them.

11.3.2 Bluetooth APIs

The TOUGHBOOK provides classes that manage Bluetooth functionality, such as scanning for devices, connecting with devices, and managing data transfer between devices. The Bluetooth API supports both "Classic Bluetooth" and Bluetooth Low Energy.

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, etc.

- 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.

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 call callback functions on events from the HID host.
BluetoothHidDeviceAppQosSettings	Represents the Quality of Service (QoS) settings for a Bluetooth HID Device application.
BluetoothHidDeviceAppSdpSettings	Represents the Service Discovery Protocol (SDP) settings for a Bluetooth HID Device application.
BluetoothManager	High level manager used to obtain an instance of an `BluetoothAdapter` and to conduct overall Bluetooth Management.
BluetoothServerSocket	A listening Bluetooth socket.
BluetoothSocket	A connected or connecting Bluetooth socket.

Below are some code example for the usage of Bluetooth interfaces:

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

Below is sample service to interact with a Bluetooth APIs.

```
// A service that interacts with the BLE device via the Android BLE API.
public class BLEService extends Service {

  private final static String TAG = "BLEService";
  private BluetoothManager mBluetoothManager
  private BluetoothAdapter mBluetoothAdapter;
  private String mBluetoothDeviceAddress;
  private BluetoothGatt mBluetoothGatt;
  private int mConnectionState = STATE_DISCONNECTED;
  private static final int STATE_DISCONNECTED = 0;
  private static final int STATE_CONNECTING = 1;
  private static final int STATE_CONNECTED = 2;

  public final static String ACTION_GATT_CONNECTED =
   "com.niap.ble.ACTION_GATT_CONNECTED";
  public final static String ACTION_GATT_DISCONNECTED =
   "com.niap.ble.ACTION_GATT_DISCONNECTED";
  public final static String ACTION_GATT_SERVICES_DISCOVERED =
   "com.niap.ble.ACTION_GATT_SERVICES_DISCOVERED";
  public final static String ACTION_DATA_AVAILABLE = "com.niap.ble.ACTION_DATA_AVAILABLE";
  public final static String EXTRA_DATA = "com.niap.ble.EXTRA_DATA";

  // Various callback methods defined by the BLE API.
  private final BluetoothGattCallback mGattCallback = new BluetoothGattCallback() {

    @Override
    public void onConnectionStateChange(BluetoothGatt gatt, int status, int newState) {
      String intentAction;
      if (newState == BluetoothProfile.STATE_CONNECTED) {
        intentAction = ACTION_GATT_CONNECTED;
        mConnectionState = STATE_CONNECTED;
        broadcastUpdate(intentAction);
        Log.i(TAG, "Connected to GATT server.");
        Log.i(TAG, "Attempting to start service discovery:" + mBluetoothGatt.discoverServices());
      } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
        intentAction = ACTION_GATT_DISCONNECTED;
        mConnectionState = STATE_DISCONNECTED;
        Log.i(TAG, "Disconnected from GATT server.");
        broadcastUpdate(intentAction);
      }
    }

    @Override
    // New services discovered
```

```
    public void onServicesDiscovered(BluetoothGatt gatt, int status) {
        if (status == BluetoothGatt.GATT_SUCCESS) {
            broadcastUpdate(ACTION_GATT_SERVICES_DISCOVERED);
        } else {
            Log.w(TAG, "onServicesDiscovered received: " + status);
        }
    }


    @Override
    // Result of a characteristic read operation
    public void onCharacteristicRead(BluetoothGatt gatt, BluetoothGattCharacteristic characteristic,
    int status) {
        if (status == BluetoothGatt.GATT_SUCCESS) {
            broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
        }
    }
};
```