Android Keystore system

The Android Keystore system lets you store cryptographic keys in a container to make them more difficult to extract from the device. Once keys are in the keystore, you can use them for cryptographic operations, with the key material remaining non-exportable. Also, the keystore system lets you restrict when and how keys can be used, such as requiring user authentication for key use or restricting keys to use only in certain cryptographic modes. See the Security Features (#SecurityFeatures) section for more information.

The keystore system is used by the `KeyChain` (/reference/android/security/KeyChain) API, introduced in Android 4.0 (API level 14), as well as the Android Keystore provider feature, introduced in Android 4.3 (API level 18). This document goes over when and how to use the Android Keystore system.

Security features

The Android Keystore system protects key material from unauthorized use in two ways. First, it reduces the risk of unauthorized use of key material from *outside* the Android device by preventing the extraction of the key material from application processes and from the Android device as a whole. Second, the keystore system reduces the risk of unauthorized use of key material *within* the Android device by making apps specify the authorized uses of their keys and then enforcing those restrictions outside of the apps' processes.

Extraction prevention

Key material of Android Keystore keys is protected from extraction using two security measures:

- Key material never enters the application process. When an app performs cryptographic operations using an Android Keystore key, behind the scenes plaintext, ciphertext, and messages to be signed or verified are fed to a system process that carries out the cryptographic operations. If the app's process is compromised, the attacker might be able to use the app's keys but can't extract their key material (for example, to be used outside of the Android device).

- Key material can be bound to the secure hardware of the Android device, such as the Trusted Execution Environment (TEE) (https://source.android.com/docs/security/features/trusty) or Secure Element (SE). When this feature is enabled for a key, its key material is never exposed outside of secure hardware. If the Android OS is compromised or an attacker can read the device's internal storage, the attacker might be able to use any app's Android Keystore keys on the Android device, but it can't extract them from the device. This feature is enabled only if the device's secure hardware supports the particular combination of key algorithm, block modes, padding schemes, and digests the key is authorized to be used with.

  To check whether the feature is enabled for a key, obtain a `KeyInfo` (/reference/android/security/keystore/KeyInfo) for the key. The next step depends on your app's target SDK version:

  - If your app targets Android 10 (API level 29) or higher, inspect the return value of `getSecurityLevel()` (/reference/android/security/keystore/KeyInfo#getSecurityLevel()). Return values matching `KeyProperties.SecurityLevelEnum.TRUSTED_ENVIRONMENT` or `KeyProperties.SecurityLevelEnum.STRONGBOX` indicate that the key resides within secure hardware.
  - If your app targets Android 9 (API level 28) or lower, inspect the boolean return value of `KeyInfo.isInsideSecurityHardware()` (/reference/android/security/keystore/KeyInfo#isInsideSecureHardware()).

Hardware security module

Supported devices running Android 9 (API level 28) or higher can have a *StrongBox Keymaster* (https://source.android.com/docs/security/features/keystore), an implementation of the Keymaster or Keymint HAL that resides in a hardware security module-like secure element. While hardware security modules can refer to many different implementations of key-storage where a Linux kernel compromise can't reveal them, such as TEE, StrongBox explicitly refers to devices such as embedded Secure Elements (eSE) or on-SoC secure processing units (iSE).

The module contains the following:

- Its own CPU

- Secure storage
- A true random-number generator
- Additional mechanisms to resist package tampering and unauthorized sideloading of apps
- A secure timer
- A reboot notification pin (or equivalent), like general-purpose input/output (GPIO)

To support low-power StrongBox implementations, a subset of algorithms and key sizes are supported:

- RSA 2048
- AES 128 and 256
- ECDSA, ECDH P-256
- HMAC-SHA256 (supports key sizes between 8 bytes and 64 bytes, inclusive)
- Triple DES
- Extended Length APDUs
- Key Attestation
- Amendment H support for upgrade

When generating or importing keys using the `KeyStore` (https://developer.android.com/reference/java/security/KeyStore) class, you indicate a preference for storing the key in the StrongBox Keymaster by passing `true` to the `setIsStrongBoxBacked()` (https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.Builder#setIsStrongBoxBacked(boolean)) method.

Note: If the StrongBox Keymaster isn't available for the given algorithm and key size associated with a key, the framework throws a `StrongBoxUnavailableException` (/reference/android/security/keystore/StrongBoxUnavailableException). If you get this exception, try using TEE for your key storage as a fallback option.

Although StrongBox is a little slower and resource constrained (meaning that it supports fewer concurrent operations) compared to TEE, StrongBox provides better security guarantees against physical and side-channel attacks. If you want to prioritize higher security guarantees over app resource efficiency, we recommend using StrongBox on the devices where it is available. Wherever StrongBox isn't available, your app can always fall back to TEE to store key materials.

Key use authorizations

To avoid unauthorized use of keys on the Android device, Android Keystore lets apps specify authorized uses of their keys when they generate or import the keys. Once a key is generated or imported, its authorizations can't be changed. Authorizations are then enforced by the Android Keystore whenever the key is used. This is an advanced security feature that is generally useful only if your requirements are that a compromise of your application process after key generation/import (but not before or during) can't lead to unauthorized uses of the key.

Supported key use authorizations fall into the following categories:

- **Cryptography:** the key can only be used with authorized key algorithms, operations, or purposes (encrypt, decrypt, sign, verify), padding schemes, block modes, or digests.
- **Temporal validity interval:** the key is authorized for use only during a defined interval of time.
- **User authentication:** the key can only be used if the user has been authenticated recently enough. See Require user authentication for key use (#UserAuthentication).

As an additional security measure for keys whose key material is inside secure hardware (see `KeyInfo.isInsideSecurityHardware()` (/reference/android/security/keystore/KeyInfo#isInsideSecureHardware()) or, for apps targeting Android 10 (API level 29) or higher, `KeyInfo.getSecurityLevel()` (/reference/android/security/keystore/KeyInfo#getSecurityLevel())), some key use authorizations might be enforced by the secure hardware, depending on the Android device. Secure hardware normally enforces cryptographic and user authentication authorizations. However, secure hardware doesn't usually enforce temporal validity interval authorizations, because it normally doesn't have an independent, secure real-time clock.

You can query whether a key's user authentication authorization is enforced by the secure hardware using `KeyInfo.isUserAuthenticationRequirementEnforcedBySecureHardware()` (/reference/android/security/keystore/KeyInfo#isUserAuthenticationRequirementEnforcedBySecureHardware())
.

Choose between a keychain and the Android Keystore provider

Use the `KeyChain` (/reference/android/security/KeyChain) API when you want system-wide credentials. When an app requests the use of any credential through the `KeyChain` API, users can choose, through a system-provided UI, which of the installed credentials an app can access. This lets several apps use the same set of credentials with user consent.

Use the Android Keystore provider to let an individual app store its own credentials, which only that app can access. This provides a way for apps to manage credentials that only they can use while providing the same security benefits that the `KeyChain` API provides for system-wide credentials. This method doesn't require the user to select the credentials.

Use the Android Keystore provider

To use this feature, you use the standard `KeyStore` (/reference/java/security/KeyStore) and `KeyPairGenerator` (/reference/java/security/KeyPairGenerator) or `KeyGenerator` (/reference/javax/crypto/KeyGenerator) classes along with the `AndroidKeyStore` provider introduced in Android 4.3 (API level 18).

`AndroidKeyStore` is registered as a `KeyStore` type for use with the `KeyStore.getInstance(type)` (/reference/java/security/KeyStore#getInstance(java.lang.String)) method and as a provider for use with the `KeyPairGenerator.getInstance(algorithm, provider)` (/reference/java/security/KeyPairGenerator#getInstance(java.lang.String, java.lang.String)) and `KeyGenerator.getInstance(algorithm, provider)` (/reference/javax/crypto/KeyGenerator#getInstance(java.lang.String, java.lang.String)) methods.

Generate a new private or secret key

To generate a new `KeyPair` containing a **`PrivateKey`** (/reference/java/security/PrivateKey), you must specify the initial X.509 attributes of the certificate. You can use **`KeyStore.setKeyEntry()`** (/reference/java/security/KeyStore#setKeyEntry(java.lang.String, java.security.Key, char[], java.security.cert.Certificate[])) to replace the certificate at a later time with a certificate signed by a certificate authority (CA).

To generate the key pair, use a **`KeyPairGenerator`** (/reference/java/security/KeyPairGenerator) with **`KeyGenParameterSpec`** (/reference/android/security/keystore/KeyGenParameterSpec):

Kotlin (#kotlin) Java (#java)

```
/*
 * Generate a new EC key pair entry in the Android Keystore by
 * using the KeyPairGenerator API. The private key can only be
 * used for signing or verification and only with SHA-256 or
 * SHA-512 as the message digest.
 */
val kpg: KeyPairGenerator = KeyPairGenerator.getInstance(
        KeyProperties.KEY_ALGORITHM_EC,
        "AndroidKeyStore"
)
val parameterSpec: KeyGenParameterSpec = KeyGenParameterSpec.Builder(
        alias,
        KeyProperties.PURPOSE_SIGN or KeyProperties.PURPOSE_VERIFY
).run {
    setDigests(KeyProperties.DIGEST_SHA256, KeyProperties.DIGEST_SHA512)
    build()
}

kpg.initialize(parameterSpec)

val kp = kpg.generateKeyPair()
```

Import encrypted keys into secure hardware

Android 9 (API level 28) and higher lets you import encrypted keys securely into the keystore using an ASN.1-encoded key format. The Keymaster then decrypts the keys in the keystore, so the content of the keys never appears as plaintext in the device's host memory. This process provides additional key decryption security.

Note: This feature is supported only on devices that ship with Keymaster 4 or higher.

To support secure importing of encrypted keys into the keystore, complete the following steps:

1. Generate a key pair that uses the `PURPOSE_WRAP_KEY` (/reference/android/security/keystore/KeyProperties#PURPOSE_WRAP_KEY) purpose. We recommend that you add attestation to this key pair as well.
2. On a server or machine that you trust, generate the ASN.1 message for the `SecureKeyWrapper`.

   The wrapper contains the following schema:

   ```
   KeyDescription ::= SEQUENCE {
       keyFormat INTEGER,
       authorizationList AuthorizationList
   }

   SecureKeyWrapper ::= SEQUENCE {
       wrapperFormatVersion INTEGER,
       encryptedTransportKey OCTET_STRING,
       initializationVector OCTET_STRING,
       keyDescription KeyDescription,
       secureKey OCTET_STRING,
       tag OCTET_STRING
   }
   ```

3. Create a `WrappedKeyEntry` (/reference/android/security/keystore/WrappedKeyEntry) object, passing in the ASN.1 message as a byte array.
4. Pass this `WrappedKeyEntry` object into the overload of `setEntry()` (/reference/java/security/KeyStore#setEntry(java.lang.String,%20java.security.KeyStore.Entry,%20java.security.KeyStore.ProtectionParameter)) that accepts a `Keystore.Entry` (/reference/java/security/KeyStore.Entry) object.

Work with keystore entries

You can access the `AndroidKeyStore` provider through all the standard **`KeyStore`** (/reference/java/security/KeyStore) APIs.

List entries

List entries in the keystore by calling the **`aliases()`** (/reference/java/security/KeyStore#aliases()) method:

KotlinJava (#java)
(#kotlin)

```
/*
 * Load the Android KeyStore instance using the
 * AndroidKeyStore provider to list the currently stored entries.
 */
val ks: KeyStore = KeyStore.getInstance("AndroidKeyStore").apply {
   load(null)
}
val aliases: Enumeration<String> = ks.aliases()
```

Sign and verify data

Sign data by fetching the **`KeyStore.Entry`** (/reference/java/security/KeyStore.Entry) from the keystore and using the **`Signature`** (/reference/java/security/Signature) APIs, such as **`sign()`** (/reference/java/security/Signature#sign()):

KotlinJava (#java)
(#kotlin)

```
/*
 * Use a PrivateKey in the KeyStore to create a signature over
 * some data.
 */
val ks: KeyStore = KeyStore.getInstance("AndroidKeyStore").apply {
    load(null)
}
val entry: KeyStore.Entry = ks.getEntry(alias, null)
if (entry !is KeyStore.PrivateKeyEntry) {
    Log.w(TAG, "Not an instance of a PrivateKeyEntry")
    return null
}
val signature: ByteArray = Signature.getInstance("SHA256withECDSA").run {
```

```
    initSign(entry.privateKey)
    update(data)
    sign()
}
```

Similarly, verify data with the `verify(byte[])` (/reference/java/security/Signature#verify(byte[])) method:

KotlinJava (#java)
(#kotlin)

```
/*
 * Verify a signature previously made by a private key in the
 * KeyStore. This uses the X.509 certificate attached to the
 * private key in the KeyStore to validate a previously
 * generated signature.
 */
val ks = KeyStore.getInstance("AndroidKeyStore").apply {
    load(null)
}
val entry = ks.getEntry(alias, null) as? KeyStore.PrivateKeyEntry
if (entry == null) {
    Log.w(TAG, "Not an instance of a PrivateKeyEntry")
    return false
}
val valid: Boolean = Signature.getInstance("SHA256withECDSA").run {
    initVerify(entry.certificate)
    update(data)
    verify(signature)
}
```

Require user authentication for key use

When generating or importing a key into the `AndroidKeyStore`, you can specify that the key is only authorized to be used if the user has been authenticated. The user is authenticated using a subset of their secure lock screen credentials (pattern/PIN/password, biometric credentials).

This is an advanced security feature that is generally useful only if your requirements are that a compromise of your application process after key generation/import (but not before or during) can't bypass the requirement for the user to be authenticated to use the key.

When a key is only authorized to be used if the user has been authenticated, you can call `setUserAuthenticationParameters()` (/reference/android/security/keystore/KeyGenParameterSpec.Builder#setUserAuthenticationParameters(int,%20int)) to configure it to operate in one of the following modes:

Authorize for a duration of time

> All keys are authorized for use as soon as the user authenticates using one of the credentials specified.

Authorize for the duration of a specific cryptographic operation

> Each operation involving a specific key must be individually authorized by the user.
>
> Your app starts this process by calling `authenticate()` (https://developer.android.com/reference/android/hardware/biometrics/BiometricPrompt#authenticate(android.hardware.biometrics.BiometricPrompt.CryptoObject,%20android.os.CancellationSignal,%20java.util.concurrent.Executor,%20android.hardware.biometrics.BiometricPrompt.AuthenticationCallback)) on an instance of `BiometricPrompt`.

For each key that you create, you can choose to support a strong biometric credential (/reference/android/security/keystore/KeyProperties#AUTH_BIOMETRIC_STRONG), a lock screen credential (/reference/android/security/keystore/KeyProperties#AUTH_DEVICE_CREDENTIAL), or both types of credentials. To determine whether the user has set up the credentials that your app's key relies on, call `canAuthenticate()` (/reference/android/hardware/biometrics/BiometricManager#canAuthenticate(int)).

If a key only supports biometric credentials, the key is invalidated by default whenever new biometric enrollments are added. You can configure the key to remain valid when new biometric enrollments are added. To do so, pass `false` into `setInvalidatedByBiometricEnrollment()` (/reference/android/security/keystore/KeyGenParameterSpec.Builder#setInvalidatedByBiometricEnrollment(boolean))
.

Learn more about how to add biometric authentication capabilities into your app, including how to show a biometric authentication dialog (/training/sign-in/biometric-auth).

Supported algorithms

- `Cipher` (/reference/javax/crypto/Cipher)
- `KeyGenerator` (/reference/javax/crypto/KeyGenerator)
- `KeyFactory` (/reference/java/security/KeyFactory)
- `KeyStore` (supports the same key types as `KeyGenerator` and `KeyPairGenerator`)
- `KeyPairGenerator` (/reference/java/security/KeyPairGenerator)
- `Mac` (/reference/javax/crypto/Mac)
- `Signature` (/reference/java/security/Signature)
- `SecretKeyFactory` (/reference/javax/crypto/SecretKeyFactory)

Blog articles

See the blog entry Unifying Key Store Access in ICS (http://android-developers.blogspot.com/2012/03/unifying-key-store-access-in-ics.html).

Last updated 2024-01-03 UTC.