Cryptography

This document describes the proper way to use Android's cryptographic facilities and includes some examples of its use. If your app requires greater key security, use the [Android Keystore system](https://developer.android.com/training/articles/keystore.html).

**Note:** Except where specified, this advice applies to all Android versions.

Specify a provider only with the Android Keystore system

If you're using the [Android Keystore system](https://developer.android.com/training/articles/keystore.html), you **must** specify a provider.

In other situations, however, Android doesn't guarantee a particular provider for a given algorithm. Specifying a provider without using the Android Keystore system could cause compatibility problems in future releases.

Choose a recommended algorithm

When you have the freedom to choose which algorithm to use (such as when you do not require compatibility with a third-party system), we recommend using the following algorithms:

| Class | Recommendation |
| --- | --- |
| Cipher | AES in either CBC or GCM mode with 256-bit keys (such as AES/GCM/NoPadding) |
| MessageDigest | SHA-2 family (eg, SHA-256) |
| Mac | SHA-2 family HMAC (eg, HMACSHA256) |
| Signature | SHA-2 family with ECDSA (eg, SHA256withECDSA) |

Perform common cryptographic operations

The following sections include snippets that demonstrates how you can complete common cryptographic operations in your app.

Encrypt a message

byte[] plaintext = ...;  
KeyGenerator keygen = KeyGenerator.getInstance("AES");  
keygen.init(256);  
SecretKey key = keygen.generateKey();  
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");  
cipher.init(Cipher.ENCRYPT\_MODE, key);  
byte[] ciphertext = cipher.doFinal(plaintext);  
byte[] iv = cipher.getIV();

Generate a message digest

byte[] message = ...;  
MessageDigest md = MessageDigest.getInstance("SHA-256");  
byte[] digest = md.digest(message);

Generate a digital signature

You need to have a PrivateKey object containing the signing key, which you can generate at runtime, read from a file bundled with your app, or obtain from some other source depending on your needs.

byte[] message = ...;  
PrivateKey key = ...;  
Signature s = Signature.getInstance("SHA256withECDSA");  
s.initSign(key);  
s.update(message);  
byte[] signature = s.sign();

Verify a digital signature

You need to have a PublicKey object containing the signer's public key, which you might read from a file bundled with your app, [extract from a certificate](https://developer.android.com/reference/javax/security/cert/Certificate.html#getPublicKey()), or obtain from some other source depending on your needs.

byte[] message = ...;  
byte[] signature = ...;  
PublicKey key = ...;  
Signature s = Signature.getInstance("SHA256withECDSA");  
s.initVerify(key);  
s.update(message);  
boolean valid = s.verify(signature);

Implementation complexities

There are some details of the Android cryptography implementation that seem unusual but are present due to compatibility concerns. This section discusses the ones that you'll most likely encounter.

OAEP MGF1 message digest

RSA OAEP ciphers are parameterized by two different message digests: the “main” digest and the MGF1 digest. There are Cipher identifiers that include digest names, such as Cipher.getInstance("RSA/ECB/OAEPwithSHA-256andMGF1Padding"), which specify the main digest and leave the MGF1 digest unspecified. For Android Keystore, SHA-1 is used for the MGF1 digest, whereas for other Android cryptographic providers, the two digests are the same.

To have more control over the digests that your app uses, you should request a cipher with OAEPPadding, as in Cipher.getInstance("RSA/ECB/OAEPPadding"), and provide an OAEPParameterSpec to init() to explicitly choose both digests.

Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPPadding");  
// To use SHA-256 the main digest and SHA-1 as the MGF1 digest  
cipher.init(Cipher.ENCRYPT\_MODE, new OAEPParameterSpec("SHA-256", "MGF1", MGF1ParameterSpec.SHA1, PSource.PSpecified.DEFAULT));  
// To use SHA-256 for both digests  
cipher.init(Cipher.ENCRYPT\_MODE, new OAEPParameterSpec("SHA-256", "MGF1", MGF1ParameterSpec.SHA256, PSource.PSpecified.DEFAULT));

Deprecated functionality

The following sections describe deprecated functionality that you should no longer use in your app.

Bouncy Castle algorithms

The [Bouncy Castle](https://www.bouncycastle.org/) implementations of many algorithms [are deprecated](https://android-developers.googleblog.com/2018/03/cryptography-changes-in-android-p.html). This only affects cases where you explicitly request the Bouncy Castle provider, as shown in the following example:

Cipher.getInstance("AES/CBC/PKCS7PADDING", "BC");  
// OR  
Cipher.getInstance("AES/CBC/PKCS7PADDING", Security.getProvider("BC"));

As noted above, requesting a specific provider is discouraged, so if you follow that guideline, this deprecation should not affect you.

Password-based encryption ciphers without an IV

Password-based encryption (PBE) ciphers that require an initialization vector (IV) can obtain it from the key, if it's suitably constructed, or from an explicitly-passed IV. When passing a PBE key that doesn't contain an IV and no explicit IV, the PBE ciphers on Android currently assume an IV of zero.

When using PBE ciphers, always pass an explicit IV, as shown in the following code snippet:

SecretKey key = ...;  
Cipher cipher = Cipher.getInstance("PBEWITHSHA256AND256BITAES-CBC-BC");  
byte[] iv = new byte[16];  
new SecureRandom().nextBytes(iv);  
cipher.init(Cipher.ENCRYPT\_MODE, key, new IvParameterSpec(iv));

Crypto provider

As of Android 9 (API level 28), the Crypto Java Cryptography Architecture (JCA) provider has been removed. If your app requests an instance of the Crypto provider, such as by calling the following method, a[NoSuchProviderException](https://developer.android.com/reference/java/security/NoSuchProviderException) occurs.

SecureRandom.getInstance("SHA1PRNG", "Crypto");