Security guidelines

Android has built-in security features that significantly reduce the frequency and impact of application security issues. The system is designed so that you can typically build your apps with the default system and file permissions and avoid difficult decisions about security.

The following core security features help you build secure apps:

- The Android application sandbox, which isolates your app data and code execution from other apps.
- An application framework with robust implementations of common security functionality such as cryptography, permissions, and secure interprocess communication (IPC).
- Technologies like address space layout randomization (ASLR) (https://en.wikipedia.org/wiki/Address_space_layout_randomization), no-execute (NX) (https://en.wikipedia.org/wiki/NX_bit), ProPolice, safe_iop (https://code.google.com/archive/p/safe-iop/wikis/README.wiki), OpenBSD (https://www.openbsd.org/) `dlmalloc` and `calloc`, and Linux `mmap_min_addr` to mitigate risks associated with common memory management errors.
- User-granted permissions to restrict access to system features and user data.
- Application-defined permissions to control application data on a per-app basis.

It's important to be familiar with the Android security best practices on this page. Following these practices as general coding habits help you avoid inadvertently introducing security issues that adversely affect your users.

Authentication

Authentication is a prerequisite for many key security operations. To control access to protected assets like user data, app functionality, and other resources, you'll need to add authentication to your Android app.

You can improve your user's authentication experience by integrating your app with Credential Manager (/training/sign-in/passkeys). Credential Manager is an Android Jetpack

library that unifies API support for most major authentication methods, including passkeys, passwords, and federated sign-in solutions such as Sign-in with Google (/training/sign-in/credential-manager).

To further enhance security for your app, consider adding biometric authentication (/training/sign-in/biometric-auth) methods such as fingerprint scans or facial recognition. Good candidates for adding biometric authentication might include apps for financial, healthcare, or identity management.

Android's autofill framework (/guide/topics/text/autofill) can ease the sign-up and sign-in process, reducing error rates and user friction. Autofill integrates with password managers, allowing users to select complex, randomized passwords that can be stored and retrieved easily and securely.

App integrity

The Play Integrity API (/google/play/integrity) helps you check that interactions and server requests are coming from your genuine app binary running on a genuine Android-powered device. By detecting potentially risky and fraudulent interactions, such as from tampered app versions and untrustworthy environments, your app's backend server can respond with appropriate actions to prevent attacks and reduce abuse.

Data storage

The most common security concern for an application on Android is whether the data that you save on the device is accessible to other apps. There are three fundamental ways to save data on the device:

- Internal storage
- External storage
- Content providers

The following sections describe the security issues associated with each approach.

Internal storage

By default, files that you create on internal storage (/guide/topics/data/data-storage#filesInternal) are accessible only to your app. Android implements this protection, and it's sufficient for most applications.

Avoid the deprecated `MODE_WORLD_WRITEABLE` (/reference/android/content/Context#MODE_WORLD_WRITEABLE) and `MODE_WORLD_READABLE` (/reference/android/content/Context#MODE_WORLD_READABLE) modes for IPC files. They don't provide the ability to limit data access to particular applications, and they don't provide any control of data format. If you want to share your data with other app processes, consider using a content provider (/guide/topics/providers/content-providers) instead, which offers read and write permissions to other apps and can make dynamic permission grants on a case-by-case basis.

External storage

Files created on external storage (/guide/topics/data/data-storage#filesExternal), such as SD cards, are globally readable and writable. Because external storage can be removed by the user and also modified by any application, only store non-sensitive information using external storage.

Perform input validation (#input-validation) when handling data from external storage as you would with data from any untrusted source. Don't store executables or class files on external storage prior to dynamic loading. If your app does retrieve executable files from external storage, make sure the files are signed and cryptographically verified prior to dynamic loading.

Content providers

Content providers (/guide/topics/providers/content-providers) offer a structured storage mechanism that can be limited to your own application or exported to allow access by other applications. If you don't intend to provide other applications with access to your `ContentProvider` (/reference/android/content/ContentProvider), mark it as `android:exported=false` (/guide/topics/manifest/provider-element#exported) in the application manifest. Otherwise, set the `android:exported` attribute to `true` to let other apps access the stored data.

When creating a `ContentProvider` that is exported for use by other applications, you can specify a single permission (/guide/topics/manifest/provider-element#prmsn) for reading and

writing, or you can specify distinct permissions for reading and writing. Limit your permissions to those required to accomplish the task at hand. Keep in mind that it's usually easier to add permissions later to expose new functionality than it is to take them away and impact existing users.

If you are using a content provider for sharing data between only your own apps, we recommend using the `android:protectionLevel` (/guide/topics/manifest/permission-element#plevel) attribute set to `signature` protection. Signature permissions (/guide/topics/permissions/overview#signature) don't require user confirmation, so they provide a better user experience and more controlled access to the content provider data when the apps accessing the data are signed (/tools/publishing/app-signing) with the same key.

Content providers can also provide more granular access by declaring the `android:grantUriPermissions` (/guide/topics/manifest/provider-element#gprmsn) attribute and using the `FLAG_GRANT_READ_URI_PERMISSION` (/reference/android/content/Intent#FLAG_GRANT_READ_URI_PERMISSION) and `FLAG_GRANT_WRITE_URI_PERMISSION` (/reference/android/content/Intent#FLAG_GRANT_WRITE_URI_PERMISSION) flags in the `Intent` (/reference/android/content/Intent) object that activates the component. The scope of these permissions can be further limited by the `<grant-uri-permission>` (/guide/topics/manifest/grant-uri-permission-element) element.

When accessing a content provider, use parameterized query methods such as `query` (/reference/android/content/ContentProvider#query(android.net.Uri,%20java.lang.String%5B%5D,%20java.lang.String,%20java.lang.String%5B%5D,%20java.lang.String))() , `update` (/reference/android/content/ContentProvider#update(android.net.Uri,%20android.content.ContentValues,%20java.lang.String,%20java.lang.String%5B%5D))() , and `delete()` (/reference/android/content/ContentProvider#delete(android.net.Uri,%20java.lang.String,%20java.lang.String%5B%5D)) to avoid potential SQL injection from untrusted sources. Note that using parameterized methods is not sufficient if the `selection` argument is built by concatenating user data prior to submitting it to the method.

Don't have a false sense of security about the write permission. The write permission allows SQL statements that make it possible for some data to be confirmed using creative `WHERE` clauses and parsing the results. For example, an attacker might probe for the presence of a specific phone number in a call log by modifying a row only if that phone number already

exists. If the content provider data has predictable structure, the write permission might be equivalent to providing both reading and writing.

Permissions

Because Android sandboxes applications from each other, applications must explicitly share resources and data. They do this by declaring the permissions they need for additional capabilities not provided by the basic sandbox, including access to device features such as the camera.

Permission requests

Minimize the number of permissions that your app requests. Restricting access to sensitive permissions reduces the risk of inadvertently misusing those permissions, improves user adoption, and makes your app less vulnerable for attackers. Generally, if a permission isn't required for your app to function, don't request it. See the guide to evaluating whether your app needs to declare permissions (/training/permissions/evaluating).

If possible, design your application in a way that doesn't require any permissions. For example, rather than requesting access to device information to create a unique identifier, create a UUID (/reference/java/util/UUID) for your application. (Learn more in the section about user data (#user-data)). Or, rather than using external storage (which requires permission), store data on internal storage.

In addition to requesting permissions, your application can use the `<permission>` (/guide/topics/manifest/permission-element) element to protect IPC that is security sensitive and is exposed to other applications, such as a `ContentProvider` (/reference/android/content/ContentProvider). In general, we recommend using access controls other than user-confirmed permissions where possible, because permissions can be confusing for users. For example, consider using the signature protection level (/guide/topics/manifest/permission-element#plevel) on permissions for IPC communication between applications provided by a single developer.

Don't leak permission-protected data. This occurs when your app exposes data over IPC that is available only because your app has permission to access that data. The clients of your app's IPC interface might not have that same data-access permission. More details on the frequency and potential effects of this issue appear in the research paper Permission

Re-Delegation: Attacks and Defenses (https://www.usenix.org/legacy/event/sec11/tech/full_papers/Felt.pdf) , published at USENIX.

Permission definitions

Define the smallest set of permissions that satisfy your security requirements. Creating a new permission is relatively uncommon for most applications, because the system-defined permissions (/reference/android/Manifest.permission) cover many situations. Where appropriate, perform access checks using existing permissions.

If you need a new permission, consider whether you can accomplish your task with a signature protection level (/guide/topics/manifest/permission-element#plevel). Signature permissions are transparent to the user and allow access only by applications signed by the same developer as the application performing the permission check.

If creating a new permission is still required, declare it in the app manifest using the `<permission>` (/guide/topics/manifest/permission-element) element. Apps using the new permission can reference it by adding a `<uses-permission>` (/guide/topics/manifest/uses-permission-element) element in their manifest files. You can also add permissions dynamically by using the `addPermission()` (/reference/android/content/pm/PackageManager#addPermission(android.content.pm.PermissionInfo)) method.

If you create a permission with the dangerous protection level, there are a number of complexities that you need to consider:

- The permission must have a string that concisely expresses to the user the security decision they are required to make.
- The permission string must be localized to many different languages.
- Users might choose not to install an application because a permission is confusing or perceived as risky.
- Applications might request the permission when the creator of the permission hasn't been installed.

Each of these poses a significant nontechnical challenge for you as the developer while also confusing your users, which is why we discourage the use of the dangerous permission level.

Networking

Network transactions are inherently risky for security, because they involve transmitting data that is potentially private to the user. People are increasingly aware of the privacy concerns of a mobile device, especially when the device performs network transactions, so it's very important that your app implement all best practices toward keeping the user's data secure at all times.

IP networking

Networking on Android is not significantly different from other Linux environments. The key consideration is making sure that appropriate protocols are used for sensitive data, such as `HttpsURLConnection` (/reference/javax/net/ssl/HttpsURLConnection) for secure web traffic. Use HTTPS over HTTP anywhere that HTTPS is supported on the server, because mobile devices frequently connect on networks that aren't secured, such as public Wi-Fi hotspots.

Authenticated, encrypted socket-level communication can be easily implemented using the `SSLSocket` (/reference/javax/net/ssl/SSLSocket) class. Given the frequency with which Android devices connect to unsecured wireless networks using Wi-Fi, the use of secure networking is strongly encouraged for all applications that communicate over the network.

Some applications use localhost (https://en.wikipedia.org/wiki/Localhost) network ports for handling sensitive IPC. Don't use this approach, because these interfaces are accessible by other applications on the device. Instead, use an Android IPC mechanism where authentication is possible, such as with a `Service` (/reference/android/app/Service). Binding to the non-specific IP address `INADDR_ANY` is worse than using loopback, because it allows your application to receive requests from any IP address.

Make sure that you don't trust data downloaded from HTTP or other insecure protocols. This includes validation of input in `WebView` (/reference/android/webkit/WebView) and any responses to intents issued against HTTP.

Telephony networking

The Short Message Service (SMS) protocol was primarily designed for user-to-user communication and isn't well suited for apps that want to transfer data. Due to the limitations of SMS, we recommend using Firebase Cloud Messaging

(https://firebase.google.com/docs/cloud-messaging/) (FCM) and IP networking for sending data messages from a web server to your app on a user device.

Be aware that SMS is neither encrypted nor strongly authenticated on either the network or the device. In particular, any SMS receiver should expect that a malicious user might have sent the SMS to your application. Don't rely on unauthenticated SMS data to perform sensitive commands. Also, be aware that SMS can be subject to spoofing and/or interception on the network. On the Android-powered device itself, SMS messages are transmitted as broadcast intents, so they can be read or captured by other applications that have the `READ_SMS` (/reference/android/Manifest.permission#READ_SMS) permission.

Input validation

Insufficient input validation is one of the most common security problems affecting applications, regardless of what platform they run on. Android has platform-level countermeasures that reduce the exposure of applications to input validation issues, and we recommend that you use those features where possible. Also, we recommend using type-safe languages to reduce the likelihood of input validation issues.

If you are using native code, any data read from files, received over the network, or received from an IPC has the potential to introduce a security issue. The most common problems are buffer overflows (https://en.wikipedia.org/wiki/Buffer_overflow), use after free (https://en.wikipedia.org/wiki/Double_free#Use_after_free), and off-by-one errors (https://en.wikipedia.org/wiki/Off-by-one_error). Android provides a number of technologies, like ASLR and Data Execution Prevention (DEP), that reduce the exploitability of these errors, but they don't solve the underlying problem. You can prevent these vulnerabilities by carefully handling pointers and managing buffers.

Dynamic, string-based languages such as JavaScript and SQL are also subject to input validation problems due to escape characters and script injection (https://en.wikipedia.org/wiki/Code_injection).

If you are using data within queries that are submitted to an SQL database or a content provider, SQL injection can be an issue. The best defense is to use parameterized queries, as discussed in the section about content providers (#content-providers). Limiting permissions to read-only or write-only can also reduce the potential for harm related to SQL injection.

If you can't use the security features discussed in this section, make sure to use well-structured data formats and verify that the data conforms to the expected format. While blocking specific characters or performing character replacement can be an effective strategy, these techniques are error prone in practice, and we recommend avoiding them when possible.

User data

The best approach for user data security is to minimize the use of APIs that access sensitive or personal information. If you have access to user data, avoid storing or transmitting it if you can. Consider whether your application logic can be implemented using a hash or non-reversible form of the data. For example, your app might use the hash of an email address as a primary key to avoid transmitting or storing the email address. This reduces the chances of inadvertently exposing data, and it also reduces the chance of attackers attempting to exploit your app.

Authenticate your user whenever access to private data is required, and use modern authentication methods such as passkeys (https://developers.google.com/identity/passkeys) and Credential Manager (/training/sign-in/passkeys). If your app needs to access personal information, keep in mind that some jurisdictions might require you to provide a privacy policy explaining your use and storage of that data. Follow the security best practice of minimizing access to user data to simplify compliance.

Also, consider whether your application could inadvertently expose personal information to other parties, such as third-party components for advertising or third-party services used by your application. If you don't know why a component or service requires personal information, don't provide it. In general, reducing the access to personal information by your application reduces the potential for problems in this area.

If your app requires access to sensitive data, evaluate whether you need to transmit it to a server or if you can run the operation on the client. Consider running any code using sensitive data on the client to avoid transmitting user data. Also, make sure that you don't inadvertently expose user data to other applications on the device through overly permissive IPC, world-writable files, or network sockets. Overly permissive IPC is a special case of leaking permission-protected data, discussed in the Permission requests (#requesting-permissions) section.

If a Globally Unique Identifier (GUID) is required, create a large, unique number and store it. Don't use phone identifiers such as the phone number or IMEI, which might be associated with personal information. This topic is discussed in more detail in the page about best practices for unique identifiers (/training/articles/user-data-ids).

Be careful when writing to on-device logs. On Android, logs are a shared resource and are available to an application with the `READ_LOGS` (/reference/android/Manifest.permission#READ_LOGS) permission. Even though the phone log data is temporary and erased on reboot, inappropriate logging of user information could inadvertently leak user data to other applications. In addition ton't logging PII, limit log usage in production apps. To easily implement this, use debug flags and custom `Log` classes with easily configurable logging levels.

WebView

Because `WebView` (/reference/android/webkit/WebView) consumes web content that can include HTML and JavaScript, improper use can introduce common web security issues such as cross-site scripting (https://en.wikipedia.org/wiki/Cross_site_scripting) (JavaScript injection). Android includes a number of mechanisms to reduce the scope of these potential issues by limiting the capability of `WebView` to the minimum functionality required by your application.

If your application doesn't directly use JavaScript within a `WebView`, *don't* call `setJavaScriptEnabled` (/reference/android/webkit/WebSettings#setJavaScriptEnabled(boolean))(). Some sample code uses this method; if you repurpose sample code that uses it in a production application, remove that method call if it's not required. By default, `WebView` doesn't execute JavaScript, so cross-site scripting is not possible.

Use `addJavaScriptInterface()` (/reference/android/webkit/WebView#addJavascriptInterface(java.lang.Object,%20java.lang.String)) with particular care, because it lets JavaScript invoke operations that are normally reserved for Android applications. If you use it, expose `addJavaScriptInterface()` only to web pages from which all input is trustworthy. If untrusted input is allowed, untrusted JavaScript might be able to invoke Android methods within your app. In general, we recommend exposing `addJavaScriptInterface()` only to JavaScript that is contained within your application APK.

If your application accesses sensitive data with a `WebView`, consider using the `clearCache()` (/reference/android/webkit/WebView#clearCache(boolean)) method to delete any

files stored locally. You can also use server-side headers, such as `no-store`, to indicate that an application should not cache particular content.

Devices running platforms older than Android 4.4 (API level 19) use a version of `webkit` (/reference/android/webkit/package-summary) that has a number of security issues. As a workaround, if your app is running on these devices, it must confirm that `WebView` objects display only trusted content. To make sure your app isn't exposed to potential vulnerabilities in SSL, use the updatable security `Provider` (/reference/java/security/Provider) object as described in Update your security provider to protect against SSL exploits (/training/articles/security-gms-provider). If your application must render content from the open web, consider providing your own renderer so you can keep it up to date with the latest security patches.

Credential requests

Credential requests are a vector for attack. Here are some tips to help you make credential requests in your Android apps more secure.

Minimize credential exposure

- **Avoid unnecessary credential requests**. To make phishing attacks more conspicuous and less likely to be successful, minimize the frequency of asking for user credentials. Instead, use an authorization token and refresh it. Request only the minimum amount of credential information necessary for authentication and authorization.
- **Store credentials securely**. Use Credential Manager (/training/sign-in/passkeys) to enable passwordless authentication using passkeys or to implement federated sign-in using schemes such as Sign in with Google. If you must use traditional password authentication, don't store user IDs and passwords on the device. Instead, perform initial authentication using the username and password supplied by the user, and then use a short-lived, service-specific authorization token.
- **Limit the scope of permissions**. Don't request broad permissions for a task that only requires a more narrow scope.
- **Limit access tokens**. Use short-lived tokens operations and API calls.

- **Limit authentication rates**. Rapid, successive authentication or authorization requests can be a sign of a brute-force attack. Limit these rates to a reasonable frequency while still allowing for a functional and user-friendly app experience.

Use secure authentication

- **Implement passkeys**. Enable passkeys as a more secure and user-friendly upgrade to passwords.
- **Add biometrics**. Offer the ability to use biometric authentication (/training/sign-in/biometric-auth) such as fingerprint or facial recognition for added security.
- **Use federated identity providers**. Credential Manager supports federated authentication providers such as Sign in with Google (/training/sign-in/credential-manager).
- **Encrypt communication** Use HTTPS and similar technologies to ensure the data your app sends over a network is protected.

Practice secure account management

- Connect to services that are accessible to multiple applications using `AccountManager` (/reference/android/accounts/AccountManager). Use the `AccountManager` class to invoke a cloud-based service, and don't store passwords on the device.
- After using `AccountManager` to retrieve an `Account` (/reference/android/accounts/Account), use `CREATOR` (/reference/android/accounts/Account#CREATOR) before passing in any credentials so that you don't inadvertently pass credentials to the wrong application.
- If credentials are used only by applications that you create, you can verify the application that accesses the `AccountManager` using `checkSignatures` (/reference/android/content/pm/PackageManager#checkSignatures(int,%20int))(). Alternatively, if only one application uses the credential, you might use a `KeyStore` (/reference/java/security/KeyStore) for storage.

Stay vigilant

- **Keep your code up-to-date**. Be sure to update your source code, including any third-party libraries and dependencies, to guard against the latest vulnerabilities.

- **Monitor suspicious activity**. Look for potential misuse, such as patterns of authorization misuse.
- **Audit your code**. Perform regular security checks against your codebase to look for potential credential request issues.

API key management

API keys are a critical component of many Android apps, enabling them to access external services and perform essential functions such as connecting to mapping services, authentication, and weather services. However, exposing these sensitive keys can have severe consequences, including data breaches, unauthorized access, and financial losses. To prevent such scenarios, developers should implement secure strategies for handling API keys throughout the development process.

To protect services from misuse, API keys must be carefully protected. To secure a connection between the app and a service that uses an API key, you need to secure the access to the API. When your app is compiled, and your app's source code includes API keys, it's possible for an attacker to decompile the app and find these resources.

This section is intended for two groups of Android developers: those who work with infrastructure teams on their continuous delivery pipeline, and those who deploy standalone apps in the Play store. This section outlines best practices for how to handle API keys, so your app can communicate with services securely.

Generation and storage

Developers should treat API key storage as a critical component of data protection and user privacy using a defense-in-depth approach.

Strong key storage

For optimal key management security, use the Android Keystore (/privacy-and-security/keystore) , and encrypt stored keys using a robust tool such as the security-crypto (/jetpack/androidx/releases/security#security-crypto-1.0.) Jetpack library or Tink Java (https://github.com/tink-crypto/tink-java).

The following example uses the Jetpack security-crypto library to create encrypted shared preferences (/reference/kotlin/androidx/security/crypto/EncryptedSharedPreferences).

Kotlin Java (#java)
(#kotlin)

```
val masterKey = MasterKey.Builder(context)
  .setKeyScheme(MasterKey.KeyScheme.AES256_GCM)
  .build()

val encryptedSharedPreferences = EncryptedSharedPreferences.create(
  context,
  "secret_shared_prefs",
  masterKey,
  EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
  EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
)

// use the shared preferences and editor as you normally would
encryptedSharedPreferences.edit()
```

Source control exclusion

Never commit API keys to your source code repository. Adding API keys to source code risks exposure of keys to public repositories, shared code examples, and accidentally-shared files. Instead, use Gradle plugins such as the secrets-gradle-plugin (https://github.com/google/secrets-gradle-plugin) to work with API keys in your project.

Environment-specific keys

If possible, use separate API keys development, testing, and production environments. Use environment-specific keys to isolate each environment, reducing the risk of exposing production data and allowing you to disable compromised keys without affecting your production environment.

Usage and access control

Secure API key practices are essential for protecting your API and your users. Here's how to prepare your keys for optimal security:

- **Generate unique keys for each app**: Use separate API keys for each app to help identify and isolate compromised access.
- **Implement IP restrictions**: If possible, limit API key usage to specific IP addresses or ranges.
- **Limit mobile app key usage**: Limit API key usage to specific mobile apps by bundling them with the key or by using app certificates.
- **Log and monitor for suspicious activity**: Implement API usage logging and monitoring mechanisms to detect suspicious activity and prevent potential abuse.

Note: Your service should provide features for restricting keys to a particular package or platform. For example, the Google Maps API (https://developers.google.com/maps/api-security-best-practices#restricting-api-keys) limits key access by package name and signing key.

OAuth 2.0 provides a framework for authorizing access to resources. It defines standards for how clients and servers should interact, and it allows for secure authorization. You can use OAuth 2.0 to restrict API key usage to specific clients, and define the access scope so that each API key only has the minimum level of access required for their intended purpose.

Key rotation and expiration

To reduce the risk of unauthorized access through undiscovered API vulnerabilities, it is important to rotate API keys regularly. The ISO 27001 (https://www.iso.org/standard/27001) standard defines a compliance framework for how often to perform key rotation. For most cases, a key rotation period between 90 days to 6 months should be adequate. Implementing a robust key management system can help you streamline these processes, improving the efficiency of your key rotation and expiration needs.

General best practices

- **Use SSL/HTTPS**: Always use HTTPS communication to encrypt your API requests.
- **Certificate pinning**: For an extra layer of security, you can consider implementing certificate pinning (/privacy-and-security/security-config#CertificatePinning) to check which certificates are considered valid.

- **Validate and sanitize user input**: Validate and sanitize user input to prevent injection attacks that could expose API keys.
- **Follow security best practices**: Implement general security best practices in your development process, including secure coding techniques, code reviews, and vulnerability scanning.
- **Stay informed**: Stay updated on the latest security threats and best practices for API key management.
- **SDKs up-to-date**: Make sure your SDKs and libraries are updated to the latest version.

Cryptography

In addition to providing data isolation, supporting full-filesystem encryption, and providing secure communications channels, Android provides a wide array of algorithms for protecting data using cryptography.

Know which Java Cryptography Architecture (JCA) security providers your software uses. Try to use the highest level of the pre-existing framework implementation that can support your use case. If applicable, use the Google-provided providers in the Google-specified order.

If you need to securely retrieve a file from a known network location, a simple HTTPS URI might be adequate and requires no knowledge of cryptography. If you need a secure tunnel, consider using `HttpsURLConnection` (/reference/javax/net/ssl/HttpsURLConnection) or `SSLSocket` (/reference/javax/net/ssl/SSLSocket) rather than writing your own protocol. If you use `SSLSocket`, be aware that it doesn't perform hostname verification. See Warnings about using `SSLSocket` directly (/training/articles/security-ssl#WarningsSslSocket).

If you find that you need to implement your own protocol, don't implement your own cryptographic algorithms. Use existing cryptographic algorithms, such as the implementations of AES and RSA provided in the `Cipher` (/reference/javax/crypto/Cipher) class. Additionally, follow these best practices:

- Use 256-bit AES for commercial purposes. (If unavailable, use 128-bit AES.)
- Use either 224- or 256-bit public key sizes for elliptic curve (EC) cryptography.
- Know when to use CBC, CTR, or GCM block modes.

- Avoid IV/counter reuse in CTR mode. Ensure that they're cryptographically random.
- When using encryption, implement integrity using the CBC or CTR mode with one of the following functions:
    - HMAC-SHA1
    - HMAC-SHA-256
    - HMAC-SHA-512
    - GCM mode

Use a secure random number generator, `SecureRandom` (/reference/java/security/SecureRandom), to initialize any cryptographic keys generated by `KeyGenerator` (/reference/javax/crypto/KeyGenerator). Use of a key that is not generated with a secure random number generator significantly weakens the strength of the algorithm and may allow offline attacks.

If you need to store a key for repeated use, use a mechanism, such as `KeyStore` (/reference/java/security/KeyStore), that provides long term storage and retrieval of cryptographic keys.

Interprocess communication

Some apps attempt to implement IPC using traditional Linux techniques such as network sockets and shared files. However, we recommend instead that you use Android system functionality for IPC such as `Intent` (/reference/android/content/Intent), `Binder` (/reference/android/os/Binder) or `Messenger` (/reference/android/os/Messenger) with a `Service` (/reference/android/app/Service), and `BroadcastReceiver` (/reference/android/content/BroadcastReceiver). The Android IPC mechanisms let you verify the identity of the application connecting to your IPC and set security policy for each IPC mechanism.

Many of the security elements are shared across IPC mechanisms. If your IPC mechanism isn't intended for use by other applications, set the `android:exported` attribute to `false` in the component's manifest element, such as for the `<service>` (/guide/topics/manifest/service-element#exported) element. This is useful for applications that consist of multiple processes within the same UID or if you decide late in development that

you don't actually want to expose functionality as IPC, but you don't want to rewrite the code.

If your IPC is accessible to other applications, you can apply a security policy by using the `<permission>` (/guide/topics/manifest/permission-element) element. If the IPC is between apps that are your own and are signed with the same key, use a `signature-level` permission in the `android:protectionLevel` (/guide/topics/manifest/permission-element#plevel).

Intents

For activities and broadcast receivers, intents are the preferred mechanism for asynchronous IPC on Android. Depending on your application requirements, you might use `sendBroadcast` (/reference/android/content/Context#sendBroadcast(android.content.Intent))(), `sendOrderedBroadcast` (/reference/android/content/Context#sendOrderedBroadcast(android.content.Intent,%20java.lang.String))() , or an explicit intent to a specific application component. For security purposes, explicit intents are preferred.

Caution: If you use an intent to bind to a `**Service**` (/reference/android/app/Service), use an explicit (/guide/components/intents-filters#Types) intent to keep your app secure. Using an implicit intent to start a service is a security hazard, because you can't be certain what service will respond to the intent and the user can't see which service starts. Beginning with Android 5.0 (API level 21), the system throws an exception if you call `**bindService()**` (/reference/android/content/Context#bindService(android.content.Intent,%20android.content.ServiceConnection,%20int)) with an implicit intent.

Note that ordered broadcasts can be *consumed* by a recipient, so they might not be delivered to all applications. If you are sending an intent that must be delivered to a specific receiver, you must use an explicit intent that declares the receiver by name.

Senders of an intent can verify that the recipient has permission by specifying a non-null permission with the method call. Only applications with that permission receive the intent. If data within a broadcast intent might be sensitive, consider applying a permission to make sure that malicious applications can't register to receive those messages without appropriate permissions. In those circumstances, you might also consider invoking the receiver directly, rather than raising a broadcast.

Note: Intent filters aren't security features. Components can be invoked with explicit intents and might not have data that would conform to the intent filter. To confirm that it is properly formatted for the invoked receiver, service, or activity, perform input validation within your intent receiver.

Services

A `Service` (/reference/android/app/Service) is often used to supply functionality for other applications to use. Each service class must have a corresponding `<service>` (/guide/topics/manifest/service-element) declaration in its manifest file.

By default, services aren't exported and can't be invoked by any other application. However, if you add any intent filters to the service declaration, it is exported by default. It's best if you explicitly declare the `android:exported` (/guide/topics/manifest/service-element#exported) attribute to be sure it behaves the way you intend it to. Services can also be protected using the `android:permission` (/guide/topics/manifest/service-element#prmsn) attribute. By doing so, other applications need to declare a corresponding `<uses-permission>` (/guide/topics/manifest/uses-permission-element) element in their own manifest to be able to start, stop, or bind to the service.

Note: If your app targets Android 5.0 (API level 21) or higher, use the `**JobScheduler**` (/reference/android/app/job/JobScheduler) to execute background services.

A service can protect individual IPC calls that are made into it with permissions. This is done by calling `checkCallingPermission()` (/reference/android/content/Context#checkCallingPermission(java.lang.String)) before executing the implementation of the call. We recommend using the declarative permissions in the manifest, since those are less prone to oversight.

Caution: Don't confuse client and server permissions; ensure that the called app has appropriate permissions and verify that you grant the same permissions to the calling app.

Binder and Messenger interfaces

Using `Binder` (/reference/android/os/Binder) or `Messenger` (/reference/android/os/Messenger) is the preferred mechanism for RPC style IPC on Android. They provide well-defined interfaces that enable mutual authentication of the endpoints, if required.

We recommend that you design your app interfaces in a way that doesn't require interface-specific permission checks. `Binder` and `Messenger` objects aren't declared within the application manifest, and therefore you can't apply declarative permissions directly to them. They generally inherit permissions declared in the application manifest for the `Service` (/reference/android/app/Service) or `Activity` (/reference/android/app/Activity) within which they are implemented. If you are creating an interface that requires authentication and/or access controls, you must explicitly add those controls as code in the `Binder` or `Messenger` interface.

If you are providing an interface that does require access controls, use `checkCallingPermission()` (/reference/android/content/Context#checkCallingPermission(java.lang.String)) to verify whether the caller has a required permission. This is especially important before accessing a service on behalf of the caller, as the identity of your application is passed to other interfaces. If you are invoking an interface provided by a `Service`, the `bindService()` (/reference/android/content/Context#bindService(android.content.Intent,%20android.content.ServiceConnection,%20int)) invocation can fail if you don't have permission to access the given service. If you need to allow an external process to interact with your app but it doesn't have the necessary permissions to do so, you can use the `clearCallingIdentity()` (/reference/android/os/Binder#clearCallingIdentity()) method. This method performs the call to your app's interface as though your app were making the call itself, rather than the external caller. You can restore the caller permissions later with the `restoreCallingIdentity()` (/reference/android/os/Binder#restoreCallingIdentity(long)) method.

For more information about performing IPC with a service, see Bound Services (/guide/components/bound-services).

Broadcast receivers

A `BroadcastReceiver` (/reference/android/content/BroadcastReceiver) handles asynchronous requests initiated by an `Intent` (/reference/android/content/Intent).

By default, receivers are exported and can be invoked by any other application. If your `BroadcastReceiver` is intended for use by other applications, you might want to apply security permissions to receivers using the `<receiver>` (/guide/topics/manifest/receiver-element) element within the application manifest. This prevents applications without appropriate permissions from sending an intent to the `BroadcastReceiver`.

Security with dynamically loaded code

We strongly discourage loading code from outside of your application APK. Doing so significantly increases the likelihood of application compromise due to code injection or code tampering. It also adds complexity around version management and application testing —and it can make it impossible to verify the behavior of an application, so it might be prohibited in some environments.

If your application does dynamically load code, the most important thing to keep in mind is that the dynamically loaded code runs with the same security permissions as the application APK. The user makes a decision to install your application based on your identity, and the user expects that you provide any code run within the application, including code that is dynamically loaded.

Many applications attempt to load code from insecure locations, such as downloaded from the network over unencrypted protocols or from world-writable locations such as external storage. These locations could let someone on the network modify the content in transit or another application on a user's device to modify the content on the device. On the other hand, modules included directly within your APK can't be modified by other applications. This is true whether the code is a native library or a class being loaded using `DexClassLoader` (/reference/dalvik/system/DexClassLoader).

Security in a virtual machine

Dalvik is Android's runtime virtual machine (VM). Dalvik was built specifically for Android, but many of the concerns regarding secure code in other virtual machines also apply to Android. In general, you don't need to concern yourself with security issues relating to the virtual machine. Your application runs in a secure sandbox environment, so other processes on the system can't access your code or private data.

If you're interested in learning more about virtual machine security, familiarize yourself with some existing literature on the subject. Two of the more popular resources are:

- Securing Java (http://www.securingjava.com/toc.html)
- Related 3rd party Projects (https://www.owasp.org/index.php/Category:Java#tab=Related_3rd_Party_Projects)

This document focuses on areas that are Android specific or different from other VM environments. For developers experienced with VM programming in other environments, there are two broad issues that might be different about writing apps for Android:

- Some virtual machines, such as the JVM or .NET runtime, act as a security boundary, isolating code from the underlying operating system capabilities. On Android, the Dalvik VM is not a security boundary—the application sandbox is implemented at the OS level, so Dalvik can interoperate with native code in the same application without any security constraints.
- Given the limited storage on mobile devices, it's common for developers to want to build modular applications and use dynamic class loading. When doing this, consider both the source where you retrieve your application logic and where you store it locally. Don't use dynamic class loading from sources that aren't verified, such as unsecured network sources or external storage, because that code might be modified to include malicious behavior.

Security in native code

In general, we recommend using the Android SDK for application development, rather than using native code with the Android NDK (/tools/sdk/ndk). Applications built with native code are more complex, less portable, and more likely to include common memory-corruption errors such as buffer overflows.

Android is built using the Linux kernel, and being familiar with Linux development security best practices is especially useful if you are using native code. Linux security practices are beyond the scope of this document, but one of the most popular resources is Secure Programming HOWTO - Creating Secure Software (http://www.dwheeler.com/secure-programs).

An important difference between Android and most Linux environments is the application sandbox. On Android, all applications run in the application sandbox, including those written with native code. A good way to think about it for developers familiar with Linux is to know that every application is given a unique User Identifier (UID) with very limited permissions. This is discussed in more detail in the Android Security Overview (https://source.android.com/tech/security/index.html), and you should be familiar with application permissions even if you are using native code.

Last updated 2024-02-26 UTC.