Signing Your Applications

This document provides information about signing your Android applications prior to publishing them for mobile device users.

Quickview

- All Android apps *must* be signed
- You can sign with a self-signed key
- How you sign your apps is critical — read this document carefully
- Determine your signing strategy early in the development process

In this document

Overview
Signing Strategies
Basic Setup for Signing
Signing in Debug Mode
Signing for Public Release
  Obtain a suitable private key
  Compile the application in release mode
  Sign your application with your private key
  Align the final APK package
  Compile and sign with Eclipse ADT
Securing Your Private Key

See also

Versioning Your Applications
Preparing to Publish

Overview

The Android system requires that all installed applications be digitally signed with a certificate whose private key is held by the application's developer. The Android system uses the certificate as a means of identifying the author of an application and establishing trust relationships between applications. The certificate is not used to control which applications the user can install. The certificate does not need to be signed by a certificate authority: it is perfectly allowable, and typical, for Android applications to use self-signed certificates.

The important points to understand about signing Android applications are:

- All applications *must* be signed. The system will not install an application that is not signed.
- You can use self-signed certificates to sign your applications. No certificate authority is needed.
- When you are ready to release your application for end-users, you must sign it with a suitable private key. You can not publish an application that is signed with the debug key generated by the SDK tools.
- The system tests a signer certificate's expiration date only at install time. If an application's signer certificate expires after the application is installed, the application will continue to function normally.
- You can use standard tools — Keytool and Jarsigner — to generate keys and sign your application .apk files.
- Once you have signed the application, use the `zipalign` tool to optimize the final APK package.

The Android system will not install or run an application that is not signed appropriately. This applies wherever the Android system is run, whether on an actual device or on the emulator. For this reason, you must set up signing for your application before you will be able to run or debug it on an emulator or device.

The Android SDK tools assist you in signing your applications when debugging. Both the ADT Plugin for Eclipse and the Ant build tool offer two signing modes — *debug mode* and *release mode*.

- While developing and testing, you can compile in debug mode. In debug mode, the build tools use the Keytool utility, included in the JDK, to create a keystore and key with a known alias and password. At each compilation, the tools then use the debug key to sign the application .apk file. Because the password is known, the tools don't need to prompt you for the keystore/key password each time you compile.
- When your application is ready for release, you must compile in release mode and then sign the .apk with your private key. There are two ways to do this:
  - Using Keytool and Jarsigner in the command-line. In this approach, you first compile your application to an *unsigned* .apk. You must then sign the .apk manually with your private key using Jarsigner (or similar tool). If you do not have a suitable private key already, you can run Keytool manually to generate your own keystore/key and then sign your application with Jarsigner.
  - Using the ADT Export Wizard. If you are developing in Eclipse with the ADT plugin, you can use the Export Wizard to compile the application, generate a private key (if necessary), and sign the .apk, all in a single

process using the Export Wizard.

Once your application is signed, don't forget to run `zipalign` on the APK for additional optimization.

Signing Strategies

Some aspects of application signing may affect how you approach the development of your application, especially if you are planning to release multiple applications.

In general, the recommended strategy for all developers is to sign all of your applications with the same certificate, throughout the expected lifespan of your applications. There are several reasons why you should do so:

- Application upgrade – As you release updates to your application, you will want to continue to sign the updates with the same certificate or set of certificates, if you want users to upgrade seamlessly to the new version. When the system is installing an update to an application, it compares the certificate(s) in the new version with those in the existing version. If the certificates match exactly, including both the certificate data and order, then the system allows the update. If you sign the new version without using matching certificates, you will also need to assign a different package name to the application — in this case, the user installs the new version as a completely new application.
- Application modularity – The Android system allows applications that are signed by the same certificate to run in the same process, if the applications so requests, so that the system treats them as a single application. In this way you can deploy your application in modules, and users can update each of the modules independently if needed.
- Code/data sharing through permissions – The Android system provides signature-based permissions enforcement, so that an application can expose functionality to another application that is signed with a specified certificate. By signing multiple applications with the same certificate and using signature-based permissions checks, your applications can share code and data in a secure manner.

Another important consideration in determining your signing strategy is how to set the validity period of the key that you will use to sign your applications.

- If you plan to support upgrades for a single application, you should ensure that your key has a validity period that exceeds the expected lifespan of that application. A validity period of 25 years or more is recommended. When your key's validity period expires, users will no longer be able to seamlessly upgrade to new versions of your application.
- If you will sign multiple distinct applications with the same key, you should ensure that your key's validity period exceeds the expected lifespan of *all versions of all of the applications*, including dependent applications that may be added to the suite in the future.
- If you plan to publish your application(s) on Android Market, the key you use to sign the application(s) must have a validity period ending after 22 October 2033. The Market server enforces this requirement to ensure that users can seamlessly upgrade Market applications when new versions are available.

As you design your application, keep these points in mind and make sure to use a suitable certificate to sign your applications.

Basic Setup for Signing

Before you begin, you should make sure that Keytool is available to the SDK build tools. In most cases, you can tell the SDK build tools how to find Keytool by setting your `JAVA_HOME` environment variable to references a suitable JDK. Alternatively, you can add the JDK version of Keytool to your `PATH` variable.

If you are developing on a version of Linux that originally came with GNU Compiler for Java, make sure that the system is using the JDK version of Keytool, rather than the gcj version. If Keytool is already in your `PATH`, it might be pointing to a symlink at `/usr/bin/keytool`. In this case, check the symlink target to be sure it points to the Keytool in the JDK.

If you will release your application to the public, you will also need to have the Jarsigner tool available on your machine. Both Jarsigner and Keytool are included in the JDK.

Signing in Debug Mode

The Android build tools provide a debug signing mode that makes it easier for you to develop and debug your application, while still meeting the Android system requirement for signing your .apk. When using debug mode to build your app, the SDK tools invoke Keytool to automatically create a debug keystore and key. This debug key is then used to automatically sign the .apk, so you do not need to sign the package with your own key.

The SDK tools create the debug keystore/key with predetermined names/passwords:

- Keystore name: "debug.keystore"
- Keystore password: "android"
- Key alias: "androiddebugkey"
- Key password: "android"
- CN: "CN=Android Debug,O=Android,C=US"

If necessary, you can change the location/name of the debug keystore/key or supply a custom debug keystore/key to use. However, any custom debug keystore/key must use the same keystore/key names and passwords as the default debug key (as described above). (To do so in Eclipse/ADT, go to **Windows** > **Preferences** > **Android** > **Build**.)

Caution: You *cannot* release your application to the public when signed with the debug certificate.

Eclipse Users

If you are developing in Eclipse/ADT (and have set up Keytool as described above in Basic Setup for Signing), signing in debug mode is enabled by default. When you run or debug your application, ADT signs the .apk with the debug certificate, runs `zipalign` on the package, then installs it on the selected emulator or connected device. No specific action on your part is needed, provided ADT has access to Keytool.

Ant Users

If you are using Ant to build your .apk files, debug signing mode is enabled by using the `debug` option with the `ant` command (assuming that you are using a `build.xml` file generated by the `android` tool). When you run `ant debug` to compile your app, the build script generates a keystore/key and signs the .apk for you. The script then also aligns the .apk with the `zipalign` tool. No other action on your part is needed. Read Building and Running Apps on the Command Line for more information.

Expiry of the Debug Certificate

The self-signed certificate used to sign your application in debug mode (the default on Eclipse/ADT and Ant builds) will have an expiration date of 365 days from its creation date.

When the certificate expires, you will get a build error. On Ant builds, the error looks like this:

```
debug:
[echo] Packaging bin/samples-debug.apk, and signing it with a debug key...
[exec] Debug Certificate expired on 8/4/08 3:43 PM
```

In Eclipse/ADT, you will see a similar error in the Android console.

To fix this problem, simply delete the `debug.keystore` file. The default storage location for AVDs is in `~/.android/` on OS X and Linux, in `C:\Documents and Settings\<user>\.android\` on Windows XP, and in `C:\Users\<user>\.android\` on Windows Vista and Windows 7.

The next time you build, the build tools will regenerate a new keystore and debug key.

Note that, if your development machine is using a non-Gregorian locale, the build tools may erroneously generate an already-expired debug certificate, so that you get an error when trying to compile your application. For workaround information, see the troubleshooting topic I can't compile my app because the build tools generated an expired debug certificate.

Signing for Public Release

When your application is ready for release to other users, you must:

1. Obtain a suitable private key
2. Compile the application in release mode
3. Sign your application with your private key
4. Align the final APK package

If you are developing in Eclipse with the ADT plugin, you can use the Export Wizard to perform the compile, sign, and align procedures. The Export Wizard even allows you to generate a new keystore and private key in the process. So if you use Eclipse, you can skip to Compile and sign with Eclipse ADT.

1. Obtain a suitable private key

In preparation for signing your application, you must first ensure that you have a suitable private key with which to sign. A suitable private key is one that:

- Is in your possession
- Represents the personal, corporate, or organizational entity to be identified with the application
- Has a validity period that exceeds the expected lifespan of the application or application suite. A validity period of more than 25 years is recommended.

  If you plan to publish your application(s) on Android Market, note that a validity period ending after 22 October 2033 is a requirement. You can not upload an application if it is signed with a key whose validity expires before that date.
- Is not the debug key generated by the Android SDK tools.

The key may be self-signed. If you do not have a suitable key, you must generate one using Keytool. Make sure that you have Keytool available, as described in Basic Setup.

To generate a self-signed key with Keytool, use the `keytool` command and pass any of the options listed below (and any others, as needed).

> **Warning:** Keep your private key secure. Before you run Keytool, make sure to read Securing Your Private Key for a discussion of how to keep your key secure and why doing so is critically important to you and to users. In particular, when you are generating your key, you should select strong passwords for both the keystore and key.

Keytool Option	Description
`-genkey`	Generate a key pair (public and private keys)
`-v`	Enable verbose output.
`-alias <alias_name>`	An alias for the key. Only the first 8 characters of the alias are used.
`-keyalg <alg>`	The encryption algorithm to use when generating the key. Both DSA and RSA are supported.
`-keysize <size>`	The size of each generated key (bits). If not supplied, Keytool uses a default key size of 1024 bits. In general, we recommend using a key size of 2048 bits or higher.
`-dname <name>`	A Distinguished Name that describes who created the key. The value is used as the issuer and subject fields in the self-signed certificate. Note that you do not need to specify this option in the command line. If not supplied, Jarsigner prompts you to enter each of the Distinguished Name fields (CN, OU, and so on).
`-keypass <password>`	The password for the key. As a security precaution, do not include this option in your command line. If not

	supplied, Keytool prompts you to enter the password. In this way, your password is not stored in your shell history.
`-validity <valdays>`	The validity period for the key, in days. **Note:** A value of 10000 or greater is recommended.
`-keystore <keystore-name>.keystore`	A name for the keystore containing the private key.
`-storepass <password>`	A password for the keystore. As a security precaution, do not include this option in your command line. If not supplied, Keytool prompts you to enter the password. In this way, your password is not stored in your shell history.

Here's an example of a Keytool command that generates a private key:

```
$ keytool -genkey -v -keystore my-release-key.keystore
-alias alias_name -keyalg RSA -keysize 2048 -validity 10000
```

Running the example command above, Keytool prompts you to provide passwords for the keystore and key, and to provide the Distinguished Name fields for your key. It then generates the keystore as a file called `my-release-key.keystore`. The keystore and key are protected by the passwords you entered. The keystore contains a single key, valid for 10000 days. The alias is a name that you — will use later, to refer to this keystore when signing your application.

For more information about Keytool, see the documentation at http://java.sun.com/j2se/1.5.0/docs/tooldocs/#security

2. Compile the application in release mode

In order to release your application to users, you must compile it in release mode. In release mode, the compiled application is not signed by default and you will need to sign it with your private key.

Caution: You can not release your application unsigned, or signed with the debug key.

With Eclipse

To export an *unsigned* .apk from Eclipse, right-click the project in the Package Explorer and select **Android Tools** > **Export Unsigned Application Package**. Then specify the file location for the unsigned .apk. (Alternatively, open your `AndroidManifest.xml` file in Eclipse, open the *Overview* tab, and click **Export an unsigned .apk**.)

Note that you can combine the compiling and signing steps with the Export Wizard. See Compiling and signing with Eclipse ADT.

With Ant

If you are using Ant, you can enable release mode by using the `release` option with the `ant` command. For example, if you are running Ant from the directory containing your `build.xml` file, the command would look like this:

```
ant release
```

By default, the build script compiles the application .apk without signing it. The output file in your project `bin/` will be `<your_project_name>-unsigned.apk`. Because the application .apk is still unsigned, you must manually sign it with your private key and then align it using `zipalign`.

However, the Ant build script can also perform the signing and aligning for you, if you have provided the path to your keystore and the name of your key alias in the project's `build.properties` file. With this information provided, the build script will prompt you for your keystore and alias password when you perform `ant release`, it will sign the package and then align it. The final output file in `bin/` will instead be `<your_project_name>-release.apk`. With these steps automated for you, you're able to skip the manual procedures below (steps 3 and 4). To learn how to specify your keystore and alias in the `build.properties` file, see Building and Running Apps on the Command Line.

3. Sign your application with your private key

When you have an application package that is ready to be signed, you can do sign it using the Jarsigner tool. Make sure that you have Jarsigner available on your machine, as described in Basic Setup. Also, make sure that the keystore containing your private key is available.

To sign your application, you run Jarsigner, referencing both the application's .apk and the keystore containing the private key with which to sign the .apk. The table below shows the options you could use.

Jarsigner Option	Description
`-keystore <keystore-name>.keystore`	The name of the keystore containing your private key.
`-verbose`	Enable verbose output.
`-storepass <password>`	The password for the keystore. As a security precaution, do not include this option in your command line unless you are working at a secure computer. If not supplied, Jarsigner prompts you to enter the password. In this way, your password is not stored in your shell history.
`-keypass <password>`	The password for the private key. As a security precaution, do not include this option in your command line unless you are working at a secure computer. If not supplied, Jarsigner prompts you to enter the password. In this way, your password is not stored in your shell history.

Here's how you would use Jarsigner to sign an application package called `my_application.apk`, using the example keystore created above.

```
$ jarsigner -verbose -keystore my-release-key.keystore
my_application.apk alias_name
```

Running the example command above, Jarsigner prompts you to provide passwords for the keystore and key. It then modifies the .apk in-place, meaning the .apk is now signed. Note that you can sign an .apk multiple times with different keys.

To verify that your .apk is signed, you can use a command like this:

```
$ jarsigner -verify my_signed.apk
```

If the .apk is signed properly, Jarsigner prints "jar verified". If you want more details, you can try one of these commands:

```
$ jarsigner -verify -verbose my_application.apk
```

or

```
$ jarsigner -verify -verbose -certs my_application.apk
```

The command above, with the `-certs` option added, will show you the "CN=" line that describes who created the key.

> **Note:** If you see "CN=Android Debug", this means the .apk was signed with the debug key generated by the Android SDK. If you intend to release your application, you must sign it with your private key instead of the debug key.

For more information about Jarsigner, see the documentation at http://java.sun.com/j2se/1.5.0/docs/tooldocs/#security

4. Align the final APK package

Once you have signed the .apk with your private key, run `zipalign` on the file. This tool ensures that all uncompressed data starts with a particular byte alignment, relative to the start of the file. Ensuring alignment at 4-byte boundaries provides a performance optimization when installed on a device. When aligned, the Android system is able to read files with `mmap()`, even if they contain binary data with alignment restrictions, rather than copying all of the data from the package. The benefit is a reduction in the amount of RAM consumed by the running application.

The `zipalign` tool is provided with the Android SDK, inside the `tools/` directory. To align your signed .apk, execute:

```
zipalign -v 4 your_project_name-unaligned.apk your_project_name.apk
```

The `-v` flag turns on verbose output (optional). `4` is the byte-alignment (don't use anything other than 4). The first file argument is your signed .apk (the input) and the second file is the destination .apk file (the output). If you're overriding an existing .apk, add the `-f` flag.

> **Caution:** Your input .apk must be signed with your private key **before** you optimize the package with `zipalign`. If you sign it after using `zipalign`, it will undo the alignment.

For more information, read about the zipalign tool.

Compile and sign with Eclipse ADT

If you are using Eclipse with the ADT plugin, you can use the Export Wizard to export a *signed* .apk (and even create a new keystore, if necessary). The Export Wizard performs all the interaction with the Keytool and Jarsigner for you, which allows you to sign the package using a GUI instead of performing the manual procedures to compile, sign, and align, as discussed above. Once the wizard has compiled and signed your package, it will also perfom package alignment with `zipalign`. Because the Export Wizard uses both Keytool and Jarsigner, you should ensure that they are accessible on your computer, as described above in the Basic Setup for Signing.

To create a signed and aligned .apk in Eclipse:

1. Select the project in the Package Explorer and select **File > Export**.
2. Open the Android folder, select Export Android Application, and click **Next**.

   The Export Android Application wizard now starts, which will guide you through the process of signing your application, including steps for selecting the private key with which to sign the .apk (or creating a new keystore and private key).
3. Complete the Export Wizard and your application will be compiled, signed, aligned, and ready for distribution.

Securing Your Private Key

Maintaining the security of your private key is of critical importance, both to you and to the user. If you allow someone to use your key, or if you leave your keystore and passwords in an unsecured location such that a third-party could find and use them, your authoring identity and the trust of the user are compromised.

If a third party should manage to take your key without your knowledge or permission, that person could sign and distribute applications that maliciously replace your authentic applications or corrupt them. Such a person could also sign and distribute applications under your identity that attack other applications or the system itself, or corrupt or steal user data.

Your reputation as a developer entity depends on your securing your private key properly, at all times, until the key is expired. Here are some tips for keeping your key secure:

- Select strong passwords for the keystore and key.
- When you generate your key with Keytool, *do not* supply the `-storepass` and `-keypass` options at the command line. If you do so, your passwords will be available in your shell history, which any user on your computer could access.
- Similarly, when signing your applications with Jarsigner, *do not* supply the `-storepass` and `-keypass` options at the command line.
- Do not give or lend anyone your private key, and do not let unauthorized persons know your keystore and key

passwords.

In general, if you follow common-sense precautions when generating, using, and storing your key, it will remain secure.

↑ Go to top