**Signing Your Applications**

**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**

1. [Signing Process](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#signing)
2. [Signing Strategies](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#strategies)
3. [Basic Setup for Signing](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#setup)
4. [Signing in Debug Mode](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#debugmode)
5. [Signing Release Mode](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#releasemode)
   1. [Obtain a suitable private key](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#cert)
   2. [Compile the application in release mode](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#releasecompile)
   3. [Sign your application with your private key](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#signapp)
   4. [Align the final APK package](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#align)
   5. [Compile and sign with Eclipse ADT](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#ExportWizard)
6. [Securing Your Private Key](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#secure-key)

**SEE ALSO**

1. [Versioning Your Applications](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/versioning.html)
2. [Preparing to Publish](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/preparing.html)

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 on an emulator or a device if it is not signed.
* To test and debug your application, the build tools sign your application with a special debug key that is created by the Android SDK build tools.
* When you are ready to release your application for end-users, you must sign it with a suitable private key. You cannot publish an application that is signed with the debug key generated by the SDK tools.
* You can use self-signed certificates to sign your applications. No certificate authority is needed.
* 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.
* After you sign your application for release, we recommend that you use thezipalign 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](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#setup) for your application before you can run it or debug it on an emulator or device.

**Signing Process**

The Android build process signs your application differently depending on which build mode you use to build your application. There are two build modes: *debug mode* and *release mode*. You use debug mode when you are developing and testing your application. You use release mode when you want to build a release version of your application that you can distribute directly to users or publish on an application marketplace such as Google Play.

When you build in *debug mode* the Android SDK build tools use the Keytool utility (included in the JDK) to create a debug key. Because the SDK build tools created the debug key, they know the debug key's alias and password. Each time you compile your application in debug mode, the build tools use the debug key along with the Jarsigner utility (also included in the JDK) to sign your application's .apk file. Because the alias and password are known to the SDK build tools, the tools don't need to prompt you for the debug key's alias and password each time you compile.

When you build in *release mode* you use your own private key to sign your application. If you don't have a private key, you can use the Keytool utility to create one for you. When you compile your application in release mode, the build tools use your private key along with the Jarsigner utility to sign your application's .apk file. Because the certificate and private key you use are your own, you will have to provide the password for the keystore and key alias.

The debug signing process happens automatically when you run or debug your application using Eclipse with the ADT plugin. Debug signing also happens automatically when you use the Ant build script with the debug option. You can automate the release signing process by using the Eclipse Export Wizard or by modifying the Ant build script and building with the release option.

**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 Google Play, the key you use to sign the application(s) must have a validity period ending after 22 October 2033. Google Play enforces this requirement to ensure that users can seamlessly upgrade applications when new versions are available.

As you design your application, keep these points in mind and make sure to use a [suitable certificate](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#cert) to sign your applications.

**Basic Setup for Signing**

Before you begin, make sure that the Keytool utility and Jarsigner utility are available to the SDK build tools. Both of these tools are available in the JDK. In most cases, you can tell the SDK build tools how to find these utilities by setting your JAVA\_HOME environment variable so it references a suitable JDK. Alternatively, you can add the JDK version of Keytool and Jarsigner 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.

**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 and Jarsigner as described above in [Basic Setup for Signing](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#setup)), signing in debug mode is enabled by default. When you run or debug your application, ADT signs the .apk file 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 file, debug signing mode is enabled by using the debug option with the antcommand (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](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/building/building-cmdline.html#DebugMode) 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](https://stuff.mit.edu/afs/sipb/project/android/docs/resources/faq/troubleshooting.html#signingcalendar).

**Signing in Release Mode**

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

1. [Obtain a suitable private key](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#cert)
2. [Compile the application in release mode](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#releasecompile)
3. [Sign your application with your private key](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#signapp)
4. [Align the final APK package](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#align)

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](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#ExportWizard).

**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 Google Play, 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](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#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](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#secure-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://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html>

**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 yourAndroidManifest.xml file in Eclipse, select the **Manifest** 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](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#ExportWizard).

**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 ant.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 ant.properties file, see [Building and Running Apps on the Command Line](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/building/building-cmdline.html#ReleaseMode).

**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](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#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. |
| -sigalg | The name of the signature algorithim to use in signing the APK. Use the valueMD5withRSA. |
| -digestalg | The message digest algorithim to use in processing the entries of an APK. Use the value SHA1. |
| -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 -sigalg MD5withRSA -digestalg SHA1 -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.

**Caution:** As of JDK 7, the default signing algorithim has changed, requiring you to specify the signature and digest algorithims (-sigalg and -digestalg) when you sign an APK.

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://docs.oracle.com/javase/6/docs/technotes/tools/windows/jarsigner.html>

**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 file (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](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/help/zipalign.html) 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 withzipalign. 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](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html#setup).

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).

1. 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.