Signing Builds for Release

Android OS images use cryptographic signatures in two places:

1. Each .apk file inside the image must be signed. Android's Package Manager uses an .apk signature in two ways:
    - When an application is replaced, it must be signed by the same key as the old application in order to get access to the old application's data. This holds true both for updating user apps by overwriting the .apk, and for overriding a system app with a newer version installed under **`/data`**.
    - If two or more applications want to share a user ID (so they can share data, etc.), they must be signed with the same key.
2. OTA update packages must be signed with one of the keys expected by the system or the installation process will reject them.

Release keys

The Android tree includes *test-keys* under **`build/target/product/security`**. Building an Android OS image using **`make`** will sign all .apk files using the test-keys. Since the test-keys are publicly known, anybody can sign their own .apk files with the same keys, which may allow them to replace or hijack system apps built into your OS image. For this reason it is critical to sign any publicly released or deployed Android OS image with a special set of *release-keys* that only you have access to.

To generate your own unique set of release-keys, run these commands from the root of your Android tree:

```
$ subject='/C=US/ST=California/L=Mountain View/O=Android/OU=Android/CN=Android/emailAddress=android@android.com'
$ mkdir ~/.android-certs
$ for x in releasekey platform shared media; do \
    ./development/tools/make_key ~/.android-certs/$x "$subject"; \
done
```

`$subject` should be changed to reflect your organization's information. You can use any directory, but be careful to pick a location that is backed up and secure. Some vendors choose to encrypt their private key with a strong passphrase and store the encrypted key in source control; others store their release keys somewhere else entirely, such as on an air-gapped computer.

To generate a release image, use:

```
$ make dist
$ ./build/tools/releasetools/sign_target_files_apks \
    -o \    # explained in the next section
    -d ~/.android-certs out/dist/*-target_files-*.zip \
    signed-target_files.zip
```

The **`sign_target_files_apks`** script takes a target-files .zip as input and produces a new target-files .zip in which all the .apks have been signed with new keys. The newly signed images can be found under **`IMAGES/`** in **`signed-target_files.zip`**.

Signing OTA packages

A signed target-files zip can be converted into a signed OTA update zip using the following procedure:

```
$ ./build/tools/releasetools/ota_from_target_files \
    -k ~/.android-certs/releasekey \
    signed-target_files.zip \
    signed-ota_update.zip
```

Signatures and sideloading

Sideloading does not bypass recovery's normal package signature verification mechanism—before installing a package, recovery will verify that it is signed with one of the private keys matching the public keys stored in the recovery partition, just as it would for a package delivered over-the-air.

Update packages received from the main system are typically verified twice: once by the main system, using the

`RecoverySystem.verifyPackage()` (http://developer.android.com/reference/android/os/RecoverySystem.html#verifyPackage) method in the android API, and then again by recovery. The RecoverySystem API checks the signature against public keys stored in the main system, in the file **`/system/etc/security/otacerts.zip`** (by default). Recovery checks the signature against public keys stored in the recovery partition RAM disk, in the file **`/res/keys`**.

By default, the target-files .zip produced by the build sets the OTA certificate to match the test key. On a released image, a different certificate must be used so that devices can verify the authenticity of the update package. Passing the **`-o`** flag to **`sign_target_files_apks`**, as shown in the previous section, replaces the test key certificate with the release key certificate from your certs directory.

Normally the system image and recovery image store the same set of OTA public keys. By adding a key to *just* the recovery set of keys, it is possible to sign packages that can be installed only via sideloading (assuming the main system's update download mechanism is correctly doing verification against otacerts.zip). You can specify extra keys to be included only in recovery by setting the PRODUCT_EXTRA_RECOVERY_KEYS variable in your product definition:

```
vendor/yoyodyne/tardis/products/tardis.mk

 [...]

PRODUCT_EXTRA_RECOVERY_KEYS := vendor/yoyodyne/security/tardis/sideload
```

This includes the public key **`vendor/yoyodyne/security/tardis/sideload.x509.pem`** in the recovery keys file so it can install packages signed with it. The extra key is *not* included in otacerts.zip though, so systems that correctly verify downloaded packages do not invoke recovery for packages signed with this key.

Certificates and private keys

Each key comes in two files: the *certificate*, which has the extension .x509.pem, and the *private key*, which has the extension .pk8. The private key should be kept secret and is needed to sign a package. The key may itself be protected by a password. The certificate, in contrast, contains only the public half of the key, so it can be distributed widely. It is used to verify a package has been signed by the corresponding private key.

The standard Android build uses four keys, all of which reside in **`build/target/product/security`**:

testkey

Generic default key for packages that do not otherwise specify a key.

platform

Test key for packages that are part of the core platform.

shared

Test key for things that are shared in the home/contacts process.

media

Test key for packages that are part of the media/download system.

Individual packages specify one of these keys by setting LOCAL_CERTIFICATE in their Android.mk file. (testkey is used if this variable is not set.) You can also specify an entirely different key by pathname, e.g.:

```
device/yoyodyne/apps/SpecialApp/Android.mk

 [...]

LOCAL_CERTIFICATE := device/yoyodyne/security/special
```

Now the build uses the **`device/yoyodyne/security/special.{x509.pem,pk8}`** key to sign SpecialApp.apk. The build can use only private keys that are *not* password protected.

Advanced signing options

When you run the **`sign_target_files_apks`** script, you must specify on the command line a replacement key for each key used in the build. The **`-k`** ***`src_key`*****`=`** ***`dest_key`*** flag specifies key replacements one at a time. The flag **`-d`** ***`dir`*** lets you specify a directory with four keys to replace all those in **`build/target/product/security`**; it is equivalent to using **`-k`** four times to specify the mappings:

```
build/target/product/security/testkey  = dir/releasekey
build/target/product/security/platform = dir/platform
build/target/product/security/shared   = dir/shared
build/target/product/security/media    = dir/media
```

For the hypothetical tardis product, you need five password-protected keys: four to replace the four in **`build/target/product/security`**, and one to replace the additional **`keydevice/yoyodyne/security/special`** required by SpecialApp in the example above. If the keys were in the following files:

```
vendor/yoyodyne/security/tardis/releasekey.x509.pem
vendor/yoyodyne/security/tardis/releasekey.pk8
vendor/yoyodyne/security/tardis/platform.x509.pem
vendor/yoyodyne/security/tardis/platform.pk8
vendor/yoyodyne/security/tardis/shared.x509.pem
vendor/yoyodyne/security/tardis/shared.pk8
vendor/yoyodyne/security/tardis/media.x509.pem
vendor/yoyodyne/security/tardis/media.pk8
vendor/yoyodyne/security/special.x509.pem
vendor/yoyodyne/security/special.pk8           # NOT password protected
vendor/yoyodyne/security/special-release.x509.pem
vendor/yoyodyne/security/special-release.pk8   # password protected
```

Then you would sign all the apps like this:

```
$ ./build/tools/releasetools/sign_target_files_apks -d vendor/yoyodyne/security/tardis -k vendor/yoyodyne/special=ven
```

This brings up the following:

```
Enter password for vendor/yoyodyne/security/special-release key>
Enter password for vendor/yoyodyne/security/tardis/media key>
Enter password for vendor/yoyodyne/security/tardis/platform key>
Enter password for vendor/yoyodyne/security/tardis/releasekey key>
Enter password for vendor/yoyodyne/security/tardis/shared key>
    signing: Phone.apk (vendor/yoyodyne/security/tardis/platform)
    signing: Camera.apk (vendor/yoyodyne/security/tardis/media)
    signing: Special.apk (vendor/yoyodyne/security/special-release)
    signing: Email.apk (vendor/yoyodyne/security/tardis/releasekey)
        [...]
    signing: ContactsProvider.apk (vendor/yoyodyne/security/tardis/shared)
    signing: Launcher.apk (vendor/yoyodyne/security/tardis/shared)
rewriting SYSTEM/build.prop:
  replace:  ro.build.description=tardis-user Eclair ERC91 15449 test-keys
     with:  ro.build.description=tardis-user Eclair ERC91 15449 release-keys
  replace: ro.build.fingerprint=generic/tardis/tardis/tardis:Eclair/ERC91/15449:user/test-keys
     with: ro.build.fingerprint=generic/tardis/tardis/tardis:Eclair/ERC91/15449:user/release-keys
    signing: framework-res.apk (vendor/yoyodyne/security/tardis/platform)
rewriting RECOVERY/RAMDISK/default.prop:
  replace:  ro.build.description=tardis-user Eclair ERC91 15449 test-keys
     with:  ro.build.description=tardis-user Eclair ERC91 15449 release-keys
  replace: ro.build.fingerprint=generic/tardis/tardis/tardis:Eclair/ERC91/15449:user/test-keys
     with: ro.build.fingerprint=generic/tardis/tardis/tardis:Eclair/ERC91/15449:user/release-keys
using:
    vendor/yoyodyne/security/tardis/releasekey.x509.pem
for OTA package verification
done.
```

After prompting the user for passwords for all password-protected keys, the script re-signs all the .apk files in the input target .zip with the release keys. Before running the command, you can also set the ANDROID_PW_FILE environment variable to a temporary filename; the script then invokes your editor to allow you to enter passwords for all keys (this may be a more convenient way to enter passwords).

`sign_target_files_apks` also rewrites the build description and fingerprint in the build properties files to reflect the fact that this is a signed build. The **`-t`** flag can control what edits are made to the fingerprint. Run the script with **`-h`** to see documentation on all flags.

Manually generating keys

Android uses 2048-bit RSA keys with public exponent 3. You can generate certificate/private key pairs using the openssl tool from openssl.org (https://www.openssl.org/):

```
# generate RSA key
$ openssl genrsa -3 -out temp.pem 2048
Generating RSA private key, 2048 bit long modulus
....+++
.....................+++
e is 3 (0x3)

# create a certificate with the public part of the key
$ openssl req -new -x509 -key temp.pem -out releasekey.x509.pem -days 10000 -subj '/C=US/ST=California/L=San Narciso/

# create a PKCS#8-formatted version of the private key
$ openssl pkcs8 -in temp.pem -topk8 -outform DER -out releasekey.pk8 -nocrypt

# securely delete the temp.pem file
$ shred --remove temp.pem
```

The openssl pkcs8 command given above creates a .pk8 file with *no* password, suitable for use with the build system. To create a .pk8 secured with a password (which you should do for all actual release keys), replace the **`-nocrypt`** argument with **`-passout stdin`**; then openssl will encrypt the private key with a password read from standard input. No prompt is printed, so if stdin is the terminal the program will appear to hang when it's really just waiting for you to enter a password. Other values can be used for the-passout argument to read the password from other locations; for details, see the openssl documentation (http://www.openssl.org/docs/man1.0.1/apps/openssl.html#PASS-PHRASE-ARGUMENTS).

The temp.pem intermediate file contains the private key without any kind of password protection, so dispose of it thoughtfully when generating release keys. In particular, the GNUshred utility may not be effective on network or journaled filesystems. You can use a working directory located in a RAM disk (such as a tmpfs partition) when generating keys to ensure the intermediates are not inadvertently exposed.

Creating image files

Once you have signed-target-files.zip, you need to create the image so you can put it onto a device. To create the signed image from the target files, run the following command from the root of the Android tree:

```
$ ./build/tools/releasetools/img_from_target_files signed-target-files.zip signed-img.zip
```

The resulting file, **`signed-img.zip`**, contains all the .img files. To load an image onto a device, use fastboot as follows:

```
$ fastboot update signed-img.zip
```

Last updated June 20, 2017.