An Introduction To
Popular Android Exploits
and what makes them possible

April, 2016

Questions

Can a benign service call a dangerous service without the user knowing?

Can Google Play determine whether code has security holes and prevent it from being added to the market?

Can code that has security holes be exploited by a third party?

Does proguard prevent repackaging attacks?

Obfuscation

Proguard:

Proguard makes it a bit harder to Reverse Engineer, but it will still be possible (and the APKtool gives you the possibility to debug). Moreover, you cannot use all of proguard optimization because you will not be able to convert classes to dex. In fact, you can only use shrink and agressive overloading.

Bottom line: proguard lets you shrink you code about 30% but it will not make your application hackproof.

`http://proguard.sourceforge.net/`

`http://developer.android.com/tools/help/proguard.html`

`https://groups.google.com/forum/#!topic/android-developers/MsXzNCqWKpc`

Reference

What follows is from:

Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications

Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna

Network and Distributed Sysytem Symposium

February, 2014

Google Bouncer

Bouncer quietly and automatically scans apps (both new and previously uploaded ones) and developer accounts in Google Play with its reputation engine and cloud infrastructure.

But

1. Bouncer can be fingerprinted - an app can know that it is being scanned by the Bouncer
2. External code can be added to a running app that was downloaded from the Play store
3. Android does not enforce security checks on such code
4. Numerous benign apps add external code routinely
5. Developers of many benign apps are unaware of or do not properly implement protection mechanisms

Android OS Environment

Additional considerations

1. External code can come from anywhere, including other app stores such as Amazon's App Store
2. Android OS treats the store that the device manufacturer prefers differently from others:

   To install APKs from other stores users enable the sideloading setting, otherwise the OS rejects apps that do not originate from the preferred store.
3. Users may be forced to accept external code (assumed benign - and most are)
   otherwise their app won't have some desired feature or may not update
4. Android OS does not check the integrity of class files & applications are run without checking signatures

Common Exploits

Malware escapes offline analysis

Once an application has been approved by Google Bouncer, admitted to the store and installed by users, it can download and execute additional code that could be malware

Malware is injected into benign apps

Attacker can replace the original code with malicious code because Android OS does not enforce security checks

Attacker runs the malicious code with the original permissions of the app!

Instruments

Class Loaders

- Allow programs to load additional classes
- Used by Android programs to load classes from arbitrary files
- Android class loader accepts `apk`, `dex`, `jar` formats
- No restrictions on location or source of a class
- App may download an `apk` file from the internet and use `DexClassLoader` to load the contained classes then the methods of those classes can be invoked to run malware

Instruments

`DexClassLoader`

- A class loader that loads classes from `.jar` and `.apk` files containing a `classes.dex` entry. This can be used to execute code not installed as part of an application.

  This class loader requires an application-private, writable directory to cache optimized classes. Use:

  ```
  File dexOutputDir = context.getDir("dex", 0);
  ```

  to create such a directory

- From class `Context`:

  mode: Use 0 or `MODE_PRIVATE` as default, `MODE_WORLD_READABLE` and `MODE_WORLD_WRITEABLE` control permissions (dangerous and deprecated)

`http://developer.android.com/reference/dalvik/system/DexClassLoader.html`
`http://developer.android.com/reference/android/content/Context.html`

Instruments

Package Contexts

- A `Context` object is associated with an app on load
- The object provides access to the app's resource
- Android OS provides `createPackageContext` to create contexts for other installed apps - just need to know their package name
- This allows an app to access resources of another app, & create a class loader for loading classes of that app
- If flags `CONTEXT_INCLUDE_CODE` and `CONTEXT_IGNORE_SECURITY` are set when calling `createPackageContext` the OS does not verify that the app originates from the same developer or that the app to be loaded satisfies any criteria
- So, an app can load and execute code from any app

`http://developer.android.com/reference/android/content/Context.html`

Instruments

Package Contexts

- Many applications use `CONTEXT_IGNORE_SECURITY`
- If an attacker manages to install a package with the same name as an app that is careless about checking integrity and authenticating then the attacker's code can be executed with the app's permissions

Instruments

Context

- Interface to global information about an application environment. Allows access to application-specific resources and classes, as well as up-calls for application-level operations such as launching activities, broadcasting and receiving intents, etc.
- Constants:
  - `CONTEXT_IGNORE_SECURITY`: ignore any security restrictions on the Context being requested, allowing it to always be loaded.
  - `CONTEXT_INCLUDE_CODE`: include the application code with the context.
- `createPackageContext(String name, int flags)`:
  The returned `Context` is the same as what the named application gets when it is launched, containing the same resources and class loader

Instruments

Native Code

- Apps are allowed to run native code via JNI
- Android OS makes some checks on resource access for example, creating a network socket, an op done by root, is not allowed directly for the native code but is allowed with permission
- But an attacker can run native code in several different ways, without the burden of conforming to a well-defined API
- Native code can be downloaded and executed at runtime
- Only need app to have internet permission
- All current root exploits are native code - added after app installation

Instruments

`Runtime.exec`

- Allows apps to execute arbitrary binaries
- Gives an app access to a bash shell
- No check is made on the binary to be executed
- An attacker can use system calls to execute arbitrary binaries

Instruments

APK installation

- The Package Manager can be used by an owner to install and uninstall apps
- The PM prompts to have owner accept permissions
- The PM requires a signed certificate to install
  But it does not check anything about the certificate
  It is only used to determine whether two apps have the same source
- If an attacker can replace the `apk` that a benign app tries to install, the app does not detect the switch unless it implements a custom verification mechanism

Sideloading

- An owner has to enable sideloading in the system settings to install apps from any source other than the preferred store of the device manufacturer
- But any user who wants to use an alternative application store has to do the same
- To assist users in the process of setting up their devices, providers of such application stores usually offer detailed instructions on how to find the sideloading settings, often without warning about potential security implications.
- Thus, it is reasonable to assume that sideloading is enabled on a considerable fraction of Android devices
- Facebook has used direct updates instead of going through Google Play in the past

Why is Loading External Code Allowed?

There are legitmate reasons for loading external code

- Until recently, developers did beta testing on a subset of users via external `apk` loading.
- Apps can be extended by installing additional modules. But such apps are on their own in checking whether the add-ons are legitimate
- Framework developers use external `apk` loading to auto-update their frameworks to all users
- Unfortunately, implementations of such auto-updaters can be flawed and vulnerable to injection attacks

Exploiting External Code Loading

Evasion of the Bouncer

- Benign code that does the following:
  1. uses permission to visit the internet
  2. uses permission to write files to external storage
  3. Activity contains a single button - when pressed, code is downloaded from a site and the user sees a browser (to hide the download)
  4. The downloaded code is executed
- Bouncer did not detect the potential to download and execute malicious code
- Bouncer did not even request the download
- Same results for available anti-virus apps

Exploiting External Code Loading

Code substitution

- Android OS directs responsibility for checking the integrity and authenticity of external code to the app or framework developers

  Some apps download external code via http and are subject to man-in-the-middle attacks

  Some apps download external code to storage that is write-accessible to other apps

Exploiting External Code Loading

Improper package name usage

- The same package name can be used by several different applications as long as they are not installed on the same device

  An attacker can write malicious code in a package with a well-known name

  If the package is downloaded and installed, all apps using the package will be affected.

Exploiting External Code Loading

Self-update of an advertising framework

- A game that includes an advertising framework that can update itself via http
- Framework checks for updates upon game start
- Update is downloaded and started via `DexClassLoader`
- Connection was taken over and malicious file plus an MD5 hash was served
- The MD5 hash was checked for file integrity but no authentication was done
- The app expects a particular class name and a method to execute. These can be determined from decoding via `apktool`

Exploiting External Code Loading

Bootstrapping mechanism of a shared framework

- Framework allows developers to create apps for several platforms
- Device-specific framework runtime runs the code
- Android version is an app that can be started by any app that is based on it
- Code loading the framework runtime into an app is generated automatically for the developer
  Uses `createPackageContext` with hard-coded package name
- Loading code does not verify the integrity of the loaded app so any package with the right name is accepted

Exploiting External Code Loading

Bootstrapping mechanism of a shared framework

- If attacker can install bogus code with the same package name and required class, when an app based on the framework is launched, the bogus code will run

Intent Spoofing

- Intent: main method of interprocess communication used to start activities and services and notify broadcast receivers
- Component can be configured to accept intents from components of other apps with `android:exported="true"` in the manifest
- Registering an implicit intent makes it exported automatically
- Example:

```
am start \
  -a android.intent.action.SENDTO \
  -d mailto:adam@palominolabs.com \
  --es com.paypal.android.p2pmobile.Amount 9.99 \
  --ei com.paypal.android.p2pmobile.ParamType 42 \
  -n com.paypal.android.p2pmobile/.activity.SendMoneyActivity
```

see `http://blog.palominolabs.com/2013/05/13/android-security/`

Intent Interception

- A malicious app receives an intent that was not intended for it.
- Can cause a leak of sensitive information
- Can result in the malicious component being activated instead of the legitimate component.
- However, a broadcast may be secured with a permission - then a component will not receive that intent unless it has that permission
- Example:

  `https://play.google.com/store/apps/details?id=uk.co.ashtonbrsc.android.intentintercept`

Android Malware Examples

- Millions of phones have bitcoin mining malware
  `https://www.theguardian.com/technology/2014/mar/27/android-bitcoin-dogecoin-mining-malware`
- Adware problems
  `http://www.digitaltrends.com/mobile/google-removes-adware-app-from-google-play/`
- Virus Shield: fake app
  `https://www.theguardian.com/technology/2014/apr/10/fake-android-antivirus-app-developer-virus-shield`
- Android.bankun: banking malware
  `http://www.webroot.com/blog/2013/07/03/android-bankun-bank-information-stealing-application-on-your-android-device/`
- Android.koler: drive-by-download:
  `http://techspective.net/2015/07/02/drive-by-downloads-android-koler/`
- Premium SMS messages
  `http://netsecurity.about.com/od/securityadvisorie1/a/How-To-Protect-Yourself-From-Premium-Sms-Text-Message-Scams.htm`
- Dendroid: remote access trojan
  `https://blog.lookout.com/blog/2014/03/06/dendroid/`