# Execute This!

## Looking into code-loading techniques on Android

Sebastian Poeplau
(with Yanick Fratantonio, Antonio Bianchi,
Christopher Kruegel, Giovanni Vigna

# Code loading in Android

* Apps can load code dynamically at runtime

  * E.g., download code from the Internet

  * Various ways (DexClassLoader, CreatePackageContext, etc.)

* Good news: Permissions enforced on external code

* Bad news: No additional checks

```java
// Create cache directory if necessary
String dexDir = "LoaderOptimized";
File optimized = context.getDir(dexDir, 0);
try {
    optimized.createNewFile();
} catch (IOException e) {
    logView.append("Error: execution failed\n");
    Log.e(TAG, "Dex dir creation failed", e);
}

// Create the class loader
DexClassLoader loader = new DexClassLoader(
        mExecutableFile.getAbsolutePath(),
        optimized.getAbsolutePath(),
        null, context.getClassLoader());
try {
    // Find class and method
    Class<?> remote = loader.loadClass("MyRemoteClass");
    Method run = remote.getMethod("doSomething",
            Context.class);

    // Instantiate the class
    Object code = remote.newInstance();

    // Invoke the method on the instance
    run.invoke(code, context);
} catch (ClassNotFoundException e) {
    logView.append("Error: execution failed\n");
    Log.e(TAG, "Unable to load the class", e);
} catch (InstantiationException e) {
    logView.append("Error: execution failed\n");
    Log.e(TAG, "Unable to instantiate the class", e);
} catch (IllegalAccessException e) {
    logView.append("Error: execution failed\n");
    Log.e(TAG, "Access to the class forbidden", e);
} catch (NoSuchMethodException e) {
```

# Implications

1. Malicious apps

   ✳ Download arbitrary additional code to circumvent offline analysis

   ✳ Reminder: Checks run at the store

   ✳ Conceptual flaw in the stores' vetting process

2. Benign apps

   ✳ ...use code-loading techniques as well (details later)

   ✳ Must implement custom security mechanisms

   ✳ Dangerous vulnerabilities

# Proof-of-concept exploits

# Bypassing Google Bouncer

* Simple downloader app

    * Connects to our server

    * Downloads a payload

    * Executes the payload

* Submitted to Google Play in April 2013, accepted within 90 minutes

* Allows to run arbitrary code on users' devices

    * Even targeted payloads possible

    * Remark: we refrained from using it on other people's devices...
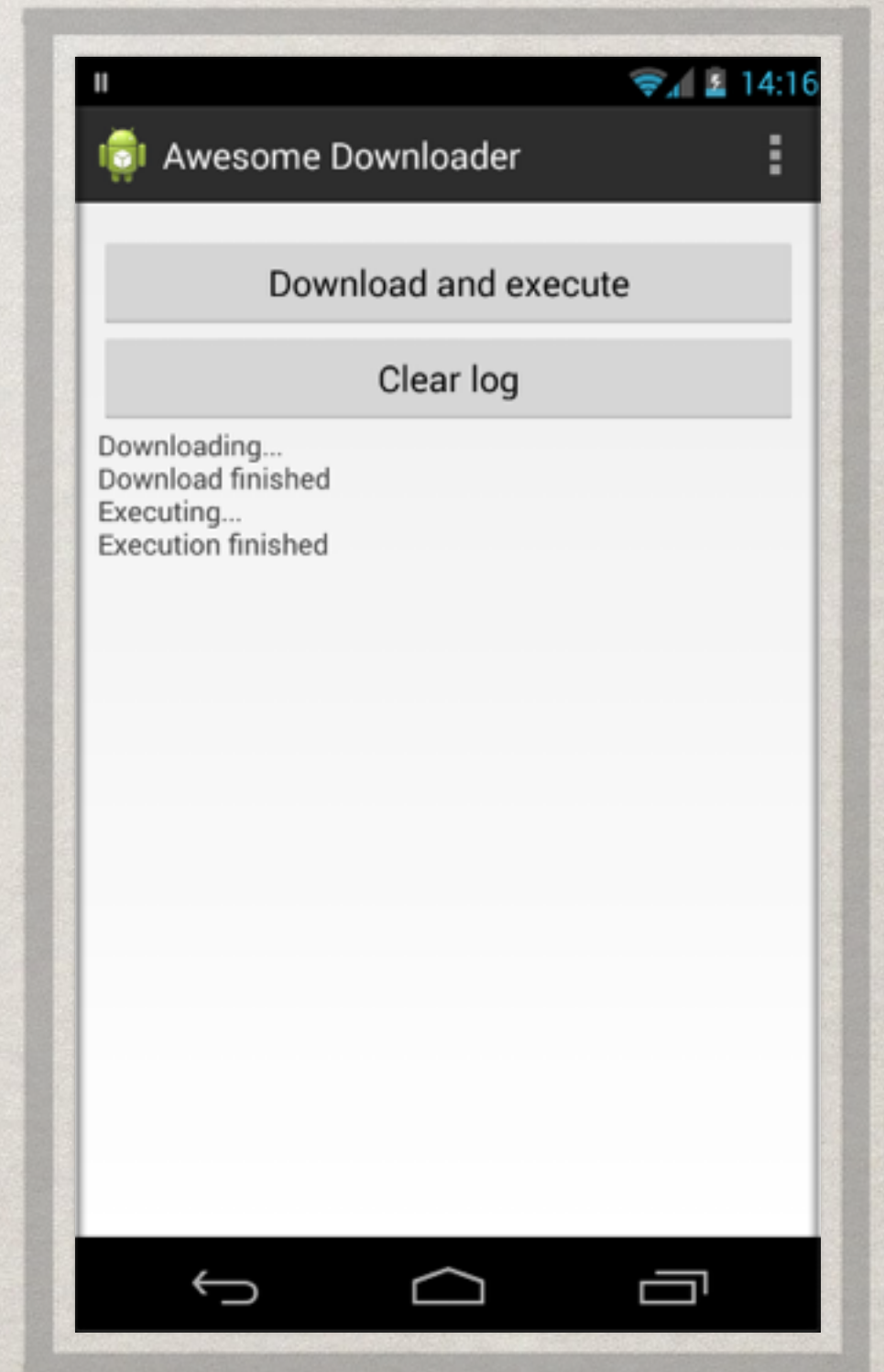
# Gunzombie exploit

* Benign app, among top 50 in November 2012, millions of users

* Includes advertisement framework AppLovin

    * Framework tries to download updates...

    * ...on every app launch...

    * ...via HTTP!

    * No real integrity/authenticity checks

* App is vulnerable to code injection (by hijacking the HTTP connection)
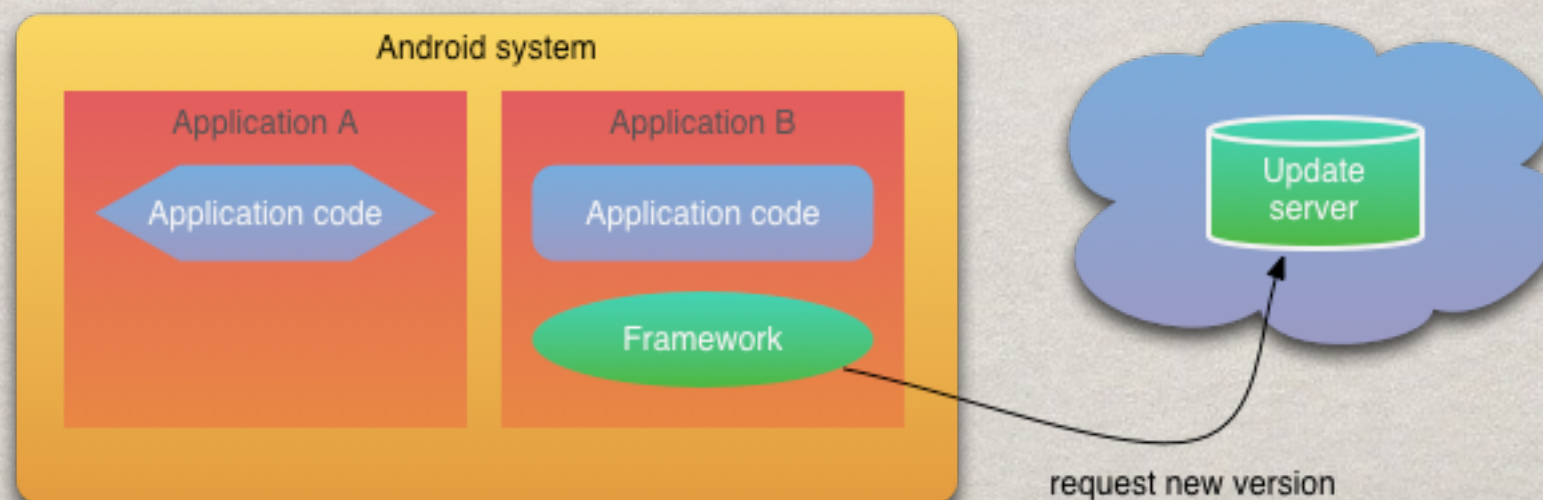
# Attacking a shared framework

* Popular framework for app development (not named here)

  * Installed as a stand-alone app

  * Loaded via app identifier

  * App identifiers are not globally unique!

* We inject code by installing an app with the same identifier first
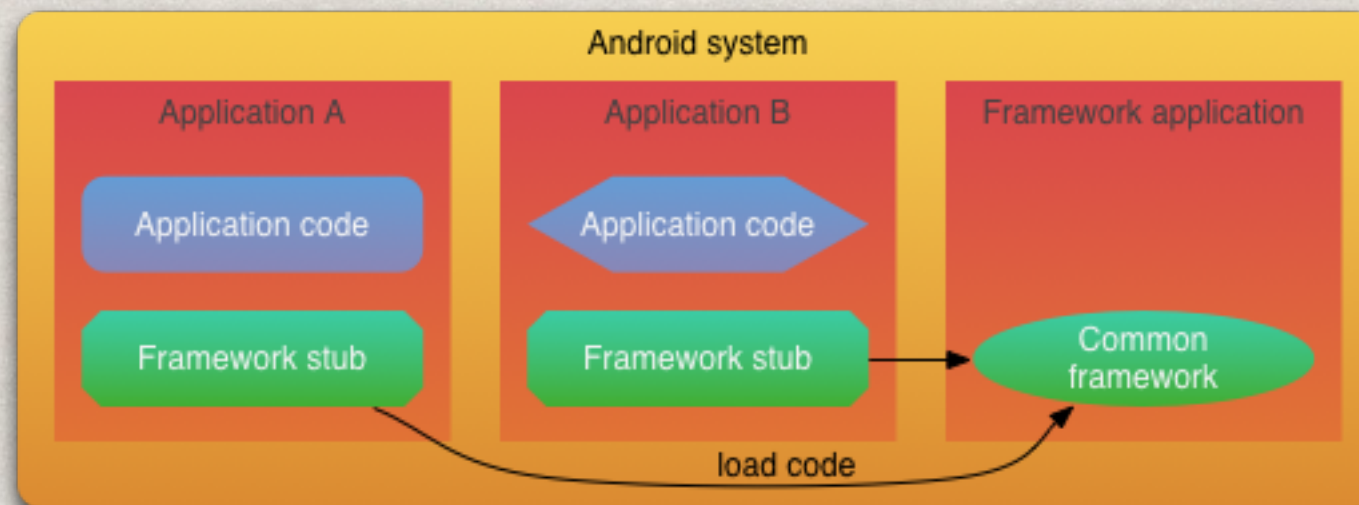
# Large-scale study

# How prevalent is the problem?

- Goal: assess percentage of apps vulnerable to code injection due to dynamic loading

- Test set: 1,632 apps from Google Play, each with 1,000,000+ installations

- Secondary test sets: top 50 free apps as of November 2012 and August 2013, respectively

- Technique: static analysis, heuristics to detect code-loading techniques (more later)

# Loading techniques

- Various ways to load external code

    - Load JARs, APKs, DEX files (compiled Java code)

    - Linux shared objects (native code)

    - Load code from other apps

    - Install APKs (requires user approval)

- Various pitfalls...

    - Insecure downloads using HTTP

    - Download to world-writable storage locations

    - Assumption of package name uniqueness

# Detection approach

* Goal: find code loading and detect vulnerable implementations

* Construct CFG with the help of Androguard

* Transformation into SSA

* Context-insensitive call graph construction based on class hierarchy analysis

* Heuristics based on backward slicing

    * Determine value of sensitive API parameters

        * Example *createPackageContext(name, flags)*: check that flags cause runtime environment to load code

    * Classification step based on heuristics

    * Heuristics for all previously mentioned loading techniques

# Analysis results

☀ 9.25% out of 1,632 apps vulnerable

☀ Similar situation among top apps

☀ Alarming tendency: more vulnerable apps in top 50 in August 2013 than November 2012

☀ Different motivations for use of code loading

  ☀ Updates (e.g., AppLovin)

  ☀ Shared components
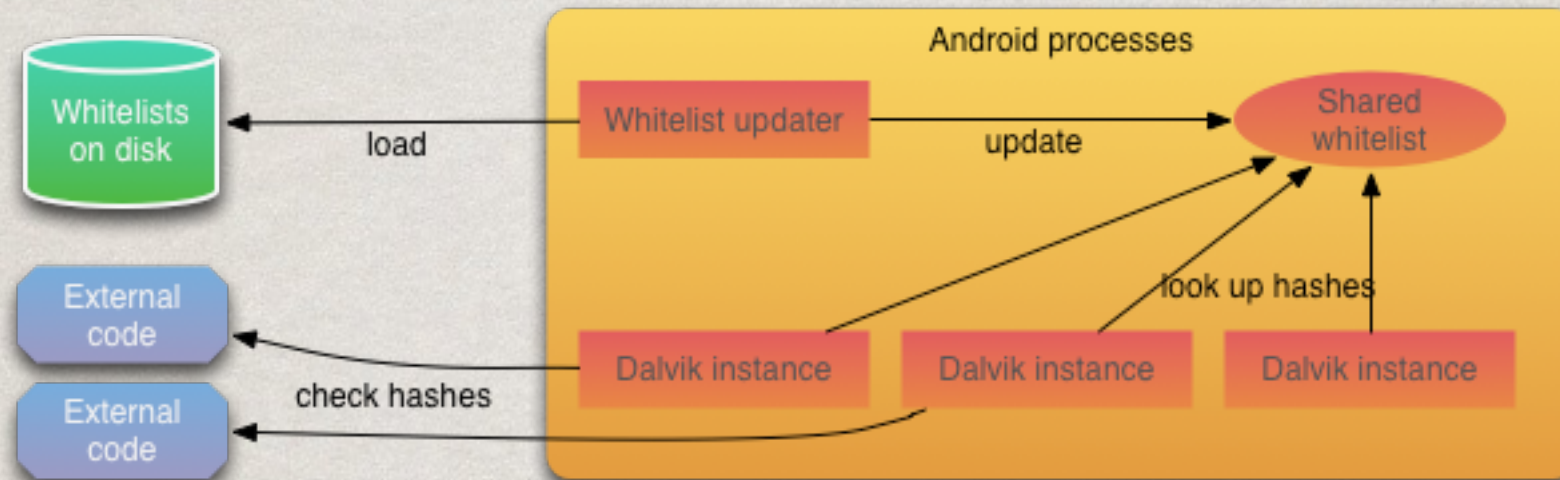
  ☀ A/B and beta testing

  ☀ Loading add-ons

# Our Protection Mechanism

# Whitelisting scheme
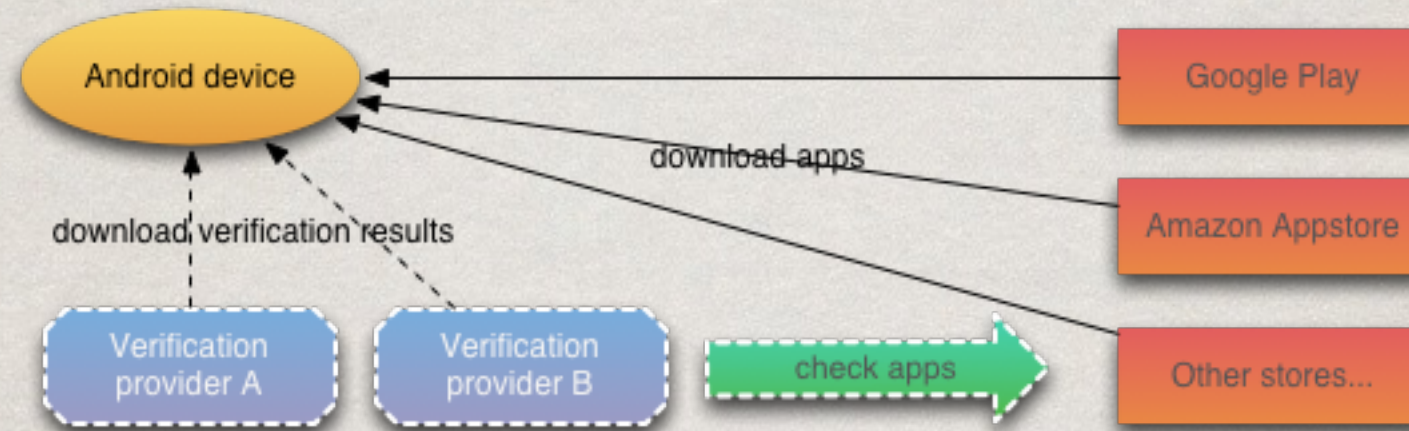


- Trusted entities (e.g. app stores) publish whitelists

  - Comparable to code signatures

  - Users can choose from different whitelist providers

- Code is checked against whitelist before execution

- Prevents all exploits mentioned before

# Implementation



- Based on standard Android 4.3

- Modification of DVM

  - Reminder: DVM executes Java code for apps

  - Apps have to ask DVM to load external code

  - DVM processes keep shared whitelist in memory

  - Negligible performance penalty

- Problem: native code (more later)

# Limitations and Future Work

# Native code

- Cannot control loading in native code

- Prohibiting native code entirely is not an option

- Idea: adapt Google Native Client

  - Sandbox for running native code in browsers

  - Available for ARM architecture

  - Restrict native code, so that malicious external native code is not a problem

  - Subject to ongoing research...

# Practicality

❋ Modification of the Android system

   ❋ Requires update or reinstallation

   ❋ Realistically only deployable to new devices

   ❋ For ideal distribution Google has to approve

❋ Verification providers

   ❋ Stores already check every single app

   ❋ Adding checks of external code is feasible

# Conclusion

# Conclusion

* Large-scale study on external code-loading in benign and malicious Android apps

* 9.25% of popular benign apps are vulnerable, millions of users at risk

* Malicious apps can evade detection

* Proposed a flexible protection scheme

# Thank You! Questions?