Grab’n Run

Secure and Practical
Dynamic Code Loading for Android Applications

Luca Falsina,
Yanick Fratantonio,
Stefano Zanero,
Christopher Kruegel,
Giovanni Vigna,
Federico Maggi
Dynamic code loading (DCL)

- Android system allows applications to load additional code from external sources at runtime.
- Sources can come from either:
  - a local code container stored on the device.
  - a remote URL location (first fetch, then execute the code).

- Minimize code memory footprint
- Flexible code reuse (one library - many apps)
- Live code update during execution
How to perform DCL in Android?

There are several ways to load code dynamically in Android (Class Loaders, Context.createPackageContext, System.exec).

In this presentation I will just focus on Class Loaders:

- Java objects allowing programs to load additional classes.

A ClassLoader implementation that operates on a list of files and directories in the local file system.

A ClassLoader that loads classes from JAR and APK containers with a classes.dex entry.
Typical remote DCL workflow using DexClassLoader API

1. Fetch `myLibrary-dex.jar` from a remote URL

2. Store `myLibrary-dex.jar` on the phone storage.

3. Prepare a directory for the optimized version of the loaded `dex` files.

4. Instantiate `DexClassLoader` object:

   ```java
   DexClassLoader mDexClassLoader = new DexClassLoader(
       myLibJarPath, 
       dexOutputDir.getAbsolutePath(), 
       null, 
       getClass().getClassLoader());
   ```

5. Load "com.example.ClassA" class (by looking at the optimized `dex` files).

   ```java
   Class<?> loadedClass = mDexClassLoader.loadClass("com.example.ClassA");
   ```
How to perform DCL badly in Android?

Here are the mistakes that a benign developer may introduce and make DCL completely insecure:

1. Developer fails to retrieve code in a safe way.
2. Developer fails to store code...
   a. ...by saving the container in a modifiable location.
   b. ...by saving cached dex classes in a modifiable location.
1. Developer fails to **retrieve** code in a **safe** way

1. Fetch *myLibrary-dex.jar* from a **remote** URL using **HTTP**

2. Store *myLibrary-dex.jar* on the phone storage.

3. Prepare a directory for the optimized version of the loaded *dex* files.

4. Instantiate *DexClassLoader* object:

   ```java
   DexClassLoader mDexClassLoader = new DexClassLoader(  
       myLibJarPath,  
       dexOutputDir.getAbsolutePath(),  
       null,  
       getClass().getClassLoader());
   ```

5. Load "com.example.ClassA" class (by looking at the **optimized** *dex* files)

   ```java
   Class<?> loadedClass = mDexClassLoader.loadClass("com.example.ClassA");
   ```
2a. Developer fails to **store** code in a **safe way**
..by **saving the container** in a **modifiable** location

---

1. Fetch *myLibrary-dex.jar* from a remote URL using HTTPS properly.

2. Store *myLibrary-dex.jar* on the phone storage in a *world-writable location* (i.e. external storage).

3. Prepare a directory for the optimized version of the loaded *dex* files.

4. Instantiate *DexClassLoader* object:

   ```java
   DexClassLoader mDexClassLoader = new DexClassLoader(
       myLibJarPath,
       dexOutputDir.getAbsolutePath(),
       null,
       getClass().getClassLoader());
   ```

5. Load “com.example.ClassA” class (by looking at the *optimized dex* files)

   ```java
   Class<?> loadedClass = mDexClassLoader.loadClass("com.example.ClassA");
   ```

---

Overwrite JAR before loading
2b. Developer fails to **store** code in a **safe way**
..by saving cached **dex** in a **modifiable** location

---

**Developer App**

1. Fetch *myLibrary-dex.jar* from a **remote** URL using **HTTPS** properly.

2. Store *myLibrary-dex.jar* on the phone storage **in an application-private location**.

3. Prepare a directory **in a world-writable location** for the optimized **dex** files.

4. Instantiate *DexClassLoader* object:

   ```java
   DexClassLoader mDexClassloader = new DexClassLoader(
       myLibJarPath, 
       dexOutputDir.getAbsolutePath(), 
       null, 
       getClass().getClassLoader());
   ```

5. Load "com.example.ClassA" class (by looking at the **optimized** **dex** files)

   ```java
   Class<?> loadedClass = mDexClassLoader.loadClass("com.example.ClassA");
   ```

---

**Code Injection**

**Overwrite DEX classes before loading**
How frequent are these errors?

Poeplau et al. prove that DCL is heavily used in Android and that implementing DCL in a secure way is indeed challenging.

Considering the top 50 free applications in August 2013:

- 31 (62%) of them make use of DCL
- 17 (34%) of them make use of ClassLoaders
- 8 (16%) of them were flagged as vulnerable

DynamicLoadApk, an open-source project to simplify the use of DCL (over 2000 stars, and 1000 forks on GitHub).

- Source containers are loaded from external storage.
- No integrity or authentication checks on containers.
Threat model

Attacker’s goal is *executing arbitrary code within the context of a target benign application*. 

Attacker can exploit *vulnerabilities* in:

- **Device**. App execution with non-root privileges, read/write access on storage
- **Network communication channel**. MITM on all unencrypted connection
- **Remote server**. We assume the attacker is not able to compromise at least *one server*
Goals

1. Design a **code-verification protocol** suitable to secure DCL.
2. Implement it as a *practical*, developer-friendly **Java library** to replace native API.
3. **Tradeoff** between efficiency, security, and simplicity.
4. Help developers to **migrate** existing applications effortlessly from the native API for DCL to the proposed one.
Workflow to explain our code verification protocol

1. Fetch a **signed** copy of `myLibrary-dex.jar` from a **remote** URL (HTTP / HTTPS).

2. Store `myLibrary-dex.jar` on the phone storage in an **application-private dir**.

3. Save the optimized version of the loaded **dex** files in an **application-private dir**.

4. Retrieve developer certificate via **HTTPS (TRUSTED ELEMENT)** and store it in an **application-private dir**.

5. Verify the signatures of all the file entries of `myLibrary-dex.jar` against the **trusted certificate**.

Load "com.example.ClassA" class.

Discard `myLibrary-dex.jar` copy in the application-private folder.
Given a certain container (i.e., APK, JAR), it is possible to retrieve the package name(s) associated to its classes.

Link package name to the trusted certificate and the container location.

How to associate containers and certificates?

Package name

Trusted certificate

Provided by the developer

Container file path

Dev Cert

Compare container against trusted certificate

Automatically generated

Given a certain container (i.e., APK, JAR), it is possible to retrieve the package name(s) associated to its classes.
## Features’ Comparison Table

<table>
<thead>
<tr>
<th>DexClassLoader (Native API)</th>
<th>Features</th>
<th>SecureDexClassLoader (GNR API)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fetch code from remote URL</td>
<td><img src="checkmark.png" alt="Yes" /></td>
</tr>
<tr>
<td></td>
<td>Store code in app-private location</td>
<td><img src="checkmark.png" alt="Yes" /></td>
</tr>
<tr>
<td></td>
<td>Container verification <em>(integrity and developer authentication)</em></td>
<td><img src="checkmark.png" alt="Yes" /></td>
</tr>
<tr>
<td></td>
<td>Dynamic code loading</td>
<td><img src="checkmark.png" alt="Yes" /></td>
</tr>
</tbody>
</table>
SecureDexClassLoader API

Listing 1: DexClassLoader code snippet.

```java
/*
 * Omitted steps, as they are ‘‘up to the developer’’:
 * 
 * - retrieval of the code container
 * - storage of the container to jarContainerPath
 * - creation of the dexOutputDirPath
 */
DexClassLoader loader = new DexClassLoader(
    jarContainerPath, dexOutputDirPath, null, getClassLoader());

Class<?> klass = loader.loadClass("com.example.MyClass");
MyClass obj = (MyClass) klass.newInstance();
```

Listing 2: SecureDexClassLoader code snippet.

```java
Map<String, URL> pToCert = new HashMap<String, URL>();
pToCert.put("com.foo", new URL("https://bar.com/cert.pem");

SecureLoaderFactory factory = new SecureLoaderFactory(this);
SecureDexClassLoader loader = factory.createDexClassLoader(  
    "http://something.com/dev/exampleJar.jar",  
    null, getClassLoader(), pToCert);

Class<?> klass = loader.loadClass("com.example.MyClass");

if (klass != null) // Is signature valid?
    MyClass obj = (MyClass) klass.newInstance();
```
Grab’n Run: Release

- We released *Grab’n Run* as an **open-source** library on Github.
- The project is online from *November 2014* at [www.grabnrun.org](http://www.grabnrun.org) and it is receiving attention from the community of the Android developers :)
- At the current state, it wraps and enhances *DexClassLoader* API.
We measured the execution time of a simple profiling app using both *DexClassLoader* and GNR API for local DCL. We collected timestamps of 100 executions of this app on a Nexus 5.

Performance overhead

<table>
<thead>
<tr>
<th>Phase</th>
<th>Mean [ms]</th>
<th>Median [ms]</th>
<th>Std Deviation [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DexClassLoader [Total Time]</td>
<td>334.20</td>
<td>332.00</td>
<td>23.50</td>
</tr>
<tr>
<td>-- Setup</td>
<td>331.91</td>
<td>330.00</td>
<td>23.60</td>
</tr>
<tr>
<td>-- First Load Operation</td>
<td>1.55</td>
<td>1.00</td>
<td>0.70</td>
</tr>
<tr>
<td>-- Second Load Operation</td>
<td>0.30</td>
<td>0.00</td>
<td>0.46</td>
</tr>
<tr>
<td>SecureDexClassLoader [Total Time]</td>
<td>1,386.13</td>
<td>1,237.00</td>
<td>322.05</td>
</tr>
<tr>
<td>-- Setup</td>
<td>1,384.13</td>
<td>1,234.00</td>
<td>322.09</td>
</tr>
<tr>
<td>-- Fetch Remote Certificate</td>
<td>972.32</td>
<td>822.00</td>
<td>321.32</td>
</tr>
<tr>
<td>-- Verify Signature</td>
<td>86.17</td>
<td>82.00</td>
<td>18.22</td>
</tr>
<tr>
<td>-- First Load Operation</td>
<td>1.32</td>
<td>1.00</td>
<td>0.66</td>
</tr>
<tr>
<td>-- Second Load Operation</td>
<td>0.39</td>
<td>0.00</td>
<td>0.60</td>
</tr>
</tbody>
</table>
Performance overhead

\[ \text{Slowdown} = \frac{\text{Time of execution with GNR}}{\text{Time of execution with DexClassLoader}} \]
User study

We performed a user study on 12 Android developers.

Starting from a skeleton application, developers had to fetch a remote code container, store it on the phone, and load dynamically a class inside of it.

- **Phase 1**: Implement it using *DexClassLoader* API.
- **Phase 2**: Implement it using GNR API (*).
- **Phase 3**: Send their code and fill in a comparative survey on the two solutions.

(*) Not required to setup the certificate used to validate the code, nor the endpoint to store it.
Developers could consult any online resources (including the *official* Android documentation).

We explicitly asked the participants to treat this experiment as if they were adding a functionality to their own, very popular real application with millions of users.

<table>
<thead>
<tr>
<th>Error (<em>Triggering example</em>)</th>
<th>% developers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch code in an unsafe way</td>
<td>75.0% (9/12)</td>
</tr>
<tr>
<td><em>(Use HTTP connection instead of HTTPS)</em></td>
<td></td>
</tr>
<tr>
<td>Store code in a world-writable area</td>
<td>50.0% (6/12)</td>
</tr>
<tr>
<td><em>(Save code container on external storage)</em></td>
<td></td>
</tr>
<tr>
<td>Store code in a world-writable area</td>
<td>0.0% (0/12)</td>
</tr>
<tr>
<td><em>(Wrongly initialize optimized cache folder)</em></td>
<td></td>
</tr>
<tr>
<td>Miss or fail to implement security checks</td>
<td>100.0% (12/12)</td>
</tr>
<tr>
<td><em>(Do not implement any custom integrity check)</em></td>
<td></td>
</tr>
</tbody>
</table>
# Users evaluation: DexClassLoader (Native API) vs SecureClassLoader (Grab ’n Run API)

## Average time for development

Average of the times in minutes that each developer declared as required to implement the DCL functionality:

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using DexClassLoader (Native API)</td>
<td>139</td>
</tr>
<tr>
<td>Using SecureClassLoader (Grab ’n Run API)</td>
<td>37</td>
</tr>
</tbody>
</table>

## Final evaluation of DexClassLoader

*Please provide an average mark on your satisfaction after having used DexClassLoader (Native API).*

<table>
<thead>
<tr>
<th>Score</th>
<th>Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 or 2 (Disappointing)</td>
<td>6/12</td>
</tr>
<tr>
<td>3</td>
<td>5/12</td>
</tr>
<tr>
<td>4 or 5 (Excellent)</td>
<td>1/12</td>
</tr>
</tbody>
</table>

## Final evaluation of SecureClassLoader

*Please provide an average mark on your satisfaction after having used SecureClassLoader (Grab ’n Run API).*

<table>
<thead>
<tr>
<th>Score</th>
<th>Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 or 2 (Disappointing)</td>
<td>0/12</td>
</tr>
<tr>
<td>3</td>
<td>0/12</td>
</tr>
<tr>
<td>4 or 5 (Excellent)</td>
<td>12/12</td>
</tr>
</tbody>
</table>

## Grab ’n Run learning overhead

*Please quantify the effort in learning how to use Grab ’n Run API over the Native API (i.e., DexClassLoader).*

<table>
<thead>
<tr>
<th>Score</th>
<th>Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 or 2 (Almost zero)</td>
<td>12/12</td>
</tr>
<tr>
<td>3</td>
<td>0/12</td>
</tr>
<tr>
<td>4 or 5 (Extremely broad)</td>
<td>0/12</td>
</tr>
</tbody>
</table>
### Easy to implement

*Look at the two applications you prepared, which one between the two was easier to implement?*

<table>
<thead>
<tr>
<th>Option</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>First application (DexClassLoader) was easier to implement.</td>
<td>0/12</td>
</tr>
<tr>
<td>Second application (SecureDexClassLoader) was easier to implement.</td>
<td>11/12</td>
</tr>
<tr>
<td>Both of them were too difficult.</td>
<td>1/12</td>
</tr>
</tbody>
</table>

### Readability

*Look at the two snippets of code you implemented for the applications, which one is easier to read and understand at a first glance?*

<table>
<thead>
<tr>
<th>Option</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>First application (DexClassLoader) is way easier to read.</td>
<td>0/12</td>
</tr>
<tr>
<td>Second application (SecureDexClassLoader) is way easier to read.</td>
<td>10/12</td>
</tr>
<tr>
<td>They are more or less equally easy to understand.</td>
<td>2/12</td>
</tr>
<tr>
<td>Both of them are difficult to read.</td>
<td>0/12</td>
</tr>
</tbody>
</table>

### Flexibility

*Between the two analyzed solutions, which one do you think offers more flexibility and features for your Android applications?*

<table>
<thead>
<tr>
<th>Option</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>DexClassLoader (Native API).</td>
<td>1/12</td>
</tr>
<tr>
<td>SecureDexClassLoader (Grab 'n Run API).</td>
<td>11/12</td>
</tr>
</tbody>
</table>

### Code maintainability

*If you decide to change the remote location of the APK used as source for DCL, or if you plan to perform DCL from a second remote APK, stored at a different URL, which one of the two applications would be easier to fix?*

<table>
<thead>
<tr>
<th>Option</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>First application (DexClassLoader) would be easier to fix.</td>
<td>1/12</td>
</tr>
<tr>
<td>Second application (SecureDexClassLoader) would be easier to fix.</td>
<td>10/12</td>
</tr>
<tr>
<td>The amount of work would be exactly the same for both of them.</td>
<td>1/12</td>
</tr>
</tbody>
</table>

### Security

*Which one between the two applications do you think is more secure?*

<table>
<thead>
<tr>
<th>Option</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>First application (DexClassLoader) is more secure.</td>
<td>0/12</td>
</tr>
<tr>
<td>Second application (SecureDexClassLoader) is more secure.</td>
<td>11/12</td>
</tr>
<tr>
<td>They are the same, security-wise.</td>
<td>1/12</td>
</tr>
</tbody>
</table>
An easy way for developers to port existing applications to use GNR secure API.

Prototype implemented as a Python script that relies on Androguard and apktool.

Input:
- An APK container to be patched
- The code containers used as sources for DCL
- The binding between each container and the trusted certificate that must be used for its signature verification

Output:
★ A fully-working patched APK that makes use of GNR API instead of native DexClassLoader one.
Limitations

- **Binding containers to certificate through package name**
  - Good assumption with APK (unique).
  - Less suitable for JAR (a package name may be present in more than one JAR...).

- **Limited API coverage**
  - Some API could be included easily (e.g., PathClassLoader)
  - Some other would require new ad-hoc solutions

- **Challenges in code reuse**
  - Our protocol imports copies of container in app-private folders..
Future works

1. Optimize **GNR**.
   a. Improve performance by adding extra caching strategies.
   b. Extend GNR to wrap other native API for DCL.

2. Refine and widen the evaluation of the **repackaging tool** (e.g., perform a user study asking developers to patch their own apps with our script).
Conclusions

1. Propose and design a novel code verification protocol to prevent unsafe implementations of DCL
2. Implement this approach and release it as an open-source library
3. Evaluate extensively our work through a user study with 12 participants
4. Design and implement the repackaging tool to patch apps to use our newly-developed library

Thanks for your attention! Check out Grab’n Run: www.grabnrun.org
Some bonus slides... Only for the braves ;}
Simple use cases of DCL

Silent updates

- Keep updated third-party libraries used by an app (both on features and security).
- Decoupling updates on the application from updates of non-standalone libraries.

Extensibility

- In games or premium version of applications
- The main app downloads an external APK with the extra features purchased by the user and load this code.
How to perform DCL in Android?

2. Package contexts:
   a. Whenever Android loads an app, it associates it with a Context object.
   b. It is also possible to create contexts for other apps (identified by package name) to load not only another app resources but even its classes -> Code Injection Party :(

   ```java
   public abstract Context createPackageContext (String packageName, int flags)
   ```
   - **packageName**: Name of the application’s package.
   - **flags**: Option flags, one of CONTEXT_INCLUDE_CODE or CONTEXT_IGNORE_SECURITY.
   - **Returns**: A Context for the application.

3. Native code, Runtime.exec..
# DexClassLoader Android API

## Public Constructors

```java
public DexClassLoader (String dexPath, String optimizedDirectory, String libraryPath, ClassLoader parent)
```

Creates a `DexClassLoader` that finds interpreted and native code. Interpreted classes are found in a set of DEX files contained in Jar or APK files.

The path lists are separated using the character specified by the `path.separator` system property, which defaults to `:`.

**Parameters**

- `dexPath`: the list of jar/apk files containing classes and resources, delimited by `File.pathSeparator`, which defaults to `:` on Android.
- `optimizedDirectory`: directory where optimized dex files should be written; must not be `null`.
- `libraryPath`: the list of directories containing native libraries, delimited by `File.pathSeparator`, may be `null`.
- `parent`: the parent class loader.

## Public Methods

```java
public Class<?> loadClass (String className)
```

Loads the class with the specified name. Invoking this method is equivalent to calling `loadClass(className, false)`.

**Note:** In the Android reference implementation, the second parameter of `loadClass(String, boolean)` is ignored anyway.

**Parameters**

- `className`: the name of the class to look for.

**Returns**

the `Class` object.

**Throws**

`ClassNotFoundException` if the class cannot be found.

---

**Full class name** = Package name + Simple class name

= `com.example + ClassA`

= `com.example.ClassA`
DexClassLoader Android API: simple code example

```java
ClassA classAInstance = null;
String myLibJarPath = getDir("containers", MODE_PRIVATE).getAbsolutePath() + "/myLibrary-dex.jar";
File dexOutputDir = getDir("dex", MODE_PRIVATE);

DexClassLoader mDexClassLoader = new DexClassLoader(myLibJarPath, dexOutputDir.getAbsolutePath(), null, getClass().getClassLoader());

try {
    Class<?> loadedClass = mDexClassLoader.loadClass("com.example.ClassA");
    classAInstance = (ClassA) loadedClass.newInstance();

    // Do something with the loaded object classAInstance
    // i.e. classAInstance.doSomething();
}

catch (ClassNotFoundException e) {
    e.printStackTrace();
}

catch (InstantiationException e) {
    e.printStackTrace();
}

catch (IllegalAccessException e) {
    e.printStackTrace();
}
```

Full class name = Package name + Simple class name
= com.example + ClassA
= com.example.ClassA
classes.dex... How to add it to a library?

```
$ dx --dex --output=myLibrary-dex.jar myLibrary.jar
processing archive myLibrary.jar...
processing META-INF/MANIFEST.MF...
processing com/example/ClassA.class...
processing com/example/ClassB.class...
processing .classpath...
processing .project...
processing .settings/org.eclipse.jdt.core.prefs...
writing .classpath; size 295...
writing .project; size 368...
writing .settings/org.eclipse.jdt.core.prefs; size 587.
writing classes.dex; size 884...
```

When you export an APK container from the ADT or Android Studio, **dx is called and so translation from Java Bytecode (.class) to Dalvik Bytecode (.dex) is automatically performed.**

```
$ jar xvf ../myLibrary.jar
$ tree
com
  example
    ClassA.class
    ClassB.class
META-INF
  MANIFEST.MF
3 directories, 3 files
```

```
$ jar xvf ../myLibrary-dex.jar
$ tree
.
  classes.dex
  META-INF
    MANIFEST.MF
1 directory, 2 files
```
Package name: what and where?

“A package is a grouping of related classes, interfaces and enumerations providing access protection and name space management.”

- Every Java class is associated to a package (its name is recalled in the first line of the class).
Package name in APK containers

- APK containers must have just **ONE** package name (chosen by the developer when a new application is created).
- Package name is stored in the *Android Manifest*.

![AndroidManifest.xml](image)

- It must be **unique** and must **not change** during the whole lifecycle of the app (Penalty: no admission to *Google Play Store*).
- It can be found easily by querying a *PackageManager* object:

```java
PackageManager.getPackageArchiveInfo(archiveFilePath, 0).packageName
```
Every JAR container can have many package names.

Potentially \( n \) classes may imply \( n \) package names.

No straightforward approach to recover them

- In this scenario a convenient solution is parsing `classes.dex` file and from class entries extract package names.

![Diagram showing class entries and JAR package names](image)
Caching of signature verifications

Jar to be verified

JAR Package Names

- com.example
- com.example.system
- it.polimi.test

Already Checked Package Names Set

- com.example
- com.example.system
- it.polimi.test

loadClass("com.example.ClassA")

This package name is not in the set yet

Full Signature Verification on the JAR

Load class com.example.ClassA

loadClass("it.polimi.test.ClassB")

This package name is already in the set

Load class it.polimi.test.ClassB
Grab’n Run (GNR)

- GNR is a **simple** Java library compatible with any Android project to secure DCL operations.
- GNR implements our *code verification protocol*.
- GNR solves all the presented **issues** and it grants also some extra features.
SecureDexClassLoader API (1)

To instantiate this object you need to call a method on a helper class called `SecureLoaderFactory`:

```java
public SecureDexClassLoader createDexClassLoader(String dexPath, String libraryPath, ClassLoader parent, Map<String, URL> packageNameToCertificateMap)
```

**Parameters:**

- `dexPath` - the list of jar/apk files containing **classes and resources**; these paths could be either local URLs pointing to a location in the device or URLs that links to a **resource** stored in the web via HTTP/HTTPS.
- `packageNameToCertificateMap` - a map that couples each **package name** to a **remote URL** which contains the **certificate** that must be used to validate all the classes that belong to that package before loading them at run time.

**Returns:**

- a `SecureDexClassLoader` object which can be used to **load dynamic code securely**.
SecureDexClassLoader API (2)

After having obtained a reference, you can invoke the usual `loadClass` method on the object:

```
public Class<?> loadClass(String className)
throws ClassNotFoundException
```

**Parameters:**
- `className` - the full class name to load.

**Returns:**
- either a `class` to load at runtime accordingly to `className` if the verification process succeeds or a `null` pointer in case that at least one of the security constraints for secure dynamic class loading is violated.

**Throws:**
- `ClassNotFoundException` - this exception is raised whenever no security constraint is violated but the target class is not found in any of the available source containers.
Comparison among code snippets

`jarContainerPath` contains the path to the target container for class loading.

P.S. No exception handling is shown in these snippets because this code is exactly the same for both solutions.
Hey! Hold on a sec.. Damn!

When you design systems for **mobile devices**, you must take into account **bad/missing Internet connectivity**.

- **Performance mantra:** “**Caching is the way..**”.
  Remember and apply this also to **remote containers** and **certificates**.
- **Outcome:** GNR can handle situations where the smartphone is **offline**.
Caching for containers (APK/JAR)

While SecureLoaderFactory is parsing dexPath string variable, each substring falls in one of these two cases:

1. Local URL Path:
   - Check if a container exists at URL location
   - Compute its digest
   - Look in the private-app caching container folder for a container with name = digest + (.jar/.apk)
   - Import the container at URL Local Path into the app private container folder
   - Use the cached version of the target container
   - Skip this URL
   - Rename the imported container as name = digest + (.jar/.apk)
Caching for containers (APK/JAR)

2. Remote URL Path:

- Try to **download** the file at Remote URL
- Check in the helper file if Remote URL exists
- Compute its **digest**
- Look in the *private-app caching container* folder for a container with the **name** associated to the URL
- Check that the cached container is **fresh enough** with timestamp
- Use the **cached** version of the target container
- **Import** the container into the *app private container* folder
- Rename the imported container as **name = digest + (.jar/.apk)**
- Add an entry to the helper file

<table>
<thead>
<tr>
<th>Remote URL</th>
<th>Cached cont name [Digest + (.jar/.apk)]</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Helper File Structure**

- **Remote URL**
- **Cached cont name** [Digest + (.jar/.apk)]
- **Timestamp**

**Add an entry** to the helper file
Interesting scenario/use case:
Automatic silent updates for remote libraries

Through **SecureDexClassLoader** it is possible to handle this scenario in an easy *and secure* way.

She stores on the web.. A **redirect HTTP** link which points to the **latest version** of **myLibrary-dex.jar**, signed with the developer private key.

http://myLibrary.com/downloads/mobile/latest

An **HTTPS** link which points to the **developer certificate**, stored in a safe location under the dev domain.

https://myLibrary.com/developerCert.pem
Interesting scenario/use case:

Automatic silent updates for remote libraries

1. Create a `SecureDexClassLoader` instance with the `redirect HTTP link` as `dexPath` and the `map package names` pointing to the `developer certificate`.

2. Load the `target classes` from the latest `myLibrary.jar version` :)

---

```java
Map<String, URL> packageNamesToCertMap = new HashMap<String, URL>();
packageNamesToCertMap.put("com.example", new URL("https://myLibrary.com/developerCert.pem"));

SecureLoaderFactory mSecureLoaderFactory = new SecureLoaderFactory(this);
SecureDexClassLoader mSecureDexClassLoader = mSecureLoaderFactory.createDexClassLoader( "http://myLibrary.com/downloads/mobile/latest" null,
packageNamesToCertMap,
getClass().getClassLoader());

Class<?> loadedClass = mSecureDexClassLoader.loadClass("com.example.ClassA");
```

Et voilá! Done..

It was pretty straightforward for both actors, wasn’t it?
Benefits from SecureDexClassLoader

- This API grants both integrity and developer authentication on each used container.
- It is able to hide and manage all the security issues shown in DexClassLoader.
- SecureDexClassLoader is able to manage both remote and local containers.
- It is possible to load concurrently many classes (GNR is thread safe).
- Performance overhead is mitigated thanks to a caching systems on containers, certificates and previous signature verification on containers.
- Caching system makes possible to partially use the library also when the mobile is offline.
Performance overhead

We measured timestamps of a simple profiling app using both DexClassLoader and GNR API. We executed it on a Nexus 5 for 100 times.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Mean [ms]</th>
<th>Median [ms]</th>
<th>Std Deviation [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DexClassLoader [Total Time]</td>
<td>334.20</td>
<td>332.00</td>
<td>23.50</td>
</tr>
<tr>
<td>Setup</td>
<td>331.91</td>
<td>330.00</td>
<td>23.60</td>
</tr>
<tr>
<td>First Load Operation</td>
<td>1.55</td>
<td>1.00</td>
<td>0.70</td>
</tr>
<tr>
<td>Second Load Operation</td>
<td>0.30</td>
<td>0.00</td>
<td>0.46</td>
</tr>
<tr>
<td>SecureDexClassLoader [Total Time]</td>
<td>1,386.13</td>
<td>1,237.00</td>
<td>322.05</td>
</tr>
<tr>
<td>Setup</td>
<td>1,384.13</td>
<td>1,234.00</td>
<td>322.09</td>
</tr>
<tr>
<td>Fetch Remote Certificate</td>
<td>972.32</td>
<td>822.00</td>
<td>321.32</td>
</tr>
<tr>
<td>Verify Signature</td>
<td>86.17</td>
<td>82.00</td>
<td>18.22</td>
</tr>
<tr>
<td>First Load Operation</td>
<td>1.32</td>
<td>1.00</td>
<td>0.66</td>
</tr>
<tr>
<td>Second Load Operation</td>
<td>0.39</td>
<td>0.00</td>
<td>0.60</td>
</tr>
</tbody>
</table>
Performance overhead

We measured timestamps of a simple profiling app using both DexClassLoader and GNR API. We executed it on a Nexus 5 for 100 times.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Mean [ms]</th>
<th>Median [ms]</th>
<th>Std Deviation [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DexClassLoader [Total Time]</td>
<td>6.28</td>
<td>6.00</td>
<td>1.39</td>
</tr>
<tr>
<td>Setup</td>
<td>3.98</td>
<td>4.00</td>
<td>1.04</td>
</tr>
<tr>
<td>First Load Operation</td>
<td>1.42</td>
<td>1.00</td>
<td>0.70</td>
</tr>
<tr>
<td>Second Load Operation</td>
<td>0.44</td>
<td>0.00</td>
<td>0.61</td>
</tr>
<tr>
<td>SecureDexClassLoader [Total Time]</td>
<td>90.42</td>
<td>90.00</td>
<td>8.73</td>
</tr>
<tr>
<td>Setup</td>
<td>88.25</td>
<td>87.50</td>
<td>8.55</td>
</tr>
<tr>
<td>Verify Signature</td>
<td>61.39</td>
<td>61.00</td>
<td>6.79</td>
</tr>
<tr>
<td>First Load Operation</td>
<td>1.33</td>
<td>1.00</td>
<td>0.55</td>
</tr>
<tr>
<td>Second Load Operation</td>
<td>0.49</td>
<td>0.00</td>
<td>0.77</td>
</tr>
</tbody>
</table>

Cached Resource
1. No DEX translation
2. No certificate fetch
We introduced several **caching improvements** to speedup GNR. Thanks to these strategies:

1. Each **remote** code container and certificate is *fetched only once* across the different executions.

2. Each **local** code container is *imported only once* into the application-private folder.

3. The **signature** of each code container is *checked only once* and the result of the verification is propagated to all the package names in that container.
## Performance overhead - No Cache

<table>
<thead>
<tr>
<th>Statistic Measure</th>
<th>Mean</th>
<th>Median</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DexClassLoader</td>
<td>334.20</td>
<td>332.00</td>
<td>23.50</td>
</tr>
<tr>
<td>DexClassLoader Setup</td>
<td>331.91</td>
<td>330.00</td>
<td>23.60</td>
</tr>
<tr>
<td>DexClassLoader Load Operation</td>
<td>1.55</td>
<td>1.00</td>
<td>0.70</td>
</tr>
<tr>
<td>DexClassLoader Second Load Operation</td>
<td>0.30</td>
<td>0.00</td>
<td>0.46</td>
</tr>
<tr>
<td>SecureDexClassLoader</td>
<td>1386.13</td>
<td>1237.00</td>
<td>322.05</td>
</tr>
<tr>
<td>SecureDexFactory Preparation</td>
<td>1384.13</td>
<td>1234.00</td>
<td>322.09</td>
</tr>
<tr>
<td>Download Container</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Download Certificate</td>
<td>972.32</td>
<td>822.00</td>
<td>321.32</td>
</tr>
<tr>
<td>Verify Signature</td>
<td>86.17</td>
<td>82.00</td>
<td>18.22</td>
</tr>
<tr>
<td>Load Operation</td>
<td>1.32</td>
<td>1.00</td>
<td>0.66</td>
</tr>
<tr>
<td>Second Load Operation</td>
<td>0.39</td>
<td>0.00</td>
<td>0.60</td>
</tr>
<tr>
<td>Wipe Cached Data</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>
## Performance overhead - Cache

<table>
<thead>
<tr>
<th>Statistic Measure</th>
<th>Mean</th>
<th>Median</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DexClassLoader</td>
<td>6.28</td>
<td>6.00</td>
<td>1.39</td>
</tr>
<tr>
<td>DexClassLoader Setup</td>
<td>3.98</td>
<td>4.00</td>
<td>1.04</td>
</tr>
<tr>
<td>DexClassLoader Load Operation</td>
<td>1.42</td>
<td>1.00</td>
<td>0.70</td>
</tr>
<tr>
<td>DexClassLoader Second Load Operation</td>
<td>0.44</td>
<td>0.00</td>
<td>0.61</td>
</tr>
<tr>
<td>SecureDexClassLoader</td>
<td>90.42</td>
<td>90.00</td>
<td>8.73</td>
</tr>
<tr>
<td>SecureDexFactory Preparation</td>
<td>88.25</td>
<td>87.50</td>
<td>8.55</td>
</tr>
<tr>
<td>Download Container</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Download Certificate</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Verify Signature</td>
<td>61.39</td>
<td>61.00</td>
<td>6.79</td>
</tr>
<tr>
<td>Load Operation</td>
<td>1.33</td>
<td>1.00</td>
<td>0.55</td>
</tr>
<tr>
<td>Second Load Operation</td>
<td>0.49</td>
<td>0.00</td>
<td>0.77</td>
</tr>
<tr>
<td>Wipe Cached Data</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>
Grab’n Run: User study

We performed a user study on 12 Android developers to verify whether:

1. GNR is easier to use than native DexClassLoader API.
2. GNR learning overhead for a developer is little, once he knows how to use native API.
Repackaging tool: Functioning (1)

1. Analyze APK with Androguard
   - Is a valid APK?
   - Permissions
   - Sensitive points for dynamic code loading

2. Disassemble APK with apktool
   - AndroidManifest.xml
   - Resources
   - Smali Classes

3. Add missing permissions to AndroidManifest.xml
   - AndroidManifest.xml
   - Missing GNR permissions
   - Patched AndroidManifest.xml
Repackaging tool: Functioning (2)

4. Copy GNR files in smali folder of the APK

5. Inject user settings in smali code of RepackHandler class

6. Patch sensitive points in smali classes of the APK

7. Rebuild patched APK

GNR Smali Classes + Smali Classes

RepackHandler Smali Class

Translate user settings into smali code...

..And inject them into the RepackHandler smali class.

Smali Classes

Patch

Sanitized Smali Classes

Patched Android Manifest.xml

Resources

APK TOOL

Patched APK

Smali Classes

Patched APK
Related works in the field

1. Remote Code Execution
   
   [Poeplau et al.] Dalvik VM modification + external service to analyze dynamically loaded code components

2. Code Verification
   
   [Vidas and Christin] Public-key infrastructure backed by the domain name system