Possible ideas to ensure the integrity of the client side app we proposed:

Authors:

Ankur Debnath(111205030)
ankurdebnath1994@gmail.com

Anoma Barua(111205041)
anoma.besu@gmail.com

Sangita Ray(111205059)
sangitar18@gmail.com

Sayak Haldar(111205038)
sayakhaldar5@gmail.com

Supervisor:

Manas Hira
manas@cs.becs.ac.in

12th August,2015

Abstract

We are trying to design an attendance maintaining system which consists of server (teachers or attendance maintainer) side app and client (students or attendance giver) side app.In our previous document, we mention how to authenticate a single user in our map so, we can ensure that only the intended user provides his/her attendance. So that no attendance giver or student can provide his/her attendance via a compromised version of the client app. Now, creating a compromised version of the app requires two thing: reverse engineering and repackaging. There is no way we can protect our app fully(100%) from these processes. Also, there exists open source tools which can help people to do reverse engineering Like: APKtool and Dex2Jar. But, we can slow down these processes by following some techniques. We are going to discuss those techniques throughout the document now.

1 Using Android Market Licensing service tool:

Android Market Licensing service tool is a powerful tool for protecting developed applications against unauthorized use. If we use Google Play Store to publish our app,we can use the Google Play Licensing service. Though the Google Play Licensing service is primarily intended for paid applications that wish to verify that the current user did in fact pay for the application on Google Play, any application (including free apps) may use the licensing service to initiate the download of an APK expansion file. The License Verification Library (LVL) is a key component of it. It is a collection of helper classes that greatly simplify the work that you need to do to add licensing to your application and is available as a downloadable package of the Android SDK.Out of the box, the LVL protects against casual piracy; users who try to copy APKs directly from one device to another without purchasing the application. But it cannot protect you from determined attackers who are willing to disassemble and reassemble code to hack around the service. To make things hard, even for technically skilled attackers who attempt to decompile your application and remove or disable LVL-related code, there are few techniques which can be applied together. We will discuss those techniques now

- Obfuscate one's application to make it difficult to reverse-engineer.
- Modifying the licensing library itself to make it difficult to apply common cracking techniques.
- Making one's application tamper-resistant.
- Offloading license validation to a trusted server.

1.1 Technique:Code Obfuscation:

The first line of defense in our application should be code obfuscation. Code obfuscation will not protect against automated attacks, and it doesn't alter the flow of your program. However, it does make it more difficult for attackers to write the initial attack for an application, by removing symbols that would quickly reveal the original structure of a compiled application. As such, we strongly recommend using code obfuscation in all LVL installations.

Now, what does an obfuscator do? Suppose, an application is compiled and converted into .dex files and packaged in an APK for distribution on devices. The bytecode contains references to the original code — packages, classes, methods, and fields all retain their original (human readable) names in the compiled code. Attackers use this information to help reverse-engineer the program, and ultimately disable the license check.To protect from the attackers obfuscators replace these names with short, machine generated alternatives. Rather than seeing a call to dontAllow(), an attacker would see a call to a(). This makes it more difficult to intuit the purpose of these functions without access to the original source code.

here are a number of commercial and open-source obfuscators available for Java that will work with Android. One of them is ProGuard. However, we could always check for other obfuscators.

1.2 Technique: Modifying the license library:

The second line of defense against attack from crackers is to modify the license verification library in such a way that it's difficult for an attacker to modify the disassembled code and get a positive license check as result.

This actually provides protection against two different types of attack: it protects against attackers trying to crack our application, but it also prevents attacks designed to target other applications (or even the stock LVL distribution itself) from being easily ported over to our application. The goal should be to both increase the complexity of our application's bytecode and make our application's LVL implementation unique.

When modifying the license library, there are three areas that we should focus on:

- The core licensing library logic.
- The entry/exit points of the licensing library.
- How your application invokes the licensing library and handles the license response.

In the case of the core licensing library, we primarily need to focus on two classes which comprise the core of the LVL logic: LicenseChecker and LicenseValidator.

Our goal should be to modify these two classes as much as possible, in any way possible, while still retaining the original function of the application. Here are some ideas to get us started:

- Replacing switch statements with if statements.
- Using XOR or hash functions to derive new values for any constants used and check for those instead.
- Removing unused code. For instance, if we are sure that we won't need swappable policies,we could remove the Policy interface and implement the policy verification inline with the rest of LicenseValidator.
- Moving the entirety of the LVL into our own application's package.
- Spawning additional threads to handle different parts of license validation.
- Replacing functions with inline code where possible.

For example, consider the following function from LicenseValidator:

```
public void verify(PublicKey publicKey, int responseCode, String signed-
Data,String signature)
// ... Response validation code omitted for brevity ...
 switch (responseCode) {
// In Java bytecode, LICENSED will be converted to the constant 0 × 0
case LICENSED:
case LICENSED_OLD_KEY:
LicenseResponse limiterResponse = mDeviceLimiter.isDeviceAllowed(userId);
handleResponse(limiterResponse, data);
break;
// NOT_LICENSED will be converted to the constant 0x × 1
case NOT_LICENSED:
handleResponse( LicenseResponse.NOT_LICENSED, data);
break;
// ... Extra response codes also removed for brevity ...
}
```

In this example, an attacker might try to swap the code belonging to the LICENSED and NOT_LICENSED cases, so that an unlicensed user will be treated as licensed. The integer values for LICENSED (0x0) and NOT_LICENSED (0x1) will be known to an attacker by studying the LVL source, so even obfuscation makes it very easy to locate where this check is performed in your application's bytecode.

To make this more difficult, consider the following modification:

```
public void verify(PublicKey publicKey,int responseCode, String signed-
Data, String signature)
// ... Response validation code omitted for brevity ...
// Compute a derivative version of the response code
// Ideally, this should be placed as far from the responseCode switch as
possible,
// to prevent attackers from noticing the call to the CRC32 library, which
would be
// a strong hint as to what we're done here. If you can add additional
transformations elsewhere in before this value is used, that's even better.
java.util.zip.CRC32 crc32 =new java.util.zip.CRC32();
crc32.update(responseCode);
int transformedResponseCode = crc32.getValue();
// ... put unrelated application code here ...
// crc32(LICENSED) == 3523407757
if (transformedResponse == 3523407757) {
 LicenseResponse limiterResponse = mDeviceLimiter.isDeviceAllowed(userId);
 handleResponse(limiterResponse, data);
} // ... put unrelated application code here ...
// crc32(LICENSED_OLD_KEY) == 1007455905
if (transformedResponseCode == 1007455905) {
 LicenseResponse limiterResponse = mDeviceLimiter.isDeviceAllowed(userId);
 handleResponse(limiterResponse, data);
} // ... put unrelated application code here ...
// crc32(NOT_LICENSED) == 2768625435 if (transformedResponseCode
== 2768625435):
  userIsntLicensed();
}
}
```

In this example, we've added additional code to transform the license response code into a different value. We've also removed the switch block, allowing us to inject unrelated application code between the three license response checks.

Also for the entry/exit points, the attackers may try to write a counterfeit version of the LVL that implements the same public interface, then try to swap out the relevant classes in our application. To prevent this, we could consider adding additional arguments to the LicenseChecker constructor, as well as allow() and dontAllow() in the LicenseCheckerCallback. For example, we could pass in a nonce (a unique value) to LicenseChecker that must also be present when calling allow().

Note: Renaming allow() and dontAllow() won't make a difference, since we are already using an obfuscator. The obfuscator will automatically rename these functions for us. (This is a four step process of securing the integrity of the app and we are now discussing second step)

Attackers also may try and attack the calls in our application to the LVL. For example, if we display a dialogue on license failure with an "Exit" button and some attacker comment out the line that display that window the app might be running forever. We could prevent this , we could invoke a different activity to inform the user that their license is invalid, and immediately terminating the original Activity; we could add additional finish() statements to other parts of our code that get will get executed in case the original one gets disabled; or we could set a timer that will cause your application to be terminated after a timeout. It's also a good idea to defer the license check until your application has been running a few minutes, since attackers will be expecting the license check to occur during your application's launch.

1.3 Technique: Make your application tamper-resistant:

In order for an attacker to remove the LVL from our code, they have to modify our code. Unless done precisely, this can be detected by our code. There are a few approaches you can use here.

The most obvious mechanism is to use a lightweight hash function, such as CRC32, and build a hash of our application's code. We can then compare this checksum with a known good value. We can find the path of your application's files by calling context.GetApplicationInfo() — just be sure not to compute a checksum of the file that contains our checksum! (Consider storing this information on a third-party server.) Also, We can check to see if your application is debuggable. If our application tries to keep itself from performing normally if the debug flag is set, it may be harder for an attacker to compromise. Now, we will discuss some techniques for making our app tamper-resistent.

1.3.1 Verifying app's signing certificate at runtime:

In a nutshell, we, developers must sign applications with our private key/certificate (contained in a .keystore file) before the app can be installed on user devices. The signing certificate must stay consistent throughout the life of the app, and typically have an expiry date of 25 years in the future.

The consistency of the developer signing certificate is relied upon by the Android system when dealing with app upgrades. For example, we could create an app with the same App ID as Facebook, We couldn't trick users into upgrading to my version as it's not signed with Facebook's certificate. As developers, we must keep this certificate private otherwise we risk others being able to sign applications as us.We also should Keep our private key (.keystore file) out of source control and in a separately secured and backed-up system.

The app signature will be broken if the .apk is altered in any way — unsigned apps cannot typically be installed. In order for the altered .apk to be installed, the attacker must resign it. However if this technique alone is implemented then it could easily be broken. We are not going to discuss the techniques to break this approach in the current context.

1.3.2 Verifying the installer:

Each app contains the identifier of the app which installed it. Therefore, with a very simple check you could have your app verify the installer ID.

This example assumes we are only distributing via the Google Play Store:

```
private static final String PLAY_STORE_APP_ID = "com.android.vending";
public static booleanverifyInstaller(final Context context){
final String installer = context.getPackageManager().
getInstallerPackageName(context.getPackageName());
return installer != null&& installer.startsWith(PLAY_STORE_APP_ID);
}
```

The snippet shows getting the InstallerPackageName from the PackageManager and verifying against the known value of the Google Play Store, com.android.vending.

1.3.3 Environment checks:

The final two checks are aimed at assessing the environment the app is running in. Outside of development, it's unlikely that your app should be running on an

emulator, and releasing apps with debuggable enabled is discouraged as it allows connected computers to access and debug the app via the Android Debug Bridge.

1.3.3.1.Emulator:

If your app is being run on an emulator outside the development process, it gives an indication that someone other than you is trying to analyze the app.

These emulator checks look for private or hidden system properties that give a strong indication the system is an emulator. It's not fool-proof — with custom ROMs and rooted devices, these values could be modified.

First we define a helper method which uses Java Reflection to allow us to access these hidden system properties:

```
private static String getSystemProperty(String name)
throws Exception {
Class systemPropertyClazz = Class.forName("android.os.SystemProperties");
return(String) systemPropertyClazz.getMethod("get", new Class[]
{ String.class }).invoke(systemPropertyClazz, new Object[] { name });
```

Now we check ro.hardware, ro.kernel.qemu and ro.product.model properties for known values that emulators use.

```
public static boolean checkEmulator() {
try {
boolean goldfish = getSystemProperty("ro.hardware").contains("goldfish");
boolean emu = getSystemProperty("ro.kernel.qemu").length()  0;
boolean sdk = getSystemProperty("ro.product.model").equals("sdk");
if (emu || goldfish || sdk) {
      return true;
}
} catch (Exception e) {
}
```

This particular check needs to access hidden system properties. Though there are checks which do not require accessing hidden system properties. Some off them

can be found in this stackoverflow link.

1.3.3.2. Debuggable:

Allowing apps to be debugged when installed on an Android device is something that, as developers, we only enable during the development process. Therefore, if debugging occurs on a live build of your app, it's likely that someone other than you is trying to analyze the app.

It's a common first step for attackers to decompile the app, enable the debuggable flag (in the Android Manifest file) and recompile, allowing them to attach the debugger to help better understand how the app works.

Here's a snippet to check for the debuggable flag at runtime:

```
public static boolean checkDebuggable(Context context){
return (context.getApplicationInfo().flags
& ApplicationInfo.FLAG_DEBUGGABLE) != 0;
}
```

The debuggable flag is accessible as one of the flags on the ApplicationInfo object, which is accessible from context. You will likely want to wrap the checkDebuggable() method in if statement to only check on live builds.

2 Watermarking our app:

We could follow some mechanism to create a watermark of our android app. However, There are several key challenges to incorporate dynamic watermarking into current Android app development practice and make it easily deployable by arbitrating parties.

Firstly, the state of the art dynamic Java watermarking techniques need extensive intervention from developers to embed a watermark. For example, SandMark requires developer to manually annotate source code to indicate where watermark can be inserted, and to manually give input to drive the software when embedding a watermark. These manual interventions make it cumbersome to apply this technique in real practice and hard to be made right.

Secondly, it is desirable to have automatic watermark recognition so that it can handle thousands of apps with little human effort in an online and realtime manner. To recover embedded watermarks, SandMark leverages programmable Java Debug Interface to access memory objects on the heap in order

to infer object reference relationships. However, there is no known programmable debugging interface available on Android. Even worse, manually providing input is required for watermark extraction phase as well. Obviously, this cannot scale to handle the large number of watermark recognition requests for thousands of Android apps submitted to current app markets on a daily basis.

Thirdly, Android apps, although mainly developed in Java language, have significant difference from desktop Java software. For example, Android apps depend more heavily on event-driven mechanisms and the underlying execution environment to work correctly. Unlike legacy Java applications that have a single entry point named main, Android apps in general have multiple entry points. Now, there are solutions available for this problem in the documentation of AppInk design. Here, the solutions are given followed by the theory of designing of an app which could watermark our app. But, the problem is it's all theoritical.

3 Problems and challenges:

So, we can see that there's no way we could protect our app from reverse engineering or repackaging 100%. But, we could make the job more tougher for hackers and we discussed several approaches which could be applied altogether as defense lines. But again, there are problems in this approach. Like, there's is a tool named "DASHO". It offers many things to protect one's developed app. Like: it acts as an obfuscator. It also makes an app tamper resistent and also provides the option of watermarking an app. but it is not a free tool. We could use ProGuard (an open source tool) for only Obfuscation part. But then we have to develop the a tool which could do the watermarking job from the given design of AppInk app from almost scratch. We need to put huge effort on these things.

4 References:

1. http://android-developers.blogspot.in/2010/09/securing-android-lvl-applications.html
2. https://www.airpair.com/android/posts/adding-tampering-detection-to-your-android-app
3. https://developer.android.com/tools/publishing/app-signing.html
4. https://www.owasp.org/index.php/Architectural_Principles_That_Prevent_Code_Modification_or_Reverse_Engineering
5. https://www.owasp.org/index.php/Technical_Risks_of_Reverse_Engineering_and_Unauthorized_ Code_Modification#Code_Modification_.2F_Injection_Technical_Risks

6. AppInk documentation

7. http://proguard.sourceforge.net/