

# Rogue Mobile App - Breaking Security Checks

A rogue android mobile app can be created in just a few hours, either from the original app or from scratch. It can capture data store data, change data, reuse data, schedule, transfer, call any APIs, etc. It can also capture all 2FA including OTP, as OTP is received on the same mobile device only.

It can bypass all other security checks very easily. Anti-tampering check is the only way to ensure that application code is not tampered, but it can be broken even if written in C++.

Current Protection	Bypassing
Playstore checks	They check very basis items like permissions, etc. It is impossible for them to detect code tampering. Rogue apps can always be uploaded to external stores/sites, bypassing even these basic checks.
Obfuscation	Obfuscation does not Obfuscate public classes, public methods, public variables, constants, and strings. All stealing and substitution are always done using public objects and strings.
App monitoring service	Upload rogue app on Playstore, marking it private. App monitoring services can't see private apps. Alternatively, an app can be uploaded to a private site. There are many other ways of bypassing his service.
Anti-tampering check	Substitute checksum/hashvalue of RSA signature/classes.dex file from the original app in either java/ de-assembled C++ program of the rogue app.
OTP based authentication	Put code in the rogue app to read SMS and get OTP. Either automatically call confirmation API with OTP or pass this OTP to thief.
Certificate Pinning	But all certificates and corresponding passwords are always hardcoded in the application. Thus, they can be easily obtained from the decompiled original application.
App Hardening	App hardening does not harden Android/java classes. Stealing can be carried out using either static analysis of decrypted code or dynamic analysis using modified Android/java classes, on a rooted device.
Grid /Challenge Based Authentication	Rogue app can capture grid data/challenge questions and store locally. Later on, it can use this data to confirm transactions without the user entering it.
Biometric	Biometrics in the app is just a string. It can be stolen or substituted like any other data.
Location based & IP based Risk Engines	Steal location coordinates along with authentication data in rogue apps and substitute while carrying out fraudulent transaction. IP-based risk engines no good because of changing location.

# Breaking Anti-Tampering Check

Anti-tampering check is the only way to ensure that application code is not tampered with. This code can be written in java or C++. This check is based on either hash value of app signature or checksum value of classes.dex file. This check is either done at the app level or at the server level.

Since both of these properties can be obtained externally from the original app without modifying the app and they are static in nature, these obtained properties can be substituted in the appropriate variables of the modified app to bypass anti-tampering checks.

There are many ways of breaking. This document describes one simple approach.

## Cracking C++ Code – OWASP Recommended Method

JNI C++ code calls underlying java classes. Here reference of class `java.security.MessageDigest` is changed to `my.security.MessageDigest`, Here `my.security.MessageDigest` is class that extends of class `java.security.MessageDigest`. In this extended class the signature of the original app is hardcoded. Thus, C++ code will always get the value of the original app and hence the anti-tampering check will always show no tampering.

Even though C++ library can't be decompiled, it can always be disassembled using a tool like IDA-PRO. The cracking can be done at the assembly code level.

This example shows anti-tampering using the app RSA signature, a most used method. Exactly the same method can be used by extending the `JarFile` or `ZipFile` class and returning the checksum of the original object in the extended class.

### Original Code

```
jobject signatures = (*env)->GetObjectField(env, packageInfo, fid);
jobject signature = (*env)->GetObjectArrayElement(env, signatures, 0);
jclass sigclass = env->GetObjectClass(signature);
jmethodID toarray = env->GetMethodID(sigclass, "toArray", "()IB*");
jobject bytearray = env->CallObjectMethod(signature, toarray);
jclass mdclass = env->FindClass("java/security/MessageDigest");
jmethodID mdgetinsmethod = env->GetStaticMethodID(mdclass, "getInstance", "(Ljava/lang/String;)Ljava/security/MessageDigest;");
jobject mdobj = env->CallStaticObjectMethod(mdclass, mdgetinsmethod, env->NewStringUTF("SHA1"));

.....

.....
```

## Cracked Code

```
object signatures = (*env)->GetObjectField(env, packageInfo, fid);
object signature = (*env)->GetObjectArrayElement(env, signatures, 0);
jclass sigclass = env->GetObjectClass(signature);
jmethodID toarray = env->GetMethodID(sigclass, "toArray", "()IB");
object bytearray = env->CallObjectMethod(signature, toarray);

jclass mdclass = env->FindClass("my/security/MessageDigest");

jmethodID mdgetinsmethod = env->GetStaticMethodID(mdclass, "getInstance", "(Ljava/lang/String;)Ljava/security/MessageDigest");
object mdobj = env->CallStaticObjectMethod(mdclass, mdgetinsmethod, env->NewStringUTF("SHA1"));

....
....
```

## Cracking Java Code – OWASP Recommended Method

Exactly the same as used in cracking JNI C++ code can be adopted here. code can be cracked either at java level or smali level. Again the reference of class java.security.MessageDigest is changed to my.security.MessageDigest, Here my.security.MessageDigest is class that extends of class java.security.MessageDigest. In this extended class the signature of the original app is hardcoded. Thus, java code will always get the value of the original app and hence the anti-tampering check will always show no tampering.

Exactly the same method can be used by extending the JarFile or ZipFile class and returning the checksum of the original object in the extended class.

## Cracking C++ Code using ziplib

C++ ziplib/minizip can be used for decompressing APK file and obtaining hashvalue or checksum of either RSA signature file or classes.dex file.

This can be simply cracked by copying the original apk to the data directory and changing the location reference of the apk file to the data directory by cracking disassembled code using a tool like IDA-PRO.

## Conclusion

If the anti-tampering check is at the app level, then this check can be simply bypassed/omitted, because this check does not prevent exploitation of APIs.

If this check is at the server level, this check can be broken using methods described here.

Once the anti-tampering check is broken, the app can be create/modified any way.