Abusing Android In-app Billing feature thanks to a misunderstood integration

Insomni'hack 18

22/03/2018 – Jérémy MATOS

whois securingapps

- Developer background
- Worked last 12 years in Switzerland on security solutions
  - Focus on mobile since 2010
- Recent OWASP Geneva co-chapter leader
- Freelance application security consultant
- Consulting to build security in software
  - GDPR: No, have a talk with @sadamiste for that
  - Mobile
  - Web
  - Cloud
  - Internet Of Things
  - Bitcoin/Blockchain

@Securingapps

Agenda

- 1. Android In-app billing in a nutshell
- 2. Real-life exploitation in a rather popular game: getting free credits
  - Java reverse engineering
  - Writing a Java hook with Xposed framework
  - Bytecode patching of application and redistribution
- 3. Lessons learned
- 4. Recommendations

1. Android In-app billing in a nutshell

- **Goal: show that Java reverse engineering can cause a loss of value in real-life**
- Target: Android In-app billing feature
- Allow developers to sell content in their app, e.g.
  - subscribtions to magazines
  - premium features
  - extra content in games
- Payment is handled by Google
  - Requires Google Play services
  - No credit card data exposed to developers
- Documentation available at https://developer.android.com/google/play/billing/index.html

1. Android In-app billing in a nutshell

2. Real-life exploitation 1/13

- (Used to be) rather popular game: PandaPop
- In-app purchases to buy credits
  - New weapons
  - Extra lifes
- Step 1: Download the APK archive: e.g. from `apkpure.com`
  - Avoid executing this binary, or in an emulator

2. Real-life exploitation 2/13

- Step 2: Prepare emulator
  - 1. We will use Genymotion emulator
    - Fast (thanks to x86 image)
    - Rooting possible in 1 click
    - Free version available
  - 2. Install OpenGAPPS to have Google Play Services
    - Sign-in with valid a gmail account
    - Install Google Play Games
    - Wait for the various Google applications to be updated

2. Real-life exploitation 3/13

- Step 3: use jadx free tool to convert automatically an APK in readable Java source code
  - 1. converts Dalvik bytecode to Java bytecode
  - 2. decompiles Java bytecode in Java source code
  - 3. displays the results in an IDE for analysis

- Step 4: Look for instances of `IInAppBillingService`
  - This interface cannot be renamed

    Proguard, you must add the following line to your Proguard configuration file:

    ```
    -keep class com.android.vending.billing.**
    ```

  - Nothing is obfuscated! ⚠

2. Real-life exploitation 4/13

- Step 5: Easily review related implementation classes
  - Fascinating code in method `purchaseProduct` of class `com.prime31.GoogleIABPlugin`

```
254         }
254         Log.w("Prime31", "CANNOT fetch sku type due to either inventory not being queried or it returned no valid skus.");
255         this._currentSkuBeingPurchased = sku;
257         if (sku.equalsIgnoreCase("android.test.purchased")) {
258             Log.i("Prime31", "fixing Google bug where they think the sku " + sku + " is a subscription. resetting to type inapp");
            itemType = IabHelper.ITEM_TYPE_INAPP;
            }
```

- Step 6: Find out what `android.test.purchased` means
  - Google is our friend
  - https://developer.android.com/google/play/billing/billing_testing.html

- **android.test.purchased**

  When you make an In-app Billing request with this product ID, Google Play responds as though you successfully purchased an item.

  The response includes a JSON string, which contains fake purchase information (for example, a fake order ID).

2. Real-life exploitation 5/13

- Step 7: force value to `android.test.purchased` and see what happens
  - Let's write a hook that forces sku of `purchaseProduct` to this value

```
210     public void purchaseProduct(final String sku, final String developerPayload) {
```

- Using **Xposed** framework
  - Overload the behavior of an application by intercepting calls in the Dalvik virtual machine
  - No change to the original `apk` file
  - Implement a hook with **Android Studio** in an independent `apk`

2. Real-life exploitation 6/13

- Prerequisites
  - Rooted device to installed the **`Xposed`** libraries
  - Emulator (to avoid smartphone bricking…)
- Install hooking framework in Genymotion device
  - Install terminal application
    - Drag/drop `terminal.apk`
    - Start it and type `su`
    - Check that root access is prompted and validate you are really root on the device
  - Drag/drop `XposedInstaller_3.1.5.apk`
    - Start it and choose install
    - Reboot the Genymotion device

2. Real-life exploitation 7/13

```
public class Tutorial implements IXposedHookLoadPackage {

    public void handleLoadPackage(final LoadPackageParam lpparm) throws Throwable {

        String ourPackageName = "com.sgn.pandapop.gp";
        String ourClassToHook = "com.prime31.GoogleIABPlugin";


        if (!lpparm.packageName.equals(ourPackageName)) {
            return;
        }

        XposedBridge.log("Hooking loaded");

        findAndHookMethod(ourClassToHook, lpparm.classLoader, "purchaseProduct", String.class, String.class, (XC_MethodHook) beforeHookedMethod(param) → {
                String s1 = (String) param.args[0];
                String s2 = (String) param.args[1];
                XposedBridge.log("On purchase product " + s1 + " --- " + s2);
                param.args[0] = "android.test.purchased";
        });

    }
}
```

2. Real-life exploitation 8/13

- Step 8: Deploy hook
  - Build hook apk with Android Studio
  - Copy it to Genymotion device
  - Activate hook in Xposed configuration panel
  - Reboot Genymotion device
  - Enjoy free credits

- **Why does it work ?**
  - According to Google documentation, no signature is returned with this test value so verification should fail
  - The vulnerability is easy to find in the reversed source code

2. Real-life exploitation 9/13

com.prime31.GoogleIABProxyActivity | com.prime31.GoogleIABPluginBase | com.prime31.util.IabHelperImpl

```
358         return true;
359     }
360     try {
361         Purchase purchase = new Purchase(this.mPurchasingItemType, purchaseData, dataSignature);
362         try {
363             String sku = purchase.getSku();
364             if (this.mPurchaseListener != null) {
365                 this.mPurchaseListener.onIabPurchaseCompleteAwaitingVerification(purchaseData, dataSignature);
366             }
367             if (!autoVerifySignatures || purchase.isTestSku() || Security.verifyPurchase(this.mSignatureBase64, purchaseData, dataSignature)) {
368                 logDebug("Purchase signature successfully verified.");
369                 if (this.mPurchaseListener != null) {
370                     this.mPurchaseListener.onIabPurchaseFinished(new IabResult(0, "Success"), purchase);
371                 }
```

```
119     public boolean isTestSku() {
120         return this._sku.startsWith("android.test");
        }
```

Security bypass
for test value
in production code

2. Real-life exploitation 10/13

- **Step 9: Bytecode patching**
  - Hook requires a rooted smartphone
  - We want to update the original `apk` to be able to deploy it on any device
- Android doesn't use Java bytecode but Smali
  - `classes.dex` contains the Java classes converted to Smali bytecode
  - Smali bytecode can be transformed back and forth to human readble instructions
- Principle
  - Get readable smali of original class
  - Get readable smali of hook
  - Manual merge hook in original class
  - Rebuild the APK with the new smali code (including signature)

2. Real-life exploitation 11/13

- Convert APK to readable smali with command
  - `java -jar apktool.jar d your.apk`
- Edit manually smali code in `your/smali`
- Recompiling with apktool loses native library, instead
  - `cp your.apk yourPatched.apk`
  - `java -jar smali_2.1.1.jar your/smali -o classes.dex`
    (to compile smali code)
  - `zip yourPatched.apk classes.dex`
  - `zip --delete yourPatched.apk "META-INF/*"`
    (to delete the existing signature)

2. Real-life exploitation 12/13

prime31 — vi GoogleIABPlugin.smali — 80×24

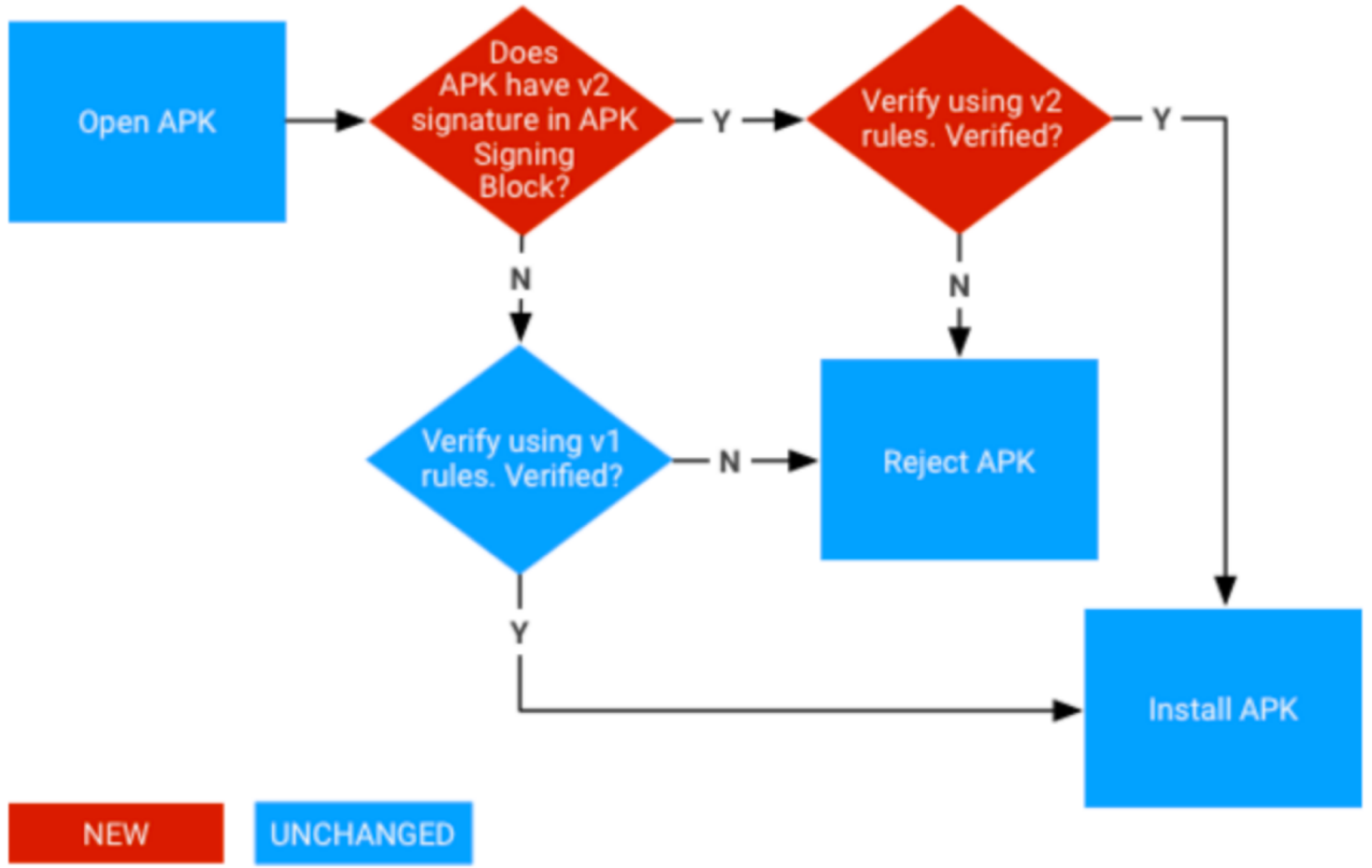
```
.method public purchaseProduct(Ljava/lang/String;Ljava/lang/String;)V
    .locals 8
    .param p1, "sku"    # Ljava/lang/String;
    .param p2, "developerPayload"    # Ljava/lang/String;

    .prologue
    .line 160
    const-string p1, "android.test.purchased"
    invoke-virtual {p0}, Ljava/lang/Object;->getClass()Ljava/lang/Class;

    move-result-object v4

    invoke-virtual {v4}, Ljava/lang/Class;->getSimpleName()Ljava/lang/String;

    move-result-object v4

    const-string v5, "purchaseProduct"

    const/4 v6, 0x2

    new-array v6, v6, [Ljava/lang/Object;
```

2. Real-life exploitation 13/13

- **Sign the APK with the key of your choice !**
- Generate a new signing key with
  `keytool -genkey -v -keystore patch.keystore -alias patch -keyalg RSA -keysize 2048 -validity 10000`
  Enter whatever your want in password and certificate info
- Sign APK with `jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore patch.keystore yourPatched.apk patch`
- Ensure signature is OK `jarsigner -verify -verbose -certs yourPatched.apk`
- Deploy to a non rooted device and play ;)
- We could even publish in the Play Store under a new name !

But wait, Android signature v2 is safer ?

- Signature v2 from Android 7.0+
- V1 signature accepted for compatilibty reasons Android 6.0 and below
- => Just provide a v1 signature in the APK ....



3. Lessons learned 1/4

- Never let debug code in production app
  - Special test cases should be removed for official build !
  - Poor design choice by Google to accept test value in production
- **An access control decision client side is insecure by design**
- Google documentation is misleading: cf https://developer.android.com/google/play/billing/billing_best_practices.html

Validating purchase details

It's **highly recommended** to validate purchase details on a server that you trust. If you cannot use a server, however, it's still possible to validate these details within your app on a device.

3. Lessons learned 2/4

Validate on a device

If you cannot run your own server, you can still validate purchase details within your Android app.

> **Warning:** This form of verification isn't truly secure. Because your purchase verification logic is bundled within your app, this logic becomes compromised if your app is reverse-engineered.

You should obfuscate your Google Play public key and In-app Billing code so it's difficult for an attacker to reverse-engineer security protocols a

- **in-app billing can't be used to buy credits**
  - Designed to purchase original content that is not guessable
  - Otherwise always possible to modify the counter via hooking or bytecode patching

Consumable products

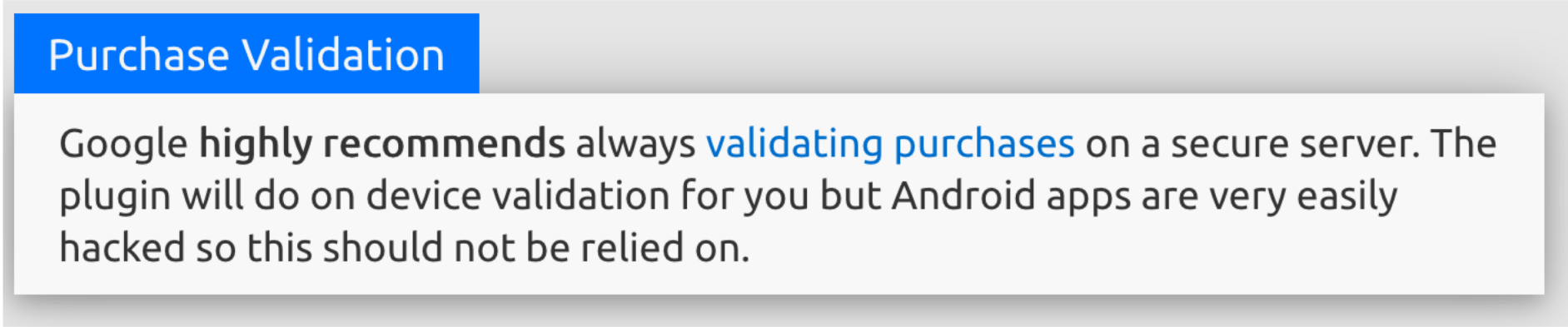
In contrast, you can implement consumption for products that can be made available for purchase multiple times. Typically, these products provide certain temporary effects. For example, the user's in-game character might gain life points or gain extra gold coins in their inventory. Dispensing the benefits or effects of the purchased product in your

3. Lessons learned 3/4

- **Responsible disclosure: no one cared**
- Game editor of PandaPop:
  - /dev/null
- Prime 31: wrote the Android in-app purchase integration code
  - Round 1: Quick feedback through their ticketing tool
    - « This vulnerability doesn't make any sense »
    - « The developer should be checking the sku of the purchased product »
  - Round 2: I buy the plugin 70 USD
    - Unity plugin: C# wrapper on top of Java Android API
    - I am supposed to receive integration documents

3. Lessons learned 4/4

- Developer doc is in fact just a link to their basic website



- Yet API supports signature verification on an external server
  - But provided C# demo does not use it

- Round 3: detailed slides back to Prime 31
  - « Excellent! Many thanks for these, I look forward to reading them today »
  - Since then: /dev/null
  - As a customer I don't get any fix

4. Recommendations

- 0. Use Proguard obfuscation to slow down a reverser
- 1. Use NDK to embed sensitive logic in C code
  - With JNI possible to call C librairies via the `native` keyword
  - Much more effort to reverse and patch binary code (e.g ARM)
- 2. Use a backend for validating purchases
  - Still possible to hook/patch the response of the server
- 3. Only sell « real » content
  - and not something easy to guess like a counter
  - e.g Angry Birds sell extra levels
  - and they also use NDK for calls to validation server

4. Recommendations

com.rovio.rcs.payment.google.GooglePlayPaymentProvider

```
    private static native void paymentFinished(long j, String str, int i, String str2, String str3);

    private static native void restoreDone(long j);

    private static native void restoreFailed(long j);

    private static native void skuDetailsLoaded(long j, SkuDetails[] skuDetailsArr);

239 public GooglePlayPaymentProvider(long j) {
241     this.f5016b = j;
247     Globals.registerActivityListener((IActivityListener) this);
250     IntentFilter intentFilter = new IntentFilter("com.android.vending.billing.PURCHASE_UPDATED");
251     this.f5021g = new C18591(this);
260     this.f5017c.registerReceiver(this.f5021g, intentFilter);
270     new Thread(new C18602(this)).start();
    }
```

Conclusion

- **Android Java reverse engineering is really easy** with jadx
- **You cannot trust the Java code running in your Android app**
  - Modifying and resigning an APK is not difficult
- Only server side code can be considered secure
- Google recommendations for in-app purchases are incomplete and misleading
  - By design most in-app uses cases are not possible to secure
  - Only secure use case: download impredictable content from server

Any question ?

contact@securingapps.com

Bonus

- Possible to debug in Android Studio a reversed app
  - Jadx can export to an Android Studio project
  - Add `android:debuggable=''true''` in `AndroidManifest.xml`
  - Resign app
  - Deploy and start debugging from Android Studio