Abusing Google Play Billing for fun and unlimited credits!

Guillaume Lopes - @Guillaume_Lopes

RANDORISEC

28/03/2019

Who am I?

- Senior Penetration Tester at Randorisec
  - 10 years of experience in different fields (Active Directory, Windows, Linux, Web applications, Wifi, Android)
- Member of the Checkmarx Application Security Research Team
  - https://www.checkmarx.com/category/blog/technical-blog/
- Play CTF (Hackthebox, Insomni'hack, Nuit du Hack, Bsides Lisbon, etc.
  - Gives a hand to the Tipi'hack team

Agenda

1. Google Play Billing Presentation
2. Known Vulnerabilities
3. Vulnerable Applications
4. Conclusion

Google Play Billing Presentation

How does it works?

Google Play Billing Presentation

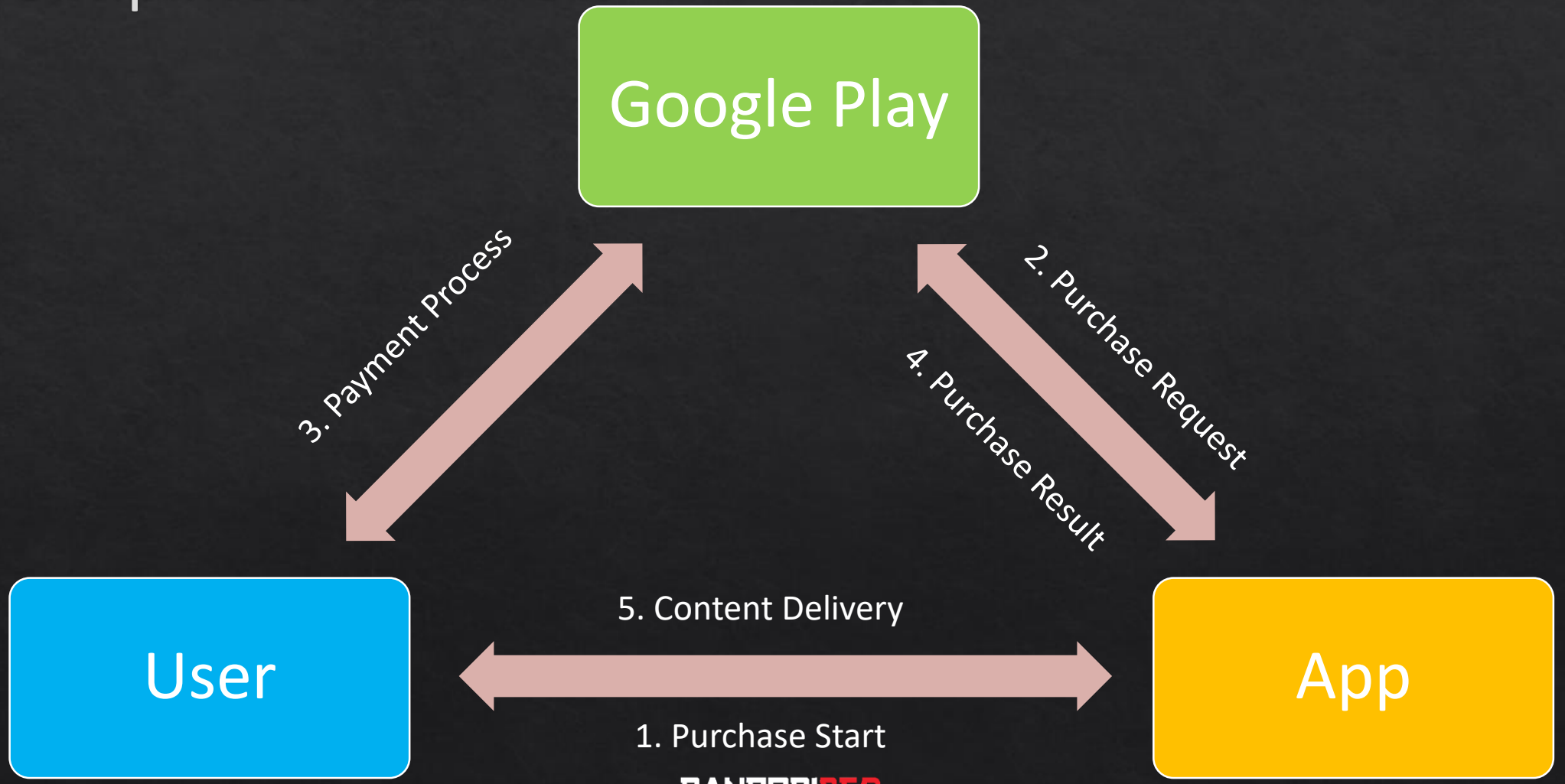
- Android framework that allows to easily monetize applications with in-app purchases and subscriptions
  - Subscriptions to magazines
  - Premium features
  - Extra content in games

Google Play Billing Presentation

- Payment is handled by Google
  - Need to have Google Play in your device
  - Credit card not exposed to the developers
  - Products need to be defined in the Google Play Console
  - Tracking made by Google

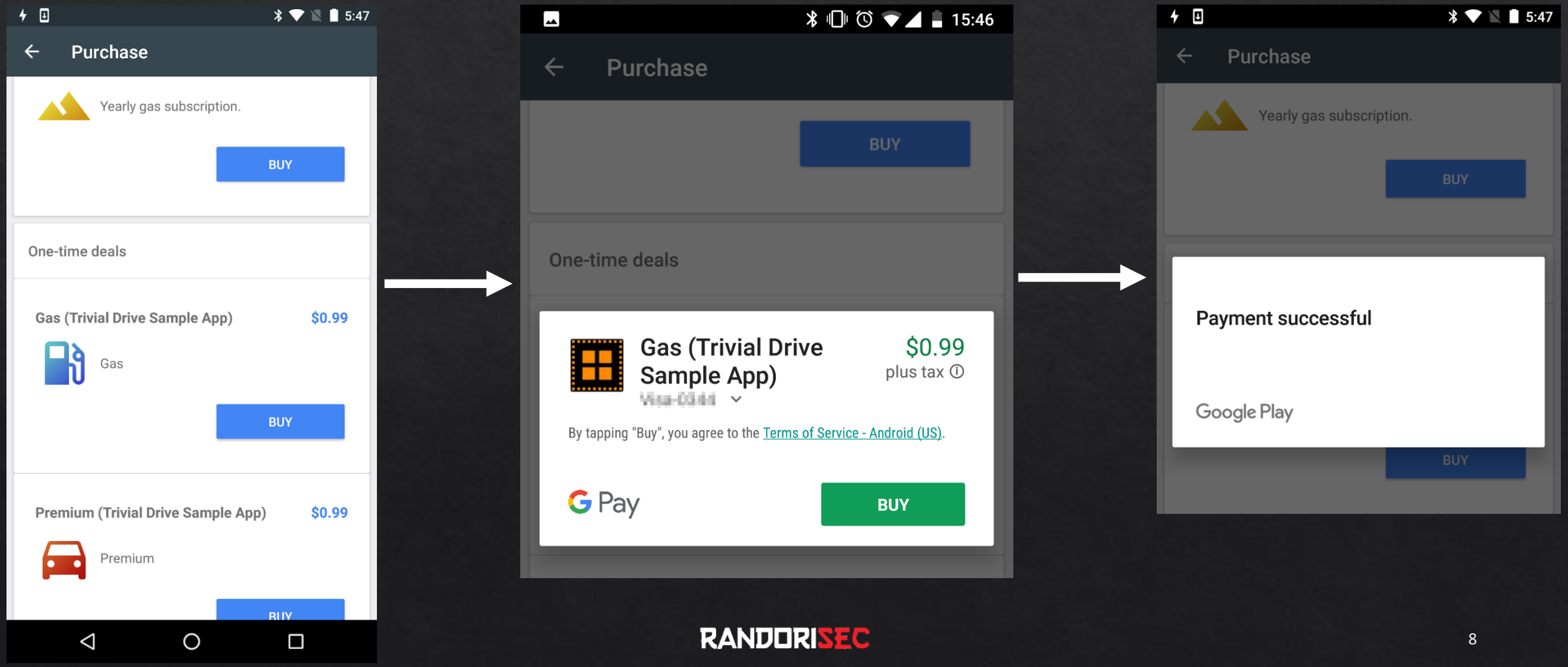
Google Play Billing Presentation

- Simplified Workflow



Google Play Billing Presentation

- Payment process handled by Google



Google Play Billing Presentation

- Google returns a JSON object containing (not exhaustive)
  - **purchaseState**: Integer with 2 possible values 0 (Purchased) or 1 (Canceled)
  - **purchaseToken**: String generated by Google Play to uniquely identify the transaction
  - **signature**: String representing the signature of the purchase

Google Play Billing Presentation

- Google Play signs the JSON string that contains the response data for a purchase
  - The Google Play Console generates an RSA key pair for each application
  - The private key is associated to the application used

Google Play Billing Presentation

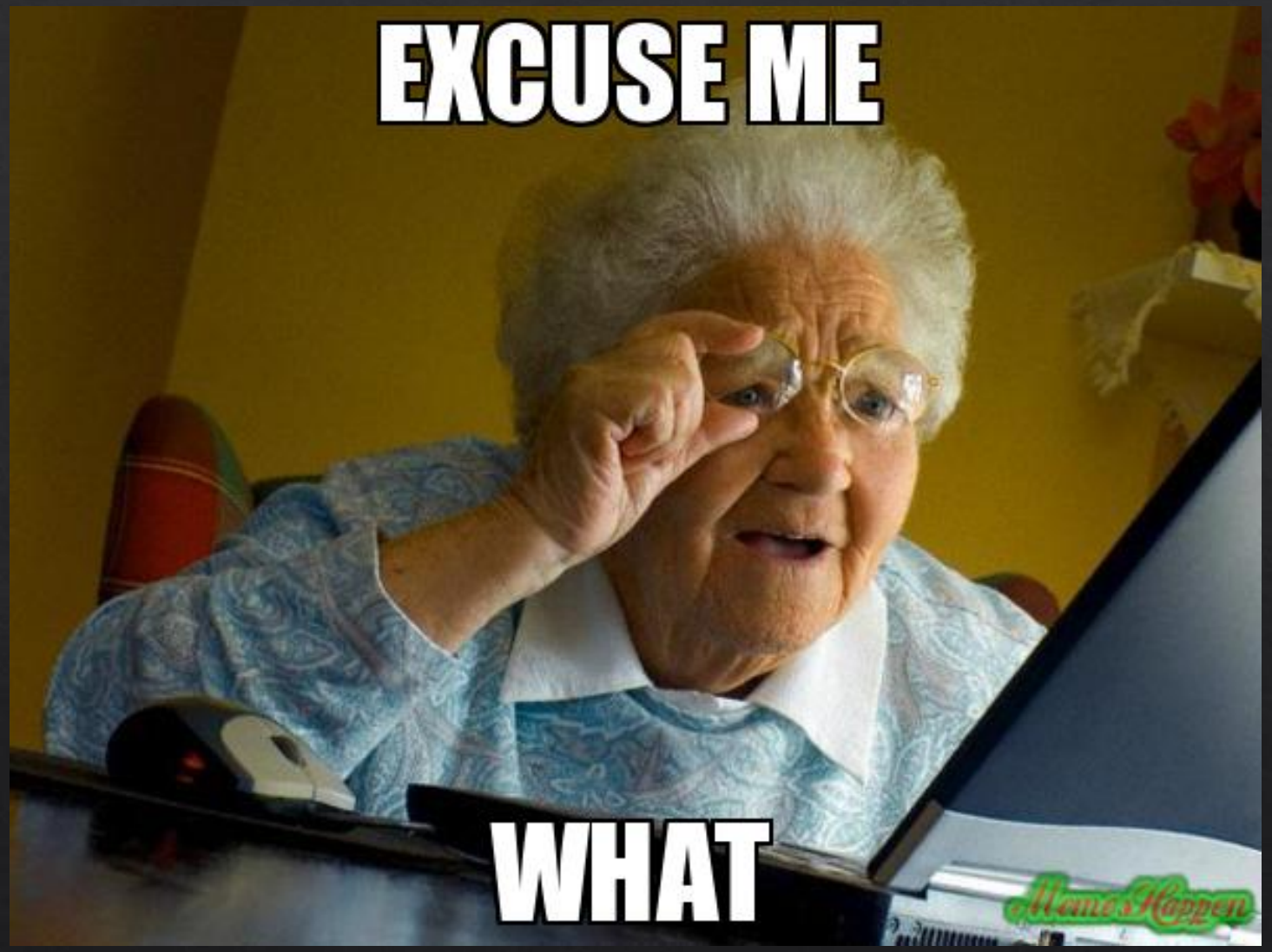
- Trivial Drive v2
  - Sample app
  - Example on how to use the Google Play Billing API

```
/**
 * Security-related methods. For a secure implementation, all of this code should be implemented on
 * a server that communicates with the application on the device.
 */
public class Security {
    private static final String TAG = "IABUtil/Security";

    private static final String KEY_FACTORY_ALGORITHM = "RSA";
    private static final String SIGNATURE_ALGORITHM = "SHA1withRSA";

    /**
     * Verifies that the data was signed with the given signature, and returns the verified
     * purchase.
     * @param base64PublicKey the base64-encoded public key to use for verifying.
     * @param signedData the signed JSON string (signed, not encrypted)
     * @param signature the signature for the data, signed with the private key
     * @throws IOException if encoding algorithm is not supported or key specification
     * is invalid
     */
    public static boolean verifyPurchase(String base64PublicKey, String signedData,
            String signature) throws IOException {
        if (TextUtils.isEmpty(signedData) || TextUtils.isEmpty(base64PublicKey)
                || TextUtils.isEmpty(signature)) {
            BillingHelper.logWarn(TAG, "Purchase verification failed: missing data.");
            return false;
        }

        PublicKey key = generatePublicKey(base64PublicKey);
        return verify(key, signedData, signature);
    }
```

Google Play Billing Presentation

```
/**
 * Security-related methods. For a secure implementation, all of this code should be implemented on
 * a server that communicates with the application on the device.
 */
```

Google Play Billing Presentation



Google Play Billing Presentation

- Google recommends to validate purchase details on a server controlled by the developer

> **Note:** It's **highly recommended** to verify purchase details using a secure backend server that you trust. When a server isn't an option, you can perform less-secure validation within your app.

- However, it is still possible to verify the purchase on the device by validating the signature

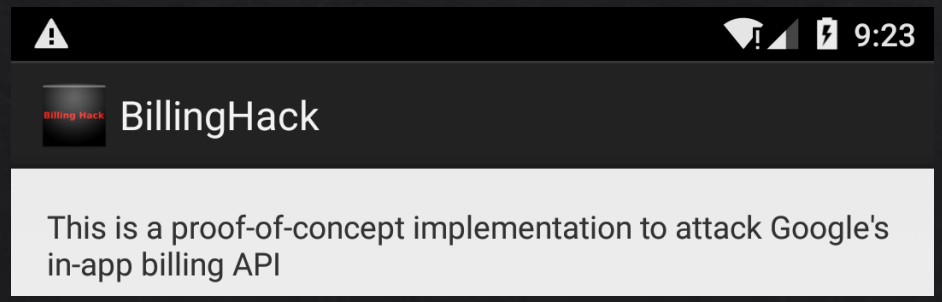
> **Warning:** This form of verification isn't truly secure because it requires you to bundle purchase verification logic within your app. This logic becomes compromised if your app is reverse-engineered.

Known Vulnerabilities

A little bit of history

Known Vulnerabilities

- 2013: Dominik Schürmann found 2 vulnerabilities allowing to bypass the payment process
  - Bug disclosed to Google (Hall of Fame)
- Dominik developed an app as PoC called BillingHack
  - Just need to launch the app in background



Known Vulnerabilities

1. A malicious app is able to impersonate the Google Play billing service (com.android.vending)

- Define an Intent filter with a high priority

```
<intent-filter android:priority="2147483647" >
    <action android:name="com.android.vending.billing.InAppBillingService.BIND" />
</intent-filter>
```

Known Vulnerabilities

2. The signature verification returns true, if the signature is an empty string

```
public static boolean verifyPurchase(String base64PublicKey, String signedData, String signature) {
    if (signedData == null) {
        Log.e(TAG, "data is null");
        return false;
    }

    boolean verified = false;
    if (!TextUtils.isEmpty(signature)) {
        PublicKey key = Security.generatePublicKey(base64PublicKey);
        verified = Security.verify(key, signedData, signature);
        if (!verified) {
            Log.w(TAG, "signature does not match data.");
            return false;
        }
    }
    return true;
}
```

Known Vulnerabilities

- Google fixed these 2 vulnerabilities by applying the following modifications

  1. Every app using the Google Play Billing API should define which is the targeted package for the intent

```
Intent serviceIntent = new Intent("com.android.vending.billing.InAppBillingService.BIND");
serviceIntent.setPackage("com.android.vending");
```

Source: Trivial Drive v2

Known Vulnerabilities

2. The function checking the signature was modified in order to return true only if the signature is valid

```
public static boolean verifyPurchase(String base64PublicKey, String signedData, String signature) {
    if (TextUtils.isEmpty(signedData) || TextUtils.isEmpty(base64PublicKey) ||
            TextUtils.isEmpty(signature)) {
        Log.e(TAG, "Purchase verification failed: missing data.");
        return false;
    }

    PublicKey key = Security.generatePublicKey(base64PublicKey);
    return Security.verify(key, signedData, signature);
}
```

Source: Trivial Drive v2

Known Vulnerabilities

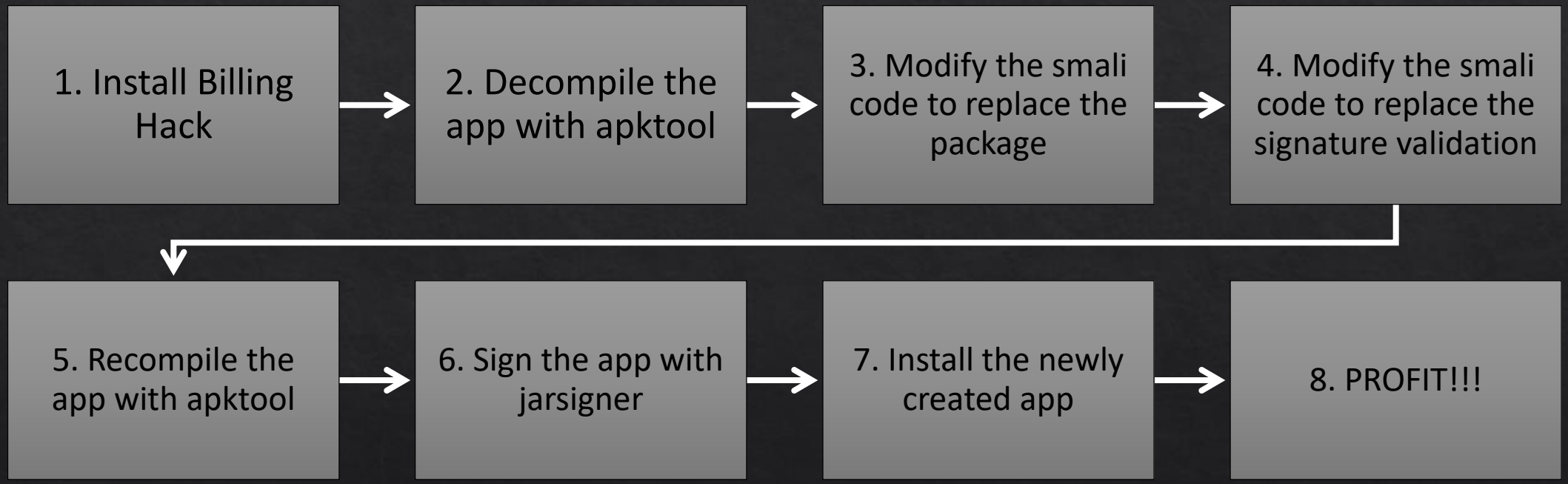
Known Vulnerabilities

- To sum-up: If your app is performing the verification process locally, you can always circumvent the payment by
  1. Binding the Intent service to an app you control
  2. Modify the signature verification in order to return always true

- The main “problem” is to find how the app is performing the signature verification!

Known Vulnerabilities

- Hacking Steps



Vulnerable Applications

How to obtain unlimited credits?

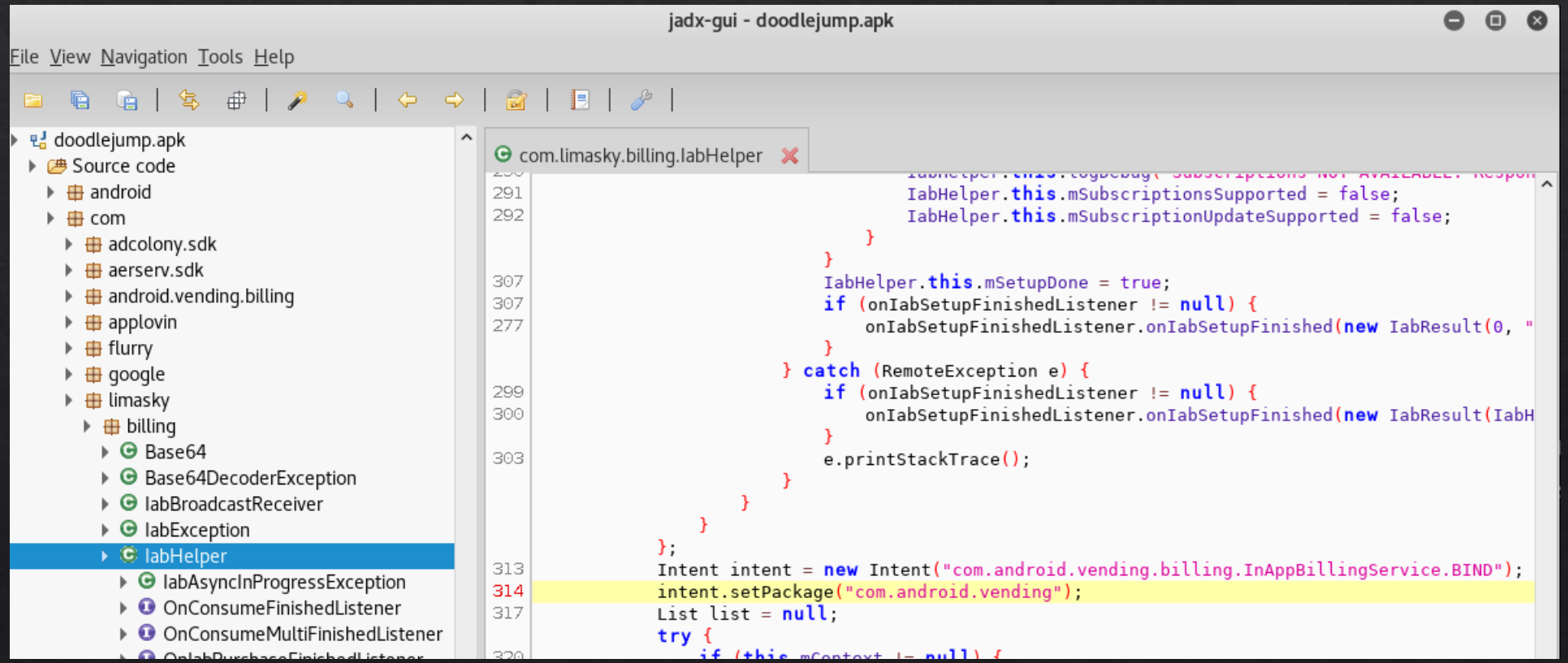
Vulnerable Applications

- Doodle Jump (com.lima.doodlejump)
  - Platform game (“How high can you get?”)
  - “Named Best of 2015 by Google Play editors”
  - Buy different items, but you need candies!

RANDORISEC

Vulnerable Applications

- Very easy to modify in order to buy items for free!
  - Replace “com.android.vending” by “org.billinghack”



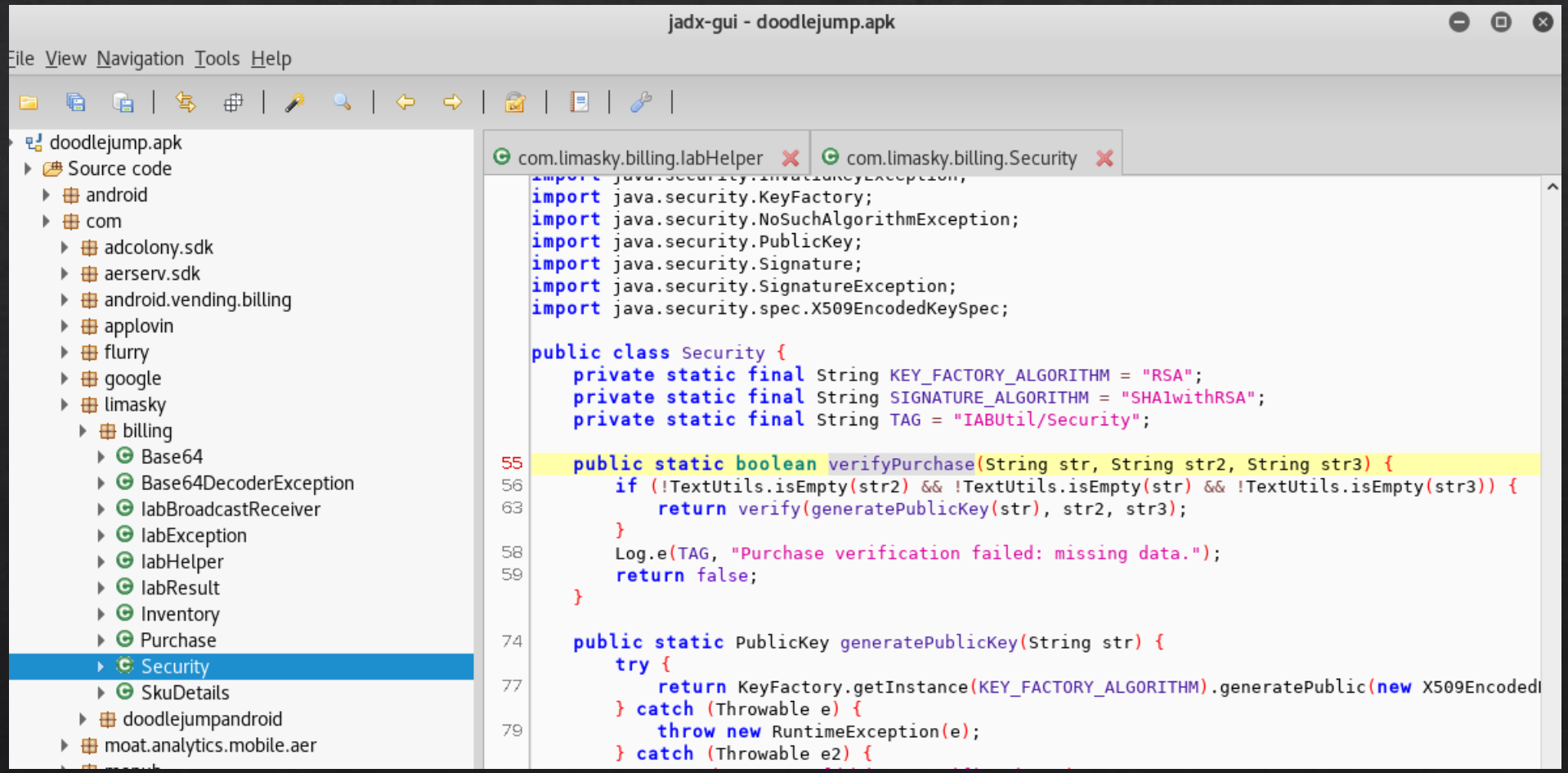
Vulnerable Applications

- Smali code

```
.line 313
new-instance v1, Landroid/content/Intent;

const-string v0, "com.android.vending.billing.InAppBillingService.BIND"

invoke-direct {v1, v0}, Landroid/content/Intent;-><init>(Ljava/lang/String;)V

.line 314
const-string v0, "org.billinghack"

invoke-virtual {v1, v0}, Landroid/content/Intent;->setPackage(Ljava/lang/String;)Landroid/content/Intent;
```

Vulnerable Applications

- Then, modify the “verifyPurchase” function to return true



Vulnerable Applications

- Smali code

```
.line 63
:goto_0
const/4 v0, 0x1
return v0

.line 62
:cond_1
invoke-static {p0}, Lcom/limasky/billing/Security;-
>generatePublicKey(Ljava/lang/String;)Ljava/security/PublicKey;

move-result-object v0
.line 63
invoke-static {v0, p1, p2}, Lcom/limasky/billing/Security;-
>verify(Ljava/security/PublicKey;Ljava/lang/String;Ljava/lang/String;)Z

move-result v0

goto :goto_0
.end method
```

Vulnerable Applications

DEMO

Vulnerable Applications

- Snoopy Pop (com.jamcity.snoopypop)
  - Game similar to Bubble Witch but with Snoopy
  - You can buy coins and lives

Vulnerable Applications

- Unity library is used for the graphics
  - But Unity also offers a Google Play Billing interface
  - However Unity does not offer server-side validation

Point of validation

It is best practice to validate the receipt at the point where your application's content is distributed.

- **Local validation:** For client-side content, where all content is contained in the application and is enabled once purchased, the validation should take place on the target device, without the need to connect to a remote server. Unity IAP is designed to support local validation within your application. See **Local validation** below for more information.
- **Remote validation:** For server-side content, where content is downloaded once purchased, the validation should take place on the server before the content is released. Unity does not offer support for server-side validation; however, third-party solutions are available, such as Nobuyori Takahashi's IAP project.

  - Source: https://docs.unity3d.com/Manual/UnityIAPValidatingReceipts.html

Vulnerable Applications

- Most of the Unity's code is written in Mono .NET
  - These DLLs are stored on /assets/bin/Data/Managed

```
# ls assets/bin/Data/Managed/
Analytics.dll  Assembly-CSharp-firstpass.dll  Facebook.Unity.dll  mscorlib.dll  Stores.dll  System.Xml.dll
UnityEngine.Analytics.dll  UnityEngine.Purchasing.dll  winrt.dll  Apple.dll  Common.dll  Facebook.Unity.IOS.dll
P31RestKit.dll  System.Core.dll  System.Xml.Linq.dll  UnityEngine.dll  UnityEngine.UI.dll  Assembly-CSharp.dll
Facebook.Unity.Android.dll  Mono.Security.dll  Security.dll  System.dll  Tizen.dll UnityEngine.Networking.dll
Validator.dll
```

  - The most interesting one is Security.dll
    - This DDL contains a function called "Validate" which verify the signature of the purchase

Vulnerable Applications

- With DnSpy, a .NET decompiler
  - It's trivial to obtain and modify the .NET code
- The "Validate" function throws an exception when the signature is invalid

GooglePlayValidator

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace UnityEngine.Purchasing.Security
6  {
7      // Token: 0x0200001A RID: 26
8      internal class GooglePlayValidator
9      {
10         // Token: 0x060000E3 RID: 227 RVA: 0x00006AD0 File Offset: 0x00004ED0
11         public GooglePlayValidator(byte[] rsaKey)
12         {
13             this.key = new RSAKey(rsaKey);
14         }
15
16         // Token: 0x060000E4 RID: 228 RVA: 0x00006AE8 File Offset: 0x00004EE8
17         public GooglePlayReceipt Validate(string receipt, string signature)
18         {
19             byte[] bytes = Encoding.UTF8.GetBytes(receipt);
20             byte[] signature2 = Convert.FromBase64String(signature);
21             if (!this.key.Verify(bytes, signature2))
22             {
23                 throw new InvalidSignatureException();
24             }
25             Dictionary<string, object> dictionary = (Dictionary<string, object>
26             object obj;
27             dictionary.TryGetValue("orderId", out obj);
28             object obj2;
29             dictionary.TryGetValue("packageName", out obj2);
30             object obj3;
```

Vulnerable Applications

- Then, we just need to remove the code performing the check

```
6  000E  call      uint8[] [mscorlib]System.Convert::FromBase64String(string)
7  0013  stloc.1
8  0014  ldarg.0
9  0015  ldfld     class UnityEngine.Purchasing.Security.RSAKey UnityEngine.Purchasing.Security.GooglePlayValidator::key
10 001A  ldloc.0
11 001B  ldloc.1
12 001C  callvirt  instance bool UnityEngine.Purchasing.Security.RSAKey::Verify(uint8[], uint8[])
13 0021  brtrue    17 (002D) ldarg.1
14 0026  nop
```

```
public GooglePlayReceipt Validate(string receipt, string signature)
{
    byte[] bytes = Encoding.UTF8.GetBytes(receipt);
    byte[] array = Convert.FromBase64String(signature);
    Dictionary<string, object> dictionary = (Dictionary<string, object>)
    object obj;
    dictionary.TryGetValue("orderId", out obj);
    object obj2;
    dictionary.TryGetValue("packageName", out obj2);
```

Vulnerable Applications

- At the end, we replace your modified DLL in the app
  - Don’t forget to modify the setPackage
  - Rebuild with apktool
  - And PROFIT!

- DEMO

Vulnerable Applications

- Fruit Ninja (com.halfbrick.fruitninjafree)
  - Famous game where you need to cut fruits (like a ninja!)
  - More than 100 millions of downloads

Vulnerable Applications

- Java Native Interface (JNI)
  - JNI allows to interact with native code (C/C++) from Java/Kotlin
  - In short, you can embedded a shared library and your app can call functions from this library
- FruitNinja implements sensitive functions using JNI
  - And mostly for InApp Billing functions

```
private static native void GotDisplayCostNative(String str, float f, String str2, String str3);
private static native void PurchaseResultNative(String str, boolean z, boolean z2, String str2, String str3);
private static native void UnsolicitedReceiptNative(String str, boolean z, String str2, String str3);
```

Vulnerable Applications

- Need to reverse engineer the shared library

```
kali# ls -lh libmortargame.so
-rw-r--r-- 1 root root 24M sept. 14 00:23 libmortargame.so
kali# strings libmortargame.so| grep PurchaseResultNative
PurchaseResultNative
kali#
```

- Shared library coded in C++
  - Time consuming!
  - Difficult to rebuild a new shared library

Vulnerable Applications

- However, it seems that the signature validation is poorly made
  - So it’s possible to bypass the payment

- DEMO

Conclusion

That’s it?

Conclusion

- Developers use different techniques to perform the Google Play Billing payment
  - Obfuscation
  - Shared library
  - Nothing!
- However, the signature validation is mainly performed locally inside the app

Conclusion

- On 30 apps tested
  - 15 apps were vulnerable (bypass payment)
  - Only 4 apps used an external endpoint to perform additional checks

- I contacted some editors, but I never got an answer
  - The issues are still present

Conclusion

- Regarding other Billing libraries, Google is the only one allowing local validation
  - Amazon IAP (In-App Purchase) needs a server to retrieve the content
  - Samsung In-App Purchase uses a server to validate the purchase

Thanks for the support!

Questions?

References 1/3

- Google Play Billing documentation
  - https://developer.android.com/google/play/billing/index.html
- Google Play Billing Best Practices
  - https://developer.android.com/google/play/billing/billing_best_practices.html
- Google Play In-App Billing Library Hacked
  - https://www.schuermann.eu/2013/10/29/google-play-billing-hacked.html

References 2/3

- Billing Hack Source Code
  - https://github.com/dschuermann/billing-hack
- Google prevents vulnerable apps on the Play Store
  - https://support.google.com/faqs/answer/7054270?hl=en
- Amazon documentation
  - https://developer.amazon.com/fr/docs/in-app-purchasing/iap-rvs-for-android-apps.html
- Samsung documentation
  - https://developer.samsung.com/iap#overview

References 3/3

- Get Freebies by Abusing the Android InApp Billing API
  - https://www.checkmarx.com/blog/abusing-android-inapp-billing-api/
- Abusing Android In-app Billing feature thanks to a misunderstood integration
  - https://www.securingapps.com/blog/BsidesLisbon17_AbusingAndroidInappBilling.pdf