

PIVA

Purposefully Insecure and Vulnerable Android Application

To Download:

git clone <https://github.com/HTBridge/pivaa.git>

Decompiling:

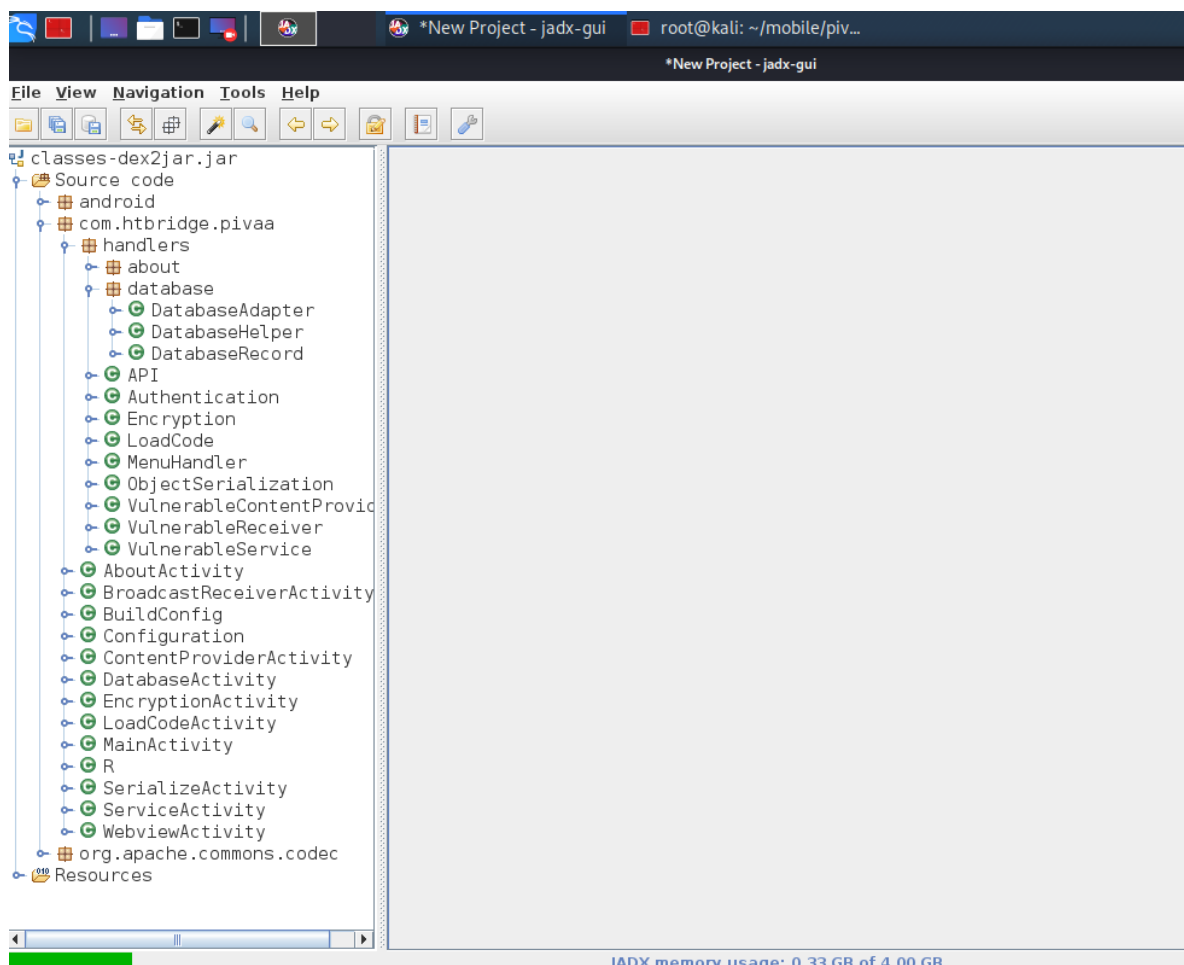
Way 1 : (view source code)

unzip pivaa.apk

d2j-dex2jar classes.dex

jadx-gui < **FULL** path to classes-dex2jar.jar>

```
root@kali: ~/mobile/pivaa/decompile/unzipped
root@kali: ~/mobile/pivaa/decompile/unzipped 114x31
root@kali:~/mobile/pivaa/decompile/unzipped# ls
AndroidManifest.xml  assets  classes.dex  classes-dex2jar.jar  META-INF  res  resources.arsc
root@kali:~/mobile/pivaa/decompile/unzipped# jadx-gui /root/mobile/pivaa/decompile/unzipped/classes-dex2jar.jar
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
INFO - output directory: classes-dex2jar
INFO - loading ...
INFO - converting to dex: classes-dex2jar.jar ...
```



Way 2: (can view smali, AndroidManifest.xml , modify and recompile)

apktool d pivaa.apk

```
root@kali: ~/mobile/pivaa/decompile/unzipped
root@kali: ~/mobile/pivaa/decompile/pivaa 125x30
root@kali:~/mobile/pivaa/decompile/pivaa# ls
AndroidManifest.xml  apktool.yml  assets  original  res  smali
root@kali:~/mobile/pivaa/decompile/pivaa# cat AndroidManifest.xml
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.htbridge.pivaa">
    <uses-permission android:name="android.permission.GET_ACCOUNTS"/>
    <uses-permission android:name="android.permission.READ_PROFILE"/>
    <uses-permission android:name="android.permission.READ_CONTACTS"/>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.INTERNET"/>
```

Rebuild and Sign the apk:

apktool b pivaa -o pivaa.s.apk

keytool -genkey -v -keystore test.keystore -alias Test -keyalg RSA -keysize 1024 -sigalg SHA1withRSA -validity 10000

jarsigner -keystore test.keystore pivaa.s.apk -sigalg SHA1withRSA -digestalg SHA1 Test

jarsigner --verify --verbose pivaa.s.apk

Vulnerabilities:

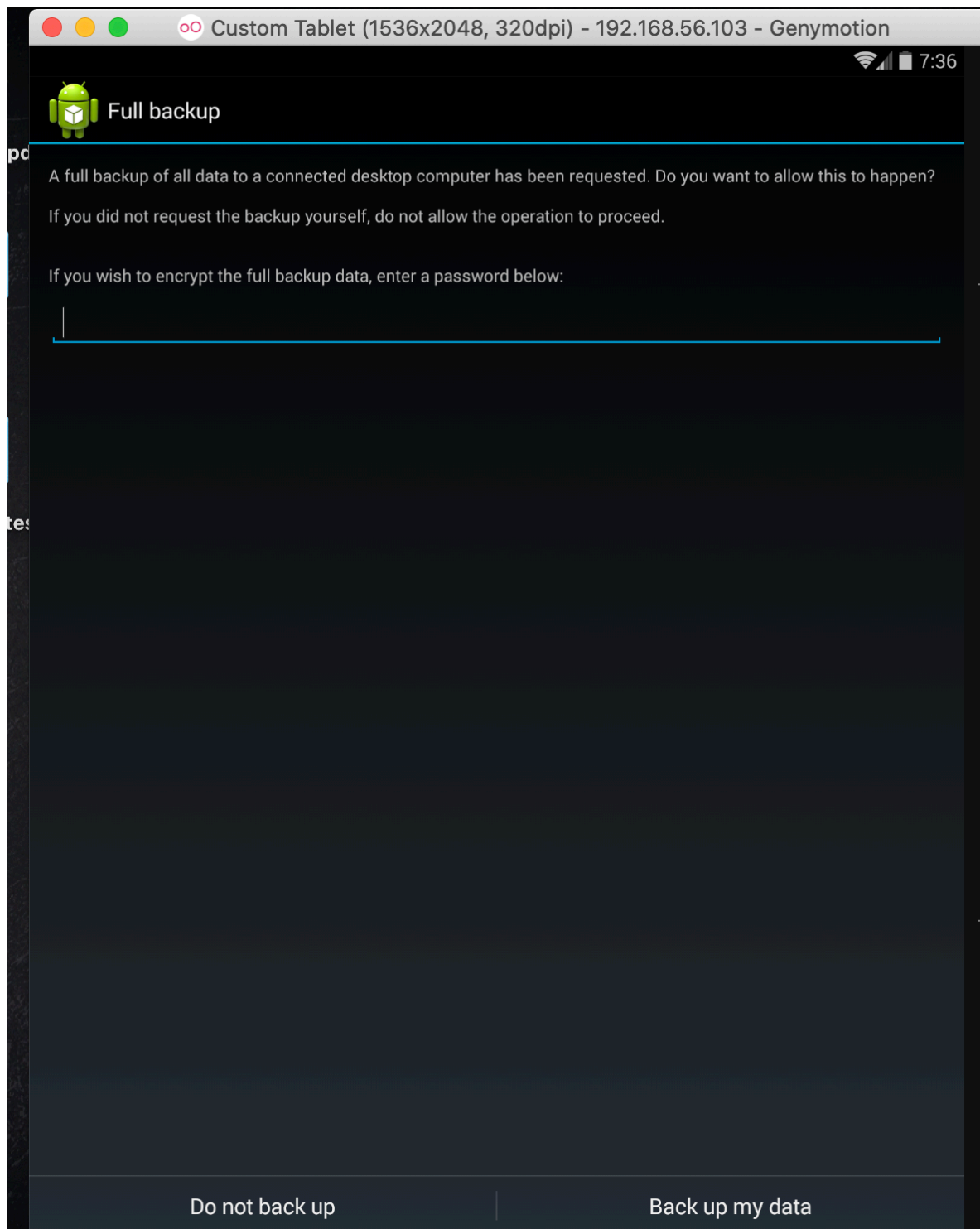
Enabled Application Backup

The mobile application uses external backup functionality (default Android backup mechanism) that may store inside sensitive data from the application. In certain conditions, this may lead to information disclosure (e.g. when a backup server or your Gmail account is compromised).

```
<application android:allowBackup="true" android:debuggable="true" android:icon="@mipmap/ic_launcher" android:label="@string/app_name"
android:protectionLevel="dangerous" android:roundIcon="@mipmap/ic_launcher_round" android:supportRtl="true"
android:theme="@style/AppTheme">
```

adb backup -apk -shared com.htbridge.pivaa --> this command creates a backup.ab file

```
root@kali: ~/mobile/pivaa/decompile/pivaa 125x30
root@kali:~/mobile/pivaa/decompile/pivaa# adb backup -apk -shared com.htbridge.pivaa
Now unlock your device and confirm the backup operation...
```



Enter the below command to convert the backup file into readable format.
`cat backup.ab | (dd bs=24 count=0 skip=1; cat) | zlib-flate -uncompress > backup_compressed.tar`

OR download an android backup extractor tool.

Recommendation :

close this feature whenever the application is deployed

Hardcoded data

The mobile application contains debugging or potentially sensitive hardcoded data. An attacker with an access to the mobile application file can easily extract this data from the application and use it in any further attacks if it contains any internal or confidential information.

```
com.htbridge.pivaa.Configuration X
1 package com.htbridge.pivaa;
2
3 public class Configuration {
4     public String default_author_database_item = "High-Tech Bridge";
5     public String default_title_database_item = "My Vulnerable Android Application";
6     public String password = "verycomplicatedpassword";
7     public String url_json = "https://www.htbridge.com/ssl/api/v1/load_all/1510314123771.1";
8     public String url_webview = "https://www.htbridge.com/ssl/";
9     public String url_webview_link_1 = "file:///etc/hosts";
10    public String url_webview_link_2 = "https://xss.rocks/scriptlet.html";
11    public String username = "test";
12 }
```

Recommendation:

Remove these data from the code as they are only useful for testing.

WebView Related Attacks:

```
com.htbridge.pivaa.WebviewActivity X
41
42 }
43 };
44 ((Button) findViewById(R.id.button_webview_link1)).setOnClickListener(new OnClickListener() {
45     public void onClick(View view) {
46         EditText editText = (EditText) WebviewActivity.this.findViewById(R.id.url_webview);
47         WebviewActivity.this.myWebView.loadUrl(WebviewActivity.this.config.url_webview_link_1);
48         editText.setText(WebviewActivity.this.config.url_webview_link_1, BufferType.EDITABLE);
49     }
50 });
51 ((Button) findViewById(R.id.button_webview_link2)).setOnClickListener(new OnClickListener() {
52     public void onClick(View view) {
53         EditText editText = (EditText) WebviewActivity.this.findViewById(R.id.url_webview);
54         WebviewActivity.this.myWebView.loadUrl(WebviewActivity.this.config.url_webview_link_2);
55         editText.setText(WebviewActivity.this.config.url_webview_link_2, BufferType.EDITABLE);
56     }
57 });
58 ((Button) findViewById(R.id.button_webview_link3)).setOnClickListener(new OnClickListener() {
59     public void onClick(View view) {
60         EditText editText = (EditText) WebviewActivity.this.findViewById(R.id.url_webview);
61         WebviewActivity.this.myWebView.loadUrl("http://google.com");
62         editText.setText("http://google.com", BufferType.EDITABLE);
63     }
64 });
65 ((Button) findViewById(R.id.button_xss)).setOnClickListener(new OnClickListener() {
66     public void onClick(View view) {
67         Log.i("htbridge", "Clicked XSS button");
68         WebviewActivity.this.myWebView.loadUrl("javascript:alert('Hello World!')");
69     }
70 });
```

here we can do a Path traversal attack

Loading an external URL

Since JS is enabled
The application is vulnerable to XSS

Remote URL load in WebView

Loading a remote URL can be a dangerous practice in WebView. Verify the interactions made with WebView and ensure the trustworthiness, integrity and reliability of third-party URLs used in the mobile application.

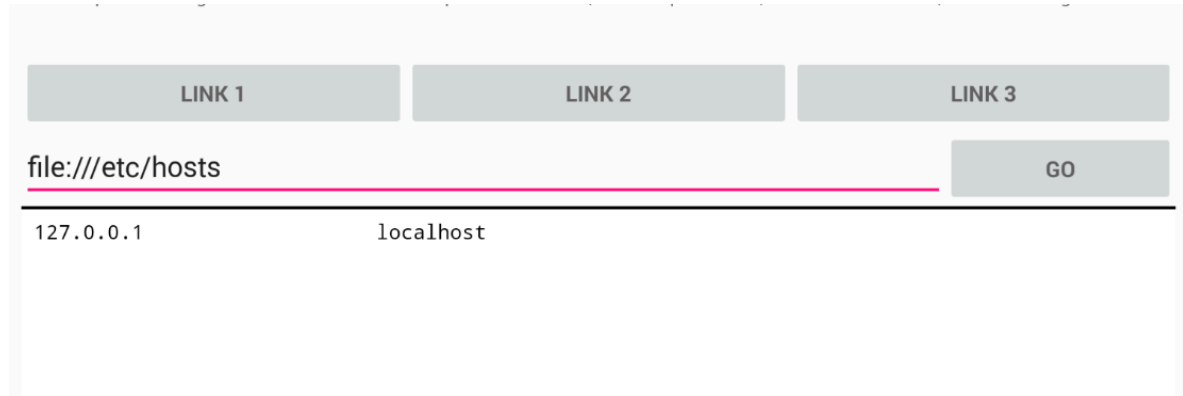
This can be found in link 3

Path Traversal

The mobile application may be vulnerable to path traversal vulnerability. The mobile application uses NSTemporaryDirectory() method that can be misused if not implemented properly.

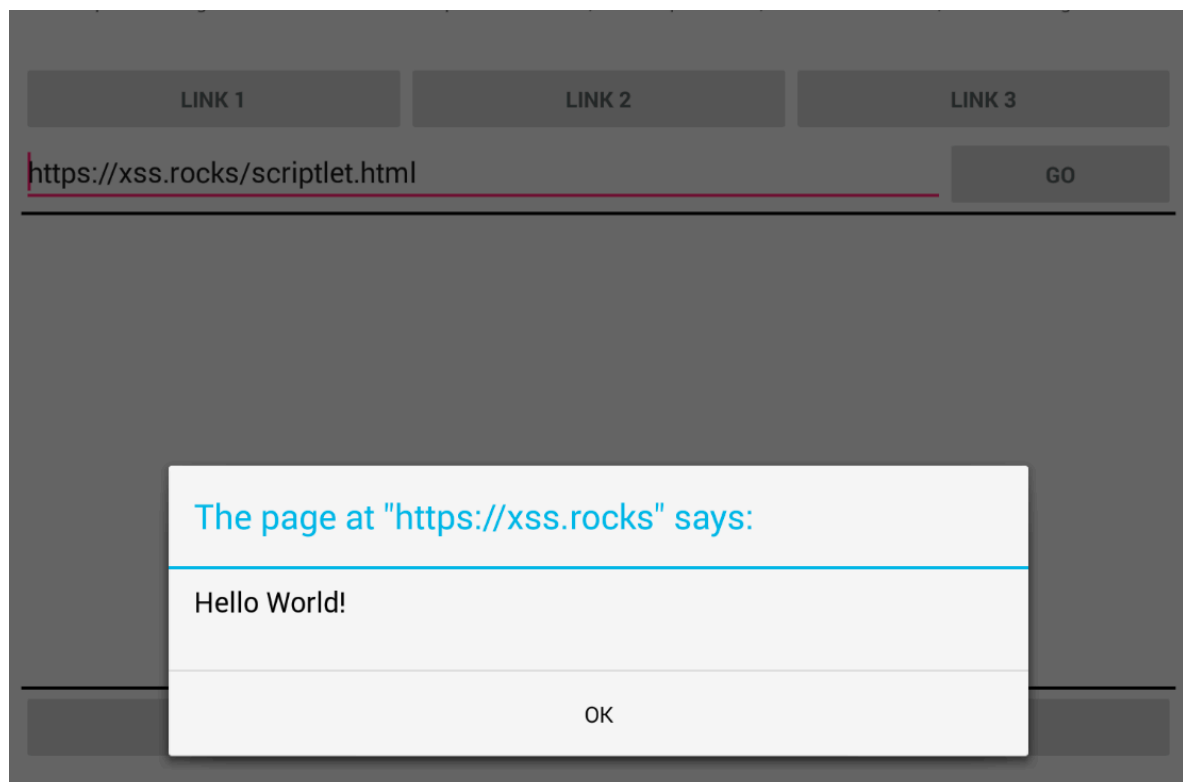
here we find Path traversal

This can be found in links 1 and 2



JS enabled in a WebView

The mobile application has enabled JavaScript in WebView. By default, JavaScript is disabled in WebView, if enabled it can bring various JS-related security issues, such as Cross-Site Scripting (XSS) attacks.



Recommendations:

Disable Javascript.

Input sanitization for inputs taken from the user and if possible best to whitelist specific inputs.

Enabled Debug Mode

The mobile application has debug mode enabled. Debug mode is used by application developers during development process and should be disabled when application is in production. This mode can expose technical information and can facilitate reverse engineering of the application.

```
<application android:allowBackup="true" android:debuggable="true" android:icon="@mipmap/ic_launcher" android:label="@string/app_name"
android:protectionLevel="dangerous" android:roundIcon="@mipmap/ic_launcher_round" android:supportRtl="true"
android:theme="@style/AppTheme">
```

```
adb shell          --> get shell on the device
ps                 --> find the process id of "com.htbridge.pivaa"
adb forward tcp:12345 jdwp:<above id>
jdb -attach localhost:12345
```

now we can view classes , methods , break on methods and view local variables modify them at any step.

Recommendations:

Close debug mode

User-supplied input in SQL queries

Inclusion of user-supplied input into SQL queries can potentially lead to a local SQL injection vulnerability in the mobile application. The correct approach is to use prepared SQL statements beyond user's control.

```
((Button) findViewById(R.id.button_insert_database)).setOnClickListener(new OnClickListener() {
    public void onClick(View view) {
        EditText editText = (EditText) DatabaseActivity.this.findViewById(R.id.author_database);
        DatabaseActivity.this.db.addRecord(new DatabaseRecord(((EditText) DatabaseActivity.this.findViewById(R.id.author_database)).getText().toString()));
        DatabaseActivity.this.renderListView();
    }
});
((Button) findViewById(R.id.button_raw_sql_database)).setOnClickListener(new OnClickListener() {
    public void onClick(View view) {
        String rawSQLQuery = DatabaseActivity.this.db.rawQuery(((EditText) DatabaseActivity.this.findViewById(R.id.author_database)).getText().toString(), null);
        Log.d("htbridge", rawSQLQuery);
        ((TextView) DatabaseActivity.this.findViewById(R.id.output_database)).setText(rawSQLQuery, BufferType.NORMAL_TEXT);
        DatabaseActivity.this.renderListView();
    }
});
```

Recommendation:

Use prepared statements

Creation of world readable or writable files

The mobile application creates files with world readable or writable permissions. Such files can be accessed and modified by other applications, including malicious ones, thus imperiling the application's data integrity.



```

48
49 UserLoginTask(String str, String str2) {
50     this.mUsername = str;
51     this.mPassword = str2;
52     Authentication authentication = new Authentication();
53     authentication.createLockFile(MainActivity.this.getApplicationContext(), str + ":" + str2);
54     authentication.saveCache(MainActivity.this.getApplicationContext(), this.mUsername, this.mPassword);
55     authentication.saveLoginInfoExternalStorage(MainActivity.this.getApplicationContext(), this.mUsername, this
56 }

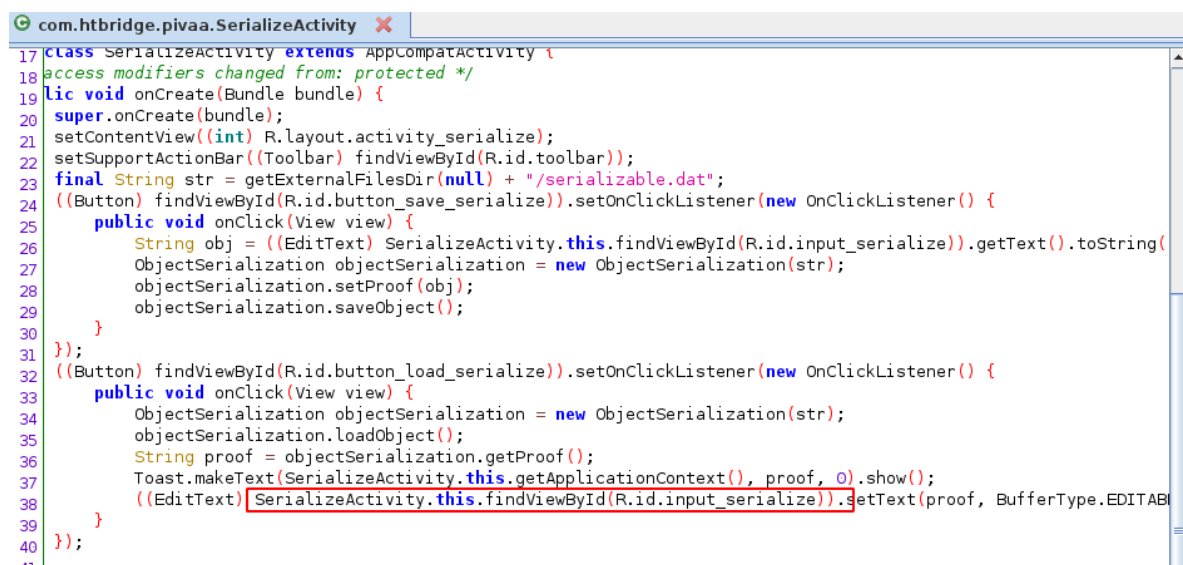
```

Recommendation :

This data should be either encrypted or stored on the server.

Object deserialization

Object deserialization performed on an untrusted resource (e.g. user-supplied input or external storage), can be dangerous if the data for deserialization is tampered by an attacker.



```

17 class SerializeActivity extends AppCompatActivity {
18     access modifiers changed from: protected */
19     public void onCreate(Bundle bundle) {
20         super.onCreate(bundle);
21         setContentView((int) R.layout.activity_serialize);
22         setSupportActionBar((Toolbar) findViewById(R.id.toolbar));
23         final String str = getExternalFilesDir(null) + "/serializable.dat";
24         ((Button) findViewById(R.id.button_save_serialize)).setOnClickListener(new OnClickListener() {
25             public void onClick(View view) {
26                 String obj = ((EditText) SerializeActivity.this.findViewById(R.id.input_serialize)).getText().toString();
27                 ObjectSerialization objectSerialization = new ObjectSerialization(str);
28                 objectSerialization.setProof(obj);
29                 objectSerialization.saveObject();
30             }
31         });
32         ((Button) findViewById(R.id.button_load_serialize)).setOnClickListener(new OnClickListener() {
33             public void onClick(View view) {
34                 ObjectSerialization objectSerialization = new ObjectSerialization(str);
35                 objectSerialization.loadObject();
36                 String proof = objectSerialization.getProof();
37                 Toast.makeText(SerializeActivity.this.getApplicationContext(), proof, 0).show();
38                 ((EditText) SerializeActivity.this.findViewById(R.id.input_serialize)).setText(proof, BufferType.EDITABLE);
39             }
40         });
41     }

```

This makes the application vulnerable to insecure deserialization which can lead to RCE usually.

Recommendation:

Usually best to not use serialization , rather use Jason format when sending data.
Sanitize user input.

Cleartext SQLite database

The mobile application uses an unencrypted SQLite database. This database can be accessed by an attacker with physical access to the mobile device or a malicious application with root access to the device. The application should not store sensitive information in clear text.

```
eActivity x com.htbridge.pivaa.DatabaseActivity x com.htbridge.pivaa.handlers.database.DatabaseHelper x
10 import java.util.ArrayList;
11
12 public class DatabaseHelper extends SQLiteOpenHelper {
13     private static final String[] COLUMNS = {KEY_ID, KEY_TITLE, KEY_AUTHOR};
14     private static final String DATABASE_NAME = "pivaaDB";
15     private static final int DATABASE_VERSION = 1;
16     private static final String KEY_AUTHOR = "author";
17     private static final String KEY_ID = "id";
18     private static final String KEY_TITLE = "title";
19     private static final String TABLE_DATA = "data";
20 }

root@kali:~/mobile/pivaa/decompile/pivaa# adb pull /data/data/com.htbridge.pivaa/databases/pivaaDB .
/data/data/com.htbridge.pivaa/databases/pivaaDB: 1 file pulled. 9.8 MB/s (20480 bytes in 0.002s)
root@kali:~/mobile/pivaa/decompile/pivaa# adb pull /data/data/com.htbridge.pivaa/databases/pivaaDB-journal .
/data/data/com.htbridge.pivaa/databases/pivaaDB-journal: 1 file pulled. 6.9 MB/s (12824 bytes in 0.002s)
root@kali:~/mobile/pivaa/decompile/pivaa# ls
AndroidManifest.xml  apktool.yml  assets  backup.ab  backup_compressed.tar  original  pivaaDB  pivaaDB-journal  res  smali
root@kali:~/mobile/pivaa/decompile/pivaa# file pivaaDB
pivaaDB: SQLite 3.x database, user version 1, last written using SQLite version 3007011
root@kali:~/mobile/pivaa/decompile/pivaa# file pivaaDB-journal
pivaaDB-journal: data
root@kali:~/mobile/pivaa/decompile/pivaa# sqlite3 pivaaDB
SQLite version 3.31.1 2020-01-27 19:55:54
Enter ".help" for usage hints.
sqlite> .tables
android_metadata  data
sqlite> select * from data
...>
1|My Vulnerable Android Application|High-Tech Bridge
2|My Vulnerable Android Application|High-Tech Bridge
3|My Vulnerable Android Application|High-Tech Bridge
4|My Vulnerable Android Application|High-Tech Bridge
5|My Vulnerable Android Application|High-Tech Bridge
6|ahmedtitle|ahmed
7|My Vulnerable Android Application|High-Tech Bridge
sqlite> █
```

Recommendations:

Dont store sensitive data on users device.

If it is necessary , Encrypt the data with a string encryption and make the keys on the server.

Missing tapjacking protection

By default, Android allows applications to draw over some portions of the phone screen and permits touch events to be sent to mobile applications in the lower 'layer'. This can be used by an attacker to trick application users into performing some sensitive actions in a legitimate application (e.g. send a payment) they do not otherwise intend doing.

Weak encryption

Weak or badly implemented encryption algorithms can endanger data storage and transmission used by the mobile application.

Hardcoded encryption keys

Hardcoded encryption keys can jeopardize secure data storage and transmission within the mobile application under certain circumstances.

Dynamic load of code

The mobile application uses dynamic load of executable code. Under certain circumstances, dynamic load of code can be dangerous. For example, if the code is located on an external storage (e.g. SD card), this can lead to code injection vulnerability if the external storage is world readable and/or writable and an attacker can access it.

Usage of unencrypted HTTP protocol

The mobile application uses HTTP protocol to send or receive data. The design of HTTP protocol does not provide any encryption of the transmitted data, which can be easily intercepted if an attacker is located in the same network or has access to data channel of the victim.

Weak hashing algorithms

The mobile application uses weak hashing algorithms. Weak hashing algorithms (e.g. MD2, MD4, MD5 or SHA-1) can be vulnerable to collisions and other security weaknesses, and should not be used when reliable hashing of data is required.

Predictable Random Number Generator

The mobile application uses predictable Random Number Generator. Under certain conditions this can jeopardize the entire data encryption performed by the mobile application.

Exported Content Providers with insufficient protection

The mobile application contains exported content providers. Content providers are used to share data between different applications. Other applications, including malicious ones, can access exported content providers. This can lead to various security issues, including information disclosure. To securely export your content provider, you can set-up 'android:protectionLevel' or 'android:grantUriPermissions' attributes in Android Manifest file in order to restrict access to your content providers.

Exported Broadcast Receivers

The mobile application contains an exported receiver enabling other applications, including malicious ones, to send intents without restrictions. By default, Broadcast Receivers is exported in Android, as the result any application will be able to send an intent to the Broadcast Receiver of the application. To define which applications can send intents to mobile application's Broadcast Receiver set relevant permissions in the Android Manifest file.

Exported Services

The mobile application contains an exported service. By default, in Android services are not exported and cannot be invoked by other applications. However, if an intent filter is defined in Android Manifest file, it is exported by default. Particular attention should be given to the exported services, as without the specific permissions, they can be used by any other applications including malicious ones.

Deprecated setPluginState in WebView

The mobile application uses a deprecated setPluginState method in WebView. The 'setPluginState' method was deprecated in Android's API level 18 because plugins will not be supported anymore in the future and should not be used. Example of plugins are embedded PDF reader, Flash plugin, etc.

Temporary file creation

The mobile application creates temporary files. Despite that cache files are usually private by default, it is recommended to make sure that temporary files are securely deleted when they are not required by the application anymore.

Untrusted CA acceptance

The mobile application may allow some untrusted Certificate Authorities (CA) to be used. TrustManager allows developers to accept certificates not trusted by Android. This feature can tremendously facilitate MiTM attacks.

Usage of banned API functions

The mobile application uses some of the banned API functions. API functions are usually banned for compelling security and privacy reasons and shall not be used.

Self-signed CA enabled in WebView

The mobile application's WebView accepts self-signed and otherwise untrusted certificates. This may create a risk of Man-in-the-Middle (MITM) attacks under many circumstances.

Usage of weak Initialization Vector

When Cipher Block Chaining (CBC) or Cipher Feedback (CFB) modes are used in encryption, the Initialization Vector must be unpredictable and random.

Possible Man-In-The-Middle Attack

Improper or missing hostname verification exposes mobile application users to MITM attacks under certain conditions.