Übung 4 - Android

Student: Philip Magnus

1 Lab + App Network Traffic Inspection/Interception Setup

The steps in this writeup were performed on a Windows 11 (x64) system.

1.1 Lab environment using a Android device and/or emulator (e.g. Android Emulator)

The following steps describe the setup of the needed lab environment. For ease of use I choose to setup an android emulator via `Android Studio`.

First `Android Studio` was installed via the JetBrains Toolbox.

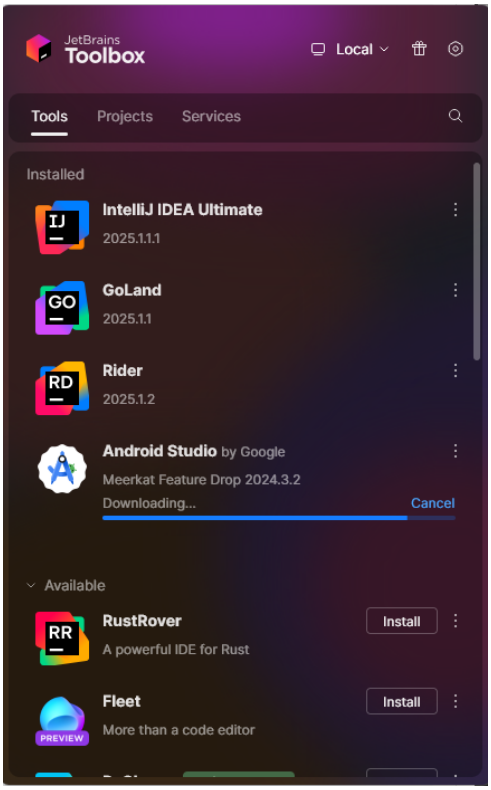


Figure 1: Install Android Studio

In `Android Studio` we open the `Virtual Device Manager` and open the setup wizard for creating a new virtual device.

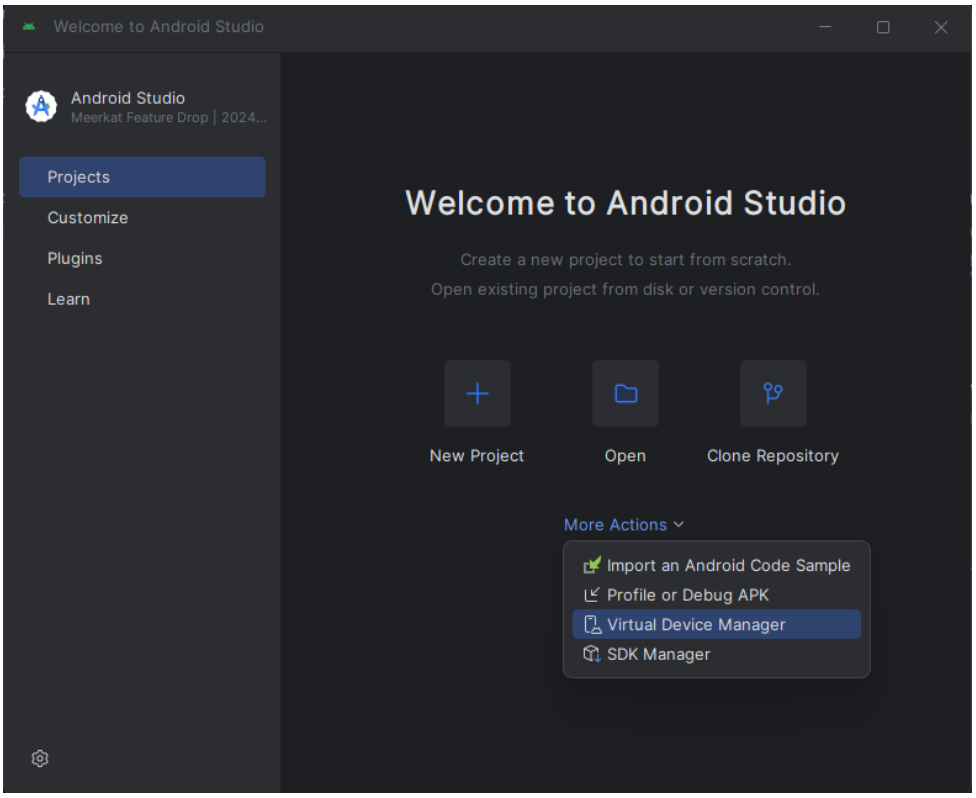


Figure 2: VirtualDev Manager

In the device setup I chose to use the Google APIs for an easier setup of `httptoolkit` later on. Android 15 was chosen as the Android version to install on the virtual device.

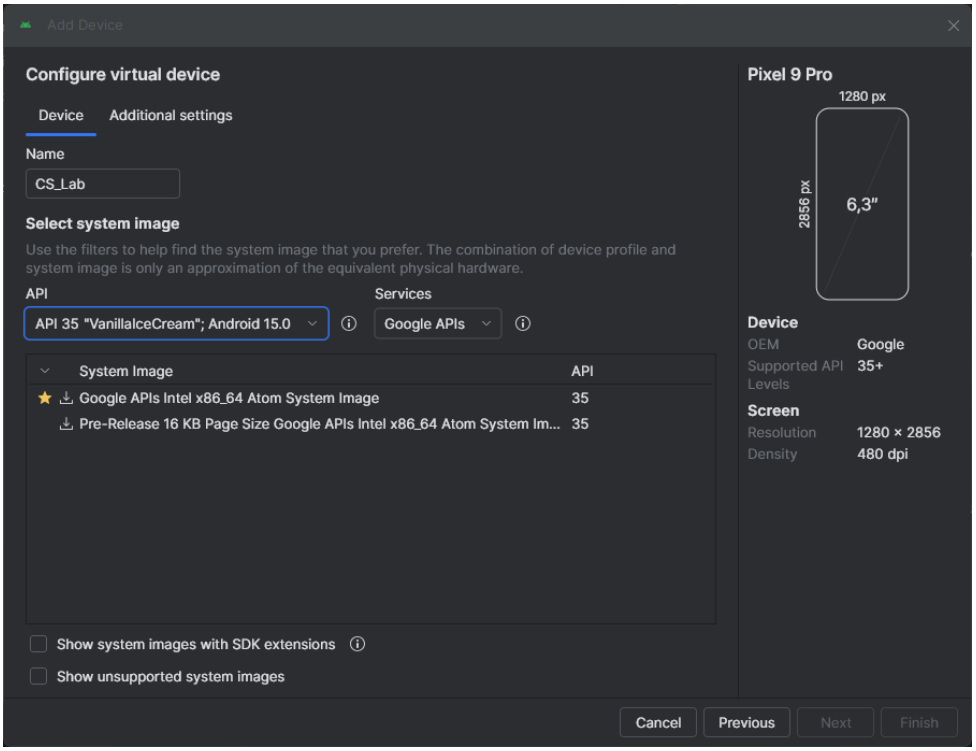


Figure 3: Device APIs

As the system image `Google APIs Intel x86_64 Atom System Image` was used. For our scenario there is no need for the pre-relese image. With clicking the `Finish` button the installation of the device started.

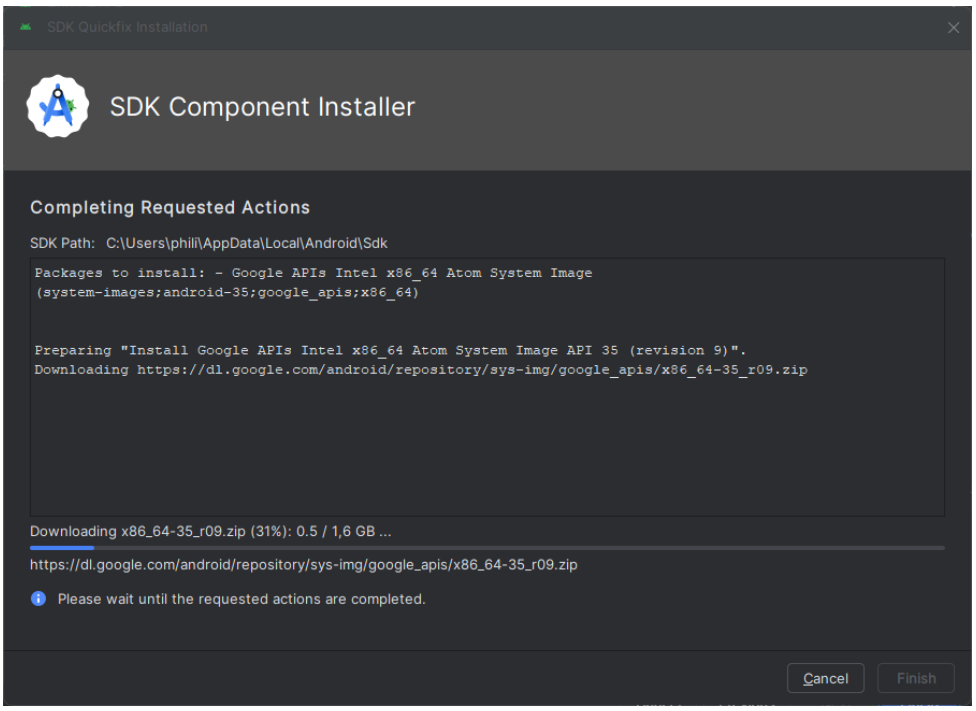


Figure 4: Emulator Install

After the installation the "fresh" virtual device can be started simply via the virtual device manager. After starting the device I was greeted by an almost empty home screen.

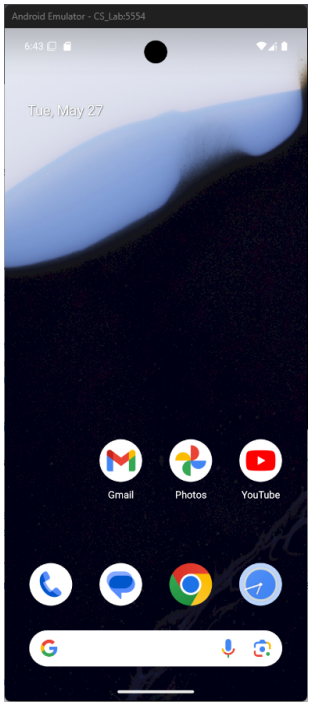


Figure 5: Fresh device emulator

1.2 Choose an arbitrary app with HTTP(S) backend communication

As the target app I chose the `ORF.at News` app. The app is currently in version 1.99.37 and needs at least Android 5, or the `Lollipop, API 21`, installed.

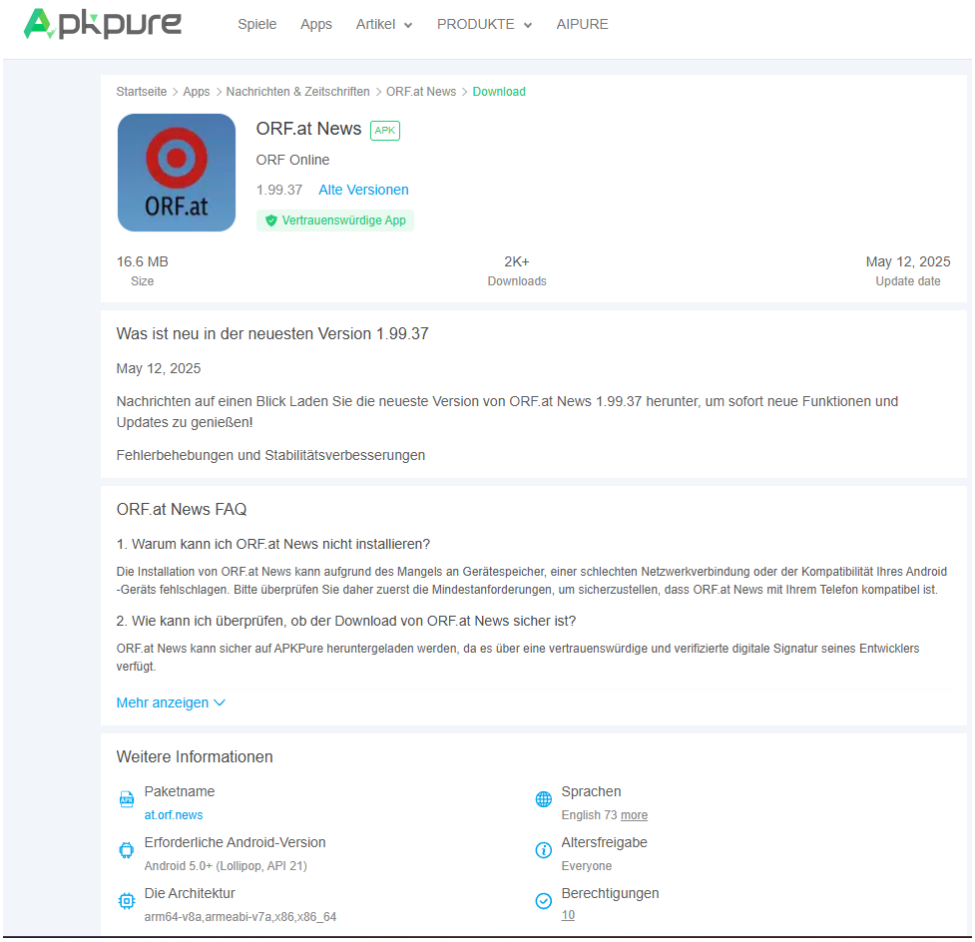


Figure 6: ApkPure

Installing the app is very simple and can be achieved by dragging the app onto the virtual device. The app is then automatically installed and can be used from there.

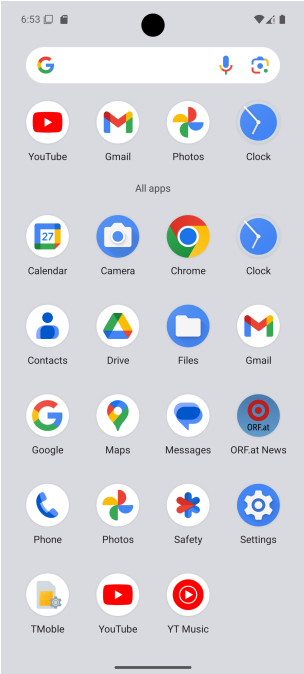


Figure 7: ORF.at

Next I installed `httptoolkit`. After the installation under intercept I chose the `Android device via ADB` setup. The setup completes pretty much automatically and lets me sniff any http/https traffic on the device.

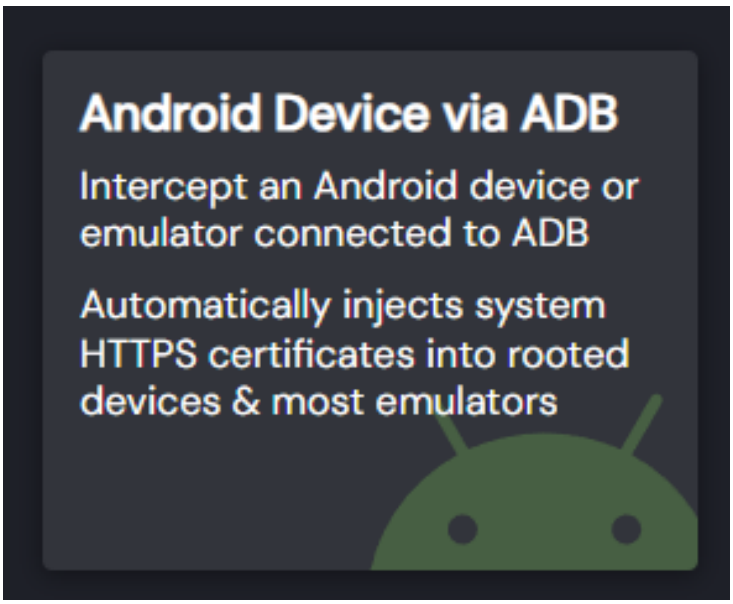


Figure 8: via ADB

During the setup I only needed to approve `httptoolkit`'s access only once on the virtual device.

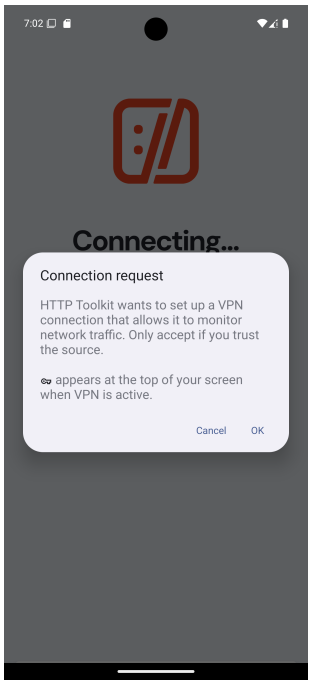


Figure 9: approve toolkit

After the approval I was able to see all traffic on the device. For example opening a random article in the `ORF.at News` app resulted in quiet some observable http traffic.

Figure 10: article in app

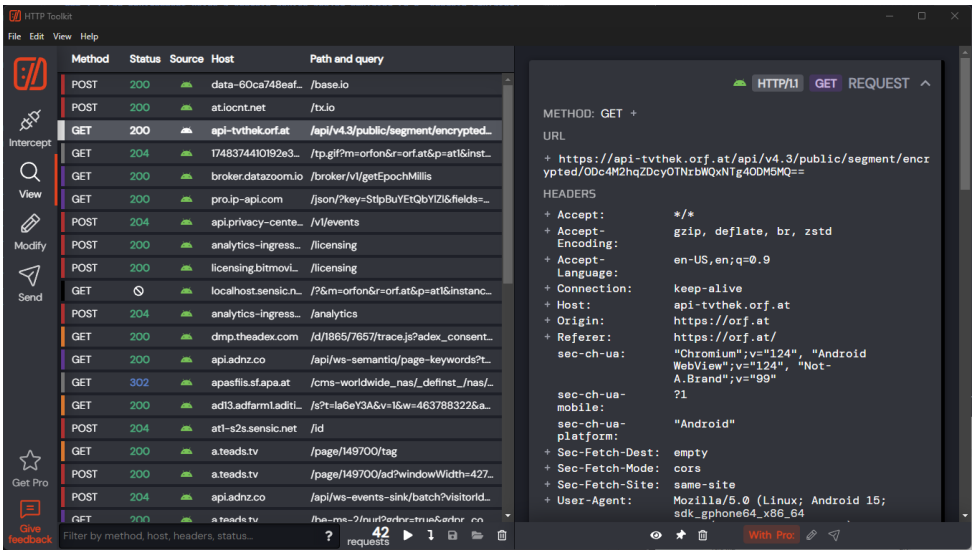


Figure 11: http toolkit traffic

2 Perform Static and Dynamic Analysis

The following steps describe the static and dynamic analysis of the app. The goal is to find out how the app communicates with the backend and what data is sent/received.

2.1 Static: inspect the source code to analyze the HTTP(S) implementation

2.1.1 Decompile the application, e.g. using Bytecode Viewer, APK Editor Studio, apktool, JADX, Ghidra, etc.

First I downloaded the `JADX` software from the official Github repository. After extracting the downloaded archive, `JADX` can be started through the included .exe file.

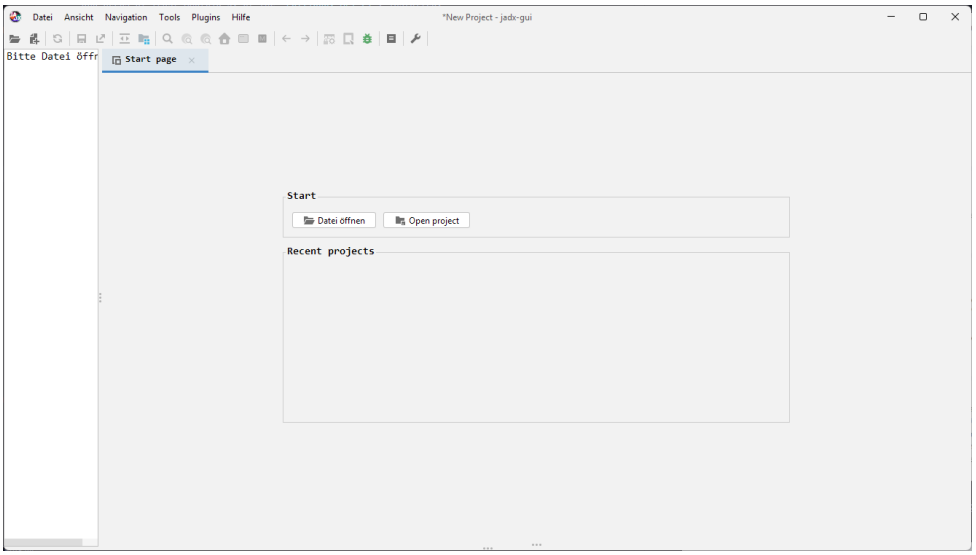


Figure 12: JADX start

By clicking the `Open File` I was able to select the file to decompile.

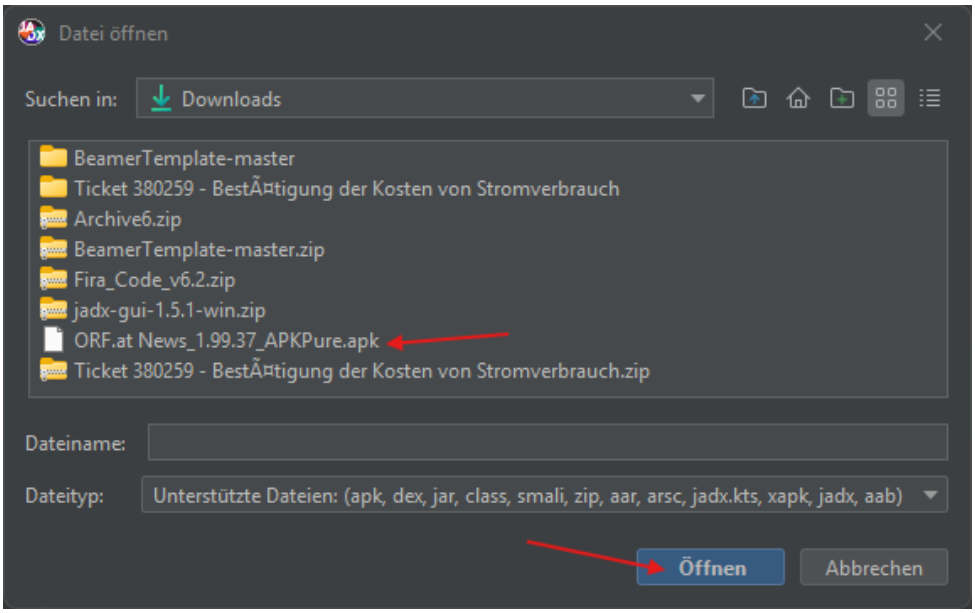


Figure 13: Opening APK

After opening the file, `JADX` started decompiling the app. The decompilation took a few seconds and afterwards I was able to browse through the code of the app.

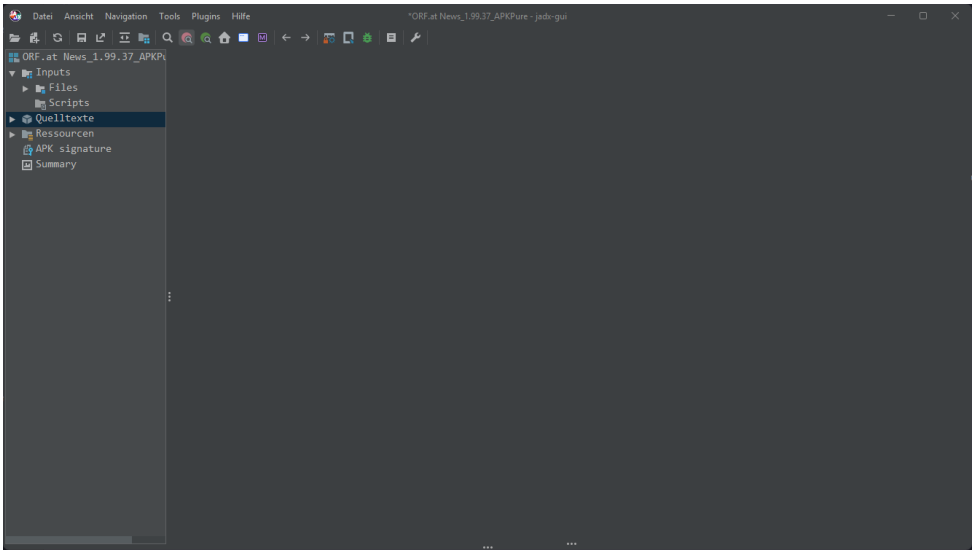


Figure 14: Decompiled code

2.1.2 How is HTTP(S) is implemented?

In the decompiled code are several hints on how the app communicates with the backend. The app uses the `Retrofit2` and `OkHttp` library to handle HTTP requests. The app seems to use `Retrofit2` for making https calls. `Retrofit2` is a type-safe HTTP client which in turn implements the `OkHttp` library to handle the actual network requests.

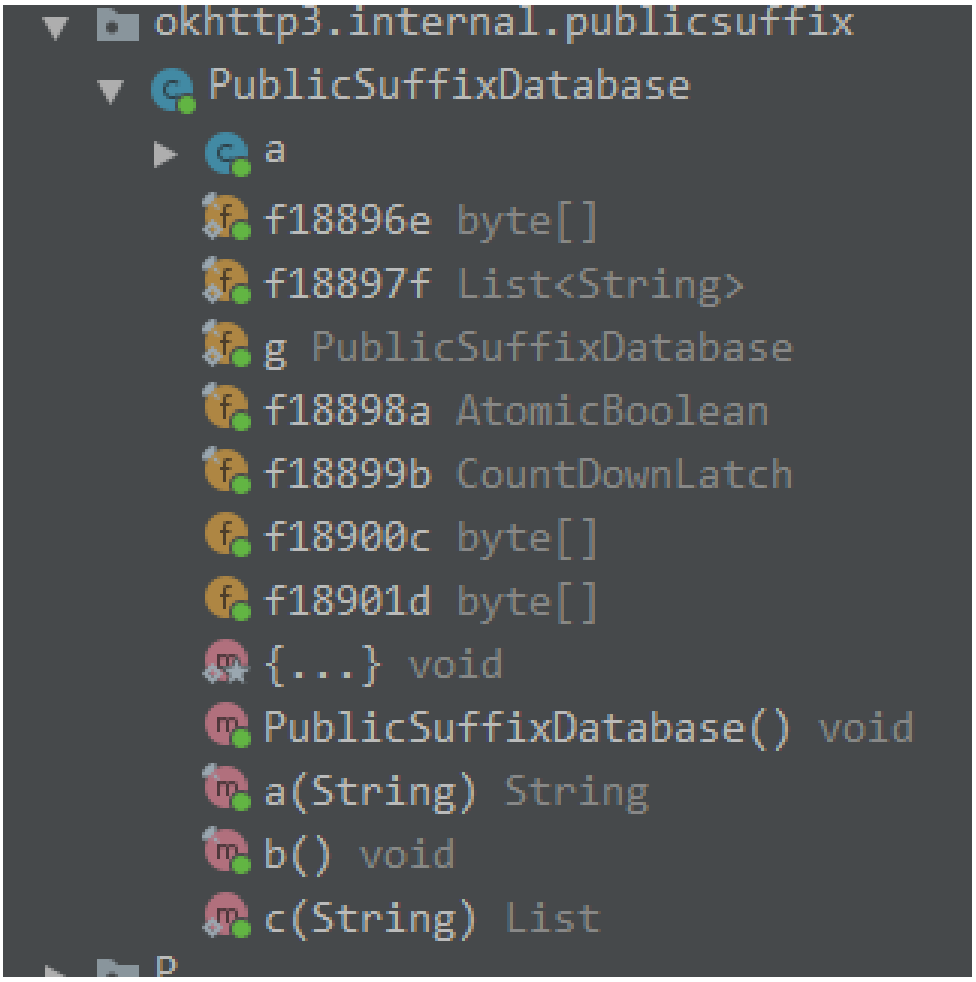


Figure 15: OkHttp

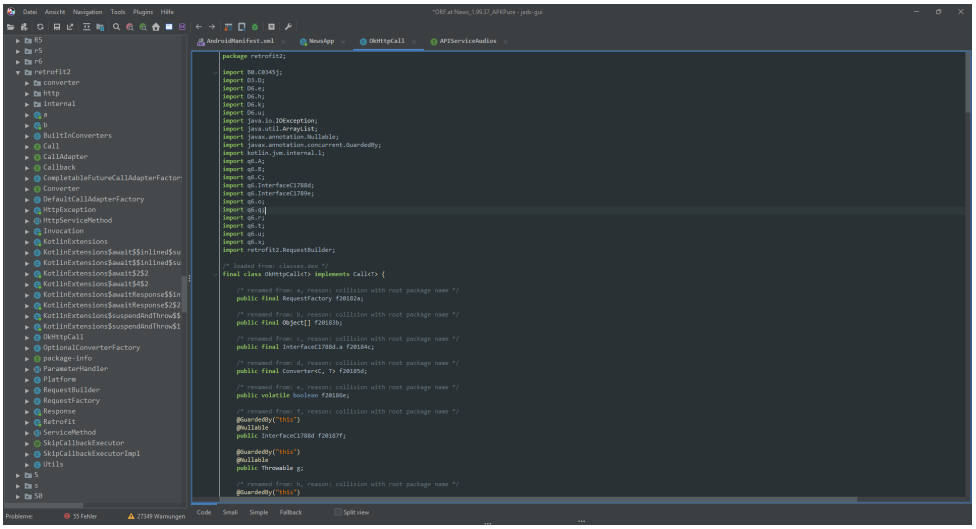


Figure 16: Retrofit2 decompilation

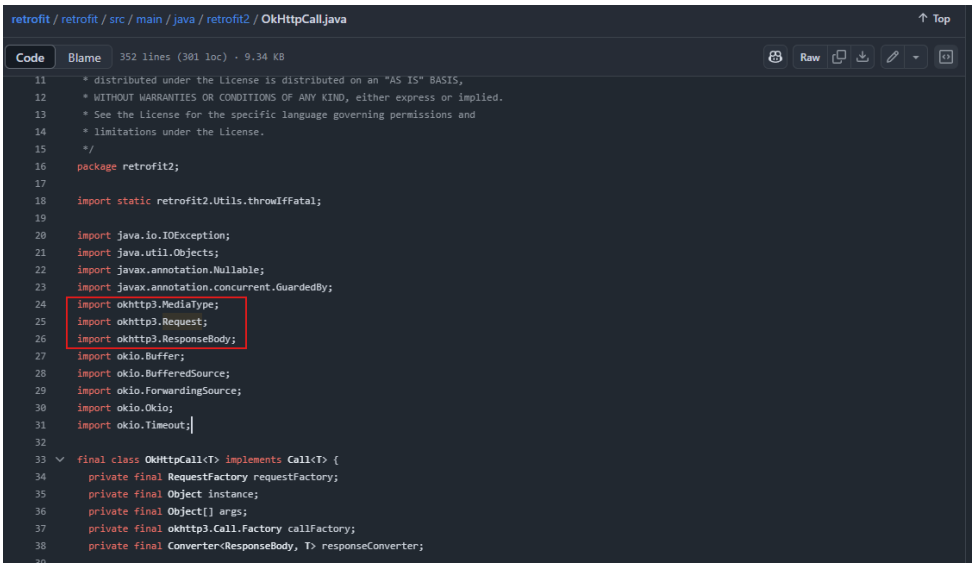


Figure 17: Retrofit2 OkHttp implementation

The app does not only use `Retrofit2` and `OkHttp` for network requests, it also uses `android-async-http` from `loopj` for some network requests. `android-async-http` in turn implements the `httpclient-android` library to handle the actual network requests. This library is a port of the Apache HttpClient to Android.

The following code snippet shows the usage of `android-async-http` in the decompiled code:

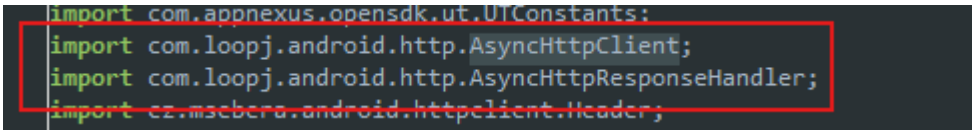
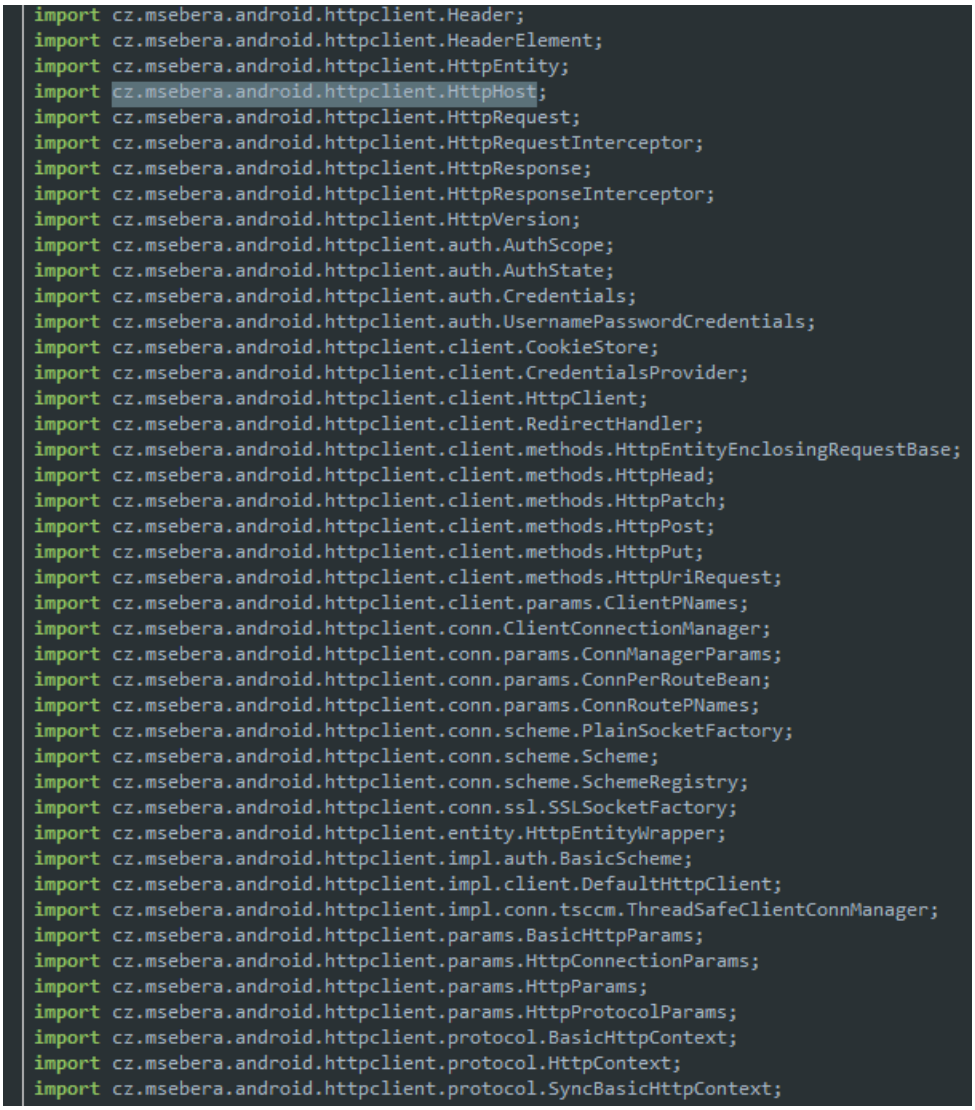


Figure 18: AsyncHttpClient

As we can see in the code snippet, the `httpclient-android` library is imported and used to make network requests.



```
import cz.msebera.android.httpclient.Header;
import cz.msebera.android.httpclient.HeaderElement;
import cz.msebera.android.httpclient.HttpEntity;
import cz.msebera.android.httpclient.HttpHost;
import cz.msebera.android.httpclient.HttpRequest;
import cz.msebera.android.httpclient.HttpRequestInterceptor;
import cz.msebera.android.httpclient.HttpResponse;
import cz.msebera.android.httpclient.HttpResponseInterceptor;
import cz.msebera.android.httpclient.HttpVersion;
import cz.msebera.android.httpclient.auth.AuthScope;
import cz.msebera.android.httpclient.auth.AuthState;
import cz.msebera.android.httpclient.auth.Credentials;
import cz.msebera.android.httpclient.auth.UsernamePasswordCredentials;
import cz.msebera.android.httpclient.client.CookieStore;
import cz.msebera.android.httpclient.client.CredentialsProvider;
import cz.msebera.android.httpclient.client.HttpClient;
import cz.msebera.android.httpclient.client.RedirectHandler;
import cz.msebera.android.httpclient.client.methods.HttpEntityEnclosingRequestBase;
import cz.msebera.android.httpclient.client.methods.HttpHead;
import cz.msebera.android.httpclient.client.methods.HttpPatch;
import cz.msebera.android.httpclient.client.methods.HttpPost;
import cz.msebera.android.httpclient.client.methods.HttpPut;
import cz.msebera.android.httpclient.client.methods.HttpUriRequest;
import cz.msebera.android.httpclient.client.params.ClientPNames;
import cz.msebera.android.httpclient.conn.ClientConnectionManager;
import cz.msebera.android.httpclient.conn.params.ConnManagerParams;
import cz.msebera.android.httpclient.conn.params.ConnPerRouteBean;
import cz.msebera.android.httpclient.conn.params.ConnRoutePNames;
import cz.msebera.android.httpclient.conn.scheme.PlainSocketFactory;
import cz.msebera.android.httpclient.conn.scheme.Scheme;
import cz.msebera.android.httpclient.conn.scheme.SchemeRegistry;
import cz.msebera.android.httpclient.conn.ssl.SSLSocketFactory;
import cz.msebera.android.httpclient.entity.HttpEntityWrapper;
import cz.msebera.android.httpclient.impl.auth.BasicScheme;
import cz.msebera.android.httpclient.impl.client.DefaultHttpClient;
import cz.msebera.android.httpclient.impl.conn.tsccm.ThreadSafeClientConnManager;
import cz.msebera.android.httpclient.params.BasicHttpParams;
import cz.msebera.android.httpclient.params.HttpConnectionParams;
import cz.msebera.android.httpclient.params.HttpParams;
import cz.msebera.android.httpclient.params.HttpProtocolParams;
import cz.msebera.android.httpclient.protocol.BasicHttpContext;
import cz.msebera.android.httpclient.protocol.HttpContext;
import cz.msebera.android.httpclient.protocol.SyncBasicHttpContext;
```

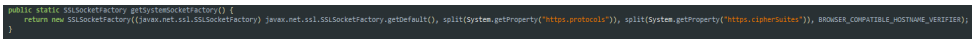
Figure 19: httpclient-android import

2.1.3 Which TLS versions and cipher suites are supported?

Accordrding to the official `OkHttp` documentation "OkHttp supports modern TLS features (TLS 1.3, ALPN, certificate pinning)". Given that the app uses `OkHttp` to handle the network requests, it can be assumed that the app supports TLS 1.3 and the cipher suites provided by `OkHttp`.

For more information on the supported cipher suites, the official Github repository can be consulted.

For requests made with `android-async-http` the supported TLS versions and cipher suites depend on the underlying `httpclient-android` library. In the implementation I could not find any specific supported TLS version but the library uses the Cypher Suites provided by the Android system.



```
public static SSLSocketFactory getSystemSocketFactory() {
    return new SSLSocketFactory((javax.net.ssl.SSLSocketFactory) javax.net.ssl.SSLSocketFactory.getDefault(), split(System.getProperty("https.protocols")), split(System.getProperty("https.cipherSuites")), BROWSER_COMPATIBLE_HOSTNAME_VERIFIER);
}
```

Figure 20: System Cipher Suites

2.1.4 Is a Network Security Configuration (NSC) used? Is Certificate Pinning implemented?

Neither a Network Security Configuration (NSC) nor Certificate Pinning was found in the decompiled code. The app seems to rely on the default behavior of `OkHttp` and `Retrofit2` for handling SSL/TLS connections.

I came to this conclusion because the necessary configurateion files for NSC were not present in the decompiled code. The quickest way to check for the presence of a NSC is to look for a file named `network_security_config.xml` in the `res/xml` directory of the decompiled app.

Figure 21: No NSC found in AndroidManifest.xml

The app was written with an API level of 21 in mind, if I am not mistaken this means that the app does not support a Network Security Configuration (NSC), as NSC was introduced in API level 23.

2.2 Dynamic: inspect generated network traffic & analyze communication to backend servers

2.2.1 Is any sensitive information sent to the backend server?

After generating some traffic within the app and anylzing it with `httptoolkit`, I was not able to find any sensitive information being sent to the backend server. The app seems to only send the necessary data for fetching news articles and other related content.

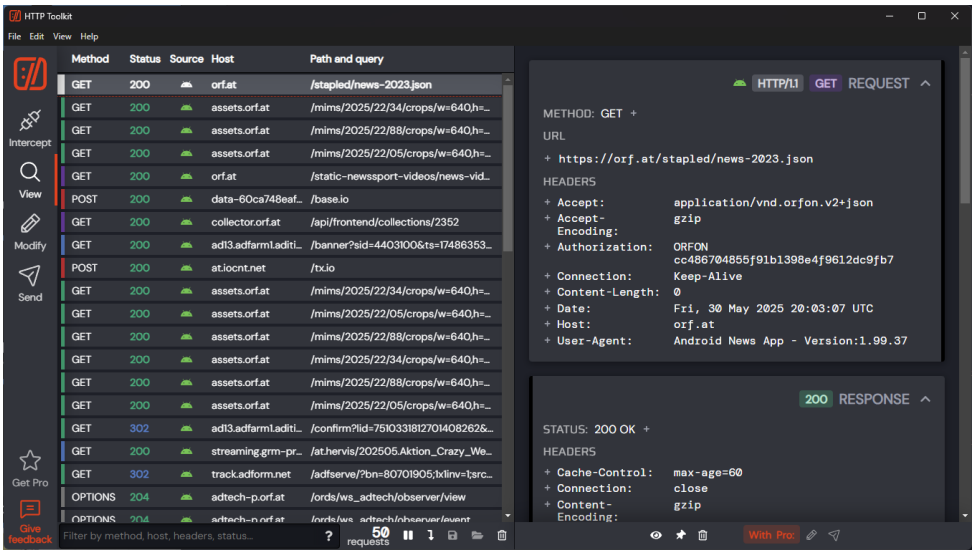


Figure 22: Traffic analyzation

2.2.2 Does the app collect analytics?

The app does seem to collect some analytics data, but it is not clear what exactly is being collected. The app does not seem to collect any sensitive information, but it does

send some data to the backend server for analytics purposes. By looking at the traffic with some educated guessing, I assume the app collects some analytics data for serving advertisements to the user.

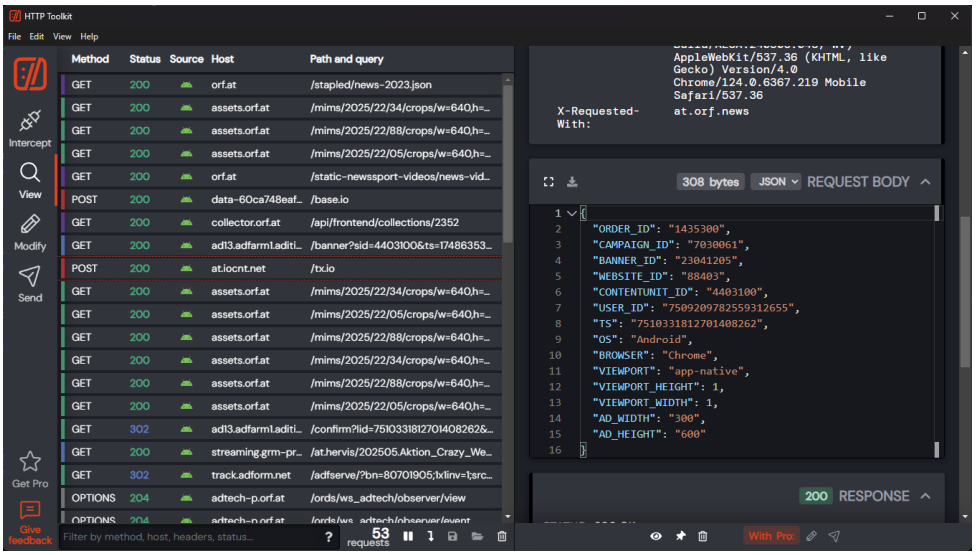


Figure 23: Ad Tracking

Some other minor analytics like the android version are send to the backend probably to ensure compatibility with the app.

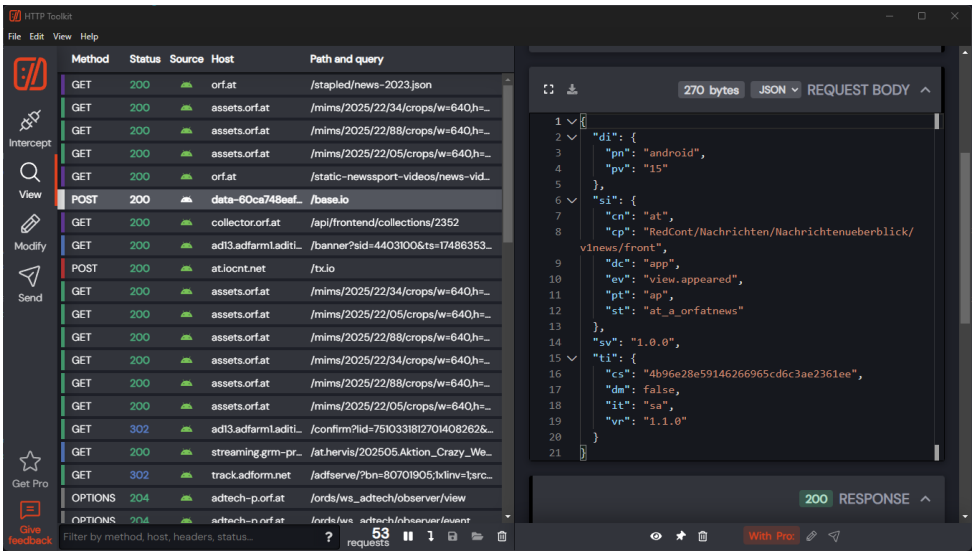


Figure 24: Minor Tracking

In the decompiled code I found some configuration files hinting at a `Firebase` integration. The app seems to send some analytics data to `Firebase` for further processing, although I was not able to see any live traffic confirming this theory.

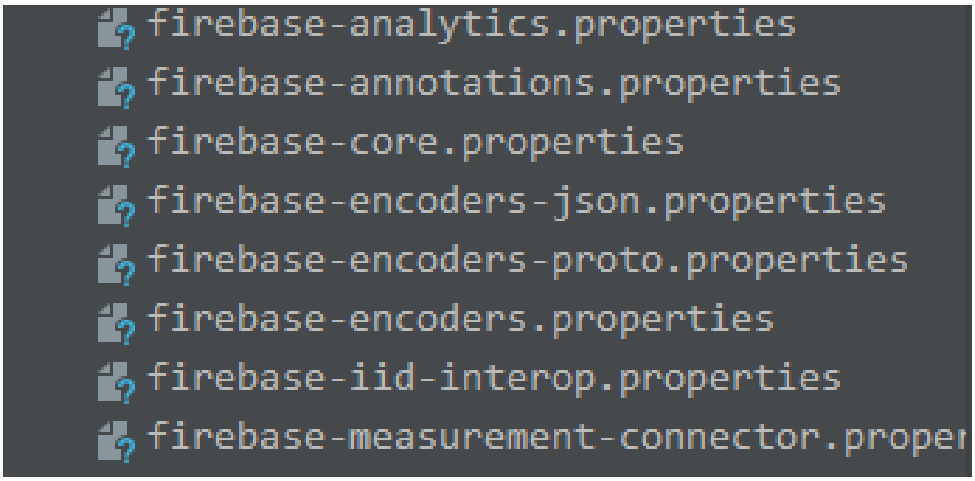


Figure 25: Firebase integration

2.2.3 Are there any hard-coded secrets in the app (i.e. authorization tokens, etc.)?

One secrets seem to be hard-coded in the app. The app does use a hardcoded API auth-token for accessing the backend server.

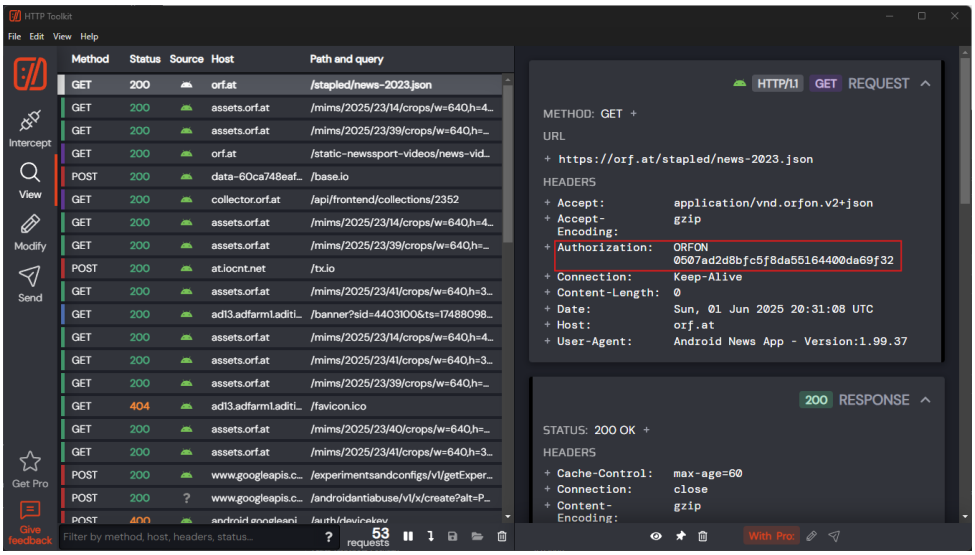


Figure 26: Auth Header

The following code snippet shows the usage of the hardcoded API auth-token in the decompiled code:

```
public final void p(StoryScreen storyScreen, String str) {
    this.asyncHttpClient.removeHeader(HttpHeaders.ACCEPT);
    this.f8369C0.removeHeader(HttpHeaders.ACCEPT);
    this.f8372D0.removeHeader(HttpHeaders.ACCEPT);
    this.asyncHttpClient.addHeader(HttpHeaders.ACCEPT, "application/vnd.orfon.v2+json");
    this.f8369C0.addHeader(HttpHeaders.ACCEPT, "application/vnd.orfon.v2+json");
    this.f8372D0.addHeader(HttpHeaders.ACCEPT, "application/vnd.orfon.v2+json");
    if (A.isInternetAvailable(this)) {
        this.f8369C0.cancelAllRequests(true);
        Locale locale = Locale.US;
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z", locale);
        simpleDateFormat.setTimeZone(DesugarTimeZone.getTimeZone("UTC"));
        String format = simpleDateFormat.format(new Date());
        this.f8369C0.addHeader("Date", format);
        AsyncHttpClient asyncHttpClient = this.f8369C0;
        StringBuilder sb = new StringBuilder("ORFON ");
        sb.append(a(str + format + "92945e642bbc8c90dddb0c971d74f996").toLowerCase(locale));
        asyncHttpClient.addHeader("Authorization", sb.toString());
        this.f8369C0.get(getApplicationContext(), str, new C2034b(this, storyScreen));
    }
}
```

Figure 27: Hardcoded Auth Token

Beyond that I was not able to find any hard-coded secrets in the app. The app seems to rely on the backend server for further authentication and does not store any sensitive information locally.

2.2.4 Any other interesting findings in the communication (e.g. in regard to OWASP Mobile Top 10)?

Regarding the OWASP Mobile Top 10, I did not find any major issues in the app. The most interesting part would concern the apps supply chain security. This would fall under `M2: Inadequate Supply Chain Security`. Some of the libraries in used in the app are rather old and have not been updated in quite some time, sometimes years. This could lead to potential security issues in the future, as the libraries might not be maintained anymore and could contain vulnerabilities.

If those packages have been downloaded by another library or were purposefully integrated into the app is not clear.

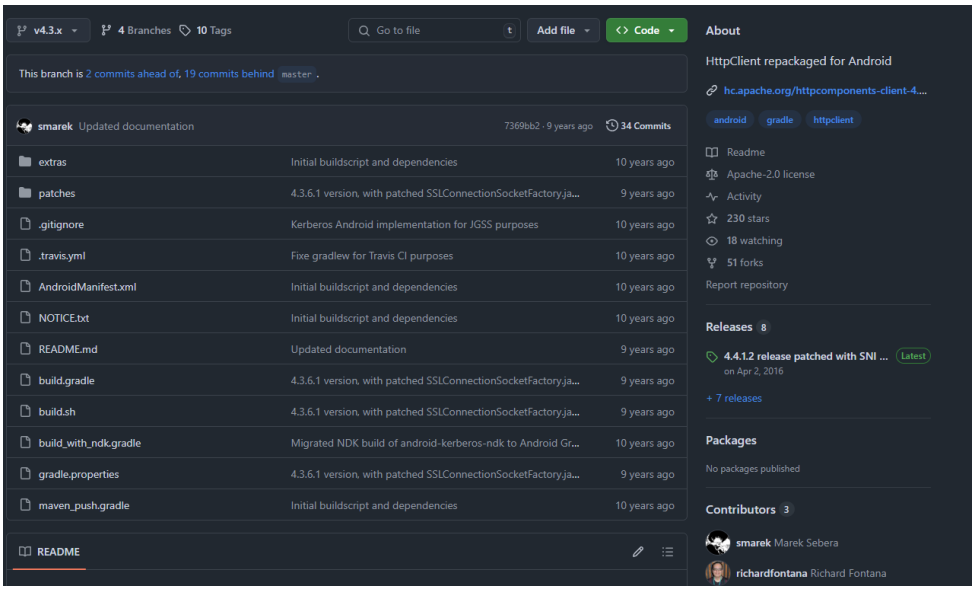


Figure 28: Unmaintained repository - httpclient

3 Conclusion

In conclusion, the app does not seem to have any major security issues. The app uses `OkHttp` and `Retrofit2` for handling network requests, which are both well-known libraries in the Android ecosystem. The app does not use a Network Security Configuration (NSC) or Certificate Pinning, also it does not seem to send any sensitive information to the backend server. It should be noted that NSC was introduced in API level 23, which is higher than the minimum API level of the app (21). They also should consider switching to libraries that are actively maintained.