# Security Analysis of Android application

System Security

Morten Lyng Rosenquist
*Faculty of Technical Sciences*
*Aarhus University*
Aarhus, Denmark
201706031

December 13, 2021

***Abstract*—In this report a preliminary security analysis is performed on an Android mobile application. GulOgGratis, a public Marketplace, is the targeted application. Several security properties are essential when managing such a platform. The users' data needs to be handled properly and the application must respect the users' privacy. Several improvements was found by investigating decompiled source code and network traffic. Measures were not taken to harden adversaries gaining insight of the applications functionality. However, no severe vulnerability was found. Users' actions are monitored heavily and primarily used to personalize advertisements.**

***Index Terms*—Security Analysis, Android Security, Software Security, Network Security, man-in-the-middle**

## I. Introduction

This report will document a preliminary security analysis of a mobile application. The application will be looked at from four various aspects: *Software Security*, *Network Security*, *Authentication* and *Privacy*. Four areas that is all key in keeping an application secure. The system that is analysed is *GulOgGratis* which is an online marketplace targeting the Danish market. Establishing such a community requires storing a significant amount of data where some will have a sensitive nature. Users create their accounts and publishes an item they want to sell. Others then communicate with the seller and come to an agreement. GulOgGratis integrates with a range of other services such as; NemID, Parcel Services, Analytical Services, Social Platforms and Ad Services. The marketplace can be accessed either as a website or mobile application. The mobile application is both available for iOS and Android. This report will solely focus on the android application and it's dependencies.

### A. Threat Modelling

Modelling the threats for the application we look at four key points. *Threat Model*, whom are we protecting against. *Attack Surface*, which parts of the system can be attacked. *Policies*, which security properties are to be provided. *Mechanisms*, how are the security properties provided. The adversaries are competitors looking for a competitive edge, insiders and criminals with malicious intents. Their attack vectors are the application binaries, access to users accounts, network traffic and the data stored in various cloud services. The threat modelling is as follows:

**Threat Model:** competitors, insiders, criminals
**Attack Surface:**

- Application: decompilation, evil version and broken access
- Network Traffic: man-in-the-middle, eavesdropping, manipulation and dos
- Integrations: over-sharing, data loss and leakage

**Expected Policies:** data authenticity/confidentiality/integrity, privacy and availability
**Expected Mechanisms:** code obfuscation, access control, secure networking (pinning / certificate transparency), use of trustworthy integrations, permission handling

The mentioned policies and mechanisms are what is expected from the application. A plausible attack could be an attacker getting access to an user's account. This could be achieved by investigating decompiled code and then knowing which traffic to eavesdrop when the user tries to authenticate. If the attacker is able to mount such an attack, he is capable of retrieving all data of the user and perform actions on the user's behalf. Another attack could revolve around the user's privacy. Users' actions are likely heavily monitored. As this will be stored in the cloud, it creates a privacy concern for the user, if these actions can be linked to him. The data can be accessed by some insider, and he might be able to see sensitive data such as his location.

## II. Software Security

### A. Static Analysis

The first step is understanding the structure and behaviour of the application. As the source code is not publicly available we must be satisfied with the second-best option which is decompilation. There is a range of tools to decompile an android application, the one we will use is jadx[1]. The newest apk is retrieved from apkpure[2]. The apk is then decompiled using jadx resulting in two folders, one with source code and the other with resources. The source code folder contains all the logic of the application together with the wide range of libraries used. While the resource folder contains static resources such as images, xml configuration files, hard-coded strings, etc.

It is not easy to identify which parts should be investigated in such a decompilation. Firstly a manual search was conducted searching for keywords. The keywords was based on; use of cryptographic algorithms, secrets, keys or passwords. Nothing interesting resulted from the manual search. Then the source code was looked at to understand the flow and structure. It was clear that no apparent use of obfuscation is used. The code is both easy to read and understand. An example can be seen on Figure 1.

```
public class GGApplication extends MultiDexApplication {
    private static final String TAG = "dk.guloggratis.GGApplication";
    private AdManager adManager;
    private AppRatingChecker appRatingChecker;
    private AppStatusModel appStatusModel;
    private Map<String, String> mErrorMap;
    private GGRestApi mRestService;
    private AppSettings mSettings;
    private GGRestApi mSyncRestService;
    private Tealium tealium;
    private ZendeskInitializer zendeskInitializer;
```

Fig. 1: Part of GGApplication class

This is a snippet from the start of a class, that initializes services and makes them available across the application. There is neither use of name obfuscation nor string encryption. As a consequence it is easy for an adversary to understand the code flow and in this case identify some of the services used in the application.

To ease the static analysis MobSF[3] is used to identify known flaws and vulnerabilities. MobSF gives the application a security score of 35/100, which might indicate that there is some vulnerabilities. Looking at the report there are several configurations, that is tagged as medium or high severity. One is the allowance of application data backup. This is a flag that defaults to true. If sensitive data is stored this should be considered actively set to false. The code analysis shows and extensive amount of logging. Examples of this can been seen in a search on Figure 2.

These log statements are called in callback functions from an external library containing Dibs logic. Dibs is basically

```
Log.v(str, "PaymentAccepted: " + map.toString());
Log.v(str, "paymentCancelled: " + map.toString());
Log.v(TAG, "paymentWindowLoaded");
Log.v(TAG, "cancelUrlLoaded");
Log.v(TAG, "failedLoadingPaymentWindow");
```

Fig. 2: Search of log statements

NETS which is used to perform payments. These log statements can contain sensitive payment information and should not be included in a release build. A way to avoid this is using timber[4]. Timber is a wrapper that enables the possibility of only enabling log statements in debug builds. Timber is used for some parts of the application but neglected in other parts.

Another potential flaw is seen in the Dibs library. The library utilizes a webViewClient, which is embedding a browser in the application. The code can be seen on Figure 3. The code enables Javascript in the embedded browser and adds an interface to Java code. This is a known vulnerability to all android sdk versions below 17[3]. This is not an issue, since the application has a minimum sdk version of 21.

```
this.b.getSettings().setJavaScriptEnabled(true);
this.b.setWebViewClient(new a());
if (!this.g) {
    this.b.addJavascriptInterface(this.c, "Android");
    this.b.setWebViewClient(new WebViewClient() { //
```

Fig. 3: Dibs library embedded webview

Another webview vulnerability is seen the Zendesk library. This can be seen on figure 4. Here the webview allows debugging. This means it's possible to attach an debugger and see/control the execution of the webview.

```
if (Build.VERSION.SDK_INT >= 19) {
    WebView.setWebContentsDebuggingEnabled(true);
}
```

Fig. 4: Zendesk library embedded webview

The report from MobSF as well display the use of SSL pinning. This can be seen on figure 5 and will be analysed later in the attempt of a man-in-the-middle attack.

| | | | |
|---|---|---|---|
| This App uses SSL certificate pinning to detect or prevent MITM attacks in secure communication channel. | secure | **CVSS V2:** 0 (info) **OWASP MASVS:** MSTG-NETWORK-4 | zendesk/core/ZendeskNetworkModule.java<br>zendesk/support/SupportSdkModule.java<br>dk/guloggratis/GGApplication.java<br>dk/guloggratis/utils/NetworkUtils.java |

Fig. 5: Usage of SSL pinning

[1] https://github.com/skylot/jadx

[2] https://apkpure.com/guloggratis/dk.guloggratis - at the time of the analysis the newest version was 2.5.4, but version 5.0.1 was released November 25.

[3] https://github.com/MobSF/Mobile-Security-Framework-MobSF

[4] https://github.com/JakeWharton/timber

### B. Dynamic Analysis

A dynamic analysis was performed using MobSF. A emulator with sdk version 29 is spawned locally using genymotion[5]. MobSF then installs an instrumented version with configurations of our choice. The defaults are selected which includes: installing root certificate, setup https proxy, SSL pinning bypass, etc. Then some execution flows are performed on the device and a report is generated. It's then possible to see various information of the execution. The network traffic can be seen. These requests are sent to Burp Suite[6], and will be analyses in the next section. The host names and their geolocation can be seen together with trackers. It is also possible to see log statements. A example can be seen on figure 6. It shows that URLs together with query parameters are logged.

```
12-09 04:36:47.127 13843 13941 D
dk.guloggratis.GGApplication: execute:
https://api.guloggratis.dk/modules/gg_app/user/
user_info?
token=1c0752690b542f45d871a371521aa1f313b5dae3&
device=android&
version=2.5.4&
device_model=Samsung+Galaxy+S10&
operation_system=10\n'
```

Fig. 6: Logging HTTP request

The generated report also shows the local files stored in the applications own data directory. An example is a XML file stored in SharedPreferences. A snippet of the file can be seen on figure 7. It is seen that the token stored is the one used in HTTP requests.

```
<map>
    <string name="pref_user_phone2_countrycode">DK</string>
    <string name="pref_user_address">Borgergade 4 1</string>
    <int name="pref_ad_price" value="100" />
    <string name="pref_username">MRosenquist</string>
    <string name="pref_old_product_identifier">free</string>
    <string name="pref_session_token">1c0752690b542f45d871a371521aa1f313b5dae3</string>
```

Fig. 7: Part of xml preference file

## III. Network Security

In the dynamic analysis the host names were reported. The application has many external dependencies, but the domain we are primarily interested in is, *api.guloggratis.dk*. This is the api responsible of the business logic and data. The server's TLS configuration is analysed with Qualys SSL Labs[7]. The overview can be seen on figure 8. It is seen the server supports old version of TLS, weak ciphers and handshakes. The certificate is issued by Cloudflare and it is only domain validated. Meaning it only proves they are the owner of the site but there is no organization information required to achieve such a certificate.

[5]https://www.genymotion.com/
[6]https://portswigger.net/burp
[7]https://www.ssllabs.com

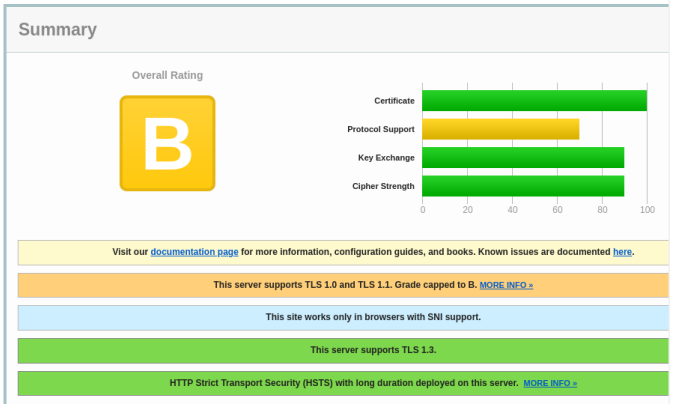


Fig. 8: TLS configuration of server

An attempt of performing a man-in-the-middle attack will be conducted. This revolves around setting us between the application and the server which it is communicating with. This was already performed in the dynamic analysis using MobSF. Instead of using MobSF we will do the instrumentation and configuration ourselves to fully understand the behaviour. The setup consists with the same emulator as before but now with Burp Suite as the proxy. Firstly, we install the apk, disable data and proxy the Wi-Fi to Burp Suite. Then the app is started and resulting in the TLS handshake failing. This is because the phone does not trust Burp Suite's certificate. The certificate is then installed in the emulators trust store. Then traffic can be intercepted in the emulators browser but the TLS connection still fails in the application. This is caused by either SSL pinning or certificate transparency. Remembering from the static analysis MobSF showed that this was present. Delving into the decompiled code it is seen that certificate transparency is used for the main server. The code can be seen on figure 9

```
private Retrofit getRetrofitBuilder(OkHttpClient.Builder builder) {
    return new Retrofit.Builder().baseUrl(BuildConfig.SERVER_URL).client(builder.build())
}
```

Fig. 9: Retrofit builder implementing certificate transparency with OkHttp

It is not apparent that certificate transparency is implemented here. However building Retrofit on top of OkHttp activates certificate transparency[4]. Thus some bypassing is necessary. It is unclear how MobSF performed this bypass, but it uses Frida[8], which has several tools for this purpose. We will instead try with a patched apk generated with apk-mitm[9]. Success, it is now possible to intercept traffic in Burp Suite. The generated request from the dynamic analysis is then sent to Burp Suite for extra foundation. Multiple execution flows are performed. The traffic is then investigated. An overview of some requests can be seen on figure 10

Having the setup running it is now possible to perform an attack. Looking at the example of setting an item for sale. We

[8]https://frida.re/docs/android/
[9]https://github.com/shroudedcode/apk-mitm

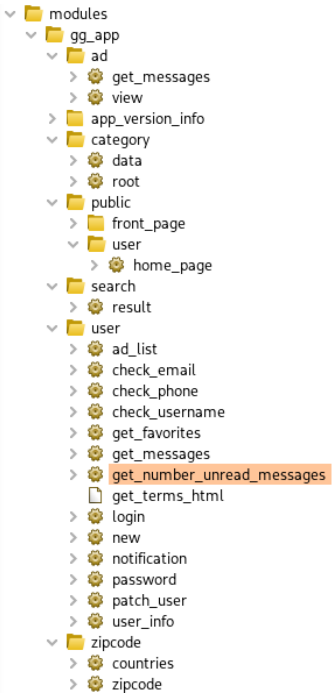


Fig. 10: HTTP Requests for <host name>/modules/

know by analysing the code and requests we want to intercept *modules/gg_app/ad/* requests. The interception can be seen on figure 11. The headline and description is than changed to *mitm*. The resulting item can be seen on figure 12.

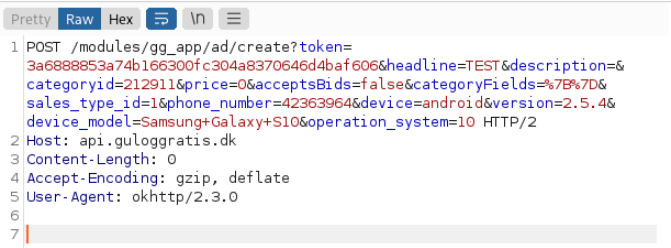


Fig. 11: Create new ad request

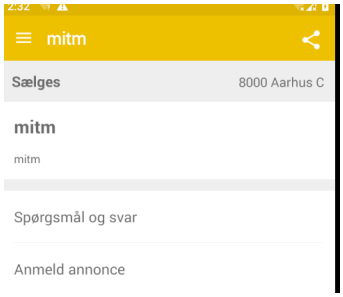


Fig. 12: Manipulated final item

This is an example of manipulating the requests moving from client to server. Burp Suite allow us to also manipulate the response. A result of this can be seen on figure 13. Here the JSON received from server is manipulated.


Fig. 13: Manipulated response from server

In general the networking between application and server is quite easy to understand. It is not a standard REST api which follows all the standards. It is routed by *modules/gg_app/<entity>/<action>*. An example can be seen for the *ad* entity on figure 14.

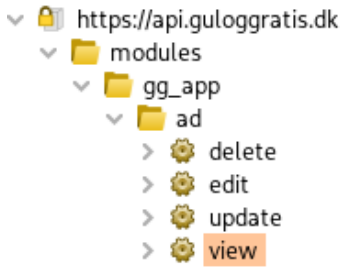


Fig. 14: Example of actions for the ad entity

The HTTP methods are not used as expected. An example is that deleting an ad is done with a GET request. In general the nature of the traffic is the application targeting an action with some query parameters and then the server returning some JSON.

## IV. Authentication

It is essential for the users of the application to have accounts. It enables user interaction and many of the features that are desired in a marketplace application. User's expect their actions and settings persisted across platforms. However, they expect some privacy. Other users should not be able to see their private details. This all leads to the need of having authentication methods in the application. The way it is handled in the application is by allowing three authentication methods: Facebook, Google and email address. Going forward we will only look at email address, as that is GulOgGratis own implementation.

When you create an account the default information is entered and you press create. Looking at the network traffic some validation of various fields are performed. An example is verifying the email is not in use. The application then sends the information to the server, an account is created, the application performs a login request and the server returns tokens.

However the email was never verified. The server returns a JWT (JSON Web Token) and another token consisting of forty characters. A decoded JWT can be seen on figure 15. The header indicates that the token is signed using HMAC-SHA256 and the payload contains some user information. It is unclear looking at the network traffic and the source code what the JWT is used for. Meanwhile the other undefined token is stored locally and used in every HTTP request.

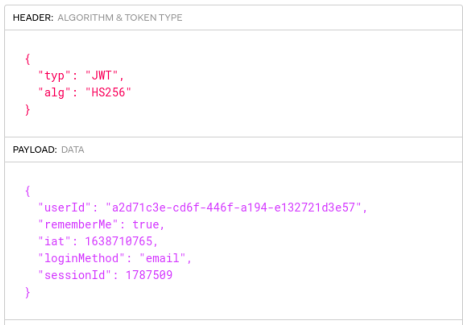


Fig. 15: Decoded JSON Web Token

Having created the account you can sign in using your email and password. The application then calls the api with a login action and the tokens are returned. The user is then authenticated and ready to use them for further actions. The used token is persisted in the application's local directory. When the user then signs out no network traffic is detected. However looking at the code it is seen that the token, user address and user id are removed from local storage.

The password used for authenticating is quite unrestricted. The only requirements are that they need to be at least eight signs and a combination of numbers and letters. Additionally, when changing the password it is allowed to change to one that has already been used, and changing it to the exact same. Resetting the password results in no received email even though a request is detected. Trying to perform the reset on the web site results in an instantaneous received email. Inspecting that traffic it is interesting to see that it utilizes another hostname and the request is completely different. This might indicate that the API the mobile application utilizes is becoming obsolete.

In the system it is possible to validate your account with NemID. This ensures you are the person who you indicate you are. This is a safety for others interacting with you. The validation can only happen through the web site.

## V. Privacy

Mobile applications have the potential of collecting massive amount of data from their users. This can be used to increase user experience but also lead to intrusive behaviour. Permissions is a great place to see if something is out of the ordinary. Some of the most interesting permissions related to privacy of the application:

- INTERNET & ACCESS NETWORK STATE
- MANAGE ACCOUNTS & GET ACCOUNTS & USE CREDENTIALS
- ACCESS FINE LOCATION & ACCESS COARSE LOCATION

The networking permissions are obviously necessary. The account permissions are used to authenticate with either Google or Facebook. The most interesting are the location related. When opening the application the user is prompted whether he will allow location tracking when using the app. This setting is persistent and can not be disabled from the application. The user has to disable it in settings. Additionally there should be an option to allow for one session. The location permission does not seem necessary for the application. The only use is related to a map shown on items for sale. As Google Maps is used for this feature it might be a needed permission.

The data collection of the application revolves around how users use the application and advertisement. Almost whenever button is pressed a chain of requests are performed. A chain can be seen on figure 16.

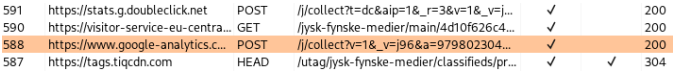


Fig. 16: Analytics chain on user actions

An example of the request can be seen on Figure 17. This request is triggered by the car category when searching for items.


Fig. 17: Analytics request when pressed on the cars(biler) category

These requests send the user's actions to external services. It is seen that the event is sent to Google Analytics and Doubleclick. Doubleclick is an advertisement company owned by Google. This result in targeted advertisements for the user dependent on his behaviour in the application. There is a range of trackers. An overview can be seen on Table I.

The analytics trackers are used to improve user experience. Looking at metrics regarding user sessions and actions. CrashLytics is used to report crashes and tell the developers the state of the application at the incident. The advertisement trackers are retrieving information from the users behaviour and thus targeting advertisements thereof. If the user authenticates with both Facebook and Google, it is expected that the tracking goes beyond the scope of this application. Meaning the behaviour of

| Company | Tracker Name | Category |
|---|---|---|
| Facebook | | |
| | Ads | Advertisement |
| | Analytics | Analytics |
| | Audience | Analytics |
| Google | | |
| | AdMob | Advertisement |
| | Analytics | Analytics |
| | CrashLytics | Crash reporting |
| | DoubleClick | Advertisement |
| Tealium | | Analytics |

TABLE I: Overview of some of the trackers

the user might be visual in ads he receives on other platforms. Other than that there is no integration to Social Networks.

## VI. Discussion

Retrieving a basic understanding the code and flow of the application did not require much time. This is the result of not utilizing obfuscation when compiling the release build. As per the documentation[1] this does not require a lot of effort. It is done by enabling Proguard in the build.gradle. Extensive logging is as well present. This might be of no harm but for a release build these should not be present. Lack of obfuscation and logging only gives an adversary an unnecessary edge in finding potential vulnerabilities. External libraries might be the attack vector for an attacker. He might know a vulnerable library and target a application using it. The Dibs and Zendesk library both utilized a embedded view. One added an interface to call Java code from javascript while the other allowed debugging. The interface has been known to be dangerous. If an adversary where to control the Javascript called from the webview, he would be able to perform severe attacks. However, this was patched in sdk version 17 and the minimum sdk version is 21 for the application.

The TLS configuration of the server resulted in a decent rating. Though, the support for TLS 1.0 and TLS 1.1 should be deactivated. The targeted api versions of the application all support TLS 1.2[2]. The certificate of the server is only domain validated. This is the lowest amount of validation. Users know their data is encrypted however they cant truly trust who is at the receiving end. The certificate should be at least be upgraded to organization validated.

In the attempt of the man-in-the-middle attack there was hurdles. The device has to trust the proxy's certificate and a patched apk is required. This might not be realistic to pull of on an user's device. However, an adversary can mount severe attacks if he is to achieve the setup. The attacks he can perform is editing requests/response, DOS or just eavesdropping. All these are extremely critical and combined violates all policies. In understanding the network traffic an adversary did not need to mount a man-in-the-middle attack, he could just use the application himself.

A token stored in the application local directory are used for each request. Backups of the application data is allowed. Meaning if an adversary are to retrieve a backup of the data, he would be able to perform requests on the user's behalf. This could potentially be avoided by encrypting the token. The token is retrieved by authenticating in the application. First time users creates their account. Leading to immediate authentication with no email confirmation. This is flaw and results in users being able to create accounts with other persons' emails. Additionally, the passwords are allowed to be quite weak and it is possible to change your password to the same or reusing older ones. This encourages brute force attacks. After having authenticated in the application the token is stored and used in further sessions. Therefore, if the phone is unlocked anyone can access the application. This could be resolved by introducing "something you are" authenticity in the form of finger or face scan. Currently the application only uses "something you know" and keeps the user authenticated after first sign in.

The application is not as intrusive as witnessed from other horror stories. The permissions is quite ordinary and expected. However, location permission handling could be improved. There should be transparency in what the permission is used for and it should be possible to limit it for sessions. Advertisements are shown in the application and they are personalized by a range of trackers. These trackers retrieve continuous data from the users' behaviour. This is sort of expected from nowadays applications. But it should be described in a Privacy Policy or Terms and Condition accessible for the user. These are nowhere to be found in the application.

## VII. Conclusion

A preliminary security analysis was performed on GulOg-Gratis' Android application. The analysis was conducted on the basis of a constructed threat model. It was broken down to four different aspects and it was shown that there are flaws more severe then others. Simple mechanisms should be implemented to harden adversaries achieving insight in their application. Improvements can be made in regards of network traffic, authentication methods and privacy transparency. The application is not vulnerable to any apparent and simple to pull of exploits. However, an adversary with a strong motivation might be able to mount an attack. Either by performing a man-in-the-middle attack or stealing the user's token. This would result in a complete violation of almost all security properties.

## References


[1] Android. *Shrink, obfuscate, and optimize your app*. URL: https://developer.android.com/studio/build/shrink-code. (accessed: 19.12.2021).

[2] Android. *SSLSocket*. URL: https://developer.android.com/reference/javax/net/ssl/SSLSocket. (accessed: 19.12.2021).

[3] AVG. *Analyzing an Android WebView Exploit*. URL: https://www.avg.com/en/signal/analyzing-an-android-webview-exploit. (accessed: 19.12.2021).

[4] Matthew Dolan. *Android Security: Certificate Transparency*. URL: https://appmattus.medium.com/android-security-certificate-transparency-601c18157c44. (accessed: 19.12.2021).