Preliminary Security Analysis Of The McDonalds Application

Daniel Christopher Biørrith - 201909298, **Bastian Aron Kramer - 201908709**
Group 3

I. Introduction

In an ever more technological world, everything gets more and more digitalized. This is also the case for major restaurant chains, that either have to follow suit or fall behind their competitors. The most well-known chains today have their own applications, in which you can find special offers, games, and even online ordering. One such application, which is perhaps the largest and most well-known of them all, is the McDonald's application.
The application makes it easier than ever to buy food, collect offers and even collect points-per-payment one may use to buy even more food. What's not to like? However, such digitalization does not come without risk. An application that handles the private data of the users, and also directly handles online payments, must have a high level of security. This includes using cryptographic methods when handling sensitive data, as well as securing the different layers of communication between the application and the internet from malicious attacks.

The purpose of this report is to perform a preliminary security analysis of the McDonald's application. First, we will present the threat model of the application. Secondly, we will be decompiling the application to try and determine the software security of the application and find possible vulnerabilities using static analysis. This will be followed by an analysis of the network security, by trying to do a man-in-the-middle attack. Finally, the authentication and privacy of the application will be examined, followed by a conclusion on the security of the application as a whole.

A. Introduction to the application

The application is, as mentioned, an application developed by McDonald's. It is a worldwide application that has specific use cases and features unlocked based on the country it's used in. As we're using it in Denmark, we will focus on the features found in the Danish version. The main features are:

- Two-factor authentication.
- Location tracking to see the nearest McDonald's.
- Online ordering and Real-time Transactions for payment via either card or Google Pay. MobilePay is not available. The application provides the user the possibility of storing their card information for future use.
- The application provides special offers in the form of coupons. When these are redeemed, a QR code appears that exists and may be scanned for 2 minutes.
- If a user's private QR code has been scanned prior to paying, they will earn a certain amount of points that may be used to pay for food.

With all these features, the application seemed to be a perfect fit for the purpose of this project. It seems interesting to investigate how the app handles user privacy with user data required to create a profile, such as name, phone number, email address, and payment details. Furthermore, it may be interesting to see if there are any possible exploits during the use of the Real-time Transactions.

From searching online, successful attempts at reverse engineering the application has already been done[1][2]. To our knowledge, McDonald's has updated their application so that these attacks are no longer possible. However, we're not sure if the application has increased security or simply patches the holes.

II. Threat model

The following section will present the results of the threat analysis of the McDonald's application, by doing threat modeling, finding attack surfaces, and finding possible adversaries. This is an important step in order to identify potential threats and vulnerabilities in a system, so as to understand and mitigate risks.

A. Adversaries

The following have been identified as possible adversaries of the application:

- **Users/Customers:** They may try to find vulnerabilities that can give them more coupons

and points, which may provide better/more offers and/or free food. They may also try to find a way to exploit the purchasing feature of the app and try to order food without payment.
- **Hackers/Cyber Criminals:** This group of adversaries may be interested in the sensitive information the application has stored. This may be general sensitive information such as age, sex, or contact information. However, it can also be the payment information of the users if that's not properly encrypted.
- **Inside man:** This would be an adversary working for the application development team. One such adversary has inside knowledge of how the application works, as well as important encryption knowledge. This may be exploited for their own benefit, and can also be used to create security holes to exploit.
- **Competitors:** A competitor might want to worsen the quality and availability of the application through denial-of-service attacks.

B. Attack Surface

Next, we identify the possible attack surfaces of the application. These are:

- **The application:** This includes testing the actual application for bugs, as well as decompiling the application and looking at the code base to look for exploits and shortcomings.
- **Network communication and APIs:** Here many possible attacks exist, such as trying to intercept communication between the application and the endpoint. The application must encrypt transmitted data to avoid others sniffing sensitive information.
- **Payment processing:** If the application handles payment information, it may be targeted for attacks like credit card fraud or payment gateway compromise.
- **User authentication:** The login mechanism could be vulnerable to attacks like credential stuffing or password guessing.

Not all of these will be tested in this report. The ones we will focus on are the testing of the application and the network communication, though an analysis of the authentication will be performed.

C. Properties

Finally, we will define properties we believe are of key importance for the application. As the application handles user-sensitive data and handles transactions for purchases, it should/must uphold the following properties:

- **Privacy:** This is very crucial for the application as it stores user-sensitive data. To uphold privacy it is important to ensure the data will be handled with much care. Additionally, it must do this in order to comply with EU's GDPR laws.
- **Authenticity:** In the same note as above, the application handles user-sensitive information. This must be kept private to the user by the use of authentication, meaning only the user can access their own data.
- **Accountability:** Following the previous two properties, it is important to be able to deem who is accountable for specific actions. This is especially true when the application handles real-time purchases, for the user to be able to confirm a transaction, and for the developer to show the transaction.
- **Availability:** As a company you're always interested in having the application available for the users at all times. If the application is unavailable it will lead to a loss of business. This is especially important when considering the application has over 100 million downloads, meaning little downtime may have very large consequences.

III. Software Security

This section examines the software security of the Mcdonald's application through static analysis.

A. App decompilation

In order to analyze the code, we first had to decompile the application APK. The APK was obtained using APK Pure[3]. Afterward, it was decompiled to the Java source code using dex2jar[4], as well as with JADX to see if gave a different result (it did not). Finally, the code was inspected using JADX-gui [5]. The JADX decompilation produced 93 different errors, meaning the decompiled code would have a certain level of obfuscation and missing files.

From inspecting the code, we found that much of it had been successfully decompiled into code (as in, it is not only comments) with a few exceptions, as well as to XML files. However, many of the names and types appear very random, making it seem as if the developers put work into purposely obfuscating the code. This makes the code very hard to decipher and understand what is used in which context. For an example, see figure 1.

Because of this obfuscation, it is very hard for an adversary to reverse-engineer the application and understand the underlying mechanisms. One important

```
public final class AppApplication extends j67 implements AppBui

    @Metadata(m317d1 = {"\u0000\f\n\u0000\n\u0002\u0010\u0002\n
    /* renamed from: mcdonalds.app.AppApplication$a */
    /* loaded from: classes2.dex */
    public static final class C10311a extends li5 implements kh
        /* JADX WARN: 'super' call moved to the top of the meth
        public C10311a() {
            super(1);
            AppApplication.this = r1;
        }

        @Override // com.kh5
        public de5 invoke(uz8 uz8Var) {
            Context context;
            MarketConfiguration loadMarketConfiguration;
            uz8 uz8Var2 = uz8Var;
            ji5.m10338f(uz8Var2, "$this$startKoin");
            h09 h09Var = h09.ERROR;
            ji5.m10338f(uz8Var2, "<this>");
            ji5.m10338f(h09Var, "level");
            tz8 tz8Var = uz8Var2.f50721a;
            pz8 pz8Var = new pz8(h09Var);
            Objects.requireNonNull(tz8Var);
            ji5.m10338f(pz8Var, "logger");
            tz8Var.f49375b = pz8Var;
            MarketConfigurationManager marketConfigurationManag
            ji5.m10338f(marketConfigurationManager, "marketConf
            Context context2 = AppApplication.this;
            ji5.m10338f(context2, "context");
            if (marketConfigurationManager.getMarketConfigurati
                MarketConfiguration marketConfigurationPreferen
                if (marketConfigurationPreference == null || (1
                    context = null;
                } else {
                    Configuration configuration = context2.getR
                    ji5.m10339e(configuration, "context.resourc
                    Locale locale = loadMarketConfiguration.get
                    Locale.setDefault(locale);
```

Fig. 1: Example of function with very generic input names.

thing to note is that strings and log statements have not been obfuscated. Because of this, if one had a lot of patience and time, the code could possibly be deciphered. The code was attempted de-obfuscated using Jadx, though with no luck.

B. Manual inspection

Despite the obfuscation, we still examined the code manually. The application appears to be written in either Java or Kotlin (there is a folder called Kotlin, yet the code looks like Java code. We're not experts in either, but we believe it might be Kotlin transpiled to Java) and uses a Firebase[6] database for storing data. The application is very large. Using LOC to quantify the size of the code[7], we found it to be 1734579 lines of code, 1279779 of which is pure Java.

It goes without saying that the application is too large to go through line by line. Instead, we searched for specific, important keywords and inspected the code in areas where these were present. Additionally, we manually searched folders with names that could possibly have vulnerabilities, such as `Login`, `GooglePay`, and `Payment`. The structure makes it hard to figure out what happens where especially

```
public q61(String str) throws EncryptionException {
    if (Pattern.compile("([A-F][0-9]){5}\\|([A-F][0-9]){512}").matcher(str).find() && str.length() == 518)
        this.f43804c = new SecureRandom();
        String[] split = str.split("\\|");
        try {
            try {
                PublicKey generatePublic = KeyFactory.getInstance("RSA").generatePublic(new RSAPublicKeySpec
                try {
                    this.f43802a = Cipher.getInstance("AES/CCM/NoPadding");
                    try {
                        Cipher cipher = Cipher.getInstance("RSA/None/PKCS1Padding");
                        this.f43803b = cipher;
                        cipher.init(1, generatePublic);
                        return;
                    } catch (InvalidKeyException e) {
                        throw new EncryptionException(ud1.m3855s0("Invalid public key: ", str), e);
                    } catch (NoSuchAlgorithmException e2) {
                        throw new EncryptionException("Problem instantiation RSA Cipher Algorithm", e2);
                    } catch (NoSuchPaddingException e3) {
                        throw new EncryptionException("Problem instantiation RSA Cipher Padding", e3);
                    }
                } catch (NoSuchAlgorithmException e4) {
                    throw new EncryptionException("Problem instantiation AES Cipher Algorithm", e4);
                } catch (NoSuchPaddingException e5) {
                    throw new EncryptionException("Problem instantiation AES Cipher Padding", e5);
                }
            } catch (InvalidKeySpecException e6) {
                throw new EncryptionException(ud1.m3855s0("Problem reading public key: ", str), e6);
            }
        } catch (NoSuchAlgorithmException e7) {
            throw new EncryptionException("RSA KeyFactory not found.", e7);
        }
    }
    throw new EncryptionException(ud1.m3855s0("Invalid public key: ", str), null);
}
```

Fig. 2: The encryption types of credit cards in the application.

with the obfuscated names. By finding a class called `EncryptedCard`, we managed to track back through classes (thanks to exception throws) and find that the card is encrypted with AES CCM and RSA with a PKCS1 padding, as shown in figure 2, which both provide a high level of security.

Generally, the application uses Adyen, a Dutch firm that specializes in secure payments, which bodes well for the security of the application regarding in-app payments. We have not been able to find anything about Adyen having any vulnerabilities online.

We also discovered that the application had an implementation of an AES obfuscation. This means that the developers actively tried to obfuscate the application, which was done based on a GitHub implementation called PiracyChecker[8]. The obfuscation is done with AES CBC encryption with PKCS5 padding. Though one may tamper with the CBC ciphertext, it does not help deobfuscate the application in this case. We tried to download the very first version of the application. This was because we hoped the name scheming from the old application, released in 2018, would help us understand the newest version. This was without luck, as it also had the obfuscator implemented.

After the initial manual inspection, we performed a keyword search. Sadly, due to the obfuscation, it was very hard to find the context around the places where the keywords appeared, and it led to very few noteworthy findings (that haven't already been mentioned), even if the keywords were present. A list of the searched keywords is:

- HMac
- SHA1,2,3, 256, 512
- Card, Pay(ment)
- Encrypt

- IV, Initialize, vector
- Coupon
- Password
- AES, ECB, CBC, CTR, GCM, CCM
- Nonce
- TLS

Many of these we found in a bunch of instances (both hundreds and thousands of appearances), but it is too hard and time-costly to decipher given the timeframe of the project.

By searching for the keyword Crypto, we found that the application uses the library `javax.crypto`, which throughout the whole application has been included in 30 different files, many of which have names such as ak8 and ll8, leaving little room for understanding without going through the whole application. The library is considered secure when used correctly, though it had a security blunder patched in 2022 [9]. Finally, a search for `random` shows the application makes sure to use `SecureRandom` throughout the application, as opposed to normal random which is unsafe in Java[10]. Generally, the application appears to be well secured and very hard for adversaries to find any vulnerabilities in.

C. Static analysis using MobSF and ImmuniWeb

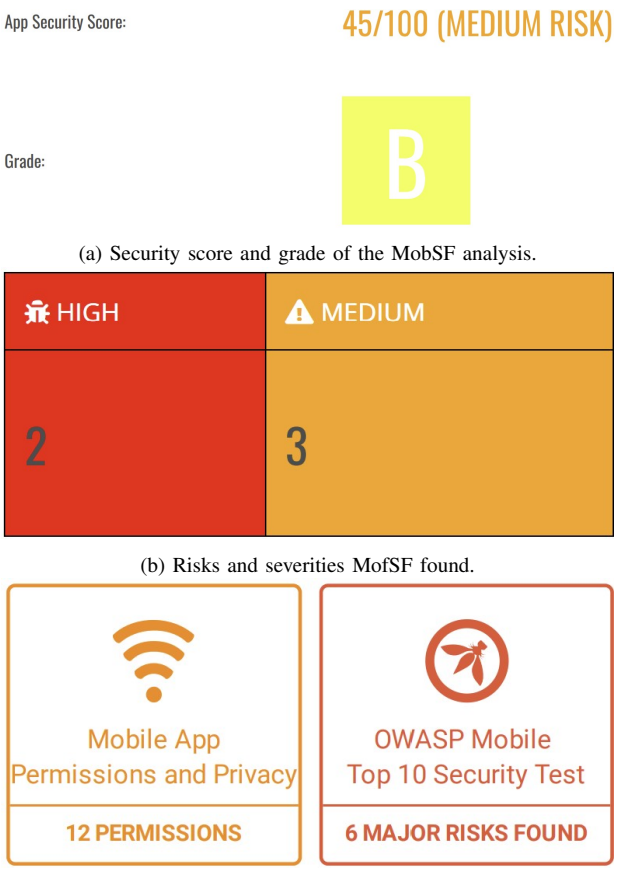
Though the manual inspection showed good results for safety, the application is still too large to go through manually. As such, we put the app through the static analyzers MobSF and ImmuniWeb. A recap of the results is shown in figure 3.

As the results of both of these give a combined 150+ pages, we will not be going into them in detail. From the results, we see that the MobSF score is only 45/100, meaning the application is of medium risk, which goes a bit against the findings of the manual inspections. We will now go through the most important noteworthy findings.

1. Application permissions: It is stated that 6 permissions are of possible danger. These are based on location, camera access, external storage read/write, and audio recording. These will be discussed in a later section.

2. Manifest analysis: Here, it states that clear text traffic is enabled for the application. This makes eavesdropping possible, which makes possible attacks easier. We will test this when performing the man-in-the-middle attack later.

3. Hardcoded secrets: Here the secrets worth mentioning are the Firebase Database URL and their Google API keys, as these are hard-coded into their application. It's hard to say anything regarding the Firebase security, as that would depend on the settings/rules set inside the database. The Google API keys, though not usually available publicly, can be found quite easily, from what we've found[11], as the Google Maps SDK sends the API key in such a way that it can easily be found. The many other 'secrets' MobSF found appear as they contain keywords such as 'password', 'logout', and 'auth', and from our examination we deemed the found ones to be of no concern. Additionally, some URLs were found, which will be tested in the Network Security section.



(a) Security score and grade of the MobSF analysis.

(b) Risks and severities MofSF found.

(c) Result of the ImmuniWeb static analysis.

Fig. 3: The overall static analysis results from Mobsf and ImmuniWeb.

Next, we will go through the findings of the Immuniweb analysis. Here, some of the same risks and permission notices are present, which we will not repeat.

OWASP Mobile Top 10 Security Test: OWASP top 10 is a standard for ranking the top 10 biggest security risks[12], where Immuniweb tests for all these. A potential security risk it found is external data in SQL queries, which in turn might potentially lead to local SQL injection vulnerability[13]. From looking at the stated possible vulnerability, it's very hard to say in which context it appears and how big of an effect

it might have, due to the name obfuscation. Another important notice is that the application in some places uses the insecure hash MD5. Once again, it is hard to know if the context MD5 is used in is for sensitive data. The analysis also brings up the lack of anti-emulation techniques. These may be implemented and used to make reverse engineering harder. Other notices include warnings of serializing objects, hardcoded URLs, and the use of intents. The URLs will be examined later. We will not be going into further detail regarding the rest.

After going through the reports of MobSF and ImmuniWeb, as well as performing a manual inspection, the application appears to have put a great deal of work into the security application, with only minor warnings being present. We do not believe the scores of the report reflect the actual security, as many of their warnings are based on false positives.

IV. Network Security

This section will examine the network security properties of the application. First, APIs found in the static analysis will be analyzed. Afterward, an attempt to perform a man-in-the-middle attack will be presented,

A. Testing hostnames

As mentioned in the previous section, the static analysis tools found a list of API URLs hardcoded into the application. In table I a collection of the URLs we deemed as interesting to test is presented.

The APIs have all been tested with the Qualys SSL Labs analysis tool[14]. The tool provides each API with a grade score, ranging from A+ as the best score to T as the worst possible score[15].

As evident from the table, `prd-euw-gmal-mcdonalds.firebaseio.com` only has a grade score of B. The reason for this is that it supports the TLS protocols 1.1 and 1.0 to which the grade cap is set to B. The protocols are considered rather weak for many reasons, such as the use of weak cipher suites and SHA-1, and better options exist in the form of TLS 1.2 and TLS 1.3. The reason that most of the rest are graded A is that they use TLS 1.2, which has weak encryption methods. Though it is not as effective and secure as TLS 1.3, TLS 1.2 is still considered safe if implemented correctly.

URL	GRADE
gmaliteapp.mcd.ddbcph.dk	A
prd-euw-gmal-mcdonalds.firebaseio.com	B
mcdonaldsapps.com	A
mcdonalds.com	A
symbolize.corp.google.com	A+
mcdonaldsdigital.com	A
nullpointer.wtf	A+
mikepenz.com	A

TABLE I: A table of the URLs tested with Qualys SSL Labs and their respective grade

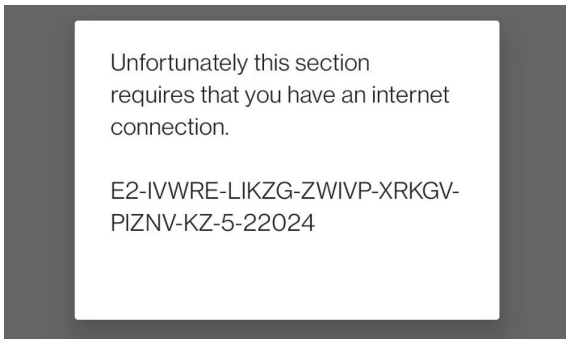


Fig. 4: Applicatin response to the MitM proxy server attack.

B. Performing man-in-the-middle-attacks

Man-in-the-Middle (MitM) is a cyberattack where a malicious actor intercepts communication between two systems. This is done with the hopes of capturing web traffic, from which sensitive information may be obtained, or learning the package structure so as to send destructive packages. We tried to carry out such a MitM attack on the McDonald's application. The attack was conducted in a controlled network environment in order to guarantee safety and reliability. The setup consisted of an Android phone with the application installed, from which network traffic was redirected to a proxy server hosted by a laptop. The server was set up using the mitmproxy tool [16]. At first, we attempted to intercept the traffic produced by the application without any additional certificates, though without success. It resulted in the error message shown in figure 4. This implies that the app has some sort of security implemented, and the proxy server was detected as a distrusted device.

As the next step, we tried installing a trusted CA X.509 certificate onto the device, which was generated using mitmproxy. This did not change the outcome, meaning we once again got the result shown in figure 4. This would imply the application uses certificate pinning; a standard used in applications since Android 7.0 [17]. The conclusion to this would be that the application is not applicable to a MitM attack under normal circumstances.

Finally, it was attempted to modify the APK file, inserting a malicious root certificate into it, and afterward installing it onto the Android device. The certificate was generated using mitmproxy, which was added to the pinned list of certificates in the APK file using a

tool called apk-mitm [18]. Sadly, this attempt was also without success, as once again we were met with the error seen in figure 4. It seems as if the application has an implementation of pinning that apk-mitm was not able to locate and disable. This could very well be because of the obfuscation implemented into the application.

A possible last attempt would be to install the certificate at the system level of an Android emulator, though this was not carried out due to time constraints and the scope of the project.

The conclude, these tests show that the safety of the application is of high quality, and demonstrates that the developers have put significant thought into the security of the application.

V. Authentication

Next, we analyzed the authentication method in the application. Authentication is a critical aspect of any application's security, as it verifies the identity of users before they can access personal data within the application, or access certain features. In the context of the McDonald's application it does the following:

- Protect User Information: As users are likely to have personal and potentially sensitive data stored within the app, such as their names, contact details, and payment information, it's crucial to prevent unauthorized access to this data.
- Ensure Transactional Security: The app allows for transactions for ordering food, making payments, or saving favorite orders. Authentication may help ensure that these actions are only carried out by authorized users. This can prevent fraudulent transactions and misuse of the app.
- Enable User Account Management: Authentication enables users to manage their accounts, such as updating their personal details, changing passwords, reviewing order history, and managing payment methods. Without proper authentication, these functions would not be possible.

These, among others, are important reasons for having a safe authentication system.

The McDonald's application uses a user-Oriented authentication, with a standard username/password login. Here, the username is a provided email of the user, and the password must consist of at least 8 characters and include at least one large letter, one small letter, and a number. In addition to this, two-factor authentication is implemented, where the user receives an email containing a six-digit number which must be put into the application in order to complete the login. Said code remains valid for four hours. Upon logging in, a user remains logged in even after closing the application, implying an authentication token must be given to the user. If a user logs out and tries to log back in, a six-digit number is once again sent to their private email, which must be put into the application.

A test of the password reset function was also made. Here, it starts with activating the process in the login screen by pressing "Har du glemt din kode?" (have you forgotten your password?). After, the email of the account must be provided. If an account with the email exists (which is not shown to the user), an email will be sent with a link to a website where the password can be reset. Here a new password had to be entered two times to set it as the new password.

The combination of the password and the six-digit number essentially renders out a brute-force approach unfeasible. Many other methods for gaining access exist, such as intercepting a token from a package or getting hold of a leaked token. However, as we were unable to analyze sent packages, we do not know how doable it is, nor how secure the application is in this regard.

A. Suggest improvements to the authentication

After testing the authentication and holding it against the threat model analysis, as well as considering the general use-cases of the application, we consider the two-factor authentication to be sufficient for the application. However, an application can always become more secure. Here, we suggest a few authentication mechanisms to even further improve the security of the application.

- Biometric Authentication: This involves the use of unique biological characteristics of the user such as fingerprints or facial recognition. The reason that biometric data can improve security is that it's unique to each individual, making it a robust form of authentication. However, there can be some privacy concerns regarding the storage and use of biometric data.
- Geo-location Verification: This is a mechanism that can flag or block access if a login attempt is made from a new or unrecognized location by tracking the geographic location of the device. This method is usually used as an additional layer of security. It is not completely safe since tools like VPNs may be used to change your perceived location. The method could help if a login is made in an unusual location, especially if the username and password have been compromised.

VI. Privacy

This section will present the potential findings of analyzing the privacy of the applications.

A. Android Permissions

Through static analysis, we found 12 different Android permissions in the application. We only discuss the ones marked as dangerous by MofSF, Immuniweb, or by both. This narrowed it down to 6 permission. A list of the permissions follows:

- CAMERA: This permission is required when the app needs to access the device's camera, which it does when scanning QR codes at the restaurant. It allows the application to collect images at any time.
- RECORD AUDIO: We have no clue as to when the application needs audio. Perhaps this is for a feature in one of the other countries.
- ACCESS FINE LOCATION: This permission allows the app to access precise location information from the location-based services of the device. This is often in the form of GPS. The application uses the locations to recommend restaurants that are close.
- ACCESS COARSE LOCATION: This is similar to ACCESS FINE LOCATION, but it only provides approximate location information. The location is derived from network location sources such as cell towers and Wi-Fi. We assume the use of it is the same as the above.
- WRITE EXTERNAL STORAGE: This permission allows an app to write (save) files to the external storage of the device. This could include things like saving photos, videos, or other files that you create or modify within the app. It might also be used for downloading content for offline use. We're not sure which feature this is needed for. Perhaps this is for a feature in one of the other countries than Denmark.
- READ EXTERNAL STORAGE: This permission allows the app to read files from the external storage of the device, like photos and videos. This probably works in close correlation with WRITE EXTERNAL STORAGE, as the files written must be read again.

Most of the above is mainly guesswork as to where and why the permissions are required. As we cannot properly examine the code due to obfuscation, none of this can be confirmed nor denied. A problem with these Android permissions is that if a hacker gets control of the application, the Privacy of the users may be seriously compromised.

B. Privacy data collection

By looking at the privacy policy of the application, we found that McDonald's collects the following information (translated using DeepL translator [19]):

- The operating system and browser type of your computer or mobile device;
- The type of your mobile device and its settings;
- UDID (unique device identifier) or MEID (mobile equipment identifier) of your mobile device;
- Serial numbers of devices and components;
- Advertising identifiers (IDFA, IFA, or similar) or similar identifiers;
- Referring website (the page from which you entered our site) or application;
- Online activity on other websites, applications, or social media;
- Other internet traffic information, including information about which websites you access and how often you log in to social media, streaming services, etc;
- Your communications to or about us on social media;
- Order number and receipt generated by mobile ordering using MOP;
- Activity related to the way you use the Online Services, such as which pages you visit on our sites or in our mobile apps.

Additionally, it states that it will collect information regarding the user's exact location using geolocation and technologies such as GPS and Wi-Fi. Furthermore, they may collect information from other companies and organizations, as well as public information. An example is when interacting with them on social media.

Though it might be normal in today's world, the sheer amount of data the application may (and probably does) collect is astounding. The application has the potential to track a user's every location as well as store it all. Though much is probably used for marketing purposes, it is worrying how much different data is collected, generally without the knowledge of the user, as close to no one reads the actual privacy policies. If an adversary manages to get hold of the data it will have very large consequences.

C. External services

The application also integrates with some external services/trackers. A list of the ones found is presented in table II.

Tracker	**Categories**
Facebook Login	Identification
Facebook Share	Possible sharing information
Google Analytics	Analytics
Google CrashLytics	Crash reporting
Google Firebase Analytics	Analytics
Google Tag Manager	Analytics
HockeyApp	Crash reporting

TABLE II: A table of the different Trackers found in the application

It shows the application integrates with Facebook, Google, and Firebase. The use of Facebook in the application is not apparent from using the app, though it makes sense when considering their privacy policy (advertisement and interactions with their social media). Perhaps, the application has a Facebook feature in other countries than Denmark. The Google integration is apparent, as the application has its own modified Google Maps integrated into the location (on Android, and on iOS it uses Apple Maps). Though these are all standard and well-known services, they must be integrated properly to ensure complete safety, and they may be a possible point of weakness in the application. Once again we cannot say if this is the case due to the obfuscation of the code.

VII. Conclusion

In this project, a preliminary security analysis of the McDonald's application has been performed. From initial research, it was found that we were not the first to do this, and two severe examples of insecurities in the applications were discovered. These were however patched up before our analysis. Now, it seems the application has stepped up the security a lot. After making an initial threat model, we found it very challenging to perform a software analysis, due to intentional obfuscation of the code. Furthermore, the application proved very resilient to man-in-the-middle attacks, probably because of an implementation of certificate pinning that proved secure when tampering with the APK to include certificates. An investigation into the authentication was done, finding the existing one fulfilling for the use-case of the application, though with some possible improvements provided. Finally, the privacy of the application was analyzed, finding that the application had the permission to store a lot of sensitive user data, possibly more than one would imagine necessary for an application developed for a fast food chain.

In conclusion, the application seems very secure, and we did not find many glaring weaknesses. Though, with the amount of sensitive data stored, the developers will have to keep improving upon the security of the application as the level of attackers increases in the coming years.

References

[1] scrxtchy. (2019) Re: Reverse engineering mcdonalds app. [Online]. Available: https://scrxtchy.github.io/posts/re_maccas/

[2] Ferib. (2020) Reversing the mcdonalds mobile application to get free food. [Online]. Available: https://ferib.dev/blog.php?l=post/Reversing_the_McDonalds_Mobile_Application_to_get_free_food

[3] APKPure. (2023) Apk pure. [Online]. Available: https://apkpure.com/

[4] Pxb1988. (2023) dex2jar. [Online]. Available: https://github.com/pxb1988/dex2jar

[5] Skylot. (2023) Jadx. [Online]. Available: https://github.com/skylot/jadx

[6] Google. (2023) Firebase. [Online]. Available: https://firebase.google.com/

[7] Cgag. (2023) Loc. [Online]. Available: https://github.com/cgag/loc

[8] J. Santos and J. Fiquitiva. (2018) Piracychecker. [Online]. Available: https://github.com/javiersantos/PiracyChecker/tree/master

[9] P. Ducklin. (2022-04-20) Critical cryptographic java security blunder patched – update now! [Online]. Available: https://nakedsecurity.sophos.com/2022/04/20/critical-cryptographic-java-security-blunder-patched-update-now/

[10] P. Garg. (2011-12-14) Secure random number generation in java. [Online]. Available: https://resources.infosecinstitute.com/topic/random-number-generation-java/

[11] (2014) Is google api key sensitive information? [Online]. Available: https://security.stackexchange.com/questions/52863/is-google-api-key-a-sensitive-information-that-needs-to-be-protected

[12] OWASP and kingthorin. (2023) Owasp top ten. [Online]. Available: https://owasp.org/www-project-top-ten/

[13] OWASP. (2023) Sql injection. [Online]. Available: https://owasp.org/www-community/attacks/SQL_Injection

[14] Qualys. (2023) Qualys ssl labs. [Online]. Available: https://www.ssllabs.com/

[15] ——. (2020) Ssl server rating guide. [Online]. Available: https://github.com/ssllabs/research/wiki/SSL-Server-Rating-Guide?fbclid=IwAR03BUNlQUhF0XTxtmJi3S3phHCzfJdEgoNTFNYn7BYGFp06

[16] A. Cortesi, F. Valentini, and D. Weinstein. (2023) Mitm proxy - an interactive https proxy. [Online]. Available: https://mitmproxy.org/

[17] A. Pandey. (2020-06-14) Ssl pinning in android. [Online]. Available: https://mailapurvpandey.medium.com/ssl-pinning-in-android-90dddfa3e051

[18] shroudedcode. (2022-04-17) apk-mitm: A cli application that automatically prepares android apk files for https inspection. [Online]. Available: https://github.com/shroudedcode/apk-mitm

[19] D. SE. (2023) Deepl translator. [Online]. Available: https://www.deepl.com/translator