# On the Struggle Bus: A Detailed Security Analysis of the m-tickets App

# On the Struggle Bus: A Detailed Security Analysis of the m-tickets App

Jorge Sanz Maroto, Haoyu Liu, and Paul Patras

School of Informatics, The University of Edinburgh, UK

**Abstract.** The growing shift from private to public transportation and the increasing use of smartphones have lead to the development of digital transport ticketing systems. Such systems allow transport operators to enhance their services and income, therefore are important assets that require secure implementation and protocols. This paper uncovers a range of vulnerabilities in the m-tickets app used by Lothian Buses, one of the leading transport operators in the United Kingdom (UK). The vulnerabilities identified enable attackers to predict, reactivate and modify tickets, all of which can have damaging consequences to the operator's business. We further reveal poor implementation of encryption mechanisms, which can lead to information leakage, as well as how adversaries could harness the operator's infrastructure to launch Denial of Service attacks. We propose several improvements to mitigate the weaknesses identified, in particular an alternative digital ticketing system, which can serve as a blueprint for increasing the robustness of similar apps.

## 1 Introduction

As of 2020, 3.5 billion smartphones have been produced [12], equivalent to 45.1% of the world population. The transportation industry is catching up with this trend and transitioning from cash-based ticketing systems to digital tickets. In a market that was estimated to be worth $500 billion in 2017 [13], the economic impact of public transport ticketing apps is ever-growing. As these systems become more widespread, it is vital that their operation cannot be tampered with for illicit purposes and user data remain protected.

This paper investigates the security and robustness of **m-tickets**, a popular local transport ticketing app deployed among others by Lothian Buses. Lothian Buses manages the majority of public transport operations in Edinburgh, UK, and the Lothian region; it is also the biggest public municipal bus company in the UK, serving approximately 2.3 million passengers per week with a fleet of over 840 buses, and has a daily revenue of approximately £440,000 [9]. We use this as a case study to reveal multiple weaknesses public transport ticketing apps suffer from, including the prediction of tickets and availability issues. Additionally, we propose solutions to the problems identified, in order to improve the security of such systems, whilst maintaining the intended functionality of the official apps.

**Prior Work.** One of the most notable vulnerabilities in the UK public transport ticketing system was discovered by two Dutch security researchers in 2008 [15].

By exploiting the fact that the older version of the London transport system's Oyster card used Mifare 1k chips, the researchers were able to extract an Oyster card's encryption key and use this to clone and modify other cards as desired. This triggered a rapid response by the UK government, which led to a swap of all Oyster cards in circulation with newly developed, encrypted cards, despite the massive cost incurred.

In terms of transport apps, *get me there*, which can be used for purchasing tickets valid in the Greater Manchester Metrolink tram system, was recently compromised, allowing hackers to create free tickets and defraud operators [14], while posting the methodology used on Reddit [11]. The group explained how they were able to extract the private keys used to build the ticket QR codes directly from the source code, making the findings public without responsible disclosure. The app was developed by *Corethree*, the same company that developed the app in used by Lothian Buses, which we scrutinise in this study.

**Contributions.** To the best of our knowledge, there are no scientific papers undertaking a formal security analysis on public transportation apps. This paper aims to fill this important gap and stimulate further research on this topic. As such, we make the following key contributions:

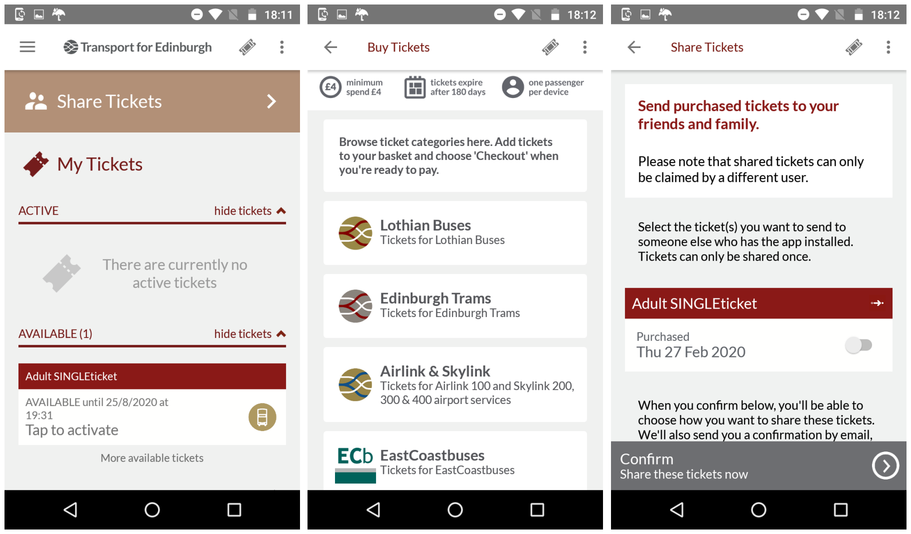
1. We reverse-engineer the m-tickets app, revealing an exploit that enables to predict valid tickets for any future date; additionally, we devise a method to modify the characteristics of any given ticket.
2. We design a simple app that works side by side with the official one, to reactivate old, expired tickets, thereby converting a single ticket purchase into an unlimited source of tickets.
3. We propose an alternative system to fix all the vulnerabilities identified and preserve the intended app functionality.

**Responsible Disclosure.** Prior to the submission of this manuscript, we contacted both the transport operator using this app and the company developing the app, to disclose the vulnerabilities found. The developers are now aware of the problems we discovered and are working towards fixing these vulnerabilities.

## 2 The m-tickets App

Lothian Buses is a company primarily owned by The City of Edinburgh Council (91% ownership), which operates the majority of bus services that run in Edinburgh and some throughout the surrounding Midlothian, East Lothian and West Lothian counties. The so called *Lothian City* division provides the local bus operations with an extensive network of routes that are active 24 hours/day, 365 days/year. In addition, the company owns four other divisions with a focus on sightseeing, private services, and commuter routes.

Given the size of its customer base and the rapid uptake of mobile technology, the company has adopted a mobile app to offer a ticket purchasing and storage service to users. The m-tickets app is developed by *Corethree*, an award winning company [1] specialised in solving the ticketing challenges faced by public and private transport companies. Some of the apps in the company's portfolio serve



There are currently no active tickets

AVAILABLE until 25/8/2020 at 19:31

Browse ticket categories here. Add tickets to your basket and choose 'Checkout' when you're ready to pay.

Send purchased tickets to your friends and family.

Please note that shared tickets can only be claimed by a different user.

Select the ticket(s) you want to send to someone else who has the app installed. Tickets can only be shared once.

When you confirm below, you'll be able to choose how you want to share these tickets. We'll also send you a confirmation by email,

(a) Main screen (b) Buying different tickets (c) Sharing a ticket

**Fig. 1.** Screenshots of the functionality of the m-tickets app.

Transport for Greater Manchester, Transport of London, Northern Link Ferries, Translink, and many more. The m-tickets app is compiled from the same source code for both Android and iOS platforms, has over 200,000 downloads, and we estimate 20%-35% of these correspond to active users [16]. Using the number of weekly customers, we expect the Lothian Buses app accounts for 12.2%-21.3% of the tickets purchased on a daily basis, generating between £19.5 and £34.2 million in revenue per year. Even though this is clearly an important asset for the company, the app was known to have several connectivity and availability problems [6], which we investigate in depth in this paper.

In this work, we focus on the Android version of the app, specifically version 9.7 released on the 17$^{th}$ of July 2019, which at the time of writing is the latest version. Once the app is opened, the user is greeted with a screen displaying the number of tickets available or active (see Fig. 1a). The user has the option to buy (Fig. 1b) or share (Fig. 1c) tickets displayed next to the available ones. Ticket sharing is performed by asking the user for the recipient's email address. The recipient will receive an email containing a hyperlink that, once clicked, adds the sent tickets to their respective list of available tickets. Both sharing and buying of tickets are not available without Internet access.

## 3 Adversary Model

We expect an attacker to already have a copy of the Android app and have basic understanding of the Android app ecosystem. Additionally, we expect the adversary to have some reverse-engineering knowledge and the appropriate tools to intercept traffic from and to the app. Lastly, the attacker would have basic networking and programming knowledge, enough to identify vulnerable code.

Parts of the app that may be prone to attacks and possible scenarios include:

1. **Financial Interest.** The app's main purpose is to provide tickets to users; however, this has an implied given cost. An attacker may attempt to exploit this application to overcome the financial burden, by figuring out a way of obtaining valid tickets without paying.
2. **Denial of Service.** Attackers may attempt to take control of the resources used by the transport operator and seek to disrupt the standard behavior of the app or servers. This may involve flooding target victims with unsolicited messages, which in the process can also harm the reputation of the operator, as the source of the hijacked resources would be attributed to them.
3. **Reputation Damage.** In addition, hacktivists may seek to publish on dedicated platformed (e.g. Pastebin) information about how to obtain free tickets, simply due to a certain ideology.
4. **Privacy Breach.** Attackers may also seek to leak databases or files containing information about the users of the m-tickets app. This would be done for financial gains or, again, to harm the company's reputation.

## 4 Methodology

Next we describe the methodology used to analyse the m-tickets app.

### 4.1 Vulnerability Analysis

To study the app, we employ both **Static Analysis**, reverse-engineering and code auditing whilst the app is *not* running, and **Dynamic Analysis**, which covers any activity and tests done whilst the app *is* running.

**Static Analysis:** We first reverse-engineer the Android application package (APK) of m-ticket by using *dex2jar*[1] and *jd-gui*.[2] *dex2jar* is a tool that decompiles the .dex file inside the APK to a .jar file, which is a combination of Java classes aggregated as a single file; *jd-gui* further unpacks a .jar file into separate .class files. Some degree of obfuscation is inherently implemented during the compilation of the APK, which means our reverse-engineered code loses all the method and property names. However, given the fact that Java is a static strong typed language, class names are still well preserved, which can reveal sufficient information for subsequent analysis.

A careful analysis reveals that no functionality is implemented in Java per se. Instead a `NOTICE` file indicates that the core functionality is implemented using the *Xamarin* cross-platform C# application development tool,[3] which allows creating a single application in C# that can be compiled into Android, iOS, and Windows apps. Indeed, C# code was compiled with MonoVM to shared objects and the Java code is responsible for linking the classes in the shared objects and constructing the overall functionality of the app.

Knowing that the overall functionality lies in the shared objects, we extract the C# code from these objects, seeking to understand the functionality of the

[1] dex2jar Github, `https://github.com/pxb1988/dex2jar`
[2] Java Decompiler, `http://java-decompiler.github.io/`
[3] Xamarin, `https://dotnet.microsoft.com/apps/xamarin`

app and reverse-engineer its features. Shared objects built with Xamarin act as wrappers of Dynamic-Link Libraries (DLLs), which hold the actual functionality of the app. We extract these DLLs using a small script [2]. Lastly, we use *JetBrains dotPeek*[4] to decompile DLLs and retrieve the original source code.

Overall, the app consists of 88 DLLs with a total of 9,990 classes. However, the main functionality of the app is within the *Core* DLL, with 282 classes.

**Dynamic Analysis:** We split the dynamic analysis into two different phases: one concerning the communications between the app and the server, and the second focusing on analysing the internals of the app and what is stored in the phone once the app is installed.

**Phone internals:** Android is a mobile operating system based on the Linux kernel. The default installation restricts the access to multiple files, in order to prevent novice users from deleting/modifying critical functionality. However, this also means that the default version of Android does not allow a user to view the files any app uses/creates. Therefore, in order to further analyse the behaviour of the app, we use a rooted Android phone. *Rooting* is the process of allowing Android smartphone users to attain privileged control of the operating system; this can be done by asking the manufacturer of the phone to provide a code to de-activate the smartphone's protections. With a rooted phone, we can see what files our target m-tickets app would use upon execution. All the app-related information is stored in the `/data/data` folder as shown below:

```
net.corethree.lothianbuses
├── cache
├── code_cache
│   └── com.android.opengl.shaders_cache
├── databases
│   ├── com.microsoft.appcenter.persistence
│   ├── corethree
│   └── google_app_measurement.db
├── files
│   ├── .config
│   │   ├── activated_tickets.xml
│   │   ├── alert_notifications.xml
│   │   ├── data.json
│   │   ├── ticket_last_opened
│   │   └── ticket_open_dts
│   ├── .local
│   │   └── share
│   └── appcenter
│       └── database_large_payloads
└── shared_prefs
    ├── AppCenter.xml
    └── net.corethree.lothianbuses_preferences.xml
```

[4] dotPeek – Free .NET Decompiler and Assembly Browser, `https://www.jetbrains.com/decompiler/`

We are now able to read and analyse all information the app saves and how information storage is handled. Nevertheless, to be able to modify this information, we first need to disable *Security-Enhanced Linux*, a kernel security module that provides a mechanism for supporting access-control security policies. In this case, it would not allow the execution of any program, if there was any tampering of the files by an external process. This avoids malicious apps from stealing data from other apps. In order to disable SE-Linux, it is sufficient to obtain a root shell via the Android Debug Bridge (adb), and type `setenforce 0`.

Additionally, there are occasions where the behaviour of the app may be unexpected, therefore we also use *Frida*[5] to trace events. Frida is a dynamic code instrumentation toolkit that allows the injection of snippets of JavaScript or own library into native Android apps. We use this tool to trace the files being opened at certain points or which functions were triggered at certain times.

**Communications:** Modern day apps consist of two main parts: the app itself and the server with which it communicates. On the app side, we perform static analysis and examine the phone internals. However, the extraction of information from the server is not straightforward and we can only attempt "black box" penetration testing. This consists of performing a vulnerability analysis without access to any of the server's source code. As such, we can observe what messages go to and come from the server, but not the server's inner logic, which makes it hard to identify flaws.

For this part, we built a man-in-the-middle (MITM) setup, using an Alfa Atheros AR9271 Wi-Fi adaptor to set up a controlled hot spot on a laptop, to which the phone connects. We then route the traffic from the adaptor to Burp,[6] an integrated platform for performing security testing of web applications. Additionally, we install Burp certificates on the phone, so that the phone would trust the communications. Finally, the laptop connects to the Internet using its integrated Wi-Fi adopter, thereby allowing to intercept and modify whatever the app running on the phone sends and receives from the server.

We notice the phone compresses requests prior to transmission, hence we load a dedicated module into Burp to decompress requests for inspection.

### 4.2 Connectivity and Availability Analysis

A key concern for Lothian Buses app users is the app's availability. It has been reported that in some cases the app would stop working and require Internet connection in order to start, or would take too long to launch even when a connection is available [7]. Therefore, we analyse the minimum Internet connection speed required and the amount of bandwidth consumed when the app launches.

To this end, we use *BradyBound*,[7] an app that throttles the phone's Internet connection speed down to a user-defined value. Furthermore, we track the amount of data consumed by simply accessing *Settings>Apps>m-tickets>Data_usage* before and after starting the application, and calculating the difference. We execute all tests with a Motorola Moto G (3rd Generation) running Android v6.0.1.

[5] Fida analyzer, `https://frida.re/docs/android/`

[6] Burp analyzer, `https://portswigger.net/burp`

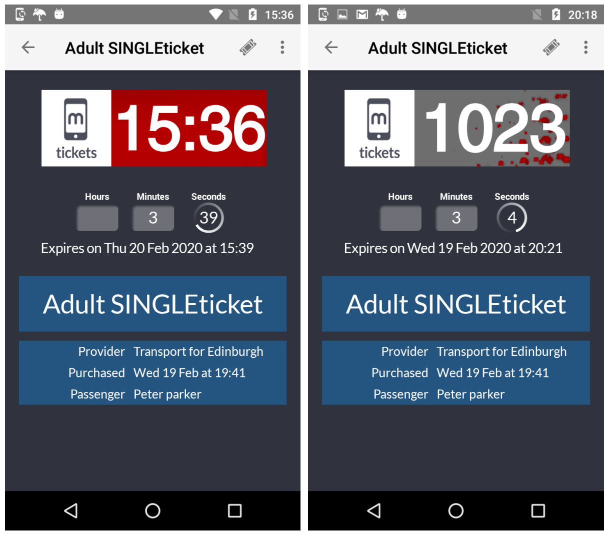
[7] BradyBound, `https://m.apkpure.com/bradybound/com.oxplot.bradybound`

## 5 Security Analysis

In this section we describe in detail the vulnerabilities found using the methodology described previously. Most weaknesses are exploited when the phone is off-line, taking advantage of the fact that the app can work without Internet access. We reveal how to predict, duplicate, and modify tickets as explained next. We also describe several functionality problems encountered in the app.

### 5.1 Generation of Tickets

One of the main goals of our study is to assess how securely ticket generation is handled and how difficult it would be for an attacker to craft valid tickets while evading payment. In order to accomplish this, we first need to understand how the app generates a legitimate ticket.



15:36
Adult SINGLEticket
m tickets 15:36
Hours Minutes 3 Seconds 39
Expires on Thu 20 Feb 2020 at 15:39
Adult SINGLEticket
Provider Transport for Edinburgh
Purchased Wed 19 Feb at 19:41
Passenger Peter parker

20:18
Adult SINGLEticket
m tickets 1023
Hours Minutes 3 Seconds 4
Expires on Wed 19 Feb 2020 at 20:21
Adult SINGLEticket
Provider Transport for Edinburgh
Purchased Wed 19 Feb at 19:41
Passenger Peter parker

(a) Ticket time view. (b) Ticket token view.

**Fig. 2.** Screenshots of an active ticket, alternating between a view of the current time (left) and the daily token (right).

Once a ticket is purchased, the user has the option to activate it whenever they board the bus. An activated ticket has a certain expiration time, which depends on ticket type (e.g. single ticket, day ticket, etc.), which bus drivers can check when presented with a view of the running app, as exemplified in Fig. 2. The ticket comprises several distinctive elements:

- *Top Title* – Describes the ticket type at the top of the ticket.
- *Watermark* – Visible in the central part of the ticket, comprising the m-tickets logo and a dynamic text block showing the current time and a numeric **token** on a changing background, in an alternating fashion. The numeric token is the same for all tickets activated during the same day, i.e. it is not unique to a ticket.

– *Remaining time* – Small countdown in the centre showing the remaining time until the ticket becomes invalid.
– *Lower body* – Shows information including ticket type, ticket provider, date of purchase, and passenger's name.

One implementation decisions made by the app developer is the activation and generation of tickets without Internet connectivity. The downside to this is that the app itself is in charge of generating the ticket, and not the server. This means an attacker with access to the source code could attempt to understand and replicate the process of generating tickets. Clearly, the numeric token is what bus drivers check in order to decide whether a ticket is valid. Hence, understanding how valid numeric tokens are obtained can compromise the underlying mechanism.

Analysing the source-code, one particular function stands out, namely `GenerateWatermark()`, located inside the `Core.Utilities` module. This function will be called whenever a ticket is activated, performing the following computation:

$$token = \left\lfloor \frac{(x-c)^2}{seed} \times 10^4 \mod 10^4 \right\rfloor, \tag{1}$$

where $\lfloor \cdot \rfloor : \mathbb{R} \to \mathbb{Z}$ denotes the floor function, $c$ is a date constant with value 01/01/1990 and $x$ represent the current date. The app uses this formula to create the numeric tokens, which are displayed to drivers for validation when the passenger is boarding the bus.

To accurately predict a token, it is necessary to understand how the *seed* variable is obtained. By performing a text pattern search through all of the app's files, we identify a particularly interesting string, namely `''Ticket.Seed'': ''71473''`, located in `files/.config/data.json`. Creating numeric tokens with the logic shown above and this *seed* value across different days, and comparing against tokens for the same days embedded in legitimate tickets, the values match perfectly. This means that the alleged *seed* is nothing but a hidden hard-coded value, rather than an actual seed of a pseudo-random sequence.

Besides, although the existence of the modulo and the floor operations in Eq. 1 makes this computation irreversible, we show in Fig. 3 that this function exhibits obvious periodic patterns, meaning that it does not qualify as a one-way function. As shown in Fig. 3, the mapping between current date $x$ and the token value presents a period-like relation, and the period gradually becomes longer as more time elapses from the fixed referenced date $c$. Thus an attacker can simply modify the system date and collect some data to recover the underlying function through trial and error.

**Finding:** An attacker can retrieve the procedure and relevant variables (which are unfortunately hard-coded) from the app source code, easily generate a valid numeric token for the current day, and embed that into a Graphics Interchange Format (GIF) image that resembles a genuine ticket, thereby evading payment. We also conclude that reverse-engineering of the application is not necessarily needed to predict the token of any future date, since the token generation algorithm reveals naive periodic patterns.

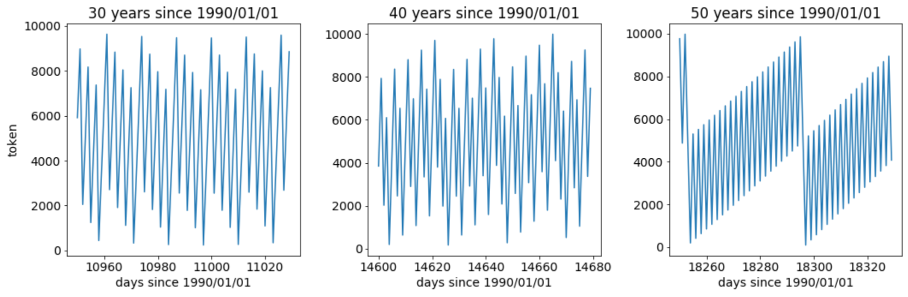


**Fig. 3.** Graphical illustration of token values, as the time since 01/01/1990 grows.

## 5.2 Re-activation of Expired Tickets

If one can already predict tickets, what would be the purpose of reactivating expired tickets? Predicting a ticket is one thing, but generating animated images on a phone is not straightforward. An attacker may need to replicate the layout of the official app to perfection and build a new app from scratch in order to exploit the vulnerability discussed in the previous subsection.

Therefore, we investigate whether it may be possible to reactivate an expired ticket, by analysing how the app saves the state of tickets. To this end, we examine the changes made on the app whenever a ticket is activated, first saving all the files in the home directory of the app prior to the activation of a ticket, then comparing them against those changed after the ticket expired.

```
try {
    Runtime.getRuntime().exec("su -c rm -rf /data/data/net.corethree.
         lothianbuses");
    Runtime.getRuntime().exec("su -c rm /sdcard/.storage/atl.txt");
    Runtime.getRuntime().exec("su -c cp -rp /data/data/tickets /data/data
         /net.corethree.lothianbuses");
    Toast errorToast = Toast.makeText(MainActivity.this, "Tickets
         restored!", Toast.LENGTH_SHORT);
    errorToast.show();
} catch (IOException e) {
    Toast errorToast = Toast.makeText(MainActivity.this, "Was not
         successfull!", Toast.LENGTH_SHORT);
    errorToast.show();
}
```

**Fig. 4.** Source code of demo app exploiting ticket re-activation vulnerability uncovered.

Most files seem to be modified, however the app would not make any requests over the Internet connection. All of the modified files are inside the `net.corethree.lothianbuses` folder, except for a small `/.storage/atl.txt` file created after the activation of the first ticket. After analysing the decompiled app code, it is clear that this file is just a back up of `activated_tickets.xml`, a file used to store the serialised activated tickets. This means that the content of `net.corethree.lothianbuses` is the representation of the state of the app. Hence, we can save its contents, activate as many tickets as previously purchased,

```
DateTime universalTime = app.CheckInLastSuccessfulTimestamp.ToUTCDateTime
    ().ToUniversalTime();
        if (universalTime < DateTime.UtcNow.AddDays(-5.0))
          app.CheckIn_BlockSession = true;
        else if (universalTime < DateTime.UtcNow.AddDays(-3.0))
          app.CheckIn_ShowWarning = true;
```

**Fig. 5.** Code snippet mitigating the reactivation of expired tickets.

and then swap the saved folder with the one used by the app, thereby restoring all the tickets as if the app was never opened in the first place. To facilitate repeated testing of this vulnerability and demonstrate the simplicity of the attack, we build a small app, which exploits this process, as detailed in Fig. 4.

In the above, we save the state of `net.corethree.lothianbuses` into a folder called `tickets`, and then use the app to substitute the files in the official app with those saved in this folder. However, it appears that after one week of testing, the vulnerability can no longer be exploited. Since our exploit would return to the state of the app after purchasing the tickets, from the apps point of view we had not been connected to the Internet for more than 5 days, which is one of the security measures that Corethree seem to have implemented. However, by the very fact that this is a response to a certain event, we expect to find the relevant implementation in the app's source code. Indeed, the code checks if the value `CheckInLastSuccessfulTimestamp` minus 5 days is less than 0, as shown in Fig. 5, where `CheckInLastSuccessfulTimestamp` was extracted from the timestamp saved as `CILST` in the `net.corethree.lothianbuses_preferences.xml` file, as shown in Fig. 6 (see line 4).

```
1 <?xml version='1.0' encoding='utf-8' standalone='yes' ?>
2 <map>
3  ...
4  <string name="CILST">2020-03-02 14:47:50Z</string>
5  <string name="ShowVouchersDownloadedNotifications">True</string>
6  <string name="NSSC">b0cc9f95ba012d9c3cca728af8379307</string>
7  ...
8 </map>
```

**Fig. 6.** Excerpt from XML file containing m-tickets app preferences.

However, the app does not check whether `CILST` is larger than the current time, meaning that an attacker can set the `CILST` to year 2030, and the exploit would work for the next 10 years.

**Finding:** By restoring the application state prior to ticket activation and modifying the XML file containing the app preferences, an attacker can reactivate expired tickets, which stay valid for any specified duration.

## 5.3 Modification of Tickets

Being able to re-activate tickets, next we explore the different type of tickets the app offers and whether these could be modified by an adversary. Excluding the fact that the app offers different tickets for different routes, there are 2 main

```
    ...
    "Name":"Adult SINGLEticket",
    "Subtitle":"Purchased Wednesday, 19 February 2020",
    "SortOrder":"0",
    "TTL":"3600",
    "Language":"",
    "TimeStamp":"2020-02-19T19:41:49.436Z",
    "CommonChildType":"Node",
    "AncestorIDArray":["pXTloFK","tgadTUCW_paymentsuccess"],
    "ComparisonHash":"3f484560fc614c438f194b5f419b88be",
    "Lifetime":5,
    "Interval":0,
    "Tags":{"Voucher.TypeID":"792c-56f8-403d-aed1-8e11af0",
    ...
}
```

**Fig. 7.** JSON fragment of an Adult Single ticket.

type of tickets: *Single-Adult* and *All-day* tickets. We purchase both types and activate them on the same day, to understand the technical difference between them. Perhaps unsurprisingly, the two are virtually the same, except that a user has 5 mins to use a *Single-Adult*, whilst the *All-day* ticket can be used for 24 hrs.

Knowing this, we analyse how the app identifies and stores different type of tickets, and find that the majority of ticket data is stored in the `data.json` file. The file is relatively large, containing information such as the app's layout, user tokens, URLs from where to download images and, most importantly, the characteristics of purchased tickets, as exemplified in Fig. 7.

Examining Fig. 7, note that tickets are defined by a JSON structure, which encompasses their characteristics. Therefore, our first attempt is to change the values of a *Single-Adult* ticket to those of an *All-day* ticket. However, after modifying `data.json`, the app would not open without an Internet connection, suggesting a security provision was implemented to prevent this exploit. We then use the Frida framework to trace precisely what happens internally when the app blocks the modified `data.json`. The trace reveals that both `data.json` and `lothianbuses_preferences.xml` are opened at program execution start. Reviewing the code again and identifying where these files are being used, it appears `data.json` is hashed with `ContentRoot` and the devices GUID, which are given in the `lothianbuses_preferences.xml` file. The hash is then compared with the value stored into `NSSC` (line 6 in Fig. 6). This procedure is illustrated in Fig. 8. Therefore, an attacker aiming to modify anything in the app, should change the hash stored in `NSSC` for a new one that passes the checks.

```
function onStart(context){
  content_root = extract_from_preferences("ContentRoot");
  nssc = extract_from_preferences("NSSC");
  data_json_str = read_file("data.json");
  md5 = MD5(data_json_str + "|" + content_root + "|" + context.guid);
  if (md5 == nssc){ parse_tickets(); }
  else{ delete_history(); }
}
```

**Fig. 8.** Pseudocode of procedure implemented by the m-tickets app to avoid modification of ticket characteristics. By reversing the hashing applied and retrieving key variables stored by the app, this can be circumvented.

```
 1 salt = "3497788798ffff545zhif8";
 2 shared_secret = "b70f578f-974d-4efd-a93a-43c8b4f6cd9d";
 3 function encrypt(plaintext){
 4   prs = HMACSHA1(shared_secret, salt); # pseudo-random string
 5   key = prs[: key_size / 8];
 6   IV = prs[: block_size / 8];
 7   cipher = AES(IV, key, plaintext);
 8   ciphertext = base64_encode(cipher);
 9   return ciphertext;
10 }
```

**Fig. 9.** Encryption logic implemented by the m-tickets app.

Since the hash is crafted based on values that we already have, we can write a small C# script to replicate the creation of the hash and use it to modify the app. We are now able to change any of the characteristics of a ticket. For example, we could make a single Adult ticket last for months, if we changed the ticket's "Lifetime" property.

**Finding:** By replicating the hashing mechanism applied to the tickets data store and overwriting key variables in the m-tickets preferences file, an attacker can extend the lifetime of tickets at will.

## 5.4 Hard-coded Keys and Tokens

After decompiling the app, we notice that some of the information being stored is encrypted, since the developers included custom cryptography classes in addition to imported C# crypto libraries. Although our proposed attacks do not exploit any encrypted information, it is still worth analysing the encryption algorithms, so as to understand if any potential weakness may exist once new functionalities or features are integrated into m-ticket.

The app adopts the Advanced Encryption Standard with Cipher Block Chaining (AES-CBC), a block cipher encryption scheme commonly used to provide strong confidential guarantees [4]. This algorithm uses three key instruments to ensure secrecy: a salt, an Initialisation Vector (IV), and a key. The salt is used to avoid brute-force attacks against the resulting cipher-text, the IV ensures semantic security, and the key is used to encrypt the actual plain-text. This algorithm by design is robust against both passive and active adversaries, but unfortunately, our analysis reveals that it is not utilised correctly, resulting in possible information leakage.

As shown in Fig. 9, both the key and IV are derived from a salt and a shared secret, which turn out to be hard-coded right above the encryption function (lines 1–2). That is to say, as long as an attacker obtains these strings, any encrypted information can be easily deciphered on Android phones. Whilst it is obvious that the seeds and secrets have to be stored locally for the program to work in an offline environment, the developers should have been mindful of how easy it is to decompile apps.

**Finding:** Although state-of-the-art encryption is adopted, key elements aimed at ensuring secrecy are hard-coded in the m-tickets app. Hence cipher-text is straightforward to reverse.

### 5.5 Root Checker Bypass and Enabling Screenshots

A key step in exploiting the re-activation and modification of tickets is the ability to have full control of the phone (root access), while maintaining full use of the app. For this reason, checking whether the app has a root checker was one of our first priorities after decompiling. Corethree implement a root checker function that looks for certain files or binaries, denying access to the app if found, as revealed in Fig. 10. Unfortunately, having a rooted phone, the system is not to be trusted. In this case the app asks the system to look for certain files, but since an attacker controls the system, they can manipulate the response stating that the relevant files do not exist. To showcase this, we use *Magisk hide*, a module of the Magisk manager,[8] which hides the root files from whatever app it is instructed to.

```
public bool isDeviceRooted(Context context)
   {
     return  tags.Contains("test-keys") ||
         File.Exists("/system/app/Superuser.apk") ||
         executeCommand(check_su_binary) != null ||
         isPackageInstalled("eu.chainfire.supersu") ||
         findBinary("su");
   }
```

**Fig. 10.** m-tickets root checker function.

Another feature Corethree implemented in the app is the inability to take screenshots whilst the app is in use, so as to prevent users from sharing screenshots of purchased tickets. However, if an attacker has root access to their phone, they can disable the permission granted to apps to block screenshots. In our case we used the *smali patcher* module from the Magisk rooter.

**Finding:** The app root checker can be bypassed, thereby enabling reverse-engineering and modifying of the original app functionality.

### 5.6 Password Reset Issues

The majority of vulnerabilities found up to now were in the app source code. However, one part of the ticketing ecosystem we do not have access to is the source code of the server logic. Hence, we carry out a "black box" analysis, by which we intercept the network traffic towards/from the server and seek to make sense of the back-end. In particular, we uncover two main problems with the password reset procedure.

To understand the vulnerabilities, let us first examine the standard behavior of a password reset. After requesting a password rest, the user would receive a URL of the form `https://passwordreset.corethree.net/<11upper-lower-casecharacters>`. This link would expire within 75 min after the reset request. However, it appears the user could request a password reset as may times as desired and the server would send a new link to reset the password, without

[8] Magisk, `https://magiskmanager.com/`

invalidating the old one. This means that an attacker could request many resets and increase the probability of guessing the victim's password reset link. Arguably, the probability of brute-forcing a valid URL online is relatively small. For instance, assuming a brute-force rate of $10^5$ attempts/sec and a reset password rate of $10^5$ requests/sec, the probability of guessing a valid URL is

$$p_{guess} = \frac{(10^5 \times (60 \times 75))^2}{52^{11}} = 2.69 \times 10^{-4}.$$

However, any further increase in compute power could lower the work factor.

Aside from this threat, the fact that the server allows a user to request as many password resets as desired, creates an opportunity for malicious actors who may exploit this weakness to launch Denial of Service (DoS)/Email flood attacks towards other companies or individuals, using the Lothian Buses server resources, further damaging the transport operator's reputation in the process.

**Finding:** Poorly implemented password reset mechanisms lowers the barrier to brute-forcing user credentials and launching DoS/Email flood attacks using the transport operator's computing infrastructure.

### 5.7 Availability

The main purpose of an e-ticket app for public transport is to enable users to purchase tickets and use them at any point in time. The service must be thus available at all times. Following recent reports about app availability issues [5], we decide to run a network test and analyse the Internet connection needed to run the app and amount of data exchanged over this. Unsurprisingly, during 10 tests whereby we open and close the app, the average amount of data consumed is 45kB and the time required to load the app did not vary with download speeds of 19kB/s and above. However, examining the source code again, we notice that whenever the app is opened and any error occurs for whatever simple reason, the app closes and all information is erased, as when a tamper attempt is detected. As a result, the app has to re-download all data and validate it before displaying it to the user when re-opened. This leads to a 400kB increase in data consumption and approximately 12s boot-up time with an un-throttled connection.

To avoid this nuisance, modern programming languages force the user to implement `try-catch` statements, which permit a program to continue executing even if a small part of it encounters an exception. However, the m-tickets app is peppered with `try` instructions that are not followed by an appropriate `catch` logic. This leads to frequent occasions where the program crashes or gets stuck.

**Findings:** A combination of aggressive error handling practice and inappropriate use of `try-catch` statements leads to a history of poor app availability. Occasional users will always be forced to have an Internet connection.

## 6 Recommendations

Given the security vulnerabilities identified in the m-tickets app, we propose a set of solutions that can be deployed to address the exposed problems. We also explain why some of the implementation decisions made by Corethree are insecure, and suggest simple alternatives.

## 6.1 Tickets

Clearly, the whole purpose of the app is the secure purchase and use of tickets. Security is largely an abstract concept that is not straightforward to measure [17]. However, in essence it should reflect how hard it is for an attacker to read or modify information they are not authorised to. Taking a look at the current design of an e-ticket in the m-tickets/Lothian Buses app and the weaknesses described in Sec. 5–5.3, to begin with, the validation of a ticket should not rely on the bus driver. This is because the process is prone to error, as the driver may fail to recognise the difference between a valid and a crafted ticket. Indeed, previous studies show that humans are the weakest link in information security [3].

Secondly, users can be selfish and app decompiling is increasingly accessible. Therefore, the process of ticket generation should not be client-side, to avoid users tampering with it in order to circumvent payment. Instead, this process should be entirely server-based, whereby the user receives a valid ticket upon purchase, but remains unaware of how it was created.

Thirdly, there is currently no way of knowing whether active e-tickets are being re-used. The task of deleting a used ticket is handled by the app, yet as shown in Sec. 5.2, a user can control the app's behaviour on their phone. The same applies to the illicit modification of the characteristics of tickets (Sec. 5.3). Once again, to circumvent these problems, the user should be provided with a ticket generated on the server side and which cannot be modified by the app.

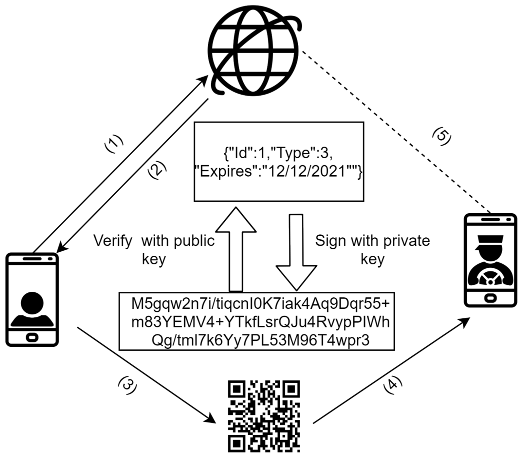

(1)
(2)
(5)
{"Id":1,"Type":3, "Expires":"12/12/2021""}
Verify with public key
Sign with private key
M5gqw2n7i/tiqcnI0K7iak4Aq9Dqr55+ m83YEMV4+YTkfLsrQJu4RvypPIWh Qg/tml7k6Yy7PL53M96T4wpr3
(3)
(4)

**Fig. 11.** Blueprint of proposed secure alternative ticketing system.

**Alternative ticketing system:** Strengthening the ticketing system may require a complete redesign. In what follows, we propose a simple alternative, which although arguably not flawless, mitigates the vulnerabilities identified. The proposed system consists of (1) a QR code validation protocol that substitutes driver-based visual validation; (2) an additional private app that bus drivers would use to validate tickets; and (3) an RSA signature algorithm to safely maintain the tickets.

We illustrate the envisioned alternative ticketing system in Fig. 11 and summarise its operation below.

**Step 1:** User sends a payment for some amount of tickets of certain type, which they want to purchase.

**Step 2:** Server crafts each ticket in JSON format, which contains all the information needed to identify the ticket, including a *Unique_id* to avoid ticket reuse; *Ticket_type* to specify if the ticket is Adult Single, All day, etc.; an *Expiry_date* to verify that the ticket is still valid. Once the JSON is crafted, the server would use a private key $k_{prv}$ to sign the JSON and send the result to the app.

**Step 3:** When the user wants to activate a ticket, the app builds a QR representation of the encrypted ticket and displays it on the phone's screen.

**Step 4:** The bus driver uses their app to scan the QR code. The app contains a public key $k_{pub}$, which is used to verify that the ticket has not been tampered with. It also checks that the unique ID was not used in the past. If the ticket appears valid, the app indicates approval and stores the ticket's unique ID.

**Step 5:** Periodically, the bus driver's app connects to the server and sends the unique IDs that were scanned. At the same time, it is updated with information of other valid/invalid unique IDs that have changed recently.

The downside to this system is that every bus driver must have a smartphone, which increases CAPEX. The advantages might out-weight the cost, since (1) the user only holds signed tickets and cannot craft tickets while subverting payment; (2) the public key could be made available to anyone, since it only serves in verifying if a ticket was tampered with; (3) the ticket duplication weakness is removed, since an attacker would have almost no time to use a copy of a ticket due to the unique IDs; and (4) modification of tickets becomes infeasible, since digital signatures are proven to be secure [8].

### 6.2 Hard-coding and Availability

Having a flawless program is almost impossible. However, historically communities have come together to create standards, so that users/developers have the means of checking the correctness of their programs. A widely-known security standard is the OWASP Secure Coding Practices [10], which lays out practices developers should follow to make a program secure. Hard-coding and Availability issues we found in the m-tickets app are covered in these standards. Hence we recommend following these checklists when developing future versions of the app, to avoid the same or other pitfalls.

### 6.3 Password Reset

Not limiting the number of password resets a person can request has implication on (1) user account security (as it simplifies brute-forcing); (2) can facilitate DoS attacks towards third parties; and (3) can damage the reputation of the app provider. To avoid these, developers could enforce, e.g. a 10-second restriction between each password reset. This would be unnoticeable to the user, since it is roughly the time it takes to check email, while adversarial actors would be unable to perform any of the attacks discussed in Sec. 5.6. Additionally, it is good practice to disable the last password reset link after issuing a new one for the same account.

## 7 Conclusions

In this paper analyse the security and robustness of the m-tickets system used by Lothian Buses, a leading UK transport operator. We identify a range of vulnerabilities pertaining to ticket generation and life-cycle, app functionality, and back-end logic. To mitigate these, we provide design recommendations which Corethree, the developer, should implement, especially given that parts of older highly-vulnerable versions of the ticketing app remain in use and suggest other iterations of the system might be at risk. This includes those sold to other transport companies in the UK. Lastly, we present the blueprint of an alternative ticketing system, which should help in the development of future secure apps supporting public transport worldwide.

## References

1. Corethree website. `https://www.corethree.net/`
2. Unpacking Xamarin mono DLL from libmonodroid_bundle.app.so. `https://reverseengineering.stackexchange.com/a/17330`
3. Accenture: Why humans are still security's weakest link (May 2019)
4. Doomun, R., et al.: AES-CBC Software Execution Optimization (Aug 2012)
5. Edinburgh News: Edinburgh commuters face more ticket app failures (Sept 2018)
6. Edinburgh Trams: TfE mtickets. `https://edinburghtrams.com/news/tfe-mtickets` (Aug 2018)
7. Google Play Store: Lothian buses m-tickets
8. Lindenberg, C., Wirt, K., Buchmann, J.: Formal Proof for the Correctness of RSA-PSS. IACR Cryptology ePrint Archive (Jan 2006)
9. Lothian Buses Limited: Consolidated Financial Statements 2018. First edn. (2019)
10. OWASP: Secure coding practices. `https://owasp.org/www-pdf-archive/OWASP_SCP_Quick_Reference_Guide_v2.pdf`
11. Reddit: Activists release code to generate free public transportation tickets. `https://www.reddit.com/r/manchester/comments/cyefu5/activists_release_code_to_generate_free_public/` (2019)
12. Statista: Number of smartphone users worldwide from 2016 to 2021. `https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/`
13. The Business Research Company: Transit and Ground Passenger Transportation (Public Transport) Global Market Briefing 2018. First edn. (2018)
14. The Telegraph: Public transport apps hacked to create free tickets and defraud operators (Sept 2019)
15. Wired: Hackers Crack London Tube's Ticketing System. `https://www.wired.com/2008/06/hackers-crack-l/` (Jun 2008)
16. Xu, Q., Erman, J., Gerber, A., Mao, Z., Pang, J., Venkataraman, S.: Identifying diverse usage behaviors of smartphone apps. In: ACM SIGCOMM IMC (2011)
17. Zalewski, J., et al.: Can we measure security and how? In: Proc. Annual Workshop on Cyber Security and Information Intelligence Research (2011)