Phase 2

Introduction

In October 2024, a privacy and security review was conducted for three Android messaging apps, namely, Bale, Eitaa, and Rubika. This recent Phase 2 assessment followed the Phase 1 review which was conducted in December 2023. The Phase 1 review focused on using static analysis and reverse engineering to evaluate encryption methods and platform-level privacy concerns. As in Phase 1, the project time was limited and did not allow for a total understanding of all app functionalities, therefore the assessment was focused to a list of primary concerns.

This Phase 2 assessment used dynamic analysis to validate the findings from Phase 1 and to explore the following concerns:

- *Interoperability:* The apps interoperate allowing users of different apps to message each other. Are communications between the target applications secure? What type of encryption is being used to enable this interoperability?
- *Unexpected Transmission of Private Data:* Do the apps activate any sensors (e.g., a user's microphone or camera) or send any user data (e.g., location, identifier, phone numbers, or other personally identifiable information) in an unexpected way?
- *Changes from Telegram:* Two of the apps rely heavily on Telegram code. How closely do the applications' implementations match that of the official Telegram app, and what, if any, significant changes have been made?
- *Use of AI:* Rubika is marketed as the Iranian version of WeChat, advertised as a preferred "all-in-one" multi-service application in Iran. The application's public-facing documentation claims that the application uses artificial intelligence (AI) for image analysis, e.g., to detect women who are not wearing a hijab. Is there any evidence this process occurs on client devices?
- *Security Review:* Do the applications contain design or implementation vulnerabilities that could be exploited by mobile application hackers?
- *Encryption:* What types of encryption are used in the apps overall?

APKs

The following application APKs were investigated in this assessment:

- Eitaa (v6.4.2, SHA256: 943d25d2cb842ee91e404922c9eeb7433158ba14ee5da821de3870cd92676731)
- Rubika (v3.7.5, SHA256: 9f4ca46bbcec994063376f18cc3c3f7adcdf7c41fd5de9eabaafc4c050d4da6d)

- Bale (v9.41.5, SHA256: 9bb94f028bb34e97123b26ca7baefd10c7191fa61b3c6ecbd1f4928a75bc3f8f)

A summary of the findings for each objective is described below.

Interoperability

All three apps could exchange messages with each other through a backend process called MXB Message Exchange. Static analysis revealed that MXB supported interoperability between Bale, Eitaa, Rubika, IGap, Gap, Soroush, and Chavosh messengers. To use this interoperability, users first had to activate the MXB feature in their messaging app's Settings menu. Once enabled, an MXB Register Request was sent from the app to the app's normal backend server. This registration request included the user's phone number, nickname, and avatar.

If a second user, using a different messaging app, wanted to message the first user (who was one of their Android contacts), the second user could tap a button on the contacts screen of their app to connect with the first user. This action sent an MXB GetUserRegisterInfo request, containing the first user's phone number, to the second user's messaging app backend server. In response, the messaging app's backend returned the first user's information and a list of apps they used. This process created a virtual MXB user on the second user's app, allowing them to message the first user.

The apps did not communicate with the MXB servers directly, and therefore, the exact functioning of the MXB interoperability was obfuscated. Further, the assessment team could not fully test the feature dynamically, being limited to a single valid account on the apps. However, based on the gathered information, the assessment team believes it likely that when a virtual MXB user was messaged, extra associated information about the virtual MXB user instructed the messaging app's backend to forward the sender's message, decrypted and in cleartext, to an MXB server. The MXB server would then presumably route the message to the correct destination app backend, from where it would be forwarded to a user's app, with the sending user being created as a virtual MXB user on the recipient's app.

Unexpected Transmission of Private Data

None of the apps employed end-to-end encryption, and therefore, all chat conversations and information about participants (e.g., their names, phone numbers, and contacts) were readable by the applications' backend servers. In the case of Eitaa, unsent draft messages were additionally reported to the application's backend server.

The assessment team did not note sensor-based cases of unexpected data sent, such as unexpected enabling of a user's microphone or camera.

Additionally, in all three apps, when users clicked URLs in messages that were sent to them, they were redirected to the application's backend server with the original URL in the query string. This would effectively allow the servers to monitor which websites were viewed by users within the app.

Changes from Telegram

Eitaa in particular was a fork of Telegram with few changes or additional features. The main differences with Telegram were found to be:

- Networking code was modified so that Telegram RPC messages were sent over HTTPS to Iranian servers rather than using Telegram's servers.
- The ability to have secret chats was removed. Telegram offers the option for end-to-end encrypted messaging, but no chats were end-to-end encrypted in Eitaa.
- The option to interoperate with other Iranian messengers was added, as described above in the interoperability section regarding MXB.
- The trends feature was added as a way to explore popular public channels.

Rubika included the Telegram libraries in the source code under the package name *org.rbmain.tgnet*; however, it was not observed being used in any of the chat functionality during dynamic analysis. Instead, Rubika used the package *ir.aaap.messengercore* to implement chat functionality, which used JSON over HTTPS with a superficial layer of custom encryption to obfuscate traffic. It is possible that Telegram's source code was being used for other functionality in the application, such as video/voice calls, but the assessment team was unable to confirm this via dynamic analysis.

Bale was not based on Telegram source code.

Use of AI

The assessment team found no evidence of AI being used to analyze message content, including photos, on the client device. As the backend servers were able to read the messages from all three apps, the assessment team believes that content filtering was occurring on the backend.

Security Review

The assessment team was unable to conduct a through security review of the applications in this phase of the assessment, primarily due to time constraints and challenges related to reverse engineering Bale's messaging protocol and defeating

its obfuscation. General application security concerns for each of the applications were discussed previously in the Phase 1 report.

Encryption

All three apps employed different forms of client-server encryption, but none had end-to-end encryption enabled to keep conversations between participants protected from the backend servers. Eitaa transmitted Telegram objects over HTTPS rather than the encrypted MTProto protocol. Rubika used a superficial custom AES encryption scheme that essentially transmitted the decryption key with each message. Bale was based on the Actor Messaging Platform's MTProto encryption that used AES encryption with a shared secret generated between the client and server.

Eitaa

Description

The *Eitaa* application did not make use of static or runtime obfuscation (with the exception of SSL Certificate Pinning, which was trivial to bypass), so the assessment team was able to inspect *Eitaa's* application traffic and use of the Android OS framework to confirm findings identified in Phase 1. In summary, all significant findings from the exploratory analysis were confirmed, along with additional privacy concerns including a lack of end-to-end encryption, transmission of URLs and unsent draft messages to *Eitaa's* servers, and harvesting of contact data in the application background.

While *Eitaa* did have the ability to access a user's microphone, camera, and GPS location data, the assessment team did not observe any inappropriate or unexpected attempts to access them, and modern versions of Android make it difficult to access these systems without a user noticing. Rather, the main privacy concern for this application appeared to be the fact that all user message content and browsing activity within the app were visible to *Eitaa* backend servers.

SSL Pinning

Rudimentary SSL Certificate Pinning was found to be in place, initially preventing the application from being intercepted by an intermediate proxy such as Burp Suite. The assessment team was able to bypass this mechanism using dynamic instrumentation with the *Frida* tool and the following public script:

```
function DisableSSLPinning() {
    var ArrayList = Java.use("java.util.ArrayList");
    var TrustManagerImpl = Java.use('com.android.org.conscrypt.TrustManagerImpl');
    // checkTrustedRecursive() recursively builds certificate chains until a valid
chain is found or all possible paths are exhausted
    TrustManagerImpl.checkTrustedRecursive.implementation = function(certs,
ocspData, tlsSctData, host, clientAuth, untrustedChain, trustAnchorChain, used) {
        // return empty trusted chain
        return ArrayList.$new();
    };
}

if (Java.available) {
    Java.perform(DisableSSLPinning);
} else {
```

```
    console.log("[!] Java VM is not available!");
}
```

After attaching to the *Eitaa* process with the *Frida* CLI tool and executing the above script, the assessment team was able to proxy application traffic for further testing.

Encryption

After bypassing SSL Certificate Pinning using dynamic instrumentation techniques, the assessment team was able to capture and inspect traffic as it flowed between the *Eitaa* client application and servers. The application used the open-source Telegram protocol to handle messaging; however, the Telegram source code was modified to send all messages to the */eitaa/index.php* endpoint on various *Eitaa* web servers. The following snippet shows a redacted HTTP request that was sent when a user sent a chat message on the platform. Note that the message content was transmitted to the server without any additional encryption beyond TLS:

```
POST https://sadegh.eitaa.ir/eitaa/index.php HTTP/1.1
Content-Type: text/stream
Content-Length: <redacted>
host: sadegh.eitaa.ir
Connection: Keep-Alive
User-Agent: okhttp/3.12.13

....<redacted identifier>........<redacted session ID>.....................This
message was sent to the server in cleartext.......................
```

Unexpected Transmission of Private Data

The assessment team developed custom *Frida* scripts along with using an intercepting proxy to monitor the application for unexpected capture and transmission of private data, including use of a user's microphone and camera, clipboard access, and access to a user's contact information and location information. Of these, only Android contact information was found to be transmitted periodically in the background, as shown in the following redacted HTTP request:

```
POST https://sadegh.eitaa.ir/eitaa/index.php HTTP/1.1
Content-Type: text/stream
Content-Length: <redacted>
host: sadegh.eitaa.ir
```

```
Connection: Keep-Alive
User-Agent: okhttp/3.12.13

....<redacted identifier>........<redacted session
identifier>..............<redacted contact phone
number>........John....Smith............<redacted contact phone
number>........Jane......Doe..........
```

Additionally, when searching for new contacts to add, the contact information was sent to the server in cleartext as the user typed:

```
POST /eitaa/index.php HTTP/2
Content-Type: text/stream
Content-Length: 112
Host: mostafa.eitaa.ir
Connection: Keep-Alive
Accept-Encoding: gzip, deflate, br
User-Agent: okhttp/3.12.13

....<redacted identifier>...<redacted session identifier>.............I'm searching
for a contact..............
```

A notable finding was that draft messages that were typed but never actually sent were transmitted to the server at somewhat random intervals. For example, the following redacted HTTP request was observed some time after typing a message in a private chat but with the user never hitting the send button:

```
POST https://alireza.eitaa.com/eitaa/index.php HTTP/1.1
Content-Type: text/stream
Content-Length: <redacted>
host: alireza.eitaa.com
Connection: Keep-Alive
User-Agent: okhttp/3.12.13

.....<redacted identifier>........<redacted session
identifier>................Draft message.........
```

This feature was most likely related to Telegram's ability to save draft messages between devices. However, it also enabled the *Eitaa* server to see unfinished user messages as they were being typed and before they were sent.

Eitaa was also found to detect URLs in chat messages and send them to the server when users clicked links that were sent to them. The server redirected the user to the site, as shown in the following redacted HTTP request and response.

Request:

```
GET https://search.eitaa.com/?url=https%3A%2F%2F<redacted>.com HTTP/1.1
host: search.eitaa.com
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: <redacted>
X-Requested-With: org.chromium.webview_shell
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
```

Response:

```
HTTP/1.1 302 Found
Server: nginx
Date: <redacted>
Content-Type: text/html; charset=UTF-8
Connection: keep-alive
Cache-Control: private, must-revalidate
Location: https://<redacted>.com
pragma: no-cache
```

Frida Scripts

The following script was used to hook Android framework calls to relevant camera and microphone methods to alert the assessment team if the sensors were being accessed unexpectedly:

```
Java.perform(function() {
        var audioRecord = Java.use("android.media.AudioRecord");
```

```
audioRecord.startRecording.overload("android.media.MediaSyncEvent").implementation
= function (v) {
                console.log("startRecording(MediaSyncEvent) called");

audioRecord.startRecording.overload("android.media.MediaSyncEvent").call(this, v);
        };
        audioRecord.startRecording.overload().implementation = function (v) {
                console.log("startRecording() called");
                audioRecord.startRecording.overload().call(this, v);
        };
        var camera = Java.use("android.hardware.Camera");
        camera.takePicture.overload("android.hardware.Camera$ShutterCallback",
"android.hardware.Camera$PictureCallback",
"android.hardware.Camera$PictureCallback",
"android.hardware.Camera$PictureCallback").implementation = function (v) {
                console.log("camera.takePicture() called");
        };
        camera.takePicture.overload("android.hardware.Camera$ShutterCallback",
"android.hardware.Camera$PictureCallback",
"android.hardware.Camera$PictureCallback").implementation = function (v) {
                console.log("camera.takePicture() called");
        };
});
```

The following script was used to hook attempts to access the device's location data:

```
Java.perform(function() {
        var locationManager = Java.use("android.location.LocationManager");
        locationManager.getCurrentLocation.overload("java.lang.String",
"android.location.LocationRequest", "android.os.CancellationSignal",
"java.util.concurrent.Executor", "java.util.function.Consumer").implementation =
function (v) {
                console.log("getCurrentLocation called");
        };
        locationManager.getCurrentLocation.overload("java.lang.String",
"android.os.CancellationSignal", "java.util.concurrent.Executor",
"java.util.function.Consumer").implementation = function (v) {
                console.log("getCurrentLocation called");
        };

locationManager.getCurrentLocation.overload('android.location.LocationRequest',
```

```
'android.os.CancellationSignal', 'java.util.concurrent.Executor',
'java.util.function.Consumer').implementation = function (v) {
                console.log("getCurrentLocation called");
        };
        locationManager.requestLocationUpdates.overload("java.lang.String", "long",
"float", "android.location.LocationListener").implementation = function (v) {
                console.log("requestLocationUpdates called");
        };
        var mapHelper = Java.use("ir.eitaa.helper.MapHelper");
        mapHelper.startLocationTracking.implementation = function (v) {
                console.log("startLocationTracking called");
        };
        mapHelper.stopLocationTracking.implementation = function (v) {
                console.log("stopLocationTracking called");
        };
});
```

The following script was used to hook attempts to access the Clipboard, e.g., for sniffing confidential data such as passwords:

```
Java.perform(function() {
        var clip = Java.use("android.content.ClipData$Item");
        clip.coerceToText.implementation = function (v) {
                console.log("read text from clipboard: " + v);
        };
});
```

Rubika

Description

The *Rubika* application contained a large amount of functionality, though the assessment team focused primarily on privacy concerns related to messaging. Unlike *Eitaa*, *Rubika* employed a custom encryption mechanism on top of SSL Certificate Pinning to obfuscate traffic being sent to the backend server. However, the encryption was largely superficial as the AES key used for encrypting messages was transmitted to the server along with the messages themselves. Details about this encryption mechanism and how the assessment team was able to bypass it are discussed below in the *Encryption* section.

In summary, *Rubika* servers had the capability to intercept and read all messages and contact data sent through the app. The assessment team used dynamic instrumentation techniques to attempt to identify other privacy concerns, such as inappropriate or unexpected use of a user's camera, microphone, and GPS location data, but no suspicious attempts to access these sensors were observed at the time of assessment.

SSL Pinning

Rudimentary SSL Certificate Pinning was found to be in place, initially preventing the application from being intercepted by an intermediate proxy such as Burp Suite. The assessment team was able to bypass this mechanism using dynamic instrumentation with the *Frida* tool and the following public script:

```
function DisableSSLPinning() {
    var ArrayList = Java.use("java.util.ArrayList");
    var TrustManagerImpl = Java.use('com.android.org.conscrypt.TrustManagerImpl');
    // checkTrustedRecursive() recursively builds certificate chains until a valid
chain is found or all possible paths are exhausted
    TrustManagerImpl.checkTrustedRecursive.implementation = function(certs,
ocspData, tlsSctData, host, clientAuth, untrustedChain, trustAnchorChain, used) {
        // return empty trusted chain
        return ArrayList.$new();
    };
}

if (Java.available) {
    Java.perform(DisableSSLPinning);
} else {
```

```
    console.log("[!] Java VM is not available!");
}
```

After attaching to the *Rubika* process with the *Frida* CLI tool and executing the above script, the assessment team was able to proxy application traffic for further testing.

Encryption

The assessment team was able to confirm all significant findings from the initial exploration of the apps in Phase 1, including that no actual end-to-end encryption was being used for private messages or channels. Instead, messages were encrypted using AES-CBC encryption with a key that was obfuscated and transmitted in the body of all relevant requests to the server. The following HTTP request snippet shows the structure of a normal chat request being sent to the *Rubika* servers:

```
POST https://messenger<redacted>.iranlms.ir/ HTTP/1.1
Content-Type: application/json; charset=utf-8
Content-Length: <redacted>
host: messenger<redacted>.iranlms.ir
Connection: Keep-Alive
User-Agent: okhttp/3.12.12

{"auth":"<redacted>","is_background":false,"api_version":6,"data_enc":"<redacted>"}
```

The *auth* JSON field contained the obfuscated AES key, and the *data_enc* field contained the base64-encoded encrypted message content. The assessment team developed the following Python script to decode the key and use it to decrypt the plaintext message content:

```
import base64
import json
from Crypto.Cipher import AES
from Crypto.Util.Padding import *

def decodeAuthKey(key):
    decoded = ""
    for k in key:
        if k.islower():
            c = ((32 - (ord(k) - 97)) % 26) + 97
```

```
        decoded += chr(c)

    return decoded

def makeKey(auth):
    key = ""
    str2 = auth[16:24] + auth[0:8] + auth[24:32] + auth[8:16]
    for s in str2:
        if s.islower():
            c = (((ord(s) - 97) + 9) % 26) + 97
        key += chr(c)
    return key

msg_json = json.loads(msg)

decoded_auth = decodeAuthKey(msg_json["auth"])
data_enc = base64.b64decode(msg_json["data_enc"])

aes_iv = b"\x00" * 16
aes_key = makeKey(decoded_auth).encode()

cipher = AES.new(aes_key, AES.MODE_CBC, aes_iv)

out = unpad(cipher.decrypt(data_enc), 16).decode()

print(out)
```

Processing the request with the script above produced the following output, confirming that the *Rubika* messaging servers had the capability to intercept and read all messages sent within the app:

```
{"input":
{"is_mute":false,"object_guid":"<redacted>","rnd":<redacted>,"text":"<PLAINTEXT
CHAT MESSAGE>"},"client":
{"app_name":"Main","app_version":"3.7.5","lang_code":"fa","package":"app.rbmain.a","temp_code":"<
```

Unexpected Transmission of Private Data

As expected, *Rubika* generally transmitted any content entered into the app by the user to various servers. In addition to message content, which included message

text, attachments, and voice message recordings, contact details were also transmitted, as shown in the decrypted code snippet below:

```
{"input":{"address_book_items":
[{"first_name":"<redacted>","last_name":"<redacted>","phone":"<redacted>"}]},"client":
{"app_name":"Main","app_version":"3.7.5","lang_code":"fa","package":"app.rbmain.a","temp_code":"<
```

Like *Eitaa*, *Rubika* was found to redirect users to a third-party site when users clicked on links in chat messages. The URL the user attempted to visit was obfuscated within the *q* URL query parameter, as in the following HTTP request.

Request:

```
GET https://url.rubika.ir/?q=<redacted>&k=<redacted> HTTP/1.1
host: url.rubika.ir
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: <redacted>
X-Requested-With: org.chromium.webview_shell
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Language: en-US,en;q=0.9
```

As shown in the response below, the application read the URL from the *q* parameter and set a *target_url* cookie containing the original value.

Response:

```
HTTP/1.1 302 Found
Server: nginx/1.26.1
Date: <redacted>
Content-Type: text/html
Content-Length: 0
Connection: keep-alive
Location: /index.html
Vary: Accept, Cookie, origin
Allow: GET, HEAD, OPTIONS
X-Frame-Options: DENY
X-Content-Type-Options: nosniff
```

```
Referrer-Policy: same-origin
Set-Cookie: target_url="<redacted URL from original link>"; expires=<redacted> Max-Age=86400; Path=/
X-Forwarded-For: <redacted>
```

The application was also found to store the user's chat history and other account-related information locally on the device in the file */data/data/app.rbmain.a/databases/RubikaMessenger_0*. This file was a SQLite database containing all chat messages and detailed contact information, including first name, last name, date of birth, phone number, and author GUIDs. There was no immediate privacy risk associated with storing this information on the client device; however, it could be recovered if a malicious actor were to gain access to the device.

Frida Scripts

The assessment team leveraged all *Frida* scripts described in the *Eitaa* section of the report to monitor attempts by the application to access components such as the camera, microphone, or GPS location data. No suspicious attempts to access these sensors were observed at the time of assessment.

Bale

Description

In the static analysis of Phase 1, the assessment team gained the least insight into *Bale* due to its use of obfuscation. Unlike *Eitaa* and *Rubika*, the *Bale* APK uses Android's r8 minification which makes reverse engineering the application's source code significantly more difficult. There were therefore no concrete findings in Phase 1. During this Phase 2 assessment, the assessment team was able to use dynamic analysis to overcome this obstacle and understand how *Bale* messaging works.

Even with dynamic analysis the *Bale* application still posed several challenges. Notably, the assessment team found that HTTP requests were used for only a small portion of *Bale* functionality, while most network traffic used a custom protocol. The protocol was found to be MTProtoV2, which is not end-to-end encrypted. This protocol and the assessment team's evaluation of it is discussed below in the *Network Protocol* section.

SSL Pinning

After authenticating to *Bale*, a list of hashes was downloaded from http://hash.bale.ai/hashes-android-json. These hashes were found to be consumed by an unusual Java pinning library eu.geekplace.javapinning, which was built into the app. It was possible to bypass this pinning by calculating the public key SHA256 hash of the assessment team's proxy certificate. *Bale* HTTPS traffic could be intercepted by adding that SHA256 value for each host in the file where the list was stored at */data/data/ir.nasim/shared_prefs/ssl_pins.ini.xml*.

The assessment team found that only a small portion of application functionality used HTTP requests, mainly for the following functionalities:

- A list of hashes for SSL pinning was downloaded from http://hash.bale.ai/hashes-android-json
- Images were downloaded from https://siloo.bale.ai
- Hashes of static resources were downloaded from https://tooshle.bale.ai
- Dynamic configuration values were downloaded from https://assets.bale.ai/configs.json

Network Protocol

Most *Bale* traffic was sent to port 443 of the host *arbaeen.ble.ir*. While port 443 was used, the traffic was not sent over the TLS protocol; rather, a custom TCP protocol

with encrypted payloads was in use. The assessment team used *Frida* (see script below) to trace the callbacks that opened this socket and discovered that the protocol in use was MTProto V2 from the Actor Messaging Platform, rather than MTProto from Telegram.

The Actor Messaging Platform is an abandoned instant messaging app that was built by ex-Telegram developers. The last public commit was in December 2016. *Bale* is largely a fork of this platform, with some notable changes.

One interesting change is that Actor double encrypts messages, using both the AES algorithm and the Russia-developed Kuznyechik block cipher in sequence. At the time of assessment, *Bale* removed the Kuznyechik block cipher and only used AES encryption for data payloads.

The MTProto V2 protocol is described in the following documentation: https://github.com/actorapp/actor-platform/tree/master/docs/protocol.

Encryption

On first start up of *Bale*, the app performed a Diffie-Hellman key exchange with the *Bale* server to generate a shared *auth_master_key*. An *auth_id* value was created based on the first eight bytes of the SHA256 hash of this master key. These values were saved in the file */data/data/ir.nasim/shared_prefs/properties.ini*. In future exchanges with the server, the *auth_id* value was sent to identify the current user and to determine the shared encryption key for the server to use.

Message encryption and decryption in *Bale* used the AES-128 algorithm with a random initialization vector transmitted in plaintext and a 16-byte slice of the auth master key as the encryption key.

Overall, the use of a shared symmetric key meant the server was able to observe and log all messages sent between *Bale* users. The Actor Messaging Platform was noted to contain end-to-end encryption functionality, but the *Bale* app was confirmed to not use this functionality and the Actor Messaging Platform end-to-end encryption code appeared to be removed from *Bale*. The use of an abandoned custom protocol rather than industry-standard TLS for protecting user message confidentiality means that user messages may also be at risk from third-party adversaries who can discover protocol flaws; however, a protocol review was not in scope for this assessment.

Additional Functionality

While *Bale* was built on the Actor Messaging Platform, a large portion of custom functionality had been added which used Protobufs sent over the MTProtoV2

protocol. The assessment team used dynamic instrumentation tools to log the Protobufs sent while browsing the app's other features, which include *Flow*, a stories-like feature, and *Services*, which offers a number of third-party integrations, such as with payment services, business chatbots, and ChatGPT. The assessment team did not notice any particularly unusual behaviors here, though testing of financial functionality was out of scope of the assessment.

Unexpected Transmission of Private Data

The assessment team did not observe any unexpected attempts where *Bale* accessed a user's microphone, camera, or GPS location data, and modern versions of Android make it difficult to access these systems without a user noticing. In Phase 1 a structure was noticed which appeared to report user location to the *Bale* backend, but the assessment team did not find this data to be transmitted during dynamic testing. Similar to the other apps, the main privacy concern for *Bale* was that all user message content and browsing activity within the app was visible to the backend servers.

Frida Script

The script shown below was used to dynamically instrument the app and observe the data sent to *arbaeen.ble.ir:443*:

```
function encodeHex(byteArray) {
  const HexClass = Java.use('org.apache.commons.codec.binary.Hex');
  const StringClass = Java.use('java.lang.String');
  const hexChars = HexClass.encodeHex(byteArray);
  return StringClass.$new(hexChars).toString();
}

function protoReflect(arg0, arg1) {
  if (arg0.$className.indexOf("ai.bale.proto") !== -1) {
    console.log("==== PROTO REFLECT");
    console.log(arg1);
    console.log(arg0.toString());
  }
}

Java.perform(function x(){
  console.log("Inside java perform function");
  var my_class = Java.use("ir.nasim.rb2");
```

```
  my_class.b.overload('[B', '[B').implementation = function(iv, data){
    console.log("==== DECRYPT");
    console.log("IV  : " + encodeHex(iv));
    var out = this.b(iv, data);
    console.log("Data: " + encodeHex(out));
    return out;
  }

  my_class.d.overload('[B', '[B').implementation = function(iv, data){
    console.log("==== ENCRYPT");
    console.log("IV  : " + encodeHex(iv));
    console.log("Data: " + encodeHex(data));
    return this.d(iv, data);
  }

  my_class = Java.use("ir.nasim.rpd");
  my_class.a.overload('java.lang.String', 'com.google.protobuf.p0',
'com.google.protobuf.p0', 'ir.nasim.it3').implementation = function(arg0, arg1,
arg2, arg3){
    console.log("==== RPD");
    console.log(arg0);
    return this.a(arg0, arg1, arg2, arg3);
  }

  my_class = Java.use("ir.nasim.jx7");
  my_class.f.overload('java.lang.Object', 'java.lang.String').implementation =
function(arg0, arg1){
    protoReflect(arg0, arg1);
    return this.f(arg0, arg1);
  }

  my_class.g.overload('java.lang.Object', 'java.lang.String').implementation =
function(arg0, arg1){
    protoReflect(arg0, arg1);
    return this.g(arg0, arg1);
  }

  my_class.h.overload('java.lang.Object', 'java.lang.String').implementation =
function(arg0, arg1){
    protoReflect(arg0, arg1);
    return this.h(arg0, arg1);
```

```
  }

  my_class.i.overload('java.lang.Object', 'java.lang.String').implementation =
function(arg0, arg1){
    protoReflect(arg0, arg1);
    return this.i(arg0, arg1);
  }

});
```