Assessment Report

Introduction

During December 2023 a review was conducted of three mobile applications. Please note that this review was purposefully brief, the reviewers were given an aggressive timeline to meet so there are many cases where a total understanding of functionality was not possible.

This involved initial exploration reverse engineering of the following three Android APKs:

- *Eitaa* (v6.3.7, SHA256: f9dcd28ddb923d85e44c7f9ef7dec011763a7985e23e877da34feda7d27994c9)
- *Rubika* (v3.5.7, SHA256: f9102e5f0ca3a35b681d19455f8a10e9caa16d2622ac799adb5ba2660f6d8678)
- *Bale* (v9.10.35, SHA256: 5b778885affbbf1fbd55457582a5d8730eb767bc1971f50cbc44837346532178)

Each of these apps contain messaging functionality, and the main objective of the review was to determine whether public claims that these apps use end-to-end encryption (E2EE) for user to user messaging could be verified. A secondary goal (if time allowed) was to report obvious and notable security/privacy concerns for users of the apps; more work may be done on this topic in the future.

For the purposes of this review, E2EE is defined as a system whereby only the sender and the receiver of messages between devices are able to read their contents. Third parties, including the application server, are unable to read or modify the data since they don’t have the key.

In summary, the authors of this report did not find evidence of modern E2EE functionality between the end users in any of the three apps in scope for review. While the apps tended to connect to servers using the HTTPS protocol for network traffic transport, there was no E2EE protecting users against eavesdropping by the server. Additionally, several privacy and platform-level concerns were noted with these apps.

Eitaa was largely a clone of Telegram with the E2EE secret chat feature disabled.

Rubika contained numerous components, and all of them sent data using custom cryptography that had significant weaknesses and was not E2EE.

Bale was obfuscated and the reviewers are therefore less certain about E2EE functionality, but no recognized protocols for performing E2EE were found.

This review was limited to reverse engineering of the provided APKs, and the applications were not run or dynamically analyzed. The majority of the reverse engineering was performed using JADX.

If more in-depth analysis is needed the review team suggests working to overcome the operational challenges to ensure that the apps can be dynamically analyzed. Intercepting all requests made by the apps would allow a greater and more efficient understanding of the normal operation of the apps, which could lead to finding additional security and privacy concerns. Further, this would enable the team to gain a clearer picture on *Bale* which was obfuscated and therefore difficult to reverse engineer compared to the other apps.

Eitaa (E2EE)

Summary

The *Eitaa* application was largely built on top of open source Telegram code. By default, Telegram chats are not end-to-end encrypted, the MTProto protocol only encrypts data from the client to the server. To use end-to-end message encryption in Telegram, a user has to explicitly start a new secret chat. However, as confirmed by a point of contact who was able to run the app, the Eitaa application did not contain an option for secret chats, and secret chats were explicitly not supported. No additional encryption functionality had been added, therefore all messages sent by Eitaa users can be intercepted, read, and modified by the Eitaa server.

Backend Connections

The main changes to the Telegram source occurred in the *ir.eitaa.tgnet.ConnectionsManager* class. The purpose behind these changes appeared to be to wrap the Telegram protocols inside HTTPS requests to send to a custom list of datacenters run by Eitaa operators.

```
    public void fillDatacenters() {
         if (this.datacenters.size() == 0) {
             if (this.isTestBackend == 0) {
                 DataCenter dataCenter = new DataCenter(1);
                 this.datacenters.put(Integer.valueOf(dataCenter.datacenterId),
         dataCenter);
                 ArrayList arrayList = new ArrayList();
                 arrayList.add("alzheimer.eitaa.com");
                 arrayList.add("fateme.eitaa.com");
  [...]
                 arrayList.add("ghasem.eitaa.com");
                 arrayList.add("mohsen.eitaa.com");
                 arrayList.add("hossein.eitaa.com");
                 arrayList.add("ghasem.eitaa.ir");
                 arrayList.add("mohsen.eitaa.ir");
                 arrayList.add("hossein.eitaa.ir");
  [...]
                 arrayList.add("armita.eitaa.com");
                 arrayList.add("majid.eitaa.com");
                 arrayList.add("mostafa.eitaa.com");
                 arrayList.add("alireza.eitaa.com");
                 arrayList.add("hosna.eitaa.com");
```

```
[...]
                arrayList.add("armita.eitaa.ir");
                arrayList.add("majid.eitaa.ir");
                arrayList.add("mostafa.eitaa.ir");
                arrayList.add("alireza.eitaa.ir");
                arrayList.add("hosna.eitaa.ir");
[...]
            dataCenter2.addAddressAndPort("dev.eitaa.com", 443, 0);
            dataCenter2.addAddressAndPort("dev.eitaa.com", 443, 2);
            dataCenter2.addAddressAndPort("dev.eitaa.com", 443, 4);
        }
    }
```

It is presumed that the servers listed are running a Telegram-compatible backend server which first unwraps the payloads from the HTTP requests. *ir.eitaa.helper.http.HelperHttp* contained a simple class for sending HTTP requests. Line 355 of *ir.eitaa.tgnet.DataCenter* instantiated this class:

```
    public synchronized HelperHttp getGenericConnection() {
         if (this.connection == null || this.createNewConnection) {
             this.createNewConnection = false;
             this.connection = new HelperHttp(getCurrentAddress(0),
         getCurrentPort(0), "/eitaa/index.php");
         }
         return this.connection;
    }
```

getGenericConnection() was called by *sendMessagesToTransport()* in *ir.eitaa.tgnet.ConnectionsManager*. *sendMessagesToTransport()* is a major custom function that all *Eitaa* messages pass through. It is too long to reproduce here, but worth noting that it wraps custom *TL_clientRequest* objects, with the following fields:

```
    public class TLRPC$TL_clientRequest extends TLObject {
     public boolean appPause;
     public int buildVersion;
     public int flags;
     public boolean foregreoundConnection;
     public String imei;
     public boolean isData;
     public boolean isWifi;
     public String lang;
    public int layer;
    public byte[] packed_data;
    public String token;
}
```

The *imei* field is discussed further under the “Eitaa (Additional Concerns)” section, as it is more of a privacy concern than relating to end-to-end encryption.

Authentication

The Telegram authentication had been modified in minor ways. The code for two step verification was slightly modified from open source versions of Telegram with a new *TL_twoStep_sendCode* class added, but in practice appeared to work largely the same: using an auth code, password, and recovery email address.

The message payloads had been modified to add a token value. If the client did not have a token, or had an expired token, the *refreshToken()* method was run from *ir.eitaa.tgnet.ConnectionsManager*. This sent a *TL_AppInfo* object to the server, which contained the following fields:

```
public String app_version;
    public int build_version;
    public String device_model;
    public String lang_code;
    public String sign = "";
    public String system_version;
```

In response to this information about a user’s device, the server returned a new token, which was sent with all future requests. The purpose of this additional token was not clear considering it was additional to the existing Telegram authentication.

Eitaa (Additional Concerns)

The following section contains privacy concerns that were noted while inspecting *Eitaa*. These concerns are in addition to the lack of end-to-end encryption and involve functionality that has been added to the original *Telegram* source code.

IMEI

In the *TL_clientRequest* objects sent with each message to the server, the app attempts to include the user's International Mobile Equipment Identity (IMEI). Since Android 10, accessing the IMEI requires READ_PRIVILEGED_PHONE_STATE, so unless *Eitaa* is installed as a privileged app (e.g. a device owner app), it should not be able to access the IMEI on more recent Android versions. If the IMEI could not be fetched, a random value was used:

```
this.imei = preferences.getString("imei", UUID.randomUUID().toString());
```

URL Whitelists

Another addition to the Telegram code was a URL whitelist for the in-app browser. URLs were fetched by the main thread of *ir.eitaa.tgnet.ConnectionsManager*, calling the *saveUrlWhiteList()* method in *ir.eitaa.messenger.MessagesController*. This saved the whitelisted URLs locally to the application data directory, encrypted using AES-ECB . If, when browsing to a URL, it was not matched in the whitelist, then it was encoded and appended to "https://search.eitaa.com/?url=" first. If the URL was whitelisted, the normal Telegram in-app browser behavior occurred. This is a possible vector for tracking the browsing behavior of users in the app.

Platform Level Concerns

Since *Eitaa* was heavily based on *Telegram*, the team compared the Android manifest of *Eitaa* to that of a recent *Telegram* open source version (10.2.9). There were only minor differences between the two, and the review team did not note any significant new permissions or exported receivers added to *Eitaa*, except a new *PaymentsActivity*, with an exported receiver with the *eitaapay* Android scheme:

```
            <intent-filter android:icon="@drawable/ic_launcher"
        android:priority="1">
                <action android:name="android.intent.action.VIEW"/>
                <category android:name="android.intent.category.BROWSABLE"/>
                <category android:name="android.intent.category.DEFAULT"/>
                <data android:scheme="eitaapay"/>
            </intent-filter>
```

The purpose of this activity appeared to be to scan barcodes or open URLs that request payment, but the payments themselves occurred online and not using in-app code.

For completeness, below are counts of exported resources:

- Exported Activities: 7
- Exported Services: 6
- Exported Receivers: 5
- Exported Providers: 1

The team briefly attempted to locate the implementations of these resources in the code but ultimately did not identify any obvious malicious code paths stemming from them; rather, they pertained to copied *Telegram* code as the team had anticipated.

The following privacy-relevant permissions were noted, however these were all present in the *Telegram* manifest and the team was unable to confirm whether they were used by additional first party code within the time allotted for this review:

```
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.GET_ACCOUNTS"/>
    <uses-permission android:name="android.permission.READ_CONTACTS"/>
    <uses-permission android:name="android.permission.WRITE_CONTACTS"/>
    <uses-permission android:name="android.permission.MANAGE_ACCOUNTS"/>
    <uses-permission android:name="android.permission.READ_PROFILE"/>
    <uses-permission android:name="android.permission.WRITE_SYNC_SETTINGS"/>
    <uses-permission android:name="android.permission.READ_SYNC_SETTINGS"/>
    <uses-permission android:name="android.permission.AUTHENTICATE_ACCOUNTS"/>
    [...]
    <uses-permission android:name="android.permission.REQUEST_INSTALL_PACKAGES"/>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

The application was also found to enable backups, allowing an attacker with physical access to the device to download all private application data:

```xml
<application android:allowAudioPlaybackCapture="true" android:allowBackup="true"
        android:appComponentFactory="androidx.core.app.CoreComponentFactory"
        android:hardwareAccelerated="@bool/useHardwareAcceleration"
        android:icon="@drawable/ic_launcher" android:label="@string/AppName"
        android:largeHeap="true"
        android:manageSpaceActivity="ir.eitaa.ui.ExternalActionActivity"
        android:name="ir.eitaa.messenger.ApplicationLoader"
        android:requestLegacyExternalStorage="true" android:supportsRtl="false"
        android:theme="@style/Theme.TMessages.Start"
        android:usesCleartextTraffic="true">
```

Rubika (E2EE)

Summary

The *Rubika* app used message encryption, but it was not end-to-end, and in fact the encryption key was effectively transmitted to the backend server together with each encrypted payload. The encryption used in the *Rubika* application contained a number of significant flaws and provided no defense against an attacker that could read network traffic from the application.

The *Rubika* app contained multiple separate components including several different messenger services, however all of them appeared to eventually encapsulate messages in the same way. The app included a messaging service in *ir.aaap.messengercore*, another suite of services containing additional messenger code in *ir.resaneh1.iptv.model.MessengerInput*, and core libraries in *androidMessenger* which included a message proxy which understood the Telegram TLRPC message protocol.

To understand the lack of end-to-end encryption, this analysis begins with the authentication process and works towards the message sending and receiving functionality.

Auth

Authentication was performed in *ir.aaap.messengercore.LoginUtils*, and was based on a modified version of Telegram authentication. First, a code was requested for the user's phone number, and the phone number and the confirmation code received over SMS were then used to sign in. Additionally, *Rubika* generated an RSA public key (or fetched an existing one from storage) which was sent to the server when signing in:

```
    private void generateKeyIfNeeded(KeyValueStorageHelper keyValueStorageHelper,
         CoreEncryptionHelper coreEncryptionHelper) {
         if (getPrivateKey(keyValueStorageHelper) == null) {
             KeyPair generateKey = coreEncryptionHelper.generateKey();
             keyValueStorageHelper.setPublicKey(generateKey.getPublic());
             keyValueStorageHelper.setPrivateKey(generateKey.getPrivate());
         }
     }
```

coreEncryptionHelper.generateKey() was located in *androidMessenger.KeyEncryptionHelper.CoreEncryptionHelperImpl*, where it was noted that the RSA keys were 1024-bit, which is considered a small keysize and has not been recommended for use by NIST since 2010:

```
public KeyPair generateKey() {
     try {
         KeyPairGenerator keyPairGenerator =
     KeyPairGenerator.getInstance("RSA");

         keyPairGenerator.initialize(1024);
```

After being generated, the user's public key and private keys were stored using the *ir.aaap.messengercore.KeyValueStorageHelper* class:

```
public void setPublicKey(PublicKey publicKey) {
     this.publicKey = publicKey;
     if (this.keyEncryptionHelper == null) {
         return;
     }
     this.keyValueStorage.setString(Key.publicKey.name(),
     this.keyEncryptionHelper.toString(publicKey));
 }
```

Internally, *keyValueStorage* was implemented using Android Shared Preferences, in the file *androidMessenger.keyValueStorageHelper.KeyValueStorageImpl*. It is bad practice to store app secrets in shared preferences, since they are not encrypted, and can be accessed on rooted devices.

As the final part of the authentication flow, *ir.aaap.messengercore.LoginUtils* received an auth token from the server. This auth token was asymmetrically encrypted using the public key provided by the user client, so the client then used their private key to decrypt the auth token, and store it in the key value helper.

```
str4 = coreEncryptionHelper2.decryptRSA(signInOutput.auth,
          LoginUtils.this.getPrivateKey(keyValueStorageHelper));
```

This auth token was the output of the authentication procedure and used to identify the client in future exchanges with the server. However, it was unexpectedly also used as the key for message encryption, as described below.

Message Encryption

getMessageUtils().sendMessage() was the function used to send messages. It was called from *ir.aaap.messengercore.CoreMainClassImpl* and was passed the user's auth token as the first argument:

```
public int callSendMessage(final String str, final ChatType chatType, Message
    message, boolean z, final LoadListener<SendMessageResult> loadListener) {
    if (str != null && chatType != null) {
        final SendMessageException checkMessageInput =
    checkMessageInput(message);
        return
    getMessageUtils().sendMessage(getKeyValueStorageHelper().getAuth(), str,
    chatType, message, z, getNetworkHelper(), getRubinoUtils(), new
    LoadListener<SendMessageOutput>() {
```

sendMessage() called *sendMessageInner()* in *ir.aaap.messengercore.MessageUtils*. Higher-level network methods that sent data to the server backend were found in *ir.aaap.messengercore.network.NetworkHelperImpl*. All these methods called *this.network.sendV5OrV6()* to handle the communication, for instance see *sendMessage()* at line 461:

```
@Override // ir.aaap.messengercore.network.NetworkHelper
 public int sendMessage(String str, SendMessageInput sendMessageInput,
    RetryObject retryObject, NetworkHelper.ResponseListener<SendMessageOutput>
    responseListener) throws Exception {
    return this.network.sendV5OrV6(str, "sendMessage",
    toJson(sendMessageInput), retryObject, getListenerV5(responseListener,
    SendMessageOutput.class));
 }
```

The *sendV5OrV6()* method was in *androidMessenger.network.NetworkImpl*, which called *getSendObservable()*:

```
public int sendV5OrV6(String str, String str2, JSONObject jSONObject,
    RetryObject retryObject, Network.ResponseListener responseListener) throws
    Exception {
    return sendInner(getSendObservable(true, false, str, str2, jSONObject),
    str, null, str2, jSONObject, retryObject, responseListener);
 }
```

The third parameter to *getSendObservable()* was the auth token originally passed into this chain of methods via *getKeyValueStorageHelper().getAuth()*:

```
    private Observable<Response<ResponseBody>> getSendObservable(boolean z, boolean
        z2, String str, String str2, JSONObject jSONObject) throws IOException {
[...]
        JSONObject jsonInput = getJsonInput(z, z2, str, str2, jSONObject);
[...]
        return
        ApiRequestMessangerRx.getInstance().send(jsonInput).subscribeOn(Schedulers.m801io());
    }
```

getJsonInput() was also in *androidMessenger.network.NetworkImpl*:

```
   private JSONObject getJsonInput(boolean z, boolean z2, String str, String str2,
        JSONObject jSONObject) {
        NetworkHelper.VersionEncryptionHelper versionEncryptionHelper;
        MessangerInput2 messangerInput2 = new MessangerInput2(str);
[...]
        return messangerInput2.getJsonObjectV6(str2, jSONObject,
        this.versionEncryptionHelper.getPrivateKey());
    }
```

getJsonObjectV6() first called *getJsonObjectV5()*, in *androidMessenger.model.MessangerInput2*:

```
    public JSONObject getJsonObjectV5(String str, JSONObject jSONObject) {
        this.dataJson = jSONObject;
        this.method = str;
        JSONObject jSONObject2 = new JSONObject();
        try {
            String str2 = this.auth;
            if (str2 != null) {
                jSONObject2.put("auth", str2);
            } else {
                jSONObject2.put("tmp_session", this.tmp_session);
            }
            jSONObject2.put("api_version", 5);
            makeDataV5();
            jSONObject2.put("data_enc", this.data_enc);
        } catch (JSONException e) {
            MyLog.handleException(e);
        }
        return jSONObject2;
    }
```

In *getJsonObjectV6()*, the message was additionally signed with the user's RSA private key. Note that in the V5 format, the auth token was added directly to the message body. In the V6 format, the auth token was instead first processed by *EncryptionHelper.encodeChars()*:

```
    public JSONObject getJsonObjectV6(String str, JSONObject jSONObject, PrivateKey
        privateKey) {
        JSONObject jsonObjectV5 = getJsonObjectV5(str, jSONObject);
        try {
            jsonObjectV5.put("api_version", 6);
            String str2 = this.auth;
            if (str2 != null) {
                 jsonObjectV5.put("auth", EncryptionHelper.encodeChars(str2));
                 jsonObjectV5.put("sign", EncryptionHelper.signRsa(privateKey,
        jsonObjectV5.getString("data_enc")));
            }
        } catch (JSONException e) {
            e.printStackTrace();
        } catch (Exception e2) {
            e2.printStackTrace();
        }
        return jsonObjectV5;
    }
```

encodeChars() performed simple linear shifts to the auth token characters, which were easily reversible:

```
    public static String encodeChars(String str) {
        if (str == null) {
            return null;
        }
        char[] charArray = str.toCharArray();
        String str2 = BuildConfig.FLAVOR;
        for (char c : charArray) {
            if (Character.isUpperCase(c)) {
                c = ((29 - (c - 65)) % 26) + 65;
            } else if (Character.isLowerCase(c)) {
                c = ((32 - (c - 97)) % 26) + 97;
            } else if (Character.isDigit(c)) {
                c = ((13 - (c - 48)) % 10) + 48;
            }
            str2 = str2 + Character.toString((char) c);
        }
        return str2;
    }
```

makeDataV5() was the function that built JSON objects to be sent to the server. It encrypted the JSON message payload using *EncryptionHelper.encryptAuth()* and the user's auth token as the second argument:

```
    public void makeDataV5() {
        String str;
        if (this.data_enc != null) {
            return;
        }
        String str2 = this.auth;
        if (str2 != null && !str2.isEmpty()) {
            str = this.auth;
        } else {
            String str3 = this.tmp_session;
            str = (str3 == null || str3.isEmpty()) ? null : this.tmp_session;
        }
        if (str != null) {
            JSONObject jSONObject = new JSONObject();
            try {
                jSONObject.put("input", this.dataJson);
                jSONObject.put("client", this.client.getClientJsonObject());
                jSONObject.put("method", this.method);
            } catch (JSONException e) {
                e.printStackTrace();
            }
            this.client = null;
            this.method = null;
            this.api_version = "5";
            try {
                this.data_enc = EncryptionHelper.encryptAuth(jSONObject.toString(),
        str);
                this.data = null;
                this.dataJson = null;
            } catch (Exception unused) {
            }
        }
    }
```

The *encryptAuth()* function was in *androidMessenger.network.EncryptionHelper*, and used AES CBC encryption on the data:

```
    public static String encryptAuth(String str, String str2) {
        try {
            byte[] makeKey = makeKey(str2);
            IvParameterSpec ivParameterSpec = new IvParameterSpec(ivBytes);
            SecretKeySpec secretKeySpec = new SecretKeySpec(makeKey, "AES");
            Cipher cipher = Cipher.getInstance("AES/CBC/PKCS7PADDING");
            cipher.init(1, secretKeySpec, ivParameterSpec);
            byte[] doFinal = cipher.doFinal(str.getBytes());
            MyLog.m4128e("LogAPIMessenger", str);
            return Base64.encodeToString(doFinal, 0);
        } catch (Exception unused) {
            return null;
        }
    }
```

The *makeKey()* method in *androidMessenger.network.EncryptionHelper* performed easily reversible linear transformations on the auth code to turn it into an AES encryption key:

```
    static byte[] makeKey(String str) {
        String substring = str.substring(0, 8);
        String substring2 = str.substring(8, 16);
        String str2 = str.substring(16, 24) + substring + str.substring(24, 32) +
        substring2;
        StringBuilder sb = new StringBuilder(str2);
        for (int i = 0; i < sb.length(); i++) {
            if (sb.charAt(i) >= '0' && sb.charAt(i) <= '9') {
                sb.setCharAt(i, (char) ((((str2.charAt(i) - '0') + 5) % 10) + 48));
            }
            if (sb.charAt(i) >= 'a' && sb.charAt(i) <= 'z') {
                sb.setCharAt(i, (char) ((((str2.charAt(i) - 'a') + 9) % 26) + 97));
            }
        }
        return sb.toString().getBytes();
    }
```

Note also that the initialization vector used in the AES encryption used a static valid of all zeroes. AES-CBC encryption with a static initialization vector is vulnerable to multiple attacks that can recover plaintext.

```
    private static final byte[] ivBytes = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0};
```

However, the obvious attack against the encryption used in V6 messages would be to get the original value of the auth token by reversing the operation of *EncryptionHelper.encodeChars()* on the “auth” field in intercepted messages. Then encode this using the same operation as *makeKey()* to retrieve the AES key. Using this key, the full message payloads in the “data_enc” field could be decrypted. In previous message versions, this process is even more straightforward as the auth token is sent directly.

Rubika (Additional Concerns)

Platform Level Concerns

The team analyzed the application manifest to map the attack surface of the application and identify potentially malicious permissions and exposed services that could be abused to compromise the privacy and security of users. The following list contains the total number of each type of resource that was found to be shared with other applications:

- Exported Activities: 3
- Exported Services: 15
- Exported Receivers: 4
- Exported Providers: 1

The team briefly attempted to locate the implementations of these resources in the code but ultimately did not identify any obvious malicious code paths stemming from them.

The team noted the following permissions which could be used to undermine the security and privacy of the device:

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.READ_CONTACTS"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

The team also noted that the application permitted cleartext traffic, but there was no network security configuration file present. This results in cleartext HTTP traffic being permitted to all domains

```
<application android:allowBackup="false"
    android:appComponentFactory="androidx.core.app.CoreComponentFactory"
    [...]" android:usesCleartextTraffic="true">
```

The team noticed the package com.facebook.stetho.dumpapp.plugins.FilesDumperPlugin

```
public void dump(DumperContext dumperContext) throws DumpException {
    Iterator<String> it = dumperContext.getArgsAsList().iterator();
```

```
        String nextOptionalArg = ArgsHelper.nextOptionalArg(it,
        BuildConfig.FLAVOR);
        if ("ls".equals(nextOptionalArg)) {
            doLs(dumperContext.getStdout());
        } else if ("tree".equals(nextOptionalArg)) {
            doTree(dumperContext.getStdout());
        } else if ("download".equals(nextOptionalArg)) {
            doDownload(dumperContext.getStdout(), it);
        } else {
            doUsage(dumperContext.getStdout());
            if (BuildConfig.FLAVOR.equals(nextOptionalArg)) {
                return;
            }
            throw new DumpUsageException("Unknown command: " + nextOptionalArg);
        }
    }
```

The team was unable to locate all potential uses of the *Stetho* library within *Rubika* due to time constraints, however two instances were found to be used in the classes *ir.resaneh1.iptv.apiMessanger.ApiRequestMessanger* and *ir.resaneh1.iptv.apiMessanger.ApiRequestMessangerRx*, where *Stetho* hooks HTTP requests to the backend. One example, shown below, involves attaching a *Stetho* interceptor to all requests when initializing the REST API service:

```
    public void setRestApiService() {
        HttpLoggingMessanger httpLoggingMessanger = new
        HttpLoggingMessanger(this.currentAccount);
        if (MyLog.isDebugAble) {
            httpLoggingMessanger.setLevel(HttpLoggingInterceptor.Level.BODY);
        } else {
            httpLoggingMessanger.setLevel(HttpLoggingInterceptor.Level.NONE);
        }
        OkHttpClient.Builder addNetworkInterceptor = new
        OkHttpClient.Builder().addInterceptor(new Interceptor() { // from class:
        ir.resaneh1.iptv.apiMessanger.ApiRequestMessanger.1
            @Override // okhttp3.Interceptor
            public Response intercept(Interceptor.Chain chain) throws IOException {
                ApiCacheObject apiCacheObject;
                String str;
                IOException iOException;
                Response response;
                String str2;
        Request build =
chain.request().newBuilder().addHeader(HttpHeaders.CONTENT_TYPE,
"application/json").build();
        CacheDatabaseHelper cacheDatabaseHelper =
CacheDatabaseHelper.getInstance(((BaseController)
ApiRequestMessanger.this).currentAccount);
        String appVersion =
UpdateUtils.getAppVersion(ApplicationLoader.applicationContext);
        Buffer buffer = new Buffer();
        if (build.body() != null) {
            build.body().writeTo(buffer);
        }
        Charset charset = ApiRequestMessanger.UTF8;
        String str3 = null;
        MediaType contentType = build.body() != null ?
build.body().contentType() : null;
        boolean canCache =
ApiRequestMessanger.this.canCache(build.url().toString());
        if (contentType == null || contentType.subtype() == null || !
contentType.subtype().equals("json")) {
            apiCacheObject = null;
            str = BuildConfig.FLAVOR;
        } else {
            charset = contentType.charset(ApiRequestMessanger.UTF8);
            String readString = buffer.readString(charset);
            if (canCache) {
                try {
                    apiCacheObject = cacheDatabaseHelper.getApiCache("-",
readString, appVersion);
                } catch (Exception unused) {
                    str = readString;
                    apiCacheObject = null;
                }
            } else {
                apiCacheObject = null;
            }
            str = readString;
        }
        if (canCache && apiCacheObject != null && apiCacheObject.output !=
null && apiCacheObject.expiredTime.longValue() >
System.currentTimeMillis()) {
            return new
Response.Builder().request(build).protocol(Protocol.HTTP_2).message(BuildConfig.FLAVOR).
json"), apiCacheObject.output)).addHeader("fromCache", BuildConfig.FLAVOR).build();
        }
        try {
            response = chain.proceed(build);
            iOException = null;
        } catch (IOException e) {
            iOException = e;
            response = null;
        }
        if (response == null || !response.isSuccessful()) {
            DataCenterManager.getInstance().increastApiCouner();
            ApiRequestMessanger.this.setRestApiService();
            if (apiCacheObject != null && (str2 = apiCacheObject.output) !=
null) {
                return new
Response.Builder().request(build).protocol(Protocol.HTTP_2).message(BuildConfig.FLAVOR).
json"), str2)).addHeader("fromCache", BuildConfig.FLAVOR).build();
            } else if (iOException == null) {
                return response;
            } else {
                throw iOException;
            }
        }
        if (canCache) {
            BufferedSource source = response.body().source();
            source.request(Long.MAX_VALUE);
            String readString2 =
source.buffer().clone().readString(charset);
            try {
                str3 = ((MessangerOutput)
ApplicationLoader.getGson().fromJson(readString2, (Class<Object>)
MessangerOutput.class)).cache;
            } catch (Exception unused2) {
            }
            Long l = 0L;
            if (str3 != null) {
                try {
                    l = Long.valueOf(Long.parseLong(str3));
                } catch (Exception unused3) {
                }
                cacheDatabaseHelper.addOrUpdateApiCache(new
ApiCacheObject("-", str, appVersion, readString2,
Long.valueOf(System.currentTimeMillis() + (l.longValue() * 1000))));
            }
        }
        return response;
    }
}).addInterceptor(httpLoggingMessanger).addNetworkInterceptor(new
StethoInterceptor());
TimeUnit timeUnit = TimeUnit.SECONDS;
```

```
        restApiService = (RestApiMessanger) new
        Retrofit.Builder().baseUrl(DataCenterManager.getInstance().getApiUrl()).addConverterFact
        timeUnit).readTimeout(25L, timeUnit).writeTimeout(25L, timeUnit).build()).build().create
    }
```

Under the correct conditions, this could allow real time interception and modification of any network traffic bound for the REST API. The team was unable to confirm statically whether this library was used in any malicious contexts, but future work, including dynamic analysis, could increase certainty.

Bale (E2EE)

The *Bale* application was obfuscated, most likely using Proguard, and therefore significantly more difficult to reverse engineer than the other applications. Without performing dynamic analysis against the application, the review team were unable to make definitive statements about the use of end-to-end encryption, but some guesses are possible based on evidence that was recovered from the obfuscated package within the review timeframe.

While various encryption functionality was found in *Bale*, the team did not locate any obfuscated code that actually appeared to be performing end-to-end messaging encryption using a recognized protocol (such as the Signal protocol).

Communication with the server occurred over Google Protobufs, and the names of these survived in the obfuscated application. For instance, *MessagingOuterClass$RequestSendMessage* appeared to be used to send a message to other users. This class contained a “peer” field with the *PeersStruct* type, as did many classes in the application:

```
public final class MessagingOuterClass$RequestSendMessage extends
        GeneratedMessageLite<MessagingOuterClass$RequestSendMessage, a> implements
        com.google.protobuf.g1 {
[...]
    private PeersStruct$OutExPeer exPeer_;
    private Int32Value isOnlyForUser_;
    private MessagingStruct$Message message_;
    private PeersStruct$OutPeer peer_;
    private MessagingStruct$MessageOutReference quotedMessageReference_;
    private long rid_;
```

The peer class was defined at *ai.bale.proto.PeersStruct$Peer* and had “id” and “type” fields:

```
public final class PeersStruct$Peer extends GeneratedMessageLite<PeersStruct$Peer,
        a> implements ec0 {
[...]
    private int id_;
    private int type_;
```

The type enum was found at *ai.bale.proto.fc0*, where there was a “PeerType_ENCRYPTEDPRIVATE” type:

```java
public enum fc0 implements o0.c {
    PeerType_UNKNOWN(0),
    PeerType_PRIVATE(1),
    PeerType_GROUP(2),
    PeerType_ENCRYPTEDPRIVATE(3),
    UNRECOGNIZED(-1);
```

However, in all usages of this enum that the review team found, the "PeerType_ENCRYPTEDPRIVATE" type was never set. Only "fc0.PeerType_PRIVATE" and "fc0.PeerType_GROUP" were explicitly set by builders of Peer objects (for instance in *PeersStruct$OutPeer m1*).

Similarly, messages could be of several different types, see the following from *ai.bale.proto.MessagingStruct$Message*, and only one type appeared to support encryption:

```java
    public enum b {
        BANK_MESSAGE(1),
        BINARY_MESSAGE(2),
        DELETED_MESSAGE(3),
        DOCUMENT_MESSAGE(4),
        EMPTY_MESSAGE(5),
        JSON_MESSAGE(7),
        NASIM_ENCRYPTED_MESSAGE(8),
        ORDER_MESSAGE(9),
        PURCHASE_MESSAGE(10),
        SERVICE_MESSAGE(11),
        STICKER_MESSAGE(12),
        TEMPLATE_MESSAGE(13),
        TEMPLATE_MESSAGE_RESPONSE(14),
        TEXT_MESSAGE(15),
        UNSUPPORTED_MESSAGE(16),
        GIFT_PACKET_MESSAGE(17),
        PREMIUM_MESSAGE(18),
        NEW_PREMIUM_MESSAGE(19),
        BOUGHT_PREMIUM_MESSAGE(20),
        ADVERTISEMENT_MESSAGE(21),
        POLL_MESSAGE(22),
        CROWD_FUNDING_MESSAGE(23),
        ANIMATED_STICKER_MESSAGE(24),
        STORY(100),
        TRAIT_NOT_SET(0);
```

The type of message called *NasimEncryptedMessage* was defined at *ai.bale.proto.MessagingStruct$NasimEncryptedMessage*:

```
public final class MessagingStruct$NasimEncryptedMessage extends
        GeneratedMessageLite<MessagingStruct$NasimEncryptedMessage, a> implements
        com.google.protobuf.g1 {
[...]
    private com.google.protobuf.j key_;
    private int messageLength_;
    private com.google.protobuf.j message_;
    private com.google.protobuf.j signature_;
```

The review team could not determine from source code analysis under what circumstances Nasim Encrypted Messages were used, and how they were constructed. Further time spent on dynamic analysis would help determine whether these messages were actually used in practice (as opposed to standard *MessagingStruct$Message* unencrypted messages) and could give insights into how the keys and signatures are formed.

Bale (Additional Concerns)

Card Encryption

In a separate location (*ar.a*) to the potential messaging encryption noted earlier, AES encryption was discovered with a static, hardcoded key:

```
    public final String a(String str) {
        try {
[...]
            byte[] bytes2 = "J@NcRfUjXn2r5u8x".getBytes(forName2);
            v.g(bytes2, "this as java.lang.String).getBytes(charset)");
            SecretKeySpec secretKeySpec = new SecretKeySpec(bytes2, "AES");
            Cipher cipher = Cipher.getInstance("AES");
            cipher.init(2, secretKeySpec, ivParameterSpec);
            byte[] doFinal = cipher.doFinal(Base64.decode(str, 0));
[...]
    }
```

The usage of this form of easily reversible encryption occurred in the context of encrypting user's credit card data and their Nasim "Shaparak" Public Key in a local preferences file.

Location

During authentication, an *AuthStruct$AuthSession* payload was sent from the app to the server containing the following fields:

```
public final class AuthStruct$AuthSession extends
        GeneratedMessageLite<AuthStruct$AuthSession, a> implements gb {
[...]
    private int appId_;
    private int authHolder_;
    private int authTime_;
    private int id_;
    private CollectionsStruct$Int64Value lastActivityAt_;
    private CollectionsStruct$StringValue lastIpAddress_;
    private DoubleValue latitude_;
    private DoubleValue longitude_;
    private String appTitle_ = "";
    private String deviceTitle_ = "";
    private String authLocation_ = "";
```

The sending of user location via latitude and longitude values during each authentication could be considered another privacy concern.

Platform Level Concerns

The team analyzed the application manifest to map the attack surface of the application and identify potentially malicious permissions and exposed services that could be abused to compromise the privacy and security of users. The following list contains the total number of each type of resource that was found to be shared with other applications:

- Exported Activities: 2
- Exported Services: 7
- Exported Receivers: 6
- Exported Providers: 1

The team briefly attempted to locate the implementations of these resources in the code but ultimately did not identify any obvious malicious code paths stemming from them.

The team noted the following permissions which could be used to undermine the security and privacy of the device:

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.READ_PRIVILEGED_PHONE_STATE"/>
<uses-permission android:maxSdkVersion="29"
        android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.MANAGE_ACCOUNTS"/>
<uses-permission android:name="android.permission.AUTHENTICATE_ACCOUNTS"/>
<uses-permission android:name="android.permission.GET_ACCOUNTS"/>
<uses-permission android:name="android.permission.READ_PROFILE"/>
<uses-permission android:name="android.permission.USE_CREDENTIALS"/>
[...]
<uses-permission android:name="android.permission.REQUEST_INSTALL_PACKAGES"/>
```

The permission *READ_PRIVILEGED_PHONE_STATE* is usually reserved for trusted system applications and it would allow the app to query unique device identifiers, such as the IMEI, that would normally be unavailable to low privilege apps for privacy reasons. This could facilitate tracking users' activity, but it was unclear from static analysis whether it would be functional in practice, as the permission should be ignored for non-system apps per the Android documentation.

The team analyzed the Network Security Configuration and found that cleartext HTTP traffic was permitted to the following hosts:

```xml
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config cleartextTrafficPermitted="true">
        <domain includeSubdomains="true">192.168.0.233</domain>
        <domain includeSubdomains="true">192.168.43.221</domain>
        <domain includeSubdomains="true">127.0.0.1</domain>
        <domain includeSubdomains="true">10.0.2.2</domain>
        <domain includeSubdomains="true">ep.bale.tel</domain>
        <domain includeSubdomains="true">ep.bale.ai</domain>
        <domain includeSubdomains="true">hash.bale.ai</domain>
        <domain includeSubdomains="true">185.13.231.71</domain>
    </domain-config>
</network-security-config>
```

The team was unable to identify traffic that was sent to the hosts above due to time constraints on the assessment as well as obfuscation present in the APK.

Lastly, the application was found to use Google Firebase for backend persistence at the following URL:

https://najva-1104.firebaseio.com