Encrypted Communications It's not what it seems!

Neelkumar P. Patel (1101357), Jay Sukhadiya (1110211)
Department of Computer Science
Lakehead University
Thunder Bay, Ontario
Email: {npatel46, jsukhadi}@lakeheadu.ca

***Abstract*—Secure communications is when two entities communicate with each other and the communication is completely secured from a third party or the intruder. This type of communication process can be achieved using encryption and decryption mechanism. The implementation of Email encryption is performed with the intension of showing the working of encryption and decryption mechanism. This web application allows user to send an encrypted electronic-mail to any domain and it can be only decrypted using the receiver's private key. Whereas, messaging android application was developed to replicate existing WhatsApp application to see whether, it performs end-to-end encryption or it's not what it seems. The application allows users to securely communicate with each other via performing actual end-to-end encryption.**

***Keywords*—Encryption; E-Mail; Messaging Application; JAVA; Android, WhatsApp; End-to-End Encryption.**

I. Introduction

Encryption communication is basically known as cryptography, which is the art and science of making a system that is capable of providing information security. So, when two or more devices communicate via an application that features any kind of encryption, the information will be transmitted as a unreadable text rather than an insecure plain text. By using the right kind of encryption, one can build a system where only the people engaged in the communication can access the right data or information. Encryption algorithms are divided into two categories: Symmetric key encryption and Asymmetric key encryption. In symmetric key, only one secret key is shared between the users and is used for encryption and decryption. In asymmetric key, a public key and a private key are used for encrypting and decrypting the message respectively.

Most of the messaging applications or platforms provides encryption of user's data. It can be a myth, whether the encryption is actually performed, even if it is performed, can the encrypted data be easily decrypting by the organization as it can have access to private keys. Based on these scenarios, two different application were implemented to capture and understand the real life encryption and decryption mechanism taking place in various messaging application or platforms.

The Email encryption is a web application developed in JAVA language using Integrated Development Environment (IDE) Net Beans 8.2. It allows user to send encrypted mails to any electronic-mail (Email) id. The messages are encrypted using the public key of the receiver and can be decrypted using the private key of the receiver. This application was not made live to a particular domain. It completely works only on local host. Hence, the public and private keys were both stored locally.

An android application was developed named "ChatApplication", in JAVA using Android Studio 3.1.6. The project is completely online and uses real time Firebase database. It allows the user to register, login and perform chatting with other users. The public key is stored on the server and the private key of individual users is stored on their personal devices. Private key is not stored nor fetch anywhere from the server. This paper describes the implementation of both the application and conclude whether the real life encryptions is actually what it seems.

II. Email Encryption Implementation

The email encryption is a web application, where the both public and private keys are stored locally. Using this application, a particular users can send an encrypted mail to any person. The GUI allows user to enter the receiver's email id and the message, which is needed to be encrypted as shown in Fig. 1.

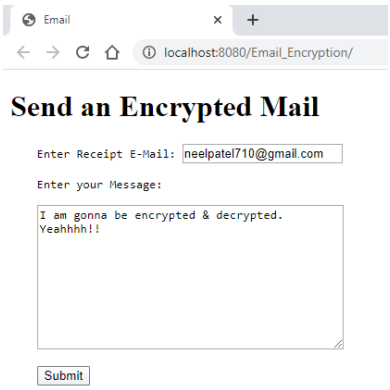


Fig. 1: GUI of Email application.

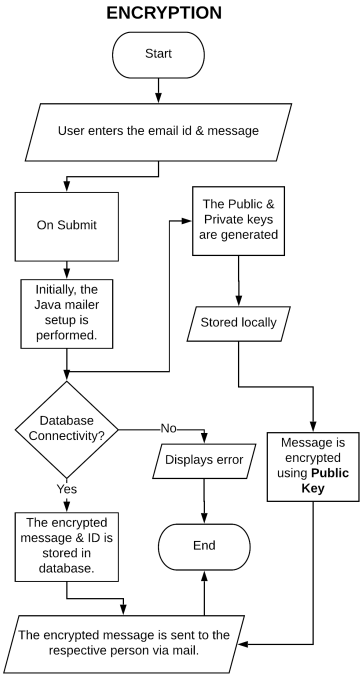


Fig. 2: Email Encryption Flow.

The encryption process of the application is shown in Fig. 2. It is performed using the public key. Whereas, the decryption process is performed using the private key of the same user, which owns the public key used in the encryption phase. The decryption process of the application is shown in Fig. 3.

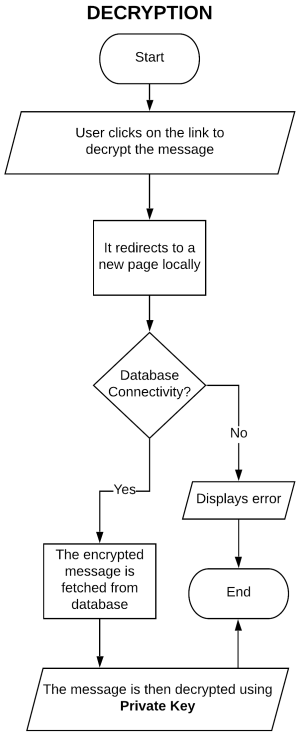


Fig. 3: Email Decryption Flow.

When the submit button is clicked, The EncryptMessage.java Servlet is called which performs actions based on the request. Initially, the Servlet setup the mailer, which is javax.mailer.jar, by configuring the Gmail properties. Then, the first important step is the generation of keys. The keys are created using the GenerateKeys.java class file. Later, the message is encrypted using the RSA algorithm by calling the methods of PerformEncryptDecrypt.java java class file. The message is encrypted using the public key. The mail consisting the encrypted message is sent via javax.mailer library and messages are displayed as shown in Fig. 4. The preview of the mail is shown in Fig. 5.

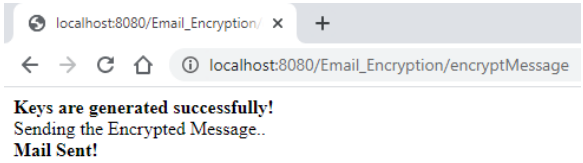


Fig. 4: Encrypted mail sent.

Fig. 5: Preview of encrypted mail.

The mail also includes a link which can be used to decrypt the message. The link is redirected to a localhost (which can also be redirected to a domain, if available) along with the message ID. Then, the respective encrypted message is fetched from the database and decrypted by calling the methods of PerformEncryptDecrypt.java file with the help of private key. Hence, the plaintext is retrieved back and displayed to the intended user on the browser as shown in Fig. 6.

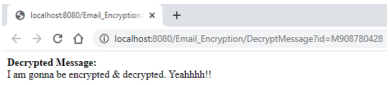


Fig. 6: Decrypted message of the mail.

The library javax.mail.jar is used to send for sending mail [1]. It requires the properties to be set such as SMTP host, TCP Port, authentication, password, etc. The code snippet is shown in Appendix A. The Public and private keys are generated by GenerateKeys.java as shown in Appendix B. Once the keys are generated, the encryption and decryption operations are defined in PerformEncryptDecrypt.java, shown in Appendix C. Then, this operation are called in EncryptMessage.java servlet and DecryptMessage.java servlet to encrypt and decrypt the message respectively. The code snippet is shown in Appendix D.

III. Chat Application Implementation

The messaging application allows users to register and login into their account through username and password. On login, it displays the list of all users that are registered to the

application. It allows the user to chat with any user just by clicking on the name. The messages stored are completely in encrypted form, which are encrypted using the receiver's public key. So that, only the receiver's private key can decrypt it. These keys are generated during the registration process. The public key is stored on the server & the private key is securely stored on the local storage of user's device and keys are unique for each users. The encryption and decryption process of the application is shown in Fig. 7 and 8 respectively.

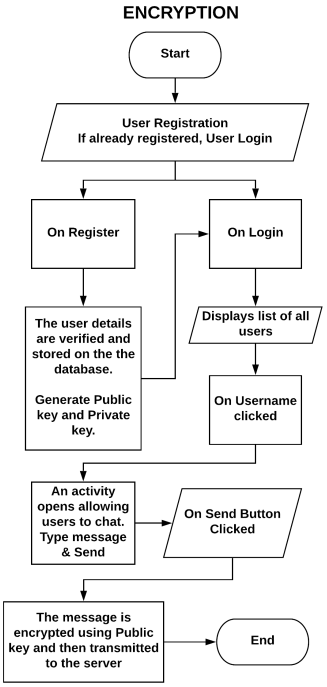


Fig. 7: Messaging Encryption Flow.

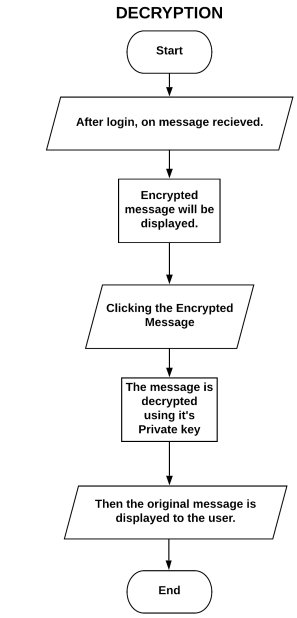
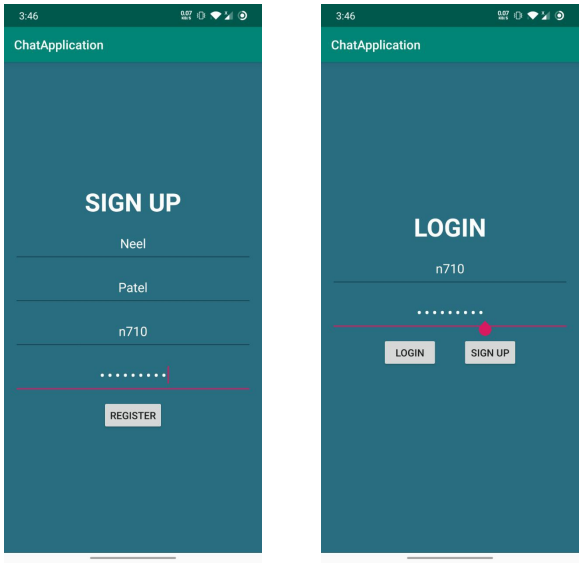


Fig. 8: Messaging Decryption Flow.

Firstly, the user needs to register himself, on clicking "Register" button, an onClick event listener of the button will be executed which register the user and generated public and private keys. The private key is stored as a file having named "{username}_private" with no file extension. This isn't the actual private key. It is the encoded version of private key using Base64 encoder. Whereas, the public key is encoded with Base64 encoder and stored on the server, which is accessible by any user. The keys are generated using KeyPairGenerator and the algorithm used is RSA. Once the user is registered, the user can logged in using the username and password as shown in Fig.9.



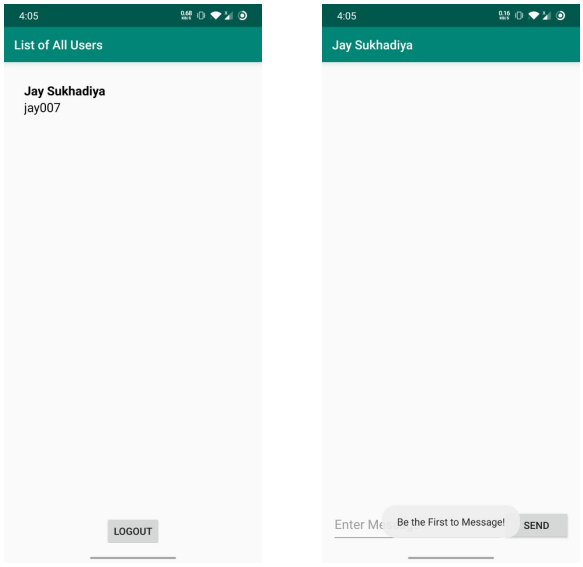
(a) User Registration. (b) User Login.

Fig. 9: Login and Registration Activity.

A list of all the users is displayed and on clicking the names, two users can start chatting with end-to-end encryption scenario. When a sender type the message and click on the "Send" button as shown in Fig. 10. The message is fetch from the edit text of application. The public key of the receiver is fetched from the server and decoded using the Base64 decoder. Once, the actual public key is obtained, the message is encrypted and saved on the server. The RecyclerView refresh itself by notifyDataSetChanged() and the encrypted data is displayed on the chat window. From receiver point of view, the user sees the encrypted message and when the user clicks on the message, the onClick event listener of the view is called and the message is decrypted using the private key of the receiver, which is stored locally. This private key is fetched from the local storage and decoded using Base64 decoder.Then after obtaining the actual private key, the message is decoded. The process is shown in Fig. 11.

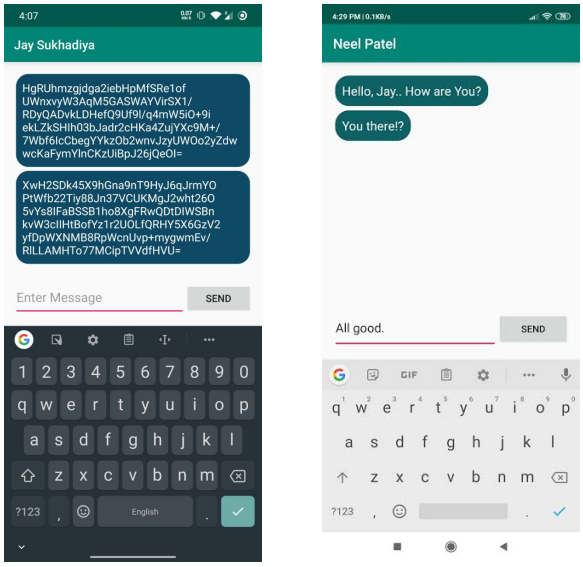
The implementation of login and registration activity is similar as the most of the application possess. The registration activity includes the generation of keys, where the private

key is stored on local storage and public key on the server as shown in Appendix E. The ChatActivity.java involves the encryption and decryption of messages. For sender's view, the message is encrypted using receiver's public key and then stored on the server. For receiver's view, an encrypted message is fetched and displayed to the user. Each message has an onClick listener which performs decryption of the message using the receiver's private key. The code snippet is shown in Fig. F.



(a) Users list. (b) Messaging window.

Fig. 10: List of Users and Chat Window.



(a) Sender's view. (b) Receiver's view.

Fig. 11: Encryption and Decryption.

IV. Analysis

Public key encryption says that if a message is encrypted using the public key of receiver, then it can be only decrypted using private key of the same receiver. However, in this case, the private key is stored locally, as specified by the most messaging application. Then how once a message is encrypted using public key is decrypted just to display in the chat window. As, none has the access to the private accepts the owner of the key. This leads to two scenarios in terms of any application that provides end-to-end encryption:

1. The Private key is also stored on the server. Example: Gmail, it stores to generate smart replies.
2. The unencrypted message is stored on the local device first then it is encrypted and stored on the server.

In the case 2, if the device is comprised then the messages will be in readable format as it is in unencrypted form. Also, a background task can also be created by the an organization in Android to fetch the unencrypted message from both users and sort them according to their timestamps.

V. Conclusion

Encryption is all about keeping the user's content privacy restricted to oneself. Implementation of web application as well as an android application provides a brief idea about the real-life encryption scenario. Storing the public key on the server and private key locally, preserving the rule of not providing access to anyone. Hence, the unencrypted message is needed to be stored locally before encrypting the message. Hence, once the device is compromised or any organization can collect the unencrypted texts of users and sorted them by respective timestamps the user's privacy will be hindered or the private key is needed to be stored on the server, which provides organization an advantage to access the data whenever required.

References

[1] https://javaee.github.io/javamail/#Download_JavaMail_Release

[2] Zhao Jingling, Zhang Huiyun, Cui Baojiang, "Sentence Similarity Based on Semantic Vector Model", Ninth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, IEEE, 2014

[3] Xinchen Xu, Feiyue Ye, "Sentences Similarity Analysis Based on Word Embedding and Syntax Analysis", 17th IEEE International Conference on Communication Technology, IEEE, 2017.

[4] Fu Cheng, An Bo, Han Xianpei & Sun Le, "ISCAS_NLP at SemEval-2016 Task 1: Sentence Similarity Based on Support Vector Regression using Multiple Features.", Association for Computational Linguistics, 2016.

[5] Atish Pawar, Vijay Mago, "Challenging the Boundaries of Unsupervised Learning for Semantic Similarity", IEEE Access, IEEE, 2019.

Appendix

A. Setting up the mailer

```
1  //Setting up Java Mailer.
2  String fromID = "npatel46@lakeheadu.ca";
3  String toMail = request.getParameter("
       ↪ receiptEmail");
4  String bodyMessage = request.getParameter("
       ↪ message");
5  String password = (new SecurePassword()).
       ↪ getPassword();
6
7  //Adding GMAIL properties.
8  Properties props = System.getProperties();
```

```
props.put("mail.smtp.host", "smtp.gmail.com"); //SMTP Host
props.put("mail.smtp.port", "587"); //TLS Port
props.put("mail.smtp.auth", "true"); //enable authentication
props.put("mail.smtp.starttls.enable", "true"); //enable STARTTLS

//Authenticating the connection along with username & password.
//Also, setting up the session.
Session session = Session.getInstance(props, new javax.mail.Authenticator() {
@Override
  protected PasswordAuthentication getPasswordAuthentication() {
    return new PasswordAuthentication(fromID, password);
  }
});

//Setting up the Mail. (From, To, Body).
MimeMessage msg = new MimeMessage(session);
msg.setFrom(new InternetAddress(fromID, "no-reply"));
msg.setRecipients(Message.RecipientType.TO, InternetAddress.parse(toMail, false));
msg.setSubject("Secret Message");
msg.setContent(encryptedMessage+"<br><a href='http://localhost:8080/"
+ "Email_Encryption/DecryptMessage?id="+msgIDGen+"'>Decrypt it.</a>","text/html");

//Sending the Mail.
Transport.send(msg);
```

B. Generating keys

```
public GenerateKeys(int length) throws Exception{
  this.keyPairGen = KeyPairGenerator.getInstance("RSA");
  this.keyPairGen.initialize(length);
}
public void createKeys(){
  this.keyPair = keyPairGen.generateKeyPair();
  this.privateKey = keyPair.getPrivate();
  this.publicKey = keyPair.getPublic();
}
public void writeToFile() throws Exception{

  //Writing Public Key to a File
  File f = new File(Path+"publickey");
  FileOutputStream fos = new FileOutputStream(f);
  fos.write(this.publicKey.getEncoded());
  fos.flush();fos.close();
  //Writing Private Key to a File
  f = new File(Path+"privatekey");
  fos = new FileOutputStream(f);
  fos.write(this.privateKey.getEncoded());
  fos.flush();fos.close();
}
```

C. Operations of encryption and decryption

```
public GenerateKeys(int length) throws Exception{

  public PrivateKey getPrivate(String filename) throws Exception {
    byte[] keyBytes = Files.readAllBytes(new File(filename).toPath());
    PKCS8EncodedKeySpec spec = new PKCS8EncodedKeySpec(keyBytes);
    KeyFactory kf = KeyFactory.getInstance("RSA");
    return kf.generatePrivate(spec);
  }

  public PublicKey getPublic(String filename) throws Exception {
    byte[] keyBytes = Files.readAllBytes(new File(filename).toPath());
    X509EncodedKeySpec spec = new X509EncodedKeySpec(keyBytes);
    KeyFactory kf = KeyFactory.getInstance("RSA");
    return kf.generatePublic(spec);
  }

  public String encryptText(String msg, PublicKey key) throws Exception {
    this.cipher.init(Cipher.ENCRYPT_MODE, key);
    return Base64.getEncoder().encodeToString(cipher.doFinal(msg.getBytes("UTF-8")));
  }

  public String decryptText(String msg, PrivateKey key) throws Exception {
    this.cipher.init(Cipher.DECRYPT_MODE, key);
    return new String(cipher.doFinal(Base64.getDecoder().decode(msg)), "UTF-8");
  }
}
```

D. Encryption and decryption of the message

```
//Encrypting the message
PerformEncryptDecrypt edobj = new PerformEncryptDecrypt();
PublicKey publicKey = edobj.getPublic(gkobj.Path+"publickey");
String encryptedMessage = edobj.encryptText(bodyMessage, publicKey);
out.write("<br>Sending the Encrypted Message ..<br>");

//Decrypting the message
PerformEncryptDecrypt edobj = new PerformEncryptDecrypt();
PrivateKey privateKey = edobj.getPrivate((new GenerateKeys()).Path+"privatekey");
String decryptedMessage = edobj.decryptText(encryptedMessage, privateKey);
```

E. Generation and storing of keys during registration

```
myRef = FirebaseDatabase.getInstance().getReference().child("Users");
try {
  keyPairGenerator = KeyPairGenerator.getInstance("RSA");
  keyPairGenerator.initialize(1024);
  keyPair = keyPairGenerator.generateKeyPair();
  publicKeyString = Base64.getEncoder().encodeToString(keyPair.getPublic().getEncoded());
} catch (Exception e) {
Toast.makeText(RegistrationActivity.this, e.toString(), Toast.LENGTH_SHORT).show();
}

//Storing Private Key to Local Storage
File path = RegistrationActivity.this.getFilesDir();
File file = new File(path, userName + "_private");
try {
  FileOutputStream stream = new FileOutputStream(file);
  stream.write(keyPair.getPrivate().getEncoded());
  stream.close();
} catch (Exception e) {
Toast.makeText(RegistrationActivity.this, "File Error: " + e.toString(), Toast.LENGTH_SHORT).show();
}

//Simple Encoding the Password
String encodedPassword = Base64.getEncoder().encodeToString(password.getBytes());
newUser.setPassword(encodedPassword);
newUser.setPublicKey(publicKeyString);
myRef.child("M" + userID).setValue(newUser);
```

F. Encrypting and Decrypting of message in Android application

```
//Encrypting Message
try {
  byte[] keyBytes = Base64.getDecoder().decode(receiverDetails[0].getPublicKey());
  X509EncodedKeySpec spec = new X509EncodedKeySpec(keyBytes);
  KeyFactory kf = KeyFactory.getInstance("RSA");
  Cipher cipher = Cipher.getInstance("RSA");
  cipher.init(Cipher.ENCRYPT_MODE, kf.generatePublic(spec));
  userMessage = Base64.getEncoder().encodeToString(cipher.doFinal(userMessage.getBytes("UTF-8")));
} catch (Exception e) {
Toast.makeText(ChatActivity.this, e.toString(), Toast.LENGTH_SHORT).show();
}

//Storing encrypted Message on Server
Messages MessagesObject = new Messages(user1_id, userMessage);
myRef.push().setValue(MessagesObject);

//Decrypting Message
eachItem.setOnClickListener(new View.OnClickListener() {
  @RequiresApi(api = Build.VERSION_CODES.O)
  @Override
  public void onClick(View view) {
    try {
      String decryptMessage = getShowMessage();
      byte[] keyBytes = new byte[1024];
      File path = ChatActivity.this.getFilesDir();
      File file = new File(path, current.getUserName() + "_private");
      try {
        FileInputStream in = new FileInputStream(file);
        in.read(keyBytes);
        in.close();
      } catch (Exception e) {
        Toast.makeText(ChatActivity.this, "File Error: " + e.toString(), Toast.LENGTH_SHORT).show();
      }
      PKCS8EncodedKeySpec spec = new PKCS8EncodedKeySpec(keyBytes);
      KeyFactory kf = KeyFactory.getInstance("RSA");
      Cipher cipher = Cipher.getInstance("RSA");
      cipher.init(Cipher.DECRYPT_MODE, kf.generatePrivate(spec));
      decryptMessage = new String(cipher.doFinal(Base64.getDecoder().decode(decryptMessage)), "UTF-8");
      setShowMessage(decryptMessage);
    } catch (Exception e) {
    Toast.makeText(ChatActivity.this, e.toString(), Toast.LENGTH_SHORT).show();
    }
  }
```