NudgeMe: E2E Encrypted Data Passing Between Clients

Chris Tomy
chris.tomy.19@ucl.ac.uk
University College London
London, United Kingdom

ABSTRACT

This document presents the method used by NudgeMe[1] to securely send data from one mobile client to another, using RSA encryption and a server. We explain and justify the libraries and technologies used, from (mostly) a software engineer's perspective. We also comment on some limitations to the current method and propose potential solutions for future implementations.

KEYWORDS

asymmetric, encryption, rsa, flutter, mobile, development

1 INTRODUCTION

For our Systems Engineering module (*COMP0016*), we were tasked with building a mobile application to solve the problem of the general public's low well-being, which is likely a result of a lack of movement, exercise or some other metric. Being able to track and share this with trusted people may help in some way. However, this kind of data is likely sensitive, so we decided to incorporate some mechanism to improve the privacy of the user. We ended up using asymmetric encryption.

The mobile application is built in Flutter, and the back-end server in Go. It is common for modern mobile apps to use Google Firebase as a back-end, but we wished to avoid relying on a cloud service like Firebase. (Also, we would have needed to further investigate the privacy of data passing through Firebase.)

2 RELATED WORKS & APPROACHES

A well-known approach to protecting personal privacy in communications is by using PGP encryption. This has been criticised due to it's lack of perfect forward secrecy[2]. A better approach would be to use Extended Triple Diffie-Hellman (X3DH) to set up a shared secret key, which can then be used to securely communicate further. This provides forward secrecy and cryptographic deniability[9].

Our final approach is actually closer to PGP, except uses RSA encryption. Using RSA limits us in even further ways than PGP, as will be discussed, but has the vast benefit of being simple to implement. If needed, one may be able to roll their own RSA encryption. This is a consideration since we are using Flutter (and the underlying language: Dart) which does not have as wide a selection of libraries as C++ or Java, for example.

3 REQUIREMENTS & ASSUMPTIONS

- Our goal is to transfer some string, the plaintext, from one mobile device to another.
- This plaintext is **not arbitrary**. (We are not designing an E2E encrypted **messaging** app.)
- The length of the plaintext will be less than or equal to 256 bytes.
- We can use a single server to facilitate this.
- The language the mobile client will use is Dart.
- The language the server will use is Go.

The limitations imposed for the programming languages are due to us already implementing the rest of the project in these language.

4 ADDRESSING THE LIMITATIONS OF RSA

Now that we have defined our assumptions/requirements, we can properly address some of the major limitations of using the RSA cryptosystem:

- *PKCS #1 v1.5 is vulnerable against Bleichenbacher's attack[11]* — the crypto library which we ended up using uses a newer definition of RSA which patches this problem.
- *An attack exists against the newer signature scheme, RSASSA-PKCS1-v1_5[11]* — this attack allows an adversary to forge signatures, essentially claiming to be the true sender of a message[6]. Although the crypto library we used *is* vulnerable to this attack, we do not use the signature scheme. Repudiation is not a concern.
- *Encryption with a 2048-bit RSA public key will fail with messages longer than 256 bytes[10]* — as mentioned in the requirements, we can define the format of the data ourselves, so we have imposed the restriction of using lengths of less than or equal to 256 characters.
- *Lack of forward secrecy[10]* — this problem does exist in our implementation. If we were a malicious server, we could hold onto ciphertexts, eventually find a way to access the private key of any user and then decrypt any messages (ciphertexts) sent to them in the past. Therefore, the feasibility of exploiting this is equivalent of the feasibility of an attacker gaining access to a mobile device's filesystem[2]. In the future, if the data transferred by our system proves to be even more sensitive, then this is one area that may need improvement.

5 IMPLEMENTATION

See our GitHub repositories[3] for the full code and documentation.

5.1 Overall Approach

Here we summarise the system. There are essentially two parts to our system: the message/data passing performed by the server and the encryption/decryption which is performed client-side.

[1] http://students.cs.ucl.ac.uk/2020/group26/index.html

[2] Since, if you had access to the filesystem, you could read the private key stored on device.

[3] https://github.com/UCLComputerScience/COMP0016_2020_21_Team26 and https://github.com/thevirtuoso1973/team26-nudgeme-backend

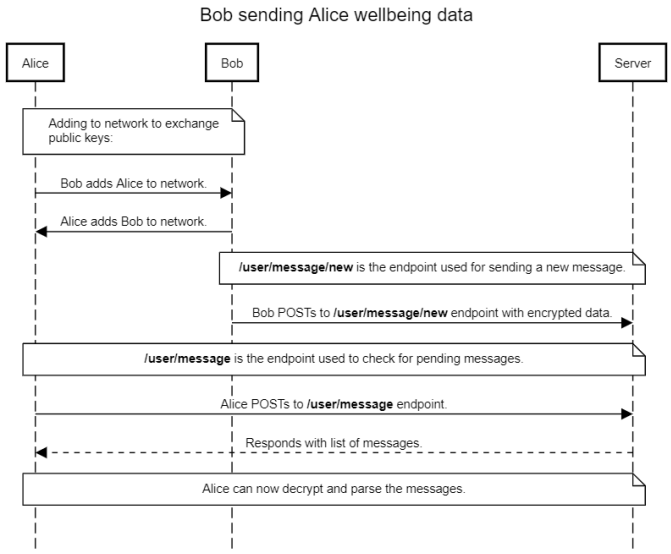

Figure 1: Sequence diagram of a user sending another user data.

To pass data from one client to another, the sender can POST data to the server where it will be stored as a pending message to a specified user. All clients will periodically POST to the server to retrieve any pending messages[4]. An important part of this is that clients/users are identified by a string, and authenticated by a password. This is described later.

To perform the client-side operations between a pair of users, they need to have exchanged public keys. This is what happens under the hood when two users add each other. Then they can encrypt/decrypt data as needed using a crypto library.

See Figure 1 for a sequence diagram of the interaction between the server and another user.

5.2 Cryptography Libraries

To authenticate clients, we use the bcrypt package from the golang cryptography libraries[5]. A simpler approach may be to use a traditional hash function like SHA-256, but we have no need for deterministic outputs, so we prefer the stronger protections bcrypt provides[12].

On the client's side, we use two packages *encrypt* and *pointycastle*, to perform encryption/decryption and key generation respectively. *PointyCastle* implements PKCS #1 version 2.0 of RSA encryption[1].

5.3 User Initialisation

Client devices need to perform two tasks to set up, generate their RSA key-pair and register themselves with the server.

We use the *pointycastle* library to generate the key-pair. This gives us key objects *RSAPublicKey* and *RSAPrivateKey*. However, we need to store this persistently so we manually encode the keys into a PEM format string. To store the keys in PEM, the middle section consists of the base64 of the ASN.1 sequence of components, defined in the appendix of RFC 8017[7]. We use the PKCS #1 definition of RSA, since this is the one recognised by the parser in our chosen *encrypt* library[8].

The identifier for the client is the fingerprint, derived from the public key. This reduces the association between the actual user. A random string may also suffice, but we chose to use the public key fingerprint since it offers the convenient side effect of allowing our server to check that no two users have the same public key (although collisions are quite rare[3]).

The fingerprint is generated from the modulus and exponent of the public key.

```
String getFingerprint(RSAPublicKey key) => SHA1Digest()
    .process(getPublicKeyBytes(key))
    .map((e) => e.toRadixString(16)) // convert to hex
    .join();

Uint8List getPublicKeyBytes(RSAPublicKey key) {
  final asn = ASN1Sequence();
  asn.add(ASN1Integer(key.modulus));
  asn.add(ASN1Integer(key.exponent));
  return asn.encode();
}
```

The client also generates their own random alphanumeric password. The client then POSTs their identifier and password to the (HTTPS) server, registering them. The server will store the identifier and the hash of the password (using bcrypt).

The private, public key and password are stored using *shared_preferences*, which is a flutter package that provides a key-value store. Notably, other installed applications cannot access data stored in this way[4].

5.4 Key Exchange (Adding Users)

This is mostly implementation specific. At the simplest level, the process would consist of copying the string of the public key into a text input box and adding that string onto a local database. This does not need to be a private data store, since these are public keys.

Our solution offers further convenience, by allowing users to scan a QR code, which is essentially a concatenation of the identifier and public key PEM string. Alternatively, for remote users we also support sending a deeplink which prefills the appropriate fields.

However, we prefer the scanning QR code method to be used since it does not rely on an external server, providing better privacy guarantees. For example, if our server was malicious, they could serve a public key that *the server* owns, instead of the one owned by the user (say, User 2) that the initial user (say, User 1) wanted to add. So the server would be able to decrypt any data sent through it which is intended for User 2.

5.5 Sending Data

The server hosts an endpoint, */user/message/new*, which expects the sender's identifier, receiver's identifier, sender's password and the data itself. The data is JSON, and since a string is valid JSON, can be just considered as a string.

The server will check if the given password matches the one stored alongside the sender's identifier:

[4]This might seem more appropriate as a GET request, however, the server will delete the data as soon as it is passed along. So it is modifying state.
[5]https://github.com/golang/crypto

```
err := bcrypt.CompareHashAndPassword(stored,
                        []byte(password))
```

Note that a client can, according to the API, send a message to any other client. This is fine since the receiving client can just discard these messages.

5.6 Receiving Data

The client will periodically execute some code to retrieve pending messages. This uses the *workmanager*[6] package.

```
Workmanager.initialize(callbackDispatcher,
    isInDebugMode: kDebugMode);
Workmanager.registerPeriodicTask("refresh_friend_task",
    REFRESH_FRIEND_KEY,
    frequency: Duration(minutes: 15),
    initialDelay: Duration(seconds: 10));
```

This provides the app with an execution environment outside the main Flutter environment, which checks for new data sent from any friends. Workmanager causes a delay of around 15 minutes to receive data, which is acceptable since we did not aim to develop an *instant messaging* app. (This is where a service like Google's Firebase may be useful, as it allows you to instantly send messages/notifications[5].)

The actual code that executes is one that POSTs to the */user/message* endpoint, with the identifier and password. Note that the authentication here is important to only allow the corresponding identifier to get their messages. (Although, since the data *is* still encrypted, one could discard the entire password authentication if encryption is always used client side and there is no risk of misbehaviour from users.)

After receiving the response, the client iterates through any messages and uses the *encrypt* package to perform decryption[7].

6 CONCLUSION

We have presented a method of secure data passing between mobile clients running a Flutter application. This consists of mobile clients registering themselves with a server, then sending and checking for pending messages. We have achieved this without the use of a cloud service like Firebase.

One limitation which may need to be addressed is the constantly increasing list of registered users. Due to storage concerns, there may need to be a way for the server to remove users safely, such that only inactive users are removed. One solution could be for mobile clients to send a request to the server every week, just to notify the server that they are still an active user. So periodically, the server would delete any user who hasn't notified the server of their existence in a week. The pending data or message list is less of a storage concern since data is deleted once passed along to the intended receiver.

Another enhancement would be for users to transfer their public/private keys and network to another device. Currently, if the client is installed on a new device the initialisation procedure runs again, creating a new pair of keys and requiring the user to add all their friends to their network again.

However, the most useful addition to the overall NudgeMe system may be to leverage this secure data passing further by supporting more metrics to track. With the privacy of end-to-end encryption, users would be far more willing to track and send statistics of phone usage, screen time, messages per day, etc. We look forward to seeing future iterations of NudgeMe.

ACKNOWLEDGMENTS

Thanks to the following people:

- Saachi Pahwa — for assisting the development of NudgeMe.
- Dr Joseph Connor, CarefulAI — our client for proposing the original idea, and providing feedback.

REFERENCES

[1] bcgit. [n.d.]. RSA. https://github.com/bcgit/pc-dart/blob/master/tutorials/rsa.md. Accessed: 2021-03-21.

[2] Nikita Borisov, Ian Goldberg, and Eric Brewer. 2004. Off-the-Record Communication, or, Why Not to Use PGP. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society* (Washington DC, USA) *(WPES '04)*. Association for Computing Machinery, New York, NY, USA, 77–84. https://doi.org/10.1145/1029179.1029200

[3] CodesInChaos. [n.d.]. How many RSA keys before a collision? https://crypto.stackexchange.com/a/2559. Accessed: 2021-03-20.

[4] Android Documentation. [n.d.]. Data and file storage overview. https://developer.android.com/training/data-storage. Accessed: 2021-03-20.

[5] Google. [n.d.]. Firebase Cloud Messaging. https://firebase.google.com/products/cloud-messaging. [Online; accessed 21-March-2021].

[6] T. Izu, M. Takenaka, and T. Shimoyama. 2007. Analysis on Bleichenbacher's Forgery Attack. In *The Second International Conference on Availability, Reliability and Security (ARES'07)*. 1167–1174. https://doi.org/10.1109/ARES.2007.38

[7] Ed. et al K. Moriarty. 2016. *PKCS #1: RSA Cryptography Specifications Version 2.2.* RFC 8017. IETF. https://tools.ietf.org/html/rfc8017

[8] leocavalcante. [n.d.]. Encrypt - rsa.dart. https://github.com/leocavalcante/encrypt/blob/4.x/lib/src/algorithms/rsa.dart. Accessed: 2021-03-20.

[9] Moxie Marlinspike. [n.d.]. The X3DH Key Agreement Protocol. https://signal.org/docs/specifications/x3dh/. Accessed: 2021-03-20.

[10] Soatok. [n.d.]. Please Stop Encrypting with RSA Directly. https://soatok.blog/2021/01/20/please-stop-encrypting-with-rsa-directly/. Accessed: 2021-03-20.

[11] Wikipedia contributors. 2020. PKCS 1 — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=PKCS_1&oldid=989288999. [Online; accessed 20-March-2021].

[12] Wikipedia contributors. 2021. Bcrypt — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Bcrypt&oldid=1012852295. [Online; accessed 21-March-2021].

[6] https://pub.dev/packages/workmanager

[7] Yes, *encrypt* also performs decryption.