Nervous Fish Final Report

Course: Context project - TI2806
Group: NervousFish
Date: 22 June 2017
Developers:

- 4501160 - Kilian Callebaut
- 4447859 - Eric Cornelissen
- 4457668 - Stas Mironov
- 4475208 - Joost Verbraeken
- 4488628 - Cornel de Vroomen

Abstract

In this document we will discuss the Android application we developed during the Context Project (TI2806) at the TU Delft, named NervousFish. The document goes into the functionality of the application and the process of developing the app. Besides that a brief discussion of user tests is given, and the document ends with a discussion on what the future of the app could be.
NervoushFish, the app that has been developed, is an app which can be used to securely exchange public keys (meant to be used for public/private key cryptography by other applications) via either Bluetooth, NFC and QR. A special focus has been on Bluetooth because it is vulnerable to Man in the Middle (MITM) attacks. We developed special methods to prevent these attacks from spying /interfering with the exchange, so users can be sure they received the correct key.
Besides that, the application offers some quality of life feature like a contacts list to manage the keys one received or the ability to see your own public/private key pairs.

Table of contents

Introduction

Problem description

Electronic communication can be insecure. But it can be done secure by using public-private key cryptography. However, for this to work people need each other's public key. Just sending this via an email is not secure, as there is a possibility that this data is changed by a Man-In-The-Middle. Up until now, these have been exchanged in key signing parties[1], but this is a time consuming and error prone way to exchange keys. To solve this problem, we created a mobile application that can be used to exchange public keys securely and easily via Bluetooth, NFC and QR codes.
However, communication over Bluetooth is not very secure, because there are several way to execute a Man-In-The-Middle (MITM for short) attack on a Bluetooth connection (Mutchukota, T. R., Panigrahy, S. K., & Jena, S. K, 2011). Our app provides an easy and intuitive way to secure the exchange of data over Bluetooth, immune to MITM attacks, using visual or rhythmic patterns. This process uses symmetric key encryption in the background.
So, the problem can be summarized as: “How can we securely exchange public keys via Bluetooth?”. Which has been solved through a variety of ways, either by enhancing the security of Bluetooth or by provided other means to exchange public keys.

End-user’s requirements

I want to have an app that:

- Can generate private/public-keypairs
- Stores my own private/public-keypairs
- Stores the public keys of other person, identified by their name and have the ability to manage them
- Can distribute my own public key to other persons using some MITM-proof signature method (preferably over bluetooth)
- Stores all data encrypted
- The app is primarily designed for android
- Can show me my own public key and also the ones of all my contacts and have the possibility to copy it

According to these requirements we can deduce that, following the first 3 points, our app should have some kind of storage mechanism which is tweaked to mostly support key pairs and maybe some extra data on top of that while also preferably being able to generate those key pairs itself. Keeping in mind that this app should be specifically designed for the android platform, we knew from the start that we would have to deal with a specific filesystem and had some constraints

[1] https://en.wikipedia.org/wiki/Key_signing_party

regarding the processing power. Luckily, there exist a lot of well optimized libraries that can do the key pair generation, in our case libraries that support RSA and ED25119[2] keys, and are easy to implement. The challenge would be to create a pojo-based structure which can encapsulate all the necessary data.
This leads us to the next, fourth requirement, where we need to create ways to transfer the previously generated/created data objects over some mediums which would be MITM proof. Some methods like NFC and QR codes immediately come to mind as they are innately MITM proof. A more widespread and well known medium would be Bluetooth but in order to make it MITM proof we would need to create an extra layer on top of it which would make the exchange not only more user friendly but also prevent MITM attacks during the exchange. The fifth requirement also requires this mechanism to be fully encrypted which is logical as the very first requirement is a lot less safe if the key pairs are saved in plain text or something like base64 encoding on a publicly accessible part of the android filesystem.
The last requirement in the list above is more an embodiment of certain features the app should have rather than a very specific task. After we've generated key pairs, created profiles and exchanged contacts, it would be nice to make it all visible and manageable for the user of our app. Thus we would need to establish a real time link with our database which would be able to encrypt and decrypt the data objects and add some sugar to it and present it in a pretty and concrete manner in order to assist the user but not detract from their user experience.

Product overview

Our product is an android application named *NervousFish*. It is a small app that requires permissions for Bluetooth and NFC, but doesn't rely on an internet connection.
First off, users can create a profile by entering their name, a password and selecting the type of key they want to use. This data is stored securely in an encrypted database (see "Developed functionalities")
Once the user has created a profile they can login. In order to login they have to enter their password using a built-in custom keyboard. Any custom keyboard, like SwiftKey, can record what is being typed, and are therefore not safe enough for passwords. After logging in, the user reaches the main screen. In the main activity they can view their contact list, add new contacts by exchanging public keys, sort these contacts on name or key type and access the settings menu, which we will explain in detail below.
A new contact is added when a public key is exchanged. Exchanging can be done over Bluetooth, NFC or via QR codes.

- When the user chooses to use Bluetooth, they can send a request to exchange public-keys to a nearby Bluetooth devices and choose how they wants to prove their identity. Currently, this can be done in two ways:
    - The two users both tap the same rhythm. A K-means clustering algorithm is used analyze the tabs and generates a code. This code must be the same on both devices in order for the exchange to be successful.

[2] https://ed25519.cr.yp.to/

    - The two users see an image and have to tap a number of elements that they see on the image. Based on the selected elements, a code is generated. Both users have to tap on the same elements in the same order in order for the exchange to be successful.

  The public key and other date of the user is encrypted using the obtained code, which serves as a symmetric key. The encrypted data is then sent (by both users) over Bluetooth to the other user. When data is received it can only be decrypted if they generated the same code, otherwise the decrypted message will be meaningless garbage.
- When the users chooses to use NFC to exchange their public key, they'll see an animation that shows how this method works. As is the case with NFC, users have to let their devices touch. When the devices are ready to beam (i.e. exchange keys), the activity changes and request the user to "Tap to beam". Then, if they tap, their public key is sent to the other user.
- When the user chooses to use QR codes, the user will be shown a QR code that contains their name and public key. This QR code can be scanned by another user. Below the QR code is a button that opens a different app to scan the QR code. Once the code is scanned, the contact described in the QR code is added as a contact.

When the public key and name of the other person is received, the app adds the new contacts to the user's contacts. When the contact already exists, the app prompts you what you want to do (i.e. rename / add public key to other contact / cancel)

Besides the functionality that is about the public key exchange between users, we also have some quality of life features built in the app. For example, in the main screen you will see a list with all your contacts. This, upon further inspection by clicking on them, will reveal a more detailed screen that includes the name and the keys assigned to the contact. The name can be changed and the contact can be deleted within two clicks. We do have a lot of notifications to ensure the action selected was intended by the user and have a way out in case of a misclick. If the users wishes to inspect his own keys and bring a little structure in his collections, similar possibilities as above are implemented in the key management screen where you can rename remove and generate key pairs linked to your profile.

Our app also comes with a few configurable settings which all are accessed by viewing the general or the profile screen. For now you can predefine the default top level bluetooth encryption mechanisms explained above and in the profile screen you can change your name and password use in the app.

Reflection (software engineering perspective)

Product reflection

Robustness

In the first weeks we didn't validate the user input from the application at all, what resulted in unexpected behaviour. The parameters passed to functions were not validated, leading to a neglect of the *"stupidity"* principle. To improve on this we decided to validate all user input and give a suitable error message when the input does not meet the requirements. We also validate all non-private methods using Apache Commons Validate[3] methods. Another thing we did to make the program more robust was guaranteeing that most values won't change by making them final. All these decisions show that we put some thought into the *"Can't happen"* principle . Our architecture design, from the very beginning, relied on a lot of interfaces and abstraction which made our code a lot more robust regarding the *"Dangerous implements"* principle. It also improves maintainability and extendability.
What we still want to do is to handle exceptions in a better way. For example, when there is an issue with the database an exception is thrown, logged, its stack trace printed, and then the app crashes. Instead we want to give the user an error message and recover from the error (in this case by creating a new database). The focus on improving the exceptions should improve our score on the *"Paranoia"* principle.

Correctness

To verify that the program works as intended, we wrote JUnit tests and Instrumentation tests. Because everything that can be private is private (so also the constructors of all modules) we had to use reflection to get access to these methods. This is cumbersome and ugly, but in Java there is no alternative to get access to private methods. This was thoroughly discussed with the software engineering TA to ensure it was deemed necessary. Instrumentation tests are run on an Android device or emulator. They work great as part of integration and requirements tests of the app, but cost a lot of time to get working. Keeping in mind that they require quite a lot of computing power we'll be more selective in future regarding the example scenarios we include, so no 5 valid usernames, but just a single one (unless we want to test corner cases).
Later we also added asserts to verify that every single parameter of each private function conformed to what we expected from the parameter to reduce the chance that the methods operates on incorrect values by accident. This worked very well so we'll try to take it with us for future projects.

[3] https://commons.apache.org/proper/commons-validator/

Maintainability

In the Architecture Design Document you can find a lot about our attempts to make NervousFish as maintainable as possible. The service locator took us quite some time to get right, and even then it still is not great because of a limitation of Java (the inability to construct an interface on runtime). Even worse, because passing objects between activities is extremely inefficient (requires serialization / deserialization every time) we had to use the built-in Application singleton from Android. That is also something we cannot improve unfortunately, as it is a Android project specific barrier.
Coding all modules against an interface was something new for some of use, but it was something all of used liked. This, in combination with eager use of abstractions, make for very extendable code. There were a few code style issues which made us improve as developers but also some that we felt were not very useful. We made every variable that should not change final what was very ugly, but also made a program a lot more robust. Next time we prefer a language like Kotlin in which final is the default. We also did not use any modifier when a function was package private. We also were in doubt about the use of `this.`. It's ugly and doesn't really add anything except for more clarity. Next time we'll consider omitting `this.` and instead explicitly indicating variable when it is static or from an outer class.

Portability

NervousFish separates the front-end from the back-end completely. The front-end can access all modules via the so-called service locator. Every service locator chooses which modules it wants to use. For example, for the Android platform we defined a service locator that uses Android-compatible modules. By changing a single reference a different service locator can be used for the whole application what makes it pretty easy to port the app to other platforms. Next time we will probably use a service locator again, but then we want to combine it with dependency injection. Unfortunately this was very inefficient to do in Android, because Android prohibits direct dependency injection between activities, but in other frameworks that should not be a problem. We also often used adapter between external libraries and the app, for example a FileSystemAdapter that handled all filesystem related code. This system worked quite well and next time we'll abstract even more things away like a class responsible for threads, so that there is less duplicate code and it is easier to switch from implementation later on.

Extensibility

All modules whose implementation may change in the future implement an interface. There are also a few abstract classes within our code base. Adding more specific functionality can be achieved by extending those classes and adding the necessary code. This results in many interfaces that are used by only a single class. Although we like the flexibility this interface-driven-design gives - and prefer to do this again next time - we are not so sure about the naming convention (i.e. prefixing every interface with "I") and will instead postfix interfaces with "able" next time (think of Serializable, Comparable, Runnable, …).

Process reflection

Sprint workflow

Working in sprints was enforced by the guidelines of this project so obviously we adhered to this method. Our sprints ranged from saturday/sunday until friday so it's fair to call them weekly sprints. Our sprints were kicked off by a meeting in which we made up the sprint backlog from the overarching backlog by reforming and adding new issues to our git project. While doing this we designed user stories which we include in these issues and assigned a certain timespan with a priority tag to these issues. In the first few weeks we chose to use Planning Poker to plan the tasks because the team discusses each item thoroughly with Planning Poker, what is important when the project starts. Later on we reached consensus over the tasks by using Magic Estimation, what we learned in the agile lectures and what went a lot faster and smoother.

Issue workflow

We made several columns in our Waffle. On the left side was "Backlog" the place where all issues start. The next column was "Sprint backlog" where all issues for the coming week were placed. These columns were important and we'll continue to use them. We then had several columns we may omit next time. The first one was Ongoing user-stories and was meant for issues that were just containers of sub-issues. These sub-issues were placed in the column Ready that indicated that people could start working on it. However, because there were sub-sub-issues and sub-sub-sub-issues, the overview we got disappointed. Next time we'll search for a tool that's suitable for tracker such sub-sub-issues. Next there were the columns Requirement testing and Unit testing so that the programmer doesn't forget to test first, but in practice these were not needed anymore after a few weeks because we got accustomed to doing so. The columns In Progress, Review and Done are nice to have and we'll certainly continue to use them. The only improvement could be the fact that the issues weren't automatically transferred from one column to another upon its state or the change thereof which could be found on our git repository. For the issues we had to define the user story, the what, the why, the how and the definition of done. We liked that approach because it is easy to deduce from the requirements while also being explanatory enough for anyone to have an idea what the goals is of that issue. Because of these reasons we will continue to use this method.

Tools

We took this project as a chance to experiment with some new tools. As IDE we used Intellij (as part of Android Studio) which works great and we'll surely continue to use in the future. For the documents we initially used Quip which worked very well, but unfortunately turned out to be paid. We switched to Google Docs later on which works, albeit being a bit slow, fine. We had to use GitHub for the project, but we were allowed to experiment with a different issue tracker fortunately. We chose Waffle that worked quite well and we'll probably use again next time, but

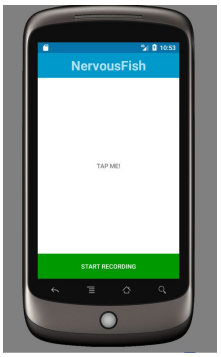
there were a few bugs in it that caused issues to not be tracked correctly. We reflected on this more thoroughly in the previous paragraph. Travis CI cost some time to get working and although it's free, the performance was not good. Builds took over half an hour, with the upper bound being 50 minutes which results in a failed build. So, next time we'll consider to use a paid alternative like Jenkins with Amazon Web Services.

Reviews

Every issue within our project was considered to be done when a pull request linked to the issue in question would be approved and merged. We used the built-in system of Github for making pull requests and reviewing them. Every pull request could be reviewed by any member, but we always tried to assign the most appropriate members to review first. In order for a pull request to be merged with the develop branch we made the requirement of at least two reviews that approved the pull request in question and no unanswered reviews that had requested changes. Sometimes this increased the time that an issue needed in order to be integrated with the main codebase but usually led to very neat and optimal code. The pull requests and their reviews ensured that every member was up to date with all of the code of our project which we certainly found to be very important. Because of this we'll always try to use such a system for our further projects. One other thing that may be important to note is the fact that every week we reviewed the way we reviewed individual issues and provided feedback on the global picture and progress of our project. We usually did this as part of our sprint retrospective.

Developed functionalities

Public key exchange via Bluetooth, NFC and QR



The main purpose of our app is to exchange public keys in a secure manner. A requirement was to do this via Bluetooth, and on top of that we added the possibility to exchange this via NFC and a QR code.
Bluetooth is the most insecure option out of the 3, and a MITM attack would be possible here. This was a challenge because we had to create a mechanism to prevent a MITM attack from happening. Our solution was to have an extra verification method on top of the default 6 numbers Bluetooth uses as verification.
Both of the verification methods create a key which will be used to encrypt the contact. This contact will be sent as an encrypted byte array to the other device and will be decrypted there. The key used for decryption is the one created by the verification method and is, assuming that the other user tapped the same rhythm / pattern, the same key.
This way it's a lot more difficult to intercept the data by using a MITM attack, because a MITM does not only have to crack the Diffie-Hellman challenge that's used by Android for Bluetooth communication, but also has to guess the key that the users share to decrypt the messages. The visual approach requires 6 elements to be

selected out of a grid of 12 elements, which gives 2.985.984 possible combinations. The rhythmic verification on the other hand has a much higher entropy, because the rhythm that you tap can be arbitrarily long.
NFC is another method in our app to exchange public keys. When you use NFC you have to put the phones in close proximity of each other and “beam” the data to the other device. Because of this, it is hard for a MITM to get in between the connection and change the data which is send.
The last method to exchange public keys is by scanning a QR code. This is the most simple method where the QR code just encodes the public key and name of the contact. This can be scanned by the other device to obtain the public key / name. Assuming that the users take care that other people (or camera’s) don’t see their QR codes, it is a very safe and easy method to exchange their public key and name.

Encrypted database

NervousFish is an app to exchange very confidential information, namely, keys. These apps must be stored securely so that other apps cannot just steal these very secret keys.
After several iterations, the database and it’s encryption became structured as followed: the database, meaning the user’s profile and it’s contacts, is saved in one file while the user’s password is saved in another file. The user’s password is hashed with a SHA-256 hash, meaning the program can only check if a given password matches the saved password. With this password an encryption key is made, based on PBE with MD5 and DES encryption algorithms, that can unlock a cipher that encrypts and decrypts strings.
The database in the application itself is an object with a profile and a list of contacts. To encrypt this object, we first translate it to a JSON string which we then encrypted using the cipher described above and this encrypted string is the one that we eventually write to the database file.
The database decryption is done once at the start of the programme. When the user puts in his password he decrypts the database file. The content in this file is loaded as an object and saved in the programme. Each operation on the database happens on this object and each update is followed by writing the updated object encrypted to the database file. To make sure we don’t always have to reenter the password we save the encryption key described above.

Custom keyboard

We created a custom keyboard to provide some extra security to the user. Third-party keyboards (for example Swiftkey or Swype) can track what the user is typing, which is not desired in our application because they could steal the user’s password in that case. For that reason we provide a custom keyboard that does not track what is being typed, which can be verified because the app is open-source.

Key management

Our intention was not only to make a secure app and implement the requirements, but also to make a usable app that users could use in real life. To do this we implemented some features that let the user handle the public keys of all his/her contacts and to let the user manage his own profile.
The main screen which user uses is a contact list with the names of their contacts. When the user pressed one of these contacts, the app shows additional information about the contact, lets the user change or delete the contact, and shows his/her public key(s).

Interaction Design

Introduction

For our interaction design research we used the “thinking out loud” method. This is suitable for our application because we had a hard time finding out if everything in the app was intuitive to use, and what users would think when they used the app. Even though our app, especially early on, will only be used by people who are computer savvy, we tried to make the app as easy as possible so people who have less experience with computers can use use it as well.
So the question we tried to answer with the these user tests is whether or not users who are not exceptionally good with computers can also use the app with ease. Ease of use, nowadays, is very important. Some apps have a lot of features which makes it hard to use and other apps have less features but are very easy to use. If users can choose between these two variants, the second one will prove more popular.

Method

We asked two pairs of people to test our app. The first one being the parents of one of the team members. They were our main subject in this experiment because they are both average with a computer and don’t know a lot about how public/private key encryption works. The other pair were two friends of a group member which both study computer science, this pair is more or less our target audience. We asked the pairs to do 3 things:

- Exchange their public key via one of the 3 methods possible in our app
- Delete a contact
- Exchange their public key via another one of the 3 methods

We wanted to say as little as possible so that the test subjects really have to figure out how to do what within the app. While figuring this out we asked them to think out loud and we recorded what they said so we could listen it at a later point in time.

Results

Because we used qualitative analysis we could easily identify improvement points from how the users handled our app and where they had problems. The users also gave direct feedback by speaking out loud and giving their thoughts about the app. Most important are:

- The explanations in the popups are not always clear.
- When creating a profile, it is not clear that a public/private key pair has to be generated.
- The no devices found when pairing via Bluetooth can be confusing.
- It's not clear in the Bluetooth exchange that the two tap patterns should be the same.
- In the visual verification layer it's not clear how much elements have to be tapped.
- It is not clear that you can click on a contact in the contacts list.

Conclusion and discussion

It was very useful to have two different pairs of people try our app. The parents and the computer science students both have a different look on the app. Most of the useful improvement point came from the parents who don't know a lot about how it works. The test with the computer science students was a stress test, where they tried to break our app.
Even though the feedback listed above are all stuff which is wrong with our app. There were a lot of things that were done right and were really intuitive to use, which is a good sign.
A large finding was that it is really important to write clear text in your app, and clearly explain what should be done at any point in time. This also applies to popups and error messages, they were sometimes not clear enough.
For a follow up on our app it will be key to have a larger group test it. We now only had two pairs of people test which is a small sample size. On the plus side of this small sample size is that we had time to go in depth with the users.

Evaluation

Functional modules

The individual components within the app are all more or less functional, although there are some known issues with in some components (for more details, see failure analysis) and some features missing in other components. Roughly speaking there are 5 functional components in the final product.
Connecting and exchanging keys, either via Bluetooth, NFC or QR codes, is the first functional part. There are a still a small number of issues with Bluetooth and NFC, this functionality is usable in the real world. All three ways to exchange keys can be used and work most of the time.

Managing your contacts, and to an extend their public keys, is the second functional component. It is working without any known issues, but the functionality and user experience could still be improved here.
The settings part of the application is a separate functional module as well. It lacks some functionality, mostly luxurious options like “Remember my bluetooth verification method”. The more important functions, like viewing your own keys, are working.
The login part of the application can also be considered a separate application. It is functioning as intended, but could be improved from a user experience point of view.
And finally the first-time-use functionality. Users interact only once with this part of the app, when they first use it. It is also fully functional and properly styled.

Entire product

The final product as a whole is an application with a consistent feel and style. It has many feature you would expect of a mobile application with high security concerns like always having to login, a custom keyboard and extra security measures for communication and saving data. Besides that the app is consistent with the Android ecosystem, utilizing many of its features and design elements.
All in all the app is working and usable with one weak spot in usability. As discussed in the Interaction Design section, especially less tech-savvy users have trouble using some parts of the application. This is one of the most important things to consider improving when development is continued.

Failure analysis

There are still a few issues in the final product left, the issues we are aware of that need fixing are:

- The Bluetooth connection does not work 100% of the time, especially starting a connection proofs difficult. For unknown reasons devices are unable to pair after 2 exchanges. Luckily this is not a common usecase for the application, so it shouldn’t cause much trouble.
- The NFC support currently has some issues when the application is not in the correct (i.e. NFC) android activity. The intention is to only allow NFC beaming from within this activity, but currently devices are able to receive data from within any activity in which case the data is not always handled correctly.

Outlook

Possible improvements

Multiple key management: we like the idea of a user having the ability to add/generate multiple key pairs in order to provide the flexibility of using different keys for different purposes. We have designed the app with this extension in mind already, so adding this feature won't take a lot of time. What hindered us were the more pressing issues that we had to finish for the main product and some game breaking bugs came up so we just haven't had the time to implement this feature.
Multiple key exchange: as a followup to having multiple keys available, it would also be nice to be able to select what key to exchange, and possibly to exchange multiple keys. In Bluetooth and NFC this can be achieved by adding the additional keys conditionally to the stream of data. In QR this could be achieved by generating a new QR code for each key.
Contact exchange: Once the app is used in the real world, it would be nice to exchange the keys of contacts you have exchanged with others. This is a Web-of-Trust and allows you to provide your friends at home with the key of a friend that lives far away whom you recently visited. This way your friends at home don't actually have to travel to that friend in order to get their key.
API: At the beginning of our project when we made up the project planning we decided to make a public API that other apps could use. The NervousFish app is used to manage all public keys of the user and other apps can ask NervousFish to encrypt a message for a certain person and to decrypt a message from a certain person. The private key must not be exposed through the API to other apps because that would be a huge security leak.

References

- Mutchukota, T. R., Panigrahy, S. K., & Jena, S. K. (n.d.). Man-in-the-Middle Attack and its Countermeasure in Bluetooth Secure Simple Pairing. Retrieved June 21, 2017, from http://dspace.nitrkl.ac.in/dspace/bitstream/2080/1524/1/MITM-SSP.pdf