BACHELOR THESIS
COMPUTER SCIENCE

RADBOUD UNIVERSITY

Accessing PEP data

Author:
Sander Hendrix
s4231589

First supervisor/assessor:
Prof.dr. B.P.F. Jacobs
`bart@cs.ru.nl`

Second supervisor/assessor:
Dr. Sietse Ringers
`sringers@cs.ru.nl`

April, 2018

Abstract

The PEP project lacks a way for study participants to access their data, as current access methods are not applicable to participants. We looked at alternative authentication methods, and provide a proof of concept of browser based access and authentication using the IRMA project. For doing so, we take a look at some of the inner workings of the PEP technology.

Contents

Chapter 1

Introduction

The Parkinson's POM is a collaboration between the Radboudumc, Radboud University, ParkinsonNet, and Verily. Research data includes highly privacy sensitive data such as DNA, MRI scans, and information from wearables. This data is stored and shared using the Digital Security Group's 'PEP' infrastructure. PEP stand for "Polymorphic Encryption and Pseudonymisation". It aims to store and exchange data in a privacy friendly way. It uses advanced encryption techniques, combined with distributed pseudonymisation and access management to accomplish this.

The current system implements infrastructure for medical researchers, but does not have a way for study participants to access their own data.

1.1 Aim

Current authentication methods and access methods are non applicable to participants. Our aim is to find both an alternative authentication method, and an alternative access method.

Chapter 2

Preliminaries

Our aim boils down to data access. Before we can talk about data access, we first have to introduce some details on how data is actually stored and some design principles of PEP.

2.1 Polymorphic Encryption and Pseudonymisation

Data is stored using 'polymorphic encryption' under a 'polymorphic pseudonym'. Without getting too technical, the main advantage of polymorphic encryption over 'normal' encryption is that the latter can only be decrypted using *one* key, while polymorphic encryption can be decrypted using *multiple* keys.

The polymorphic qualities are achieved by generating local keys for some client: it gets both a local private pseudonym key and data key. These keys are derived from, respectively, the master pseudonym key and the master data key. Because these local keys are derived from the master keys, they get a special property: data encrypted with the master key can be 'transformed' so that it can be decrypted by these local keys.

In many cases some party needs a persistent identifier for the person (i.e. study participant in our case) along with the data. To facilitate this, PEP uses 'polymorphic pseudonyms': like the encrypted data, these pseudonyms can be transformed into other pseudonyms, local to some party.

For both a better layman's introduction (with a padlock analogy accompanied by pictures) and the mathematical principles behind the technique we refer the reader to the PEP white paper [9].

2.2 Internal communication

Messages in PEP's are signed using certificates. For the different components that are active within PEP, these certificates can be generated beforehand.

This is not the case for other users and programs however. They first have to get their certificate somehow, and consequently their requests cannot be signed the same way as the other messages are.

The component within PEP that governs these certificates is the key server. It delegates this ‘initial step’ to the specialised authentication server. There, one can get a token that can be used to get a certificate at the key server. This token contains the user’s identifier and group, and is signed using a secret shared between the authentication server and the key server. Tokens are exchanged using the OAuth protocol [3]. The OAuth protocol only describes a protocol to ask and receive tokens. Thus, in turn, it delegates the notion of authorization and thereby that of authentication.

For this, we need some way of identifying participants. This can either be done via a pseudonym, or via their unique identifier. Research assessors (organisational staff for the project) access a study participant’s ‘dossier’ (i.e. the data they have access to) by entering a so called ‘Salesforce ID’: a fifteen digit long number, already in use in other Radboudumc systems that identifies a participant. These numbers form the basis for pseudonym generation used to store the study data. Since our aim is to let participants view their own data, and only theirs, there is no need to use a pseudonym. Hence, we too will use this ‘Salesforce ID’ to identify participants.

Chapter 3

Research

3.1 Authentication

The authentication server currently uses SURFconext[1] and the researcher's Radboud log in as authentication method. For study participants to gain access, they should be able to authenticate themselves in some alternative way. Since they do not have a Radboudumc or University account, an alternative authentication method needed to be found. We already have an identifier, the participants 'Salesforce ID', and need to find an authentication method.

3.1.1 Popular methods

Authentication methods come in many shapes and forms. From the simple user name and password combination we are all too familiar with, to something once thought to be futuristic as an iris scanner. We needed something simple and secure.

Simple in a sense that one should be able to authenticate oneself without intricate methods or exotic hardware. Considerations were the state participants are in: one should be reminded of the fact that these people are suffering from Parkinson's disease, and need not learn or do something difficult in addition to everything they already have to go through.

Serious contenders were the more 'tried and true'-methods such as a user name and password, or some kind of token or card. The workings of a user name and password are presumed known: often a email address or nickname in combination with a password. Smart cards are physical, often credit/debit card sized, cards with some storage. These cards need to interact with a compatible reader in order to function. A contemporary example of the use of a smart card would be the cards used by citizens of

[1] See: `https://www.surf.nl/diensten-en-producten/surfconext/index.html` (Dutch)

Estonia: this state issued card is used for identification by most of their government-citizen related interactions, such as tax declarations and their electronic voting system. Another example would be banks issuing readers that interact with their debit cards.

Advantages and disadvantages

User name and password combinations are often denoted as 'something one knows'. They have the advantage of being relatively simple to implement and have almost zero cost associated with them. Their main disadvantage is being 'yet another' set of user name and password to remember, and therefore able to be forgotten. This adds to the risk of people writing their combination or their passwords down, or choosing simple passwords.

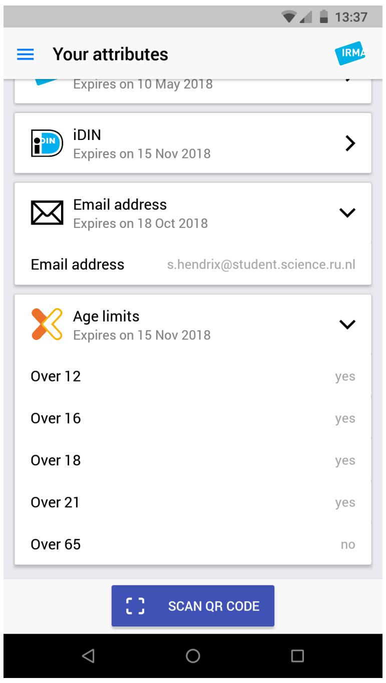
A password's strength lies in its length and the chosen character set: both longer passwords, and bigger character sets (e.g. special characters) take orders of magnitude longer to brute force than shorter or simpler passwords. Users, when given the chance, often tend to forgo these measures [2] [7]. Forcing these attributes, by setting a minimum length and a required number of non-alphabetical characters, often leads to passwords being significantly harder to remember and more likely to be written down [10]. In addition, users are often inclined to reveal their passwords [5].

Physical tokens, smart cards, or some other specialised device alleviate this main disadvantage and differ by their nature of being 'something one has/possesses'. Moreover, some implementations use an additional step often in the form of a PIN, effectively combining 'something one has' and 'something one knows': resulting in two layers of security and mitigating unauthorised usage. Disadvantages include user mobility—each devices requires some kind of reader—, cost, and performance [1]. They also open up other attack vectors such as power analysis attacks [6]. Finally, if a 'something one knows' step, such as a PIN, is not in place they can easily be used by an unauthorised party.

Taking into account these advantages and disadvantages: we considered a user name and password to be too insecure, but at the same time found the use of a smart card to be too complex. An in between method of something that would be as easy as showing a card, but not as cumbersome in practice as physical smart cards was preferred.

3.1.2 IRMA

Our eyes fell to another project of the Digital Security group: the IRMA project. IRMA stands for 'I Reveal My Attributes'. It aims to be a privacy-friendly way of authenticating, by only disclosing relevant properties that one has or is. It does so with 'attributes' that one can acquire and subsequently use. These attributes can range from being relatively simple such


13:37

Your attributes

Expires on 10 May 2018

iDIN
Expires on 15 Nov 2018

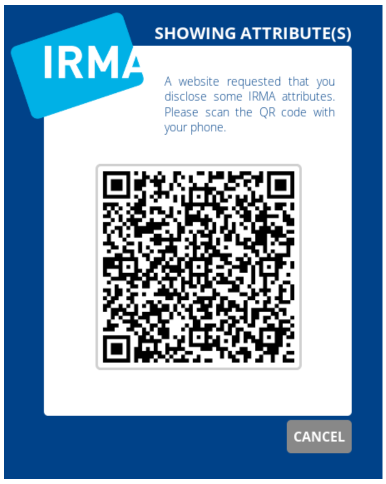
Email address
Expires on 18 Oct 2018

Email address	s.hendrix@student.science.ru.nl

Age limits
Expires on 15 Nov 2018

Over 12	yes
Over 16	yes
Over 18	yes
Over 21	yes
Over 65	no

(a) Example of the IRMA app with an email address and multiple age limit attributes shown


SHOWING ATTRIBUTE(S)

A website requested that you disclose some IRMA attributes. Please scan the QR code with your phone.

(b) A page requesting some attribute(s)

Figure 3.1: IRMA

as a ‘yes’ or ‘no’ to an age limit claim, to something more complex as an address. IRMA uses a smart phone app to manage, acquire, and reveal these attributes. See figure 3.1a for an example of how these attributes may look. A request to see an attribute is done via QR-codes. See figure 3.1b for an example of a typical request.

A use case, as an example, could be a store selling alcohol. Certainly, the store and shopkeeper only need to know whether a potential buyer is old enough, say: older than eighteen years old. Disclosing an ‘over 18’ attribute issued by a trusted party (i.e. the state) would suffice in this case, instead of having to show one’s whole passport or ID. This ensures that no unnecessary data, such as a national identification number or full names, is ‘leaked’ in the process.

IRMA takes the advantages of smart card based approaches, without the added cost and complexity. It also requires a PIN on attribute usage, thus combining ‘something you have’ and ‘something you know’. The PIN requirement also mitigates the risk of attribute misuse in case of theft of the device. A disadvantage of using IRMA is the need of a smart phone. Though smart phones are quite ubiquitous, it is not guaranteed that all participants own such a device. It also inherits the disadvantage of possible loss that smart cards have, though it should be noted that re-issuing the required attribute is the same as the initial process.

Implementation

We want participants to be able to authenticate themselves. We will do so by proof of identity using an IRMA attribute. We established that we will use the participants 'Salesforce ID' as identity in section 2.2. Ergo, the IRMA attribute will contain the participants 'Salesforce ID'.

This IRMA based approach has the added benefit of hiding the strenuous details of the concept of 'identity' from the end user: there is no need to remember the fifteen digit long 'Salesforce ID' number. Rather, it is more or less made opaque for the participant in the form of the IRMA attribute: our implementation can simply prompt the participant for their attribute, to which the participant only has to accept to continue.

3.2 Technical details

We will now go into more detail surrounding the relevant systems in place, and our additions to those systems.

3.2.1 OAuth

The OAuth protocol was briefly mentioned in section 2.2 as the protocol that the authentication server uses to exchange tokens.

OAuth is an open standard for so called 'access delegation': a resource owner (e.g. user) can grant a client access to said resource on another service. In other words, it allows a user to grant some service A access to some resource located at some other service B. 'Granting access', i.e. authorizing, is done by the resource owner at the service that holds the relevant resource. This is often accomplished by means of the resource owner authenticating themselves. A often used use case, that is also used for PEP, is using an account from a third party to log in to a service without giving the latter the password.

The OAuth specification defines four grant types, of which the PEP authentication server implements only one: the authorization code grant. Moreover, it also only accepts clients who implement PKCE [8]: a technique to mitigate against an attack on this grant.

Because this grant plays an important role in our additions to the PEP project, we will discuss the protocol briefly. It consists of four steps:

1. Authorization Request: the client directs the user's user-agent to the authorization endpoint. The authorization server establishes whether the resource owner grants or denies the client's access request.

2. Authorization Response: if the user grants the client access, the authorization server redirects the user-agent back to the client with an authorization code.

3. Access Token Request: the client requests an access token from the authorization server's token endpoint by including the authorization code received in the previous step.

4. Access Token Response: If valid, the authorization server responds by giving an access token. (read: the OAuth token)

The first step is one of the more complicated steps. The 'establishing whether the resource owner grants or denies the client's access request' part of the protocol is left to the authorization server and not part of the actual OAuth protocol. What needs to happen here, is between the server and the resource owner—i.e. the user. This is the actual authorization from the resource owner: the user says 'yes' or 'no'. This is often done by authentication: the user confirms by authenticating, and denies by cancelling the authentication.

Note that the 'client' is the program that the resource owner (the 'user' above) is using. The researchers use a native desktop application. Participants will use their browser to navigate to our public facing web server.

3.2.2 Current systems

Core library

The core library, internally often called 'corelib', exposes the main functionality of the PEP project. It does so by communicating with many of the core components of the PEP infrastructure. Besides very important aspects as logging, these core components are the key server, access manager, transcryptor and storage facility.

The key server checks the validity of the OAuth token and hands out the certificates that are used internally. It it also the only component that has the master data and pseudonym keys.

The access manager serves to check whether a request may be performed by someone. Requests are actions such as reading or writing data. Access is determined by means of 'groups': the group that is defined in the initial OAuth token made by the authentication server determines what actions can be done.

The transcryptor's main functionality lies in its role in transforming encrypted pseudonyms and data, as explained in section 2.1.

Finally, the storage facility stores the encrypted data.

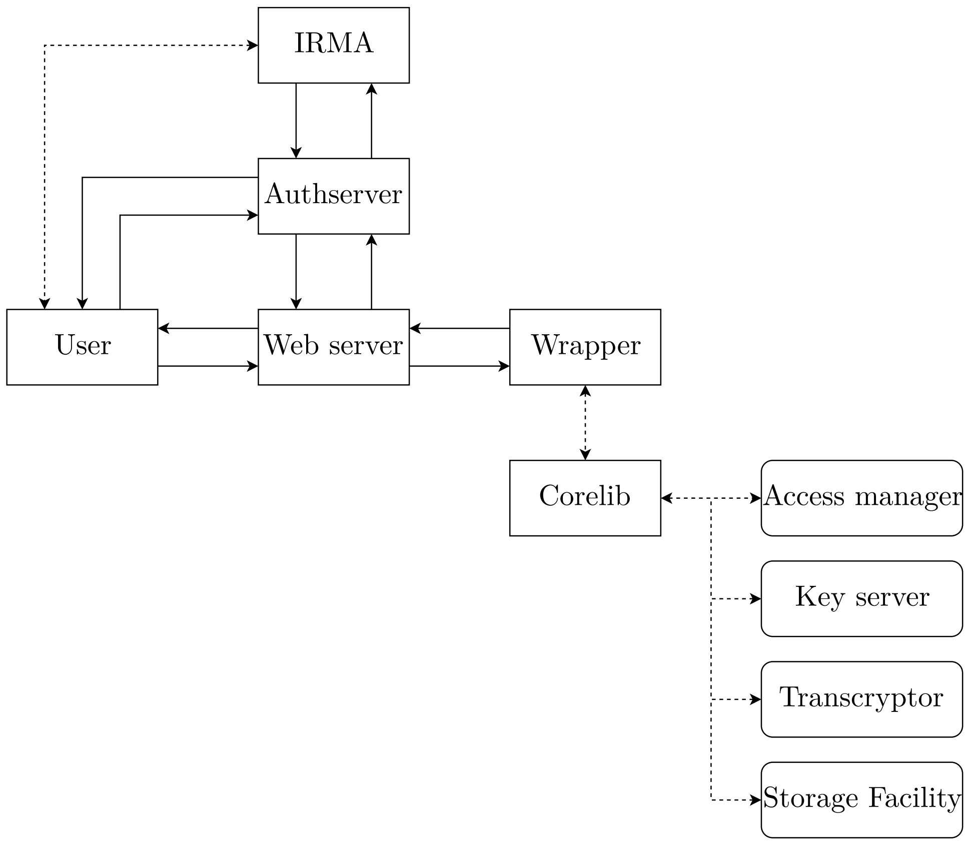


Figure 3.2: Interactions between the different components. Lines denote communication between two components. Solid lines are new interactions introduced, dotted lines are already present or given. See 3.2.4 for an overview of the final communications

Background The polymorphic qualities of PEP are achieved by a joint effort of the key server, the access manager, and the transcryptor. Local keys for some client are in fact blinded keys. They are derived from the master pseudonym key and the master data key only known to the key server, and scalars from both the access manager and the transcryptor. That is why decryption is only possible if all three grant their approval. The corelib handles the requests and receiving approval for us.

Authentication server

The authentication server speaks the OAuth protocol, as described in section 3.2.1. The tokens it issues are JSON web tokens [4] (often abbreviated as JWT[2]). They contain both an identifier and a group.

[2]Besides the RFC [4], see: `https://jwt.io/`

3.2.3 Additions

In this section we will talk about the additions we made to the existing PEP infrastructure. Independent additions, i.e. ones that did not exist prior, are the web server and wrapper. Modifications were made to the authentication server.

Authentication server

We will talk about the authentication server in terms of the OAuth protocol. If we fill in the abstract terms used in section 3.2.1 we get:

1. Authorization Request: the web server directs the participant's browser to the authentication server. The authentication server establishes whether the participants wants to log in (i.e. provide their IRMA attribute).
   - This is when the IRMA protocol starts: the authentication server prompts the participant to scan a QR code.
   - The participant scans the QR code and confirms they wants to share their IRMA attribute.
2. Authorization Response: if the participant does indeed provide their IRMA attribute, then the authentication server redirects the participant's browser back to the web server with an authorization code.
3. Access Token Request: the web server requests an access token from the authentication server using the authorization code.
4. Access Token Response: if the code is valid, then the authentication server gives the web server a token.

We will now take a closer look how exactly we implemented these steps.

Authorization code Currently, the server immediately redirects anyone accessing it to a SURFconext log in. There, a researcher or medical staff member can log in with their usual account. This had to be changed so our authentication method would be available. We chose to leave the current addresses in use in place, and instead opted for another link (say `/auth/staff` and `/auth/participant`).

Once the SURFconext-method or our method has an ID and group, the code for generating the authorization code is the same. Thus, the existing code could be made generic and was subsequently moved to a 'common part' of the server, and the SURFconext specific part was left at the current address.

As per section 3.1.2, we chose IRMA for the 'authorization by authentication'-step outlined in section 3.2.1. Details on the inner workings of the IRMA-protocol can be found on their website[3].

The aforementioned 'new address' points to a simple page that immediately loads the IRMA screen as seen in figure 3.1b. The participant is greeted with the usual QR code, scans it with the app, and confirms within the app. The authentication server verifies the information it gets from the IRMA server and returns the participant back to the website with the authorization code.

Much like the OAuth tokens that are generated, IRMA uses JWTs for communication. We implemented the following steps:

1. We generate a signed JWT, asking for the attribute with the participants 'Salesforce ID'.
2. Using `irma_js`[4], we present a QR-code that the user can scan.
3. `irma_js` handles the communication between the IRMA API server and the user's token (i.e. the smart phone app).
4. If the user consents, we get a signed JWT back. We verify its signature and extract the 'Salesforce ID' attribute out of the JWT. We send the it to the common part.
5. The common part redirects the participant back with an authorization code grant. (see step 2 in section 3.2.1)

Access token This part of the OAuth flow is the same regardless of authentication method. The underlying logic thus largely remains unmodified. Some SURFconext specific parts were made generic.

Web server and wrapper

The web server forms our main point of interaction with the participants, through their browser. It is written in Flask: a micro framework written in Python.

A wrapper was written in C++. It 'wraps' the core library. A 'wrapper' can be seen as a thin layer of code built around a library. It serves to translate a library's existing interface into a compatible interface. This can be done for several reasons.

In our case, it is done for language interoperability: the corelib is written in C++, the web server in Python. The wrapper is also written in C++, but

[3]See: `https://credentials.github.io/protocols/irma-protocol/`

[4]Provided by the IRMA team, see: `https://github.com/privacybydesign/irma_js`

provides Python bindings and an easy way to import it the same as Python-native code. This means that for the web server, it is just another import and no further steps have to be made to bridge the C++/Python gap.

It should also be noted that wrappers normally often wrap most if not all functions and subroutines of a library. Our wrapper only does so for functions that are relevant to us. Thus, it is not a complete wrapper of the corelib.

Operation The web server imports the wrapper. The wrapper exposes a 'retrieve' function that, when given a token, does the following:

1. Initialises and enrolls itself
2. Uses the 'Salesforce ID' (that is stored in the token) to request a polymorphic pseudonym (needed, because that is what is used internally)
3. Enumerates the pseudonym's data
4. Retrieves said data

It returns all the retrieved data. The web server takes the data, parses it, prepares a HTML document with a table containing the data, and serves this page to the participant's browser.

Subtleties The core library and most existing code make use of the C++ implementation of the ReactiveX[5] extensions. Among other things, this makes it easy for the programmer to make certain parts of a program asynchronous. This presents challenges for us, as their is no real system to integrate this with our website. To overcome this hurdle, the wrapper's retrieve function primitively synchronises the used PEP functions by simply waiting for them to be done.

3.2.4 Overview

The implemented flow, as can be seen in figure 3.2, is as follows—leaving the details of the IRMA API server/participant interaction and the inner core library workings mostly opaque:

1. *User* (participant) browses to the *web server* (a website);
2. *Web server* redirects the *user* to the *authentication server*;
3. *Authentication server* starts the *IRMA* procedure;
4. Upon successful *IRMA* authentication, the *authentication* server redirects the user back to the *web server* with an access code grant;

[5]See: `http://reactivex.io/`

5. *Web server* takes the access code and requests an access token from the *authentication server*;

6. If the access code is valid, the *authentication server* exchanges it for an access token and gives that back to the *web server*;

7. *Web server* takes the OAuth token and passes it on to the *wrapper*;

8. *Wrapper* takes the OAuth token, retrieves the data and returns it to the *web server*;

9. *Web server* parses the data and returns it in a table to the *user* (the participant's browser)

3.2.5 Implementation notes

- PEP uses both an 'id' and a 'group': we set the 'id' to the 'id' IRMA attribute, what in turn is set to the 'Salesforce ID'; we set 'group' hard coded to 'participant'. Systems are in place for a 'group' IRMA attribute, meaning this log in could be re-purposed for other groups in the future.

- The existence (nb: not their contents) of IRMA attributes is managed by an IRMA scheme manager. The attributes have to be added to an IRMA scheme manager, and the key material has to be exchanged. Because this is a proof of concept, attributes have yet to be added to a public scheme manager. Our proof of concept can always be tested by running a local instance of a scheme manager[6].

3.3 Scope

The issuing of attributes is left out of scope for this project. The idea is that interested participants can request and acquire an attribute at the Radboudumc. Though issuing is a relatively simple process to set up, constraints have to be made, and more importantly somehow enforced, as to whomever may issue these attributes. I.e. a web page only accessible from within the Radboudumc.

With that in place, the only steps required to issue an attribute would be a medical staff member who would insert the 'Salesforce ID' number, and the participant who would scan the issuance QR-code.

[6]See: https://github.com/privacybydesign/irma-demo-schememanager

Chapter 4

Conclusions

4.1 Discussion and future work

We note that the access manager does currently not incorporate a way to limit access by ID, only by group. The wrapper only requests data from the ID in the OAuth token, which is set to the ID contained in the IRMA attribute. Effectively enforcing access management. Since the authentication server, web server, and wrapper are all PEP operated, and there is no user or user-agent interaction (and therefore no real attack vector) between receiving the token and requesting the data, we see no real problem with this method. Still, it would be more in line with the functionality of the different PEP components if the access manager would perform additional checks.

We also acknowledge that our synchronisation process is quite crude, and may be able to be optimised or handled in some better fashion.

Other future work includes attribute issuance, as noted in section 3.3. This is mainly a question about logistics (how, where, by whom). An example using unsigned JWTs is included when running the server in its 'debug'-mode.

Finally, we would like to expand and formalise the functions of the web server in combination with the wrapper into an open API.

4.2 Analysis

We set out to establish a way for study participants to gain access to their study data. The importance of this task is further emphasised with the proximity of the European Union's new 'General Data Protection Regulation'. Its article 15 contains a 'right of access', intended for those whose data is being handled or processed. Looking back on our work, is there anything we would have done differently?

We share the view that access to one's own data is important. The current (not including our additions) implementation does allow for a simpler scheme where a participant may visit a research assessor: the assessor could then load a participant's info and allow them to look at the screen. This might satisfy the 'right to access' clause already. We opted for browser based access. Primarily because of its accessibility: browser based services are accessible from both mobile and desktop. It also removes the hurdle of physically having to go to a research assessor. Although one could also argue that accessibility is not that big of a concern. The question then would revolve around the required ease of access.

We also introduced new possible attack vectors. Most of the PEP infrastructure was only accessible from specific applications, and we introduced a public facing web server that can be accessed by anyone.

While we stand for the security of our implementation, some of the security aspects are not within our reach. Mainly, the end transport is only as secure as one's connection. Attacks include corporate style man-in-the-middle 'attacks' whereby a company intercepts all their employees' traffic and re-signs it with their own pre-installed certificate, or a similar case where the participant's own device is compromised. We do note that these practices are relatively easily avoidable by keeping devices up-to-date, using self-owned devices, using Firefox (who does not use system certificates) and practising safe browsing habits. Therefore, we do believe our implementation to be fully secure within reason.

We also believe our authentication method using IRMA to be fully secure. A scenario where someone gains access to both a participants phone and their app requires circumvention of, or access to, both the phone's lock screen and the IRMA PIN.

4.3 Experience

Here we will discuss some or our experiences making our proof of concept.

Our experience with the PEP technology and its implementation can be described as somewhat 'mixed'. The novelty of the technology is more than exciting. However, the size of the project can daunting for a new comer. This is further emphasised by the lack of documentation in a lot of places — though it should be noted that work is being done to provide documentation. This necessitated a dive into the code, to figure out interactions ourselves. The project uses ReactiveX, whose observerable and subscriber pattern can be quite difficult to grasp, further exacerbating the complexity. Luckily, once we had familiarised ourselves with ReactiveX, the existing code was very readable and proved to be useful as a guide in implementing our own methods.

Another hurdle was the the project's use of CMake for build management. We found its use to be very complex due to the size of the project, sub-components having separate CMake files, and a large chain of dependencies with, again, their own build files. This would have been less of a concern, if the components would have been dynamically linked. They were not, they were set up to compile into statically linked libraries. Nevertheless, our bridge to Python requires dynamically linked libraries to function. Changing the project from static to dynamic required a lot of time in the form of online reading and digging into CMake files and CMake sub-files that are pulled in at compile time.

The other new technology was IRMA. Including IRMA as authentication method was a relative breeze compared to the time spend on the C++ parts of PEP. Building a new version of the Android app (with our new attributes added) turned out to be easy. Running the API server (for testing purposes) caused little trouble as well, as everything is at least somewhat documented. Initial set up requires some effort, as new keys and signatures have to be generated and exchanged. Fortunately, after everything is set up, starting and stopping the server requires little effort and near zero maintenance.

Bibliography

[1] David Chadwick. Smart cards aren't always the smart choice. *Computer*, 32(12):142–143, 1999.

[2] Dinei Florencio and Cormac Herley. A large-scale study of web password habits. In *Proceedings of the 16th international conference on World Wide Web*, pages 657–666. ACM, 2007.

[3] D. Hardt. The oauth 2.0 authorization framework. RFC 6749, RFC Editor, October 2012. `https://www.rfc-editor.org/rfc/rfc6749.txt`.

[4] M. Jones, J. Bradley, and N. Sakimura. Json web token (jwt). RFC 7519, RFC Editor, May 2015. `https://www.rfc-editor.org/rfc/rfc7519.txt`.

[5] Daniel V Klein. Foiling the cracker: A survey of, and improvements to, password security. In *Proceedings of the 2nd USENIX Security Workshop*, pages 5–14, 1990.

[6] Thomas S Messerges, Ezzat A Dabbish, and Robert H Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE transactions on computers*, 51(5):541–552, 2002.

[7] Robert Morris and Ken Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, 1979.

[8] N. Sakimura, J. Bradley, and N. Agarwal. Proof key for code exchange by oauth public clients. RFC 7636, RFC Editor, September 2015. `https://www.rfc-editor.org/rfc/rfc7636.txt`.

[9] Eric Verheul, Bart Jacobs, Carlo Meijer, Mireille Hildebrandt, and Joeri de Ruiter. Polymorphic encryption and pseudonymisation for personalised healthcare. Cryptology ePrint Archive, Report 2016/411, 2016. `https://eprint.iacr.org/2016/411`.

[10] Moshe Zviran and William J Haga. Password security: an empirical study. *Journal of Management Information Systems*, 15(4):161–185, 1999.