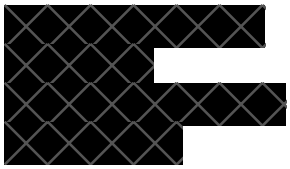
Information Security (2019-2020)

Infection Suspect Registration

Final Report

Group 3

Alexander Van Nevel



Abstract

In the last few weeks as the health crisis that is COVID-19 has derailed our everyday lives, calls for so-called “Contact Tracing” applications as a way forward for our society have grown louder. However, big concerns remain, like potential hazards for privacy and government tracking, new attack surfaces for hackers and the willingness of the public to allow for this kind of far-reaching tracking of our everyday lives.

For our course on information security we investigated the security concerns regarding such a contact tracing system and formed a proposal for how one could perform widespread contact tracing utilizing modern technology while minimizing potential privacy risks for end users.

The scheme we will be detailing is based primarily on asymmetric cryptography and digital signatures, where both the end-users of whom we want to trace interactions as well as the server-side application run by the national health authority have a public-private keypair used for communicating. The clients continuously broadcast a message, using a wireless communication technology such as Bluetooth, encrypted using the public key of the collection server. The clients receiving this message will at first store it local and on-device. Only once they are diagnosed with an illness, the broadcast messages they received over the course of the incubation period can be shared with the server application of the health authority. Locally stored interactions will be removed once an incubation period has passed as to minimize the potential exposure in case of a breach of the client’s device.

We also propose a way for health authorities to anonymously track social distancing efforts while retaining individual user’s anonymity in the process.

Finally we detail the proof of concept we implemented as a demonstration of our proposal.

Security & Privacy Concerns

In this chapter we discuss some of the high-level security requirements of a contact tracing application as well as the associated potential risks. We perform this analysis on the three main components of our tracing scheme.

Client-side interaction tracking

The first part of any contact tracing scheme is the manner through which interactions between users are observed and logged. Our scheme proposes the use of wireless communication to broadcast a message in the direct neighbourhood of the user. This does imply a number of security & privacy concerns:
We need to make sure that these broadcasts are in fact sent by the user they are claiming to originate from and no one else (**non-repudiation & data-integrity**), we ensure this by generating a public-private keypair on the client side used for *signing* interaction messages. It is possible that the user's public key gets compromised because of a security compromise of their mobile device, but to limit the possible risk that could follow out of such a hack the user has the ability to reset their own keypair.
We must also ensure that the broadcasted messages cannot be read or used as a fingerprint for the client by receiving parties which would allow for tracking of the user (**confidentiality of the broadcast**). Our implementation uses the collection server's private key to encrypt every broadcasted message (**confidentiality of the broadcast**) and includes a timestamp in the encrypted payload to ensure that its value changes periodically.
Another concern of ours was the potential for rebroadcasting attacks, where someone is able to pick up a broadcasted message and start sending out that same message in a different location, to limit the potential for this sort of attacks we use the timestamp in the signed message as a validation on the age of the broadcast as well as a rotation mechanism for collection server public key, limiting the useful lifespan of every broadcast.
Wireless communications also imply a risk factor in our solution, these are rather unsafe with their security being compromised several times in the past. So when exchanging messages via bluetooth (broadcasting and receiving) we need to isolate against additional attack vectors: Input validation of the messages and protection against flooding attacks would be essential; to protect against attacks where the client receives an overwhelming amount of messages.

Finally the way a user is informed of the potential risk of being infected is also another area of concern. Once a user is marked as infected by a health professional their close-contact encounters are identified through the broadcast messages they received. As a way to ensure **non-repudiation** and **authenticity** of the illness report, this reporting may only be

made by registered health professionals. Each of them also has a public-private keypair registered with the health authority. When a user falls ill they hand over their received broadcasts to the health professional who sends them in a signed (using their registered private key), encrypted message to the collection server. To keep the data of the clients safe, it is necessary that the information that is collected from the client is limited to the absolute minimum the application needs, therefore interactions only leave a user's device for the purpose of an illness report by a health professional otherwise the broadcast messages they receive stay on-device for the duration of the incubation period.

Server-side infection registration

With regards to the registration of a new infection a few major concerns do exist. First we need to be sure that an infection report was made by a health professional, for this we need to be able to assert **data-origin authentication** (the report must originate from a doctor and cannot have been tampered with in transit). We attain this using a digital signature of the infection report message for which the health professional has a keypair registered with the health authority running the contact tracing, this also ensures **non-repudiation** (an infection report cannot be disputed afterwards). These elements are critical to ensure no false reports can be made undermining user's confidence in the system.

Through the use of encryption of the illness report (using the public key of the collection server) we also ensure **data-confidentiality**. We don't want a rogue party to be able to see the contents of an infection report for they would be able to extract some information (like the number of interactions of the infected client) as well as the public key of the reporting physician which may allow them to extrapolate the identity of the infected client (the broadcasts themselves are already in an encrypted form and thus their contents cannot be read).

By not allowing self-reporting and mandating an interaction with a registered health professional who can see the number of interactions they are reporting along with the disease report (but as mentioned before would not be in the capacity to see their contents), this provides an additional barrier making it harder for bad actors to abuse the system with fake keypairs or rogue reports.

Finally the availability of this system is also important if a public health authority were to strongly rely on it. Fortunately thanks to the decentralized mechanics of our proposal, an outage of the collection server would not significantly impact the client-side interaction logging, given that clients only need to obtain the new public key of the server roughly once every day. Only the infection suspect notifications and report processing would be delayed.

Gathering of anonymized interaction statistics

Since we keep track of all the statistics by saving them in some persisted datasource, we don't want to keep any sensitive data like keys, exact locations or user information like an

IP address which could possibly allow for connecting a user's real world identity to their public key used in broadcast messages. We try to achieve this with a separate reporting server. This server would consist of a single endpoint and a datasource holding the statistics. In case of a hack we can ensure no sensitive data was stolen, given that only aggregated statistics are stored. In case the hackers tampered with the statistics, we could fallback to a backup.
This server will have an endpoint for a client to send statistics to the server. Given that we don't want to use the user's keypair for broadcast messages to authenticate their reports to this server the endpoint is at risk of being spammed with useless information or skewed statistics, taking all the servers' resources. We attempt to partially prevent this by using a token for sending the aggregate statistics that will become valid after a specified amount of time (e.g. 15 minutes). Only then a client will be able to send a report. The only problem that remains is a user requesting a lot of tokens, thus circumventing this measure. If we want to prevent this as well, a firewall that filters rogue IP addresses could solve this or a more radical solution would have to ask for some sort of identification which then might threaten privacy. We conclude this to be a delicate balance between **privacy** and **accuracy** of the statistics.
The problem of clients sending false information still partially remains. A rogue client can rebroadcast messages in a specific location thus making others report false statistics or just send some made up statistics. The latter can be fixed to a certain extent by using some statistical methods like outlier detection.

Our Proposal

1. Client-side interaction tracking

In this paragraph we will be detailing the mechanisms used for generating the broadcast messages, keeping track of received broadcasts and getting notified of a potential infection risk. Then we'll discuss potential security concerns we identified and discuss which mitigations could be implemented to minimize their impact.

Generating broadcast messages

A user will have an application installed on their device, which on first use generates a public-private keypair. For our practical implementation we chose RSA 2048 bit, since it should be sufficiently safe until about 2030. In the real world, we'd rather go for ECC instead.

This per-client public-private keypair will be used as a:

1) Unique identifier for checking whether a client has a potential risk of disease (see later).
2) Signing the messages broadcasted by this client.

We propose a server (which we will be naming the **collection server** throughout this report) dealing with reporting of the diseased status of the user as well as the close-contact interactions they have had during the incubation period, this server has its own public-private keypair. The public key of the server needs to be obtained by the client through a secure channel like HTTPS to ensure the data integrity and authenticity of the public key. The endpoint of this server would be preconfigured in the source code of the client application.

Before we move on to the broadcast messages itself, let's first take a look at the general mechanism using which messages are constructed in our application. The first step is constructing an unsigned, unencrypted message which contains the timestamp and public key of the sender as header and a payload. The next step is to sign this message using our client's private key using HMAC SHA-256 and append the signature. This results in a signed, unencrypted message as shown below. Finally we want to encrypt this signed message.

Unfortunately, this message is too big to encrypt asymmetrically with the collection server's public key, and even if it were possible, asymmetric encryption on large messages is rather slow. The solution is to use a Key Exchange Mechanism. We first generate a symmetric key to use with AES-128 GCM, which should be secure enough for the foreseeable future. In our practical implementation, we used RSA-KEM. Then we use that symmetric key to encrypt our signed message. Finally, the symmetric key gets encrypted with the public key of the collection server using RSA and it is appended to the signed, encrypted message along with the same timestamp used earlier, but this timestamp is unencrypted. The unencrypted timestamp is required for the collection server to know which private key to use for decryption (as described later, the collection server keypair periodically rotates).

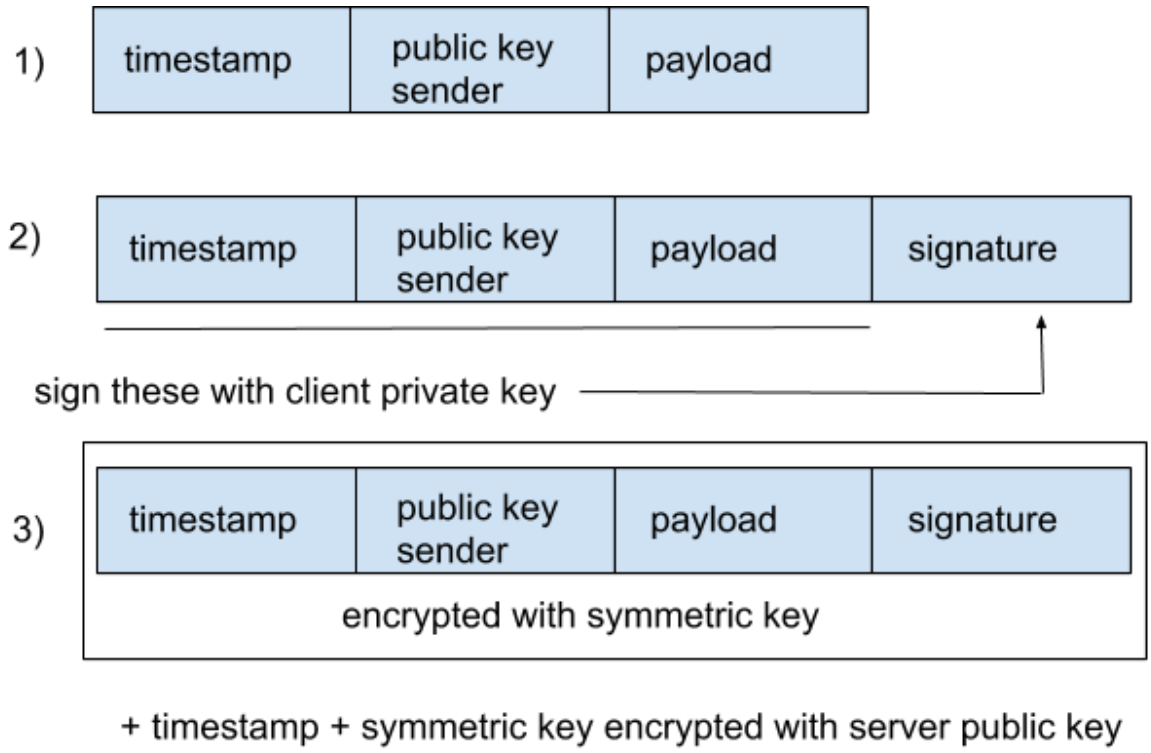


Figure 1. Visualisation of message construction
Final message seen in step 3

Let's take a look at how the broadcasting works: The sending client will construct a message with an empty payload using the approach described above. The client then broadcasts this message over Bluetooth.

Processing received broadcasts

A broadcasted message *M* is received through Bluetooth. The receiving client does a basic sanity check to verify *M* has the correct (fixed) length. The receiving client constructs a new message in the same way as described earlier, but this time with as payload the message *M*. The newly constructed message is then stored locally. Note that we don't want to send these interaction messages to the server unconditionally. Sharing these with the server is a potential privacy hazard, given that the server can decrypt their contents, which is why we chose to only share these messages with the server in case of a reported illness (see later).

To clear up storage, messages that are a little bit older than the incubation period of the disease (a timeframe which needs to be preconfigured in the application) will be deleted periodically. We don't want the threshold to be the exact same as the incubation period,

since not all people experience the same incubation period, so it should probably be a little bit longer.

Infection risk notification

When a client is online he is able to verify his risk status. This risk status let the user know if there was any interaction with someone else who appeared to be infected, later on. Periodically, every few seconds, the status is updated in the client tab. The risk status is obtained by making a base-64 encoding of the client's public key and sending it in a request to a specific endpoint of the collection server. The response body of that call is a message containing a status code and timestamp, encrypted with the client's public key. This ensures that only that specific client can obtain the infection status of the keypair. A 404 code means that there is no risk for potential infection, a 200 code means that there is risk for a potential infection and when the code is not recognised, we inform the user that the received code is unknown. To limit the load on this endpoint, a timeout is set to limit calls to the endpoint per user to one call every x seconds (we chose 10 seconds for the proof-of-concept, but this could be set much higher for a real application). However, as detailed before, it is an additional attack vector that needs to be protected using additional measures like a firewall to prevent DDOS attacks.

Security concerns

A big rather social concern is "when do we inform people they are potentially at risk?" We do not want to send too many messages as this could decrease trust in the accuracy or cause unnecessary panic in the population. Therefore, we were thinking of setting a threshold on the number broadcast messages that need to be received before we mark an encounter as a close-contact (and potentially riskful) encounter. Before we could set this threshold however there first needs to be further study of both the infectiousness of the disease and what level of interaction is needed for an encounter to be high risk. In this proof of concept we weren't able to further investigate this, but it is most definitely a point of further study in this field. Also in order to suppress the amount of false positives we might get it doesn't seem unreasonable to only enable the broadcasting/receiving of messages when they go outside. Otherwise, someone living close to a busy road or in an apartment could receive messages from people they haven't been in contact with, like the neighbour up- or downstairs. This however does imply that the interaction logging may be incomplete, for instance what if you welcome an infected visitor into your house. Further investigation into the likelihood of false positives could provide answers as to whether this mitigation is necessary.
Other potential risks can come from the always-on wireless connection, this way attacks might happen on your mobile device, something we already hinted at before. Fully mitigating the possible attacks that stem from the use of wireless communications is very hard, message validation and protections against flooding attacks could potentially solve some of our concerns in this regard. However it does still remain a major hazard. Also people might be able to track you using this always-on wireless connection, we already

attempt to somewhat mitigate this by making fingerprinting using the broadcast messages impossible. However, identifiers of the wireless radios in your mobile device might still allow bad actors to track you. Other risks like the rebroadcasting of the infection risk message are reduced by the rotating keypair of the collection server and the timestamp included in the message, however they are still a real possibility we couldn't fully mitigate (for instance what if a rogue user intercepts your broadcast messages and also starts broadcasting them in an entirely different location at the same time, it would be very hard to mitigate against this without including for instance location data to the broadcast but that would inherently compromise privacy).

2. Reporting illness & Sharing interaction data

In this paragraph we will detail the server-side mechanisms used for key-management, illness reporting by a medical professional and finally the processing of interaction data reported by an infected client. Then we'll discuss potential security concerns we identified and consider which mitigations could be implemented to minimize their impact.

Server-side key-management

As noted in previous paragraphs the public key of the server responsible for infection tracking (the collection server) is used to encrypt the contents of messages broadcasted and sent by the client. Using a single static keypair for this would mean that if this keypair were breached, the contents of all broadcast messages could be read by the public. Of course, this is a risk we want to minimize as much as possible. Secondly it also means that the same keypair is used for all broadcasts, even older ones for which the incubation period of the disease has long passed. This also is a potential risk for user's privacy.

Our proposal therefore uses a rotating keypair on the server-side. Given that incubation periods are typically measured in days we opt for a daily rotation of the keypair (however this could be modified based on the specifics of the disease). The client running on the end-user's mobile device has a predefined HTTPS protected endpoint configured through which it can obtain the current public key of the collection server to be used for encrypting its messages.

After generating a new keypair the server also persists a copy of the public and private key. Our recommendation would be to ensure that the implementation of the endpoint for providing the public key to clients only ever has access to the public key information. The private key should only be accessible by the mechanism responsible for processing infection reports (for which the received broadcast messages must be decrypted), this could even happen on a separate server not exposed to the public internet.

After a configured incubation period has passed we would strongly recommend for the collection server implementation to delete the keypair, this would ensure that old interaction data is rendered useless even if the client were unable to delete the older broadcast messages it had logged (for instance because the device had run out of battery).

Illness reporting by a medical professional

When a client is ill they will go to a doctor in order to get tested. We will then use the doctor's device to send the interaction messages of the ill client to the server. The client has to hand over his or her interactions to the doctor using for instance bluetooth or some other wireless communication (the specifics mechanisms of this exchange were not implemented in our proof-of-concept however data-origin authentication & integrity of the interactions sent by the ill client would be required). Because the received broadcast messages are signed using the receiving clients public key upon receipt we already have non-repudiation and data-integrity of the interactions received by the client (with the one concern however still remaining that a client could erase their interactions prior to their consultation, however in a voluntary system such like our proposal this is impossible to protect against). The doctor will finally upload these messages over to the collection server. To do that he will put all the interactions and the public key of the diseased client inside his own signed message, using a separate keypair that is registered with the national health authority (not the keypair for his own logging client) thus ensuring data-origin and non-repudiation of the disease report. The additional upside of mandating an interaction with a health professional is that we do not have to take into account the potential risk of self reporting causing mass panic among the population and attest that there is in fact a real person attached to the keypair.

The server receives the doctor's message, which is constructed in the same way as described earlier. Due to the fact that this message is encrypted and signed, bad actors cannot see the public key of the doctor or the number of interactions they are reporting (which could allow for the identification of the diseased user). The collection server will first decrypt the symmetric key using the private key that was assigned for the timestamp of the message. Then, decrypt the message using that symmetric key, and finally verify the signature of the decrypted message and check if the public key sent by the doctor is a registered one. It will do this by checking a database with the public keys of health providers maintained by the national health authority.

We now have signed, encrypted interaction messages. Each of these interaction messages is decrypted and verified in the following way. First we get the timestamp from the message and look in the historical keypairs stored by the collection server for the appropriate private key. Then the message as well as its payload (the received broadcast) is decrypted using this key. We check if the timestamps of the interaction message and the

broadcast message it contains as a payload are not too far apart for extra security. At each step we verify the signature, firstly that of the receiving client that is being marked as ill and secondly that of the broadcasting client to ensure the authenticity and integrity of the broadcast.

Once we decrypt the payload we obtain the broadcast message containing the public key of the client this diseased user has interacted with. As mentioned before we verify its integrity and authenticity using the signature. Once the broadcast message's signature is verified we create a secure hash (SHA-256) of the sender's public key. This hash together with the timestamp is then stored in a database. Now we are able to check if a user has been in contact with the illness by looking if the hash of the client's public key is in the database. This information is used for the risk endpoint of our collection server as detailed earlier, by using a secure hash we also minimize the risk of an information leak of our database. By also storing a timestamp along with the public key we can periodically remove the messages that fall outside the incubation period from the database thus further limiting the potential privacy concerns for end users.

Security concerns

Risk of an exposed server keypair

As highlighted earlier an exposed server keypair could breach the confidentiality of the broadcast messages and allow for tracking of users. Therefore we strongly insist on using rotating keypairs as to limit the risk of an exposed keypair on one hand and to guarantee a user's privacy once the incubation period has passed through the deletion of old keypairs.

Risk of an exposed medical worker's keypair

When a medical worker's keypair is exposed you would be able to send disease reports to the collection server. People could get fake risk messages on their devices. We can mitigate this kind of attack by putting the exposed keypair on a blacklist. We would recommend using a logging service that maintains a record of a doctor's actions and detects malicious behavior. Besides this logging service, doctors would also be able to report that their keypair is exposed, the national health authority should provide adequate procedures for this also ensuring that no one else can report the exposure of a keypair besides the doctors themselves.

In case this does happen it would be wise to also store (a hash of) the public key of the reporting doctor alongside their reported infection suspect interaction in the database. So that we can contact users if they were the victim of a fake report causing a risk notification on their mobile device. It is important to put adequate procedures in place for when this occurs because it could undermine user's trust in the service.

3. Gathering of anonymized interaction statistics

In this paragraph we discuss an extension to the contact tracing system in detail, namely a mechanism of gathering anonymized statistics to evaluate the efficacy of social distancing measures.

Client-side reporting of interaction statistics

We specified earlier that a client will keep track of received broadcast messages in the local storage of their device. Due to the use of broadcasting, this will be a list that gets updated continuously. In our scheme the client will send their aggregate statistics of their interactions to a reporting server. This message will contain an approximate location, within a few kilometers of accuracy, and the number of interactions that were observed by the client in a preconfigured interval up to the moment reporting. To limit the rate at which these interactions are sent to the server (and prevent rogue clients from sending a lot of fake interactions statistics), the client has to request and use a JSON web token to authenticate their request. This token will be included in the request. The structure of a full request is shown below.

Latitude	Longitude	Amount of interactions	Token

Server-side processing of these statistics

To collect all the interaction statistics we opted to use a separate server. We prefer this solution as it prevents the coupling of the keypairs used in broadcasts to location data via the client's IP address for example. However this does rely on a trust factor because one can not fully rule out that some malicious health authority does try to link this information. No keys are required to send the statistics to the reporting server. The client simply requests a JSON web token that can be used to send the data. The server signs the web token in a way that the token will only be valid after a certain amount of time (e.g. 15 minutes). This way we have a mechanism of limiting the rate at which interactions can be sent. We can discard the report if the token turns out to be invalid, thus not wasting any server resources and can use this time between reports to process the received information.

Security concerns

For this proof-of-concept the two servers are run on the same server simply using a different port. In a real life application these would need to be run on two separate servers. That way the reporting server doesn't have access to any sensitive data and is unable to

couple the data received for disease reporting with the location data obtained for the statistics.
To limit request rates we use a JSON web token. This token will only become valid after a specified amount of time. However a rogue user could still request a large number of tokens thus allowing them to spam our server, a web application firewall could somewhat mitigate this.
Another concern that remains is a rogue client broadcasting valid messages, thus skewing the reports made by real clients. This could be partially fixed through outlier detection. But getting a sufficiently correct outlier detection is very hard to achieve. We could make the reporting more reliable by coupling it with the keypairs used for infection reporting but this would come at the cost of user privacy.
Finally the access to the logged statistics should also be protected (as to ensure only authorized users can see this information) and safeguards for GDPR compliance should be put into place (e.g. correctly informing the clients of the way their location is being monitored through a privacy policy).

Proof-of-concept

The goal of making the proof of concept was to closely resemble the process of generating and sending the encrypted with all security measures in place. But all this without working out the details and some technical things like sending messages via bluetooth. Also A lot of info is shown in the proof-of-concept to easily show what is going on under the hood. A fully functional app for the public would probably be nothing more but a button turn tracking on/off and a download interactions button. And maybe a check risk button but that might also just be called periodically.

There are 3 applications, the client Application is used to do the tracking send/receive interactions and notify people for potential risk. The doctor app is to diagnose clients with Covid-19 and send their interactions to the collection server to warn the clients that had an interaction with this person. Then we have the reporting app that is used to make reports about the amounts of registered interactions.

Technical Details

We used express.js [5] to make the backend of the application paired with handlebars [6] for the frontend templating. All of the encryption and signing was done using Digital Bazaar's forge library [7].

Simplifications

For the proof-of-concept we used an Express.js server running on multiple ports resembling the different servers, normally we would use multiple servers for this but to

make it easier for ourselves we did it like this. We have 2 servers, a collection server and a reporting server. The collection server is used to store interactions and check risk. The reporting server is used to report the aggregate statistics of the amount of interactions encountered by a client (this could be used for a heatmap of infection hazards by location). There are 3 applications that are all on the same website to make it easy-to-use but normally these should be separate mobile/web applications.
We also used RSA 2048 bit encryption for our keypairs, a better option would be to use ECC but since not a lot of JavaScript libraries support it we used RSA instead. We also don't use Bluetooth to transport the messages but do this manually by copy & pasting or typing them in the other application.

Use

The proof-of-concept is bundled with a `docker-compose.yml` file allowing you to start both the MongoDB database and the Node.JS server application with one single command.

```
docker-compose up
```

Once docker-compose has finished, you can use the proof-of-concept by navigating to http://localhost:3000 with your web browser. In order to fully test its functionality we advise you to open a second browser window (a private tab or different browser client so that you can demo the application as if two different users were interacting given that we use local storage for storing the keypair and interactions). For further details we refer to the README.md file in the root of the Proof-of-Concept project.

The Client Application

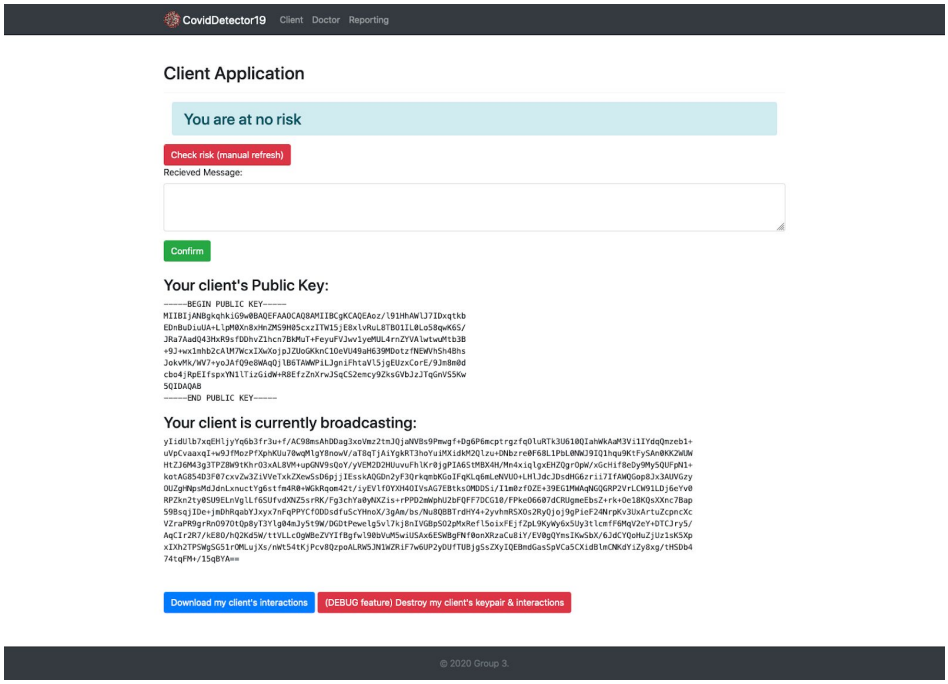


Figure 2: The "Client" page of our web application.

Upon first use you should navigate to the Client page using the header menu and will be prompted to generate a new client keypair. Afterwards you will see the page shown in figure 2 displaying a PEM-formatted version of your public key as well as the current broadcast message of your client (which would in practice be broadcasted using Bluetooth). You also get the option of downloading your client's interactions (which is later used in simulating a disease report) and to reset the application.

On the client page a text-field is shown in which you can input a broadcast message from another client (a separate browser or in private-tab with the client open) to simulate two clients interacting.

Finally at the top of the page your current infection status is shown as well as a button to manually request an update of this status.

Reporting Disease

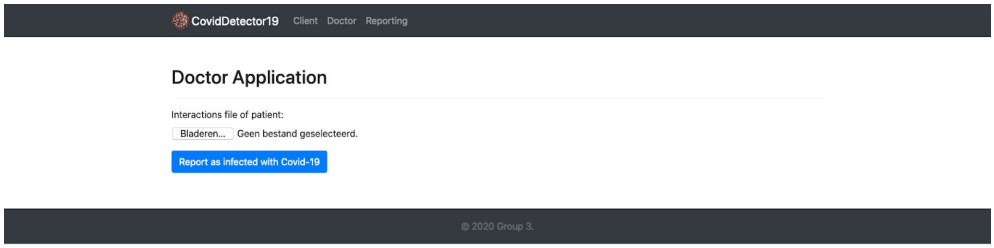


Figure 3: The "Doctor" page of our web application

To report a disease the client would normally send the received broadcasts to a registered health professional via Bluetooth. For the proof-of-concept we used a simpler approach to simulate that interaction. The ill client can download his/her interactions to a JSON file (on the client page), that JSON file can then be uploaded on the doctor page (see figure 3). A registered health professional keypair was statically configured using a config-file in our proof-of-concept allowing you to simulate a report, in practice however a doctor would use their own local keypair.

Sending of Aggregate Statistics

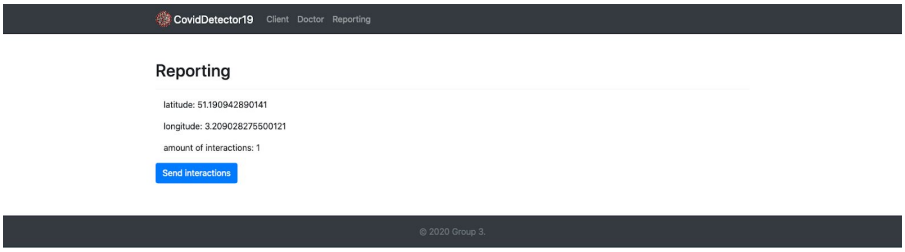


Figure 4: The "Reporting" page of our web application

All received interactions are saved in LocalStorage using JavaScript in our proof-of-concept. We also store the timestamp of the latest interaction that was included in the statistics in order to know where to start counting interactions for the next report.

To send the statistics to the server, we count the amount of interactions after this timestamp. This amount we'll send to the server along with a location and a JSON web token. In order to send, we request a token from the statistics server. As detailed before the token only becomes valid after a certain amount of time to (partially) prevent abuse, in the proof-of-concept this token will become valid after 30 seconds. After waiting 30 seconds we send the statistics to the server.

The server will drop the request in case of an invalid token. You can simulate this by navigating to the reporting page of our proof-of-concept (as seen in figure 4 and clicking on "Send interactions" to engage the described mechanism). Given that user locations can only be accessed on HTTPS-enabled endpoints we cannot simulate exact locations in this locally running proof-of-concept, therefore a location has been preset.

Viewing of Aggregate Statistics

To view the statistics a heatmap would in practice be used, however for this proof-of-concept we just created an extra endpoint to view all the raw statistics. In practice one would protect this endpoint only allowing people registered by the national health authority to see this data. However given that it was only mentioned as an extension to the main subject of this project (infection suspect registration) we decided that this implementation sufficed for our proof-of-concept.

Conclusion

As we highlighted over the course of this report, it is in fact possible to create a contact tracing scheme with aggregate statistics collection that safeguards end-user's privacy through extensive use of asymmetric cryptography and signatures. We provided safeguards against the most common concerns an end user might have, like fingerprinting and de-anonymization of their interactions (through the use of encrypted broadcast messages) and far-reaching oversight by the government (by storing the interactions on-device unless one actually is tested positive by a medical professional). This anonymity however comes at a cost, there is a risk of rogue users skewing statistics or corrupting reported interactions. We somewhat attempted to mitigate these by mandating interaction with a physician before riskful interactions are reported and notifications are sent out to the clients at-risk and by limiting the frequency at which statistics reporting can occur as to prevent skewing of the data by rogue clients. However this doesn't fully prevent some (more subtle attacks). Also concerns like false-positives and false-negatives caused by the inherently unpredictable nature of wireless communication are hard to mitigate and warrant further research. An ethical and health care component is also associated with this topic, especially as to the course of action with people who've had a potential riskful interaction: how should they be informed and what do we do if they don't follow recommendations.

Therefore a solution as proposed in this report could only be useful to society if adequate follow-up of riskful contacts is in place, otherwise its efficacy is limited only to those who use the app and follow its recommendations after a riskful interaction. It should thus be part of a wider strategy of contact tracing, and the application's source code should be transparent to the public so that its correct working can be independently verified.

Sources

[1] M. Hashemian, D. Knowles, J. Calver, W. Qian, M. Bullock, S. Bell, R. Mandryk, N. Osg and K. Stanley, "iEpi", *Proceedings of the 2nd ACM international workshop on Pervasive Wireless Healthcare - MobileHealth '12*, 2012.
[2] Chia-Chun Wu, Wei-Bin Lee and Woei-Jiunn Tsaur, "A Secure Authentication Scheme with Anonymity for Wireless Communications", *IEEE Communications Letters*, vol. 12, no. 10, pp. 722-723, 2008.
[3] C. Troncoso et al., 2020. Decentralized Privacy-Preserving Proximity Tracing. [online] Available at: <https://github.com/DP-3T/documents> [Accessed 4 April 2020].
[4] R. Fenwick et al., 2020. Help stop COVID-19 with crowdsourced data. [online] Available at: <https://www.covid-watch.org/article> [Accessed 4 April 2020].
[5] TJ Holowaychuk et al., 2010. Express.js. [online] Available at: <https://expressjs.com/> [Accessed 16 May 2020].
[6] Yehuda Katz, 2011.Handlebars. [online] Available at: <https://handlebarsjs.com/> [Accessed 16 May 2020].
[7] Digital Bazaar inc., 2010. forge. [online] Available at: <https://github.com/digitalbazaar/forge> [Accessed 16 May 2020].