crocofile

Exam Project

https://github.com/andreiho/crocofile

Web Development Team 10 Fall

Web Security Module

Lecturer

Danny Kallas

Group 3

Andrei Horodinca

Jonas Skovsgaard Christensen

Sarah Braun

Index

1. Problem Formulation

In this project we set out to create a host-proof, anonymous, encrypted file-sharing platform with the possibility to deny file ownership (given that none of the user’s session data during the upload gets intercepted).

How is it host proof?

The files get encrypted in the browser and only the user has the passphrase to decrypt.

How is it anonymous?

The user does not have to be registered to upload a file. To avoid legal complications we are logging IP addresses, so it is up to the user to chose an anonymity network / proxy / vpn for uploading if they want to be truly anonymous.
In order to share files users have to be registered for exchanging passphrases with people they do not know personally. Usernames are all visible and tagging a file with a username is completely arbitrary. So a user can always claim someone else tagged them. Since we only log the IP on fileupload (and not the logged in user if there is one), the file uploader's anonymity is solely dependent on that IP address.

2. Introduction

As of the time of handing in a user can:

- / index route

  upload (encrypt) a file
- / download route (+ file id)

  download (decrypt) a file
- / deletion route (+ file id)

  delete a file (if a password has been set on upload)
- / vault route

  see all uploaded files and tags, get their respective download links and see users they are tagged with, as well as the time they have been uploaded
- / login /logout routes

  log in if registered and log out if logged in
- / registration route

  register with a username and password

- site wide

  message other online users if registered and logged in

After giving an overview over our security considerations in the planning phase, we will go through the separate features and layers of our application, and highlight where they have been applied.
The features discussed in the following include:

- User Registration
- User Login
- File Upload (encryption)
- File Download (decryption)
- File Deletion
- Messaging

We will furthermore examine the server-side setup and the database, as well as arguing UX considerations. We will give a short description of the technologies used and then focus on explaining how they have been implemented, describing hurdles we encountered and how they have been overcome.

3. Security considerations in the planning phase

After having established that we want to implement an encrypted file-sharing service we tried to identify possible attack vectors in order to streamline our planning accordingly.
Since we wanted anonymity, data economy was one of the obvious paradigms. Since we do not store any sensitive user data, the files and communication become more or less our biggest weaknesses.

As the system is supposed to be host-proof and encryption thereby happening in the browser, the security of the files is given, even in case of a compromised server or database. On the other hand certain XSS attacks would have grave consequences.
However, since retain IP addresses of the users uploading files, there is still some information to be protected in the database as well.

As of concept, it is a file-sharing service, so users are to share their passphrases to the files. This makes the channel they use to exchange those, the other area of increased interest.
We were planning on using OTR and WebRTC for this. Communications should not be logged and rather self-destruct after being read. There is no user review system planned in the scope of the project, so users have to trust each other delivering the right passphrase to a file when an exchange is negotiated.

As first and foremost design principle we chose:

Never assume that your secrets are safe

We believe this to be one of the most important principles in general. When opting for security one might be easily tempted to think one should keep the workings of the application secret and the code obscured. We believe this to be wrong. If an application is vulnerable open source, it is just as vulnerable with obfuscated code. There is a score of tools to reverse-engineer the architecture by simply analyzing responses.
Furthermore it is impossible to anticipate all attack vectors and many vulnerabilities of technology in use might not yet be discovered.

A good strategy is to compartmentalize the application, so that the failure of a safety measure compromises the least possible amount.
Where possible, access to crucial data should be dependent on more than one secret, utilizing different technology and / or living in a different application layer.

A great example of applying this design principle, is perfect forward secrecy, implemented for example in the OTR protocol, which assumes a shared secret or key to get compromised, so it generates new ones each session. Not using a static keypair and not reusing them, also gives the user deniability (i.e they can claim it was not their communication).

The next design principle we chose was:

Securing the weakest link

This is sort of self-explanatory. Often strong and virtually “unbreakable” crypto is simply bypassed by weaknesses in the implementation around it. To the extent that security researchers stress out, how as an attacker, often the strong encryption serves the sole purpose of informing you which part to concentrate on.
So a strong hashing algorithm is absolutely useless if the action a program takes after verifying the password can be triggered without ever supplying a password.
An experienced attacker, as well as exploitation kits can be relied on to sniff out the path of least resistance.

Another principle that should be followed regardless of the type of application is:

Failing securely

It is not only bad user experience if a user is confronted by an unanticipated error, it is also a like point for intrusion. Generally all unexpected behavior should be caught and “disarmed”.
In any case an application’s reaction to unexpected behavior should be the least possible privilege and access and the information given to a user about the cause should be carefully weighed.

In general while examining common attack vectors as to relevance to our project, it quickly emerged that our attention should be centered around the client-side, as the crucial processing of data all happens in the browser.

→ ***see riskanalysis.xls***

4. Application setup and architecture

We are using the Python micro-framework Flask on the backend, served by nginx via uWSGI. Our database is PostgreSQL.

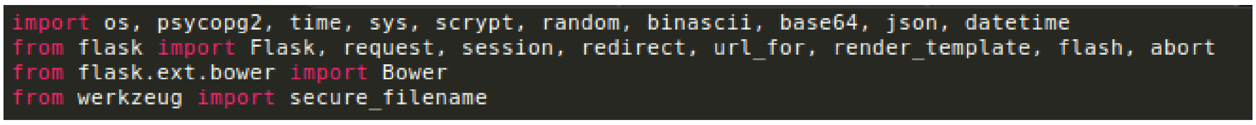
To manage the Javascript dependencies bower is our tool of choice, and the pip package manager for the Python modules. The required modules are saved to requirements.txt and can be installed via

```
~$ pip install requirements.txt
```

equivalent to bower’s

```
~$ bower install
```

The application is run from a virtual environment to which the dependencies are installed and environment variables set. The configuration object is an environment variable, so we can easily switch between development and production mode. They differ mostly in the debugging information being displayed. The upload directory path is also part of the environment as well as the database connection string.
It isolates the application from the rest of the system, as well as pointing only to the specified python version, in our case 3.4 (most systems need to have more than one python version installed). This keeps the application clean and manageable.



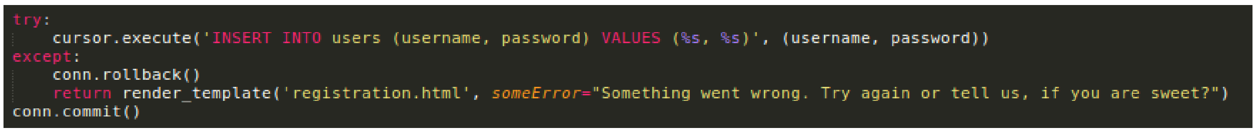
```
import os, psycopg2, time, sys, scrypt, random, binascii, base64, json, datetime
from flask import Flask, request, session, redirect, url_for, render_template, flash, abort
from flask.ext.bower import Bower
from werkzeug import secure_filename
```

Picture 4.1.: Our imports

We are making use of the jinja templating engine, that comes with Flask. Other than that most notably of the routing and session modules.
In accordance with the project requirements we omitted to use the authentication module.

4.1 The registration

The registration form requires only a username and a password. This leaves us open to account spam, but since throwaway email services like 10minutemail allow making multiple accounts with ease, even when more information is required, we do not see it as a big problem.
A possible solution to discourage users from filling up disk space, would be to limit account number per IP address over time (store the IP address in memory or a log file) and to detect suspicious registration patterns that indicate automation. In any case, data economy is our priority.
The input fields get validated server-side and the error messages get injected into the template via the jinja environment.
Psycopg2, the PostgreSQL adapter module, takes care of a lot of sanitization.
In Python (as in other languages), to avoid SQL injection, we need to take care that we do not concatenate input values with the SQL statement string.
By using the variable placeholder %s we ensure that the input is a string. The dangerous possibilities would be using Python string concatenation (+) or string parameters interpolation (%) to create the query. Psycopg2 converts python objects to SQL literals.
One safe way of passing parameters is:



```
try:
    cursor.execute('INSERT INTO users (username, password) VALUES (%s, %s)', (username, password))
except:
    conn.rollback()
    return render_template('registration.html', someError="Something went wrong. Try again or tell us, if you are sweet?")
conn.commit()
```

Picture 4.2.: A safe way of passing a variable to a SQL string in Python

An important lesson learned was, that for positional variables binding, the second argument must always be a sequence and since in python a single value tuple requires a comma, it would be passed like:

```
(variable, )
```

When we first put our application live, we did not have the surrounding try, catch statement in place. This lead to the application effectively crashing on a failed transaction. It turns out, that it needs to be rolled back explicitly as we now do via

```
conn.rollback()
```

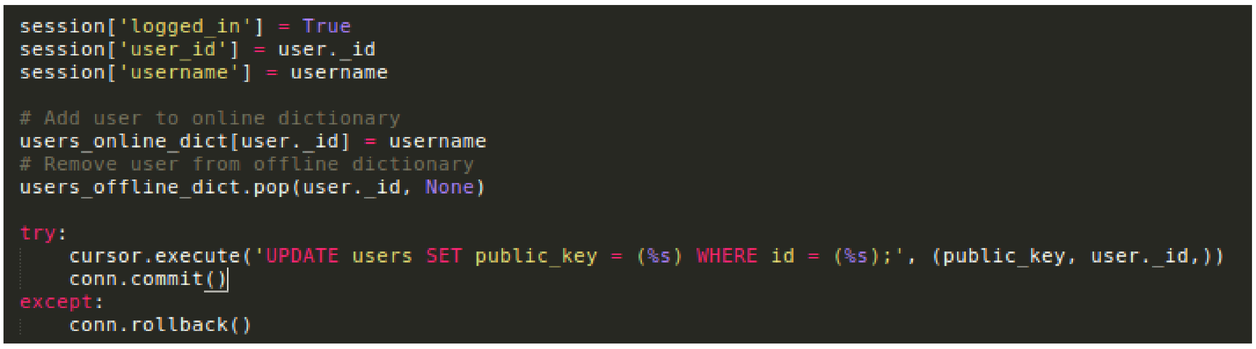
As a hashing algorithm for the password we decided on scrypt, a modern key derivation function, that could be considered a successor of bcrypt, which was formerly one of the algorithms heavily recommended in online forums.
Expressed simplified, scrypt is deliberately more memory expensive and takes longer to compute than bcrypt. This makes parallel dictionary attacks on scrypt essentially more costly.

4.2 The Login

The login makes use of a context User class, that contains methods to keep track of the login attempts per IP address and user.
On init the application loads all users as instantiations of that context class into memory.
To give a possible attacker no room to guess usernames we inject “Wrong username or password” as an error message on each failed attempt. The username the attempt was made on, if exists, gets stored in a dictionary alongside the IP address. 3 wrong password attempts on the same username, or 3 wrong attempts from the same IP address result in a 3 minute (in theory, we use 1 minute, because we are tired of locking ourselves out) ban from attempting. Upon successful login, or elapse of the ban, the IP (and possibly username) get removed from the dictionary.

When loading the login page an RSA 2048 bit key-pair gets created client-side. On successful login, the PEM formatted public key gets stored in the user table. This is important for user identity verification when messaging and used to encrypt the messages. More on that in chapter 4.4.

Username and userid get stored to a session variable and the user gets removed from an offline users dictionary and added to the online users dict respectively.



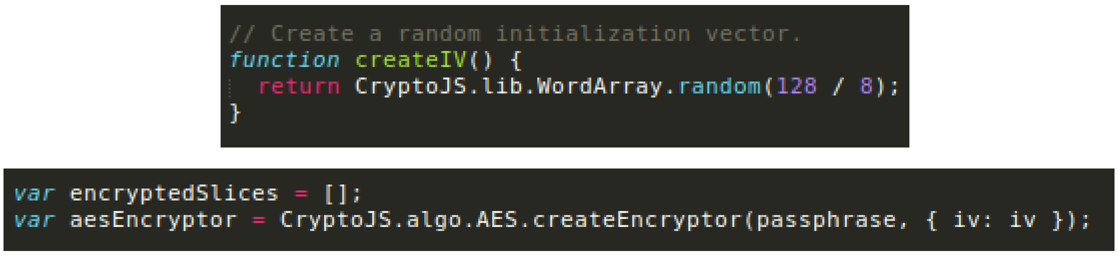
```
session['logged_in'] = True
session['user_id'] = user._id
session['username'] = username

# Add user to online dictionary
users_online_dict[user._id] = username
# Remove user from offline dictionary
users_offline_dict.pop(user._id, None)

try:
    cursor.execute('UPDATE users SET public_key = (%s) WHERE id = (%s);', (public_key, user._id,))
    conn.commit()
except:
    conn.rollback()
```

Picture 4.3.: The code handling the user login

4.3 The file upload

Files are sliced into chunks, which are then encrypted with help of the CryptoJS library.
CryptoJS supports AES-128, AES-192, and AES-256. When using a passphrase it automatically generates a 256 bit key.

We are using progressive ciphering which allows for bigger file-sizes, hence the slicing. Slicing is also necessary for the asynchronous file upload.
With progressive ciphering a key and initialization vector (IV) are used to create an instance of the AES file encryptor. The IV and passphrase are needed for decryption.

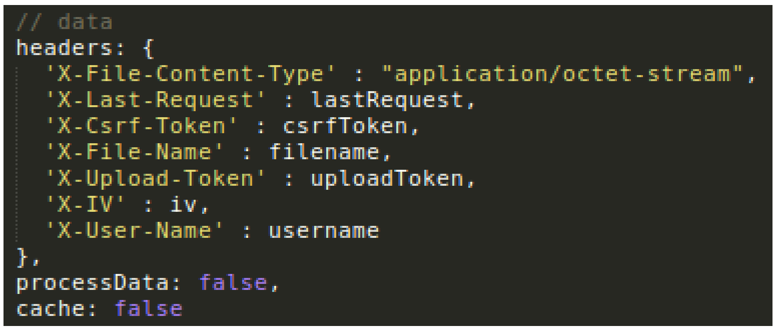


```
// Create a random initialization vector.
function createIV() {
  return CryptoJS.lib.WordArray.random(128 / 8);
}
```

```
var encryptedSlices = [];
var aesEncryptor = CryptoJS.algo.AES.createEncryptor(passphrase, { iv: iv });
```

Picture 4.4.: Creating the initialization vector and instantiating the AES file encryptor

When the file gets submitted a unique upload token to identify the file gets created. The IV and upload token are saved as global variables on the client.
We send an XHR to the server with the upload form’s csrf token, uploadToken, filename, IV, optional username and optional deletion password. There is also a “last request” parameter set to false, indicating, that this is one of multiple requests using this csrf token. This is needed to remove the token from the session on the last (or only) request.



```
// data
headers: {
  'X-File-Content-Type' : "application/octet-stream",
  'X-Last-Request' : lastRequest,
  'X-Csrf-Token' : csrfToken,
  'X-File-Name' : filename,
  'X-Upload-Token' : uploadToken,
  'X-IV' : iv,
  'X-User-Name' : username
},
processData: false,
cache: false
```

Picture 4.5 The XHR headers for announcing a following file upload

Based on the request not containing any chunks of the file the server concatenates the filename (sanitized by the imported secure_filename utility), with a current timestamp, inserts IV, IP address (of the uploading user), `<current timestamp>_<original filename>` and username into the files table.

The database returns the record id of the file (auto incrementing primary key) and the filename gets assigned to:

```
<fileid>_<current timestamp>_<original filename>
```

A session variable gets set with:

```
upload_token -> filename
```

On success the server returns the upload_token to the client. If this matches the client begins slices the file into binary chunks of maximum 65536 byte length. We learned the hard way, that the byte length has to be divisible by 16 (otherwise the file gets corrupted during encryption).
Then the passphrase from the form (by default generated by us, optionally chosen by the user) gets retrieved and and the AES encryptor gets instantiated.
We create an array of file readers equal to the length of the slice array (reusing the same file reader creates complication due to the Filereader API being asynchronous).
The Filereader reads the slices into array buffers, from which we create first a Uint8Array (JavaScript typed array of 8-bit unsigned integers) and then a wordarray (CryptoJS array of 32-bit words). This step was getting our file corrupted, when we still used an arbitrary binary chunksize. It worked with files smaller than our max byte length, which finally gave the clue.
The wordarray gets encrypted, converted to Base64 and pushed to an encrypted slices array.
If on the last Filereader's

```
onloadend
```

event the encrypted slices array is of equal length as the slices array, the finalizer method gets called on the encryptor which adds one more slice.
Now we iterate over the encrypted slices array and send each to the server via xhr, sending the length of the array as parameter alongside the current chunk number, the upload and csrf token and the last request "boolean".

On the server side on first request, a directory with the filename (stored in the session with the upload token as key) gets created inside our application's upload directory.
In this directory each chunk gets stored with its chunk number as name.

On the last chunk csrf and upload token get removed from the session and a success response sent to the client.
Here the user gets a chance to save their download link (containing the file id), optional deletion link and their passphrase.

4.4 The file download

When visiting the download page (file id is a url parameter), an XHR asks the server for the number of chunks, the iv and the filename (as in the database, meaning timestamp and old filename).
Upon success the client issues an XHR per chunk number (which is also the chunk name).
The server sends a json string, containing the chunk number and the Base64 encoded chunk.
The client success handler sets the downloaded chunk in an array at the index of its chunk number.
Failing to do it like this at first and just pushing them as they came, caused us quite another headache.
Again due to the asynchronous nature of the process, with bigger files, the chunks did not necessary arrive in the correct sequence, so the majority of them was corrupted.
Upon the last downloaded chunk the user can enter their passphrase and decrypt the file.
An aes decryptor gets created with the IV and passphrase and each chunk gets parsed and decrypted and pushed to a decrypted chunk array (note, that this is a synchronous process, so we do need to ensure the sequence any longer).
When the last chunk is decrypted, the finalizer gets called on the decryptor and pushed to the array. We extract the old filename from the current file address and give the user the option to rename it before download.
When the user downloads the file the decrypted chunks get converted back to a uint8 array and pushed to a containing array. This array of uint8 arrays gets then converted to a blob (binary large object) from which we can create an object url.
With HTML 5 specifications we can user an href specifying the filename in the download attribute and programmatically click it, so the download gets started as soon as it is ready.

4.5 File deletion

The file deletion is relatively straightforward. The user browses to their deletion link, that contains the fileid as a variable. Using the Werkzeug Routing built in to Flask, we can declare the type of the variable like:

```
<int:fileid>
```

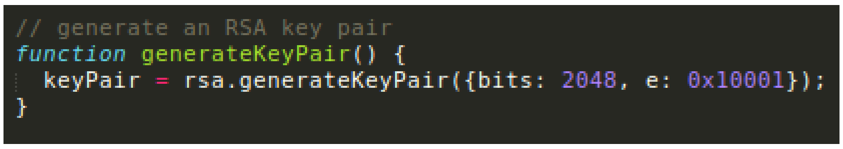
Wekzeug takes care of the further sanitization. They then get prompted to enter their passphrase (which is also hashed with scrypt) and if it verifies, both the database record and the directory containing the file get deleted.

4.5 The messaging

For the messaging we decided to make use the WebRTC API which allows for browser-based real time communication between clients, including audio, video and text. It is one of the more recent technologies in the W3C specifications and still a work in progress. It is implemented in the major recent browsers, but there are some impediments. It enforces the usage of encryption on both sides. TorrentFreak reported in January that browsers supporting it, are compromising the security of VPN-Tunnels, by allowing the true IP addresses of users to be read.

We are making use of the PeerJS library a wrapper for the browser's implementation to provide a complete, easy to use API.
During development we were also using their PeerServer, but since it does not provide TLS, we took to hosting our own PeerServer (NodeJS). It needs to run on the same virtual machine as our application server, since utilizing the same self-signed certificates as our application server, the user does not have to add it separately to their chain (otherwise they'd need to browse to the URL of the peer server and manually accept it, unless it was issued by a CA. The peer server acts purely as connection broker, meaning the data does not go through the server, it only negotiates the connection.
We also make use of a google STUN server, to avoid problems when connecting to a peer behind a NAT.

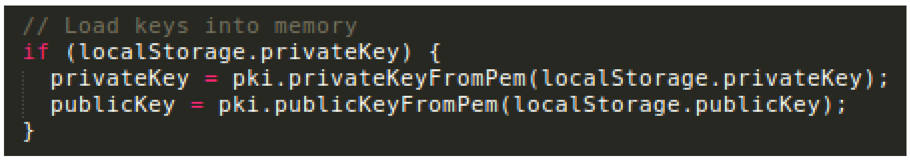
As mentioned earlier, upon login a 2048 bit RSA key pair gets created, with help of the Forge JS crypto library and the public key gets saved to the database in PEM format.
When choosing CryptoJS for the symmetric encryption, we were not yet aware of the need of asymmetric encryption for the messaging, as we were planning to use an OTR library. However this turned out to be too complicated and not really necessary, since we need no continuous channel between the 2 parties of a communication. The messages are supposed to self-destruct after being read by the user, so a shared secret for symmetric encryption, as it gets used by OTR is not really necessary.



```
// generate an RSA key pair
function generateKeyPair() {
  keyPair = rsa.generateKeyPair({bits: 2048, e: 0x10001});
}
```

Picture 4.6.: A safe way of passing a variable to a SQL string in Python

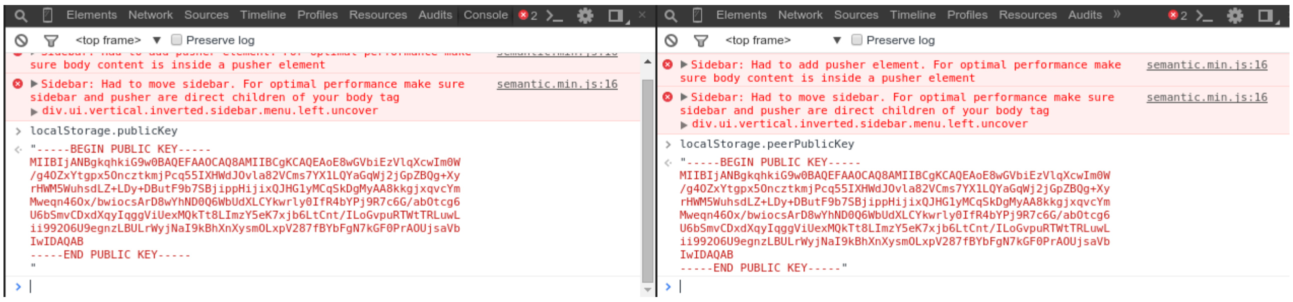
However, even apart from encrypting, we need a key pair to verify the identity of a user.
After the key generation both the public and the private key get stored in the local storage in PEM format and loaded into memory and converted from PEM to a forge.pki object on each page refresh.



```
// Load keys into memory
if (localStorage.privateKey) {
  privateKey = pki.privateKeyFromPem(localStorage.privateKey);
  publicKey = pki.publicKeyFromPem(localStorage.publicKey);
}
```

Picture 4.7: Loading the keys into memory on page refresh

Only the public key gets stored in the database.



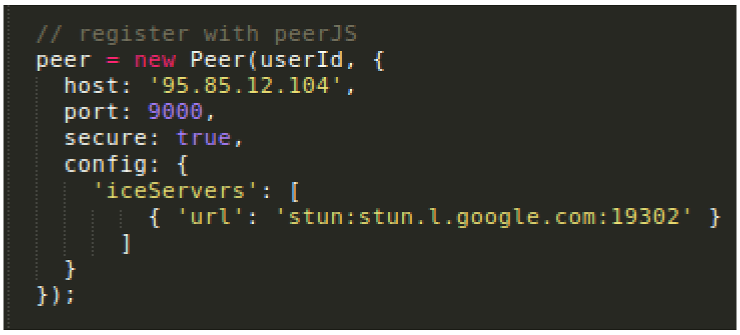
```
sure body content is inside a pusher element
▶Sidebar: Had to move sidebar. For optimal performance make sure     semantic.min.js:16
sidebar and pusher are direct children of your body tag
▶div.ui.vertical.inverted.sidebar.menu.left.uncover
> localStorage.publicKey
< "-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAoE8wGVbiEzVlqXcwIm0W
/g4OZxYtgpx5OncztkmjPcq55IXHWdJOvla82VCms7YX1LQYaGqWj2jGpZBQg+Xy
rHWM5WuhsdLZ+LDy+DButF9b7SBjippHijixQJHG1yMCqSkDgMyAA8kkgjxqvcYm
Mweqn46Ox/bwiocsArD8wYhND0Q6WbUdXLCYkwrly0IfR4bYPj9R7c6G/abOtcg6
U6bSmvCDxdXqyIqggViUexMQkTt8LImzY5eK7xjb6LtCnt/ILoGvpuRTWtTRLuwL
ii992O6U9egnzLBULrWyjNaI9kBhXnXysmOLxpV287fBYbFgN7kGF0PrAOUjsaVb
IwIDAQAB
-----END PUBLIC KEY-----
"
>
```

```
▶Sidebar: Had to add pusher element. For optimal performance make     semantic.min.js:16
sure body content is inside a pusher element
▶Sidebar: Had to move sidebar. For optimal performance make sure      semantic.min.js:16
sidebar and pusher are direct children of your body tag
▶div.ui.vertical.inverted.sidebar.menu.left.uncover
> localStorage.peerPublicKey
< "-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAoE8wGVbiEzVlqXcwIm0W
/g4OZxYtgpx5OncztkmjPcq55IXHWdJOvla82VCms7YX1LQYaGqWj2jGpZBQg+Xy
rHWM5WuhsdLZ+LDy+DButF9b7SBjippHijixQJHG1yMCqSkDgMyAA8kkgjxqvcYm
Mweqn46Ox/bwiocsArD8wYhND0Q6WbUdXLCYkwrly0IfR4bYPj9R7c6G/abOtcg6
U6bSmvCDxdXqyIqggViUexMQkTt8LImzY5eK7xjb6LtCnt/ILoGvpuRTWtTRLuwL
ii992O6U9egnzLBULrWyjNaI9kBhXnXysmOLxpV287fBYbFgN7kGF0PrAOUjsaVb
IwIDAQAB
-----END PUBLIC KEY-----"
>
```

Picture 4.8.: A user's public key in their own and their peer's localStorage

In the current development state of our project a peer’s public key is requested from the server on each outgoing or incoming message.

A logged in user also gets registered on the PeerServer, using their userid (retrieved from the template). The “secure” boolean indicates TLS.

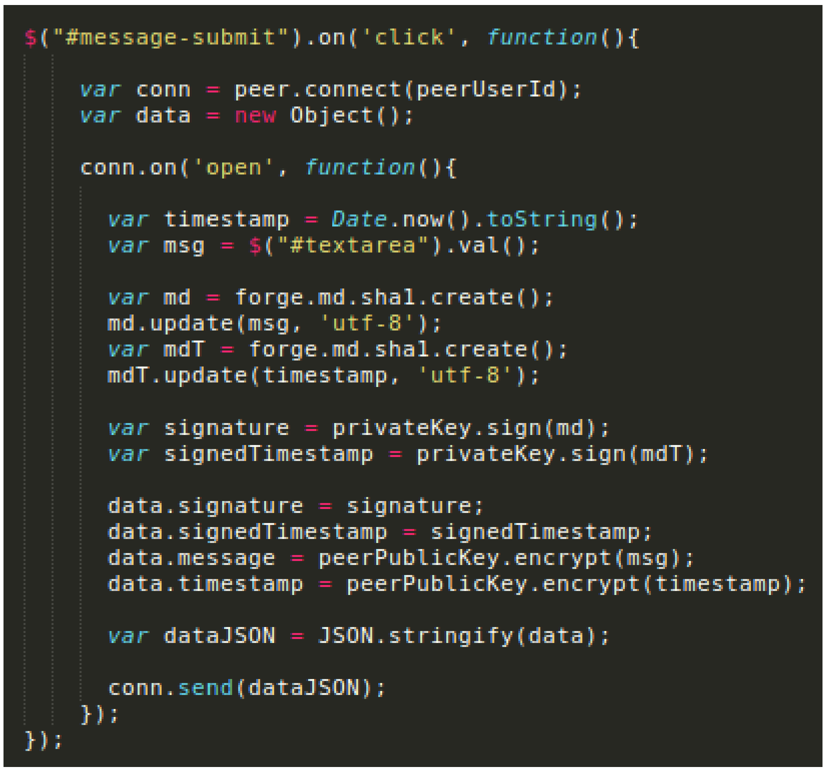


```
// register with peerJS
peer = new Peer(userId, {
  host: '95.85.12.104',
  port: 9000,
  secure: true,
  config: {
    'iceServers': [
        { 'url': 'stun:stun.l.google.com:19302' }
      ]
  }
});
```

Picture 4.9.: Registering a user via PeerJS

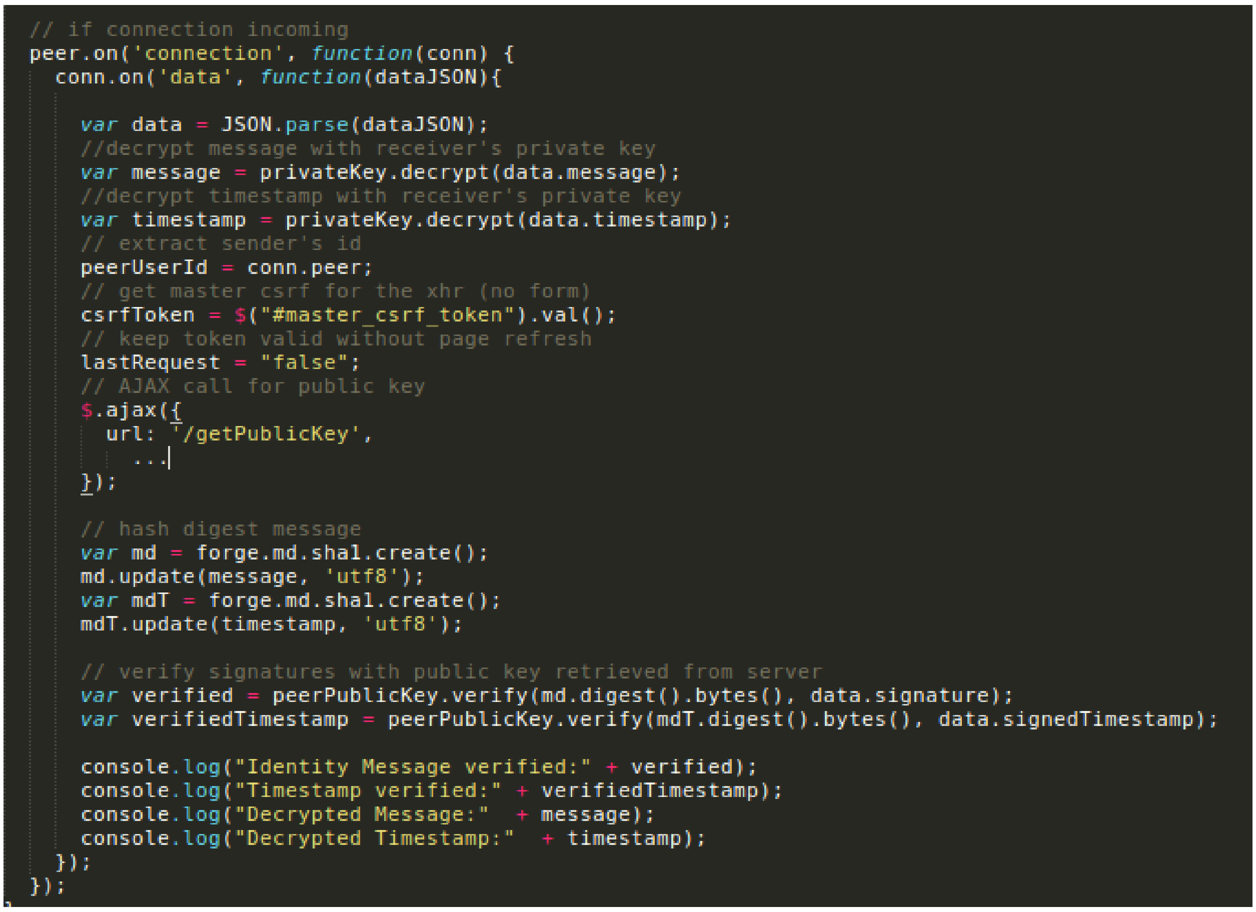
It is easy to see how someone could impersonate another user, by simply registering with someone else’s user id.

So if a user is requesting a connection to another peer (by clicking an online user’s name in the browser) and XHR is made to the server, that requests this user’s public key.
A message signed with a private key, can be verified with the respective public key.
The message to be send and a timestamp are signed with the sender’s private key and then encrypted to the receiver’s public key. (*Picture 4.10)*



```
$("#message-submit").on('click', function(){

    var conn = peer.connect(peerUserId);
    var data = new Object();

    conn.on('open', function(){

      var timestamp = Date.now().toString();
      var msg = $("#textarea").val();

      var md = forge.md.sha1.create();
      md.update(msg, 'utf-8');
      var mdT = forge.md.sha1.create();
      mdT.update(timestamp, 'utf-8');

      var signature = privateKey.sign(md);
      var signedTimestamp = privateKey.sign(mdT);

      data.signature = signature;
      data.signedTimestamp = signedTimestamp;
      data.message = peerPublicKey.encrypt(msg);
      data.timestamp = peerPublicKey.encrypt(timestamp);

      var dataJSON = JSON.stringify(data);

      conn.send(dataJSON);
    });
});
```

Picture 4.10.: Signing the message and timestamp with the sender's private key encrypting it with the receiver's public key



```
// if connection incoming
peer.on('connection', function(conn) {
  conn.on('data', function(dataJSON){

    var data = JSON.parse(dataJSON);
    //decrypt message with receiver's private key
    var message = privateKey.decrypt(data.message);
    //decrypt timestamp with receiver's private key
    var timestamp = privateKey.decrypt(data.timestamp);
    // extract sender's id
    peerUserId = conn.peer;
    // get master csrf for the xhr (no form)
    csrfToken = $("#master_csrf_token").val();
    // keep token valid without page refresh
    lastRequest = "false";
    // AJAX call for public key
    $.ajax({
      url: '/getPublicKey',
        ...
    });

    // hash digest message
    var md = forge.md.sha1.create();
    md.update(message, 'utf8');
    var mdT = forge.md.sha1.create();
    mdT.update(timestamp, 'utf8');

    // verify signatures with public key retrieved from server
    var verified = peerPublicKey.verify(md.digest().bytes(), data.signature);
    var verifiedTimestamp = peerPublicKey.verify(mdT.digest().bytes(), data.signedTimestamp);

    console.log("Identity Message verified:" + verified);
    console.log("Timestamp verified:" + verifiedTimestamp);
    console.log("Decrypted Message:"  + message);
    console.log("Decrypted Timestamp:"  + timestamp);
  });
});
```

Picture 4.11.: Receiving a message, decrypting it and verifying the signature

The receiver requests the sender's public key upon incoming connection (*Picture 4.11).*
They then verify the signatures and decrypt the message and timestamp with their own private key.

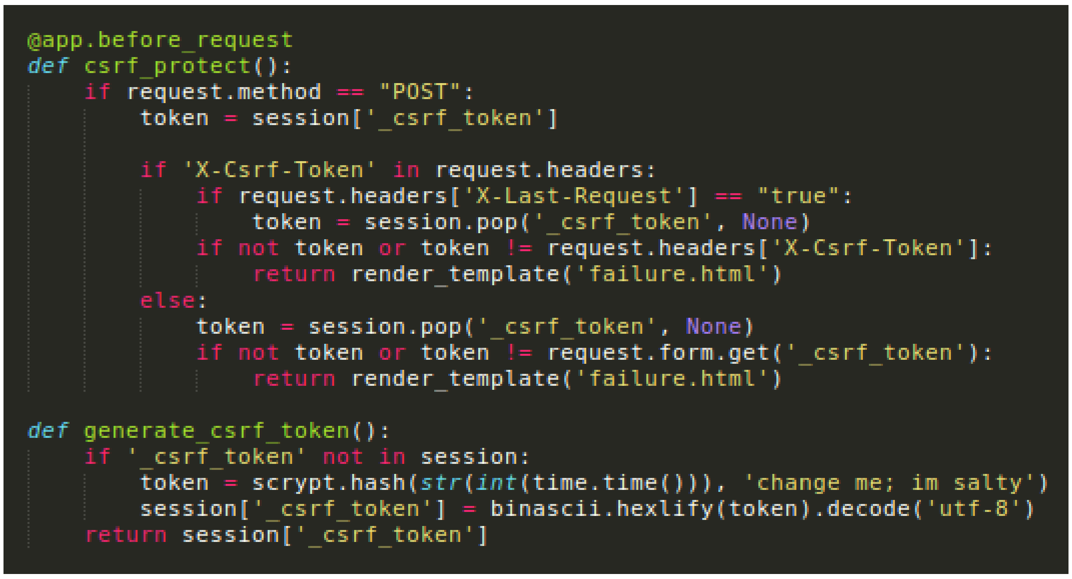
The timestamp is then used to verify that the message is recent to avoid session replay attacks. After a user dismisses a message, it is destroyed.

4.6 CSRF Protection

Where cross site scripting protection measures serve to protect the user from having his or her trust in the server exploited by malicious third parties, CSRF (cross site request forgery) protection ensures that the server's trust in the user and the integrity of the user's connection are not exploited by malicious users sending an unexpected payload along with users' requests.

Protecting against this is fairly simple and adds an additional element of protection to the server, particularly when dealing with form data. In practice, what this involves for our case is to simply add a unique hash passed to the client per the page load and - upon form submission - ensuring that this matches, providing a one-to-one relation between CSRF protection tokens (the hashes sent to clients) and the users' requests.

This - known as per-request CSRF protection - eventually will become a problem on large, AJAX-heavy sites, which is where the per-session CSRF protection comes into play.



```python
@app.before_request
def csrf_protect():
    if request.method == "POST":
        token = session['_csrf_token']

        if 'X-Csrf-Token' in request.headers:
            if request.headers['X-Last-Request'] == "true":
                token = session.pop('_csrf_token', None)
            if not token or token != request.headers['X-Csrf-Token']:
                return render_template('failure.html')
        else:
            token = session.pop('_csrf_token', None)
            if not token or token != request.form.get('_csrf_token'):
                return render_template('failure.html')

def generate_csrf_token():
    if '_csrf_token' not in session:
        token = scrypt.hash(str(int(time.time())), 'change me; im salty')
        session['_csrf_token'] = binascii.hexlify(token).decode('utf-8')
    return session['_csrf_token']
```

Picture 4.12.: Generating and validating CSRF tokens

As per the code shown above, our implementation of the CSRF token has two implementations (the if-else block). The former relying on two content headers, identifying the payload as a token, then contiguously passing the token to the server along with each AJAX call, providing a per-session CSRF token. The latter of the two way of doing it is the simpler, more traditional per-request token, which simply relies on a token, which is generated on page load - more specifically as a function call when the page template is rendered - then verifying said token against the user's session state, thereby verifying the integrity of the form request.

5. Server, Proxy, And Database Configuration

5.1 The technologies

The entrypoint for all requests to the system is an Nginx proxy, on which all traffic on ports 443 and 80, which is to say HTTPS and HTTP respectively, is received.

While it can hypothetically be used as a load balancer when scaling up the application, the primary purpose of this proxy server for the Crocofile application is as a way to terminate TLS and ensure that any plain HTTP requests are appropriately redirected to the the respective port, i.e. 443.

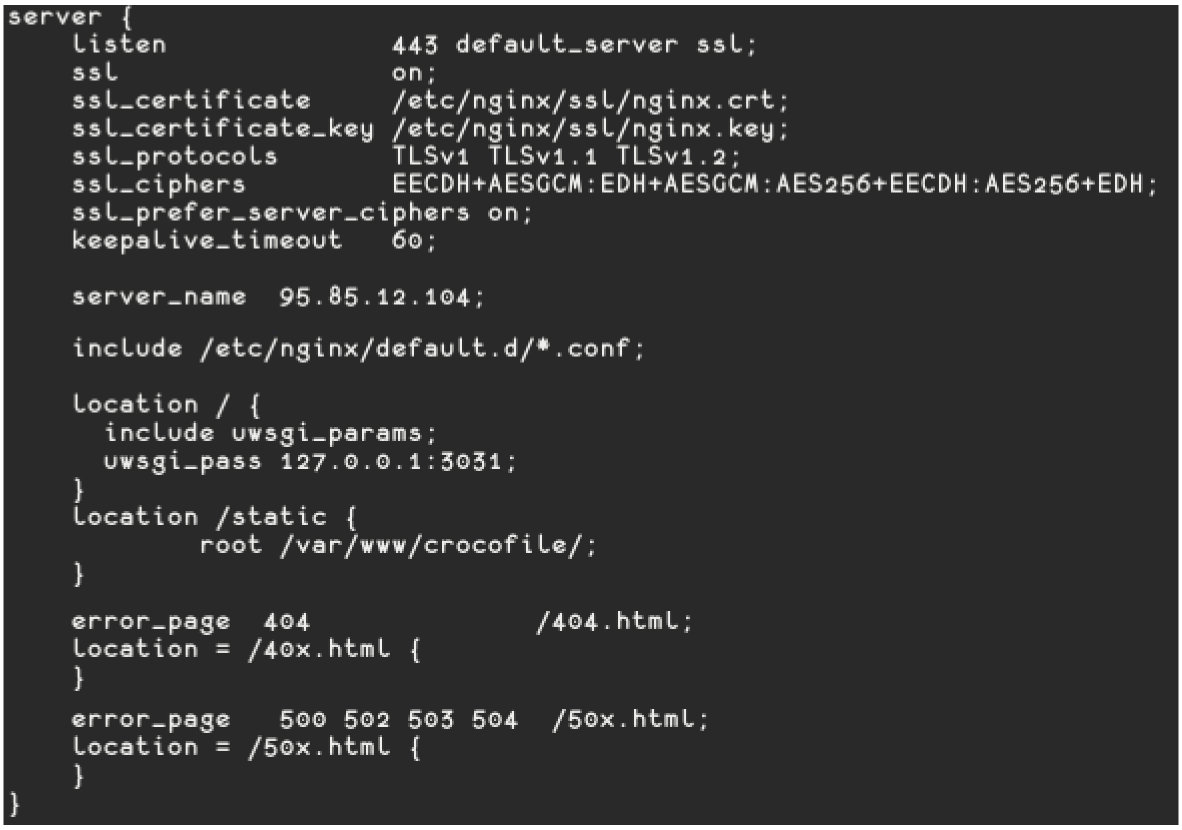
From the proxy server, requests are directed to the application server on port 3031, which accepts requests only from localhost, i.e. is inaccessible to the outside world.
The application server used for this is uWSGI, an implementation of the WSGI (Web Server Gateway Interface) specification. Aside from an application server being imperative, as the alternative is a fragile, development-only HTTP server, uWSGI provides a thread safe way of running the application concurrently across multiple processes.

The application itself consists of the microframework, Flask, which as opposed to larger frameworks like Ruby on Rails, primarily is concerned with mapping routes, i.e. URLs to controller logic. Meaning that the application structure is - rather than the macroframework tradition of Model-View-Controller - purely views and controllers, providing a clearer distinction of database and application concerns.

The database used is PostgreSQL. This was chosen over other relational database servers, in part due to its strict adherence to the SQL standard. Furthermore, PostgreSQL has several other advantages over MySQL, the other widely available open source RDBMS. Namely that PostgreSQL implements a ROLES system for managing user privileges as well as more options for using stored procedures than any common alternative. Lastly, PostgreSQL is highly ACIDic, focusing strongly on the integrity of writes as well as the durability of the data stored. It is widely lauded as the most security oriented database server, e.g. the recommendation presented in The Database Hackers Handbook (*Wiley,* 2005*)*:
"By default, PostgreSQL is probably the most security-aware database available.".

5.2 Security considerations with server configuration

As mentioned http connections get terminated and redirected to https by nginx. With the SSL (TLS) configuration we paid special attention to the cipher suites we want to allow.



```
server {
    listen              443 default_server ssl;
    ssl                 on;
    ssl_certificate     /etc/nginx/ssl/nginx.crt;
    ssl_certificate_key /etc/nginx/ssl/nginx.key;
    ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers         EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH;
    ssl_prefer_server_ciphers on;
    keepalive_timeout   60;

    server_name  95.85.12.104;

    include /etc/nginx/default.d/*.conf;

    location / {
      include uwsgi_params;
      uwsgi_pass 127.0.0.1:3031;
    }
    location /static {
            root /var/www/crocofile/;
    }

    error_page  404              /404.html;
    location = /40x.html {
    }

    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
    }
}
```

Picture 5.1.:   /etc/nginx/nginx.conf

As per the section of our nginx.conf file shown above, our SSL configuration accepts only a narrow subset of the recommended cipher suite when targeting modern browsers, recommended by multiple major open source projects, among which Mozilla. This subset was chosen due purely to the strength of the encryption used as backwards compatibility is not an issue for our use case. This is also the reason we opted to force Transport Layer Security, rather than supporting older, less secure SSL versions. As a natural consequence of these considerations, our proxy server configuration also specifies for connections to prefer server ciphers, i.e. to abide by the cipher suite outlined here. Additional security measures like HSTS and OCSP Stapling are not possible with self-signed certificates.

Moving on to the location blocks of the proxy server configuration, the primary block, i.e. what maps to the root path of the server points directly to port 3031. However, as it uses the 127.0.0.1 address and a port which is not allowing external traffic, all communication between the nginx proxy and the application server happens over the loopback networking interface, isolating the application server from outside attackers, in compliance with both the second and third of our design principles in relation to security. This is to say that in doing this, we are simultaneously securing the fragile core - meaning the application and app server - of our application by way of isolation, and at the same time we are adding an increased level of security in the event of failures. This is to say that should any unrecoverable error occur on the application server, no debug information or unintended output is leaked through the reverse proxy, and the application will simply be unreachable or - if nginx is provided with a static file for its 500 route - yield an “internal server error” response to the client.

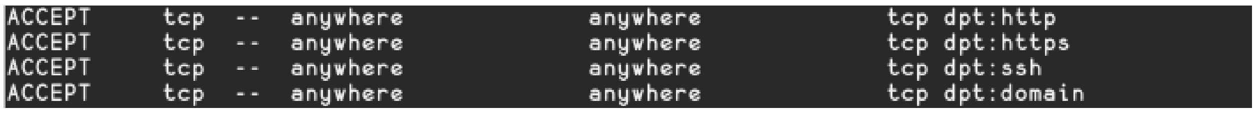
5.3 Server host configuration

In addition to the security measures applying directly to the proxy- and application server settings, a number of tweaks to the server host have been made to ensure that not just the web application and web server are kept safe, but also that potential holes in the system at large are removed or minimized.

The first place to focus our attention was the necessary, open port 22, which is to say the port allowing us to access our server over SSH. Securing this was a matter of disabling password-based login and instead relying on our own, whitelisted public RSA keys, which - as per good security practice - are individually secured with a passphrase. This adds an additional level of protection to this exchange, as the SSH user has to provide both something he or she know as well as something he or she has (the key). Naturally, relying on key exchange over password entry takes issues like bruteforce attacks out of the equation entirely.

Another element of SSH security was to disable remote root logins. This simply minimizes the risk of privilege escalation attacks, by reducing the space of users to three, pre-approved users.

The next step in securing the host is filtering ports, i.e. firewall features. This has been done through iptables, the traditional Linux feature for allowing or disallowing traffic on certain ports.



```
ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:http
ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:https
ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:ssh
ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:domain
```

As per the image above, which is the result of running iptables -L to list the open ports, our production system exposes - as is to be expected - an http port purely for sending a redirect response to the https port, an https port, the aforementioned port 22 for SSH, and lastly a port of a DNS server in anticipation of the server needing domain services. Had this not been almost universally necessary, it would’ve also been wise to close this port.

In addition to the ports mentioned above, HTTPS traffic is allowed on the port, 9000, on which our system is running a Node.js server, which works as a connection broker for our peer-to-peer user chat system over WebRTC. This is of course also necessary and is under the same restrictions as our default port, 443.

6. UX considerations with respects to security

Regardless of the type of project you are working on, a well-designed application should implement a well-balanced combination of security, aesthetics, usability and so on. Although UX and security often clash with each other, we as developers have to learn to work together with the designers to ensure that the end-user is not affected by the security and design decisions conflicting during development.

To use an analogy, the level of security applied in an application is similar to the amount of salt we pour onto a plate of our favorite dish: it has to be just enough to make the food flavorful. If we add too much or too little, then our taste buds are unhappy.

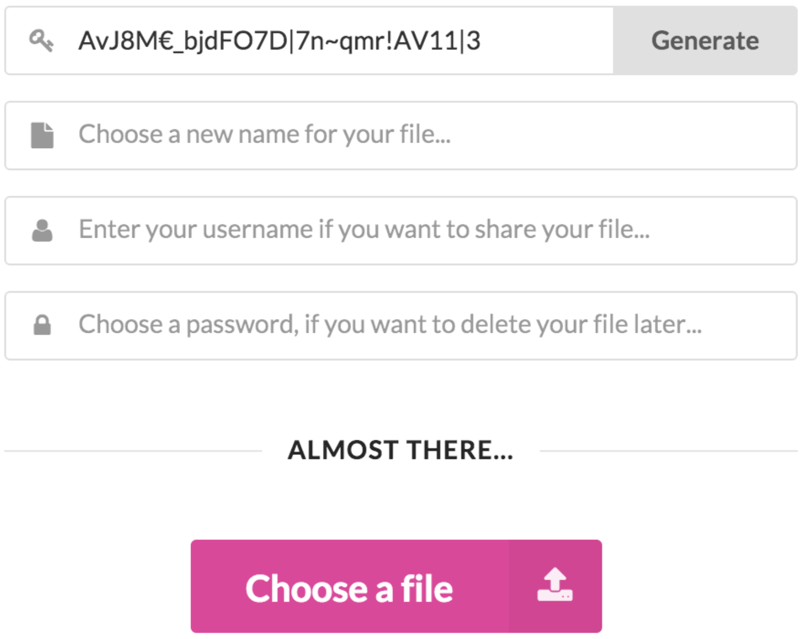
So far we've established that, while security is of utmost importance, aesthetics and usability must not be left out of the equation, if the purpose of the application is not to serve only experienced and tech-savvy people (not that some of them wouldn't enjoy a well-designed and easy-to-use application). In the following paragraphs we'll have a look at how our application has grown from a rather straightforward idea, to something more complex that needed the touch of a designer to turn it into something usable and enjoyable.

In the very beginning of the project, we didn't know precisely how things would come together in the end. We had an idea of what we wanted to make, but if you asked us to draw it down into a wireframe, nothing pretty would've come out. We've learned along the way, with every little thing implemented, how we should tie everything up together.

The premise is to keep everything as simple and down-to-earth as possible. One of the most important design principles is in fact making an application or website simple. Applications often try to do more than they are supposed to. We hate that, so we kept it at a minimum.
First of all, you don't need an account to use our application. Why would you? The whole purpose is to give the user the possibility to share their files in a secure manner, without too much trouble.

If, however, you want an account, it takes under 30 seconds to create one. We don't ask you for your name, birthday and so on. We don't need all that stuff. We only need a username and password. It is seamless and straightforward, and enforces the security indirectly, by not providing us with sensitive data that we have to add extra protection for.

Second of all, we never ask of you to do more than you need to. Uploading a file for secure sharing is as easy as pressing a button and choosing a file. We don't require you to do anything else, but we give you the possibility to:

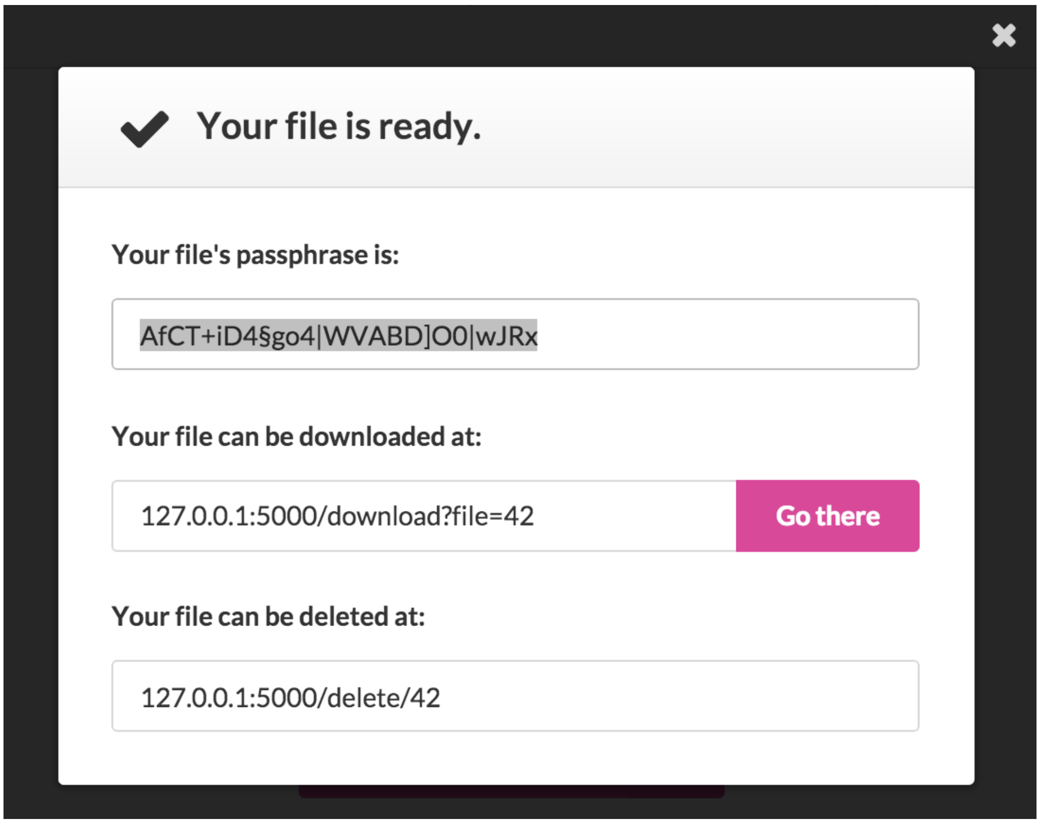

AvJ8M€_bjdFO7D|7n~qmr!AV11|3
Generate
Choose a new name for your file...
Enter your username if you want to share your file...
Choose a password, if you want to delete your file later...
ALMOST THERE...
Choose a file

- Choose your own encryption passphrase. Maybe you don't want to write down a passphrase for every file you upload, then choose one you will remember.
- Choose a different name for your file. It doesn't have to be called what it is on your computer. Call it whatever you want, if you want.
- Enter your username if you want your file to be publicly shareable (meaning other users can ask you for the password). You don't have to enter anything, if you want to share your file with only people you know.
- Enter a deletion password, if you want to destroy your file later, permanently.

In essence, you need to be in control, but now because we want you to, rather because your want to be in control. It's your choice.

Uploading a file and sharing it is as simple as it can get. Once you've chosen your file, it will get encrypted and uploaded in the background. Once that was done, we provide you with your chosen passphrase, your download link and your deletion link (if you provided a deletion password).

Small things make the application delightful use, such as automatically selecting the passphrase value when it is given to you, so you can just press CMD/CTRL + C on your keyboard to copy it. Or giving you a button to follow the download link, together with the actual link that you can copy and share with others. We make it easy for you to quickly grab the necessary things and hand them over to the persons you want to share your file with.



Your file is ready.

Your file's passphrase is:

AfCT+iD4§go4|WVABD]O0|wJRx

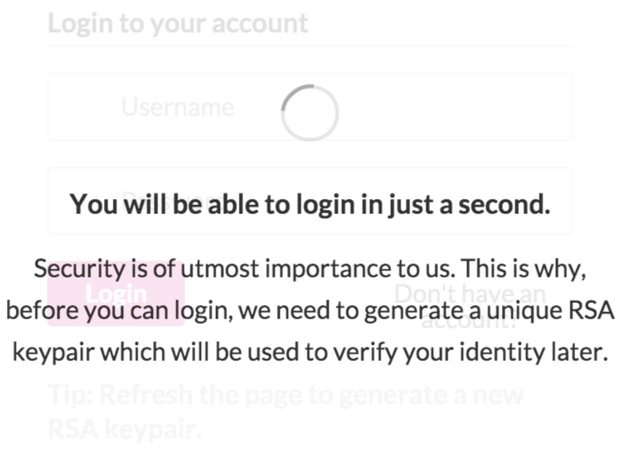
Your file can be downloaded at:

127.0.0.1:5000/download?file=42

Your file can be deleted at:

127.0.0.1:5000/delete/42

Besides making it easy for you to use the application, we also make sure we inform about everything that is going on. Being a security-driven application, it is important for us that you know what we are doing in the background. Transparency is extremely important, that is why we do not keep a secret our choice of encryption algorithms and so on.



You will be able to login in just a second.

Security is of utmost importance to us. This is why, before you can login, we need to generate a unique RSA keypair which will be used to verify your identity later.

Tip: Refresh the page to generate a new RSA keypair.

Overall throughout the application we are trying to exhibit craftsmanship, which is again another design principle. We attempt to finish the product to the every little detail, because when users sense that something is well crafted, they're compelled to trust the people who made it. And what better feeling is there than knowing that you built a secure file-sharing service that your users trust to the very last little feature.

7. Conclusion and Prospects

We have come a long way from our initial rather vague plan. We got to experiment with time-proven symmetric and assymetric cryptography, a modern hashing algorithm, TLS configuration, new W3C specifications and both of the biggest JavaScript crypto libraries. We learned more about keeping a host system (and our own machines) secure, server configuration in general and especially with respects to security. Trade-offs between usability and security have been evaluated. And we have been impressed with the default security mesures of the slim Flask framework. There was fairly little we could and had to do to prevent the XSS which in our planning phase we were the most worried about. The werkzeug routing that Flask uses sanitizes url variables and parameters.

Also the PostgreSQL defaults and the psycopg2 adapter impressed us.

There is still a lot to be improved, but as a proof of concept it holds. We learned along the way and some best practices we adapted in our later code, should also be adapted in the early lines. It is probably very common that a project at the stage we are in now, where the structure cristalizes and the chosen technologies have been proved, needs some serious refafctoring.

The intense computing tasks, such as generating a keypair, de- and encryption make the application rather slow at times and especially the former sometimes results in a crashed script or unresponsive browser. This could be countered by utalizing webworkers. Futhermore the encryption during messaging could be improved, to provide forward secrecy and deniability. We also did not actively take control of the session handling and right now closing the browser without logging out, clutters our memory with orphaned session variables.

As for UX there is, as of writing, no feedback on verification errors and the timestamp being recent is not checked.

On the serverside we generated a large dhparam to speed up the TLS keyexchange and prevent the OpenSSL default, which we are currently not including.

Overall it was a thrilling project and we are all sure to have learned a great deal and are eager to continue learning about the fields we have only just entered.

References

Projects we looked at for inspiration and clarification:

https://github.com/hushfile

https://github.com/STRML/securesha.re-client/tree/master/

https://github.com/OTRMan/

The libraries our crypto libs are based on and source of of valuable information:

http://www-cs-students.stanford.edu/~tjw/jsbn/

Recommended ciphers and TLS config:

https://wiki.mozilla.org/Security/Server_Side_TLS

SSL configuration tests (that work without domain):

https://www.digicert.com/help/

https://ssldecoder.org/