CSU34031 Project 2 Report
Network Security Application

Rafael Mendes
Student No: 17330951
mendesr@tcd.ie

April 14, 2020

Contents

1 Introduction

The goal of this project was to develop some kind of web-based application to secure an aspect of social media e.g. Facebook Wall, Facebook Messenger, Twitter, etc. By securing we mean that if a user decided to send some message using any of the many methods available, it would actually be encrypted and only a select amount of users (chosen by the original sender) would be able to decrypt it. As a result to allow for any kind of encryption and decryption to be carried out, a key management system would need to be put in place. Furthermore a user of this application would need to have the ability to add and remove people from their "trusted group". Instead of building something directly on top of one of these apps such as Facebook or Twitter, I instead decided to implement my own application with its own user base. Once some piece of text is encrypted it can then be copied and sent to its appropriate destination through any of the methods previously listed. The receiver would then simply log in to my application to decrypt the cipher text which they have received.

A demo for this application can be found here:
`https://www.youtube.com/watch?v=bqZkkmr_fEw`

GitHub Repo:
`https://github.com/MendesRafa/Network-Security-App`

2 Architecture

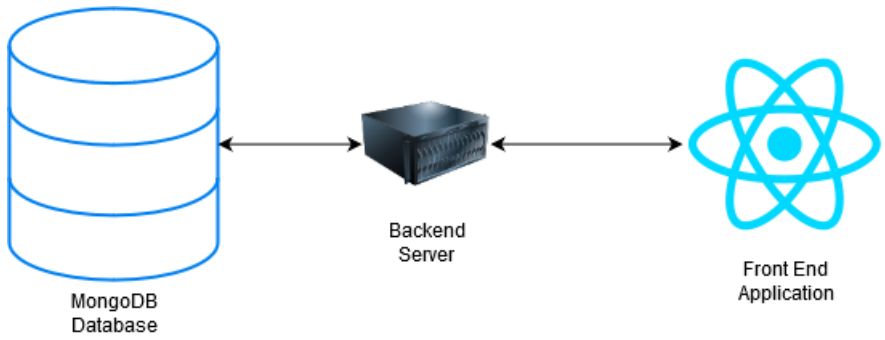


Figure 1: Application Architecture

My application can be roughly divided into three sections as shown in the above figure. The database which was set up using MongoDB is what holds the information of every user, this information being their username, password, group of trusted users, and both their private and public key. The back end server which was deployed locally using NodeJS and the Express module, handles requests from the front end and communicates with the database. The front end application created using the React web framework acts

as way to receive input from the user, it then forwards any requests for sending or receiving data to the back end server.

2.1 Back End

As mentioned above our back end is comprised of a locally run NodeJS Express server listening in on port 3001 and the database it communicates with. When the server is first started up, it simply establishes a connection with our database and begins to listen for requests on the designated port. Overall this server handles 6 different request coming from our React Application. These are requests for: registering an account, logging into an account, getting a list of registered users, updating a user's trusted group, encrypting a message and decrypting a message.

2.1.1 Register Request

When a user is first registered in our application, their username and password are sent alongside this request to our server. Once we've checked that the username is valid and unique and that the password is also valid we create an empty trusted users group array and generate an RSA Key for the user which we then split into its private and public key. We then create a new database entry based on our predefined Data Schema, attach the user's information to it and add this to our database.

2.1.2 Login Request

When a user wishes to login, the server simply searches the database for the username and password supplied in the request. If it finds a match the server returns only that user's trusted group and no other information pertaining to them to the front end application. Otherwise a response is sent to the front end application signalling that the username and password are not valid.

2.1.3 Get Users Request

This request is made when a user logs in, as we want to display the usernames of all the users registered in our system which the logged in user may then choose from to add to their trusted group. This request simply returns only the usernames of all users in our database.

2.1.4 Update Group Request

This request is made when a user removes or adds someone to their trusted group. All that is carried out is an update of a specific user's trusted group in our database. Both the logged in user's name and the already updated group (i.e. the group with the additional user, or without the removed user) are sent to the server in the request.

2.1.5 Encrypt Message Request

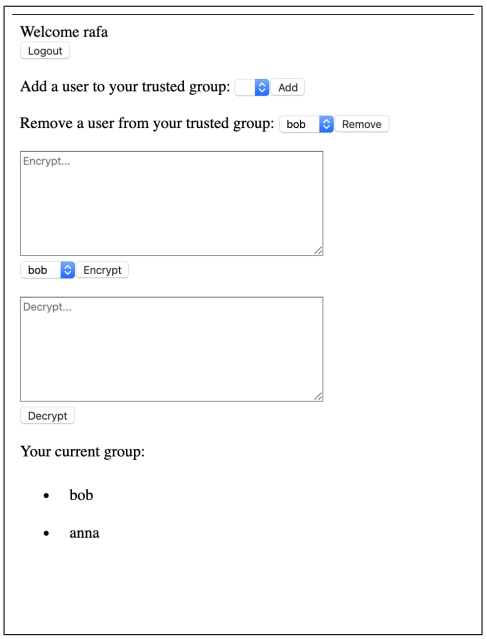
The encrypt request handler receives the username of the user we are sending the message to and the message for encryption in the request. Once we verify that both these passed in parameters are valid we go ahead an encrypt the message and return this encrypted message to the front end application.

2.1.6 Decrypt Message Request

Similarly to the encrypt request handler, the decrypt request handler receives the username of the logged in user as well as the message to be decrypted in the request. Again we verify that both these inputs are valid and once we do that we can decrypt the message. The decrypted message is then sent back to our front end application.

2.2 Front End

(a) Screen 1 (Before Log in or Register)



(b) Screen 2 (After Log in or Register)

The front end portion of our application which was created using the React Framework is divided into two screens as shown above. The first screen acts as a landing page where users are prompted to register or log in with their accounts. Once the user successfully logs in or registers themselves they are brought to the second screen, where they can choose to add and remove people from their trusted group as well as encrypt and decrypt messages. The front end app simply serves as place to get input from the user, the only state variables it keeps are the username of the currently logged in user, their password (which is set to null as soon as log in or registration is completed), their trusted group, the usernames of all users in the system and a boolean variable isLoggedIn which is utilised to control the app's rendering of either the log in screen or the screen post login. All of the actions

(login and register, add and remove users from the group, encrypt and decrypt) are sent as requests for the back end server to carry out the required actions and return the necessary information whether that be the encrypted or decrypted text for example. As such the only actions carried out by the front end application are simply updating the information displayed to the user.

3 Group Membership

Managing group membership is handled both by the front end and the back end portions of the application. When a user wants to add or remove a user from their trusted group, they simply select the user they want to add or remove and click the add or remove button. Under the hood, the front end application keeps a state variable for the current user's trusted group so given the input of the user to be added or removed, the application updates that state variable accordingly. Furthermore a request is sent to server containing the username of the logged in user as well as the updated group. As a result the server is able to update this user's entry in the database with their updated group.

N.B: Adding a user to your group, does not automatically add you to theirs

4 Key Management Infrastructure

Key generation and management is only carried out by our back end server. When a new user is first registered our back end server uses a NodeJS module called node-rsa to generate an RSA key for the user. This key however is actually the combination of the public and private key so before we store these in our database alongside the rest of the information about our new user we must split the key. Luckily the node-rsa package allows us to do so as it lets us export keys in specific formats, the two I chose were 'pkcs1-der' which allows for the exporting of the private key as a Binary Buffer from the whole Key and 'pkcs8-public-der' which does the same expect it exports a public key as a Binary Buffer. Both of these keys are then stored in the database and are only accessed by the server when necessary for encryption or decryption.

5 Encryption and Decryption

The actual encryption and decryption of the text is only carried out by our back end server. In the case of encryption, the front end app sends the message to be encrypted and the username of the receiver in the form of a request to the server. We then look in our database for that username and acquire the corresponding public key which we then use to encrypt the message. Only the encrypted message is then returned to the front end application in the from of a Buffer which is then converted to a String before being displayed to the user. Similarly when decrypting text the server receives the message to decrypt and the logged

in user from the front end application in the form of a request. The server then looks for this user in our database this time acquiring their private key and then carries out the decryption. Yet again only the decrypted text is returned to the front end application which it then converts from a Buffer to a String which is then displayed to the user.

6 Conclusion and Observations

Although a user name and password are required to utilise this application, communication between the server and database and react app and server is not encrypted in any way and therefore can be listened to. Furthermore our database which contains our user's private and public key can also be the target of an attack. Overall this application is not the most secure, but works well as a prototype to demonstrate how text encryption and decryption could be carried out in a web context.

7 Appendix - Code

7.1 Front End Application

```
import React, {Component} from "react";
import axios from 'axios';

//this is the front end potion of our application
//which handles all user interaction and then
//forwards any necessary actions/requests to
//our server
class App extends Component {
  state = {
    users: [],
    user: null,
    password: null,
    isLoggedIn: false,
    group: []
  };

  //get request for all the usernames of users
  //that have registered themselves in our app
  getUsersFromDB = () => {
    fetch('http://localhost:3001/api/getUsers')
      .then((data) => data.json())
      .then((res) => this.setState({users: res.data}));
  };

  //post request to register a new user
  //based on inputted username and password
  //usernames must be unique, on success we
  //enter the logged in stage and our application
  //shows the user more functions, on failure
```

```
//a pop up alert comes up and the user is requested
//to try again
register = (username, password) => {
  axios.post('http://localhost:3001/api/register', {
    user: username,
    password: password,
  })
  .then((res) => {
    if(res.data.success){
      this.setState({isLoggedIn: true });
      this.getUsersFromDB();
    }
    else {
      window.alert(res.data.error);
      this.setState({user: null, password: null});
      document.getElementById('password').value='';
      document.getElementById('usernam').value='';
    }
  });
};

//get request for logging in a user based on the
//inputted username and password, on success we
//enter the logged in stage and the rest of the app
//is avaibale for use, on failure a pop up alert
//requests the user to try again
login = (username, password) => {
  axios.get('http://localhost:3001/api/login', {
    params: {
      user: username,
      password: password
    }
  })
  .then((res) => {
    if(res.data.success){
      this.setState({isLoggedIn: true, group: res.data.data });
      this.getUsersFromDB();
    }
    else {
      window.alert(res.data.error);
      this.setState({user: null, password: null});
      document.getElementById('password').value='';
      document.getElementById('usernam').value='';
    }
  });
};

//post request to update the users trusted group
//based on an inputted other user to be added
addToGroup = () => {
  var e = document.getElementById('userSelect');
```

```
    var user = e.options[e.selectedIndex].value;

    //new user directly added to the trusted group
    this.state.group.push(user);

    //we use this dummy setState to update the UI
    this.setState({group: this.state.group});

    axios.post('http://localhost:3001/api/updateGroup', {
      user: this.state.user,
      update: this.state.group
    });
  };

  //post request to update the users trusted group
  //based on an inputted other user to be removed
  removeFromGroup = () => {
    var e = document.getElementById('userDelete');
    var user = e.options[e.selectedIndex].value;

    //remove the user from the group
    var newGroup = this.state.group.filter(e => e !== user);

    //update the current user's group in the app to be
    //the new group i.e. the one with the removed user
    this.setState({group: newGroup});

    axios.post('http://localhost:3001/api/updateGroup', {
      user: this.state.user,
      update: newGroup
    });
  };

  //get request to encrypt a message meant for
  //a trusted user only which is selected from a drop down
  //on success the original message is replaced by the encrypted
  //version the app for the user to copy and send to the desired
  //destination, on failure the textarea is simply reset
  encrypt = () => {
    var message = document.getElementById('encrypt').value;

    var e = document.getElementById('userEncrypt');
    var user = e.options[e.selectedIndex].value;


    axios.get('http://localhost:3001/api/encrypt', {
      params: {
        user: user,
        message: message
      }
    })
```

```
    .then((res) => {
      if(res.data.success){
        document.getElementById('encrypt').value=res.data.message;
      }
      else {
        window.alert(res.data.error);
        document.getElementById('encrypt').value='';
      }
    });
  };

  //get request to decrypt a message meant only for the current
  //logged in user and only them on success the encrypted message
  //is replaced by the decrypted original message on failure the textarea
  //is simply reset
  decrypt = () => {
    var message = document.getElementById('decrypt').value;
    axios.get('http://localhost:3001/api/decrypt', {
      params: {
        user: this.state.user,
        message: message
      }
    })
    .then((res) => {
      if(res.data.success){
        var bufferOriginal = Buffer.from(res.data.message.data);
        document.getElementById('decrypt').value=bufferOriginal.toString();
      }
      else {
        window.alert(res.data.error);
        document.getElementById('decrypt').value='';
      }
    });
  };

  //html to be render for our application, divided into two sections
  //the first describes the app after logging in or registering an account
  //the second describes the app before logging in or registering an account
  render() {
    if (this.state.isLoggedIn) {
      return (
        <div>
          <div>
          Welcome {this.state.user}
          </div>
          <button onClick={() => this.setState({user: null, password: null,
              isLoggedIn: false, group: []})}>
            Logout
          </button>
          <br/>
          <br/>
```

```
          <label>Add a user to your trusted group: </label>
          <select id="userSelect">
          {this.state.users.map((data) => {
            if (data.user != this.state.user && !this.state.group.includes(data.
                user)){
              return (
                <option value={data.user} key={data.user}>{data.user}</option>
              );
            }
          })}
          </select>
          <button onClick={() => this.addToGroup()}>
              Add
          </button>
          <br/>
          <br/>
          <label>Remove a user from your trusted group: </label>
          <select id="userDelete">
          {this.state.group.map((data) => {
            return(
              <option value={data} key={data}>{data}</option>
            )
          })}
          </select>
          <button onClick={() => this.removeFromGroup()}>
              Remove
          </button>
          <br/>
          <br/>
          <div>
            <textarea
              id="encrypt"
              type="text"
              placeholder="Encrypt..."
              style={{ width: '300px' , height: '100px'}}
            />
          </div>
          <select id="userEncrypt">
          {this.state.group.map((data) => {
            return(
              <option value={data} key={data}>{data}</option>
            )
          })}
          </select>
          <button onClick={() => this.encrypt()}>
            Encrypt
          </button>
          <br/>
          <br/>
          <div>
            <textarea
```

```
            id="decrypt"
            type="text"
            placeholder="Decrypt..."
            style={{ width: '300px' , height: '100px'}}
          />
        </div>
        <button onClick={() => this.decrypt()}>
          Decrypt
        </button>
        <br/>
        <br/>
        <div>
          Your current group:
          <ul id="group">
            {this.state.group.length <= 0 ? 'EMPTY' : this.state.group.map((
                data) => (
              <li style = {{ padding: '10px'}} key={data}>
                {data}
              </li>
            ))}
          </ul>
        </div>
      </div>
    );
  }
  else {
    return (
      <div>
        <br/>
        <div style = {{paddding: '10px'}}>
          <input
            id="usernam"
            type="text"
            onChange={(e) => this.setState({ user: e.target.value })}
            placeholder="Username"
            style={{ width: '200px' }}
          />
          <br/>
          <input
            id="password"
            type="password"
            onChange={(e) => this.setState({ password: e.target.value })}
            placeholder="Password"
            style={{ width: '200px' }}
          />
          <br/>
          <button onClick={() => this.register(this.state.user, this.state.
              password)}>
            Register
          </button>
```

```
            <button onClick={() => this.login(this.state.user, this.state.
                password)}>
              Login
            </button>
          </div>
        </div>
      );
    }
  }
}

export default App;
```

7.2 Back End Server and Data Schema

```
//for communicating with our DB
const mongoose = require('mongoose');

//for running our server
const express = require('express');

//for running our server specificallylocally
var cors = require('cors');

//to make our requests more human readable
const bodyParser = require('body-parser');

//to make our logs more human readable
const logger = require('morgan');

//our defined data structure for our application
const Data = require('./data');

//to access files from the file system
const fs = require('fs');

//to generate our public and private key for each user
const NodeRSA = require('node-rsa');

//this is the backend portion of our application
//which handles all actions and requests sent to
//it by the front end. It's also the only part
//which communicates with our mongodb application
//we've set up

var mongopassword = "";


//password required to access our database read from a local
//file for security reasons
try {
  mongopassword = fs.readFileSync('mongo_password.txt', 'UTF-8');
```

```
} catch(err){
  console.log(err);
}

//server set up
const API_PORT = 3001;
const app = express();
app.use(cors());
const router = express.Router();

//db connection set up
const dbRoute = 'mongodb+srv://admin:'+mongopassword+'@cluster0-2m7nf.mongodb.net/test?retryWrites=true&w=majority'

mongoose.connect(dbRoute, {useNewUrlParser: true, useUnifiedTopology: true });

let db = mongoose.connection;

db.once('open', () => console.log('connected to the database'));

db.on('error', console.error.bind(console, 'MongoDB connection erro:'));

//response parser and logger set up
app.use(bodyParser.urlencoded({extended:false}));
app.use(bodyParser.json());
app.use(logger('dev'));

//register request handler
router.post('/register', async (req, res) => {
  //new empty data set to be added
  //to the database
  let data = new Data();

  //empty group of trusted users
  const group = [];

  //user name and password of user
  //passed through the request
  const {user, password} = req.body;

  //check to see if username and password are valid
  //if it fails we return a response with an error
  if (!user || !password) {
    return res.json({
      success: false,
      error: 'invalid inputs',
    });
  }

  //check to see if the username is unique or not
  const isDuplicate = await Data.exists({user: user});
```

```

  //given that it is unique...
  if (!isDuplicate) {
    //generate a full RSA key using the NodeRSA module
    const key = new NodeRSA({b: 2048});

    //split the full key into its public and private parts
    //which are saved as Binary Buffers
    const private_key=key.exportKey('pkcs1-der');
    const public_key=key.exportKey('pkcs8-public-der');

    //add all the necessary components to our
    //predifined data structure
    data.public_key=public_key;
    data.private_key=private_key;
    data.user = user;
    data.password = password;
    data.group = group;

    //save the data to our database
    data.save((err) => {
      if (err) return res.json({success: false, error: err});
      return res.json({success: true});
    });
  }
  //if it's not unique we return a response to the
  //front end with an error
  else {
    return res.json({
      success: false,
      error: 'username already in use',
    });
  }
});

//login request handler
router.get('/login', (req, res) => {
  //username and password passed in
  //throught the query params in the request
  const {user, password} = req.query;

  //if the username or password are invalid
  //an error response is sent to the front end
  if (!user || !password) {
    return res.json({
      success: false,
      error: 'invalid inputs',
    });
  }

  //we look for the passed in username and password combination
```

```
  //on success we only return their trusted group to the application
  //and no other sensitive information that may be stored in the db
  //on failure we return an error message to the front end
  Data.findOne({user: user, password: password}, 'group -_id', (err, data) => {
    if (err) return res.json({success: false, error: err});
    if (data==null){
      return res.json({success: false, error: "invalid username or password"});
    }
    else {
      return res.json({success: true, data: data.group});
    }
  });
});

//getUsers request handler
router.get('/getUsers', (req, res) => {
  //we simply return ONLY all the usernames
  //present in our database
  Data.find({}, 'user -_id', (err, data) => {
    if (err) return res.json({success: false, error: err});
    return res.json({success: true, data: data});
  });
});

//updateGroup request handler
router.post('/updateGroup', (req, res) => {
  //we get the user whose group must be updated
  //and the updated group from our request
  const {user, update} = req.body;

  //we simply search for that user in the database
  //and update the group key with the new group
  Data.updateOne({user: user}, {group: update}, (err) => {
    if (err) return res.json({success: false, error: err});
    return res.json({success: true});
  });
});

//encrypt request handler
router.get('/encrypt', (req, res) => {
  //we get the user that we want to send the message to
  //and the message that we want to send from the request
  const {user, message} = req.query;

  //make sure the user and the message are valid inputs
  //otherwise we send an error response to the front
  //end application
  if (!message || !user) {
    return res.json({
      success: false,
      error: 'invalid inputs',
```

```
    });
  }

  //we look for the user in our database and
  //ONLY acquire their publick key which we require
  //for our encryption and no other information
  Data.findOne({user: user}, 'public_key -_id', (err, data) => {
    if (err) return res.json({success: false, error: err});
    if (data==null){
      return res.json({success: false, error: "invalid username or password"});
    }
    else {
      //we then encrypt our message with the user's
      //public key (in this case the "user" is who we are
      //sending our message to and not the logged in user)
      //any errors are caught and sent back to the front end app
      try {
        const key = new NodeRSA();
        key.importKey(data.public_key,'pkcs8-public-der');
        var encryptedMessage=key.encrypt(message, 'base64');
      }
      catch (error) {
        return res.json({success: false, error: "error while encrytping"});
      }


      //only the encrypted message is returned to the front
      //end app and no other sensitive information
      return res.json({success: true, message: encryptedMessage});
    }
  });
});

//decrypt request handler
router.get('/decrypt', (req, res) => {
  //we get the currently logged in user
  //and the message to be decrypted
  //from the request
  const {user, message} = req.query;

  //we ensure the messsage is valid
  //otherwise we send an error response
  //to the front end
  if (!message) {
    return res.json({
      success: false,
      error: 'invalid inputs',
    });
  }

  //we look for the user in our database
```

```
  //and acquire ONLY their private key which we need
  //for decryption and no other information
  Data.findOne({user: user}, 'private_key -_id', (err, data) => {
    if (err) return res.json({success: false, error: err});
    if (data==null){
      return res.json({success: false, error: "invalid username or password"});
    }
    else {
      //we then decrypt our message with the user's
      //private key (in this case the "user" is the
      //logged in user)
      //any errors are caught and sent back to the front end app
      try {
        const key = new NodeRSA();
        key.importKey(data.private_key,"pkcs1-der");
        var decryptedMessage=key.decrypt(message);
      }
      catch (error) {
        return res.json({success: false, error: "error while decrypting"});
      }

      //only the decrypted message is returned to the front
      //end app and no other sensitive information
      return res.json({success: true, message: decryptedMessage});
    }
  });
});

app.use('/api', router);

app.listen(API_PORT, () => console.log(`listening on port ${API_PORT}`));
```

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

//our data format for every user
//in our application
const DataSchema = new Schema(
  {
    user: String,
    password: String,
    group: Array,
    public_key: Buffer,
    private_key: Buffer
  },
  {timestamps: true}
);

module.exports = mongoose.model("Data", DataSchema);
```