3 December 2017

Chess681

Gaming Project

ISA 681

Collaborators: Ruchita Garde, Aruna Sindhuja Peri

Table of Contents

1. Introduction

The main purpose of this report is to describe the Web based multiplayer Chess game 'Chess681' developed by us. This report documents the programming languages and packages used by us, the design, architecture and structuring of the code, game rules and how it is supposed to be played, the encryption techniques and detailed explanation about why, we believe, this game is secure.

2. Design and Architecture

2.1. Software specifications

You need the following things installed on your system to play the game. The installation instructions have been provided in section 3.

Server:

- Node JS – Web Server Node.js v8.9.1
- Express JS - Framework
- Socket.io - helps use socket communication to make real time communication possible
- Mongo DB along with Mongoose ; Server version: 3.4.10
- Passport JS for authentication
- Handlebars.js to make html templates

Client:

- HTML5, CSS3, Twitter bootstrap and Font awesome for the UI
- Javascript utilities for client-side logic:
  JQuery – library to support our client-side scripting in handlebars
  Lodash – used for math functions
  Moment.js – handles the timings for our game when run
  Messenger.js – used for our network communication
- Socket.io client

2.2. Architecture

The Architecture consists of the following modules:

- User authentication and registration process (/login & /register)
- Real time multi-player gaming logic (See gaming logic section)
- RESTful API (/api) for client server communication
- Real time monitoring dashboard (/monitor)

Figure 1 shows the architecture of the application.

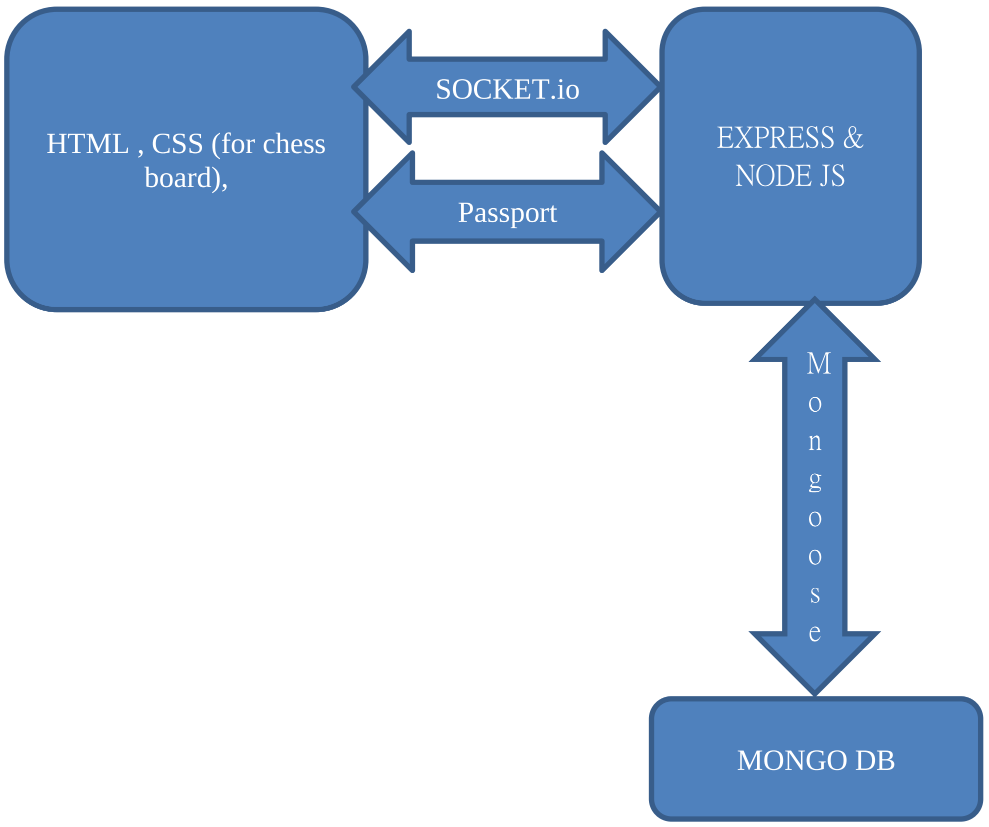


Figure 1. Architecture

3. Instructions

3.1. Installation Instructions

- Prerequisites:
  1) Node JS
     Download and install 8.9.2 LTS for Windows (x64) from https://nodejs.org/en/
  2) Mongo DB
     For this project we have installed MongoDB from the Community Center. Install mongodb-win32-x86_64-2008plus-ssl-3.4.10-signed (for Windows 64 bit). We will be running it on the default port (27017) and using the default database chess_681.

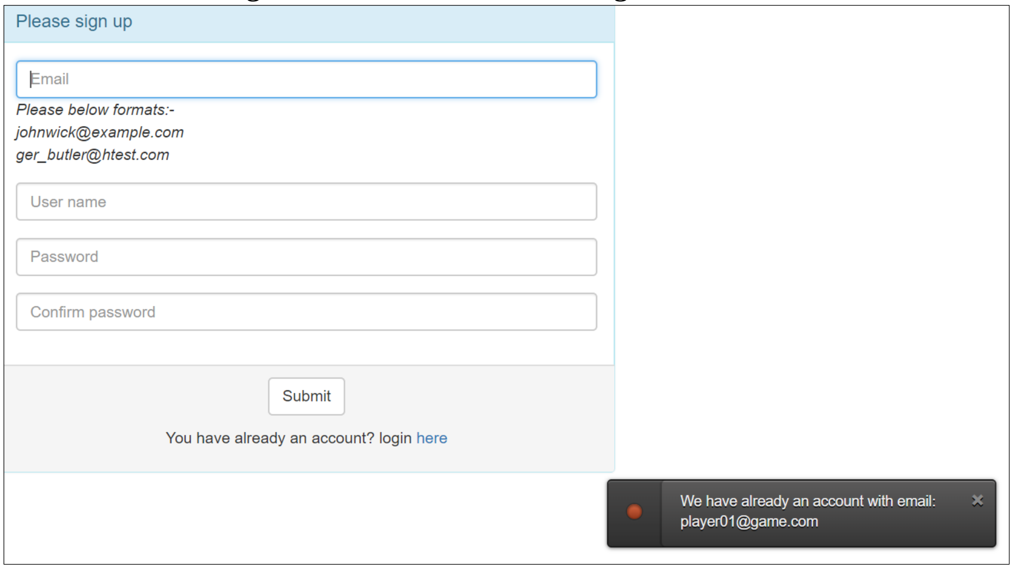
- Steps to run the application
  1) Start your mongoDB server and client:
     (1) run command prompt as an administrator
     ➔ Navigate to bin folder inside MongoDB and execute the command below:
     ➔ mongod.exe --dbpath "C:\Program Files\MongoDB\data\db"
     Example: C:\Program Files\MongoDB\Server\3.4\bin>mongod.exe --dbpath "C:\Program Files\MongoDB\data\db"
     (2) run command prompt as an administrator
     ➔ Navigate to bin folder inside MongoDB and execute the command below:
     ➔ mongo.exe
     Example: C:\Program Files\MongoDB\Server\3.4\bin>mongo.exe

2) Unzip the project folder
3) Run node.js command prompt
   ➔ Navigate to the project folder
   Example: C:\Users\sindh\Chess681_ArunaRuchita>
   ➔ Run the below commands
   > npm install
   > node app.js OR > node .

4) Open your browser and run the localhost https://localhost:3000/ to see the chess681 web app running.

3.2. Playing Instructions and Game Rules

This game can be played by multiple users, 2 users per game room. A user needs to first register using his username, email id and password. If he has already done so, he can login using his email id and password. User can also change his password through the accounts page.
If a user already exists in the database and he attempt to register again the application will not allow the user to register and will show a message at the bottom of the screen as shown below:



Please sign up

Email

Please below formats:-
johnwick@example.com
ger_butler@htest.com

User name

Password

Confirm password

Submit

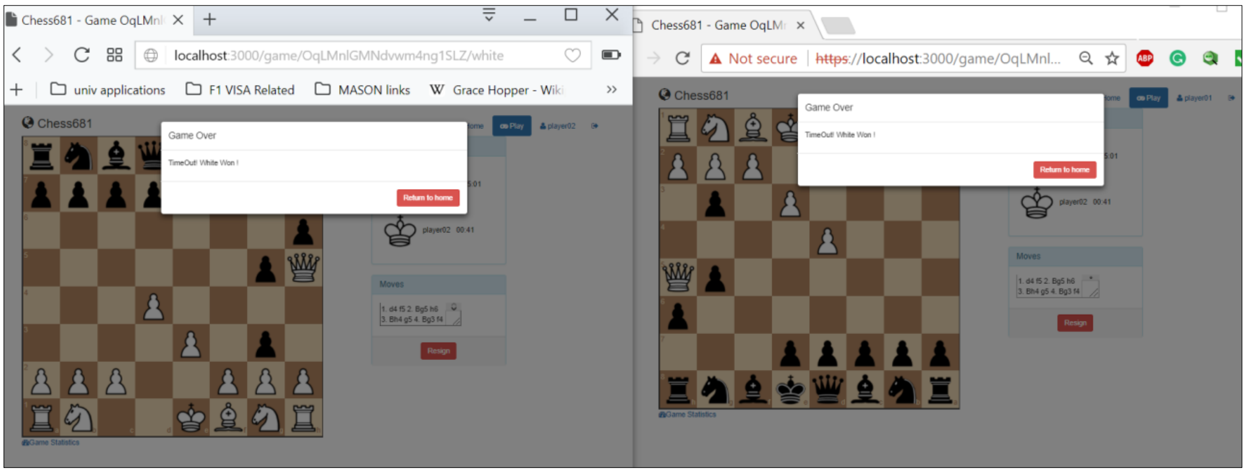
You have already an account? login here

We have already an account with email: player01@game.com

Once logged in, the following is a general flow of the game:
1. Each player can create a game by sending a 'create-game' event to the server.
2. The server creates a new game and replies to the player with a randomly generated token for the game.
3. The player sends this token to other players and waits for them to join the game.
4. Others players join the game by sending a 'join-game' event to the server.
5. Once both players join the game, the server joins players sockets to the same socket.io room.
6. At this point, the game starts: each player can send one valid event (by chess logic) and then wait for the other player.
7. The move is broadcasted to the other player, and he will have a chance to play.
8. When the game is over, each player is notified and sockets leave the game room.

Please note that, if the first player leaves the game after the 2nd player has started the game, the game room will still keep its session. The 2nd player can then share the first player’s game room link with another player, and start playing the game.

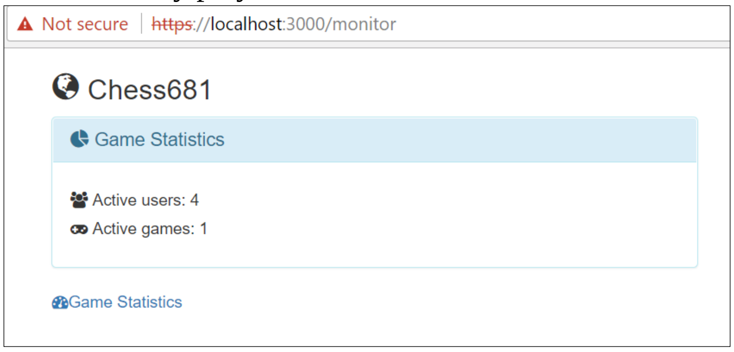
A timer keeps track of the time each player gets to make a move. If a side does not make a move for 300 seconds, the other player is declared as winner. The player can also see all the game moves on the bottom of the application on the right hand side panel 'Moves'.


Game Over

TimeOut! White Won !

1. d4 f5 2. Bg5 h6
3. Bh4 g5 4. Bg3 f4

If either of the players leave, the other player automatically wins.
Game statistics: At any point in time, you can see how many games are currently being played, and how many players are online.


Not secure | https://localhost:3000/monitor

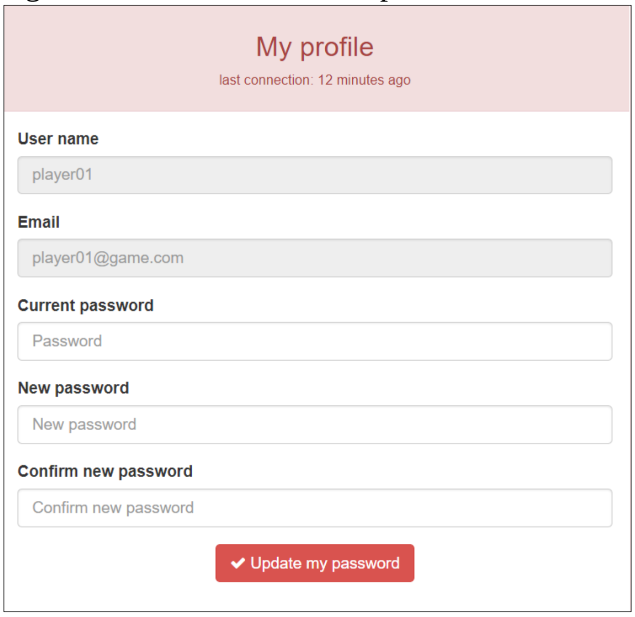
Chess681

Game Statistics

Active users: 4
Active games: 1

Game Statistics

A player also has the provision to change his password, and see when he was last active when is logs in and clicks on his own profile name.


My profile

last connection: 12 minutes ago

User name	player01
Email	player01@game.com
Current password	Password
New password	New password
Confirm new password	Confirm new password

Update my password

4. Code description and Game Rules

4.1. Code structure and files description:

- App.js: This file marks the starting point of the game. This launches the game on localhost at port 3000. This also has all the routes defined.
- Package.json and packages-lock.json: These files have all the dependencies our game requires. Package.json instructs npm which dependencies packages it needs to install.
- Views folder: This folder has all the handlebars files with .hbs extension which handles HTML templating for User Interface.
- Gamepages folder: This folder contains all the javascript files witht .js extension. These files route all the REST API requests like get and post, to get data from user and send it to the database, or fetch data from database and display to the user.
- Models folder: This folder contains mongoDB schemas for defining users and games.
- Public → chessboardjs and chessjs folders: These folders contain files to design the chessboard, its look and feel, and the logic of the chess game. These are MIT licensed chess APIs. Thanks to [1] and [2] whose libraries, developed by MIT have the chess game logic upon which we have based our application.
- Public → javascript →chess681.js: This file has the logic and working of the app.
- Public → bootstrap, fontawesome, lodash, messenger folders: These folders are other APIs and frameworks required.
- Config → database.js: This file creates connection with the database.
- Config → passport.js: This file handles user authentication.
- Config → config.js: This file handles socket connection. It creates sessions for games, handles new connections and disconnections.
- Config → util.js: This file creates salt hashes to store passwords.
- Certificates folder: This folder contains certificates generated for SSL.

5. Why This Game Is Secure?

5.1. Assurance Case

Figure 2 shows a diagrammatic representation of the Security Assurance Case of our application [3] [4].

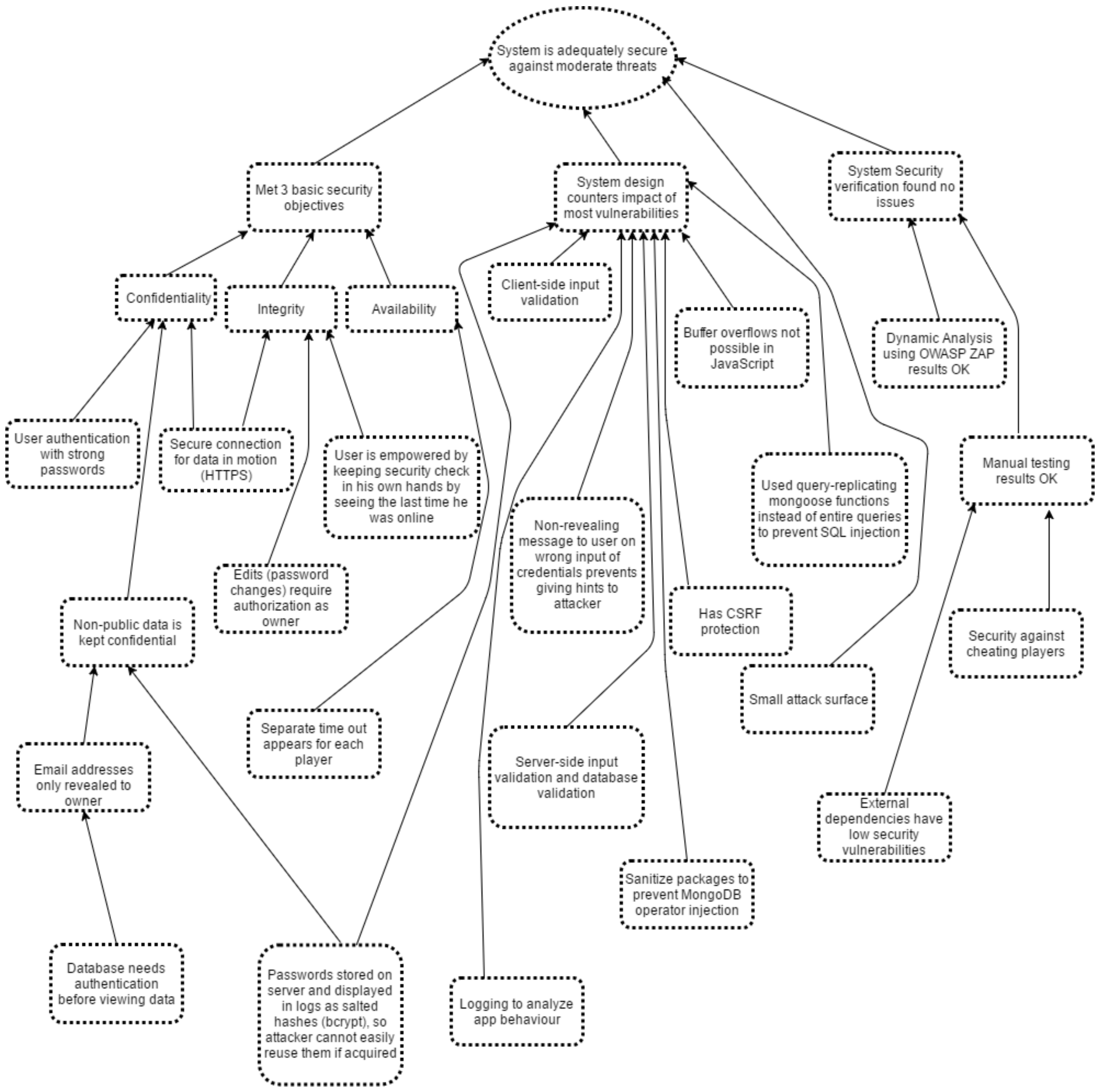
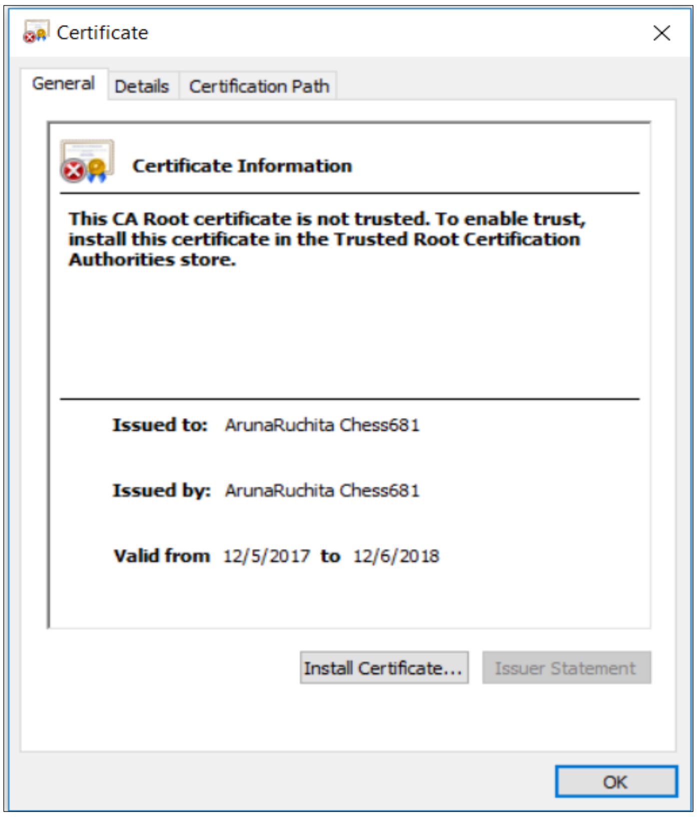

System is adequately secure against moderate threats
Met 3 basic security objectives
System design counters impact of most vulnerabilities
System Security verification found no issues
Confidentiality
Integrity
Availability
Client-side input validation
Buffer overflows not possible in JavaScript
Dynamic Analysis using OWASP ZAP results OK
User authentication with strong passwords
Secure connection for data in motion (HTTPS)
User is empowered by keeping security check in his own hands by seeing the last time he was online
Used query-replicating mongoose functions instead of entire queries to prevent SQL injection
Manual testing results OK
Non-revealing message to user on wrong input of credentials prevents giving hints to attacker
Edits (password changes) require authorization as owner
Non-public data is kept confidential
Has CSRF protection
Security against cheating players
Small attack surface
Separate time out appears for each player
Email addresses only revealed to owner
Server-side input validation and database validation
External dependencies have low security vulnerabilities
Sanitize packages to prevent MongoDB operator injection
Database needs authentication before viewing data
Passwords stored on server and displayed in logs as salted hashes (bcrypt), so attacker cannot easily reuse them if acquired
Logging to analyze app behaviour

Figure 2. Security Assurance Case

5.2. SSL/TLS and HTTPS connection

- We created a self-signed SSL certificate using OpenSSL. OpenSSL is a full-featured toolkit for TLS and SSL protocols. We generated an RSA private key. The connection is encrypted and authenticated using TLS 1.2 (a very strong protocol), ECDHE_RSA with P-256 (a strong key exchange), and AES_128_GCM (a strong cipher). Figure 3. Shows the screenshot of the SSL certificate we generated.



Certificate

General | Details | Certification Path

Certificate Information

This CA Root certificate is not trusted. To enable trust, install this certificate in the Trusted Root Certification Authorities store.

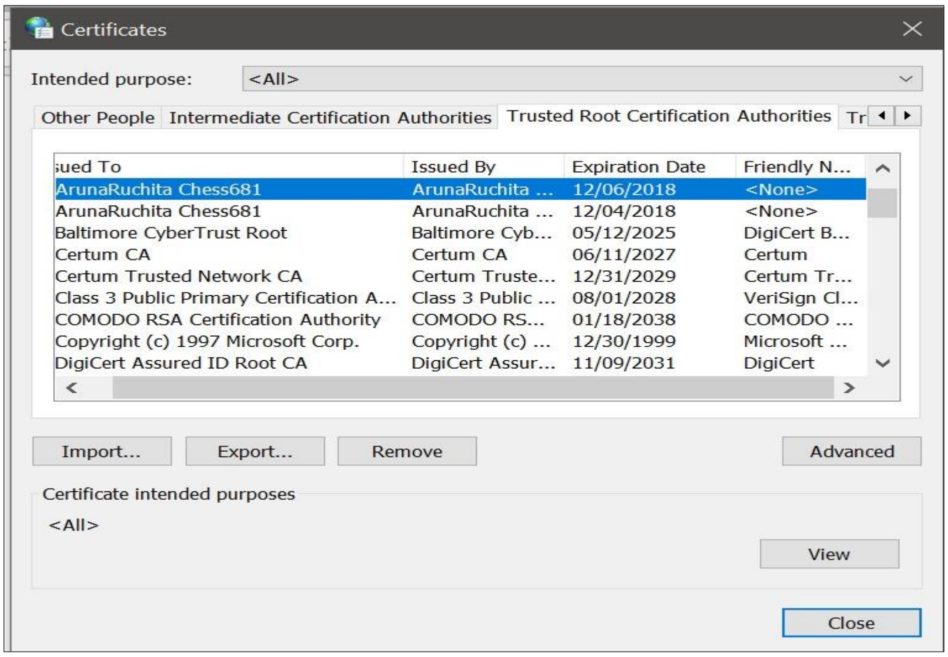
Issued to: ArunaRuchita Chess681

Issued by: ArunaRuchita Chess681

Valid from 12/5/2017 **to** 12/6/2018

Figure 3. SSL Certificate

- We also made sure that our game is running on a secure connection by using ‘https’ rather than running it on ‘http’[5]. We aim to provide the security objectives of confidentiality and integrity with this.
- Figure 4. Shows that the self-signed certificate created by us has been added to the list of Trusted Root Certification Authorities.



Certificates

Intended purpose: <All>

Other People | Intermediate Certification Authorities | Trusted Root Certification Authorities | Tr

;ued To	Issued By	Expiration Date	Friendly N...
ArunaRuchita Chess681	ArunaRuchita ...	12/06/2018	<None>
ArunaRuchita Chess681	ArunaRuchita ...	12/04/2018	<None>
Baltimore CyberTrust Root	Baltimore Cyb...	05/12/2025	DigiCert B...
Certum CA	Certum CA	06/11/2027	Certum
Certum Trusted Network CA	Certum Truste...	12/31/2029	Certum Tr...
Class 3 Public Primary Certification A...	Class 3 Public ...	08/01/2028	VeriSign Cl...
COMODO RSA Certification Authority	COMODO RS...	01/18/2038	COMODO ...
Copyright (c) 1997 Microsoft Corp.	Copyright (c) ...	12/30/1999	Microsoft ...
DigiCert Assured ID Root CA	DigiCert Assur...	11/09/2031	DigiCert

Certificate intended purposes

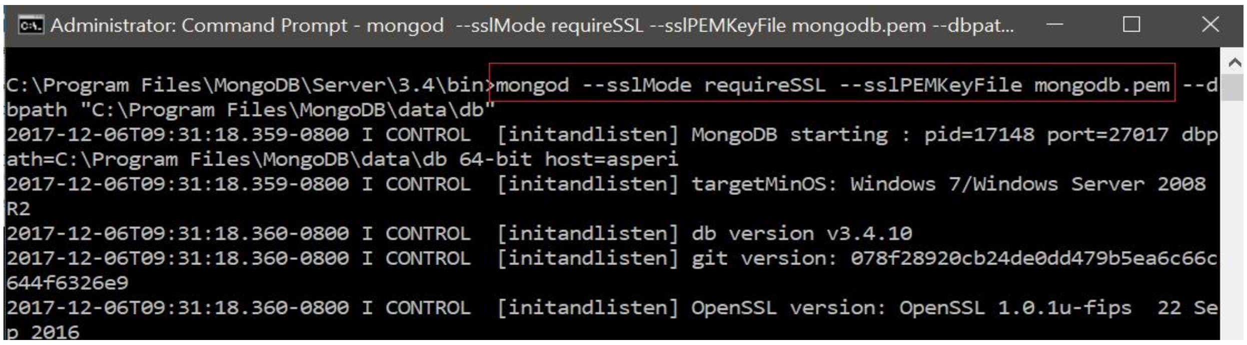
<All>

Figure 4. Trusted root certification authorities

- Attempt at using Mongo SSL-TLS connection:

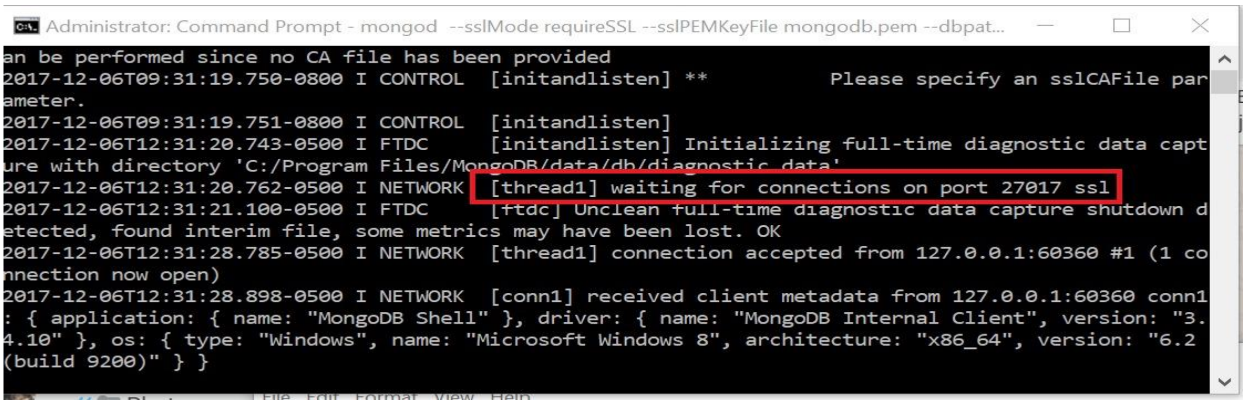
We tried configuring SSL on the Mongo Shell.
Although, the server and client were successfully communicating with each other, our application could not connect because we could not handle our mongoose.js scripts to run it via a secure SSL/TLS. As part of future scope, SSL can be configured on MongoDB as well.
Thanks to [10] we got our selves introduced to a new database and a new concept.

Figures 5, 6, and 7 shows our attempt at configuring SSL on the DB, in which we were successful.


Administrator: Command Prompt - mongod --sslMode requireSSL --sslPEMKeyFile mongodb.pem --dbpat...

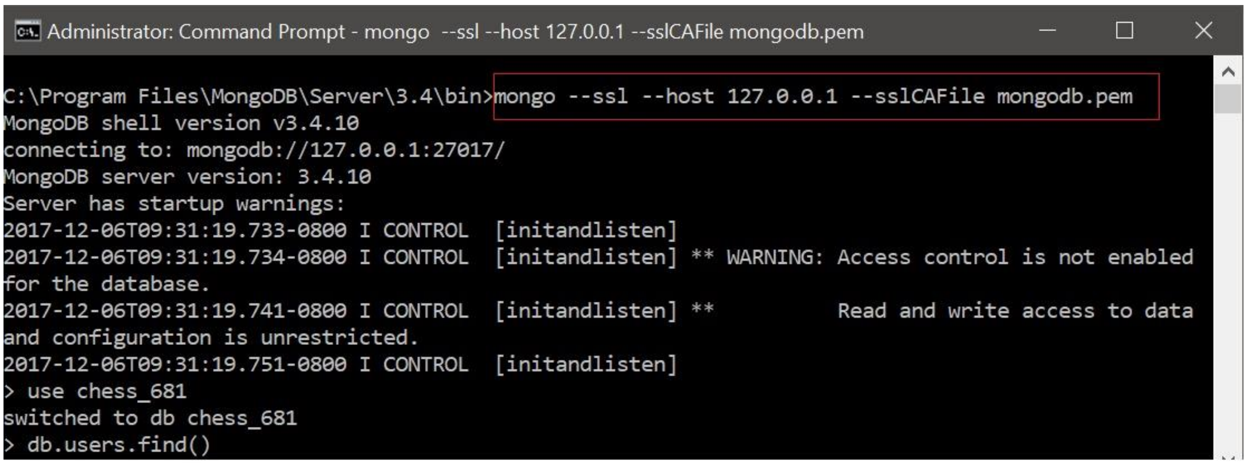
```
C:\Program Files\MongoDB\Server\3.4\bin>mongod --sslMode requireSSL --sslPEMKeyFile mongodb.pem --d
bpath "C:\Program Files\MongoDB\data\db"
2017-12-06T09:31:18.359-0800 I CONTROL  [initandlisten] MongoDB starting : pid=17148 port=27017 dbp
ath=C:\Program Files\MongoDB\data\db 64-bit host=asperi
2017-12-06T09:31:18.359-0800 I CONTROL  [initandlisten] targetMinOS: Windows 7/Windows Server 2008
R2
2017-12-06T09:31:18.360-0800 I CONTROL  [initandlisten] db version v3.4.10
2017-12-06T09:31:18.360-0800 I CONTROL  [initandlisten] git version: 078f28920cb24de0dd479b5ea6c66c
644f6326e9
2017-12-06T09:31:18.360-0800 I CONTROL  [initandlisten] OpenSSL version: OpenSSL 1.0.1u-fips  22 Se
p 2016
```

Figure 5. Command for SSL configuration on mongod server


Administrator: Command Prompt - mongod --sslMode requireSSL --sslPEMKeyFile mongodb.pem --dbpat...

```
an be performed since no CA file has been provided
2017-12-06T09:31:19.750-0800 I CONTROL  [initandlisten] **          Please specify an sslCAFile par
ameter.
2017-12-06T09:31:19.751-0800 I CONTROL  [initandlisten]
2017-12-06T12:31:20.743-0500 I FTDC     [initandlisten] Initializing full-time diagnostic data capt
ure with directory 'C:/Program Files/MongoDB/data/db/diagnostic.data'
2017-12-06T12:31:20.762-0500 I NETWORK  [thread1] waiting for connections on port 27017 ssl
2017-12-06T12:31:21.100-0500 I FTDC     [ftdc] Unclean full-time diagnostic data capture shutdown d
etected, found interim file, some metrics may have been lost. OK
2017-12-06T12:31:28.785-0500 I NETWORK  [thread1] connection accepted from 127.0.0.1:60360 #1 (1 co
nnection now open)
2017-12-06T12:31:28.898-0500 I NETWORK  [conn1] received client metadata from 127.0.0.1:60360 conn1
: { application: { name: "MongoDB Shell" }, driver: { name: "MongoDB Internal Client", version: "3.
4.10" }, os: { type: "Windows", name: "Microsoft Windows 8", architecture: "x86_64", version: "6.2
(build 9200)" } }
```

Fig 6. Connection started with SSL


Administrator: Command Prompt - mongo --ssl --host 127.0.0.1 --sslCAFile mongodb.pem

```
C:\Program Files\MongoDB\Server\3.4\bin>mongo --ssl --host 127.0.0.1 --sslCAFile mongodb.pem
MongoDB shell version v3.4.10
connecting to: mongodb://127.0.0.1:27017/
MongoDB server version: 3.4.10
Server has startup warnings:
2017-12-06T09:31:19.733-0800 I CONTROL  [initandlisten]
2017-12-06T09:31:19.734-0800 I CONTROL  [initandlisten] ** WARNING: Access control is not enabled
for the database.
2017-12-06T09:31:19.741-0800 I CONTROL  [initandlisten] **          Read and write access to data
and configuration is unrestricted.
2017-12-06T09:31:19.751-0800 I CONTROL  [initandlisten]
> use chess_681
switched to db chess_681
> db.users.find()
```

Fig 7. Mongoose connection started with SSL

5.3. Authentication with strong passwords

5.3.1. Strong rules

Users are authenticated with strong passwords. Regular expression patterns have been provided on both client (hbs) and server (javascript) to make sure that the password has minimum length of 8, and to ensure that it contains at least one number, one uppercase letter and one lowercase letter. We have used Passport.js which is a strong authentication middleware for Node.js [6]. It uses secure keys to validate identity.

5.3.2. Authorization before edit

Users who want to edit their passwords first need to be authorized to do so by entering their old passwords. We aim to provide the first security objective i.e. confidentiality with authentication.

5.3.3. User empowerment for security

After logging in, on the accounts page of a user, he can see when he was last active. This makes it easy for the user to track malicious activity, and in a way alerts user to change his password in case he finds something wrong.

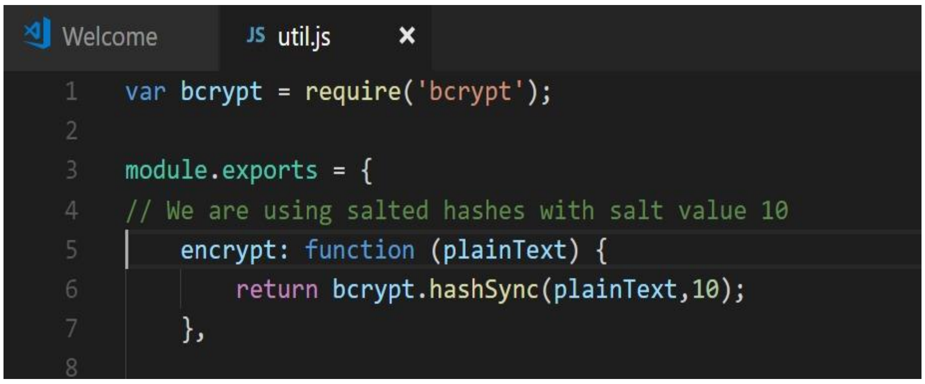
5.4. Confidentiality of non-public data

5.4.1. Email addresses only revealed to owner

Although the user is required to enter his email id while registering, only his username is displayed when he is playing online with other users. This helps protect his privacy and identity.

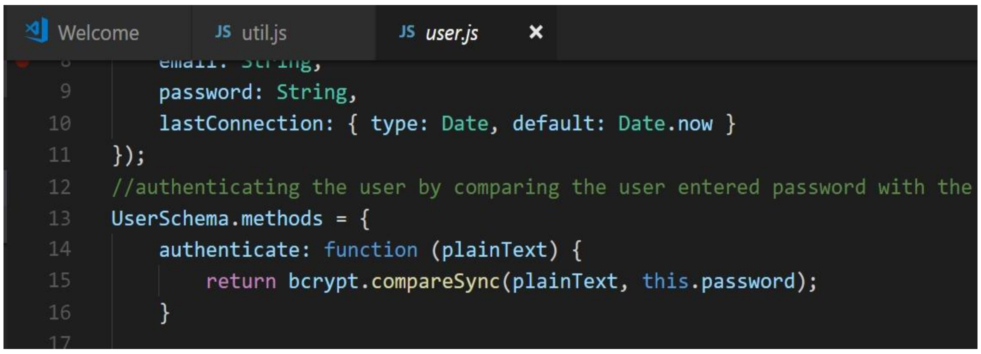
5.4.2. Passwords stored in salted hashes

We are storing passwords in the form of salted hashes in the database to make sure that, even if an attacker gains access to the database, he will not be able to view the passwords. They appear as salted hashes even in database and the logging. We have used the “bcrypt” npm node module to perform salted hashing with salt value 10.
Figures 8 and 9 show code snippets. Figure 10. shows the database entries with salted hashes. Please note that all the passwords used in all three entries are the same, but the salted hashes for each of them are different. This ensures our “bcrypt” node module is functioning as expected.



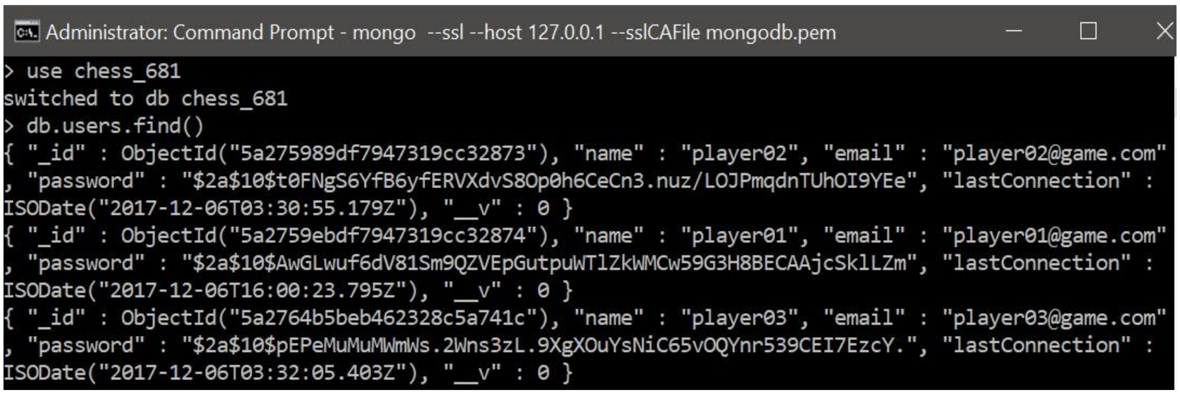
```
var bcrypt = require('bcrypt');

module.exports = {
// We are using salted hashes with salt value 10
    encrypt: function (plainText) {
        return bcrypt.hashSync(plainText,10);
    },

```

Figure 8. Salt Hashing



```
    password: String,
    lastConnection: { type: Date, default: Date.now }
});
//authenticating the user by comparing the user entered password with the
UserSchema.methods = {
    authenticate: function (plainText) {
        return bcrypt.compareSync(plainText, this.password);
    }
```

Figure 9. Salt hash comparing for authentication

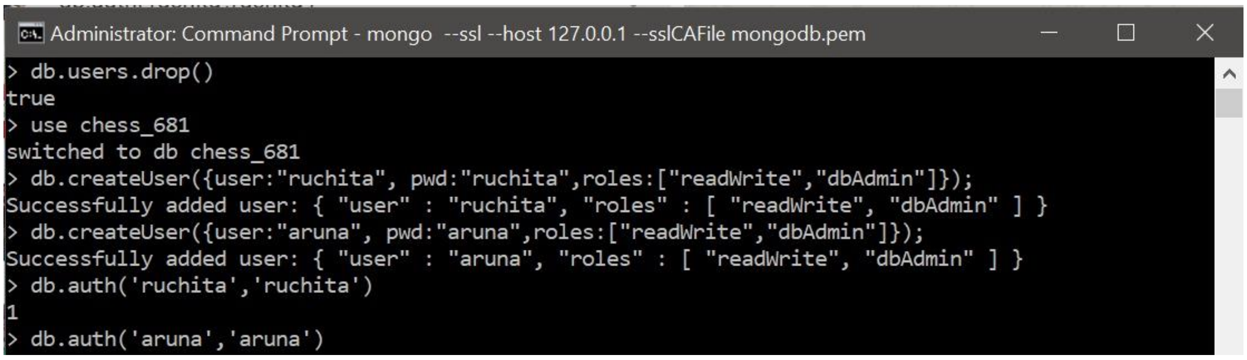

Administrator: Command Prompt - mongo --ssl --host 127.0.0.1 --sslCAFile mongodb.pem

```
> use chess_681
switched to db chess_681
> db.users.find()
{ "_id" : ObjectId("5a275989df7947319cc32873"), "name" : "player02", "email" : "player02@game.com"
, "password" : "$2a$10$t0FNgS6YfB6yfERVXdvS8Op0h6CeCn3.nuz/LOJPmqdnTUhOI9YEe", "lastConnection" :
ISODate("2017-12-06T03:30:55.179Z"), "__v" : 0 }
{ "_id" : ObjectId("5a2759ebdf7947319cc32874"), "name" : "player01", "email" : "player01@game.com"
, "password" : "$2a$10$AwGLwuf6dV81Sm9QZVEpGutpuWTlZkWMCw59G3H8BECAAjcSklLZm", "lastConnection" :
ISODate("2017-12-06T16:00:23.795Z"), "__v" : 0 }
{ "_id" : ObjectId("5a2764b5beb462328c5a741c"), "name" : "player03", "email" : "player03@game.com"
, "password" : "$2a$10$pEPeMuMuMWmWs.2Wns3zL.9XgXOuYsNiC65vOQYnr539CEI7EzcY.", "lastConnection" :
ISODate("2017-12-06T03:32:05.403Z"), "__v" : 0 }
```

Figure 10. Database Entry for Salted Hashes

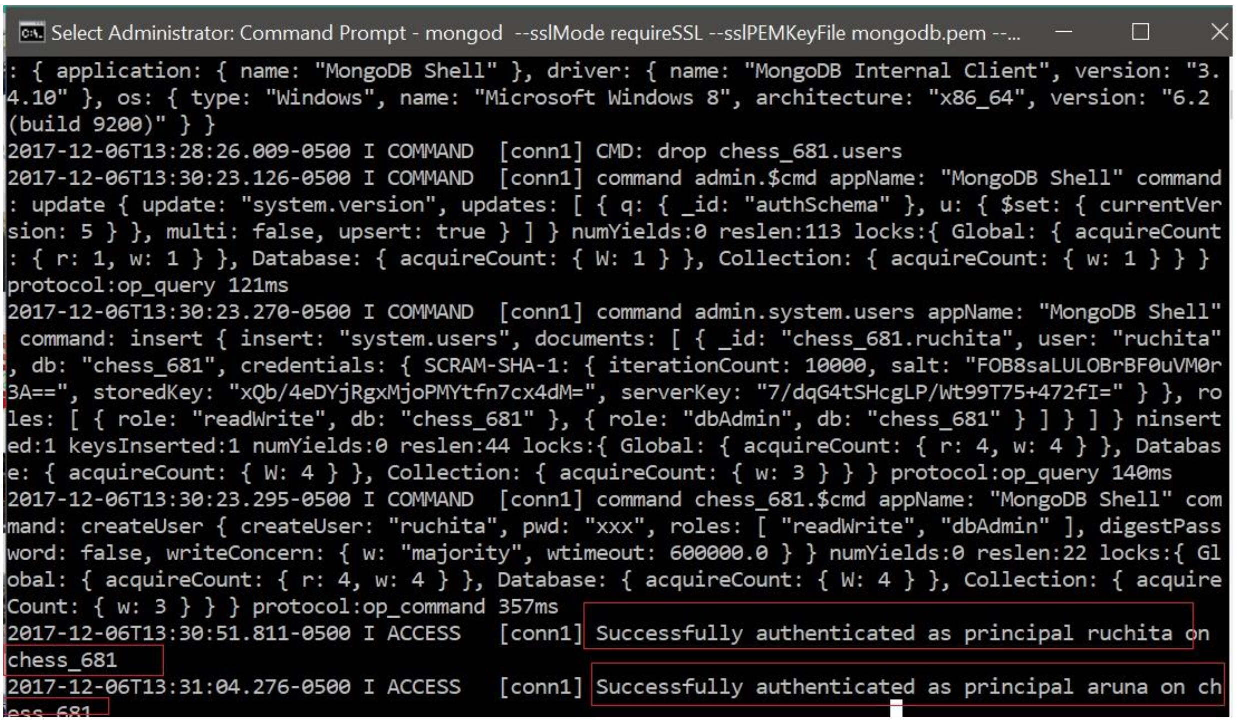
5.4.3. Database Authentication

Database authentication is the process or act of confirming that a user who is attempting to log in to a database is authorized to do so, and is only accorded the rights to perform activities that he or she has been authorized to do [7]. We created 2 users for our database with admin and read and write rights. This ensured that nobody without the credentials will be able to query the database. Figures 11, 12, and 13 show screenshots show the commands and the rights of authorized users for our DB.


Administrator: Command Prompt - mongo --ssl --host 127.0.0.1 --sslCAFile mongodb.pem

```
> db.users.drop()
true
> use chess_681
switched to db chess_681
> db.createUser({user:"ruchita", pwd:"ruchita",roles:["readWrite","dbAdmin"]});
Successfully added user: { "user" : "ruchita", "roles" : [ "readWrite", "dbAdmin" ] }
> db.createUser({user:"aruna", pwd:"aruna",roles:["readWrite","dbAdmin"]});
Successfully added user: { "user" : "aruna", "roles" : [ "readWrite", "dbAdmin" ] }
> db.auth('ruchita','ruchita')
1
> db.auth('aruna','aruna')
```

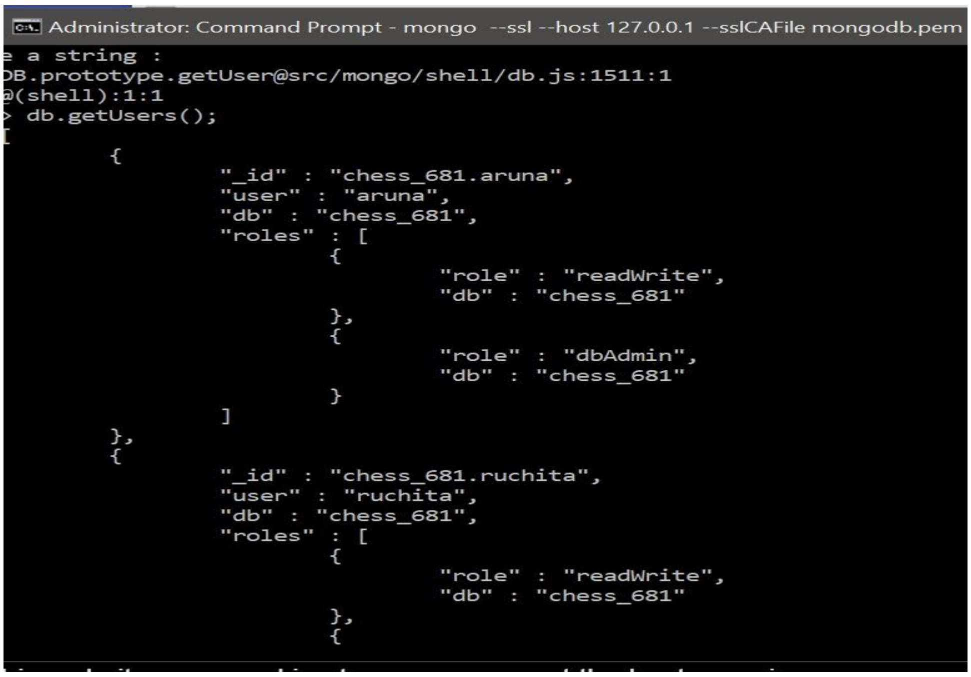
Figure 11. Creating DB Users



Select Administrator: Command Prompt - mongod --sslMode requireSSL --sslPEMKeyFile mongodb.pem --...

```
: { application: { name: "MongoDB Shell" }, driver: { name: "MongoDB Internal Client", version: "3.
4.10" }, os: { type: "Windows", name: "Microsoft Windows 8", architecture: "x86_64", version: "6.2
(build 9200)" } }
2017-12-06T13:28:26.009-0500 I COMMAND  [conn1] CMD: drop chess_681.users
2017-12-06T13:30:23.126-0500 I COMMAND  [conn1] command admin.$cmd appName: "MongoDB Shell" command
: update { update: "system.version", updates: [ { q: { _id: "authSchema" }, u: { $set: { currentVer
sion: 5 } }, multi: false, upsert: true } ] } numYields:0 reslen:113 locks:{ Global: { acquireCount
: { r: 1, w: 1 } }, Database: { acquireCount: { W: 1 } }, Collection: { acquireCount: { w: 1 } } }
protocol:op_query 121ms
2017-12-06T13:30:23.270-0500 I COMMAND  [conn1] command admin.system.users appName: "MongoDB Shell"
 command: insert { insert: "system.users", documents: [ { _id: "chess_681.ruchita", user: "ruchita"
, db: "chess_681", credentials: { SCRAM-SHA-1: { iterationCount: 10000, salt: "FOB8saLULOBrBF0uVM0r
3A==", storedKey: "xQb/4eDYjRgxMjoPMYtfn7cx4dM=", serverKey: "7/dqG4tSHcgLP/Wt99T75+472fI=" } }, ro
les: [ { role: "readWrite", db: "chess_681" }, { role: "dbAdmin", db: "chess_681" } ] } ] } ninsert
ed:1 keysInserted:1 numYields:0 reslen:44 locks:{ Global: { acquireCount: { r: 4, w: 4 } }, Databas
e: { acquireCount: { W: 4 } }, Collection: { acquireCount: { w: 3 } } } protocol:op_query 140ms
2017-12-06T13:30:23.295-0500 I COMMAND  [conn1] command chess_681.$cmd appName: "MongoDB Shell" com
mand: createUser { createUser: "ruchita", pwd: "xxx", roles: [ "readWrite", "dbAdmin" ], digestPass
word: false, writeConcern: { w: "majority", wtimeout: 600000.0 } } numYields:0 reslen:22 locks:{ Gl
obal: { acquireCount: { r: 4, w: 4 } }, Database: { acquireCount: { W: 4 } }, Collection: { acquire
Count: { w: 3 } } } protocol:op_command 357ms
2017-12-06T13:30:51.811-0500 I ACCESS   [conn1] Successfully authenticated as principal ruchita on
chess_681
2017-12-06T13:31:04.276-0500 I ACCESS   [conn1] Successfully authenticated as principal aruna on ch
ess_681
```

Figure 12. Authenticating DB Users



Administrator: Command Prompt - mongo --ssl --host 127.0.0.1 --sslCAFile mongodb.pem

```
e a string :
DB.prototype.getUser@src/mongo/shell/db.js:1511:1
@(shell):1:1
> db.getUsers();
[
        {
                "_id" : "chess_681.aruna",
                "user" : "aruna",
                "db" : "chess_681",
                "roles" : [
                        {
                                "role" : "readWrite",
                                "db" : "chess_681"
                        },
                        {
                                "role" : "dbAdmin",
                                "db" : "chess_681"
                        }
                ]
        },
        {
                "_id" : "chess_681.ruchita",
                "user" : "ruchita",
                "db" : "chess_681",
                "roles" : [
                        {
                                "role" : "readWrite",
                                "db" : "chess_681"
                        },
                        {
```

Figure 13. Displaying authenticated users

5.5.Separate time-out for each player

Each player gets 300 seconds i.e. 5 min to play for each of his turns. If he doesn’t make a move within the time the opponent automatically wins by a time-out. This prevents the game from going on forever, and makes it fair-play.

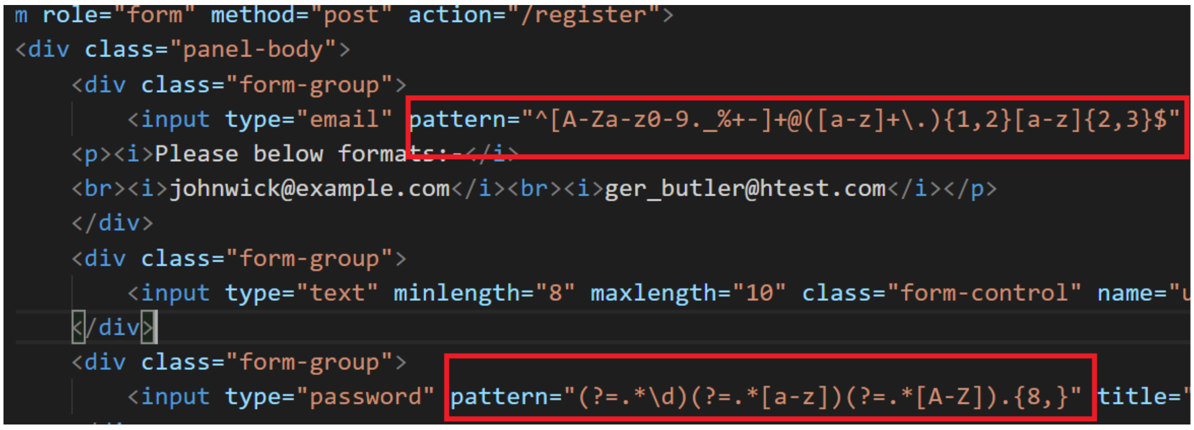
5.6. Input Validation

5.6.1. Client-side input validation

This program performs rigorous input validation. We believe client-side input validation is important to aid the user and to provide ease-of-use for the user. We believe, the easier an application is to use, the more likely is the user to follow security norms and take part of his security responsibility in his own hands. The following fields are taken as string input from the user in case of new registration:

Whitelisting of email field

- Email – The entered email should match the regex pattern provided on the server side. We have used regular expressions on our client side as shown below:



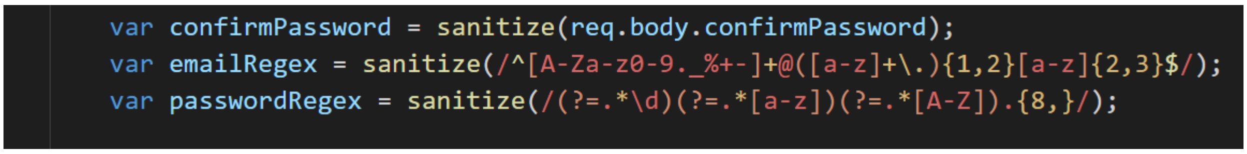
```
m role="form" method="post" action="/register">
<div class="panel-body">
    <div class="form-group">
        <input type="email" pattern="^[A-Za-z0-9._%+-]+@([a-z]+\.){1,2}[a-z]{2,3}$"
    <p><i>Please below formats:-</i>
    <br><i>johnwick@example.com</i><br><i>ger_butler@htest.com</i></p>
    </div>
    <div class="form-group">
        <input type="text" minlength="8" maxlength="10" class="form-control" name="u
    </div>
    <div class="form-group">
        <input type="password" pattern="(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}" title="
```

Passwords

- Maximum length of the password can be 15 characters.
- Confirm password – This field confirms that the password you have entered the correct password that you wanted to enter. Since we are not displaying the password, having the confirm password field becomes a required security feature.

5.6.2. Server-side input validation

Layer 1: We perform server-side validation using regular expressions before passing any data to MongoBD queries in our JavaScript files which handle post requests from the router.



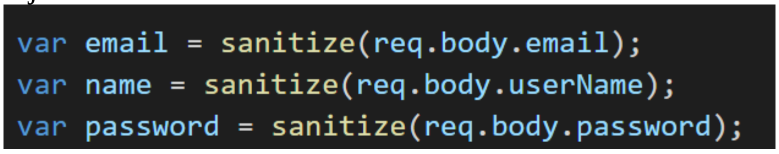
```
var confirmPassword = sanitize(req.body.confirmPassword);
var emailRegex = sanitize(/^[A-Za-z0-9._%+-]+@([a-z]+\.){1,2}[a-z]{2,3}$/);
var passwordRegex = sanitize(/(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}/);
```

5.6.3. Using mongoose functions instead of complete queries

Layer 2: Query-replicating pre-existing mongoose functions as available in mongoose documentation are used to query the database, instead of writing complete queries.
For example: `findOneAndUpdate(req.user.id)`

5.6.4. Mongo-sanitize package

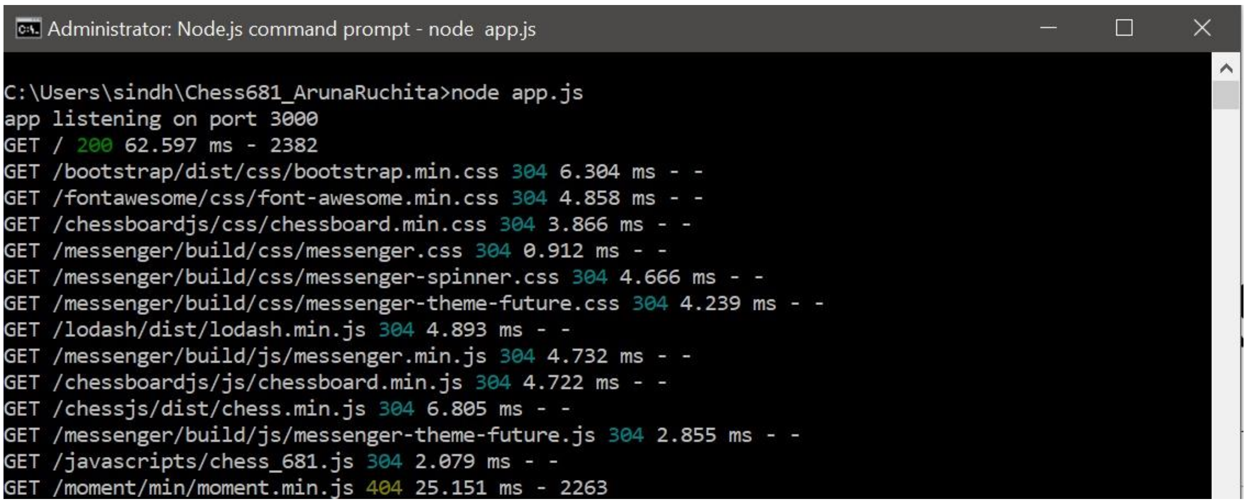
Layer 3: We are using an npm package called `mongo-sanitize` which is applied to each function which queries the database. This module searches for any keys in objects that begin with a '$' sign and strips them out. Object keys starting with a '$' are reserved MongoDB as operators. Sanitizing our http requests prevents attacks like mongo injection which is a noSQL injection attack.



```
var email = sanitize(req.body.email);
var name = sanitize(req.body.userName);
var password = sanitize(req.body.password);
```

5.7.Security logging

Our application has a logging mechanism which logs all the http request events the application handles. Figure 14 shows a screenshot which shows all the logs generated while playing the game. These log files can then be analyzed for potential attackers' information. Since we are saving our passwords as salted hashes in the database, they will appear in encrypted form in logs if they have to.



```
C:\Users\sindh\Chess681_ArunaRuchita>node app.js
app listening on port 3000
GET / 200 62.597 ms - 2382
GET /bootstrap/dist/css/bootstrap.min.css 304 6.304 ms - -
GET /fontawesome/css/font-awesome.min.css 304 4.858 ms - -
GET /chessboardjs/css/chessboard.min.css 304 3.866 ms - -
GET /messenger/build/css/messenger.css 304 0.912 ms - -
GET /messenger/build/css/messenger-spinner.css 304 4.666 ms - -
GET /messenger/build/css/messenger-theme-future.css 304 4.239 ms - -
GET /lodash/dist/lodash.min.js 304 4.893 ms - -
GET /messenger/build/js/messenger.min.js 304 4.732 ms - -
GET /chessboardjs/js/chessboard.min.js 304 4.722 ms - -
GET /chessjs/dist/chess.min.js 304 6.805 ms - -
GET /messenger/build/js/messenger-theme-future.js 304 2.855 ms - -
GET /javascripts/chess_681.js 304 2.079 ms - -
GET /moment/min/moment.min.js 404 25.151 ms - 2263
```

Figure 14. Application logging mechanism

5.8.Providing Limited Information

5.8.1. Wrong password message

When a user is trying to sign in, and uses an unregistered email id to do so, or uses a wrong email or password, a message is displayed telling the user "Invalid login or password". Having such an ambiguous message prevents an attacker from getting any hints.

5.8.2. Hiding passwords

On the login page and new registration page, password field is hidden and not displayed to the user.

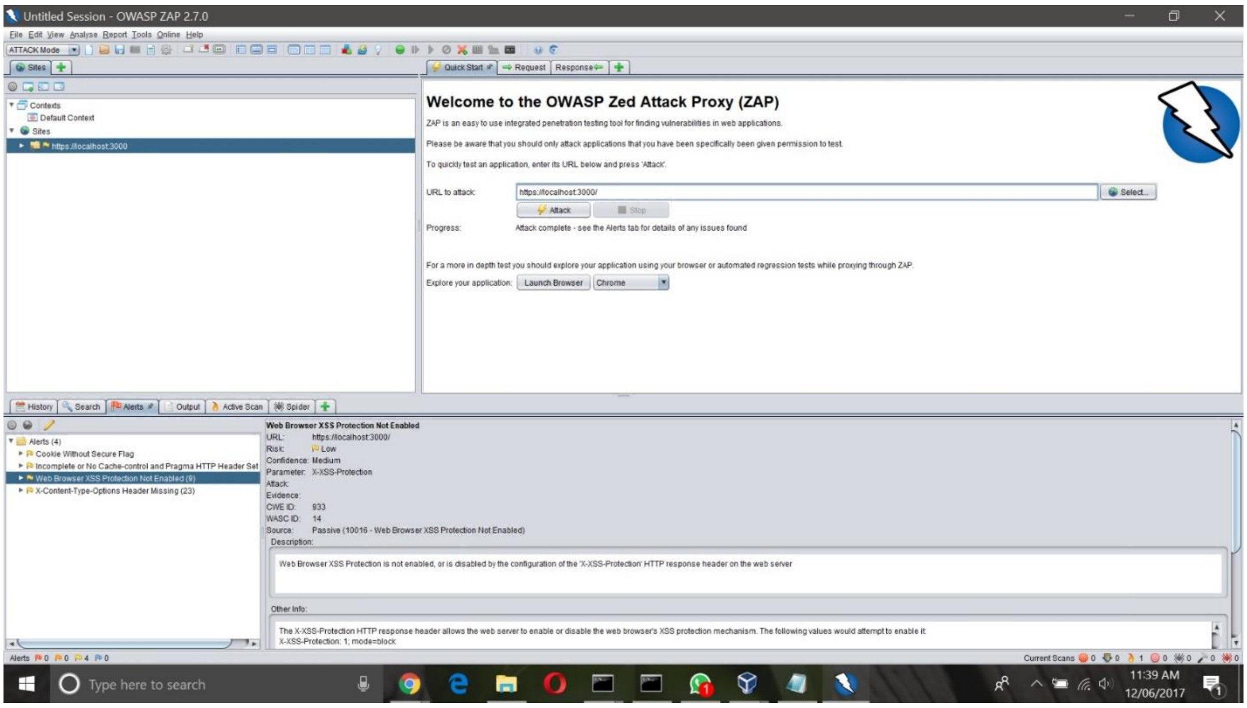
5.9.Small Attack Surface

There are just 3 places in the game where we take input from the user: Login page, register new user page, and Play game page. Of these, only Login and Register take input as string from user which has strict input validation. The third page, which lets user decide if he wants to play black or white is a radio button, and thus it provides lesser scope for an attack.
The game can only be played through the UI. Although a text box at the side lets the user view all the game moves, you cannot "type" in a chessboard position to make a move, thus significantly reducing the attack surface.

5.10. Performed System Security Verification

5.10.1. Dynamic Analysis

We performed dynamic analysis of our application using OWASP ZAP. Figure 15. shows a screenshot of the analysis the tool performed on the application.



Welcome to the OWASP Zed Attack Proxy (ZAP)

ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications.

Please be aware that you should only attack applications that you have been specifically been given permission to test.

To quickly test an application, enter its URL below and press 'Attack'.

URL to attack: https://localhost:3000/

Progress: Attack complete - see the Alerts tab for details of any issues found

For a more in depth test you should explore your application using your browser or automated regression tests while proxying through ZAP.

Explore your application: Launch Browser Chrome

Alerts (4)
- Cookie Without Secure Flag
- Incomplete or No Cache-control and Pragma HTTP Header Set
- Web Browser XSS Protection Not Enabled (9)
- X-Content-Type-Options Header Missing (23)

Web Browser XSS Protection Not Enabled

URL:	https://localhost:3000/
Risk:	Low
Confidence:	Medium
Parameter:	X-XSS-Protection
Attack:	
Evidence:	
CWE ID:	933
WASC ID:	14
Source:	Passive (10016 - Web Browser XSS Protection Not Enabled)

Description:

Web Browser XSS Protection is not enabled, or is disabled by the configuration of the 'X-XSS-Protection' HTTP response header on the web server

Other Info:

The X-XSS-Protection HTTP response header allows the web server to enable or disable the web browser's XSS protection mechanism. The following values would attempt to enable it:
X-XSS-Protection: 1; mode=block

Alerts 0 0 4 0 Current Scans 0 0 1 0 0 0 0

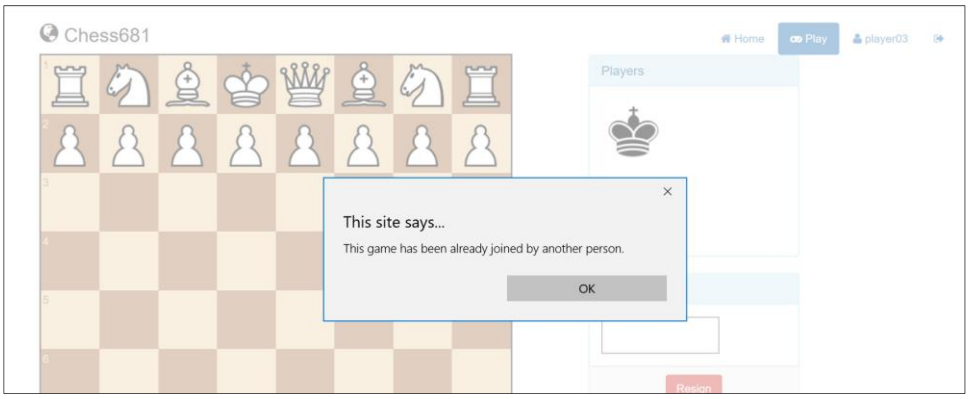
Figure 15. OWASP ZAP

Making the application secure was more of an iterative and a recursive process. We run the OWASP-ZAP, and as we get issues we had tried to solve the issues it shows, and kept running it after each change iteratively. One of the most common attacks this tool showed was SQL injections. That explains the 3-layer security we have for input validation.

5.10.2. Manual Testing

- **Security against cheating players**

This application has been designed in such a way that once a game room fills up with exactly 2 players, a third player will not be able to use the URL to join the game, nor will he be able to see a game in progress as shown below:



This site says...

This game has been already joined by another person.

OK

- **External dependencies have low security vulnerabilities**
    - The main external packages we are using are chessboardjs and chessjs. We have manually played the game many times, with many different combinations of game

moves to check if there are vulnerabilities in that code, and we found none. We are using the latest version of these libraries. Thanks to [1] and [2] again for chess libraries.

- The other external packages used in our project are node.js, mongoDB, and all other npm utility packages. We have made sure that we are using the latest, most updated versions of all of these. Additionally, these are all open-source, and the last updates for most of them have happened in the recent weeks/months.

5.11. Preventing URL attack

All the URLs ever accessed by all players are only generated randomly using the randomString function in utils.js. The allowed set of characters are only alphabets and digits as mentioned in util.js. This prevents an attack like "http://webserver/cgi-bin/phf?Qalias=x%0a/bin/cat%20/etc/passwd". From [3] I would like to mention that this concept was mentioned in PHE case study in lecture slides in 'Lecture 2 Slide 5" which helped us consider this perspective while testing our website with OSWASP-ZAP.

5.12. Preventing buffer overflows with the choice of language

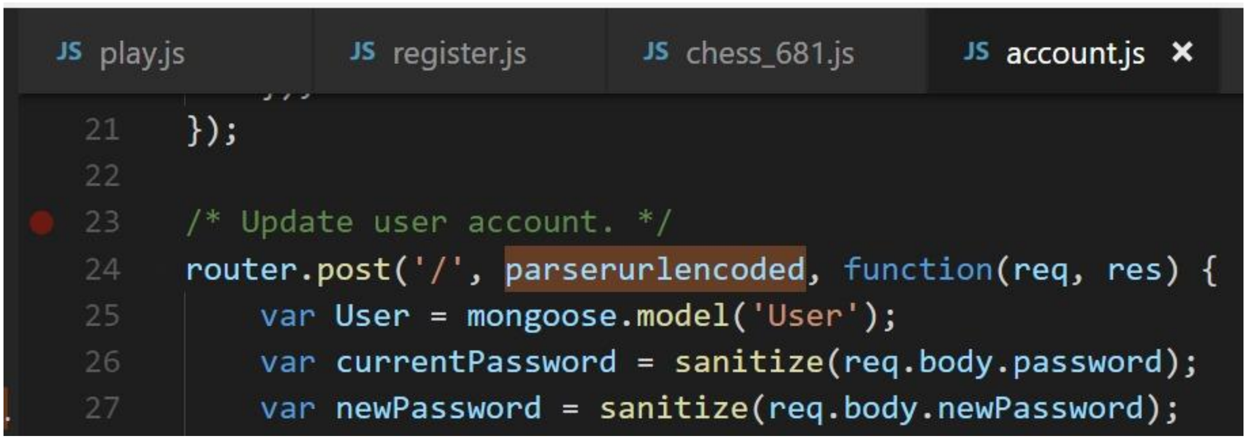
JavaScript doesn't have any raw arrays, everything is a managed object. There are no pointers or ways of addressing outside of an object or calling functions that can write past the end of an object.
Thus, we chose JavaScript as it is a good choice of language in preventing buffer overflows**.**

5.13. CSRF Protection

CSRF stands for Cross-site Request Forgery. Since we are using node and express, we are using body-parser to parse JSON in the following way [8][9]:

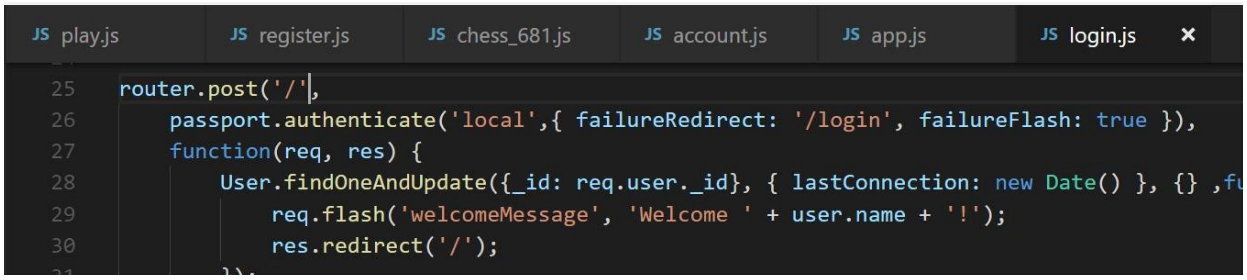
```
var bodyParser = require('body-parser');
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({
    extended: true
}));
```

This generates the need for having CSRF protection because currently, our API can be called from either a JavaScript XHR request or a traditional HTML form. This means an attacker could create a malicious HTML form with form-encoded data and submit to our API.
Solution:
We added body parser specifically to the routes that need them, and removed them from all other middleware API routes.
Consider Figures 16 and 17 as examples of this.



```
});

/* Update user account. */
router.post('/', parserurlencoded, function(req, res) {
    var User = mongoose.model('User');
    var currentPassword = sanitize(req.body.password);
    var newPassword = sanitize(req.body.newPassword);
```

Figure 16. parseurlencoded added where required



```
router.post('/',
    passport.authenticate('local',{ failureRedirect: '/login', failureFlash: true }),
    function(req, res) {
        User.findOneAndUpdate({_id: req.user._id}, { lastConnection: new Date() }, {} ,fu
            req.flash('welcomeMessage', 'Welcome ' + user.name + '!');
            res.redirect('/');
```

Figure 17. Body parser removed from all other routes

Bibliography

[1] http://chessboardjs.com/
[2] https://github.com/jhlywa/chess.js
[3] Course Work slides – Dr David A Wheeler
[4] https://github.com/coreinfrastructure/best-practices-badge/blob/master/doc/security.md
[5] https://justin.kelly.org.au/how-to-create-a-self-sign-ssl-cert-with-no-pa/
[6] https://blog.risingstack.com/node-hero-node-js-authentication-passport-js/
[7] https://www.techopedia.com/definition/27388/database-authentication
[8] https://fosterelli.co/dangerous-use-of-express-body-parser
[9] https://github.com/expressjs/body-parser#express-route-specific
[10] https://docs.mongodb.com/manual/tutorial/configure-ssl/