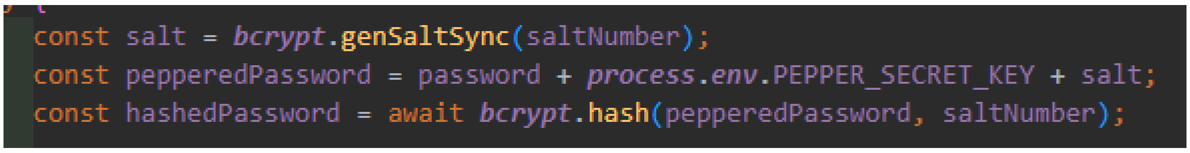
Ace Shyjan – 100390438, Mateusz Gorecki – 100392715, Troy Betts – 100395289

Hashing, Salt and Peppering

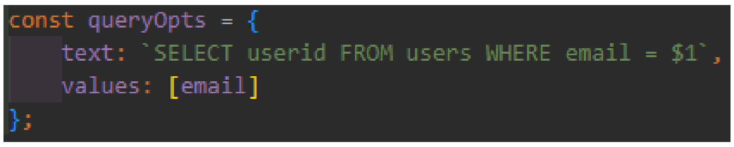
We create a salt value using bcrypt and pepper the password using a server-side SECRET key and finally hash the salt + peppered password.



```
const salt = bcrypt.genSaltSync(saltNumber);
const pepperedPassword = password + process.env.PEPPER_SECRET_KEY + salt;
const hashedPassword = await bcrypt.hash(pepperedPassword, saltNumber);
```

SQL Injection

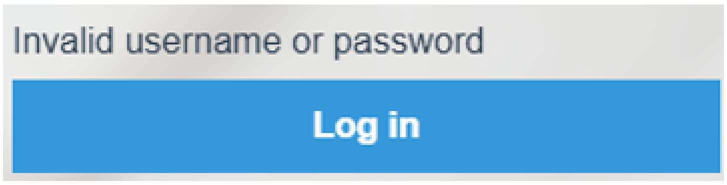
We handle SQL Injection by using parameterized queries rather than string concatenation.



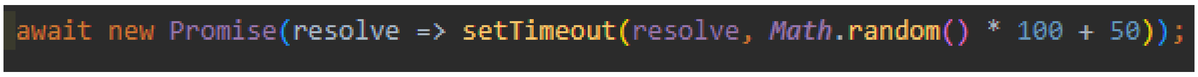
```
const queryOpts = {
    text: `SELECT userid FROM users WHERE email = $1`,
    values: [email]
};
```

Account Enumeration

Account Enumeration is handled through 2 steps, we first use generic responses for account login, and then we add random delays to avoid varied response times based off account existence.



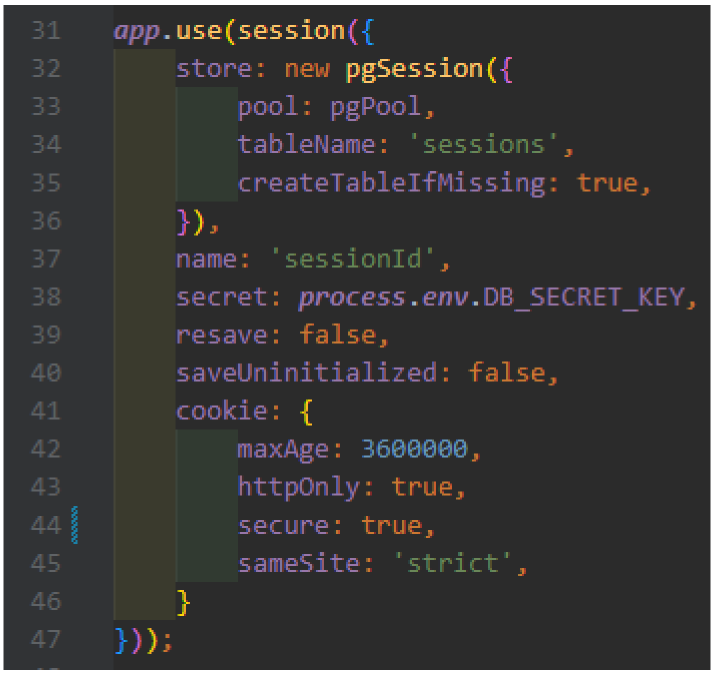
Invalid username or password



```
await new Promise(resolve => setTimeout(resolve, Math.random() * 100 + 50));
```

Session Hijacking and Cross-Site Request Forgery

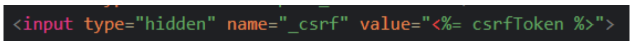
We prevent Session Hijacking using secure cookies with `httpOnly: true`, `secure: true`. We use Anti-CSRF tokens on all requests which can modify states, such as POST, PET, DELETE requests. These tokens are verified on the server side to ensure that the request came from a legitimate user. The cookie also has sameSite: strict'` enabled, meaning cookies are only sent in same-origin requests.



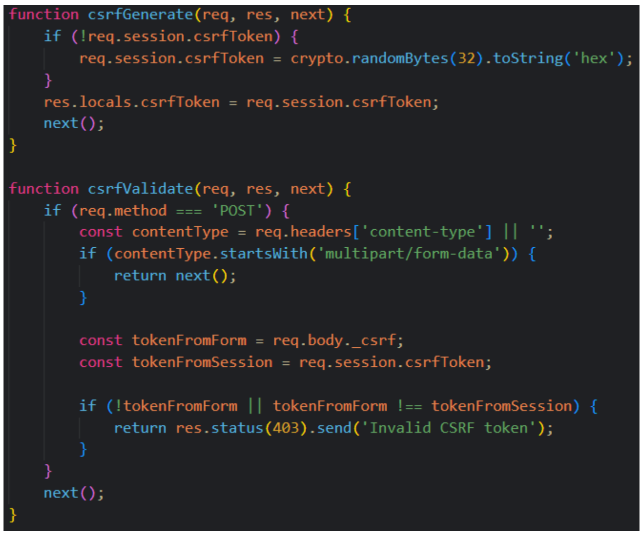
```
app.use(session({
    store: new pgSession({
        pool: pgPool,
        tableName: 'sessions',
        createTableIfMissing: true,
    }),
    name: 'sessionId',
    secret: process.env.DB_SECRET_KEY,
    resave: false,
    saveUninitialized: false,
    cookie: {
        maxAge: 3600000,
        httpOnly: true,
        secure: true,
        sameSite: 'strict',
    }
}));
```

Ace Shyjan – 100390438, Mateusz Gorecki – 100392715, Troy Betts – 100395289

Whenever a session is created, a random csrf token is generated using the crypto module. This is then stored as a hidden value in every form that uses the post method. On the server side the token is then validated after every post request and compared to the token stored in the session data.



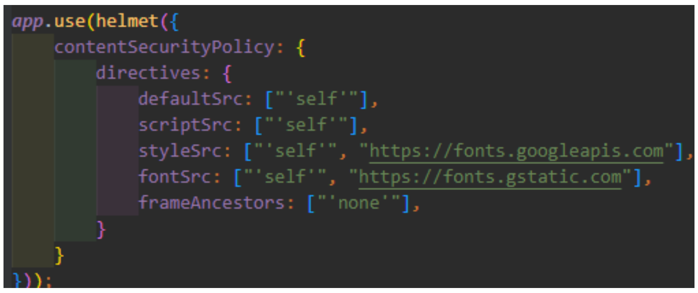
```
<input type="hidden" name="_csrf" value="<%= csrfToken %>">
```



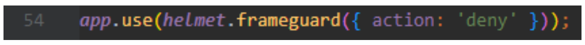
```
function csrfGenerate(req, res, next) {
    if (!req.session.csrfToken) {
        req.session.csrfToken = crypto.randomBytes(32).toString('hex');
    }
    res.locals.csrfToken = req.session.csrfToken;
    next();
}

function csrfValidate(req, res, next) {
    if (req.method === 'POST') {
        const contentType = req.headers['content-type'] || '';
        if (contentType.startsWith('multipart/form-data')) {
            return next();
        }

        const tokenFromForm = req.body._csrf;
        const tokenFromSession = req.session.csrfToken;

        if (!tokenFromForm || tokenFromForm !== tokenFromSession) {
            return res.status(403).send('Invalid CSRF token');
        }
    }
    next();
}
```

Cross-Site Scripting and Clickjacking

We prevent Cross-Site Scripting by using a good Content Security Policy to only allow constant from the same origin and trusted script sources. We prevent Clickjacking by setting the X-Frame-Options header to DENY, which blocks the embedding of the site any frame, regardless of the origin. This is done using the helmet module.



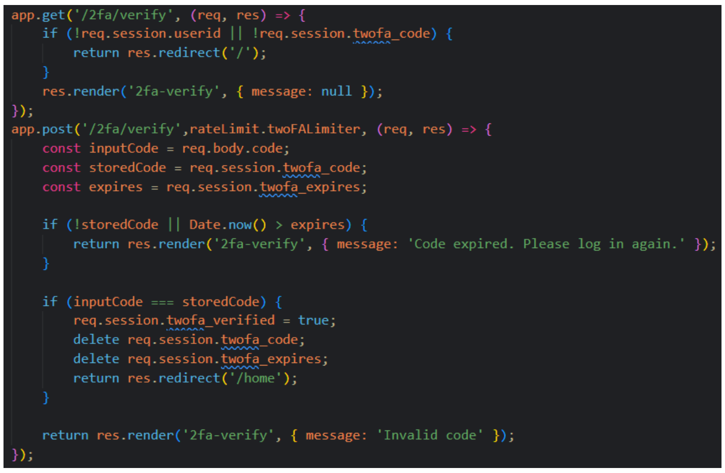
```
app.use(helmet({
    contentSecurityPolicy: {
        directives: {
            defaultSrc: ["'self'"],
            scriptSrc: ["'self'"],
            styleSrc: ["'self'", "https://fonts.googleapis.com"],
            fontSrc: ["'self'", "https://fonts.gstatic.com"],
            frameAncestors: ["'none'"],
        }
    }
}));
```



```
app.use(helmet.frameguard({ action: 'deny' }));
```

2 Factor Authentication

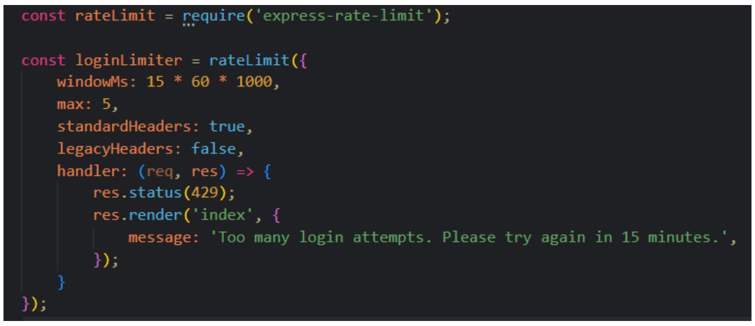
We use email 2fa authentication whenever the user logins, this generates a random 6 digit number stored server side and sent to the users email. It is set to expire after 5 minutes.



```
app.get('/2fa/verify', (req, res) => {
    if (!req.session.userid || !req.session.twofa_code) {
        return res.redirect('/');
    }
    res.render('2fa-verify', { message: null });
});
app.post('/2fa/verify',rateLimit.twoFALimiter, (req, res) => {
    const inputCode = req.body.code;
    const storedCode = req.session.twofa_code;
    const expires = req.session.twofa_expires;

    if (!storedCode || Date.now() > expires) {
        return res.render('2fa-verify', { message: 'Code expired. Please log in again.' });
    }

    if (inputCode === storedCode) {
        req.session.twofa_verified = true;
        delete req.session.twofa_code;
        delete req.session.twofa_expires;
        return res.redirect('/home');
    }

    return res.render('2fa-verify', { message: 'Invalid code' });
});
```

Rate Limiting

Using the express rate limit module, we set a limit on how many requests a user can send, which after 5 attempts blocks the user for 15 minutes. A similar thing is done for 2fa requests. Helps prevents from DDOS attacks.



```
const rateLimit = require('express-rate-limit');

const loginLimiter = rateLimit({
    windowMs: 15 * 60 * 1000,
    max: 5,
    standardHeaders: true,
    legacyHeaders: false,
    handler: (req, res) => {
        res.status(429);
        res.render('index', {
            message: 'Too many login attempts. Please try again in 15 minutes.',
        });
    }
});
```