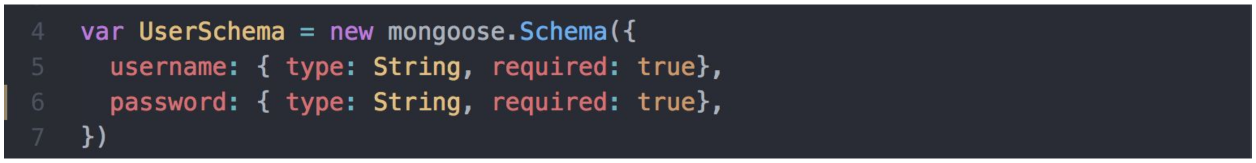
Kelly Lunghamer
kgl277

Assignment 2

Link to project: https://github.com/klunghamer/Application-Security-Project

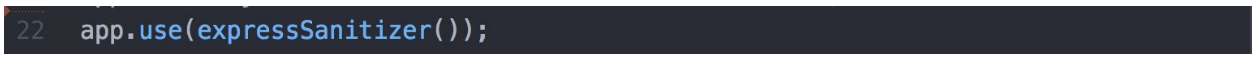
The primary way that I made this web app resilient against common web vulnerabilities is through the user text input and file upload.

I used MongoDB for this project, which is not vulnerable to SQL injections, but is still vulnerable to other injections. To prevent attackers from inputting {$ne:null} in the username and password fields or similar attacks, I configured Mongoose to only accept strings as input.

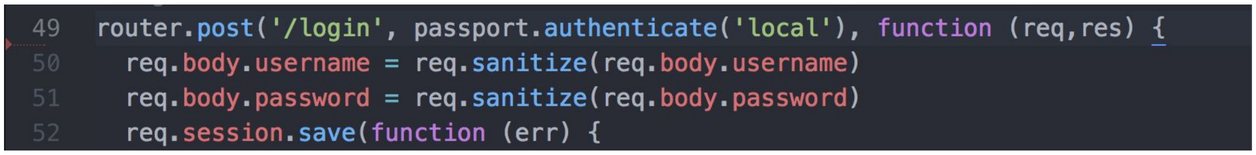


```
var UserSchema = new mongoose.Schema({
  username: { type: String, required: true},
  password: { type: String, required: true},
})
```

Another preventative measure before Mongoose switched all input to strings was to use express-sanitizer middleware to remove all script tags and other javascript DOM methods, such as alert(). This should prevent any cross-site scripting and also any attempts for cookie theft. Here is an example on the login route:

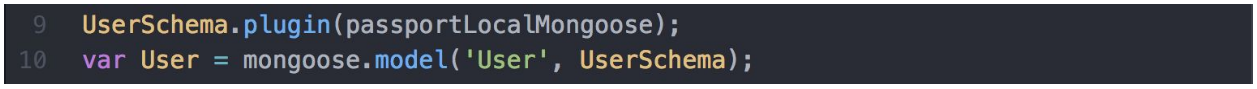


```
app.use(expressSanitizer());
```

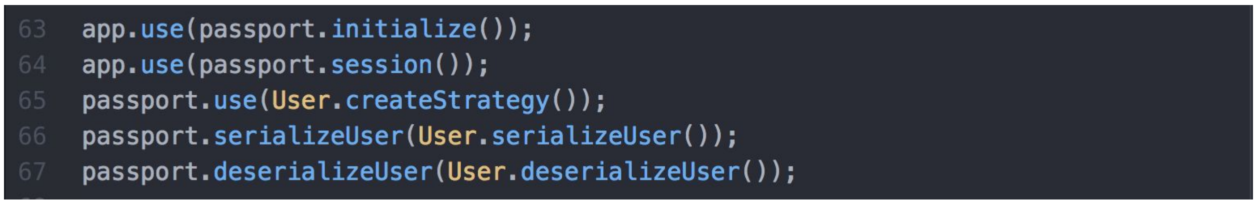


```
router.post('/login', passport.authenticate('local'), function (req,res) {
  req.body.username = req.sanitize(req.body.username)
  req.body.password = req.sanitize(req.body.password)
  req.session.save(function (err) {
```

To protect passwords in case database data is stolen, each password is salted and hashed using passport-local-mongoose, which is an add-on to passport for session support. Code snippets to show implementation below:

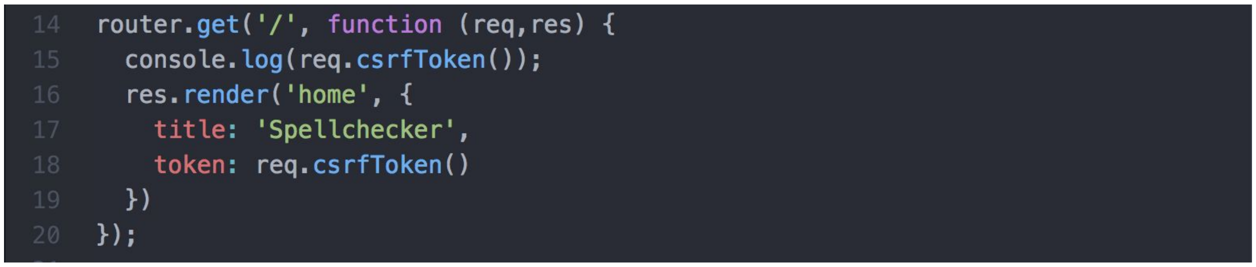


```
UserSchema.plugin(passportLocalMongoose);
var User = mongoose.model('User', UserSchema);
```

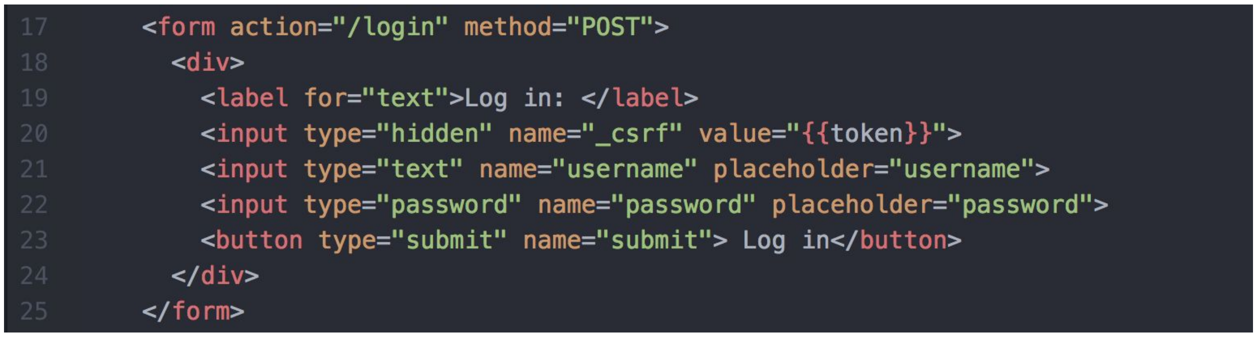


```
app.use(passport.initialize());
app.use(passport.session());
passport.use(User.createStrategy());
passport.serializeUser(User.serializeUser());
passport.deserializeUser(User.deserializeUser());
```

To prevent cross-site request forgery, I used express-session and csurf to create tokens for the GET methods which valid POST requests on all forms. For example when the user is at the home screen, a token is created using req.csrfToken() and placed as the hidden value on the login form:

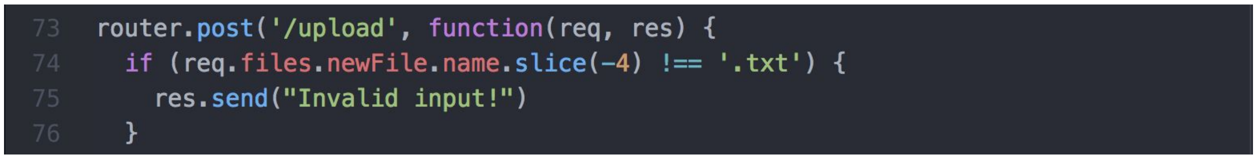


```
router.get('/', function (req,res) {
  console.log(req.csrfToken());
  res.render('home', {
    title: 'Spellchecker',
    token: req.csrfToken()
  })
});
```



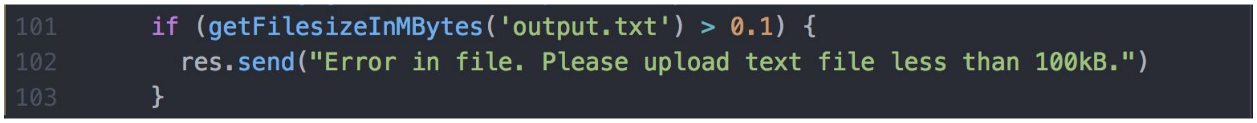
```
    <form action="/login" method="POST">
      <div>
        <label for="text">Log in: </label>
        <input type="hidden" name="_csrf" value="{{token}}">
        <input type="text" name="username" placeholder="username">
        <input type="password" name="password" placeholder="password">
        <button type="submit" name="submit"> Log in</button>
      </div>
    </form>
```

Lastly, I validated the user input of the text file for the spellchecker. If the user inputs a file without the .txt extension, they are unable to upload.



```
router.post('/upload', function(req, res) {
  if (req.files.newFile.name.slice(-4) !== '.txt') {
    res.send("Invalid input!")
  }
```

I also check the file size of the text file in case the user changes the extensions to trick the previous validation. I only allowed files less than 100kB to be processed.



```
    if (getFilesizeInMBytes('output.txt') > 0.1) {
      res.send("Error in file. Please upload text file less than 100kB.")
    }
```