

# IERG 4210 Tutorial 08

Securing web page (II):

- In principle: Cookie related security issues
- In practice: Point by point checklist for Phase 4A

Shizhan Zhu

# Logistics

- Content for today:
  - Provide sample codes for most check points specified in Phase 4A.
  - Cover the principle interpretation of cookie related security issues (CSRF, session maintenance, authentication remember, etc.).
- This tutorial only provides one type of implementation.
  - There is no necessity to completely follow the tutorial if you think you are a strong student. Your creativity is strongly encouraged. A good usage of this tutorial tends to be like this: you only refer to the part where you cannot figure out.
  - If you choose to follow, then thinking while copying.
  - There might exist somewhere not suitable to your current project, you need to modify and debug yourself.
  - Please follow tutorial 8, if there exists differences between tutorial 7 and 8.
  - Logistics on submission (branch info and README) please refer to Page 18 of tutorial 7.

# Project structure review

Better to do the modularizing,  
though a very long app.js also  
works.

- App/server.js: entry of project (other name needs specifying in package.json for eb's reference).
- Shop\*\*\*.config.js: configuration for database connection, etc. (Optional)
- Public/ : contain client-side elements: css style sheets / image source / form handling javascript.
- Views/ : html templates if you use handlebars.
- Routes/ : server side node scripts under express routing.
- Node\_modules/ : Your installed off-the-shelf packages, ignored by git.
- \*\*\*.sql : (Optional) better include this initial database generating script file so that TA is able to run your codes locally, maybe to do some modifications.

# Task 5.1 Create user table

- Here password column refers to two elements: salt and salted password (or you can specify same salt for all users).
- In sample codes, there are two users: shopadmin (password: 123456) and shopcommon (password: 234567). (In your implementation please rename the username and password. Better use email add for username.)
- In sample codes each user is assigned a different salt. It is your choice whether using a unique salt throughout or do it like I do.
- This step has nothing to do with the project, but in the offline fashion. But you need your project configuration to generate the salt and salted password.

# Task 5.1 Create user table

- Create database: in mysql console type

```
CREATE TABLE users;
```

- Write a simple script (put it anywhere existing node configure, e.g. the root directory of your project) foo.js:

→\_→

- You need to npm install crypto package.
- You don't need to submit this source file as source code.

foo.js

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

function hmacPassword (password)
{
  var salt = 'as3qw4taegtgew5t4';
  var hmac =
  crypto.createHmac('sha256', salt);
  console.log(salt); // zhu
  hmac.update(password);
  return hmac.digest('base64');
}

console.log(hmacPassword('123456'))
```

# Task 5.1 Create user table

- For each user, you just input different user plain text password and run it: `node foo.js`, to see the result: two lines, one for salt and one for salted password.
- How to generate salt? You can randomly generate using program, or a much simpler method: turn to [random.org](http://random.org) for salt generation.
- Now you can insert new user record:

```
CREATE TABLE `users` (  
  `uid` int(11) NOT NULL AUTO_INCREMENT,  
  `username` varchar(512) NOT NULL,  
  `salt` varchar(512) NOT NULL,  
  `saltedPassword` varchar(512) NOT NULL,  
  `admin` int(1) DEFAULT NULL,  
  PRIMARY KEY (`uid`)  
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;
```

```
INSERT INTO `users` VALUES  
(1, 'shopadmin', 'as3qw4taegtgew5t4', 'TqgwkZHcXmM+yt7zqRjxy5WBgeZEfp1Yq2agUI6ppwI=', 1), (2, 'shopcommon', 'asgfpegb34qwhehesbsb', 'XFeZwHpF6nWyt/3DTZWQLj4pAc9wVf3puL76gsG/nvg=', 0);
```



# Task 5.1 Create user table

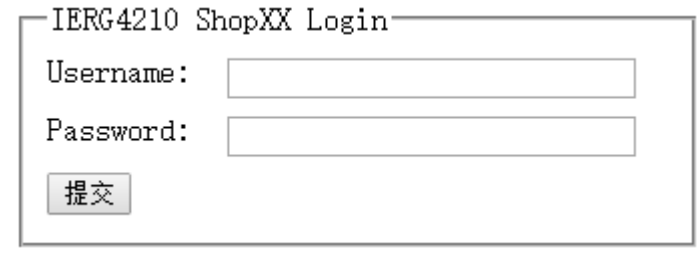
- Your users table in the database should look like this:

```
+-----+-----+-----+-----+
| uid | username | salt | saltedPassword | admin |
+-----+-----+-----+-----+
| 1 | shopadmin | as3qw4taegtgew5t4 | TqgwkZHcXmM+yt7zqRjxy5WBgeZEfp1Yq2agUI6ppwI= | 1 |
| 2 | shopcommon | asgfpegb34qwehesbsb | XFeZwHpF6nWyt/3DTZWQLj4pAc9wVf3puL76gsG/nvg= | 0 |
+-----+-----+-----+-----+
```

- Do not directly copy my salt and password. Otherwise -> you might face troubles!
- If you want to add features like changing password via email in the future, I suggest that you force the username to be the email address (which coincidence with the lecture notes), and the username type becomes 'email' (next page).
- Suggest to restrict the username or email to be unique.

# Task 5.2 login page

- Create a new html template in `views/`: You may name it as `login.handlebars`. Do some decoration.
- Make sure it has a form containing username input (or email add input) and plaintext password input, and the submit button.
- The simplest views looks like this:



```
IERG4210 ShopXX Login
Username: 
Password: 

```

- It seems the submit button does not need to implement on your own. Why?
- Hints: `<div><input type="submit" /></div>` is enough for the button.
- Similarly, use `type="text" / "email"` and `type="password"` .



# Task 5.2 login page

- Client side input validation, for user friendliness purpose.
- For example, if you want the password only matches such pattern: maximum length 512, only contain digits ranged between 20 to 10,000,000 (occurrence between 6 and 512)

- You can use:

```
<div><input type="password" name="password" required maxlength="512" pattern="^\x20-\x7E]{6,512}$" title="Invalid Password" /></div>
```

- Notice the regexp, refer to [http://www.w3schools.com/jsref/jsref\\_obj\\_regexp.asp](http://www.w3schools.com/jsref/jsref_obj_regexp.asp)
- Similarly for the username (which could contain word besides digits), you can use: `pattern="^\w- ']{4,512}$"`

# Task 5.2 login page

- You can change the pattern as you like, e.g. also permit word characters in password. (pure digit password could be hacked with brute-force)
- The most important prevent of SQL injection is actually the prepare statement, demonstrated later.
- Now comes the exciting part: form action.
- `<form method="POST" action="api/login"> ... </form>`
- You need to use POST as method instead of GET, since this submission contains sensitive info (password) and should be transparent to users.
- We need to implement the action function.
- By the way do you know how to handle form submission in the express framework?

# Task 5.2 login, server side action function

- Under this framework, server side action function are implemented and stored in the routes/ directory.
- Inside the functions are grouped like this:

```
• // Global variables:  
• Var config = require(...), ...;  
• // Global functions:  
• Function hmacPassword(...) {  
•     ...  
• }  
• // functions for actions  
• Module.exports(pool, path) {  
•     var app = express.Router();  
•     app.use(...) / get(...) / post(...) {  
•         ...  
•     }  
• }  
• }
```

# Task 5.2 login, server side action function

- As we are developing login authentication functionality, we create a new file `auth.api.js` (or anything else you like).

## `Auth.api.js`

```
// TODO: require your needed packages, define your
global functions (e.g. hmacPassword) here.
// functions for actions
Module.exports(pool, path) {
  var app = express.Router();
  app.post('/api/login',function (req,res){
    // TODO:
    // 1. Input validation / sanitization
    // 2. Quit if input invalid
    // 3. Query database with prepare ...
    // 3.1 if error , then ...
    // 3.2 if no record, then ...
    // 3.3 if OK, then ...

  });
}
```

Your form action addr.

```
req.checkBody('username', 'Invalid
Username').
isLength(4, 512)
.matches(Your regexp);
```

Express-validator  
package required

```
req.checkBody('password', 'Invalid
Password').
isLength(6, 512)
.matches(Your regexp);
```

```
if (req.validationErrors()) {
  return
  res.status(400).json({'Invalid
Input': req.validationErrors()}).end();
}
```

# Task 5.2 login, server side action function

- Query the database with prepare statement (to avoid SQL injection).

```
function hmacPassword (password,salt) {
    var hmac = crypto.createHmac('sha256', salt);
    //console.log(salt); // zhu
    hmac.update(password);
    return hmac.digest('base64');
}
// 3. Query database with prepare statement
// Please note the codes posted on the lecture notes
// 7 Page 27 only uses one single salt for all users,
// which is different from my implementation.
// Sample codes see next page.
```

- Routes with /admin prefix(before /api/login), check this issue if a bug arises.
- Note it is only a part of this source file, we need to add more in later development.

# Task 5.2 login, server side action function

```
pool.query('SELECT * FROM users WHERE username = ? LIMIT 1',
  [req.body.username],
  function (error, result) {
    if (error) {
      console.error(error);
      return res.status(500).json({'dbError': 'check server log'}).end();
    }

    var submittedSaltedPassword = hmacPassword(req.body.password, result.rows[0].salt);

    //console.log(submittedSaltedPassword); //I made a mistake here and this is how to debug
    //console.log(result.rows[0].saltedPassword); // Output in the right position.

    // Didn't pass the credential.
    if (result.rowCount === 0 || result.rows[0].saltedPassword !== submittedSaltedPassword) {
      return res.status(400).json({'loginError': 'Invalid Credentials'}).end();
    }

    req.session.regenerate(function(err) {
      //The purpose for these parts of codes would be covered later.
      req.session.username = req.body.username;
      req.session.admin = result.rows[0].admin;
      res.status(200).json({'loginOK': 1}).end();
    });
  }
);
```



# Task 5.2 login, server side action function

- Have we finished? NO. You can even not able to access the login page.
- Let's have a look at how the project runs, and how the login source script takes effect in the project.

App.js

```
var app = express();
app.engine('handlebars', exphbs({defaultLayout: 'main'}));
app.set('view engine', 'handlebars');

// serve static files from the public folder
app.use(express.static('public'));

// for parsing application/x-www-form-urlencoded
app.use('/admin/api', bodyParser.urlencoded({extended:true}));
// this line must be immediately after express.bodyParser()!
// Reference: https://www.npmjs.com/package/express-validator
app.use('/admin/api', expressValidator());

// authentication routers run really first
app.use('/admin', authAPIRouter(dbPool));

// backend routers run first
app.use('/admin/api', backEndAPIRouter(dbPool));
app.use('/admin', backEndRouter(dbPool));

// frontend router runs last
app.use('/', frontEndRouter(dbPool));
```

# Task 5.2 login, server side action function

- Usually if someone wants to access the admin page, he always inputs /admin instead of /admin/login.
- As stated before, we cannot access the login page before credential validation.
- Hence, we need to redirect unauthorized admin page, and always render the login page at initial state, and authentication failure state.

```
... Auth.api.js
Module.exports = function (pool, path) {
  var app = express.Router();
  console.log('login:A');
  // TODO: path add prefix '/admin'
  // TODO: use session (discussed later)

  app.get('/login', function (req, res) {
    // TODO: render login page
    console.log('login:B');
  });
  app.post('/api/login', function (req, res) {
    console.log('login:C');
    // TODO: I have shown in Page 12-14
  });
  app.use('/', function (req, res, next) {
    console.log('login:D');
    // TODO: if OK, then next route (admin)
    // otherwise back to the login page
  });
}
```

Covered later.

Do it yourself.

Covered later.

# Task 5.2 login, server side action function

- Notice the console output ABCD, can you guess what would be outputted at each of following moment in the procedure:
- You just node app.js?                      Output:                      login:A
- You visit /admin?                      Output:                      login:D and login:B
- You refresh the page?                      Output:                      login:B
- You type in with a wrong credential?      Output:                      login:C and login:B
- You then type in the correct one?      Output:                      login:C and login:D
- Why?

# Task 5.3 Session management

- Actually your authentication is recorded via the session management.
- Hence, your login implementation involves session management.
- Also in order to remember authentication, we apply cookie manipulation.
- Since we are using the express-session framework, I would like to recommend you have a deep reading on the documentation <https://github.com/expressjs/session>
- Answers are mostly covered in the documentation.

# Task 5.3 session management

- Session handler configuration.

```
app.use(session({
  name: // set your cookie name
  secret: // similar to how you generate salt
  resave: false,
  saveUninitialized: false,
  cookie: { path: path, maxAge: 1000*60*60*24*3,
  httpOnly: true }
}))
```

- In page 14, after authenticating the credential (correct case), we store the info in session object, which would be used when redirect to the admin page.
- The regenerate function is to avoid session fixation vulnerability. (Always change session id when reach credential validation part)

# Task 5.4 Validate the token

- Steps in 5.3 only validate the credential (which earns a session token), but we haven't validate the token itself. Maybe the token isn't desired by the current user.
- In page 16, the last part of the code -> where we do the token validation.

```
app.use('/', function (req, res, next) {
  if (req.session && req.session.admin)
    return next();
  return req.xhr ?
res.status(400).json({'loginError': 'Session
Expired'}).end() : res.redirect('/admin/login');
}); // This defines a response to the /admin request.
// next: You have implement another routes response
for /admin in Phase 3. Here 'next' just calls for
that function (implemented in backend.js).
// Hence the running order defined in app.js (Page 15)
is rather crucial.
```



# Task 5.5 log out feature

- If everything goes smoothly, you should now be able to login to see your lovely admin page again!
- Congrats!
- Idea of implementing the logout feature: (suppose happened in admin page)
- In admin page, add a form (e.g. only a button), whose action function is defined in another routes, called `/admin/logout`.
- Implement the action function, destroy the session and redirect to the `/admin/login` page function (the one do the login page render work).
- You may want to refer to the `express-session` documentation to find how to destroy session.

# Task 1-4

- For task 1-4, first you need to do a global check and modification on your project.
- Task 1: (Non-specific check) **ALL** input form content restriction, server side content sanitization.
- Task 2: Put **ALL** sql query into prepare statement.
- Task 3: Apply the csrf package, covered in next page. Better first read the documentation: <https://www.npmjs.com/package/csrf>
- Task 4: Avoid global variable. More precisely, all user-specific data must not appear in global variables.
  - Specifically be careful in function, don't miss 'var', otherwise becomes global.

# Task 3 Preventing CSRF using csrf

- For all the form – action function pair, this involves two changes.
- In the form, add a new line for receiving the hidden nonce from server:

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

- When submitting the form, the received hidden nonce are also submitted.
- In the action function, two functions are involved:
  - The one do the render work, add a new csrf object as function param, and inside the function, pass it to the handlebar: `res.render('send', { csrfToken: req.csrfToken() })`
  - The one do the credential validation, add a new csrf object as function param to receive the nonce. The checking process is automatically done by the package.

# Task 1 Context-dependent output sanitization

- One more thing, besides some general checking on each form and server side sanitization, you also need to perform context-dependent sanitization.
- You don't need to do the actual implementation since packages have been there for you.
- Do some slight modification, e.g. the way to require package, to achieve this goal.

# Interactive Q&A session for phase 4A

- Thanks you!