# IERG 4210 Tutorial 07

Securing web page (I): code injection / authentication

Shizhan Zhu

# Content for today

- Phase 4 preview
  - From now please pay attention to the security issue of your website
  - This tutorial focuses on preventing injection, and authentication issues (2nd part of Lecture 7). For session storage issue we would cover in later tutorial.
- Authentication of admin user
  - introduction to Hash function - Avoid plain-text password storage

- Interactive Q&A session for previous phases
  - If you still have difficulties on content for previous content, please refer to:
    - W3school: very detailed tutorials on HTML / CSS / JavaScript and related issues like JSON / database / jQuery / Ajax … (can have a glimpse on PHP if you have spare time)
    - nodeschool: tutorial on node recommended by the official site

# What to do after phase 3?

- Our page contains possibility being attacked by various malicious activities.
- We have implemented a user page and an admin page. However, anyone can visit the admin page for now.
- We need to add admin user management database (only user with special privilege could visit admin page and do operations), in which many security issues are involved.
- Use cookies to remember authentication, without security vulnerability.
- Other issues that might result in security vulnerability.

# Overview of Phase 4

- Task on securing your page away from injection

- (I) XSS injection: from user's trust on the website.
- Attackers insert client side script into web pages.
- Example: a story.
- Common behavior: Input the required form with critical chars < >…
- Solution: have a strict and thorough check throughout your web page form input.
- Alternatively forbid critical chars as input.

# Overview of Phase 4

- (II) SQL injection
- Similarly inserting characters that might change the database retrieve pattern:
- SELECT * FROM users WHERE name = '' OR '1'='1';
- Step 1: Always do pattern checking before searching the database:
- 
```
req.checkBody('username', 'Invalid Username')
                        .isLength(4, 512)
                        .matches(inputPattern);
```

Where inputPattern is a regular expression. Same thing for password.

- Step 2: Use prepare statement: (See next page)

# Overview of Phase 4

- ```
  pool.query('SELECT * FROM users WHERE username
    = ? AND password = ? LIMIT 1',
                req.body.username, req.body.password],
                function (error, result) {
  ```

…

```
}
```

Refer to Page 27 of Lecture notes 7.

A mild reminder is that, you need to replace the password into a salted one, which would be covered in later slides. Thinking while copying codes.
```

# Overview of Phase 4

- (III) CSRF vulnerabilities:
- Comes from the website's trust on the user. Cookie!
- Example: You have been authenticated by a site (info stored in cookie). Someone sends you a malicious link (which would invoke some operation like pay some money on e-bank without your approval).
- The solution toward this type of attack would be covered in next week tutorial. (it is related to session management)

# Overview of Phase 4

- Except for securing your site over above 3 issues, we also need to …
- (IV) Task on authentication – the first thing you need to do in phase 4
  - Put up a login page before the admin page
  - Do authentication in this step.
  - Avoid plain password using hash function
- (V) Session management and CSRF prevent (cookies, covered in later tutorial)
- (VI) Apply digital certificate (covered in later tutorial)

# Before working on Phase 4

Node package that might be useful to your project:

- Express (you should have used it)
- Handlebars / express-handlebars (you should have used it)
- Mysql (you should have used it)
- Body-parser
- Any-db
- Express-validator
- Express-session
- Connect-redis
- Other package listed in course / tutorial / spec …

- Always npm install, respect others' work, reduce your workload.

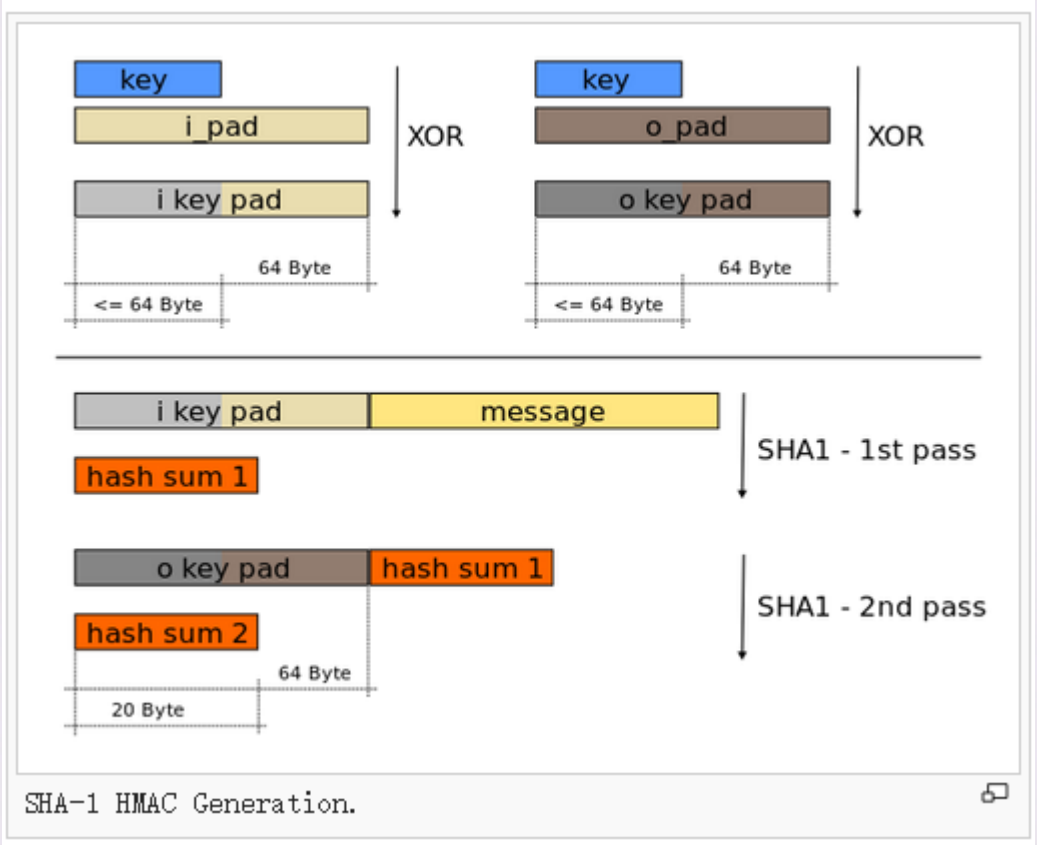
# Authentication of admin user

- Create a new table in your database stores user information.
- We need to store user’s name, email add and password.
- We cannot directly store plain text password, rather, store salted password (which can hardly be retrieved).
- You want to see what salted password looks like? If you have a Ubuntu with you, try checking: sudo vim /etc/shadow. See the thing after your account name.
- Ubuntu user password is encrypted with SHA-512, a variant of hash function method. Only brute-force method might directly hack it.

# Principle of password verification

- Server never store and never compare the original version of password as plain text (denoted as M)! Rather, it compares on $h_K(M)$, the result of the designed function h, with key K.
- Principle when designing the function:
  - 1) Easy to compute h(M)
  - 2) Computationally infeasible to find M from h(M)
  - 3) Computationally infeasible to find collision (X != Y, but h(X)=h(Y))
- Here hash function serves as a good function.

# Hash function



SHA-1 HMAC Generation.

- Please take it easy since you don’t need to implement the algorithm yourself (call it directly).
- But if you are interested you can see it in detail via http://en.wikipedia.org/wiki/HMAC

# Admin login

- In our case, M is the original password, K is called salt, and $h_K(M)$ is the salted password.
- In our database, we store salt and salted password instead of the original password.
- During comparison, we calculate the salted input using the stored salt, and compare it with the stored salted password.
- How to call the related functions?
- See next page.

# Admin login

- To call the function: `hmacPassword(req.body.password)`
- To realize this function:

```
function hmacPassword (password) {
    var hmac = crypto.createHmac('sha256', passwordSalt); // use crypto package!!!
    hmac.update(password);
    return hmac.digest('base64');
}
```

- Again, this code assume all users use the same salt. However, you are required to create and store a specific salt for each user. Try your hard to modify it!

# Admin login

- Another mild reminder: what you implement into the project, is the procedure to verify users. However, initially there is no user in your database. Hence, you need work by hand to generate salt, calculate salted password by hand and save them into the database in the offline fashion.
- You can also build a function that can add user on site, though you need to take more security issues into account…

# Add the login page

- 1. Implement all needed function in a new routes, e.g. auth.api.js
  - Salt password
  - Compare with database (mind SQL injection, also other places)
  - What if authenticated, and what if not…
- 2. Implement the login html page: login.handlebars
  - Check for the input (avoid code injection)
  - Decorate the page with other style if you like.
- Remaining issues: Remember authentication using cookie.
- It would be covered in next tutorial, along with CSRF prevent issue.

# So what to do for Phase 4 should be clear

- This week, you can work on:
  - (IV) login page and user authentication.
  - (I) and (II) Input form checking to avoid XSS and SQL injection.

- Remaining parts, covered in later tutorials:
  - (IV) remember authentication using cookie
  - (III) CSRF prevent
  - (V) Session management (These three things are related)

  - (VI) Digital certificate.

# Mild Reminder

- I would try to release sample codes for your reference this or next week.

- Please note your instance URL in your REAME file so that TA can check your website effect at once. (Otherwise, I don’t have your database and I cannot run your web page on my desktop…)

# Interactive Q&A session for previous phases

- Thanks you!