Project Tutorial 4

Protecting your database and passwords

Dr. Yuzuko Nakamura
Dept. of Computer Science
Y.Nakamura@ucl.ac.uk

Validating data submitted through forms

Check something was submitted

- Useful PHP functions for checking something was submitted:
  - `isset($_POST['user_name'])` - check that a value exists.
  - `trim($_POST('user_name'))` - remove whitespace characters from start and end of string.
  - `empty($_POST('user_name'])` - check that a variable holds a value other than empty string, zero, null, false (and some others).

Check format conforms to what is expected

- `is_numeric($var)` - true if a number or string representing a number.
- `preg_match('/pattern/', $string)` - regular expression pattern matching.
  - https://www.php.net/manual/en/function.preg-match.php
- `filter_var` - many options for checking values.
  - See: https://www.php.net/manual/en/filter.filters.php

```
$email = filter_var($_POST['email'], FILTER_SANITIZE_EMAIL);
if (filter_var($email, FILTER_VALIDATE_EMAIL)) {
  echo "$email is a valid email address.";
}
else {
  echo "$email is NOT a valid email address.";
}
```

HTTP responses can be constructed

- HTTP requests can be constructed and sent to a server independently of using a browser.
- Form responses aren't the only way people can/will attempt to send requests to a server.
- Always validate received data on the server side even if you already validated on the client side, or your forms restrict the type of data you expect to receive.

Storing passwords

Storing user passwords in your database

- Best practice: don't store user passwords anywhere.
- On receipt, hash user's password, store the long hash in your database, throw away password.
- Hash functions are deterministic:
  - When user enters their password, it gets hashed to the same value and can be checked for equality.
  - Possible but extremely unlikely to get the same hash having entered a different password.

Other advice

- Good to salt passwords (add a per-user extra to the password) to make it harder to guess a user's password even if it hashes to the same value as another user.
- Use additional validation:
  - Check for common words and reject, e.g., "password".
  - Enforce minimum length.

Hashing passwords in MySQL

- MySQL defines various functions for encryption and generating hash codes.

```
INSERT INTO Users (username, password) VALUES
('someone', SHA('password'))
```

- To check a password, compare the SHA values:

```
SELECT username FROM Users WHERE username =
'$user' AND password = SHA('$password')
```

SHA / SHA1

- SHA = Secure Hash Algorithm
  - Returns a 40 character (160-bit) hexadecimal hash code.
  - Represent with VARCHAR(40).
  - More secure than MD5.
- If SSL is set up on your server, using SHA2 is better.
- Reference: https://dev.mysql.com/doc/refman/5.5/en/encryption-functions.html

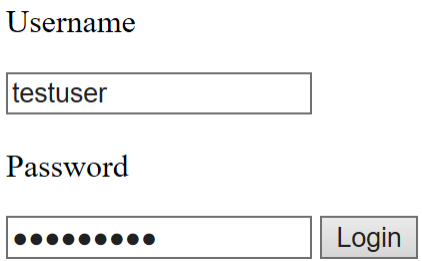
Hashing passwords in PHP

- Alternatively, can hash passwords in PHP before sending them to the database.
  `$passhash = password_hash($password, PASSWORD_DEFAULT);`
  - Automatically hashes password with a salt.
  - Store using VARCHAR(60) (longer because it includes algorithm and salt info)
- Verify using password_verify($password, $hash) function.
  - https://www.php.net/manual/en/function.password-hash.php

Use password input element

```
<input type="password" name="password">
```

- Displays an input field but characters typed appear as dots or asterisks:



Need HTTPS to keep passwords safe

- But note: form data (including passwords) are still sent in plaintext to the server.
- Can be compromised by someone sniffing internet traffic.
- Need to use HTTPS to send passwords across the internet securely.

Storing your database credentials

- Last time: put database credentials in mysqli_connect() args.
  - Not a good practice!
  - Definitely not good if you're checking that code into a public repository.
- Store credentials in a file outside www folder and outside version control and load them into PHP as variables.
- Extra precautions can be taken (encrypt that file, etc.).

Protecting your database

SQL injection attacks

- Example: data loss.
- Website asks for email using input field.
- Hacker inputs: `x'; DROP TABLE members; --';`
- Sent to database as SQL query:

```
SELECT email, passwd, login_id FROM members
WHERE email = 'x'; DROP TABLE members; --';
```

How it works

- x'; ends the previous statement even if it results in no data.
- DROP TABLE members; drops a table.
- -- is an SQL comment that causes the rest of the query set up by the programmer to be ignored

SQL injection attack example 2

- Example: steal data.
- Hacker inputs: `1234 or TRUE;`
- Sent to database as SQL query:
  ```
  SELECT * FROM Users WHERE user_id = 1234 OR TRUE;
  ```
- Hacker potentially gets access to entire table.

SQL injection attack example 3

- Example: reset all passwords.
- Register new user SQL query:
```
UPDATE Users SET password = SHA('$password')
WHERE account_id = $account_id
```
- Hacker inputs: (account ID) `1234 OR TRUE`
(password) `mypass'), admin=('1`
- SQL query sent to server:
```
UPDATE Users SET password = SHA('mypass'),
admin=('1') WHERE account_id = 1234 OR TRUE
```

Denial of service (DoS) injection

```
SELECT * FROM Catalog JOIN Catalog JOIN Catalog
JOIN Catalog ORDER BY id;
```

- Get server to perform a query generating billions of rows.
  - Severe slow down or crash.
  - SQL servers can be configured to cap load.

Injection is a real problem

- 2009 Data Breach Investigations Report Verizon Business RISK Team:
  - “When hackers are required to work to gain access, SQL injection appears to be the uncontested technique of choice.”
  - “In 2008, this type of attack ranked second in prevalence (utilized in 16 breaches) and first in the amount of records compromised (79 percent of the aggregate 285 million).”
- Imperva (July 2012). "Imperva Web Application Attack Report" (PDF).
  - “Retailers suffer 2x as many SQL injection attacks as other industries. / While most web applications receive 4 or more web attack campaigns per month, some websites are constantly under attack. / One observed website was under attack 176 out of 180 days, or 98% of the time.”
- See https://en.wikipedia.org/wiki/SQL_injection

How to mitigate SQL injections

- Escape quotation marks so that they are not interpreted as quotations in a SQL query.
- Enforce SIMPLE structures on inputted data.

```
UPDATE Users SET password = SHA('xyzzy\'),
admin=('1') WHERE account_id = 1234
```

If an integer is expected, make sure input is an integer.

No longer ends password string. (Will be rejected as an improperly formed query.)

Making a SQL query

```
$password = mysqli_real_escape_string($connection,
$_POST["password"]);
$id = (int) $_POST["account_id"];
```

  - Mysqli function optionally uses character set of database to generate escaped string.
- Use the PHP functions rather than attempt to write your own.

Does that solve the problem?

- No, but mitigation techniques get more complicated.
  - Stored procedures
  - Testing tools to look for vulnerabilities
  - Etc.

HTML / JavaScript injection

- User can enter HTML and/or Javascript into a form field as data (e.g. into user comments or reviews submission form).
- When the data is displayed to another user, the HTML/Javascript is echoed to the page.
  - Can inject links, ads, images, etc.
- Worse, Javascript code can be run and access a lot more information.
  - Known as Cross-Site Scripting.

Same strategy: check the input

```
$comments = htmlspecialchars($_POST['comments']);
```

- Replace HTML chars like < and > to their HTML version < and >

Use server logs to detect unusual activity

- Server records its activity in log file containing messages.
  - On WAMP see the Apache access and error logs, and the MySQL log.
- Log files should be reviewed and/or analysed regularly to see if anything unusual is happening.
  - Also a good place to look if a server is not working correctly or fails to start up.
- Server configuration determines level of detail and what gets logged.

Apache Access Log

```
127.0.0.1 - - [12/Jan/2011:00:53:02 +0000] "GET / HTTP/1.1" 200 4166
127.0.0.1 - - [12/Jan/2011:00:53:03 +0000] "GET /index.php?img=gifLogo
HTTP/1.1" 200 4549
127.0.0.1 - - [12/Jan/2011:00:53:03 +0000] "GET /index.php?img=pngWrench
HTTP/1.1" 200 741
127.0.0.1 - - [12/Jan/2011:00:53:03 +0000] "GET /index.php?img=pngPlugin
HTTP/1.1" 200 548
127.0.0.1 - - [12/Jan/2011:00:53:03 +0000] "GET
/index.php?img=pngFolderGo HTTP/1.1" 200 694
127.0.0.1 - - [12/Jan/2011:00:53:06 +0000] "GET /index.php?img=favicon
HTTP/1.1" 200 1429
127.0.0.1 - - [12/Jan/2011:01:21:13 +0000] "GET /phpmyadmin/ HTTP/1.1"
200 2718
127.0.0.1 - - [12/Jan/2011:01:21:15 +0000] "GET /phpmyadmin/js/common.js
HTTP/1.1" 200 13404
```

- Records IP address of source, time stamp, the HTTP message

Logs can be analysed

- Find out how your server is being accessed.
  - Which pages/services are used most.
  - How many distinct IP addresses requests come from (rough measure of how many visitors).
  - Look for rogue behaviour.
    - e.g., why are so many requests coming from one IP address.
    - Is it a hacker running a script against your site?
- Wide range of analysis tools available, e.g. AWStats

Summary

- Responsible use of databases in web applications requires thorough validation of data before it interacts with the database.
- Don’t keep passwords around when you can avoid it.

TODOs this week:

- Update your SQL database design or incorporate security precautions as you like (not mandatory).
- Begin implementing some of the features that require inserting or reading data from the database.