

Security

Multimedia Design 2
Week 13

Security

- Basic concepts
- Login example
- SQL injection
- Cross-site scripting

Auth*

- Authentication — *who are you?*
- Authorization — *are you allowed to do that?*

Intention + trust

- Did you **intend** to do that?
- Do you **trust** the code you're running?
Where did it come from?

Types of authentication

- Something you know (a password)
- Something you have (a key)
- Something you are (a fingerprint)

What can go wrong

- Lose your bank account holdings
- Lose your database contents
- Unauthorized access of content
- User tricked into performing a task
- User's credentials are compromised
- Etc.

Basic login example

- User submits a username and password
- Website compares credentials against those stored in a database
- A session is established
- Session is ended

Caveat

- Security is a sliding scale
- This example is optimized for **simplicity** not security

The login form

```
<form action="login.php" method="post">
  <label>
    Username
    <input type="text" name="username" />
  </label>
  <label>
    Password
    <input type="password" name="password" />
  </label>
  <input type="submit" value="Login" />
</form>
```

The login script

```
<?php
session_start();

if (check_credentials()) {
    $_SESSION['user'] = $_POST['username'];
}

header('Location: index.php');

?>
```

Verifying later

```
<?php  
  
session_start();  
  
if (!empty($_SESSION['user'])) {  
    echo "User is logged in.";  
} else {  
    echo "User is not logged in.";  
}  
  
?>
```

SQL injection

```
// User submitted a search request
$query = $_GET['q'];

// Check the posts for the search query
mysql_query("
    SELECT *
    FROM posts
    WHERE content IS LIKE '%$query%'
");

// What could go wrong?
```

The problem

- What happens if a user searches for:
`' ; delete from posts ;`
- The `$query` variable breaks out of a `SELECT` and deletes everything

```
SELECT * FROM posts WHERE  
content IS LIKE '%'; delete  
from posts;%'
```

Two solutions

- Escape user input with `mysql_real_escape_string`
- Use prepared SQL statements

Prepared SQL

- Allows you to safely pass data to a SQL command
- You don't have to worry about forgetting to escape an input
- PHP's PDO library is useful here

Using PDO

```
$pdo = new PDO($dsn);
```

```
$query = $pdo->prepare("
    SELECT *
    FROM posts
    WHERE content IS LIKE '%?%'
");
```

```
$query->execute(array($_GET['q']));
$posts = $query->fetchAll();
```

Cross-site scripting

- Commonly abbreviated XSS
- When code gets on the page that wasn't intended to get there
- Code gets executed by unwitting users and makes a mess of things

The problem

```
<?php  
echo "You searched for: {$_GET['q']}";  
?>
```

Insert evil code in the '...' part of the URL + share the link
<http://example.org/?q=<script>...</script>>

Some evil XSS code

```
// Transfer some cash  
document.write('');
```

```
// Steal the session cookie  
document.write('
```

The evil XSS code is now **displayed** instead of executed

XSS is hard to solve

- XSS is a hard problem, but it is solvable
- Requires planning + attention to detail
- Each context requires different kinds of escaping: **character data, element attributes, JavaScript, stylesheets, etc.**

Rules of thumb

- Handle untrustworthy data with care
- All data is untrustworthy

Resources

- [SQL injection cheat sheet](#)
- [XSS cheat sheet](#)
- [25 most dangerous programming errors](#)
- [OWASP Top 10 project](#)
- [HTML Purifier library](#)