SQL Injection and XSS

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

Outcomes

After today's lecture you will be able to:

- Describe, execute, and defend against a SQL Injection attack
- Describe, execute, and defend against a Cross Site Scripting attack

Inspiration

“Up to a point, it is better to just let the snags [bugs] be there than to spend such time in design that there are none.” – Alan M. Turing

SQL Injection

SQL Language

- Widely used database query language
- Fetch a set of records

```
SELECT * FROM Accounts WHERE Username='Alice'
```

- Add data to the table

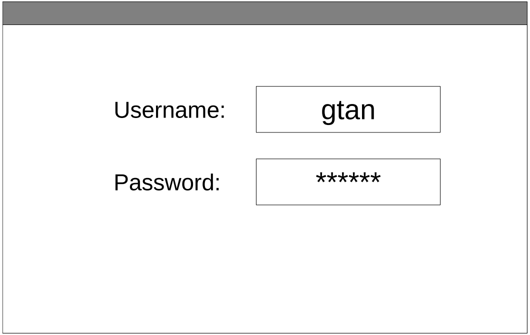
```
INSERT INTO Accounts(Username, Password) VALUES ('Alice', 'helloworld')
```

- Modify data

```
UPDATE Accounts SET Password='hello' WHERE Username='Alice'
```

- Query syntax (mostly) independent of vendor

Example Web App



Username: gtan

Password: ******

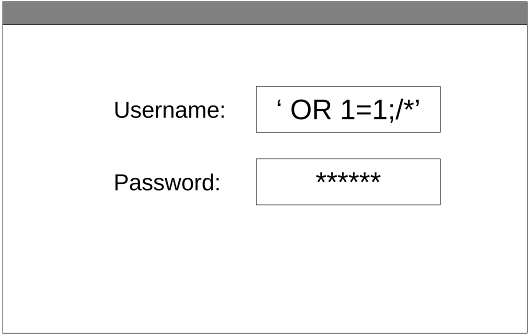
Constructing SQL Query from User Input

```
$result = mysql_query(
  "SELECT * FROM Accounts".
  "WHERE Username = '$username'".
  "AND Password = '$password';");
if (mysql_num_rows($result) > 0)
  $login = true;
```

Resulting SQL Query

```
SELECT * FROM Accounts
WHERE Username = 'gtan'
AND Password = 'geheim';
```

SQL Injection Example



Username: ‘ OR 1=1;/*’

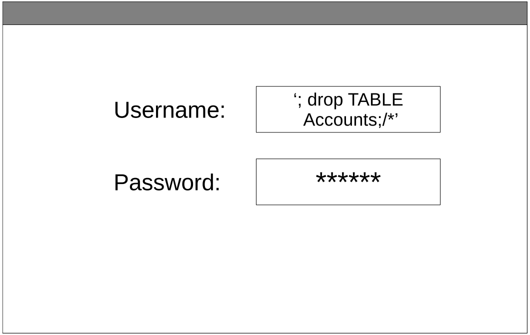
Password: ******

Resulting SQL Query

```
SELECT * FROM Accounts
WHERE Username = '' OR 1=1;/*'
AND Password = 'geheim';
```

OOPS!

SQL Injection Example



Username: ‘; drop TABLE Accounts;/*’

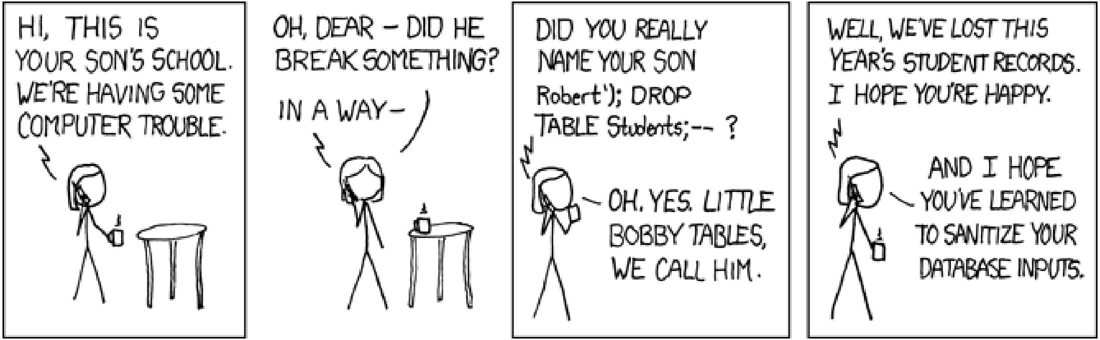
Password: ******

Resulting SQL Query

```
SELECT * FROM Accounts
WHERE Username = '';
drop TABLES Accounts;
/*'AND Password = 'geheim';
```

OOPS!

Exploits of a Mom


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY –
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

http://xkcd.com/327/

SQL Injection

- Vulnerability: any application in any programming language that connects to a SQL database
- Typical books such as “PHP & MySQL for Dummies” contain examples with security vulnerabilities!
- Note the common theme to many injection attacks: concatenating strings, some of them user input, and then interpreting the result

Examples of Real SQL Injection Attacks

- Oklahoma Department of Corrections divulges thousands of social security numbers (2008)
  - Sexual and Violent Offender Registry for Oklahoma
  - Data repository lists both offenders and employees

CardSystems Attack (June 2005)

- CardSystems was a major credit card processing company
- Put out of business by a SQL injection attack
  - Credit card numbers stored unencrypted
  - Data on 263,000 accounts stolen
  - 43 million identities exposed

Preventing SQL Injection

- Input validation
  - Filtering input: apostrophes, semicolons, percent symbols, hyphens, underscores, …
    - Any character that has special meanings
  - Check the data type (e.g., make sure it's an integer)
- **Whitelisting** what's allowed
  - Allow only a well-defined set of safe values
  - Better than **blacklisting** "bad" characters
    - May forget to filter out some characters

“Blacklists” are useful for testing

- Identify some data you should not accept
  - But don’t use this blacklist as your rules
- Instead, use blacklists to test your whitelist rules
  - I.e., use (subset of) a blacklist as test cases
  - to ensure your whitelist rules won’t accept them
- In general, regression test should check that “forbidden actions” are actually forbidden
  - E.g., Apple iOS’s “goto fail” vulnerability (CVE-2014-1266)
    - Its SSL/TLS implementation accepted valid certificates (good) and invalid certificates (bad).
    - No one tested it with invalid certificates!

Escaping Quotes

- For valid string inputs use escape characters to prevent the quote becoming part of the query
  - Example: escape(o’brien) = o”brien
    - E.g., ANSI SQL mode in MySQL
  - Another example: Convert ’ into ’
    - E.g., MySQL mode in MySQL
  - Different databases have different rules for escaping
  - Only works for string inputs

Prepared Statements

- Metacharacters such as ’ in queries provide distinction between data and control
- In most injection attacks data are interpreted as control – this changes the semantics of a query or a command
- Bind variables; ? placeholders guaranteed to be data (not control)
- Prepared statements allow creation of static queries with bind variables. This preserves the structure of intended query.

Prepared Statement

Vulnerable:

```
String updateString = "SELECT * FROM Account WHERE Username" +
username + " AND Password = " + password;
stmt.executeUpdate(updateString);
```

Not Vulnerable:

```
PreparedStatement login = con.preparedStatement("SELECT *
      FROM Account WHERE Username = ? AND Password = ?");
login.setString(1, username);
login.setString(2, password);
login.executeUpdate();
```

Mitigating Impact of Attack

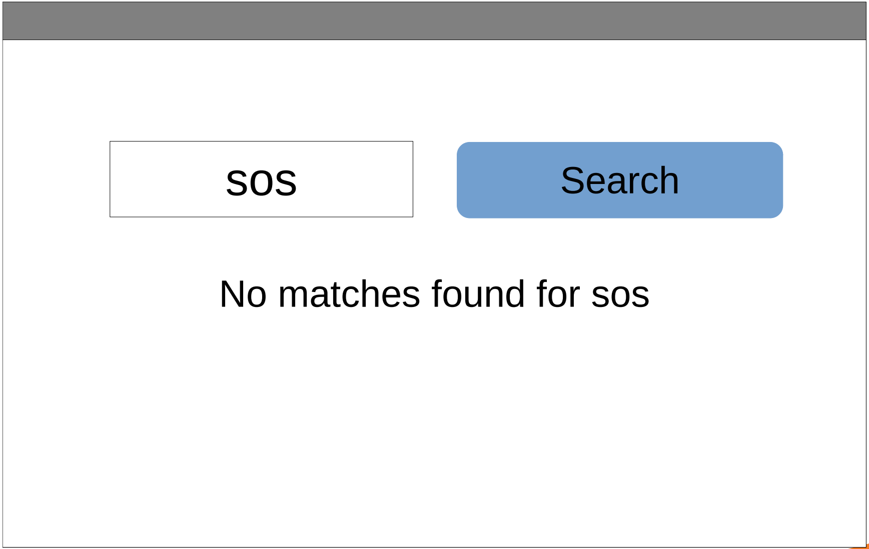
- Encrypt sensitive data stored in database
- Limit privileges (defense in depth)
- Harden DB server and host OS

Input Validation: XSS (Cross-Site Scripting)

Web Application (In)Security

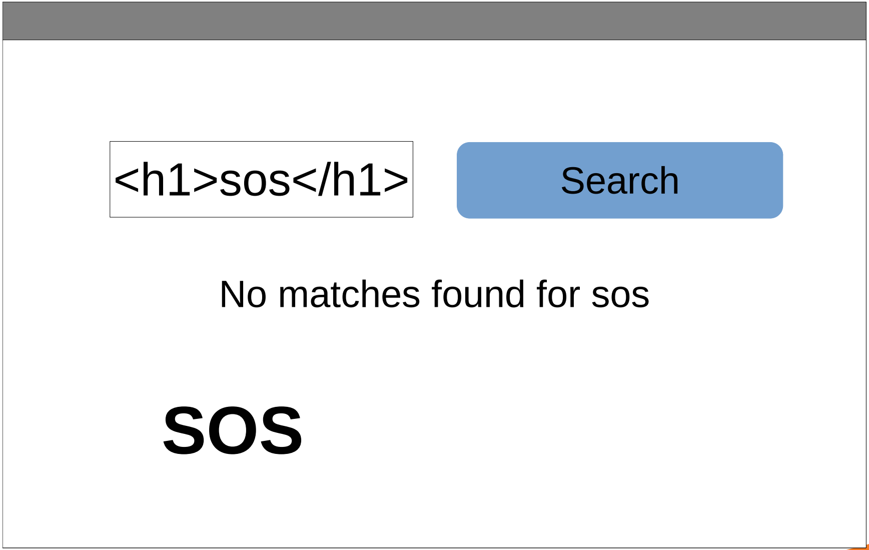
- Increasingly, web applications become obvious targets to attack
- Modern-day web browser
  - More like an OS
  - Allow downloading and installing web-applications, which take untrusted input

XSS (Cross site scripting)



No matches found for sos

XSS (Cross site scripting)



<h1>sos</h1>

No matches found for sos

SOS

XSS (Cross site scripting)

- What can happen if we enter more complicated HTML code as search term?

```
<img = "http://www.sxpam.org/advert.jpg">

<script langauge="javascript">alert('aloha');</script>
```

XSS (Cross site scripting)

- aka HTML injection
- Vulnerability: User input, possible including executable content (JavaScript, VBScript, ActiveX, ...) is echoed back in a webpage
- But why is this a security problem?

XSS – Scenario

❶ User A injects HTML into a website,
(e.g. webforum, book review on `amazon.com`, ...),
Which is echoed back to User B later

❷ this allows website defacement, or tricking User B to follow link to `anotheronlinebookshop.com`

❸ worse still, B's web browser will execute any javascript included in the injected HTML...

❹ This is done in the context of the vulnerable site, i.e., using B's cookies for this site...

https://www.youtube.com/watch?v=cbmBDiR6WaY

Why XSS is a Security Problem?

- The problem is that what attackers inject might be viewed by a victim in the victim's browser
  - The injected code will be run on the vitim's computer
  - With the origin from a trusted web site
- XSS injects malicious scripts into trusted web sites such as a banking web site
- Affects web sites, built using any language or technology, that echoes back user input in a webpage

XSS

- Countermeasures
  - Input validation
    - Blocking ”
      ” is not enough
    - Pseudo-urls, stylesheets, encoded inputs (%3C codes “<”), etc.
    - Hard to do in practice (see Samy Worm)
  - Principle of least privilege
    - Turn off scripting languages, restrict access to cookies, don’t store sensitive data in cookies, …

MySpace Worm

- Used script injection
- Started on “samy” MySpace page
- Everybody who visits an infected page, becomes infected and adds “samy” as a friend and hero
- 5 hours later “samy” has 1,005,831 friends
  - Was adding 1,000 friens per second at its peak

More Input Validation Problems

- From servers’ point of view, any data form the client cannot be trusted
- Data in web forms, incl. hidden form fields.
  Hidden form fields, e.g.

  ```
  <INPUT TYPE=HIDDEN NAME="price" VALUE="50">
  ```

  are not shown in browser, unless you click View -> Page source ..., and may be altered
- Data in cookies
  - cookies, stored at client-side, can be altered
  - Such data always has to be re-validated

Are there any questions?