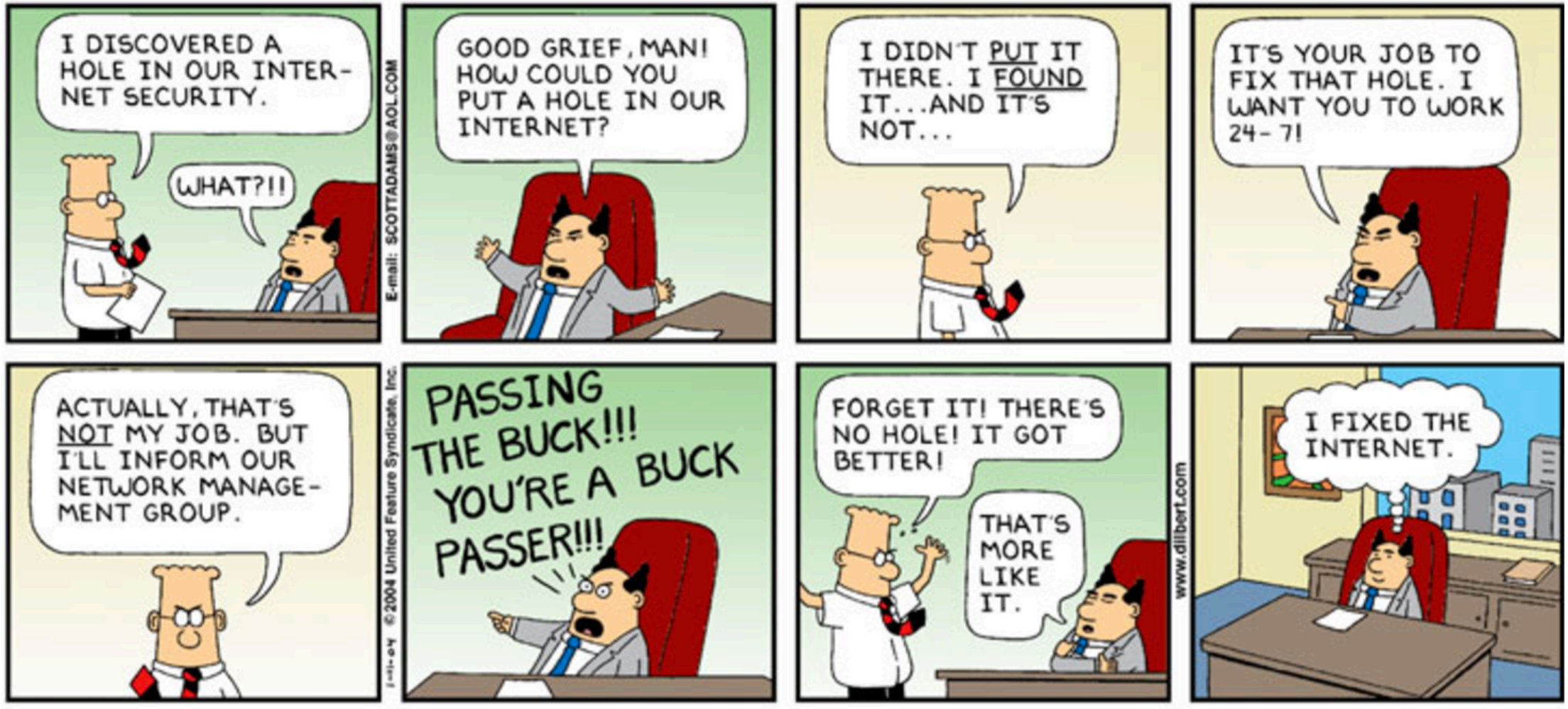
CSC 337



Web Security Vulnerabilities

Rick Mercer, Marty Stepp Justen Stepka, OWASP, Adam Doupe

Why Worry about Web Security?

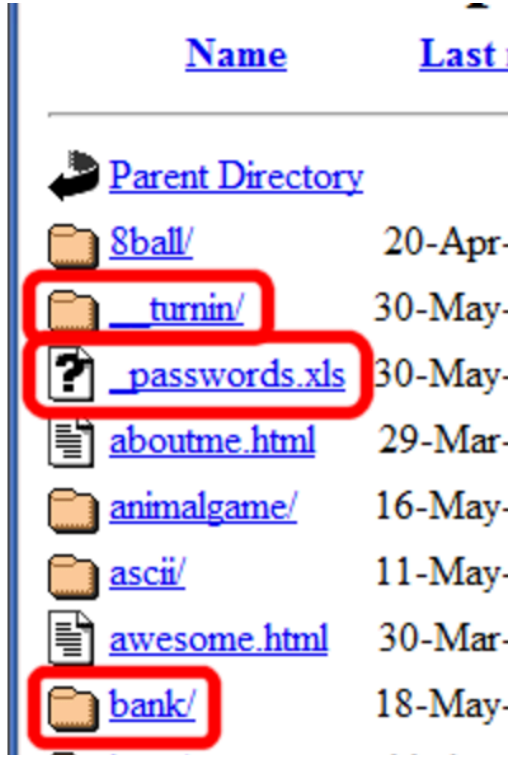
- We assumed valid input & kind users while using localhost
- When a website is available to anyone, we get invalid input and evil users to
  - **Read private data** (passwords, credit card #, grades, prices)
  - **Change data** (change grades, prices of products, passwords)
  - **Spoof** (pretend to be someone they are not)
  - **Damage or shut down the site**, so that it cannot be successfully used by others
  - **Harm the reputation or credibility** of the organization
  - **Spread viruses** and other malware

A few attacks

- **Information Leakage**: Allowing an attacker to look at data, files, etc. that he/she should not be allowed to see
- **Invalid input**: No client side or server side validation
- **Cross-Site Scripting (XSS)** or HTML Injection: Inserting malicious HTML or JavaScript content into a web page
- **SQL Injection**: Inserting malicious SQL query code to reveal or modify sensitive data
- **Man in the Middle**: Someone reads internet traffic
- **Session Hijacking**: Stealing another user's session cookie to masquerade as that user

Information Leakage

- The attacker can look at data, files, etc. that should not be seen
- Files on web server that should not be there
  - or read/write permissions are too generous
- Directories that list their contents (indexing)
  - can be disabled on web server
- Guess the names of files, directories, …
  - see `loginfail.php`, try `loginsuccess.php`
  - see `user.php?id=123`, try `user.php?id=456`
  - see `/data/public`, try `/data/private`



Form Validation

- **Form validation**: ensuring the HTML form's values are correct
- Some types of validation:
  - preventing blank values (email address)
  - ensuring the type of values
    - integer, real number, currency, phone number, Social Security number, postal address, email address, date, credit card number, ...
  - ensuring the format and range of values (ZIP code must be a 5-digit integer)
  - ensuring that values fit together (user types email twice, and the two must match)

Validating Input

- Validation can be performed:
  - Client-side before an HTML form is submitted
    - can lead to a better user experience, but not secure
  - Server-side in PHP code, after the form is submitted
    - needed for truly secure validation
  - On both the client and the Server
    - best mix of convenience and security but requires most effort to program

Client Side Validation

- Set the min and max of a value

```
<form action="demo_form.asp">  Enter a date before 1980-01-01:
<input type="date" name="bday" max="1979-12-31"><br>
```

- Use `required`
- Use regular expressions to force things like correct phone number formatting or zip code

Validating Input on the Server

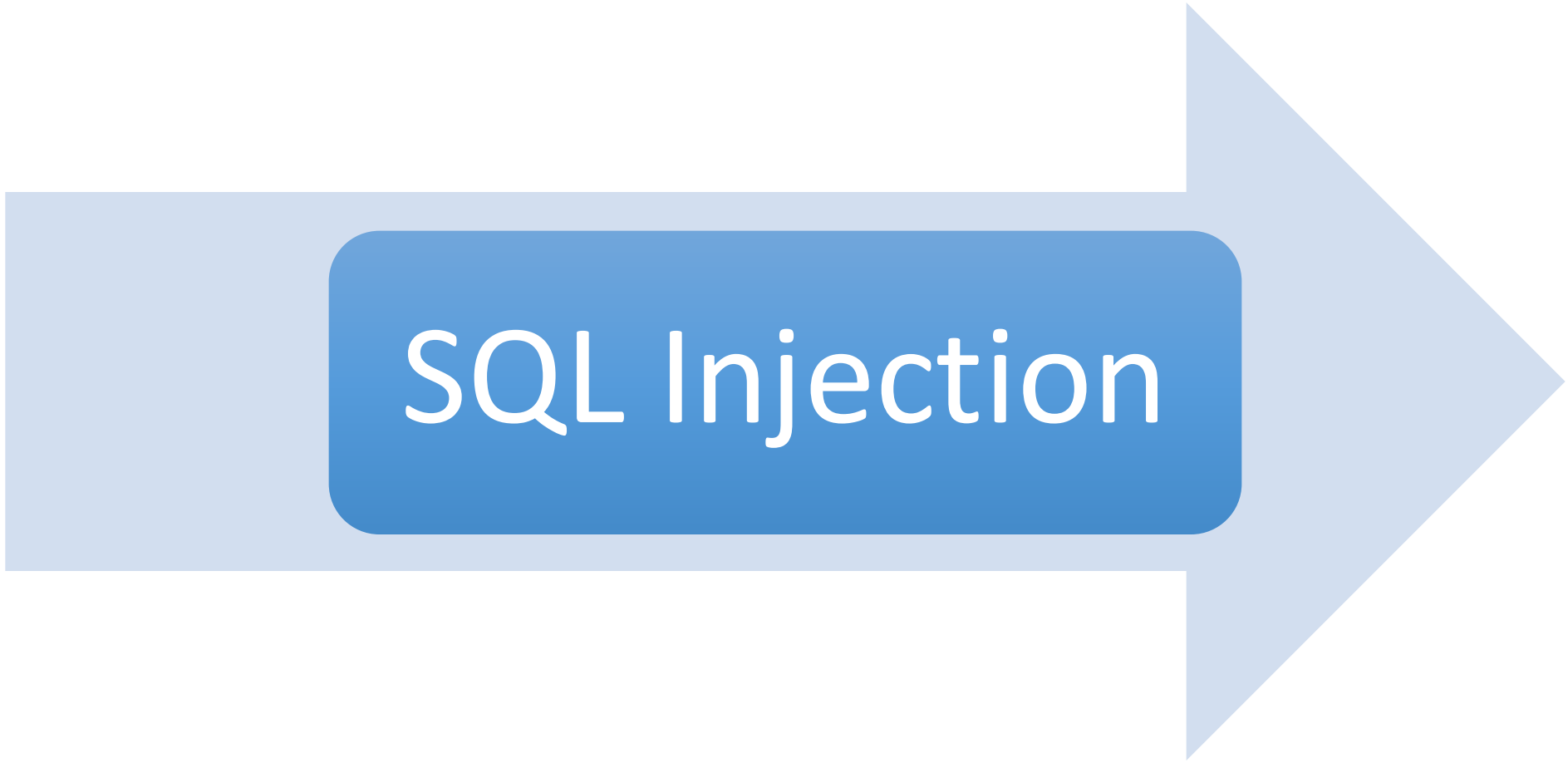
```
$city = $_POST["city"];
$state = $_POST["state"];
$zip = $_POST["zip"];
if (!$city || strlen($state) != 2 || strlen($zip) != 5) {
  Get a message in the form that was just submitted
}
```

- Examine parameter values, and if they are bad, show an error message back in the form
  - One did this with invalid authentication

Credit Cards

- Check patterns: different for visa, amex, discover
- Use checksum to validate correct number
  - Lune algorithm
  - PHP implementation on the right

```php
/* Luhn algorithm number checker - (c) 2005-2008 shaman - www.planzero.org
 * This code has been released into the public domain, however please
 *  give credit to the original author where possible. */
function luhn_check($number) {
  // Strip any non-digits (useful for credit card numbers with spaces and hyphens)
  $number=preg_replace('/\D/', '', $number);
 // Set the string length and parity
 $number_length=strlen($number);
 $parity=$number_length % 2;
 // Loop through each digit and do the maths
 $total=0;
 for ($i=0; $i<$number_length; $i++) {
     $digit=$number[$i];
     // Multiply alternate digits by two
     if ($i % 2 == $parity) {
         $digit*=2;
     // If the sum is two digits, add them together (in effect)
        if ($digit > 9) {
            $digit-=9; }
      }
      // Total up the digits
      $total+=$digit;
 }
 // If the total mod 10 equals 0, the number is valid
  return ($total % 10 == 0) ? TRUE : FALSE;
}
```


SQL Injection

SQL Injection

By Adam Doupé, candidate for security position at UofA

- Allows attacker to alter semantics of SQL query
- Consequences
  - Steal database
  - Alter database
  - Bypass login

SQL Injection – Example 1 ok

```
"select * from 'users' where 'id' ='" + $id + "';"

$id = "10"    // User entered 10

select * from 'users' where 'id' = '10';
```

SQL Injection – Example 1 bad

```
"select * from 'users' where 'id' ='" + $id + "';"

$id = "-1' or 1=1"  // User entered -1' or 1=

select * from 'users' where 'id' = '-1' or 1=1';
```

SQL Injection – Example 2 ok

```
"select * from 'users' where 'id' ='" + $id + "';"

$id = "10"  // User entered 10

select * from 'users' where 'id' = '10';
```

SQL Injection – Example 2 bad

```
"select * from 'users' where 'id' ='" + $id + "';"
```

```
$id = "-1'; drop table 'users';#"
```

```
select * from 'users' where 'id' = '-1'; drop table
'users';#';
```

SQL Injection – Example 3 bad

```
"select * from 'users' where 'id' ='" + $id + "';"
```

```
$id = "-1'; insert into 'admin' ('username', 'password')
values ('adamd', 'pwned');#"
```

```
select * from 'users' where 'id' = '-1'; insert into
'admin' ('username', 'password') values ('adamd',
'pwned');#';
```

SQL Injection – Prevention

- Use prepared statements with bindParam
  - Specify structure of query then provide arguments
- Prepared statements – *example from Josh not using password_verify*

```
function verifyCredentials($name, $pass) {
  $stmt = $db->prepare("select * from 'users' where 'username'
        = :name and 'password' = SHA1( CONCAT(:pass, 'salt'))");
  $stmt->bindParam(':name', $name);  // $name is a parameter
  $stmt->bindParam(':pass', $pass);  // $pass is a parameter
  // . . .
```

Sanitize Input

- Three of the top five vulnerabilities have one thing in common
  - A lack of input sanitization
- All three exploits are leveraged by data sent to the Web server by the end user
- Input comes in the form of GET and POST requests
- Use htmlspecialchars so it destroys SQL queries and HTML code

```
<?php
$str = '<a href="test">Test</a>';
echo htmlspecialchars($str);
?>
```

Output:

Test

Cross Site Scripting XSS

Cross Site Scripting (XSS)

- Occurs when an attacker uses a web application to send malicious code, generally in the form of a script, to a different end user
- This is a wide spread problem
- **Stored** attacks are injected code permanently stored on the target web server (database, message forum, visitor log, comment field, etc)
- **Reflected** attacks are delivered via another route such as an e-mail message. A user is then tricked into clicking on the malicious link or submitting data. The browser then executes the code from what is considered a ‘trusted’ server

XSS

- A user injects and executes arbitrary JavaScript code in your page
- JavaScript is often able to be injected because of a previous HTML injection

  `indexl.php?quote=`**`<script type='text/javascript'>alert('pwned');</script>`**

- Injected script code can
  - masquerade as the original page and trick user into entering sensitive data
  - steal a user's cookies
  - masquerade as the user and submit data on their behalf
    - submit forms, click buttons, etc.

Environments Affected

- All web servers, application servers, and web application environments are susceptible to cross site scripting.
- XSS Flaws can be difficult to identify and remove from a web application
- The best way to find flaws is to perform a security review of the code and search for all places where input from an HTTP request could possibly make its way into the HTML output

OWASP

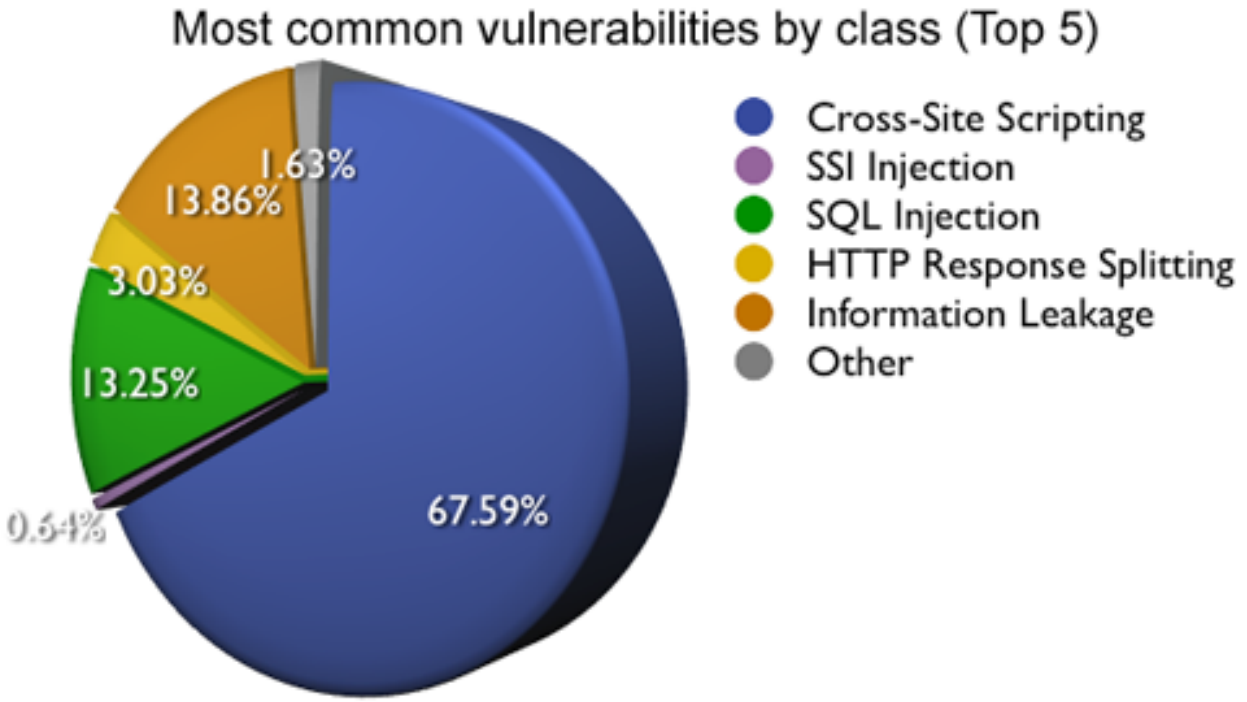
- Open Web Application Security Project (OWASP)
  - Started in 2000
  - http://www.owasp.org
  - Advance knowledge of web application and web security issues.
  - Generic testing tools
  - Knowledge base center

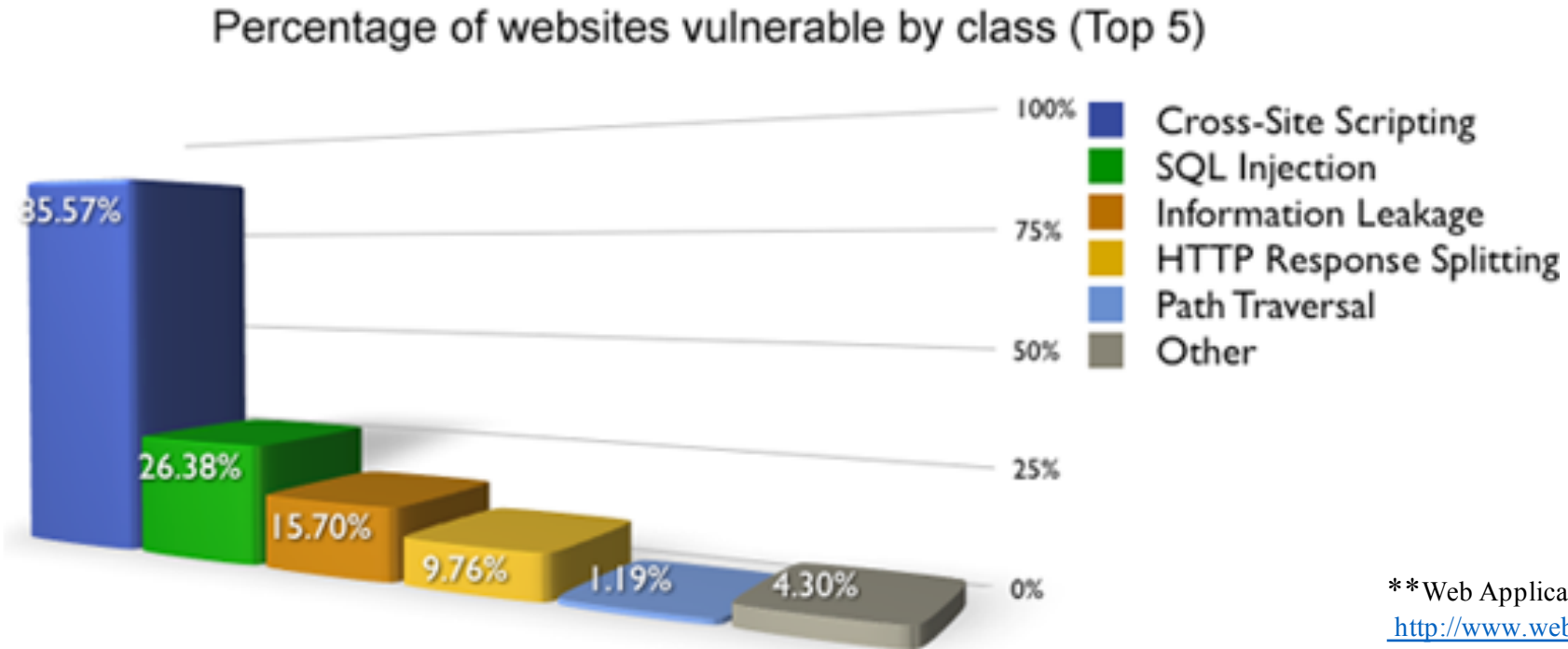
Background

- Covers top classes, types, or categories of web application vulnerabilities.
- No reliable source of statistics about web application security problems.
- The top ten list is in no particular order
- Ongoing effort, targeted towards largest audience possible

How Bad Is It?

- **Bad**





(Server-side Include)

**Web Application Security Consortium (WASC)
http://www.webappsec.org/projects/statistics/

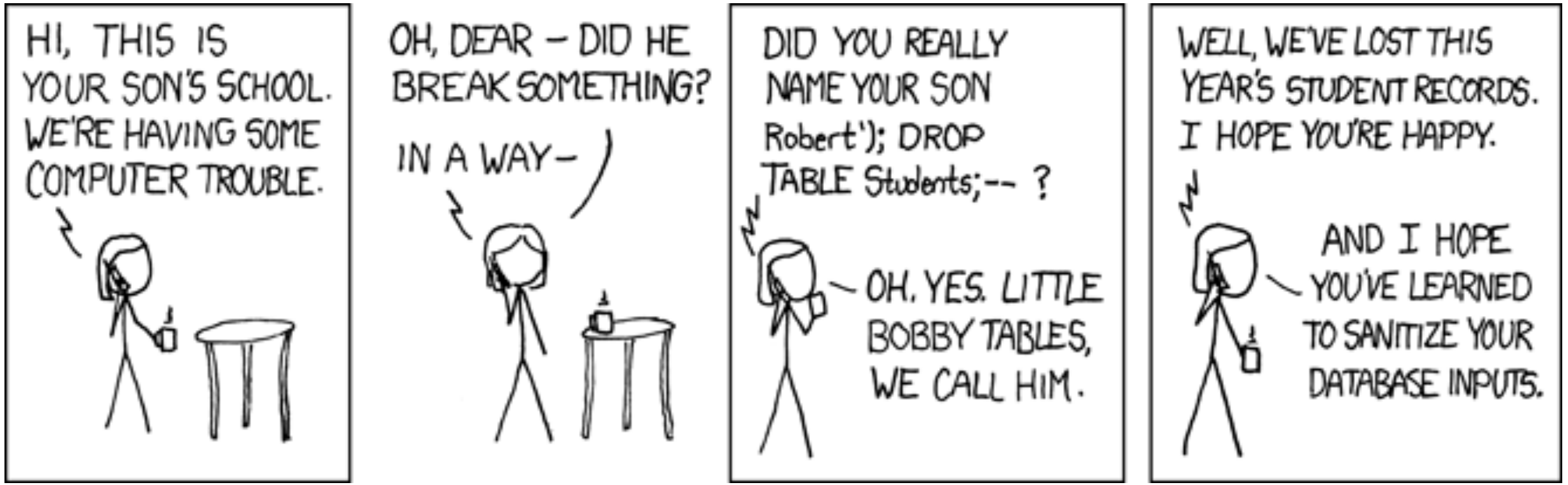
How Bad Is It?

- Pretty Bad
  - 31,373 Sites Tested

Threat Classification	No. of Vulns	Vuln. %	No. of Sites	% of Vuln. Sites
Brute Force	66	0.04%	66	0.21%
Content Spoofing	663	0.45%	218	0.69%
Cross Site Scripting	100,059	67.59%	26,531	84.57%
Directory Indexing	292	0.20%	168	0.54%
HTTP Response Splitting	4,487	3.03%	3,062	9.76%
Information Leakage	20,518	13.86%	4,924	15.70%
Insufficient Authentication	84	0.06%	1	0.00%
Insufficient Authorization	23	0.02%	4	0.01%
Insufficient Session Expiration	46	0.03%	1	0.00%
OS Commanding	143	0.10%	44	0.14%
Path Traversal	426	0.29%	374	1.19%
Predictable Resource Location	651	0.44%	173	0.55%
SQL Injection	19,607	13.25%	8,277	26.38%
SSI Injection	950	0.64%	298	0.95%
XPath Injection	14	0.01%	6	0.02%
	148,029	100.00%	44,147	

**Web Application Security Consortium (WASC) http://www.webappsec.org/projects/statistics/

CSC 337



Security Issues Needed on the Final Project

Rick Mercer

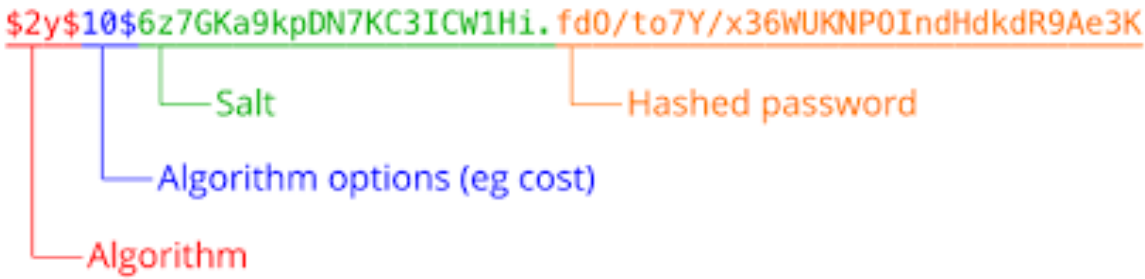
Outline

- Three Security Issues and Protections Needed in the final project
  - Store salted hashed passwords
  - Protecting against SQL Injection
  - Protecting against Cross Site Scripting with htmlspecialchars

Passwords should not be plain text

- Your table should also already have a password column
  - Could be `'password' varchar(255)`
  - Must have 60 characters
- Your password is likely stored in plain text
  - Is this a good idea?
- Do not store the password in plain text format
- Hash *and* salt the password
  - Read http://blog.codinghorror.com/youre-probably-storing-passwords-incorrectly/

1) Use PHP password_hash



- PHP's password_hash hashes and salts data such as passwords

```php
<?php
$pwd = '1234';      // <- What the user typed as plain text
$hashed_pwd = password_hash($pwd, PASSWORD_DEFAULT);
echo $hashed_pwd;  // Store this value below, not '1234'
?>
```

- Store the hashed values that lok like this

```
$2y$10$xDyg2vxlMnGaUNPGcbHQIeljUs4Jqdofoz3F4P1/lElF.vR5EV5.K
```

Then you need to confirm passwords

- `password_verify` is the other useful needed PHP function

```
boolean password_verify(string $pwd, string $hash)
```

2) Avoid SQL Injection with `bindParam`

- Remember Injected SQL can
  - change the query to output others' data (revealing private information)
  - insert a query to modify existing data (increase bank account balance)
  - delete existing data (; DROP TABLE students; --)
  - bloat the query to slow down the server (JOIN a JOIN b JOIN c ...)

How bad can it get?

- The programmer expects a number, but gets a string

```
"SELECT * FROM users WHERE id=" . $ID
```

- User enters

ID: 1; DROP TABLE users

- Prepared statement becomes two statements

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

Preventing SQL Injection

- Use prepared statements and bind parameters (new)--more secure

```
public function getGradesFor($studentName) {
  $stmt = $this->DB->prepare (
        "Select * from grades where student_id = :studentName" );
  $stmt->bindParam( 'studentName', $studentName );
  $stmt->execute ();
  return $stmt->fetchAll ( PDO::FETCH_ASSOC );
}
```

- This is more secure because PDO assembles dynamic SQL and prepares it after adding user data **`$studentName`** as a string
  - Not as part of the the where clause

3) use htmlspecialchars on all user input

- Validate all headers, cookies, query strings, from fields, and hidden fields with htmlspecialchar($user_input). It changes

  < to <

  > to ≷

  (to (

) to)

  # to #

  & to &

use htmlspecialchars($string)

- Escape them with php's `htmlspecialchar` function
  - Returns an HTML-escaped version of the string

```
$text = "<script>hi 2 u & me</script>";
$text = htmlspecialchars($text);
echo $text
```

- Output

```
&lt;script&gt;hi 2 u &amp; me&lt;/script&gt;
```

In Summary, everyone's final project must

- Use salted hashed passwords, use PHP's
  - password_hash($pwd, PASSWORD_DEFAULT);
  - boolean password_verify($pwd, $hash)
- Use prepared statements with bindParam

```
public function getGradesFor($studentName) {
  $stmt = $this->DB->prepare (
        "Select * from grades where student_id = :studentName" );
  $stmt->bindParam ( 'studentName', $studentName );
```

- Use PHP's htmlspecialchars on all input

```
$text = htmlspecialchars($text);
```