CSE 484 / CSE M 584: Computer Security and Privacy

Web security: Lab 2 and Context

Fall 2017

Jared Moore
jlcmoore@cs.uw.edu

Thanks to Franzi Roesner, Dan Boneh, Dieter Gollmann, Dan Halperin, Yoshi Kohno, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...

Looking Forward

- Today: Introduction to Lab 2 + related concepts
- Wednesday & Monday: More web security
  - **No class or office hours on Friday!**

- Lab #2 out; due **11/20**
- Final Project Deadline #1 due **Friday**

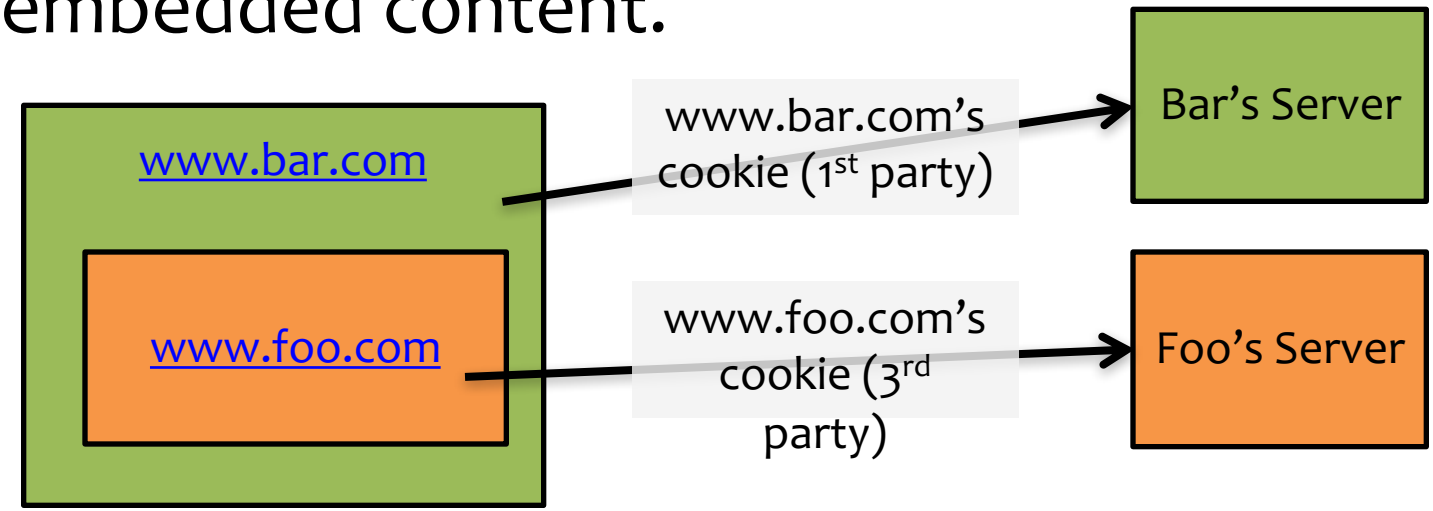
- Section this week: More lab 2 and clickjacking

Same-Origin Policy (Cookies)

- **For cookies:** Only code from same origin can read/write cookies associated with an origin.
  - Can be set via Javascript (`document.cookie=…`) or via `Set-Cookie` header in HTTP response.
  - Can narrow to subdomain/path (e.g., http://example.com can set cookie scoped to http://account.example.com/login.)
  - Secure cookie: send only via HTTPS.
  - HttpOnly cookie: can't access using JavaScript.

Same-Origin Policy (Cookies)

- Browsers automatically include cookies with HTTP requests.
- **First-party cookie:** belongs to top-level domain.
- **Third-party cookie:** belongs to domain of embedded content.



XSS: Cross-Site Scripting

- **Idea:** Place user-provided data in the page.
  - Makes page more interactive and personal.
- **Threat:** Improperly used data can be interpreted as code.
- **Solutions?**
  - Sanitize/validate input. (e.g., `htmlspecialchars()`)
  - Browser detection/prevention.

Server Side Scripts Review

- Before a webpage is sent to you, code is executed by the server
- Can be use to set and read cookies for authentication
- You will need a basic script to receive captured cookies
- We will use PHP

Lab 2

Overview

- Pikachu, Meowth, and Cookies
  - XSS; **Today**
- Jailbreak
  - SQL Injection; **Today** if time
- Hack your 4.0!
  - XSRF; **Wednesday** or **Monday**

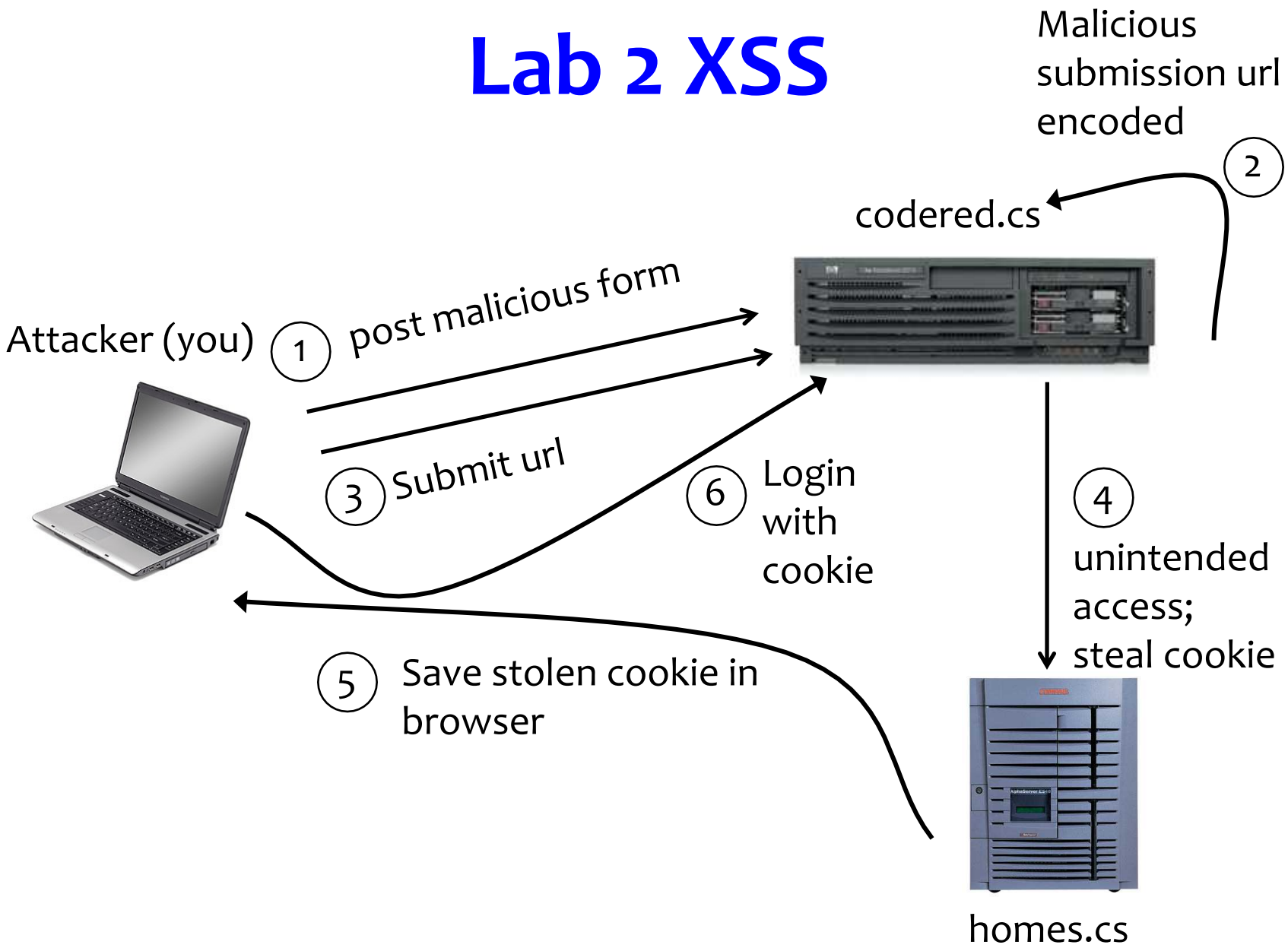
Lab 2 XSS

- Give the TAs (codered.cs) a link with a XSS vulnerability.
- TAs will ‘visit’ this link, and their cookie will be stolen.
- The process of stealing cookie involves sending it to a place you control.
- You’ll save the cookie, read it, and use it to log in

Tools

- Web browser (Firefox or Chrome)
- Cookie editing capability
- A php script on homes.cs to capture cookies
  - (see lab details)

Lab 2 XSS


Malicious submission url encoded
2
codered.cs
Attacker (you)
1
post malicious form
3
Submit url
6
Login with cookie
4
unintended access; steal cookie
5
Save stolen cookie in browser
homes.cs

Demo

Preventing Cross-Site Scripting

- Any user input and client-side data <u>must</u> be preprocessed before it is used inside HTML
- Remove / encode HTML special characters
  - Use a good escaping library
    - OWASP ESAPI (Enterprise Security API)
    - Microsoft's AntiXSS
  - In PHP, htmlspecialchars(string) will replace all special characters with their HTML codes
    - ' becomes ' " becomes " & becomes &
  - In ASP.NET, Server.HtmlEncode(string)

Evading XSS Filters

- Preventing injection of scripts into HTML is hard!
  - Blocking “<” and “>” is not enough
  - Event handlers, stylesheets, encoded inputs (%3C), etc.
  - phpBB allowed simple HTML tags like <b>

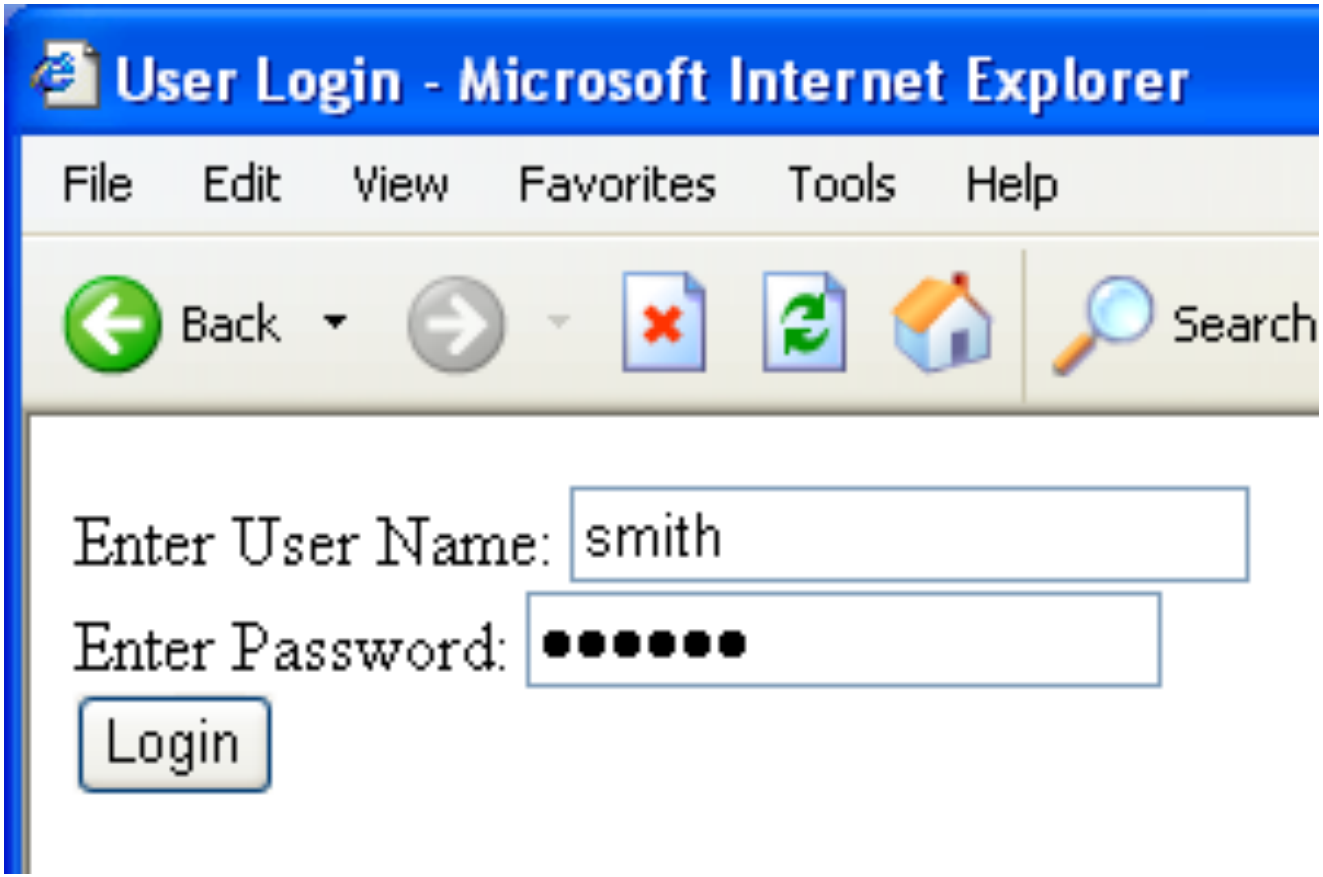
```
<b c=“>” onmouseover=“script” x=“<b ”>Hello<b>
```

- Beware of filter evasion tricks (XSS Cheat Sheet)
  - If filter allows quoting (of <script>, etc.), beware of malformed quoting: `<IMG """><SCRIPT>alert("XSS")</SCRIPT>">`
  - Long UTF-8 encoding
  - Scripts are not only in <script>:

```
<iframe src=‘https://bank.com/login’ onload=‘steal()’>
```

SQL Injection

Typical Login Prompt

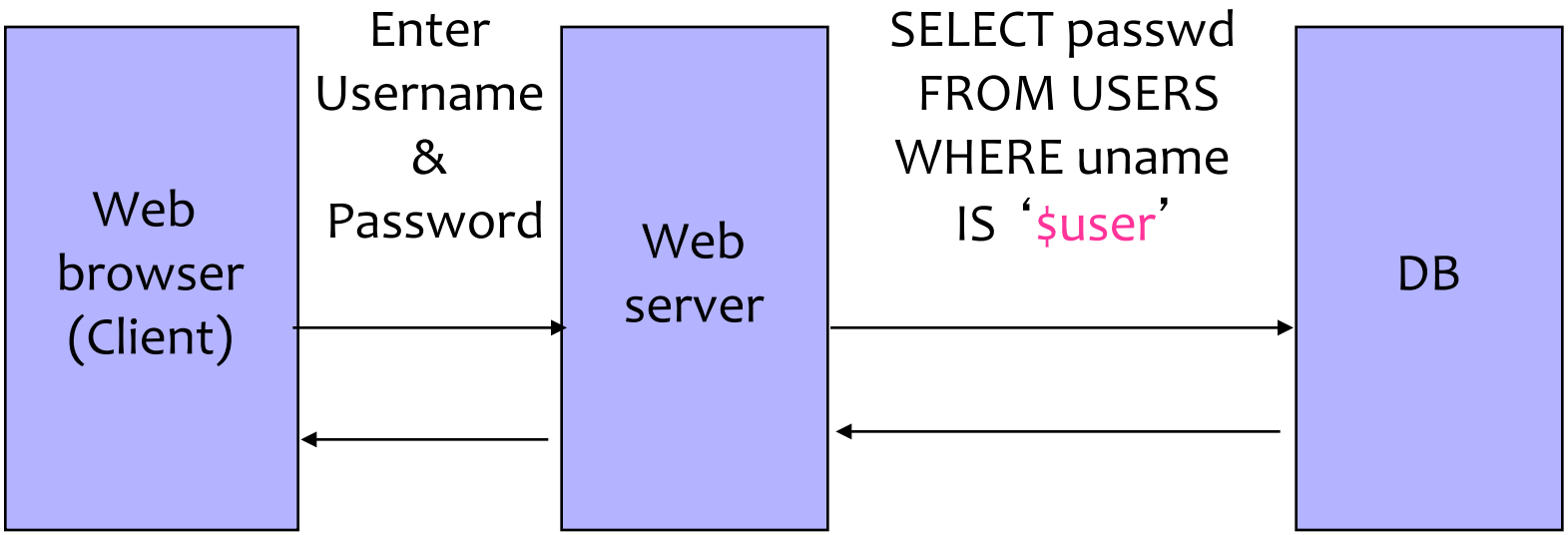


Typical Query Generation Code

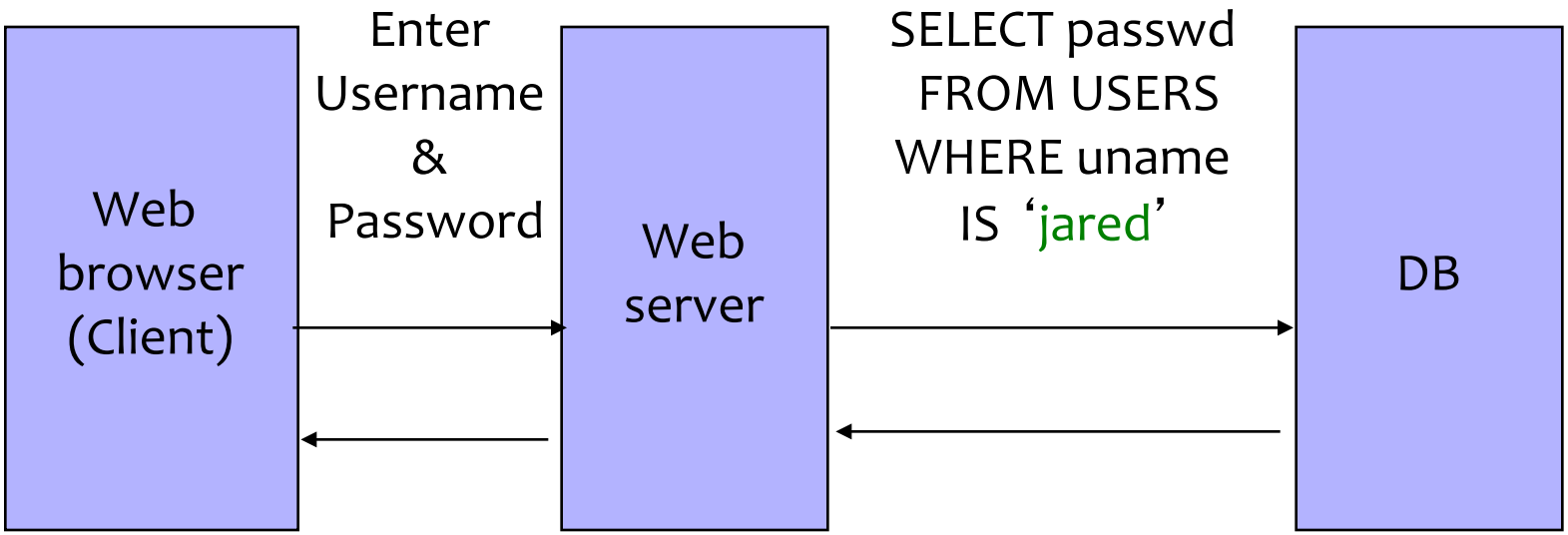
```
$selecteduser = $_GET['user'];
$sql = "SELECT Username, Key FROM Key " .
        "WHERE Username='$selecteduser'";
$rs = $db->executeQuery($sql);
```

What if **'user'** is a malicious string that changes the meaning of the query?

User Input Becomes Part of Query

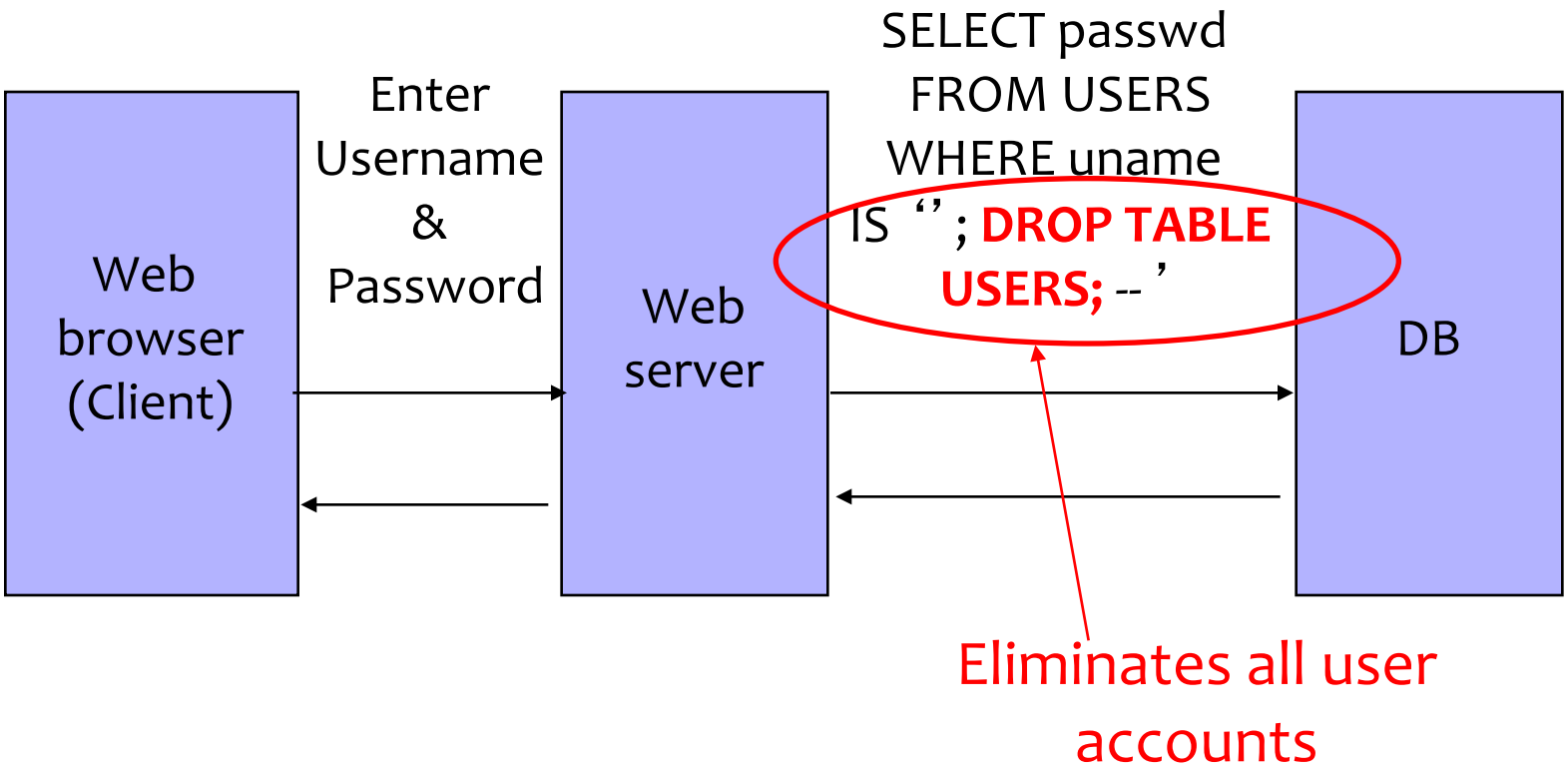


Normal Login

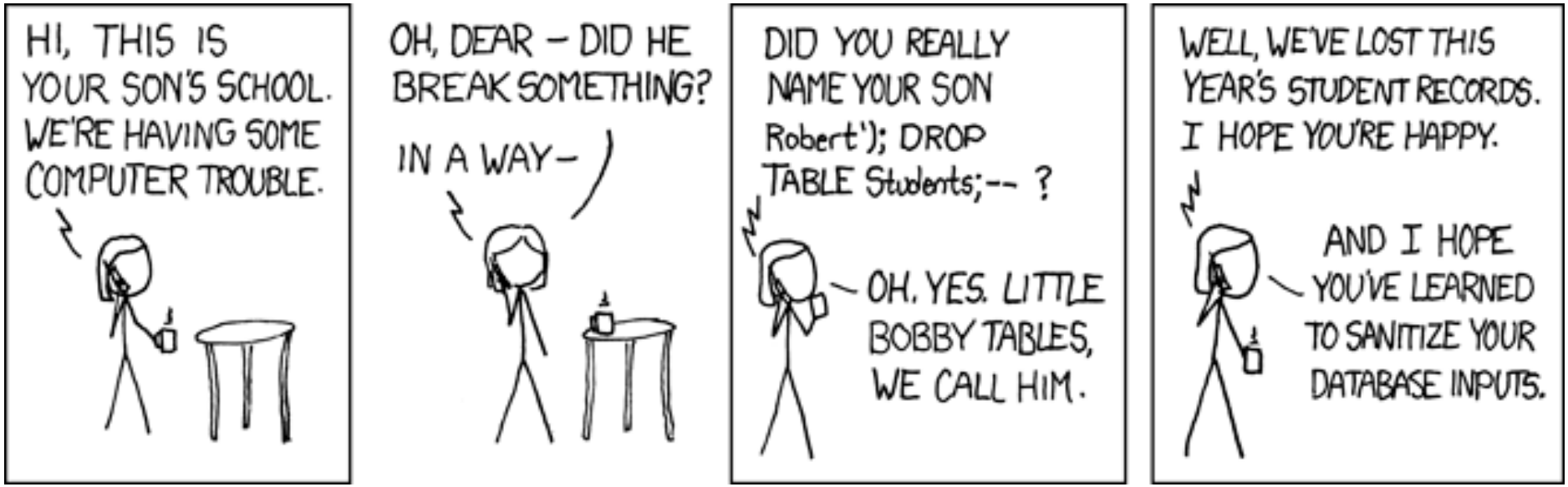


Malicious User Input

SQL Injection Attack

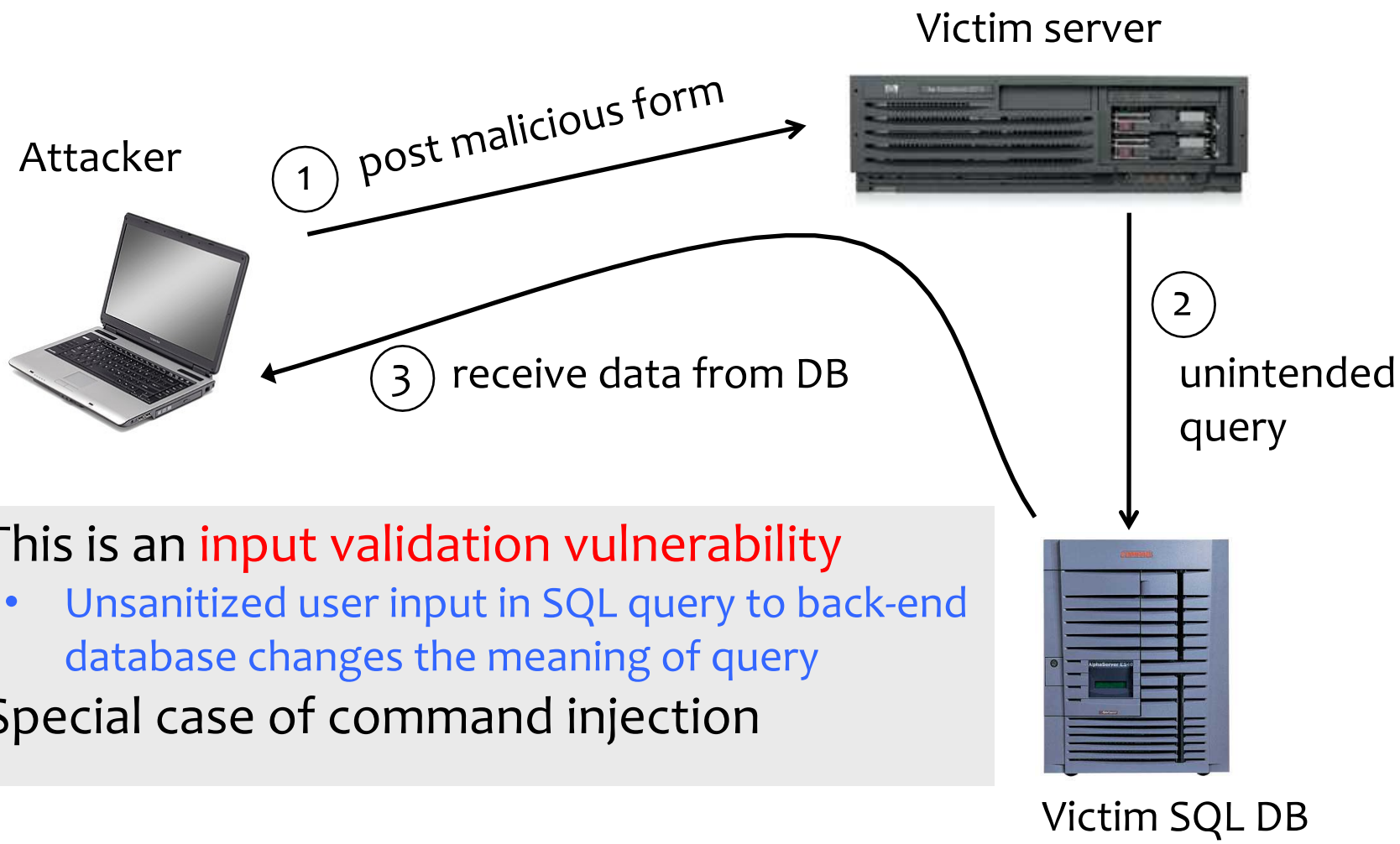


Exploits of a Mom



http://xkcd.com/327/

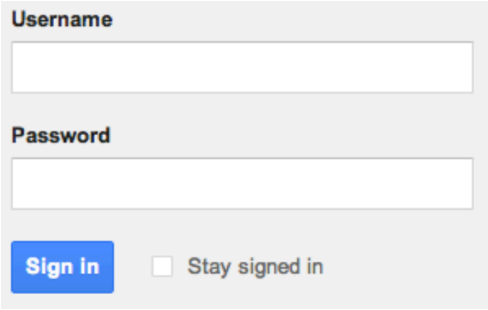
SQL Injection: Basic Idea



- This is an input validation vulnerability
  - Unsanitized user input in SQL query to back-end database changes the meaning of query
- Special case of command injection

Authentication with Backend DB

```
set UserFound = execute(
    "SELECT * FROM UserTable WHERE
    username= ' " & form("user") & " ' AND
    password= ' " & form("pwd") & " ' " );
```



User supplies username and password, this SQL query checks if user/password combination is in the database

```
If not UserFound.EOF
    Authentication correct
  else Fail
```

Only true if the result of SQL query is not empty, i.e., user/pwd is in the database

Using SQL Injection to Log In

- User gives username **’ OR 1=1 --**
- Web server executes query

```
set UserFound=execute(
    SELECT * FROM UserTable WHERE
    username= ‘ ’ OR 1=1 -- … );
```

Always true!

Everything after -- is ignored!

- Now <u>all</u> records match the query, so the result is not empty ⇒ correct “authentication”!

Preventing SQL Injection

- Validate all inputs
  - Filter out any character that has special meaning
    - Apostrophes, semicolons, percent, hyphens, underscores, …
    - Use escape characters to prevent special characters form becoming part of the query code
      - E.g.: escape(O'Connor) = O\'Connor
  - Check the data type (e.g., input must be an integer)

Prepared Statements

```
PreparedStatement ps =
    db.prepareStatement("SELECT pizza, toppings, quantity, order_day "
                + "FROM orders WHERE userid=? AND order_month=?");
ps.setInt(1, session.getCurrentUserId());
ps.setInt(2, Integer.parseInt(request.getParamenter("month")));
ResultSet res = ps.executeQuery();
```

Bind variable (data placeholder)

- Bind variables: placeholders guaranteed to be data (not code)
- Query is parsed without data parameters
- Bind variables are typed (int, string, …)

http://java.sun.com/docs/books/tutorial/jdbc/basics/prepared.html