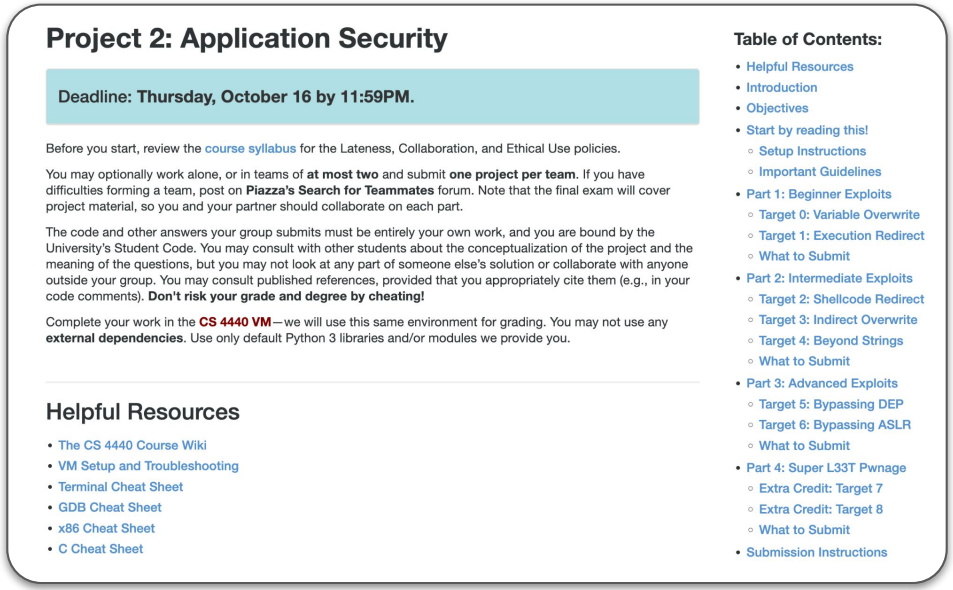
Week 8: Lecture B

Web Attacks

Thursday, October 16, 2025

Announcements

- **Project 2: AppSec** released
  - **Deadline: tonight** by 11:59PM



Project 2: Application Security

Deadline: Thursday, October 16 by 11:59PM.

Before you start, review the course syllabus for the Lateness, Collaboration, and Ethical Use policies.

You may optionally work alone, or in teams of **at most two** and submit **one project per team**. If you have difficulties forming a team, post on **Piazza's Search for Teammates** forum. Note that the final exam will cover project material, so you and your partner should collaborate on each part.

The code and other answers your group submits must be entirely your own work, and you are bound by the University's Student Code. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., in your code comments). **Don't risk your grade and degree by cheating!**

Complete your work in the **CS 4440 VM**—we will use this same environment for grading. You may not use any **external dependencies**. Use only default Python 3 libraries and/or modules we provide you.

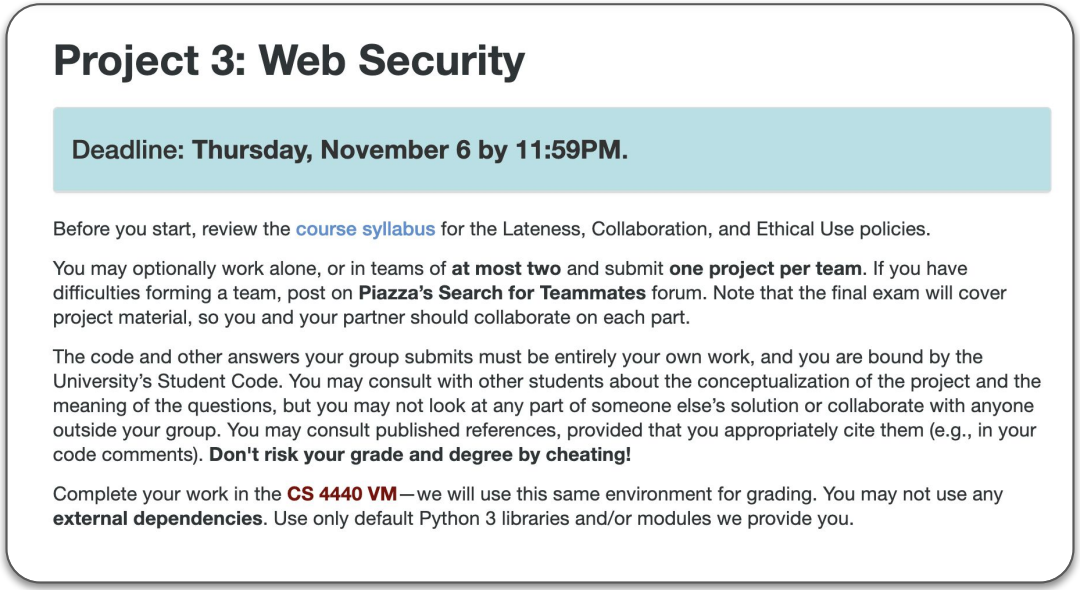
Helpful Resources

- The CS 4440 Course Wiki
- VM Setup and Troubleshooting
- Terminal Cheat Sheet
- GDB Cheat Sheet
- x86 Cheat Sheet
- C Cheat Sheet

Table of Contents:

Announcements

- **Project 3: WebSec** released
  - **Deadline:** Thursday, November 6th by 11:59PM



Project 3: Web Security

Deadline: Thursday, November 6 by 11:59PM.

Before you start, review the course syllabus for the Lateness, Collaboration, and Ethical Use policies.

You may optionally work alone, or in teams of **at most two** and submit **one project per team**. If you have difficulties forming a team, post on **Piazza's Search for Teammates** forum. Note that the final exam will cover project material, so you and your partner should collaborate on each part.

The code and other answers your group submits must be entirely your own work, and you are bound by the University's Student Code. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., in your code comments). **Don't risk your grade and degree by cheating!**

Complete your work in the **CS 4440 VM**—we will use this same environment for grading. You may not use any **external dependencies**. Use only default Python 3 libraries and/or modules we provide you.

Announcements

Announcements

AI and Careers Q&A Panel

When: Thursday Oct 16, 3:45–4:45 PM **Where:** Sorenson Molecular Biotech 2650 SMBB

Come explore key questions shaping the future of work with Artificial Intelligence:

- What is it really like to work with AI in today's industries?
- How is AI transforming different career paths and opportunities?
- How can students prepare now for careers that will involve AI collaboration?
- How do internships or academic research translate into real-world AI careers?

Panelists

Berton Earnshaw
Recursion Pharmaceuticals

Mary Hall
Director & Professor
Kahlert School of Computing

Ashlii Madsen
Senior Vice President
Zions Bancorporation

Abhisek Trivedi
Data Science Manager
Adobe

Suggested Topics

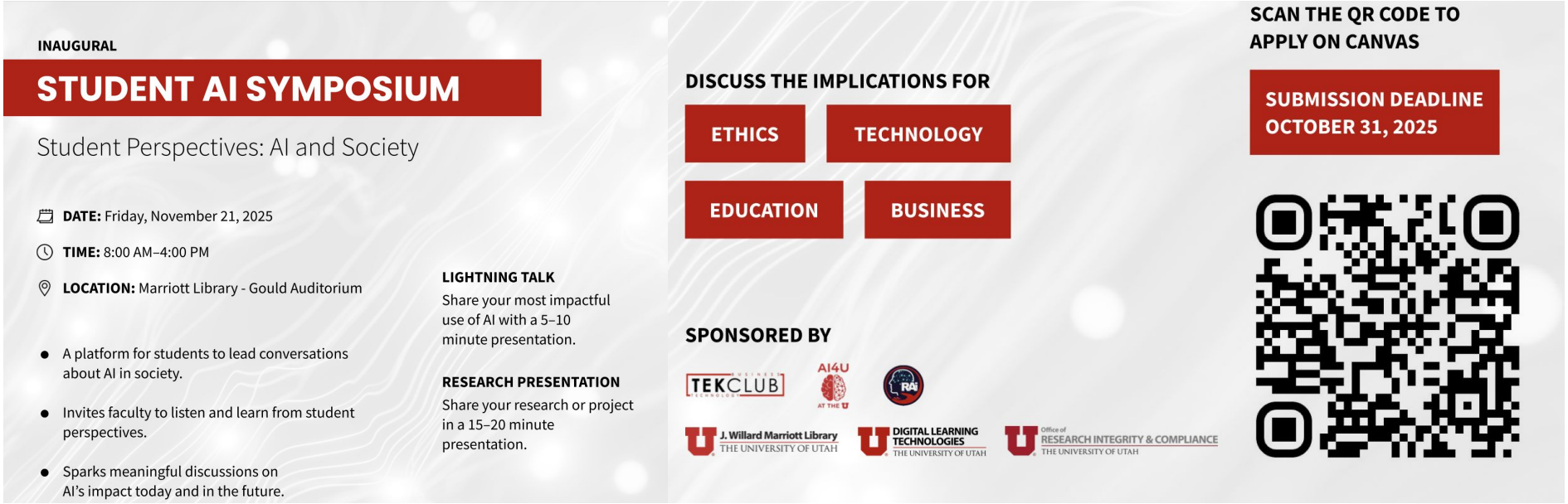
Come prepared with questions about:

- Landing a job in today's market
- Standing out as a new graduate
- How AI is changing careers
- Misconceptions about AI work

RSVP

Hosted by University of Utah ▪ Networking and light refreshments to follow

Announcements



Announcements

WHAT

STUDENT TEAMS OF ALL SKILL LEVELS
WITH INDUSTRY MENTORS

WHEN

Friday, October 24
4:30 PM - 12:00 AM

WHERE

WEB 1230
72 S Central Campus Dr
Salt Lake City, UT 84112

REGISTER

SCAN THE QR CODE
FOR MORE DETAILS
AND TO REGISTER

Announcements

utahsec

See Discord for meeting info!

utahsec.cs.utah.edu

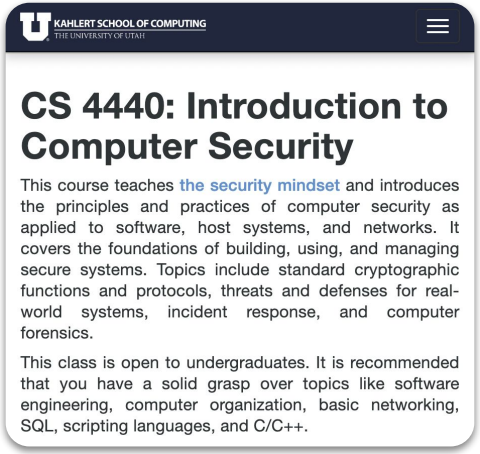
Questions?

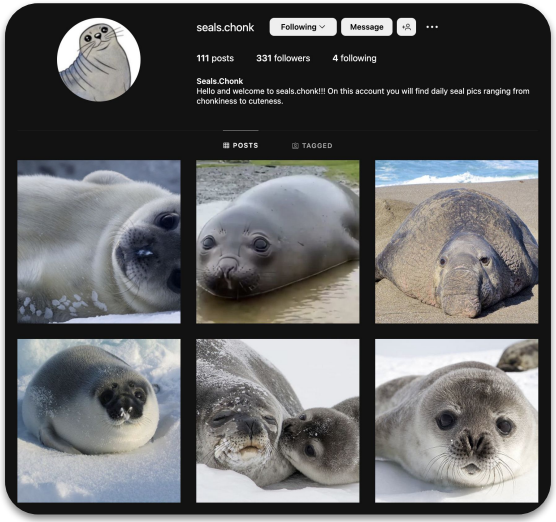
Last time on CS 4440...

Intro to the Web Platform
HTTP
Cookies
Javascript

What is the Web?

- **What is it?**
  - A venue for me to ridicule Broncos fans
  - A place to view (and share) pictures of seals
  - The location where I host the CS 4440 website





Web Security: Two Tales

- **Web Browser** (the **client** side)
  - ???
  - ???

Web Security: Two Tales

- **Web Browser** (the **client** side)
  - Requests a **resource**
  - **Renders** it for the user

Web Security: Two Tales

- **Web Browser** (the **client** side)
  - Requests a **resource**
  - **Renders** it for the user
- **Web Application** (the **server** side)
  - ???
  - ???

Web Security: Two Tales

- **Web Browser** (the **client** side)
  - Requests a **resource**
  - **Renders** it for the user
- **Web Application** (the **server** side)
  - **Transmits** resource to the client
  - **Interfaces** with the client
    - Session cookies to keep “state”
    - Dynamic content (e.g., JavaScript)

Stateless vs. Stateful Communication

- **Stateless**

??

?

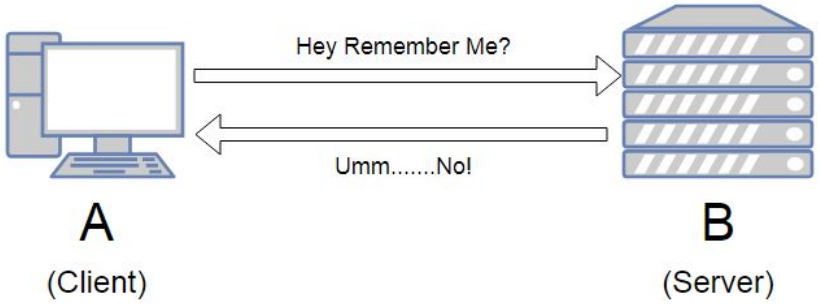
- **Stateful**

??

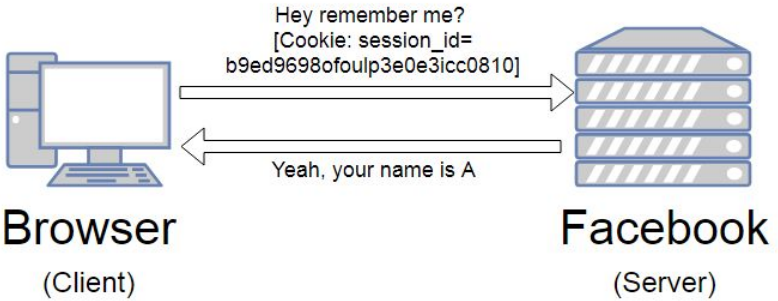
?

Stateless vs. Stateful Communication

- **Stateless**



- **Stateful**

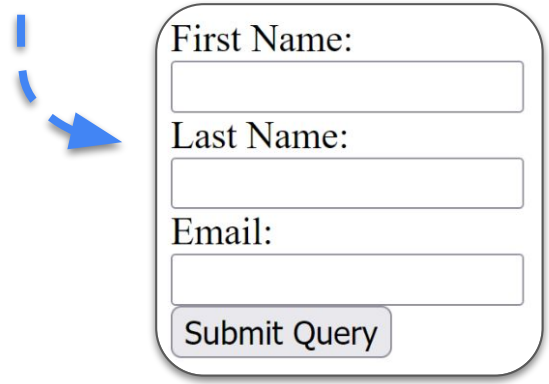


HyperText Markup Language (HTML)

- Describes **content** and **formatting** of web pages
  - Rendered within browser window
- **HTML features**
  - **Static** document description language
  - Links to external pages, images by **reference**
  - User input sent to server via **forms**
- **HTML extensions**
  - Additional media (e.g., PDF, videos) via **plugins**
  - Embedding **programs** in other languages (e.g., **Java**) provides **dynamic content** that can**:**
    - Interacts with the user
    - Modify the browser user interface
    - Access the client computer environment

```
<form action="home.html">
    First Name:<br>
    <input type="text" name="first_name">
</br>
    Last Name:<br>
    <input type="text" name="last_name">
</br>
    Email:<br>
    <input type="text" name="email">
</br>
    <input type="submit" name="Submit">

</form>
```



Uniform Resource Locator (URL)

- **Reference to a web resource** (e.g., a website)
  - Specifies its **location** on a computer network
  - Specifies the mechanism for **retrieving it**

- **Example:** `http://www.cs.utah.edu/class?name=cs4440#homework`
  - **Protocol:** How to **retrieve** the web resource
  - **Path:** Identifies the **specific resource** to access (case insensitive)
  - **Query:** Assigns values to specified **parameters** (case sensitive)
  - **Fragment:** Location of a **resource subordinate** to another

Uniform Resource Locator (URL)

- **Reference to a web resource** (e.g., a website)
  - Specifies its **location** on a computer network
  - Specifies the mechanism for **retrieving it**

- **Example:** `http://www.cs.utah.edu/class?name=cs4440#homework`
  - **Protocol:** How to **retrieve** the web resource
    - `HTTP`
  - **Path:** Identifies the **specific resource** to access (case insensitive)
    - `www.cs.utah.edu/class`
  - **Query:** Assigns values to specified **parameters** (case sensitive)
    - `name=cs4440`
  - **Fragment:** Location of a **resource subordinate** to another
    - `#homework`

HTTP Requests

- What type of HTTP request is this?

```
<form action="http://cs4440.eng.utah.edu/project3/login?" method="POST">
    <input name="username" value="attacker" type="hidden"/>
    <input name="password" value="l33th4x" type="hidden"/>
</form>
```

What type of HTTP request is this?

GET request

0%

POST request

0%

None of the above

0%

HTTP Requests

- What type of HTTP request is this? **POST**

```
<form action="http://cs4440.eng.utah.edu/project3/login?" method="POST">
    <input name="username" value="attacker" type="hidden"/>
    <input name="password" value="l33th4x" type="hidden"/>
</form>
```

- What about this?

```
http://cs4440.eng.utah.edu/project3/search?q=Test
```

HTTP Requests

- What type of HTTP request is this? **POST**

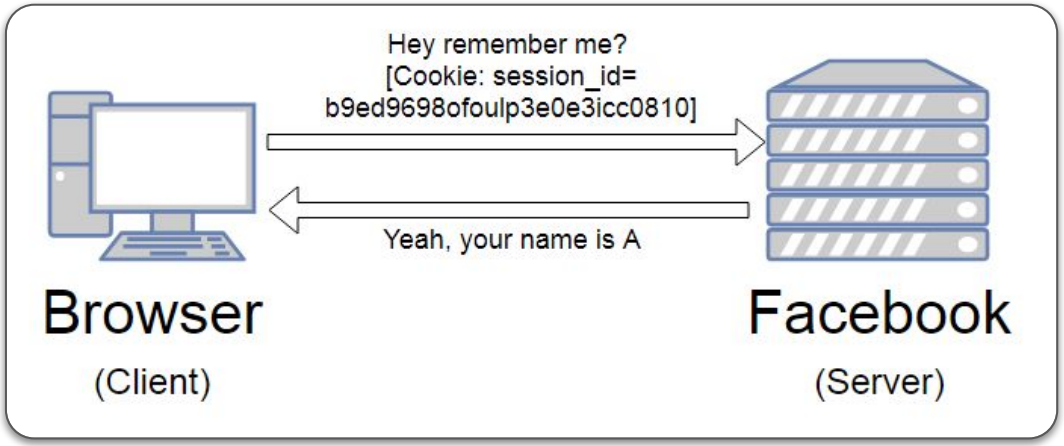
```
<form action="http://cs4440.eng.utah.edu/project3/login?" method="POST">
    <input name="username" value="attacker" type="hidden"/>
    <input name="password" value="l33th4x" type="hidden"/>
</form>
```

- What about this? **GET**

```
http://cs4440.eng.utah.edu/project3/search?q=Test
```

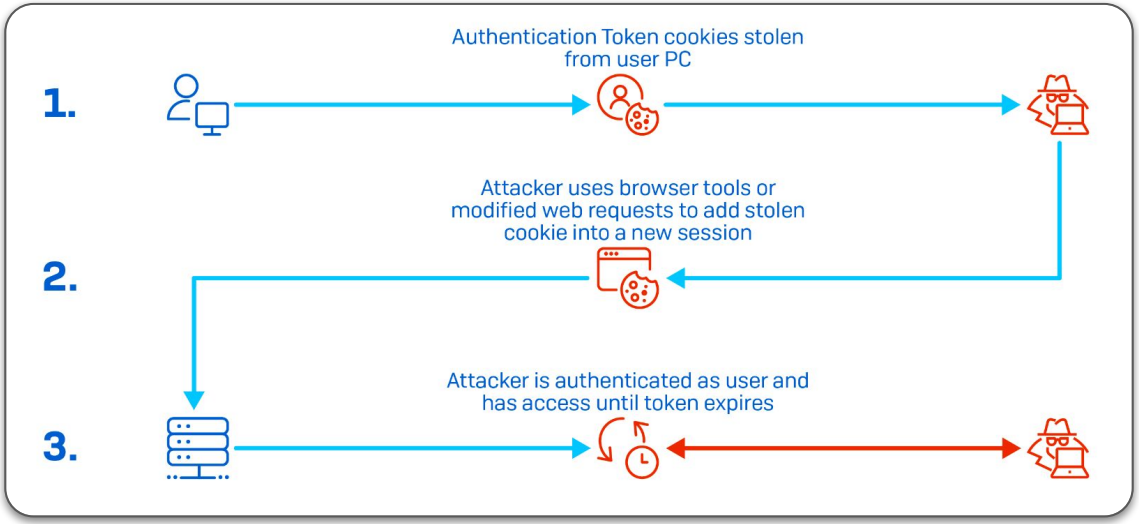
HTTP Cookies

- **Small chunks of info stored on a computer associated with a specific server**
  - When you access a website, it might store information as a cookie
  - Every time you visit that server, the cookie is re-sent to the server
  - **Effectively used to hold state information over multiple sessions**



HTTP Cookies

- **Cookies are stored on your computer and can be controlled or manipulated**
  - Many sites require that you enable cookies to access the site's full capabilities
  - Their storage on your computer naturally **lends itself to cookie exploitation**



JavaScript

- **A powerful, popular web programming language**
  - Scripts embedded in web pages returned by web server
  - Scripts **executed** by browser (client-side scripting). Can:
    - Alter contents of a web page
    - Track events (mouse clicks, motion, keystrokes)
    - Read/set cookies
    - Issue web requests and read replies

Embedding JavaScript within HTML

- Code enclosed within **<script>** tags
- **Defining functions**

```
<script type="text/javascript">
    function hello() { alert("Hello world!"); }
</script>
```

- **Event handlers** embedded in HTML

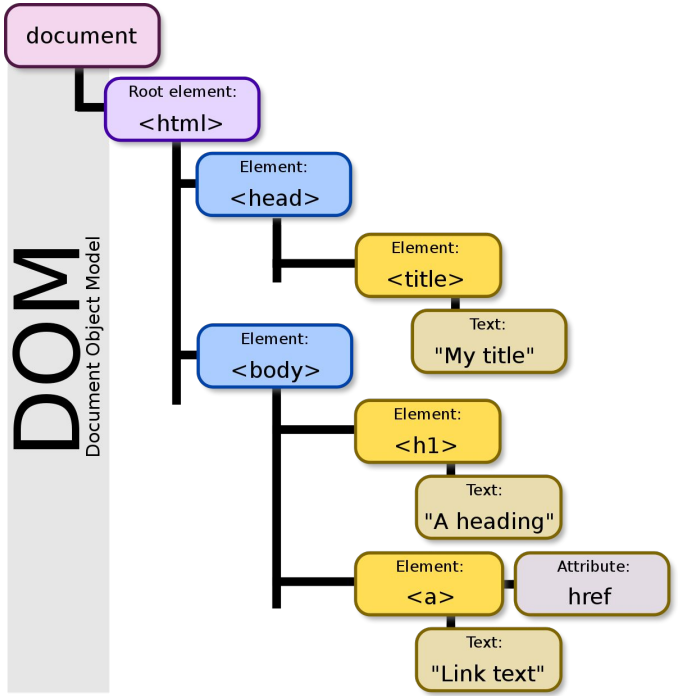
```
<img src="picture.gif"
onMouseOver="javascript:hello()">
```

- **Built-in functions** **can change content** of a window: **click-jacking attack**

```
<a onMouseUp="window.open('http://www.evilsite.com')"
href="http://www.trustedsite.com/">Trust me!?</a>
```

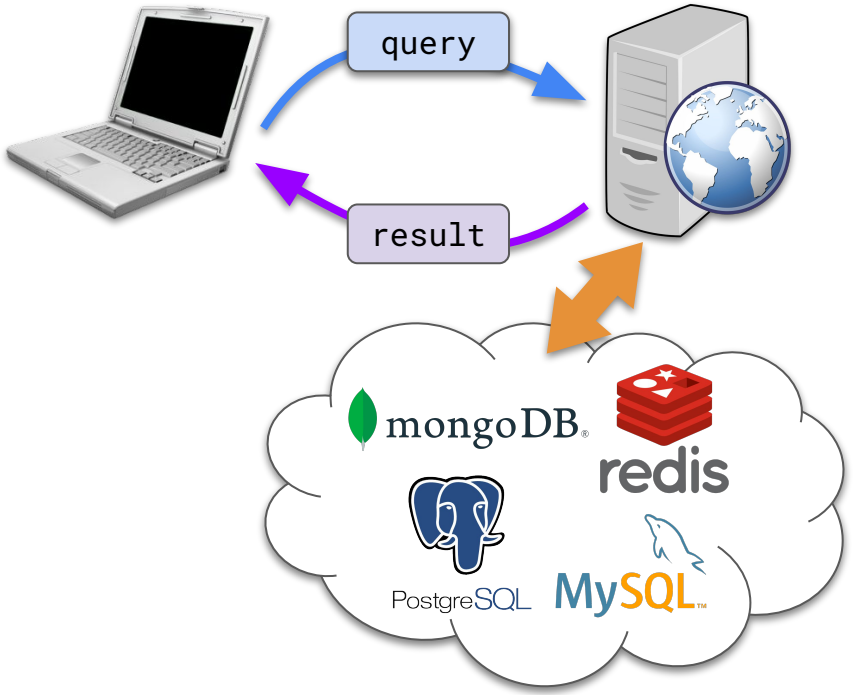
Document Object Model (DOM Tree)

- **Platform- and language-neutral interface**
  - Allows programs and scripts to dynamically **access/update** document **content**, **structure**, **style**
- Backbone of modern web browser plugins
- You can access and update the DOM Tree yourself via browser's **web developer tools**



Web Databases

- **Databases:** how we store data on the server-side
  - Data **stored** by **server**
  - Data **queried** by **client**
  - Query **executed** by **server**
- A massive component of modern web applications
  - **Examples:** record keeping, user account management
- Popular DB Software:
  - MySQL, PostgreSQL
  - Redis, MongoDB



Structured Query Language (SQL)

- **A language to ask ("query") databases questions**
  - Information stored in **tables**; columns = **attributes**, rows = **records**
- **Fundamental operations:**
  - "`SELECT`" : express queries
  - "`INSERT`" : create new records
  - "`UPDATE`" : modify existing data
  - "`DELETE`" : delete existing records
  - "`UNION`" : combine results of multiple queries
  - "`WHERE`/`AND`/`OR`" : conditional operations
- **Syntactical Tips:**
  - "`*`" : all
  - "`NULL`" : nothing
  - "`-- `" : comment-out the rest of the line (note the space at the end)

Structured Query Language (SQL)

- **A language to ask ("query") databases questions**
- E.g, How many users have the location **Salt Lake City**?
  - `"SELECT COUNT(*) FROM 'users' WHERE location='Salt Lake City'"`
- E.g., Is there a user with username "**bob**" and password "**abc123**"?
  - `"SELECT * FROM 'users' WHERE username='bob' AND password='abc123'"`
- E.g., Completely delete this table!
  - `"DROP TABLE 'users'"`

Example DB and SQL Queries

- **Table name:** `users`

ID	username	password	passHash	location
1	Prof Nagy	c4ntgu3$$m3!	0x12345678	Salt Lake, UT
2	Average User	password123	0x87654321	Boulder, CO
3	Below Average	password	0x81726354	Denver, CO

- `SELECT * FROM users WHERE passHash = 0x87654321;`
  - ???
- `SELECT * FROM users WHERE id = 1;`
  - ???
- `SELECT password FROM users WHERE username = “Below Average”;`
  - ???

Example DB and SQL Queries

- **Table name:** `users`

ID	username	password	passHash	location
1	Prof Nagy	c4ntgu3$$m3!	0x12345678	Salt Lake, UT
2	Average User	password123	0x87654321	Boulder, CO
3	Below Average	password	0x81726354	Denver, CO

- `SELECT * FROM users WHERE passHash = 0x87654321;`
  - Will return **Average User**
- `SELECT * FROM users WHERE id = 1;`
  - Will return just **Prof Nagy**
- `SELECT password FROM users WHERE username = “Below Average”;`
  - Will return **Below Average’s password**

Questions?

This time on CS 4440...

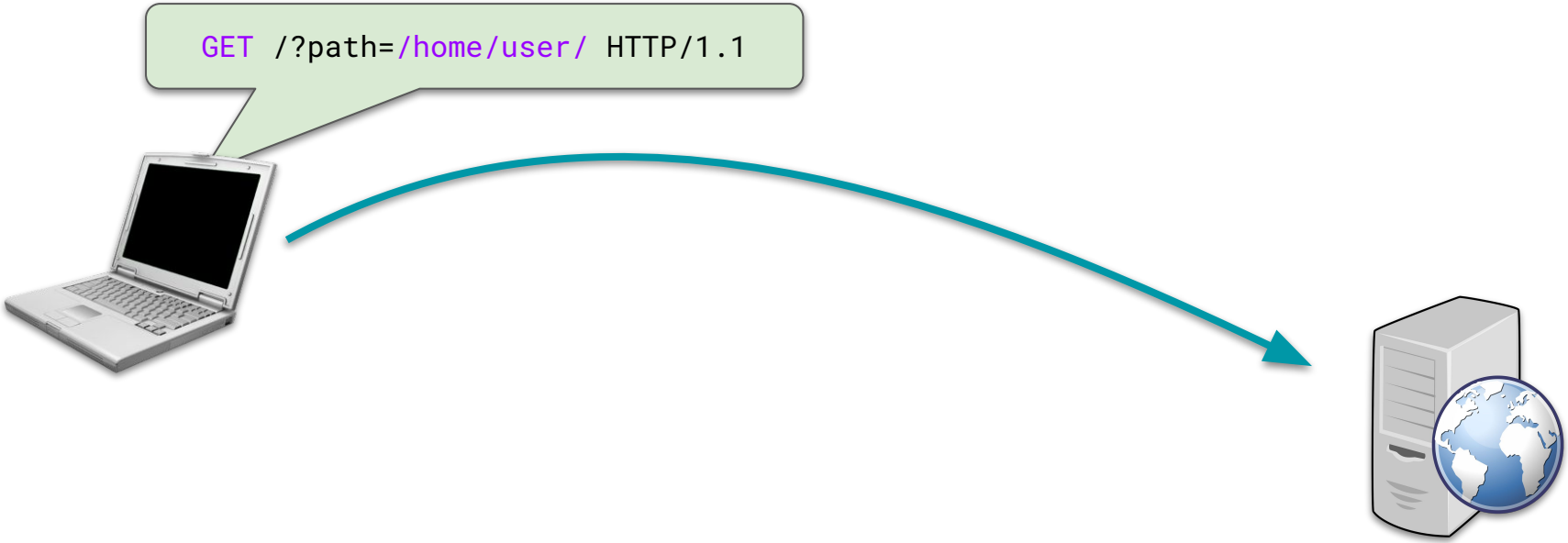
Web Attacks
SQL Injection
Cross-site Scripting
Cross-site Request Forgery
Project 3 Tips

Food for Thought

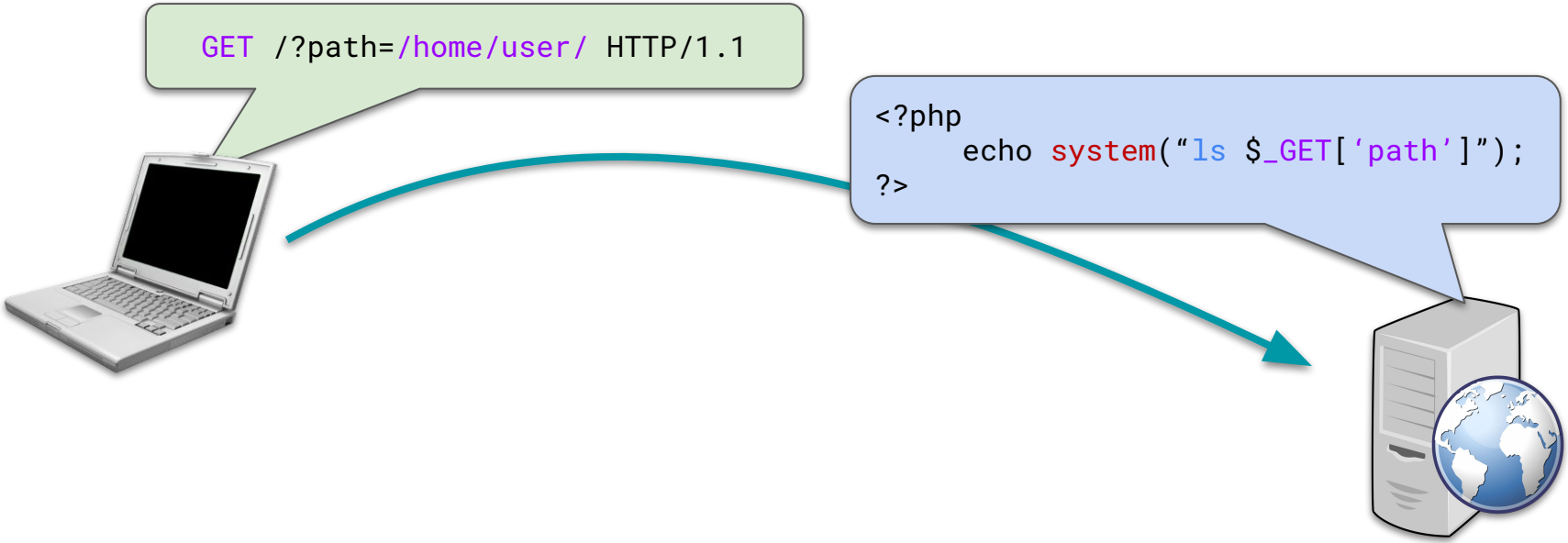
- SQL databases and other web applications operate on **users' inputs**
  - E.g., SQL queries, HTTP GET and POST requests
  - That's how we interact with their **server-side applications**!

- **Question:** can we assume that all user input will only ever be **data**?

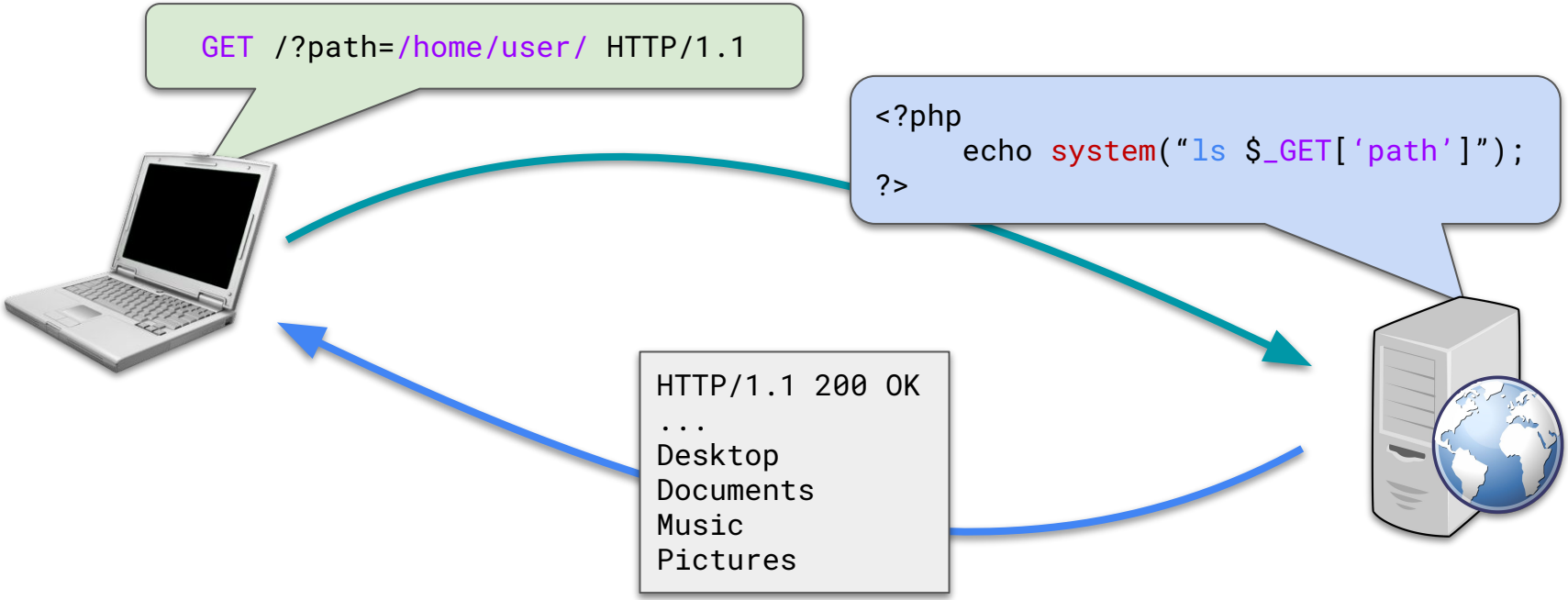
Web Applications



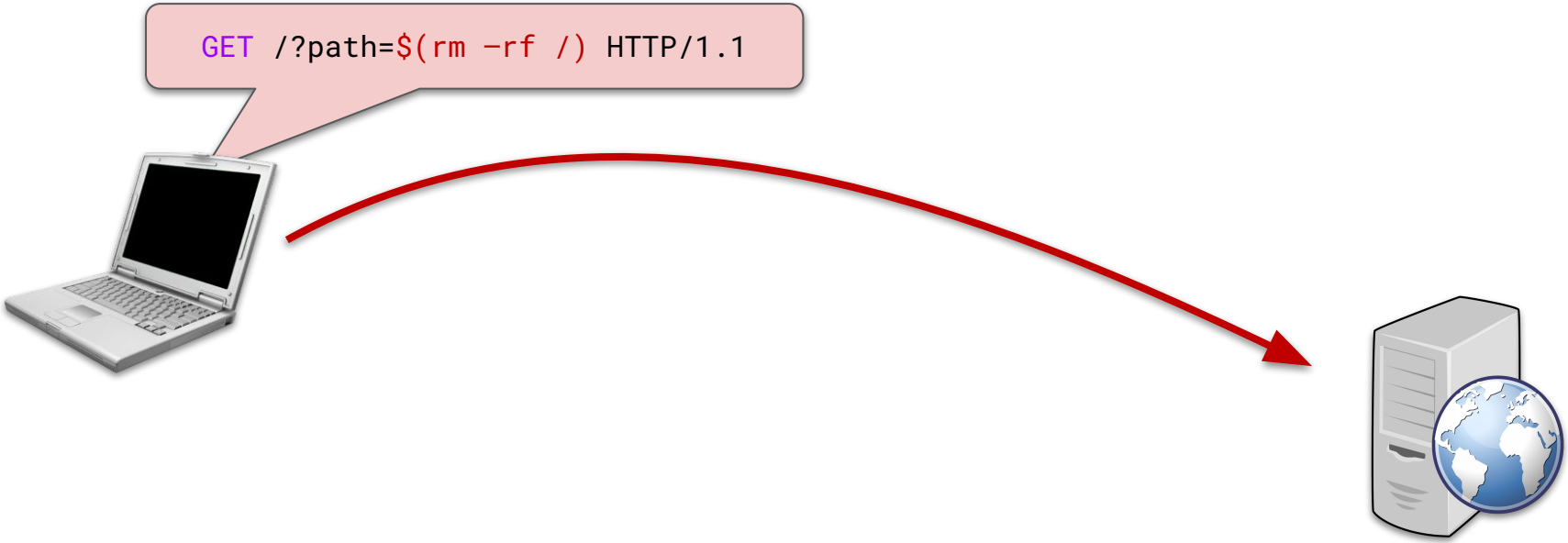
Web Applications



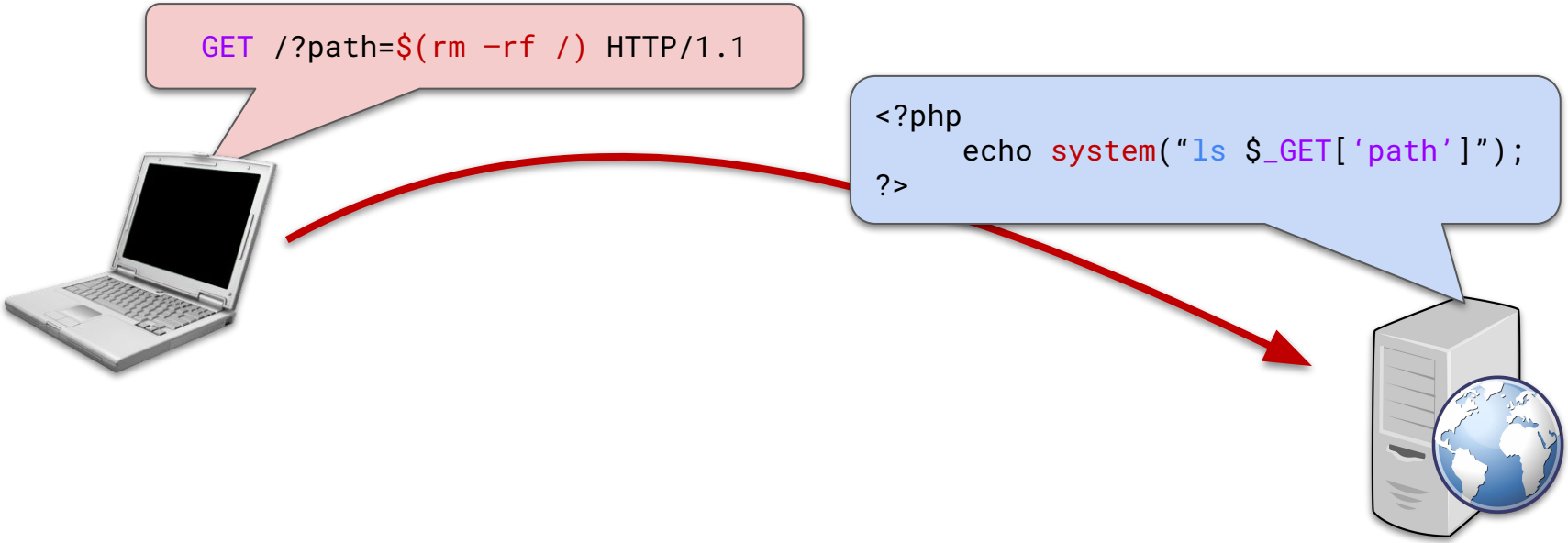
Web Applications



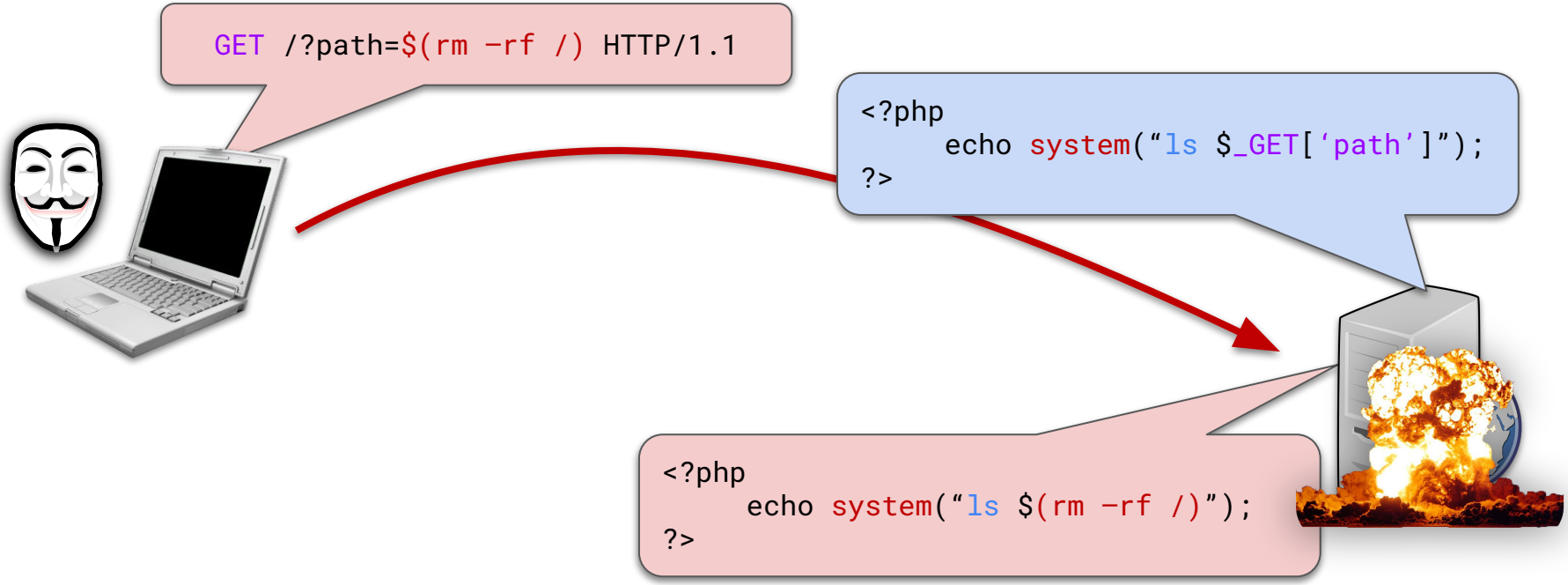
Web Applications



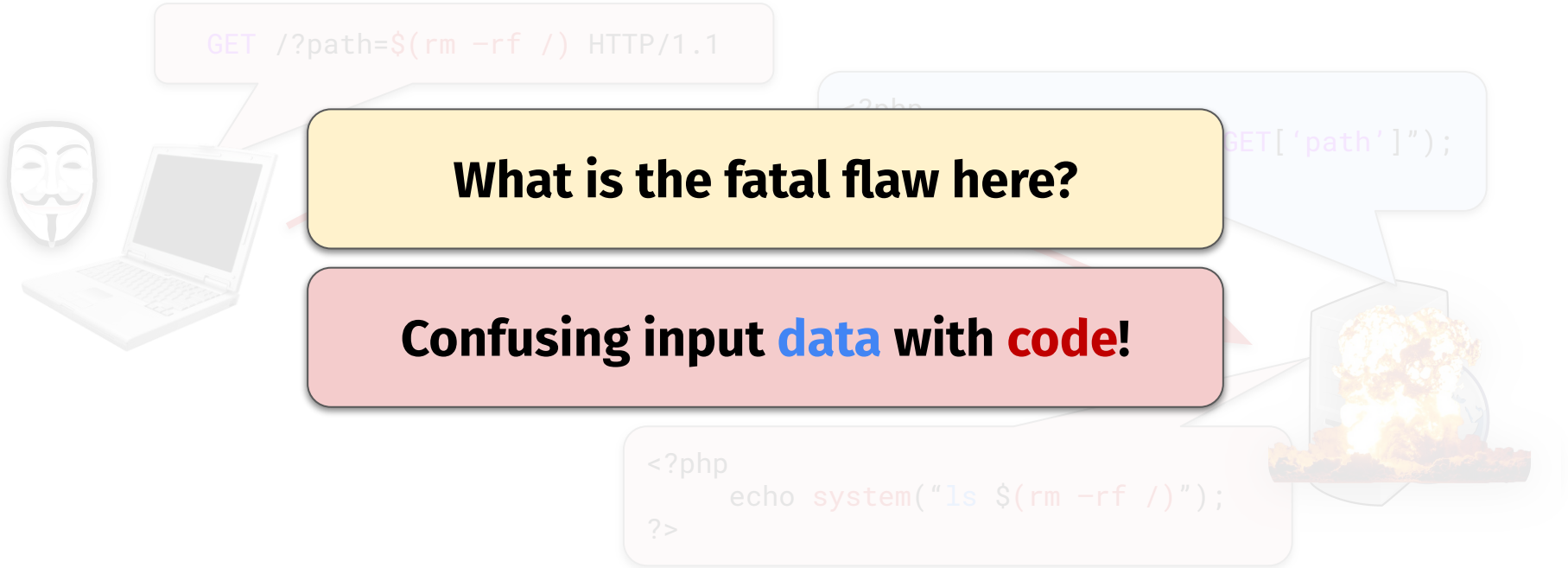
Web Applications



Web Applications



Web Applications

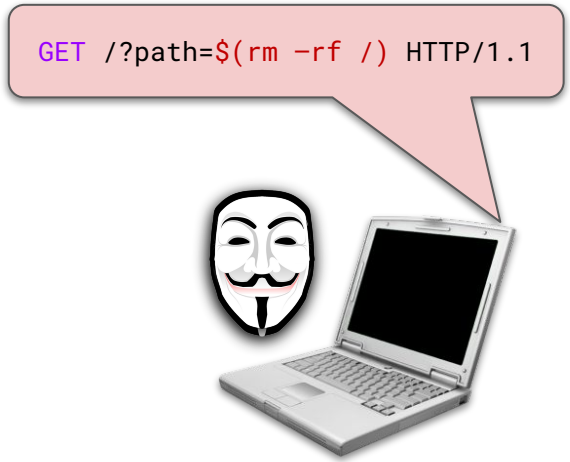


What is the fatal flaw here?

Confusing input data with code!

Code Injection

- Confusing **data** with **code**
  - Programmer expected user would only send data
  - Instead, got (and **unintentionally executed**) code
- **A common and dangerous class of attacks**
  - Shell Injection
  - SQL Injection
  - Cross-Site Scripting
  - Control-flow Hijacking (buffer overflows)



SQL Injection

Recap: SQL Queries

- **A language to ask (“query”) databases questions**
- E.g, How many users have the location **Salt Lake City**?
  - `“SELECT COUNT(*) FROM 'users' WHERE location='Salt Lake City'”`
- E.g., Is there a user with username “**bob**” and password “**abc123**”?
  - `“SELECT * FROM 'users' WHERE username='bob' AND password='abc123'”`
- E.g., Completely delete this table!
  - `“DROP TABLE 'users'”`

Recap: Structured Query Language (SQL)

- A language to ask ("
- E.g, How many users
  - "SELECT COUNT(*
- E.g., Is there a user w
  - "SELECT * FROM
- E.g., Completely dele
  - "DROP TABLE `us

SQL Injection Attacks

- **Target:** web server hosting a **SQL database**
  - One of the most popular database languages today

SQL Injection Attacks

- **Target:** web server hosting a **SQL database**
  - One of the most popular database languages today
- **Attacker goal:** inject or modify database **commands** to **read** or **alter** database info

SQL Injection Attacks

- **Target:** web server hosting a **SQL database**
  - One of the most popular database languages today
- **Attacker goal:** inject or modify database **commands** to **read** or **alter** database info
- **Attacker tools:** ability to send **requests** to web server (e.g., via an ordinary browser)

SQL Injection Attacks

- **Target:** web server hosting a **SQL database**
  - One of the most popular database languages today
- **Attacker goal:** inject or modify database **commands** to **read** or **alter** database info
- **Attacker tools:** ability to send **requests** to web server (e.g., via an ordinary browser)
- **Key trick:** web server **allows characters** in attacker's input to be **interpreted as SQL** control elements (rather than just as data)

A Simple Command Injection

- **Consider an SQL query where the attacker chooses $id:**

```
SELECT * FROM users WHERE id = $id;
```

- **What can an attacker do?**

A Simple Command Injection

- **Consider an SQL query where the attacker chooses $id:**

```
SELECT * FROM users WHERE id = $id;
```

- **What can an attacker do?**
  - `$id = NULL UNION SELECT * FROM users`

- **Effect upon execution?**

SELECT * FROM users WHERE id = NULL UNION SELECT * FROM users;

Returns the user whose id is "NULL"

0%

Returns no users since no user has id "NULL"

0%

None of the above

0%

A Simple Command Injection

- **Consider an SQL query where the attacker chooses $id:**

```
SELECT * FROM users WHERE id = $id;
```

- **What can an attacker do?**
  - `$id = NULL UNION SELECT * FROM users`

- **Effect upon execution?**

```
SELECT * FROM users WHERE id =
NULL UNION SELECT * FROM users;
```

  - Will return the full list of users in the database!

Abusing Comment Encoding

- **Consider an SQL query where the attacker chooses $name and $ssn:**

```
SELECT * FROM faculty WHERE name = $name AND ssn = $ssn
```

- **What can an attacker do?**

Abusing Comment Encoding

- **Consider an SQL query where the attacker chooses $name and $ssn:**

```
SELECT * FROM faculty WHERE name = $name AND ssn = $ssn
```

- **What can an attacker do?**
  - $name = "'StefanNagy'"
  - $ssn = ????????????????

Abusing Comment Encoding

- **Consider an SQL query where the attacker chooses $name and $ssn:**

```
SELECT * FROM faculty WHERE name = $name AND ssn = $ssn
```

- **What can an attacker do?**
  - $name = "'StefanNagy' --"
  - String " -- " is MySQL **code-comment syntax**

- **Effect upon execution?**

Abusing Comment Encoding

- **Consider an SQL query where the attacker chooses $name and $ssn:**

```
SELECT * FROM faculty WHERE name = $name AND ssn = $ssn
```

- **What can an attacker do?**
  - $name = "'StefanNagy' --"
  - String " -- " is MySQL **code-comment syntax**

- **Effect upon execution?**

```
SELECT * FROM faculty WHERE name =
  'StefanNagy' -- AND ssn = $ssn;
```

  - Can be leveraged to **discard remaining clauses** of the query

Bypassing String Escaping

- **Consider an SQL query where the attacker chooses $city:**

```
SELECT * FROM users WHERE location='$city';
```

- **How can we bypass the single-quotes?**

Bypassing String Escaping

- **Consider an SQL query where the attacker chooses $city:**

```
SELECT * FROM users WHERE location='$city';
```

- **How can we bypass the single-quotes?**
  - `$city = SLC'; DELETE FROM users WHERE 1='1`
  - We add **two single-quotes:** one after city name, the other near query end

- **Effect on the query?**

Bypassing String Escaping

- **Consider an SQL query where the attacker chooses $city:**

```
SELECT * FROM users WHERE location='$city';
```

- **How can we bypass the single-quotes?**
  - $city = SLC'; DELETE FROM users WHERE 1='1
  - We add **two single-quotes:** one after city name, the other near query end

- **Effect on the query?**

```
SELECT * FROM users WHERE location = 'SLC';
      DELETE FROM users WHERE 1='1';
```

  - Our two quotation marks will "**escape**" (i.e., **close-out**) the city name
  - In this scenario, escaping allows us to **modify the query** with additional logic

Abusing String Arithmetic

- **Consider an SQL query where the attacker chooses `$city`:**

```
SELECT * FROM users WHERE location='$city';
```

- **What can an attacker do?**
  - `$city = anything' = '`
  - The second quote creates **an empty string** on the right-hand side

- **Effect on the query?**

Abusing String Arithmetic

- **Consider an SQL query where the attacker chooses $city:**

```
SELECT * FROM users WHERE location='$city';
```

- **What can an attacker do?**
  - $city = anything' = '
  - The second quote creates **an empty string** on the right-hand side

- **Effect on the query?**

```
SELECT * FROM users WHERE location =
    'anything' = '';
```

Abusing String Arithmetic

- **Consider an SQL query where the attacker chooses $city:**

```
SELECT * FROM users WHERE location='$city';
```

- **What can an attacker do?**
  - $city = anything' = '
  - The second quote creates **an empty string** on the right-hand side

- **Effect on the query?**

```
SELECT * FROM users WHERE location =
    'anything' = '';
```

  - The query statement will always evaluate to **TRUE**
  - Forcing a true statement will force **the entire query to be true**

Abusing String Arithmetic

- Consider

```
WHERE location = 'anything' = '';
```

- What can an attacker do?
  - `$city = anything' = '`
  - The second quote creates **an empty string** on the right-hand side
- Effect on the query?

```
SELECT * FROM users WHERE location =
'anything' = '';
```

  - The query statement will always evaluate to **TRUE**
  - Forcing a true statement will force **the entire query to be true**

Abusing String Arithmetic

```
WHERE location = 'anything' = '';
```

```
(str) location == (str) 'anything'
```

FALSE

Abusing String Arithmetic

```
WHERE location = 'anything' FALSE = '';
```

```
(str) location == (str) 'anything'
```

FALSE

```
FALSE == ''
```

Abusing String Arithmetic

```
WHERE location = 'anything' FALSE = '';
```

```
(str) location == (str) 'anything'
```

FALSE

```
(bool) FALSE == (str) ''
```

Type Mismatch!

Abusing String Arithmetic

```
WHERE location = 'anything' FALSE = '';
```

```
(str) location == (str) 'anything'
```

FALSE

```
(bool) FALSE == (str) ''
```

Type Mismatch!

```
(int) FALSE == (int) ''
```

Abusing String Arithmetic

```
WHERE location = 'anything' FALSE = '';
```

```
(str) location == (str) 'anything'
```

FALSE

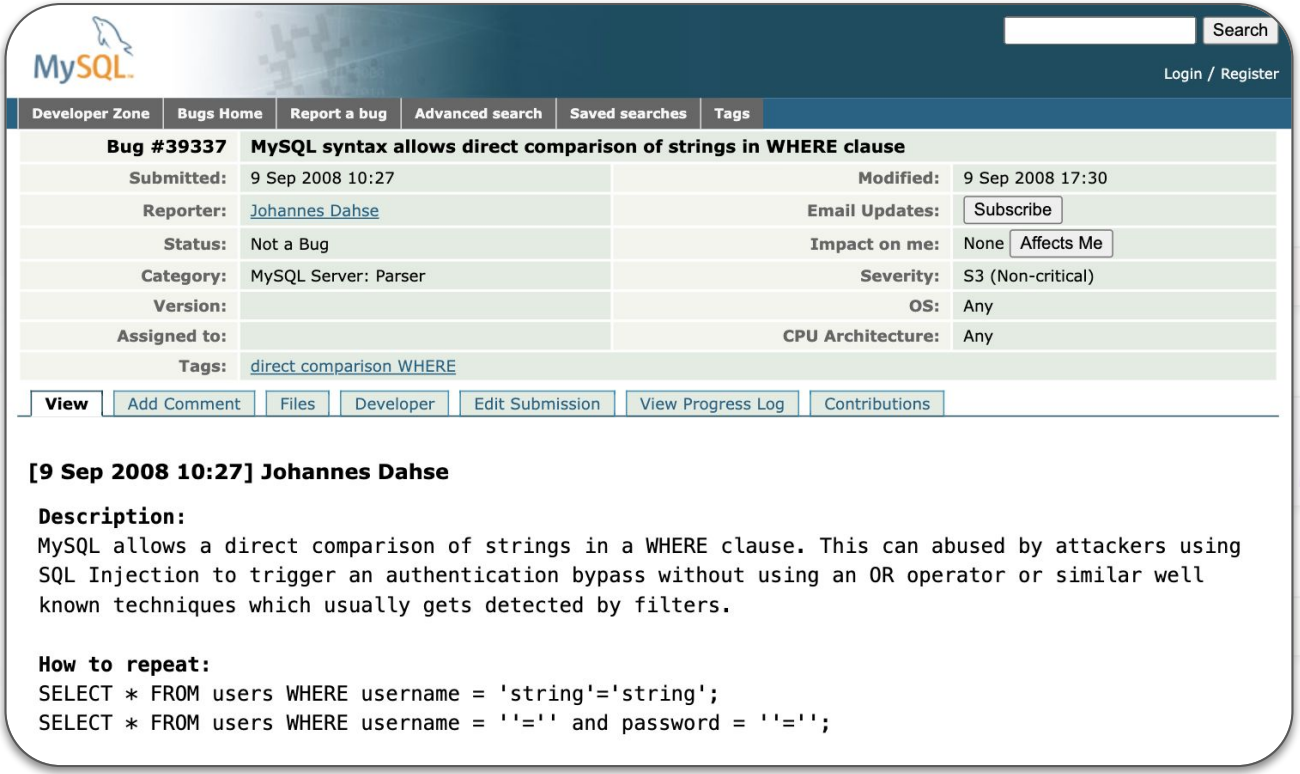
```
(bool) FALSE == (str) ''
```

Type Mismatch!

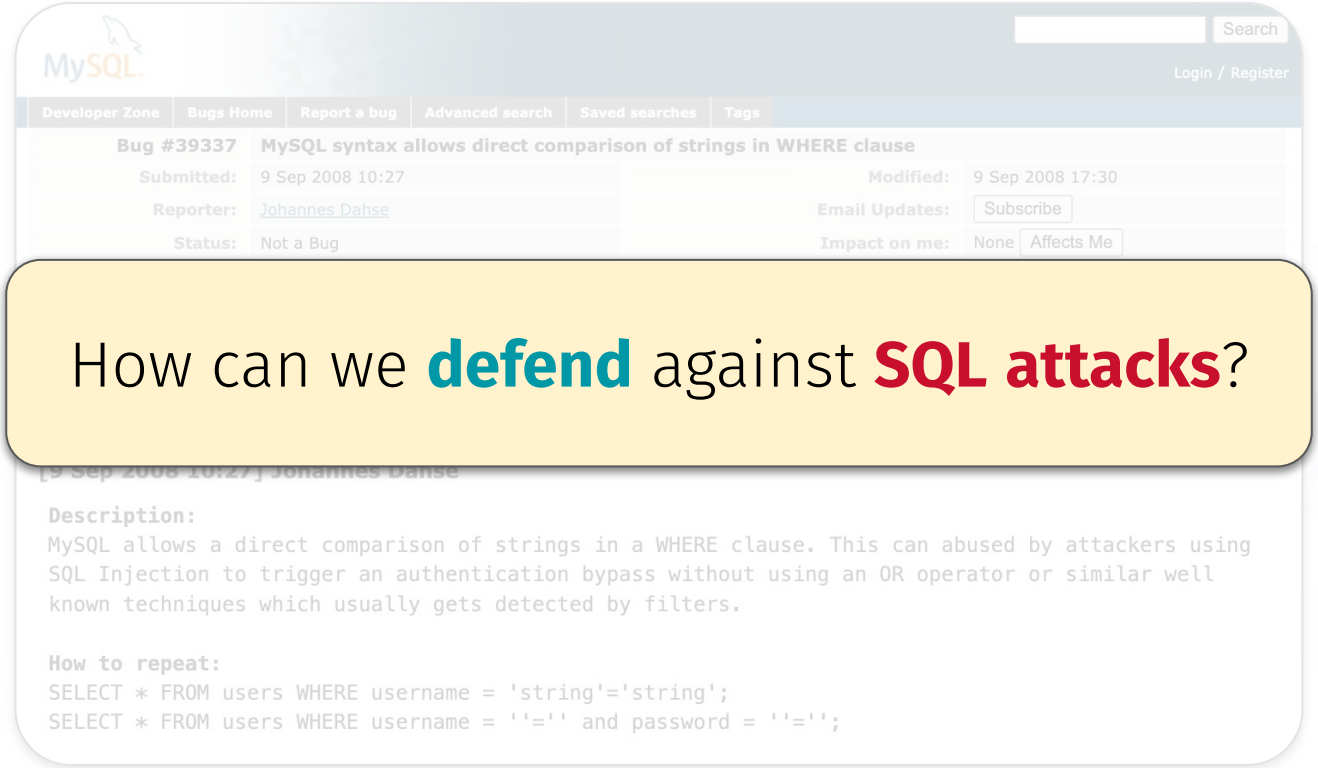
```
(int) FALSE 0 == (int) '' 0
```

TRUE

Abusing String Arithmetic



Abusing String Arithmetic



How can we **defend** against **SQL attacks**?

Preventing SQL Injection

- **Input Sanitization:** identify and **escape** non-data input
  - **Escaping** = to handle differently
  - Usually just cut-out that part

- Common escaping targets:
  - SQL **control** characters (quotes, comments, etc.)
  - SQL **command** keywords (DELETE, WHERE, FROM, etc.)

- **Result:** attack query interpreted as **garbage**—and fails!

Preventing SQL Injection

- **Example:** escaping single quotes

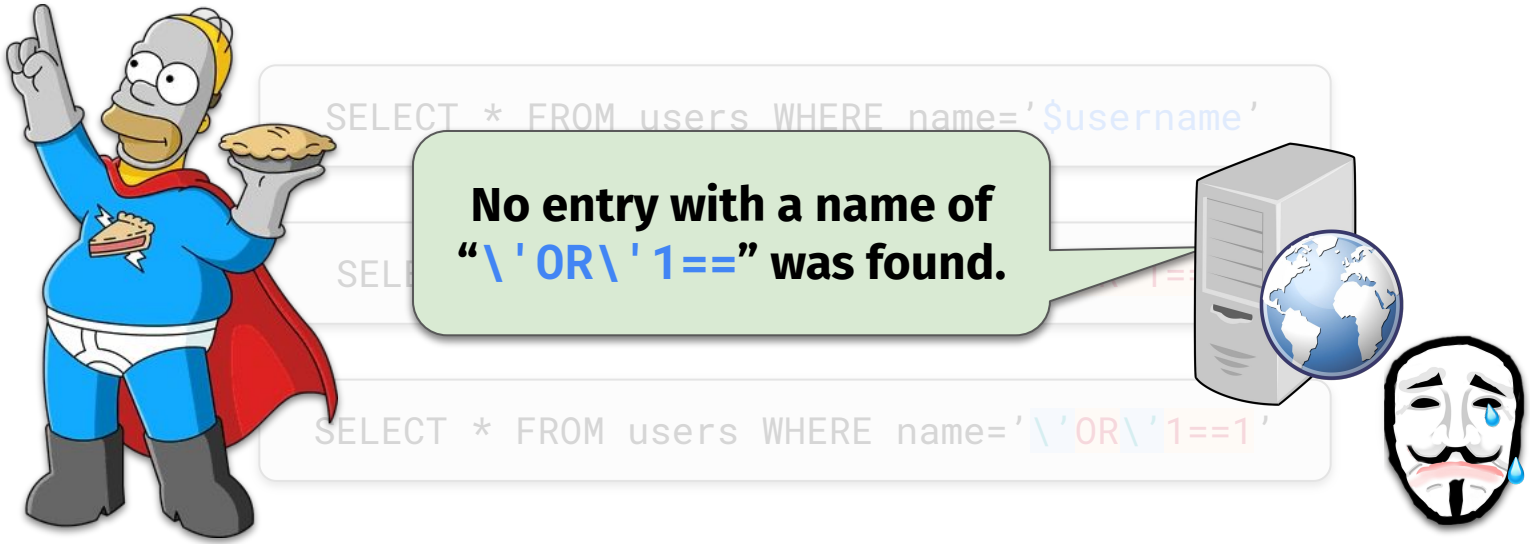
```
SELECT * FROM users WHERE name='$username'
```

```
SELECT * FROM users WHERE name=''OR'1==1'
```

```
SELECT * FROM users WHERE name='\'OR\'1==1'
```

Preventing SQL Injection

- **Example:** escaping single quotes



Preventing SQL Injection

- **Prepared Statements:** “pin” data elements
  - Declares what parts of the query are data **prior to** the user’s input making its way into the query

- **Example:**

```
$st = $db->prepare("SELECT * FROM users WHERE name=?");
$stmt->bind_param(“s”, $username);
$stmt->execute();
```

```
$username='' OR '1==1'
```

Preventing SQL Injection

- **Prepared Statements:** "pin" data elements
  - Declares what parts of the query are data **prior** to use's input making its way into the query
- **Pin**

No entry with a name of "'OR'1==" was found.

```
$db-                                    WHERE
  ->bind_param("s", $username);
  ->execute();
```

$username='OR'1==1'

Questions?

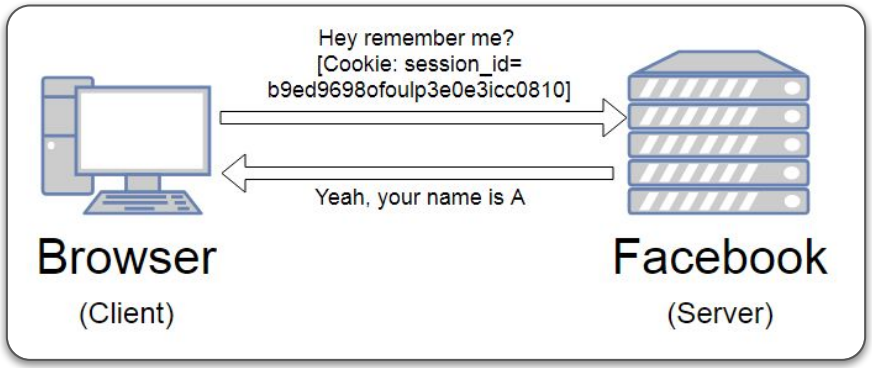
Cross-site Request Forgery (CSRF)

Cookie Chaos

- Cookies enable **???**

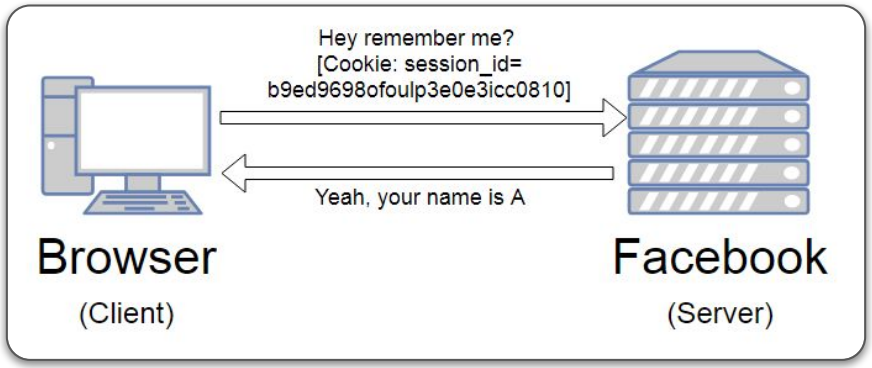
Cookie Chaos

- Cookies enable **persistent** **interaction**
  - Even **after you have left** the website!
- **So, how could cookies be exploited?**



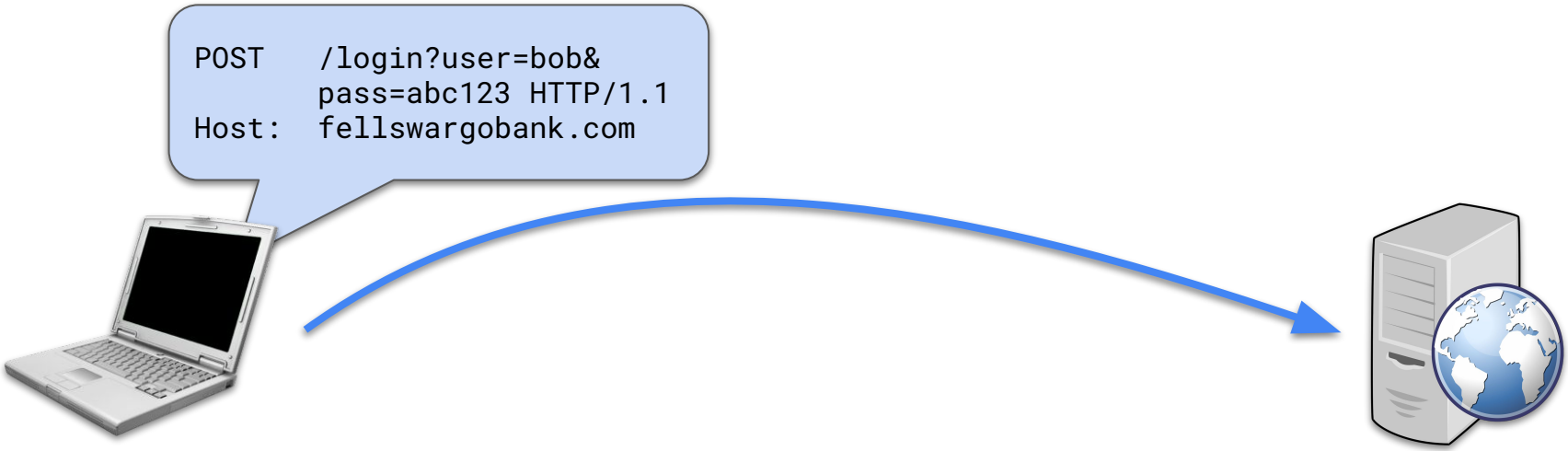
Cookie Chaos

- Cookies enable **persistent** **interaction**
  - Even **after you have left** the website!
- **So, how could cookies be exploited?**
- An **attacker-controlled** **website** gets you to perform an operation on a secure site that you have **a login cookie for… without your approval!**



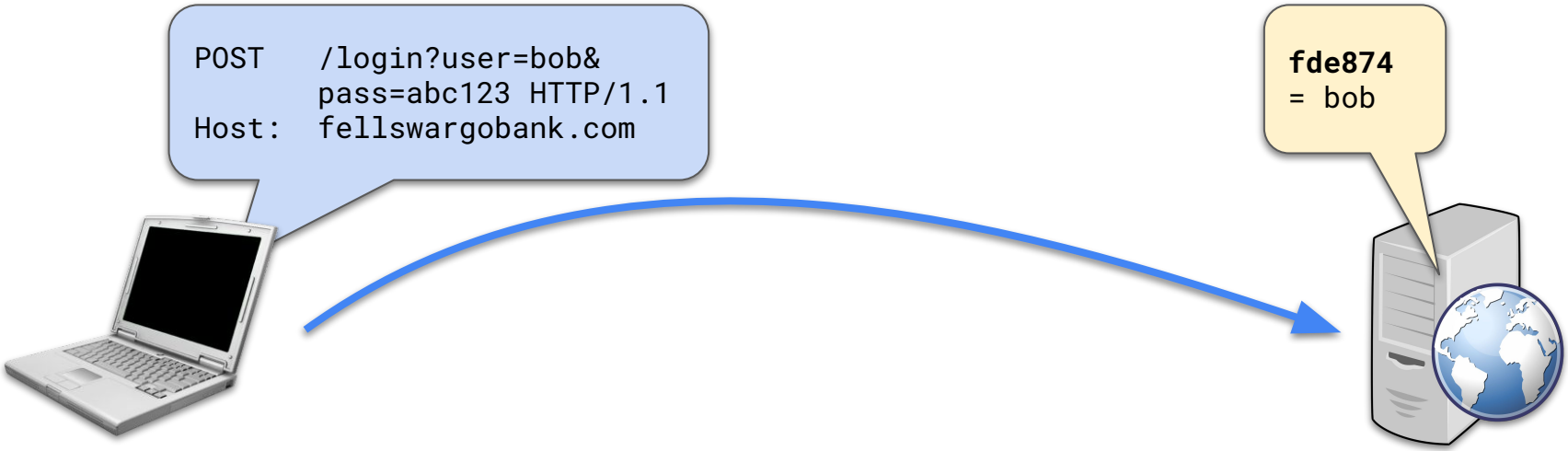
Cross-site Request Forgery (CSRF)

- Suppose you log in to **FellsWargoBank.com**



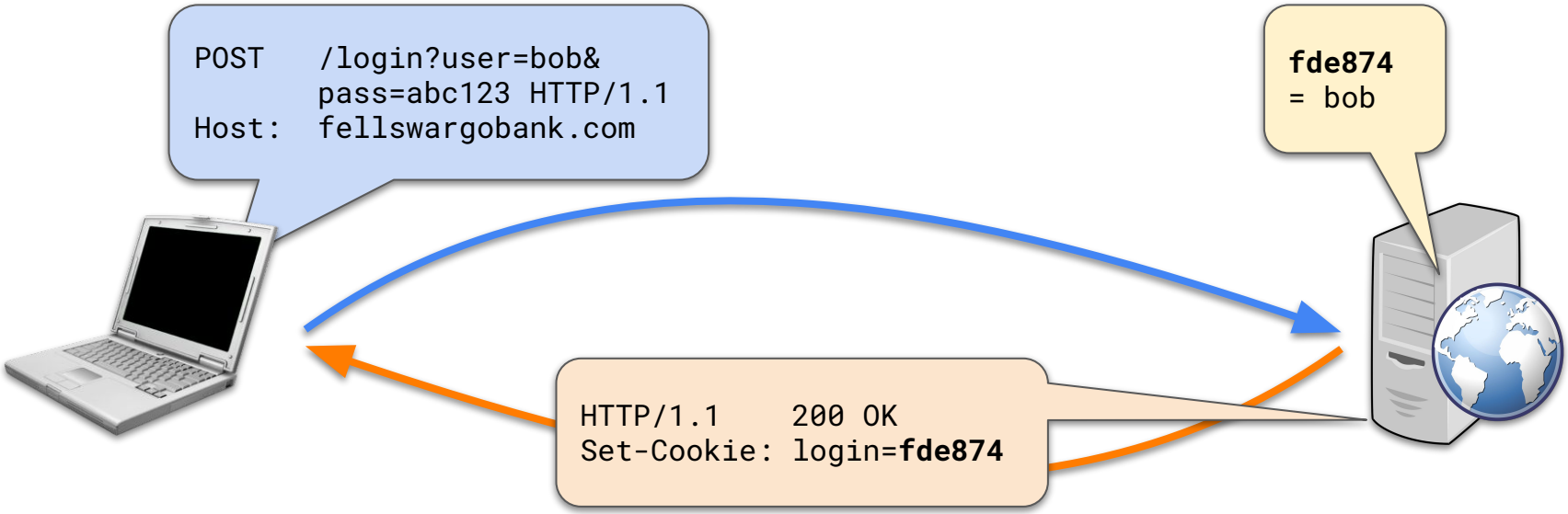
Cross-site Request Forgery (CSRF)

- Suppose you log in to **FellsWargoBank.com**



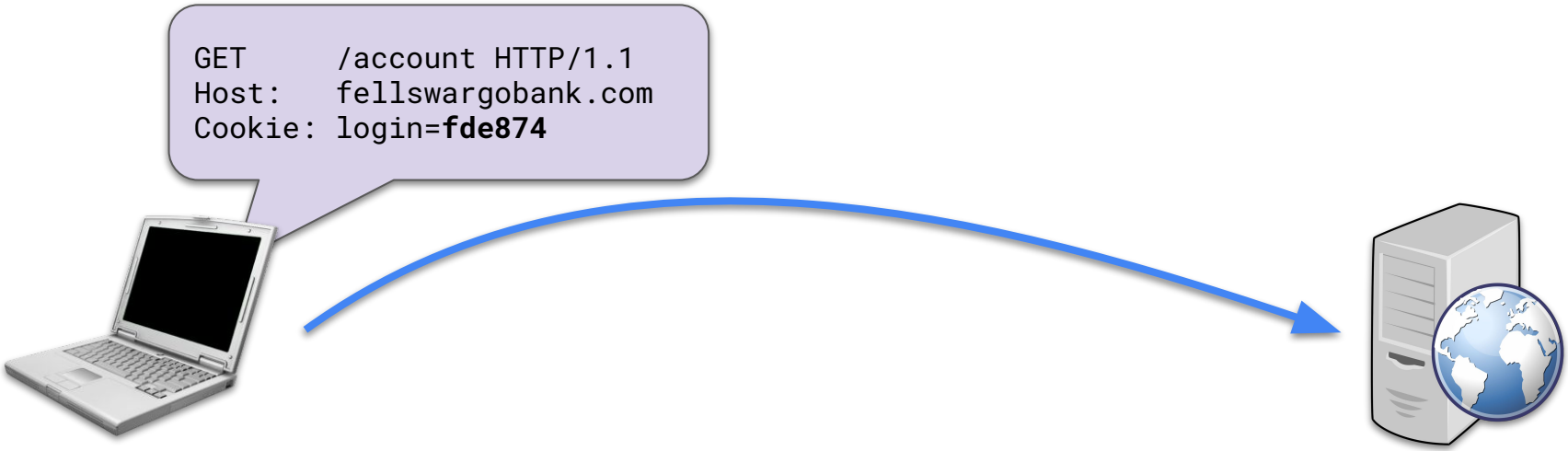
Cross-site Request Forgery (CSRF)

- Suppose you log in to **FellsWargoBank.com**



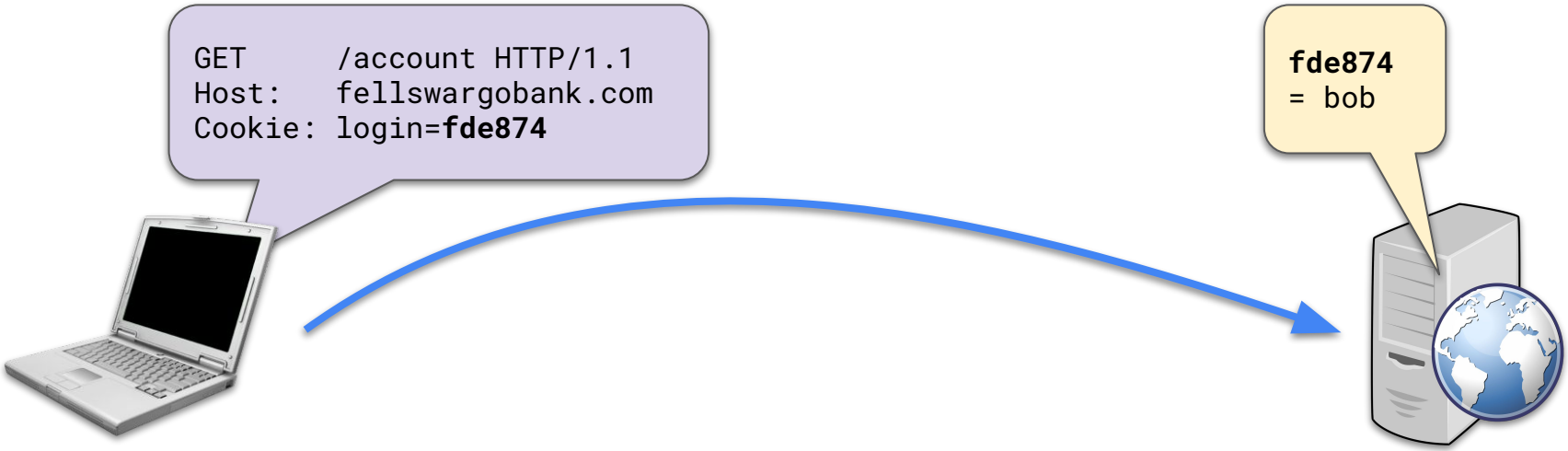
Cross-site Request Forgery (CSRF)

- Suppose you log in to **FellsWargoBank.com**



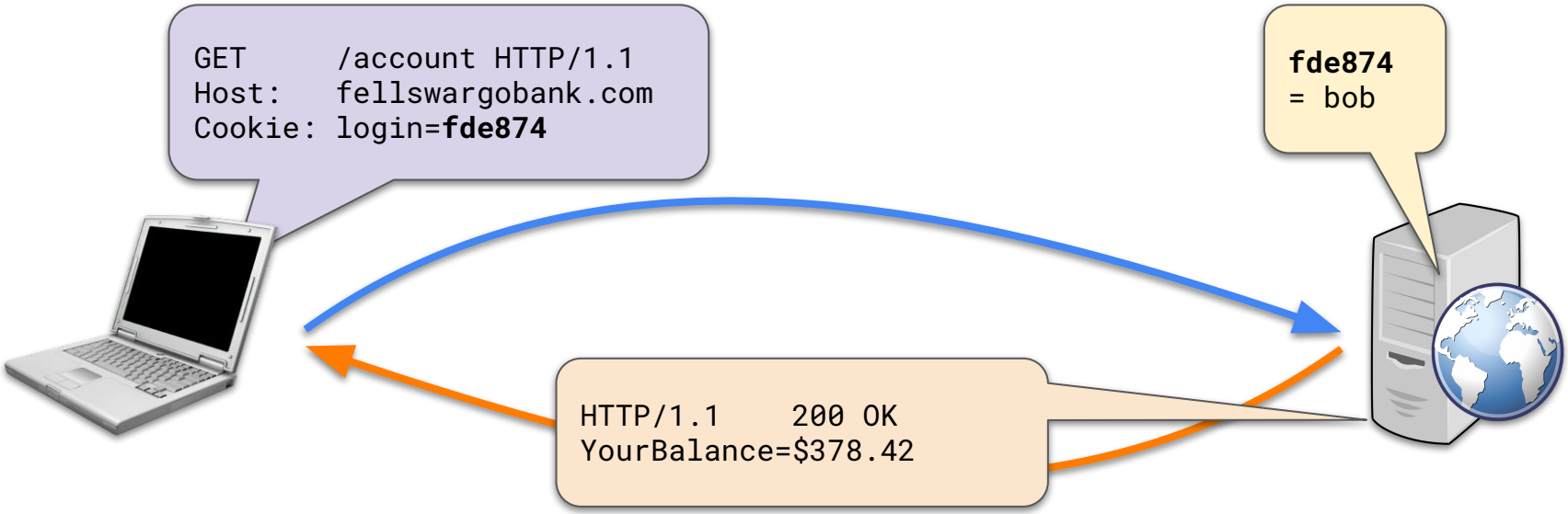
Cross-site Request Forgery (CSRF)

- Suppose you log in to **FellsWargoBank.com**



Cross-site Request Forgery (CSRF)

- Suppose you log in to **FellsWargoBank.com**

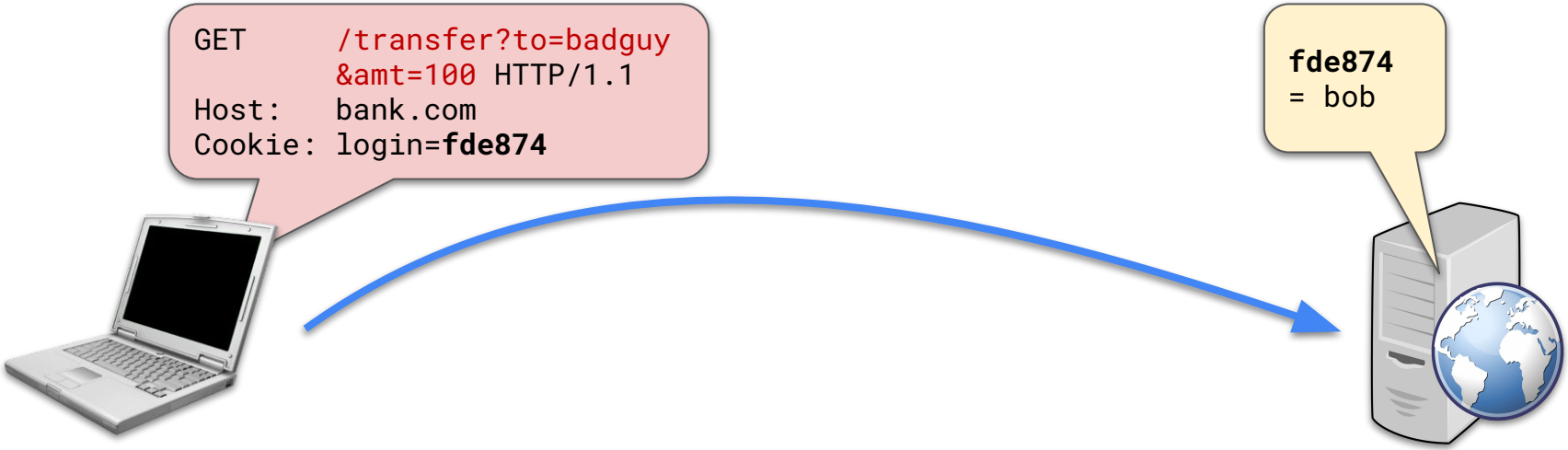


Cross-site Request Forgery (CSRF)

- Then, you **click a sketchy link** from someone that **messaged you on TikTok**...
  - `http://fellswargobank.com/transfer?to=badguy&amt=100`

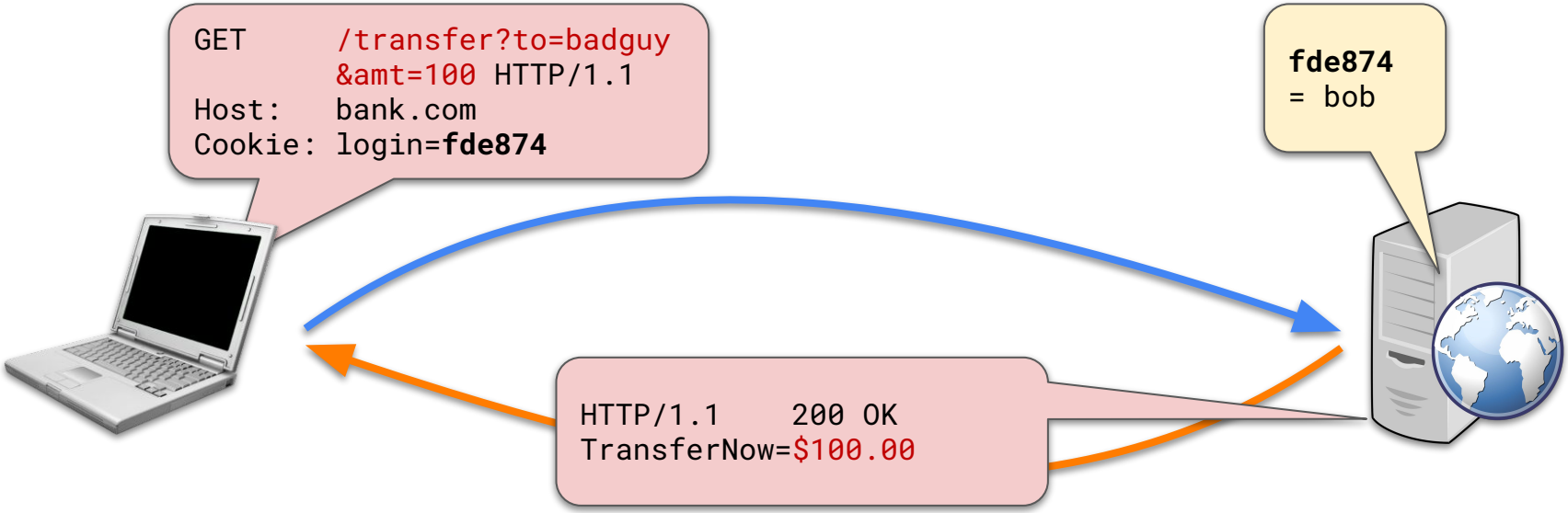
Cross-site Request Forgery (CSRF)

- Then, you **click a sketchy link** from someone that **messaged you on TikTok**...
  - `http://fellswargobank.com/transfer?to=badguy&amt=100`



Cross-site Request Forgery (CSRF)

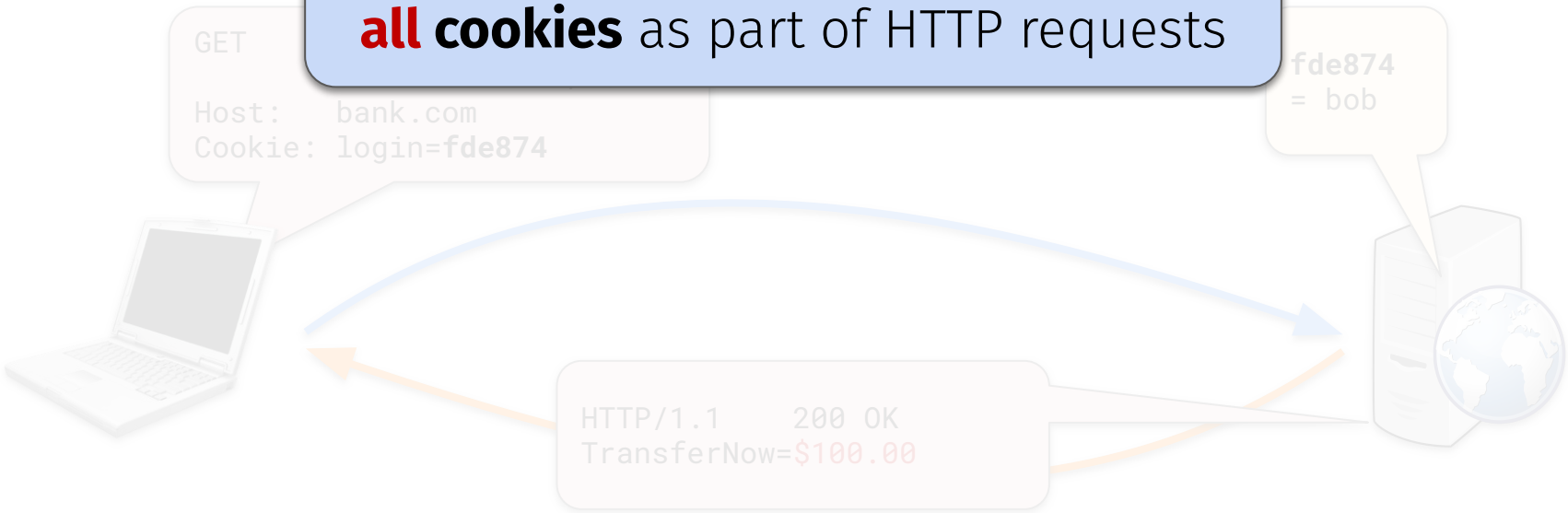
- Then, you **click a sketchy link** from someone that **messaged you on TikTok**...
  - `http://fellswargobank.com/transfer?to=badguy&amt=100`



Cross-site Request Forgery (CSRF)

- Then, you **click a sketchy link** from someone that **messaged you on TikTok**...
  - http://

Browser will **automatically re-send all cookies** as part of HTTP requests

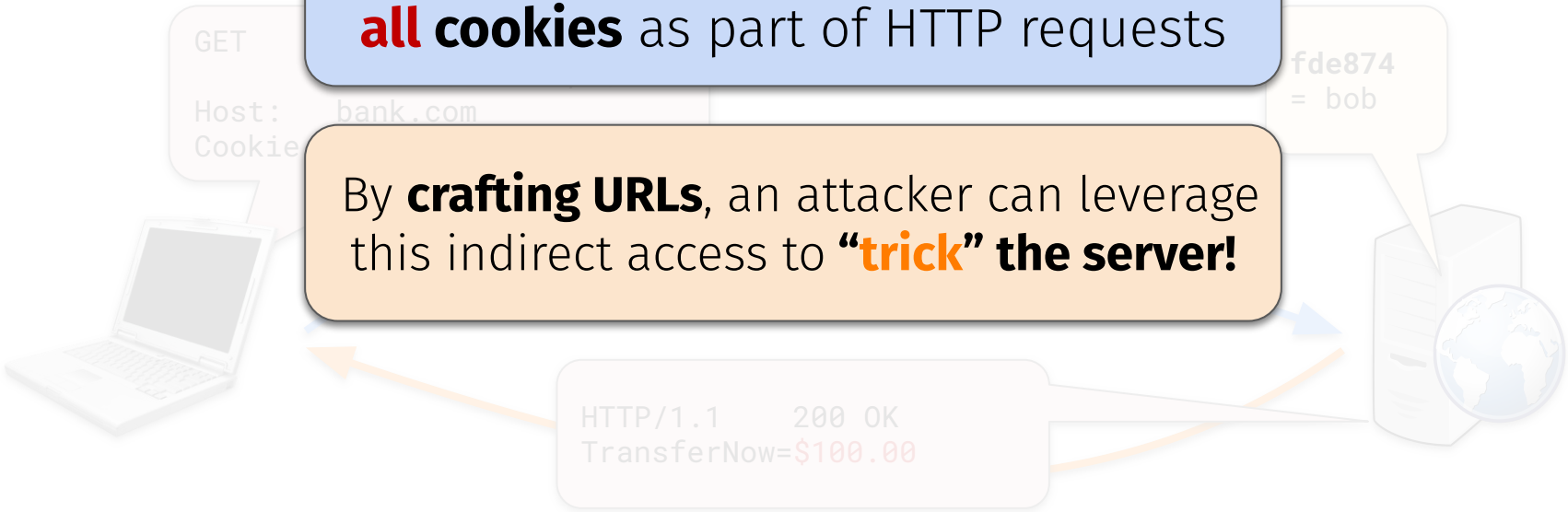


Cross-site Request Forgery (CSRF)

- Then, you **click a sketchy link** from someone that **messaged you on TikTok**...
  - http://

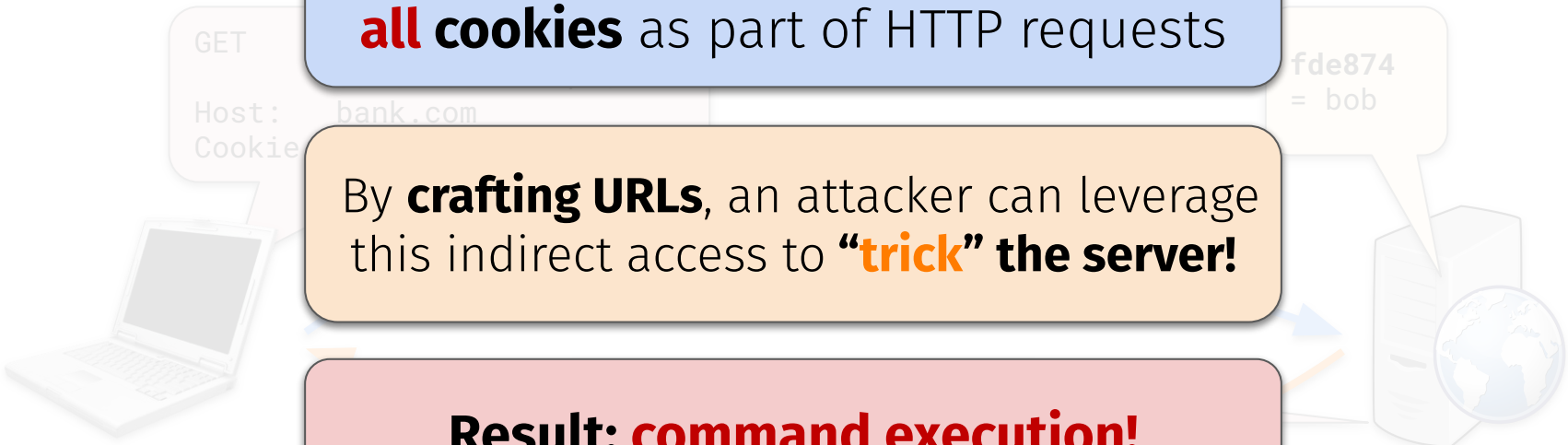
Browser will **automatically re-send all cookies** as part of HTTP requests

By **crafting URLs**, an attacker can leverage this indirect access to **"trick" the server!**



Cross-site Request Forgery (CSRF)

- Then, you **click a sketchy link** from someone that **messaged you on TikTok**...
  - http://



Browser will **automatically re-send all cookies** as part of HTTP requests

By **crafting URLs**, an attacker can leverage this indirect access to **"trick" the server!**

Result: command execution!

Preventing CSRF

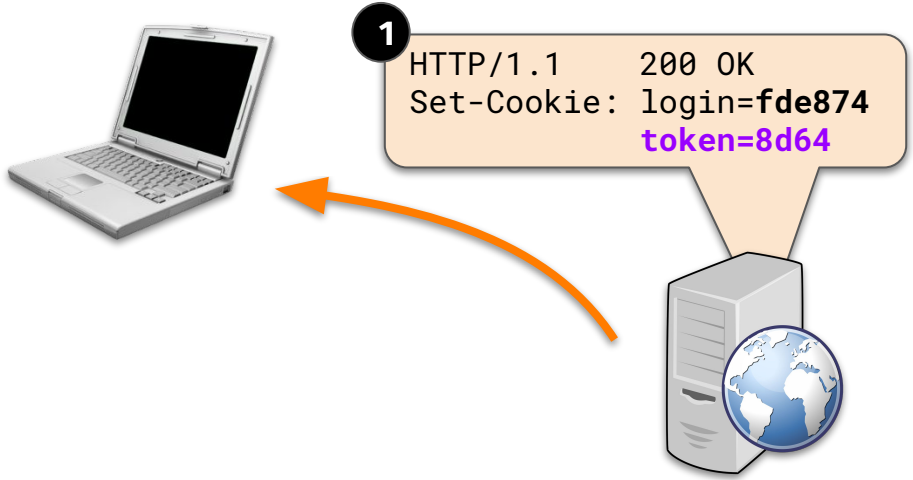
- **Idea:** “authenticate” that user action **originates from our bank website**
  - Called the **Same Origin Policy (SOP)**

- **Fundamental approach:** each “action” gets a **token associated** with it
  - On a new action (page), verify that the associated token is present and correct
  - Token provided in the command must match the token saved in cookie
  - Attacker **can’t find token for another user**, thus **can’t make actions on user’s behalf**

Preventing CSRF

- **Idea:** "authenticate" that user action **originates from our bank website**
  - Called the **Same Origin Policy (SOP)**

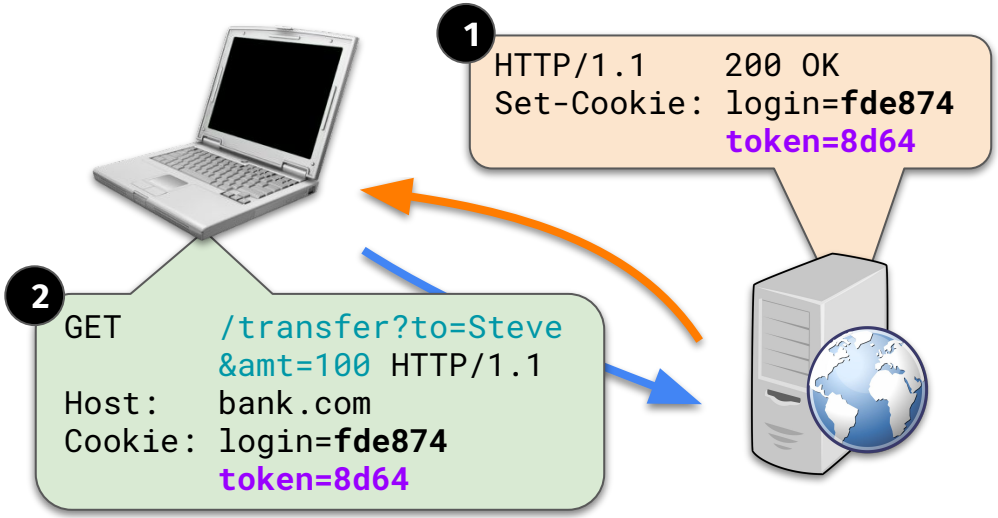
- **Fundamental approach:** each "action" gets a **token associated** with it
  - On a new action (page), verify that the associated token is present and correct
  - Token provided in the command must match the token saved in cookie
  - Attacker **can't find token for another user**, thus **can't make actions on user's behalf**



Preventing CSRF

- **Idea:** "authenticate" that user action **originates from our bank website**
  - Called the **Same Origin Policy (SOP)**

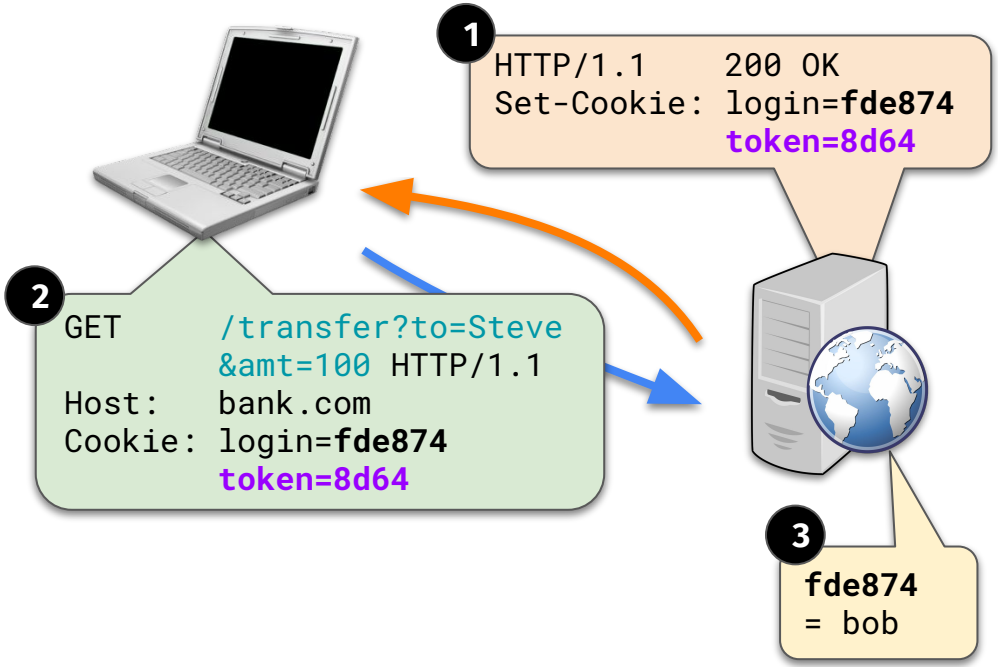
- **Fundamental approach:** each "action" gets a **token associated** with it
  - On a new action (page), verify that the associated token is present and correct
  - Token provided in the command must match the token saved in cookie
  - Attacker **can't find token for another user**, thus **can't make actions on user's behalf**



Preventing CSRF

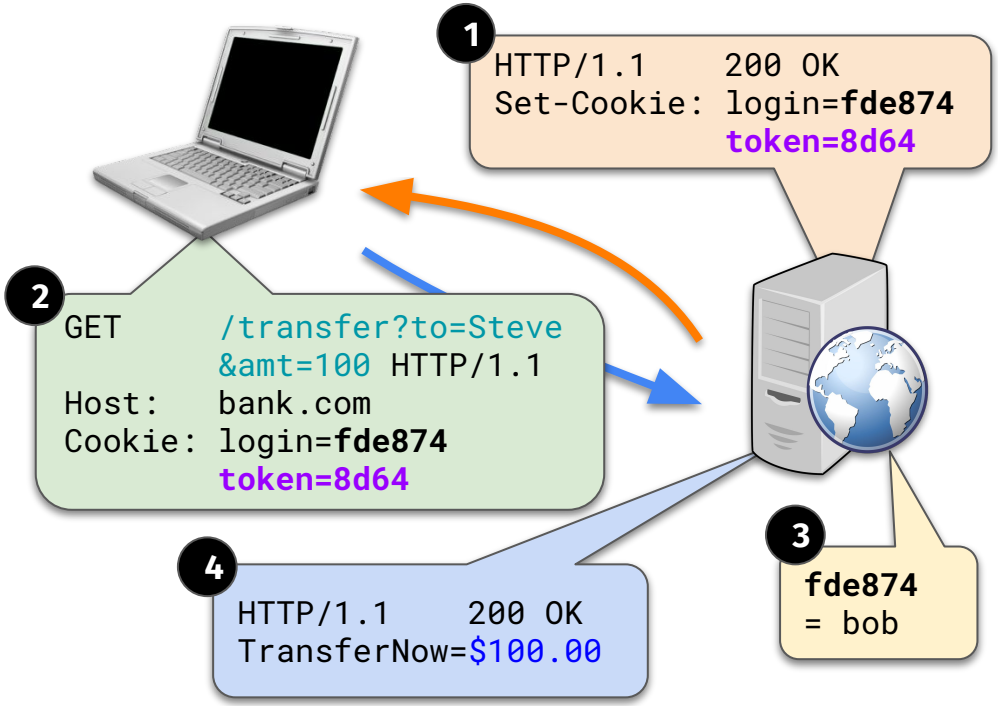
- **Idea:** "authenticate" that user action **originates from our bank website**
  - Called the **Same Origin Policy (SOP)**

- **Fundamental approach:** each "action" gets a **token associated** with it
  - On a new action (page), verify that the associated token is present and correct
  - Token provided in the command must match the token saved in cookie
  - Attacker **can't find token for another user**, thus **can't make actions on user's behalf**



Preventing CSRF

- **Idea:** "authenticate" that user action **originates from our bank website**
  - Called the **Same Origin Policy (SOP)**
- **Fundamental approach:** each "action" gets a **token associated** with it
  - On a new action (page), verify that the associated token is present and correct
  - Token provided in the command must match the token saved in cookie
  - Attacker **can't find token for another user**, thus **can't make actions on user's behalf**



Questions?

Cross-site Scripting (XSS)

Recap: JavaScript

- Rather than static HTML, pages can be expressed dynamically as **programs**
  - Say, one written in **JavaScript**
  - **Transmitted as text, rendered by client's browser**

```
<script type="text/javascript">
    function hello() { alert("Hello world!"); }
</script>
```

```
<img src="picture.gif"
onMouseOver="javascript:hello()">
```

Cross-site Scripting (XSS)

- **Vulnerability:** lack of **input sanitization** on a trusted site

Cross-site Scripting (XSS)

- **Vulnerability:** lack of **input sanitization** on a trusted site

- **Attack:** attacker submits **code as data** to a trusted site
  - Later, the trusted website serves that malicious script to users
  - **Persistent (stored) XSS:** malicious script **injected on vulnerable site** by attacker hosted for a while (e.g., an image, a form post, a malicious advertisement)
  - **Non-persistent (reflected) XSS:** victim **unintentionally sends malicious script** to vulnerable site, and gets malicious resulting page (generated by trusted site)

Cross-site Scripting (XSS)

- **Vulnerability:** lack of **input sanitization** on a trusted site

- **Attack:** attacker submits **code as data** to a trusted site
  - Later, the trusted website serves that malicious script to users
  - **Persistent (stored) XSS:** malicious script **injected on vulnerable site** by attacker hosted for a while (e.g., an image, a form post, a malicious advertisement)
  - **Non-persistent (reflected) XSS:** victim **unintentionally sends malicious script** to vulnerable site, and gets malicious resulting page (generated by trusted site)

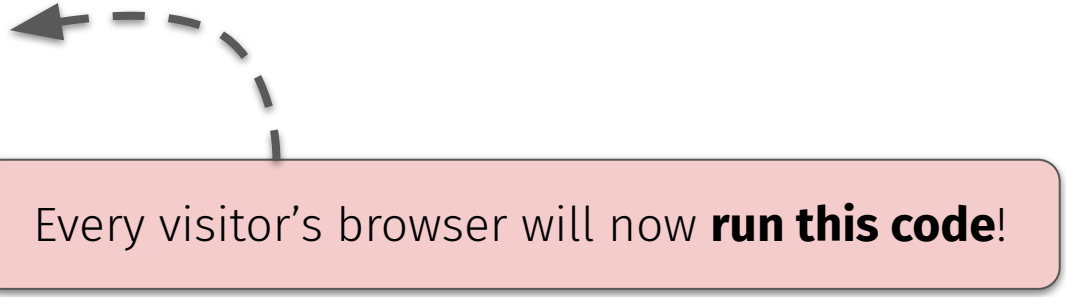
The **attacker's scripts** run **as if they were a part of the trusted site!**

XSS Examples

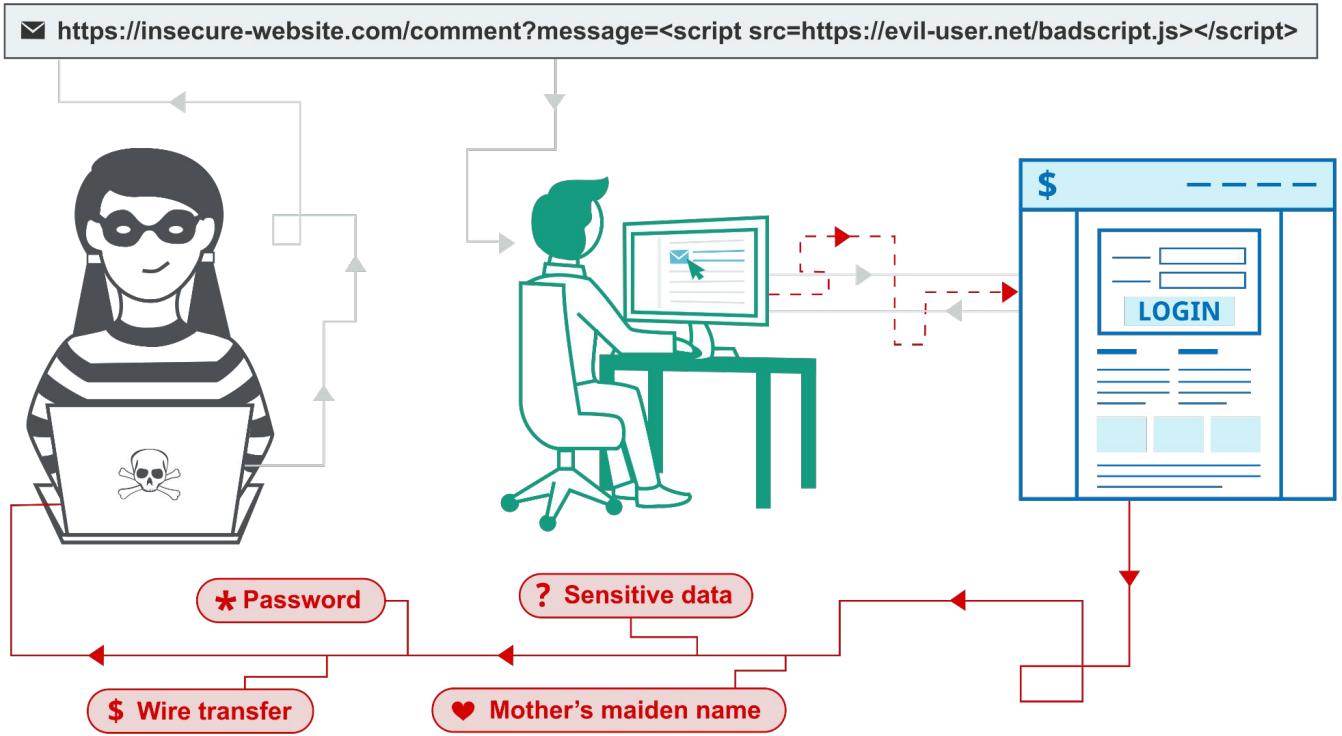
```
<html>
  <title> My guestbook </title>
  <body>
    All you comment belong to me!<br />
    Alice: You make weird references<br />
    Bob: It is supposed to be, “All your base belong to me!”<br />
    …
    Mallory: Never mind :)
      <script>
        alert(“XSS injection”);
      </script><br />
</body>
</html>
```

XSS Examples

```
<html>
  <title> My guestbook </title>
  <body>
    All you comment belong to me!<br />
    Alice: You make weird references<br />
    Bob: It is supposed to be, “All your base belong to me!”<br />
    …
    Mallory: Never mind :)
      <script>
        alert(“XSS injection”);
      </script><br />
</body>
</html>
```



XSS Examples



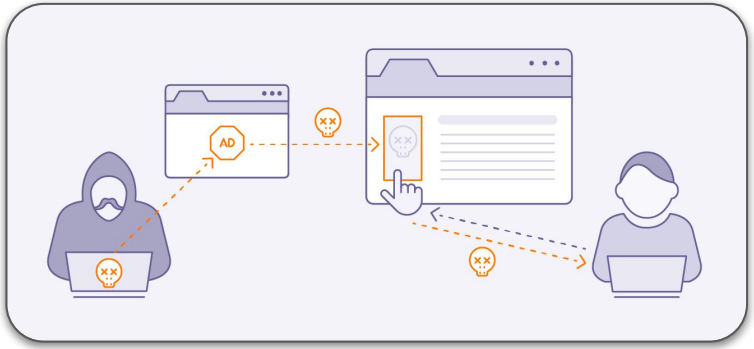
Preventing XSS

- Make sure that **data gets processed as data**, and not erroneously **executed as code**!

- **Escape special characters!**
  - Which ones? Depends how your **$data** is presented
    - Inside an HTML document? `<div>$data</div>`
    - Inside a tag? `<a href="http://site.com/$data">`
    - Inside Javascript code? `var x = "$data";`
  - Make sure to escape every last instance!
  - Many existing frameworks can let you declare what is user-controlled data to automatically perform escaping on!

Summary: types of XSS

- **XSS Goal:** trick browsers into giving **undue access to attacker's JavaScript**
- **Stored XSS:** attacker leaves JavaScript lying around on a benign web service
  - **Victim visites site and browser executes it!**
- **Reflected XSS:** attacker gets user to click on specially crafted URL with script in it
  - **Service then reflects it back to victim's browser!**
- **Heavily used by malvertising campaigns!**

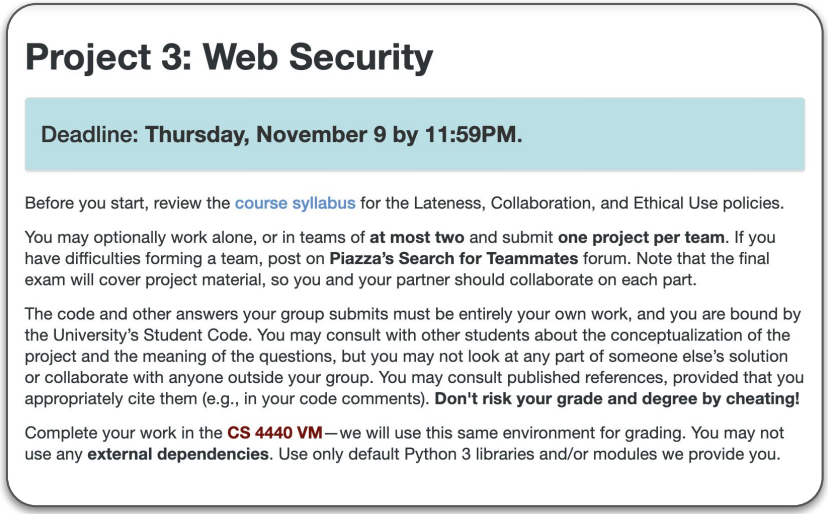


Questions?

Project 3 Tips

Project 3 Overview

- Centered around **web exploitation**
  - Help prepare you to write safer web apps!
- **Part 1:**
  - SQL injection
- **Parts 2–3:**
  - Basic CSRF and XSS attacks
  - Advanced (and realistic) XSS
- Extra credit: **20 points**



Project 3: Web Security

Deadline: Thursday, November 9 by 11:59PM.

Before you start, review the course syllabus for the Lateness, Collaboration, and Ethical Use policies.

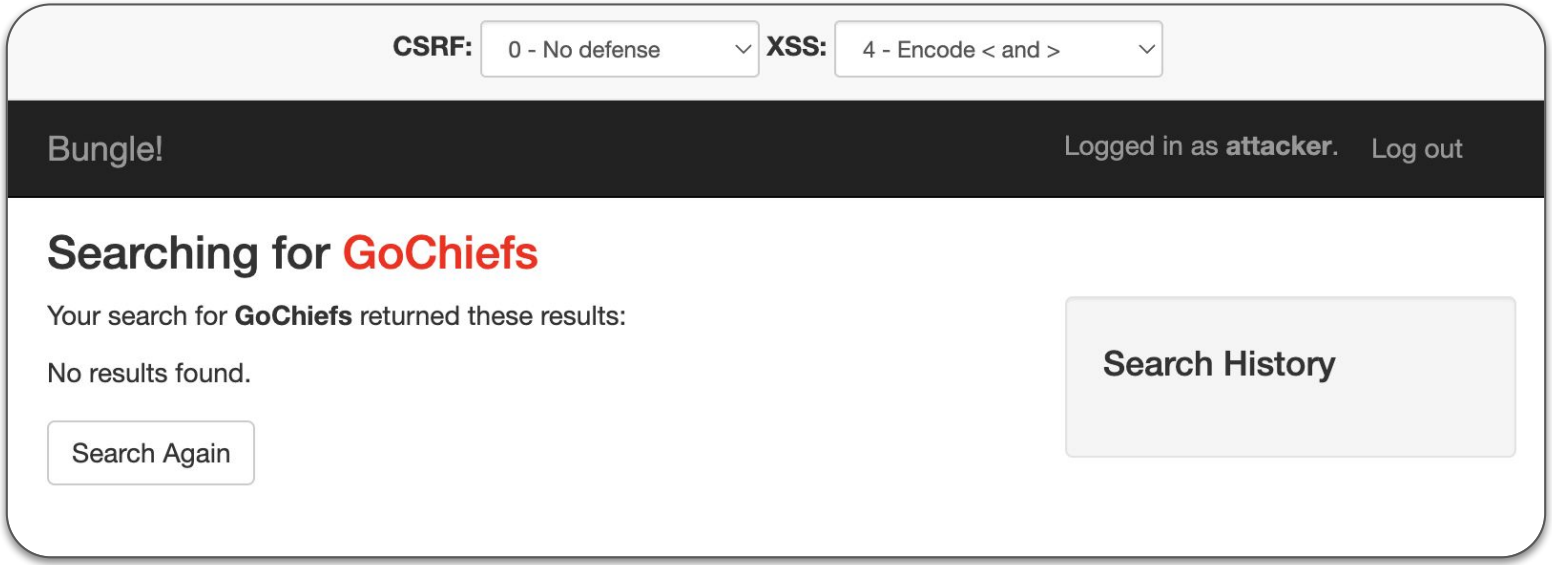
You may optionally work alone, or in teams of **at most two** and submit **one project per team**. If you have difficulties forming a team, post on **Piazza's Search for Teammates** forum. Note that the final exam will cover project material, so you and your partner should collaborate on each part.

The code and other answers your group submits must be entirely your own work, and you are bound by the University's Student Code. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., in your code comments). **Don't risk your grade and degree by cheating!**

Complete your work in the **CS 4440 VM**—we will use this same environment for grading. You may not use any **external dependencies**. Use only default Python 3 libraries and/or modules we provide you.

The BUNGLE Website

- We've created a fictitious search engine website named **BUNGLE**
  - **Your job:** demonstrate attacks to help this startup improve their web security



Tips: SQL Injection

- **Part 1:** how will your input **SQL query** be represented on the **server-side**?
  - Like we did in lecture today, **write-out** the query **before** your attack input

Example: before attacker input

```
SELECT * FROM faculty WHERE name = $name AND ssn = $ssn
```

Tips: SQL Injection

- **Part 1:** how will your input **SQL query** be represented on the **server-side**?
  - Like we did in lecture today, **write-out** the query **before and after** your attack input
  - Similar exercise to stack diagrams in Project 2—what **query state** are you aiming for?

Example: before attacker input

```
SELECT * FROM faculty WHERE name = $name AND ssn = $ssn
```

Example: desired query state

```
SELECT * FROM faculty WHERE name =
  'StefanNagy' -- AND ssn = $ssn;
```

Tips: CSRF and XSS

- **Parts 2–3:** what **interface** are you targeting, and what **request** does it take?
  - Read **BUNGLE**'s documentation! https://cs.utah.edu/~snagy/courses/cs4440/wiki/bungle

Search Results (`/search`)

The search results page accepts `GET` requests and prints the search string, supplied in the `q` query parameter, along with the search results. If the user is logged in, the page also displays the user's recent search history in a sidebar.

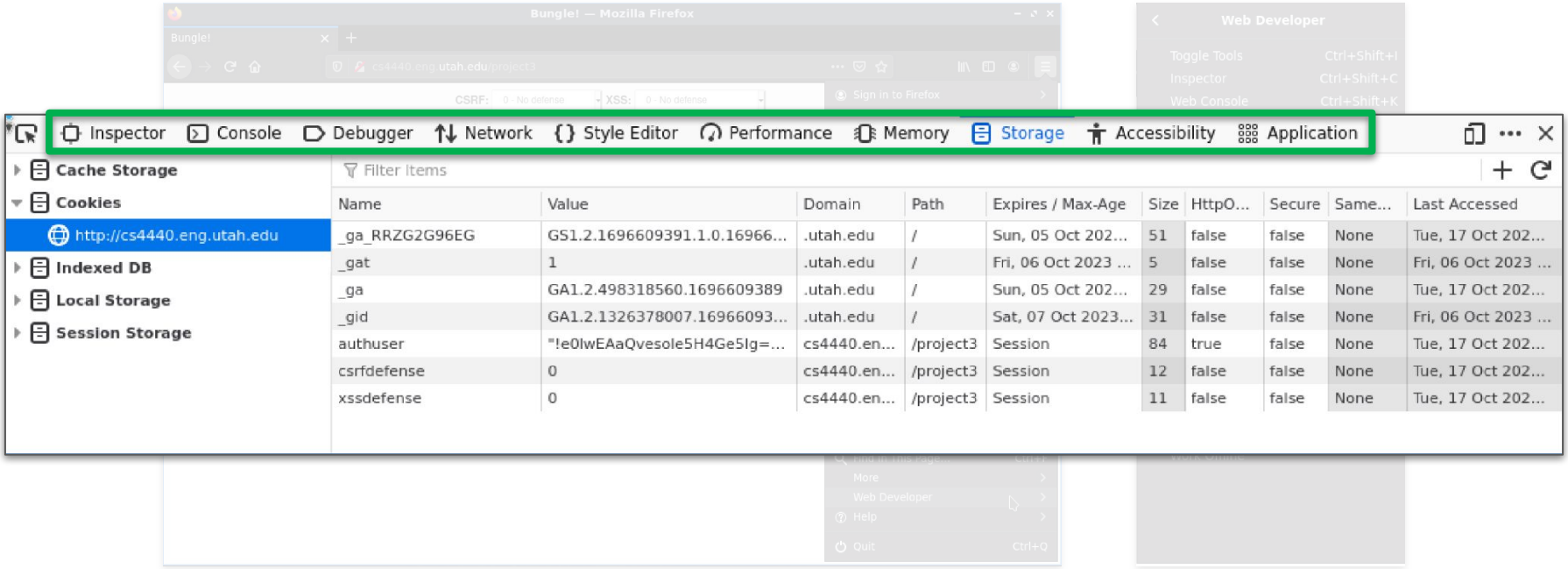
Note: Since actual search is not relevant to this project, you might not receive any results.

Login Handler (`/login`)

The login handler accepts `POST` requests and takes plaintext `username` and `password` query parameters. It checks the user database to see if a user with those credentials exists. If so, it sets a login cookie and redirects the browser to the main page. The cookie tracks which user is logged in; manipulating or forging it is **not** part of this project.

Tips: CSRF and XSS

- **Parts 2–3:** familiarize yourself with the browser's **DOM tree** and **dev tools**



Tips: CSRF and XSS

- **Parts 2–3:** we give you a **skeleton attack template**—you'll fill it out
- **Part 2:** your attacks will be slightly modified versions of this skeleton
- **Part 3:** first craft your attacks atop the template, then try to construct them in their **URL-only** attack form

```
<html>
    <body>
        <!-- Stealthy IFrame (leave here) -->
        <iframe name="BlankPage" style="visibility:hidden;"></iframe>

        <!-- Update any "..." fields accordingly! -->
        <form action="http://cs4440.eng.utah.edu/project3/...?"
              target="BlankPage"
              name="EvilPayload"
              method="...">

            <input name="csrfdefense" value="..." type="...">
            <input name="xssdefense" value="..." type="...">
            <input name="username" value="attacker" type="..."/>
            <input name="password" value="l33th4x" type="..."/>

            <!-----------------------------
            | Your attack code goes here!
            ----------------------------->
        </form>

        <!-- Launch the attack! -->
        <script>
            document.EvilPayload.submit();
        </script>

        <!-- Stealty redirect (leave here) -->
        <meta http-equiv="refresh" content="1; URL=http://cs4440.eng.utah.edu/project3"/>
    </body>
</html>
```

Tips: CSRF and XSS

- Work in a text editor of your choice
  - Construct your attacks step-by-step there
  - Then open and test them within VM's **Firefox**
  - Debug via browser console, alert boxes, etc.
- **Part 2** deliverables are **HTML files**
- **Part 3** deliverables are **URLs**
  - **Suggestion:** master first as HTML files, then convert them to their **URL-only** attack form

Dad why is my sister's name rose?

Because your mother loves roses

Thanks dad

No Problem Vim

Questions?

Next time on CS 4440...

SSL/TLS, certificates, HTTPS attacks and defenses