CSC 405
Session Hijacking

Alexandros Kapravelos
akaprav@ncsu.edu

Cookies

A cookie is an item of data that a web server saves to your computer's hard disk via a web browser

Cookies allow web servers to store and track information about users

Request cookies 6

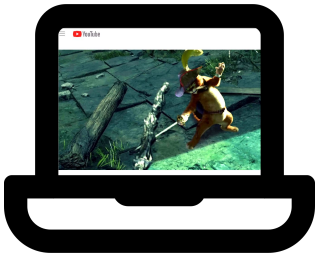
Name	Value
tz	America/New_York
color_mode	{"color_mode":"light","light_theme":{"name":"light","c...
logged_in	yes
dotcom_user	amgaweda
has_recent_activity	1
preferred_color_mode	light

Cookies stored by GitHub.com

Store **almost** any alphanumeric information (under 4KB)

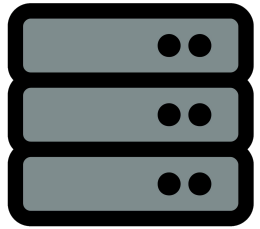
Due to privacy, can **only be read from the issuing domain**

Setting Cookies



Client

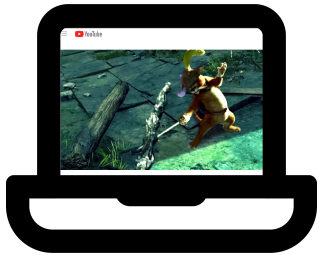
Request



Server

```
GET / HTTP/1.1
Host: www.github.com
User-Agent: Mozilla/5.0 ...
Accept: text/html, ...
```

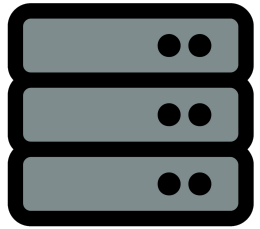
Setting Cookies



Client

Request

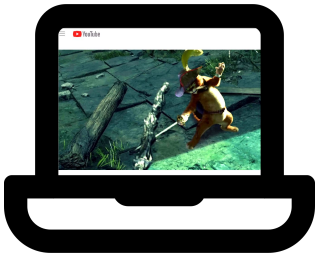
Response



Server

```
HTTP/1.1 200 OK
Server: GitHub.com
Date: Mon, 10 Apr 2023 ...
Content-Type: text/html
Set-Cookie: logged_in=no
...
```

Setting Cookies

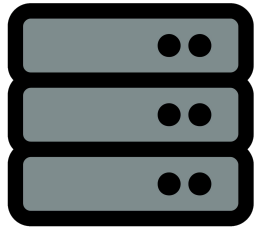


Client

Request

Response

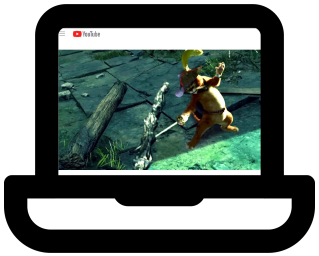
New Request



Server

```
GET /url HTTP/2
Host: www.github.com
Cookie: preferred_color_mode=light;
tz=America%2FNew_York; color_mode=...;
logged_in=yes; dotcom_user=amgaweda;
...
```

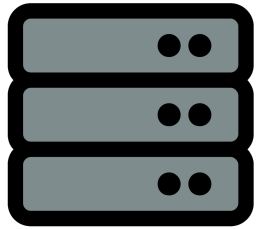
Setting Cookies



Request

Response

New Request



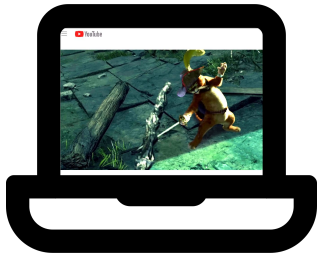
Client

Server

Not malicious, but cookies can leak things like Dr. Gaweda prefers light mode when on GitHub

```
GET /url HTTP/2
Host: www.github.com
Cookie: preferred_color_mode=light;
tz=America%2FNew_York; color_mode=...;
logged_in=yes; dotcom_user=amgaweda;
...
```

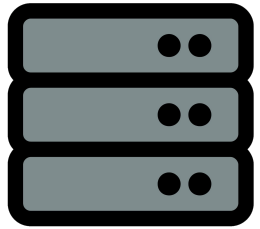
Setting Cookies



Request

Response

New Request



Client

Server

If websites blindly accept cookie data, **then** we've got a vulnerability

```
GET /url HTTP/2
Host: www.github.com
Cookie: preferred_color_mode=light;
tz=America%2FNew_York; color_mode=...;
logged_in=yes; dotcom_user=amgaweda;
...
```

PHP Cookies

```
setcookie(name, value, expire, path, domain, secure, httponly);
```

Parameter	Description	Example
name	Name of the cookie, so the server knows what to grab; like a variable	dotcom_user
value	Value of the cookie	amgaweda
expire	(Optional) The Unix timestamp of the expiration date. Generally, you'd use **time() + some number of seconds**. If not set, cookie expires when the browser closes	time() + 60 * 60 * 24 * 7 (Expire in 1 Week)
path	(Optional) Path of the cookie; if it is **/** it is available over the entire domain	/
domain	(Optional) Internet domain of cookie; if it is **example.com**, it is available across all domains, like **images.example.com** and **www.example.com**	github.com
secure	(Optional) Whether the cookie must use a secure connection (**https://**); true or false	FALSE
httponly	(Optional) Whether the cookie can only be access via HTTP; if TRUE, JavaScript cannot access it	FALSE

PHP Cookies

Since websites do not inherently maintain a "state", cookies allow the server to pass information from one page to another

```php
if (isset($_COOKIE['username'])) {
  $username = $_COOKIE['username'];
}
```

PHP Cookies

Since websites do not inherently maintain a "state", cookies allow the server to pass information from one page to another

```php
if (isset($_COOKIE['username'])) {
  $username = $_COOKIE['username'];
}
```

Where's the vulnerability?

PHP Sessions

Since cookie data is vulnerable to XSS attacks, it may be more appropriate to store the information on the server

Frameworks like PHP manage this by creating a single cookie **PHPSESSID** that contains a numeric value and links to some temporary file

`start_session()`

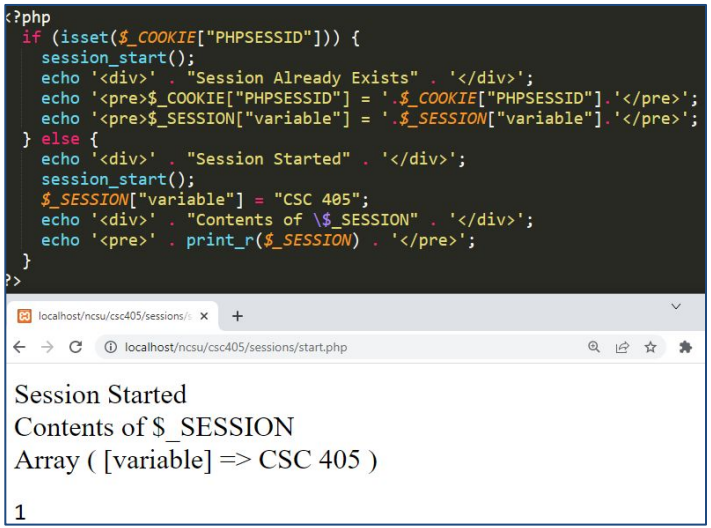
Creates Session on Server

Sets a Cookie `PHPSESSID`

`$_SESSION["key"] = value`

Sets a key-value pair for this session

```php
<?php
 if (isset($_COOKIE["PHPSESSID"])) {
   session_start();
   echo '<div>' . "Session Already Exists" . '</div>';
   echo '<pre>$_COOKIE["PHPSESSID"] = '.$_COOKIE["PHPSESSID"].'</pre>';
   echo '<pre>$_SESSION["variable"] = '.$_SESSION["variable"].'</pre>';
 } else {
   echo '<div>' . "Session Started" . '</div>';
   session_start();
   $_SESSION["variable"] = "CSC 405";
   echo '<div>' . "Contents of \$_SESSION" . '</div>';
   echo '<pre>' . print_r($_SESSION) . '</pre>';
 }
?>
```



PHP Sessions

Since cookie data is vulnerable to XSS attacks, it may be more appropriate to store the information on the server

Frameworks like PHP manage this by creating a single cookie **PHPSESSID** that contains a numeric value and links to some temporary file

Since the **PHPSESSID** is already set when PHP starts the session, the saved variables are still saved until the session cookie expires

```php
<?php
 if (isset($_COOKIE["PHPSESSID"])) {
   session_start();
   echo '<div>' . "Session Already Exists" . '</div>';
   echo '<pre>$_COOKIE["PHPSESSID"] = '.$_COOKIE["PHPSESSID"].'</pre>';
   echo '<pre>$_SESSION["variable"] = '.$_SESSION["variable"].'</pre>';
 } else {
   echo '<div>' . "Session Started" . '</div>';
   session_start();
   $_SESSION["variable"] = "CSC 405";
   echo '<div>' . "Contents of \$_SESSION" . '</div>';
   echo '<pre>' . print_r($_SESSION) . '</pre>';
 }
?>
```

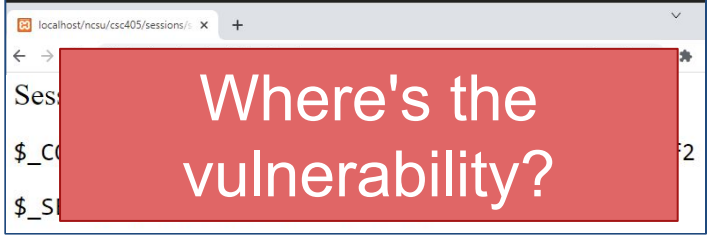
PHP Sessions

Since cookie data is vulnerable to XSS attacks, it may be more appropriate to store the information on the server

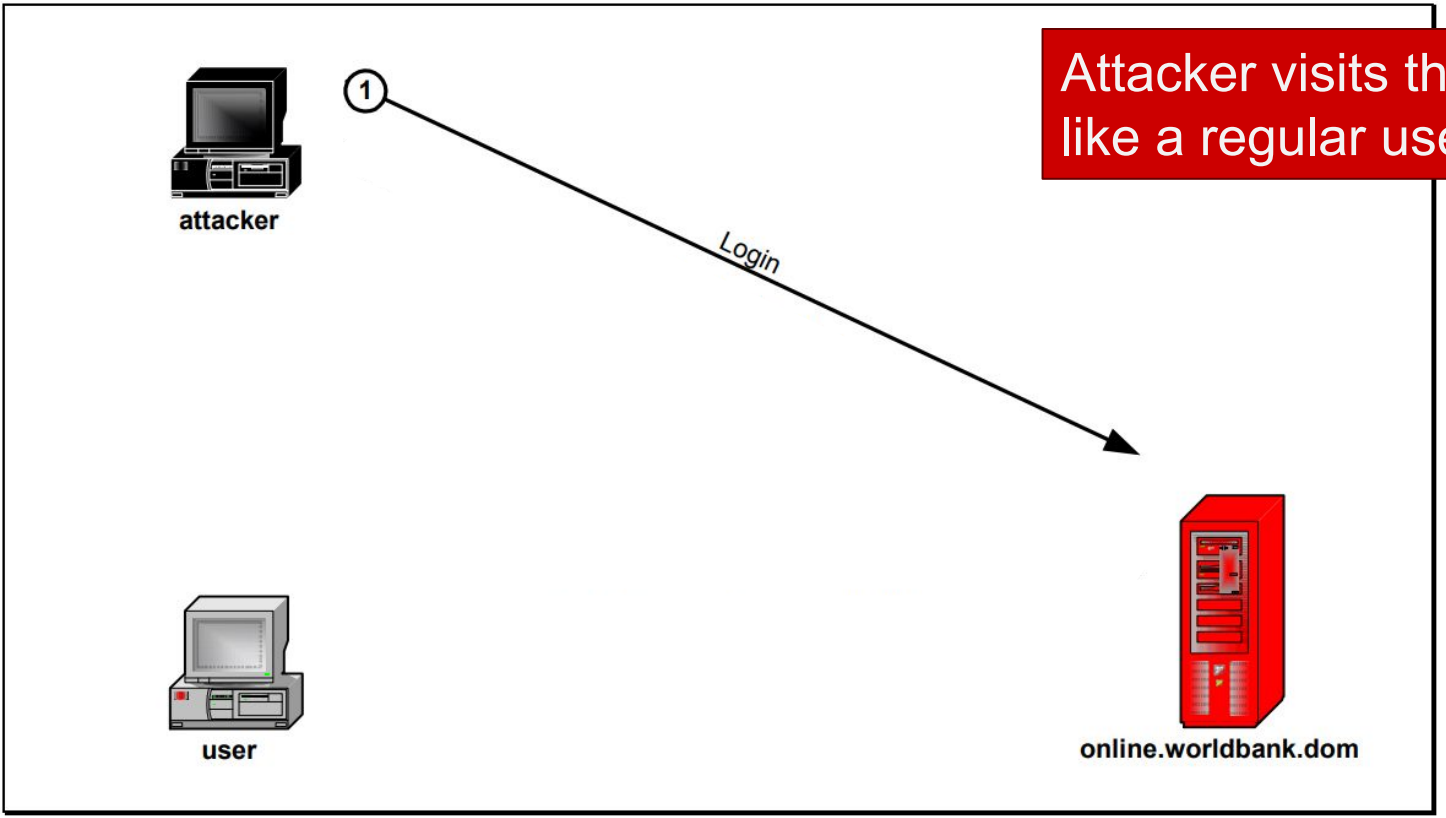
Frameworks like PHP manage this by creating a single cookie **PHPSESSID** that contains a numeric value and links to some temporary file

Since the **PHPSESSID** is already set when PHP starts the session, the saved variables are still saved until the session cookie expires

```php
<?php
 if (isset($_COOKIE["PHPSESSID"])) {
   session_start();
   echo '<div>' . "Session Already Exists" . '</div>';
   echo '<pre>$_COOKIE["PHPSESSID"] = '.$_COOKIE["PHPSESSID"].'</pre>';
   echo '<pre>$_SESSION["variable"] = '.$_SESSION["variable"].'</pre>';
 } else {
   echo '<div>' . "Session Started" . '</div>';
   session_start();
   $_SESSION["variable"] = "CSC 405";
   echo '<div>' . "Contents of \$_SESSION" . '</div>';
   echo '<pre>' . print_r($_SESSION) . '</pre>';
 }
?>
```

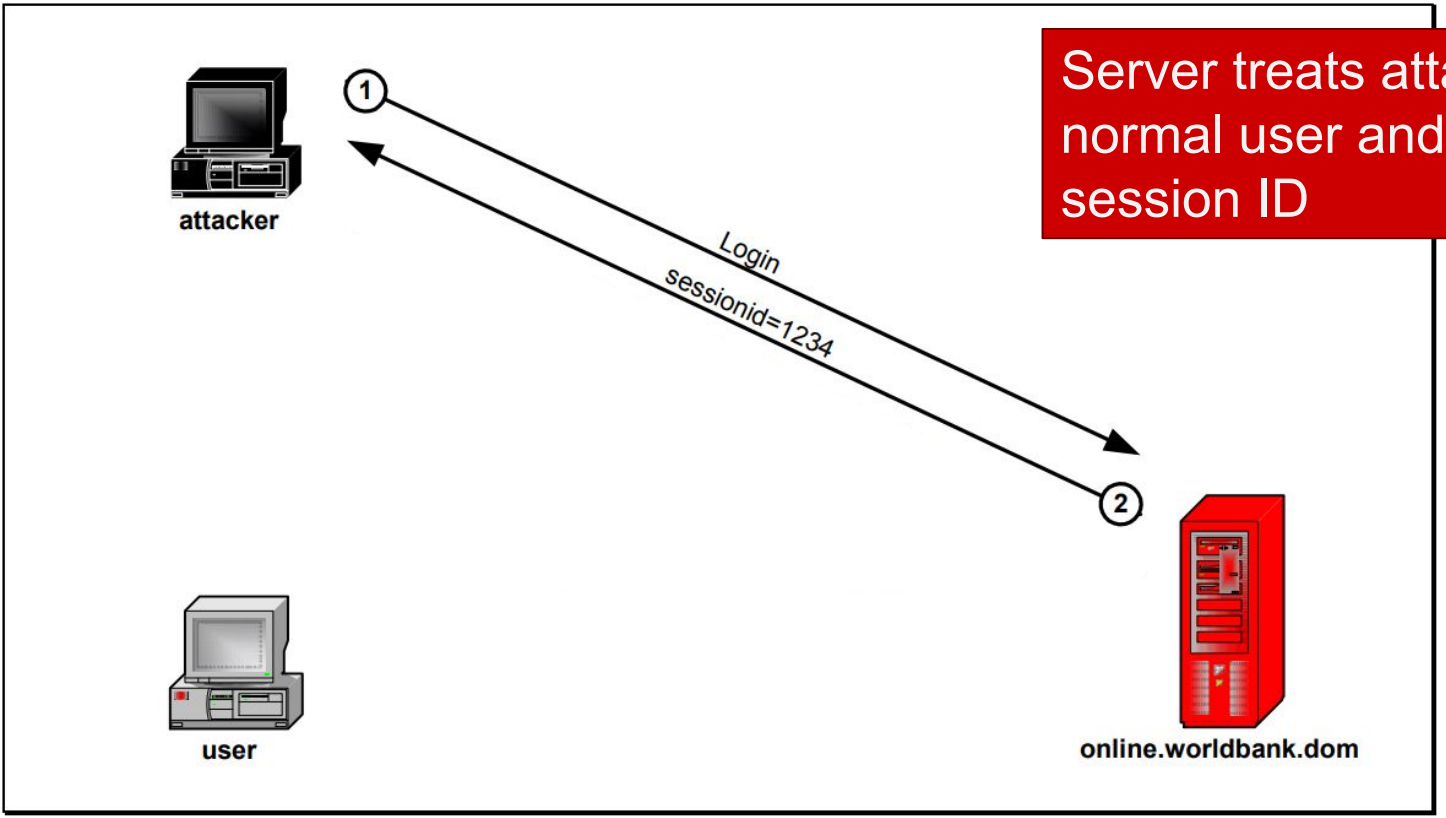


Session Fixation



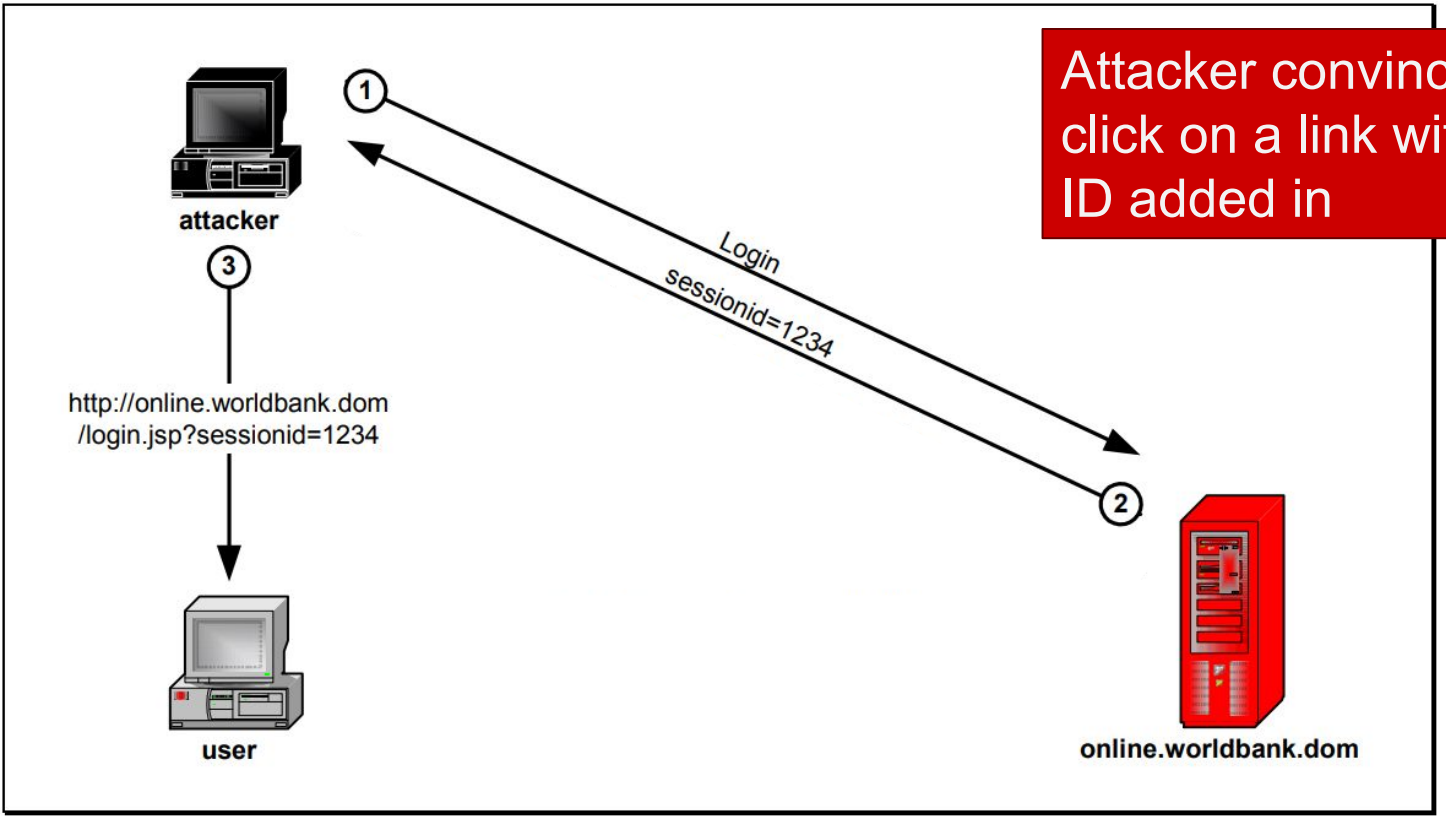
Attacker visits the target location like a regular user

Session Fixation



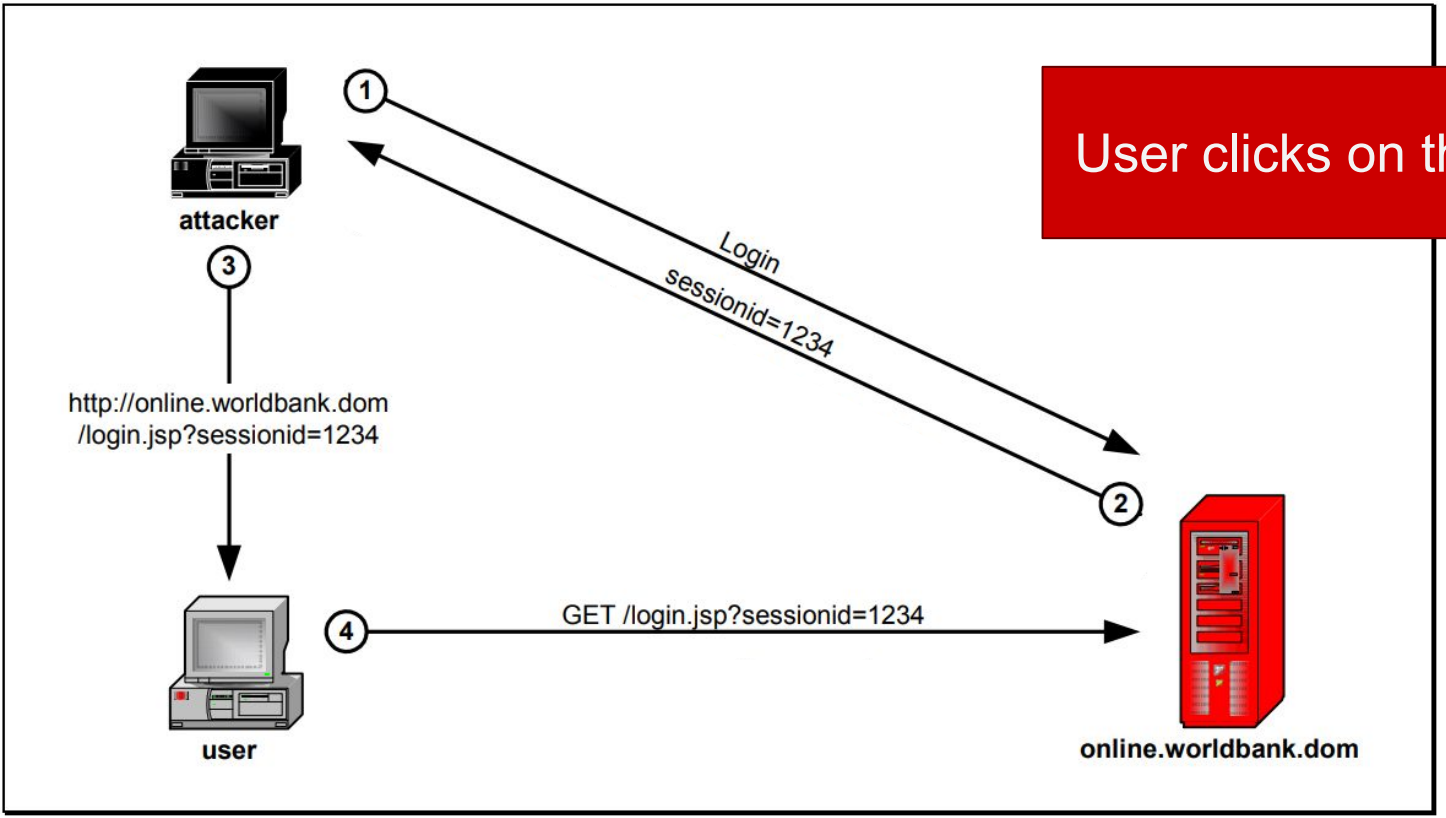
Server treats attacker as a normal user and issues them a session ID

Session Fixation



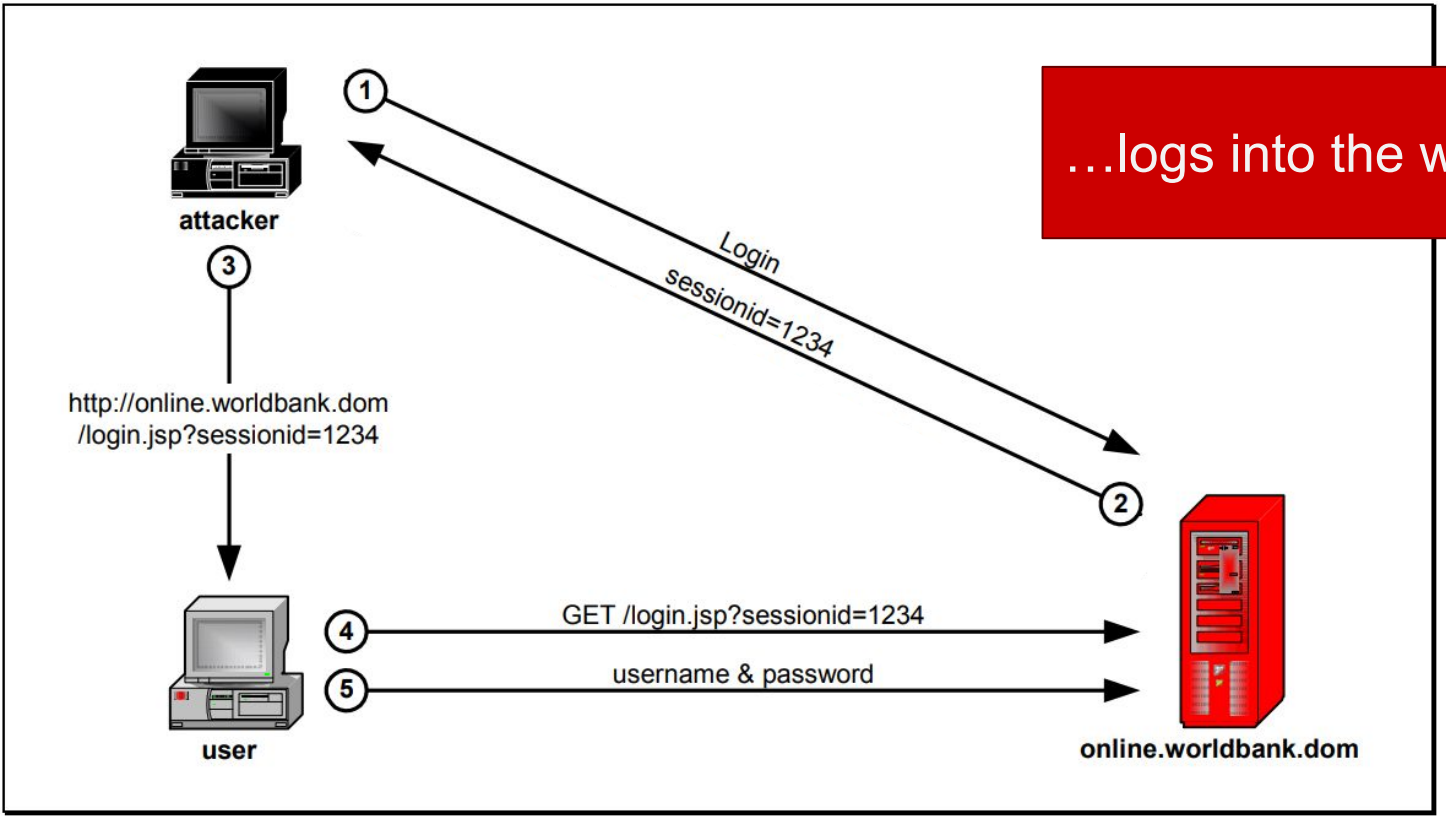
Attacker convinces a user to click on a link with the session ID added in

Session Fixation



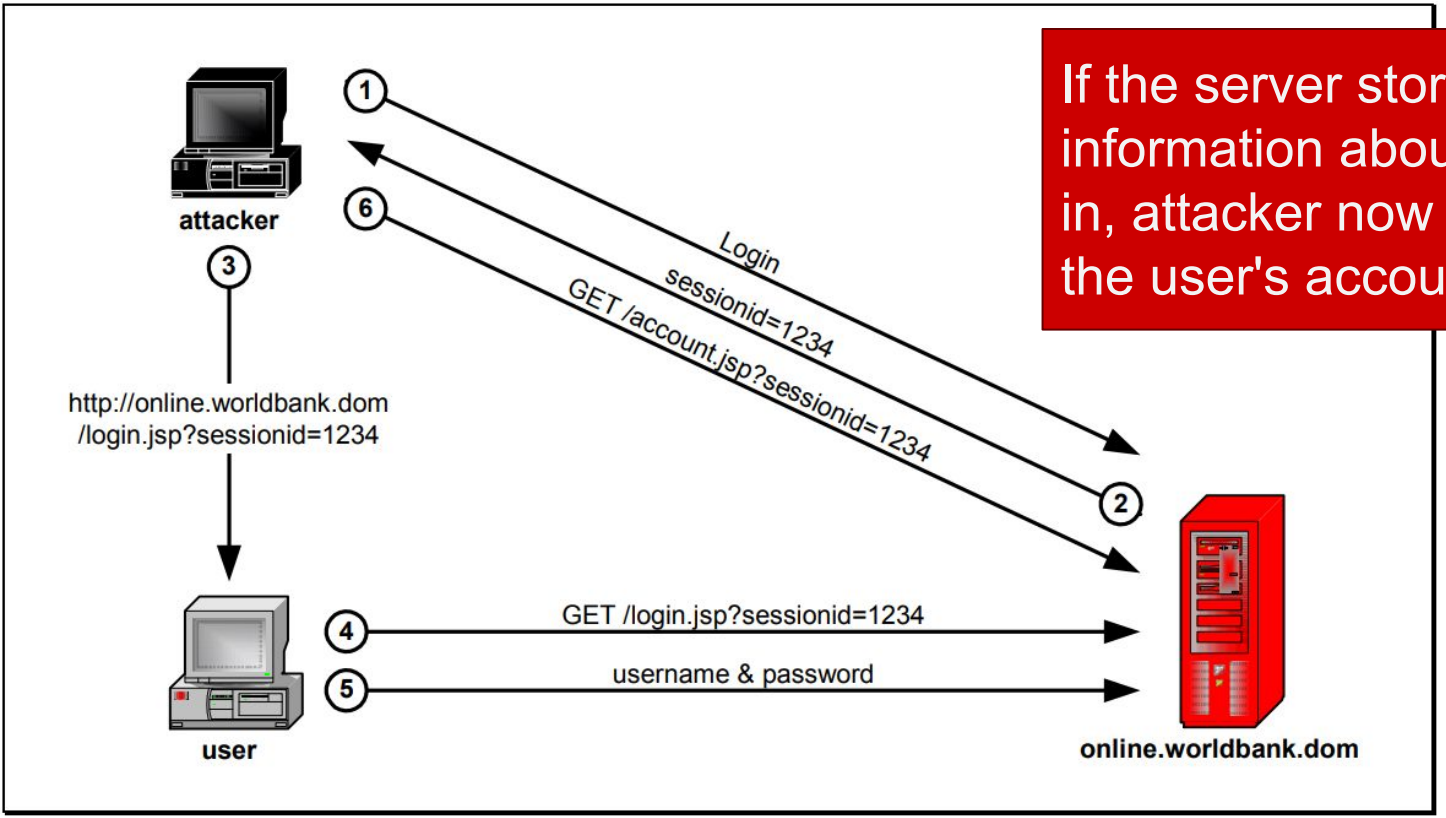
User clicks on the link…

Session Fixation



…logs into the website…

Session Fixation



If the server stores some information about being logged in, attacker now has access to the user's account

Session Fixation

- If the application blindly accepts an existing Session ID, then the initial setup phase is not necessary
- Session IDs should always be regenerated after login and never allowed to be "inherited"
- Session fixation can be compromised with cross-site scripting to achieve Session ID initialization (e.g., by setting the cookie value)
- M. Kolsek, "Session Fixation Vulnerability in Web-based Applications"

Stopping Session Fixations

- Additionally, you can track the address the user originally uses when they log in and check to ensure it is still the same address during use

```
$_SESSION['ip'] = $_SERVER['REMOTE_ADD'];
if ($_SESSION['ip'] != $_SERVER['REMOTE_ADDR'])
  // Man-in-the-Middle Attack
```

Stopping Session Fixations

- Likewise, sessions should be **destroyed** (deleted) as soon as possible

```
$_SESSION = array();
setcookie(
  session_name(),
  '',
  time() - 60 * 60 * 24 * 365,
  '/');
session_destroy();
```

Wipe the **$_SESSION** array

Stopping Session Fixations

- Likewise, sessions should be **destroyed** (deleted) as soon as possible

```
$_SESSION = array();
setcookie(
  session_name(),
  '',
  time() - 60 * 60 * 24 * 365,
  '/');
session_destroy();
```

Wipe the **$_SESSION** array

…and have its cookie expire last year

Stopping Session Fixations

- Likewise, sessions should be **destroyed** (deleted) as soon as possible

```
$_SESSION = array();
setcookie(
  session_name(),
  '',
  time() - 60 * 60 * 24 * 365,
  '/');
session_destroy();
```

Wipe the **$_SESSION** array

…and have its cookie expire last year

…and delete the session tmp file

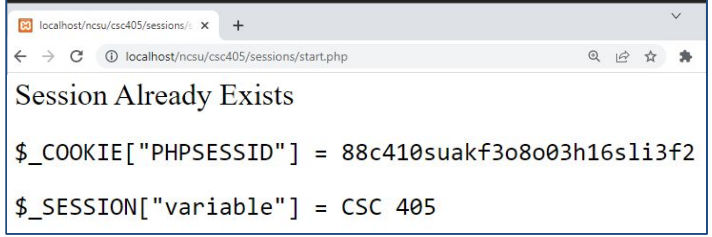
PHP Sessions

Since cookie data is vulnerable to XSS attacks, it may be more appropriate to store the information on the server

Frameworks like PHP manage this by creating a single cookie **PHPSESSID** that contains a numeric value and links to some temporary file

But **this** has to get saved somewhere
...right?

```php
<?php
  if (isset($_COOKIE["PHPSESSID"])) {
    session_start();
    echo '<div>' . "Session Already Exists" . '</div>';
    echo '<pre>$_COOKIE["PHPSESSID"] = '.$_COOKIE["PHPSESSID"].'</pre>';
    echo '<pre>$_SESSION["variable"] = '.$_SESSION["variable"].'</pre>';
  } else {
    echo '<div>' . "Session Started" . '</div>';
    session_start();
    $_SESSION["variable"] = "CSC 405";
    echo '<div>' . "Contents of \$_SESSION" . '</div>';
    echo '<pre>' . print_r($_SESSION) . '</pre>';
  }
?>
```



phpinfo();

Session data gets stored in a **`tmp`** folder on the server, typically specified by PHP or via configuration files

```php
<?php
    phpinfo();
?>
```

PHP 8.2.0 - phpinfo()

localhost/ncsu/csc405/sessions/info.php

Directive	Local Value
session.save_path	C:\xampp\tmp

C > Local Disk (C:) > xampp > tmp

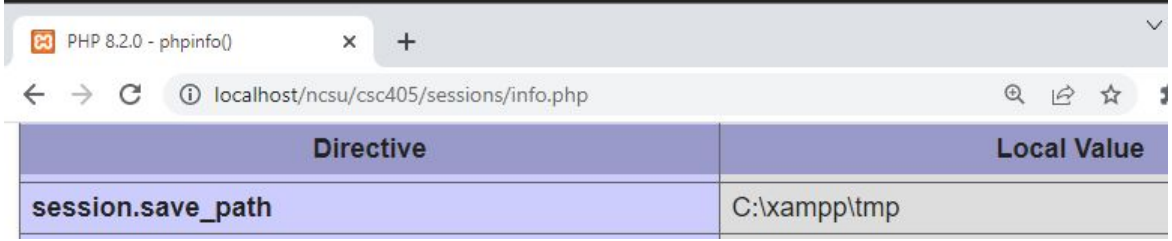
- sess_66nunge52uhuho2pdgceb7950o
- sess_88c410suakf3o8o03h16sli3f2
- sess_e3i7tj66tgqa3l169lpc9qg9jd

The **PHPSESSID** from the last slide

phpinfo();

Session data gets stored in a **`tmp`** folder on the server, typically specified by PHP or via configuration files

```php
<?php
    phpinfo();
?>
```

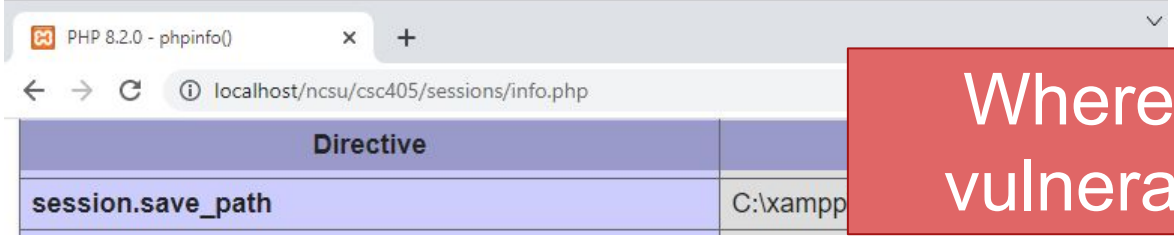


If we look inside, it's just a plaintext file

phpinfo();

Session data gets stored in a **`tmp`** folder on the server, typically specified by PHP or via configuration files

```php
<?php
    phpinfo();
?>
```



Where's the vulnerability?

If we look inside, it's just a plaintext file

Parameter Attacks

- Parameter manipulation
  - The resources accessible are determined by the parameters to a query
  - If client-side information is blindly accepted, one can simply modify the parameter of a legitimate request to access additional information
    - `GET /cgi-bin/profile?userid=1229&type=medical`
    - `GET /cgi-bin/profile?userid=1230&type=medical`

Parameter Attacks

- Parameter manipulation
  - The resources accessible are determined by the parameters to a query
  - If client-side information is blindly accepted, one can simply modify the parameter of a legitimate request to access additional information
    - `GET /cgi-bin/profile?userid=1229&type=medical`
    - `GET /cgi-bin/profile?userid=1230&type=medical`
- Parameter creation
  - If parameters from the URL are imported into the application, can be used to modify the behavior
    - `GET /cgi-bin/profile?userid=1229&type=medical&admin=1`

PHP register_global

- The `register_global` directive makes request information, such as the **`GET`/`POST`** variables and cookie information, available as **global variables**

- Variables can be provided so that particular, unexpected execution paths are followed

Server (Mis)Configuration

- FTP servers and web servers often run on the same host

Server (Mis)Configuration

- FTP servers and web servers often run on the same host
- If data can be uploaded using FTP and then requested using the web server it is possible to
  - Execute programs using CGI (upload to cgi-bin)
  - Execute programs as web application
  - …

Server (Mis)Configuration

- FTP servers and web servers often run on the same host
- If data can be uploaded using FTP and then requested using the web server it is possible to
  - Execute programs using CGI (upload to cgi-bin)
  - Execute programs as web application
  - …
- If a web site allows uploaded files (e.g., images) it might be possible to upload content that is then requested as a code component (e.g., a PHP file)

Server (Mis)Configuration

- Numerous areas where Code and Data are mixed in Web Applications

- Anywhere that strings are concatenated to produce output for another program/parser creates possible problems
  - HTTP
  - HTML
  - SQL
  - Command Line
  - SMTP
  - …

OS Command Injection Attacks

- **Main Issue**: Incorrect (or complete lack of) validation / sanitation of user input that results in the **execution of OS commands** on the server

OS Command Injection Attacks

- **Main Issue**: Incorrect (or complete lack of) validation / sanitation of user input that results in the **execution of OS commands** on the server

- Strings that are passed to a function can evaluate code or include code from a file (language-specific)
  - **`system()`** - run OS commands
  - **`eval()`** - interpret String and execute
  - **`popen()`** - execute a file
  - **`include()`** - load a PHP file
  - **`require()`** - load a PHP file (crash if not found)

OS Command Injection Attacks

- Example: CGI program executes a **`grep`** command over a server file using the user input as parameter
  - Implementation 1:
    ```
    system("grep $exp phonebook.txt");
    ```
    - By providing:
      `foo; echo '1024 35 1386...' > ~/.ssh/authorized_keys; rm`
      one can obtain interactive access and delete the text file

OS Command Injection Attacks

- Example: CGI program executes a **`grep`** command over a server file using the user input as parameter
  - Implementation 1:
    ```
    system("grep $exp phonebook.txt");
    ```
    - By providing:
      `foo; echo '1024 35 1386...' > ~/.ssh/authorized_keys; rm`
      one can obtain interactive access and delete the text file
  - Implementation 2:
    ```
    system("grep \"$exp\" phonebook.txt");
    ```
    - By providing
      `\"foo; echo '1024 35 1386...' > ~/.ssh/authorized_keys; rm \"`
      one can steal the password file and delete the text file

OS Command Injection Attacks

- Example: CGI program executes a **`grep`** command over a server file using the user input as parameter
  - Implementation 1:
    `system("grep $exp phonebook.txt");`
    - By providing:
      `foo; echo '1024 35 1386...' > ~/.ssh/authorized_keys; rm`
      one can obtain interactive access and delete the text file
  - Implementation 2:
    `system("grep \"$exp\" phonebook.txt");`
    - By providing
      `\"foo; echo '1024 35 1386...' > ~/.ssh/authorized_keys; rm \"`
      one can steal the password file and delete the text file
  - Implementation 3:
    `system("grep", "-e", $exp, "phonebook.txt");`
    - In this case the execution is similar to an **`execve()`** and therefore more secure (no shell parsing involved)

Preventing OS Command Injection

- Command injection is a sanitization problem
  - Never trust outside input when composing a command string

- Many languages provide built-in sanitization routines
  - PHP **`escapeshellarg($str)`**: escapes any existing single quotes allowing one to pass a string directly to a shell function and having it be treated as a single safe argument
  - PHP **`escapeshellcmd($str)`**: escapes any characters in a string that might be used to trick a shell command into executing arbitrary commands **`(#&;`|*?~<>^()[]{}$\, \x0A`** and **`\xFF.`** **`'`** and **`"`** are escaped only if they are not paired)