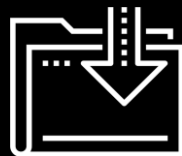













Exploring Servers

Cybersecurity
Web Development Day 2



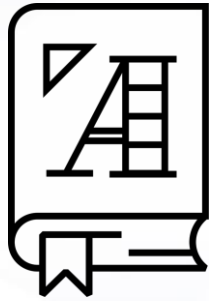
Today's Objectives

By the end of class, you will be able to:

-  Define caching and distinguish between private and public caches.
 -  Explain the process behind web cache poisoning
 -  Explain how cookies and sessions provide ways for servers to remember specific
 -  information about the client.
 -  Describe two common session attacks: Impersonation and forgery.
 -  Distinguish between client-side and server-side validation.
 -  Explain how session hijacking is an impersonation attack on server sessions.
 -  Discuss why servers often store sensitive, larger, and long-term data in sessions instead of cookies.
 -  Inspect weak session IDs generated by a live web app.
-



Cache



Caching is the process of browsers and servers saving and re-using responses in order to reduce load time.

Private and Public Caches

There are two types of Caches: Private and Public



Private Cache: Local cache that serves a single, dedicated user.

- While not as scalable as public caches, they offer immense benefits for single users.



Public Cache: Distributed cache that serves popular files to several users.

- Often sit between you and the destination server you're requesting resources from.
-

Private and Public Caches

When it is best to use Private, Public or No Cache.

Private Cache

- Response with cookies
- Resources provided by HTTPS protocol
- Certain resources only available for one specific user or authorized users.
- Only used by one client

Public Cache

- Popular demand (requested often)
- Not changed often

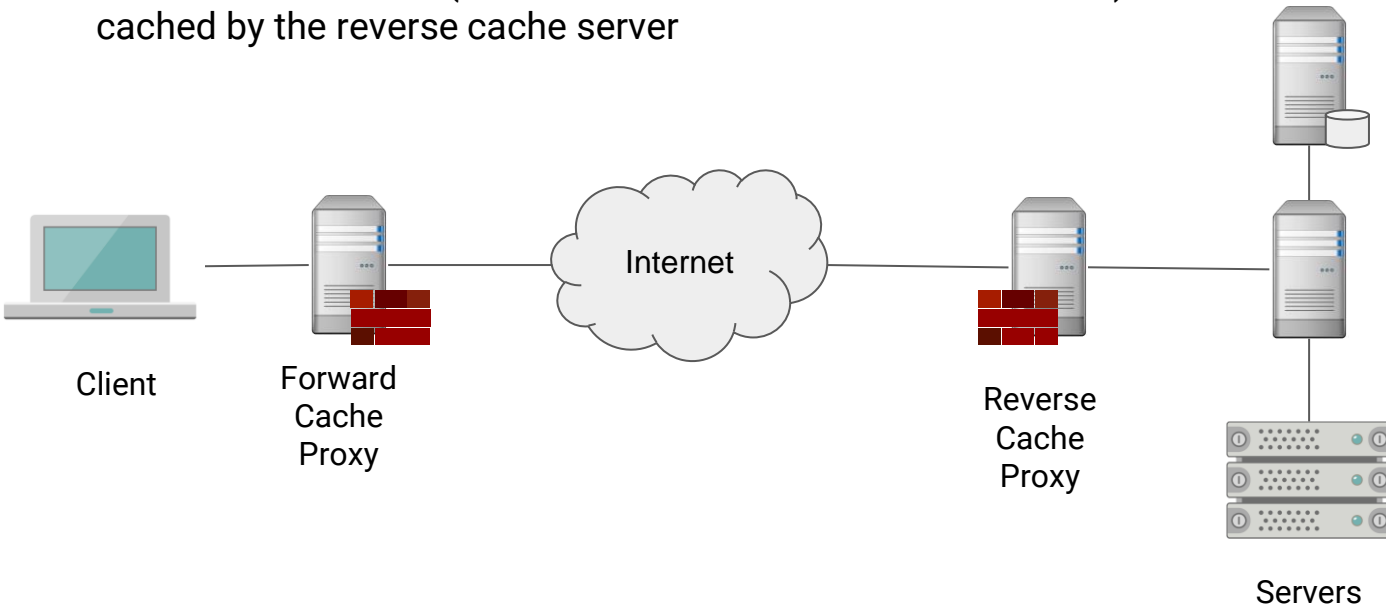
No cache

- Frequently changed objects
 - Time sensitive info (POST request, dynamic info)
-

Forward and Reverse Server Proxy Caching

In **forward cache proxy**, requests coming from the client machine (left) that go out to the internet are cached by the forward proxy cache server

In **reverse cache proxy**, requests coming in to the servers (right) from users on the internet (not on the same network as the servers) are cached by the reverse cache server



In this diagram, the client machine is requesting a web page hosted on the web server.

The client's forward cache proxy will go out to the internet on behalf of the client and retrieve the web page.

As the request comes into the network, the reverse proxy will cache the request after it provides the request to the forward cache proxy.

Now both the forward and reverse cache proxy servers will have the request cached for future use.

Commonly Caches Status Codes

Caches commonly store responses to the status codes.



200 OK: responses are cached because resources don't (or, shouldn't) move frequently. If the resource lived at a given URL in the past, it probably lives there now.



404 NOT FOUND: responses are cached because resources that don't exist at an endpoint are unlikely to suddenly appear.



301 MOVED PERMANENTLY responses indicate that the URL used to request the resource is “invalid”, in the sense that the resource now lives at a different link, and will never “move back” to the URL used in the request.



206 PARTIAL CONTENT: responses are cached under the same logic as 200 OK responses.

Cache-Control

Cache-control is an HTTP header used to specify browser caching policies in both client requests and server responses. Directives include:



Max-Age defines in seconds the mount of time it takes for a cached copy of a resource to expire.



No-Cache indicated that a browser may cache a response, but first must submit a validation request to an origin server.



No-Store: Browsers aren't allowed to cache a response and instead must pull it from the server each time it's requested.



Public: The public response directive indicated that a resource can be cached by any cache.

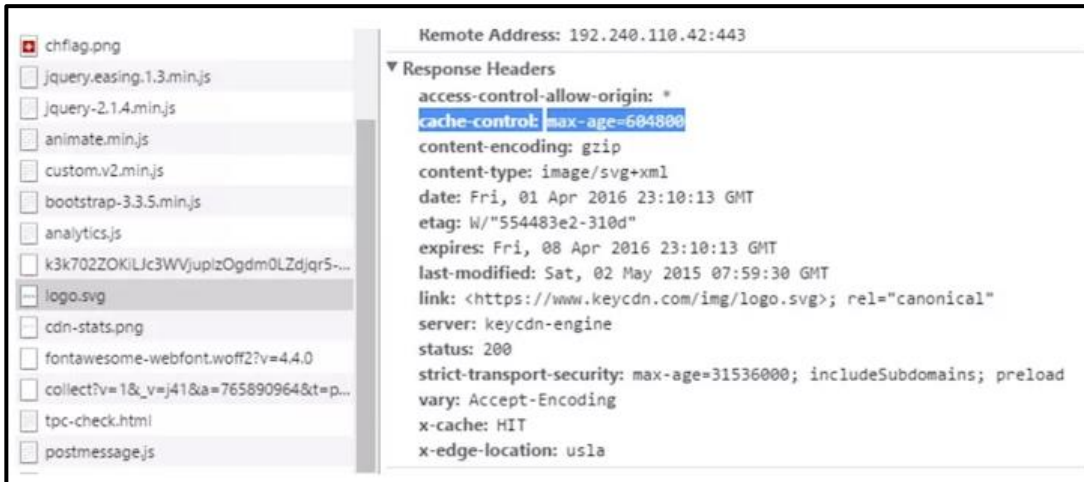


Private: The private response directive indicates that a resource is user specific, it can still be cached out, but only on a client device.

Cache Control

Servers can use the Cache-Control header to instruct clients and intermediary servers as to whether they should cache the response it appears in, and if so, how.

Cache-control directives determine who caches the response, under what conditions and for how long.



Cache Control

The Cache-Control header lets servers specify which resources to cache.

```
HTTP/1.1 200 OK
Date:
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.21
Cache-Control: no-cache
Set-Cookie: SESSID=8toks; httponly
Content-Encoding: gzip
Content-Length: 698
function getStats(event) {
    ...
```

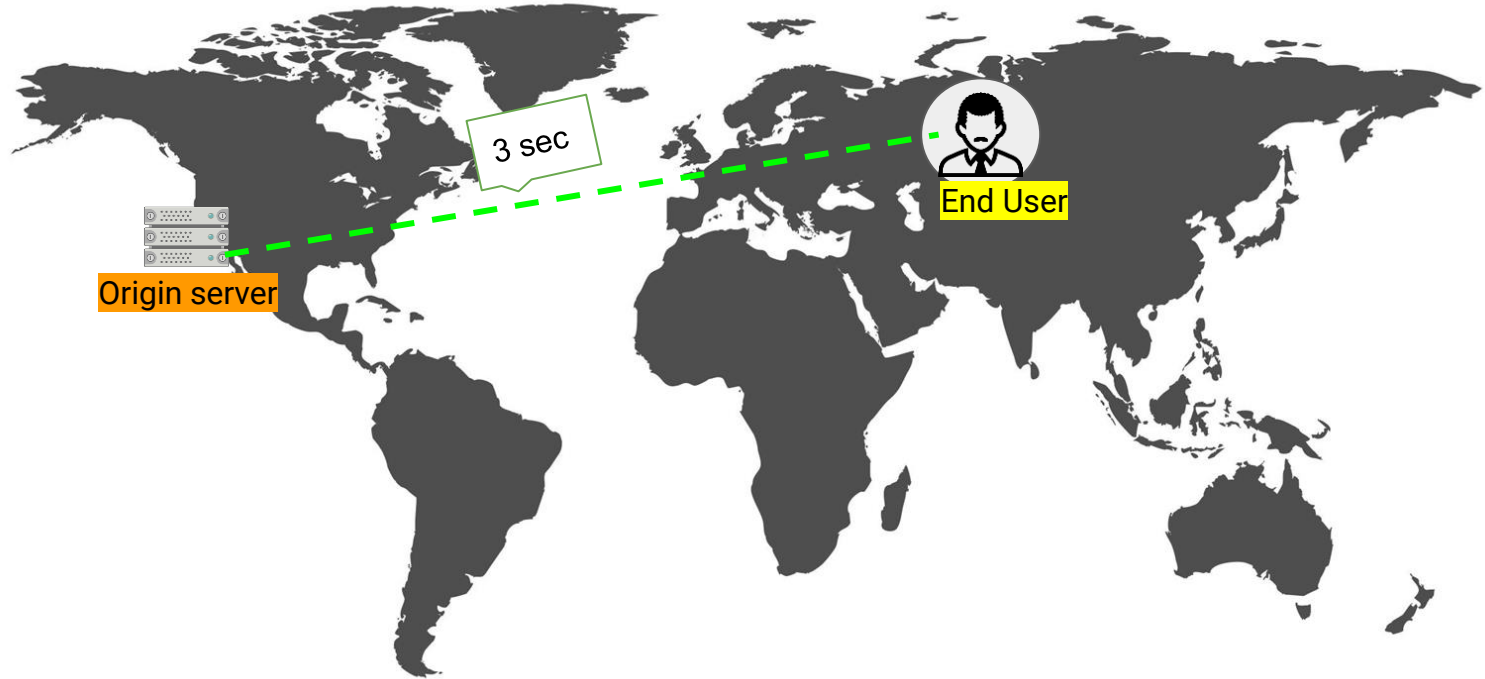
Content Delivery Networks

The variety of caching headers can make manual cache management overwhelming.

- **Content delivery networks (CDN)** are the transparent backbone of the Internet in charge of content delivery.
- CDNs allow for granular cache policy management through a user-friendly dashboard, relieving you of the need to manually tweak individual headers.

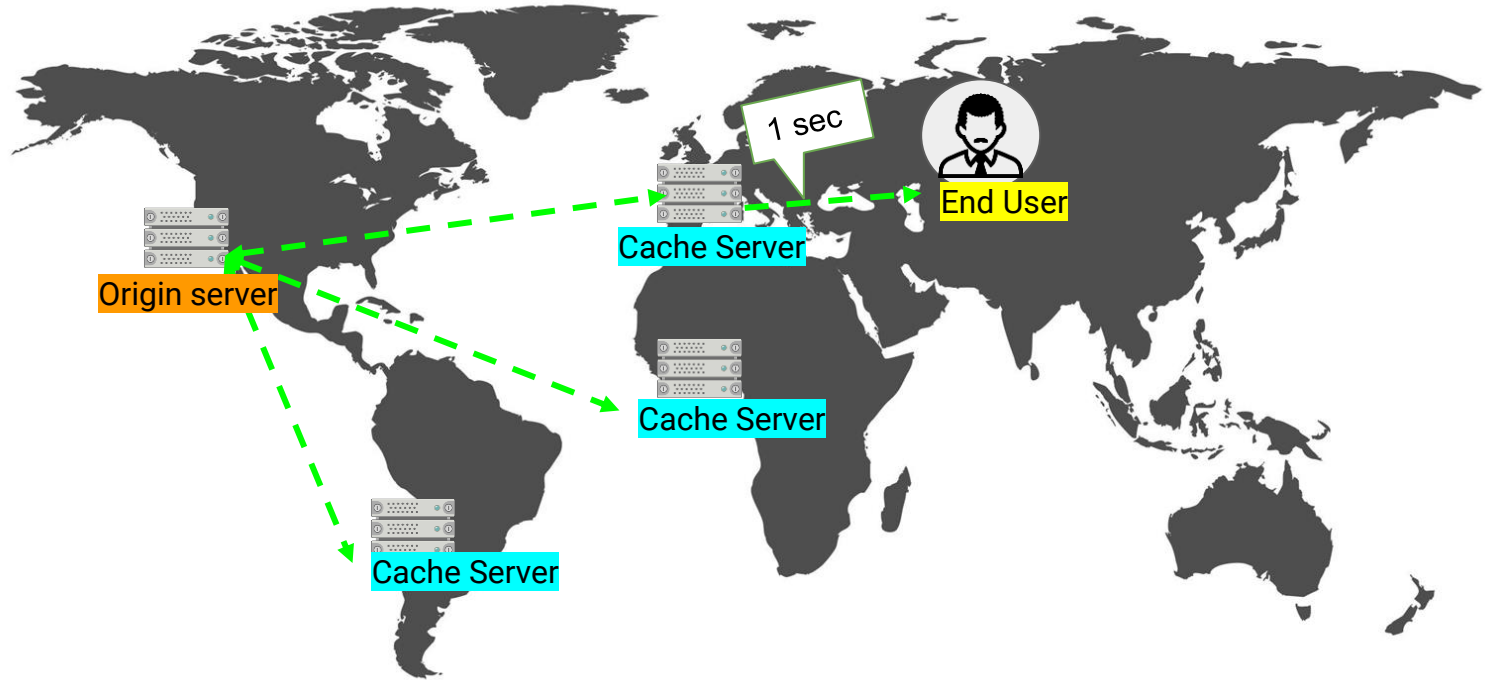
Content Delivery Networks

CDNs augment the browser caching process using proxies.



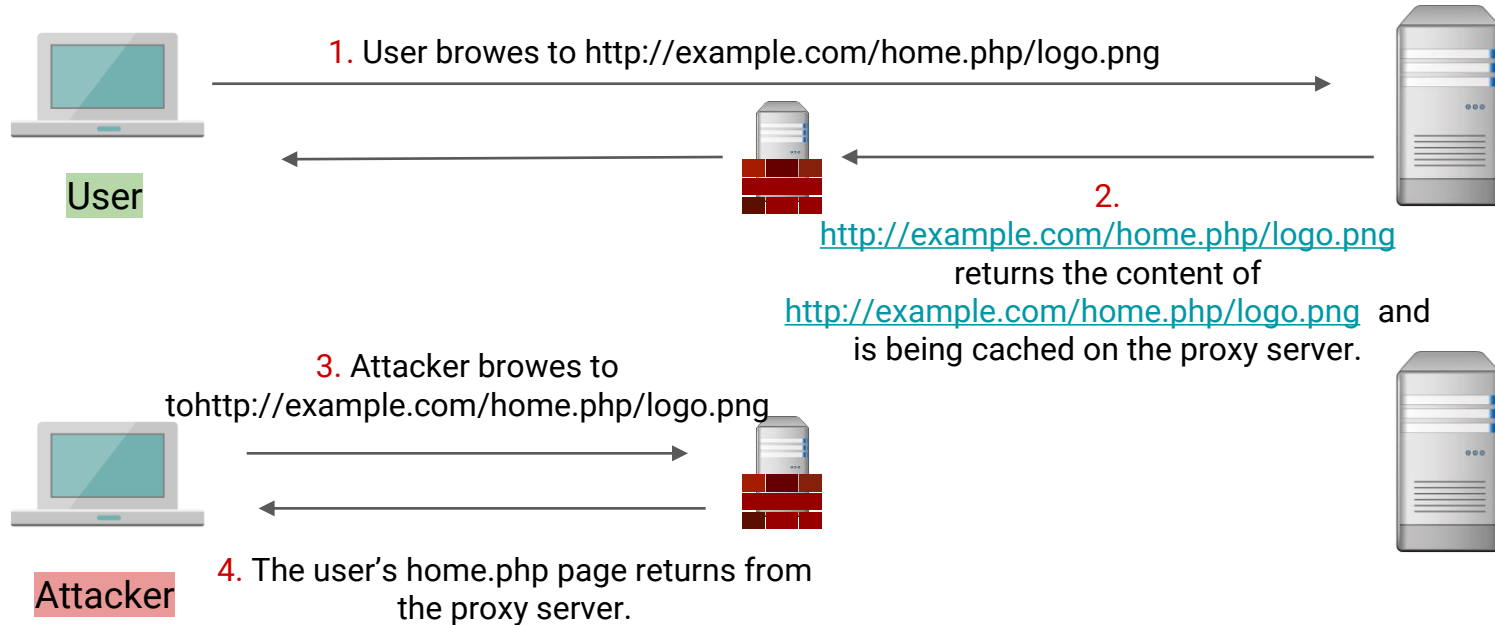
Content Delivery Networks

CDNs augment the browser caching process using proxies.



Web Cache Poisoning

In the next activity, we'll look at web cache poisoning.





Activity: Web Cache Poisoning

In this activity, you will research an important attack called Web Cache Poisoning.

[Activities/Stu_Web_Cache_Poisoning/README](#)

Suggested Time:
15 Minutes





Times Up! Let's Review.

Web Cache Poisoning

Cookies and Sessions

HTTP Request

HTTP requests don't contain information about the application you're using.

Has implications for the following functionalities:

Authorization: How do you remember if the current user is logged in?

Authentication: How do you know that the current user is who they say they are?

User preferences: How do you remember which time zone the user wants to use?



Cookies and **sessions** are two mechanisms that can "remember" this kind of information about a given user across requests.

Cookies

Cookies are pieces of key/value data that client can send to servers to provide information about the identity of the user who sent it.

01

The client sends a request to a server.

02

The *server* responds with the requested data, a a cookie with the Set-Cookie header.

03

The client then stores these cookies as strings in a local database.

04

When the client sends a request to that server later, it sends the cookies as well.

Cookies for Authentication

A server needs to remember that the user is logged in, and should not redirect their request for, e.g. <https://private.site/personal> to <https://private.site/login>.

01

The user logs in via an HTTP POST request to, e.g., `https://private.site/login`

02

The server authenticates the user, and sends an HTTP response with the protected data at `https://private.site/personal`

03

In the response, it sets a header, which might look like: `Set-Cookie: authenticated=true`

04

When the browser makes subsequent requests to `https://private.site`, it will *always* send the authenticated cookie in its HTTP request.

05

This way, the server can check the value of `authenticated` to verify that requests for private resources come from legitimate users.



A session is data used by a server to remember details about a user.

Session IDs

When a server creates a session, it gives that session an ID. It then sends that ID to the client in a cookie

Server Sends

Set-Cookie: **PHPSESSID=100**

Set-Cookie: security=high

Browser Stores

"PHPSESSID=100;security=high"

On subsequent requests, the browser sends the cookie to the server, and the server uses the ID to look up the user's session.

A closer look at Session IDs

Servers that use sessions must still set a cookie containing the user session ID.



User logs in and interacts with applications via HTTP request



Server creates a session and stores data about the user.



Server sends HTTP response with a Set-Cookie: `sessionid=exampleBadSessionId` header.



On subsequent requests, the client sends the cookie `sessionid=exampleBadSessionId`, and the server uses the value `exampleBadSessionId` to look up the user's session in its database.

Two Common Attacks

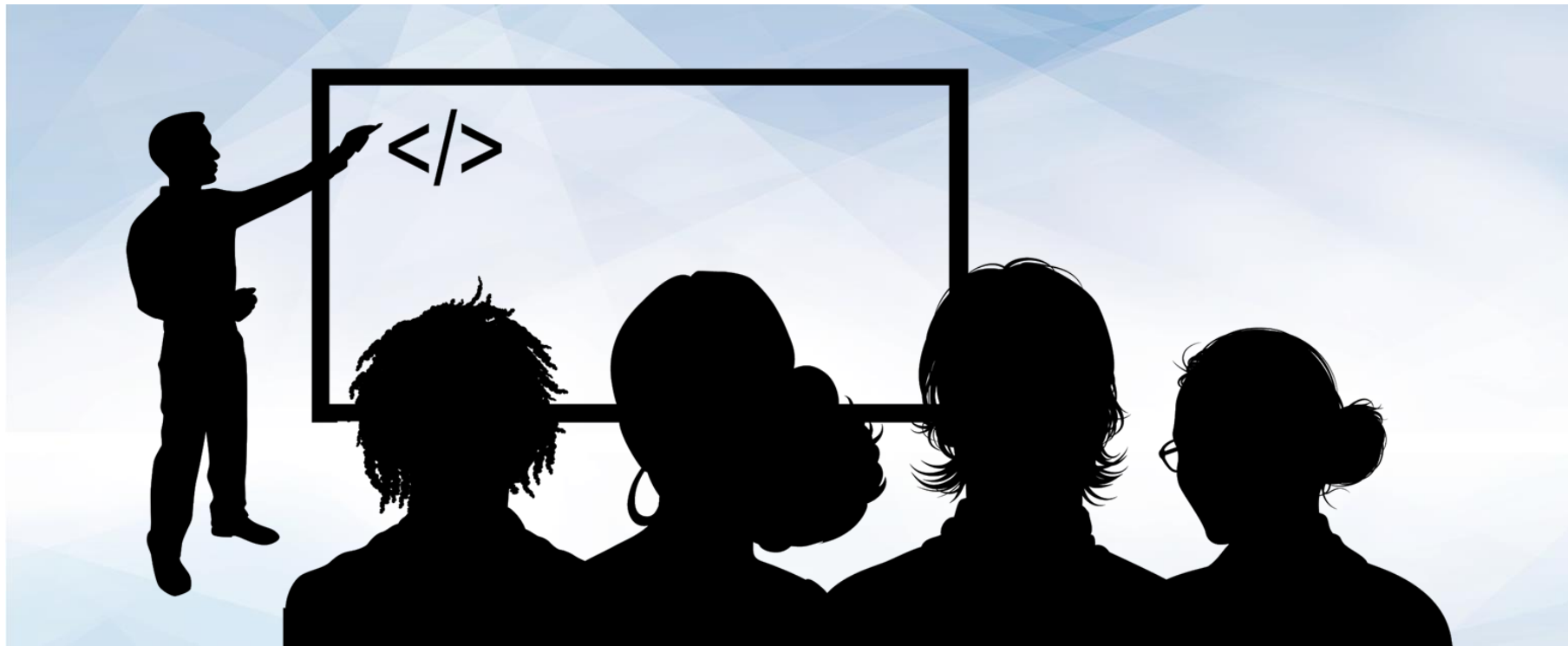
Since sessions still require the use of a potentially vulnerable session ID cookie, they're susceptible to attack.

Impersonation: Stealing and sending a user's cookies and/or session ID, so the web application thinks you're someone else.

Forgery: Creating fake cookies and / or session IDs that the application understands and interprets "correctly", but which lead to unintended behavior.



Tampering Cookies



Instructor Demonstration

Gruyere Demo



Student Activity: Manipulating Cookies

In this activity, you will use Gruyere to explore how cookies can be used for authentication and how they can be tampered with.

[Activities/Stu_Manipulating_Cookies/README](#)

Suggested Time:
20 Minutes

9





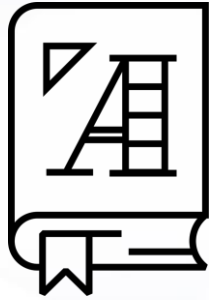
Times Up! Let's Review.

Manipulating Cookies

Break



Validation



Client-side validation prevents the user from sending certain data through a form.



Why is client-side validation important?

Session Hijacking

Anatomy of a Session Attack

Servers will process any request with a correct session cookie, even if it comes from an attacker.

1. A legitimate user, Jane, log into a baking application and receives a cookie like: `SESSID=100`
 1. An attacker sends request from his own computer with the cookie `SESSID=100`
 1. The server responds to the attacker as if they were Jane, because they used a cookie containing the Jane's session ID.
-

Anatomy of a Session Attack

Accessing the Victim's Session ID:

This attack requires knowing the victim's session ID. Two methods:

1. **Read it through JavaScript**

Attackers can read victims' cookies by exploiting an XSS vulnerability.

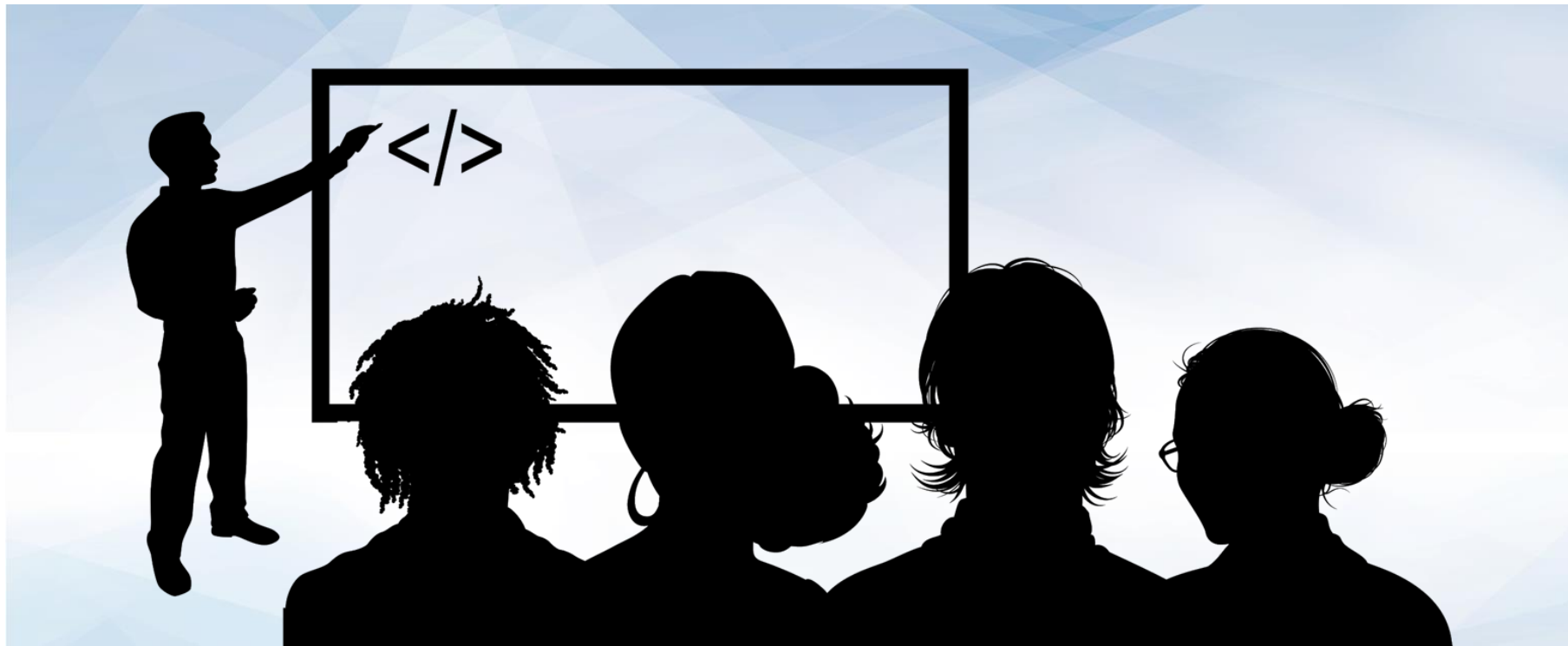
1. **Guess:** Some servers generate session IDs that are easy to guess.

In this case, attackers can forge session IDs.



Instructor Demonstration

httponly



Instructor Demonstration

Weak Session IDs



Student Activity: Weak Session IDs

In this activity, you'll use BurpSuite to study DVWA's weak session ID algorithm.

[Activities/Stu_Weak_Session_IDs/README](#)

Suggested Time:
20 Minutes














Times Up! Let's Review.

Weak Session IDs

Today's Objectives

By the end of class, you will be able to:

-  Define caching and distinguish between private and public caches.
 -  Explain the process behind web cache poisoning
 -  Explain how cookies and sessions provide ways for servers to remember specific
 -  information about the client.
 -  Describe two common session attacks: Impersonation and forgery.
 -  Distinguish between client-side and server-side validation.
 -  Explain how session hijacking is an impersonation attack on server sessions.
 -  Discuss why servers often store sensitive, larger, and long-term data in sessions instead of cookies.
 -  Inspect weak session IDs generated by a live web app.
-