



HTTP with Sessions and Cookies

Cybersecurity
Web Development Day 1



Class Objectives

By the end of today's class, we'll be able to:



Understand HTTP requests and responses.



Use the curl command-line tool to make GET and POST requests and examine the responses.



Manage cookies using the Chrome extension Cookie-Editor.



Use Chrome's Developer Tools to audit HTTP request and response headers.

The Web from a Security Perspective

Intro to Offensive Security

This week will prepare us for the next four units, focused on **Offensive Security**.

In this unit, we'll learn the foundations of modern web application architecture.



Then, we will learn popular exploits in Web Vulnerabilities.



After that, we will dive into ethical hacking and penetration testing.



The Web: A Security Perspective

In previous units, we learned about system administration and networking concepts.

This week, we will build on those concepts as we explore how web-based communication works in the client-server model.

We'll examine how we communicate over the web (networking) and how websites and web applications are created and deployed (system administration).

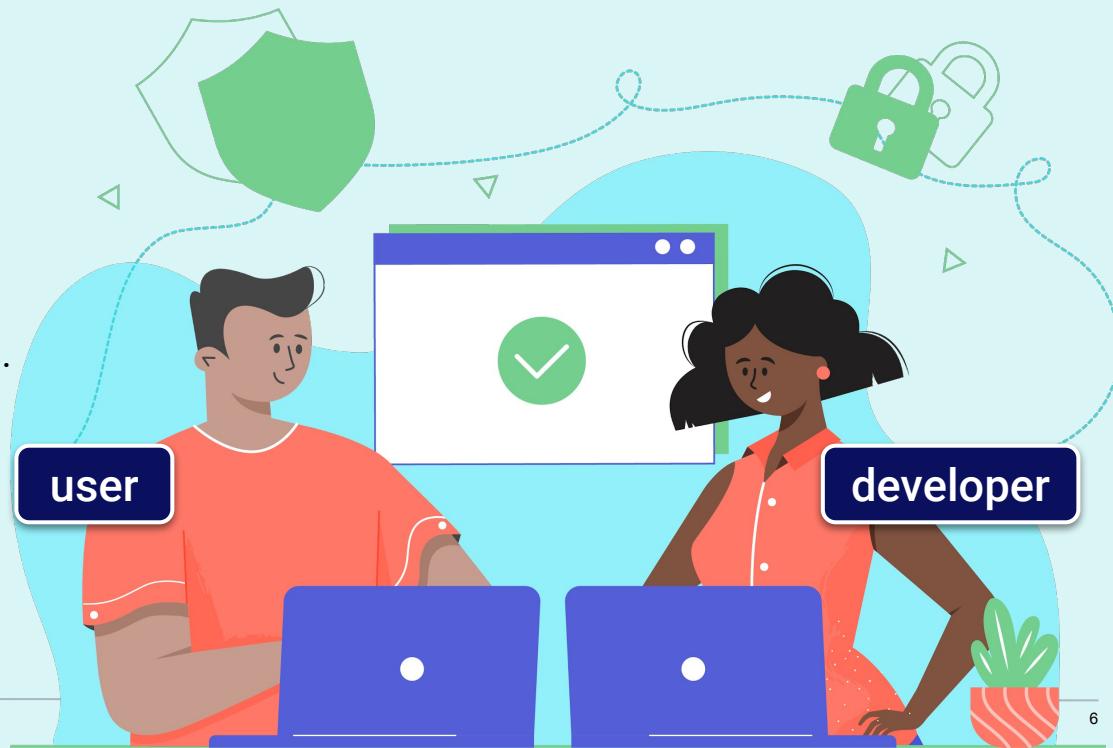


Web Development Overview

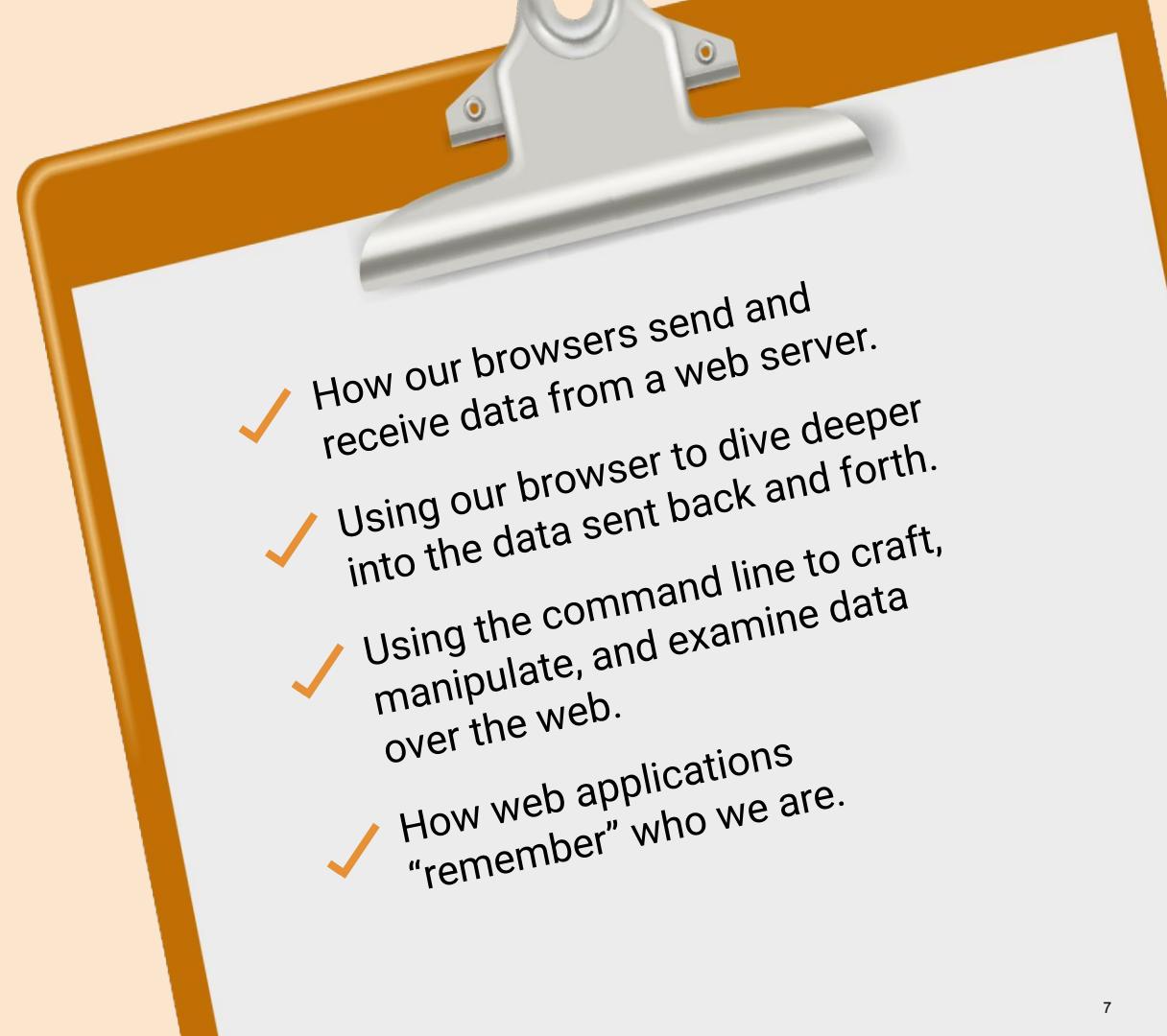
Throughout this week, we will look at how web applications work from the perspective of the user and the web developer/architect.

This week will cover the following:

- What happens behind the scenes when we visit a webpage.
- How websites “remember” us through cookies and sessions.
- The architecture of web applications.
- Deploying and managing data in a web application.



Today's topics will focus on what happens when we do everyday online activities, such as visit a website or log into an e-commerce site.



- ✓ How our browsers send and receive data from a web server.
- ✓ Using our browser to dive deeper into the data sent back and forth.
- ✓ Using the command line to craft, manipulate, and examine data over the web.
- ✓ How web applications "remember" who we are.

The Web: A Security Perspective

It is increasingly important for security professionals to understand the web.

- ▶ Organizations are incorporating more web-based applications and services, such as Salesforce, into their operations.
- ▶ Companies are moving their on-premises infrastructure and services to the cloud, e.g. using cloud-based backup services for servers.
- ▶ Most people access the web on a daily basis, whether for shopping or streaming videos.



The Web: A Security Perspective

Some of the major security roles that focus on web technologies are:

- ▶ **Web application security engineers** must deeply understand how their company's web application works and how to secure it.
- ▶ **Penetration testers** are ethical hackers who try to find and exploit vulnerabilities and gain privileged access to web applications.
- ▶ **Risk assessors** must understand the risk profiles and tolerances within web applications so they can help companies follow best practices and policies.





BRIAN BARRETT SECURITY 03.31.2020 03:32 PM

Hack Brief: Marriott Got Hacked. Yes, Again

The hotel chain has suffered its second major breach in 16 months. Here's how to find out if you're affected.



Up to 5.2 million members of the Marriott Bonvoy loyalty program may have had their personal information stolen, the hotel chain announced Tuesday. PHOTOGRAPH: PAUL YEUNG/BETTY IMAGES

IN NOVEMBER 2018, hotel giant Marriott disclosed that it had suffered one of the [largest breaches in history](#). That hack compromised the information of 500 million people who had made a reservation at a Starwood hotel. On Tuesday, Marriott announced that it had once again been hit, with up to 5.2 million guests at risk. Which is a kind of progress, in a way?

The details of this latest hack seem to be not quite as devastating as the last one, too, given that sensitive information like passport numbers doesn't seem to be affected. Still, that a major company could get hit twice in such a relatively short time frame underscores how at-risk your data is—and how not enough is being done to protect it.

The Hack

According to details [provided](#) by Marriott on Tuesday, the intrusion dates back to mid-January, when someone used the credentials of two franchise property employees—whether those credentials were stolen is unclear at this point—to access an “unexpected amount of

The Web: A Security Perspective

The threats embedded in the modern web are constantly growing.

-  The web represents most organizations' largest attack surfaces.
-  The number of users accessing the web at all times, through PCs, phones, or other devices, is constantly growing.
-  The collective amount of personal data stored on the web is growing significantly. Attackers can steal and sell it.

Understanding the Web: HTTP Requests and Responses

Preliminary Information

Securing the web requires in-depth knowledge of topics we covered previously:



Client-server architecture



HyperText Transfer Protocol (HTTP)



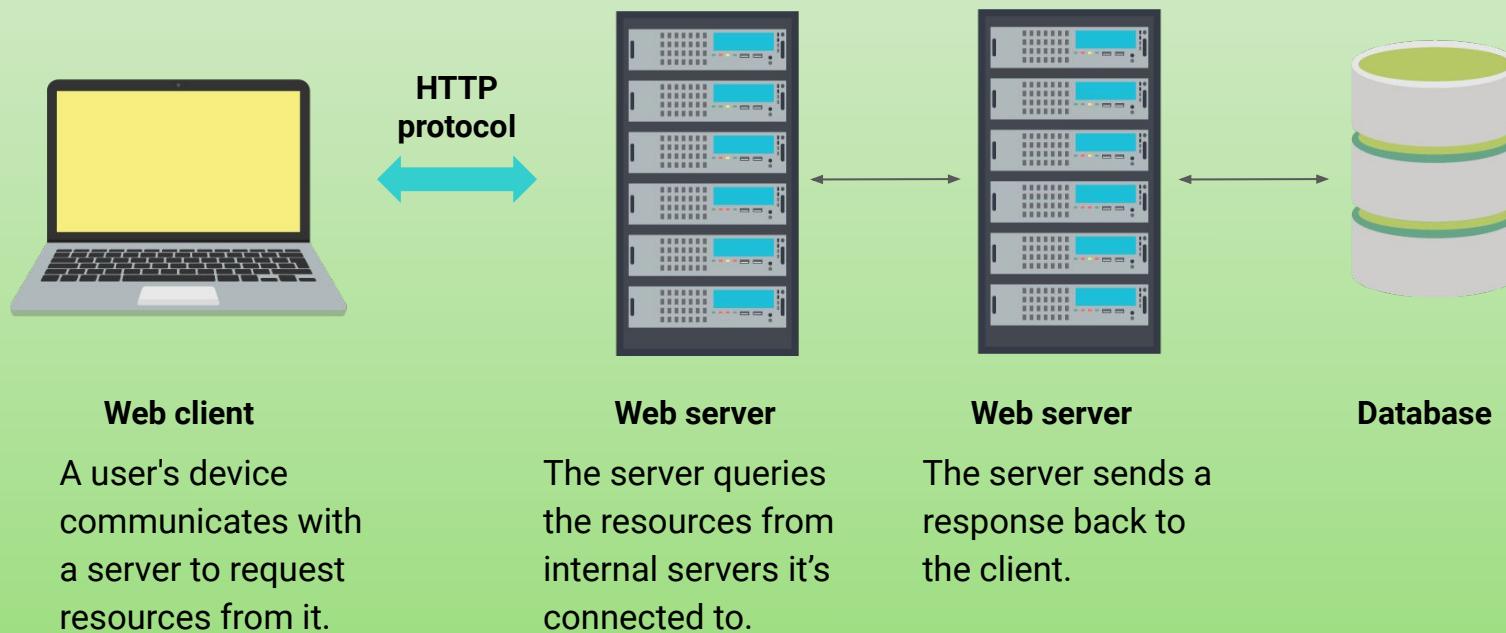
HTTP requests



HTTP responses

Client-Server Architecture

The client-server model is an exchange of information, a cycle of **requests and responses** between **clients and servers**.



HTTP Requests

Clients and servers use the **HTTP protocol** to communicate on the web.



HTTP is a Layer 7: Application Layer protocol used to transfer **webpages**, **static assets**, such as images and HTML/markup files, and **raw data**, such as MP4 video or MP3 audio.



When a client initiates communication, it generates an **HTTP request**. The server receives the request, finds the requested resource, and sends it back.



1



2



HTTP Methods

There are various types of requests, known as **HTTP methods**, which indicate the specific actions between the client and server. For example:

01

A user **requests data** from a server:

I'd like to see my friend's photos on Facebook.

02

A user **gives data** to a server:

Here are my credentials for my LinkedIn account.

03

A user **updates data** that already exists on a server:

Two new accounts are added to the employee directory database.

HTTP Methods

There are various HTTP methods:

HTTP Method	Description
GET	Requests data <i>from</i> a server.
HEAD	Identical to GET, but the server does not send the response body.
POST	Sends data <i>to</i> a source, often changing or updating a server.
PUT	Replaces current data with the new value.
DELETE	Deletes a specified resource.
CONNECT	Establishes a tunnel to the server.
OPTIONS	Lists the communication options for target resource.

HTTP Methods

We'll focus on GET and POST requests.

HTTP Method	Description	Example
GET	Requests data <i>from</i> a server.	When you open a browser and go to amazon.com, the HTTP client (your browser) asks to GET the data that the URL (amazon.com) points to. That data is the webpage.
POST	Sends data <i>to</i> a source, often changing or updating a server.	Once your browser goes to amazon.com, you log into your Amazon account. The client sends a POST request that contains your credentials for logging in.

Anatomy of a GET Request

```
GET /js/analytics.js HTTP/1.1  
Host: www.target-server.com  
Connection: keep-alive  
Upgrade-Insecure-Request: 1  
Accept: text/js, text/html. */*  
Accept-Language: en-us  
Accept-Encoding: qzip, deflate  
User-Agent: Mozilla/4.0
```

Request Line

Headers

White space

Request Body
(optional)

Request Line

```
GET /js/analytics.js HTTP/1.1
```

Host: www.target-server.com

Connection: keep-alive

Upgrade-Insecure-Request: 1

Accept: text/js, text/html. */*

Accept-Language: en-us

Accept-Encoding: qzip, deflate

User-Agent: Mozilla/4.0

A request line contains the request method, the name of the requested resource, and the version of HTTP in use.

The request line can also contain query parameters, which the client can use to send data to the server.

Headers

```
GET /js/analytics.js HTTP/1.1
Host: www.target-server.com
Connection: keep-alive
Upgrade-Insecure-Request: 1
Accept: text/js, text/html. */*
Accept-Language: en-us
Accept-Encoding: qzip, deflate
User-Agent: Mozilla/4.0
```

Headers contain additional details about the requested resource. They contain many actions with security implications, such as authentication and remembering user resources.

You don't have to remember every type of header. We'll cover some of the most important ones in class.

Whitespace

```
GET /js/analytics.js HTTP/1.1
Host: www.target-server.com
Connection: keep-alive
Upgrade-Insecure-Request: 1
Accept: text/js, text/html. */*
Accept-Language: en-us
Accept-Encoding: qzip, deflate
User-Agent: Mozilla/4.0
```



Whitespace is a blank line indicating the end of the request.

HTTP Request: Request Line

```
GET /js/analytics.js HTTP/1.1
Host: www.target-server.com
Connection: keep-alive
Upgrade-Insecure-Request: 1
Accept: text/js, text/html. */*
Accept-Language: en-us
Accept-Encoding: qzip, deflate
User-Agent: Mozilla/4.0
```

The request method is GET.

The requested resource is /js/analytics.js.

This resource is the file path from the domain stated in the header.

The request browser uses HTTP version 1.1.

HTTP Request: Headers

```
GET /js/analytics.js HTTP/1.1  
Host: www.target-server.com  
Connection: keep-alive  
Upgrade-Insecure-Request: 1  
Accept: text/js, text/html. */*  
Accept-Language: en-us  
Accept-Encoding: qzip, deflate  
User-Agent: Mozilla/4.0
```

Host contains the domain name of the target server.

Note that the protocol (HTTP) + host + file path in the request line is the URL we see in the browser:

<http://www.target-server.com/js/analytics.js>

HTTP Request: Headers

```
GET /js/analytics.js HTTP/1.1
Host: www.target-server.com
Connection: keep-alive
Upgrade-Insecure-Request: 1
Accept: text/js, text/html. */*
Accept-Language: en-us
Accept-Encoding: qzip, deflate
User-Agent: Mozilla/4.0
```

Connection tells the server to keep open the TCP connection used for this HTTP transfer after sending the response.

This allows the client-server to reuse the connection for later requests.

The alternative is performing a TCP handshake, which is much slower.

HTTP Request: Headers

```
GET /js/analytics.js HTTP/1.1
Host: www.target-server.com
Connection: keep-alive
Upgrade-Insecure-Request: 1
Accept: text/js, text/html. */*
Accept-Language: en-us
Accept-Encoding: qzip, deflate
User-Agent: Mozilla/4.0
```

Upgrade-Insecure-Requests tells the server to turn this HTTP connection into HTTPS, so the response and future communications will be encrypted.

Accept tells the server that the client expects a JavaScript or HTML document in response, but will accept data of any type (*/*).

User-Agent tells the server that this request comes from a Mozilla 4.0 browser.

Headers

→ Other important headers include:

- **Authorization:** Contains credentials to authenticate a user with a server.
- **Referer:** Contains the address of the previous webpage that linked to the currently requested page. Allows servers to identify where people are visiting from, and can use that data for analytics, logging, or optimized caching. (If a link from a Google search led to the current page, the referrer is Google.)
- **Cookie:** Contains stored HTTP cookies previously sent by the server with the Set-Cookie response header.

Requests with Query Parameters

GET requests can also request data with query parameters.

GET /articles?tag=latest&author=jane HTTP/1.1

Host: www.target-server.com
Connection: keep-alive
Upgrade-Insecure-Request: 1
Accept: text/js, text/html. */*
Accept-Language: en-us
Accept-Encoding: qzip, deflate
User-Agent: Mozilla/4.0

Query parameters are useful for specifying the parts of a resource to receive or send data.

This GET request specifically requests articles with two parameters: the **latest** articles, created by author **jane**.

Requests with Query Parameters

GET requests can also request data with query parameters.

```
GET /articles?tag=latest&author=jane HTTP/1.1
```

Note the syntax for parameters:

```
[path]?[firstParam]=[value]&[secondParam]=[value]
```

Query parameters are useful for specifying the parts of a resource to receive or send data.

This GET request specifically requests articles with two parameters: the `latest` articles, created by author `jane`.

Anatomy of a POST Request

HTTP POST requests are similar to GET requests, but include a request body below the whitespace.

```
POST /api/users HTTP/1.1
Host: www.target-server.com
User-Agent: Mozilla/4.0
Accept: */
Content-Length: 23
Content-Type: application/x-www-form-urlencoded
```

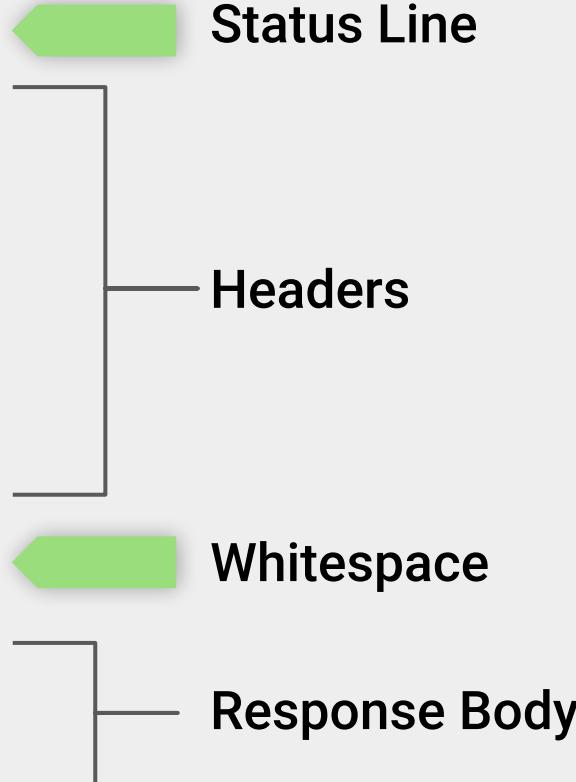
username=Jane+Doe&password=p455

The request body may contain information such as login credentials or a file to be uploaded.

In the current example, our request body contains login credentials.

Anatomy of an HTTP Response

```
HTTP/1.1 200 OK
Date: Nov 12 02:12:12 2018
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) {
...
}
```

Anatomy of an HTTP Response

```
HTTP/1.1 200 OK
Date: Nov 12 02:12:12 2018
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) {
...
}
```

A **status line** contains the response status code and phrase that translates the result, such as OK or Error.

Headers contain additional information about the response, similar to response headers.

A **whitespace** separates the header with the following response body.

A **response body** contains the resource requested by the client.

Anatomy of an HTTP Response

HTTP/1.1 200 OK

Date: Nov 12 02:12:12 2018

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) {
```

```
...
```

The response uses unencrypted **HTTP/1.1**.

The status code is **200 OK**, meaning the request was processed properly.

HTTP Response: Status Line

HTTP/1.1 200 OK

Date: Nov 12 02:12:12 2018

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) {
```

```
...
```

Date contains a timestamp of when the response was generated.

Server indicates the server is running PHP version 5.5.9 on Lubuntu with kernel version 4.21.

HTTP Response: Status Line

```
HTTP/1.1 200 OK
Date: Nov 12 02:12:12 2018
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) {
...
}
```

Set-Cookie tells the client to create a cookie called **SESSID** with the value **8toks**, and that this cookie can only be set by the server with HTTP.

We'll discuss cookies and **httponly** in greater detail later.

HTTP Response: Response Body

```
HTTP/1.1 200 OK
Date: Nov 12 02:12:12 2018
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) {
...
}
```

Below the whitespace is the response body, which contains the source code of the resource requested in the GET request.

Status Codes

General status codes include:

- ▶ **200** codes indicate *success*.
- ▶ **300** codes indicate *multiple choices*, meaning the server can respond to the request in more than one way.
- ▶ **400** codes indicate *client errors*, meaning the client sent an improperly formatted request.
 - You probably recognize the **404** error.
- ▶ **500** codes indicate *server errors*, meaning the server application failed somehow.



404 ERROR

PAGE NOT FOUND





Activity: HTTP Request and Response

In this activity you will investigate HTTP logs to figure out the sequence of events that took place during an attack on your company's web server.

Suggested Time:
15 Minutes





Time's Up! Let's Review.

Examining HTTP and Cookies with DevTools

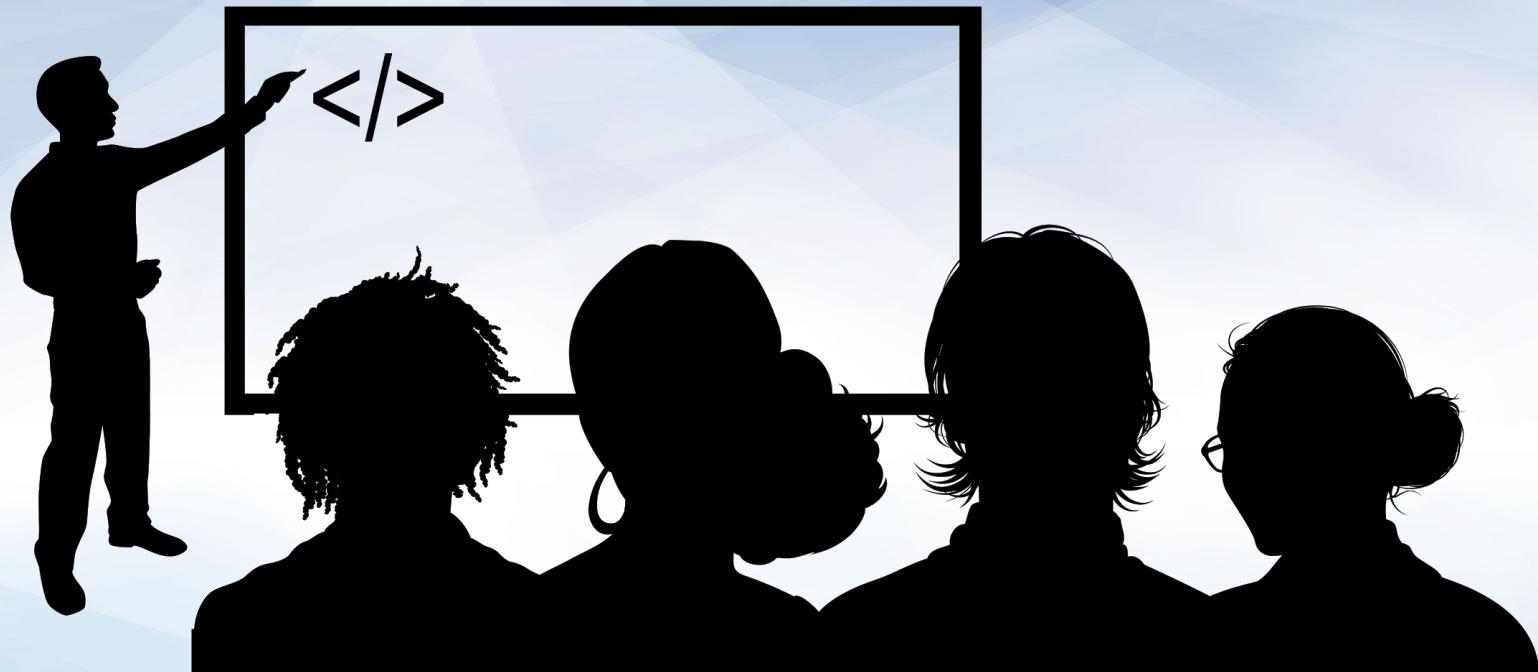
Chrome Developer Tools

Now we will deconstruct and examine a web application using the tools within a browser. These allow us to:

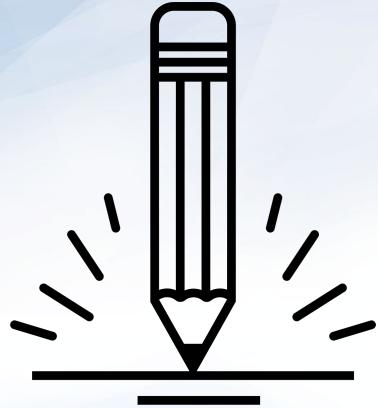
- ▶ **Understand** the request and response flows between an HTTP client and server.
- ▶ **Examine** the security implementations of a website through its response headers.
- ▶ **Identify** where cookies are used in the HTTP request-response cycle.

The screenshot shows the Chrome Developer Tools Network tab while viewing the Google homepage. The main content area displays the Google logo with a red heart above it, and a small illustration of a scientist at a desk. Below the logo is a search bar with a microphone icon and two buttons: "Google Search" and "I'm Feeling Lucky". A message at the bottom reads: "To public health workers and to researchers in the scientific community, thank you". The Network tab lists 24 requests. The first request is for the page itself, with a status of 200, type of gif, initiator "(index)", size of 61 KB, and time of 9 ms. Other requests include various fonts, images (like the logo and scientist), and scripts for Google services like Analytics and Ads. The timeline at the bottom shows the progression of these requests over approximately 2.69 seconds, ending with the "DOMContentLoaded" event.

Name	Status	Type	Initiator	Size	Time
thank-you-public... KF0mCnqEu92F...	200	gif	(index)	61 KB	9 ms
KFOlCnqEu92Fr...	200	font	(index)	10 KB	10 ms
KFOlCnqEu92Fr...	200	font	(index)	7 KB	7 ms
i2_eec824b0.png	200	png	(index)	12 KB	12 ms
data:image/png;...	200	png	(index)	0 KB	0 ms
googlemic_color...	200	png	(index)	2 KB	2 ms
desktop_search...	200	webp	(index)	2 KB	2 ms
AI08mxEsxr5Y...	204	text	(index)	61 KB	25 ms
gen_204?is=w... nav_logo299.webp	204	text	(index)	40 B	41 ms
rs=ACT900FbDq...	200	webp	(index)	2 KB	2 ms
rs=ACT900FbDq...	200	script	(index)	1 KB	21 ms
gen_204?atyp=c...	204	text	is=ACT...	17 B	29 ms
m=RqkLvf,aa,ab...	200	script	is=ACT...	8 ms	
m=NpD4ec,wkYr...	200	script	is=ACT...	2 ms	
rs=A2AYrTsm3M...	200	script	(index)	4 ms	
m=OG6ZHu,T7X...	200	script	is=ACT...	3 ms	
cb=gapi.loaded_0	200	script	is=AA2...	3 ms	
so?origin=https...	200	text	is=AA2...	2 ms	
gen_204?atyp=c...	204	text	is=ACT...	17 B	83 ms
ui	204	text	m=Rqx...	0 B	44 ms
gen_204?atyp=c...	204	text	(index)1	12 KB	43 ms



Instructor Demonstration Chrome DevTools



Activity: Inspecting with Developer Tools

In this activity, you will use your browser's Developer Tools to examine the request and response set from www.crowdstrike.com.

Suggested Time:
10 Minutes





Time's Up! Let's Review.



Countdown timer

15:00

(with alarm)

Break



curl

Introducing curl

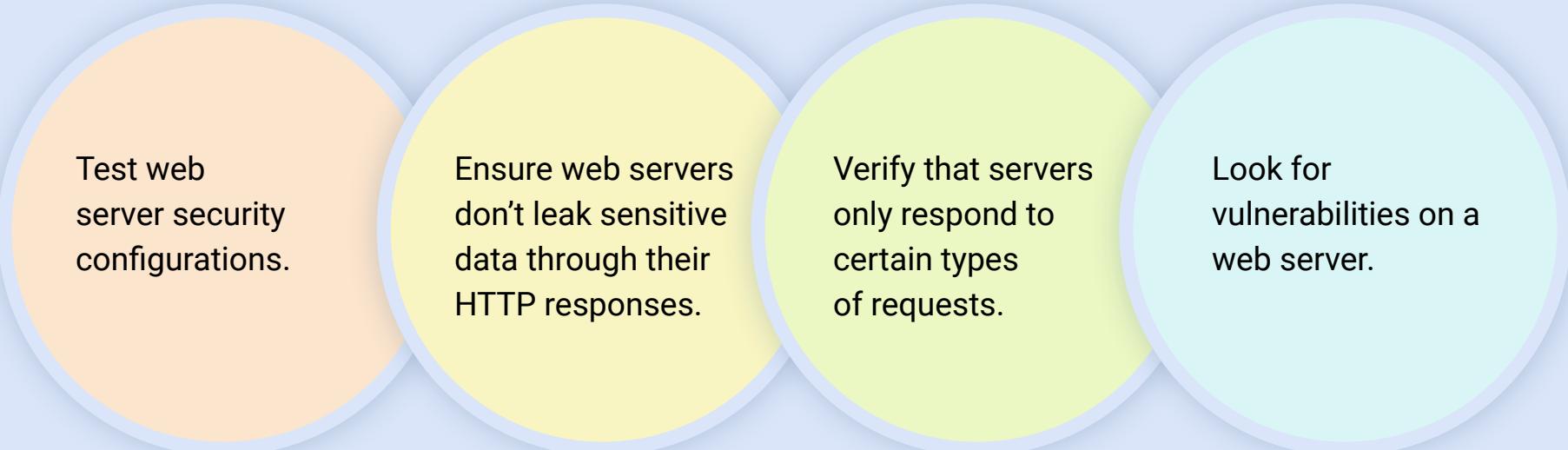
We can't always examine HTTP requests and responses through a browser.

- Sometimes, the tools you can use to send and receive HTTP requests are limited.
- For example, when working through a container that has no user interface, you'll need a command-line tool to send and receive HTTP requests.
- Cybersecurity professionals need to be able to quickly test HTTP requests in a way that can be automated, but also allows them to make adjustments.
- The command-line tool curl allows security professional to do exactly this.



curl

curl is a command-line client that allows us to send data to and from servers. This allows security professionals to:



Test web server security configurations.

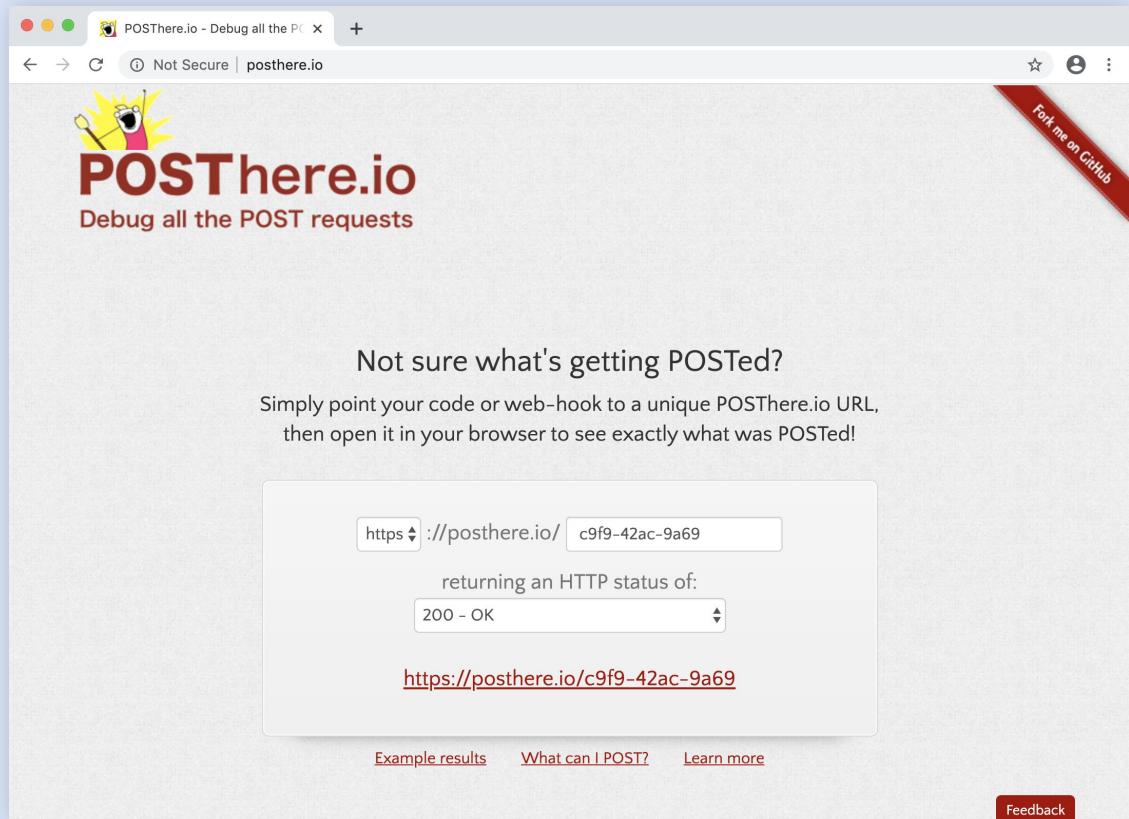
Ensure web servers don't leak sensitive data through their HTTP responses.

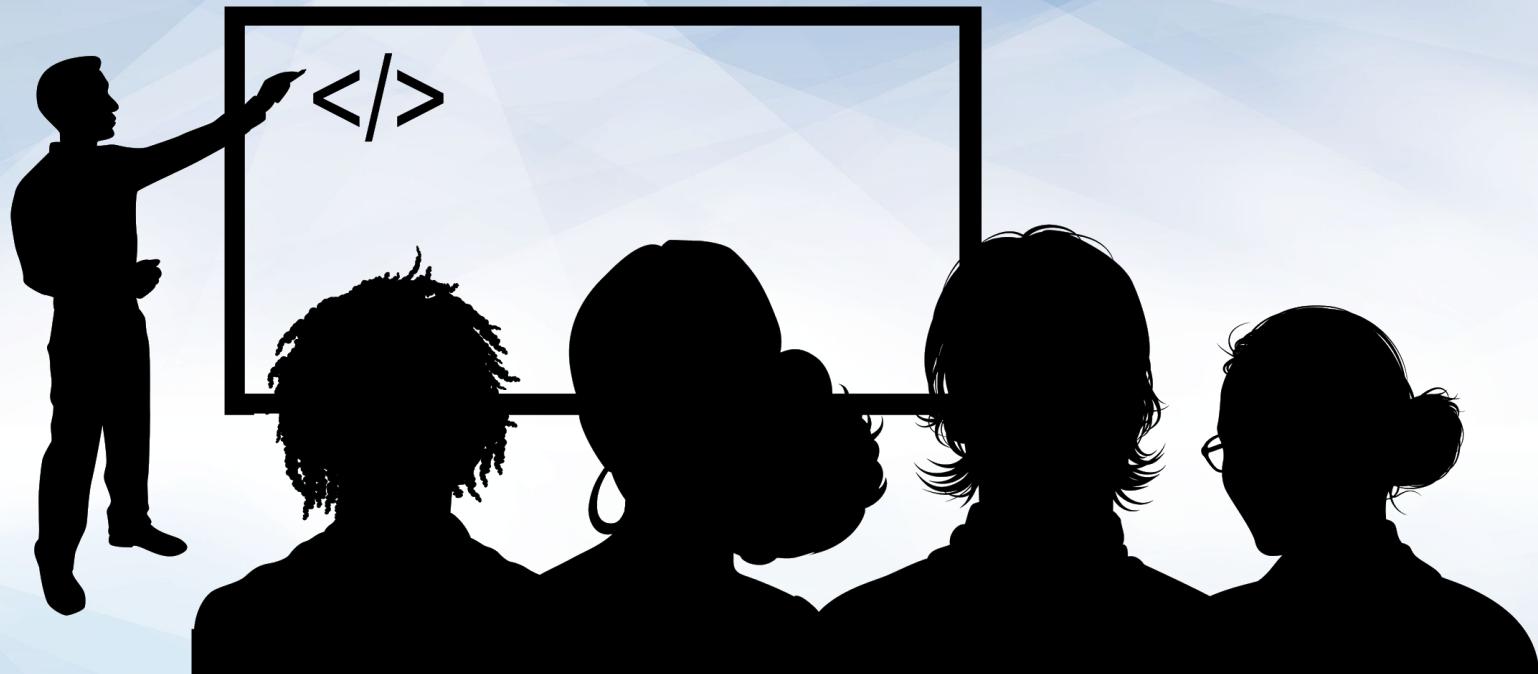
Verify that servers only respond to certain types of requests.

Look for vulnerabilities on a web server.

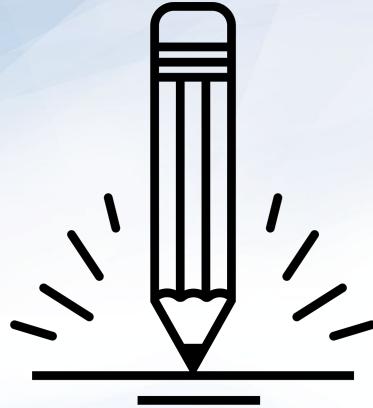
curl

We'll demonstrate curl using the website POSThere.io.





Instructor Demonstration
curling



Activity: Using curl

In this activity, you will write various `curl` commands to interact with an HTTP web server and return responses.

Suggested Time:
15 Minutes





Time's Up! Let's Review.

Sessions and Cookies

Sessions and Cookies

HTTP resources are inherently **stateless**, meaning that when your browser requests a webpage, the webpage can't distinguish you from others.

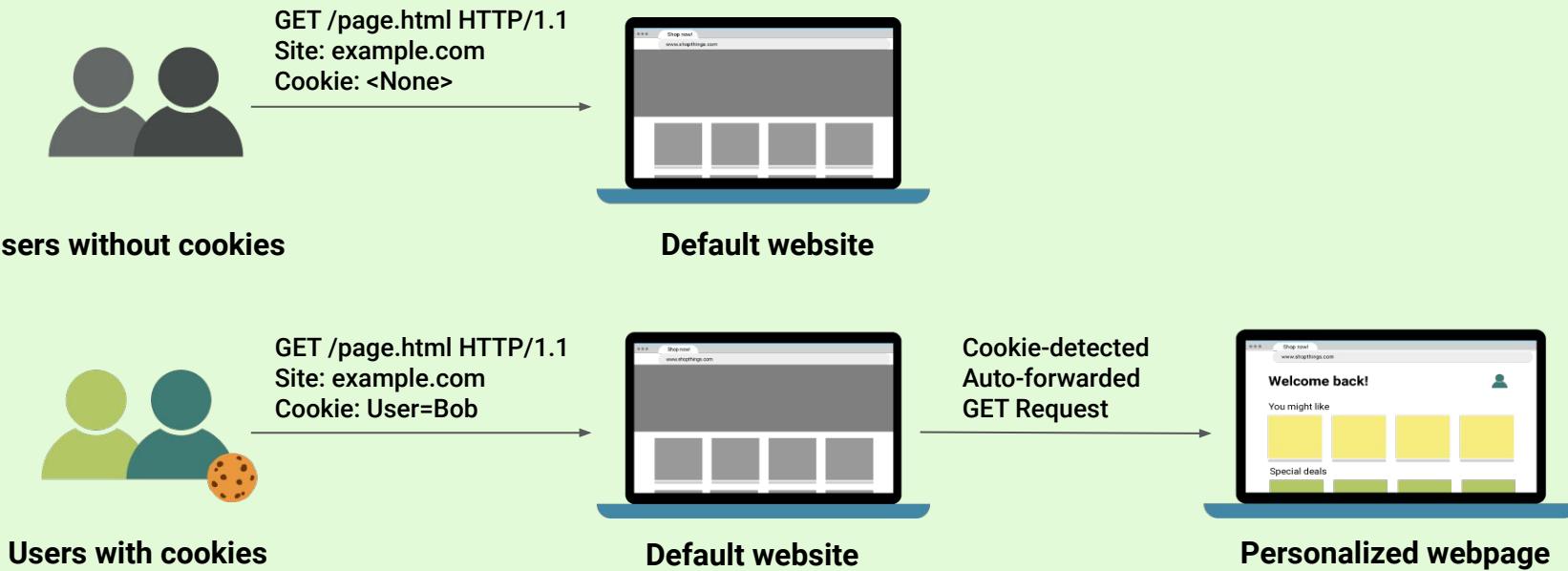
Websites need a way to deliver content that is specific to each user. To do so, they establish sessions with cookies.

- **Sessions** are unique server-side sets of user data used to customize webpages for the specific user accessing them.
- **Cookies** are small pieces of text data that, when sent by an HTTP server's response header, are saved by the user's HTTP client.



Sessions and Cookies

You visit a shopping website and register an account with your first and last name. As soon as you do this, a session is created that saves your name to the site's database.



How a Session is Created

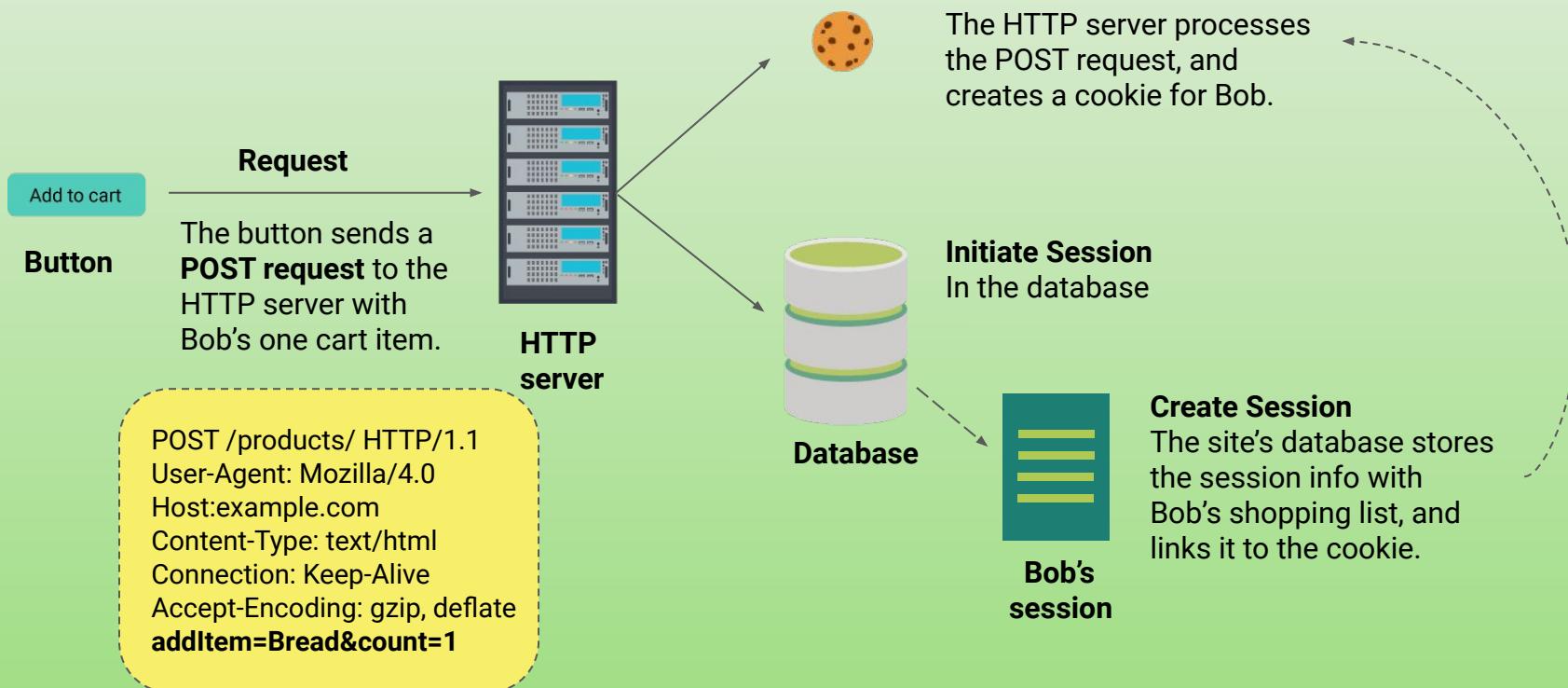
Before the session:

Bob visits a shopping site and clicks a button to add one loaf of bread to his shopping cart.



Bob's browser

How a Session is Created

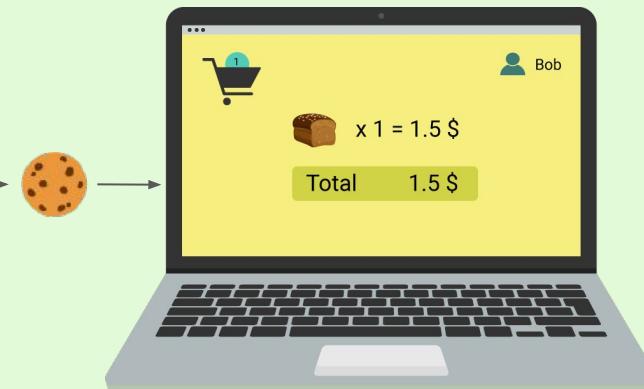


How a Session is Created

The **HTTP server** sends a **response** back to Bob's browser with:

- A **cookie**, sent through the **response header**.
- Bob's **session information**, embedded in the **response body**.

HTTP/1.1 200 OK
Date: Fri, 13 Mar 2020 12:28:53 GMT
Server: Apache/2.2.14 (Win 32)
Content-Length: 88
Content-Type: text/html
Set-Cookie: cart=Bob
<html><head>Bob's cart<head>...

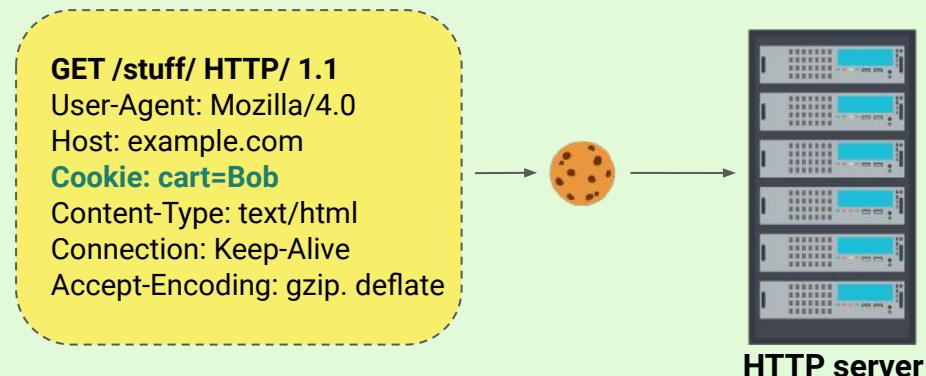


Bob's browser

How a Session is Created

Resuming the response:

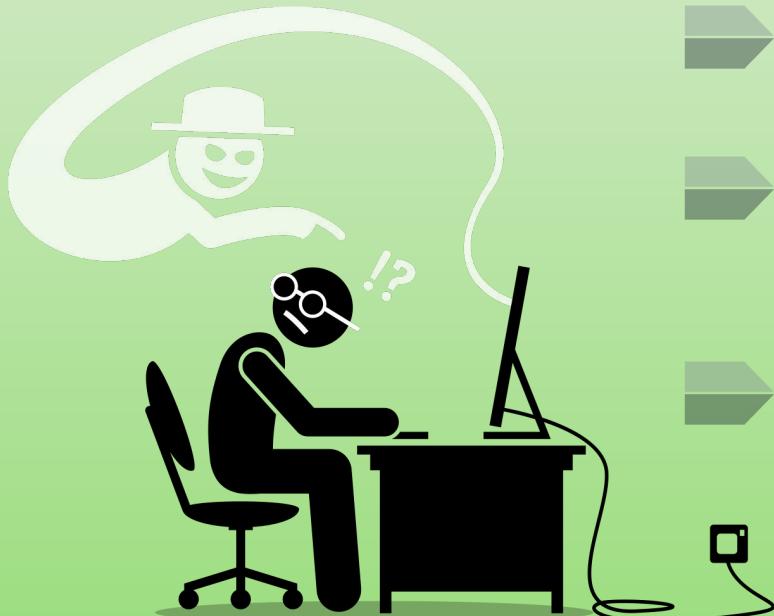
When Bob revisits the webpage, his browser sends the cookie back through the **request header**.



Implementations of cookies vary between sites. Some are more secure than others.



If Bob's account was the fifth account to ever be created on a website, his cookie might look like: **Set-Cookie: Session-ID=5**



An attacker can change their cookie's value so that when they make a request, they do so as Bob.



Modern session cookies contain uniquely hashed values specific to users, e.g. **Set-Cookie: Session-ID=299367cd4c0e2c6d3ae6e09eda4e0590**



While this keeps session ID cookies from being reverse engineered, an attacker can commit man-in-the-middle attacks by stealing cookies.

Session with Cookie-Editor

We'll use a Chrome extension called Cookie-Editor to take a closer look at sessions and cookies in our browser.

We'll complete the following steps:

1

Install the
Chrome extension
Cookie-Editor.

2

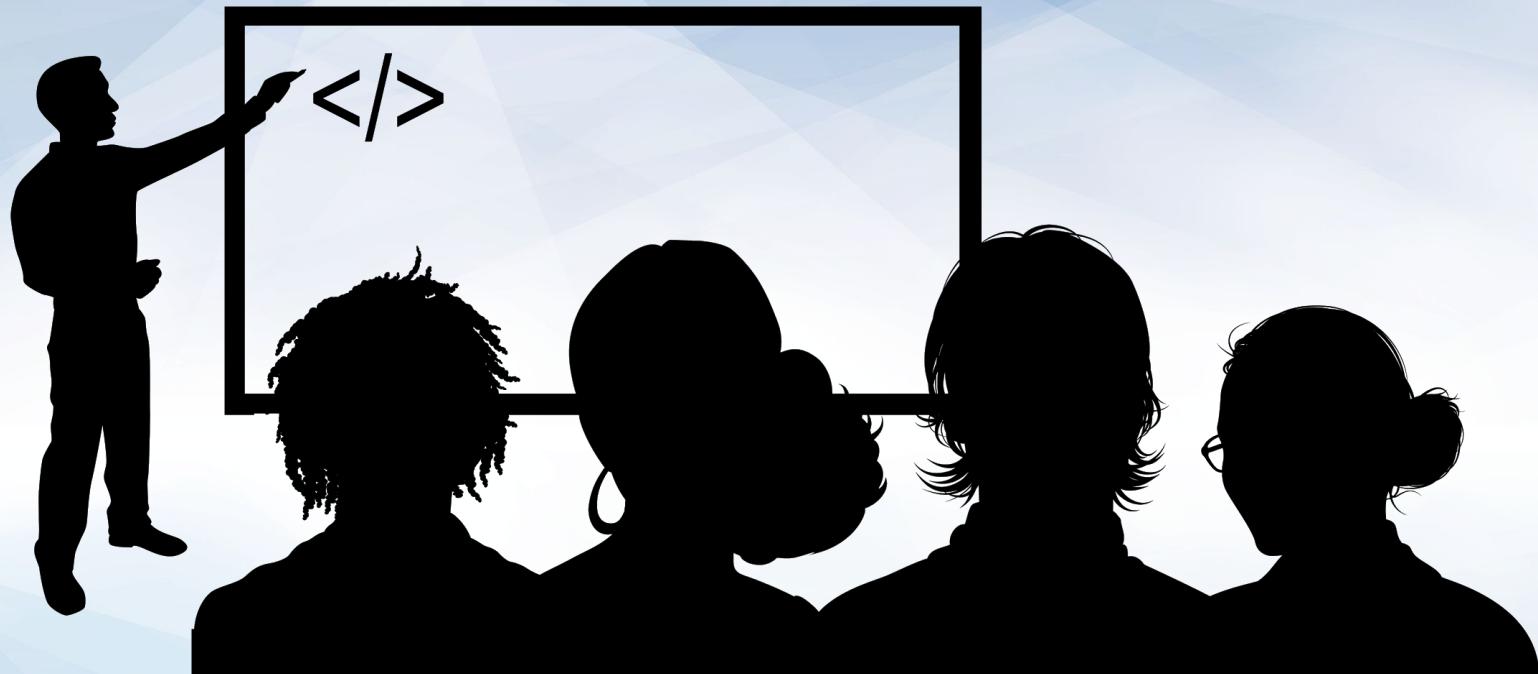
Create a user on a
webpage and
export the cookie
to the clipboard
with Cookie-Editor.

3

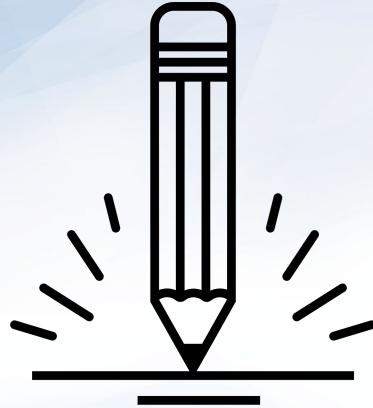
Create a new user,
overwriting the old
user's cookie, and
their session.

4

Import the first user's
session to show how a
simple tool can be used
to swap sessions.



Instructor Demonstration
Cookie-Editor



Activity: Swapping Sessions

In this activity, you'll use Cookie-Editor to examine how an attacker can hijack a session.

Suggested Time:
25 Minutes





Time's Up! Let's Review.

*The
End*