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:
A Security Perspective

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 consider how web applications work from the perspective of the user and the web developer/architect.

This week, we'll 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.

Overview of Today's Topics

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

Introduction to the Modern Web

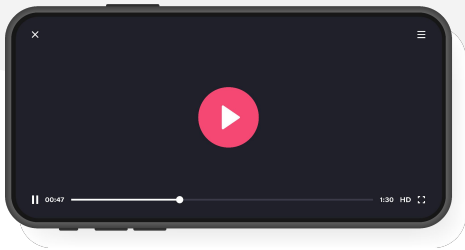
Introduction to the Modern Web

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.

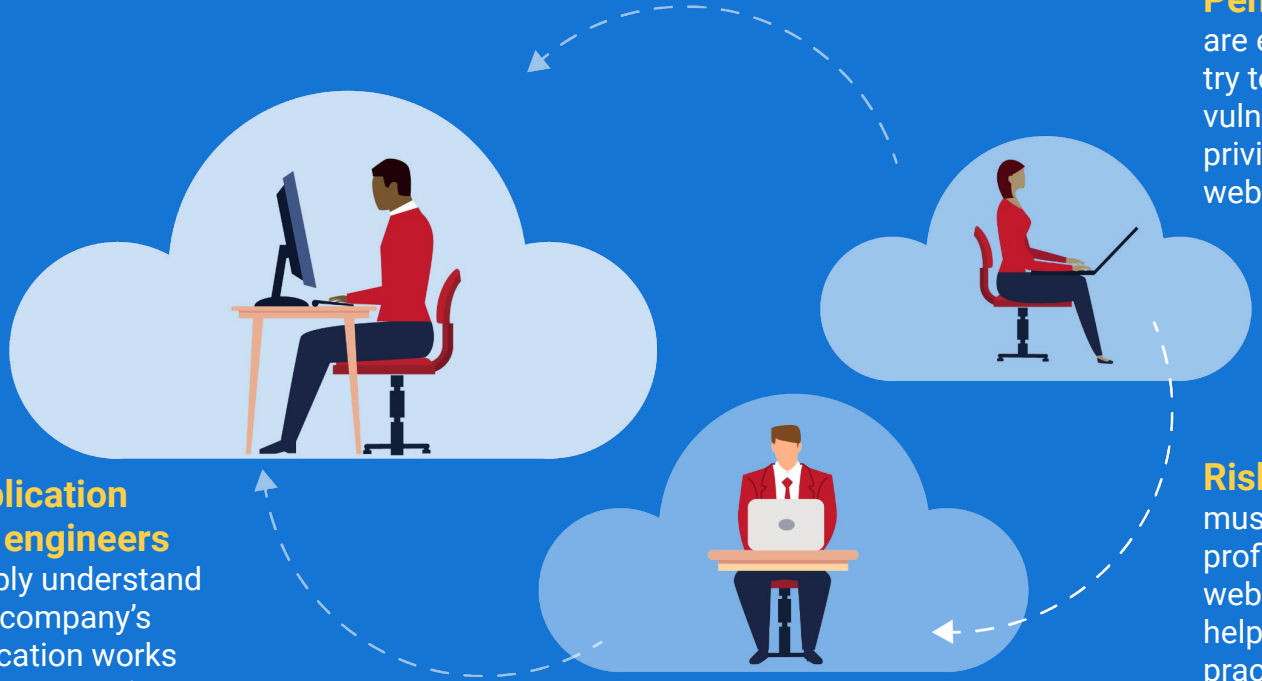
Companies are moving their on-premises infrastructure and services to the cloud—e.g., using cloud-based backup services for servers.

Most people access the web on a daily basis, whether for shopping or streaming videos.



Introduction to the Modern Web

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



Penetration testers are ethical hackers who try to find and exploit vulnerabilities and gain privileged access to web applications.

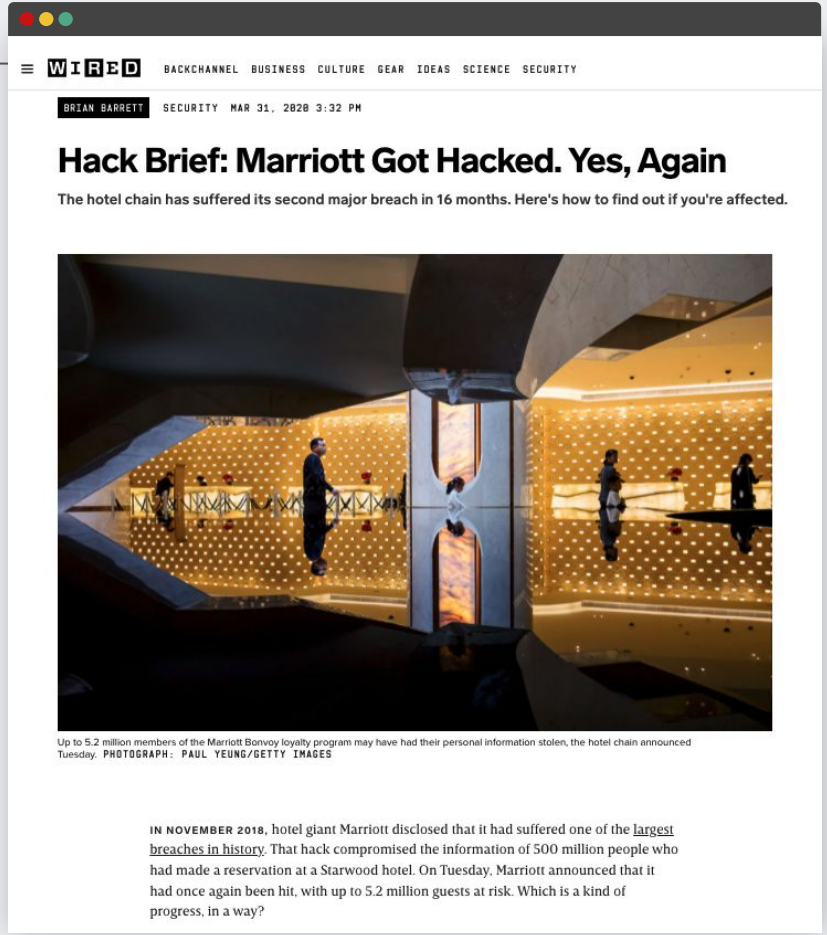
Risk assessors must understand the risk profiles and tolerances within web applications so they can help companies follow best practices and policies.

Web application security engineers must deeply understand how their company's web application works and how to secure it.

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, and other devices, is growing constantly.
- The collective amount of personal data stored on the web is growing significantly.
- Attackers can steal and sell this data.



WIRED BACKCHANNEL BUSINESS CULTURE GEAR IDEAS SCIENCE SECURITY

BRIAN BARRETT SECURITY MAR 31, 2020 3:32 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/GETTY 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?

Source: WIRED

Understanding the Web:
HTTP Requests and Responses

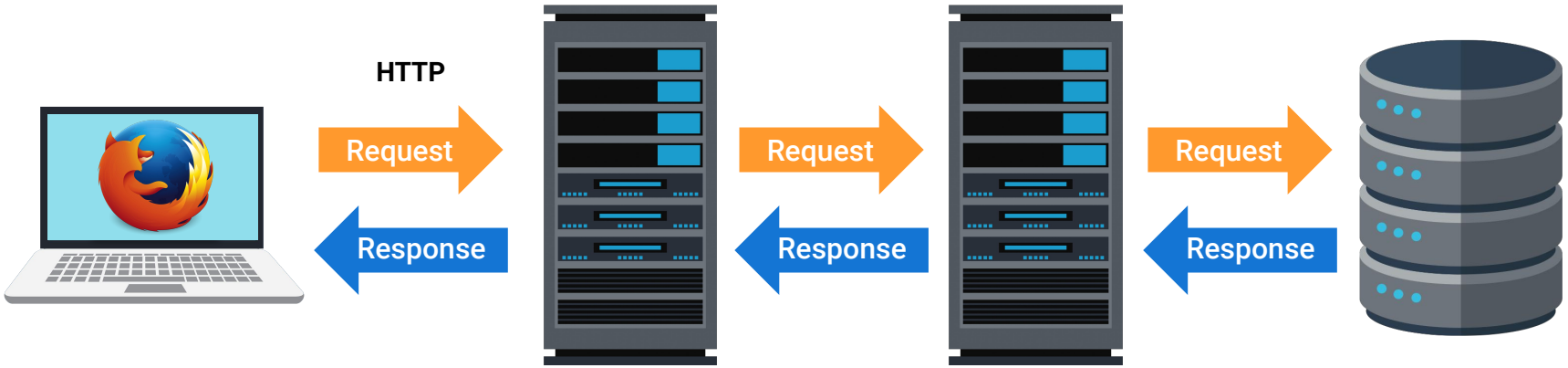
Preliminary Information

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

- Client–server architecture
- HTTP (Hypertext Transfer Protocol)
- 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.



Client

A user's device communicates with a server to request resources from it.

Web server

The server queries the resources from internal servers it's connected to.

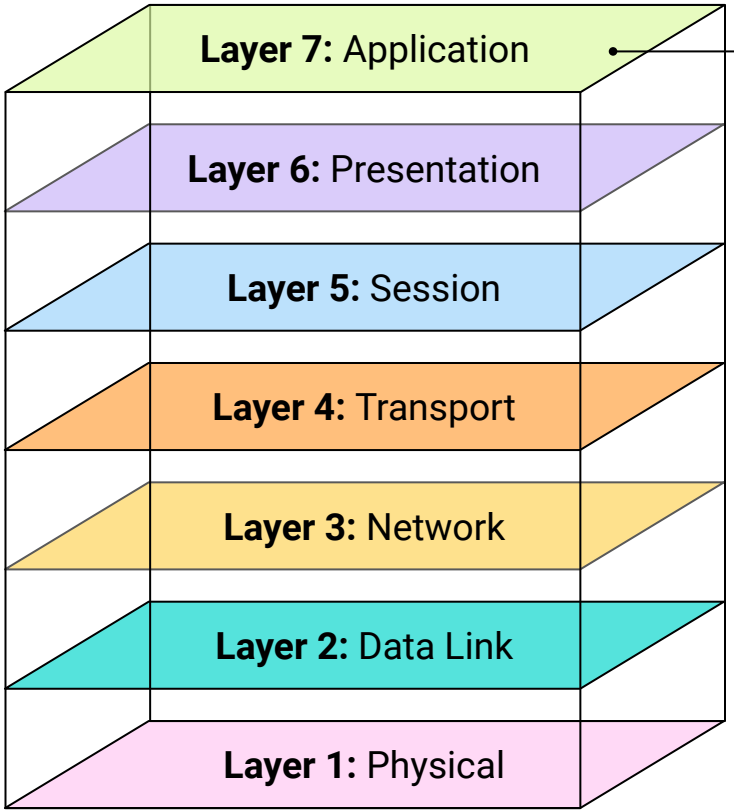
Application server

The application server sends and receives data from the database server.

Database server

The data resides within the database server.

HTTP Requests

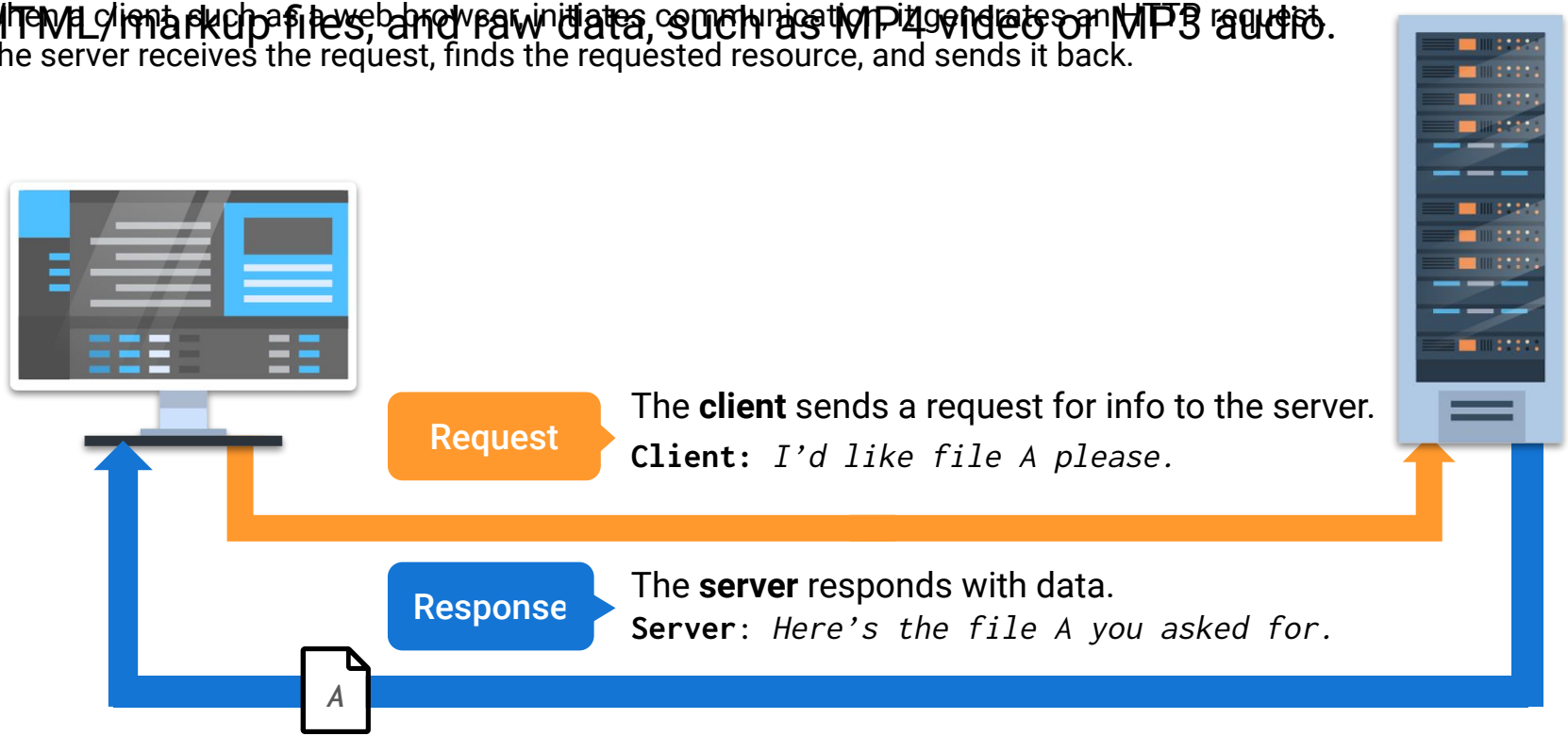


HTTP is an **OSI Layer 7** protocol that websites and web applications use to communicate with user devices.

HTTP Requests

HTTP is 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, such as a web browser, initiates communication it generates an HTTP request. The server receives the request, finds the requested resource, and sends it back.



HTTP Methods

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

A user can request data from a server.

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

A user can give data to a server.

"Here are my credentials for my LinkedIn account."

A user can update data on a server.

"I'd like to reset my Zoom password."

HTTP Request Methods

There are various HTTP request methods:

HTTP Method	Description
GET	Requests data *from* a server.
HEAD	Identical to GET, but the server does not send the response body.
POST	Sends data *to* 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 Request Methods

We'll focus on GET and POST requests.

Get

Requests data *from* 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.

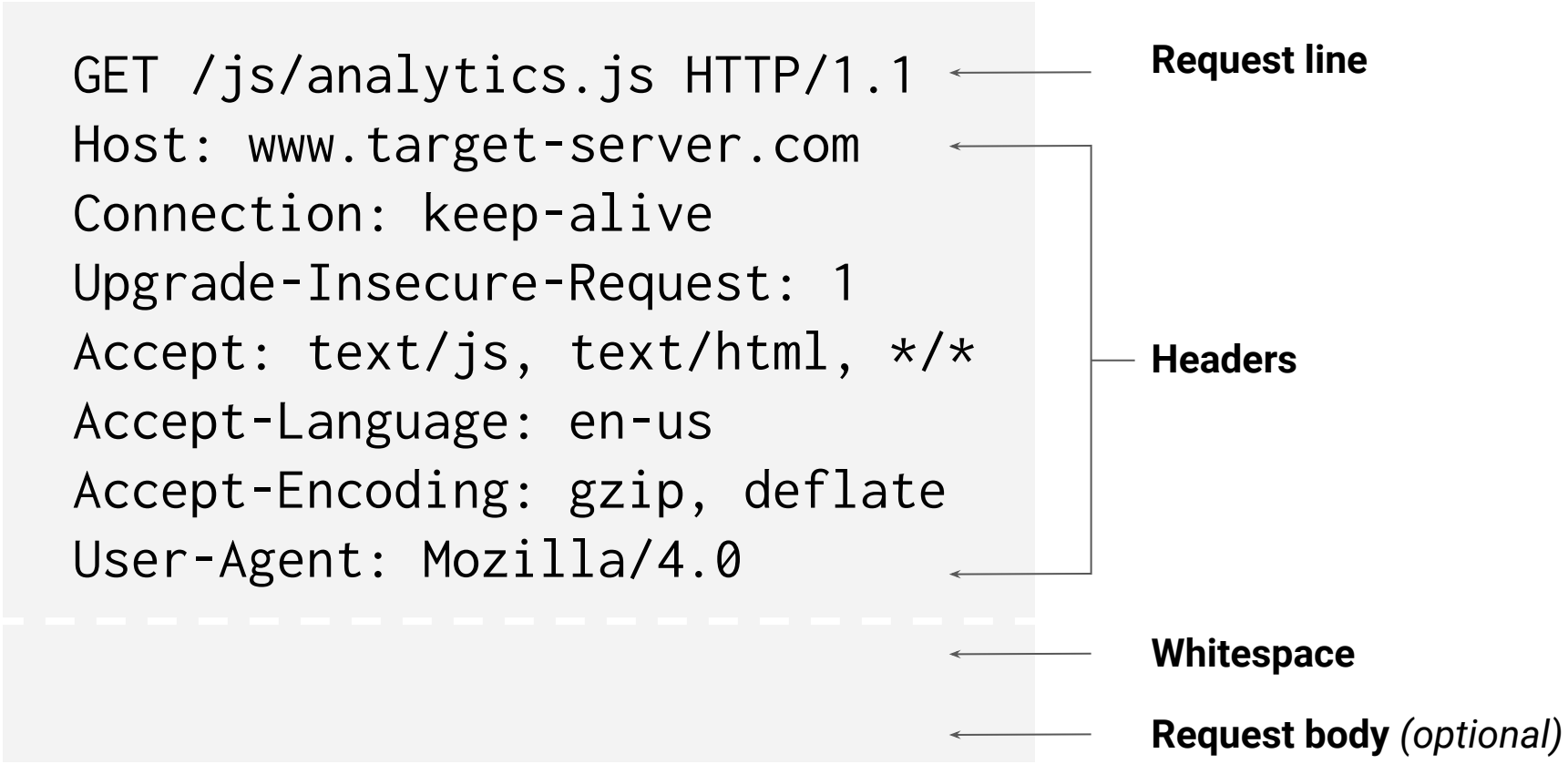
Post

Sends data *to* a source, often changing or updating a server.

Once your browser goes to amazon.com, you log in to your Amazon account. The client sends a POST request that contains your credentials for logging in.

The Anatomy of HTTP Requests and Responses

Anatomy of a GET Request



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: gzip, deflate
User-Agent: Mozilla/4.0
```

Request line

- Contains the request method, the name of the requested resource, and the version of HTTP in use.
- Can also contain **query parameters**, which the client can use to send data to the server.

Headers

Whitespace

Request body *(optional)*

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: gzip, deflate
User-Agent: Mozilla/4.0
```

Request line

Headers

- 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

Request body *(optional)*

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: gzip, deflate
User-Agent: Mozilla/4.0

```

Request line

Headers

Whitespace

- A blank line indicating the end of the request.

Request body *(optional)*

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: gzip, 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: gzip, 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: gzip, deflate
User-Agent: Mozilla/4.0
```

`Connection` instructs 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: gzip, deflate
User-Agent: Mozilla/4.0
```

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

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

`User-Agent` informs 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 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: gzip, 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
```

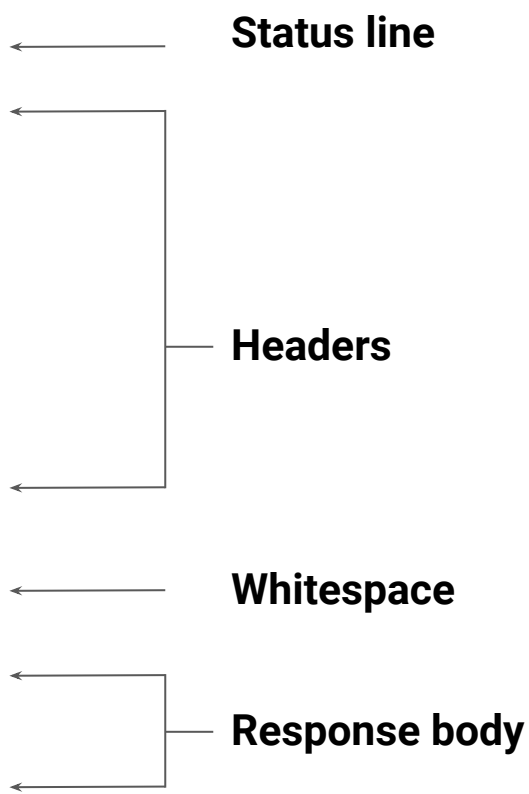
The request body may contain information such as login credentials or a file to be uploaded.

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-lubuntu4.21
Cache-Control: no-cache
Set-Cookie: SESSID=8toks; httponly
Content-Encoding: gzip
Content-Length: 698


function getStats(event) {
...
```



Anatomy of an HTTP Response

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 request headers.
Whitespace	Separates the header with the following response body.
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-lubuntu4.21
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.

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-lubuntu4.21
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.

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-lubuntu4.21
Cache-Control: no-cache
Set-Cookie: SESSID=8toks; httponly
Content-Encoding: gzip
Content-Length: 698


function getStats(event) {
...
```

`Set-Cookie` instructs 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.

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-lubuntu4.21
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.
500 codes	Indicate server errors, meaning the server application failed somehow.

You probably recognize the 404 error.

404 ERROR

Sorry, the page not found

Pro Tip:

A standard reference site for general headers and web information is Mozilla Developer Network.

Key Takeaways

- HTTP requests are sent from an HTTP client to an HTTP server.
- HTTP responses are sent back from the HTTP server as a response to the client.
- HTTP requests include a request line, request header, and optional request body.
- HTTP responses include a status line, response header, and usually a response body.
- Query parameters allow you to be specific about the parts of a resource you want to send or receive data from.
- GET requests ask an HTTP server for resources, such as a whole webpage.
- POST requests send data to an HTTP server's resources, such as login credentials or images for a webpage.
- PUT requests also send data to an HTTP server, but are often used to overwrite resources, such as updating part of a webpage.
- OPTIONS requests ask an HTTP server to respond with all HTTP methods that the HTTP server is programmed to respond to.

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.

Questions?

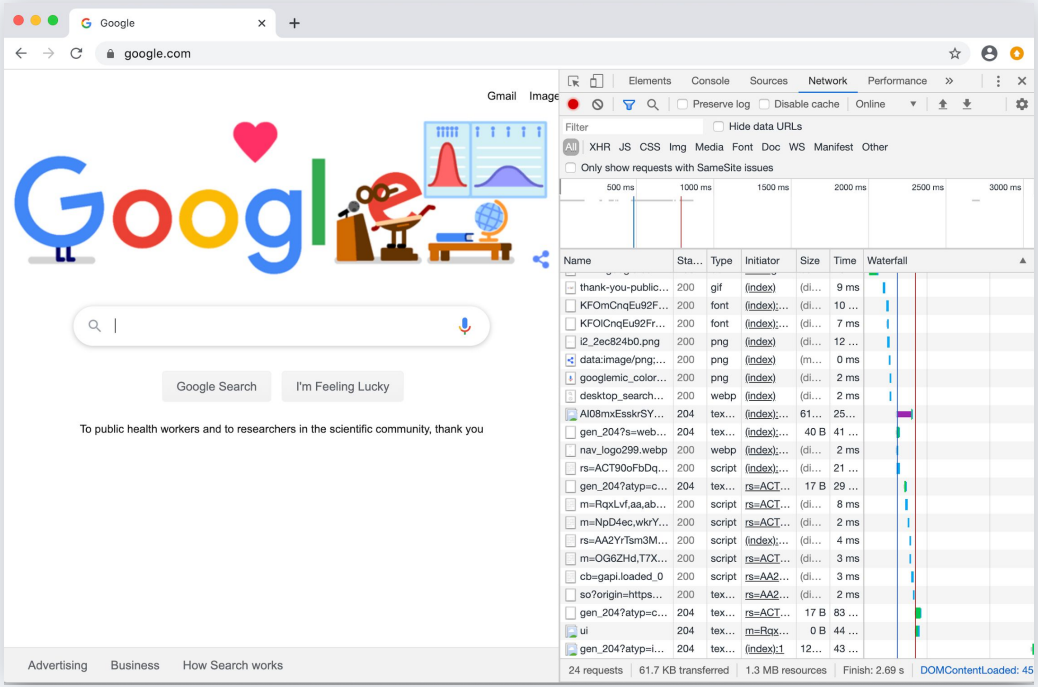
Examining HTTP and Cookies with DevTools

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.



Instructor Demonstration

Chrome DevTools

Activity: Inspect 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.

Questions?

https://www.crowdstrike.com/

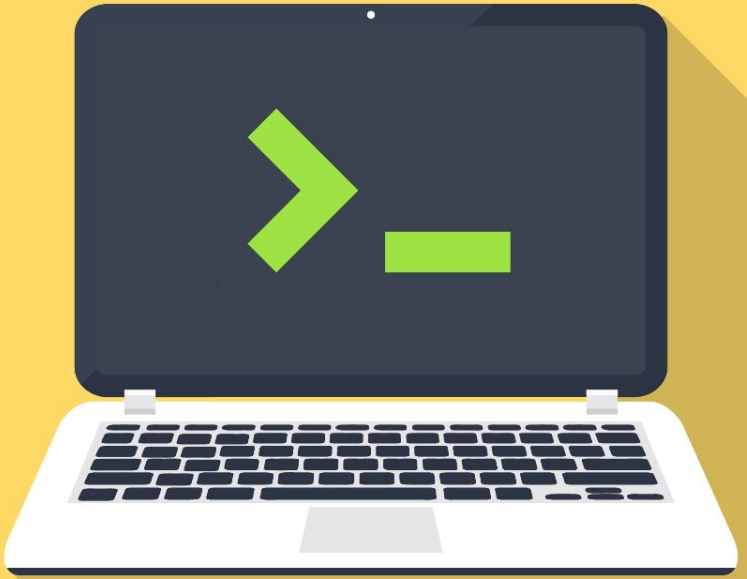
Countdown timer

15:00

(with alarm)

Use curl

It’s not always possible to examine HTTP requests and responses through a browser. Sometimes the tools you can use to send and receive HTTP requests are limited.



Introducing `curl`

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.

The command-line tool **`curl`** allows security professionals to quickly test HTTP requests in a way that can be automated while allowing them to make adjustments.

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

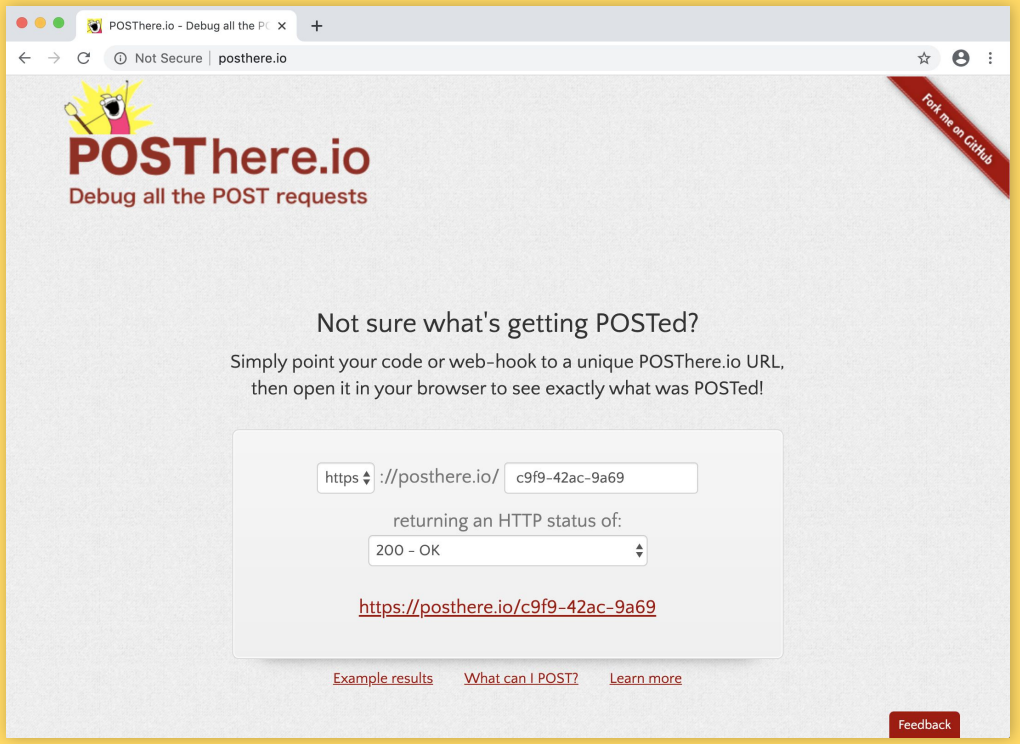
`curl`

In a security context, we use `curl` to send customized HTTP requests that allow 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 request types.
- Look for vulnerabilities on a web server.

`curl`

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



Instructor Demonstration

`curling`

Activity: Use `curl`

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

Suggested Time:

15 Minutes

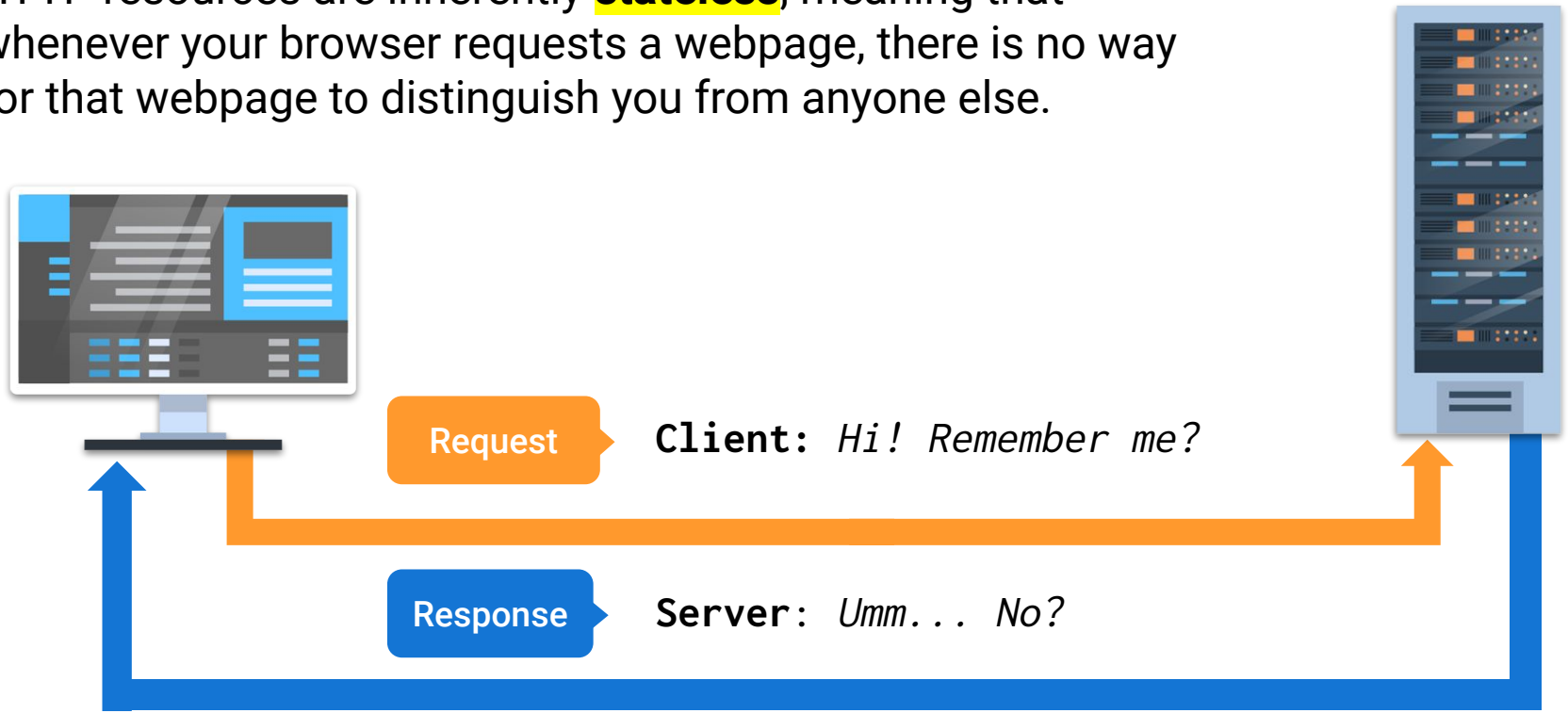
Time's Up! Let's Review.

Questions?

Sessions and Cookies

Sessions and Cookies

HTTP resources are inherently **stateless**, meaning that whenever your browser requests a webpage, there is no way for that webpage to distinguish you from anyone else.



Sessions and Cookies

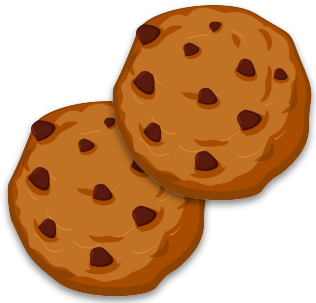
Websites need a way to deliver content that is specific to each user. To do so, they establish sessions with cookies.

Sessions

Unique server-side sets of user data used to customize webpages for the specific user accessing them.

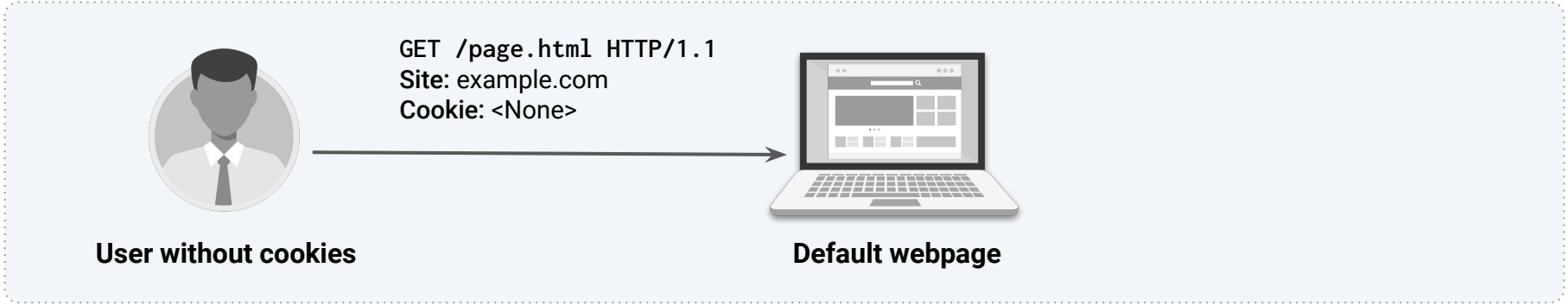
Cookies

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

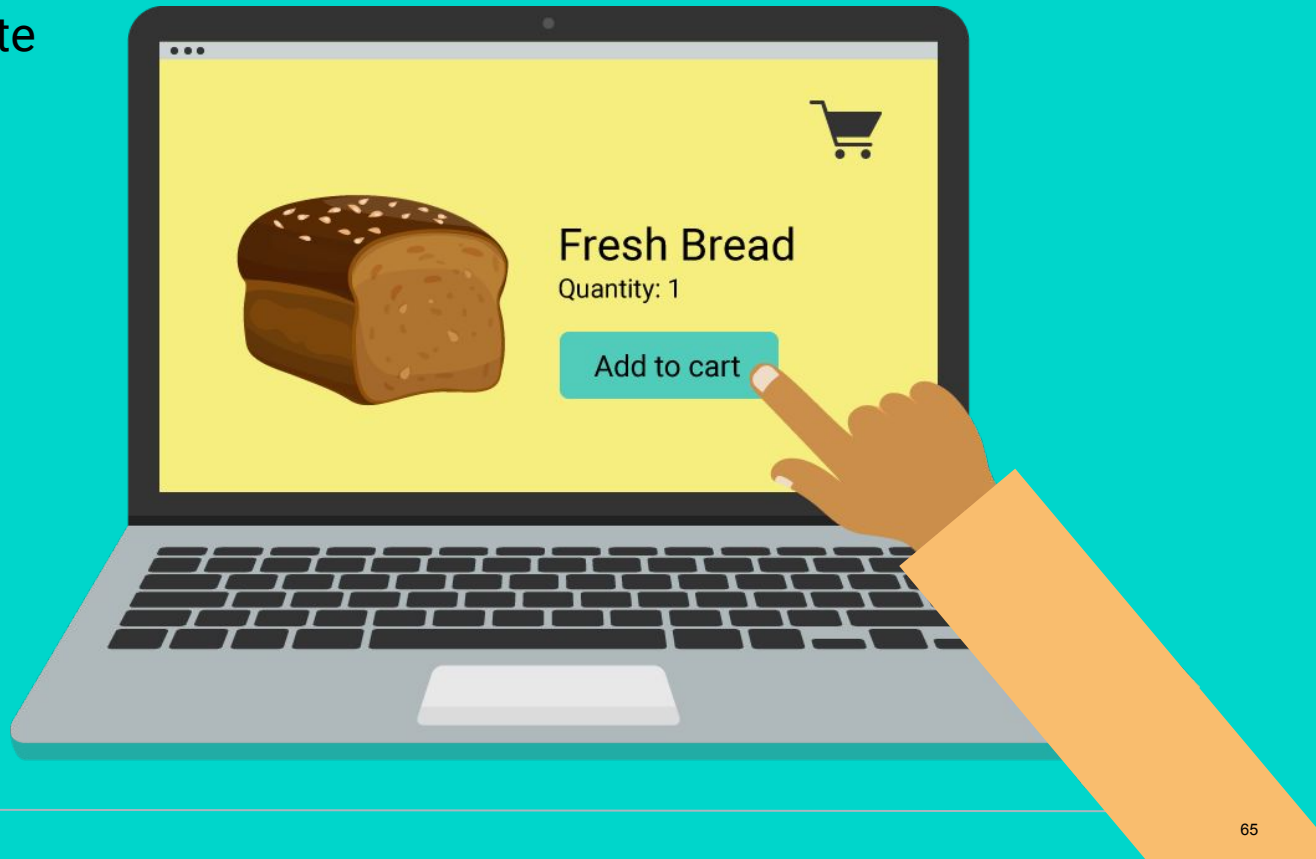
You visit a shopping website and register an account with your first and last name. As soon as you do this, a session is created that saves your name to the site's database.



How a Session Is Created

Before the session:

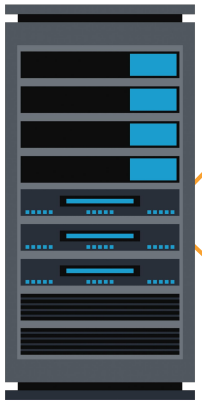
Bob visits a shopping site and clicks a button to add one loaf of bread to his shopping cart.



How a Session Is Created

Post request

The button sends a **POST request** to the HTTP server with Bob's one cart item.

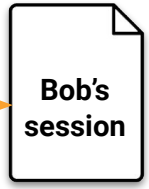


Create cookie
The HTTP server processes the POST request and creates a cookie for Bob.

Initiate session
In the database

Database

```
POST /products/ HTTP/1.1
User-Agent: Mozilla/4.0
Host:example.com
Content-Type: text/html
Connection: Keep-Alive
Accept-Encoding: gzip, deflate
addItem=Bread&count=1
```



Create session
The site's database stores the session info with Bob's shopping list and links it to the cookie.

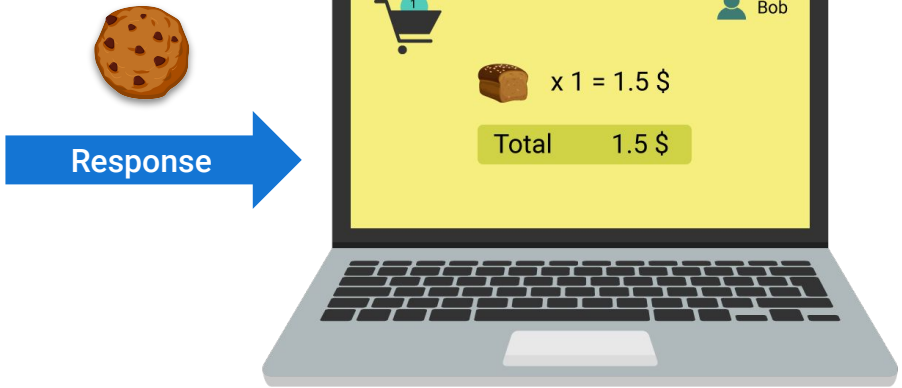
How a Session Is Created

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

A **cookie**, sent through the **response header**.

```
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>...
```

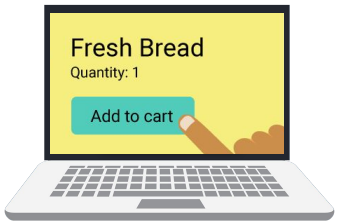
Bob's **session information**, embedded in the **response body**.



How a Session Is Created

Resuming the response:

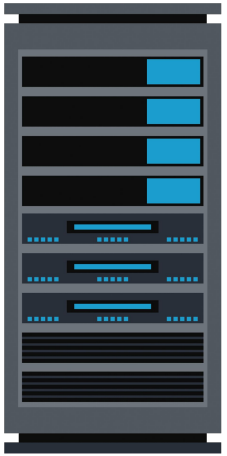
When Bob revisits the webpage, his browser sends the cookie back through the **request header**.



GET /stuff/ HTTP/ 1.1
User-Agent: Mozilla/4.0
Host: example.com
Cookie: cart=Bob
Content-Type: text/html
Connection: Keep-Alive
Accept-Encoding: gzip. deflate

Response

HTTP server



How a Session Is Created

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.

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

Session with Cookie-Editor

We'll complete the following steps:

01 Install or enable the Chrome extension Cookie-Editor.

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

03 Create a new user, which will overwrite the old user's cookie and session.

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

Instructor Demonstration

Cookie-Editor

Activity: Swap 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.

Questions?


The End