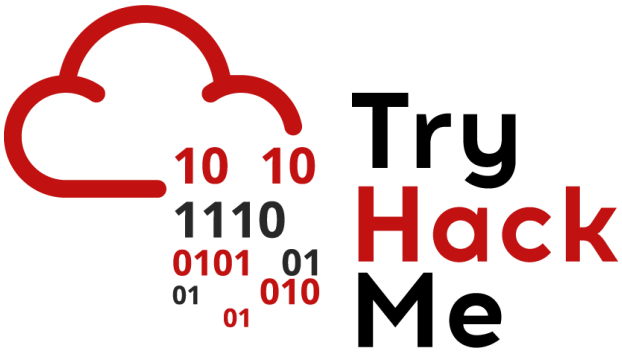



A website is usually run on what is called a web server. A web server is what's needed to essentially make the web site accessible on the wider internet (amongst other things). A computer needs a standardised way of communicating with a web server. There are so many different flavours of operating systems, browsers, and devices and we need to make sure this communicate is consistent across all of them.

*Enter HTTP.* Hyper Text Transfer Protocol(HTTP) is this standardised method we're talking about. HTTP usually works in the form of requests where a client (something like a browser) sends a request to complete a particular action to the website (technically the server). These actions can range from logging in and retrieving pages to adding some data (depending on the application of the website).

GET /login HTTP/1.1 **[1]**
Host: localhost:3000 **[2]**
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:61.0) Gecko/20100101 Firefox/61.0 **[3]**
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 **[4]**
Accept-Language: en-GB,en;q=0.5 **[5]**
Accept-Encoding: gzip, deflate **[6]**
Connection: close **[7]**
Upgrade-Insecure-Requests: 1 **[8]**

**[1]**
The first part is known as the HTTP verb that tells the server what kind of action the client is requesting. The most common types are:

- GET - used to retrieve resources, to pull a book off the shelf or open to a specific page of that book
- POST - used to change state on the server, to write something down

While this is what they are commonly used for, they can be implemented incorrectly.

The second part refers to the path the action is directed towards - in this case the client is telling the server to **get** the **login** page. The third part refers to the protocol - in this case it is version 1.1 of the HTTP protocol.

Lines **[2]-[8]** are known as HTTP Headers. Headers are used by both the client and server to pass extra information to each respective entity. Headers are usually in the form **Name: Value.** Here's a brief description of some of the request headers:

- **Host[2]:** Used to pass the domain name of the server. This is useful in a situation where a server hosts multiple web sites so the server will know which page to pass back to the browser
- **User-Agent[2]:** specifies what browser made the request. This is useful because different web pages render differently on web browsers so servers would know what exactly to server depending on what browser requested it.

When the server receives this request, it will respond to the action with an HTTP response. In the case of the following request, this is the response:

HTTP/1.1 200 OK**[1]**
X-Powered-By: Express**[2]**
Content-Type: text/html; charset=utf-8**[3]**
Content-Length: 1493**[3]**
ETag: W/"5d5-ZdJhoKmkW86HklS/Wy+dOEaa80A"**[4]**
Date: Fri, 29 Nov 2019 00:20:52 GMT**[5]**
Connection: close**[6]**

<!DOCTYPE html>
<html>

**[1]**
The first part of **[1]** contains the version of the HTTP protocol that the server uses. The second part is known as a **status/response code**. Response codes are used by the server to indicate the status of a request. They are divided into the following classes:

1. **1XX** - Information Requests
2. **2XX** - Successful Requests
3. **3XX** - Redirects
4. **4XX** - Client Errors
5. **5XX** - Server Errors

**[2]-[8]** are response headers used to pass information to the client. You may notice that **[2]** is prefaced with "X-" . This is usually the format of a custom header, so anytime you see something like that, it's worth finding out more. After **[6]**, depending on whether the request is

successful, the server will usually pass in content of a page. A client can usually request various different files from a browser, but the most common ones are:

- HTML - syntax and language used to define the structure of a web page
- Javascript - language used to perform actions related to HTML
- CSS - used to add styling to web pages

Here are some other things to note about the HTTP protocol that will be useful later on:

- HTTP is stateless: the server has no way to keep track of the order of requests the client is sending
- HTTP is unencrypted and is usually used with TLS to form HTTPS, an encrypted form of HTTP which uses certificates to verify that the website really is what is claims to be (i.e. google versus goggles)
- HTTP commonly runs on port 80 while HTTPS commonly runs on port 443 - these can be changed
- There's a lot more that goes into making a connection from a client to the server and HTTP is only part of that

If HTTP can't keep state, how does the server keep track of what the client is doing, especially if they've logged in and are buying products?

*Enter Cookies*. Cookies are a key value pair in the form of *name:value* and can be used for various purposes. For now, we'll focus on the session management. Session Management refers to how the server keeps track of the actions performed by a client. Sessions are usually created with the following workflow:

- User sends username and password to the server to authenticate themselves
- Server checks if users details are correct and sets a cookie
- Every time the user performs an action, the browser sends the cookie as part of the request to the server, which then checks to cookie to ensure the user is authorised to perform a particular action

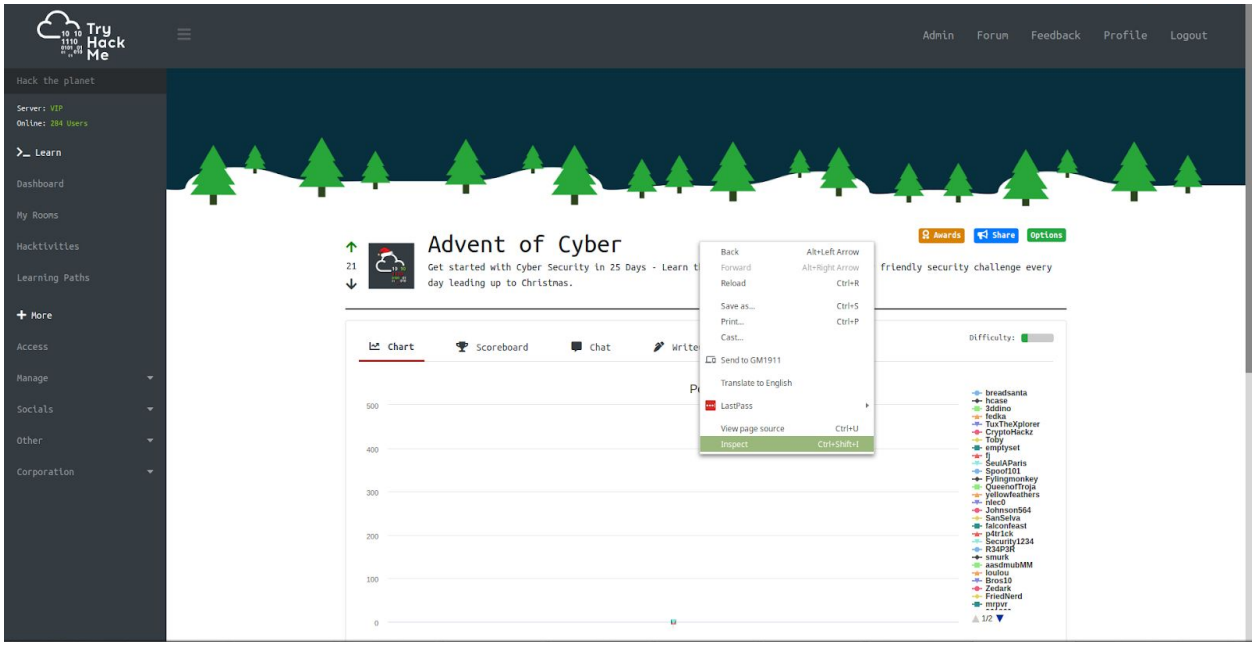
An important note to consider is that cookie values will be encoded from time to time. A browser and server may not be able to interpret particular characters so the value placed in a cookie is encoded when it's sent and decoded under the hood (at either end).
A fairly common encoding type is Base64 Encoding:

| Index | Binary | Char | Index | Binary | Char | Index | Binary | Char | Index | Binary | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 000000 | A | 16 | 010000 | Q | 32 | 100000 | g | 48 | 110000 | w |
| 1 | 000001 | B | 17 | 010001 | R | 33 | 100001 | h | 49 | 110001 | x |
| 2 | 000010 | C | 18 | 010010 | S | 34 | 100010 | i | 50 | 110010 | y |
| 3 | 000011 | D | 19 | 010011 | T | 35 | 100011 | j | 51 | 110011 | z |
| 4 | 000100 | E | 20 | 010100 | U | 36 | 100100 | k | 52 | 110100 | 0 |
| 5 | 000101 | F | 21 | 010101 | V | 37 | 100101 | l | 53 | 110101 | 1 |
| 6 | 000110 | G | 22 | 010110 | W | 38 | 100110 | m | 54 | 110110 | 2 |
| 7 | 000111 | H | 23 | 010111 | X | 39 | 100111 | n | 55 | 110111 | 3 |
| 8 | 001000 | I | 24 | 011000 | Y | 40 | 101000 | o | 56 | 111000 | 4 |
| 9 | 001001 | J | 25 | 011001 | Z | 41 | 101001 | p | 57 | 111001 | 5 |
| 10 | 001010 | K | 26 | 011010 | a | 42 | 101010 | q | 58 | 111010 | 6 |
| 11 | 001011 | L | 27 | 011011 | b | 43 | 101011 | r | 59 | 111011 | 7 |
| 12 | 001100 | M | 28 | 011100 | c | 44 | 101100 | s | 60 | 111100 | 8 |
| 13 | 001101 | N | 29 | 011101 | d | 45 | 101101 | t | 61 | 111101 | 9 |
| 14 | 001110 | O | 30 | 011110 | e | 46 | 101110 | u | 62 | 111110 | + |
| 15 | 001111 | P | 31 | 011111 | f | 47 | 101111 | v | 63 | 111111 | / |
| *padding* | | = | | | | | | | | | |

*What would an attacker do:* an attacker would first try to identify how the cookie are used for session management. Using Burp Suite, they would intercept every request mentioned in the flow above to examine how the server sets the cookie. Common hallmarks of insecure session management include:

- **Using a fixed cookie value:** If an attacker obtains the cookie value of a user and this cookie value is always the same, then the attacker can use this to gain access to a user's account. The remediation is that cookie values should be randomly regenerated whenever a user authenticates, so even if an attacker obtains this cookie value once, they wouldn't use it to gain persistent access to users' tokens.
- **Using a predictable value as part of a cookie:** if a server creates cookies that use predictable values such as username or numbers, an attacker could just set their own cookie that a server would authenticate as a different user. The remediation to this is to ensure that cookie values are completely random

While you can use Burp Suite to manipulate cookies, you can also use the inspect element features of browsers. Right click anywhere on the browser and click the inspect button (alternatively you can use the Ctrl-Shift-I combination on Windows/Linux or Cmd-Opt-I on Mac):




This will bring up the developer tools console. Once you have this open, select the Application tab on the top and click the cookies button on the left hand side. This will give you a list of all your cookies categorised per website. You can change the name and value of any cookie but double clicking it.