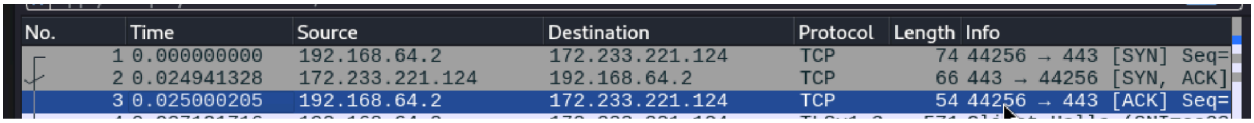
David Toledo

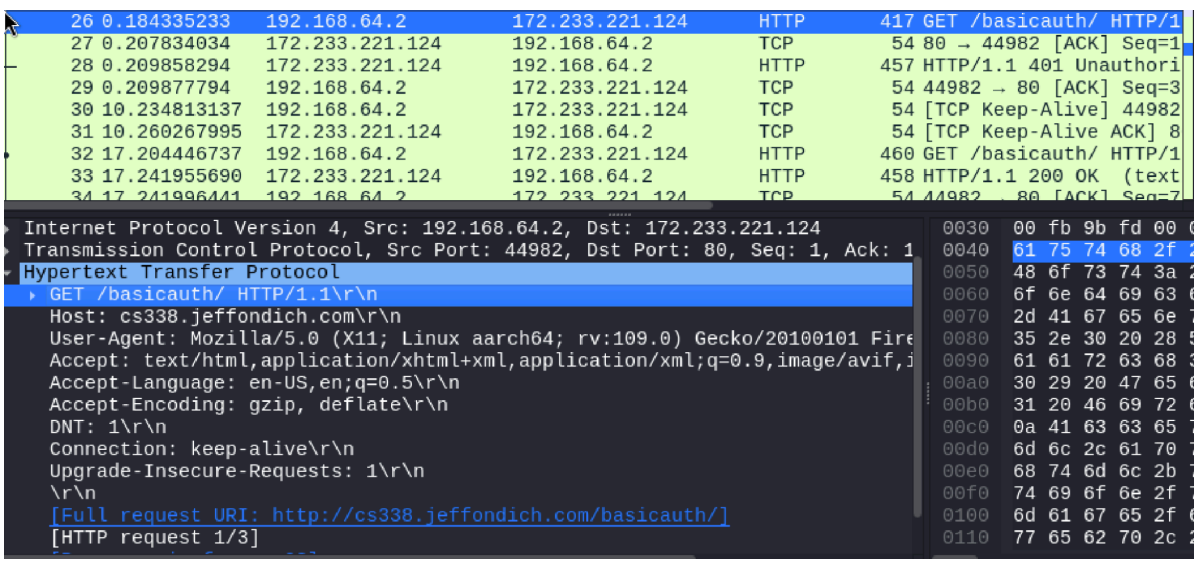
Basicauth cs338

Our conversation with cs338.jeffondich.com/basicauth/ starts with the typical TCP handshake, establishing a connection between the two servers. We can observe the [SYN], [SYN, ACK], [ACK] combination between the client and server in these 3 frames.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.64.2	172.233.221.124	TCP	74	44256 → 443 [SYN] Seq=
2	0.024941328	172.233.221.124	192.168.64.2	TCP	66	443 → 44256 [SYN, ACK]
3	0.025000205	192.168.64.2	172.233.221.124	TCP	54	44256 → 443 [ACK] Seq=

After about 23 frames later, which are filled with more TCP and TLSv1.3 requests and responses, we are met with our first HTTP protocol request. This is a GET request asking for the basicauth site of cs338.jeffondich.com. We can see in the encoded bytes that it is willing to accept many answers, the one which I believe we need is the text/html.



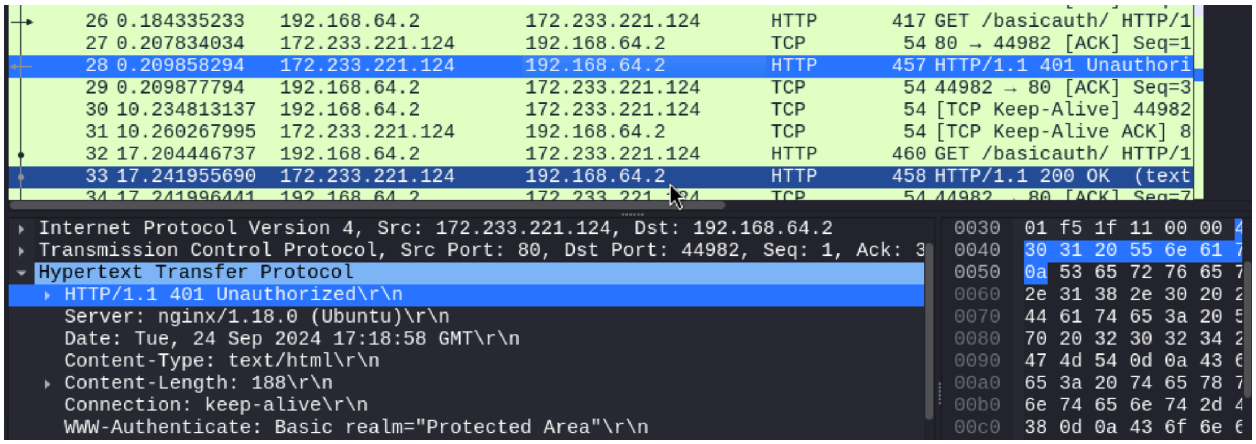
26	0.184335233	192.168.64.2	172.233.221.124	HTTP	417	GET /basicauth/ HTTP/1
27	0.207834034	172.233.221.124	192.168.64.2	TCP	54	80 → 44982 [ACK] Seq=1
28	0.209858294	172.233.221.124	192.168.64.2	HTTP	457	HTTP/1.1 401 Unauthori
29	0.209877794	192.168.64.2	172.233.221.124	TCP	54	44982 → 80 [ACK] Seq=3
30	10.234813137	192.168.64.2	172.233.221.124	TCP	54	[TCP Keep-Alive] 44982
31	10.260267995	172.233.221.124	192.168.64.2	TCP	54	[TCP Keep-Alive ACK] 8
32	17.204446737	192.168.64.2	172.233.221.124	HTTP	460	GET /basicauth/ HTTP/1
33	17.241955690	172.233.221.124	192.168.64.2	HTTP	458	HTTP/1.1 200 OK (text

```
Internet Protocol Version 4, Src: 192.168.64.2, Dst: 172.233.221.124
Transmission Control Protocol, Src Port: 44982, Dst Port: 80, Seq: 1, Ack: 1
Hypertext Transfer Protocol
  GET /basicauth/ HTTP/1.1\r\n
  Host: cs338.jeffondich.com\r\n
  User-Agent: Mozilla/5.0 (X11; Linux aarch64; rv:109.0) Gecko/20100101 Fire
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,i
  Accept-Language: en-US,en;q=0.5\r\n
  Accept-Encoding: gzip, deflate\r\n
  DNT: 1\r\n
  Connection: keep-alive\r\n
  Upgrade-Insecure-Requests: 1\r\n
  \r\n
  [Full request URI: http://cs338.jeffondich.com/basicauth/]
  [HTTP request 1/3]
```

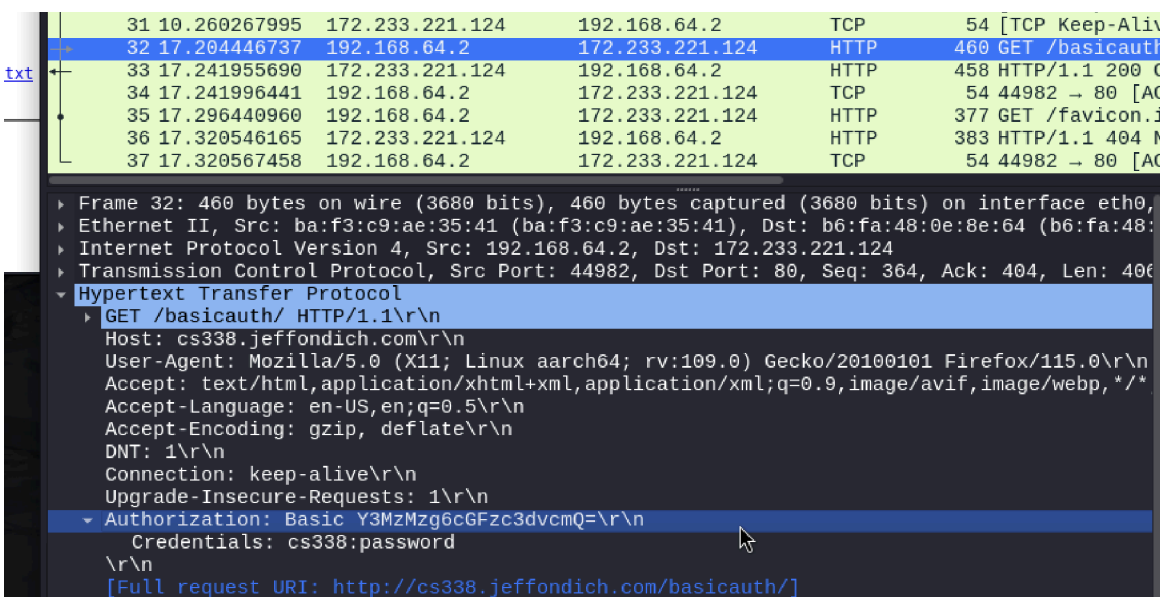
```
0030  00 fb 9b fd 00
0040  61 75 74 68 2f
0050  48 6f 73 74 3a
0060  6f 6e 64 69 63
0070  2d 41 67 65 6e
0080  35 2e 30 20 28
0090  61 61 72 63 68
00a0  30 29 20 47 65
00b0  31 20 46 69 72
00c0  0a 41 63 63 65
00d0  6d 6c 2c 61 70
00e0  68 74 6d 6c 2b
00f0  74 69 6f 6e 2f
0100  6d 61 67 65 2f
0110  77 65 62 70 2c
```

The server replies with a TCP packet and then a HTTP response, which has a status code of 401 Unauthorized. According to “Restricting Access with HTTP Basic Authentication” by NGINX DOCS ([https://docs.nginx.com/nginx/admin-guide/security-controls/configuring-http-basic-

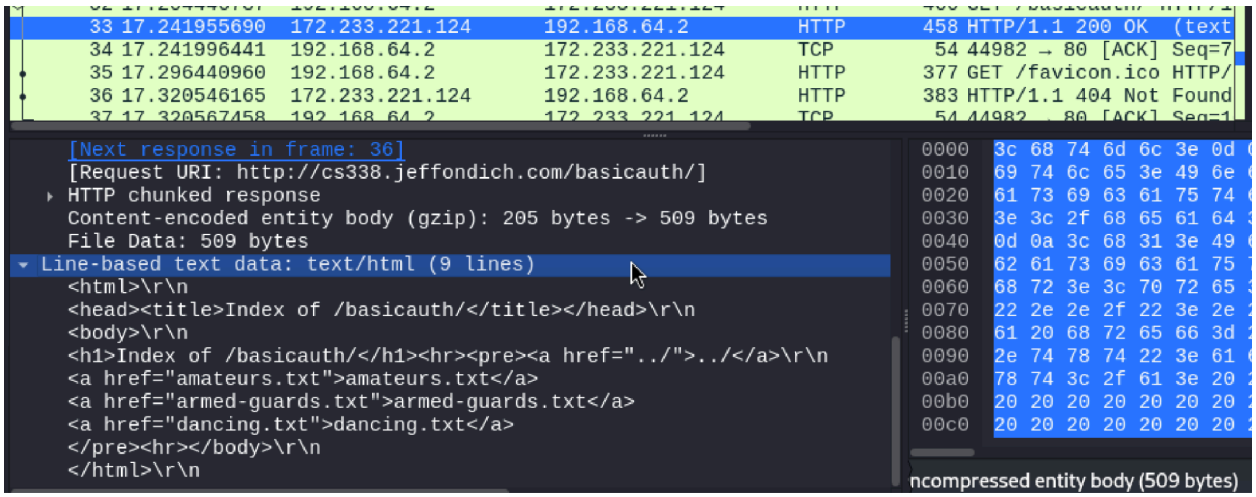
authentication/) the server would send out a 401 Unauthorized response when the “name and password do not match the password file”. Since we had just requested for the site, not having given a header with an appropriate WWW-Authenticate header, this server did not provide us with the protected area section of the website.



The website then requests us to sign in. After signing in, we see the client send out a HTTP GET request, however in this request we see the appropriate header containing the correct username/password as shown by the Wireshark decoded display of the bytes in “Credentials”.

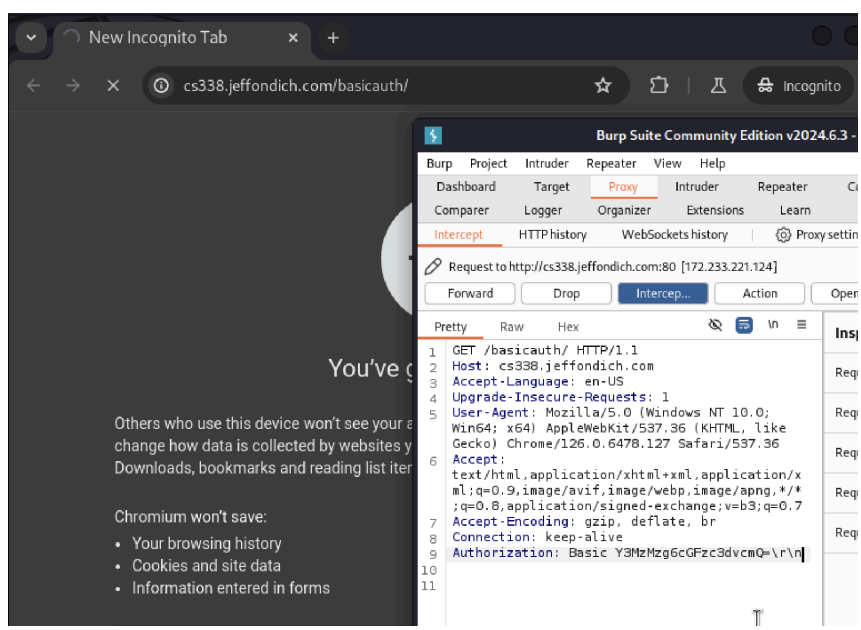


Following this GET, we see the server respond with an HTTP 20z0 OK response, as the authorization username and password has been confirmed. Hence, in this response we are sent the text/html data we were requesting.

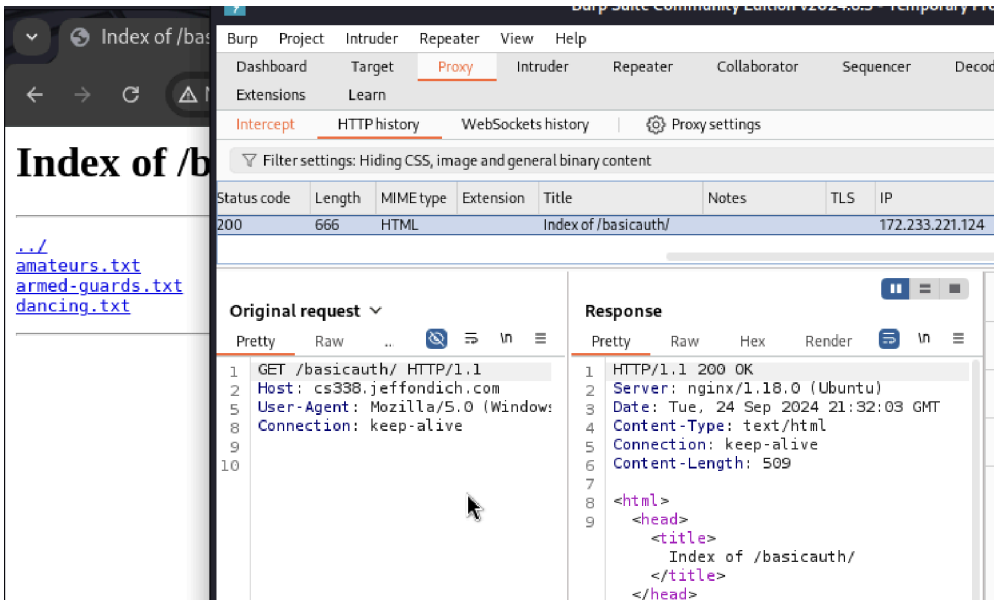


Here we see the basics of authentication and authorization being displayed. Before we tell the server we are someone with access to this webpage, we are simply told we cannot access it through the 401, and we are asked to say who we are through a username and password. By providing a correct username and password, we are then authenticated by the server and authorized to see the webpage.

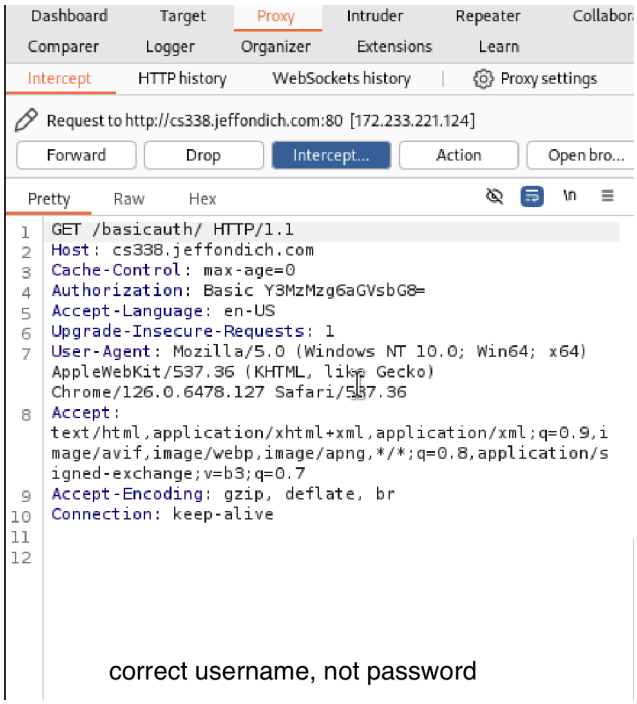
I was curious to see what would happen if I intercepted by initial request to the webpage before it asked me to login by inserting the Authorization header. In doing so, I saw that when I forwarded this, it worked.



This was the next frame that show up once I forwarded my request. I was able to see the webpage without signing in, confirmed by the 200 OK sent by the server. One unexpected issue was that Burp was only showing me the original request, and if I tried to view my edited request, burp showed me a blank slate. However, if it was just the original request, the server would have replied with a 401 as seen in Wireshark above, so I believe by edit allowed me to sign in.

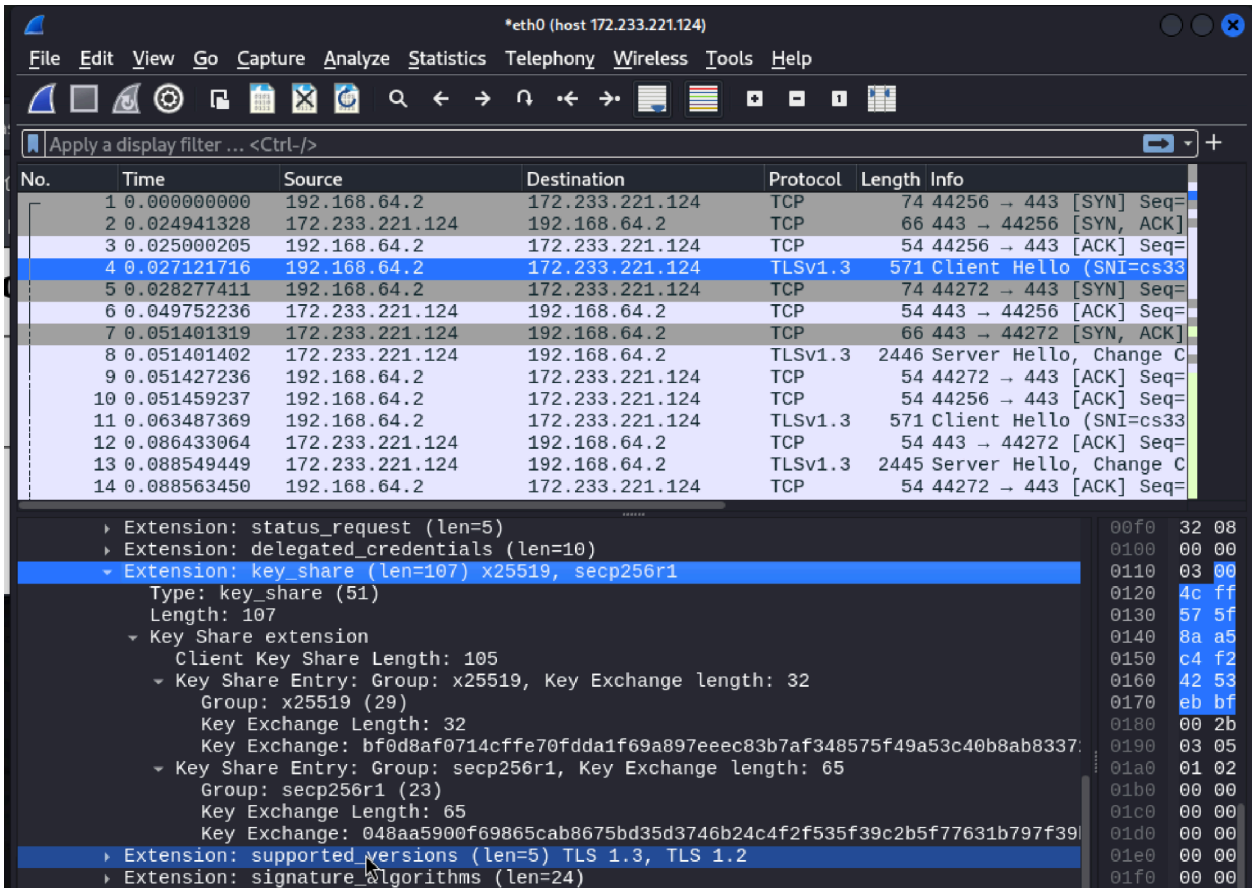


Alongside this, we see that the sent username/password is seen by the server as: Basic Y3MzMzg6cGFzc3dvcmQ=\r\n. I attempted to see how the username/password combination would look if it was wrong. Here we can see that the username/password combination is sent as Y3MzMzg6aGVsbG8=. Through this we can see that “Y3MzMzg6” is common in both the scenarios, show us that this is the encrypted version of the username “cs338”. Knowing about this encryption takes us into figuring out where it is coming from and who exactly is checking if it is correct. Since I am able to insert the Authentication



correct username, not password

myself, this shows us that the browser is sending our typed username/password to the server, and the server checks the validity of what we sent. Where the encryption is occurring, I would like to guess that it is stemming from the password file creation utility programs, and whatever algorithm was used to encrypt the original passwords in the password files created by the host of the website, as then shared to the browser in order to encrypt any attempts to log in, letting the server simply compare if there are any matches. I did find a TLS packet in wireshark which contained something containing a “key_share”, which I think could be what could be used for any encryption on the webpage, but I am not sure.



After we are logged in, we have access to the website, we are given links to click. Whenever we click on a link within the website, we send a GET request to the server for that file. There are a couple of TCP packets being sent between the server and clients which hold ACKS and a FIN, but we are also sent the file 4 frames later.