Analyzing Basic HTTP Authentication with Wireshark

1. Creating the filter.

The first step to analyzing network traffic with Wireshark is creating the appropriate filter for capturing traffic. Without a filter, the output would be flooded with irrelevant packets – I got over 150 packets in 5 seconds without actively using the browser. We'd want Wireshark to only display the packets going to and from the server hosting http://cs338.jeffondich.com/basicauth. First, find the IP address of that server using nslookup:

```
┌──(kali㉿kali)-[~]
└─$ nslookup cs231.jeffondich.com
Server:         192.168.109.2
Address:        192.168.109.2#53

Non-authoritative answer:
Name:   cs231.jeffondich.com
Address: 45.79.89.123
```

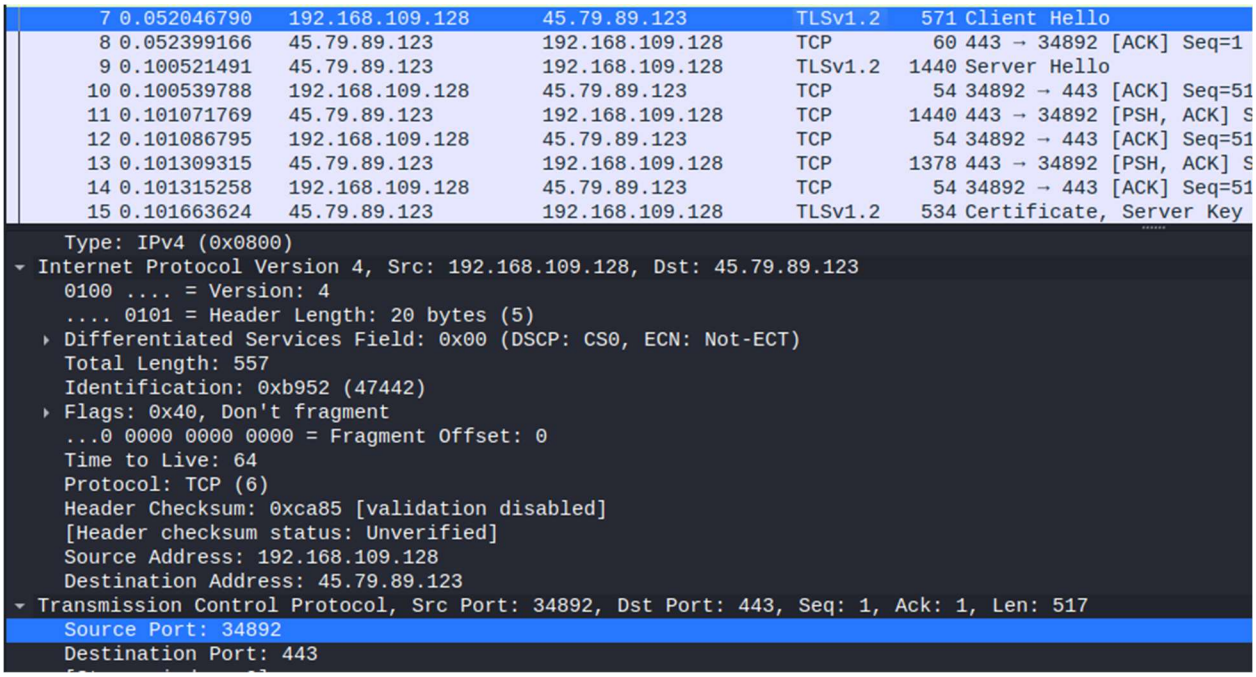
The output contains two IP addresses: 192.168.109.2 and 45.79.89.123. We want the latter address for the filter; when I tried accepting both, the first IP address mostly accepted DNS packets, which aren't relevant to our task.

2. Packets sent before credentials are entered

When we go to the website in the browser for the first time or through an incognito tab, it prompts us to enter the username and password. At the same time, several packets are exchanged between the client and the server that tell us how requests that lack the proper credentials are handled:

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000000 | 192.168.109.128 | 45.79.89.123 | TCP | 74 | 34892 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval |
| 2 | 0.000328268 | 192.168.109.128 | 45.79.89.123 | TCP | 74 | 36674 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval= |
| 3 | 0.049288520 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | 443 → 34892 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 |
| 4 | 0.049288778 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | 80 → 36674 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 |
| 5 | 0.049356042 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 34892 → 443 [ACK] Seq=1 Ack=1 Win=64240 Len=0 |
| 6 | 0.049407785 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 36674 → 80 [ACK] Seq=1 Ack=1 Win=64240 Len=0 |
| 7 | 0.052046790 | 192.168.109.128 | 45.79.89.123 | TLSv1.2 | 571 | Client Hello |
| 8 | 0.052399166 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | 443 → 34892 [ACK] Seq=1 Ack=518 Win=64240 Len=0 |
| 9 | 0.100521491 | 45.79.89.123 | 192.168.109.128 | TLSv1.2 | 1440 | Server Hello |
| 10 | 0.100539788 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 34892 → 443 [ACK] Seq=518 Ack=1387 Win=63756 Len=0 |
| 11 | 0.101071769 | 45.79.89.123 | 192.168.109.128 | TCP | 1440 | 443 → 34892 [PSH, ACK] Seq=1387 Ack=518 Win=64240 Len=1386 [TCP se |
| 12 | 0.101086795 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 34892 → 443 [ACK] Seq=518 Ack=2773 Win=63756 Len=0 |
| 13 | 0.101309315 | 45.79.89.123 | 192.168.109.128 | TCP | 1378 | 443 → 34892 [PSH, ACK] Seq=2773 Ack=518 Win=64240 Len=1324 [TCP se |
| 14 | 0.101315258 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 34892 → 443 [ACK] Seq=518 Ack=4097 Win=63756 Len=0 |
| 15 | 0.101663624 | 45.79.89.123 | 192.168.109.128 | TLSv1.2 | 534 | Certificate, Server Key Exchange, Server Hello Done |
| 16 | 0.101669894 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 34892 → 443 [ACK] Seq=518 Ack=4577 Win=63756 Len=0 |
| 17 | 0.111209950 | 192.168.109.128 | 45.79.89.123 | TLSv1.2 | 212 | Client Key Exchange, Change Cipher Spec, Encrypted Handshake Messa |
| 18 | 0.111590754 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | 443 → 34892 [ACK] Seq=4577 Ack=676 Win=64240 Len=0 |
| 19 | 0.114629513 | 192.168.109.128 | 45.79.89.123 | TLSv1.2 | 85 | Encrypted Alert |
| 20 | 0.114844349 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | 443 → 34892 [ACK] Seq=4577 Ack=707 Win=64240 Len=0 |
| 21 | 0.114885651 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 34892 → 443 [FIN, ACK] Seq=707 Ack=4577 Win=63756 Len=0 |
| 22 | 0.115149012 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | 443 → 34892 [ACK] Seq=4577 Ack=708 Win=64239 Len=0 |
| 23 | 0.117517515 | 192.168.109.128 | 45.79.89.123 | TCP | 74 | 36676 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval= |
| 24 | 0.160310550 | 45.79.89.123 | 192.168.109.128 | TLSv1.2 | 174 | Change Cipher Spec, Encrypted Handshake Message, Application Data |
| 25 | 0.160369642 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 34892 → 443 [RST] Seq=708 Win=0 Len=0 |
| 26 | 0.164731583 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | 80 → 36676 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 |
| 27 | 0.164814541 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 36676 → 80 [ACK] Seq=1 Ack=1 Win=64240 Len=0 |
| 28 | 0.165089470 | 192.168.109.128 | 45.79.89.123 | HTTP | 403 | GET /basicauth/ HTTP/1.1 |
| 29 | 0.165372354 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | 80 → 36676 [ACK] Seq=1 Ack=350 Win=64240 Len=0 |
| 30 | 0.212983725 | 45.79.89.123 | 192.168.109.128 | HTTP | 457 | HTTP/1.1 401 Unauthorized (text/html) |
| 31 | 0.213024073 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 36676 → 80 [ACK] Seq=350 Ack=404 Win=63837 Len=0 |
| 32 | 5.613723582 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 36674 → 80 [FIN, ACK] Seq=1 Ack=1 Win=64240 Len=0 |
| 33 | 5.614235773 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | 80 → 36674 [ACK] Seq=1 Ack=2 Win=64239 Len=0 |
| 34 | 5.661899708 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | 80 → 36674 [FIN, PSH, ACK] Seq=1 Ack=2 Win=64239 Len=0 |
| 35 | 5.661922916 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 36674 → 80 [ACK] Seq=2 Ack=2 Win=64240 Len=0 |
| 36 | 10.215534589 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | [TCP Keep-Alive] 36676 → 80 [ACK] Seq=349 Ack=404 Win=63837 Len=0 |
| 37 | 11.231629586 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | [TCP Keep-Alive] 36676 → 80 [ACK] Seq=349 Ack=404 Win=63837 Len=0 |

Wireshark separates the packets into distinct categories by coloring them, which helps in understanding all this data. The green packets are sent to or from port 80 of the server, which is the default port used for HTTP, and use TCP. Purple packets also use TCP, but they are sent to or from another port. In this case, the client port is 34892 and the server port is 443, as can be seen in the highlighted lines on the screenshot below. However, later in the process, other ports are used client-side, namely 36674, 34892, and 36676.




The red packet is a reset packet – the Info field indicates the [RST] flag is set to 1, which resets the communication between the client and the server.

Now that we have a general understanding of what the categories mean, let's look at several groups of frames to reconstruct the communication. The first 6 packets are two TCP handshakes – one is with the process running at port 80, another with the process at port 443. I believe the client reaches out to both ports because port 80 hosts the website itself, while port 443 has the authorization process; we'll see more evidence for this later on. Packets 7-22 and 24 initialize the authentication and encryption service created by NGINX, the server used to host this website.

The client and the server exchange hello's, as well as encryption information. In packet 17, they create a certificate and exchange public keys. The public key is for an algorithm used by the server called Diffie-Hellman Key Exchange Algorithm. This algorithm allows both the client and the server to create a key that can be used for decrypting messages encrypted with the public key in the future (https://www.educba.com/diffie-hellman-key-exchange-algorithm/). In the next several packets exchanged between the client and the encryption service, they exchange encrypted handshakes and the security specs change twice. However, only the public and private keys change, and the algorithm used remains the same.

After finishing its communication with the encryption service, the client sends a reset request to the server. This packet showed up several times in my experiments, and it's not clear why it happens. My

current understanding is that the communication is reset to simplify future communications since a handshake was already performed and security keys were generated.

Packets 23 and 26-35 show the process of trying to access a password-protected website:

| | | | | | | |
|---|---|---|---|---|---|---|
| 23 | 0.117517515 | 192.168.109.128 | 45.79.89.123 | TCP | 74 | 36676 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=169249337 TSecr=0 WS=128 |
| 24 | 0.160310550 | 45.79.89.123 | 192.168.109.128 | TLSv1.2 | 174 | Change Cipher Spec, Encrypted Handshake Message, Application Data |
| 25 | 0.160369642 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 34892 → 443 [RST] Seq=708 Win=0 Len=0 |
| 26 | 0.164731583 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | 80 → 36676 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 |
| 27 | 0.164814541 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 36676 → 80 [ACK] Seq=1 Ack=1 Win=64240 Len=0 |
| 28 | 0.165089470 | 192.168.109.128 | 45.79.89.123 | HTTP | 403 | GET /basicauth/ HTTP/1.1 |
| 29 | 0.165372354 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | 80 → 36676 [ACK] Seq=1 Ack=350 Win=64240 Len=0 |
| 30 | 0.212983725 | 45.79.89.123 | 192.168.109.128 | HTTP | 457 | HTTP/1.1 401 Unauthorized (text/html) |
| 31 | 0.213024073 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 36676 → 80 [ACK] Seq=350 Ack=404 Win=63837 Len=0 |
| 32 | 5.613723582 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 36674 → 80 [FIN, ACK] Seq=1 Ack=1 Win=64240 Len=0 |
| 33 | 5.614235773 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | 80 → 36674 [ACK] Seq=1 Ack=2 Win=64239 Len=0 |
| 34 | 5.661899708 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | 80 → 36674 [FIN, PSH, ACK] Seq=1 Ack=2 Win=64239 Len=0 |
| 35 | 5.661922916 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 36674 → 80 [ACK] Seq=2 Ack=2 Win=64240 Len=0 |
| 36 | 10.215534589 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | [TCP Keep-Alive] 36676 → 80 [ACK] Seq=349 Ack=404 Win=63837 Len=0 |
| 37 | 11.231629586 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | [TCP Keep-Alive] 36676 → 80 [ACK] Seq=349 Ack=404 Win=63837 Len=0 |
| 38 | 11.231992369 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | [TCP Keep-Alive ACK] 80 → 36676 [ACK] Seq=404 Ack=350 Win=64240 Len=0 |
| 39 | 21.435923821 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | [TCP Keep-Alive] 36676 → 80 [ACK] Seq=349 Ack=404 Win=63837 Len=0 |
| 40 | 21.436225200 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | [TCP Keep-Alive ACK] 80 → 36676 [ACK] Seq=404 Ack=350 Win=64240 Len=0 |

We can see another TCP handshake between the client and the server (23, 26, 27), followed by a GET request for the HTML of the page (28). However, since the username and password haven't been entered yet, the client gets a 401 Unauthorized error back (30). After the error has been acknowledged (31-35), the client sends the sever several Keep-Alive packets while it's waiting for the user the enter the credentials. These packets check that a connection to the server still exists. Out of curiosity, I ran Wireshark without entering the credentials, and I found that after a couple of minutes or about 8 Keep-Alive packets, the connection is seemingly terminated with a [FIN] packet. However, I was still able to enter the username and password and access the data, then continue with the experiment.

3. Credentials and how they're processed

After the user enters the credentials, the client makes another GET request (41). This request contains information about the credentials, so it's approved by the server and gets a return code 200 OK (43). After that, the website is loaded successfully. Let's take a deeper look at the GET request in packet 41.

| | | | | | | |
|---|---|---|---|---|---|---|
| 41 | 27.081543929 | 192.168.109.128 | 45.79.89.123 | HTTP | 446 | GET /basicauth/ HTTP/1.1 |
| 42 | 27.081979768 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | 80 → 36676 [ACK] Seq=404 Ack=742 Win=64240 Len=0 |
| 43 | 27.130872732 | 45.79.89.123 | 192.168.109.128 | HTTP | 458 | HTTP/1.1 200 OK (text/html) |
| 44 | 27.130891579 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 36676 → 80 [ACK] Seq=742 Ack=808 Win=63837 Len=0 |
| 45 | 27.252298240 | 192.168.109.128 | 45.79.89.123 | HTTP | 363 | GET /favicon.ico HTTP/1.1 |
| 46 | 27.252804155 | 45.79.89.123 | 192.168.109.128 | TCP | 60 | 80 → 36676 [ACK] Seq=808 Ack=1051 Win=64240 Len=0 |
| 47 | 27.300317706 | 45.79.89.123 | 192.168.109.128 | HTTP | 383 | HTTP/1.1 404 Not Found (text/html) |
| 48 | 27.300334787 | 192.168.109.128 | 45.79.89.123 | TCP | 54 | 36676 → 80 [ACK] Seq=1051 Ack=1137 Win=63837 Len=0 |

The only difference between the first GET request and this request is that the latter has the Authorization header, which contains the username and password. We learn several things from analyzing this packet. First, the credentials are sent to the server to be checked. This makes sense from a security standpoint – if the user could verify a password client-side, they would need to have access to the database storing hash values of the passwords, which is a huge security risk. Second, we see that HTTP doesn't encrypt the data it sends over the web at all, so Wireshark shows us the username and

```
▾ Hypertext Transfer Protocol
  ▸ GET /basicauth/ HTTP/1.1\r\n
    Host: cs338.jeffondich.com\r\n
    User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8\r\n
    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
  ▾ Authorization: Basic Y3MzMzg6cGFzc3dvcmQ=\r\n
      Credentials: cs338:password
```

password in the credentials field. We could also retrieve them ourselves from the security header by converting the string after “Basic” in the highlighted line from base64 to ASCII and get the same result.

The authorization header follows the format “username:password”. It should be encrypted using the keys generated as a part of the conversation with the NGINX server’s encryption service, because this experiment shows how easy it is to get this password for anybody on the same network as the client.

In summary, HTTP Basic authorization doesn’t do a proper job of keeping a website secure, since it’s possible to get the unencrypted data sent over the web with relatively simple tools like Wireshark. This also illustrates an important difference between the terms “encoding” and “encryption” discussed in class – the login credentials are encoded in base64 when they’re sent to the server, but they’re not encrypted, which makes decoding them extremely easy.