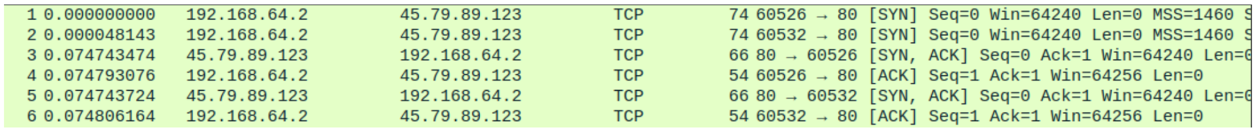
Thea Traw

HTTP’s Basic Authentication: A Story

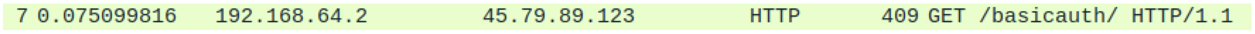
To begin, the client (the browser, 192.168.64.2) initiates two instances of the TCP 3-way handshake with the server (the website, 45.79.89.123). The client establishes a connection both with port 60526 and 60532 to server port 80 (which is HTTP).



```
  1 0.000000000   192.168.64.2          45.79.89.123          TCP        74 60526 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460
  2 0.000048143   192.168.64.2          45.79.89.123          TCP        74 60532 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460
  3 0.074743474   45.79.89.123          192.168.64.2          TCP        66 80 → 60526 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=
  4 0.074793076   192.168.64.2          45.79.89.123          TCP        54 60526 → 80 [ACK] Seq=1 Ack=1 Win=64256 Len=0
  5 0.074743724   45.79.89.123          192.168.64.2          TCP        66 80 → 60532 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=
  6 0.074806164   192.168.64.2          45.79.89.123          TCP        54 60532 → 80 [ACK] Seq=1 Ack=1 Win=64256 Len=0
```

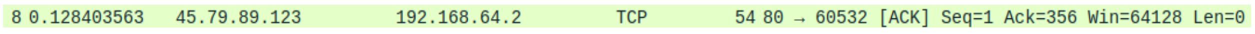
Here, we see the standard [SYN], [SYN, ACK], [ACK] interaction occur twice. For port 60526, it’s in frames 1, 3, and 4. And for port 60532, it’s in frames 2, 5, and 6.

With the TCP handshake complete, the client is now free to send a GET request to the server to ask for /basicauth/. However, there have been no credentials provided by the user and so there is no Authorization header included in the packet.



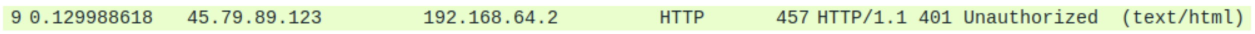
```
 7 0.075099816   192.168.64.2          45.79.89.123          HTTP       409 GET /basicauth/ HTTP/1.1
```

Then the server replies to the GET request and says, “yep, got your message.”



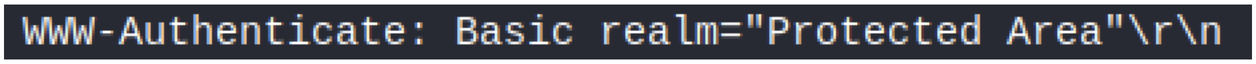
```
 8 0.128403563   45.79.89.123          192.168.64.2          TCP        54 80 → 60532 [ACK] Seq=1 Ack=356 Win=64128 Len=0
```

However, unfortunately for the client, the server follows that response with a “sorry, but you’re not allowed to have that.” A 401 Unauthorized message is sent back.



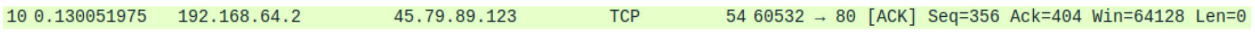
```
 9 0.129988618   45.79.89.123          192.168.64.2          HTTP       457 HTTP/1.1 401 Unauthorized  (text/html)
```

The WWW-Authenticate header tells us that the basic realm is called “Protected Area.” (Within which live the contents that we would like to see.) The client needs to authenticate itself before it can have access (that is, is authorized) to view the protection space or “realm.”



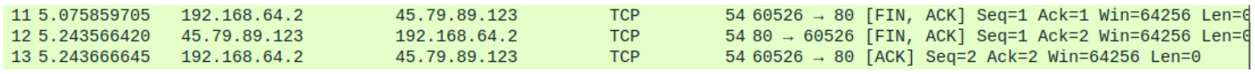
```
WWW-Authenticate: Basic realm="Protected Area"\r\n
```

With nothing else to say (yet), the client says, “okay, fine. I hear you.”



```
10 0.130051975   192.168.64.2          45.79.89.123          TCP        54 60532 → 80 [ACK] Seq=356 Ack=404 Win=64128 Len=0
```

(Aside: I notice that this acknowledgement comes from port 60532 (which also sent the original GET request) instead of its partner, port 60526. And then the latter decides to end its connection with the server anyway.

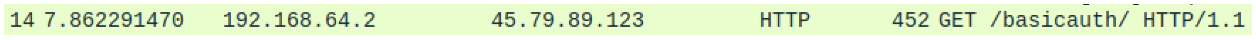


```
11 5.075859705   192.168.64.2          45.79.89.123          TCP        54 60526 → 80 [FIN, ACK] Seq=1 Ack=1 Win=64256 Len=
12 5.243566420   45.79.89.123          192.168.64.2          TCP        54 80 → 60526 [FIN, ACK] Seq=1 Ack=2 Win=64256 Len=
13 5.243666645   192.168.64.2          45.79.89.123          TCP        54 60526 → 80 [ACK] Seq=2 Ack=2 Win=64256 Len=0
```

Port 60532 now is the only one in communication with port 80. I’m not actually sure what port 60526 brought to this interaction (it never sent a GET request) and now it has closed. Perhaps it was a quirk of the browser behavior? It seems especially unnecessary given that only a single port was used by the client in other times I performed this sequence of events. And the outcome was the same with only one port as it was with two. But in any case, port 60526 is finished and only port 60532 is still in a handshake with port 80.)

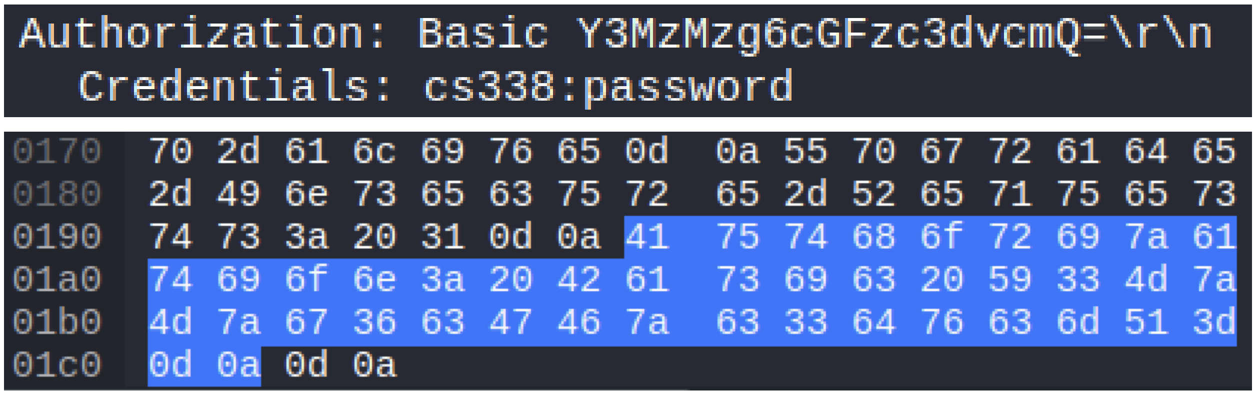
Now, the user has provided some information to the browser: its username and password. Now the client has both strings and will concatenate them like so: username:password. (This is why the colon character cannot be used in a username or password, as it has an important special meaning.) Then the client encodes the username:password into an octet sequence and then into basic-credentials using base64. So cs338:password becomes Y3MzMzg6cGFzc3dvcmQ= in base64.

Now, the client tries again to ask for /basicauth/ by sending a GET request (as it now has the authentication information).



```
14 7.862291470    192.168.64.2          45.79.89.123          HTTP       452 GET /basicauth/ HTTP/1.1
```

The packet with this request has the Authorization header. Here we can see that the credentials are sent in cleartext. (Base64 isn't an encryption scheme at all–it's just another way to represent plaintext.) Now the client is saying "here I am, with my username and password. Let me in please."



```
Authorization: Basic Y3MzMzg6cGFzc3dvcmQ=\r\n
  Credentials: cs338:password

0170  70 2d 61 6c 69 76 65 0d  0a 55 70 67 72 61 64 65
0180  2d 49 6e 73 65 63 75 72  65 2d 52 65 71 75 65 73
0190  74 73 3a 20 31 0d 0a 41  75 74 68 6f 72 69 7a 61
01a0  74 69 6f 6e 3a 20 42 61  73 69 63 20 59 33 4d 7a
01b0  4d 7a 67 36 63 47 46 7a  63 33 64 76 63 6d 51 3d
01c0  0d 0a 0d 0a
```

The highlighted blue is the hex representation of the credentials in base64.

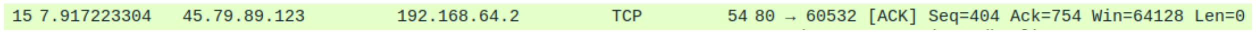
The use of base64 is to prevent the message from being corrupted / messed with as it is sent, because mostly everything will interpret text in the same way.

The browser doesn't do any checking of whether the username:password is valid. Instead, as per nginx's explanation, there is a file on the server that stores the username:password pairs and what each one is authorized for. (That is, the browser has no idea who is authorized for what; the server has all of that information. Otherwise the user could potentially do nefarious things and alter the authorization information, as the browser is under user control.)

This is a very insecure user authentication scheme. As the RFC says, this isn't considered to be secure without TLS or some other protection because the username and password are passed over the network as cleartext. Anyone with Wireshark can see it!

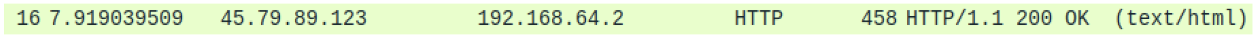
(Another aside: In my first attempt at observing this process, I encountered a bunch of noise that somehow was related to TLS. (Which seemed quite odd, given that we're just using HTTP and the "Basic" Authentication Scheme.) Unfortunately, I didn't take a screenshot, as I figured it would happen again when I replicated the steps. It did not. Anyway, I saw a bunch of “Hello Retry Request, Change Cipher Spec” messages, as well as some others that were somewhat recognizable after some investigation. When I Googled around a bit on TLS (because we haven’t talked about that in class yet), some pieces looked familiar (like “client hello” and “server hello”) but mostly it didn’t seem like the proper structure of a TLS handshake. And nothing ended up encrypted anyway–the credentials (username:password) were still cleartext, so something clearly fell through. Anyway, I thought that was strange. And this was a bit of a tangent, but I am still curious about it. (I really should have taken screenshots in case this was some kind of anomaly! And if it’s a run-of-the-mill thing, then I suppose I would like to know about it nonetheless.)

Now, the server says, “okay, I have received your request.”



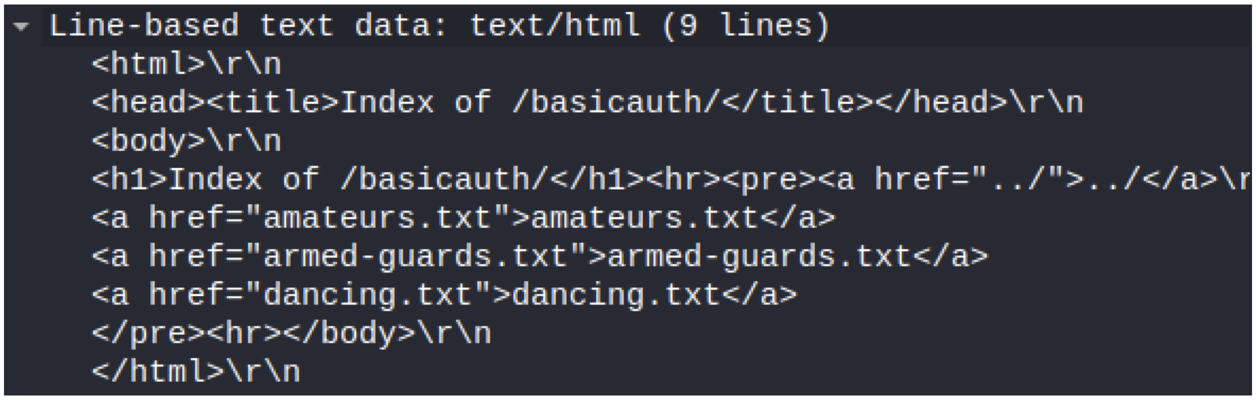
```
15 7.917223304    45.79.89.123          192.168.64.2          TCP          54 80 → 60532 [ACK] Seq=404 Ack=754 Win=64128 Len=0
```

And if the credentials are in the password file on the nginx server (which they are), then the user will be authorized to see the desired subdirectory. So the server now sends our favorite message, which says that we can see /basicauth/!



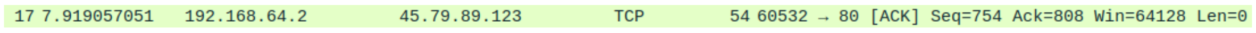
```
16 7.919039509    45.79.89.123          192.168.64.2          HTTP       458 HTTP/1.1 200 OK  (text/html)
```

So then the client is sent /basicauth/ in the packet’s payload, as per the RFC.



```
▾ Line-based text data: text/html (9 lines)
    <html>\r\n
    <head><title>Index of /basicauth/</title></head>\r\n
    <body>\r\n
    <h1>Index of /basicauth/</h1><hr><pre><a href="../">../</a>\r
    <a href="amateurs.txt">amateurs.txt</a>
    <a href="armed-guards.txt">armed-guards.txt</a>
    <a href="dancing.txt">dancing.txt</a>
    </pre><hr></body>\r\n
    </html>\r\n
```

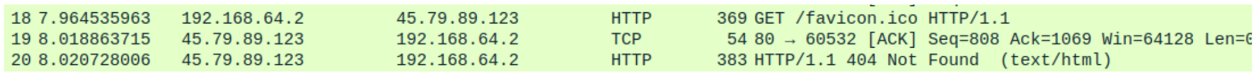
The client replies, “great, thanks for sending me this! I got it.”



```
17 7.919057051    192.168.64.2          45.79.89.123          TCP          54 60532 → 80 [ACK] Seq=754 Ack=808 Win=64128 Len=0
```

Now, the authentication and authorization process is over and a success!

The client does ask for some other things, like the favicon. But the server says, “sorry, that’s not here,” and the client says, “oh, okay.”



```
18 7.964535963    192.168.64.2          45.79.89.123          HTTP        369 GET /favicon.ico HTTP/1.1
19 8.018863715    45.79.89.123          192.168.64.2          TCP          54 80 → 60532 [ACK] Seq=808 Ack=1069 Win=64128 Len=
20 8.020728006    45.79.89.123          192.168.64.2          HTTP        383 HTTP/1.1 404 Not Found  (text/html)
```