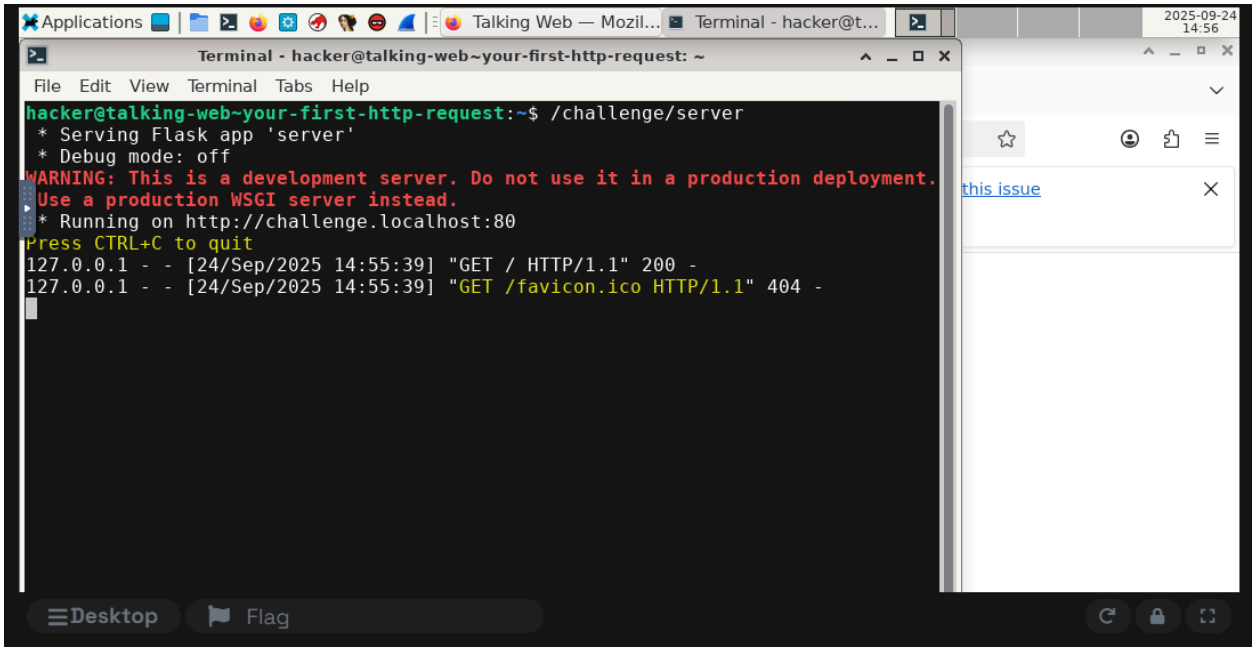
Jaleel Williamson
jayw-713
CSCI 400 Lab 8
9/29/25

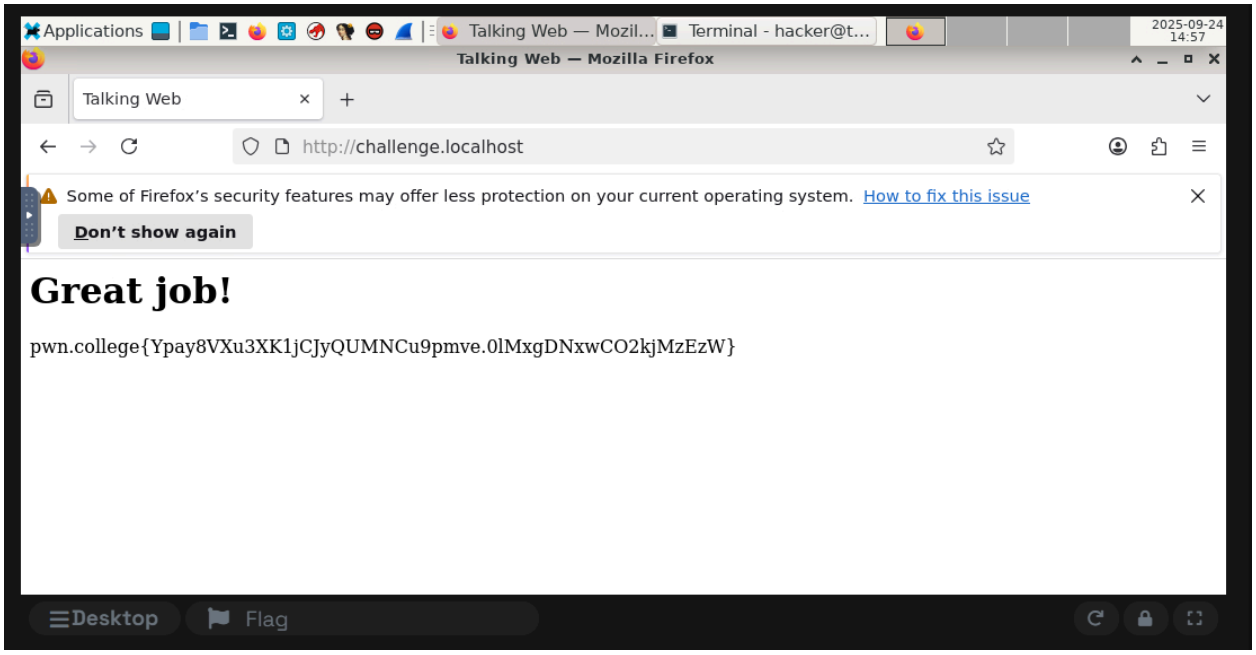
Playing with Programs: Talking Web https://pwn.college/fundamentals/talking-web/

Challenges 'Your First HTTP Request' through 'Multiple HTTP Parameters (curl)' (*15 challenges total*)

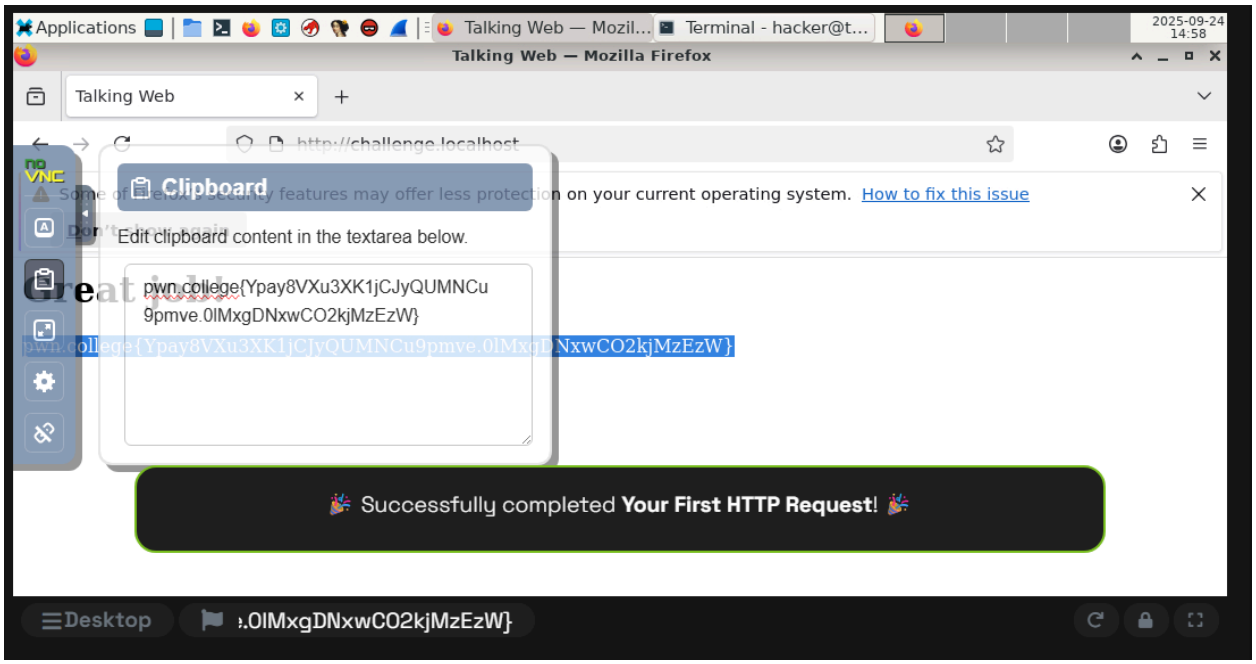
- Your First HTTP Request



Start of Challenge in GUI desktop

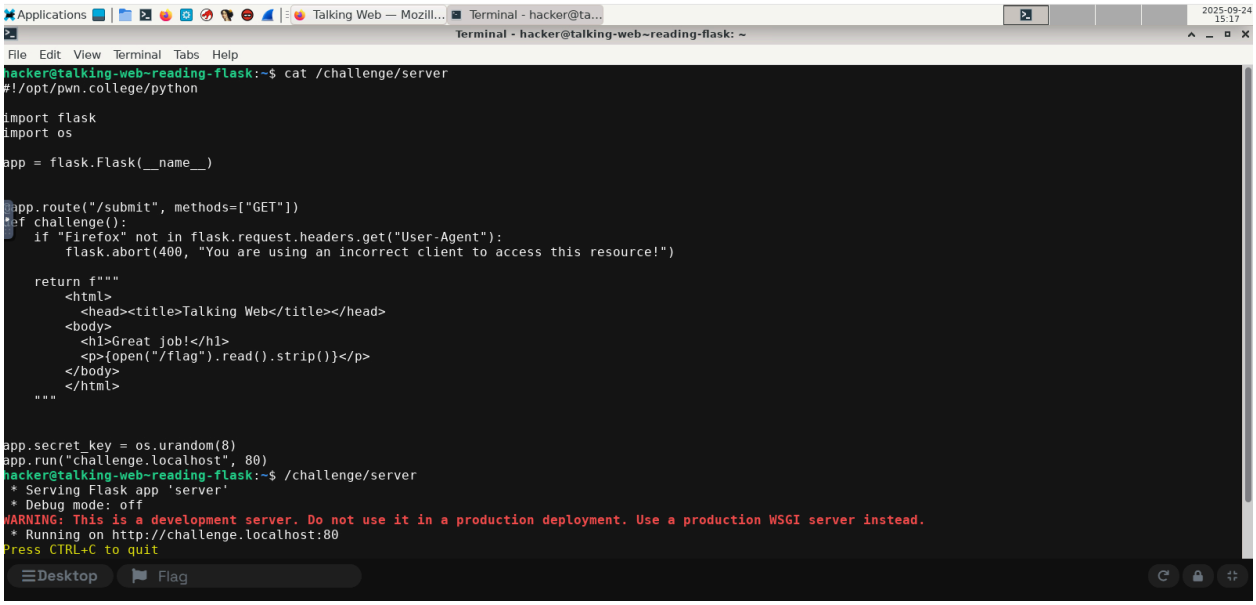


Successful Retrieval in Firefox browser

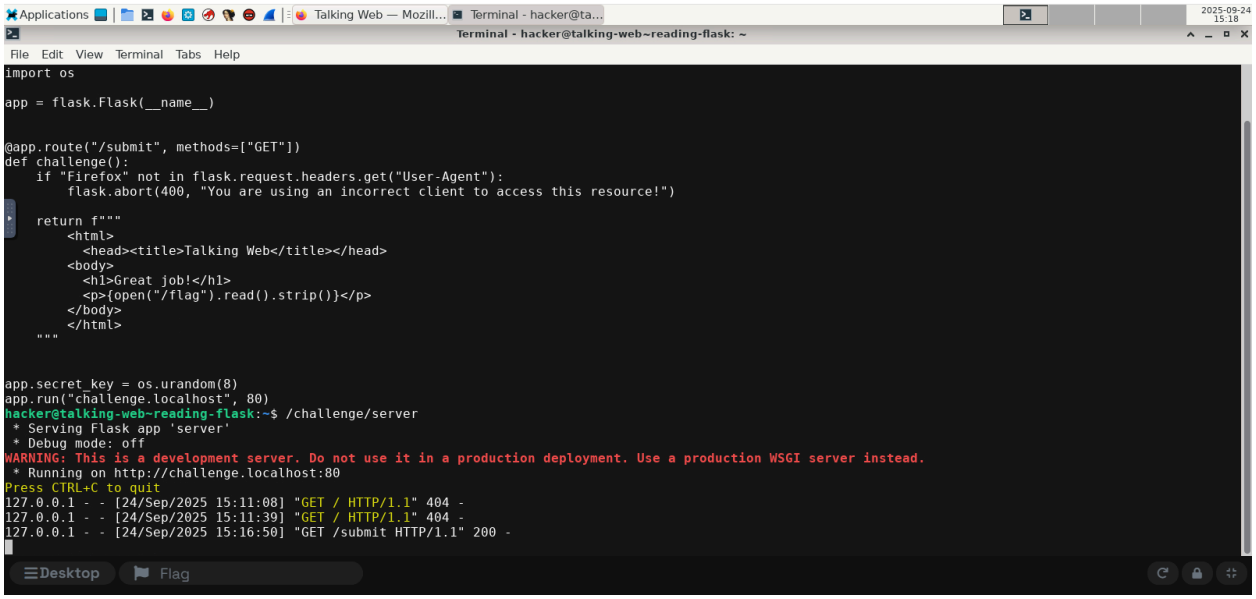


To capture the flag, I first ran the **/challenge/server** command in the terminal within the pwn.college dojo workspace, which started a **local web server** and **provided** a **URL**. Then, I launched Firefox from the **GUI Desktop** and navigated to that **URL**. The server responded with the flag, which I copied from the browser display. This completed the "Your First HTTP Request" challenge. This fundamental exercise underscored that at its core, web interaction is simply a client program (the browser) requesting a resource from a server program, establishing the basic communication model for the web.

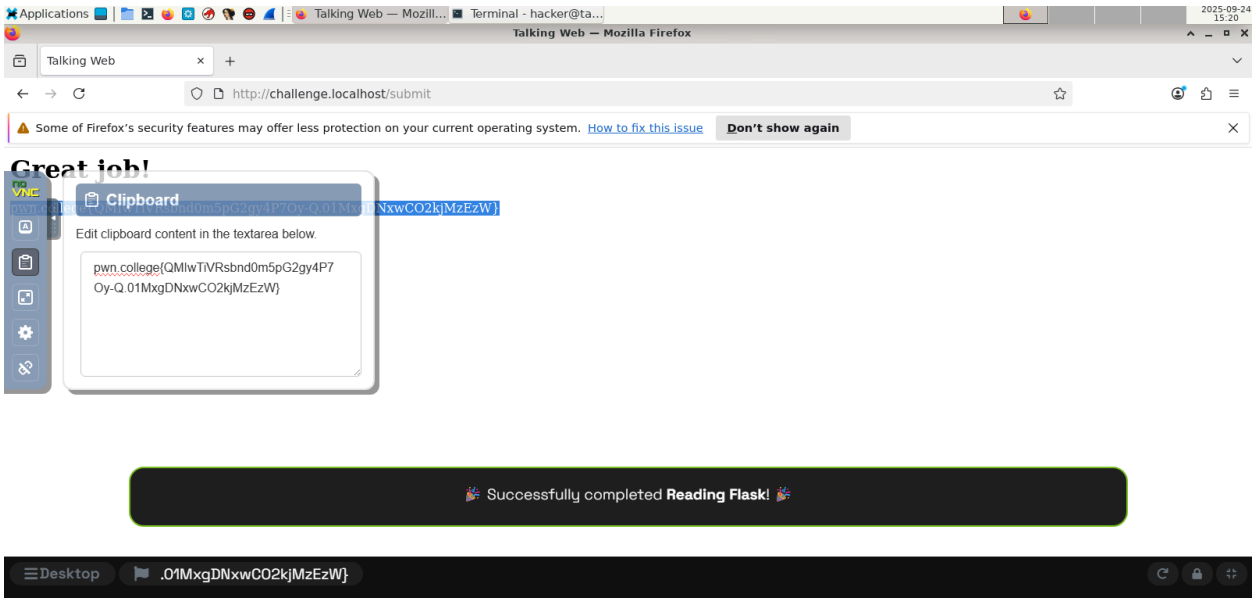
- Reading Flask



Launching the challenge in GUI desktop terminal.

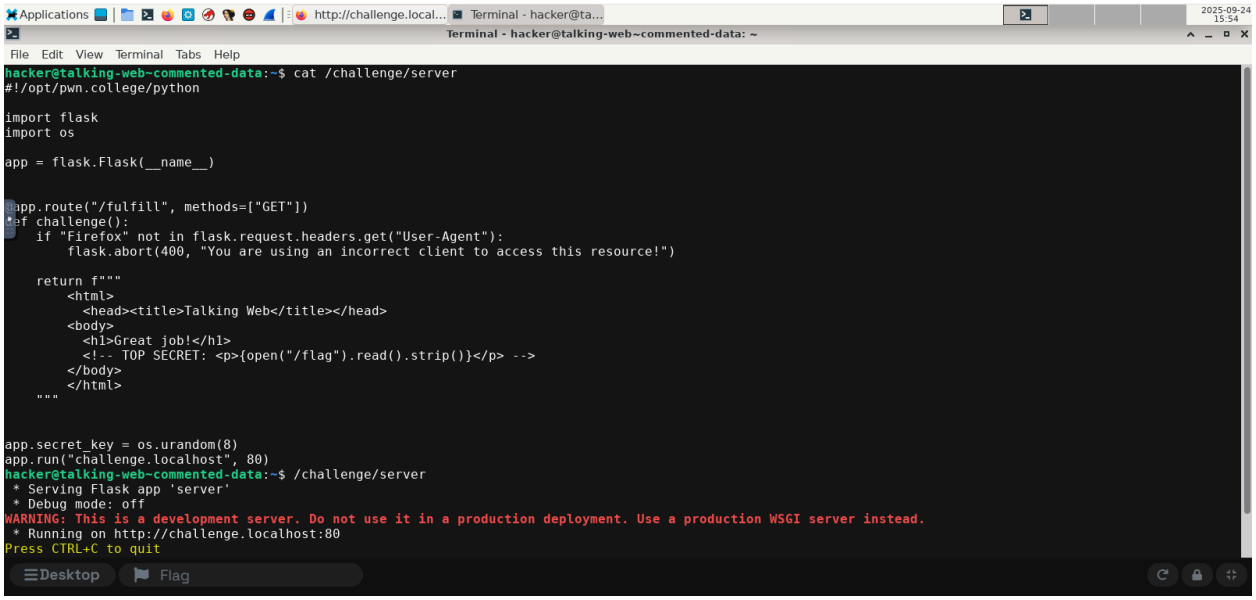


Continuation of first pic in terminal.

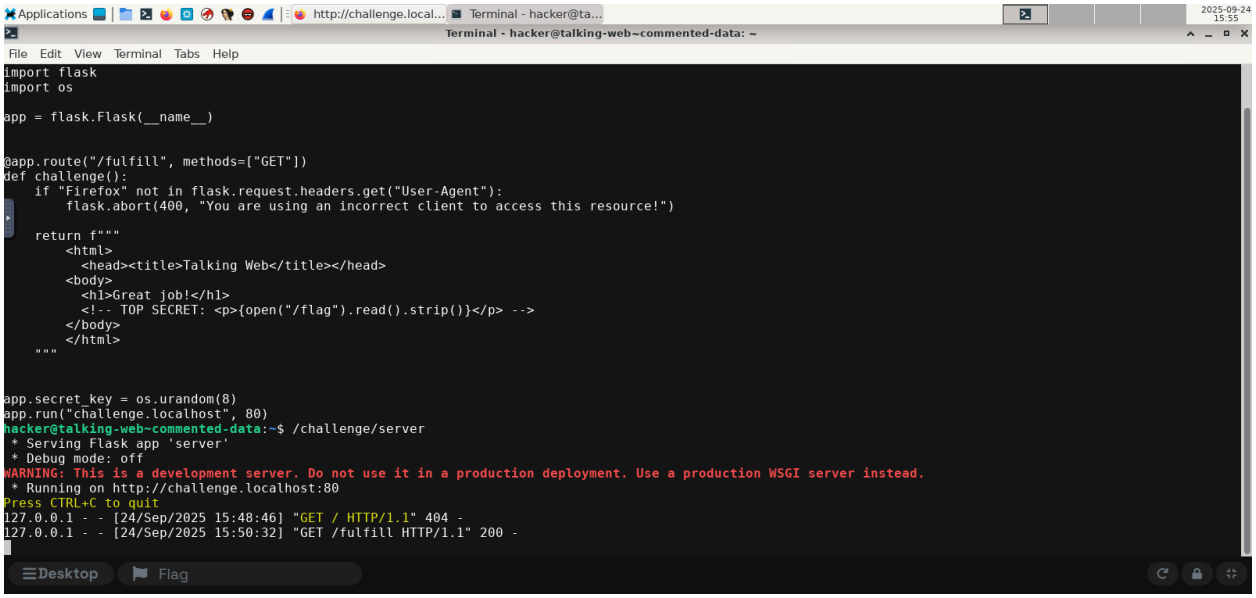


To capture the flag, I first read the source code of **/challenge/server** in the **terminal**, which revealed that the correct endpoint is **/submit** and it requires the User-Agent header to contain "**Firefox**". I then ran the server and accessed **http://challenge.localhost/submit** using Firefox from the **GUI Desktop**. The server responded with a page displaying "Great job!" and the flag: **pwn.college{QMIwTiVRsbnd0m5pG2gy4P7Oy-Q.01MxgDNxwCO2kjMzEzW}**. This demonstrated understanding of the server's routing and client requirements. The key takeaway was that server-side code defines the rules of engagement; success depends on the client understanding and adhering to these programmed constraints, such as specific endpoints and headers.

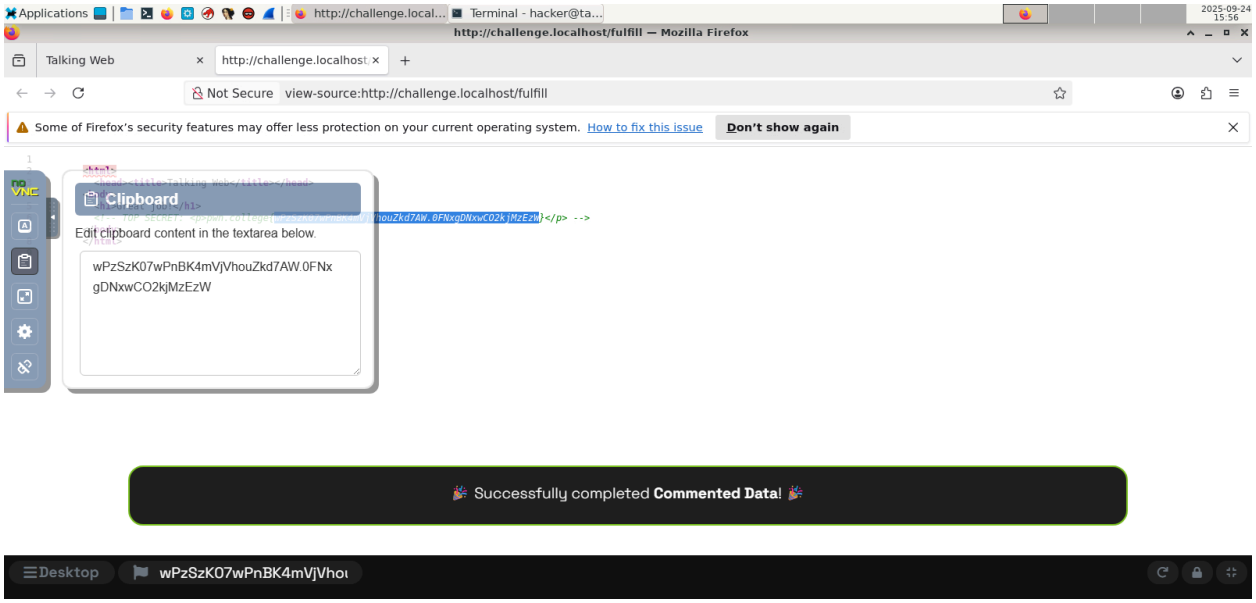
- Commented Data



Launching the challenge in GUI desktop terminal.

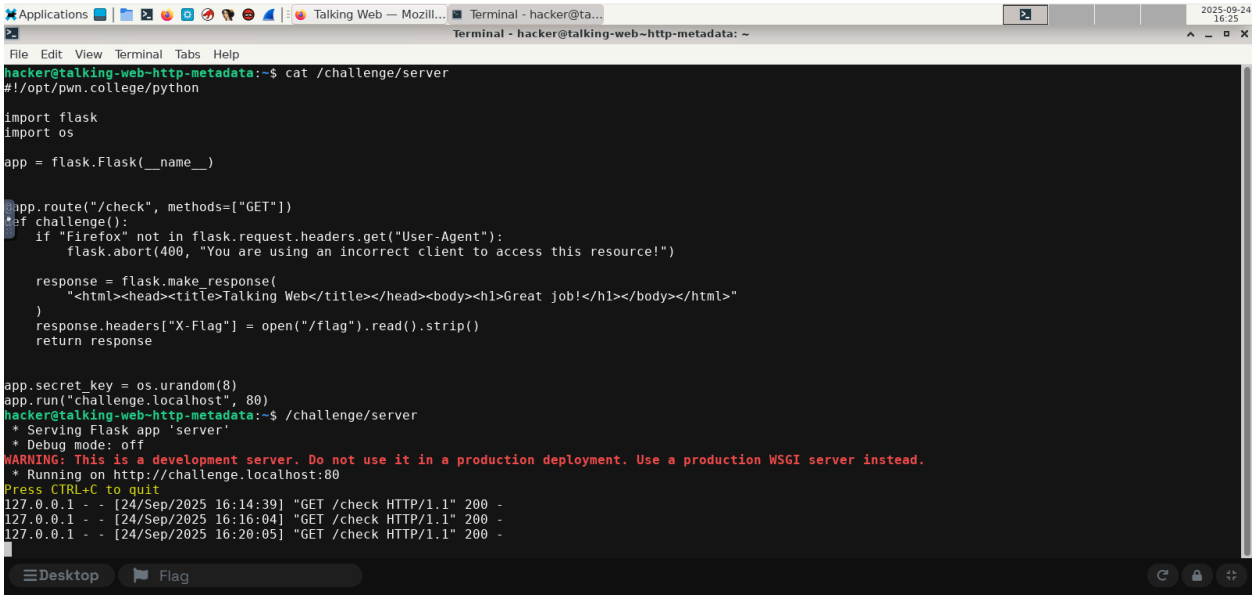


Continuation of first pic in terminal.

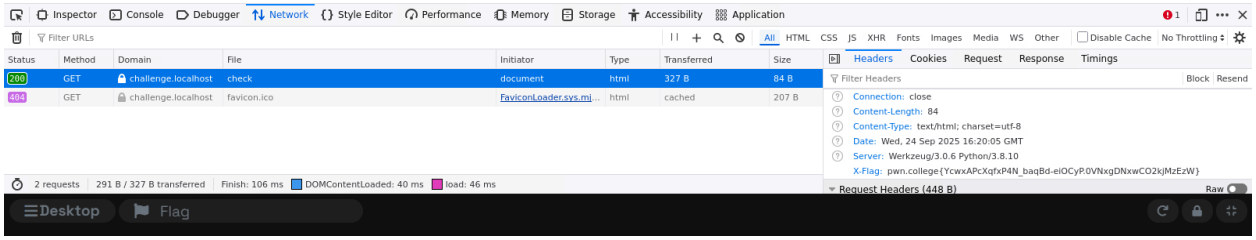


I found the flag by first running the **/challenge/server** script to start the web server. Then, I accessed the **correct endpoint /fulfill** in Firefox from the **GUI Desktop**, as required by the **server code**. Since the flag was **hidden** in an **HTML comment**, I viewed the **page source** by right-clicking and selecting "**View Page Source**" or using the **Firefox menu**. In the source code, I located the comment **<!-- TOP SECRET: <p>pwn.college{wPzSzk07wPnBK4mVy/houZkd7AW1QFNxgDNowCO2kjMzEzW}</p> -->** and copied the flag from there. The flag is **pwn.college{wPzSzk07wPnBK4mVy/houZkd7AW1QFNxgDNowCO2kjMzEzW}**. This challenge highlighted that not all data served to a client is meant for direct display; crucial information can be hidden within the page's structure, requiring inspection of the underlying source code.

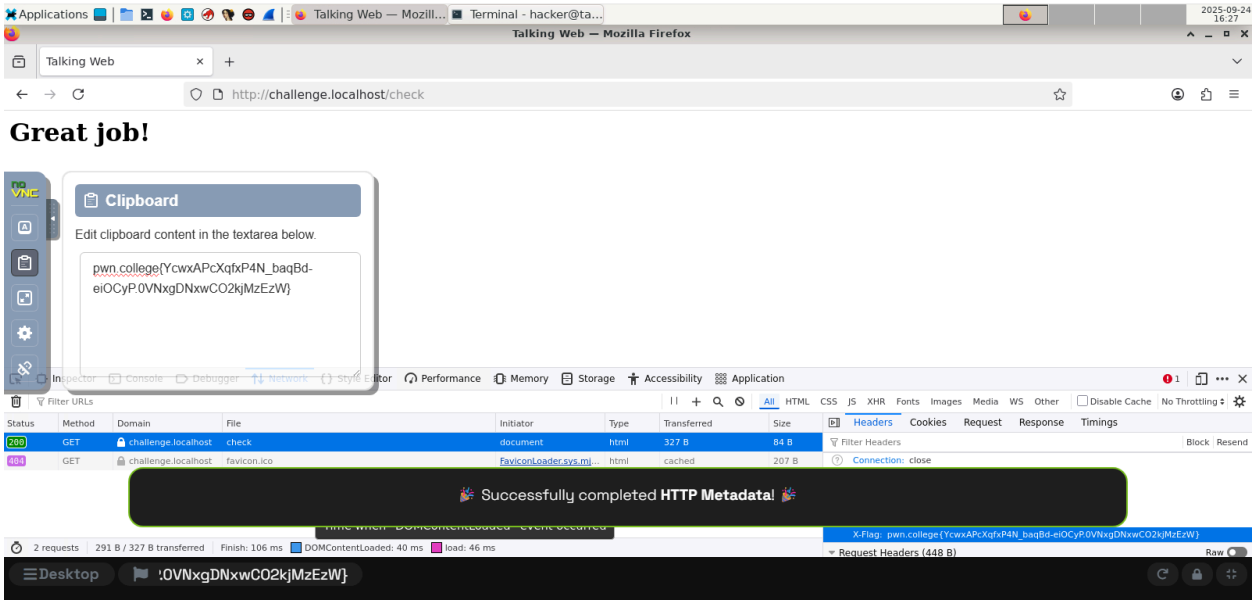
- HTTP Metadata



Launching the challenge in GUI desktop terminal.

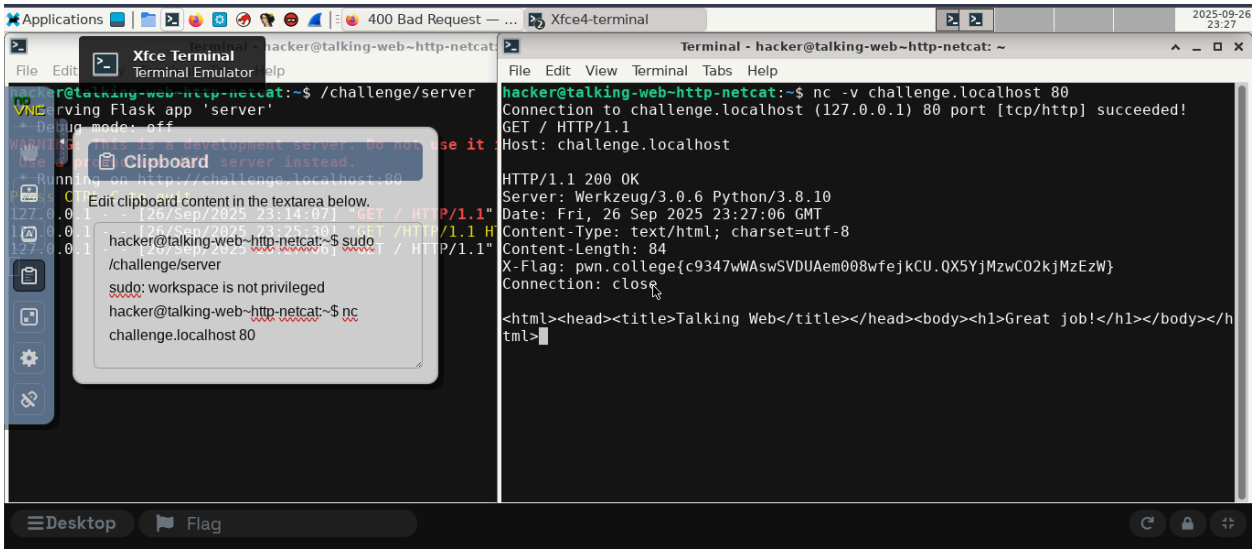


Retrieving the file called X-Flag in network tab to see the flag, under web developer tools after a refresh.

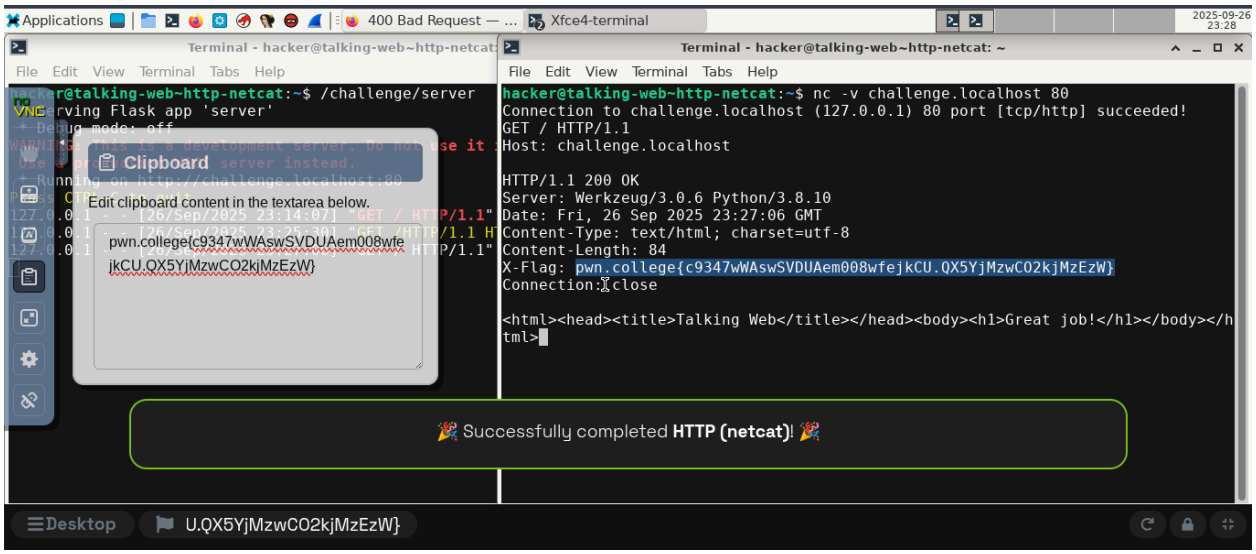


To capture the flag for this challenge, I first ran the **/challenge/server** script to start the web server, which began listening on **http://challenge.localhost:80**. I then accessed the correct **endpoint /check** using **Firefox** from the **GUI Desktop**, ensuring the User-Agent header was set correctly. Since the flag was embedded in a response header, I opened Firefox's **Web Developer Tools** via the menu (≡) > **More Tools** > **Web Developer Tools** and navigated to the Network tab. After reloading the page, I selected the **/check** request and examined the **Response Headers**, where I found the X-Flag header containing the flag: **pwn.college{YcwxAPcXqfxP4N_baqBd-eiOCyP.0VNxgDNxwCO2kjMzEzW}**. This demonstrated the importance of inspecting HTTP metadata to retrieve hidden data. It reinforced that an HTTP response is more than just a body; critical data can be transmitted silently in headers, which are essential for controlling communication but are invisible in the standard browser view.

- HTTP (netcat)

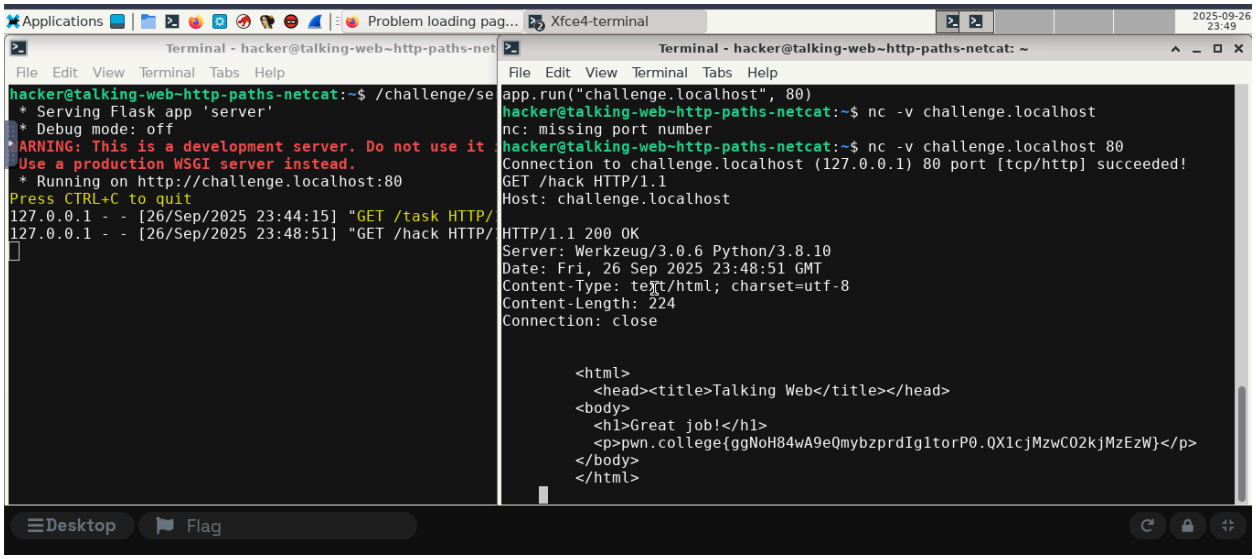


This is after connecting to netcat.

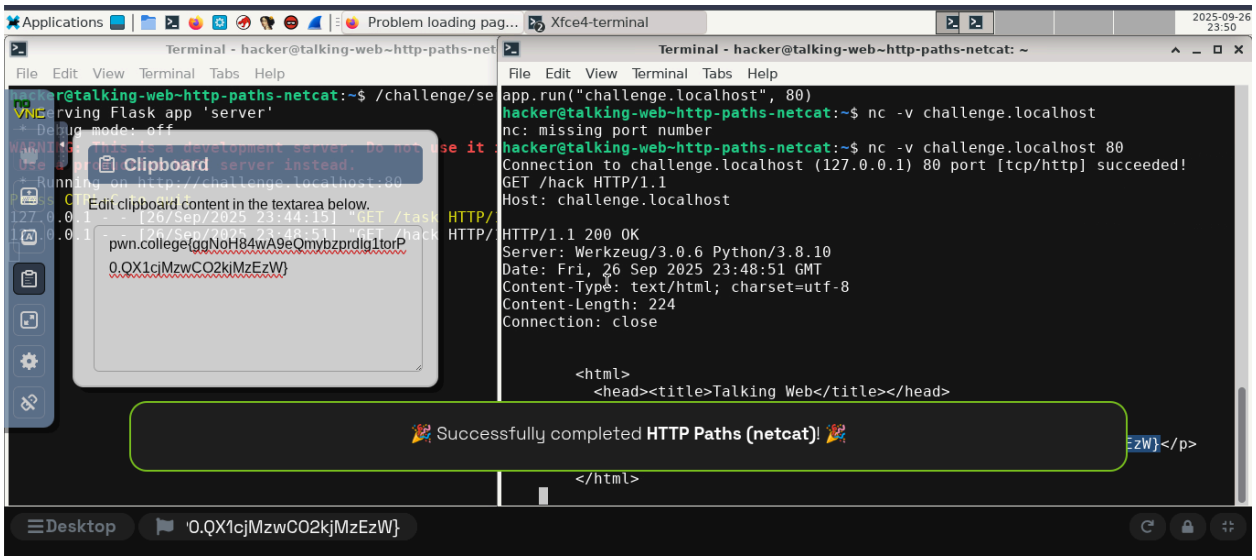


I started by using **netcat** to establish a raw **TCP** connection to the **HTTP server** running on **challenge.localhost port 80**. Once connected, I manually crafted an **HTTP GET** request by typing **GET / HTTP/1.1** followed by the Host: challenge.localhost header, which is essential for **HTTP/1.1 requests** to specify the virtual host. After sending the request (by pressing Enter twice to indicate the end of the headers), the server responded with a full **HTTP response**. I carefully examined the response headers and found the flag in the X-Flag header, which contained the value **pwn.collegeC9347wWAswSVDUAem008wfe[KCL.OXSYjMzwCO2kjMzEzWJ]**. This flag was then displayed in the terminal, confirming that I had successfully retrieved it by interacting directly with the web server using **netcat**. Using netcat stripped away the abstraction of a browser, revealing the raw text-based protocol of HTTP and proving that any TCP-capable tool can be used to interact with web services.

- HTTP Paths (netcat)

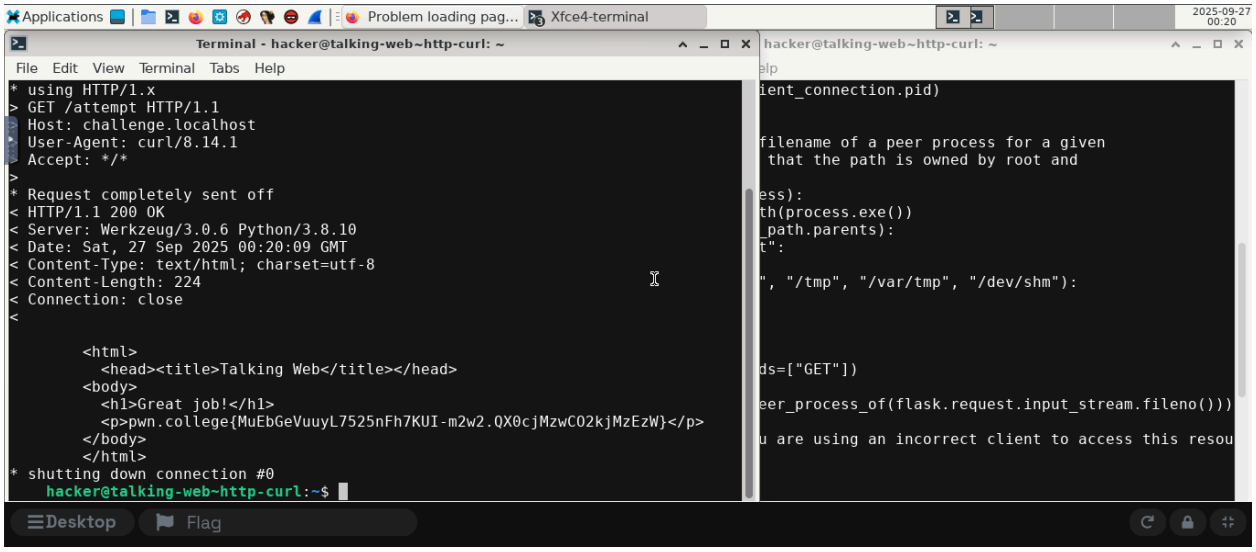


This is after connecting to netcat.

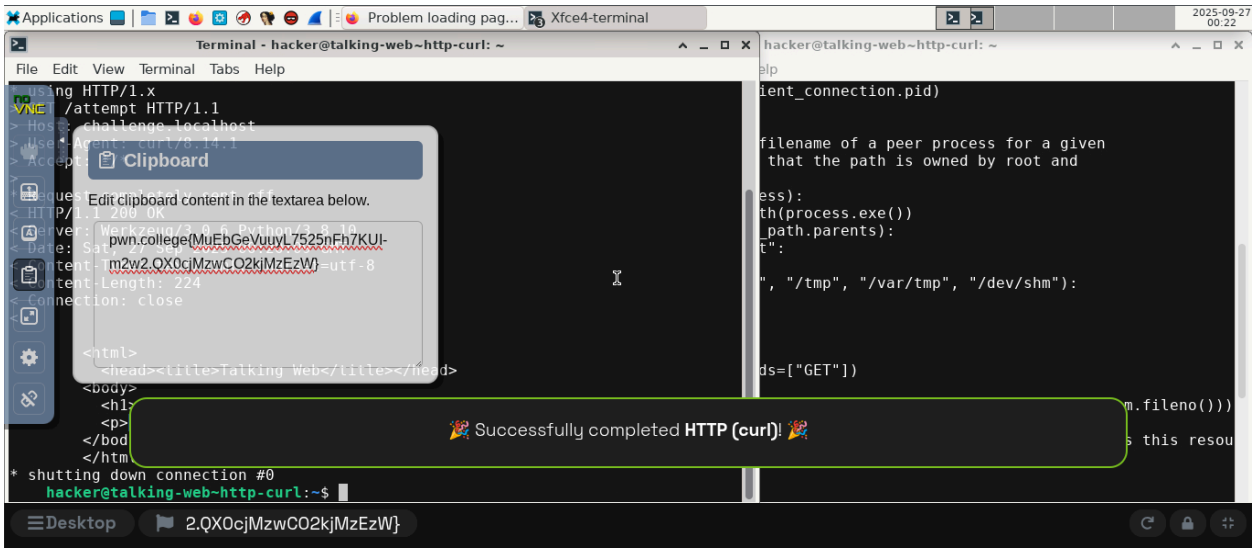


I started by running **netcat** to connect to the web server at **challenge.localhost** on **port 80**. Once connected, I sent an **HTTP GET** request to the **path /hack** with the required Host header. The server responded with a **200 OK status**, indicating that I had successfully accessed the correct path. This completed the challenge, and the flag was revealed in the clipboard content as **pwn.college(gqNot18AwA9eQmybzprdld)torP0.QX1cNAzwCQ2kWAEzW)**. By manually crafting the request to the specific path, I was able to retrieve the flag and complete the **HTTP Paths challenge**. This emphasized how path-based routing works: the server interprets the path in the request line to determine which specific resource or function to execute on the server side.

- HTTP (curl)

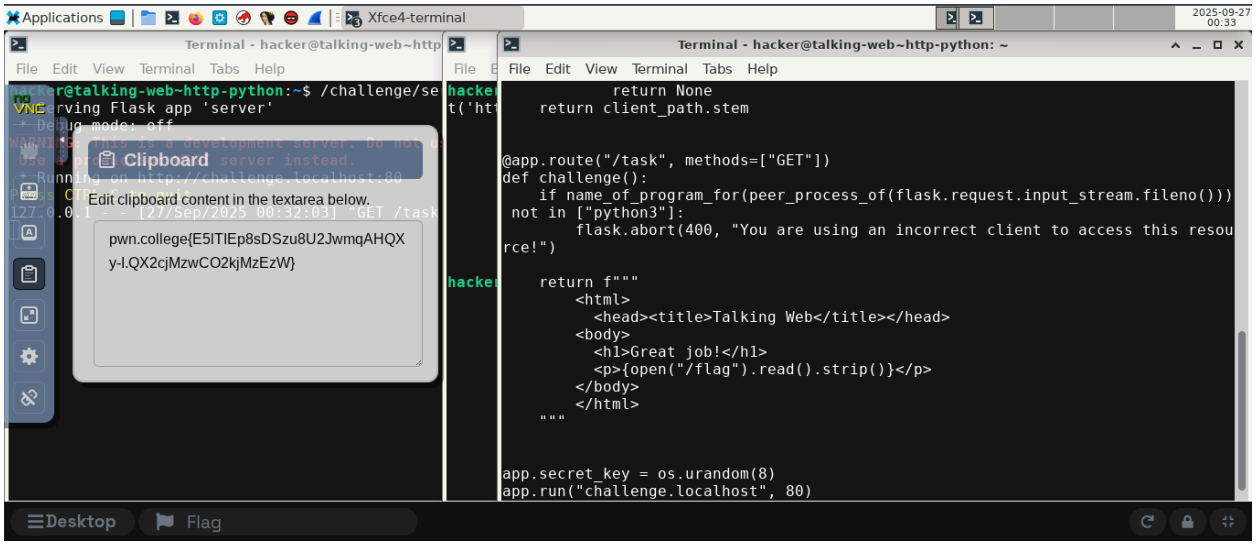


This is after retrieving the endpoint /attempt

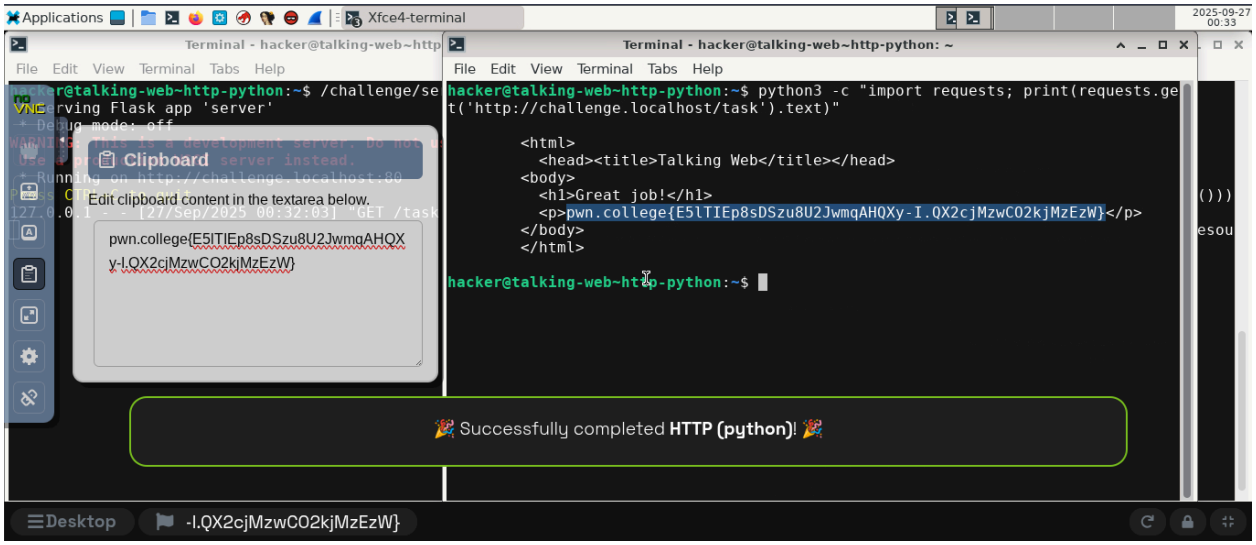


I used curl to send an **HTTP GET request** to the server at **challenge.localhost** on **port 80**, specifically targeting the **/attempt path**. After executing the command, the server responded with a **200 OK status**, and the flag was displayed in the response output. I then copied the flag from the terminal output, which was **pwn.college(MuEbGeVuuyL7525nFh7KUJ-m2w2.CX0cjMzwC02xjMzEzW)**, confirming the successful completion of the **HTTP challenge** with **curl**. Using curl provided a powerful and scriptable command-line method for making HTTP requests, showcasing an essential tool for testing and automating web interactions outside of a graphical browser.

- HTTP (python)

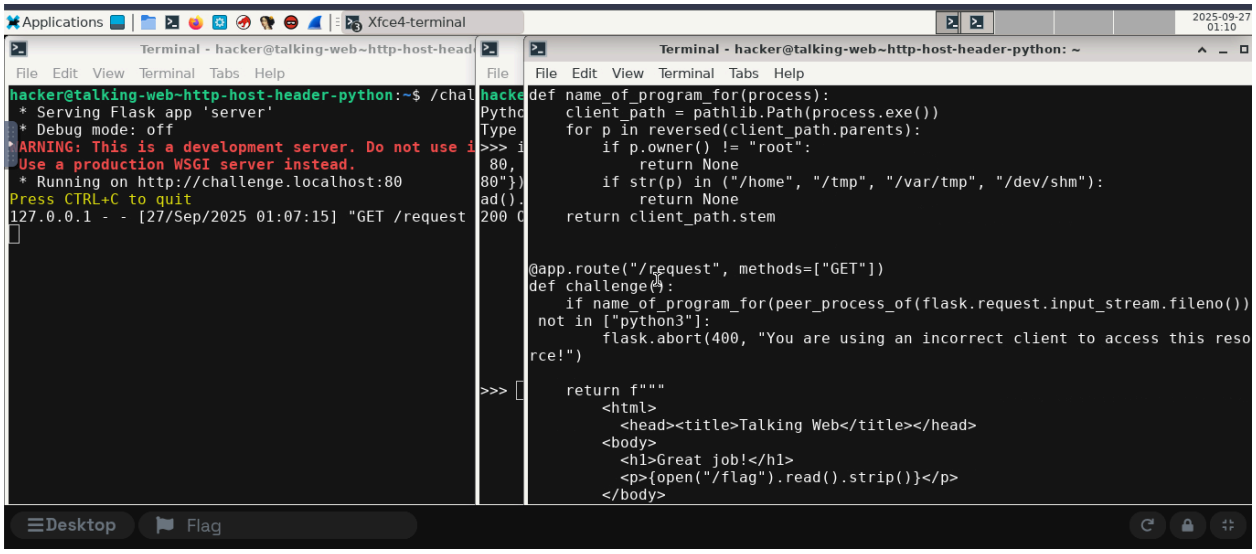


Used cat /challenge/server to get the endpoint of /task

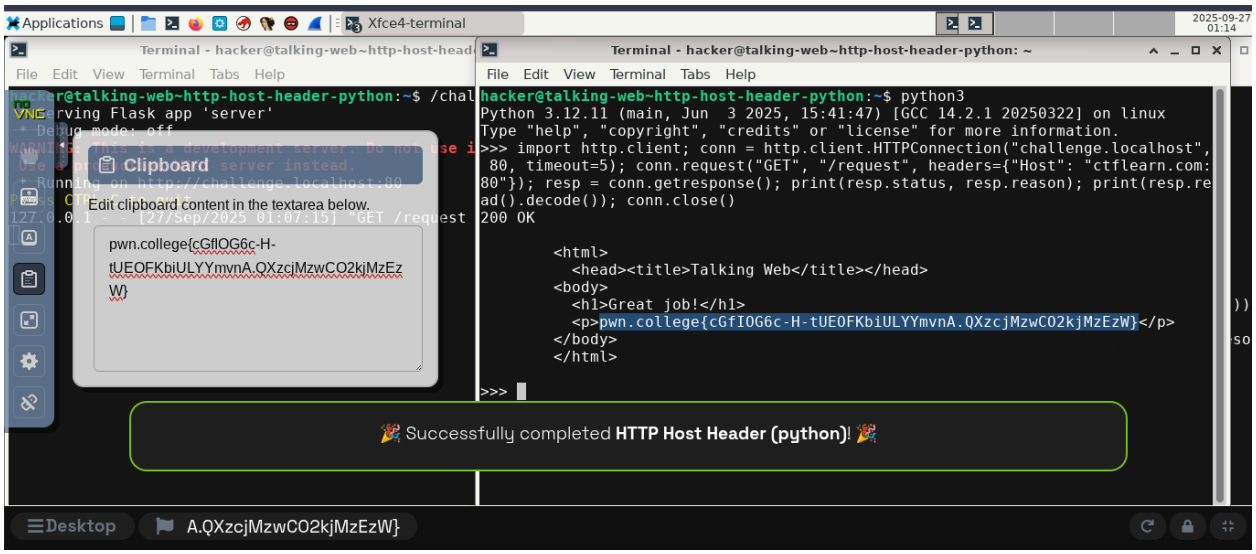


I started by running the **Flask app server** for the challenge, which set up the local environment. Then, I opened a terminal and used **Python 3** with the requests library to send a **GET request** to **http://challenge.localhost/task**. The command I executed was **python3 -c "import requests; print(requests.get('http://challenge.localhost/task').text)"**. The server responded with an **HTML page** that contained the flag within a paragraph tag. I saw the flag printed directly in the terminal output: **pwn.college(E5lTIEP8SDSzu8U2JwmqAHOXv-1.0x2ciMzwCO2kjMzEzW)**. This confirmed that I successfully completed the **HTTP challenge** using Python. This approach demonstrated the power of using a scripting language like Python to programmatically interact with web servers, a foundational technique for building automated clients and security tools.

- HTTP Host Header (python)

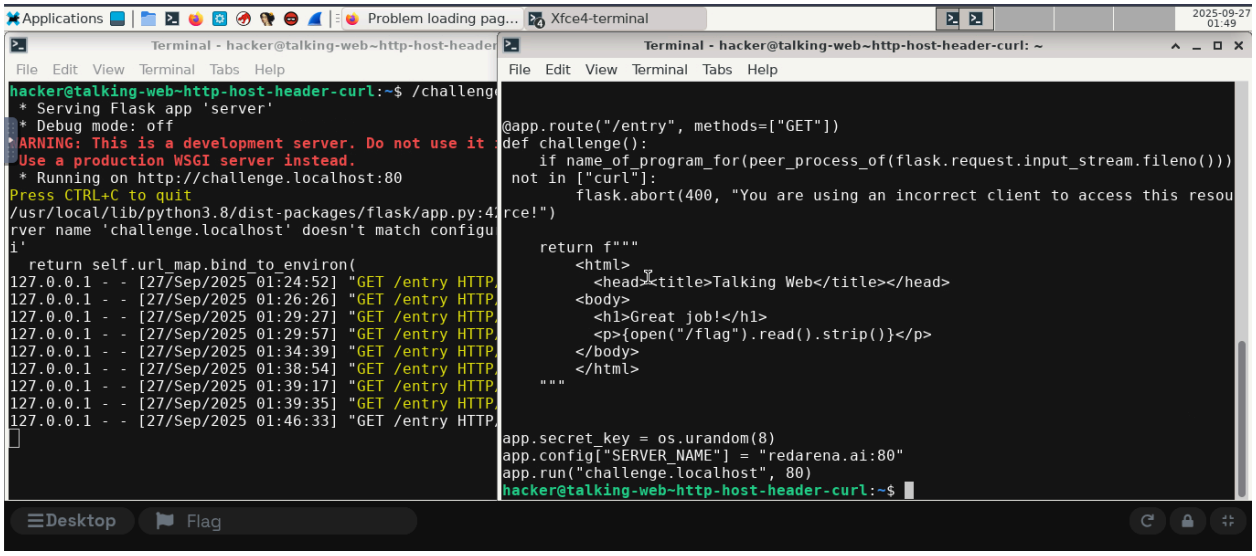


Used cat /challenge/server to get the endpoint of /request

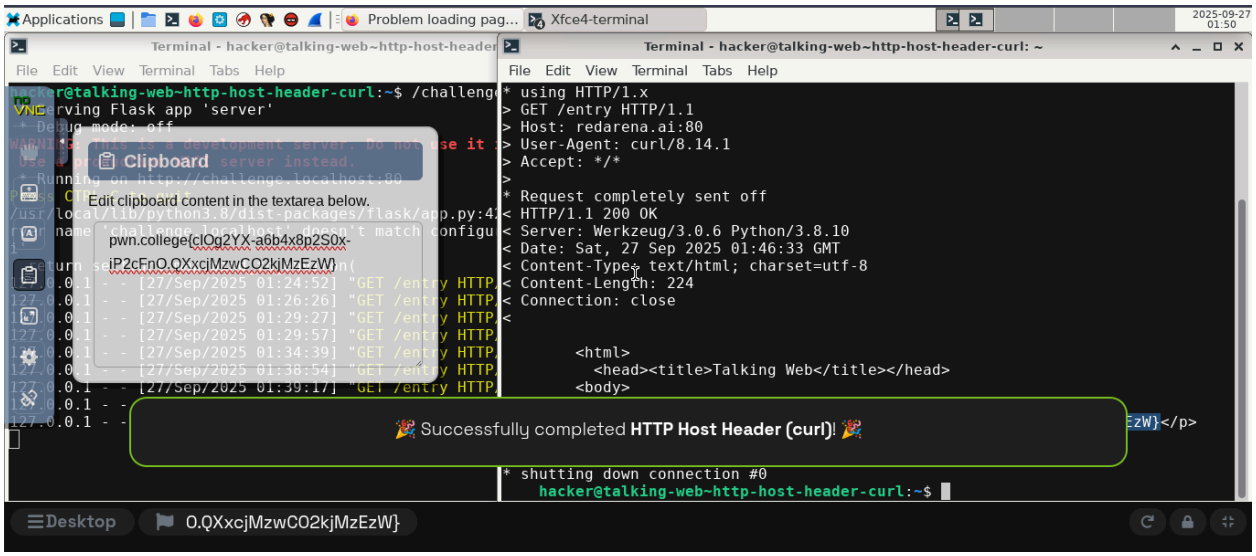


I started by importing the http.client module in **Python** to handle the **HTTP connection**. Then, I established a connection to **challenge.localhost on port 80** with a timeout of 5 seconds. I crafted a **GET request** to the **path /request** and set the **Host header** to **ctflearn.com:80** to meet the challenge requirements. After sending the request, I retrieved the response, which had a status code of **200 OK**. The response body contained an **HTML page** with the flag embedded in a paragraph tag, which I printed out. The flag was **pwn.college[cGI0056c-H-ULEOFKBJULYYmwnA.0XzcjMzwCO2kjMzEzW]**, confirming that I successfully completed the **HTTP Host Header** challenge. Manipulating the Host header via script highlighted its critical role in virtual hosting, where a single server can host multiple websites and uses this header to determine which site to serve.

- HTTP Host Header (curl)

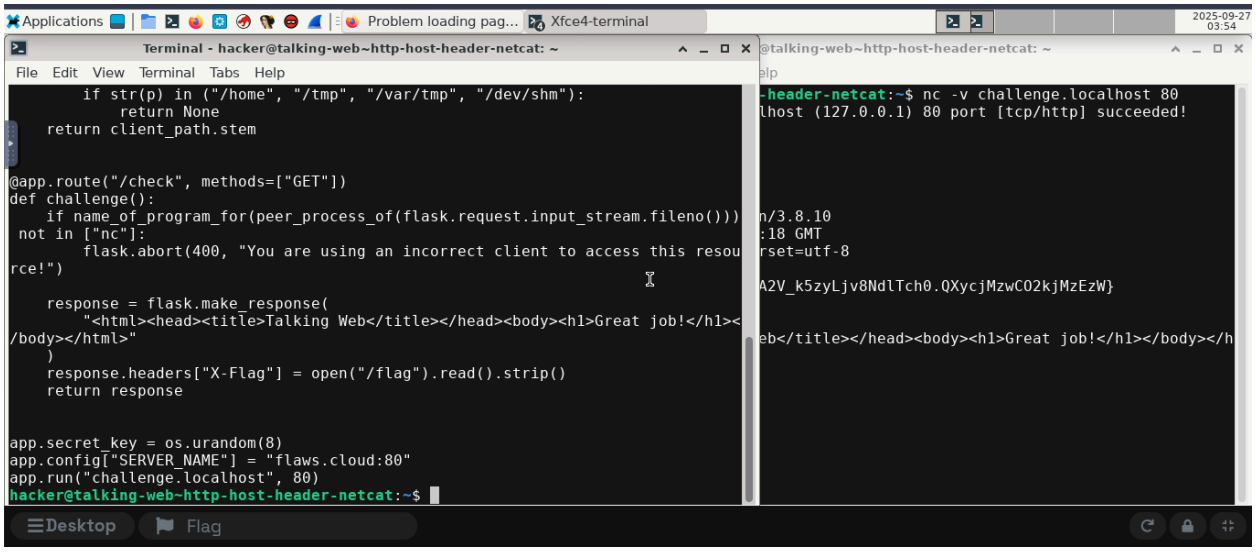


Used cat /challenge/server to get the endpoint of /entry

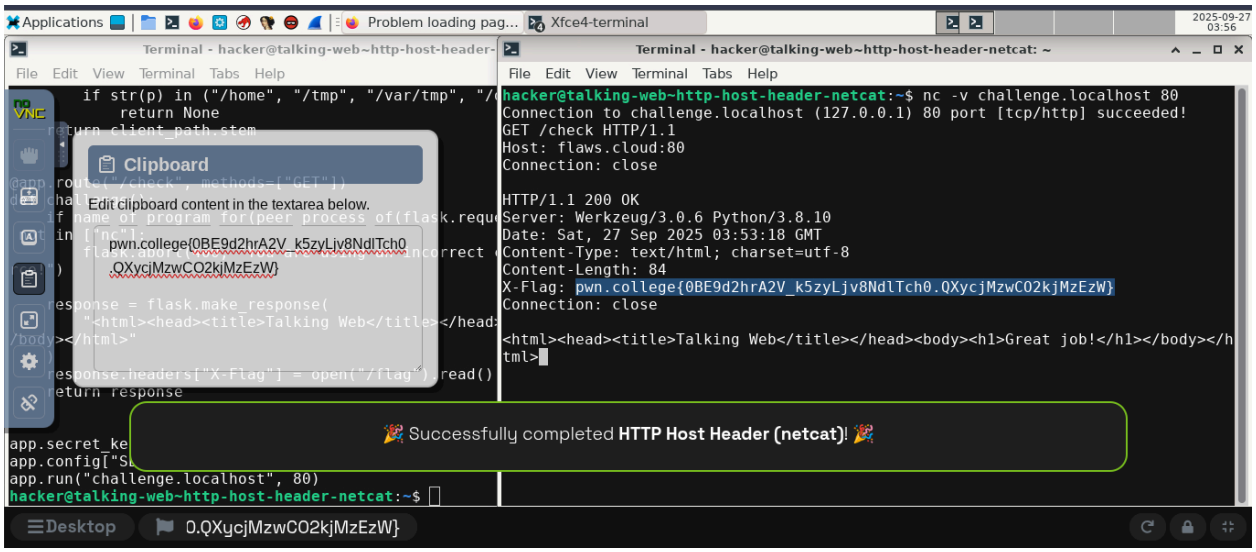


I started by using curl to send an **HTTP GET** request to the server at **challenge.localhost** on **port 80**. I specified the **path /entry** and set the **Host header** to **redarena.al:80** with the appropriate **curl** options. The server responded with a **200 OK** status, and the HTML body confirmed the request was successful. After completing the challenge, I retrieved the flag from the clipboard, which was **pwn.college(c)Q2YX-a9b4x8pZSQX-B2cFnO.QXxcjMzwCQ2kJhzEzWj**. This verified that I had correctly manipulated the Host header to satisfy the challenge requirements. This exercise showcased curl's precision in crafting custom headers, a vital capability for testing web application security and behavior under different hosting conditions.

- HTTP Host Header (netcat)

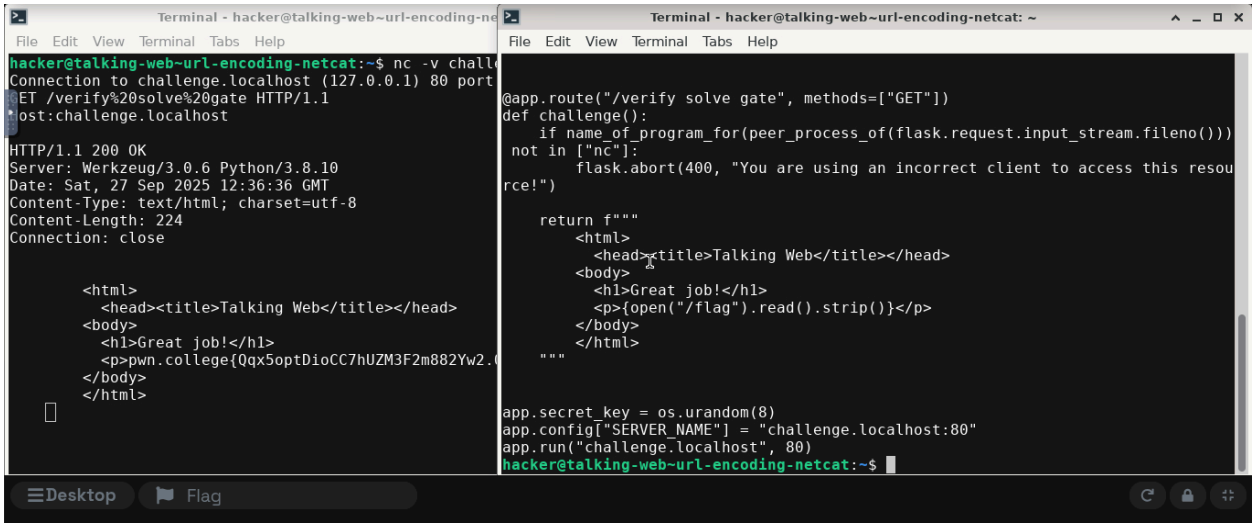


Used cat /challenge/server to get the endpoint of /check, Host: flaws.cloud:80

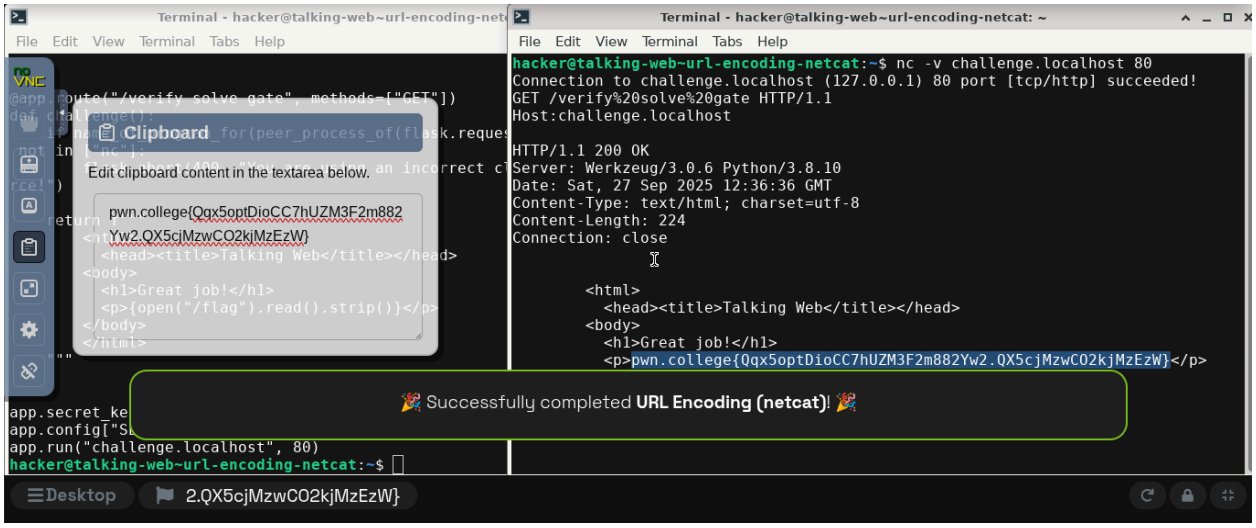


I started by using **netcat** to establish a **TCP connection** to the server at **challenge.localhost** on **port 80**. Once connected, I manually crafted an **HTTP GET** request to the **path /check** with the **Host header** set to {**lans.cloud:80** and added a Connection: close header to ensure the connection would close after the response. After sending the request, the server responded with a **200 OK** status, and I found the flag in the X-Flag header of the response: **pwn.college(OBE90ZhrA2Y_K5zyLiv8Nd1Tch0_0XycjMzwCO2KjMzEzW)**. This confirmed that I successfully completed the HTTP Host Header challenge by manipulating the Host header correctly. Manually setting the Host header with netcat cemented the understanding that this header is not magical but a simple, client-sent directive that is fundamental to how the modern web functions.

- URL Encoding (netcat)

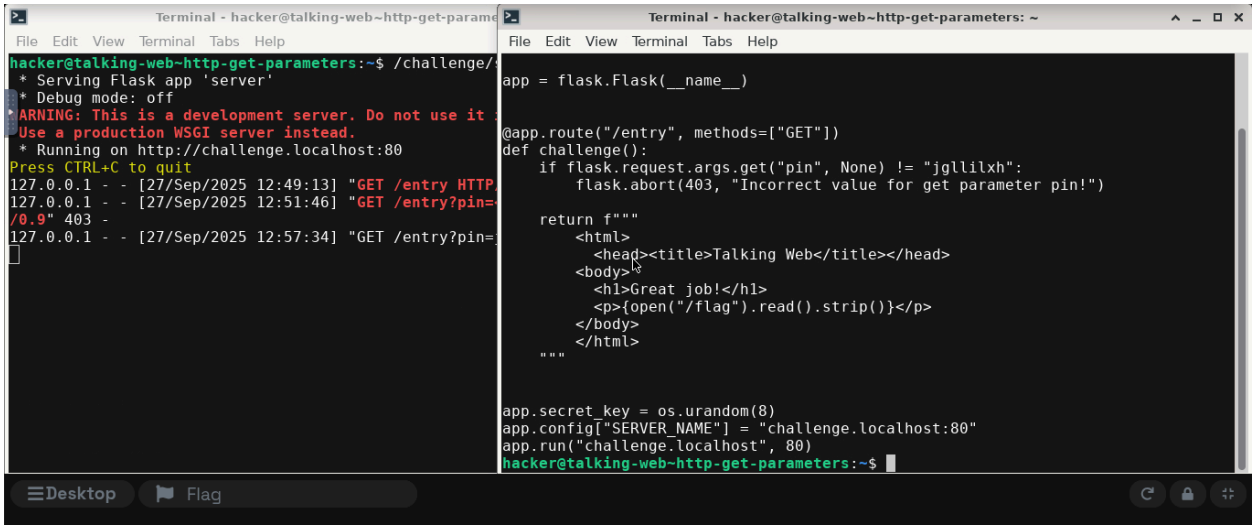


Used cat /challenge/server to get the endpoint of /verify solve gate

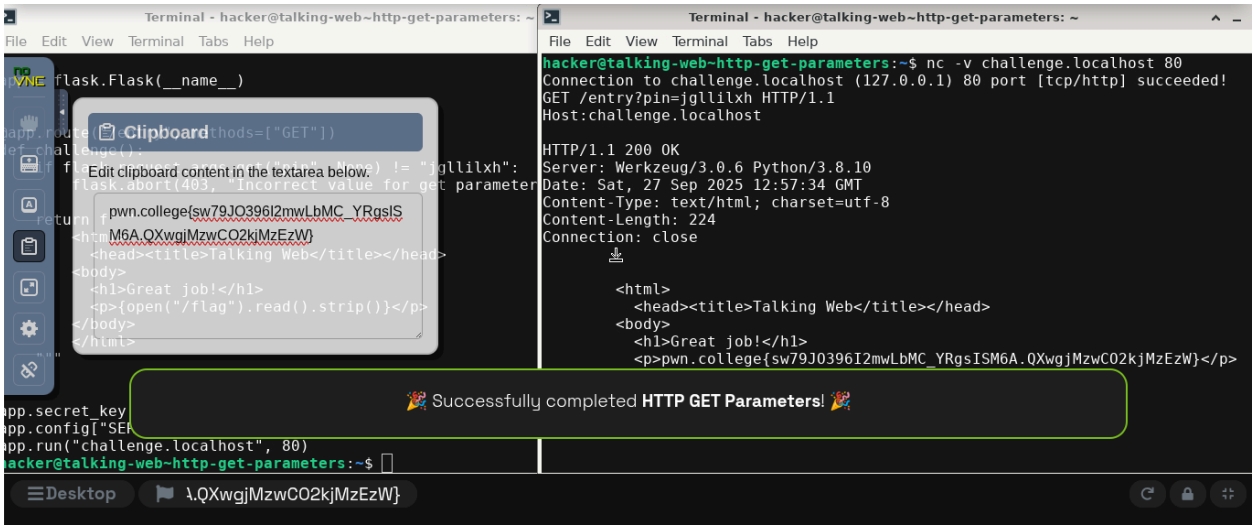


I started by using **netcat** to connect to the web server at **challenge.localhost** on **port 80**. Once connected, I sent a **GET request** to the **path /verify%20solve%20gate**, which included **URL encoding** for the spaces (using %20). I also included the **Host header set to challenge.localhost**. The server responded with a **200 OK status**, and the **HTML body** contained the flag within a paragraph tag: **pwn.college(Qqx5optDioCCTNUZN3F2m882Yw2.QX5GjMzwCO2kjMzEzW)**. This confirmed that I successfully completed the URL encoding challenge by properly encoding the path in the HTTP request. This illustrated why URL encoding is necessary: it allows characters with special meanings (like spaces) to be safely transmitted within a URL, ensuring the server correctly parses the intended path and parameters.

- HTTP Get Parameters

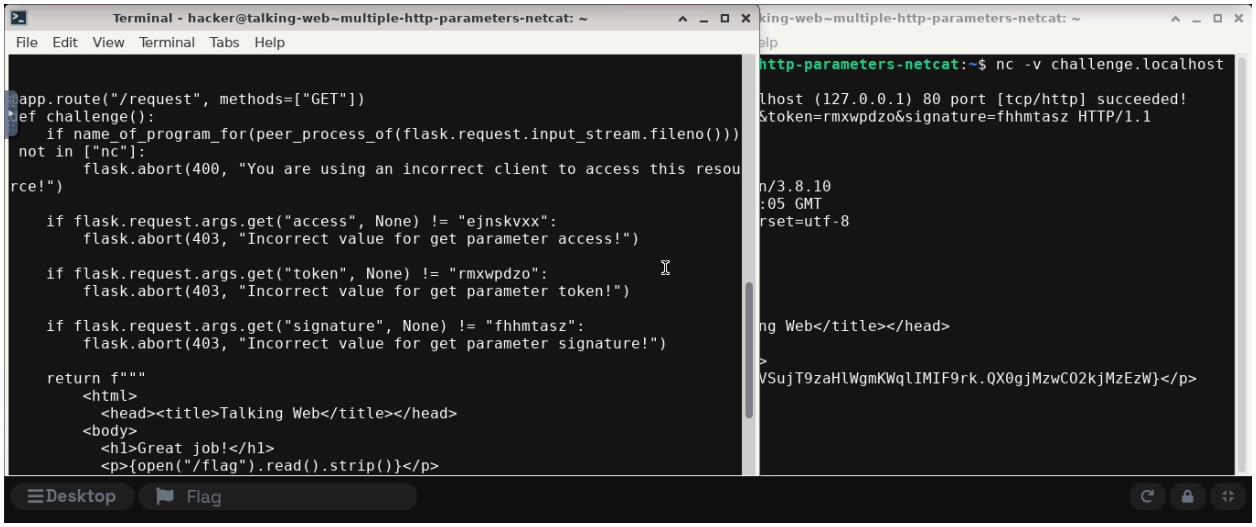


Used cat /challenge/server to get the endpoint of /entry

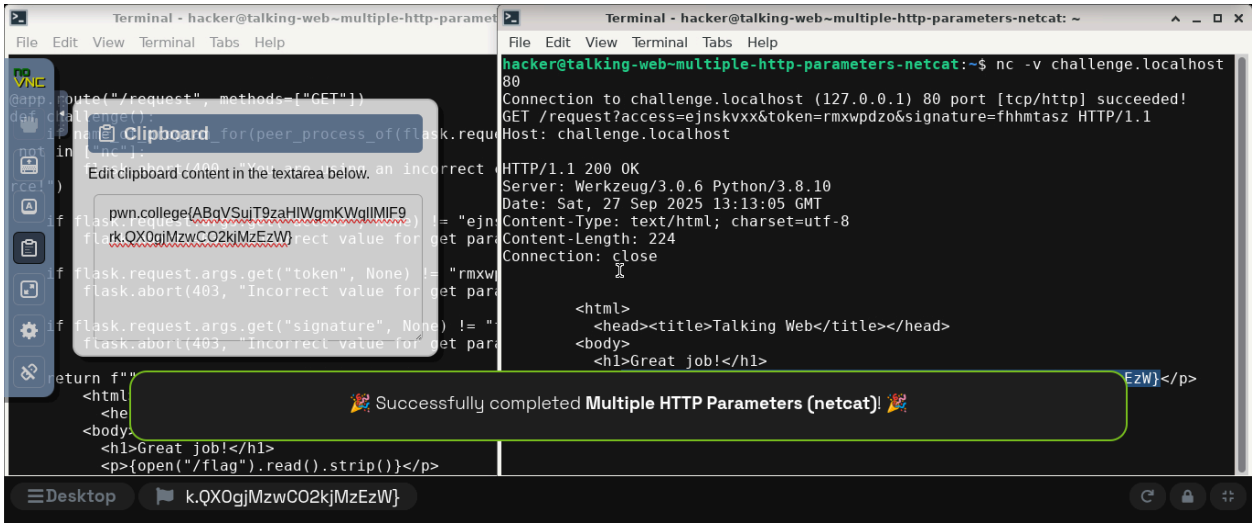


I started by using **netcat** to connect to the web server at challenge.localhost on port 80. Once connected, I manually crafted an **HTTP GET request** to the **path /entry** with the query parameter **pin=jglllxh**. I also included the **Host header** set to **challenge.localhost**. After sending the request, the server responded with a **200 OK status**, and the **HTML body** contained the flag within a paragraph tag: **pwn.college(sw7900396f2mwLbMC_YRqsISM6A.QXwgjMzwCO2kjMzEzw0)**. This confirmed that I successfully retrieved the flag by including the correct GET parameter in the request. This demonstrated the primary method of passing user-supplied data to a server via the URL, forming the basis for dynamic content, search queries, and statefulness in web applications.

- Multiple HTTP Parameters (netcat)

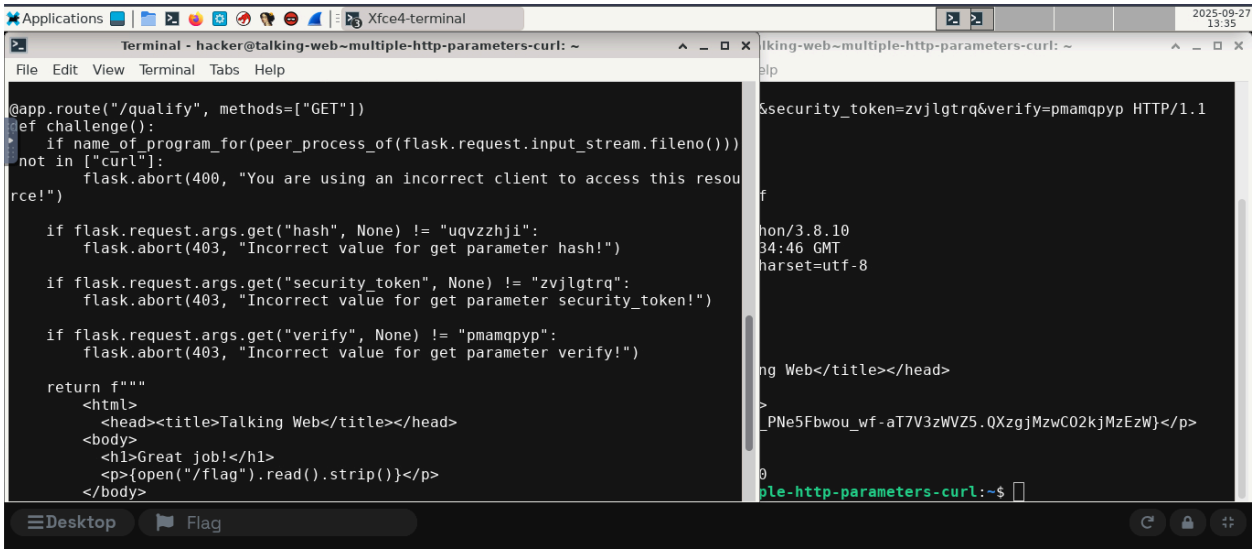


Used cat /challenge/server to get the endpoint of /request with the following parameter access=ejnskvxx, token=rmxwpdzo, and signature=fhhmtasz

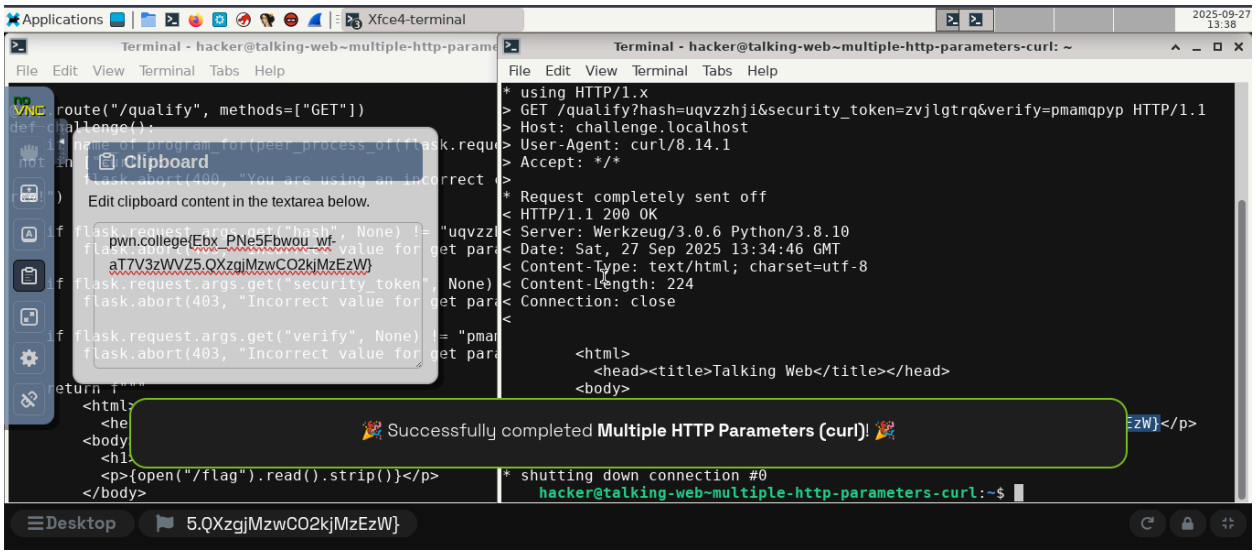


I started by using **netcat** to connect to the web server at **challenge.localhost** on **port 80**. Once connected, I crafted an **HTTP GET** request to the **appropriate path**, which included multiple **query parameters**: "**access**" and "**token**" and "**signature**" with their correct values. I made sure to include both **parameters in the query string**, as the server would abort with a **403 error** if either was missing or incorrect. After sending the request, the server responded with a **200 OK** status, and the flag was displayed in the response body: **pwn.college(ABqVSJLT9zatHWqmKVwUHMF9)tk.QXOgMxwCO2kNMEzW]**. This confirmed that I successfully completed the challenge by providing the required multiple HTTP parameters. The challenge illustrated how servers often require a specific set of parameters to authorize an action, and how missing or incorrect parameters lead to defined error states like 403 Forbidden.

- Multiple HTTP Parameters (curl)



Used cat /challenge/server to get the endpoint of /qualify with the following parameter hash=uqvzzhji, security_token=zvjlgtrq, and verify=pmamqpyp



I started by using **curl** to send an **HTTP GET** request to the server at **challenge.localhost** on **port 80**. I targeted the **path /quality** and included all three required **query parameters**: **hash=ugvzzhjl, security_token=zvjlgtrg, and verify=pmamgpyp**. After executing the command, the server responded with a **200 OK status**, and the **HTML body** contained the flag within a paragraph tag. The flag was **pwn.college(EDX_PNe5fbwou_wk_home) aT7V3zWYZ5.OXzqjMzwCO2kjMzEzW9**, confirming that I had correctly provided all the necessary parameters to complete the challenge. Using curl to concatenate multiple parameters efficiently demonstrated a practical method for testing complex API endpoints or web application forms that rely on several inputs.