WAPH-Web Application Programming and Hacking

Instructor: Dr. Phu Phung

Student

Name: Vishal Kothapalli

Email: kothapvl@mail.uc.edu

Short-bio:

Repository Information

Respository's URL: [https://github.com/kothapvl-uc/waph-kothapvl.git]

This is a private repository for Vishal Kothapalli to store all code from the course. The organization of this repository is as follows.

Lab 1 - Foundations of the Web

The lab's overview

This lab is designed to provide hands-on experience with web technologies and HTTP protocol. It's divided into two parts:

Part I focuses on understanding the underlying mechanisms of web communication. You'll learn to use Wireshark to analyze network traffic and explore HTTP interactions using telnet.

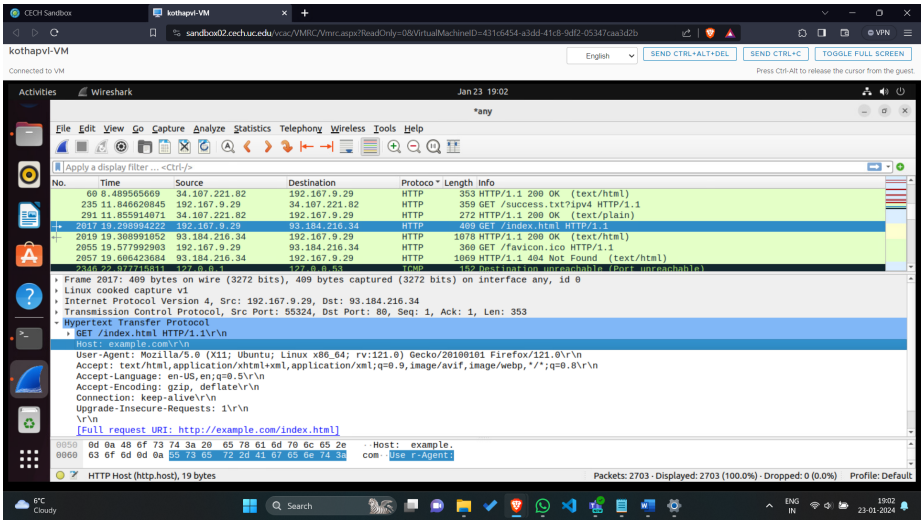
Part II delves into web application development. You'll create simple web applications using both C and PHP, incorporating user input and exploring how HTTP GET and POST requests function within these applications.

Also, include a direct clickable link to the lab folder on GitHub.com so that it can be viewed when grading, for example, https://github.com/phungph-uc/waph-phungph/tree/main/labs/lab1.

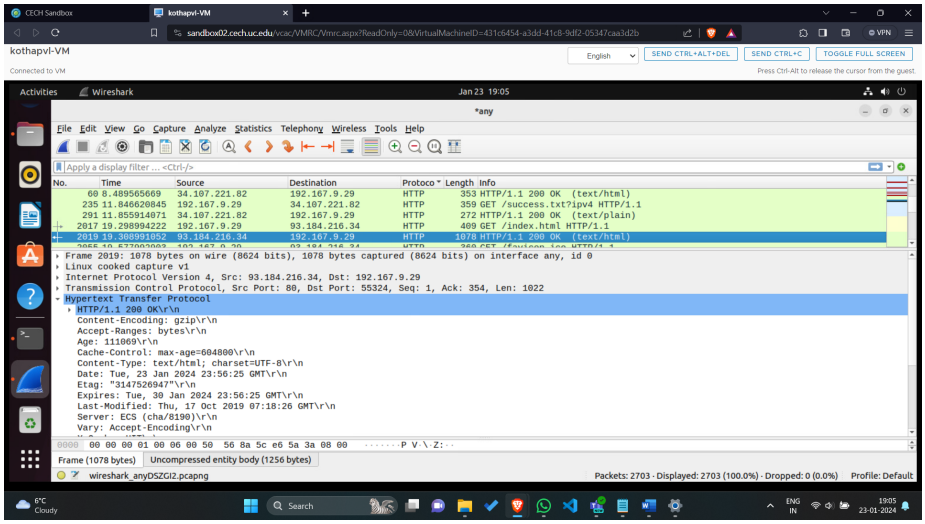
Part I - The Web and HTTP Protocol

Task 1. Familiar with the Wireshark tool and HTTP protocol

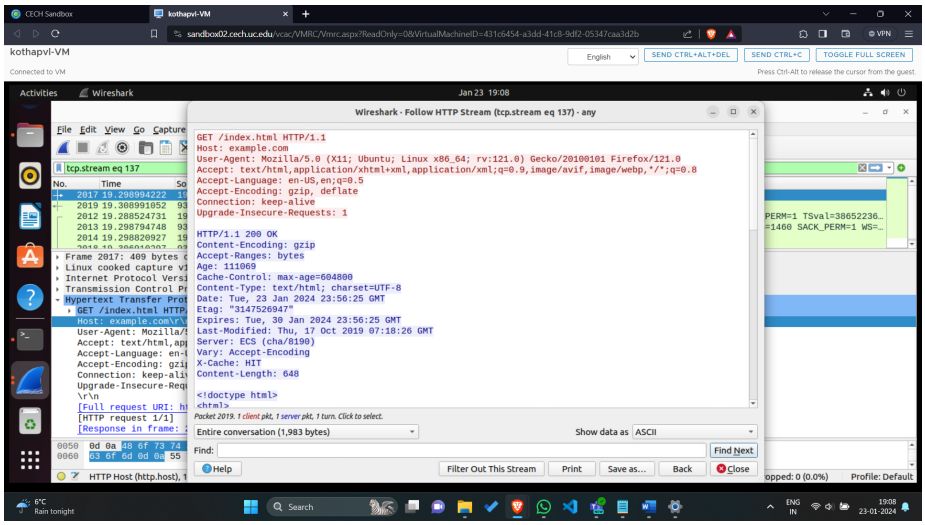
Summary of Wireshark tool: 1. Captured network traffic and filtered for HTTP packets using Wireshark's search bar. 2. Analyzed individual requests and responses, dissecting headers, parameters, and content. 3. Gained insights into website interactions, resource usage, and potential protocol issues.



HTTP-Request



Response

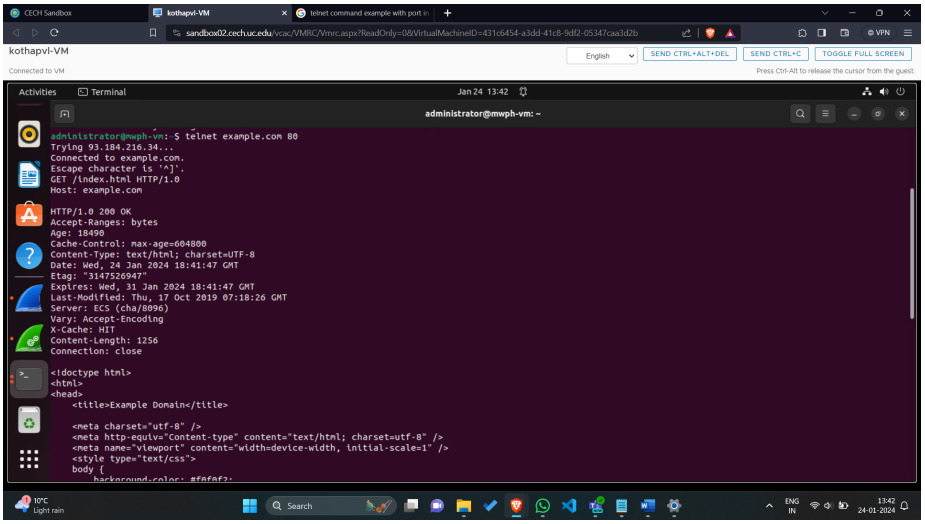


Stream

Task 2. Understanding HTTP using telnet and Wireshark

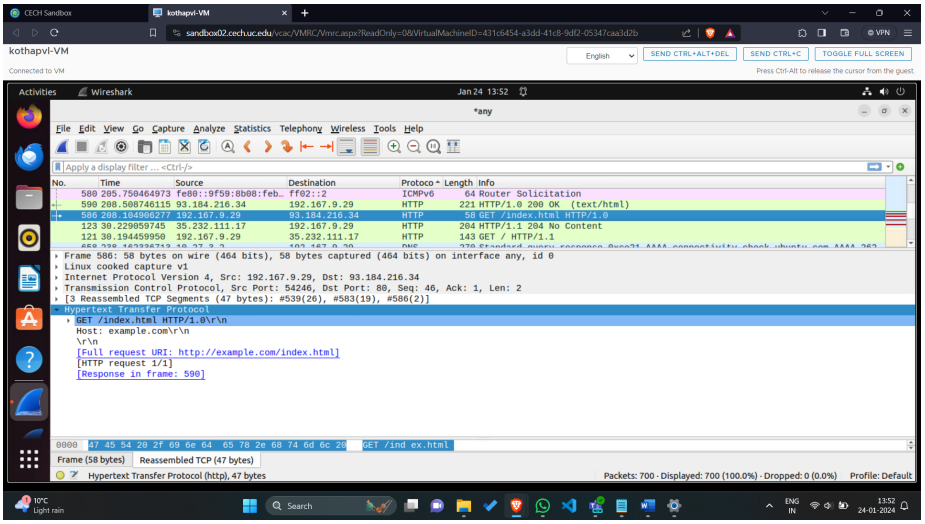
Sending a minimal HTTP request with Telnet involves connecting to the server on port 80 and manually typing the request line (GET /) and Host header. Wireshark captures network traffic, allowing you to filter for HTTP messages and analyze them. You can view details like IP addresses, ports, request methods, response codes, and headers, gaining valuable insights into the communication

between your browser and the web server.

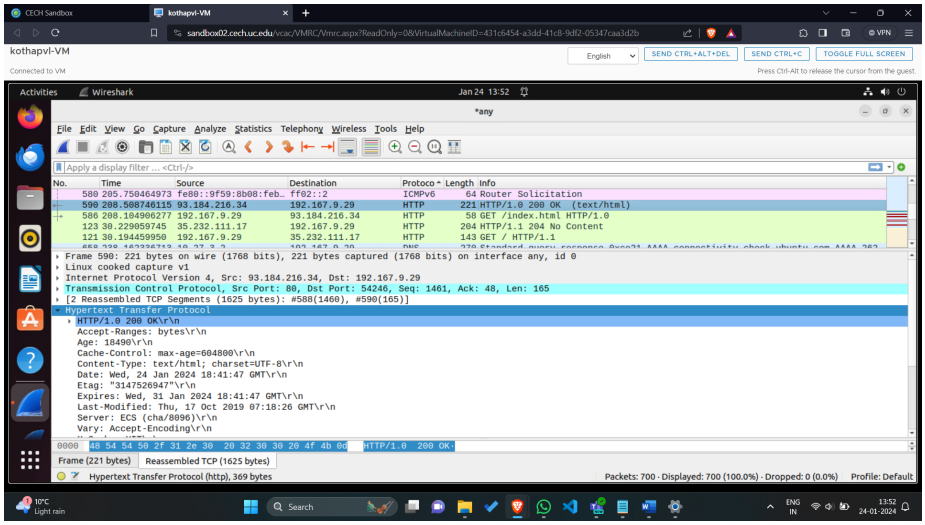


Terminal

Key distinctions emerge when comparing HTTP requests generated using telnet in a terminal with those constructed by web browsers. Telnet requests often exhibit a more streamlined structure, focusing primarily on fundamental elements such as the desired resource and target host. This stark contrast arises from telnet's nature as a manual command line interface, lacking the automated procedures that browsers employ to populate requests with an array of headers. These additional headers, commonly associated with cookies, user agent details, and content negotiation preferences, are integral to a browser's ability to deliver a seamless and compatible web experience.



Telnet



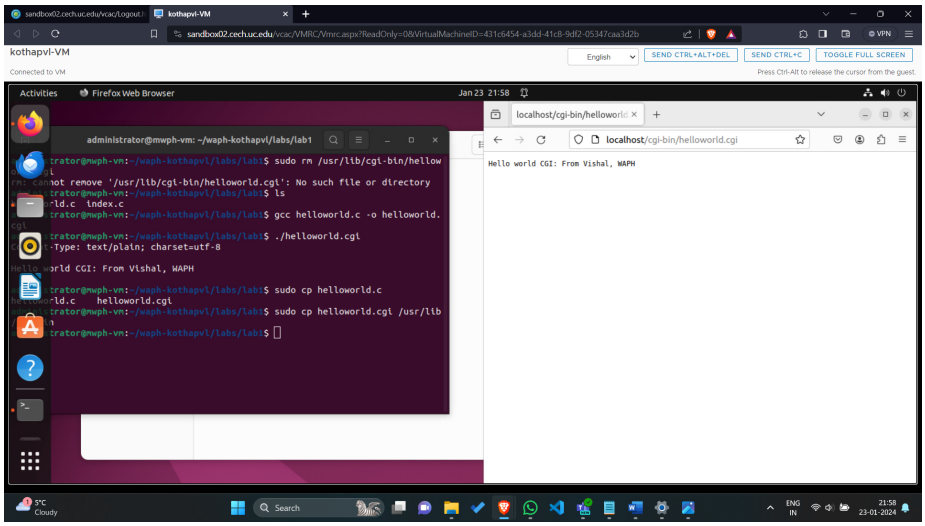
There is no major difference in the HTTP responses by the browser and telnet

Part II - Basic Web Application Programming

Task 1. (10 pts) CGI Web applications in C

a. I followed the below steps:

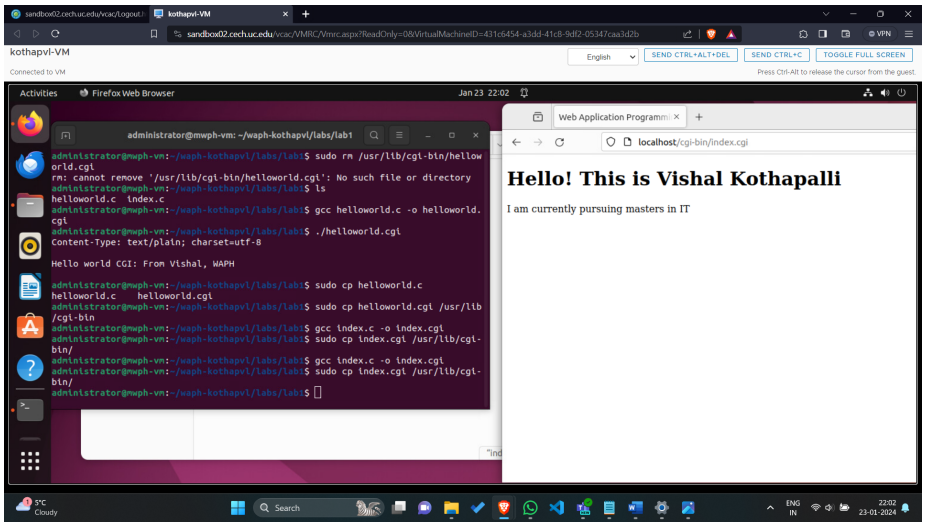
1. Write the C program.
2. Compile the program.
3. Place the executable in the CGI directory- Typically named cgi-bin within your web server's document root.
4. Access the application: Use a web browser to send requests to the URL pointing to the CGI script (e.g., http://yourserver/cgi-bin/yourprogram.cgi).



CGI

b. This program is a simple C CGI script that generates and displays basic HTML code. It sends an HTTP header specifying the content type as text/html. It then builds the HTML document structure using printf statements. The HTML code includes elements like head, title, body, and h1 for basic formatting. Finally, it returns 0 to indicate successful execution. This program demonstrates how C CGI scripts can dynamically generate HTML content.

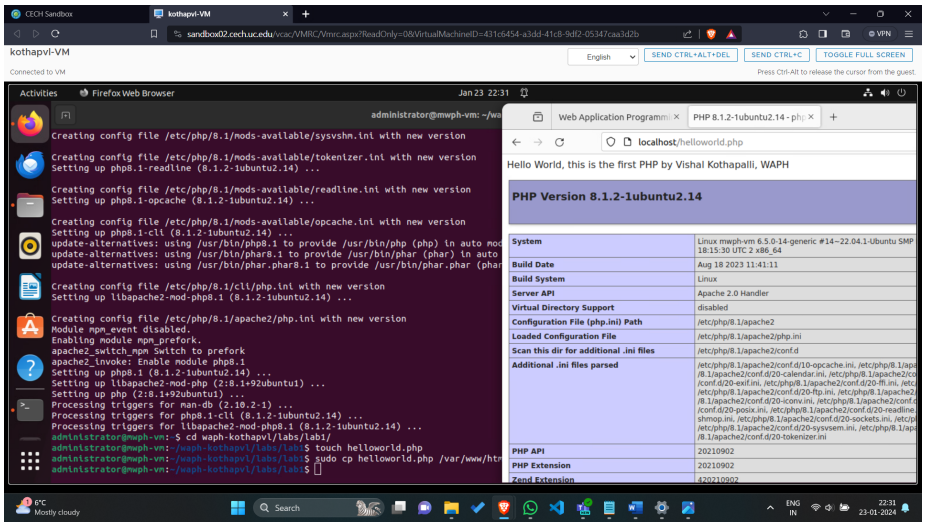
Included file `index.c`: `C #include <stdio.h> int main() { printf("Content-Type: text/html\n\n"); printf("<!DOCTYPE html>\n"); printf("<html>\n"); printf("<head>\n"); printf("<title>Web Application Programming and Hacking</title>\n"); printf("</head>\n"); printf("<body>\n"); printf("<h1>Hello! This is Vishal Kothapalli</h1>\n"); printf("<p>I am currently pursuing masters in IT</p>\n"); printf("</body>\n"); printf("</html>\n"); return 0; }`



HTML CGI

Task 2 (10 pts). A simple PHP Web Application with user input.

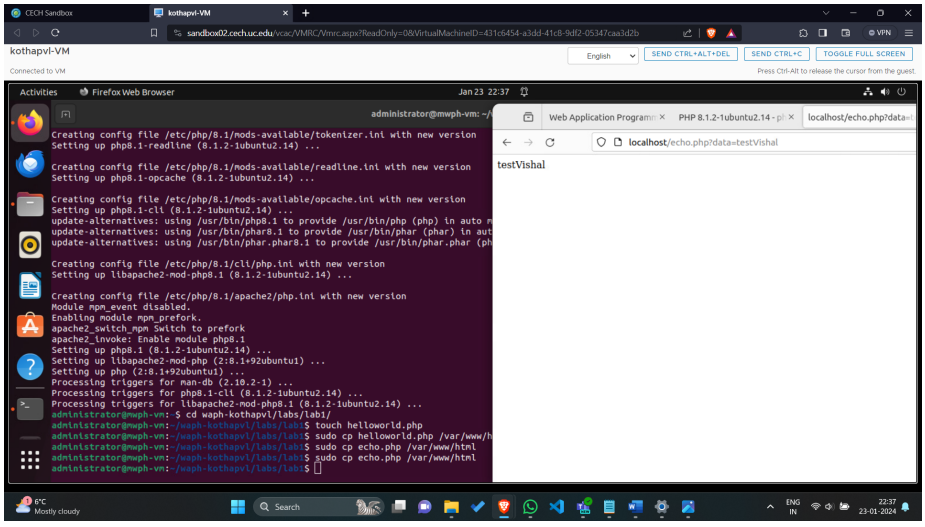
a. I began by enabling PHP functionality within the Ubuntu terminal environment. Within this file, I carefully crafted code designed to output a greeting to the world, my name, and a comprehensive display of server-side PHP information through strategic employment of the echo and phpinfo() functions, respectively. To facilitate web accessibility, I subsequently transferred the file to its designated location within the web server's root directory, identified as "var/www/html". To conclude the process, I initiated a web browser session and navigated to the precise URL http://localhost/helloworld.php, successfully rendering the intended content.



PHP

b. I created a PHP file named “echo.php” using Sublime Text. This PHP code is designed to retrieve the value associated with the “Data” parameter from incoming requests. The retrieval is based on checking either the URL for GET requests or the request body for POST requests. Once the value is obtained, the code echoes (prints) it to the output. In my example, I inputted my name as the value for the variable ‘Data’.

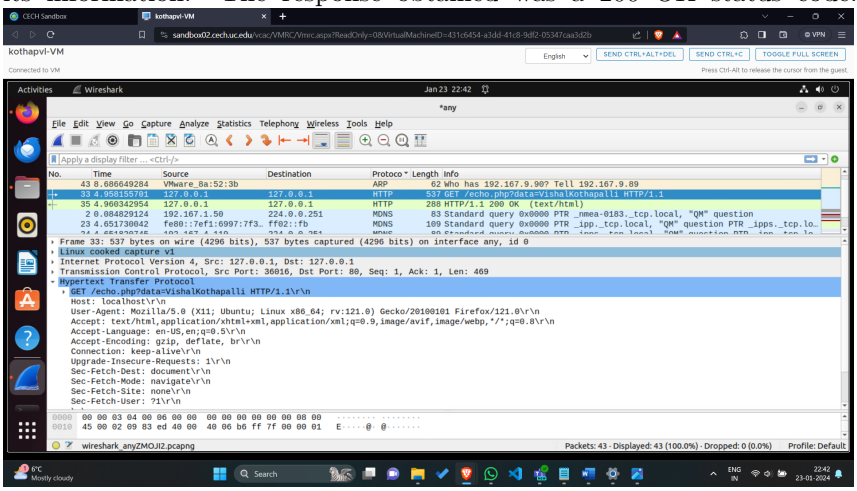
After creating the file, I moved it to the /var/www/html path and accessed the page using the URL - http://localhost/echo.php. However, it’s important to note that since the code can be accessed by users directly, there’s a risk of potential manipulation by users aiming to inject malicious content into the application.



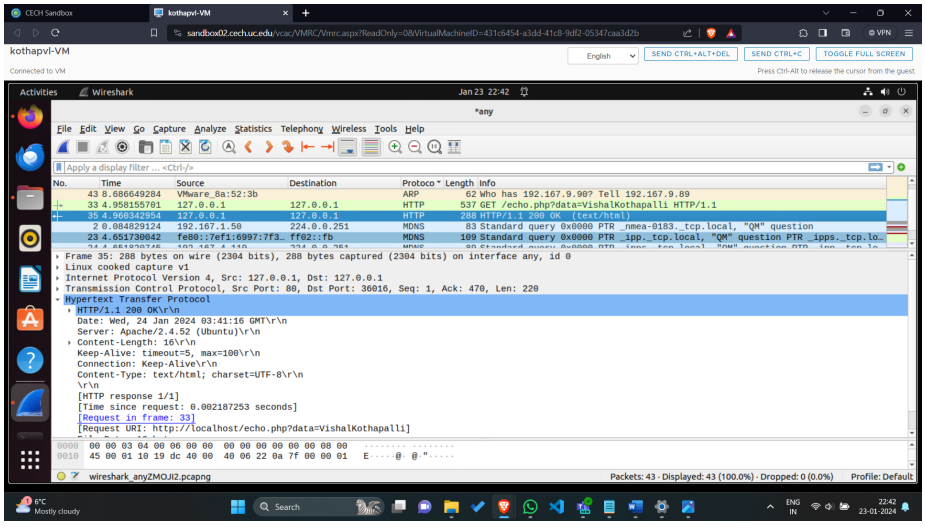
PHP- Name

Task 3 (10 pts). Understanding HTTP GET and POST requests.

a. I utilized Wireshark to capture network traffic while accessing the echo file through a browser. After stopping Wireshark, I filtered the captured traffic to focus on HTTP messages. In the first scenario, I observed a GET request that included the provided data in its information. The response obtained was a 200 OK status code.



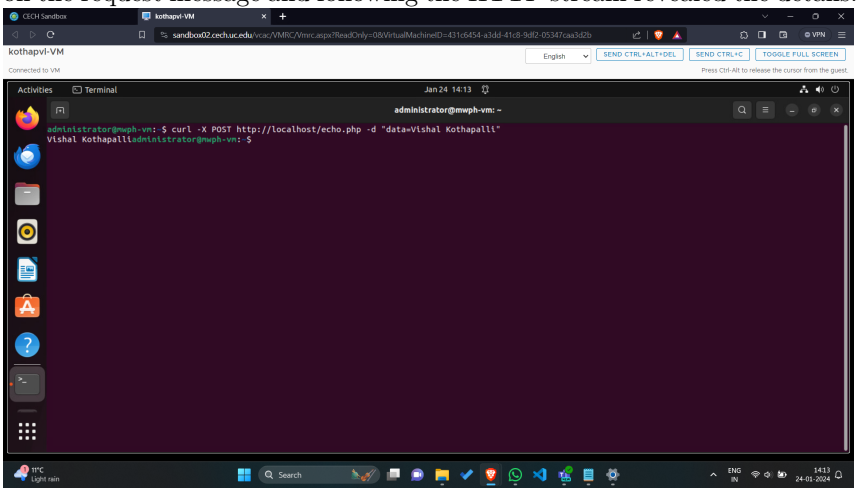
PHP- Request in Wireshark



PHP- Response in Wireshark

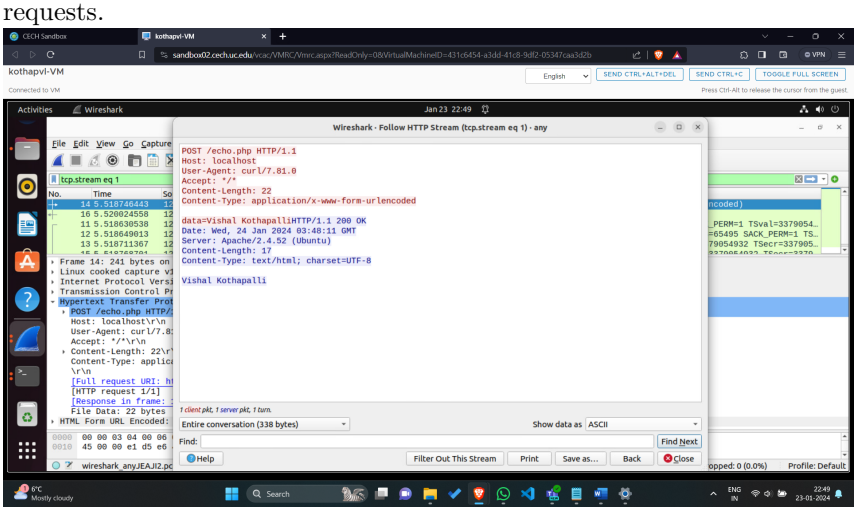
b. In another instance, I opened Wireshark, initiated the browser, and used the curl command to simulate an HTTP POST request to the local server. The

curl command sent the data “Hello World from Tejeswar Reddy Nalijeni” to the PHP script named echo.php. After stopping the Wireshark traffic capture, I filtered HTTP messages and examined the POST request. Clicking on the request message and following the HTTP stream revealed the details.

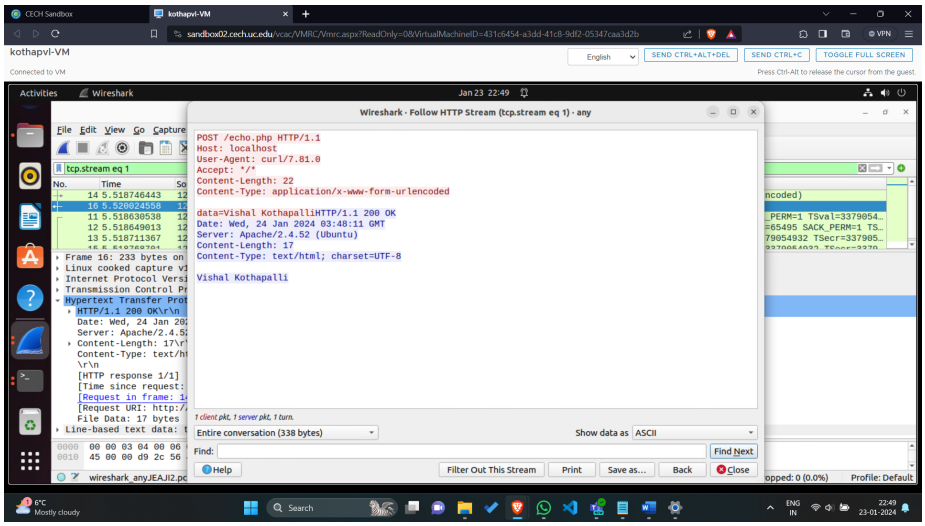


HTTP curl- terminal

c. It’s crucial to note that the key distinction between GET and POST requests lies in data transmission. GET includes parameters in the URL, while POST sends them directly to the request body. Despite this difference, both scenarios resulted in a 200 OK status code, indicating successful requests.



HTTP POST Request in Wireshark



HTTP POST Response in Wireshark