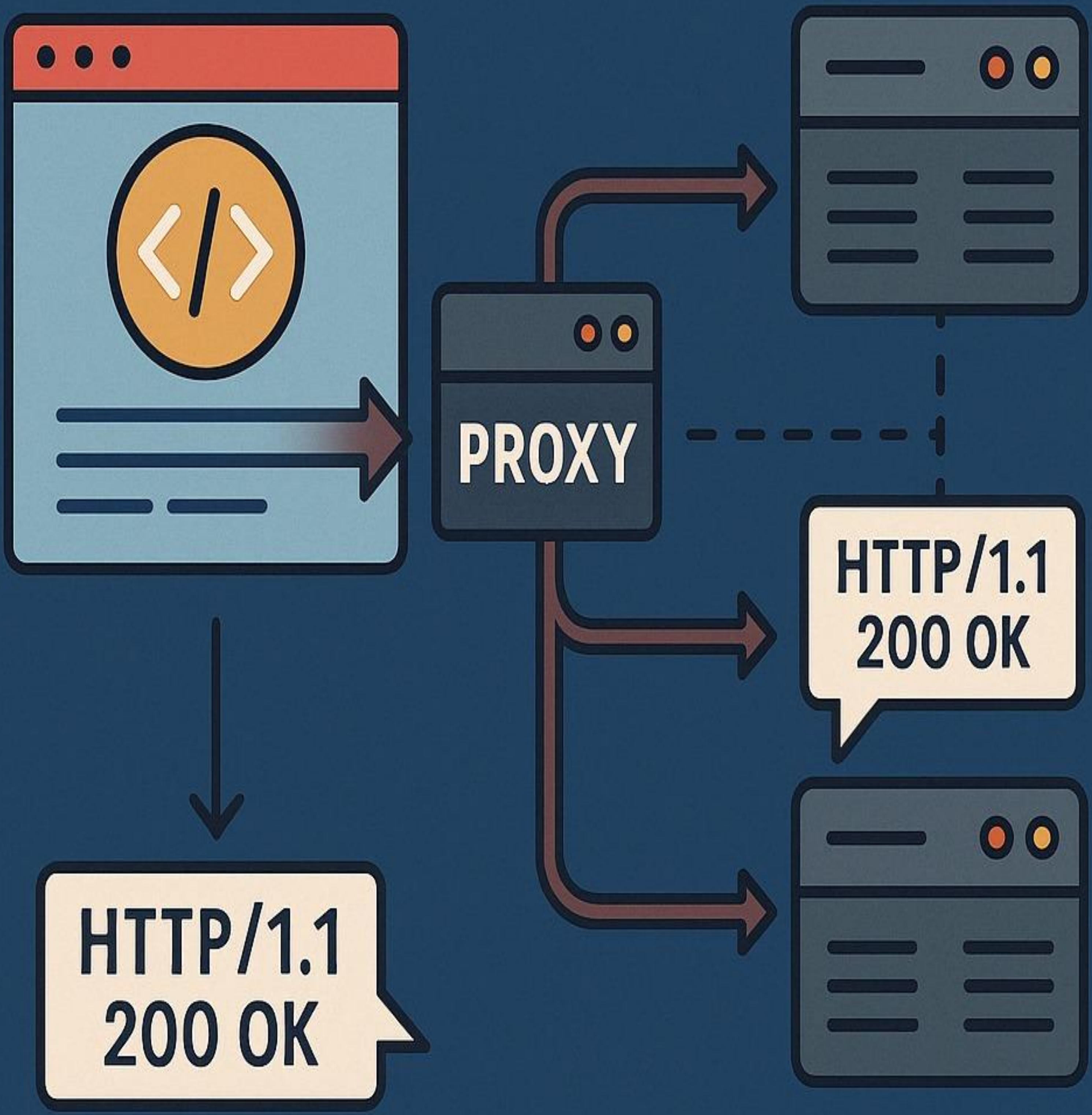
HTTP REQUEST SMUGGLING AND RESPONSE SPLITTING



Sri Lanka Institute of Information Technology

Try Hack Me Room

Hacker's HTTP Heist

Room link

https://tryhackme.com/jr/hackershttpsplitandsmugg

Student Name – Wanasinghe N.K

Student ID – IT23221000

IE2062 - Web Security

B.Sc. (Hons) in information Technology Specializing in Cyber Security

Table of content

Room Overview

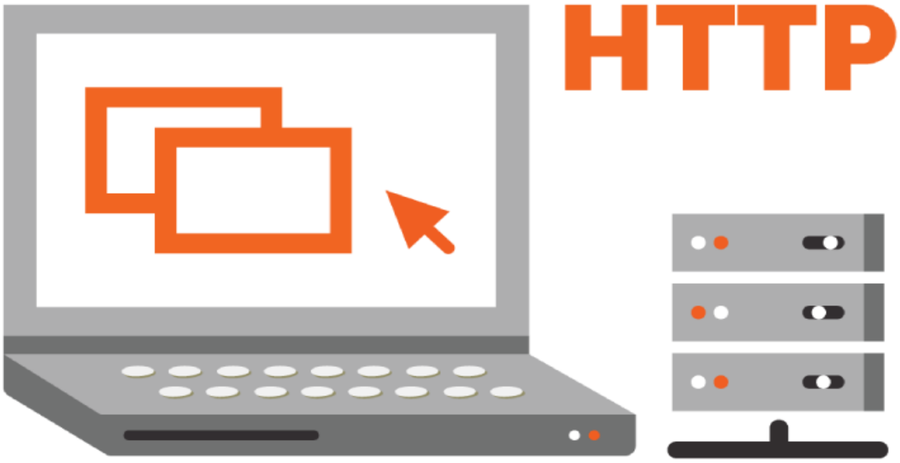
Vulnerability Title

HTTP Response Splitting and Request Smuggling

A Brief Summary of the Room's Objective

The mission of this room on TryHackMe is to uncover the risks of unsecured HTTP handling parts.

By combining the theoretical walk-through, hands-on labs, and a final CTF task, the room improves the audience's knowledge so that they can detect, exploit, and mitigate the vulnerability of HTTP Response Splitting and HTTP Request Smuggling—two web security threats that are simple yet highly impactful.



Explanation of the Key Vulnerabilities/Concepts Covered

HTTP Response Splitting

This vulnerability occurs when an attacker-controlled input field is used in an HTTP response header.

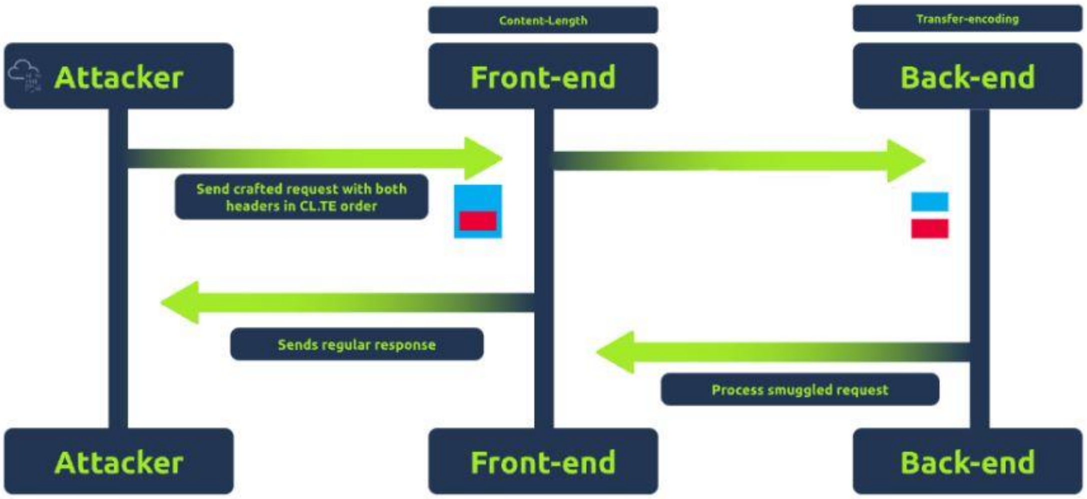
In this case, an attacker could split the HTTP response header into several parts using CRLF characters **(\r\n)** to manipulate headers or even insert malicious parameters.

HTTP Request Smuggling

An even more intricate and complex flaw that results from the mismatch between a front-end and a back-end HTTP parsing logic.

This is called desynchronization.

Adversaries can bypass firewalls and WAFs with hidden payloads by sending specially crafted requests using conflicting headers like **Content-Length** and **Transfer-Encoding.**



Every concept is discussed thoroughly that includes weaponization techniques, the ways it can be executed and, most importantly, prevention.

Introduction to the Room

This TryHackMe room aims to let the user know about and guide them through two of the more esoteric though highly impactful web vulnerabilities: **HTTP Response Splitting and HTTP Request Smuggling**. These techniques may not be the most obvious kinds of security breaches, however they can result in severe consequences such as cache poisoning, cross-site scripting, unauthorized access, and session hijacking.

This room contains all the theoretical knowledge and practice challenges that gives a chance to the users to explore how these harmless-looking headers and requests could be misused into bypassing controls or exploitation of server logic.

At the end of the room, attendees have an idea of how they can occur but more importantly, they will think as a real-world attacker that identifies vulnerabilities, exploits them.

This room is ideal for:

- Beginner to intermediate enthusiasts
- Bug bounty hunters
- Web developers keen on securing their applications
- Most importantly, anyone curious about minor oversights leading to significant security holes

What Inspired the Room (Through my bug bounty experience)

This room derives inspiration from my very first-hand experience in bug bounty hunting where o frequently encountered with web applications and observed their behaviors through tools like Burp Suite. During this time, I did come across a multitude of types of vulnerabilities, but manipulating HTTP requests and responses stood out to my experience.

By constant interaction with raw HTTP traffic, I understood how much minor misconfigurations or trivial details left in request processing can completely compromise security. This experience inspired me to design a room about what I consider to be two underestimated but powerful threats - HTTP Response Splitting and HTTP Request Smuggling.

Therefore, this room is aimed at helping others to have the same learning experience as I did-starting from the stand points of the basics and going through hands-on exploitation to understanding enough to recognize and defend against these attacks in the real world.

Learning Objectives

By completing this room, users will be able to:

Understand HTTP communication basics and the relevance of request/response model.

users will learn how misused headers can hijack or poison server responses.

Identify and take advantage of HTTP Response Splitting in actual applications.

Analyze conflicting headers to find and carry out HTTP Request Smuggling attacks.

Learn effective mitigation strategies, including input sanitization, header validation, and secure proxy configuration.

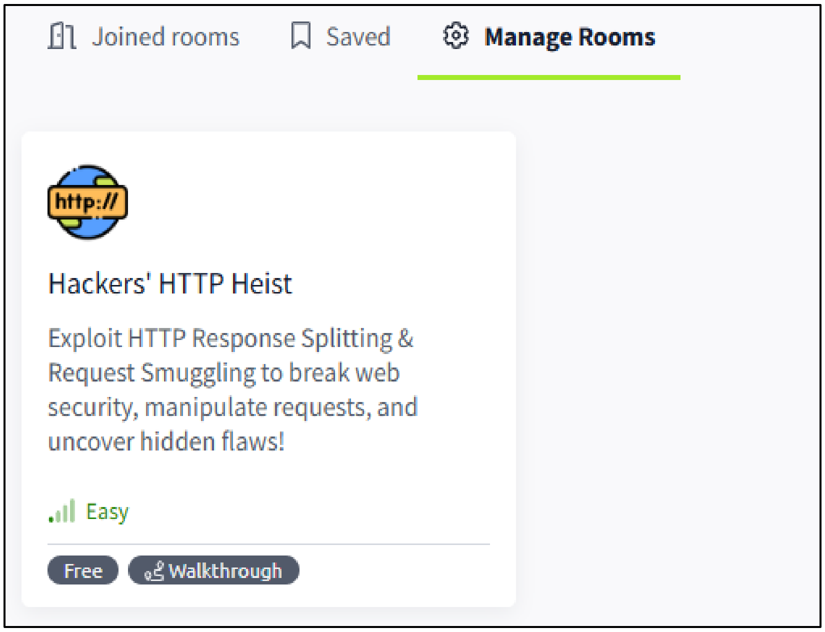
Employ all the methods learned on a final CTF challenge, mimicking actual bug bounty exploitation flow.

Room Structure

This room contains 10 tasks including all the theoretical knowledge and hands-on practicals. In this section I'll break down the creation process of each task.

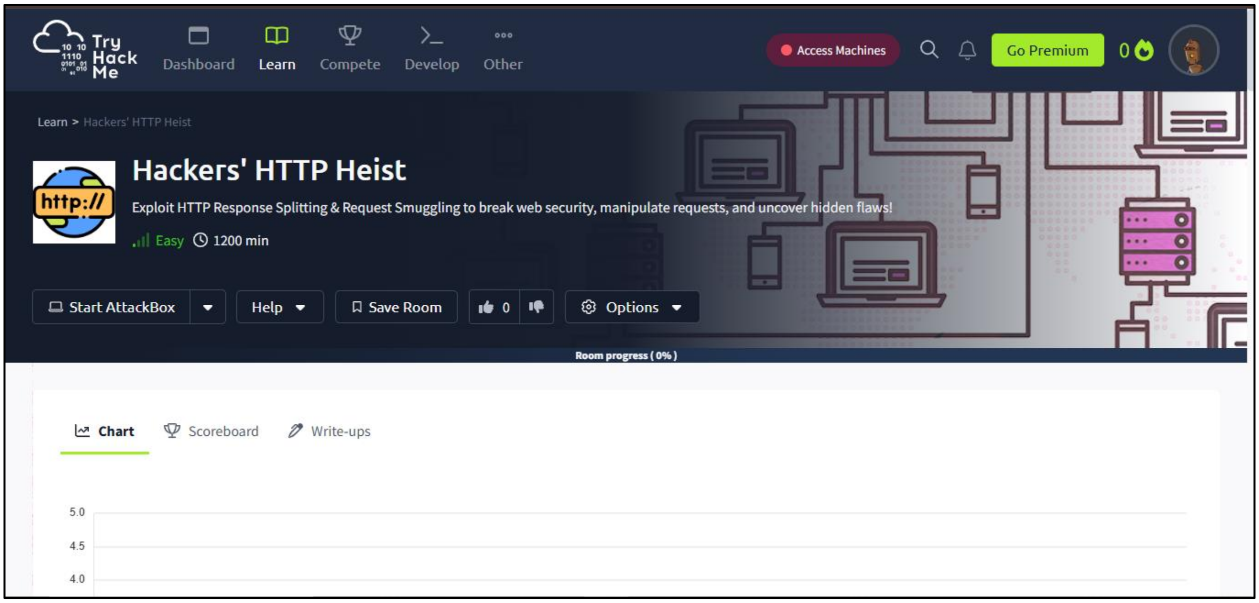
Room cover page

Room name – Hackers' HTTP Heist



Hackers' HTTP Heist

Exploit HTTP Response Splitting & Request Smuggling to break web security, manipulate requests, and uncover hidden flaws!



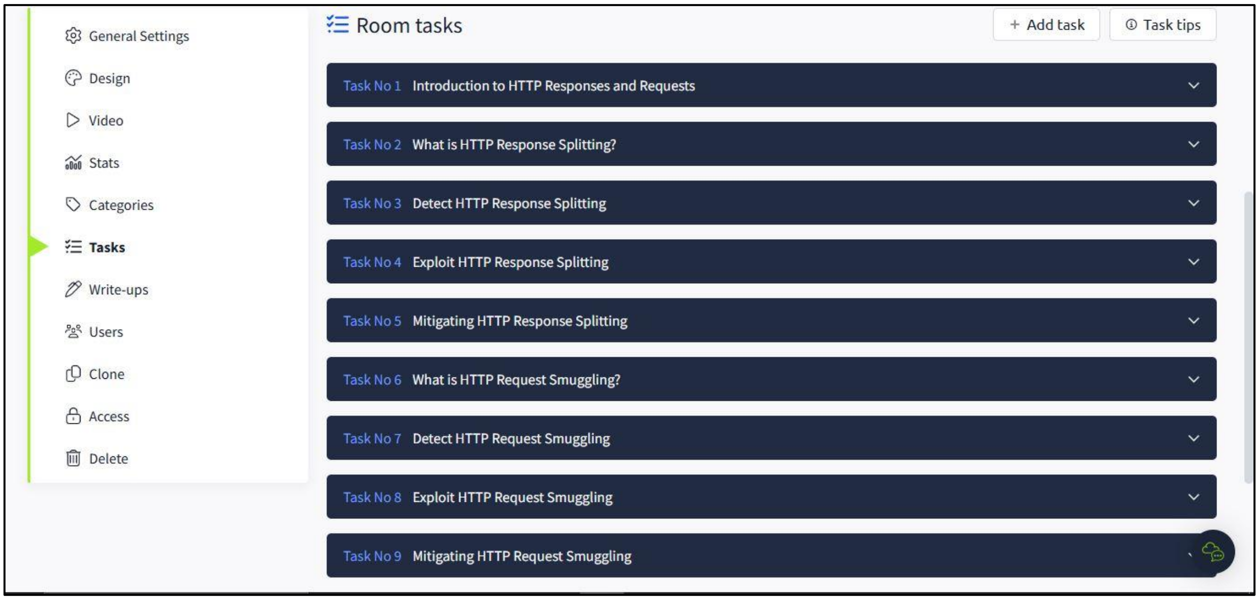
Learn > Hackers' HTTP Heist

Hackers' HTTP Heist

Exploit HTTP Response Splitting & Request Smuggling to break web security, manipulate requests, and uncover hidden flaws!

Easy 1200 min

Tasks



Room tasks

Task No 1 Introduction to HTTP Responses and Requests

Task No 2 What is HTTP Response Splitting?

Task No 3 Detect HTTP Response Splitting

Task No 4 Exploit HTTP Response Splitting

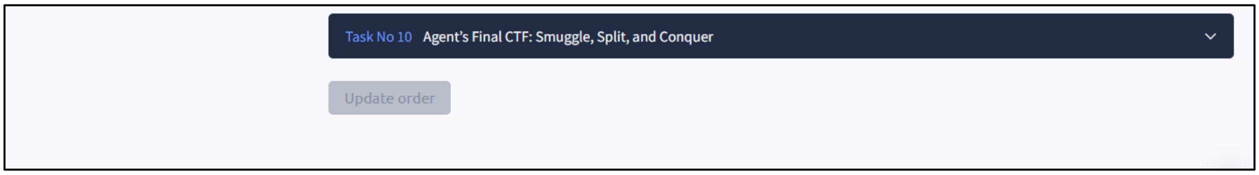
Task No 5 Mitigating HTTP Response Splitting

Task No 6 What is HTTP Request Smuggling?

Task No 7 Detect HTTP Request Smuggling

Task No 8 Exploit HTTP Request Smuggling

Task No 9 Mitigating HTTP Request Smuggling



Task No 10 Agent's Final CTF: Smuggle, Split, and Conquer

Breakdown of each section/challenge

Task 01

Name – Introduction to HTTP Responses and Requests

Objective - To understand the HTTP request-response model and how communication flows between clients and servers.

Type – Theory (Easy)

Task description-

The first task introduces you to the HTTP protocol, which serves as the backbone for web communications. It describes the request-response model, the client (typically a browser) sends an request to the server, and in turn, the server replies with the appropriate content and status information.

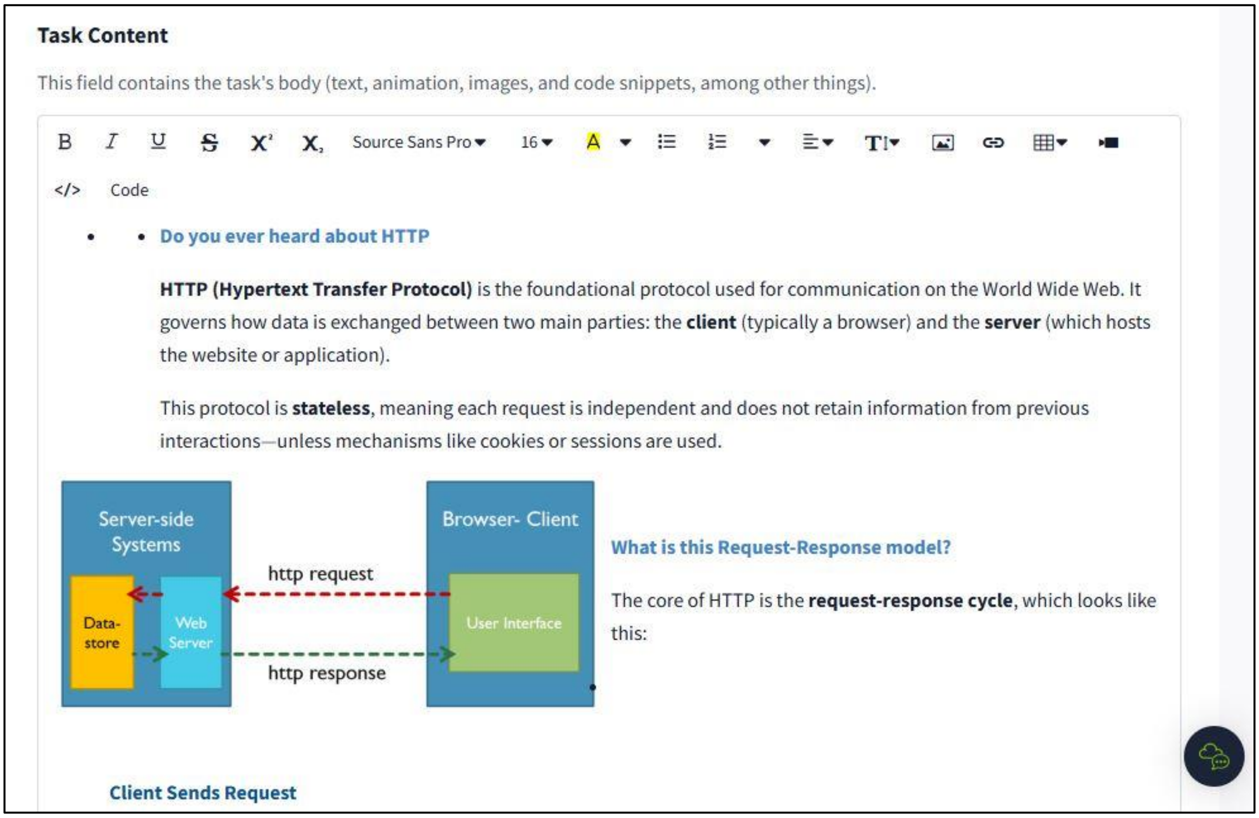
Task Breakdown -

The construction of an HTTP request (headers, URL, and an optional body).

The structure of an HTTP response (status code, headers, and body).

The function of headers, which essentially hold metadata imperative for communication and application logic.

Furthermore, the task describes **proxies** like load balancers and caching servers, which act as intermediaries and can unintentionally become targets or vectors for web-based attacks when poorly configured.



Task Content

This field contains the task's body (text, animation, images, and code snippets, among other things).

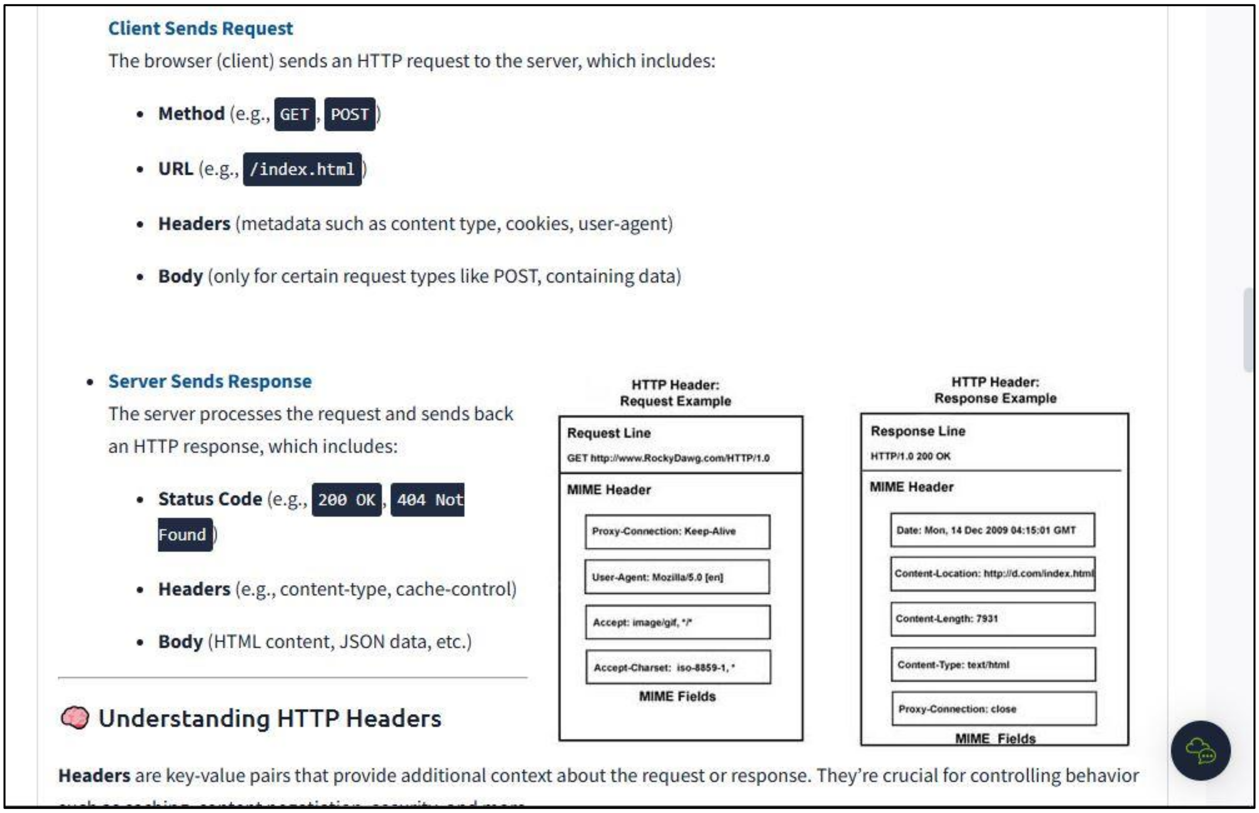
- - **Do you ever heard about HTTP**

    HTTP (Hypertext Transfer Protocol) is the foundational protocol used for communication on the World Wide Web. It governs how data is exchanged between two main parties: the **client** (typically a browser) and the **server** (which hosts the website or application).

    This protocol is **stateless**, meaning each request is independent and does not retain information from previous interactions—unless mechanisms like cookies or sessions are used.

What is this Request-Response model?

The core of HTTP is the **request-response cycle**, which looks like this:



Client Sends Request

The browser (client) sends an HTTP request to the server, which includes:

- **Method** (e.g., `GET`, `POST`)
- **URL** (e.g., `/index.html`)
- **Headers** (metadata such as content type, cookies, user-agent)
- **Body** (only for certain request types like POST, containing data)

- **Server Sends Response**

  The server processes the request and sends back an HTTP response, which includes:

  - **Status Code** (e.g., `200 OK`, `404 Not Found`)
  - **Headers** (e.g., content-type, cache-control)
  - **Body** (HTML content, JSON data, etc.)

HTTP Header:
Request Example

Request Line
GET http://www.RockyDawg.com/HTTP/1.0

MIME Header
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/5.0 [en]
Accept: image/gif, */*
Accept-Charset: iso-8859-1, *
MIME Fields

HTTP Header:
Response Example

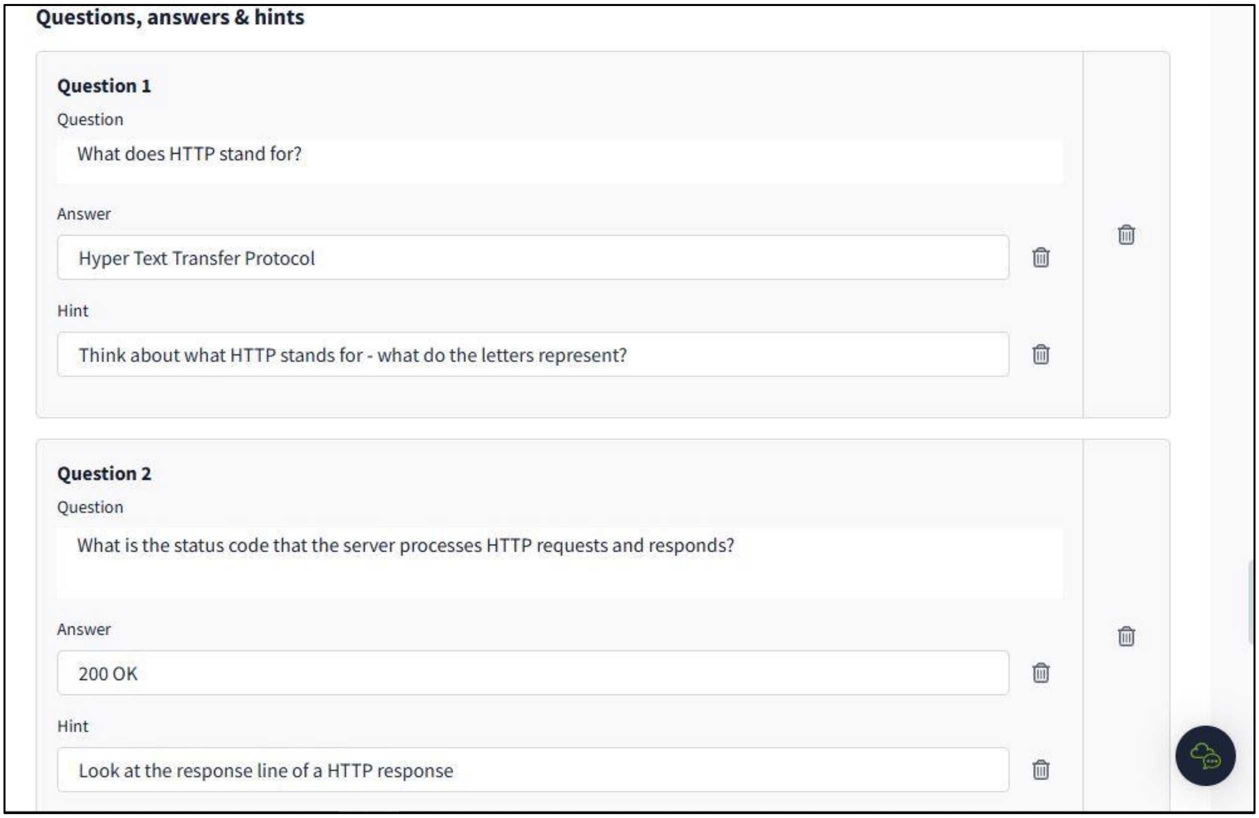
Response Line
HTTP/1.0 200 OK

MIME Header
Date: Mon, 14 Dec 2009 04:15:01 GMT
Content-Location: http://d.com/index.html
Content-Length: 7931
Content-Type: text/html
Proxy-Connection: close
MIME Fields

🧠 Understanding HTTP Headers

Headers are key-value pairs that provide additional context about the request or response. They're crucial for controlling behavior such as caching, content negotiation, security, and more.

At the end this task contains some small questions for the users.



Questions, answers & hints

Question 1

Question

What does HTTP stand for?

Answer

Hyper Text Transfer Protocol

Hint

Think about what HTTP stands for - what do the letters represent?

Question 2

Question

What is the status code that the server processes HTTP requests and responds?

Answer

200 OK

Hint

Look at the response line of a HTTP response

Task 02

Name – What is HTTP Response Splitting?

Objective - users will earn how HTTP Response Splitting works and its security impact.

Type – Theory (Easy)

Task description-

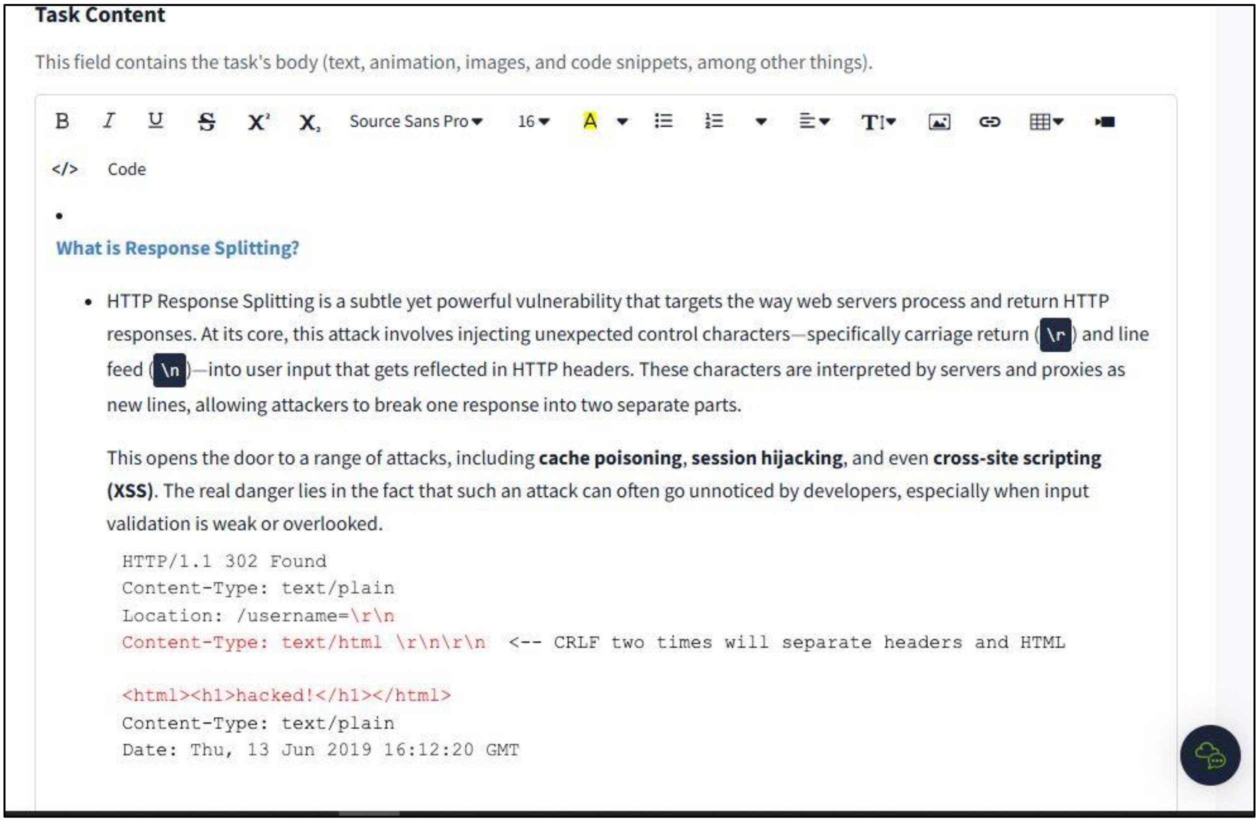
HTTP Response Splitting establishes a form of attack in which an attacker inputs special characters into HTTP headers to split a single response into two. The consequences of this include cache poisoning, session hijacking, and XSS attacks.

Task Breakdown -

Injecting CRLF (\r\n) characters into a header for the purpose of creating a second, malicious response.

Demonstrating how browsers process multiple Set-Cookie headers, which could result in overwriting legitimate session data.

Showing a real-world example where the attacker replaces a valid session ID with his own and impacts.



Task Content

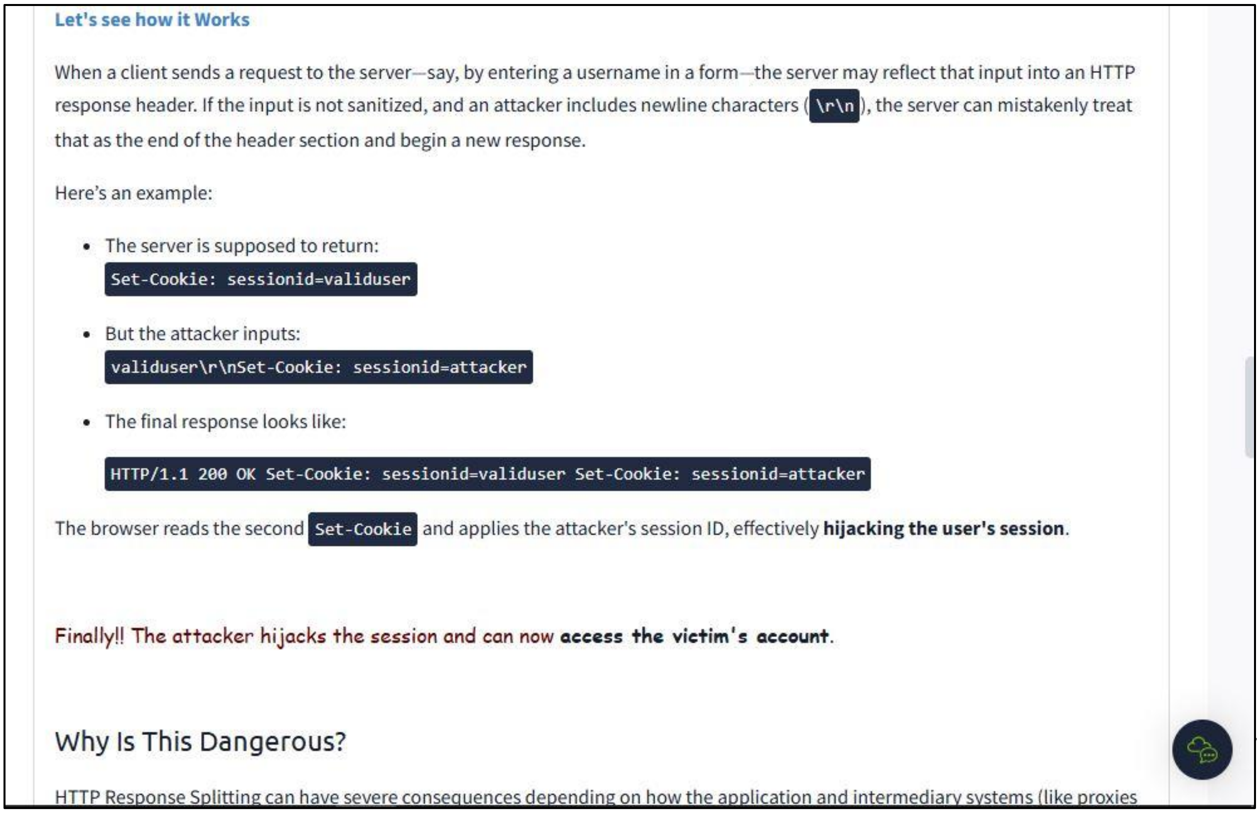
This field contains the task's body (text, animation, images, and code snippets, among other things).

What is Response Splitting?

- HTTP Response Splitting is a subtle yet powerful vulnerability that targets the way web servers process and return HTTP responses. At its core, this attack involves injecting unexpected control characters—specifically carriage return (`\r`) and line feed (`\n`)—into user input that gets reflected in HTTP headers. These characters are interpreted by servers and proxies as new lines, allowing attackers to break one response into two separate parts.

  This opens the door to a range of attacks, including **cache poisoning**, **session hijacking**, and even **cross-site scripting (XSS)**. The real danger lies in the fact that such an attack can often go unnoticed by developers, especially when input validation is weak or overlooked.

```
HTTP/1.1 302 Found
Content-Type: text/plain
Location: /username=\r\n
Content-Type: text/html \r\n\r\n  <-- CRLF two times will separate headers and HTML

<html><h1>hacked!</h1></html>
Content-Type: text/plain
Date: Thu, 13 Jun 2019 16:12:20 GMT
```



Let's see how it Works

When a client sends a request to the server—say, by entering a username in a form—the server may reflect that input into an HTTP response header. If the input is not sanitized, and an attacker includes newline characters (`\r\n`), the server can mistakenly treat that as the end of the header section and begin a new response.

Here's an example:

- The server is supposed to return:
  `Set-Cookie: sessionid=validuser`
- But the attacker inputs:
  `validuser\r\nSet-Cookie: sessionid=attacker`
- The final response looks like:

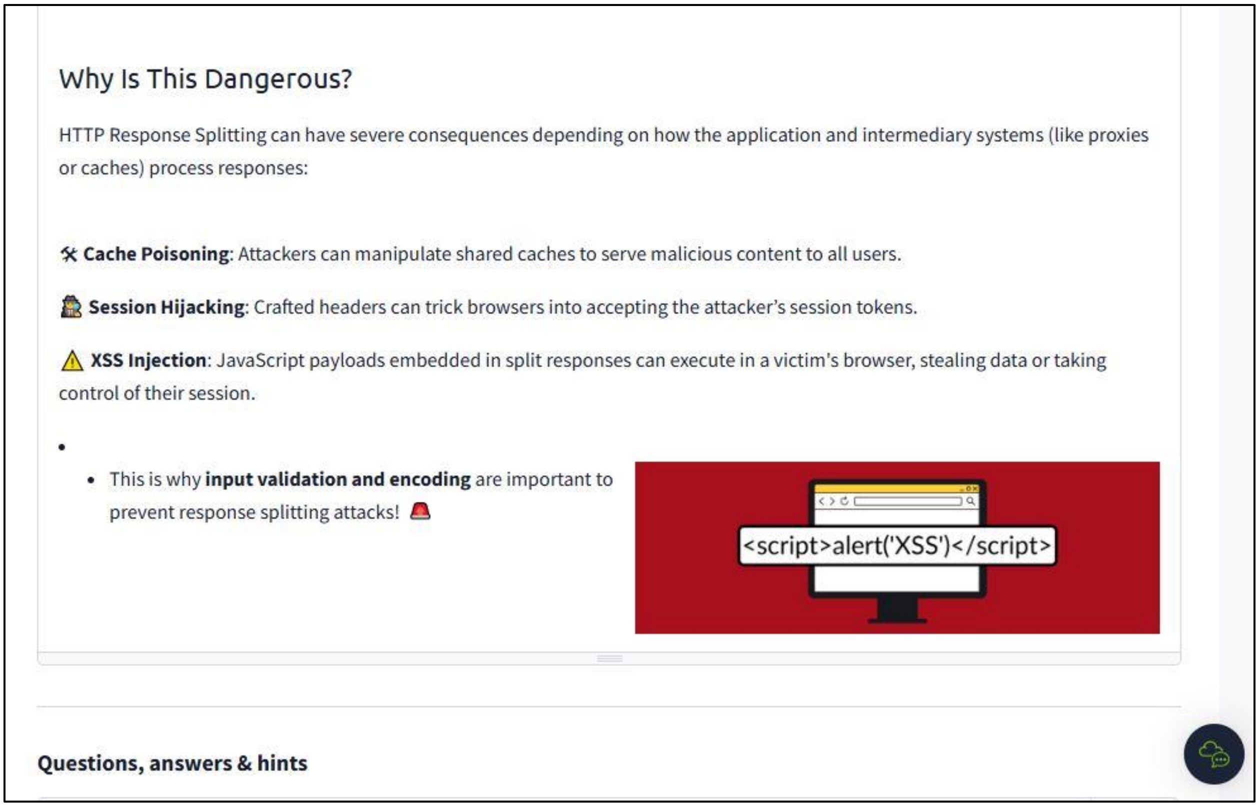
  `HTTP/1.1 200 OK Set-Cookie: sessionid=validuser Set-Cookie: sessionid=attacker`

The browser reads the second `Set-Cookie` and applies the attacker's session ID, effectively **hijacking the user's session**.

Finally!! The attacker hijacks the session and can now **access the victim's account**.

Why Is This Dangerous?

HTTP Response Splitting can have severe consequences depending on how the application and intermediary systems (like proxies



Why Is This Dangerous?

HTTP Response Splitting can have severe consequences depending on how the application and intermediary systems (like proxies or caches) process responses:

🛠 **Cache Poisoning**: Attackers can manipulate shared caches to serve malicious content to all users.

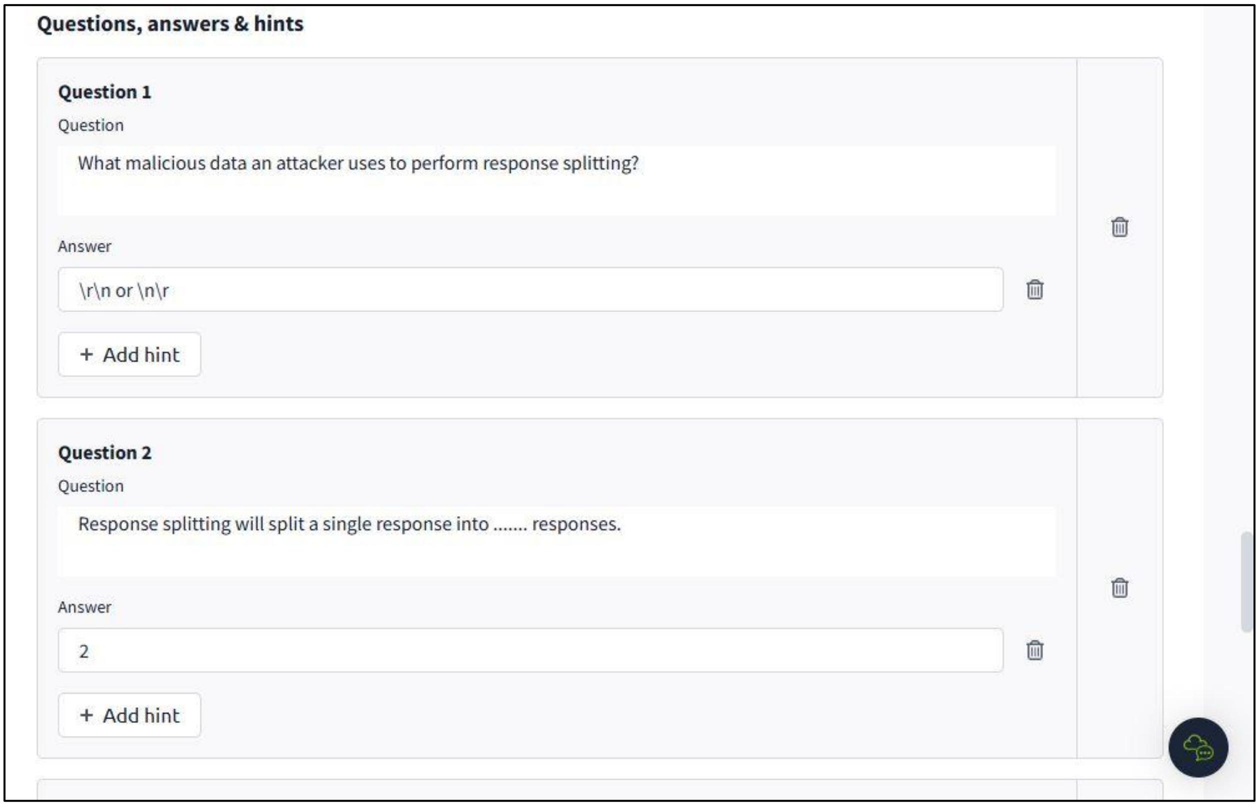
🕵️ **Session Hijacking**: Crafted headers can trick browsers into accepting the attacker's session tokens.

⚠️ **XSS Injection**: JavaScript payloads embedded in split responses can execute in a victim's browser, stealing data or taking control of their session.

-
  - This is why **input validation and encoding** are important to prevent response splitting attacks! 🚨

Questions, answers & hints

At the end this task contains some small questions for the users.



Questions, answers & hints

Question 1

Question

What malicious data an attacker uses to perform response splitting?

Answer

\r\n or \n\r

Question 2

Question

Response splitting will split a single response into .... responses.

Answer

2

Task 03

Name – Let’s detect HTTP Response Splitting

Objective - In this task, users will test if a simple web application is vulnerable to HTTP Response Splitting by injecting header-breaking characters into a URL.

Type – practical (Medium)

Task description-

The task introduces an instance of HTTP response splitting as a result of failing to sanitize user inputs in a web application properly.

A PHP-based demo page (task3.php) simulates the behavior of a vulnerable server to let users manipulate the lang parameter to 'split' headers and simulate setting an extra cookie in an attack scenario while protected and controlled.

The simulation is used since modern web servers (like Apache and PHP) already filter response headers and enforce rules to avoid this.

The lab, therefore, visualizes the effects and offers a greater understanding with a hypothetical situation where those protections would not be there.

Task Breakdown –

First, I installed all the necessary tools like PHP and Apache. I instruct users to use the **pre-build AttackBox in Try Hack Me.**

sudo apt install libapache2-mod-php8.3

sudo a2enmod php8.3

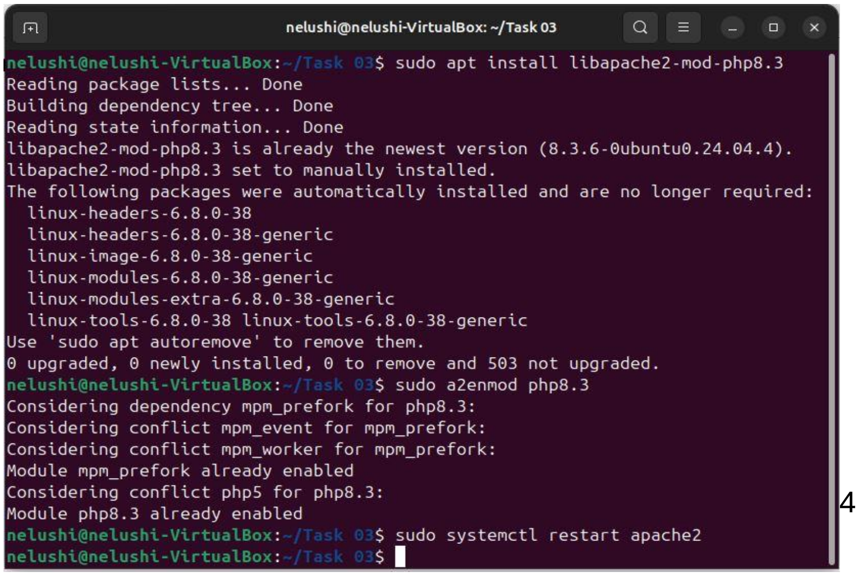
To check the versions

apache2 -v

php -v

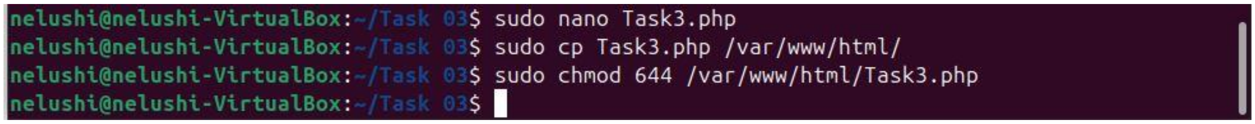
Start apache

Sudo systemctl start apache



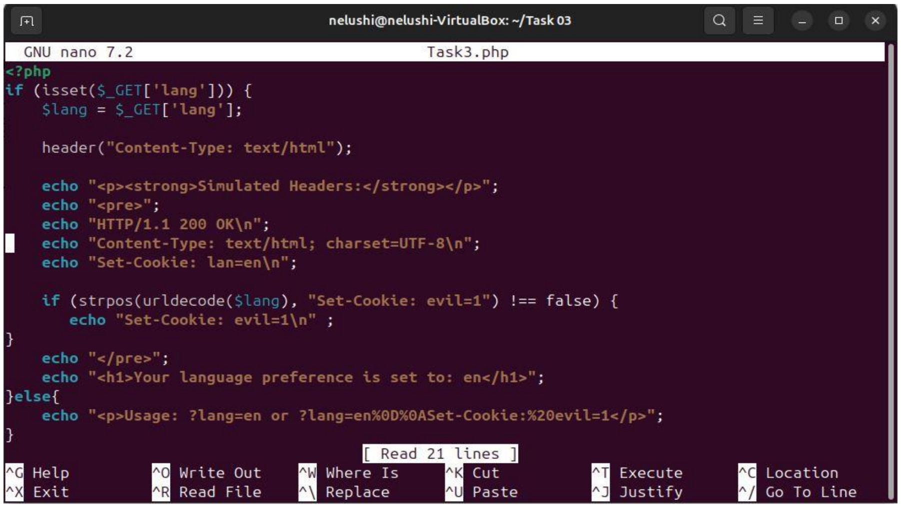
```
nelushi@nelushi-VirtualBox:~/Task 03$ sudo apt install libapache2-mod-php8.3
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
libapache2-mod-php8.3 is already the newest version (8.3.6-0ubuntu0.24.04.4).
libapache2-mod-php8.3 set to manually installed.
The following packages were automatically installed and are no longer required:
  linux-headers-6.8.0-38
  linux-headers-6.8.0-38-generic
  linux-image-6.8.0-38-generic
  linux-modules-6.8.0-38-generic
  linux-modules-extra-6.8.0-38-generic
  linux-tools-6.8.0-38 linux-tools-6.8.0-38-generic
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 503 not upgraded.
nelushi@nelushi-VirtualBox:~/Task 03$ sudo a2enmod php8.3
Considering dependency mpm_prefork for php8.3:
Considering conflict mpm_event for mpm_prefork:
Considering conflict mpm_worker for mpm_prefork:
Module mpm_prefork already enabled
Considering conflict php5 for php8.3:
Module php8.3 already enabled
nelushi@nelushi-VirtualBox:~/Task 03$ sudo systemctl restart apache2
nelushi@nelushi-VirtualBox:~/Task 03$ 
```

Gave permissions and created a php file and moved the file to apache root directory (/var/www/html/).



```
nelushi@nelushi-VirtualBox:~/Task 03$ sudo nano Task3.php
nelushi@nelushi-VirtualBox:~/Task 03$ sudo cp Task3.php /var/www/html/
nelushi@nelushi-VirtualBox:~/Task 03$ sudo chmod 644 /var/www/html/Task3.php
nelushi@nelushi-VirtualBox:~/Task 03$ 
```

Next, I created the task3.php file.



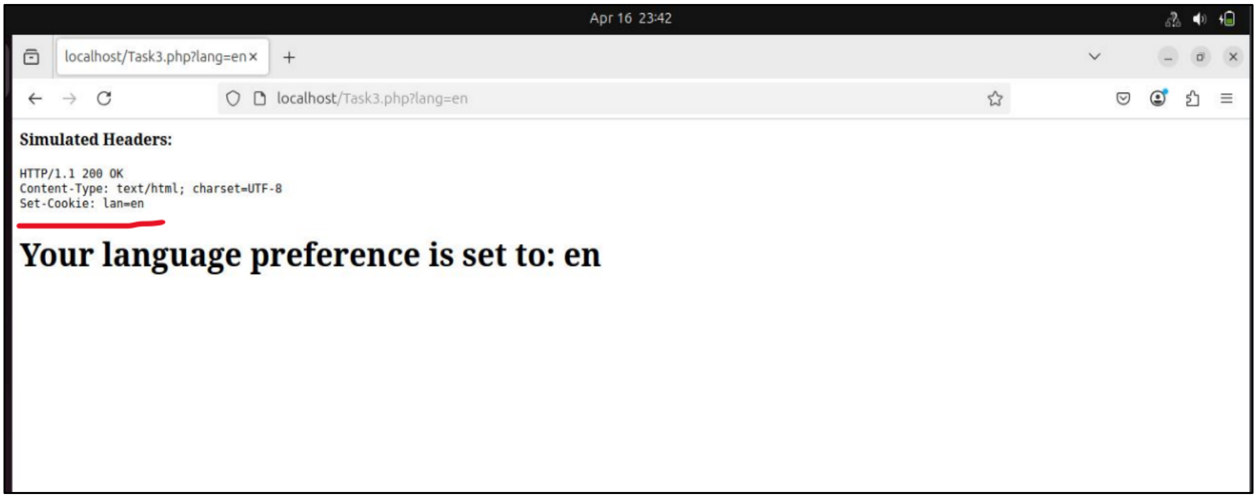
```
  GNU nano 7.2                              Task3.php
<?php
if (isset($_GET['lang'])) {
    $lang = $_GET['lang'];

    header("Content-Type: text/html");

    echo "<p><strong>Simulated Headers:</strong></p>";
    echo "<pre>";
    echo "HTTP/1.1 200 OK\n";
    echo "Content-Type: text/html; charset=UTF-8\n";
    echo "Set-Cookie: lan=en\n";

    if (strpos(urldecode($lang), "Set-Cookie: evil=1") !== false) {
       echo "Set-Cookie: evil=1\n" ;
}
    echo "</pre>";
    echo "<h1>Your language preference is set to: en</h1>";
}else{
    echo "<p>Usage: ?lang=en or ?lang=en%0D%0ASet-Cookie:%20evil=1</p>";
}
                                    [ Read 21 lines ]
^G Help        ^O Write Out   ^W Where Is    ^K Cut         ^T Execute     ^C Location
^X Exit        ^R Read File   ^\ Replace     ^U Paste       ^J Justify     ^/ Go To Line
```

Expected outcome –

When you access http://localhost/task3.php?lang=en in the browser,

You will see simulated headers and your language preference displayed as normal.

Set-Cookie: lan=en



Simulated Headers:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Set-Cookie: lan=en
```

Your language preference is set to: en

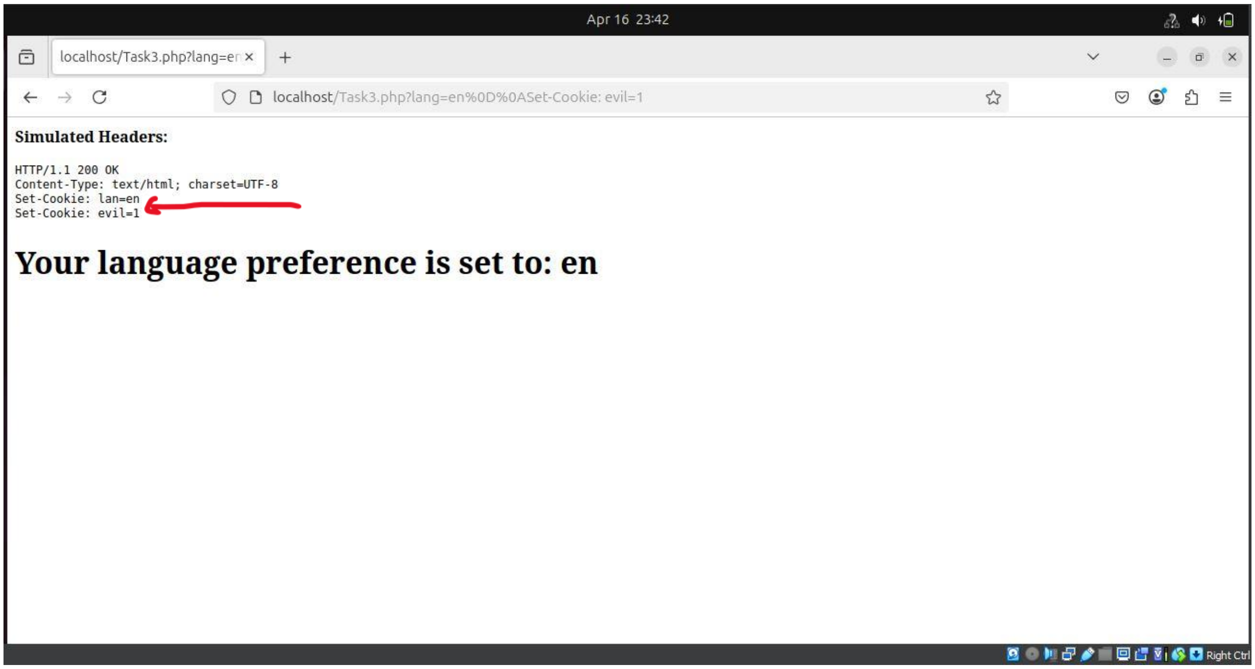
Next, I injected a CRLF and fake header with this,

http://localhost/task3.php?lang=en%0D%0ASet-Cookie:%20evil=1

Then you will see **two simulated Set-Cookie lines** displayed in the browser output.

Set-Cookie: lan=en

Set-Cookie: evil=1

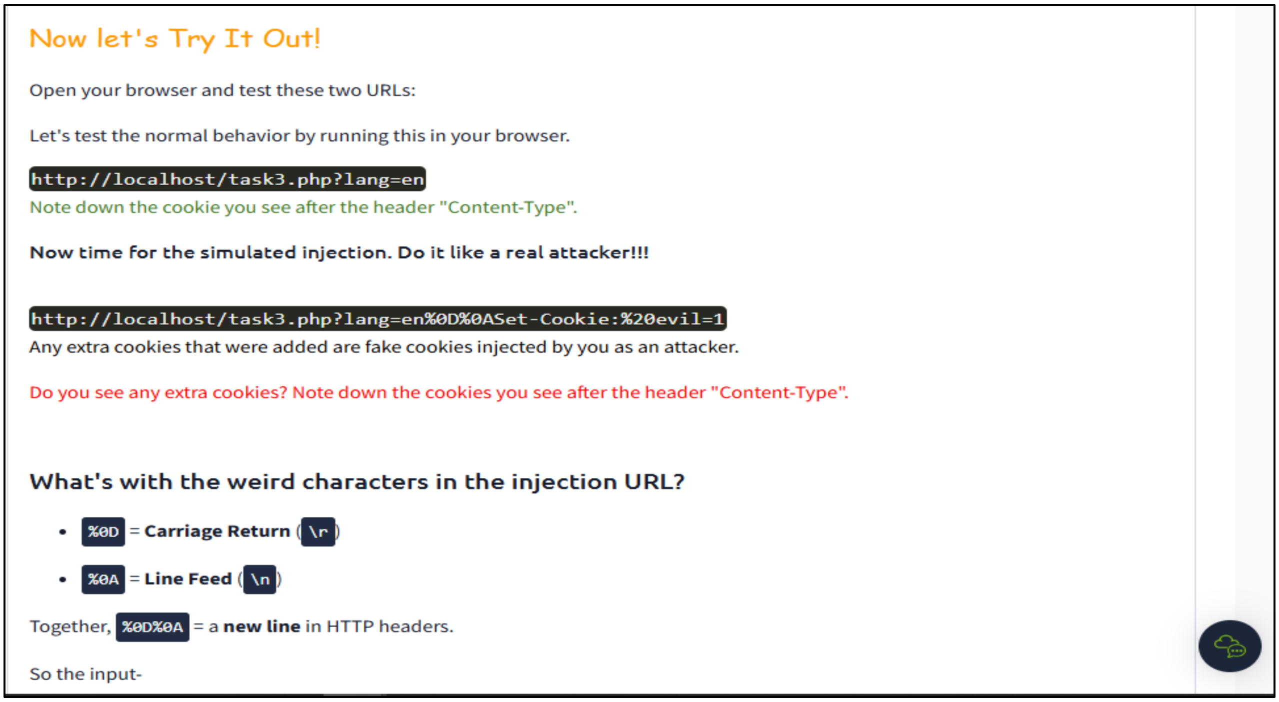

Apr 16 23:42
localhost/Task3.php?lang=en
localhost/Task3.php?lang=en%0D%0ASet-Cookie: evil=1
Right Ctrl

Simulated Headers:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Set-Cookie: lan=en
Set-Cookie: evil=1
```

Your language preference is set to: en

HTTP Response Splitting is another concept under which injecting CRLF characters (%0D%0A) allows attackers to break into HTTP headers and create their own, ultimately leading to cache poisoning, XSS, or session fixation.



Now let's Try It Out!

Open your browser and test these two URLs:

Let's test the normal behavior by running this in your browser.

`http://localhost/task3.php?lang=en`

Note down the cookie you see after the header "Content-Type".

Now time for the simulated injection. Do it like a real attacker!!!

`http://localhost/task3.php?lang=en%0D%0ASet-Cookie:%20evil=1`

Any extra cookies that were added are fake cookies injected by you as an attacker.

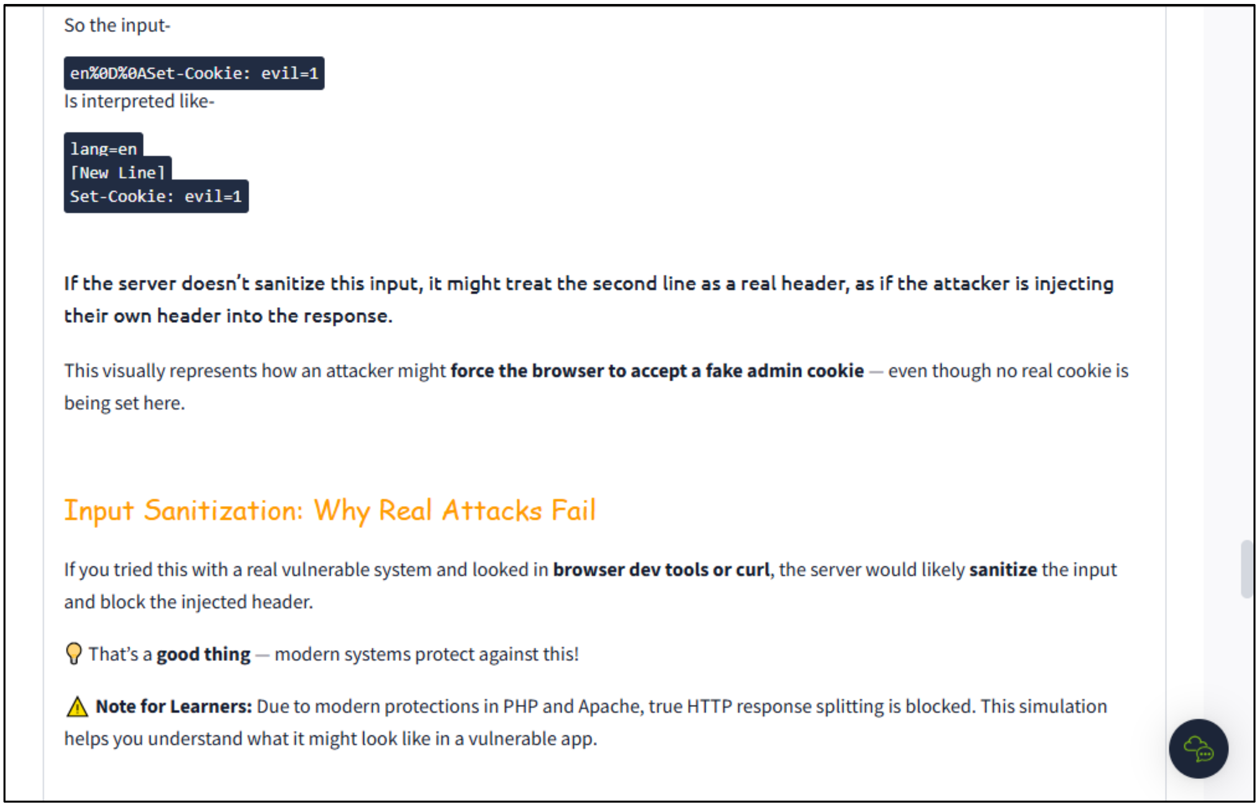
Do you see any extra cookies? Note down the cookies you see after the header "Content-Type".

What's with the weird characters in the injection URL?

- `%0D` = **Carriage Return** (`\r`)
- `%0A` = **Line Feed** (`\n`)

Together, `%0D%0A` = a **new line** in HTTP headers.

So the input-



So the input-

```
en%0D%0ASet-Cookie: evil=1
```

Is interpreted like-

```
lang=en
[New Line]
Set-Cookie: evil=1
```

If the server doesn't sanitize this input, it might treat the second line as a real header, as if the attacker is injecting their own header into the response.

This visually represents how an attacker might **force the browser to accept a fake admin cookie** — even though no real cookie is being set here.

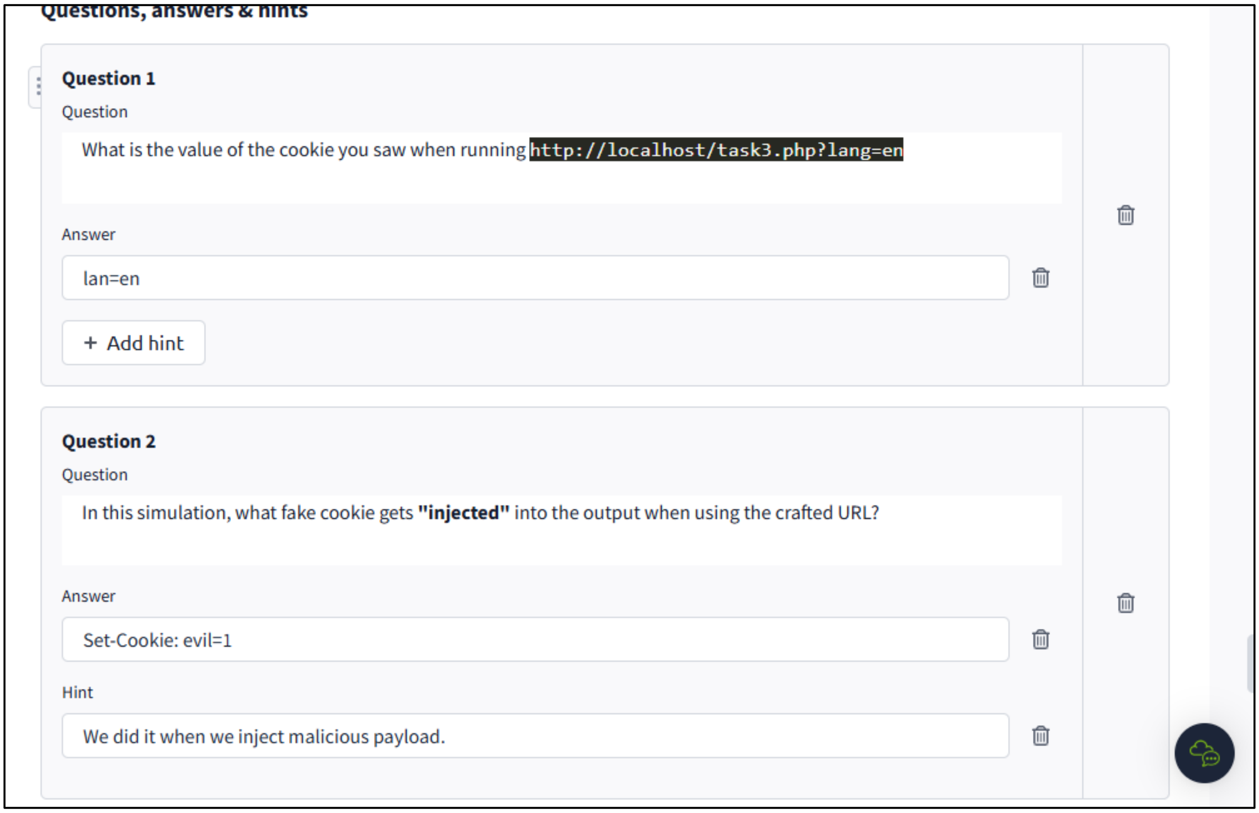
Input Sanitization: Why Real Attacks Fail

If you tried this with a real vulnerable system and looked in **browser dev tools or curl**, the server would likely **sanitize** the input and block the injected header.

💡 That's a **good thing** — modern systems protect against this!

⚠️ **Note for Learners:** Due to modern protections in PHP and Apache, true HTTP response splitting is blocked. This simulation helps you understand what it might look like in a vulnerable app.

At the end this task contains some small questions for the users.



Questions, answers & hints

Question 1

Question

What is the value of the cookie you saw when running `http://localhost/task3.php?lang=en`

Answer

lan=en

Question 2

Question

In this simulation, what fake cookie gets **"injected"** into the output when using the crafted URL?

Answer

Set-Cookie: evil=1

Hint

We did it when we inject malicious payload.

Task 04

Name – Let’s exploit HTTP Response Splitting

Objective - Users will see how this vulnerability could be exploited in a real-world scenario — maybe to escalate privileges, steal cookies, or bypass login checks.

Type – practical (Medium)

Task description-
In this challenge, we fully exploit HTTP Response Splitting to create an unauthorized admin cookie, building on the earlier task’s simulation of HTTP header injection.

Without ever legitimately logging in, the attacker can get admin access by bypassing authentication and altering the response.

Task Breakdown -

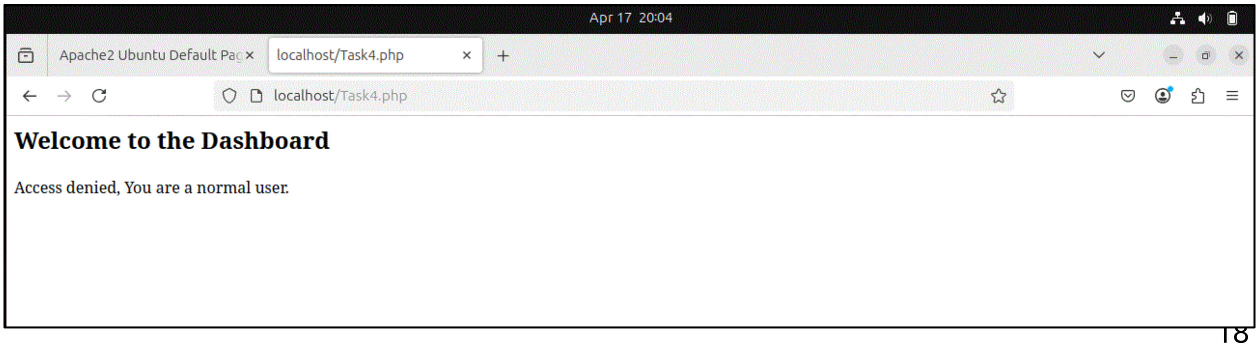
Description of the Exploit Scenario

In Task 3, we used unsanitized input to simulate setting a second cookie (evil=1). We'll use this method to actively trick the server into considering us as an admin in Task 4.

- The first thing we do is change the lang parameter in task3.php to insert a fake **Set-Cookie: evil=1** header.
- This causes the browser to store an illegal cookie.
- The script looks for this forged cookie when it visits task4.php.
- The script provides admin access if it is present **even though the user is not properly authenticated.**

Expected outcome –

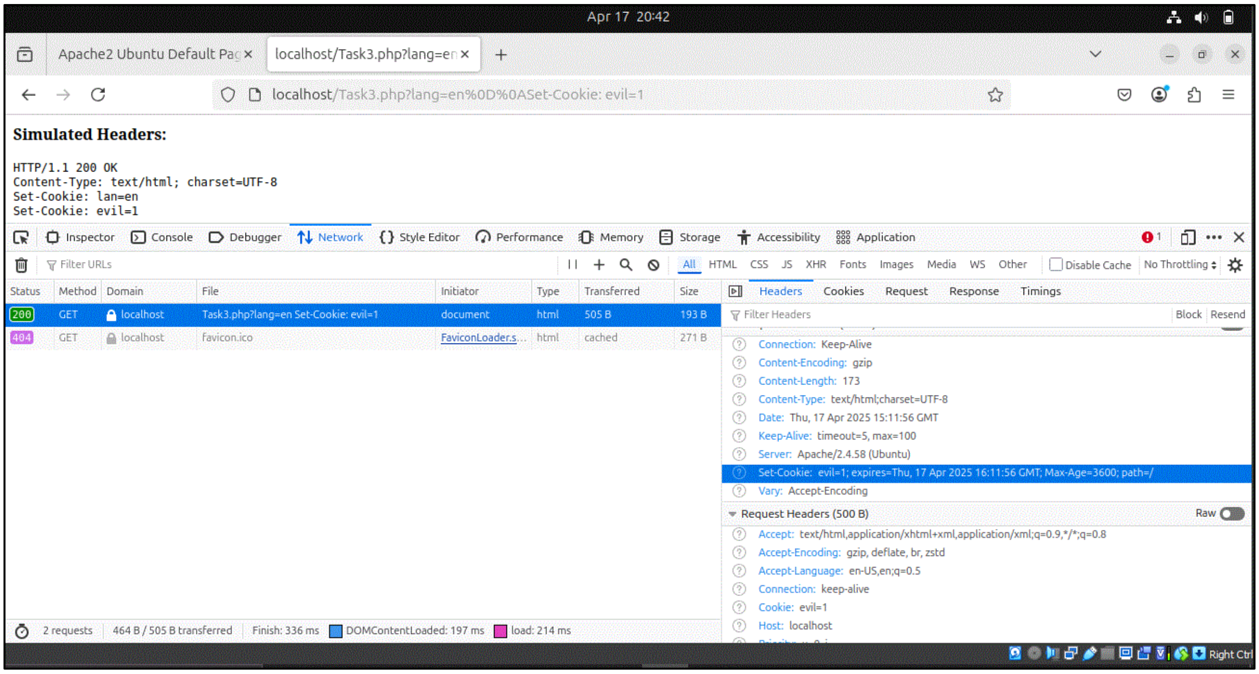
1. Visit Task4.php normally – You are treated as a normal user and no admin access.



Welcome to the Dashboard

Access denied, You are a normal user.

2. Next, inject a cookie via http://localhost/task3.php?lang=en%0D%0ASet-Cookie:%20evil=1 – A 2nd cookie evil=1 is simulated in the HTTP response.



Apr 17 20:42
Apache2 Ubuntu Default Pag
localhost/Task3.php?lang=en
localhost/Task3.php?lang=en%0D%0ASet-Cookie: evil=1
Inspector
Console
Debugger
Network
Style Editor
Performance
Memory
Storage
Accessibility
Application
Filter URLs
All HTML CSS JS XHR Fonts Images Media WS Other
Disable Cache
No Throttling
Headers Cookies Request Response Timings
Filter Headers
Block Resend
Raw
Right Ctrl

Simulated Headers:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Set-Cookie: lan=en
Set-Cookie: evil=1
```

Status	Method	Domain	File	Initiator	Type	Transferred	Size
200	GET	localhost	Task3.php?lang=en Set-Cookie: evil=1	document	html	505 B	193 B
404	GET	localhost	favicon.ico	FaviconLoader.s...	html	cached	271 B

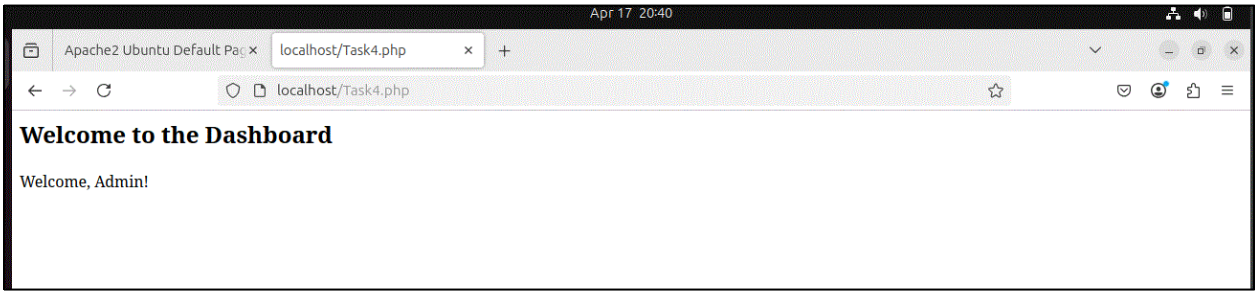
2 requests | 464 B / 505 B transferred | Finish: 336 ms | DOMContentLoaded: 197 ms | load: 214 ms

```
Connection: Keep-Alive
Content-Encoding: gzip
Content-Length: 173
Content-Type: text/html;charset=UTF-8
Date: Thu, 17 Apr 2025 15:11:56 GMT
Keep-Alive: timeout=5, max=100
Server: Apache/2.4.58 (Ubuntu)
Set-Cookie: evil=1; expires=Thu, 17 Apr 2025 16:11:56 GMT; Max-Age=3600; path=/
Vary: Accept-Encoding
```

Request Headers (500 B)

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
Cookie: evil=1
Host: localhost
```

3. Revisit Task4.php after injection – You are welcomed as an admin because of the fake cookie evil=1.



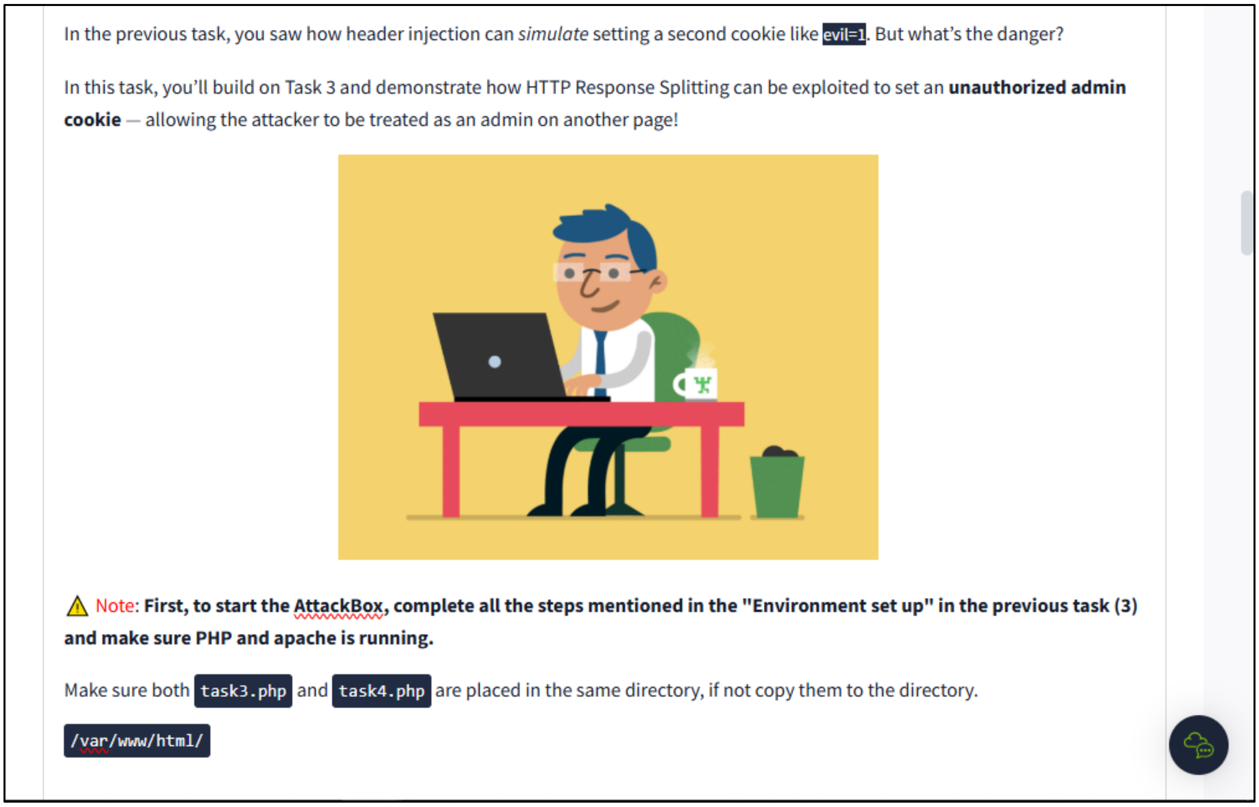
Welcome to the Dashboard

Welcome, Admin!

This task shows how dangerous the header manipulation can be.
Even everything seems normal in the browser, **fake cookies** can allow attackers to

- Bypass login mechanisms
- Access restricted areas
- Perform unauthorized actions **without** a valid session!

Modern web servers usually sanitize such injections, but if input validation is poor, vulnerabilities like this can still be exploited.



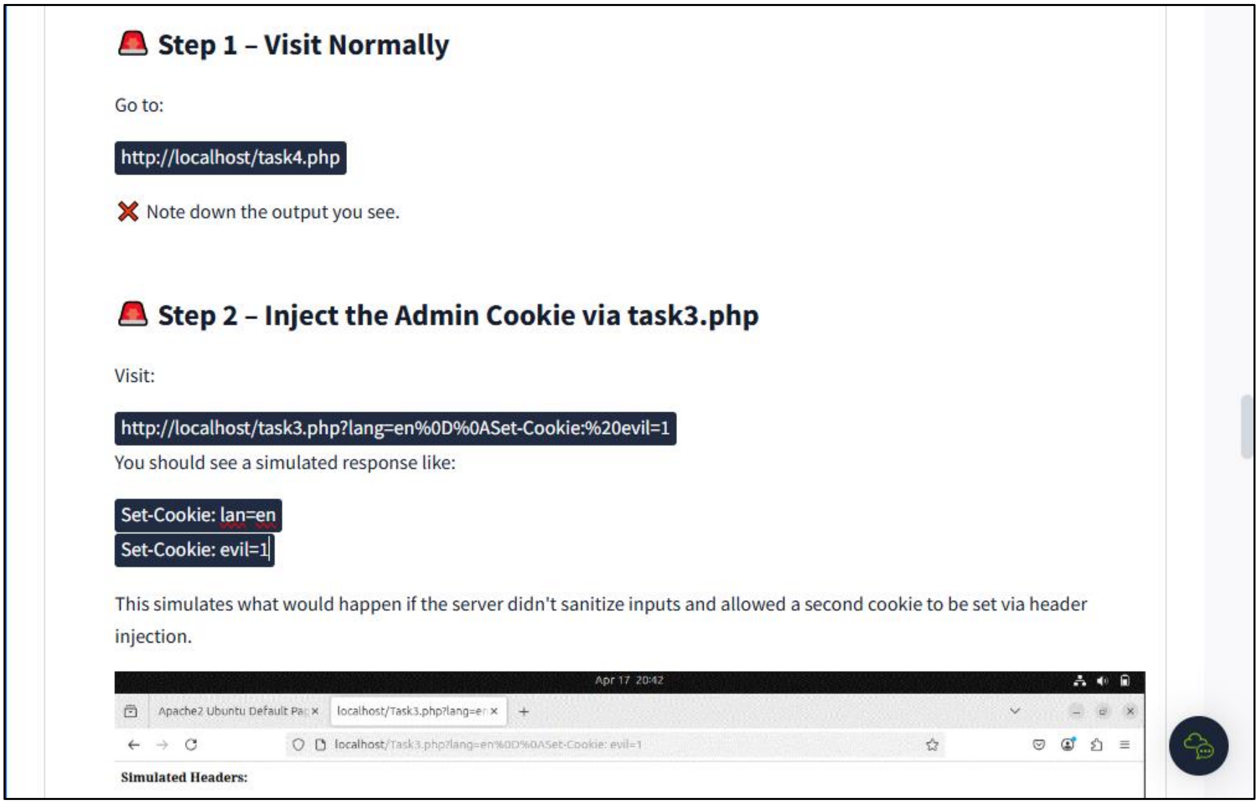
In the previous task, you saw how header injection can *simulate* setting a second cookie like `evil=1`. But what's the danger?

In this task, you'll build on Task 3 and demonstrate how HTTP Response Splitting can be exploited to set an **unauthorized admin cookie** — allowing the attacker to be treated as an admin on another page!

⚠ Note: **First, to start the AttackBox, complete all the steps mentioned in the "Environment set up" in the previous task (3) and make sure PHP and apache is running.**

Make sure both `task3.php` and `task4.php` are placed in the same directory, if not copy them to the directory.

`/var/www/html/`



Step 1 – Visit Normally

Go to:

`http://localhost/task4.php`

❌ Note down the output you see.

Step 2 – Inject the Admin Cookie via task3.php

Visit:

`http://localhost/task3.php?lang=en%0D%0ASet-Cookie:%20evil=1`

You should see a simulated response like:

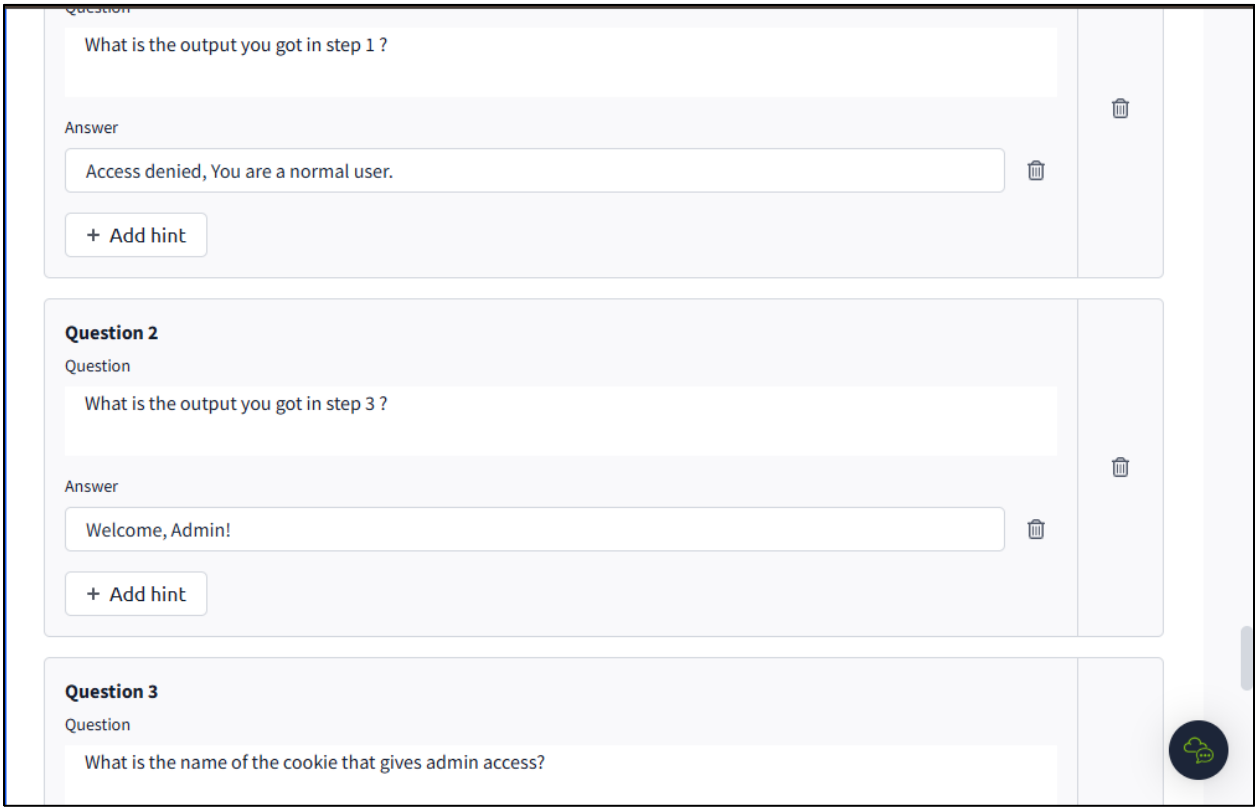
```
Set-Cookie: lan=en
Set-Cookie: evil=1
```

This simulates what would happen if the server didn't sanitize inputs and allowed a second cookie to be set via header injection.

localhost/Task3.php?lang=en%0D%0ASet-Cookie: evil=1

Simulated Headers:

At the end this task contains some small questions for the users.



Question

What is the output you got in step 1 ?

Answer

Access denied, You are a normal user.

Question 2

Question

What is the output you got in step 3 ?

Answer

Welcome, Admin!

Question 3

Question

What is the name of the cookie that gives admin access?

Task 05

Name – Mitigating HTTP Response Splitting

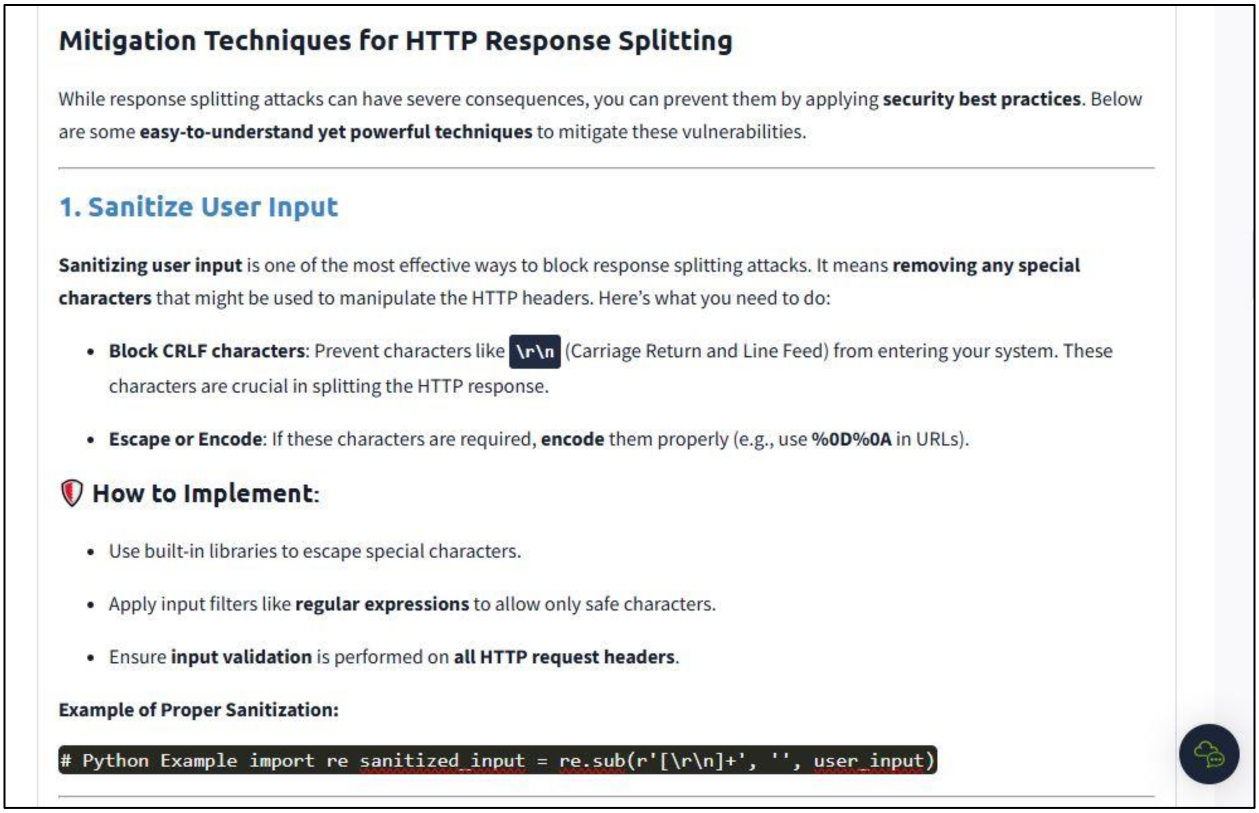
Objective – Users will learn and implement effective security measures to prevent HTTP Response Splitting attacks.

Type – Theory (Easy)

Task description-

This task focuses on mitigation techniques for HTTP Response Splitting. It highlights how developers can protect web applications from these types of attacks by applying secure coding practices, server configurations, and security tools.

Task Breakdown – The task focuses on various kinds of mitigation methods and their explanations.



Mitigation Techniques for HTTP Response Splitting

While response splitting attacks can have severe consequences, you can prevent them by applying **security best practices**. Below are some **easy-to-understand yet powerful techniques** to mitigate these vulnerabilities.

1. Sanitize User Input

Sanitizing user input is one of the most effective ways to block response splitting attacks. It means **removing any special characters** that might be used to manipulate the HTTP headers. Here's what you need to do:

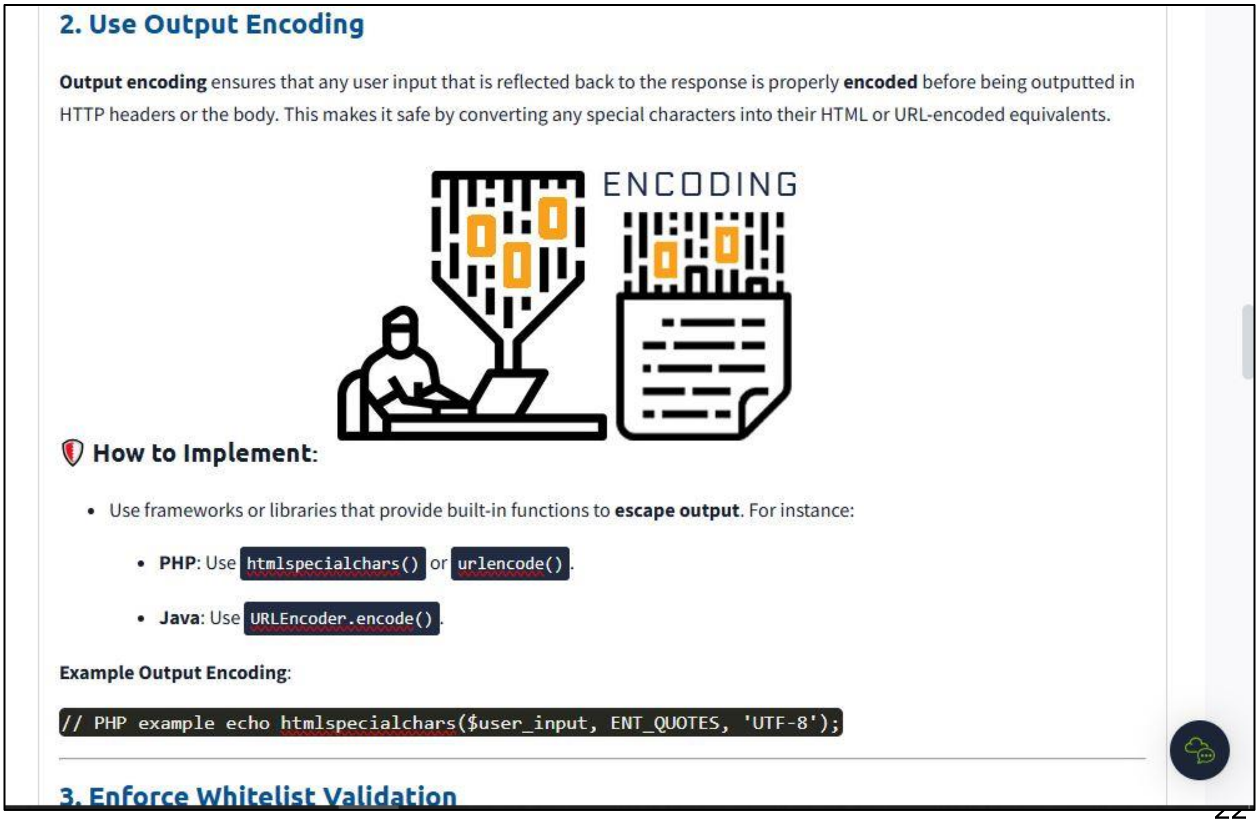
- **Block CRLF characters**: Prevent characters like `\r\n` (Carriage Return and Line Feed) from entering your system. These characters are crucial in splitting the HTTP response.
- **Escape or Encode**: If these characters are required, **encode** them properly (e.g., use **%0D%0A** in URLs).

How to Implement:

- Use built-in libraries to escape special characters.
- Apply input filters like **regular expressions** to allow only safe characters.
- Ensure **input validation** is performed on **all HTTP request headers**.

Example of Proper Sanitization:

```
# Python Example import re sanitized_input = re.sub(r'[\r\n]+', '', user_input)
```



2. Use Output Encoding

Output encoding ensures that any user input that is reflected back to the response is properly **encoded** before being outputted in HTTP headers or the body. This makes it safe by converting any special characters into their HTML or URL-encoded equivalents.

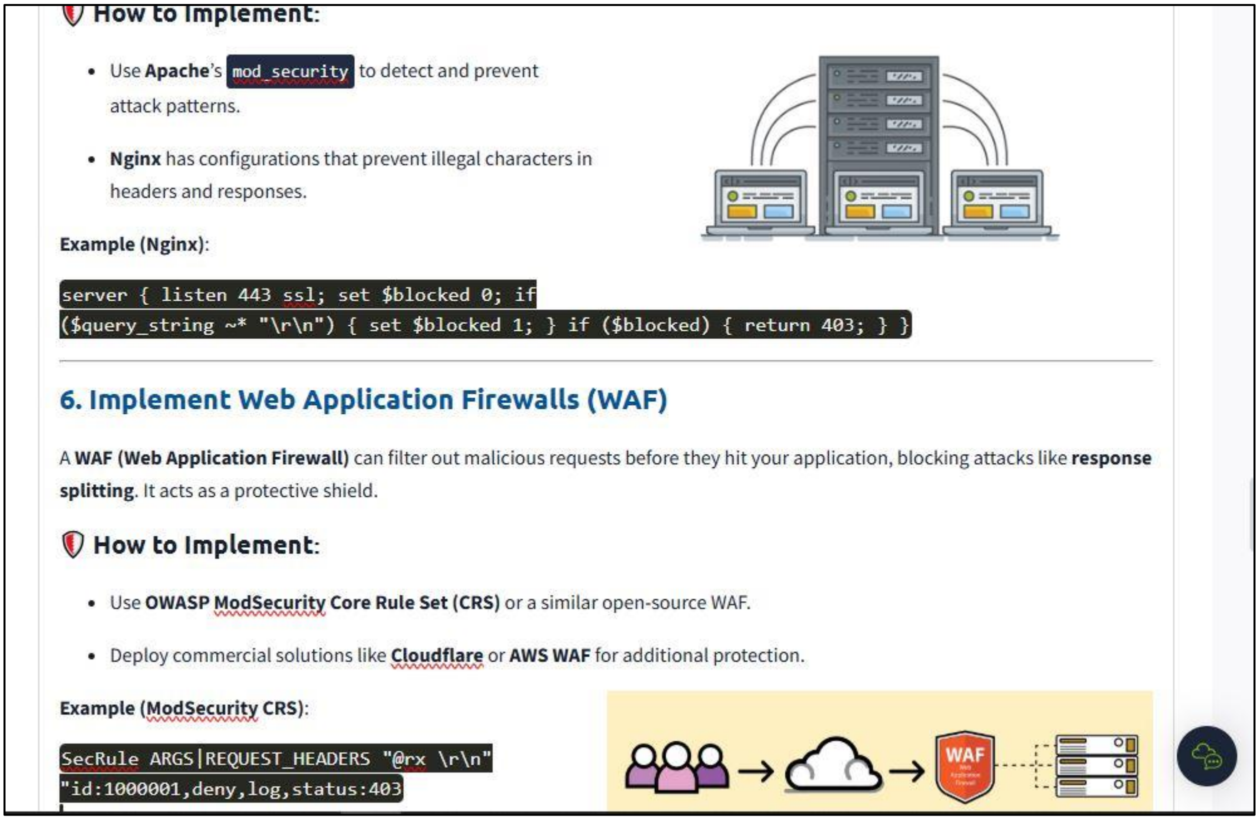
How to Implement:

- Use frameworks or libraries that provide built-in functions to **escape output**. For instance:
  - **PHP**: Use `htmlspecialchars()` or `urlencode()`.
  - **Java**: Use `URLEncoder.encode()`.

Example Output Encoding:

```
// PHP example echo htmlspecialchars($user_input, ENT_QUOTES, 'UTF-8');
```

3. Enforce Whitelist Validation



How to Implement:

- Use **Apache**'s `mod_security` to detect and prevent attack patterns.
- **Nginx** has configurations that prevent illegal characters in headers and responses.

Example (Nginx):

```
server { listen 443 ssl; set $blocked 0; if
($query_string ~* "\r\n") { set $blocked 1; } if ($blocked) { return 403; } }
```

6. Implement Web Application Firewalls (WAF)

A **WAF (Web Application Firewall)** can filter out malicious requests before they hit your application, blocking attacks like **response splitting**. It acts as a protective shield.

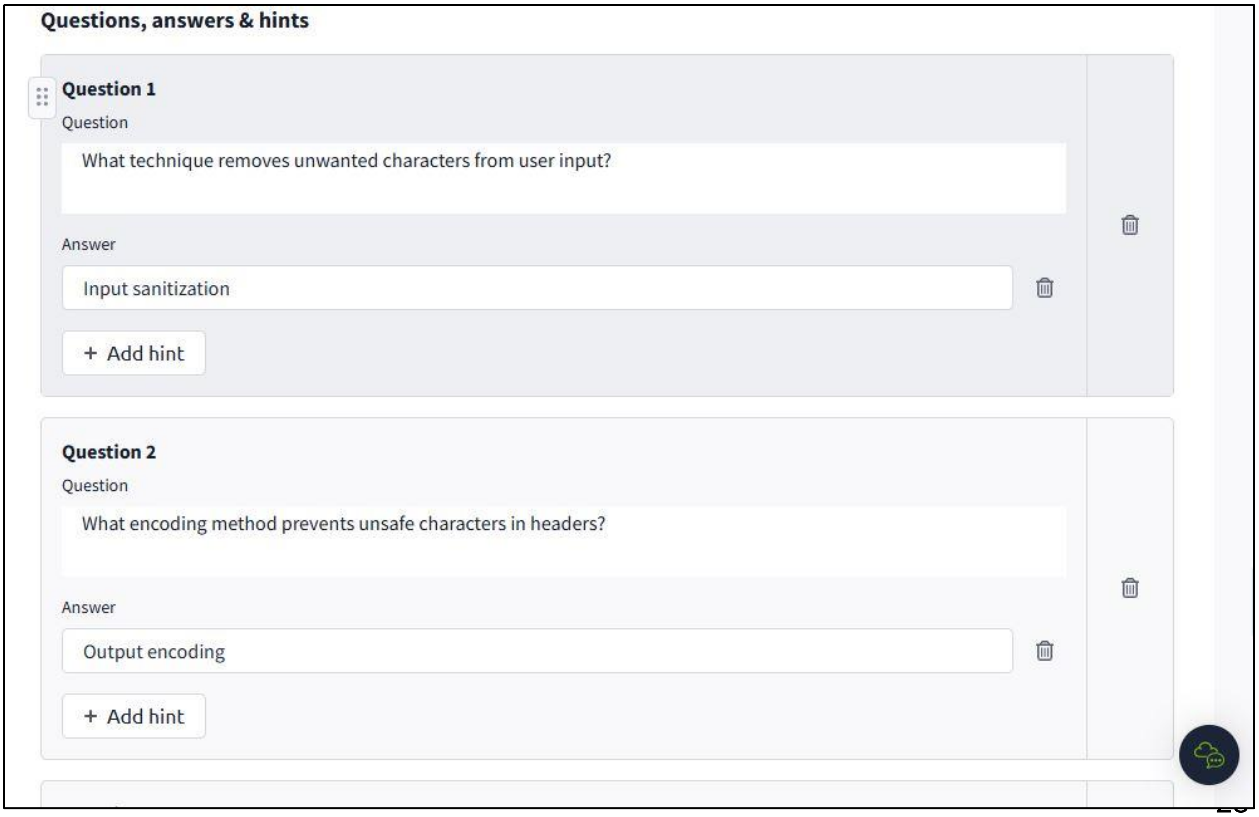
How to Implement:

- Use **OWASP ModSecurity Core Rule Set (CRS)** or a similar open-source WAF.
- Deploy commercial solutions like **Cloudflare** or **AWS WAF** for additional protection.

Example (ModSecurity CRS):

```
SecRule ARGS|REQUEST_HEADERS "@rx \r\n"
"id:1000001,deny,log,status:403
```

At the end this task contains some small questions for the users.



Questions, answers & hints

Question 1

Question

What technique removes unwanted characters from user input?

Answer

Input sanitization

Question 2

Question

What encoding method prevents unsafe characters in headers?

Answer

Output encoding

Task 06

Name – What is HTTP Request Smuggling?

Objective – To understand HTTP Request Smuggling and how it exploits parsing discrepancies between frontend and backend servers

Type – Theory (Easy)

Task description-

This task focuses on Request Smuggling attack where an attacker tries to exploit the difference in behavior between the frontend and backend servers in parsing HTTP requests. By altering the headers such as Content-Length and Transfer-Encoding, hidden requests can be smuggled past security controls and executed on the backend.

Task Breakdown -

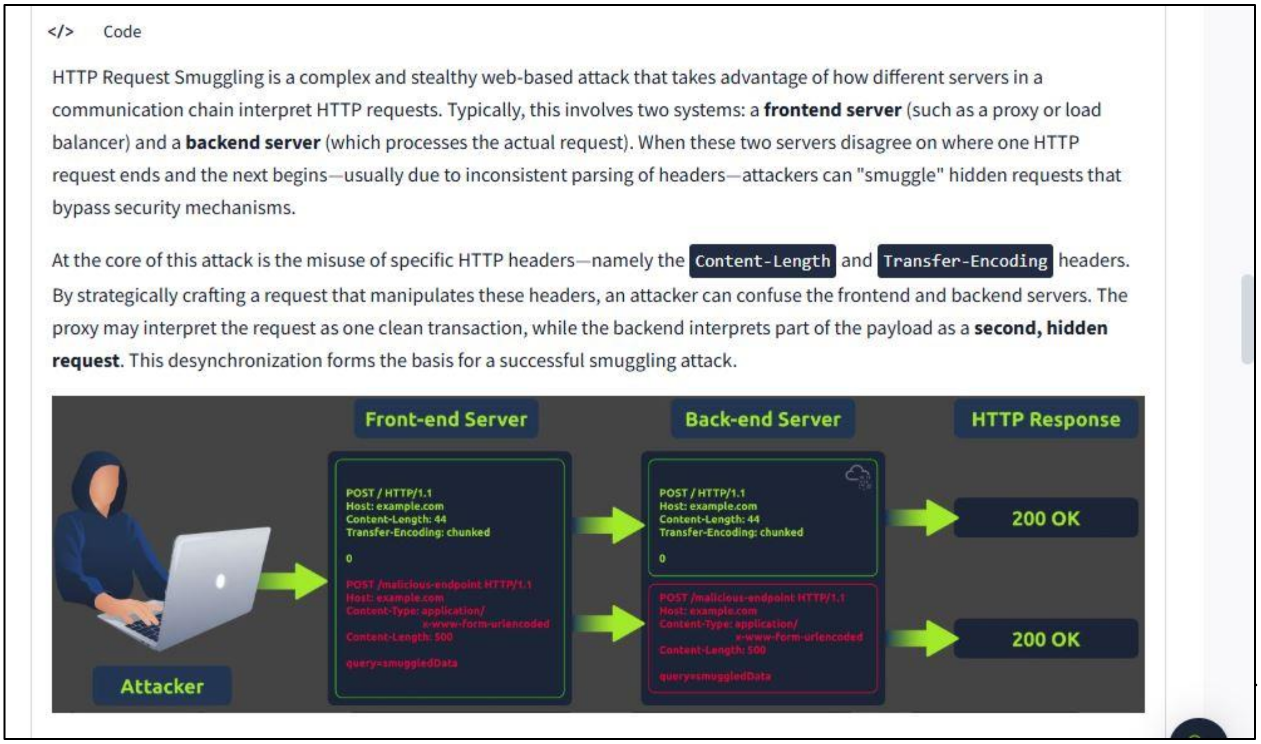
Introduces HTTP Request Smuggling and how it makes use of parsing inconsistencies.

Explains the importance of Content-Length and Transfer-Encoding headers.

Shows how a single request can be variously interpreted by an intermediary such as a proxy and by the backend servers.

Gives an example of an attack in which a hidden GET /admin request gets executed.

Further illustrates the impact of the attack.



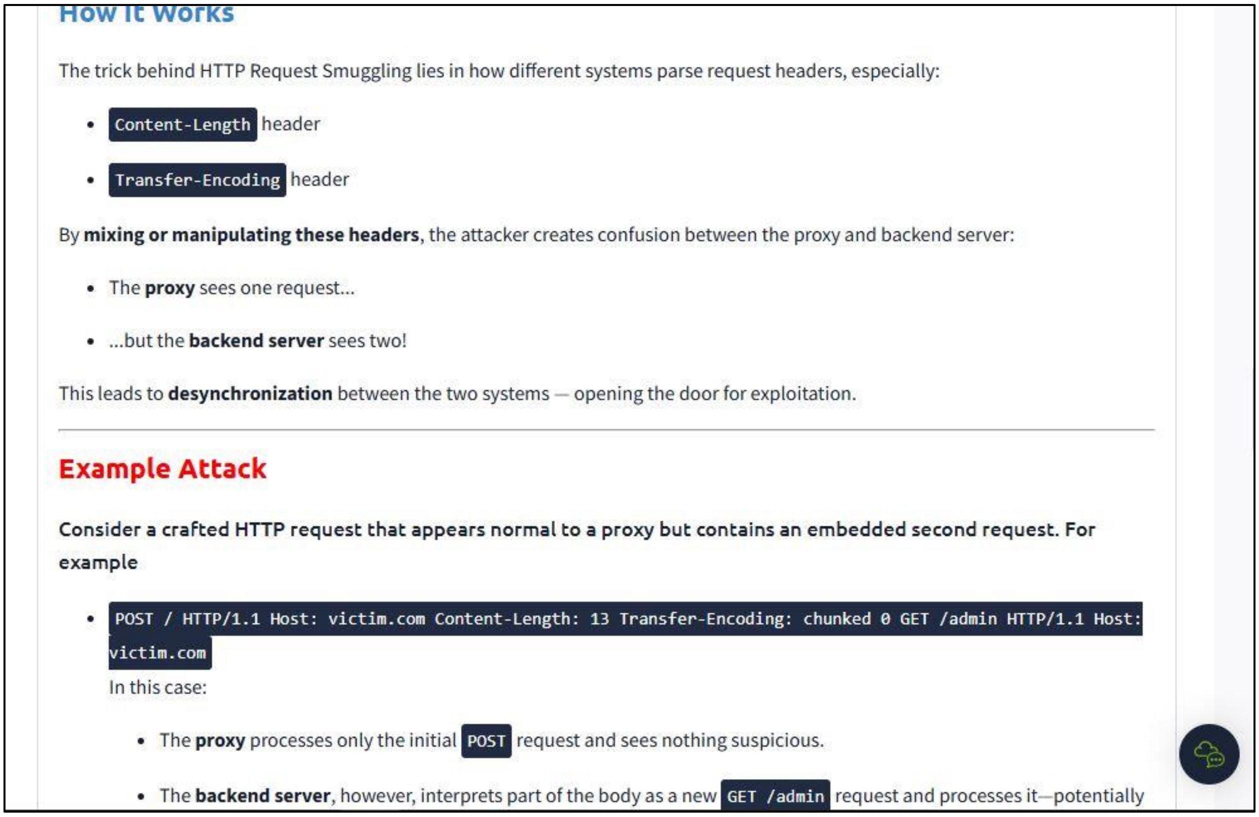
HTTP Request Smuggling is a complex and stealthy web-based attack that takes advantage of how different servers in a communication chain interpret HTTP requests. Typically, this involves two systems: a **frontend server** (such as a proxy or load balancer) and a **backend server** (which processes the actual request). When these two servers disagree on where one HTTP request ends and the next begins—usually due to inconsistent parsing of headers—attackers can "smuggle" hidden requests that bypass security mechanisms.

At the core of this attack is the misuse of specific HTTP headers—namely the `Content-Length` and `Transfer-Encoding` headers. By strategically crafting a request that manipulates these headers, an attacker can confuse the frontend and backend servers. The proxy may interpret the request as one clean transaction, while the backend interprets part of the payload as a **second, hidden request**. This desynchronization forms the basis for a successful smuggling attack.

```
POST / HTTP/1.1
Host: example.com
Content-Length: 44
Transfer-Encoding: chunked

0

POST /malicious-endpoint HTTP/1.1
Host: example.com
Content-Type: application/
              x-www-form-urlencoded
Content-Length: 500

query=smuggledData
```

```
POST / HTTP/1.1
Host: example.com
Content-Length: 44
Transfer-Encoding: chunked

0
```

```
POST /malicious-endpoint HTTP/1.1
Host: example.com
Content-Type: application/
              x-www-form-urlencoded
Content-Length: 500

query=smuggledData
```



How It Works

The trick behind HTTP Request Smuggling lies in how different systems parse request headers, especially:

- `Content-Length` header
- `Transfer-Encoding` header

By **mixing or manipulating these headers**, the attacker creates confusion between the proxy and backend server:

- The **proxy** sees one request...
- ...but the **backend server** sees two!

This leads to **desynchronization** between the two systems — opening the door for exploitation.

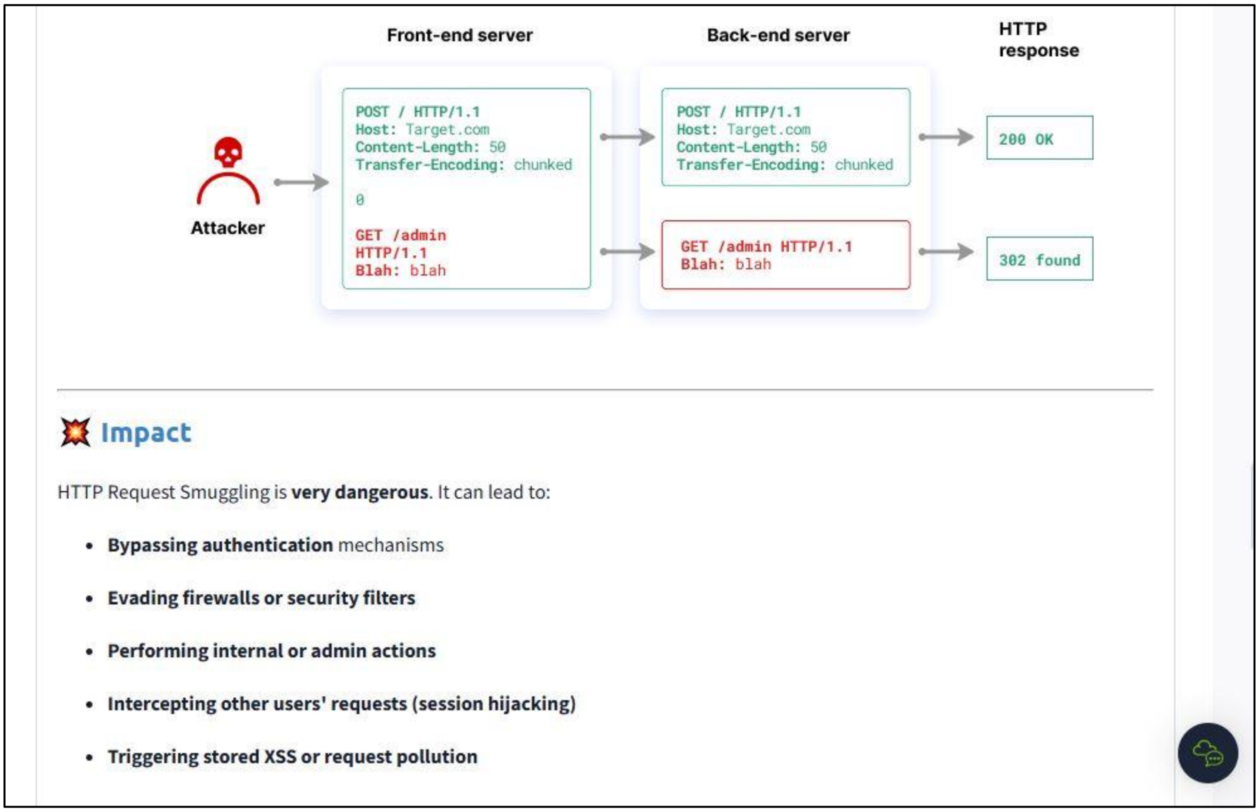
Example Attack

Consider a crafted HTTP request that appears normal to a proxy but contains an embedded second request. For example

- `POST / HTTP/1.1 Host: victim.com Content-Length: 13 Transfer-Encoding: chunked 0 GET /admin HTTP/1.1 Host: victim.com`

  In this case:

  - The **proxy** processes only the initial `POST` request and sees nothing suspicious.
  - The **backend server**, however, interprets part of the body as a new `GET /admin` request and processes it—potentially



```
POST / HTTP/1.1
Host: Target.com
Content-Length: 50
Transfer-Encoding: chunked

0

GET /admin
HTTP/1.1
Blah: blah
```

```
POST / HTTP/1.1
Host: Target.com
Content-Length: 50
Transfer-Encoding: chunked
```

```
GET /admin HTTP/1.1
Blah: blah
```

```
200 OK
```

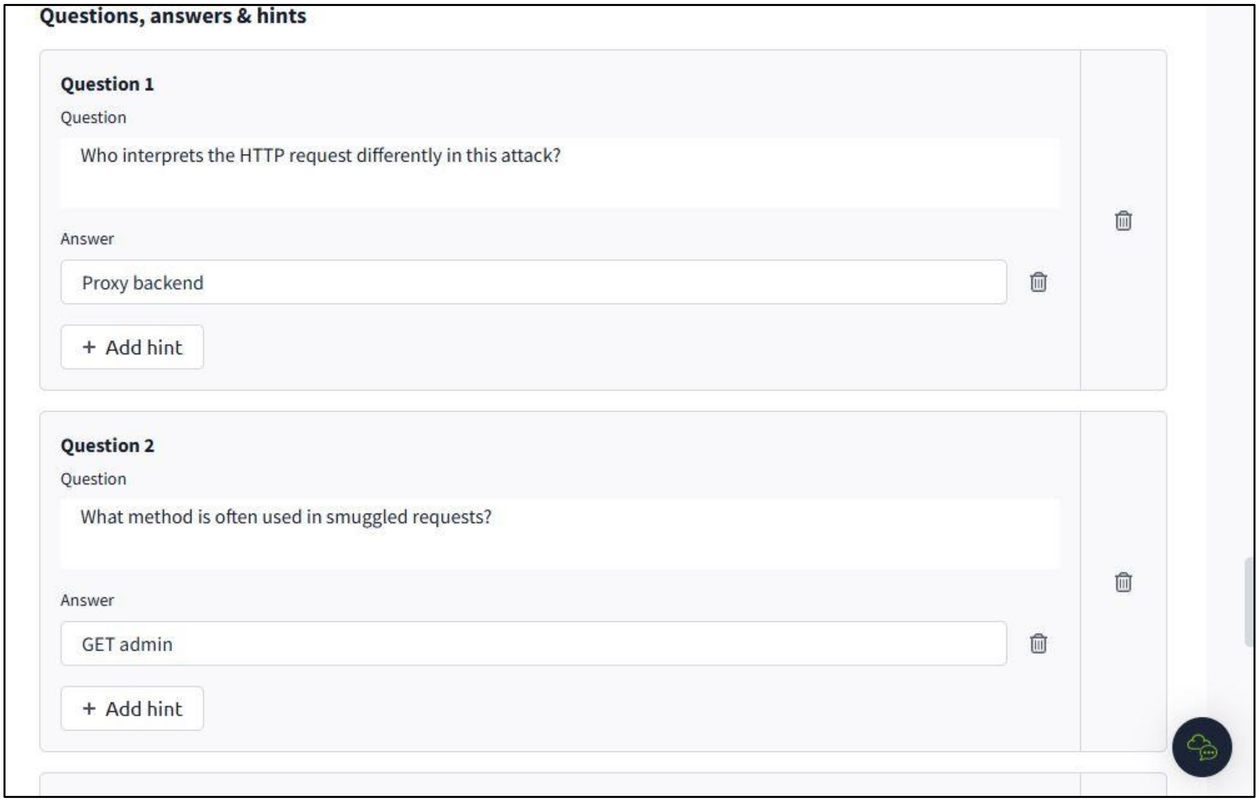
```
302 found
```

💥 Impact

HTTP Request Smuggling is **very dangerous**. It can lead to:

- **Bypassing authentication** mechanisms
- **Evading firewalls or security filters**
- **Performing internal or admin actions**
- **Intercepting other users' requests (session hijacking)**
- **Triggering stored XSS or request pollution**

At the end this task contains some small questions for the users.



Questions, answers & hints

Question 1

Question

Who interprets the HTTP request differently in this attack?

Answer

Proxy backend

Question 2

Question

What method is often used in smuggled requests?

Answer

GET admin

Task 07

Name – Let’s detect HTTP Request Smuggling

Objective - Simulate how a web server might misinterpret HTTP requests with conflicting headers, leading to a request smuggling scenario.

Type – practical (Medium)

Task description-

In this task, we simulate an HTTP Request Smuggling attack using a simple web server.

By manipulating HTTP request data, attackers can sneak hidden requests that the server might process without the application's knowledge. This leads to serious security breaches.

Task Breakdown -

Exploit Scenario Description

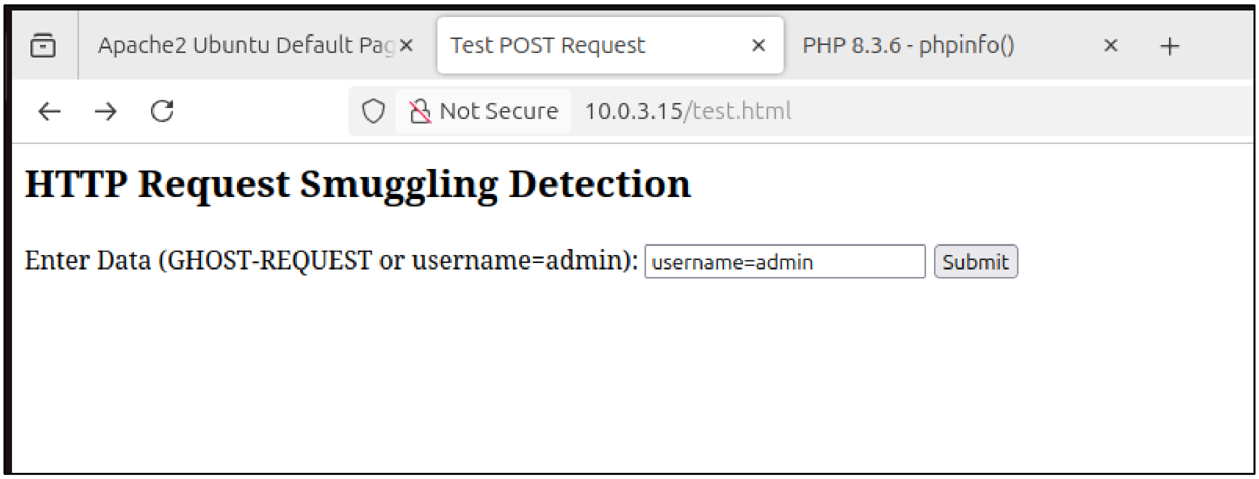
In this simulation, we set up a PHP script that reads POST request bodies and looks for suspicious patterns.

- First, a normal POST request is submitted with data like username=admin and no issues are detected.
- Next, a smuggled payload (GHOST-REQUEST) is injected into the POST body.
- The PHP script detects the hidden malicious content and raises a red alert, flagging the suspicious behavior.

This demonstrates how backend servers could mistakenly process hidden requests if input is not properly sanitized.

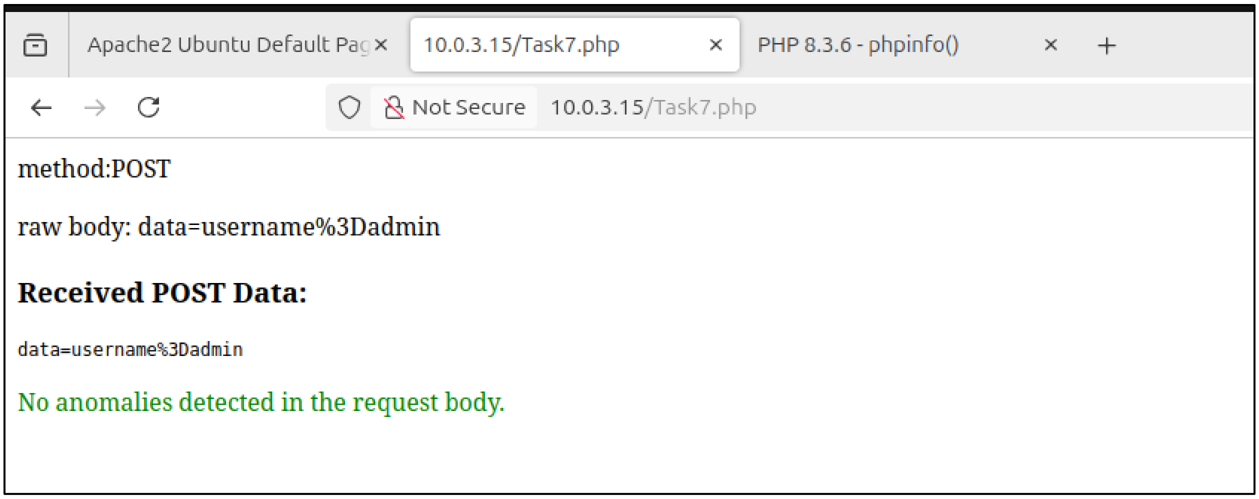
Expected outcome -

1. Submit a POST request with **username=admin** via HTML form – The server responses to this normally, displaying a green output: *No anomalies detected.*



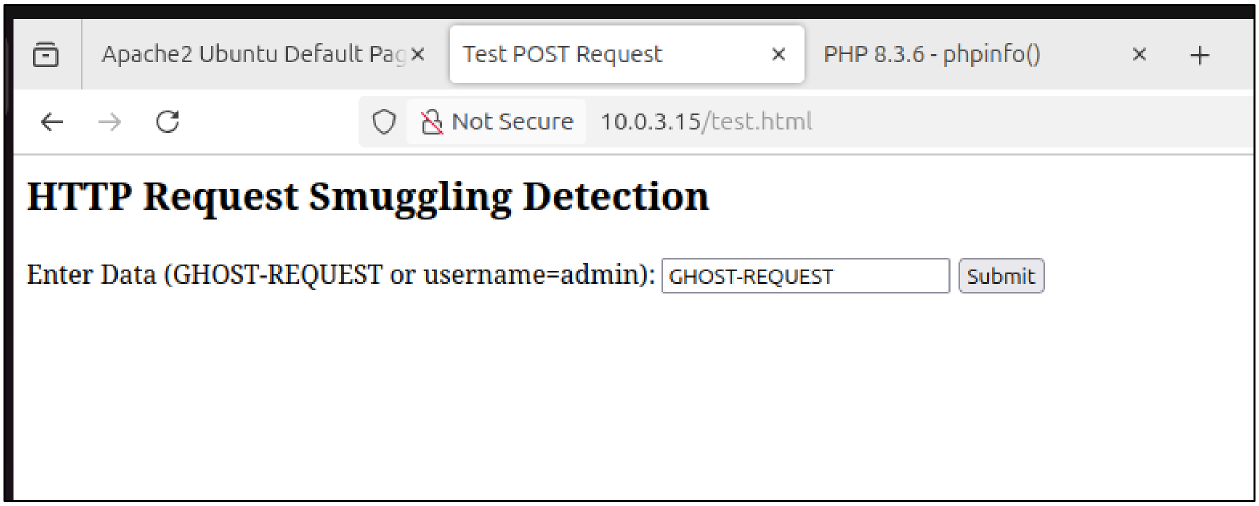
HTTP Request Smuggling Detection

Enter Data (GHOST-REQUEST or username=admin): username=admin



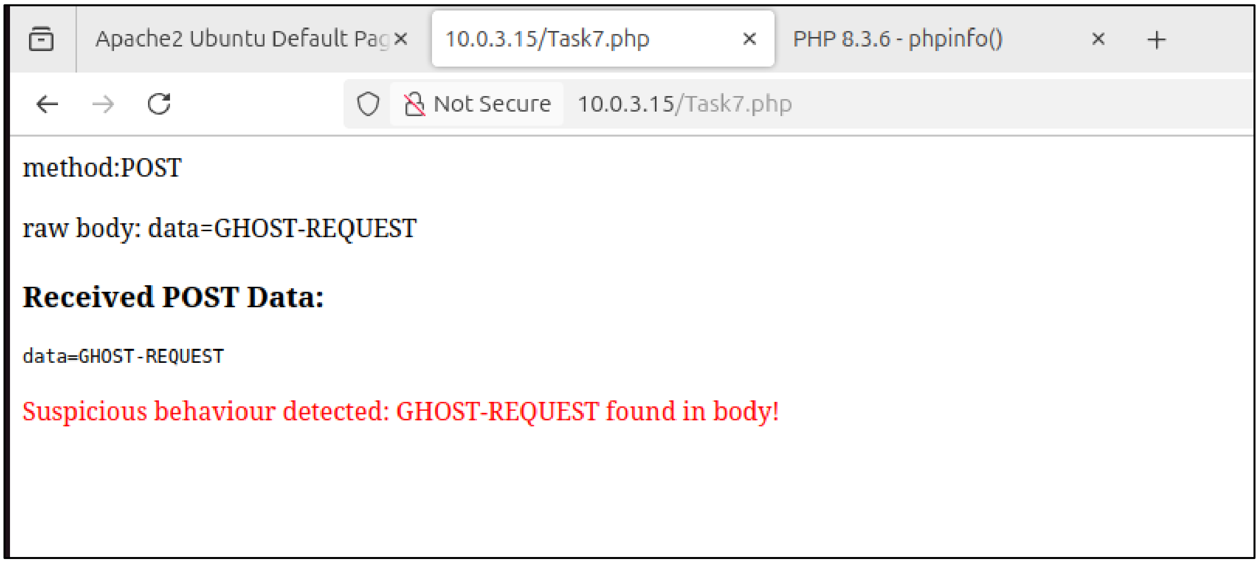
```
method:POST

raw body: data=username%3Dadmin

Received POST Data:

data=username%3Dadmin

No anomalies detected in the request body.
```

2. Submit a POST request with **GHOST-REQUEST** via the form – Server detects the suspicious behavior and outputs a red warning: *GHOST-REQUEST found!.*



HTTP Request Smuggling Detection

Enter Data (GHOST-REQUEST or username=admin): GHOST-REQUEST



method:POST

raw body: data=GHOST-REQUEST

Received POST Data:

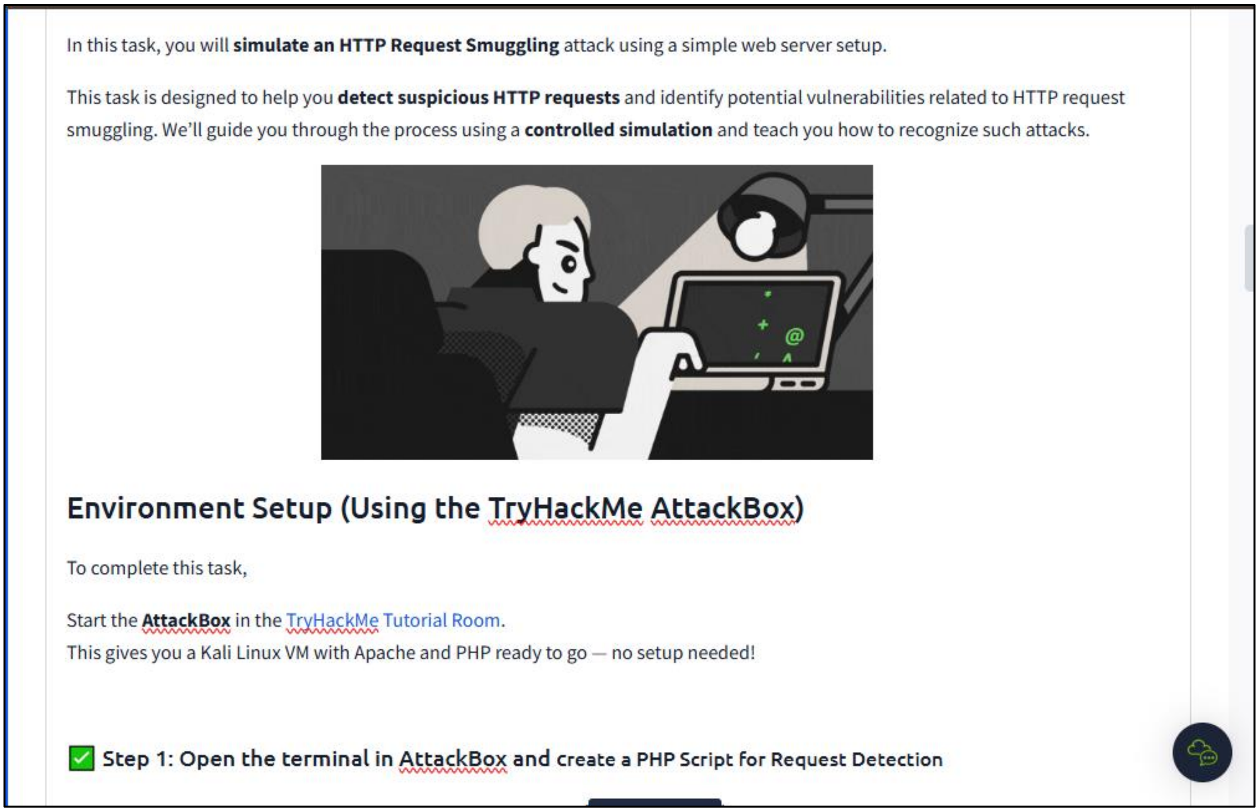
```
data=GHOST-REQUEST
```

Suspicious behaviour detected: GHOST-REQUEST found in body!

This task shows why HTTP Request Smuggling is a critical web application security issue.

If servers interpret incoming requests differently than applications expect, attackers could

- Bypass authentication
- Access restricted resources
- Poison web caches
- Smuggle unauthorized requests past security controls



In this task, you will **simulate an HTTP Request Smuggling** attack using a simple web server setup.

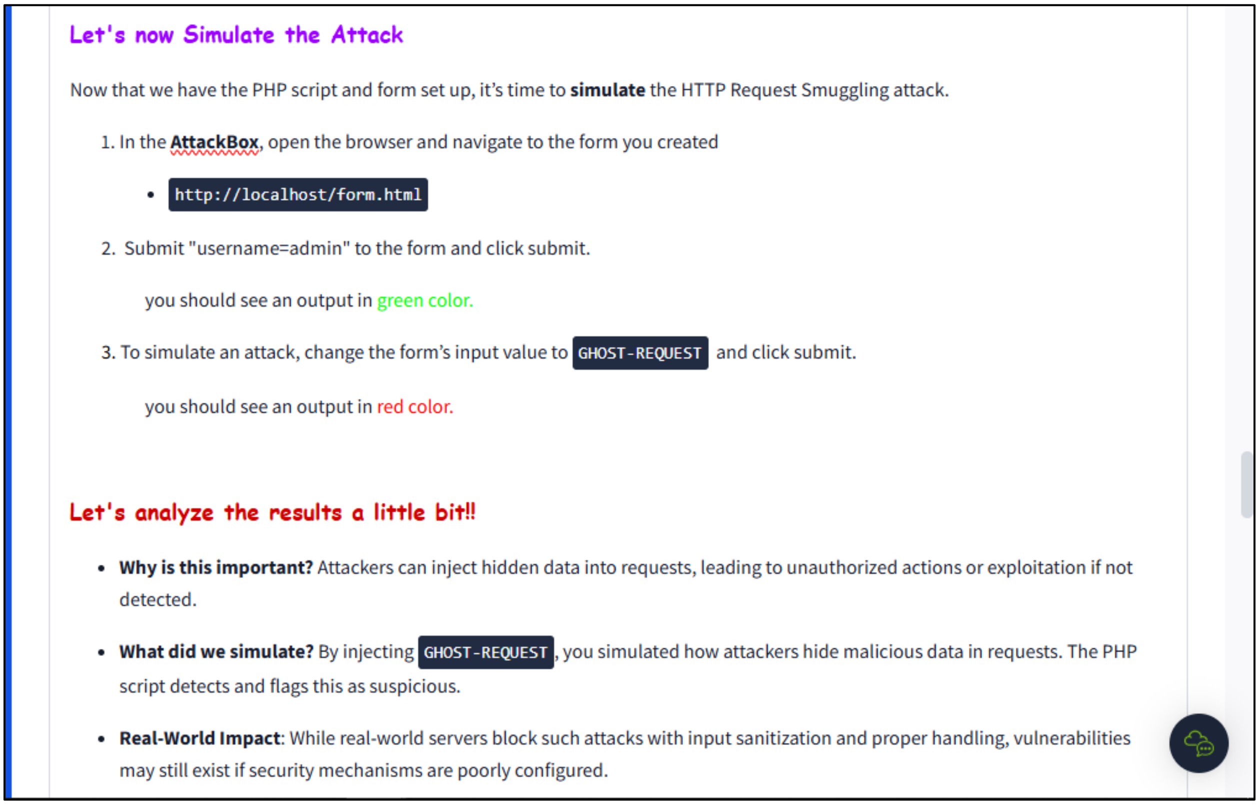
This task is designed to help you **detect suspicious HTTP requests** and identify potential vulnerabilities related to HTTP request smuggling. We'll guide you through the process using a **controlled simulation** and teach you how to recognize such attacks.

Environment Setup (Using the TryHackMe AttackBox)

To complete this task,

Start the **AttackBox** in the TryHackMe Tutorial Room.
This gives you a Kali Linux VM with Apache and PHP ready to go — no setup needed!

✅ Step 1: Open the terminal in AttackBox and create a PHP Script for Request Detection



Let's now Simulate the Attack

Now that we have the PHP script and form set up, it's time to **simulate** the HTTP Request Smuggling attack.

1. In the **AttackBox**, open the browser and navigate to the form you created
   - `http://localhost/form.html`
2. Submit "username=admin" to the form and click submit.

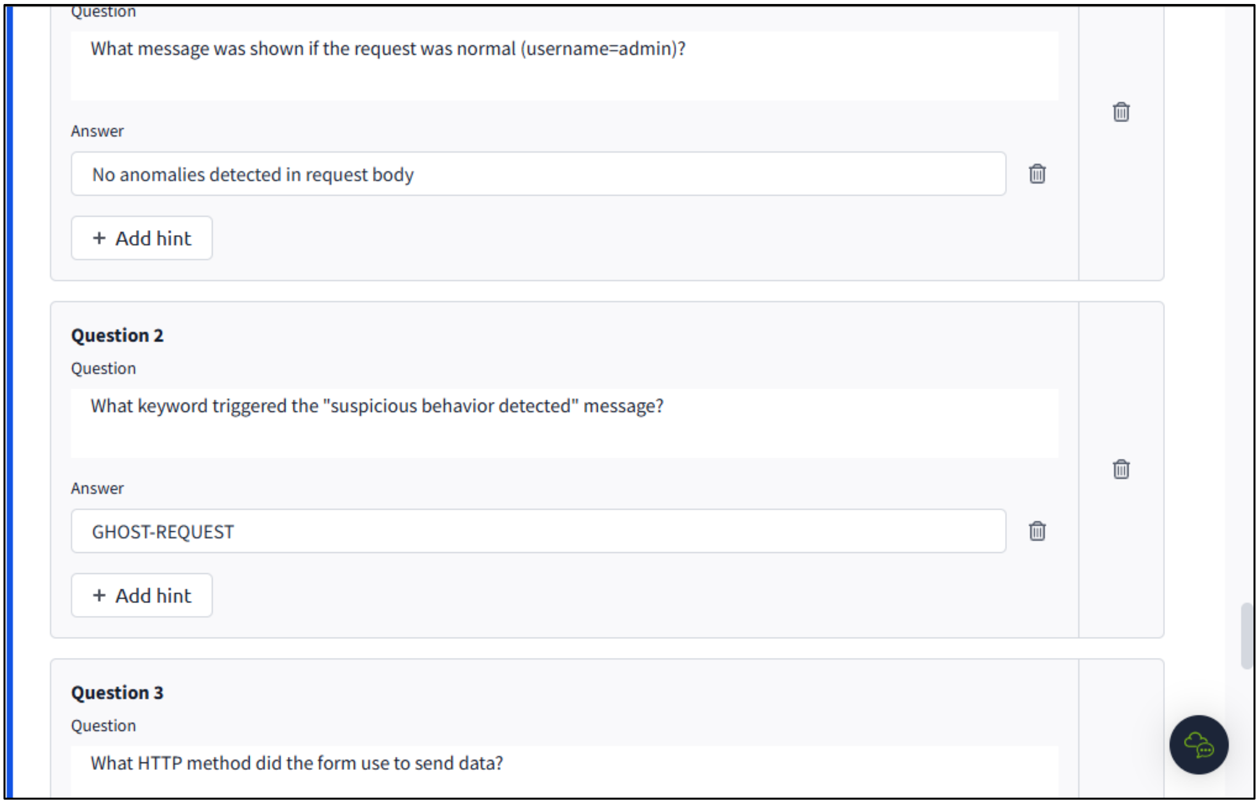
   you should see an output in green color.
3. To simulate an attack, change the form's input value to `GHOST-REQUEST` and click submit.

   you should see an output in red color.

Let's analyze the results a little bit!!

- **Why is this important?** Attackers can inject hidden data into requests, leading to unauthorized actions or exploitation if not detected.
- **What did we simulate?** By injecting `GHOST-REQUEST`, you simulated how attackers hide malicious data in requests. The PHP script detects and flags this as suspicious.
- **Real-World Impact**: While real-world servers block such attacks with input sanitization and proper handling, vulnerabilities may still exist if security mechanisms are poorly configured.

At the end this task contains some small questions for the users.



Question

What message was shown if the request was normal (username=admin)?

Answer

No anomalies detected in request body

Question 2

Question

What keyword triggered the "suspicious behavior detected" message?

Answer

GHOST-REQUEST

Question 3

Question

What HTTP method did the form use to send data?

Task 08

Name – Let’s exploit HTTP Request Smuggling

Objective - Users will learn how an attacker could inject a smuggled request to bypass intended behavior.

Type – practical (Medium)

Task description-

In this simulation task, we explore the basic idea about HTTP Request Smuggling, a technique where attackers exploit mismatches between front-end and back-end servers parse HTTP requests.

Instead of doing a real attack (which can be complex and dangerous), I safely simulate the attack by detecting hidden HTTP requests inserted into POST bodies.

Task Breakdown -

Exploit Scenario Description

In real-world HTTP Request Smuggling, attackers can

- Bypass authentication
- Access hidden admin panels
- Poison caches
- Hijack user sessions

They achieve this by crafting requests that confuse server parsing logic, for example, by manipulating Content-Length and Transfer-Encoding: chunked headers.

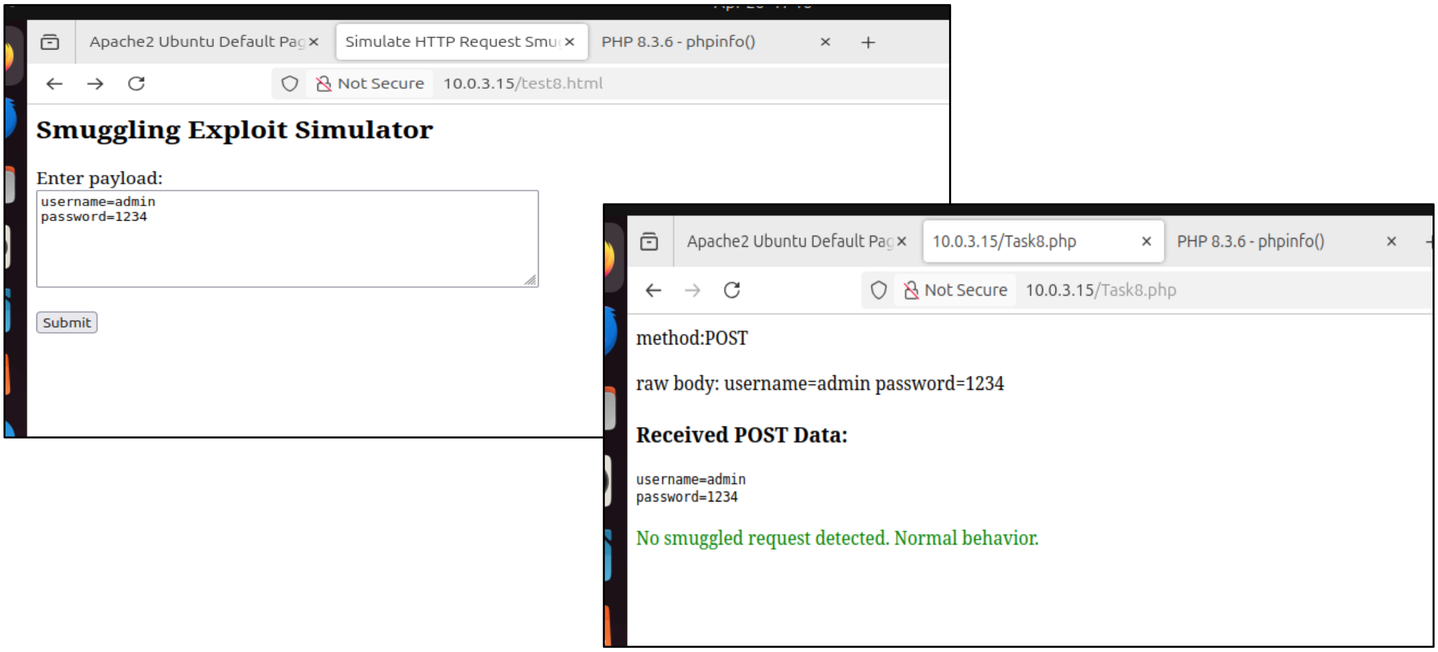
So in this simulation

- A form submits data to a PHP script.
- If the submitted data contains a hidden **GET /admin HTTP/1.1** request, the server simulates an exploit by granting access to a fake admin panel.
- Otherwise, normal behavior is shown.

This allows us to safely learn the smuggling concept without needing multiple server systems.

Expected outcome -

1. Submit normal input like **username=admin&password**=1234 from the form – The server responds a green message: *No smuggled request detected. Normal behavior*.



Smuggling Exploit Simulator

Enter payload:

```
username=admin
password=1234
```

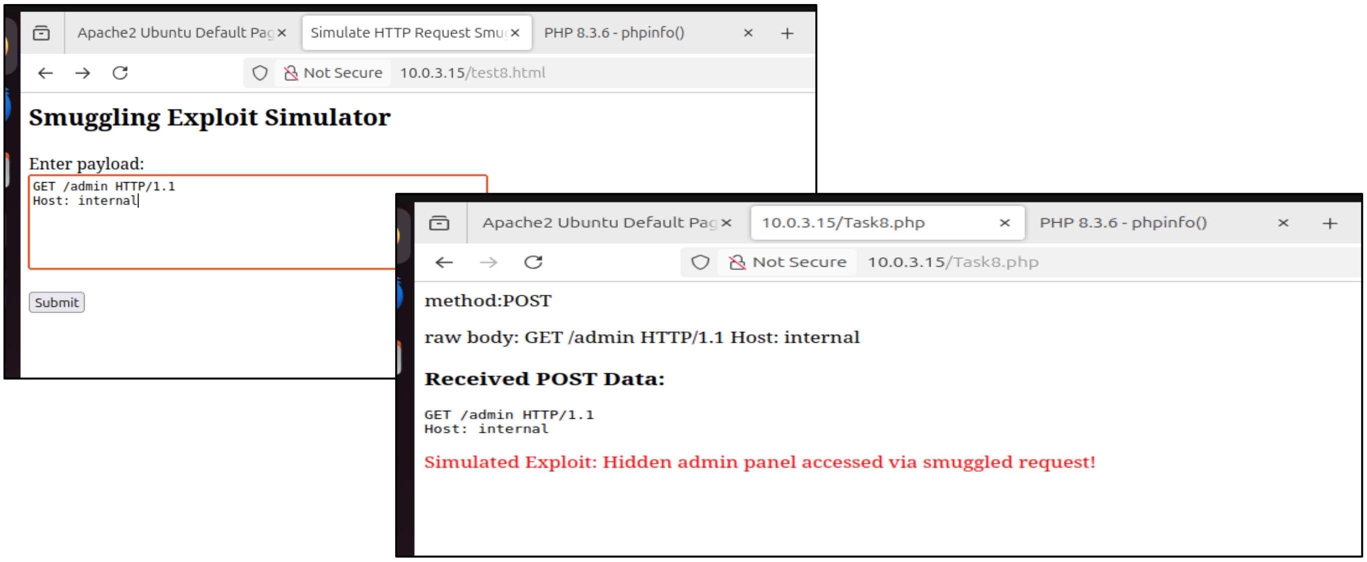
```
method:POST

raw body: username=admin password=1234
```

Received POST Data:

```
username=admin
password=1234
```

No smuggled request detected. Normal behavior.

2. Submit a smuggling payload like
   GET /admin HTTP/1.1
   Host: internal - The server detects the hidden request and responds with a red warning:
   Simulated Exploit: Hidden admin panel accessed via smuggled request!



Apache2 Ubuntu Default Pag ×
Simulate HTTP Request Smu ×
PHP 8.3.6 - phpinfo() ×
Not Secure 10.0.3.15/test8.html
Submit

Smuggling Exploit Simulator

Enter payload:

```
GET /admin HTTP/1.1
Host: internal
```

Apache2 Ubuntu Default Pag ×
10.0.3.15/Task8.php ×
PHP 8.3.6 - phpinfo() ×
Not Secure 10.0.3.15/Task8.php

method:POST

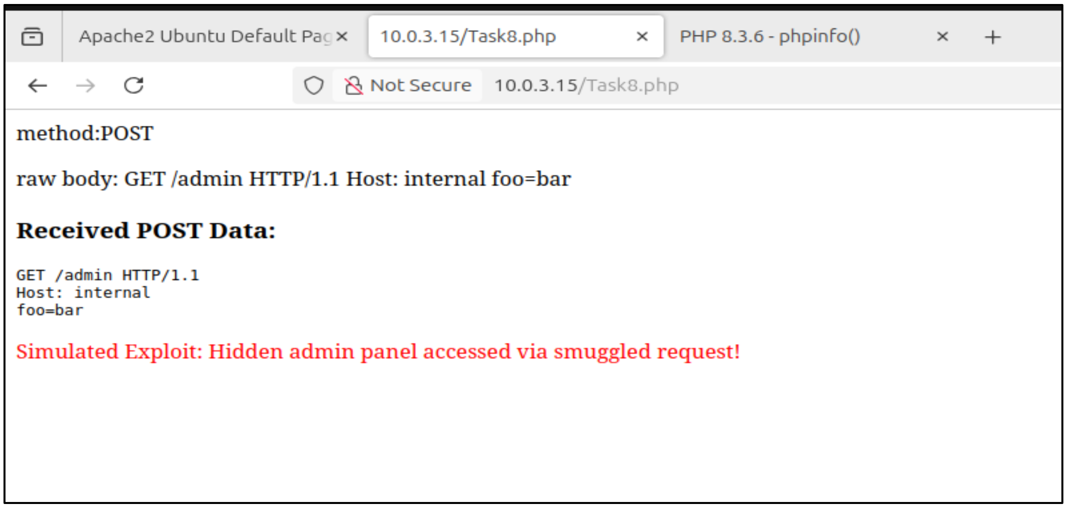
raw body: GET /admin HTTP/1.1 Host: internal

Received POST Data:

```
GET /admin HTTP/1.1
Host: internal
```

Simulated Exploit: Hidden admin panel accessed via smuggled request!

3. Submit mixed/random payloads (Extra challenge)



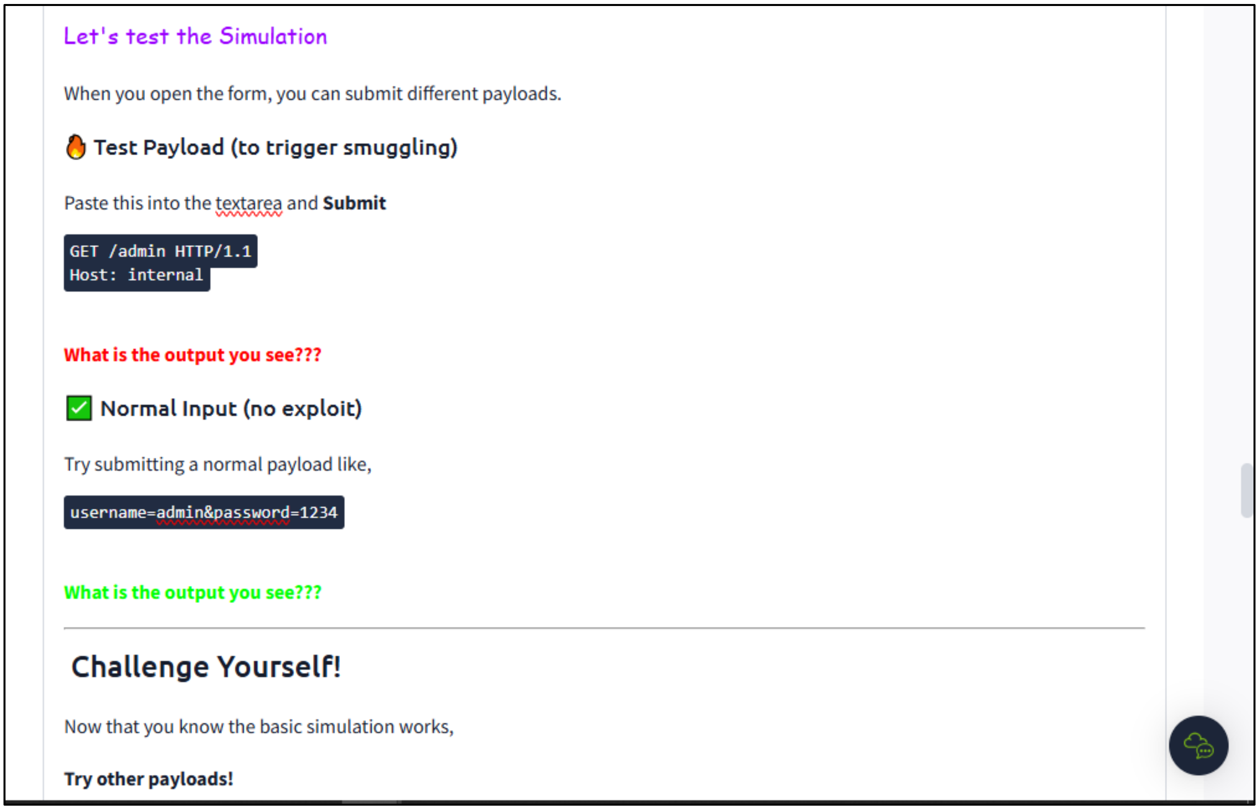
method:POST

raw body: GET /admin HTTP/1.1 Host: internal foo=bar

Received POST Data:

```
GET /admin HTTP/1.1
Host: internal
foo=bar
```

Simulated Exploit: Hidden admin panel accessed via smuggled request!



Let's test the Simulation

When you open the form, you can submit different payloads.

🔥 Test Payload (to trigger smuggling)

Paste this into the textarea and **Submit**

```
GET /admin HTTP/1.1
Host: internal
```

What is the output you see???

✅ Normal Input (no exploit)

Try submitting a normal payload like,

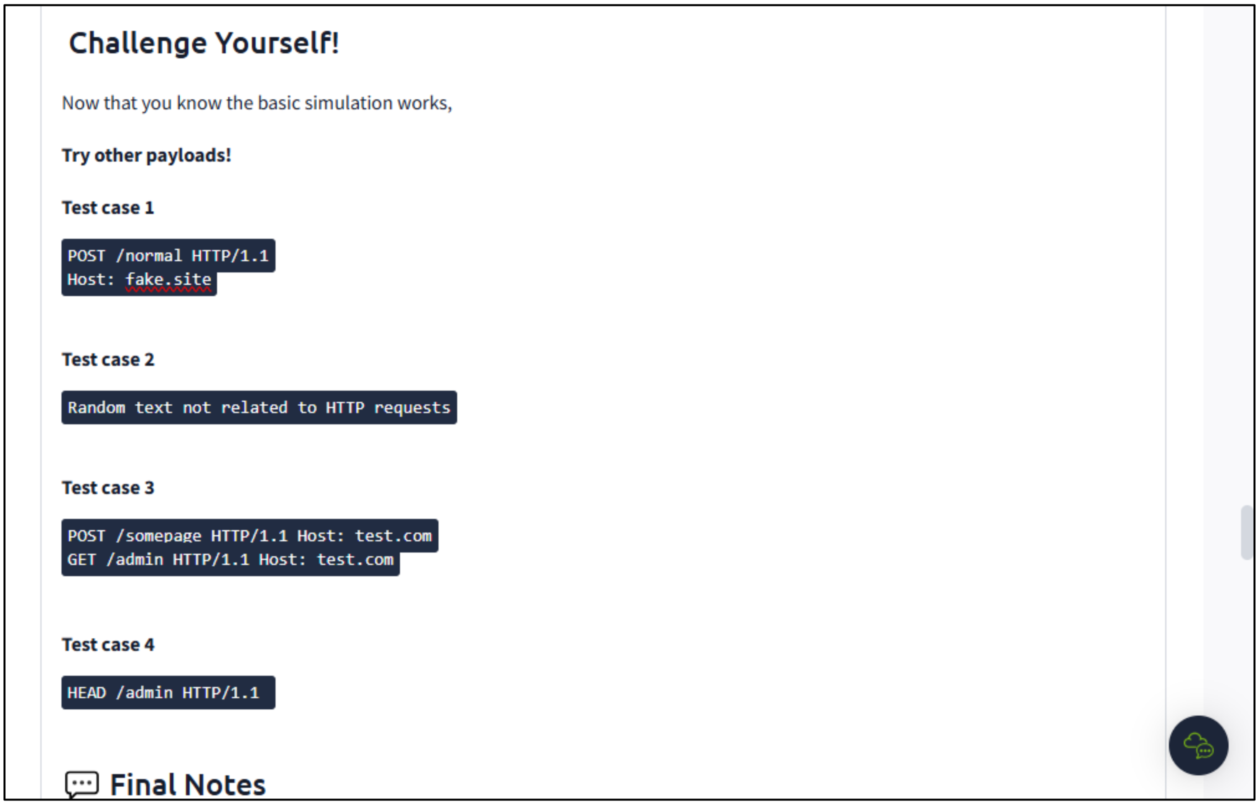
```
username=admin&password=1234
```

What is the output you see???

Challenge Yourself!

Now that you know the basic simulation works,

Try other payloads!



Challenge Yourself!

Now that you know the basic simulation works,

Try other payloads!

Test case 1

```
POST /normal HTTP/1.1
Host: fake.site
```

Test case 2

```
Random text not related to HTTP requests
```

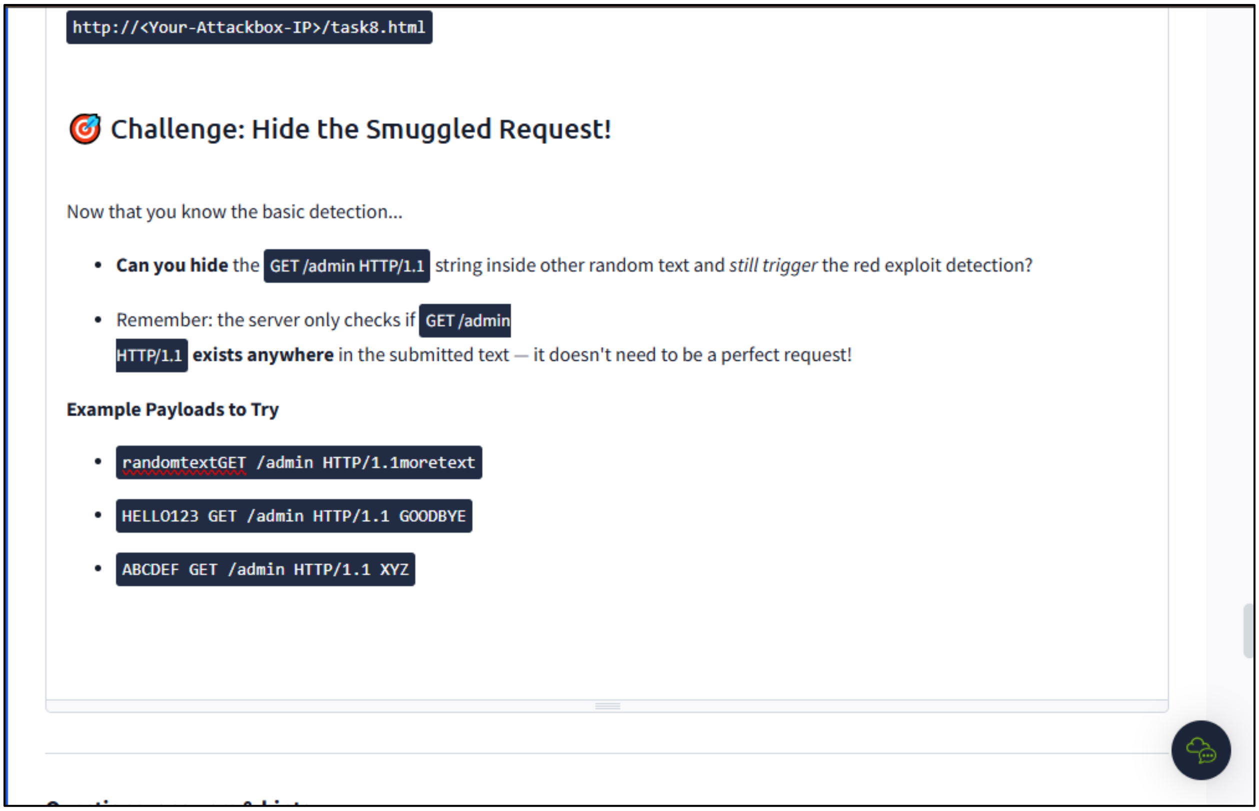
Test case 3

```
POST /somepage HTTP/1.1 Host: test.com
GET /admin HTTP/1.1 Host: test.com
```

Test case 4

```
HEAD /admin HTTP/1.1
```

💬 Final Notes



`http://<Your-Attackbox-IP>/task8.html`

🎯 Challenge: Hide the Smuggled Request!

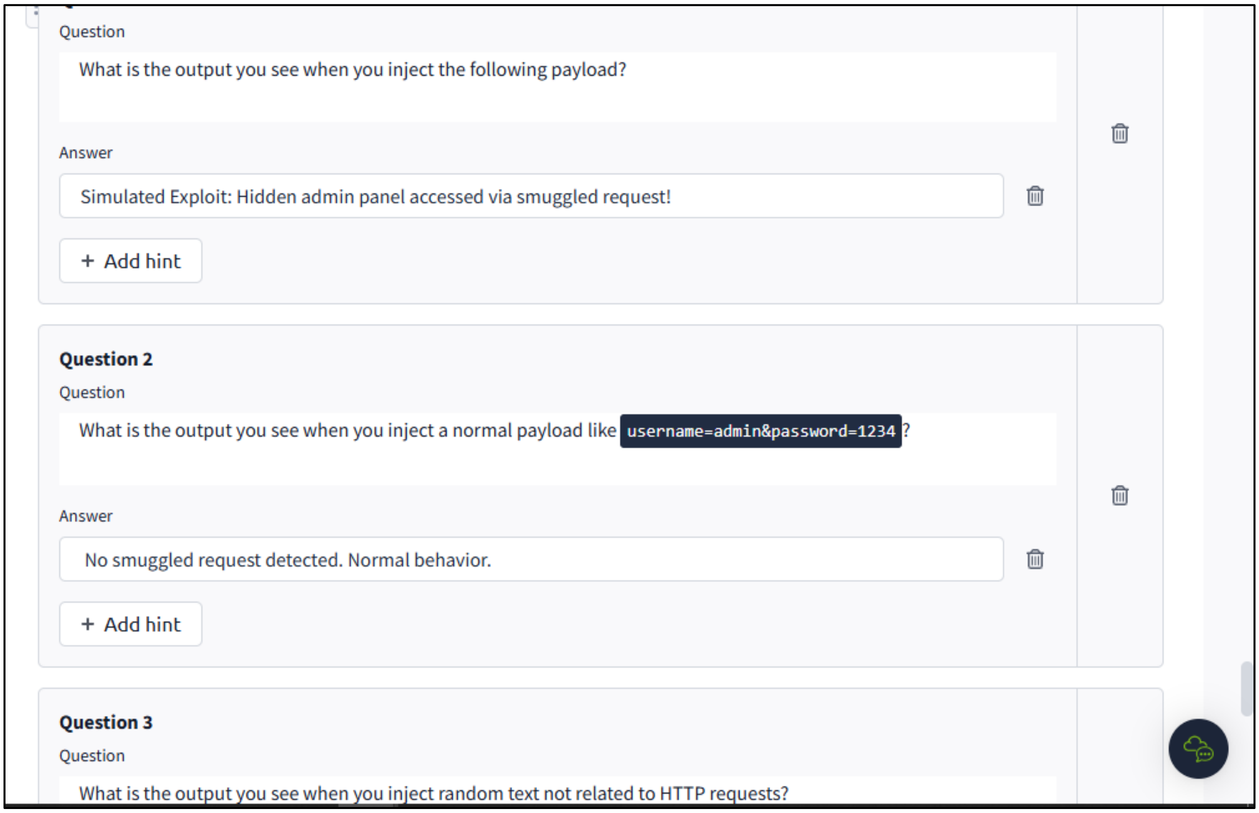
Now that you know the basic detection...

- **Can you hide** the `GET /admin HTTP/1.1` string inside other random text and *still trigger* the red exploit detection?
- Remember: the server only checks if `GET /admin HTTP/1.1` **exists anywhere** in the submitted text — it doesn't need to be a perfect request!

Example Payloads to Try

- `randomtextGET /admin HTTP/1.1moretext`
- `HELLO123 GET /admin HTTP/1.1 GOODBYE`
- `ABCDEF GET /admin HTTP/1.1 XYZ`

At the end this task contains some small questions for the users.



Question

What is the output you see when you inject the following payload?

Answer

Simulated Exploit: Hidden admin panel accessed via smuggled request!

Question 2

Question

What is the output you see when you inject a normal payload like `username=admin&password=1234`?

Answer

No smuggled request detected. Normal behavior.

Question 3

Question

What is the output you see when you inject random text not related to HTTP requests?

Task 09

Name – Mitigating HTTP Request Smuggling

Objective – Users will be able to master the techniques to protect web applications from HTTP Request Smuggling attacks

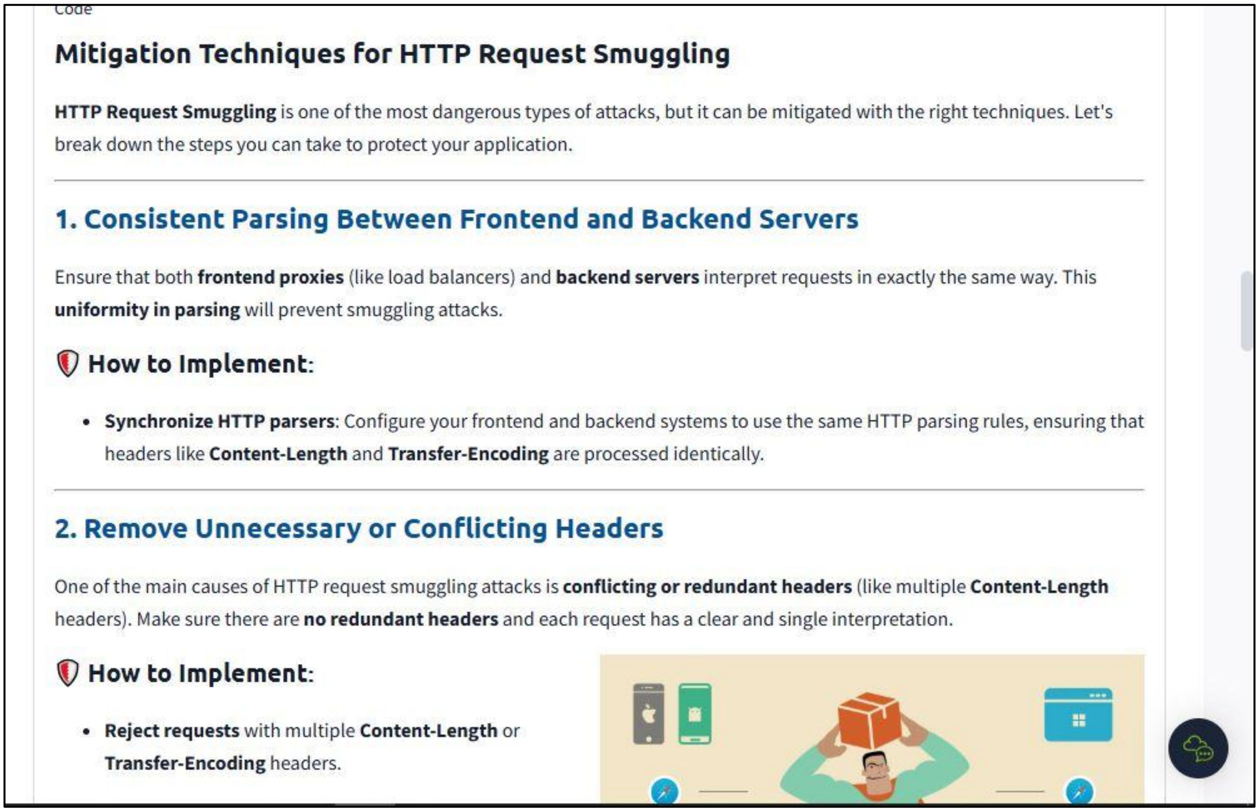
Type – Theory (Easy)

Task description-

This 9th task focuses on implementing effective mitigation techniques to protect against HTTP Request Smuggling attacks by ensuring consistent parsing, header validation, and using security tools.

Task Breakdown -

The task focuses on various kinds of mitigation methods and their explanations.



Code

Mitigation Techniques for HTTP Request Smuggling

HTTP Request Smuggling is one of the most dangerous types of attacks, but it can be mitigated with the right techniques. Let's break down the steps you can take to protect your application.

1. Consistent Parsing Between Frontend and Backend Servers

Ensure that both **frontend proxies** (like load balancers) and **backend servers** interpret requests in exactly the same way. This **uniformity in parsing** will prevent smuggling attacks.

How to Implement:

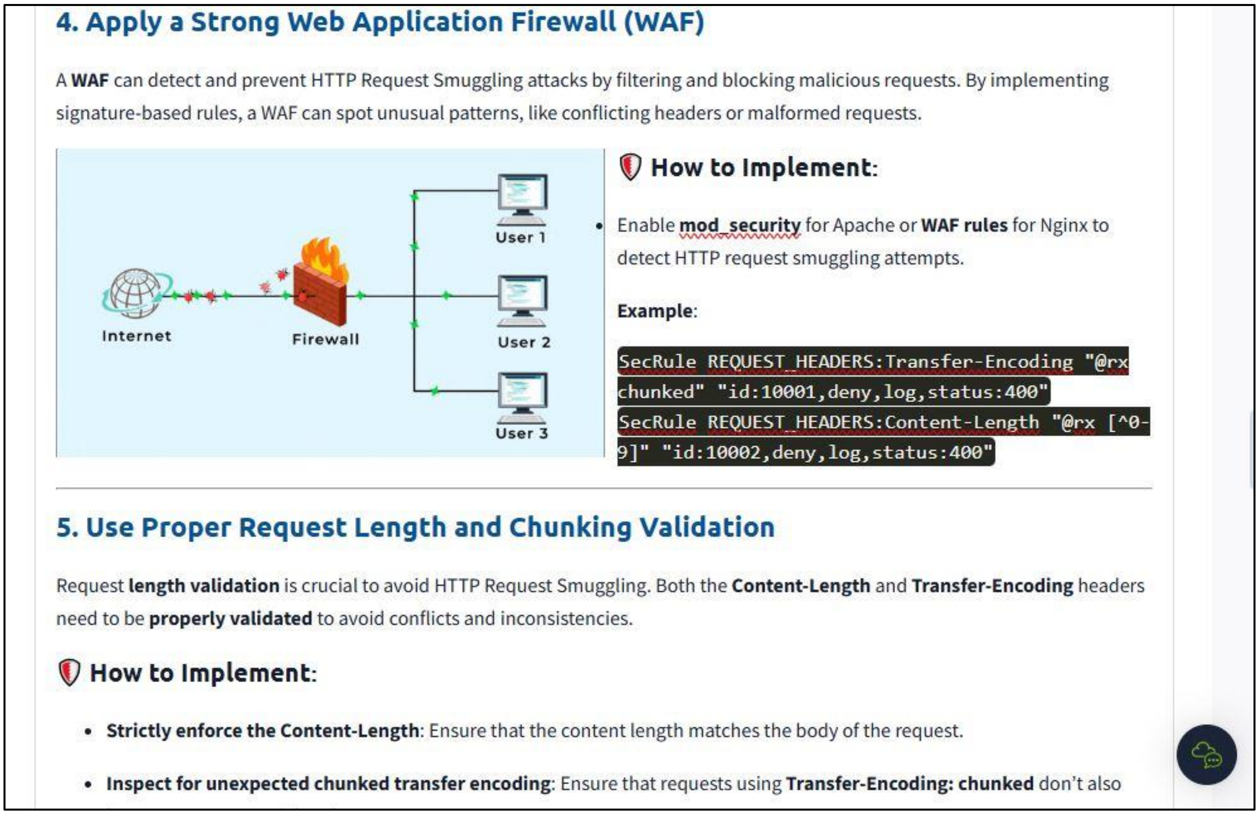
- **Synchronize HTTP parsers**: Configure your frontend and backend systems to use the same HTTP parsing rules, ensuring that headers like **Content-Length** and **Transfer-Encoding** are processed identically.

2. Remove Unnecessary or Conflicting Headers

One of the main causes of HTTP request smuggling attacks is **conflicting or redundant headers** (like multiple **Content-Length** headers). Make sure there are **no redundant headers** and each request has a clear and single interpretation.

How to Implement:

- **Reject requests** with multiple **Content-Length** or **Transfer-Encoding** headers.



4. Apply a Strong Web Application Firewall (WAF)

A **WAF** can detect and prevent HTTP Request Smuggling attacks by filtering and blocking malicious requests. By implementing signature-based rules, a WAF can spot unusual patterns, like conflicting headers or malformed requests.

How to Implement:

- Enable **mod_security** for Apache or **WAF rules** for Nginx to detect HTTP request smuggling attempts.

Example:

```
SecRule REQUEST_HEADERS:Transfer-Encoding "@rx
chunked" "id:10001,deny,log,status:400"
SecRule REQUEST_HEADERS:Content-Length "@rx [^0-
9]" "id:10002,deny,log,status:400"
```

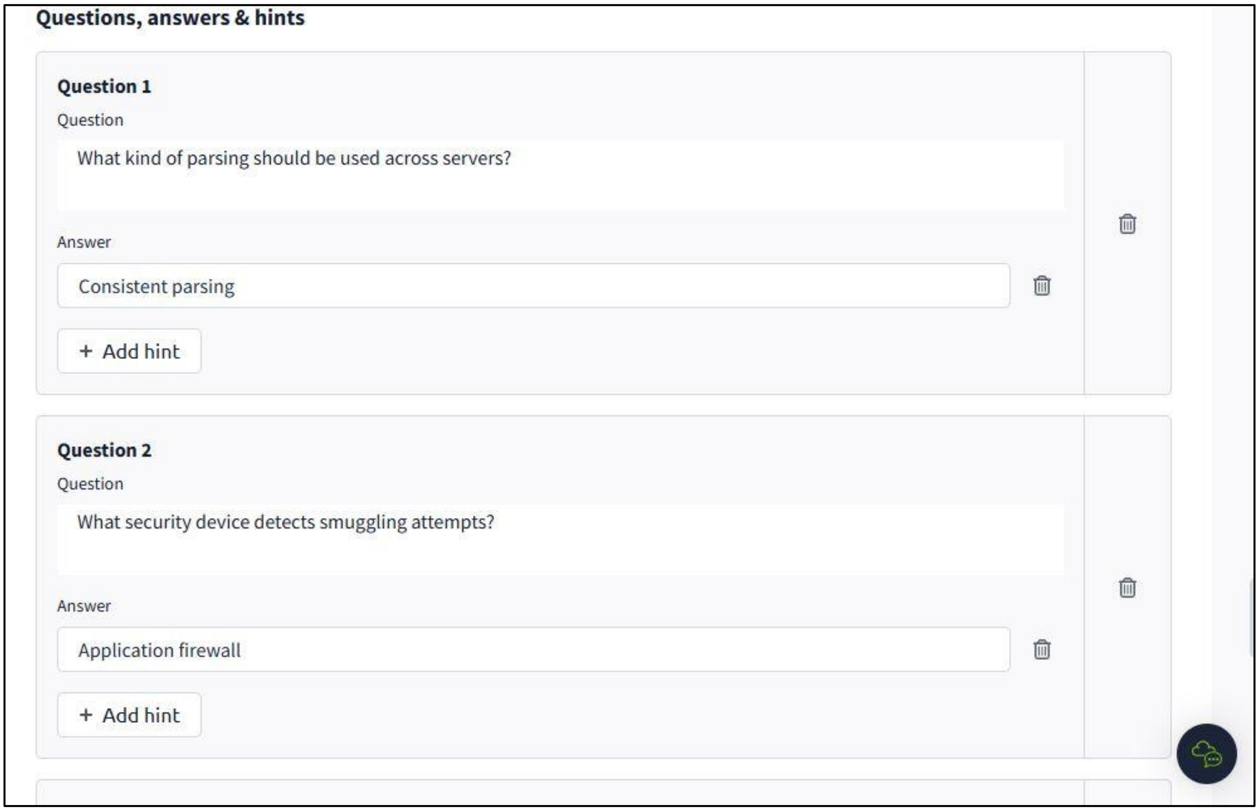
5. Use Proper Request Length and Chunking Validation

Request **length validation** is crucial to avoid HTTP Request Smuggling. Both the **Content-Length** and **Transfer-Encoding** headers need to be **properly validated** to avoid conflicts and inconsistencies.

How to Implement:

- **Strictly enforce the Content-Length**: Ensure that the content length matches the body of the request.
- **Inspect for unexpected chunked transfer encoding**: Ensure that requests using **Transfer-Encoding: chunked** don't also

At the end this task contains some small questions for the users.



Questions, answers & hints

Question 1

Question

What kind of parsing should be used across servers?

Answer

Consistent parsing

Question 2

Question

What security device detects smuggling attempts?

Answer

Application firewall

Task 10 (CTF)

Name – Agent's Final CTF: Smuggle, Split, and Conquer

Objective - Exploit the simulated HTTP Request Smuggling and Response Splitting vulnerabilities to uncover, decode, and submit the hidden flag.

Type – practical (Hard)

Task description-

In this final CTF style simulation task, we explore the combination of HTTP Request Smuggling and HTTP Response Splitting that user learned in previous tasks to perform a CTF challenge.

Instead of launching a real-world multi-server attack (which would be complex and dangerous), we simulate these vulnerabilities safely by detecting specially crafted requests and response manipulations.

Users will learn how attackers combine smuggling and splitting to access hidden admin resources.

Task Breakdown -

Exploit Scenario Description

In real-world attacks, combining **HTTP Request Smuggling** with **HTTP Response Splitting** allows attackers to,

- Bypass authentication layers
- Access hidden internal functionality (like admin panels)
- Inject fake responses into victim connections
- Hijack sessions or poison server caches

They achieve this by confusing the server's request parsers and manipulating HTTP headers to split responses.

In this simulation task,

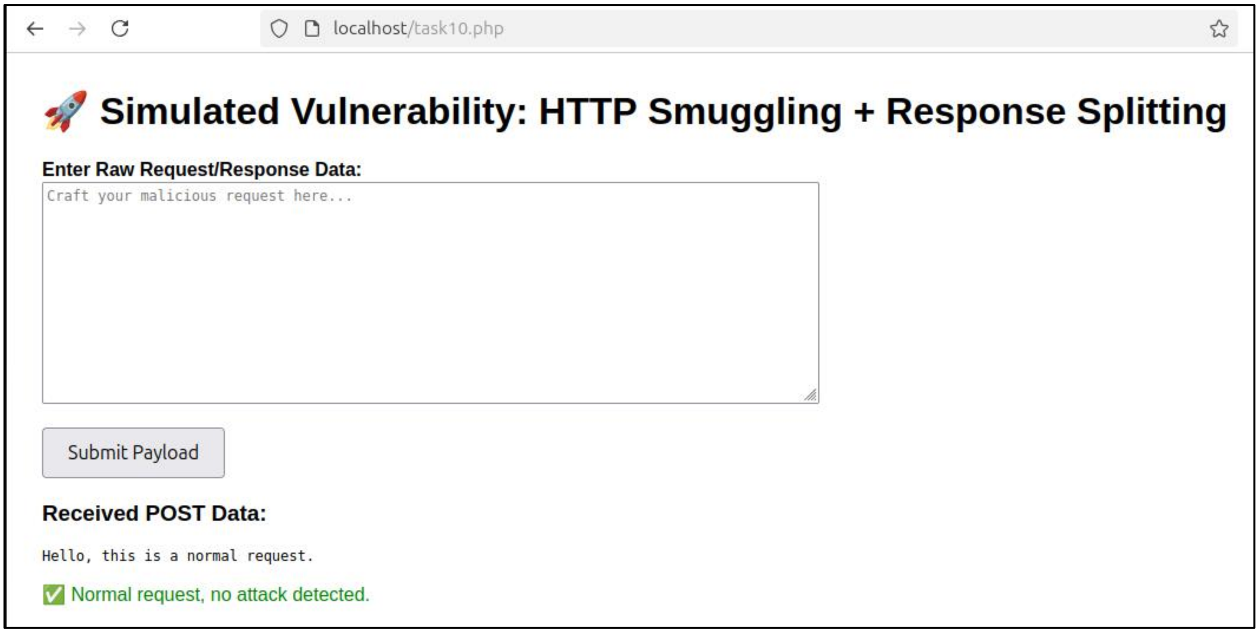
- A form submits raw HTTP-like data to a PHP script.
- If the data contains a **hidden GET /admin HTTP/1.1 request** (smuggling) plus a **fake HTTP/1.1 200 OK header** inserted (response splitting),

Then the server simulates an exploit and reveals a secret encoded flag inside a the admin panel.

This allows us to safely practice the combination of smuggling and splitting concepts without needing real proxy-backend setups.

Expected outcome -

1. Submit normal data like **username=admin&password=1234** via the form - The server responds with a green message: *No attack detected. Normal behavior.*



🚀 Simulated Vulnerability: HTTP Smuggling + Response Splitting

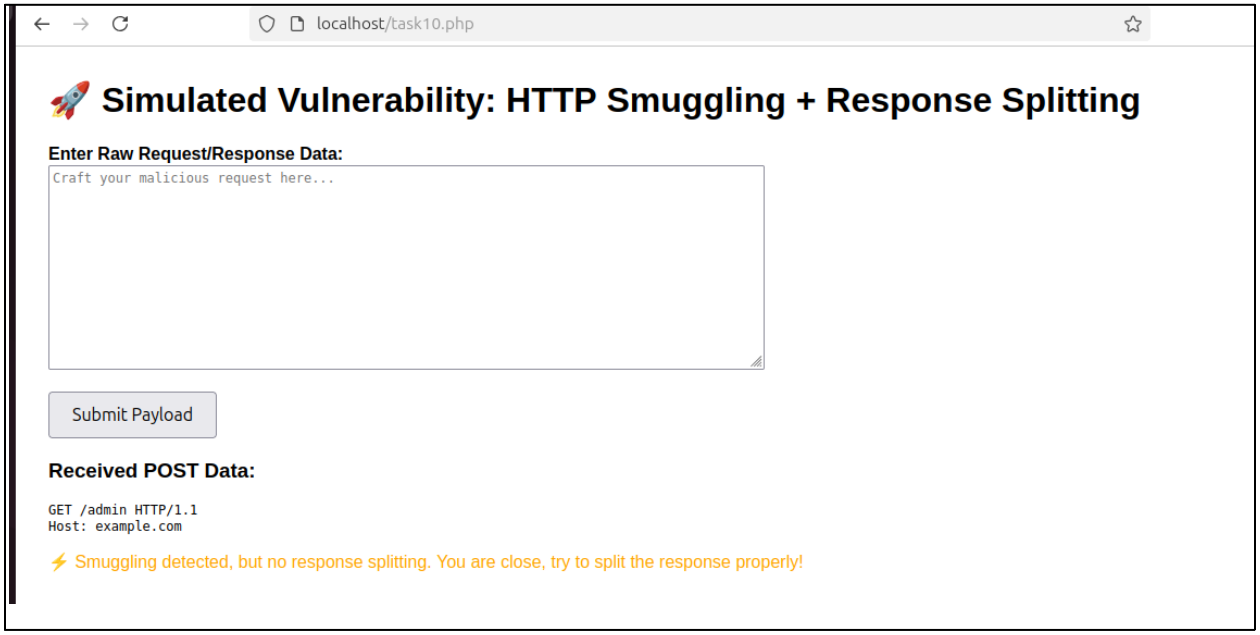
Enter Raw Request/Response Data:

Received POST Data:

```
Hello, this is a normal request.
```

✅ Normal request, no attack detected.

2. Submit an only-smuggling payload like
GET /admin HTTP/1.1
Host: internal **-** The server detects smuggling and warns: *Smuggling detected, no response splitting. You are close!*



🚀 Simulated Vulnerability: HTTP Smuggling + Response Splitting

Enter Raw Request/Response Data:

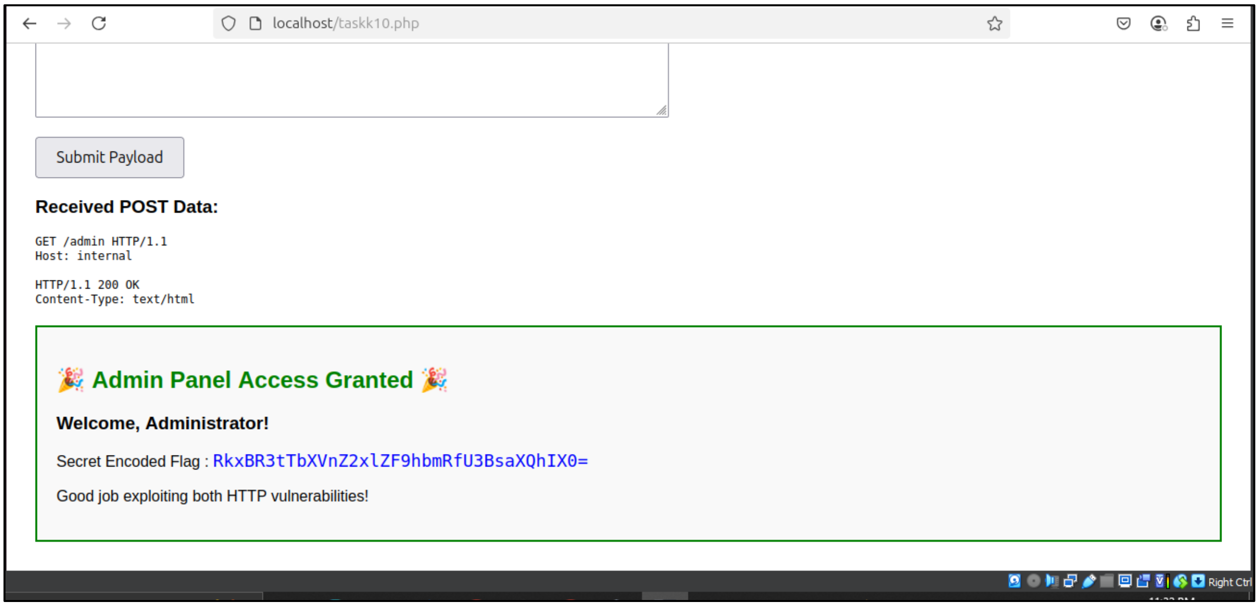
Received POST Data:

```
GET /admin HTTP/1.1
Host: example.com
```

⚡ Smuggling detected, but no response splitting. You are close, try to split the response properly!

3. Submit a payload combining smuggling and splitting like
   GET /admin HTTP/1.1
   Host: internal\r\n\r\nHTTP/1.1 200 OK\r\nContent-Type: text/html\r\n\r\n

The server detects both smuggling and splitting and grants access to the simulated admin panel, showing the encoded flag!



Received POST Data:

```
GET /admin HTTP/1.1
Host: internal

HTTP/1.1 200 OK
Content-Type: text/html
```

🎉 Admin Panel Access Granted 🎉

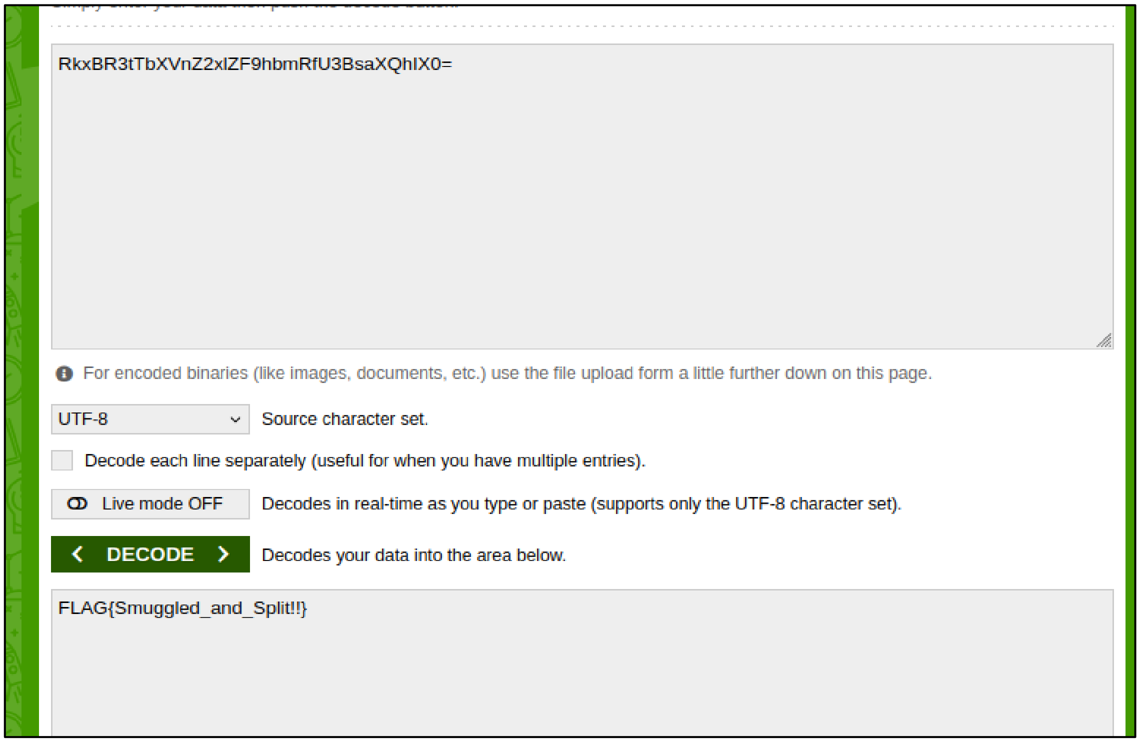
Welcome, Administrator!

Secret Encoded Flag : RkxBR3tTbXVnZ2xlZF9hbmRfU3BsaXQhIX0=

Good job exploiting both HTTP vulnerabilities!

Finally, the flag you find is Base64 encoded. So, you must decode it first before submitting it as the final answer.

Final flag = FLAG{Smuggled_and_Split!!}



```
RkxBR3tTbXVnZ2xlZF9hbmRfU3BsaXQhIX0=
```

For encoded binaries (like images, documents, etc.) use the file upload form a little further down on this page.

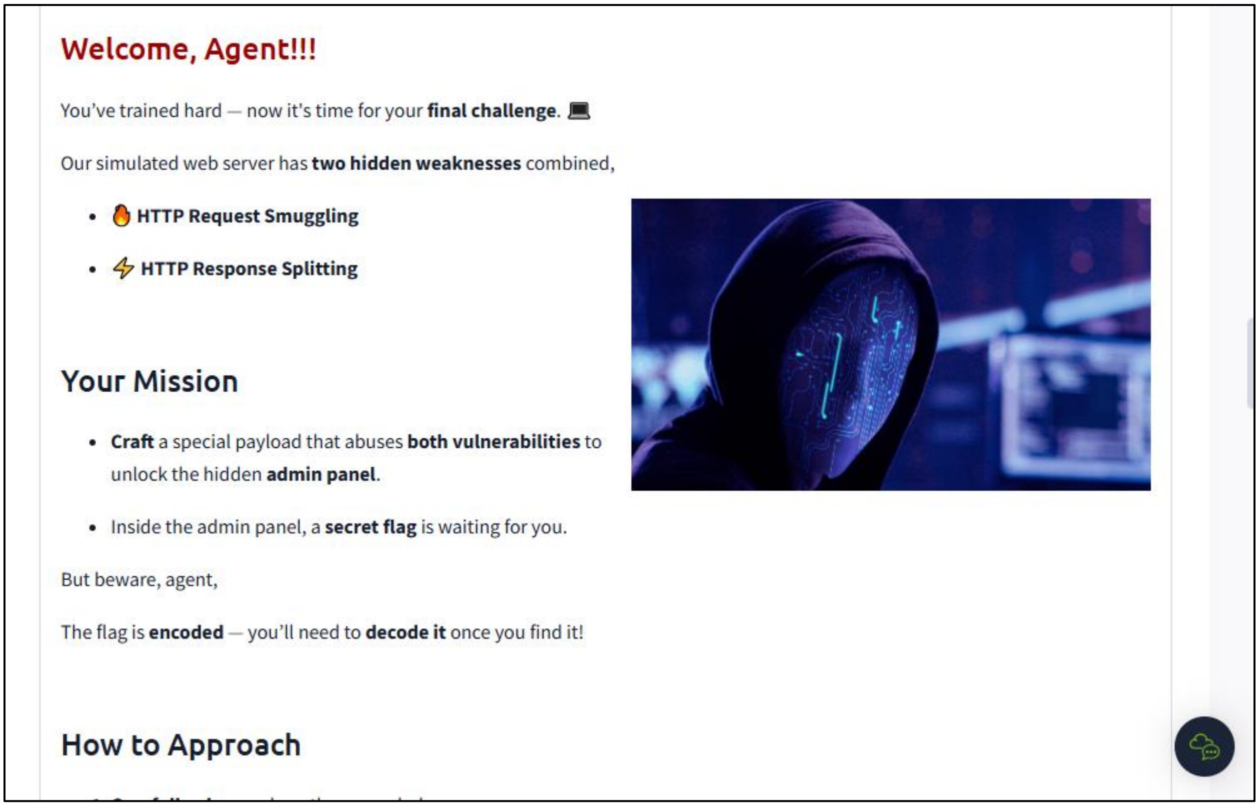
UTF-8 Source character set.

□ Decode each line separately (useful for when you have multiple entries).

Live mode OFF Decodes in real-time as you type or paste (supports only the UTF-8 character set).

< DECODE > Decodes your data into the area below.

```
FLAG{Smuggled_and_Split!!}
```



Welcome, Agent!!!

You've trained hard — now it's time for your **final challenge**. 💻

Our simulated web server has **two hidden weaknesses** combined,

- 🔥 **HTTP Request Smuggling**
- ⚡ **HTTP Response Splitting**

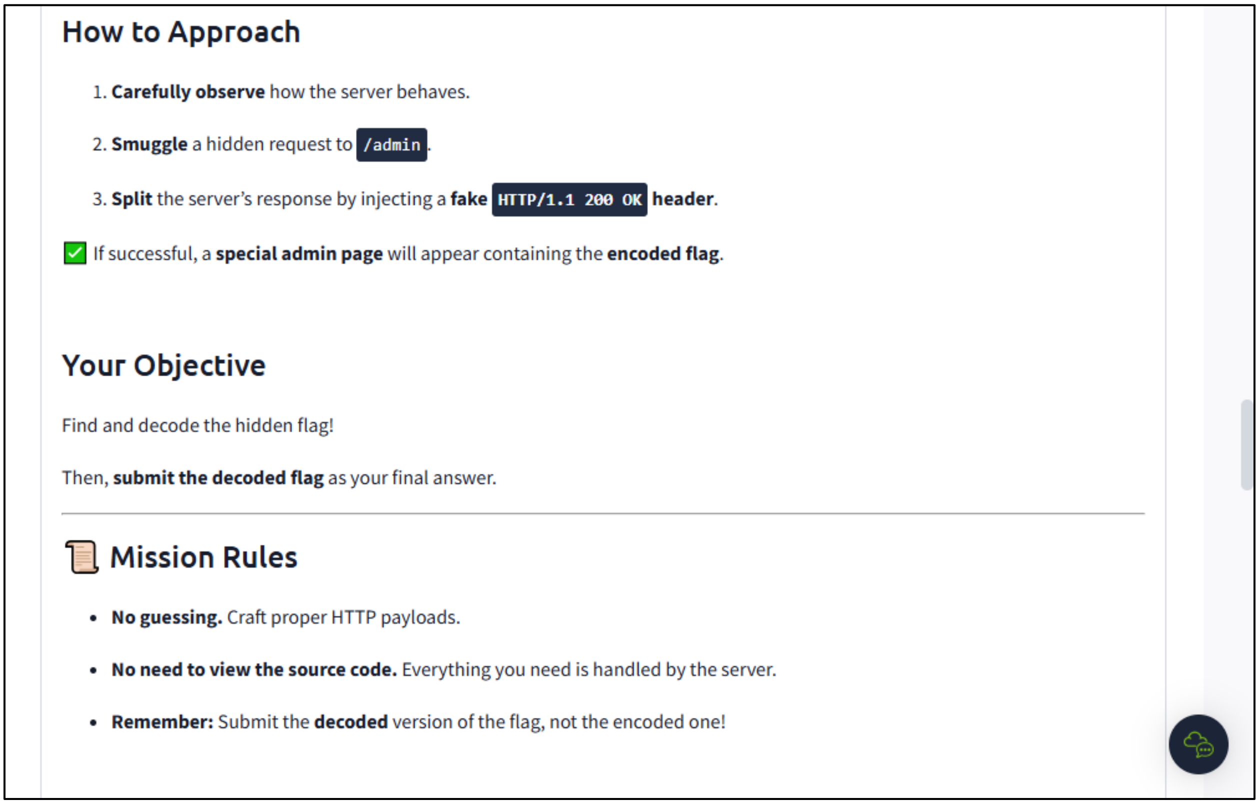
Your Mission

- **Craft** a special payload that abuses **both vulnerabilities** to unlock the hidden **admin panel**.
- Inside the admin panel, a **secret flag** is waiting for you.

But beware, agent,

The flag is **encoded** — you'll need to **decode it** once you find it!

How to Approach



How to Approach

1. **Carefully observe** how the server behaves.
2. **Smuggle** a hidden request to `/admin`.
3. **Split** the server's response by injecting a **fake** `HTTP/1.1 200 OK` **header**.

✅ If successful, a **special admin page** will appear containing the **encoded flag**.

Your Objective

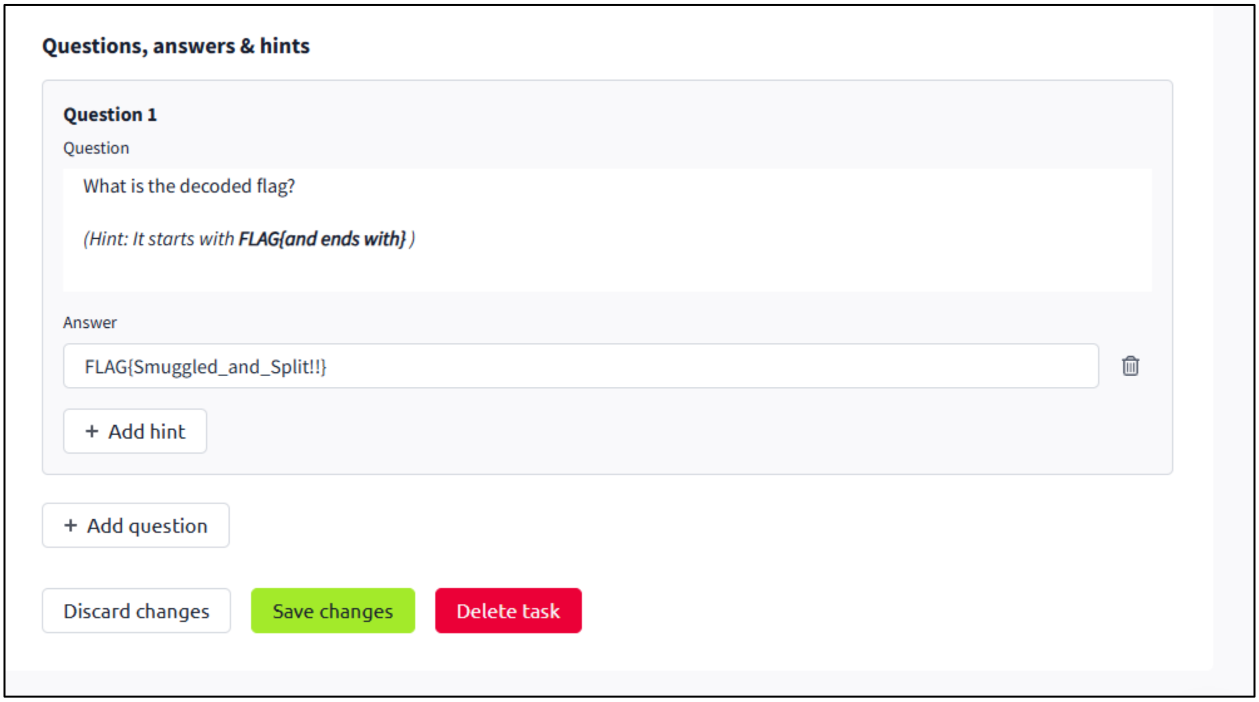
Find and decode the hidden flag!

Then, **submit the decoded flag** as your final answer.

📜 Mission Rules

- **No guessing.** Craft proper HTTP payloads.
- **No need to view the source code.** Everything you need is handled by the server.
- **Remember:** Submit the **decoded** version of the flag, not the encoded one!

At the end this task contains some small questions for the users.



Questions, answers & hints

Question 1

Question

What is the decoded flag?

*(Hint: It starts with **FLAG{and ends with}**)*

Answer

FLAG{Smuggled_and_Split!!}

Room link

https://tryhackme.com/jr/hackershttpsplitandsmugg

Reflection

Personal insights/lessons learned while designing the room

I learned many facts while designing this room. Most of them are listed below.

- Web vulnerabilities are subtle

Small differences in request handling by a server may cause big security problems.

- Real-world security threats

HTTP request smuggling and response splitting exploits can generate an unauthorized access problem, that will escalate into a major security breach.

- Mitigation is just as important.

Understanding how to protect against exploits is just as valid as understanding how to exploit.

- Encourages exploration

Testing payloads and hiding exploits helps develop critical thinking.

- Simulated environment is critical

Safer simulated attacks allow learners to explore their vulnerabilities.

- Stepwise learning works

Constructing a task into a number of simple steps helps the learners to understand elaborate exploits.

- Crafting payloads is crucial

Constructing malicious payloads and finding hidden exploits is a key area of study in security.

- Decoding is part of the game

Understanding the encoding of data (for example, Base64) is required for finding the hidden flags or information.

- A Balance between challenge and learning

Tasks need to be created in a way that is very much challenging yet still reasonably achievable by learners.

- Simulate, Don't Break.

Simulated training allows for the learning of real exploitation and defense against real vulnerabilities.

How the room contributes to the learning community

- Hands-on experience

Provides real-life, practical scenarios for users to engage with HTTP vulnerabilities like request smuggling and response splitting.

- Skill-building

Builds technical skills by simulating advanced security attacks in a secure, risk-free environment, familiarizing learners with common web vulnerabilities.

- Develops critical thinking

Stimulates learners to come up with specific attack payloads, which induce problem-solving and innovation.

- Promote collaboration

Promotes discussion and exchange of knowledge because learners can compare different solutions and approaches within the community.

- Real-world relevance

The vulnerabilities simulated are very much related to web security concerns nowadays, so this offers users knowledge they can relate to current security practices.

- Promotes security awareness

Educates learners about HTTP-based vulnerabilities, which are common but not always targeted, to promote more secure web development practice.