Cybersecurity Wargame

Internship Task Report

Sathaye College vileparle east

Program: Digisuraksha Parhari

Foundation Internship Issued By: Digisuraksha Parhari Foundation

Supported By: Infinisec Technologies Pvt. Ltd.

Team members:

Name	Roll no	Labs
• Sneha Devkate	162	Leviathan
• Yashashri Sawant	206	Krypton
• Sonali Pawar	195	Natas

Leviathan

Completed by: Sneha Madhukar Devkate /roll no.162

Objective:

The primary goal of Leviathan is to progress through each level by discovering the password for the next user account. This is achieved by identifying and exploiting common security oversights within a Linux environment. The challenges are crafted to be approachable, requiring no prior programming knowledge—just a solid understanding of basic Unix/Linux commands and a problem-solving mindset.

Key concept:

- **File and Directory Permissions**: Understanding how permissions affect access and how misconfigurations can lead to vulnerabilities.
- **Hidden Files and Directories**: Using commands like ls -la to uncover hidden content that may contain crucial information.
- **Binaries**: Recognizing and exploiting binaries that execute with elevated privileges.
- **Command-Line Tools**: Utilizing tools such as **strings**, **ltrace**, to analyse and manipulate binaries.

Level 0 to 1:

• Objective:

Locate the password for leviathan1.

• Tools Used:

ls, cd, cat,grep

• Solution Logic:

1. Access the .backup directory:

cd .backup

2. Identify the bookmarks.html file.

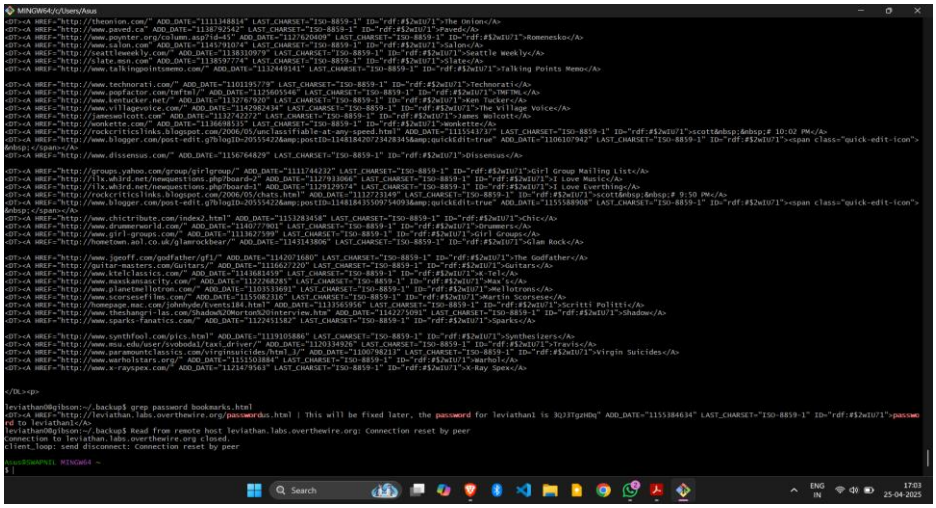
3. Search for the keyword "password" inside the file:

grep "password" bookmarks.html

4. Extract the password from the matching line.

• Password:

3QJ3TgzHDq



Level 1 to 2:

• Objective:

Discover the password for leviathan2.

• Tools Used:

file, ltrace

• Solution Logic:

1. Identify the check binary using ls.

2. Confirm it's an executable file using:

file check

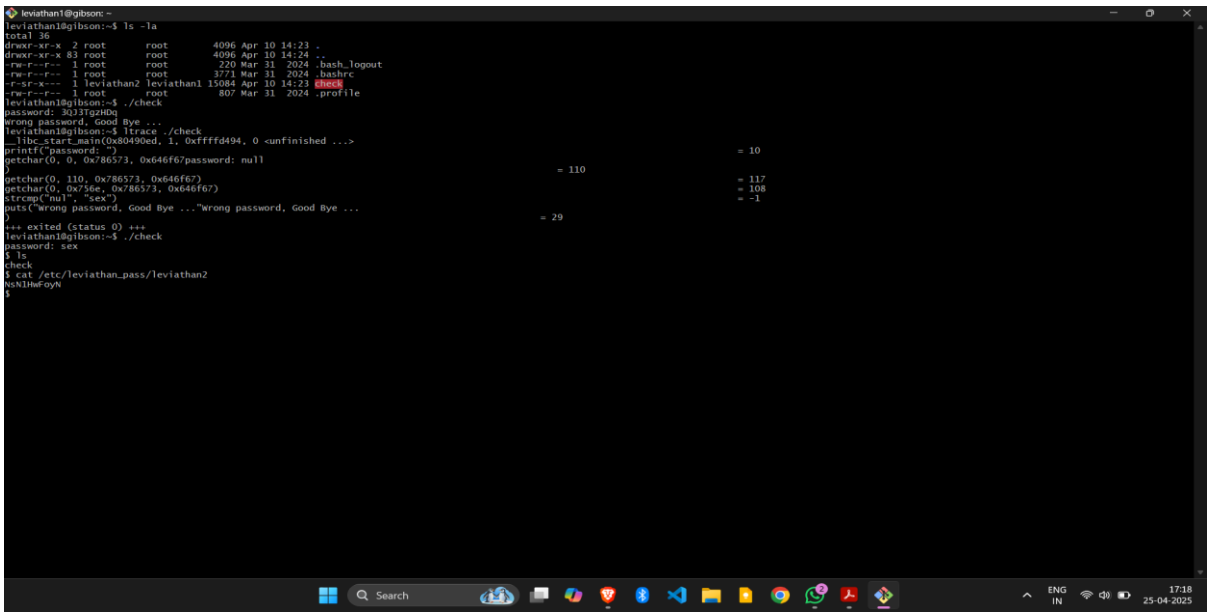
3. Use ltrace to trace library calls and spot the password comparison:

ltrace ./check

4. Use the discovered password to successfully execute the binary.

• Password:

NSN1HwFoyN



Level 2 to 3:

Objective:

Gain access to leviathan3.

• Tools Used:

ltrace, touch, mkdir, bash

• Solution Logic:

1. Identify the printfile binary using ls.

2. Use ltrace to observe how the binary processes input:

ltrace ./printfile

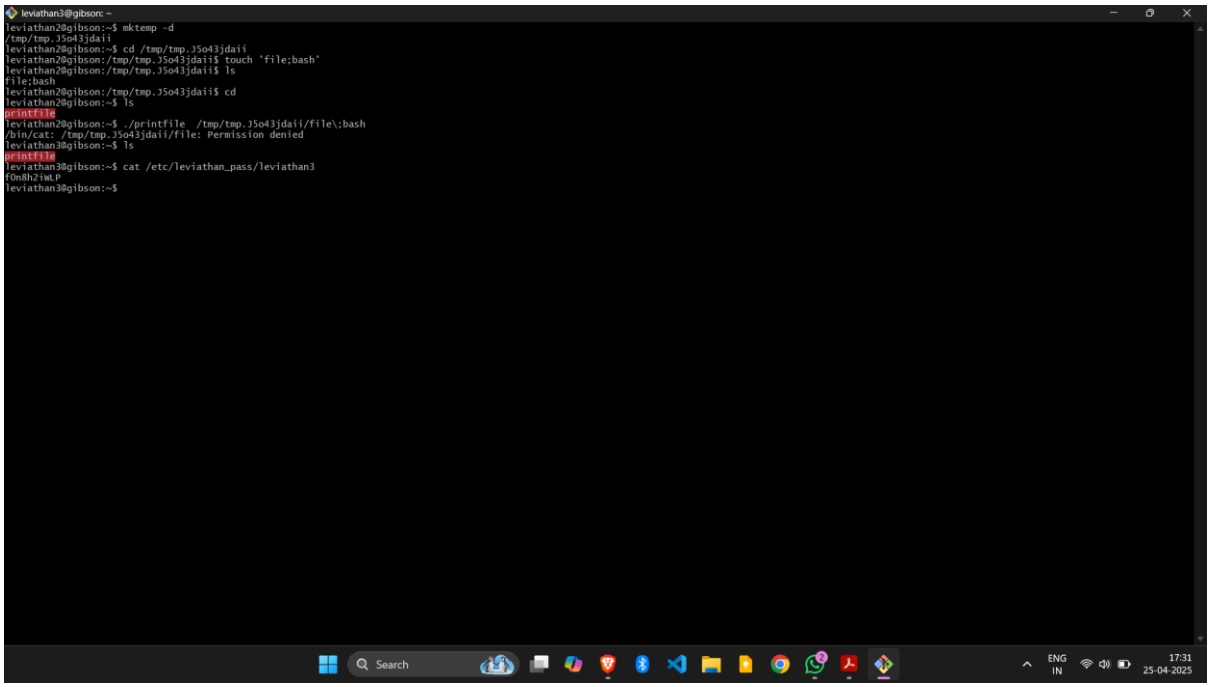
3. Notice that it improperly handles filenames.

4. Craft a file or directory name containing a command injection, such as test; bash.

5. Run the binary with the crafted name to trigger a shell.

• Password:

F0n8h2iWLP



Level 3 to 4:

• Objective:

Retrieve the password for leviathan4.

• Tools Used:

ltrace

• Solution Logic:

1. Locate the level3 binary using ls.

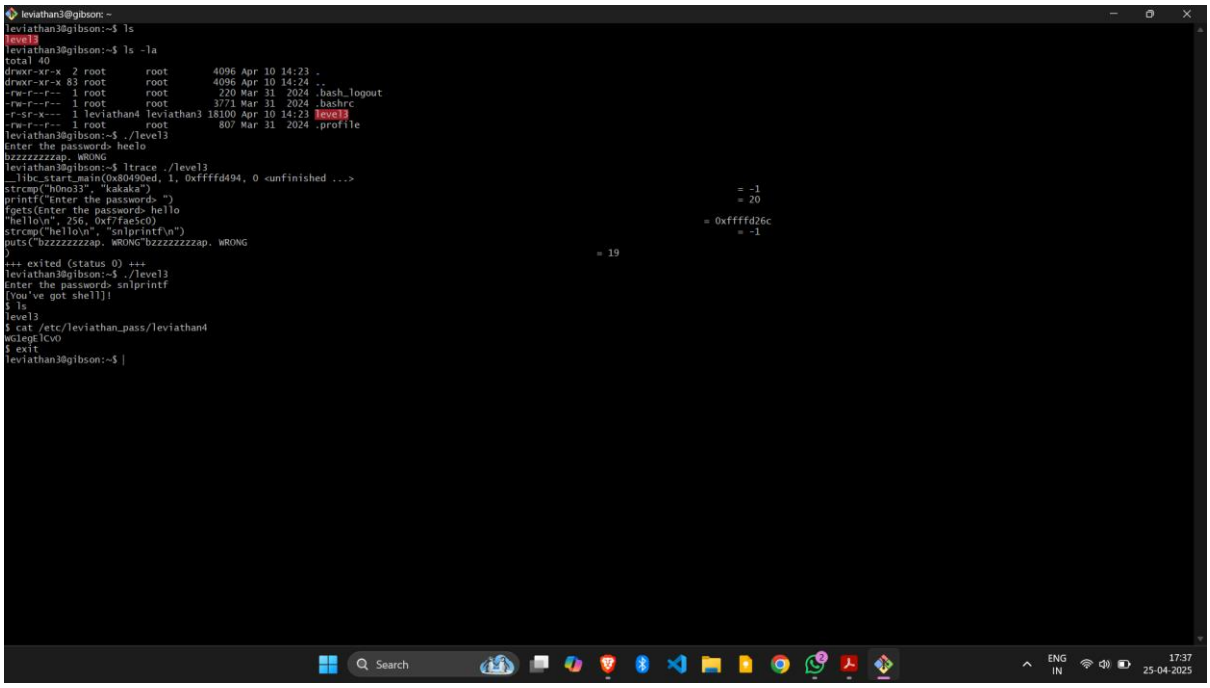
2. Use ltrace to observe the function calls and spot the expected password:

ltrace ./level3

3. Enter the correct password when prompted to gain access.

• Password:

wGlegElCv0



Level 4 to 5:

Objective:

Find the password for leviathan5.

• Tools Used:

ls, cd, ./bin, binary-to-ASCII conversion

• Solution Logic:

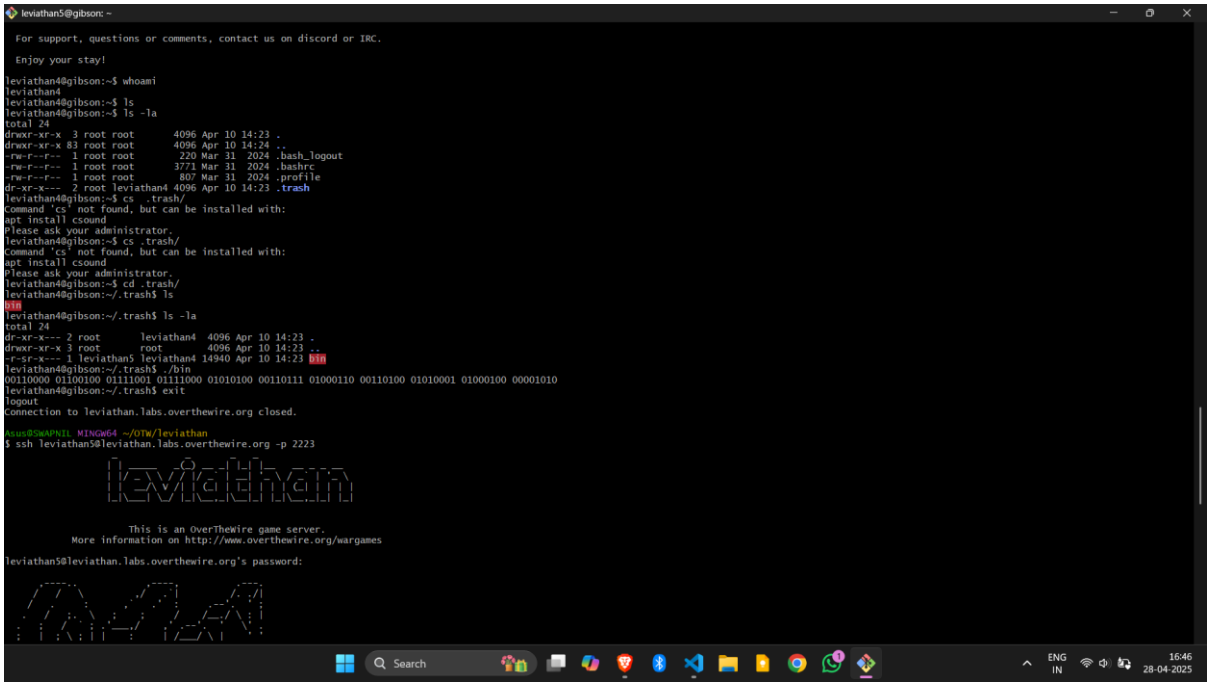
1. Navigate into the .trash directory using cd .trash.

2. Run the bin executable to get binary output:

./bin

3. Convert the binary output to ASCII (you can use online converters or a script) to reveal the password.

• Password:



Level 5 to 6:

• Objective:

Access leviathan6.

• Tools Used:

ln, symbolic links

• Solution Logic:

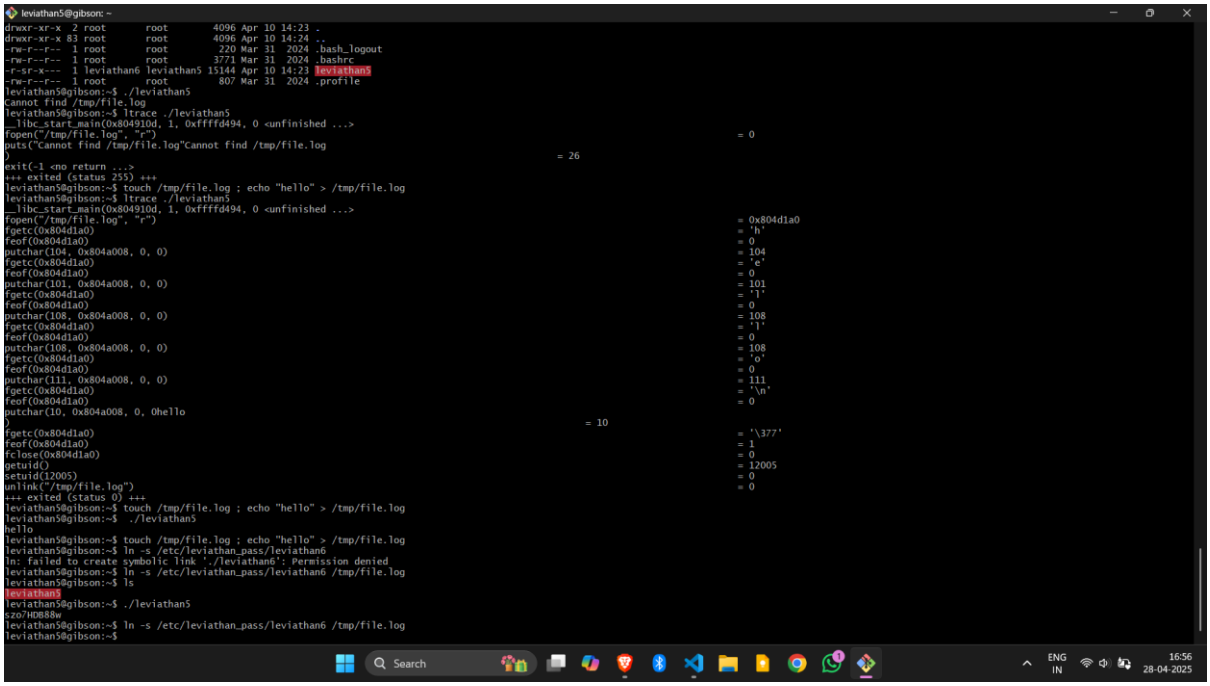
1. Create a symbolic link in /tmp that points to the password file:

ln -s /etc/leviathan_pass/leviathan6 /tmp/file.log

2. Run the leviathan5 binary, which reads from /tmp/file.log, revealing the password.

• Password:

Sz07HDB88w



Level 6 to 7:

• Objective:

Obtain the password for leviathan7.

• Tools Used:

Python scripting, brute-force approach

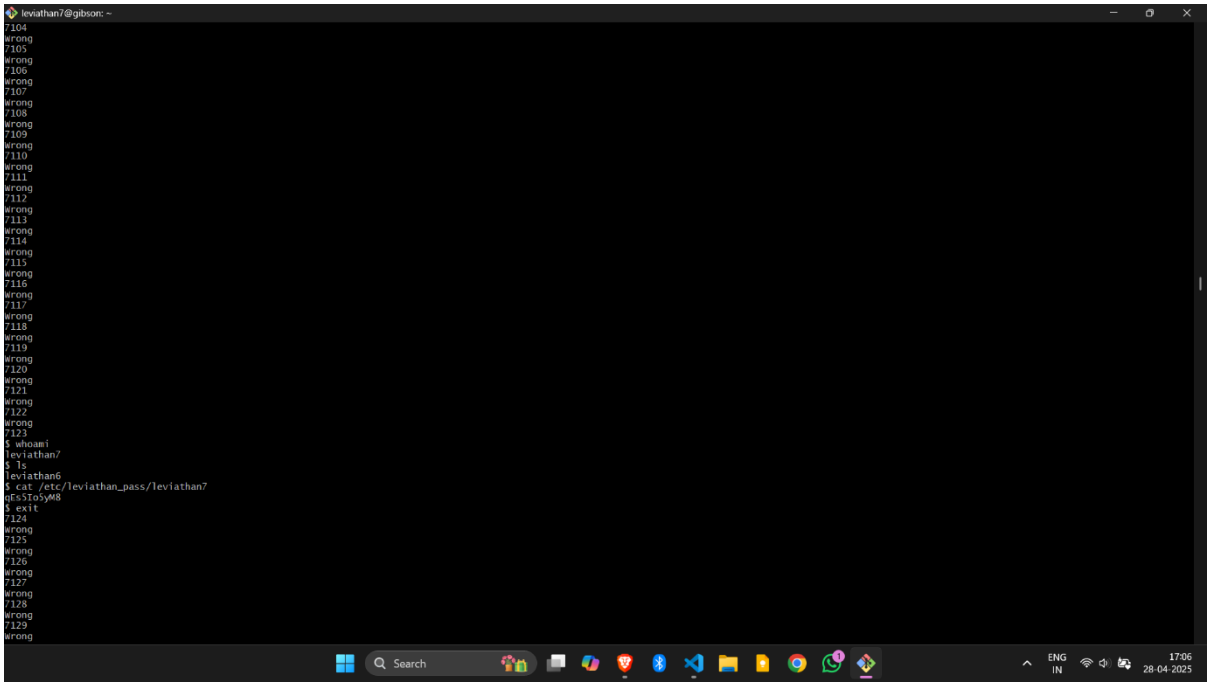
• Solution Logic:

1. Write a Python script to brute-force the 4-digit PIN required by the leviathan6 binary.

2. Iterate through all possible combinations until the correct PIN is found and access is granted.

• Password:

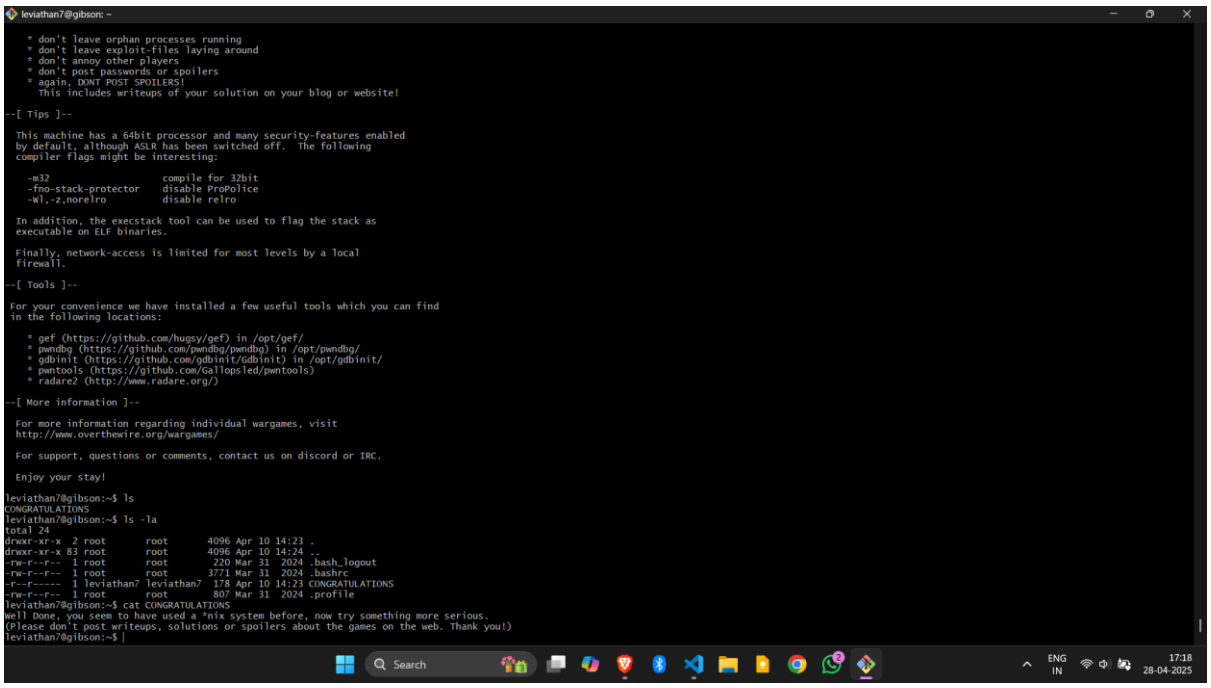
qEs5Io5yM8



Level 7 to 8:

• Objective:

Complete the final level.



Conclusion:

The OverTheWire Leviathan lab helped me improve my basic Linux and security skills. By solving small challenges, I learned how to find hidden files, understand permissions, and work with simple binaries. It was a good practice to build my problem-solving skills, and now I feel more confident to move on to tougher cybersecurity tasks.

Name:Yashashri Rajan Sawant-206-Sathaye college

OverTheWire Krypton Lab Report (Levels 0–7)

Objective

The OverTheWire Krypton lab is designed to build foundational cryptographic skills through practical challenges. Participants learn to identify, analyze, and break various classical ciphers, gaining hands-on experience with encoding schemes, substitution ciphers, polyalphabetic ciphers, and stream ciphers. The goal is to develop both theoretical understanding and practical problem-solving abilities in cryptanalysis.

Tools Used

- Linux command line utilities for remote access and file handling.
- Base64 decoding tools.
- Cryptanalysis platforms like CyberChef .
- Text editors for examining ciphertext.
- Python scripting for automating repetitive tasks.
- Knowledge of classical cipher techniques and cryptanalysis methods.

Level 0-1

Challenge

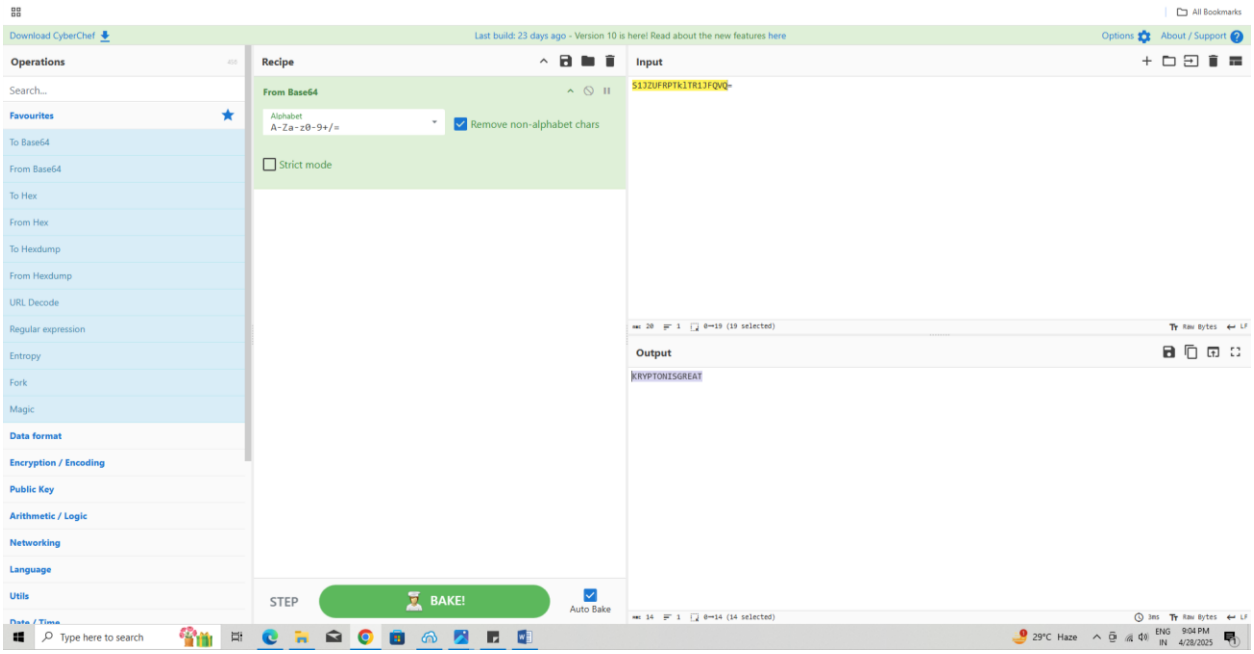
The initial challenge required logging into the Krypton server using SSH, but the password was not given in plain text. Instead, it was encoded in Base64, a common encoding scheme that translates binary data into ASCII characters.

Solution

The solution involved recognizing that the password was Base64 encoded. By decoding the Base64 string, I retrieved the original password. This step was straightforward but essential, as it introduced the concept of data encoding, which is often mistaken for encryption. After decoding, I used the password to successfully log in to the server via SSH.

What I Learned

- The difference between encoding and encryption.
- How to decode Base64 encoded data.
- Basic remote login procedures using SSH.

- CYBERCHEF website below:
-


Level 1-2

PASSWORD: KRYPTONISGREAT

Challenge

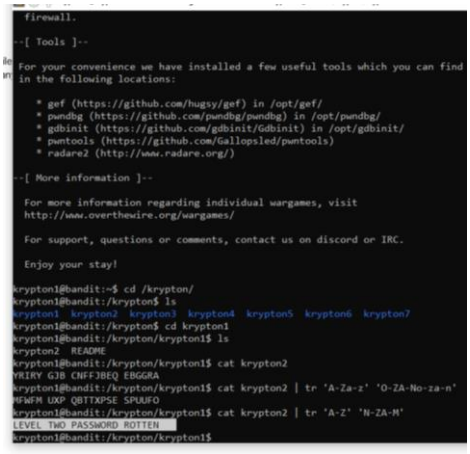
The password for the next level was encrypted using a Caesar cipher, which shifts each letter by a fixed number of positions in the alphabet.

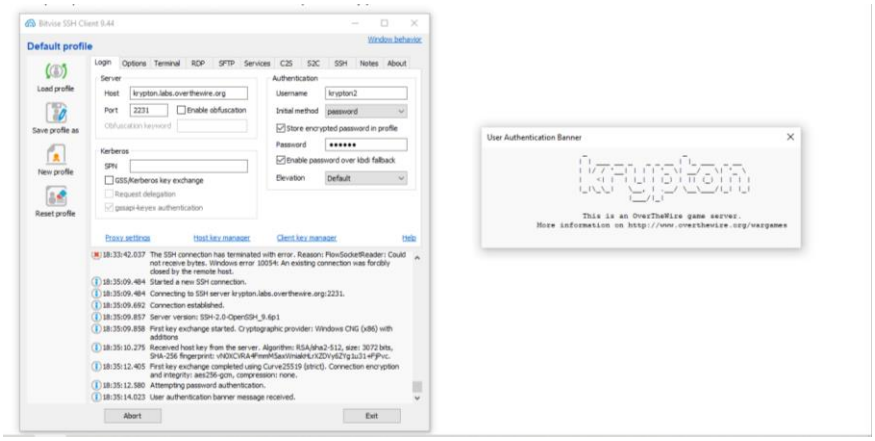
Solution

I identified the cipher as a Caesar shift, likely ROT13, a common variant where letters are shifted by 13 places. By applying the reverse shift, the ciphertext was transformed back into readable plaintext. This process involved understanding how the alphabet wraps around and how each letter is shifted consistently. Recognizing the pattern and applying the inverse operation revealed the password.

What I Learned

- The mechanics of the Caesar cipher and letter shifting.
- How to apply inverse operations to decrypt substitution ciphers.
- The simplicity and vulnerability of fixed-shift ciphers.





Level 2-3

PASSWORD: ROTTEN

Challenge

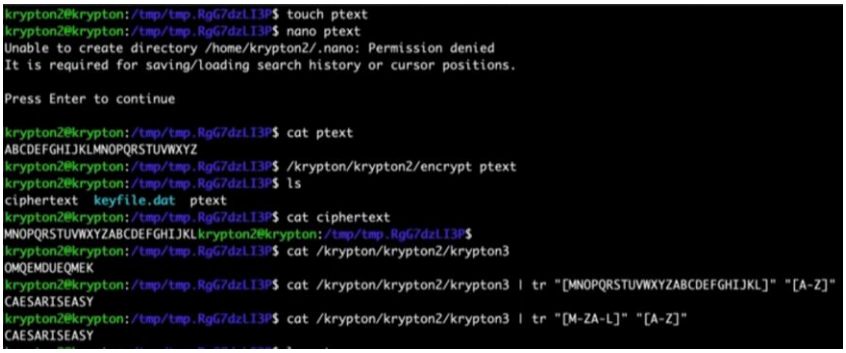
Again, the password was encrypted with a Caesar cipher, but this time the shift value was unknown, requiring a more methodical approach.

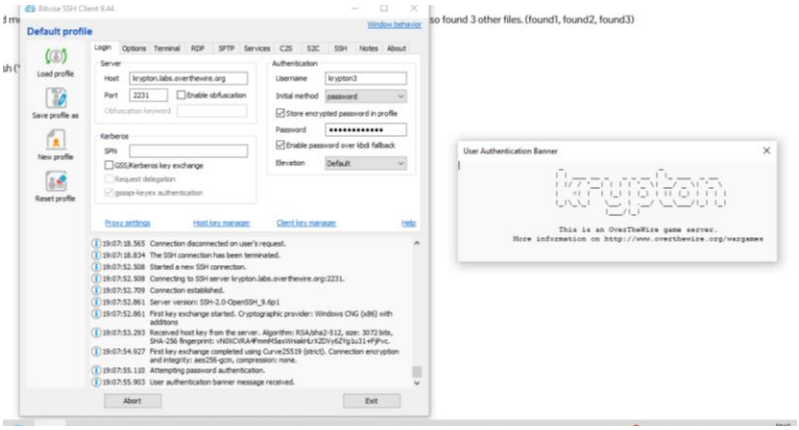
Solution

Without knowledge of the shift, I used a brute-force approach, systematically trying all possible shifts from 1 to 25. Each shifted text was examined to identify meaningful English words. This trial-and-error method is effective against simple ciphers with small key spaces. The correct shift produced readable text, revealing the password.

What I Learned

- The effectiveness of brute-force attacks on weak ciphers.
- The importance of recognizing language patterns to identify correct decryptions.
- How automation can streamline repetitive cryptanalysis tasks.





Level 3-4

PASSWORD: CAESARISEASY

Challenge

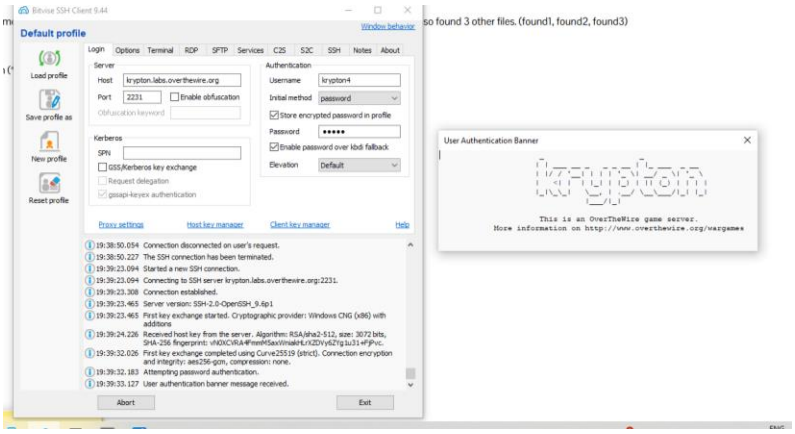
The password was encrypted with a Caesar cipher, but the shift was not a common value and had to be deduced by analyzing an encryption program provided.

Solution

I studied the encryption program to understand its logic and how it applied the shift. By encrypting known plaintext inputs using the program and comparing the outputs, I was able to deduce the exact shift value. This reverse-engineering approach allowed me to decrypt the original ciphertext and retrieve the password.

What I Learned

- How to analyze and interpret encryption code to uncover keys.
- The value of chosen plaintext attacks, where encrypting known data helps reveal encryption parameters.
- The practical application of cryptanalysis beyond theoretical methods.



Level 4-5

PASSWORD: BRUTE

Challenge

The password was encrypted with a Vigenère cipher, a polyalphabetic cipher that applies multiple Caesar shifts based on a keyword.

Solution

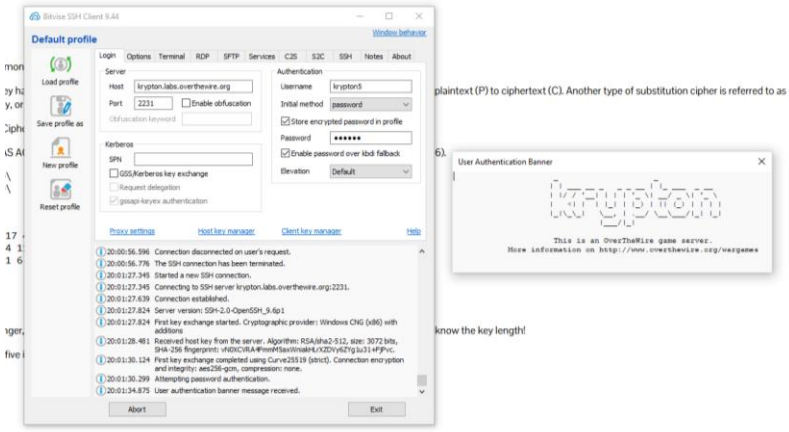
With the key length provided, I segmented the ciphertext accordingly. Each segment corresponds to letters encrypted with the same Caesar shift. By performing frequency analysis on each segment, I identified the most likely shift values (key letters). Using these shifts, I reconstructed the keyword and decrypted the ciphertext to reveal the password.

What I Learned

- How polyalphabetic ciphers complicate frequency analysis.
- The advantage of knowing the key length in breaking Vigenère ciphers.

- Techniques for segmenting ciphertext and analyzing each segment independently.

```
online learning > freq_analysis.py
import sys

if __name__=="__main__":

    char_table = {}
    char_total = 0
    groupsize = 0

    #Check Argument Counts
    if len(sys.argv) != 3:
        print("Usage: python3 freq_analysis.py filename groupsize")
        exit(0)
    else:

        try:
            groupsize = int(sys.argv[2])
        except:
            print("groupsize must be an int")
            exit(0)

        # Try to Open the specified file
        try:
            with open(sys.argv[1]) as fh:
                lines = fh.readlines()
        except:
            print("No file named '" + sys.argv[1] + "'")
            exit(0)

        for line in lines:
            line = line.replace(" ", "")
            line = line.replace("\n", "")
            for i in range(len(line) - groupsize):
                group = line[i:i+groupsize]
                if group in char_table:
                    char_table[group] += 1
                else:
                    char_table[group] = 1

        char_table = sorted(char_table.items(), key=lambda x: x[1], reverse=True)

        for char in char_table:
            print(char[0] + ":\t" + str(char[1]))
```



Level 5-6

PASSWORD:RANDOM

Challenge

The password was again encrypted with a Vigenère cipher, but this time the key length was unknown.

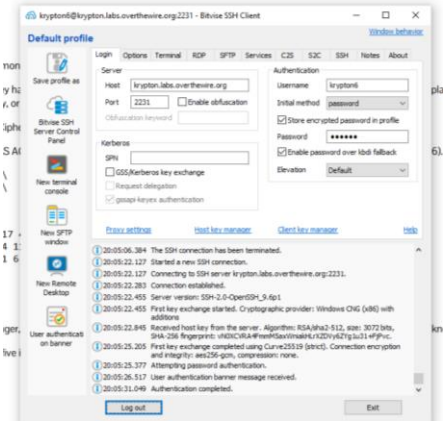
Solution

I applied cryptanalytic techniques such as the Kasiski examination and Friedman test to estimate the key length. These methods analyze repeated patterns and statistical properties of the ciphertext to infer likely key lengths. Once the key length was estimated, I proceeded with frequency analysis on each segment to deduce the key and decrypt the message.

What I Learned

- How to estimate unknown key lengths using statistical methods.
- The importance of pattern repetition in polyalphabetic cipher analysis.
- Combining multiple cryptanalytic techniques to solve more complex challenges.

```python
import sys

if __name__=='__main__':

    rep_seq = []
    rep_seq_spaces = []
    tallys = [0]*20

    if len(sys.argv) != 2:
        print("Usage: python3 keyLength.py filename")
        exit(0)
    else:

        try:
            with open(sys.argv[1]) as fh:
                lines = fh.readlines()
        except:
            print("No file named '" + sys.argv[1] + "'")
            exit(0)

        for line in lines:
            line.replace(" ", "")
            line.replace("\n", "")
            for length in range(3,5+1):
                for i in range(len(line)-length+1):
                    q = line[i:i+length]
                    for j in range(i+1,len(line)-length+1):
                        r = line[j:j+length]
                        if q == r:
                            rep_seq.append(q)
                            rep_seq_spaces.append(j-i)

        for n in rep_seq_spaces:
            for i in range(1,20):
                if n % i == 0:
                    tallys[i] = tallys[i] + 1

        s = sum(tallys)
        if s == 0:
            print("No repeating sequences")
        else :
            print("Chance key length is: ")
            for i in range(1,len(tallys)):
                print(str(i) + " = " + str(round((tallys[i]/s)* 100,2)) + "%")
```



Level 6-7

PASSWORD: LFSRISNOTRANDOM

Challenge

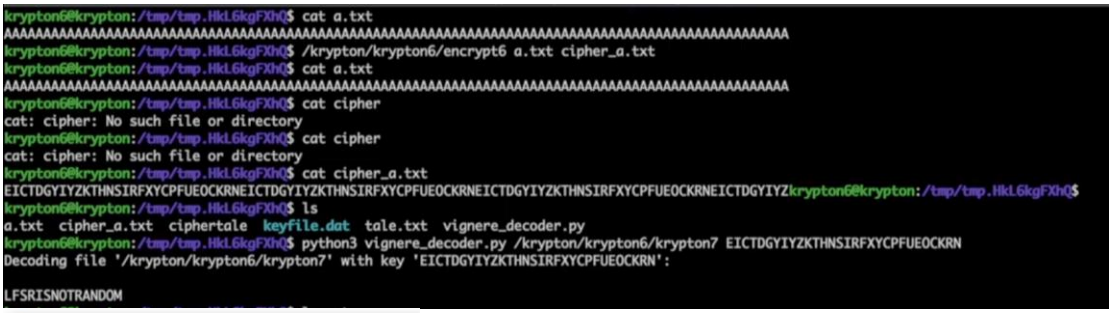
This level involved a stream cipher, where the ciphertext was produced by XORing plaintext with a key stream. The goal was to recover the key stream to decrypt the message.

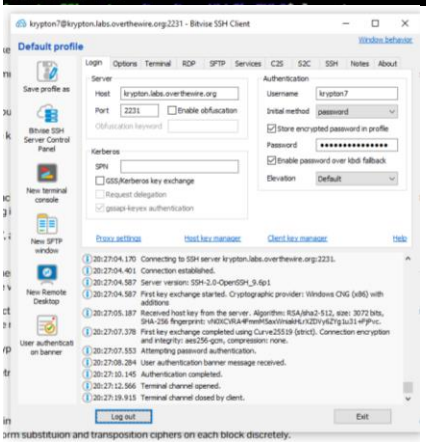
Solution

I exploited the properties of XOR operations in stream ciphers. By encrypting a known plaintext (such as a string of identical characters), I obtained a ciphertext that directly revealed the key stream when compared to the known plaintext. Using the recovered key stream, I decrypted the original ciphertext to retrieve the password.

What I Learned

- The fundamental operation of stream ciphers and XOR encryption.
- How chosen plaintext attacks can expose key streams if the cipher is improperly implemented.
- The critical role of key randomness and secrecy in stream cipher security.





Conclusion

The OverTheWire Krypton lab effectively guided me through a variety of classical cryptographic challenges, each requiring a deeper level of understanding and more sophisticated cryptanalysis techniques. Starting with simple encoding and substitution ciphers, the lab gradually introduced polyalphabetic ciphers and stream ciphers, emphasizing the evolution of cryptographic complexity. The solutions involved a combination of theoretical knowledge, analytical thinking, and practical tool usage. This experience reinforced the importance of cipher design, key management, and the vulnerabilities that arise from predictable or weak encryption methods. Overall, the Krypton lab was an invaluable learning journey into the world of cryptography and cryptanalysis.

Natas

Level 0:

1. Challenge:

When you access the page, you'll see a message saying:

"You can find the password for the next level on this page."

But if you just look at the page normally, you won't spot any password displayed.

2. How to Approach:

- Right-click anywhere on the page and select "View Page Source".
- Alternatively, you can press Ctrl + U (Windows) or Command + Option + U (Mac) to directly open the source.
- The source code reveals the real structure of the page.
- Look through it carefully — especially for anything inside HTML comments (marked by <!-- -->).

3. Solution:

- By reading the page's HTML source, you'll find a hidden comment that contains the password for Level 1.

4. Tools Used:

- Regular web browser.
- Use the View Page Source option.

5. Why This Works (Logic):

- Developers sometimes leave important information inside HTML comments for testing, and they forget to remove it.
- While it's hidden from normal viewers, it's still visible to anyone who checks the page source.
- Tip: Always check the source when something feels missing or hidden!

Level 1:

1. Challenge:

Again, the page says:

"You can find the password for the next level on this page."

However, just like before, nothing is visible in the normal page content.

2. How to Approach:

- This time, instead of just viewing the source, open Developer Tools.
- Press F12 or right-click → Inspect.
- Go to the Elements tab, where you can see the page's live HTML structure.
- Browse through the elements carefully, looking especially for any hidden comments.

3. Solution:

- Inside the HTML structure, you'll find a hidden comment containing the password for Level 2.

4. Tools Used:

- Browser's built-in Developer Tools (specifically the Elements tab).

5. Why This Works (Logic):

- HTML comments are still loaded by your browser, even if they aren't displayed on the page.
- Inspecting the DOM (Document Object Model) lets you see everything the server sends — even stuff hidden from normal view.
- Learning to use Developer Tools is crucial for web security analysis.

Level 2:

1. Challenge:

Visiting the main page doesn't seem to offer anything useful — no visible password or obvious hints.

2. How to Approach:

- View the page source again by right-clicking → View Page Source.
- Look carefully for clues.
- You'll find a link in the code pointing to an "images" folder or a similar directory.

3. Solution:

- Manually open the directory .
- Inside that folder, look for files.
- One of those files will contain the password you need for Level 3.

4. Tools Used:

- Web browser for manually exploring directories.

5. Why This Works (Logic):

- Many web servers accidentally expose entire folders if directory listing isn't disabled.
- By spotting a reference to a folder in the source, you can manually explore it.
- This technique is called Directory Enumeration and is very useful in web security.

Level 3:

1. Challenge:

Again, nothing useful shows up on the main page.

2. How to Approach:

- Think like a search engine.
- Many sites have a robots.txt file to tell search engines what NOT to index.
- Check if there are any "disallowed" directories listed.

3. Solution:

- In robots.txt, you'll find a Disallow entry pointing to a hidden folder.
- Navigate to that folder manually in your browser.
- Inside, you'll find the password needed for Level 4.

4. Tools Used:

- Just a web browser.
- Accessing robots.txt manually.

5. Why This Works (Logic):

- The robots.txt file tells search engines what not to index — but it doesn't hide the files from people who visit directly.
- Often, sensitive areas are listed there by mistake.
- Tip: Always check robots.txt — it might leak hidden directories.

Level 4:

1. Challenge:

When you open the page normally, it says access is denied.

2. How to Approach:

- This time, the problem lies in HTTP headers.
- Specifically, the Referer header is being checked.
- The page expects visitors to come from a specific place (like natas5).

Steps:

- Open Developer Tools → Network tab.
- Or use a tool like cURL.
- Modify the request's Referer header .

3. Solution:

- After changing the Referer header, refresh or resend the request.
- Now the server will think you came from the correct place, and it will show the password.

4. Tools Used:

- Browser's Developer Tools (Network tab).
- Or tools like cURL for manually editing headers.

5. Why This Works (Logic):

- Some websites trust the Referer header too much.

- Since browsers let users fake the Referer easily, it’s not a secure method.
- Moral: Never trust user-controlled data like headers!

Level 5:

1. Challenge:

When you visit the page, it says "Access denied."

2. How to Approach:

- Open Developer Tools → Application tab → Cookies section.
- Look for a cookie named "loggedin".
- Its value will likely be 0.

3. Solution:

- Change the value of the loggedin cookie from 0 to 1.
- Refresh the page.
- Now you should see the password for Level 6.

4. Tools Used:

- Browser’s Developer Tools → Application → Cookies.

5. Why This Works (Logic):

- Some apps trust cookies too much without re-verifying their contents.
- If the cookie simply says "loggedin=0" or "loggedin=1", it’s easy to flip it manually.
- Tip: Always inspect cookies when access is denied!

Level 6:

1. Challenge:

When you open the page normally, you don't find anything obvious.

2. How to Approach:

- View the page source by right-clicking → View Page Source.
- Carefully scan through the HTML for any hidden references — like comments or hidden files.

3. Solution:

- In the page source, you will find a reference to a hidden file (such as a .php or .inc file).
- Access that hidden file manually through the URL.
- Inside, you'll find the password for Level 7.

4. Tools Used:

- Web browser to view page source and access hidden files.

5. Why This Works (Logic):

- Developers sometimes forget that referencing a hidden file inside the source still exposes it to anyone who checks.
- Always check for links or hidden files mentioned in source code.

Level 7:

1. Challenge:

The webpage shows a simple interface where you can switch between different "pages" (e.g., home, about).

2. How to Approach:

- Look at the URL parameters.
- You'll notice that it loads different pages .
- Think about manipulating that input.

3. Solution:

- This allows you to access files outside the intended directory.
- In this case, it fetches the password for Level 8.

4. Tools Used:

- Web browser by manually editing the URL.

5. Why This Works (Logic):

- If the application doesn't properly clean the file path input, users can trick it into reading sensitive files elsewhere on the server.
- ../ tells the server to go up one directory.

Level 8:

1. Challenge:

This time the password is encoded using an XOR operation in the server's PHP script.

2. How to Approach:

- View the source code provided.
- Understand that XOR encryption is being used.
- If you know how XOR works, you know that you can reverse it.

3. Solution:

- Use the XOR property: If A XOR B = C, then C XOR A = B
- Write a simple Python script to decode the encrypted password by applying XOR with the known key.

4. Tools Used:

- Python for writing a script to reverse the XOR.

5. Why This Works (Logic):

- XOR is symmetric.
- If you know the key (or can guess it), you can easily undo the encryption.

Level 9:

1. Challenge:

You’re allowed to input search strings that the server uses in a grep command.

2. How to Approach:

- If the server doesn’t sanitize your input properly, you can inject your own commands.
- Try adding a semicolon ; to end the current command and start a new one.

3. Solution:

- Submit something like: cat /etc/natas_webpass/natas10
- The semicolon breaks out of the grep command and runs your custom command to read the password file.

4. Tools Used:

- Browser’s form submission.
- Basic knowledge of Linux commands.

5. Why This Works (Logic):

- When input isn't sanitized, you can perform Command Injection, tricking the server into running commands you want.

Level 10:

1. Challenge:

Same as Level 9, but now characters like ;, |, and & are blocked.

2. How to Approach:

- Even if semicolons are blocked, there are still ways to separate commands.
- One method is using newline characters (%0A) or special variables like $IFS (Internal Field Separator).

3. Solution:

- Craft a payload like:. /etc/natas_webpass/natas11
- Or use URL encoding to inject commands while bypassing simple character filters.

4. Tools Used:

- URL Encoder tool (online or manually).
- Burp Suite or cURL to send customized requests.

5. Why This Works (Logic):

- Basic filters often miss complex bypass techniques.
- Things like newline characters or $IFS can act as hidden separators in Linux commands.
- Good security must validate input properly, not just block a few characters.

Level 11:

1. Challenge:

After logging in, the application sets a cookie called data.
This cookie looks like a long Base64 string.

2. How to Approach:

- Base64 decode the cookie first.
- Notice that the decoded result is actually XOR encrypted.
- The plaintext hidden inside is a JSON object
- If you know part of the plaintext, you can recover the encryption key by XORing the known plaintext with the ciphertext.

3. Solution:

- Recover the XOR key using the known structure of the JSON object.
- Modify the JSON .
- Re-encrypt it with the recovered key.
- Base64 encode the final result and replace the cookie with your crafted one.
- After refreshing the page with the new cookie, the password for Level 12 will be displayed.

4. Tools Used:

- Python script to automate:
    - Base64 decoding/encoding,
    - XOR operations.
- Browser's Developer Tools → Application → Cookies tab.

5. Why This Works (Logic):

- XOR encryption is easy to break when you know some of the plaintext.
- Poor encryption combined with predictable formats (like JSON) makes this attack possible.
- Always inspect cookies closely — they might hide important data.

Level 12:

1. Challenge:

The site lets users upload images (it checks that the file ends with .jpg).

2. How to Approach:

- Check if the server actually verifies the content of the file — not just the file extension.
- If the validation only looks at the filename (not the file type), you might be able to upload a PHP script disguised as a .jpg.

3. Solution:

- Create a file named shell.php.jpg.
- Inside, write PHP code like:

<?php echo shell_exec($_GET['cmd']); ?>

- Upload the file via the upload form.
- Once uploaded, open the file in your browser.
- Append something like ?cmd=cat /etc/natas_webpass/natas13 to the URL to execute commands and read the next password.

4. Tools Used:

- A text editor to create the payload file.
- Browser to upload and access the uploaded file.

5. Why This Works (Logic):

- If the server only checks the file extension but doesn't validate the file contents or MIME type, attackers can easily upload malicious scripts disguised as images.
- Always enforce strict server-side checks for uploads!

Level 13

1. Challenge:

The server now checks not just the file extension, but also the MIME type (it expects image/jpeg).

2. How to Approach:

- Intercept the upload request using a tool like Burp Suite or Postman.
- Modify the Content-Type header manually to image/jpeg, even though you're sending PHP code.

3. Solution:

- Create a PHP payload file as before.
- Intercept the file upload request.
- Change the header to: image/jpeg
- Upload the file successfully.
- Access the uploaded file and execute commands by passing parameters in the URL, such as:

  cmd=cat /etc/natas_webpass/natas14

4. Tools Used:

- Burp Suite (Proxy feature) or Postman.
- Custom crafted PHP payload.

5. Why This Works (Logic):

- Some servers trust the Content-Type header from the client without verifying the real file contents.
- By manually spoofing the header, attackers can upload executable code even when file type checks are in place.

Level 14:

1. Challenge:

The page has a login form where the input is directly used inside a raw SQL query without proper escaping.

2. How to Approach:

- Classic SQL Injection.
- Inject SQL syntax into the username or password fields.

3. Solution:

- For username, input: OR 1=1 --
- For password, enter anything (it gets ignored).
- This works because:
    - ' ends the string,
    - OR 1=1 always evaluates true,
    - -- comments out the rest of the SQL query.

4. Tools Used:

- Browser to submit the form manually.

5. Why This Works (Logic):

- If user input is directly inserted into a SQL query without proper escaping or prepared statements, attackers can manipulate the database.
- Always sanitize and parameterize database queries to prevent SQL Injection.

Level 15:

1. Challenge:

No error messages or feedback are shown — but based on server behavior, you can guess if your input was correct.

2. How to Approach:

- Use Boolean-based Blind SQL Injection.
- Ask yes/no questions about the database and observe how the page responds.

3. Solution:

- Inject a payload like:natas16" AND password LIKE BINARY "a%" --
- This checks if the password starts with "a".
- Automate the process:
    - Write a Python script that tries all possible characters (a-z, A-Z, 0-9),
    - Character-by-character, build the full password.

4. Tools Used:

- Python script using the requests module.

5. Why This Works (Logic):

- Even without visible errors, the server behavior (response or delay) can leak information.
- By guessing character-by-character and checking if the server behaves differently, you can reconstruct the full password.

Level 16:

1. Challenge:

The server filters some dangerous characters (;, |, &), but command injection is still possible using command substitution.

2. How to Approach:

- Try injecting with $() syntax, which the server might not filter.
- This command checks if the password starts with "a".

3. Solution:

- Use the $(...) command substitution trick.
- Inject a guessing payload to brute-force the password character-by-character.
- Write a Python script:
    - Submit different guesses (^a, ^b, etc.).
    - Analyze if the server behavior changes when a correct guess is made.

4. Tools Used:

- Python scripts using the requests module.

5. Why This Works (Logic):

- Even though basic separators like ; are blocked, $() allows command execution.
- Smart attackers use available gaps (like overlooked syntax) to achieve command execution.

Level 17:

1. Challenge:

Again, no direct output is visible. But here you can exploit time delays.

2. How to Approach:

- Use Time-based Blind SQL Injection.
- Inject queries that cause a deliberate delay (e.g., SLEEP(2)) if a certain condition is true.

3. Solution:

- Inject a payload like:natas18" AND IF(password LIKE BINARY "a%", SLEEP(2), 0) --
- If the condition is true (password starts with "a"), the server response will be delayed.
- Write a script to:
    - Measure response times,
    - Determine correct password character-by-character based on delays.

4. Tools Used:

- Python with requests and time modules.

5. Why This Works (Logic):

- By forcing the server to delay responses based on whether your guess is right, you can slowly reconstruct secret data without any visible error messages.

Level 18:

1. Challenge:

Admin privileges are tied to the session ID (PHPSESSID) — and these IDs are predictable.

2. How to Approach:

- The idea is to brute-force the session IDs.
- Try session IDs like 1, 2, 3... up to 640.

3. Solution:

- Create a Python script that:
    - Changes the PHPSESSID cookie in each request,
    - Checks the server's response to see if admin access is granted.
- Once you find the correct session, you can view the password for Level 19.

4. Tools Used:

- Python with requests.
- Browser’s Developer Tools to monitor cookies.

5. Why This Works (Logic):

- When session IDs are predictable and the pool is small, brute-forcing becomes very easy.
- Secure session management should use random and unpredictable session IDs.

Level 19:

1. Challenge:

The session ID (PHPSESSID) is now Base64-encoded.

2. How to Approach:

- Understand that the session ID encodes user data (like userid:admin).
- Try encoding various userid values to Base64 manually.

3. Solution:

- Guess simple user IDs like 1-admin, 2-admin, 3-admin, etc.
- Encode them in Base64.
- Example:echo -n "1-admin" | base64
- Set your PHPSESSID cookie to the encoded value and refresh the page until admin access is obtained.

4. Tools Used:

- Python or online Base64 encoder/decoder.

5. Why This Works (Logic):

- Base64 encoding is reversible — it’s not encryption.
- If the server doesn’t protect encoded data properly, it’s easy to forge session cookies.

Level 20:

1. Challenge:

The application saves a temporary session file — but it doesn't lock it properly, leading to a race condition.

2. How to Approach:

- Race conditions happen when two operations happen at nearly the same time without synchronization.
- You can:
    - Send two quick requests,
    - One writing new data,
    - One reading the session before the write is complete.

3. Solution:

- Use threading or parallel requests.
- First request sets a custom message (like "I'm admin").
- Second request quickly reads the session before the new message is properly saved.

4. Tools Used:

- Python with requests and threading modules.

5. Why This Works (Logic):

- When files or databases are accessed without proper locks, multiple actions can collide, exposing sensitive information.
- Attackers use race conditions to read or manipulate data before it's safely stored.

Level 21:

1. Challenge:

There are two related subdomains — one called "experimenter" and one the main site. They both share the same session system.

2. How to Approach:

- Access the experimenter subdomain first.
- Log in and check your session ID (PHPSESSID) in the browser cookies.
- Then, reuse the same session ID on the main site.

3. Solution:

- After logging in to experimenter, copy your current session ID.
- Switch to the main site and manually set the PHPSESSID cookie to the one from experimenter.
- The server treats you as already logged in or with elevated permissions — revealing the password.

4. Tools Used:

- Browser Developer Tools → Application → Cookies.
- Or Python's requests library to control cookies manually.

5. Why This Works (Logic):

- When different parts of a site share session storage without proper separation, you can transfer session IDs and reuse privileges between them.
- This vulnerability is known as Session Fixation.

Level 22:

1. Challenge:

The server uses HTTP redirection to hide important information.

2. How to Approach:

- By default, browsers automatically follow redirects.
- We need to disable automatic redirects to view the raw HTTP response before the redirection happens.

3. Solution:

- use Python's requests module with: requests.get(url, allow_redirects=False)
- This way, you catch the hidden content (password) before being redirected.

4. Tools Used:

- cURL tool.
- Python with requests library.

5. Why This Works (Logic):

- Redirects usually hide sensitive data that was sent in the original response.
- By disabling redirects, you can inspect everything the server first sends.

Level 23:

1. Challenge:

The server uses strcmp() in PHP to compare the password.

2. How to Approach:

- In PHP, if strcmp() receives a non-string input like an array, it throws a warning and returns false.
- The application might handle the warning incorrectly and give access.

3. Solution:

- Submit the password field as an array instead of a string.
- The backend comparison breaks, and you bypass authentication.

4. Tools Used:

- Browser (modify the URL manually).
- Or cURL for sending requests.

5. Why This Works (Logic):

- PHP Type Juggling vulnerabilities happen when developers don't strongly check input types.
- Arrays, numbers, booleans, and strings behave differently, causing security issues.

Level 24:

1. Challenge:

The server checks the User-Agent header using preg_match().

2. How to Approach:

- The regex pattern they use is weak and can be bypassed by crafting special payloads.
- Inject weird values into the User-Agent header.

3. Solution:

- Send a User-Agent like:User-Agent: <?php echo shell_exec('cat /etc/natas_webpass/natas25'); ?>
- If the server logs User-Agents (common for analytics) and then includes the log in a page without sanitizing, you get Remote Code Execution (RCE).

4. Tools Used:

- Burp Suite or cURL to craft custom headers.

5. Why This Works (Logic):

- Weak regular expressions can be bypassed.
- If untrusted input like headers is inserted into a file without escaping, attackers can inject code.

Level 25:

1. Challenge:

The server uses include() based on user-controlled input.

2. How to Approach:

- Inject PHP code into server log files (like access logs).
- Then, make the server include that log file by manipulating URL parameters.

3. Solution:

- Set a User-Agent (or any header) to PHP code during a request.
- Then, trigger include("logs/access.log") by passing a crafted parameter
- When the server includes the poisoned log file, your PHP code gets executed.

4. Tools Used:

- Browser (for sending modified headers).
- Burp Suite for crafting precise requests.

5. Why This Works (Logic):

- Combining log injection with file inclusion vulnerabilities leads to Remote Code Execution.
- It's a classic double-vulnerability chain.

Level 26:

1. Challenge:

The server allows uploading data that gets serialized and stored.
Serialized PHP objects can be dangerous if not properly handled.

2. How to Approach:

- If an application unserializes user-controlled data, you can inject malicious PHP objects.
- These objects can execute code during the destruction phase using magic methods like __destruct().

3. Solution:

- Create a custom PHP class with a __destruct() function that executes commands.
- Serialize the object and upload it via the provided upload feature.
- When the server unserializes your object, your __destruct() method runs and leaks the password.

4. Tools Used:

- A PHP script to create the malicious serialized object.
- Base64 encoder if necessary to encode the payload.

5. Why This Works (Logic):

- PHP Object Injection occurs when untrusted serialized data is processed.
- Without proper validation, an attacker can control the behavior of an application.

Level 27:

1. Challenge:

The server obfuscates usernames using XOR before storing or comparing them.

2. How to Approach:

- XOR encryption is reversible.
- If you can guess or know some input, you can craft a username that XORs correctly to match admin.

3. Solution:

- Carefully craft a username that, when XORed with the server's key, matches "admin".
- You may need to brute-force or guess based on server behavior.
- Login with the crafted username to retrieve the password.

4. Tools Used:

- Python script to perform XOR operations and generate correct inputs.

5. Why This Works (Logic):

- XOR is a weak form of obfuscation when the key is short or predictable.
- If the system doesn't securely manage the key, XOR-protected values can be forged.

Level 28:

1. Challenge:

The server encrypts user data by XOR-ing it and Base64 encoding the result.

2. How to Approach:

- If you know part of the plaintext structure (e.g., JSON format), you can perform a Known Plaintext Attack.
- You can XOR the known plaintext against the ciphertext to find the key.

3. Solution:

- Extract parts of the ciphertext by Base64 decoding.
- XOR the decoded data with your guessed plaintext.
- Recover the key.
- Use the key to decrypt the full secret and get the password.

4. Tools Used:

- Python script for:
    - Base64 decoding,
    - XOR operations,
    - Reconstructing the full decrypted text.

5. Why This Works (Logic):

- XOR encryption is symmetric and predictable.
- If part of the encrypted data's format is known, the encryption key can easily be recovered and reused.

Level 29:

1. Challenge:

The server encrypts JSON objects using XOR + Base64.

2. How to Approach:

- Similar to Level 28.
- Decrypt the JSON object, modify it (e.g., change "admin": false to "admin": true), and re-encrypt it.

3. Solution:

- Base64 decode the cookie.
- XOR decrypt it using the key you recover (from known parts of JSON structure).
- Modify the JSON string to make yourself an admin.
- XOR encrypt the modified JSON with the same key.

- Base64 encode and send it back.

4. Tools Used:

- Python script for:
    - Base64 decoding/encoding,
    - XOR encrypting/decrypting,
    - JSON manipulation.

5. Why This Works (Logic):

- XOR is symmetric.
- Modifying decrypted JSON fields and re-encrypting can easily trick a server relying on weak encryption mechanisms.

Level 30:

1. Challenge:

The server uses XOR encryption again, but now it focuses on validating passwords stored in encrypted form.

2. How to Approach:

- The password is encrypted.
- Knowing that the plaintext structure is predictable (like {"password":...}), perform a Known Plaintext Attack.

3. Solution:

- Guess the beginning of the plaintext.
- XOR it against the ciphertext to recover the encryption key.
- Use the key to decrypt the whole password blob.

4. Tools Used:

- Python script for XOR decryption and Base64 decoding.

5. Why This Works (Logic):

- XOR encryption is completely vulnerable if attackers know even a small part of the plaintext.
- Predictable formats like JSON make it even easier to crack the whole secret.

Level 31:

What to do:

- Decrypt cookie slightly.
- Flip specific bits to change "admin":false → "admin":true without fully decrypting.
- Resend the cookie.

Tools used:

- Python (for bit-flipping).

Logic:

- In XOR encryption, flipping a bit in the ciphertext flips it in the decrypted text too!

Level 32:

What to do:

- Server uses extract() function dangerously.
- Pass your own variable in URL like:?file=/etc/natas_webpass/natas33
- Server will include the file you want.

Tools used:

- Browser / cURL.

Logic:

- extract() on user input can let attackers define variables like $file and control the behavior.

Level 33:

What to do:

- The server encrypts input with multiple layers:
    - First, JSON structure (e.g., { "admin": false })
    - Then XOR encryption
    - Then Base64 encoding

- You need to reverse all layers:

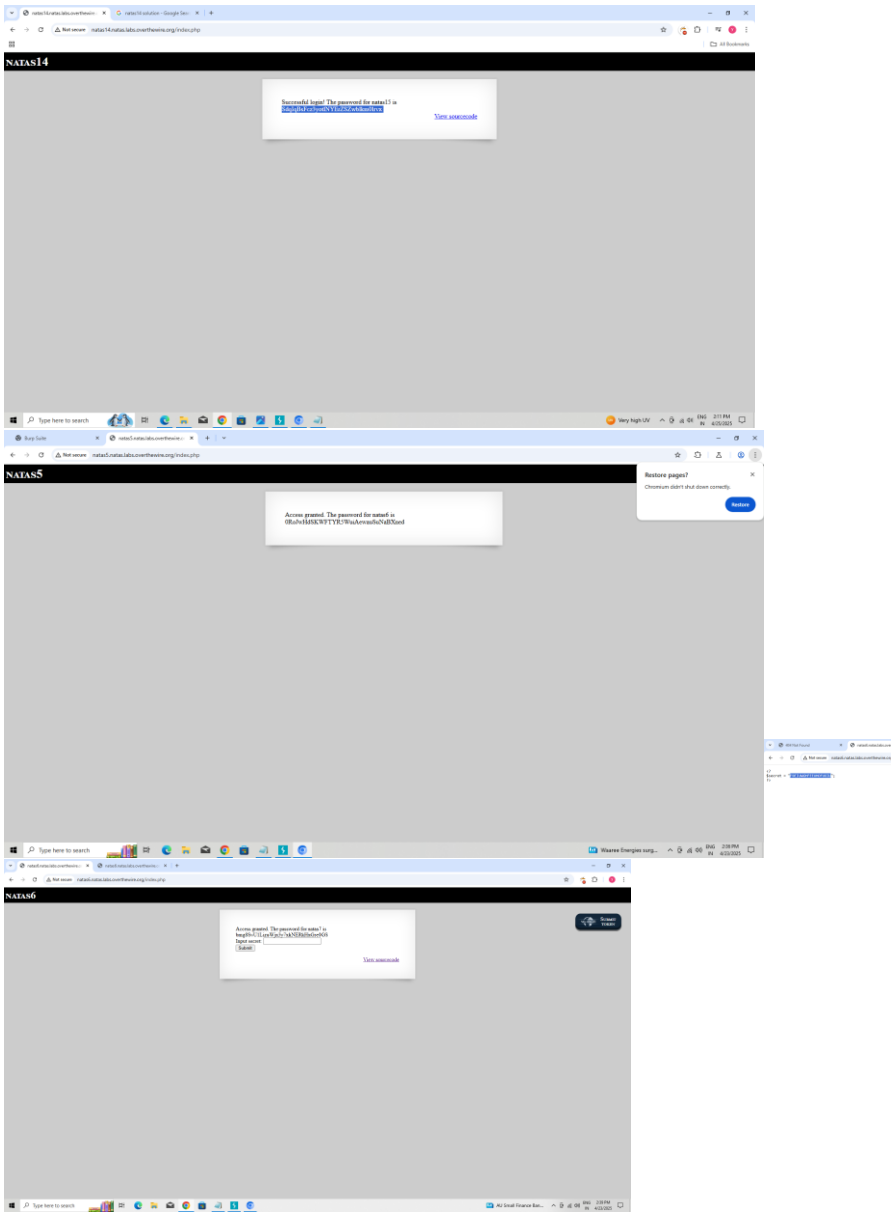
1. Base64 decode the cookie.
2. XOR decrypt using the right key.
3. Change JSON from "admin": false to "admin": true.
4. XOR encrypt again.
5. Base64 encode the new data.
6. Send the updated cookie.

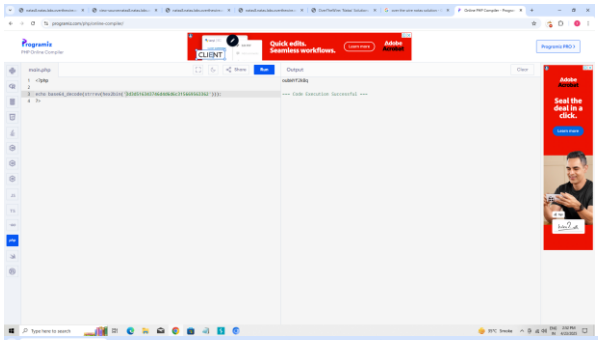
Tools used:

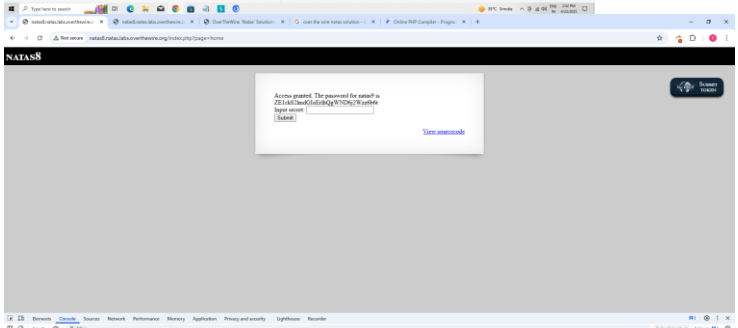
- Python script (handling Base64, XOR, JSON).

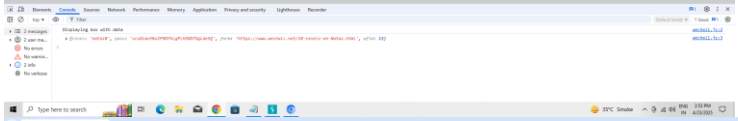
Logic:

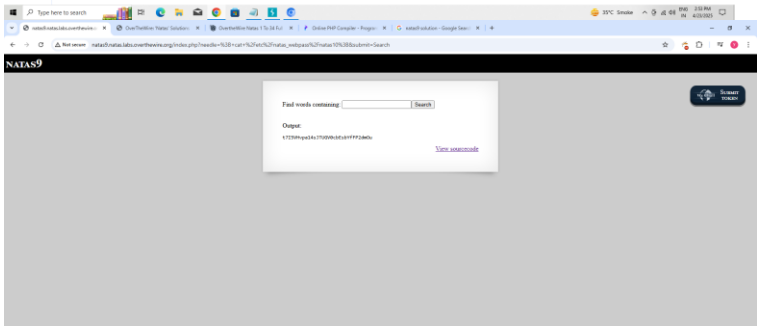
- If you understand how the app encrypts and encodes, you can reverse every step and modify important fields (like admin access!).

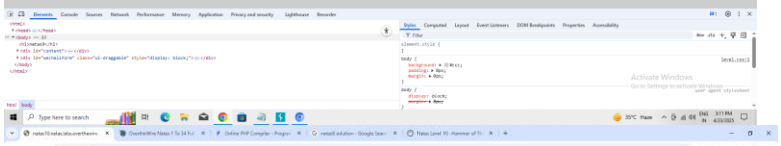


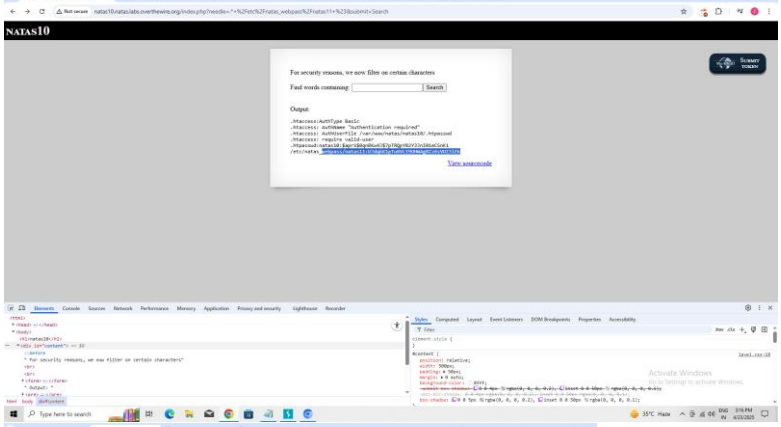


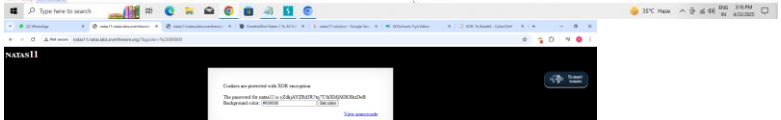

NATAS8
Access granted. The password for natas9 is
Submit
View sourcecode

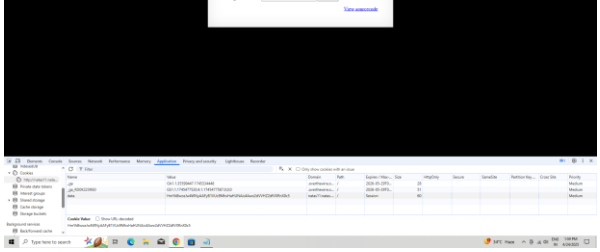


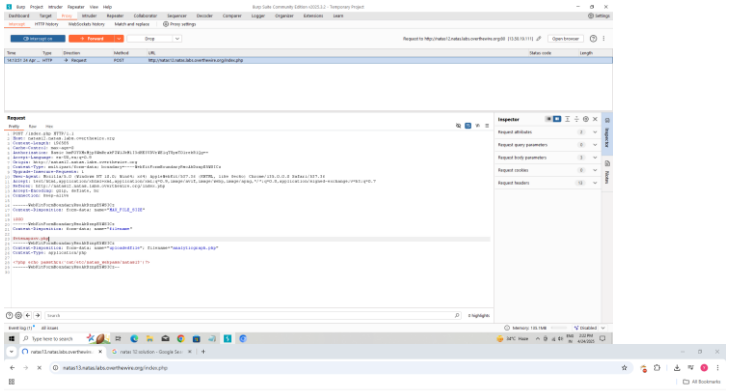

NATAS9
Find words containing:
Search
Output:
View sourcecode

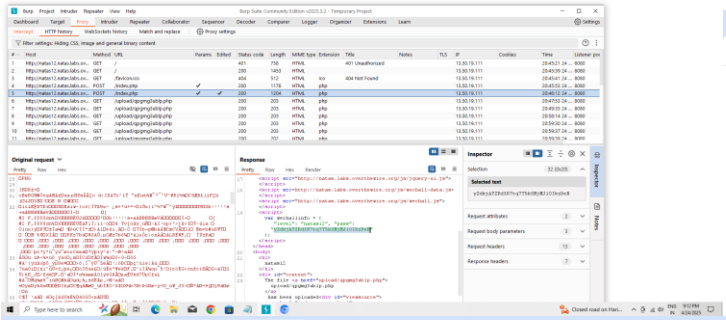


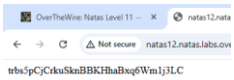

NATAS10
For security reasons, we now filter on certain characters
Find words containing:
Search
Output:
View sourcecode

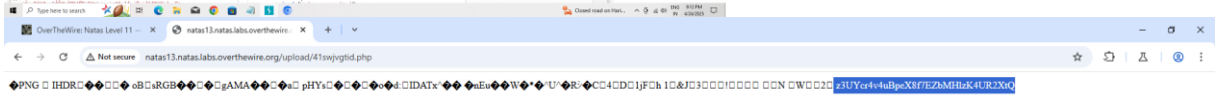


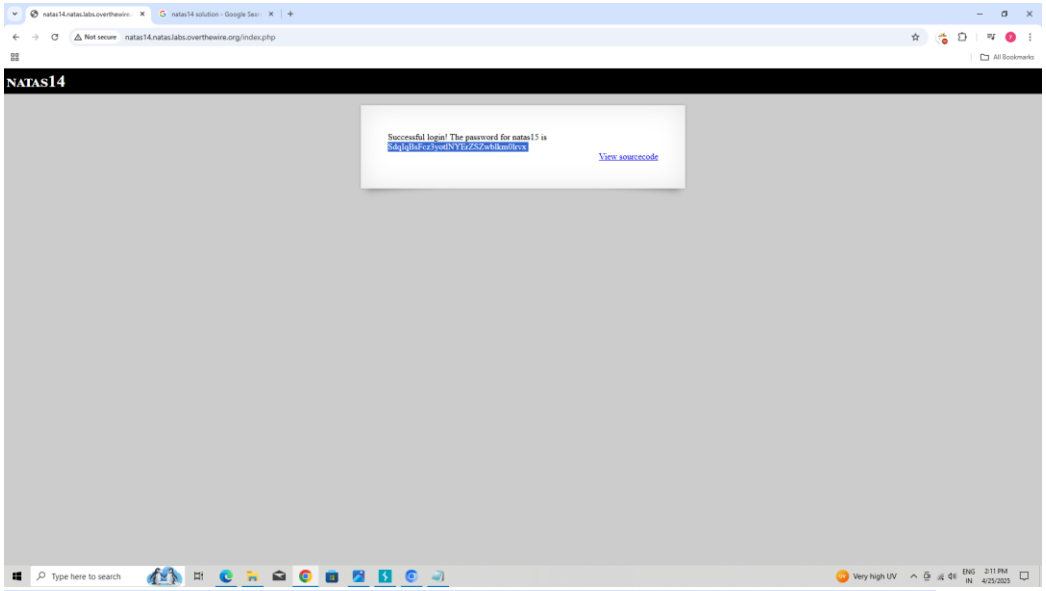

NATAS11
View sourcecode

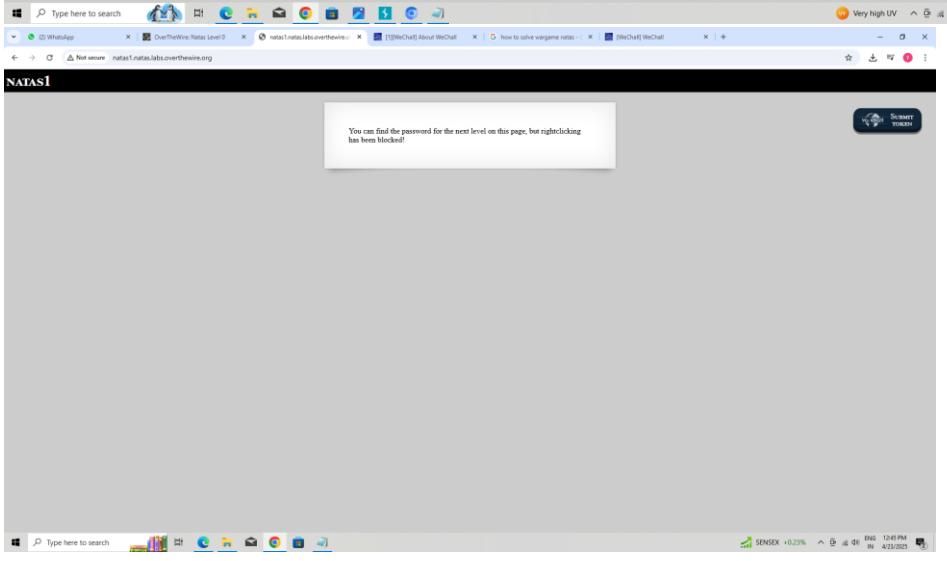


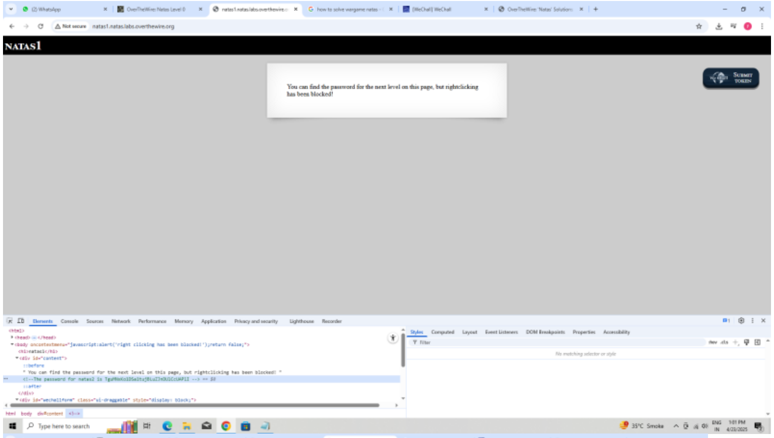


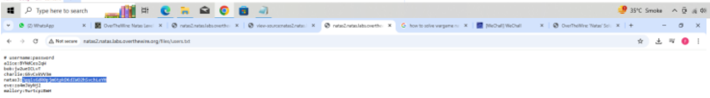

trbs5pCjCrkuSknBBKHhaBxq6Wm1j3LC

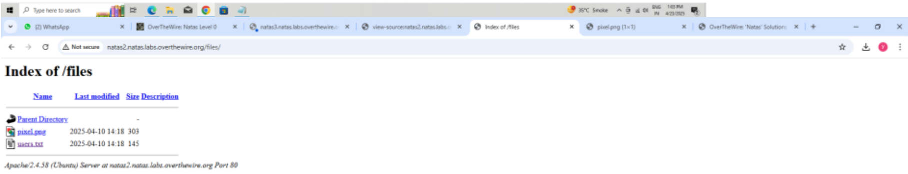

natas13.natas.labs.overthewire.org/upload/41swjvgtid.php
PNG IHDR gAMA pHYs IDATx
z3UYcr4v4uBpeX8f7EZbMHIzK4UR2XtQ

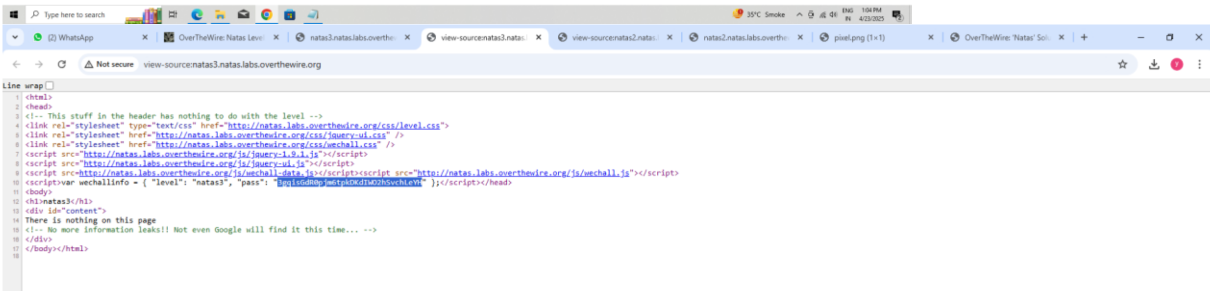

NATAS14
Successful login! The password for natas15 is
SdqIqBsFcz3yotlNYErZSZwblkm0lrvx
View sourcecode

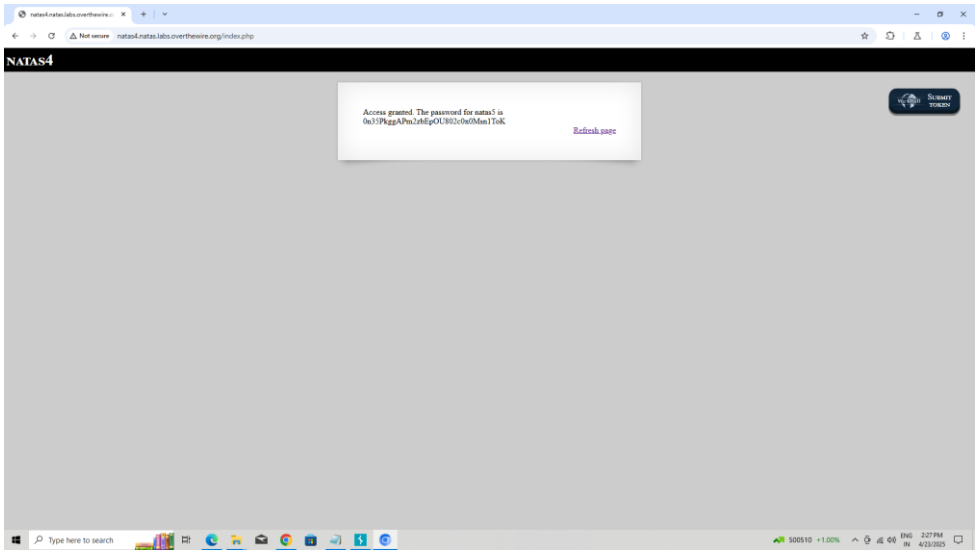

NATAS1
You can find the password for the next level on this page, but rightclicking has been blocked!


NATAS1
You can find the password for the next level on this page, but rightclicking has been blocked!




natas2.natas.labs.overthewire.org/files/
Index of /files
Name Last modified Size Description
Parent Directory -
pixel.png 2025-04-10 14:18 303
users.txt 2025-04-10 14:18 145
Apache/2.4.58 (Ubuntu) Server at natas2.natas.labs.overthewire.org Port 80


Not secure view-source:natas3.natas.labs.overthewire.org
Line wrap
<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>
<body>
<h1>natas3</h1>
<div id="content">
There is nothing on this page
<!-- No more information leaks!! Not even Google will find it this time... -->
</div>
</body></html>


Not secure
natas4.natas.labs.overthewire.org/index.php
NATAS4
Access granted. The password for natas5 is
0n35PkggAPm2zbEpOU802c0x0Msn1ToK
Refresh page
SUBMIT TOKEN
Type here to search
500510 +1.00%
ENG
IN
2:27 PM
4/23/2025