Cybersecurity Wargame Internship Task Report

Team members:

- Omkar Ugale Roll.no.:212

Program: Digisuraksha Parhari

Foundation Internship Issued By: Digisuraksha Parhari Foundation

Supported By: Infinisec Technologies Pvt. Ltd.

OverTheWire – KRYPTON

- Task Description:

The Krypton wargame is designed as a beginner-friendly introduction to the fascinating world of classical cryptography. Each level challenges players to decode ciphered messages, using different techniques and clues along the way. By successfully cracking the codes, players uncover access keys that allow them to advance to the next stage. It's a fun and engaging way to start learning about the art of encryption and decryption, while building a solid foundation in basic cryptographic principles.

- Tools used:

1. Linux Terminal: You'll often access the game remotely, so being comfortable with SSH (Secure Shell) is important.
2. Base64 Decoder: Some levels involve Base64-encoded text, so having a decoding tool handy will save you time.
3. Caesar Cipher Decoder: A classic cipher you'll definitely encounter; online or manual decoders will help you crack these quickly.
4. Online Tools: Platforms like CyberChef and dCode are incredibly useful for all sorts of encoding, decoding, and cryptography tasks.
5. Text Editors and Command-Line Tools: Basic tools like nano, vim, cat, and grep are essential for viewing and searching through files efficiently

- Level-by-level breakdown:

1. Krypton Level 0 → Level 1

- Task Overview:

Your task for Level 0 is to simply connect to the Krypton server using SSH with the credentials provided.

- How It Was Solved:

1. Open your terminal and log in to the system remotely by running the following terminal instruction:

   ssh krypton0@krypton.labs.overthewire.org -p 2231

2. When prompted, enter the access key:
   krypton0 (this is given on the Krypton website).

- Finding the Next Access Key:

Once you're logged in, you'll need to locate the access key for the next level. You can find it by displaying the contents of a file called keyfile.dat. Simply run:

cat /krypton/krypton0/keyfile.dat

Inside, you'll find the access key needed to move on to Level 1!

2. Krypton Level 1 → Level 2

- Task Overview:

In this level, the hidden access key is ciphered using a Caesar cipher — specifically, a ROT13 variation.

- Decrypting Principle:

1. ROT13 is a simple cipher that shifts each letter of the alphabet by 13 places.
   For example:
2. A becomes N, B becomes O, C becomes P, and so on.
3. Applying ROT13 twice will return the original text, making it a very beginner-friendly cipher to crack.

- Command Used:

1. To decode the message, run:

cat /krypton/krypton1/keyfile.dat | tr 'A-Za-z' 'N-ZA-Mn-za-m'
Here's what this does:

- cat prints out the contents of the keyfile.dat.
- tr (short for "translate") shifts the letters according to the ROT13 rule, substituting each character appropriately.

- Background Logic:

1. The tr terminal instruction is a handy tool for performing simple text substitutions.
2. In this case, it takes all uppercase (A-Z) and lowercase (a-z) letters and shifts them 13 positions forward, perfectly solving ROT13 encryption.

Once you run the terminal instruction, you'll get the access key needed to move on to Level 2!

3. Krypton Level 2 → Level 3

- Task Overview:
  1. In this level, you're given a file containing ciphered text.
  2. The hint provided tells you two important things:
     - The text has been shifted, suggesting a Caesar cipher.
     - All the characters are uppercase, which narrows things down a bit.

- Reapproach to solving Strategy:

1. View the Encrypted File:
   Start by displaying the contents of the file with:

   cat /krypton/krypton2/keyfile.dat

2. Brute Force the Caesar Cipher:
   Since you know it's a Caesar cipher but don't know the exact shift, the best approach is to try all possible 26 shifts

until you spot readable English text.

- You can manually rotate the text yourself (a bit tedious), or
- Use an online tool like dCode's Caesar Cipher Solver, which can automatically test all shifts and show the possible plaintexts.

3. Find the Correct Plaintext:

- Scan through the decoded outputs to find the one that makes sense in English.
- Look for a line that clearly contains a access key or instructions.

4. Extract the Access Key:

- Once you find the correct decrypted text, locate and note down the access key — it will be needed to move on to Level 3

4.Krypton Level 3 → Level 4

- Task Overview:

1. In this level, the access key is hidden inside text ciphered with a monoalphabetic substitution cipher.
2. Unlike a Caesar cipher where letters are shifted uniformly, here each letter has been randomly substituted for another, making it a bit trickier.

- Reapproach to solving Strategy:
- Analyze the Cipher Text:
  Start by viewing the ciphered file:

  cat /krypton/krypton3/keyfile.dat

- Perform Frequency Analysis:
  Since it's a monoalphabetic cipher, frequency analysis becomes your best friend. In English, certain letters appear more often (like E, T, A, O, I, N, etc.).
  - Look for the most common letters in the ciphertext and guess their likely real counterparts based on standard English letter frequencies.
  - Gradually map out substitutions and adjust as you recognize more patterns and words.
- Use Helpful Tools:

- CyberChef has a handy "Frequency Analysis" feature that visualizes letter usage, making guessing easier.
- dCode offers substitution solvers that can automate some of the work if you provide some known mappings.
- (Optional) If you're comfortable with scripting, you can even use Python to automate parts of the mapping process.

- End Goal:
  - Keep tweaking the letter mappings until the decrypted text makes sense.
  - Once you spot the readable English access key, you'll be ready to move on to Level 4!

4. Krypton Level 4 → Level 5

- Task Overview:

This level is again a monoalphabetic substitution cipher, but this time the substitution isn't random — it's based on a custom key provided in a script.

- Decrypting Principle:

1. The key used for the cipher is:

THEQUICKBROWNFXJMPSVLAZYDG

2. This custom key defines how the letters are mapped:
   - Letters from the English alphabet are substituted based on the order in this key.
   - Any remaining letters that don't appear in the key are filled in alphabetically afterward.

- Steps to Decode:

1. Create Two Strings:
   - One string representing the normal alphabet (ABCDEFGHIJKLMNOPQRSTUVWXYZ).
   - Another string representing the cipher alphabet based on the provided key.

2. Use Python or Bash:
   - You can use the tr terminal instruction in bash or a simple Python script to substitute the ciphered text back into readable English.
   - Example using tr in the terminal:

   cat /krypton/krypton4/keyfile.dat | tr 'THEQUICKBROWNFXJMPSVLAZYDG' 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

(Adjust the mapping order depending on what needs to be substituted.)

- End Goal:

After substitution, the plaintext will reveal the access key you need to move on to Level 5!

5. Krypton Level 5 → Level 6

- Task Overview:

In this level, the access key is hidden inside a message that has been encoded using Base64.

- How It Was Solved:

1. View and Decode the File:
   You can simply read and decode the file in one step by running:

   cat /krypton/krypton5/keyfile.dat | base64 -d
   - cat displays the contents of the file.
   - The base64 -d terminal instruction decodes the Base64-encoded text back into readable form.
2. Retrieve the Access Key:
   - After decoding, the plaintext access key will be shown directly in your terminal.
   - Make sure to copy it — you'll need it to log into Level 6!

6. Krypton Level 6 → Level 7

- Task Overview:

For the final level, you're tasked with breaking a more complex cipher, such as a Vigenère cipher or possibly a custom encryption logic.

- General Method Followed to Solve:

1. Look for known text patterns that could give clues about the key or the structure of the plaintext.
2. Use online tools like CyberChef's Vigenère Solver to help automate the decryption process.

3. If necessary, guess common keywords (like "KEY", "PASSWORD", etc.) that might have been used as the cipher key.

- What We Learned Throughout Krypton:

- Fundamentals of Classical Ciphers:
  Gained hands-on experience with Caesar ciphers, substitution ciphers, Base64 encoding, and the Vigenère cipher.
- Linux Command-Line Skills:
  Learned how to effectively use tools like tr, cat, base64, and ssh to interact with remote systems and manipulate text data.
- Pattern Recognition and Frequency Analysis:
  Understood the importance of recognizing letter patterns and using frequency analysis to crack substitution-based encryption.
- Analytical and Logical Thinking:
  Practiced approaching problems methodically, testing hypotheses, and solving challenges step-by-step — critical skills for cryptography and cybersecurity work.

- Final Reflection:

The Krypton wargame offers a fantastic introduction to the world of classical cryptography and Linux terminal instruction-line basics. It's designed to be beginner-friendly while still providing plenty of intellectual challenges, making it perfect for anyone just starting their journey into cybersecurity, ethical hacking, or cryptographic problem-solving. By completing it, you've built a strong foundation that will serve you well in more advanced challenges ahead!

OverTheWire-Natas

- Task Description

  The Natas wargame is designed to introduce players to the fundamentals of server-side web security. Through a series of progressively challenging levels, players learn how to uncover access keys hidden in web pages, explore source code, analyze browser cookies, manipulate HTTP requests, and much more. It's a hands-on way to build a strong foundation in web security concepts — perfect for anyone starting out in ethical hacking, penetration testing, or cybersecurity in general.

- Level 0: Introduction to Viewing Page Source

  URL: http://natas0.natas.labs.overthewire.org

  Username: natas

  Access Key: natas

1. Task Overview:
   When you access the page, you'll see the message:
   "You can find the access key for the next level on this page."
   However, the access key is not visible in the rendered content on the web page.

2. Method Followed:
   - To solve this, inspect the HTML structure. In most browsers, you can do this by right-clicking on the page and selecting "View Page Source".
   - Once you open the page source, look through the HTML code for an HTML comment that contains the access key.
3. How It Was Solved:

Inside the HTML source, you'll find a hidden comment that reveals the access key for the next level.

4. Utilities Employed:

Web browser's "View Page Source" feature. This is an essential tool for web security and development, allowing you to inspect the raw HTML and other elements behind the webpage.

5. Logic:

Developers sometimes leave sensitive information, like access keys, in HTML comments. These comments aren't displayed on the page itself but can be seen when viewing the source code. Recognizing this can help you uncover hidden information on a webpage.

- Level 1: Exploring Developer Tools

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

  Username : nata1

  Access Key : (Obtained from Level 0)

1. Task Overview:

   When you visit the page, you'll see the message:
   "You can find the access key for the next level on this page."
   However, there is no visible access key in the rendered content.

2. Method Followed:

   - No access key visible directly on the page, so the next step is to use browser developer tools.
   - Press F12 (or right-click and select "Inspect") to open the browser's developer tools.
   - Navigate to the "Elements" tab where the HTML structure of the page is displayed.
   - Look for HTML comments in the DOM (Document Object Model) that contain the hidden access key.

3. How It Was Solved:

   - By inspecting the HTML source through the developer tools, you will find a

   hidden HTML comment containing the access key for the next level.

4. Utilities Employed:

    - Browser Developer Tools (Elements tab): This tool allows you to inspect the live HTML of the page, including hidden elements and comments that are not rendered visually on the page.

5. Logic:

    - Developers sometimes leave sensitive information in HTML comments, which are hidden from view on the page but still accessible through the source code.
    - Inspecting the DOM is crucial in finding such hidden elements or comments, as they can contain valuable data like access keys.

- Level 2: Directory Enumeration

- URL:
  http://natas2.natas.labs.overthewire.org
- Username:
  natas2
- Access Key:
  (Obtained from Level 1)

1. Method Followed:

- Main Page:
  The main page doesn't provide any useful information, which suggests the approach to solving might be hidden elsewhere.

- View Page Source:
    - View the source code of the page to look for clues. Here, you find a reference to an "images" folder.
- Directory Traversal:
    - Navigate to http://natas2.natas.labs.overthewire.org/files/ to access the folder listing. This may contain files that could help in the next step.
- Locate Access Key:

- Within the folder listing, look for a file that contains the access key for Level 3.

2. Utilities Employed:

- Web Browser: Used for manual folder traversal and inspecting the page source code.

3. Logic:

- Directory Enumeration:
  By analyzing the page source and understanding that web servers might expose hidden directories or files, you were able to access the path a folder listing. This method helps in finding unlinked resources, such as text files, which might contain the access key.

- Level 3: Utilizing robots.txt

- URL:
  http://natas3.natas.labs.overthewire.org
- Username:
  natas3
- Access Key:
  (Obtained from Level 2)

1. Method Followed:

- Main Page:
  The main page offers no useful information, suggesting that the approach to solving might lie elsewhere.
- Access robots.txt:

Visit http://natas3.natas.labs.overthewire.org/robots.txt to view the robots.txt file. This file can contain references to directories that should be excluded from search engine indexing.

- Navigate to Disallowed Directory:

Inside the robots.txt file, look for disallowed directories. These are locations that web crawlers are instructed to avoid but could contain important files.

- Navigate to the disallowed folder to find a file that contains the access key.

2. Utilities Employed:

- Web Browser: Used to access the robots.txt file and access the path the disallowed directories.

3. Logic:

- robots.txt:
  The robots.txt file is a standard used by websites to tell search engines which pages or directories they should not index. By examining this file, you can identify hidden directories that may contain sensitive information, such as access keys, which are not intended for public access.

- Level 4: Referer Header Manipulation

  - URL:
    http://natas4.natas.labs.overthewire.org
  - Username:
    natas4
  - Access Key:
    (Obtained from Level 3)
  - Method Followed:

1. Initial Access:

Visiting the page results in a message denying access.

2. Modify the "Referer" Header:

   - Open browser developer tools (Network tab) or use a tool like cURL.
   - Modify the "Referer" header in the HTTP request to:
     http://natas5.natas.labs.overthewire.org

3. Bypass and Retrieve Access Key:

    - Refresh or resend the request with the modified header to bypass the restriction and view the access key.

4. Utilities Employed:

    - Browser Developer Tools (Network tab)
    - cURL or similar tools to modify HTTP headers manually

5. Logic:

    - Referer Header Control:
      Some web applications check the "Referer" header to determine where a request is coming from. By forging this header, you can trick the application into granting access that would normally be restricted.

- Level 5: Cookie Manipulation

- URL:
  http://natas5.natas.labs.overthewire.org
- Username:
  natas5
- Access Key:
  (Obtained from Level 4)

- Method Followed:

    1. Initial Access:
       The page shows an "access denied" message.

    2. Inspect Cookies:

        - Open the browser's developer tools (Application tab).
        - Review the cookie values set by the server.

3. Modify the Cookie:
    - Find a cookie named loggedin set to 0.
    - Change its value to 1.
4. Bypass and Retrieve Access Key:
    - Refresh the page after modifying the cookie to gain access and view the access key.

- Utilities Employed:

Browser Developer Tools (Application tab) to view and edit cookies manually.

- Logic:

Cookie-Based Authentication:
Some web applications use cookies to track authentication states. If the cookie values are poorly secured, they can be manually modified to impersonate a logged-in user and gain unauthorized access.

- Level 6: Hidden Files in Source Code

- URL:
  http://natas6.natas.labs.overthewire.org
- Username:
  natas6
- Access Key:
  (Obtained from Level 5)

- Method Followed:

1. Initial Access:
   The main page alone doesn't reveal useful information.
2. Inspect the Source Code:
    - View the HTML source of the page.
    - Look for hints — a hidden reference to a file (e.g., a "includes" or a configuration file).
3. Access the Hidden File:
    - Navigate to the mentioned file.

    - Locate the access key or necessary credentials inside it.

- Utilities Employed:

Web Browser (View Page Source feature).

- Logic:

Hidden Files Exposure:
Web developers sometimes leave sensitive files linked in the source code but not visible on the page. Manually reviewing the source helps find these overlooked vulnerabilities.

- Level 7: Directory Traversal

- URL:
  http://natas7.natas.labs.overthewire.org

- Username:
  natas7

- Access Key:
  (Obtained from Level 6)

- Method Followed:

1. Initial Observation:
   The page has URLs with parameters like ?page=home, indicating that it dynamically loads pages based on GET parameters.
2. Exploit Directory Traversal:
    - Try to traverse directories by manipulating the page parameter.

- Attempt to access restricted files outside the intended folder.

3. Access the Access Key File:

Visit:

?page=../../../../etc/natas_webpass/natas8

Retrieve the access key for the next level.

- Utilities Employed:

Browser (manipulating the URL manually).

- Logic:

Path Traversal Vulnerability:
If user input is directly used in file paths without proper sanitization, attackers can manipulate the path (../) to access sensitive files on the server.

- Level 8: XOR-Based Access Key Encoding

- URL:
  http://natas8.natas.labs.overthewire.org
- Username:
  natas8
- Access Key:
  (Obtained from Level 7)

- Method Followed:

1. Initial Observation:
   Viewing the source code reveals that the access key is validated by comparing it to an encoded string generated via an XOR operation.
2. Reverse the XOR:
   - Understand that if A XOR B = C, then B = A XOR C.
   - Use this property to reverse the encoded access key.
3. Recover the Original Access Key:

- Write a small Python script to perform the XOR reversal.
- Extract the original access key used for authentication.

4. Utilities Employed:

Python script (to reverse the XOR encoding).

5. Logic:

XOR Encryption Property:
XOR is a reversible operation. By knowing the encoded result and the encryption method, the original input can be recovered easily.

- Level 9: Command Injection via Input

- URL:
  http://natas9.natas.labs.overthewire.org
- Username:
  natas9
- Access Key:
  (Obtained from Level 8)

- Method Followed:

1. Initial Observation:
   The application uses the input provided by the user in a grep terminal instruction without proper sanitization.
2. Command Injection:
   - Inject a terminal instruction separator like ;.
   - Example input:

; cat /etc/natas_webpass/natas10

3. Retrieve Access Key:

The injected terminal instruction executes alongside grep, displaying the access key for the next level.

- Utilities Employed:
- Browser input form (to submit payload).
- Command Injection Techniques (basic shell terminal instruction knowledge).

- Logic:

Command Injection Vulnerability:
When user input is improperly sanitized and directly passed to system terminal instructions, attackers can inject arbitrary terminal instructions to execute on the server.

- Level 10: Filter Bypass in Command Injection

- URL:
  http://natas10.natas.labs.overthewire.org
- Username:
  natas10
- Access Key:
  (Obtained from Level 9)

- Method Followed:

1. Initial Observation:
   The application passes user input to a shell terminal instruction, but it filters out characters like ;, |, and &.
2. Bypass Strategy:

- Use a newline character (%0A) or the shell's internal field separator ($IFS) to separate terminal instructions instead.
- Example Payload:

. /etc/natas_webpass/natas11

- Submit this carefully crafted payload to bypass the character filters.

- Utilities Employed:

Payload encoding (manual or using tools).

Burp Suite or an online URL Encoder (to craft encoded payloads).

- Logic:

Filter Evasion:
Although direct separators are blocked, alternative encoding (newline, $IFS) or differently structured terminal instructions can successfully bypass input filters and achieve code execution.

- Level 11: Cookie Manipulation & XOR Decryption

- URL:
  http://natas11.natas.labs.overthewire.org
- Username:
  natas11
- Access Key:
  (Obtained from Level 10)

- Method Followed:

1. Initial Observation:
   The application sets a cookie named data, which contains a Base64-encoded, XOR-ciphered JSON object.
   - This JSON includes a showaccess key field.
2. How It Was Solved Steps:
   - Decode the cookie using Base64.
   - XOR the decoded data against a known plaintext such as:

{"showaccess key":"no","bgcolor":"#ffffff"}

to derive the encryption key.

- Modify the JSON to:

{"showaccess key":"yes","bgcolor":"#ffffff"}

- Encrypt the modified JSON with the recovered key.
- Base64 encode the final ciphered string.
- Replace the original data cookie with your new crafted one.

- Utilities Employed:
    - Python script for XOR encryption/decryption.
    - Base64 encoding/decoding tools.
    - Browser Developer Tools (Application tab) for cookie editing.

- Logic:
    - XOR Reversibility:
      If the encryption method and part of the plaintext are known, the key can be extracted.
    - Cookie Tampering:
      By altering the showaccess key flag from "no" to "yes", you can bypass restrictions and reveal the next access key.

- Level 12: File Upload Exploit

- URL:
  http://natas12.natas.labs.overthewire.org
- Username:
  natas12
- Access Key:
  (Obtained from Level 11)

- Method Followed:

1. Initial Observation:
   The web application allows uploading files, supposedly only .jpg images.
2. Source Code Analysis:

Only the file extension is checked, not the actual file content or MIME type.

3. How It Was Solved Steps:

- Create a PHP shell file disguised as a JPEG (e.g., shell.php.jpg).
- Content of the file:

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

- Upload this file using the website's upload form.
- After upload, access the uploaded file directly in the browser.
- Append a cmd parameter to execute terminal instructions, e.g.:

?cmd=cat /etc/natas_webpass/natas13

- Utilities Employed:
- Custom PHP payload creation.
- Browser upload interface to upload the file.
- Browser URL bar to trigger terminal instruction execution.

- Logic:
- Weak File Validation:
  If the server only checks file extensions but does not validate MIME type or file contents, it's possible to upload executable code.
- Code Execution via File Upload:
  When uploaded to a web-accessible folder, the server may interpret the PHP code, allowing arbitrary terminal instruction execution.

- Level 13: File Upload with MIME Check

- URL:
  http://natas13.natas.labs.overthewire.org

- Username:
  natas13

- Access Key:
  (Obtained from Level 12)

- Method Followed:

1. Initial Observation:
   Upload functionality still exists, but now with an added MIME type check (expects an image).

2. Source Code Insight:

Server checks if the uploaded file's Content-Type is something like image/jpeg.

3. How It Was Solved Steps:

- Create a PHP payload (e.g., reverse shell or terminal instruction execution script).
- Use Burp Suite or Postman to intercept the upload request.
- Modify the Content-Type header of the file to image/jpeg, even though the file is PHP code.
- Complete the upload and access the file to execute terminal instructions like:

?cmd=cat /etc/natas_webpass/natas14

- Utilities Employed:
  - Burp Suite (Proxy Intercept tool)
  - Postman (to manually craft and send HTTP requests)
  - Custom PHP payload

- Logic:
  - MIME Type Spoofing:
    Servers often trust the MIME type sent by the client without deeply verifying file contents.
  - Bypassing Basic Upload Checks:
    By setting Content-Type: image/jpeg, a PHP file can bypass naive checks and be executed by the server once uploaded.

- Level 14: SQL Injection – Login Bypass

- URL:
  http://natas14.natas.labs.overthewire.org
- Username:
  natas14
- Access Key:
  (Obtained from Level 13)

- Method Followed:

1. Initial Observation:
   - The page contains a login form with username and access key fields.
   - Source code reveals that the application uses raw SQL queries with user input, which can be vulnerable to SQL injection.
2. How It Was Solved:
   - Use a classic SQL Injection payload to bypass authentication:
   - For username field:
     ' OR 1=1 --
   - For access key field:
     (any value, as it will be ignored by the SQL query)
3. Payload Explained:
   - The ' terminates the SQL query's string for username.
   - The OR 1=1 ensures the SQL query always returns true, effectively bypassing the access key check.
   - The -- is a comment operator in SQL, so the rest of the query is ignored.

- Utilities Employed:
  - Browser (to interact with the login form)
  - SQL injection techniques (to craft and inject the payload)

- Logic:
- SQL Injection:
    - When the application fails to properly sanitize user input, attackers can manipulate the SQL query to always evaluate to true (1=1).
    - This bypasses authentication, allowing access without the correct access key.

- Level 15: Blind SQL Injection (Boolean-Based)

- URL:
  http://natas15.natas.labs.overthewire.org
- Username:
  natas15
- Access Key:
  (Obtained from Level 14)

- Method Followed:
- Initial Observation:
    - The application does not display any visible error messages or responses that indicate a failed login attempt.
    - However, changes in the backend behavior occur when different inputs are provided.
- How It Was Solved:
    - Use a blind SQL injection technique by exploiting a Boolean-based condition:
        - Example input for the username:
          natas16" AND access key LIKE BINARY "a%" –
        - This SQL injection checks if the access key starts with "a". The server will respond differently based on whether the condition is true or false.
- Attempt all possible options Attack:
    - Automate the character-by-character guessing of the access key.
    - For each character, use the LIKE BINARY "a%" condition to progressively guess the access key (e.g., "b%", "c%", "d%", etc.).

- Utilities Employed:

    - Python script with the requests module (to automate and send requests)
    - Boolean-based SQL injection (to infer correct access key characters through response timing or behavior)

- Logic:
    - Blind SQL Injection:
        - Even without a visible response, the backend will behave differently depending on whether the injected condition evaluates to true or false.
        - By iterating over potential characters (a-z) and checking for these subtle differences, you can determine the correct access key.

- Level 16: Blind Command Injection

- URL:

http://natas16.natas.labs.overthewire.org

- Username:
  natas16

- Access Key:
  (Obtained from Level 15)

- Method Followed:

1. Initial Observation:

- The web application filters user input but still allows terminal instruction execution via grep.

- Command execution is indirectly possible through blind terminal instruction injection.

2. How It Was Solved:

- Inject a terminal instruction using $(...) for terminal instruction substitution:

- Example terminal instruction to reveal the access key:
  $(grep ^a /etc/natas_webpass/natas17)

- This terminal instruction will use grep to search for lines in the /etc/natas_webpass/natas17 file that start with "a", effectively displaying the access key for the next level.

3. Brute-Force Attack:

- Use a Python script to automate the character-by-character brute-force attempt. This will test for each character in the access key and extract it by checking the responses.

- Utilities Employed:

  - Python script (for automating the injection and response checking)

  - requests module (to send HTTP requests programmatically)

- Logic:

- Command Injection via $():

  - Even though dangerous characters like ;, &, etc., are filtered, the $() syntax is allowed for terminal instruction substitution, enabling terminal instruction execution through the shell.

- Level 17: Blind Time-Based SQL Injection
- URL:
  http://natas17.natas.labs.overthewire.org
- Username:
  natas17
- Access Key:
  (Obtained from Level 16)

- Method Followed:

1. Initial Observation:

- No visible response changes are shown when input is submitted.
- This is a Blind SQL Injection challenge, where the response time can be used to infer the result of a query.

2. How It Was Solved:

- Use the SLEEP() function in SQL to introduce a delay if the access key matches.
- The payload checks if the first character of the access key matches "a", and if true, it causes the query to sleep for 2 seconds:
- Example payload:
  natas18" AND IF(access key LIKE BINARY "a%", SLEEP(2), 0) --
- A delay in response indicates that the condition (access key starts with "a") is true, and this helps in identifying the correct access key character by character.

3. Automating the Process:

- Use Python to send requests with payloads and measure the response time, waiting for delays to detect each correct character in the access key.
- The script should test each character and adjust the query to test different possible access key characters.

- Utilities Employed:
  - Python (for automating the SQL injection and time-based responses)
  - requests module (to send HTTP requests programmatically)
  - time module (to measure response times for identifying delays)

- Logic:
- Blind Time-Based SQL Injection:
    - Since the application gives no visible output, time delays introduced by the SLEEP() function are used to infer the correct access key.
    - A significant delay (e.g., 2 seconds) confirms that the guessed character is correct, enabling brute-forcing of the access key.

- Level 18: Session ID Enumeration

- URL:
  http://natas18.natas.labs.overthewire.org
- Username:
  natas18
- Access Key:
  (Obtained from Level 17)

- Method Followed:

1. Initial Observation:

- Admin access is controlled by a session ID. The goal is to enumerate possible session IDs to gain admin privileges.
- The system likely uses predictable session IDs.

2. How It Was Solved:

- Attempt all possible options Session IDs:
    - Try session IDs from 1 to 640 (or 1024 depending on setup) by modifying the PHPSESSID cookie value.
- For each session ID, set the PHPSESSID cookie and check if admin access is granted.
- When the correct session ID is found, it will grant admin privileges, allowing access to the next access key.

3. Utilities Employed:

- Python with the requests module to automate the brute-forcing process by manipulating session IDs.

- Optionally, Burp Suite can be used to monitor and modify cookies.

4. Logic:

- Session ID Enumeration:
    - If the session ID is predictable or has a small enough range (1 to 640), it's possible to brute-force the valid session ID.
    - This exploit works because the system doesn't protect against predictable session IDs, allowing attackers to gain unauthorized access.

- Level 19: Encrypted Session ID

- URL:
  http://natas19.natas.labs.overthewire.org
- Username:
  natas19
- Access Key:
  (Obtained from Level 18)

- Method Followed:
- Initial Observation:
- The PHPSESSID is Base64 encoded.
- This encoding represents a session ID where predictable data (like user:admin) is encoded.

- How It Was Solved:
    - Attempt all possible options Base64 Encodings:
        - Attempt all possible options possible Base64 encodings of numbers, e.g., starting from 1 (encoded as MTox for 1:admin).
        - Check which Base64 encoding results in the session ID that grants admin=1.
- Utilities Employed:
    - Base64 encoder/decoder for encoding and decoding session data.
    - Python script to automate the brute-forcing process by encoding the possible combinations and testing them.

- Logic:
- Base64 Encoding:
    - Base64 encoding is reversible. If the system encodes predictable data (like userid:admin) and the encoding scheme is known, it can be decoded and manipulated.
    - By brute-forcing Base64 encoded session IDs, you can eventually find one that corresponds to an admin session.

- Level 20: Race Condition in File Lock

- URL:
  http://natas20.natas.labs.overthewire.org
- Username:
  natas20
- Access Key:
  (Obtained from Level 19)

- Method Followed:

1. Initial Observation:

- The application saves a message in a temporary session file and reads it back. However, it does not properly lock the file during the operation, leading to a race condition.

2. How It Was Solved:

- Exploit the Race Condition:
    - Send two rapid requests:
        1. The first request sets the message.
        2. The second request reads the message.
- By sending these requests in quick succession, we exploit the race condition, which can lead to reading another user's session (in this case, the admin's session).
- Utilities Employed:
- Python with the requests and threading modules to send simultaneous requests and exploit the race condition.

- Logic:

- Race Condition:
    - A race condition occurs when the sequence of operations is not properly synchronized, allowing one operation (reading the session file) to occur before the other (writing the session data), thereby leaking data.
    - In this case, we can read the session data before it's fully written, potentially leaking the admin's session or other critical data.

- Level 21: Session File Disclosure

- URL:
  http://natas21.natas.labs.overthewire.org
- Username:
  natas21
- Access Key:
  (Obtained from Level 20)

- Method Followed:

1. Initial Observation:

- Two subdomains (e.g., experimenter.natas21... and the main domain) share the same session mechanism. The admin uses one subdomain, while you access the other.

2. How It Was Solved:

- Reuse Session ID:
    - Find the session ID from one subdomain (for example, from experimenter.natas21...).
    - Use that session ID in the main domain to gain elevated access, potentially as the admin.

3. Utilities Employed:

- Browser cookies and Developer Tools to inspect and extract session IDs.
- Python requests module to automate the session hijacking and reuse.

4. Logic:

- Session Fixation:
    - The two subdomains share the same session ID, so if you can capture the session ID from one subdomain, you can use it on the other subdomain to gain the same level of access, such as admin privileges.

Level 22: Log Injection + LFI

- URL:
  http://natas22.natas.labs.overthewire.org
- Username:
  natas22
- Access Key:
  (Obtained from Level 21)

- Method Followed:

1. Initial Observation:
    - The application uses redirection to block access to the content you need, likely hiding the access key in the process.
2. How It Was Solved:
    - Disable Redirection:
        - Use Python or cURL to prevent automatic redirection. This allows you to read the content from the first response, where the access key might be hidden.
        - In cURL, use -L disabled: curl -L (disable automatic redirection).
        - In Python, set allow_redirects=False using the requests module to manually control the redirection behavior.
3. Utilities Employed:
    - cURL with -L disabled to stop automatic redirection.
    - Python requests with allow_redirects=False to control HTTP redirection.

- Logic:
- Redirection Control:
    - Redirection typically hides the desired content. By disabling redirection, you can ensure that the first response is returned, potentially revealing the hidden information (access key).

Level 22: Log Injection + LFI

- URL:
  http://natas22.natas.labs.overthewire.org
- Username:
  natas22

- Access Key:
  (Obtained from Level 21)

- Method Followed:

1. Initial Observation:
    - The application uses redirection to block access to the content you need, likely hiding the access key in the process.
2. How It Was Solved:
    - Disable Redirection:
        - Use Python or cURL to prevent automatic redirection. This allows you to read the content from the first response, where the access key might be hidden.
        - In cURL, use -L disabled: curl -L (disable automatic redirection).
        - In Python, set allow_redirects=False using the requests module to manually control the redirection behavior.
3. Utilities Employed:
    - cURL with -L disabled to stop automatic redirection.
    - Python requests with allow_redirects=False to control HTTP redirection.

- Logic:

- Redirection Control:
    - Redirection typically hides the desired content. By disabling redirection, you can ensure that the first response is returned, potentially revealing the hidden information (access key).

- Level 23: Eval() Code Injection

- URL:
  http://natas23.natas.labs.overthewire.org
- Username:
  natas23
- Access Key:
  (Obtained from Level 22)

- Method Followed:

1. Initial Observation:

- The application compares the access key using strcmp(). PHP's type juggling can be exploited to bypass the comparison.

2. How It Was Solved:

- Abuse PHP Type Juggling:
    - By sending the access key as an array, like ?passwd[]=, the strcmp() function, which expects a string, will fail. This results in bypassing the access key comparison.

3. Utilities Employed:

- Browser (to manually test the input)
- cURL (for testing requests)

- Python (for scripting or automating requests)

4. Logic:

- PHP Type Juggling:
    - In PHP, if strcmp() is given a non-string input (like an array), it throws a warning and skips over the logic. This allows you to bypass the validation check for the access key

- Level 24: Backdoor via User-Agent
- URL:
  http://natas24.natas.labs.overthewire.org
- Username:
  natas24
- Access Key:
  (Obtained from Level 23)

- Method Followed:

1. Initial Observation:

- The application uses preg_match() to filter inputs, specifically the User-Agent header.

2. How It Was Solved:

- Bypass the Regex:
    - Inject a malformed header or a PHP-like payload in the User-Agent to exploit the regex matching logic and bypass the server's checks.

3. Utilities Employed:

- cURL (for sending custom headers)
- Python (with requests library for adding custom headers)

- Logic:
- Unsafe Use of preg_match():
    - The app uses regex matching on headers like User-Agent, which can be bypassed if the regex isn't strictly written or sanitized. This creates a backdoor allowing code injection.

- Level 25: Loose Type Comparison

- URL:
  http://natas25.natas.labs.overthewire.org
- Username:
  natas25
- Access Key:
  (Obtained from Level 24)

- Method Followed:

1. Initial Observation:
    - The application includes a file based on user input (e.g., your name).
2. How It Was Solved:
    - Exploit Loose Type Comparison:
        - Inject a custom PHP file (for example, via the logs) and access it through the vulnerable include() statement.
        - This allows for remote code execution by including arbitrary PHP code.
3. Utilities Employed:
    - cURL (for forged headers and log injection)

4. Logic:
    - Injection + File Inclusion:
        - By exploiting the combination of a loose type comparison and file inclusion, an attacker

can inject malicious PHP code into log files or other accessible locations and have it executed remotely when the app includes it.

- Level 26: Hidden Backdoor File

- URL:
  http://natas26.natas.labs.overthewire.org
- Username:
  natas26
- Access Key:
  (Obtained from Level 25)

- Method Followed:

1. Initial Observation:

- The web app allows uploading files, and it saves serialized PHP objects to disk.

2. How It Was Solved:

- Exploit PHP Object Serialization:
  - Create a custom PHP class with a __destruct() method. This method can execute arbitrary code when the object is destroyed during the deserialization process.
  - By uploading the crafted object and triggering the deserialization, you can achieve remote code execution (RCE).

3. Utilities Employed:

- Custom PHP Class:
  - For creating a class with a destructive method.
- Python (for Base64 Encoding):
  - To serialize and Base64 encode the crafted PHP object.

- Logic:
- Deserialization of Untrusted Input:
  - When an application deserializes objects without proper validation, it can lead to arbitrary code execution if a malicious payload is included in the object.

- Level 27: XOR Obfuscation

- URL:
  http://natas27.natas.labs.overthewire.org
- Username:
  natas27
- Access Key:
  (Obtained from Level 26)

- Method Followed:

1. Initial Observation:

- Usernames are obfuscated using XOR encryption before being stored or compared.

2. How It Was Solved:

- Forge a Username:
    - Carefully craft a username that, when XORed using the application's key, will match the target username (admin).
    - This usually involves controlling the input to collide with admin after obfuscation.

3. Utilities Employed:

- Python XOR Script:
    - To automate XOR operations and generate a suitable payload.

- Logic:
- XOR Principles:
    - If A XOR B = C, then A = B XOR C and B = A XOR C.
    - By manipulating the input and knowing how XOR behaves, you can reverse-engineer or forge a valid obfuscated username that maps to admin or any privileged account.

- Level 28: XOR with Known Key

- URL:
  http://natas28.natas.labs.overthewire.org
- Username:
  natas28

- Access Key:
  (From Level 27 output)

- Method Followed:

1. Initial Observation:
    - Application XORs the data with a key and then Base64 encodes it.
    - We are given enough information (like some known plaintext) to perform an attack.
2. How It Was Solved:

- Known Plaintext Attack:
    - If you know part of the original unciphered data, you can XOR the ciphered data against the plaintext to reveal the key.
- Decrypt the Rest:
    - Use the key to XOR-decrypt the full encoded message.

3. Utilities Employed:

- Python Script:
    - For automating XOR and Base64 decoding operations.

- Logic:
    - XOR Decryption Principle:
    - If ciphertext = plaintext XOR key,
      then key = ciphertext XOR plaintext.
- Once you get the key, you can decrypt or re-encrypt anything the app expects.

- Level 29: JSON-Based Encryption

- URL:
  http://natas29.natas.labs.overthewire.org
- Username:
  natas29
- Access Key:

(From Level 28 output)

- Method Followed:

1. Initial Observation:
    - A JSON object is being XOR-ciphered and then base64-encoded.
    - Fields like "admin": false can potentially be flipped.
2. How It Was Solved:
    - Decrypt the cookie by Base64 decoding and XOR-ing.
    - Modify the JSON (change "admin": false → "admin": true).
    - Encrypt it back (XOR with the same key).
    - Base64 encode the result and set it as the new value.
3. Utilities Employed:

- Python Script:
    - Handle JSON, XOR operations, and Base64 transformations.

4. Logic:

- XOR Encryption:
    - XOR is symmetric: encrypting and decrypting use the same operation.
    - By flipping values in the plaintext and reapplying the encryption, you can manipulate server behavior.

- Level 30: Known Plaintext Attack

- URL:
- Username:
  natas30
- Access Key:
  (from Level 29 output)

- Method Followed

1. Observation:

- The application uses XOR-ciphered data to validate access keys.
- A known plaintext attack is possible because part of the plaintext is predictable (for example, JSON structure like {"access key": "..."} or padding patterns).

2. How It Was Solved:

- Guess the known plaintext structure.
- XOR the known plaintext with the corresponding ciphertext portion to derive the encryption key.
- Use the recovered key to decrypt the entire ciphered message and recover the access key.

- Utilities Employed:
- Python XOR Decryption Script:
    - Base64 decoding
    - XOR operations
    - Key recovery and decryption

- Logic:
    - XOR encryption is symmetric and vulnerable to known-plaintext attacks.
    - Once the key is derived, you can decrypt the full ciphertext.
    - Thus, predictable structures (like JSON fields) can expose the entire secret.

- Level 31: Server-Side Verification

- URL:
- Username:
  natas31
- Access Key:
  (from Level 30 output)

- Method Followed

1. Observation:

- The server decrypts user-controlled input (XOR + Base64 encoding) and validates fields like "admin": false.
- The server trusts the decrypted data — no independent verification!
- Minimal bit changes can flip important logic (example: "false" → "true").

2. How It Was Solved:

- Reuse the ciphered blob (Base64 + XOR).
- Flip bits precisely at the right spot to change "false" into "true" inside the ciphered payload.
- Re-encode and send it back — gaining admin privileges without needing the full key.

3. Utilities Employed:

- Python Script:
    - Base64 decode
    - Bit-flipping
    - Base64 re-encode

- Logic:
    - Bit-flipping attacks allow manipulating ciphertext directly.
    - In XOR-based encryption, flipping a bit in ciphertext flips the corresponding bit in plaintext.
    - Control the data without needing to decrypt the whole payload or recover the key

- Level 32: PHP Variable Injection

- URL:
- Username:
  natas32
- Access Key:
  (from Level 31 output)

- Method Followed

1. Observation:

- The server uses PHP's extract() function on user-controlled input ($_GET / $_POST).
- extract() turns array keys into variables.
- If the server code expects a sensitive variable (like $file or $admin), you can inject and control it.

2. How It Was Solved:

- Send a crafted request that overrides important server variables.
- For example:

- This tricks the server into including or reading unintended files — leaking sensitive data like the access key for the next level.

- Utilities Employed:
    - Browser or cURL to send crafted requests
    - Basic knowledge of PHP internals and common variable names

- Logic:
    - Using extract() on untrusted input is very dangerous.
    - It gives users the ability to define server variables, altering program logic or causing file inclusions, bypasses, and leaks.

- Level 33: Multi-Stage Encryption

- URL:
- Username:
  natas33
- Access Key:
  (from Level 32 output)

- Method Followed

1. Observation:

- The server encrypts user input using multiple layers:
    - Base64 encoding
    - XOR encryption
    - JSON structure
- The application expects trusted decoded data to determine user privileges.

2. How It Was Solved:

- Reverse the entire process step-by-step:

    1. Base64 decode the cookie or input.
    2. XOR decrypt the result using a derived key.
    3. Parse JSON to understand and modify fields (e.g., change "admin": false → "admin": true).
    4. Rebuild: JSON → XOR encrypt → Base64 encode.
- Resend the modified, re-ciphered payload to gain admin access.

- Utilities Employed:
- Python script to automate:
    - Base64 decode/encode
    - XOR decrypt/encrypt
    - JSON parsing/manipulation

- Logic:
    - By mimicking the server's decryption logic, you can manipulate the payload at the raw data level.
    - Multi-layer encoding isn't effective if each layer is individually reversible.

- Level 34: Final Task Overview – Obfuscation, Injection, Encoding

- URL:
- Username:

natas34

- Access Key:
  (from Level 33 output)

- Method Followed

1. Observation:

- Final Boss Level: a mix of everything learned:
  - Input obfuscation
  - XOR encoding
  - Base64 layers
  - Loose server validation
  - Potential for injection (code/logic/file)

2. How It Was Solved:

- Download or study the source code carefully.
- Map the data flow:
  - How user input is handled, encoded, decoded, verified.
- Reverse the encoding/encryption just like in previous levels.
- Craft a payload that:
  - Modifies key variables or behaviors (e.g., admin: true)
  - Bypasses checks hidden under layers of encoding.
- Use Python for the decoding/rebuilding work.

- Use Burp Suite (or manual crafting) to precisely control your final request.

- Utilities Employed:
  - Python scripting (base64 + XOR reversing)
  - Burp Suite (to intercept/modify HTTP requests)
  - Manual source code review (searching for critical lines: eval, exec, include, extract, etc.)

- Logic:

- All Natas concepts combined:
    - Weak crypto + unsafe deserialization + logic flaws + input-based behavior.
- Persistence and methodical reversing are key.
- Think in multiple layers, like peeling an onion — each encoding hides another simple bug.

- Final Reflection:

  The OverTheWire Natas challenges provide a comprehensive learning experience in web security, focusing on common vulnerabilities such as SQL injection, file upload flaws, terminal instruction injection, and session management weaknesses. Each level walks players through the process of identifying and exploiting these security flaws using various tools like Burp Suite, Python scripting, and browser developer tools. By advancing through the levels, participants learn key concepts in web application security, including input validation, encoding, session hijacking, and privilege escalation. Natas offers a hands-on approach to understanding web security, making it an invaluable resource for anyone interested in ethical hacking and penetration testing, enhancing their skills in finding and exploiting web-based vulnerabilities

OverTheWire-Leviathan

- Task Description

OverTheWire Leviathan is a beginner-to-intermediate wargame focused on basic Linux and binary exploitation techniques. It teaches players how to explore file permissions, discover hidden files, analyze simple binaries, and exploit vulnerabilities like hardcoded access keys, symbolic link abuse, and basic buffer overflows. Each level typically involves finding or exploiting minor security oversights to gain access to the next user account, helping players build practical skills in privilege escalation, file system navigation, and understanding insecure coding practices in a hands-on, real-world style environment.

- Level 0 → 1

- Task Description:
  Locate the access key for leviathan1.
- Utilities Employed:
  ls, cd, cat, grep
- How It Was Solved Logic:

    1. Access the .backup folder:

```
cd .backup
```

    2. Identify the bookmarks.html file.
    3. Search for the keyword "access key" inside the file:

```
grep "access key" bookmarks.html
```

    4. Extract the access key from the matching line.

- Access Key:
  rioGegei8

- Level 1 → 2
- Task Description:
  Discover the access key for leviathan2.
- Utilities Employed:
  file, ltrace
- How It Was Solved Logic:

    1. Identify the check binary using ls.
    2. Confirm it's an runnable binary using:

```
file check
```

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

```
ltrace ./check
```

    4. Observe the correct access key from the strcmp function output.
    5. Use the discovered access key to successfully execute the binary.

- Access Key:
  ougahZi8a

- Level 2 → 3
- Task Description:
  Gain access to leviathan3.
- Utilities Employed:
  ltrace, touch, mkdir, bash
- How It Was Solved 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 folder name containing a terminal instruction injection, such as test; bash.
    5. Run the binary with the crafted name to trigger a shell.

- Access Key:
  Ahdiemo1j

- Level 3 → 4
- Task Description:
  Retrieve the access key for leviathan4.
- Utilities Employed:
  ltrace
- How It Was Solved Logic:

    1. Locate the level3 binary using ls.
    2. Use ltrace to observe the function calls and spot the expected access key:

ltrace ./level3

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

- Access Key:
  vuH0cox6m

- Level 4 → 5
- Task Description:
  Find the access key for leviathan5.
- Utilities Employed:
  ls, cd, ./bin, binary-to-ASCII conversion

- How It Was Solved Logic:

    1. Navigate into the .trash folder 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 access key.

- Access Key:
Tith4okei

- Level 5 → 6

- Task Description:
Access leviathan6.

- Utilities Employed:
ln, symbolic links

- How It Was Solved Logic:

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

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

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

- Access Key:
Ugaoee4li

- Level 6 → 7

- Task Description:
Obtain the access key for leviathan7.

- Utilities Employed:
Python scripting, brute-force approach

- How It Was Solved 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.

- Access Key:
ahyMaeBo9

- Level 7 → 8

- Task Description:
  Complete the final level.
- Utilities Employed:
  strings, grep
- How It Was Solved Logic:
  1. Use strings to extract readable strings from the leviathan7 binary:
     strings leviathan7
  2. Look for hardcoded access keys or hints in the output.
  3. Use the discovered information to gain access.
- Access Key:
  loVzZ6mT
- Final Reflection:

The OverTheWire Leviathan challenges offer a hands-on approach to learning cybersecurity by guiding players through a series of progressively difficult tasks that involve binary analysis, file manipulation, encoding/decoding, and exploiting common vulnerabilities like terminal instruction injection and brute-forcing. Each level teaches practical skills such as using tools like ltrace, grep, and Python scripting, while also showcasing various attack techniques and how they can be exploited. By completing the levels, participants gain valuable experience in ethical hacking, penetration testing, and understanding real-world security flaws, making Leviathan an excellent educational resource for aspiring cybersecurity professionals.