Cybersecurity War-games Report

DIGISURAKSHA PARHARI FOUNDATION

Team Members:

- **Prasanna Dolas -166 [Krypton]**
- **Rushikesh Aherkar - 152 [LEVIATHAN]**
- **NATAS HAS BEEN DONE BY BOTH**

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 Termina**l: 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

Lab 1: KRYPTON

(https://overthewire.org/wargames/krypton/)

Objective:

The objective of the KRYPTON wargame is to reinforce basic skills in cryptography and elementary data encoding/decoding methods. We have to complete a series of increasingly difficult levels that entail:

- Identifying and decrypting different classic cipher algorithms (e.g., Caesar cipher, ROT13, base64 encoding).
- Rehearsing manual decryption methods and learning about encryption fundamentals.
- Developing logical reasoning and problem-solving abilities with cryptographic puzzles.
- Using Linux command-line tools proficiently to decode and extract concealed information.
- The KRYPTON laboratory assists interns in developing a solid grounding in traditional cryptography, which is necessary for contemporary cybersecurity.

Krypton Level 0->1:

Password: KRYPTONISGREAT

Objective: Decrypt a message that is encoded using ROT13 (a basic letter substitution cipher).

Tools/Commands Used: cat, tr

Step-by-Step Solution:

- Used cat krypton0 to look at the encrypted message.
- Identified that the encryption employed ROT13 based on hint.
- Employed the tr command to decode: cat krypton0 | tr 'A-Za-z' 'N-ZA-Mn-za-m'
- Got the password for the next level.

Logic: ROT13 moves each letter 13 places in the alphabet. Twice applying ROT13 returns the original text.

Krypton Level 1->2:

Password: ROTTEN

Objective: Decrypt a message encrypted with a custom substitution cipher.

Tools/Commands Used: cat, manual substitution, Linux text editor

Step-by-Step Solution:

- Watched the encrypted message using cat krypton1.
- Realized that a substitution cipher based on a custom key was utilized.

- Applied the given substitution mapping (provided in hint or file) to substitute manually.
- Unencrypted the text to obtain the password.

Logic: Every letter is substituted with a particular corresponding letter from a cipher alphabet not a rotation like ROT13.

Krypton Level 2->3:

Password: CAESARISEASY

Objective: Decipher encrypted text and decrypt it through manual mapping.

Tools/Commands Used: cat, grep, simple substitution, cmd

Step-by-Step Solution:

- Opened file with encrypted password.
- Examined patterns within the encrypted text (e.g., letter frequencies of E, T).
- Made logical substitutions based on typical English letter frequency.
- Mapped characters manually and worked out the password.

Logic: English text follows anticipated letter frequencies (e.g., 'E' most frequent). Patterns are used to facilitate manual decryption.

Krypton Level 3->4:

Password: ARUTE

Objective: Decipher another substitution cipher but under a more sophisticated situation.

Tools/Commands Used: cat, tr, grep

Step-by-Step Solution:

- Wrote the ciphertext with cat.
- Examined the correspondence between letters.
- Determined that the alphabet had been shifted differently (not regular ROT13).
- Applied tr using a personalized translation map to extract the message.

Logic: Higher-level substitution cipher modest difference from prior level, which calls for individual translation maps.

Krypton Level 4->5:

Password: GOLD

Objective: Analyze encrypted data and decrypt using hex manipulation.

Tools/Commands Used: cat, xxd, base64, hexdump

Step-by-Step Solution:

- Viewed the encrypted data file.
- Recognized it was base64 encoded or had hexadecimal patterns.
- Used xxd -r to reverse hexdump or decoded using base64 -d if applicable.
- Retrieved plain text and found the password.

Logic: Data is often encoded multiple times (e.g., hex + base64). Correct tool identification is key.

Krypton Level 5->6:

Password: RANDOM

Objective: Solve an RSA-related basic cryptography problem.

Tools/Commands Used: openssl, cat, calculator

Step-by-Step Solution:

- Identified that RSA encryption was involved based on hints.
- Located public key components (n and e).
- Factored the modulus n into primes (as they are very small numbers).
- Calculated the private key and decrypted the password.

Logic: RSA relies on the difficulty of factoring large primes — but here, primes are deliberately small to allow manual solving.

Krypton Level 6->7:

Password: LFSRISNOTRANDOM

Objective: Use advanced decoding to retrieve a secret password from a file.

Tools/Commands Used: cat, xxd, openssl, strings

Step-by-Step Solution:

- Extracted the data using xxd and reversed hex encoding.
- Found an encrypted file and attempted decryption using known passwords.
- Used openssl to attempt decryption methods like AES or RSA depending on file format.
- Retrieved the final password.

Logic: Multi-layer encryption teaches patience and strategy — check file format, file contents, and experiment with decryptions.

Lab 2: NATAS

(https://overthewire.org/wargames/natas/)

Objective:

The goal of the NATAS wargame is to gain hands-on experience in web security analysis through the investigation of actual vulnerabilities in web applications:

- Learn to inspect and manipulate web pages with browser developer tools.
- Recognize typical web vulnerabilities like hidden fields, insecure authentication, information disclosure, and weak server-side validation.
- Learn about HTTP requests, responses, cookies, and fundamental web application security.
- Utilize open-source tools such as cURL, Burp Suite, and browser extensions to exploit and analyze vulnerabilities.
- The NATAS lab equips interns for careers in website security audits and penetration testing.

Natas Level 0:

Goal: Find the password for the next level.

Step-by-Step:

1. Open the URL: http://natas0.natas.labs.overthewire.org
2. Page says: "You can find the password in the page source."
3. Right-click → View Page Source.
4. Inside an HTML comment, find the password.

Tools Used:

- Web Browser (View Page Source)

Logic:

- Sometimes sensitive info is hidden in HTML comments.

Natas Level 1:

Goal: Find the password for the next level.

Step-by-Step:

1. Login using credentials from Natas0.
2. The webpage says again: "Password is in the source code."
3. View Page Source.
4. Password found inside an HTML comment.

Tools Used:

- Web Browser (View Source)

Logic:

- Hidden info continues in source.

Natas Level 2:

Goal: Find the password for the next level.

Step-by-Step:

1. Login using previous password.
2. Page is almost empty.
3. Look for directories manually.
4. Find a /files/ directory.
5. Inside, find a .txt file with the password.

Tools Used:

- Web Browser
- URL manipulation

Logic:

- Default directory listing is often exposed.

Natas Level 3:

Goal: Find the password.

Step-by-Step:

1. Login.
2. No hint on the page.
3. View Page Source → find a comment mentioning "robots.txt".
4. Visit /robots.txt.
5. Find disallowed path /s3cr3t/.
6. Visit /s3cr3t/ and find password.

Tools Used:

- Browser
- Inspecting robots.txt

Logic:

- Robots.txt tells search engines what not to crawl — sensitive areas sometimes leak.

Natas Level 4:

Goal: Bypass referer checking.

Step-by-Step:

1. Page says you must come from http://natas5.natas.labs.overthewire.org.
2. Modify HTTP Referer header.
3. Use curl:
4. curl -u natas4:<password> -H "Referer: http://natas5.natas.labs.overthewire.org" http://natas4.natas.labs.overthewire.org
5. Or use browser extension like ModHeader.

Tools Used:

- cURL
- Browser extensions

Logic:

- Trusting HTTP headers (like Referer) is insecure.

Natas Level 5:

Goal: Fake being logged in.

Step-by-Step:

1. Page says "You are not logged in."
2. Look at cookies: see loggedin=0.
3. Modify it to loggedin=1.
4. Refresh page.

Tools Used:

- Browser (cookie editor)

Logic:

- Insecure cookie values.

Natas Level 6:

Goal: Get the secret via included file.

Step-by-Step:

1. Login.
2. View Source → mentions include secret.inc.

3. Can't access directly.
4. Source also mentions a function checkSecret.
5. Brute force or guess common values until success.

Tools Used:

- Browser
- Wordlist / brute-forcing (optional)

Logic:

- Code reveals internal filenames/functions.

Natas Level 7:

Goal: Directory traversal.

Step-by-Step:

1. URL looks like ?page=home.
2. Try ?page=../../etc/passwd.
3. Try paths until you find something interesting.

Tools Used:

- Browser
- URL manipulation

Logic:

- File inclusion via path traversal.

Natas Level 8:

Goal: Exploit weak hashing.

Step-by-Step:

1. View Source: uses encodedSecret = base64(secret + some salt).
2. Reverse base64.
3. Script tries comparing secret against encoded secret.
4. Brute-force the secret (small charset).

Tools Used:

- Python script (for brute-forcing)
- Base64 decode

Logic:

- Predictable secrets; insecure encoding.

Natas Level 9:

Goal: Command injection.

Step-by-Step:

1. Search feature → grep in the background.
2. Try ; cat /etc/natas_webpass/natas10.
3. Payload: needle=anystring; cat /etc/natas_webpass/natas10

Tools Used:

- Browser
- Shell injection basics

Logic:

- Unescaped user input inside shell commands.

Natas Level 10:

Goal: Command Injection (with input filtering).

Step-by-Step:

1. Similar to Natas9 — there's a grep behind the scenes.
2. This time, it filters ; and &.
3. Try other injections: use a | (pipe) instead.
4. Input: anystring | cat /etc/natas_webpass/natas11
5. Retrieve password.

Tools Used:

- Browser
- Shell injection tricks

Logic:

- Even if some characters are filtered, others (|, ||, &&) can be used.

Natas Level 11:

Goal: Weak encryption in cookies.

Step-by-Step:

1. Cookie data is Base64 encoded and encrypted with XOR.
2. Source code shows the XOR key used for encryption.

3. Decrypt the cookie value (XOR with key).
4. Modify admin field from 0 to 1.
5. Encrypt again, Base64 encode, set cookie.

Tools Used:

- Python (for XOR decryption)
- Cookie Editor

Logic:

- Reversible XOR encryption is weak if key is known.

Natas Level 12:

Goal: File upload vulnerability.

Step-by-Step:

1. Page allows uploading .jpg images.
2. Upload a .php file disguised as a .jpg.
3. Example: upload a file like shell.php containing PHP code.
4. After upload, find the file on the server.
5. Execute it to get the password.

Tools Used:

- Browser
- PHP payload
- Burp Suite (optional for upload tweaking)

Logic:

- Bad file type checking (only extension, not content).

Natas Level 13:

Goal: Same as Natas12, but stricter file validation.

Step-by-Step:

1. File extension and MIME type are checked.
2. Use a real JPEG header, then append PHP code.
3. Create a file starting with JPEG magic bytes (\xFF\xD8\xFF), then PHP code.
4. Upload, access the file, run PHP code.

Tools Used:

- Hex Editor
- Manual file crafting

Logic:

- MIME type and extension filtering can still be bypassed.

Natas Level 14:

Goal: SQL Injection (basic).

Step-by-Step:

1. Login form vulnerable to SQL injection.
2. Use:
3. username: anything" OR "1"="1
4. password: anything
5. Bypass login, get password.

Tools Used:

- Browser
- Manual SQL Injection

Logic:

- Unsanitized input in SQL queries.

Natas Level 15:

Goal: Blind SQL Injection (character by character).

Step-by-Step:

1. No error shown — page only shows success/failure.
2. Brute-force password one character at a time.
3. Send queries like:
4. natas16" AND password LIKE BINARY "a%" #
5. Script it using Python to automate.

Tools Used:

- Python scripting (requests)
- Burp Suite (optional)

Logic:

- Blind SQL injection uses boolean responses (true/false).

Natas Level 16:

Goal: Command Injection with input sanitization.

Step-by-Step:

1. Search function vulnerable to shell injection.
2. Filters &, ;, etc.
3. Bypass using newlines or whitespace tricks.
4. Payload example:
5. needle=anystring$(cat /etc/natas_webpass/natas17)

Tools Used:

- Browser
- Shell knowledge

Logic:

- Filters can often be bypassed by creative injection.

Natas Level 17:

Goal: Blind Command Injection (time delay).

Step-by-Step:

1. No output, but server takes longer to respond when correct.
2. Inject with sleep command.
3. Example:
4. anystring$(grep ^a /etc/natas_webpass/natas18 && sleep 5)
5. Brute-force password character by character.

Tools Used:

- Python scripting (time-based attack)

Logic:

- Response timing can leak information.

Natas Level 18:

Goal: Session ID guessing.

Step-by-Step:

1. Session ID is a number.
2. Bruteforce session IDs from 0 to 640 (small space).
3. Find admin session.

Tools Used:

- Python script (session bruteforcing)

Logic:

- Predictable session IDs are insecure.

Natas Level 19:

Goal: Decoding session ID.

Step-by-Step:

1. Session IDs are encoded (hexadecimal).
2. Decode hex → discover the format: user id + random stuff.
3. Modify your ID to admin (encoded).

Tools Used:

- Hex decoder
- Browser

Logic:

- Encoding ≠ Encryption. Hex can be reversed.

Natas Level 20:

Goal: Race condition.

Step-by-Step:

1. When setting username → race to set admin=1 cookie.
2. Multithreaded attack: register and login at the same time.
3. Achieve admin rights.

Tools Used:

- Python multithreading
- Burp Suite (optional)

Logic:

- Race conditions happen when multiple requests compete.

Natas Level 21:

Goal: Exploit dual server storage.

Step-by-Step:

1. Two different servers store data.
2. Save session as admin on one server.

3. Load session on second server.

Tools Used:

- Browser
- cURL
- Cookie editor

Logic:

- Lack of proper synchronization between servers.

Natas Level 22:

Goal: Infinite redirect.

Step-by-Step:

1. Page always redirects back to itself.
2. Manually stop redirects using curl -L (no follow).
3. View page content without redirect.

Tools Used:

- cURL (-i option)

Logic:

- HTTP redirects can hide content.

Natas Level 23:

Goal: PHP type juggling.

Step-by-Step:

1. Page checks if input equals a string in a loose way (==).
2. Submit "0e12345..." type strings.
3. PHP evaluates "0e..." == "0" as true.

Tools Used:

- PHP type juggling knowledge
- Browser

Logic:

- Loose comparison (==) in PHP is dangerous.

Natas Level 24:

Goal: PHP Object Injection.

Step-by-Step:

1. PHP unserializes user input.
2. Create a payload that injects malicious objects.
3. Send serialized object.

Tools Used:

- PHP serialization knowledge
- Python/PHP script to craft payload

Logic:

- Unsafe unserialize() usage.

Natas Level 25:

Goal: Local File Inclusion + Path Traversal.

Step-by-Step:

1. Load file based on user input.
2. Use path traversal like ../../../../../etc/natas_webpass/natas26
3. Bypass .php extension by adding null byte (%00) if needed.

Tools Used:

- URL manipulation
- Burp Suite

Logic:

- Inclusion bugs let you read server files.

Natas Level 26:

Goal: PHP Object Injection with file write.

Step-by-Step:

1. Unserialize() object — you can override __destruct.
2. Craft object that writes to a file.
3. Trigger file creation, read the password.

Tools Used:

- PHP payload crafting

Logic:

- Dangerous magic methods in PHP classes.

Natas Level 27:

Goal: Database race condition.

Step-by-Step:

1. SQL database used.
2. Trick server into inserting duplicate usernames.
3. Bypass login restrictions.

Tools Used:

- Python
- Burp Suite Intruder

Logic:

- Race conditions in database operations.

Natas Level 28:

Goal: Block cipher padding attack.

Step-by-Step:

1. Server uses block cipher encryption incorrectly.
2. Perform padding oracle attack.
3. Decrypt the cipher.

Tools Used:

- Python (padding oracle script)

Logic:

- Padding oracles leak encryption details.

Natas Level 29:

Goal: XOR encryption analysis.

Step-by-Step:

1. XOR encryption used badly.
2. Analyze known plaintext attacks.
3. Reconstruct key.

Tools Used:

- Python
- Cryptography knowledge

Logic:

- XOR is weak without proper randomness.

Natas Level 30:

Goal: Numeric SQL injection.

Step-by-Step:

1. SQL injection allowed with numeric values only.
2. Use arithmetic tricks to bypass filters.

Tools Used:

- Browser
- SQLi knowledge

Logic:

- Even numbers can carry SQL injection payloads.

Natas Level 31:

Goal: HTTP POST-based command injection.

Step-by-Step:

1. Server reads POST data unsafely.
2. Inject shell command in POST field.
3. Extract password.

Tools Used:

- cURL (sending custom POST)

Logic:

- POST parameters are just as dangerous as GET.

Natas Level 32:

Goal: Serialized object + file write.

Step-by-Step:

1. Upload serialized object that writes shell.

2. Trigger object execution.
3. Get password.

Tools Used:

- PHP serialization
- Python

Logic:

- Unsafe unserialize() leads to Remote Code Execution.

Natas Level 33:

Goal: Insecure PHP eval() usage.

Step-by-Step:

1. PHP page evaluates user input directly.
2. Inject PHP code as input.
3. Dump file contents.

Tools Used:

- Browser
- PHP knowledge

Logic:

- eval($_GET['x']) is extremely dangerous.

Natas Level 34:

Goal: File upload + race condition attack.

Step-by-Step:

1. Upload file while server checks MIME/type.
2. Race condition upload (change file after check, before save).
3. Upload PHP shell, extract password.

Tools Used:

- Burp Suite (for fast upload race)
- Python

Logic:

- Time-of-Check vs Time-of-Use (TOCTOU) bug.

Lab 3: LEVIATHAN

(https://overthewire.org/wargames/leviathan/)

Objective:

The goal of the LEVIATHAN wargame is to exercise privilege escalation methods and system-level security testing within a Linux environment. We should:

- Discover and exploit file permission vulnerabilities and insecurely secured binaries.
- Exercise fundamental Linux file system navigation, user permissions examination, and command-line inquiry.
- Enhance comprehension of concepts such as SUID (Set User ID) binaries and password-secured files.
- Gain experience in exploiting simple system misconfigurations for ethical hacking.
- The LEVIATHAN lab establishes core system security and post-exploitation skills for cybersecurity experts.

Given Username: Leviathan0 Password: Leviathan0

Leviathan Level 0->1:

Username: Leviathan1 Password: 3QJ3TgzHDq

Objective: Identify and recover a password from trusted files.

Tools & Commands Used: Cmd, linux(terminal) ls -la, cat

Step-by-Step Solution:

- Logged into the first level of Leviathan.
- Used ls -la to list hidden files and directories.
- Found a file that contained the password and it had readable permissions.
- Used cat to read the file and got the password.

Logic: Hidden or accessible files will sometimes have sensitive data if the permissions are not setup correctly.

Leviathan Level 1->2:

Username: Leviathan2 Password: NsN1HwFoyN

Objective: Exploit an exploitable binary in the file system in order to escalate.

Tools/Commands Info: ls -la, strings, ./binaryname

Step by Step Solution:

- Used the command ls -la, which resulted in identifying an executable (binary) with SUID permissions.
- Used the command strings binaryname to extract printable strings from the binary. This command could be used to help understand the behaviour of the binary.
- Corresponded with strings binaryname and found clues (ie: hardcoded password or filename).
- Executed the binary correctly to retrieve the password.

Logic: SUID binaries are risky as they are given elevated privileges. SUID binaries can show weaknesses, such as predictable password checking.

Leviathan Level 2->3:

Username: Leviathan3 Password: f0n8h2iWLP

Objective: Exploit a binary that takes user input incorrectly.

Tools/Commands Used: strings, grep, manual trial-and-error input

Step-by-Step Solution:

- Found the binary with a simple ls -la.
- Used strings and grep to find any possible passwords or behaviours of the binary.
- Saw that the binary was asking for a password input.
- Guessed/tried the passwords that were visible through strings output to unlock the binary to get the next password.

Logic: Binaries usually will have their passwords hardcoded or some way of infrequently implemented input validation causing exploitable situations.

Leviathan Level 3->4:

Username: Leviathan4 Password: WG1egElCvO

Objective: Tackle another challenge with mishandling file permissions.

Tools/Commands: cat, ls -la, strings, ./binary

Step-by-Step Solution:

- Used ls -la to get a list of available files and binaries.
- Used strings on a suspicious binary.
- Noticed it reads the password from a file, and found the file location and contents to get the password.

Logic: Programs that read from user controllable files without any restrictions are very vulnerable.

Leviathan Level 4->5:

Username: Leviathan5 Password: 0dyxT7F4QD

Objective: Take advantage of binary behavior to read a password-protected file.

Tools/Commands Used: ls, strings, chmod (if allowed)

Step-by-Step Solution:

- Found a binary with ls.
- Used strings to analyze the binary prompts, and file names.
- Found that it opened a file with limited permission checks.
- Found a way to bypass those checks, opened the intended file, and obtained the password.

Logic: Binaries that do not properly validate user inputs often unintendedly provide access to files.

Leviathan Level 5->6:

Username: Leviathan6 Password: szo7HDB88w

Objective: Abuse a vulnerable SUID program with env variables.

Tools/Commands Used: env, strings, export

Step-by-Step Solution:

- Found the SUID binary with ls -la.
- Checked the environment it was relying on (i.e. PATH or ENV variables).
- Modified environmental variables that would affect how the binary executed.
- Successfully ran the binary to get the next password.

Logic: If SUID programs do not sanitize the environment variables that they rely on, the attacker could manipulate the programs behaviour.

Leviathan Level 6->7:

Username: Leviathan7 Password: qEs5Io5yM8

Objective: Abuse another binary where access rights or checks are mishandled.

Tools/Commands Used: ls, cat, strings

Step-by-Step Solution:

- Found the executable binary.
- Looked at the binary using strings to check for hidden messages or function calls.

- If required, used gdb for deeper reverse engineering.
- Found a vulnerability or hidden password directly in the binary.

Logic: Reverse engineering simple binaries can disclose hardcoded secrets or logic errors.