Tech: Cybersecurity

Hrithik.P

25BYB0215

Task 1: 3 Levels

gdg_part1

Tools Used

- Command Prompt

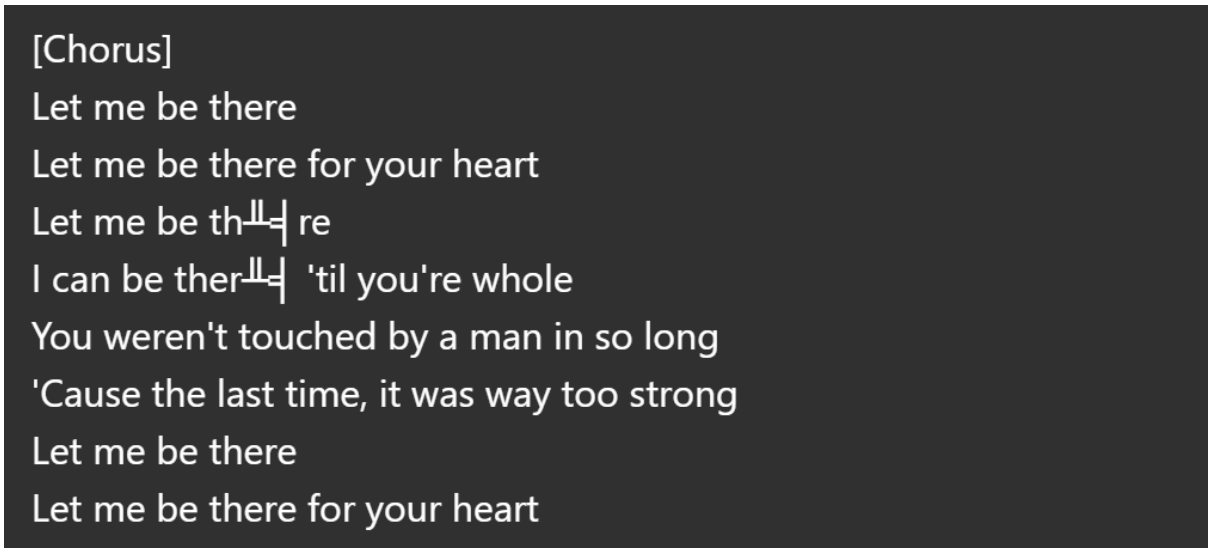
- Notepad (encoding inspection)

Approach

The file `xotwod.txt` initially appeared to contain only song lyrics.

However, certain characters displayed inconsistently, indicating an encoding or homoglyph trick.

On closer inspection, the letter `e` in the word "there" was replaced with a Cyrillic `e`, which visually resembles the Latin character but differs in Unicode representation.



Conclusion

The repeated altered word highlighted the intended flag fragment.

Flag Fragment

part1 = there

gdg_part2

Challenge Description

In this stage of the challenge, a single image file named heheheha.png was provided.

At first glance, the file appeared to be a normal PNG image containing logos and stickers.

Given the context of a CTF, it was assumed that the image contained hidden data using steganographic techniques.

Tools Used

Kali Linux

zsteg (LSB steganography analysis tool)

Terminal

Initial Analysis

The image was first inspected visually and showed no obvious anomalies.

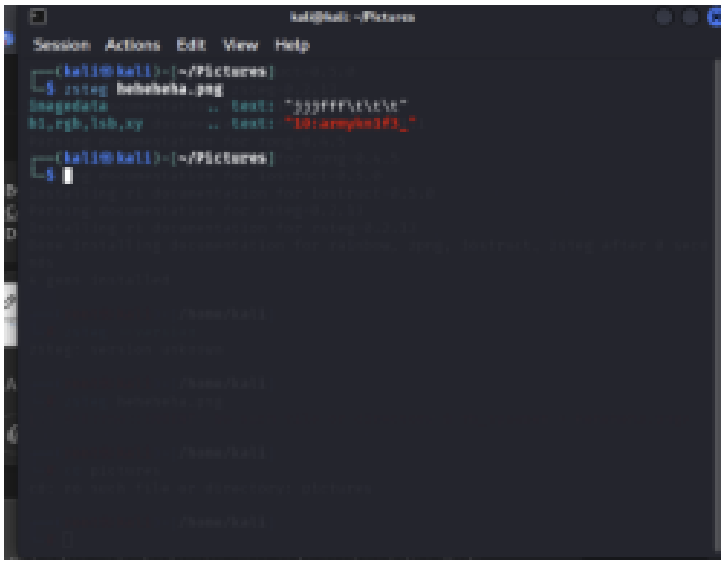
Basic checks such as metadata inspection and file structure analysis did not reveal any hidden information.

The tool zsteg was chosen because it is specifically designed to detect hidden data in PNG and BMP images.

The image was analyzed using the following command:

zsteg heheheha.png

zsteg systematically checked different color channels, bit planes, and pixel traversal orders.



Key Finding

Among the outputs produced by zsteg, the following line contained readable hidden text:

b1,rgb,lsb,xy: text: "10-armykm1f3"

This indicates that:

b1: 1 bit per channel was used

rgb: Data was hidden across the Red, Green, and Blue channels

lsb: Least Significant Bit steganography was used

xy: Pixels were read in standard row-wise order

Conclusion

The readable string extracted from the image represents the second fragment of the flag.

The data was successfully hidden using LSB steganography within the RGB channels of the image.

Flag Fragment

part2 = 10-armykm1f3

gdg_part3

In Part 3 of the challenge, a compressed file named qr_code_zipbomb.rar was provided.

The description suggested that the archive contained a very large number of QR codes, hinting at automation rather than manual scanning.

The objective was to extract meaningful information (the third fragment of the flag) hidden among thousands of decoy QR codes.

The archive size was small, but extraction revealed thousands of QR images.

Manually opening and scanning each QR code was impractical.

This indicated the use of a zipbomb-style noise attack, where most data is intentionally useless.

Therefore, the correct approach was to automate QR scanning using Python.

Operating System: Windows

Python Version: 3.11.8

Tools Used:

zipfile (built-in Python module)

Pillow (image handling)

pyzbar (QR decoding)

Command Prompt (CMD)

Step 1: Script Preparation

A Python script was created to:

Open the compressed archive directly

Iterate through all image files

Decode any QR code found

Print the extracted content

This avoided manual extraction and ensured safe processing of the zipbomb.

The script was executed using:

python read_qr.py

During execution:

The script scanned thousands of QR codes

Most QR codes contained decoy messages, such as:

“Keep scanning”

“Nothing useful here”

“Just a decoy”

The output intentionally flooded the terminal with noise

This behavior was expected and confirmed the challenge’s deceptive natur

Amid the decoy output, one QR code produced a unique Base64-encoded string:

PDwtLS0tcGFydDM9Z2cxb2x9LS0tLT4+

This was the only non-decoy output.

The extracted string was decoded from Base64:

<!---part3=gg1ol}--->

From this, the third fragment of the flag was identified as:

gg1ol}

combined flag

CTF{there-10-armykm1f3-gg1ol}

Task 2: Hidden Recipe

This CTF challenge required examining an unfamiliar network service hosted on a private IP using a Kali Linux virtual machine running in VMware Workstation. During initial reconnaissance, several open ports were discovered on the target system at 192.168.29.111. Early interaction attempts relied on common HTTP-based tools such as curl, but most ports responded with empty replies, indicating the service was not a typical web application. One port exposed a custom banner identifying itself as NRDP/2025.2.0.0, pointing toward a proprietary or nonstandard protocol.

Further probing with OpenSSL revealed that port 35743 supported TLS connections, and a successful TLS 1.2 handshake could be established. However, the service did not display a conventional banner and instead appeared to wait for direct user input. Manual commands like HELP, MENU, and FLAG produced incomplete or inconsistent outputs, suggesting that the service operated interactively. Attempts to script or automate interaction using piped input, printf, and tools such as ncat consistently failed, resulting in timeouts or no responses. This behavior indicated that the service likely required a genuine interactive terminal and was intentionally designed to reject non-interactive input, a common tactic in CTF challenges to prevent automation.

To overcome this limitation, the OpenSSL client was executed through rlwrap to provide proper terminal and readline support. By using the command rlwrap openssl s_client -connect 192.168.29.111:35743 -tls1_2 -quiet, a stable interactive session was successfully established. Within this environment, issuing the FLAG command prompted a valid response from the server, revealing the flag and confirming successful completion of the challenge.

Other investigative approaches, such as filesystem inspection, browser-based testing, HTTP method enumeration, and analysis of mounted disk images, were deliberate distractions and yielded no meaningful results. This reinforced that the intended solution centered on understanding and correctly interacting with

the custom protocol rather than exploiting web services or extracting stored data. Overall, the challenge emphasized practical skills in low-level service interaction, TLS-based socket communication, and the necessity of proper terminal handling when working with interactive network services.