The goal of this puzzle is to expose the participant to basic steganography, file recovery, and cryptography principles and techniques. The puzzle can reasonably be categorized into three respective parts: video to image, image to .tar containing a .txt and .py script, modifying the .txt information and py script to generate a barcode containing the flag

Step one: The participant is provided with a video file which they are expected to investigate using common techniques, eventually discovering the existence of JPG data within the .mov file, from which they must figure out how to extract this image from the video. The participant can recover the image in a variety of ways since it's simply appended to the end of the video file. They are intended to use the numbers in the filename as the byte size of the hidden image (to help with extraction) but they should also be able to recover arbitrary sizes until the image is contained in the recovered data (1000, 10000, 100000).

Step two: Once the participant has recovered the hidden image they face another steganography challenge. This step expects the user to extract a hidden file from the image which was hidden using Least Significant Bit (LSB) steganography. This is a very common method of hiding data in images. After extracting the data from the image the participant is provided with a .tar.gz file they can unzip, leading them to step three.

Step three: The participant is presented with a .txt file of seemingly random strings of numbers and a python script with unhelpfully labeled variables and methods. They need to investigate the functionality of the python script and recognize it generates an image in black and white, from ones and zeroes. At this point the contestant should realize they need to convert the .txt file to binary and feed it to the .py script to generate an image, which is a QR code. (The script requires numpy and Image be imported to work but it's technically not required to solve the challenge as the user can also use their own binary to qr code script to solve it). I chose to include the python script to make the emphasis of the challenge conceptual rather than the emphasis being on developing a binary to qr code script. I made sure to leave out a few details in the python script to ensure the participant has to at least look and attempt to understand the .py script before running.

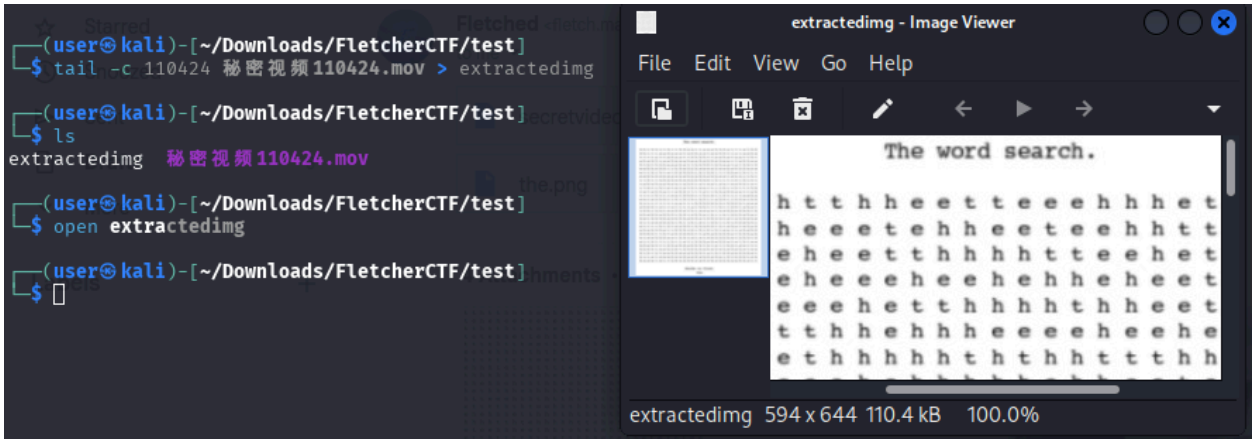
Intended Solution:

Part one: Investigate file for appended information (I prefer to do this with binwalk)

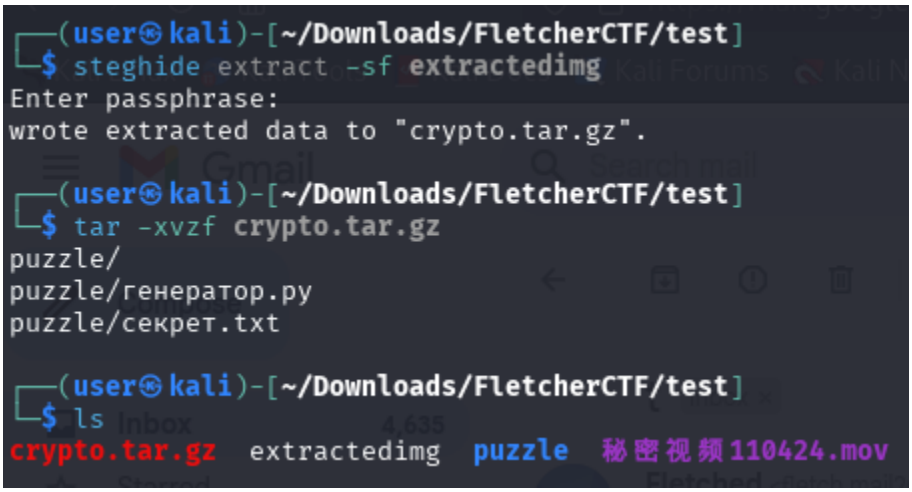
```
┌──(user㉿kali)-[~/Downloads/FletcherCTF/test]
└─$ binwalk 秘密视频110424.mov

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
23141         0×5A65          YAFFS filesystem root entry, big endian, type symlink, v1 root directory
4029116       0×3D7ABC        JBOOT STAG header, image id: 13, timestamp 0×CB77B9BD, image size: 472359480 bytes, image JBOOT checksum: 0×6935, header JBOOT checksum: 0×76B1
22407231      0×155E83F       JBOOT STAG header, image id: 5, timestamp 0×51014AEA, image size: 1369710892 bytes, image JBOOT checksum: 0×1A00, header JBOOT checksum: 0×980
24232526      0×171C24E       JPEG image data, JFIF standard 1.01
```

We find JPEG image data in the .mov file, this suggests there's an image we are

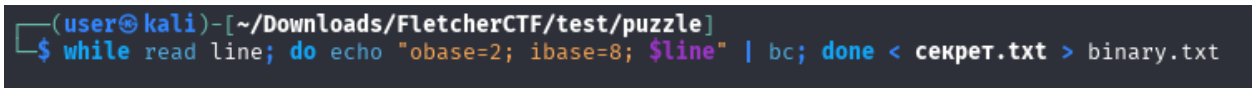
supposed to recover. I use tail to attempt recovering a file at the “tail” end of the video file, using 110424 as recovered file size from context clues or the hint in the readme.



Part two: We use a tool like steghide to extract information that may be hidden with LSB steganography, trying first with no passphrase. We get a tarball file that can be extracted with tar -xvzf crypto.tar.gz, giving us a directory with a .txt and .py file.



Part three: After examining the .py script and realizing it produces a qr code from a binary file, we recognize that the .txt file is in octal and we convert the octal to binary. You can easily do this with an online calculator or a script. I'm using a batch script to convert to binary and write to a new .txt file.



After this I add leading zeros in the new file until every row has 25 characters.
I then add this as the input file for the .py script and make the script run func_1() when called.

```
  GNU nano 8.3
from PIL import Image
import numpy as np

def func_1(input_file="binary.txt", output_file="out.png"):
    with open(input_file, "r") as f:
        string = f.read().replace(" ", "").replace("\n", "")

    if len(string) != 625:
        raise ValueError("Error! Incorrect input!")

    rarray = np.array([int(char) for char in string]).reshape((25, 25))

    img = Image.new("1", (25, 25))
    p = img.load()

    for y in range(25):
        for x in range(25):
            p[x, y] = 0 if rarray[y, x] == 1 else 255

    img = img.resize((250, 250), Image.NEAREST)
    img.save(output_file)
    print(f"Saved {output_file}")

if __name__ == "__main__":
        func_1()
```

When we run the script we are given a qr code that produces the flag when scanned: