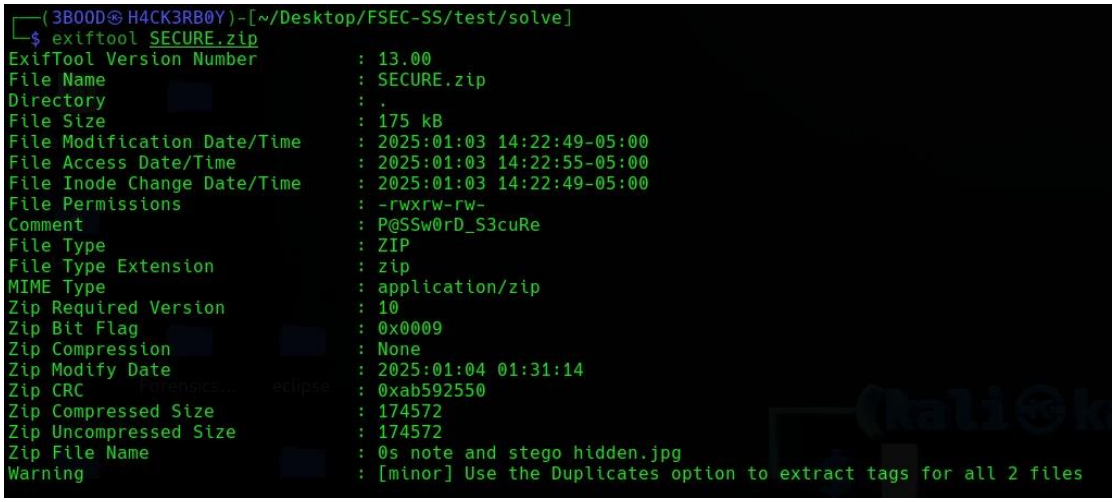


Examine the metadata of the ZIP file, and you'll find this comment: **P@SSw0rD_S3cuRe**

Examine the type of the file, and you'll notice: **Compression=store**. This indicates that the file is stored without compression, making it vulnerable to a **known-plaintext attack**.

What is a known-plaintext attack?

It's a cryptographic attack where the attacker has access to both the plaintext (known data) and its corresponding ciphertext (encrypted data). By leveraging this knowledge, the attacker can deduce the encryption key or password.

```
┌──(3BOOD㉿H4CK3RB0Y)-[~/Desktop/FSEC-SS/test/solve]
└─$ zipinfo SECURE.zip
Archive:  SECURE.zip
Zip file size: 175038 bytes, number of entries: 2
-rw-a--     3.1 fat   174560 BX stor 25-Jan-04 01:31 0s note and stego hidden.jpg
-rw-a--     3.1 fat       89 BX stor 25-Jan-04 01:35 keys.txt
2 files, 174649 bytes uncompressed, 174649 bytes compressed:  0.0%
```

Using the tool zipinfo, examine the contents of the ZIP file to view the file names, sizes, and permissions. Inside, you'll find two files:

1. **key.txt** – a small text file.
2. **0s note and stego hidden.jpg** – an image file.

```
┌──(3BOOD㉿H4CK3RB0Y)-[~/Desktop/FSEC-SS/test/solve]
└─$ echo -ne '\xFF\xD8\xFF\xE0\x00\x10\x4A\x46\x49\x46\x00\x01\x01\x02\x00\x25' > known_header.bin

┌──(3BOOD㉿H4CK3RB0Y)-[~/Desktop/FSEC-SS/test/solve]
└─$ hexdump -C known_header.bin
00000000  ff d8 ff e0 00 10 4a 46  49 46 00 01 01 02 00 25  |......JFIF.....%|
00000010
```

To exploit the JPEG file, I used its **hexadecimal header**—the first **16 bytes** of the file:

FF D8 FF E0 00 10 4A 46 49 46 00 01 01 02 00 25

I wrote these known bytes into a .bin file using the following command:

echo -ne '\xFF\xD8\xFF\xE0\x00\x10\x4A\x46\x49\x46\x00\x01\x01\x02\x00\x25' > known_header.bin

This .bin file serves as the **known plaintext** for the attack.

```
┌──(3BOOD㉿H4CK3RB0Y)-[~/Desktop/FSEC-SS/test/solve]
└─$ bkcrack -C SECURE.zip -c "0s note and stego hidden.jpg" -p known_header.bin
bkcrack 1.7.0 - 2024-05-26
[14:25:05] Z reduction using 9 bytes of known plaintext
100.0 % (9 / 9)
[14:25:05] Attack on 752227 Z values at index 6
Keys: cfceec80 60030234 3d117674
65.6 % (493568 / 752227)
Found a solution. Stopping.
You may resume the attack with the option: --continue-attack 493568
[14:35:30] Keys
cfceec80 60030234 3d117674
```

I used the tool **bkcrack** to perform the known-plaintext attack. Here's the command I executed:

bkcrack -C SECURE.zip -c "0s note and stego hidden.jpg" -p known_header.bin

Explanation of the Command:

- **-C SECURE.zip**: Specifies the target encrypted ZIP file (SECURE.zip).
- **-c "0s note and stego hidden.jpg"**: Specifies the encrypted file inside the ZIP that we're targeting (the JPEG file).
- **-p known_header.bin**: Provides the known plaintext file (known_header.bin), which contains the first 16 bytes of the JPEG file.

This command performs the known-plaintext attack and extracts the **three main encryption keys** required to decrypt the ZIP file.

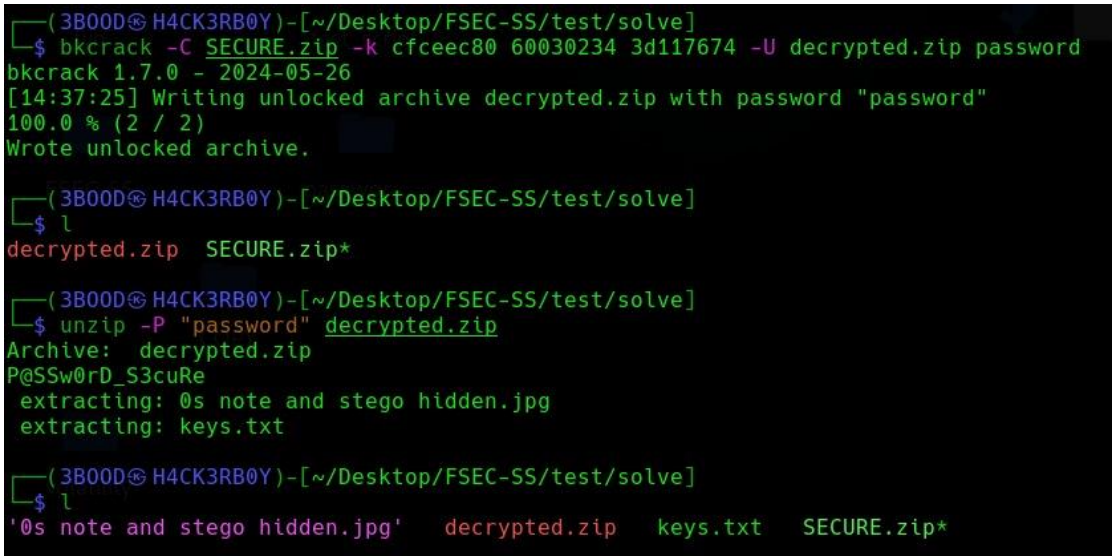
Extracted Keys:

The attack successfully recovered the following keys:

Key 1: cfceec80

Key 2: 60030234

Key 3: 3d117674



I used the following command to create a new decrypted ZIP file:

bkcrack -C SECURE.zip -k cfceec80 60030234 3d117674 -U decrypted.zip password

Explanation of the Command:

- **-C SECURE.zip**: Specifies the target encrypted ZIP file (SECURE.zip).
- **-k cfceec80 60030234 3d117674**: Provides the three encryption keys recovered from the known-plaintext attack.
- **-U decrypted.zip**: Creates a new ZIP file (decrypted.zip) with the same contents as the original but without encryption.
- **password**: A placeholder password for the new ZIP file (it will not be encrypted).

This command creates a new ZIP file (decrypted.zip) containing the same files as the original but protected with the new password (password).

To extract the files from the new ZIP file, I used the following command:

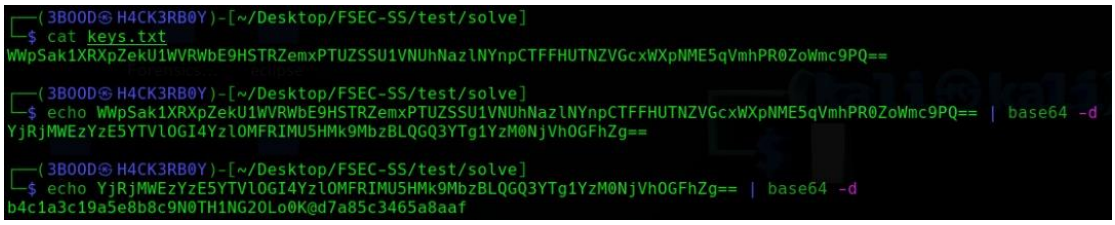
unzip -P "password" decrypted.zip

xplanation of the Command:

- **-P "password"**: Specifies the password for the ZIP file (password).
- **decrypted.zip**: The decrypted ZIP file to extract.

This command extracts the two files from the ZIP archive:

1. **0s note and stego hidden.jpg** – the image file.
2. **keys.txt** – the text file.



The first file contained a **Base64-encoded string**. I decoded it twice using the following commands:

echo WWpSak1XRXpZekU1WVRWbE9HSTRZemxPTUZSSU1VNUhNazlNYnpCTFFHUTNZVGcxWXpNME5qVmhPR0ZoWmc9PQ== | base64 -d

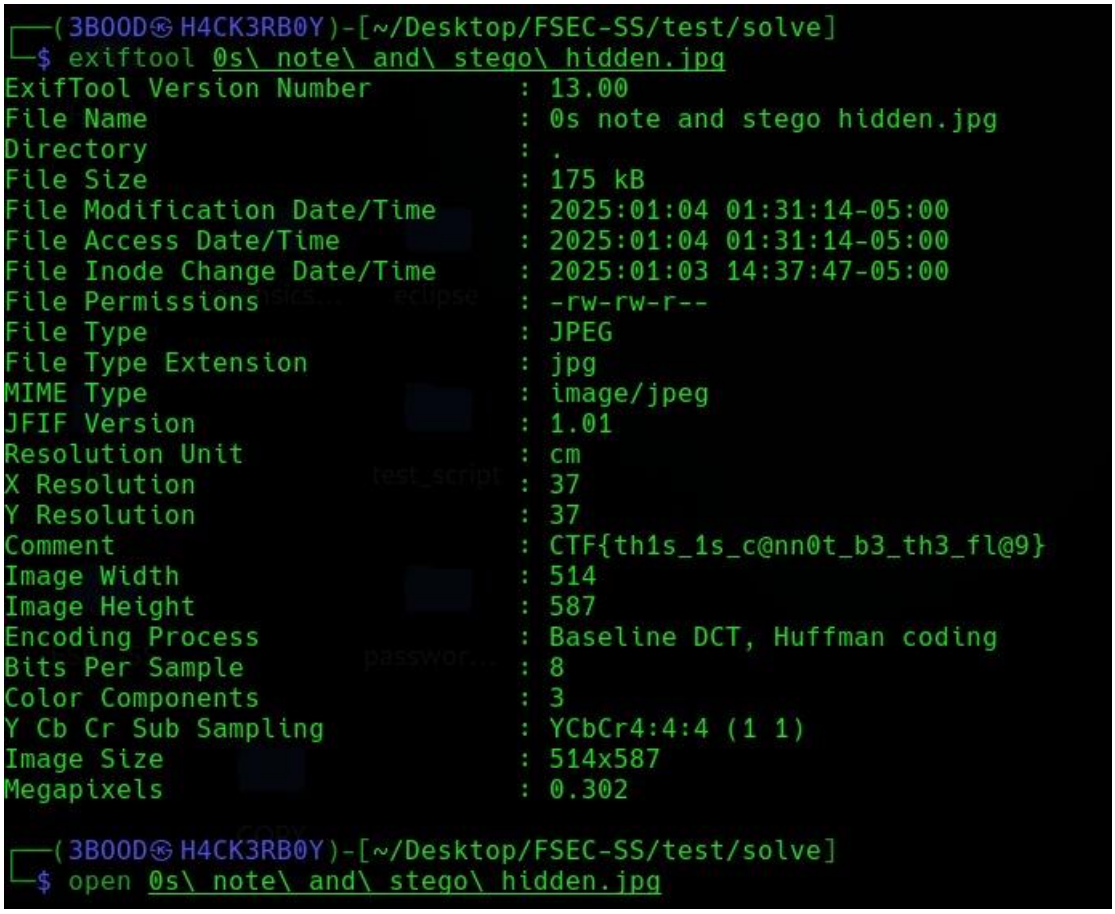
echo YjRjMWEzYzE5YTVlOGI4YzlOMFRIMU5HMk9MbzBLQGQ3YTg1YzM0NjVhOGFhZg== | base64 -d

output: b4c1a3c19a5e8b8c9N0TH1NG2OLo0K@d7a85c3465a8aaf

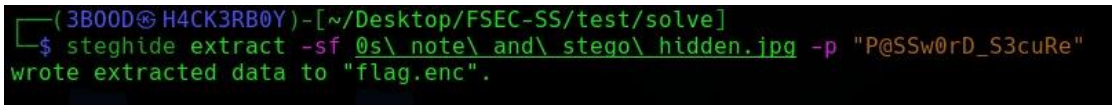
The decoded string appears to consist of **three distinct parts**:

1. **b4c1a3c19a5e8b8c9**
2. **N0TH1NG2OLo0K@**
3. **d7a85c3465a8aaf**

Based on the file name (keys.txt), these parts likely represent **keys or passwords** that can be used in the next steps of the challenge.



Upon examining the JPEG file (0s note and stego hidden.jpg), I found a **fake flag** embedded in its metadata. This was likely placed as a distraction. The file name, **stego hidden**, strongly suggests that the real flag is hidden using **steganography**. This points to the use of the **steghide** tool to extract hidden data.



Using the password found in the ZIP file's metadata (**P@SSw0rD_S3cuRe**), I ran the following command to extract hidden data from the JPEG file:

steghide extract -sf 0s\ note\ and\ stego\ hidden.jpg -p "P@SSw0rD_S3cuRe"

Explanation of the Command:

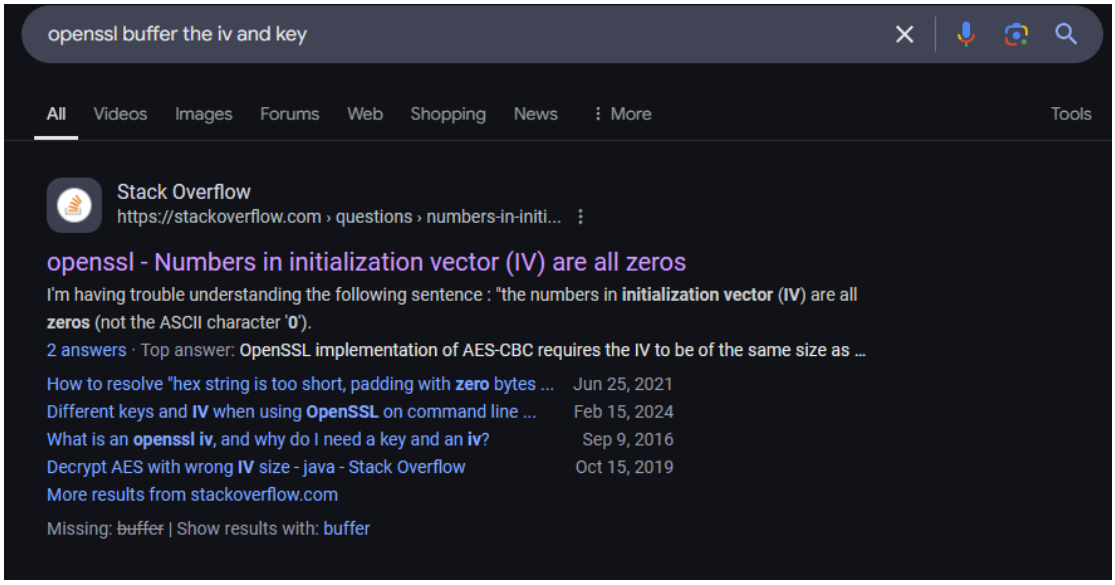
- **extract**: Instructs steghide to extract hidden data from the file.
- **-sf "0s note and stego hidden.jpg"**: Specifies the JPEG file to analyze.
- **-p "P@SSw0rD_S3cuRe"**: Provides the password required to extract the hidden data.

Since the file name suggests that the image holds hidden information, I decided to open the JPEG file (0s note and stego hidden.jpg) to look for any visual clues or hints about how to decrypt flag.enc.

The hint to OpenSSL and the 256-bit encryption method in the PNG file can be interpreted from the paragraph as follows:

- "One approach to safeguarding your data involves a method that's known for its strength and reliability," which hints at **AES (Advanced Encryption Standard)**, a widely-used and strong encryption method.
- The line "To make sure everything stays secure, consider exploring robust encryption techniques, one of which involves using a powerful 256-bit key" directly refers to **256-bit encryption**, which is commonly associated with **AES-256**.
- "There are trusted resources available that can help you apply these methods to keep your systems safe from prying eyes," suggests using trusted resources, such as **OpenSSL**, which is commonly used for AES-256 encryption and decryption.

The mention of the "256-bit key" strongly indicates AES-256, supported by OpenSSL, a widely-used cryptographic tool, aligning with the paragraph's focus on encryption strength and reliability.



The issue is that we don't have the IV (Initialization Vector) and key to decrypt the flag. Since partitioning is crucial, it might be relevant now. The image suggests that the remaining IV and key are zeros, and after further investigation, we found that they are actually padded with zeros.

To decrypt the flag.enc file, I used the openssl tool with the following command:

openssl enc -d -aes-256-cbc -in flag.enc -out flag_de.txt -K b4c1a3c19a5e8b8c9 -iv d7a85c3465a8aaf

Explanation of the Command:

- **enc -d**: Decrypts the input file.
- **-aes-256-cbc**: Specifies the encryption algorithm (AES-256 in CBC mode).
- **-in flag.enc**: The encrypted input file.
- **-out flag_de.txt**: The output file where the decrypted data will be saved.
- **-K b4c1a3c19a5e8b8c9**: The encryption key (hexadecimal string).
- **-iv d7a85c3465a8aaf**: The initialization vector (IV) used for decryption.

After decryption, I viewed the contents of the output file (flag_de.txt) using the following command:

cat flag_de.txt

the flag:
FSEC-SS{kn0wn-pl41nt3xt_@tt@ck_&_0p3nssl_3ncrypt10n}