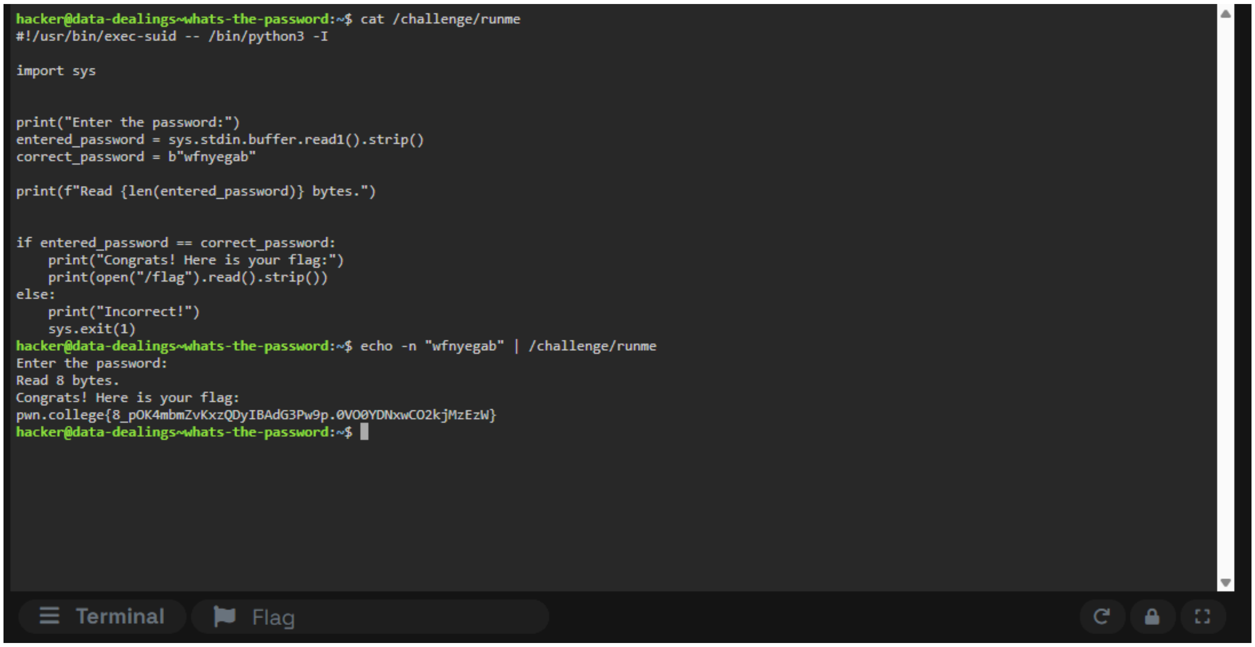
Jaleel Williamson
jayw-713
CSCI 400 Lab 2
9/8/25 to 9/10/25

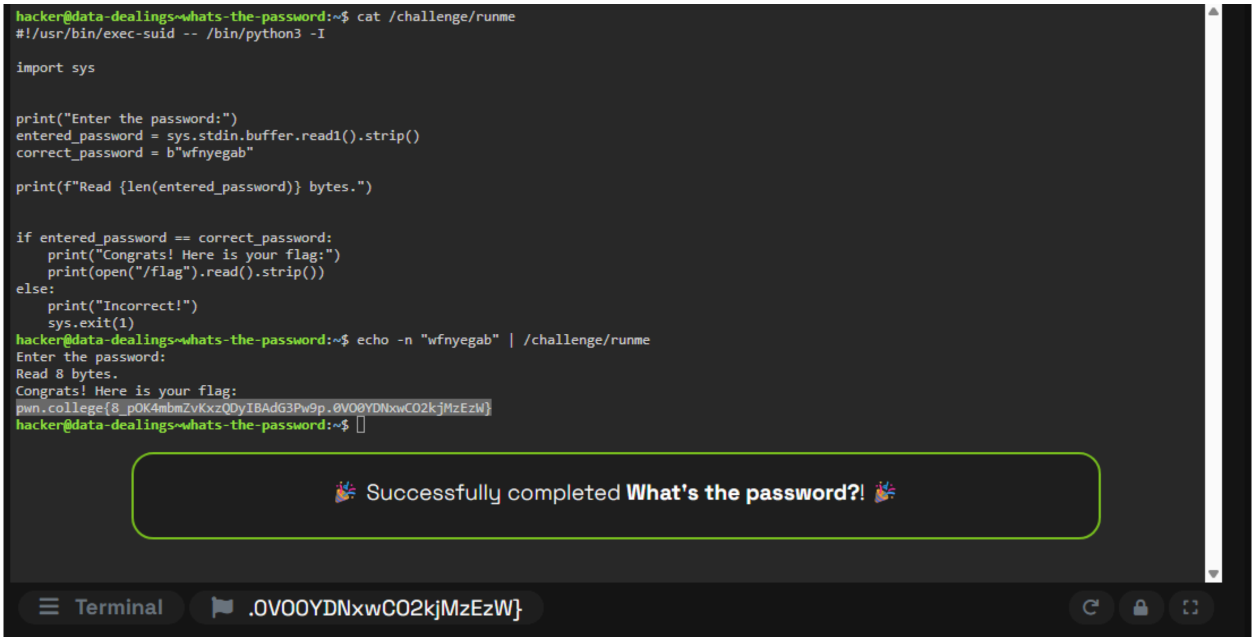
Dealing with Data: https://pwn.college/fundamentals/data-dealings/

- **What's the password**



```
hacker@data-dealings~whats-the-password:~$ cat /challenge/runme
#!/usr/bin/exec-suid -- /bin/python3 -I

import sys


print("Enter the password:")
entered_password = sys.stdin.buffer.read1().strip()
correct_password = b"wfnyegab"

print(f"Read {len(entered_password)} bytes.")


if entered_password == correct_password:
    print("Congrats! Here is your flag:")
    print(open("/flag").read().strip())
else:
    print("Incorrect!")
    sys.exit(1)
hacker@data-dealings~whats-the-password:~$ echo -n "wfnyegab" | /challenge/runme
Enter the password:
Read 8 bytes.
Congrats! Here is your flag:
pwn.college{8_pOK4mbmZvKxzQDyIBAdG3Pw9p.0VO0YDNxwCO2kjMzEzW}
hacker@data-dealings~whats-the-password:~$ 
```

This screenshot is the same as the following screenshot before successful capture.

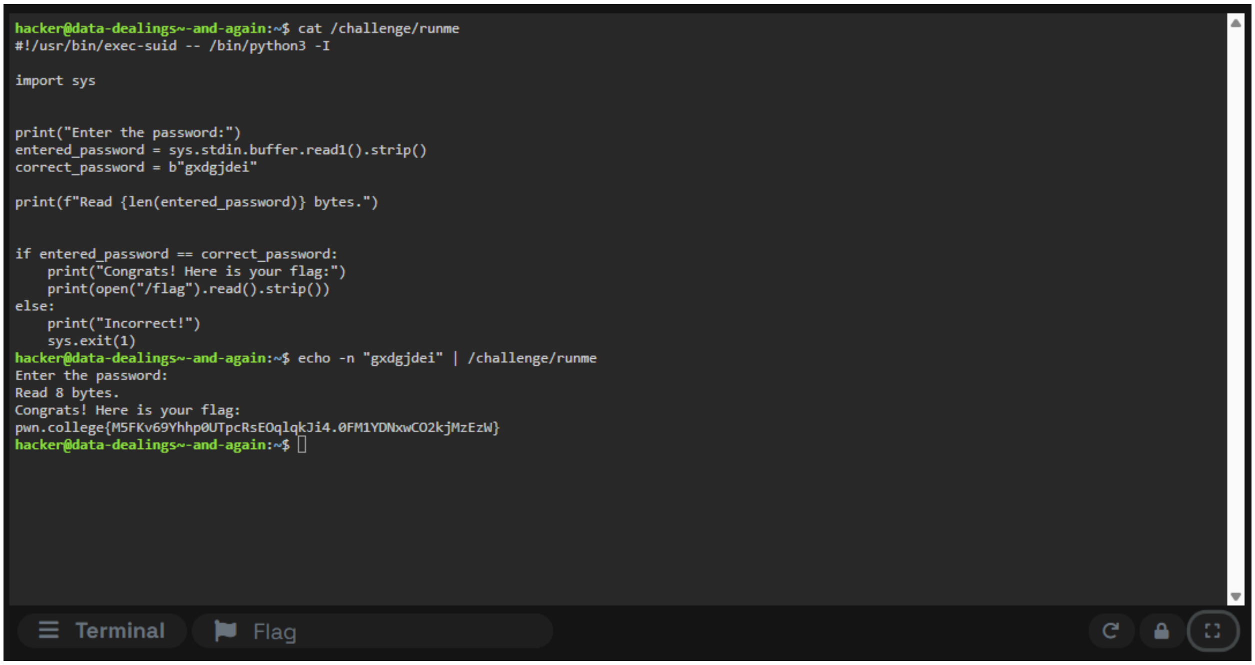


```
hacker@data-dealings~whats-the-password:~$ cat /challenge/runme
#!/usr/bin/exec-suid -- /bin/python3 -I

import sys


print("Enter the password:")
entered_password = sys.stdin.buffer.read1().strip()
correct_password = b"wfnyegab"

print(f"Read {len(entered_password)} bytes.")


if entered_password == correct_password:
    print("Congrats! Here is your flag:")
    print(open("/flag").read().strip())
else:
    print("Incorrect!")
    sys.exit(1)
hacker@data-dealings~whats-the-password:~$ echo -n "wfnyegab" | /challenge/runme
Enter the password:
Read 8 bytes.
Congrats! Here is your flag:
pwn.college{8_pOK4mbmZvKxzQDyIBAdG3Pw9p.0VO0YDNxwCO2kjMzEzW}
hacker@data-dealings~whats-the-password:~$ 
```

🎉 Successfully completed **What's the password?!** 🎉

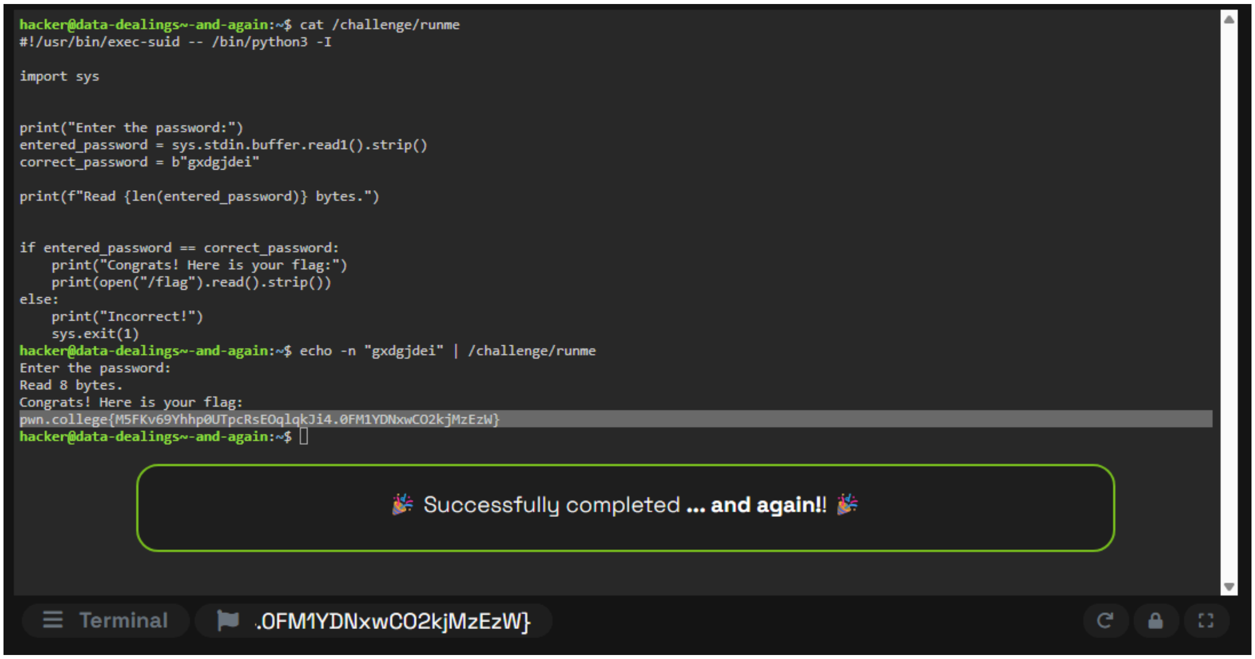
To get the flag, I read the source code of **/challenge/runme** using **cat** and found the **hard-coded** password **wfnyegab**. Then, I piped this password directly into the program using **echo -n "wfnyegab" | /challenge/runme**, which **bypassed** the check and output the flag. This worked because the program **compares** the **input** directly to this **byte string**.

- **... and again!**



```
hacker@data-dealings~-and-again:~$ cat /challenge/runme
#!/usr/bin/exec-suid -- /bin/python3 -I

import sys


print("Enter the password:")
entered_password = sys.stdin.buffer.read1().strip()
correct_password = b"gxdgjdei"

print(f"Read {len(entered_password)} bytes.")


if entered_password == correct_password:
    print("Congrats! Here is your flag:")
    print(open("/flag").read().strip())
else:
    print("Incorrect!")
    sys.exit(1)
hacker@data-dealings~-and-again:~$ echo -n "gxdgjdei" | /challenge/runme
Enter the password:
Read 8 bytes.
Congrats! Here is your flag:
pwn.college{M5FKv69Yhhp0UTpcRsEOqlqkJi4.0FM1YDNxwCO2kjMzEzW}
hacker@data-dealings~-and-again:~$ 
```

This screenshot is the same as the following screenshot before successful capture.

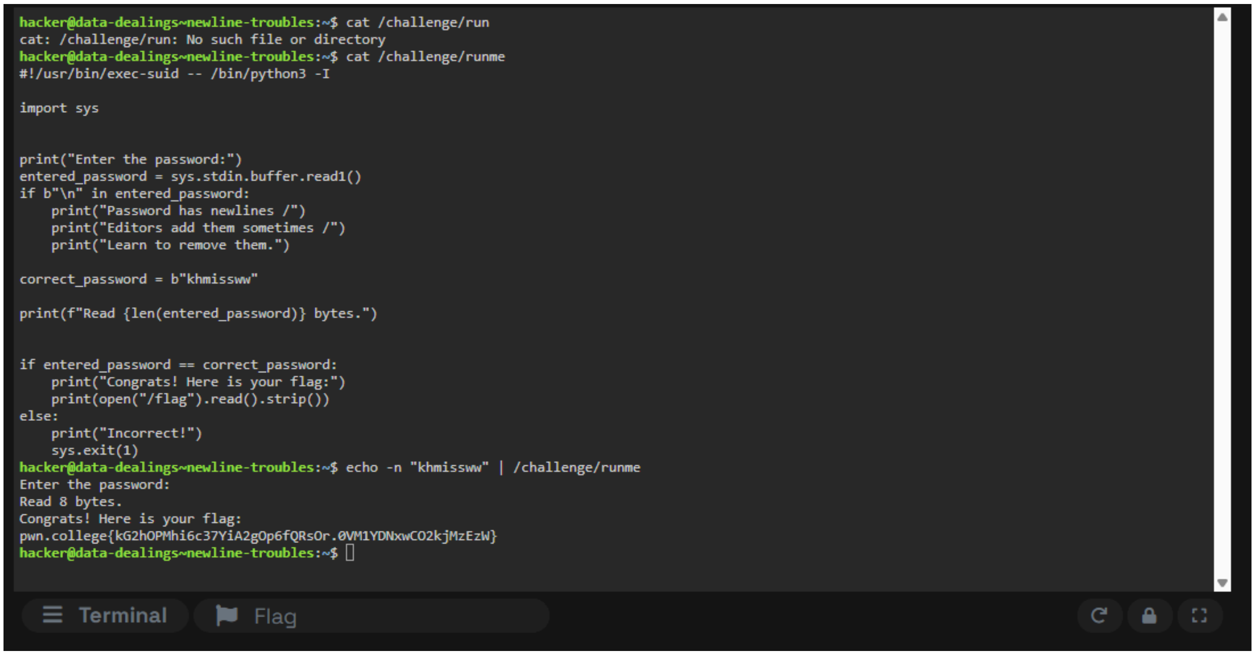


```
hacker@data-dealings~-and-again:~$ cat /challenge/runme
#!/usr/bin/exec-suid -- /bin/python3 -I

import sys


print("Enter the password:")
entered_password = sys.stdin.buffer.read1().strip()
correct_password = b"gxdgjdei"

print(f"Read {len(entered_password)} bytes.")


if entered_password == correct_password:
    print("Congrats! Here is your flag:")
    print(open("/flag").read().strip())
else:
    print("Incorrect!")
    sys.exit(1)
hacker@data-dealings~-and-again:~$ echo -n "gxdgjdei" | /challenge/runme
Enter the password:
Read 8 bytes.
Congrats! Here is your flag:
pwn.college{M5FKv69Yhhp0UTpcRsEOqlqkJi4.0FM1YDNxwCO2kjMzEzW}
hacker@data-dealings~-and-again:~$ 
```

🎉 Successfully completed **... and again!!** 🎉

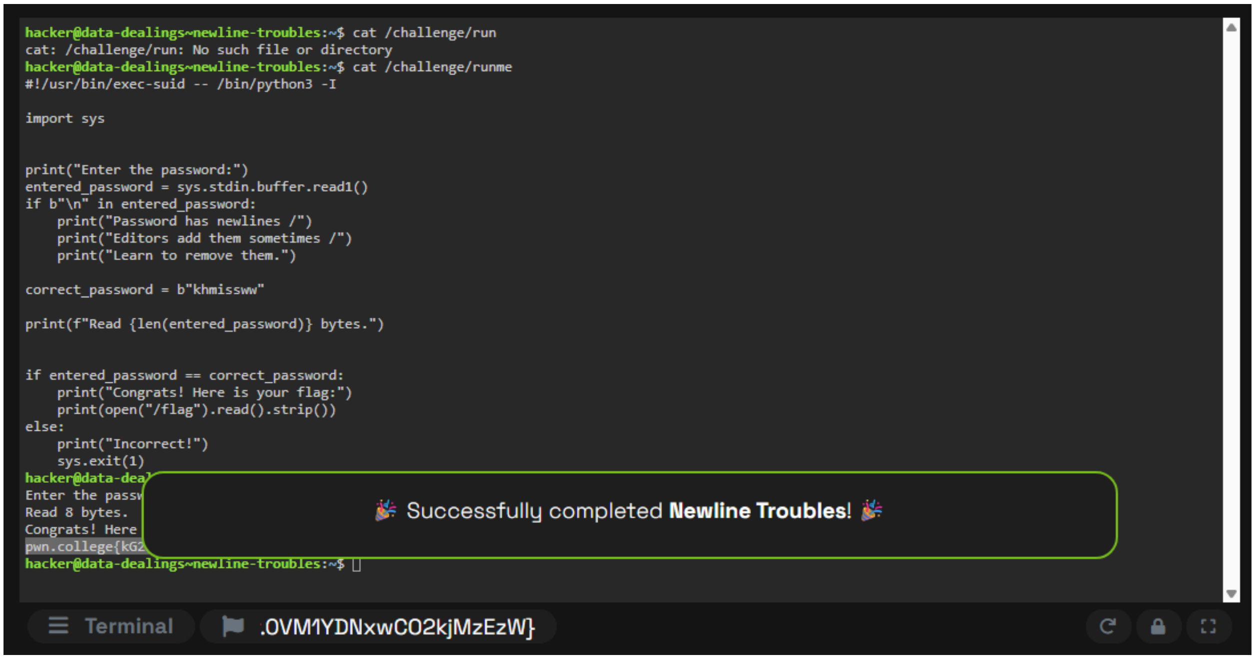
To get the flag, I read the source code of **/challenge/runme** using **cat** and found the **hard-coded** password **gxdgjdei**. Then, I piped this password directly into the program using **echo -n "gxdgjdei" | /challenge/runme**, which **bypassed** the check and output the flag. This worked because the program compares the input directly to this byte string.

- **Newline troubles**



```
hacker@data-dealings~newline-troubles:~$ cat /challenge/run
cat: /challenge/run: No such file or directory
hacker@data-dealings~newline-troubles:~$ cat /challenge/runme
#!/usr/bin/exec-suid -- /bin/python3 -I

import sys


print("Enter the password:")
entered_password = sys.stdin.buffer.read1()
if b"\n" in entered_password:
    print("Password has newlines /")
    print("Editors add them sometimes /")
    print("Learn to remove them.")

correct_password = b"khmissww"

print(f"Read {len(entered_password)} bytes.")


if entered_password == correct_password:
    print("Congrats! Here is your flag:")
    print(open("/flag").read().strip())
else:
    print("Incorrect!")
    sys.exit(1)
hacker@data-dealings~newline-troubles:~$ echo -n "khmissww" | /challenge/runme
Enter the password:
Read 8 bytes.
Congrats! Here is your flag:
pwn.college{kG2hOPMhi6c37YiA2gOp6fQRsOr.0VM1YDNxwCO2kjMzEzW}
hacker@data-dealings~newline-troubles:~$ 
```

This screenshot is the same as the following screenshot before successful capture.

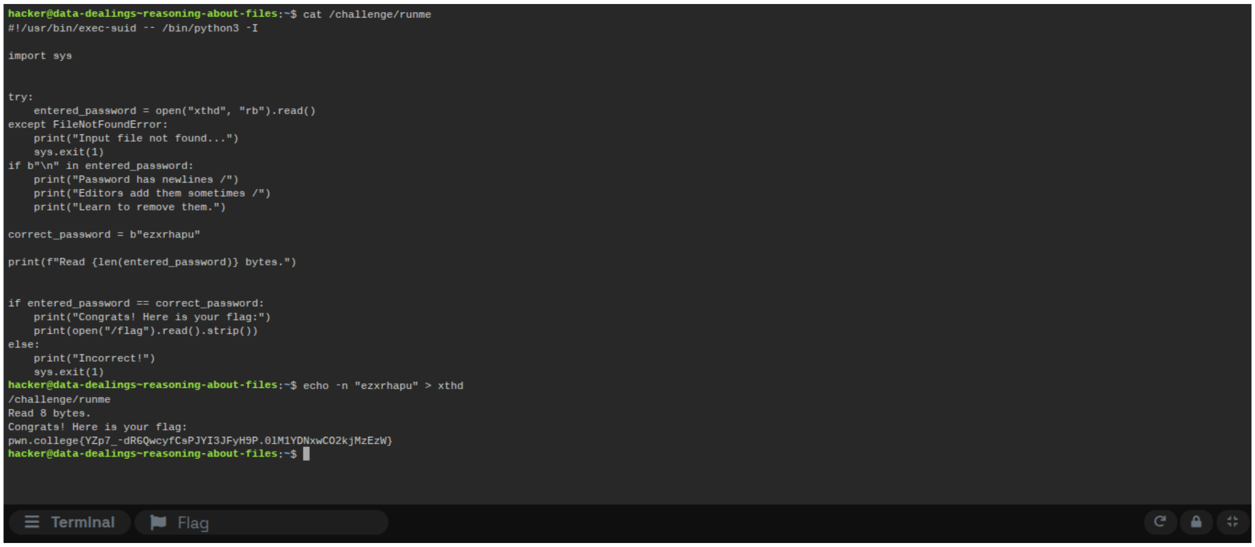


```
hacker@data-dealings~newline-troubles:~$ cat /challenge/run
cat: /challenge/run: No such file or directory
hacker@data-dealings~newline-troubles:~$ cat /challenge/runme
#!/usr/bin/exec-suid -- /bin/python3 -I

import sys


print("Enter the password:")
entered_password = sys.stdin.buffer.read1()
if b"\n" in entered_password:
    print("Password has newlines /")
    print("Editors add them sometimes /")
    print("Learn to remove them.")

correct_password = b"khmissww"

print(f"Read {len(entered_password)} bytes.")


if entered_password == correct_password:
    print("Congrats! Here is your flag:")
    print(open("/flag").read().strip())
else:
    print("Incorrect!")
    sys.exit(1)
hacker@data-dea
Enter the passw
Read 8 bytes.
Congrats! Here
pwn.college{kG2
hacker@data-dealings~newline-troubles:~$ 
```

🎉 Successfully completed **Newline Troubles**! 🎉

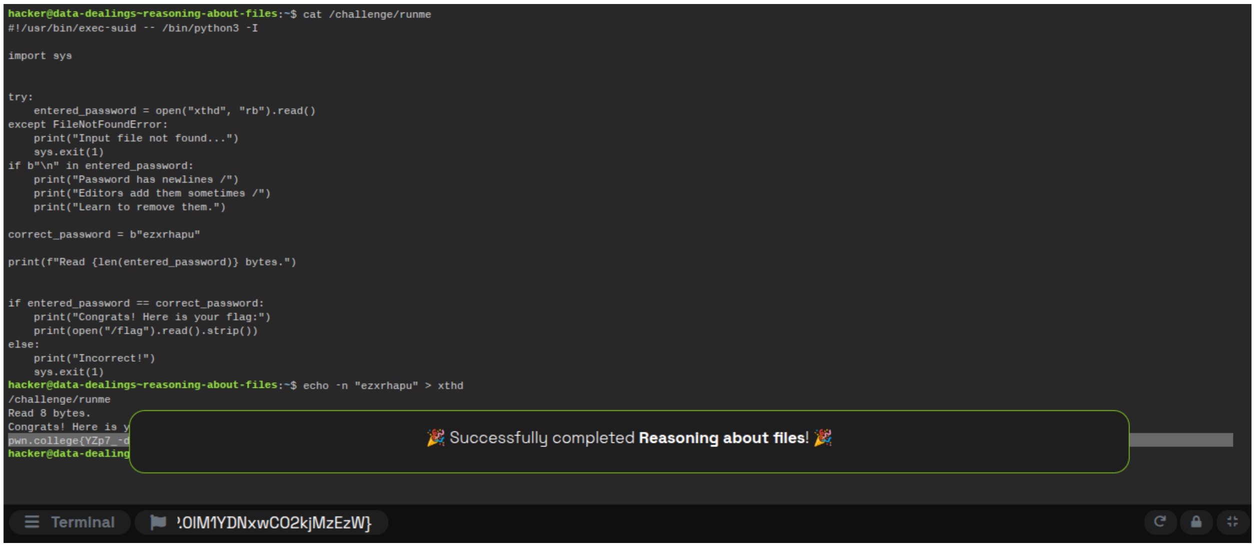
At first I made a **mistake** by writing **cat /challenge/run** instead of **cat /challenge/runme**. Once I ran the right command to retrieve the flag, I read the source code of **/challenge/runme** using **cat** and found the **hard-coded** password **"khmissww"**. Since the program **rejects** input with **newlines**, I used **echo -n "khmissww"** to send the password **without** a **newline character**, which allowed the program to match the password and output the flag.

- **Reasoning about files**



```
hacker@data-dealings~reasoning-about-files:~$ cat /challenge/runme
#!/usr/bin/exec-suid -- /bin/python3 -I

import sys


try:
    entered_password = open("xthd", "rb").read()
except FileNotFoundError:
    print("Input file not found...")
    sys.exit(1)
if b"\n" in entered_password:
    print("Password has newlines /")
    print("Editors add them sometimes /")
    print("Learn to remove them.")

correct_password = b"ezxrhapu"

print(f"Read {len(entered_password)} bytes.")


if entered_password == correct_password:
    print("Congrats! Here is your flag:")
    print(open("/flag").read().strip())
else:
    print("Incorrect!")
    sys.exit(1)
hacker@data-dealings~reasoning-about-files:~$ echo -n "ezxrhapu" > xthd
/challenge/runme
Read 8 bytes.
Congrats! Here is your flag:
pwn.college{YZp7_-dR6QwcyfCsPJYI3JFyH9P.0lM1YDNxwCO2kjMzEzW}
hacker@data-dealings~reasoning-about-files:~$ 
```

This screenshot is the same as the following screenshot before successful capture.

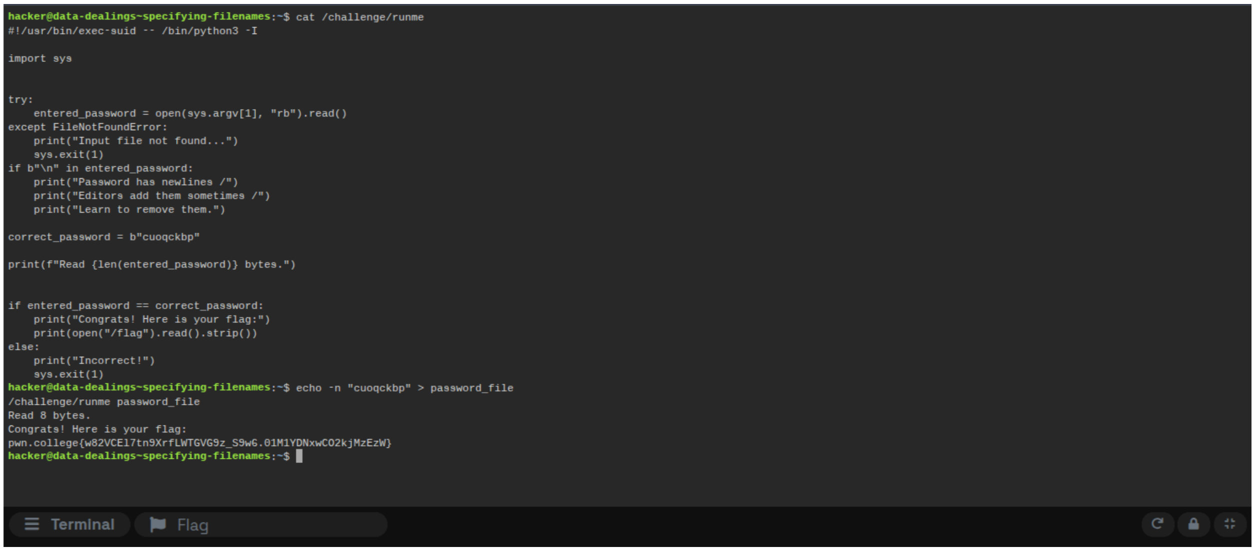


```
hacker@data-dealings~reasoning-about-files:~$ cat /challenge/runme
#!/usr/bin/exec-suid -- /bin/python3 -I

import sys


try:
    entered_password = open("xthd", "rb").read()
except FileNotFoundError:
    print("Input file not found...")
    sys.exit(1)
if b"\n" in entered_password:
    print("Password has newlines /")
    print("Editors add them sometimes /")
    print("Learn to remove them.")

correct_password = b"ezxrhapu"

print(f"Read {len(entered_password)} bytes.")


if entered_password == correct_password:
    print("Congrats! Here is your flag:")
    print(open("/flag").read().strip())
else:
    print("Incorrect!")
    sys.exit(1)
hacker@data-dealings~reasoning-about-files:~$ echo -n "ezxrhapu" > xthd
/challenge/runme
Read 8 bytes.
Congrats! Here is y
pwn.college{YZp7_-d
hacker@data-dealing
```

🎉 Successfully completed **Reasoning about files**! 🎉

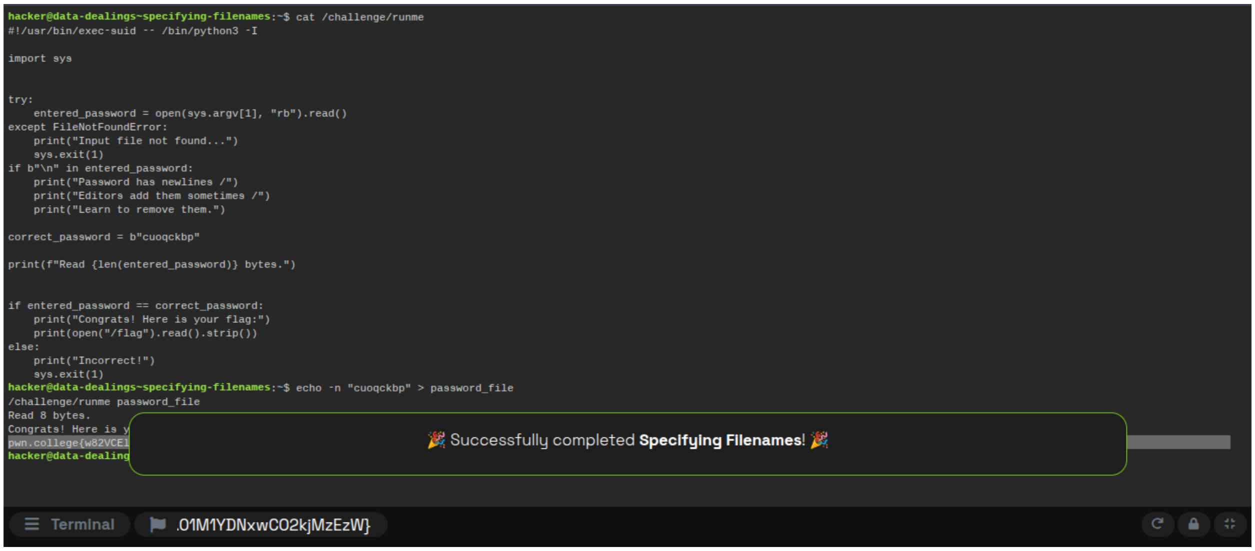
In this challenge, the program read the password from a file named **xthd**. I first viewed the source code using cat /challenge/runme to identify the correct password **(ezxrhapu)**. I then created the file **xthd** without a newline using **echo -n "ezxrhapu" > xthd**. Finally, I executed the program, which read the file and output the flag. This demonstrates how programs can rely on external files for input, which is common in system configurations. Ensuring the file content matches exactly (no extra newlines) is critical for success.

- **Specifying filenames**



```
hacker@data-dealings~specifying-filenames:~$ cat /challenge/runme
#!/usr/bin/exec-suid -- /bin/python3 -I

import sys


try:
    entered_password = open(sys.argv[1], "rb").read()
except FileNotFoundError:
    print("Input file not found...")
    sys.exit(1)
if b"\n" in entered_password:
    print("Password has newlines /")
    print("Editors add them sometimes /")
    print("Learn to remove them.")

correct_password = b"cuoqckbp"

print(f"Read {len(entered_password)} bytes.")


if entered_password == correct_password:
    print("Congrats! Here is your flag:")
    print(open("/flag").read().strip())
else:
    print("Incorrect!")
    sys.exit(1)
hacker@data-dealings~specifying-filenames:~$ echo -n "cuoqckbp" > password_file
/challenge/runme password_file
Read 8 bytes.
Congrats! Here is your flag:
pwn.college{w82VCEl7tn9XrfLWTGVG9z_S9w6.01M1YDNxwCO2kjMzEzW}
hacker@data-dealings~specifying-filenames:~$
```

This screenshot is the same as the following screenshot before successful capture.

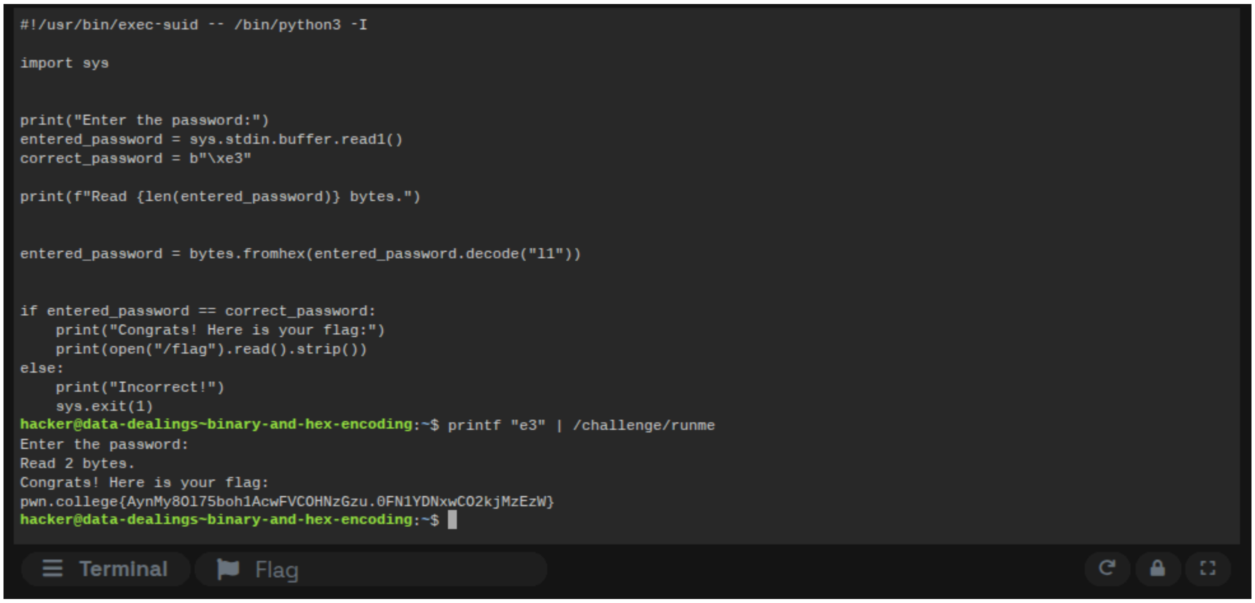


```
hacker@data-dealings~specifying-filenames:~$ cat /challenge/runme
#!/usr/bin/exec-suid -- /bin/python3 -I

import sys


try:
    entered_password = open(sys.argv[1], "rb").read()
except FileNotFoundError:
    print("Input file not found...")
    sys.exit(1)
if b"\n" in entered_password:
    print("Password has newlines /")
    print("Editors add them sometimes /")
    print("Learn to remove them.")

correct_password = b"cuoqckbp"

print(f"Read {len(entered_password)} bytes.")


if entered_password == correct_password:
    print("Congrats! Here is your flag:")
    print(open("/flag").read().strip())
else:
    print("Incorrect!")
    sys.exit(1)
hacker@data-dealings~specifying-filenames:~$ echo -n "cuoqckbp" > password_file
/challenge/runme password_file
Read 8 bytes.
Congrats! Here is y
pwn.college{w82VCEl
hacker@data-dealing
```

🎉 Successfully completed **Specifying Filenames**! 🎉

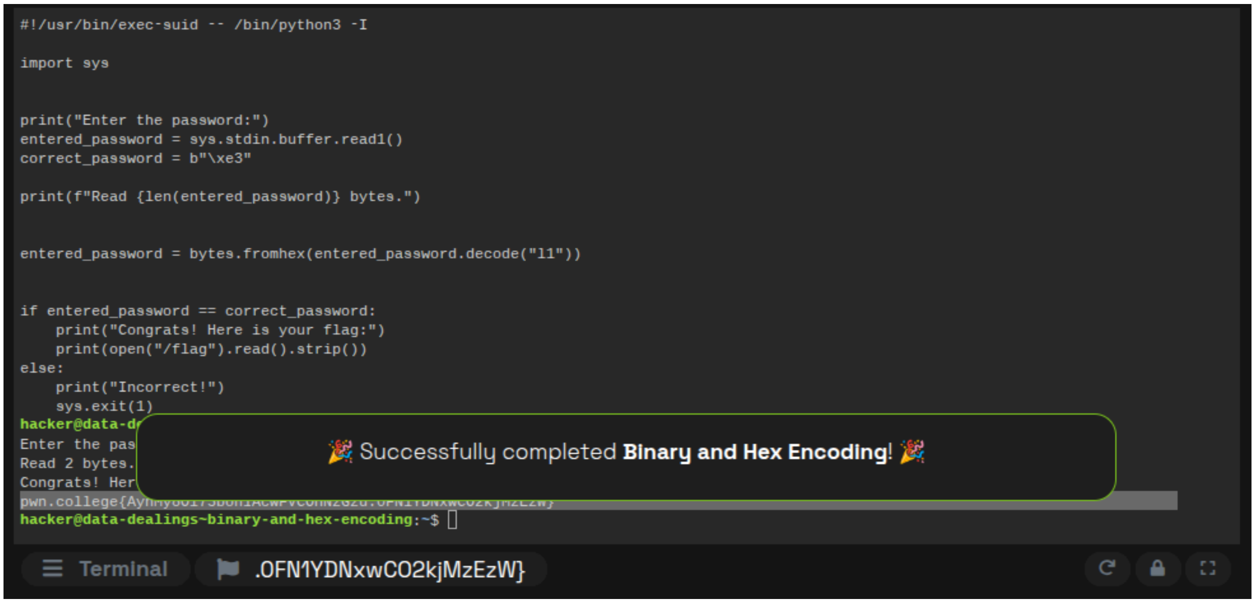
To retrieve the flag, I examined the source code of **/challenge/runme** and found that it reads the password from a file specified as a command-line argument. The correct password was **"cuoqckbp"**. I created a file with this password using **echo -n "cuoqckbp" > password_file** to avoid any newline characters, then executed the program with the file as an argument: **/challenge/runme password_file**. This allowed the program to **read** the **correct password** and **output** the flag. Command-line arguments provide a flexible way to pass inputs to programs. Proper error handling (for missing files) is essential in such designs.

- **Binary and hex encoding**



```
#!/usr/bin/exec-suid -- /bin/python3 -I

import sys


print("Enter the password:")
entered_password = sys.stdin.buffer.read1()
correct_password = b"\xe3"

print(f"Read {len(entered_password)} bytes.")


entered_password = bytes.fromhex(entered_password.decode("l1"))


if entered_password == correct_password:
    print("Congrats! Here is your flag:")
    print(open("/flag").read().strip())
else:
    print("Incorrect!")
    sys.exit(1)
hacker@data-dealings~binary-and-hex-encoding:~$ printf "e3" | /challenge/runme
Enter the password:
Read 2 bytes.
Congrats! Here is your flag:
pwn.college{AynMy80175boh1AcwFVCOHNzGzu.0FN1YDNxwCO2kjMzEzW}
hacker@data-dealings~binary-and-hex-encoding:~$ 
```

This screenshot is the same as the following screenshot before successful capture.

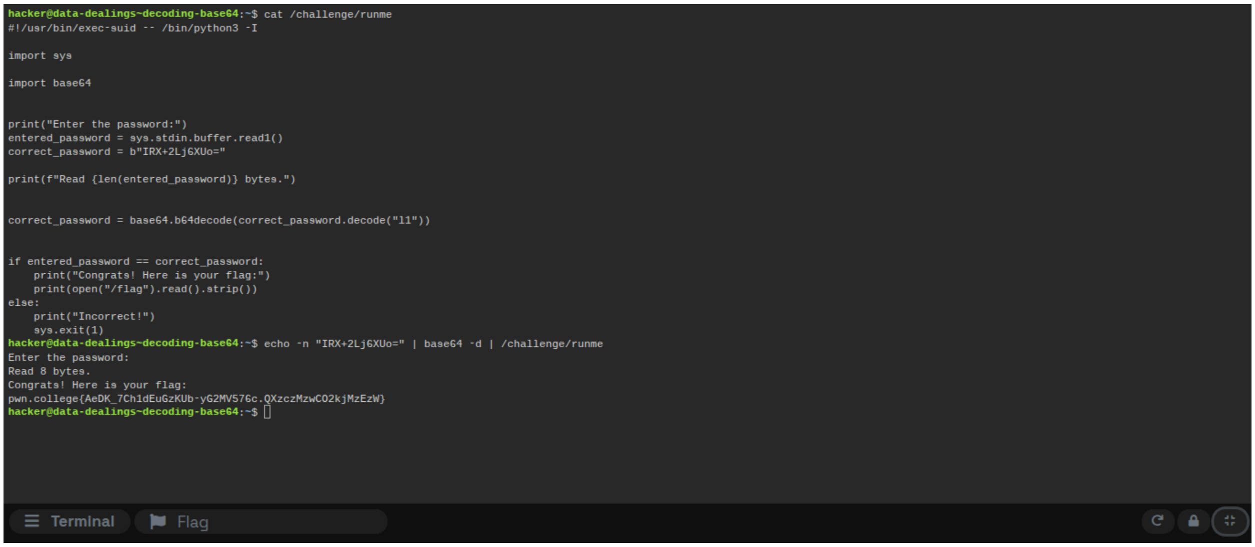


```
#!/usr/bin/exec-suid -- /bin/python3 -I

import sys


print("Enter the password:")
entered_password = sys.stdin.buffer.read1()
correct_password = b"\xe3"

print(f"Read {len(entered_password)} bytes.")


entered_password = bytes.fromhex(entered_password.decode("l1"))


if entered_password == correct_password:
    print("Congrats! Here is your flag:")
    print(open("/flag").read().strip())
else:
    print("Incorrect!")
    sys.exit(1)
hacker@data-d
Enter the pas
Read 2 bytes.
Congrats! Her
pwn.college{Ay
hacker@data-dealings~binary-and-hex-encoding:~$ 
```

🎉 Successfully completed **Binary and Hex Encoding**! 🎉

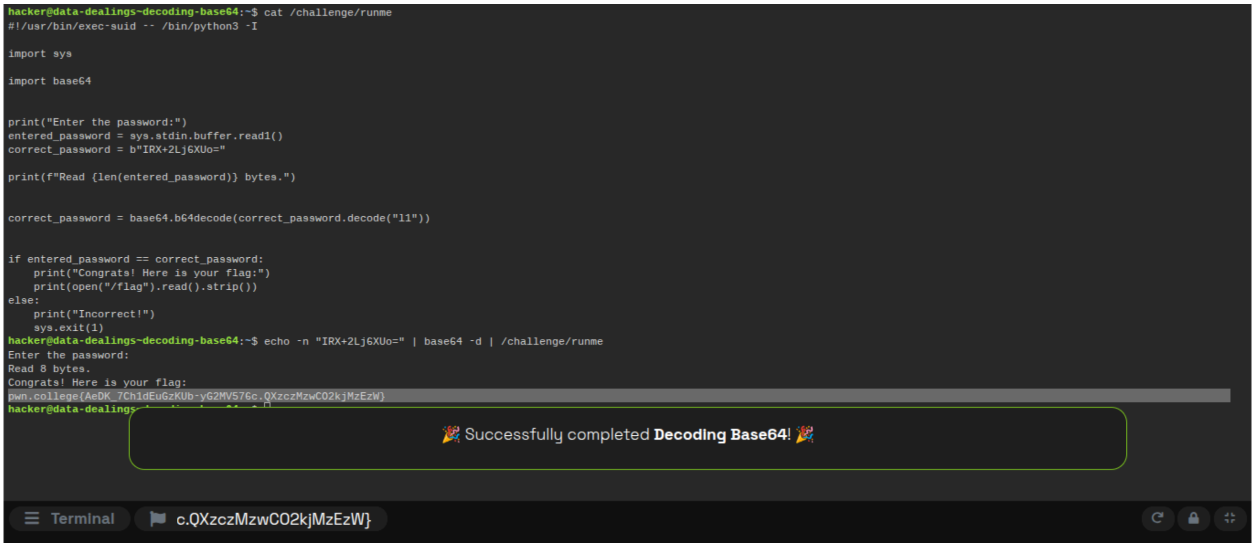
To retrieve the flag, I analyzed the source code and found that the correct password is the byte **\xe3**, but the program **expects** its **hexadecimal representation** as **input**. I used **printf "e3" | /challenge/runme** to send the hex string **"e3"** without any newlines, which the program converted to bytes and matched against the correct password, resulting in the flag being displayed.

- ### Decoding Base64



```
hacker@data-dealings~decoding-base64:~$ cat /challenge/runme
#!/usr/bin/exec-suid -- /bin/python3 -I

import sys

import base64


print("Enter the password:")
entered_password = sys.stdin.buffer.read1()
correct_password = b"IRX+2Lj6XUo="

print(f"Read {len(entered_password)} bytes.")


correct_password = base64.b64decode(correct_password.decode("l1"))


if entered_password == correct_password:
    print("Congrats! Here is your flag:")
    print(open("/flag").read().strip())
else:
    print("Incorrect!")
    sys.exit(1)
hacker@data-dealings~decoding-base64:~$ echo -n "IRX+2Lj6XUo=" | base64 -d | /challenge/runme
Enter the password:
Read 8 bytes.
Congrats! Here is your flag:
pwn.college{AeDK_7Ch1dEuGzKUb-yG2MV576c.QXzczMzwCO2kjMzEzW}
hacker@data-dealings~decoding-base64:~$ 
```

This screenshot is the same as the following screenshot before successful capture.

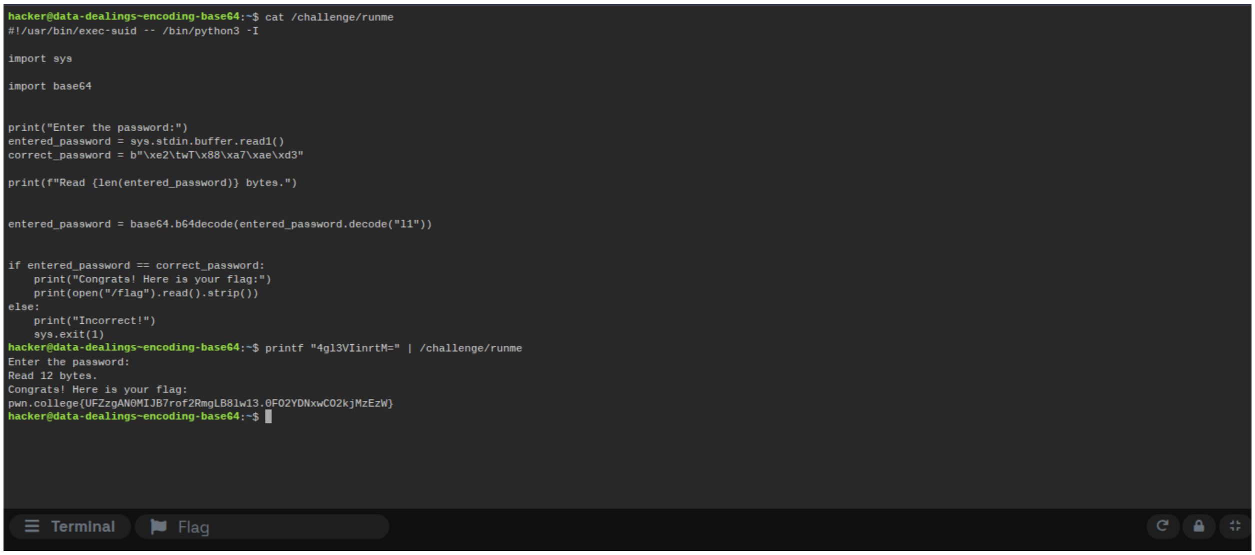


```
hacker@data-dealings~decoding-base64:~$ cat /challenge/runme
#!/usr/bin/exec-suid -- /bin/python3 -I

import sys

import base64


print("Enter the password:")
entered_password = sys.stdin.buffer.read1()
correct_password = b"IRX+2Lj6XUo="

print(f"Read {len(entered_password)} bytes.")


correct_password = base64.b64decode(correct_password.decode("l1"))


if entered_password == correct_password:
    print("Congrats! Here is your flag:")
    print(open("/flag").read().strip())
else:
    print("Incorrect!")
    sys.exit(1)
hacker@data-dealings~decoding-base64:~$ echo -n "IRX+2Lj6XUo=" | base64 -d | /challenge/runme
Enter the password:
Read 8 bytes.
Congrats! Here is your flag:
pwn.college{AeDK_7Ch1dEuGzKUb-yG2MV576c.QXzczMzwCO2kjMzEzW}
hacker@data-dealings
```

🎉 Successfully completed **Decoding Base64**! 🎉

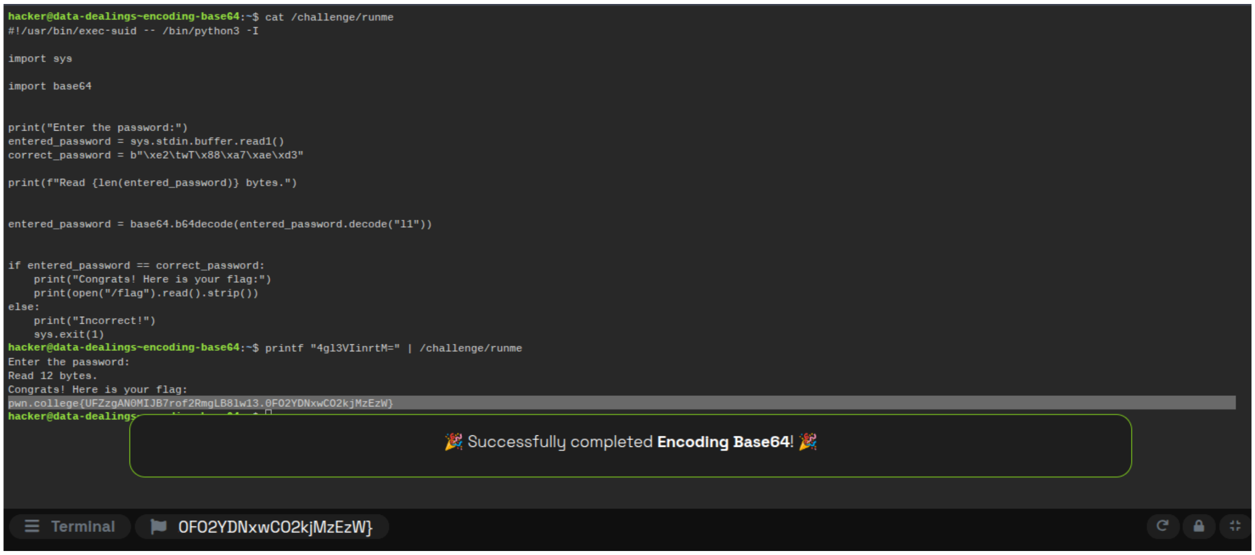
To retrieve the flag, I decoded the **base64 string "IRX+2Lj6XUo="** using the **base64 -d command** and **piped** the **raw bytes** to the program. This ensured that the input **matched** the decoded password, causing the program to output the flag.

- **Encoding Base64**



```
hacker@data-dealings~encoding-base64:~$ cat /challenge/runme
#!/usr/bin/exec-suid -- /bin/python3 -I

import sys

import base64


print("Enter the password:")
entered_password = sys.stdin.buffer.read1()
correct_password = b"\xe2\twT\x88\xa7\xae\xd3"

print(f"Read {len(entered_password)} bytes.")


entered_password = base64.b64decode(entered_password.decode("l1"))


if entered_password == correct_password:
    print("Congrats! Here is your flag:")
    print(open("/flag").read().strip())
else:
    print("Incorrect!")
    sys.exit(1)
hacker@data-dealings~encoding-base64:~$ printf "4gl3VIinrtM=" | /challenge/runme
Enter the password:
Read 12 bytes.
Congrats! Here is your flag:
pwn.college{UFZzgAN0MIJB7rof2RmgLB8lw13.0FO2YDNxwCO2kjMzEzW}
hacker@data-dealings~encoding-base64:~$ 
```

This screenshot is the same as the following screenshot before successful capture.



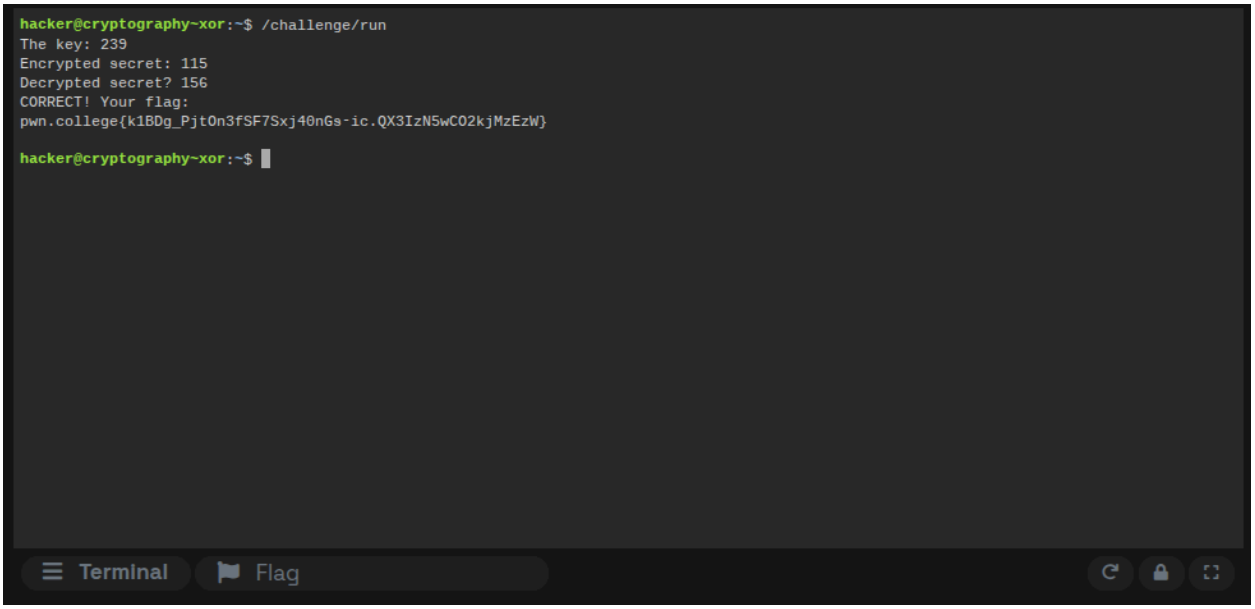
```
hacker@data-dealings~encoding-base64:~$ cat /challenge/runme
#!/usr/bin/exec-suid -- /bin/python3 -I

import sys

import base64


print("Enter the password:")
entered_password = sys.stdin.buffer.read1()
correct_password = b"\xe2\twT\x88\xa7\xae\xd3"

print(f"Read {len(entered_password)} bytes.")


entered_password = base64.b64decode(entered_password.decode("l1"))


if entered_password == correct_password:
    print("Congrats! Here is your flag:")
    print(open("/flag").read().strip())
else:
    print("Incorrect!")
    sys.exit(1)
hacker@data-dealings~encoding-base64:~$ printf "4gl3VIinrtM=" | /challenge/runme
Enter the password:
Read 12 bytes.
Congrats! Here is your flag:
pwn.college{UFZzgAN0MIJB7rof2RmgLB8lw13.0FO2YDNxwCO2kjMzEzW}
hacker@data-dealings
```

🎉 Successfully completed **Encoding Base64**! 🎉

To retrieve the flag, I **base64** encoded the correct password **bytes b"\xe2\twT\x88\xa7\xae\xd3"** to get the **string 4gl3VIinrtM=**. I then piped this string to the program using **printf "4gl3VIinrtM=" | /challenge/runme**, which decoded the input and matched it against the correct password, resulting in the flag being displayed.

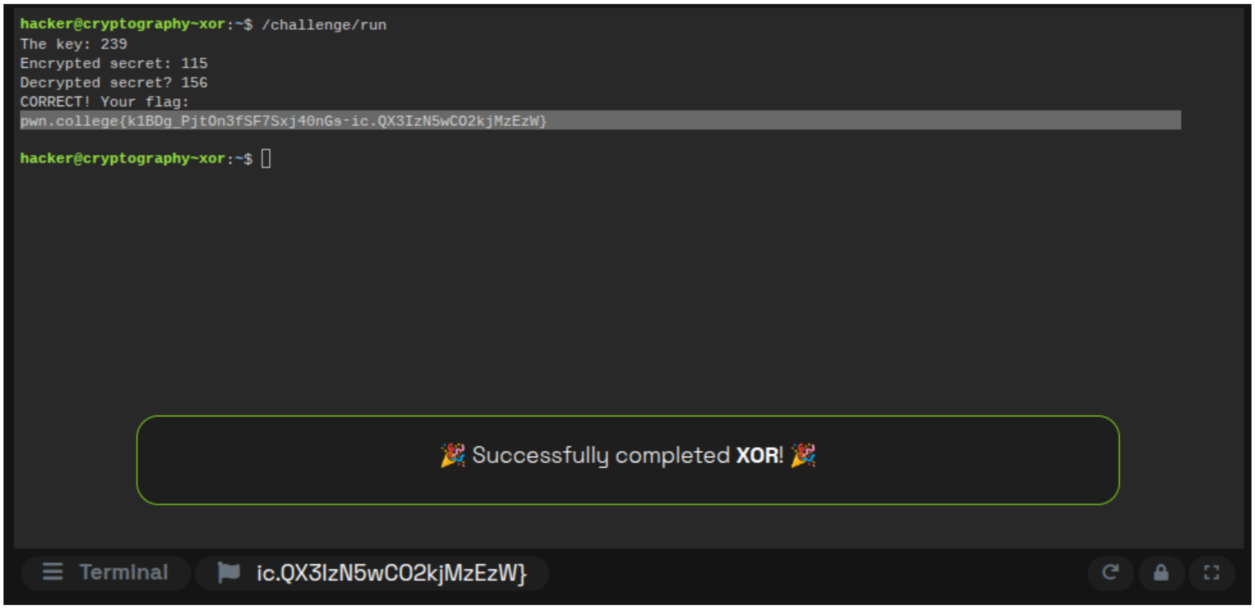
Cryptography: https://pwn.college/intro-to-cybersecurity/cryptography/

- **XOR**



```
hacker@cryptography~xor:~$ /challenge/run
The key: 239
Encrypted secret: 115
Decrypted secret? 156
CORRECT! Your flag:
pwn.college{k1BDg_PjtOn3fSF7Sxj40nGs-ic.QX3IzN5wCO2kjMzEzW}

hacker@cryptography~xor:~$ 
```

This screenshot is the same as the following screenshot before successful capture.

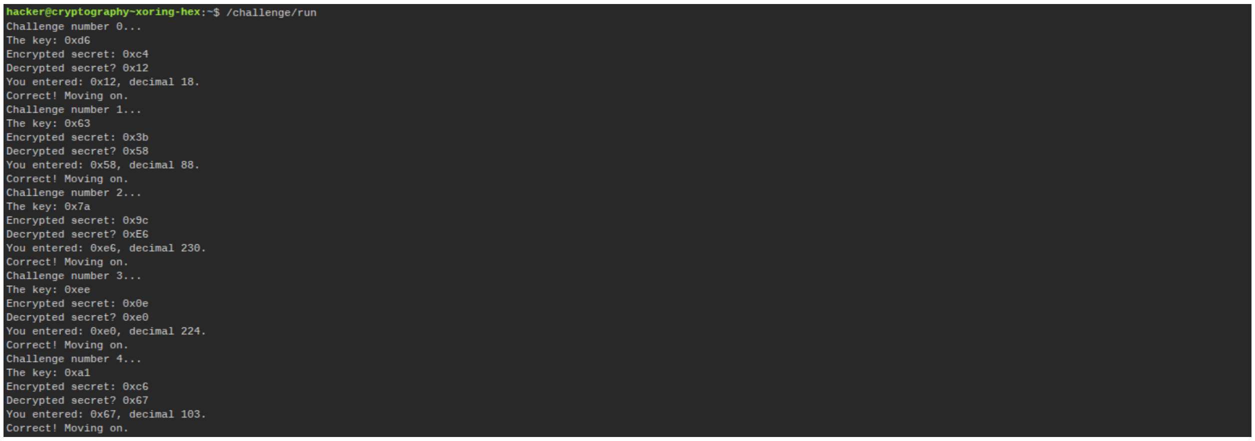


```
hacker@cryptography~xor:~$ /challenge/run
The key: 239
Encrypted secret: 115
Decrypted secret? 156
CORRECT! Your flag:
pwn.college{k1BDg_PjtOn3fSF7Sxj40nGs-ic.QX3IzN5wCO2kjMzEzW}

hacker@cryptography~xor:~$ 
```

🎉 Successfully completed **XOR**! 🎉

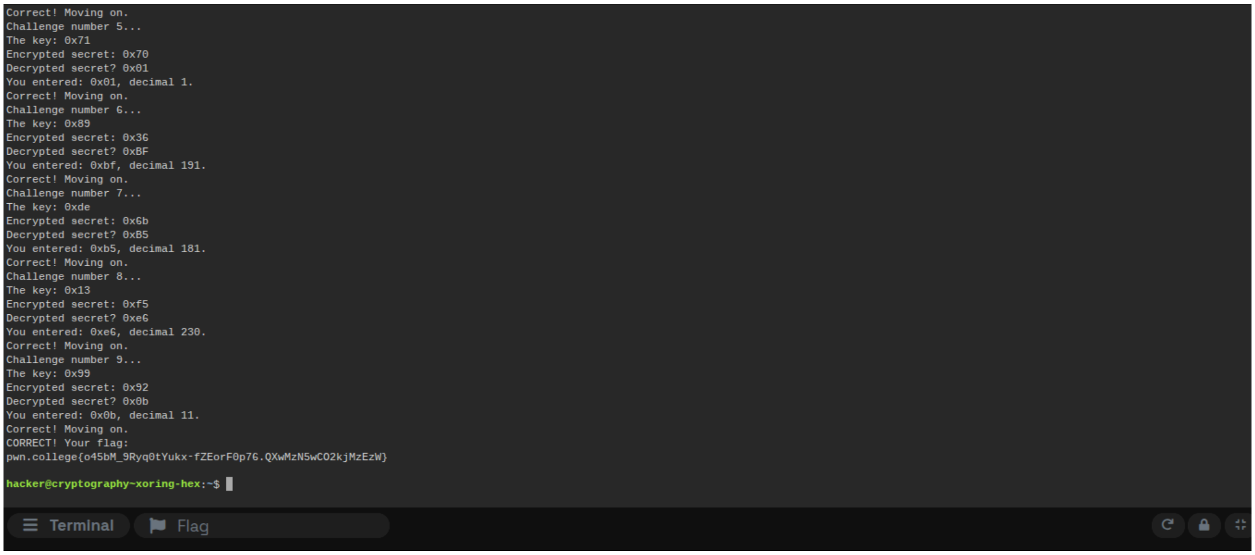
I retrieved the flag by running the challenge program in the terminal, which provided me with a **key (239)** and an **encrypted secret (115)**. Knowing that **XOR** is **self-inverse**, I **decrypted** the **secret** by calculating the **XOR** of the **encrypted secret** and the **key: 115 XOR 239 = 156**. After entering **156** as the **decrypted secret**, the program verified my answer and displayed the flag, proving I successfully solved the **XOR** cryptography challenge.

- **XORing Hex**



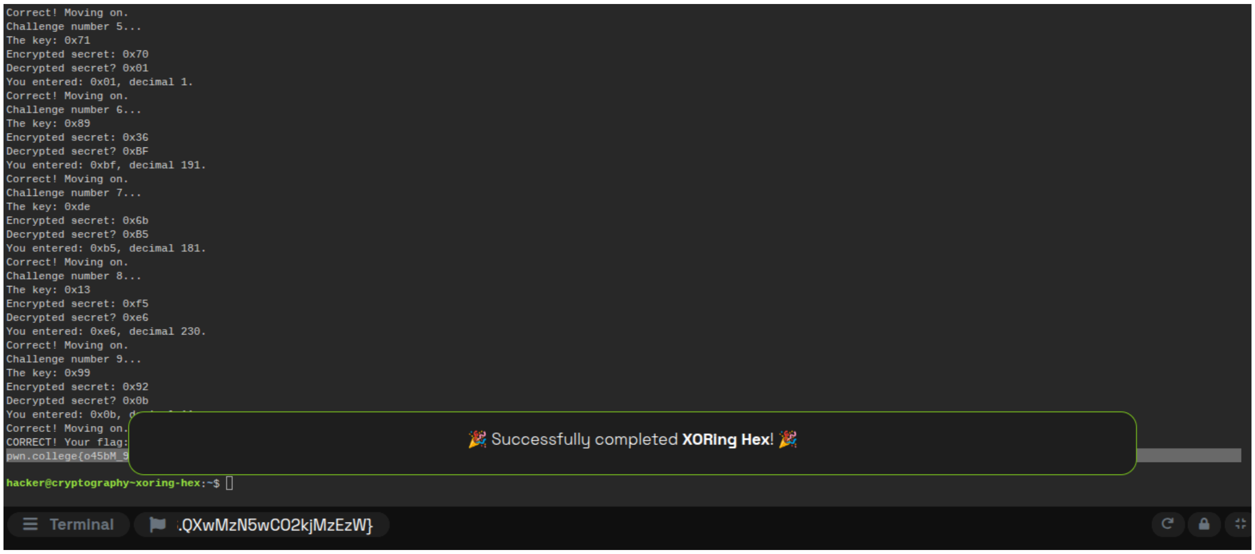
```
hacker@cryptography~xoring-hex:~$ /challenge/run
Challenge number 0...
The key: 0xd6
Encrypted secret: 0xc4
Decrypted secret? 0x12
You entered: 0x12, decimal 18.
Correct! Moving on.
Challenge number 1...
The key: 0x63
Encrypted secret: 0x3b
Decrypted secret? 0x58
You entered: 0x58, decimal 88.
Correct! Moving on.
Challenge number 2...
The key: 0x7a
Encrypted secret: 0x9c
Decrypted secret? 0xE6
You entered: 0xe6, decimal 230.
Correct! Moving on.
Challenge number 3...
The key: 0xee
Encrypted secret: 0x0e
Decrypted secret? 0xe0
You entered: 0xe0, decimal 224.
Correct! Moving on.
Challenge number 4...
The key: 0xa1
Encrypted secret: 0xc6
Decrypted secret? 0x67
You entered: 0x67, decimal 103.
Correct! Moving on.
```

Continues into the next screenshot.



```
Correct! Moving on.
Challenge number 5...
The key: 0x71
Encrypted secret: 0x70
Decrypted secret? 0x01
You entered: 0x01, decimal 1.
Correct! Moving on.
Challenge number 6...
The key: 0x89
Encrypted secret: 0x36
Decrypted secret? 0xBF
You entered: 0xbf, decimal 191.
Correct! Moving on.
Challenge number 7...
The key: 0xde
Encrypted secret: 0x6b
Decrypted secret? 0xB5
You entered: 0xb5, decimal 181.
Correct! Moving on.
Challenge number 8...
The key: 0x13
Encrypted secret: 0xf5
Decrypted secret? 0xe6
You entered: 0xe6, decimal 230.
Correct! Moving on.
Challenge number 9...
The key: 0x99
Encrypted secret: 0x92
Decrypted secret? 0x0b
You entered: 0x0b, decimal 11.
Correct! Moving on.
CORRECT! Your flag:
pwn.college{o45bM_9Ryq0tYukx-fZEorF0p76.QXwMzN5wCO2kjMzEzW}

hacker@cryptography~xoring-hex:~$
```

This screenshot is the same as the following screenshot before successful capture.

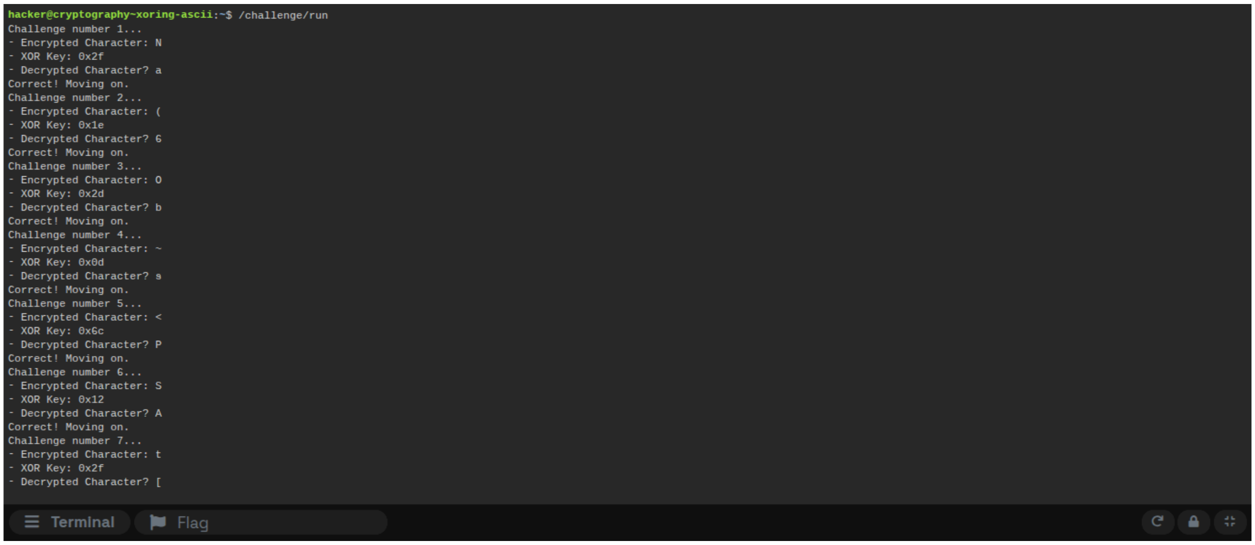


```
Correct! Moving on.
Challenge number 5...
The key: 0x71
Encrypted secret: 0x70
Decrypted secret? 0x01
You entered: 0x01, decimal 1.
Correct! Moving on.
Challenge number 6...
The key: 0x89
Encrypted secret: 0x36
Decrypted secret? 0xBF
You entered: 0xbf, decimal 191.
Correct! Moving on.
Challenge number 7...
The key: 0xde
Encrypted secret: 0x6b
Decrypted secret? 0xB5
You entered: 0xb5, decimal 181.
Correct! Moving on.
Challenge number 8...
The key: 0x13
Encrypted secret: 0xf5
Decrypted secret? 0xe6
You entered: 0xe6, decimal 230.
Correct! Moving on.
Challenge number 9...
The key: 0x99
Encrypted secret: 0x92
Decrypted secret? 0x0b
You entered: 0x0b, d
Correct! Moving on.
CORRECT! Your flag:
pwn.college{o45bM_9

hacker@cryptography~xoring-hex:~$ 
```

🎉 Successfully completed **XORing Hex**! 🎉

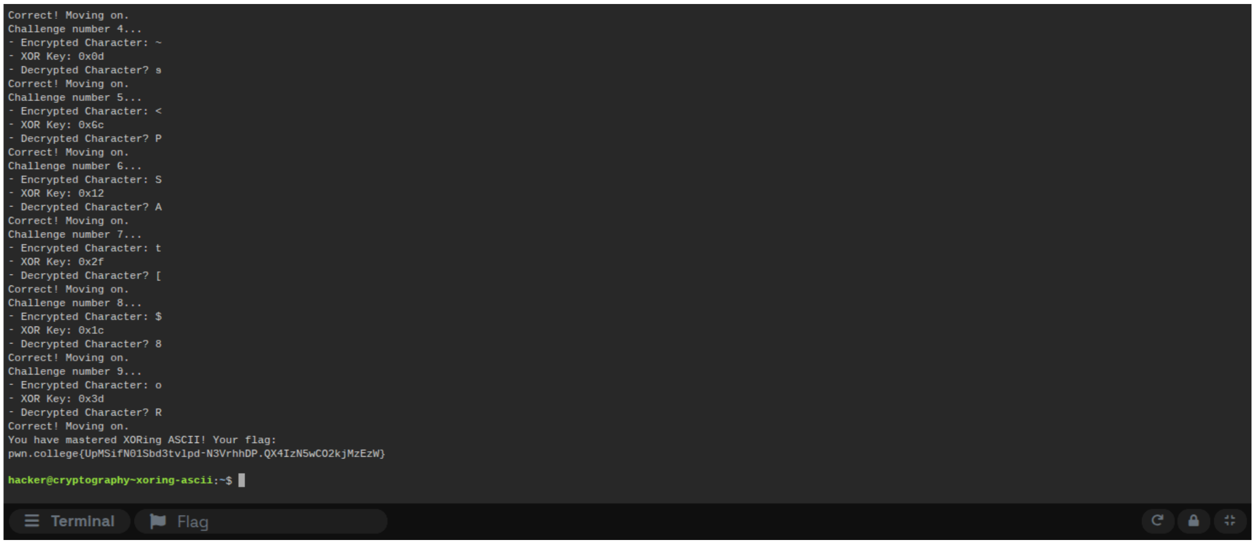
I retrieved the flag by successfully completing a series of **XOR** challenges in the terminal. Each challenge provided a **key** and an **encrypted secret** in hexadecimal, and I computed the **decrypted secret** by **XORing** the two values together. After **correctly** entering all **ten decrypted secrets**, the program verified my answers and awarded me the flag, which serves as proof of my understanding of **hexadecimal XOR** operations in cryptography.

- **XORing ASCII**



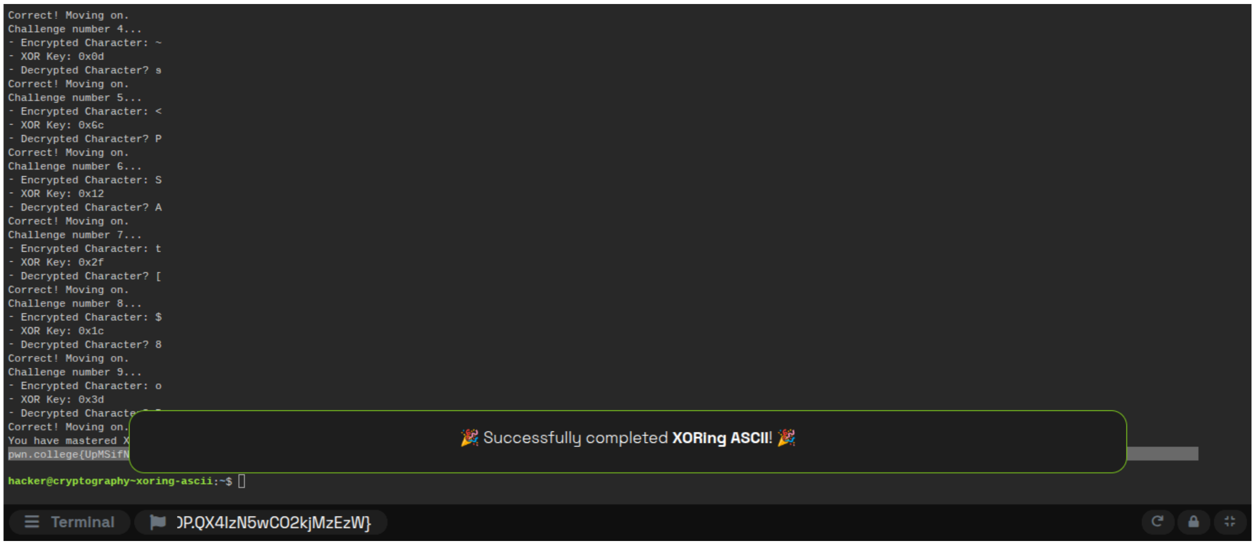
```
hacker@cryptography~xoring-ascii:~$ /challenge/run
Challenge number 1...
- Encrypted Character: N
- XOR Key: 0x2f
- Decrypted Character? a
Correct! Moving on.
Challenge number 2...
- Encrypted Character: (
- XOR Key: 0x1e
- Decrypted Character? 6
Correct! Moving on.
Challenge number 3...
- Encrypted Character: O
- XOR Key: 0x2d
- Decrypted Character? b
Correct! Moving on.
Challenge number 4...
- Encrypted Character: ~
- XOR Key: 0x0d
- Decrypted Character? s
Correct! Moving on.
Challenge number 5...
- Encrypted Character: <
- XOR Key: 0x6c
- Decrypted Character? P
Correct! Moving on.
Challenge number 6...
- Encrypted Character: S
- XOR Key: 0x12
- Decrypted Character? A
Correct! Moving on.
Challenge number 7...
- Encrypted Character: t
- XOR Key: 0x2f
- Decrypted Character? [
```

Continues into the next screenshot.



```
Correct! Moving on.
Challenge number 4...
- Encrypted Character: ~
- XOR Key: 0x0d
- Decrypted Character? s
Correct! Moving on.
Challenge number 5...
- Encrypted Character: <
- XOR Key: 0x6c
- Decrypted Character? P
Correct! Moving on.
Challenge number 6...
- Encrypted Character: S
- XOR Key: 0x12
- Decrypted Character? A
Correct! Moving on.
Challenge number 7...
- Encrypted Character: t
- XOR Key: 0x2f
- Decrypted Character? [
Correct! Moving on.
Challenge number 8...
- Encrypted Character: $
- XOR Key: 0x1c
- Decrypted Character? 8
Correct! Moving on.
Challenge number 9...
- Encrypted Character: o
- XOR Key: 0x3d
- Decrypted Character? R
Correct! Moving on.
You have mastered XORing ASCII! Your flag:
pwn.college{UpMSifN01Sbd3tvlpd-N3VrhhDP.QX4IzN5wCO2kjMzEzW}

hacker@cryptography~xoring-ascii:~$
```

This screenshot is the same as the following screenshot before successful capture.

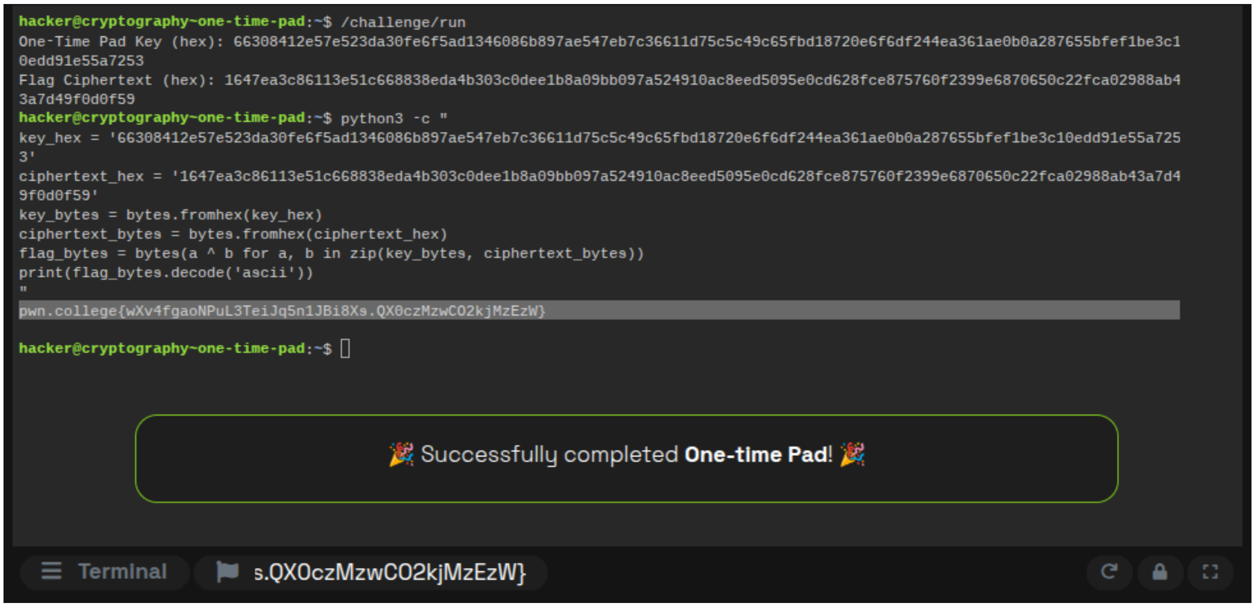


```
Correct! Moving on.
Challenge number 4...
- Encrypted Character: ~
- XOR Key: 0x0d
- Decrypted Character? s
Correct! Moving on.
Challenge number 5...
- Encrypted Character: <
- XOR Key: 0x6c
- Decrypted Character? P
Correct! Moving on.
Challenge number 6...
- Encrypted Character: S
- XOR Key: 0x12
- Decrypted Character? A
Correct! Moving on.
Challenge number 7...
- Encrypted Character: t
- XOR Key: 0x2f
- Decrypted Character? [
Correct! Moving on.
Challenge number 8...
- Encrypted Character: $
- XOR Key: 0x1c
- Decrypted Character? 8
Correct! Moving on.
Challenge number 9...
- Encrypted Character: o
- XOR Key: 0x3d
- Decrypted Characte
Correct! Moving on.
You have mastered X
pwn.college{UpMSifN

hacker@cryptography~xoring-ascii:~$
```

🎉 Successfully completed **XORing ASCII**! 🎉

I retrieved the flag by successfully completing a series of **XOR** challenges that involved decrypting **ASCII** characters. For each challenge, I was given an encrypted character and a hexadecimal key. I used the **XOR** operation between the **ASCII** value of the encrypted character and the key to find the decrypted character. After **correctly** entering all **nine decrypted characters**, the program confirmed my mastery of **XOR** with **ASCII** and awarded me the flag as proof of my understanding.

- **One-time Pad**

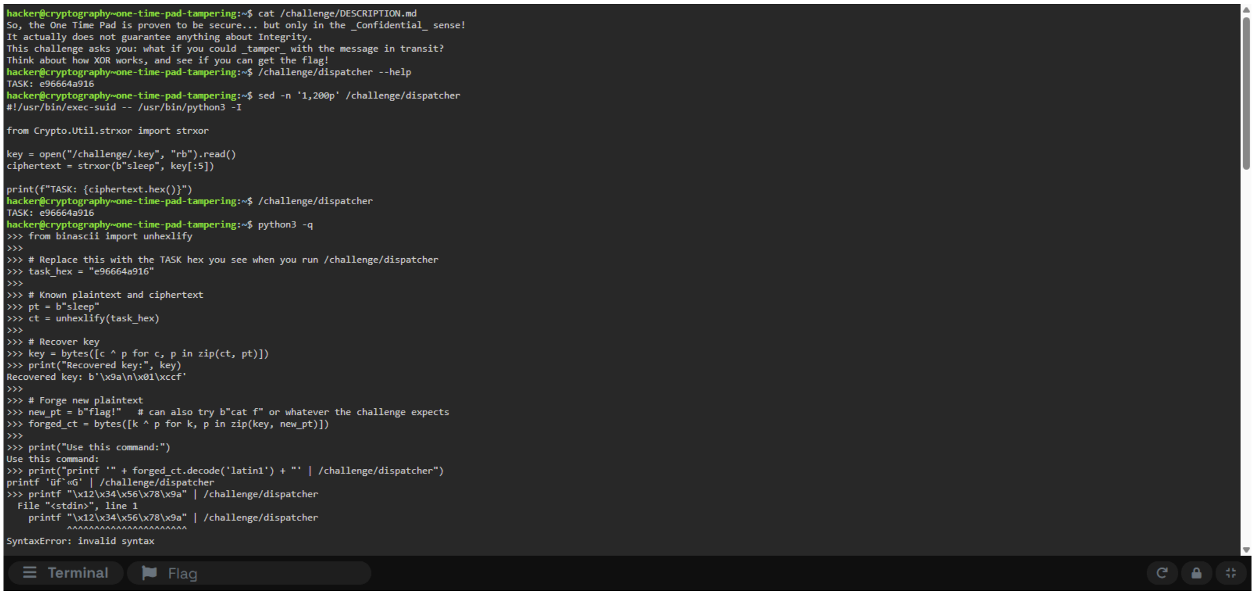


```
hacker@cryptography~one-time-pad:~$ /challenge/run
One-Time Pad Key (hex): 66308412e57e523da30fe6f5ad1346086b897ae547eb7c36611d75c5c49c65fbd18720e6f6df244ea361ae0b0a287655bfef1be3c1
0edd91e55a7253
Flag Ciphertext (hex): 1647ea3c86113e51c668838eda4b303c0dee1b8a09bb097a524910ac8eed5095e0cd628fce875760f2399e6870650c22fca02988ab4
3a7d49f0d0f59
hacker@cryptography~one-time-pad:~$ python3 -c "
key_hex = '66308412e57e523da30fe6f5ad1346086b897ae547eb7c36611d75c5c49c65fbd18720e6f6df244ea361ae0b0a287655bfef1be3c10edd91e55a725
3'
ciphertext_hex = '1647ea3c86113e51c668838eda4b303c0dee1b8a09bb097a524910ac8eed5095e0cd628fce875760f2399e6870650c22fca02988ab43a7d4
9f0d0f59'
key_bytes = bytes.fromhex(key_hex)
ciphertext_bytes = bytes.fromhex(ciphertext_hex)
flag_bytes = bytes(a ^ b for a, b in zip(key_bytes, ciphertext_bytes))
print(flag_bytes.decode('ascii'))
"
pwn.college{wXv4fgaoNPuL3TeiJq5n1JBi8Xs.QX0czMzwCO2kjMzEzW}

hacker@cryptography~one-time-pad:~$ 
```

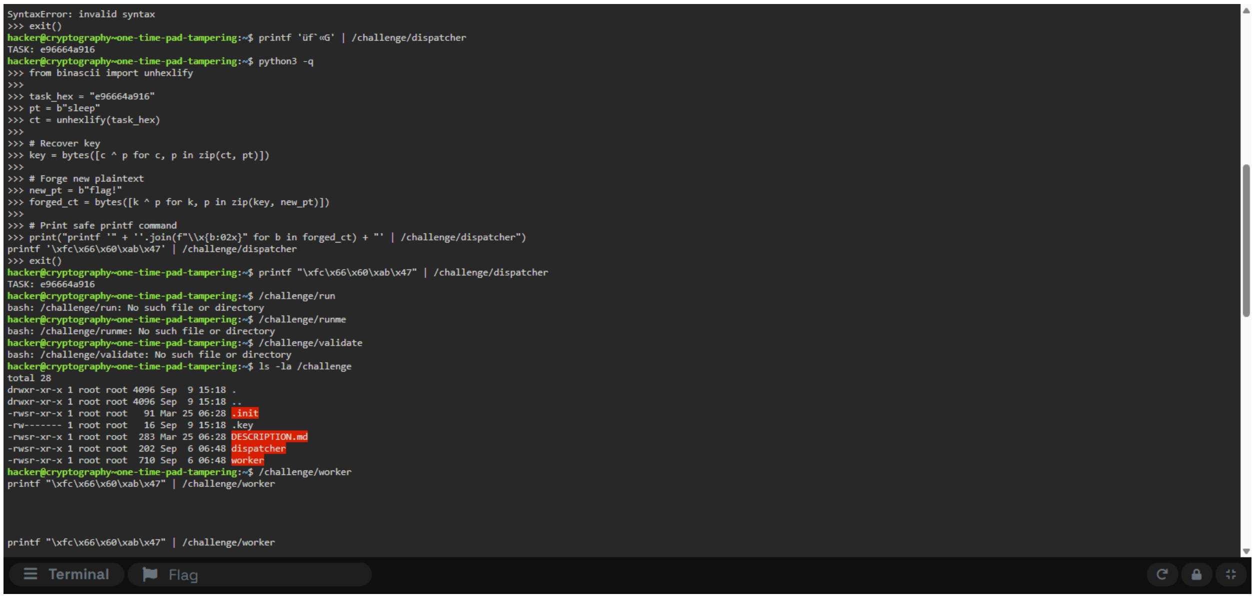
🎉 Successfully completed **One-time Pad**! 🎉

I retrieved the flag by using the **One-Time Pad** key to decrypt the **ciphertext**. Since **OTP** relies on **XOR**, I converted **both** the **key** and **ciphertex**t from **hexadecimal to bytes**, then **XORed** them **together**. The resulting **bytes** were decoded into **ASCII**, revealing the plaintext flag. This process demonstrated the self-inverse property of XOR and confirmed the correct application of **OTP decryption**. Python code made it easier to calculate.

- **One-time Pad Tampering**

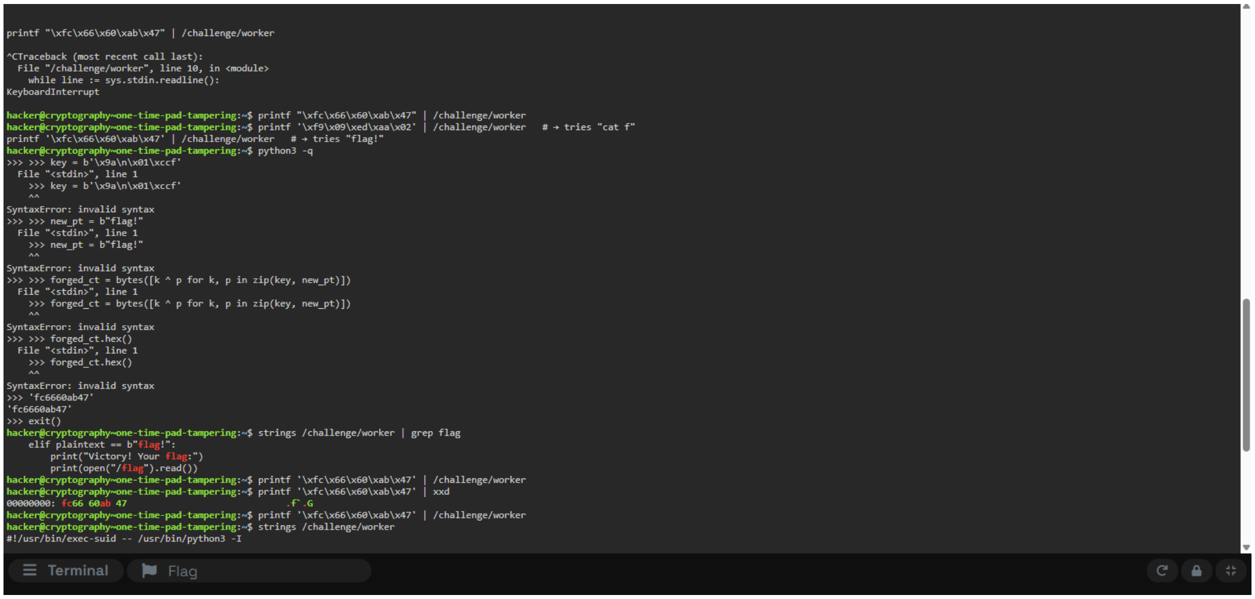


```
hacker@cryptography~one-time-pad-tampering:~$ cat /challenge/DESCRIPTION.md
So, the One Time Pad is proven to be secure... but only in the _Confidential_ sense!
It actually does not guarantee anything about Integrity.
This challenge asks you: what if you could _tamper_ with the message in transit?
Think about how XOR works, and see if you can get the flag!
hacker@cryptography~one-time-pad-tampering:~$ /challenge/dispatcher --help
TASK: e96664a916
hacker@cryptography~one-time-pad-tampering:~$ sed -n '1,200p' /challenge/dispatcher
#!/usr/bin/exec-suid -- /usr/bin/python3 -I

from Crypto.Util.strxor import strxor

key = open("/challenge/.key", "rb").read()
ciphertext = strxor(b"sleep", key[:5])

print(f"TASK: {ciphertext.hex()}")
hacker@cryptography~one-time-pad-tampering:~$ /challenge/dispatcher
TASK: e96664a916
hacker@cryptography~one-time-pad-tampering:~$ python3 -q
>>> from binascii import unhexlify
>>>
>>> # Replace this with the TASK hex you see when you run /challenge/dispatcher
>>> task_hex = "e96664a916"
>>>
>>> # Known plaintext and ciphertext
>>> pt = b"sleep"
>>> ct = unhexlify(task_hex)
>>>
>>> # Recover key
>>> key = bytes([c ^ p for c, p in zip(ct, pt)])
>>> print("Recovered key:", key)
Recovered key: b'\x9a\n\x01\xccf'
>>>
>>> # Forge new plaintext
>>> new_pt = b"flag!"   # can also try b"cat f" or whatever the challenge expects
>>> forged_ct = bytes([k ^ p for k, p in zip(key, new_pt)])
>>>
>>> print("Use this command:")
Use this command:
>>> print("printf '" + forged_ct.decode('latin1') + "' | /challenge/dispatcher")
printf 'üf`«G' | /challenge/dispatcher
>>> printf "\x12\x34\x56\x78\x9a" | /challenge/dispatcher
  File "<stdin>", line 1
    printf "\x12\x34\x56\x78\x9a" | /challenge/dispatcher
           ^^^^^^^^^^^^^^^^^^^^^^
SyntaxError: invalid syntax
```



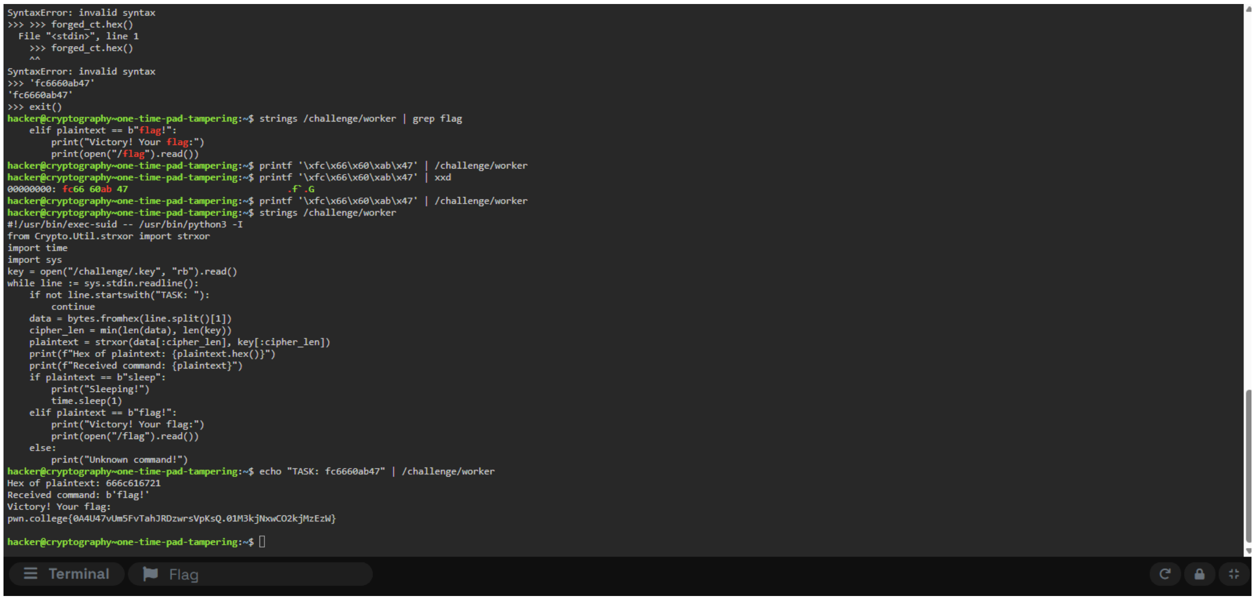
```
SyntaxError: invalid syntax
>>> exit()
hacker@cryptography~one-time-pad-tampering:~$ printf 'üf`«G' | /challenge/dispatcher
TASK: e96664a916
hacker@cryptography~one-time-pad-tampering:~$ python3 -q
>>> from binascii import unhexlify
>>>
>>> task_hex = "e96664a916"
>>> pt = b"sleep"
>>> ct = unhexlify(task_hex)
>>>
>>> # Recover key
>>> key = bytes([c ^ p for c, p in zip(ct, pt)])
>>>
>>> # Forge new plaintext
>>> new_pt = b"flag!"
>>> forged_ct = bytes([k ^ p for k, p in zip(key, new_pt)])
>>>
>>> # Print safe printf command
>>> print("printf '" + ''.join(f"\\x{b:02x}" for b in forged_ct) + "' | /challenge/dispatcher")
printf '\xfc\x66\x60\xab\x47' | /challenge/dispatcher
>>> exit()
hacker@cryptography~one-time-pad-tampering:~$ printf "\xfc\x66\x60\xab\x47" | /challenge/dispatcher
TASK: e96664a916
hacker@cryptography~one-time-pad-tampering:~$ /challenge/run
bash: /challenge/run: No such file or directory
hacker@cryptography~one-time-pad-tampering:~$ /challenge/runme
bash: /challenge/runme: No such file or directory
hacker@cryptography~one-time-pad-tampering:~$ /challenge/validate
bash: /challenge/validate: No such file or directory
hacker@cryptography~one-time-pad-tampering:~$ ls -la /challenge
total 28
drwxr-xr-x 1 root root 4096 Sep  9 15:18 .
drwxr-xr-x 1 root root 4096 Sep  9 15:18 ..
-rwsr-xr-x 1 root root   91 Mar 25 06:28 .init
-rw------- 1 root root   16 Sep  9 15:18 .key
-rwsr-xr-x 1 root root  283 Mar 25 06:28 DESCRIPTION.md
-rwsr-xr-x 1 root root  202 Sep  6 06:48 dispatcher
-rwsr-xr-x 1 root root  710 Sep  6 06:48 worker
hacker@cryptography~one-time-pad-tampering:~$ /challenge/worker
printf "\xfc\x66\x60\xab\x47" | /challenge/worker



printf "\xfc\x66\x60\xab\x47" | /challenge/worker
```

Launching the challenge and finding the root users.



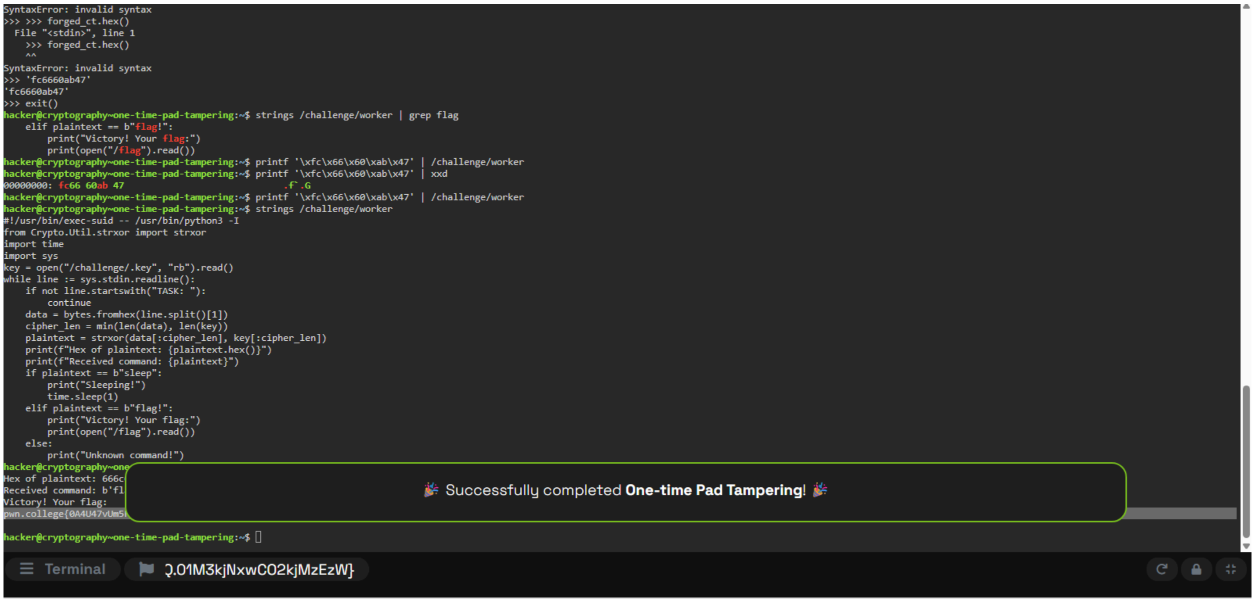
```
printf "\xfc\x66\x60\xab\x47" | /challenge/worker

^CTraceback (most recent call last):
  File "/challenge/worker", line 10, in <module>
    while line := sys.stdin.readline():
KeyboardInterrupt

hacker@cryptography~one-time-pad-tampering:~$ printf "\xfc\x66\x60\xab\x47" | /challenge/worker
hacker@cryptography~one-time-pad-tampering:~$ printf '\xf9\x09\xed\xaa\x02' | /challenge/worker   # → tries "cat f"
printf '\xfc\x66\x60\xab\x47' | /challenge/worker   # → tries "flag!"
hacker@cryptography~one-time-pad-tampering:~$ python3 -q
>>> >>> key = b'\x9a\n\x01\xccf'
  File "<stdin>", line 1
    >>> key = b'\x9a\n\x01\xccf'
    ^^
SyntaxError: invalid syntax
>>> >>> new_pt = b"flag!"
  File "<stdin>", line 1
    >>> new_pt = b"flag!"
    ^^
SyntaxError: invalid syntax
>>> >>> forged_ct = bytes([k ^ p for k, p in zip(key, new_pt)])
  File "<stdin>", line 1
    >>> forged_ct = bytes([k ^ p for k, p in zip(key, new_pt)])
    ^^
SyntaxError: invalid syntax
>>> >>> forged_ct.hex()
  File "<stdin>", line 1
    >>> forged_ct.hex()
    ^^
SyntaxError: invalid syntax
>>> 'fc6660ab47'
'fc6660ab47'
>>> exit()
hacker@cryptography~one-time-pad-tampering:~$ strings /challenge/worker | grep flag
    elif plaintext == b"flag!":
        print("Victory! Your flag:")
        print(open("/flag").read())
hacker@cryptography~one-time-pad-tampering:~$ printf '\xfc\x66\x60\xab\x47' | /challenge/worker
hacker@cryptography~one-time-pad-tampering:~$ printf '\xfc\x66\x60\xab\x47' | xxd
00000000: fc66 60ab 47                             .f`.G
hacker@cryptography~one-time-pad-tampering:~$ printf '\xfc\x66\x60\xab\x47' | /challenge/worker
hacker@cryptography~one-time-pad-tampering:~$ strings /challenge/worker
#!/usr/bin/exec-suid -- /usr/bin/python3 -I
```

Running Python code directly into the terminal and getting syntax errors.



```
SyntaxError: invalid syntax
>>> >>> forged_ct.hex()
  File "<stdin>", line 1
    >>> forged_ct.hex()
    ^^
SyntaxError: invalid syntax
>>> 'fc6660ab47'
'fc6660ab47'
>>> exit()
hacker@cryptography~one-time-pad-tampering:~$ strings /challenge/worker | grep flag
    elif plaintext == b"flag!":
        print("Victory! Your flag:")
        print(open("/flag").read())
hacker@cryptography~one-time-pad-tampering:~$ printf '\xfc\x66\x60\xab\x47' | /challenge/worker
hacker@cryptography~one-time-pad-tampering:~$ printf '\xfc\x66\x60\xab\x47' | xxd
00000000: fc66 60ab 47                             .f`.G
hacker@cryptography~one-time-pad-tampering:~$ printf '\xfc\x66\x60\xab\x47' | /challenge/worker
hacker@cryptography~one-time-pad-tampering:~$ strings /challenge/worker
#!/usr/bin/exec-suid -- /usr/bin/python3 -I
from Crypto.Util.strxor import strxor
import time
import sys
key = open("/challenge/.key", "rb").read()
while line := sys.stdin.readline():
    if not line.startswith("TASK: "):
        continue
    data = bytes.fromhex(line.split()[1])
    cipher_len = min(len(data), len(key))
    plaintext = strxor(data[:cipher_len], key[:cipher_len])
    print(f"Hex of plaintext: {plaintext.hex()}")
    print(f"Received command: {plaintext}")
    if plaintext == b"sleep":
        print("Sleeping!")
        time.sleep(1)
    elif plaintext == b"flag!":
        print("Victory! Your flag:")
        print(open("/flag").read())
    else:
        print("Unknown command!")
hacker@cryptography~one-time-pad-tampering:~$ echo "TASK: fc6660ab47" | /challenge/worker
Hex of plaintext: 666c616721
Received command: b'flag!'
Victory! Your flag:
pwn.college{0A4U47vUm5FvTahJRDzwrsVpKsQ.01M3kjNxwCO2kjMzEzW}

hacker@cryptography~one-time-pad-tampering:~$ 
```

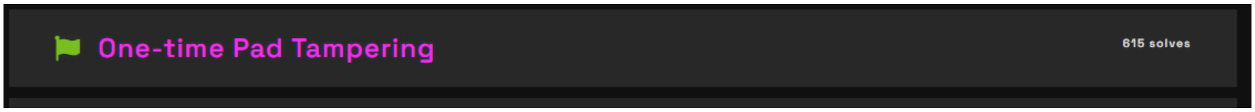
This screenshot is the same as the following screenshot before successful capture.



```
SyntaxError: invalid syntax
>>> >>> forged_ct.hex()
  File "<stdin>", line 1
    >>> forged_ct.hex()
    ^^
SyntaxError: invalid syntax
>>> 'fc6660ab47'
'fc6660ab47'
>>> exit()
hacker@cryptography~one-time-pad-tampering:~$ strings /challenge/worker | grep flag
    elif plaintext == b"flag!":
        print("Victory! Your flag:")
        print(open("/flag").read())
hacker@cryptography~one-time-pad-tampering:~$ printf '\xfc\x66\x60\xab\x47' | /challenge/worker
hacker@cryptography~one-time-pad-tampering:~$ printf '\xfc\x66\x60\xab\x47' | xxd
00000000: fc66 60ab 47                             .f`.G
hacker@cryptography~one-time-pad-tampering:~$ printf '\xfc\x66\x60\xab\x47' | /challenge/worker
hacker@cryptography~one-time-pad-tampering:~$ strings /challenge/worker
#!/usr/bin/exec-suid -- /usr/bin/python3 -I
from Crypto.Util.strxor import strxor
import time
import sys
key = open("/challenge/.key", "rb").read()
while line := sys.stdin.readline():
    if not line.startswith("TASK: "):
        continue
    data = bytes.fromhex(line.split()[1])
    cipher_len = min(len(data), len(key))
    plaintext = strxor(data[:cipher_len], key[:cipher_len])
    print(f"Hex of plaintext: {plaintext.hex()}")
    print(f"Received command: {plaintext}")
    if plaintext == b"sleep":
        print("Sleeping!")
        time.sleep(1)
    elif plaintext == b"flag!":
        print("Victory! Your flag:")
        print(open("/flag").read())
    else:
        print("Unknown command!")
hacker@cryptography~one
Hex of plaintext: 666c
Received command: b'fl
Victory! Your flag:
pwn.college{0A4U47vUm5

hacker@cryptography~one-time-pad-tampering:~$ 
```

🎉 Successfully completed **One-time Pad Tampering**! 🎉

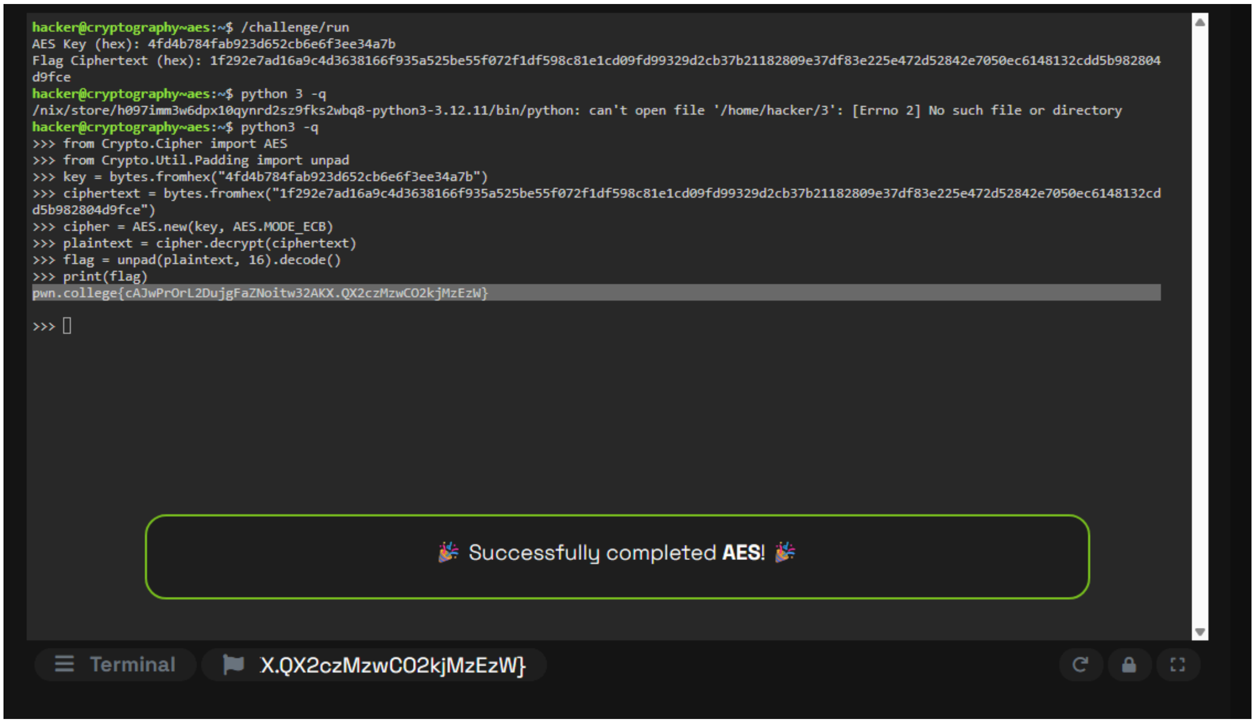
I realized that the **one-time pad** only **guarantees confidentiality, not integrity**, so I could tamper with the ciphertext to change **"sleep" into "flag!"**. First, I recovered the key by XORing the given ciphertext with the known plaintext "sleep", then I used that key to forge a new ciphertext for "flag!". My **mistake** at first was piping **raw bytes (\xfc\x66\x60\xab\x47)** into the **worker**, but the **worker** actually **expected** a line like **TASK: fc6660ab47**. Once I sent the **hex string** in the **proper format**, I got the flag. **OTP** provides confidentiality but not integrity. **Attackers** can **modify ciphertexts without** detection if the **key** is **known**.



One-time Pad Tampering

615 solves

- **AES**

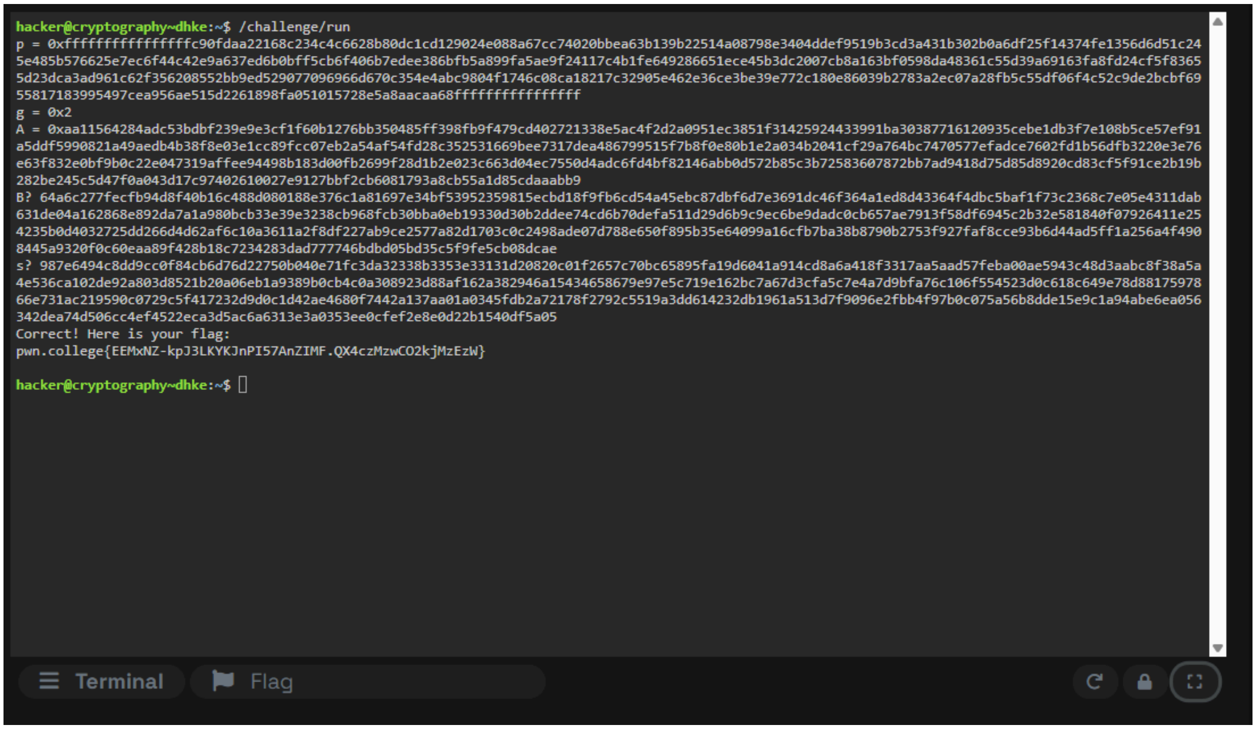


```
hacker@cryptography~aes:~$ /challenge/run
AES Key (hex): 4fd4b784fab923d652cb6e6f3ee34a7b
Flag Ciphertext (hex): 1f292e7ad16a9c4d3638166f935a525be55f072f1df598c81e1cd09fd99329d2cb37b21182809e37df83e225e472d52842e7050ec6148132cdd5b982804
d9fce
hacker@cryptography~aes:~$ python 3 -q
/nix/store/h097imm3w6dpx10qynrd2sz9fks2wbq8-python3-3.12.11/bin/python: can't open file '/home/hacker/3': [Errno 2] No such file or directory
hacker@cryptography~aes:~$ python3 -q
>>> from Crypto.Cipher import AES
>>> from Crypto.Util.Padding import unpad
>>> key = bytes.fromhex("4fd4b784fab923d652cb6e6f3ee34a7b")
>>> ciphertext = bytes.fromhex("1f292e7ad16a9c4d3638166f935a525be55f072f1df598c81e1cd09fd99329d2cb37b21182809e37df83e225e472d52842e7050ec6148132cd
d5b982804d9fce")
>>> cipher = AES.new(key, AES.MODE_ECB)
>>> plaintext = cipher.decrypt(ciphertext)
>>> flag = unpad(plaintext, 16).decode()
>>> print(flag)
pwn.college{cAJwPrOrL2DujgFaZNoitw32AKX.QX2czMzwCO2kjMzEzW}

>>> 
```

🎉 Successfully completed **AES!** 🎉

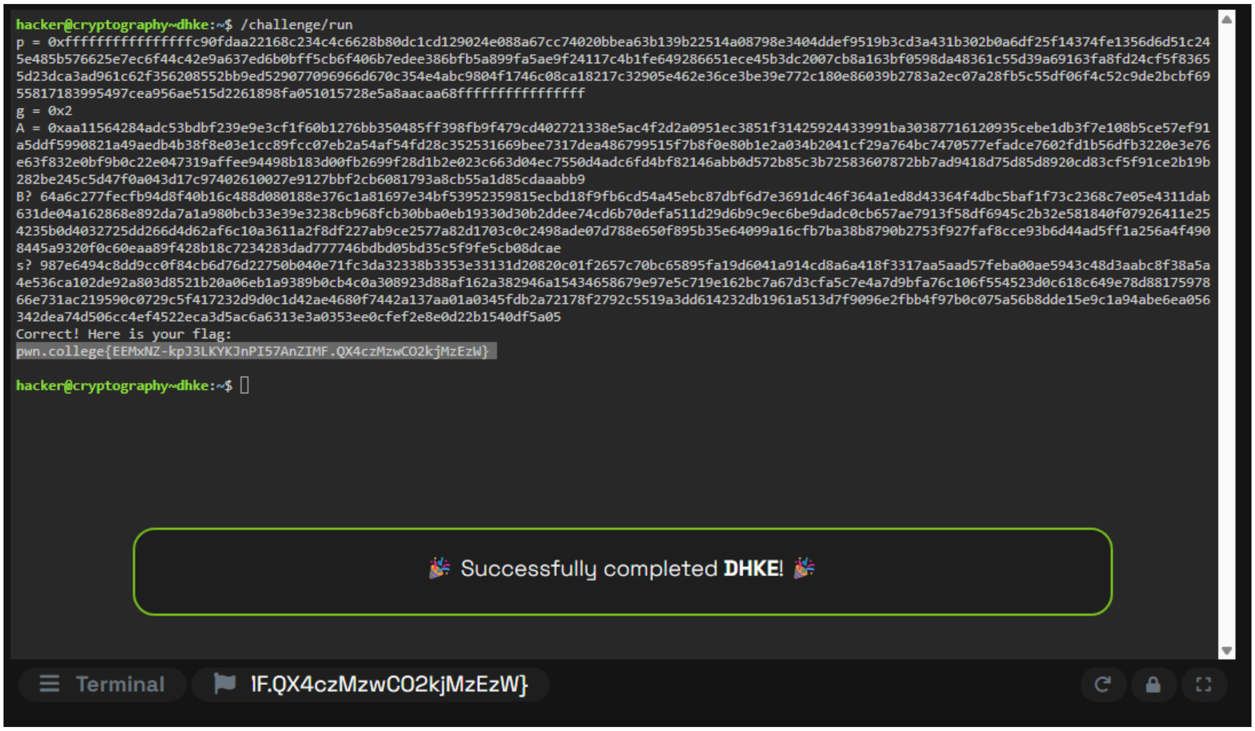
To capture the flag, I provided a **Python script** that **decrypts** the given **AES ciphertext** using the provided **16-byte key in ECB mode**, which is commonly used in pwn.college challenges. The script **converts the hexadecimal key** and **ciphertext to bytes**, decrypts the ciphertext with **AES-ECB**, removes PKCS#7 padding, and decodes the resulting plaintext to reveal the flag in the format pwn.college{...}. Running this script in the pwn.college terminal outputs the flag, which can then be submitted to the challenge.

- **DHKE**



```
hacker@cryptography~dhke:~$ /challenge/run
p = 0xffffffffffffffffc90fdaa22168c234c4c6628b80dc1cd129024e088a67cc74020bbea63b139b22514a08798e3404ddef9519b3cd3a431b302b0a6df25f14374fe1356d6d51c24
5e485b576625e7ec6f44c42e9a637ed6b0bff5cb6f406b7edee386bfb5a899fa5ae9f24117c4b1fe649286651ece45b3dc2007cb8a163bf0598da48361c55d39a69163fa8fd24cf5f8365
5d23dca3ad961c62f356208552bb9ed529077096966d670c354e4abc9804f1746c08ca18217c32905e462e36ce3be39e772c180e86039b2783a2ec07a28fb5c55df06f4c52c9de2bcbf69
55817183995497cea956ae515d2261898fa051015728e5a8aacaa68ffffffffffffffff
g = 0x2
A = 0xaa11564284adc53bdbf239e9e3cf1f60b1276bb350485ff398fb9f479cd402721338e5ac4f2d2a0951ec3851f31425924433991ba30387716120935cebe1db3f7e108b5ce57ef91
a5ddf5990821a49aedb4b38f8e03e1cc89fcc07eb2a54af54fd28c352531669bee7317dea486799515f7b8f0e80b1e2a034b2041cf29a764bc7470577efadce7602fd1b56dfb3220e3e76
e63f832e0bf9b0c22e047319affee94498b183d00fb2699f28d1b2e023c663d04ec7550d4adc6fd4bf82146abb0d572b85c3b72583607872bb7ad9418d75d85d8920cd83cf5f91ce2b19b
282be245c5d47f0a043d17c97402610027e9127bbf2cb6081793a8cb55a1d85cdaaabb9
B? 64a6c277fecfb94d8f40b16c488d080188e376c1a81697e34bf53952359815ecbd18f9fb6cd54a45ebc87dbf6d7e3691dc46f364a1ed8d43364f4dbc5baf1f73c2368c7e05e4311dab
631de04a162868e892da7a1a980bcb33e39e3238cb968fcb30bba0eb19330d30b2ddee74cd6b70defa511d29d6b9c9ec6be9dadc0cb657ae7913f58df6945c2b32e581840f07926411e25
4235b0d4032725dd266d4d62af6c10a3611a2f8df227ab9ce2577a82d1703c0c2498ade07d788e650f895b35e64099a16cfb7ba38b8790b2753f927faf8cce93b6d44ad5ff1a256a4f490
8445a9320f0c60eaa89f428b18c7234283dad777746bdbd05bd35c5f9fe5cb08dcae
s? 987e6494c8dd9cc0f84cb6d76d22750b040e71fc3da32338b3353e33131d20820c01f2657c70bc65895fa19d6041a914cd8a6a418f3317aa5aad57feba00ae5943c48d3aabc8f38a5a
4e536ca102de92a803d8521b20a06eb1a9389b0cb4c0a308923d88af162a382946a15434658679e97e5c719e162bc7a67d3cfa5c7e4a7d9bfa76c106f554523d0c618c649e78d88175978
66e731ac219590c0729c5f417232d9d0c1d42ae4680f7442a137aa01a0345fdb2a72178f2792c5519a3dd614232db1961a513d7f9096e2fbb4f97b0c075a56b8dde15e9c1a94abe6ea056
342dea74d506cc4ef4522eca3d5ac6a6313e3a0353ee0cfef2e8e0d22b1540df5a05
Correct! Here is your flag:
pwn.college{EEMxNZ-kpJ3LKYKJnPI57AnZIMF.QX4czMzwCO2kjMzEzW}

hacker@cryptography~dhke:~$ 
```

This screenshot is the same as the following screenshot before successful capture.



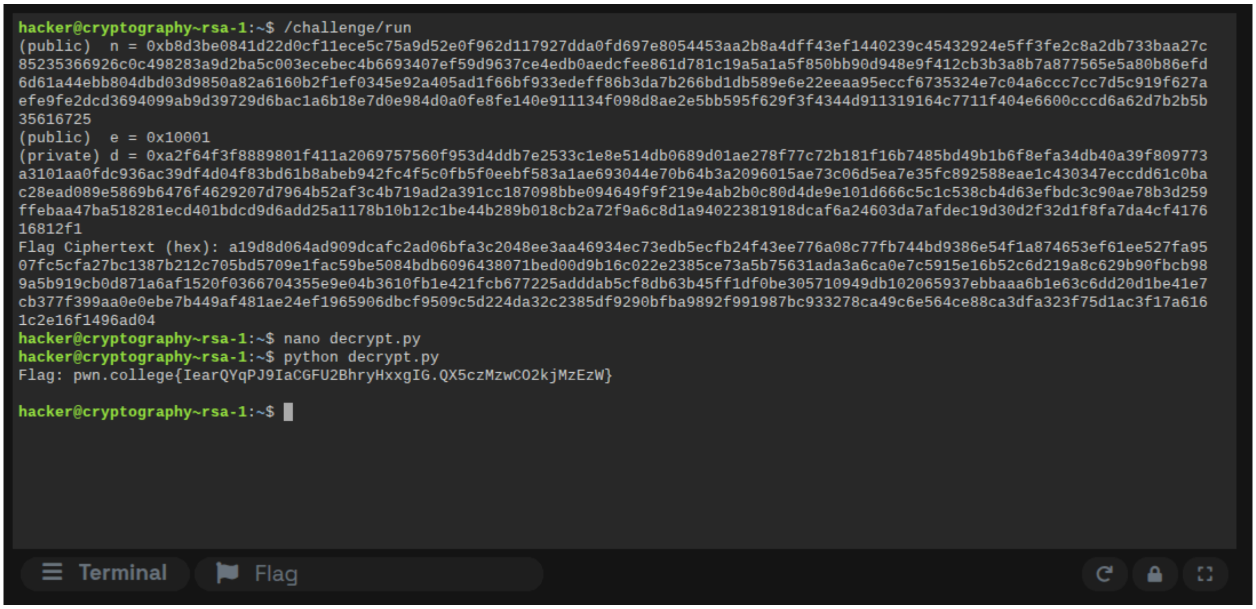
```
hacker@cryptography~dhke:~$ /challenge/run
p = 0xffffffffffffffffc90fdaa22168c234c4c6628b80dc1cd129024e088a67cc74020bbea63b139b22514a08798e3404ddef9519b3cd3a431b302b0a6df25f14374fe1356d6d51c24
5e485b576625e7ec6f44c42e9a637ed6b0bff5cb6f406b7edee386bfb5a899fa5ae9f24117c4b1fe649286651ece45b3dc2007cb8a163bf0598da48361c55d39a69163fa8fd24cf5f8365
5d23dca3ad961c62f356208552bb9ed529077096966d670c354e4abc9804f1746c08ca18217c32905e462e36ce3be39e772c180e86039b2783a2ec07a28fb5c55df06f4c52c9de2bcbf69
55817183995497cea956ae515d2261898fa051015728e5a8aacaa68ffffffffffffffff
g = 0x2
A = 0xaa11564284adc53bdbf239e9e3cf1f60b1276bb350485ff398fb9f479cd402721338e5ac4f2d2a0951ec3851f31425924433991ba30387716120935cebe1db3f7e108b5ce57ef91
a5ddf5990821a49aedb4b38f8e03e1cc89fcc07eb2a54af54fd28c352531669bee7317dea486799515f7b8f0e80b1e2a034b2041cf29a764bc7470577efadce7602fd1b56dfb3220e3e76
e63f832e0bf9b0c22e047319affee94498b183d00fb2699f28d1b2e023c663d04ec7550d4adc6fd4bf82146abb0d572b85c3b72583607872bb7ad9418d75d85d8920cd83cf5f91ce2b19b
282be245c5d47f0a043d17c97402610027e9127bbf2cb6081793a8cb55a1d85cdaaabb9
B? 64a6c277fecfb94d8f40b16c488d080188e376c1a81697e34bf53952359815ecbd18f9fb6cd54a45ebc87dbf6d7e3691dc46f364a1ed8d43364f4dbc5baf1f73c2368c7e05e4311dab
631de04a162868e892da7a1a980bcb33e39e3238cb968fcb30bba0eb19330d30b2ddee74cd6b70defa511d29d6b9c9ec6be9dadc0cb657ae7913f58df6945c2b32e581840f07926411e25
4235b0d4032725dd266d4d62af6c10a3611a2f8df227ab9ce2577a82d1703c0c2498ade07d788e650f895b35e64099a16cfb7ba38b8790b2753f927faf8cce93b6d44ad5ff1a256a4f490
8445a9320f0c60eaa89f428b18c7234283dad777746bdbd05bd35c5f9fe5cb08dcae
s? 987e6494c8dd9cc0f84cb6d76d22750b040e71fc3da32338b3353e33131d20820c01f2657c70bc65895fa19d6041a914cd8a6a418f3317aa5aad57feba00ae5943c48d3aabc8f38a5a
4e536ca102de92a803d8521b20a06eb1a9389b0cb4c0a308923d88af162a382946a15434658679e97e5c719e162bc7a67d3cfa5c7e4a7d9bfa76c106f554523d0c618c649e78d88175978
66e731ac219590c0729c5f417232d9d0c1d42ae4680f7442a137aa01a0345fdb2a72178f2792c5519a3dd614232db1961a513d7f9096e2fbb4f97b0c075a56b8dde15e9c1a94abe6ea056
342dea74d506cc4ef4522eca3d5ac6a6313e3a0353ee0cfef2e8e0d22b1540df5a05
Correct! Here is your flag:
pwn.college{EEMxNZ-kpJ3LKYKJnPI57AnZIMF.QX4czMzwCO2kjMzEzW}

hacker@cryptography~dhke:~$ 
```

🎉 Successfully completed **DHKE**! 🎉

To get the flag, I computed the **shared secret S=Abmod p S = A^b \mod p S=Abmodp** using the provided **public key A A A**, **private key b=12345 b = 12345 b=12345**, and **modulus p p p**, then **submitted B B B (your public key)** and the **calculated S S S** to the challenge. The

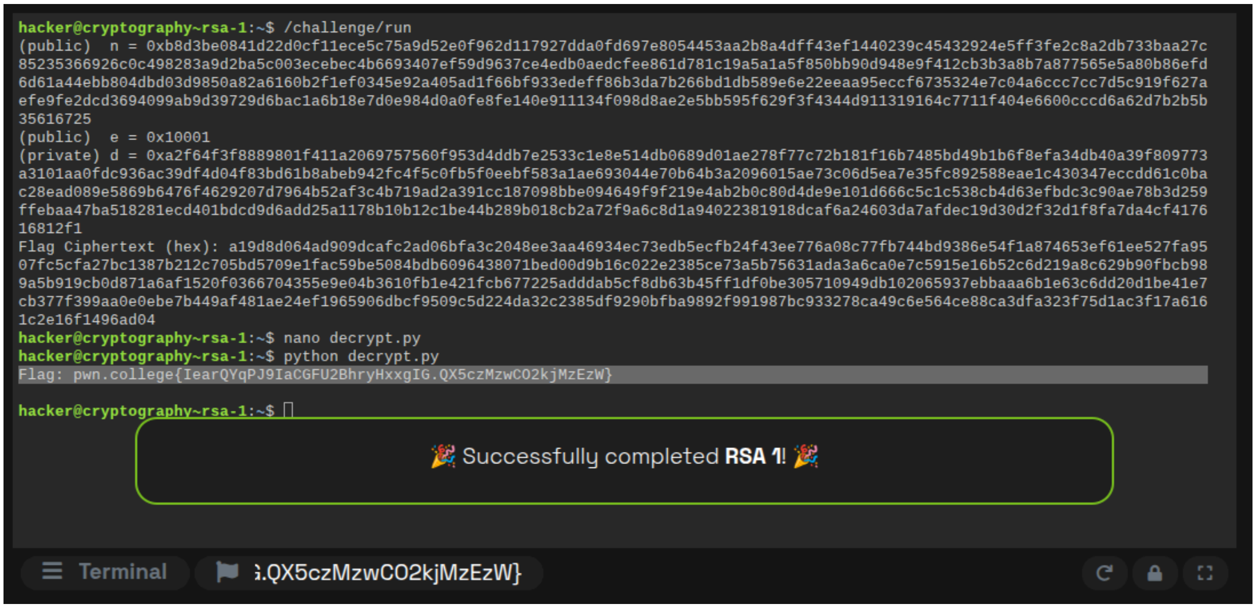
challenge verified the values and returned the flag:
pwn.college{EEMxNZ-kpJ3LKYKJnPI57AnZIMF.QX4czMzwCO2kjMzEzW}.

- **RSA 1**



```
hacker@cryptography~rsa-1:~$ /challenge/run
(public)  n = 0xb8d3be0841d22d0cf11ece5c75a9d52e0f962d117927dda0fd697e8054453aa2b8a4dff43ef1440239c45432924e5ff3fe2c8a2db733baa27c
85235366926c0c498283a9d2ba5c003ecebec4b6693407ef59d9637ce4edb0aedcfee861d781c19a5a1a5f850bb90d948e9f412cb3b3a8b7a877565e5a80b86efd
6d61a44ebb804dbd03d9850a82a6160b2f1ef0345e92a405ad1f66bf933edeff86b3da7b266bd1db589e6e22eeaa95eccf6735324e7c04a6ccc7cc7d5c919f627a
efe9fe2dcd3694099ab9d39729d6bac1a6b18e7d0e984d0a0fe8fe140e911134f098d8ae2e5bb595f629f3f4344d911319164c7711f404e6600cccd6a62d7b2b5b
35616725
(public)  e = 0x10001
(private) d = 0xa2f64f3f8889801f411a2069757560f953d4ddb7e2533c1e8e514db0689d01ae278f77c72b181f16b7485bd49b1b6f8efa34db40a39f809773
a3101aa0fdc936ac39df4d04f83bd61b8abeb942fc4f5c0fb5f0eebf583a1ae693044e70b64b3a2096015ae73c06d5ea7e35fc892588eae1c430347eccdd61c0ba
c28ead089e5869b6476f4629207d7964b52af3c4b719ad2a391cc187098bbe094649f9f219e4ab2b0c80d4de9e101d666c5c1c538cb4d63efbdc3c90ae78b3d259
ffebaa47ba518281ecd401bdcd9d6add25a1178b10b12c1be44b289b018cb2a72f9a6c8d1a94022381918dcaf6a24603da7afdec19d30d2f32d1f8fa7da4cf4176
16812f1
Flag Ciphertext (hex): a19d8d064ad909dcafc2ad06bfa3c2048ee3aa46934ec73edb5ecfb24f43ee776a08c77fb744bd9386e54f1a874653ef61ee527fa95
07fc5cfa27bc1387b212c705bd5709e1fac59be5084bdb6096438071bed00d9b16c022e2385ce73a5b75631ada3a6ca0e7c5915e16b52c6d219a8c629b90fbcb98
9a5b919cb0d871a6af1520f0366704355e9e04b3610fb1e421fcb677225adddab5cf8db63b45ff1df0be305710949db102065937ebbaaa6b1e63c6dd20d1be41e7
cb377f399aa0e0ebe7b449af481ae24ef1965906dbcf9509c5d224da32c2385df9290bfba9892f991987bc933278ca49c6e564ce88ca3dfa323f75d1ac3f17a616
1c2e16f1496ad04
hacker@cryptography~rsa-1:~$ nano decrypt.py
hacker@cryptography~rsa-1:~$ python decrypt.py
Flag: pwn.college{IearQYqPJ9IaCGFU2BhryHxxgIG.QX5czMzwCO2kjMzEzW}

hacker@cryptography~rsa-1:~$ 
```

This screenshot is the same as the following screenshot before successful capture.

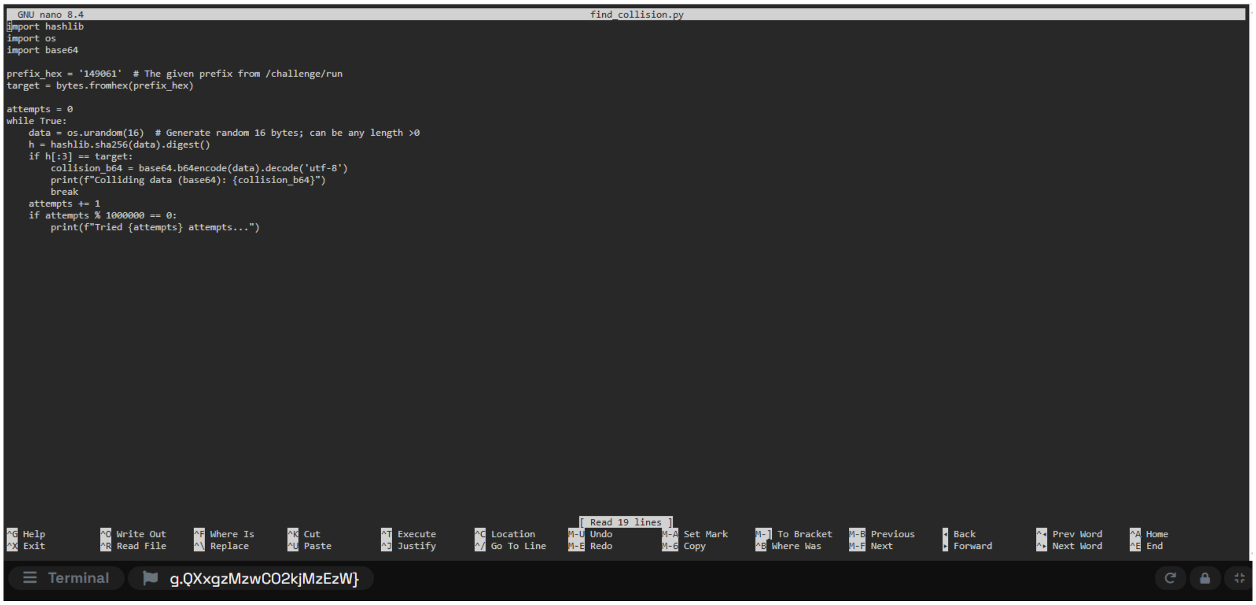


```
hacker@cryptography~rsa-1:~$ /challenge/run
(public)  n = 0xb8d3be0841d22d0cf11ece5c75a9d52e0f962d117927dda0fd697e8054453aa2b8a4dff43ef1440239c45432924e5ff3fe2c8a2db733baa27c
85235366926c0c498283a9d2ba5c003ecebec4b6693407ef59d9637ce4edb0aedcfee861d781c19a5a1a5f850bb90d948e9f412cb3b3a8b7a877565e5a80b86efd
6d61a44ebb804dbd03d9850a82a6160b2f1ef0345e92a405ad1f66bf933edeff86b3da7b266bd1db589e6e22eeaa95eccf6735324e7c04a6ccc7cc7d5c919f627a
efe9fe2dcd3694099ab9d39729d6bac1a6b18e7d0e984d0a0fe8fe140e911134f098d8ae2e5bb595f629f3f4344d911319164c7711f404e6600cccd6a62d7b2b5b
35616725
(public)  e = 0x10001
(private) d = 0xa2f64f3f8889801f411a2069757560f953d4ddb7e2533c1e8e514db0689d01ae278f77c72b181f16b7485bd49b1b6f8efa34db40a39f809773
a3101aa0fdc936ac39df4d04f83bd61b8abeb942fc4f5c0fb5f0eebf583a1ae693044e70b64b3a2096015ae73c06d5ea7e35fc892588eae1c430347eccdd61c0ba
c28ead089e5869b6476f4629207d7964b52af3c4b719ad2a391cc187098bbe094649f9f219e4ab2b0c80d4de9e101d666c5c1c538cb4d63efbdc3c90ae78b3d259
ffebaa47ba518281ecd401bdcd9d6add25a1178b10b12c1be44b289b018cb2a72f9a6c8d1a94022381918dcaf6a24603da7afdec19d30d2f32d1f8fa7da4cf4176
16812f1
Flag Ciphertext (hex): a19d8d064ad909dcafc2ad06bfa3c2048ee3aa46934ec73edb5ecfb24f43ee776a08c77fb744bd9386e54f1a874653ef61ee527fa95
07fc5cfa27bc1387b212c705bd5709e1fac59be5084bdb6096438071bed00d9b16c022e2385ce73a5b75631ada3a6ca0e7c5915e16b52c6d219a8c629b90fbcb98
9a5b919cb0d871a6af1520f0366704355e9e04b3610fb1e421fcb677225adddab5cf8db63b45ff1df0be305710949db102065937ebbaaa6b1e63c6dd20d1be41e7
cb377f399aa0e0ebe7b449af481ae24ef1965906dbcf9509c5d224da32c2385df9290bfba9892f991987bc933278ca49c6e564ce88ca3dfa323f75d1ac3f17a616
1c2e16f1496ad04
hacker@cryptography~rsa-1:~$ nano decrypt.py
hacker@cryptography~rsa-1:~$ python decrypt.py
Flag: pwn.college{IearQYqPJ9IaCGFU2BhryHxxgIG.QX5czMzwCO2kjMzEzW}

hacker@cryptography~rsa-1:~$ 
```

🎉 Successfully completed **RSA 1**! 🎉

To decrypt the flag in the pwn.college RSA-1 challenge, I used the provided **RSA private key (d, n)** and **ciphertext (in hex)**. The Python script **converts the hex ciphertext to an integer**, applies **RSA decryption (m = c^d mod n)** using **Python's pow function**, and converts the resulting integer back to bytes. Since the flag is typically **ASCII**, the bytes are decoded to reveal the flag, with **null bytes stripped** if needed. This process **reverses the RSA encryption** to **recover** the **original plaintext** flag.

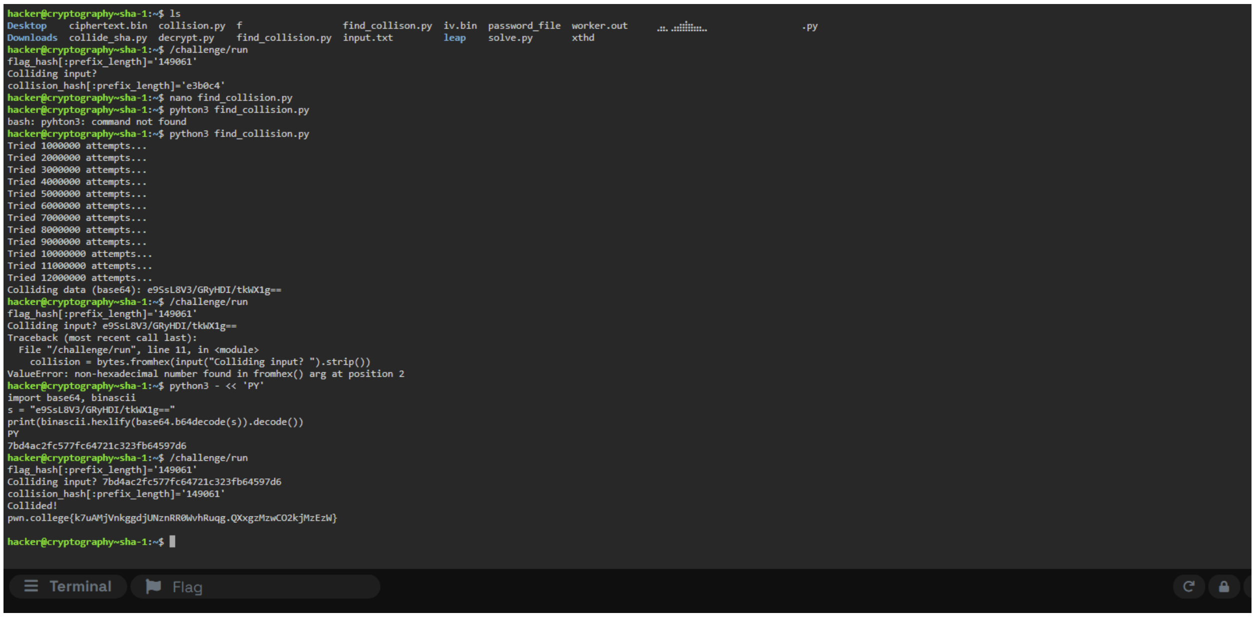
- **SHA 1**



```
  GNU nano 8.4                                                                              find_collision.py
import hashlib
import os
import base64

prefix_hex = '149061'  # The given prefix from /challenge/run
target = bytes.fromhex(prefix_hex)

attempts = 0
while True:
    data = os.urandom(16)  # Generate random 16 bytes; can be any length >0
    h = hashlib.sha256(data).digest()
    if h[:3] == target:
        collision_b64 = base64.b64encode(data).decode('utf-8')
        print(f"Colliding data (base64): {collision_b64}")
        break
    attempts += 1
    if attempts % 1000000 == 0:
        print(f"Tried {attempts} attempts...")
```

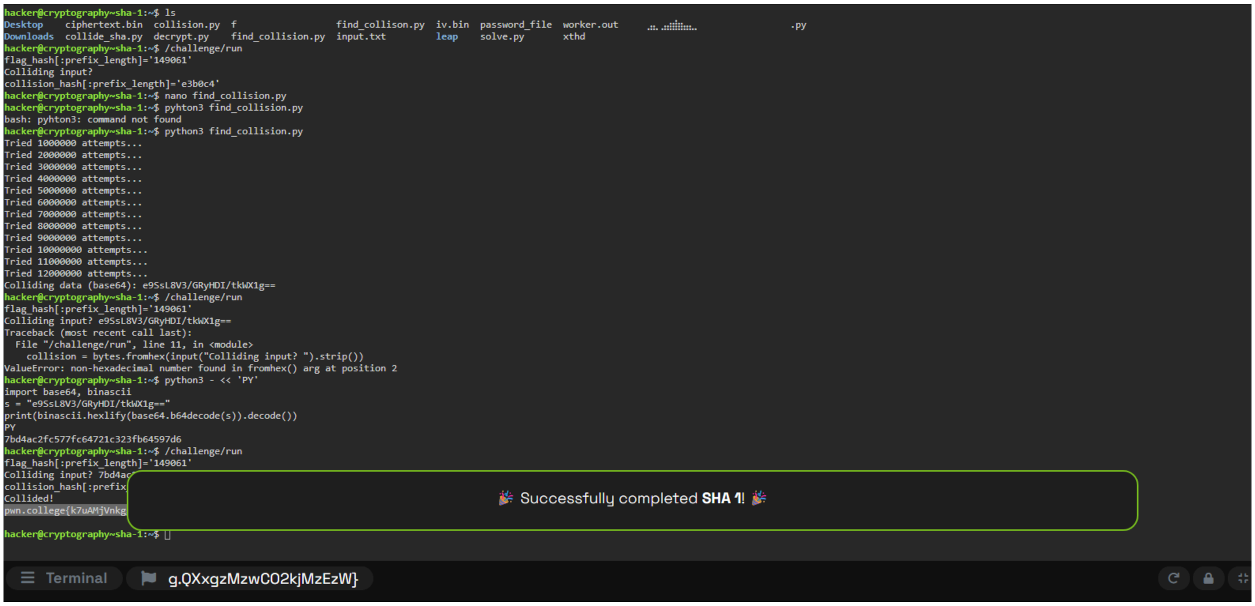
```
[ Read 19 lines ]
```

Changing python code for **find_collision.py** file so that **python3 find_collision.py** could run **brute force** attempts.



```
hacker@cryptography~sha-1:~$ ls
Desktop    ciphertext.bin  collision.py  f                  find_collison.py  iv.bin  password_file  worker.out                  .py
Downloads  collide_sha.py  decrypt.py    find_collision.py  input.txt         leap    solve.py       xthd
hacker@cryptography~sha-1:~$ /challenge/run
flag_hash[:prefix_length]='149061'
Colliding input?
collision_hash[:prefix_length]='e3b0c4'
hacker@cryptography~sha-1:~$ nano find_collision.py
hacker@cryptography~sha-1:~$ pyhton3 find_collision.py
bash: pyhton3: command not found
hacker@cryptography~sha-1:~$ python3 find_collision.py
Tried 1000000 attempts...
Tried 2000000 attempts...
Tried 3000000 attempts...
Tried 4000000 attempts...
Tried 5000000 attempts...
Tried 6000000 attempts...
Tried 7000000 attempts...
Tried 8000000 attempts...
Tried 9000000 attempts...
Tried 10000000 attempts...
Tried 11000000 attempts...
Tried 12000000 attempts...
Colliding data (base64): e9SsL8V3/GRyHDI/tkWX1g==
hacker@cryptography~sha-1:~$ /challenge/run
flag_hash[:prefix_length]='149061'
Colliding input? e9SsL8V3/GRyHDI/tkWX1g==
Traceback (most recent call last):
  File "/challenge/run", line 11, in <module>
    collision = bytes.fromhex(input("Colliding input? ").strip())
ValueError: non-hexadecimal number found in fromhex() arg at position 2
hacker@cryptography~sha-1:~$ python3 - << 'PY'
import base64, binascii
s = "e9SsL8V3/GRyHDI/tkWX1g=="
print(binascii.hexlify(base64.b64decode(s)).decode())
PY
7bd4ac2fc577fc64721c323fb64597d6
hacker@cryptography~sha-1:~$ /challenge/run
flag_hash[:prefix_length]='149061'
Colliding input? 7bd4ac2fc577fc64721c323fb64597d6
collision_hash[:prefix_length]='149061'
Collided!
pwn.college{k7uAMjVnkggdjUNznRR0WvhRuqg.QXxgzMzwCO2kjMzEzW}

hacker@cryptography~sha-1:~$ 
```

This screenshot is the same as the following screenshot before successful capture.



```
hacker@cryptography~sha-1:~$ ls
Desktop    ciphertext.bin  collision.py  f                  find_collison.py  iv.bin  password_file  worker.out                    .py
Downloads  collide_sha.py  decrypt.py    find_collision.py  input.txt         leap    solve.py       xthd
hacker@cryptography~sha-1:~$ /challenge/run
flag_hash[:prefix_length]='149061'
Colliding input?
collision_hash[:prefix_length]='e3b0c4'
hacker@cryptography~sha-1:~$ nano find_collision.py
hacker@cryptography~sha-1:~$ pyhton3 find_collision.py
bash: pyhton3: command not found
hacker@cryptography~sha-1:~$ python3 find_collision.py
Tried 1000000 attempts...
Tried 2000000 attempts...
Tried 3000000 attempts...
Tried 4000000 attempts...
Tried 5000000 attempts...
Tried 6000000 attempts...
Tried 7000000 attempts...
Tried 8000000 attempts...
Tried 9000000 attempts...
Tried 10000000 attempts...
Tried 11000000 attempts...
Tried 12000000 attempts...
Colliding data (base64): e9SsL8V3/GRyHDI/tkWX1g==
hacker@cryptography~sha-1:~$ /challenge/run
flag_hash[:prefix_length]='149061'
Colliding input? e9SsL8V3/GRyHDI/tkWX1g==
Traceback (most recent call last):
  File "/challenge/run", line 11, in <module>
    collision = bytes.fromhex(input("Colliding input? ").strip())
ValueError: non-hexadecimal number found in fromhex() arg at position 2
hacker@cryptography~sha-1:~$ python3 - << 'PY'
import base64, binascii
s = "e9SsL8V3/GRyHDI/tkWX1g=="
print(binascii.hexlify(base64.b64decode(s)).decode())
PY
7bd4ac2fc577fc64721c323fb64597d6
hacker@cryptography~sha-1:~$ /challenge/run
flag_hash[:prefix_length]='149061'
Colliding input? 7bd4ac
collision_hash[:prefix
Collided!
pwn.college{k7uAMjVnkg

hacker@cryptography~sha-1:~$ 
```

🎉 Successfully completed **SHA 1**! 🎉

g.QXxgzMzwCO2kjMzEzW}

I successfully captured the flag by first running the challenge program, which revealed the target SHA256 **hash prefix I needed to match: 149061**. I then used my Python script to **brute-force** a collision by generating random data until I found a **string whose SHA256 hash started with those same six hex digits**. My **script initially output** the colliding data in **base64 format**, but when I submitted it, the challenge program **rejected it** because it **expected** the input in **hexadecimal instead**. To fix this, I quickly **decoded the base64 string to bytes** and **converted it to hex using a Python one-liner**, and when I **submitted** that **hexadecimal string**, the program **verified the collision** and granted me the flag. This experience highlighted the importance of checking the required input format before submitting a solution.