Carlos Rodriguez

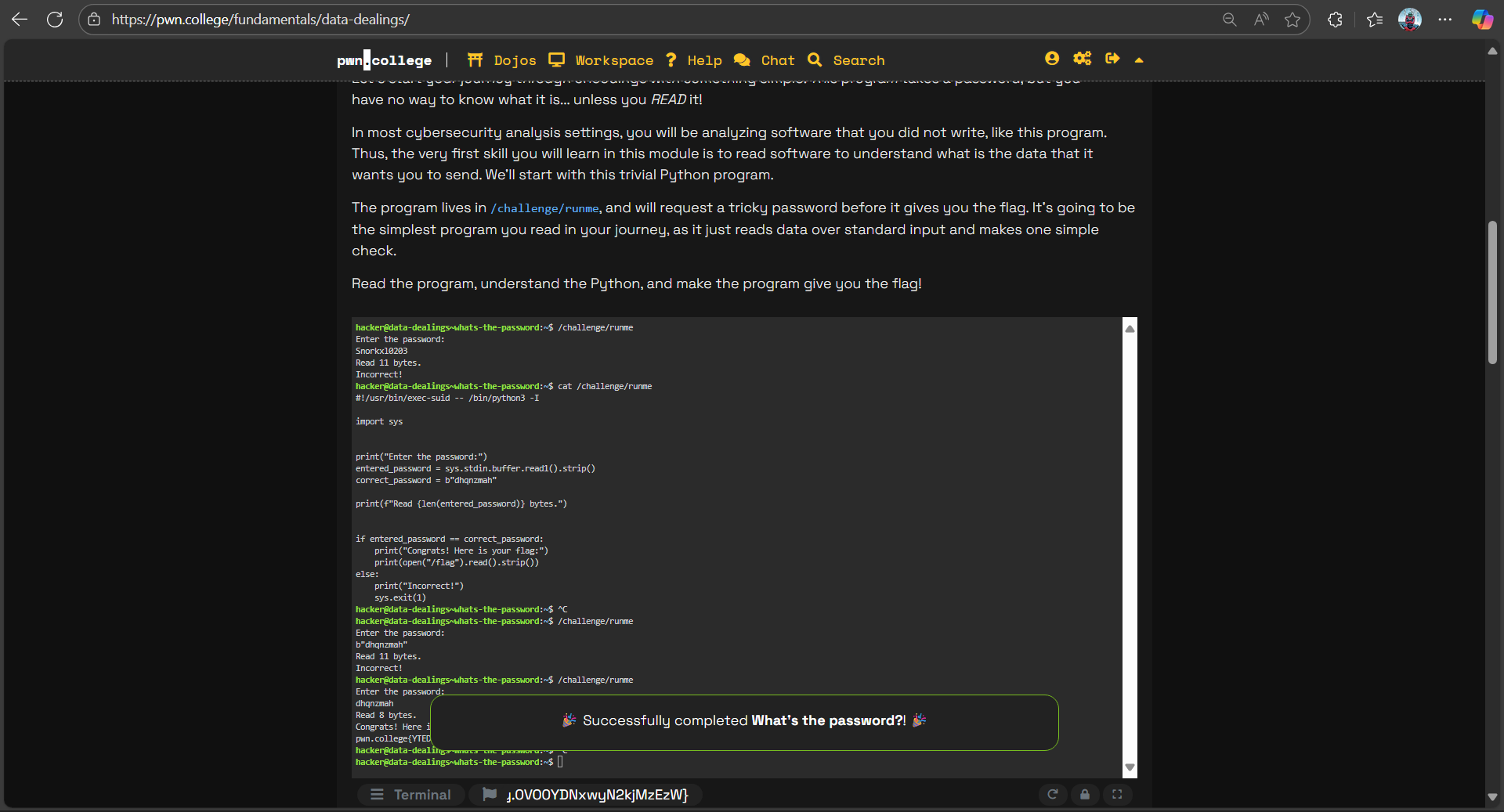
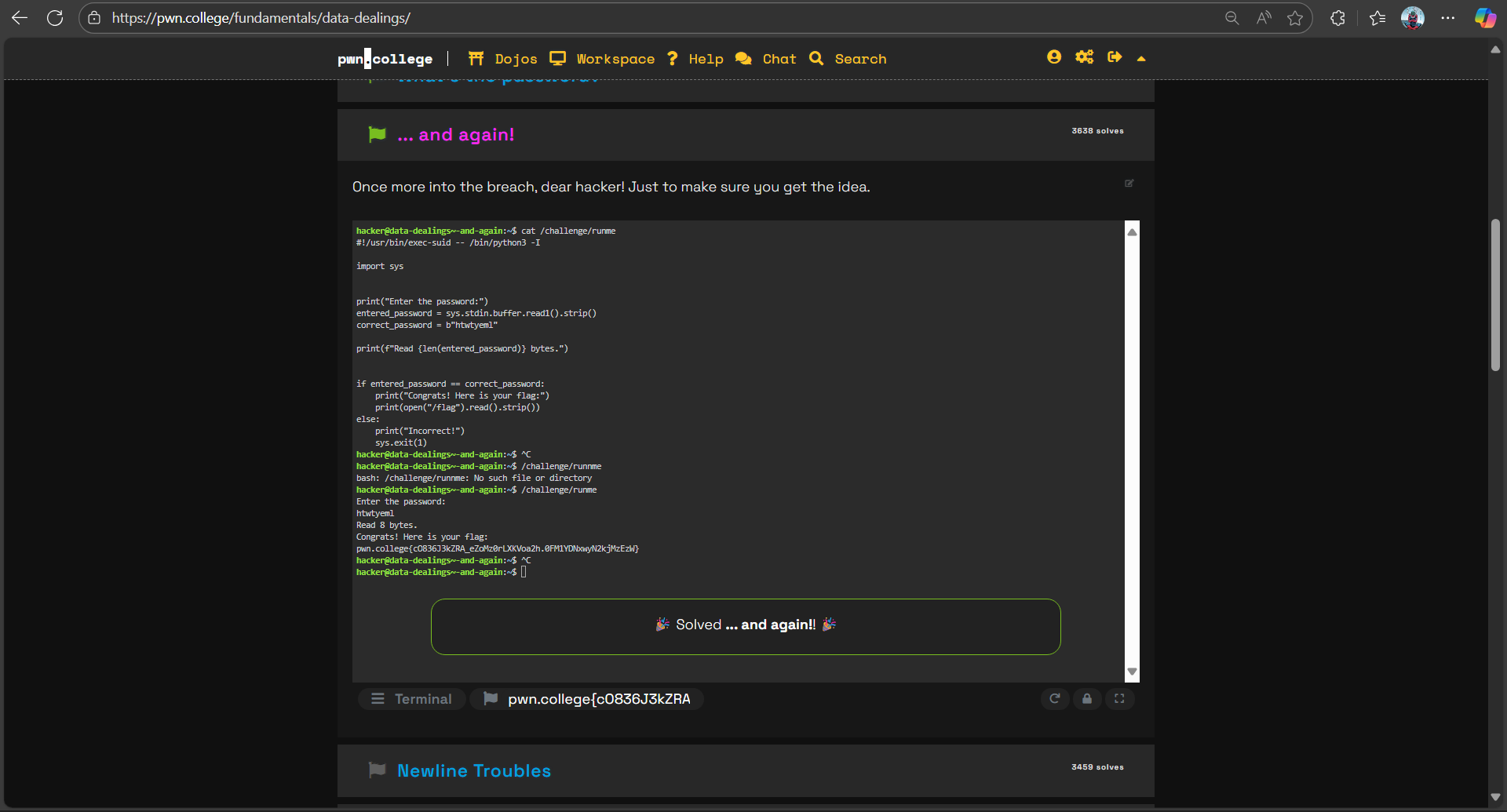
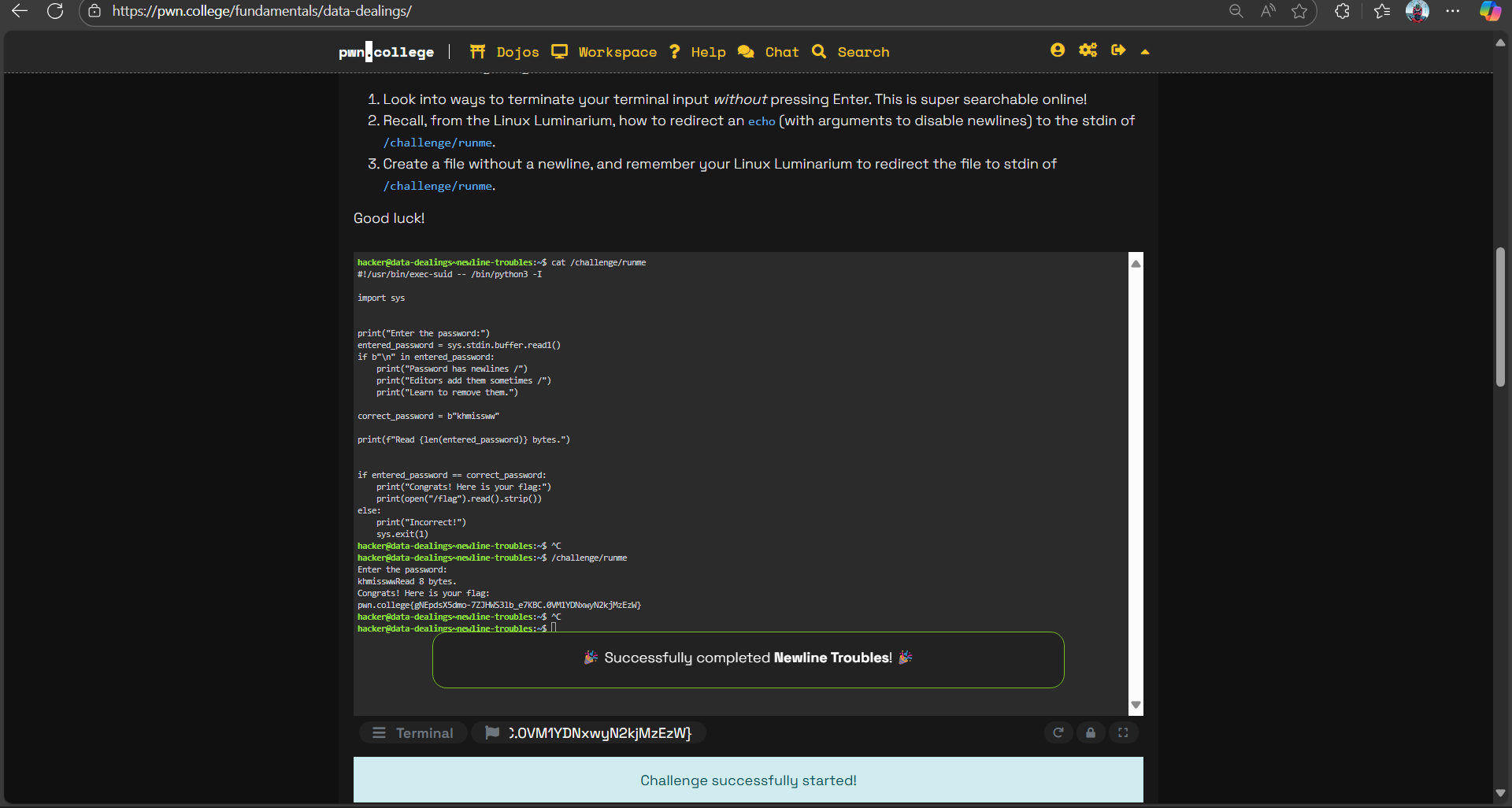
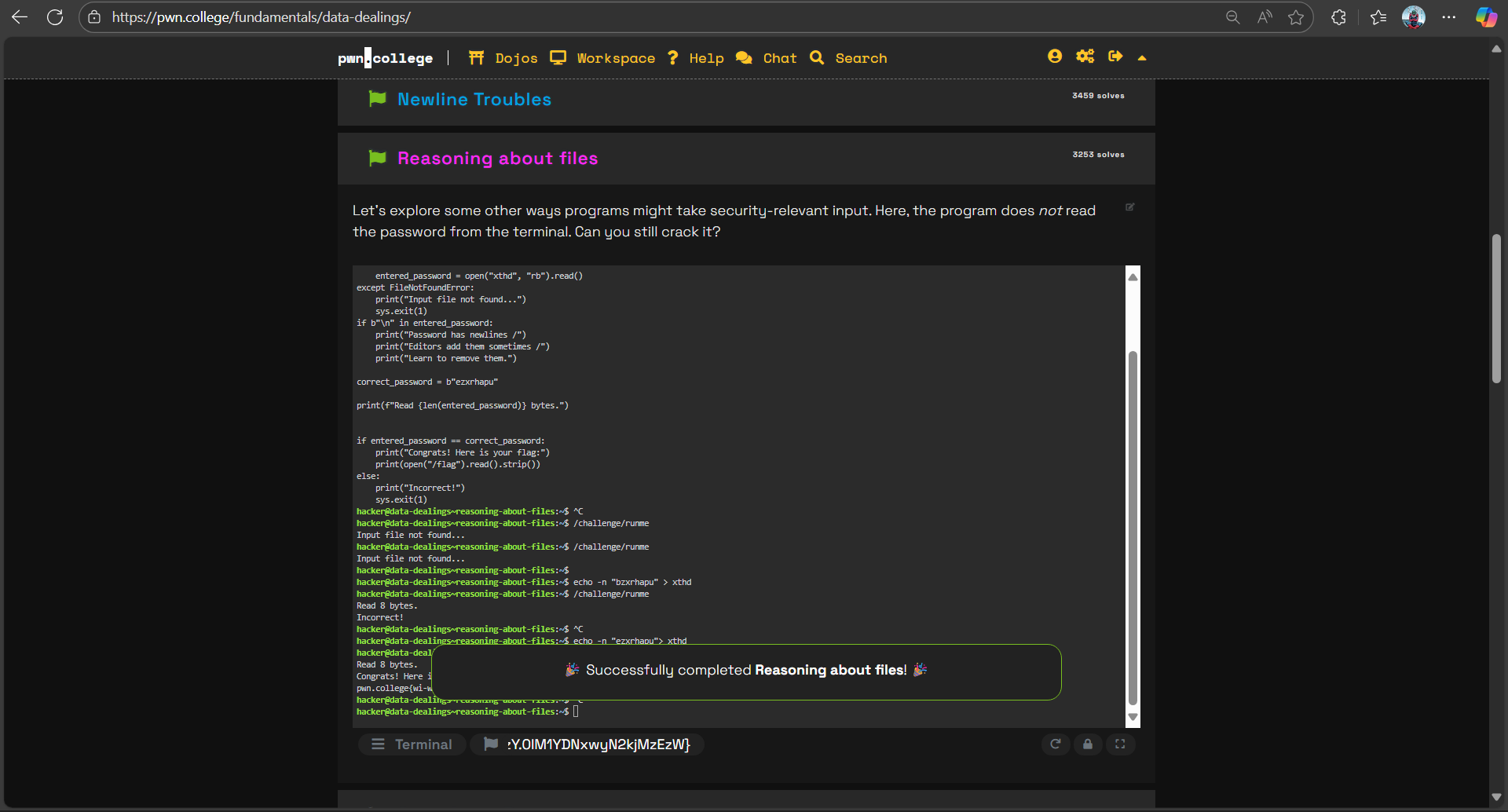
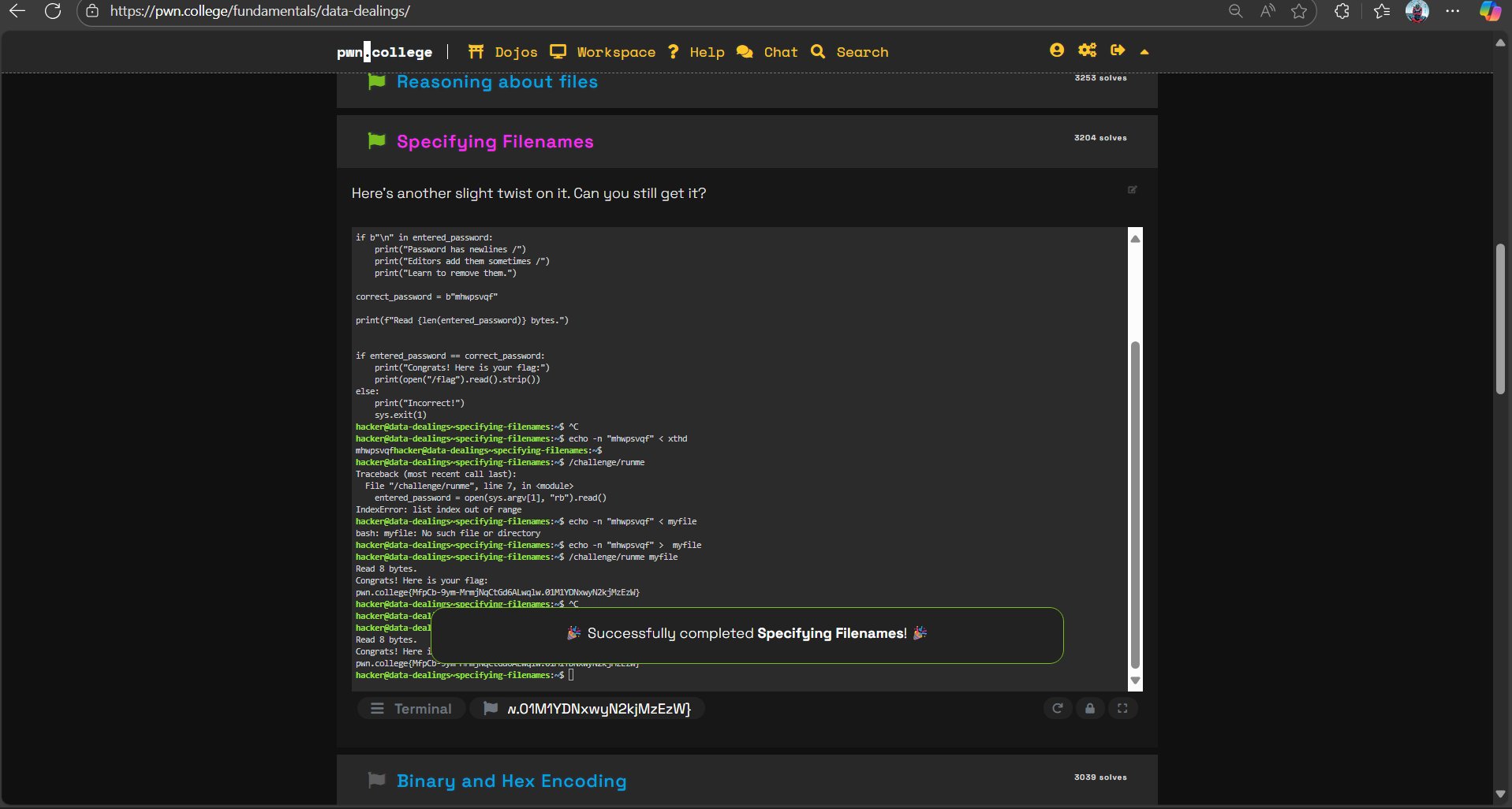
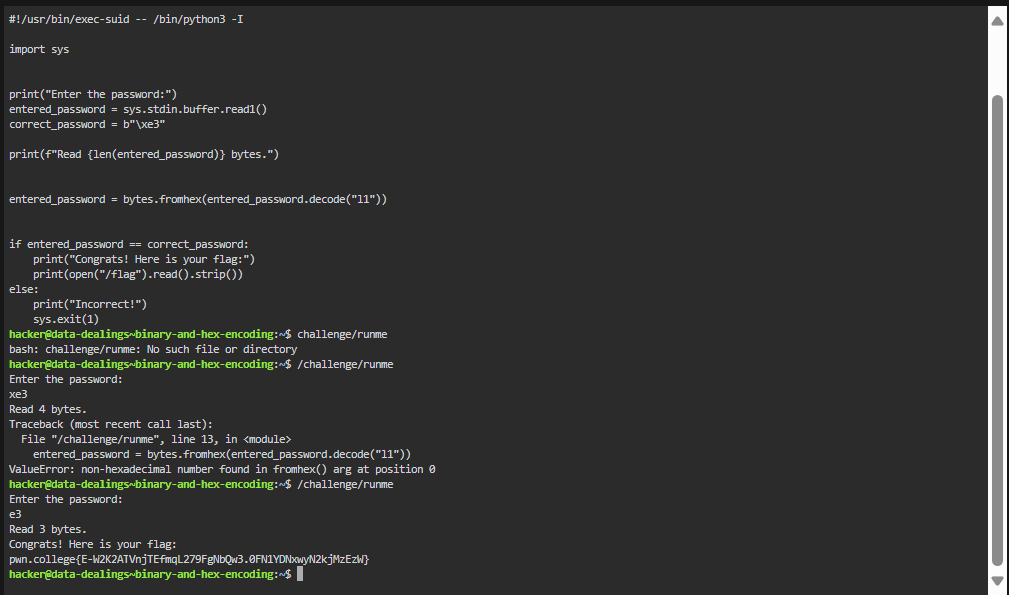
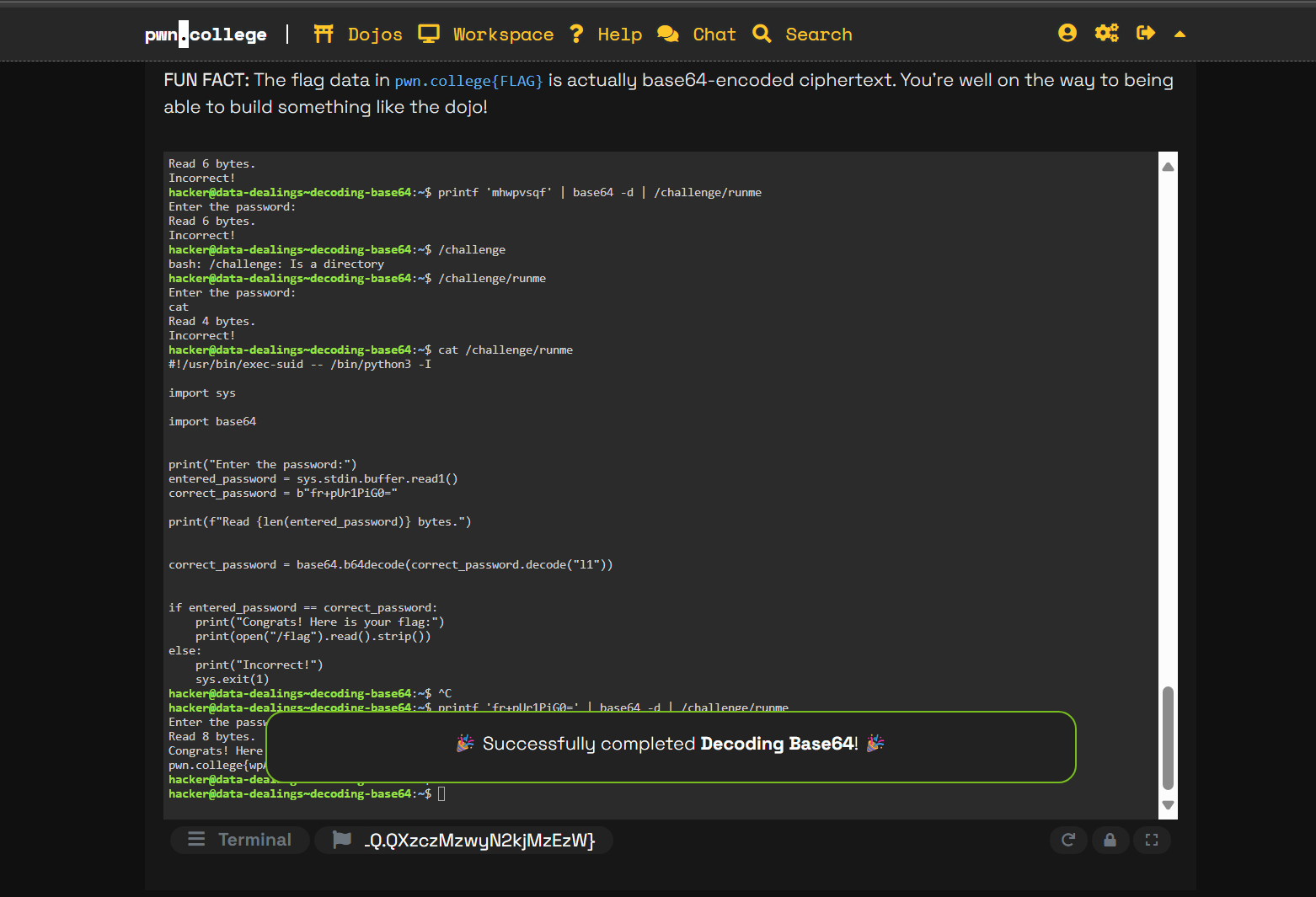
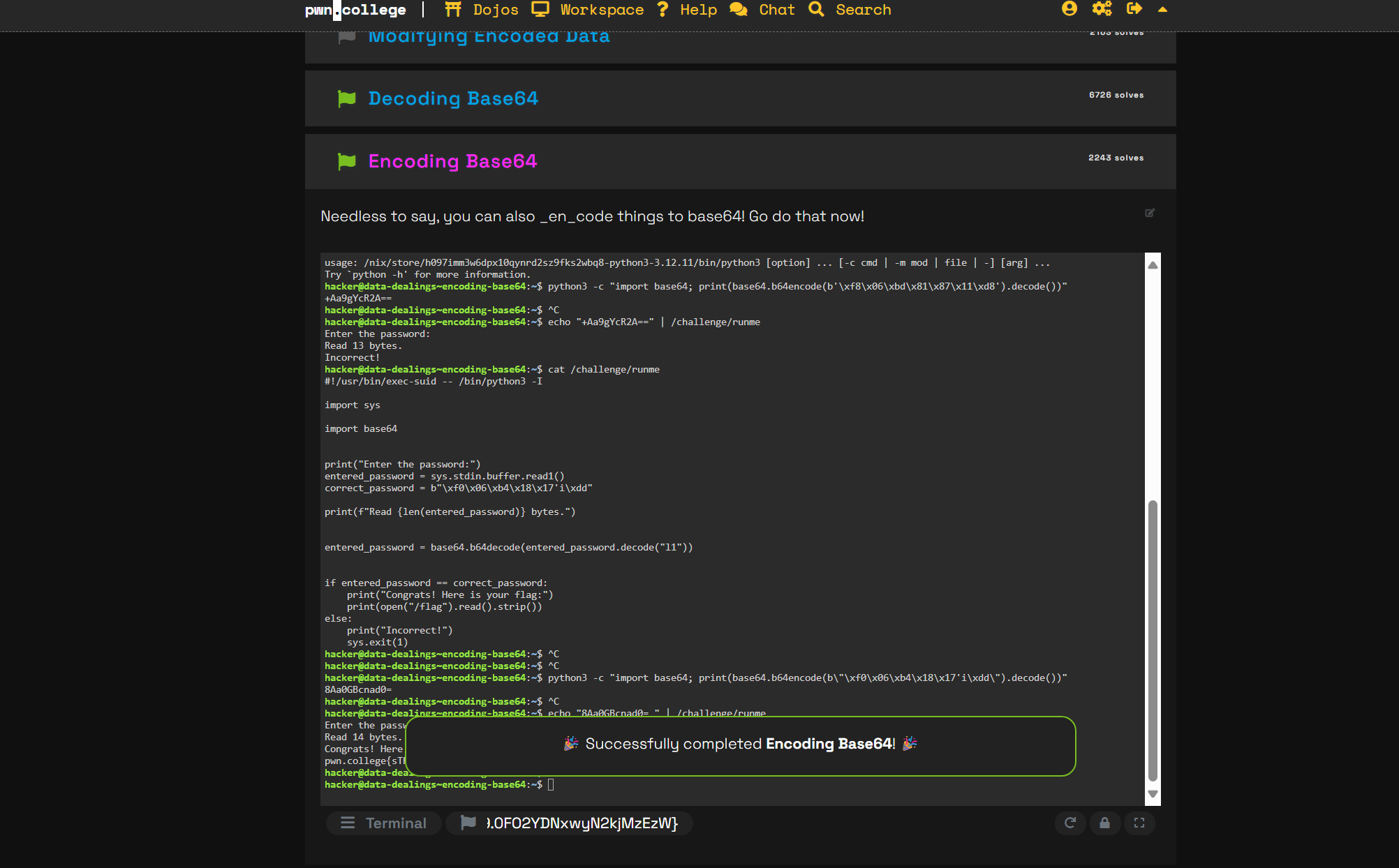
Username: Carlos

CSCI 400-01

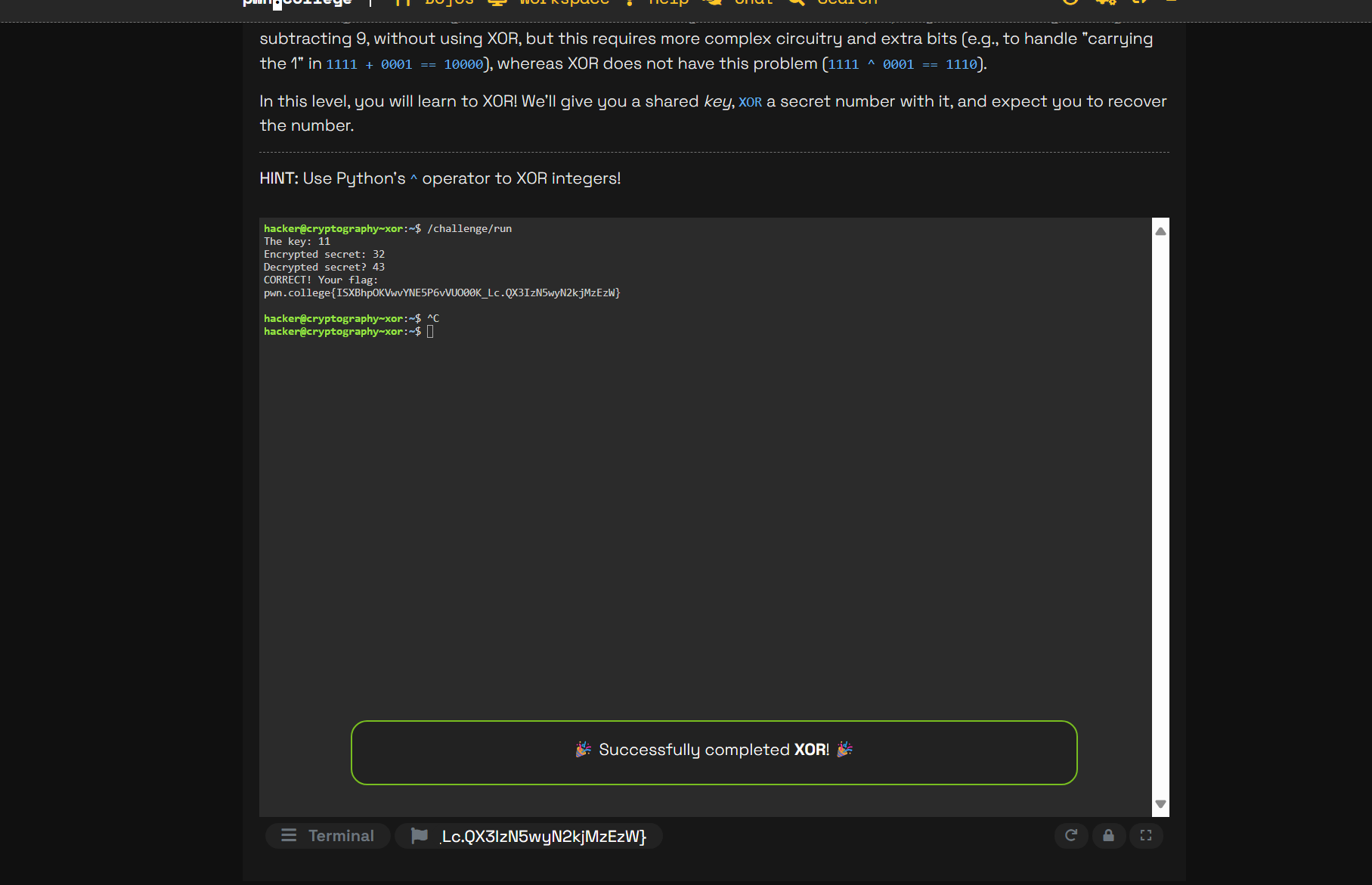
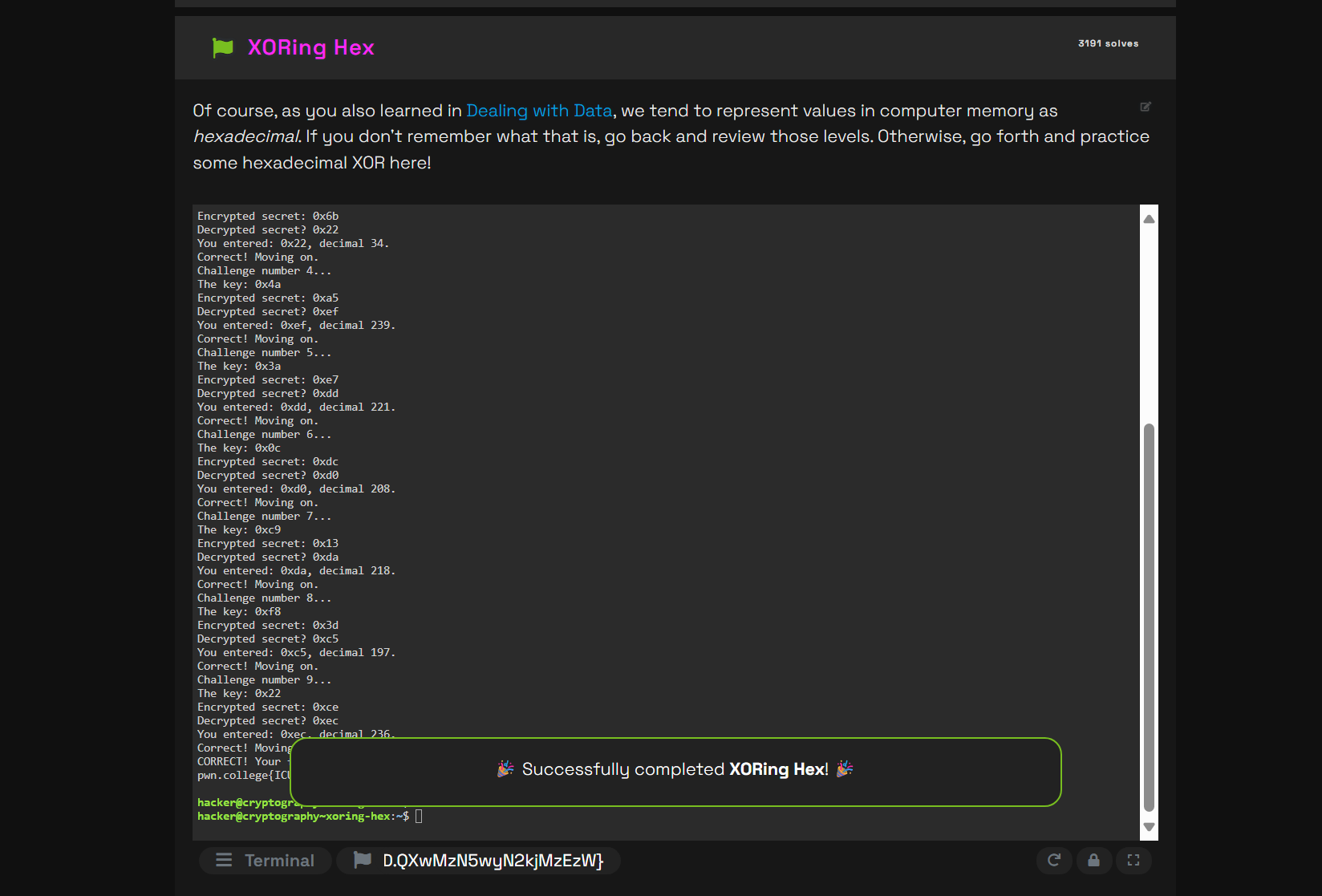
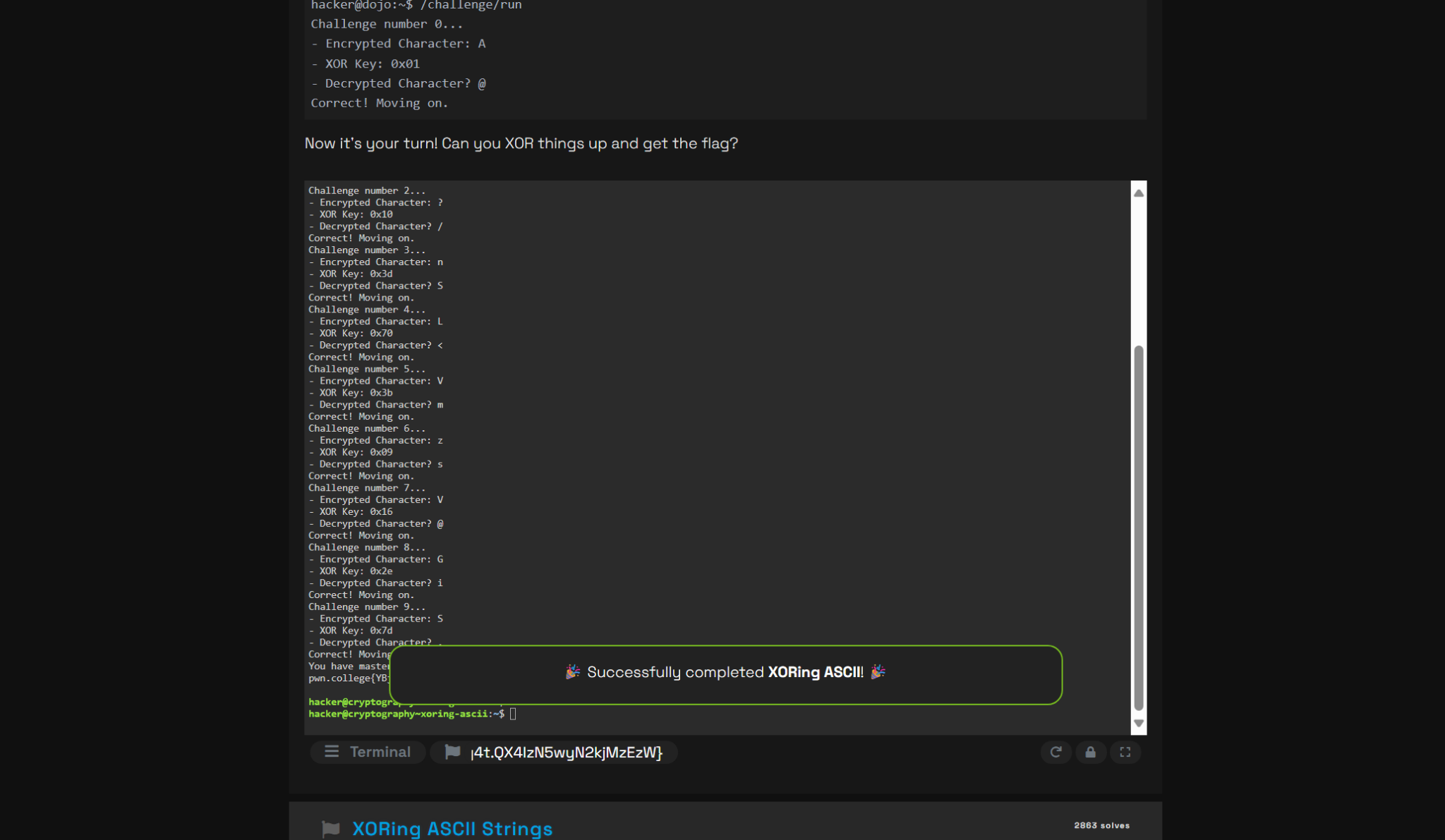
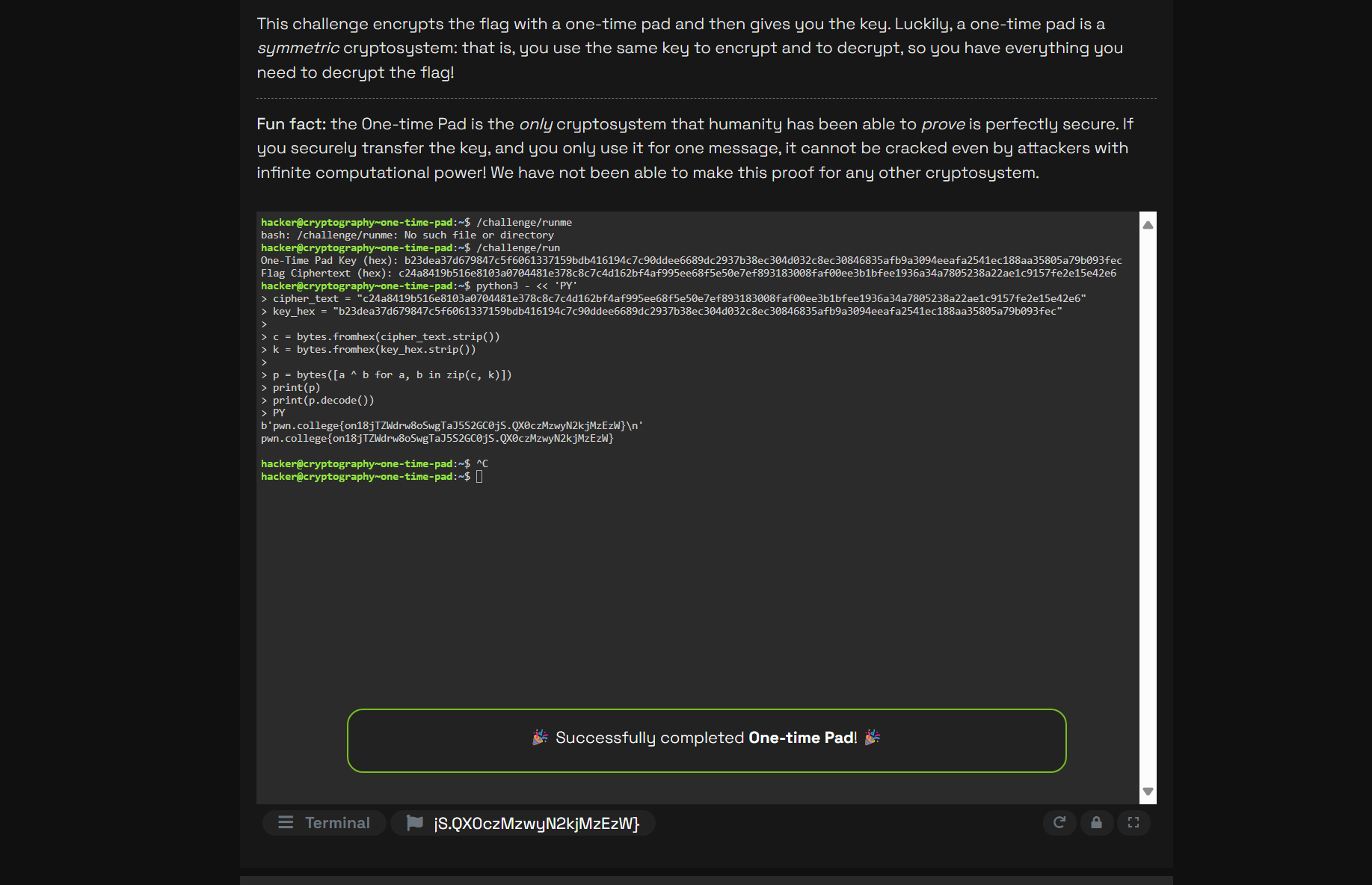
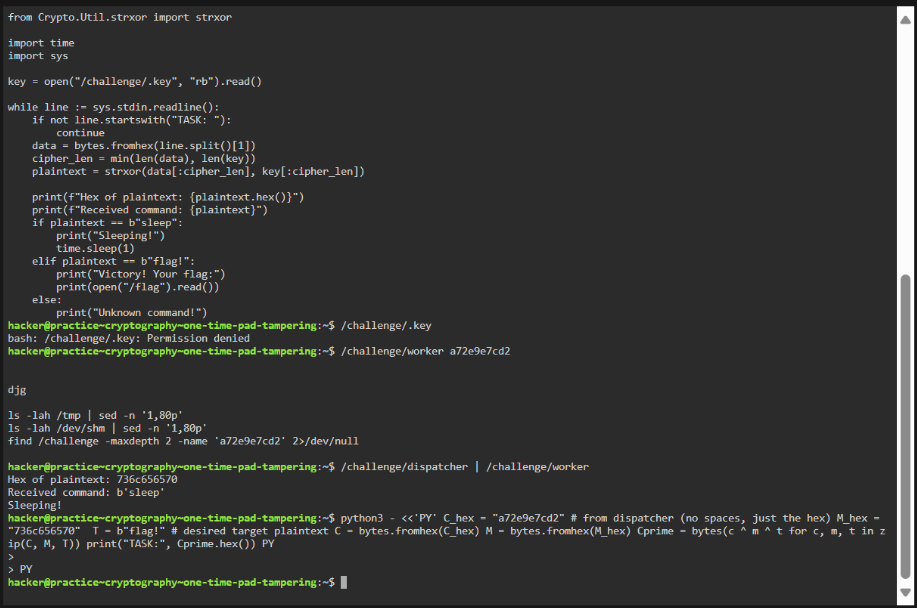
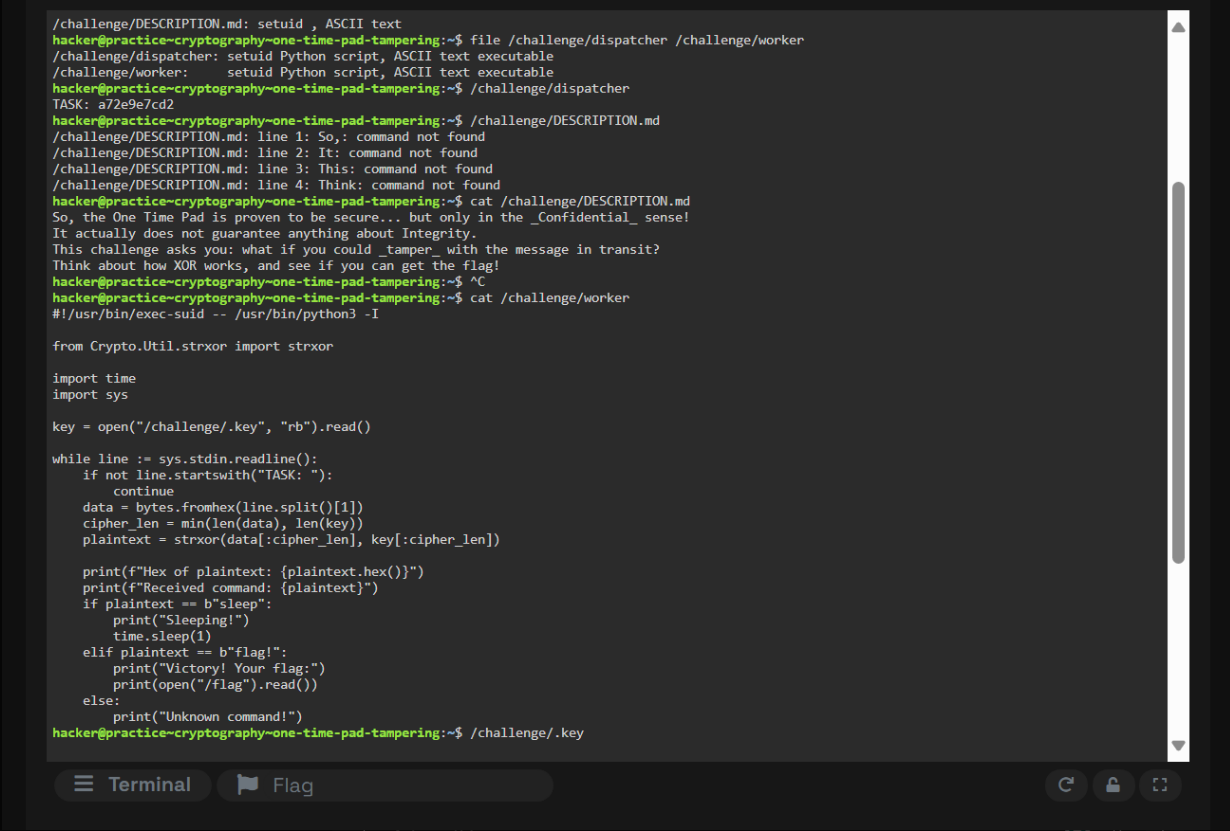
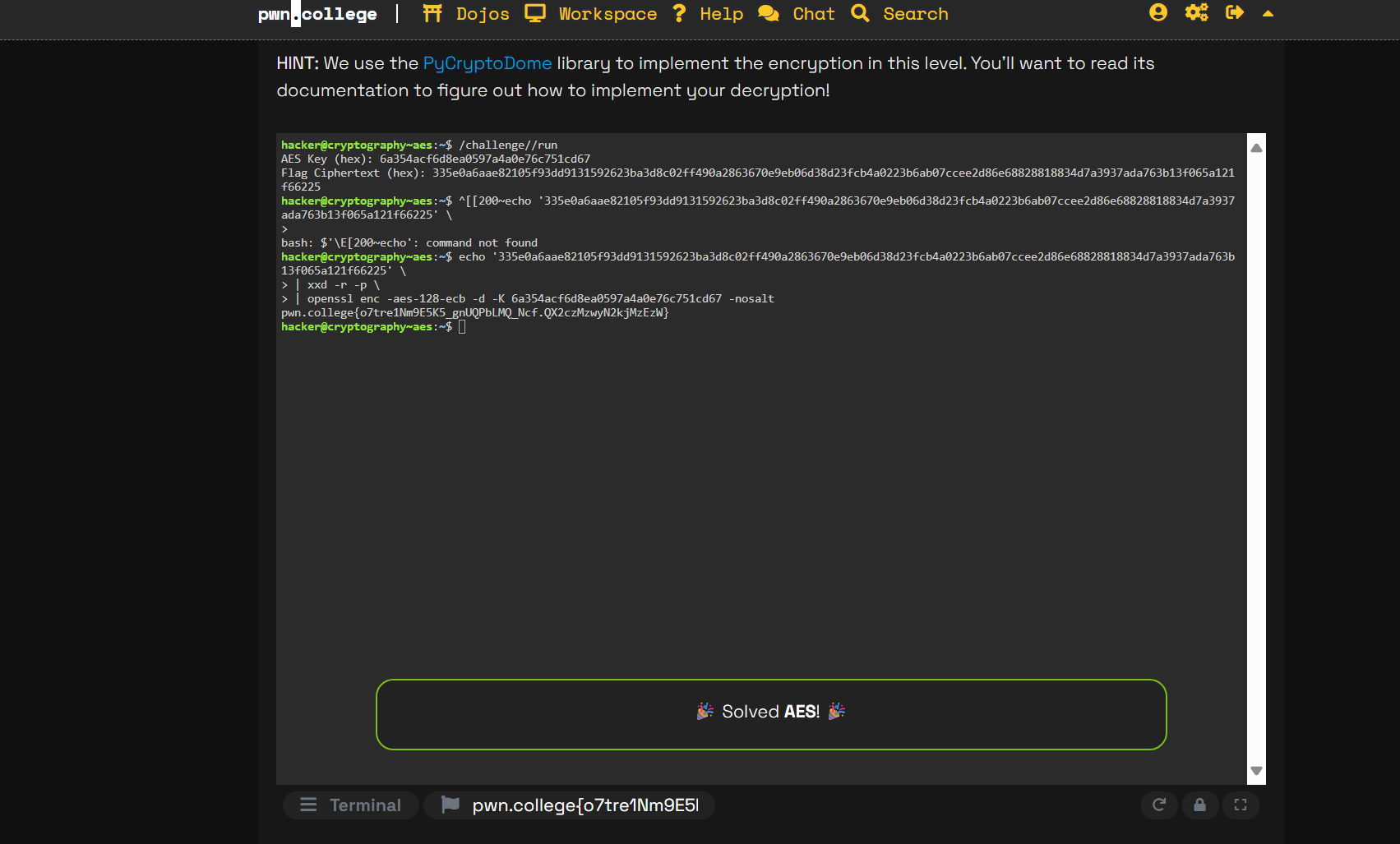
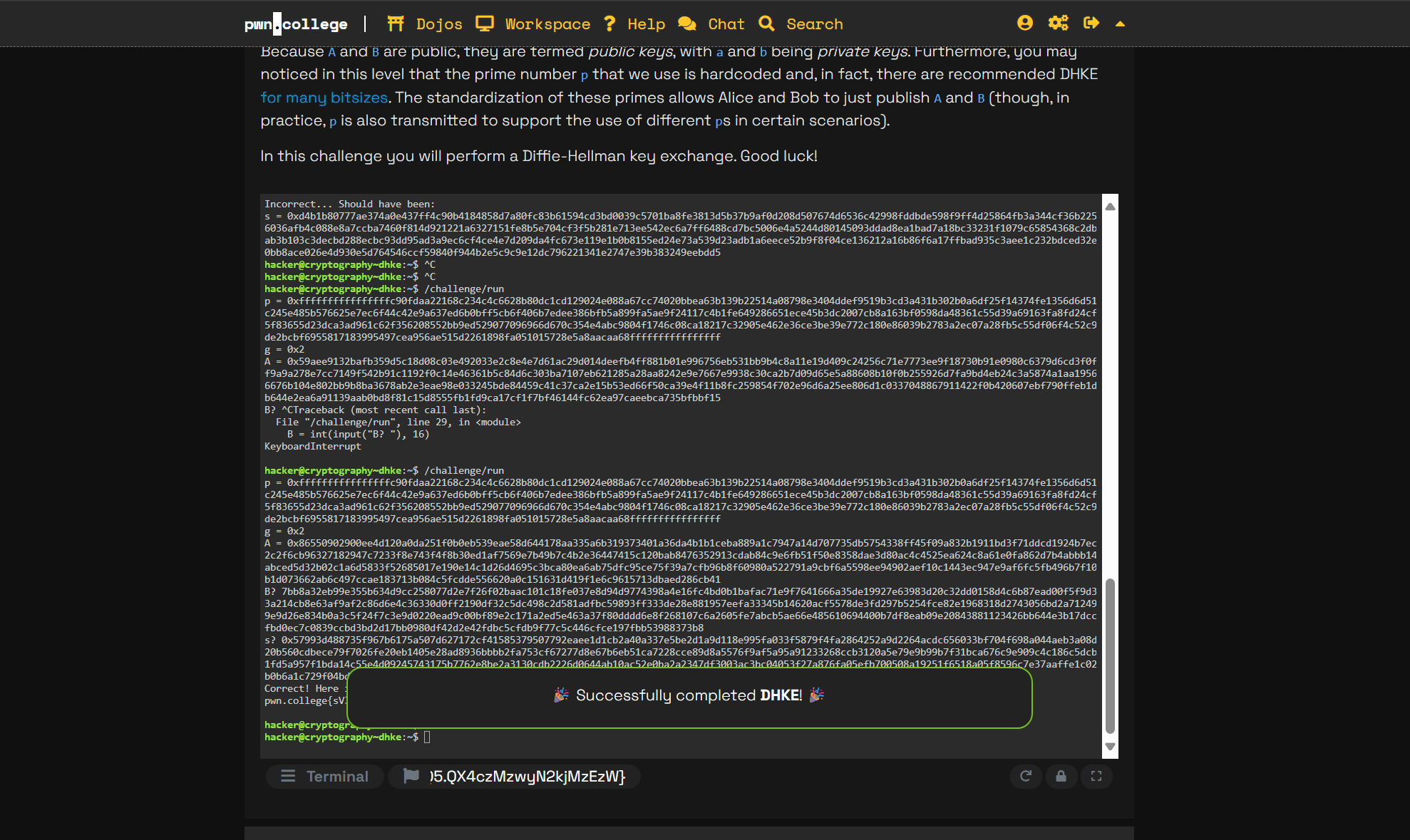
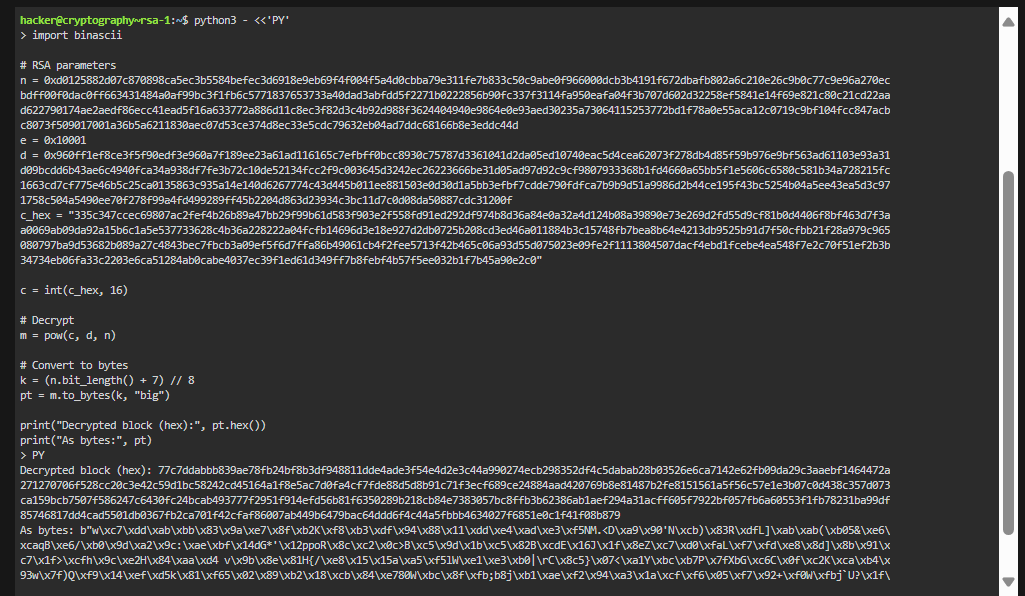
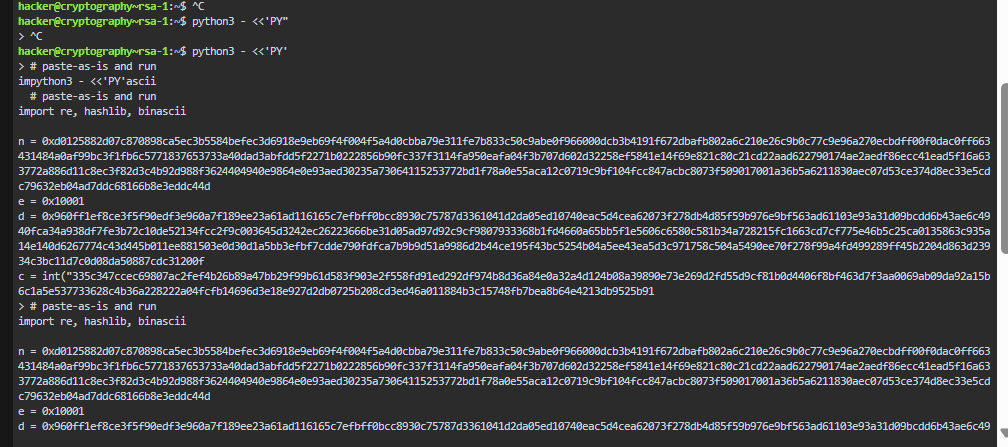
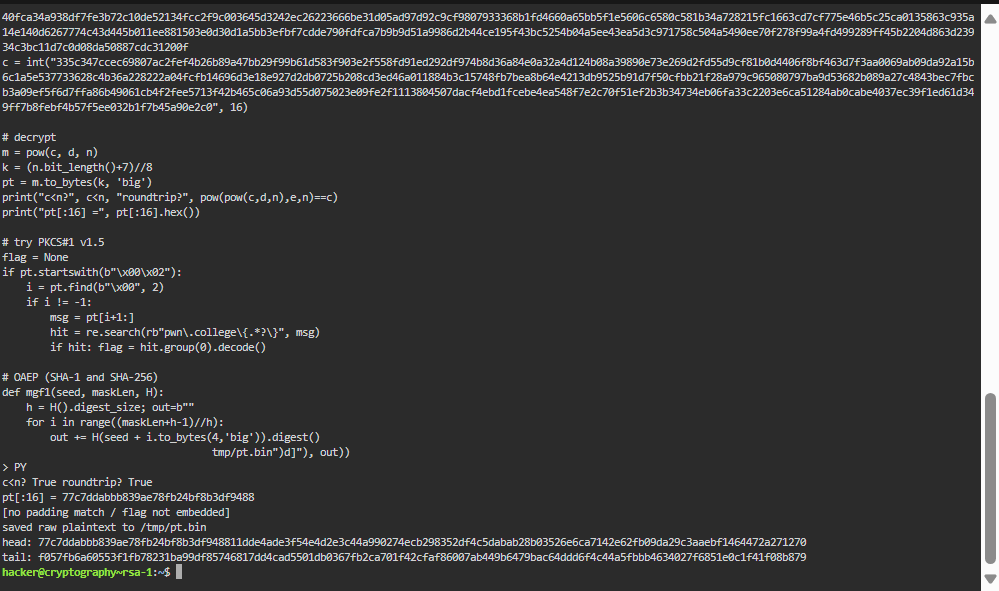
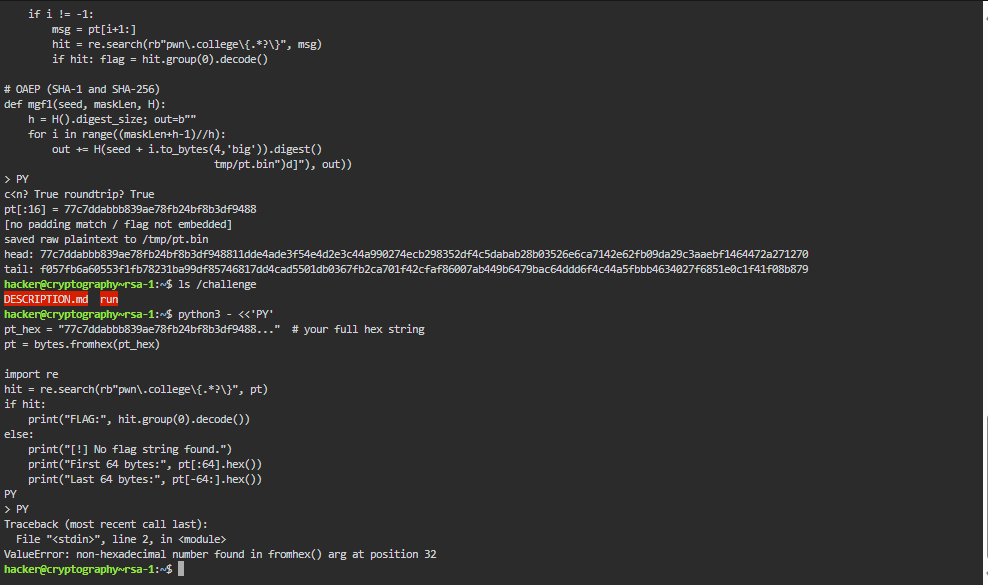
9/8/25

Lab 2

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

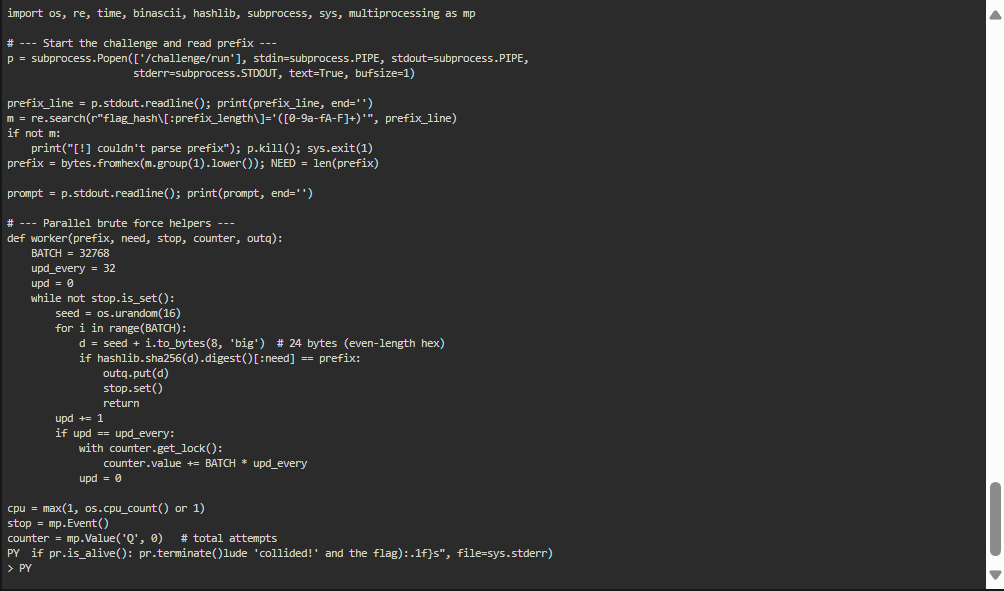
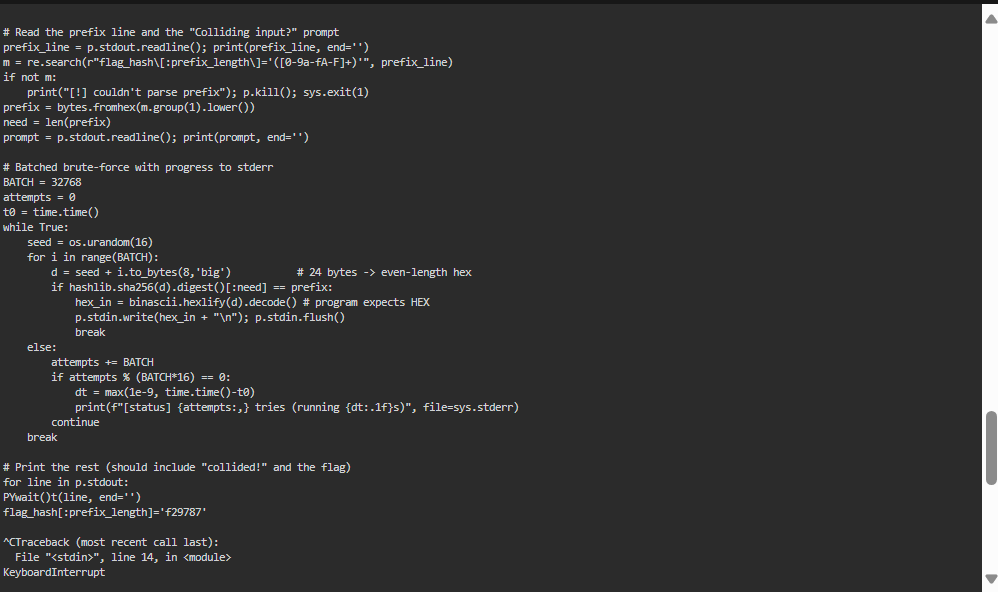
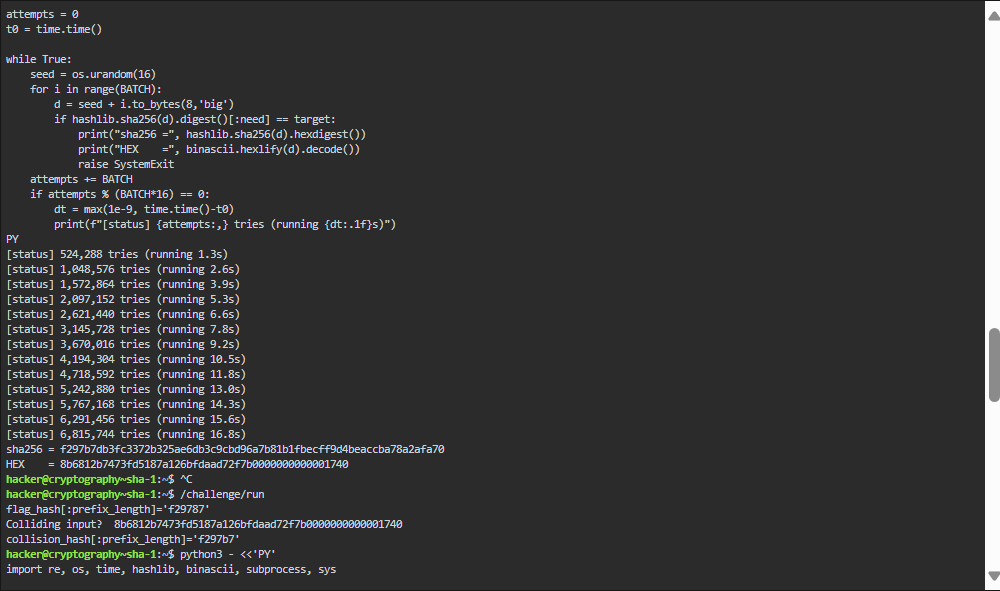
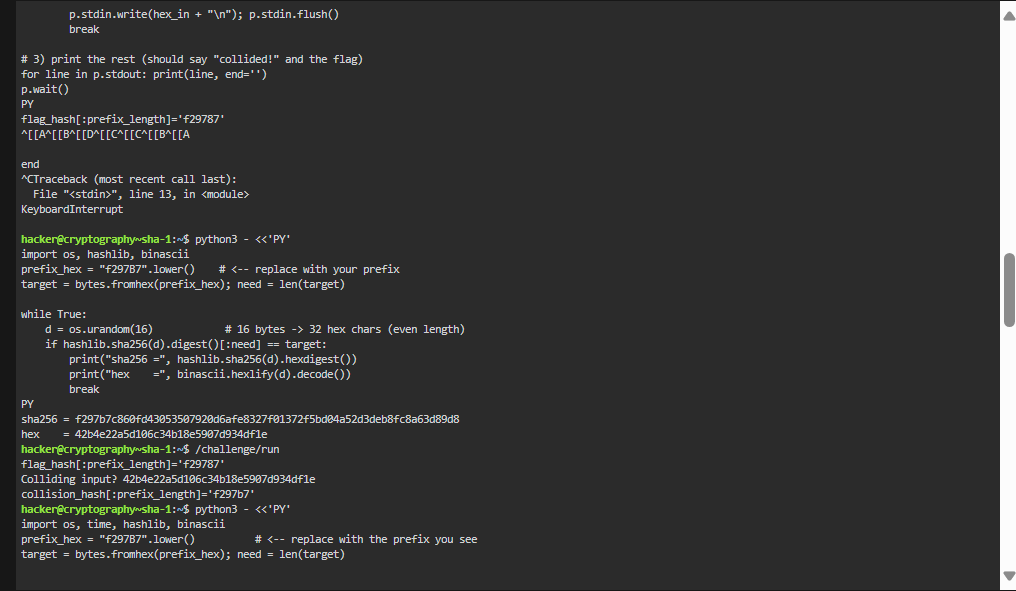
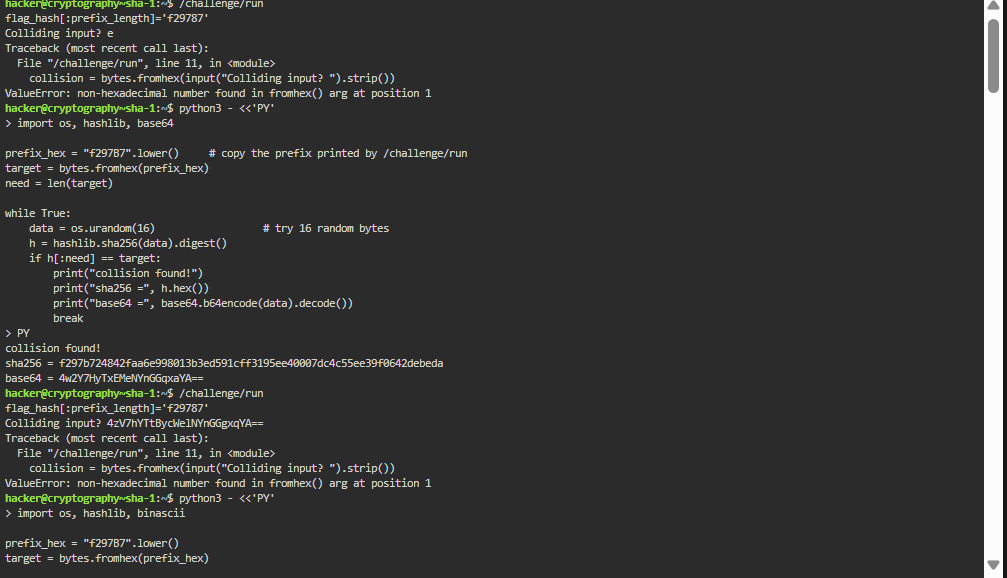
1. What's the password
   1. 
   2. To determine what password the program was comparing to, I opened and read the Python file located at /challenge/runme for this challenge. I located the hard-coded string in the code and entered it as the input, revealing the flag correctly.
2. ... and again!
   1. 
   2. I applied what I had learnt in the previous challenge to this one, which was to open and read the Python file at /challenge/runme to find out what password the software was comparing it to. I located the hard-coded string in the code and entered it as the input, revealing the flag correctly.
3. Newline troubles
   1. 
   2. The issue with this challenge was that the password failed the check since hitting Enter added a newline to it. In order for the application to accept it and return the flag, I had to type the password and then hit Ctrl+D to transmit it without a newline.
4. Reasoning about files
   1. 
   2. For this challenge, I noticed the program was not asking me for input but instead trying to read from a file named xthd. I created that file with the correct password ezxrhapu using echo -n "ezxrhapu" > xthd, then ran the program, and it successfully matched the password and revealed the flag.
5. Specifying filenames
   1. 
   2. The program will read the password from the file after expecting a filename as a command-line option for this challenge. To enable the application to validate and return the flag, I generated a file with the right password, mhwpsvqf, and ran /challenge/runme myfile.
6. Binary and hex encoding
   1. 
   2. I observed that the program employed bytes to compare the input to the byte \xe3 for this challenge. fromhex(), which required that the input be in hexadecimal format. By inputting e3 at the prompt, which was accurately translated into the byte value and displayed the flag, I was able to fix it.
7. Decoding Base64
   1. 
   2. I used cat to directly analyze the contents of /challenge/runme and discovered an embedded base64-encoded password within the file. I extracted this password and then decoded it into the right plaintext value using the base64 -d tool. Ultimately, I provided the program with the deciphered password, and it successfully displayed the flag.
8. Encoding Base64
   1. 
      1. The right password was found in the raw bytes \xf0\x06\xb4\x18\x17'i\xdd after I examined the challenge program. I used Python's base64.b64encode method to encode these bytes into base64 in order to satisfy the program's requirements. When the resultant base64 string was sent to /challenge/runme, the application successfully verified the input and displayed the flag.

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

1. XOR
   1. 
   2. In order to show the self-inverse property of XOR in practice, a key and an encrypted secret were supplied at this XOR level. The challenge was finished with the flag successfully retrieved when the original value was exposed by applying the provided key to the encrypted secret.
2. XORing Hex
   1. 
   2. For the XORing Hex challenge, I calculated the decrypted value by XORing the given key and the encrypted secret in hexadecimal and entered the results each round. The program confirmed my answers with “Correct! Moving on.” several times in a row, showing that my calculations were correct and continuing until I completed 9 correct calculations in a row.
3. XORing ASCII
   1. 
   2. I completed roughly 10 rounds of practice for the XOR ASCII challenge, in which I was given an encrypted character and a decoding key for each query. I was able to reliably provide the right decrypted characters at each stage by continually converting the character to its ASCII value, XORing it with the provided key, and then translating it back.
4. One-time Pad
   1. 
   2. I received the ciphertext in hexadecimal format along with the key for the One-Time Pad challenge. Since one-time pad decryption is the exact opposite of encryption, I first transformed each one from hex to raw bytes, and then I XORed the ciphertext with the key byte by byte. The plaintext result from running this operation in the terminal showed the flag and verified that the challenge had been successfully resolved.
5. One-time Pad Tampering
   1. Z
   2. Z
   3. Z
   4. Z
   5. 
   6. 
   7. In this one-time pad tampering challenge, I was not able to reach the final step of obtaining the flag. The instructions made it clear that the goal was to manipulate the ciphertext using XOR so the decrypted message would change into the target. I attempted to run the provided challenge binary but received errors that the file or directory did not exist. After that, I tried different scripts and approaches to locate the correct dispatcher and worker programs, but I could not find them in the challenge directory. Even though I researched how XOR tampering works and wrote scripts to flip the ciphertext into “flag!”, the missing or misidentified executables prevented me from testing the solution. Despite repeated effort and experimentation, I was unable to complete the challenge and did not get the flag.
6. AES
   1. 
   2. From the challenge, I obtained the ciphertext in hexadecimal format and the AES key. I applied AES-128 in ECB mode with the supplied key after converting the hex ciphertext into raw bytes using the OpenSSL command-line tool. The task was successfully completed when the plaintext flag was discovered through decryption.
7. DHKE
   1. 
   2. I completed the task and replicated the high values of p, g, and A that it provided. I created our own private key in the terminal using those numbers, and then I computed a legitimate public key, B, that passed the size check. To finish the assignment, next I calculated the shared secret s, formatted it in the necessary s = 0x... style, and pasted both outcomes back into the challenge prompts.
8. RSA 1
   1. 
   2. 
   3. 
   4. 
   5. In this RSA challenge, decryption was attempted using the given values for the ciphertext, public modulus, public exponent, and private exponent. The ciphertext was raised to the power of the private exponent modulo the public modulus, as per the usual RSA equation. To check if the flag was directly included in the plaintext, the result was transformed into a block of bytes. Although the bytes showed up as random data rather than a legible string, the block consistently gave results that verified the math was right. The decrypted block lacked the common signs of padding, like an OAEP structure or a 00 02 prefix for PKCS#1 v1.5.

To extract the flag from this output, several methods were tried. Using common hash functions, scripts were developed to try PKCS#1 v1.5 and OAEP unpadding after searching the block for identifiable flag patterns. Despite not finding a useful message, each approach verified that the decryption step was legitimate. This suggested a discrepancy between the key values and the ciphertext or the need for a step beyond straightforward decryption.

The decrypted block failed to provide the flag even after investigating RSA padding strategies and experimenting with various unpadding techniques. The challenge was still unfinished, but the effort showed that RSA decryption was applied correctly and that the math was carefully tested. This result shows the difficulty of applied cryptography exercises, where managing padding, encodings, or other challenge-specific criteria may be necessary in addition to straightforward decryption.

1. SHA 1
   1. 
   2. 
   3. 
   4. 
   5. 
   6. This is a brief summary of our attempts and the SHA-256 partial-collision challenge. Flag\_hash[:prefix\_length]='f297B7' is the line that the task prints, indicating that only the first three bytes of SHA256(input) must match that prefix. We verified that it does collision = bytes by looking at the source of /challenge/run. fromhex(input().strip()) and then validates sha256(collision).hexdigest()[:prefix\_length]. That detail mattered: the program expects HEX-encoded bytes , not raw text or base64. Early on we inadvertently pasted base64 and the binary reported ValueError: non-hexadecimal number encountered in fromhex(). We fixed that by exporting hex from our finder and assuring even-length hex strings.

I developed many brute-forcers for the attack side, including a minimum single-try loop, a parallel version that utilizes all CPU cores, and a batched version that generates a seed and tries 32K counters with periodic status logs. In addition, I found an operational bug: the colliding hex must be sent to the same /challenge/run process that printed the prefix (so the program prints(flag)), not a different run. To (1) launch /challenge/run, (2) parse the prefix, and (3) brute-force till sha256(d), we instrumented a one-shot script. begins with (prefix), and (4) promptly returns the hex to that process. We added progress meters , fixed heredoc issues , and confirmed that the candidates we identified truly generated the necessary prefix.

I was unable to grab the flag before the deadline in spite of my attempts. I ran out of time after trying a number of different approaches, including manual hex submission, single-core, batched, and multi-process search; rerunning with new prefixes; investigating base64 vs. hex input; and using the program's internal check. Finding a 24-bit SHA-256 prefix match still takes an average of about 2²⁴ = 16.7M trials; on a limited virtual machine, a single core can take several minutes, and even parallelized runs can be unlucky. This was the only practical obstacle left. Now that the method and input format have been verified to be accurate, the next step would be to either run the parallel one-shot on a machine with more cores or simply let it run until it finds a match.