→ What's The Program? (Password):-

Given is a Python program, 'Cat' it and read it 'understand' the correct input and use it to crack the Program.

→ ...and again!:-

Once again do same as above.

→ Newline Troubles:-

Now, the program does one thing different. It does not ignore 'Enter' that we press on terminal when entering password. This causes our entered_pass to contain a new line, and since correct_pass has no newline, comparision fails.
This is called as 'Errant Delimiters in Data', happens all time & leads to crazy ammount of lost time.
we did,

printf "password" > new_line_troubles
/chall/..... < new_line_troubles

→ Reasoning about files:-

Now, the Program does not read password from terminal. :)-

If we look at code properly, it takes input directly from a file named 'ypdmnop', so let's save password in a file named 'ypdmnop' and then give it as stdin or run it directly, so it will look for this file, cat it and read pass from it.

entered_password = open("ypdmnop", "rb").read().

→ **Specifying Filenames**

Now, we have another twist...

we run file as an argument.

→ **Binary & Hex Encoding:**

Python has two types of strings-like constraints:

strings 'str' ("abcd") and bytes 'b' (b"abcd").

Bytes are what is actually stored in computer's memory.

Bytes are based on binary, where 8 bits = 1 byte, however humans understand decimals (0 to 9) so in binary everything is considered in Power & base of '2' (0 & 1). Ten values are represented roughly by $\log_2(10) = 3.3219...$ bits and we got weird situation, where binary '1001' is '9' but binary '1100' (still 4 bits) being '12' (two decimal digits) i.e. Decimal does not have clean bit boundaries.

This lack of bit boundaries make reasoning about relationship between binary & decimal complex.

Therefore, we use Hex (Hexadecimal) for showing relationship between Decimal & Hex is easy.

Hex ⇄ binary ⇄ Decimal.

for numerical constant's Python's notation is to prepend '0b' for binary, '0x' for hexadecimal and only no's for decimal.

i.e. 11 = 0b1011 = 0xb

3 = 0b0011 = 0x3

17 = 0b10001 = 0x11

Some useful knowledge about Python:

→ If we print(n) a no. or convert it into string with str(n), number will be represented in base 10 (Decimal).

→ We can get hexadecimal string representation of number using hex(n).

→ We can get binary representation of no. using 'bin(n)'.

→ Converting string to no. using int(s) will read it as base 10 no. by default.

→ We can convert string in our desirable system using second argument of base (2, 16, 10).

int (s, 16) → hex

int (s, 2) → bin

int (s, 10) → dec. (default)

→ We can auto-identify number base using int (s, 0), which require prefix on string. (0b → bin, 0x → hex, nothing → dec).

Now, just give hex values as input as it only accepts '\' hex value, ignore escape character '\x'.

→ More Hex:

2 hex digits ⇒ 1 bytes.
1 hex digits → Nibble.

Now, we have to, figure out what value we want our data to have at end, encode that value in hex, and send hex bytes.

→ Decoding Hex:

Now, we will decode hex values;
for this, we have to send raw binary data to programs stdin.. There are few ways to do this:

1. Write python script to output data to stdout and pipe that to challenge's stdin. This would involve using raw byte interface to stdout: sys.stdout.buffer.write.
2. Write python script to run challenge & interact with it directly.
3. for an increasingly hacky solution, echo -e -n "\xAA\xBB" will print out bytes to stdout that we can pipe.

→ Decoding Practice:

Now, let's Practice decoding different bases..

→ Encoding Practice:

Now let's do encoding.

→ Hex-encoding ASCII:

In Python we convert 'str' (string) into its equivalent bytes by doing 'my_string.encode()'. Similarly to decode bytes into string 'my_string.decode()'.
We use ASCII To map characters to byte values.
. Every ASCII character is of 1 Byte.
Uppercase letters are '0x40+letter_index'. eg: A is '0x41', 'F' is '0x46, Z is '0x5a'.
lowercase letters are '0x60+letter_index' eg: a is '0x61', f is '0x66', z is '0x7a'.
The numeric characters we see are not bytes of those values, they are ASCII encoded number character. '0x30+number', so 0 is '0x30', 7 is '0x37'.
Useful special characters are also used to map;
eg: '/' is '0x2f', space is '0x20', newline is '0x0a'
We can see manual of ASCII by using 'man ascii'.
We can use ascii to encode our strings in Python, using 'my_string.encode("ascii")'.
Standard ASCII doesn't define values above '0x80'
We will get exception for decoding.
eg: b"\x80".decode("ascii") wont work.

→ Nested Encoding:

Now, we will encode our strings and numbers multiple time. (This causes garbage values most of the times).

→ Hex-encoding UTF-8:

As, computing went international, emojis were added, People needed to be able to use more than 256 possible characters at a time. Hence, 'UTF-8' encoding comes in picture. UTF-8 is specific multi-byte encoding of Unicode, a globalized standard character set containing essentially all characters known to humanity, plus emojis. UTF-8 is one of many ways to encode into Unicode.

UTF-8 is backward-compatible with standard ASCII. i.e. UTF-8 values have same coded values as ASCII. UTF-8 supports over 1,000,000 characters.

UTF-8 is by default, how python strings are specified in. we can add emoji's into python as strings and they will be converted into byte representation.

Now, we will learn to craft emoji bytes. We have to create raw bytes representing UTF-8 emoji, hex encode them and send to program.

→ UTF Mixups:

UTF-8 is used by majority of websites on internet. But outside web, other encodings are present in significant numbers. For various (misguided) reasons, Windows systems often use a different Unicode encoding: UTF-16. This encoding represents same Unicode characters using different byte values. This causes much confusion and occasionally, Security Vulnerabilities.

A common way encoding mixups leads to security vulnerabilities is by incorrectly decoding data to perform security checks, then correctly (differently) decoding it later to actually carry out securitysensitive actions. If security checks are performed on bad data, then dangerous data can be missed.

→ Modifying encoded-data:

Till now, we have seen a few types of encoding: UTF-8, UTF-16, extended ASCII (latin1), and hex encoding. But if we encode any emoji or text into UTF-8 and while decoding mess up or modify encoded code, we get different output or sometimes errors as, some bytes can be decoded properly. For UTF-8 this is due to complex algorithm to specify data. For hex encoding this is due to only numbers '0' through '9' and letters 'A' through 'F' being valid in hex.

When a security flaw allows data to be corrupted, this can enable an attacker to carefully transform data to their purposes. We will learn this about how to protect data from this later, but for now, this is it.

→ Decoding Base64:

ASCII and UTF-8 are encodings meant for very specific Data i.e. for text. Hex encoding is more general, and we can apply it to any data. It can be used to transfer information via some medium where it is hard to write arbitrary binary code, such as piece of paper or certain communication Protocols. It is however, inefficient as it doubles size of data by outputting two ASCII hex digits for every byte.

'Base64' comes from the fact that there are '64 characters' used in each output character. This may vary, but standard base64 encoding uses an 'alphabet' of uppercase letters 'A through Z', lowercase letters 'a to z', digits '0 to 9' and '+' and '/' symbols. This results in 64 output symbols, and each symbol can encode 2^6 possible input symbols, or 6 bits of data.

This means, to encode '1 byte' (8 bits) we need more than one base 64 output character. In fact we need two: One that encodes the first '6 bits' and one that encodes remaining '2 bits' (with 4 bits of second output character being unused). For marking, base64 encoded data appends an '=' for every two unused bits.

eg: echo -n A | base 64

QQ==

echo -n AA | base64

QUE=

echo -n AAA | base64

QUFB

echo -n AAAA | base64

QUFBQQ==

→ Encoding base 64:

Now, lets encode in base 64.

→ **Dealing with Obfuscation:**

Security oblivious developers often use encoding-based obfuscation in lieu of encryption. This type of obfuscation typically fails to prevent determined hackers from accessing data in question, especially once they read software logic implementing it.