ECSC Estonia Prequalifier - Roll your own crypto

Challenge source code:

```
#!/usr/bin/env python3
import utils

with open("flag.txt", "rb") as f:
    Flag = f.read()

alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz.,!?
_{}0123456789"
for x in Flag.decode():
    assert x in alphabet


# list of cyberchef approved encryption methods
TRANSFORMS = {}
TRANSFORMS["b64encode"] = utils.make_b64enc
TRANSFORMS["b64decode"] = utils.make_b64dec
TRANSFORMS["hex"] = utils.make_hex
TRANSFORMS["bytes"] = utils.make_bytes
TRANSFORMS["encode"] = utils.make_encode
TRANSFORMS["decode"] = utils.make_decode
TRANSFORMS["xor"] = utils.make_xor
TRANSFORMS["md5"] = utils.make_md5


if __name__ == '__main__':
    print("Welcome to the DDC Crypto Challenge Creation Tool!")
    print("Lets make a high quality challenge!")

    # bytestring of the flag
    flag = Flag

    for _ in range(200):
        action = input("What should we do to the flag?")

        # finished!
        if action == "done":
            break

        # Transform should exist
```

```python
        if action not in TRANSFORMS:
            print('Invalid transform! Try again.')
            continue

        # Apply the Transform
        if action == "xor":
            xorval = input("what would you like to xor the message with?(hex):")
            # cyberchef kinda chugs for cribs longer than 3
            xorval = bytes.fromhex(xorval)[:3]
            succ, msg = TRANSFORMS[action](flag, xorval)
        else:
            succ, msg = TRANSFORMS[action](flag)

        print(msg[:25], msg[-25:])
        print(set(msg))
        print(all(msg[i] == msg[0] for i in range(len(msg) - 1)))
        print(len(msg))
        if type(msg) == bytes:
            print(msg.count(b'x'))
        # your transform should succeed
        if not succ:
            print(msg)
            exit()

        # pls no cheesey unsolveable challenges
        if len(msg) < len(Flag):
            print('How are the players meant to solve it if the ciphertext is shorter than the flag???')
            print(f'Your challenge has a ciphertext which is {len(msg)} bytes long, but the flag was {len(Flag)}')
            exit()

        # LETS GOOOOOO
        flag = msg
        print(f"you successfully applied {action} to the flag! This challenge is looking great so far!")


    # Nicely done! Just some formalities left.
    print("Well done, you've created a crypto challenge!!!")
    flag_guess = input("Lets just confirm that you ended with the same ciphertext I did: ")

    if len(flag_guess) != len(flag):
        print(f"Oh no, thats not even the right length... By my math the encryption should be {len(flag)} characters.")
        exit()

    if flag_guess != flag:
        print("Wait that's not right, lets try again")
        exit()

    print(f"Nice! Lets deploy the challenge now, i wonder if they'll ever
be able to recover {Flag} from {flag} :monkahmm:")
```

So the goal is to somehow obtain a known result without knowing the original input by applying different operations to it.
The `md5` operation can immediately be discarded as it reduces the length of the flag to 16 bytes.

Right off the bat it seems that this is a challenge where you need to somehow lose information through base64 decode, because python's `b64decode` function just blatantly ignores all nonvalid bytes when decoding. This means that decoding for example the bytestring `b'abc\x00defg'` will only attempt to decode `b'abcdefg'` and return the result of that.

With this knowledge, the goal of the challenge becomes clear - manipulate the input such that we lose information by creating bytes that are non-valid base64 bytes.

We have three other interesting operations - `xor`, `encode`, and `hex`.
One thing to notice is that, if we first `encode` and then `hex` the original flag, the only characters that are in the new flag are `[0-9a-f]`, which means we just reduced the number of possible characters to 16 in our new flag. Now, if we repeat it once again, we can further get rid of the letters `a-f`, since all their character's ascii values can be represented with bytes between `0x61` and `0x66`. So, after two encodings and hexes, we are now left with a string consisting of only numbers. Now these numbers can be `0x66`, but for simplicity, lets repeat the encode and hex once again so that now all values are bytes in the range `0x31` to `0x39` (the hex values of `b'0'` to `b'9'`).

Now the strategy ahead is simple - since we know all the values' byte ranges, let's do an xor operation to make it so that most of them are now invalid base64 characters, and then do a base64decode. Then do a base64encode to obtain back the original numbers (this is usually the case, but there are some edge cases when this is not true*). After that we just repeat it until we obtain a string of a single character, which we know.

> * In my testing if the flag contained characters such as ? it would result in characters that were originally not in the input string when base64 decoding, but the challenge flag luckily did not have that so I did not have to think of a way to deal with that.

Solve script:

```
from pwn import *

FLAG_LENGTH = 62
```

```python
io = remote('10.42.4.59', 9999)


def do_round(val, val2 = None):
    log.info(f'Sending {val}')
    io.sendline(val)
    if val == b'xor':
        log.info(io.recvuntil(b'what would you like to xor the message 
with? (hex):'))
        log.info(f'Sending {val2}')
        io.sendline(val2)

    log.info(io.recvuntil(b'What should we do to the flag?'))



log.info(io.recvuntil(b'What should we do to the flag?'))

for i in range(13):
    # Only numbers 0 - 10
    do_round(b'hex')
    do_round(b'encode')

do_round(b'xor', b'18')
do_round(b'b64decode')
do_round(b'b64encode')

# Now only characters 43 and 47 exist
do_round(b'xor', b'53')
do_round(b'b64decode')
do_round(b'b64encode')
do_round(b'decode')

io.sendline(b'done')
log.info(io.recvuntil(b'Lets just confirm that you ended with the same 
ciphertext I did: '))
io.sendline(b'x' * 507762 + b'w')
log.info(io.recvall())
```

The reason why there are multiple round of hexing and encoding is because when we base64 decode we might lose some info, and it's possible that it will be shorter than the flag then, so just in case let's just make it ridicolously big so that doesn't happen by repeating hexing and encodings, since that always doubles the length of the message.

I won't go in to depth as to what the `xor` values specifically result in, but the goal was to just get rid of as much characters as possible by creating bytes that are not valid base64. After the second round of xor, we are only left with one character when we base64 decode, except the last character which will turn into a `b'w'` instead of an `b'x'`. I got the length of the flag by sending the same input but then sending a wrong flag, since the server tells you how long of a flag it got, from that subtract 1 to obtain the string of `b'x'` and just add the `b'w'` to it and then obtain the flag.