Never Gonna Let You Crypto - GPN CTF 2024

This is the introductory crypto challenge from GPN CTF 2024. The challenge revolves around leveraging the associativity and self-inversion of the xor operation to recover a randomly generated key, and use it to decrypt the flag

Challenge Overview

Title: Never Gonna Let You Crypto
Author: s1nn105
Difficulty: Beginner
Points: 56
Solves: 430
Flag: `GPNCTF{One_T1me_p4ds_m4y_n3v3r_b3_r3u53d!!!}`

Reading through the challenge

We are provided with 2 files: `chall.py`, which contains our encryption scheme, and `FLAG.enc`, which is our encoded flag.

chall.py

`chall.py` encrypts a message by generating a random 5 byte `key`, and xoring the message with the `key` in 5 byte chunks.

```
import os
def encrypt(message,key):
    message = message.encode()
    out = []
    for i in range(len(message)):
        out+= [message[i]^key[i%len(key)]]
    return bytes(out).hex()
FLAG = "GPNCTF{fake_flag}"
key = os.urandom(5)

print(encrypt(FLAG,key))
```

The FLAG variable contains a fake_flag, but we're assuming that the original source just has the real flag there in plaintext. The xor'd message is a list of integers, which are first converted to bytes and then to hex, which is then printed.

Observe that the key is randomly generated, so we can't hope that running our program will get their key. We would bruteforce the key, but there is going to be a better way.

FLAG.enc

This file just contains the encrypted flag, which we know from the above source is a hex string: `d24fe00395d364e12ea4ca4b9f2da4ca6f9a24b2ca729a399efb2cd873b3ca7d9d1fb3a66a9b73a5b43e8f3d`.

Leveraging properties of xor

The xor takes in two input bits, and outputs a single bit that is 0 if the inputs are equal, and 1 if the inputs are not equal. For example, `1 xor 0 = 1`, and `1 xor 1 = 0`. xor is typically denoted with the oplus symbol $\oplus$, and in python is the caret operator `^`. $\oplus$ is an operation on bits, but we can define the xor of two bit strings of the same length as just xoring their corresponding bits. To xor two ASCII strings, just xor them as bit strings.

$\oplus$ is useful for encryption since it's a self inverting operation: if I encrypt x with key y, I can decrypt x by xoring again by y. To show this, first observe that xoring any value with itself will always give a string of all 0s, since $0 \oplus 0 = 1 \oplus 1 = 0$. Second, xoring any string with a string of all 0s gives the string back, since $1 \oplus 0 = 1$ and $0 \oplus 0 = 0$. Third, observe that xor is associative, meaning for any bits x, y, z we have

$$(x \oplus y) \oplus z = x \oplus (y \oplus z).$$

Now, assume that I am given encrypted text $z = x \oplus y$, and know the key y. I can recover x by as

$$z \oplus y = (x \oplus y) \oplus y = x \oplus (y \oplus y) = x \oplus 0 = x.$$

From this, we deduce that $x \oplus y = z \Leftrightarrow x = y \oplus z \Leftrightarrow y = x \oplus z$.

What this means is that, if we knew the randomly generated 5 byte key, we could xor the key with the encrypted text to recover the flag. The only issue is that we *don't* know the key. However, we do know some information about the flag which we can use to get the key: the flag format `GPNCTF{}`.

Leveraging the flag format to recover the key

The encrypted text is produced by repeating the key in 5 byte blocks, that is character i of the output is character i of the flag xor'd with character $i\%5$ of the flag. The important part to know is that the first five characters of output is `output[:5] = FLAG[:5] xor key`. Since we just showed that xor is self inverting, it follows that `key = output[:5] xor FLAG[:5]`, but we know the two values on the right, output from FLAG.enc, and FLAG[:5] since we know all flags start with `GPNCT`. Therefore, we have an exploit.

Crafting our exploit

Recall that our FLAG is encrypted as since we know all flags start with `GPNCT`. Therefore, we have an exploit.

Crafting our exploit

Recall that our FLAG is encrypted as

1. Convert the ASCII characters to bytes
2. xor the flag with the key in blocks of 5 characters,
3. convert the corresponding ints to bytes, and
4. convert the list of bytes to hex.

Therefore, to decrypt, we

1. Convert from hex to a list of bytes
2. xor the first five bytes of output with `GPNCT` to recover the key
3. xor the output with the key in blocks of 5 characters,
4. convert the corresponding ints to ASCII

Implementing this, we get the following solve script:

```
from pwn import *

output =
'd24fe00395d364e12ea4ca4b9f2da4ca6f9a24b2ca729a399efb2cd873b3ca7d9d1fb3a66a9b73a5b43e8f3d'

output_bytes = bytes.fromhex(output)
print(output_bytes)

flag_fragment = b'GPNCT'
output_fragment = output_bytes[:5]
```

```
key = []
for i in range(len(flag_fragment)):
    key += [flag_fragment[i] ^ output_fragment[i]]
print(key)

flag = []
for i in range(len(output_bytes)):
    flag += [output_bytes[i]^key[i%len(key)]]
print(''.join(map(chr, flag)))
```

which gives a key of [149, 31, 174, 64, 193] and a flag of `GPNCTF{One_T1me_p4ds_m4y_n3v3r_b3_r3u53d!!!}`