TurboCipher

- **Category** Crypto
- **Release date** 2022 Oct 29

How that works

The source code of the server:

`crypto_turbocipher/server.py`

```python
from Crypto.Util.number import bytes_to_long, getPrime, getRandomRange
import socketserver
import signal
from typing import Callable
from secret import FLAG, fast_turbonacci, fast_turbocrypt


class Handler(socketserver.BaseRequestHandler):

    def handle(self):
        signal.alarm(0)
        main(self.request)


class ReusableTCPServer(socketserver.ForkingMixIn, socketserver.TCPServer):
    pass


def sendMessage(s, msg):
    s.send(msg.encode())


def receiveMessage(s, msg):
    sendMessage(s, msg)
    return s.recv(4096).decode().strip()


def turbonacci(n: int, p: int, b: int, c: int) -> int:
    if n < 2:
        return n

    return (b * turbonacci(n - 1, p, b, c) +
            c * turbonacci(n - 2, p, b, c)) % p


def lcg(x: int, m: int, n: int, p: int) -> int:
    return (m * x + n) % p


def turbocrypt(pt: int, k: int, f: Callable[[int], int]) -> int:
    return sum((f(i + 1) - f(i)) for i in range(k, pt))


def menu(s) -> int:
    sendMessage(s, "Choose one option\n")
    sendMessage(s, "1. Encrypt flag\n")
    sendMessage(s, "2. Encrypt ciphertext\n")
    sendMessage(s, "3. Exit\n")

    return int(receiveMessage(s, "> "))


def main(s):
    while True:
        b, c, p = getPrime(512), getPrime(512), getPrime(512)

        try:
            for i in range(10):
                assert turbonacci(i, p, b, c) == fast_turbonacci(i, p, b, c)
                assert turbocrypt(i, -1, int) == fast_turbocrypt(i, -1, int)

            break
        except Exception as e:
            sendMessage(s, e)

    sendMessage(s, "Welcome to TurboCipher. Please login first\n")
    sendMessage(s, f"MathFA enabled. Parameters:\n{p = }\n{b = }\n{c = }\n\n")
```

```
    nonce = getRandomRange(1, p)
    sendMessage(s, f"Please, use {nonce = } to generate for your TurbOTP\n")

    otp = int(receiveMessage(s, "OTP: "))

    if otp != fast_turbonacci(nonce, p, b, c):
        sendMessage(s, "Login incorrect\n")
        exit(1)

    sendMessage(s, "Login successful\n")

    m, n, k = getPrime(512), getPrime(512), getPrime(512)

    def f(x: int) -> int:
        return lcg(x, m, n, p)

    while (option := menu(s)) != 3:
        if option == 1:
            assert len(FLAG) * 8 <= p.bit_length()
            sendMessage(
                s, f"ct = {fast_turbocrypt(bytes_to_long(FLAG), k, f) % p}\n")

        if option == 2:
            pt = receiveMessage(s, "pt = ").strip().encode()
            assert len(pt) * 8 <= p.bit_length()
            sendMessage(
                s, f"ct = {fast_turbocrypt(bytes_to_long(pt), k, f) % p}\n")

    sendMessage(s, "Thank you for using TurboCipher. See you soon!\n")


if __name__ == "__main__":
    socketserver.TCPServer.allow_reuse_address = True
    server = ReusableTCPServer(("0.0.0.0", 1337), Handler)
    server.serve_forever()
```

So there are two things happening here. Two functions – **turbonacci** and **turbocrypt** are given, and are at the center of the challenge.

First, the server verifies the `OTP`, which is the `turbonacci` of a given `nonce`.

Turbonacci is a *recursive fibonacci* function. The thing is, with the given nonce (which is made with **GetPrime(512)**) there is no way in hell that the turbonacci sequence will figure it out without breaking the computer.

On top of that, the turbonacci function multiples the values with `b` and `c` mod `P`.

Given that our number is going to be huge, we need to look at faster implementations for finding the nth fibonacci. Since we want the Nth number, and not the whole sequence there are a few options. The *golden ratio* option is probably best, but since there's weird things happening with the `b` and `c` multiplication I gave up.

The other option is Matrix Exponentiation.

Basically:

```
[[1,1],[1,0]]^n * [0,1] = [F(n-1),F(n)]
```

To account for the B and C values, we just need to multiply the starting matrices with the values, which becomes:

```
[[b*1,1],[c*1,0]]^n
```

but our `n` is still huge, which is where the `mod p` comes in.

Once we pass the OTP test, we need to `turbodecrypt` the flag. The flag is a LGC of random numbers. I have no background in math, but it was actually pretty straightforward to recover the parameters and the flag!

How to solve

The first part calculates the OTP through the method. The task file does not provide the code of the method, but `turbonacci` the recursion-based implementation is provided through the method.

The second part uses `fast_turbocrypt` a method to encrypt the plaintext. Similarly, the task file does not provide the code for this method, but `turbocrypt` the method provides a recursive implementation.

After connecting to the operating environment, the system first gives the values of `p` , `b` , `c` and and `nonce` requires the input of `OTP` . Its calculation is carried out by linear recursive algorithm, and its relationship is as follows, the value of OTP is the `f(nonce)` return value of:

```
f(0) = 0
f(1) = 1
f(n) = b * f(n-1) + c * f(n-2) % {p}
```

The matrix transformation form corresponding to the above linear recurrence relation is:

$$\begin{bmatrix} 0 & 1 \\ c & b \end{bmatrix} \mathbf{x} \begin{bmatrix} f(i) \\ f(i+1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \end{bmatrix}$$

From this, we can power the above transformation matrix `nonce` , and then multiply it by the initial `[f(0) , f(1)]` vector to get it `[f(nonce) , f(nonce + 1)]` . It should be noted that all calculations must be performed under the modulus `p` .

```
nc <IP of the challenge> <PORT>
Welcome to TurboCipher. Please login first
MathFA enabled. Parameters:
p = <p>
b = <b>
c = <c>

Please, use nonce =
1211xxxxxxxxxxxxxxxxxxxxxxxxxxxx14914398694949126532958974218454931067848504186010477435233142473104314546967571465078355301862135110796 to generate for your TurbOTP
OTP:
```

`step1.sage`

```
from Crypto.Util.number import bytes_to_long, long_to_bytes

p = <REPLACE WITH THE VALUE OF p FROM ONLINE INSTANCE>
b = <REPLACE WITH THE VALUE OF b FROM ONLINE INSTANCE>
c = <REPLACE WITH THE VALUE OF c FROM ONLINE INSTANCE>

nonce = <REPLACE WITH THE VALUE OF nonce FROM ONLINE INSTANCE>

m = matrix(GF(p), [[0,1], [c,b]])

x0 = 0
x1 = 1

m = m ^ nonce

result = m * vector([x0, x1])

otp = result[0]; otp
print("OTP =", otp)
```

Execution:

```
sage step1.sage
OTP = <otp value>
```

then we can continue to the 2nd part of the challenge.

```
## PART 1 ##

Welcome to TurboCipher. Please login first
MathFA enabled. Parameters:
p = <p value>
b = <b value>
c = <c value>

Please, use nonce = <nonce value> to generate for your OTP

## Put the OTP ##

OTP: <otp value>
Login successful

## PART 2 ##

Choose one option
1. Encrypt flag
2. Encrypt ciphertext
3. Exit
```

The next step is to decrypt the FLAG ciphertext, which starts by generating three random prime numbers, `m`, `n` and `k`. Given the plaintext `pt`, the ciphertext `ct` is calculated by the following equation:

$$\begin{aligned} ct &= \sum_{i=k}^{pt-1} lcg(i+1) - lcg(i) \\ &= lcg(pt) - lcg(k) \end{aligned}$$

`lcg` is defined as:

$$lcg(x) \equiv m * x + no \pmod{p}$$

In the above equation the ciphertext `ct` and `p` of the FLAG are known and the plaintext `pt`, `m`, `n` and `k` are unknown.

This allows us to encrypt the two known plaintexts through the encryption interface provided by the system and then use the corresponding ciphertexts to solve the linear equations to obtain the FLAG plaintext.

```
nc <IP of the challenge> <PORT>
...<skip PART 1 already shown previously>...
Choose one option
1. Encrypt flag
2. Encrypt ciphertext
3. Exit
> 1
ct = <Receive the value for original ct>
Choose one option
1. Encrypt flag
2. Encrypt ciphertext
3. Exit
> 2
pt = Z_JUST_NEED_TO_PUT_A_SENTENCE_FOR_THE_TEST_CASE
ct = <Receive the value for ct1>
Choose one option
1. Encrypt flag
2. Encrypt ciphertext
3. Exit
> 2
pt = Y_JUST_NEED_TO_PUT_A_SENTENCE_FOR_THE_TEST_CASE
ct = <Receive the value for ct2>
Choose one option
1. Encrypt flag
2. Encrypt ciphertext
3. Exit
> 3
Thank you for using TurboCipher. See you soon!
```

We get 2 values of encrypted `ct` for 2 known plaintexts of `pt` AND the value `ct` of the encrypted flag.

`step2.sage`

```
from Crypto.Util.number import bytes_to_long, long_to_bytes

p = <p value>
b = <b value>
c = <c value>

nonce = <nonce value>

m = matrix(GF(p), [[0,1], [c,b]])

x0 = 0
x1 = 1

m = m ^ nonce

result = m * vector([x0, x1])

otp = result[0]; otp

ct = <value of original ct>

pt1 = b"Z_JUST_NEED_TO_PUT_A_SENTENCE_FOR_THE_TEST_CASE"
ct1 = <value of ct received when put Z_JUST_NEED_TO_PUT_A_SENTENCE_FOR_THE_TEST_CASE>

pt2 = b"Y_JUST_NEED_TO_PUT_A_SENTENCE_FOR_THE_TEST_CASE"
ct2 = <value of ct received when put Y_JUST_NEED_TO_PUT_A_SENTENCE_FOR_THE_TEST_CASE>

delta1  = (ct - ct1) % p
delta2  = (ct - ct2) % p

p1 = bytes_to_long(pt1)
p2 = bytes_to_long(pt2)

t = (p2 - p1) % p
t = pow(t, -1, p)

t = t * (delta1 - delta2)
m = t % p

flag = delta1 * pow(m, -1, p) + p1
flag = int(flag % p)

print("FLAG =", long_to_bytes(flag))
```

Execution:

```
sage step2.sage
FLAG = b'HTB{C*****5_m***5_Cryp**_***_*********???}'
```

We got the flag of this crypto challenge.