Angstrom CTF 2021

Dysfunctional

> ***Challenge's Description:*** My "functional programming in C" library finally works! Although, now that I realize it, it seems more... dysfunctional. Either way, I've used it for my cutting edge flag encryption algorithm along with this file - here's the encrypted flag.

Nếu không thể truy cập vào file source của server, bạn có thể tim thấy nó ở đây:
dysfx, not_flag, encrypted_flag

TL; DR

1. Chương trình sẽ đọc byte trong 2 file **flag** và **not_flag**
2. Từng byte trong file **flag** sẽ được nối với 3 bytes sinh ra bởi ***/dev/urandom,*** như vậy, từ mảng 1-byte flag ban đầu, giờ chúng ta đã có một mảng mới với kích cỡ các phần tử là 4-byte, gọi mảng này là mảng **originalArr**
3. Mỗi phần tử của **originalArr** sẽ gồm 2 word, mỗi word(2 bytes) sẽ được xor với các tham số có sẵn(chẳng hạn 0xdead, 0x1337...), tính toán và sinh ra một index
4. Ánh xạ index này đến mảng byte đọc từ file **not_flag.** Lấy một word tại index đó.Suy ra: từ phần tử 2 word ban đầu –sinh ra–> 2 word mới, và 2 word mới này tạo thành một số trong file **encrypted_flag:**

```
5400f172 949c5165 c6bc4607 c621d63f c20c0970 f2b3f53c aabb67f9 fe0e7abb 6a

```

6. **Solve**: Từ file **encrypted_flag** suy ra được flag sẽ có 40 ký tự –> tách mỗi số thành 2 word, tìm trong file **not_flag** –> index. Vì index tạo thành từ các phép tính mang tính **reversible,** vì vậy mà từ mỗi dword được khôi phục lại chúng ta sẽ tách lấy ***byte cuối cùng***, chính là một byte của flag

---

Ngụ ý tên của challenge và viết solve script

Tên Dysfunctional

- Sẽ có 2 hàm rất dễ thấy là **closure()** và **compose()**. Tuy nhiên thực chất, 2 hàm này sẽ được build lại lúc chạy chương trình(**sub_FFD()** sẽ chịu trách nhiệm cho việc tính lại các giá trị cần thiết cho việc build một hàm như ***func_start, func_size*** v.v.)
  - chức năng thật của **closure()** function: sẽ chịu trách nhiệm cho việc tính toán các phần tử của **originalArr**, tham chiếu đến mảng byte từ file **not_flag**
    - Chú ý, sẽ có 2 bộ tham số khác nhau cho hàm **closure():** *(0xdead, 0xbeef, 0xfeed)* và *(0x1337, 0xcafe, 0x1234)*
  - 2 lần gọi hàm compose thực ra là:
    - ***compose1*** : Gọi hàm **closure()** ***theo thứ tự*** bộ tham số (0xdead, 0xbeef, 0xfeed), tiếp theo là với tham số (0x1337, 0xcafe, 0x1234)
    - ***compose2***: (0x1337, 0xcafe, 0x1234), tiếp theo là (0xdead, 0xbeef, 0xfeed)
  - map gọi **compose1** và truyền vào mảng originalArr để xử lý, sinh ra một mảng mới **resArr1**
  - **resArr1** truyền vào map với hàm compose2, xử lý và sinh ra các giá trị trong file **encrypted_flag**

```
v5 = closure(0xDEADLL, 0xBEEFLL, 0xFEEDLL, lookup);
v6 = closure(0x1337LL, 0xCAFELL, 0x1234LL, lookup);
compose1 = compose(v5, v6);
compose2 = compose(v6, v5);
resArr1 = map(compose1, originalArr, v15);
encrypted = map(compose2, resArr1, v15);
for ( k = 0; k < (signed int)v15; ++k )
  printf("%x ", *(unsigned int *)(4LL * k + encrypted));
```

Solve Script

Bạn có thể tìm thấy một solve script của challenge này quanh đây.