Comprehensive 2 writeup

Assumed knowledge:

- basic Python syntax

Building a picture of what the program does

When we first open the program we are greeted by this truly vile list comprehension:

```
m = '[?????]'
n = '[?????]'

a = 'abcdefghijklmnopqrstuvwxyz'
p = ' !"#$%&\'()*+,-./:;⟺?@[\\]^_`{|}~'

assert len(m) == 63 and set(m).issubset(set(a + p))
assert len(n) == 7  and set(n).issubset(set(a))
assert m.count('tjctf{') == 1 and m.count('}') == 1 and m.count(' ') == 5

print(str([x for z in [[[ord(m[i]) ^ ord(n[j // 3]) ^ ord(n[i - j - k]) ^
ord(n[k // 21]) for i in range(j + k, j + k + 3)] for j in range (0, 21, 3)]
for k in range(0, len(m), 21)] for y in z for x in y])[1:-1])
```

We can make a start at trying to understand what the program does by looking at the program's inputs and outputs. The assert statements give us some vital information about the inputs to the program. Variable m is 63 characters long and its entirely composed of lowercase alphanumeric characters and punctuation characters; it also contains the strings `tjctf{` and `}` once and it contains five spaces.

Looking at the output we can see it is just an array of numbers, which do not seem to be in the ASCII range:

```
[1, 18, 21, 18, 73, 20, 65, 8, 8, 4, 24, 24, 9, 18, 29, 21, 3, 21, 14, 6, 18,
83, 2, 26, 86, 83, 5, 20, 27, 28, 85, 67, 5, 17, 2, 7, 12, 11, 17, 0, 2, 20,
12, 26, 26, 30, 15, 44, 15, 31, 0, 12, 46, 8, 28, 23, 0, 11, 3, 25, 14, 0, 65]
```

We now know that the program takes in a 7 character key and a 63 character message, and produces an array of numbers.

Providing our own inputs

Now that we understand the inputs and outputs of the program it is useful to provide our own test data for the program to operate on. This is so that we know any change we make to the program itself hasn't changed the functionality of the program. I chose the following as my inputs:

```
m = "paddingandstuffaaaaaaa_tjctf{beans really do make you fartalot}"
n = "abcdefg"
```

These inputs produce the following output:

```
17, 3, 7, 6, 8, 14, 4, 1, 15, 0, 20, 18, 16, 0, 1, 7, 4, 5, 6, 5, 4, 3,
62, 20, 11, 1, 23, 6, 24, 0, 2, 5, 11, 21, 69, 22, 0, 7, 11, 8, 30, 70,
7, 15, 65, 13, 2, 9, 4, 66, 26, 9, 16, 68, 1, 5, 23, 16, 6, 10, 10, 18, 26
```

Turns out the output from the challenge description includes the brackets from the array syntax but the actual program's output doesn't… However this doesn't matter as long as it is consistent. Personally I find it easiest to just run the program as follows:

```
./comprehensive_2.py | sha256sum
3fe6aac14d8c703a2b2517eb36c4692b3c4ad63cd4f11a37f19a513f84c2af25  -
```

This means you can just check the hash each time you run the program, much quicker than checking each number.

Structural Changes

Now that we have an idea of what the program does we can move on to changing it to make it easier to understand.

Personally I use `black` (`https://github.com/psf/black`) to reformat all of my python code in a consistent way. This makes it `PEP8` compliant and considerably more readable.

```
cat comprehensive_2-original.py | black - > comprehensive_2.py
```

This is now the contents of `comprehensive_2.py`:

```
 #!/usr/bin/env python
m = "paddingandstuffaaaaaaa_tjctf{beans really do make you fartalot}"
n = "abcdefg"


a = "abcdefghijklmnopqrstuvwxyz"
p = " !\"#$%&'()*+,-./:;⟺?@[\\]^_`{|}~"

assert len(m) == 63 and set(m).issubset(set(a + p))
assert len(n) == 7 and set(n).issubset(set(a))
assert m.count("tjctf{") == 1 and m.count("}") == 1 and m.count(" ") == 5

print(
    str(
        [
            x
            for z in [
```

```
                [
                    [
                        ord(m[i]) ^ ord(n[j // 3]) ^ ord(n[i - j - k]) ^ ord(n[k // 21])
                        for i in range(j + k, j + k + 3)
                    ]
                    for j in range(0, 21, 3)
                ]
                for k in range(0, len(m), 21)
            ]
            for y in z
            for x in y
        ]
    )[1:-1]
)
```

Simplifying the program

I first renamed `n` to `key` and `m` to `message`, and re-ran `black`. This makes it a bit easier to understand what is going on:

```
#!/usr/bin/env python
message = "paddingandstuffaaaaaaa_tjctf{beans really do make you fartalot}"
key = "abcdefg"


a = "abcdefghijklmessagenopqrstuvwxyz"
p = " !\"#$%&'()*+,-./:;⟸⟹?@[\\]^_`{|}~"

assert len(message) == 63 and set(message).issubset(set(a + p))
assert len(key) == 7 and set(key).issubset(set(a))
assert (
    message.count("tjctf{") == 1 and message.count("}") == 1 and message.count(" ") == 5
)

print(
    str(
        [
            x
            for z in [
                [
                    [
                        ord(message[i])
                        ^ ord(key[j // 3])
                        ^ ord(key[i - j - k])
                        ^ ord(key[k // 21])
                        for i in range(j + k, j + k + 3)
```

```
                    ]
                    for j in range(0, 21, 3)
                ]
                for k in range(0, len(message), 21)
            ]
            for y in z
            for x in y
        ]
    )[1:-1]
)
```

Now that the program is easier to read I want to try and take it out of a list comprehension and into an actual for loop. There is a very strange syntax at use here where there are nested for loops in a single list comprehension. Although this may look strange it is actually just a technique to go for flatting lists:

```
[
    [i for i in range(k, k + 4)]
    for k in range(4)
]
```

Returns this:

```
[[0, 1, 2, 3], [1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6]]
```

But if we put in the extra for loops:

```
[
    y
    for z in [
        [i for i in range(k, k + 4)]
        for k in range(4)
    ]
    for y in z
]
```

Returns this:

```
[0, 1, 2, 3, 1, 2, 3, 4, 2, 3, 4, 5, 3, 4, 5, 6]
```

By decomposing to nested for loops the flattening code disappears. I'm going to start with the outermost loop which isn't part of the flatting technique:

```
[
    [
        ord(message[i])
        ^ ord(key[j // 3])
        ^ ord(key[i - j - k])
        ^ ord(key[k // 21])
        for i in range(j + k, j + k + 3)
    ]
```

```
    for j in range(0, 21, 3)
]
for k in range(0, len(message), 21)
```

This becomes:

```
out = []
for k in range(0, len(message), 21):
    for j in range(0, 21, 3):
        for i in range(j + k, j + k + 3):
            out.append(
                ord(message[i])
                ^ ord(key[j // 3])
                ^ ord(key[i - j - k])
                ^ ord(key[k // 21])
            )

print(str(out)[1:-1])
```

Now that it is in the form of a loop I want to make each loop as simple as possible. This can be done making all of the loops run from 0 to `n` in increments of one and scaling the variable appropriately:

```
for k, j, i in it.product(range(3), range(7), range(3)):
    out.append(
        ord(message[i + (3 * j) + (21 * k)])
      ^ ord(key[j])
      ^ ord(key[i])
      ^ ord(key[k])
    )
```

This can be simplified further by looking at the value the indexes take as we go through the loop:

```
for k, j, i in it.product(range(3), range(7), range(3)):
    print(
        (i + (3 * j) + (21 * k)),
        (i, j, k)
    )
```

Outputs:

```
0 (0, 0, 0)
1 (1, 0, 0)
2 (2, 0, 0)
3 (0, 1, 0)
4 (1, 1, 0)
5 (2, 1, 0)
6 (0, 2, 0)
7 (1, 2, 0)
```

```
8 (2, 2, 0)
...
54 (0, 4, 2)
55 (1, 4, 2)
56 (2, 4, 2)
57 (0, 5, 2)
58 (1, 5, 2)
59 (2, 5, 2)
60 (0, 6, 2)
61 (1, 6, 2)
62 (2, 6, 2)
```

From this it is much easier to see the looping pattern of each variable. We can use modular arithmetic to get each of `i`, `j` and `k` from the overall index in the string. The below code implements this:

```
def get_key_indexes(n):
    i = n % 3
    j = n // 3 % 7
    k = n // 21
    return i, j, k

for n, m in enumerate(message):
    i,j,k = get_key_indexes(n)
    out.append(
        ord(m)
      ^ ord(key[i])
      ^ ord(key[j])
      ^ ord(key[k])
    )
```

Cracking the cipher

So we now know that each element of the original message is XORed with either one or three elements of the key. It can only be one or three because either none of them or two of them do and XOR is a self inverse operation, so performing it twice with the same input does nothing.

I tried various methods to try and solve this mathematically, mainly centering around the property of `tjctf{` being a part of the message. However this lead to a contradiction in what a member of the key was so I gave up with maths.

I then focused on the fact there were only seven characters in the key. This and the fact the encryption / decryption process is very cheap at a hardware level. I decided to try and bruteforce the key. This ended up working fantastically well and led to me getting the flag.

I decided to write my bruteforce in `C` because `C` is quick as fuck boiiii. But in all

seriousness it is a low level language which I have used extensively before and therefore it was just the easiest tool for the job.

Implementing the bruteforce

There were relatively few challenges with writing this program, the hardest parts we're probably parsing the input and generating the keys.

Parsing the input Now of course instead of simply copying a copy of the output into the source file and converting it to a C format. I decided to write a function to parse the contents of a file:

```c
unsigned char * parse_input(const char * fin) {
    /* open the input file */
    FILE * fp = fopen(fin, "r");

    /* parse the input */
    unsigned char * input = calloc(63L, sizeof(unsigned char));
    for (int i = 0; i < 63; i++) {
        switch (i) {
            case 0:
                fscanf(fp, "[%hhu, ", &input[i]);
                break;
            case 62:
                fscanf(fp, "%hhu]", &input[i]);
                break;
            default:
                fscanf(fp, "%hhu, ", &input[i]);
                break;
        }
    }

    /* Close the file */
    fclose(fp);

    return input;
}
```

Its pretty simple, it has a case for the start of the input and one for the end then everything else is in the same format.

Generating the keys This proved to be quite difficult as I had some issues with trying to modify a variable declared as `char *`. Forgetting that `C` does not approve of such heathen declarations and you must instead declare it `char []`… Past that the actual algorithm was fairly easy, I wasn't sure whether it would properly enumerate through all the keys when I wrote it. But I ran some tests and it work perfectly!

```c
/* transform the key to the next one */
bool mutate_key(char * key) {
    for (int i = 0; i < 7; i++) {
        if (key[i]++ == 'z') {
            key[i] = 'a';
        } else {
            return true;
        }
    }
    return false;
}
```

The idea is that you just increment the value of each character in turn, returning `true` each time. Continue incrementing this same character until you reach `'z'`, then you set the current character to `'a'` and you go to increment the next character along. Performing the same increment logic, until you reach a `'z'` on the last character, then you return `false`.

Decrypting the ciphertext This was the easiest bit as we have already worked out how to get `i`, `j` and `k` from the string index. Then it was just a matter of performing the XORs:

```c
/*
 * Perform a decryption with the given key
 */
void decrypt(FILE * fp, const unsigned char * input, const char * key) {
    /* Copy in the input as we are going to modify it */
    unsigned char enc[64];
    memcpy(enc, input, 63);

    for (int i = 0; i < 63; i++) {
        /* perform the decryption step */
        enc[i] ^= key[i % 3] ^ key[i / 3 % 7] ^ key[i / 21];
    }

    if (check_predicate((char *) enc)) {
        fprintf(fp, "%s -> %s\n", key, enc);
        fprintf(stderr, "%s -> %s\n", key, enc);
    }
}
```

Checking for a valid decryption Finally we need to check whether a given decryption is valid. I originally planned on doing this with grep / another tool after doing all the decryptions. Generating ~80Gb of output in a couple of minutes quickly dismantled that plan. So I had used the assert statements at the top of the python file and, chose those which were the most simple to

implement in `C`:

```c
/* check the assertions on the decrypted output */
bool check_predicate(const char * dec) {
    /* count("}") == 1 */
    char * first_bracket = strchr(dec, '}');
    if (first_bracket == NULL || strchr(first_bracket + 1, '}') != NULL) {
        return false;
    }

    /* count("tjctf{") == 1 */
    char * tjctf = strstr(dec, "tjctf{");
    if (tjctf == NULL || strstr(tjctf + 1, "tjctf{") != NULL) {
        return false;
    }

    /* count(" ") == 5 */
    char * space = strchr(dec, ' ');
    for (int i = 0; i < 5; i++) {
        if (space == NULL) {
            return false;
        }
        space = strchr(space + 1, ' ');
    }
    if (space != NULL) {
        return false;
    }

    return true;
}
```

Getting the flag

Now that we've written the code we just need to put it all together and compile it:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

unsigned char * parse_input(const char *);
void decrypt(FILE *, const unsigned char *, const char *);
bool check_predicate(const char *);
bool mutate_key(char *);

int main (void) {
```

```
    /* get the input */
    unsigned char * input = parse_input("../out.txt");

    /* open the output file */
    FILE * fp = fopen("output", "w");

    /* perform bruteforce */
    char key[] = "aaaaaaa";
    do {
        decrypt(fp, input, key);
    } while (mutate_key(key));

    /* close resources */
    fclose(fp);
    free(input);
    exit(EXIT_SUCCESS);
}
```

I used `gcc brute.c -Wall -Wextra -std=c99 -pedantic -Ofast -o brute` to compile the program, but basically anything will work most of the flags are either error checking / optimisations anyway.

Now all that is left is to wait while it cracks the message. It checked around 4.5 million keys a second on my machine resulting in a total run time of ~30 minutes.

Below is the program's output:

```
isacapo -> hata o sagashiteimasu ka? dozo, tjctf{sumimasen_flag_kudasaii}.
isacaps -> hata o sagashiteim}oi ka? dozo, tjctf{siqumasen_flag_kudasaiua2
```

Which clearly reveals the flag: `tjctf{sumimasen_flag_kudasaii}`

My thanks to everyone to made tjctf, it was definitely one of my favourites from the year so far!

– The full bruteforce code and deobfuscated python code are available along with the this writeup in our Github repo.