Bruteforcing a simple crypto challenge

This write-up describes how I solved **Catch the criminal 1**, which was a crypto challenge on GenZ Hack season 1.

The challenge contained two files: `encode.c`, which was an encoded written in C, and `encrypted_data` which contained the flag in binary form created by the encoder.

Analysing the code

Most of `encode.c` is just boilerplate around reading and writing to the file. The interesting bit that actually does the encoding is the following code snippet:

```
rbcnt = 2;
for (fptr = 0; fptr < fs; fptr = fptr + 2)
{
  if (rbcnt > (fs - fptr))
  {
    rbcnt = fs - fptr;
  }
  fread(dbuf, sizeof(uint8_t), rbcnt, fin);
  if ((rbcnt % 2) == 0)
  {
    tmp = (dbuf[0] ^ 0xa5) & mask;
    dbuf[0] = ~(dbuf[1] & mask);
    dbuf[1] = ~tmp;
    fwrite(dbuf, sizeof(uint8_t), rbcnt, fout);
  }
  else
  {
    dbuf[0] = ~(dbuf[0] & mask);
    fwrite(dbuf, sizeof(uint8_t), rbcnt, fout);
  }
}
```

The code is hard to understand for someone not used to working with C (like me). Based on their names `fread` probably read files and `fwrite` writes them, but I needed to research further how they work. We can use `man` pages for that.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

The function fread() reads nmemb items of data, each size
bytes long, from the stream pointed to by stream, storing
them at the location given by ptr.
```

Based on that, I now know that the relevant `fread` call `fread(dbuf,`

`sizeof(uint8_t), rbcnt, fin);` reads `rbcnt` bytes from `fin` to `dbuf`. After reading more help pages I annotated the source code with helpful comments to understand what is going on:

```
// `rbcnt` controls how many bytes will be read per each iteration; by default 2
rbcnt = 2;
// go thru the full file contents, increment `fptr` by 2 each iteration
for (fptr = 0; fptr < fs; fptr = fptr + 2)
{
  // max out `rbcnt` (amount of bytes to read) at the
  // remaining contents length => do not overflow
  // practically speaking `rbcnt` will be 2 or 1
  if (rbcnt > (fs - fptr))
  {
    rbcnt = fs - fptr;
  }

  // read `rbcnt` bytes from the file to variable `dbuf`,
  // i.e. 2 bytes or the file length, whichever is shorter
  fread(dbuf, sizeof(uint8_t), rbcnt, fin);

  // practically checks if we just read 2 bytes
  if ((rbcnt % 2) == 0)
  {
    // perform some magic tricks to both bytes in `dbuf`
    tmp = (dbuf[0] ^ 0xa5) & mask;
    dbuf[0] = ~(dbuf[1] & mask);
    dbuf[1] = ~tmp;
    // write both bytes in dbuf to the output file
    fwrite(dbuf, sizeof(uint8_t), rbcnt, fout);
  }
  // nope, we only read one byte
  else
  {
    // perform a magic trick to the one byte remaining
    dbuf[0] = ~(dbuf[0] & mask);
    // write the one byte in dbuf to the output file
    fwrite(dbuf, sizeof(uint8_t), rbcnt, fout);
  }
}
```

To simplify things even further, here is essentially the same thing in pseudocode:

```
repeat until file ends {
  read two bytes from input file
  perform magic tricks to said two bytes
  write two bytes with magic tricks applied to output file
```

```
}
```

At this point "magic tricks" was a mystery to me. By copy-pasting the definition of `mask` into a Python interpreter I knew that it was 255 (0xFF), so ANDing a byte with it would just result in the original byte. XOR is reversible, so I knew that the encoding was reversible, I just did not know how. I was trying to reorder things and hoping I would just make a decoder by accident, before I realized something important:

Each two bytes are totally independent from each other. There are about 65 thousand possible combinations of two bytes of information. 65 thousand is a fairly small number.

Bruteforce time

I wrote a small and hacky Python script that

1. Writes all possible pairs of two bytes in `tmpdecode.bin` and stores the pairs and their file positions.
2. Runs a compiled version (`gcc -o encode encode.c`) of the provided encode program, encoding the contents of `tmpdecode.bin` to `tmpdecode.enc`.
3. Reads the encoded two-byte pairs `tmpdecode.enc`, and creates a lookup table of original and encoded byte pairs at the same index.
4. Reads an encrypted file specified as an argument and looks up the original bytes using the lookup table created earlier, and writes the decrypted result to a file.

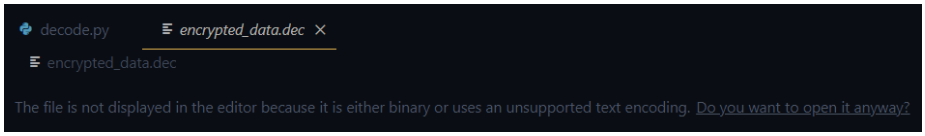
```
import subprocess, struct, sys

key = {}
indexes = {}
bytes_count = 0

with open('tmpdecode.bin', 'wb') as f:
  for first in range(256):
    for second in range(256):
      combo = bytes([first, second])
      f.write(combo)
      indexes[bytes_count] = combo
      bytes_count += 2

subprocess.check_output(['./encode', 'tmpdecode.bin', 'tmpdecode.enc'])

with open('tmpdecode.enc', 'rb') as f:
  for byte in range(0, bytes_count, 2):
    buf = struct.unpack('BB', f.read(2))
    key[buf] = indexes[byte]
```

```
with open(sys.argv[1], 'rb') as fin, open(sys.argv[1] + '.dec', 'wb') as fout:
  while True:
    try:
      buf = struct.unpack('BB', fin.read(2))
    except:
      print("EOF!")
      break
    fout.write(key[buf])
```

So let's run it.

```
$ python3 decode.py encrypted_data
EOF!
```

No errors, great! Now let's just read the flag.



I thought it was a human readable flag? How did that not work? Let's use `file` to figure it out what is going on:

```
$ file encrypted_data.dec
encrypted_data.dec: PNG image data, 663 x 284, 8-bit/color RGBA, non-interlaced
```

Ah.

```
$ mv encrypted_data.dec encrypted_data.png
```

And now we can open the file and read the flag. Not being able to copy-paste the flag sucks, but I managed to submit it anyways and got my well-deserved points.

Summary

I'm mostly a developer/sysadmin working with higher level languages, not a security researcher or a C developer. I still had fun and managed to solve the challenge by finding a flaw in the encoding/encryption. Not sure if that was the intended solution, but it works so it counts, right? I also learned new things which can help me make my own programs more secure in the future.