Crypto

(100) A lost message

We are given two files: `key.png` and `readme.pdf`. From readme we can learn the following information:

- Images from key.png file correspond to numbers according to the „La Smorfia". The actual mappings are described in this wikipedia entry: https://it.wikipedia.org/wiki/La_Smorfia
- The time spend on the island (15 weeks and 3 days = 108 days) is important
- Something about the emperor and latin manuscripts, which hints us at the Caesar cipher.
- The additional hex string, which corresponds to the „encrypted" message: 727f00340e075a6b3a69146f2d3e3a67403c343e101d052b1a58623d3c1a0e53087c00245b6e00771d1f1005316e08693e24000714

By trial and error, we ended up taking a xor of the hex string, the numbers from key.png and value 108. We got the following string as a result: `C9_ckX9_t6z_YavV6ykJ_z6_rk0bK_Qgz9_Ck_n0Bk_Zu_m6_h0iq`, which looked promising.

Performing ceasar cipher on letters, we get the following string: `W9_weR9_n6t_SupP6seD_t6_le0vE_Kat9_We_h0Ve_To_g6_b0ck`, which looked even more promising.

Performing ceasar cipher on digits, we get: `W3_weR3_n0t_SupP0seD_t0_le4vE_Kat3_We_h4Ve_To_g0_b4ck`, which turned out to be a flag.

(200) Darth stuff

We are given two web servers, which turn out to work in exactly the same manner. Upon connecting to one of them, we are greeted with two hexencoded numbers:

- p, which as name suggests, turns out to be prime
- another large number, smaller than p

We are asked by the server to provide our own value. Almost any hexencoded number will get accepted and will result in server giving us another message in the following format:

- The string „CFB”
- 16 bytes encoded in base64
- 23/25 (depending on used server) bytes encoded in base64

The CFB string hints, that we have to perform AES decryption with mode CFB and therefore the first base64 string could be either key or IV and the second base64 string could be an encrypted message. Using the first string as a key and zeroes as an IV didn't provide correct results, therefore we must probably search for the key somewhere else.

This is where the first part of the task comes into play. As we are given a prime moduli and some other value and we are tasked with providing our own value, it heavily resembles the Diffie-Hellman key exchange scheme. The only thing we're missing is the generator for the scheme, but as it's pretty standard for it to be a single-digit number, we can take a guess (and it turned out that used g was equal to 2).

Therefore, we simply must follow the DHKE scheme to obtain the shared secret. It turned out, that the 16 least significant bytes of the shared secret was a key and provided base64 strings were the IV and the encrypted message. Upon decrypting them we are provided with two parts of the flag, first given by the first server and the second given by the second server.

(300) Mind your keys

We are given two sets of files: 20000 RSA public keys in .pem format and corresponding 20000 encrypted messages. The keys appeared to be strong, therefore our first line of attack was to check if some of them shared some common divisors.

First, we've exported all the moduli from all key files using open ssl: `cat keyXXXXX.pem | openssl rsa -noout -pubin -modulus`. Then, we bruteforced through every pair of them, calculating their GCD. After some processing, we found out that the keys `key19440.pem` and `key15088.pem` had a common divisor. Therefore, knowing the factorisation of modulus from `key19440.pem`, we could calculate all the other information required to create the corresponding private key.

First, we created the private key plaintext asn1 file:

```
asn1=SEQUENCE:rsa_key

[rsa_key]
version=INTEGER:0
modulus=INTEGER:8852983930060336687510863188851466956768367628453131368879112384313...
pubExp=INTEGER:65537
privExp=INTEGER:6648822329944455372679321393339170294827945354112381188889175756533...
p=INTEGER:8720812354028632440495785017700416116466719956448783943826917002173771752...
q=INTEGER:1015155878909680562221398235963073566637283630099630019001448387633204489...
e1=INTEGER:863871001149791443058099033140227679449806154649518674387358975511728126...
e2=INTEGER:737222163065555894813086159611006352602878940002773870094068000686859857...
coeff=INTEGER:670632968545787190208496483753504723990376549330229809090198610346704...
```

Then, we converted it to the .der file using openssl: `openssl asn1parse -genconf rsa_plaintext -out private.der`

Then, converted to the .pem file: `openssl rsa -in private.der -inform der -out private.pem`

Base64 decoded the corresponding ciphertext: `cat msgs/msgs19440.enc | base64 -d > enc`

And finally, decrypted it: `openssl rsautl -decrypt -inkey private.pem -in enc -oaep`, getting the flag: `{FLG:sh4r1ng_s3cr3ts_w34k3ns_th3m}`