John the Rocker

We are provided with a rsa private key. Judging from the challenge name, we are supposed to use the John the Ripper utility to crack it's password. We ran it using the rockyou.txt wordlist (which is one of the most extensive common password list available).

John the ripper utility: https://www.openwall.com/john/. rockyou.txt: https://github.com/brannondorsey/naive-hashcat/releases/download/data/rockyou.txt.

After solving our dependencies errors:

```
$ sudo apt install ocl-icd-opencl-dev
```

During the event we followed this guide: https://null-byte.wonderhowto.com/how-to/crack-ssh-private-key-passwords-with-john-ripper-0302810/.

```
$ python ssh2john.py idrsa.id_rsa > rsa.hash # Convert the ssh key to john format
$ john --wordlist=rockyou.txt rsa.hash # Crack the key using our wordlist
$ john rsa.hash --show # Show the password we cracked
idrsa.id_rsa:!!**john**!!

1 password hash cracked, 0 left
```

The password john found is `!!**john**!!`

Our flag is **VishwaCTF{!!**john**!!}**

Tallest Header (Solved after the event)

We are given a file that has seemingly had their header tempered with: file.extension

I will cut to what lead us to solve this challenge and not bother with the red herrings.

When running strings on the file:

```
$ strings file.extension
...
^PK
oT,
info.txtkey = [2,1,3,5,4]
ciphertext = RT1KC _YH43 3DRW_ T1HP_ R3M7U TA1N0PK
encrypt.pyPK
oT,
info.txtPK
```

We can see some sort of cipher text that we cant seem to decrypt just yet, and the strings "encrypt.pyPK" and "info.txtPK". Being the master of file format recognition, I compared this to the strings output of a zip file and the name of the files were present with a PK after them.

A quick lookup to find the header of zip files: https://en.wikipedia.org/wiki/ZIP_(file_format).

And we can just paste the correct values in a hex editor:

The first four bytes were changed to:

03 F2 0B E1 -> 50 4B 03 04

Now the file is recognized as a zip file.

unzipping it gives us two files:

encrypt.py

```
def encrypt(key, plaintext):
    plaintext = "".join(plaintext.split(" ")).upper()
    ciphertext = ""
    for pad in range(0, len(plaintext) % len(key) * -1 % len(key)):
        plaintext += "X"
    for offset in range(0, len(plaintext), len(key)):
        for element in [a - 1 for a in key]:
            ciphertext += plaintext[offset + element]
        ciphertext += " "
    return ciphertext[:-1]
```

info.txt

```
key = [2,1,3,5,4]

ciphertext = RT1KC _YH43 3DRW_ T1HP_ R3M7U TA1N0
```

A quick test of the script with the key seems to indicate that we are dealing with a permutation type of algorithm:

```
print(encrypt([2,1,3,5,4], "VISHWACTF HELLO WORLD"))

""" output
IVSWH CATHF LELWO ROLXD
"""
```

Before actually bothering reading the code for the encryption we decided to try and test if running the encryption with the ciphertext and the key will put it back in it's original form:

```
print(encrypt([2,1,3,5,4], "RT1KC _YH43 3DRW_ T1HP_ R3M7U TA1N0"))

""" output
TR1CK Y_H34 D3R_W 1TH_P 3RMU7 AT10N
"""
```

Removing the spaces and putting that in flag format we get:
VishwaCTF{TR1CKY_H34D3R_W1TH_P3RMU7AT10N}

JumbleBumble

We are provided 2 files, a script and it's output. This seems to be an RSA challenge where we need to find a vulnerability in the original script:

```
import random
from Crypto.Util.number import getPrime, bytes_to_long

flags = []

with open('stuff.txt', 'rb') as f:
    for stuff in f.readlines():
        flags.append(stuff)


with open('flag.txt', 'rb') as f:
    flag = f.read()
    flags.append(flag)

random.shuffle(flags)

for rand in flags:
    p = getPrime(1024)
    q = getPrime(1024)
    n = p * q
    e = 4
    m = bytes_to_long(rand)
    c = pow(m, e, n)
    with open('output.txt', 'a') as f:
        f.write(f'{n}\n')
        f.write(f'{e}\n')
        f.write(f'{c}\n\n')
```

The first thing that seems weird is the small exponent **4** (the variable e).

If we take one of the outputs:

```
13429318136571852145145649411772491995332035293730226791251563560301809436776254817224615289443655170569736246470393909437556925322131303835353484836979951773463320270476900331307190257818047623952712925834888009156673785486596961309758468135925110107730363598538946152166857127804719928347657986687722906484485830771687328344892838668063970810268430714107514868174858412310311572137489165417257889449057032018101807656492665066046884558691042751221915774948607359726402125391345567795296366495183103950017877281952404893826385252866830903527304184990783022333007975894148881834118051727610738520081545675298240582993
4
49620546793711579782402639693377280522728526662509820457210884055445991049736319471760879817907638541914701296886947181962112393611224773708306316648797562964914200791382533487387764578519064974083856
```

We can see that the ciphertext "c" (3rd number) is clearly smaller than the modulus "n" (1st number), and that is due to the small exponent "e" (2nd number).

This means that the ciphertext hasn't been trimmed by the modulus and that a simple root to the 4th power will bring it back to the plaintext value.

Here is how I express that in code:

```python
lines = []

with open("JumbleBumble.txt", "r") as f:
    lines = f.readlines()

# https://stackoverflow.com/a/356206
def find_invpow(x,n):
    """Finds the integer component of the n'th root of x,
    an integer such that y ** n <= x < (y + 1) ** n.
    """
    high = 1
    while high ** n <= x:
        high *= 2
    low = high/2
    while low < high:
        mid = int((low + high) // 2) + 1
        if low < mid and mid**n < x:
            low = mid
        elif high > mid and mid**n > x:
            high = mid
        else:
            return mid
    return mid + 1

while len(lines):
    if lines[-1] == '\n':
        lines.pop()
```

```
    c = int(lines.pop()) # We get the 3rd number
    e = int(lines.pop()) # We get the 2nd number
    n = int(lines.pop()) # We get the 1st number

    m = int(find_invpow(c, 4)) # We put the ciphertext to the 4th root in order to
get our plaintext
    print(bytes.fromhex(hex(m)[2:])) # This is just converting the plaintext to a
readable format
```

Running this one of the outputs stands out:

```
b'VishwaCTF{c4yp70gr2phy_1s_n07_e25y}'
```

Our flag is: **VishwaCTF{c4yp70gr2phy_1s_n07_e25y}**
