Erel Regev

Table of Contents

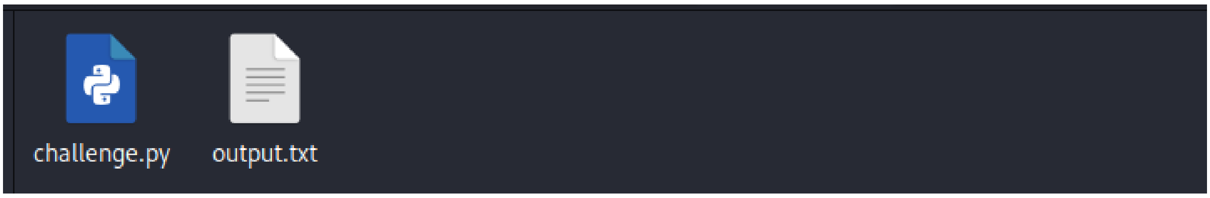
Intro

Challenge description:

CHALLENGE DESCRIPTION

I encrypted a secret message with RSA but I lost the modulus. Can you help me recover it?

Received files:



Viewing output.txt:

```
Flag: 05c61636499a82088bf4388203a93e67bf046f8c49f62857681ec9aaaa40b4772933e0abc83e938c84ff8e67e5ad85bd6eca167585b0cc03eb1333b1b1462d9d7c25f44e53bcb568f0f05219c0147f7dc3cbad45dec2f34
```

Viewing challenge.py:

```python
#!/usr/bin/python3
from Crypto.Util.number import getPrime, long_to_bytes, inverse
flag = open('flag.txt', 'r').read().strip().encode()

class RSA:
    def __init__(self):
        self.p = getPrime(512)
        self.q = getPrime(512)
        self.e = 3
        self.n = self.p * self.q
        self.d = inverse(self.e, (self.p-1)*(self.q-1))
    def encrypt(self, data: bytes) -> bytes:
        pt = int(data.hex(), 16)
        ct = pow(pt, self.e, self.n)
        return long_to_bytes(ct)
    def decrypt(self, data: bytes) -> bytes:
        ct = int(data.hex(), 16)
        pt = pow(ct, self.d, self.n)
        return long_to_bytes(pt)

def main():
    crypto = RSA()
    print ('Flag:', crypto.encrypt(flag).hex())

if __name__ == '__main__':
    main()
```

Challenge.py

```python
#!/usr/bin/python3
from Crypto.Util.number import getPrime, long_to_bytes, inverse
flag = open('flag.txt', 'r').read().strip().encode()

class RSA:
    def __init__(self):
        self.p = getPrime(512)
        self.q = getPrime(512)
        self.e = 3
        self.n = self.p * self.q
        self.d = inverse(self.e, (self.p-1)*(self.q-1))
    def encrypt(self, data: bytes) -> bytes:
        pt = int(data.hex(), 16)
        ct = pow(pt, self.e, self.n)
        return long_to_bytes(ct)
    def decrypt(self, data: bytes) -> bytes:
        ct = int(data.hex(), 16)
        pt = pow(ct, self.d, self.n)
        return long_to_bytes(pt)

def main():
    crypto = RSA()
    print ('Flag:', crypto.encrypt(flag).hex())

if __name__ == '__main__':
    main()
```

This Python script appears to implement a simple RSA encryption and decryption scheme. Let's break down the script and understand how it works:

- The script imports necessary functions from the Crypto.Util.number module, which provides various number-theoretic operations used in cryptography, like generating prime numbers and converting between integers and bytes.
- The flag to be encrypted is read from a file called 'flag.txt' and is stored as bytes.

- The RSA class is defined with the following methods:

- __init__: Initializes the RSA instance by generating two random 512-bit prime numbers (p and q) and calculating the modulus n and private exponent d based on these primes and a fixed public exponent e (set to 3 in this case).

- encrypt: Takes data as bytes, converts it to an integer (pt), then encrypts it using the public key (e, n) by calculating the ciphertext (ct) as ct = pt^e mod n.

- decrypt: Takes ciphertext data as bytes, converts it to an integer (ct), then decrypts it using the private key (d, n) by calculating the plaintext (pt) as pt = ct^d mod n.

- The main function creates an instance of the RSA class, encrypts the flag using the encrypt method, and prints the encrypted flag in hexadecimal format.

- The if __name__ == '__main__': block ensures that the main function is executed only when the script is run directly, not when it's imported as a module.

Erel Regev

dec.py

```python
import gmpy2
from Crypto.Util.number import long_to_bytes

# Encrypted flag in numeric format
encrypted_flag = int("05c61636499a82088bf4388203a93e67bf046f8c49f62857681ec9aaaa40b4772933e0abc83e938c84ff8e67e5ad85bd6eca167585b0cc03eb1333b1b1462d9d7c25f44e53b

# Calculate the cube root using gmpy2
cube_root = gmpy2.iroot(encrypted_flag, 3)

if cube_root[1]:
    decrypted_flag = long_to_bytes(int(cube_root[0]))
    flag_str = decrypted_flag.decode('utf-8')
    if flag_str.startswith('HTB{') and flag_str.endswith('}'):
        print("Decrypted Flag:", flag_str)
    else:
        print("Decryption failed: Incorrect flag syntax.")
else:
    print("Decryption failed: Cube root is not an exact integer.")
```

We import the necessary libraries:

gmpy2: A library for arbitrary-precision arithmetic and number-theoretic functions.

Crypto.Util.number: A module from the PyCryptodome library that provides utility functions for number conversions.

We define the encrypted flag in numeric format:

- The encrypted flag is provided as a long integer in hexadecimal format.
- We convert the hexadecimal string to an integer using the int() function with base 16.
- We calculate the cube root using gmpy2.iroot():

The gmpy2.iroot() function calculates the integer part of the nth root of an integer.

Here, we calculate the cube root (3rd root) of the encrypted flag using gmpy2.iroot(encrypted_flag, 3).

We check if the cube root calculation was successful:

- The gmpy2.iroot() function returns a tuple where the first element is the integer part of the root, and the second element indicates whether the root was exact.
- If the second element of the tuple is True, it means the cube root was exact and the decryption process can proceed.
- If the cube root was exact, we decrypt the flag and check its syntax:

We use the long_to_bytes() function from Crypto.Util.number to convert the decrypted integer back to bytes.

We then decode the bytes as UTF-8 to get a string representation of the flag.

We check if the decrypted flag starts with the expected prefix "HTB{" and ends with the expected suffix "}".

If the syntax is correct, we print the decrypted flag; otherwise, we print a failure message.

If the cube root was not exact, we print a failure message indicating that the decryption failed.

```
┌──(kali㉿kali)-[~/…/HTB/Challenges/Lost_Modulus/Lost Modulus]
└─$ python3 dec.py
Decrypted Flag: HTB{[redacted]4}
```