Erel Regev

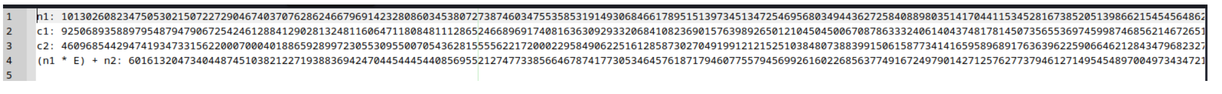
Table of Contents

Intro

Challenge description:

Received files:

Viewing output.txt:



```
n1: 1013026082347505302150722729046740370762862466796914232808603453807273874603475535853191493068466178951513973451347254695680349443627258408898035141704411534528167385205139866215454564862
c1: 9250689358897954879479067254246128841290281324811606471180848111286524668969174081636309293320684108236901576398926501210450450067087863332406140437481781450735655369745998746856214672651
c2: 4609685442947419347331562200070004018865928997230553095500705436281555562217200022958490622516128587302704919912121525103848073883991506158773414165958968917636396225906646212843479682327
(n1 * E) + n2: 60161320473404487451038212271938836942470445444544085695521274773385664678741773053464576187179460775579456992616022685637749167249790142712576277379461271495454897004973434721
```

Viewing RSAisEasy.py:

```python
#!/usr/bin/env python3
from Crypto.Util.number import bytes_to_long, getPrime
from secrets import flag1, flag2
from os import urandom

flag1 = bytes_to_long(flag1)
flag2 = bytes_to_long(flag2)

p, q, z = [getPrime(512) for i in range(3)]

e = 0x10001

n1 = p * q
n2 = q * z

c1 = pow(flag1, e, n1)
c2 = pow(flag2, e, n2)

E = bytes_to_long(urandom(69))

print(f'n1: {n1}')
print(f'c1: {c1}')
print(f'c2: {c2}')
print(f'(n1 * E) + n2: {n1 * E + n2}')

```

RSAisEasy.py


```python
#!/usr/bin/env python3
from Crypto.Util.number import bytes_to_long, getPrime
from secrets import flag1, flag2
from os import urandom

flag1 = bytes_to_long(flag1)
flag2 = bytes_to_long(flag2)

p, q, z = [getPrime(512) for i in range(3)]

e = 0x10001

n1 = p * q
n2 = q * z

c1 = pow(flag1, e, n1)
c2 = pow(flag2, e, n2)

E = bytes_to_long(urandom(69))

print(f'n1: {n1}')
print(f'c1: {c1}')
print(f'c2: {c2}')
print(f'(n1 * E) + n2: {n1 * E + n2}')

```

Erel Regev

Converting flags to long integers:

flag1 and flag2 are converted to long integers using the bytes_to_long function from Crypto.Util.number.

Generating three random 512-bit prime numbers:

Three random prime numbers (p, q, and z) are generated using the getPrime function. These primes will be used to create different RSA moduli.

Setting the public exponent:

The public exponent e is set to 65537 (0x10001), which is a common choice for RSA encryption.

Calculating the RSA moduli:

Two different RSA moduli (n1 and n2) are calculated by multiplying the prime numbers.

- One uses p and q.
- The other uses q and z.
-

This creates two distinct moduli.

Encrypting the flags:

The flags flag1 and flag2 are encrypted using the pow function with the exponent e and the respective RSA moduli (n1 and n2).

Generating a random value E:

A random 69-byte value E is generated using urandom. This value will be used to add randomness to the challenge.

Printing the challenge parameters:

The script prints the values of n1, c1, c2, and the expression (n1 * E) + n2.

dec.py

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

# Given values
n1 = 1013026082347505302150722729046740370762862466796914232808603453807273874603475535853191493068466178951513973451347254695680349443627258408898035141704
c1 = 9250689358897954879479067254246128841290281324811606471180848111286524668969174081636309293320684108236901576398926501210450450067087863332406140437481
c2 = 4609685442947419347331562200070004018865928997230553095500705436281555562217200022958490622516128587302704919912121525103848073883991506158773414165958
t = 60161320473404487451038212271938836942470445444544085695521274773385664678741773053464576187179460775579456992616022685637749167249790142712576277379461

# GCD of n1 and t to find q
q = gmpy2.gcd(n1, t)

# Calculate p and a
p = n1 // q
a = t // q

# Calculate z
z = a % p

# Given e value
e = 0x10001

```

```python

# Calculate phi1 and find the modular inverse of e mod phi1
phi1 = (p - 1) * (q - 1)
d1 = pow(e, -1, phi1)

# Decrypt c1 to get flag1
flag1 = pow(c1, d1, n1)

# Calculate n2 and phi2
n2 = q * z
phi2 = (q - 1) * (z - 1)

# Calculate the modular inverse of e mod phi2
d2 = pow(e, -1, phi2)

# Decrypt c2 to get flag2
flag2 = pow(c2, d2, n2)

# Print the decrypted flags
print("FLAG1 =", long_to_bytes(flag1))
print("FLAG2 =", long_to_bytes(flag2))
```

- The given values n1, c1, c2, and t are provided in the challenge.
- The greatest common divisor (gcd) of n1 and t is calculated to determine q.
- Using q, p is calculated by dividing n1 by q.
- a is calculated by dividing t by q.
- z is calculated as the remainder when a is divided by p.
- The value e is provided (0x10001) which is often used as the public exponent in RSA.
- The Euler's totient function phi1 is calculated using (p - 1) * (q - 1).
- The modular inverse of e modulo phi1 is calculated using the pow function, giving us d1.
- The encrypted message c1 is decrypted using d1 and n1, resulting in flag1.
- n2 is calculated by multiplying q and z.
- The Euler's totient function phi2 is calculated using (q - 1) * (z - 1).
- The modular inverse of e modulo phi2 is calculated using the pow function, giving us d2.
- The encrypted message c2 is decrypted using d2 and n2, resulting in flag2.
- The decrypted flags are then printed using the long_to_bytes function, converting the numerical values to strings.

This script performs the RSA decryption for both flag1 and flag2 using the provided values. The decrypted flags are then printed, completing the decryption process.

```
┌──(kali㉿kali)-[~/…/HTB/Challenges/RSAisEasy/RSAisEasy]
└─$ python3 dec.py
FLAG1 = b'HTB{1_[redacted]d'
FLAG2 = b'_[redacted]t?}'
```

```
┌──(kali㉿kali)-[~/…/HTB/Challenges/RSAisEasy/RSAisEasy]
└─$ HTB{1[redacted]t?}
```