Cryptohack RSA

-

Challenge Room Walkthrough

Pawe Murdzek
310850

Warsaw University of Technology

December 19, 2025

Contents

Introduction . . . 2
1. Starter . . . 2
1.1. Modular Exponentiation . . . 2
1.2. Public Keys . . . 2
1.3. Euler's Totient . . . 2
1.4. Private Keys . . . 2
1.5. RSA Decryption . . . 2
1.6. RSA Signatures . . . 3
2. Primes Part 1 . . . 3
2.1. Factoring . . . 3
2.2. Inferius Prime . . . 3
2.3. Monoprime . . . 3
2.4. Square Eyes . . . 3
2.5. Manyprime . . . 4
3. Public Exponent . . . 4
3.1. Salty . . . 4
3.2. Modulus Inutilis . . . 4
3.3. Everything is Big . . . 4
3.4. Crossed Wires . . . 4
3.5. Everything is Still Big . . . 5

Introduction

This document serves as a walkthrough for the "RSA" challenge category on the Cryptohack platform.

1. Starter

1.1. Modular Exponentiation

— **Code:**
```python
print(pow(101,17,22663))
```
— **Flag:** `19906`
— **Explanation:** Modular exponentiation ($m^e \pmod n$) is the fundamental operation in RSA. Python's `pow(base, exp, mod)` uses the efficient binary exponentiation algorithm.

1.2. Public Keys

— **Code:**
```python
e = 65537
p = 17
q = 23
n = p*q
m = 12
print(pow(m,e,n))
```
— **Flag:** `301`
— **Explanation:** The public key consists of (e, n). Encryption is $c = m^e \pmod n$. In this case, n is the product of two small primes.

1.3. Euler's Totient

— **Code:**
```python
p = 857504083339712752489993810777
q = 1029224947942998075080348647219
print((p-1)*(q-1))
```
— **Flag:** `882564595536224140639625987657529300394956519977044270821168`
— **Explanation:** $\phi(n)$ counts numbers less than n coprime to n. For $n = pq$, $\phi(n) = (p-1)(q-1)$. It is required to compute the private key.

1.4. Private Keys

— **Code:**
```python
p = 857504083339712752489993810777
q = 1029224947942998075080348647219
phi = (p-1)*(q-1)
e = 65537
d = pow(e,-1,phi)
print(d)
```
— **Flag:** `121832886702415731577073962957377780195510499965398469843281`
— **Explanation:** The private key d is the modular inverse of $e \pmod{\phi(n)}$. It allows reversing the encryption operation.

1.5. RSA Decryption

— **Code:**
```python
n = 882564595536224140639625987659416029426239230804614613279163
e = 65537
p = 857504083339712752489993810777
q = 1029224947942998075080348647219
c = 77578995801157823671636298847186723593814843845525223303932
phi = (p-1)*(q-1)
d = pow(e,-1,phi)
answer = pow(c,d,n)
print(answer)
```
— **Flag:** `13371337`
— **Explanation:** Decryption is $m = c^d \pmod n$. Given the factors p, q, we calculate d and recover the original message.

1.6. RSA Signatures

— **Code:**

```
from Crypto.Hash import SHA256
from Crypto.Util.number import bytes_to_long

n = 152165...3803 # (long N)
d = 111759...2689 # (long D)

hash_obj = SHA256.new(data=b'crypto{Immut4ble_m3ssag1ng}')
s = pow(bytes_to_long(hash_obj.digest()), d, n)
print(s)
```

— **Flag:** `1348073840459009...475`

— **Explanation:** A signature is a hash of the message "decrypted" with the private key. Verification involves "encrypting" the signature with the public key and comparing hashes.

2. Primes Part 1

2.1. Factoring

— **Code:**

```
from factordb.factordb import FactorDB
N = 510143758735509025530880200653196460532653147
f = FactorDB(N)
f.connect()
print(min(f.get_factor_list()))
```

— **Flag:** `19704762736204164635843`

— **Explanation:** RSA is vulnerable if N can be factored. Public databases like FactorDB store factors for known or weak moduli.

2.2. Inferius Prime

— **Code:**

```
from Crypto.Util.number import inverse, long_to_bytes
e = 3
n = 742449129124467073921545687640895127535705902454369756401331
ciphertext = 39207274348578481322317340648475596807303160111338236677373
p = 752708788837165590355094155871
q = 986369682585281993933185289261
phi = (p - 1) * (q - 1)
d = inverse(e, phi)
plaintext = pow(ciphertext, d, n)
print("Decoded plaintext:", long_to_bytes(plaintext).decode())
```

— **Flag:** `crypto{N33d_b1g_pR1m35}`

— **Explanation:** When factors p and q are known, calculating the private key d and decrypting the flag is trivial.

2.3. Monoprime

— **Code:**

```
n = 171731...591
e = 65537
ct = 161367...942
d = pow(e, -1, n-1)
m = pow(ct, d, n)
print(bytes.fromhex(hex(m)[2:]).decode())
```

— **Flag:** `crypto{0n3_pr1m3_41n7_pr1m3_l01}`

— **Explanation:** If n is prime, then $\phi(n) = n - 1$. This violates the requirement for n to be a product of two primes, making it trivial to find d.

2.4. Square Eyes

— **Code:**

```
from Crypto.Util.number import *
from math import isqrt
N = 535860...449
e = 65537
c = 222502...896
p = isqrt(N)
phi = p * (p - 1)
d = inverse(e, phi)
print(long_to_bytes(pow(c, d, N)).decode())
```

— **Flag:** `crypto{squar3_r00t_i5_f4st3r_th4n_f4ct0r1ng!}`

— **Explanation:** If $n = p^2$, then $\phi(n) = p(p - 1)$. Factoring is just a matter of calculating the integer square root.

2.5. Manyprime

— **Code:**

```
from Crypto.Util.number import long_to_bytes
n = 580642...37
e = 65537
ct = 320721...64
factorize = [9282...21, ..., 1728...49]
phi = 1
for i in factorize:
    phi *= (i-1)
d = pow(e, -1, phi)
print(long_to_bytes(pow(ct, d, n)))
```

— **Flag:** `crypto{700_m4ny_5m4ll_f4c70r5}`

— **Explanation:** Multi-prime RSA uses $n = p_1 p_2 \dots p_k$. The totient is $\phi(n) = \prod(p_i - 1)$. If all factors are known, d can be derived.

3. Public Exponent

3.1. Salty

— **Code:**

```
from Crypto.Util.number import long_to_bytes
ct = 44981230718212183604274785925793145442655465025264554046028251311164494127485
print(long_to_bytes(ct).decode())
```

— **Flag:** `crypto{saltstack_fell_for_this!}`

— **Explanation:** Using $e = 1$ means $c \equiv m \pmod{n}$. No encryption actually occurs.

3.2. Modulus Inutilis

— **Code:**

```
import gmpy2
from Crypto.Util.number import long_to_bytes
ct = 24325...957
pt, exact = gmpy2.iroot(ct, 3)
if exact:
    print(long_to_bytes(int(pt)).decode())
```

— **Flag:** `crypto{N33d_m04R_p4dd1ng}`

— **Explanation:** If e is very small and $m^e < n$, the modulo operation is irrelevant. The plaintext is simply the e-th root of the ciphertext.

3.3. Everything is Big

— **Code:**

```
from Crypto.Util.number import long_to_bytes, inverse

def convergents(e_list):
    n, d = [], []
    for i in range(len(e_list)):
        if i == 0: ni, di = e_list[i], 1
        elif i == 1: ni, di = e_list[i]*e_list[i-1] + 1, e_list[i]
        else:
            ni = e_list[i]*n[i-1] + n[i-2]
            di = e_list[i]*d[i-1] + d[i-2]
        n.append(ni); d.append(di)
    return n, d

def get_cf_expansion(x, y):
    cf = []
    while y:
        cf.append(x // y)
        x, y = y, x % y
    return cf

# ... (logic loop using continued fractions for Wiener's Attack)
```

— **Flag:** `crypto{s0m3th1ng5_c4n_b3_t00_b1g}`

— **Explanation:** Wiener's Attack uses Continued Fractions to recover d when it is very small ($d < \frac{1}{3}n^{1/4}$).

3.4. Crossed Wires

— **Code:**

```
import math, random
from Crypto.Util.number import inverse, long_to_bytes

def factor_n_from_d(n, d, e=65537):
    k = d * e - 1
    while True:
        g = random.randint(2, n - 1)
        t = k
        while t % 2 == 0:
            t //= 2
            x = pow(g, t, n)
            if x > 1:
                y = math.gcd(x - 1, n)
```

```
                if 1 < y < n: return y, n // y

p, q = factor_n_from_d(N, d)
phi = (p-1)*(q-1)
e_total = math.prod(friend_exponents)
d_final = inverse(e_total, phi)
print(long_to_bytes(pow(cipher, d_final, N)).decode())
```

- **Flag:** `crypto{3ncrypt_y0ur_s3cr3t_w1th_y0ur_fr1end5_publ1c_k3y}`
- **Explanation:** Knowing d for a specific e allows factoring N. Once factored, we can calculate the modular inverse for any other exponent used to encrypt the message.

3.5. Everything is Still Big

- **Code:**

```
from Crypto.Util import number
# N, e, c, p, q provided
phi = (q-1)*(p-1)
d = number.inverse(e, phi)
print(number.long_to_bytes(pow(c, d, N)).decode())
```

- **Flag:** `crypto{bon3h5_4tt4ck_i5_sr0ng3r_th4n_w13n3r5}`
- **Explanation:** This challenge requires the Boneh-Durfee attack, which is a lattice-based approach to break RSA when d is small ($d < n^{0.292}$).