Cryptohack General

-

Challenge Room Walkthrough

Paweł Murdzek

Warsaw University of Technology

November 4, 2025

Contents

Introduction

This document provides a walkthrough for the "Mathematics" challenge category on the Cryptohack platform.

1. Modular Math

1.1. Quadratic Residues

— **Code:**

```python
# Set the given values
p = 29
numbers_to_check = [14, 6, 11]

# memory
found_residue = None
root1 = None
root2 = None

# Loop through all possible roots 'a' from 1 to p-1
for a in range(1, p):
    # Calculate the square of 'a' modulo p
    square = (a * a) % p

    # Check if this square is one of the numbers we're looking for
    if square in numbers_to_check:
        found_residue = square
        root1 = a
        root2 = p - a
        break

# Print the results
if found_residue is not None:
    print(f"p = {p}")
    print(f"List of numbers = {numbers_to_check}")
    print("---------------------------------")
    print(f"Found Quadratic Residue: {found_residue}")
    print(f"The square roots are: {root1} and {root2}")

    # Calculate the smaller root for the flag
    flag = min(root1, root2)
    print(f"The smaller square root (the flag) is: {flag}")
else:
    print("Could not find a quadratic residue in the list.")
```

— **Flag:** `8`

— **Explanation:** The code implements a "brute-force" method to find a **quadratic residue**. A quadratic residue x modulo p is a number for which there exists a number a such that $a^2 \equiv x \pmod{p}$. The loop `for a in range(1, p)` iterates through all possible values of a (from 1 to $p-1$). Inside the loop, the square is calculated: `square = (a * a) % p`. The code then checks if this calculated square is on the list of candidates (`numbers_to_check`). If the square a^2 is equal to one of the target numbers, it means that number is a quadratic residue. The code then saves both square roots: a and $p-a$. This is the correct second root because $(p-a)^2 \equiv (-a)^2 \equiv a^2 \pmod{p}$. The flag is the smaller of these two roots.

1.2. Legendre Symbol

— **Code:**

```python
import ast

def solve_qr_large(p, candidates):
    """Finds quadratic residue and calculates its largest square root modulo p."""
    if p % 4 != 3:
        print(f"Error: p={p} does not satisfy p % 4 == 3")
        return

    print(f"Prime p = {p} (p % 4 = {p % 4})\n")

    # Find quadratic residue using Euler's Criterion
    exponent_test = (p - 1) // 2
    found_residue = None

    for a in candidates:
        result = pow(a, exponent_test, p)
        if result == 1:
            print(f"[+] Found QR: {a}")
            found_residue = a
            break
        print(f"[-] Not QR: {a}")

    if not found_residue:
        print("No quadratic residue found")
        return

    # Calculate square roots: a^((p+1)/4) % p
    root1 = pow(found_residue, (p + 1) // 4, p)
    root2 = p - root1
    flag = max(root1, root2)

    print(f"\nRoots: {root1}, {root2}")
    print(f"Flag (larger root): {flag}")
    return flag

def parse_input_file(filename="output.txt"):
```

```
    """Reads and parses p and list of integers from file."""
    try:
        with open(filename, 'r') as f:
            lines = [line.strip() for line in f if line.strip()]

        if len(lines) < 2:
            print(f"Error: File must contain at least 2 lines")
            return None, None

        p_line = next((line for line in lines if line.startswith(('p =', 'p='))), None)
        ints_line = next((line for line in lines if line.startswith(('ints =', 'ints='))), None)

        if not p_line or not ints_line:
            print("Error: Could not find 'p =' or 'ints =' in file")
            return None, None

        p = int(p_line.split('=', 1)[1].strip())
        candidates = ast.literal_eval(ints_line.split('=', 1)[1].strip())

        print(f"Successfully parsed '{filename}'")
        return p, candidates

    except FileNotFoundError:
        print(f"Error: File '{filename}' not found")
        return None, None
    except Exception as e:
        print(f"Error parsing file: {e}")
        return None, None

if __name__ == "__main__":
    p, candidates = parse_input_file("output.txt")
    if p and candidates:
        solve_qr_large(p, candidates)
```

— **Flag:** 93291799125366706806545638475797430512104976066103610269938025709952247020061090804870186195285

— **Explanation:** This code uses **Euler's Criterion** to quickly check if a number a is a quadratic residue modulo p. Instead of calculating all squares $x^2 \pmod p$, it calculates the **Legendre Symbol** $\left(\frac{a}{p}\right)$, which is defined as:

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod p$$

In the code, this corresponds to `exponent_test = (p - 1) // 2` and `result = pow(a, exponent_test, p)`.

— If `result == 1`, a is a quadratic residue.

— If `result == p - 1` (which is -1), a is a quadratic non-residue.

When a quadratic residue a is found, the code must calculate its square roots. It uses a special property of prime numbers p that satisfy the condition $p \equiv 3 \pmod 4$. For such primes, there is a simple formula to calculate the square roots of a:

$$x \equiv \pm a^{(p+1)/4} \pmod p$$

In the code, this corresponds to `root1 = pow(found_residue, (p + 1) // 4, p)`, and the second root is `root2 = p - root1`. The flag is the larger of the calculated roots.

1.3. Modular Square Root

— **Code:**

```
import re

def tonelli_shanks(a, p):
    """Tonelli-Shanks algorithm for finding modular square root when p % 4 == 1."""
    if pow(a, (p - 1) // 2, p) != 1:
        return None

    # Factor p - 1 as Q * 2^S
    Q, S = p - 1, 0
    while Q % 2 == 0:
        Q //= 2
        S += 1

    # Find quadratic non-residue
    z = 2
    while pow(z, (p - 1) // 2, p) == 1:
        z += 1

    # Initialize
    c = pow(z, Q, p)
    R = pow(a, (Q + 1) // 2, p)
    t = pow(a, Q, p)
    M = S

    while t != 1:
        # Find smallest i where t^(2^i) == 1
        i, temp_t = 0, t
        while temp_t != 1:
            temp_t = pow(temp_t, 2, p)
            i += 1

        # Update values
        b = pow(c, 1 << (M - i - 1), p)
        R = (R * b) % p
        t = (t * b * b) % p
        c = (b * b) % p
        M = i

    return R
```

```python
def find_modular_sqrt(a, p):
    """Find modular square root. Returns the smaller of the two roots."""
    a = a % p
    if a == 0:
        return 0

    if p % 4 == 3:
        r = pow(a, (p + 1) // 4, p)
    elif p % 4 == 1:
        r = tonelli_shanks(a, p)
    else:
        return None

    return min(r, p - r) if r else None

def main():
    with open('output.txt', 'r') as f:
        content = f.read()

    # Parse a and p from file
    a = int(re.search(r'a\s*=\s*(\d+)', content).group(1))
    p = int(re.search(r'p\s*=\s*(\d+)', content).group(1))

    print(f"a = {str(a)[:60]}...")
    print(f"p = {str(p)[:60]}...")
    print(f"p % 4 = {p % 4} ({'Tonelli-Shanks' if p % 4 == 1 else 'Simple formula'})\n")

    # Calculate modular square root
    result = find_modular_sqrt(a, p)

    if result is None:
        print("No square root exists")
    else:
        print(f"The smaller of the two roots is: {result}")
        print(f"\nVerification: {result}^2 % p == a ? {pow(result, 2, p) == a % p}")

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

— **Flag:** 236233930768304863832777329858048929893213750552050038833827105205373474786235177964731417681795

— **Explanation:** This code calculates the modular square root modulo p for any prime number p.

1. **Case** $p \equiv 3 \pmod 4$: This is the simple case, the same as in the previous task. The roots are calculated directly from the formula $x \equiv \pm a^{(p+1)/4} \pmod p$.
2. **Case** $p \equiv 1 \pmod 4$: In this situation, the simple formula does not work. The code must implement the **Tonelli-Shanks algorithm**, which is a general method for finding square roots modulo a prime number.
   — First, the algorithm checks if a is a quadratic residue at all, using Euler's Criterion (`pow(a, (p - 1) // 2, p) != 1`).
   — Then, it factors $p-1$ into $Q \cdot 2^S$, where Q is odd.
   — It finds z, any **quadratic non-residue** modulo p.
   — It initializes the variables c, R, t, M.
   — In the main loop `while t != 1`, the algorithm iteratively updates these variables, "correcting" the root candidate R, until t becomes 1. The final value of R is the desired root.

Finally, the code returns the smaller of the two roots: R and $p - R$.

1.4. Chinese Remainder Theorem

— **Code:**

```python
from functools import reduce
from operator import mul

def solve_crt(remainders, moduli):
    """Solves system of congruences using Chinese Remainder Theorem."""
    N = reduce(mul, moduli)
    return sum(a * (N_i := N // n) * pow(N_i, -1, n) for a, n in zip(remainders, moduli)) % N

remainders = [2, 3, 5]
moduli = [5, 11, 17]

result = solve_crt(remainders, moduli)

print(f"System: z  {remainders[0]} (mod {moduli[0]}), z  {remainders[1]} (mod {moduli[1]}), z  {remainders[2]} (mod {moduli[2]})")
print(f"Solution: {result}\n")
print(f"Verification: {result} % {moduli[0]} = {result % moduli[0]}, {result} % {moduli[1]} = {result % moduli[1]}, {result} % {moduli[2]} = {result % moduli[2]}")
```

— **Flag:** 872

— **Explanation:** The code is a direct, constructive implementation of the **Chinese Remainder Theorem (CRT)**. The goal is to find a single number z that satisfies a system of congruences (gives specific remainders a_i when divided by different, pairwise coprime moduli n_i).

$$\begin{cases} z \equiv 2 \pmod 5 \\ z \equiv 3 \pmod{11} \\ z \equiv 5 \pmod{17} \end{cases}$$

The formula used in the code is the standard solution to CRT:

$$z \equiv \sum_i a_i \cdot N_i \cdot M_i \pmod{N}$$

Where:
— `a` (a_i) is the required remainder (e.g., 2, 3, 5).
— `n` (n_i) is the modulus (e.g., 5, 11, 17).
— `N` is the product of all moduli: $N = \prod n_i = 5 \cdot 11 \cdot 17$.
— `N_i` (in code `N_i := N // n`) is N/n_i (e.g., $N_1 = N/5 = 11 \cdot 17$).
— `pow(N_i, -1, n)` is M_i, which is the **modular inverse** of N_i modulo n_i (e.g., $M_1 = (11 \cdot 17)^{-1} \pmod 5$).
Each term in the sum $a_i \cdot N_i \cdot M_i$ is constructed such that it is equal to $a_i \pmod{n_i}$ and simultaneously $0 \pmod{n_j}$ for all $j \neq i$. Summing these terms gives the final solution z, which satisfies all conditions at the same time.

2. Brainteasers Part 1

2.1. Successive Powers

— **Pen-and-Paper Solution:** Let the given sequence be $S = \{s_1, s_2, s_3, \ldots\}$. The problem states that these are successive powers of an integer z modulo a prime p. This gives the relation:

$$s_{i+1} \equiv s_i \cdot z \pmod{p}$$

The sequence is: $\{588, 665, 216, 113, 642, 4, 836, 114, \ldots\}$

Finding z
The "on paper" hint suggests looking for a simple relationship. Let's check the sixth and seventh terms:
— $s_6 = 4$
— $s_7 = 836$
Using our relation:

$$s_7 \equiv s_6 \cdot z \pmod{p}$$
$$836 \equiv 4 \cdot z \pmod{p}$$

This congruence means that $836 = 4z + k \cdot p$ for some integer k. The simplest case, and the one suggested by the hint, is to assume $k = 0$.

$$836 = 4z$$
$$z = \frac{836}{4}$$
$$\mathbf{z = 209}$$

Finding p
Now that we have $z = 209$, we can find p. The relation $s_{i+1} \equiv s_i \cdot z \pmod{p}$ can be rewritten as:

$$s_i \cdot z - s_{i+1} \equiv 0 \pmod{p}$$

This means that p must be a divisor of the integer $s_i \cdot z - s_{i+1}$ for every i. Therefore, p must be a common divisor of all such expressions. We can find p by calculating the Greatest Common Divisor (GCD) of a few of these expressions.

Expression 1 (using $s_1 = 588$ and $s_2 = 665$)

$$X_1 = s_1 \cdot z - s_2$$
$$X_1 = 588 \cdot 209 - 665$$
$$X_1 = 122892 - 665 = 122227$$

Expression 2 (using $s_4 = 113$ and $s_5 = 642$)

$$X_2 = s_4 \cdot z - s_5$$

$$X_2 = 113 \cdot 209 - 642$$

$$X_2 = 23617 - 642 = 22975$$

Calculating GCD
Now we find $p = \text{GCD}(122227, 22975)$ using the Euclidean Algorithm.

$$122227 = 5 \cdot 22975 + 7352$$
$$22975 = 3 \cdot 7352 + 919$$
$$7352 = 8 \cdot 919 + 0$$

The GCD is 919. This is a three-digit prime number, which matches the problem description.

Conclusion
The solution is:

$$\mathbf{p = 919}$$

$$\mathbf{z = 209}$$

— **Flag:** `crypto{919,209}`
— **Explanation:** (Explanation is already included in the problem text)

2.2. Adrien's Signs

— **Code:**

```python
from random import randint

a = 288260533169915
p = 1007621497415251

FLAG = b'crypto{????????????????????}'


def encrypt_flag(flag):
    ciphertext = []
    plaintext = ''.join([bin(i)[2:].zfill(8) for i in flag])
    for b in plaintext:
        e = randint(1, p)
        n = pow(a, e, p)
        if b == '1':
            ciphertext.append(n)
        else:
            n = -n % p
            ciphertext.append(n)
    return ciphertext


# Legendre symbol function
def legendre(a, p):
    """Compute the Legendre symbol (a/p)"""
    return pow(a, (p - 1) // 2, p)


def decrypt_flag(ciphertext, p):
    """Decrypt the ciphertext using Legendre symbol"""
    binary_string = ''
    for n in ciphertext:
        # Check if n is a quadratic residue using Legendre symbol
        leg = legendre(n, p)
        if leg == 1:
            # Quadratic residue -> bit is '1'
            binary_string += '1'
        else:
            # Not a quadratic residue -> bit is '0'
            binary_string += '0'

    # Convert binary string to bytes
    flag = ''
    for i in range(0, len(binary_string), 8):
        byte = binary_string[i:i+8]
        flag += chr(int(byte, 2))

    return flag


# Read ciphertext from output.txt
with open('output.txt', 'r') as f:
    content = f.read().strip()
    ciphertext = eval(content)

decrypted_flag = decrypt_flag(ciphertext, p)
print(f"Decrypted flag: {decrypted_flag}")
```

— **Flag:** `crypto{p4tterns_1n_re5idu3s}`

— **Explanation:** Encryption and decryption are based entirely on the properties of the **Legendre Symbol** $\left(\frac{a}{p}\right)$.

1. **Configuration of p**: The challenge uses a prime p such that $p \equiv 3 \pmod 4$. This has a key consequence: $(p-1)/2$ is an odd number. From Euler's Criterion, it follows that $\left(\frac{-1}{p}\right) \equiv (-1)^{(p-1)/2} \equiv -1 \pmod p$. This means that -1 is a **quadratic non-residue** modulo p.
2. **Configuration of a**: The value a must be a **quadratic residue** (its Legendre symbol $\left(\frac{a}{p}\right) = 1$). Consequently, $n \equiv a^e \pmod p$ is *always* a quadratic residue, regardless of e, because $\left(\frac{n}{p}\right) = \left(\frac{a^e}{p}\right) = \left(\frac{a}{p}\right)^e = 1^e = 1$.
3. **Encrypting bit '1'** (per `encrypt_flag`): Ciphertext $c = n$. We calculate the Legendre symbol of c: $\left(\frac{c}{p}\right) = \left(\frac{n}{p}\right) = 1$.
4. **Encrypting bit '0'** (per `encrypt_flag`): Ciphertext $c = -n \pmod p$. We calculate the Legendre symbol of c: $\left(\frac{c}{p}\right) = \left(\frac{-n}{p}\right) = \left(\frac{-1}{p}\right)\left(\frac{n}{p}\right) = (-1) \cdot (1) = -1$.
5. **Decryption** (`decrypt_flag`): The code reverses this process. For each number n (which is c) in the ciphertext, it calculates its Legendre symbol `leg = legendre(n, p)` (using Euler's Criterion: `pow(a, (p - 1) // 2, p)`).
   — If `leg == 1`, it means it's a quadratic residue, and thus bit '1'.
   — If `leg != 1` (meaning it's equal to $p-1$, which corresponds to -1), it means it's a quadratic non-residue, and thus bit '0'.

2.3. Modular Binomials

— **Code:**

```
from math import gcd
"""
    c1 = (2p + 3q) ^ e1 % N
    c2 = (5p + 7q) ^ e2 % N
    ------------------------------------------------------------------------------------
    N = pq => Binomial: mid elements % N == 0
=>  c1 = (2p) ^ e1  +  (3q) ^ e1 % N
    c2 = (5p) ^ e2  +  (7q) ^ e2 % N
    ------------------------------------------------------------------------------------
    raise c1 to e2 and c2 to e1: to create similar exponent
    c1 ^ e2 = (2p) ^ (e1e2) + (3q) ^ (e1e2) % N  (1)
    c2 ^ e1 = (5p) ^ (e1e2) + (7q) ^ (e1e2) % N  (2)
    ------------------------------------------------------
    Let (1) * 5 ^ (e1e2) - (2) * 2 ^ (e1e2) to get rid of p:
=>  t = 5 ^ (e1e2) * c1 ^ e2  - 2 ^ (e1e1) * c2 ^ e1 = ( 15 ^ (e1e2) - 14 ^ (e1e2)) * q
    ------------------------------------------------------------------------------------
=>  t % q == 0 and we haved N % q == 0:
=>  q = gcd(t, N)
    p = N / q
"""

N = 14905562257842714057932724129575002825405393502650869767115942606408600343380327866258982402447992564988466588305174271674657844352454543958847568190372446723549627752274442789184236490768272313
e1 = 12886657667389660800780796462970504910193928992888518978200029826975978624718627799215564700096007849924866627154987365059524315097631111242449314835868137
e2 = 12110586673991788415780355139635579057920926864887110308343229256046868242179445444897790171351302575188607117081580121488253540215781625598048021161675697
c1 = 14010729418703228234352465883041270611113735889838753433295478495763409056136734155612156934673988344882629541204985909650433819205298939877837314145082403528055884752079219150739849992921393501
c2 = 14386997138637978860748278986945098648507142864584111124202580365103793165811666987664851210230009375267398957979494066880296418013345006977654742303441030008490816239306394492168516278328851511

x1 = pow(5, e1 * e2, N) * pow(c1, e2, N)
x2 = pow(2, e1 * e2, N) * pow(c2, e1, N)

# t = x1 - x2
q = gcd(x1 - x2, N)

p = N // q
print("crypto{"+str(p)+","+str(q)+"}")
```

— **Flag:** `crypto{112274000169258486390262064441991200608556376127408952701514962644340921899196091557519`

— **Explanation:** This code uses the **Binomial Expansion (Newton's Binomial Theorem)** in the ring modulo $N = pq$.

1. **Key observation**: For $N = pq$, the expression $(ap + bq)^e \pmod N$ simplifies significantly. According to the binomial theorem:
$$(ap + bq)^e = \sum_{k=0}^{e} \binom{e}{k} (ap)^k (bq)^{e-k}$$
2. All terms in this sum, except for the first ($k = e$) and the last ($k = 0$), contain both a factor of p (from $(ap)^k$, where $k \geq 1$) and q (from $(bq)^{e-k}$, where $e - k \geq 1$). This means that all "middle" terms are divisible by $pq = N$, so they vanish (are equal to $0 \pmod N$).
3. Only the outermost terms remain: $(ap)^e + (bq)^e$.
4. Hence we have the simplified congruences: $c_1 \equiv (2p)^{e1} + (3q)^{e1} \pmod N$ $c_2 \equiv (5p)^{e2} + (7q)^{e2} \pmod N$

5. **Elimination of p**: To isolate q, the code brings both expressions to a common exponent e_1e_2 by exponentiating:
   — $c_1^{e2} \equiv (2p)^{e1e2} + (3q)^{e1e2} \pmod{N}$
   — $c_2^{e1} \equiv (5p)^{e1e2} + (7q)^{e1e2} \pmod{N}$
6. It then creates a linear combination that will remove the p term. It multiplies the first equation by 5^{e1e2} and the second by 2^{e1e2} and subtracts them (in the code `t = x1 - x2`):

$$t = 5^{e1e2}c_1^{e2} - 2^{e1e2}c_2^{e1}$$

$$t \equiv [5^{e1e2}(2p)^{e1e2} + 5^{e1e2}(3q)^{e1e2}] - [2^{e1e2}(5p)^{e1e2} + 2^{e1e2}(7q)^{e1e2}] \pmod{N}$$

$$t \equiv [(10p)^{e1e2} + (15q)^{e1e2}] - [(10p)^{e1e2} + (14q)^{e1e2}] \pmod{N}$$

7. The terms containing p (i.e., $(10p)^{e1e2}$) cancel out:

$$t \equiv (15q)^{e1e2} - (14q)^{e1e2} \equiv q^{e1e2}(15^{e1e2} - 14^{e1e2}) \pmod{N}$$

8. This equation shows that t is a multiple of q. Since N is also a multiple of q, q must be a common divisor of t and N.
9. By calculating the **Greatest Common Divisor** $\text{GCD}(t, N)$ (in the code `q = gcd(x1 - x2, N)`), we recover the factor q. With q, we calculate $p = N/q$.

2.4. Broken RSA

— **Code:**

```
# Import the long_to_bytes function from the Crypto.Util.number module
from Crypto.Util.number import long_to_bytes

# p is a large prime number, which serves as the modulus in the RSA-like scheme
p = 27772857409875257529415990911214211975844307184430241451899407838750503024323367895540981606586709985980003435082116995888017731426634845808624796292507989171497629109450825818587383112280639037
# c is the ciphertext, which is the result of encrypting the message m (c = m^e mod p)
c = 11303174761894431146735697569489134747234975144162172162401674567273034831391936916397234068346115459134602443963604063679379285919302225719050193590179240191429612072131629779948379821039610415
# e is the public exponent
e = 16

# This function calculates the modular square root of a number 'a' modulo a prime 'p'.
# It implements the Tonelli-Shanks algorithm.
def modular_sqrt(a, p):
    # First, check if a has a square root modulo p using the Legendre symbol.
    # If legendre_symbol(a, p) is not 1, 'a' is a quadratic non-residue and has no square root.
    if legendre_symbol(a, p) != 1:
        return 0
    elif a == 0:
        return 0
    elif p == 2:
        return 0
    # A simpler case for primes p congruent to 3 modulo 4.
    elif p % 4 == 3:
        return pow(a, (p + 1) // 4, p)

    # Tonelli-Shanks algorithm for other primes.
    # 1. Decompose p-1 into s * 2^e where s is odd.
    s = p - 1
    e_s = 0
    while s % 2 == 0:
        s //= 2
        e_s += 1

    # 2. Find a quadratic non-residue 'n' modulo p.
    n = 2
    while legendre_symbol(n, p) != -1:
        n += 1

    # 3. Initialize the variables for the main loop.
    x = pow(a, (s + 1) // 2, p)
    b = pow(a, s, p)
    g = pow(n, s, p)
    r = e_s

    # 4. Main loop of the algorithm to find the square root.
    while True:
        t = b
        m = 0
        for m in range(r):
            if t == 1:
                break
            t = pow(t, 2, p)

        if m == 0:
            return x

        gs = pow(g, 2 ** (r - m - 1), p)
        g = (gs * gs) % p
        x = (x * gs) % p
        b = (b * g) % p
        r = m

def legendre_symbol(a, p):
    ls = pow(a, (p - 1) // 2, p)
    return -1 if ls == p - 1 else ls


for i in range(e):
    f = c
    chk = i
```

```python
    for j in range(4):
        f = modular_sqrt(f, p)
        if f == 0:
            break
        if chk%2 == 1:
            f = p - f
        chk //= 2
    if f > 0:
        try:
            print(long_to_bytes(f))
        except Exception:
            pass # Ignore conversion errors for non-flag candidates
```

— **Flag:** `crypto{m0dul4r_squ4r3_r00t}`

— **Explanation:** The encryption is $c \equiv m^{16} \pmod{p}$. To recover m, we must calculate the **16th root** of c modulo p.

1. **Problem**: We cannot use the standard RSA mechanism (calculating the private key $d \equiv e^{-1} \pmod{\phi(p)}$), because $e = 16$ is not coprime to $\phi(p) = p - 1$ (both are even). $\mathrm{GCD}(16, p-1) \neq 1$.
2. **Solution**: We use the fact that $16 = 2^4$. Calculating the 16th root is equivalent to **calculating the square root four times**:

$$m \equiv c^{1/16} \equiv \left(\left(\left(c^{1/2}\right)^{1/2}\right)^{1/2}\right)^{1/2} \pmod{p}$$

3. **Ambiguity**: Each square root operation $x = \sqrt{y} \pmod{p}$ (performed by the `modular_sqrt` function using the Tonelli-Shanks algorithm) has two solutions: r and $p - r$ (i.e., $\pm r$).
4. After four such operations, we get $2^4 = 16$ possible candidates for the original message m.
5. **Implementation**: The code systematically checks all 16 possibilities.
   — The loop `for i in range(e)` (where `e=16`) iterates through numbers from 0 to 15.
   — The loop `for j in range(4)` performs four successive square root calculations.
   — The variable `i` (copied to `chk`) acts as a binary counter (from 0000 to 1111). Its bits (checked by `chk%2 == 1`) decide whether to choose the "positive" root f or the "negative" root $p - f$ at each step.
6. In this way, the code tests all 16 possible combinations of $\pm$ signs, until it finds the one that, when converted to bytes (`long_to_bytes`), yields a readable flag.

2.5. No Way Back Home

— **Code:**

```python
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
from Crypto.Util.number import inverse, long_to_bytes
from hashlib import sha256

p, q = (10699940648196411028170713430726559470427113689721202803392638457920771439452897032229838317321639599506283870585924807089941510579727013041135771337631951, 119566763878365121514807449798691
vka = 124641741967121300068241280971408306625050636261192655845274494695382484894973990899018981438824398885984003880665335336872849819983045790478166909381968949910717906136475842568208640203811766
vkakb = 114778245184091677576134046724609868204771151111446457870524843414356897479473739627212552495413311985409829523700919603502616667323311977056345059189257932050632105761365449853358722065048 8
vkb = 656889784012771314738234583279864566711023716801133564063044000658392310250365927310489958482763796192142867733518062042165471200051231000803669302278594531742806625723640933967704113303831708
c_hex = 'fef29e5ff72f28160027959474fc462e2a9e0b2d84b1508f7bd0e270bc98fac942e1402aa12db6e6a36fb380e7b53323'
c = bytes.fromhex(c_hex)

# Calculate modulus n
n = p * q

# --- Divide by p to work modulo q (as v = p * x) ---
# Check if the division is exact (i.e., vka, vkakb, vkb are multiples of p)
if vka % p != 0 or vkakb % p != 0 or vkb % p != 0:
    print("Error: Intermediate values are not multiples of p.")
    exit()

# xka = (vka / p) = (x * k_A) mod q
xka = (vka // p)
# xkakb = (vkakb / p) = (x * k_A * k_B) mod q
xkakb = (vkakb // p)
# xkb = (vkb / p) = (x * k_B) mod q
xkb = (vkb // p)

# --- Recover private key k_A modulo q ---
# k_A = xkakb * inverse(xkb, q) mod q
# xkakb / xkb = (x * k_A * k_B) / (x * k_B) = k_A mod q
k_A = (xkakb * inverse(xkb, q)) % q

# --- Recover secret value x modulo q ---
# x = xka * inverse(k_A, q) mod q
# xka / k_A = (x * k_A) / k_A = x mod q
x = (xka * inverse(k_A, q)) % q

# --- Calculate original secret v mod n ---
# v = p * x mod n
v = (p * x) % n

# --- Derive the AES key and decrypt ---
# The key is sha256(v)
key = sha256(long_to_bytes(v)).digest()

# The expected key from out.txt for verification
# expected_key_hex = '6bd39a9a6846f1acf573e5830dcd9eac29722311a0ef06392c8f3c4779bc664d'
# print(f"Recovered Key: {key.hex()}")
# print(f"Expected Key:  {expected_key_hex}")

# Decrypt the ciphertext c with AES-ECB
```

```
cipher = AES.new(key, AES.MODE_ECB)
decrypted_padded = cipher.decrypt(c)
flag = unpad(decrypted_padded, 16)

print(f"\nRecovered FLAG: {flag.decode()}")
```

— **Flag:** `crypto{1nv3rt1bl3_k3y_3xch4ng3_pr0t0c0l}`

— **Explanation:**

I. Exploiting the Structure of v

Since v is a multiple of p, we can divide all intermediate values by p and move the calculations to the modular field $\mathbb{Z}_q$, simplifying the relations:

$$\begin{cases} xka & = vka//p \equiv (x \cdot k_A) \pmod{q} \\ xkakb & = vkakb//p \equiv (x \cdot k_A \cdot k_B) \pmod{q} \\ xkb & = vkb//p \equiv (x \cdot k_B) \pmod{q} \end{cases}$$

II. Recovering the Secret Key k_A

Alice's secret key k_A, coprime to n, can be recovered by dividing $xkakb$ by xkb in the modular sense:

$$k_A \equiv xkakb \cdot xkb^{-1} \pmod{q}$$

Where xkb^{-1} is the modular inverse of xkb modulo q.

III. Recovering the Secret Factor x

After recovering k_A, the value x (a component of v) is calculated from the relation xka:

$$x \equiv xka \cdot k_A^{-1} \pmod{q}$$

IV. Final Calculation of v and Decryption

1. **Recovering** v: The full value of v is calculated:

$$v = (p \cdot x) \pmod{n}$$

2. **Deriving the AES Key**: The symmetric key key is the SHA-256 hash of v converted to bytes:

$$\text{key} = \text{SHA-256}(\text{long\_to\_bytes}(v))$$

3. **Decryption**: The ciphertext c is decrypted using AES-ECB with the key key and the padding is removed.

As a result of the calculations, the decrypted flag is: $crypto\{1nv3rt1bl3\_k3y\_3xch4ng3\_pr0t0c0l\}$

3. Brainteasers Part 2

3.1. Ellipse Curve Cryptography

— **Code:**

— **Flag:**

— **Explanation:**

3.2. Roll your Own

— **Code:**

```
from pwn import *
import json

def solve():
    # Connect to the challenge server
    conn = remote('socket.cryptohack.org', 13403)

    # Receive the prime q
    conn.recvuntil(b'Prime generated: ')
    q_hex = conn.recvline().strip().decode().strip('"')
    q = int(q_hex, 16)
    log.info(f"Received q: {q}")

    # Choose n and g to make the DLP easy
    # We choose n = q^2 and g = q + 1.
    # This satisfies the server's condition pow(g, q, n) == 1 because:
    # (q+1)^q mod q^2 = 1 + q*q + ... = 1 (mod q^2) by binomial expansion.
    n = q * q
```

```
    g = q + 1

    log.info(f"Using n = q^2: {n}")
    log.info(f"Using g = q + 1: {g}")

    # Send g and n to the server
    conn.recvuntil(b'Send integers (g,n) such that pow(g,q,n) = 1: ')
    payload = json.dumps({"g": hex(g), "n": hex(n)})
    conn.sendline(payload.encode())
    log.info("Sent g and n")

    # Receive the public key h
    try:
        conn.recvuntil(b'Generated my public key: ')
        h_hex = conn.recvline().strip().decode().strip('"')
        h = int(h_hex, 16)
        log.info(f"Received h: {h}")
    except EOFError:
        log.error("Server closed connection. Parameters were likely rejected.")
        conn.close()
        return

    # Solve for x
    # The server computed h = pow(g, x, n) = pow(q+1, x, q^2).
    # By binomial expansion, (q+1)^x = 1 + x*q (mod q^2).
    # So, h = 1 + x*q (mod q^2).
    # This means h - 1 = x*q.
    # We can find x by x = (h - 1) / q.
    x = (h - 1) // q
    log.info(f"Found x: {x}")

    # Send x to the server
    conn.recvuntil(b'What is my private key: ')
    payload = json.dumps({"x": hex(x)})
    conn.sendline(payload.encode())

    # Receive the flag
    response = conn.recvall()
    log.success(f"Received response: {response.decode()}")
    conn.close()

if __name__ == "__main__":
    solve()
```

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

— **Explanation:** (Explanation is already included in the document. This is an attack on the discrete logarithm problem through a **malicious choice of parameters**.

1. **Parameter Choice**: We choose $n = q^2$ and $g = q + 1$.
2. **Binomial Expansion (Newton's)**: This technique is used twice.
   — To prove to the server that $g^q \equiv 1 \pmod{n}$:

$$(q+1)^q \equiv \binom{q}{0} q^0 + \binom{q}{1} q^1 + \ldots \equiv 1 + q \cdot q + \ldots \equiv 1 \pmod{q^2}$$

   — To recover x from $h \equiv g^x \pmod{n}$:

$$h \equiv (q+1)^x \pmod{q^2} \equiv \binom{x}{0} q^0 + \binom{x}{1} q^1 + \ldots \equiv 1 + x \cdot q \pmod{q^2}$$

3. **Linear Algebra**: Instead of a difficult discrete logarithm problem, we reduce the problem to a simple linear equation $h \equiv 1 + xq \pmod{q^2}$.
4. **Recovering** x: This equation is trivial to solve. It means $h - 1 = xq + k \cdot q^2$ for some k. Dividing by q, we get $\frac{h-1}{q} = x + kq$. Assuming $x < q$ (which is standard for private keys), k must be 0. Thus, x can be calculated by simple integer division: $x = (h-1)//q$.

)

3.3. Unencryptable

— **Code:**

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

# Values from output.txt
N = 0x7fe8cafec59886e9318830f33747cafd200588406e7c42741859e15994ab62410438991ab5d9fc94f386219e3c27d6ffc73754f791e7b2c565611f8fe5054dd132b8c4f3eadcf1180cd8f2a3cc756b06996f2d5b67c390adcba9d444697b13d1
e = 0x10001
c = 0x5233da71cc1dc1c5f21039f51eb51c80657e1af217d563aa25a8104a4e84a42379040ecdfdd5afa191156ccb40b6f188f4ad96c58922428c4c0bc17fd5384456853e139afde40c3f95988879629297f48d0efa6b335716a4c24bfee36f714d34

# DATA from source.py
DATA_hex = "372f0e88f6f7189da7c06ed49e87e0664b988ecbee583586dfd1c6af99bf20345ae7442012c6807b3493d8936f5b48e553f614754deb3da6230fa1e16a8d5953a94c886699fc2bf409556264d5dced76a1780a90fd22f3701fdbcb183d
m_data = int(DATA_hex, 16)

# The fixed point property m^e = m (mod N) implies that m^(e-1) = 1 (mod p) and m^(e-1) = 1 (mod q).
# Since e-1 = 65536 = 2^16, this is a perfect setup for a Pollard's p-1 like attack.
# We can try to find a factor by computing gcd(m^(2^k) - 1, N) for k from 1 to 16.
# We are looking for a k where m^(2^k) is not 1 (mod N), but m^(2^k) - 1 shares a factor with N.

a = m_data
for k in range(1, 17): # e-1 = 2^16, so we check up to k=16
    a = pow(a, 2, N)
    p = math.gcd(a - 1, N)
    if 1 < p < N:
```

```
            print(f"Factor found at k={k}")
            break
    else: # This else belongs to the for loop, it runs if the loop completes without break
        p = 1 # Reset p if no factor was found

    if p == 1 or p == N:
        print("Failed to factor N with this method.")
    else:
        # Once we have p, we can find q
        q = N // p

        # Now we can calculate the private key d
        phi = (p - 1) * (q - 1)
        d = inverse(e, phi)

        # Decrypt the flag ciphertext
        m_flag = pow(c, d, N)
        flag = long_to_bytes(m_flag)

        print(f"p = {hex(p)}")
        print(f"q = {hex(q)}")
        print(f"flag: {flag.decode()}")
```

— **Flag:** `crypto{R3m3mb3r!_F1x3d_P0iNts_aR3_s3crE7s_t00}`

— **Explanation:** I. The Key Vulnerability We are given m_{data}, for which:

$$m_{\text{data}}^{e} \equiv m_{\text{data}} \pmod{N}$$

Rearranging (and assuming $\gcd(m_{\text{data}}, N) = 1$), we get:

$$m_{\text{data}}^{e-1} \equiv 1 \pmod{N}$$

We know that $e = 0x10001 = 65537$, so $e - 1 = 65536 = 2^{16}$. Therefore:

$$m_{\text{data}}^{2^{16}} \equiv 1 \pmod{N}$$

This means that $m_{\text{data}}^{2^{16}} \equiv 1 \pmod{p}$ and $m_{\text{data}}^{2^{16}} \equiv 1 \pmod{q}$.
II. Attack Method (Variant of Pollard's p-1) The attack consists of iteratively checking powers of m_{data}. We define $a_k = m_{\text{data}}^{2^k} \pmod{N}$.
The `for` loop in the code successively calculates $a_1, a_2, \ldots, a_{16}$. In each iteration k:
1. It calculates $a_k = (a_{k-1})^2 \pmod{N}$. (In code: `a = pow(a, 2, N)`)
2. It calculates $p_{\text{test}} = \gcd(a_k - 1, N)$.

The attack succeeds if, for some k, we find a p_{test} such that $1 < p_{\text{test}} < N$. This happens when $a_k \equiv 1 \pmod{p}$, but $a_k \not\equiv 1 \pmod{q}$ (or vice versa). In that case, $a_k - 1$ is a multiple of p, but not of q, so $\gcd(a_k - 1, p \cdot q) = p$.
III. Decrypting the Flag After finding the factor p:
1. The second factor is calculated: $q = N/p$.
2. Euler's totient function is calculated: $\phi(N) = (p-1)(q-1)$.
3. The private key d is calculated as the modular inverse of e:

$$d \equiv e^{-1} \pmod{\phi(N)}$$

   (In code: `d = inverse(e, phi)`)
4. The flag is decrypted by calculating:

$$m_{\text{flag}} = c^{d} \pmod{N}$$

   (In code: `m_flag = pow(c, d, N)`)
5. Finally, the numerical representation of the flag m_{flag} is converted back to a byte string (`long_to_bytes`).

3.4. Cofactor Cofantasy

— **Code:**

— **Flag:**
— **Explanation:**

3.5. Real Eisenstein

— **Code:**

— **Flag:**
— **Explanation:**

4. Primes

4.1. Prime and Prejudice

— Code:

```python
from Crypto.Util.number import *
import itertools
from tqdm import tqdm
import json
from pwn import remote

HOST = "socket.cryptohack.org"
PORT = 13385

def generate_prime_basis(n):
    """Generate all primes up to n using Sieve of Eratosthenes."""
    sieve = [True] * n
    for i in range(3, int(n**0.5) + 1, 2):
        if sieve[i]:
            sieve[i*i::2*i] = [False] * ((n - i*i - 1) // (2*i) + 1)
    return [2] + [i for i in range(3, n, 2) if sieve[i]]


def miller_rabin_test(n, max_base):
    """
    Miller-Rabin primality test using all prime bases < max_base.
    Returns True if n is probably prime, False if definitely composite.
    """
    prime_bases = generate_prime_basis(max_base)

    if n == 2 or n == 3:
        return True
    if n % 2 == 0:
        return False

    # Write n-1 as 2^r * s
    r, s = 0, n - 1
    while s % 2 == 0:
        r += 1
        s //= 2

    # Test with each prime base
    for base in prime_bases:
        if base >= n: continue
        x = pow(base, s, n)
        if x == 1 or x == n - 1:
            continue

        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True


def extended_gcd(a, b):
    """Extended Euclidean algorithm. Returns (gcd, s, t) where gcd = a*s + b*t."""
    s, s1 = 0, 1
    t, t1 = 1, 0
    r, r1 = b, a

    while r != 0:
        q = r1 // r
        r1, r = r, r1 - q * r
        s1, s = s, s1 - q * s
        t1, t = t, t1 - q * t

    return (r1, s1, t1)


def chinese_remainder_theorem(residues, modulos):
    """
    Solve system of congruences: x ≡ residues[i] (mod modulos[i]).
    Returns (solution, combined_modulo) or (-1, -1) if no solution exists.
    """
    result_mod = modulos[0]
    result_res = residues[0]

    for residue, modulo in zip(residues[1:], modulos[1:]):
        g = GCD(result_mod, modulo)

        # Check if solution exists
        if residue % g != result_res % g:
            return -1, -1

        _, s, t = extended_gcd(modulo // g, result_mod // g)
        result_res = result_res * (modulo // g) * s + residue * (result_mod // g) * t
        result_mod *= modulo // g
        result_res %= result_mod

    return result_res, result_mod


def legendre_symbol(a, p):
    """Compute Legendre symbol (a/p) = a^((p-1)/2) mod p."""
    if p < 2: return 0
    return pow(a, (p - 1) // 2, p)


def find_valid_residues(primes, num_factors):
    """
    Find residues that satisfy the Legendre symbol condition for fooling Miller-Rabin.
    """
    valid_residues = []
    ks = [1, 998244353, 233]  # Multipliers for constructing the pseudoprime

    for prime in primes:
        residue_set = set()
        larger_primes = generate_prime_basis(200 * prime)[1:]
```

```
        for q in larger_primes:
            if legendre_symbol(prime, q) == q - 1:
                residue_set.add(q % (4 * prime))

        valid_residues.append(list(residue_set))

    # Filter residues to work with all factors
    filtered_residues = []
    for idx, residue_list in enumerate(valid_residues):
        prime = primes[idx]
        modulus = prime * 4
        current_set = set(residue_list)

        for i in range(1, num_factors):
            new_set = set()
            for res in residue_list:
                try:
                    transformed = ((res + ks[i] - 1) * inverse(ks[i], modulus)) % modulus
                    if transformed % 4 == 3:
                        new_set.add(transformed)
                except ValueError:
                    pass # No inverse
            current_set = current_set.intersection(new_set)

        filtered_residues.append(current_set)

    return filtered_residues, ks


def generate_strong_pseudoprime(primes, filtered_residues, ks, num_factors,
                                min_bits=600, max_bits=900):
    """
    Generate a strong pseudoprime that passes Miller-Rabin test.
    Returns (pseudoprime, factors) or (None, None) if not found.
    """
    total_combinations = 1
    for res_set in filtered_residues:
        total_combinations *= len(res_set)
    print(f"Total combinations to try: {total_combinations}")

    for residue_tuple in itertools.product(*filtered_residues):
        residues = []
        modulos = []

        # Build CRT system from residue tuple
        for i, res in enumerate(residue_tuple):
            residues.append(res)
            modulos.append(primes[i] * 4)

        # Add constraints from ks multipliers
        try:
            residues.append(ks[1] - inverse(ks[2], ks[1]))
            modulos.append(ks[1])
            residues.append(ks[2] - inverse(ks[1], ks[2]))
            modulos.append(ks[2])
        except ValueError:
            continue # No inverse

        solution, combined_mod = chinese_remainder_theorem(residues, modulos)

        if solution == -1:
            continue

        # Start search from a large value
        candidate = 2**73 * combined_mod + solution

        for _ in tqdm(range(100000)):
            if isPrime(candidate):
                # Build the pseudoprime from candidate and its multiples
                pseudoprime = candidate
                factors = [candidate]

                for i in range(1, num_factors):
                    factor = ks[i] * (candidate - 1) + 1
                    factors.append(factor)
                    pseudoprime *= factor

                # Check if it passes Miller-Rabin and has correct bit length
                if miller_rabin_test(pseudoprime, 64):
                    bit_len = pseudoprime.bit_length()
                    if min_bits <= bit_len <= max_bits:
                        print(f"Found pseudoprime! isPrime check: {isPrime(pseudoprime)}")
                        print(f"Pseudoprime: {pseudoprime}")
                        print(f"Factors: {factors}")
                        return pseudoprime, factors

            candidate += combined_mod

    return None, None


# Main execution: Generate strong pseudoprime
NUM_FACTORS = 3
prime_bases = generate_prime_basis(64)
print(f"Number of prime bases: {len(prime_bases)}")
print(f"Prime bases: {prime_bases}")

filtered_residues, ks_multipliers = find_valid_residues(prime_bases, NUM_FACTORS)
print(f"Filtered residue sets: {filtered_residues}")

pseudoprime, factors = generate_strong_pseudoprime(
    prime_bases, filtered_residues, ks_multipliers, NUM_FACTORS
)
found = (pseudoprime is not None)

def solve(p, facs):
    """
    This function will connect to the server and send the payload.
    Uses the generated strong pseudoprime and its factors.
    p = product of facs, where p is composite but passes Miller-Rabin

    The key: since p is composite, we can choose 'a' such that gcd(a, p) > 1
    or such that a^(p-1) is NOT 1 mod p.
    """

    # Strategy: Make 'a' share a factor with p
```

```
    # If a = k * facs[0] for some k, then gcd(a, p) = facs[0] > 1
    # The server checks a < p, so we need a to be smaller than p but share a factor

    # Try using one of the factors directly or a small multiple
    for i, fac in enumerate(facs):
        print(f"\nTrying with factor {i}: {fac}")

        # Try small multiples of this factor
        for k in range(2, 10):
            a = k * fac
            if a >= p:
                break

            print(f"Trying a = {k} * factor = {a}")

            # Check what we get locally
            result = pow(a, p-1, p)
            print(f"Local test: pow(a, p-1, p) = {result}")

            r = remote(HOST, PORT)
            r.recvuntil(b"primes!\n")

            payload = json.dumps({"base": a, "prime": p})
            r.sendline(payload.encode())

            response = r.recvline().decode()
            try:
                data = json.loads(response)
                if "Response" in data:
                    print(data["Response"])
                    if "crypto{" in data["Response"]:
                        print(f"\n*** FOUND FLAG! ***")
                        r.close()
                        return
                else:
                    print(data)
            except json.JSONDecodeError:
                print(response)

            r.close()

if __name__ == "__main__":
    if found:
        print(f"Generated pseudoprime with bit length: {pseudoprime.bit_length()}")
        print(f"Factors: {factors}")
        solve(pseudoprime, factors)
    else:
        # Fallback to precomputed value if generation fails
        print("Failed to generate suitable pseudoprime. Using precomputed value.")
        pre_p = 469798609534603893863473134676646808169185062633869151526950254060014316132338480373801203001371970898492049132172018803362846875416047020083984032145780517789408817757629235831339573
        pre_facs = [190805096574824360341517400306381535441501, 1079323138806282361628108420455314051012301, 2280263150530483863414002621370221375351301]
        solve(pre_p, pre_facs)
```

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

— **Explanation:** This problem exploits a weakness in the **Fermat primality test**, which is a simplified version of the **Miller-Rabin test**.

1. **Fermat vs Miller-Rabin**: The server we connect to (`solve`) only uses the Fermat test: it checks if $a^{p-1} \equiv 1 \pmod{p}$. The full Miller-Rabin test (implemented in `miller_rabin_test`) is stronger and checks additional conditions.
2. **Objective**: The goal is to find a number p that is **composite**, but passes the Miller-Rabin test for all small bases a (here: all primes up to 64). Such a number is called a **strong pseudoprime**. At the same time, p must be constructed so that the Fermat test $a^{p-1} \equiv 1 \pmod{p}$ fails for some base a that the server does not check.
3. **Construction (Mathematics)**: The code in `generate_strong_pseudoprime` constructs such a "fake" prime p as a product of several prime factors ($p = f_1 \cdot f_2 \cdot f_3$). Using advanced techniques (including the **Chinese Remainder Theorem** to combine conditions and the **Legendre Symbol** to set them), the code imposes conditions on the factors f_i such that p "fools" the Miller-Rabin test for all bases $a < 64$. This is a known construction (e.g., Arnault's).
4. **The Attack (Function `solve`)**: Once we have our composite number p and we know its factors (`facs`), we can easily fail the Fermat test.
   — Fermat's Little Theorem ($a^{p-1} \equiv 1 \pmod{p}$) only applies when p is prime, or (for composite numbers) when $\text{GCD}(a, p) = 1$.
   — The code sends the server a base a that is *not* coprime to p. It chooses a as a multiple of one of the known factors of p, e.g., $a = 2 \cdot f_1$ (in the code `a = k * fac`).
   — For such an a, $\text{GCD}(a, p) = \text{GCD}(2f_1, f_1 f_2 f_3) \geq f_1 > 1$.
   — Because $\text{GCD}(a, p) > 1$, the server, when calculating `pow(a, p-1, p)`, will almost certainly *not* get 1. This reveals that p is composite, which the server rewards with a flag.