

MATHEMATICS

MODULAR MATH:

1. Quadratic Residues

W tym zadaniu kod został zorganizowany w funkcję `find_quadratic_residue`, która ma na celu znalezienie reszty kwadratowej z podanej listy kandydatów.

Implementuje ona metodę brute-force. Funkcja iteruje przez każdą możliwą wartość `potential_root` w zakresie od 1 do `modulus - 1`. Dla każdej z tych wartości, kwadrat jest obliczany przy użyciu wbudowanej, zoptymalizowanej funkcji `pow(potential_root, 2, modulus)`.

Następnie skrypt sprawdza, czy obliczony kwadrat znajduje się na liście `candidates`. Jeśli tak, oznacza to znalezienie pasującej reszty. Funkcja identyfikuje wtedy oba pierwiastki kwadratowe: `root1 = potential_root` oraz `root2 = (-root1) % modulus`. Flagą jest mniejsza z tych dwóch wartości.

Wynik:

```
Znaleziona reszta kwadratowa: 6
Pierwiastki kwadratowe to: 8 i 21
Mniejszy pierwiastek (flaga) to: 8
```

2. Legendre Symbol

To ćwiczenie polega na znalezieniu reszty kwadratowej dla dużej liczby pierwszej p , która spełnia warunek $p \equiv 3 \pmod{4}$. Kod składa się z dwóch głównych funkcji: `parse_input_file` oraz `solve_qr_special_case`.

Funkcja `parse_input_file` odpowiedzialna jest za wczytanie danych z pliku. Odczytuje ona plik linia po linii, a następnie w pętli `for` wyszukuje te zaczynające się od '`p =`' oraz '`ints =`'. Znalezione wartości są odpowiednio parsowane.

Funkcja `solve_qr_special_case` implementuje logikę zadania. Najpierw wykorzystuje Kryterium Eulera do szybkiego znalezienia reszty kwadratowej. Oblicza `euler_exponent = (prime - 1) // 2` i sprawdza, dla którego kandydata num wynik potęgowania `pow(num, euler_exponent, prime)` jest równy 1. Po znalezieniu reszty `found_qr`, kod wykorzystuje specjalny wzór dla $p \equiv 3 \pmod{4}$ do obliczenia pierwiastków.

Oblicza wykładnik $\text{root_exponent} = (\text{prime} + 1) // 4$,
a następnie pierwiastek $r_1 \equiv (\text{found_qr})^{\text{root_exponent}} \pmod p$.

Wynik:

```
Flaga (większy pierwiastek): 9329179912536670680654563847579743051210497606610361026993802570995224702006109080487018619528599  
87276802009798538471858912676574255085595480529025359214420955212306216145858457506093948136821068862986203695885760470746837  
2384278049741369153506182660264876115428251983455344219194133033177700490981696141526
```

3. Modular Square Root

To zadanie wymagało obliczenia pierwiastka kwadratowego modulo p dla dowolnej liczby pierwszej, włączając w to przypadki, gdy $p \equiv 1 \pmod 4$.

Kod rozpoczyna się od funkcji `main_sqrt_task`, która parsuje plik `output.txt` w celu znalezienia `a_val` i `p_val`. Proces ten odbywa się przy użyciu podstawowych operacji na ciągach znaków, takich jak `split()` i `startswith()`.

Główna logika zawarta jest w funkcji `find_modular_sqrt_general`, która działa jak przełącznik:

- Jeśli $p \equiv 3 \pmod 4$, stosuje prosty wzór $r \equiv a^{((p+1)/4)} \pmod p$.
- Jeśli $p \equiv 1 \pmod 4$, wywołuje funkcję `compute_tonelli_shanks`.

Funkcja `compute_tonelli_shanks` implementuje pełny algorytm Tonelliego-Shanksa. Znajduje ona rozkład $p-1 = Q \cdot 2^S$, następnie wyszukuje nieresztę kwadratową `non_residue` i w pętli `while t_val != 1` iteracyjnie oblicza poprawny pierwiastek.

Ostatecznie zwracany jest mniejszy z dwóch pierwiastków: r oraz $p - r$.

Wynik:

```
Mniejszy z dwóch pierwiastków to: 23623393076830486383277732985804892989321375055205003883382710520537347478623517796473141768  
17953359071871560041125289919247146074907151612762640868199621186559522068338032600913118822240160212226722431393621804612326  
467324658488404254582579308878565833796009677617385967828778513184893556798228131551230457052851120994481464267551101600025155  
924188504321036418158110715484562842635078055894450736575653818505213679696756997607553107846235770764400377476817603024349249  
321136400617387776011946222441927580241808539162442725406544196255728572849162772740798986479486452073497374574454404050571  
56897508368531939120
```

4. Chinese Remainder Theorem

Ten kod rozwiązuje układ kongruencji przy użyciu Chińskiego Twierdzenia o Resztach (CRT). Funkcja `solve_crt_explicit` implementuje wzór CRT w sposób jawnym, krok po kroku.

Proces przebiega następująco:

1. Obliczany jest total_product (wielki moduł N), poprzez iterację po wszystkich modułach z listy moduli.
2. Następnie, używając funkcji zip, kod iteruje jednocześnie po listach remainders (a_i) i moduli (n_i).
3. Dla każdej pary (a_i, n_i) w pętli, obliczane są:
 - $N_i = \text{total_product} // n_i$
 - Odwrotność modularna $M_i = N_i^{-1} \pmod{n_i}$, przy użyciu $\text{pow}(N_i, -1, n_i)$
4. Każdy obliczony składnik $a_i \cdot N_i \cdot M_i$ jest dodawany do łącznej sumy solution.

Ostatecznym wynikiem jest $\text{solution \% total_product}$.

Wynik:

```
Weryfikacja:  
872 % 5 = 2 (oczekiwano 2)  
872 % 11 = 3 (oczekiwano 3)  
872 % 17 = 5 (oczekiwano 5)  
  
Flaga: 872
```

BRAINTEASERS P.1

1. Succesive Powers

Otrzymuję ciąg S kolejnych potęg liczby z modulo p . Daje to relację:

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

Znajdowanie z : Biorąc szósty i siódmy wyraz ciągu, $s_6 = 4$ i $s_7 = 836$, otrzymuję:

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

Zakładając najprostszy przypadek (że p jest znacznie większe i $836 = 4z$), obliczam:

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

Znajdowanie p : Mając z , mogę przekształcić relację do:

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

Oznacza to, że p musi być wspólnym dzielnikiem wszystkich takich wyrażeń. Obliczam dwa takie wyrażenia:

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

$$X_2 = s_4 \cdot z - s_5 = 113 \cdot 209 - 642 = 22975$$

Obliczenie $p = \text{NWD}(122227, 22975)$ daje wynik 919.

Zadanie w pliku jest wykonane metodą brute-force.

2. Adrien's Signs:

Celem było odzyskanie flagi z output.txt. Analiza kodu source.py pokazuje, że bity flagi są szyfrowane przy użyciu liczby pierwszej
 $p = 1007621497415251$ (dla której $p \equiv 3(\text{mod } 4)$) oraz bazy
 $a = 288260533169915$, która jest resztą kwadratową modulo p .

Obliczana jest wartość $n \equiv a^e(\text{mod } p)$. Wartość n jest zawsze resztą kwadratową, ponieważ:

$$\left(\frac{n}{p}\right) = \left(\frac{a^e}{p}\right) = \left(\frac{a}{p}\right)^e = 1^e = 1$$

Jeśli bit to '**1**', do szyfrogramu dodawane jest n (reszta kwadratowa)

Jeśli bit to '**0**', dodawane jest $-n(\text{mod } p)$ (niereszta kwadratowa, ponieważ):

$$\left(\frac{-n}{p}\right) = \left(\frac{-1}{p}\right) \cdot \left(\frac{n}{p}\right) = (-1) \cdot (1) = -1$$

Skrypt deszyfrujący odwraca ten proces. Dla każdego elementu c_val z szyfrogramu oblicza symbol Legendre'a $\text{pow}(c_val, (p-1)//2, p)$:

Jeśli wynik to **1**, bit to '**1**'

Jeśli wynik to **p-1** (czyli -1), bit to '**0**'

Połączenie bitów i konwersja na tekst daje flagę.

Flaga: crypto{p4tterns_in_re5idu3s}

3. Modular Binomials

Podatność wynika z zastosowania rozwinięcia dwumianowego (Newtona) modulo $N = pq$. Dla $c \equiv (ap + bq)^e(\text{mod } N)$, wszystkie środkowe wyrazy rozwinięcia są podzielne przez $pq = N$, więc znikają. Zostaje tylko:

$$c \equiv (ap)^e + (bq)^e \pmod{N}$$

Otrzymuję dwa równania (używając danych z PDF, gdzie $A = 2, B = 3, C = 5, D = 7, e_1 = 3, e_2 = 3$):

$$c_1 \equiv (2p)^3 + (3q)^3 \pmod{N}$$

$$c_2 \equiv (5p)^3 + (7q)^3 \pmod{N}$$

Aby wyeliminować p , sprowadzam je do wspólnego wykładnika $e = e_1 \cdot e_2 = 9$:

$$x_1 = C^e \cdot c_1^{e_2} \equiv 5^9 \cdot ((2p)^3 + (3q)^3)^3 \equiv 5^9(2p)^9 + 5^9(3q)^9 \pmod{N}$$

$$x_2 = A^e \cdot c_2^{e_1} \equiv 2^9 \cdot ((5p)^3 + (7q)^3)^3 \equiv 2^9(5p)^9 + 2^9(7q)^9 \pmod{N}$$

Gdy odejmę $t = x_1 - x_2$, składniki $(10p)^9$ kasują się. Zostaje wyrażenie t , które jest wielokrotnością q . Obliczenie NWD(t, N) pozwala odzyskać czynnik q . Drugi czynnik p obliczam jako $N//q$.

Flaga: crypto{11227400016925848639026206444199120060855637612740895
2701514962644340921899196091557519}

4. Broken RSA

Szyfrowanie to $ct \equiv m^{16} \pmod{p}$. Ponieważ wykładnik $e = 16$ nie jest względnie pierwszy z $\phi(p) = p - 1$ (oba są parzyste), nie można obliczyć standardowego klucza prywatnego d .

Atak polega na wykorzystaniu faktu, że $16 = 2^4$. Obliczenie pierwiastka 16-go stopnia jest równoważne czterokrotnemu obliczeniu pierwiastka kwadratowego:

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

Każda operacja modular_sqrt (pierwiastka kwadratowego) daje dwa rozwiązania: r oraz $p - r$ (czyli $\pm r$). Ponieważ operację wykonuję 4 razy, generuje to $2^4 = 16$ możliwych kandydatów na wiadomość m .

Skrypt BrokenRSA.py iteruje przez wszystkie 16 ścieżek. Na każdym z 4 kroków decyduje, czy wziąć "dodatni" pierwiastek f_{next} , czy "ujemny" $p - f_{\text{next}}$, na podstawie bitów licznika i . Każdy kandydat, który jest poprawny (daje się zdekodować na bajty ASCII), jest wypisywany.

Flaga: crypto{m0dul4r_squ4r3_r00t}

5. No Way Back Home

Analiza skryptu szyfrującego pokazuje, że tajny klucz v jest tworzony jako:

$$v \equiv p \cdot x \pmod{n}$$

Oznacza to, że v (oraz wszystkie opublikowane wartości vka, vkakb, vkb) jest wielokrotnością p .

Ta podatność pozwala na przeniesienie wszystkich obliczeń do ciała \mathbb{Z}_q (modulo q) poprzez podzielenie wszystkich znanych wartości przez p :

- $xka = vka // p \equiv x \cdot k_A \pmod{q}$
- $xkakb = vkakb // p \equiv x \cdot k_A \cdot k_B \pmod{q}$
- $xkb = vkb // p \equiv x \cdot k_B \pmod{q}$

Skrypt odwraca ten proces:

Odzyskuje k_A (modulo q) poprzez:

$$k_A = xkakb \cdot (xkb)^{-1} \pmod{q}$$

Odzyskuje x (modulo q) poprzez:

$$x = xka \cdot (k_A)^{-1} \pmod{q}$$

Rekonstruuje pełne v jako:

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

Używa v do wygenerowania klucza AES: `key = sha256(long_to_bytes(v))`

Deszyfruje szyfrogram c przy użyciu obliczonego klucza

Flaga: `crypto{1nv3rt1bl3_k3y_3xch4ng3_pr0t0c01}`

BRAINTEASERS P.2

1. Roll Your Own

— **Flaga:** crypto{Grabbing_Flags_with_Pascal_Paillier}

— **Wyjaśnienie:** Atak polega na obejściu problemu logarytmu dyskretnego (DLP) poprzez zmanipulowanie parametrów protokołu.

Manipulacja parametrami: Kluczem do ataku jest zignorowanie propozycji serwera i wysłanie własnych, specjalnie spreparowanych wartości. Otrzymuję od serwera liczbę pierwszą q i odsyłam:

$$n = q^2$$

$$g = q + 1$$

Serwer wymaga, aby spełniony był warunek $g^q \equiv 1 \pmod{n}$. Parametry spełniają ten warunek, ponieważ przy obliczaniu $(q+1)^q \pmod{q^2}$, wszystkie składniki rozwinięcia poza pierwszym (który wynosi 1) stają się wielokrotnościami q^2 , a więc redukują się do zera modulo q^2 .

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

Obliczenie klucza publicznego: Serwer oblicza $h \equiv g^x \pmod{n}$. Dla w/w parametrów to równanie również się upraszcza:

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

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

Redukcja: W ten sposób trudny problem logarytmu dyskretnego (znalezienie x z $h \equiv g^x$) zostaje zredukowany do prostego równania liniowego:

$$h \equiv 1 + xq \pmod{q^2}$$

Odzyskanie: Przekształcenie równania daje $h - 1 \equiv xq \pmod{q^2}$. Oznacza to, że $h - 1$ jest wielokrotnością xq . Zakładając, że tajny klucz x jest mniejszy od q (co jest standardową praktyką), mogę bezpiecznie odzyskać x przez proste dzielenie całkowitoliczbowe:

$$x = (h - 1) // q$$

Wynik:

```
[*] Found x (int): 1283757731295699921432686448559060293505307308875789649642404285745716643645391272663223
083645706275600438996709312464715393964466279492
[*] Receiving all data
[*] Receiving all data: 0B
[*] Receiving all data: 56B
[+] Receiving all data: Done (56B)
[*] Closed connection to socket.cryptothon.org port 13403
[+] Received response: {"flag": "crypto{Grabbing_Flags_with_Pascal_Paillier}"}
PS C:\Users\Piwlubie\OneDrive\Pulpit\CryptoHack_Challenges\MATHEMATICS\MODULAR MATH> 
```

2. Unencryptable

Dane m_{data} spełniają:

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

Zakładając $\gcd(m_{\text{data}}, N) = 1$, przekształcam:

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

Dla $e = 6553 \rightarrow e - 1 = 65536 = 2^{16}$. Zatem:

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

co implikuje, że ta relacja zachodzi osobno modulo p i modulo q :

$$m_{\text{data}}^{2^{16}} \equiv 1 \pmod{p}, m_{\text{data}}^{2^{16}} \equiv 1 \pmod{q}.$$

Metoda ataku (wariant Pollard-p-1)

a) iteracyjnie liczę potęgi

$$a_k = m_{\text{data}}^{2^k} \pmod{N}$$

dla $k = 1, 2, \dots, 16$. W każdej iteracji sprawdzam

$$\text{ptest} = \gcd(a_k - 1, N).$$

Jeśli dla pewnego k zachodzi $a_k \equiv 1 \pmod{p}$ ale $a_k \not\equiv 1 \pmod{q}$, to $a_k - 1$ jest wielokrotnością pale nie q . Wtedy:

$$1 < \gcd(a_k - 1, N) = p < N,$$

i otrzymuję dzielnik p liczby N .

Algorytm:

$$a_0 = m_{\text{data}} \bmod N.$$

Dla k od 1 do 16 wykonuję:

- $a_k = a_{k-1}^2 \bmod N$.
- $ptest = \gcd(a_k - 1, N)$.
- Jeśli $1 < ptest < N$ — mamy czynnik $p = ptest$ i przerwam.

Deszyfrowanie flagi po znalezieniu p

1. Wyznaczam drugi czynnik: $q = N/p$.
2. Obliczam $\varphi(N) = (p-1)(q-1)$.
3. Obliczam klucz prywatny d jako odwrotność modularną e modulo $\varphi(N)$:

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

4. Odszyfrowuję szyfrogram c :

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

5. Zamień liczbę m_{flag} na bajty i odczytuję flagę

Wynik:

```
0Z001a01
flag: crypto{R3m3mb3r!_F1x3d_P0iNts_aR3_s3crE7s_t00}
```

PRIMES

1. Prime and Prejustice

Zadanie to wykorzystuje ograniczenia testu pierwszości Fermat, który jest prostszą wersją bardziej zaawansowanego testu Millera-Rabina.

Różnica między nimi: Serwer (dostępny przez funkcję solve) posługuje się wyłącznie testem Fermata, sprawdzając warunek $a^{p-1} \equiv 1 \pmod{p}$. Pełny test Millera-Rabina (zaimplementowany jako miller_rabin_test) jest dokładniejszy i weryfikuje dodatkowe własności liczby.

Należy znaleźć liczbę p , która jest złożona, ale przechodzi test Millera-Rabina dla wszystkich małych podstaw a (w tym przypadku dla wszystkich liczb pierwszych mniejszych niż 64). Taka liczba nazywana jest silną liczbą pseudopierwszą. Jednocześnie liczba p musi być tak zbudowana, aby test Fermata zakończył się niepowodzeniem dla pewnej podstawy a , której serwer nie bada.

Kod w funkcji generate_strong_pseudoprime tworzy taką "fałszywą" liczbę pierwszą p jako iloczyn kilku mniejszych liczb pierwszych ($p = f_1 \cdot f_2 \cdot f_3$). Wykorzystując CRT oraz Legendre'a kod dobiera czynniki f_i w taki sposób, aby p udawała liczbę pierwszą w teście Millera-Rabina dla wszystkich baz $a < 64$. Jest to znana konstrukcja matematyczna (np. metoda Arnaulta).

Sposób ataku (funkcja solve): Gdy już dysponuję złożoną liczbą p i znam jej czynniki pierwsze (fac), mogę w prosty sposób spowodować, że nie przejdzie ona testu Fermata.

- Małe Twierdzenie Fermata ($a^{p-1} \equiv 1 \pmod{p}$) zachodzi tylko wtedy, gdy p jest liczbą pierwszą lub (dla liczb złożonych) gdy $\text{NWD}(a, p) = 1$.
- Kod przesyła do serwera podstawę a , która *nie* jest względnie pierwsza z p . Wybiera się a jako wielokrotność jednego ze znanych czynników p , na przykład $a = 2 \cdot f_1$ (w kodzie: $a = k \cdot \text{fac}$).
- Dla takiej wartości a , $\text{NWD}(a, p) = \text{NWD}(2f_1, f_1f_2f_3) \geq f_1 > 1$.
- Ponieważ $\text{NWD}(a, p) > 1$, serwer obliczając $\text{pow}(a, p-1, p)$ z dużym prawdopodobieństwem *nie* otrzyma wyniku 1. Ujawnia to, że p jest liczbą złożoną, za co serwer przyznaje flagę.

Wynik:

```
47586885383525697957272457838473
[*] Opening connection to socket.cryptocheck.org on port 13385
[*] Opening connection to socket.cryptocheck.org on port 13385: Trying 134.122.111.232
[+] Opening connection to socket.cryptocheck.org on port 13385: Done
Server response: [*] Success: You passed all my tests! Here's the first byte of my flag: crypto{Forging_Primes_with_Francois_Arnault}

*** FOUND FLAG! ***
[*] Closed connection to socket.cryptocheck.org port 13385
PS C:\Users\Piwlkolubie\OneDrive\Pulpit\CryptoHack_Challenges\MATHEMATICS\MODULAR MATH> █
```