

## GENERAL

### ENCODING:

#### 1. ASCII

*nums* – lista liczb, które odpowiadają kodom ASCII poszczególnych znaków.

`".join(chr(n) for n in nums)` – dla każdej liczby w liście funkcja `chr(n)` zamienia ją na odpowiadający znak ASCII, a `join` łączy te znaki w jeden ciąg tekstowy.

`print(flag)` – wyświetlenie otrzymanego ciągu znaków.

#### 2. HEX

*hex\_str* – ciąg znaków w formacie szesnastkowym, gdzie każda para cyfr odpowiada jednemu bajtowi tekstu.

`bytes.fromhex(hex_str)` – konwertuje zapis szesnastkowy na dane binarne (bajty).

`.decode()` – zamienia te bajty na zwykły ciąg znaków ASCII.

`print(flag)` – wyświetla uzyskany tekst.

#### 3. BASE64

*hex\_str* – ciąg w formacie szesnastkowym.

`bytes.fromhex(hex_str)` – zamienia zapis hex na bajty.

`base64.b64encode(data)` – koduje te bajty w formacie Base64, który służy do zapisu danych binarnych w postaci znaków ASCII.

`.decode()` – przekształca wynik Base64 z bajtów na zwykły tekst.

`print(b64.decode())` – wyświetla końcowy ciąg w Base64.

#### 4. INTEGER TO MESSAGE

*integer* – liczba całkowita reprezentująca zakodowaną wiadomość w postaci liczbowej.

*long\_to\_bytes(integer)* – funkcja z biblioteki `Crypto.Util.number` przekształca liczbę w odpowiadającą jej sekwencję bajtów.

*.decode('utf-8')* – zamienia bajty na ciąg znaków w formacie tekstowym (UTF-8).

*print(...)* – wyświetla odszyfrowaną wiadomość.

#### 5. ENCODING CHALLENGE

*W tym zadaniu należało napisać klienta sieciowego, który automatycznie łączy się z serwerem `socket.cryptohack.org` na porcie 13377, odbiera dane w formacie JSON i dekoduje je w zależności od typu. Każdy otrzymany obiekt zawierał pola `type` oraz `encoded`, gdzie `typ` określał sposób zakodowania wiadomości. Program rozpoznawał pięć formatów: `base64`, `hex`, `rot13`, `bigint` oraz `utf-8`, a następnie odpowiednio przetwarzał dane przy użyciu standardowych funkcji języka Python. Otrzymany wynik był wysyłany z powrotem do serwera w postaci obiektu `{"decoded": "<tekst>"}`. Proces ten powtarzał się aż do momentu otrzymania końcowej wiadomości zawierającej flagę. Skrypt wykorzystywał bibliotekę `pwn` do obsługi połączenia sieciowego oraz moduły `json`, `base64` i `codecs` do przetwarzania danych.*

## XOR

### 1. XOR Starter

*W tym ćwiczeniu zastosowano operację XOR do modyfikacji bajtów etykiety. Program importuje funkcję xor z biblioteki pwn. Zmienna label przechowuje ciąg bajtów b"label". Wywołanie xor(label, 13) wykonuje XOR każdego bajtu z wartością 13 i zwraca zmodyfikowane bajty. Następnie składana jest flaga przez dołączenie prefiksu b"crypto{", wyniku operacji XOR i sufiksu b"}". Na końcu wynikowe bajty są dekodowane do tekstu i wyświetlane.*

### 2. XOR Properties

*W zadaniu odzyskano flagę poprzez operacje XOR na trzech kluczach i zaszyfrowanym bajtowym ciągu. Przekształcono hexy na bajty (bytes.fromhex). Funkcja bxor(a,b) wykonuje XOR bajt po bajcie dla dwóch ciągów. Z rekordu  $KEY2 \oplus KEY1$  oraz znanego  $KEY1$  odzyskano  $KEY2$  przez ponowny XOR. Następnie z  $KEY2 \oplus KEY3$  i odzyskanego  $KEY2$  wyznaczono  $KEY3$ . Mając  $K1$ ,  $K2$  i  $K3$  odjęte logicznie, odszyfrowano flagę przez XOR z  $FLAG\_xor\_ALL$ .*

### 3. Hidden XOR p.1

*Zadanie polegało na odszyfrowaniu tekstu zaszyfrowanego jednobajtowym kluczem XOR. Najpierw  $ct = bytes.fromhex(...)$  konwertuje heksadecymalny ciąg na bajty. Następnie pętla `for k in range(256)` próbuje każdego możliwego klucza 0–255. Dla każdego klucza tworzony jest kandydat  $pt = bytes(b \oplus k \text{ for } b \text{ in } ct)$  przez XOR każdego bajtu z  $k$ . Kod próbuje zdekodować  $pt$  do ASCII w try/except, aby pominąć nieprawidłowe sekwencje bajtów. Dodatkowa filtracja `all(32 <= ord(c) <= 126 for c in s)` zapewnia, że wypisywane są tylko w pełni drukowalne ciągi.*

### 4. Hidden XOR p.2

*W tym ćwiczeniu odszyfrowano szyfrogram XOR z powtarzalnym kluczem, wykorzystując znany prefiks crypto{ do odzyskania części klucza. Program konwertuje heksadecymalny szyfrogram na bajty przy pomocy binascii.unhexlify, a znany tekst na heksadecymalny zapis, po czym bierze odpowiednie fragmenty i wykonuje XOR bajtów, by uzyskać część klucza (xor\_hex). Następnie odzyskano pełny klucz myXORkey. Zaczęto powtarzać go cyklicznie do długości szyfrogramu i dla każdego bajtu szyfrogramu wykonuje XOR z odpowiadającym bajtem klucza. Otrzymane bajty są składane do plaintext\_bytes i próbowane do dekodowania jako ASCII. Jeśli dekodowanie się powiedzie, wynikiem jest flaga w czytelnej postaci.*

## 5. LEMUR XOR

Skrypt wykonuje odszyfrowanie obrazu przez operację XOR między dwoma plikami obrazów i zapisuje wynik jako nowy plik. Program używa PIL (Pillow) do wczytania i konwersji obrazów do trybu RGB oraz numpy do reprezentacji pikseli jako tablic bajtów. Najpierw sprawdza, czy obrazy mają takie same wymiary; w przeciwnym razie zgłasza błąd. Następnie tworzy tablice pikseli `arr1` i `arr2` i oblicza `xor_arr = np.bitwise_xor(arr1, arr2)`, co wykonuje XOR każdego kanału RGB par pikseli. Wynik konwertowany jest z powrotem na obraz `uint8` i zapisywany do pliku `xored.png`. Wejście: dwa obrazy o tych samych rozmiarach (`flag.png`, `lemur.png`). Wyjście: odsłonięty obraz zapisany w `xored.png`. Ograniczenia: obrazy muszą mieć tę samą rozdzielczość i format RGB; operacja zakłada, że ukryta treść powstaje przez XORowanie bajtów.



## MATHEMATICS

### 1. GCD

Zwraca największy wspólny dzielnik dwóch liczb. Funkcja używa algorytmu Euklidesa: w pętli zamienia  $(a, b)$  na  $(b, a \% b)$  aż  $b$  będzie 0, wtedy  $a$  to GCD.

### 2. Extended GCD

Ten kod realizuje rozszerzony algorytm Euklidesa, który oprócz największego wspólnego dzielnika ( $\text{gcd}$ ) zwraca liczby całkowite  $u$  i  $v$  spełniające równanie:

$$u \cdot p + v \cdot q = \text{gcd}(p, q)$$

Dla danych  $p = 26513$  i  $q = 32321$  funkcja `extended_gcd` działa rekurencyjnie:

Jeśli  $b == 0$ , zwraca  $(a, 1, 0)$  jako bazę rekurencji.

W przeciwnym razie wywołuje `extended_gcd(b, a \% b)`, a następnie oblicza współczynniki  $x = y1$  i  $y = x1 - (a // b) * y1$ .

### 3. Modular Arithmetic 1

Tutaj bardzo proste zadanie.

### 4. Modular Arithmetic 2

Też napisane wprost.

### 5. Modular Inverting

Z małego twierdzenia Fermata mamy:

$$a^{p-1} \equiv 1 \pmod{p}$$

Można zapisać jako:

$$a \cdot a^{p-2} \equiv 1 \pmod{p}$$

Co daje:

$$a^{-1} \equiv a^{p-2} \pmod{p}$$