Course Information

- **Course Code:** CDS113
- **Course Name:** Applied Cryptography

Student's Information

- **Student's Full Name:** Vasileios Andrianopoulos
- **Student Code:** ΜΠΚΕΔ2303
- **Cryptohack Username:** billandriano
- **Cryptohack Score:** 1125

Challenge Page : GENERAL

ENCODING

1. ASCII

ASCII is a 7-bit encoding standard which allows the representation of text using the integers `0`-127.

Using the below integer array, convert the numbers to their corresponding ASCII characters to obtain a flag.

`[99, 114, 121, 112, 116, 111, 123, 65, 83, 67, 73, 73, 95, 112, 114, 49, 110, 116, 52, 98, 108, 51, 125]`

> In Python, the `chr()` function can be used to convert an ASCII ordinal number to a character (the `ord()` function does the opposite).

You have solved this challenge!

Σχόλια :

Θα χρησιμοποιήσουμε τη συνάρτηση chr() που μετατρέπει έναν αριθμό Unicode σε χαρακτήρα ascii.

In [30]:

```python
ascii_list_numbers=[99, 114, 121, 112, 116, 111, 123, 65, 83, 67, 73, 73, 95, 112, 114, 49, 110, 116, 52, 98, 1(

ascii_string=""

for i in ascii_list_numbers:
    ascii_string+=chr(i)

print(ascii_string)
```

```
crypto{ASCII_pr1nt4bl3}
```

2. Hex

Another common encoding scheme is Base64, which allows us to represent binary data as an ASCII string using an alphabet of 64 characters. One character of a Base64 string encodes 6 binary digits (bits), and so 4 characters of Base64 encode three 8-bit bytes.

Base64 is most commonly used online, so binary data such as images can be easily included into HTML or CSS files.

Take the below hex string, *decode* it into bytes and then *encode* it into Base64.

`72bca9b68fc16ac7beeb8f849dca1d8a783e8acf9679bf9269f7bf`

> In Python, after importing the base64 module with `import base64`, you can use the `base64.b64encode()` function. Remember to decode the hex first as the challenge description states.

Σχόλιο:

Ακολουθώντας το hint της εκφώνησης, θα χρησιμοποιήσουμε τη συνάρτηση bytes.fromhex(). Η συνάρτηση αυτή παίρνει όρισμα έναν δεκαεξαδικό αριθμό και δημιουργεί ένα αντικείμενο bytes. Στη προκειμένη περίπτωση αυτό το αντικείμενο αντιστοιχεί σε μια σειρά γραμμάτων στο μοντέλο ascii.

In [3]:

```python
hex = '63727970746f7b596f755f77696c6c5f62655f776f726b696e675f776974685f6865785f737472696e67735f615f6c6f747d'
bytes.fromhex(hex)
```

Out[3]: b'crypto{You_will_be_working_with_hex_strings_a_lot}'

3. Base64

Another common encoding scheme is Base64, which allows us to represent binary data as an ASCII string using an alphabet of 64 characters. One character of a Base64 string encodes 6 binary digits (bits), and so 4 characters of Base64 encode three 8-bit bytes.

Base64 is most commonly used online, so binary data such as images can be easily included into HTML or CSS files.

Take the below hex string, *decode* it into bytes and then *encode* it into Base64.

`72bca9b68fc16ac7beeb8f849dca1d8a783e8acf9679bf9269f7bf`

> In Python, after importing the base64 module with `import base64`, you can use the `base64.b64encode()` function. Remember to decode the hex first as the challenge description states.

Σχόλιο :

Ακολουθώντας το hind της εκφώνησης, θέλουμε να μετατρέψουμε ένα bytes αντικείμενο σε μορφή base64. Ακολουθώντας την προηγούμενη λογική, πρώτα μετατρέπουμε το δεκαεξαδικό αριθμό σε Bytes και μετά τον μετατρέπουμε σε μορφή base64 χρησιμοποιώντας τη συνάρτηση base64.b64encode().

In [4]:
```
import base64

hex_string="72bca9b68fc16ac7beeb8f849dca1d8a783e8acf9679bf9269f7bf"

#We decode first to bytes with bytes.fromhex and then we encode with base64.b64encode

print(base64.b64encode(bytes.fromhex(hex_string)))
```

b'crypto/Base+64+Encoding+is+Web+Safe/'

4. Bytes and Big Integers

Cryptosystems like RSA works on numbers, but messages are made up of characters. How should we convert our messages into numbers so that mathematical operations can be applied?

The most common way is to take the ordinal bytes of the message, convert them into hexadecimal, and concatenate. This can be interpreted as a base-16/hexadecimal number, and also represented in base-10/decimal.

To illustrate:

```
message: HELLO
ascii bytes: [72, 69, 76, 76, 79]
hex bytes: [0x48, 0x45, 0x4c, 0x4c, 0x4f]
base-16: 0x48454c4c4f
base-10: 310400273487
```

> Python's PyCryptodome library implements this with the methods `bytes_to_long()` and `long_to_bytes()`. You will first have to install PyCryptodome and import it with `from Crypto.Util.number import *`. For more details check the FAQ.

Convert the following integer back into a message:

`11515195063862318899931685488813747395775516287289682636499965282714637259206269`

Σχόλιο :

Στην παρούσα άσκηση, θα χρησιμοποιήσουμε τη συνάρτηση long_to_bytes() όπου δεν είναι ενσωματωμένη στη βιβλιοθήκη της Python. Για αυτό της εισάγουμε από τη βιβλιοθήκη Crypto.Util.number . Στη συνέχεια καλώντας τη συνάρτηση αυτή μετατρέπει έναν μεγάλο αριθμό σε αντικείμενο bytes.

In [5]:
```
from Crypto.Util.number import *

base_10_string=11515195063862318899931685488813747395775516287289682636499965282714637259206269

print(long_to_bytes(base_10_string))
```

b'crypto{3nc0d1n6_4ll_7h3_w4y_d0wn}'

XOR

1. XOR Starter

XOR is a bitwise operator which returns `0` if the bits are the same, and 1 otherwise. In textbooks the XOR operator is denoted by ⊕, but in most challenges and programming languages you will see the caret `^` used instead.

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

For longer binary numbers we XOR bit by bit: `0110 ^ 1010 = 1100`. We can XOR integers by first converting the integer from decimal to binary. We can XOR strings by first converting each character to the integer representing the Unicode character.

Given the string `label`, XOR each character with the integer `13`. Convert these integers back to a string and submit the flag as `crypto{new_string}`.

> 💡 The Python `pwntools` library has a convenient `xor()` function that can XOR together data of different types and lengths. But first, you may want to implement your own function to solve this.

Σχόλιο:

Στην Python, ο τελεστής XOR (^) δεν μπορεί να εφαρμοστεί άμεσα σε χαρακτήρες, αλλά μόνο σε αριθμούς. Για αυτό θα χρειαστεί να μετατρέψουμε τον κάθε χαρακτήρα του string 'label' σε αριθμό. Επομένως, θα χρησιμοποιήσουμε τη συνάρτηση ord(), στη συνέχεια θα κάνουμε xor με το 13 και τέλος το αποτέλεσμα θα το μετατρέψουμε σε ascii, με τη συνάρτηση chr(). Επαναλμβάνουμε για κάθε χαρακτήρα του string 'label'.

In [7]:
```python
import string


given_string="label"
anwser=""
for i in given_string:
    anwser+=chr(ord(i)^13)

print('crypto{'+anwser+'}')
```

```
crypto{aloha}
```

2. XOR Properties

In the last challenge, you saw how XOR worked at the level of bits. In this one, we're going to cover the properties of the XOR operation and then use them to undo a chain of operations that have encrypted a flag. Gaining an intuition for how this works will help greatly when you come to attacking real cryptosystems later, especially in the block ciphers category.

There are four main properties we should consider when we solve challenges using the XOR operator

```
Commutative: A ⊕ B = B ⊕ A
Associative: A ⊕ (B ⊕ C) = (A ⊕ B) ⊕ C
Identity: A ⊕ 0 = A
Self-Inverse: A ⊕ A = 0
```

Let's break this down. Commutative means that the order of the XOR operations is not important. Associative means that a chain of operations can be carried out without order (we do not need to worry about brackets). The identity is `0`, so XOR with `0` "does nothing", and lastly something XOR'd with itself returns zero.

Let's put this into practice! Below is a series of outputs where three random keys have been XOR'd together and with the flag. Use the above properties to undo the encryption in the final line to obtain the flag.

```
KEY1 = a6c8b6733c9b22de7bc0253266a3867df55acde8635e19c73313
KEY2 ^ KEY1 = 37dcb292030faa90d07eec17e3b1c6d8daf94c35d4c9191a5e1e
KEY2 ^ KEY3 = c1545756687e7573db23aa1c3452a098b71a7fbf0fddddde5fc1
FLAG ^ KEY1 ^ KEY3 ^ KEY2 = 04ee9855208a2cd59091d04767ae47963170d1660df7f56f5faf
```

> Before you XOR these objects, be sure to decode from hex to bytes.

Σχόλιο:

Από την εκφώνηση της άσκησης, μάθαμε πως ο δακτύλιος xor είναι αντιμεταθετικός. Επομένως, για να βρούμε το FLAG, η τελευταία σχέση FLAG ^ KEY1 ^ KEY3 ^ KEY2 = c1545756687e7573db23aa1c3452a098b71a7fbf0fddddde5fc1 ισοδυναμεί με

FLAG ^ KEY1 ^ (KEY3 ^ KEY2) = c1545756687e7573db23aa1c3452a098b71a7fbf0fddddde5fc1, που ισοδυναμεί με

FLAG ^ (KEY1 ^ (KEY3 ^ KEY2)) = μεταθετικότητα = (KEY1 ^ (KEY3 ^ KEY2)) ^ FLAG = c1545756687e7573db23aa1c3452a098b71a7fbf0fddddde5fc1

Και άρα έχουμε (κάνω xor με KEY1 ^ (KEY3 ^ KEY2) και στα δύο μέλη)

(KEY1 ^ (KEY3 ^ KEY2)) ^ (KEY1 ^ (KEY3 ^ KEY2)) ^ FLAG = (KEY1 ^ (KEY3 ^ KEY2)) ^ c1545756687e7573db23aa1c3452a098b71a7fbf0fddddde5fc1

Όμως λόγω της ύπαρξης του αντιθέτου στοιχείου, το (KEY1 ^ (KEY3 ^ KEY2)) ^ (KEY1 ^ (KEY3 ^ KEY2)) κάνει 0 και άρα 0 ^ FLAG= FLAG. (το 0 είναι ουδέτερο στοιχείο στη xor)

FLAG = KEY1 ^ (KEY3 ^ KEY2) ^ c1545756687e7573db23aa1c3452a098b71a7fbf0fddddde5fc1

FLAG = KEY1 ^ (c1545756687e7573db23aa1c3452a098b71a7fbf0fddddde5fc1 ^ (KEY3 ^ KEY2))

Όπου το KEY1 το γνωρίζω και το KEY2 ^ KEY3 = KEY3 ^ KEY2

In [8]:
```
def xor_bytes(bytes1, bytes2):
    #Εφαρμόζει τη λειτούργια xor σε 2 ζεύγη bytes (byte1,byte2)
    return bytes(x ^ y for x, y in zip(bytes1, bytes2))

KEY1 = bytes.fromhex('a6c8b6733c9b22de7bc0253266a3867df55acde8635e19c73313')
KEY2_XOR_KEY1 = bytes.fromhex('37dcb292030faa90d07eec17e3b1c6d8daf94c35d4c9191a5e1e')
KEY2_XOR_KEY3 = bytes.fromhex('c1545756687e7573db23aa1c3452a098b71a7fbf0fddddde5fc1')

FLAG = xor_bytes(KEY1,xor_bytes(bytes.fromhex('04ee9855208a2cd59091d04767ae47963170d1660df7f56f5faf'),KEY2_XOR_I


print(FLAG)
```

```
b'crypto{x0r_i5_ass0c1at1v3}'
```

3. Favourite byte !!!!!!

For the next few challenges, you'll use what you've just learned to solve some more XOR puzzles.

I've hidden some data using XOR with a single byte, but that byte is a secret. Don't forget to decode from hex first.

`73626960647f6b206821204f21254f7d694f7624662065622127234f726927756d`

Σχόλιο:

Γνωρίζουμε πως η μυστική λέξη έχει γίνει XOR με μόνο ένα (άγνωστο) byte. Αυτό σημαίνει πως αρκεί να κάνουμε XOR με όλα τα πιθανά στοιχεία στο μοντέλο ascii και να δούμε αν το τελικό string ξεκινάει με "crypto". Αυτό θα σημαίνει πως έχουμε βρει το flag.

In [12]:
```
string = "73626960647f6b206821204f21254f7d694f7624662065622127234f726927756d"

#Μετατρέπουμε το δεκαεξαδικό String σε bytes
string_ord = [o for o in bytes.fromhex(string)]


for order in range(256):
    #Το πιθανό flag σε μορφή λίστας αφού έχει γίνει πρώτα xor με το πιθανό μυστικό byte.
    possible_flag_ord = [order ^ o for o in string_ord]

    #Μετατρέπουμε το πιθανό flag σε string μετατρέποντας τα στοιχεία του σε ascii
    possible_flag = "".join(chr(o) for o in possible_flag_ord)
    if possible_flag.startswith("crypto"):
        flag = possible_flag
        break

print(flag)
```

```
crypto{0x10_15_my_f4v0ur173_by7e}
```

4. You either know, XOR you don't

I've encrypted the flag with my secret key, you'll never be able to guess it.

> Remember the flag format and how it might help you in this challenge!

`0e0b213f26041e480b26217f27342e175d0e070a3c5b103e2526217f27342e175d0e077e263451150104`

Σχόλιο:

Εδώ εργαστήκαμε πειραματικά με μια παρόμοια λογική όπως στην προηγούμενη άσκηση.

Γνωρίζουμε δηλαδή πως το αποκωδικοποιημένο μήνυμα ξεκινάει με "crypto{" .

Ξέρουμε όμως πως το κρυπτογραφημένο μήνυμα (πιθανώς) έχει γίνει μόνο XOR με το κλειδί χωρίς να εφαρμοστεί κάποιος επιπλέον αλγόριθμος. Επομένως, το πρώτο στοιχείο του κλειδιού έχει γίνει XOR με το πρώτο στοιχείο του (plaintext) μηνύματος. Δηλαδή,

key[0] ^ 'c' = enc_msg[0] (φυσικά πρέπει ord('c') για να επιτευχθεί η xor)

Από τις ιδιότητες της xor, αυτό σημαίνει πως key[0] = enc_msg[0] ^ 'c'

Όμως τα enc_msg[0], 'c' είναι γνωστά και άρα αν γίνουν xor μεταξύ τους θα μου δώσοουν τον πρώτο χαρακτήρα του κλειδιού.

Για τον δεύτερο χαρακτήρα, θα συνεχίσω με παρόμοια λογική. Δηλαδή,

key[0]+key[1] ^ 'cr' = enc_msg[0]+enc_msg[1]

Και άρα τελικά το

key = enc_msg[0]+enc_msg[1]+...+enc[6] + 'crypto{'

Ωστόσο για να γίνει XOR το κλειδί με το encrypted message και να γίνει η αποκρυπτογράφηση, θα πρέπει να έχουν το ίδιο length. Άρα γεμίζουμε ένα String με τους χαρακτήρες του κλειδιού μέχρι να αποκρυπτογραφηθεί το μήνυμα. Το μήνυμα θεωρείται αποκρυπτογραφημένο, όταν το τελευταίο στοιχείο είναι '}'. Επομένως, τρέχοντας τον αλγόριθμό για το κλειδί key = myXORKe , βλέπουμε πως δεν σταματάει ποτέ ο βρόγχος while μιας και δεν συναντάται ποτέ το '}' στο τέλος του decrypted message. Για αυτό υποθέσαμε πως το κλειδί δεν είναι το myXORke αλλά το myXORkey .

In [2]:
```
def xor_bytes(bytes1, bytes2):
    return bytes(x ^ y for x, y in zip(bytes1, bytes2))

enc_msg = bytes.fromhex('0e0b213f26041e480b26217f27342e175d0e070a3c5b103e2526217f27342e175d0e077e263451150104')

dec_msg = 'crypto{'

key = xor_bytes(enc_msg[:8], dec_msg[:8].encode('utf-8'))
key = key.decode('utf-8')+'y'

#key='myXORke'+'y'

pos_dec_msg = xor_bytes(key.encode('utf-8'), enc_msg)

cnt = 0
new_key = key

while chr(pos_dec_msg[-1]) != '}':

    new_key += key[cnt % 8]  # Τροποποιήθηκε για να περνάει πιο αποτελεσματικά το κλειδί προσθέτοντας ένα-ένα τ

    pos_dec_msg = xor_bytes(new_key.encode('utf-8'), enc_msg)
    cnt += 1

print(pos_dec_msg)
```

```
b'crypto{1f_y0u_Kn0w_En0uGH_y0u_Kn0w_1t_4ll}'
```

MATHEMATICS

1. Greatest Common Divisor

The Greatest Common Divisor (GCD), sometimes known as the highest common factor, is the largest number which divides two positive integers (a,b).

For `a = 12, b = 8` we can calculate the divisors of a: `{1,2,3,4,6,12}` and the divisors of b: `{1,2,4,8}`. Comparing these two, we see that `gcd(a,b) = 4`.

Now imagine we take `a = 11, b = 17`. Both `a` and `b` are prime numbers. As a prime number has only itself and `1` as divisors, `gcd(a,b) = 1`.

We say that for any two integers `a,b`, if `gcd(a,b) = 1` then `a` and `b` are coprime integers.

If `a` and `b` are prime, they are also coprime. If `a` is prime and `b < a` then `a` and `b` are coprime.

> Think about the case for `a` prime and `b > a`, why are these not necessarily coprime?

There are many tools to calculate the GCD of two integers, but for this task we recommend looking up Euclid's Algorithm.

Try coding it up; it's only a couple of lines. Use `a = 12, b = 8` to test it.

Now calculate `gcd(a,b)` for `a = 66528, b = 52920` and enter it below.

Σχόλιο:

Ο αλγόριθμος δουλεύει με την ακόλουθη λογική:

Ελέγχει αν ο αριθμός b είναι διάφορος του μηδενός. Όσο ο αριθμός b δεν είναι μηδέν, ανταλλάσσει τις τιμές των a και b με το b και το υπόλοιπο της διαίρεσης a % b αντίστοιχα. Αυτό σημαίνει ότι το a γίνεται b και το b γίνεται το υπόλοιπο της διαίρεσης a δια B. Όταν ο αριθμός b γίνει μηδέν, επιστρέφει το a ως τον ΜΚΔ των αρχικών κα ***ι b.

In [26]:
```
def gcd(a, b):
    while b != 0:
        t = b
        b = a % b
        a = t
    return a
print(gcd(52920,66528))
```

1512

2. Extended GCD

Let `a` and `b` be positive integers.

The extended Euclidean algorithm is an efficient way to find integers `u,v` such that

`a * u + b * v = gcd(a,b)`

> Later, when we learn to decrypt RSA, we will need this algorithm to calculate the modular inverse of the public exponent.

Using the two primes `p = 26513, q = 32321`, find the integers `u,v` such that

`p * u + q * v = gcd(p,q)`

Enter whichever of `u` and `v` is the lower number as the flag.

> Knowing that `p,q` are prime, what would you expect `gcd(p,q)` to be? For more details on the extended Euclidean algorithm, check out this page.

Σχόλιο:

Η συνάρτηση extended_gcd(a, b) επιστρέφει το ΜΚΔ των δύο αριθμών a και b, αλλά και τα αντίστροφα στοιχεία. Ο Γενικευμένος αλγόριθμος του Ευκλείδη επιστρέφει τρία αποτελέσματα: τον ΜΚΔ των a και b και δύο στοιχεία u και v τέτοια ώστε ua + vb = gcd(a, b).Θα πρέπει να λάβουμε υπόψη πως αν ένα από τα α και β ειναι 0 τότε η συνάρτηση επιστρέφει μια τριάδα που περιλαμβάνει:

Τον μεγαλύτερο από τους δύο αριθμούς ως το ΜΚΔ (αν ο μεγαλύτερος είναι το 0, επιστρέφει αυτόν). Το 0 ως το πρώτο αντίστροφο στοιχείο. Το 1 ως το δεύτερο αντίστροφο στοιχ

Στην περίπτωση που κανένας από τους δύο αριθμούς δεν είναι μηδέν, η συνάρτηση καλείται αναδρομικά με νέες τιμές α και β. Το νέο α θα ειναι το υπόλοιπο της διαίρεσης (β δια α) και το νέο β θα ειναι το α και άρα θα κληθεί ξανά η συνάρτηση extended_gcd(β % α, α).

Το κομμάτι αυτό της κλήσης επιστρέφει την τριάδα (gcd, v - (b // a) * u, u όπου:

gcd είναι ο ΜΚΔ των αριθμών a και b. u και v είναι τα αντίστροφα στοιχεία που προκύπτουν από την αναδρομική κλήση. Η σχέση v - (b // a) * u υπολογίζει το νέο αντίστρ.

είο.71, 1371).

In [27]:
```python
def extended_gcd(a,b):
    if min(a,b)==0:
        return (max(a,b),0,1)
    else:
        (gcd, u, v) = extended_gcd(b % a, a)
        return (gcd, v - (b // a) * u, u)

p = 26513
q = 32321

print(extended_gcd(26513, 32321))
```

```
(1, 10245, -8404)
```

3. Modular Arithmetic 1

Imagine you lean over and look at a cryptographer's notebook. You see some notes in the margin:

```
4 + 9 = 1
5 - 7 = 10
2 + 3 = 5
```

At first you might think they've gone mad. Maybe this is why there are so many data leaks nowadays you'd think, but this is nothing more than modular arithmetic modulo 12 (albeit with some sloppy notation).

You may not have been calling it modular arithmetic, but you've been doing these kinds of calculations since you learnt to tell the time (look again at those equations and think about adding hours).

Formally, "calculating time" is described by the theory of congruences. We say that two integers are congruent modulo m if `a ≡ b mod m`.

Another way of saying this, is that when we divide the integer `a` by `m`, the remainder is `b`. This tells you that if m divides a (this can be written as `m | a`) then `a ≡ 0 mod m`.

Calculate the following integers:

```
11 ≡ x mod 6
8146798528947 ≡ y mod 17
```

The solution is the smaller of the two integers.

Σχόλιο:

Υπολογίζουμε τα υπόλοιπα των διαιρέσεων και επιστρέφουμε τον μικρότερο αριθμό από τα x και y που βρήκαμε.

In [29]:
```python
a= 11%6
b= 8146798528947%17

if a<b:
    print(a)
else:
    print(b)
```

```
4
```

4. Modular Arithmetic 2 !!

We'll pick up from the last challenge and imagine we've picked a modulus `p`, and we will restrict ourselves to the case when `p` is prime.

The integers modulo `p` define a field, denoted F_p.

> ✏ If the modulus is not prime, the set of integers modulo `n` define a ring.

A finite field F_p is the set of integers `{0,1,...,p-1}`, and under both addition and multiplication there is an inverse element `b` for every element `a` in the set, such that `a + b = 0` and `a * b = 1`.

> ✏ Note that the identity element for addition and multiplication is different! This is because the identity when acted with the operator should do nothing: `a + 0 = a` and `a * 1 = a`.

Lets say we pick `p = 17`. Calculate $3^{17} \bmod 17$. Now do the same but with $5^{17} \bmod 17$.

What would you expect to get for $7^{16} \bmod 17$? Try calculating that.

This interesting fact is known as Fermat's little theorem. We'll be needing this (and its generalisations) when we look at RSA cryptography.

Now take the prime `p = 65537`. Calculate $273246787654^{65536} \bmod 65537$.

Did you need a calculator?

Από το μικρό θεώρημα του Fermat έχουμε ότι a^(p-1) = 1 mod p. Άρα η απάντηση είναι 1.

5. Modular Inverting

As we've seen, we can work within a finite field F_p, adding and multiplying elements, and always obtain another element of the field.

For all elements `g` in the field, there exists a unique integer `d` such that `g * d ≡ 1 mod p`.

This is the multiplicative inverse of `g`.

Example: `7 * 8 = 56 ≡ 1 mod 11`

What is the inverse element: `3 * d ≡ 1 mod 13`?

> Think about the little theorem we just worked with. How does this help you find the inverse of an element?

In [3]:
```
pow(3, 13-2, 13)
```
Out[3]: 9

Challenge Page : Mathematics

MODULAR MATH

1. Quadratic Residues

We've looked at multiplication and division in modular arithmetic, but what does it mean to take the square root modulo an integer?

For the following discussion, let's work modulo `p = 29`. We can take the integer `a = 11` and calculate $a^2 = 5 \bmod 29$.

As $a = 11, a^2 = 5$, we say the square root of `5` is `11`.

This feels good, but now let's think about the square root of `18`. From the above, we know we need to find some integer `a` such that $a^2 = 18$

Your first idea might be to start with `a = 1` and loop to `a = p-1`. In this discussion `p` isn't too large and we can quickly look.

Have a go, try coding this and see what you find. If you've coded it right, you'll find that for all $a \in F_p^*$ you never find an `a` such that $a^2 = 18$.

What we are seeing, is that for the elements of F^*_p, not every element has a square root. In fact, what we find is that for roughly one half of the elements of F_p^*, there is no square root.

> We say that an integer `x` is a *Quadratic Residue* if there exists an `a` such that $a^2 = x \bmod p$. If there is no such solution, then the integer is a *Quadratic Non-Residue*.

In other words, `x` is a quadratic residue when it is possible to take the square root of `x` modulo an integer `p`.

In the below list there are two non-quadratic residues and one quadratic residue.

Find the quadratic residue and then calculate its square root. Of the two possible roots, submit the smaller one as the flag.

> If $a^2 = x$ then $(-a)^2 = x$. So if `x` is a quadratic residue in some finite field, then there are always two solutions for `a`.

```
p = 29
ints = [14, 6, 11]
```

In [5]:
```
from euclidean_algorithm import *

def is_perfect_square(n):
    i = 1
    while i * i <= n:
        if i * i == n:
            return True
        i += 1
    return False

def quad_res(a,p):
    pos_a= [i for i in range(1,p) if gcd(p,i)==1]
    return_list=[]
    for b in pos_a:
        if is_perfect_square(p*b+a)==True:
             return_list.append(p*b+a)
    return return_list

def main():
    square_root=[]
    p = 29
    ints = [14, 6, 11]
    for i in ints:
       if quad_res(i,p) != []:
        for j in quad_res(i,p):
```

```python
            square_root.append(j**0.5)
    return square_root

anwsers=[]
for i in main():
    anwsers.append(i)
print(min(anwsers))
```

8.0

2. Legendre Symbol

In Quadratic Residues we learnt what it means to take the square root modulo an integer. We also saw that taking a root isn't always possible.

In the previous case when `p = 29`, even the simplest method of calculating the square root was fast enough, but as `p` gets larger, this method becomes wildly unreasonable.

Lucky for us, we have a way to check whether an integer is a quadratic residue with a single calculation thanks to Legendre. In the following, we will assume we are working modulo a prime `p`.

Before looking at Legendre's symbol, let's take a brief detour to see an interesting property of quadratic (non-)residues.

```
Quadratic Residue * Quadratic Residue = Quadratic Residue
Quadratic Residue * Quadratic Non-residue = Quadratic Non-residue
Quadratic Non-residue * Quadratic Non-residue = Quadratic Residue
```

> Want an easy way to remember this? Replace "Quadratic Residue" with `+1` and "Quadratic Non-residue" with `-1`, all three results are the same!

So what's the trick? The Legendre Symbol gives an efficient way to determine whether an integer is a quadratic residue modulo an odd prime `p`.

Legendre's Symbol: $(a / p) \equiv a^{(p-1)/2} \bmod p$ obeys:

```
(a / p) = 1 if a is a quadratic residue and a ≢ 0 mod p
(a / p) = -1 if a is a quadratic non-residue mod p
(a / p) = 0 if a ≡ 0 mod p
```

Which means given any integer `a`, calculating `pow(a,(p-1)//2,p)` is enough to determine if `a` is a quadratic residue.

Now for the flag. Given the following 1024 bit prime and 10 integers, find the quadratic residue and then calculate its square root; the square root is your flag. Of the two possible roots, submit the larger one as your answer.

> So Legendre's symbol tells us which integer is a quadratic residue, but how do we find the square root?! The prime supplied obeys `p = 3 mod 4`, which allows us easily compute the square root. The answer is online, but you can figure it out yourself if you think about Fermat's little theorem.

Challenge files:
 - output.txt

In [9]:
```python
p = 10152403517453989048540857567108526178875896518906016448438569080146616735666703667793299888972547658242173

ints = [25081841204695904475894082974192007718642931811040324543182130088804239047149283334700530600468528298920

for i in ints:
    if pow(i,(p-1)//2,p)==1:
        print(pow(i,(p+1)//4,p))
```

93291799125366706806545638475797430512104976066103610269938025709952247020061090804870186195285998727680200979853848718589126765742550855954805290253592144209552123062161458584575060939481368210688629862036958857604707468372384278049741369153506182660264876115428251983455344219194133033177700490981696141526

3. Chinese Remainder Theorem

The Chinese Remainder Theorem gives a unique solution to a set of linear congruences if their moduli are coprime.

This means, that given a set of arbitrary integers a_i, and pairwise coprime integers n_i, such that the following linear congruences hold:

> Note "pairwise coprime integers" means that if we have a set of integers $\{n_1, n_2, \ldots, n_i\}$, all pairs of integers selected from the set are coprime: $\gcd(n_i, n_j) = 1$.

$$x \equiv a_1 \bmod n_1$$
$$x \equiv a_2 \bmod n_2$$
$$\ldots$$
$$x \equiv a_n \bmod n_n$$

There is a unique solution $x \equiv a \bmod N$ where $N = n_1 * n_2 * \ldots * n_n$.

In cryptography, we commonly use the Chinese Remainder Theorem to help us reduce a problem of very large integers into a set of several, easier problems.

Given the following set of linear congruences:

```
x ≡ 2 mod 5
x ≡ 3 mod 11
x ≡ 5 mod 17
```

Find the integer `a` such that `x ≡ a mod 935`

> Starting with the congruence with the largest modulus, use that for `x ≡ a mod p` we can write `x = a + k*p` for arbitrary integer `k`.

In [10]:
```python
from Crypto.Util.number import *

print((2*11*17*inverse(11*17,5) + 3*5*17*inverse(5*17,11) + 5*5*11*inverse(5*11,17)) % 935)
```

872

Challenge Page : SYMMETRIC CIPHERS

HOW AES WORKS

1. Keyed Permutations

AES, like all good block ciphers, performs a "keyed permutation". This means that it maps every possible input block to a unique output block, with a key determining which permutation to perform.

> A "block" just refers to a fixed number of bits or bytes, which may represent any kind of data. AES processes a block and outputs another block. We'll be specifically talking the variant of AES which works on 128 bit (16 byte) blocks and a 128 bit key, known as AES-128.

Using the same key, the permutation can be performed in reverse, mapping the output block back to the original input block. It is important that there is a one-to-one correspondence between input and output blocks, otherwise we wouldn't be able to rely on the ciphertext to decrypt back to the same plaintext we started with.

What is the mathematical term for a one-to-one correspondence?

crypto{bijection}

2. Resisting Bruteforce

If a block cipher is secure, there should be no way for an attacker to distinguish the output of AES from a random permutation of bits. Furthermore, there should be no better way to undo the permutation than simply bruteforcing every possible key. That's why academics consider a cipher theoretically "broken" if they can find an attack that takes fewer steps to perform than bruteforcing the key, even if that attack is practically infeasible.

> How difficult is it to bruteforce a 128-bit keyspace? Somebody estimated that if you turned the power of the entire Bitcoin mining network against an AES-128 key, it would take over a hundred times the age of the universe to crack the key.

It turns out that there is an attack on AES that's better than bruteforce, but only slightly – it lowers the security level of AES-128 down to 126.1 bits, and hasn't been improved on for over 8 years. Given the large "security margin" provided by 128 bits, and the lack of improvements despite extensive study, it's not considered a credible risk to the security of AES. But yes, in a very narrow sense, it "breaks" AES.

Finally, while quantum computers have the potential to completely break popular public-key cryptosystems like RSA via Shor's algorithm, they are thought to only cut in half the security level of symmetric cryptosystems via Grover's algorithm. This is one reason why people recommend using AES-256, despite it being less performant, as it would still provide a very adequate 128 bits of security in a quantum future.

What is the name for the best single-key attack against AES?

crypto{biclique}

3. Structure of AES

Σχόλιο :

Θα χρειαστεί να μετατρέψουμε όλα τα στοιχεία του πίνακα σε χαρακτήρες ascii και να επιστρέψουμε τον νέο πίνακα σε μορφή string

In [14]:
```
def bytes2matrix(text):
    """ Converts a 16-byte array into a 4x4 matrix.  """
    return [list(text[i:i+4]) for i in range(0, len(text), 4)]

def matrix2bytes(matrix):
    i=0
    j=0
    for i in range(0,len(matrix)):
        for j in range(0,len(matrix)):
            #μετατρέπει το κάθε στοιχείο του πίνακα σε χαρακτήρα στο ascii μοντέλο
            matrix[i][j]=chr((matrix[i][j]))
    #μετατρέπουμε τον πίνακα σε String διαγράφοντας τα '[',']',"'",' ',','
    return str(matrix).replace('[','').replace(']','').replace(',','').replace("'",'').replace(" ",'')

matrix = [
    [99, 114, 121, 112],
    [116, 111, 123, 105],
    [110, 109, 97, 116],
    [114, 105, 120, 125],
]

print(matrix2bytes(matrix))
```

```
crypto{inmatrix}
```

3. Round Keys

Σχόλια:

Η συνάρτηση add_round_key(s,k) παίρνει δύο αντικείμενα ως όρισμα. Τον πίνακα state (s) και τον πίνακα round_key (k). Θα χρειαστεί να κάνει XOR ανάμεσα σε κάθε byte των δύο πινάκων και να αποθηκεύσει το αποτέλεσμα σε έναν νέο πίνακα. Αυτό το αποτέλεσμα θα μετατραπεί σε String με βάση το matrix2bytes που δημιουργήσαμε στην προηγούμενη άσκηση.

In [15]:
```python
from matrix2bytes import *

state = [
    [206, 243, 61, 34],
    [171, 11, 93, 31],
    [16, 200, 91, 108],
    [150, 3, 194, 51],
]

round_key = [
    [173, 129, 68, 82],
    [223, 100, 38, 109],
    [32, 189, 53, 8],
    [253, 48, 187, 78],
]


def add_round_key(s, k):
    i=0
    j=0
    addroundkeymatrix=[[0 for col in range(4)] for row in range(4)]
    for i in range(0,len(s)):
        for j in range(0,len(s)):
            addroundkeymatrix[i][j]=s[i][j]^k[i][j]
    return addroundkeymatrix

print(matrix2bytes(add_round_key(state, round_key)))
```

```
crypto{r0undk3y}
```

4. Confusion through Substitution

Σχόλιο:

Για κάθε στοιχείο στον πίνακα state, θα βρούμε την αντίστοιχη θέση του στον πίνακα sbox.

In [16]:
```python
from matrix2bytes import *

s_box = (
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
    0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
    0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
    0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16,
)

inv_s_box = (
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF,
```

```
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D,
)

state = [
    [251, 64, 182, 81],
    [146, 168, 33, 80],
    [199, 159, 195, 24],
    [64, 80, 182, 255],
]



def sub_bytes(s, sbox=s_box):
    i=0
    j=0

    #Δημιουργούμε έναν μηδενικό πίνακα 4x4
    sub_bytes_matrix = [[0 for col in range(4)] for row in range(4)]


    for i in range(0,len(s)):
        for j in range(0,len(s)):
            try:
              #το hex(s[i][j]) είναι της μορφής '0xAB' και άρα το A μου δείχνει τη σειρά του πίνακα sbox και άρ
              #θα πάρω μια αξιοποιήσημη τιμή για τη θέση.

              #το hex(s[i][j]) είναι της μορφής '0xAB' και άρα το B μου δείχνει τη στήλη του πίνακα sbox και άρ
              #θα πάρω μια αξιοποιήσημη τιμή για τη θέση.

              #Επειδή το sbox είναι σε μορφή λίστας και όχι πίνακα της μορφής sbox[i][j] θα πρέπει το position
              #προτίστως τη σειρά και μετά τη στήλη και άρα τελικώς να καταλήξει στο στοιχείο sbox[position]

              position=16 * int(hex(s[i][j])[2],16) + int(hex(s[i][j])[3],16)
            except:
               #Στη περίπτωση που το hex(s[i][j]) είναι της μορφής '0xA' και άρα δεν υπάρχει το 4ο στοιχείο
               position=16 * int(hex(s[i][j])[2],16)
            sub_bytes_matrix[i][j]=sbox[position]

    return sub_bytes_matrix



print(matrix2bytes(sub_bytes(state, sbox=inv_s_box)))
```

```
crypto{l1n34rly}
```

5. Diffusion through Permutation

Σχόλιο:

Εδώ απλά αλλάζουμε τις γραμμές στο inv_shift_rows με βάση τον αλγόριθμο.

In [18]:

```
from matrix2bytes import *

def shift_rows(s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[1][1], s[2][1], s[3][1], s[0][1]
    s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2]
    s[0][3], s[1][3], s[2][3], s[3][3] = s[3][3], s[0][3], s[1][3], s[2][3]



def inv_shift_rows(s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[3][1], s[0][1], s[1][1], s[2][1]
    s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2]
    s[0][3], s[1][3], s[2][3], s[3][3] = s[1][3], s[2][3], s[3][3], s[0][3]




# learned from http://cs.ucsb.edu/~koc/cs178/projects/JT/aes.c
xtime = lambda a: (((a << 1) ^ 0x1B) & 0xFF) if (a & 0x80) else (a << 1)


def mix_single_column(a):
    # see Sec 4.1.2 in The Design of Rijndael
    t = a[0] ^ a[1] ^ a[2] ^ a[3]
    u = a[0]
    a[0] ^= t ^ xtime(a[0] ^ a[1])
    a[1] ^= t ^ xtime(a[1] ^ a[2])
    a[2] ^= t ^ xtime(a[2] ^ a[3])
    a[3] ^= t ^ xtime(a[3] ^ u)
```

```python
def mix_columns(s):
    for i in range(4):
        mix_single_column(s[i])


def inv_mix_columns(s):
    # see Sec 4.1.3 in The Design of Rijndael
    for i in range(4):
        u = xtime(xtime(s[i][0] ^ s[i][2]))
        v = xtime(xtime(s[i][1] ^ s[i][3]))
        s[i][0] ^= u
        s[i][1] ^= v
        s[i][2] ^= u
        s[i][3] ^= v

    mix_columns(s)



state = [
    [108, 106, 71, 86],
    [96, 62, 38, 72],
    [42, 184, 92, 209],
    [94, 79, 8, 54],
]

inv_mix_columns(state)
inv_shift_rows(state)

print(matrix2bytes(state))
```

```
crypto{d1ffUs3R}
```

6. Bringing It All Together

Σχόλιο:

Πλέον εφαρμόζουμε κλιμακωτά ό,τι αναπτύξαμε στις προηγούμενες ασκήσεις.

Επομένως θα χρειαστεί να εισάγουμε στο script μας κάποιες συναρτήσεις που ήταν ανσωματωμένες στις προηγούμενες ασκήσεις.

Εισάγουμε τις συναρτήσεις bytes2matrix, shift_rows, inv_shift_rows, inv_sub_bytes, mix_single_column, mix_columns, inv_mix_columns.

Στη συνάρτηση decrypt, απλά εφαρμόζουμε με τη σειρά τα βήματα της αποκτυπτοράφησης όπως μας υποδεικνύει και η εικόνα της εκφώνησης.

In [19]:
```python
from sbox import *
from matrix2bytes import *
from add_roundkey import *

N_ROUNDS = 10

key        = b'\xc3,\\\xa6\xb5\x80^\x0c\xdb\x8d\xa5z*\xb6\xfe\\'
ciphertext = b'\xd1O\x14j\xa4+O\xb6\xa1\xc4\x08B)\x8f\x12\xdd'

def bytes2matrix(text):
    return [list(text[i:i+4]) for i in range(0, len(text), 4)]

def shift_rows(s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[1][1], s[2][1], s[3][1], s[0][1]
    s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2]
    s[0][3], s[1][3], s[2][3], s[3][3] = s[3][3], s[0][3], s[1][3], s[2][3]



def inv_shift_rows(s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[3][1], s[0][1], s[1][1], s[2][1]
    s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2]
    s[0][3], s[1][3], s[2][3], s[3][3] = s[1][3], s[2][3], s[3][3], s[0][3]

def inv_sub_bytes(s, sbox=inv_s_box):
    for i in range(len(s)):
        for j in range(len(s[i])):
            s[i][j] = (sbox[s[i][j]])


# learned from http://cs.ucsb.edu/~koc/cs178/projects/JT/aes.c
xtime = lambda a: (((a << 1) ^ 0x1B) & 0xFF) if (a & 0x80) else (a << 1)


def mix_single_column(a):
    # see Sec 4.1.2 in The Design of Rijndael
```

```
    t = a[0] ^ a[1] ^ a[2] ^ a[3]
    u = a[0]
    a[0] ^= t ^ xtime(a[0] ^ a[1])
    a[1] ^= t ^ xtime(a[1] ^ a[2])
    a[2] ^= t ^ xtime(a[2] ^ a[3])
    a[3] ^= t ^ xtime(a[3] ^ u)


def mix_columns(s):
    for i in range(4):
        mix_single_column(s[i])


def inv_mix_columns(s):
    # see Sec 4.1.3 in The Design of Rijndael
    for i in range(4):
        u = xtime(xtime(s[i][0] ^ s[i][2]))
        v = xtime(xtime(s[i][1] ^ s[i][3]))
        s[i][0] ^= u
        s[i][1] ^= v
        s[i][2] ^= u
        s[i][3] ^= v

    mix_columns(s)

def expand_key(master_key):
    """
    Expands and returns a list of key matrices for the given master_key.
    """

    # Round constants https://en.wikipedia.org/wiki/AES_key_schedule#Round_constants
    r_con = (
        0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40,
        0x80, 0x1B, 0x36, 0x6C, 0xD8, 0xAB, 0x4D, 0x9A,
        0x2F, 0x5E, 0xBC, 0x63, 0xC6, 0x97, 0x35, 0x6A,
        0xD4, 0xB3, 0x7D, 0xFA, 0xEF, 0xC5, 0x91, 0x39,
    )

    # Initialize round keys with raw key material.
    key_columns = bytes2matrix(master_key)
    iteration_size = len(master_key) // 4

    # Each iteration has exactly as many columns as the key material.
    i = 1
    while len(key_columns) < (N_ROUNDS + 1) * 4:
        # Copy previous word.
        word = list(key_columns[-1])

        # Perform schedule_core once every "row".
        if len(key_columns) % iteration_size == 0:
            # Circular shift.
            word.append(word.pop(0))
            # Map to S-BOX.
            word = [s_box[b] for b in word]
            # XOR with first byte of R-CON, since the others bytes of R-CON are 0.
            word[0] ^= r_con[i]
            i += 1
        elif len(master_key) == 32 and len(key_columns) % iteration_size == 4:
            # Run word through S-box in the fourth iteration when using a
            # 256-bit key.
            word = [s_box[b] for b in word]

        # XOR with equivalent word from previous iteration.
        word = bytes(i^j for i, j in zip(word, key_columns[-iteration_size]))
        key_columns.append(word)

    # Group key words in 4x4 byte matrices.
    return [key_columns[4*i : 4*(i+1)] for i in range(len(key_columns) // 4)]


def decrypt(key, ciphertext):
    round_keys = expand_key(key) # Remember to start from the last round key and work backwards through them whe


    state=bytes2matrix(ciphertext)
    # Initial add round key step
    state=add_round_key(state, round_keys[-1])


    for i in range(N_ROUNDS - 1, 0, -1):
        inv_shift_rows(state)
        inv_sub_bytes(state, sbox=inv_s_box)
        state=add_round_key(state, round_keys[i])
```

```
        inv_mix_columns(state)

    # Run final round (skips the InvMixColumns step)
    inv_shift_rows(state)
    inv_sub_bytes(state, sbox=inv_s_box)
    state=add_round_key(state, round_keys[0])

    # Convert state matrix to plaintext

    return matrix2bytes(state)

print(decrypt(key, ciphertext))
```

```
crypto{MYAES128}
```

SYMMETRIC STARTER

1. Modes of Operation Starter

The previous set of challenges showed how AES performs a keyed permutation on a block of data. In practice, we need to encrypt messages much longer than a single block. A *mode of operation* describes how to use a cipher like AES on longer messages.

All modes have serious weaknesses when used incorrectly. The challenges in this category take you to a different section of the website where you can interact with APIs and exploit those weaknesses. Get yourself acquainted with the interface and use it to take your next flag!

Play at https://aes.cryptohack.org/block_cipher_starter

Σχόλιο-Λύση:

Χρησιμοποιώντας τα εργαλεία που μας δίνονται στο website https://aes.cryptohack.org/block_cipher_starter, παίρνουμε πρώτα το ciphertext από το output κάνοντας submit στο πεδίο encrypt_flag.

{"ciphertext":"188bc9522883a992cd99f9b2b8ec84fb150c1abe17e7a8ed04a768ad1d11fc07"}

Στη συνέχεια βάζουμε το ciphertext 188bc9522883a992cd99f9b2b8ec84fb150c1abe17e7a8ed04a768ad1d11fc07 στο πεδίο decrypt και πατάμε submit

{"plaintext":"63727970746f7b626c30636b5f6331706833723555f3472335f663435375f217d"}

Και τέλος, πάμε στη καρτέλα HEX ENCODER/DECODER όπου εκεί βάζουμε το δεκαεξαδικο string '63727970746f7b626c30636b5f6331706833723555f3472335f663435375f217d' στο αντίστοιχο πεδίο και το μετρέπουμε σε text.

crypto{bl0ck_c1ph3r5_4r3_f457_!}

2. Passwords as Keys

It is essential that keys in symmetric-key algorithms are random bytes, instead of passwords or other predictable data. The random bytes should be generated using a cryptographically-secure pseudorandom number generator (CSPRNG). If the keys are predictable in any way, then the security level of the cipher is reduced and it may be possible for an attacker who gets access to the ciphertext to decrypt it.

Just because a key looks like it is formed of random bytes, does not mean that it necessarily is. In this case the key has been derived from a simple password using a hashing function, which makes the ciphertext crackable.

For this challenge you may script your HTTP requests to the endpoints, or alternatively attack the ciphertext offline. Good luck!

Play at https://aes.cryptohack.org/passwords_as_keys

Σχόλιο:

Εφόσον έχουμε ένα wordlist txt αρχείο, θα κάνουμε bruteforce το ciphertext με το hash της πιθανής λέξης κλειδί.

Θα μπορούσαμε να λειτουργήσουμε και online με τη βιβλιοθήκη requests ωστόσο αυτό θα ήταν και χρονοβόρο αλλά και επικύνδυνο στο να μας αποκλείσει ο server (αν και πιθανώς θα έχει προνοήσει για αυτό το cryptohack)

In [ ]:

```
from Crypto.Cipher import AES
import hashlib
import binascii

with open("words.txt",'r') as f:
    words = [w.strip() for w in f.readlines()]

def decrypt(ciphertext_hex, words):
```

```
    for w in words:

        #Μετατρέπουμε τη λέξη w σε md5 hash
        attempted_key = hashlib.md5(w.encode()).hexdigest()

        ciphertext = bytes.fromhex(ciphertext_hex)

        key=bytes.fromhex(attempted_key)

        cipher = AES.new(key, AES.MODE_ECB)

        decrypted = cipher.decrypt(ciphertext)
        try:
            result = binascii.unhexlify(decrypted.hex())

            if result.startswith('crypto{'.encode()):
                return result

        except:
            continue

print(decrypt("c92b7734070205bdf6c0087a751466ec13ae15e6f1bcdd3f3a535ec0f4bbae66",words))
```

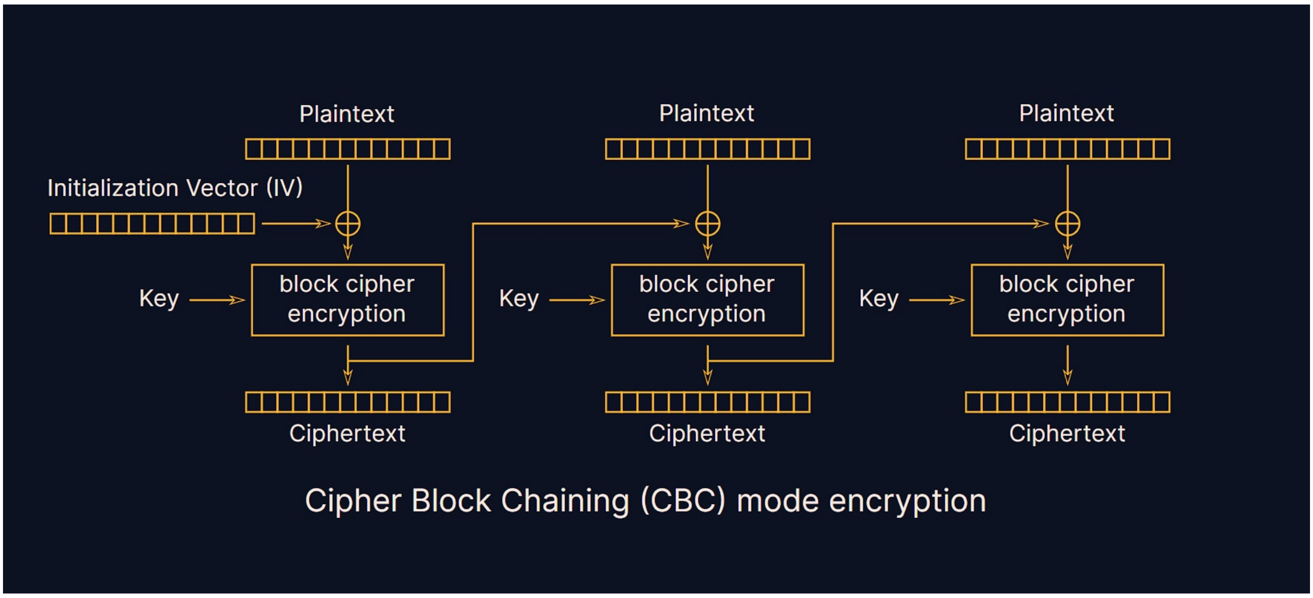
b'crypto{k3y5__r__n07__p455w0rdz?}'

BLOCK CIPHERS

1. ECB CBC WTF

SOURCE

```
from Crypto.Cipher import AES


KEY = ?
FLAG = ?


@chal.route('/ecbcbcwtf/decrypt/<ciphertext>/')
def decrypt(ciphertext):
    ciphertext = bytes.fromhex(ciphertext)

    cipher = AES.new(KEY, AES.MODE_ECB)
    try:
        decrypted = cipher.decrypt(ciphertext)
    except ValueError as e:
        return {"error": str(e)}

    return {"plaintext": decrypted.hex()}


@chal.route('/ecbcbcwtf/encrypt_flag/')
def encrypt_flag():
    iv = os.urandom(16)

    cipher = AES.new(KEY, AES.MODE_CBC, iv)
    encrypted = cipher.encrypt(FLAG.encode())
    ciphertext = iv.hex() + encrypted.hex()

    return {"ciphertext": ciphertext}
```



Σχόλια:

Αρχικά παρατηρούμε πως η συνάρτηση encrypt_flag επιστρέφει το iv μαζί με το encypted μήνυμα. Επομένως οι 32 πρώτοι χαρακτήρες του ciphertext είναι το iv.

Όμως, στο source μας δίνεται και η συνάρτηση decrypt και άρα το μόνο που μένει είναι να αντιστρέψουμε τη μέθοδο ecb προκειμένου να αποκρυπτογραφίσουμε το μήνυμα.

Για την κρυπτογράφηση του μηνύματος το plaintext χωρίστηκε σε 2 ίσα blocks των 32 bits. Και άρα κρυπτογραφήθηκε όπως παρακάτω,

ciphertext = ciphertext1 + encrypt(key, plaintext2 ^ ciphertext1) , ciphertext1 = encrypt(key, plaintext1 ^ iv)

Επομένως, για την αποκρυπτογράφηση, έγινε

ciphertext1 = encrypt(key, plaintext1 ^ iv) και άρα plaintext1 = decrypt(ciphertext1) ^ iv

Επίσης, το ciphertext2 = encrypt(key, plaintext2 ^ encrypt(key, plaintext1 ^ iv)) και άρα plaintext2 = decrypt(plaintext2) ^ ciphertext1

Επομένως, το plaintext = decrypt(ciphertext1) ^ iv + decrypt(plaintext2) ^ ciphertext1

In [7]:
```
import requests
from pwn import xor

def decrypt(ciphertext):
    url = "https://aes.cryptohack.org/ecbcbcwtf/decrypt/"
    response = requests.get(url+ciphertext)
    response_json = response.json()['plaintext']

    return response_json

def main():
    ciphertext_iv="a42488d0ca4c29bcd7fb313ede555f810d693be62af32a237691610be8f0a40295d8f42c88575422f445eaaf29dd

    iv=bytes.fromhex(ciphertext_iv[0:32])

    ciphertext_1st_block=ciphertext_iv[32:64]
    ciphertext_2nd_block=ciphertext_iv[64:]

    plaintext_1st_block=bytes.fromhex(decrypt(ciphertext_1st_block))
    plaintext_2nd_block=bytes.fromhex(decrypt(ciphertext_2nd_block))

    return xor(iv,plaintext_1st_block).decode() + xor(bytes.fromhex(ciphertext_1st_block),plaintext_2nd_block).


print(main())
```

```
crypto{3cb_5uck5_4v01d_17_!!!!!}
```

2. ECB CBC WTF

Ο παρακάτω κώδικας είναι φτιαγμένος να τρέχει Online, πράγμα που τον καθιστά αρκετά χρονοβόρο.

Παρατήρουμε από το source πως στη κλήση της συνάρτησης encrypt(plaintext) επιστρέφεται ένα ciphertext της μορφής plaintext+FLAG.

Άρα σκεφτόμαστε ως εξής.

Εφόσον το plaintext το ορίζουμε εμείς, μπορούμε να μετακινήσουμε το FLAG έτσι ώστε ο **πρώτος** χαρακτήρας του FLAG να είναι ο

τελευταίος χαρακτήρας του 1ου ciphertext block.

πχ. Αν το plaintext = 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA' (31 φορές το A) και το κάθε block έχει length 32 τότε το 1ο block του ciphertext θα είναι της μορφής 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'+x, όπου x είναι το πρώτο γράμμα του FLAG.

Επομένως το μόνο που έχουμε να κάνουμε είναι να κρυπτογραφήσουμε κάθε string της μορφής AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**x** για όλα τα x που ανήκουν στους χαρακτήρες ascii και να το συγκρίνουμε με το 1ο Block του ciphertext που μας επέστρεψε η αρχική μας είσοδος.

Αν αυτα τα δύο strings ταυτίζονται τότε ο ascii χαρακτήρας που βάλαμε είναι το πρώτο γράμμα του FLAG.

Συνεχίζουμε αφαιρώντας 1 A από το αρχικό string που εισάγαμε προηγουμένως στη συνάρτηση encrypt.

Τώρα το ciphertext θα είναι της μορφής AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**xy**, όπου το x το βρήκαμε προηγουμένως.

Άρα, το νέο string έχει την ίδια μορφή με το παραπάνω και επομένως εφαρμόζουμε την ίδια διαδικασία μέχρι να βρούμε το επόμενο ascii στοιχείο που θα είναι το 2ο γράμμα για το FLAG.

Συνεχίζουμε μέχρι το τελευταίο στοιχείο να γίνει '}' που σημαίνει πως θα έχουμε βρει το flag

In [10]:
```python
import requests
import codecs


url = "https://aes.cryptohack.org/ecb_oracle/encrypt/"

#we find the position

def find_same_blocks(response_json_list):
    for i in response_json_list:
        if response_json_list.count(i)>1:
            return i


def find_character(reference,found):

    for i in range(0, 255):
        plaintext_test="A"*(31-len(found))+found+chr(i)
        plaintext_test_hex=plaintext_test.encode().hex()

        response = requests.get(url+plaintext_test_hex)
        response_json = response.json()['ciphertext']

        if reference==response_json[0:64]:
            return chr(i)

secret=''
cnt=0

while secret=="" or secret[-1]!="}":

    plaintext_test="A"*(31-cnt)

    plaintext_test_hex=plaintext_test.encode().hex()

    response = requests.get(url+plaintext_test_hex)
    response_json = response.json()['ciphertext']

    found_character=find_character(response_json[0:64],secret)

    secret+=found_character
    cnt+=1
print(secret)
```

```
crypto{p3n6u1n5_h473_3cb}
```

3. Flipping Cookie

Σχόλιο :

Ρίχοντας μια ματιά στο source, παρατηρούμε πως η συνάρτηση check_admin, επιπστρέφει το FLAG μόνο στην περίπτωση που το cookie ξεκινά με "admin=True". Ωστόσο, από τη συνάρτηση get_cookie() μας επιστρέφεται ένα encrypted μήνυμα το οποίο ξεκινά με "admin=False;".

Επομενώς, θα πρέπει να αλλάξουμε το κρυπτογραφημένο μήνυμα έτσι ώστε να ξεκινά με "admin=True" ώστε να νομίζει Οι τρόποι για να το πετύχουμε είναι είτε αποκρυπτογραφώντας το μήνυμα, είτε αλλάζοντας το iv. Και αυτό γιατί το string που θέλουμε να τροποποιήσουμε βρίσκεται στο πρώτο block.

Άρα έχουμε,

plaintext1 = cipher0 ^ dec(cipher1) => d(ipherc1) = laintextp1 ^ ipherc , όπου το plaintext1= 'admin=False;expi'

Όμως , θέλουμε το correct_plaintext1='admin=True;expir' και άρα plaintext1 = newcipher0 ^ dec(cipher1) => plaintext1 = newcipher0 ^ cipher0 ^ dec(cipher1)

Άρα, cipher ^ iv = plain => cipher = plain ^ iv => correct_plaintext1 = cipher ^ new_iv => new_iv = correct_plaintext1 ^ cipher => new_iv = correct_plaintext1 ^ plaintext1 ^ iv

In [12]:
```python
from Crypto.Util.number import *
import requests

def getcookie():
    url = "https://aes.cryptohack.org/flipping_cookie/get_cookie/"
    request = requests.get(url)
    return request.json()["cookie"]

def getflag(cookie, iv):
    url = f"https://aes.cryptohack.org/flipping_cookie/check_admin/{cookie}/{iv}"
    request = requests.get(url)
    return request.json()

cookie = getcookie()
print(f"Cookie: {cookie}")
print(f"iv : {cookie[:32]}")

iv = cookie[:32]

plaintext_b = b'admin=False;expi' #Πρέπει να είναι 32bit
plaintext_a = b'admin=True;expir' #Πρέπει να είναι 32bit

plaintext_b_hex = hex(bytes_to_long(plaintext_b))
plaintext_a_hex = hex(bytes_to_long(plaintext_a))

new_iv = hex(int(plaintext_b_hex, 16) ^ int(plaintext_a_hex, 16) ^ int(iv, 16))[2:]

print(f"New iv : {iv}")

print(getflag(cookie[32:], new_iv))
```

```
Cookie: d78e0d48d72707d2dbccb8caf381d59701c948b817c7e9ebb71dac8d37499bb2c9f0f5b72e1751f05a60460e824a26aa
iv : d78e0d48d72707d2dbccb8caf381d597
New iv : d78e0d48d72707d2dbccb8caf381d597
{'flag': 'crypto{4u7h3n71c4710n_15_3553n714l}'}
```

4. Lazy CBC

Σχόλιο :

Εδώ το πρόβλημα είναι πως χρησιμοποιείται το κλειδί ως iv για την κρυπτογράφηση.

Ας υποθέσουμε πως το plaintext είναι 32 bytes και άρα μπορεί να χωριστεί σε 2 blocks κατά την αποκρυπτογράφηση.

Τότε για να πάρω το πρώτο block plaintext θα χρειαστεί να γίνει decrypted(ciphertext1) ^ key όπως επίσης για να πάρω το 2ο block θα χρειαστεί να γίνει ciphertext1 ^ decrypted(ciphertext2)

Άρα αυτό που χρειαζόμαστε είναι το decrypted(ciphertext1) και το decrypted(ciphertext2)

Όμως αν κάνουμε manipulate το 2ο block γεμίζοντάς το με μηδενικά και αυξήσουμε το string κατά 1 block τότε θα

1ο block (plaintext) : decrypted(ciphertext1) ^ key 2ο block (plaintext) : ciphertext1 ^ decrypted(ciphertext2) 3ο block (plaintext) : ciphertext2 ^ decrypted(ciphertext3) = (ciphertext2=μηδενικά) = decrypted(ciphertext3)

Και άρα

1ο block (plaintext) : decrypted(ciphertext1) ^ key 2ο block (plaintext) : ciphertext1 ^ decrypted(ciphertext2) 3ο block (plaintext) : decrypted(ciphertext3)

Όμως, αν αντικαταστήσω το 3ο block με το 1ο block ciphertext παρατηρώ το ε

1ο block (plaintext) : decrypted(ciphertext1) ^ key 2ο block (plaintext) : ciphertext1 ^ decrypted(ciphertext2) (Δεν μας απασχολεί) 3ο block (plaintext) : decrypted(ciphertext1)

Επομένως κρυπτογραφώ ένα τυχαίο μήνυμα 3 block και στο ciphertext που θα πάρω αντικαθηστώ το 2ο block με μηδενικά και το 3ο block με το 1ο

Βάζοντάς το στο receive παίρνω το αποκρυπτογραφημένο plaintext όπου για να πάρω το κλειδί θα χρειαστέι να κάνω xor το 1ο με το 3ο block.

In [1]:
```python
import requests
```

```
def encrypt(plaintext):
    url = "https://aes.cryptohack.org/lazy_cbc/encrypt/"+plaintext
    request = requests.get(url)
    return request.json()["ciphertext"]

def get_flag(key):
    url = "https://aes.cryptohack.org/lazy_cbc/get_flag/"+key
    request = requests.get(url)
    return request.json()["plaintext"]

def receive(ciphertext):
    url = "https://aes.cryptohack.org/lazy_cbc/receive/"+ciphertext
    request = requests.get(url)
    return request.json()["error"][19:]

plaintext="756172656c617a79756172656c617a79756172656c617a79756172656c617a79" #hex of "uarelazyuarelazyuarelazyu


ciphertext=encrypt(plaintext)

changed_ciphertext=ciphertext[:32]+"0"*32+ciphertext[:32]

invalid_plaintext=receive(changed_ciphertext)

key=hex(int(invalid_plaintext[:32],16)^int(invalid_plaintext[64:],16))

print(bytes.fromhex(get_flag(key[2:])))
```

```
b'crypto{50m3_p30pl3_d0n7_7h1nk_IV_15_1mp0r74n7_?}'
```

RSA

Starter

1. RSA Starter 1

In [1]:
```
print(pow(101,17,22663))
```

```
19906
```

2. RSA Starter 2

In [2]:
```
print(pow(12,65537,17*23))
```

```
301
```

In [ ]:
```
**3. RSA Starter 3**
```

In [3]:
```
print(1029224947942998075080348647218*857504083339712752489993810776)
```

```
882564595536224140639625987657529300394956519977044270821168
```

In [ ]:
```
**4. RSA Starter 4**
```

In [4]:
```
print(pow(65537,-1,1029224947942998075080348647218*857504083339712752489993810776))
```

```
121832886702415731577073962957377780195510499965398469843281
```

In [ ]:
```
**5. RSA Starter 5**
```

In [5]:
```
print(pow(77578995801157823671636298847186723593814843845525223303932,121832886702415731577073962957377780195510
```

```
13371337
```

In [ ]:
```
**6. RSA Starter 6**
```

In [6]:
```
import hashlib

N = 15216583654836731327639981224133918855895948374072384050848479908982286890731769486609085918857664046075375
d = 11175901210643014262548222473449533091378848269490518850474399681690547281665059317155831692300453197335735

m="crypto{Immut4ble_m3ssag1ng}"

sha256_hash = hashlib.sha256(m.encode()).hexdigest()

# Convert the hash to an integer
sha256_int = int(sha256_hash, 16)

signed = pow(sha256_int,d,N)
```

```
print(signed)
```

```
1348073840459009080333983164923845437618318974497068312990976607887770658328242268671054521727579737670967235889
4231550335007974983458408620258478729775647818876610072903021235573923300070103666940534047644900475773318682585
7726981556174514774484411981507104208189953472359211118120686567829981680649609654517194910725690576367011904297
6004719326188609286202411848782645276651353386073472412422830515891422525048839967364573288207757525266246186097
2889771112594906884441454355959482925283992539925713424132009768721389828848907099772040836383856524605008942907
0834903831097574069405408669782374716862966616858390834 75
```

PRIMES PART 1

1. Factoring

    Σχόλιο:

    Από το εργαλείο που μας δίνει η εκφώνηση έχουμε

    510143758735509025530880200653196460532653147 = 19704762736204164635843 ·
    25889363174021185185929

In [7]:
```python
if 19704762736204164635843<25889363174021185185929:
    print(19704762736204164635843)
else:
    print(25889363174021185185929)
```

```
19704762736204164635843
```

2. Monoprime

In [8]:
```python
from Crypto.Util.number import inverse, long_to_bytes

n = 171731371218065444125482536302245915415603318380280392385291836472299752747934607246477508507827284075763910
e = 65537
ct = 161367550346730604451454756189028938964941280347662098798775466019463375610700074840105776873791605070092 5

p=n-1
d=inverse(e,p)

decrypted_m = pow(ct,d,n)

print(long_to_bytes(decrypted_m))
```

```
b'crypto{0n3_pr1m3_41n7_pr1m3_l0l}'
```

In [ ]:
```
**3. Manyprime**
```

In [9]:
```python
from Crypto.Util.number import inverse, long_to_bytes

n = 580642391898843192929563856870897799650883152718761762932292482252152591279871421569162037190419036435041797
e = 65537
ct = 320721490534624434149993723527322977960556510750628354856260732098109692581338409999983376131354918370047 6

p=int("580642 391898 843191 487404 652150 193463 439642 600155 214610 402815 446275 117822 457602 964108 279991

d=inverse(e,p)

decrypted_m = pow(ct,d,n)

print(long_to_bytes(decrypted_m))
```

```
b'crypto{700_m4ny_5m4ll_f4c70r5}'
```

In [ ]:
```
**4. Salty**
```

In [10]:
```python
from Crypto.Util.number import getPrime, inverse, bytes_to_long, long_to_bytes


n = 110581795715958566206600392161360212579669637391437097703685154237017351570464767725324182051199901920318211
e = 1
ct = 44981230718212183604274785925793145442655465025264554046028251311164494127485

pt = pow(ct, 1, n)
decrypted = long_to_bytes(pt)

print(decrypted)
```

```
b'crypto{saltstack_fell_for_this!}'
```

5. Modulus Inutilis

In [11]:
```python
from Crypto.Util.number import long_to_bytes
```

```python
from gmpy2 import iroot


n = 172582129161919485363485484709380042442695445600390092447219592935548224980470754036584298652018163633118058
e = 3
ct = 2432510536179037603099418448354112923733506559730754802640013529198651801512221898204733584110377593813286

#pt = int(pow(ct, 1/3)) Δεν βγάζει καλή προσέγγιση


decrypted = long_to_bytes(iroot(ct,3)[0])

print(decrypted)
```

```
b'crypto{N33d_m04R_p4dd1ng}'
```

DIFFIE-HELLMAN

STARTER

1. Diffie-Hellman Starter 1

In [12]:
```python
g = 209
p = 991
fc = 1

x=1
while x!=p and (g * x) % p != fc:
    x+=1
if (g * x) % p == fc:
    print(x)
```

```
569
```

In [ ]:
```
**2. Diffie-Hellman Starter 2**
```

In [13]:
```python
p = 28151

number=1

for i in range(p):
    for n in range(p):
       if number%p==i^n:
        print(i)
       else:
         number+=1
```

```
1
```

ELLIPTIC CURVES

BACKGROUND

1. Background Reading

FLAG = Abelian group

2. Point Negation

In [15]:
```python
from Crypto.Util.number import inverse

a = 497
b = 1768
p = 9739

P = (493, 5564)
Q = (1539, 4742)
R = (4403,5202)

def point_addition(P, Q):
    O = (0, 0)

    if P == O:
        return Q

    if Q == O:
```

```
        return P

    x1, y1 = P[0], P[1]
    x2, y2 = Q[0], Q[1]

    if x1 == x2 and y1 == -y2:
        return 0

    if P != Q:
        lam = ((y2 - y1) * inverse(x2 - x1, p)) % p
    else:
        lam = ((3 * x1**2 + a) * inverse(2 * y1, p)) % p

    x3 = (lam**2 - x1 - x2) % p
    y3 = (lam * (x1 - x3) - y1) % p

    summation = (x3, y3)

    return summation

S = point_addition(point_addition(point_addition(P, P), Q), R)
print(S)
```

```
(4215, 2162)
```

HASH FUNCTIONS

PROBABILITY

1. Jack's Birthday Hash

In [18]:
```
n = 2 ** 11

P = 1

for i in range(1, n):
    P = (1 - 1/n) ** i
    nP = 1 - P

    if nP > 0.5:
        print(i)
        break
```

```
1420
```

2. Jack's Birthday Confusion

In [20]:
```
from math import factorial

n = 2048

for i in range(n):
    probability = 1 - factorial(n) / (factorial(n - i) * pow(n, i))

    if probability > 0.75:
        print(i)
        break
```

```
76
```

In [ ]:

Loading [MathJax]/extensions/Safe.js