UNIVERSIDAD CENTRAL DEL ECUADOR

COMPUTACIÓN

OCTAVO SEMESTRE

CRIPTOGRAFÍA Y SEGURIDAD DE LA INFORMACIÓN

TEMA: Plataforma Cryptohack

Integrantes:

- Arias Basantes Joffre David
- Fiallos Checa Fátima Carolina
- Flores Armijo Byron Rigoberto
- Hurtado Tinoco Kevin David
- Lechon Lechon Cristian Alexander
- Pila Aguaisa Jordi Fernando
- Pujota Pineda Angelo Fabricio
- Tipán López Edgar Vinicio

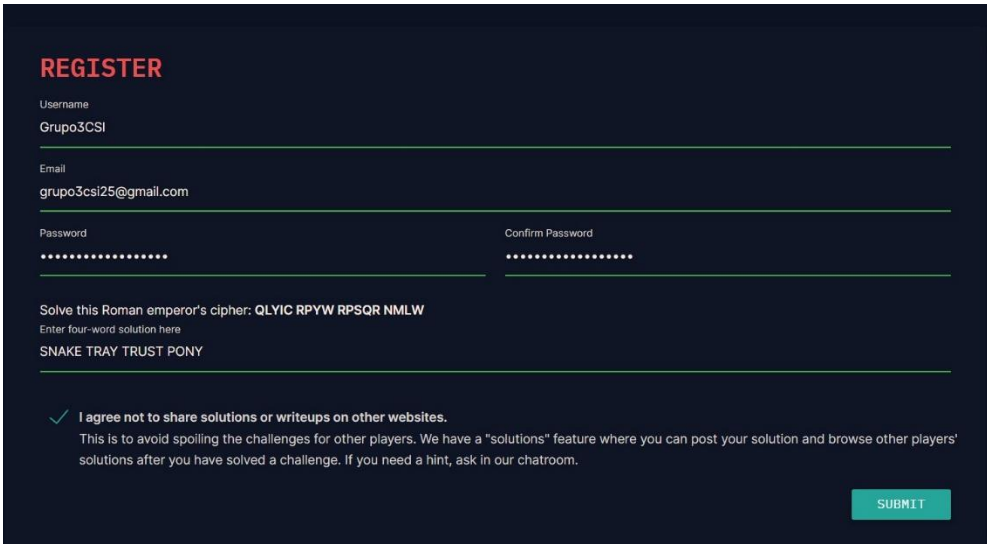
16/06/2025

Reto 1: La cuenta de correo electrónico del grupo

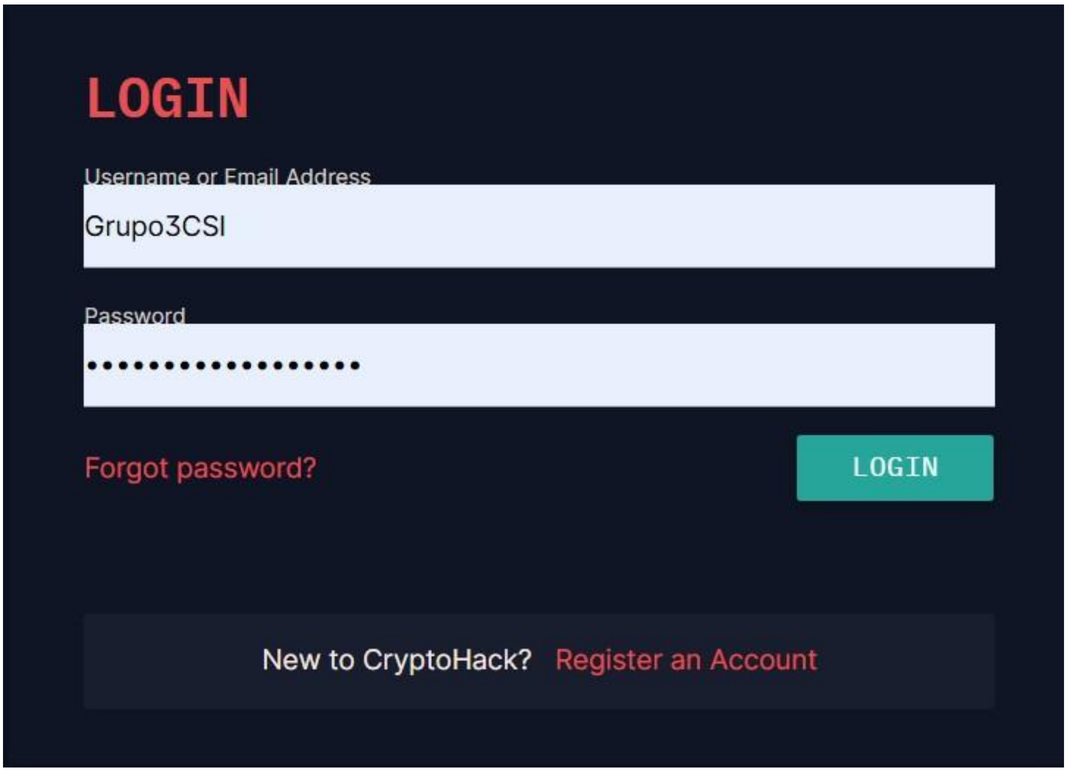
- Contraseña segura validada en Kaspersky

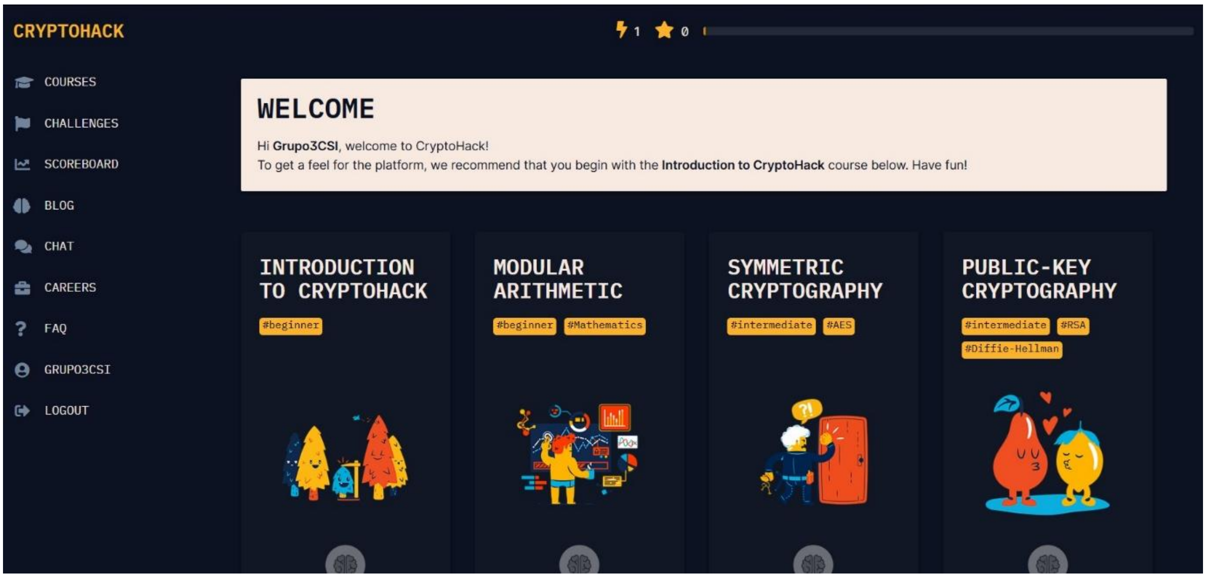
- Creación del correo electrónico del Grupo

- Creación de cuenta en Cryptohack.org



- Inicio de sesión en la cuenta


LOGIN
Username or Email Address
Grupo3CSI
Password
Forgot password?
LOGIN
New to CryptoHack? Register an Account


CRYPTOHACK
1
0
COURSES
CHALLENGES
SCOREBOARD
BLOG
CHAT
CAREERS
FAQ
GRUPO3CSI
LOGOUT
WELCOME
Hi Grupo3CSI, welcome to CryptoHack!
To get a feel for the platform, we recommend that you begin with the Introduction to CryptoHack course below. Have fun!
INTRODUCTION TO CRYPTOHACK
#beginner
MODULAR ARITHMETIC
#beginner #Mathematics
SYMMETRIC CRYPTOGRAPHY
#intermediate #AES
PUBLIC-KEY CRYPTOGRAPHY
#intermediate #RSA #Diffie-Hellman

Reto 2: Mathematics

MATEMÁTICAS MODULARES Alternar

Residuos cuadráticos — 25 puntos · 15306 Resuelve · 62 Soluciones

Hemos visto la multiplicación y la división en aritmética modular, pero ¿qué significa tomar la raíz cuadrada módulo como un número entero?

Para la siguiente discusión, trabajemos módulo $p = 29$. Podemos tomar el número entero $un = 11$ y calcular $un^2 = 5 \mod 29$.

Como $un = 11$ años, $un^2 = 5$, decimos que la raíz cuadrada de 5 es 11.

Esto se siente bien, pero ahora pensemos en la raíz cuadrada de 18. A partir de lo anterior, sabemos que necesitamos encontrar algún número entero un de tal manera que $un^2 = 18$

Tu primera idea podría ser empezar con $un = 1$ y bucle a $un = p - 1$. En esta discusión p no es demasiado grande y podemos comprobar rápidamente todas las opciones.

Pruébalo, intenta codificar esto y mira lo que encuentras. Si lo has codificado correctamente, lo encontrarás para todos $una \in \mathbb{F}_p^*$ nunca encuentras un un de tal manera que $un^2 = 18$.

Lo que estamos viendo, es que para los elementos de $\mathbb{F}_p^*$, no todos los elementos tienen una raíz cuadrada. De hecho, lo que encontramos es que para aproximadamente la mitad de los elementos de $\mathbb{F}_p^*$, no hay raíz cuadrada.

> Decimos que un número entero x es un *Residuo Cuadrático* si existe un un de tal manera que $un^2 \equiv x \mod p$. Si no existe tal solución, entonces el número entero es un *no residuo cuadrático.*

En otras palabras, x es un residuo cuadrático cuando es posible sacar la raíz cuadrada de x modulo an integer p.

En la siguiente lista hay dos residuos no cuadráticos y un residuo cuadrático.

Encuentra el residuo cuadrático y luego calcula su raíz cuadrada. De las dos raíces posibles, envíe la más pequeña como bandera.

El problema del ejercicio es de encontrar el residuo cuadrático y luego calcular su raíz cuadrada, envié la más pequeña como bandera.

Datos:

$$p = 29 \quad ints = [14, 6, 11]$$

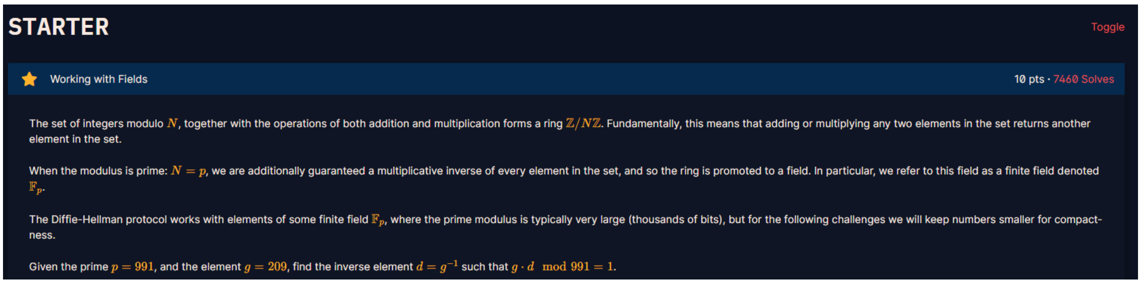
Para resolver este ejercicio se creó un pequeño algoritmo usando Python.

```
# Definir el número primo módulo
p = 29
# Lista de números para los que queremos saber si son residuos cuadráticos
ints = [14, 6, 11]
# Lista para guardar los posibles valores cuya raíz cuadrada al módulo p da uno de los números anteriores
raices_cuadradas = []
# Revisamos cada número 'a' desde 0 hasta p - 1
for a in range(p):
    # Calculamos el cuadrado de 'a' módulo p
    cuadrado_mod_p = (a * a) % p

    # Si ese resultado está en la lista de números, lo guardamos
    if cuadrado_mod_p in ints:
        raices_cuadradas.append(a)
# Imprimimos el menor valor encontrado (la raíz cuadrada más pequeña)
if raices_cuadradas:
    print("bandera:", min(raices_cuadradas))
else:
    print("No se encontró ninguna raíz cuadrada.")
```

```
bandera: 8
```

Reto 3: Diffie-Hellman



STARTER Toggle

Working with Fields — 10 pts · 7460 Solves

The set of integers modulo N, together with the operations of both addition and multiplication forms a ring $\mathbb{Z}/N\mathbb{Z}$. Fundamentally, this means that adding or multiplying any two elements in the set returns another element in the set.

When the modulus is prime: $N = p$, we are additionally guaranteed a multiplicative inverse of every element in the set, and so the ring is promoted to a field. In particular, we refer to this field as a finite field denoted $\mathbb{F}_p$.

The Diffie-Hellman protocol works with elements of some finite field $\mathbb{F}_p$, where the prime modulus is typically very large (thousands of bits), but for the following challenges we will keep numbers smaller for compact-ness.

Given the prime $p = 991$, and the element $g = 209$, find the inverse element $d = g^{-1}$ such that $g \cdot d \mod 991 = 1$.

El problema consiste en encontrar el inverso multiplicativo de un número en un campo finito.

¿Qué es un inverso multiplicativo?

Dado un número g en un campo finito F_p, el inverso multiplicativo de g, denotado d, es el número tal que:

$$g \cdot d \equiv 1 \, mod \, p$$

Datos del problema:

1. p = 991 (es primo, así que estamos en un campo finito F_{991})
2. g = 209

- **Paso 1:**

Aplicar el algoritmo extendido de Euclides que calcula el MCM para verificar que existe el inverso multiplicativo de:

$$MCM\ (991, 209)$$

$991 = 4 \times 209 + 155$

$209 = 1 \times 155 + 54$

$155 = 2 \times 54 + 47$

$54 = 1 \times 47 + 7$

$47 = 6 \times 7 + 5$

$7 = 1 \times 5 + 2$

$5 = 2 \times 2 + 1$

$2 = 2 \times 1 + 0$

Como llegamos a 1, significa que 209 y 991 son coprimos y el inverso existe

- **Paso 2:**

Ahora vamos hacia atrás para expersar 1 como combinación lineal de 209 y 991

$1 = 5 - 2 \times 2$

$1 = 5 - 2 \times (7 - 1 \times 5) = 3 \times 5 - 2 \times 7$

$1 = 3 \times (47 - 6 \times 7) - 2 \times 7 = 3 \times 47 - 20 \times 7$

$$1 = 3 \times 47 - 20 \times (54 - 47) = 23 \times 47 - 20 \times 54$$

$$1 = 23 \times (155 - 2 \times 54) - 20 \times 54 = 23 \times 155 - 66 \times 54$$

$$1 = 23 \times 155 - 66 \times (209 - 1 \times 155) = 89 \times 155 - 66 \times 209$$

$$1 = 89 \times (991 - 4 \times 209) - 66 \times 209 = 89 \times 991 - 422 \times 209$$

Entonces:

$$1 = 89 \cdot 991 - 422 \cdot 209$$

Pasamos a la forma:

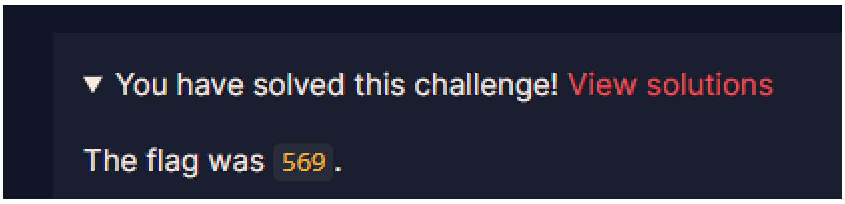
$$-422 \cdot 209 \equiv 1 \; mod \; 991$$

Y como estamos en módulo 991, el inverso de 209 es:

$$d = -422 \; mod \; 991$$

Para convertir un número negativo al módulo, simplemente sumamos el módulo:

$$-422 + 991 = 569$$



Reto 4: Hash Functions

El cumpleaños de Jack Hash

La función hash JACK11 produce una salida determinista de 11 bits (es decir, una cadena binaria de 11 posiciones), y queremos determinar cuántos secretos únicos se necesitan, en promedio, para tener una probabilidad del 50% de que uno de ellos tenga el mismo hash que el secreto de Jack.

1.Posibles Salidas

Para calcular las posibles conocemos que un bit puede tomar valores entre 0 y 1, además siempre obtendremos una salida de 11 bits por tanto tenemos:

$2^{11} = 2048$ hashes posibles

2. Probabilidad de colisión

Para que un secreto tenga el mismo hash que el de Jack, hay una probabilidad es:

$$\frac{1}{2048}$$

Por tanto, la probabilidad de que un único secreto no colisione es de:

$$\left(1 - \frac{1}{2048}\right) = \frac{2047}{2048}$$

Si probamos k secretos distintos (independientes), la probabilidad de que **ninguno** colisione es:

$$P\ (no\ colisión) = \left(1 - \frac{1}{2048}\right)^k$$

Queremos que la probabilidad de al menos una colisión sea del 50%, es decir:

$$P(colisión) = 1 - \left(1 - \frac{1}{2048}\right)^k = 0.5$$

o

$$P\ (colisión) = \left(\frac{2047}{2048}\right)^k = 0.5$$

3.Despejamos k

Aplicamos logaritmo donde $a^k = x \Rightarrow k = \log_a(x)$, donde:

$$a = \frac{2047}{2048}\ ;\ x = 0.5$$

Entonces,

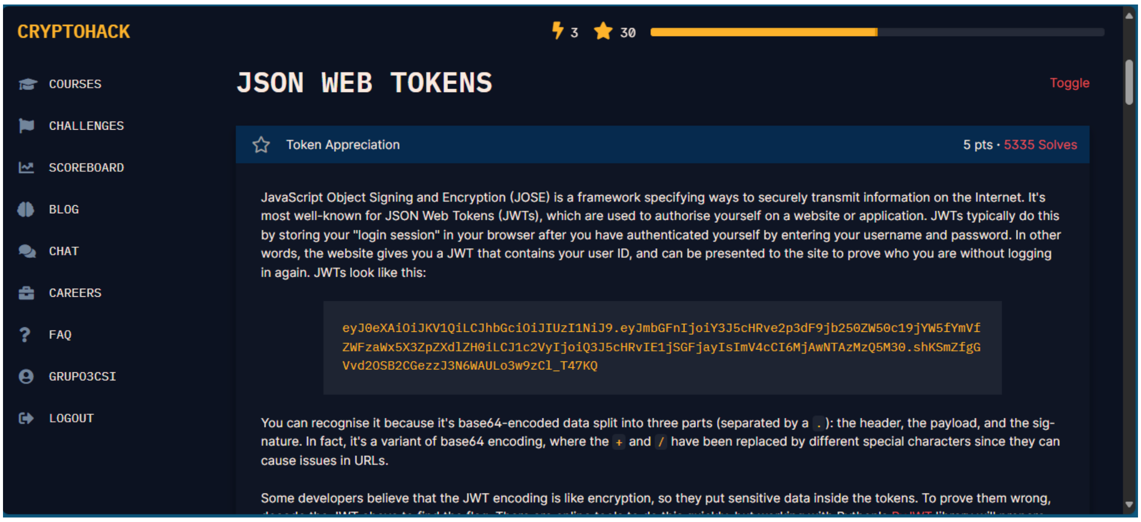
$$k = log_{\frac{2047}{2048}}(0.5)$$

$$\therefore\ K = 1419.21882 \ldots \approx 1420$$

Por lo que necesitamos al menos 1420 secretos únicos, para tener una **probabilidad del 50%** de colisionar con el hash del secreto de Jack usando la función **JACK11**.

Reto 5: Crypto on the web

JSON Web Tokens – Token Appreciation



Para resolver el reto se usó una página online de referencia que da el ejercicio [Welcome to PyJWT — PyJWT 2.10.1 documentation]() que proporciona el código ejemplo para poder descodificar el texto adjunto modificando para que se adapte al texto:

```python
# filepath: c:\Users\fcaro\proyectos cripto\JSON_Web_Tokens.py
import jwt

token = "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJmbGFnIjoiY3J5cHRve2p3dF9jb250ZW50c19jYW5fYmVfZWFzaWx5X3ZpZXd

decoded = jwt.decode(token, options={"verify_signature": False})
print(decoded)
```

```
PS C:\Users\fcaro\proyectos cripto> & C:/Users/fcaro/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/fcaro/proyectos cripto/JSON_W
eb_Tokens.py"
{'flag': 'crypto{jwt_contents_can_be_easily_viewed}', 'user': 'Crypto McHack', 'exp': 2005033493}
PS C:\Users\fcaro\proyectos cripto>
```

Los JWT tienen tres partes:

1. **Header** – indica el tipo de token y el algoritmo usado (como HMAC-SHA256).
2. **Payload** – contiene la información útil, como el nombre de usuario, tiempo de expiración.
3. **Signature** – una firma digital generada con una clave secreta.

Siendo la respuesta correcta: `crypto{jwt_contents_can_be_easily_viewed}` que es flag.

Reto 6: Lattices

Gaussian Reduction

Para resolver este reto, nos guiaremos en el algoritmo proporcionado por el ejercicio en cuestión, además de apoyarnos en un archivo escrito en lenguaje Python en Google Colab.

El ejercicio consiste en recrear el algoritmo de reducción Gaussiana, que recibe dos vectores base (trabajamos en la 2da dimensión). Se sustrae una combinación lineal del primer vector al segundo, hasta que sea imposible hacerlos más pequeños.

Código en Google Colab:

1. Primero, creamos una función para el cálculo del producto punto entre vectores

```python
def producto_punto(vector1, vector2, length):
  # Comprobacion en caso de que la longitud de ambos vectores no sea la misma
    if len(vector1) != length or len(vector2) != length:
        raise ValueError("Los vectores deben tener la misma longitud.")

  # Comprobacion en caso de que la longitud de ambos vectores sea cero
    if length == 0:
        return 0

  # En caso de que los vectores sean apropiados, se prosigue
    suma_productos = 0
    for i in range(length):
        suma_productos += (vector1[i] * vector2[i])
    return suma_productos
```

2. Creamos el algoritmo, basándonos en la imagen proporcionada en el reto

```python
def reduccion_gaussiana_latices(b1, b2):
    if len(b1) != 2 or len(b2) != 2:
        raise ValueError("Los vectores de la base deben ser de dimensión 2 para este ejercicio")

    # Hacemos copias de los vectores para no modificar los originales
    copia_b1 = list(b1)
    copia_b2 = list(b2)

    # Calculamos la longitud de los vectores al cuadrado para comparaciones
    # Aqui usamos la funcion creada previamente, ya que ||v|| = v . v
    long_cuad_b1 = producto_punto(copia_b1, copia_b1, 2)
    long_cuad_b2 = producto_punto(copia_b2, copia_b2, 2)

    # Comprobacion en caso de que b1 sea cero
    if long_cuad_b1 == 0:
        if long_cuad_b2 == 0:
            return [0, 0], [0, 0]
        else:
            # Por si b1 es cero, pero b2 no
            return [0, 0], copia_b2

    # Algoritmo principal
    while True:
        # 1. Aseguramos que ||b1|| <= ||b2||
        if long_cuad_b2 < long_cuad_b1:
            copia_b1, copia_b2 = copia_b2, copia_b1
            long_cuad_b1, long_cuad_b2 = long_cuad_b2, long_cuad_b1 # Actualizar las longitudes al cuadrado

        # 2. Calculamos el coeficiente mu (coeficiente de Gram-Schmidt)
        # mu = (b1 . b2) / (b1 . b1)
        # Redondeamos al entero más cercano (floor es el mas comun para estos algoritmos)
        pro_punto_b1_b2 = producto_punto(copia_b1, copia_b2, 2)
        mu = round(pro_punto_b1_b2 / long_cuad_b1)

        # 3. Si mu es cero, no se puede reducir más b2 respecto a b1
        if mu == 0:
            break

        # 4. Reducimos b2: b2 = b2 - mu * b1
        for i in range(2):
            copia_b2[i] -= mu * copia_b1[i]

        # Actualizamos la longitud al cuadrado de b2 después de la reducción
        len_sq_b2 = producto_punto(copia_b2, copia_b2, 2)

    return copia_b1, copia_b2
```

3. Comprobamos la funcionalidad del algoritmo, y reducimos los vectores

```python
u = [87502093, 123094980]
v = [846835985, 9834798552]
x,y = reduccion_gaussiana_latices(u,v)
print(f'Vectores luego de la reducción: {x}, {y}')
print(f'Reducción final: {producto_punto(x,y,len(x))}')
```

```
Vectores luego de la reducción: [87502093, 123094980], [-4053281223, 2941479672]
Reducción final: 7410790865146821
```

Reto 7: ZKPS

Honest Verifier Zero Knowledge

En este desafío se busca demostrar la propiedad de Zero Knowledge (ZK) para un verificador honesto (HVZK), mediante la construcción de un simulador que pueda generar un transcripto válido sin conocer el testigo (w). Este es un concepto clave en las pruebas de conocimiento cero, en el cual un probador (P) puede convencer a un verificador (V) de que conoce un testigo sin revelar nada sobre dicho testigo.

Pasos para la resolución:

- **Paso 1: Generar z aleatorio en $F_q^{\cdot}$**

El primer paso consiste en generar un valor aleatorio dentro del conjunto, es decir, los elementos no nulos del grupo. Este valor se utiliza para simular la interacción del probador con el verificador. La razón de generar un valor aleatorio es que el simulador no tiene acceso al testigo por lo que debe crear un valor aleatorio para continuar con la construcción del transcripto sin necesidad de conocer el testigo.

- **Paso 2: Generar un reto aleatorio e**

El siguiente paso es generar un reto e de forma aleatoria, tomando un valor del espacio del reto. En un protocolo de ZK real, el reto es enviado por el verificador después de recibir el valor a, pero en este caso, el simulador puede generar el reto después de conocer el valor z, ya que no hay dependencia del testigo en este paso.

- **Paso 3: Calcular *a*:**

En este paso, el simulador debe calcular a usando la fórmula $y = g^z \, mod \, p$ donde g es el generador y p es el módulo primo dado en el sistema. Esta es la parte crítica del simulador, ya que el valor de la debe ser generado de manera que el verificador acepte el transcripto como válido en un protocolo real. Este cálculo se basa en el valor aleatorio z, sin necesidad de conocer el testigo w.

Paso 4: Generar el transcripto (a,e,z)

Finalmente, el simulador genera el transcripto (a, e, z) con los valores calculados en los pasos anteriores. Este transcripto es indistinguible de uno que podría ser generado en una interacción real entre un probador y un verificador, lo que asegura que el verificador no aprende nada sobre el testigo w, ya que el simulador no tiene acceso a este testigo. El verificador acepta el transcripto como si fuera parte de una interacción legítima con el probador.

Para este desafío se desarrolló un pequeño programa en Python, en Google Colab:

```python
import random

# Parámetros del sistema (p, q, g, y)
p = 23  #módulo primo
q = 11  # orden
g = 5   # generador
y = 8   # Valor público para y = g^w mod p

# Paso 1: Generar aleatoriamente z en F_q^* (elementos no nulos de Z_q)
z = random.randint(1, q-1)

# Paso 2: Generar el reto e aleatorio
e = random.randint(1, q-1)

# Paso 3: Calcular a = g^z mod p
a = pow(g, z, p)

# Paso 4: Devolver el transcripto (a, e, z)
print(f"Transcripto generado: a = {a}, e = {e}, z = {z}")
```

Y esto nos dio como resultado:

```
Transcripto generado: a = 2, e = 10, z = 2
```

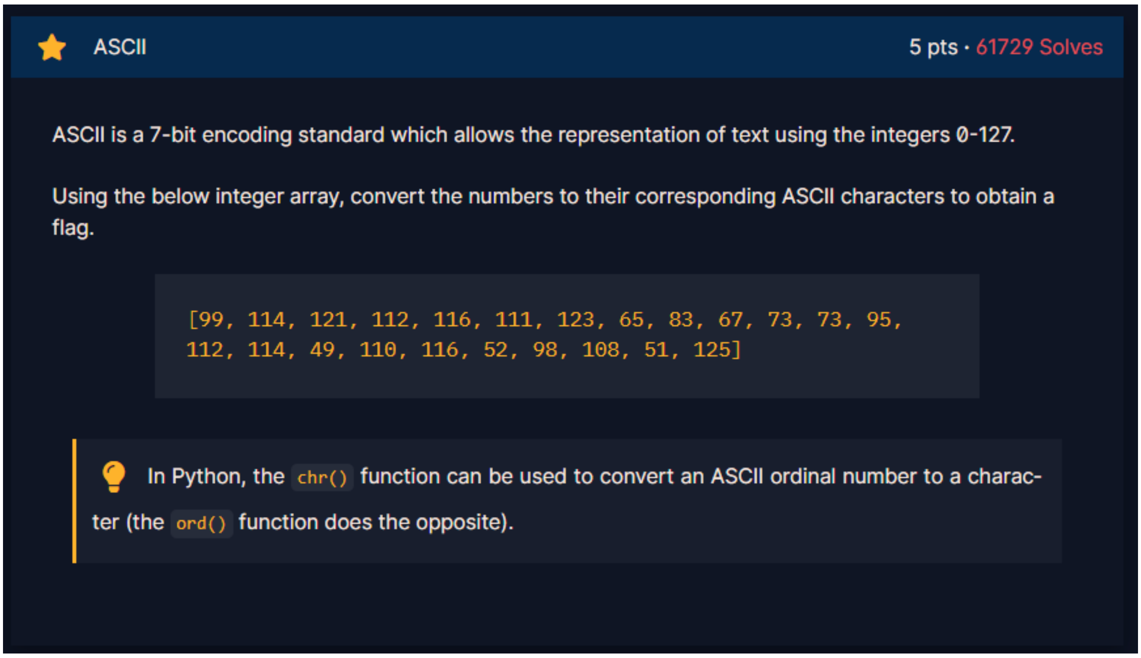
El desafío ha sido resuelto con éxito generando un transcripto válido (a, e, z) mediante un simulador. Este transcripto es indistinguible de uno generado en una interacción real entre un probador y un verificador, demostrando que el sistema cumple con la propiedad de Zero Knowledge para un Verificador Honesto (HVZK). Este enfoque asegura que el verificador no obtiene ninguna información adicional sobre el testigo \(w \), cumpliendo con la propiedad de no revelar nada sobre el testigo mientras se demuestra su conocimiento.

Reto 8: ASCII

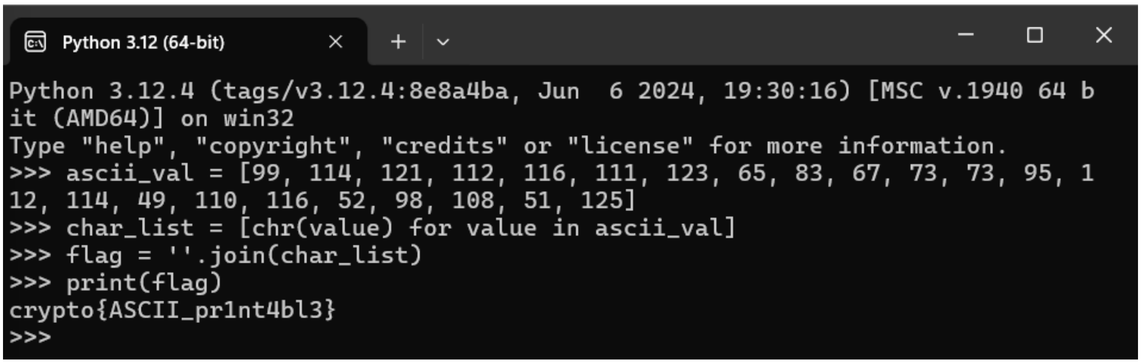
ASCII es un estándar de codificación de 7 bits que permite la representación de texto mediante los números enteros del 0 al 127. Con la matriz de números enteros que se muestra a continuación, convierta los números a sus caracteres ASCII correspondientes para obtener un indicador.

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

En Python, la `chr()` función se puede utilizar para convertir un número ordinal ASCII en un carácter (la `ord()` función hace lo contrario).



Para resolver el ejercicio realizamos un pequeño script en Python:



Primero, se define una lista llamada `ascii_val`, que contiene números que corresponden a caracteres como letras y símbolos.

Luego, mediante una comprensión de lista, se convierte cada número en su carácter correspondiente utilizando la función `chr()`.

Estos caracteres se almacenan en la lista `char_list`.

Después, la función `.join(char_list)` se usa para unir todos los caracteres en una sola cadena.

Finalmente, el código imprime el resultado, que es el texto "`crypto{ASCII_pr1nt4bl3}`".