Universidad Nacional de Córdoba
Diplomatura Universitaria en Ciberseguridad

Criptografía Aplicada
Trabajo final

Sofía Luján

Introducción

La asignatura Criptografía Aplicada dictada en la Diplomatura de Ciberseguridad de la Universidad Nacional de Córdoba propuso como trabajo final resolver desafíos criptográficos. Estos desafíos muestran las características y posibles dificultades de las diferentes técnicas criptográficas que se han analizado en el curso. El objetivo del trabajo final es resolver y documentar la resolución de los desafíos, en donde se debe describir y analizar el problema, demostrar la solución y las diferentes maneras que se puede evitar.

Desafíos

Aleatoriedad

Generador de números aleatorios de Java

El desafío consiste en adivinar el siguiente número de 32 bits a ser producido por un generador de números pseudoaleatorios compatibles con el utilizado en Java.
En Java, se emplea un método llamado "generador congruencial lineal" para generar secuencias de números pseudoaleatorios. Este método se basa en una fórmula recurrente en la que cada número de la secuencia se calcula multiplicando el número anterior por una constante (conocida como multiplicador), sumándole otra constante (el incremento) y luego tomando el residuo cuando se divide por una tercera constante (el módulo). El valor de partida para esta secuencia se denomina "semilla" o "seed".

```
public static int next(long seed) {
  long seed2 = (seed * multiplier + addend) & mask;
  return (int)(seed2 >>> 16);
}
```

En donde las constantes tienen los siguientes valores:

```
private static final long multiplier = 0x5DEECE66DL;
private static final long addend = 0xBL;
private static final long mask = (1L << 48) - 1;
```

Al hacer dos llamadas al servidor para obtener dos números consecutivos pseudo aleatorios:

```
List<Long> lista = new ArrayList<>();
lista.add(-515911456L);
lista.add(-1911918238L);
```

Para obtener el primer número, el algoritmo generador de números aleatorios aplicó la operación `(seed * multiplier +addend) & mask` y definio una nueva semilla. Sobre la nueva semilla realizó la operación `seed2 >>> 16 (`desplazamiento de bits) para obtener el número pseudo aleatorio *-515911456.* En binario es:

```
new seed    0000 0000 0000 0000 1110 0001 0011 1111 1101 0000 1110 0000 xxxx xxxx xxxx xxxx
next number 0000 0000 0000 0000 0000 0000 0000 0000 1110 0001 0011 1111 1101 0000 1110 0000  -515911456
```

Como conocemos parte de la nueva semilla, específicamente los 48 bits menos significativos, puedo determinar el valor completo de la semilla. Simplemente necesitamos probar todos los valores posibles en un rango de 0 a 65535 hasta encontrar el valor que, cuando se utiliza en el generador de números aleatorios, produce el siguiente número conocido, que en este caso es -1911918238.

```
long seed = 0L;
for (int i = 0; i < 65536; i++) {
  seed = lista.get(0) * 65536 + i;
  if (next(seed) == lista.get(1).intValue()) {
    System.out.println("\nSeed: " + seed);
```

```
    break;
  }
}
```

Una vez obtenida la semilla, podemos calcular los siguientes números aleatorios:

```
Random random1 = new Random((seed ^ multiplier) & mask);
int primerNro = random1.nextInt();
int segundoNro = random1.nextInt();
int terceroNro = random1.nextInt();
System.out.printf("PrimerNro: %d \nSegundo nro: %d \nTercer nro: %d", primerNro,
segundoNro, terceroNro);
```

El problema del generador de números aleatorios de Java es que para calcular la siguiente semilla y número utiliza todos valores constantes, lo que genera una alta correlación. Es necesario variables externas impredecibles para que realmente sea una generados de números aleatorios.

Clave generada a partir de la fecha y hora

El desafío consiste en descifrar un mensaje cifrado con una clave generada a partir de la fecha y hora de creación del mensaje. Para obtener el mensaje encriptado se realiza una request GET al servidor:

```
email = 'solujan@gmail.com'
url_request =
f'https://ciberseguridad.diplomatura.unc.edu.ar/cripto/timerand/{email}/challenge'
# Solicitud
request = requests.get(url_request)
# Body de la respuesta
texto_enc_completo = request.content
```

El siguiente es el mensaje obtenido:

From: User user@example.com

Date: Sun Apr 3 16:31:19 UTC 2022

To: solujan@gmail.com

IsG8XTnwCS4+NSi7XN5yudL3r9D2f6WLbTRM0HPTGeumxAvNzlysoBzoIbFP8dNRLv3SkTNdQ8NuKHEIcycAPrN1d0UikoM1W1mpME081zxdW
GmDPBkIWvOpThust+kI27NFeh8beZEoY464zCS8NJXSOqAjr6mXcoV7sDU9BN87bQgrTZpA5Foq1iKk+XGeCJ2QbwizKIpY5/kqZAcDhd6TA5lNt
pMDROifkbAO2xN/iSXAaX5LHOe51ZmKgdI7yKsIfFiwY93wsVzbAagoz2VOBm2h2vdh391YJ0BWbu3LNvHQKgDToDOIbctJq0PhlnSMPJTlcBjk2
9CvaXUBKb6evg2wt+7piHq6xjL3tZUJ8HdKMTw7J4Btmi4M8YdzDCuwEuf0lY4D8sFlvCyUjGPz5OOqutWazVL4DXp1vXKsBqJQaGrn2mnT/nX45
DuNzq+Crjg7RQfrs9doTTdEyAImHd5ZcQsma9TJ9vMxRJ4k2YnqlcjxuHkNIyh4aUtj7d1qQTQ0ldYUof5q0bbJ7rcY5THNkwFlGdLhbhDK+sgZ6Px+
RbWX7SS/GGGfaYtn

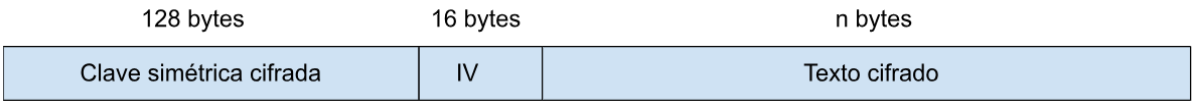
En el encabezado, además del emisor y el destinatario, aparece la hora de creación del mensaje. El cuerpo contiene una secuencia de bytes codificada en base64.
La información que es necesaria del mensaje es el cuerpo del mensaje y la fecha en que fue creado el mensaje:

```
byte_list = texto_enc_completo.split(b'\n')
fecha = byte_list[1]
message_body = byte_list[4]
```

En la variable `message_body` tengo el contenido del mensaje codificado en base64 por lo que es necesario decodificarlo para obtener una cadena de bytes:

```
texto_encriptado_bytes = base64.b64decode(message_body)
```

Los primeros 128 bytes de esa cadena contienen una clave simétrica que ha sido cifrada con RSA de 1024 bits. El resto de la cadena, es el mensaje cifrado con AES-128 en modo CBC, utilizando relleno PKCS7, y con el IV colocado como primer bloque:



Divididos la cadena de bytes para solo quedarnos con la clave simétrica cifrada:

```
clave_simetrica_cifrada = texto_encriptado_bytes[:128]
iv = texto_encriptado_bytes[128:128+16]
texto_cifrado = texto_encriptado_bytes[128+16:]
```

La clave simétrica ha sido generada mediante la aplicación del algoritmo MD5 a la hora de creación del mensaje, expresada como el tiempo Unix expresado en microsegundos (es decir, la cantidad de microsegundos segundos transcurridos desde las cero horas del 1 de enero de 1970 UTC).
La fecha y hora de la creación del mensaje es algo conocido ya que es devuelto por el servidor, por eso a continuación convierto la fecha y la hora a formato epoch:

```
# Crear un objeto struct_time con la fecha y hora en que se creo el mail
created_time= time.struct_time((2022, 4, 3, 16, 31, 19, 0, 0, 0))

# Convertir el objeto struct_time en un objeto datetime
datetime_obj = datetime.datetime.fromtimestamp(time.mktime(created_time))

# Obtener el valor de tiempo en segundos
timestamp_seconds = datetime_obj.timestamp()

# Convertir el valor de tiempo a microsegundos
timestamp_microseconds = int(timestamp_seconds * 1000000)
```

Es necesario observar que la fecha y hora que figuran en el mensaje se encuentran expresadas con una precisión de segundos, no de microsegundos, por lo que existen 1.000.000 de claves posibles asociadas con ella.
El valor de timestamp_microseconds es de 1649003479000000. A través de fuerza bruta, voy a recorrer todos los valores posibles de 0 a 1000000 para utilizar ese valor como key_seed de MD5. Cuando pueda convertir el resultado en ASCII, quiere decir que obtuve el resultado correcto.

```
for i in range(1000000):

  now = timestamp_microseconds+i

  # Este es un mal mecanismo para generar una clave
  key_seed = now.to_bytes(8,"big")
  key = md5(key_seed).digest()

  cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default_backend())
  decryptor = cipher.decryptor()
  try:
    # Decrypt the data
    decrypted_data = decryptor.update(texto_cifrado) + decryptor.finalize()
    # Remove PKCS7 padding
    unpadder = symmetric_padding.PKCS7(128).unpadder()
    unpadded_data = unpadder.update(decrypted_data)
    unpadded_data += unpadder.finalize()
    decoded_text = unpadded_data.decode('ascii')
    print("Decrypted Data:", decoded_text)
    break
  except UnicodeDecodeError:
```

```
    pass
  except ValueError:
    pass
```

Resultado:

```
Decrypted Data: I lately lost a preposition; It hid, I thought, beneath my chair And angrily I cried, "Perdition!
Up from out of under there." Correctness is my vade mecum, And straggling phrases I abhor, And yet I wondered,
"What should he come Up from out of under for?"                  -- Morris Bishop
```

Para evitar este tipo de ataque es recomendado no utilizar informacion que se manda de forma plana en el cuerpo del mensaje, se podria utilizar un numero aleatorio para generar la key seed.

Cifrado de flujo

Secuencia cifrante repetida

En este desafío, el servidor proporciona 5 textos cifrados con un algoritmo de cifrado de flujo (no importa cuál).
Para obtener los 5 textos, se realiza una llamada GET al servidor:

```
message_encoded_in_base64_list = []
for index in range(0,5):
  url = 'https://ciberseguridad.diplomatura.unc.edu.ar/cripto/stream/solujan@gmail.com/challenge/'+str(index)
  request = requests.get(url)
  # Body de la respuesta
  message_encoded_in_base64_list.append(request.content)
```

y se obtienen los siguientes textos cifrantes:

```
[b'K7bkhpK7hTKaf1fF72wk4RKjUAOqiGZ0YhMs8IiRe51GQJ37KPoa',
b'P6n3koWqkySFYELQ+XI69QywTxO+nXJgcgw+4ZiGbohTVYPoOusK',
b'UHpfQoAnrJq2OPYzOWRz3+TB15PV+wIW52q6povLTRssWLJLjmdC',
b'gD96Hp0+gAi6QYaECUY5leY+8aPgIl7soUN7Qe9xn/ziUwrCj5jJ',
b'Q9WL7vnW71j5HD6shQ5GiXDMM2/C4Q4cDnBCneT6EvQvKf+URpd2']
```

Los textos cifrantes están codificados en base64, por lo que genero una lista de los mismos textos decodificados:

```
texto_encriptado_bytes_list = []
for msg in message_encoded_in_base64_list:
  texto_encriptado_bytes_list.append(base64.b64decode(msg))
```

Dichos textos cumplen con los siguientes caracteristicas:

- Tres están cifrados con la misma secuencia cifrante.
- Están compuestos por caracteres ASCII imprimibles, sin espacios en blanco ni fines de línea. Esto implica que sus códigos están entre 33 y 126 inclusive.
- Dos textos están compuestos por repeticiones de un mismo carácter. Por ejemplo, “AAAAAAAAAAAAAA” o “))))))))))))))))))))”
- Todos los textos tienen la misma longitud.
- Uno de los textos es la representación decimal de un número par. Por ejemplo “12345678”

Teniendo en cuenta que se utiliza un algoritmo de cifrado de flujo, entonces cada elemento individual del mensaje original (P) se encripta de manera secuencial utilizando el correspondiente elemento de la secuencia de claves (K), lo que produce un elemento cifrado en la secuencia de texto encriptado (C).

$$Ci = Ki \oplus Pi$$

También considero que la clave tiene la misma longitud del texto.
Al tener todas las secuencias cifrantes y conocer los patrones de los textos planos (un número par y 2 textos con letras repetidas) entoces se puede definir una posible K:

$$Ki = Ci \oplus Pi$$

La cantidad de secuencias de clave es la cantidad de letras posibles (sus códigos están entre los códigos 33 y 126 inclusive en ASCII) por la cantidad de textos cifrados que dio el servidor (en este caso es 5).
Mi solución tiene 2 loops anidados: el primer loop recorre las secuencias cifradas y el segundo loop los caracteres posibles, y de esta manera genero un posible k (xor entre secuencia cifrada y el posible texto plano)

```
for each in range(0, len(alt_texto_encriptado_bytes_list)):
  cifrado_i = alt_texto_encriptado_bytes_list[each]
  all_textos_encriptado_bytes_list = list(alt_texto_encriptado_bytes_list)

  for letra in range(33, 127):
    posible_key = bytearray()
    [posible_key.append(cifrado_i[i] ^ letra) for i in range(0,len(cifrado_i))]
```

Con la clave posible, obtengo el texto claro de las otras secuencias cifradas y verifico si se cumplen las condiciones definidas en el enunciado. Para ello, utilizo algunas funciones auxiliares que me permiten hacer operaciones repetitivas como el xor entre dos listas de bytes, validar si un texto está en ASCII imprimible, validar si en un texto se repite la misma letra y validar si un texto es un número y además par.

```
    validate_same_letter_int = 0
    validate_numero_par_int = 0
    validate_all_ascii_imprimible = 0
    for texto_encriptado_bytes in all_textos_encriptado_bytes_list:
      texto_claro = xor_between_list(posible_key, texto_encriptado_bytes)
      if is_validate_ascii_imprimible(texto_claro):
        validate_all_ascii_imprimible +=1
      if validate_same_letter(texto_claro):
        validate_same_letter_int += 1
      if validate_numero_y_par(texto_claro):
        validate_numero_par_int += 1
```

Si la clave posible genera textos claros que cumplen con todas las condiciones del enunciado, entonces la clave posible es una clave valida.

```
if validate_same_letter_int == 2 and validate_numero_par_int == 1 and
validate_all_ascii_imprimible >=3:
      for texto_encriptado_bytes in all_textos_encriptado_bytes_list:
        texto_claro = xor_between_list(posible_key, texto_encriptado_bytes)
        secuencia_cifrante_b64 = base64.b64encode(posible_key)
        print(f'secuencia_cifrante_b64 {secuencia_cifrante_b64}')
```

Y obtenemos los textos claros y la secuencia cifrante codificada en base64:

```
secuencia_cifrante_b64 b'GY/RtKOMtQKjRmT231Qc0yqWaTWYu1RGVCoYx76gSK51c6XOHM0s'
texto_claro: bytearray(b'29521700993308828596232269476133338547 6')

secuencia_cifrante_b64 b'GY/RtKOMtQKjRmT231Qc0yqWaTWYu1RGVCoYx76gSK51c6XOHM0s'
texto_claro: bytearray(b'&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&')
```

```
secuencia_cifrante_b64 b'GY/RtKOMtQKjRmT231Qc0yqWaTWYu1RGVCoYx76gSK51c6XOHM0s'
texto_claro:
bytearray(b'I\xf5\x8e\xf6#\xab\x19\x98\x15~\x92\xc5\xe60o\x0c\xceW\xbe\xa6M@VP\xb3@\xa2a5
k\x05\xb5Y+\x17\x85\x92\xaan')

secuencia_cifrante_b64 b'GY/RtKOMtQKjRmT231Qc0yqWaTWYu1RGVCoYx76gSK51c6XOHM0s'
texto_claro:
bytearray(b'\x99\xb0\xab\xaa>\xb25\n\x19\x07\xe2r\xd6\x12%F\xcc\xa8\x98\x96x\x99\n\xaa\xf
5ic\x86Q\xd1\xd7R\x97 \xaf\x0c\x93U\xe5')

secuencia_cifrante_b64 b'GY/RtKOMtQKjRmT231Qc0yqWaTWYu1RGVCoYx76gSK51c6XOHM0s'
texto_claro: bytearray(b'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ')
```

Utilizar la misma clave para cifrar textos planos no es una práctica recomendada ya que conociendo dos secuencias cifradas es posible obtener el texto plano. Por eso es recomendado utilizar un vector de inicialización (IV) que se combine con la clave para cifrar. Por es necesario que el IV sea lo suficientemente grande para que el mismo IV no se utilice dos veces en el mismo mensaje.

Cifrado de bloques

Falsificación en modo ECB

En este desafío, el servidor permite a un usuario registrarse con un correo electrónico, y devuelve un perfil, opcionalmente cifrado con AES en modo ECB. El objetivo del challenge es agregar un par role=admin a la query string que permita al atacante acceder a información restringida.
El cifrado por bloques AES en modo ECB se caracteriza por dividir el texto claro en bloques iguales (en este caso de 16 bytes), cada uno de estos es cifrado de manera separada con la misma clave. Es decir que bloques idénticos de texto claro producirán idénticos textos cifrados. Cuando el tamaño del texto claro es igual al tamaño del bloque, completa el bloque con relleno utilizando el padding estandarizado en PKCS7. En PKCS7 se aplican tantos bytes como sea necesario para que el mensaje sea múltiplo del tamaño de bloque, y el valor de esos bytes es la longitud del relleno. Si el mensaje original era múltiplo del tamaño de bloque, el relleño es un bloque adicional con todos sus bytes con el valor 16.
La metodología que voy a utilizar para resolver el challenge es hacer múltiples llamadas con distintos emails. El servidor me va a devolver los perfiles (textos claros conocidos) y los textos cifrados correspondiente. Divido cada texto cifrado en bloques de 16 bytes y los reordeno de manera tal que me permita obtener un texto cifrado con role=admin.
Para obtener el perfi, genero una función auxiliar:

```
def get_profile(email, email_register):
  url_request = f'...r/cripto/ecb-forge/{email}/register?email={email_register}'
  request = requests.get(url_request)
  message = request.content
  print_block(message.decode('ascii'), 16)
  return message
```

Si el usuario registró el correo solujan@gmail.com (email_register), el perfil estará compuesto por un conjunto de pares atributo-valor, de la forma =, y separados el carácter &, de la misma forma que en la query string de una URL:

```
user=solujan@gmail.com&id=547&role=user
```

En este caso, el único texto bajo nuestro control es el correo electrónico. Por lo que es necesario enviar distintas direcciones de email de manera tal que al reorder ciertos bloques obtenga el texto que yo necesito.
El primer email mandado `adminadnmi@bbb` y el objetivo es dejar `role=` al final del segundo bloque:

```
email = 'solujan@gmail.com'
email_register = 'adminadnmi@bbb'
message = get_profile(email, email_register)
```

El resultado obtenido es:

```
u s e r = a d m i n a d n m i @
b b b & i d = 9 2 1 & r o l e =
u s e r
```

El segundo email mandado es `adminadnmi@userbbbbbbbbbbbbadmin` y el objetivo es posicionar `admin` al comienzo del tercer bloque:

```
email_register_2 = "adminadnmi@userbbbbbbbbbbbbadmin"
message_2 = get_profile(email, email_register_2)
```

El resultado obtenido es:

```
u s e r = a d m i n a d n m i @
u s e r b b b b b b b b b b b b
a d m i n & i d = 1 3 6 & r o l
e = u s e r
```

El tercer email mandado es `adminadnmi@usebbbbbbbadmin` y el objetivo es generar un bloque de 48 bytes para que el algoritmo de cifrado genere un cuarto bloque que solo sea padding:

```
email_register_3 = "adminadnmi@usebbbbbbbadmin"
message_3 = get_profile(email, email_register_3)
```

```
u s e r = a d m i n a d n m i @
u s e b b b b b b b a d m i n &
i d = 4 4 3 & r o l e = u s e r
```

Ahora vamos a reordenar los bloques de 16 bytes, tomando los dos primeros bloques del primer perfil, para que `role=` quede posicionado al final del segundo bloque. Sumo el tercer bloque del segundo perfil obtenido, para que `admin` quede posicionado al comienzo del tercer bloque. Agrego un cuarto bloque con los primeros 16 bytes del primer perfil solo para completar bloque y el algoritmo de cifrado tenga que agregar todo un bloque de padding.

```
u s e r = a d m i n a d n m i @
b b b & i d = 9 2 1 & r o l e =
a d m i n & i d = 1 3 6 & r o l
u s e r = a d m i n a d n m i @
```

Si convierto todo a bytes, es necesario agregar el último bloque del tercer perfil obtenido que representa el padding:

```
admin_replace = message_hex[:64] + message_hex_2[64:64+32] + message_hex[:32] +
message_hex_3[64+32:]
```

```
08 25 42 43 e0 6b 80 de be f0 89 60 1d d6 88 83
e3 06 c7 19 e6 f4 64 0d c4 ee 01 2d 92 69 95 eb
ad 42 87 01 60 83 94 6a 71 eb f1 09 f6 d3 39 7e
08 25 42 43 e0 6b 80 de be f0 89 60 1d d6 88 83
38 16 9f a9 6e 5f 61 be 62 24 44 56 f7 12 48 52
```

Luego, la lista de bytes se codifica en base64:

```
b64 = base64.b64encode(bytes.fromhex(admin_replace)).decode()
```

y el resultado es:

```
CCVCQ+BrgN6+8IlgHdaIg+MGxxnm9GQNxO4BLZJpleutQocBYIOUanHr8Qn20zl+CCVCQ+BrgN6+8Ilg
HdaIgzgWn6luX2G+YiREVvcSSFI=
```

AES en modo ECB encripta cada bloque independiente. Si el tamaño del texto claro es más largo que el tamaño del bloque es posible comenzar a ver patrones en la secuencias cifradas. Bloques con mis texto claro, producen el mismo bloque cifrado. El uso de un vector de inicialización garantiza que cada vez que cifras un mensaje con la misma clave, pero con un IV (vector de inicialización) diferente, obtendrás un resultado distinto. Esto asegura que la salida cifrada sea única para cada combinación de mensaje y IV, incluso si la clave es la misma.

Cambio de bits en modo CBC

El challenge consiste en registrarse con un correo electrónico y un conjunto de datos adicionales a un servidor, éste devuelve un perfil cifrado con AES en modo CBC. El usuario que registré tiene el correo solujan@gmail.com y como datos "Sof!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!", por lo tanto el perfil tendrá la forma:

```
user=solujan@gmail.com;data=Sof!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!;role=user
```

El texto anterior representado en bloques de 16 bytes se ve de la siguiente manera:

```
u s e r = s o l u j a n @ g m a
i l . c o m ; d a t a = S o f !
! ! ! ! ! ! ! ! ! ! ! ! ! ! !
! ! ! ! ! ! ! ! ! ! ! ! ! ! ;
r o l e = u s e r
```

La representación encriptada en AES en modo CBC en bloques de 16 bytes es:

```
12 d0 52 24 21 be b0 af f4 4c 0d 99 1b e8 fd e5
06 d3 1a b5 31 77 c9 af 97 1f 5f f1 8f 1c b5 ab
cf 40 18 06 eb c0 08 28 a7 ec e0 8e a0 ca 5b 90
27 ae a7 35 0f 63 af 35 6a c3 f0 49 a1 83 cb 5d
da c6 03 dc 3e 01 e9 bd 53 70 3a 5e d2 e1 9a 58
```

En un cifrado AES en modo CBC, los bloques de texto claro se cifran uno tras otro, y el resultado cifrado de un bloque se utiliza como vector de inicialización para el siguiente bloque. Esto significa que si un atacante puede alterar bits en el texto cifrado, esto afectará cómo se descifran los bloques posteriores.

El "bit flipping attack" implica que el atacante conozca el texto claro de algún bloque y desee cambiar su contenido sin conocer la clave de cifrado. Para lograr esto, el atacante modifica los bits en el texto cifrado de un bloque, lo que resulta en la modificación de bits correspondientes en el bloque descifrado.
Para resolver el challenge vamos a modificar el tercer bloque del texto cifrado para realizar cambios en el texto original en el cuarto bloque.
Para ello hacemos la operación xor entre el tercer bloque (a_bloque) y un bloque generado. El bloque generado contiene `;role=admin;` más 4 bytes de `b'\x00'` para rellenar el bloque de 16 bytes.
Luego completamos los 5 bloques concatenando 2 bloques antes y dos despues con `b'\x00'`

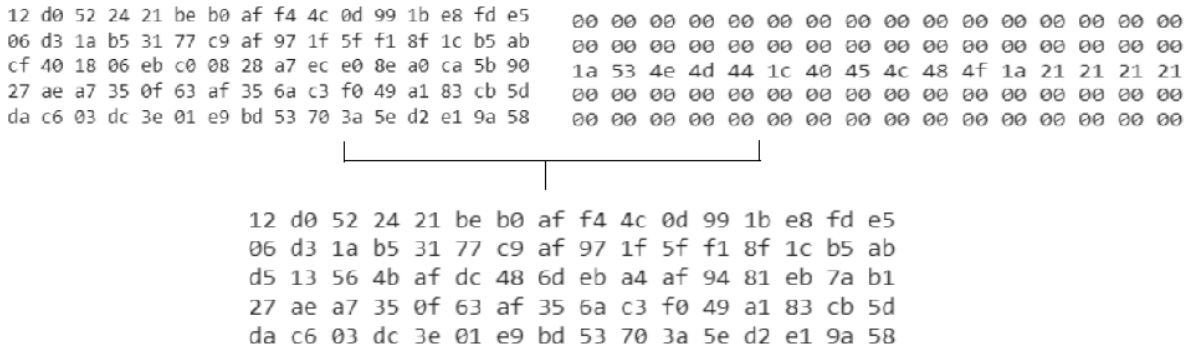
```
a_block = b'!' * AES.block_size
flipper = xor_bytes(a_block, b';role=admin;'.ljust(AES.block_size, b'\x00'))
padded = flipper.rjust(AES.block_size*3, b'\x00').ljust(len(ctx), b'\x00')
```

La variable *padded* se ve de la siguiente manera:

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1a 53 4e 4d 44 1c 40 45 4c 48 4f 1a 21 21 21 21
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Hacemos xor de la lista de bytes encriptada devuelta por el servidor y los bloques generados:

```
new_cipher = xor_bytes(bytes_list_encrypted, padded)
```




```
b64 = base64.b64encode(bytes.fromhex(iv + new_cipher.hex())).decode()
```

Concateno el IV y la nueva lista de bytes, y luego codifico en base64 el resultado final es el nuevo mensaje encriptado codificado a base64.

Para protegerse contra este tipo de ataque, es importante utilizar técnicas de autenticación y garantizar la integridad de los datos. Por ejemplo, se pueden utilizar códigos de autenticación de mensajes (MAC) o firmas digitales para verificar que el mensaje no ha sido modificado.

Descifrado en modo ECB

En este desafío, cuando el usuario envía un mensaje codificado en base64 al servidor, esta toma ese mensaje, lo combina con un mensaje secreto y luego devuelve el resultado después de cifrarlo utilizando el algoritmo AES en modo ECB, todo ello codificado en base64.
El enfoque fundamental para abordar este desafío implica la observación de las respuestas del servidor al enviar diversos mensajes seleccionados, lo que permitirá la obtención de información relevante.
La solución propuesta se basa en las recomendaciones que plantea el desafio:

- Determinar la longitud del mensaje secreto. La longitud del texto cifrado será igual a la longitud del texto claro elegido, más la longitud del mensaje secreto, más la longitud del relleno. Para ello podemos enviar primero un mensaje vacío, luego uno de longitud 1, luego otro de longitud 2, etc. Cuando cambie la longitud del texto cifrado será porque la longitud de nuestro mensaje, sumada a la longitud del mensaje secreto, completó un múltiplo del tamaño de bloque, y habremos averiguado la longitud deseada.

La siguiente función define una variable auxiliar que concatena la variable 'a' y envía el mensaje al servidor. Verificamos que el largo de la respuesta del servidor no haya cambiado. Cuando el largo de la respuesta cambia, quiere decir que el mensaje concatenado con el mensaje secreto completó el bloque, y de esta forma podemos conocer el largo del mensaje secreto.

```
current_len = 0
data = ""
while starting_len >= current_len:
  current_str = get_message(data)
  current_len = len(current_str)
  data += "a"
```

- Enviar un mensaje de 15 bytes. Esto significa que el primer bloque estará conformado por los 15 bytes que conocemos, y el primer byte del mensaje secreto, que no conocemos. Extraer el primer bloque del texto cifrado.

Si envio el string '*aaaaaaaaaaaaaaa'* de 15 bytes, el servidor va a devolver el texto cifrado de los 15 bytes más la primera letra del mensaje secreto.

a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	?
7e	d4	71	7e	af	46	82	54	53	08	ad	fa	ec	c0	0f	d7

Con el objetivo de determinar el valor de la primera letra en el mensaje secreto, procedo a enviar un mensaje que consta de 16 bytes. Varío el último byte del mensaje probando diferentes caracteres imprimibles, hasta dar con uno que, al ser cifrado, resulta en la misma salida que obtuve previamente.
La función get_next_letter recibe como argumento un mensaje inicial y un número count que indica la cantidad de bytes que quiero comparar:

```
def get_next_letter(mensaje:str, bloque:str, count:int):
  for letter in string.printable:
```

```
    new_message = mensaje + letter
    new_message_enc = get_message(new_message)
    if new_message_enc[:count] == bloque:
      print(f'La letra es: {letter}')
      return letter
  print(f'Failed!')
```

- Enviar sucesivos mensajes de 16 bytes. Los primeros 15 bytes deben ser iguales a los 15 bytes del paso anterior. Variar el último byte hasta que el primer bloque del texto cifrado sea igual al obtenido en el paso anterior. El primer byte del mensaje secreto será, pues, el byte utilizado como último byte del texto claro correspondiente a ese bloque.

La siguiente función va descifrando bloque a bloque el texto secreto:

```
bloque_inicial = 'aaaaaaaaaaaaaaa'
text_decrypt = ''
for block_size in range(32, 500, 32):
  letter = ''
  for indice in range(len(bloque_inicial), -1, -1):
    bloque_aux = bloque_inicial[:indice] + text_decrypt + letter
    letter_get = get_next_letter(bloque_aux,
get_message(bloque_inicial[:indice])[:block_size],block_size)
    letter = letter + letter_get
    text_decrypt = text_decrypt + letter
  print(f'text_decrypt: \'{text_decrypt}\'')
```

Luego de ejecutar la función permite obtener el mensaje secreto que es la canción “All for Leyna” de Billy Joel:

She stood on the tracks
Waving her arms
Leading me to that third rail, shock!
Quick as a wink
She changed her mind

She gave me a night
That's all it was
What will it take until I stop
Kidding myself, wasting my time, whoa

There's nothing else I can do
'Cause I'm doing it all for Leyna
I don't want anyone new
'Cause I'm living it all for Leyna
There's nothing in it for you
'Cause I'm giving it all to Leyna

Funciones de Hash

Segunda preimagen de una función de hash

Una función hash es un proceso matemático que convierte un conjunto variado de datos, como un archivo de texto, en un valor único de longitud fija, conocido como "hash". Este valor hash se puede emplear para verificar la integridad de copias de los datos originales sin necesidad de tener acceso a los datos originales en sí. La característica clave de la función hash es que no es posible revertir el proceso para obtener los datos originales a partir del valor hash.
El desafío consiste en encontrar una segunda preimagen al resultado de aplicar una función de hash de 24 bits a su dirección de correo electrónico. Se utiliza una función que consiste en aplicar SHA-256 y tomar los primeros N bits del resultado. En este caso tomaremos los primero 3 bytes (6 caracteres hexadecimales).
Los primeros 6 caracteres de mi hash de mi email se obtiene de la siguiente manera:

```
from hashlib import sha256
key = sha256('solujan@gmail.com'.encode()).hexdigest()[:6]
d9c026
```

Para obtener la segunda pre imagen utilizo fuerza bruta, esto es agregar tantos caracteres como sea necesario para encontrar los mismos 6 caracteres. La librería string tiene una variable global llamada `string.printable` que devuelve un string con todos los caracteres imprimibles. Recorro la lista de caracteres, y los voy concatenando a mi email.
En una primera implementación solo agregué un carácter, luego dos y así sucesivamente hasta encontrar una segunda pre imagen.

```
def get_segunda_imagen():
  for index in string.printable[:-6]:
    for index1 in string.printable[:-6]:
      for index2 in string.printable[:-6]:
        for index3 in string.printable[:-6]:
          text = email.encode() + index.encode() + index1.encode() +
index2.encode() + index3.encode()
          aux = sha256(text).hexdigest()[:6]
          if aux == key:
            print(text)
            return text
```

Cuando agregué el cuarto carácter encontré la segunda pre imagen:

```
segunda_pre_imagen = get_segunda_imagen()
print(sha256(text).hexdigest()[:6])
b'solujan@gmail.come1Kn'
d9c026
```

Como estoy tomando solo los primeros 24 bits del hash de 256 bits, tiene un costo computacional bajo encontrar la segunda pre imagen. Es necesario tomar mayor cantidad de bits para que sea seguro.

Autenticación

Ataque de extensión de longitud

El servidor permite registrar un email:

```
email = 'solujan@gmail.com'
url = f'.../cripto/secret-prefix-mac/{email}/challenge'
# Solicitud
request = requests.get(url)
# Body de la respuesta
message = request.content
message
```

Devuelve una un query string de la forma:

```
b'user=solujan@gmail.com&action=show&mac=4bb1e69e282cd62a9b8c87d19
9d30d0813dce6261b6f9d632b39749135f63381'
```

El servidor llevaría a cabo la solicitud dada solo si la firma es válida para el usuario. La firma utilizada aquí es un código de autenticación de mensajes (MAC), firmado con una clave no conocida, pero se sabe que tiene una longitud de 16 bytes.

El objetivo del desafío es realizar un ataque de extensión de longitud que permita falsificar la query string para agregar admin=true, tiene que contener un campo user y una MAC válida. El mensaje original está formado por la concatenación de los pares clave-valor (sin incluir el MAC), ordenados alfabéticamente por clave, y eliminando los símbolos = y &. En el caso anterior, el mensaje sería:

```
actionshowusersolujan@gmail.com
```

Sobre este mensaje se calcula un MAC de la siguiente forma:

$$MAC = SHA256\,(secreto \,||\, mensaje)$$

$$MAC = 4bb1e69e282cd62a9b8c87d199d30d0813dce6261b6f9d632b39749135f63381$$

En un ataque de extensión de longitud, es posible introducir el hash (MAC devuelta por el servidor) en el estado de la función de hash y continuar desde donde se quedó la solicitud original, siempre y cuando se conozca la longitud de la solicitud original.

SHA-256, al igual que otros algoritmos que emplean la estructura de Merkle-Damgård, es vulnerable a ataques de extensión de longitud. Debido a que el valor resultante de la función de hash es un punto intermedio válido en un mensaje más largo, es factible generar una extensión del mensaje sin requerir conocimiento del secreto.
El mensaje extendido tiene la forma:

$$mensaje.extendido = mensaje \,||\, padding \,||\, extension$$

El relleno es la información adicional requerida para llenar el bloque de 64 bytes. El mensaje original tiene una longitud de 31 bytes (31 * 8 = 248 bits). El propósito es llenar el bloque de

64 bytes (512 bits). Para agregar el relleno, es necesario añadir un "1" al principio y luego rellenar con "0" los 56 bytes restantes. Los últimos 8 bytes contienen la información del tamaño del mensaje, que incluye el tamaño de la clave, en este caso, 16 bytes

```
def pad(message):
  key_len_bytes = 16
  message_len_bytes = len(message)
  len_total_bits = (message_len_bytes + key_len_bytes)*8
  len_bytes = len_total_bits.to_bytes(2, byteorder = 'big')
  padded = message + b'\x80'.ljust(62-(len(message) + 16), b'\x00') + len_bytes
  return padded

padded = pad(mensaje_bytes)
```

El mensaje original con el padding es:

```
actionshowusersolujan@gmail.com\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x01\x78
```

Elegí la extensión admintrueusersolujan@gmail.com para cumplir con las condiciones planteadas (admin=true y que contenga un campo usuario). Con la librería sha256sum nos permite calcular el nuevo MAC

```
extension = b'admintrueusersolujan@gmail.com'
new_mac = sha256sum.SHA256(extension, original_mac, len(padded)+16).digest()
new_mac
e02f8dfab0e2cc8ec783399655b4d534553a563f223cf5a39e484ac9a94cb011
```

La nueva query string tiene la forma:

```
a=ctionshowusersolujan%40gmail.com%80%00%00%00%00%00%00%00%00%00%00%00%00%01x&admin=true&user=solujan%40gmail.com&mac=e02f8dfab0e2cc8ec783399655b4d534553a563f223cf5a39e484ac9a94cb011
```

Para evitar este tipo de ataques es posible utilizar un código de autenticación de mensajes basado en hash, que permite hasehar información, pero además ofrece una clave secreta para autenticar los datos. Así, resulta imposible ampliar la cadena de datos original, incluso si se tiene acceso al estado interno del algoritmo hash.

CBC-MAC

CBC-MAC (Cipher Block Chaining Message Authentication Code) es un método de autenticación de mensajes que utiliza el modo de cifrado por bloques conocido como Cipher Block Chaining (CBC). Se utiliza para calcular un código de autenticación para un mensaje o conjunto de datos. Este código se genera aplicando cifrado en bloques sucesivos a los datos y luego tomando el último bloque cifrado como el código de autenticación. Este código se adjunta al mensaje y se utiliza para verificar la integridad de los datos y asegurarse de que no han sido alterados. CBC-MAC no proporciona confidencialidad, solo integridad de los datos. La efectividad del algoritmo de MAC CBC-MAC se ve comprometida cuando se utilizan mensajes de longitud variable.
Sean dos mensajes M y M`:

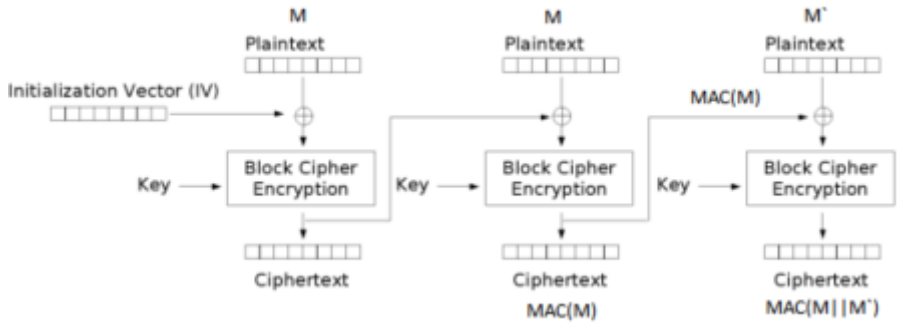
$$M = M_1 \,||\, M_2 \,||\, \ldots \,||\, M_n \qquad T = CBCMAC(M)$$

$$M' = M'_1 \,||\, M'_2 \,||\, \ldots \,||\, M'_n \qquad T' = CBCMAC(M')$$

Se puede generar un mensaje:

$$M'' = M \,||\, M'_1 \oplus T \,||\, M'_2 \,||\, \ldots \,||\, M'_n \qquad T' = CBCMAC(M'')$$

En el siguiente gráfico se pueden observar las ecuaciones anteriores:



El desafío implica la creación de query string falsificada que esté autenticada con CBC-MAC.
Para obtener el mensaje que se desea falsificar, es necesario realizar una solicitud GET a una URL con un formato específico:

```
email = 'solujan@gmail.com'
url = f'.../cripto/cbc-mac/{email}/challenge'
request = requests.get(url)
message = request.content
message
```

El contenido de la respuesta del servidor tiene la forma:

```
b'from=User
<user@example.com>&solujan@gmail.com=1000&comment=Invoice&mac=77d162008b8e3b0cf3
29d8c7679934db'
```

Extraigo el mensaje original (sin MAC):

```
originalMessage =  b'from=User
<user@example.com>&solujan@gmail.com=1000&comment=Invoice'
```

El algoritmo CBC utiliza PKCS7 como esquema de relleno. Su objetivo principal es permitir que los datos se ajusten al tamaño de bloque requerido por el algoritmo de cifrado. PKCS7 rellena los datos con un valor que indica cuántos bytes se han agregado como relleno. La siguiente función permite agregar el padding a un mensaje:

```
def pad(msg):
  padder = padding.PKCS7(128).padder()
  return padder.update(msg)+padder.finalize()
```

Y el mensaje original con padding:

```
b'from=User
<user@example.com>&solujan@gmail.com=1000&comment=Invoice\r\r\r\r\r\r\r\r\r\r\
r\r'
```

El nuevo mensaje que quiero concatenar es:

```
prependMessage = b'from=User
<user@example.com>&solujan@gmail.com=1000&comment=Invoice'
```

Para crear el $M'' = M \,||\, M'_1 \oplus T \,||\, M'_2 || \ldots || \, M'_n$ aplico operación XOR al primer bloque del nuevo mensaje y la MAC original. Luego concateno los bloques resultantes:

```
def create_new_msg(original, prependMessage, originalMac):
  newFirstBlock = bytearray(prependMessage [:16])
  for i in range (16):
    newFirstBlock[i] ^= originalMac[i]
  newFirstBlock = bytes(newFirstBlock)
  return original + newFirstBlock + prependMessage [16:]

newMessage = create_new_msg(padded_original_msg, prependMessage, mac)
```

De esta manera obtenemos el nuevo mensaje y MAC que tiene que ser enviado como query string:

```
New message b'from=User
<user@example.com>&solujan@gmail.com=1000&comment=Invoice\r\r\r\r\
r\r\r\r\r\r\r\rQE\x0b\\\x0bgCUJB\x04\x10@\x07B#example.com>&solu
jan@gmail.com=1000&comment=Invoice'
MAC: b'77d162008b8e3b0cf329d8c7679934db'
```

Criptografía de clave pública

Ataque de Broadcast sobre RSA

Puede probarse que para encontrar un texto que ha sido cifrado múltiples veces con RSA utilizando distintas claves públicas con exponente e, basta con capturar e textos cifrados para obtener el texto claro.
En este caso particular $e = 3$, por lo que es necesario hacer tres llamadas al servidor para obtener tres textos cifrados y respectiva key:

```
c = []
n = []
for each in range(0,3):
  key,cipher = get_cipher()
  c.append(int(cipher, 16))
  n.append(key)
```

Sean $(c_1, n_1), (c_2, n_2), (c_3, n_3)$ tres textos cifrados con los correspondientes módulos de las claves públicas utilizadas.
Sea m el texto claro. Entonces:

$$c_1 = m^3 \ mod \ n_1$$
$$c_2 = m^3 \ mod \ n_2$$
$$c_3 = m^3 \ mod \ n_3$$

Es decir, c_1, c_2, c_3 son los restos de dividir $x = m^3$ por n_1, n_2, n_3 respectivamente.
Por lo tanto:

$$t_1 = c_1 . N_1 . (N_1^{-1} \ mod \ n_1)$$
$$t_2 = c_2 . N_2 . (N_2^{-1} \ mod \ n_2)$$
$$t_3 = c_3 . N_3 . (N_3^{-1} \ mod \ n_3)$$

$$x = (t_1 + t_2 + t_3) \ mod \ N$$

Donde

$$N = n_1, n_2, n_3$$
$$N_1 = n_1, n_2, n_3$$
$$N_2 = n_1, n_2, n_3$$
$$N_3 = n_1, n_2, n_3$$

Una vez averiguado $x = m^3$ solo nos queda extraer la raíz cúbica para obtener el texto claro original.
Si llevamos las ecuaciones anteriores a código python:

```
def rsa_broadcast_attack(c: list, m: list):
    c0, c1, c2 = c[0], c[1],c[2]
    n0, n1, n2 = m[0], m[1], m[2]
    N0, N1, N2 = n1 * n2, n0 * n2, n0 * n1
    N = n0 * n1 * n2

    t0 = (c0 * N0 * int(gmpy2.invert(N0, n0)))
    t1 = (c1 * N1 * int(gmpy2.invert(N1, n1)))
    t2 = (c2 * N2 * int(gmpy2.invert(N2, n2)))
    x = (t0 + t1 + t2) % N
    return int(gmpy2.iroot(c, 3)[0])
```

```
cracked_int = rsa_broadcast_attack(c, n)
hacked_quote = uint_to_bytes(cracked_int)
hacked_quote
```

Y obtenemos:

```
b'A city is a large community where people are lonesome
together\n\t\t-- Herbert Prochnow'
```

Conclusión

A lo largo de este trabajo, se ha propuesto una solución para abordar los desafíos criptográficos planteados en la asignatura de la materia. Se incluye una breve exposición teórica de los algoritmos, una descripción práctica de cómo vulnerarlos y sugerencias para mejorar su seguridad.

Todo el código fuente incluido en el trabajo se encuentra en el siguiente repositorio: https://github.com/solujan/diplomatura-ciberseguridad
En el repositorio se encuentran los desafíos resueltos en el trabajo, y otros que no fueron completados.