

Informe Laboratorio 1

Sección 1

Alumno Rayen Millaman
e-mail: rayen.millaman@mail.udp.cl

Marzo de 2024

Índice

1. Descripción	2
2. Actividades	2
2.1. Algoritmo de cifrado	2
2.2. Modo stealth	2
2.3. MitM	4
3. Desarrollo de Actividades	4
3.1. Actividad 1	4
3.2. Actividad 2	6
3.3. Actividad 3	10

1. Descripción

1. Usted empieza a trabajar en una empresa tecnológica que se jacta de poseer sistemas que permiten identificar filtraciones de información a través de Deep Packet Inspection (DPI).

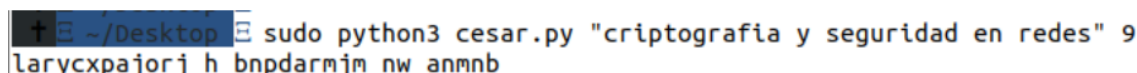
A usted le han encomendado auditar si efectivamente estos sistemas son capaces de detectar las filtraciones a través de tráfico de red. Debido a que el programa ping es ampliamente utilizado desde dentro y hacia fuera de la empresa, su tarea será crear un software que permita replicar tráfico generado por el programa ping con su configuración por defecto, pero con fragmentos de información confidencial. Recuerde que al comparar tráfico real con el generado no debe gatillar alarmas.

De todas formas, deberá hacer una prueba de concepto, en la cual se demuestre que al conocer el algoritmo, será fácil determinar el mensaje en claro.

2. Actividades

2.1. Algoritmo de cifrado

1. Generar un programa, en python3, que permita cifrar texto utilizando el algoritmo Cesar. Como parámetros de su programa deberá ingresar el string a cifrar y luego el corrimiento.



```
➤ ~/Desktop ➤ sudo python3 cesar.py "criptografia y seguridad en redes" 9
larycxpajorj h bnpdarmjm nw anmnb
```

2.2. Modo stealth

1. Generar un programa, en python3, que permita enviar los caracteres del string (el del paso 1) en varios paquetes ICMP request (un caracter por paquete en el byte menos significativo del contador ubicado en el campo data de ICMP) para que de esta forma no se gatillen sospechas sobre la filtración de datos.

Para la generación del tráfico ICMP, deberá basarse en los campos de un paquete generado por el programa ping basado en Ubuntu, según lo visto en el lab anterior disponible acá.

El envío deberá poder enviarse a cualquier IP. Para no generar tráfico malicioso dentro de esta experiencia, se debe enviar el tráfico a la IP de loopback.

```

$ ./Desktop $ sudo python3 pingv4.py "larycxpajorj h bnpdarmjm nw anmnb"
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.

```

A modo de ejemplo, en este caso, cada paquete transmite un caracter, donde el último paquete transmite la letra b, correspondiente al caracter en plano “s”.

Data (48 bytes)		
Data: 626009000000000000000101112131415161718191a1b1c1d1e1f20212223242526272		
[Length: 48]		
0000	ff ff ff ff ff ff 00 00 00 00 00 00 08 00 45 00E.
0010	00 54 00 01 00 00 40 01 76 9b 7f 00 00 01 7f 06	.T...@. v.....
0020	06 06 08 00 56 83 00 01 00 21 64 22 13 05 00 00	...V...!d"....
0030	00 00 62 60 09 00 00 00 00 00 10 11 12 13 14 15	..b`.....
0040	16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25! "\$%&
0050	26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35	&'()*+,-./012345
0060	36 37	67

2.3. MitM

1. Generar un programa, en python3, que permita obtener el mensaje transmitido en el paso2. Como no se sabe cual es el corrimiento utilizado, genere todas las combinaciones posibles e imprímalas, indicando en verde la opción más probable de ser el mensaje en claro.

```

$ sudo python3 readv2.py cesar.pcapng
0      larycxpajorj h bnpdarmjm nw anmnb
1      kzqxbwozinqi g amoczqlil mv zmlma
2      jypwavnyhmpf f zlnbypkhk lu ylkklz
3      ixovzumxglog e ykmaxojgj kt xkjky
4      hwnuytlwfknf d xjlzwnifi js wjijx
5      gvmtxskvejme c wikyvmeheh ir vihiw
6      fulswrjudild b vhjxulgdg hq uhghv
7      etkrvqitchkc a ugiwtkfcf gp tgfgu
8      dsjquphsbgjb z tfhvsjebe fo sfef
9      criptografia y seguridad en redes
10     bqhosnfqzehz x rdftqhczc dm qdcdr
11     apgnrmepdygy w qcespgbyb cl pcbbc
12     zofmqldoxcfx v pbdrofaxa bk obabp
13     ynelpkcnwbew u oacqnezwz aj nazao
14     xmdkojbmadv t nzbpmdyvy zi mzyzn
15     wlcjnia luzcu s myaolcxux yh lyxym
16     vkbimhzktybt r lxznkbwtw xg kxwxl
17     ujahlgysxas q kwymjavsv wf jwvwk
18     tizgkfxirwzr p jvxlizuru ve ivuvj
19     shyfjewhqvyq o iuwkhytqt ud hutui
20     rgxeidvgpuxp n htvjgxsp s tc gtsth
21     qfwdhcufotwo m gsuifwr or sb fsrsg
22     pevcbtensvn l frthevqnq ra erqrf
23     odubfasdmrum k eqsgdupmp qz dqpqe
24     nctaezrclqtl j dprfctolo py cpopd
25     mbszdyqbksk i coqebnkn ox bonoc

```

Finalmente, deberá indicar los 4 mayores problemas o complicaciones que usted tuvo durante el proceso del laboratorio y de qué forma los solucionó.

3. Desarrollo de Actividades

3.1. Actividad 1

Se genera un programa en python3, que permite cifrar texto utilizando el algoritmo Cesar, el cual consiste en un cifrado de sustitución monoalfabética. Para su uso se crea un script en el que se le ingresa un string a cifrar con su respectivo corrimiento, además se hace uso de la

siguiente fórmula para la simplificación de los índices cambiados.

$$P = P_i + \text{Desplazamiento} \% 26 \quad (1)$$

Donde P representa posición

```

rayen@hp:~/Documents/cripto$ sudo python3 cesar.py
Texto a cifrar: criptografia y seguridad en redes
Corrimiento: 7
Texto cifrado: jypwavyhmpfh f_zlnbypkhhk lu yklkz

```

Figura 1: Ejecución código cesar.

```

cesar.py > cifrado_cesar
1 def cifrado_cesar(texto, desplazamiento):
2     resultado = ""
3     for caracter in texto:
4         if caracter.isalpha():
5             posicion = ord(caracter.lower()) - ord('a') if caracter.islower() else ord(caracter) - ord('A')
6             nueva_posicion = (posicion + desplazamiento) % 26
7             nuevo_caracter = chr(nueva_posicion + ord('a')) if caracter.islower() else chr(nueva_posicion + ord('A'))
8             resultado += nuevo_caracter
9         else:
10            resultado += caracter
11    return resultado
12
13
14 texto = input("Texto a cifrar: ")
15
16
17 try:
18     desplazamiento = int(input("Corrimiento: "))
19     texto_cifrado = cifrado_cesar(texto, desplazamiento)
20     print("Texto cifrado:", texto_cifrado)
21 except ValueError:

```

Figura 2: Código para la funcionalidad de cifrado Cesar.

Para la Figura 2 el código presentado opera sobre un string introducido, junto con un valor de desplazamiento, invocando una función que inicialmente verifica si cada carácter del string es una letra. Si lo es, el carácter se convierte a su correspondiente valor numérico en el alfabeto (donde 'a' es 0, 'b' es 1, etc.). Luego, mediante una operación se determina y facilita la nueva posición numérica del carácter ajustado por el desplazamiento. Finalmente, este valor numérico se traduce de vuelta a su respectivo carácter alfabético, completando así el proceso de cifrado y entregando el nuevo string.

3.2. Actividad 2

Una vez que se ha obtenido el mensaje cifrado en la actividad inicial, se procede al envío de una serie de paquetes request ICMP basados en Ubuntu que equivale al número total de caracteres alfabéticos presentes en dicho mensaje. En cada paquete, un carácter específico se sitúa en el byte de menor valor dentro de la carga útil. La inyección secuencial de estos paquetes es monitoreada a través de Wireshark en la interfaz de red loopback.

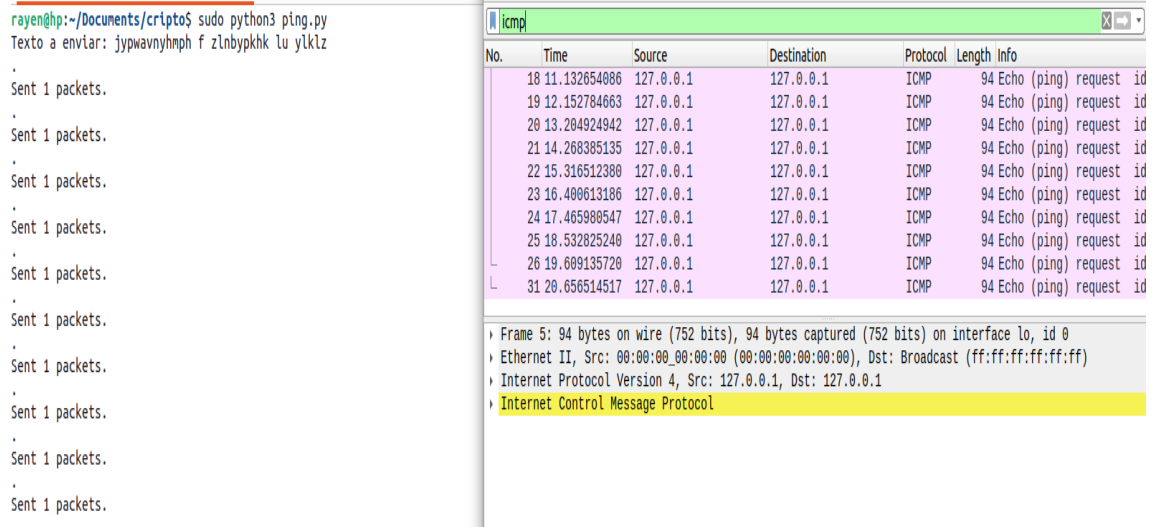


Figura 3: Ejecución del envío de paquetes request icmp.

```

Internet Control Message Protocol
  Type: 8 (Echo (ping) request)
  Code: 0
  Checksum: 0x45ea [correct]
  [Checksum Status: Good]
  Identifier (BE): 38 (0x0026)
  Identifier (LE): 9728 (0x2600)
  Sequence Number (BE): 4 (0x0004)
  Sequence Number (LE): 1024 (0x0400)
  [No response seen]
    [Expert Info (Warning/Sequence): No response seen to ICMP request]
      [No response seen to ICMP request]
      [Severity level: Warning]
      [Group: Sequence]
    Timestamp from icmp data: Mar 29, 2024 05:38:39.065793000 -03
    [Timestamp from icmp data (relative): 0.742779781 seconds]
  Data (49 bytes)
    Data: 0101010000000000101112131415161718191a1b1c1d1e1f202122232425262728292a2
    [Length: 49]

0000  ff ff ff ff ff ff 00 00 00 00 00 08 00 45 00  .....E.
0010  00 55 00 01 00 00 40 01 7c a5 7f 00 00 01 7f 00  .U...@. |.....
0020  00 01 08 00 45 ea 00 26 00 04 66 06 7e 0f 00 01  ...E.& .f~...
0030  01 01 01 01 01 00 00 00 00 00 10 11 12 13 14 15  .....
0040  16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25  ..... !"#$$%
0050  26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35  &'()*+,-./012345
0060  36 37 0c                                     67.

```

Figura 4: Contenido paquete request icmp.

Al ejecutar el script y entregar la nueva palabra cifrada, se envía los paquetes icmp con retardo de 1 segundo entre cada uno ejemplificado en la figura 3. Además como se muestra en la figura 4 cada paquete transporta un payload o data específico de 49 bytes. Este payload se estructura de la siguiente manera: comienza con un 'payload base' que incluye 3 bytes iniciales 0x01, a los cuales se añaden otros 5 bytes de valor 0x00, seguido por una secuencia de 40 bytes que varían desde 0x10 hasta 0x37. A esta estructura se le incorpora un byte adicional, que se determina según el valor numérico correspondiente al carácter actual del mensaje cifrado.

payload base 48 byte

character 1 byte (byte menos significativo)

Para incorporar el valor del timestamp, se insertó en el payload del código cuatro bytes al principio del mismo. Este procedimiento implicó la sobrescritura de los primeros cuatro bytes del payload base preexistente, lo cual no conllevó en un incremento de la longitud total del payload, sino en una sustitución de los datos originales. Provocando un 'corrimiento de bytes', donde los bytes subsiguientes se desplazan hacia adelante en el payload. La inserción del timestamp de esta manera refleja la hora exacta en la que el paquete fue enviado, mostrada en la figura 4

55	40.702121159	127.0.0.1	127.0.0.1	ICMP	94 Echo (ping) request	id=0x0003, seq=29/7424, ttl=64 (no response found!)
56	41.758842692	127.0.0.1	127.0.0.1	ICMP	94 Echo (ping) request	id=0x0003, seq=30/7680, ttl=64 (no response found!)
57	42.817986037	127.0.0.1	127.0.0.1	ICMP	94 Echo (ping) request	id=0x0003, seq=31/7936, ttl=64 (no response found!)
58	43.862158558	127.0.0.1	127.0.0.1	ICMP	94 Echo (ping) request	id=0x0003, seq=32/8192, ttl=64 (no response found!)
59	44.918098161	127.0.0.1	127.0.0.1	ICMP	94 Echo (ping) request	id=0x0003, seq=33/8448, ttl=64 (no response found!)
120	107.807072234	127.0.0.1	127.0.0.1	ICMP	94 Echo (ping) request	id=0x0004, seq=1/256, ttl=64 (no response found!)
121	108.862228723	127.0.0.1	127.0.0.1	ICMP	94 Echo (ping) request	id=0x0004, seq=2/512, ttl=64 (no response found!)
122	109.922317237	127.0.0.1	127.0.0.1	ICMP	94 Echo (ping) request	id=0x0004, seq=3/768, ttl=64 (no response found!)
127	110.990677436	127.0.0.1	127.0.0.1	ICMP	94 Echo (ping) request	id=0x0004, seq=4/1024, ttl=64 (no response found!)
128	112.058185616	127.0.0.1	127.0.0.1	ICMP	94 Echo (ping) request	id=0x0004, seq=5/1280, ttl=64 (no response found!)
129	113.130685270	127.0.0.1	127.0.0.1	ICMP	94 Echo (ping) request	id=0x0004, seq=6/1536, ttl=64 (no response found!)

Figura 5: Contenido paquete request icmp.

Para asegurar una identificación coherente de ID durante el envío de múltiples paquetes, basándose en la cantidad de caracteres del string cifrado, se emula un proceso similar a la fragmentación de paquetes, que facilita la transmisión e identificación eficaz de los datos y, en este contexto, permite que la emulación de los paquetes pasen 'desapercibidos'. Tal como se muestra en la figura 5, donde se destaca el envío del último paquete de una secuencia y el inicio de otra con un identificador distinto, en este caso con el hexadecimal 4 se actualiza al reiniciar el script, distinguiendo finalmente cada conjunto de datos como una unidad separada, como también en la figura 4 que se identifica que esta en el n° de conjunto de datos 38, en la secuencia número 4 y con un checksum correcto.

Por otro lado la secuencia numérica entre paquetes se mantiene empezando desde el valor 1 decimal hasta completar el envío de paquetes, y del mismo modo se produce un reinicio cuando se crea un nuevo conjunto de datos.


```

ping.py > [e] id_icmp
1  from scapy.all import ICMP, IP, send, Raw
2  from time import sleep, time
3
4  archivo_temporal = "id_icmp.txt"
5
6
7  def leer_id_icmp():
8      try:
9          with open(archivo_temporal, "r") as f:
10             contenido = f.read().strip()
11             if contenido.isdigit():
12                 return int(contenido)
13             else:
14                 return 0
15         except FileNotFoundError:
16             return 0
17
18  def guardar_id_icmp(id_icmp):
19      with open(archivo_temporal, "w") as f:
20          f.write(str(id_icmp))
21
22  id_icmp = leer_id_icmp()

```

Figura 6: Contenido código parte I.

El inicio del código está diseñado para gestionar la identificación de IP, leyendo desde un archivo temporal la secuencia actual del identificador, cuando concluye la ejecución del script, se invoca una función encargada de actualizar y almacenar el identificador en el archivo, preparándolo para la próxima ejecución

El uso de bibliotecas utilizadas de scapy se utilizaron para la construcción y modificación de los paquetes mientras que el uso de sleep y time se utilizaron para el manejo del tiempo.

```

-----
rayen@hp:~/Documents/cripto$ ls
cesar.py  hola.pcapng  id_icmp.txt  ping.py  read.py
-----

```

Figura 7: Contenido estructura de laboratorio 1.

```

ping.py > enviar_mensaje_oculto
29
30 def enviar_mensaje_oculto(mensaje, ip_destino='127.0.0.1'):
31     global id_icmp
32     seq_icmp = 1
33     icmp_id = (0x0001 + id_icmp) & 0xffff
34     payload_base = b'\x00' * 1 + b'\x01' * 6 + b'\x00' * 5 + bytes(range(0x10, 0x38))
35
36     for caracter in mensaje:
37         if caracter == ' ':
38             valor_caracter = 26
39         elif caracter.isalpha():
40             valor_caracter = ord(caracter.lower()) - ord('a')
41         else:
42             valor_caracter = 27
43
44         timestamp_bytes = int(time()).to_bytes(4, 'big')
45         valor_caracter_bytes = valor_caracter.to_bytes(1, 'little')
46
47         payload_total = timestamp_bytes + payload_base + valor_caracter_bytes
48
49
50
51     # Crear y enviar el paquete ICMP con el payload
52     paquete = IP(dst=ip_destino)/ICMP(type="echo-request", id=icmp_id, seq=seq_icmp)/Raw(load=payload_t
53     send(paquete)
54     seq_icmp += 1
55
56     # Esperar 1 segundo entre cada envío para cumplir con el requisito de stealth
57     sleep(1)
58
59     id_icmp += 1
60     guardar_id_icmp(id_icmp)
61
62 mensaje_a_enviar = input("Texto a enviar: ")
63 enviar_mensaje_oculto(mensaje_a_enviar)
64

```

Figura 8: Contenido código parte II.

La segunda parte del código gestiona el envío de paquetes utilizando una estructura de payload predefinida que comprende un conjunto base de bytes y una secuencia numérica. Además, se integra un intervalo de tiempo definido para la ejecución entre el envío de cada paquete, lo que permite un control sobre la cadencia de transmisión. En el contexto del conjunto de datos en cuestión, se asigna una identificación única (ID), y para cada paquete, se incorpora un timestamp que registra el momento exacto de su envío. Adicionalmente, se establece el valor numérico correspondiente a cada carácter del mensaje, con la excepción del checksum, que es calculado y adjuntado automáticamente por la biblioteca Scapy.

Para ajustarse a los requisitos, se inserta el timestamp en el payload, provocando un corrimiento de los bytes existentes. Tal y como se ilustra en la figura 8, se desplazan los bytes hacia posiciones de menor significado, asegurando que el timestamp ocupe la sección prioritaria dentro del payload.

3.3. Actividad 3

Tras encriptar un mensaje, el siguiente paso es su descifrado mediante técnicas de fuerza bruta para identificar la solución correcta, por medio del archivo 'read.py' ingresando el argumento

por la terminal. En este laboratorio, se aplicó el método a la frase 'criptografía y seguridad en redes', utilizando un desplazamiento de siete posiciones.

```
.
Sent 1 packets.
.
Sent 1 packets.
rayen@hp:~/Documents/cripto$ sudo python3 read.py
Ingrese la ruta al archivo pcap: hola.pcapng
Mensaje codificado: jypwavyhmpf f zlnbypkhk lu yklz
Rotación 0: jypwavyhmpf f zlnbypkhk lu yklz
Rotación 1: ixovzumxglog e ykmaxojgj kt xkjky
Rotación 2: hwnuytlwfknd d xjlzwnifi js wjijx
Rotación 3: gvmtxskvejme c wikyvmheh ir vihiw
Rotación 4: fulswrjudild b vhjxulgdg hq uhghv
Rotación 5: etkrvqitchkc a ugiwtkfcf gp tgfgu
Rotación 6: dsjquphsbgjb z tfhvsjebe fo sfef
Rotación 7: criptografía y seguridad en redes
Rotación 8: bqhosnfqzehz x rdftqhczc dm qdcdr
Rotación 9: apgnrmepdygy w qcespgbyb cl pcbcq
Rotación 10: zofmqldoxcfx v pbdrofafa bk obabp
Rotación 11: ynelpkcnwbew u oacqnezwz aj nazao
Rotación 12: xmdkojbmadv t nzbpmdivy zi mzyzn
Rotación 13: wlcjniauzcu s myaolcxux yh lyxym
Rotación 14: vkbimhzktybt r lxznkbwtw xg kxwxl
Rotación 15: ujahlgysxas q kwymjavsv wf jwvwk
Rotación 16: tizgkfxirwzr p jvxlizuru ve ivuvj
Rotación 17: shyfjewhqvyq o iuwkhytqt ud hutui
Rotación 18: rgxeidvgpuxp n htvjgxspc tc gtsth
Rotación 19: qfwdhucufotwo m gsuifwrwr sb fsrsg
Rotación 20: pevcbtensvn l frthevqnq ra erqrf
Rotación 21: odubfasdmrum k eqsgdupmp qz dqpqe
Rotación 22: nctaezrclqtl j dprfctolo py cpopd
Rotación 23: mbszdyqbksk i coqebnkn ox bonoc
Rotación 24: larycxpajorj h bnpdarmjm nw anmnb
Rotación 25: kzqxbwozinqi g amoczqlil mv zmlma
rayen@hp:~/Documents/cripto$
```

Figura 9: Ejecución script de lectura del archivo pcapng.

```
read.py > ...
1 from scapy.all import rdpcap, Raw
2 from colorama import Fore, init
3
4 # Inicializa colorama
5 init(autoreset=True)
6
7 def decodificar_cesar(texto, desplazamiento):
8     decodificado = ''
9     for char in texto:
10         if char.isalpha():
11             offset = ord('a') if char.islower() else ord('A')
12             decodificado += chr((ord(char) - offset - desplazamiento) % 26 + offset)
13         else:
14             decodificado += char
15     return decodificado
16
17 def leer_mensaje_de_pcap(archivo_pcap):
18     paquetes = rdpcap(archivo_pcap)
19     mensaje_codificado = ''
20     for paquete in paquetes:
21         if paquete.haslayer(Raw):
22             datos = paquete[Raw].load
23             valor_caracter = datos[-1]
24             if 0 <= valor_caracter <= 25:
25                 mensaje_codificado += chr(valor_caracter + ord('a'))
26             elif valor_caracter == 26:
27                 mensaje_codificado += ' '
28     return mensaje_codificado
29
30 def intentar_todas_rotaciones(texto_codificado):
31     resultados = []
32     for i in range(26):
33         resultados.append((i, decodificar_cesar(texto_codificado, i)))
34     return resultados
35
```

Figura 10: Código lectura archivo pcapng, primera parte.

En esta parte del código, se tiene 3 funciones con centrada principalmente en decodificar aplicando cesar en reversa, leer el paquete pcapng entregado y extraer el último byte, luego de sacado los byte se aplica fuerza bruta generando todos los posibles intercambios de caracteres.

```

35
36 def puntuacion_legibilidad(mensaje):
37     secuencias_comunes = [
38         'de', 'la', 'que', 'el', 'en', 'los', 'con', 'las', 'un', 'por', 'para', 'es',
39         'al', 'se', 'del', 'lo', 'como', 'mas', 'su', 'le', 'si', 'mi', 'me', 'esta',
40         'ya', 'que', 'por', 'su', 'al', 'lo', 'le', 'se'
41     ]
42     puntuacion = 0
43     mensaje = mensaje.lower()
44     for secuencia in secuencias_comunes:
45         puntuacion += mensaje.count(secuencia)
46     return puntuacion
47
48 def resaltar_mejor_opcion(resultados):
49     mejor_puntuacion = -1
50     rotacion_probable = 0
51
52     # Determinar la opción más probable
53     for rotacion, mensaje in resultados:
54         puntuacion_actual = puntuacion_legibilidad(mensaje)
55         if puntuacion_actual > mejor_puntuacion:
56             mejor_puntuacion = puntuacion_actual
57             rotacion_probable = rotacion
58
59     # Imprimir resultados, destacando el más probable en verde
60     for rotacion, mensaje in resultados:
61         if rotacion == rotacion_probable:
62             print(Fore.GREEN + f"R {rotacion}: {mensaje}")
63         else:
64             print(f"Rotación {rotacion}: {mensaje}")
65
66 archivo_pcap = input("Ingrese la ruta al archivo pcap: ")
67 mensaje_codificado = leer_mensaje_de_pcap(archivo_pcap)
68 print("Mensaje codificado:", mensaje_codificado)
69
70 resultados = intentar_todas_rotaciones(mensaje_codificado)
71 resaltar_mejor_opcion(resultados)
72

```

Figura 11: Código lectura archivo pcapng, segunda parte.

La segunda parte del código se centra en formar un diccionario de posibles secuencias de caracteres en el idioma español, devolviendo una puntuación para poder elegir el string mayor convincente. Luego, imprime todos los mensajes, resaltando en verde el mensaje con la puntuación de legibilidad más alta, indicando que ese mensaje es la decodificación más probablemente correcta del mensaje cifrado original.

Conclusiones y Comentarios

El desarrollo de las actividades resultó ser satisfactorio, asegurando un sólido entendimiento y manejo de Wireshark, así como del funcionamiento del protocolo ICMP y sus parámetros distintivos. La rigurosa inspección de los paquetes y la ajustada manipulación de parámetros subrayaron la crítica necesidad de un monitoreo exacto del tráfico de red, esencial para descubrir comportamientos inusuales y salvaguardar la integridad de las comunicaciones.

Issues

Uno de los problemas que presenté fue en cómo dejar bien estructurada la casilla visible del Wireshark para el payload y el timestamp sin que la cantidad de bytes descendiera a 40, como me sucedió en reiteradas ocasiones sin entender la razón de por qué sucedía.

Otro problema fue que me costó buscar una solución para que la ID pudiera ir variando según el conjunto de datos, además, el uso de un archivo txt no lo considero muy rentable en comparación con la solución que encontré por internet y con chat GPT, pero que representaban una mayor complejidad.

Finalmente, un problema de menor complejidad fue realizar el script de cifrado César, debido a que se demoraba en realizar todas las iteraciones; sin embargo, de la ayudantía me acordé de una fórmula para que el funcionamiento sea más expedito.

Indica 4 problemas y su solución