**PROJECT 1.**

Task 1.

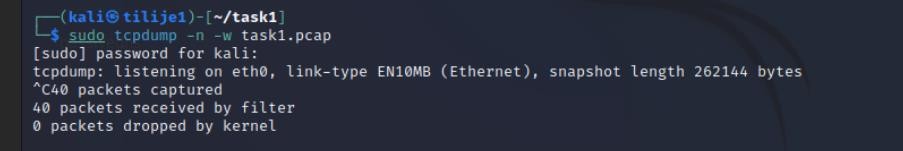
a) What is in the payload of a linux ping?

You will need to capture a ping from a linux host, and capture the traffic .

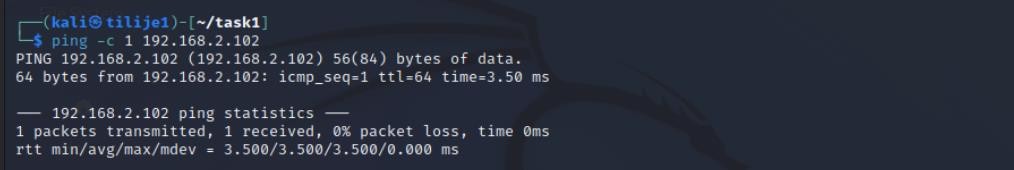
**CAPTURING AND SENDING PACKETS.**

Here we are going to capture and analyze the ICMP Echo Request and Echo Reply packets which are being pinged between two hosts from our local network via tcpdump. We will also perform an in-depth detailed analysis of the packet payload with both hexadecimal and ASCII representations.

The first step is to start the packet capture on kali1 and go to another tab to ping kali2

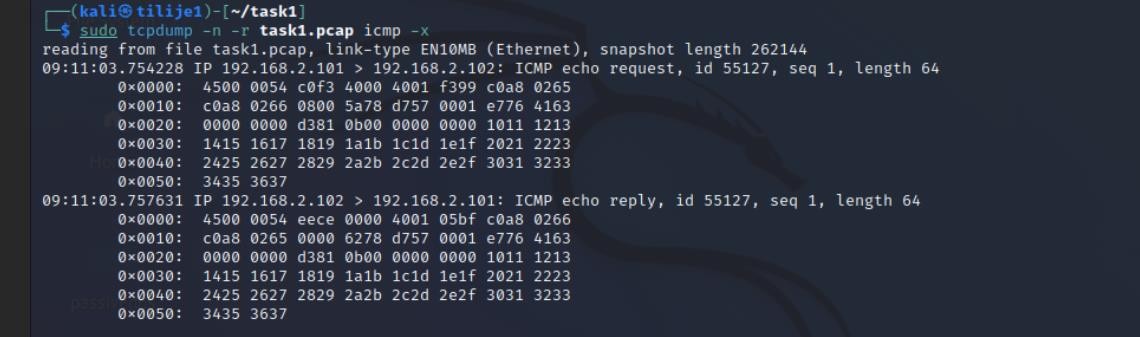


**Fig 1.1: tcpdump capture**



**Fig 1.2: pinging to kali2**

After successfully capturing the packet, we must analyze it



**Fig 1.3: Hex dump**

# ICMP Echo Request Breakdown

**IP Header:**

**4500 0054** is the IP header, and it shows the version and header length. 0f43 4000 4001 f399 signifies the total length and the flags, which identifies the packet as part of the ICMP Echo request. c0a8 0265 is the source IP and c0a8 0266 is the destination IP.

**ICMP Header:**

Type 8 (0X0800) means it is an ICMP echo request. 57a8 signifies the checksum and it is used for error detection. d757 is the identifier and it helps in matching the request and reply. Payload Analysis.

**0x0020: 0000 0000 d381 0b00 0000 0000 1011 1213**

**0x0030: 1415 1617 1819 1a1b 1c1d 1e1f 2021 2223 0x0040: 2425 2627 2829 2a2b 2c2d 2e2f 3031 3233**

**0x0050: 3435 3637**

This represents a 32-byte payload of data and we will breakdown each section;

**0x0020: 0000 0000 d381 0b00 0000 0000** - The first part is the filler data 0000 0000, which may not be used for this packet. **d381 0b00** is another byte sequence. This may be used for tracking or alignment purposes. The next block 0000 0000 is another optional byte sequence or alignment.

**0x0028: 1011 1213 -** This is where the ASCII format of the payload begins. Each byte represents the hexadecimal value corresponding to an ASCII character. In this case, 1011 1213 is increasing in byte values.

**0x0030: 1415 1617 1819 1a1b 1c1d 1e1f** - These bytes continue in order, where 14 represents an ASCII control character (non-printable), 15, 16, and so on follow the same format of their respective byte values. This pattern is repeated across the payload.

**0x0040: 2021 2223 2425 2627 2829 2a2b 2c2d 2e2f -** These

values translate to printable characters ASCII:20 spaces, 21 !, 22 ", 23 # etc. The pattern continues with increasing characters.

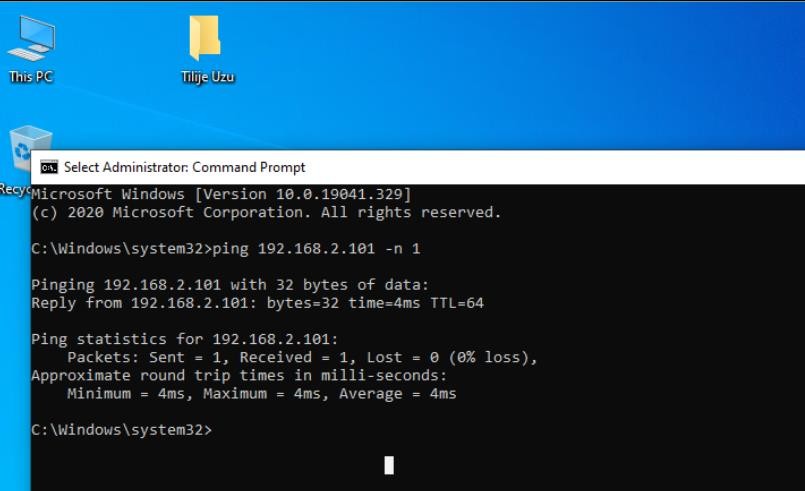
**0x0050: 3031 3233 3435 3637 -** These values correspond to ASCII

characters: 30 0, 31 1, 32 2, 33 3, etc. This continues until 7, creating the final part of the payload.

**b) What is in the payload of a windows ping?**

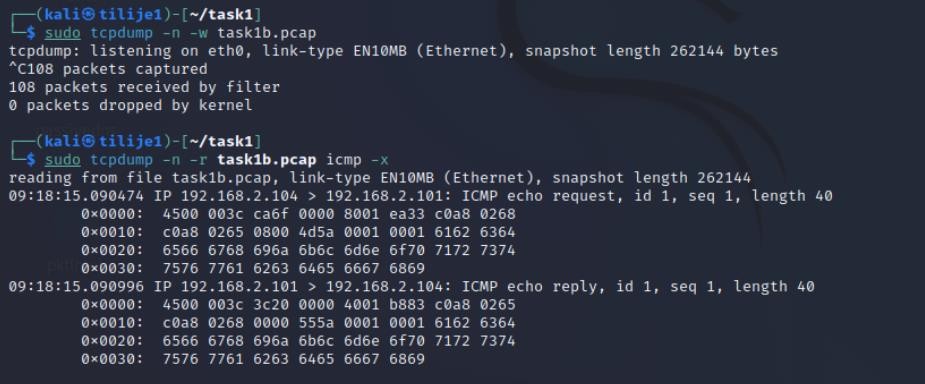
**You will need to ping from a Windows host, and capture the traffic**

To see this, we must ping from the windows machine to kali1 and capture the packets, which is shown below



**Fig 1.4 Ping from Windows.**

After that, we will kill the capture and analyze the windows packet.



**Breakdown;**

1. 4500 003c ca6f 0000 is the IP header.
2. Source IP: c0a8 0268 (192.168.2.104)
3. Destination IP: c0a8 0265 (192.168.2.101)
4. ICMP header: Type (0800): Type 8, ICMP echo request
5. Checksum (4d56): A checksum used for error checking.
6. Identifier (0001): Identifier to match the request/reply.
7. Sequence number (0005): The sequence number of the packet.

**Payload Analysis.**

**0x0020: 6162 6364 6566 6768 696a 6b6c 6d6e 6f70 0x0030: 7172 7374 7576 7761 6263 6465 6667 6869**

**0x0020: 6162 6364 6566 6768 696a 6b6c 6d6f70**

6162 6364 Matches ASCII values abcd and 6566 6768 Matches ASCII values – efgh.

**0x0030: 7172 7374 7576 7761 6263 6465 6667 6869** The rest of the payload continues following the ASCII format from q to z and completes the sequence.

**Key Differences Between Echo Request and Echo Reply (Linux)**

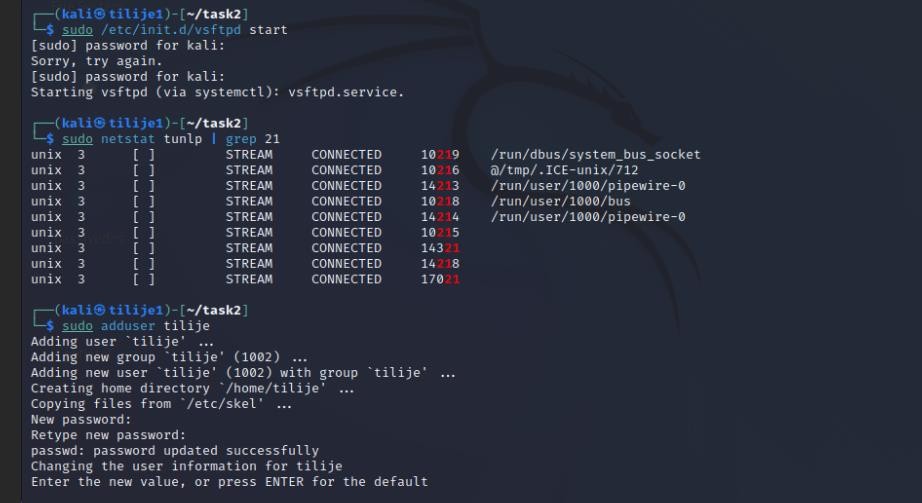
1. **ICMP type**: The request type has 8 (Echo Request) while the reply has type 0 (Echo Reply).
2. **Source and destination IP:** In request 192.168.2.101 source (kali1) 192.168.2.102 destination (kali2) In the reply these roles are reversed: 192.168.2.102. become the source 192.168.2.101 become the destination.
3. **Checksum:** The checksum varies between requests and responses. Each ICMP message recalculates the checksum to ensure data integrity.
4. **Payload:** The payload is the same in the request and response. This is normal for ICMP Ping packets. The purpose of a ping is to ensure that the data sent in the request is the same as the data returned in the response.

Task2.

Introduction

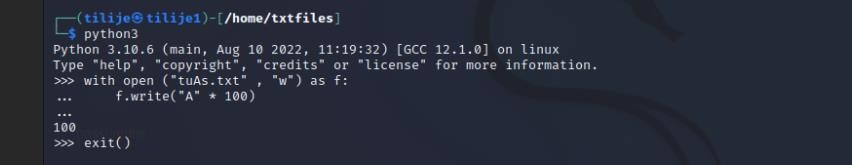
The objective of this task is to analyze clear text FTP sessions between two kali machines. Kali1 is defined as the FTP server and Kali2 acts as the FTP client. When using the vsftpd service on Kali1, we set up an FTP session where the client transfers a file from the server. All packets were captured using tcpdump and later analyzed using Wireshark to observe the data sent. The data includes login credentials and file contents. This report details the steps taken. tools used and insights gotten from the packets.

The first step was to start the FTP service on Kali1 and verify that port 21 is opened as well. After that, the new user was created. These steps are seen below;



## Fig 2.1: Starting FTP and verification

Now from this new user, the new directory was created and a python script that contains 100 As was also made.

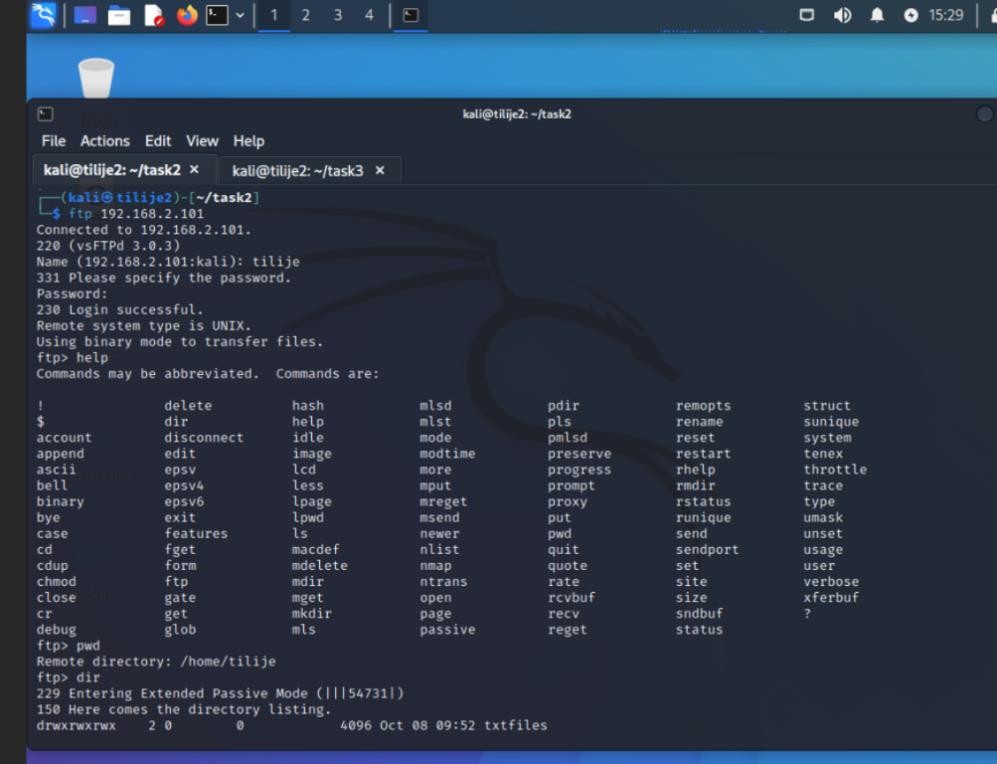


**Fig 2.2: Script to create 100 As**

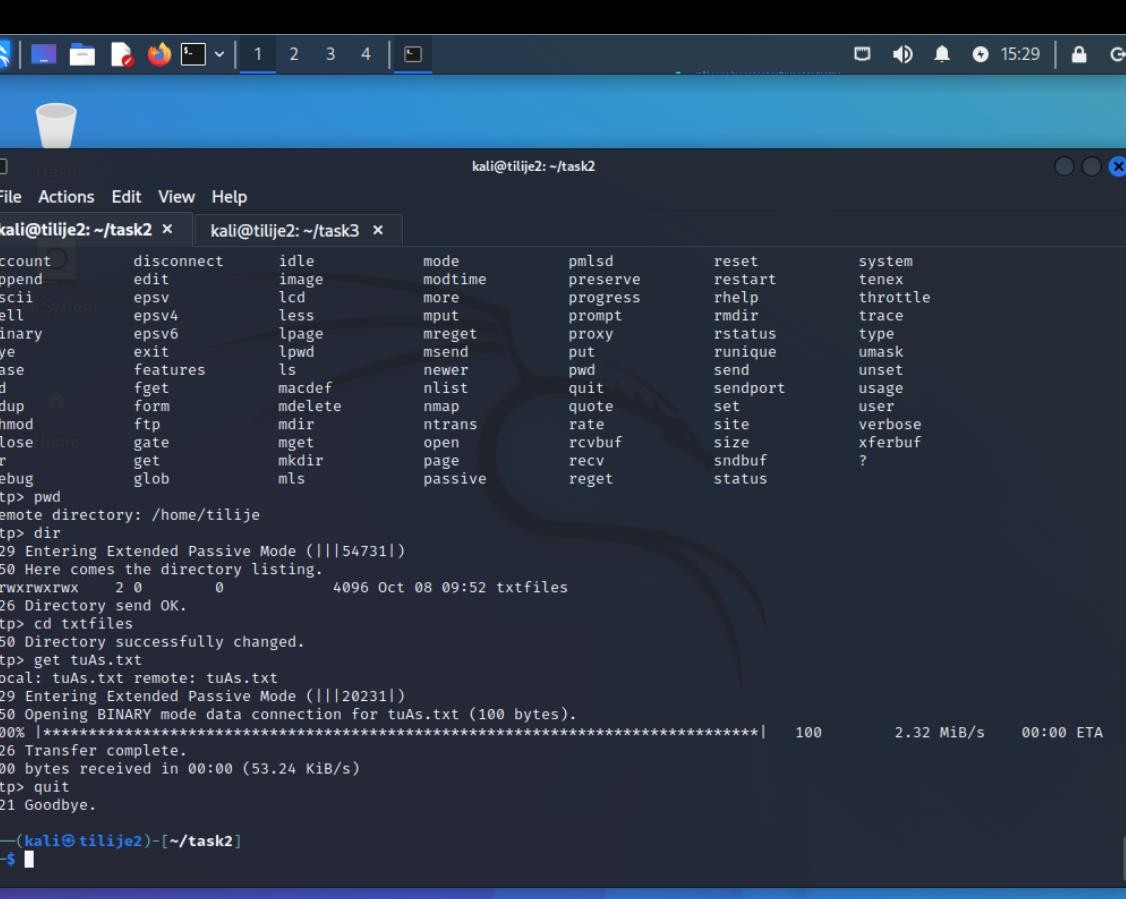
After this is done, we must go to kali2 to initiate the FTP connection. The session will prompt you to type in the username and password, which will be the credentials of the user created in kali1. Note that we must have started a TCP capture session in kali1 before you initiate the FTP session. Once connected via FTP, we must input the following commands;

1. pwd, which checks the current directory
2. help, which shows you a list of commands you may need help in
3. dir to list the contents of the current directory.

After typing the commands we must enter into txtfiles and download the the tuAs.txt file by using the get command.

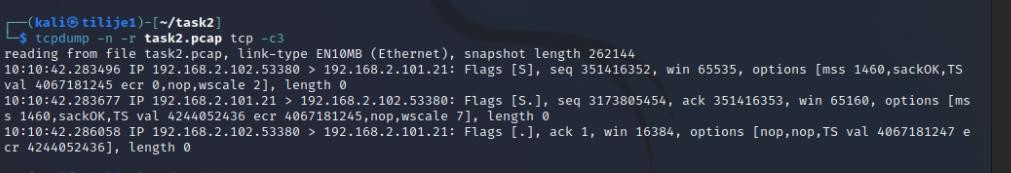


**Fig 2.3: FTP Session**



**Fig 2.4: FTP Session Continued.**

You go back to kali1 to kill the session, then analyze the relevant traffic.



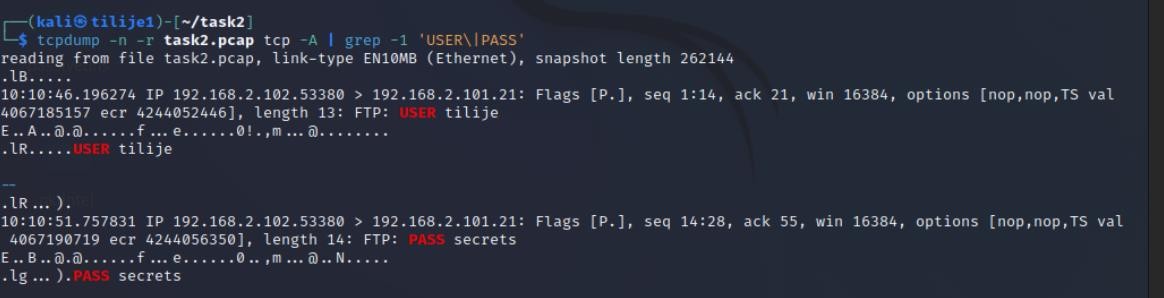
**Fig 2.5: Packet Capture**

From the first packet, the source is **192.168.2.102:53380** (client) attempting to establish a connection to the FTP server at **192.168.2.101:21** (FTP control port). The **[S]** flag indicates that this is a SYN packet, which is part of the three-way handshake that the client sends to the server to establish a connection. The default sequence number **is 351416352**. This sequence number is randomly generated by the client. and will be used to keep track of bytes sent in future communications. The window size is set to 65535, indicating that the client can receive up **to 65,535 bytes** before expecting a response from the server. A client can receive a maximum of 1460 bytes of TCP segment (MSS). The client supports Selective Acceptance, which allows the client to accept inconsistent data. The timestamp is 4067181245 and the Echo reply is 0.The scaling factor is 2, which means the window can be resized by multiplying it by 4. The payload length is 0 because this is the initial packet of the handshake. Therefore, no actual information has been sent yet.

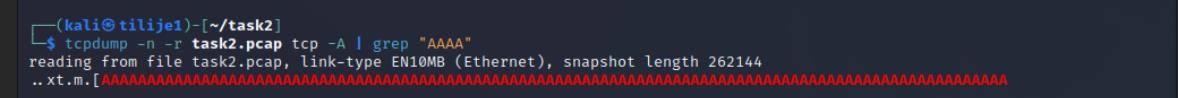
From the second packet, the server (192.168.2.101:21) responds to the client (192.168.2.102:53380) on the FTP control port. The [S.] flag indicates that this is a SYN-ACK packet, which is the server's response to a SYN request from the client. It accepts the connection request and syncs its own sequence number. The server's default sequence number is 3173805454. The acknowledgment number is 351416353, meaning that the server received the client packet and accepted the next byte (client sequence number + 1). The window size is 65160, which means the server can accept up to 65,160 bytes before it needs to be accepted. Unlike SYN packets, this packet does not contain a payload (length 0) as it is part of the handshake.

The third packet signifies the completion of the handshake.

In order to find the username and password as well as finding the file what was transferred, we use the following commands in the screenshot;

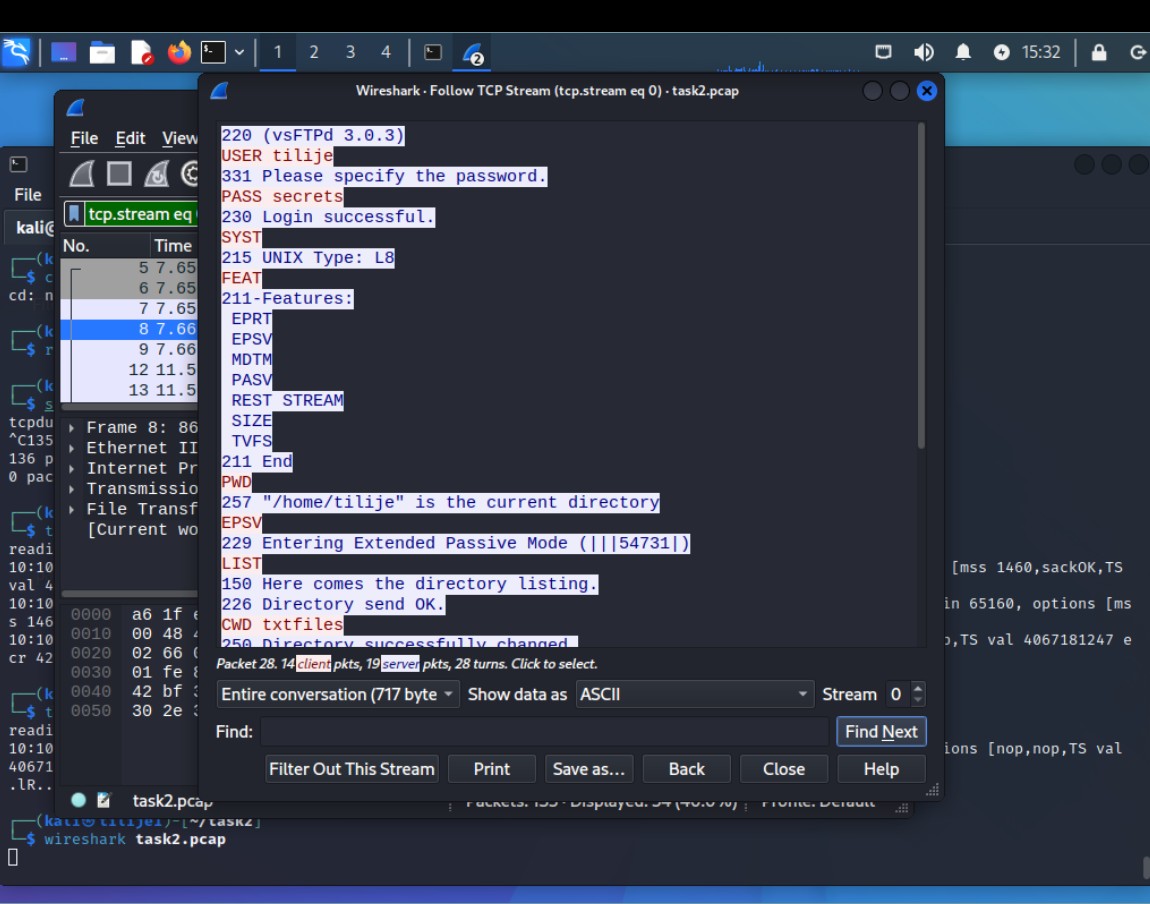


**Fig 2.6: Username and Password with tcpdump**

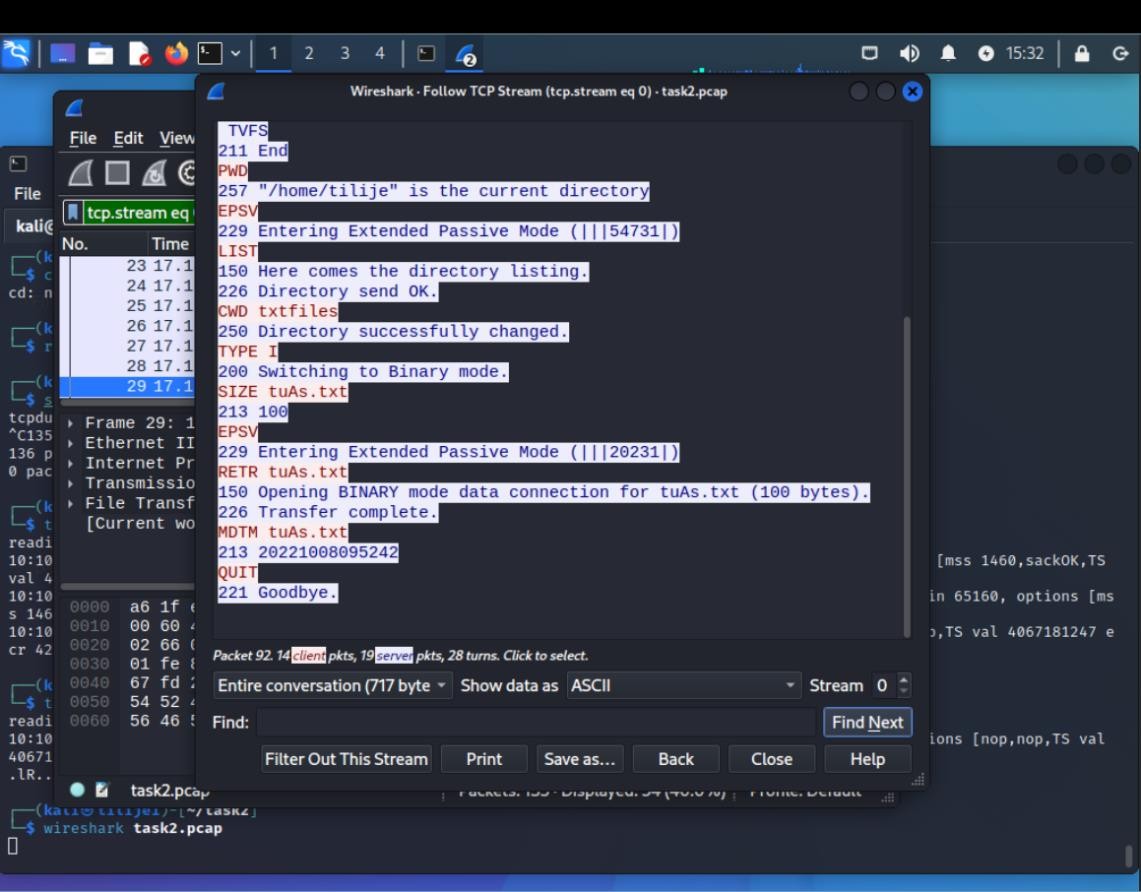


**Fig 2.7: As From tcpdump**

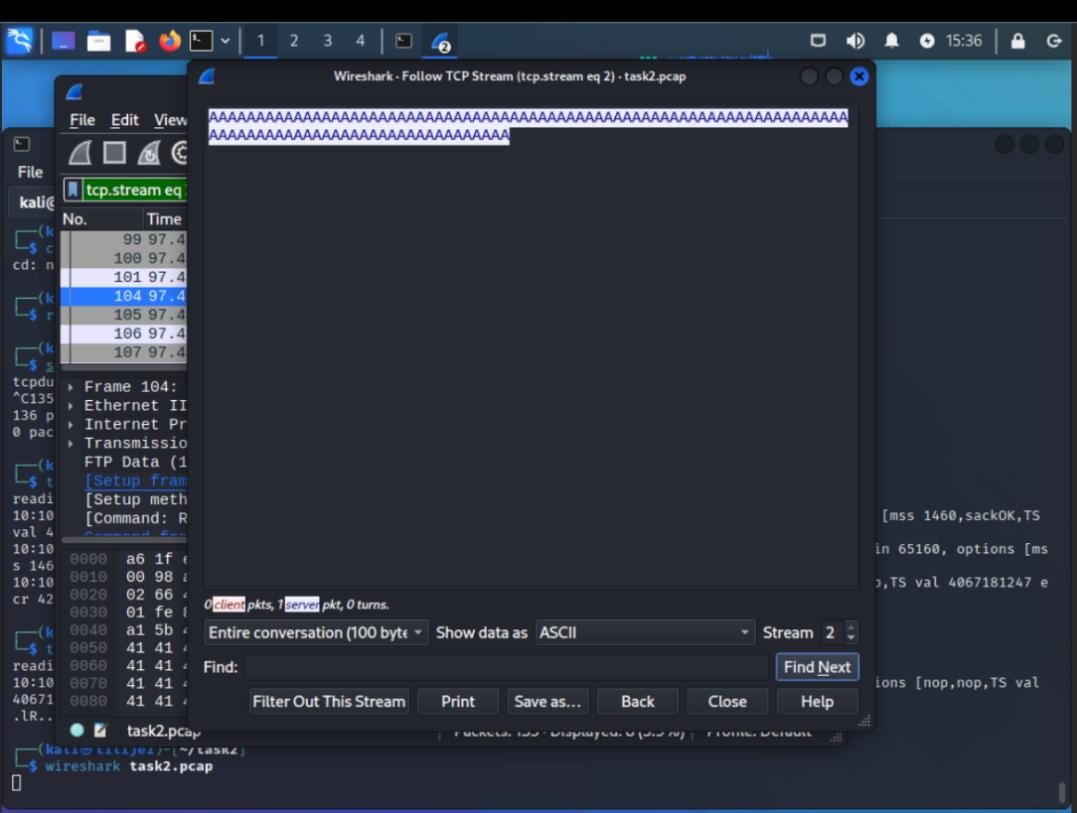
Finally, we use wireshark to follow the FTP stream. This is used to view the entire conversation between the client and the server.



## Fig 2.8: Credentials through wireshark



**Fig 2.9: Credentials through wireshark continued**



**Fig 2:10: As in wireshark**

# Task 3: Header Inspection with TCP Flags and Payload Analysis

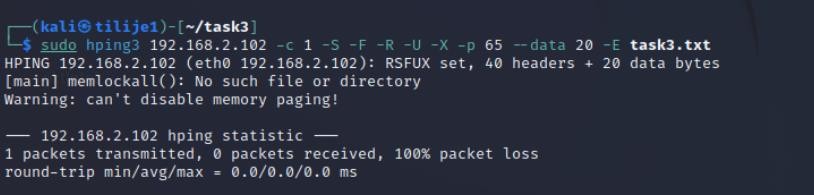
The goal of this task was to transmit a small file using hping3 with certain tcp flags. Analysis of the results of the packet to inspect its TCP and IP headers and payload will be done.

First step to take is to create a file which contains the string “Tilije Uzu”. The file will be used as the data payload in the TCP packet. After doing that, we start your packet capture using tcpdump



**Fig 3.1: Creation of the txt file and starting the capture**

After that we need to send the file using TCP and hping3 to the second kali machine targeting port 65 whilst using specific TCP flags as stated in the question.



**Fig 3.2: using hping3**

**Flags meaning;**

The -S sets the SYN flag and it is used to start a connection

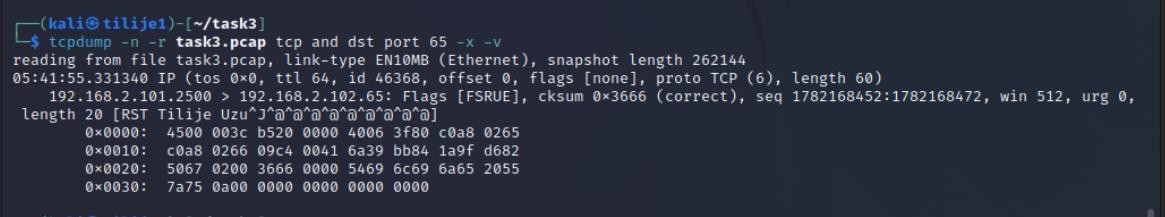
The -F sets the FIN flag and it is used to signal the end of a connection.

The -R sets the RST (reset) flag, which terminates a connection by force.

The -U sets the URG flag, which means it is an urgent data.

The -X sets the ECE flag, and it is used for congestion notification in TCP.

After sending the packet, we must kill the capture and inspect it.



Packet breakdown of the IP Header and TCP segments.

1. Type of service (00 ) indicates that it has a normal priority.
2. Time to live is 64, and it is meant to reduce as the packet moves through routers. A TTL of 64 is the default for most linux systems. In this case, TTL remained at 64 because there is no intermediary routers between the two kali machines.
3. ID is 46368 and it signifies packet’s identification number, mainly used to distinguish packets and fragments.
4. Flags are none because the packet does not have any IP-level fragmentation flags set (no fragmentation).
5. Fragment Offset: is 0, showing that no fragmentation has happened. For a fragmentation to occur, the packet size must exceed the max transmission unit of a network. However, the total byte length is just 60, and it is way below the

MTU.

1. The Flags [FSRUE] indicates the TCP flags that was used.
2. The Sequence Numbers, seq 1782168452:1782168472 indicates that a data of 20 bytes is being sent
3. Window Size: 512 , The window size controls the flow of data, indicating how much data the sender is willing to receive.

**The Hex Dump Analysis.**

**0×0000: 4500 003c b520 0000 4006 3f80 c0a8 0265 0×0010: c0a8 0266 09c4 0041 6a39 bb84 1a9f d682**

**0×0020: 5067 0200 3666 0000 5469 6c69 6a65 2055**

**0×0030: 7a75 0a00 0000 0000 0000 0000**

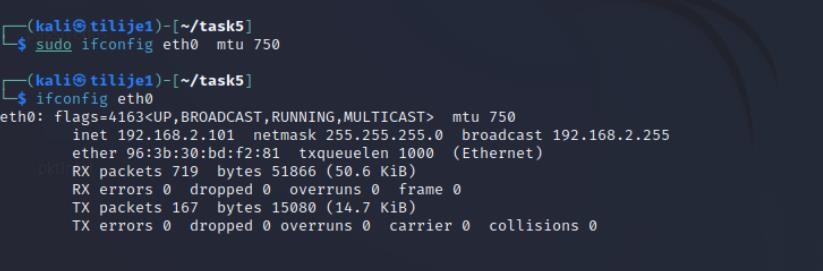
We can see that the TCP and IP headers occupy the first 20 bytes and the payload starts at 0x002c and is also 20 bytes. **5469 6c69 6a65 2055 7a75 0a00** corresponds with the string that was inside the file.

**Task 5 : IPv4 Fragmentation**

**Introduction.**

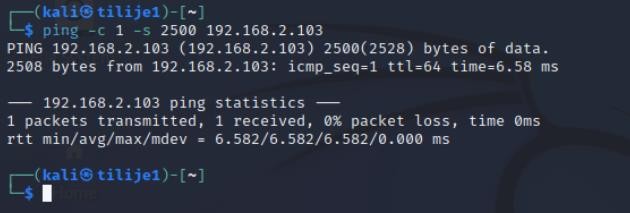
In this task, we are meant to conduct an experiment to investigate how IPv4 fragmentation occurs when sending huge ICMP packets that are larger than a network interface's MTU (Maximum Transmission Unit). Changing the MTU on a Kali virtual machine, using tcpdump to capture the fragmented packets, and examining the various fragments were part of the steps involved. This will be explained in full, as well as the analysis of each fragment in depth, emphasizing important details such fragments offset, checksum, flags, and how the pieces vary from one another.

First step was to set the MTU to 750, which will be done using kali1



## Fig 5.1: Changing MTU to 750

**Th**e second step was to start the packet capture in kali1 and then send a large ping to kali2 to trigger fragmentation. After sending the ping, we kill the capture and analyze the pcap file.

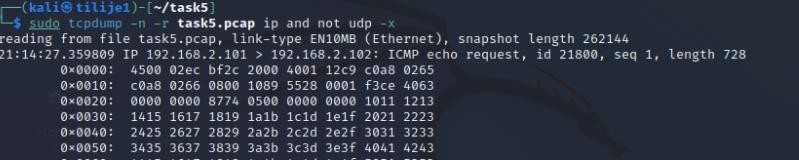


# Fragmentation Overview

When a packet surpasses the MTU (750 in this case). IPv4 breaks the packets into smaller pieces. Each fragment holds a piece of the original packet, and it is put back together at the destination. There are key fields that aid management in fragmentation, these may include Identification, Fragment Offsets, Flags etc.

**Fragments Analysis.**

Now, let’s breakdown and analyze each of the fragments captured **Fragment 1: ICMP Echo Request.**

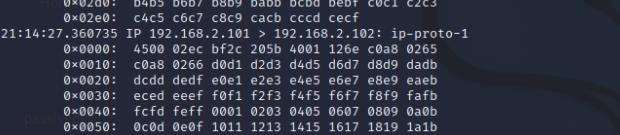


**Fig: 5.2: First Fragment**

The packet’s total length (02ec) is 748 bytes long (728 bytes of data and a 20-byte IP header), the identification (bf2c) is used to show the fragments of the same packet.

The flags and Fragment offset (2000) indicate that more fragments follow and the TTL (40) is 64. The checksum of the IP header is 0x12c9.

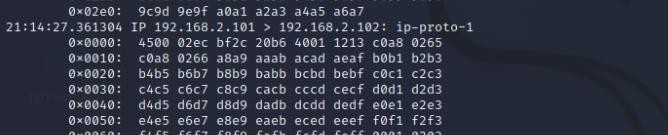
**Fragment 2**



**Fig: 5.3: Second Fragment**

The total length (02ec) is the same as the first fragment (748 bytes). The identification (bf2c) matches the first fragment, which indicates that it is part of the original packet. More Fragments is still still set and the fragment offset 205b indicates that this is the second fragment of the packet, this means the first fragment carried 92 bytes and this one followed it. This fragment also has the next portion of the ICMP echo request data (d0d1, d2d3…).

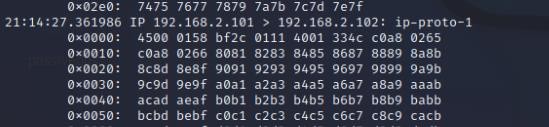
**Fragment 3.**



**Fig: 5.3: Third Fragment**

The identification (bf2c) matches the first two fragments and the fragment offset(20b6) starts after the first 182 bytes.

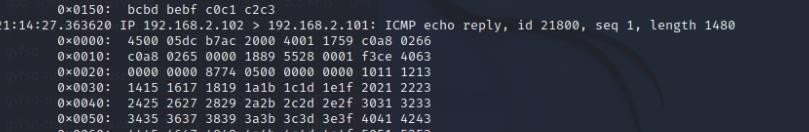
**Fragment 4.**



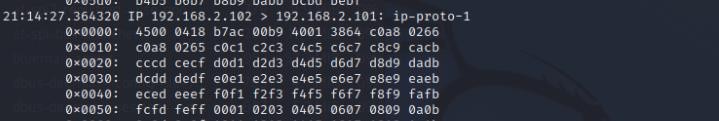
**Fig 5.4: Fourth Fragment**

Fragment 4 is the last fragment of the ICMP Echo Request. The total length (0158) is 344 bytes which indicates smaller fragments. The MF is cleared (0) which signals the end of the packet and the offset (0111) of 273 shows that this fragment completes the packet.

**Fragment 5.**



The identification (b7ac) is new for the ICMP Echo Reply packet and it shows that more fragments is expected.



The MF is set to 0, indicating that this is the last fragment of the reply.

How Reassembly Occurs

The Reassembly process occurs at Kali2's IP layer (192.168.2.102). After each block is received, the destination device's IP layer (Kali2) collects and buffers the incoming fragments. It then uses several key fields in the IP header, such as Identification, Fragment Offset, and More Fragments (MF), to correctly reconstruct the original ICMP packet. Task 5.

Detailed reassembly steps

1. To receive fragments at destination, Kali2 starts receiving fragmented packets from Kali1. Each fragment is processed at the IP layer. The destination machine uses the Identification in the IP header to make sure that the fragments belong to the same original packet.
2. **Specifying buffer and fragment sequences**: Each fragment has an Identification equal to 0xbf2c which is the same for all fragments. This indicates that all fragments are in the same packet. The Fragment Offset field is important in determining the position of each fragment within the original packet. For example, Fragment 1 offset is 0 fragment 2 has an offset of 92, fragment 3 has an offset of 182 and so on. Kali2 buffers these fragments and uses the Fragment Offset field to arrange them in the correct order for reassembly.
3. **More Fragments flags (MF**): The MF flag tells Kali2 whether to expect more blocks. In fragment 1, the MF flag is set to 1, which means there are more fragments. The same flag is set in the next blocks (Block 2, Block 3). However, in block 4, the MF flag is set to 0, indicating that it is the last fragment in the sequence. This tells Kali2 that no more fragments will arrive.

1. **Reassembling the original packet:** Once all the fragments are received,

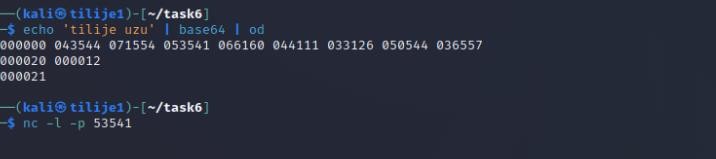
Kali2 uses the Fragment Offset value to merge the original 2,500 byte ICMP Echo request. The Fragment Offset field helps ensure that the fragments are sorted correctly, Fragment 1 starts at offset 0. Fragment 2 starts at offset 92, Fragment 3 starts at offset 182, Fragment 4 completes the packet.

1. **Checksum confirmation: During** reassembly, Kali2 verifies the integrity of each segment's IP header by checking the header checksum. If the block contains corrupted headers (such as an incorrect checksum), the entire packet can be discarded. In this case, we found that there were inconsistencies in the information provided across certain groups in the review. This may affect the reassembly process, however, assuming the checksum is correct, the packet is successfully reassembled.
2. **ICMP layer processing**: After the IP layer reassembles the original ICMP packets, It is then forwarded to the ICMP layer for processing. The ICMP layer recognizes the packet as an Echo request, telling Kali2 to create an ICMP Echo Reply. The Echo Reply is then disassembled (if necessary) and sent back to Kali1.

# Task 6: TCP Handshake, Data Transmission, and Connection Closure Analysis

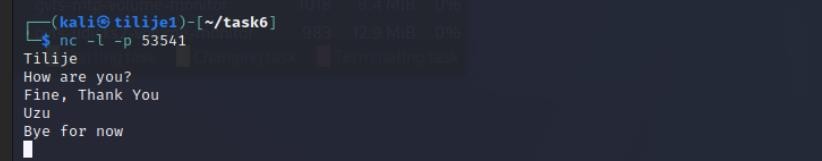
In this task, we are to perform a sequence of operations that involve a three-way TCP handshake, transmitting data between two kali machines. and properly terminating the connection. This task will show you how to establish a TCP connection between the client and the server. Below we break down the important points, including TCP flags, sequence numbers, acknowledgment numbers, and more.

First step was to encode a small string, in this case my full name, to try and simulate sending encoded data between two hosts. After that, we pick a port and set up the netcat listener for our other kali OS.

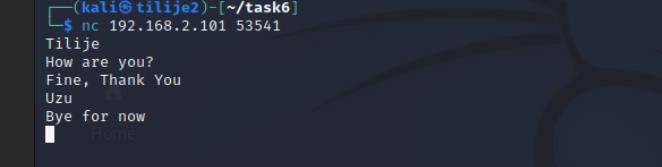


## Fig 6.1 My name in base 64

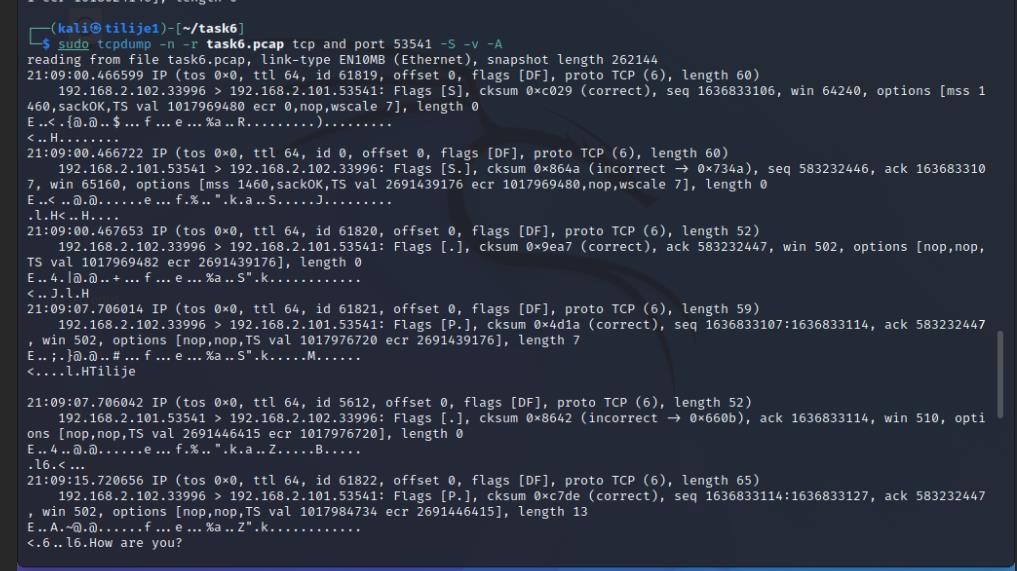
On kali2, we connect to the Netcat listener sent by kali1 by specifying the ip address and also the port. After that we can start use tcpdump to capture the packets in the conversation. The conversation between both of them can be seen below



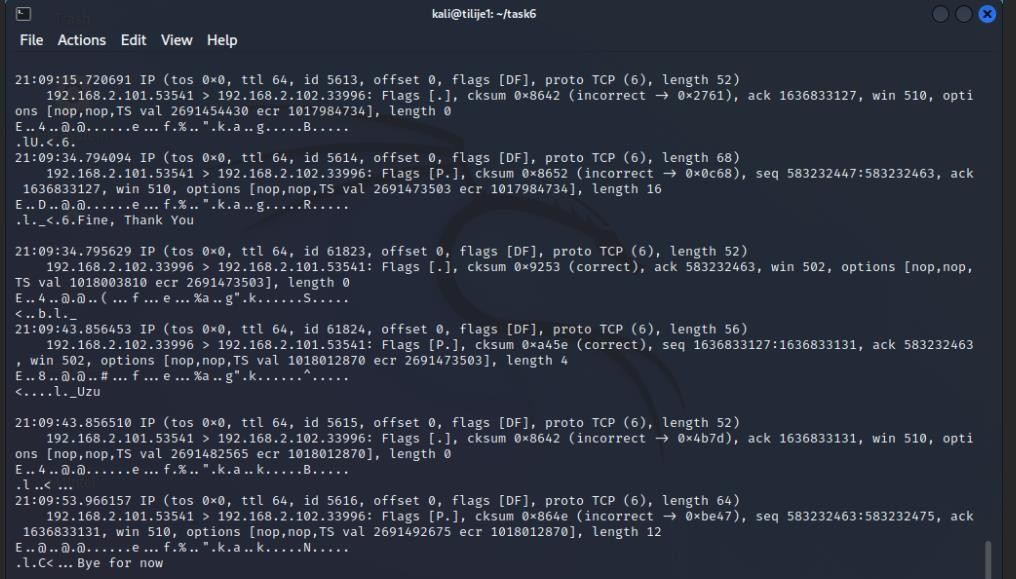
## Fig 6.1 Netcat session

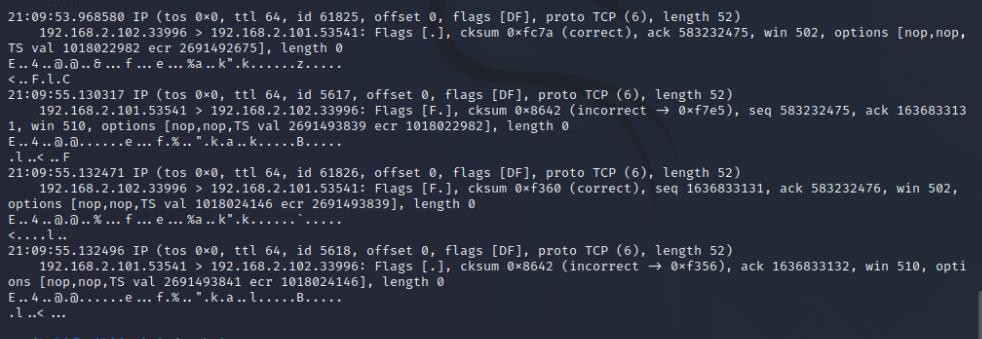


After capturing the session we can view, analyze the packets and filter the relevant traffic using the command that will be shown in the screenshot below.



## Fig 6.2 Packet Analysis





# Comprehensive Analysis of the Packet Exchange

The first thing that happens here is that the client (tilije2) starts initiating the TCP connection to the server (tilije1). The client sends a SYN packet to the server. The server acknowledges this request by responding with a SYN-ACK, signaling that it is ready to establish a connection. Finally, the client completes the connection (threeway handshake) by sending an ACK packet.

After the handshake is complete, the server and the client can begin to exchange data between each other. We can see from the packet that different messages such as

“Tilije”, “How are you” were sent between tilije1 and tilije2. Each data that was transmitted has an acknowledgement. This shows that what was sent, was received.

After the data exchange is finished, the server sends a FIN packet to signal client that it wants to close the connection and the client acknowledges this by sending an ACK. The client then sends its FIN packet, to show that it wants to send its connection as well. The server acknowledges the last FIN flag with an ACK, thereby terminating the process.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sr**  **No**  **.** | **Source IP** | **Destinatio n IP** | **Flag**  **s** | **Sequence No.** | **Acknowledgm ent No.** | **Activity** |
| 1 | 192.168.2.1  02 | 192.168.2.1  01 | SY  N  (S) | 1636833106 | 0 | Client initiates connection (SYN) |
| 2 | 192.168.2.1  01 | 192.168.2.1  02 | SY  NAC  K | 583232446 | 1636833107 | Server responds with SYN-  ACK |
| 3 | 192.168.2.1  02 | 192.168.2.1  01 | AC  K | 1636833107 | 583232447 | Client acknowled ges SYNACK  (Handshak e complete) |
| 4 | 192.168.2.1  02 | 192.168.2.1  01 | PSH  -  AC | 1636833107:163683  3114 | 583232447 | Client sends data: "Tilije" (7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | K |  |  | bytes) |
| 5 | 192.168.2.1  01 | 192.168.2.1  02 | AC  K | 583232447 | 1636833114 | Server acknowled ges receipt of "Tilije" |
| 6 | 192.168.2.1  02 | 192.168.2.1  01 | PSH  -  AC  K | 1636833114:163683  3125 | 583232447 | Client sends data:  "How are you?" (11  bytes) |
| 7 | 192.168.2.1  01 | 192.168.2.1  02 | AC  K | 583232447 | 1636833125 | Server acknowled ges receipt of "How  are you?" |
| 8 | 192.168.2.1  01 | 192.168.2.1  02 | PSH  -  AC  K | 583232447:5832324  63 | 1636833125 | Server sends data:  "Fine,  Thank  You" (16  bytes) |
| 9 | 192.168.2.1  02 | 192.168.2.1  01 | AC  K | 1636833125 | 583232463 | Client acknowled ges receipt of "Fine,  Thank  You" |
| 10 | 192.168.2.1  02 | 192.168.2.1  01 | PSH  - | 1636833127:163683  3131 | 583232463 | Client sends data: |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | AC  K |  |  | "Uzu" (4  bytes) |
| 11 | 192.168.2.1  01 | 192.168.2.1  02 | AC  K | 583232463 | 1636833131 | Server acknowled ges receipt of "Uzu" |
| 12 | 192.168.2.1  01 | 192.168.2.1  02 | PSH  -  AC  K | 583232463:5832324  75 | 1636833131 | Server sends data:  "Bye for now" (12  bytes) |
| 13 | 192.168.2.1  02 | 192.168.2.1  01 | AC  K | 1636833131 | 583232475 | Client acknowled ges receipt of "Bye for now" |
| 14 | 192.168.2.1  01 | 192.168.2.1  02 | FIN-  AC  K  (F) | 583232475 | 1636833131 | Server initiates connection termination  (FIN) |
| 15 | 192.168.2.1  02 | 192.168.2.1  01 | FIN-  AC  K | 1636833131 | 583232476 | Client acknowled  ges server's  FIN |
| 16 | 192.168.2.1  01 | 192.168.2.1  02 | AC  K | 583232476 | 1636833132 | Server acknowled  ges client's  FIN |

(

Connectio

n closed)

**Table Analysis.**

**Phase one (Three-Way Handshake)**

The first packet starts the connection by sending the SYN packet from the client to the server. The sequence number present is the Initial Sequence Number (ISN) of the client (1636833106) and it is a random number that is chosen when the communication is initiated. The acknowledgment number is 0 because no data has been received yet.

The second packet responds to the SYN packet with a SYN-ACK packet, to acknowledge the clients request. The sequence number (583232446) is also randomly chosen by the server (ISN). The acknowledgement number is the clients ISN + 1, hence making it 1636833107. This informs the client that the server is ready to communicate because it has received its SYN packet successfully.

The third packet finishes the handshake by sending an ACK packet to the server. The sequence number remains the same because no data has been sent yet. The acknowledgement number is the server’s ISN +1, making it 583232447. This simply means that the client has gotten the server’s STN-ACK Packet.

# Phase two (Data Transmission)

The fourth packet sends “Tilije” (7 bytes), and because the client is sending data, the PSH-ACK flag is initiated. The sequence number is 1636833107:1636833114 and this range shows that the client started sending data from byte 1636833107 and it ends at 1636833114. As we can see that the difference between them is 7 bytes, which indicates that the data that was sent (“Tilije”) was indeed 7 bytes. The acknowledgment number remains the same because no data has been gotten from the server yet.

The fifth packet acknowledges (ACK) the data “Tilije”, the sequence number (583232447) remains the same because the server is not sending any data in the packet. The acknowledgment number is 1636833114 because it shows that the server has gotten the 7 bytes which was sent by the client and is now expecting the next bytes, which starts at 1636833114. In summary, the acknowledgment number increases by the amount of data sent ( 7 bytes).

The sixth packet shows the client send “How are you?” (PSH-ACK) which is 11 bytes. The sequence number is 1636833114:1636833125 which shows the range.

1636833114 starts right after “Tilije” and ends at 1636833125. This difference shows that a difference of 11 bytes was sent.

The seventh packet acknowledges (ACK) the message sent by the client, sequence number remains unchanged, and the acknowledgement number now becomes 1636833125 because the server acknowledges the 11 bytes sent from the client.

The eighth packet shows the server sending “Fine, Thank You” (PSH-ACK) which is 16 bytes. The sequence number is 583232447:583232463 . The server sends 16 bytes of data, and the acknowledgement number remains the same because there is no data from the client.

In the ninth packet, the client acknowledges (ACK) the “Fine, Thank You”, sequence number remains the same and the acknowledgement number becomes 583232463 because the client has acknowledged all the 16 bytes from the server.

In the tenth packet, client sends “Uzu” (PSH-ACK) which represents 4 bytes, hence making the the sequence number have a range of 1636833125:1636833131.

The server acknowledges the “Uzu” sent from the client in the eleventh packet, no new data is sent so sequence number remains the same and the 4 bytes from the client is being recognized, hence changing the acknowledgement number to 163683313

The server sends “Bye for now” (PSH-ACK) in the twelfth packet and it represents 12 bytes, hence making the sequence number 583232463:583232475

The client acknowledges the “Bye for now” hence changing the acknowledgement number to 583232475

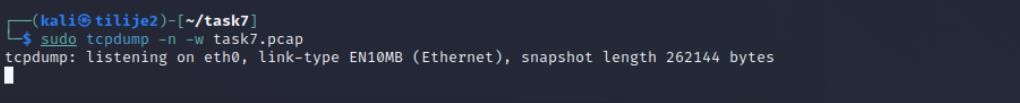
**Phase Three ( Connection Termination )**

The server initiates the connection termination (FIN-ACK),The sequence number remains the same and the server acknowledges all the data received from the client. The client, recognizes the FIN sent from the server and then send its own FIN, keeping the acknowledgement number at 583232476. Finally, The server acknowledges the clients FIN, finishing the end of the connection.

# Task 7: Use hping to communicate with a netcat listener over TCP

The goal of Task 7 is to simulate TCP communication between two hosts. Specifically simulating the behavior of "push" operations within the TCP Stack, the task uses netcat as the basis of the TCP client/server connection between client (Kali2) and server (Kali3). TCP dump will be used for analysis.

First step was to start the tcpdump capture, for the netcat session in kali2 and then start the netcat session in kali3.

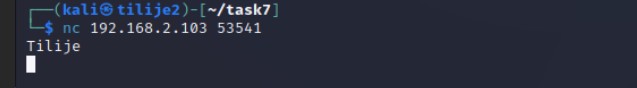


## Fig 7.1 Starting tcpdump capture

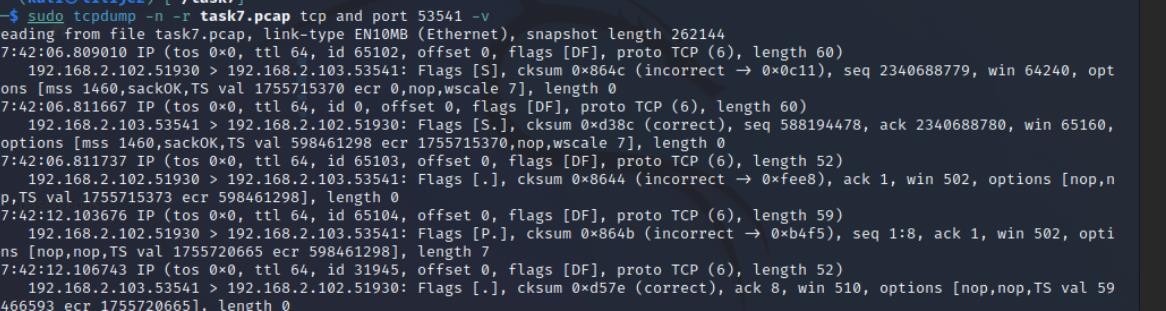


## Fig 7.2 Starting netcat session

After it has started we must send my first name to in the netcat session and analyse the capture while the netcat session is still running.



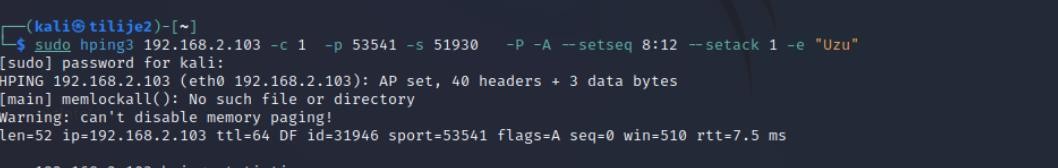
**Fig 7.2 Starting netcat session**



## Fig 7.2 Packets on the session

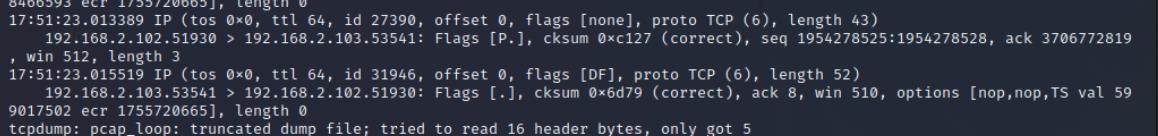
What we are focusing on is the PSH-ACK flag because it is from there that we will attempt to imitate sending my last name. Let’s focus on the sequence and acknowledgement number to do this.

As we can see the sequence number is 1:8, which signifies that my name “Tilije” was 7 bytes and the ISN is 1. In order to imitate what hs happened here, we can use the command below;



**Fig 7.3 Command to imitate the push.**

We try to simulate the push process by adding the bytes of “Uzu” + the previous seq number.



**Fig 7.4 Second push**

We can see that the second push was sent, but for some reason the sequence number that is generated is different and the second name was not sent between the two kali machines. This may occur due to the several reasons, one of them can because the TCP stack on the server does not recognize the sequence, hece leading to unexpected behaviors

# Task 8 : Paper Based Exercise

For all of these tasks, use the [] notation that we covered in "Filtering by bytes and bits".

This exercise is paper based only.

**A) Write a tcpdump filter to select TCP packets whose Acknowledgement**

**Number is less than 2047453745.**

**Introduction:**

In a TCP packet, the Acknowledgment Number is a 32-bit field that may be found in the TCP header at byte offset 8. The objective is making a tcpdump filter that only includes TCP packets with an Acknowledgment Number smaller than 2047453745 or, in hexadecimal, 0x7A1F4911. In order to achieve this we can use ;

**'tcp[8:4] < 0x7A1F4911'**

**Explanation:**

The Acknowledgment Number starts at byte 8 and spans 4 bytes. (**tcp[8:4]**).We compare the hexadecimal value of 2047453745 (0x7A1F4911) with the 32-bit value present in these 4 bytes. The tcpdump filter directly filters based on the byte offsets by using the [] notation. Also, using **< 0x7A1F4911** ensures that we only capture packets whose acknowledge number is not more than 2047453745.

**B) Write a tcpdump filter to select TCP packets whose header length is greater than 20 bytes.**

**Introduction:**

**T**he Data Offset field in the TCP header indicates the length of the TCP header.This value is stored in the first 4 bits of byte 12 and represents the number of 32-bit words in the header. This field can have a minimum value of 5, or 20 bytes. The packets with header lengths longer than 20 bytes are the ones we wish to choose. To achieve this we use;

**'tcp[12] >> 4 > 5'**

**Explanation:**

The length of a TCP header is expressed in 32-bit words (4 bytes). When the header length exceeds 20 bytes, the Data Offset field will have a value larger than 5. The filters **tcp[12] >> 4** isolates the header length (Data Offset field) by shifting the byte at offset 12 right by 4 bits and  **> 5** makes sure that we only record packets with header lengths more than five, or more than twenty bytes.

**C) Write a tcpdump filter to select TCP packets which have FIN and URG turned on. Other bits in the flags field may also be on.**

**Do not use items like ‘tcpflags’, ‘tcp-fin’, ‘tcp-urg’, etc. in answering this.**

**Introduction:**

The TCP flags are kept in byte 13 of the TCP header. Each bit in this byte represents a different flag (such as SYN, ACK, FIN, URG, etc.). Making a filter that only includes packets with both the URG and FIN flags enabled (other flags can be either on or off) is the objective of this task. In order to achieve this, we can use this filter;

**'tcp[13] & 0x21 = 0x21'**

**Explanation:**

The first bit (bit 0) of byte 13 has the FIN flag and the URG flag is at the 6th bit (bit 5) of byte 13. The filters **cp[13]** picks the offset 13 byte (which contains the TCP flags) & **0x21** is used to carry out a bitwise AND operation (where **0x20** aligns with the URG flag and **0x01** with the FIN flag). = **0x21** this makes sure that while other flags may be in any state, it guarantees that the FIN and URG flags are both set.

D) Write a tcpdump filter to select DNS packets whose AA is 1 and RA is also 1.

You need to use the UDP header to achieve this.

Assume that all UDP port 53 traffic in either direction is DNS.

You’ll need to check that the port number in either direction is 53. Do NOT use dns[] in answering this **Introduction:**

DNS mainly uses port 53 for UDP operations. The DNS header contains the Authoritative Answer (AA) and Recursion Available (RA) flags, which are found in the flags field. Creating a tcpdump filter that chooses DNS packets with both the AA and RA flags set to 1 is the goal here. To do this, we will employ byte-level filtering and the UDP header.

**'udp port 53 and udp[10] & 0x04 != 0 and udp[11] &**

**0x80 != 0'**

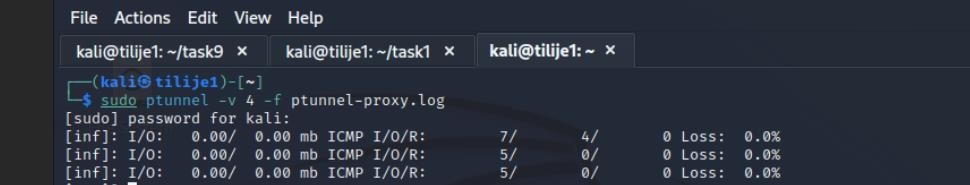
**Explanation:**

The AA flag can be found in the sixth bit of the third byte, which is byte 10 in the packet (starting from the beginning of the UDP header), **udp[10] & 0x04 != 0** checks if the AA flag is set to 1. The 8th bit of the 4th byte (byte 11 in the packet, counting from the beginning of the UDP header) contains the RA flag, **udp[11] & 0x80 != 0** checks if the RA flag is set to 1. To make sure that both flags are set and choose UDP traffic on port 53, we add **udp port 53** to the filter.

# Task 9 : ptunnel

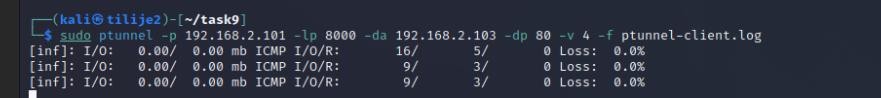
In this task, we aim to hide and signal normal HTTP traffic inside ICMP packets using a tool called ptunnel. This demonstrates that TCP traffic, such as web requests, can be encapsulated in ICMP packets, which are often used for pings and other diagnostic purposes. By leveraging ICMP, we can bypass network restrictions or firewall rules that block TCP traffic, but allows ICMP.

To do this we must set a packet capture and proxy in kali1, which is seen below;



**Fig 9.1 Kali1 proxy**

In kali2, we will need to start the ptunnel for the client as well.



## Fig 9.2 Client ptunnel

The next step will be to go to the firefox tab to enter 127.0.0.1:8000, we are doing this

to get the “Apache on kali Linux” written in kali3

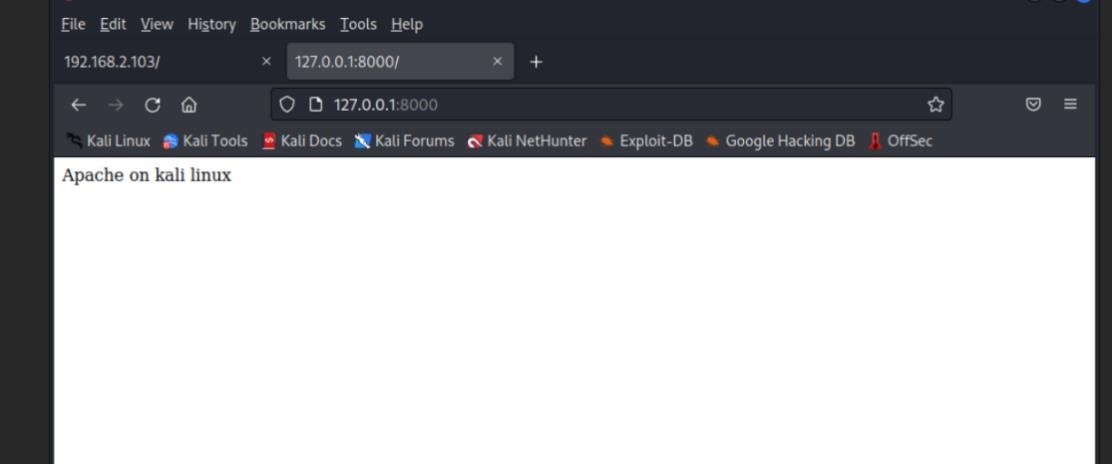
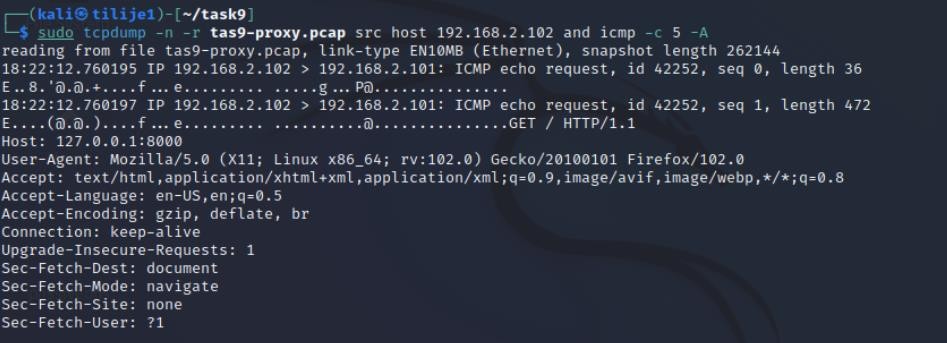


Fig 9.2 Firefox output.

After that has been done, we must kill the capture and analyze the proxy file that was captured and spot the ptunnel.



**Fig 9.3 Ptunnel analysis**.



# Packet capture analysis

In this capture, we have an ICMP echo request containing an HTTP GET request which is sent to the Kali3 web server via ptunnel. Let's take a look at the details:

1. ICMP Echo request with HTTP data - **Packet 18:20:40.927874 IP**

## 192.168.2.102 > 192.168.2.101: ICMP echo request, id

**42252, seq 0, length 36** , this is typically used for ping operations that has HTTP request data. Host IP 192.168.2.102 (Kali2, client) sends ICMP packet to 192.168.2.101 (Kali1, acting as proxy). The ICMP packet ID is 42252 with sequence number 0. Length of packet T is 36, which indicates a short initial packet. We can see the HTTP headers inside the ICMP echo request **Host: 127.0.0.1:8000 User-Agent: Mozilla/5.0**, This indicates that this ICMP packet contains an HTTP GET request directed to 127.0.0.1:8000 (

Kali3's web server), but clients access it locally through the ptunnel proxy on Kali1. Essentially, Kali2's HTTP requests are contained within this ICMP packet.

**2.** ICMP Echo Request for favicon.ico: **18:20:41.013374 IP**

**192.168.2.102 > 192.168.2.101: ICMP echo request, id**

## 42252, seq 2, length 410

Here we can see another ICMP echo request from kali2 to kali1, but this time the is larger (410) which indicates that more data in the HTTP request is being carried out. The payload contains another HTTP GET request GET

**/favicon.ico HTTP/1.1.** This is put in an ICMP packet through the p tunnel proxy.

**How does Ptunnel work in this context?**

1. Proxy Settings: We have configured ptunnel on Kali1 to act as a proxy, this means that any ICMP packets arriving on Kali1 will be on port 80 in Kali3
2. Client Communication (Kali2): On Kali2, we instructed the browser to connect to 127.0.0.1:8000 , which was the local end point of the tunnel. It is then put in an ICMP packet and sent to Kali1. Kali1 forwards the put together TCP traffic to Kali3.
3. ICMP Encapsulation: The ptunnel tool receives HTTP requests from Kali2 and hides them within the payload of ICMP echo request packets. These ICMP packets are then sent to the actual destination by the ptunnel proxy on Kali1, which is Kali3 web server.
4. Response Handling: When Kali3's web server responds, Kali1 merges with the HTTP response in an ICMP Echo response packet and sends it back to Kali2 where the traffic is opened. and the browser receives the content of the web page as if it were accessed directly.

# Task 10 : TCP Packet [Paper based task]

Consider the following TCP/IP packet:

0x0000: 4500 0040 7111 4000 4006 6bc7 ac14 0a02

0x0010: 34d7 72f2 89ea 01bb 6338 5f4c 415f eb59

0x0020: b010 01a2 82ff 0000 0101 080a 3a11 ae31

0x0030: 1d3f 844e 0101 050a 415f eb58 415f eb59

What is in this packet?

In particular, what TCP Options are being used, and what are their values?

Explain how you arrived at your answers.

Note that the class notes do not fully answer this.

We were tasked to analyze the following TCP/IP packet:

**0x0000: 4500 0040 7111 4000 4006 6bc7 ac14 0a02**

**0x0010: 34d7 72f2 89ea 01bb 6338 5f4c 415f eb59**

**0x0020: b010 01a2 82ff 0000 0101 080a 3a11 ae31**

**0x0030: 1d3f 844e 0101 050a 415f eb58 415f eb59**

The packet comprises of TCP/IP headers, along with TCP Options, We are going to analyze the contents of each header and decode the TCP options that are being used.

First step we are taking is breaking down the IP header. An IPv4 Header is 20 bytes long, that being said, we can extract the following fields from the packets (bytes 0x0000 to 0x0014). We can also identify that this is an 1Pv4 packet using the TCP protocol.

**0x0000: 4500 0040 7111 4000 4006 6bc7 ac14 0a02**

**0x0010: 34d7 72f2**

|  |  |  |
| --- | --- | --- |
| **Field** | **Value** | **Description** |
| **Version + IHL** | 4500 | IPv4, Header Length is equal to 5 words (20 bytes) |
| **Total Length** | 0040 | Total length of the packet is 64 bytes |
| **Identification** | 7111 | Identification field for packet fragmentation |
| **Flags + Offset** | 4000 | Flags: Don’t Fragment (DF), Fragment Offset: 0 |
| **TTL + Protocol** | 4006 | Time To Live (TTL) = 64, Protocol = 6 (TCP) |
| **Checksum** | 6bc7 | IP header checksum |
| **Source IP** | ac14  0a02 | Source IP = 172.20.10.2 |
| **Destination IP** | 34d7  72f2 | Destination IP is 52.215.114.242 |

Second step is to breakdown the TCP Header, In normal cases, a TCP header is also 20 bytes long ,but in this case, the TCP header includes options and this is making it longer than 20 bytes. The next 20 bytes, beginning from byte 20 (after the IP header) , shows the TCP header. This can be seen as;

**0x0014: 89ea 01bb 6338 5f4c 415f eb59 b010 01a2**

**0x0020: 82ff 0000**

|  |  |  |
| --- | --- | --- |
| **Field** | **Value** | **Description** |
| **Source Port** | 89ea | 35242 is the Source port |
| **Destination Port** | 01bb | Destination port is 443 (HTTPS) |
| **Sequence Number** | 6338  5f4c | Sequence number |
| **Acknowledgement** | 415f | Acknowledgment Number. |
| **Field** | **Value** | **Description** |
| **Number** | eb59 |  |
| **Data Offset + Flags** | b0 | ACK flag and Data Offset = 11 (44 bytes) |
| **Windows Size** | 01a2 | Windows size |
| **Checksum** | 82ff | TCP checksum |
| **Urgent Pointer** | 0000 | Urgent pointer is not being used |

DO NOTE, in the Data Offset(b0), indicates that the TCP header is 11 words or bytes, which is more than the standard 20 bytes. This shows that the TCP header has 24 bytes of TCP Options (44 - 20 =24 bytes).

Step 3 involves the TCP options analysis, after the first 20 bytes of the header, we have 24 bytes left that aligns with the TCP options, the next 24 bytes are;

**0x0020: 0101 080a 3a11 ae31 1d3f 844e 0101 050a**

**0x0030: 415f eb58 415f eb59**

Option 1 (0101) from 0x0020: 0101 is a No operation option, Option 2 which is the 0x0022: 080a 3a11 ae31 1d3f 844e, 080a “8” indicates this is a timestamp option, the length 0a means the option is 10 bytes long, 3a11 ae31 is the Timestamp Value which is the sender's timestamp: 0x3A11AE31, 1d3f 844e, (Timestamp Echo Reply) which is an echo timestamp: 0x1D3F844E.

**REFERENCES**

**"hping3 – Active Network Security Tool," Kali Linux, 2024. [Online]. Available:**

[**https://www.kali.org/tools/hping3/.**](https://www.kali.org/tools/hping3/) **[Accessed: Oct. 1, 2024].**

**F.3rs3h3n, "Analysis of the Basics of hping3," Medium, Aug. 31, 2020. [Online]. Available:https://medium.com/@f3rs3h3n/analysis-of-the-basics-of-hping3e0e879f37a21. [Accessed: Oct. 11 2024].**

1. **Secure Networks, "Mastering Data Exfiltration with Ptunnel: An Expert Guide," Secure Networks, Aug. 10, 2023. [Online]. Available:**

[**https://www.securenetworks.cloud/mastering-data-exfiltration-with-ptunnel-anexpert-guide/.**](https://www.securenetworks.cloud/mastering-data-exfiltration-with-ptunnel-an-expert-guide/) **[Accessed: Oct. 4, 2024].**

1. **"Netcat Listener," ScienceDirect. [Online]. Available:** [**https://www.sciencedirect.com/topics/computer-science/netcat-listener.**](https://www.sciencedirect.com/topics/computer-science/netcat-listener) **[Accessed:**

**Oct. 12, 2024].**