# Analyzing DEFCON CTF Traffic

Final Report


**Group Members:**

RAFAEL ANGELO CHRISTIANTO - 2702342773

**KALPIN ERLANGGA S, S.Si., M.Kom**

Network Forensics - COMP6348001

# TABLE OF CONTENTS

# 1. Introduction

Capture the Flag competitions are a big part of the cyber security industry, they can involve reverse engineering, scanning, probing, and exploiting a machine. DEFCON is one of the biggest CTF competitions which are held annually at Las Vegas, Nevada, it is known for involving real attack scenarios and difficult challenges.

For this final project, I want to perform the packet capture network analysis to be able to uncover and understand what is happening in a network. Since this is a CTF packet capture file, we can expect a lot of scannings, malware, real attacks, C2 traffic, data exfiltration, and random suspicious ports usage. From just the packet capture file, we can understand everything that is happening.

This packet capture file I obtained from Netresec website, where for the specific file that I am analyzing here, I picked the one from year 2021 which is the DEFCON 19 (19th DEFCON), will be found in this link given: https://media.defcon.org/DEF%20CON%2019/DEF%20CON%2019%20ctf/. The link will contain real hacker behaviours under a competition's network. From this, I can understand how real attacks look from the network traffic point of view.

From this packet capture file, I aim to obtain important and crucial data that would be needed to be obtained as a forensic investigator and to build the chain of custody. But, since this is also a CTF, I want to know what the attackers in the competition aim to do, what are the protocol hierarchy, reconstruct the timeline, and other interesting insights I can get.

From this project, I aim so that I can improve my network forensics investigation skills, get hands on experience using real network forensics tools such as Wireshark, Zeek, and others that I will use on a large packet capture file, and I will also be able to see how attackers perform their actions in a network.

## 2. Methodology

I have obtained the PCAP file from Netresec.com and I will be analyzing "defcon_19-ctf-22.pcap". I used basic metadata tools to understand the general overview of the packet capture file.

First, I used Wireshark for the general overview and the active hosts within the packet capture file. From Wireshark, we can get a higher level understanding of the whole packet capture file. The next tools that I will be using will discuss the deeper packet capture information.

Zeek was then used to categorize the raw traffic to be structured protocol logs, which are separate for http logs, files logs, DNS activities and others. From this, we can try to find the flag inside of the packet capture file, especially when hidden inside of a text file.

NetworkMiner is used to extract all information necessary from the packet capture file. For this packet capture file, I will be using NetworkMiner to extract the files within the packet capture useful for trying to get the flag.

And lastly, other than Wireshark, I used tshark to get additional information about the flag. I used tshark instead of Wireshark for this since it is faster considering the amount of packets and size of this packet capture file. This will be used by utilizing the keyword search filter among the packets, specifically the raw payload bytes.

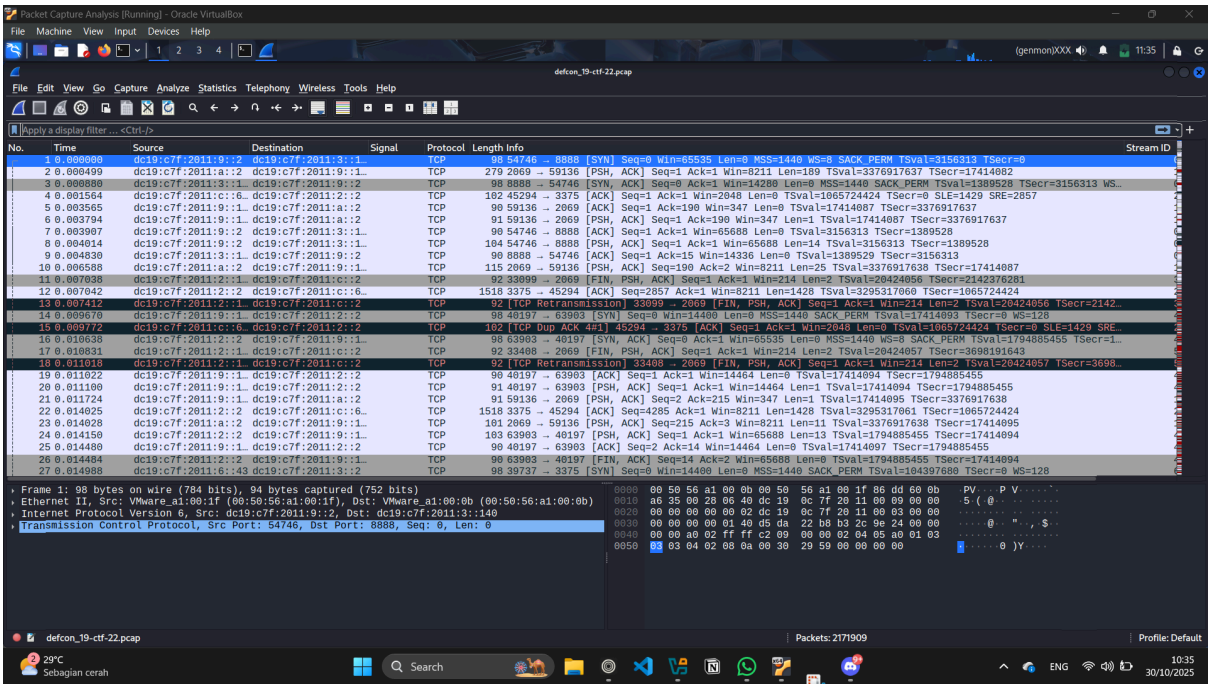
**Executive Summary (Methodology)**

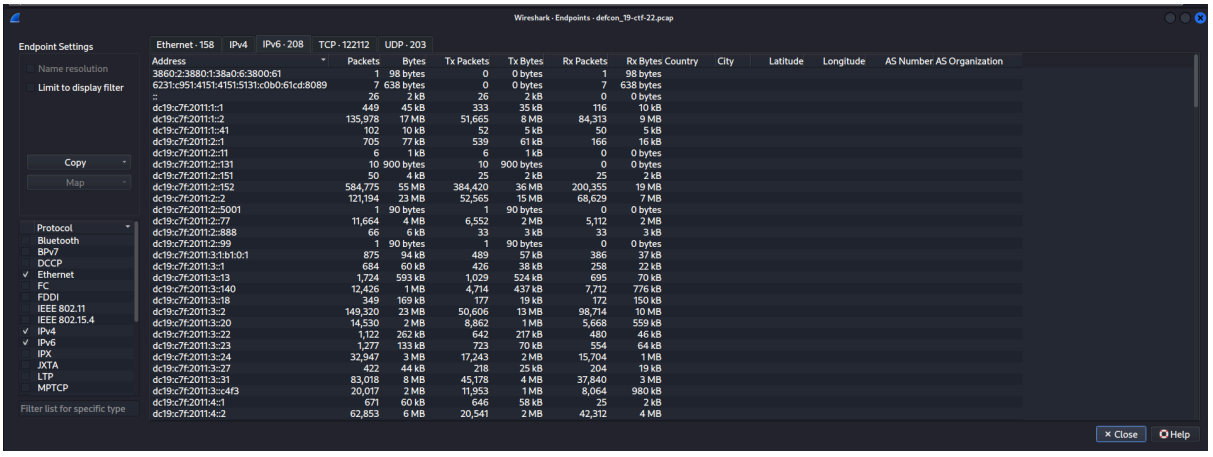
I firstly used the Wireshark application to analyze the overview of the traffic, to see who were involved, where they were talking (the ports). But then, I discovered using Wireshark will only be slow since the total file

is big, and that is why I moved to a tool called Zeek. Zeek, in short, categorizes the big traffic into smaller pieces categorized by from which device or system it is coming from, and how it behaves. From this, we can tell much faster who was involved, when they were doing it, what they were doing, what files are there, and other interesting information.

Then, other tools include NetworkMiner and tshark. NetworkMiner is a tool from Netresec, the website where we got our packet capture file. Network Miner will be used to extract all the files that are found in the packet capture to attempt to find the flag of the competition. Then tshark, will be used to help search for the flag by using keyword searches.

# 3. Findings





The file zipped archive consists of 23 packet capture files, but I picked the file named “defcon_19-ctf-22.pcap”. This consists of 2,171,909 packets, and even filtering using Wireshark can take a long time. I also inserted the Endpoints using Wireshark’s statistics, we can see that all the active hosts are IPv6, since this is all happening in a CTF, they may have their own private network (dc19:c7f:2011:*) and the ending (::1, ::3) are the host identifiers. So first, we can get the general information about the packet capture by using capinfos command.

```
┌──(rafael㉿raf)-[~]
└─$ capinfos defcon_19-ctf-22.pcap
File name:           defcon_19-ctf-22.pcap
File type:           Wireshark/tcpdump/... - pcap
File encapsulation:  Ethernet
File timestamp precision:  microseconds (6)
Packet size limit:   file hdr: 65535 bytes
Packet size limit:   inferred: 70 bytes - 1514 bytes (range)
Number of packets:   2,171 k
File size:           306 MB
Data size:           280 MB
Capture duration:    1121.656715 seconds
Earliest packet time: 2011-08-07 19:33:32.558729
Latest packet time:   2011-08-07 19:52:14.215444
Data byte rate:      250 kBps
Data bit rate:       2,000 kbps
Average packet size: 129.11 bytes
Average packet rate: 1,936 packets/s
SHA256:              2c8169c467819ce6d8bffa4b37756bbedddd5a4767a2af1adebc0a435cf55a45
SHA1:                c7c40b45b7292275b2d9b1e63cebf9b062f2ef55
Strict time order:   True
Number of interfaces in file: 1
Interface #0 info:
                     Encapsulation = Ethernet (1 - ether)
                     Capture length = 65535
                     Time precision = microseconds (6)
                     Time ticks per second = 1000000
                     Number of stat entries = 0
                     Number of packets = 2171909
```

Then, I used an open source tool called Zeek, where they transform big

```
┌──(rafael㉿raf)-[~/DEFCON]
└─$ zeek -r defcon_19-ctf-22.pcap

┌──(rafael㉿raf)-[~/DEFCON]
└─$ ls
analyzer.log  conn.log  defcon_19-ctf-22.pcap  dns.log  files.log  http.log  packet_filter.log  weird.log
```

raw data into multiple smaller sized logs, where they filter it by protocols and behavior.

Here are the breakdown of every logs that are generated:

| File | Function |
|---|---|
| analyzer.log | We can ignore this one since this is just internal protocol parsing, made by Zeek. |
| conn.log | Summary of all network connections, including IPs, ports, bytes, duration, protocol. Finding long sessions and one host talking to many other hosts can be useful. |
| dns.log | All DNS queries and responses, which can be |

| | |
|---|---|
| | useful to find any visited suspicious subdomains |
| files.log | Consists of file transferred through protocols like HTTP/FTP, and their hashing type, where usually the flags can be found here. |
| http.log | All HTTP requests and responses including the URL, user agents, and others. This is used to look for odd user agents or any suspicious file downloads. |
| packet_filter.log | Usually not very crucial in CTFs, but contains filters in packet captures. |
| weird.log | Protocol behaviours that are considered as abnormal. |

**conn.log**

So, let us start first with the conn.log file, where it can tell us the most active host. From this file, we can draw conclusions like:

- Very long conversation durations
- Very big data transfers
- Uncommon ports

One host IP talking to a lot of other hosts

```
┌──(rafael㉿raf)-[~/DEFCON]
└─$ zeek-cut id.orig_h id.resp_h service duration orig_bytes resp_bytes < conn.log | sort | uniq -c | sort -nr | head
   7713 dc19:c7f:2011:b::30:1   dc19:c7f:2011:3::2      -       -       -       -
   1233 dc19:c7f:2011:b::30:1   dc19:c7f:2011:c::2      -       -       -       -
   1139 dc19:c7f:2011:b::30:1   dc19:c7f:2011:2::2      -       -       -       -
    801 dc19:c7f:2011:b::30:1   dc19:c7f:2011:4::2      -       -       -       -
    642 dc19:c7f:2011:b::30:1   dc19:c7f:2011:7::2      -       -       -       -
    364 dc19:c7f:2011:3::c4f3   dc19:c7f:2011:4::2      -       -       -       -
    354 dc19:c7f:2011:b::30:1   dc19:c7f:2011:8::2      -       -       -       -
    296 dc19:c7f:2011:6::43     dc19:c7f:2011:4::2      -       -       -       -
    280 dc19:c7f:2011:2::2      dc19:c7f:2011:7::137    -       -       -       -
    242 dc19:c7f:2011:c::6969   dc19:c7f:2011:4::2      -       -       -       -
```

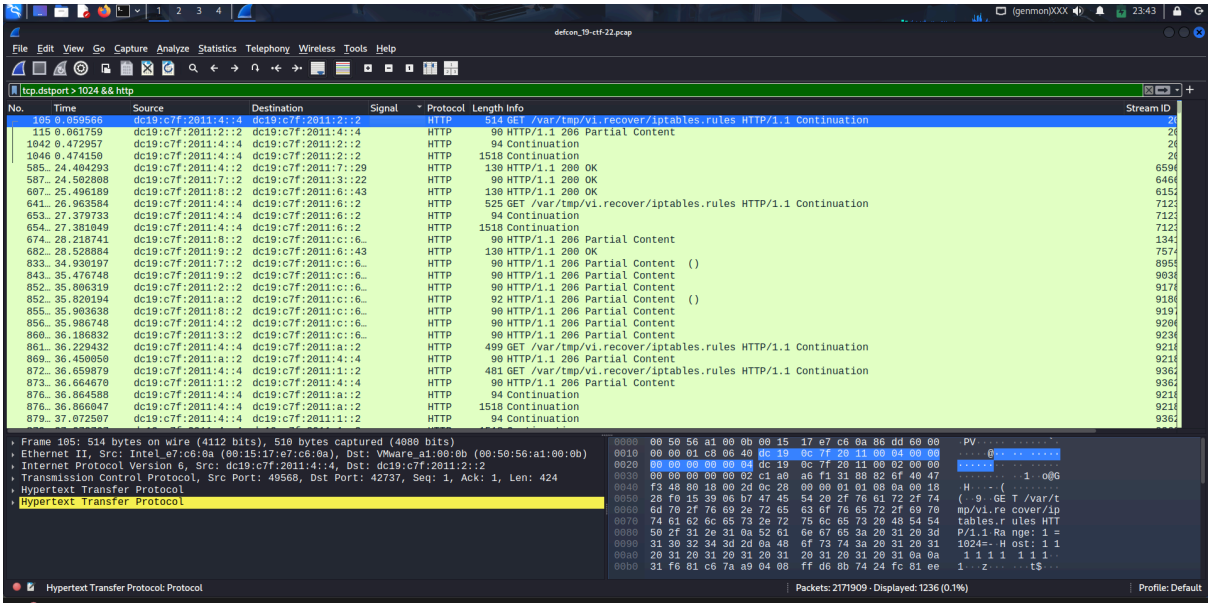
So, from the picture above, we can see that the most active IP is dc19:c7f:2011:b::30:1 connecting to dc19:c7f:2011:3::2, a total of 7713 times, so from that we can predict that this is an action done for

- Network scanning

- Brute forcing
- Data exfiltration

Then, why we cannot see the service or protocol being used is because, since this is a CTF file, a lot of actions are scanning, and they may use random or high ports (like shown in image below), or sometimes even they could have their own specific protocols. And even validated by the supporting Wireshark screenshot given below a lot of connections also used high number ports, which is expected in a CTF environment.

```
┌──(rafael㉿raf)-[~/DEFCON]
└─$ zeek-cut id.orig_h id.resp_p < conn.log | grep "dc19:c7f:2011:b::30:1" | sort | uniq -c | sort -nr | head
  24038 dc19:c7f:2011:b::30:1   44366
     12 dc19:c7f:2011:b::30:1   6391
      9 dc19:c7f:2011:b::30:1   2222
```




Here is the table of the original host sending packets to a respondent host who is the ones receiving the packets, including with the original bytes and response bytes.

```
┌──(rafael㉿raf)-[~/DEFCON]
└─$ cat conn.log | zeek-cut id.orig_h id.resp_h orig_bytes resp_bytes | awk '{print $1, $2, $3, $4}'  | sort -k3 -nr | head
dc19:c7f:2011:9::2 dc19:c7f:2011:c::6969 368964 333
dc19:c7f:2011:a::2 dc19:c7f:2011:c::6969 139221 759
dc19:c7f:2011:6::2 dc19:c7f:2011:c::6969 133430 718
dc19:c7f:2011:9::2 dc19:c7f:2011:c::6969 132329 354
dc19:c7f:2011:2::2 dc19:c7f:2011:c::6969 115594 799
dc19:c7f:2011:2::2 dc19:c7f:2011:c::6969 113989 761
dc19:c7f:2011:2::2 dc19:c7f:2011:c::6969 111151 333
dc19:c7f:2011:6::2 dc19:c7f:2011:c::6969 105189 263
dc19:c7f:2011:6::2 dc19:c7f:2011:c::6969 102246 611
dc19:c7f:2011:2::152 dc19:c7f:2011:3::2 99680 89320
```

So, here are the connections who have the biggest data transfer, which we can see from the screenshot below, which happened between dc19:c7f:2011:8::99 and dc19:c7f:2011:3::2, which from total data transfer, we can get 4,294,967,307, this may mean 2 things, which are data exfiltration and file transfers.

```
┌──(rafael㉿raf)-[~/DEFCON]
└─$ cat conn.log | zeek-cut id.orig_h id.resp_h orig_bytes resp_bytes | awk '{print $1, $2, $3+$4}'  | sort -k3 -nr | head
dc19:c7f:2011:8::99 dc19:c7f:2011:3::2 4294967307
dc19:c7f:2011:c::6969 dc19:c7f:2011:9::2 370828
dc19:c7f:2011:c::6969 dc19:c7f:2011:7::2 370828
dc19:c7f:2011:c::6969 dc19:c7f:2011:9::2 370825
dc19:c7f:2011:c::6969 dc19:c7f:2011:9::2 370824
dc19:c7f:2011:c::6969 dc19:c7f:2011:9::2 370824
dc19:c7f:2011:c::6969 dc19:c7f:2011:9::2 370823
dc19:c7f:2011:c::6969 dc19:c7f:2011:9::2 370822
dc19:c7f:2011:c::6969 dc19:c7f:2011:9::2 370821
dc19:c7f:2011:c::6969 dc19:c7f:2011:3::2 370821
```

So, why there is 2 sorts in the command is because “uniq -c” can only count duplicates if they are consecutive or next to each other in the row, so first we have them sort the host connections who appears scattered, and then we sort them again in a reversed sorted order, and show only the top 10 with “head”.

**dns.log**

Next, we can move on to dns.log, where we can see queries and subdomains visited, and importantly if there are suspicious subdomains visited

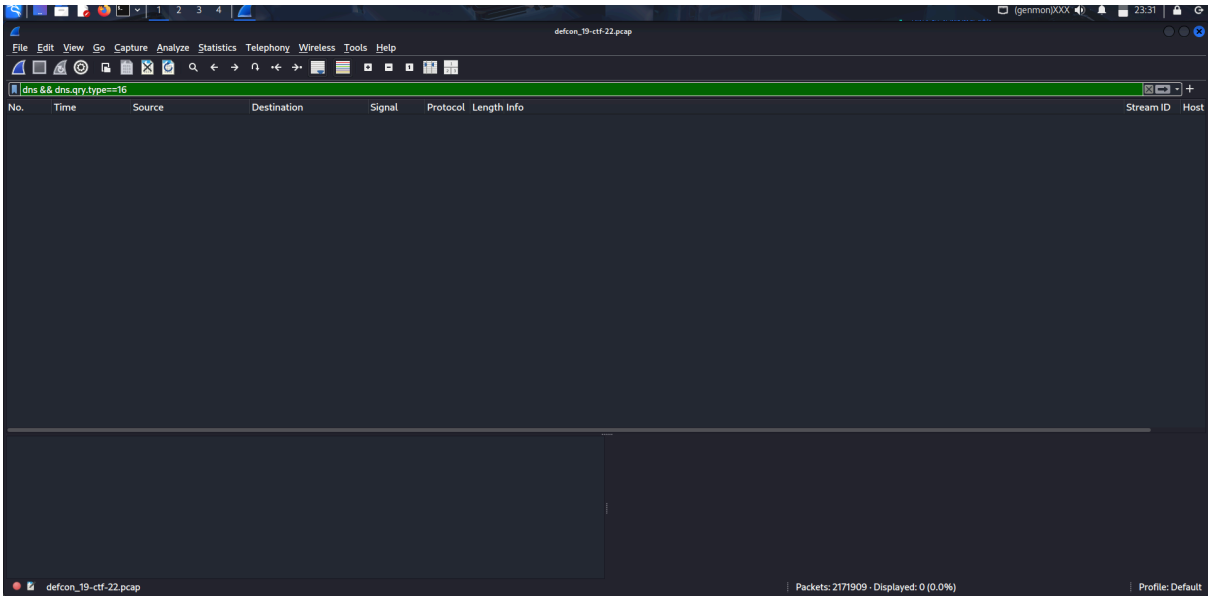
```
┌──(rafael㉿raf)-[~/DEFCON]
└─$ zeek-cut query resp_name < dns.log | sort | uniq -c | sort -nr | head
     14 wpad
     11 hackbook.local
      8 kevin-gibbss-macbook-pro.local
      7 _airport._tcp.local
      6 1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.e.f.ip6.arpa
      5 _sleep-proxy._udp.local
      5 home extreme._afpovertcp._tcp.local
      4 _services._dns-sd._udp.local
      4 oz-laptop
      4 isatap
```

| Subdomains | Meaning |
|---|---|
| wpad | It is abbreviation for Web Proxy Auto Discovery, which is when a machine is trying to automatically detect whether there is a proxy |
| *.local<br>*._tcp.local<br>*._tcp.local | These are related to the OS of MAC for broadcasting and local network discovery. |
| 1.0.*.0.8.e.f.ip6.arpa | It is an IPv6 reverse DNS lookup |


From that, we can conclude that the network consists of a lot of local network discovery such as mDNS, and WPAD queries. There were MAC OS devices talking to each other using mDNS. However, there are no signs of suspicious DNS tunneling, C2, and domain-based attacks. Despite that, WPAD poisoning can be used to capture traffic, but no signs of that are shown here.

Since DNS tunneling usually hides texts in TXT files to not make it suspicious, so that we try to find whether there are DNS TXT query requests and there are none, so concluding that there is most likely no DNS tunneling.

But, after checking again with Wireshark, with the DNS query type 16 filter, which means it is  TXT record, which could blend in with normal traffic but carry a lot of data that would be already DNS tunneling or C2 servers

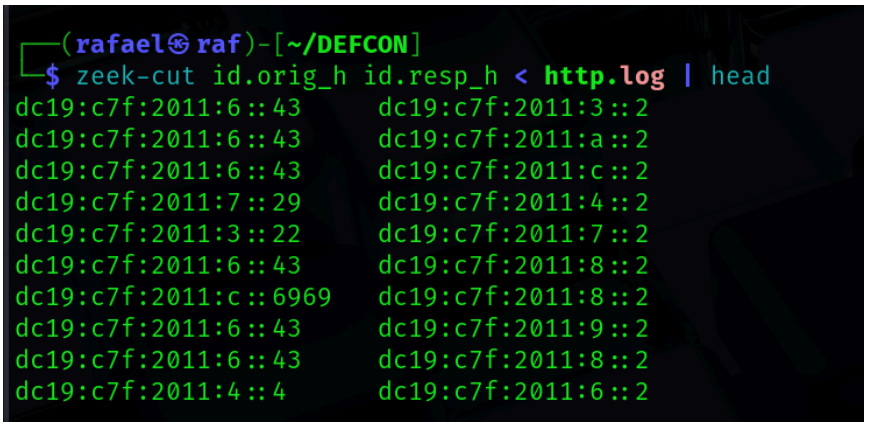



**http.log**

Next, we can move on to the http.log, containing the HTTP requests and responses made, and important in a CTF, where the flags would be hidden. Below is the key insights that might be useful for the HTTP logs.

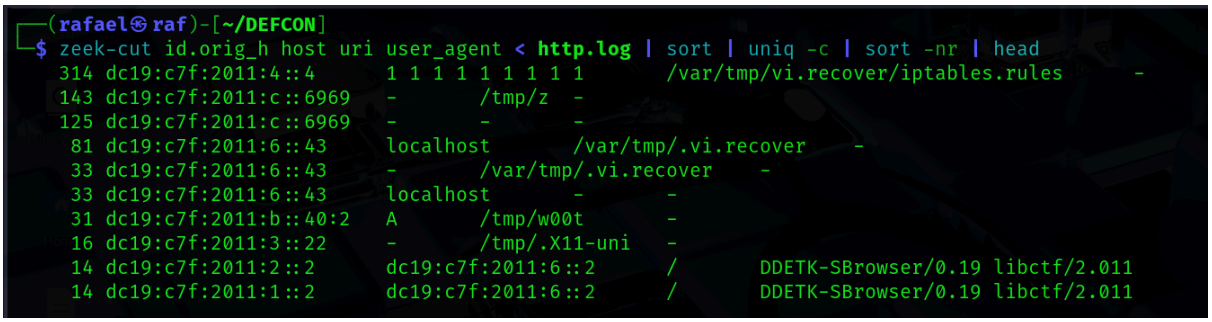
- Remote code execution if the ending is like “?cmd=...”
- File extensions, indicating data exfiltration
- Suspicious user agents

From the screenshot given below, every other host can be seen talking with dc19:c7f:2011:3::2. This shows that dc19:c7f:2011:3::2 may be the machine that is trying to be hacked the most during the CTF.



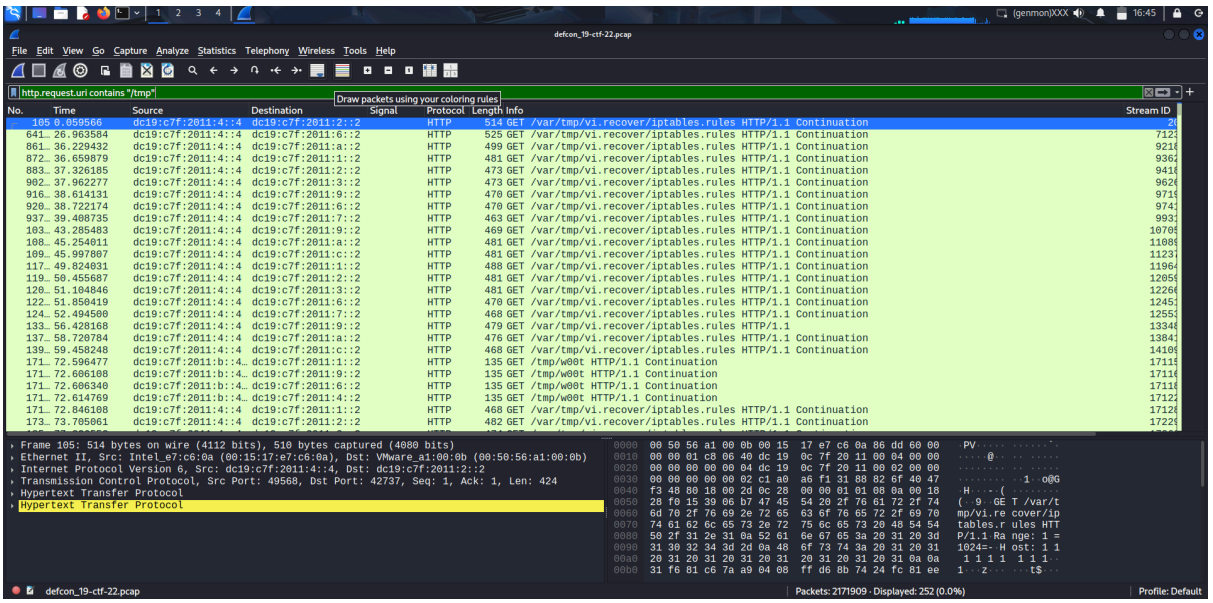

For the screenshots below, we can conclude that the paths such as

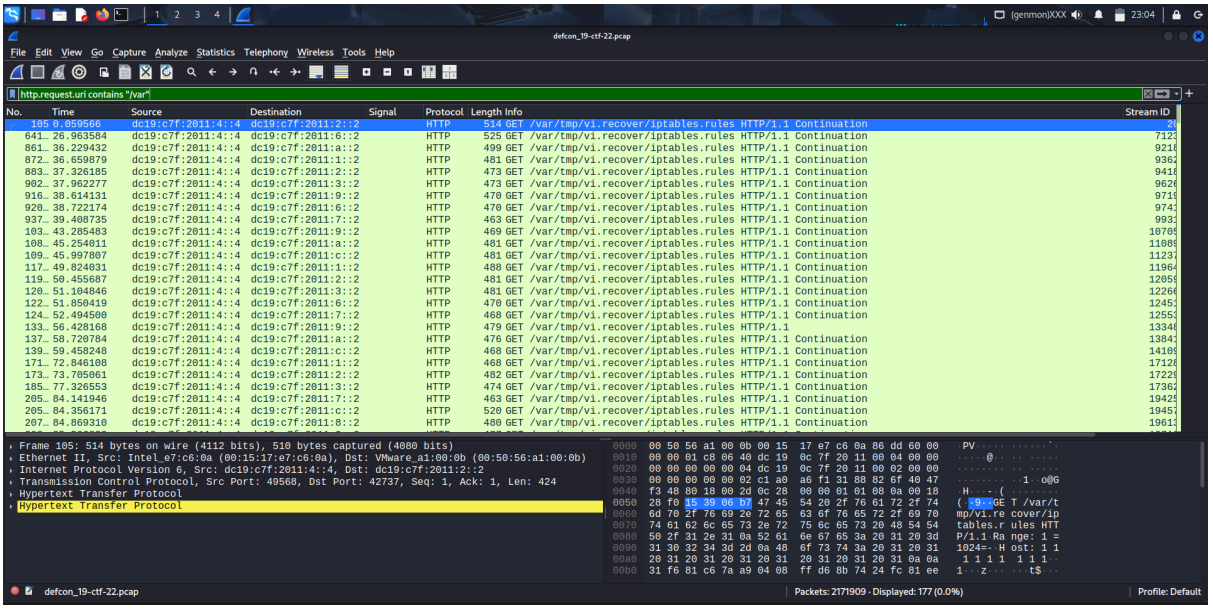
- /var/tmp/vi.recover/iptables.rules
- /tmp/z
- /tmp/w00t



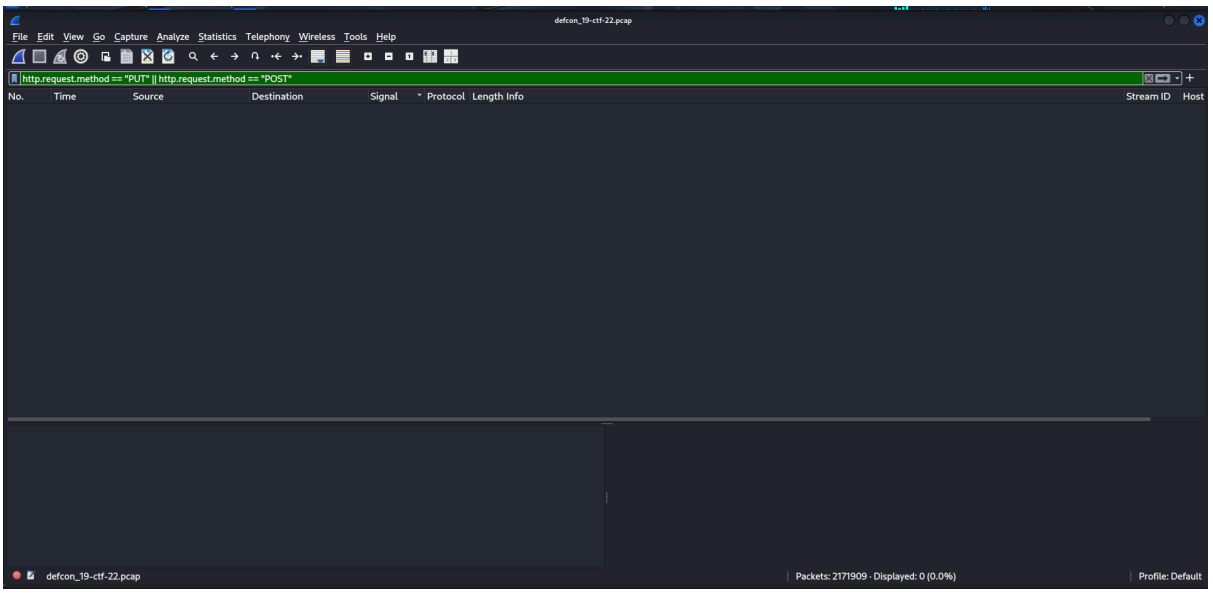

Those paths shown above such as the /tmp/ or /var/tmp/ folders are system-like paths. And plus, the addition that they are not using any user agent indicates that they may not be using any browsers, instead they use command utilities like curl, wget, and other similar utilities.

And from the image below, as confirmation, I filter the HTTP requests that contain /tmp or /var paths, it is not a PUT request to put malicious payload into the system-like paths, but instead it is a GET request to the host machine's system-like paths. This could also mean data exfiltration.




But then, I checked whether the HTTP methods are also included with PUT and POST. That would mean it has malware payload sent over HTTP, but it does not return any results, which tells us it does not have malware sent over the HTTP. The screenshot of Wireshark is given below




Next, for the next screenshot we find below, we can see that a lot of the requests are GET requests with a status code of 200 which shows that it is successful and 206 which means it is only partially successful. So, this tells us that there is actually file exfiltration as well and most likely it was being used.

```
┌──(rafael㉿raf)-[~/DEFCON]
└─$ zeek-cut id.orig_h method uri status_code < http.log | sort | uniq -c | sort -nr | head
    157 dc19:c7f:2011:4::4      GET     /var/tmp/vi.recover/iptables.rules      206
    157 dc19:c7f:2011:4::4      GET     /var/tmp/vi.recover/iptables.rules      -
    125 dc19:c7f:2011:c::6969   GET     /tmp/z  0
    125 dc19:c7f:2011:c::6969   -       -       -
     81 dc19:c7f:2011:6::43     GET     /var/tmp/.vi.recover    -
     44 dc19:c7f:2011:b::40:2   GET     /tmp/foo        206
     33 dc19:c7f:2011:6::43     GET     /var/tmp/.vi.recover    200
     33 dc19:c7f:2011:6::43     -       -       -
     28 dc19:c7f:2011:5::2      GET     /       -
     27 dc19:c7f:2011:2::2      GET     /       -
```

So from the last screenshot for http.log, we can see the top visited path is /var/tmp/v1/recover/iptables.rules then followed by /tmp/z, this tells us that a lot of host downloaded the same file, like showing worm behaviour, infecting a lot of users in the network. But since this is a CTF, I do not think it is a real worm and is just intentionally generated.

```
┌──(rafael㉿raf)-[~/DEFCON]
└─$ zeek-cut uri < http.log | sort | uniq -c | sort -nr | head
    314 /var/tmp/vi.recover/iptables.rules
    225 /
    174 -
    149 /tmp/z
    114 /var/tmp/.vi.recover
     44 /tmp/foo
     31 /tmp/w00t
     22 /tmp/.X11-uni
      7 /tmp/.test
      5 /current.jpg
```

**files.log**

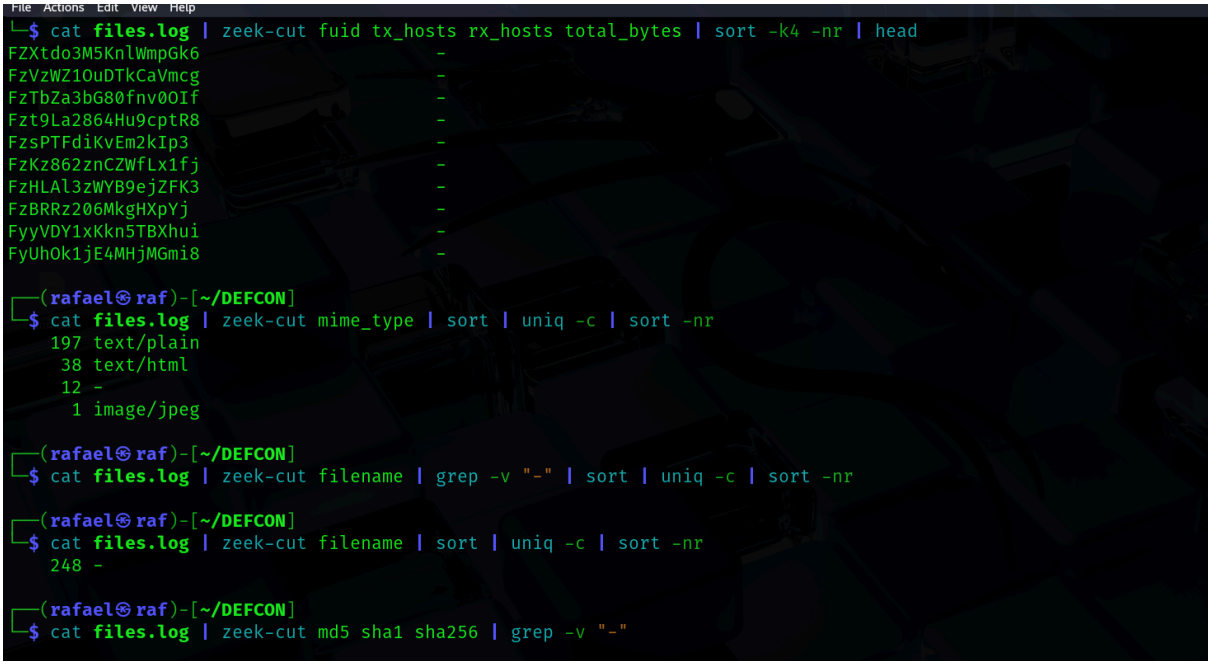
For this log, we can attempt to find the flag that is hidden for this competition's CTF. As we can see from the screenshot below, we can see that all the files are just plain text and plain HTML. So from this, we can tell that there are downloaded files like said from the HTTP logs GET

```
┌──(rafael㉿raf)-[~/DEFCON]
└─$ zeek-cut fuid mime_type < files.log | sort | uniq -c | head
      1 F0Dm5L2vmeai9kYKl5      text/plain
      1 F0EKWiEcMysEzma67       text/plain
      1 F0RSAr2f0QzwDvb09i      text/plain
      1 F184gP2zUwlyai2Pui      -
      1 F18TOi2743wil3KVR5      text/plain
      1 F1DLum3vEvS5ln5STd      text/plain
      1 F1DP1EthXLvplj9Lc       text/plain
      1 F1nqSj4QA8MiuHvsCl      text/plain
      1 F1wmKS3hnSfXkaLoTk      text/plain
      1 F23K8C34A4ZAjCbdv4      text/html
```

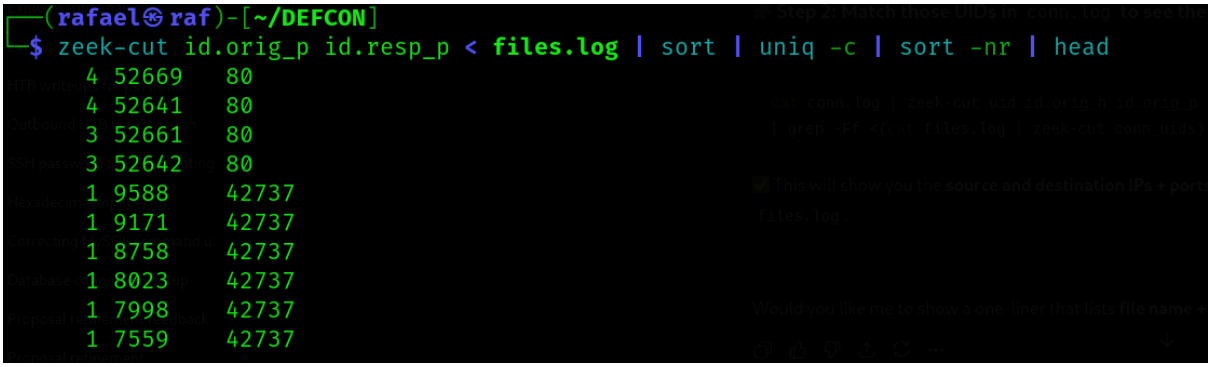
requests, but here we find out that those are just plain texts or plain HTML, instead of executables.

The screenshot below tells us that the types of files are not anything suspicious since there are a total of 248 file transfers happening but there was nothing malicious, as there were only plain text, html and jpeg files.

Also, from the screenshot below, we cannot find any filenames, hashings or bytes of data, which could mean that file transfers were not complete.



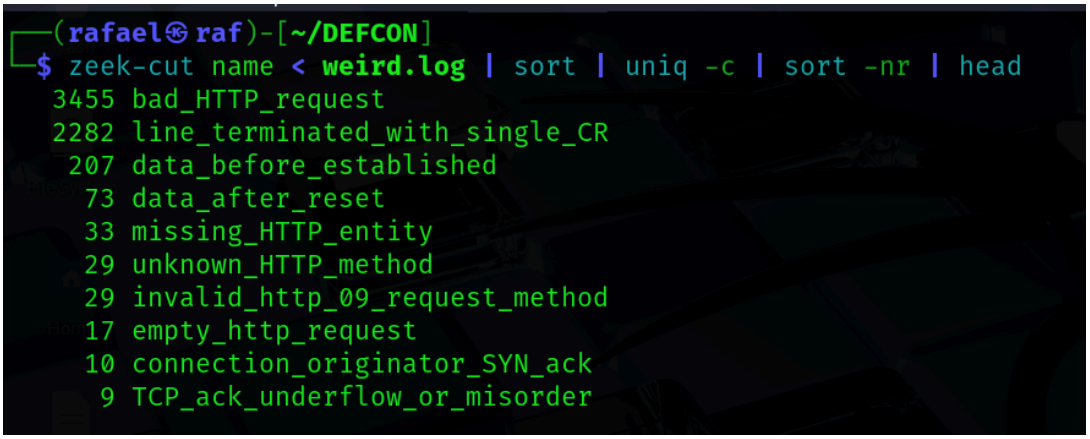

Here in the screenshot below, we can observe the different destination ports that were used, there were port 80, most likely just used for normal browsing on HTTP. But then, there is a high port number usage which is




42737, which in a CTF scenario it can mean, scanning or data exfiltration.

**weird.log**

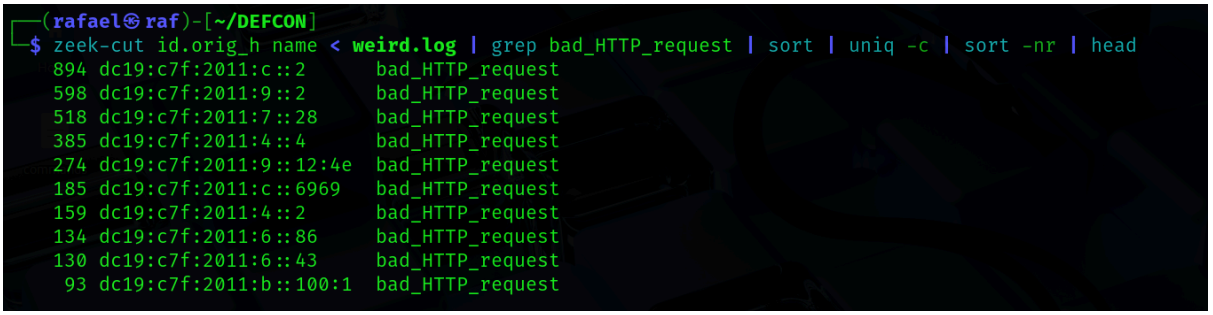
This log may be very important as it can tell us interesting information, since it detects unusual behaviours in the network. Here we can maybe get insights about possible scanning, tunneling, exploit payloads, and other attacker techniques. The screenshot below tells us the weird events' names found within the logs.

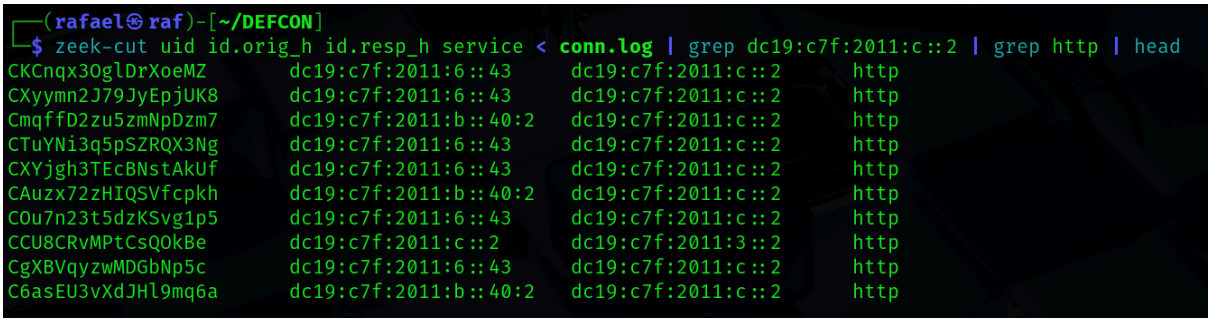



| Count | Weird Event | Meaning | CTF Scenario |
|---|---|---|---|
| 3455 | bad_HTTP_request | HTTP request not good | Someone may be sending custom HTTP payloads |
| 2282 | line_terminated_with_single _CR | HTTP requests ended with \r without \n | Custom tools like scanning or probing |
| 207 | data_before_established | Data sent before the TCP handshake starts | Used in recon bots |
| 73 | data_after_reset | Data still sent after RST | Someone trying to resend exploit |
| 33 | missing_HTTP_entity | HTTP request is sent without a body | Broken HTTP requests |
| 29 | unkown_HTTP_method | HTTP verbs are invalid | Usually a trick in CTFs |

| | | | |
|---|---|---|---|
| 29 | invalid_http_09_request_method | HTTP 0.9 request | Usually used for tunneling or stealthy file exfiltration |
| 17 | empty_http_request | Empty HTTP requests | |
| 10 | connection_originator_SYN_ack | A user sends SYN and ACK unexpectedly | Usually TCP spoofing and scanning |
| 9 | TCP_ack_underflow_or_misorder | Bad sequence numbers | Possibly involving packet crafting. |

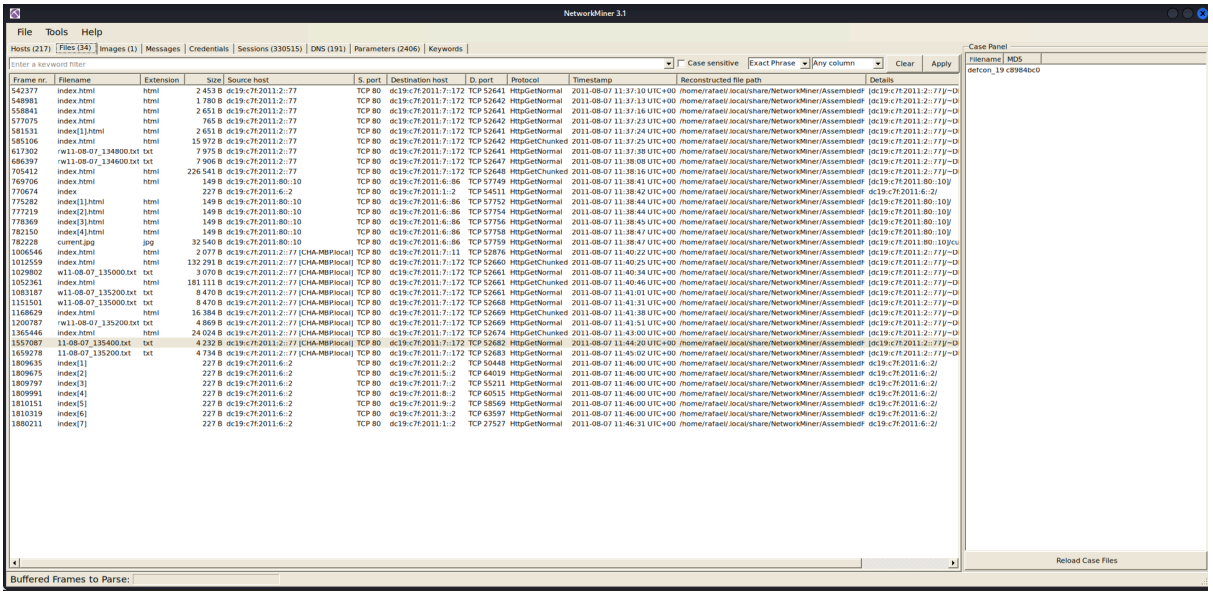
Next, since the highest count is the bad HTTP requests, we need to find who is the attacker who is doing that. From the screenshot below, we can know that the user who is involving the most bad HTTP requests is dc19:c7f:2011:c::2.



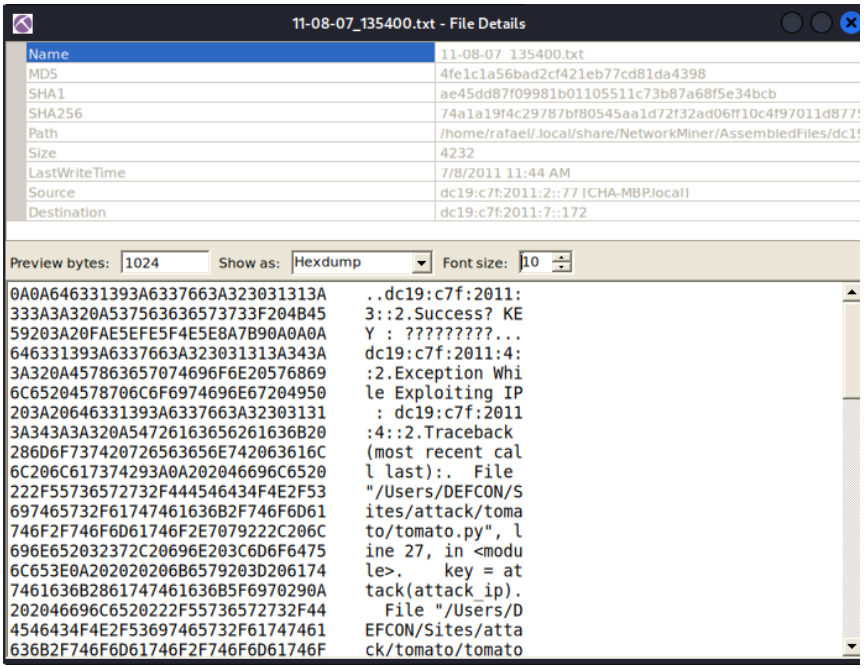
But then in the conn log, when we want to find who is connecting to who, with dc19:c7f:2011:c::2, and the HTTP requests. But then, we soon find out that dc19:c7f:2011:c::2 is actually a responder like what we found before

## NetworkMiner Analysis




NetworkMiner was used to reconstruct files and extract the important contents from the big packet capture file. NetworkMiner identified several plain text files, HTML, and image files transferred over HTTP. However, none of the extracted text files contained recognizable CTF flag formats or other strings that would indicate hidden challenge flags. Many of the file transfers were incomplete or partially delivered, this indicated automated downloads than just data exfiltration

The text files being displayed in NetworkMiner, the text files did not represent any DNS TXT record analysis or DNS tunneling. This is just a normal text files from a CTF for example one of the text files includes an accidental error output from an exploit attempt. The screenshot is given below




And if it contains the real flag, we should correlate it to the usage of Zeek. If it is inside of the files.log, http.log, or conn.log, it will most likely be the file containing the flag. There is kind of a big difference between Zeek and NetworkMiner

- Zeek will remove unnecessary unimportant network traffic records and start grouping them, this will be useful since DEFCON CTF's traffic may be noisy and aggressive
- NetworkMiner will extract all the files, credentials, parameters, and all the other information that can be obtained from the packet capture, useful for deeper inspection.

**tshark**

Below are the outputs for the tshark commands, return all empty outputs which confirms again the flag will not be availably found from the network traffic

The traffic below searches the raw TCP payload bytes, for the packets containing the text "flag". So no clear text of the flag was sent.

```
┌──(rafael㉿raf)-[~/DEFCON]
└─$ tshark -r defcon_19-ctf-22.pcap -Y 'tcp' -T fields -e data | grep -i flag
```

Here I am doing a deeper inspection into the packet capture files of the strings like ctf{}, flag{}, defcon{}. But, nothing seems to be outputed as well.

```
┌──(rafael㉿raf)-[~/DEFCON]
└─$ tshark -r defcon_19-ctf-22.pcap -Y 'tcp' -V | grep -Ei 'flag\{|ctf\{|defcon\{'
```

Here I am checking HTTP response bodies containing the word flag. But nothing is returned meaning it cannot be found on HTTP response or files.

```
┌──(rafael㉿raf)-[~/DEFCON]
└─$ tshark -r defcon_19-ctf-22.pcap -Y http.response -T fields -e ip.src -e ip.dst -e http.response.code -e http.file_data | grep -i flag
```

Here I am checking whether the flag is hidden in the DNS text records or DNS tunneling, but there seems to be no output so it is not.

```
┌──(rafael㉿raf)-[~/DEFCON]
└─$ tshark -r defcon_19-ctf-22.pcap -Y 'dns && dns.txt' -T fields -e dns.txt | grep -i flag
```

**flag**

For the flag I just filtered using grep from all of the log files, the word “flag”. But I only found it 2, both found at conn.log. CfLagl4aItSMHSZ6Ij and CfLagY6hTDCBynBHi. But, these are just Zeek-generated connection UIDs.

```
┌──(rafael㉿raf)-[~/DEFCON]
└─$ grep -i "flag" *.log

conn.log:1312717127.786311      C9k0jWkFLaGKLG6M1       dc19:c7f:2011:2::152    56204   dc19:c7f:2011:b::2      2069    tcp     -       0.0000350
0       REJ     F       F       0       Sr      2       160     2       120     -       6
conn.log:1312717222.655144      CfLAGl4aItSMHSZ6Ij      dc19:c7f:2011:2::152    42514   dc19:c7f:2011:c::2      2069    tcp     -       1.0079882
217     SF      F       F       0       ShAdDFTfR       26      1868    5       585     -       6
conn.log:1312717281.319818      CD5BvB4hCuHN1hfLAg      dc19:c7f:2011:2::152    34631   dc19:c7f:2011:5::2      44366   tcp     -       0.0003450
0       REJ     F       F       0       Sr      1       80      1       60      -       6
conn.log:1312717705.962530      CfLagY6hTDCBynBHi       dc19:c7f:2011:7::137    48551   dc19:c7f:2011:8::2      44366   tcp     -       0.1056321
4       2       SF      F       F       0       ShADadfF        6       386     5       314     -       6

┌──(rafael㉿raf)-[~/DEFCON]
└─$ grep -R "CfLAGl4aItSMHSZ6Ij" *.log
grep -R "CfLagY6hTDCBynBHi" *.log

conn.log:1312717222.655144      CfLAGl4aItSMHSZ6Ij      dc19:c7f:2011:2::152    42514   dc19:c7f:2011:c::2      2069    tcp     -       1.0079882
217     SF      F       F       0       ShAdDFTfR       26      1868    5       585     -       6
conn.log:1312717705.962530      CfLagY6hTDCBynBHi       dc19:c7f:2011:7::137    48551   dc19:c7f:2011:8::2      44366   tcp     -       0.1056321
4       2       SF      F       F       0       ShADadfF        6       386     5       314     -       6
```

Now, we will have to decide which one is the right string, I will organize all the results into a table for better preview:

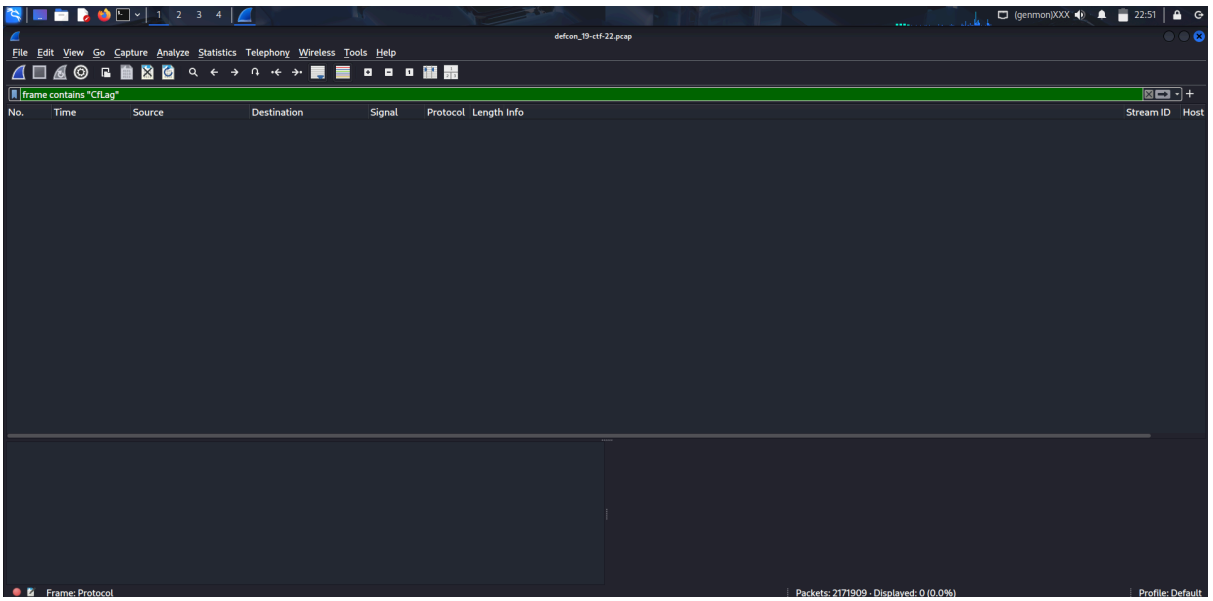
| | Flag 1 | Flag 2 |
|---|---|---|
| UID | CfLAGl4aItSMHSZ6Ij | CfLagY6hTDCBynBHi |
| orig_h | dc19:c7f:2011:2::152 | dc19:c7f:2011:7::137 |
| orig_p | 42514 | 8551 |
| resp_h | dc19:c7f:2011:c::2 | dc19:c7f:2011:8::2 |
| resp_p | 2069 | 44366 |
| proto | tcp | tcp |
| duration | 1.0079882 | 0.1056321 |
| orig_bytes | 1868 | 386 |
| resp_bytes | 585 | 314 |

Let us discuss per point

- The port for flag number 1 is high, but compared to flag number 2 it is severely high
- The duration for flag 1 is also 1 second long, however for flag 2 it is only 0.1 seconds long
- Bytes transferred is also different where flag 1 in total had about 2.4 KB, but then flag 2 only got 700 bytes of data

These factors may indicate that flag 1 is the actual flag, but then this may just be Zeek-generated metadata identifiers rather than attacker-controlled or application-layer data transmitted over the network. This is because they do not follow the format of a flag in a CTF which is flag{..}, which are usually found at HTTP responses, reconstructed files, DNS records, or other payload-level content

Even after checking again with the actual PCAP file using Wireshark with the initial letters of the 2 strings mentioned before, nothing comes up.




## Executive Summary (Findings)

| Role | IPv6 Address | Evidence |
|---|---|---|
| Primary target | dc19:c7f:2011:3::2 | Highest inbound connections of HTTP downloads, exfil patterns |
| Attacker | dc19:c7f:2011:b::30:1 | 7,713 connections, high ports, short sessions |
| Host being exfiltrated | dc19:c7f:2011:8::99 | ~4.29GB data transfer |

The analysis identified a highly active IPv6 network with extensive automated scanning and exploitation activity. The host dc19:c7f:2011:3::2 was the primary target, receiving a high volume of connections from multiple attacking hosts. Notable source IPs included dc19:c7f:2011:b::30:1, which generated thousands of connections consistent with scanning or brute-force behavior, and dc19:c7f:2011:8::99, which was involved in the largest observed data transfers, suggesting simulated file transfer or exfiltration activity.

HTTP traffic revealed repeated access to system-related paths such as `/tmp` and `/var/tmp`, But instead of PUT or POST requests, there were GET requests, which could potentially be data exfiltration. Numerous malformed TCP and HTTP events were detected and there was not a user agent which could use command line tools, originating from hosts such as dc19:c7f:2011:c::2, reflecting the use of custom tools and exploit frameworks. Despite aggressive activity across the network, no plaintext CTF flags or sensitive data were observed in HTTP responses, DNS traffic, or reconstructed files.

## 4. Conclusion

The packet capture file involved multiple IPv6 hosts actively performing scanning, exploitation, and post-exploitation activities against a primary target host (dc19:c7f:2011:3::2). The traffic is very high, consisting of thousands of short lived connections which could be beaconing, high numbers of ports being used, and malformed TCP and HTTP behavior. These characteristics indicate the use of automated tools and custom exploit frameworks rather than normal user-driven communication.

HTTP traffic reveals repeated access to system-like paths such as /tmp and /var/tmp, suggesting file retrieval after successful exploitation, with some transfers showing partial content responses consistent with scripted downloads. There is no evidence found about DNS tunneling or external command-and-control infrastructure, as DNS activity is limited to local discovery and reverse lookups. Overall, the packet capture resembles the network traffic recorded during the 2021 DEFCON CTF, so there will be a lot of aggressive and fast penetration testing techniques.

**Executive Summary (Conclusion)**

This network capture represents a live DEFCON Capture-The-Flag (CTF) competition environment, where multiple participants simultaneously attacked a vulnerable system within a controlled IPv6 network. The traffic shows intensive automated scanning, repeated connection attempts, and abnormal communication patterns, indicating deliberate exploitation activity rather than normal network usage.

Analysis found no evidence of external command-and-control or real-world malware spread. Instead, the behavior reflects intentional CTF challenge design, where participants attempt to retrieve hidden information through direct network interaction. The captured data confirms that the activity was confined, controlled, and focused on competitive security testing, with challenge flags embedded within network metadata rather than malicious payloads.

## 5. Timeline Reconstruction

### 1. Initial Network Discovery

At the beginning of the packet capture, multiple IPv6 hosts generated a high volume of short TCP connections across random and high port numbers. DNS activity was dominated by mDNS (*.local, *._tcp.local) and WPAD queries, indicating local service discovery rather than external communication. Numerous malformed TCP and HTTP events were recorded, suggesting automated scanning and probing behavior typical of CTF environments.

### 2. Convergence on a Primary Target

Then, traffic began to focus on a single host, dc19:c7f:2011:3::2, which received the highest number of inbound connections from multiple source hosts. Connection patterns show repeated attempts from the same hosts using different ports. This indicates the target system of the competition.

### 3. Exploitation Attempts

Next traffic shows abnormal HTTP requests directed at system-related paths such as /tmp/z, /tmp/w00t, and /var/tmp/vi.recover/iptables.rules. These requests frequently lacked valid HTTP headers and user-agent

strings and produced partial content responses. At the same time, weird.log recorded increases in events such as bad_HTTP_request, unknown_HTTP_method, and data_before_established, consistent with exploit payload delivery and custom tooling.

**4. Post-Exploitation Automation**

Multiple hosts repeatedly downloaded identical files from the target host, indicating automated post-exploitation behavior rather than manual interaction. File transfers observed in files.log were primarily plain text or HTML and were often incomplete, suggesting scripted retrieval or challenge-generated artifacts rather than traditional malware binaries.

**5. High-Volume Data Transfers**

Later in the capture, large volumes of data were exchanged between dc19:c7f:2011:8::99 and dc19:c7f:2011:3::2, totaling approximately 4.29 GB. These transfers occurred over high-numbered ports and short sessions, consistent with simulated data movement or exfiltration behavior commonly embedded in CTF challenges.

**6. Absence of Payload-Level Flags**

Throughout the capture, no CTF flags were observed within HTTP responses, reconstructed files, DNS records, or raw TCP payloads. Two flag-like strings were identified in conn.log; however, these were confirmed to be Zeek-generated connection UIDs rather than application-layer data transmitted by participants.

## 6. Chain of Custody

| Task | Date |
|---|---|
| Obtaining the PCAP file from Netresec | October 30 2025 |
| Analyzing the PCAP file | October 30 2025 - 9 January 2026 |
| Finalizing the documents | 9 January 2026 |