CS4415 – Final Project Report

Network Traffic Analyzer and Denial of Service Detector

Dr. Arash Habibi Lashkari

15-04-21

By: Amir David, Matthew Hunter, Julia Walker

Project Description

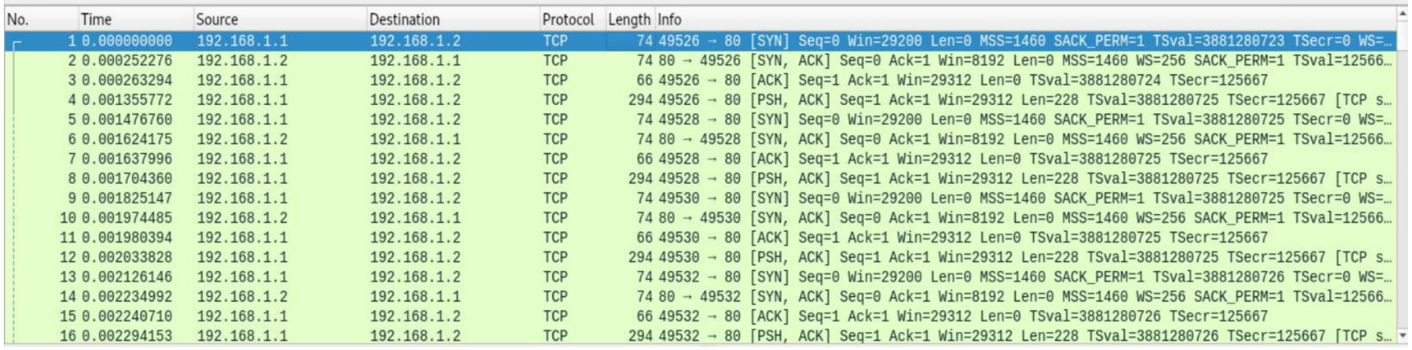
Denial-of-Service attacks attempt to disrupt the availability of a target network service by exhausting the service's resources. The attacker achieves this by creating a large amounts of network traffic directed at the target network until that network crashes. During this attack, legitimate users are either completely unable to access the network, or their user experience is severely disrupted.

Our team selected the Slowloris tool to simulate a Denial-of-Service attack. We have selected to use Slowloris as we were already familiar with it after using it in class, and it seemed to be the "strongest" or most reliable out of all the other DoS tools we have tried (such as HOIC, HULK, and Thor's Hammer). During an initial brainstorming session, our team came up with 3 characteristics for detection of the Slowloris DoS attack, namely: number of network connections opened in one-minute, large number of network requests coming from the same source IP, and the length of time that the request is kept open. Through traffic analysis of simulated attacks, and research about the operation of Slowloris we developed new features that are more specific to a Slowloris attack. The new features we selected were: 1) Detection of many active IP connections 2) Detection of a large exchange of segmented TCP packets.

Analysis

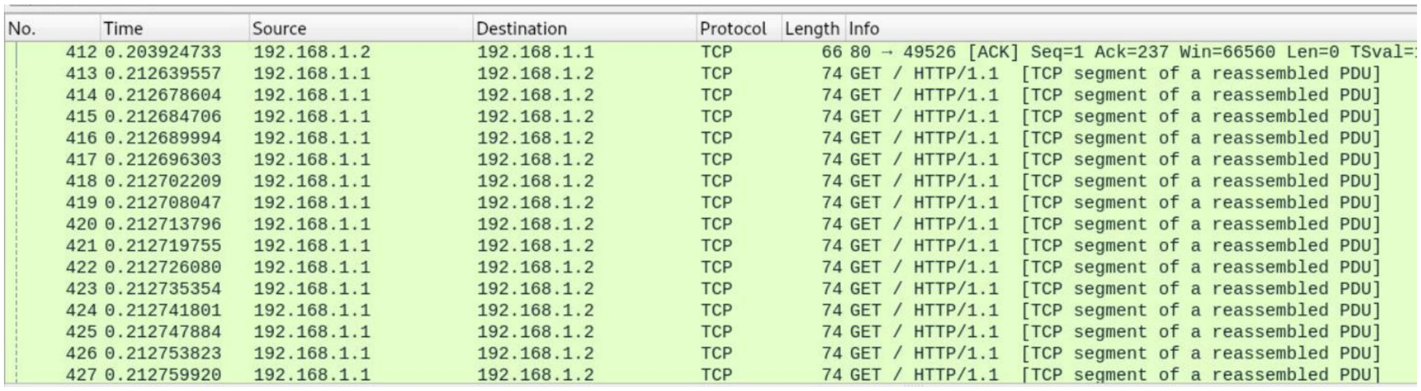
Our analysis was done through a combination of examining network traffic in Wireshark and Netstat, and by performing online research.

One of the first characteristics of Slowloris that we determined was a flood of [PSH, ACK] flags directed at the target machine. Upon online research we have found out that this is a common method used in DoS attacks, used to "take down stateful defenses "[1]. This works by pushing the packet's data directly to the receiving stack without any record of the packet.[2]



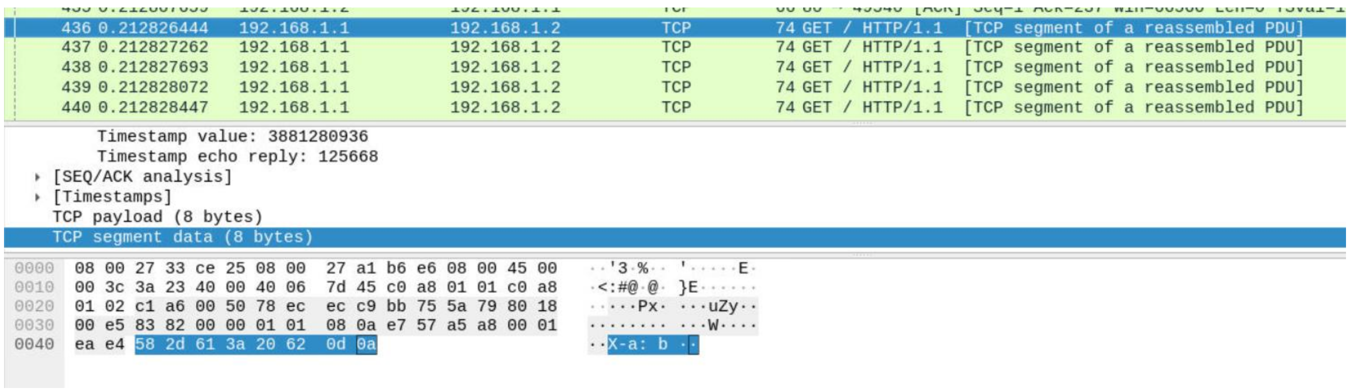
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.1.1	192.168.1.2	TCP	74	49526 → 80 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=3881280723 TSecr=0 WS=…
2	0.000252276	192.168.1.2	192.168.1.1	TCP	74	80 → 49526 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1 TSval=12566…
3	0.000263294	192.168.1.1	192.168.1.2	TCP	66	49526 → 80 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=3881280724 TSecr=125667
4	0.001355772	192.168.1.1	192.168.1.2	TCP	294	49526 → 80 [PSH, ACK] Seq=1 Ack=1 Win=29312 Len=228 TSval=3881280725 TSecr=125667 [TCP s…
5	0.001476760	192.168.1.1	192.168.1.2	TCP	74	49528 → 80 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=3881280725 TSecr=0 WS=…
6	0.001624175	192.168.1.2	192.168.1.1	TCP	74	80 → 49528 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1 TSval=12566…
7	0.001637996	192.168.1.1	192.168.1.2	TCP	66	49528 → 80 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=3881280725 TSecr=125667
8	0.001704360	192.168.1.1	192.168.1.2	TCP	294	49528 → 80 [PSH, ACK] Seq=1 Ack=1 Win=29312 Len=228 TSval=3881280725 TSecr=125667 [TCP s…
9	0.001825147	192.168.1.1	192.168.1.2	TCP	74	49530 → 80 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=3881280725 TSecr=0 WS=…
10	0.001974485	192.168.1.2	192.168.1.1	TCP	74	80 → 49530 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1 TSval=12566…
11	0.001980394	192.168.1.1	192.168.1.2	TCP	66	49530 → 80 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=3881280725 TSecr=125667
12	0.002033828	192.168.1.1	192.168.1.2	TCP	294	49530 → 80 [PSH, ACK] Seq=1 Ack=1 Win=29312 Len=228 TSval=3881280725 TSecr=125667 [TCP s…
13	0.002126146	192.168.1.1	192.168.1.2	TCP	74	49532 → 80 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=3881280726 TSecr=0 WS=…
14	0.002234992	192.168.1.2	192.168.1.1	TCP	74	80 → 49532 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1 TSval=12566…
15	0.002240710	192.168.1.1	192.168.1.2	TCP	66	49532 → 80 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=3881280726 TSecr=125667
16	0.002294153	192.168.1.1	192.168.1.2	TCP	294	49532 → 80 [PSH, ACK] Seq=1 Ack=1 Win=29312 Len=228 TSval=3881280726 TSecr=125667 [TCP s…

The next characteristic we determined was a flood of segmented TCP packets.



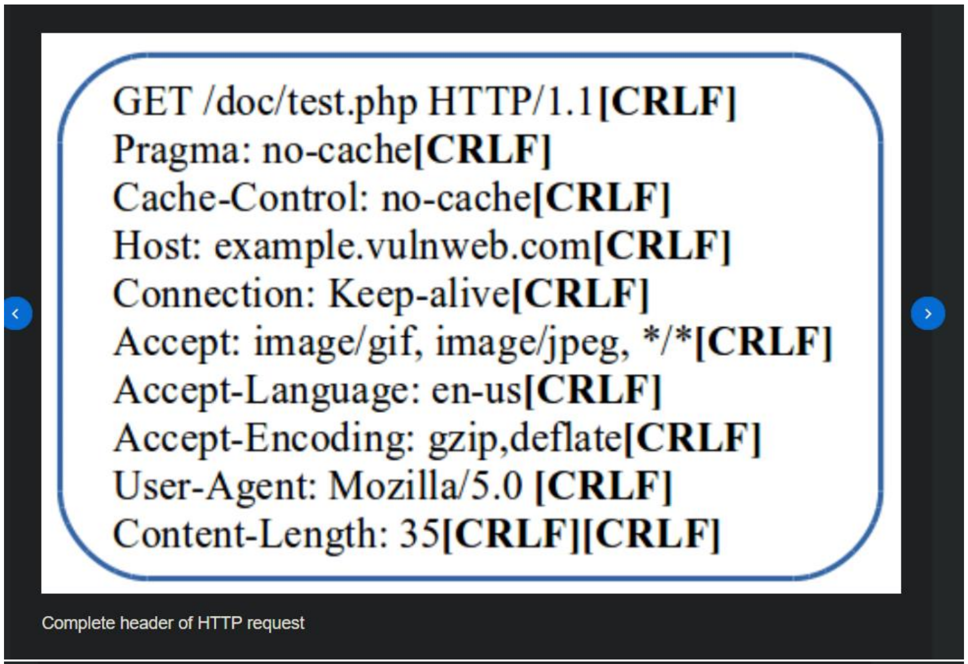
```
No.  Time          Source         Destination    Protocol Length Info
 412 0.203924733   192.168.1.2    192.168.1.1    TCP          66 80 → 49526 [ACK] Seq=1 Ack=237 Win=66560 Len=0 TSval=
 413 0.212639557   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 414 0.212678604   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 415 0.212684706   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 416 0.212689994   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 417 0.212696303   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 418 0.212702209   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 419 0.212708047   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 420 0.212713796   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 421 0.212719755   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 422 0.212726080   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 423 0.212735354   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 424 0.212741801   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 425 0.212747884   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 426 0.212753823   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 427 0.212759920   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
```

These segmented TCP packets were also determined to be containing incomplete headers, as they only contained the sequence “0d0a” in Hex (which stands for “CRLF”), instead of “0d0da0d0a” (which stands for “CRLF CRLF”) as required by the HTTP protocol specification (RFC 2616) to indicate the finish point of the header [3]. Without the terminating sequence, the victim server ends up waiting for the terminating sequence, which in turn exhausts its resources, because it tries to keep the connection alive waiting for the next segment.



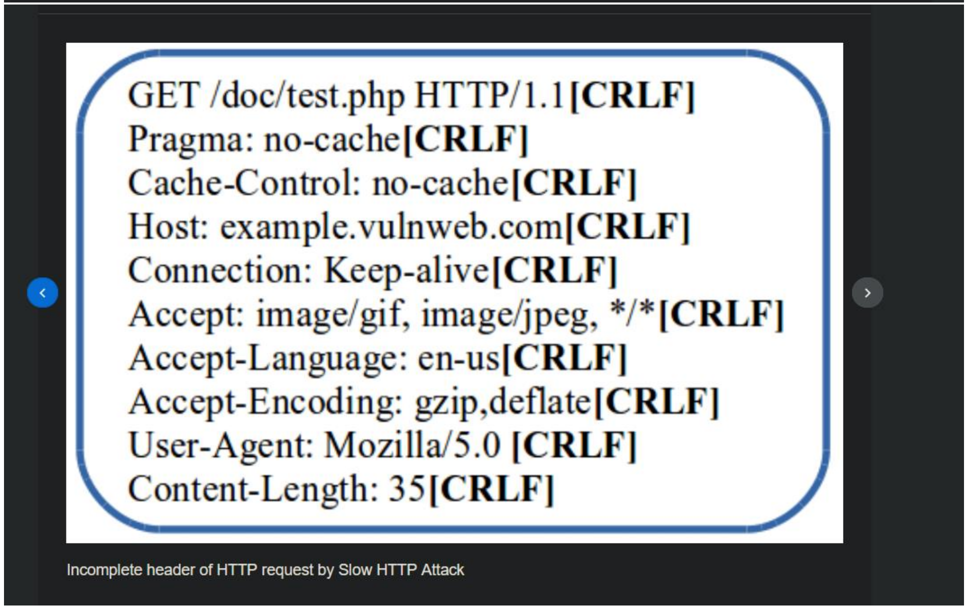
```
 436 0.212826444   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 437 0.212827262   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 438 0.212827693   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 439 0.212828072   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]
 440 0.212828447   192.168.1.1    192.168.1.2    TCP          74 GET / HTTP/1.1  [TCP segment of a reassembled PDU]

        Timestamp value: 3881280936
        Timestamp echo reply: 125668
  ▸ [SEQ/ACK analysis]
  ▸ [Timestamps]
    TCP payload (8 bytes)
    TCP segment data (8 bytes)

0000  08 00 27 33 ce 25 08 00  27 a1 b6 e6 08 00 45 00   ··'3·%·· '·····E·
0010  00 3c 3a 23 40 00 40 06  7d 45 c0 a8 01 01 c0 a8   ·<:#@·@· }E······
0020  01 02 c1 a6 00 50 78 ec  ec c9 bb 75 5a 79 80 18   ·····Px· ···uZy··
0030  00 e5 83 82 00 00 01 01  08 0a e7 57 a5 a8 00 01   ········ ···W····
0040  ea e4 58 2d 61 3a 20 62  0d 0a                     ··X-a: b ··
```

The following textbook images show a comparison between a complete vs. Incomplete HTTP header [4, 5]:



```
GET /doc/test.php HTTP/1.1[CRLF]
Pragma: no-cache[CRLF]
Cache-Control: no-cache[CRLF]
Host: example.vulnweb.com[CRLF]
Connection: Keep-alive[CRLF]
Accept: image/gif, image/jpeg, */*[CRLF]
Accept-Language: en-us[CRLF]
Accept-Encoding: gzip,deflate[CRLF]
User-Agent: Mozilla/5.0 [CRLF]
Content-Length: 35[CRLF][CRLF]
```

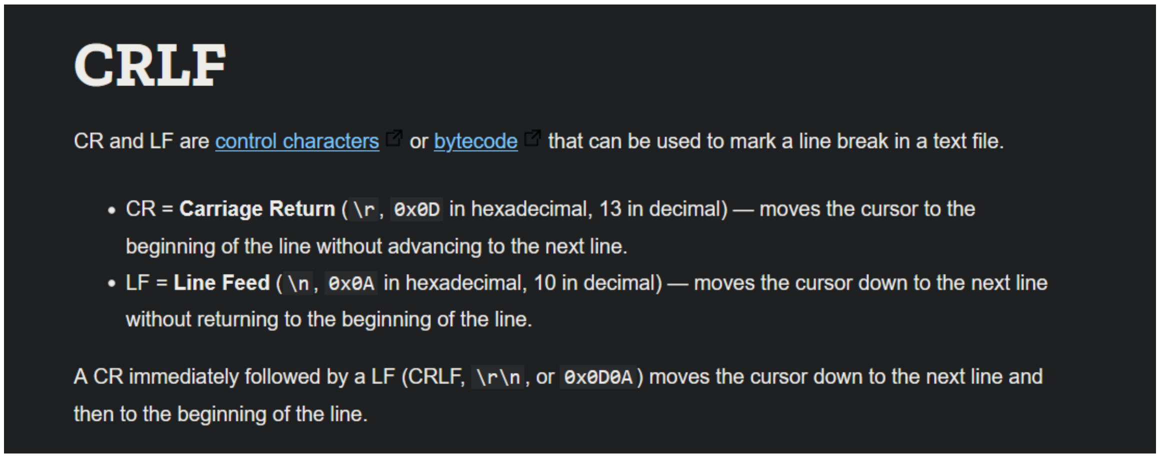
Complete header of HTTP request



```
GET /doc/test.php HTTP/1.1[CRLF]
Pragma: no-cache[CRLF]
Cache-Control: no-cache[CRLF]
Host: example.vulnweb.com[CRLF]
Connection: Keep-alive[CRLF]
Accept: image/gif, image/jpeg, */*[CRLF]
Accept-Language: en-us[CRLF]
Accept-Encoding: gzip,deflate[CRLF]
User-Agent: Mozilla/5.0 [CRLF]
Content-Length: 35[CRLF]
```

Incomplete header of HTTP request by Slow HTTP Attack

The following excerpt from the Mozilla developer's documentation shows an explanation of the CRLF signal [6]:



CRLF

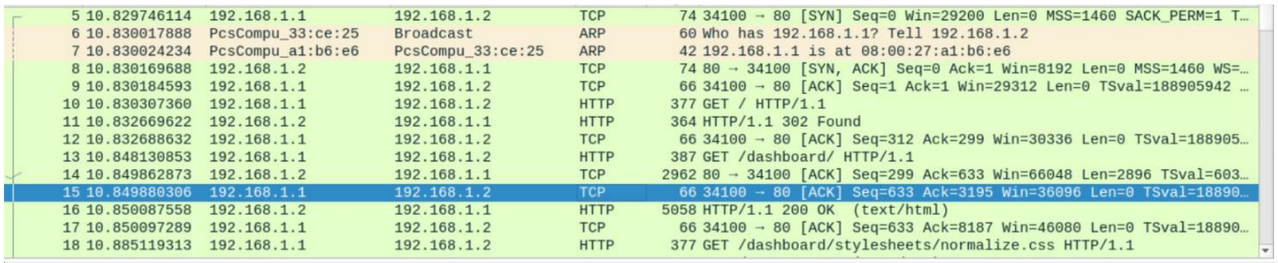
CR and LF are control characters or bytecode that can be used to mark a line break in a text file.

- CR = **Carriage Return** (`\r`, `0x0D` in hexadecimal, 13 in decimal) — moves the cursor to the beginning of the line without advancing to the next line.
- LF = **Line Feed** (`\n`, `0x0A` in hexadecimal, 10 in decimal) — moves the cursor down to the next line without returning to the beginning of the line.

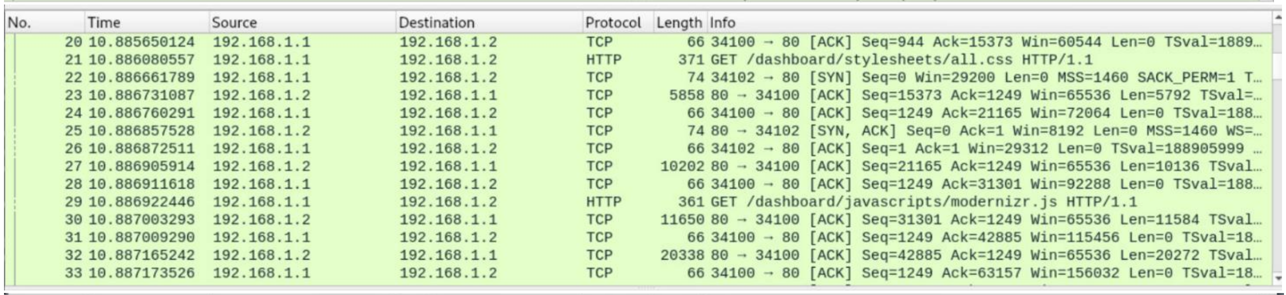
A CR immediately followed by a LF (CRLF, `\r\n`, or `0x0D0A`) moves the cursor down to the next line and then to the beginning of the line.

Our team chose the sequence of segmented TCP packets to be the first feature our application will detect.

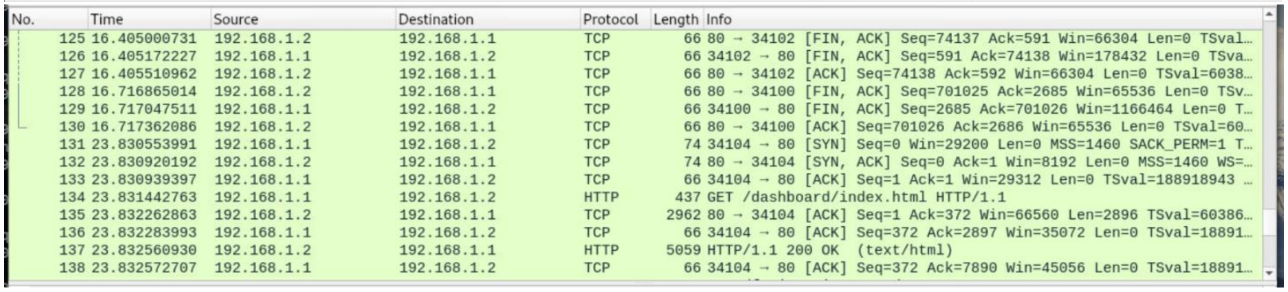
In comparison, the following screenshots show the normal network traffic in the machine when no DoS attack is performed:



```
   5 10.829746114  192.168.1.1        192.168.1.2        TCP        74 34100 → 80 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 T…
   6 10.830017888  PcsCompu_33:ce:25  Broadcast          ARP        60 Who has 192.168.1.1? Tell 192.168.1.2
   7 10.830024234  PcsCompu_a1:b6:e6  PcsCompu_33:ce:25  ARP        42 192.168.1.1 is at 08:00:27:a1:b6:e6
   8 10.830169688  192.168.1.2        192.168.1.1        TCP        74 80 → 34100 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=…
   9 10.830184593  192.168.1.1        192.168.1.2        TCP        66 34100 → 80 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=188905942 …
  10 10.830307360  192.168.1.1        192.168.1.2        HTTP      377 GET / HTTP/1.1
  11 10.832669622  192.168.1.2        192.168.1.1        HTTP      364 HTTP/1.1 302 Found
  12 10.832688632  192.168.1.1        192.168.1.2        TCP        66 34100 → 80 [ACK] Seq=312 Ack=299 Win=30336 Len=0 TSval=188905…
  13 10.848130853  192.168.1.1        192.168.1.2        HTTP      387 GET /dashboard/ HTTP/1.1
  14 10.849862873  192.168.1.2        192.168.1.1        TCP      2962 80 → 34100 [ACK] Seq=299 Ack=633 Win=66048 Len=2896 TSval=603…
  15 10.849880306  192.168.1.1        192.168.1.2        TCP        66 34100 → 80 [ACK] Seq=633 Ack=3195 Win=36096 Len=0 TSval=18890…
  16 10.850087558  192.168.1.2        192.168.1.1        HTTP     5058 HTTP/1.1 200 OK  (text/html)
  17 10.850097289  192.168.1.1        192.168.1.2        TCP        66 34100 → 80 [ACK] Seq=633 Ack=8187 Win=46080 Len=0 TSval=18890…
  18 10.885119313  192.168.1.1        192.168.1.2        HTTP      377 GET /dashboard/stylesheets/normalize.css HTTP/1.1
```



```
No.   Time          Source           Destination      Protocol Length Info
  20 10.885650124  192.168.1.1      192.168.1.2      TCP        66 34100 → 80 [ACK] Seq=944 Ack=15373 Win=60544 Len=0 TSval=1889…
  21 10.886080557  192.168.1.1      192.168.1.2      HTTP      371 GET /dashboard/stylesheets/all.css HTTP/1.1
  22 10.886661789  192.168.1.1      192.168.1.2      TCP        74 34102 → 80 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 T…
  23 10.886731087  192.168.1.2      192.168.1.1      TCP      5858 80 → 34100 [ACK] Seq=15373 Ack=1249 Win=65536 Len=5792 TSval=…
  24 10.886760291  192.168.1.1      192.168.1.2      TCP        66 34100 → 80 [ACK] Seq=1249 Ack=21165 Win=72064 Len=0 TSval=188…
  25 10.886857528  192.168.1.2      192.168.1.1      TCP        74 80 → 34102 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=…
  26 10.886872511  192.168.1.1      192.168.1.2      TCP        66 34102 → 80 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=188905999 …
  27 10.886905914  192.168.1.2      192.168.1.1      TCP     10202 80 → 34100 [ACK] Seq=21165 Ack=1249 Win=65536 Len=10136 TSval…
  28 10.886911618  192.168.1.1      192.168.1.2      TCP        66 34100 → 80 [ACK] Seq=1249 Ack=31301 Win=92288 Len=0 TSval=188…
  29 10.886922446  192.168.1.1      192.168.1.2      HTTP      361 GET /dashboard/javascripts/modernizr.js HTTP/1.1
  30 10.887003293  192.168.1.2      192.168.1.1      TCP     11650 80 → 34100 [ACK] Seq=31301 Ack=1249 Win=65536 Len=11584 TSval…
  31 10.887009290  192.168.1.1      192.168.1.2      TCP        66 34100 → 80 [ACK] Seq=1249 Ack=42885 Win=115456 Len=0 TSval=18…
  32 10.887165242  192.168.1.2      192.168.1.1      TCP     20338 80 → 34100 [ACK] Seq=42885 Ack=1249 Win=65536 Len=20272 TSval…
  33 10.887173526  192.168.1.1      192.168.1.2      TCP        66 34100 → 80 [ACK] Seq=1249 Ack=63157 Win=156032 Len=0 TSval=18…
```

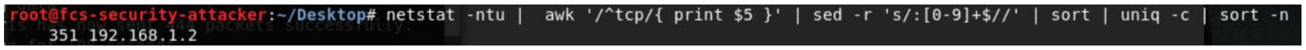


```
No.   Time          Source           Destination      Protocol Length Info
 125 16.405000731  192.168.1.2      192.168.1.1      TCP        66 80 → 34102 [FIN, ACK] Seq=74137 Ack=591 Win=66304 Len=0 TSval…
 126 16.405172227  192.168.1.1      192.168.1.2      TCP        66 34102 → 80 [FIN, ACK] Seq=591 Ack=74138 Win=178432 Len=0 TSva…
 127 16.405510962  192.168.1.2      192.168.1.1      TCP        66 80 → 34102 [ACK] Seq=74138 Ack=592 Win=66304 Len=0 TSval=6038…
 128 16.716865014  192.168.1.2      192.168.1.1      TCP        66 80 → 34100 [FIN, ACK] Seq=701025 Ack=2685 Win=65536 Len=0 TSv…
 129 16.717047511  192.168.1.1      192.168.1.2      TCP        66 34100 → 80 [FIN, ACK] Seq=2685 Ack=701026 Win=1166464 Len=0 T…
 130 16.717362086  192.168.1.2      192.168.1.1      TCP        66 80 → 34100 [ACK] Seq=701026 Ack=2686 Win=65536 Len=0 TSval=60…
 131 23.830553991  192.168.1.1      192.168.1.2      TCP        74 34104 → 80 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 T…
 132 23.830920192  192.168.1.2      192.168.1.1      TCP        74 80 → 34104 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=…
 133 23.830939397  192.168.1.1      192.168.1.2      TCP        66 34104 → 80 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=188918943 …
 134 23.831442763  192.168.1.1      192.168.1.2      HTTP      437 GET /dashboard/index.html HTTP/1.1
 135 23.832262863  192.168.1.2      192.168.1.1      TCP      2962 80 → 34104 [ACK] Seq=1 Ack=372 Win=66560 Len=2896 TSval=60386…
 136 23.832283993  192.168.1.1      192.168.1.2      TCP        66 34104 → 80 [ACK] Seq=372 Ack=2897 Win=35072 Len=0 TSval=18891…
 137 23.832560930  192.168.1.2      192.168.1.1      HTTP     5059 HTTP/1.1 200 OK  (text/html)
 138 23.832572707  192.168.1.1      192.168.1.2      TCP        66 34104 → 80 [ACK] Seq=372 Ack=7890 Win=45056 Len=0 TSval=18891…
```

As can be seen from the screenshots, the traffic does not contain a flood of [PSH, ACK] flags or segmented TCP packets, and on the other hand, it contains a lot more [ACK] and [FIN, ACK] flags, indicating the request was successfully completed.

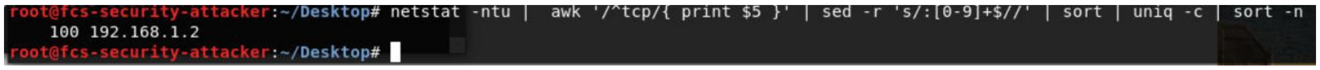
Another tool that we used to examine the network traffic during a DoS attack was Netstat. We used a custom command to group and display the number of active (open) IP connections per foreign IP address. Using this method, we could notice a big difference in the number of open IP connections in the system when the DoS attack was performed versus normal network traffic usage.

The following is an example of the result of the Netstat command when the Slowloris attack is run:



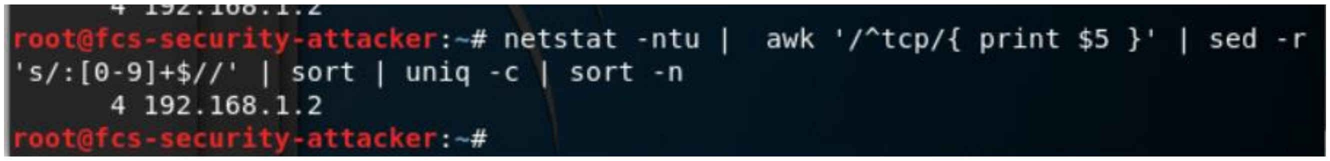
```
root@fcs-security-attacker:~/Desktop# netstat -ntu |  awk '/^tcp/{ print $5 }' | sed -r 's/:[0-9]+$//' | sort | uniq -c | sort -n
    351 192.168.1.2
```

and this is when the attack is run using the flag:"-num=100" to limit the number of connections that Slowloris opens:



```
root@fcs-security-attacker:~/Desktop# netstat -ntu |  awk '/^tcp/{ print $5 }' | sed -r 's/:[0-9]+$//' | sort | uniq -c | sort -n
    100 192.168.1.2
root@fcs-security-attacker:~/Desktop#
```

As can be seen, during the attack Slowloris opens many IP connections in the system.
In comparison, this was result during normal traffic environment, with 4 browser windows opened to access the website on the server:



```
      4 192.168.1.2
root@fcs-security-attacker:~# netstat -ntu |  awk '/^tcp/{ print $5 }' | sed -r
's/:[0-9]+$//' | sort | uniq -c | sort -n
      4 192.168.1.2
root@fcs-security-attacker:~#
```

As can be seen, during normal traffic times, the number of open IP connections in the system is significantly lower. Our team decided to use this characteristic as one of the features that the program will detect.

Feature one: Number of Segmented TCP Packets

For the first feature our team performed an analysis to determine a threshold for the number of segmented TCP packets which indicated a high likelihood of a DoS attack, based on our analysis. Our solution monitors the number of segmented TCP packets to detect the attack when the threshold is reached. We have found out that for a greater accuracy, we can combine this with an additional check for the number of [SYN, ACK] flags. This was selected as we determined that it is possible for the network to have a high rate of segmented TCP packets but in that case, usually the ratio of [SYN, ACK] flags was significantly low in comparison to the ratio of segmented packets, but when the traffic was examined during the DoS attack, the ratio of [SYN, ACK] flags was the same as the ratio of segmented packets. This was simply added to provide greater accuracy. The thresholds selected for these two characteristics were 100 for the [SYN, ACK] flags and 200 for the segmented packets.

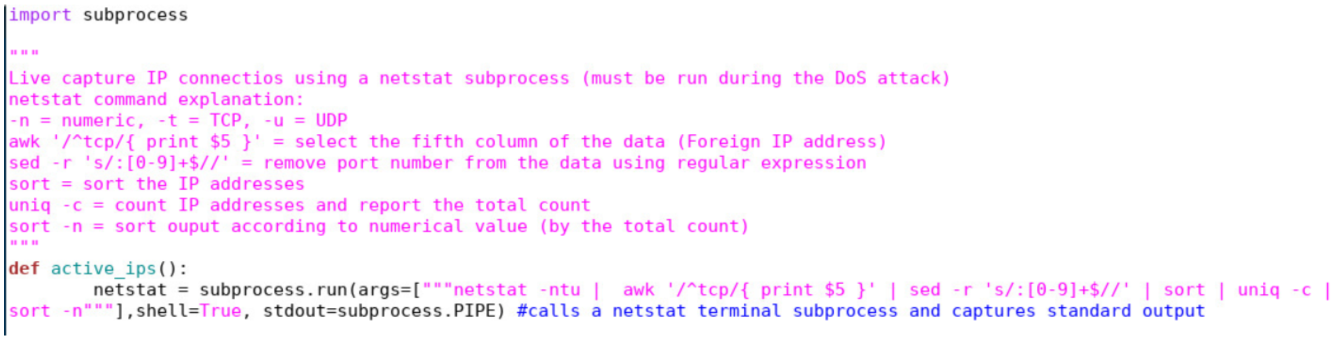
In order to extract this feature, our program used the Python built-in subprocess library to invoke a tshark command that would sniff the network traffic for a duration specified by the user when the application is started, then save the traffic to a pcap file. Following that, the application loads the traffic from the pcap file, decodes it, extracts the features, and count the number of occurrences. If the number of occurrences is above the threshold, the application displays a warning to the user indicating the high likelihood of a DoS attack.

Feature Two: Number of Open IP Connections per Foreign Address

For the second feature, our team performed an analysis to determine the threshold above which the server will be down completely or severely disrupted. We determined that the server could not operate beyond 100 open IP connections. In our solution, we monitor the number of connections by a single IP and if it exceeds this threshold, we flag it as a DoS attempt.

In order to invoke the Netstat command, our application uses the Python built-in “subprocess” library to call the command and run in the terminal.

The following excerpt from the program’s code shows the way we invoke the Netstat command using a subprocess and includes a detailed explanation of the command:



```
import subprocess

"""
Live capture IP connectios using a netstat subprocess (must be run during the DoS attack)
netstat command explanation:
-n = numeric, -t = TCP, -u = UDP
awk '/^tcp/{ print $5 }' = select the fifth column of the data (Foreign IP address)
sed -r 's/:[0-9]+$//' = remove port number from the data using regular expression
sort = sort the IP addresses
uniq -c = count IP addresses and report the total count
sort -n = sort ouput according to numerical value (by the total count)
"""
def active_ips():
        netstat = subprocess.run(args=["""netstat -ntu |  awk '/^tcp/{ print $5 }' | sed -r 's/:[0-9]+$//' | sort | uniq -c |
sort -n"""],shell=True, stdout=subprocess.PIPE) #calls a netstat terminal subprocess and captures standard output
```

When the subprocess finished, our program then takes the output, decodes it, and loops through the output. For every group of IP-connections, the program extracts the number indicating the sum as returned from the Netstat command. It checks if the number is above the threshold of 100. If the number returned is above the threshold, the program will display a warning and show the number of connections and the IP address they belong to.

Network Traffic Flows:

To generate the network traffic flows our program loads the pcap file that was saved when the network was sniffed and generates a log file (with a name selected by the user), containing the traffic flows with a source address, source port, destination address, destination port and protocol for each.

Additional Comments:

The webserver we used to simulate the DoS attack against was Apache installed in XAMPP on the Windows VM. Even though our application was developed in the Kali machine, which is the machine we used to generate the DoS traffic from, the idea is that the application could be transferred to any other Linux machine to detect DoS-like traffic. In addition, it was important to note that there was a misunderstanding regarding the disallowed use of a tool in the application, this means that the machine running our application must have tshark installed.

In addition, the application needs to be run using Python3.

Bibliography

[1] https://kb.mazebolt.com/knowledgebase/ack-psh-flood/

[2] https://osqa-ask.wireshark.org/questions/20423/pshack-wireshark-capture/

[3] https://kb.mazebolt.com/knowledgebase/slowloris-attack/

[4] https://www.researchgate.net/figure/Complete-header-of-HTTP-request_fig4_323194627

[5] https://www.researchgate.net/figure/ncomplete-header-of-HTTP-request-by-Slow-HTTP-Attack_fig5_323194627

[6] https://developer.mozilla.org/en-US/docs/Glossary/CRLF