Advances in Science, Technology and Engineering Systems Journal Vol. 5, No. 2, 293-298 (2020)

www.astesj.com

ASTESJ

ISSN: 2415-6698

Scapy Scripting to Automate Testing of Networking Middleboxes

Dr. Minal Moharir[1,*], Karthik Bhat Adyathimar[2], Dr. Shobha G[3], Vishal Soni[4]

[1]*Associate Professor, Computer Science and Engineering, RV College of Engineering, Bengaluru-560059, India*

[2]*Student, Computer Science and Engineering, RV College of Engineering, Bengaluru-560059, India*

[3]*Professors, Computer Science and Engineering, RV College of Engineering, Bengaluru-560059, India*

[4]*Senior Manager, Engineering at Citrix Systems Inc, Bengaluru-560042, India*

ARTICLE INFO

Article history:
Received: 28 December, 2019
Accepted: 22 February, 2020
Online: 23 March, 2020

Keywords:
Automatic Identification System
Anomaly Detection
Vessel traffic Behavior

ABSTRACT

Middleboxes like load balancers are being used by all the corporations to manage and support their infrastructures. These devices see a large amount of bandwidth every day. This might include a range of protocols which might be varying from network layer to the application layer. This traffic can be corrupt or malicious thus causing these devices to fail or get exploited. Citrix NetScaler Application Delivery Controller (ADC) is one of such ADC which provides supple transportation services for conventional, containerized and microservice applications from the data centre or any cloud. NetScaler supports robust security, excellent L4-L7 load balancing, authentic Global server load balancing (GSLB), and enhanced uptime. Scapy is one of the most powerful and interactive packet manipulation software. Scapy is a powerful network manipulation module which is distributed as a python module. Python being a powerful scripting language helps in exploring this in various use cases. In this paper we discuss using Scapy with Python to script application layer protocols with controlled packet structure which helps in testing these middleboxes. We use these modules to explore Citrix Netscaler ADC which is widely being used. Our present work involves development of server-client Model for the following protocols namely FTP, HTTPS and TFTP. FTP protocol developed using scapy has support for both IPv4 and IPv6. All these scripts built using scapy are open sourced.

1. Introduction

Scapy is a software developed in Python. It supports low level packet manipulation. It can be used to build tools to support testing and exploitation of networking modules. Scapy can easily interpret packets of a different variety of protocols, push them on the wire, and capture them. It can send requests and responses. The various functions of Scapy tool is as shown in figure 1. Scapy can handle most traditional undertakings like filtering, tracerouting, examining, unit tests, assaults or system revelation without much of a stretch [1]. Scapy can supplant hping, arpspoof, arp-sk, arping, p0f and even a few sections of Nmap, tcpdump, and tshark.

Scapy likewise performs extremely well on a considerable measure of other explicit assignments that most different tools can't deal with, such as sending invalid packets, infusing your own 802.11 layers, joining procedures (VLAN hopping ARP store harming, VOIP translating on WEP scrambled channel). The thought is straightforward. Scapy primarily completes couple of major functions such as sending packets and in turn accepting responses. We can build the packets with the required parameter let that be a TCP/UDP or even the lower layer details like window size, acknowledgement number. This has the huge favourable position over tools like Nmap or hping that only lets you deal with known packets. Scapy lets the user to listen to the packets on the network, filter the required ones and alter/forge/forward them [2].

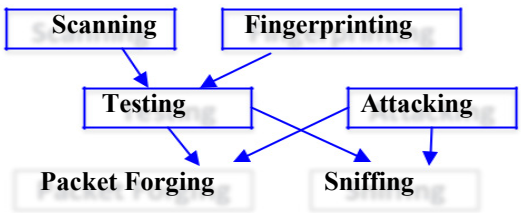

Figure 1: Scapy Functions

* Dr. Minal Moharir, minalmoharir@rvce.edu.in

2. Methodology of Protocol Enhancement to Scapy

Scapy has a Python module which lets a developer script the required features and test them easily. It is currently supported with both Python2 and Python3. It can be simply imported and its function for packet manipulation can be used. This was made use of to develop different Server-Client modules of protocols FTP, TFTP and HTTPS. These were the major requirement for the testing of Citrix Netscaler ADC. The methodology followed for packet manipulation to Scapy is shown in figure 2[3].

To achieve easy prototyping initially a TCP module is built using Scapy scripting. This lets us write any application layer protocol over it. Scapy has easy to use function for listening on the network i.e packet sniffing with the packet filters. This lets us intervene with an ongoing TCP connection on the network. This TCP module had 2 parts, listener and responder. The listener is a background thread which listens to the packets on the network with the given filter and acknowledge them. This was achieved by maintaining Sequence and Acknowledge number on which atomic operations for read and update were performed. Once the TCP data is received, this was placed in a synchronized Queue which then can be accessed using the APIs. This was built so that any higher layer protocol built can make use of it directly without worrying for the lower level details of packet filters and packet parsing. This module opens sourced and is later used with FTP and HTTPS. TFTP is a UDP protocol and it is simpler to build hence no such module had to be scripted. The main objectives of this paper is, to explore Scapy Tool with its different features support. Further to enhance TFTP, FTP, HTTPS protocol support to scapy.

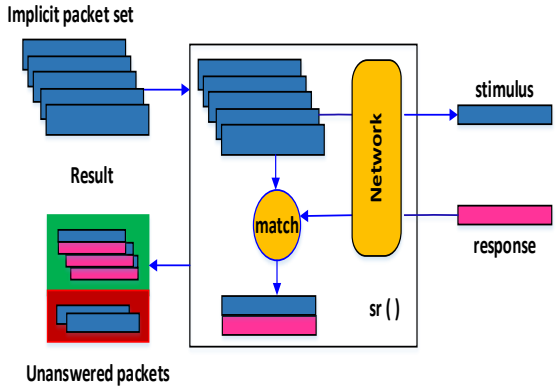

Figure 2: Packet Manipulation in Scapy

3. Protocol Enhancement to Scapy

3.1. FTP

FTP: File Transfer Protocol (FTP) is one of the most popular and preferred Internet protocols for sending and receiving huge files between computer systems over TCP/IP connections. FTP is a best example of client-server architecture. FTP is implemented on two communications channels between purchaser (client) and server. FTP comprised of two channels namely control and data channel. The overall architecture controlling is implemented by control channel. The file transmission is carried out using data channel. The clients send connection request to server before downloading any data from server side. FTP allows purchaser to upload, download, remove, and rename, flow and replica files on a server. The working of FTP protocol is as shown in figure 3. Generally, a user officially gets activated on to the FTP server to manage their desired records or files. On the other hand servers make a few or all in their content to be had without login, also referred to as anonymous FTP [4].

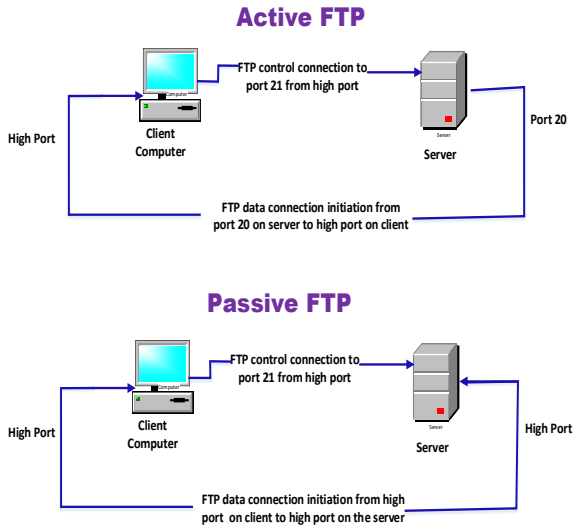

Figure 3 FTP working Principle

FTP sessions can be implemented in two modes namely passive and active mode. The active mode is implemented by initiating a client's a command channel request. In response the server initiates a data connection and starts sending data. On the other hand, in passive mode, the server uses the command channel to send the client the information it needs to open a data channel. As in passive mode the client initiates all connections, it works well across firewalls and Network Address Translation gateways. The code snippet used to implement FTP client is as shown in figure 4.

```
class FTPClient:

    __passive_cmd = ['LIST', 'STOR', 'RETR']

    def __init__(self, src, dst, sport, dport, verbose=False, logfile=None):

        self.tcp_connection = TCP_IPv4(src, dst, sport, dport)

        self.tcp_connection.handshake()

        self.close = False

        self.logfile = logfile

        self.logger_thread = Thread(target=self.logger)
        self.logger_thread.start()

        self.passive_port = None
        self.passive_mode = False
```

Figure 4: Code Snippet FTP client

Now a day's most of the file transfer applications are implemented using HTTP. Moreover, FTP is still widely used to transfer files "behind the scenes" for banking applications, the services which are used to build websites like Wix or SquareSpace and etc. It is also used in Web browsers, to download new applications [5]. The code snippet used to implement FTP server is as shown in figure 5.

TFTP service runs on well-known UDP port of 69. TFTP has to furnish its own session support as it used UDP an unreliable connectionless service. The file transferred using TFTP account for an independent exchange. The requesting client node sends either an RRQ (read request) or WRQ (write request) packet, along with the filename and the transfer mode to be used. As a response a server sends ACK (acknowledgement) packet to a received DATA packet if it is a WRQ message. Server responds with a DATA packet if it is an RRQ message (the client port is shown). On receiving each ACK message the sending host sends numbered DATA packets to the destination host. The last message contains a full-sized block of data. The numbered ACK packets is sent by the destination node for each received DATA packet [7]. The code snippet used to implement TFTP client is as shown in figure 7

```
44
45      # initialize the fields
46      def __init__(self, src, dst, sport, dport, seqno, ackno):
47          """
48          Initializes the src, dst parameters.
49          """
50          self.src = src
51          self.dst = dst
52
53          self.sport = sport
54          self.dport = dport
55
56          self.tcp_conn = TCP_IPv4(src, dst, sport, dport, seqno, ackno)
57          print(src, dst, sport, dport)
58          self.tcp_conn.listener.connection_open = True
59
60
```

Figure 5 Code Snippet FTP Server

3.2. TFTP

The tiny and fastest file transfer can be implemented using Trivial File Transfer Protocol (TFTP). TFTP is companion of User Datagram transport Protocol (UDP) which is layered on the UDP. It can be used along with Internet Protocol both the version IPv4 or IPv6. TFTP is an elementary simple file transfer protocol. It was developed around 1980. TFTP provides functionality to copy files across a network (a very basic form of FTP). It is officially published in [RFC2347]. As it is so simple, it occupies very small amount of memory which makes it more convenient to use. Therefore, booting or loading the configuration of systems is implemented using TFTP. It includes thin client, wireless base stations without any storage and routers. TFTP does not support any security features such as authentication or encryption mechanisms. All the files in the TFTP directory can be directly accessible to the user. The working of TFTP protocol is as shown in figure 6. The absence of security feature makes TFTP dangerous over the open Internet which makes TFTP suitable only on private local area networks. It is an alternative to FTP when FTP is costlier or tougher to implement. The example of these services includes down-loading firmware, software and configuration data to network devices [6].

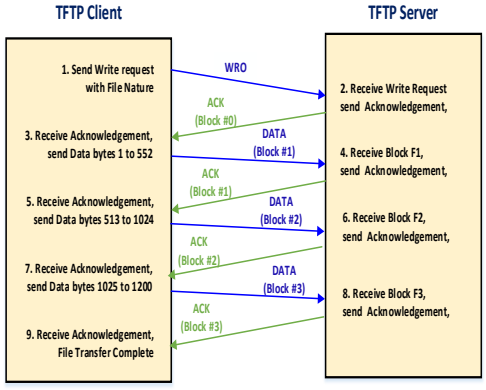

Figure 6: TFTP working Principle

TFTP service runs on well-known UDP port of 69. TFTP has to furnish its own session support as it used UDP an unreliable connectionless service. The file transferred using TFTP account for an independent exchange. The requesting client node sends either an RRQ (read request) or WRQ (write request) packet, along with the filename and the transfer mode to be used. As a response a server sends ACK (acknowledgement) packet to a received DATA packet if it is a WRQ message. Server responds with a DATA packet if it is an RRQ message (the client port is shown). On receiving each ACK message the sending host sends numbered DATA packets to the destination host. The last message contains a full-sized block of data. The numbered ACK packets is sent by the destination node for each received DATA packet [7]. The code snippet used to implement TFTP client is as shown in figure 7.

```
114 # Main client class
115 class TFTPClient:
116     # Initialize fields
117     def __init__(self, src, dst, sport, dport):
118         self.mode = "octet"
119         self.src = src
120         self.dst = dst
121         self.dport = dport
122         self.sport = sport
123         self.basic_pkt = IP(src=self.src, dst=self.dst)/UDP(sport=self.sport, dport=self.dport)
124         self.verbose = False
125
126     # Interactive shell
127     def interactive(self):
128         inp = ""
129         while "exit" not in inp:
130             sys.stdout.write(">> ")
131             inp = raw_input().split(' ')
132             if len(inp) == 1:
133                 inp.append(None)
134             self.run_command(inp[0], inp[1])
135             print "Done"
```

Figure 7: Code Snippet TFTP client

This is just like simple Automatic Repeat Request (ARQ) protocol, extended to retransmission of packet when there is a packet is lost. The last DATA packet must contain less than a maximum-sized block of data. It shows that it is the last block of the transfer. In TFTP data transmission happens in lockstep [8]. It means only one packet (either a block of data or an 'acknowledgement') is ever in flight on the network at any point of time. This windowing limitation makes TFTP low throughput and high latency protocol over links. The code snippet used to implement TFTP server is as shown in figure 8.

```
11  # For a get request, TFTPReader class will listen and store the file
12  class TFTPReader:
13      # Initializing fields
14      def __init__(self, src, dst, sport, dport, filename):
15          self.src = src
16          self.dst = dst
17          self.dport = dport
18          self.sport = sport
19          self.basic_pkt = IP(src=self.src, dst=self.dst)/UDP(sport=self.sport)
20          self.block = 1
21          self.filename = filename
22          self.verbose = False
23          with open(self.filename, "w") as f:
24              f.write("")
```

Figure 8: Code Snippet TFTP Server

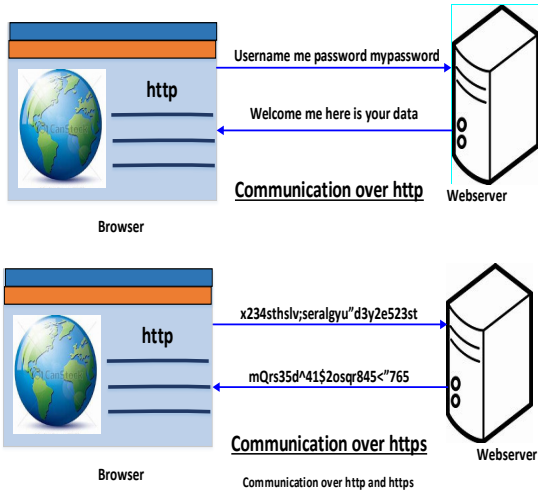

Figure 9: HTTPS working Principle

3.3. HTTPS

HTTPS is a Hyper Text Transfer Protocol Secure. The secure communication between two systems e.g. the browser and the web server is implemented using HTTPS [9]. The difference between communication over http and https is as shown in figure 9.

We can analyse form the above figure, the data transfer using http between the browser and the web server is in the hypertext format, on the other side https transfers data in the encoded format. Thus, https safeguards data from hackers for eavesdropping and changing it during the transmission between browser and webserver. Further, though the hackers able to hack the transmission, they will not be able to read it because the message is in encoded format. HTTPS uses the Secure Socket Layer (SSL) or Transport Layer Security (TLS) protocols to setup a secure link between the browser and the web server. TLS is the extended version of SSL [10].

Secure Socket Layer (SSL): SSL establishes an encrypted link between the two systems to implement data security for secure communication. The HTTPS code snippet is as shown in figure 10

```
def tls_client(ip, request, tls_version, ciphers, extensions):
    resp = None
    with TLSSocket(client=True) as tls_socket:
        try:
            tls_socket.connect(ip)
            print("Connected to server: %s" % (ip,))
        except socket.timeout:
            print("Failed to open connection to server: %s" % (ip,), file=sys.stderr)
        else:
            try:
                server_hello, server_kex = tls_socket.do_handshake(tls_version, ciphers, extensions)
            except TLSProtocolError as tpe:
                print("Got TLS error: %s" % tpe, file=sys.stderr)
            else:
                resp = tls_socket.do_round_trip(TLSPlaintext(data=request))
                tls_socket.close()

        if CONTEXT:
            print(tls_socket.tls_ctx)
    return resp
```

Figure 10 Code Snippet HTTPS

The communication can be in the form of browser to server, server to server or client to server. The main aim of SSL to ensure secure and private data transfer between the two systems. The https is essentially http over SSL. SSL sets up an encrypted link using an SSL certificate. This certificate is also known as a digital certificate.

4. Results

Scapy uses the Python interpreter as a command board. That means that you can directly use the Python language (assign variables, use loops, define functions, etc.) The paper is implemented by creating wrapper classes in Python to support different protocol using Scapy.

FTP module [11] has both server and client modules. This also has support for both IPv4 and IPv6. One of the features of the client module is to let the user specify the source IP and port for communication. This is not supported by any of the existing FTP clients. This helps in analyzing the working of the load balancer on how it performs when a large number of requests come in with different source IP/port. To achieve this, there is one more feature of selecting the number of concurrent connections to the server. Now this lets us test the middleboxes when the traffic floods in. This also supports different commands to be used on different concurrent connections with the help of command file as input. All these features are exposed as command line parameters. Also, all the features can be read with the help command (python client.py -h). Also further information is explained in detail in repo's readme.

The FTP protocol support is as shown in the following figures. The wrapper class was developed to support FTP with Scapy for IPV4 and IPV6 shown in figure 11 and figure 12 respectively.

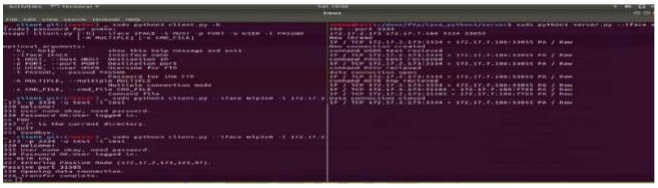

Figure 11: FTP IPV4 sessions

The file transfer with IP format with different read/write session is shown in screen shot for both IPV4 and IPV6.

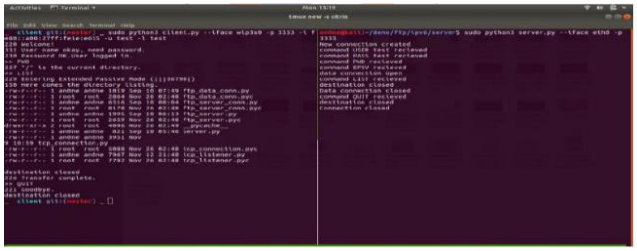

Figure 12 FTP IPV6 sessions

The implementation also supports multiple files transfer to single connection or single file transfer to multiple connections as shown in figure 13 and figure 14 respectively.

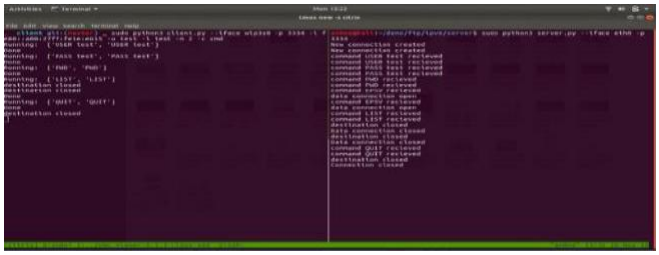

Figure 13 FTP IPV6 Multiple connections

The output shows different connection has received same file.

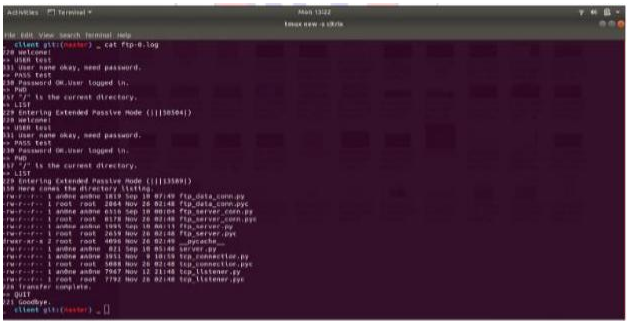

Figure 14 FTP IPV6 Multiple connections

Similarly HTTPS module was built to support various parameters to be used during SSL/TLS negotiation. This made use of one more Python module scapy-ssl_tls. This has support for selecting the Ciphersuite, TLS version as well as TLS extensions. All the features are exposed as command line parameters and can be viewed using help (python client.py -h). The HTTP request to be sent can be given as an input file specifying all the HTTP parameters and the output will be written to the file specified in the command line. It also supports curl like command by specifying the URL in the command line.

The execution of the developed wrapper class to support HTTPS with Scapy is shown in figure 15.

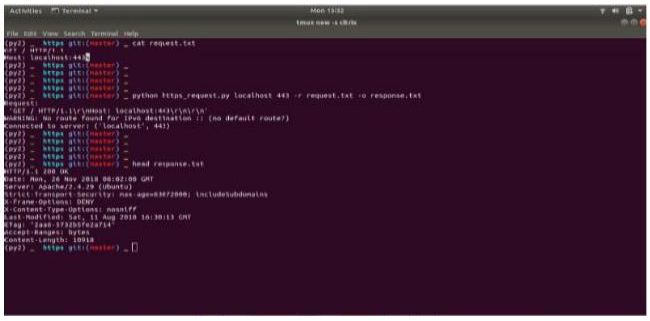

Figure 15 HTTPs client

TFTP module was also built using Scapy. TFTP is a simple request response protocol and has a simpler architecture. Similar to FTP various features like multiple concurrent requests, source IP/port options etc which are exposed as command line parameters. The connectionless small is transfer is generally implemented using trivial File Transfer (TFTP). The TFTP client server protocol support is as shown in figure 16 and 17 respectively

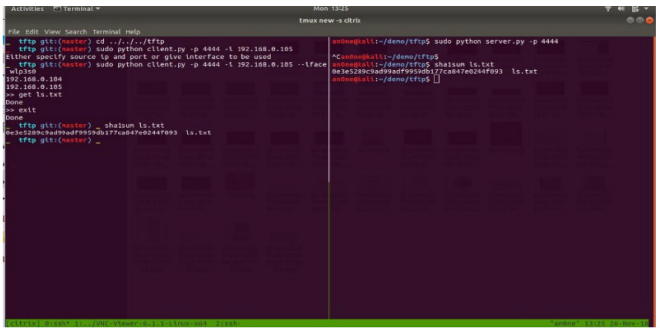

Figure 16 TFTP Client usage

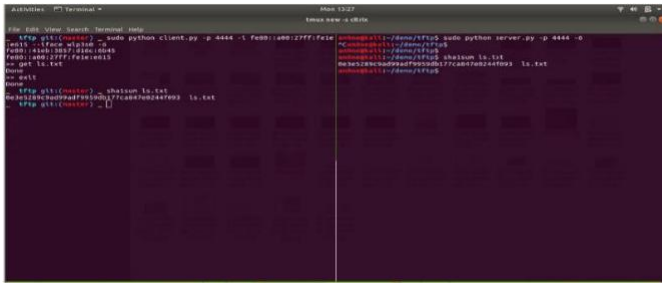

Figure 17 TFTP Server usage

5. Conclusion

There have been issues where when the load balances see a new/broken packet structures it fails to parse and causing undefined behavior. This cannot be found out general test cases since a malicious user is capable of injecting malicious packets which are not handled by the design. These cases can be very frequent and the middleboxes needs to be robust. Here we have developed python modules for protocols like FTP, TFTP and HTTPS where the packets can be generated with varied parameters like controlled TCP source IP/port, number of parallel connections, encryption parameters etc. This lets us test the load balancer with various packet structure which might not be a part of everyday traffic but might be capable of crashing or compromising of the middleboxes. For this purpose, we develop the protocol support to the system such that it we can craft our own packets from the raw structure and manipulate as we need. Using Scapy we can add additional functionalities to the protocol and use it. With Scapy it is easy to play with packets and hence it is a very powerful tool.

The enhancement to this paper can implemented as addition of data link layer protocol support to the scapy, build interactive GUI for usage, Secure protocols such as SFTP, FTPS etc can be implemented.

Conflict of Interest

The authors declare no conflict of interest.

Acknowledgment

The authors would like to thank Team Citrix for their support and guidance while implementing this project work.

References

[1] RFC4346] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, DOI 10.17487/RFC2246, January 1999, <https://www.rfc-editor.org/info/rfc2246> SSL/TLS packet structure, handshake protocol implementation, ciphers supported. Creating reliable and private connection

[2] [RFC4346] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, DOI 10.17487/RFC4346, April 2006, <https://www.rfceditor. org/info/rfc4346>. Change in ciphers supported from previous versions. Difference and improvements in security parameters.

[3] Rohith Raj S, Rohith R, Minal Moharir, Shobha G, SCAPY- A powerful interactive packet manipulation program, IEEE International Conference on Networking, Embedded and Wireless Systems (ICNEWS), 27-28 Dec. 2018

[4] Khamar Ali Shaikh, A Karthik Bhat, Minai Moharir, A Survey on SSL Packet Structure, IEEE 2nd International Conference on Computational Systems and Information Technology for Sustainable Solution (CSITSS), 21-23 Dec. 2017.DOI: 10.1109/CSITSS.2017.8447634

[5] A. Soumya Mahalakshmi et al., "A study of tools to develop a traffic generator for L4 – L7 layers", 2016 International Conference on Wireless Communications Signal Processing and Networking (WiSPNET), pp. 114-118, 2016 DOI: 10.1109/WiSPNET.2016.7566102
[6] S Bansal, N Bansal, "Scapy – A Python Tool For Security Testing", J Comput Sci Syst Biol, vol. 8, pp. 140-159, 2015System Application," Ph.D Thesis, Chongqing University, 2005.
[7] https://github.com/karthikbhata97/ScapyTCP
[8] https://github.com/karthikbhata97/scapy-ftp
[9] https://github.com/tintinweb/scapy-ssl_tls
[10] https://github.com/karthikbhata97/scapy