CNS Firewall Report

- Ajinkya Mulay [EE14BTECH11040]
- SAFWAN MAHMOOD [ES14BTECH11017]

Task 1:

Network Setup:

The setup used for this assignment is, Host1 (Host Ubuntu), Firewall (Virtual Machine), Host2 (Host Ubuntu). So, the Host1 generates packets of customized headers and sends it to the Firewall Virtual Machine (VM), where the packet is checked against a basic Firewall Rule and depending on the rule, it is either dropped or forwarded to the Host2.
Between Host1 and the Firewall VM, a raw socket packet is used, wherein any type of protocol, interface, port and addresses can be sent to the firewall. In between the Firewall and Host2 depending on the incoming protocol, a socket of that protocol is created and the packet is forwarded on this socket. Note that bridged network is used in the Host1 - Firewall VM link while in the Firewall VM - Host2, a normal socket based on IP address is sent.
We assume throughout the assignment that Host2 is protected by the firewall.
Only 1 VM was used since multiple VMs were slowing the host PC to a great extent. Even then two network interfaces were created which maintained the whole purpose of using a VM and sending data over the network.

Packet Format:

We have created our own packet headers and these along with our payload/data are packed together using the python library struct which are sent to Firewall. While packing we have a fixed format which maintains the headers and their lengths. This helps while unpacking the packet and get back all the data as well as the addresses and port information. The format of the packet is shown in the figure below:

```
    srcport = 565
    dstport = 5556
    syn=0
    ack=0
header = struct.pack('!HHIIHHHHHHHH',4,6,28,20,list_d[0],list_d[1],list_d[2],list_d[3],list_s[0],list_s[1],list_s[2],list_s[3])
tcp = struct.pack('!HHLL',srcport,dstport,syn,ack)
    udp= struct.pack('!HHHH',srcport,dstport,0,0)
    icmp = struct.pack('!BBH',9,0,1000)
    n=50
```

Packed Packet Format

Sending Data to Firewall and unpacking:

Socket was created using the Macros, AF_PACKET and the SOCK_RAW to get a raw socket (On sending device). This was bind to the “eth0” interface and the Virtual Machine settings made sure that the bridge network was active on the “eth0” interface of the host. That way every packet on that interface is obtained by the VM. The packet was sent with the method ‘socket.send’ and was received on a forever while loop using the ‘socket.recvfrom’ method.

```
while True:
    packet = s.recvfrom(65565)

    #packet string from tuple
    packet = packet[0]

    #parse ethernet header
    eth_length = 14

    eth_header = packet[:eth_length]
    eth = unpack('!6s6sH' , eth_header)
```

Receiving and one part of the unpacking

Sending data to Host2 from Firewall (If rules allow it):

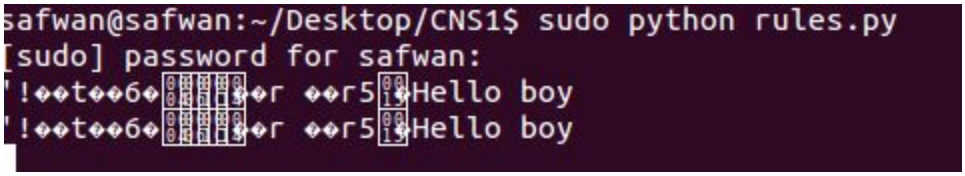
Once the packet is unpacked and the protocol (TCP, UDP, ICMP) has been identified the socket for the same protocol is created and the data is sent to a receiver listening on the same port which lies in the host ubuntu. Note here the “eth0” binding is not used. 3 Receivers were created for TCP, UDP and ICMP.

```
    if verify_rules('INCOMING','TCP',source_port) and verify_rules('INCOMING','MAC',eth_addr(packet[6:12])) and verify_rules('INCOMING'

    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    addr = (d_addr,dest_port)

        sock.connect(addr)
    sock.sendall(packet)
print("Packet verified\n")
```

If the rules are verified then the SOCK_STREAM (for tcp) is created and data is sent (Similar for UDP and ICMP)

Data Received at Firewall VM:

Data Forwarded to the Host 2 from the Firewall: (Here the random symbols are the headers, while data is the same as seen in the screenshot above and below)



Task 2:

The packets are custom formatted which is specified in the Host-1. We receive the packets in the firewall and then unpack them. We extract information like source address, destination IP and port, protocol type and header length,etc. We then check according to the rule book.
In Task 2, along with the standard rules an API had to be developed to handle a Ruleset Table. This was developed by keeping a dictionary to manage the rules. The Rules created had the following Parts:

1.) Interface Type: “Incoming” or “Outgoing”, that is the packet going from Host-1 unprotected network to protected network Host-2 as Incoming and vice versa for Outgoing.
2.) Protocol Type: We check the packet’s protocol type (here we check TCP,UDP,ICMP).
3.) IP address: We check for IP addresses which there in the blacklist and block them and also check for their version (i.e. IPv4 or IPv6).
4.) MAC address: We check for MAC addresses which there in the blacklist and block them and also check for their version.
5.) Port Range: We check whether the port number is in permissible range or not.

```
#Rules Dictionary Basic
rules_table = {
    #Default Rules
    "INCOMING": {
        #Beginning port and Ending Port
        "TCP": [(0,50)],
        "UDP": [(0,50)],
        "ICMP": [(0,50)],
        "MAC": ["90:48:9a:c1:05:7f"],
        "IPv4": ["192.168.10.104"],
        "IPv6": ["fe80::ce02:3a20:884a:5504"]
    },
    "OUTGOING": {
        "TCP": [(0,50)],
        "UDP": [(0,50)],
        "ICMP": [(0,50)],
        "MAC": ["90:48:9a:c1:05:7f"],
        "IPv4": ["192.168.10.104"],
        "IPv6": ["fe80::ce02:3a20:884a:5504"]
    }
}
```

Default Dictionary

```
def main_run ():
    #Menu
    opti = input("Optimized version of adding rules? 1. Yes 2. No")
    try:
        opti = int(opti)
    except ValueError:
        print ("Enter an integer as option")
    if opti != 1 and opti != 2:
        print ("Wrong option, exiting")
        sys.exit(0)
    menu_on = True
    while menu_on:
        option = input("What operation to be implemented in the rule table: 1. Add 2. Delete 3. Update 4. Print 5. Exit\n")
        try:
            option = int(option)
        except ValueError:
            print ("Enter an integer as option")
        if option == 1: #ADD
            first, second, third = rules_menu ()
            #opti (first, second, third)
            if first != -1 and second != -1 and third != -1:
                if opti == 1 and not optimize_rules(first, second, third):
                    print ("Redundant Rule, improvised or ignored!")
                elif opti == 1 and third1 in rules_table[first1][second1]:
                    print ("Rule already exists!")
                else:
                    rules_table[first][second].append(third)
            print ("Done Adding Rule:",first,second,third)
        elif option == 2: #DELETE
            first, second, third = rules_menu ()
            if first != -1 and second != -1 and third != -1:
                if third in rules_table[first][second]:
                    rules_table[first][second].remove(third)
                else:
                    print ("Rule to delete does not exist")
        elif option == 3: #UPDATE
            print ("Which rule do you want to update")
            first1, second1, third1 = rules_menu ()
            if first1 != -1 and second1 != -1 and third1 != -1:
                if third1 in rules_table[first1][second1]:
                    print ("What do you want to update in that rule")
                    first2, second2, third2 = rules_menu ()
                    if opti == 1 and not opimize_rules(first2, second2, third2):
                        print ("Redundant Rule, improvised or ignored!")
                    elif opti == 1 and third2 in rules_table[first2][second2]:
```

Part of the API Menu used to add, delete, update and print the Rules Table

This Ruleset table is a global dictionary which has a few default rules hard-coded while the rules inside can be managed by the methods provided:

1.) Add 2.) Delete 3.) Update 4.) Print

For each Interface - Protocol Pair a list is maintained to store the ports or the addresses depending on what type of rule it is. The verify_rules function is tuned to drop the packets, but we can set it either ways. So right now if a rule is satisfied then a False is sent, so that the packet can be dropped when received by the firewall.

```
def verify_rules (first, second, third):
    if second == "TCP" or second == "UDP" or second == "ICMP":
        for ports in rules_table[first][second]:
            if ports[0] <= third and third <= ports[1]:
                return False
    if second == "MAC" or second == "IPv4" or second == "IPv6":
        if third in rules_table[first][second]:
            return False
    return True
```

Function to Verify the Rules against the Table, called in the main code

Task 3:

Here, the performance of the program (Unoptimized) is seen on various number of rules. As the rules were increasing, so was the number of if conditions and this led to further increase in time complexity. The rules are generated randomly. Since, the packets were being generated in a continuous manner, for large number of packets we could calculate the time and the graphs could be plotted:

Results for the same are given in part 4a:

Task 4a: (This is the part which we have done)

Initially when the program starts the user is asked whether they want an optimized version of rule set or a normal rule set. Now the optimization is done by 2 methods:

1.) *Identifying and updating redundant rules:*

Whenever a new rule (other than the default hard coded rules) is being added, it is cross checked with other rules and if the same rule exists then the new rule is not added. Also, if port range of the new rule already exists or the range is very close to an existing rule then the existing rule is updated instead of creating another new rule. For example if an existing port range is from (100,120) and the new one is from (101, 130) for the same interface and protocol then the existing rule is changed to contain the port range of (100,130). This helps in reducing the number of if condition checks and is especially beneficial when there are a lot of rules to be checked.

2.) *Caching the rules:*

Whenever a packet comes in, no matter whether it is dropped or forwarded, there will be a set of parameters it has and a corresponding decision to forward or drop. Now, these parameters are stored as the key in a dictionary and its decision is set as the value. These

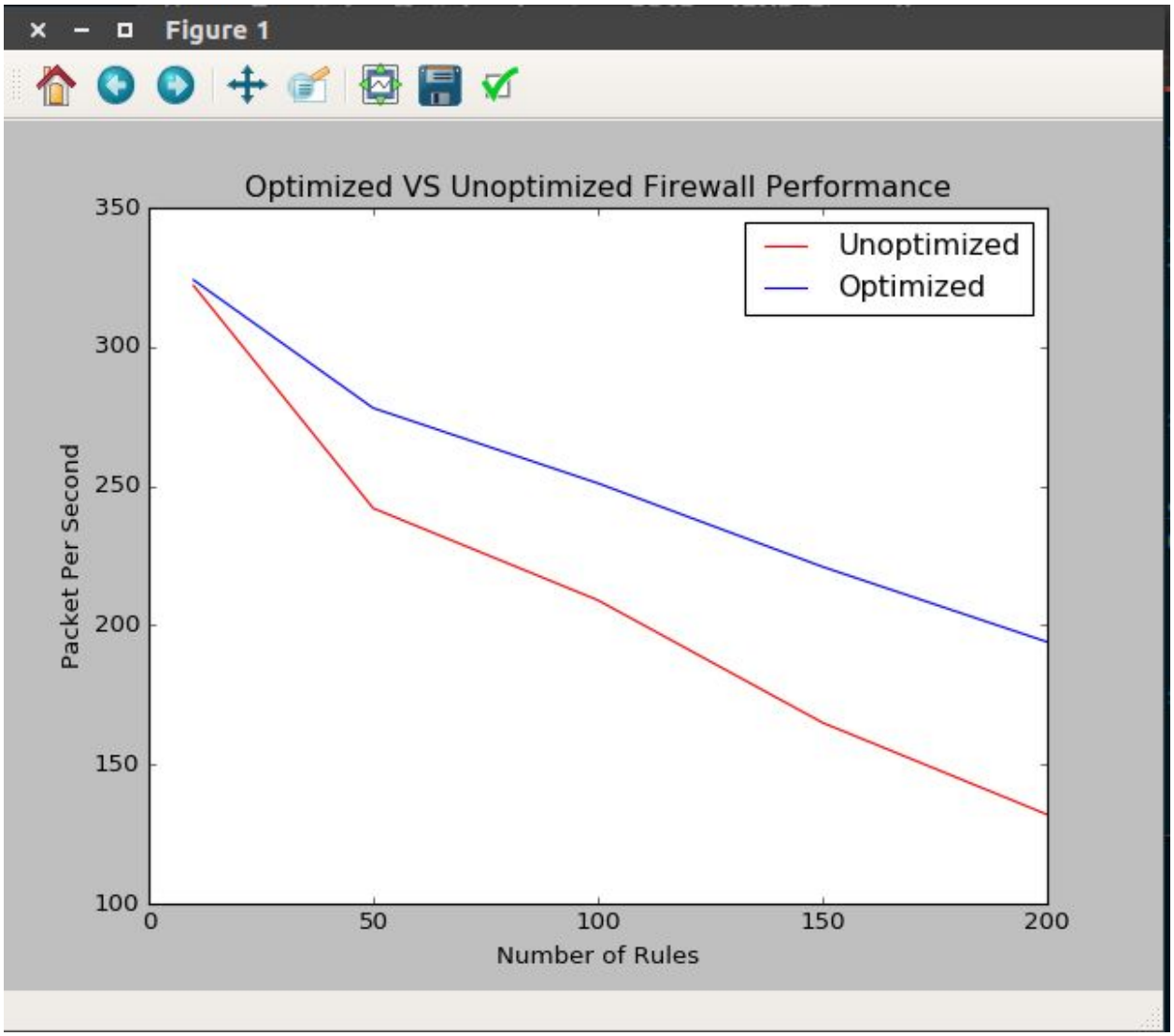
parameters include: (Interface Type, Protocol Type, MAC Address, IP Address, Port) and the decision is True if the packet is to be forwarded and False if it is to be dropped. This is stored in a global dictionary and every time a new packet is check with this dictionary first to see if the exact type of packet has passed in or not. If it has then we do not need to go into the firewall ruleset and instead we can just take the decision depending on the earlier made decision. This is not a vulnerability, since this sending device is not in the blacklist initially meaning that, it is not a threat. This, optimization is the most important since if we consider a big file being sent from Host1 to Host2, then it will have multiple packets of the exact same parameters and we can easily transfer this file without the firewall rules hampering it constantly. Instead the firewall check will happen only on the first packet. If the packet has been already verified then it will print a message indicating the same.

Performance of the above optimizations were compared with the normal execution (User is given the option to choose between optimized and unoptimized version in the program's beginning). The results of the same and the corresponding graphs are:

Packet Sent: 5,000

Unoptimized and Optimized:

Number of Rules	PPS (Optimized)	PPS (Unoptimized)
10	324	322
50	278	242
100	251	209
150	221	165
200	194	132



Inferences:

We can clearly see that for both the Optimized and Unoptimized firewalls, the PPS drops as number of rules increases. This is since the number of ‘IF conditions’ to be processed for each packet increases. But the optimized Firewall can and does perform much better because of the 2 reasons mentioned above. (i.e. the cache and redundant rules)

Task 4b: (These have not been implemented, due to lack of time but the process has been explained below)

The two types of attacks which can be easily detected are Port Scan and DoS Attack. Since we already have rules which can identify parameters like IP/MAC addresses and port ranges we can detect such attacks.

For Port Scan:

Port Scan consists of a an attacking device sending data to the victim's device on multiple ports in a short amount of time to detect the services running on those ports. This can cause the victim to be more vulnerable to attacks.

We can see that if a device with the same MAC address or IP address for the same interface and any protocol is sending data on different ports under a specific amount of time, then we can conclude that the attack is Port Scan. For example, if data is sent by the device with the same MAC and/or IP address on 10 different ports in 20 seconds then we can conclude that the attack is a Port Scan. Here, 20 is a threshold which can be changed.

For DoS:

DoS (Denial Of Service) is used by an attacking device to inject packets into the victim's device so as to stop them from accessing the service or network. So, very large number of packets are sent in a very short time continuously so as to disrupt the victim's device's operation.

This can be detected by checking the MAC and/or IP address of every device and any device which has sent (suppose 500 packets in under 2 seconds) a large number of packets to our device is considered as a DoS attack and then can be dropped forever.