SWE 4503

Writing A Sniffer

Sniffer

- Network sniffers allow you to see packets entering and exiting a target machine. As a result, they have many practical uses before and after exploitation.
- **Our sniffer's main goal is to discover hosts on a target network.** Attackers want to be able to see all of the potential targets on a network so that they can focus their reconnaissance and exploitation attempts.
- When we send a UDP datagram to a closed port on a host, that host typically sends back an ICMP message indicating that the port is unreachable. This ICMP message tells us that there is a host alive, because if there was no host, we probably wouldn't receive any response to the UDP datagram. **It's essential, therefore, that we pick a UDP port that won't likely be used.**
- The process of accessing raw sockets in Windows is slightly different than on its Linux brethren, but we want the flexibility to deploy the same sniffer to multiple platforms.

Packet Sniffing on Windows and Linux

- Windows requires us to set some additional flags through a socket input/output control **(IOCTL)**, which enables **promiscuous mode** on the network interface card.
- **IOCTL** stands for Input/Output Control, a mechanism that provides a way for user-space programs to communicate with kernel-mode components, such as device drivers.
- User-space programs often need to interact with hardware or low-level system resources. Since direct access to the kernel is not allowed for security and stability reasons, IOCTL acts as an intermediary.
- **Promiscuous mode** is a special mode for a network interface card (NIC) that allows it to capture all network packets on the local network segment, not just those addressed to its own MAC address.
- By default, a NIC processes only the packets addressed to it or broadcast packets. To enable promiscuous mode, the operating system and NIC need to be instructed to accept all packets. This requires setting specific flags or options at the NIC driver level.

Implementation

- Windows (nt): Uses socket.IPPROTO_IP for raw socket operations because raw ICMP sockets may require special handling or permissions.
- Other OS (e.g., Linux, macOS): Uses socket.IPPROTO_ICMP since ICMP is the standard protocol for tasks like sending ping requests or working with ICMP messages.

```
import socket
import os

# host to listen on
HOST = '192.168.1.203'

def main():
    # create raw socket, bin to public interface
    if os.name == 'nt':
        socket_protocol = socket.IPPROTO_IP
    else:
        socket_protocol = socket.IPPROTO_ICMP

 ❶ sniffer = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket_protocol)
    sniffer.bind((HOST, 0))
    # include the IP header in the capture
 ❷ sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

 ❸ if os.name == 'nt':
        sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

    # read one packet
 ❹ print(sniffer.recvfrom(65565))

    #  if we're on Windows, turn off promiscuous mode
 ❺ if os.name == 'nt':
        sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)

if __name__ == '__main__':
    main()
```

Explanation

1. Constructing our socket object with the parameters (IPv4, Raw Sockets, IP or ICMP socket protocol) necessary for sniffing packets on our network interface.
2. sniffer.setsockopt() method is used to configure options for a socket. socket.IPPROTO_IP Specifies that the option being configured is at the IP protocol level. socket.IP_HDRINCL is a socket option that stands for IP Header Include. When enabled (set to 1), this option allows the program to provide its own custom IP headers when sending packets.
3. The next step is to determine if we are using Windows and, if so, perform the additional step of sending an IOCTL to the network card driver to enable promiscuous mode.
4. Now we are ready to actually perform some sniffing, and in this case we are simply printing out the entire raw packet with no packet decoding.
5. After a single packet is sniffed, we again test for Windows and then disable promiscuous mode before exiting the script.

Execution

Open a terminal. If windows open an administrative terminal. If linux use ‘sudo’ before running it since we are going to use privileged instructions.

Windows:

Terminal 1 : python sniffer.py

Terminal 2 : ping nostarch.com

Terminal 1 output : (b'E\x00\x00T\xad\xcc\x00\x00\x80\x01\n\x17h\x14\xd1\x03\xac\x10\x9d\x9d\x00\x00g,\rv\x00\x01\xb6L\x1b^\x00\x00\x00\x00\xf1\xde\t\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567', ('104.20.209.3', 0))

Linux:

Terminal 1 : sudo python sniffer.py

Terminal 2 : ping nostarch.com

Terminal 1 output same as windows.

Decoding The IP Layer

In its current form, our sniffer receives all of the IP headers, along with any higher protocols such as TCP, UDP, or ICMP. The information is packed into binary form and, as shown previously, is quite difficult to understand. Let's work on decoding the IP portion of a packet so that we can pull useful information from it, such as the protocol type (TCP, UDP, or ICMP) and the source and destination IP addresses.

<table>
<tr><th colspan="7">Internet Protocol</th></tr>
<tr><td>Bit offset</td><td>0–3</td><td>4–7</td><td>8–15</td><td>16–18</td><td>19–31</td></tr>
<tr><td>0</td><td>Version</td><td>HDR length</td><td>Type of service</td><td colspan="2">Total length</td></tr>
<tr><td>32</td><td colspan="3">Identification</td><td>Flags</td><td>Fragment offset</td></tr>
<tr><td>64</td><td colspan="2">Time to live</td><td>Protocol</td><td colspan="2">Header checksum</td></tr>
<tr><td>96</td><td colspan="5">Source IP address</td></tr>
<tr><td>128</td><td colspan="5">Destination IP address</td></tr>
<tr><td>160</td><td colspan="5">Options</td></tr>
</table>

Implementation and Explanation

```
import ipaddress
import os
import socket
import struct
import sys

❶ class IP:
    def __init__(self, buff=None):
        header = struct.unpack('<BBHHHBBH4s4s', buff)
        self.ver = header[0] >> 4
        self.ihl = header[0] & 0xF

        self.tos = header[1]
        self.len = header[2]
        self.id = header[3]
        self.offset = header[4]
        self.ttl = header[5]
        self.protocol_num = header[6]
        self.sum = header[7]
        self.src = header[8]
        self.dst = header[9]
```

- The first format character (in our case, <) always speci the endianness of the data, or the order of bytes withir binary number. C types are represented in the machine native format and byte order. In this case, we're on Kal (x64), which is little-endian. In a little-endian machine, least significant byte is stored in the lower address, an most significant byte in the highest address.
- For the IP header, we need only the format characters (1-byte unsigned char), H (2-byte unsigned short), and byte array that requires a byte-width specification; 4s n a 4-byte string).
- With struct, there's no format character for a nybble (a 4-bit unit of data, also known as a nibble), so we have to do some manipulation to get the ver and hdrlen variables from the first part of the header.

```
0   1   0   1   0   1   1   0   >> 4
----------------------------
0   0   0   0   0   1   0   1
```

```
        0   1   0   1   0   1   1   0
AND     0   0   0   0   1   1   1   1
----------------------------
        0   0   0   0   0   1   1   0
```

Continued

```
❷ # human readable IP addresses
  self.src_address = ipaddress.ip_address(self.src)
  self.dst_address = ipaddress.ip_address(self.dst)

  # map protocol constants to their names
  self.protocol_map = {1: "ICMP", 6: "TCP", 17: "UDP"}
  try:
      self.protocol = self.protocol_map[self.protocol_num]
  except Exception as e:
      print('%s No protocol for %s' % (e, self.protocol_num))
      self.protocol = str(self.protocol_num)
```

Continued

```
def sniff(host):
    # should look familiar from previous example
    if os.name == 'nt':
        socket_protocol = socket.IPPROTO_IP
    else:
        socket_protocol = socket.IPPROTO_ICMP

    sniffer = socket.socket(socket.AF_INET,
                            socket.SOCK_RAW, socket_protocol)
    sniffer.bind((host, 0))
    sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

    if os.name == 'nt':
        sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)
```

Continued

```
    try:
        while True:
            # read a packet
          ❸ raw_buffer = sniffer.recvfrom(65535)[0]
            # create an IP header from the first 20 bytes
          ❹ ip_header = IP(raw_buffer[0:20])
            # print the detected protocol and hosts
          ❺ print('Protocol: %s %s -> %s' % (ip_header.protocol,
                                                ip_header.src_address,
                                                ip_header.dst_address))

    except KeyboardInterrupt:
        # if we're on Windows, turn off promiscuous mode
        if os.name == 'nt':
            sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
        sys.exit()

if __name__ == '__main__':
    if len(sys.argv) == 2:
        host = sys.argv[1]
    else:
        host = '192.168.1.203'
    sniff(host)
```

Execution

Open a terminal and run “sudo python sniffer_ip_header_decode.py”

Now, because Windows is pretty chatty, you’re likely to see output immediately. The authors tested this script by opening Internet Explorer and going to www.google.com, and here is the output from our script:

```
Protocol: UDP 192.168.0.190 -> 192.168.0.1
Protocol: UDP 192.168.0.1 -> 192.168.0.190
Protocol: UDP 192.168.0.190 -> 192.168.0.187
Protocol: TCP 192.168.0.187 -> 74.125.225.183
Protocol: TCP 192.168.0.187 -> 74.125.225.183
Protocol: TCP 74.125.225.183 -> 192.168.0.187
Protocol: TCP 192.168.0.187 -> 74.125.225.183
```

Because we aren’t doing any deep inspection on these packets, we can only guess what this stream is indicating. Our guess is that the first couple of UDP packets are the Domain Name System (DNS) queries to determine where google.com lives, and the subsequent TCP sessions are our machine actually connecting and downloading content from their web server.

Continued

To perform the same test on Linux, we can ping google.com, and the results will look something like this:

```
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
```

You can already see the limitation: we are seeing only the response and only for the ICMP protocol. But because we are purposefully building a host discovery scanner, this is completely acceptable.

Decoding ICMP

Destination Unreachable Message		
0–7	8–15	16–31
Type = 3	Code	Header checksum
Unused		Next-hop MTU
IP header and first 8 bytes of original datagram's data		

```
❶ class ICMP:
      def __init__(self, buff):

          header = struct.unpack('<BBHHH', buff)
          self.type = header[0]
          self.code = header[1]
          self.sum = header[2]
          self.id = header[3]
          self.seq = header[4]
```

Sniffer

```
import ipaddress
import os
import socket
import struct
import sys
import threading
import time

# subnet to target
SUBNET = '192.168.1.0/24'
# magic string we'll check ICMP responses for
MESSAGE = 'PYTHONRULES!' ❶

class IP:
--snip--

class ICMP:
--snip--
```

```
# this sprays out UDP datagrams with our magic message
def udp_sender(): ❷
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sender:
        for ip in ipaddress.ip_network(SUBNET).hosts():
            sender.sendto(bytes(MESSAGE, 'utf8'), (str(ip), 65212))
```

```
class Scanner: ❸
    def __init__(self, host):
        self.host = host
        if os.name == 'nt':
            socket_protocol = socket.IPPROTO_IP
        else:
            socket_protocol = socket.IPPROTO_ICMP

        self.socket = socket.socket(socket.AF_INET,
                                    socket.SOCK_RAW, socket_protocol)
        self.socket.bind((host, 0))

        self.socket.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

        if os.name == 'nt':
            self.socket.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)
```

```
def sniff(self): ❹
    hosts_up = set([f'{str(self.host)} *'])
    try:
        while True:
            # read a packet
            raw_buffer = self.socket.recvfrom(65535)[0]
            # create an IP header from the first 20 bytes
            ip_header = IP(raw_buffer[0:20])
            # if it's ICMP, we want it
            if ip_header.protocol == "ICMP":
                offset = ip_header.ihl * 4
                buf = raw_buffer[offset:offset + 8]
                icmp_header = ICMP(buf)
                # check for TYPE 3 and CODE
                if icmp_header.code == 3 and icmp_header.type == 3:
                    if ipaddress.ip_address(ip_header.src_address) in ❺
                                    ipaddress.IPv4Network(SUBNET):

                        # make sure it has our magic message
                        if raw_buffer[len(raw_buffer) - len(MESSAGE):] == ❻
                                    bytes(MESSAGE, 'utf8'):
                            tgt = str(ip_header.src_address)
                            if tgt != self.host and tgt not in hosts_up:
                                hosts_up.add(str(ip_header.src_address))
                                print(f'Host Up: {tgt}') ❼
```

```
# handle CTRL-C
except KeyboardInterrupt: ❽
    if os.name == 'nt':
        self.socket.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)

    print('\nUser interrupted.')
    if hosts_up:
        print(f'\n\nSummary: Hosts up on {SUBNET}')
    for host in sorted(hosts_up):
        print(f'{host}')
    print('')
    sys.exit()
if __name__ == '__main__':
    if len(sys.argv) == 2:
        host = sys.argv[1]
    else:
        host = '192.168.1.203'
    s = Scanner(host)
    time.sleep(5)
    t = threading.Thread(target=udp_sender) ❾
    t.start()
    s.sniff()
```

Execution

Execution in linux will be lot easier since windows requires variety of permissions. So open a terminal in kali linux and run : sudo python sniffer.py

You will get the following output:

```
Summary: Hosts up on 192.168.39.0/24
192.168.39.103
192.168.39.109
192.168.39.11
192.168.39.112
192.168.39.134 *
192.168.39.186
192.168.39.25
192.168.39.26
192.168.39.27
192.168.39.32
192.168.39.4
192.168.39.6
192.168.39.7
192.168.39.70
192.168.39.91
```