3

WRITING A 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. In some cases, you'll be able to use existing sniffing tools like Wireshark (*https://wireshark.org/*) or a Pythonic solution like Scapy (which we'll explore in the next chapter). Nevertheless, there's an advantage to knowing how to throw together your own quick sniffer to view and decode network traffic. Writing a tool like this will also give you a deep appreciation for the mature tools, as these can painlessly take care of the finer points with little effort on your part. You'll also likely pick up some new Python techniques and perhaps a better understanding of how the low-level networking bits work.

In the previous chapter, we covered how to send and receive data using TCP and UDP. This is likely how you'll interact with most network services.

But underneath these higher-level protocols are the building blocks that determine how network packets are sent and received. You'll use raw sockets to access lower-level networking information, such as the raw Internet Protocol (IP) and Internet Control Message Protocol (ICMP) headers. We won't decode any Ethernet information in this chapter, but if you intend to perform any low-level attacks, such as ARP poisoning, or are developing wireless assessment tools, you should become intimately familiar with Ethernet frames and their use.

Let's begin with a brief walk-through of how to discover active hosts on a network segment.

Building a UDP Host Discovery Tool

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.

We'll use a known behavior of most operating systems to determine if there is an active host at a particular IP address. 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. For maximum coverage, we can probe several ports to ensure we aren't hitting an active UDP service.

Why the User Datagram Protocol? Well, there's no overhead in spraying the message across an entire subnet and waiting for the ICMP responses to arrive accordingly. This is quite a simple scanner to build, as most of the work goes into decoding and analyzing the various network protocol headers. We'll implement this host scanner for both Windows and Linux to maximize the likelihood of being able to use it inside an enterprise environment.

We could also build additional logic into our scanner to kick off full Nmap port scans on any hosts we discover. That way, we can determine if they have a viable network attack surface. This is an exercise left for the reader, and we the authors look forward to hearing some of the creative ways you can expand this core concept. Let's get started.

Packet Sniffing on Windows and Linux

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. To account for this, we'll create a socket object and then determine which platform we're running on. Windows requires us to

set some additional flags through a socket input/output control (IOCTL), which enables promiscuous mode on the network interface An *input/output control (IOCTL)* is a means for user space programs to communicate with kernel mode components. Have a read here: *http://en.wikipedia.org/wiki/Ioctl*.

In our first example, we simply set up our raw socket sniffer, read in a single packet, and then quit:

```
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()
```

We start by defining the `HOST` IP to our own machine's address and constructing our socket object with the parameters necessary for sniffing packets on our network interface ❶. The difference between Windows and Linux is that Windows will allow us to sniff all incoming packets regardless of protocol, whereas Linux forces us to specify that we are sniffing ICMP packets. Note that we are using promiscuous mode, which requires administrative privileges on Windows or root on Linux. Promiscuous mode allows us to sniff all packets that the network card sees, even those not destined for our specific host. Then we set a socket option ❷ that includes the IP headers in our captured packets. The next step ❸ 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. If you're running Windows in a virtual machine, you will likely get a notification that the guest operating system is enabling promiscuous mode; you, of course, will allow it. Now we are ready to actually perform some sniffing, and in this case we are simply printing out the entire raw packet ❹ with no packet decoding. This is just to test to make sure we have the core of our sniffing code working. After a single packet is sniffed, we again test for Windows and then disable promiscuous mode ❺ before exiting the script.

Kicking the Tires

Open up a fresh terminal or *cmd.exe* shell under Windows and run the following:

```
python sniffer.py
```

In another terminal or shell window, you pick a host to ping. Here, we'll ping *nostarch.com*:

```
ping nostarch.com
```

In your first window, where you executed your sniffer, you should see some garbled output that closely resembles the following:

```
(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))
```

You can see that we've captured the initial ICMP ping request destined for *nostarch.com* (based on the appearance of the IP for *nostarch.com*, 104.20.209.3, at the end of the output). If you are running this example on Linux, you would receive the response from *nostarch.com*.

Sniffing one packet is not overly useful, so let's add some functionality to process more packets and decode their contents.

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. This will serve as a foundation for further protocol parsing later on.

If we examine what an actual packet looks like on the network, you should understand how we need to decode the incoming packets. Refer to Figure 3-1 for the makeup of an IP header.

Internet Protocol					
Bit offset	0–3	4–7	8–15	16–18	19–31
0	Version	HDR length	Type of service	Total length	
32	Identification			Flags	Fragment offset
64	Time to live		Protocol	Header checksum	
96	Source IP address				
128	Destination IP address				
160	Options				

Figure 3-1: Typical IPv4 header structure

We will decode the entire IP header (except the Options field) and extract the protocol type, source, and destination IP address. This means we'll be working directly with the binary, and we'll have to come up with a strategy for separating each part of the IP header using Python.

In Python, there are a couple of ways to get external binary data into a data structure. You can use either the `ctypes` module or the `struct` module to define the data structure. The `ctypes` module is a foreign function library for Python. It provides a bridge to C-based languages, enabling you to use C-compatible data types and call functions in shared libraries. On the other hand, `struct` converts between Python values and C structs represented as Python byte objects. In other words, the `ctypes` module handles binary data types in addition to providing a lot of other functionality, while the `struct` module primarily handles binary data.

You will see both methods used when you explore tool repositories on the web. This section shows you how to use each one to read an IPv4 header off the network. It's up to you to decide which method you prefer; either will work fine.

The ctypes Module

The following code snippet defines a new class, `IP`, that can read a packet and parse the header into its separate fields:

```
from ctypes import *
import socket
import struct

class IP(Structure):
    _fields_ = [
         ("ihl",          c_ubyte,   4),       # 4 bit unsigned char
         ("version",      c_ubyte,   4),       # 4 bit unsigned char
```

```
        ("tos",           c_ubyte,   8),       # 1 byte char
        ("len",           c_ushort, 16),       # 2 byte unsigned short
        ("id",            c_ushort, 16),       # 2 byte unsigned short
        ("offset",        c_ushort, 16),       # 2 byte unsigned short
        ("ttl",           c_ubyte,   8),       # 1 byte char
        ("protocol_num",  c_ubyte,   8),       # 1 byte char
        ("sum",           c_ushort, 16),       # 2 byte unsigned short
        ("src",           c_uint32, 32),       # 4 byte unsigned int
        ("dst",           c_uint32, 32)        # 4 byte unsigned int
    ]
    def __new__(cls, socket_buffer=None):
        return cls.from_buffer_copy(socket_buffer)

    def __init__(self, socket_buffer=None):
        # human readable IP addresses
        self.src_address = socket.inet_ntoa(struct.pack("<L",self.src))
        self.dst_address = socket.inet_ntoa(struct.pack("<L",self.dst))
```

This class creates a `_fields_` structure to define each part of the IP header. The structure uses C types that are defined in the `ctypes` module. For example, the `c_ubtye` type is an unsigned char, the `c_ushort` type is an unsigned short, and so on. You can see that each field matches the IP header diagram in Figure 3-1. Each field description takes three arguments: the name of the field (such as `ihl` or `offset`), the type of value it takes (such as `c_ubyte` or `c_ushort`), and the width in bits for that field (such as 4 for `ihl` and `version`). Being able to specify the bit width is handy because it provides the freedom to specify any length we need, not only at the byte level (specification at the byte level would force our defined fields to always be a multiple of 8 bits).

The `IP` class inherits from the `ctypes` module's `Structure` class, which specifies that we must have a defined `_fields_` structure before creating any object. To fill the `_fields_` structure, the `Structure` class uses the `__new__` method, which takes the class reference as the first argument. It creates and returns an object of the class, which passes to the `__init__` method. When we create our IP object, we'll do so as we ordinarily would, but underneath, Python invokes `__new__`, which fills out the `_fields_` data structure immediately before the object is created (when the `__init__` method is called). As long as you've defined the structure beforehand, you can just pass the `__new__` method the external network packet data, and the fields should magically appear as your object's attributes.

You now have an idea of how to map the C data types to the IP header values. Using C code as a reference when translating to Python objects can be useful, because the conversion to pure Python is seamless. See the `ctypes` documentation for full details about working with this module.

The struct Module

The `struct` module provides format characters that you can use to specify the structure of the binary data. In the following example, we'll once again define an `IP` class to hold the header information. This time, though, we'll use format characters to represent the parts of the header:

```
import ipaddress
import struct

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

        # 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"}
```

The first format character (in our case, `<`) always specifies the endianness of the data, or the order of bytes within a binary number. C types are represented in the machine's native format and byte order. In this case, we're on Kali (x64), which is little-endian. In a little-endian machine, the least significant byte is stored in the lower address, and the most significant byte in the highest address.

The next format characters represent the individual parts of the header. The `struct` module provides several format characters. For the IP header, we need only the format characters `B` (1-byte unsigned char), `H` (2-byte unsigned short), and `s` (a byte array that requires a byte-width specification; `4s` means a 4-byte string). Note how our format string matches the structure of the IP header diagram in Figure 3-1.

Remember that with `ctypes`, we could specify the bit-width of the individual header parts. 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.

Of the first byte of header data we receive, we want to assign the `ver` variable only the *high-order* nybble (the first nybble in the byte). The typical way you get the high-order nybble of a byte is to *right-shift* the byte by four places, which is the equivalent of prepending four 0s to the front of the byte, causing the last 4 bits to fall off ❶. This leaves us with only the first nybble of the original byte. The Python code essentially does the following:

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

We want to assign the `hdrlen` variable the *low-order* nybble, or the last 4 bits of the byte. The typical way to get the second nybble of a byte is to use the Boolean `AND` operator with 0xF (00001111) ❷. This applies the Boolean operation such that 0 `AND` 1 produce 0 (since 0 is equivalent to `FALSE`, and 1 is equivalent to `TRUE`). For the expression to be true, both the first part and the last part must be true. Therefore, this operation deletes the first 4 bits, as anything `AND`ed with 0 will be 0. It leaves the last 4 bits unaltered, as anything `AND`ed with 1 will return the original value. Essentially, the Python code manipulates the byte as follows:

```
      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
```

You don't have to know very much about binary manipulation to decode an IP header, but you'll see certain patterns, like using shifts and `AND` over and over as you explore other hackers' code, so it's worth understanding those techniques.

In cases like this that require some bit-shifting, decoding binary data takes some effort. But for many cases (such as reading ICMP messages), it's very simple to set up: each portion of the ICMP message is a multiple of 8 bits, and the format characters provided by the `struct` module are multiples of 8 bits, so there's no need to split a byte into separate nybbles. In the Echo Reply ICMP message shown in Figure 3-2, you can see that each parameter of the ICMP header can be defined in a struct with one of the existing format letters (`BBHHH`).

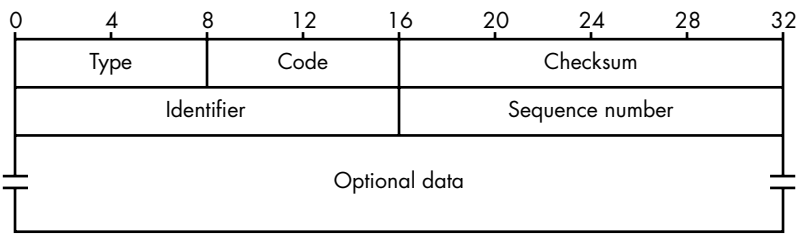

Figure 3-2: Sample `Echo Reply` ICMP message

A quick way to parse this message would be to simply assign 1 byte to the first two attributes and 2 bytes to the next three attributes:

```
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]
```

Read the `struct` documentation (*https://docs.python.org/3/library/struct.html*) for full details about using this module.

You can use either the `ctypes` module or the `struct` module to read and parse binary data. No matter which approach you take, you'll instantiate the class like this:

```
    mypacket = IP(buff)
    print(f'{mypacket.src_address} -> {mypacket.dst_address}')
```

In this example, you instantiate the IP class with your packet data in the variable `buff`.

Writing the IP Decoder

Let's implement the IP decoding routine we just created into a file called *sniffer_ip_header_decode.py*, as shown here:

```
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]
```

```
    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)

    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)

        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)
```

First, we include our `IP` class definition ❶, which defines a Python structure that will map the first 20 bytes of the received buffer into a friendly IP header. As you can see, all of the fields that we identified

match up nicely with the header structure. We do some housekeeping to produce some human-readable output that indicates the protocol in use and the IP addresses involved in the connection ❷. With our freshly minted `IP` structure, we now write the logic to continually read in packets and parse their information. We read in the packet ❸ and then pass the first 20 bytes ❹ to initialize our `IP` structure. Next, we simply print out the information that we have captured ❺. Let's try it out.

Kicking the Tires

Let's test out our previous code to see what kind of information we are extracting from the raw packets being sent. We definitely recommend that you do this test from your Windows machine, as you will be able to see TCP, UDP, and ICMP, which allows you to do some pretty neat testing (opening up a browser, for example). If you are confined to Linux, then perform the previous ping test to see it in action.

Open a terminal and type the following:

```
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.

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. We will now apply the same techniques we used to decode the IP header to decode the ICMP messages.

Decoding ICMP

Now that we can fully decode the IP layer of any sniffed packets, we have to be able to decode the ICMP responses that our scanner will elicit from sending UDP datagrams to closed ports. ICMP messages can vary greatly in their contents, but each message contains three elements that stay consistent: the type, code, and checksum fields. The type and code fields tell the receiving host what type of ICMP message is arriving, which then dictates how to decode it properly.

For the purpose of our scanner, we are looking for a type value of 3 and a code value of 3. This corresponds to the `Destination Unreachable` class of ICMP messages, and the code value of 3 indicates that the `Port Unreachable` error has been caused. Refer to Figure 3-3 for a diagram of a `Destination Unreachable` ICMP message.

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		

Figure 3-3: Diagram of `Destination Unreachable` ICMP message

As you can see, the first 8 bits are the type, and the second 8 bits contain our ICMP code. One interesting thing to note is that when a host sends one of these ICMP messages, it actually includes the IP header of the originating message that generated the response. We can also see that we will double-check against 8 bytes of the original datagram that was sent in order to make sure our scanner generated the ICMP response. To do so, we simply slice off the last 8 bytes of the received buffer to pull out the magic string that our scanner sends.

Let's add some more code to our previous sniffer to include the ability to decode ICMP packets. Let's save our previous file as *sniffer_with_icmp.py* and add the following code:

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

class IP:
--snip--

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]

def sniff(host):
--snip--
            ip_header = IP(raw_buffer[0:20])
            # if it's ICMP, we want it
         ❷ if ip_header.protocol == "ICMP":
                print('Protocol: %s %s -> %s' % (ip_header.protocol,
                        ip_header.src_address, ip_header.dst_address))
                print(f'Version: {ip_header.ver}')
                print(f'Header Length: {ip_header.ihl} TTL: {ip_header.ttl}')

                # calculate where our ICMP packet starts
              ❸ offset = ip_header.ihl * 4
                buf = raw_buffer[offset:offset + 8]
                # create our ICMP structure
              ❹ icmp_header = ICMP(buf)
                print('ICMP -> Type: %s Code: %s\n' %
                     (icmp_header.type, icmp_header.code))

        except KeyboardInterrupt:
             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)
```

This simple piece of code creates an `ICMP` structure ❶ underneath our existing `IP` structure. When the main packet-receiving loop determines that we have received an ICMP packet ❷, we calculate the offset in the raw packet where the ICMP body lives ❸ and then create our buffer ❹ and print out the `type` and `code` fields. The length calculation is based on the IP header `ihl` field, which indicates the number of 32-bit words (4-byte chunks) contained in the IP header. So by multiplying this field by 4, we know the size of the IP header and thus when the next network layer (ICMP in this case) begins.

If we quickly run this code with our typical ping test, our output should now be slightly different:

```
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
ICMP -> Type: 0 Code: 0
```

This indicates that the ping (ICMP Echo) responses are being correctly received and decoded. We are now ready to implement the last bit of logic to send out the UDP datagrams and to interpret their results.

Now let's add the use of the `ipaddress` module so that we can cover an entire subnet with our host discovery scan. Save your *sniffer_with_icmp.py* script as *scanner.py* and add the following code:

```
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()
```

This last bit of code should be fairly straightforward to understand. We define a simple string signature ❶ so that we can test that the responses are coming from UDP packets that we sent originally. Our `udp_sender` function ❷ simply takes in a subnet that we specify at the top of our script, iterates through all IP addresses in that subnet, and fires UDP datagrams at them.

We then define a `Scanner` class ❸. To initialize it, we pass it a host as an argument. As it initializes, we create a socket, turn on promiscuous mode if running Windows, and make the socket an attribute of the `Scanner` class.

The `sniff` method ❹ sniffs the network, following the same steps as in the previous example, except that this time it keeps a record of which hosts are up. If we detect the anticipated ICMP message, we first check to make sure that the ICMP response is coming from within our target subnet ❺. We then perform our final check of making sure that the ICMP response has our magic string in it ❻. If all of these checks pass, we print out the IP address of the host where the ICMP message originated ❼. When we end the sniffing process by using CTRL-C, we handle the keyboard interrupt ❽. That is, we turn off promiscuous mode if on Windows and print out a sorted list of live hosts.

The `__main__` block does the work of setting things up: it creates the `Scanner` object, sleeps just a few seconds, and then, before calling the `sniff` method, spawns `udp_sender` in a separate thread ❾ to ensure that we aren't interfering with our ability to sniff responses. Let's try it out.

Kicking the Tires

Now let's take our scanner and run it against the local network. You can use Linux or Windows for this, as the results will be the same. In the authors' case, the IP address of the local machine we were on was 192.168.0.187, so we set our scanner to hit 192.168.0.0/24. If the output is too noisy when you run your scanner, simply comment out all print statements except for the last one that tells you what hosts are responding.

```
python.exe scanner.py
Host Up: 192.168.0.1
Host Up: 192.168.0.190
Host Up: 192.168.0.192
Host Up: 192.168.0.195
```

THE IPADDRESS MODULE

Our scanner will use a library called `ipaddress`, which will allow us to feed in a subnet mask such as 192.168.0.0/24 and have our scanner handle it appropriately.

The `ipaddress` module makes working with subnets and addressing very easy. For example, you can run simple tests like the following using the `Ipv4Network` object:

```
ip_address = "192.168.112.3"

if ip_address in Ipv4Network("192.168.112.0/24"):
    print True
```

Or you can create simple iterators if you want to send packets to an entire network:

```
for ip in Ipv4Network("192.168.112.1/24"):
    s = socket.socket()
    s.connect((ip, 25))
    # send mail packets
```

This will greatly simplify your programming life when dealing with entire networks at a time, and it is ideally suited for our host discovery tool.

For a quick scan like the one we performed, it took only a few seconds to get the results. By cross-referencing these IP addresses with the DHCP table in a home router, we were able to verify that the results were accurate. You can easily expand what you've learned in this chapter to decode TCP and UDP packets as well as to build additional tooling around the scanner. This scanner is also useful for the trojan framework we will begin building in Chapter 7. This would allow a deployed trojan to scan the local network for additional targets.

Now that you know the basics of how networks work on a high and low level, let's explore a very mature Python library called Scapy.