CHAPTER 2

What You DON'T Know About Your Network

"Knowledge speaks, but wisdom listens."

Jimi Hendrix

WHAT'S RUNNING ON YOUR NETWORK MIGHT SURPRISE YOU

Modern environments boast massive infrastructures and sophisticated security technologies designed to keep the bad guys out.

What if the bad guys are already in?

Today, the defensive technology mix includes traditional firewalls, application firewalls, a demilitarized zone (DMZ), virtual private networks (VPN), anti-virus, anti-spyware, patch management infrastructures, content filters, host and network data leak protection (DLP), specialized privilege guards and security event and incident management (SEIM) solutions. Unfortunately, these systems and technologies do little to protect against new threats or hidden vulnerabilities that exist within the environment they protect. In some cases, they exist within the security solutions themselves!

In addition, the solutions today bear resemblance and similar weaknesses to those created by the French Minister of War, Andre Maginot, who in the 1930's created fortifications to protect France from a German invasion. Much like the Maginot line (see figure 2-1), modern cyber security solutions provide great protection against a direct attack, but can be circumvented by insiders through the exploitation of unknown vulnerabilities, via new attack vectors, by means of social engineering activities and can be infiltrated due to lack of deep understanding of one's own environment.

Big vs. Little

It turns out that many smaller organizations are more difficult to penetrate due to the fact that the environment is better understood by both the Information

CONTENTS

What's Running on Your Network Might Surprise You 17
Big vs. Little 17
We Care About What's Running on Our Systems 18
Why Do We Care? 19
A Quick Demonstration 21
How to Do This in Python? 23
Sample Program Output 29
OS Fingerprinting 30
OS Fingerprinting Using TCP/IP Default Header Values 30
OS Fingerprinting Using Open Port Patterns 32
What Open Ports or Services Don't You Know About? 32
How is This Useful? 33
Who's Touching Your Network? 34
Review 35
Summary Questions ... 35

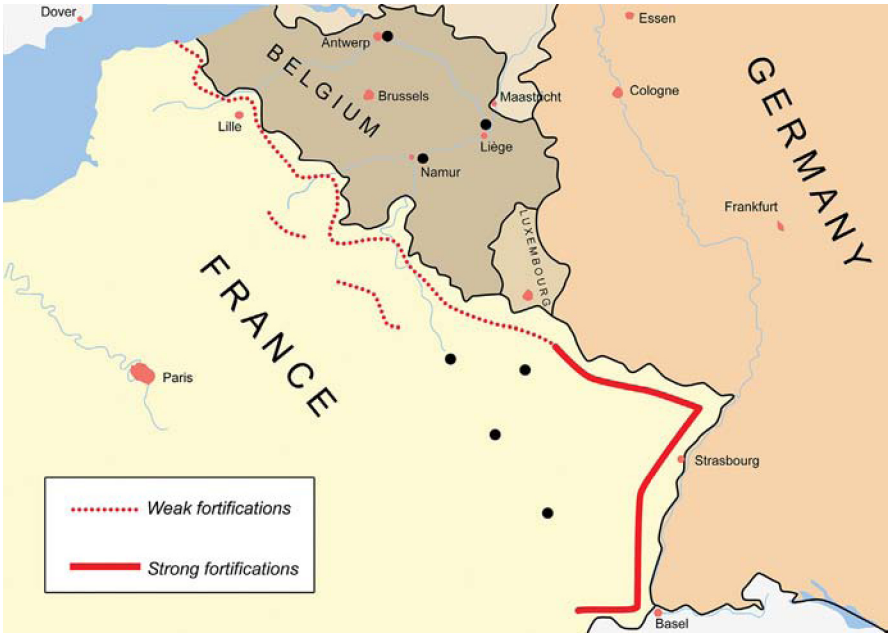

FIGURE 2-1 Map the Maginot Line.

Technology (IT) teams and the Cyber Security teams that protect them. Larger organizations in many cases have undergone numerous mergers and acquisitions along with the melding of information systems. They have also been around longer and likely employ legacy technologies, or have systems operating throughout their network that have simply been forgotten and are running services that are vulnerable.

The following statement is critically important….

> The more you know about your environment, the better you can **protect** your assets, the easier you can **detect** anomalous activity, and the faster you can **react** to new attacks and vulnerabilities.

We Care About What's Running on Our Systems

This might seem obvious as you read this, but you are likely to be surprised by systems and services that are operating on your network. We tend to think only about servers and desktop workstations, since our view of the world is that this is where the information is created, accessed and utilized. Obviously, our infrastructures are changing and what is running or attached to our network is also evolving. Let's just take a look at just a small list of devices and systems

we need to be concerned about today (I have purposely left out Servers and Desktop Workstations from the list):

- Android phones and tablets
- iOS phones and tablets
- Windows phones and tablets
- Blackberry phones and tablets
- Printers and multifunction devices (print, scan, fax)
- Copiers and Biz Centers
- Voice Over Internet Protocol (VOIP) systems
- Security cameras
- Internet radios
- Handheld personal cameras
- Near Field Communication Devices (NFC)
- Conference room phones
- Wearable technologies (fitness, surveillance see Figures 2-2–2-4)

Why Do We Care?

At the end of the day, these are all computers at their core with access to networks, the Internet and possibly your corporate infrastructure and information. The questions are:

1. Can you identify them on your network?
2. Do you know where they are located?
3. What data do they have access to?
4. Most importantly, what is the risk and potential impact they pose if compromised?

FIGURE 2-2 Wearable Camera Glasses.

FIGURE 2-3 Smart Watches.

FIGURE 2-4 Wearable Fitness Devices.

The other important aspect of the mobile, wireless, Bluetooth, wearables and NFC devices is that they tend to leave very temporal footprints. Meaning that traditional active network mapping methods may be ineffective in detecting their presence or tracking their behaviors.

Based on this brief introduction, you can see that there are significant advantages to having a firm understanding of the devices that should be attached to our networks, whether these devices are servers, workstations or mobile devices. Think of this as home-field advantage, by understanding what should be operating on your network it becomes easier to identify those devices that shouldn't be there.

As I demonstrated in Chapter 1, actively identifying devices on a network using NMAP quickly provides information about the obvious suspects. What we are looking for here are those devices that operate either in a temporal fashion or are purposely stealthy. Approaching the problem from a passive point view is different in that we have to wait for devices to reveal their presence by actively participating.

Once again we will turn to `tcpdump` to demonstrate some of the ways to capture packets in a passive manner. You might realize that I can do the same thing with Wireshark or a host of other proprietary toolsets. However, one of the problems with this approach is that in order to capture packets at the kernel level, you must be operating at a very high privilege level, and using complex and far-reaching security tools to do so is risky business. Thus my approach throughout the book will be to use simple well-known open-source technologies to perform operations at high levels of privilege. In this way we can limit the need to provide root privilege to only those processes that are absolutely necessary. Likewise our analysis tools (after we have captured the necessary packet samples) can and should operate at a user level.

A Quick Demonstration

Let's answer the following simple question. What computers on my network are hitting remote web servers? To keep things simple, I want to capture only traffic that has a destination address of Port 80. To demonstrate this, I captured some traffic off my home network with `tcpdump` using the following Linux/Unix commands:

First, I placed my eth0 adapter into promiscuous mode.

```
$ sudo ifconfig eth0 promisc
```

Translating the command

`sudo`:	Execute the command with super user privilege
`ifconfig`	Linux `ifconfig` command
`eth0`:	Specify the Ethernet adapter I wish to set
`promisc`:	Set eth0 in promiscuous mode

After completion of the command we can check the results by running `ifconfig`. As you can see the eth0 adapter is now running in promiscuous multicast mode

```
$ ifconfig eth0

eth0      Link encap:Ethernet  HWaddr 00:1e:8c:b7:6d:64

          inet6 addr: fe80::21e:8cff:feb7:6d64/64 Scope:Link

          UP BROADCAST RUNNING PROMISC MULTICAST

          MTU:1500  Metric:1

          RX packets:43842 errors:0 dropped:108

          overruns:0 frame:0

          TX packets:33 errors:0 dropped:0 overruns:0 carrier:0

          collisions:0 txqueuelen:1000

          RX bytes:4981889 (4.9 MB)  TX bytes:5723 (5.7 KB)
```

Next, I use the `tcpdump` command to collect any packets originating from source port 80.

```
$ sudo tcpdump –i eth0 –n src port 80
```

Translating the command:

`sudo:`	Run the command with super user privilege
`tcpdump:`	The command we wish to execute at privilege
`-i eth0:`	Utilize the Ethernet 0 adapter to perform the capture
`-n:`	Do not resolve IP address to name
`src port 80:`	only capture packets that have a source port of 80

As a result the command returns a barrage of data. I have snipped out the redundant entries.

```
16:37:06.559388 IP 50.62.120.26.80 > 192.168.0.22.48637: Flags [.], seq

1:1461, ack 505, win 31, length 1460

16:37:06.560713 IP 50.62.120.26.80 > 192.168.0.22.48637: Flags [.], seq

1461:2921, ack 505, win 31, length 1460

<…SNIPPED…>
```

```
16:37:53.787370 IP 108.160.165.54.80 > 192.168.0.22.48532: Flags [P.], seq
380428656:380428835, ack 1424003206, win 31624, length 179
16:37:53.889243 IP 108.160.165.54.80 > 192.168.0.22.48532: Flags [.], ack
360, win 32696, length 0
16:37:59.812185 IP 173.194.37.84.80 > 192.168.0.22.48644: Flags [.], ack
1424, win 361, options [nop,nop,sack 1 {1423:1424}], length 0
<…SNIPPED…>
16:38:09.668759 IP 23.52.91.27.80 > 192.168.0.22.48660: Flags [.], ack 445,
win 490, length 0
16:38:09.670682 IP 23.52.91.27.80 > 192.168.0.22.48660: Flags [.], seq
1:1461, ack 445, win 490, length 1460
16:38:09.670758 IP 23.52.91.27.80 > 192.168.0.22.48660: Flags [P.], seq
1461:2244, ack 445, win 490, length 783
<…SNIPPED…>
```

This results in the following unique values from a network mapping point of view

Table 2-1 Manually Identified Unique Values

Server IP	Client IP	Source Port	Destination Port
50.62.120.26	192.168.0.22	80	48637
108.160.165.54	192.168.0.22	80	48532
23.52.91.27	192.168.0.22	80	48660

How to Do This in Python?

Continuing with the theme of keeping this simple with an eye on passive network mapping, how might we approach this same solution in Python? I will add the ability to automatically generate a unique list of the Server / Client interactions over Port 443.

The script has two basic parts,

1. The Main program that:
   a. Sets up the network interface in promiscuous mode
   b. Opens a raw socket
   c. Listens and reads packets from the raw socket
   d. Calls the `PacketExtractor()` function to decode the packet
   e. Updates a list with packets that meet our port criteria
   f. Once the maximum number of packets are collected a unique list is generated
2. The `PacketExtractor()` function that:
   a. Extracts the IP Header
   b. Extracts the TCP Header
   c. Obtains the Source and Destination IP Addresses
   d. Obtains the Source and Destinations Port Numbers
   e. Makes an educated guess as to the Server vs. Client
   f. Returns a list containing ServerIP, ClientIP, ServerPort

```
# Python Script to Map Activity on a single port
# Running on Linux

# Import Standard Library Modules

import socket           # network interface library used for raw sockets
import os               # operating system functions i.e. file I/o
import sys              # system level functions i.e. exit()
from struct import *    # Handle Strings as Binary Data

# Constants

PROTOCOL_TCP = 6        # TCP Protocol for IP Layer

# PacketExtractor
#
# Purpose: Extracts fields from the IP and TCP Header
#
# Input:   packet:     buffer from socket.recvfrom() method
# Output:  list:       serverIP, clientIP, serverPort
#

def PacketExtractor(packet):

    #Strip off the first 20 characters for the ip header
    stripPacket = packet[0:20]

    #now unpack them
    ipHeaderTuple = unpack('!BBHHHBBH4s4s' , stripPacket)
```

```
# unpack returns a tuple, for illustration I will extract
# each individual values
                                        # Field Contents
verLen       = ipHeaderTuple[0]         # Field 0: Version and Length
TOS          = ipHeaderTuple[1]         # Field 1: Type of Service
packetLength = ipHeaderTuple[2]         # Field 2: Packet Length
packetID     = ipHeaderTuple[3]         # Field 3: Identification
flagFrag     = ipHeaderTuple[4]         # Field 4: Flags/Fragment Offset
RES          = (flagFrag >> 15) & 0x01 # Reserved
DF           = (flagFrag >> 14) & 0x01 # Don't Fragment
MF           = (flagFrag >> 13) & 0x01 # More Fragments
timeToLive   = ipHeaderTuple[5]         # Field 5: Time to Live (TTL)
protocol     = ipHeaderTuple[6]         # Field 6: Protocol Number
checkSum     = ipHeaderTuple[7]         # Field 7: Header Checksum
sourceIP     = ipHeaderTuple[8]         # Field 8: Source IP
destIP       = ipHeaderTuple[9]         # Field 9: Destination IP

# Calculate / Convert extracted values

version      = verLen >> 4        # Upper Nibble is the version Number
length       = verLen & 0x0F      # Lower Nibble represents the size
ipHdrLength  = length * 4         # Calculate the header length in bytes

# covert the source and destination address to dotted notation strings

sourceAddress      = socket.inet_ntoa(sourceIP);
destinationAddress = socket.inet_ntoa(destIP);

if protocol == PROTOCOL_TCP:

    stripTCPHeader = packet[ipHdrLength:ipHdrLength+20]

    # unpack returns a tuple, for illustration I will extract
    # each individual values using the unpack() function

    tcpHeaderBuffer = unpack('!HHLLBBHHH' , stripTCPHeader)

    sourcePort             = tcpHeaderBuffer[0]
    destinationPort        = tcpHeaderBuffer[1]
    sequenceNumber         = tcpHeaderBuffer[2]
    acknowledgement        = tcpHeaderBuffer[3]
    dataOffsetandReserve   = tcpHeaderBuffer[4]
    tcpHeaderLength        = (dataOffsetandReserve >> 4) * 4
    flags                  = tcpHeaderBuffer[5]
    FIN                    = flags & 0x01
    SYN                    = (flags >> 1) & 0x01
    RST                    = (flags >> 2) & 0x01
    PSH                    = (flags >> 3) & 0x01
    ACK                    = (flags >> 4) & 0x01
    URG                    = (flags >> 5) & 0x01
    ECE                    = (flags >> 6) & 0x01
```

```
        CWR                     = (flags >> 7) & 0x01
        windowSize              = tcpHeaderBuffer[6]
        tcpChecksum             = tcpHeaderBuffer[7]
        urgentPointer           = tcpHeaderBuffer[8]

        if sourcePort < 1024:
            serverIP   = sourceAddress
            clientIP   = destinationAddress
            serverPort = sourcePort
        elif destinationPort < 1024:
            serverIP   = destinationAddress
            clientIP   = sourceAddress
            serverPort = destinationPort
        else:
            serverIP   = "Filter"
            clientIP   = "Filter"
            serverPort = "Filter"

        return([serverIP, clientIP, serverPort], [SYN, serverIP, TOS,
timeToLive, DF, windowSize])
    else:
        return(["Filter", "Filter", "Filter"], [NULL, Null, Null, Null])

# ------------ MAIN SCRIPT STARTS HERE -----------------

if __name__ == '__main__':

    # Note script must be run in superuser mode
    # i.e. sudo python ..

    # Enable Promiscious Mode on the NIC
    # Make a system call
    # Note: Linux Based

    ret =  os.system("ifconfig eth0 promisc")

    # If successful, then continue
    if ret == 0:

        print "eth0 configured in promiscous mode"

        # create a new socket using the python socket module
        # AF_INET     : Address Family Internet
        # SOCK_RAW    : A raw protocol at the network layer
        # IPPROTO_TCP : Specifies the socket transport layer is TCP

        # Attempt to open the socket
        try:
            mySocket = socket.socket(socket.AF_INET, socket.SOCK_RAW,
socket.IPPROTO_TCP)
```

```
    # if successful post the result
    print "Raw Socket Open"
except:
    # if socket  fails
    print "Raw Socket Open Failed"
    sys.exit()

# create a list to hold the results from the packet capture
# We wil only save Server IP, Client IP, Server Port
# for this example.  Note we will be making and educated guess as to
# differentiate Server vs. Client

ipObservations = []
osObservations = []

# Capture a maximum of 500 observations
maxObservations = 500

# Port filter set to port 443
# TCP Port 443 is defined as the http protocol over TLS/SSL

portValue = 443

try:

    while maxObservations > 0:

        # attempt receive (this call is synchronous, and will wait)
        recvBuffer, addr = mySocket.recvfrom(255)

        # decode the received packet
        # call the local packet extract function above

        content, fingerPrint = PacketExtractor(recvBuffer)

        if content[0] != "Filter":
            # append the results to our list
            # if it matches our port
            if content[2] == portValue:
                ipObservations.append(content)
                maxObservations = maxObservations - 1
                # if the SYN flag is set then
                # record the fingerprint data in osObservations
                if fingerPrint[0] == 1:
                    osObservations.append([fingerPrint[1], \
                                           fingerPrint[2],  \
                                           fingerPrint[3],  \
                                           fingerPrint[4],  \
                                           fingerPrint[5]])
```

```
                    else:
                        # Not our port
                        continue
                else:
                    # Not a valid packet
                    continue
        except:
            print "socket failure"
            exit()

        # Capture Complete
        # Disable Promiscous Mode
        # using Linux system call
        ret =  os.system("ifconfig eth0 -promisc")

        # Close the Raw Socket
        mySocket.close()

        # Create unique sorted list
        # Next we convert the list into a set to eliminate
        # any duplicate entries
        # then we convert the set back into a list for sorting

        uniqueSrc = set(map(tuple, ipObservations))
        finalList = list(uniqueSrc)
        finalList.sort()

        uniqueFingerprints = set(map(tuple, osObservations))
        finalFingerPrintList = list(uniqueFingerprints)
        finalFingerPrintList.sort()

        # Print out the unique combinations
        print "Unique Packets"
        for packet in finalList:
            print packet
        print "Unique Fingerprints"
        for osFinger in finalFingerPrintList:
            print osFinger
    else:
        print 'Promiscious Mode not Set'
```

Sample Program Output

```
eth0 configured in promiscuous mode

Raw Socket Open

Server              Client         Port
Unique Packets
('173.194.37.62', '192.168.0.13', 443)
('199.16.156.241', '192.168.0.13',443)
('199.16.156.52', '192.168.0.13', 443)
('23.235.39.223', '192.168.0.13', 443)
('23.253.135.79', '192.168.0.13', 443)
('23.78.213.231', '192.168.0.13', 443)
('54.192.160.200','192.168.0.13', 443)
('64.233.185.132','192.168.0.13', 443)
('64.233.185.95', '192.168.0.13', 443)
('66.153.250.212','192.168.0.13', 443)
('66.153.250.240','192.168.0.13', 443)
('66.153.250.241','192.168.0.13', 443)
('69.172.216.111','192.168.0.13', 443)
('74.125.137.132','192.168.0.13', 443)
('74.125.196.154','192.168.0.13', 443)
('74.125.196.99', '192.168.0.13', 443)
('93.184.216.146','192.168.0.13', 443)

Unique Fingerprints

Server              TOS  TTL   DF  Window Size
('23.235.39.223',    0,  53,    1,   14480)
('23.253.135.79',    0,  50,    1,   14480)
('64.233.185.132',   0,  42,    0,   42540)
('64.233.185.95',    0,  42,    0,   42540)
('66.153.250.240',   0,  57,    0,   28960)
('66.153.250.241',   0,  57,    0,   28960)
('69.172.216.111',   0,  47,    1,   14480)
('74.125.137.132',   0,  46,    0,   42540)
```

As you can see from this example, it is relatively straight forward to create a simple controlled traffic capture Python script and begin to map simple behaviors on the network. This capture then can process the captured data and identify specific hosts and services they support.

A couple special notes regarding this script.

1. This is a Linux only implementation

2. The Script needs to be run with super user privilege
   `$ sudo python capture443.py`
3. The advantage over using `tcpdump` or Wireshark relates to:
   a. Finer grained control over Super User activity
   b. The simplicity of the operation
   c. The ability to target specific results

OS FINGERPRINTING

I wanted to introduce the concept of OS Fingerprinting up front, since much discussion that surrounds Network Mapping attempts to identify the Operating System that is running behind a particular IP address. This process can be more difficult using passive methods, however it is still possible to make solid arguments for a particular OS. Our focus in the coming chapters is to craft scripts that will ensure that we capture and interpret traffic and fill out the IP range, observe and identify port / service activity and provide clear information regarding what insiders and outsiders are doing.

OS Fingerprinting Using TCP/IP Default Header Values

Several well-known attributes exist for gathering information about the OS executing behind each IP address that we are passively watching. They include:

Table 2-2 Common OS Fingerprinting Fields

IP Header	Defined
TTL	Time to Live
TOS	Type of Service
DF	Don't Fragment Flag
TCP Header	**Defined**
Window	Window Size

Note, these values are only valuable when the SYN flag is set for a specific TCP packet. You will notice in the capture443.py script, I painstakingly extracted the TTL, TOS, DF from the IP Header and I extract Window Size from the TCP Header. I also create a unique list of the observed fingerprinting values. This script then can be used to record these notable header fields in order to build a more comprehensive "observed" OS fingerprints.

Based on observations from a plethora of sources, Table 2-3 provides a snapshot of observed values that can provide insight to enable fingerprinting an OS. This fingerprinting process is virtually the same for passive vs active mapping - the

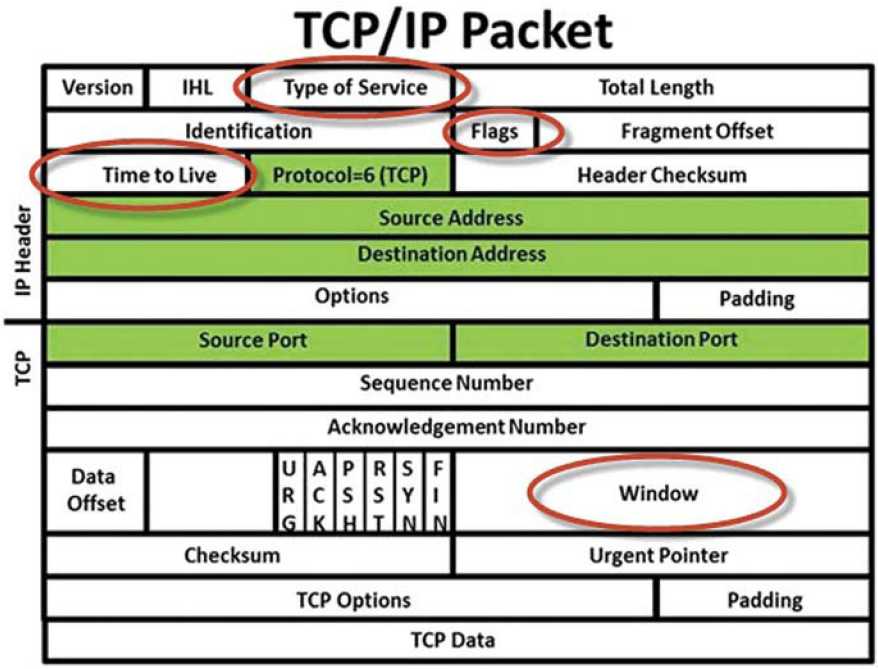

FIGURE 2-5 TCP/IP Header with Key Fingerprinting Fields Highlighted.

only difference being when observing passively, the stimulus must come from normal network traffic, not from artificially generated stimuli.

An educated guess of the OS behind the IP address is possible by creating a comprehensive list of the most common devices. It is important to point out that masking these TCP/IP header fields can be accomplished by those trying to obscure these signatures. Thus it is important to utilize multiple methods:

Table 2-3 Sampling of OS Observed Values

	Time to Live	Window Size
Observed OS	***Initial Value***	***Typical Setting***
Linux	64	5840
Open BSD	64	16,384
Solaris	255	8,760
AIX	64	16,384
Windows XP	128	65,535
Windows 2K	128	16,384
Windows 7	128	8,192
Mac OS X	64	65,535

Table 2-4 Sampling of Open Port Patterns

Port Number	Most Common Usage	OS Fingerprint Guess
445	Microsoft Active Directory	Windows
987	Microsoft Sharepoint Service	Windows
1270	Microsoft System Center Operations Manager (SCOM)	Windows
331	Apple OS Server Admin	Mac OS X
660	Mac OS Server Admin	Mac OS X
11111	Remote Configuration Interface	RedHat Linux

OS Fingerprinting Using Open Port Patterns

Another common method is to take an inventory of open port patterns. This is especially useful when collecting passive network behaviors of hosts operating within the monitored environment. Table 2-4 lists just a few of the common ports that can provide clues to the operating system running behind the IP.

We will explore OS Fingerprinting analysis using deductive and inductive reasoning in Chapter 4.

WHAT OPEN PORTS OR SERVICES DON'T YOU KNOW ABOUT?

As was recently seen with the OpenSSL 'Heartbleed' (CVE-2014-0160) and Shellshock (CVE-2014-6271) vulnerabilities, the ability to know what services are operating and on what systems is quite useful. Once again we could use tools like NMAP to discover open ports (at least during the snapshot) with the previously discussed risks. Standard network ports are assigned by the Internet Assigned Numbers Authority (IANA) via the Service Name and Transport Protocol Port Number Registry. Generally (as there is debate) an agreed upon port classification is as follows:

> *Service Ports: 1-1023* are considered well-known ports that represent services that most of us agree to abide by.
> *Service Ports: 1024 to 49151* are recognized as registered ports. They are assigned by IANA upon application and approval.
> *Service Ports: 49152–65535* are considered Dynamic, Private or Ephemeral (i.e. lasting for a short time or transient). For example, ports in this range are commonly used by clients making a connection to a server.

One way to leverage this knowledge of course is to detect traffic originating from, or going to one of these defined ports. By doing so we can

deduce services that are running on these hosts and clients that are utilizing them.

In addition to the "agreed upon" port definitions above, organizations such as the SANS Internet Storm Center have created lists of known malicious ports. For example, one compiled list contains default ports utilized by Trojans. Therefore, if you find that one these ports is being probed, it may possibly indicate that someone is attempting to communicate with a Trojan that is running on your network. Thus mapping both the request, and potentially the response to one or more of these ports would be useful in mapping as well.

How is This Useful?

Based on the simple capture443.py script I presented earlier in this chapter, along with the results shown, we could deduce the following:

Local Client 192.168.0.13 has made a secure web page connection to the following servers:

> 199.16.156.201, 23.73.162.234, 66.153.250.229, 66.153.250.234, 66.153.250.238, 66.153.250.241, 74.125.137.132, 74.125.137.154, 74.125.196.99, 74.125.230.127

This deduction was made based on the following facts:

1. IP address 192.168.0.13 is a Class C private address block. According to RFC 1918, any Class C address in the range 192.168.0.0-192.168.255.255 (which can also be denoted 192.168.0.0/16) should be considered private and non-routable. This means that I cannot directly address any Class C address within that range unless I'm connected to that very same Class C physical network.
2. Each of the other IP addresses can be geographically located. For example, addresses 199.16.156.201 is located in the Mountain View, California area. The IP addresses 66.153.25 are located in South Carolina. Each of these IP addresses communicated with the client over service port 443, which by default is the http protocol running over a secure TLS or SSL connection.

In addition, I could *infer* that client 192.168.0.13 performed a web search that provided a link to the other servers identified. I can make this inference because IP addresses 74.125.137.x belongs to Google, and it is likely that client 192.168.0.13 performed the suggested search using Google.

DEDUCTIVE VS INDUCTIVE REASONING

Deductive reasoning is based on the premise that if the predicates are true, and the logic is sound the conclusion must be valid.

The classic example is

"All men are mortal"
"Socrates was a man"
Therefore: Socrates was mortal

Inductive reasoning, on the other hand, seeks a probable or a likely explanation. A classic example of an inductive argument is:

"All politicians I have met are deceitful"
"I have just met David and he is a politician"
Therefore: David must be deceitful

Much like the inductive argument that was made:

"IP 192.168.0.13 connected to Google"
"Google is the search engine that provides links to other web sites"
Therefore: the subsequent server IP addresses must have come from Google

In both of these cases the likelihood is probable, however unlike the deductive arguments other possible conclusions exist.

In order to perform Passive Network Mapping we will be using both deductive and inductive methods throughout the process. The quality of our arguments, premises, observations and logic will determine how accurate our results will be. Based on that, it will be important to craft these arguments and observations such that they can be improved with time.

Note: Active Network Mapping also uses both methods especially during the process of OS Fingerprinting.

WHO'S TOUCHING YOUR NETWORK?

The next logical question to ask is who is actually touching your network? This includes trusted insiders, employees, IT staff (either in-house or out-sourced), and those outside your direct sphere of control. This doesn't mean just hackers, but can also mean business partners, contract employees, vendors, Internet Service Providers (ISPs), the government, and, of course, your customers. By passively collecting, classifying, analyzing and reasoning about the network activity and open ports, we can glean a tremendous amount of information including:

1. What IP addresses are insiders connecting to?
2. Where are the insiders and outsider located geographically?

3. How often and at what time of day are these services being used? Is this activity normal or abnormal?
4. What IP addresses are outsiders connecting to?
5. Where are these outsiders located geographically?

As you may quickly realize, these questions are more difficult or even in some cases impossible to answer when using active scanning methods, and force direct interaction in response stimulation. In Chapter 4 we will provide scripts that can collect and analyze targeted information that can assist in answering at least some of these questions and provide the foundation for further expanded development.

REVIEW

In Chapter 2, I examined the breadth of devices that may be running on your network that are worth considering. I also discussed their associated risks. I then setup a network capture using Linux and `tcpdump` to capture network packets using promiscuous mode. By manually examining the results I extracted the unique results shown. Next, I developed a Python script that would perform the same type of promiscuous capture, but focused on targeting network activity associated with port 443, which is typically associated with the http protocol over TLS/SSL.

The script also makes an educated guess and converted the typical source IP and destination IP into the more meaningful server vs client characterization. This allowed me to automatically generate the unique list of client server interactions occurring on port 443. Next, I examined the TCP/UDP Port mapping and defined the ranges of well known, registered and ephemeral ports. I then introduced the subtle differences between deductive and inductive reasoning that will be used in future chapters and scripts. Next, I introduced a couple of OS Fingerprinting methods that will be used in Chapter 4. And finally, we examined the additional benefits of Python Passive Network Mapping as applied to behavior of trusted insiders and outsiders.

SUMMARY QUESTIONS

1. What additional network devices will be important to map and identify on our networks and why?
2. How would you generalize the capture443.py script to allow for other targeted captures?
3. Expand the capture443.py script to implement these generalizations.
4. How might you expand capture443.py to create a comprehensive list of unique observed combination of TOS, TTL, DF and Window Size? Then implement the standalone solution.

5. What other OS Fingerprinting methods would be applicable to passive mapping activities.
6. What passive network mapping operations would be best suited for deductive reasoning?
7. What passive network mapping operations would be best suited for inductive reasoning?

Additional Resources

SANS Intrusion Detection FAQ: http://www.sans.org/security-resources/idfaq/oddports.php

IANA – The Internet Assigned Numbers Authority: http://www.iana.org/

TCPDUMP and LIBPCAP: http://www.tcpdump.org/

Introduction to LOGIC, Seventh Edition 1986, by Irving M. Copi ISBN: 0-02-325020-8 McMillan Publishing Company New York New York.

Might I also recommend a good TCP/IP text or two, e.g., Chappell & Tittell, *Guide to TCP/IP* or Stevens, *TCP/IP Illustrated, Vol. 1*. I would also offer my own "An Overview of TCP/IP Protocols and the Internet" at http://www.garykessler.net/library/tcpip.html