Port Scanning with Python and Nmap

Sumner Hearth
Cornell Hacking Club

3/1/2016

Introduction

The code and content of the instruction manual can be found online on the **Piazza page**. Code will be written in python, packages will be installed by default unless explicitly stated otherwise.

The code and content of this manual are for educational purposes only, Sumner Hearth and the Cornell Cyber Security Club / Cornell Hacking Club in no way endorse or promote illegal actions or activities.

1 Networks

1.1 Servers

Modern computers are fast, modular, and able to perform an incredibly wide range of tasks. But as user imagination grew and demand for new services soared computer networks quickly became the only way to serve requests. To quickly summarize `CS 4410`, computer operating systems manage network communication behind the scenes (for the most part). The computer has it's own MAC address associated with each network interface (i.e. Ethernet, Wifi, ...) and the network leases an IP to that interface for computers to be able to find other computers on the same network [1]. The OS then will allow various processes and programs to access data destined to "ports". These ports allow data packets to define which process (at a particular IP) should receive the information. The code is this section will briefly summarize how this is done. Python's `socket` package provides most of the resources you'll need to listen to, and send data to other computers.

```
import socket

# Tells the OS: "My address"
listen_addr = "0.0.0.0"
# Any number you want, though some need
# root priviledge or are already in use
listen_port = 1708

try:
    # Make a socket using some default parameters
    s = socket.socket(socket.AF_INET,
                      socket.SOCK_STREAM)
    # Reuse ports
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    # Listen for an address at a port
    s.bind((listen_addr,listen_port))
    # Tell the OS to hold up to 5 connections
    s.listen(5)
except Exception as e:
    print("Failed to create socket: "+e.message)

s.close()
```

Excellent, you now have a basic socket listening to a port. This means other computers can now send data to you. Python makes reading this data easy. For example, let's say we wanted to listen to the socket until somebody sent a newline, then repeat the data back to them. This could be described as an echo server of some description:

[1] We will pretend we are working on only a local network for the time being

```
import socket

# Tells the OS: "My address"
listen_addr = "0.0.0.0"
# Any number you want, though some need
# root priviledge or are already in use
listen_port = 1708

try:
    # Make a socket using some default parameters
    s = socket.socket(socket.AF_INET,
                      socket.SOCK_STREAM)
    # Reuse ports
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    # Listen for an address at a port
    s.bind((listen_addr,listen_port))
    # Tell the OS to hold up to 5 connections
    s.listen(5)
except Exception as e:
    print("Failed to create socket: "+e.message)

print("Waiting for connections...")
while True:
    conn, address = s.accept() # Blocks...

    # Block released
    print("Connected to: "+address[0])
    # Read data until newline
    data = ""
    while True:
        data = data + conn.recv(1024)
        if data.endswith(u"\r\n"):
            break
    conn.send("You said: "+data)
    conn.close()
    print("Closed connection")
```

Python supplies a `s.accept()` method on the sockets which stops code executing at that line until the operating system informs it that another computer is attempting to connect. This returns a connection `conn` and an address from which the connection originates `addr`. The address is an IP and port (much like what we supplied to listen in the first place). Running this code[2] should supply you with the following output:

[2]If you need help running the code please refer to online Unix/Linux tutorials. If you have a syntactic error check your indentation and post on Piazza for help

```
>> python echoserver.py
Waiting for connections...
```

At which point the terminal session will hang (pause) and wait for more input. This is expected as the line of code `s.accept()` has not yet received a connection. We should now test the port. We can do this using Unix's builtin `telnet` command. Open a new terminal window/tab and type `telnet localhost`[3] `1708`[4] . You should immediately see the session spring to life:

```
>> telnet localhost 1708
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```

And on the other terminal window (running python) you should see a connection has started:

```
>> python echoserver.py
Waiting for connections...
Connected to: 127.0.0.1
```

We have succesfully contacted the server running in python. Now python is accepting input and waiting for a newline to response, try typing into the `telnet` window and hitting *return* to see if the code works as expected.

```
>> telnet localhost 1708
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
HELLO PYTHON
You said: HELLO PYTHON
Connection closed by foreign host.
```

And on the python side:

```
>> python echoserver.py
Waiting for connections...
Connected to: 127.0.0.1
Closed connection
```

Perfect, python is looping, waiting for another connection after having successfully performed the operations we asked of it. See if you can figure out how to change any of the above to connect between two machines or perform other actions when a connection is made[5]

[3] This tells the computer that the server is on the same machine
[4] If you chose a different port then enter it here
[5] Not localhost anymore

1.2 Clients

For the last piece, we used `telnet` to connect to the server for us. But in the field we will not want to manually connect and interact with each port on each machine, we will want to automate the process. Luckily python's sockets aren't just for listening passively, they can actively connect to targets. The process is similar to the one used to make a server, but instead of binding and listening we use socket's `s.connect((addr, port))` feature.

```
import socket

# Server is on this machine
conn_addr = "localhost"
# We know what port we're listening to
conn_port = 1708

try:
    # Make a normal socket just like before
    connSkt = socket.socket(socket.AF_INET,
                            socket.SOCK_STREAM)
    # Tell it where to send data
    connSkt.connect((conn_addr, conn_port))
    print("Connected to {0}:{1}".format(
            conn_addr,str(conn_port))
          )
    # Do nothing for now...
    connSkt.close()
except Exception as e:
    print("Failed to connect: "+e.message)
```

This code quite simply connects to the server, sends nothing, and closes the connection. If you run it you'll find that it reports a successful connection and exits. If we look over to the python server side we'll see something odd:

```
>> python echoserver.py
Waiting for connections...
Connected to: 127.0.0.1
```

A connection is started, but it never reports `Closed connection`. Which means the server still believes the connection is open. If we run the python client again we will see that the server does not respond. Even running the `telnet localhost 1708` command does nothing (no echo). Congratulations, you've found your first DOS exploit[6]. We can easily fix this on the server side with a little knowledge of how python sockets work. Quite simply put, if the connection is dead then the `conn.recv(1024)` receives no data. So we just need to check for that in the server loop of our python server:

[6]DOS stands for Denial of Service, you've stopped a server from being able to serve clients

```
data = ""
while True:
    new_data = conn.recv(1024)
    data = data + new_data
    if data.endswith(u"\r\n") or new_data=="":
        break
```

Now back to the client. Let's try to get some information from our server using our socket's `send` and `recv` features. Replace the comment about doing nothing with the following:

```
# Send some data
connSkt.send("Hello?\r\n")
# Receive up to 1024 bytes of data
data = connSkt.recv(1024)
print("Received: "+data)
```

There, we send a message and recv some data[7]. When we run this, as expected, the python client spits out the following:

```
>> python basicclient_data.py
Connected to localhost:1708
Received: You said: Hello?
```

Now, what would happen if we set our sights on other ports? Let's try running this against some random port.... say port 22? Simply change the appropriate variable and we find:

```
python basicclient_banner.py
Connected to localhost:22
Received: SSH-2.0-OpenSSH_6.9
```

Well, we know our server isn't saying that. Must be someone else's.[8]

[7] We keep using this 1024 number, it's necessary for most socket interactions, ever wonder what would happen in we sent *more* than 1024 bytes?

[8] This is dependent on your system, on windows chances are you'll simply see "Failed to connect". This is because Windows doesn't run an ssh server by default, most Linux/Unix (including OSX) do by default, however. If you want to you can scan your Kali Linux instance to follow along from this point forward.

2 Python Port Scanning

So, there are some other services running on other ports? You may not know how to exploit them yet, but it would still be a good idea to know about them. Let's try to find them all (within reason). We will use some command line interfacing to allow us to define which ports to scan and report everything we see. You can get command line arguments using the python `sys` library. We will also introduce the socket `settimeout` method to make sure our port scanner doesn't get stuck on a server that refuses to response.

```
import socket
import sys

def scan_range(addr, start, stop):
    results = []
    # Iterate through ports
    for conn_port in xrange(start,stop):
        try:
            # Make a normal socket just like before
            connSkt = socket.socket(socket.AF_INET,
                                    socket.SOCK_STREAM)
            # Reuse ports
            connSkt.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
            # Don't wait too long...
            connSkt.settimeout(1)
            # Tell it where to send data
            connSkt.connect((addr, conn_port))

            # It's at least open
            print("[*] Open {0}".format(conn_port))
            # Send some data
            connSkt.send("Hello?\r\n")
            # Receive up to 1024 bytes of data
            data = connSkt.recv(1024)
            if data!="": print("\t"+data)
            results.append((conn_port, data))
            connSkt.close()
        except Exception as e:
            pass
    return results

# Get arguments from command line
args = sys.argv
# If first one is filename, remove
if (sys.argv[0]==__file__):
    args = sys.argv[1:]
```

```
if len(args)==3:
    addr = args[0]
    start_port = int(args[1])
    end_port = int(args[2])
    print("Scanning {0} from {1} to {2}".format(
        addr,start_port,end_port))
    results = scan_range(addr,start_port,end_port)
    # Do something with results?
else:
    print("Usage: python port_scan.py addr "+
          "start end")
```

The code should be straightforward, iterate through ports, connections tell us *something* is listening, if you can get any data back then print that as well before moving on. For my system I got the following output:

```
>> python port_scan.py localhost 1 2000
Scanning localhost from 22 to 2000
[*] Open 22
        SSH-2.0-OpenSSH_6.9
Protocol mismatch.

[*] Open 88
[*] Open 548
[*] Open 1708
        You said: Hello?
```

In conclusion, you can imagine all sorts of ways to improve this program. For example try threading each client so they don't have to wait for one another to finish. Once you're done improving it, we can introduce you to the tool that improved it all already.

3 Nmap

Nmap is one of the most powerful tools at your disposal. It features an immense number of options plus analytic and forensic tools to determine information about any system. First let's see how we can do a scan like the one we implemented in python using Nmap, but for a more detailed scan I recommend the **nmap website** and **this basic tutorial**. The `nmap` tool has a scan type called a `TCP-connect` scan which effectively does what we implemented in python. It attempts to connect to a port and reports success or failure along with some additional information:

```
>> nmap -sT localhost -p 1-2000

Starting Nmap 7.01 ( https://nmap.org ) at 2016-02-28 14:13 EST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00043s latency).
Other addresses for localhost (not scanned): ::1
Not shown: 1997 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh
548/tcp  open  afp
1708/tcp open  gat-lmd

Nmap done: 1 IP address (1 host up) scanned in 19.35 seconds
```

Interestingly nmap missed port 88 being up[9], it also crashed our echo server (see other terminal window).

```
>> python echoserver_fix.py
Waiting for connections...
Connected to: 127.0.0.1
Traceback (most recent call last):
  File "echoserver_fix.py", line 31, in <module>
    new_data = conn.recv(1024)
socket.error: [Errno 54] Connection reset by peer
>>
```

This crash would be an immediate alarm to anyone on the computer that something was up and hackers were scanning the computer, we need something more powerful, something stealthier. Let's look at the `SYN` scan. This is similar to the `TCP` scan but does something we couldn't do in python, it stops halfway through. The `s.connect...` call from before actually sent a little information to the other computer, effectively asking "is this port open?" to which the other computer responded "yes, would you like to connect?" which we in turn answered "yes!" (paraphrasing). The SYN scan stops halfway through, only listening for the initial response. The other computer assumes the network went

[9] I'm looking into this, port 88 seems to be associated with Kerberos Authentication, no reason it shouldn't respond

down or the client lost interest and carries on with its life. We can perform such a scan as follows:

```
>> sudo nmap -sS localhost -p 1-2000
```

Sudo here tells the computer that you want to run this program with more privileges, letting it do things it wouldn't otherwise have the authority to do.

```
>> sudo nmap -sS localhost -p 1-2000
Password:

Starting Nmap 7.01 ( https://nmap.org ) at 2016-02-28 14:13 EST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00043s latency).
Other addresses for localhost (not scanned): ::1
Not shown: 1997 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh
548/tcp  open  afp
1708/tcp open  gat-lmd

Nmap done: 1 IP address (1 host up) scanned in 19.35 seconds
```

Same results as before, let's look at the server:

```
>> python echoserver_fix.py
Waiting for connections...
```

It's not even reporting anyone *tried* to connect.

3.1 Features

Nmap has several interesting features for you to explore, one such example is OS scanning using the `-O` flag:

```
>> sudo nmap -sS -O localhost
Starting Nmap 7.01 ( https://nmap.org ) at 2016-02-28 14:27 EST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00018s latency).
Other addresses for localhost (not scanned): ::1
Not shown: 995 closed ports
PORT     STATE SERVICE
...
Device type: general purpose
Running: Apple Mac OS X 10.10.X|10.11.X
OS CPE: cpe:/o:apple:mac_os_x:10.10 cpe:/o:apple:mac_os_x:10.11
OS details: Apple Mac OS X 10.10 (Yosemite) - 10.11 (El Capitan) \
        (Darwin 14.0.0 - 15.0.0)
...
```

This is useful when choosing how you want to approach exploiting a system, and figuring out what services you may want to look for. There are also a wide variety of stealth features such as zombie scans, decoys, and disabling ping. I encourage you to explore the options on their website and share anything interesting you find with the group.

Happy Hunting!