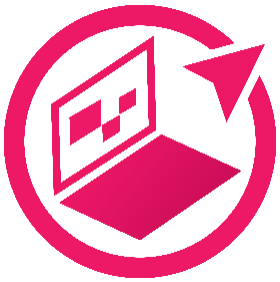

Python Tools for Network Tasks

Perp: Khalid Ebn Elwleed Mohamed Ahmed

November 2025

Contents

Introduction

Talking about network it's will always be the attractive arena for the hackers. You can exploits the network access to do almost anything, such as **scan for hosts, inject packets, sniff data,** and **remotely exploit host.**

Sometime when you conducting an attack for specific target you may find yourself does not have tools to conducting network attack, such as **netcat** or way to install one. However, in this situation you can use Python languages to build the program that help you.

Python languages is one of the most importing programming languages on the field of the Cyber security. In this article we will build **client, servers, and a TCP proxy.** And then will turned them to be our own netcat.

Usually any programmer had different module to create networked servers and clients in Python. Here we'll use the **socket** is one of the most important libraries for network programming. It allows you to create and use network connections between computers using TCP/IP protocols — the foundation of the internet. Full documentation can found here: socket — Low-level networking interface — Python 3.14.0 documentation

TCP client

TCP client is a program that initiates a connection to a TCP server so they can exchange data over the network. Let's create TCP client to see how it work.

```python
import socket

target_host = input("Enter target host (IP or domain): ")
target_port = int(input("Enter target port (1-65535): "))

# Create a TCP socket
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

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

# Send some data
client.send(b"GET / HTTP/1.1\r\nHost: " + target_host.encode() + b"\r\n\r\n")

# Receive some data
response = client.recv(4096)
print(response.decode())
client.close()
```

- Frist, we create a socket object with the `AF_INET` and `SOCK_STREAM` parameters. (`AF_INET` parameter indicates we'll use a standard IPv4 address or hostname, and `SOCK_STREAM` indicates that this will be a TCP client.)
- Second, then we connected the client to server.
- After that we send it some data as bytes.
- Finally, this stage to receive some data back and print out the response the close the socket.

UDP client

UDP is a core internet communication protocol for sending messages between hosts on an IP network that prioritizes speed and efficiency over reliability. A python UDP client it's like the TCP client with slight different. Let's break it down.

```
import socket


target_host = input("Enter target host (IP or domain): ")
target_port = int(input("Enter target port (1-65535): "))

# Create a UDP socket
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Send some data
client.sendto(b"Hello, UDP Server!", (target_host, target_port))

# Receive some data
data, add = client.recvfrom(4096)

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

- As you see, we changed the `socket` type from `SOCK_STREAM` to `SOCET_DEGRAM`.
- The next step is to simply call `sendto()2`, passing in the data and the server you want to send the data to. Because UDP is a connectionless protocol, there is no call to `connect()`

- The last step is to call `recvfrom()3` to receive UDP data back. You will also notice that it returns both the data and the details of the remote host and port.

You don't need to be a networking wizard to get useful results. We'll keep things quick, easy, and reliable — exactly what you need for regular pen testing work.

TCP server

TCP server is a minimal, single-file TCP server that accepts connections and allows the operator to read an incoming message and reply interactively. You might want to use you own TCP server when writing command shells or crafting a proxy.

The feature of this python program is listens on a user-provided IP and port, accepts multiple clients by using python threads library, and prints received client data and prompts the operator to type a response to send back. Let's check out the following code:

```
import socket
import threading


IP = input("Enter IP address to bind the server (e.g.,): ")
Port = input("Enter Port number to bind the server (e.g., 8080): ")

def main():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind((IP, int(Port)))
    server.listen(5)
    print(f"[*] Listening on {IP}:{Port}")


    while True:
        client, addr = server.accept()
        print(f"[*] Accepted connection from {addr[0]}:{addr[1]}")
        client_handler = threading.Thread(target=handle_client,
args=(client,))
        client_handler.start()


def handle_client(client_socket):
    with client_socket:
        request = client_socket.recv(1024)
        print(f"[*] Received: {request.decode('utf-8')}")
```

```
        repsonse = input("Enter a msg to send to the client: ")
        client_socket.send(repsonse.encode('utf-8'))

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

- Frist, we pass in the IP and port we want the server to listen on.
- We command the server to start listening, with maximum backlog of connection set to 5.
- Then we create a loop for waiting incoming connection when a client connects.
- Next, we receive the client socket in the `client` variable and the remote connection details in the details in the `addr` variable.
- We create a new thread that targets the `handle_client` function and pass the client socket to it as an argument. After that, we start the thread so it can manage the client connection independently.
- At the end, the main server loop is free to accept another incoming connection. In the meantime, the `handle_client function performs a recv()` operation and then sends a simple response back to the client.

You can the TCP client that we create it, and send some text packet to the server.

Conclusion

Networking remains one of the most powerful and attractive fields in cyber security. Understanding how data moves between systems gives you the ability to analyze traffic, interact with remote hosts, and even create your own tools during penetration-testing engagements.

In this article, we explored how Python can help you build essential networking components when traditional tools are unavailable. Using the socket library, we created a TCP client, a UDP client, and a TCP server capable of handling multiple connections. With the addition of Python's threading library, the server can manage several clients at the same time, making it more flexible and practical for real-world scenarios.

These examples show how Python allows you to craft custom solutions quickly—from sending and receiving data to maintaining active sessions with different clients. This makes Python a valuable tool for penetration testers, researchers, and anyone who needs full control over network communication.

By understanding these core building blocks, you gain a deeper appreciation of how network protocols work behind the scenes and prepare yourself to develop more advanced tools in the future.