Python - Sockets Library

Notebook

Introduction to Sockets

Before jumping into the programming part and why we would like to gain this knowledge let's understand what a socket is, where it's born and what it is initial purpose.

What are sockets?

Sockets were born and released in 1983 and were known as “Berkely Sockets”. Sockets allow communication between two different processes on the same or different machines. In Unix, every I/O action is done by writing or reading a file descriptor. A file descriptor is just an integer associated with an open file and it can be a network connection, a text file, a terminal, or something else.

To a programmer, a socket looks and behaves much like a low-level file descriptor. This is because commands such as **read()** and **write()** work with sockets in the same way they do with files and pipes.

Processes are presumed to communicate only between sockets of the same type but there is no restriction that prevents communication between sockets of different types.

Stream Sockets − Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order − "A, B, C". These sockets use TCP (Transmission Control Protocol) for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.

Datagram Sockets − Delivery in a networked environment is not guaranteed. They're connectionless because you don't need to have an open connection as in Stream Sockets − you build a packet with the destination information and send it out. They use UDP (User Datagram Protocol).

Even that sockets are looks old they are still used in common operation systems; this is an example of sockets services in Kali Linux:

```
(root💀kali)-[~] find / -type f -name "*.soc*"
/usr/lib/systemd/system/saned.socket
/usr/lib/systemd/system/rpcbind.socket
/usr/lib/systemd/system/systemd-journald.socket
/usr/lib/systemd/system/systemd-fsckd.socket
/usr/lib/systemd/system/systemd-initctl.socket
/usr/lib/systemd/system/systemd-journald@.socket
/usr/lib/systemd/system/systemd-networkd.socket
/usr/lib/systemd/system/systemd-journald-dev-log.socket
/usr/lib/systemd/system/syslog.socket
/usr/lib/systemd/system/dbus.socket
[ .. snippet ..]
```

Unix sockets are usually used as an alternative to network-based TCP connections when processes are running on the same machine. Data is usually still sent over the same protocols. it just stays within the same machine and knows it’s running in the same domain (hence, the name UNIX domain sockets), so it never has to bother a loopback network interface to connect to itself.

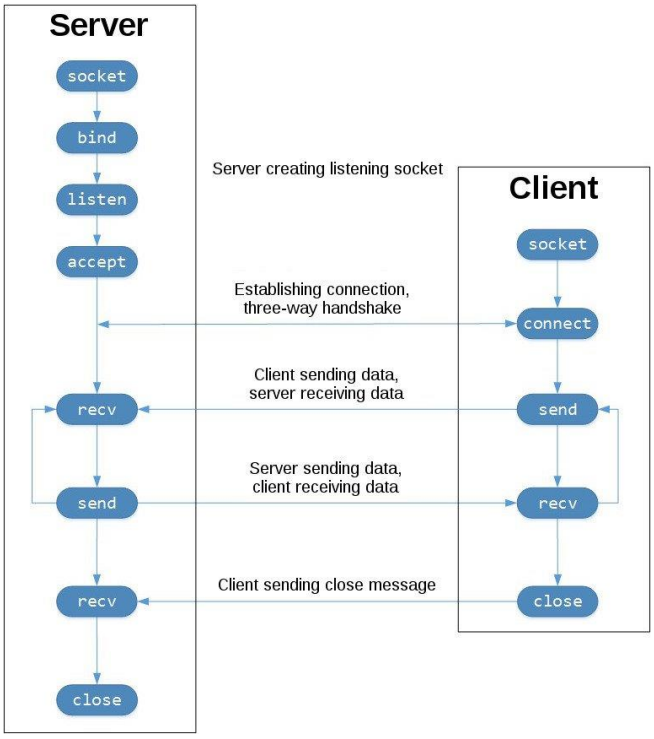
The biggest example of this is **Redis**, an extremely fast key-value store that operates entirely within memory. Redis is frequently used on the same server that's accessing it, so you'll usually be able to use sockets. At such low levels and with how fast Redis is, sockets provide a **25%-performance boost** in some synthetic benchmarks.

Socket Programming

Python socket module provides interface to the Berkeley sockets API. Python provides a convenient and consistent API that maps directly to these system calls, their C counterparts. The primary socket API functions and methods in this module are:

- socket()
- bind()
- listen()
- accept()
- connect()
- connect_ex()
- send()
- recv()
- close()

As part of its standard library, Python also has classes that make using these low-level socket functions easier.



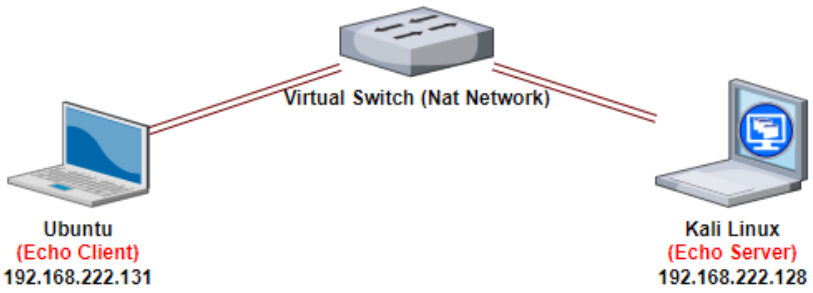
Let's Build an Echo Client and Server

Let's get our hands dirty. This chapter going to talk about how to create and program the "Echo" server and client using Python3. "Echo" means the mission of the server and the client is exchange basic TCP message. **Remember this** - we going to provide an IPv4 communication tunnel based on TCP as transport protocol.

There is no real need to create two virtual machines or use two computers to allow this communication happened as mentioned before - for our example it works great. Don't forget that sockets as mentioned above used more often for inter-communication which means myself to myself communication.

Both Virtual machines that I am using have pre-installed python3 and the sockets library (its built-in).

Lab topology:



The **echo-server.py** code (basic variant):

```
#!/usr/bin/env python3

import socket
HOST = '127.0.0.1'
PORT = 65432
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen()
conn, addr = s.accept()
print('Connected by', addr)
data = conn.recv(1024)
conn.sendall(data)
s.close()
```

The first part:

```
import socket
HOST = '192.168.222.128'
PORT = 65432
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Firstly, we import the socket module to use its functions. Create a socket object is done when the **socket()** function is called, in the code above the object is assigned to **s** variable. The **socket()** function gets two parameters the **socket.AF_INET** and the **socket.SOCK_STREAM** - these parameters define the communication tunnel, in this case the **AF_INET** is using IPv4 and the **SOCK_STREAM** is using TCP as transport layer protocol.

```
s.bind((HOST, PORT))
s.listen()
conn, addr = s.accept()
print('Connected by', addr)
```

After the socket object is created, as a server we need to **bind** it. Every network service is bind to some port - for example **SSH** bind to 22 and **FTP** bind to 21. The **bind()** function gets a tuple parameter of a **string** of the IP address and an **integer** of a port. A tuple is just a way to make sure its pre-defined values and they couldn't be changed during the process.

The **listen()** function has a backlog parameter - this parameter specifies the number of unaccepted connections that the system will allow before refusing new ones, in simple words, it will get a several possible clients.

accept() blocks and waits for an incoming connection. When a client connects, it returns a new socket object representing the connection (the **conn** variable) and a tuple holding the address of the client (the **addr** variable). The tuple will contain (host, port) for IPv4 connections.

```
data = conn.recv(1024)
conn.sendall(data)
s.close()
```

The **recv()** functions listening to an incoming data and the parameter of a number specifies how many bytes it would like to receive (the TCP buffer space). After the server receive the data, it will send it back with the **sendall()** function.

After communication is finished we close the socket using the **close()** function.

The very important thing to understand when we programming a server is that it has two sockets to work with - the first socket assigned with **s** variable that configured to listen and accept incoming connections and the other socket assigned with **conn** variable, this socket is our communication socket, it's the client socket that we use to receive and send data within this tunnel.

Let's make it more efficient and correct

I got some better idea of how we could handle socket objects - and its even important. There is a problematic point with sockets, sockets are I/O devices, like files for example, even that for today the python versions handle it by themselves it's very important to not forget to close the I/O devices.

Open I/O devices could provide a good base for some privilege escalation paths of an intruder - its will be available only if the script is running right now!

We could take it out of our responsibilities when using the **with** statement, this statement could be assigned with I/O devices and handle their expectations and even close them in the end of usage.

```
#!/usr/bin/env python3

import socket
HOST = '192.168.222.128'
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

The **echo-client.py** code:

```
#!/usr/bin/env python3

import socket
HOST = '192.168.222.128'  # Kali's IP address
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
```

```
print('Received', data.decode('ascii'))
```

In comparison to the server, the client is simple. It creates a **socket object**, connects to the server, and calls **s.sendall()** to send its message. Lastly, it calls **s.recv()** to read the server's reply and then prints it.

Projects Ideas

When it comes to cyber security there are bunch of possible tools that could be created using the **socket** library. For example:

- TCP port scanner with provided port list or provided targets
- Honeypot server
- FTP server user and password bruteforcer

TIP:
Start with the basic idea and functionality and afterwards you could append more efficiency and performance with deeper understanding of available functions of **socket** module and **threads** module functionality.