Threaded Port Scanner in Python

Introduction

Port Scanners are primarily used for Penetration Testing and Information Gathering. Essentially, we are looking for open ports in a host for one of two reasons. To ensure our servers are secure or to exploit those of someone else. An unnecessarily opened port means vulnerability and comes with a lack of security.

Therefore, it is reasonable to scan the ports of your own network in order to spot potential security gaps. To do so, we can use a popular and professional open-source software like Nmap. In this tutorial however, we will code our own port scanner in Python. That has the advantage that we can customize it as we want and that we learn something about programming. We choose Python as a language because it is simple, modern and very powerful.

Before we start, you need to know that port scanning someone else's network without their permission is a crime. Only scan your own network or one that you are permitted to. Alternatively, you might use scanme.nmap.org to experiment around. I do not take any responsibility for the abuse of the information.

Basic Functionality

Let us first take a look at the basic functionality of a port scanner. A port scanner tries to connect to an IP-Address on a certain port. Usually, when we surf the web, we connect to servers via port 80 (HTTP) or port 443 (HTTPS). But there are also a lot of other crucial ports like 21 (FTP), 22 (SSH), 25 (SMTP) and many more. In fact, there are more than 130,000 ports of which 1,023 are standardized and 48,128 reserved. So, when we build a port scanner, we better make it efficient and focus on the crucial ports.

Now, we will first look at the simplest way to scan ports with Python.

```
import socket

def portscan(port):
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.connect(("127.0.0.1", port))
        return True
    except:
        return False


print(portscan(22))
```

In this example, we create an ordinary socket that tries to connect to a target on a specific port. If it succeeds, we return True. If there is an exception, we return False. Then we print the result for the respective port.

Of course, we can also automate that by making a for-loop that scans all the standardized ports.

```
for port in range(1, 1024):
    result = portscan(port)
```

```
    if(result):
        print("Port {} is open!".format(port))
    else:
        print("Port {} is closed!".format(port))
```

Actually, this is a complete port scanner already. But there is a problem with it. When you run it, you will notice that it is extremely slow, because it scans one port after another. That can be changed with Multithreading. So, let us get into the code.

Preparing The Ports

First of all, we will need to import some libraries:

```
from queue import Queue
import socket
import threading
```

Socket will be used for our connection attempts to the host at a specific port.

Threading will allow us to run multiple scanning functions simultaneously.

Queue is a data structure that will help us to manage the access of multiple threads on a single resource, which in our case will be the port numbers. Since our threads run simultaneously and scan the ports, we use queues to make sure that every port is only scanned once.

Then, we will also define three global variables that we will use throughout the various functions:

```
target = "127.0.0.1"
queue = Queue()
open_ports = []
```

Target is obviously the IP-Address or domain of the host we are trying to scan.

The queue is now empty and will later be filled with the ports we want to scan.

And last but not least we have an empty list, which will store the open port numbers at the end.

We start by implementing the *portscan* function, which we have already talked about.

```
def portscan(port):
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.connect((target, port))
        return True
    except:
        return False
```

Here you can see a basic try-except block, in which we try to connect to our target on a specific port. If it works, we return True, which means that the port is open. Otherwise, we return False, which means that there was an error and we assume that the port is closed.

Now, before we get into the threading, we need to first define the ports we want to scan. For this, we will define another function called *get_ports*.

```
def get_ports(mode):
    if mode == 1:
        for port in range(1, 1024):
            queue.put(port)
    elif mode == 2:
        for port in range(1, 49152):
            queue.put(port)
    elif mode == 3:
        ports = [20, 21, 22, 23, 25, 53, 80, 110, 443]
        for port in ports:
            queue.put(port)
    elif mode == 4:
        ports = input("Enter your ports (seperate by blank):")
        ports = ports.split()
        ports = list(map(int, ports))
        for port in ports:
            queue.put(port)
```

In this function we have defined four possible modes. The first mode scans the 1023 standardized ports. With the second mode we add the 48,128 reserved ports. By using the third mode we focus on some of the most important ports only. And finally, the fourth mode gives us the possibility to choose our ports manually. After that we add all our ports to the queue.

Notice that when we enter the ports in mode four, we are splitting our input into a list of strings. Therefore, we need to map the typecasting function of the integer data type to every element of the list in order to use it.

Multithreading

The next thing we need to do is defining a so-called *worker* function for our threads. This function will be responsible for getting the port numbers from the queue, scanning them and printing the results.

```
def worker():
    while not queue.empty():
        port = queue.get()
        if portscan(port):
            print("Port {} is open!".format(port))
            open_ports.append(port)
        else:
            print("Port {} is closed!".format(port))
```

It is quite simple. As long as the queue is not empty, we get the next element and scan it. If the port is open, we print it and if it is not we print that as well. What we additionally do when a port is open, is adding it to our *open_ports* list.

Note: I would recommend not printing the information that a port is closed, since this information is useless and makes our output confusing. You can basically remove the else-branch.

So, now that we have implemented the functionality, we are going to write our main function, which creates, starts and manages our threads.

```
def run_scanner(threads, mode):

    get_ports(mode)

    thread_list = []

    for t in range(threads):
        thread = threading.Thread(target=worker)
        thread_list.append(thread)

    for thread in thread_list:
        thread.start()

    for thread in thread_list:
        thread.join()

    print("Open ports are:", open_ports)
```

In this function, we have two parameters. The first one is for the amount of threads we want to start and the second one is our mode. We load our ports, depending on the mode we have chosen and we create a new empty list for our threads. Then, we create the desired amount of threads, assign them our *worker* function and add them to the list. After that, we start all our threads and let them work. They are now scanning all the ports. Finally, we wait for all the threads to finish and print all the open ports once again.

```
run_scanner(100, 1)
```

By running the function like this, we scan all standardized ports. For this, we are using a hundred threads. This means that we scan around one hundred ports per second. You can increase that number if you want. The fastest I could go with my computer was around 800 port scans per second.

The results look like this:

```
Port 22 is open!
Port 25 is open!
Port 80 is open!
Port 110 is open!
Port 119 is open!
Port 143 is open!
Port 443 is open!
Port 465 is open!
Port 554 is open!
Port 563 is open!
Port 587 is open!
Port 631 is open!
Port 993 is open!
Port 995 is open!
Open ports are: [22, 25, 80, 110, 119, 143, 443, 465, 554, 563, 587, 631, 993, 995]
```