Python Network Port Scanner

By:

Abhishek Gautam

Roll Number: 1419210013

G.L.B.I.T.M

Knowledge Park III

Greater Noida

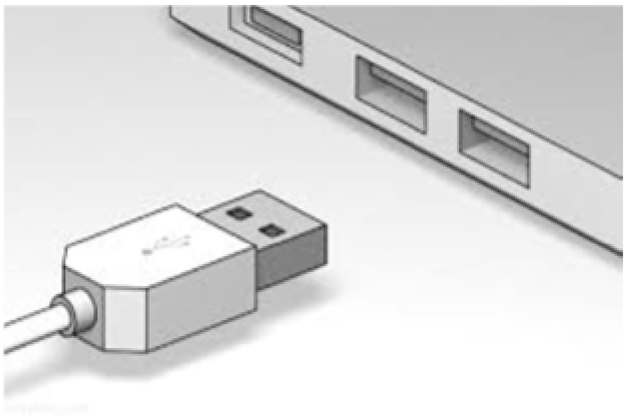
INDEX

INTRODUCTION

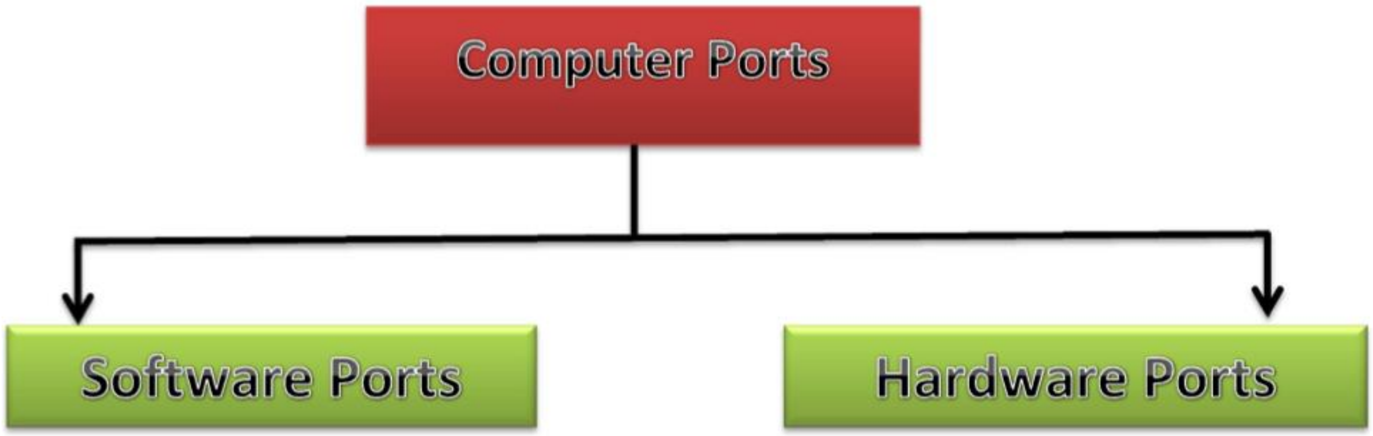
The ports acts like a gate in any communication established in any computer system. It can be regarded as an interface for the communication. The ports can be divided Into 2 subcategories as follows:

1> **Hardware Ports**
2> **Software Ports**

Most Importantly the universal fact common in each port is that "**EACH PORT IS ASSOCIATED WITH CERTAIN PROTOCOLS (FRAMEWORK)**" irrespective of the fact that it is software or hardware ports. To Establish Any Connection With the system the minimum requirements is these ports.



CLASSIFICATION OF PORTS



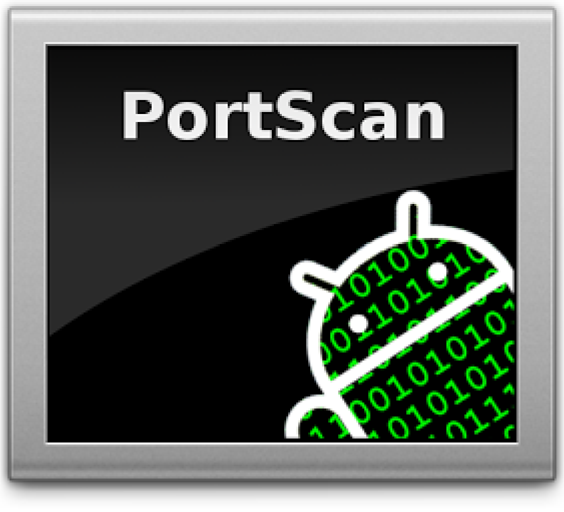
SOFTWARE PORTS

The concept of port numbers was established by the early developers of the **ARPANET** in informal co-operation of software authors and system administrators. A port is identified for each address and protocol by a 16-bit number, commonly known as the **port number.**

The transfer protocols such as the TCP(Transmission Control Protocol) and the UDP(User Datagram Protocol), specify a source and destination **PORT NUMBER** in their segment headers. A port number is a 16 bit unsigned interger ranging from **0 to 65535**.

** Port 0 cannot be used as it is reserved for TCP whereas for UDP a port zero can also be used which means simply NO PORT.

What is Port Scanning?



Definition:

A **port scan** or **portscan** can be defined as a process that sends client requests to a range of server port addresses on a host, with the goal of finding an active port.

A **port scanner** is a software application designed to probe a server or host for open ports. This is often used by administrators to verify security policies of their networks and by attackers to identify running services on a host with the view to compromise.

While not a nefarious process in and of itself, it is one used by hackers to probe target machine services with the aim of exploiting a known vulnerability of that service. However, the majority of uses of a port scan are not attacks and are simple probes to determine services available on a remote machine.

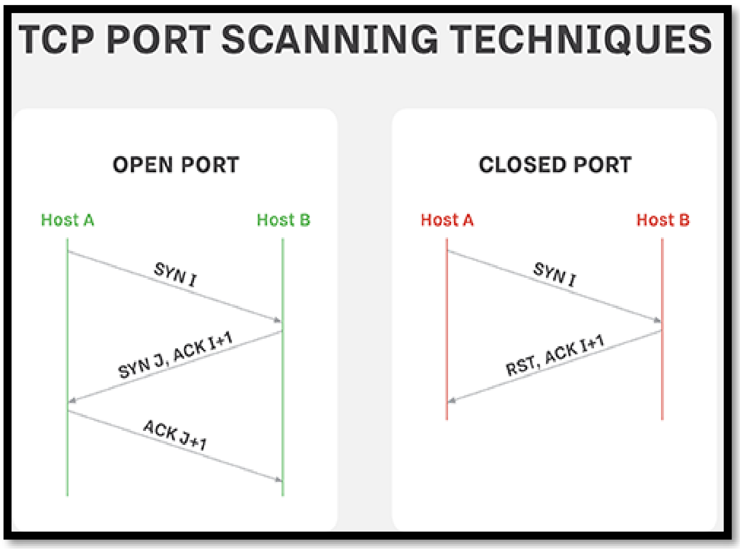
To **portsweep** is to scan multiple hosts for a specific listening port. The latter is typically used to search for a specific service, (For example, an SQL-based computer worm may portsweep looking for hosts listening on TCP port 1433)

Assumptions in Port Scanning

- All forms of port scanning rely on the assumption that the targeted host is compliant with **RFC 793 - Transmission Control Protocol.**
- Although this is the case most of the time, there is still a chance a host might send back strange packets or even generate false positives when the TCP/IP stack of the host is non-RFC-compliant or has been altered. This is especially true for less common scan techniques that are OS-dependent (FIN scanning, for example)
- The TCP/IP stack fingerprinting method also relies on these types of different network responses from a specific stimulus to guess the type of the operating system the host is running.

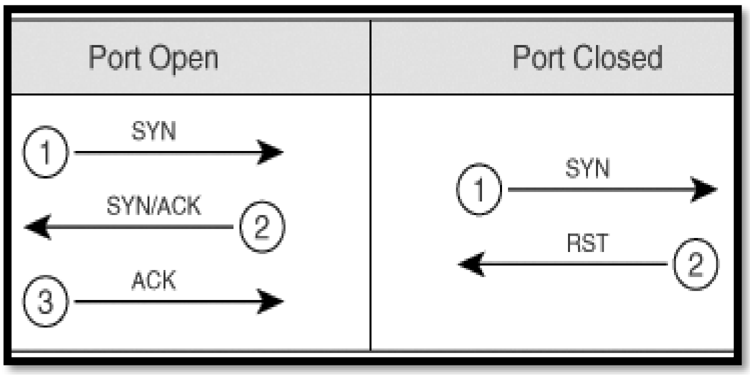
Types Of Port Scanning

1) TCP Port Scanning:



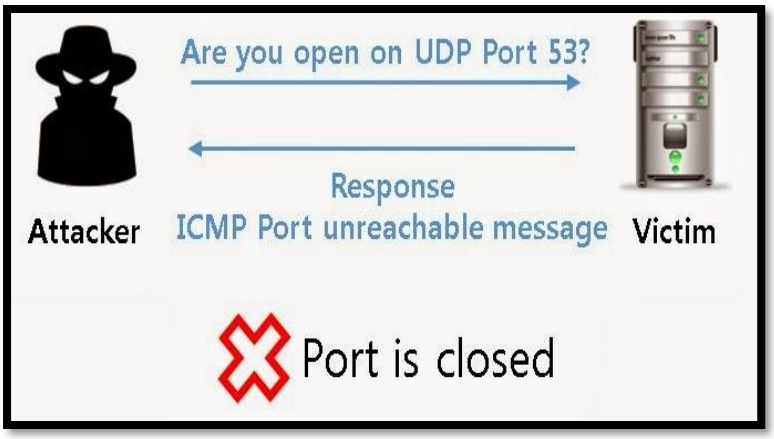
- ❖ If a port is open, the operating system completes the TCP three-way handshake, and the port scanner immediately closes the connection to avoid performing a Denial-of-service attack. Otherwise an error code is returned.
- ❖ This scan mode has the advantage that the user does not require special privileges.

2) SYN Scanning:



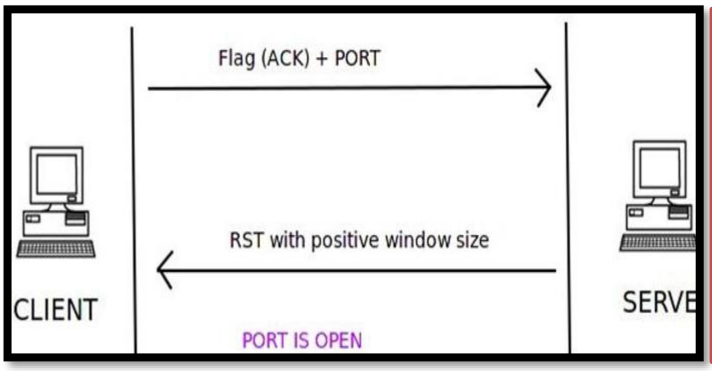
- ❖ SYN scan is another form of TCP scanning. Rather than use the operating system's network functions, the port scanner generates raw IP packets itself, and monitors for responses.
- ❖ This scan type is also known as "half-open scanning", because it never actually opens a full TCP connection.

3)UDP Scanning:



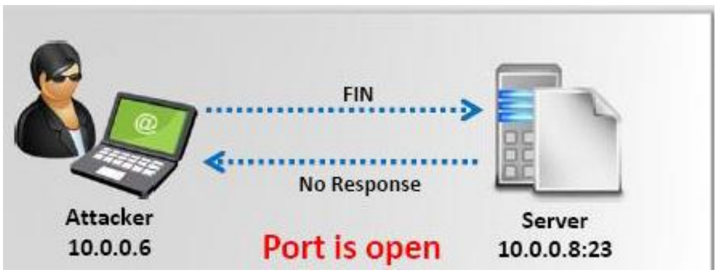
- ❖ UDP is a connectionless protocol so there is no equivalent to a TCP SYN packet.
- ❖ However, if a UDP packet is sent to a port that is not open, the system will respond with an ICMP port unreachable message. Most UDP port scanners use this scanning method, and use the absence of a response to infer that a port is open.
- ❖ However, if a port is blocked by a firewall, this method will falsely report that the port is open. If the port unreachable message is blocked, all ports will appear open. This method is also affected by ICMP rate limiting

4) ACK Scanning



ACK scanning is one of the more unusual scan types, as it does not exactly determine whether the port is open or closed, but whether the port is filtered or unfiltered. This is especially good when attempting to probe for the existence of a firewall and its rulesets. Simple packet filtering will allow established connections (packets with the ACK bit set), whereas a more sophisticated stateful firewall might not.

5) FIN Scanning



- FIN packets can bypass firewalls without modification. Closed ports reply to a FIN packet with the appropriate RST packet, whereas open ports ignore the packet on hand.

Python Port Scanner Code

```
import socket
import sys
import threading
import queue
import time
from datetime import datetime

print("""
*************************************************************
* ___  ____ ____ ___    ____ ____ ____ _  _ _  _ ____ ____  *
* |__] |  | |__/  |     [__  |    |__| |\ | |\ | |___ |__/  *
* |    |__| |  \  |     ___] |___ |  | | \| | \| |___ |  \  *
*                                                           *
*************************************************************""")
print("Made By: Abhishek Gautam")
#Defining Dictionary of common ports
common_ports = {
    "21": "FTP",
    "22": "SSH",
    "23": "Telnet",
    "25": "SMTP",
    "53": "DNS",
    "67":"DHCP",
    "68":"DHCP",
    "69":"TFTP",
    "80": "HTTP",
    "110":"POPv3",
    "123":"NTP",
    "143":"IMAP",
    "194": "IRC",
    "389":"LDAP",
    "443": "HTTPS",
    "3306": "MySQL",
    "25565": "Minecraft"
}

#printing basic info about the scans
print("\n[*]Host: {}       IP: {}  ".format(sys.argv[1],
socket.gethostbyname(sys.argv[1])))

#returns the value of host , start port and end port
def get_scan_args():
    if len(sys.argv) == 2:
        print("\n[*]Starting Port: {}       Ending Port: {}".format(0, 1024))
        return (sys.argv[1], 0, 1024)
    elif len(sys.argv) == 3:
        print("\n[*]Starting Port: {}       Ending Port: {}".format(1,
sys.argv[2]))
```

```
        return (sys.argv[1],0,  int(sys.argv[2]))
    elif len(sys.argv) == 4:
        print("\n[*]Starting Port: {}       Ending Port:
{}".format(sys.argv[2], sys.argv[3]))
        return (sys.argv[1], int(sys.argv[2]), int(sys.argv[3]))


#tries to establish a connection
def is_port_open(host, port): #Return boolean
    try:
        # create an INET, STREAMing socket
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(0.5)
        sock.connect((host, port))
    except socket.error:
        return False
    return True

def scanner_worker_thread(host):
    while True:
        port = port_queue.get()
        if is_port_open(host, port):
            if str(port) in common_ports:
                print("[*]{}({}) is OPEN!".format(str(port),
common_ports[str(port)]))
            else:
                print("[*]{} is OPEN!".format(port))
        port_queue.task_done()


scan_args = get_scan_args()
port_queue = queue.Queue()
for _ in range(30):
    t = threading.Thread(target=scanner_worker_thread, kwargs={"host":
scan_args[0]})
    t.daemon = True
    t.start()

start_time = time.time()
print("[*] Scanning started at %s    \n" %(time.strftime("%H:%M:%S")))
for port in range(scan_args[1], scan_args[2]):
    port_queue.put(port)

port_queue.join()
end_time = time.time()
print("\n\n[*] Scanning ended at %s    \n" %(time.strftime("%H:%M:%S")))
print("Done! Scanning took {:.3f} seconds.".format(end_time - start_time))
```

Python Port Scanner Code Explanation

1) Libraries Imported

1.1) Socket Library:

This module provides access to the BSD *socket* interface. This is one of the most useful library for this project as it contains all the necessary function which are requires to make a connect request and get a response from the port.

1.2) Sys Library:

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available. This library was used in the script to give access to the command line argument being passes at runtime by the user using **sys.argv**

1.3) Threading Library:

This module constructs higher-level threading interfaces on top of the lower level _thread module. This module was used in the script to implement multi-threading to fasten up the execution time of the program.

1.4) Queue Library:

The queue module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The Queue class in this module implements all the required locking semantics. It depends on the availability of thread support in Python; see the threading module. This library assigns the ports to multiple threads and ask them to do required computation to check if the port is open or not.

1.5) Time Library:

This module provides various time-related functions. We used time.time() method of library which help us to calculate the total execution time.

1.6) Datetime Library:

The datetime module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient attribute extraction for output formatting and manipulation. For related functionality, see also the time and calendar modules.

2) Inbuilt function:

2.1) print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False) :

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file* and *flush*, if present, must be given as keyword arguments.

2.2) socket.gethostbyname(hostname)

Translate a host name to IPv4 address format. The IPv4 address is returned as a string, such as '100.50.200.5'

2.3) len(list)

Returns the length of the list passed to it as an argument.

2.4) socket.socket(socket.AF_INET, socket.SOCK_STREAM)

Create a new socket using the given address family, socket type and protocol number. The address family should be AF_INET (the default),AF_INET6, AF_UNIX, AF_CAN or AF_RDS. The socket type should be SOCK_STREAM (the default), SOCK_DGRAM, SOCK_RAW or perhaps one of the other SOCK_ constants. The protocol number is usually zero and may be omitted or in the case where the address family is AF_CAN the protocol should be one of CAN_RAW or CAN_BCM. If *fileno* is specified, the other arguments are ignored, causing the socket with the specified file descriptor to return.

2.5) sock.settimeout(value)

Set a timeout on blocking socket operations. The *value* argument can be a nonnegative floating point number expressing seconds, or None. If a non-zero value is given, subsequent socket operations will raise a timeout exception if the timeout period *value* has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If None is given, the socket is put in blocking mode.

2.6) socket.connect(address)

Connect to a remote socket at *address*. If the connection is interrupted by a signal, the method waits until the connection completes, or raise a socket.timeout on timeout, if the signal handler doesn't raise an exception and the socket is blocking or has a timeout. For non-blocking sockets, the method raises anInterruptedError exception if the connection is interrupted by a signal (or the exception raised by the signal handler).

2.7) Queue.get(*block=True*, *timeout=None*)

Remove and return an item from the queue. If optional args *block* is true and *timeout* is None (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the Empty exception if no item was available within that time. Otherwise (*block* is false), return an item if one is immediately available, else raise the Empty exception (*timeout* is ignored in that case).

2.8) **Queue**.task_done**()**

Indicate that a formerly enqueued task is complete. Used by queue consumer threads. For each get() used to fetch a task, a subsequent call totask_done() tells the queue that the processing on the task is complete.

2.9) **queue**.Queue**(*maxsize=0*)**

Constructor for a FIFO queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

2.10)**threading**.Thread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, ***, *daemon=None*)**

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-*N*" where *N* is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If not None, *daemon* explicitly sets whether the thread is daemonic. If None (the default), the daemonic property is inherited from the current thread.

2.11) **Queue**.put**(*item*, *block=True*, *timeout=None*)**

Put *item* into the queue. If optional args *block* is true and *timeout* is None (the default), block if necessary until a free slot is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the Full exception if no free slot was available within that time. Otherwise (*block* is false), put an item on the queue if a free slot is immediately available, else raise the Full exception (*timeout* is ignored in that case).

2.12) **Queue.join()**

Blocks until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls task_done() to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, join()unblocks.

3) User Defined Functions:

3.1) get_scan_args( )

It is used to get the input for scan. It checks that how many command line arguments the user passed and then accordingly set the value for scan. If user specifies the host, starting port and ending port the those values are used else if end port argument is missing then it is set to **1024,** if starting port address is missing then it is set to **1.**

3.2) is_port_open(host, port)

It's a basic function to check if the port is open or not. It takes two arguments hosts and port. It's going to return a Boolean value. In python when we try to connect to a port then if the port is open then it simply connects otherwise an exception is raised

3.3) scanner_worker_thread(host)

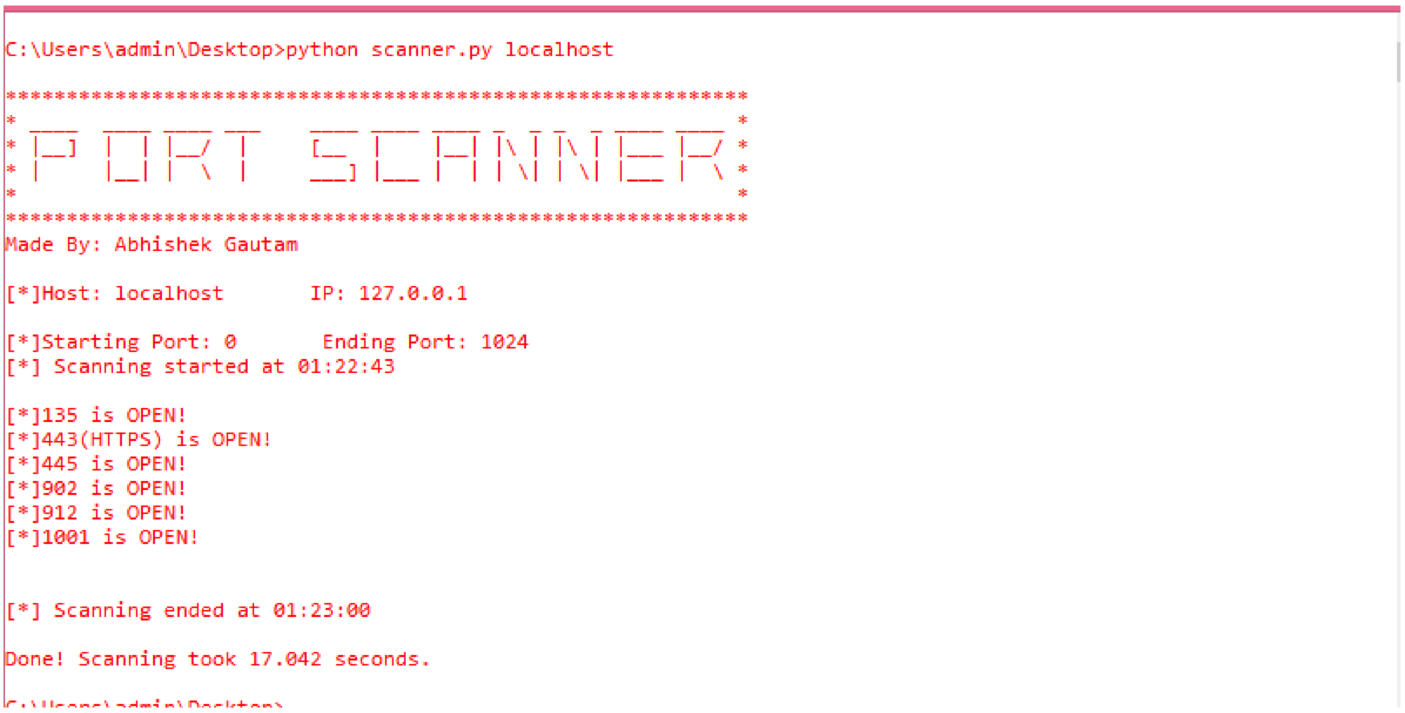
This function takes a port from the FIFO queue created and checks if port is open by using function is_port_open and prints the result accordingly.

4) Exceptions:

4.1) *exception* **socket**.error

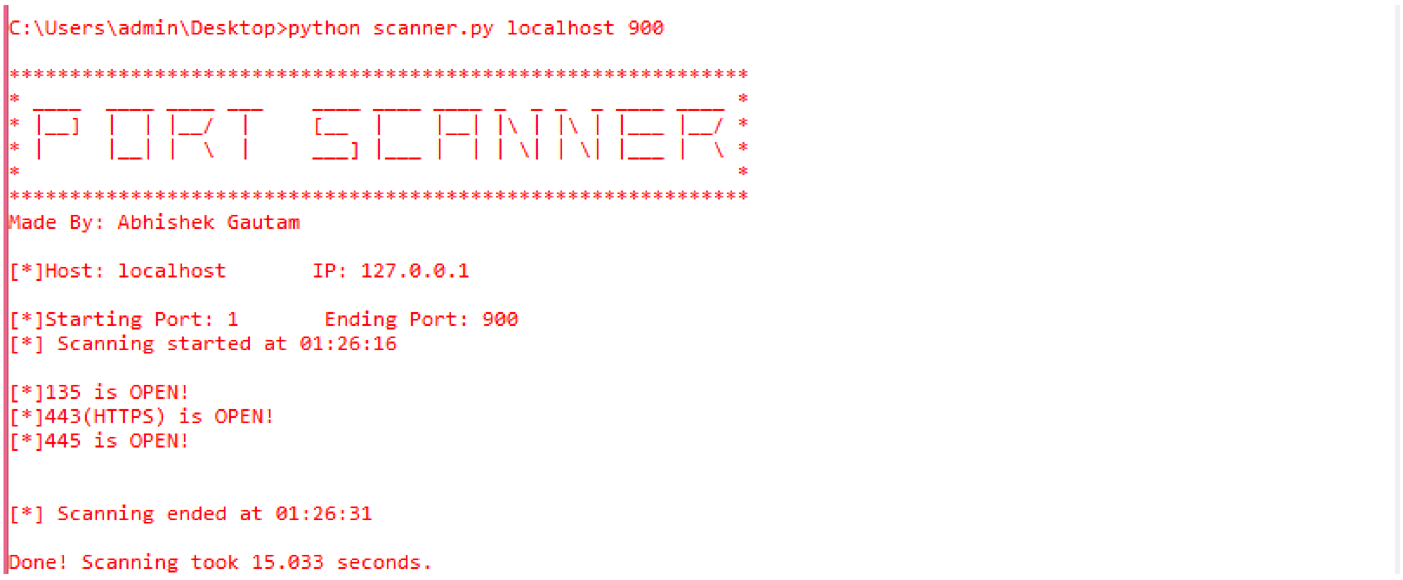
This exception is raised for socket-related errors. The accompanying value is either a string telling what went wrong or a pair (errno, string)representing an error returned by a system call, similar to the value accompanying **os.error**. See the module **errno**, which contains names for the error codes defined by the underlying operating system.

OUTPUTS



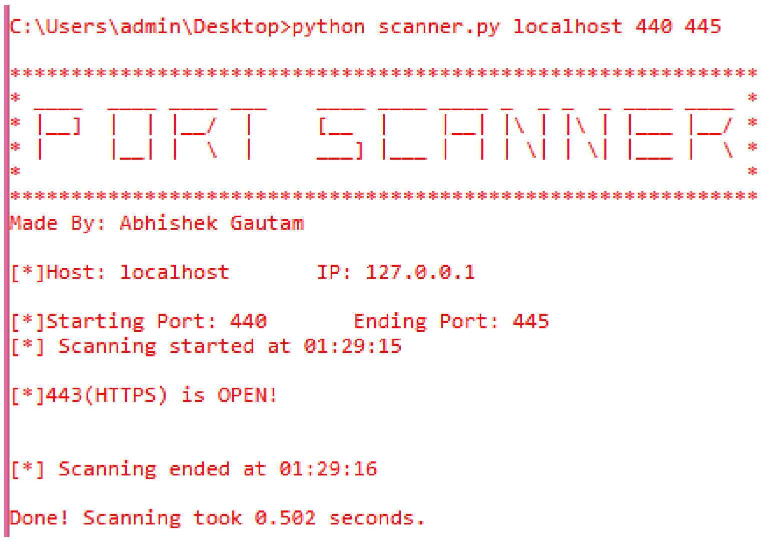
```
C:\Users\admin\Desktop>python scanner.py localhost

*************************************************************
* ____  ____ ____ ___   ____ ____ ____ _  _ _  _ ____ ____ *
* |__]  |  | |__/  |    [__  |    |__| |\ | |\ | |___ |__/ *
* |     |__| |  \  |    ___] |___ |  | | \| | \| |___ |  \ *
*                                                           *
*************************************************************
Made By: Abhishek Gautam

[*]Host: localhost       IP: 127.0.0.1

[*]Starting Port: 0       Ending Port: 1024
[*] Scanning started at 01:22:43

[*]135 is OPEN!
[*]443(HTTPS) is OPEN!
[*]445 is OPEN!
[*]902 is OPEN!
[*]912 is OPEN!
[*]1001 is OPEN!


[*] Scanning ended at 01:23:00

Done! Scanning took 17.042 seconds.

C:\Users\admin\Desktop>
```

Figure 1: Running Scan without passing any starting or ending port numbers



```
C:\Users\admin\Desktop>python scanner.py localhost 900

*************************************************************
* ____  ____ ____ ___   ____ ____ ____ _  _ _  _ ____ ____ *
* |__]  |  | |__/  |    [__  |    |__| |\ | |\ | |___ |__/ *
* |     |__| |  \  |    ___] |___ |  | | \| | \| |___ |  \ *
*                                                           *
*************************************************************
Made By: Abhishek Gautam

[*]Host: localhost       IP: 127.0.0.1

[*]Starting Port: 1       Ending Port: 900
[*] Scanning started at 01:26:16

[*]135 is OPEN!
[*]443(HTTPS) is OPEN!
[*]445 is OPEN!


[*] Scanning ended at 01:26:31

Done! Scanning took 15.033 seconds.
```

Figure 22: Running scan by passing host and end port address



```
C:\Users\admin\Desktop>python scanner.py localhost 440 445

*************************************************************
* ____  ____ ____ ___    ____ ____ ____ _  _ _  _ ____ ____ *
* |__]  |  | |__/  |     [__  |    |__| |\ | |\ | |___ |__/ *
* |     |__| |  \  |     ___] |___ |  | | \| | \| |___ |  \ *
*                                                           *
*************************************************************
Made By: Abhishek Gautam

[*]Host: localhost       IP: 127.0.0.1

[*]Starting Port: 440       Ending Port: 445
[*] Scanning started at 01:29:15

[*]443(HTTPS) is OPEN!


[*] Scanning ended at 01:29:16

Done! Scanning took 0.502 seconds.
```

Figure 3 Running scan by passing both the starting as well as ending port address

Bibliography

1) https://en.wikipedia.org/wiki/Port_scanner
2) https://docs.python.org/3/library/socket.html
3) https://docs.python.org/3.0/library/sys.html
4) https://docs.python.org/3/library/threading.html
5) https://docs.python.org/3/library/queue.html#module-queue
6) https://docs.python.org/3/library/time.html
7) https://docs.python.org/3/library/socket.html#socket.socket.settimeout