**Port Scanner Report**

**Ali Khalfan**
**Brennon York**

September 26, 2010

# Introduction

We have implemented a basic port scanner / probing device. All requirements that were stated within the problem description have been handled accordingly. Each section below briefly describes the algorithms we used to solve the problems and any issues that might have arisen.

# TCP Probing

The first step of TCP probing was detecting whether the port is open or closed. To do this, the socket will attempt to connect to the specified host. This method instantly determines the status of the port. A timeout would indicate that the port is either closed, or it might be filtered. Due to the possibility of a long timeout that might cause a hang[1], a maximum threshold has been set such that if a connection was not established in a certain period of time a timeout is assumed. Timeouts are handled using the **select()** function. In order to effectively use it, the socket was set to non-blocking during the call to the **connect()** function. Then, **select()** was invoked to handle the timeout.
The program then checks to see what service belongs to the port (using getnameinfo())
Verifying services on ports was based on what was listed in the RFCs related to the protocols. To start with, for the services that required greeting, the program simply read the buffer coming from the **connect()** response. In **SSH**, RFC4253 specifically states that the format of the SSH greeting from the server will be **SSH-protoversion-softwareversion SP comments CR LF**. For SMTP, RFC2821 specifies "220 Domain [ SP text ] CRLF" as a greeting to be sent by the service, it also specifies that 554 may be sent instead of 220. Therefore we look for either 220 or 554. The software and the version is extracted from the [ SP text ] and included in the output. However, the RFC explicitly states that the server "**MAY** include an identification of their software of version." It is understandable that the server may not include this information. However, we print whatever is under [SP text], since that is where the software and version would likely be.

**POP,** According to RFC 1939 "There are currently two status indicators: positive ("+OK") and negative ("-ERR"). Servers MUST send the "+OK" and "-ERR" in upper case. **This is the method we will use for verification.** There does not seem to be however, anyway to verify that the software or version will be printed, hence we will print the full string that comes in for POP since the software and version will most likely be included in this string.

**IMAP,** RFC 3501, states that a response in IMAP will include OK/NO/BAD. However, as in POP, there is nothing that says the version will be printed or whether there is a way to print the version at all. Hence we shall print whatever data coming out after the OK/NO/BAD message in the greeting and assume that the software and version will most likely be in that string (if anywhere)

For the services that require a request to be sent such as **WHOIS**: This is a protocol that does not require a greeting from the server. Therefore, the client (i.e. the program) sends a query and expects a response from the server. The program, here sends a dummy domain using the **send()** function. RFC3912 explicitly states the form of the reply will be "get answer $< - - - -$"Info about Smith $< CR >< LF >$". Therefore, the program checks (using the **recv()** function) to see if this is the form of the reply for verification. **HTTP** also does not send a greeting. The program sends a HEAD request utilizing the **send()**, given that a HEAD request does not send a message body. The program then accepts a response that adheres to the specifications listed RFC2616. This will all be done as in the WHOIS, using the **send()** and **recv()** functions. To verify that this is an HTTP server, the program will check for the format listed in section 6 in this RFC, where the expected status line is **HTTP-Version SP Status-Code SP Reason-Phrase CRLF**.

It should be stressed here that printing the version here will not always be reliable, whatever the service is. Most of the RFCs relax the requirement of having the software or version sent based on security. In addition, just because a service is not sending its software and version, does not mean it is not acting accordingly. This could certainly be considered a limtation in that the user is not guaranteed to see a software and service

[1] Running on tank for example, gave us a delay of over two minutes for a host within the IU subnet

everytime. We therefore, relax some of the restrictions for verifying the version information in the output. For example, the program would check to make sure that the port sends data, and that the data adheres to some minimum requirements (e.g. in the HTTP scenario, the host is expected to send the protocol version or the +OK reply). In addition, besides the RFC4253 (SSH) and the HTTP header requirements which explicitly states where the version will be printed, there is no definite way to know exactly the location of the software and version in other fields. Therefore we decided to print the entire string for IMAP, POP, and SMTP.

## UDP Probing

For UDP there were three possible states. Either the server does not respond to the sendto() function, which might indicate that the port is either open or filtered. Otherwise, an ICMP error might be sent, indicating that the port is closed. The program handles this by checking the return result of the recvfrom() function. If it is an ECONNREFUSED error then the port is assumed closed. However, if this is a valid response, meaning that the process receives a response that is not an ICMP error, then the port will be open. Perhaps the most important limitation is that servers might have a limit of how many ICMP errors they may send per time interval. Therefore, it is possible if the scan is done on all ports of a host that many ports will return an **open/filtered** response since the ICMP error will not be sent by the remote host.

## Multi-Threading

When implementing the multi-threaded version of this program the first aspect that was taken into consideration was the ability to pass the port scanning function into its own thread. To do that we needed to change the port probing function to be a pointer that could only take one argument, a voided pointer to a structure containing all of the arguments the thread needs.

This style of passing arguments allows each of the threads called to be considered "thread-safe" because, although they are only pointers, no data is being written to those areas of memory after the function is called. Another feature of passing arguments in this manner, instead of the more generic "pool" methodology, is that now each thread can run concurrently without any fear of a deadlock. If each thread is using only its own variables that were passed in and variables defined when the function was called then there is no need to mutex_lock a given section of code. Because of this there is no fear of deadlocks or threads that wait on other threads to execute specific areas of code slowing down the process in its entirety. Also, because each thread can run in its own domain, it means we do not need to worry about thread scheduling.

One issue that did arise was the ability to determine whether or not a thread were "hanging," or not returning. This was handled by the select function when attempting to connect to poll the reading of input/output buffers from the kernel since it forced a timeout. Because of this method we were able to safely ensure that each thread, whether it connected successfully or not, would return within a short period of time specified to our standards.

To ensure that all threads were completing and returning before the next set was called we had to join them into the 'main' thread (main being considered the initial thread created to execute the program). After each thread was created, and arguments passed, the program entered a second for loop that would join all of the threads just previously created. This essentially put the main program in a time out until all threads created had returned. Once they all returned, whether successful or unsuccessful, the program would reenter the main for loop and poll the next set of values to pass into the next set of threads. This allowed for the maximum amount of threads to do the maximum amount of work without indefinitely stalling the main thread of the program. Once all threads were called and execution was placed back into the main function, it calls the pthread_exit function. This allows for any threads still working after the return of the main program to continue work and exit properly once they return. The pthread_exit function was not needed anywhere other than this because the joining function earlier mentioned automatically closes the threads after they return and 'join' into the main thread.

Lastly, we implemented the threading in such a way that if the number of threads exceeds the number of IP addresses given, if the user uses the '–file' or '–prefix' command, then the number of threads will automatically be reduced to the maximum number of IP addresses. This was done because the solution to utilize

those extra threads to scan ports while scanning IP addresses would've become quite dangerous and difficult.

## Program Grammar

When calling the program there are certain guidelines that must be followed to obtain the correct results. It is a key factor to note that, although we understand the need for sanitized input, we did not implement it with this design because this work is more of a proof-of-concept rather than software to be released into the public domain.