Project 2: A Basic Port Scanner

Assigned: 10/2/12, Due: 10/26/12 (Friday)

Project Goals

Port scanner software can probe machines for open ports. Port scanners are often used by network administrators to verify the security of machines in their network. They are of interest to Internet miscreants as well, since attackers can use them to find machines to compromise simply by probing for software versions with known vulnerabilities. An example of a well known, open source port scanner is Nmap.

In this project you will create a basic port scanner with both IPv4 and IPv6 support that is written for network administrators interested in ensuring that machines on their network run only expected services. In addition to helping you gain experience with socket programming, this project will help you appreciate the interplay of various implementation of firewalls, transport protocols and operating systems.

Scanner Details

There are two important aspects to any scanning program: 1) the port states it can detect and 2) the techniques it uses to make its inference.

Port States

We expect your program to recognize the following five port states:

1. **Open:** This state implies that the application is actively accepting TCP connections or UDP datagrams on this port. Finding these is the primary goal of your program.
2. **Closed:** This state implies that the port is accessible to the scanner but there is no application listening on it.
3. **Filtered:** Filtering devices, such as firewalls, can prevent scanners from inferring if a port is open. In such cases, your program should try to infer if a port is *filtered*.
4. **Unfiltered:** This state means that the port is accessible but no inference can be made on whether it is *open* or *closed*.
5. **Open|Filtered:** There are cases when open ports give no response. In such cases, the

lack of response can either imply that the port is *open* or that it is *filtered*.

Port Scanning Techniques

Most of the scan types mentioned below are only available to privileged users since scanners send and receive raw packets, which requires root access on Unix systems. While you can experiment some on your personal PCs where you have administrator privileges, we will release a list of CS machines where you would have adequate privileges for this project. Also, note that each scan type is limited in what it can infer.

1. **TCP SYN scan (Half-Open scan):** This is the most popular scanning option since it is fast and unrestricted by firewalls that want to permit connecting on certain ports. It is also relatively stealthy, since it never completes TCP connections. The basic idea of this technique is simple: you send a TCP SYN packet, as if you were going to open a real connection and then wait for a response. A SYN/ACK indicates that the port is *open* while a RST indicates that the port is *closed*. If no response is received after several retransmissions, the port should be marked as *filtered*. The port is also marked as *filtered* if an ICMP unreachable error (type 3, code 1, 2, 3, 9, 10, or 13) is received. Finally, since a rare feature of TCP can sometimes just send the SYN packet in response to a SYN, mark the port as *open* in that case as well.
2. **TCP NULL, FIN, and Xmas scans:** The TCP specification says that if a port is *closed*, it should send back an RST in response to a packet containing anything other than a SYN, ACT or RST. No response is expected when the port is *open*. For machines compliant with this specification, scanners can do *NULL scan* (no bits of TCP flags are set), *FIN scan* (just the FIN bit is set) or *Xmas* scan (sets the FIN, PSH and URG flags, lighting the packet like a Christmas tree) to infer the state of the port. Note, however, that since firewall devices can block responses, lack of response can only mean that the port is *open|filtered*. Further, in cases where a firewall device sends an ICMP unreachable error (type 3, code 1, 2, 3, 9, 10 or 13), the port can be marked *filtered*.
3. **TCP ACK scan:** An ACK scan probe packet only has the ACK flag set. In the lack of firewall devices, both open and closed ports will return an RST packet and a scanner can conclude that the port is *unfiltered*. Ports that do not respond or send ICMP error messages back (type 3, code 1, 2, 3, 9, 10 or 13) are labeled filtered. Note that this scan cannot infer whether a port is *open* or *closed*.
4. **Protocol scan:** A protocol scan attempts to determine which transport layer protocols a host is responding to. This can be accomplished by iterating through the 8-bit protocol (known as "next header" in IPv6) field in the IP header. The rest of the packet should be empty - except in the cases of TCP, UDP, and ICMP. For those 3 protocols, a valid transport layer header should exist, though it still need not have any payload.

Project Specification

The basic idea behind a port scanner is simple: Given an IP address of a machine and a list of interesting TCP or UDP ports to scan, the scanner will connect on each port using TCP or UDP sockets, make a determination of whether or not the port is open based on success of the connection request and close the socket before moving on to the next port to scan.

Specifics of Expected Port Scanner Functionality

Your scanner should run on CS Linux machines and you must write it in C/C++. An administrator would invoke it as: "./*portScanner* [option1, ..., optionN]". Implement the following options:

- --help <display invocation options>
- --ports <ports to scan>
- --ip <IP address to scan>
- --prefix <IP prefix to scan>
- --file <file name containing IP addresses to scan>
- --speedup <parallel threads to use>
- --scan <one or more scans>
- --protocol-range <transport layer protocols to scan>

Details of each option are given below:

- **help:** When *portScanner* is invoked with this option, it should display the various options available to the user.
- **ports:** Your *portScanner* will scan all ports [1-1024] by default. However, if this option is specified, it will scan ports specified on the command line. The latter could be individual ports separated by a comma or a range.
- **ip/prefix/file:** These options allow a user to scan an individual IP address, an IP prefix, or a list of IP addresses from a file respectively. The scanner should determine whether the given address is IPv4 or IPv6 and use the appropiate version of the protocol. A file may contain a mixture of IPv4 and IPv6 addresses. When IP addresses are specified in a file, you can assume them to be one on each line. A user may invoke the *portScanner* with more than one of these options. If none of these options are specified, check for the presence of an individual IP address as an argument. If that check also fails, flag an error and ask the user to try again.
- **speedup:** The *portScanner* would be single threaded by default. This would keep the implementation simple but it will be slow. Specifying this option will allow a user to use the faster, multi-threaded version of the *portScanner*. The user will specify the

number of threads to be used.

- **scan:** The *portScanner* will preform all scans by default. However, the user may select any subset of scans that they wish. This is done using: SYN, NULL, FIN, XMAS, ACK, Protocol. Multiple of these may be specified implying that each of the listed scans should be run. For example, **./portScanner --ip 127.0.0.1 --scan ACK NULL SYN** would scan the localhost (127.0.0.1) with the TCP ACK scan, the TCP NULL scan, and the TCP SYN scan, but none of the others.
- **protocol-range:** The *portScanner* will scan all possible transport protocols for a protocol scan by default. However, the user may select any subset of the 255-number range they wish. This should be specified after this option with either a range or a comma seperated list - just like the ports to scan option.

Output: After each invocation, the *portScanner* should output a succinct summary of the list of open TCP ports on each IP address. Additionally, for each open port, it will include the name of the service that is likely running. In the case when an IP prefix is asked to be scanned, this information should be at the prefix granularity. To find services associated with ports [1-1024], visit http://www.iana.org/assignments/port-numbers.

When multiple scans are used, the results for each scan should be included in the output. Additionally, the combined inference from all of the scans run should be listed. Doing this requires you to determine a way to combine possibly contradictory results from different scans. Prepared to justify your method to the evaluator during the code review.

Verifying standard services: For **SSH**, **HTTP**, **SMTP (port numbers 25 and 587 can be used)**, **POP**, **IMAP**, and **WHOIS**, your program should verify that the port is indeed running the standard service expected on that port. Upon verification, find out the specific version of software as well. Inferring this information will require you to do one of the following: 1) parse identifying information sent by the service if it sends it or 2) send appropriate queries to cause the service to reveal this information to you. For example, SSH sends you the version information immediately upon connection but for HTTP you must send a valid query to the remote server and read back the header of its response to find the version of the software it is running.

Timeouts and malicious targets: As the machines you are expected to scan are not under your control, your program should be designed to handle arbitarily bad responses from the remote machine. No response from the remote machine should cause your program to crash. Likewise, your program should timeout in a reasonable period of time (generally a handful of seconds), no matter how the remote machine responds.

Expectations on concurrency: We require in the speedup option that multiple threads be spawned to divide the work load. Threads should not be idle due to the static division of large amounts of work. For example, if requested to use 5 threads to scan 5,000 ports you should **not** simply initially assign each thread to scan 1,000 ports - some threads may finish much faster, and should then assist the slower running threads. Additionally, your program should not continually create new threads. For example, you should **not** create a thread to scan a port, scan the port, record the results, and then destroy the thread only to repeat the process for the next port. No possible execution paths of these concurrent threads should lead to an error. Specifically, ensure that you protect all functions and library calls explicitly unless they are noted to be thread safe. All memory should be freed when your server program exits. Specifically, there should not be any memory leaks or zombie threads. Do not rely on the OS to clean up terminated threads.

Resources and Restrictions

Getting Started: Begin by familiarizing yourself with the Nmap software. A simple starting point is to scan your machine, aka, *localhost*, via "nmap 127.0.0.1". Another useful resource is *telnet*, which will allow you to interact with a server using a plain text command line. For example, "telnet burrow.cs.indiana.edu 22" will allow you to connect to the SSH service running on *burrow.cs.indiana.edu* on port 22. *burrow.cs.indiana.edu* will respond by telling you a bit about the SSH service and will wait for you to send the appropriate authentication.

Implementation Specifics: As stated earlier, you are required to use either C or C++ for this project. Additionally, you must use the natives Linux/BSD socket system calls to open sockets. You may not use other socket libraries. No credit will be given to solutions that do not adhere to these requirements. These restrictions are being made so you become familiar with the details of lower-level socket programming. If you choose to do so, you may however use libpcap capture packets, although you should still create / analyze them manually.

To create a multi-threaded version of your program, use the *pthreads* library. This library can be enabled with the *gcc* compiler using specify the -pthread option. Additionally, the -D_REENTRANT *gcc* option, which make code libraries re-entrant, may come in handy because it can help the compiler load copies of libraries where it is safe to have two threads call the same function.

The following system calls will likely be required to complete the assignment: **connect**, **htons**, **pthread_cancel**, **pthread_create**, **pthread_join**, **pthread_kill**, **recv**, **send**, **setsockopt**, **sleep**, **socket**. You may also want to look at the more advanced synchronization methods available in pthreads.

Other Resources: You cannot copy any code from the Internet. However, you are encouraged to avail other Internet resources and Linux manual pages when completing this Project. Socket tutorials such as, http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html, will be helpful in understanding socket programming. However, **you must ensure you write your own code** and that you **be explicit about what resources you use**, including web tutorials and discussions with individuals outside of your group.

Experiment with Caution: Exercise frugality in testing your program since system administrators of organizations whose machines you scan would likely get upset and complain to our system administrators, who in turn may have to limit the scope of the project beyond the point where you would find it a useful practical experience.

Roadmap: Build your program incrementally. Here is a road-map to use:

1. Get your program to scan one TCP port on an IP address using one scanning technique and display the entire output on the screen.
2. Parse the output to derive conclusions about which ports are open.
3. Expand the functionality for other TCP port scanning techniques.
4. Add scanning of transport protocols.
5. Expand the program to scan ports in a loop.
6. Verify that ports for SSH, HTTP, SMTP, POP, IMAP, and WHOIS are indeed running these services.
7. Add IPv6 support.
8. Add functionality for options related to IP addresses and ports to scan.
9. Develop a multi-threaded version of the program.
10. Detect and recover from failed/hanging threads.

Deliverables and Grading

Submit your code and project files as a single archive file (.tar or .tar.gz file formats only) via OnCourse by 11:59pm of the day of the deadline. Shorly after the submission deadline, demo slots will be posted on the Demonstration Scheduling System (a reminder will be posted on the Web board). You must schedule an appointment to demonstrate your project. Groups that fail to demonstrate their project will not receive credit for the project. If a group member fails to attend his or her scheduled demonstration time slot, it will result in a 10 point reduction in his or her grade.

In addition to testing your code for various test cases, the AIs will be explicitly evaluating the contributions of individual project partners. In cases where they determine that partners have not contributed equally, differential grading will be used. The instructor and the AIs reserve the right to determine appropriate penalty in such cases.