A Level Computer Science Non-Examined Assessment (NEA)

Sam Leonard

Contents

1 Analysis

1.1 Identification and Background to the Problem (Core)

There are many situations in which exploring a network from a "black box" perspective can be useful, such as a network engineer who wants to know what an outsider to the network such as a malicious hacker might see. This can help to secure the network against malicious threats and also in some cases detect ports that shouldn't be listening and investigate why they are open, i.e. a backdoor placed by a hacker to allow them to access a computer remotely.

1.2 Analysis of problem (Core)

The problem with looking at a network from the outside is that the purpose of the network is to allow communication inside of the network, thus very little is exposed externally. This presents a challenge as we want to know what is on the network as well as what each of them is running which is not always possible due to the limited information that services will reveal about themselves. Firewalls also play large part in making scanning networks difficult as sometimes they simply drop packets instead of sending a TCP RST packet (reset connection packet). When firewalls drop packets it becomes exponentially more difficult as you don't know whether your packet was corrupted or lost in transit or if it was just dropped.

1.3 Numbered List of Objectives (also called Success Criteria, the end user requirements) (Core)

1. Show a basic usage message when called with no arguments.
2. Show a help message when called with `-help` or `-h`.
3. Scan the 1000 most commonly used TCP ports when called with just an IP address.
4. Scan the ports specified by `-p <ports>` or `-ports <ports>`.
5. Parse either a comma separated list of ports e.g. `1,2,3,4` or a range specified set of ports e.g. `1-4`.
6. Scan all ports in each scan type when called with `-p-`.
7. Don't scan the ports specified by `-exclude-ports <ports>` in any scan.
8. Expand a Classless Inter-Domain Routing (CIDR) specified subnet when used in the target specification.
9. List all of the IP addresses that would be scanned when given the `-sL` flag.
10. Only ping each specified address when supplied the `-sn` flag.

11. When doing a ping scan (`-sn`) display the Time To Live (TTL) and latency of each host.

12. Don't ping each of the hosts before scanning to check if they are up when supplied with the `-Pn` flag.

13. Perform a TCP SYN scan on the 1000 most common TCP ports on each target specified when given the `-sS` flag.

14. Perform a TCP `Connect()` scan on the 1000 most common TCP ports on each target specified when given the `-sT` flag.

15. Perform a UDP scan on the 1000 most common UDP ports when on each target specified when given the `-sU` flag.

16. Perform version detection on the services running on each of the hosts specific when given the `-sV` flag.

1.4 Description of current system or existing solutions (Core if relevant)

Nmap is currently the most popular tool for doing port scanning and host enumeration. It supports the following scanning types:

- TCP: SYN
- TCP: `Connect()`
- TCP: ACK
- TCP: Window
- TCP: Maimon
- TCP: Null
- TCP: FIN
- TCP: Xmas
- UDP
- Zombie host/idle
- SCTP: INIT
- SCTP: COOKIE-ECHO
- IP protocol scan
- FTP: bounce scan

As well as supporting a vast array of scanning types it also can do service version detection and operating system detection via custom probes. Nmap also has script scanning which allows the user to write a script specifying exactly how they want to scan e.g. to circumvent port knocking (where packets must be sent to a sequence of ports in order before access to the final port is allowed). It also supports a plethora of options to avoid firewalls or Intrusion Detection Systems (IDS) such as sending packets with spoofed checksums/source addresses and sending decoy probes. Nmap can do many more things than I have listed above as is illustrated quite clearly by the fact there is an entire working on using nmap (https://nmap.org/book/)

1.5 Prospective Users (Desirable)

The prospective users of this system would be system administrators, penetration testers or network engineers. In my case my prospective users would be my school's system administrators and it would allow them to see an outsiders perspective on for example the server running the school's website page or to see if any of the programs on the servers were leaking information through banners etc. (most services send a banner with information like what protocol version they use and other information)

1.6 Data Dictionary (Desirable)

I looked this up and it seemed to be related to database management systems.

https://en.wikipedia.org/wiki/Data_dictionary

1.7 Data Flow Diagram (Desirable)

This seems to be fairly relevant and to do with how data goes through my program i.e. going from the network to my port scanner into a target object and other scanners before version detection and finally displaying to the user. Make a flowchart for this.

https://en.wikipedia.org/wiki/Data-flow_diagram

1.8 Data Sources (Desirable)

Not really sure about this.

1.9 Description of Solution Details, OOP/Mobile/Networking (Core if relevant)

To do all forms of scanning other than `Connect` scanning and version detection, custom packets are made to allow the half open (no full connection is made to the host) scanning used in TCP SYN scanning. Making custom packets is quite difficult because the endianness (the order the bytes are interpreted in:

big endian, most significant byte first, little endian, least significant byte first) affects how all the information packed into the packet is interpreted by the network switch, for example the IP address 192.168.1.58 packed in big endian form but interpreted being in little endian form comes out as 58.1.168.192 which is a completely different address and will mean the packet is not routed to the correct host. As well as the issues with byte order and the interpretation of information at different points the checksum which is embedded into the packet is calculated from a psuedo-header calculated from information in the underlying IP header and all of this has to be calculated in the right byte order (endianness).

I have used Python's multiprocessing module to allow me to spawn another process which listens for responses from hosts and waits for a certain amount of time before returning information on what hosts responded and in the case of ping scanning also metadata about how they responded.

In version detection scanning the relationship between the data sources and modules used is quite complex so I have used an Object Oriented Programming (OOP) approach to group the methods that act on the data along with the data itself. For example each probe defined in the nmap-service-probes file can be sent to a host and matched against a list of match directives stored in the probe, the probe class has a scan method which sends it's probe to the host and then automatically runs match and soft-match directives against the information returned by the probe.

Parsing the match and softmatch directives was quite difficult because they include regular expressions with special characters such as newlines and carriage returns in the form of `\n` and `\r` characters which python escapes to `\\n` and `\\r`. Which instead of matching a newline character and a carriage return will match a literal backslash and then an n or an r which is not what we want. To fix this I have to substitute newline and carriage returns back in where I find `\\r` and `\\r`.

1.10 Acceptable Limitations (Supplementary)

Originally I had planned to include dedicated operating system detection as an option however I ran out of time having implemented version detection. However it still does Operating system detection partially as some services are linux only and while doing service and version detection especially the Common Platform Enumeration (CPE) parts of the matched service/version will contain operating system information, such as microsoft ActiveSync would indicate that the system being scanned was a windows system which is reflected in the match directive and attached CPE information:

```
match activesync m|^.\0\x01\0[^\0]\0[^\0]\0[^\0]\0[^\0]\0[^\0]\0.*\0\0\0$|s
p/Microsoft ActiveSync/ o/Windows/ cpe:/a:microsoft:activesync/ cpe:/o:microsoft:windows/a
```

1.11 Data Volumes (Supplementary)

This seems to be about the volume of data stored in a database.
https://stackoverflow.com/questions/5566841/what-are-data-volumes#5567390

1.12 Test Strategy (Core)

I am going to use two different methods to test my program:

1. Unit testing
2. Wireshark

I am using two separate testing strategies because they are both good at different things, both of which I need to show that my project works. Firstly I am using unit testing to test some general purpose functions which are pure functions (are independent of the current state of the machine) such as `ip_range()` and other functions which I can just check the returned value against what it should be.

Wireshark is useful for the other half of the program which uses impure functions and the low level networking e.g. `make_tcp_packet()`. Wireshark makes this easy by allowing capture of all the packets going over the wire, as well as this it has a vast array of packet decoders (2231 in my install) which it can use to dissect almost any packet that would be on the network. The main benefit of wireshark is that I can see my scanners sending packets and then check whether the parsers that I have written for the different protocols are working. I can also check that the checksums in each of the various protocols is valid as wireshark does checksum verification for various protocols.

2 Design

2.1 Overall System Design (High Level Overview) (Core)

There are two types of scanning implemented for different scan types in my program.

- `Connect()`
- version
- listener / sender

`Connect()` scanning is the simplest in that it takes in a list of ports and simply calls the `socket.connect()` method on it and sees whether it can connect or not and the ports are marked accordingly as open or closed.

Version scanning is very similar to `Connect()` scanning in that it takes in a list of ports and connects to them, except it then sends a probe to the target to elicit a response and gain some information about the service running behind the port.

Listener / sender scanning does exactly what it says on the tin: it sets up a "listener" in another process to listen for responses from the host which the "sender" is sending packets to. It can then differentiate between open, open|filtered, filtered and closed ports based on whether it receives a packet

back and what flags (part of TCP packets are a one byte long section which store "flags" where each bit in the byte represents a different flag) are set in the received packet.

2.2 Design of User Interfaces HCI (Core)

I have designed my system to have a similar interface to the most common tool currently used: nmap this is because I believe that having a familiar interface will not only make it easier for someone who is familiar with nmap to use my tool it also makes it so that anything learnt using either tool is applicable to both which benefits everyone.

Based on this perception I have used the same option flags as nmap as well as similar help messages and an identical call signature (how the program is used on the command line). Running `./netscan.py <options> <target_spec>` is identical to `nmap <options> <target_spec>` in terms of which scan types will be run, which hosts will be scanned and which ports are scanned. Below you can see the help message generated by `./netscan.py --help`.

```
usage: netscan.py [-h] [-Pn] [-sL] [-sn] [-sS] [-sT] [-sU] [-sV] [-p
    PORTS]
                [--exclude_ports EXCLUDE_PORTS]
                target_spec

positional arguments:
  target_spec         specify what to scan, i.e. 192.168.1.0/24

optional arguments:
  -h, --help          show this help message and exit
  -Pn                 assume hosts are up
  -sL                 list targets
  -sn                 disable port scanning
  -sS                 TCP SYN scan
  -sT                 TCP connect scan
  -sU                 UDP scan
  -sV                 version scan
  -p PORTS, --ports PORTS
                      scan specified ports
  --exclude_ports EXCLUDE_PORTS
                      ports to exclude from the scan
```

It shows clearly which are required arguments and which are optional ones, as well as what each argument actually does. It also allows some some arguments to be called with either a short format e.g. `-p` and with a most verbose format `--ports` this allows the user to be clearer if they are using the tool as part of an automated script to perform scanning as it is more immediately obvious what the more verbose flags do.

2.3 Database Structure (ERD, Normalisation) (Core, if relevant)

I am fairly sure this is irrelevant to mine?

2.4 System Algorithms (Flowcharts) (Core)

When I have finished the first draft of the text bits I will add pictures / flowcharts

2.5 Input data Validation (Core)

My program takes very little input from the user which means that there is a very low chance of the program crashing due to user input error as the errors are detected All data which is entered is either parsed using a regular expression with the case of the ports directive (`-p`) or is run through checking functions like `ip_utils.is_valid_ip`. As well as using these checking functions whenever an IP address is converted between "long form" and "dot form" which is used in every type of scanning.

Algorithm 1 My algorithm for pretty-printing a dictionary of lists of port numbers such that ranges are specified as start-end instead of start,start+1,...,end

```
1: procedure COLLAPSE
2:     port_dictionary ← dictionary of lists of port numbers
3:     key_results ← empty list        ▷ stores the formatted result for each key
4:     for key in port_dictionary do
5:         ports ← port_dict[key]
6:         result ← key + ":{"
7:         if ports is empty then
8:             new_sequence ← FALSE
9:             for index ← 1, (length of ports) − 1 do
10:                port = ports[index]
11:                if index = 0 then
12:                    result ← result + ports[0]        ▷ append the first element
13:                    if ports[index+1] = port + 1 then
14:                        result ← result + "-"          ▷ begin a new sequence
15:                    else
16:                        result ← result + ","                  ▷ not a sequence
17:                else if port + 1 ≠ ports[index+1] then    ▷ break in sequence
18:                    result ← result + port + ","
19:                    new_sequence ← TRUE
20:                else if port + 1 = ports[index+1] & new_sequence then
21:                    result ← result + "-"
22:                    new_sequence ← FALSE
23:            result ← result + ports[(length of ports)-1] + "}"
24:            append result to key_results
       return "{" + (key_results separated by ", ") + "}"
```

Algorithm 2 My algorithm for turning a CIDR specified subnet into a list of actual IP addresses

```
1: procedure IP_RANGE
2:     network_bits ← number of network bits specified
3:     ip ← base IP address
4:     mask ← 0
5:     for maskbit ← (32 − network_bits), 31 do
6:         mask ← mask + 2^maskbit
7:     lower_bound ← ip AND mask              ▷ zero the last 32-network_bits
8:     upper_bound ← ip OR (mask XOR 0xFFFFFFFF)        ▷ turn the last
   32-network_bits to ones
9:     addresses ← empty list
10:    for address ← lower_bound, upper_bound do
11:        append CONVERT_TO_DOT(address) to addresses
       return addresses
```

2.6 Proposed Algorithms for complex structures (flow charts or Pseudo Code)

2.7 Design Data Dictionary (Core)

3 Technical Solution

3.1 Program Listing

3.2 Comments (Core)

3.3 Overview to direct the examiner to areas of complexity and explain design evidence

4 Testing

4.1 Test Plan

4.2 Test Table / Testing Evidence (Core: lots of screenshots)

5 Evaluation

5.1 Reflection on final outcome

5.2 Evaluation against objectives, end user feedback

5.3 Potential improvements

6 Appendices

You may show you program listing here
User feedback and survey data