PLCS CW2

Scanner: A Port Scanner Tool

Student ID: 2136685

Student ID: 2136685

Table of Contents

Table of Figures

1. Introduction

For my project, I chose to do a port scanning tool, helping users to view their open ports to allow them to distinguish which ones have been opened by them and which ones have been opened unbeknownst to the user, perhaps maliciously.

1.1. Tool Briefing

The port scanner tool works by taking in arguments directly from the the user's favourite terminal emulator. Of course, the tool is open-source, therefore it can be run by using the pre-compiled binary or the source code can be downloaded and the makefile, seen in Figure 1, can be modified to suit every user's needs. It requires no additional dependencies, except for the GCC compiler, therefore no other programs need to be installed or extra files to be procured to run the tool.

The tool uses the following arguments:

- "-s" <starting port> - this argument is followed by the starting port number.
- "-e" <ending port> - this argument is followed by the ending port number.
- "-h" - this argument displays the help menu, seen in Figure 2.

The "-s" and "-e" arguments must be set at the same time, as the tool will display an error if one argument is set without the other, as seen in Figure 3.

The tool scans all the ports in the range set by the arguments, showing the open ports as the tool is going through them, as seen in Figure 4.

1.2. Target Audience

This program is targeted at any professional user in a cyber security environment as it can help with reducing the attack surface. For example, a user can use the tool to quickly see which ports are open, then further diagnose which services are running on the ports using a regular program such as "netstat" on Linux.

Users with a cyber security as a hobby may also want to increase the security of their systems and overall network, in the same way that cyber security professionals want to protect a company's systems.

To be precise, any one user, regardless if they are a professional, a hobbyist or someone concerned for their cyber security, can use the tool. The tool doesn't require any deeper knowledge than what the user was searching for before downloading the tool. It's help menu shows how to use the tool and it can be displayed by running it using the "-h" argument or by it without an argument.

1.3. Programming Language and Paradigm

I chose the C programing language for a few reasons. Firstly, I was comfortable with the language, as I have been programing in C more than any other language as of recently. Secondly, the availability of resources. The C programming language is very mature, as it was released in 1972, therefore it has much extensive and vast help available on the Internet. Thirdly, the language is one of the most efficient languages available, in terms of run-time and energy consumed. Choosing the C programming language was not only due to comfortability, but also as a design choice. Therefore, the reason as of why the C language was chosen will be expanded in the design section (Pereira et al., 2017).

The C programming language is an imperative procedural language, which is perfectly suited for a simple program such as the port scanning tool. The procedural paradigm implies that code is ran line-by-line, which works fine for the tool, as it is not plagued by repeating code and inefficient code such as nested if-statements, and instead uses more efficient alternatives, for example switch cases, as seen in Figure 7. The paradigm and will be explored more in depth, along with alternative solutions in the design section below (GeeksForGeeks, 2018; M.V. Thanoshan, 2019).

2. Design

2.1. Coding in C

The C programming language was a design choice for a few reasons. The port scanner tool needs to scan through a large range of port ranges, i.e. 1 to 100 or 1 to 50000. Choosing a language that was compiled, such as C, instead of an interpreted language such as Python, allowed the program to be as fast as possible and efficient, as the resulting program can be optimised to the processor’s architecture. For example, the code can be compiled for a computer running an Intel processor, or an AMD processor and it will ensure the fastest possible runtime, whereas Python is an interpreted language and has to translate the code line-by-line to the target architecture, therefore slowing runtime. Moreover, the efficiency ensures the fastest result times possible for port scans, which is crucial in a time-crunching situation, but also for convenience, as everyone wants their tools to do their intended purpose as fast as possible (Brihadiswaran, 2020; IBM, 2010; Xie, 2022).

Secondly, expanding on efficiency, with the right compiler optimisations, for example “-O3”, compilation can be very fast and efficient in terms of Joules of energy used and the time it takes for the code to run, which is especially useful when running on battery power, i.e. laptops. For example, Pereira et al. outlines in their study, specifically in Table 3, that for a compilation of binary trees, the C language used a total of 39.80 Joules in a time of 1125 ms – a ratio of 0.035 – whereas Python used a total of 1793.46 Joules in a time of 45003 ms – a ratio of 0.040. Python simply takes too long, approximately 40 times longer than C, surging the energy consumed (Pereira et al., 2017).

Thirdly, researching and familiarity with headers. I could find the manual pages of headers I needed for the project, such as the “inet.h” header, seen in Figure 5 and 6, in the third section of the Linux man pages. Therefore, information was readily available and I knew where to look for it, helping me greatly when using the socket() and connect() procedures to send packets. Moreover, I learned to extrapolate the necessary information quickly from these man pages, making me confident in my ability to deliver the tool.

Unfortunately, I was unable to identify open UDP ports due to the nature of the UDP protocol, and my inability to find a solution in time. The UDP protocol, unlike TCP, does not acknowledge a connection or a non-connection, therefore leaving the status of the port as perhaps open, or perhaps closed. I would have liked to be able to implement this feature and to implement a service recognition tool, giving feedback to the user about what ports to close as the service using that port was not running, and what ports to open as a running service requires it (Cook, 2017; Mozilla, 2023).

2.2. Imperative Procedural Programming

C can hardly be called a multi-paradigm language. Perhaps it can do functions, but it lacks features that true multi-paradigm languages, such as C++ have. Doing the project in C, I had to code procedurally, but the code was simple enough to be coded in a procedural paradigm, therefore there were no major complexity issues. C has the “main” procedure, but I did use a “help_menu()” procedure to output the help menu as it was repeating if a few lines, as seen in Figure 8. Other than that, the code ran perfectly when executed line-by-line (Stroustrup, 2002).

I believe code should be as simple as possible, easy to understand and to pick up where you left off, or for someone else to pick it up and improve it, therefore, for this project, a procedural approach made sense. In retrospect, if I was going to add features for recognising the service running on the port or making connections with a protocol other than TCP, I would have chosen a language that I can do object-oriented programming – with polymorphism for each type of protocol – or functional programming.

3. Security Choices

3.1. Terminal Arguments and Validation

Using terminal arguments instead of static variables and buffers to store strings means that the buffer will be dynamic, according to how many arguments a user inputs. The size of the array argv will always be argc-1, with argc being the amount of arguments in the terminal command. This is especially useful when dealing with user inputs, as the alternative would have been using fgets or scanf, which can be vulnerable if implemented wrong (ISO/IEC, 1986).

Plenty of validation for the user inputs has been put in place to ensure that no erroneous data has been purposefully or accidentally inputted, nullifying the effect of potential attacks. For example, only numbers can be inputted after a flag such as “-s”, also port numbers can only be positive, so there is validation for that, morever the ending port must be larger than the starting port. The validation can be seen in Figure 10.

3.2. Secure Functions

In the C language, there are many functions that can be prone to attacks, if used incorrectly. For example, even a simple printf() function can be vulnerable if one codes “printf(x);” to output variable X, which stores an integer. The correct way to code this output would be “printf(“%d, x”);”. In the port scanning tool, one example of the correct implementation can be seen in the source code in Figure 9 (OWASP, 2023).

4. References

Brihadiswaran, G. (2020). *A Performance Comparison Between C, Java, and Python*. [online] Medium. Available at: https://medium.com/swlh/a-performance-comparison-between-c-java-and-python-df3890545f6d [Accessed 1 May 2023].

Cook, M. (2017). *TCP vs. UDP: What's the Difference?* [online] Lifesize. Available at: https://www.lifesize.com/blog/tcp-vs-udp/ [Accessed 1 May 2023].

GeeksForGeeks (2018). *Introduction of Programming Paradigms - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/introduction-of-programming-paradigms/ [Accessed 1 May 2023].

IBM (2010). *Compiled versus interpreted languages*. [online] www.ibm.com. Available at: https://www.ibm.com/docs/en/zos-basic-skills?topic=zos-compiled-versus-interpreted-languages [Accessed 1 May 2023].

ISO/IEC (1986). Programming languages - C. *Computer Languages*, [online] 11(1), p.13. doi:https://doi.org/10.1016/0096-0551(86)90017-2.

M.V. Thanoshan (2019). *What exactly is a programming paradigm?* [online] freeCodeCamp.org. Available at: https://www.freecodecamp.org/news/what-exactly-is-a-programming-paradigm/ [Accessed 1 May 2023].

Mozilla (2023). *TCP handshake*. [online] MDN Web Docs. Available at: https://developer.mozilla.org/en-US/docs/Glossary/TCP_handshake [Accessed 1 May 2023].

OWASP (2023). *Format String Software Attack*. [online] owasp.org. Available at: https://owasp.org/www-community/attacks/Format_string_attack [Accessed 1 May 2023].

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J.P. and Saraiva, J. (2017). Energy efficiency across programming languages: how do energy, time, and memory relate? *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering - SLE 2017*. [online] doi:https://doi.org/10.1145/3136014.3136031.

Stroustrup, B. (2002). *Multiparadigm Programming in Standard C++*. [online] Available at: https://www.cise.ufl.edu/~manuel/stroustrup/mpp.pdf [Accessed 1 May 2023].

Xie, P. (2022). *How Slow is Python Compared to C*. [online] Medium. Available at: https://peter-jp-xie.medium.com/how-slow-is-python-compared-to-c-3795071ce82a [Accessed 1 May 2023].

5. Appendix

```
CC = gcc
CFLAGS = -g -Wall
OBJFILES = source.o
TARGET = scanner

all: $(TARGET)

$(TARGET) : $(OBJFILES)
  $(CC) $(CFLAGS) -o $(TARGET) $(OBJFILES)

clean:
  rm -f $(OBJFILES) $(TARGET) *~
```

Figure 1: The makefile of the port scanner tool

```
> ./scanner -h

Usage:
 ./scanner -s <starting port> -e <ending port>

Scan a range of ports against your localhost.

Options:
 -s <starting port>     range starting port
 -e <ending port>       range ending port
 -h                     display this help
```

Figure 2: Tool running with the "-h" argument

```
> ./scanner -e 100

Please input all the port values.

Try './scanner -h' for more information.
```

Figure 3: The error when setting an argument without the other

```
> ./scanner -s 0 -e 1000

Starting a port scan with a range from 0 to 1000.

80    open
443   open
```

Figure 4: The scanner tool running

```c
#include <stdio.h> //stdin and stdout
#include <string.h> //string manipulation macros i.e. strncpy(), etc...
#include <errno.h> //for socket errors
#include <stdlib.h> //used for exit(), free(), malloc(), etc...
#include <arpa/inet.h> //to connect to ip address
#include <ctype.h> //checking types of data isint(), isdigit()
#include <sys/socket.h> //to initialise internet sockets
```

Figure 5: Header files used for the project

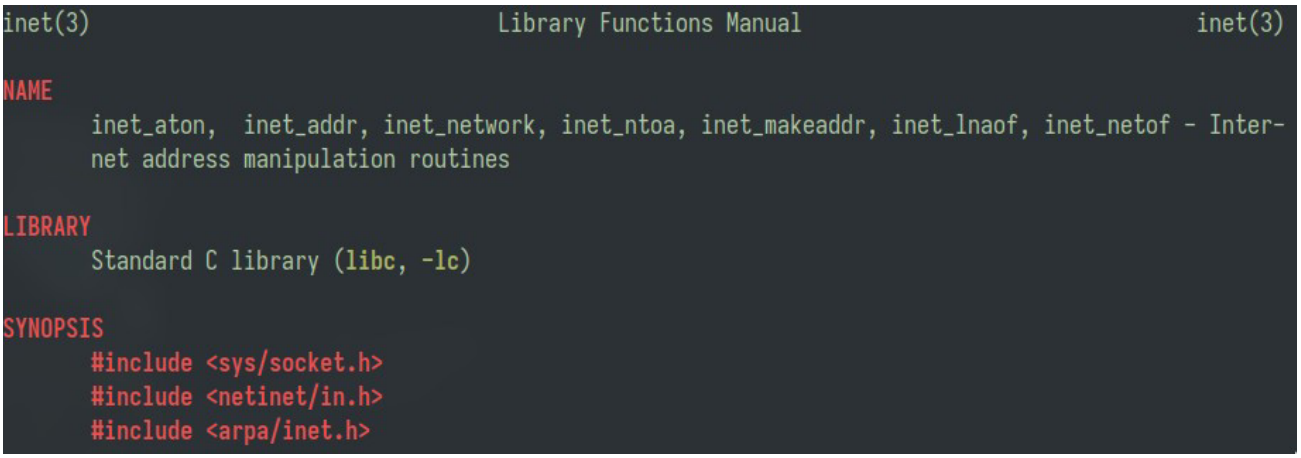

Figure 6: inet.h man page

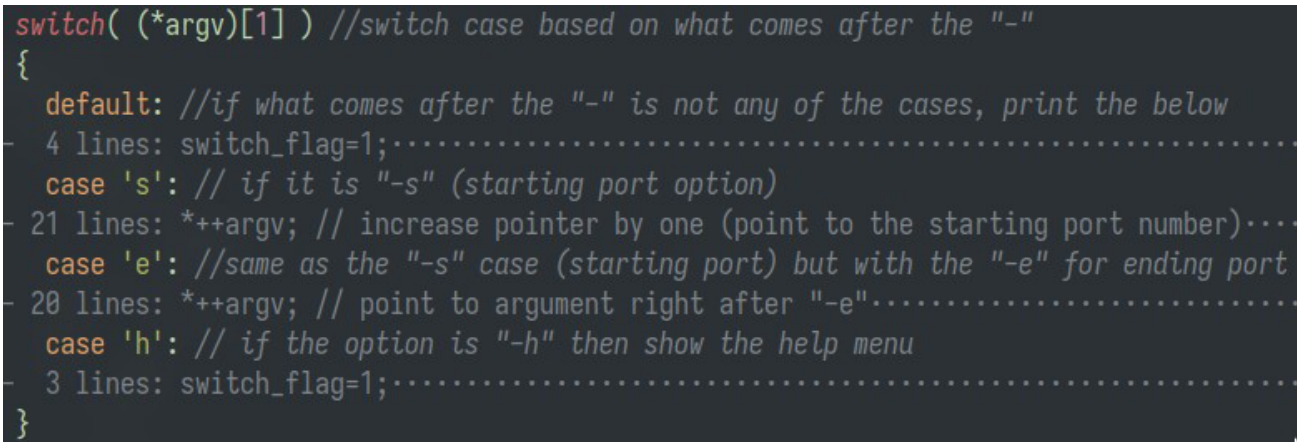

Figure 7: Switch case in source code

```
void help_menu()
{
+--  7 lines: printf("\nUsage:");·····
}

int main (int argc, char **argv)
{
+--136 lines: char hostname[100]="127.0.0.1"; //only works for local machine··
}
```

Figure 8: Procedures used in the code

```
printf("%-5d open\n", i);
```

Figure 9: Safely printing the open ports

```
    if ( (start == -1 || end == -1) && switch_flag==-1) // if start and end haven't been set
deliberately and not because someone typed "./scanner -h" or an unknown option like "-a"
    {
+----  2 lines: printf("\nPlease input all the port values.\n");·····
    }
    else if (start < -1 || end < -1)
    {
+----  2 lines: printf("\nPort values must be larger than 0.\n");·····
    }
    else if ( start>end ) //how to do this when "./scanner -s 100" to output help and when end is
smaller than start
    {
+----  2 lines: printf("\nStarting port cannot be larger than the ending port.\n");·····
    }
    else if (start >= 0 && end >= 0 && start <= end) // only start scan if start and end ports are
larger than -1 and if start is less than or equal to end
    {
+---- 30 lines: printf("\nStarting a port scan with a range from %d to %d.\n\n", start, end);·····
    }
```

Figure 10: Input sanitisation and validation