PLCS CW1

My Programming Journey

Student ID: 2136685

Table of Contents

Table of Figures

My programming journey

My first experience with programming was when I lived in a different country; I was 10 years old. My school taught children how to code programs to solve mathematical problems in C++. I wasn't very interested in this, so I laid the coding to rest. When I was 12, I loved video games, so I wanted to learn how people made games. I stumbled across a YouTube video explaining how to code a Super-Mario clone in C#, but it didn't teach me much since I copy and pasted the code.

Since then, I moved to England and started learning Python through a mobile app. Luckily for me, my new school started teaching Python too, so I got a head start. Since then, I learned more languages for my A-Level coursework, such as JS, PHP, HTML, CSS, however the most important one to learn was pseudocode, because knowing this, I could translate my code into almost any language as it taught me to understand the code I was writing.

None of my coding before university required to keep any security in mind. The most security I was taught was input sanitisation and validation for SQL injection, and storing passwords as hashes, which were useful for my A-Level coursework. However, I was never taught about security issues such as buffer overflows. This was likely because it is not a common issue with high-level languages such as Python.

Continuing, I will use Driscoll's what model to reflect on my programming journey. I used this reflective model as it simple enough to allow me to discuss the types of security mitigation methods. I adapted it by setting the context in the "what?" section. In the "so what?" section, I explained the security risk and explained the mitigation tactic. In the "now what?" section, I reflect on what I took away from learning this mitigation tactic and how I will be implementing it. (University of Nottingham, 2007)

Programming to consider cyber security

Secure C routines

What?

There are many attacks that can be done against code that bypass the intended functionality or the security practices put in place if not they are not adequate. For example, buffer overflows are a very common attack against poorly secured code. One may have a login system, however it would be useless if an attacker can use a buffer overflow attack to bypass it.

So What?

Since university, I became aware of how keeping security in mind is crucial for programming. Learning to program in C has been an experience that required me to think about security at the same time I was writing the code. Learning this taught me to be more careful using functions such as "gets()" and "scanf()" when storing user input, and instead using a more secure version, "fgets()". "gets()" and "scanf()" store an input of any size, regardless of the size and they have no checks for a buffer overflow, whereas "fgets()" takes the buffer size as a parameter and it does not allow an input more than the size specified, minus one character for the null byte at the end. (Allain, 2019; linux.die.net, 2013; linux.die.net, 2022a; Staff, 2011)

Now What?

I understand that implementing the secure versions of C routines ensures that your code cannot be used for anything else than the intended functionality and, using my previous login example, it ensures that no threat actor gets unauthorised access to an important system by bypassing the login system.

I now understand the importance of knowing the language I am coding in and how it works. Moreover, for a lower-level language like C, it is important to understand how buffers and stacks work to be able to implement the safer routines accurately.

Simple code and comments

What?

Code should be written as simple as possible, as complex code is prone to errors. Also, code should be commented as it is being written, to ensure eligibility, regardless of complexity. (Staff, 2011)

So What?

I find that after I write a bit of code I don't fully understand, when I try to go back to continue writing, it is hard to get going right away as the complexity requires me to read the code line-by-line in order to understand it again. Moreover, writing complex code that I am not sure about usually causes unwanted errors. On-the-other-hand, it is important to comment code, especially if it is complex or variable names are non-sensical, such as "i" or "j", to make the reader understand. This is especially important when working in a team. Of course, well-written code would require only a few lines of comments. (Staff, 2011)

Complex code and zero comments often come at the cost of a lot of time wasted trying to fix mistakes, however if they are overlooked and the code is implemented in a product, security issues will most definitely arise.

Now What?

I am personally guilty of never writing comments in my code and even more guilty about using non-sensical variables. After my last year's coding module, where I had to submit my code for peer-reviewing, I understood the importance of both, as no one could understand my code. Nowadays, I am better with comments and variable names, however I will still need to improve further to be able to work in a team.

Specific example of applying cyber security in my programming

Noughts and Crosses

What?

For my last year's coding module, I started coding noughts and crosses, a very popular and simple game. A player inputs a row and a column they would like their nought or cross to be in. The player wins if they have three noughts (or crosses) in a row, column or diagonally.

So What?

As seen in Figure 1, the main logic of the code is included in a while loop which is conditioned by a flag called "win", initially set to -1. The "xFlg" shows that it is X's turn, otherwise it would be O's turn. The "prGrid()" function prints the grid in its updated form, after X's move was registered and placed in the grid. After every move, it is checked if any winning conditions are met. If so, the "win" flag is set to 1 and so the game ends, as seen in Figure 2.

Selecting the row, or the column, is done by taking the input from stdin and storing it into the variable “row” or “column” respectively.

This code is not secure, due to the reasons I also explained in the section above. It uses unsecure functions such as “scanf()” and “strcmp()” which disregard the buffer size and they are vulnerable to a buffer overflow attack, allowing a player to bypass playing and win directly by skipping to a winning condition or overwriting the register holding the “win” flag to 1. (linux.die.net, 2022a; linux.die.net, 2022b)

The lack of comments makes the code less eligible, even though variables are named somewhat appropriately. A person trying to read the code will still have to understand the logic line-by-line, perhaps misunderstanding some lines and continuing to code a vulnerable piece of code.

Now What?

In the future, I should prioritise writing comments as much as writing the code. Writing good and simple code should be a priority instead of writing complex and hard to understand code. These reflections would help greatly when trying to come back to continue coding and if a team member needs to work on the code. Moreover, it ensures that if the code is to be one for a product, it will ensure that the product delivered is secure and of a higher standard, showing professionalism and efficiency.

Incorporating cyber security in my future programming activities

In the future, I want to understand all the attack vectors against a coding language before I can confidently program a system. Lower level programming languages such as C require a great deal of knowledge about memory and I currently cannot confidently say I have that knowledge. In some languages, such as JavaScript, it is common to obfuscate the code after it has been developed to make the code harder to understand, therefore making it harder to find vulnerabilities. (Snyk, 2022a)

Usually, after I program, I don’t fully test my code with all types of data – normal, boundary and erroneous data. This can be detrimental, as vulnerabilities can be overlooked if the code is not tested thoroughly. Moreover, I understand that it is important to apply debugging techniques such as breakpoints and watch windows to see the state of variables during the runtime of the code. For languages like C, using the GNU debugger is crucial to see the state of the registers and to test against buffer overflows, which I do, but not nearly enough.

In the future, when I will be developing software, I will keep in mind secure software development lifecycle (SDLC) models such as the agile method, a modern SDLC used today, but also while implementing DevSecOps. The SDLC establishes baseline requirements of the program early in the process, therefore understanding the security that needs to be put in place. DevSecOps and “shifting left” establish that security testing should start early in the development process, as testing later in the development process is more difficult and security issues are often overlooked. With each prototype, called a “sprint”, the cycle restarts, establishing new security guidelines and adding those to each prototype before, until creating a secure finished product. (Firesmith, 2015; Snyk, 2022b)

I usually code procedurally, however moving forward I want to use programming paradigms such as object-oriented programming (OOP). OOP can make variables private, therefore abstracting the data. This can make accessing and altering them impossible without getters and setters, which are tightly defined to not allow any unwanted alterations. (Gautam, 2022)

References

Allain, A. (2019). *Writing Secure Code in C - Cprogramming.com*. [online] www.cprogramming.com. Available at: https://www.cprogramming.com/tutorial/secure.html [Accessed 9 Feb. 2023].

Firesmith, D. (2015). *Four Types of Shift Left Testing*. [online] web.archive.org. Available at: https://web.archive.org/web/20150905082941/https://insights.sei.cmu.edu/sei_blog/2015/03/four-types-of-shift-left-testing.html [Accessed 9 Feb. 2023].

Gautam, S. (2022). *Abstraction in Object Oriented Programming (OOPS)*. [online] www.enjoyalgorithms.com. Available at: https://www.enjoyalgorithms.com/blog/abstraction-in-oops [Accessed 9 Feb. 2023].

linux.die.net (2013). *fgets(3) - Linux man page*. [online] linux.die.net. Available at: https://linux.die.net/man/3/fgets [Accessed 7 Feb. 2023].

linux.die.net (2022a). *scanf(3): input format conversion - Linux man page*. [online] linux.die.net. Available at: https://linux.die.net/man/3/scanf [Accessed 9 Feb. 2023].

linux.die.net (2022b). *strcpy(3): copy string - Linux man page*. [online] linux.die.net. Available at: https://linux.die.net/man/3/strcpy [Accessed 9 Feb. 2023].

Snyk (2022a). *Secure Coding Practices | What is secure coding? | Snyk*. [online] snyk.io. Available at: https://snyk.io/learn/secure-coding-practices/ [Accessed 9 Feb. 2023].

Snyk (2022b). *Secure SDLC | Secure Software Development Life Cycle | Snyk*. [online] snyk.io. Available at: https://snyk.io/learn/secure-sdlc/ [Accessed 9 Feb. 2023].

Staff, E. (2011). *Seventeen steps to safer C code*. [online] Embedded.com. Available at: https://www.embedded.com/seventeen-steps-to-safer-c-code/ [Accessed 9 Feb. 2023].

University of Nottingham (2007). *Reflection Model 2 -Driscoll (2007)*. [online] Available at: https://www.nottingham.ac.uk/library/documents/academicskillstoolkit/reflective-writing-model-driscoll.pdf [Accessed 8 Feb. 2023].

Appendix

```c
int win = -1;
while(win == -1)
{
    int xFlg = 1;
    while (xFlg==1)
    {
        printf("Enter the row: ");
        int row;
        scanf("%i",&row);

        int col;
        printf("\nEnter the column: ");
        scanf("%i",&col);

        if (strcmp(grid[row][col]," ")==0)
        {
            grid[row][col]="\033[0;31mX\033[0m";
            xFlg=-1;
            prGrid(grid);
        }
    }
```

Figure 1 - Selecting the position of the symbol

```c
if ((grid[0][0]=="\033[0;31mX\033[0m") && (grid[0][1]=="\033[0;31mX\033[0m") && (grid[0][2]=="\033[0;31mX\033[0m")){
    win=1;
}
if ((grid[1][0]=="\033[0;31mX\033[0m") && (grid[1][1]=="\033[0;31mX\033[0m") && (grid[1][2]=="\033[0;31mX\033[0m")){
            win=1;
        }
if ((grid[2][0]=="\033[0;31mX\033[0m") && (grid[2][1]=="\033[0;31mX\033[0m") && (grid[2][2]=="\033[0;31mX\033[0m")){
            win=1;
        }
if ((grid[0][0]=="\033[0;31mX\033[0m") && (grid[1][0]=="\033[0;31mX\033[0m") && (grid[2][0]=="\033[0;31mX\033[0m")){
            win=1;
        }
if ((grid[0][1]=="\033[0;31mX\033[0m") && (grid[2][1]=="\033[0;31mX\033[0m") && (grid[3][1]=="\033[0;31mX\033[0m")){
            win=1;
        }
if ((grid[0][2]=="\033[0;31mX\033[0m") && (grid[1][2]=="\033[0;31mX\033[0m") && (grid[2][2]=="\033[0;31mX\033[0m")){
            win=1;
        }
if ((grid[0][0]=="\033[0;31mX\033[0m") && (grid[1][1]=="\033[0;31mX\033[0m") && (grid[2][2]=="\033[0;31mX\033[0m")){
            win=1;
        }
if ((grid[0][2]=="\033[0;31mX\033[0m") && (grid[1][1]=="\033[0;31mX\033[0m") && (grid[2][0]=="\033[0;31mX\033[0m")){
            win=1;
        }
```

Figure 2 - Win conditions