Secure Software Development in C Language *

Shivani Gowda KS
Loyola Marymount University
sks@lion.lmu.edu

Abstract

In this paper we will be discussing the importance of secure software development in C. The best approaches to be followed in secure development, and how bad practices will lead to vulnerabilities.

1 Introduction

Software security is a top concern today, we can't risk any security vulnerabilities, Which means, our code needs to be secure and free from every type of coding errors.

When you think about software security, youprobably think about passwords and access control. Or viruses, spoofing, and phishing attacks.These are common security concerns. And security features, such data encryption and authentication protocols, mitigate these vulnerabilities. But even if you have implemented these security features, software can remain vulnerable. To ensure secure software, you need to start at the source—the code level. Otherwise, coding errors will compromise your program.

C is a very popular systems programming language. The main features of C language include low-level access to memory, a simple set of keywords, and a clean style, these features make C language suitable for system programmings like an operating system or compiler development.

2 Secure practices

Let us look into some of the most comman mistakes programmers tend to do while coding in C, how vulnerable it can be and how to mitigate it.

2.1 Format Strings Attack

At its most basic, printf() can be used to simply send an ASCII string to standard output (stdout), but its real strength lies in the use of formatting parameters. To insert values into an output string, you use format specifiers as placeholders in the string and pass the values as additional parameters to the printf function. This Format string functions in C accept a variable number of arguments, so you can also use them to print Hello World without any format specifiers.

As long as user _input is guaranteed to contain no format specifiers, this is fine. But if that value is controlled by the user, an attacker can exploit format string syntax to trigger a variety of dangerous behaviors. Let us start with a string that contains no text but lots of format specifiers, in this case code:1 %x for hexadecimal values. In C programs, variables are stored on the stack in process memory, so when printf() sees the first %x specifier, it just looks at the stack and reads the first variable after the format string. This is repeated for all four %x specifiers, so the example above will print the hex representation of four values from the stack. Depending on the program and the execution context, these could include function return addresses, variable values, pointer memory addresses, function parameters, or even user-supplied data.

*This report has been prepared for Secure Software Development classes homework By Prof. Ray Toal, 2022

```
printf("%x%x%x%x");
```

Listing 1: Non compliant code, printf() with hexadecimal value

This issue can be resolved by using puts() or formated strings2.

```
char* greeting = "Hello";
printf(greeting); // This is insecure
printf("%s", greeting); // This is secure
puts(greeting); // This is also secure
```

Listing 2: Compliant code

2.2 gets()

This function is a part of the stdio.h library of the C language. It does not check for buffer size and when provided with malicious input, it can easily cause a buffer overflow. In the above code, the attacker can give large chunk of data as input, and the gets() function will try to store all that data into the buffer without considering the buffer size which will eventually cause a buffer overflow situation that can further cause arbitrary code execution, information leak.

To solve this issue with the gets() function, programmers need to use fgets() function code:3. It limits the input length based on the buffer size. Below is the C program to demonstrate the above concept-

```
    char buf[MAX];
    printf("Please enter your name and press <Enter>\n");
    fgets(buf, MAX, stdin);  //secure
```

Listing 3: Compliant Code

2.3 strcpy()

This built-in function also doesn't check for the buffer length and can overwrite memory locations based on the malicious input. If the buffer size of destination string is more then source string, then copying is done from the source string to destination string with terminating NULL character. But if destination buffer is less, then source will copy the content without terminating NULL character. The strings may not overlap, but the destination string is large enough to receive the copy.

Use of strncpy() function instead of strcpy(). strncpy() limits the length of input based on the buffer size available. Below is the C program to demonstrate the above concept code:4

```
    char str1[2];
    char str2[] = "LoyolaMarymountUniversity";
    strncpy(str1, str2);
    printf("Copied string is: %s\n", str1);
```

Listing 4: Compliant Code

2.4 NULL Pointer Dereference

If a program dereference a pointer thinking that it is expected to be valid but turns out as NULL then it causes program crash, exit. Mitigation to this is to check if the pointer is NULL before performing any operation. Below is the C program to demonstrate the above concept code:5

```
    int val = 8;
    int* p = NULL;
    if (p == NULL) {
        puts("Pointer is NULL");
    }else {
        *p = val;
        printf("%d", *p);
    }
```

Listing 5: Compliant Code

2.5 Out of Bounds Write

In this, the software writes the data before or after the intended buffer. In the below code, array has valid indexes from 0-9 but there is an attempt to write the value on the arr[10]. As the C compiler will not check for the array bound it will write that data after the intended buffer. But when there is an attempt to print the value at index 10, it will show an error. It may cause execution of unauthorized codes, Crash and restart. Below is the C program to demonstrate the above concept code:6

```
    int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    printf("arr [0] is %d\n", arr[0]);
    printf("arr[10] is %d\n", arr[10]);
    arr[10] = 11; // allocation memory to out of bound element
    printf("arr[10] is %d\n", arr[10]);
```

Listing 6: Non-compliant code

Mitigation of this issue is to always make sure the buffer is large enough and functions used to accept input should have a buffer limit implementation.

2.6 Out of Bounds Read

In this, the software reads the data before or after the intended buffer. Attackers may use this to read sensitive information from other memory locations. Below is the C program to demonstrate the above concept code:7. Mitigation of this issue is to validate the input,and ensure to validate correct calculations for length of an argument and buffer size.

2.7 "memset" should not be used to delete sensitive data

We can carefully destroy the password so it can't be found accidentally. But Unfortunately, the compiler is not allowed to excecute memset() call as the password is not used again code:8.

Mitigation of this issue can be done by using memsets function instead of memeset. The rule for memsets() is that the call cannot be omitted; the password variable must be zeroed, regardless of optimization code:9. These are some of comman mistakes while coding in C that could lead to security issues, if you want to know and learn more, you can find some good resources in the reference page.

```
int getValueFromArray(int* arr,int size, int index)
{
    int value = -1;
      if (index < size) {    // only maximum limit is there
        value = arr[index];
    }
    return value;
}
```

Listing 7: Non-compliant code

```
    char password[128];
    if (fgets(password, sizeof(password), stdin) != 0)
    {
        password[strcspn(password), "\n\r"]) = '\0';
        validate_password(password);
        memset(password, '\0', sizeof(password));
    }
}
```

Listing 8: Non-compliant code

```
    memset_s(password, sizeof(password), '\0', sizeof(password))
```

Listing 9: Compliant code

3 Summary

The below mentioned table gives a brief summary of all the security issues that we discussed above in C with do's and don't.

Type of attacks	do's	don't
Format string attack if its a plain string	Use printf("%s", greet); or puts(greet);	printf(greet)
Input from user	fgets(buf, Max, stdin)	gets(buff)
String copying	strncpy(str1, str2)	strcpy(str1, str2)
NULL Pointer Dereference	validate the pointer, deference it only if it is not NULL	Do not dereference it directly
Out of bounds write	Check for the bounds before implementing it	Using it before chceking the bound
out of bounds read	Check for the bounds before implementing it	Using it before chceking the bound
deleting the sensitive data	use memset_s()	using memset()

4 Secure coding

Let us look into some of the important things to keep in mind when dealing one step above code level.

1. Immutability: Implement immutability as much as possible so that sharing data is never an issue.
2. Validations: Validate the input, callers and the state
3. Secure error handling: fail fast, fail loudly and if in the middle of something roll back.
4. Sanitize: Do not sanitize the code on your own, get a library that's been tested to death.
5. Reducing complexity: Complexity is an enemy of security. If you have many moving parts, many interacting subsystems, many computation paths through a function (too many ifs and switches and loops), really clever dense code (that you don't understand immediately), instances of duplicate code, functions with way too many parameters, or you accept massive or intricate inputs, you have more ways for something to go wrong, not to mention a bigger attack surface. Readability and understandability and maintainability are crucial if you can't read, understand, or maintain your code, you cannot argue it is secure.
6. Clean code: Keep your code simple, delete unwanted comments, remove dead code, and avoid redundant code.

5 References

1. Introduction to C by Prof. Ray Toal: `https://cs.lmu.edu/~ray/notes/c/`
2. Software security by Ray Toal :`https://cs.lmu.edu/~ray/notes/softwaresecurity/`
3. Format string vulnerabilities: `https://www.invicti.com/blog/web-security/format-string-vulnerabilities/`
4. C static code analysis: `https://rules.sonarsource.com/c/type/Vulnerability/RSPEC-5798`
5. OWASP Secure Coding Practices Quick Reference Guide: `https://owasp.org/www-pdf-archive/OWASP_SCP_Quick_Reference_Guide_v2.pdf`
6. CERT C coding standards: `https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard`
7. Secure coding by Prof. Ray Toal `https://cs.lmu.edu/~ray/notes/securecodingconstructs/`
8. Security issues in C: `https://www.geeksforgeeks.org/security-issues-in-c-language/`