Secure Programming using C
UE19CS257C

Team members:

Lakshmi Narayan P
Geethika K
Sneha A

Objective

To implement a simple game of Cricket using a C programming language.

Problem Definition

To implement a simple game of Cricket using a C programming language. The User will be the team manager; therefore, the game starts off by asking the user to enter a username. To pick the team, 4 files will be created for batsmen, wicketkeeper, all-rounders and bowlers and for every 4,1,2,4 players will be randomly selected respectively.
The rules of the game are that of hand cricket.
Toss - User can choose either odd or even and the system generates a random value depending on whether it's odd or even the user gets to choose to bat or ball.
Batting/Bowling - The user can choose a number from 1 to 6 and the system also generates a randomized value; if values do not match - the batsman is safe and the score is added to the scoreboard. If the value matches, then the batsman is considered out, and the next batsman comes in.

Need for Secure programming

“Secure coding is the practice of developing computer software in a way that guards against the accidental introduction of security vulnerabilities. Defects, bugs and logic flaws are consistently the primary cause of commonly exploited software vulnerabilities.” But what does this mean? Putting it in layman terms, secure programming is all about finding security risks and coming up with recommendations to tackle them.

Software vulnerabilities are, unfortunately, an ever-present risk. For that reason, we must ensure that our code is secure and protected. Insecure software can lead hackers in and this can result in compromised secrets, loss of service and damage to thousands of users.

C is the language with the highest number of reported vulnerabilities of the bunch, by far. Vulnerabilities in C account for over 50% of all reported open-source vulnerabilities since 2019, and although the number of vulnerabilities rises and fall to some extent over those years, the amount of C vulnerabilities each year far surpasses the rest of the programming languages. Because of this, it’s even more pressing to understand secure programming and implement it.

Software Requirement Specification

1: C Programming Language

As a middle-level language, C combines the features of both high-level and low-level languages. C implements algorithms and data structures swiftly, facilitating faster computations in programs. But doing so, C opens a wide range of vulnerabilities to deal with. As a developer, it gets very challenging to tackle and develop safe programs.

2: Compiler - GCC

3: Static analysis tool - Splint and flaw finder

Security Requirement specification

Safety features. (data type used, functions used)
Data types - Arrays and Linked Lists

Functions -
List the safety recommendation required for your software (CERT rules and recommendations)
Input Validation (includes character & integer)
Buffer Errors
We need to first disable the protection mechanism.

We perform two things –

a. disabling the address space randomization and b) disabling compiler protection.

This will allow us to overwrite into the execution stack and perform unbounded string copies without throwing any compiler warnings or errors hence it allows us to overflow the buffer. We then start the debugger and run the program using ‘run’. We are looking for buffer overflow, hence we add breakpoints (b) before and after the string copy function. We then run the program with a command line argument (here, “ABCD”). As expected, it breaks at the first breakpoint before copying into memory. At this point, memory has been allocated for the buffer array. We then display the contents of buff using ‘x/128bx’. We see that the memory has been allocated and contains random content. We then proceed with the execution.

Resource management vulnerability (files opening and closing) & Safety Measures

Checking return value of malloc (Used for player):
If we try to assign the value of n to a huge value (LONG_MAX). This size cannot be allocated by malloc and hence it returns a NULL pointer. As we can see, it crashes when we try to access piBuffer.

Prevention: we first check if piBuffer points to NULL. If it does not, we continue execution, else, we display that there is no memory that has been allocated and we terminate the program.

Initializing errors (In calculating the score):
The piBuffer array was not initialized before its values were accessed. It is possible that it can hold junk values. However, due to different specific implementation-based behaviors, it was initialized with zeroes.

Prevention: We add the memset function to explicitly initialize all values to 0. Now the piBuffer array will hold only 0's initially, and will hold the same values as the original array a[ ].

Integer vulnerability (When the user enters a much bigger number):
These types of vulnerabilities are created by misuse of variable types and can be exploited to bypass protections against other types of vulnerabilities, like buffer overflows. Data types include storing signed int versus unsigned int. Integer overflow vulnerabilities are caused when a value is moved into a variable type too small to hold it. Integer overflows occur when a value exceeds the maximum value that a variable can contain, and integer underflows happen when a value becomes too small to fit.

Prevention: In integer overflow it is accomplished by cutting the value down to a small enough size that it fits in the smaller value. Integer overflow and underflow vulnerabilities are caused by misuse of variable type conversions. Protecting against these vulnerabilities is fairly simple: use a language with dynamic variable typing (like Python) or make sure that variable types are explicitly specified and the correct type for the job.

Dataflow Diagram