CWE-20: Improper Input Validation

Improper input validation occurs when the Vulnerable program/system does not validate the received inputs . This issue can arise in reference to a variety of features of the input such as size, data type, range, structure (whitespaces, new lines etc) . If the program receives an altered input format which it does not expect, it may misbehave leading to arbitrary control flow of the program . This weakness can be exploited by the attacker to alter the basic functionality of the software and generate custom outputs.
Input validation is a technique to ensure that the program identifies and discards/processes potentially dangerous inputs into something valid.
The most common example of careless input validation is vulnerability of various fields of data and cookies on a website . The developer may leave these fields in modifiable form making it susceptible to attacks.

The attack model is built for the implementation phase using basic input validation strategy . When performing input validation, we consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields
There are no such requirements to run the exploit , we only need a gcc compiler and basic c++ libraries for i/o and string handling .

To run the exploit :
script.sh : the 3 test case files t1,t2,t3 contain possibly dangerous input types . This script passes them into the vulnerable.cpp program and stores the absurd outputs generated in o1,o2,o3 which are different from the expected output of the code.
mitigated.cpp : runs on test.txt which contains all possible input types (and runs correctly) to store outputs in output.txt.

How to use : Run the following script in the working directory consisting of all the files and test folder :
./script.sh
g++ mitigated.cpp
./a.out > output.txt

The vulnerable program takes input integer values sequentially . The expected values are integers ranging from 0 to INT_MAX . Thus, the possibly dangerous input types include : negative integers, fractions, integers greater than INT_MAX , characters, strings, and whitespaces, newlines etc. The program expects the inputs in the form of integer bids for a product . Thus the inputs are summed up sequentially till they reach a value beyond 200 or the user inputs the number 0 to exit the bidding .

- The vulnerable program does not check for the sign of integer input, so inputting a negative integer led to a reduction in the current bid which is not ideal.
- The program should not accept fraction values for the bid but it strips the fraction part of input and works with the remaining value. This is because when we “cin” to int type variable , only the integer part is stored and the rest remains in the cin buffer (which will be inputted the next time cin is called on an int making that new integer presumably 0).
- For character/string inputs , cin>>(int) assigns a 0 value to the variable, thus exiting the bidding war when in reality , the input could just be a typo from users' side. Ideally the program should stop only when the user inputs 0 or the bid crosses 200 mark.

- For inputs>INT_MAX, cin assigns INT_MAX to the the variable and the rest of the value remains in the buffer leading to an infinite loop (which is though terminated as the bid crosses 200) .
  All these exploits are a result of the vulnerability of cin to bad inputs . we can mitigate this weakness by using stringstream to receive the inputs instead.

    - For character/string inputs and the integer inputs > INT_MAX , (stringstream >> int_var) returns a 0 boolean value . Hence , the code is modified to execute only when (stringstream >> int_var) returns 1 , else ask for new valid input.
    - For inputs with a fraction part, ss.str() is used to get the string form of these inputs (of the form n.xyz) which is different from the string version of the input int_var (stores only n, the integer part) and so when this comparison on [ss.str() and to_string(int_var)] holds false , the program is not executed further and new valid input is asked from user.
    - For negative inputs, we simply introduce a logical condition on the input variable which ensures that the code does not run for int_var < 0 .
    - For whitespaces, the input string obtained through getline() function call is stripped off from whitespaces at the beginning and the end. The getline function also ensures that the whole line from the input stream is taken at once and then only the first integer is used as the bid. For example , the ‘Vulnerable’ program treats the input :
      1 2
      3
      as a sequence of bids “1 , 2 , 3” . However , the user cannot make 2 bids at one go in a line . The mitigated program treats the same input as two bids of “1” and “ 3”. This is again done using stringstream as follows : getline() stores the string “1 2” in str variable and stringstream(str) is used to split “1” and “2” . In one loop of the program, ss >> int_var call is made once so int_var gets only the integer 1 and then the getline() moves to the next line.