Secure Programming in C/C++

With the rapid pace of technological advancement, daily life is now intimately connected to the Internet. Critical portions of business operations, financial systems, manufacturing supply chains and military systems are also networked. Indeed, while it was originally computers and their users that communicated over networks, we now have commonplace objects (such as thermostats, light bulbs and kitchen appliances) that communicate with each other and with us. This trend is accelerating, leading to what is called the "Internet of Things." - from A report on the NSF-sponsored Cybersecurity Ideas Lab held in Arlington, Virginia on February 10-12, 2014

"In one of the most serious cyber incidents to date against our military networks, several thousand computers were infected last year by malicious software – malware. And while no sensitive information was compromised, our troops and defense personnel had to give up those external memory devices – thumb drives – changing the way they used their computers every day.

And last year we had a glimpse of the future face of war. As Russian tanks rolled into Georgia, cyber attacks crippled Georgian government websites. The terrorists that sowed so much death and destruction in Mumbai relied not only on guns and grenades but also on GPS and phones using voice-over-the-Internet.

For all these reasons, it's now clear this cyber threat is one of the most serious economic and national security challenges we face as a nation." - President Barack Obama, REMARKS BY THE PRESIDENT ON SECURING OUR NATION'S CYBER INFRASTRUCTURE, May 29, 2009

1 Introduction

It is every programmer's responsibility to write secure code. Thirty or forty years ago no one thought very seriously about this, because the world of programmers was small and filled mostly with trustworthy people, and there was no World Wide Web. This is no longer true. Hackers abound and people with malevolent intentions are out there trying to exploit vulnerabilities in code all of the time. As a programmer, you must ensure that your code cannot be used to help someone break into a system or gain access to data to which they have not been given access rights.

This small set of notes is intended to make you aware of the major problems and to give you suggestions on how to write code more securely.

The C/C++ programming languages have been the most important higher programming languages for years, mostly because of the very things that make them susceptible to attack –

- they allow close interoperability with the operating system (which itself is usually largely written in C).
- they provide direct access to memory manipulating functions (using pointers to memory locations).

These two features, when not used properly, make programs susceptible to a wide variety of attacks. There are other features that are susceptible to attacks as well. What are they? What can you do to reduce the risk of attacks? The purpose of these notes is to get you thinking about these issues, and to a limited extent, to

- make you more aware of security when you are developing code
- improve your ability to apply security principles,
- increase your skills as programmers in writing secure code.

2 Common Vulnerabilities

Three of the most common types of software vulnerabilities are

- integer errors
- input validation errors
- buffer overflows

The following exercise is a good place to start.

Exercise 1. Consider the following code[1]:

```
#include <iostream>
using namespace std;
int main()
{
    int length;
    int volume;
    int factorial[10] = {1,2,6,24,120,720,5040,40320,362880,3628800};
    int index;

    cout << "Enter the length of the cube as a whole number: ";
    cin  >> length;
    volume =  length * length * length;
    cout << "The volume is " << volume << endl;

    cout << "Which factorial do you want? Enter its index: " ;
    cin  >> index;
    cout << "Factorial[" << index << "] is " << factorial[index] << endl;
    return 0;
}
```

Which lines of code are safe? Which might be vulnerable? What changes must be made to make this code safe and secure? Can you identify sources of integer errors, input validation errors, and/or buffer overflows?

2.1 Integer Errors

In any operating system and on any machine, and integer is a fixed number of bytes. What happens when a number larger than that is stored into the memory allocated for it depends on the compiler, the operating system, and the hardware. It is pretty unpredictable. That is why the code has to make sure that this does not happen. It might cause a program crash, or it might not. It might just silently corrupt other data. Integer errors also include mistakes related to conversions between integer types, such as from unsigned to signed or vice versa.

The task of the programmer is to be aware of the various types of integer errors and to write code that prevents them from occurring. Did you know that you can always find the value of the largest positive integer on the machine, or the smallest negative one? Is there a way to use this to check if overflow will occur before performing the operations?

Activity: Visit the following site, read the material and answer the short questions there: Security Injection for CS2: Integer Errors

[1] This exercise is based on one used by Taylor and Kaza in "Security Injections@Towson:Integrating Secure Coding into Introductory Computer Science Courses, ACM Transactions on Computing Education, Vol. 16, No. 4, Article 16, June 2016.

2.2 Input Validation

Input comes from various sources and is stored into variables of various types. In the exercise above, the user was asked to enter a number. What would this program do if the user entered something other than a whole number, such as a number with a fractional part, or something that was not even a number?

Exercise 2. Consider this example:

```
#include <iostream>
using namespace std;
int main()
{
    int factorial[10] = {1,2,6,24,120,720,5040,40320,362880,3628800};
    int index;

    cout << "Enter the index of the factorial that you want to display: " ;
    cin  >> index;
    cout << "Factorial[" << index << "] is " << factorial[index] << endl;
    return 0;
}
```

What are the different ways in which the input can be "bad"? What can go "wrong"? How can you fix it?

Activity: Visit the following site, read the material, and answer the questions there: Security Injection for CS2: Input Validation

2.3 Buffer Overflows

A buffer overflow attack is one in which the perpetrator purposely writes more into a buffer than its size allows, designing the part that overflows in such a way that it can be used to run the perpetrator's malicious code, or to obtain information about the computer system that was supposed to be protected. There are many different types of buffer overflow attacks, and this small set of notes is not intended to describe how they work, but to provide guidance that will aid in your preventing them. Some buffer overflows are not malicious, but result from the program code's failing to prevent some storage from being overflowed.

Exercise 3. Consider this program:

```
#include <iostream>
using namespace std;
int main()
{
    int mydata = 1;
    int mydata2 = 2;
    int buffer[10];
    cout << "Before buffer assignment:" << endl;
    cout << "mydata = " << mydata << " and mydata2 = " << mydata2 << endl;
    for ( int i = 0; i < 15; i++ ) {
        buffer[i] = 4;
    }
    cout << "After buffer assignment:" << endl;
    cout << "mydata = " << mydata << " and mydata2 = " << mydata2 << endl;
    return 0;
}
```

Predict the output. Do not run it until you can reason through what happens. Run it and try to figure out why the output is what you see. Make sure that you compile it with the *g++* flag `-fno-stack-protector` just in case on your platform g++ has been installed to default to `-fstack-protector`.

Activity: Visit the following site, read the material, and answer the questions there: Security Injection for CS2: Buffer Overflows

Exercise 4. Figure out what this program does:

```
#pragma check_stack(off)
#include <string.h>
#include <stdio.h>
void foo(const char* input)
{
    char buf[10];
    printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
    strcpy(buf, input);
    printf("%s\n", buf);
    printf("Now the stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
}
void bar(void)
{
    printf("Augh! I've been hacked!\n");
}
int main(int argc, char* argv[])
{
    //Blatant cheating to make life easier on myself
    printf("Address of foo = %p\n", foo);
    printf("Address of bar = %p\n", bar);
    if (argc != 2)
    {
        printf("Please supply a string as an argument!\n");
        return -1;
}
    foo(argv[1]);
    return 0;
}
```

2.4 Design Issues: Encapsulation

Activity: Encapsulation Exercise

2.5 Exception Handling

Activity: Exception Handling Exercise

3 Rules to Observe When Coding

- Always validate input to public methods. Do not use the values of any data passed to a public method of a class, or to any function for that matter, unless you first validate that it is within the specified range of values.

  - In particular, always check the length of string buffer arguments.
  - Check the characters in an input string against a list of allowable characters (*white list checking*).
  - Make sure that numbers are within specified ranges.

- Many people use the C functions found in `<cstdio>` because they can be more convenient and/or useful than those from C++. Some of these are dangerous. Never use `gets()` to read from standard input. Use `fgets()` instead.

  - While `fgets()` accepts a parameter to limit the number of bytes to read, `gets()` provides no such possibility for limiting the input length. As `gets()` does not check the length of an input string before copying the string to memory, a buffer overflow attack can always be successfully mounted when `gets()` is used.

- Avoid the use of environment variables.

  - Environment variables, whether in UNIX or Windows, are strings, and they can be manipulated to be too long. Programs should not use them directly. Just recently a major security flaw was discovered in the Bash shell related to the use of these environment variables. One can create environment variables with specially-crafted values before calling the Bash shell. These variables can contain code, which gets executed as soon as the shell is invoked. The name of these crafted variables does not matter, only their contents.

- Do not call a shell to invoke another program from within a C/C++ program.

  - C and C++ programs can use the UNIX `system()` and `popen()` functions to execute commands. This is very dangerous and should never be done.
  - Instead, to call another program from within a process use `execve()` or `execle()`.

- Let the compiler detect as much as possible by enabling all compiler warnings and pay attention to these warnings. When using `gcc` or `g++`, you should use the flag `-Wall` to turn on all warnings.

- Declare variables and methods to be private whenever possible. It is safe programming practice to limit access as much as possible; everything should be declared to be private by default. This is an application of the ***Principle of Least Privilege***, which is defined as the practice of limiting access to the minimal level that will allow normal functioning.

- Make objects ***immutable*** whenever possible. An object is immutable of its state cannot be changed after it is created. Examples in C/C++ are those variables and members declared `const` in a program.

- Avoid returning references to mutable member objects from public methods.

- Avoid the usage of `strcpy()` and `strcat()`, use `strncpy()` and `strncat()` instead. The former two functions are susceptible to buffer overflow attacks, whereas the latter require a length parameter that prevents such attacks.

- Calls to one of the formatting functions `sprintf()`, `vsprintf()`, `scanf()`, `sscanf()`, `fscanf()`, `vscanf()`, `vsscanf()` or `vfscanf()` may have string arguments in its format string. Most implementations of such functions allow the specification of the maximal length of such string arguments as in the following example (here at most 100 bytes are copied from the associated variable `argv[0]`):

  ```
  sprintf( buffer, "Usage: %.100s argument\n", argv[0] );
  ```

  If the length of such a string argument is not restricted by a precision specifier such calls are a potential source for buffer overflows. If the used implementation of the formatting functions does not allow the usage of precision specifiers for string arguments, all variable string arguments have to be checked for length before such a formatting function is invoked.

- Avoid file system calls that take a filename for an input argument and prefer those calls that take file handles or file descriptors. For example, the `open()` member function of the `fstream` class takes a C string file name argument. If this is used, the length of the file name should be checked first.