Parameter Passing and the Call Stack

IA Concept: Buffer Overflow

Introduction

A comparison is often drawn between writing a good program and writing a good essay. While they differ in some distinct ways (i.e. writing more lines of code does not always get you closer to being done), they are also similar in many ways. For instance, you can take all of your thoughts and force them into one giant five-page paragraph, but this will certainly not produce a coherent essay. Although it is possible that all of your ideas will be expressed, they will not be expressed as cleanly as they would have been if they were broken up. Furthermore anyone reading this essay will be lost in a stream of consciousness.

Likewise, in order to write a good program, it is important to break up your thoughts into distinct pieces (often referred to as **decomposition**). Computer scientists have invested time and effort into building languages and tools that assist programmers in this process. The concept of modularity and objected oriented languages are a result of this, but in this lab we will focus on the most basic tool of decomposition, the **method**. Methods are coherent blocks of code that encapsulate functionality. However, in order to effectively use methods, one must be able to parameterize a method's arguments through what is known as **parameter passing**.

Parameter passing differs from language to language, and thus computer scientists tend to study the mechanisms behind parameter passing in one language in order to not only become a better programmer in the language they are studying, but also to gain better intuition for how it works in other languages. Furthermore, there are certain classes of issues that are only analyzable through an understanding of how data is manipulated during the execution of a program.

This lab focuses on one of those problems, buffer overruns, and shows how understanding the run-time stack in C++ can help to both understand and prevent these fatal errors.

In this lab, you will:

- Review basic parameter-passing concepts
- Learn about the execution stack in C++
- Be able to use this information to exploit a buffer overrun

Review

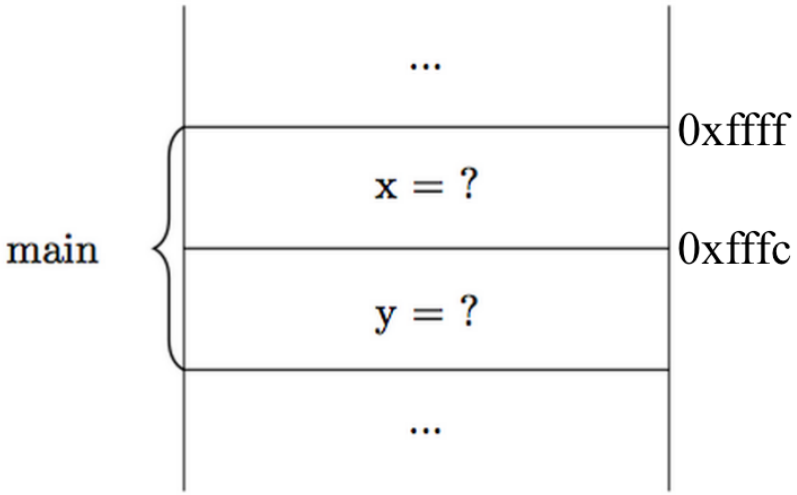
Runtime Stack

We begin with a brief review of parameter passing, looking into how the stack is utilized to make it possible. But what is the stack? The stack can be thought of as a stack of plates that grows ***downward***. Whenever a method is called, one of these plates (called a stack frame) is created with enough space to hold all the local variables. When we return from that method the stack frame for that method is popped off the stack, and its data is no more.

Let's say for example we have the following C++ code:

```
int main(){
  int x;
  int y;
  return 0;
}
```

When main is called, a stack frame is created with enough space for x and y as follows:

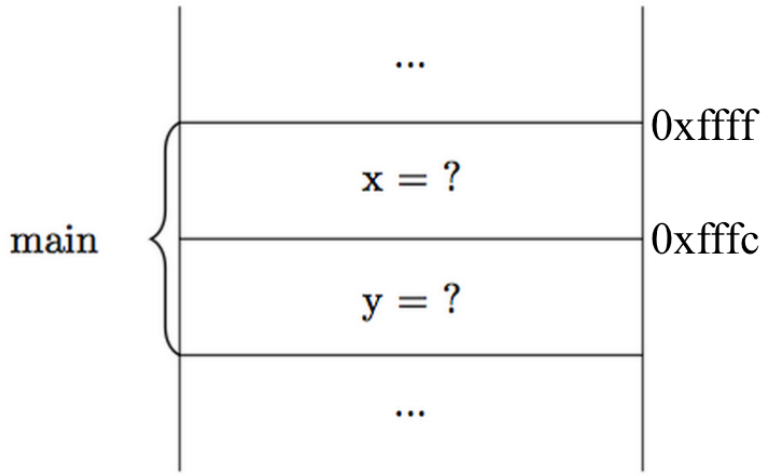


This **stack diagram** is a representation of memory in a computer. Memory is laid out logically starting at address 0x0 and ending with address 0xffffffff (notice how the numbers get ***larger*** as the stack grows ***downward***.)

If we add a method call, in addition to the variables declared inside of it, its 2 parameters are also pushed onto the stack. Say the above program is modified as follows:

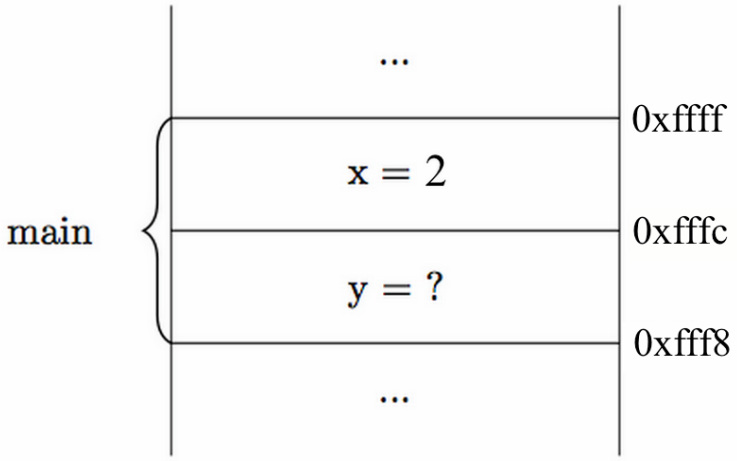
```
int foo(int z) {
  int x = z;
  return x + z;
}

int main() {
  int x = 2;
  int y = foo(x);
  return 0;
}
```

We begin by setting up the stack frame for main:

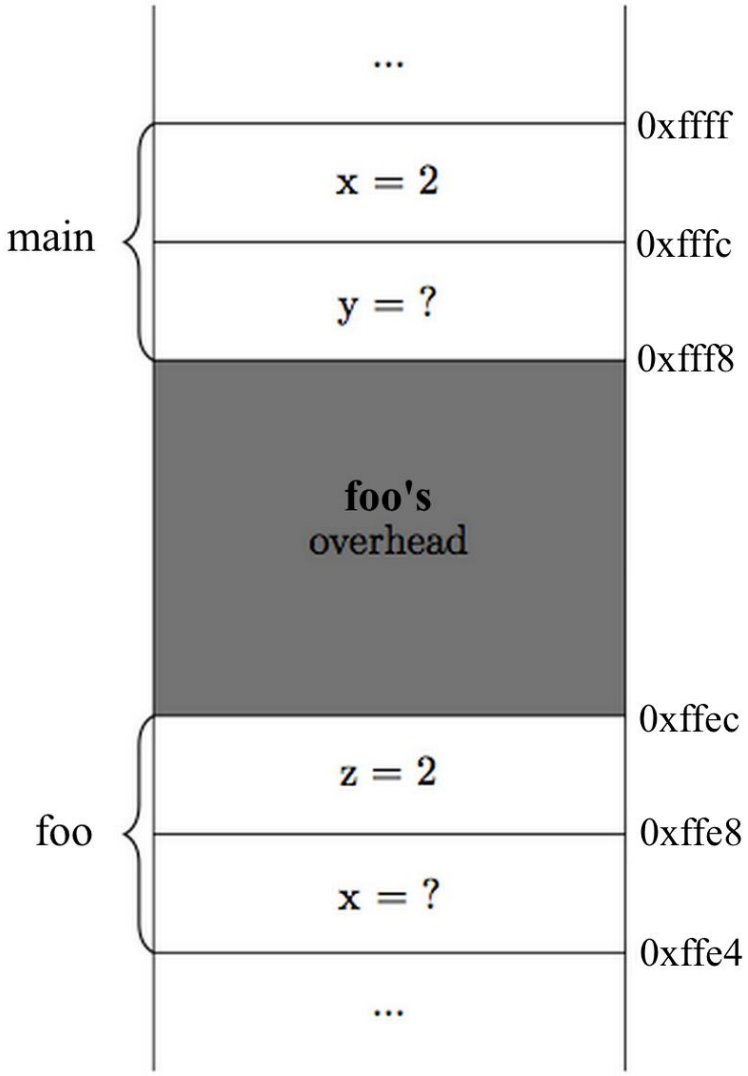


Next, we execute the line: `int x = 2;`

This will leave us with the following stack:



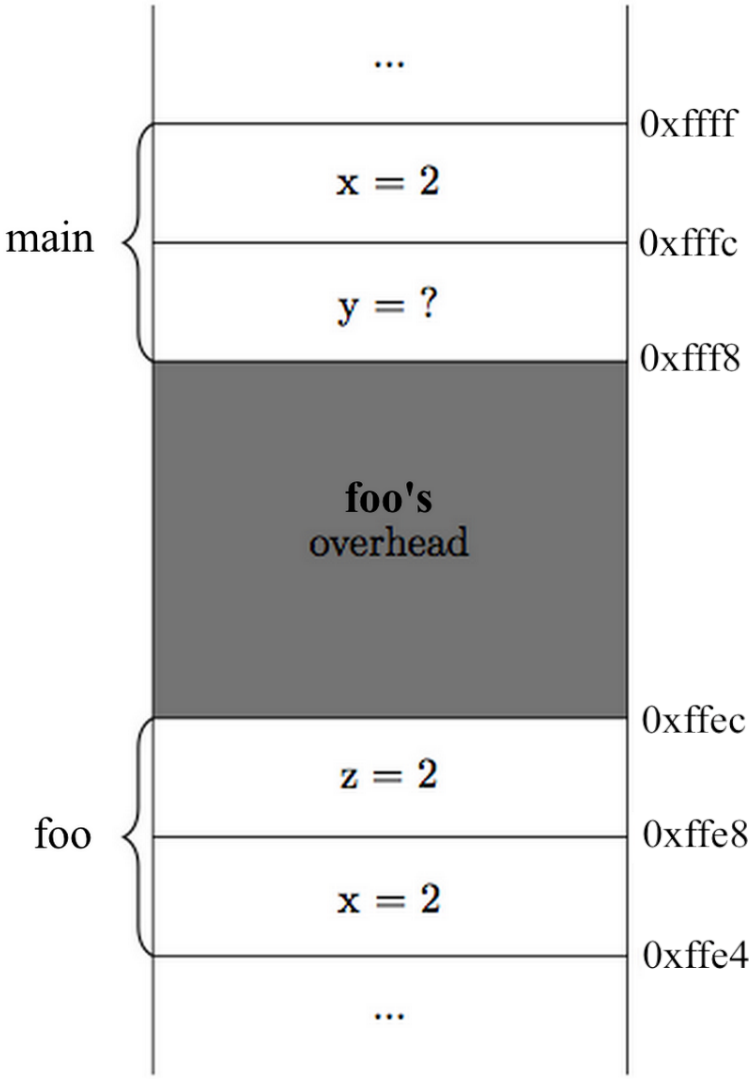
But in order to get a value for y, we must first execute the method foo. In order to prepare for foo to be called, main must place some overhead on the stack, then it must push any parameters necessary for foo on the stack. And so, when foo is called, it begins execution with the following execution stack.



Because foo is the top plate on the stack, we only see its local variables, thus there is no problem with different variables with identical names as long as they are used in a different method.

Continuing execution of the program we have the line: `int x = z;`

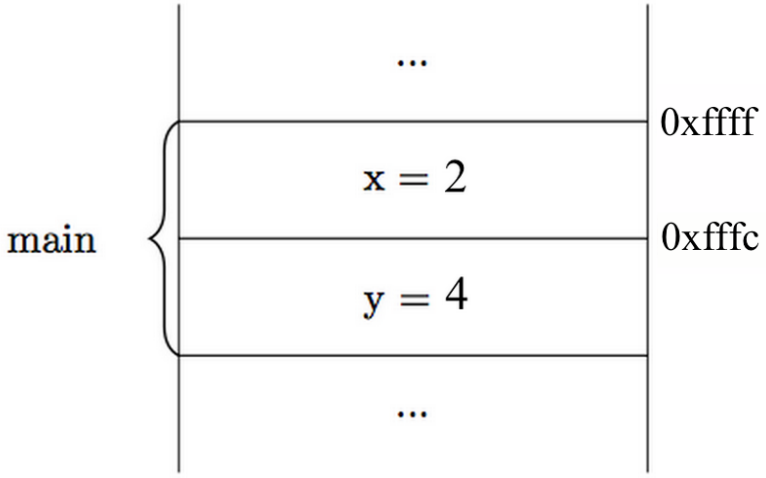
Which will write the value of z into x.



Notice that in order to attain the value of z, foo must blindly reach up and assume that main put the correct argument on the stack.

Next, we proceed to the line: `return x + z;`

This will have the effect of adding the values of x and y from foo's stack frame and signaling that this value is the return value. After we hit the closing brace we pop the stack frame for foo off the stack, and we are left with the following stack:



Where the 4 was the return value of the function call (foo). At this point the execution of our program is finished. Main will pop the rest of its stack frame, signaling the end of execution.

Exercises

1. After perfecting your C++ skills you land an internship as a software engineer at ATM inc. While buggy code is always troublesome, bugs in the ATM code can be particularly harmful and could potentially cause irrefutable damage.

   As part of your training, you are told to write a simple method that reads and authenticates account numbers and PINs. The method should prompt the user to enter an account number and a PIN (using `getline(cin, str)`) and check whether the PIN is the correct password for the given account number. Your supervisor has given you the following function:

   ```
   bool isAuthentic(char accountNum[], char pin[]);
   ```

   which takes two char arrays containing the 16-digit account number and 4-digit PIN (respectively). `isAuthentic` returns `true` if the PIN is the correct PIN for the given account number and returns `false` otherwise. You may assume that the user will only enter numeric characters. The method should return the account number if it authenticates and `""` if it does not.
   (Hint: you must convert the input into two arrays of `chars`)

   (you may add `#include` or `#define` lines and as many helper functions as you wish)

```
string getAccountNum() {

}
```

Is your code safe from buffer overflows? Why?

If the user entered an 8-digit number for the PIN instead of 4-digits, would your program behave correctly? Why?

Sometimes, programmers allocate space for local arrays and do not check to see if user input will fit the allocated space. This error is called a ***buffer overrun***. Buffer overruns in Java throw an exception, while in C++ they may go undetected. Are there advantages to the C++ approach of not throwing an exception? If so, what are they?

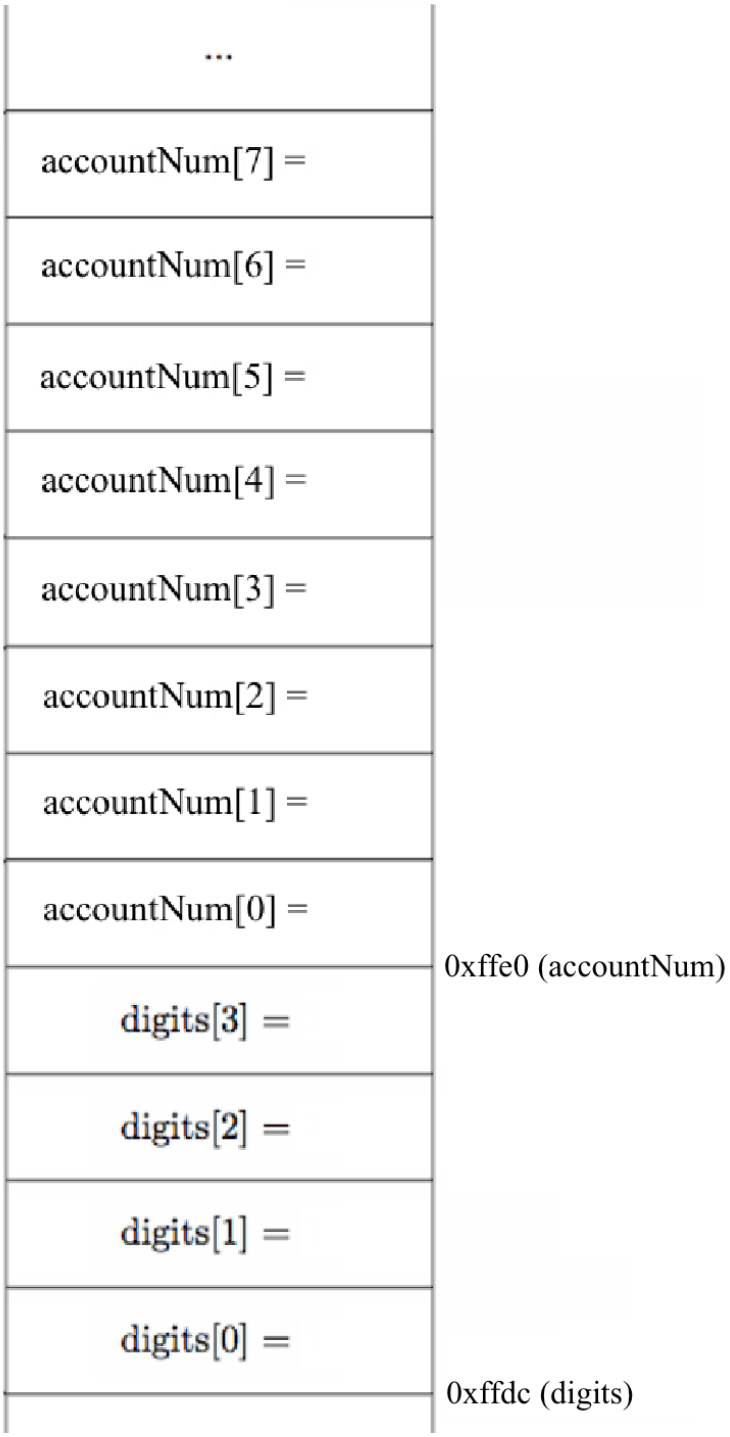
2. Just when you finish and present your code to your supervisor, someone else on your team says they have also finished the same task and claims that they have written secure code. You investigate and see the following code:

```
string getAccountNum() {
      char accountNum[16];
      char digits[4];

      cout << "Enter your 4 digit PIN: ";
      string pin_str;
      getline(cin, pin_str);

      cout << "Enter your 16 digit account number: ";
      string acc_num;
      getline(cin, acc_num);

      for (int j = 0; j < acc_num.length; j++) {
            accountNum[j] = acc_num[j];
      }
      for (int i = 0; i < pin_str.length; i++) {
            digits[i] = pin_str[i];
      }


      if (isAuthentic(accountNum, digits)) {
            return acc_num;
      }
      return "";
}
```

What is wrong with the code above?

Assume the user inputs 12345678 for the PIN and 1111222233334444 for the account number. Fill in a stack diagram just before isAuthentic is called.

*(Note: the stack grows **downward**, but arrays on the stack are arranged in **the opposite direction**, as shown below.)*


...
accountNum[7] =
accountNum[6] =
accountNum[5] =
accountNum[4] =
accountNum[3] =
accountNum[2] =
accountNum[1] =
accountNum[0] =
0xffe0 (accountNum)
digits[3] =
digits[2] =
digits[1] =
digits[0] =
0xffdc (digits)

Security Implications

In 2001, a computer worm, dubbed Code Red, was observed on the Internet. Released on July 13th, Code Red infiltrated 359,000 hosts, the largest group of computers infected, by July 19, 2001. The worm spread itself by taking advantage of a buffer overflow vulnerability; in fact, the vulnerability that the worm exploited was the exact same as the one from problem 2 above! The worm used a long string of the repeated character N to overflow a buffer. The string of N's was so long, in fact, that it overflowed past the local variables and into the **metadata** of a function, allowing the worm to execute the string following the last N as computer instructions and infect the machine. The effects of the worm included:

- Defaced the affected web site to display:
  HELLO! Welcome to http://www.worm.com! Hacked by Chinese!
- Launched denial of service attacks on several fixed IP addresses. The IP address of the white House web server was amount those.

For more information about the Code Red worm, you may visit these links:

- http://www.unixwiz.net/techtips/CodeRedII.html
- http://www.caida.org/research/security/code-red/