

Buffer Overflows

What is a Buffer Overflow?

To put it simple, it is putting too much information into a limited space. For example you declare a buffer with 8 elements in it. Now a buffer overflow would be to fill the 8-byte element buffer with 65 bytes. The problem here is that the rest 57 elements will overwrite the neighbouring memory cells. This comes particularly bad if the neighbouring memory cells are cells reserved by the operating system. If that happens then the Computer either crashes or hackers can gain higher privileges, depending on which memory area is accessed.

The Stack

You can imagine the stack like a pile of dishes in a kitchen. The last plate that is removed is the first to be served. This is the same with stacks in computers. The last thing that is *pushed* onto the stack will be the first that can be *popped* off the stack. A stack is mostly used in programming functions. Here functional parameters and local variables of the function are put on the stack in reverse order. Two other data are pushed to the stack The *frame pointer* and the *return address*. Every function is put into a frame on the stack. For example, in most programming languages the program starts with a main function that is returned to the operating system after execution. If the main function calls another function called myFunc for instance, this function needs to know where to go back after it executed. After execution the frame pointer sets the variables back to their original state and returns to the return address. The return address is the address where the flow of code continues after the function has executed.

The Stack and Buffer Overflow

Now you might say, okay i know how the stack works, so whats the buffer overflow deal? Remember where I said that buffer overflow is simply to put more information in a defined space than expected?

Lets take the myFunc function as an example. It has buffer of 65 bytes as its parameter and a buffer of 8 bytes as its local variable. Now myFunc will attempt to fill the 8 byte buffer with the elements of the 65 byte buffer. The result is clear, it will overwrite memory areas after the 8 byte buffer. In the stack however it will overwrite the frame pointer and the return address. If you fill the 65 buffer with A's the return address will be overwritten with 0x414141... (0x41 is hexadecimal for the ASCII symbol A). This means the function will return at the address 0x414141.... This location could be a read-only operating system area and with that the program will either crash or give the hacker higher privileges.

What can be done against Buffer Overflows?

On the programmers standpoint i would say careful programming especially with C/C++ functions. Especially the function `strcpy` is tricky since it does not do size checking. The function copies the content of one array into another, no matter what the sizes of both are. If the destination array is smaller than the source array, it will still do the copying. The advice I can give is, do a lot of testing, and doing error/exception handling.

Code Examples

Original Buffer Overflow Code

```
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;

void problem_func(char
*big_string)
{
    char myString[8];
    strcpy(myString,
big_string);
}

int main()
{
    int i;
    char bigstring[65];
    for(i = 0; i < 65; i++)
        bigstring[i] = 'A';
    problem_func(bigstring);

    return 0;
}
```

Error Checking / Handling Code

```
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;

void problem_func(char *big_string)
{
    char myString[8];
    strcpy(myString, big_string);
}

int main()
{
    int i;
    char bigstring[65];
    try // Error Checking
    {
        for(i = 0; i < 65; i++)
            bigstring[i] = 'A';
        if(i < 8)
            problem_func(bigstring);
        else
            throw "Buffer Overflow!";
    }
    catch(char const *err) // Error Handling
    {
        cout << "Err: " << err << endl;
        return 1;
    }
    return 0;
}
```

Explanation of the Codes

First thing you notice here is that the left original code is shorter than the other safe code. When the original code is executed, the program will simply crash or say something like “segmentation fault” on UNIX machines. The safe code will also try to fill the 8-byte buffer with 65 elements but the error handling will throw an exception and will stop the program. The try-catch block here can be seen as a safe guard of the program. Such constructs should be used in EVERY program written so errors/exceptions can be treated safely.

Taking this method of secure coding and a lot of testing into account, secure and joyful programs can be created.

So there is nothing more to say than “Happy (Safe) Hacking” and remember “Stay crispy in milk” :D