|  |
| --- |
| Overflows |
| Dylan Morrissey20075666Secure Programming and Scripting |

February 10, 2018

Table of Contents

[Introduction 2](#_Toc506284078)

[Unsigned int Overflow 2](#_Toc506284079)

[Buffer Overflow 5](#_Toc506284080)

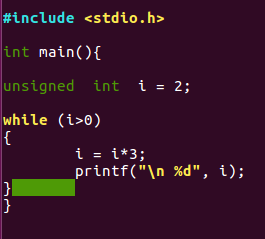
[Format String Attack 6](#_Toc506284081)

# Introduction

In this report I will be discussing 3 different types of overflows that I found when using C. I will be discussing what the over flow is, how it can be dangerous and finally how to resolve the Issue. All 3 of the short C programmes will be attach to the zip folder. I used Nano editor to write these programmes and GCC to compile them.

# Unsigned int Overflow

I will begin with an unsigned int overflow. Before we can understand what is happening first we must know what an unsigned int is. An unsigned int is still a 32-bit int data type but its values range from 0 to 4,294,967,295. It does not contain negative values unlike a signed int. Next is to look at the programme I wrote for an unsigned int overflow. Using on editor in my case I used nano editor to open the code.



Having a quick look at this code you can see that it is a while overflow which continues for ever and since the variable should never be less than 0. Inside the while loop we have a very simple multiplication which multiplies the unsigned int I by 3 every time.

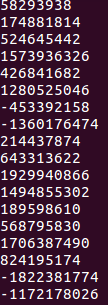
Next, we will have to compile the programme which is done as follows:



Next, we will run the code and see what happens:

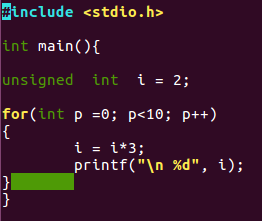


When running the code, we get a load of value which when you examine these values that might be something like this:



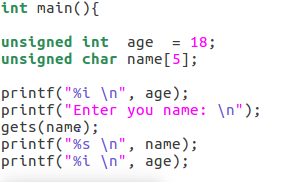
We can see that there is some minus number here and this is happening because the number is getting so large it goes outside the boundary for an unsigned int and this causes it to loop around which shouldn’t happen for an unsigned it as its lowest value is 0. This is an example of a simple unsigned int overflow.

This overflow could be fixed very simply by just improving the while loop or alternatively using a for loop for a certain number of calculation you want like below.



# Buffer Overflow

My next example will be of a simple buffer overflow. The buffer overflow is probably one of the most common bugs. A buffer overflow is simply when input for a running process such as asking a user for input exceeds the length of the buffer. So know we know what a buffer overflow is let’s have a look at my code.

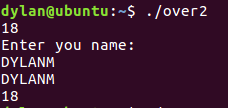


So, looking at this code we can see very simply the program assigns an unsigned int set to a value 18 and takes a user input of unsigned char. When the users input data is large enough It can overflow into the int causing the value for int this case age to change. Let’s have a look at an example of both situations:

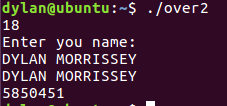
To compile this program we will have to turnoff stack protector which is done simply by:



Let’s Run the program and enter a small value for name:



Looking at this we can see it printed 18 first as the age, user entered an input in this case “DYLANM”, Which is printed out and then age of 18 is printed back out unchanged.



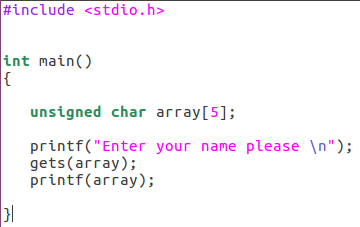
Now in this second example we can see that age 18 is printed out first then user enters their name in this case “Dylan Morrissey” much larger than the previous example and the age is printed again but in this case the value is 5850451 which is a lot larger than 18 which was entered. This was caused by the user input flowing into the age variable.

How to fix this issue:

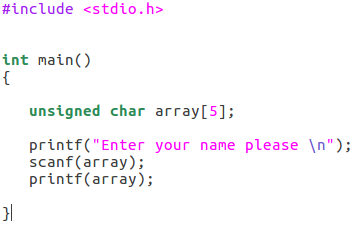
This issue is kind of fixed already with in C as this is why we disabled the stack protector as this fix smaller issues such as the above. Another way we can protect against this attack is to not use the get () function as it is quite unsafe.

# Format String Attack

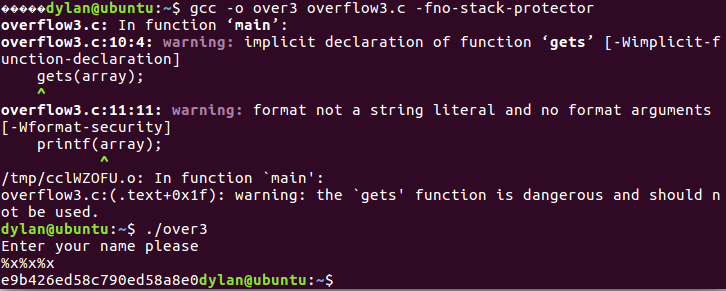
The final attack I will be looking at is a format String Attack, this is quite an easy attack it relies on the format specifiers in C to work. It works when a user enters a format specifier as an input where a programmer might not of specified a format specifier for the input. This can result in a lot of issues such as data leakage, Denial of Service and Memory overwriting. Let’s look at my code as an example:



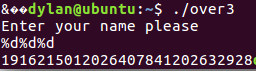
Looking at the code it is a lot easier to the previous one all it does is takes a user input for their name in this case and the prints the output. Although simple it is very easy to exploit. The attack will also work with scanf() ;



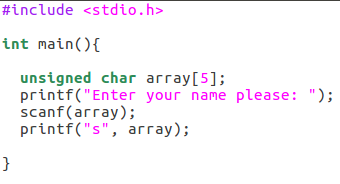
Let’s run the program and see how to use a format string attack.



Looking here we can see as my input I inputted %x%x%x. This is a format specifier for unsigned integer in hex and you can see what was returned wasn’t %x%x%x which would have been expected it was e9b426ed58c790ed58a8e0 which is a hex memory dump. You can do this for a lot of the format specifiers.



Here you can see I inputted %d%d%d which is a signed integer specifier and you can see that there was a int memory dump. It works the same as with the unsigned int and many more. Now that you have seen how easy it Is to do a format string attack let’s have a look at how to fix it.



This is a very quick fix to the issue and also leaving stack protector on will also protect against this. As it expects a value for a format specifier not an actual specifier.

# Conclusion

In conclusion there are a lot of different ways to attack code but with good programming practices most of these attacks can be avoided.