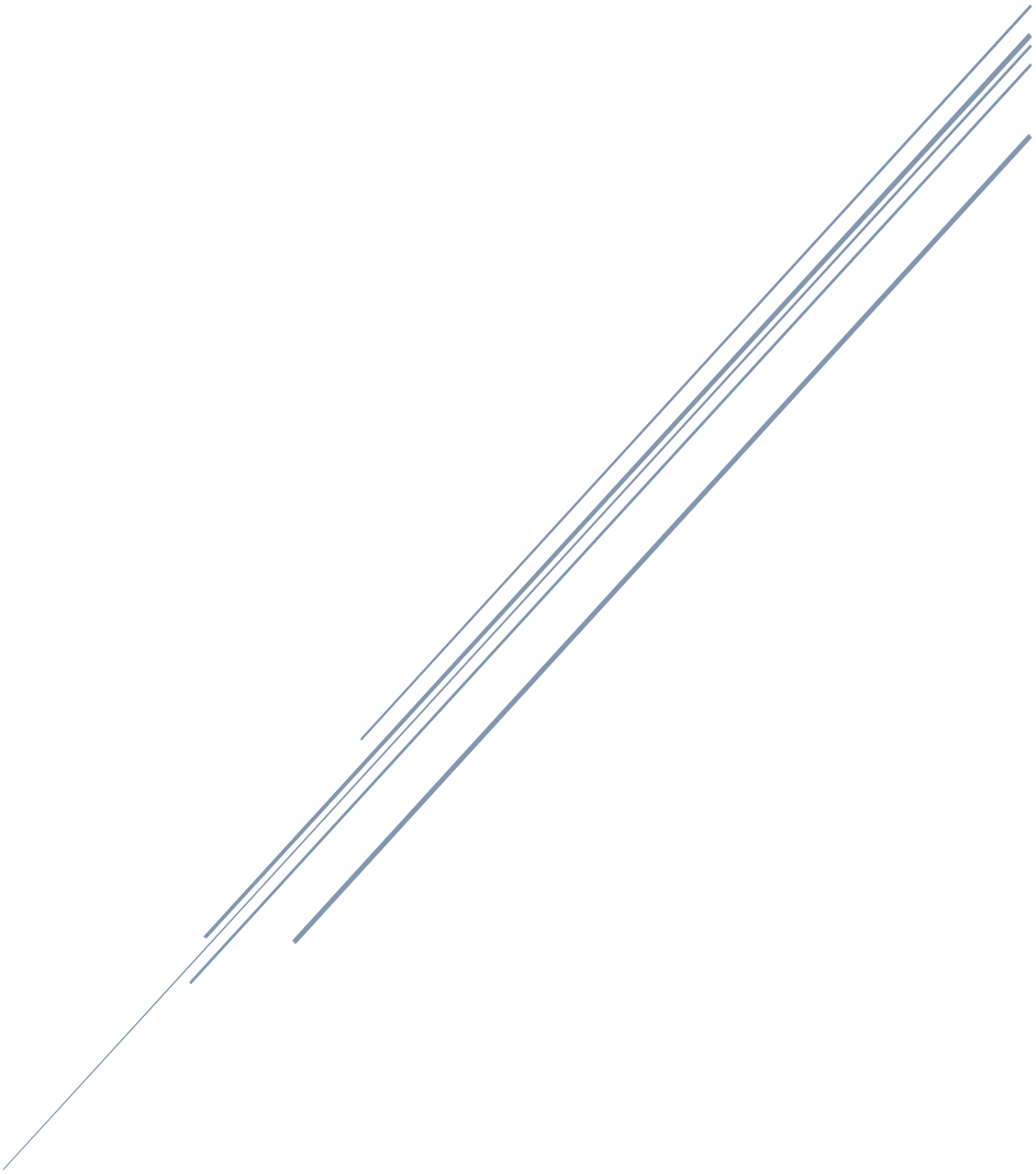
OVERFLOW ATTACKS



Jordon Coady - 20096529

SECURE PROGRAMMING & SCRIPTING

Contents

Integer Overflow Attack

In this program a user must enter the number of items they wish to purchase and the price of one item. The number of items is saved to a variable of type Int and the price per item value is saved to a variable of type Int. Then the total price is calculated by multiplying the number of items by the price per item and the value is saved to a variable of type Int. In this version of the program there are no checks done on the user input. Below is a demonstration of the program working correctly and when the values entered cause an Integer overflow.

```
┌──(kali㉿kali)-[~/SSP/Asign02]
└─$ gcc IntegerOverflow.c

┌──(kali㉿kali)-[~/SSP/Asign02]
└─$ ./a.out
Enter the number of items: 20
Enter the price per item: 5
Total price: 100

┌──(kali㉿kali)-[~/SSP/Asign02]
└─$ ./a.out
Enter the number of items: 555555
Enter the price per item: 555555
Total price: -596287287
```

The reason the value of the total price is negative is because when both variables are multiped together and the calculated value exceeds the maximum value that can be saved to an Int. This results in the value wrapping around to negative numbers.

Integer Overflow Fixed

To fix the program I added an if statement which can be seen below.

```
if (num_items > INT_MAX / item_price) {
  printf("Integer overflow detected!\n");
  return 1;
}
```

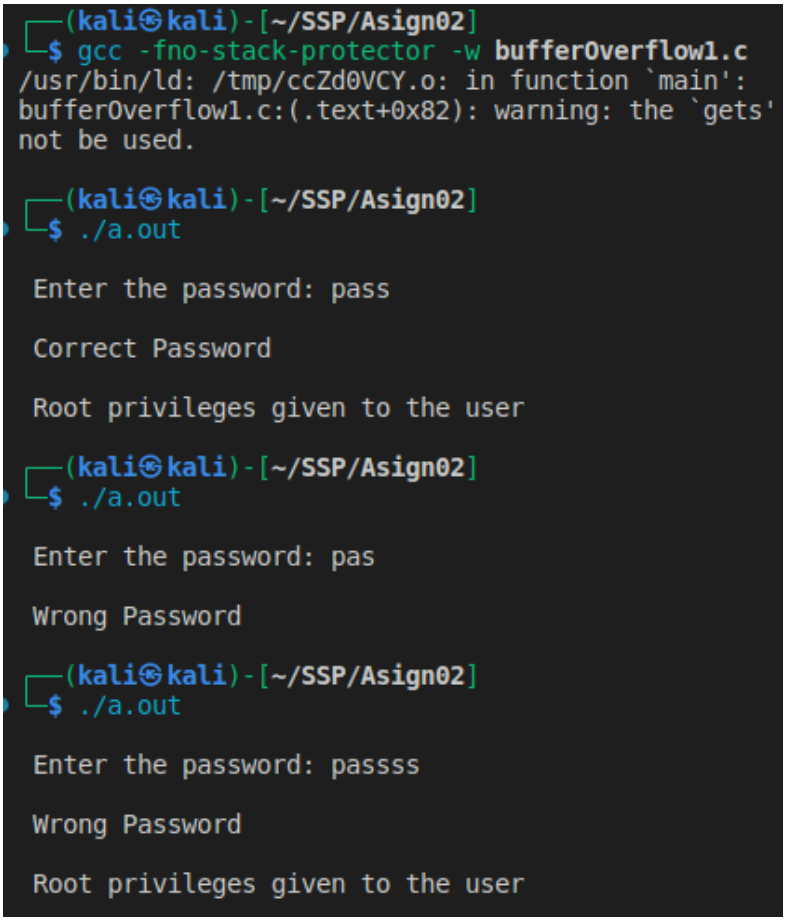
This if statement checks that the number of items isn't greater than the product of INT_MAX divided by the item price. The value of INT_MAX is 2,147,483,647 and if the result of INT_MAX divided by the item price is greater than the number of items it will result in the value overflowing or wrapping into the negative numbers if the number of items and the price of the item are multiped together. Below is an example of the if statement returning true meaning there was an integer overflow.

```
┌──(kali㉿kali)-[~/SSP/Asign02]
└─$ gcc IntegerOverflowFixed.c

┌──(kali㉿kali)-[~/SSP/Asign02]
└─$ ./a.out
Enter the number of items: 555555
Enter the price per item: 555555
Integer overflow detected!
```

Buffer Overflow Attack

This program will ask a user to enter a password, if the password matches pass the user will be granted root privileges. But if the user enters an incorrect password the program will print wrong password and terminate. Which can be seen in the image below.



But another possible outcome is that the user enters the wrong password but still gets root privileges. This occurs because the user enters in a password longer than the array used to store the given password. When this happens, an overflow occurs, and the extra letters will overflow into adjected memory and will alter the data stored there.

```
char buff[5];
int password = 0;
```

Above you can see an array is declared and initialised with a size of 5 and is followed by an int initialised with a value of 0. This means they are beside each other in memory so, when a password is entered that is bigger than five characters it will overflow to the password variable and alter the value. The reason this overflow is allowed to happen is because of the function gets(). This function doesn't check the size of the input string, instead it will try and force the string into the array and cause a buffer overflow.

Buffer Overflow Fixed

The fix for this buffer overflow only requires a small change. Instead of using gets() the function fgets() will be used.

Program susceptible to buffer overflow:

```
printf("\n Enter the password: ");
gets(buff);
```

Program fixed using fgets():

```
printf("\n Enter the password: ");
fgets(buff, 5, stdin);
```

The function fgets takes in three arguments, where is the string being stored to, the number of characters that should be copied to the array, and what input stream are the characters being read from. If used right this function will prevent a string that is too big being stored into an array that is too small. Below is the program with the fix implemented.

```
┌──(kali㉿kali)-[~/SSP/Asign02]
└─$ gcc bufferOverflowFixed.c

┌──(kali㉿kali)-[~/SSP/Asign02]
└─$ ./a.out

 Enter the password: notThePass

 Wrong Password
```

Buffer overflow 2 Attack

This program will ask the user how many numbers they want to enter. The number the user gives is stored in a variable called size and this variable is used to determine the size of the array by using this line of code **num = malloc(size * sizeof(int));** Then the user is prompted to enter n numbers. The numbers the user enters is copied to another array called small array that is initialised which a size of three. Then the numbers stored in the small array are printed out.

```
┌──(kali㉿kali)-[~/SSP/Asign02]
└─$ gcc -fno-stack-protector -w bufferOverflow2.c

┌──(kali㉿kali)-[~/SSP/Asign02]
└─$ ./a.out
How many numbers do you want to enter: 3
Enter 3 numbers: 1 2 3
1 2 3

┌──(kali㉿kali)-[~/SSP/Asign02]
└─$ ./a.out
How many numbers do you want to enter: 6
Enter 6 numbers: 1 2 3 4 5 6
1 2 3 4 5 5

┌──(kali㉿kali)-[~/SSP/Asign02]
└─$ ./a.out
How many numbers do you want to enter: 20
Enter 20 numbers: 1 2 3 4 5 6 7 8 9 10 11 12 3 4 5 6 7 8 9 20
zsh: segmentation fault  ./a.out
```

In the above image you can see three possible outcomes. In the first outcome the user enters three numbers 1, 2, 3 then the numbers are copied to the small array and printed out. In the second outcome the user decides to enter six numbers, which results in the numbers 1, 2, 3, 4, 5, 5 being printed out. The reason this happened is because the destination buffer is smaller than the source buffer and the memcpy function tried to copy more data than the destination can hold which resulted in an overflow and when an overflow occurs unexpected behaviours can to being to happen. In the last outcome the program exits with a segmentation fault meaning the memcpy function tried to access memory that wasn't allocated to the program.

```
memcpy(dest: small_num, src: nums, n: size * sizeof(int));
```

Memcpy is used to copy a block of memory from one location to another. It takes three arguments: a pointer to the destination array, a pointer to the source of the data being copied and the number of bytes that should be copied. In the image above you can see the memcpy function is trying to copy the size of the nums array which if the size is bigger than 3 will result in a buffer overflow.

Buffer Overflow 2 fixed

Instead of changing the size argument in the memcpy function to only allow three integers to be copied, I made the smaller array dynamic, meaning the input the user gives at the start is used to define the size of the second array. This means that both arrays will be the same size and no overflows can occur from the memcpy function.

```
printf("How many numbers do you want to enter: ");
scanf("%d", &size);
num = malloc(size * sizeof(int));
arr2 = malloc(size * sizeof(int));
```

Now both arrays are being dynamically sized based on the user input. This will prevent the memcpy function from copy data from a larger source to a smaller destination.