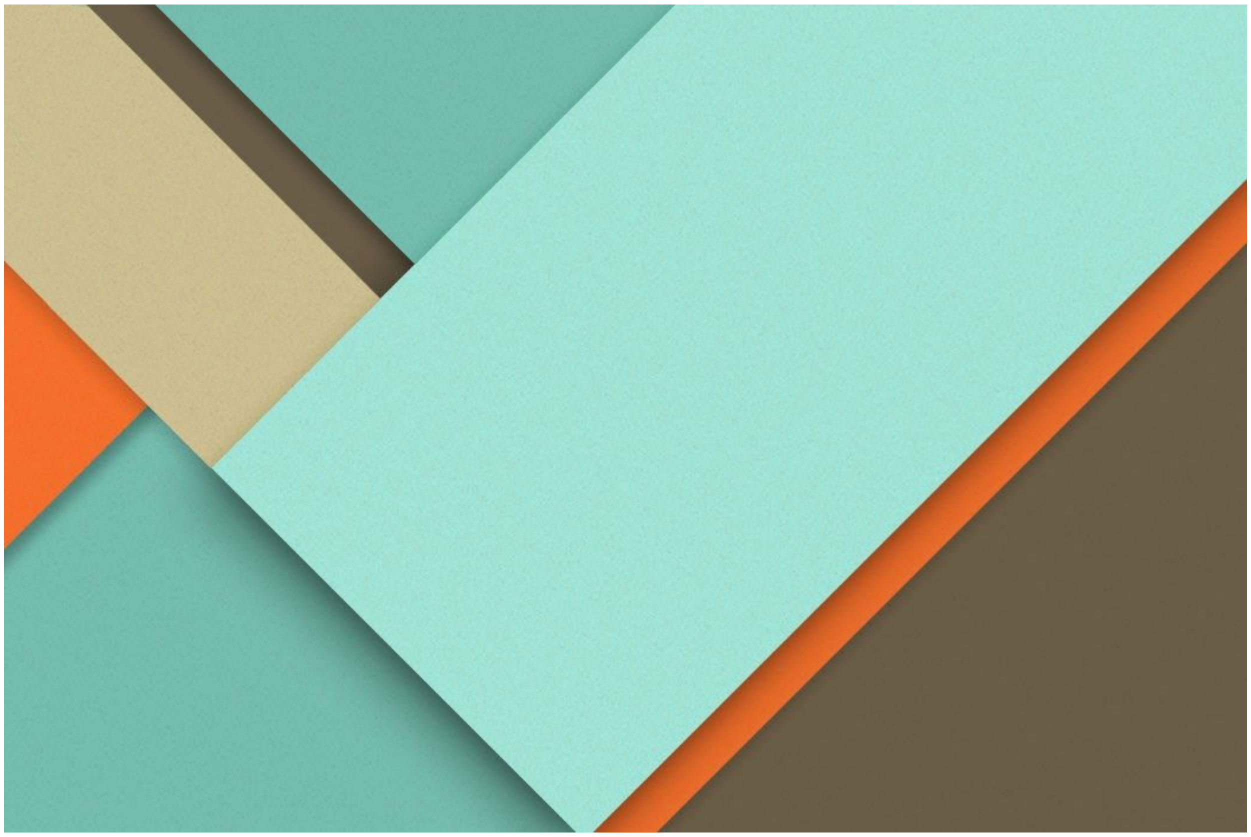

Secure Programming

Assignment One

Dara O’Sullivan
20080494
Computer Forensics
2nd Year

Overview

Below I will walk through three code examples which show in action an integer overflow, a buffer overflow and a format string vulnerability. Both integer overflow and buffer overflows allow the data to be manipulated in the program. A format string attack is also subject to manipulation but is a more serious issue as the attacker has direct access to the memory.

1. Integer Overflow

When an arithmetic operation attempts to create a numeric value that is outside of the size bounds of the integer type used to store it an integer overflow occurs.

In the program the first line we have signed chars, as char get assigned as signed char as default. In the program code we set the char calculation to 50 + 250 = 44 but as you can see without getting any error message the 250 gets changed to -6. This is because 250 is out of bounds for a signed char, the bounds are -128 to 127 for signed chars. Because 250 is too large to fit into the char literal the value therefore overflows into the bit of your char that defines negativeness. In the second line we have the unsigned chars which bounds are 0-255. Here we see 210 will set correctly as it's now within bounds. But 260 cannot be represented as an unsigned bit so it overflows and goes to 4 because it's four from what it should be.

Compiling the program we get no errors:

```
root1@ubuntu:~$ gcc -o Int Int.c
```

The output of running the program is as follows:

```
root1@ubuntu:~$ ./Int
char:   50 + -6 = 44
unsigned char:   50 + 210 = 4
```

1.1 Preventing integer overflow:

If you compile your program with gcc or clang we can ensure wraparound by using compiler flags. Wraparound will give most calculations a result not far from the actual value but this result is not correct and so we need other ways to deal with integer overflow to ensure reliable programs.

1. In C integer overflow is classified as undefined behaviour. C provides a detector for undefined behaviour, UndefinedBehaviorSanitizer (**UBSan**) which will detect for us any overflows.
2. One of the easiest ways to prevent integer overflow is to use a function, reallocarray() which is provided by C for this exact vulnerability. This function will detect integer overflow by checking the result of the arithmetic calculation is within the type size.
3. Ensuring the type size is large enough for any values which could be calculated by the arithmetic equation will prevent overflow.
4. If we are expecting integer overflow we can come up with a way of handling it. We can perform certain calculations on the inputs to rectify them before the calculation is done and the program would crash.
5. Saturated arithmetic limits operations to a fixed range between a minimum and maximum value. So for example like in our code above for the unsigned bit the highest value set will be 255 and the lowest 0. If the calculation result returns 260, 255 will be set etc.

2. String buffer overflow

A buffer overflow is caused by a program or process trying to write more data to a buffer than what is allocated. This can result in the program crashing or corrupting the memory but worse makes the program vulnerable. An informed hacker would be able to access the stack and heap of the program and execute arbitrary code.

In the code below we declared an array of size 20 bytes which will create a buffer on the stack of that size .We are simply asking users for password, if it matches wit2020 we essentially return access gained, if it doesn't we return access denied. The gets() function does not check the array bounds therefore when running this program a hacker can supply an input greater in length than 20 bytes and at a certain number of bytes over the (in this program case its 25 bytes, five bytes past the buffer array) the buffer overflow takes place. Gcc compiler gives up the message "*** stack smashing detected ***: <unknown> terminated, Aborted (core dumped)" which stops the operation. Without this detection the attacker would have been able to overwrite the integer "access" to something other than zero. Resultantly the privileges were granted to the hacker.

Compiling our program we get some errors:

```
root1@ubuntu:~$ gcc -o Buffer Buffer.c
Buffer.c: In function 'main':
Buffer.c:8:5: warning: implicit declaration of function 'gets'; did you mean 'fgets'?
[-Wimplicit-function-declaration]
    gets(buffer);
    ^~~~
    fgets
/tmp/cckkx4xs.o: In function `main':
Buffer.c:(.text+0x37): warning: the `gets' function is dangerous and should not be used.
```

Running our program with correct password:

```
root1@ubuntu:~$ ./Buffer

 Enter password :
wit2020

 Correct Password

 Root privileges given to the user
```

Running the program with wrong password:

```
root1@ubuntu:~$ ./Buffer
```

```
 Enter password :
123

Wrong Password
```

Running program with password out of buffer bounds. If stack smash detection was not used here, the attacker would be able to manipulate the program because the access flag would be a non-zero which in the program gives privileged access to users.

```
root1@ubuntu:~$ ./Buffer

 Enter password :
1234567890123456789012345

 Wrong Password
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)
```

2.1 Preventing buffer overflow:

The most reliable way to prevent a buffer overflow is at the production code level, avoiding any of the common traps within the language. This is not always easy in C as it is not a strongly typed language. There is also direct access to memory in C which makes it very vulnerable to start with. This is not always an option though so below are other methods which could be used.

1. Because buffer overflows mainly occur when using arrays and strings, if a library which has some buffer management is used within these types it will highly impact the effect of the occurrence of overflows. The more reliable the library the better.
2. Buffer overflow protection checks can be performed on the stack which will indicate if there are any alterations after a function if performed. If it has been altered the program exits with a segmentation fault.
3. As buffer overflows can be manipulative due to the fact they can move pointers if we can prevent pointers from being moved we can prevent vulnerabilities. This is done by XORing the pointers before and after they are used so the hacker can not know how to manipulate them. Known as pointer protection.

3. Format string attack

A user entering more format specifiers than what an argument is meant to take in will result in a format string vulnerability. Many functions in c, for example sn**printf()**, which we use in the code below, will retrieve as many values as specified by the user from the stack rather than how many are specified to be taken in by the code.

In our code only one of the places on the stack is occupied by an actual function argument (buffer), but we can access more, ie the value next on the stack by tricking the program. If we insert "Wit%x %x" as the input , because the program is then looking for arguments which do not belong to the program the contents of the two memory addresses after the buffer will be displayed .

Compiling the program we get the following warnings:

```
root1@ubuntu:~$ gcc -o Format Format.c
Format.c: In function 'main':
Format.c:8:1: warning: format not a string literal and no format arguments [-Wformat-security]
snprintf ( buffer, sizeof buffer, argv [1] ) ;
^~~~~~~~
```

Running program with non malicious input:

```
root1@ubuntu:~$ ./Format Bob
Data input: Bob
```

Running program with malicious input:

```
root1@ubuntu:~$ ./Format Bob%x%x
Data input: Bob7b24f780ddb71d80
```

3.1 Preventing format string attack:

1. A simple solution to prevent format string vulnerabilities is to specify a format string as part of a program rather than an input. This will prevent the ability to manipulate it and is done in the production code.
2. Use a defense tool such as as Format_Guard which provide automatic protection from printf format string.