Overflows

Assessment 2 for Secure Programming & Scripting

Zalán Tóth - 20102768
Computer Forensics and Security Year 2

SETU Waterford
School of Science and Computing
Department of Computing and Mathematics

Waterford City, Ireland
10/03/2024

Table of Contents

I. Introduction

Overflow occurs when data exceeds the storage capacity allocated for it, leading to vulnerabilities and unpredictable behavior in our application or the system. There are several types of overflows, like buffer or stack overflow.
This paper demonstrates 3 types of them:

A. Integer Overflow & Underflow
B. Floating Point Overflow
C. Heap Overflow

II. Integer Overflow & Underflow

A. Code

The code is included in a separate file as well, as requested.

```c
#include <stdio.h>
#include <limits.h>


int main() {

   printf("UNSIGNED INTEGER OVERFLOW\n");
   unsigned int unsignedInt = UINT_MAX; //Maximum value for an unsigned int
   printf("Unsigned int initial value: %u\n", unsignedInt);
   unsignedInt += 1;
   printf("Unsigned int value after overflow: %u\n\n", unsignedInt);


   printf("UNSIGNED INTEGER UNDERFLOW\n");
   //unsignedInt is already 0 (because of the overflow) which is the lowest possible number
   printf("Unsigned int initial value: %u\n", unsignedInt);
   unsignedInt -= 1;
   printf("Unsigned int value after underflow: %u\n\n", unsignedInt);

   printf("SIGNED INTEGER OVERFLOW\n");
   int signedInt = INT_MAX; // Maximum value for a signed int
   printf("Signed int initial value: %d\n", signedInt);
   signedInt += 1;
   printf("Signed int value after overflow: %d\n\n", signedInt);

   printf("SIGNED INTEGER UNDERFLOW\n");
   printf("Signed int initial value: %d\n", signedInt);
   signedInt -= 1;
   printf("Signed int value after underflow: %d\n\n", signedInt);

   return 0;
}
```

B. Code Origin

The code was written by this paper's author (Zalán Tóth).

C. How the code works

It is a basic code that demonstrates how integer overflow/underflow can happen with basic incremental or decremental operations on the edges of integers (at maximum and minimum values). When we are dealing with integers, the amount of possible values to be stored is 4294967296 (32 bits, so 2^32). When we are dealing with unsigned ones, the minimum value is 0 and the maximum value is 4294967295. So when we try to increment the maximum value, the number overflows, and becomes 0, and when we wanna decrement the minimum value which is 0, then we get the maximum value, which we call an underflow. A similar thing happens when we are dealing with signed integers, but in this case, the first byte determines if the number is negative or not. So the range goes from -2147483648 to 2147483647, which means those are the 2 limits of storing, so if we try to go beyond those limits, we get an overflow or an underflow.

D. Compiling the code

Compiling the code:

```
zalan@zalan-mobile-msi-powerplant:~/Projects/C/Assessment2$ gcc
20102768-Integer-SRC.c -o 20102768-Integer-APP.out
zalan@zalan-mobile-msi-powerplant:~/Projects/C/Assessment2$ gcc
20102768-Integer_fixed-SRC.c -o 20102768-Integer_fixed-APP.out
zalan@zalan-mobile-msi-powerplant:~/Projects/C/Assessment2$ ls
20102768-Integer-APP.out 20102768-Integer_fixed-SRC.c
20102768-Integer_fixed-APP.out 20102768-Integer-SRC.c
```

Running the code:

```
zalan@zalan-mobile-msi-powerplant:~/Projects/C/Assessment2$
./20102768-Integer-APP.out
```

E. Output and the result of running

```
zalan@zalan-mobile-msi-powerplant:~/Projects/C/Assessment2$
./20102768-Integer-APP.out
UNSIGNED INTEGER OVERFLOW
Unsigned int initial value: 4294967295
Unsigned int value after overflow: 0

UNSIGNED INTEGER UNDERFLOW
Unsigned int initial value: 0
Unsigned int value after underflow: 4294967295

SIGNED INTEGER OVERFLOW
Signed int initial value: 2147483648
```

```
Signed int value after overflow: -2147483648

SIGNED INTEGER UNDERFLOW
Signed int initial value: -2147483648
Signed int value after underflow: 2147483647

zalan@zalan-mobile-msi-powerplant:~/Projects/C/Assessment2$
./20102768-Integer_fixed-APP.out
UNSIGNED INTEGER OVERFLOW
Unsigned int initial value: 4294967295
An overflow will happen. Skipping increment.
Unsigned int value after avoiding overflow: 4294967295

UNSIGNED INTEGER UNDERFLOW
Unsigned int initial value: 0
An underflow will happen. Skipping decrement.
Unsigned int value after avoiding underflow: 0

SIGNED INTEGER OVERFLOW
Signed int initial value: 2147483647
An overflow will happen. Skipping increment.
Signed int value after avoiding overflow: -2147483648

SIGNED INTEGER UNDERFLOW
Signed int initial value: -2147483648
An underflow will happen. Skipping decrement.
Signed int value after avoiding underflow: 2147483647
```

F. Explanation of the overflow

As we have a limited amount of storage space, each integer has maximum and minimum values. An integer takes up 32 bits of space. We cannot store more values than 2^32. That means if we run out of space, the number will overflow or underflow depending on the situation as explained earlier.

G. Code that fixes the problem

The following code is a very basic fix that only works for incremental operations.
The code is included in a separate file as well as requested.

```c
#include <stdio.h>
#include <limits.h>

int main() {
```

```
    printf("UNSIGNED INTEGER OVERFLOW\n");
    unsigned int unsignedInt = UINT_MAX; // Maximum value for an
unsigned int
    printf("Unsigned int initial value: %u\n", unsignedInt);
    // Check for overflow
    if (unsignedInt == UINT_MAX) {
        printf("An overflow will happen. Skipping increment.\n");
    } else {
        unsignedInt += 1;
    }
    printf("Unsigned int value after avoiding overflow: %u\n\n\n",
unsignedInt);

    printf("UNSIGNED INTEGER UNDERFLOW\n");
    // Manually setting unsignedInt to 0 for underflow demonstration
    unsignedInt = 0;
    printf("Unsigned int initial value: %u\n", unsignedInt);
    // Check for underflow
    if (unsignedInt == 0) {
        printf("An underflow will happen. Skipping decrement.\n");
    } else {
        unsignedInt -= 1;
    }
    printf("Unsigned int value after avoiding underflow: %u\n\n\n",
unsignedInt);

    printf("SIGNED INTEGER OVERFLOW\n");
    int signedInt = INT_MAX; // Maximum value for a signed int
    printf("Signed int initial value: %d\n", signedInt);
    // Check for overflow
    if (signedInt == INT_MAX) {
        printf("An overflow will happen. Skipping increment.\n");
    } else {
        signedInt += 1;
    }
    printf("Signed int value after avoiding overflow: %d\n\n\n",
signedInt);

    printf("SIGNED INTEGER UNDERFLOW\n");
    signedInt = INT_MIN; // Manually setting to minimum for underflow
demonstration
    printf("Signed int initial value: %d\n", signedInt);
```

```
    // Check for underflow
    if (signedInt == INT_MIN) {
        printf("An underflow will happen. Skipping decrement.\n");
    } else {
        signedInt -= 1;
    }
    printf("Signed int value after avoiding underflow: %d\n\n",
signedInt);

    return 0;
}
```

H. Conclusion

While we are programming, we have to make sure that the data we are doing operations on will stay within its limits, and won't overflow/underflow. There are several types of fixes for this problem, for example, the code I provided is fairly simple and only works with basic incrementation/decrementation and does not protect for more complex operations, but in this case, it's sufficient. We have to be careful how we store data and how we do those operations on those data because if an overflow happens, it'll change the data in an unexpected way, which might lead the program to crash or other unexpected behaviors.

III. Floating Point Overflow

A. Code

The code is included in a separate file as well, as requested.

```c
#include <stdio.h>

#include <float.h>


int main() {

   float a = FLT_MAX;

   float b = a * 2;

   printf("%f\n", b);

   return 0;

}
```

B. Code Origin

The code was written by this paper's author (Zalán Tóth).

C. How the code works

Given the maximum floating number, we try to double that, which means we try to overflow the number and we get an answer of “inf”.

D. Compiling the code

```
zalan@zalan-mobile-msi-powerplant:~/Projects/C/Assessment2/floatingPointOverflow
$ gcc 20102768-Floating-SRC.c -o 20102768-Floating-APP.out
zalan@zalan-mobile-msi-powerplant:~/Projects/C/Assessment2/floatingPointOverflow
$ gcc 20102768-Floating_fixed-SRC.c -o 20102768-Floating_fixed-APP.out
```

E. Output and the result of running

```
zalan@zalan-mobile-msi-powerplant:~/Projects/C/Assessment2/floatingPointOverflow
$ ./20102768-Floating-APP.out inf
zalan@zalan-mobile-msi-powerplant:~/Projects/C/Assessment2/floatingPointOverflow
```

$ ./20102768-Floating_fixed-APP.out Overflow detected for the float, so returning the output value as a double: 6805646932770577196234083669690338508800.000000

F. Explanation of the overflow

Just as integers, floats have maximum and minimum values as well. After all, we work with computers, and computers work with finite numbers and finite resources. If the result of the operation is bigger than the float can store itself, it'll return the value "inf".

A. Code that fixes the problem

The code is included in a separate file as well as requested.

```c
#include <stdio.h>
#include <float.h>

int main() {
   float a = FLT_MAX;
   double b = (double)a * 2;
   if (b > FLT_MAX) {
       printf("Overflow detected for the float, so returning the output value as a double: %f\n",b);
   } else {
       float result = (float)b;
       printf("%f\n", result);
   }
   return 0;
}
```

G. Conclusion

We have to make sure that we won't go beyond the boundaries of the data we're trying to do operations on. In my example, I did a quick fix for the code, which all it does is detect if the result is bigger than the maximum float can be, and it'll store the result as a double instead which has more precision, so we'll get the real result, but at the cost of more storage space.

IV. Heap Overflow

B. Code

The code is included in a separate file as well as requested.

```c
#include <stdlib.h>

#include <string.h>



int main() {

   char *buf = malloc(10);

   strcpy(buf, "Hello I'm Mr. Meeseeks\n");

   free(buf);

   return 0;

}
```

C. Code Origin

The code was written by this paper's author (Zalán Tóth).

D. How the code works

The code allocates a 10-byte memory space dynamically, and then it tries to store the text “Hello I’m Mr. Meeseeks”, which is more than 10 bytes. That means the program will try to store that text in a smaller data area, which means it’ll go beyond its boundary, writing information to a space in which it was not intended.

E. Compiling the code

```
zalan@zalan-mobile-msi-powerplant:~/Projects/C/Assessment2/heapOverflow$ gcc -fno-stack-protector 20102768-Heap-SRC.c -o 20102768-Heap-APP.out
20102768-Heap-SRC.c: In function ‘main’:
20102768-Heap-SRC.c:7:5: warning: ‘__builtin_memcpy’ writing 24 bytes into a region of size 10 overflows the destination [-Wstringop-overflow=]
    7 |    strcpy(buf, "Hello I'm Mr. Meeseeks\n");
      |    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
20102768-Heap-SRC.c:6:17: note: destination object of size 10 allocated by ‘malloc’
    6 |    char *buf = malloc(10);
      |                ^~~~~~~~~~
```

```
zalan@zalan-mobile-msi-powerplant:~/Projects/C/Assessment2/heapOverflow$ gcc
20102768-Heap_fixed-SRC.c -o 20102768-Heap_fixed-APP.out
```

F. Output and the result of running

```
zalan@zalan-mobile-msi-powerplant:~/Projects/C/Assessment2/heapOverflow$
./20102768-Heap-APP.out
Hello I'm Mr. Meeseeks
zalan@zalan-mobile-msi-powerplant:~/Projects/C/Assessment2/heapOverflow$
./20102768-Heap_fixed-APP.out
Hello I'm Mr. Meeseeks
```

The first app should've cut off the string, as there isn't enough space to store that many characters, but it did not. My system even warned me when I compiled the code (as can be seen in the previous section - Compiling the code).

G. Explanation of the overflow

This overflow is basically a type of buffer overflow. In this case, the data storage we're trying the data to store was allocated dynamically (so the storage is in the heap). Because the text takes up much more space than 10 bytes, the program will write beyond the allocated memory space, which is very unsafe.

H. Code that fixes the problem

The code is included in a separate file as well as requested.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
   char *buf = malloc(strlen("Hello I'm Mr. Meeseeks\n") + 1);  //+1
for the NULL terminator
   if (buf == NULL) {
       return 1;
   }

   strcpy(buf, "Hello I'm Mr. Meeseeks\n");
   printf("%s", buf);
   free(buf);
   return 0;
}
```

I. Conclusion

When a program tries to write beyond its boundaries, it's very unsafe. The program can overwrite data which was not intended to do, and this can lead to crashes or other unexpected results in the program or even the system. We have to make sure we allocate the space for the data dynamically. In my example, I just did a quick fix. Instead of allocating a fixed amount of space (due to the hardcoded way I did my program it can be argued that it's still fixed, but that's just a basic demonstration), I just measure the length of the string I'm trying to create. I just had to add an extra byte for the NULL terminator and then we're in the safe zone. Now we have just enough space to store that message, and we won't write out of boundaries.