DEFENCE UNIVERSITY

COLLEGE OF ENGINEERING

Department of Computer and Information Technology

Specialization: Cybersecurity

Course: Programming Concepts and Security
Title: Assignment
Sub title: Integer overflow

BY:BEREKET MEKONNEN

Dr. Solomon Zemene
Date: 4 August 2020

Question A

write a program that shows integer vulnerability caused by arithmetic overflow that lead to heap overflow.

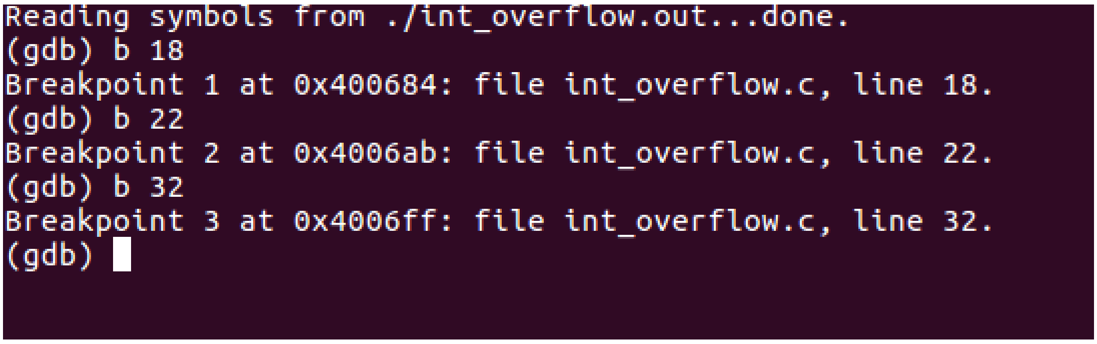
Solution:

When allocating a memory for a heap with a large number that is grater that the size of unsigned int there will be a warp around and the value more than the limit will become the lowest unsigned int. this is because of when using malloc() the data type used will be used with the cast to size_t and the value will never get to negative and below the size of the unsigned int value.

This have been demonstrated with the following code and the gdb debugger to view the behavior of the heap.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
int main(int argc, char *argv[]){
        int i,x,value;
        char *fun;

   if(argc <3) return -1;

        i = atoi(argv[1]);
        value = i*sizeof(char*);

        fun = (char *)malloc(value);
        printf("%p\n",fun );

        if (fun == NULL){
                return -1;
        }
        for(x = 0; x < i; x++){
                fun[x]= argv[2][x];
        }

        printf("%s\n",fun );

free(fun);

}
```

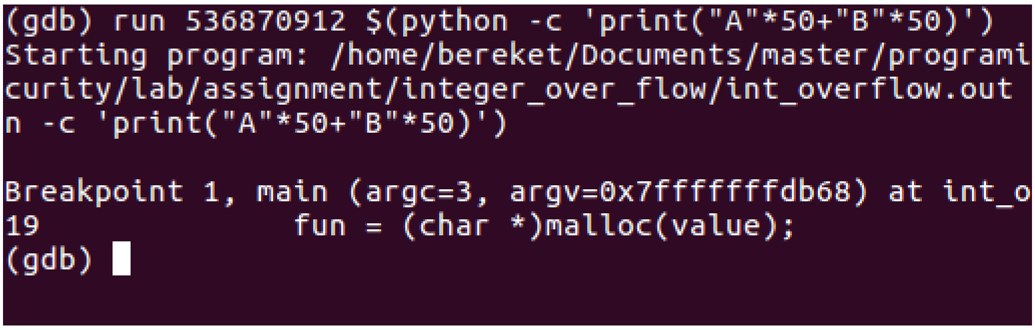
By using the above code and compiling it with the flag **-g** and after that we will execute the code from the debugger. By using **gdb ./int_overflow.out** . Then assigning braking points on pointes that we want to monitor the heap, and viewing the memory to see what is written and the size of the data that is written.



```
Reading symbols from ./int_overflow.out...done.
(gdb) b 18
Breakpoint 1 at 0x400684: file int_overflow.c, line 18.
(gdb) b 22
Breakpoint 2 at 0x4006ab: file int_overflow.c, line 22.
(gdb) b 32
Breakpoint 3 at 0x4006ff: file int_overflow.c, line 32.
(gdb) 
```

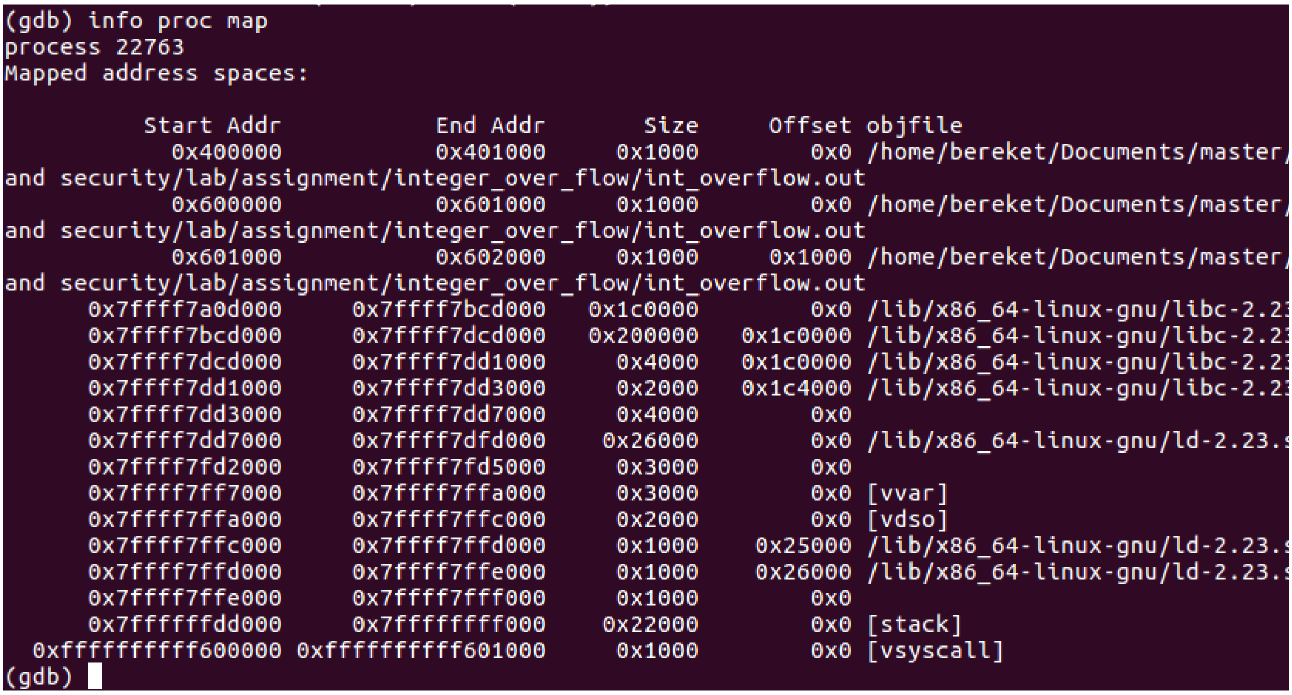
This points are before the creation of the heap, after the heap have been created and after the data have been inputed to the heap.

So when we run we have to input two(2) arguments the first argument is the length and the second argument will be the data. Now we have to know the limit of the unsigned int. And the value is 4294967295, any value that is more than this number will go back to zero and rotate. Since on the code the input have been multiplied by the sizeof(char*)=8. The input should be 4294967295/8=536870912. and we are going to input this value. The data on the second argument will be passed by using a python command line print statement and pass that as input to the second argument to simplify the writing and how much we want to write. The input will be
$(python -c 'print("A"*50+"B"*50)')



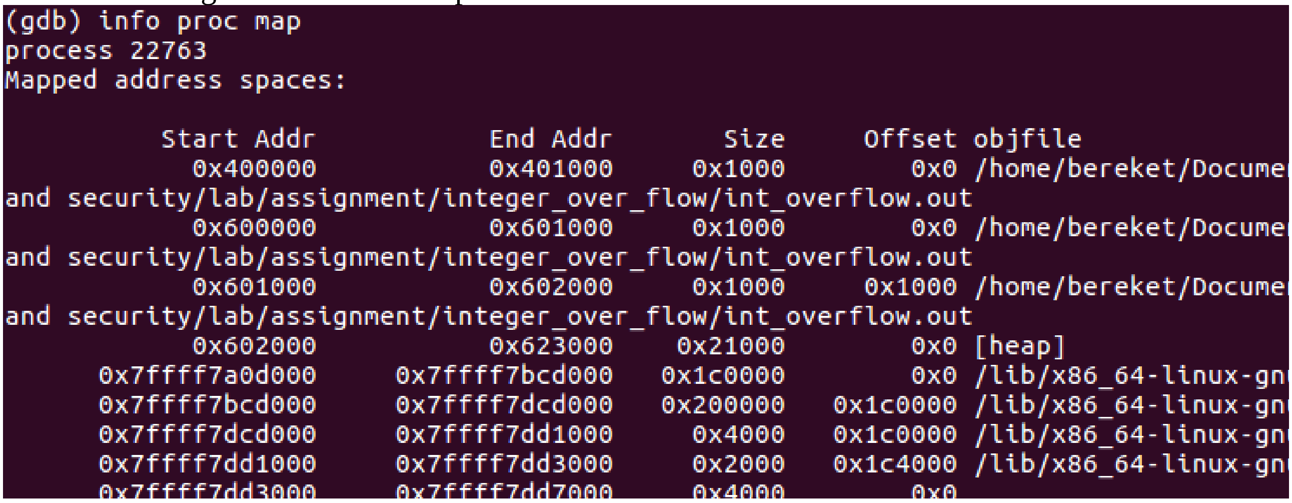
```
(gdb) run 536870912 $(python -c 'print("A"*50+"B"*50)')
Starting program: /home/bereket/Documents/master/programi
curity/lab/assignment/integer_over_flow/int_overflow.out
n -c 'print("A"*50+"B"*50)')

Breakpoint 1, main (argc=3, argv=0x7fffffffdb68) at int_o
19                fun = (char *)malloc(value);
(gdb) 
```

As shown on the above screen shot we are at the first breakpoint and we can view the memory status.



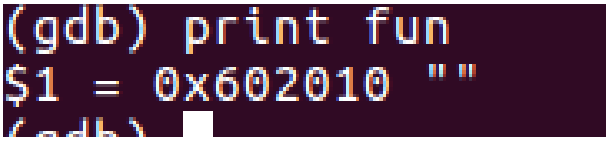
```
(gdb) info proc map
process 22763
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /home/bereket/Documents/master/
and security/lab/assignment/integer_over_flow/int_overflow.out
            0x600000           0x601000     0x1000        0x0 /home/bereket/Documents/master/
and security/lab/assignment/integer_over_flow/int_overflow.out
            0x601000           0x602000     0x1000     0x1000 /home/bereket/Documents/master/
and security/lab/assignment/integer_over_flow/int_overflow.out
      0x7ffff7a0d000     0x7ffff7bcd000   0x1c0000        0x0 /lib/x86_64-linux-gnu/libc-2.23
      0x7ffff7bcd000     0x7ffff7dcd000   0x200000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.23
      0x7ffff7dcd000     0x7ffff7dd1000     0x4000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.23
      0x7ffff7dd1000     0x7ffff7dd3000     0x2000   0x1c4000 /lib/x86_64-linux-gnu/libc-2.23
      0x7ffff7dd3000     0x7ffff7dd7000     0x4000        0x0 
      0x7ffff7dd7000     0x7ffff7dfd000    0x26000        0x0 /lib/x86_64-linux-gnu/ld-2.23.s
      0x7ffff7fd2000     0x7ffff7fd5000     0x3000        0x0 
      0x7ffff7ff7000     0x7ffff7ffa000     0x3000        0x0 [vvar]
      0x7ffff7ffa000     0x7ffff7ffc000     0x2000        0x0 [vdso]
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x25000 /lib/x86_64-linux-gnu/ld-2.23.s
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x26000 /lib/x86_64-linux-gnu/ld-2.23.s
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0 
      0x7ffffffdd000     0x7ffffffff000    0x22000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
(gdb) 
```

The heap is not created as shown on the picture above. So we will go to the next breakpoint and view the starting address of the heap.



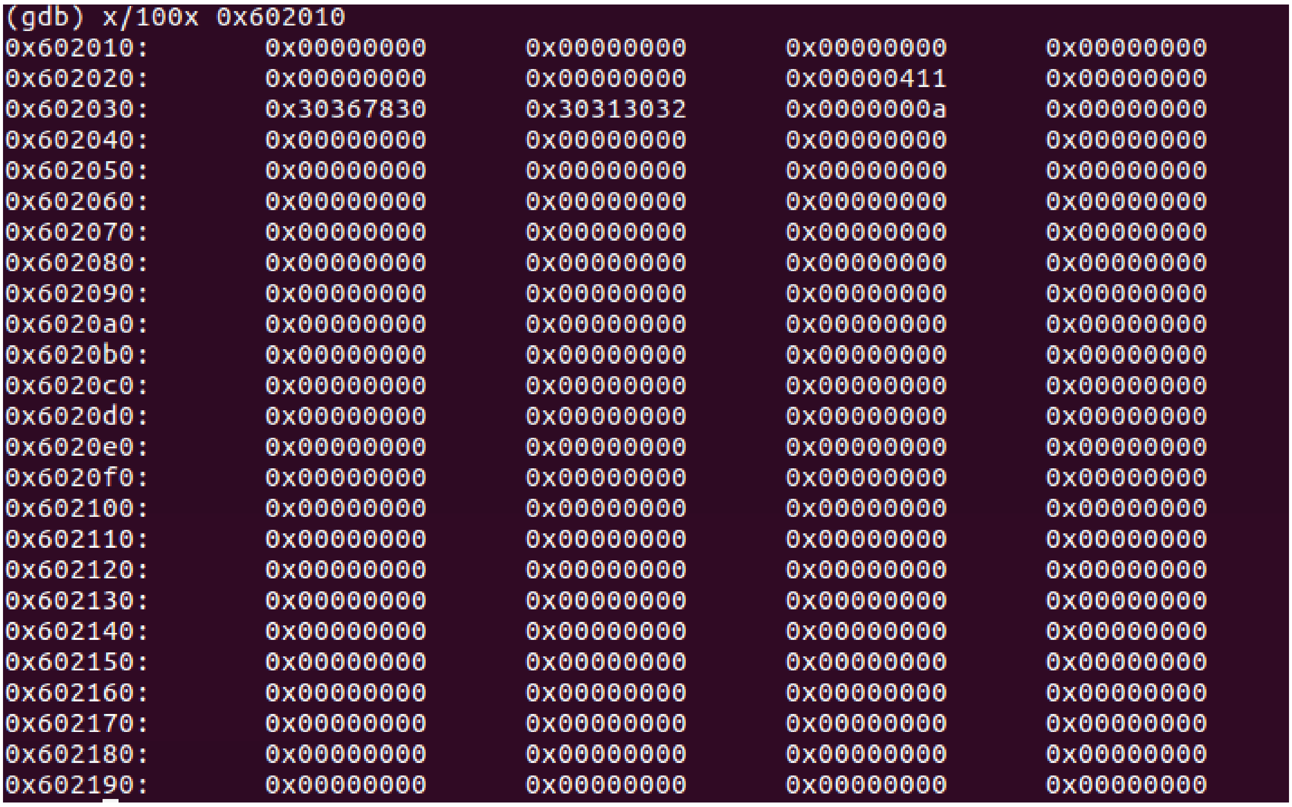
```
(gdb) info proc map
process 22763
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /home/bereket/Docume
and security/lab/assignment/integer_over_flow/int_overflow.out
            0x600000           0x601000     0x1000        0x0 /home/bereket/Docume
and security/lab/assignment/integer_over_flow/int_overflow.out
            0x601000           0x602000     0x1000     0x1000 /home/bereket/Docume
and security/lab/assignment/integer_over_flow/int_overflow.out
            0x602000           0x623000    0x21000        0x0 [heap]
      0x7ffff7a0d000     0x7ffff7bcd000   0x1c0000        0x0 /lib/x86_64-linux-gn
      0x7ffff7bcd000     0x7ffff7dcd000   0x200000   0x1c0000 /lib/x86_64-linux-gn
      0x7ffff7dcd000     0x7ffff7dd1000     0x4000   0x1c0000 /lib/x86_64-linux-gn
      0x7ffff7dd1000     0x7ffff7dd3000     0x2000   0x1c4000 /lib/x86_64-linux-gn
      0x7ffff7dd3000     0x7ffff7dd7000     0x4000        0x0
```

know the heap is created and the starting address of the heap is as shown on the figure. But the exact address of the pointer that we have declared and the heap that is allocated for that pointer can be found by using the **print fun** command.



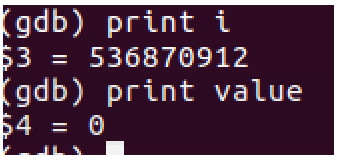
```
(gdb) print fun
$1 = 0x602010 ""
(gdb)
```

As seen the address of fun is at 0x602010 so we will the content of the memory at this address. This is before the data have been written.



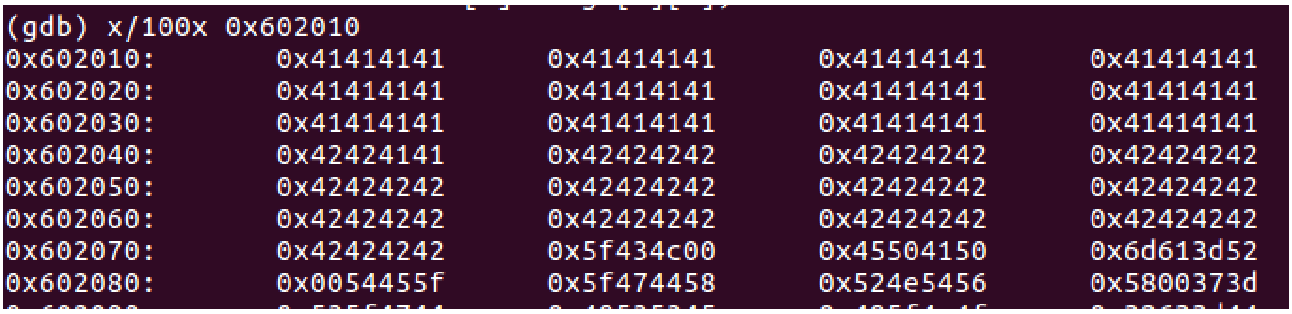
```
(gdb) x/100x 0x602010
0x602010:       0x00000000      0x00000000      0x00000000      0x00000000
0x602020:       0x00000000      0x00000000      0x00000411      0x00000000
0x602030:       0x30367830      0x30313032      0x0000000a      0x00000000
0x602040:       0x00000000      0x00000000      0x00000000      0x00000000
0x602050:       0x00000000      0x00000000      0x00000000      0x00000000
0x602060:       0x00000000      0x00000000      0x00000000      0x00000000
0x602070:       0x00000000      0x00000000      0x00000000      0x00000000
0x602080:       0x00000000      0x00000000      0x00000000      0x00000000
0x602090:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020a0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020b0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020c0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020d0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020e0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020f0:       0x00000000      0x00000000      0x00000000      0x00000000
0x602100:       0x00000000      0x00000000      0x00000000      0x00000000
0x602110:       0x00000000      0x00000000      0x00000000      0x00000000
0x602120:       0x00000000      0x00000000      0x00000000      0x00000000
0x602130:       0x00000000      0x00000000      0x00000000      0x00000000
0x602140:       0x00000000      0x00000000      0x00000000      0x00000000
0x602150:       0x00000000      0x00000000      0x00000000      0x00000000
0x602160:       0x00000000      0x00000000      0x00000000      0x00000000
0x602170:       0x00000000      0x00000000      0x00000000      0x00000000
0x602180:       0x00000000      0x00000000      0x00000000      0x00000000
0x602190:       0x00000000      0x00000000      0x00000000      0x00000000
```

know we will view the values for the allocated size and the value for the iteration of the loop that writes the data to the memory.



```
(gdb) print i
$3 = 536870912
(gdb) print value
$4 = 0
```

“i” is the input that we gave in and used for the iteration of the loop and value is the input multiplied with the sizeof(char*) which is 8*536870912 that wrap around and result to be zero(0). So when we continue and view the heap we can see it has been filled with the input data.



```
(gdb) x/100x 0x602010
0x602010:       0x41414141      0x41414141      0x41414141      0x41414141
0x602020:       0x41414141      0x41414141      0x41414141      0x41414141
0x602030:       0x41414141      0x41414141      0x41414141      0x41414141
0x602040:       0x42424141      0x42424242      0x42424242      0x42424242
0x602050:       0x42424242      0x42424242      0x42424242      0x42424242
0x602060:       0x42424242      0x42424242      0x42424242      0x42424242
0x602070:       0x42424242      0x5f434c00      0x45504150      0x6d613d52
0x602080:       0x0054455f      0x5f474458      0x524e5456      0x5800373d
```

So as it can been shown we have caused a heap overflow.

Question B

Consider the following C code segment. By calling this function from the main function demonstrate the vulnerability of the above code. Show how this code causes heap overflow attack.

```
int copying ( char *buf, int len)
{
char  *newbuf[200];
if ( len > sizeof (newbuf) )
return -1;
return memcpy (newbuf, buf, len);
}
```

The calling function from main is.

```
int main(int argc, char *argv[]){
        int input;
        if (argc<3)
                return -1;
  input = atoi(argv[1]);
  copying(argv[2] , input);
}
```

I have tried the input length in different data type of int. but the result have no vulnerability this is because the use of **sizeof(newbuf)** in the comparison statement of **if** when a comparison between different type (example unsigned and signed) before comparison the values will be converted to the same data type. The conversion will be to the type that have the highest rank. In this code fragment **sizeof(newbuf)** have the highest rank so when entering a negative value the signed negative value will be changed to the unsigned since **sizeof** return unsigned value and the negative value will become a large unsigned number and will never pass the if statement that checks the size.