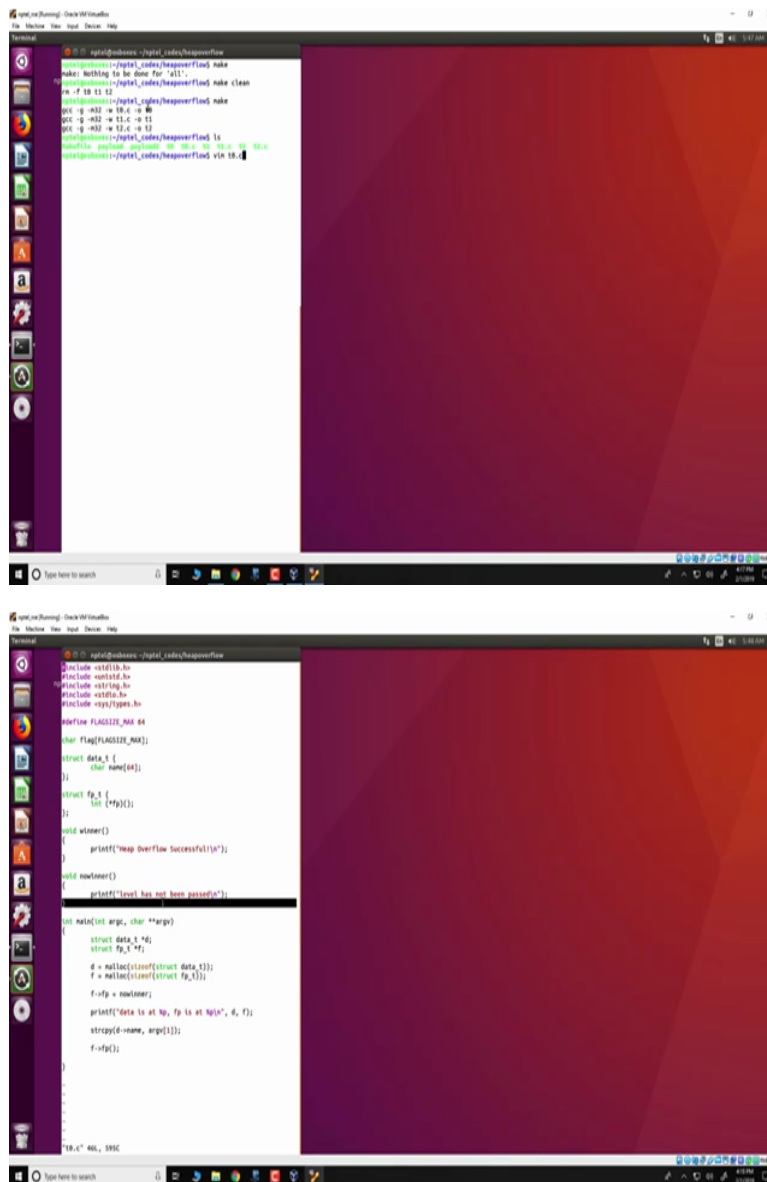


**Information Security - 5 - Secure Systems Engineering**  
**Professor Chester Rebeiro**  
**Indian Institute of Technology, Madras**  
**Heap\_Demo1**

Hello, welcome to this demonstration, so this demonstration is on heaps, we have seen in the previous lectures about how we could overflow the stack and be able to do malicious things by essentially modifying the return address which is stored on the stack, we can also do something very similar by overflowing heap locations. So this very basic introduction to a heap exploit would first take us through how a heap is organized in the program and then we would actually use the exploit.

(Refer Slide Time: 0:52)



```
root@ubuntu:~/infosec/heapoverflow# make
make: Nothing to be done for 'all'.
root@ubuntu:~/infosec/heapoverflow# make clean
rm -f *.o *.a *.elf
root@ubuntu:~/infosec/heapoverflow# make
gcc -g -m32 -o 19.o *.c -D 19
gcc -g -m32 -o 19.o *.c -D 19
gcc -g -m32 -o 19.o *.c -D 19
root@ubuntu:~/infosec/heapoverflow# ls
heapfile  progname  progname.o  19.o  19.elf  19
root@ubuntu:~/infosec/heapoverflow# ./19.elf
root@ubuntu:~/infosec/heapoverflow# ./19.elf
```

```
root@ubuntu:~/infosec/heapoverflow# cat 19.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define FLAG_SIZE_MAX 64
char flag[FLAG_SIZE_MAX];

struct data_s {
    char name[64];
};

struct fp_s {
    int (*fp)();
};

void winner() {
    printf("Heap overflow successful!\n");
}

void loser() {
    printf("Level has not been passed!\n");
}

int main(int argc, char **argv) {
    struct data_s d;
    struct fp_s f;

    d = malloc(sizeof(struct data_s));
    f = malloc(sizeof(struct fp_s));

    f->fp = winner;

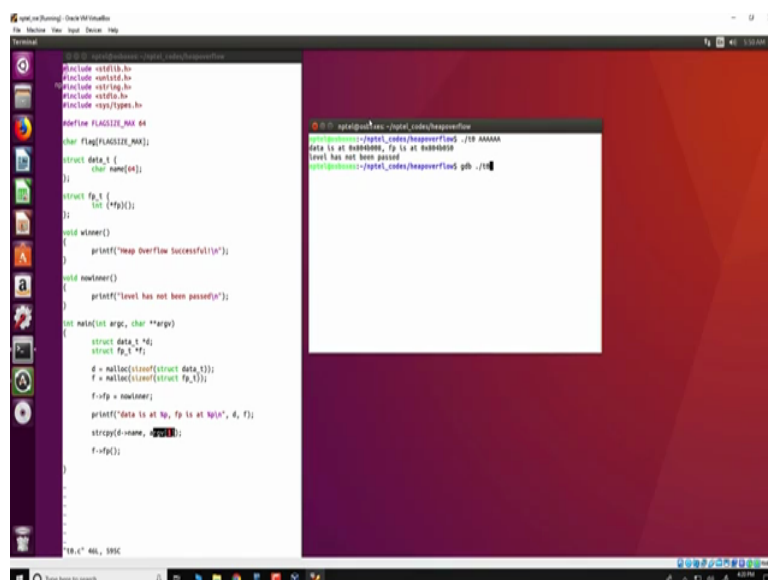
    printf("data is at %p, fp is at %p", &d, f);
    strcpy(d->name, argv[1]);
    f->fp();
}
```

We will be sharing this particular code so you could download this code from your virtual machine box and run this code after compiling it, so in order to run this specific code you could do a make as make clean over here, then make. So what we see here is that there are in fact 3 programs T 0 dot C, T 1 dot C and T 2 dot C in this specific video we will be looking at T 0 dot C.

So let us open it out T 0 dot C, okay so this essentially is a very small C program where we have two malloc statements that gets invoked, one to allocate a structure data underscore T which is of 64 bytes and has name, the other one is F which essentially allocates in the heap this structure fp underscore T, so note that FB underscore T has just one element which is a function pointer fp.

So then what we do is we initialize the function pointer in f to nowinner so know that nowinner is here and it simply prints level has not been passed. Now there is another printf and then a string copy, now string copy uses d name that is d name and copies argument 1 that is a command line argument into d name and then there is a invocation to ffp, so what you would expect over here is first the string copy gets invoked and then the fp function which is pointing to nowinner that would get invoked and you would get this particular printf getting executed that is level has not been passed.

(Refer Slide Time: 2:58)



```
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | #include <stdio.h>
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | #include <stdlib.h>
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | #include <string.h>
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | #include <sys/types.h>
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | #define FLAG_SIZE_MAX 64
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | char flag[FLAG_SIZE_MAX];
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | struct data_t {
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 |     char name[64];
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | };
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | struct fp_t {
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 |     int (*fp)();
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | };
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | void winner()
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | {
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 |     printf("heap overflow successful!\n");
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | }
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | void nowinner()
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | {
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 |     printf("level has not been passed!\n");
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | }
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | int main(int argc, char **argv)
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | {
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 |     struct data_t d;
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 |     struct fp_t f;
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 |     d = malloc(sizeof(struct data_t));
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 |     f = malloc(sizeof(struct fp_t));
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 |     f->fp = nowinner;
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 |     printf("data is at %p, fp is at %p", d, f);
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 |     strcpy(d->name, argv[1]);
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 |     f->fp();
  | ^~~~~~
gcc.c:1:1: warning: ISO C90 requires _ansi_ for prototypes that use typedefs [-Wpedantic]
1 | }
```

```
~/github/level_overflow$ ./T0 AMAMA
data is at 0x8000000, fp is at 0x8000000
level has not been passed
~/github/level_overflow$
```

```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #define FLAG_SIZE 64
6 char flag[FLAG_SIZE];
7 struct data_t {
8     char name[64];
9 };
10 struct fp_t {
11     int **fp();
12 };
13 void winner() {
14     printf("Heap overflow Successful!\n");
15 }
16 void newton() {
17     printf("level has not been passed!\n");
18 }
19 int main(int argc, char **argv) {
20     struct data_t *d;
21     struct fp_t *f;
22     f = malloc(sizeof(struct fp_t));
23     f->fp = newton;
24     printf("data is at %p, fp is at %p", d, f);
25     strcpy(d->name, argv[1]);
26     f->fp();
27 }
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

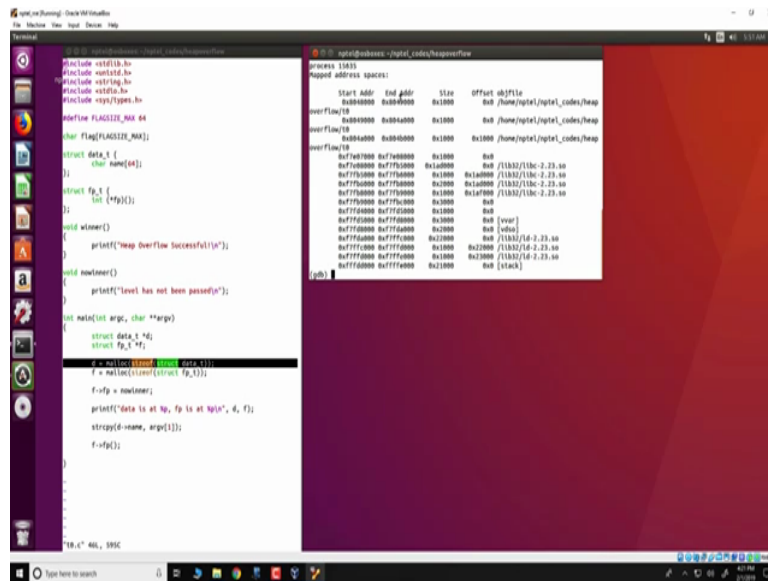
```

So let us see that happening so we run T 0 and give it a few inputs like this a few a short length argument and what we see is that it gets printed that level has not been passed. Okay, so before we go into how to actually use the exploit and by now if you have been following the course then you would have figured out that the exploit is due to this vulnerability in the string copy, note that the string copy copies something from argument 1 into d name so d name is a fixed length of 64 bytes and argument 1 is controlled by the user through the command line arguments, so the argument 1 can cause d name to overflow if it has a length which is greater than 64 bytes.

So let us just go through how this program executes with the right input first, so we will as you shall run gdb with T 0 as input list thing as usual put a breakpoint in line number 34 and we run this program giving the same shot input string, there are five to six A over here. So what we have stopped as we have seen before in the previous videos is that we have stopped at line number 34 and importantly right now there is no malloc has been called as yet.

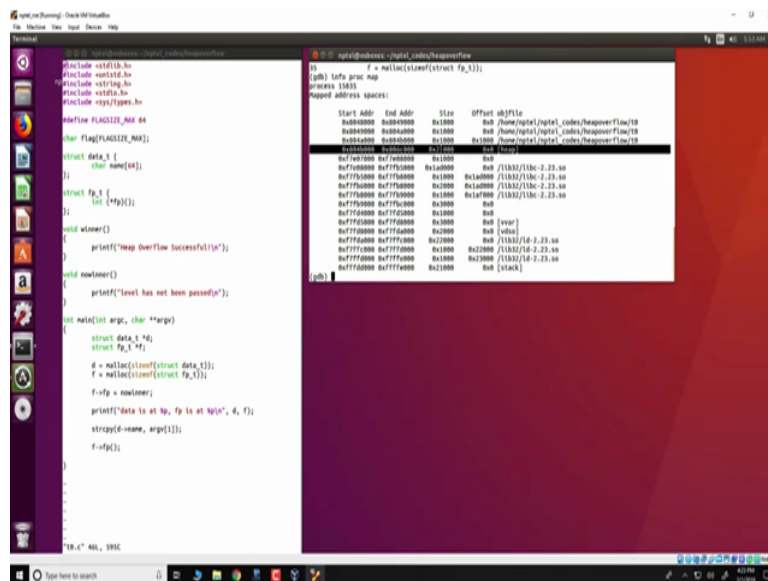
As we have seen in the theory classes since the malloc has not been invoked as yet, the initial size of the heap is 0, in fact there is no heap present in this particular program at this particular point during the execution.

(Refer Slide Time: 5:02)



So we can see this by this info propmap command and what we look at over here as we seen in the theory is that we have the virtual address space for this process, we see the various segments in the process like which are defined front by the start address and the end address.

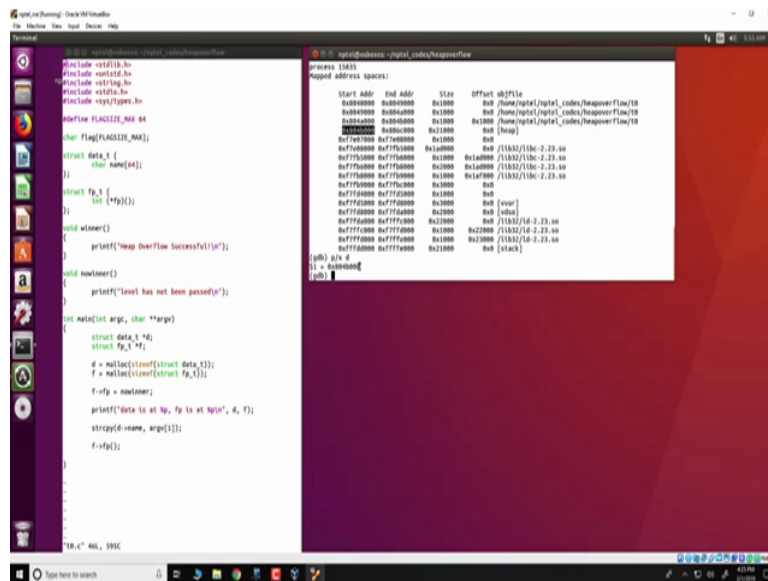
(Refer Slide Time: 5:22)



So we have the various segments for the T0 program we have the libc, stack, vds0 and so on. So what is missing here you would notice is that at this particular time since we have not execute any malloc as yet there is no heap that is present. So we will single step to another line and see that the malloc has actually been executed, we can now run this info propmap again and we would see that a heap has now been assigned to the program.

So what has happened internally is that rather since this was the first malloc that is invoked in this particular program the PT malloc code which has got linked with this particular program through libc has invoked the operating system and it has requested for a large chunk of memory of 132 kilobytes. So this memory has got attached to the locations 804b000 to 806c000 and the size is 132 kilobytes, so you can verify that this size 21000 which is in hexadecimal notation over here is in fact 132 kilobytes.

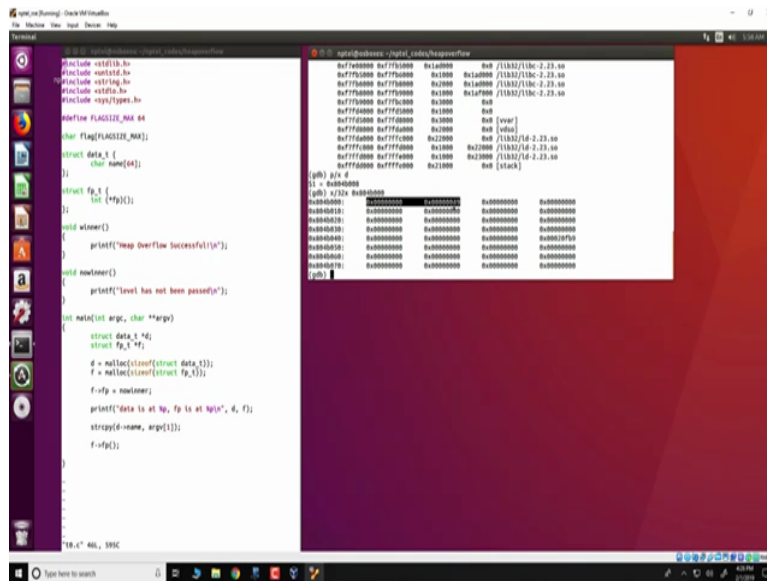
(Refer Slide Time: 6:38)



So now we will also look at what the address of `d` is `xd` and what we see is that `d` has a value `804b008`. So what PT malloc has done internally is that since you have invoked the function requesting a structure data underscore T to be allocated in the heap, so PT malloc code once it has obtained the 132 kilobytes of memory would split this memory into two components, one component which would be slightly larger than 64 bytes because data underscore T is 64 bytes and the remaining component is the free chunk of memory which has not yet been utilized, what we see over here is the address of `d` we see that it is an offset of `804b008`.

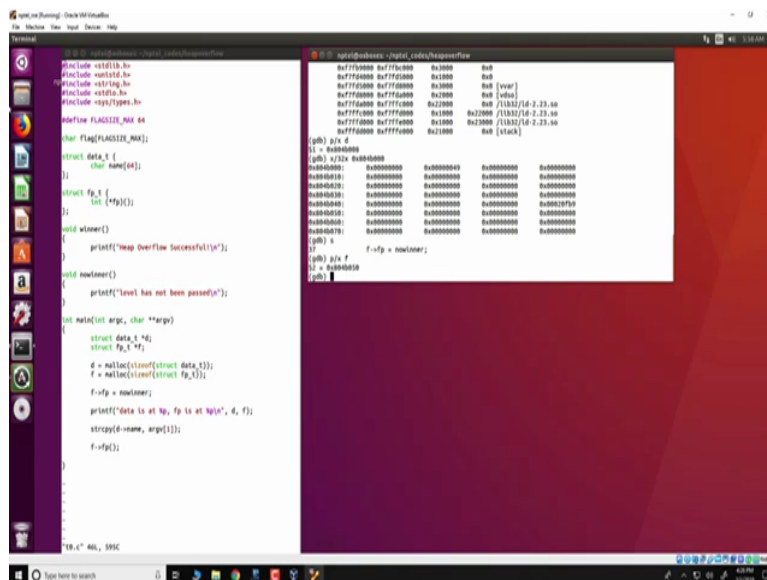
Now if you compare this with a start location of heap, you see that it is 8 bytes from the starting location of the heap, the bytes `804b000` to `804007` contains the metadata for this particular chunk of memory.

(Refer Slide Time: 8:00)



Now we can look at this metadata by executing something like this we are just dumping the heap location and we could specify the address `0x804b000`, so in this particular time we have the metadata which will be stored over here currently it has a value of 49, so 49 is in hexadecimal `01001001`, the LSB of this is 1 indicating that the previous is in use and 100 would indicate the size that is allocated for this chunk so this would be slightly greater than 64 bytes and this you could actually validate later, okay.

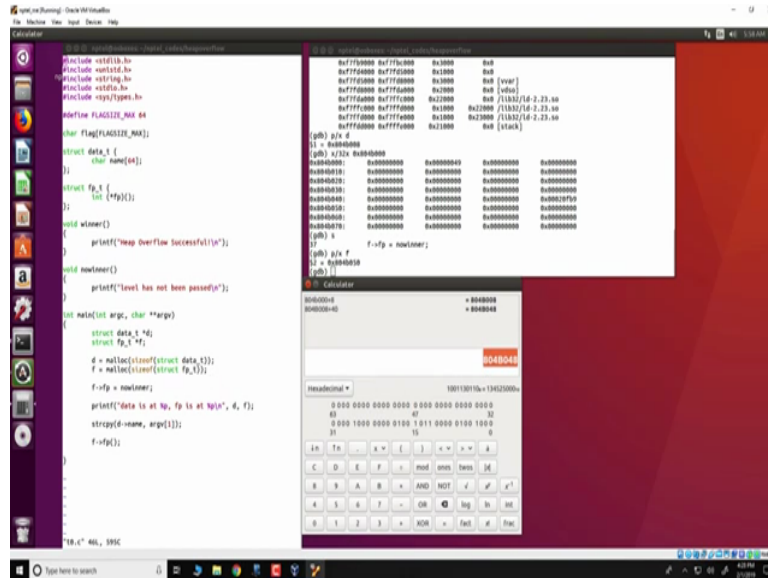
(Refer Slide Time: 9:06)



So once we single step further we would also see that the `f` would get allocated, so we look at the address part `f e slash x` or to obtain the thing for `f` and we would see that `f` is at a location

804b050, so note the offset of f from the base of the heap and we will compute actually what happens internally.

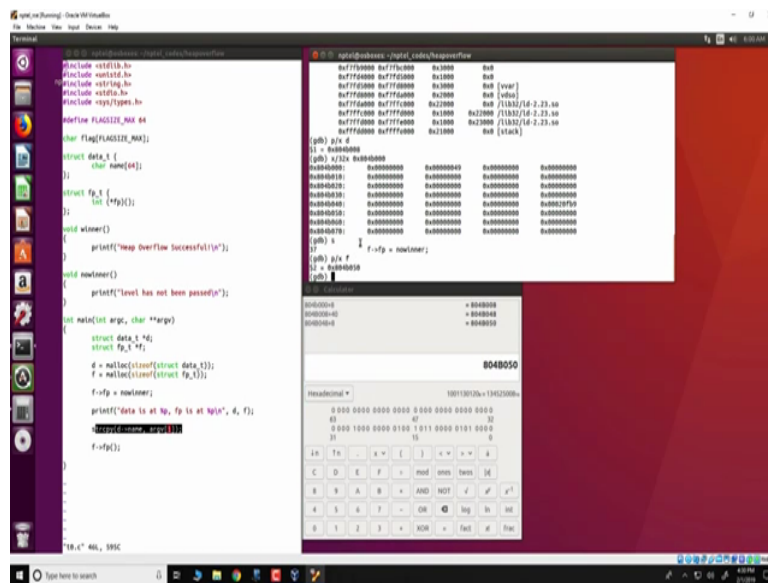
(Refer Slide Time: 9:30)



So starting from the base so we first move this thing into the programming mode and move to the hexadecimal notation and let us start out from the base and we see that the base of the heap is 804b000 and the initial eight bytes comprises of the metadata for the first memory allocation that is for d that is so it would be 8 bytes so 804B008 gives you the memory address for d.

Now since the size of struct data underscore T is 64 bytes, so therefore the PT malloc would have added 64 bytes to this, so 64 in hexadecimal notation is 40, so this is 804B048, so this location onwards signifies the end of d.

(Refer Slide Time: 10:41)



The screenshot shows a debugger window with a code editor on the left and a memory dump on the right. The code editor displays the following C code:

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

#define FLAG_SIZE_MAX 64
char flag[FLAG_SIZE_MAX];

struct data_s {
    char name[4];
};

struct fp_s {
    int (*fp)();
};

void winner() {
    printf("Heap overflow Successful!\n");
}

void newLower() {
    printf("level has not been passed\n");
}

int main(int argc, char **argv)
{
    struct data_s d;
    struct fp_s f;

    d = malloc(sizeof(struct data_s));
    f = malloc(sizeof(struct fp_s));

    f->fp = newLower;

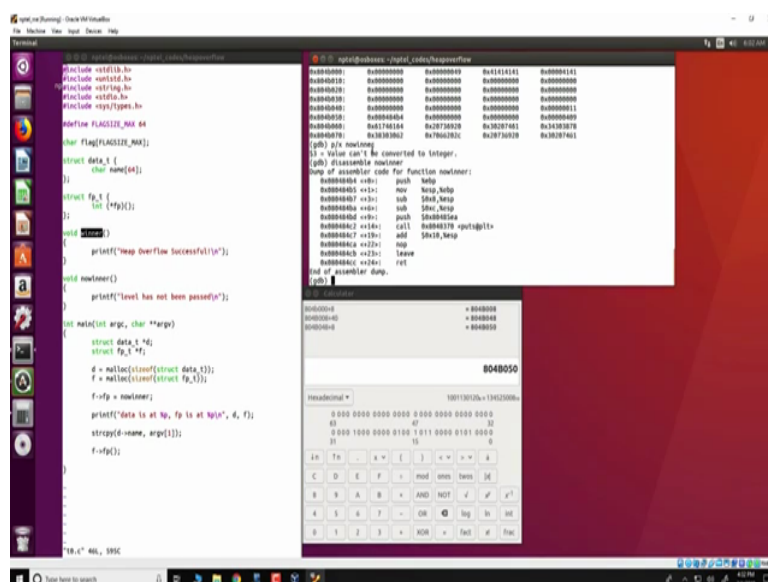
    printf("data is at %p, fp is at %p\n", d, f);

    strcpy(d->name, argv[1]);
    f->fp();
}
```

The memory dump on the right shows the addresses of variables `f` and `d`. Variable `f` is located at `0x00400000` and contains a pointer to `newLower` at `0x00400000`. Variable `d` is located at `0x00400010` and contains the string "level" at `0x00400010`. The address `0x00400008` is highlighted, which corresponds to the address of `f` plus 8 bytes of metadata.

Now the next allocation is `f`, now `f` also has metadata the it has two words of metadata and therefore it will also have 8 bytes of metadata. So if we add 8 to this, we will get `804B050` which precisely is what is the address of `f`. So in this way what we see is that we have seen how the memory is organized at this particular point in time, it has contiguous chunks of `d` and `f` which has placed side by side and the only thing which separates them is some metadata information for the `f`.

(Refer Slide Time: 11:47)



The screenshot shows the debugger window with the assembly code for the `newLower` function. The assembly code is as follows:

```
newLower:
push    ebp
mov     ebp, esp
sub     esp, 4
mov     eax, 0
mov     ecx, esp
mov     edi, esp
mov     esi, esp
push   ecx
call   @plt@%ebx@PLT
add     esp, 4
mov     eax, esp
mov     ecx, esp
mov     ebx, esp
mov     ebp, esp
ret
```

The memory dump on the right shows the addresses of variables `f` and `d`. Variable `f` is located at `0x00400000` and contains a pointer to `newLower` at `0x00400000`. Variable `d` is located at `0x00400010` and contains the string "level" at `0x00400010`. The address `0x00400008` is highlighted, which corresponds to the address of `f` plus 8 bytes of metadata.

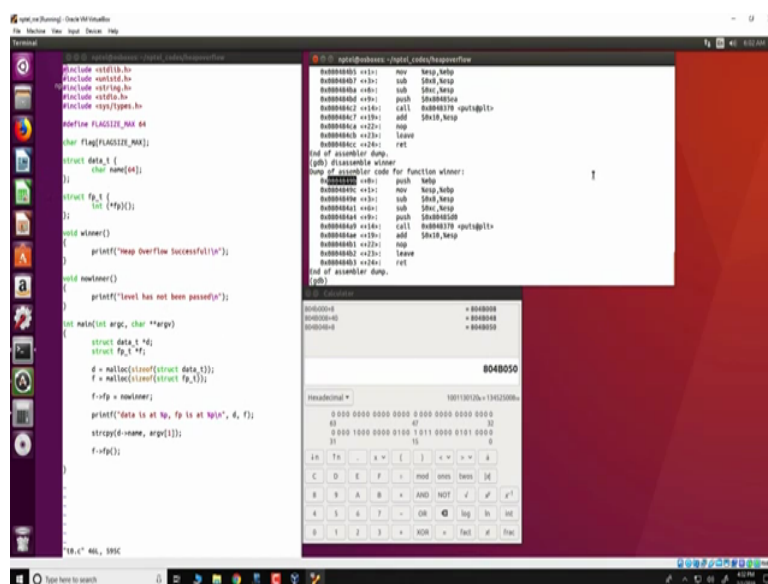
Now we will see how we can actually overflow `d` using this string copy and passing an argument which is longer and how we could actually modify the contents of `f`. So first of all let us continue to a single step and since we have given a small input we can go through



string copy and we can now look at the heap again and what we see is that since our input is all is a couple of A's and A has ASCII value of 41 in hexadecimal, so those values are actually stored from this location onwards.

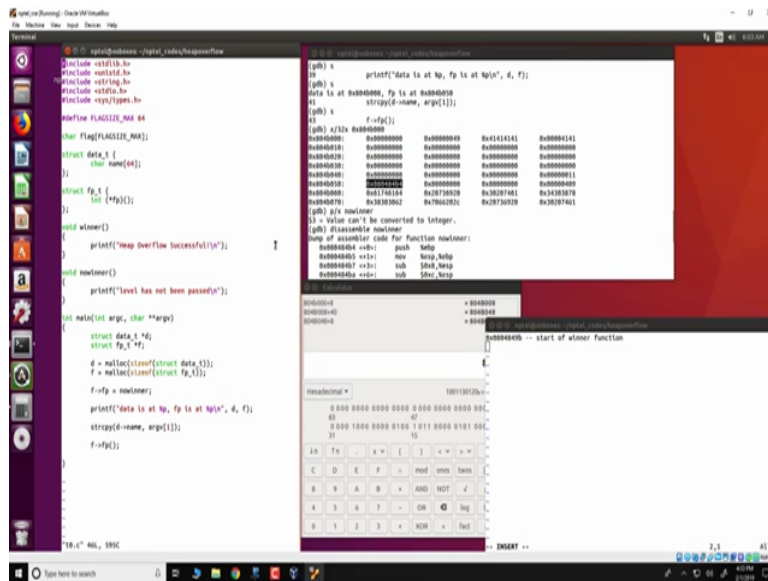
Now since data underscore t has a size of 64, so 64 bytes from here would end the structure data underscore t and then we would have f and f since we have initialized to no winner the value of no winner is present here. So we can see that 080484B4 in fact specifies the address of nowinner so that we can check like this v / x nowinner is assemble nowinner and what you would see is the start address of nowinner is indeed 080484B4.

(Refer Slide Time: 13:22)



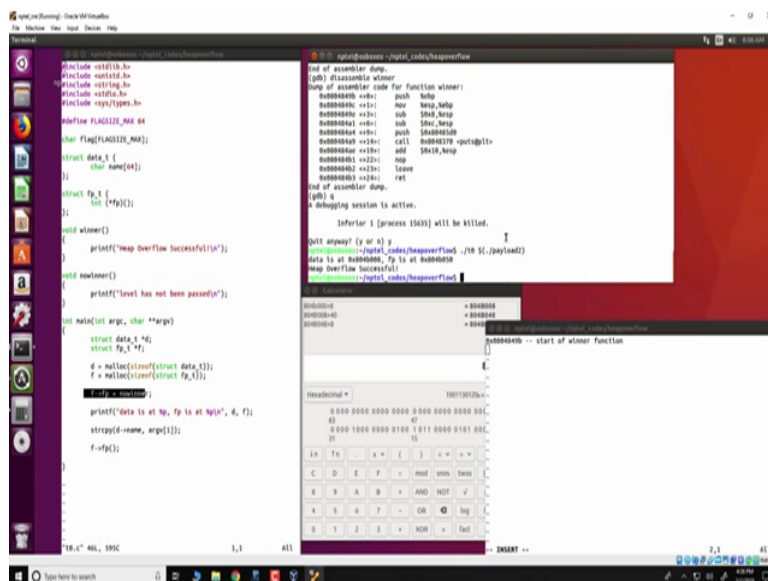
Now what we want to do is that we want to modify this program we want to subvert the execution so that this address in the heap which corresponds to nowinner is modified to that of winner. In order to do this we would first require to determine the address of winner so that would can be obtained by disassemble of the winner function and you would see that the winner function starts at the address 0804849B, so we could actually write this down somewhere.

(Refer Slide Time: 13:41)



So what we have done is we have copied the start address of the winner function I know that we need the start address to override the nowinner function location present in the stack. So let us see how we do this from here things are quite similar to what we have done so far in the past in this particular course, we can first determine that the amount of in order to overflow this particular data underscore t structure we would require 72 inputs before all of this gets filled up and then we need to specify the address for our malicious function which in this case is winner.

(Refer Slide Time: 14:32)



So we have already done this for you, so this is present in payload 2, so you see over here that what payload 2 does is that it creates 72 A's followed by the address 0804849B so this as you

can recollect is the little Indian notation. So we could run this particular program as follows, so we have simplified it for you all you all that you need to do is run T0 specify a dollar and then dot slash payload 2 and this would result in this payload 2 getting executed the string 72 A's followed by this address of winner would be then passed to T0 and we see that it has resulted in a heap overflow successful. So this happens because the winner function gets executed.

So what we have done is that we have essentially overflowed d with a lot of A's in fact 72 A's and then specified the address of the nowinner function which essentially replaces this invocation this fp thing with winner thus the winner function would get executed. So there are a lot of interesting attacks you can actually do with the heap, but the and some of them are actually uploaded to the website for this particular course and in the next video lectures we look at some more things specifically with the heap, thank you.