

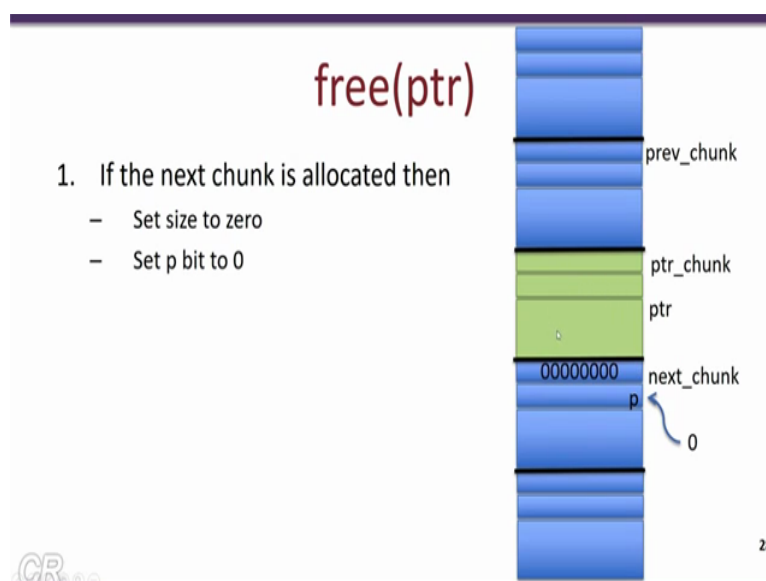
Information Security 5 Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Lecture 24
Heap Exploits

Hello and welcome to this lecture on heap exploits, so in the previous lecture we had looked at some of the internals about how ptr malloc manages the heap memory. In this lecture we will look at vulnerabilities that may occur due to this heap management and we will also see one particular exploit and look at how it works. So in the previous lecture we stopped off at how malloc uses the various bins and how these various bins store linked lists headers and this link list to be either single or doubly linked lists.

In whenever a request occurs for a chunked of memory, then malloc would look into these linked lists and allocate a chunk of memory to that request depending on the amount of memory that gets requested. Now today we will start the lecture with the free functions, so essentially what could happen when the free function gets invoked as you know the free function would take a pointer.

So internally ptr malloc would free the chunk and the allocated chunk would now become a free chunk. So let us look at details about the free function called.

(Refer Slide Time: 01:39)



So let us say that this was our heap and as we know that the heap comprises of various chunks, so these are like the various chunks that are present and let us say now that we want to free a particular pointer. So as we have seen in the previous lecture but this particular pointer would be actually pointing to a allocated chunk in the heap, so this allocated chunk comprises of the actual data which is present corresponding to that region and also some particular (mem) metadata which is also present four bytes and eight bytes before the starting pointer ptr.

So consider this particular figure which shows a part of the heap as we know the heap is actually divided into various chunks which are either allocated or freed now let us consider this free pointer statement, so as you know pointer would be a chunk of memory present in the heap out of which some areas would be the data corresponding to that particular pointer and the other areas would be some metadata that corresponding to size and type of a chunk and so on.

So when free pointer is called the first thing that would happen is that the ptr malloc free code would first set the size of the free chunk to 0, so as you can be collect from the previous lecture the size of the chunk is present in the metadata, so this metadata is set to 0 next the p bit is set to 0, so the p bit corresponds to the next chunk which is present in the heap and as you can recollect from the previous lecture the p bit stores whether the previous chunk was allocated or freed.

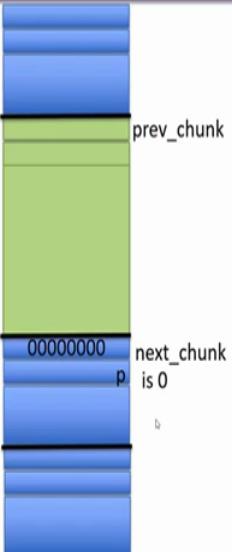
So therefore what would happen over here is that based on this pointer the ptr malloc code would obtain the location where p is present and set that particular value of p to 0 indicating that this chunk is free.

(Refer Slide Time: 04:03)

free(ptr)

2. If the previous chunk is free then
 - Coalesce the two to create a new free chunk
 - This will also require unlinking from the current bin and placing the larger chunk in the appropriate bin

Similar is done if the next chunk is free as well.

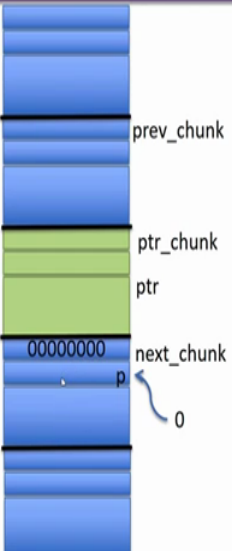


The diagram shows a vertical stack of memory blocks. At the top, there are several blue blocks. Below them is a green block labeled 'prev_chunk'. Underneath 'prev_chunk' is a blue block with '00000000' and 'p' written on it, and 'next_chunk is 0' to its right. Below this is another blue block, and then more blue blocks at the bottom.

29

free(ptr)

1. If the next chunk is allocated then
 - Set size to zero
 - Set p bit to 0



The diagram shows a vertical stack of memory blocks. At the top, there are several blue blocks. Below them is a blue block labeled 'prev_chunk'. Underneath 'prev_chunk' is a green block labeled 'ptr_chunk' with 'ptr' written below it. Below 'ptr_chunk' is a blue block with '00000000' and 'p' written on it, and 'next_chunk' to its right. A blue arrow points from the 'p' to a '0' below it.

28

For certain types of bins coalescing is possible, so what we mean by coalescing is that the ptr malloc code during the free function call could join various adjacent free chunks of memory, for example let us say that the previous chunk over here was actually free as well and this chunk to was also free, in such a case what ptr malloc can do is it can coalesce these adjacent chunks and form a much larger free chunk of memory.

So in this way the fragmentation can be reduced the sum of this large free chunk can now be used to service possibly many more malloc requests. So performing this coalescing also has a slight overhead of requiring to unlink from the current pin and placing the larger chunk in the appropriate bin depending on the size of this free chunk. So during this free function call as

well as during the malloc function call an important function that plays a role as we will see later is something known as the unlink function.

(Refer Slide Time: 05:20)

Unlinking from a free list

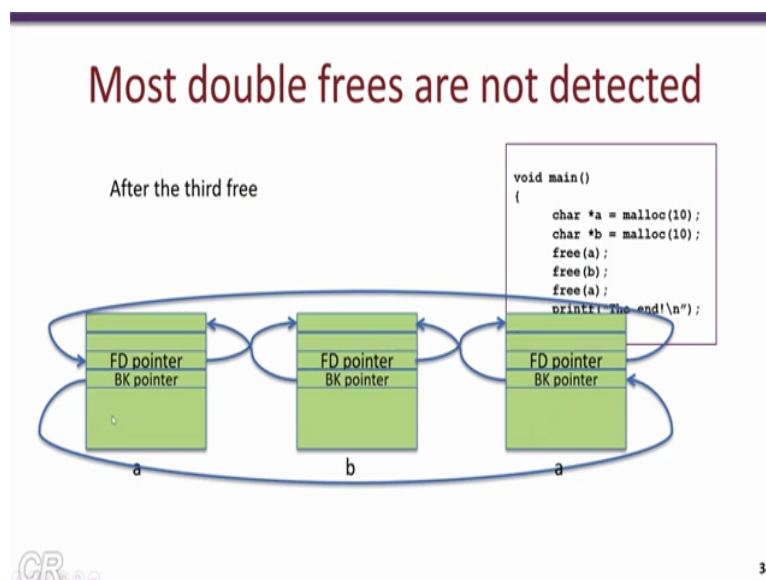
```
void unlink(malloc_chunk *P, malloc_chunk *BK, malloc_chunk *FD){
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}
```

CR 30

So the unlink function is the typical linked list type of operation where do you have a double linked list and during the malloc or during the free when you want to remove a linked list during coalescing or during malloc when you actually want to allocate this chunk of memory to a particular malloc request you will have to adjust the pointers present in this job, so essentially the forward pointer and the backward pointer should be appropriately corrected.

So for example we want to allocate this particular chunk then we ensure that the previous node or the previous chunks forward pointer points to the next chunk forward pointer similarly the next chunks back pointer points to the previous chance pointer.

(Refer Slide Time: 06:16)



Now let us look at this small program and see what can actually go wrong. So in this particular program what we have done is that we have malloc two memory regions of 10 bytes each and allocated them to pointers a and b respectively then we have freed a we have freed b and then we have freed a again, so note that a is freed twice so what can go wrong with this first of all note that your compiler will not be able to determine that your free a is invoked twice, secondly you will also notice that ptr malloc two will not be able to determine that the memory chunk a is freed twice.

Now let us look what happens internally when such a program executes, first as we know when a and a gets invoked the freed chunk of memory is added to the linked list next when free b gets invoked you would have a second chunk of memory added to the linked list now the link list pointers are appropriately adjusted so that the forward pointer corresponding to a wants to b and back pointer of b points to a and vice versa as well.

Now thirdly what would happen when you invoke free a again in such a case since ptr malloc two cannot identify that a is being freed for the second time it would simply add a third node into the linked list, so note that our linked list virtually has three elements a, b and a. Now essentially what is happening here is these two locations are what are actually the same this corresponds to the chunk a of which is freed for the first time and this corresponds to the chunk a again which is freed for the second time, so these two are in fact the same memory location.

(Refer Slide Time: 08:31)

Another malloc

Another malloc
c gets allocated the same address as a

```
void main()
{
    char *a = malloc(10);
    char *b = malloc(10);
    char *c;
    free(a);
    free(b);
    free(a);
    c = malloc(10);
}
```

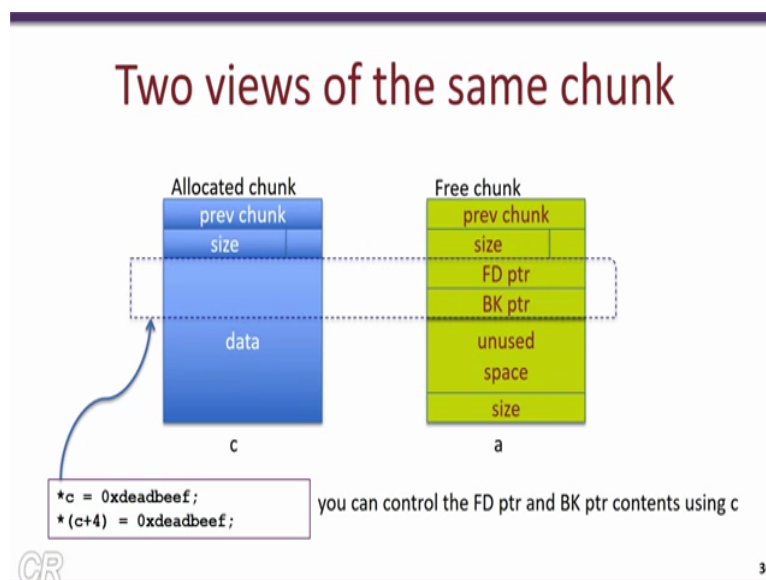
```
chester@aahalya:~/sse/malloc$ ./a.out
a=09108008
b=09108018
c=09108008
```

CR 35

Now consider the small change in the program where we actually invoke another malloc of 10 bytes and allocate the pointer to see. Now as we know what malloc would do is that it would look into the link list and it would pick one of the chunks which is available, so in this case it would actually pick the first chunk which is a and thus what you would see and what is expected is that a as well as c would point to the same memory chunk, so both of them would have the same value stored in them.

The difference between a and c is that a is freed and it is present in the linked list and c is allocated and it is an allocated chunk, so therefore what we have here is that virtually we have one chunk which is configured to be freed that is the a chunk and the same chunk is also configured by ptr malloc and allocated to c.

(Refer Slide Time: 09:38)



Now if we look at the difference between the allocated chunk and the free chunk we see the following. The allocated chunk looks something like this do you have the metadata over here corresponding to previous and the size and the various bits like nnp and you have the data which is present here while the free chunk comprises of the metadata as before the previous chunk and the size and the various flags but importantly we have the forward pointer and the back pointer present in the free chunk.

So as we know this forward and back pointer is used for managing the linked list. Now what would happen if we have statements such as this star c equal to deadbeef and star c plus 4 equal to deadbeef, so essentially this data gets written into these locations note that c and a are essentially the same location, c and a are pointing to the same memory jump therefore when we have operations like c and star c and star c plus 0 what you are essentially doing is modifying the forward pointer and the back pointer essentially when we look at this from a linked list perspective what you would notice is that we are modifying the linked list pointers.

So by appropriately setting values of star c and star c plus 4 we can modify these forward and back pointers and we could force the code to run a specific payload in other words we can actually force and exploit to execute. Now we will take a small example of such a code.

(Refer Slide Time: 11:26)

Exploiting

```
char payload[] =
"\x33\x56\x78\x12\xac\xb4\x67";

Void fun1() {}

void main()
{
    char *a = malloc(10);
    char *b = malloc(10);
    char *c;

    fun1();
    free(a);
    free(b);
    free(a);
    c = malloc(10);
    *(c + 0) = GOT entry-12 for fun1;
    *(c + 4) = payload;
    some malloc(10);
    fun1();
}
```

Need to lookout for programs that have (something) like this structure

We hope to execute payload instead of the 2nd invocation of fun1();

37

So this particular example over here is a very small code which shows how one could exploit a double free. So let us assume that we have a payload which is present here and this payload as we have seen before comprises of some hexadecimal codes which can be interpreted by the processor and will execute, in this payload we could have various things like a shell code or any other malicious code which we want to execute in that particular system.

So let us see how a double free can be used to subvert execution and force this particular payload to execute, let us assume we have a program that looks something like this. So what we have here is that we have a and b which gets pointers pointed to by two chunks of memory which is of 10 bytes each and then what we do first is invoke this function 1 it is some arbitrary function which is not important for us then we have the double free as we have seen before that is the free a followed by free b and then free a and then we have the c equal to malloc 10.

So as we have seen in the example in the previous slide this c and a would have pointers which point to the same memory chunk a is a freed memory chunk and therefore it has a forward and back pointers corresponding to the link list management well c is an allocated chunk, so once the chunk is allocated we can then modify the data present in them, so over here we have just hard coded the code but in reality you could actually have this by using say a scanf or from a network packet or things like that where you could actually modify the contents of c.

So what we modify it is with over here is the GOT entry minus 12 for function 1, so please refer to the ASLR lectures to find out what the GOT entry means and star c plus 4 is the address of the payload. Now what we do is some arbitrary malloc we invoke here and invoke function 1, so note that so now note that the GOT entry for function 1 is used to determine the address or the location of function 1.

Now if we somehow managed to change the GOT entry corresponding to function 1 we will also be able to change the instructions that get executed when function 1 gets invoked what we will be seeing in this particular exploit is that we are going to change the entry for we are going to change the GOT entry for function 1 and replace it with the payload, so thus what we see is that when function 1 gets invoked over here it would be the payload that gets executed and thus we are able to achieve a subverting of the execution using the double free of the a that has occurred.

(Refer Slide Time: 14:51)

Exploiting Heap

```
char payload[] =
"\x33\x56\x78\x12\xac\xb4\x67";

void fun1() {}

void main()
{
    char *a = malloc(10);
    char *b = malloc(10);
    char *c;

    fun1();
    free(a);
    free(b);
    free(a);
    c = malloc(10);
    *(c + 0) =GOT entry-12 for fun1;
    *(c + 4) = payload;
    some malloc(10);
    fun1();
}
```

→ Payload executes

CR

42

Exploiting

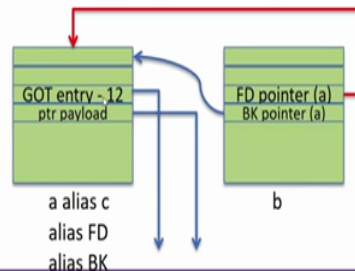
```
char payload[] =
"\x33\x56\x78\x12\xac\xb4\x67";

Void fun1() {}

void main()
{
    char *a = malloc(10);
    char *b = malloc(10);
    char *c;

    fun1();
    free(a);
    free(b);
    free(a);
    c = malloc(10);
    *(c + 0) =GOT entry-12 for fun1;
    *(c + 4) = payload;
    some malloc(10);
    fun1();
}
```

```
unlink(P) {
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}
```



41

So we will start of from this particular point and what we see over here is that we have this double linked list in which there are 3 entries a, b and a but importantly for us this a and this a are essentially the same, both of them are actually the same location so when we do malloc of 10 again and assign that particular pointer to see what we obtain is that the linked list now reduces to 2 and we have a which is freed and c which is also allocated and both of them point to the same memory chunk.

So now we are setting star c plus 0 and star c plus 4 to some GOT entry minus 12 for function 1 and payload, so therefore we are now breaking this particular link list. In see in where the forward pointers is supposed to be we are putting the GOT entry and where the back pointer is supposed to be we have putting the pointer to the payload. Now what could happen when malloc gets invoked over here?

Now when malloc gets invoked yet again what would happen is that the payload present in the back pointer gets written into the GOT entry minus 12 thus when function 1 gets invoked it will first look up the got entry but instead of getting the actual value of the address of function 1 it will get the modified address which is actually pointing to the payload thus when function 1 executes it would be this payload that gets executed, thus we are able to subvert execution and cause the payload to execute.

(Refer Slide Time: 16:43)

Ponder About


```
char *secret = "THIS IS A SECRET MESSAGE!";

int main(int argc, char **argv){
    int *a, *b, *c, *d, *e;

    a = malloc(32); /* S1 */
    b = malloc(32); /* S2 */
    c = malloc(32); /* S3 */
    free(a); /* S4 */
    d = malloc(32); /* S5 */
    free(b); /* S6 */
    free(d); /* S7 */
    e = malloc(32); /* S8 */
    my_malicious_function(e); /* S9 */
    a = malloc(32); /* S10 */
    printf("%s", a); /* S11 */
}
```

What does the heap look like after each statement S1 to S10 has completed execution?

Show how a malicious function `my_malicious_function` can be written so that S11 prints the secret message.

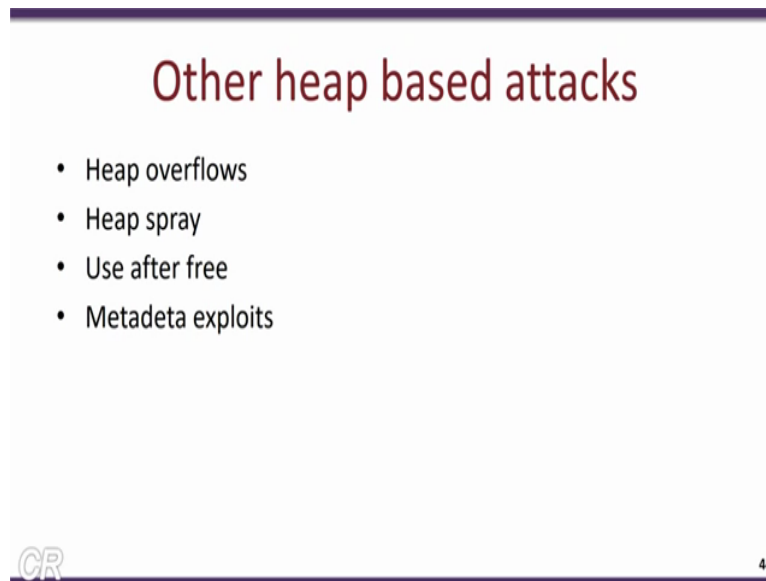


CR 43

Now this is something for you to think about, so over here we have a program which has several malloc and frees important for us is this function my underscore malicious underscore function which takes the pointer e further what you need to think about is to determine how you would write this my malicious function, so that when printf gets invoked over here it is this secret message get that gets printed on the screen.

So in order to think of this solution you must think about how the various lists are managed by the arena and by the heap and then it would not be very difficult for you to actually find a solution for this.

(Refer Slide Time: 17:24)



So the double free is just one form of attack for malloc there are various other forms like heap overflows, heap sprays, use after free and other (meat) metadata exploits, so but most of them follow the same kind of principle there the management of the free chunks and the allocated chunks are essentially manipulated so that the execution gets subverted and the payload (exec) executes.

Further on many of these attacks also just figure out a way to leaked information from other locations in the heap. So with this we will actually wind up this particular lecture, so we had seen in this last two lectures about how heaps are managed and how you could actually use vulnerabilities in the program such as a double free vulnerability to exploit the heap, thank you.