Understanding Stack Corruption With C examples

Discussion in 'C' started by poornaMoksha, Nov 23, 2011.

Tags: stack corruption

poornaMoksha
New Member

JOINED:	Jan 29, 2011
MESSAGES:	150
LIKES RECEIVED:	33
TROPHY POINTS:	0
OCCUPATION:	Software developer
LOCATION:	India

If you are an experienced C/C++ programmer then you must have definitely observed some weird behaviors of code in certain situations. These weird behaviors can be due to various reasons but most of them are because of stack corruption. For all those who have no idea about what stack corruption is, lets discuss it here.

Stack Corruption

Stack corruption is a phenomenon in which some memory locations at stack are accessed unintentionally due to wrong coding leading to change in values at those memory locations. Since the data corruption happens on stack memory locations, hence the term Stack Corruption.

There can be quite a few ways in which stack corruption may occur :

1. When due to some weirdly written code, all the stack memory gets eaten up
2. Accessing array out of bounds
3. An undefined/freed pointer pointing or storing a garbage stack address.
4. When due to some reason, the return address for a function call gets corrupted. (We will take this up as a separate article on stack exploits)

Lets discuss first three one by one.

Consuming all the stack memory

Consider the following code :

Code:

```
#include<stdio.h>

 int main(void)
```

```
{
    int a=0;
    a += 1;
    main();

    return 0;
}
```

In the above piece of code, we declare a variable 'a' inside main() and recursively call main() again and again. Now since every time main() gets called, a memory chunk on stack is allocated for main() again and again. Now, lets run the code :

Code:

```
~/practice $ ./rcrsn
 Segmentation fault
```

Here we see that as soon as I run the above program, I get a seg-fault/crash. Lets modify the code a bit and add a debug printf statement :

Code:

```
#include<stdio.h>

 unsigned int count = 1;

 int main(void)
 {
     int a=0;
     a += 1;
     printf("\n [%d] \n",count++);
     main();

     return 0;
 }
```

I added this printf() just to print how many times a new stack frame is constructed. Here is the output when we run the code :

Code:

```
 ...
 ...
 ...
 ...
 ...

 [327293]

  [327294]

  [327295]

  [327296]

  [327297]
```

```
[327298]

[327299]

[327300]

[327301]

[327302]

[327303]

[327304]

[327305]
```

So we clearly see that after calling main() 327305 times crash occurred. This happened as all the stack memory is consumed and when the main() is called again, there is no more stack memory available.

Hence a stack corruption occurred

Accessing array out of bounds

Since 'C' does not check array bound access, so its a great source of stack corruption. Consider the following code :

Code:

```
#include<stdio.h>

 unsigned int count = 1;

 int main(void)
 {
     int b = 10;
     int a[3];
     a[0] = 1;
     a[1] = 2;
     a[2] = 3;

     printf("\n b = %d \n",b);
     a[3] = 12;
     printf("\n b = %d \n",b);

     return 0;
 }
```

In the above code, I have declared an array of 3 integers (a[3]). But suppose, somehow if the code tries to change the value kept at a[3] (which is a memory location not to be accessed by the array), then lets see what happens. Here is the output :

Code:

```
~/practice $ ./rcrsn
```

```
  b = 10

  b = 12
```

We see in the above output that the value of 'b' changes from 10 to 12. But the point is that we haven't changed 'b' explicitly in the code anywhere, so how did this happen ?

Lets put some debug print statements in the code,so the code becomes :

Code:

```
#include<stdio.h>

 unsigned int count = 1;

 int main(void)
 {
     int b = 10;
     int a[3];
     a[0] = 1;
     a[1] = 2;
     a[2] = 3;

     printf("\n b = %d \n",b);
     printf("\n address of b = %x, address of a[3] = %x \n",&b, &a[3]);
     a[3] = 12;
     printf("\n b = %d \n",b);

     return 0;
 }
```

We added a debug print log to see the addresses of the illegal statement a[3]=12 and 'b'. lets run the code, here is the output :

Code:

```
~/practice $ ./rcrsn

  b = 10

  address of b = a246661c, address of a[3] = a246661c

  b = 12
```

Ahh...now we see that the illegal access of memory by the statement a[3] is done on the memory address where value of 'b' is stored. Hence we see that the value of 'b' gets changed silently. Imagine the damage this kind of problem can do in a code which has thousands of line of code.

So, this is also a type of stack corruption.

Using undefined or freed pointer holding a garbage address

Consider the following code :

Code:

```
#include<stdio.h>

 unsigned int count = 1;

 int main(void)
 {
     int b = 10;
     char *ptr;

     printf("\n garbage address held by ptr = %x\n", ptr);
     return 0;
 }
```

We see that in the above code we have left the pointer ptr uninitialized. This is a wrong practice but it still happens some times even in professionally written code. Lets see the output :

Code:

```
~/practice $ ./rcrsn

  garbage address held by ptr = 303e1f80
```

So we see here that pointer holds garbage address 303e1f80.

Now, we see that pointer holds some garbage address, when we say garbage address we mean any number can be stored there. Though bleak but there are chances that this garbage could be the address of integer 'b'. Yes, this could happen and then if incidentally we access the pointer, we may not get any run time error and the value of 'b' could change silently.

Hence this is also a form of stack corruption.

An interesting question

Now since you understand stack corruption, please go through the following code and tell me whats wrong in it?

Code:

```
#include<stdio.h>

 void f();

 int main()
 {
 int i;
 i=20;
 printf("\n i = %d \n", i);
 f();
 printf("\n i = %d \n", i);
 return 0;
 }

 void f()
 {
 int j=20;
 int* a = &j;
 *(a+8)+=7;
 }
```

I am leaving this as an exercise for all you readers. In case of any doubts, leave a message here.

Conclusion

To conclude, In this article we learned about what is stack corruption and learned different ways in which it can happen. I left the explanation of one of the ways as I intend to cover it as a separate topic in a different article.

Stay tuned for more!!!!

poornaMoksha, Nov 23, 2011 SHARE #1

(You must log in or sign up to reply here.)