[Why it’s not safe to return value on function stack](https://stackoverflow.com/questions/5185773/why-its-not-safe-to-return-value-on-function-stack)

Now imagine what would happen if an ordinary function tried to return values on the stack  
.you can,t touch any part of the stack that's above the return address,so the function would have to push the values below the return address.**But when the assembly language return is executed ,the stack pointer must be pointing to the return address(or right below it depending on your machine),so right before the RETURN ,function must move the stack pointer up,thus clearing of all the local variables**.If you are trying to return values on the stack below the return address,**you become vulnerable at the moment because an interrupt could come along.The ISR would come the stack pointer down to hold its return address and its local variables and overwrite your return value**

**Explanation:**

Suppose that you have the following call stack somewhere in your application:

1. Main routine
2. Function1's local variables
3. Function2's local variables <-- STACK POINTER

In this case main calls function1, and function1 calls function2.

Now suppose that function2 calls function3, and the return value of function3 is returned on the stack:

1. Main routine
2. Function1's local variables
3. Function2's local variables
4. Function3's local variables, including the return value <-- STACK POINTER

Function3 stores the return value on the stack, and then returns. Returning means, decreasing the stack pointer again, so the stack becomes this:

1. Main routine
2. Function1's local variables
3. Function2's local variables <-- STACK POINTER

You see, function3's stack frame is not here anymore.

Well, actually I lied a bit. The stack frame is still there:

1. Main routine
2. Function1's local variables
3. Function2's local variables <-- STACK POINTER
4. Function3's local variables, including the return value

So it seems safe to still access the stack to get the return value.

But, if there is an interrupt AFTER function3 has returned, but BEFORE function2 get's the return value from the stack, we get this:

1. Main routine
2. Function1's local variables
3. Function2's local variables
4. Interrupt function's local variables <-- STACK POINTER

And now the stack frame is really overwritten, and the return value that we desperately needed has gone.

That's why returning a return value on the stack is not safe.

The problem is similar to the one shown in this simple piece of C code:

char \*buf = (char \*)malloc(100\*sizeof(char \*));

strcpy (buf, "Hello World");

free (buf);

printf ("Buffer is %s\n",buf);

Most of the times, the memory that was used for buf will still have the contents "Hello World", but it can go horribily wrong if someone is able to allocate memory after free has been called, but before printf is called. One such example is in multi-threaded applications (and we already encountered this problem internally), like shown here:

THREAD 1: THREAD 2:

--------- ---------

char \*buf = (char \*)malloc(100);

strcpy (buf, "Hello World");

free (buf);

char \*mybuf = (char \*)malloc(100);

strcpy (mybuf, "This is my string");

printf ("Buffer is %s\n",buf);

The printf is Thread 1 may now print "Hello World", or it may print "This is my string". Anything can happen.