**NAME *: LAKSHITHA RAJ VASANADU***

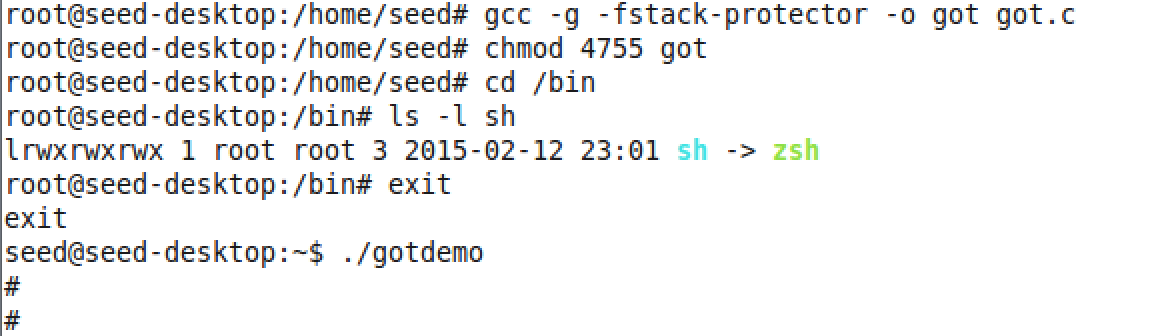
**ID *: 00001115006***

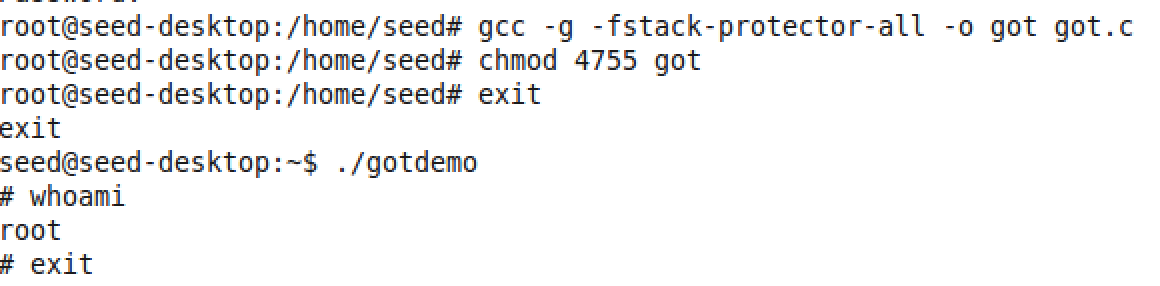
**LAB ASSIGNMENT 3 TASK 2**

**MITIGATION STRATEGIES AND EXPLORING THE STACK**

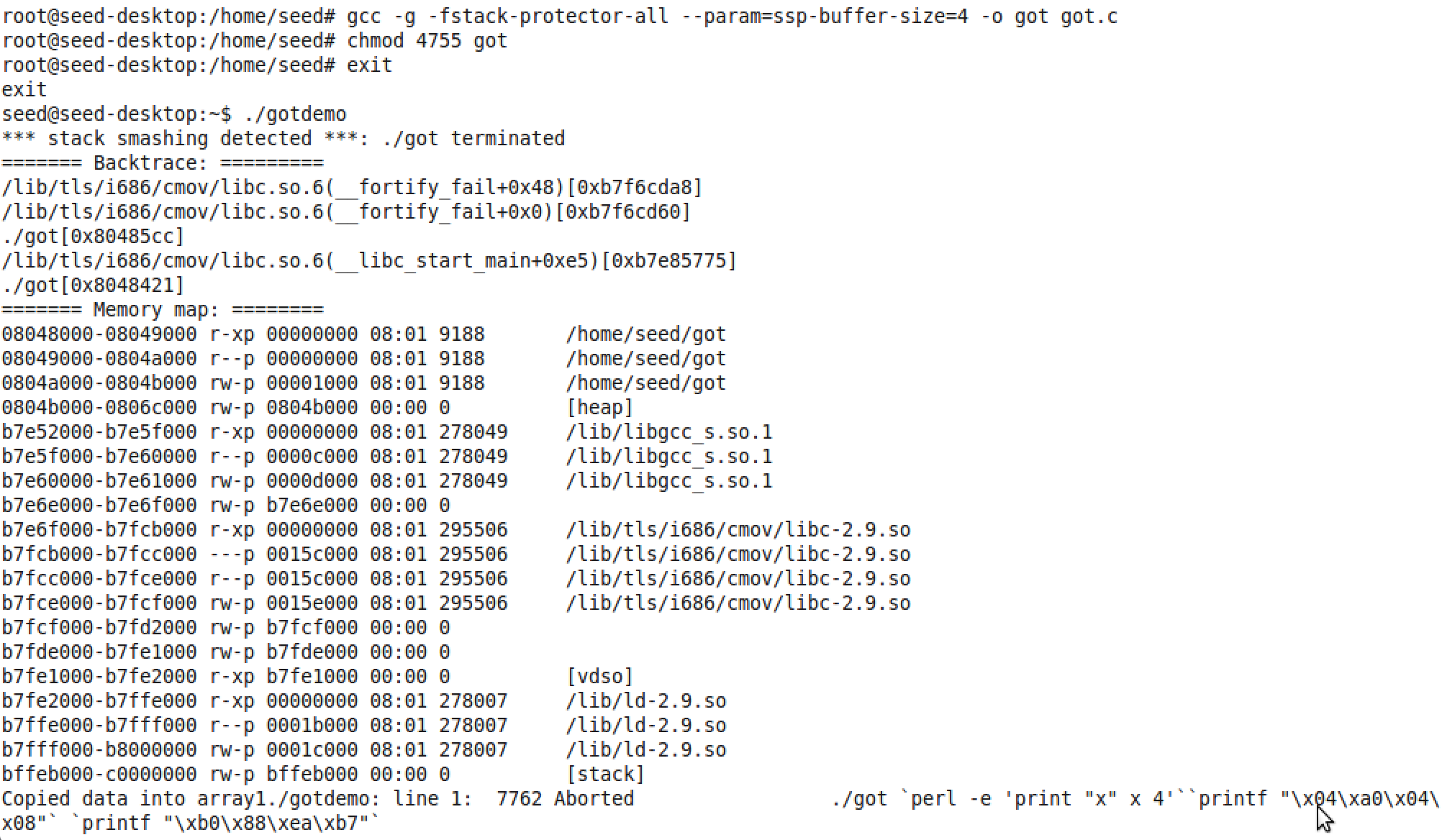
1. Both –fstack-protector and –fstack-protector-all do not detect the buffer overflow with the given program.

* **-fstack-protector** : This option emits extra code to check for buffer overflows by adding a guard variable to functions with vulnerable objects like *alloca* or buffers with size > 8 bytes. In this program, the shell is spawned with root privilege as shown below:



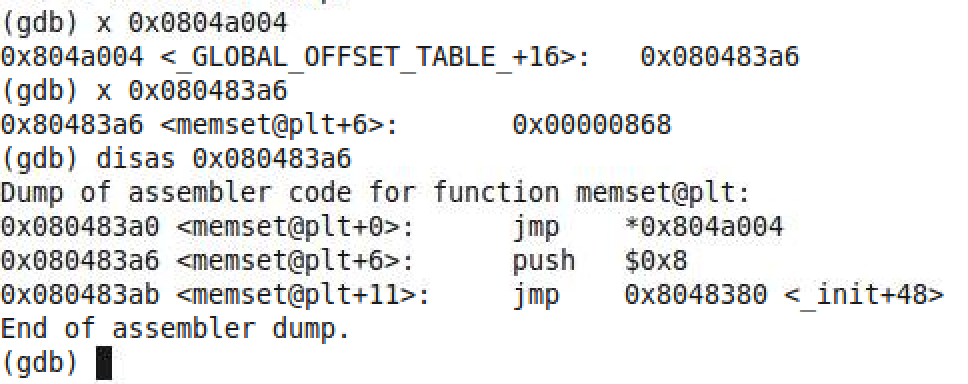
* **-fstack-protector-all** : This does the same as above but checks for stack smashing in every function. In this program, the shell is spawned with root privilege as shown below:
* 

Both of these work only for buffer > 8 bytes. In the given code, buffer size is 4 (**char array[4]**) and hence it does not work. Instead, the option **--param=ssp-buffer-size=4** can be used with gcc to explicitly specify the buffer size. This helps to protect the code from buffer overflows by aborting. It is shown here :

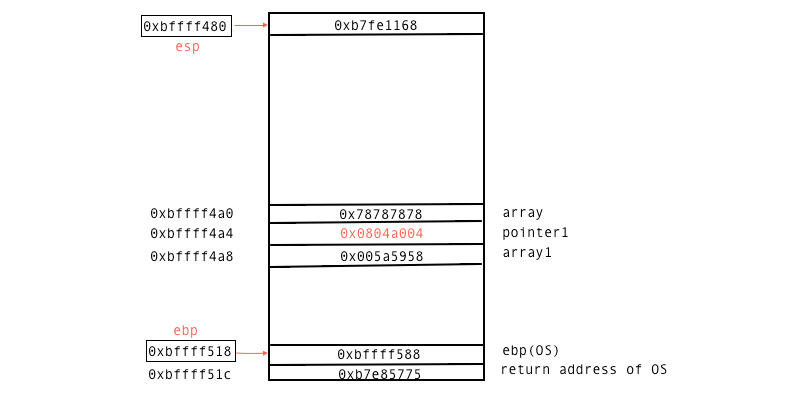


1. **After first strcpy()**

In the given program, with the **first strcpy(),** buffer is overflowed to make the **pointer1** point at the **Global Offset address** of **memset()** function. The string array contains the value “xxxx” and pointer1 is overflowed with 0x0804a004. The first memset() works in a normal manner. We observe below that pointer1 which was initially pointing to array (0xbffff4a0) is made to point to 0x0804a004 which is the global offset address of memset(). 0x0804a004 is part of our input argv[1].



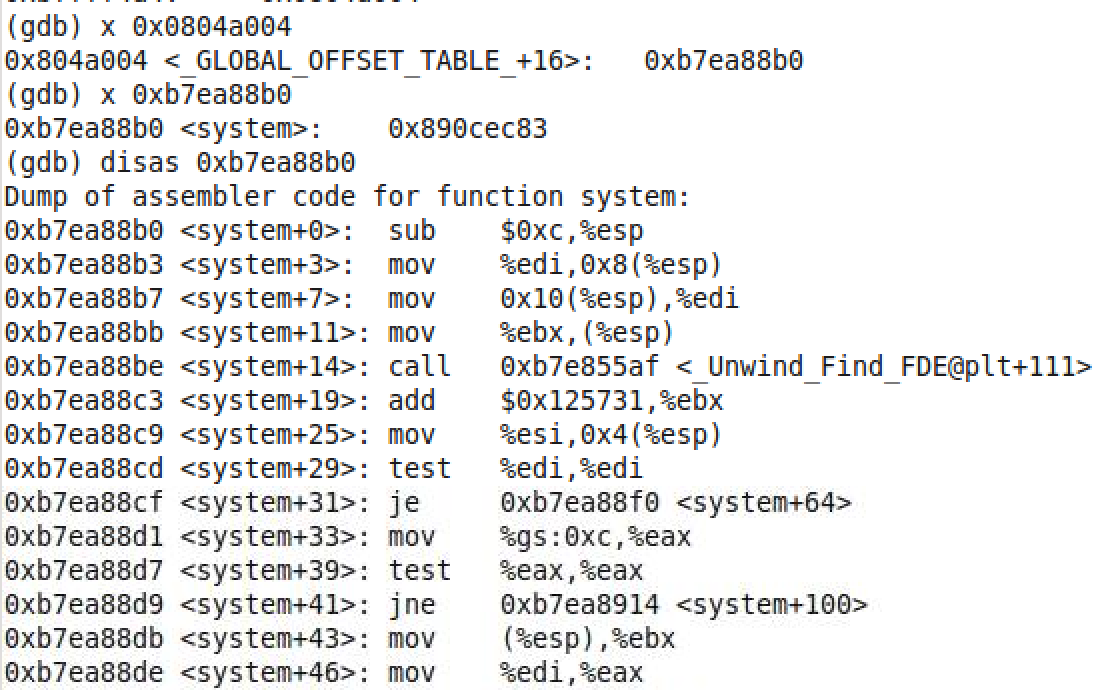
The stack at this point is shown as:



**After second strcpy()**

With the second strcpy(), the address at the GOT entry is changed to what we specified using argv[2]. Now, memset() function is patched to **system()** function. The address 0xb7ea88b0 is written here. Further calls to memset() will actually call system() with the specified parameters.

memset(array1, ‘1’, 10) will be actually called as system() with array1 as parameter. Array1 contains Array which system() tries to execute. If finds the executable Array containing the shell code to spawn the root shell.



1. The given program can be re-written as follows to protect from buffer overflows which in turn prevents arbitrary memory write. This code uses **Input Validation** as a mitigation strategy to **prevent** buffer overflow.

Here, we check whether the length of the command line inputs is less than the buffer size (i.e. array) that is statically allocated. If the condition fails, the program in aborted.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main(int argc, char \*\*argv) {

char \*pointer1 = NULL;

char array1[100] = “XYZ”;

char array[4] = “ABC”;

pointer1 = array;

if(strlen(argv[1]) >= sizeof(array)) {

printf(“Aborting the program due to long input!!”);

abort();

}

strcpy(pointer1, argv[1] );

memset(array1, '%', 10);

printf("Copied data into array1");

if(strlen(argv[2]) >= sizeof(array)) {

printf(“Aborting the program due to long input!!”);

abort();

}

strcpy(pointer1, argv[2] );

memcpy(array1, "Array ", 10);

memset(array1, '1', 10);

return 0;

}