Varada Gurjar W1113815

1. (a) Compile and run the program using the gcc options –fstack-protector and –fstackprotector-all. These options are used to detect buffer overflows when the program is being run. Explain why these options may not work with the program bufOverflow.c and how they can be enabled. (Hint: explore the use of the GCC option --param=ssp-buffer-size=)

Answer 1)

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.

While compiling bufOverflow.c the command to use the flags are

$ gcc -fstack-protector -g -o bof bufOverflow.c

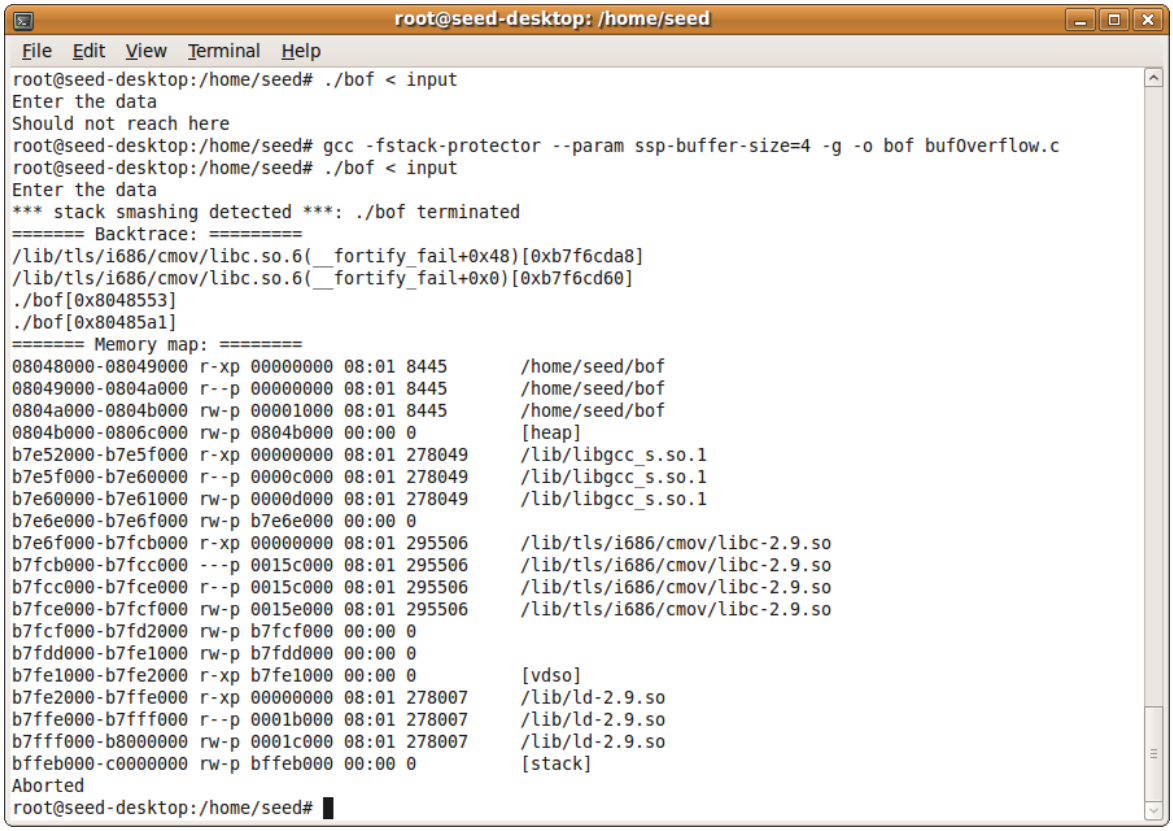
or **$ gcc -fstack-protector-all -g -o bof bufOverflow.c**

For the method bufOverflow.c the char buf[4] is 4 bytes.
Hence the use of the above flags in compiling does not prove effective.

root@seed-desktop:/home/seed# **gcc -fstack-protector -g -o bof bufOverflow.c**
root@seed-desktop:/home/seed# **./bof < input**
Enter the data
Should not reach here
root@seed-desktop:/home/seed# **gcc -fstack-protector-all -g -o bof bufOverflow.c**
root@seed-desktop:/home/seed# **./bof < input**
Enter the data
Should not reach here
root@seed-desktop:/home/seed# **gcc -fstack-protector --param ssp-buffer-size=4 -g -o bof bufOverflow.c**
root@seed-desktop:/home/seed# **./bof < input**
Enter the data
***** stack smashing detected ***: ./bof terminated**
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x48)[0xb7f6cda8]
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x0)[0xb7f6cd60]
./bof[0x8048553]
./bof[0x80485a1]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 8445 /home/seed/bof
08049000-0804a000 r--p 00000000 08:01 8445 /home/seed/bof
0804a000-0804b000 rw-p 00001000 08:01 8445 /home/seed/bof
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
b7fdd000-b7fe1000 rw-p b7fdd000 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]
Aborted

For buffer overflow protection with -fstack-protector the ssp-buffer-size parameter can be set to the minimum size of buffer that requires protection. In this case the value will be 4.



(b) Draw a diagram of the stack showing how it looks

a) immediately before the strncpy() function is executed. You must show where the argument str, saved base pointer of main, return address of main() and local variable buf are placed on the stack.

b) immediately after the strncpy() function completes. Show the contents of the buffer and how the input to the program is stored on the stack.

Answer part a)

The malinput.c file dumps text data, to be used with malicious intent in the bufOverflow program, in the input file.

seed@seed-desktop:~$ su
Password:
root@seed-desktop:/home/seed# **sysctl -w kernel.randomize_va_space=0**
kernel.randomize_va_space = 0 **//turn off address randomization**
root@seed-desktop:/home/seed# gcc -o malin malinput.c
root@seed-desktop:/home/seed# ./malin **// file input is generated**
root@seed-desktop:/home/seed# gcc -g -o bof bufOverflow.c

```
root@seed-desktop:/home/seed# ./bof < input          //bof takes the malicious dump in input file
Enter the data                                       // as an input
Should not reach here
```

The data in 'str' is the same as the text received from the input file . The length of the string 'str' is greater than the size of the buffer 'buf'. The data in input copied into buf causes a buffer overflow.

The return address of copyData in main, stored on the stack gets replaced by the address of the method message() [This is demonstrated with address values below] . As a result after the copy instead of returning back to main the copyData() method invokes the message() function.

The message() is indirectly called from the buffer overflow in copyData(). If there is sensitive information in message() this specific strncpy call can be very dangerous.

```
root@seed-desktop:/home/seed# gdb bof
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 copyData
7       puts("Should not reach here");
8       exit(0);
9       }
10
11      void copyData(char *str)                    // I initialized buf to BBB to observe change of value
12      {                                           // of x buf from 0x00424242 to 0x41414141 right after
13      char buf[4] = "BBB";                        //strncpy()
14      strncpy(buf, str, strlen(str));
15      }
16
(gdb) b 13                                          //  breakpoint set at first line of method copyData()
Breakpoint 1 at 0x8048508: file bufOverflow.c, line 13.
(gdb) run < input
Starting program: /home/seed/bof < input
Enter the data
Breakpoint 1, copyData (str=0xbffff514 "AAAAAAAA\204\004\b")
   at bufOverflow.c:13
13      char buf[4] = "BBB";
(gdb) X/i $pc
0x8048508 <copyData+6>:       mov    0x8048676,%eax
(gdb) si
0x0804850d     13       char buf[4] = "BBB";
(gdb) si
14      strncpy(buf, str, strlen(str));
(gdb) si
0x08048513     14       strncpy(buf, str, strlen(str));
(gdb) si
0x08048516     14       strncpy(buf, str, strlen(str));
```

(gdb) **X/i $pc**
0x8048516 <copyData+20>: call 0x80483f0 <strlen@plt>
(gdb) **nexti** **// nexti called to avoid stepping into strlen**
0x0804851b 14 strncpy(buf, str, strlen(str));
(gdb) **X/i $pc**
0x804851b <copyData+25>: mov %eax,0x8(%esp)
(gdb) si
0x0804851f 14 strncpy(buf, str, strlen(str));
(gdb) si
0x08048522 14 strncpy(buf, str, strlen(str));
(gdb) si
0x08048526 14 strncpy(buf, str, strlen(str)); **// calls before strncpy() will execute**
(gdb) si
0x08048529 14 strncpy(buf, str, strlen(str));
(gdb) si
0x0804852c 14 strncpy(buf, str, strlen(str));
(gdb) **X/i $pc** **// The point of debug right before the call to strncpy()**
0x804852c <copyData+42>: call 0x80483c0 <strncpy@plt>
(gdb) disas copyData

// Observing the assembly code of copyData function

Dump of assembler code for function copyData:
0x08048502 <copyData+0>: push %ebp
0x08048503 <copyData+1>: mov %esp,%ebp
0x08048505 <copyData+3>: sub $0x28,%esp // prolog ends here
0x08048508 <copyData+6>: mov 0x8048676,%eax
0x0804850d <copyData+11>: mov %eax,-0x4(%ebp)
0x08048510 <copyData+14>: mov 0x8(%ebp),%eax
0x08048513 <copyData+17>: mov %eax,(%esp)
0x08048516 <copyData+20>: call 0x80483f0 <strlen@plt>
0x0804851b <copyData+25>: mov %eax,0x8(%esp)
0x0804851f <copyData+29>: mov 0x8(%ebp),%eax
0x08048522 <copyData+32>: mov %eax,0x4(%esp)
0x08048526 <copyData+36>: lea -0x4(%ebp),%eax
0x08048529 <copyData+39>: mov %eax,(%esp)
0x0804852c <copyData+42>: call 0x80483c0 <strncpy@plt> **// Currently $pc is pointing here**
0x08048531 <copyData+47>: leave **// i.e. just before strncpy executes**
0x08048532 <copyData+48>: ret
End of assembler dump. // leave ret are the epilog

Stack just before the strncpy() executes :

- **buf and $ebp-4 value**

(gdb) x/s **buf**
0xbffff4f4: "BBB"
(gdb) x/x **$ebp-4**
0xbffff4f4: 0x00424242
(gdb) x/s **$ebp-4** **// $ebp-4 stores initialized value of 'buf'.**
0xbffff4f4: "BBB"

- **$ebp**

(gdb) x/x **$ebp**
0xbffff4f8: 0xbffff528 **// address of new ebp is** 0xbffff4f8 **storing old ebp address** 0xbffff528

- **$ebp+4**

(gdb) x/x **$ebp+4**
0xbffff4fc: 0x0804857f **// return address of copyData in main stored at $ebp+4**

(gdb) x/s str
0xbffff514: "AAAAAAAA\204\004\b"
(gdb) x/x $esp **// the input file data is in 'str'**
0xbffff4d0: 0xbffff4f4
(gdb) x/x $esp+4
0xbffff4d4: 0xbffff514

// Observing the assembly code of main

(gdb) **disas main**
Dump of assembler code for function main:
0x08048533 <main+0>: lea 0x4(%esp),%ecx
0x08048537 <main+4>: and $0xfffffff0,%esp
0x0804853a <main+7>: pushl -0x4(%ecx)
0x0804853d <main+10>: push %ebp
0x0804853e <main+11>: mov %esp,%ebp
0x08048540 <main+13>: push %ecx
0x08048541 <main+14>: sub $0x24,%esp
0x08048544 <main+17>: mov 0x4(%ecx),%eax
0x08048547 <main+20>: mov %eax,-0x18(%ebp)
0x0804854a <main+23>: mov %gs:0x14,%eax
0x08048550 <main+29>: mov %eax,-0x8(%ebp)
0x08048553 <main+32>: xor %eax,%eax
0x08048555 <main+34>: movl $0x804867a,(%esp)
0x0804855c <main+41>: call 0x8048410 <puts@plt>
0x08048561 <main+46>: lea -0x14(%ebp),%eax
0x08048564 <main+49>: mov %eax,0x4(%esp)
0x08048568 <main+53>: movl $0x8048689,(%esp)
0x0804856f <main+60>: call 0x80483e0 <scanf@plt>
0x08048574 <main+65>: lea -0x14(%ebp),%eax
0x08048577 <main+68>: mov %eax,(%esp)
0x0804857a <main+71>: **call 0x8048502 <copyData>**
0x0804857f <main+76>: mov $0x0,%eax // return address of copyData in main
0x08048584 <main+81>: mov -0x8(%ebp),%edx
0x08048587 <main+84>: xor %gs:0x14,%edx
0x0804858e <main+91>: je 0x8048595 <main+98>
0x08048590 <main+93>: call 0x8048400 <__stack_chk_fail@plt>
0x08048595 <main+98>: add $0x24,%esp
0x08048598 <main+101>: pop %ecx
0x08048599 <main+102>: pop %ebp
0x0804859a <main+103>: lea -0x4(%ecx),%esp
0x0804859d <main+106>: ret
End of assembler dump.

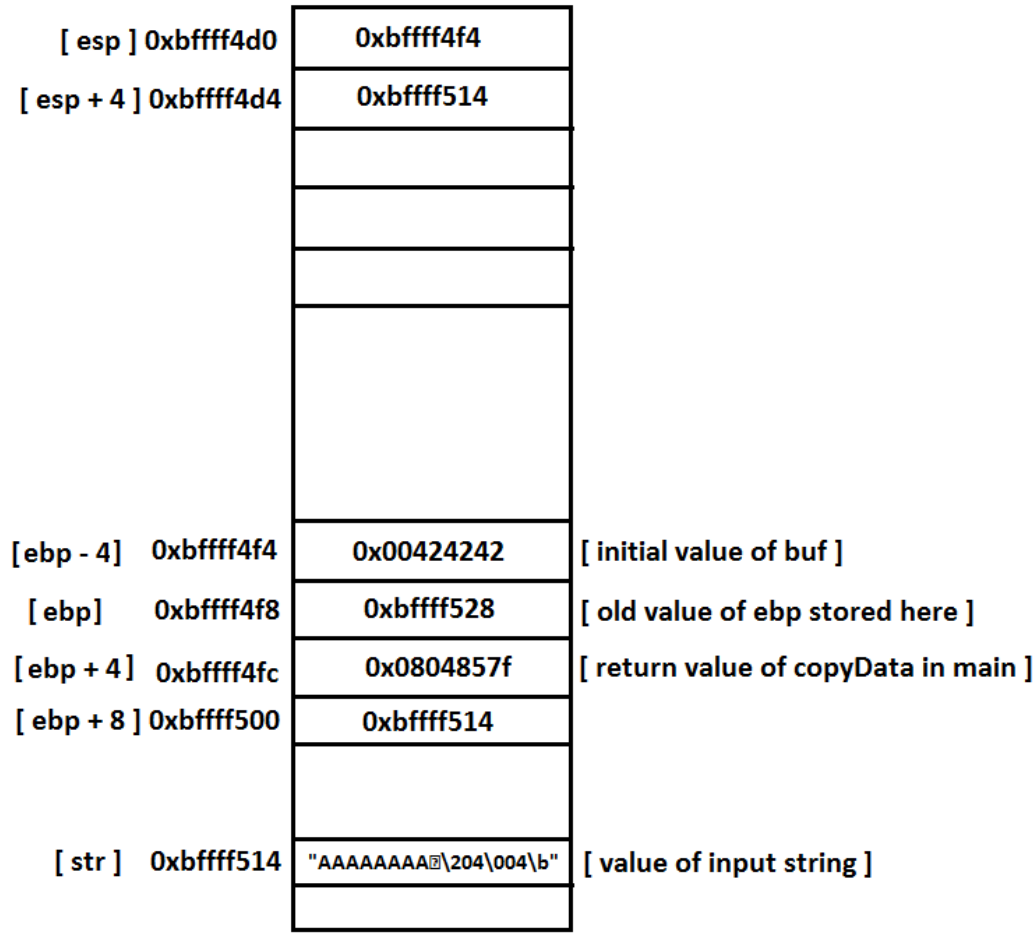
❖ **(gdb) x/x $ebp+4** **// $ebp+4 has same value as the address highlighted in main**

0xbffff4fc: **0x0804857f** **// return address of copyData() function is stored at $ebp+4**

(gdb) x/x $ebp+8

0xbffff500: 0xbffff514

Stack before strcpy()



(gdb) **n**

15 }

(gdb) **x/i $pc** **// strncpy() function executed**

0x8048531 <copyData+47>: leave

Stack just after the strncpy() executes :

❖ (gdb) x/s **buf**

0xbffff4f4: **"AAAAAAAA◆\204\004\b\024◆◆◆\024◆◆◆\030◆◆◆\234\203\004\b◆◆◆◆AAAAAAAA◆\204\004\b"**

(gdb) **x/x $ebp-4**

0xbffff4f4: 0x41414141 **// $ebp-4 stores value of 'buf' after overflow.**

(gdb) **disas message**

Dump of assembler code for function message:

0x080484e4 <message+0>: push %ebp // prolog of the message function

0x080484e5 <message+1>: mov %esp,%ebp

0x080484e7 <message+3>: sub $0x8,%esp

0x080484ea <message+6>: movl $0x8048660,(%esp) **// address of the message function**

0x080484f1 <message+13>: call 0x8048410 <puts@plt>

0x080484f6 <message+18>: movl $0x0,(%esp)

0x080484fd <message+25>: call 0x8048420 <exit@plt>

End of assembler dump.

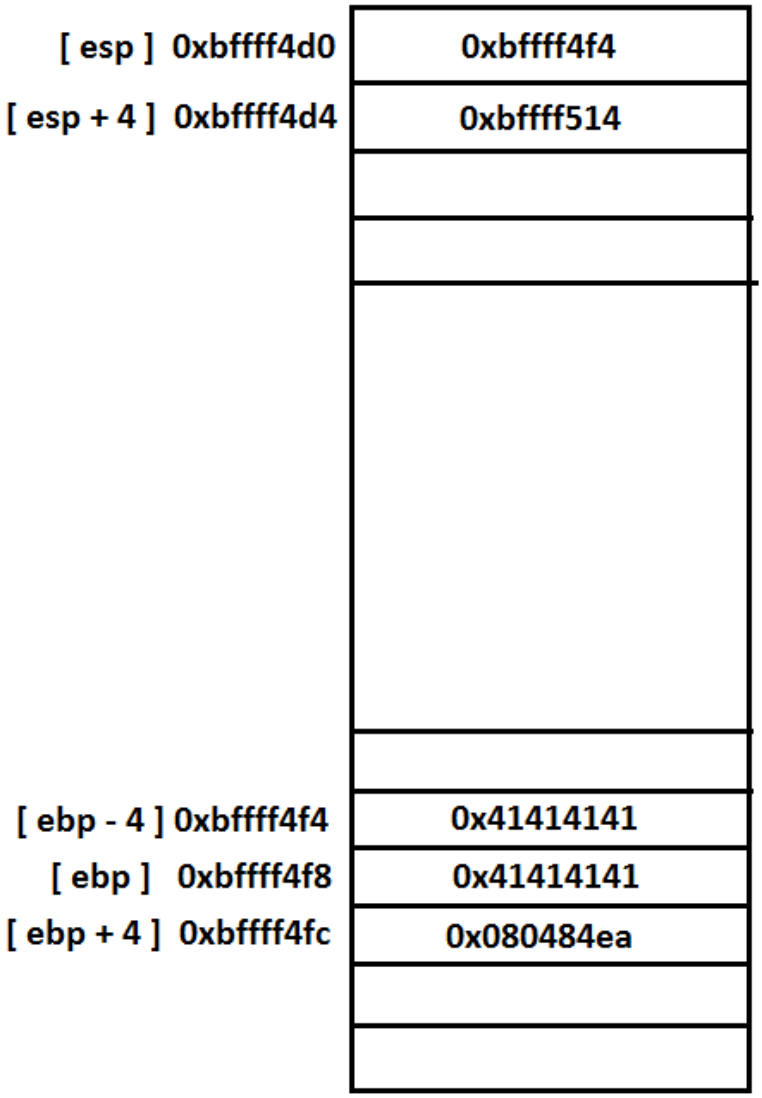
(gdb) **x/x $ebp+4**
0xbffff4fc: **0x080484ea** **//after strncpy call, $ebp+4 now stores value of message() address**

(gdb) **x/x $ebp**
0xbffff4f8: **0x41414141** **// $ebp and $ebp-4 contain the 'str' contents.**

(gdb) x/x $ebp-8
0xbffff4f0: 0x00000000

(gdb) x/x $esp
0xbffff4d0: 0xbffff4f4
(gdb) x/x $esp+4
0xbffff4d4: 0xbffff514

Stack just after strcpy function executes



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

Answer)

- **Strlcpy()** can be used within copyData as a mitigation strategy, within copyData :

This function is similar to strncpy(), but it copies at most *size-1* bytes to *dest*, always adds a terminating null byte.
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.

The code of bufOverflow can be modified as below :

```
# include <stdio.h>
# include <string.h>
# include <stdlib.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 siz == 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 */
        }
```

```
void message()
{
puts("Should not reach here");
exit(0);
}

void copyData(char *str)
{
char buf[4] = "BBB";
size_t length = strlcpy(buf, str, 4);
if (length >= 4)
        {  printf("\nTruncation occurred: Possible data loss");
           printf("\n buf : %s \n", buf);  }
}

           int main(int argc , char * argv[])
{
char data[12];
printf("Enter the data\n");
scanf("%s", data);
copyData(data);
return 0;
}
```

root@seed-desktop:/home/seed# **gcc -g -o bof bufOverflow.c**
root@seed-desktop:/home/seed# **./bof < input**
Enter the data

Truncation occurred: Possible data loss
buf : AAA

This mitigation has checked if the copy done into buf in copyData is valid and preventing buffer overflow.

A message if the length of source is greater than destination is given to alert user that the destination will have loss of data. Although buf has a size of 4 just three A's get copied as the last byte contains the null terminator.

This is a safe method to use to avoid buffer overflow because size of destination is passed as an input. If handled correctly and right size of the destination buffer is known this is a safe method and buffer overflow mitigation strategy.