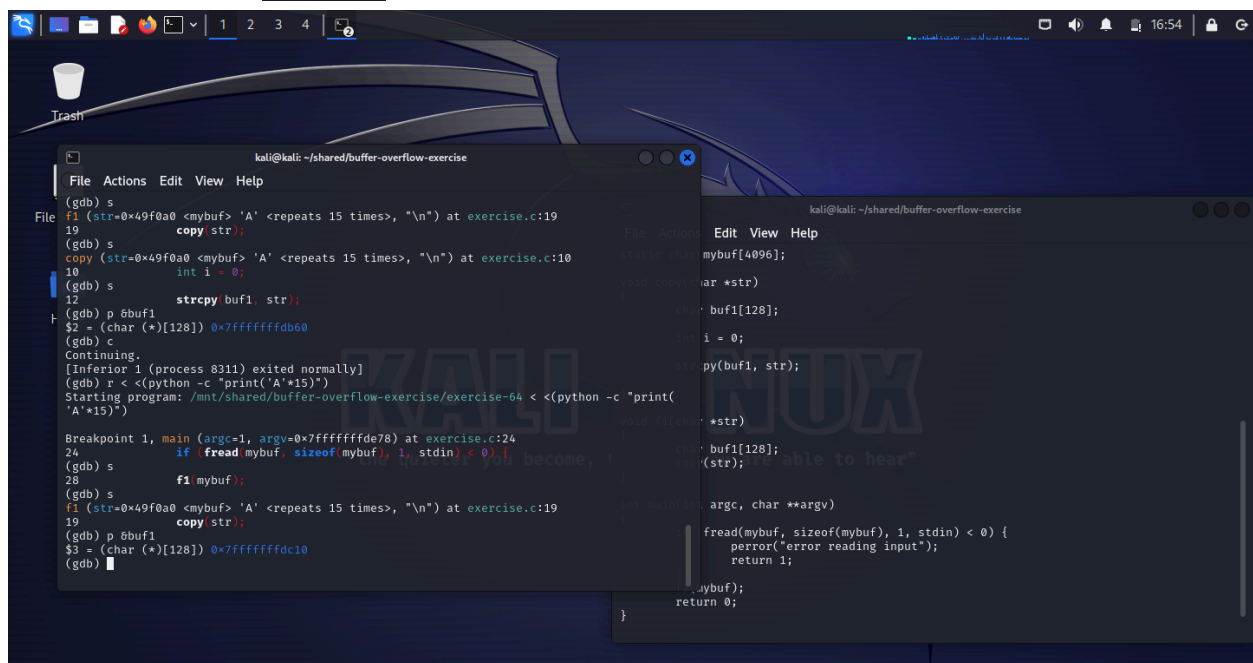


Maximiliano Brizzio  
CS-576  
Lab 6 - Buffer Overflow

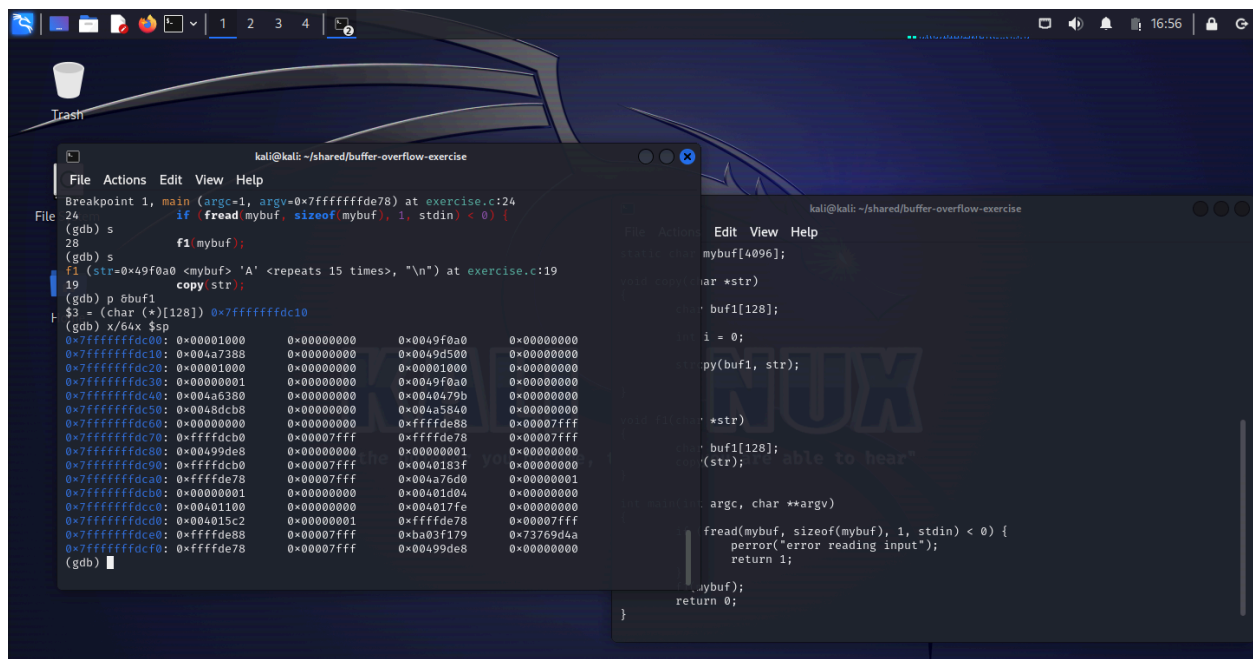
Task 1:

To find the buffer start address, there are a few things we need to set up first. I started by running `gdb exercise-64`. Once in gdb, I set a breakpoint at `main` and run the program using the command `r <<(python -c "print('A'*15)")`. Then I take steps of the program until I enter the `f1` function. Once inside of the `f1` function, I print out the address of the `buf1` variable using the command `p &buf1`. The base address is `0x7ffffffdc10`.



```
kali@kali: ~/shared/buffer-overflow-exercise
(gdb) s
Breakpoint 1, main (argc=1, argv=0x7ffffffde78) at exercise.c:24
24 if (fread(mybuf, sizeof(mybuf), 1, stdin) < 0) { become, I
(gdb) s
28 f1(mybuf);
(gdb) s
f1(str=0x49f0a0 <mybuf> 'A' <repeats 15 times>, "\n") at exercise.c:19
19 copy(str);
(gdb) p &buf1
$3 = (char (*)[128]) 0x7ffffffdc10
(gdb) c
Continuing.
[Inferior 1 (process 8311) exited normally]
(gdb) r <<(python -c "print('A'*15)")
Starting program: /mnt/shared/buffer-overflow-exercise/exercise-64 <<(python -c "print('A'*15)")
Breakpoint 1, main (argc=1, argv=0x7ffffffde78) at exercise.c:24
24 if (fread(mybuf, sizeof(mybuf), 1, stdin) < 0) { become, I
(gdb) s
28 f1(mybuf);
(gdb) s
f1(str=0x49f0a0 <mybuf> 'A' <repeats 15 times>, "\n") at exercise.c:19
19 copy(str);
(gdb) p &buf1
$3 = (char (*)[128]) 0x7ffffffdc10
(gdb) |
```

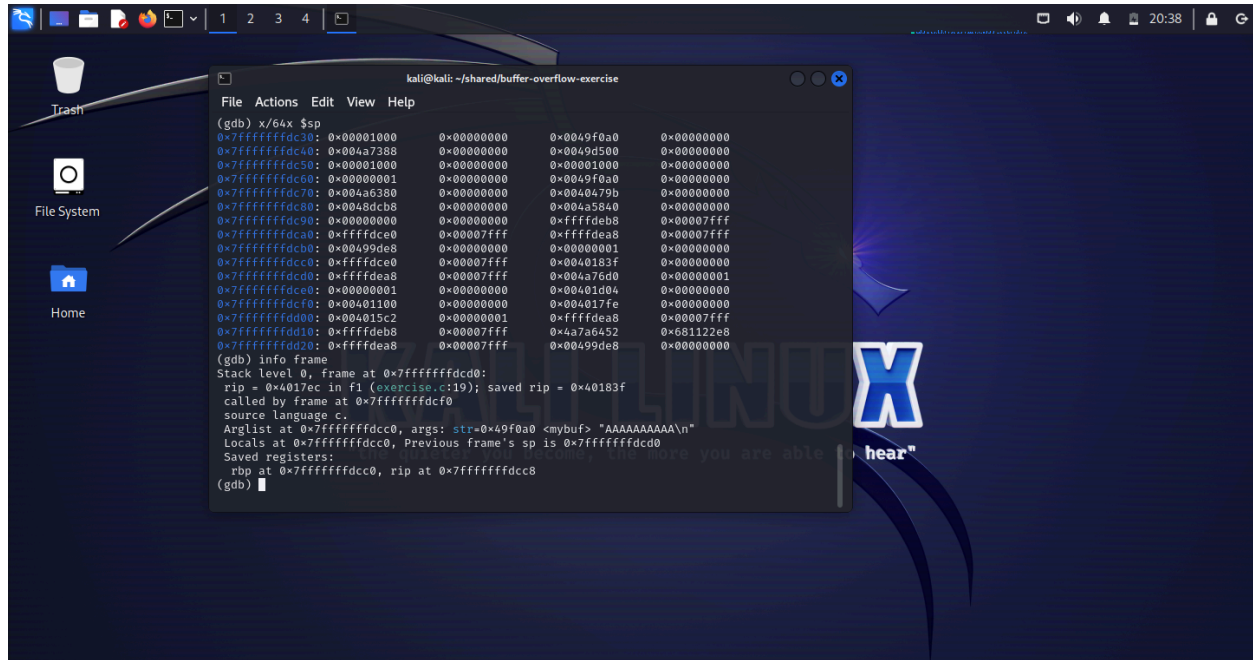
Next, we can print out the stack and check where `buf1` is stored:



```
kali@kali: ~/shared/buffer-overflow-exercise
Breakpoint 1, main (argc=1, argv=0x7ffffffde78) at exercise.c:24
24 if (fread(mybuf, sizeof(mybuf), 1, stdin) < 0) { become, I
(gdb) s
28 f1(mybuf);
(gdb) s
f1(str=0x49f0a0 <mybuf> 'A' <repeats 15 times>, "\n") at exercise.c:19
19 copy(str);
(gdb) p &buf1
$3 = (char (*)[128]) 0x7ffffffdc10
(gdb) x/64x $sp
0x7ffffffdc00: 0x00001000 0x00000000 0x0049f0a0 0x00000000
0x7ffffffdc10: 0x00a73888 0x00000000 0x0049d500 0x00000000
0x7ffffffdc20: 0x00001000 0x00000000 0x00001000 0x00000000
0x7ffffffdc30: 0x00000001 0x00000000 0x0049f0a0 0x00000000
0x7ffffffdc40: 0x004a6380 0x00000000 0x0040479b 0x00000000
0x7ffffffdc50: 0x0048dc88 0x00000000 0x004a5840 0x00000000
0x7ffffffdc60: 0x00000000 0x00000000 0xffffde88 0x00007fff
0x7ffffffdc70: 0xffffde80 0x00007fff 0xffffde78 0x00007fff
0x7ffffffdc80: 0x00a99de8 0x00000000 0x00000001 0x00000000
0x7ffffffdc90: 0xffffde80 0x00007fff 0x0040183f 0x00000000
0x7ffffffdca0: 0xffffde78 0x00007fff 0x004a76d0 0x00000001
0x7ffffffdcb0: 0x00000001 0x00000000 0x00401d04 0x00000000
0x7ffffffdcc0: 0x00401100 0x00000000 0x004017fe 0x00000000
0x7ffffffdcd0: 0x004015c2 0x00000001 0xffffde78 0x00007fff
0x7ffffffdce0: 0xffffde88 0x00007fff 0xba03f179 0x737b9d4a
0x7ffffffdcf0: 0xffffde78 0x00007fff 0x00499de8 0x00000000
(gdb) |
```

## Task 2:

Because we know the size of `buf1` (128 bytes) and its start address in the stack, we can calculate the bounds of it. But if we want to use `buf1` to overwrite the return address of `f1`, we have to find the return address first. To do this, I run the command `info frame` which under the 'saved rip' value stores the return address of `f1`. The address is `0x40183f`.

A screenshot of a Kali Linux desktop environment. The desktop background is dark blue with a large white 'X' logo and the text 'hear"'. On the left side, there are icons for 'Trash', 'File System', and 'Home'. A terminal window is open in the center, displaying GDB output. The window title is 'kali@kali: ~/shared/buffer-overflow-exercise'. The output shows a memory dump of the stack, followed by the command `(gdb) info frame` and its output. The output indicates that the return address (rip) is `0x40183f` in the `f1` frame.

```
kali@kali: ~/shared/buffer-overflow-exercise
(gdb) x/64x $sp
0x7fffffffdc30: 0x00001000  0x00000000  0x0049f0a0  0x00000000
0x7fffffffdc40: 0x004a7388  0x00000000  0x0049d500  0x00000000
0x7fffffffdc50: 0x00001000  0x00000000  0x00001000  0x00000000
0x7fffffffdc60: 0x00000001  0x00000000  0x0049f0a0  0x00000000
0x7fffffffdc70: 0x004a6380  0x00000000  0x0040479b  0x00000000
0x7fffffffdc80: 0x0048dcb8  0x00000000  0x004a5840  0x00000000
0x7fffffffdc90: 0x00000000  0x00000000  0xffffdeb8  0x00007fff
0x7fffffffcca0: 0xffffdce0  0x00007fff  0xffffdea8  0x00007fff
0x7fffffffccb0: 0x00499de8  0x00000000  0x00000001  0x00000000
0x7fffffffccc0: 0xffffdce0  0x00007fff  0x0040183f  0x00000000
0x7fffffffcd0: 0xffffdea8  0x00007fff  0x004a76d0  0x00000001
0x7fffffffcd0: 0xffffdce0  0x00000000  0x00401d04  0x00000000
0x7fffffffcd0: 0x00401100  0x00000000  0x004017fe  0x00000000
0x7fffffffdd00: 0x004015c2  0x00000001  0xffffdea8  0x00007fff
0x7fffffffdd10: 0xffffdeb8  0x00007fff  0x4a7a6452  0x681122e8
0x7fffffffdd20: 0xffffdea8  0x00007fff  0x00499de8  0x00000000
(gdb) info frame
Stack level 0, frame at 0x7fffffffdd0:
 rip = 0x4017ec in f1 (exercise.c:19); saved rip = 0x40183f
 called by frame at 0x7fffffffdd0
 source language c.
 Arglist at 0x7fffffffdd0, args: str=0x49f0a0 <mybuf> "AAAAAAAAA\n"
 Locals at 0x7fffffffdd0, Previous frame's sp is 0x7fffffffdd0
 Saved registers:
  rbp at 0x7fffffffdd0, rip at 0x7fffffffdd0
(gdb) |
```

Now using the return address, we can cross reference the stack to look for the same hexadecimal string (`0x40183f`). It appears that from stack address `0x7fffffffcca-7fffffffccc` (inclusive) is where the return address of `f1` is kept.

We know `buf1` starts at `0x7fffffffdc10`, so to overwrite the return address of `f1`, we need to make `buf1` at least 188 bytes long ( $0x7fffffffdc10 + BC = 0x7fffffffccc$  and  $BC = 188$ ). Because the size of `buf1` is designed to be 128 bytes, we need to overwrite by  $188 - 128 = 60$  bytes.

