



Below is the assembly code (left) and source code (right) for the **greet** function up to the call to **gets()**, which requests input from the user, writing it to the buffer.

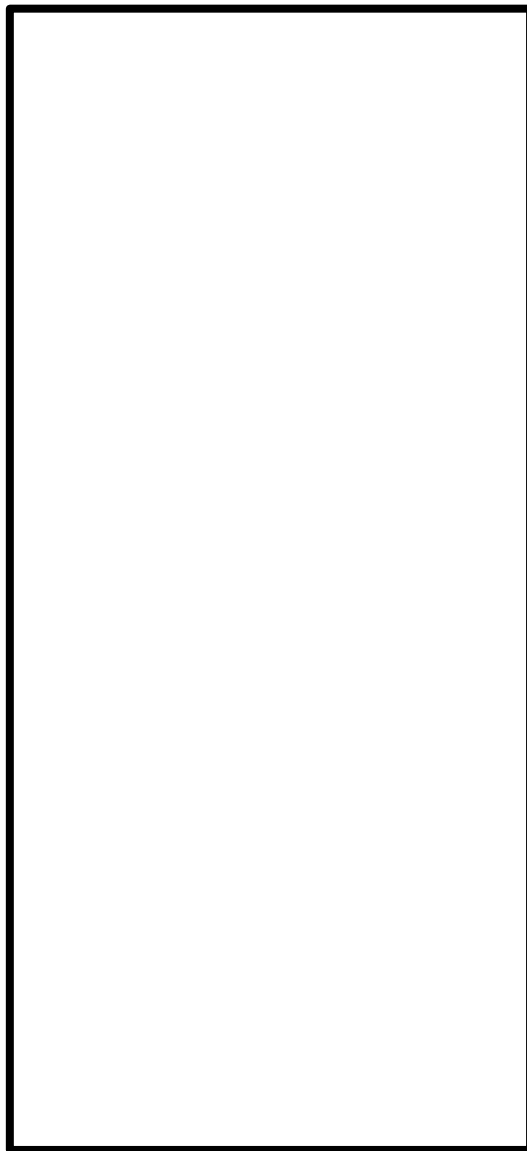
```
sub    $0x28,%rsp
movl   $0x2,0x1c(%rsp)
mov    $0x402008,%edi
mov    $0x0,%eax
callq  0x401050 <printf@plt>
mov    %rsp,%rax
mov    %rax,%rdi
callq  0x401060 <gets@plt>
```

```
# define BUF_LEN 16
int greet(){
    int num = 2;
    char buf[BUF_LEN];

    printf("What is your name?");
    if (gets(buf)){ ...
    // eventually, return num;
```

🌟 **Exercise: Draw a stack diagram** (visual layout of stack frame) for **greet**. Include how big the stack frame is, boxes for where different variables / values are laid out in memory, and how many bytes each takes up. You'll do something similar in Assign5!

higher addresses



_____ bytes big

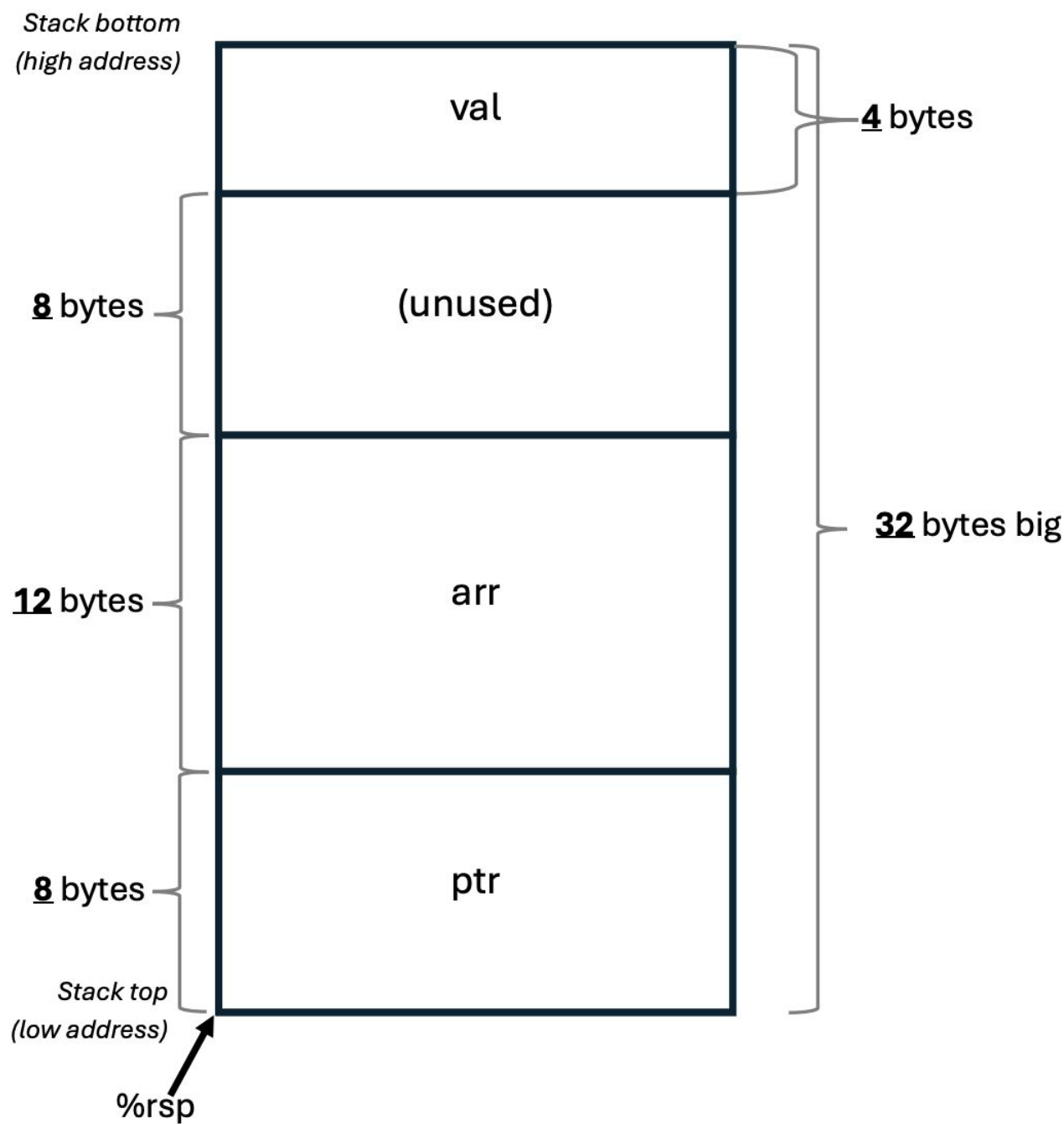
lower addresses

rsp →

(*rsp* after sub instruction to descend 0x28 bytes)



For reference, an example stack diagram! (**made-up, not for our greet function :D**)





Solution to greet() stack exercise, and drawing out the key variables!

```

sub    $0x28,%rsp
movl   $0x2,0x1c(%rsp)
mov     $0x402008,%edi
mov     $0x0,%eax
callq  0x401050 <printf@plt>
mov     %rsp,%rax
mov     %rax,%rdi
callq  0x401060 <gets@plt>

```

```

# define BUF_LEN 16
int greet(){
    int num = 2;
    char buf[BUF_LEN];

    printf("What is your name?");
    if (gets(buf)){ ...
        // eventually, return num;
    }
}

```

1

Goal: Inside `./smash` is a function `greet()` that returns 2 all the time, or does it? Here, we'll design an input that hacks it to return 107 instead! :)

```

bbyan@myth64:~/CS107_TA/assignments/lab6$ ./smash
You've got 5 tries to blow this popsicle stand. Go!
What is your name? Jerry Cain
Hello, Jerry Cain! Buffer has space for 16 chars, your name is length 10
greet() returned 2
What is your name? Ben yan
Hello, Ben yan! Buffer has space for 16 chars, your name is length 7.
greet() returned 2
What is your name? Water. Earth. Fire. Air. Long ago, the four nations 1
lived together in harmony. Then, everything changed when the Fire Nation
attacked.
Hello, Water. Earth. Fire. Air. Long ago, the four nations lived togethe
r in harmony. Then, everything changed when the Fire Nation attacked. I b
uffer has space for 16 chars, your name is length 134.
You have overrun the buffer by 119 chars.
Segmentation fault
bbyan@myth64:~/CS107_TA/assignments/lab6$

```



2

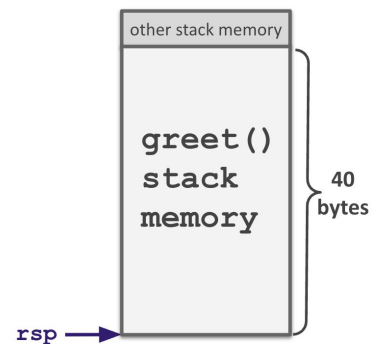
Q1: How much stack space is initially reserved for `greet`?

A: 40 bytes. Stack pointer `$rsp` descends by 0x28 (40 decimal) bytes to make space.

```

sub    $0x28,%rsp
movl   $0x2,0x1c(%rsp)
mov     $0x402008,%edi
mov     $0x0,%eax
callq  0x401050 <printf@plt>
mov     %rsp,%rax
mov     %rax,%rdi
callq  0x401060 <gets@plt>

```



3

Q2: Where is `num` (int variable, equal to 2) stored, relative to stack pointer `$rsp`?

A: 28 bytes up from `$rsp` (bedrock of `greet`'s stack), or 8 bytes from the top.

```

sub    $0x28,%rsp
movl   $0x2,0x1c(%rsp)

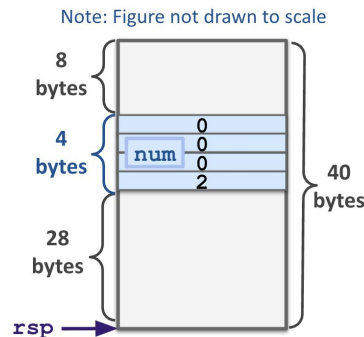
```

This moves a 4-byte integer (value 2) into address `$rsp + 28` in decimal.

```

mov     %rsp,%rax
mov     %rax,%rdi
callq  0x401060 <gets@plt>

```



4

Q3: Where is `buf` stored, relative to `$rsp`?

A: Directly at `$rsp`! We have `$rsp` passed into `gets(buf)` as its 1st parameter (`$rdi`), so that's where the buffer starts.

```

sub    $0x28,%rsp

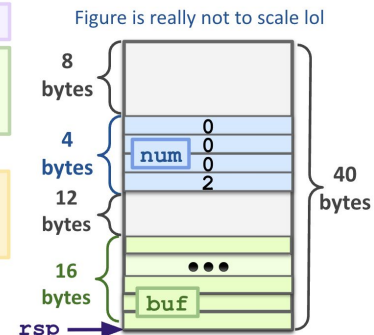
```

We know from the source code that `buf` is 16 bytes large, which leaves 12 bytes between the buffer's end and `num`.

```

callq  0x401050 <printf@plt>
mov     %rsp,%rax
mov     %rax,%rdi
callq  0x401060 <gets@plt>

```



5

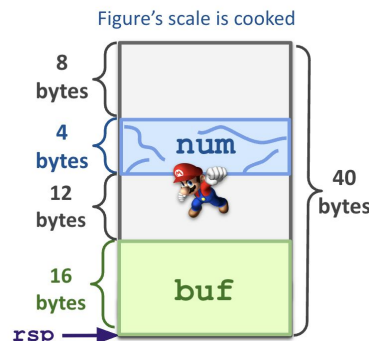
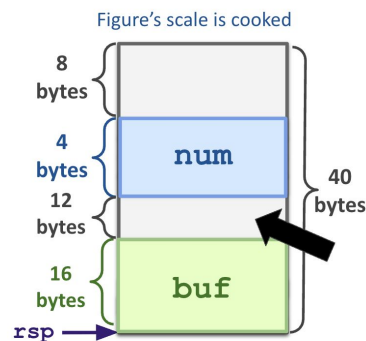
In the `greet` function, how much space (in bytes) is there on the stack between the end of `buf` and the start of `num`?

A: 12 bytes (middle gray area on right)

```

sub    $0x28,%rsp
movl   $0x2,0x1c(%rsp)
mov     $0x402008,%edi
mov     $0x0,%eax
callq  0x401050 <printf@plt>
mov     %rsp,%rax
mov     %rax,%rdi
callq  0x401060 <gets@plt>

```



Now, what happens when we write characters past `buf`?

And via buffer overflow, can we infiltrate the supervillain lair of `num`, the variable we return?





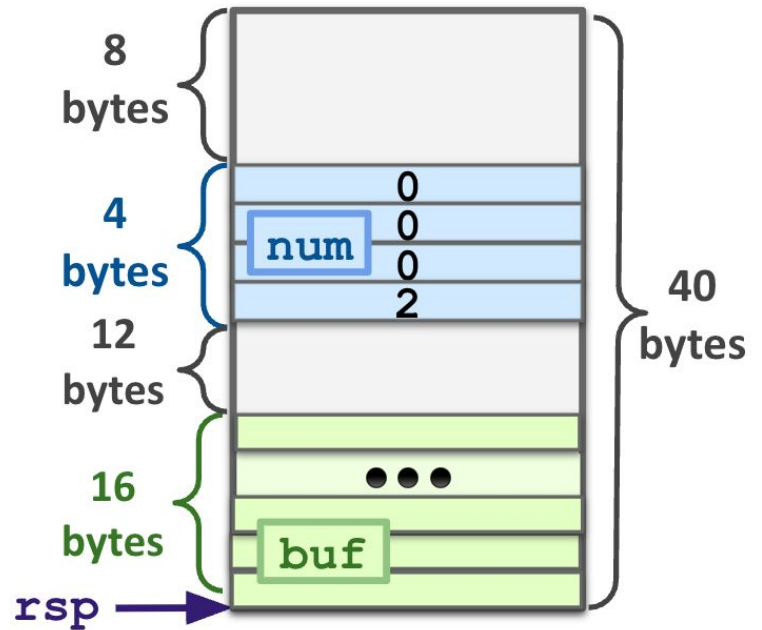
Solution to greet() stack exercise, and drawing out the key variables!
See page 1 / the slides on how the stack diagram was constructed :)

```
sub    $0x28,%rsp
movl   $0x2,0x1c(%rsp)
mov     $0x402008,%edi
mov     $0x0,%eax
callq  0x401050 <printf@plt>
mov     %rsp,%rax
mov     %rax,%rdi
callq  0x401060 <gets@plt>
```

```
# define BUF_LEN 16
int greet() {
    int num = 2;
    char buf[BUF_LEN];

    printf("What is your name?");
    if (gets(buf)) { ...
    // eventually, return num;
```

Figure's scale is cooked

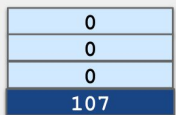


4-panel comic strip on how to override num to 107 with buffer overflow!

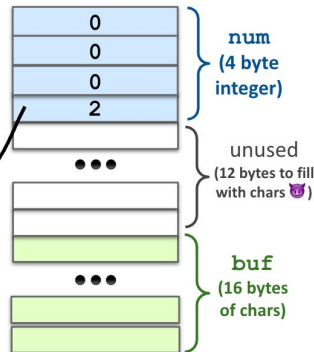
1

Here, we'll accomplish the impossible:
tricking `greet()` to return 107! 🍷

In other words, in memory, can we can
override the lowest byte of `num` to be 107?



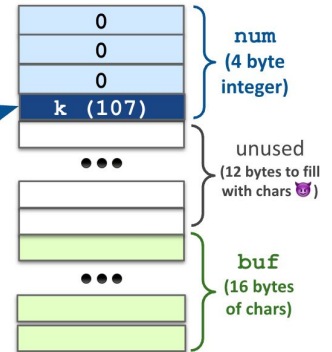
Note that `gets()` will start writing input
characters in `buf`, overflowing up to `num`.



2

Here, we'll accomplish the impossible:
tricking `greet()` to return 107! 🍷

Yes! Note that the ASCII character with
value 107 is `k`, so in the bytes that
belong to `num`, we have to place a `k`.
Due to `num` being an `int` data type, the
`k` will be interpreted / casted as 107.

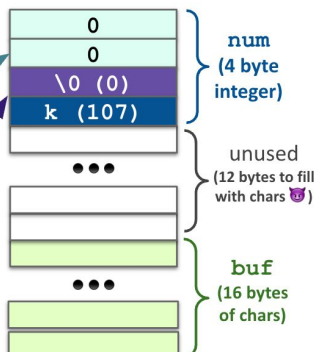


3

Here, we'll accomplish the impossible:
tricking `greet()` to return 107! 🍷

Note higher bytes stay 0 as `num` was
initialized to 2. To make `num` even
larger, you can override them also :)

Also, `gets()` follows the last character
with a `\0`, which keeps ASCII value 0!



4

Here, we'll accomplish the impossible:
tricking `greet()` to return 107! 🍷

The `buffer` + `unused` memory is
28 bytes, so our input should be a
string of 29 chars total, ending in `k` :)

Example: `aaaaaa...aak`
28 copies of `a`

