Getting Started

Download the tarball for this session. It will include the following files:

```
driver        64-bit executable
driver.c      C driver source
bomb.h        declaration for "bomb"
bomb.o        64-bit object code for "bomb"
```

The driver is pretty simple:

```
. . .
#include "bomb.h"

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

   if ( argc != 2 ) {
      printf("Must supply a string on the command line.\n");
      exit(1);
   }

   bomb(argv[1]);

   return 0;
}
```

Getting Started

Try running the driver a few times:

```
> driver hmmm
Segmentation fault (core dumped)
> driver pleasedontdothat
Segmentation fault (core dumped)
> driver whatdohyouwant?
Segmentation fault (core dumped)
```

The exercise is to determine the characteristics the command-line string must have in order to avoid triggering a segmentation fault…

… without having access to the source code for the function `bomb()`.

Debugging

A first thought might be to examine the program in gdb:

```
(gdb) break main
Breakpoint 1 at 0x4005cf: file driver.c, line 8.
(gdb) run hmmm
Starting program: driver hmmm

Breakpoint 1, main (argc=2, argv=0x7fffffffe0e8) at driver.c:8
8                   if ( argc != 2 ) {

(gdb) next
13                  bomb(argv[1]);

(gdb) p argv[1]
$1 = 0x7fffffffe3fa "hmmm"

(gdb) next

Program received signal SIGSEGV, Segmentation fault.
0x00000000004006b1 in bomb ()

(gdb) backtrace
#0  0x00000000004006b1 in bomb ()
#1  0x00000000004005fc in main (argc=2, argv=0x7fffffffe0e8) at driver.c:13
(gdb)
```

Stepping in Machine Code

Without the source for `bomb.c`, we can't step into the call in the usual way, but we can step through the machine code:

```
(gdb) break bomb
Breakpoint 2 at 0x400608
(gdb) run hmmm
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/wmcquain/2505/notes/gdb/bomb/driver hmmm

Breakpoint 1, main (argc=2, argv=0x7fffffffe0e8) at driver.c:8
8                   if ( argc != 2 ) {
(gdb) next
13                  bomb(argv[1]);
(gdb) step

Breakpoint 2, 0x0000000000400608 in bomb ()
(gdb) disassem
Dump of assembler code for function bomb:
   0x0000000000400604 <+0>:        push   %rbp
   0x0000000000400605 <+1>:        mov    %rsp,%rbp
=> 0x0000000000400608 <+4>:        sub    $0x30,%rsp
   0x000000000040060c <+8>:        mov    %rdi,-0x28(%rbp)
   0x0000000000400610 <+12>:       cmpq   $0x0,-0x28(%rbp)
```

Analyzing the Executable

At this point, the old frame pointer (`rbp`) for `main`'s stack frame has been saved to the stack, and `rbp` has been moved to the beginning of `bomb`'s frame:

```
(gdb) disassem
Dump of assembler code for function bomb:
   0x0000000000400604 <+0>:     push   %rbp
   0x0000000000400605 <+1>:     mov    %rsp,%rbp
=> 0x0000000000400608 <+4>:     sub    $0x30,%rsp
   0x000000000040060c <+8>:     mov    %rdi,-0x28(%rbp)
   0x0000000000400610 <+12>:    cmpq   $0x0,-0x28(%rbp)
```

We can step through the machine code, instruction by instruction, using `ni`:

```
(gdb) ni
0x000000000040060c in bomb ()
(gdb) disassem
Dump of assembler code for function bomb:
   0x0000000000400604 <+0>:     push   %rbp
   0x0000000000400605 <+1>:     mov    %rsp,%rbp
   0x0000000000400608 <+4>:     sub    $0x30,%rsp
=> 0x000000000040060c <+8>:     mov    %rdi,-0x28(%rbp)
   0x0000000000400610 <+12>:    cmpq   $0x0,-0x28(%rbp)
   0x0000000000400615 <+17>:    jne    0x40061c <bomb+24>
```

Disassembling the Code

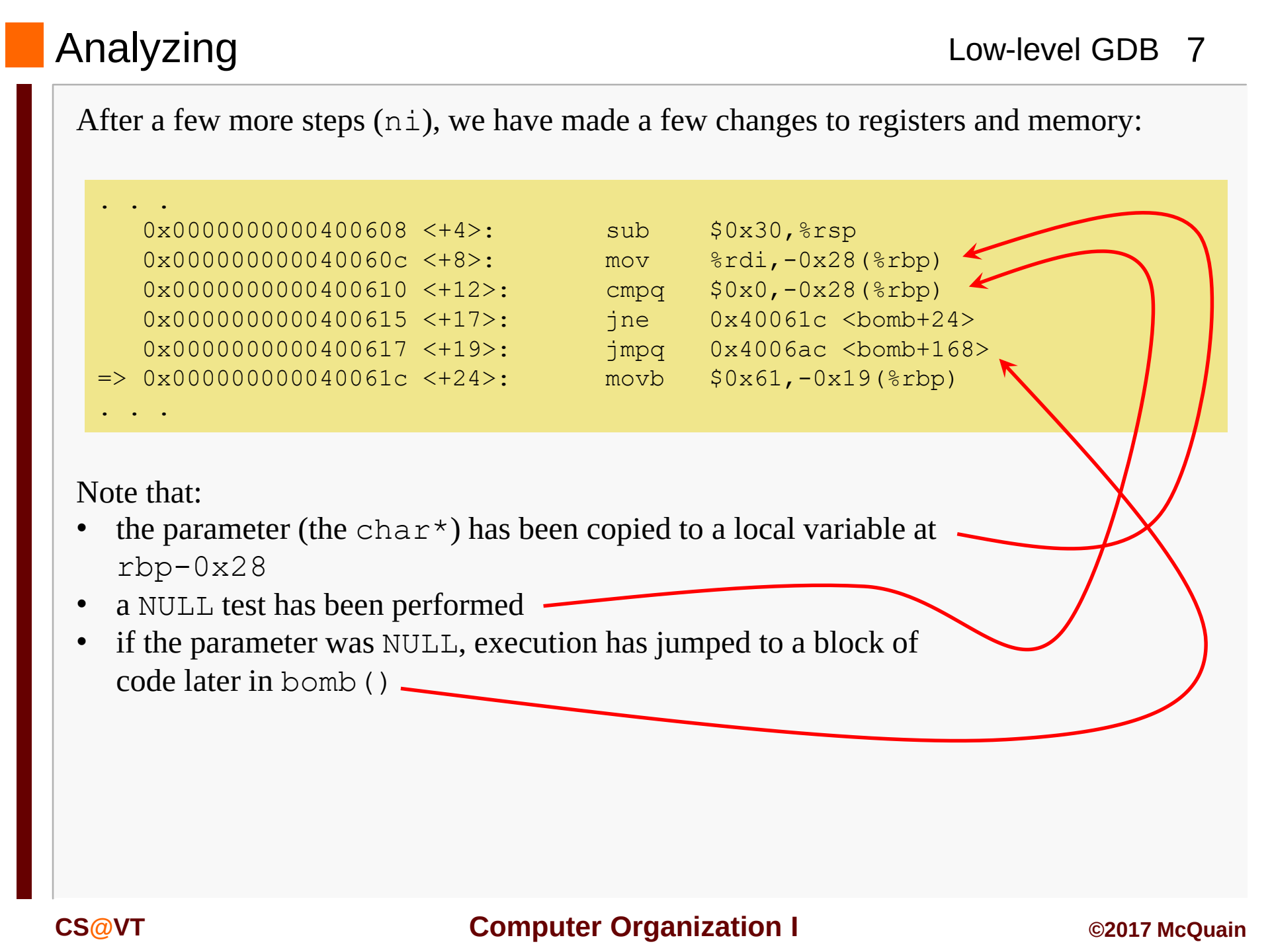
The `disassem` command lets us display an assembly language view of the code:

```
Dump of assembler code for function bomb:
   0x0000000000400604 <+0>:     push   %rbp
   0x0000000000400605 <+1>:     mov    %rsp,%rbp
   0x0000000000400608 <+4>:     sub    $0x30,%rsp
=> 0x000000000040060c <+8>:     mov    %rdi,-0x28(%rbp)
   0x0000000000400610 <+12>:    cmpq   $0x0,-0x28(%rbp)
   0x0000000000400615 <+17>:    jne    0x40061c <bomb+24>
   0x0000000000400617 <+19>:    jmpq   0x4006ac <bomb+168>
   0x000000000040061c <+24>:    movb   $0x61,-0x19(%rbp)
   0x0000000000400620 <+28>:    movb   $0x7a,-0x1a(%rbp)
   0x0000000000400624 <+32>:    movq   $0x0,-0x8(%rbp)
   0x000000000040062c <+40>:    movq   $0x0,-0x10(%rbp)
   0x0000000000400634 <+48>:    mov    -0x28(%rbp),%rax
   0x0000000000400638 <+52>:    mov    %rax,-0x18(%rbp)
   0x000000000040063c <+56>:    jmp    0x400677 <bomb+115>
   0x000000000040063e <+58>:    mov    -0x18(%rbp),%rax
   0x0000000000400642 <+62>:    movzbl (%rax),%eax
   0x0000000000400645 <+65>:    cmp    -0x19(%rbp),%al
   0x0000000000400648 <+68>:    jge    0x40064c <bomb+72>
   0x000000000040064a <+70>:    jmp    0x4006ac <bomb+168>
   0x000000000040064c <+72>:    mov    -0x18(%rbp),%rax
   0x0000000000400650 <+76>:    movzbl (%rax),%eax
. . .
```

Analyzing

After a few more steps (`ni`), we have made a few changes to registers and memory:

```
. . .
   0x0000000000400608 <+4>:     sub    $0x30,%rsp
   0x000000000040060c <+8>:     mov    %rdi,-0x28(%rbp)
   0x0000000000400610 <+12>:    cmpq   $0x0,-0x28(%rbp)
   0x0000000000400615 <+17>:    jne    0x40061c <bomb+24>
   0x0000000000400617 <+19>:    jmpq   0x4006ac <bomb+168>
=> 0x000000000040061c <+24>:    movb   $0x61,-0x19(%rbp)
. . .
```



Note that:

- the parameter (the `char*`) has been copied to a local variable at `rbp-0x28`
- a `NULL` test has been performed
- if the parameter was `NULL`, execution has jumped to a block of code later in `bomb()`

Analyzing

Let's see where that `jmpq` would take us:

```
. . .
   0x00000000004006ac <+168>:     mov    $0x0,%eax
   0x00000000004006b1 <+173>:     mov    (%rax),%rax
   0x00000000004006b4 <+176>:     mov    %rax,-0x8(%rbp)
. . .
```

(While you're in `disassem` you can hit return to see more code.)

```
    mov     $0x0,%eax          # eax = 0

    mov     (%rax),%rax        # rax = *eax = *NULL !!
```

So, the `jmpq` would take us to code that will dereference a `NULL` pointer, triggering a `segfault` error!!

Examining Values

Step through a few instructions at the beginning of the function:

```
. . .
   0x000000000040060c <+8>:     mov    %rdi,-0x28(%rbp)
   0x0000000000400610 <+12>:    cmpq   $0x0,-0x28(%rbp)
   0x0000000000400615 <+17>:    jne    0x40061c <bomb+24>
   0x0000000000400617 <+19>:    jmpq   0x4006ac <bomb+168>
=> 0x000000000040061c <+24>:    movb   $0x61,-0x19(%rbp)
. . .
```

Let's examine a few things:

```
. . .
(gdb) p/x $rbp
$2 = 0x7fffffffdfe0
(gdb) p/x $rbp-28
$3 = 0x7fffffffdfc4
. . .
```

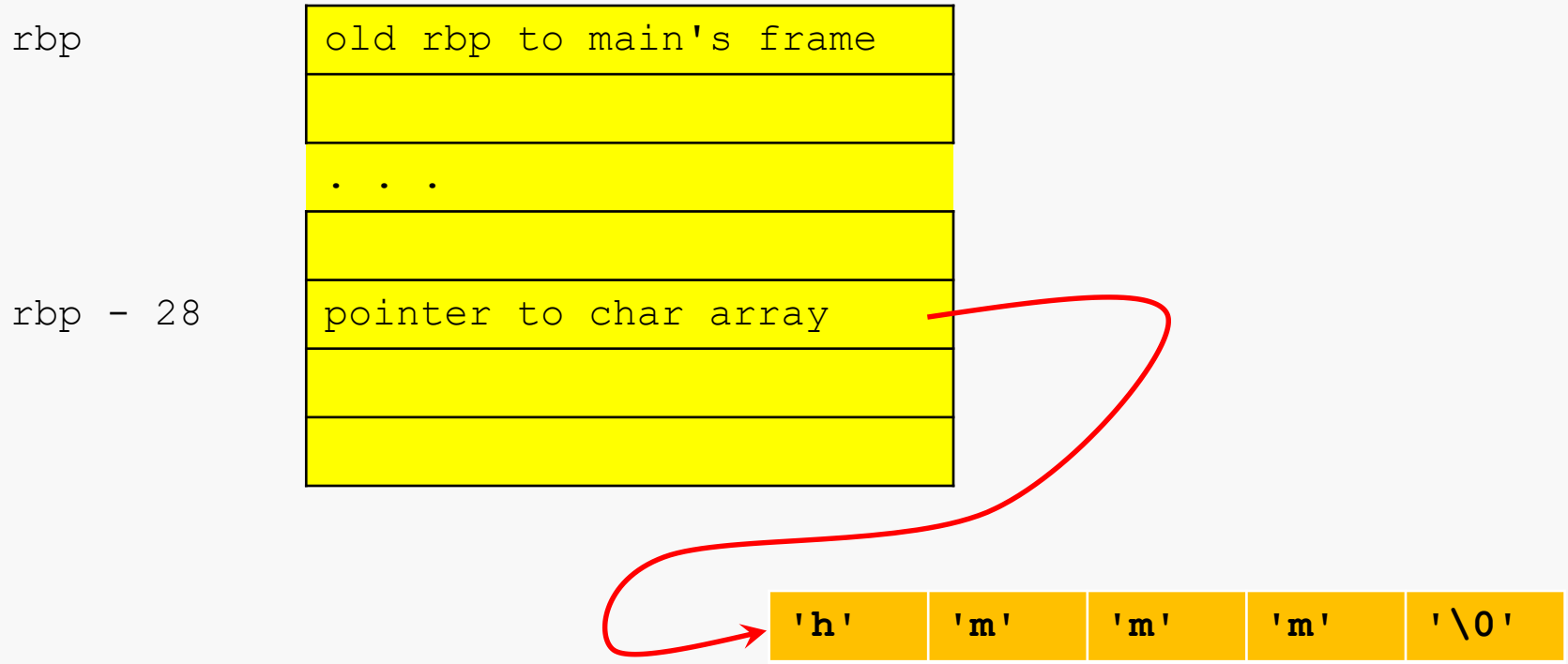
That makes some sense (`0xfe0 - 0x28 == 0xfc4`), but those are stack addresses.

We know that `$rbp - 28` is the `char*`… we should check what it's pointing to…

Examining Values

Let's think a bit:

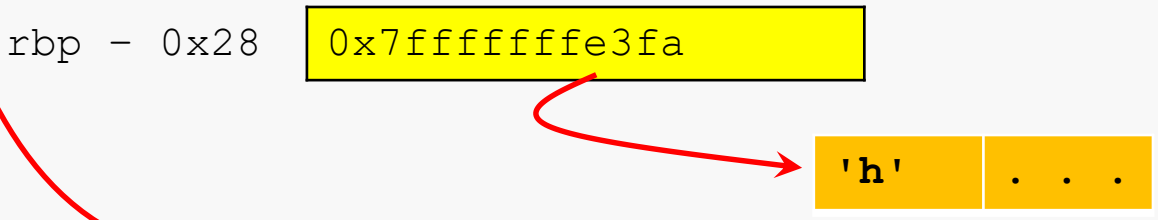
- `$rbp - 28` is the address of (a pointer to) the parameter, which is a `char*`
- `$rpb - 28` is a `char**`



Examining Values

Let's examine the details:

```
. . .
(gdb) p/x *(char**)($rbp - 0x28)
$12 = 0x7fffffffe3fa
. . .
```



$rbp – 0x28 is logically a char**

But gdb doesn't have any type information.

We have to typecast so that gdb "knows" that: (char) ($rbp – 0x28)**

Then we dereference to get the char* that points to the array.

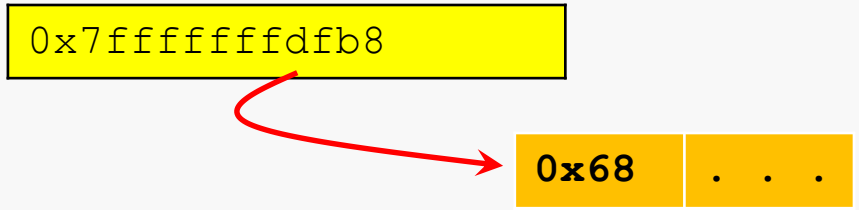
Here's the address of string[0]

Examining Values

Let's examine the details:

```
. . .
(gdb) p/x *(*(char**)($rbp - 0x28))
$9 = 0x68
. . .
```

Here's the value of string[0]



We already have the pointer to string[0].

We dereference that to get the value of string[0].

Confused?

Let's examine the details:

```
. . .
(gdb) p/x *(*(char**)($rbp - 0x28))
$9 = 0x68
. . .
```

```
$rbp - 0x28                 # address where the parameter to the function
                            # is stored on the stack; the parameter is a
                            # char*, so this is a pointer to a char*, so
                            # this is a char**
```

```
(char**)($rbp - 0x28)       # but gdb has no type information, so we must
                            # typecast to "tell" gdb this is a char**
```

```
*(char**)($rbp - 0x28)      # if we dereference, this gives us the value
                            # that $rbp - 0x28 points to; so, this gives
                            # the value of the parameter to the function;
                            # so this gives us a pointer to the char
                            # array set by the caller
```

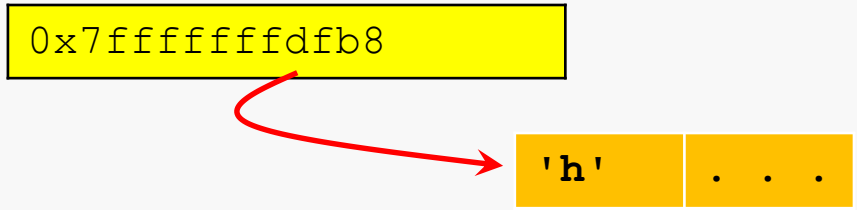
Examining Values

On the previous slides, we saw the value of `string[0]` as an ASCII code…

We can use another typecast to display that value as a character:

```
. . .
(gdb) p (char)*(*(char**)($rbp - 0x28))
$10 = 104 'h'
. . .
```

Here's the value of string[0], interpreted as a char



We must cast to display the value as a character.

Examining Values

So, first the function checks whether the parameter to `bomb()` is `NULL`:

```
. . .
   0x000000000040060c <+8>:     mov    %rdi,-0x28(%rbp)
   0x0000000000400610 <+12>:    cmpq   $0x0,-0x28(%rbp)
   0x0000000000400615 <+17>:    jne    0x40061c <bomb+24>
   0x0000000000400617 <+19>:    jmpq   0x4006ac <bomb+168>
   0x000000000040061c <+24>:    movb   $0x61,-0x19(%rbp)
. . .
```

If not, we jump here and initialize some local variable

If NULL, we jump here

```
. . .
   0x00000000004006ac <+168>:   mov    $0x0,%eax
   0x00000000004006b1 <+173>:   mov    (%rax),%rax
. . .
```

And dereference NULL

So, let's not pass in NULL

Examining the Code

Some locals are being set…

```
. . .
   0x000000000040061c <+24>:    movb   $0x61,-0x19(%rbp)
   0x0000000000400620 <+28>:    movb   $0x7a,-0x1a(%rbp)
   0x0000000000400624 <+32>:    movq   $0x0,-0x8(%rbp)
   0x000000000040062c <+40>:    movq   $0x0,-0x10(%rbp)
=> 0x0000000000400634 <+48>:    mov    -0x28(%rbp),%rax
. . .
```

The `movb` instructions are setting two **one-byte** local variables… to what?

```
. . .
(gdb) print (char) 0x61
$13 = 97 'a'
. . .
(gdb) print (char) 0x7a
$15 = 122 'z'
. . .
```

To the (ASCII codes for the) characters `'a'` and `'z'`.

Examining the Code

Some more locals are being set…

```
. . .
   0x000000000040061c <+24>:    movb   $0x61,-0x19(%rbp)
   0x0000000000400620 <+28>:    movb   $0x7a,-0x1a(%rbp)
   0x0000000000400624 <+32>:    movq   $0x0,-0x8(%rbp)
   0x000000000040062c <+40>:    movq   $0x0,-0x10(%rbp)
=> 0x0000000000400634 <+48>:    mov    -0x28(%rbp),%rax
. . .
```

The `movq` instructions are setting two local variables to zero. Counters, maybe?

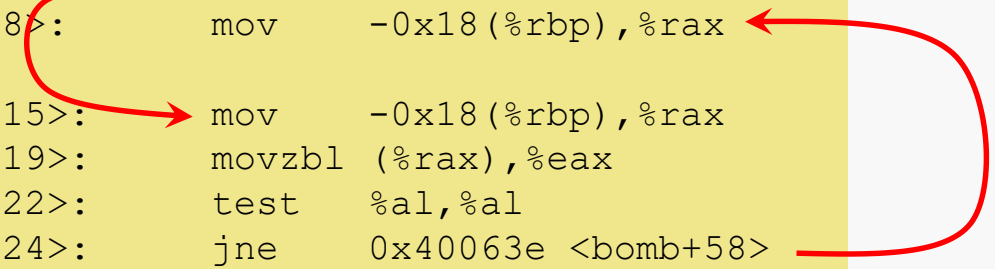
The `mov` instruction is setting `$rax` to point to the beginning of the **`char`** array.

Maybe the function is going to do a traversal…

Examining the Code

Examine the jump target and surrounding code:

```
. . .
   0x0000000000400638 <+52>:    mov    %rax,-0x18(%rbp)
   0x000000000040063c <+56>:    jmp    0x400677 <bomb+115>
   0x000000000040063e <+58>:    mov    -0x18(%rbp),%rax
. . .
   0x0000000000400677 <+115>:   mov    -0x18(%rbp),%rax
   0x000000000040067b <+119>:   movzbl (%rax),%eax
   0x000000000040067e <+122>:   test   %al,%al
   0x0000000000400680 <+124>:   jne    0x40063e <bomb+58>
. . .
```



This looks like a while loop…

Examining the Code: the Loop Test

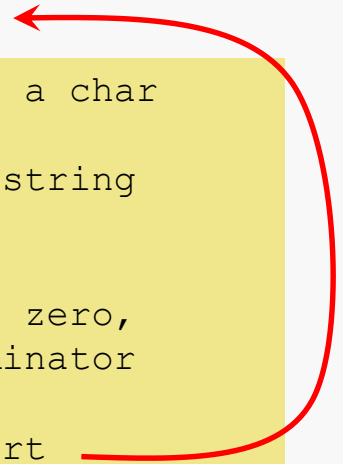
We thought that `rax` was pointing into the **`char`** array…

```
    mov    -0x18(%rbp),%rax        # let's assume rax points to a char

    movzbl (%rax),%eax             # eax = current char in the string

    test   %al,%al                 # al is the low byte of eax
                                   # this simply compares al to zero,
                                   #    whichis the string terminator

    jne    0x40063e <bomb+58>      # this jumps to the loop start
```

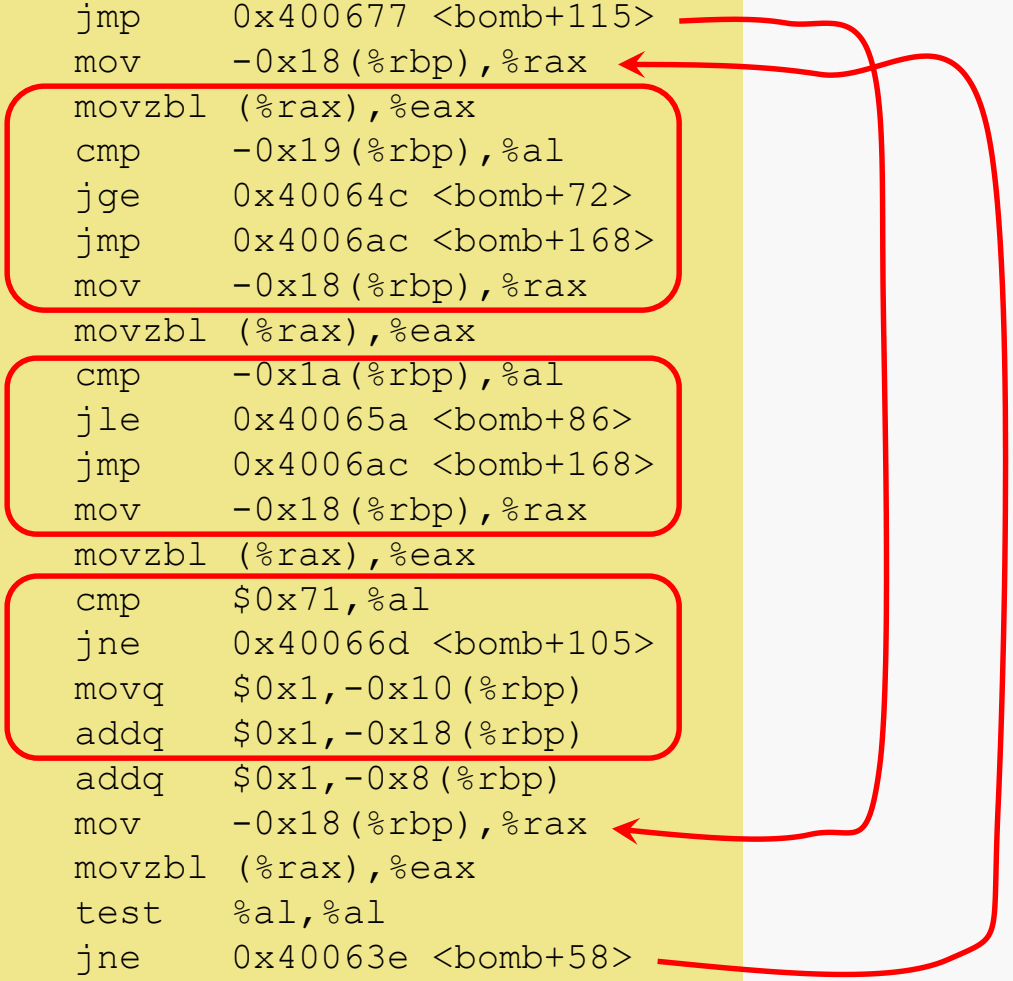


So, the loop seems to be traversing the **`char`** array until a terminator is found.

Examining the Code

Here's the apparent loop body:

```
. . .
   0x000000000040063c <+56>:    jmp    0x400677 <bomb+115>
   0x000000000040063e <+58>:    mov    -0x18(%rbp),%rax
   0x0000000000400642 <+62>:    movzbl (%rax),%eax
   0x0000000000400645 <+65>:    cmp    -0x19(%rbp),%al
   0x0000000000400648 <+68>:    jge    0x40064c <bomb+72>
   0x000000000040064a <+70>:    jmp    0x4006ac <bomb+168>
   0x000000000040064c <+72>:    mov    -0x18(%rbp),%rax
   0x0000000000400650 <+76>:    movzbl (%rax),%eax
   0x0000000000400653 <+79>:    cmp    -0x1a(%rbp),%al
   0x0000000000400656 <+82>:    jle    0x40065a <bomb+86>
   0x0000000000400658 <+84>:    jmp    0x4006ac <bomb+168>
   0x000000000040065a <+86>:    mov    -0x18(%rbp),%rax
   0x000000000040065e <+90>:    movzbl (%rax),%eax
   0x0000000000400661 <+93>:    cmp    $0x71,%al
   0x0000000000400663 <+95>:    jne    0x40066d <bomb+105>
   0x0000000000400665 <+97>:    movq   $0x1,-0x10(%rbp)
   0x000000000040066d <+105>:   addq   $0x1,-0x18(%rbp)
   0x0000000000400672 <+110>:   addq   $0x1,-0x8(%rbp)
   0x0000000000400677 <+115>:   mov    -0x18(%rbp),%rax
   0x000000000040067b <+119>:   movzbl (%rax),%eax
   0x000000000040067e <+122>:   test   %al,%al
   0x0000000000400680 <+124>:   jne    0x40063e <bomb+58>
. . .
```

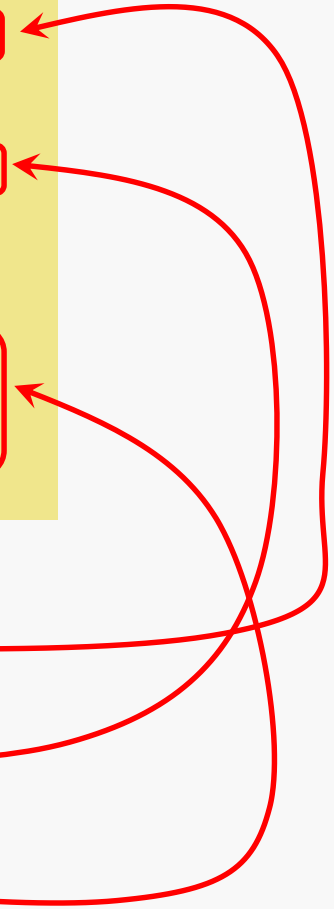


Examining the Code

Let's examine the loop control:

```
. . .
   0x000000000040063c <+56>:    jmp    0x400677 <bomb+115>
   0x000000000040063e <+58>:    mov    -0x18(%rbp),%rax
   0x0000000000400642 <+62>:    movzbl (%rax),%eax
. . .
   0x000000000040066d <+105>:   addq   $0x1,-0x18(%rbp)
   0x0000000000400672 <+110>:   addq   $0x1,-0x8(%rbp)

   0x0000000000400677 <+115>:   mov    -0x18(%rbp),%rax
   0x000000000040067b <+119>:   movzbl (%rax),%eax
   0x000000000040067e <+122>:   test   %al,%al
   0x0000000000400680 <+124>:   jne    0x40063e <bomb+58>
. . .
```



$rbp – 0x18 holds a pointer into the char array

We are stepping to the next character in the array here

If that character is not 0 ('\0'), we continue the loop

Examining the Code

This looks like a control structure, maybe an if…:

```
. . .
   0x0000000000400642 <+62>:    movzbl (%rax),%eax
   0x0000000000400645 <+65>:    cmp    -0x19(%rbp),%al
   0x0000000000400648 <+68>:    jge    0x40064c <bomb+72>
   0x000000000040064a <+70>:    jmp    0x4006ac <bomb+168>
   0x000000000040064c <+72>:    mov    -0x18(%rbp),%rax
. . .
```

Fetch current char from the string

Compare it to 'a'

If >= 'a', proceed

If not… boom!

So, our string had better not contain any characters that precede 'a' (in ASCII ordering).

Examining the Code

Looks like another if:

```
. . .
   0x0000000000040064c <+72>:       mov    -0x18(%rbp),%rax
   0x0000000000400650 <+76>:       movzbl (%rax),%eax
   0x0000000000400653 <+79>:       cmp    -0x1a(%rbp),%al
   0x0000000000400656 <+82>:       jle    0x40065a <bomb+86>
   0x0000000000400658 <+84>:       jmp    0x4006ac <bomb+168>
   0x000000000040065a <+86>:       mov    -0x18(%rbp),%rax
. . .
```

Fetch current char from the string

Compare it to 'z'

If <= 'z', proceed

If not… boom!

So, our string had better not contain any characters that follow 'z' (in ASCII ordering).

So, our string must contain only lower-case letters…

But… two of our earlier test strings satisfied that and we still blew up…

Examining the Code

There's another if:

```
. . .
   0x000000000040065a <+86>:    mov    -0x18(%rbp),%rax
   0x000000000040065e <+90>:    movzbl (%rax),%eax
   0x0000000000400661 <+93>:    cmp    $0x71,%al
   0x0000000000400663 <+95>:    jne    0x40066d <bomb+105>
   0x0000000000400665 <+97>:    movq   $0x1,-0x10(%rbp)
   0x000000000040066d <+105>:   addq   $0x1,-0x18(%rbp)
. . .
```

Fetch current char from the string

Compare it to ?

Not equal, proceed

Equal, set a flag?

What's 0x71?

```
. . .
(gdb) print (char) 0x71
$16 = 113 'q'
. . .
```

Maybe the string must contain a 'q'? Is the value set at $rbp – 0x10 used later?

Examining the Code

Here's where the value at $rbp = 0x10 is checked:

```
. . .
   0x000000000040068b <+135>:     cmpq   $0x0,-0x10(%rbp)
   0x0000000000400690 <+140>:     jne    0x400694 <bomb+144>
   0x0000000000400692 <+142>:     jmp    0x4006ac <bomb+168>
   0x0000000000400694 <+144>:     mov    -0x28(%rbp),%rax
. . .
```

Is flag == 0?

No, proceed

Yes, bad news

So, the string must contain a `'q'`.

Let's try that…

```
> driver bequiet
Segmentation fault (core dumped)
```

So, there must be at least one more constraint…