15-213/15-513/14-513 Recitation: Bomb Lab Cont.

Sep 15, 2023

Agenda

- Reminders
- OH Etiquette Reminder
- Bomb Lab Continuation

Reminders

- Bomb Lab is due next Thursday (Sep 21)
- Written 1 Peer Reviews has gone out

OH Etiquette Reminder

- **Good description:** “On phase 3. I’ve figured out that we need some string and an int, but I’m not sure how to figure out more info”
- **Bad description: “**bomb lab help”
  - (We know that already- that’s why you’re at OH. Write more details!)

- **If you are in the queue with a bad description, you are actually not in the queue**

OH Etiquette Reminder

- **Professors have office hours too! (they don't bite, we promise)**
  - They get lonely…

Bomblab!!

Bomb Hints

- ***Dr. Evil*** may be evil, but he isn't cruel. You may assume that functions do what their name implies
  - i.e. phase_1() is most likely the first phase. printf() is just printf(). If there is an explode_bomb() function, it would probably help to set a breakpoint there!
- Use the man pages for library functions.
  - Although you can examine the assembly for snprintf(), we assure you that it's easier to use the man pages ($ man snprintf) than to decipher assembly code for system calls.
- Most cryptic function calls you'll see (e.g. callq … <_exit@plt>) are also calls to C library functions.
  - You can safely ignore the @plt as that refers to dynamic linking.

Objdump

- $ objdump -d [name of executable] > [any file name]
  - Saves the assembly code of the executable into the file.
  - Feel free to annotate the assembly in your favorite text editor.

```
[dalud@angelshark:~/.../15213/s17/bomb16] $ objdump -d example > example.asm
```

```
0000000000400560 <function>:
  400560: 48 83 ec 18           sub    $0x18,%rsp              // Setting things up
  400564: 48 89 7c 24 08        mov    %rdi,0x8(%rsp)
  400569: 48 83 7c 24 08 00     cmpq   $0x0,0x8(%rsp)          // Checks $rdi against 0
  40056f: 74 0a                 je     40057b <function+0x1b>  // Jumps to the "if branch" if equal
  400571: b8 00 00 00 00        mov    $0x0,%eax
  400576: e8 0a 00 00 00        callq  400585 <quit>           // Calls "quit" (else branch)
  40057b: b8 01 00 00 00        mov    $0x1,%eax               // Makes $rax = 1 (if branch)
  400580: 48 83 c4 18           add    $0x18,%rsp              // Cleaning stuff up
  400584: c3                    retq                           // Looks like the return value is 1
```

Assembly Reminders

- Operand Types
  - Immediate: $0x400, $-533 → a constant prefixed with $
  - Register: %rax, %r12 → value in the register
  - Memory: (%rax) → memory at address given by register

- D(Rb, Ri, S) = Mem[Reg[Rb] + S*Reg[Ri] + D]
  - D: constant displacement
  - Rb: base register (any of the 16 registers)
  - Ri: index register (any except for %rsp)
  - S: scale (1, 2, 4, or 8)
  - $100(%rax, %rsi, 2)

Basic GDB tips

- Many commands have shortcuts. Dissasemble → disas. Disable → dis
  - Do not mix these up! Disable will disable all your breakpoints, which may cause you to blow up your bomb.

- (gdb) print [any valid C expression]
  - This can be used to study any kind of local variable or memory location
  - Use casting to get the right type (e.g. print *(long *)ptr)

- (gdb) x [some format specifier] [some memory address]
  - Examines memory. See the handout for more information. Same as print *(addr), but more convenient.

- (gdb) set disassemble-next-line on
  (gdb) show disassemble-next-line
  - Shows the next assembly instruction after each step instruction

- (gdb) info registers — Shows the values of the registers
- (gdb) info breakpoints — Shows all current breakpoints
- (gdb) quit — Exits gdb

Text User Interface (TUI) mode

WARNING – do not use!

Although the TUI mode is very convenient, it has been known to accidentally set off student's bombs during Bomblab (but is fine for future labs like malloc).

The course staff is not responsible if your bomb goes off due to the TUI, and will not remove the explosion from Autolab.

What to do

- Don't understand what a big block of assembly does? GDB
- Need to figure out what's in a specific memory address? GDB
- Can't trace how 4 – 6 registers are changing over time? GDB
- Have no idea how to start the assignment? Writeup
- Need to know how to use certain GDB commands? Writeup, Bootcamp
  - Also useful: http://csapp.cs.cmu.edu/3e/docs/gdbnotes-x86-64.pdf
- Don't know what an assembly instruction does? Lecture slides
- Confused about control flow or stack discipline? Lecture slides

Activity

Form Pairs

- One student needs a laptop
- SSH into a shark machine and type these commands:
- $ wget http://www.cs.cmu.edu/~213/activities/rec4.tar
- $ tar xvpf rec4.tar
- $ cd rec4
- $ make
- $ gdb act1

Activity Walkthroughs

Source code for Activity 1 (Abridged)

```
#include <stdio.h>

int main(int argc, char** argv) {
    int ret = printf("%s\n", argv[argc-1]);
    return ret;  // number of characters printed
}

// Follow along on the handout!
```

Activity 1 trace

- (gdb) disassemble main // show the assembly instructions in main
- (gdb) print (char*) [0x4…] // hex code from <+18>
  // prints a string

- Find the seemingly random $0x… value in the assembly code
- Q: Does the printed value correspond to anything in the C code?

- (gdb) break main
- (gdb) run 15213
- (gdb) print argv[1] // Q: What does this print out?
- (gdb) continue
- (gdb) quit // exit GDB; agree to kill the running process

Activity 3

- Activity 3 has a Bomb Lab feel to it. It will print out “good args!” if you type in the right numbers into the command line. Use GDB to find what numbers to use, and if you get stuck, look at the handout.

- $ cat act3.c // display the source code of act3
- $ gdb act3

- Q. Which register holds the return value from a function?
- (Hint: Use disassemble in main and look at what register is used right after the function call to compare)

Stacks Review (AttackLab Prep)

Manipulating the stack

What instructions do we typically use to change the stack pointer, %rsp?

Growing the stack:

Shrinking the stack:

Manipulating the stack

What instructions do we typically use to change the stack pointer, %rsp?

Growing the stack:

- `sub $0x28, %rsp`
- `push %rbx`
- `callq my_function`

Shrinking the stack:

Manipulating the stack

What instructions do we typically use to change the stack pointer, %rsp?

Growing the stack:

- `sub $0x28, %rsp`
- `push %rbx`
- `callq my_function`

Shrinking the stack:

- `add $0x28, %rsp`
- `pop %rbx`
- `retq`

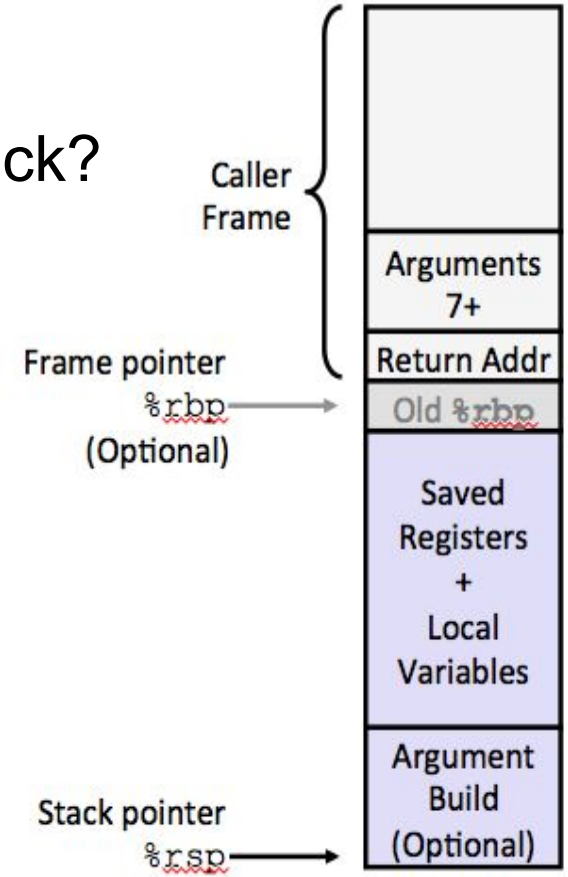
x86-64 Stack Frames

What kinds of data are stored on the stack?

x86-64 Stack Frames

What kinds of data are stored on the stack?

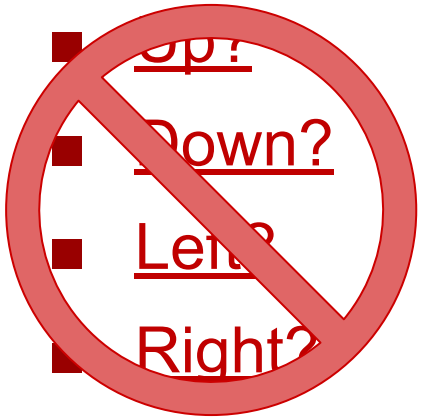
- Saved registers
- Local variables
- Arguments (7+)
- Saved return address



Which way does the stack grow?

- Up?
- Down?
- Left?
- Right?

Which way does the stack grow?



It depends on how you draw it!

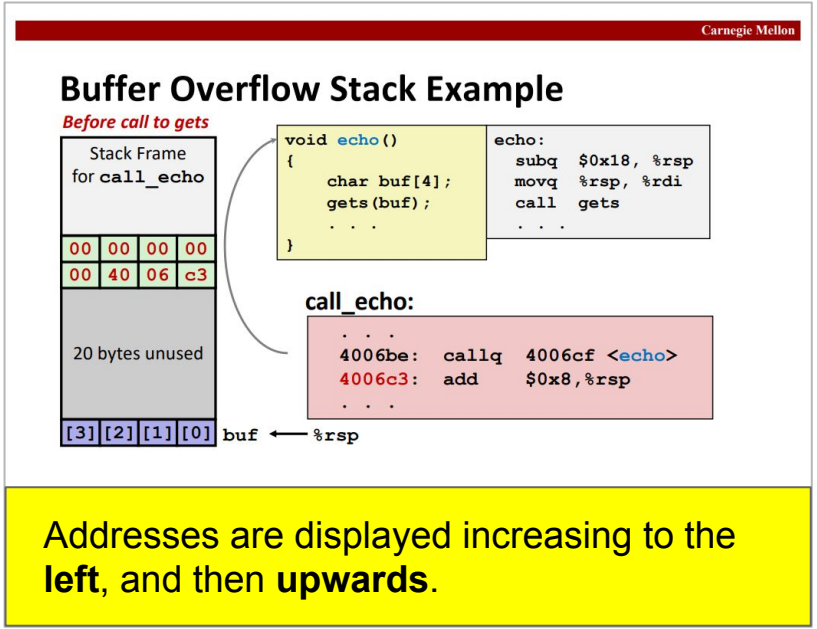
The stack always grows towards **lower addresses** in x86-64.

(Informally, this usually means "down".)

Be aware of this possible ambiguity when reading diagrams.

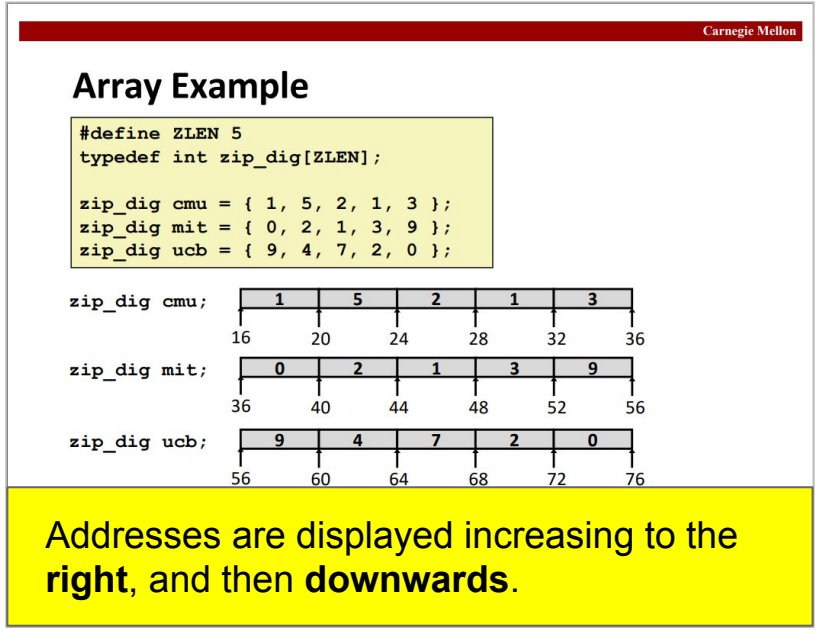
Drawing memory

Stack diagrams



Addresses are displayed increasing to the **left**, and then **upwards**.

Everything else



Addresses are displayed increasing to the **right**, and then **downwards**.

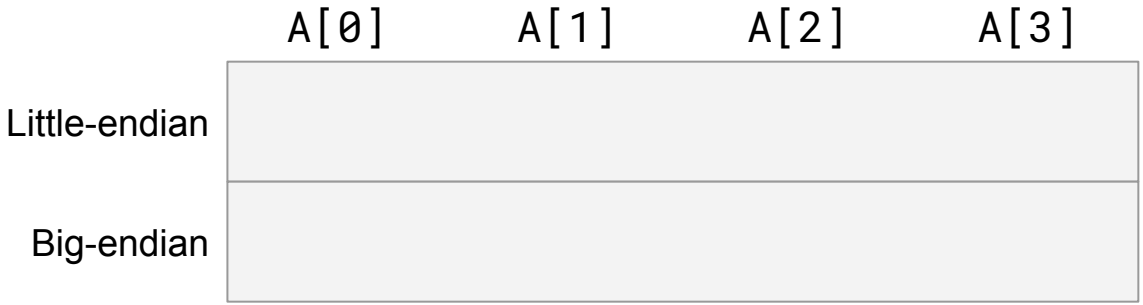
Endianness

- Describes how integers are represented as bytes.
- Little-endian means that the least-significant 8 bits of an integer are stored at the lowest address.

32-bit integer

`0x01020304`

"Big end" "Little end"



Endianness

- Describes how integers are represented as bytes.
- Little-endian means that the least-significant 8 bits of an integer are stored at the lowest address.

32-bit integer

0x01020304

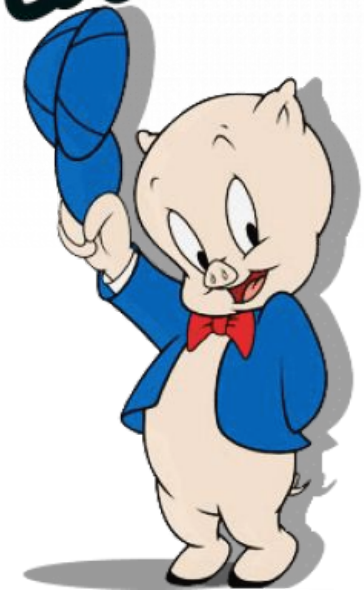
"Big end" "Little end"

	A[0]	A[1]	A[2]	A[3]
Little-endian	0x04	0x03	0x02	0x01
Big-endian	0x01	0x02	0x03	0x04

Questions?

- Ask on piazza or come to OH for longer questions
- Have a list of things your tried before you come to OH
- Notice how we asked you to read the write-up like 4 times already? That's because we want you to **read the write-up**!
- Good luck agents :) The fate of your computer realms are in your hands

"That's all Folks!"



Appendix

- Assembly help
- Problem walkthroughs

Quick Assembly Info

- $rdi holds the first argument to a function call, $rsi holds the second argument, and $rax will hold the return value of the function call.
- Many functions start with “push %rbx” and end with “pop %rbx”. Long story short, this is because %rbx is “callee-saved”.
- The stack is often used to hold local variables
  - Addresses in the stack are usually in the 0x7fffffff… range
- Know how $rax is related to $eax and $al.
- Most cryptic function calls you’ll see (e.g. callq … <_exit@plt>) are calls to C library functions. If necessary, use the Unix man pages to figure out what the functions do.

Source code for Activity 2 (Abridged)

```
#include <string.h>

int stc(char*, char*);  // Defined in a separate assembly file
int main(int argc, char** argv) {
        int ret = stc("15213", argv[argc-1]);
        argv[0] = '\0';       // Forces gcc to generate a callq instead of jmp
return ret;
}

// Follow along on the handout!
```

Activity 2 trace

- $ gdb act2
- (gdb) break main
- (gdb) disas main
- (gdb) run
- (gdb) print /x $rsi // '/x' means print in hexadecimal
- (gdb) print /x $rdi
- Q. RDI and RSI are registers that pass the first two arguments. Looking at their values, which is the first argument to main (the 'argc' argument)? Why?

- (gdb) disassemble main // note the call to stc at <+17>
- (gdb) break stc // main calls the stc function, so we'll study that function too
- (gdb) continue
- Q. How could you view the arguments that have been passed to stc?
  - Try both of these: "print /x $rdi", "x /s $rdi"

Activity 2 trace

- (gdb) run 18213 // gdb will ask if you want to restart; choose yes

- (gdb) continue // Q. Which function is in execution now?

- (gdb) disassemble // note the “=>” on the left side
- (gdb) nexti // step through a single x86 instruction
- (gdb) // just press enter 3 to 4 times
  - GDB will repeat your previous instruction. Useful for single-stepping.

- (gdb) disassemble
- Q. Now where are the “=>” characters printed?
- (gdb) quit

Activity 3 trace

- (gdb) run 5208 10000
- About to run push %rbx
- $rdi = 5208
- $rsi = 10000
- $rbx = [$rbx from somewhere else]
- $rax = [garbage value]

- Stack:

[some old stack items]

- (gdb) nexti

```
push   %rbx
mov    %rdi,%rbx
add    $0x5,%rbx
add    %rsi,%rbx
cmp    $0x3b6d,%rbx
sete   %al
movzbq %al,%rax
pop    %rbx
retq
```

Activity 3 trace

- About to run mov %rdi, %rbx
- $rdi = 5208
- $rsi = 10000
- $rbx = [$rbx from somewhere else]
- $rax = [garbage value]

- Stack:

[$rbx from somewhere else]
[some old stack items]

- (gdb) nexti

```
push   %rbx
mov    %rdi,%rbx
add    $0x5,%rbx
add    %rsi,%rbx
cmp    $0x3b6d,%rbx
sete   %al
movzbq %al,%rax
pop    %rbx
retq
```

Activity 3 trace

- About to run add $0x5, %rbx
- $rdi = 5208
- $rsi = 10000
- $rbx = 5208
- $rax = [garbage value]

- Stack:

[$rbx from somewhere else]
[some old stack items]

- (gdb) nexti

```
push   %rbx
mov    %rdi,%rbx
add    $0x5,%rbx
add    %rsi,%rbx
cmp    $0x3b6d,%rbx
sete   %al
movzbq %al,%rax
pop    %rbx
retq
```

Activity 3 trace

- About to run add %rsi, %rbx
- $rdi = 5208
- $rsi = 10000
- $rbx = 5213
- $rax = [garbage value]

- Stack:

[$rbx from somewhere else]
[some old stack items]

- (gdb) nexti

```
push   %rbx
mov    %rdi,%rbx
add    $0x5,%rbx
add    %rsi,%rbx
cmp    $0x3b6d,%rbx
sete   %al
movzbq %al,%rax
pop    %rbx
retq
```

Activity 3 trace

- About to run cmp 0x3b6d, %rbx
  & other instructions
- $rdi = 5208
- $rsi = 10000
- $rbx = 15213 (= 0x3b6d)
- $rax = [garbage value]

- Stack:

[$rbx from somewhere else]

[some old stack items]

- (gdb) nexti
- (gdb) nexti
- (gdb) nexti

```
push   %rbx
mov    %rdi,%rbx
add    $0x5,%rbx
add    %rsi,%rbx
cmp    $0x3b6d,%rbx
sete   %al
movzbq %al,%rax
pop    %rbx
retq
```

Activity 3 trace

- About to run pop %rbx
- $rdi = 5208
- $rsi = 10000
- $rbx = 15213 = 0x3b6d
- $rax = 1

- Stack:

[$rbx from somewhere else]
[some old stack items]

- (gdb) nexti

```
push   %rbx
mov    %rdi,%rbx
add    $0x5,%rbx
add    %rsi,%rbx
cmp    $0x3b6d,%rbx
sete   %al
movzbq %al,%rax
pop    %rbx
retq
```

Activity 3 trace

- About to run retq
- $rdi = 5208
- $rsi = 10000
- $rbx = [$rbx from somewhere else]
- $rax = 1

```
push   %rbx
mov    %rdi,%rbx
add    $0x5,%rbx
add    %rsi,%rbx
cmp    $0x3b6d,%rbx
sete   %al
movzbq %al,%rax
pop    %rbx
retq
```

- Stack:

[some old stack items]

Activity 4

Use what you have learned to get act4 to print “Finish.”

The source code is available in act4.c if you get stuck. Also, you can ask TAs for help understanding the assembly code.

Activity 4 trace

- $ gdb act4
- (gdb) disassemble main
- Note 3 functions called: strtoq, compute, fwrite
- If you look at the strtoq man page:
    - convert a string to a long integer
- Fwrite is probably a print function. Print values stored into $rdi immediately before calling fwrite
    - Why are they put into $rdi?
- Look at addresses at <+60> and <+94>, may be different when you do this
- (gdb) x /s 0x4942c0
    - “Please rerun with a positive number argument\n”
- (gdb) x /s 0x4942f0
    - “Argument was not a positive integer

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000400af0 <+0>:     sub    $0x8,%rsp
   0x0000000000400af4 <+4>:     cmp    $0x1,%edi
   0x0000000000400af7 <+7>:     je     0x400b1b <main+43>
   0x0000000000400af9 <+9>:     mov    0x8(%rsi),%rdi
   0x0000000000400afd <+13>:    mov    $0xa,%edx
   0x0000000000400b02 <+18>:    xor    %esi,%esi
   0x0000000000400b04 <+20>:    callq  0x401e80 <strtoq>
   0x0000000000400b09 <+25>:    test   %eax,%eax
   0x0000000000400b0b <+27>:    js     0x400b3d <main+77>
   0x0000000000400b0d <+29>:    mov    %eax,%edi
   0x0000000000400b0f <+31>:    callq  0x400f20 <compute>
   0x0000000000400b14 <+36>:    xor    %eax,%eax
   0x0000000000400b16 <+38>:    add    $0x8,%rsp
   0x0000000000400b1a <+42>:    retq
   0x0000000000400b1b <+43>:    mov    0x2bcc46(%rip),%rcx        # 0x6bd768 <stderr>
   0x0000000000400b22 <+50>:    mov    $0x2d,%edx
   0x0000000000400b27 <+55>:    mov    $0x1,%esi
   0x0000000000400b2c <+60>:    mov    $0x4942c0,%edi
   0x0000000000400b31 <+65>:    callq  0x4025b0 <fwrite>
   0x0000000000400b36 <+70>:    mov    $0x1,%eax
   0x0000000000400b3b <+75>:    jmp    0x400b16 <main+38>
   0x0000000000400b3d <+77>:    mov    0x2bcc24(%rip),%rcx        # 0x6bd768 <stderr>
   0x0000000000400b44 <+84>:    mov    $0x24,%edx
   0x0000000000400b49 <+89>:    mov    $0x1,%esi
   0x0000000000400b4e <+94>:    mov    $0x4942f0,%edi
   0x0000000000400b53 <+99>:    callq  0x4025b0 <fwrite>
   0x0000000000400b58 <+104>:   mov    $0x1,%eax
   0x0000000000400b5d <+109>:   jmp    0x400b16 <main+38>
End of assembler dump.
```

Activity 4 trace

- (gdb) disassemble compute
- We want it to print “Finish”. Note that the code jumps to <puts> at <+85>. Print the value stored into $rdi immediately before <+80>
- (gdb) x /s 0x494290
  - ”Finish”
- Want to get to either <+77> or <+80>
  - What happens if we get to <+75>?
- Because of <+75>, we know we have to jump to get to the puts jump at <+85>

```
(gdb) disassemble compute
Dump of assembler code for function compute:
   0x0000000000400f20 <+0>:     lea    (%rdi,%rdi,2),%eax
   0x0000000000400f23 <+3>:     mov    %eax,%edx
   0x0000000000400f25 <+5>:     and    $0xf,%edx
   0x0000000000400f28 <+8>:     nopl   0x0(%rax,%rax,1)
   0x0000000000400f30 <+16>:    cmp    $0x4,%edx
   0x0000000000400f33 <+19>:    ja     0x400f53 <compute+51>
   0x0000000000400f35 <+21>:    jmpq   *0x494298(,%rdx,8)
   0x0000000000400f3c <+28>:    nopl   0x0(%rax)
   0x0000000000400f40 <+32>:    and    $0x1,%eax
   0x0000000000400f43 <+35>:    mov    %eax,%edx
   0x0000000000400f45 <+37>:    jmp    0x400f30 <compute+16>
   0x0000000000400f47 <+39>:    nopw   0x0(%rax,%rax,1)
   0x0000000000400f50 <+48>:    sar    $0x2,%eax
   0x0000000000400f53 <+51>:    mov    %eax,%edx
   0x0000000000400f55 <+53>:    and    $0xf,%edx
   0x0000000000400f58 <+56>:    test   %eax,%eax
   0x0000000000400f5a <+58>:    jns    0x400f30 <compute+16>
   0x0000000000400f5c <+60>:    repz retq
   0x0000000000400f5e <+62>:    xchg   %ax,%ax
   0x0000000000400f60 <+64>:    add    %eax,%eax
   0x0000000000400f62 <+66>:    jmp    0x400f53 <compute+51>
   0x0000000000400f64 <+68>:    nopl   0x0(%rax)
   0x0000000000400f68 <+72>:    sub    $0x1,%eax
   0x0000000000400f6b <+75>:    jmp    0x400f53 <compute+51>
   0x0000000000400f6d <+77>:    nopl   (%rax)
   0x0000000000400f70 <+80>:    mov    $0x494290,%edi
   0x0000000000400f75 <+85>:    jmpq   0x4027d0 <puts>
End of assembler dump.
```

Activity 4 trace

- There are 7 jumps. 3 to <+51>, 2 to <+16>, 1 to <puts>, and then:
  - jmpq *0x494298(,%rdx,8)
  - Should jump to address *0x494298 + 8 * $rdx
  - Note, may be different when you do this
- (gdb) x /x *0x494298
  - 0x400f70 <compute+80>
- The only way this get us to where we want to go is if $rdx = 0.

```
(gdb) disassemble compute
Dump of assembler code for function compute:
   0x0000000000400f20 <+0>:     lea    (%rdi,%rdi,2),%eax
   0x0000000000400f23 <+3>:     mov    %eax,%edx
   0x0000000000400f25 <+5>:     and    $0xf,%edx
   0x0000000000400f28 <+8>:     nopl   0x0(%rax,%rax,1)
   0x0000000000400f30 <+16>:    cmp    $0x4,%edx
   0x0000000000400f33 <+19>:    ja     0x400f53 <compute+51>
   0x0000000000400f35 <+21>:    jmpq   *0x494298(,%rdx,8)
   0x0000000000400f3c <+28>:    nopl   0x0(%rax)
   0x0000000000400f40 <+32>:    and    $0x1,%eax
   0x0000000000400f43 <+35>:    mov    %eax,%edx
   0x0000000000400f45 <+37>:    jmp    0x400f30 <compute+16>
   0x0000000000400f47 <+39>:    nopw   0x0(%rax,%rax,1)
   0x0000000000400f50 <+48>:    sar    $0x2,%eax
   0x0000000000400f53 <+51>:    mov    %eax,%edx
   0x0000000000400f55 <+53>:    and    $0xf,%edx
   0x0000000000400f58 <+56>:    test   %eax,%eax
   0x0000000000400f5a <+58>:    jns    0x400f30 <compute+16>
   0x0000000000400f5c <+60>:    repz retq
   0x0000000000400f5e <+62>:    xchg   %ax,%ax
   0x0000000000400f60 <+64>:    add    %eax,%eax
   0x0000000000400f62 <+66>:    jmp    0x400f53 <compute+51>
   0x0000000000400f64 <+68>:    nopl   0x0(%rax)
   0x0000000000400f68 <+72>:    sub    $0x1,%eax
   0x0000000000400f6b <+75>:    jmp    0x400f53 <compute+51>
   0x0000000000400f6d <+77>:    nopl   (%rax)
   0x0000000000400f70 <+80>:    mov    $0x494290,%edi
   0x0000000000400f75 <+85>:    jmpq   0x4027d0 <puts>
End of assembler dump.
```

Activity 4 trace

- Working backwards from <+21> with $rdx = 0
- cmp $0x4, %edx
  - ja will jump to <+51> if 4 > $edx. Let's try $edx = 0
- Want $edx = 0. Thus from <+3> want $eax = 0
- lea (%rdi,%rdi,2),%eax
  - Does $eax = $rdi + 2 * $rdi = 3 * $rdi
  - We want $edx = $eax = 0, so $rdi = 0
- Since the input $rdi = 0, let's run with 0.
- (gdb) run 0
  - What happens?

```
(gdb) disassemble compute
Dump of assembler code for function compute:
   0x0000000000400f20 <+0>:     lea    (%rdi,%rdi,2),%eax
   0x0000000000400f23 <+3>:     mov    %eax,%edx
   0x0000000000400f25 <+5>:     and    $0xf,%edx
   0x0000000000400f28 <+8>:     nopl   0x0(%rax,%rax,1)
   0x0000000000400f30 <+16>:    cmp    $0x4,%edx
   0x0000000000400f33 <+19>:    ja     0x400f53 <compute+51>
   0x0000000000400f35 <+21>:    jmpq   *0x494298(,%rdx,8)
   0x0000000000400f3c <+28>:    nopl   0x0(%rax)
   0x0000000000400f40 <+32>:    and    $0x1,%eax
   0x0000000000400f43 <+35>:    mov    %eax,%edx
   0x0000000000400f45 <+37>:    jmp    0x400f30 <compute+16>
   0x0000000000400f47 <+39>:    nopw   0x0(%rax,%rax,1)
   0x0000000000400f50 <+48>:    sar    $0x2,%eax
   0x0000000000400f53 <+51>:    mov    %eax,%edx
   0x0000000000400f55 <+53>:    and    $0xf,%edx
   0x0000000000400f58 <+56>:    test   %eax,%eax
   0x0000000000400f5a <+58>:    jns    0x400f30 <compute+16>
   0x0000000000400f5c <+60>:    repz retq
   0x0000000000400f5e <+62>:    xchg   %ax,%ax
   0x0000000000400f60 <+64>:    add    %eax,%eax
   0x0000000000400f62 <+66>:    jmp    0x400f53 <compute+51>
   0x0000000000400f64 <+68>:    nopl   0x0(%rax)
   0x0000000000400f68 <+72>:    sub    $0x1,%eax
   0x0000000000400f6b <+75>:    jmp    0x400f53 <compute+51>
   0x0000000000400f6d <+77>:    nopl   (%rax)
   0x0000000000400f70 <+80>:    mov    $0x494290,%edi
   0x0000000000400f75 <+85>:    jmpq   0x4027d0 <puts>
End of assembler dump.
```

Activity 4 trace

- Compare the code to the assembly. Does it do what you expected?
- What do the jump statements to <+16> and <+51> correspond to?
- Working backwards like this could be helpful in bomb lab.

```
(gdb) disassemble compute
Dump of assembler code for function compute:
   0x0000000000400f20 <+0>:     lea    (%rdi,%rdi,2),%eax
   0x0000000000400f23 <+3>:     mov    %eax,%edx
   0x0000000000400f25 <+5>:     and    $0xf,%edx
   0x0000000000400f28 <+8>:     nopl   0x0(%rax,%rax,1)
   0x0000000000400f30 <+16>:    cmp    $0x4,%edx
   0x0000000000400f33 <+19>:    ja     0x400f53 <compute+51>
   0x0000000000400f35 <+21>:    jmpq   *0x494298(,%rdx,8)
   0x0000000000400f3c <+28>:    nopl   0x0(%rax)
   0x0000000000400f40 <+32>:    and    $0x1,%eax
   0x0000000000400f43 <+35>:    mov    %eax,%edx
   0x0000000000400f45 <+37>:    jmp    0x400f30 <compute+16>
   0x0000000000400f47 <+39>:    nopw   0x0(%rax,%rax,1)
   0x0000000000400f50 <+48>:    sar    $0x2,%eax
   0x0000000000400f53 <+51>:    mov    %eax,%edx
   0x0000000000400f55 <+53>:    and    $0xf,%edx
   0x0000000000400f58 <+56>:    test   %eax,%eax
   0x0000000000400f5a <+58>:    jns    0x400f30 <compute+16>
   0x0000000000400f5c <+60>:    repz retq
   0x0000000000400f5e <+62>:    xchg   %ax,%ax
   0x0000000000400f60 <+64>:    add    %eax,%eax
   0x0000000000400f62 <+66>:    jmp    0x400f53 <compute+51>
   0x0000000000400f64 <+68>:    nopl   0x0(%rax)
   0x0000000000400f68 <+72>:    sub    $0x1,%eax
   0x0000000000400f6b <+75>:    jmp    0x400f53 <compute+51>
   0x0000000000400f6d <+77>:    nopl   (%rax)
   0x0000000000400f70 <+80>:    mov    $0x494290,%edi
   0x0000000000400f75 <+85>:    jmpq   0x4027d0 <puts>
End of assembler dump.
```