ANITA’S SUPER AWESOME RECITATION SLIDES
15/18-213: Introduction to Computer Systems Assembly and GDB, 3 Feb 2014
Anita Zhang
MANAGEMENT AND STUFF

- Bomb Lab due Tues, 11 Feb 2014, 11:59 PM
  - This is my favorite lab!
- Buf Lab out Tues, 11 Feb 2014, 11:59 PM
  - One week long lab
WHAT’S ON THE MENU TODAY?

- Help (again)
- Books (again)
- Motivation
- Registers & Assembly
- Bomb Lab Overview
- GDB
- More Bomb Lab
- Walkthrough
HELPING US, HELPING YOU?

- Email us: 15-213-staff@cs.cmu.edu
  - TAs + Professors ➔ More coverage, fast replies
- All projects on Autolab: autolab.cs.cmu.edu
- Office Hours: Sun-Thurs, 6:00 PM – 8:00 PM
  - Wean 5207
WHAT HAVE YOU READ?


WHY ARE WE DOING THIS AGAIN?
INSIGHT FOR THE INQUISITIVE

- Why are we not learning about the stack yet?
  - Because x86_64
- “Technology note”
  - x86(_64) only.. For now
What are Registers?

- **Register**
  - Some place in hardware that stores bits
  - It is NOT on the stack or in main memory

- **Important**
  - When moving data between registers and memory, only the DATA moves, not the register
**Registers and All Them Bits**

- Quad = 64 bits
- Doubleword = 32 bits
- Word = 16 bits
- Byte = 8 bits

These are all parts of the same register
WHAT WE’RE WORKING WITH

- x86_64 conventions on the next slide

Specials
- %eip – instruction pointer
  - Points to the NEXT instruction to execute
- %esp – stack pointer
  - Points to top of the stack
- %eax – holds the return value
  - Also general purpose

Conditional Flags
- Sit in a special register of its own
- Don’t really need to worry about it
**x86_64, LOTS OF REGISTERS!**

<table>
<thead>
<tr>
<th>64 bits wide</th>
<th>32 bits wide</th>
<th>16 bits wide</th>
<th>8 bits wide</th>
<th>8 bits wide</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
<td>%ax</td>
<td>%ah</td>
<td>%al</td>
<td>Return Value</td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%bx</td>
<td>%bh</td>
<td>%bl</td>
<td>Callee Save</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%cx</td>
<td>%ch</td>
<td>%cl</td>
<td>4th Argument</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%dx</td>
<td>%dh</td>
<td>%dl</td>
<td>3rd Argument</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%si</td>
<td></td>
<td>%sil</td>
<td>2nd Argument</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%di</td>
<td></td>
<td>%dil</td>
<td>1st Argument</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%bp</td>
<td></td>
<td>%bpl</td>
<td>Callee Save</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%sp</td>
<td></td>
<td>%spl</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
<td>%r8w</td>
<td></td>
<td>%r8b</td>
<td>5th Argument</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
<td>%r9w</td>
<td></td>
<td>%r9b</td>
<td>6th Argument</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
<td>%r10w</td>
<td></td>
<td>%r10b</td>
<td></td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
<td>%r11w</td>
<td></td>
<td>%r11b</td>
<td></td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
<td>%r12w</td>
<td></td>
<td>%r12b</td>
<td></td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
<td>%r13w</td>
<td></td>
<td>%r12b</td>
<td></td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
<td>%rw</td>
<td></td>
<td>%14b</td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
<td>%r15w</td>
<td></td>
<td>%15b</td>
<td></td>
</tr>
</tbody>
</table>
Some More Definitions

- Memory Addressing
  - How assemblers denote memory locations
    - Direct
    - Indirect
    - Relative
    - Absolute
    - ...
  - Many different syntactical ways to represent the same address
**Reasons Why Intel is Ridiculous and Awesome**

- Operations can take several forms:
  - Register-to-Register
  - Register-to-Memory / Memory-to-Register
  - Immediate-to-Register / Immediate-to-Memory
  - One address operations (push, pop)
x86(_64) Common Addressing

- Offset(Base, Index, Scale)
- $D(Rb, Ri, S) \rightarrow Mem[Rb + Ri*S + D]$
  - $D$ can be any signed integer
  - Scale is 1, 2, 4, 8 (assume 1 if omitted)
  - Assume 0 for base if omitted
Representing Addresses

- Using parenthesis
  - Most of the time parenthesis means dereference
    - This is still only x86(_64)

- Examples of parenthesis usage:
  - (%eax)
    - Contents of memory at address stored, %eax
  - (%ebx, %ecx)
    - Contents of memory stored at address, %ebx + %ecx
  - (%ebx, %ecx, 8)
    - Contents of memory stored at address, %ebx + 8*%ecx
  - 4(%ebx, %ecx, 8)
    - Contents of memory stored at address, %ebx + 8*%ecx + 4
Representing Addresses

- Using parenthesis
  - Sometimes parenthesis are used just for addressing
    - This is still only x86(_64)

- Example
  - `lea1 (%ebx, %ecx, 8), destination`
    - Take only the values ➡️ %ebx + 8*%ecx
    - Does not dereference, uses the calculated value directly
  - `jmpq *0x402660(,%rax,8)`
    - The * does the dereference

- Examples of not using parenthesis
  - `%eax`
    - Use the value in %eax!
  - `$0x213`
    - A constant value
**Review of Conditionals/Flags**

- Most operations will set conditional flags
  - Bit operations
  - Arithmetic
  - Comparisons...

- **Core idea**: For conditionals, look one instruction before it to see whether it is true or false
  - Will be explained
FLAGS WE (MIGHT) CARE ABOUT

- Carry (CF)
  - Arithmetic carry/ borrow

- Parity (PF)
  - Odd or even number of bits set

- Zero (ZF)
  - Result was zero

- Sign (SF)
  - Most significant bit was set

- Overflow (OF)
  - Result does not fit into the location
PREP FOR ALL THE CHEAT SHEETS

- Warning: The following slides contain lots of assembly instructions.
  - All from CS:APP (our textbook BTW)
  - We’re not going over every single one...
  - Use it as a reference for Bomb Lab

- Quick note on Intel vs. AT&T
  - This is AT&T syntax (also, Bomb Lab syntax)
    - Looks like: “src, dest”
  - Intel tends to follow “dest, src”
    - Check out their ISA sometime
# All the Cheat Sheets (Movement)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb</td>
<td>S, D Move byte</td>
</tr>
<tr>
<td>movw</td>
<td>S, D Move word</td>
</tr>
<tr>
<td>movl</td>
<td>S, D Move doubleword</td>
</tr>
<tr>
<td>movsbw</td>
<td>S, D Move byte to word (sign extended)</td>
</tr>
<tr>
<td>movsbl</td>
<td>S, D Move byte to doubleword (sign extended)</td>
</tr>
<tr>
<td>movswl</td>
<td>S, D Move word to doubleword (sign extended)</td>
</tr>
<tr>
<td>movzbw</td>
<td>S, D Move byte to word (zero extended)</td>
</tr>
<tr>
<td>movzbl</td>
<td>S, D Move byte to doubleword (zero extended)</td>
</tr>
<tr>
<td>movzwl</td>
<td>S, D Move word to doubleword (zero extended)</td>
</tr>
<tr>
<td>pushl</td>
<td>S Push double word (Mem[%esp] ← S; %esp = %esp 4)</td>
</tr>
<tr>
<td>popl</td>
<td>D Pop double word (D ← Mem[%esp]; %esp = %esp + 4)</td>
</tr>
</tbody>
</table>
# All the Cheat Sheets (Bit Ops)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>S, D</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEAL</td>
<td>S, D</td>
<td>D ← &amp;S (Load address of source into destination)</td>
</tr>
<tr>
<td>INC</td>
<td>D</td>
<td>D ← D + 1</td>
</tr>
<tr>
<td>DEC</td>
<td>D</td>
<td>D ← D – 1</td>
</tr>
<tr>
<td>NEG</td>
<td>D</td>
<td>D ← –D</td>
</tr>
<tr>
<td>NOT</td>
<td>D</td>
<td>D ← ~D</td>
</tr>
<tr>
<td>ADD</td>
<td>S, D</td>
<td>D ← D + S</td>
</tr>
<tr>
<td>SUB</td>
<td>S, D</td>
<td>D ← D – S</td>
</tr>
<tr>
<td>IMUL</td>
<td>S, D</td>
<td>D ← D * S</td>
</tr>
<tr>
<td>XOR</td>
<td>S, D</td>
<td>D ← D ^ S</td>
</tr>
<tr>
<td>OR</td>
<td>S, D</td>
<td>D ← D</td>
</tr>
<tr>
<td>AND</td>
<td>S, D</td>
<td>D ← D &amp; S</td>
</tr>
<tr>
<td>SAL</td>
<td>k, D</td>
<td>D ← D &lt;&lt; k</td>
</tr>
<tr>
<td>SHL</td>
<td>k, D</td>
<td>D ← D &lt;&lt; k</td>
</tr>
<tr>
<td>SAR</td>
<td>k, D</td>
<td>D ← D &gt;&gt; k (arithmetic shift)</td>
</tr>
<tr>
<td>SHR</td>
<td>k, D</td>
<td>D ← D &gt;&gt; k (logical shift)</td>
</tr>
</tbody>
</table>
### All the Cheat Sheets (Specials)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
</table>
| **imull**   | R[%edx]:R[%eax] ← S * R[%eax]  
Signed multiply of %eax by S  
Result stored in %edx:%eax |
| **mull**    | R[%edx]:R[%eax] ← S * R[%eax]  
Unsigned multiply of %eax by S  
Result stored in %edx:%eax |
| **cltd**    | R[%edx]:R[%eax] ← SignExtend(R[%eax])  
Sign extend %eax into %edx |
| **idivl**   | R[%edx] ← R[%edx]:R[%eax] mod S;  
R[%eax] ← R[%edx]:R[%eax] ÷ S  
Signed divide of %eax by S  
Quotient stored in %eax  
Remainder stored in %edx |
| **divl**    | R[%edx] ← R[%edx]:R[%eax] mod S;  
R[%eax] ← R[%edx]:R[%eax] ÷ S  
Unsigned divide of %eax by S  
Quotient stored in %eax  
Remainder stored in %edx |
# All the Cheat Sheets (Comparisons)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpb S2, S1</td>
<td>Compare byte S1 and S2, Sets conditional flags based on S1 – S2.</td>
</tr>
<tr>
<td>cmpw S2, S1</td>
<td>Compare word S1 and S2, Sets conditional flags based on S1 – S2.</td>
</tr>
<tr>
<td>cmpl S2, S1</td>
<td>Compare double word S1 and S2, Sets conditional flags based on S1 – S2.</td>
</tr>
<tr>
<td>testb S2, S1</td>
<td>Compare byte S1 and S2, Sets conditional flags based on S1 &amp; S2.</td>
</tr>
<tr>
<td>testw S2, S1</td>
<td>Compare word S1 and S2, Sets conditional flags based on S1 &amp; S2.</td>
</tr>
<tr>
<td>testl S2, S1</td>
<td>Compare double word S1 and S2, Sets conditional flags based on S1 &amp; S2.</td>
</tr>
</tbody>
</table>
## All the Cheat Sheets (Set)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sete</strong>/ <strong>setz</strong></td>
<td>D (D \leftarrow ZF) (&quot;set if equal to 0&quot;)</td>
</tr>
<tr>
<td><strong>setne</strong>/ <strong>setnz</strong></td>
<td>D (D \leftarrow \sim ZF) (set if not equal to 0)</td>
</tr>
<tr>
<td><strong>sets</strong></td>
<td>D (D \leftarrow SF) (set if negative)</td>
</tr>
<tr>
<td><strong>setns</strong></td>
<td>D (D \leftarrow \sim SF) (set if nonnegative)</td>
</tr>
<tr>
<td><strong>setg</strong>/ <strong>setnle</strong></td>
<td>D (D \leftarrow \sim (SF \land OF) \land \sim ZF) (set if greater (signed &gt;))</td>
</tr>
<tr>
<td><strong>setge</strong>/ <strong>setnl</strong></td>
<td>D (D \leftarrow \sim (SF \land OF)) (set if greater or equal (signed &gt;=))</td>
</tr>
<tr>
<td><strong>setl</strong>/ <strong>setnge</strong></td>
<td>D (D \leftarrow SF \land OF) (set if less than (signed &lt;))</td>
</tr>
<tr>
<td><strong>setle</strong>/ <strong>setng</strong></td>
<td>D (D \leftarrow (SF \land OF) \land ZF) (set if less than or equal (signed &lt;=))</td>
</tr>
<tr>
<td><strong>seta</strong>/ <strong>setnbe</strong></td>
<td>D (D \leftarrow \sim CF \land \sim ZF) (set if above (unsigned &gt;))</td>
</tr>
<tr>
<td><strong>setae</strong>/ <strong>setnb</strong></td>
<td>D (D \leftarrow \sim CF) (set if above or equal (unsigned &gt;=))</td>
</tr>
<tr>
<td><strong>setb</strong>/ <strong>setnae</strong></td>
<td>D (D \leftarrow CF) (set if below (unsigned &lt;))</td>
</tr>
<tr>
<td><strong>setbe</strong>/ <strong>setna</strong></td>
<td>D (D \leftarrow CF \land ZF) (set if below or equal (unsigned &lt;=))</td>
</tr>
</tbody>
</table>
# All the Cheat Sheets (Jump)

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>Label</td>
</tr>
<tr>
<td></td>
<td>Jump to label</td>
</tr>
<tr>
<td>jmp</td>
<td>*Operand</td>
</tr>
<tr>
<td></td>
<td>Jump to specified locations</td>
</tr>
<tr>
<td>je/ jz</td>
<td>Label</td>
</tr>
<tr>
<td></td>
<td>Jump if equal/ zero (ZF)</td>
</tr>
<tr>
<td>jne/ jnz</td>
<td>Label</td>
</tr>
<tr>
<td></td>
<td>Jump if not equal/ nonzero (~ZF)</td>
</tr>
<tr>
<td>js</td>
<td>Label</td>
</tr>
<tr>
<td></td>
<td>Jump if negative (SF)</td>
</tr>
<tr>
<td>jns</td>
<td>Label</td>
</tr>
<tr>
<td></td>
<td>Jump if nonnegative (~SF)</td>
</tr>
<tr>
<td>jg/ jnle</td>
<td>Label</td>
</tr>
<tr>
<td></td>
<td>Jump if greater (signed) (~(SF ^ OF) &amp; ~ZF)</td>
</tr>
<tr>
<td>jge/ jnl</td>
<td>Label</td>
</tr>
<tr>
<td></td>
<td>Jump if greater or equal (signed) (~(SF ^ OF))</td>
</tr>
<tr>
<td>jl/ jnge</td>
<td>Label</td>
</tr>
<tr>
<td></td>
<td>Jump if less (signed) (SF ^ OF)</td>
</tr>
<tr>
<td>jle/ jng</td>
<td>Label</td>
</tr>
<tr>
<td></td>
<td>Jump if less or equal (signed) ((SF ^ OF)</td>
</tr>
<tr>
<td>ja/ jnbe</td>
<td>Label</td>
</tr>
<tr>
<td></td>
<td>Jump if above (unsigned) (~CF &amp; ~ZF)</td>
</tr>
<tr>
<td>jae/ jnb</td>
<td>Label</td>
</tr>
<tr>
<td></td>
<td>Jump if above or equal (unsigned) (~CF)</td>
</tr>
<tr>
<td>jb/ jnae</td>
<td>Label</td>
</tr>
<tr>
<td></td>
<td>Jump if below (unsigned) (CF)</td>
</tr>
<tr>
<td>jbe/ jna</td>
<td>label</td>
</tr>
<tr>
<td></td>
<td>Jump if below or equal (unsigned) (CF</td>
</tr>
</tbody>
</table>
# All the Cheat Sheets (cmove)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmove/ cmovz</td>
<td>S, R ( S \leftarrow R ) if Equal/ zero (ZF)</td>
</tr>
<tr>
<td>cmovne/ cmovnz</td>
<td>S, R ( S \leftarrow R ) if Not equal/ not zero (~ZF)</td>
</tr>
<tr>
<td>cmovs</td>
<td>S, R ( S \leftarrow R ) if Negative (SF)</td>
</tr>
<tr>
<td>cmovns</td>
<td>S, R ( S \leftarrow R ) if Nonnegative (~SF)</td>
</tr>
<tr>
<td>cmovg/ cmovnle</td>
<td>S, R ( S \leftarrow R ) if Greater (signed &gt;) (~SF &amp; OF) &amp; ~ZF)</td>
</tr>
<tr>
<td>cmovgge/ cmovnl</td>
<td>S, R ( S \leftarrow R ) if Greater or equal (signed &gt;=) (~SF &amp; OF)</td>
</tr>
<tr>
<td>cmovl/ cmovnge</td>
<td>S, R ( S \leftarrow R ) if Less (signed &lt;) (SF &amp; OF)</td>
</tr>
<tr>
<td>cmovle/ cmovg</td>
<td>S, R ( S \leftarrow R ) if Less or equal (signed &lt;=) ((SF &amp; OF) | ZF)</td>
</tr>
<tr>
<td>cmova/ cmovnbe</td>
<td>S, R ( S \leftarrow R ) if Above (unsigned &gt;) (~CF &amp; ~ZF)</td>
</tr>
<tr>
<td>cmovvae/ cmovnb</td>
<td>S, R ( S \leftarrow R ) if Above or equal (unsigned &gt;=) (~CF)</td>
</tr>
<tr>
<td>cmovb/ cmovnae</td>
<td>S, R ( S \leftarrow R ) if Below (unsigned &lt;) (CF)</td>
</tr>
<tr>
<td>cmovbe/ cmovna</td>
<td>S, R ( S \leftarrow R ) if Below or equal (unsigned &lt;=) (CF &amp; SF)</td>
</tr>
</tbody>
</table>
# All the Cheat Sheets (Calling)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>call Label</td>
<td>Push return and jump to label</td>
</tr>
<tr>
<td>call *operand</td>
<td>Push return and jump to specified location</td>
</tr>
</tbody>
</table>
| leave       | Prepare stack for return. Set stack pointer to %ebp and pop top stack into %ebp. In assembly: 
\[
\text{mov} \ %ebp, \ %esp \\
\text{pop} \ %ebp
\] |
| ret         | Pop return address from stack and jump there |
JUMPS, IN DEPTH

- The test instruction is usually followed by jump if equal/ not equal
- For conditional jumps, it is usually the second argument greater/less than first argument

```
JUMP IN DEPTH

test %al,%al
jne 4011ed

if ((%al & %al) != 0)
jump to 4011ed

- The test instruction is usually followed by jump if equal/ not equal

cmpl $0x5,0x14(%rsp)
jg 4011d0

if (0x14(%rsp) > $0x5)
jump to 4011d0

- For conditional jumps, it is usually the second argument greater/less than first argument
```
JE, JNE, JLE, JGE, ETC

- Jump if equal == Jump if zero
  - If the previous result was 0, jump
- Jump if not equal == Jump if not zero
  - If the previous result was not 0, jump
- Don’t worry about the conditional flags
  - Just remember “if second argument greater/less than first argument”
Dr. Evil and Bomblab

- 6 stages, each asking for input
  - Wrong input → bomb explodes (lose 1/2 point)
    - Score rounds up, so first explosion is free
  - Each stage may have multiple answers

- You get:
  - Bomb executable
  - Partial source of Dr. Evil mocking you

- Speed up next phase traversal with a text file
  - Place answers on each line
  - Run with bomb as ./bomb <solution file>
HOW IT WORKS

“But how do I find the solutions if I don’t have C code to work from?”

- Read a lot of bomb disassembly
  - All of the phases are just loops and patterns
- GDB

If you’re not working on a shark machine, your bomb won’t work.

- Will get an “illegal host” error
WORKING THROUGH THIS thing

- Read the disassembly
  - `phase_1`, `phase_2`, `phase_3`...
  - `explode_bomb`
  - Possible to reason through solutions without using GDB

- GNU Debugger
  - Step through each instruction, examine registers..
  - Set up breakpoints
  - Make sure to run “kill” when you hit the `explode_bomb` breakpoint
    - You’re screwed once you hit here, so why not exit?
But I Don’t Know How to GDB??

- Here have a cheat sheet
  - [http://csapp.cs.cmu.edu/public/docs/gdbnotes-x86-64.pdf](http://csapp.cs.cmu.edu/public/docs/gdbnotes-x86-64.pdf)
  - Everything you need to use GDB to solve bomb lab
- The Internet has a great range of commands you might find useful
GDB’s Most Useful

- **run/ run <arguments>**
  - Runs the program up till the next breakpoint.

- **disassemble/ disas**
  - Shows the current function with an arrow to the next
  - **WARNING:** shortcut “disa” disables all breakpoints

- **step/ stepi/ nexti**
  - **stepl** steps to the next line of Assembly.
  - **nexti** does the same but doesn’t stop in function calls.
  - **stepl n** or **nexti n** steps through n lines.
GDB’s Most Useful

- **break <location>**
  - Sets breakpoint. Location can be function name or address.
  - Stop at an instruction address with `break *address`
  - You have to reset your break points when you restart GDB!

- **x <address/register>**
  - Dereference the address or value in the register and print the contents to the console
  - Give it a format to print out to, ie. “x/s” prints as string

- **p <address/register/variable>**
  - Print the contents of the register, or the variable, or the address to the console
  - Give it a format to print out to, ie. “p/s” prints as string
GDB’s Most Useful

- Saving breakpoints (in GDB)
  - `(gdb) save breakpoints file.txt`  
    - This saves all your breakpoints to `file.txt`
  - `(gdb) source file.txt`  
    - This restores breakpoints from `file.txt`
GETTING STARTED

- Download and untar ON A SHARK MACHINE
  - `tar xvf labhandout.tar`
- `shark> objdump -d bomb > filename`
  - Outputs the whole bomb assembly code to a filename
- `shark> objdump -t bomb > filename`
  - Contains locations of globals, variables, etc
- `shark> strings bomb > filename`
  - All printable strings used in your bomb
- `shark> gdb bomb`
  - Prepares to run the bomb in gdb
Speed up the Wait

- When you have solutions, put it into a text file
  - Separate each solution with a newline
  - Your bomb will auto-advance completed phases with pre-filled solutions
- Then when you run gdb next time:
  - `(gdb)> run solution_file`
**Bomb Lab Specifics**

- `int sscanf (const char *s, const char *format, ...);`
  - `s`
    - Source string to retrieve data from
  - `format`
    - Formatting string used to get values from the source string
  - `...`
    - Depending the format string, one location (address) per formatter used to hold values extracted from source string
**SSCANF Example**

```c
#include <stdio.h>

int main () {
    char sentence[]="Rudolph is 12 years old";
    char str[20];
    int i;
    sscanf (sentence,"%s %*s %d", str, &i);
    printf ("%s -> %d\n", str, i);
    return 0;
}
```

- Outputs: Rudolph -> 12
Relevance to Bomb Lab

- Why do we care about sscanf?
  - Mainly used to read in arguments
  - Keep track of which locations the read in values will be stored
    - Important for knowing where arguments will be stored
    - And how they will be used
    - They will usually be store in memory/ on the stack
Jump tables
- In memory, you can think of it as an “array” of locations to jump to
- Using assembly it is possible to index into the “array”
- Each entry of the array will hold addresses of instructions
The tip-off is something like this:

- `jmpq *0x400600(,%rax,8)`
  - Empty base means implied 0
  - `%rax` is the “index”
  - 8 is the “scale”
    - In a jump tables, 64-bit machine addresses are 8 bytes
  - `*` indicates a dereference (as in regular C)
    - Like `leal`: does not do a dereference even with parenthesis

- Put it all together: “Jump to the address stored in the address 0x400600 + `%rax*8`”

Using GDB (example output): `x/8g 0x400600`

```
0x400600: 0x000000000004004d1 0x000000000004004c8
0x400610: 0x000000000004004c8 0x000000000004004be
0x400620: 0x000000000004004d7 0x000000000004004be
0x400630: 0x000000000004004c8 0x000000000004004be
```
Demo Time
CREDITS & QUESTIONS

- StackOverflow on Assembly Projects
- P. 274 of CS:APP – x86_64 Registers
- P. 171 - 221 of CS:APP – Assembly Instructions
- CPlusPlus Reference on sscanf