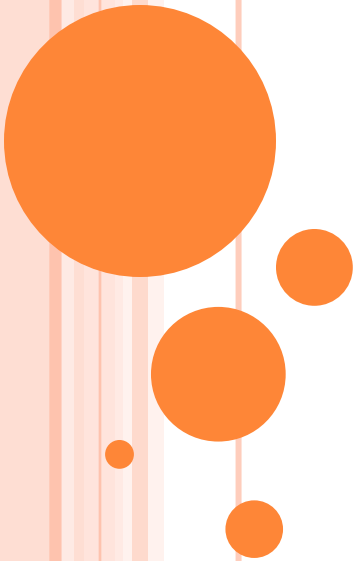


Anita's Super Awesome Recitation Slides

15/18-213: Introduction to Computer Systems
Assembly and GDB, 4 Jun 2013

Anita Zhang, Section M

Management and Stuff

- Bomb Lab due Tues, 11 Jun 2013, 11:59 pm EST
  - Apparently for distance students it's 2 days after
  - This is my favorite lab!
- Buf Lab out Tues, 11 Jun 2013, 11:59 pm EST
  - Due the week after
- FAQ on the main site
  - Has some stuff
  - Answers to "Permission denied" errors, etc

What's on the Menu Today?

- Help (again)
- Books (again)
- Motivation
- Registers
- Assembly
- Bomb Lab Overview
- GDB
- Walkthrough
- More Bomb Lab

Helping Us, Helping You?

- Email us: 15-213-staff@cs.cmu.edu
  - Please attach C files if you have a specific question
  - Responses within 2 minutes (record!)
- IRC: irc.freenode.net, ##213
  - Anita polls it every 3 hours
- Videos on Blackboard
- Everything else, Autolab: autolab.cs.cmu.edu
- Office hours: Sun-Thurs, 6pm – 9pm, Gates 5205
  - Both Michael and Anita will be there (mostly)
  - We leave at 7:30pm if no one shows up

What Have You Read?

- Randal E. Bryant and David R. O'Hallaron, *Computer Systems: A Programmer's Perspective, Second Edition*, Prentice Hall, 2011
- Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language, Second Edition*, Prentice Hall, 1988
- Koenig, Andrew. *C Traps and Pitfalls*. Reading, MA: Addison-Wesley, 1988
- Kernighan, Brian W., and Rob Pike. *The Practice of Programming*. Reading, MA: Addison-Wesley, 1999

Why Are We Doing This Again?

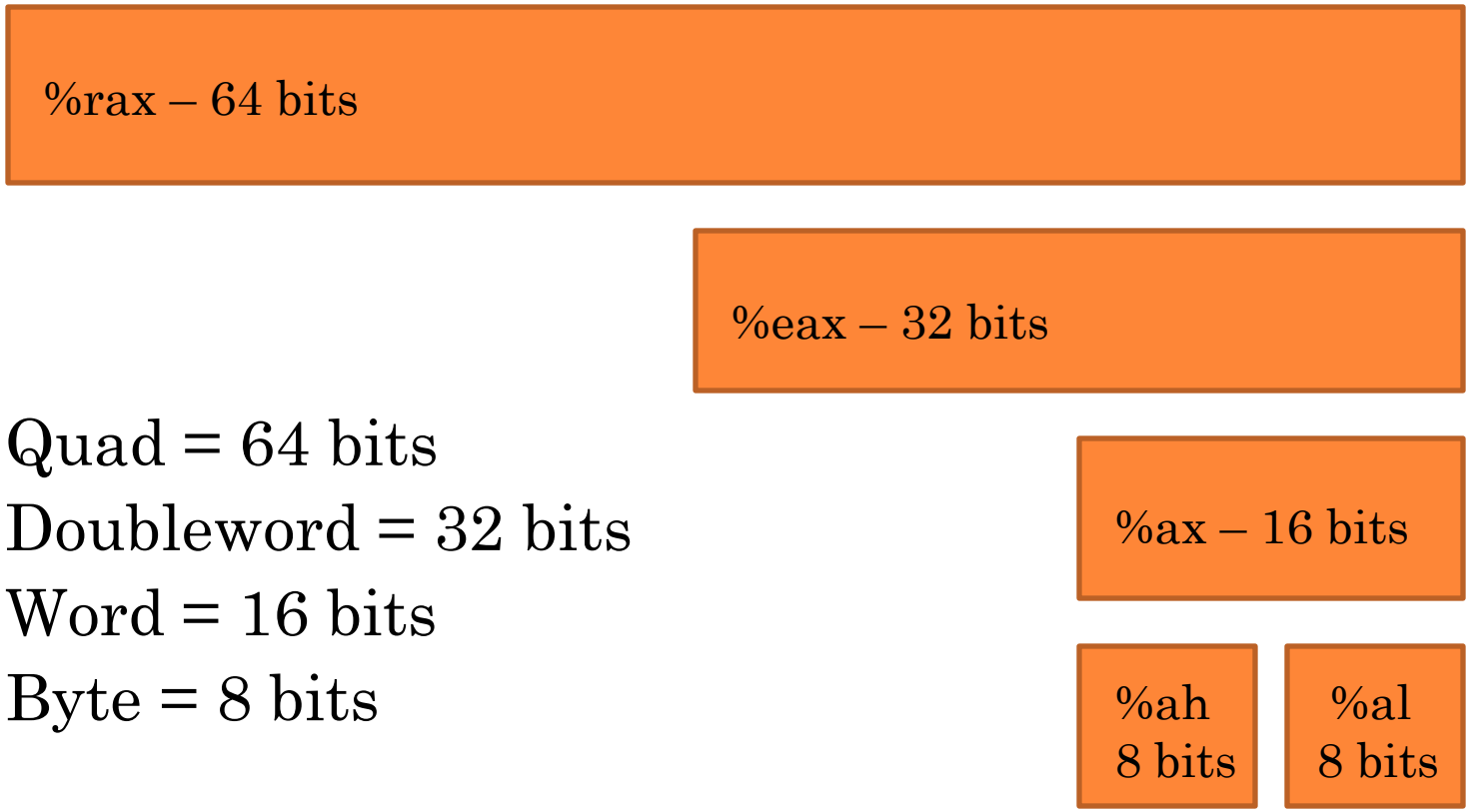
Definitions and Conventions

- Register
  - Some place in hardware that stores bits
- Caller save
  - Saved by the caller of a function
  - Before a function call, the caller must save any caller save register values it wants preserved
- Callee save
  - Saved by the callee of a function
  - The callee is required to save and restore the values in these registers if it is using them in the function

Insight for the Inquisitive

- Why are we not learning about the stack yet?
  - Because x86_64
- "Technology note"
  - x86(_64) only

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

- General Purpose (x86)
  - Caller Save: %eax, %ecx, %edx
  - Callee Save: %ebx, %esi, %edi, %ebp, %esp
  - x86_64 conventions on the next slide
- Specials
  - %eip – instruction pointer
  - %ebp – frame pointer
  - %esp – stack pointer
- Conditional Flags
  - Sit in a special register of its own
  - You only need to worry about the ones mentioned later

x86_64, LOTS of Registers!

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

Some More Definitions

- Memory Addressing
  - How assemblers denote memory locations
    - Direct
    - Indirect
    - Relative
    - Absolute
    - …
  - Syntax differs, addresses do not

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)
  - Did I miss any?

Reasons Why Intel is Ridiculous and Awesome

- x86(_64) Addressing (some kind of indirect)
  - Offset(Base, Index, Scale)
  - D(Rb, Ri, S) → 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

Reasons Why Intel is Ridiculous and Awesome

- 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

Reasons Why Intel is Ridiculous and Awesome

- Using parenthesis
  - Sometimes parenthesis are used just for addressing
    - This is still only x86(_64)
- Example
  - `leal (%ebx, %ecx, 8), destination`
    - Take the address, %ebx + 8*%ecx
    - Does not dereference, uses the calculated value directly
- 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 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)

Instruction		Effect
movb	S, D	Move byte
movw	S, D	Move word
movl	S, D	Move doubleword
movsbw	S, D	Move byte to word (sign extended)
movsbl	S, D	Move byte to doubleword (sign extended)
movswl	S, D	Move word to doubleword (sign extended)
movzbw	S, D	Move byte to word (zero extended)
movzbl	S, D	Move byte to doubleword (zero extended)
movzwl	S, D	Move word to doubleword (zero extended)
pushl	S	Push double word
popl	D	Pop double word

All the cheat Sheets (Bit Ops)

Instruction		Effect
LEAL	S, D	D ← &S (Load effective address of source into destination)
INC	D	D ← D + 1
DEC	D	D ← D – 1
NEG	D	D ← –D
NOT	D	D ← ~D
ADD	S, D	D ← D + S
SUB	S, D	D ← D – S
IMUL	S, D	D ← D * S
XOR	S, D	D ← D ^ S
OR	S, D	D ← D | S
AND	S, D	D ← D & S
SAL	k, D	D ← D << k
SHL	k, D	D ← D << k
SAR	k, D	D ← D >> k (arithmetic shift)
SHR	k, D	D ← D >> k (logical shift)

All the cheat Sheets (Specials)

Instruction		Effect
imull	S	R[%edx]:R[%eax] ← S * R[%eax] Signed full multiply of %eax by S Result stored in %edx:%eax
mull	S	R[%edx]:R[%eax] ← S * R[%eax] Unsigned full multiply of %eax by S Result stored in %edx:%eax
cltd		R[%edx]:R[%eax] ← SignExtend(R[%eax]) Sign extend %eax into %edx
idivl	S	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	S	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)

Instruction		Effect
cmpb	S2, S1	Compare byte S1 and S2, Sets conditional flags based on S1 – S2.
cmpw	S2, S1	Compare word S1 and S2, Sets conditional flags based on S1 – S2.
cmpl	S2, S1	Compare double word S1 and S2, Sets conditional flags based on S1 – S2.
testb	S2, S1	Compare byte S1 and S2, Sets conditional flags based on S1 & S2.
testw	S2, S1	Compare word S1 and S2, Sets conditional flags based on S1 & S2.
testl	S2, S1	Compare double word S1 and S2, Sets conditional flags based on S1 & S2.

All the cheat Sheets (Set)

Instruction		Effect
sete/ setz	D	D ← ZF (“set if equal to 0”)
setne/ setnz	D	D ← ~ZF (set if not equal to 0)
sets	D	D ← SF (set if negative)
setns	D	D ← ~SF (set if nonnegative)
setg/ setnle	D	D ← ~(SF ^ OF) & ~ZF (set if greater (signed >))
setge/ setnl	D	D ← ~(SF ^ OF) (set if greater or equal (signed >=))
setl/ setnge	D	D ← SF ^ OF (set if less than (signed <))
setle/ setng	D	D ← (SF ^ OF) | ZF (set if less than or equal (signed <=))
seta/ setnbe	D	D ← ~CF & ~ZF (set if above (unsigned >))
setae/ setnb	D	D ← ~CF (set if above or equal (unsigned >=))
setb/ setnae	D	D ← CF (set if below (unsigned <))
setbe/ setna	D	D ← CF | ZF (set if below or equal (unsigned <=))

All the cheat Sheets (Jump)

Instructions		Effect
jmp	Label	Jump to label
jmp	*Operand	Jump to specified locations
je/ jz	Label	Jump if equal/ zero (ZF)
jne/ jnz	Label	Jump if not equal/ nonzero (~ZF)
js	Label	Jump if negative (SF)
jns	Label	Jump if nonnegative (~SF)
jg/ jnle	Label	Jump if greater (signed) (~(SF ^ OF) & ~ZF)
jge/ jnl	Label	Jump if greater or equal (signed) (~(SF ^ OF))
jl/ jnge	Label	Jump if less (signed) (SF ^ OF)
jle/ jng	Label	Jump if less or equal (signed) ((SF ^ OF) | ZF)
ja/ jnbe	Label	Jump if above (unsigned) (~CF & ~ZF)
jae/ jnb	Label	Jump if above or equal (unsigned) (~CF)
jb/ jnae	Label	Jump if below (unsigned) (CF)
jbe/ jna	label	Jump if below or equal (unsigned) (CF | ZF)

All the cheat Sheets (cmove)

Instruction		Effect
cmove/ cmovz	S, R	S ← R if Equal/ zero (ZF)
cmovne/ cmovnz	S, R	S ← R if Not equal/ not zero (~ZF)
cmovs	S, R	S ← R if Negative (SF)
cmovns	S, R	S ← R if Nonnegative (~SF)
cmovg/ cmovnle	S, R	S ← R if Greater (signed >) (~(SF ^ OF) & ~ZF)
cmovge/ cmovnl	S, R	S ← R if Greater or equal (signed >=) (~(SF ^ OF))
cmovl/ cmovnge	S, R	S ← R if Less (signed <) (SF ^ OF)
cmovle/ cmovg	S, R	S ← R if Less or equal (signed <=) ((SF ^ OF) | ZF)
cmova/ cmovnbe	S, R	S ← R if Above (unsigned >) (~CF & ~ZF)
cmovae/ cmovnb	S, R	S ← R if Above or equal (unsigned >=) (~CF)
cmovb/ cmovnae	S, R	S ← R if Below (unsigned <) (CF)
cmovbe/ cmovna	S, R	S ← R if Below or equal (unsigned <=) (CF | SF)

All the cheat Sheets (Calling)

Instruction		Effect
call	Label	Push return and jump to label
call	*operand	Push return and jump to specified location
leave		Prepare stack for return. Set stack pointer to %ebp and pop top stack into %ebp. In assembly (AT&T syntax of source, destination): *mov %ebp, %esp* *pop %ebp*
ret		Pop return address from stack and jump there

Dr. Evil and Bomblab

- 6 stages, each asking for input
  - Wrong input → bomb explodes (lose 1/2 point)
  - 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
    - Or just dead simple (see the demo)
  - **GDB**
- If you're not working on a shark machine, your bomb won't work.
  - Will get "illegal host"

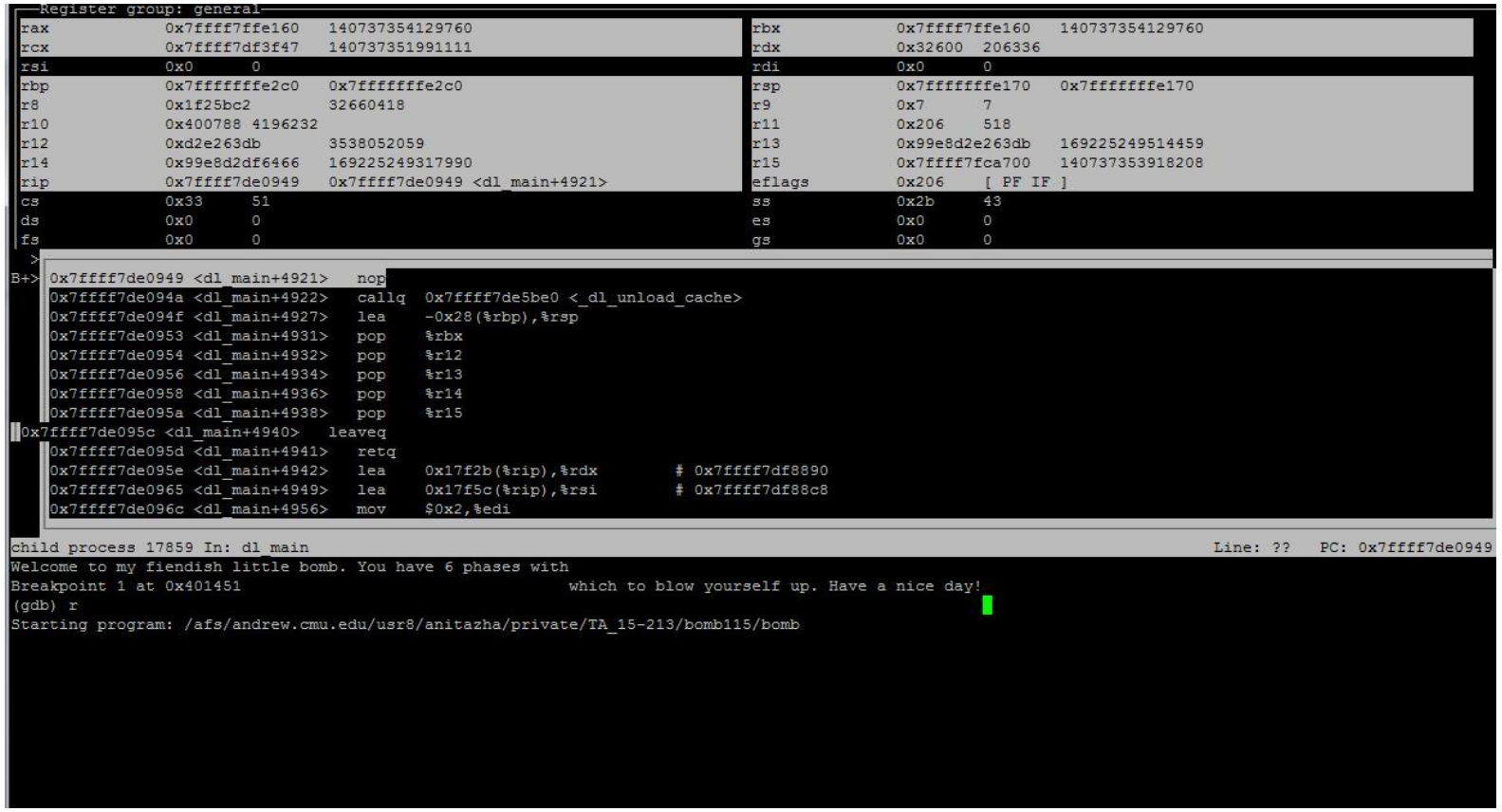
Working Through This thing

- Read the disassembly
  - phase_1, phase_2, phase_3….
  - explode_bomb
  - Understand what's going on
- GNU Debugger
  - Step through each instruction, examine registers..
  - Set up breakpoints
  - Make sure to type "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
  - Everything you need to use GDB to solve bomblab

Fancy GDB


Register group: general
rax 0x7ffff7ffe160 140737354129760
rbx 0x7ffff7ffe160 140737354129760
rcx 0x7ffff7df3f47 140737351991111
rdx 0x32600 206336
rsi 0x0 0
rdi 0x0 0
rbp 0x7fffffffe2c0 0x7fffffffe2c0
rsp 0x7fffffffe170 0x7fffffffe170
r8 0x1f25bc2 32660418
r9 0x7 7
r10 0x400788 4196232
r11 0x206 518
r12 0xd2e263db 3538052059
r13 0x99e8d2e263db 169225249514459
r14 0x99e8d2df6466 169225249317990
r15 0x7ffff7fca700 140737353918208
rip 0x7ffff7de0949 0x7ffff7de0949 <dl_main+4921>
eflags 0x206 [PF IF]
cs 0x33 51
ss 0x2b 43
ds 0x0 0
es 0x0 0
fs 0x0 0
gs 0x0 0
B+> 0x7ffff7de0949 <dl_main+4921> nop
0x7ffff7de094a <dl_main+4922> callq 0x7ffff7de5be0 <_dl_unload_cache>
0x7ffff7de094f <dl_main+4927> lea -0x28(%rbp),%rsp
0x7ffff7de0953 <dl_main+4931> pop %rbx
0x7ffff7de0954 <dl_main+4932> pop %r12
0x7ffff7de0956 <dl_main+4934> pop %r13
0x7ffff7de0958 <dl_main+4936> pop %r14
0x7ffff7de095a <dl_main+4938> pop %r15
0x7ffff7de095c <dl_main+4940> leaveq
0x7ffff7de095d <dl_main+4941> retq
0x7ffff7de095e <dl_main+4942> lea 0x17f2b(%rip),%rdx # 0x7ffff7df8890
0x7ffff7de0965 <dl_main+4949> lea 0x17f5c(%rip),%rsi # 0x7ffff7df88c8
0x7ffff7de096c <dl_main+4956> mov $0x2,%edi
child process 17859 In: dl_main
Line: ?? PC: 0x7ffff7de0949
Welcome to my fiendish little bomb. You have 6 phases with
Breakpoint 1 at 0x401451 which to blow yourself up. Have a nice day!
(gdb) r
Starting program: /afs/andrew.cmu.edu/usr8/anitazha/private/TA_15-213/bomb115/bomb

Fancy GDB Commands

- Layout commands split GDB into cool windows
  - May/ may not lag a lot.
  - Has a tendency to not work properly sometimes
- *layout asm*
  - Splits GDB into assembly and GDB command
- *layout src*
  - Splits GDB into C source and GDB command
- *layout regs*
  - Splits GDB into register window with either source or assembly, and GDB command
- Arrow, page up/down to traverse layout windows
- **ctrl+x a to switch back to normal GDB**

Getting Started

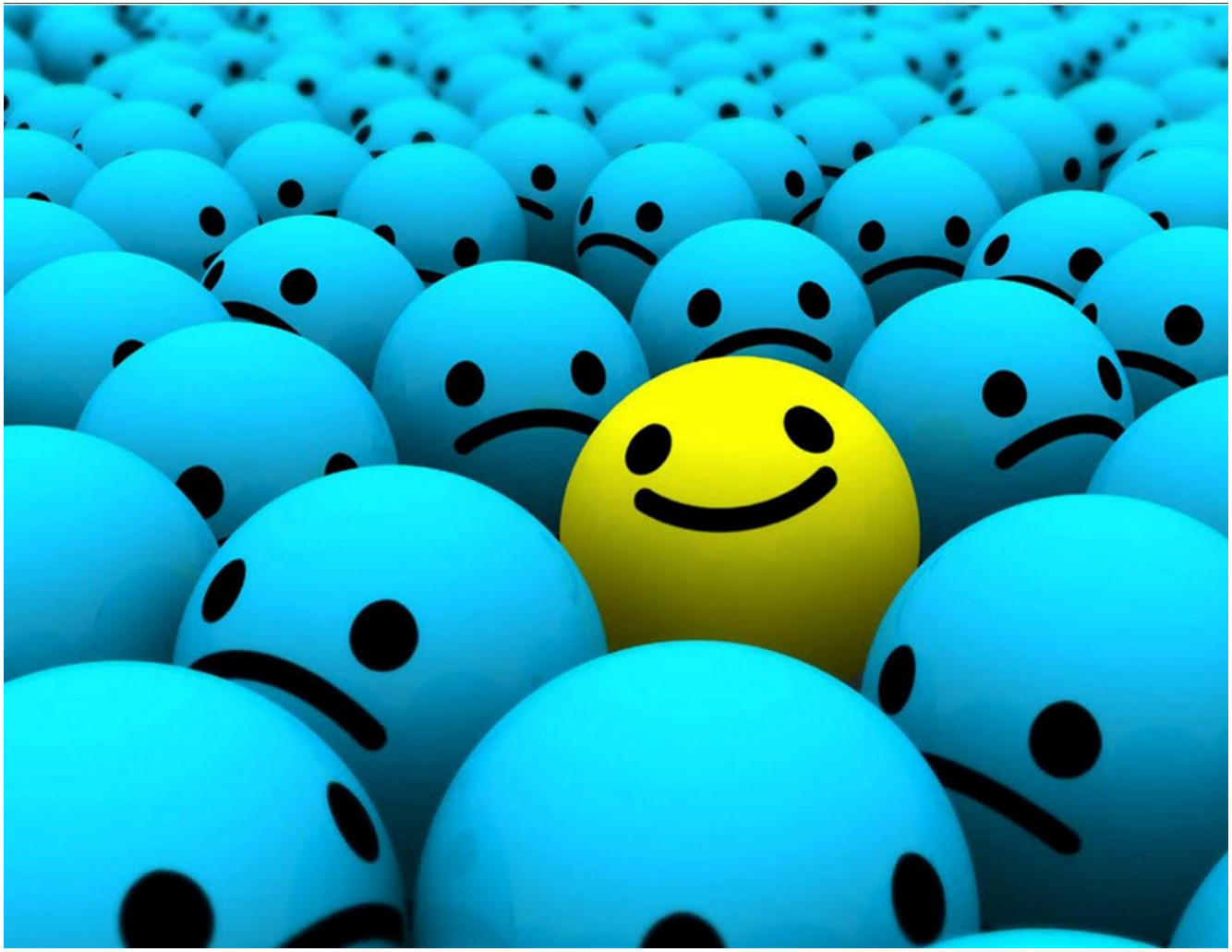
- Download and untar ON A SHARK MACHINE

- shark> objdump –d bomb > *disassembly filename*
- shark> objdump –t bomb > *symbol table filename*
- shark> strings bomb > *strings filename*
- shark> gdb bomb

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*

Demo Time



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

```
#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?
  - Mostly used to read in arguments
  - Note of which locations read in values will be stored
    - Important for knowing where arguments will be stored
    - And how they will be used

More Bomb Lab Specifics

- Jump tables
  - In memory is an “array” of locations
  - In assembly it is possible to index into this “array”
  - Each entry of the array will potentially hold addresses to the next instruction to go to

Jump Tables

- 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 table with addresses, 64-bit machines addresses are 8 bytes
    - * indicates a dereference (as in regular C)
      - Like `leal`; does not do a dereference just 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: 0x00000000004004d1 0x00000000004004c8
0x400610: 0x00000000004004c8 0x00000000004004be
0x400620: 0x00000000004004c1 0x00000000004004d7
0x400630: 0x00000000004004c8 0x00000000004004be
```

Credits & Questions

- http://stackoverflow.com/questions/757398/what-are-some-ways-you-can-manage-large-scale-assembly-language-projects
- P. 274 of CS:APP – x86_64 Registers
- P. 171 - 221 of CS:APP – Assembly Instructions
- http://www.cplusplus.com/reference/cstdio/sscanf/