

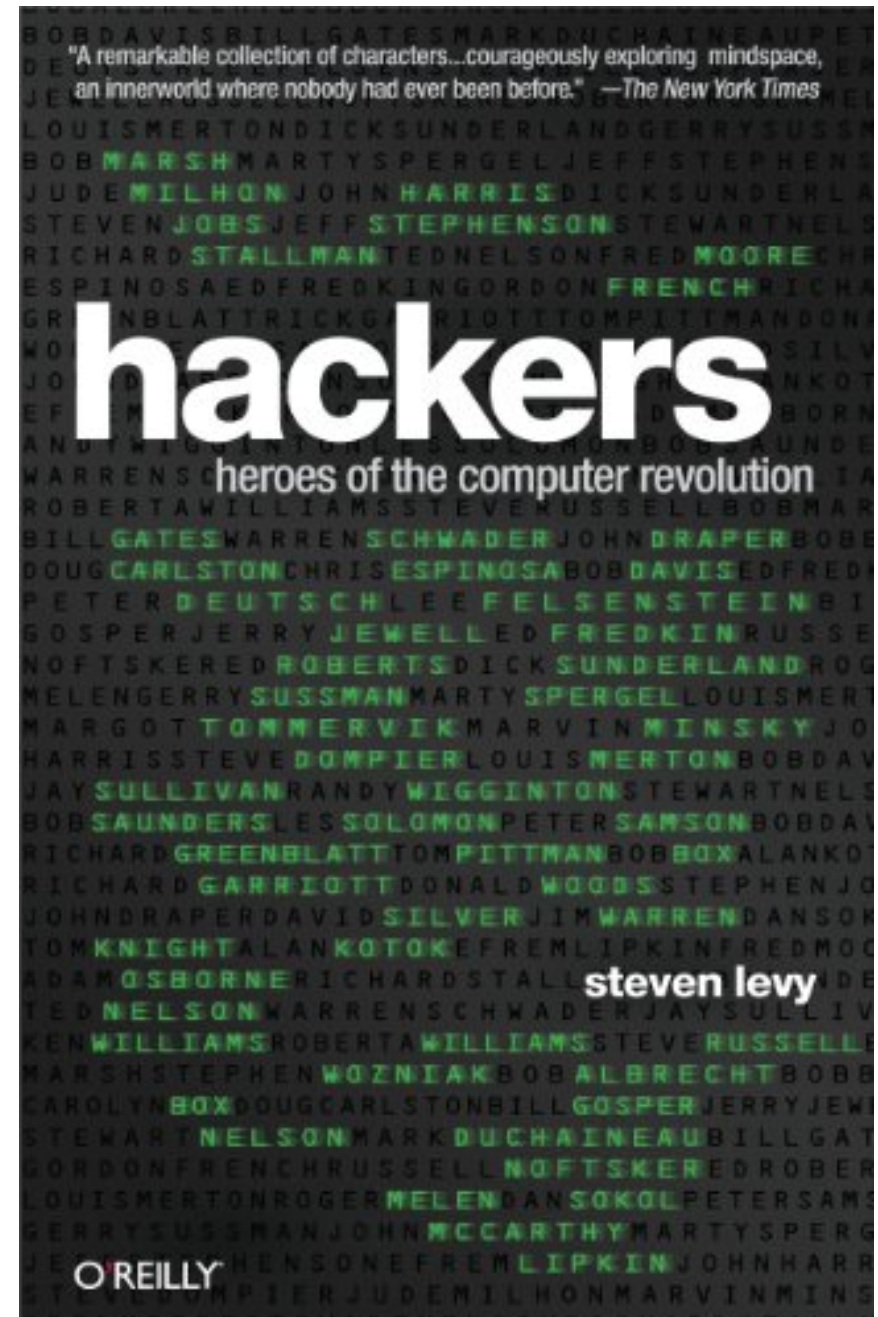
Assembly & BombLab Introduction

15-213 (18-213): Introduction to Computer Systems
Recitation 4, Feb 4 2013

Alexey Tumanov (atumanov)
Section C, Monday 11:30 – 12:20, WeH 5302

Outline

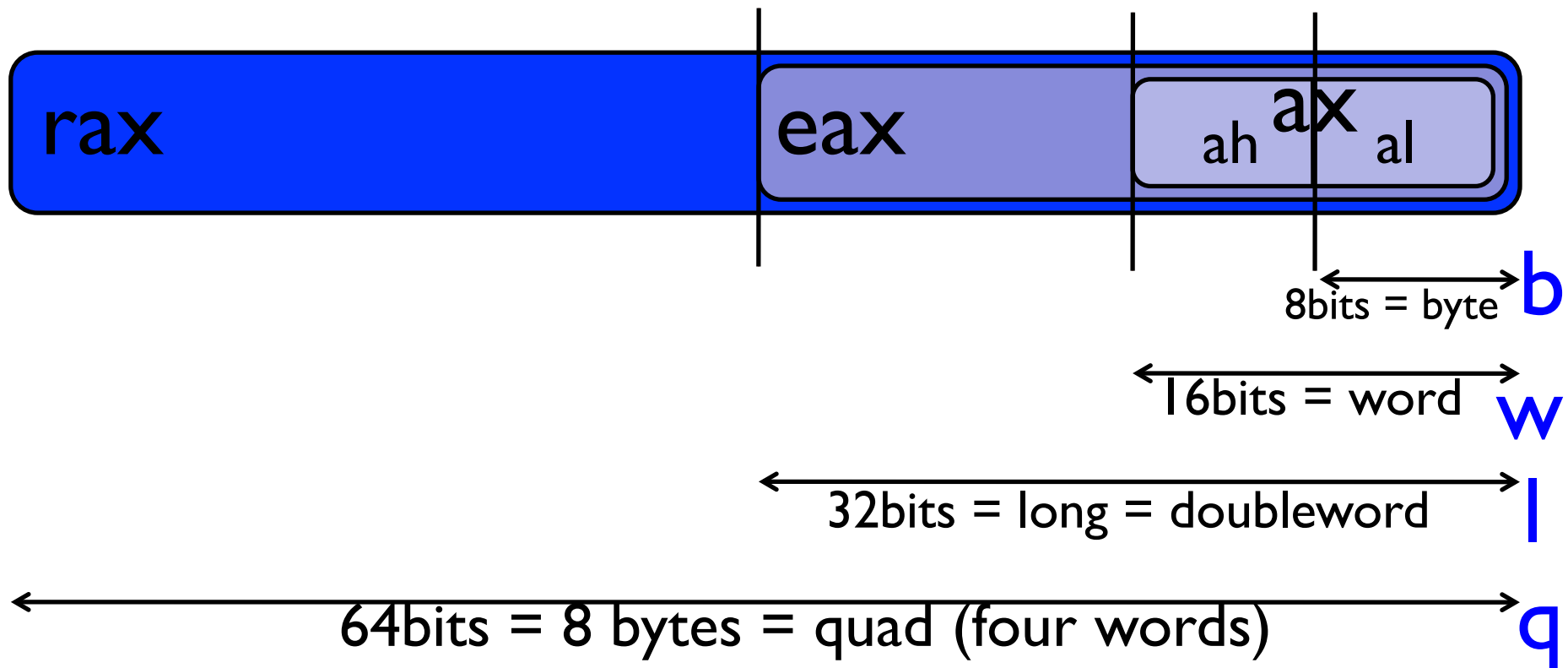
- Introduction to Assembly
 - register overview
 - assembly instructions
 - assembly instruction operands
 - managing control flow
- Introduction to Bomblab
 - lab overview
 - auxiliary tools
 - GNU debugger (gdb)



Administrativia

- FAQ: <http://www.cs.cmu.edu/~213/faq.html>
 - E.g. Why do I get “permission denied” error?
 - Read through before you ask questions
 - Check back for any future updates
- Style guide: <http://www.cs.cmu.edu/~213/codeStyle.html>
 - consistent indentation
 - reasonable variable names
 - document important parts of the code
 - fit within 80 character line width
 - ~/.vimrc or ~/.emacs
- TA Feedback form: <https://www.ugrad.cs.cmu.edu/ta/feedback>
 - open until Feb 17

Registers (32 & 64 bit)



Registers

- General purpose registers:
 - eax (accumulator), ebx (base), ecx (counter), edx
 - esi, edi (used for string manipulation by convention)
 - rax, rbx, rcx, rdx, rsi, rdi, r8 - r15, rbp (x86_64)
- Special purpose registers:
 - EIP – instruction pointer register
 - EBP – branch pointer register
 - ESP – stack pointer register
 - eflags (carry, parity, zero, sign, overflow)
- x86 → x86_64
 - ebp/rbp is no longer special
 - 8 new registers
 - 64 bit registers, subsuming 32 bit registers
 - full-width instructions : L → Q (e.g., movl → movq)

Assembly Instructions: Arithmetic + Logic

- Binary operators: $OP\ SRC, DST \Leftrightarrow DST = OP(DST, SRC)$
 - addl, subl, imull, sarl, sall, shrl, shll, xorl, andl, orl
 - operand order matters:
 - e.g., SARL %eax, %ebx $\rightarrow$ ebx = ebx >> eax

- Unary operators: $OP\ DST \Leftrightarrow DST = OP(DST)$
 - incl, decl, negl, notl

Assembly Operands

■ Immediate

- `$0xff` , `$5` , `$-1`
- decimal, octal, or hexadecimal (as in C)
- size: byte, word, long, or quad

■ Register

- `%rax`, `%r14`
- e.g., `XORL %ebx, %ebx` (Q: what does this do?)
- e.g., `MOVL %ecx, %eax` : `eax` $\leftarrow$ `ecx`

■ Memory

- `D(Rb, Ri, S)`
- `8(%ebx)`, `12(%ebx, %ecx, 4)`
- e.g., `MOVL 8(%ebx), %eax` : move 1st arg from stack to `%eax`

Special Operations

- CLTQ (*may* see it in bomblab):
 - convert long to quad : `%rax = SignExtend(%eax)`
 - `eax = 0x05` → `rax = 0x05` ; `eax = 0x80000000` → `rax = 0xffffffff 80000000`
- LEAL: load effective address
 - does not dereference memory
 - used as an arithmetic operator to perform memory address calculations
 - commonly used for calculating offsets into an array in a loop
 - example: `LEAL (%eax, %eax, 4):` `eax = eax * 5`
- MOVZBL
 - move byte to long with zero fill

Memory Addressing Modes

- Most general form: Offset(Base, Index, Scale)
 - $D(Rb, Ri, S) \rightarrow \text{Mem}[Rb + Ri * S + D]$
 - Offset (D) can be any signed integer
 - Scale is 1,2,4,8 (for byte, word, long, quad strides)
 - if omitted, assume 1
 - Base: if omitted, assume 0x0
- Special forms:
 - $(\%eax) \rightarrow \text{contents of memory at address stored in } \%eax$
 - $(\%ebx, \%ecx) \rightarrow \text{Mem}[\text{Reg}[\%ebx] + \text{Reg}[\%ecx]]$
 - $(\%ebx, \%ecx, 8) \rightarrow \text{Mem}[\text{Reg}[\%ebx] + 8 * \text{Reg}[\%ecx]]$

Memory Addressing: Example

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Operand	Value
%eax	
0x104	
\$0x108	
4(%eax)	
9(%eax,%edx)	
0xfc(, %ecx, 4)	
(%eax, %edx, 4)	

Memory Addressing: Example

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

note the difference

Operand	Value	Why?
%eax	0x100	register contents
0x104	0xAB	absolute addr
\$0x108	0x108	immediate operand
4(%eax)	0xAB	contents of 0x104
9(%eax,%edx)	0x11	contents of 0x10c
0xfc(, %ecx, 4)	0xff	contents of 0xfc+4
(%eax, %edx, 4)	0x11	contents of 0x10c

Control: Compare, Flags, and Jumps

- Explicit flag setting instructions:
 - `cmpl b, a` → execute `a - b` and set **eflags**
 - `testl b, a` → execute `a&b` and set **eflags**
- Flags: OF, SF, ZF, CF
 - **OF**: overflow (`a<0 && b>0 && diff > 0`) || (`a>0 && b<0 && diff<0`)
 - **SF**: sign bit set (`a-b < 0`)
 - **ZF**: `a==b`
 - **CF**: previous operation resulted in a carry out from MSB
- Conditional jumps:
 - `je/jne` (ZF), `js/jns` (SF),
 - `jg/jge/jl/jle` (SF, OF, ZF): **if** `a >, >=, <, <= b` **then** jump
 - e.g., `JLE`: jumps iff `(SF^OF) | ZF` (why?)
 - `cmpl $5, $6 ; JLE .Lexit` (Q: will this jump to `.Lexit`?)

Control Flow Example

```
■ void cond(int a, int *p) {  
    if (p && a > 0)  
        *p += a;  
}
```

<pre>MOVL 8(%ebp), %edx MOVL 12(%ebp), %eax TESTL %eax, %eax JE .LEXIT TESTL %edx, %edx JLE .LEXIT ADDL %edx, (%eax) .LEXIT</pre>	
---	--

Control Flow Example: Test & JLE

OF, SF, ZF not set => JLE won't jump

Register group: general

eax	0xffffd394	-11372	ecx	0x14e66e84	-1528402300
edx	0x5	5	ebx	0x2c2114	2895860
esp	0xffffd378	0xffffd378	ebp	0xffffd378	0xffffd378
esi	0x0	0	edi	0x0	0
eip	0x80483a3	0x80483a3 <cond+15>	eflags	0x206	[PF IF]
cs	0x23	35	ss	0x2b	43
ds	0x2b	43	es	0x2b	43
fs	0x0	0	gs	0x63	99

```

0x804839d <cond+9>  test    %eax,%eax
0x804839f <cond+11>  je      0x80483a7 <cond+10>
0x80483a1 <cond+13>  test    %edx,%edx
0x80483a3 <cond+15>  jle     0x80483a7 <cond+19>
0x80483a5 <cond+17>  add     %edx,%eax
0x80483a7 <cond+19>  pop     %ebp
0x80483a8 <cond+20>  ret
0x80483a9 <main>     push    %ebp
0x80483aa <main+1>    mov     %esp,%ebp
0x80483ac <main+3>    sub     $0x18,%esp
0x80483af <main+6>    movl    $0x0,-0x4(%ebp)
0x80483b6 <main+13>   lea     -0x4(%ebp),%eax
0x80483b9 <main+16>   mov     %eax,0x4(%esp)
0x80483bd <main+20>   movl    $0x5,(%esp)
0x80483c4 <main+27>   call   0x8048394 <cond>
  
```

child process 7527 In: cond (gdb) Line: 5 PC: 0x80483a3

Control Flow Example

```

■ void cond(int a, int *p) {
    if (p && a > 0)
        *p += a;
}

```

```

    MOVL    8(%ebp), %edx
    MOVL    12(%ebp), %eax
    TESTL   %eax, %eax
    JE      .LEXIT
    TESTL   %edx, %edx
    JLE     .LEXIT
    ADDL    %edx, (%eax)
    .LEXIT

```

```

void goto_cond(int a, int *p)
{
    if (p == 0) goto LEXIT;
    if (a <= 0) goto LEXIT;
    *p += a;
LEXIT:
    return;
}

```

jX	Condition	Description
JE	ZF	Equal/Zero
JLE	(SF^OF) ZF	Less or Equal/Signed

Bomblab Overview

- Series of stages asking for an input string
 - input string can be anything: e.g., an array of chars or a number
 - `./bomb < mysolutions` # one input string per line
- An incorrect input string causes the bomb to “explode”
 - ½ point deducted for each explosion – they cannot be “undone”
 - it is possible to try multiple incorrect answers without explosion (carefully)
- What’s provided: `./bomb` , `bomb.c` (partial source)
 - full C source is NOT provided
- What to do:
 - figure out the expected input string for each phase
- This must be done on shark machines
- Exercise great care
 - e.g., don’t just enter garbage and see what happens

Toolz: GDB

- `gdb <binary>`
- Ultra useful gdb cheatsheet
 - <http://csapp.cs.cmu.edu/public/docs/gdbnotes-ia32.txt>
- Useful commands:
 - advance EIP: **s** , **n** , **si** , **ni** , **c** , **run**
 - print things out:
 - `print <expression>` : e.g., `print variable_name`
 - `x [/format] <address>` : print contents of memory
 - how much to print and how to print is dictated by format
 - disassemble: **disas** address
 - layout :
 - `layout split` , `layout regs`, `layout src`, `layout asm`
 - `focus next (fs n)`

Toolz: GDB (continued)

- querying state:
 - info break
 - info frame (highly useful summary of the current activation record)
 - info program
 - info registers
- working with breakpoints:
 - break <address> ; break <function_name>
 - enable <breakpoint_number>
 - disable <breakpoint_number>
 - delete <breakpoint_number>
- start/stop execution:
 - run [command line args]: e.g., **run** *partial_answer_file*
 - quit (and confirm, if quitting a running executable)

GNU Debugger

Register group: general

rax	0x603da0 6307232	rbx	0x7fffffff338 140737488347960
rcx	0x1	rdx	0x6 6
rsi	0x7ffff7f92007 140737353687047	rdi	0x603da0 6307232
rbp	0x0 0x0	rsp	0x7fffffff248 0x7fffffff248
r8	0x7ffff7dd8e10 140737351880208	r9	0x7ffff7fca700 140737353918208
r10	0x7ffff7fdd90 140737488347024	r11	0x7ffff7a76110 140737348329744
r12	0x400c10 4197392	r13	0x7fffffff330 140737488347952
r14	0x0 0	r15	0x0
rip	0x40128b 0x40128b <phase_1>	eflags	0x202
cs	0x33 51	ss	0x2b 43
ds	0x0 0	es	0x0 0
fs	0x0 0	gs	0x0 0

Turning C into Object Code

```

> 0x40128b <phase_1> sub $0x8,%rsp
0x40128f <phase_1+4> mov $0x402708,%esi
0x401294 <phase_1+9> callq 0x4012cf <strings_not_equal>P
0x401299 <phase_1+14> test %eax,%eax
0x40129b <phase_1+16> je 0x4012a2 <phase_1+23>
0x40129d <phase_1+18> callq 0x4014a1 <explode_bomb>
0x4012a2 <phase_1+23> add $0x8,%rsp
0x4012a6 <phase_1+27> retq program(p1.c p2.c)
0x4012a7 nop
0x4012a8 nop
0x4012a9 nop
0x4012aa nop
0x4012ab text nop
0x4012ac nop
0x4012ad nop
  
```

Compiler (gcc -S)

Assembler (gcc or as)

Linker (gcc or ld)

Child process 12402 In: phase_1

0x000000000040128b in phase_1 ()
(gdb)

Line: ?? PC: 0x40128b

makeAllDistances.sh
memsrv.22.11
memsrv.log.tgz

GNU Debugger

Register group: general

rax	0x603da0 6307232	rbx	0x7fffffff338 140737488347960
rcx	0x1	rdx	0x6 6
rsi	0x7ffff7f92007 140737353687047	rdi	0x603da0 6307232
rbp	0x0 0x0	rsp	0x7fffffff248 0x7fffffff248
r8	0x7ffff7dd8e10 140737351880208	r9	0x7ffff7fca700 140737353918208
r10	0x7ffff7fdd90 140737488347024	r11	0x7ffff7a76110 140737348329744
r12	0x400c10 4197392	r13	0x7fffffff330 140737488347952
r14	0x0 0	r15	0x0
rip	0x40128b 0x40128b <phase_1>	eflags	0x202
cs	0x33 51	ss	0x2b 43
ds	0x0 0	es	0x0 0
fs	0x0 0	gs	0x0 0

regs

Turning C into Object Code

```

> 0x40128b <phase_1> sub $0x8,%rsp
0x40128f <phase_1+4> mov $0x402708,%esi
0x401294 <phase_1+9> callq 0x4012cf <strings_not_equal>
0x401299 <phase_1+14> test %eax,%eax
0x40129b <phase_1+16> je 0x4012a2 <phase_1+23>
0x40129d <phase_1+18> callq 0x4014a1 <explode_bomb>
0x4012a2 <phase_1+23> add $0x8,%rsp
0x4012a6 <phase_1+27> retq
0x4012a7 nop
0x4012a8 nop
0x4012a9 nop
0x4012aa nop
0x4012ab nop
0x4012ac nop
0x4012ad nop
  
```

asm

Child process 12402 In: phase_1

0x000000000040128b in phase_1 ()
(gdb)

command

GNU Debugger

```

Register group: general
rax 0x603da0 6307232
rcx 0x1 1
rsi 0x7ffff7f92007 140737353687047
rbp 0x0 0
r8 0x7ffff7dd8e10 140737351880208
r10 0x7ffff7fd90 140737488347024
r12 0x400c10 4197392
rip 0x40128b 0x40128b <phase_1>
ds 0x0 0
fs 0x0 0
rbx 0x7fffffff338 140737488347960
rdx 0x6 6
rdi 0x603da0 6307232
rsp 0x7ffff7fca700 0x7ffff7fca700
r9 0x7ffff7fca700 140737353918208
r11 0x7ffff7a76110 140737348329744
r13 0x7ffff7f330 140737488347952
r15 0x0
eflags 0x202
ss 0x0
es 0x0
gs 0x0

Turning C into Object Code

> 0x40128b <phase_1> sub $0x8,%rsp
0x40128f <phase_1+4> mov $0x102700,%esi
0x401294 <phase_1+9> callq 0x4012cf <strings_not_equal>P
0x401299 <phase_1+14> test %eax,%eax
0x40129b <phase_1+16> je 0x4012a2 <phase_1+23>
0x40129d <phase_1+18> callq 0x4014a1 <explode_bomb>
0x4012a2 <phase_1+23> add $0x8,%rsp
0x4012a6 <phase_1+27> retq
0x4012a7 nop
0x4012a8 nop
0x4012a9 nop
0x4012aa nop
0x4012ab nop
0x4012ac nop
0x4012ad nop

Child process 12402 In: phase_1
0x000000000040128b in phase_1 ()
(gdb)

```

Toolz: objdump, strings

■ `objdump -d <binary>`

- disassembles the <binary>, dumps assembly output
- redirect stdout to an output file: `objdump -d ./bomb > bomb.S`
- redirect stdout to a pipe : `objdump -d ./bomb | less`
- `objdump -M intel` # if you want intel syntax
 - hint: stick with default

■ `strings <binary>`

- dumps collections of printable characters from <binary>
- may include:
 - function names, string literals
 - password hashes? <grin>

Bomb Walkthrough

- How to get started with defusing the bomb
- How to blow up the bomb