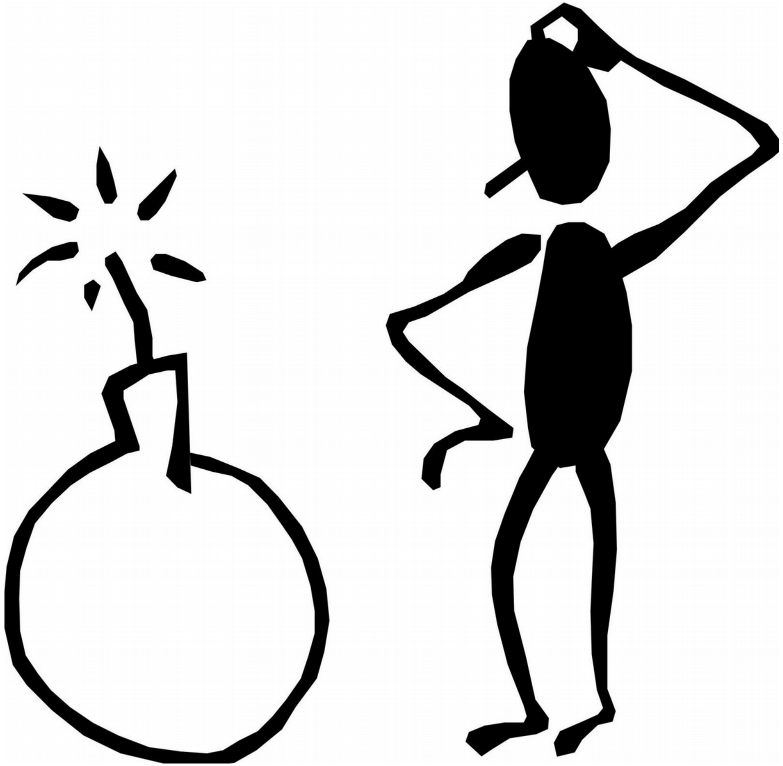
15-213 Recitation 4: Bomb Lab

2 Feb 2016

Monil Shah, Shelton D'Souza, Ralf Brown

Agenda

- Bomb Lab Overview
- Assembly Refresher
- Introduction to GDB
- Bomb Lab Demo



Downloading Your Bomb

- **Please read the writeup.** ***Please read the writeup.*** ***<u>Please read the writeup.</u>***
- Your bomb is **unique** to you. Dr. Evil has created one ~~million~~ billion bombs, and can distribute as many new ones as he pleases.
  - if you download a second bomb, it will be different from the first!
- Bombs have six phases which get progressively ~~harder~~ more fun to use.
- Bombs can only run on the shark clusters. They **will** blow up if you attempt to run them locally.

Exploding Your Bomb

- Blowing up your bomb notifies Autolab.
  - Dr. Evil takes **0.5** of your points each time the bomb explodes.
- Inputting the correct string moves you to the next phase.
- Jumping between phases detonates the bomb – you have to solve them in the given order.

```
jbiggs@makoshark ~/school/ta-15-213-f14/bomb170 $ ls
bomb  bomb.c  README
jbiggs@makoshark ~/school/ta-15-213-f14/bomb170 $ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Who does Number Two work for!?

BOOM!!!
The bomb has blown up.
Your instructor has been notified.
jbiggs@makoshark ~/school/ta-15-213-f14/bomb170 $ 
```

Examining Your Bomb

- You get:
  - An executable
  - A README file
  - A heavily redacted source file
- The source file just makes fun of you
  - The executable is **not** compiled from the source you get
- Outsmart Dr. Evil by examining the executable

x64 Assembly: General-Purpose Registers

Return	%rax	%eax
	%rbx	%ebx
Arg 4	%rcx	%ecx
Arg 3	%rdx	%edx
Arg 2	%rsi	%esi
Arg 1	%rdi	%edi
Stack ptr	%rsp	%esp
	%rbp	%ebp

%r8	%r8d	Arg 5
%r9	%r9d	Arg 6
%r10	%r10d	
%r11	%r11d	
%r12	%r12d	
%r13	%r13d	
%r14	%r14d	
%r15	%r15d	

x64 Assembly: Special Registers

- The x86 ISA contains *many* more registers, but most are useful only to operating systems
- %RIP is the **instruction pointer**, which is the address of the next instruction to execute
  - you can only indirectly access this register, using control-flow instructions
- %EFLAGS contains various CPU control and status bits
  - the interesting ones to us are the **condition codes**
- %XMMS0 through %XMMS7 are 128-bit wide registers used for SIMD instructions (not part of Bomb Lab)
  - hold two `doubles`, four `floats`, four `ints`, eight `shorts`, or 16 `chars`
  - SIMD instructions operate on all elements simultaneously

x64 Assembly: Condition Codes

- Arithmetic instructions set the condition codes according to the result of the operation
  - ZF (zero flag): the result was zero
  - SF (sign flag): the sign bit of the result is set
  - CF (carry flag): the operation caused a carry out of the highest bit (unsigned overflow/underflow)
  - OF (overflow flag): the operation caused signed overflow/underflow
- Examples:
  - add 0x7F + 1 => 0x80: ZF=0, SF=1, CF=0, OF=1
  - add 0xFF + 1 => 0x00: ZF=1, SF=0, CF=1, OF=0
- Used by conditional jumps, instructions like ADC and the SETcc family

x64 Assembly: Operands

Type	Syntax	Example	Notes
Constants	Start with $	`$-42` `$0x15213b`	Don't mix up decimal and hex
Registers	Start with %	`%esi` `%rax`	Can store values or addresses

Memory Locations	Parentheses around a register or an addressing mode	`(%rbx)`
		`0x1c(%rax)`
		`0x4(%rcx, %rdi, 0x1)`
			Parentheses dereference. Look up addressing modes!

x64 Assembly: Addressing Modes

- %rax
- (%rax)
- 0x18(%rax)
- (%rax,%rbx)
- (%rax,%rbx,4)
- 0x40(%rax,%rbx,8)
- General form:
  - disp(basereg,idxreg,scale)
  - scale = 1, 2, 4, or 8
  - disp is signed value up to 32 bits

- Value of %rax
- Content of Mem[%rax]
- Content of Mem[%rax+0x18]
- Mem[%rax+%rbx]
- Mem[%rax+4*%rbx]
- Mem[%rax+8*%rbx+0x40]
- content of:

  Mem[basereg+scale*idxreg+disp]

x64 Assembly: Instructions

- Each instruction has a mnemonic name, e.g. **sub** (subtract), **cmp** (compare)
- Each instruction has zero, one, two, or (in rare cases) three operands
  - some instructions have *implicit* operands, e.g. the LOOP instruction uses %RCX and CMPS uses both %RSI and %RDI
  - conditional jumps use one or more condition codes as implied by their mnemonic
- At most one operand may be a memory location
- Linux assembly appends a letter specifying the operation size (b = byte, q = 64 bits, etc.) to the mnemonic if the operands don't imply the size
  - e.g. if the only operand is a memory location

x64 Assembly: Arithmetic and Movement Operations

Instruction	Effect
`add (%rdx), %r8`	`r8 += value of memory[rdx]`
`mul $3, %r8`	`r8 *= 3`
`sub $1, %r8`	`r8--`

- *math ops set condition codes*

Move with **S**ign-extension, from **L**ong to **Q**uad

Instruction	Effect
`mov %rbx, %rdx`	`rdx = rbx`
`movslq %ebx, %rdx`	`rdx = ebx sign-extended`

- *moves **don't** change condition codes*

Instruction	Effect
`lea (%rax,%rbx,2), %rdx`	`rdx = rax + rbx * 2`

- *doesn't dereference or set cond codes*

Load **E**ffective **A**ddress

x64 Assembly: More Arithmetic/Logical Operations

Instruction	Effect
`shl $2, %r8`	
`shr %cl, %r8`	
`sar $3, %r8`	
`inc %r8`	
`neg %r8`	
`imul %rbx, %rdx`	
`and $0x7f, %rdx`	
`or $1, %r8`	
`xor %rax, %rdx`	
`not %rdx`	

x64 Assembly: More Arithmetic/Logical Operations

Instruction	Effect
`shl $2, %r8`	`r8 <<= 2`
`shr %cl, %r8`	`r8 >>= cl, zero-filled`
`sar $3, %r8`	`r8 >>= 3, sign bit copied`
`inc %r8`	`r8++`
`neg %r8`	`r8 = -r8`
`imul %rbx, %rdx`	`rdx *= rbx`
`and $0x7f, %rdx`	`rdx &= 0x7f`
`or $1, %r8`	`r8 |= 0x01`
`xor %rax, %rdx`	`rdx ^= rax`
`not %rdx`	`rdx = ~rdx`

x64 Assembly: Comparisons

- Comparison instructions **cmp** and **test** are used to set condition codes
  - **cmp b,a** computes `a-b` and discards the result
  - **test b,a** computes `a&b` and discards the result
- Pay attention to **operand order** for **cmp**
  - comparison seems backwards because Linux assembly uses the opposite operand order that Intel used when it defined the mnemonics

```
cmpl %r9, %r10
jg 8675309
```

⇔

If `%r10 > %r9`, then jump to `8675309`

x64 Assembly: Jumps

Instruction	Effect	Instruction	Effect
`jmp`	Always jump	`ja`	Jump if above (unsigned >)
`je/jz`	Jump if eq / zero	`jae`	Jump if above / equal
`jne/jnz`	Jump if !eq / !zero	`jb`	Jump if below (unsigned <)
`jg`	Jump if greater	`jbe`	Jump if below / equal
`jge`	Jump if greater / eq	`js`	Jump if sign bit is 1 (neg)
`jl`	Jump if less	`jns`	Jump if sign bit is 0 (pos)
`jle`	Jump if less / eq	`jo`	Jump if signed overflow

x64 Assembly: A Quick Drill

```
cmp $0x15213, %r12
jge 0xdeadbeef
```

If ______, jump to addr `0xdeadbeef`

```
cmp %rax, %rdi
jae 0x15213b
```

If ______, jump to addr `0x15213b`

```
test %r8, %r8
jnz 0x15213
jmp *(%rsi)
```

If ______, jump to _____, otherwise _________.

x64 Assembly: A Quick Drill

```
cmp $0x15213, %r12
jge 0xdeadbeef
```

If **%r12 >= 0x15213**, jump to addr `0xdeadbeef`

```
cmp %rax, %rdi
jae 0x15213b

test %r8, %r8
jnz 0x15213
jmp *(%rsi)
```

x64 Assembly: A Quick Drill

```
cmp $0x15213, %r12
jge 0xdeadbeef
```

`cmp %rax, %rdi`
`jae 0x15213b`

```
test %r8, %r8
jnz 0x15213
jmp *(%rsi)
```

If the *unsigned value* of %rdi is at or above the *unsigned value* of %rax, jump to `0x15213b`

x64 Assembly: A Quick Drill

```
cmp $0x15213, %r12
jge 0xdeadbeef

cmp %rax, %rdi
jae 0x15213b

test %r8, %r8
jnz 0x15213
jmp *(%rsi)
```

If **%r8 & %r8** is nonzero, jump to 0x15213, otherwise **jump to the address stored in memory location %rsi**.

x64 Assembly: A Quick Drill

```
cmp $0x15213, %r12
jge 0xdeadbeef

cmp %rax, %rdi
jae 0x15213b
```

```
test %r8, %r8
jnz 0x15213
jmp *(%rsi)
```

Only unconditional jumps can use registers or memory locations to specify the target

If **%r8 & %r8** is nonzero, jump to 0x15213, otherwise **jump to the address stored in memory location %rsi**.

x64 Assembly: Subroutines

`call 0x15213` `...more code...`	Push address of instruction following the call on the stack, then jump to `0x15213`
`[at addr 0x15213:]` `push %r12`	Push callee-saved registers on stack.
`...subroutine body...`	Perform subroutine
`pop %r12`	Restore registers from stack
`ret`	Pop return address and jump there

x64 Assembly: Jump Tables

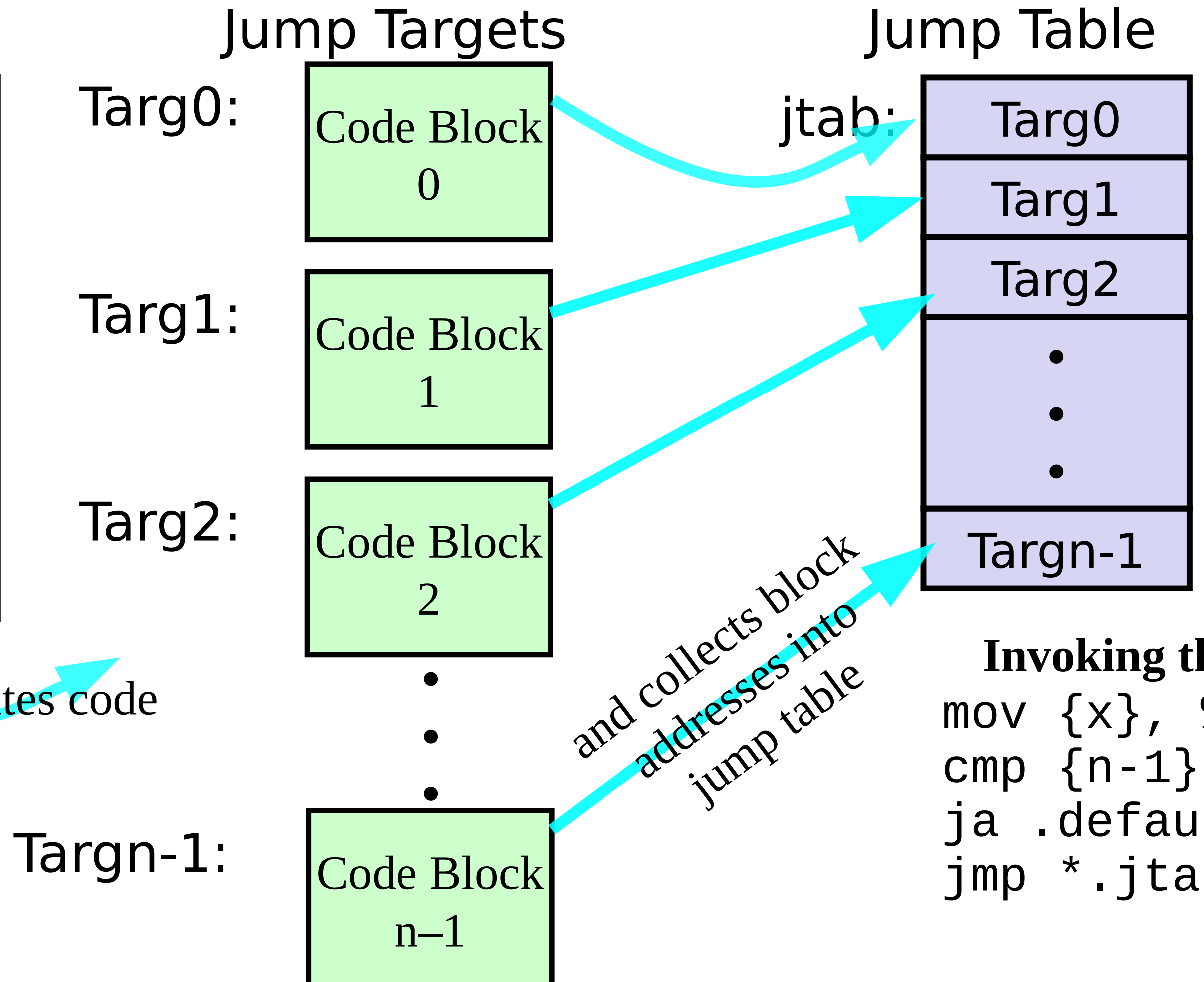
- C `switch` statements are commonly implemented using jump tables
- The address of each `case` statement is stored in an array, and the value of the `switch` condition is used as an index
  - e.g. `jmp *.jumptable(,%r9,8)`
- You can also have an array of function addresses, and invoke the appropriate function for a given context
  - in C: (func_ptr_array[which])(...arguments...)
  - asm equivalent: set up args, then `call (%rsi,%rax,8)`
- C++ uses a similar approach, called a Virtual Method Table, to permit function inheritance among object classes
  - asm code looks like `call $0x18(%rbp)`

x64 Assembly: Jump Tables (2)

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Compiler generates code



Invoking the switch():

```
mov {x}, %r8
cmp {n-1}, %r8
ja .default
jmp *.jtab(,%r8,8)
```

x64 Assembly: The SETcc Instructions

- These instructions set the low byte of their destination to 0x00 or 0x01 depending on whether the specified combination of condition codes holds
- The mnemonics for the conditions are the same as for the conditional branch instructions

Opcode	Condition	Description	Opcode	Condition	Description
sete	ZF==1	equal / zero	setge	~(SF^OF)	signed >=
setne	ZF==0	not equal / nonzero	setl	(SF^OF)	signed <
sets	SF==1	negative	setle	(SF^OF)|ZF	signed <=
setns	SF==0	nonnegative	seta	~CF&~ZF	above (unsigned)
setg	~(SF^OF)&~ZF	signed greater	setb	CF	below (unsigned)

(Ab)using LEA for Arithmetic

- LEA is not just for getting the address of an item in memory
- It lets you compute anything of the form

  R3 = R1 + S * R2 + constant

  (where S = 1, 2, 4, or 8)
- It takes fewer instructions and less time than the equivalent `mov` and `add` sequence, and doesn't clobber the condition codes

```
mov r2, r3
shl $k, r3
add r1, r3
add $constant, r3
```

- GCC's optimizer likes it: `x = y + z` can be turned into a LEA instruction

What's This 'repz retq' ?

- You'll have noticed that some functions end with

  `rep; ret` or `repz retq`

  instead of just plain

  `ret`
- The `rep[z]` prefix only affects certain instructions; in this context, it is a no-op
- This is a gcc speed optimization that works around a limitation in branch prediction in early x86-64 processors
- For all the gory details, visit http://repzret.org/p/repzret/ (but that's beyond the scope of 15-213)

Defusing Your Bomb

- `objdump -t bomb` examines the symbol table
- `objdump -d bomb` disassembles all bomb code
- `strings bomb` prints all printable strings
- `gdb bomb` will open the **G**NU **Deb**ugger
  - step through your program and examine
    - registers
    - the stack
    - contents of program memory
    - instruction stream (and disassemble individual functions)

Using gdb: Overview

- Commands can be abbreviated to the shortest unique prefix
  - some very common commands have one- or two- letter alternate forms
  - prefixes are underlined on the following slides
- If you forget the details of a command, use the command

  help {command}
- Just pressing <enter> (i.e. an empty command) will repeat the previous command – this is handy for stepping through a program one instruction at a time
- Many commands re-use their previous argument if entered without one

Using gdb: Stopping Execution

- break <location>
  - stop execution at function name or address
    - use format *0xNNNN to break at an address
  - breakpoints must be set each time you start gdb, but not each time you restart the program you are debugging
- info breakpoints / i b
  - show current breakpoints
- disable <number>
  - temporarily turn off the breakpoint numbered <number>
- delete <number>
  - permanently forget breakpoint <number>

Using gdb: Running/Stepping

- run [<args>]
  - run program with command-line arguments <args>
  - re-uses previous <args> if none given
  - program runs until breakpoint, termination, or crash
- continue
  - resume running after hitting a breakpoint
- stepi / si [<count>]
  - execute exactly <count> (default one) instructions – re-uses last count!
- nexti / ni [<count>]
  - like stepi, but treats function calls as a single instruction

Using gdb: Examining Code

- disassemble <fun> (**not** dis, which is disable)
- bt / backtrace / where [<count>]
  - show the call stack - “how did I get here?”
  - show only <count> levels of calls, defaults to everything back to the program start
- up [<count>]
  - move up the call stack to the caller of the current function
  - repeats <count> times: “up 3” goes to caller's caller's caller.
- down [<count>]
  - go back down the call stack by <count> (default one) levels

Using gdb: Examining Data

- info registers / i r
  - print decimal and hex values in all general-purpose registers and EFLAGS
- print [/x or /d] <expression>
  - use $eax, $rdi, etc. for registers (yes, include the dollar sign)
- x 0xADDRESS,   x $register
  - prints what's at the given address or at the address stored in the register
  - the default display is one word (4 bytes)
  - specify format and the number of items to display: /s, /[num][size][format]
    - x/8a 0x15213   show 8 qwords, with symbolic addresses where possible
    - x/4wd 0xdeadbeef   show 4 words in decimal

Using gdb: Examining Data As You Go

- display [/{format}] <expr>
  - display the current value of the expression each time program execution stops
- info display / i di
  - show list of current auto-display expressions
- disable display <number>
  - temporarily disable the numbered expression
- delete display <number>
  - permanently remove the numbered auto-display expression

Sample GDB Run

Let's fire up the debugger:

```
[ralf@catshark ~/test]$ gdb crashes
GNU gdb (GDB) 7.6
    [...snip...]
Reading symbols from /usr22/ralf/test/crashes...(no debugging
symbols found)...done.
```

Start running the program:

```
(gdb) r one two three
Starting program: /usr22/ralf/test/crashes one two three
one
two
three

Program received signal SIGSEGV, Segmentation fault.
0x0040054e in strlength ()
```

Sample GDB Run (2)

OK, let's see how we got to the crash location:

```
(gdb) bt
#0  0x0040054e in strlength ()
#1  0x00400587 in main ()
```

What's the instruction that caused the crash?

```
(gdb) disas      <== note: defaults to function we're in
Dump of assembler code for function strlength:
   0x00400540 <+0>:     mov    $0x0,%eax
   0x00400545 <+5>:     jmp    0x40054e <strlength+14>
   0x00400547 <+7>:     add    $0x1,%eax
   0x0040054a <+10>:    add    $0x1,%rdi
=> 0x0040054e <+14>:    cmpb   $0x0,(%rdi)
   0x00400551 <+17>:    jne    0x400547 <strlength+7>
   0x00400553 <+19>:    repz retq
End of assembler dump.
```

Sample GDB Run (3)

What caused that boldfaced instruction to segfault? Check its operands:

```
(gdb) p $rdi
$1 = 0
```

Take a look at the calling function:

```
(gdb) up
#1  0x00400587 in main ()
(gdb) disas
Dump of assembler code for function main:
   0x00400555 <+0>:     push   %r12        <== preserve callee-saved regs
   0x00400557 <+2>:     push   %rbp
   0x00400558 <+3>:     push   %rbx
   0x00400559 <+4>:     mov    %rsi,%r12   <== put arg2 into %r12
   0x0040055c <+7>:     mov    $0x1,%ebx
   0x00400561 <+12>:    jmp    0x400578 <main+35>
```

[...continued on next slide...]

Sample GDB Run (4)

```
   0x00400563 <+14>:    mov    %rbp,%rsi
   0x00400566 <+17>:    mov    $0x40068c,%edi
   0x0040056b <+22>:    mov    $0x0,%eax
   0x00400570 <+27>:    callq  0x4003c0 <printf@plt>
   0x00400575 <+32>:    add    $0x1,%ebx
   0x00400578 <+35>:    movslq %ebx,%rax
   0x0040057b <+38>:    mov    (%r12,%rax,8),%rbp
   0x0040057f <+42>:    mov    %rbp,%rdi
   0x00400582 <+45>:    callq  0x400540 <strlength>
=> 0x00400587 <+50>:    test   %eax,%eax
   0x00400589 <+52>:    jg     0x400563 <main+14>
   0x0040058b <+54>:    mov    $0x0,%eax
   0x00400590 <+59>:    pop    %rbx
   0x00400591 <+60>:    pop    %rbp
   0x00400592 <+61>:    pop    %r12
   0x00400594 <+63>:    retq
End of assembler dump.
```

Sample GDB Run (5)

If we had returned instead of crashing, we would resume main() at 0x400587, so let's set a breakpoint at that call to strlength()

```
(gdb) break *0x400582
Breakpoint 1 at 0x400582
(gdb) run        <== note how 'run' re-uses the previous arguments
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /usr22/ralf/test/crashes one two three

Breakpoint 1, 0x00400582 in main ()
(gdb) i r        <== shortcut for "info registers"
rax            0x1      1
rbx            0x1      1
rcx            0x0      0
rdx            0x7fffffffe790   140737488349072
rsi            0x7fffffffe768   140737488349032
rdi            0x7fffffffea98   140737488349848
```

Sample GDB Run (6)

RDI is the actual important value, so let's auto-display it:

```
(gdb) display $rdi
1: $rdi = 140737488349848
(gdb) c
Continuing.
one

Breakpoint 1, 0x00400582 in main ()
1: $rdi = 140737488349852
(gdb)            <== note how just hitting <enter> repeats the 'continue' command
Continuing.
two

Breakpoint 1, 0x00400582 in main ()
1: $rdi = 140737488349856
```

Sample GDB Run (7)

I seem to have forgotten what breakpoints I set....

```
(gdb) i b          <== shortcut for “info breakpoints”
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400582 <main+45>
        breakpoint already hit 3 times
(gdb) i di         <== shortcut for “info display”
Auto-display expressions now in effect:
Num Enb Expression
1:   y  $rdi
```

We're done with this sample run

```
(gdb) quit
A debugging session is active.

        Inferior 1 [process 16716] will be killed.

Quit anyway? (y or n) y
```

Sample GDB Run: The Source

```
#include <stdio.h>

int strlength(const char *s)
{
  int len = 0 ;
  while (*s)
    {
    len++ ;
    s++ ;
    }
  return len ;
}
```

```
int main(int argc, char **argv)
{
  int i = 1 ;
  while (strlength(argv[i]) > 0)
    {
    printf("%s\n",argv[i]) ;
    i++ ;
    }
  return 0 ;
}
```

Analyzing the Compiled Code: main()

Function prologue, copy second parameter (argv) into %r12

```
0x00400555 <+0>:     push   %r12
0x00400557 <+2>:     push   %rbp
0x00400558 <+3>:     push   %rbx
0x00400559 <+4>:     mov    %rsi,%r12
```

int i = 1 ;

```
0x0040055c <+7>:     mov    $0x1,%ebx
```

while (...) is implemented with the test at the end:

```
0x00400561 <+12>:    jmp    0x400578 <main+35>
```

copy argv[i] into %rbp:

```
0x00400578 <+35>:    movslq %ebx,%rax
0x0040057b <+38>:    mov    (%r12,%rax,8),%rbp
```

function calls need their first parameter in %rdi:

```
0x0040057f <+42>:    mov    %rbp,%rdi
```

call strlength(argv[i]):

```
0x00400582 <+45>:    callq  0x400540 <strlength>
```

check if the result > 0:

```
0x00400587 <+50>:    test   %eax,%eax
0x00400589 <+52>:    jg     0x400563 <main+14>
```

Analyzing the Compiled Code (2)

The loop body

argv[i] is still in %rbp; function needs it in %rsi as its second argument:

```
0x00400563 <+14>:    mov    %rbp,%rsi
```

put address of string “%s\n” into function's first argument:

```
0x00400566 <+17>:    mov    $0x40068c,%edi
0x0040056b <+22>:    mov    $0x0,%eax
```

and call printf(“%s\n”,argv[i]):

```
0x00400570 <+27>:    callq  0x4003c0 <printf@plt>
```

finally, execute i++

```
0x00400575 <+32>:    add    $0x1,%ebx
```

and drop back into the loop test:

```
0x00400578 <+35>:    movslq %ebx,%rax
0x0040057b <+38>:    mov    (%r12,%rax,8),%rbp
etc.
```

Analyzing the Compiled Code (3)

Function exit: return 0;

```
0x0040058b <+54>:    mov    $0x0,%eax
```

Restore callee-saved registers:

```
0x00400590 <+59>:    pop    %rbx
0x00400591 <+60>:    pop    %rbp
0x00400592 <+61>:    pop    %r12
0x00400594 <+63>:    retq
```

Analyzing the Compiled Code: strlength()

Function prologue – none, because we don't use any callee-saved registers

int len = 0;

```
    0x00400540 <+0>:     mov    $0x0,%eax
```

The while loop is implemented with the test at the bottom:

```
    0x00400545 <+5>:     jmp    0x40054e <strlength+14>
```

while (*s)

```
    0x0040054e <+14>:    cmpb   $0x0,(%rdi)    <== note the size letter on 'cmp'
    0x00400551 <+17>:    jne    0x400547 <strlength+7>
```

loop body: len++; s++ ;

```
    0x00400547 <+7>:     add    $0x1,%eax
    0x0040054a <+10>:    add    $0x1,%rdi
```

fall through to the test again

```
    0x0040054e <+14>:    cmpb   $0x0,(%rdi)
    0x00400551 <+17>:    jne    0x400547 <strlength+7>
```

finally, return 'len', which was convienently stored in %eax all along

```
    0x00400553 <+19>:    repz retq   <== here's that workaround we mentioned
```

If You Get Stuck

- **Please read the writeup.** ***Please read the writeup.*** ***Please read the writeup.*** ***Please read the writeup!***
- CS:APP Chapter 3
- View lecture notes and course FAQ at http://www.cs.cmu.edu/~213
- Office hours Sunday through Thursday 5:00-9:00pm in WeH 5207
- Post a **private** question on Piazza
- `man gdb`, `man sscanf`, `man objdump`, gdb's `help` command

Bomb Lab Demo...