15-213 Recitation: Bomb Lab

September 17th 2018

Agenda

- Logistics
  - Bomb Lab Overview
  - Introduction to GDB
  - GDB and Assembly Tips

What is Bomb Lab?

- An exercise in reading x86-64 assembly code.
- A chance to practice using GDB (a debugger).

- Why?
  - x86 assembly is low level machine code. Useful for understanding security exploits or tuning performance.

  - GDB can save you days of work in future labs "cough Malloc cough" and can be helpful long after you finish this class.

Downloading Your Bomb

- All the details you'll need are in the write-up, which you most definitely have to read carefully before starting this lab anyway.

  Moving on.

Downloading Your Bomb

- Fine, here are some highlights of the write-up:
  - Bombs can only run on the shark machines. They fail if you run them locally or on another CMU server.
  - Each bomb is unique! If you download a second bomb, it will be different. You cannot mix and match bombs. Stick to only one bomb.
  - Bombs have six phases which get progressively harder.

Detonating Your Bomb

- Blowing up your bomb automatically notifies Autolab
  - ***Dr. Evil*** deducts 0.5 points each time the bomb explodes.
  - It's very easy to prevent explosions using break points in GDB. More information on that soon.
- Inputting the correct string moves you to the next phase.
- Don't tamper with the bomb. Skipping or jumping between phases detonates the bomb.
- You have to solve the phases in order they are given. Finishing a phase also notifies Autolab automatically.

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.

GDB

- You can open gdb by typing into the shell:
- $ gdb

- This is the notation we're using for the next few slides:
- $ cd // Type the command into the bash shell
- (gdb) break // The command should be typed in GDB

Form Pairs

- One student needs a laptop
- Login to a shark machine and type these commands:
- $ wget http://www.cs.cmu.edu/~213/activities/rec3.tar
- $ tar xvpf rec3.tar
- $ cd rec3
- $ make
- $ gdb act1

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
}
```

Activity 1

- (gdb) break main // tells GDB to pause right before entering main
- (gdb) run 15213 // starts execution with the argument “15213”

- You should see GDB print out:
- Breakpoint 1, main (argc=1, argv=[…]) at act1.c:5

- (gdb) continue // this continues execution until another break point or the end of execution

- (gdb) clear main // remove the breakpoint at function main
- (gdb) run 15213 // Q: What happens now?

Activity 1 cont

- (gdb) disassemble main // show the assembly instructions in main
- (gdb) print (char*) [0x4…] // hex code from <+14>
  // 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 18213
- (gdb) print argv[1] // Q: What does this print out?
- (gdb) quit // exit GDB; agree to kill the running process

Activity 2

- $ gdb act2
- (gdb) break 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
- (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?

Activity 2 cont.

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

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

- (gdb) disassemble
- (gdb) stepi // 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. Where are the “=>” characters printed on the left side?

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.

- $ 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)

Activity 3 cont.

- (gdb) disassemble compare
- Q. Where is the return value set in compare?

- (gdb) break compare
- Now run act3 with two numbers
- Q. Using nexti or stepi, how does the value in register %rbx change, leading to the cmp instruction?

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]

```
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]

```
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]

```
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]

```
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]

```
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]

```
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

- Stack:

[some old stack items]

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

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 walkthrough

- $ gdb act4
- (gdb) disassemble main
- Note 4 function calls: strtoq, compute, and fwrite
- $ man strtoq
    - 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?
- (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 walkthrough

- (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
- (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 print

```
(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 walkthrough

- 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
- (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 walkthrough

- Working backwards from <+21> with $rdx = 0
- cmp $0x4, %edx
  - will not set CF or ZF, so ja will not jump to <+51>
- Want $edx = 0. Thus from <+3> want $edx = 0
- lea (%rdi,%rdi,2),%eax
  - Does $eax = $rdi + 2 * $rdi = 3 * $rdi
  - We want $edx = 0, so $rdi = 0
- Since the input $rdi = 0, let's run his 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 walkthrough

- 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.
```

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, 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

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.

Quick Assembly Info

- $ 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
```

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? Handout
- Need to know how to use certain GDB commands? Handout
  - Also useful: http://csapp.cs.cmu.edu/2e/docs/gdbnotes-x86-64.pdf
- Don’t know what an assembly instruction does? Lecture slides
- Confused about control flow or stack discipline? Lecture slides

Text User Interface (TUI) mode (optional)

- (gdb) layout asm
- (gdb) layout reg
- (gdb) focus cmd
- Switch between TUI and regular mode with Ctrl-X + Ctrl-A
- TUI mode is buggy on the shark machines, so you still need to know how to use regular mode in case TUI glitches out.
- Tip: Only use TUI when single stepping your code. For all other use cases, use regular mode. If you see glitches and your screen get garbled, you might have to exit GDB and start over.