15-213
Introduction to Computer Systems

With Your TA!
Schedule

- News
- GDB
- Assembly Code
- Bomblab
- Bomblab Example
News

- Datalab will be graded by next Monday
- Scores will show up on Autolab.
  - Questions? Complaints?
  - Email the TA that graded your lab. (Their andrewID will appear at the bottom of your feedback)
- TA's will rotate
  - So no one TA will grade two of your labs.
- Labs will be hand graded and handed back in recitation
  - Please update your Autolab profile specifying which recitation you will pick up your lab in.
GDB
Gnu DeBugger

• Step through program execution
• Examine values of program variables.
• Trap system signals (such as SIGSEGV)
• Set breakpoints to halt execution at any point
• Watch variables to see when they change.
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a, b, c;

    a = 4;
    b = 10;
    c = a * b;

    printf("A is %d, b is %d, and c is %d\n", a, b, c);

    return 0;
}
```

**GDB Example**

```
(gdb) break simple.c:9
Breakpoint 1 at 0x804839e: file simple.c, line 9.
(gdb) run
Starting program: 15213/rec2/a.out
Breakpoint 1, main () at simple.c:9

9   c = a * b;
(gdb) print a
$1 = 4
(gdb) print b
$2 = 10
(gdb) print c
$3 = 134513642
(gdb) where
#0  main () at simple.c:9
(gdb) continue
Continuing.
A is 4, b is 10, and c is 40
```

Program exited normally.
Some GDB Commands

- **run [arg1 [arg2 [...]]]**
  - executes the program with specified arguments
- **break [file.c:]line# | functionName | memAddr**
  - sets a break point
    - breaks execution BEFORE executing the statement!!!!!
- **print varName | $register**
  - prints a variable or register's value.
- **steipi**
  - step through one instruction in assembly
Some GDB Commands (cont)

- disas \textit{function}\]
  - show the disassembly of the current code (or the function)

- continue
  - continue program execution after stopping at a breakpoint.

- info \textit{break} | \textit{registers} | .....
  - shows information about breakpoints/registers/.....
Assembly Code
x86 Assembly

- Variables ==> Registers
  - %esp -> Stack Pointer
  - %ebp -> Stack Base Pointer
  - %eax -> Function Return Value
  - %eip -> Instruction Pointer
  - (a bunch of other ones)
x86_64 Assembly

• Variables ==> Registers
  - %rsp -> Stack Pointer
  - %rbp -> Stack Base Pointer
  - %rax -> Function Return Value
  - %rip -> Instruction Pointer
  - %rdi, %rsi, %rdx, %rcx -> Function Arguments
  - (and a bunch-bunch more)
Assembly Addressing

(R) ==> *(Reg(R))
- The memory at address stored in register R

D(R) ==> *(Reg(R) + D)
- The memory at the address (R + (constant D))
- ex: 4(%eax) ==> *(%eax + 4)

D(Rb,Ri,S) ==> *(Reg(Rb) + Reg(Ri)*S + D)
- Constant Displacement 'D'
- Base Register 'Rb'
- Index Register 'Ri'
- Scale (1,2,4,8..)
## Addressing Examples

<table>
<thead>
<tr>
<th>%eax</th>
<th>0xb800</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>0x10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$4(%eax)</td>
<td>4 + 0xb800</td>
<td>0xb804</td>
</tr>
<tr>
<td>(%eax,%ecx)</td>
<td>0xb800 + 0x10</td>
<td>0xb810</td>
</tr>
<tr>
<td>(%eax,%ecx,$4)</td>
<td>0xb800 + 4*0x10</td>
<td>0xb840</td>
</tr>
<tr>
<td>$4(%eax,%ecx)</td>
<td>4 + 0xb800 + 0x10</td>
<td>0xb814</td>
</tr>
<tr>
<td>$0xFF0000(%eax,%ecx,$4)</td>
<td>0xFF0000+0xb800+4*0x10</td>
<td>0xFFb840</td>
</tr>
</tbody>
</table>
Arithmetic Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl</td>
<td>Src, Dest</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subl</td>
<td>Src, Dest</td>
<td>Dest = Dest - Src</td>
</tr>
<tr>
<td>imull</td>
<td>Src, Dest</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>sall</td>
<td>Src, Dest</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarl</td>
<td>Src, Dest</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shrl</td>
<td>Src, Dest</td>
<td>Dest = Dest &gt;&gt;&gt; Src</td>
</tr>
<tr>
<td>xorl</td>
<td>Src, Dest</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andl</td>
<td>Src, Dest</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orl</td>
<td>Src, Dest</td>
<td>Dest = Dest</td>
</tr>
<tr>
<td>incl</td>
<td>Dest</td>
<td>Dest ++</td>
</tr>
<tr>
<td>decl</td>
<td>Dest</td>
<td>Dest --</td>
</tr>
<tr>
<td>negl</td>
<td>Dest</td>
<td>Dest = -Dest</td>
</tr>
<tr>
<td>notl</td>
<td>Dest</td>
<td>Dest = ~Dest</td>
</tr>
</tbody>
</table>
Examples

- C function with some simple math
- Let's examine the assembly code
  - both unoptimized and optimized
- Step through this code with GDB
Bomblab
Bomblab

- Solve a series of stages by finding the password for a function
- *We give you a compiled binary*
- *You read the assembly code to figure out the passwords*
Bomblab Hints

• If it blows up, you're doing it wrong!
• Use GDB to step through the program, following execution and watching what happens to variables
• Figure out what checks are made and how to pass them
Bomblab Example

- Lets return to the example we had and try to get it to return certain output values.
A note on Bomblab

- You can usually make some guesses and solve each stage that way.
- But, if you are stuck, just work through each line of assembly and try to re-construct the C-code.
Final Thoughts

- There is LOTS of documentation for this stuff on the internet.
- Become comfortable with GDB, you'll have to use it a lot.
- Remember: Office Hours: Mon-Thurs: 6:30-9:30pm in WeH 5304
- 15-213-staff@cs.cmu.edu !!!