CSE 351
GDB + Lab 2
Lab 2

• Already out!
• Due Friday, February 3, 2017 at 5:00pm

• Reading and understanding x86_64 assembly
• Debugging and disassembling programs

• Today:
  • General debugging for C with GDB
GDB

• GNU Debugger

• GDB is your best friend
  • start, stop, peek in, poke at your program

• Today we will be going over many of the features that will make GDB a great resource for you this quarter

• Useful in future classes!
  • CSE 333, CSE 451, CSE 484 etc.
Breakpoints

- In order to step through code, we need to be able to pause execution.

- GDB allows you to set breakpoints, just like when you debugged Java programs in Eclipse or jGRASP.

- `break` (b for short) command creates breakpoints.
- `info break` shows your breakpoints
int bar(int x) {
    int a;
    a = baz(x);
    return a + 1;
}

int foo() {
    int x, y, z;
    x = 5;
    y = bar(x);
    z = y - 2;
    return z;
}

int baz(int x) {
    return x << 1;
}
int foo() {
    int x,y,z;
    x = 5;
    y = bar(x);
    z = y - 2;
    return z;
}

int bar(int x) {
    int a;
    a = baz(x);
    return a + 1;
}

int baz(int x) {
    return x << 1;
}
Printing

- `print` to look at values
- `x` to examine memory
- `help x` to see how to use it
  - `help` anything else!
Printing

How can I display something persistently?
  display /i $pc  (current instruction)
  display /x $rax  (contents of %rax in hex)
  display /16bd $rdi  (16 bytes of memory pointed to by %rdi as integers in decimal)
Debugging

• GDB will stop you when you get an error
  • null-dereference, 1/0

• backtrace (bt) shows how you got there
  • Viewing a backtrace can be very helpful in debugging.

• list shows you C code
• disas shows you assembly
  • objdump as well
Register Conventions

- **Parameters:** `%rdi, %rsi, %rdx, %rcx, %r8, %r9`
- **Return value:** `%rax`
- We’ll see how this is used in phase_1 of the lab
Register Conventions

- Let’s say one of your functions looks like
  ```
  foo(){
    int bar = some + complex + calculation;
    int bar2 = complex_subroutine();
    return bar * bar2;
  }
  ```
- What happens to ‘bar’ if it was in a register?
- Some registers are caller-saved, others callee-saved
- Why have a calling convention? Linked libraries, ...
The x86 Calling Convention

<table>
<thead>
<tr>
<th>Caller-Saved Registers</th>
<th>Callee-Saved Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%rbx</td>
</tr>
<tr>
<td>%rdi</td>
<td>%r12</td>
</tr>
<tr>
<td>%rsi</td>
<td>%r13</td>
</tr>
<tr>
<td>%rdx</td>
<td>%r14</td>
</tr>
<tr>
<td>%rcx</td>
<td>%rbp</td>
</tr>
<tr>
<td>%r8</td>
<td>%rsp</td>
</tr>
<tr>
<td>%r9</td>
<td></td>
</tr>
<tr>
<td>%r10</td>
<td></td>
</tr>
<tr>
<td>%r11</td>
<td></td>
</tr>
</tbody>
</table>

**Return Value**

Arguments 1-6

Temporaries

Base Pointer

Stack Pointer
Control Flow

- 1-bit condition code registers \([CF, SF, ZF, OF]\)
- Set as side effect by arithmetic instructions or by `cmp`, `test`

**CF - Carry Flag**
- Set if addition causes a carry out of the most significant (leftmost) bit.

**SF - Sign Flag**
- Set if the result had its most significant bit set (negative in two’s complement)

**ZF - Zero Flag**
- Set if the result was zero

**OF - Overflow Flag**
- If the addition with the sign bits off yields a result number with the sign bit on or vice versa
Lab 2

• Requires you to defuse “bombs” by entering a series of passcodes
  • Not real bombs/viruses/etc!
• Each passcode is validated by some function
  • You only have access to the assembly code
• It’s your job to determine what passcodes will prevent the program from ever calling the `explode_bomb()` function
• Each student has a different bomb
Lab 2 Files

- **bomb**
  - The executable bomb program
- **bomb.c**
  - This is the entry point for the bomb program, not including the phase_* functions
- **defuser.txt**
  - Place your passcodes here once you solve each phase, separated by newline
  - Can be passed as an argument to prevent you from entering the passcodes manually each time
  - `run defuser.txt` from within GDB
Lab 2 Notes

- The bomb uses `sscanf`, which parses a string into values
- Example:
  ```c
  int a, b;
  sscanf("123, 456", "%d, %d", &a, &b);
  ```
- The first argument is parsed according to the format string
- Specifiers like printf
Lab 2 Tips

• Print out the disassembled phases
  • `objdump -d bomb > bomb.s`
  • You can then print out `bomb.s`
  • Mark the printouts up with notes
• Try to work backwards from the “success” case of each phase
• Remember that some addresses are pointing to strings located elsewhere in memory
  • Print them out in GDB
Lab 2 Phase 1