CSE 351
More GDB, Intro to x86 Calling Conventions, Control Flow, & Lab 2
GDB Exercise – Display Assembly

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

Others:
- `disas`
- `layout asm (Ctrl-X a to exit...)`
- `or just print it all out! (objdump -d bomb...)`
Register Conventions Intro

• Where do parameters and return values go for function calls?
  • Parameters: %rdi, %rsi, %rdx, %rcx, %r8, %r9
  • Return value: %rax
  • We’ll see how this is used in phase_1 of the lab
Function Calls & Registers Intro

• 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, ...
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
Control Flow Examples

x86:

```
  test  %rax, %rax ; set ZF to 1 if rax == 0
  je    <location>   ; jump if ZF == 1

  cmp  %rax, %rbx
  jg    <location>
      (hint: jg checks if ZF = 0 and SF = OF)

  cmp  %rax, %rbx
  xor  %rbx, %rbx
  js    <location>
      (hint: js checks if MSB of result = 1)
```

Result:

- Jumps to <location> if rax == 0
- *rax and rbx are interpreted as signed then compared, if rbx > rax we jump to <location>*
- *Never jumps to <location>*
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, and it calls functions whose source code is not available to you

- **defuser.txt**
  - Contains passcodes, each separated by a newline
  - Place your passcodes here once you solve each phase
  - Can be passed as an argument to prevent you from entering the passcodes manually each time
  - To do this, you can run `set args defuser.txt` from within GDB and then whenever you run your program, it will automatically read its input from defuser.txt
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
• After this code is run, a = 123 and b = 456
Lab 2 Tips

• Print out the disassembled phases
  • To disassemble a program, run `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

• Let’s Dive In!