CSE 351: The Hardware/Software Interface

Section 3: Control flow, assembly, and Lab 2
Control Flow

- do-while: a useful variation on the while loop
  ```
  int value;
  do {
      value = value + 1;
  } while (value != 4);
  ```

- exit condition is only relevant after executing the body of the loop once
Switch Statements

- switch statement (compare to repeated if-else)
  ```
  int computeSomething(int value) {
    switch (value) {
      case 0:
      case 1:
        value = value + 1;
        break;
      default:
        value = value - 1;
    }
  }
  ```

- in absence of "break", code execution will "fall through"
Switch Statements (continued)

- switch statement (compare to repeated if-else)

```java
int computeSomething(int value) {
    switch (value) {
        case 0:
            // break; <- after commenting this out, execution continues through the "default" logic as well
        case 1:
            value = value + 1;
            default:
                value = value - 1;
            return value;
    }
}
```
Goto

- Can be useful in limited cases, but are often considered bad style (see "Go To Statement Considered Harmful", Dijkstra 1968)

```c
int badCode(int value) {
    start:
    value ++;
    if (value > 2)
        goto end;
    else
        goto start;

    end:
    return value;
}
```
x86 Basics

- Used by overwhelming majority of servers, desktops, and laptops today

- Extremely backwards compatible
  - pro: learning 32-bit x86 will teach you a lot about 64-bit x86
  - con: ...but may be difficult because of decisions made a long time ago

- Can be difficult to parse at a glance!
Three Basic Kinds of Instructions

- Perform arithmetic function on register or memory data
  - e.g. \texttt{addq} \ $45, \%rax

- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
  - e.g. \texttt{movq} \ %rax, \(%rdx\)

- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches
What Is A Register (again) (again)?

- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)

- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but especially x86
x86 vs. x86-64

- Simplest: it’s bigger! (64 bits vs. 32 bits)

- What does this really mean?
  - \(2^{32}\) bytes = 4,294,967,296 bytes = 4 gigabytes
  - amount of memory accessible, size of important things (registers, integers, etc.)

- How about in terms of the assembly we will be looking at?
  - adds a new size prefix: q, for 8-byte chunks
  - extends registers both in number (%r8-%r15) and in size (%eax is now contained in %rax)
  - changes some elements of function calls etc.
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>x86</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Extend existing registers, and add 8 new ones; *all* accessible as 8, 16, 32, 64 bits.

64-bits wide
Basic Instructions

- **Arithmetic**
  - add, sub, mul, idiv

- **Logical/Bitwise**
  - and, or, xor, neg, sal/shl (equivalent), sar/shr

- **Control**
  - jmp, je, jne, jg, jl, jle, jge
  - Use after test or cmp instructions
    - test – bitwise AND, sets flags
    - cmp – subtraction, sets flags
  - ret, used to return from a function

- **Other**
  - Stack instructions: push, pop
  - Data manipulation: mov, enter, leave
Calling conventions

- Return value will be put in %rax

- x86-64 has many extra registers compared to 32-bit x86

- Registers are much faster than stack, so x86-64 puts the first six arguments into registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
Lab 2

- Use gdb, objdump, etc. to defuse six bombs

- The files involved:
  - bomb, an executable bomb (takes code phrases as input)
  - bomb.c, defines the entry point of the program and calls functions whose source code is not available to you
  - defuser.txt, contains pass phrases for each stage, separated by newlines. Add each passphrase here as you discover it

- Start early!
  - Like lab 1, this can often take more time than expected
  - We have lots of office hours to help you, but this works better earlier than later
Lab 2 notes

- Each student in the class has a different bomb; no two have the same answers

- Put the pass phrases you’ve already discovered in defuser.txt so that you don’t have to type them in every time

- gdb has built-in help for all its functions, and is extensively documented online

- Unix commands **man** and **apropos** (searches **man** pages) are your friend!
Lab 2 notes

- The bomb uses function `sscanf`, which parses a string into values
- As an example:

  ```c
  int a, b;
  sscanf("123, 456", "%d, %d", &a, &b);
  ```

- The first argument is parsed according to the format string of the second argument
- Upon success, the values of `a` and `b` will be set to `123` and `456`, respectively
- Refer to `man 3 sscanf` for more information