Assembly and Bomblab

Outline

x86 Architecture

- Program counter
  - Contains address of next instruction
  - *eip* (x86), *rip* (x86-64)
- Stack registers
  - Contain addresses of base and top of current stack frame
  - Covered tomorrow in lecture
  - *esp* and *ebp* (x86), *rsp* and *rbp* (x86-64)
- General purpose registers
  - *eax*, *ebx*, *ecx*, *edx*, *esi*, *edi* (x86)
  - *rax*, *rbx*, *rcx*, *rdx*, *rsi*, *rdi*, *r*8, *r*9, *r*10, *r*11, *r*12, *r*13, *r*14, *r*15 and sometimes *rbp* (x86-64)
- Condition codes
- Other stuff
  - Control registers, segment selectors, debug registers, SIMD registers, floating point registers, etc

Data Types

- Integer data
  - Data values (signed and unsigned)
    - 1, 2, or 4 bytes (or 8 on x86-64)
  - Addresses
    - 4 bytes (x86) or 8 bytes (x86-64)
- Floating point data
  - 4, 8 or 10 bytes
- No aggregate data types!

Operand Types

- Immediate value
  - Examples: `$0x15213`, `$-18213`
  - Like a C constant, prefixed with '$'
  - 1, 2 or 4 bytes (or 8 on x86-64)
- Register
  - Examples: `%esi`, `%eax`
  - Some instructions (e.g. `div`) use specific registers
- Memory
  - Examples: `(%esi)`, `12(%eax,%ebx,4)`
  - Format is `O(Rb,Ri,S)`
    - `Rb` is the base address register
    - `Ri` is the index address register
    - `S` is the index scale (1, 2, 4 or 8)
    - `O` is a constant offset
  - Equivalent to C style `Rb[Ri*S + O]`

Memory access

- `movl src,dst`
  - Example: `movl $0x15213,%eax`
  - Moves data between registers and memory
  - Immediate value to register or memory
  - Register to other register or memory
  - Memory to register
- `leal src,dst`
  - Example: `leal (%eax,%eax,2),%eax`
  - Computes an address specified by `src` and saves it in `dst`
  - Does not actually dereference `src`!
  - Sometimes used by compilers as a fast alternative to `imul`
    - Example above triples %eax

Arithmetic Operations

Two operand commands:

Format	Result
`addl src,dst`	`dst += src`
`subl src,dst`	`dst -= src`
`imull src,dst`	`dst *= src`
`sall src,dst`	`dst <<= src`
`sarl src,dst`	`dst >>= src`
`xorl src,dst`	`dst ^= src`
`andl src,dst`	`dst &= src`
`orl src,dst`	`dst |= src`

One operand commands:

Format	Result
`incl dst`	`dst++`
`decl dst`	`dst--`
`negl dst`	`dst = -dst`
`notl dst`	`dst = ~dst`

There are also 64 bit equivalents (e.g. `addq`).

Arithmetic Example

```
void foo () {

  int a = 0;
  int b = 2;


  int c = a - b;


  int d = c << 2;

}
```

```
pushq %rbp
movq %rsp, %rbp
movl $0, -16(%rbp)
movl $2, -12(%rbp)
movl -12(%rbp), %edx
movl -16(%rbp), %eax
subl %edx, %eax
movl %eax, -8(%rbp)
movl -8(%rbp), %eax
sall $2, %eax
movl %eax, -4(%rbp)
leave
ret
```

Condition Codes

- Set as side-effect of arithmetic operations in the `eflags` register
- CF set on unsigned integer overflow
- ZF set if result was 0
- SF set if result was negative
- OF set on signed integer overflow
- `testl a,b` and `cmpl a,b` are similar to `andl a,b` and `subl a,b` but *only* set condition codes
- Use `set* reg` instructions to set register `reg` based on state of condition codes.

Conditionals

- Change the instruction pointer with the `j*` operations
  - `jmp dst` unconditionally jumps to the address `dst`
  - Use other jump variants (e.g. `jne` or `jg`) to conditionally jump
    - Usually a `testl` or `cmpl` followed by a conditional jump
- Conditional moves added in the x686 standard
  - `cmov* src,dst`
  - Significantly faster than a branch
  - GCC does not use these by default for 32 bit code to maintain backwards compatibility

Conditional Example

```
void bar() {

   int a = 2;
   int b = 0;
   if (a > 7) {

      b++;
   }
}
```

```
  pushq   %rbp
  movq    %rsp, %rbp
  movl    $2, -8(%rbp)
  movl    $0, -4(%rbp)
  cmpl    $7, -8(%rbp)
  jle     .L3
  addl    $1, -4(%rbp)
.L3:
  leave
  ret
```

Overview

- Series of stages, all asking for a password
- Give the wrong password and the bomb explodes
  - You lose a half point every time your bomb explodes
  - The bomb should never explode if you're careful
- We give you the binary, you have to find the passwords
- The binary *ONLY* runs on the shark machines

GDB - GNU Debugger

- Syntax: `$> gdb ./bomb`
- Useful commands
  - `run <args>` Runs the bomb with specified command line arguments
  - `break <location>` Stops the bomb just before the instruction at the specified location is about to be run
  - `info functions` Lists the names of all functions.
  - `stepi` Steps the program one instruction. `nexti` will do the same, but skipping over function calls.
  - `print <variable>` Prints the contents of a variable
  - `x/<format> <address>` Prints contents of the memory area starting at the address in a specified format
  - `disassemble <address>` Displays the assembly instructions near the specified address
  - `layout <type>` Changes the layout of GDB. `layout asm` followed by `layout reg` is great
  - `help` and `help <command>` Explains GDB usage.

Others

- `strings`
  - Dumps all strings in the binary
  - Function names, string literals, etc
- `objdump`
  - The `-d` option disassembles the bomb and outputs the assembly to the terminal
  - The `-t` option dumps the symbol table (all function and global variable names) to the terminal
  - You probably want to redirect the output into a file
    `objdump -d ./bomb > bomb_asm`

Walkthrough

Example bomb walkthrough