Reverse Engineering

CS-UY 3943-G / CS-GY 9223-H

Josh Hofing

What the heck is RE?

- Reverse Engineering is a broad field
  - Hardware reversers
  - Software reversers
  - Firmware reversers
- The field of understanding how things work
  - *Especially* the things we're not supposed to understand
    - Every large piece of software has dark corners that they assume nobody will ever look at

What we'll be focusing on

- 64-bit x86 binaries
  - Running on Linux
  - Without *too* much obfuscation involved

x86_64

- AKA amd64, or i64, or x86 64-bit
  - The unintended successor to Intel’s x86 architecture
    - Intel had a thing called Itanium that they thought would go *really* well
      - It didn’t
    - AMD decided to extend Intel’s existing 32-bit architecture to 64-bits
      - And here we are!

x86_64

- This is a CISC, variable-length instruction set, multi-sized register access instruction set.
  - CISC means Complex Instruction Set Computing
    - A single instruction can do a bunch of different things at once
      - Memory accesses, register reads, etc.
  - Variable-length instruction set
    - Different instructions can be different sizes
      - x86_64 instructions can be anything from 1 to 16 bytes long
  - Multi-sized register access means that you can access certain parts of a register which are different sizes

Registers

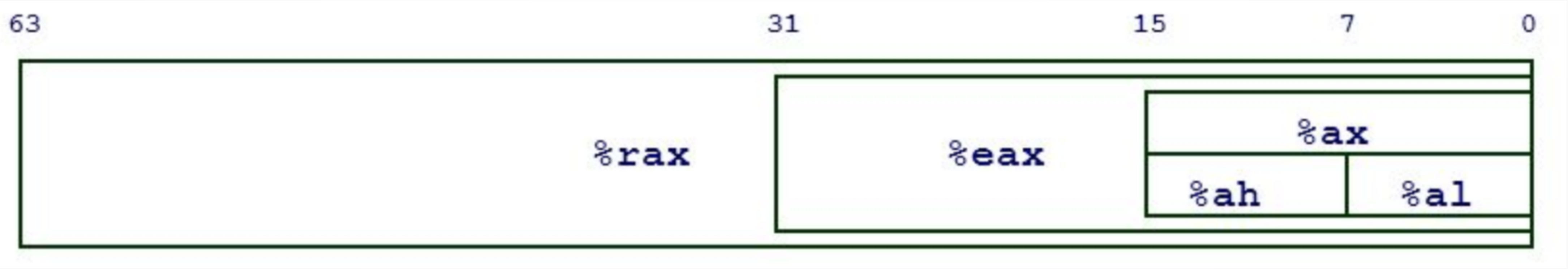
- So what is a register?
- Basically a local variable on the CPU
  - The "64" in x86_64 means the registers are 64 bits
  - Instantly accessible by the CPU
- x86_64 has *many*
  - rax, rbx, rcx, rdx, rdi, rsi, rsp, rip, r8-r15, and more!

Registers

- Special registers
  - RIP: the instruction pointer
  - RSP: the stack pointer
  - RBP: the base pointer

Registers

- Sized accesses



http://slideplayer.com/slide/9428596/29/images/12/64-bit+registers+Multiple+access+sizes+%25rax,+%25rbx,+%25rcx,+%25rdx.jpg

Instructions

- Represents a single operation for the CPU to perform*
- Different types
  - Data movement
    - mov rax, [rsp - 0x40]
  - Arithmetic
    - add rbx, rcx
  - Control-flow
    - jne 0x80004000

Instructions

- Represents a single operation for the CPU to perform*
- Again, x86 is a CISC architecture
- repne scasb
  - Repeats up to ecx times over memory at edi looking for NULL byte, decrementing ecx each byte
  - (Essentially) strlen() in a single instruction!

Instructions

- What should the CPU execute?
- Determined by the RIP register
  - IP = instruction pointer

- Fetch the instruction at the address in RIP
- Decode it
- Run it

Instructions

- Example: `mov rax, 0xdeadbeef`

Operation

Register

Immediate

Instructions

- Example: `mov rax, [0xdeadbeef + rbx * 4]`

Operation

Register

Effective address

Get the contents from this memory address

Example

0x0804000: mov eax, 0xdeadbeef

0x0804005: mov ebx, 0x1234

0x080400a: add, rax, rbx

0x080400d: inc rbx

0x0804010: sub rax, rbx

0x0804013: mov rcx, rax

Register Values:

rip = 0x0804000

rax = 0x0

rbx = 0x0

rcx = 0x0

rdx = 0x0

Example

0x0804000: mov eax, 0xdeadbeef

0x0804005: mov ebx, 0x1234

0x080400a: add, rax, rbx

0x080400d: inc rbx

0x0804010: sub rax, rbx

0x0804013: mov rcx, rax

Register Values:

rip = 0x0804005

rax = 0xdeadbeef

rbx = 0x0

rcx = 0x0

rdx = 0x0

Example

0x0804000: mov eax, 0xdeadbeef

0x0804005: mov ebx, 0x1234

0x080400a: add, rax, rbx

0x080400d: inc rbx

0x0804010: sub rax, rbx

0x0804013: mov rcx, rax

Register Values:

rip = 0x080400a

rax = 0xdeadbeef

rbx = 0x1234

rcx = 0x0

rdx = 0x0

Example

0x0804000: mov eax, 0xdeadbeef

0x0804005: mov ebx, 0x1234

0x080400a: add, rax, rbx

0x080400d: inc rbx

0x0804010: sub rax, rbx

0x0804013: mov rcx, rax

Register Values:

rip = 0x080400d

rax = 0xdeadd123

rbx = 0x1234

rcx = 0x0

rdx = 0x0

Example

```
0x0804000: mov eax, 0xdeadbeef
0x0804005: mov ebx, 0x1234
0x080400a: add, rax, rbx
0x080400d: inc rbx
0x0804010: sub rax, rbx
0x0804013: mov rcx, rax
```

Register Values:

rip = 0x0804010

rax = 0xdeadd123

rbx = 0x1235

rcx = 0x0

rdx = 0x0

Example

0x0804000: mov eax, 0xdeadbeef

0x0804005: mov ebx, 0x1234

0x080400a: add, rax, rbx

0x080400d: inc rbx

0x0804010: sub rax, rbx

0x0804013: mov rcx, rax

Register Values:

rip = 0x0804013

rax = 0xdeadbeee

rbx = 0x1235

rcx = 0x0

rdx = 0x0

Example

0x0804000: mov eax, 0xdeadbeef

0x0804005: mov ebx, 0x1234

0x080400a: add, rax, rbx

0x080400d: inc rbx

0x0804010: sub rax, rbx

0x0804013: mov rcx, rax

Register Values:

rip = 0x0804016

rax = 0xdeadbeee

rbx = 0x1235

rcx = 0xdeadbeee

rdx = 0x0

Control Flow

- How to express conditionals in x86?
  - Conditional jumps
    - jnz <address>
    - je <address>
    - jge <address>
    - jle <address>
    - Etc
  - They jump if their condition is true, and just go to the next instruction otherwise
  - What are these checking to decide?

EFLAGS

- EFLAGS is everyone's favorite register to forget
  - But it's important
  - As the name implies, flags are stored here
  - Many instructions set them, for example…
    - add rax, rbx sets the o (overflow) flag if the sum is greater than a 64-bit register can hold, and wraps around
      - You can jump based on that with a jo instruction
    - The most important thing to remember is the cmp instruction

      ```
      cmp rax, rbx
      jle error
      ```

      - Jumps if rax <= rbx

Memory: It's All Just Bytes!

Memory

- Instructions, numbers, strings, everything!
- Always represented in hex

- `add rax, rbx                 == 48 01 d8`
- `mov rax, 0xdeadbeef          == 48 c7 c0 ef be ad de`
- `mov rax, [0xdeadbeef] == 67 48 8b 05 ef be ad de`

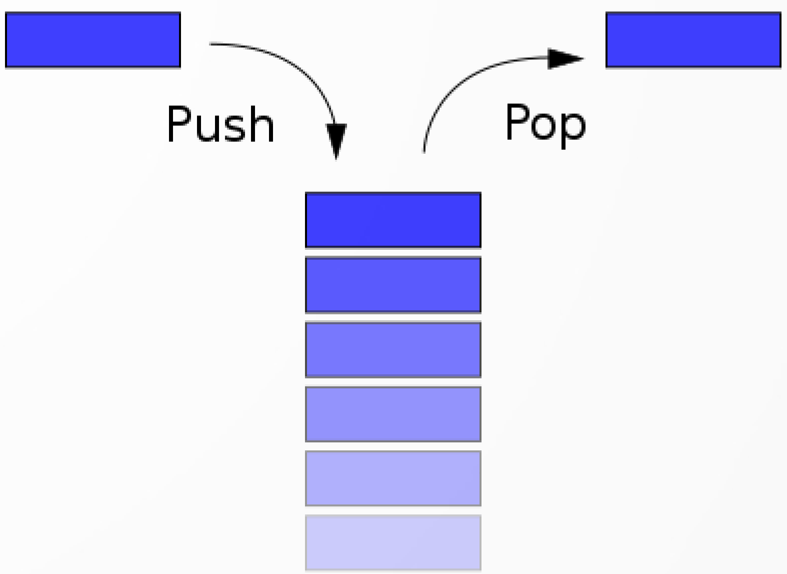
- `"Hello" == 48 65 6c 6c 6f`

Addresses

- Memory ~= a big array
  - Indices into this "array" are memory addresses
- From earlier: `mov rax, [0xdeadbeef]`
  - Square brackets means "get the data at this address"
  - Analagous to the C/C++ syntax:
    - rax = *0xdeadbeef;

The Stack

- From data structures
- LIFO
  - "Push" things on to the top
  - "Pop" things off the top
- Built in to most architectures
  - Nothing fancy! Just memory, rsp, and rbp!



The Stack

- rsp register is the "stack pointer"
  - Points to the current top of the stack
- `push rax` ends up decreasing rsp
- `pop rax` ends up increasing rsp
- Seems backwards?

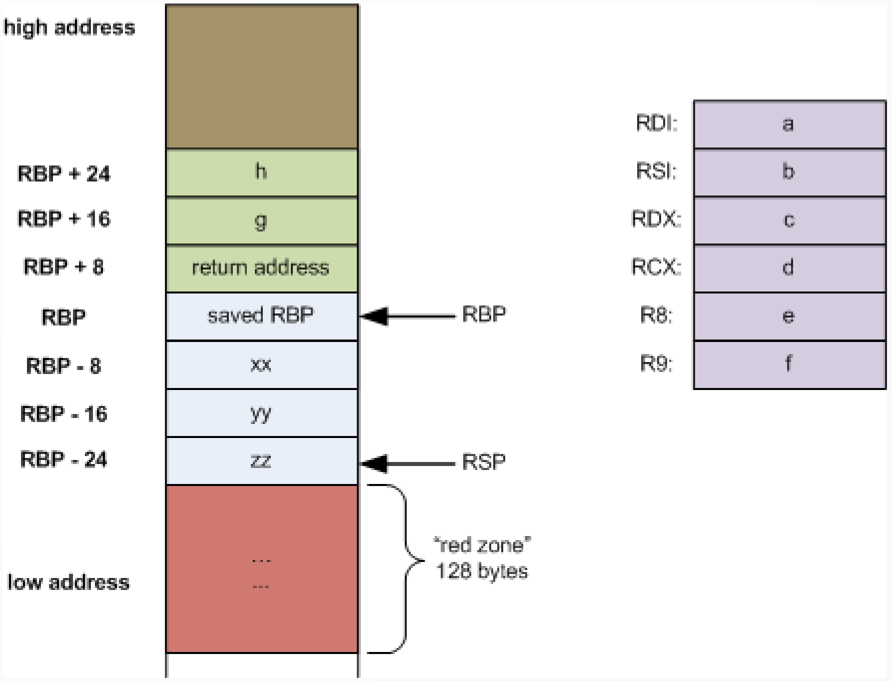
THE STACK GROWS DOWN

The Stack

- `push rax`
  - Decrements rsp by 8
  - Moves the contents of rax into memory at the new rsp
  - `sub rsp, 8`
  - `mov [rsp], rax`
- `pop rax`
  - `mov rax, [rsp]`
  - `add rsp, 8`

Calling Functions

- What do functions look like in assembly?
  - It’s just code that ends in a ret instruction
  - Usually, it does stuff with the stack to create a Stack Frame
    - push rbp
    - mov rbp, rsp
    - sub rsp, 0x100
  - call <address>
    - Pushes rip
    - Jumps to <address>
  - ret
    - Pops rip
    - Cleanup stackframe first!



Calling Conventions

- How do I pass arguments into my functions?
  - This is entirely by *convention*, so there's no real rules
  - However, everything we'll touch is using the SystemV AMD64 ABI, so the following calling conventions hold:
    - The first 6 arguments to a function are passed, from left-to-right, in these registers:
      - rdi, rsi, rdx, rcx, r8, r9
    - Further arguments are pushed to the stack
    - The return value of the function is stored in rax when the function returns
- It takes a long time for people to really remember this
  - Don't feel bad about needing a reference!

Please experiment!

- The Compiler Explorer (at https://godbolt.org) is an excellent resource
  - Type in some C (or C++), and it live-updates with the generated assembly, with corresponding lines highlighted
  - Play around with it, and try to understand how various high-level constructs map into assembly
- There's a lot of other great resources about this stuff online, search around and learn!
- Also, I really hope everyone has a disassembler they're happy with by now. You'll need it for the homework!