

Backdoor, Endianness, x86-64

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

Announcements

- Homework 4 due Monday after break on Gradescope
 - You have written most of this code already
 - Lab 7 may provide a fast way to get started
- Regrade requests for midterm 1 due Friday after break

Backdoors

Backdoor: secret way in to do new unexpected things

- Get around the normal barriers of behavior
- Ex: a way in to allow me to take complete control of your computer

Exploit - a way to use a vulnerability or backdoor that has been created

- Our exploit today: a **malicious payload**
 - A passcode and program
 - If it ever gets in memory, run my program regardless of what you want to do

Ethics, Business, Tech

Are there reasons to do this? Not to do this?

- No technical reason not to, it's easy to do!

Ethics, Business, Tech

Are there reasons to do this? Not to do this?

- No technical reason not to, it's easy to do!
- Ethical implications
- Business implications (lawsuits, PR, etc)

Ethics, Business, Tech

Are there reasons to do this? Not to do this?

- No technical reason not to, it's easy to do!
- Ethical implications
- Business implications (lawsuits, PR, etc)

Can we make a system where one bad actor can't break it?

Ethics, Business, Tech

Are there reasons to do this? Not to do this?

- No technical reason not to, it's easy to do!
- Ethical implications
- Business implications (lawsuits, PR, etc)

Can we make a system where one bad actor can't break it?

- Code reviews, double checks, verification systems, automated verification systems, ...

Why does this work?

Why?

Why does this work?

- **It's all bytes!**
- Everything we store in computers are bytes
- We store code and data in the same place: memory

It's all bytes

Memory, Code, Data... It's all bytes!

- **Enumerate** - pick the meaning for each possible byte
- **Adjacency** - store bigger values together (sequentially)
- **Pointers** - a value treated as address of thing we are interested in

Enumerate

Enumerate - pick the meaning for each possible byte

What is 8-bit 0x54?

Unsigned integer

Signed integer

Floating point w/ 4-bit exponent

ASCII

Bitvector sets

Our example ISA

eighty-four

positive eighty-four

twelve

capital letter T: T

The set {2, 3, 5}

Flip all bits of value in r1

Adjacency

Adjacency - store bigger values together (sequentially)

- An array: build bigger values out of many copies of the same type of small values
 - Store them next to each other in memory
 - Arithmetic to find any given value based on index

Adjacency

Adjacency - store bigger values together (sequentially)

- Records, structures, classes
 - Classes have fields! Store them adjacently
 - Know how to access (add offsets from base address)
 - If you tell me where object is, I can find fields

Pointers

Pointers - a value treated as address of thing we are interested in

- A value that really points to another value
- Easy to describe, hard to use properly
- We'll be talking about these a lot in this class!

Pointers

Pointers - a value treated as address of thing we are interested in

- Give us strange new powers (represent more complicated things), e.g.,
 - Variable-sized lists
 - Values that we don't know their type without looking
 - Dictionaries, maps

Programs Use These!

How do our programs use these?

- Enumerated icodes, numbers
- Adjacenty stored instructions (PC+1)
- Pointers of where to jump/goto (addresses in memory)

ToyISA Instructions

So far, only dealing with 8-bit machine!

icode	b	meaning
0		$rA = rB$
1		$rA \&= rB$
2		$rA += rB$
3	0	$rA = \sim rA$
	1	$rA = !rA$
	2	$rA = -rA$
	3	$rA = pc$
4		$rA = \text{read from memory at address } rB$
5		write rA to memory at address rB
6	0	$rA = \text{read from memory at } pc + 1$
	1	$rA \&= \text{read from memory at } pc + 1$
	2	$rA += \text{read from memory at } pc + 1$
	3	$rA = \text{read from memory at the address stored at } pc + 1$ For icode 6, increase pc by 2 at end of instruction
7		Compare rA as 8-bit 2's-complement to 0 if $rA \leq 0$ set $pc = rB$ else increment pc as normal

64-bit Machines

64-bit machine: The registers are 64-bits

- i.e., r0, but also PC

Important to have large values. Why?

64-bit Machines

64-bit machine: The registers are 64-bits

- i.e., r0, but also PC

Important to have large values. Why?

- Most important: PC and memory addresses
- How much memory could our 8-bit machine access?

64-bit Machines

64-bit machine: The registers are 64-bits

- i.e., r0, but also PC

Important to have large values. Why?

- Most important: PC and memory addresses
- How much memory could our 8-bit machine access? **256 Bytes**

64-bit Machines

64-bit machine: The registers are 64-bits

- i.e., r0, but also PC

Important to have large values. Why?

- Most important: PC and memory addresses
- How much memory could our 8-bit machine access? **256 Bytes**
- Late 70s - 16 bits:

64-bit Machines

64-bit machine: The registers are 64-bits

- i.e., r0, but also PC

Important to have large values. Why?

- Most important: PC and memory addresses
- How much memory could our 8-bit machine access? **256 Bytes**
- Late 70s - 16 bits: **65536 Bytes**

64-bit Machines

64-bit machine: The registers are 64-bits

- i.e., r0, but also PC

Important to have large values. Why?

- Most important: PC and memory addresses
- How much memory could our 8-bit machine access? **256 Bytes**
- Late 70s - 16 bits: **65536 Bytes**
- 80s - 32 bits:

64-bit Machines

64-bit machine: The registers are 64-bits

- i.e., r0, but also PC

Important to have large values. Why?

- Most important: PC and memory addresses
- How much memory could our 8-bit machine access? **256 Bytes**
- Late 70s - 16 bits: **65536 Bytes**
- 80s - 32 bits: \approx **4 billion bytes**

64-bit Machines

64-bit machine: The registers are 64-bits

- i.e., r0, but also PC

Important to have large values. Why?

- Most important: PC and memory addresses
- How much memory could our 8-bit machine access? **256 Bytes**
- Late 70s - 16 bits: **65536 Bytes**
- 80s - 32 bits: **≈ 4 billion bytes**
- Today's processors - 64 bits:

64-bit Machines

64-bit machine: The registers are 64-bits

- i.e., r0, but also PC

Important to have large values. Why?

- Most important: PC and memory addresses
- How much memory could our 8-bit machine access? **256 Bytes**
- Late 70s - 16 bits: **65536 Bytes**
- 80s - 32 bits: **≈ 4 billion bytes**
- Today's processors - 64 bits: **2^{64} addresses**

Aside: Powers of Two

Value	base-10	Short form	Pronounced
2^{10}	1024	Ki	Kilo
2^{20}	1,048,576	Mi	Mega
2^{30}	1,073,741,824	Gi	Giga
2^{40}	1,099,511,627,776	Ti	Tera
2^{50}	1,125,899,906,842,624	Pi	Peta
2^{60}	1,152,921,504,606,846,976	Ei	Exa

Example: 2^{27} bytes

Aside: Powers of Two

Value	base-10	Short form	Pronounced
2^{10}	1024	Ki	Kilo
2^{20}	1,048,576	Mi	Mega
2^{30}	1,073,741,824	Gi	Giga
2^{40}	1,099,511,627,776	Ti	Tera
2^{50}	1,125,899,906,842,624	Pi	Peta
2^{60}	1,152,921,504,606,846,976	Ei	Exa

Example: 2^{27} bytes = $2^7 \times 2^{20}$ bytes

Aside: Powers of Two

Value	base-10	Short form	Pronounced
2^{10}	1024	Ki	Kilo
2^{20}	1,048,576	Mi	Mega
2^{30}	1,073,741,824	Gi	Giga
2^{40}	1,099,511,627,776	Ti	Tera
2^{50}	1,125,899,906,842,624	Pi	Peta
2^{60}	1,152,921,504,606,846,976	Ei	Exa

Example: 2^{27} bytes = $2^7 \times 2^{20}$ bytes = 2^7 MiB = 128 MiB

64-bit Machines

How much can we address with 64-bits?

64-bit Machines

How much can we address with 64-bits?

- 16 EiB (2^{64} addresses = $2^4 \times 2^{60}$)

64-bit Machines

How much can we address with 64-bits?

- 16 EiB (2^{64} addresses = $2^4 \times 2^{60}$)
- But I only have 8 GiB of RAM

A Challenge

There is a disconnect:

- Registers: 64-bits values
- Memory: 8-bit values (i.e., **1 byte** values)
 - Each address addresses an 8-bit value in memory
 - Each address points to a 1-byte slot in memory

A Challenge

There is a disconnect:

- Registers: 64-bits values
- Memory: 8-bit values (i.e., **1 byte** values)
 - Each address addresses an 8-bit value in memory
 - Each address points to a 1-byte slot in memory
- How do we store a 64-bit value in an 8-bit spot?

Rules

Rules to break “big values” into bytes (memory)

1. Break it into bytes
2. Store them adjacently
3. Address of the overall value = smallest address of its bytes
4. Order the bytes
 - If parts are ordered (i.e., array), first goes in smallest address
 - Else, hardware implementation gets to pick (!!)
 - Little-endian
 - Big-endian

Ordering Values

Little-endian

- Store the low order part/byte first
- Most hardware today is little-endian

Big-endian

- Store the high order part/byte first

Example

Store [0x1234, 0x5678] at address 0xF00

Endianness

Why do we study endianness?

- It is **everywhere**
- It is a source of weird bugs
- Ex: It's likely your computer uses:
 - Little-endian from CPU to memory
 - Big-endian from CPU to network
 - File formats are roughly half and half

Moving up!

Assembly

General principle of all **assembly languages**

- Code (text, not binary!)
- 1 line of code = 1 machine instruction
- One-to-one reversible mapping between binary and assembly
 - We do not need to remember binary encodings!
 - A program will turn text to binary for us!

Assembly

Features of assembly

- Automatic addresses - use **labels** to keep track of addresses
 - Assembler will remember location of labels and use where appropriate
 - Labels will not exist in machine code
- Metadata - data about data
 - Data that helps turn assembly into code the machine can use
- As complicated as machine instructions
 - There are a lot of instructions, and it is one-to-one!

Assembly Languages

There are many assembly languages

- But, they're backed by hardware!
- Two big ones these days: x86-64 and ARM
 - You likely have machines that use one of these
- Others: RISC-V, MIPS, ...

We will focus on **x86-64**

x86-64

x86-64 has a weird and long history

- Expansion of the 8086 series (Intel)
 - 8086, 8286, 8386, 8486, x86
- AMD expanded it with AMD64
- Intel decide to use same build, but called it x86-64
- Backwards compatible with the 8086 series

x86-64

Two dialects - two ways to write the same thing

- Intel - likely using with Windows

```
mov QWORD PTR [rdx+0x227],rax
```

- AT&T - likely using with anything else

```
movq %rax,0x227(%rdx)
```

We will use AT&T dialect

AT&T x86-84 Assembly

instruction source, destination

- Instruction followed by 0 or more operands (arguments)
- 4 types of operands:
 - Number (immediate value): `$0x123`
 - Register: `%rax`
 - Address of memory: `(%rax)` or `24` or `labelname`
 - Value at an address in memory: `(%rax)` or `24` or `labelname`

AT&T x86-84 Assembly

`mylabelname:`

- Label - remember the address of next thing to use later

`.something something`

- Metadirective - extra information that is not code
- How the code works with other things (i.e., talk to OS)
- Ex: `.globl main`

`// we can have comments!`

Addressing Memory

2130(%rax, %rsp, 8)

- Address can have up to 4 parts: 2 numbers, 2 registers
- Combines as: $2130 + \%rax + (\%rsp * 8)$
- Common usage from this example:
 - `rax` - address of an object in memory
 - 2130 - offset of an array into the object
 - `rsp` - index into the array
 - 8 - size of the values in the array
- Don't need all parts: `(%rax)` or `4(%rax)`
- This is all one operand (one memory address)

hello.s example

Registers

rax is a 64-bit register

Instructions

Instructions have different versions depending on number of bits to use

- `movq` - 64-bit move
 - q = quad word
- `movl` - 32-bit move
 - l = long
- There are encodings for shorter things, but we will mostly see 32- and 64-bit

More powerful than our ISA

Instructions can move/operate between memory and register

- `movq %rax, %rcx` - register to register
 - Remember our icode 0
- `movq (%rax), %rcx` - memory to register
 - Remember our icode 3
- `movq %rax, (%rcx)` - register to memory
 - Remember our icode 4
- `movq $21, %rax` - Immediate to register
 - Remember our icode 6 (b=0)

Note: at most one memory address per instruction

Other Instructions

Other instructions work the same way

- `addq %rax, %rcx` — `rcx += rax`
- `subq (%rbx), %rax` — `rax -= M[rbx]`
- `xor`, `and`, and others work the same way!
- Assembly has virtually no 3-argument instructions
 - All will be modifying something (i.e., `+=`, `&=`, ...)

Load Effective Address

Load effective address: `leaq 4(%rcx), %rax`

- Performs memory address calculation
- Stores address, not value at the address in memory

Jumps

`jmp foo`

- Unconditional jump to `foo`
- `foo` is a label or memory address
- Need `jmp*` to use register value

Conditional jumps

- `jl, jle, je, jne, jg, jge, ja, jb, js, jo`

Unlike our Toy ISA, these do not compare given register to 0

Jumps

Condition codes - 4 1-bit registers set by every math operation, `cmp`, and `test`

- Result for the operation compared to 0 (if no overflow)
- Example:

```
addq $-5, %rax
// ...code that doesn't set condition codes...
je foo
```

 - Sets condition codes from doing math (subtract 5 from `rax`)
 - Tells whether result was positive, negative, 0, if there was overflow, ...
 - Then jump if the result of operation should have been = 0

Jumps: compare...

```
cmpq %rax, %rdx
```

- Compare checks result of $\text{rdx} - \text{rax}$ and sets condition codes
- How $\text{rdx} - \text{rax}$ compares with 0
- Be aware of ordering!
 - if rax is bigger, sets $<$ flag
 - if rdx is bigger, sets $>$ flag

Jumps: ... and test

```
testq %rax, %rdx
```

- Sets the condition codes based on rdx & rax
- Less common

Neither save their result, just set condition codes!

Function Calls: Calling Conventions

`callq myfun`

- Push return address, then jump to `myfun`
- Convention: Store arguments in registers and stack before call
 - First 6 arguments (in order): `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
 - If more arguments, pushed onto stack (last to first)

`retq`

- Pop return address from stack and jump back
- Convention: store return value in `rax` before calling `retq`

This is similar to our Toy ISA's function calls in homework 4

Debugger

Debugger - step through code!

- You will be using this for lab 7
- Experience seeing results of these instructions step-by-step
- **Please read the x86-64 summary reading before lab!**