Software Reverse Engineering

COMP 293A | Spring 2022 | University of the Pacific | Jeff Shafer

x86 and x64 Assembly Code

Malware

Cryptocurrency Mining

- *Not exactly "malware", but unwelcome*

- Generic scenario
  1. Websites commonly pull in JavaScript from third parties (e.g. ad networks, fonts, menu APIs, …)
  2. Third-party scripting site is hacked and malware is introduced
  3. Result: Many websites are now distributing malware to their users *that weren't directly hacked themselves*

- JavaScript is running in 100% unbreakable sandbox (we hope…) but can still cause *mischief* on user's PC

Cryptocurrency Mining

- BrowseAloud web screen reader service
  - https://www.texthelp.com

- BrowseAloud (`ba.js`) JavaScript altered to include invocation of Monero cryptocurrency miner
  - https://coinhive.com/

Cryptocurrency Mining

- 4000-5000 websites affected
  - cuny.edu, uscourts,gov, …
  - Many .gov sites (US and international) due to accessibility requirements
- Monero miner is mostly harmless – just spikes your CPU to 100% if your ad blocker didn’t kill it outright
- Could have easily been used for ad popups, password stealing, or fake updates to install malware

Prevention

Before:

```
<script src="//www.browsealoud.com/plus/scripts/ba.js"
type="text/javascript"></script>
```

After (Using Subresource Integrity Attribute)

```
<script src="//www.browsealoud.com/plus/scripts/ba.js"
integrity="sha256-
Abhisa/nS9WMne/YX+dqiFINl+JiE15MCWvASJvVtIk="
crossorigin="anonymous"></script>
```

Challenge – What if third-party changes their script?

Cryptocurrency Mining

- https://scotthelme.co.uk/protect-site-from-cryptojacking-csp-sri/

x86 Assembly

Roadmap

Win32 API → x86 Assembly → Disassemblers

Instruction Set Architecture

- **Instruction Set Architecture (ISA)** is the interface between hardware and software
  - Specifies the format of processor instructions
  - Specifies the format of memory addresses (and addressing modes)
  - Specifies the primitive operations the processor can perform
- **ISA is the “contract” between the *hardware designer* and the assembly-level *programmer***
  - Documented in a manual

Learning Objective

- **Full knowledge of x86 and x64 ISA**
- Intel 64 and IA-32 Architectures Software Developers Manual
  - https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
  - Combined PDF (Nov 2020) is **5066 pages long!**

(Real) Learning Objectives

- Recognize assembly code structures *as produced by a compiler*
- Trace through a subroutine or small collection of subroutines and decipher code operations and high-level objective of code

---

- Writing assembly programs from scratch is NOT a learning objective
  - *But inline assembly of a function or two might be valuable*

History

- The x86 lineage traces back to an 8-bit architecture
  - 8080 – 8 bits
  - 8086, 8088 – 16 bits
  - 80286 – 16 bits
  - 80386 – 32 bits
  - 80486 – 32 bits
  - ….
- Many questions of “why is the design like this?” can be answered with “backwards compatibility”

Instructions

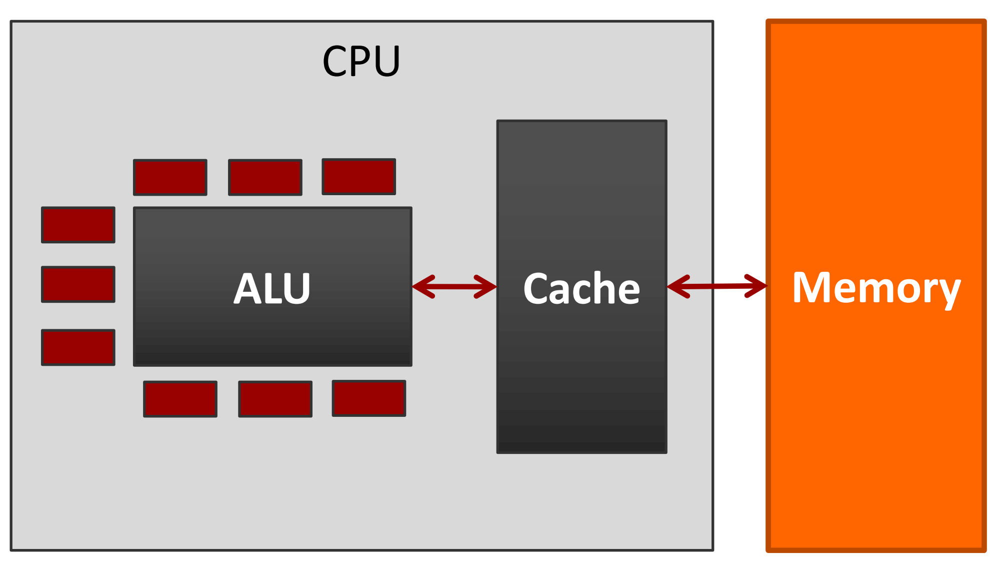
- Data manipulation
  - ADD, SUB, SHR, AND, OR, XOR, …
- Data transfer
  - PUSH, POP, MOV, XCHG, LEA, …
- Branching and conditionals
  - JMP, CALL, RET, CMP, …

Instructions

- Instruction format
  - Op-code (think “operation”): e.g. ADD
  - Operand (think “data”): e.g. 2+10
- Instructions can have 0, 1, or 2 operands
  - Immediate value
  - Register
  - Memory location

Data Sources

Where can we explicitly place data in assembly programming?



1. **Registers**
   - On the CPU itself
   - Very close to ALU
   - Tiny
   - Access time: 1 cycle
2. **Memory**
   - Off-chip
   - Large
   - Access time: 100+ cycles

Examples

- **Add 10 to a register**

```
add eax, 10
```

- **Add 10 to some location in memory**

```
add BYTE PTR [var], 10
```

Congratulations!

You’re now an x86 assembly programming expert!

Registers

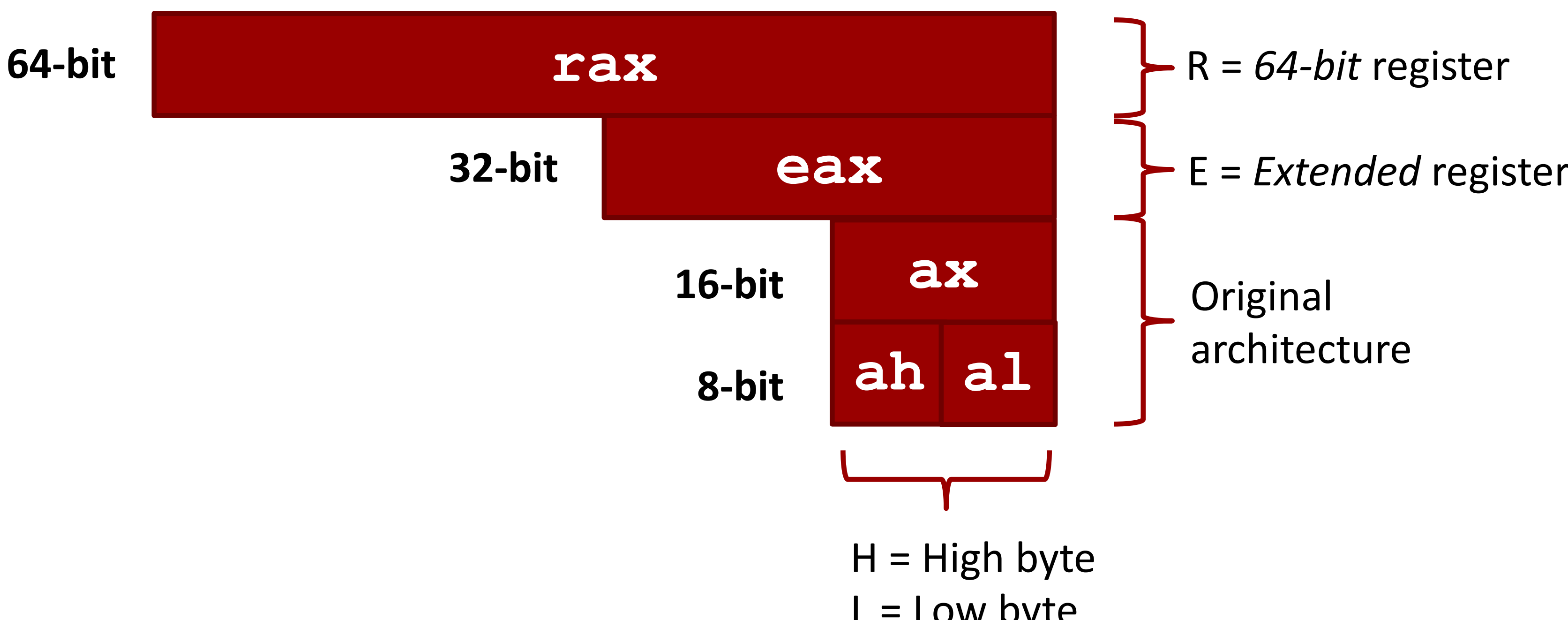
- Original general purpose registers in 16-bit Intel architecture
  - Accumulator Register (**`ax`**)
  - Base Register (**`bx`**)
  - Counter Register (**`cx`**)
  - Data Register (**`dx`**)
  - Stack Pointer Register (**`sp`**)
  - Stack Base Pointer Register (**`bp`**)
  - Source Index Register (**`si`**)
  - Destination Index Register (**`di`**)

A, B, C, D

Register Naming Convention

Example for "A" Register

This is *one* physical register, but the name varies depending on the bits accessed



(Typical) Register Uses

- Accumulator Register (`ax`)
  - Addition, Multiplication, Return Values
- Base Register (`bx`)
- Counter Register (`cx`)
  - Counter
- Data Register (`dx`)
- Stack Pointer Register (`sp`)
  - The stack!
- Stack Base Pointer Register (`bp`)
  - Function arguments and local variables
- Source Index Register (`si`) / Destination Index Register (`di`)
  - Memory transfer instructions

Special Purpose Registers

- `eip`
  - Instruction Pointer
  - Points to next instruction to execute
- `eflags`
  - Stores outcome of calculation
  - Controls CPU operation
- Segment Registers
  - `cs` – Code Segment
  - `ds` – Data Segment
  - `ss` – Stack Segment

Operands (recap)

- Instruction operands can be
  - Immediate value
    - "Add 10 to the value in **`eax`**"
  - Register
    - "Add the value in **`ebx`** to the value in **`eax`**"
  - Memory location
    - "Go to memory *somewhere* and retrieve a value. Add it to the value in **`eax`**"

Memory Addressing

- How to we determine what memory address to access? (a.k.a. determine the *effective address*)
- Direct mode (address is in instruction)
  - `mov edx, [0x01020304]`
- Register indirect mode (address is in register)
  - `mov edx, [eax]`
- Base + Displacement
  - `mov edx, [ebp + 0x10]`
  - `mov edx, [eax + ebx * 8]`
    # Useful for arrays! Base + Offset*Size

Functions

- General rules
- Functions must preserve all registers
  - Except for `eax`, `ecx`, and `edx`
  - Except for `esp` (updated according to calling convention)
- Return value of function is saved in `eax` (32 bits) or `edx:eax` (64 bits)

Calling Conventions

- How do I pass data into functions?
- How is data returned from functions?
- Answer will vary depending on compiler

Calling Conventions

Calling Conventions

- `__cdecl`
  - **Most common convention**
  - Function arguments passed on the stack (pushed right to left)
  - *Calleer* cleans the stack after use
- `__stdcall`
  - **Convention used in Win32 API**
    (which means you will see both conventions in malware!)
  - Function arguments passed on the stack (pushed right to left)
  - *Callee* cleans the stack

https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x86-architecture

Calling Conventions

- **`thiscall`**
  - Used in C++ code
  - Function arguments passed on the stack (pushed right to left)
  - Microsoft compilers
    - "this" pointer passed in **`ecx`** register
    - *Callee* cleans the stack
  - GNU compilers
    - "this" pointer pushed onto stack last
    - *Caller* cleans the stack

https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x86-architecture

Calling Conventions

- `__fastcall`
  - Accelerated!
  - First two arguments are passed in `ecx` and `edx`
  - Remaining arguments are passed on the stack (pushed right to left)
  - *Callee* cleans the stack

https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x86-architecture

Stack

- The stack grows DOWN (from high mem to low mem)
  - **Positive offset** from base pointer `ebp` = Function argument
  - **Negative offset** from base pointer `ebp` = Local variable of function
- Example of how the stack works
  - http://www.csr-group.com/resources/Stack-graphical-handout.pdf

x64 Assembly

More Memory!

- x64 ISA is a 64-bit architecture with legacy 32-bit mode that is backwards compatible with x86
- Key change – memory addresses are now 64 bits wide
  - 2^{32} = ~4GB of addressable memory
  - 2^{64} = ~ 16 exabytes of addressable memory
- Implicit in this is that all registers are now 64 bits wide

More Registers!

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

32-bit Registers

Wider 64-bit Registers

New 64-bit Registers

https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture

x64 Calling Convention

- Only one calling convention! *(Yay simplicity)*
- Arguments
  - Integers: `rcx`, `rdx`, `r8`, `r9`
  - Floating-point: `xmm0-xmm3` (SSE registers)
  - Additional arguments passed on stack
- Return value
  - Integer: `rax`
  - Floating-point: `xmm0`

https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture

Upcoming Events

- Tuesday March 10th
  - **Project 1 Proposal Due**