SP2024 Week 03 • 2024-02-08

ARM Assembly

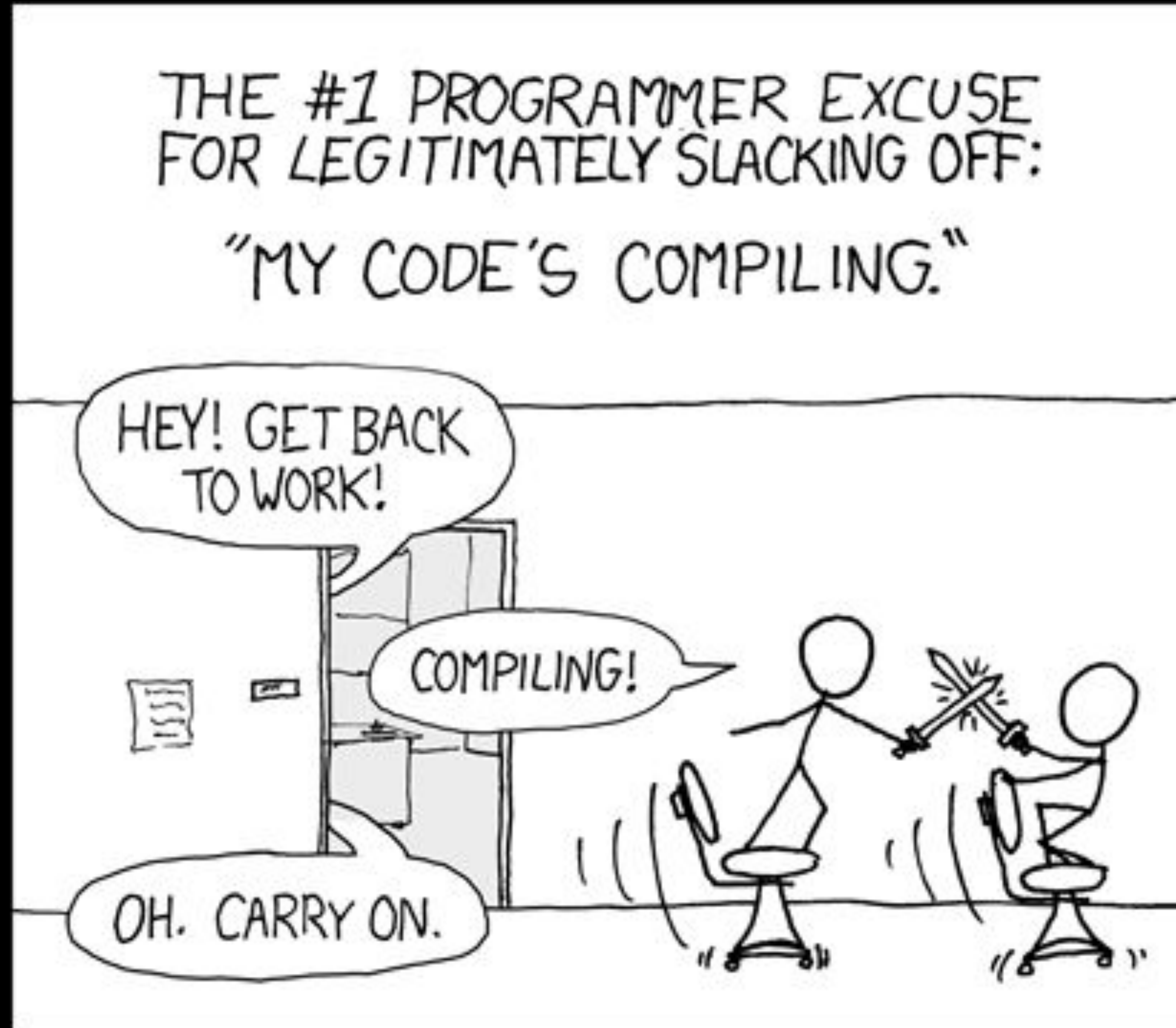
Sam Ruggerio and Richard Liu

Announcements

- Next Weekend, Feb 16 @ 10pm: LACTF Starts
  - We will be in person again! Free pizza/food for participants!

ctf.sigpwny.com

sigpwny{costs_an_arm_and_a_leg}



A Recap on Assembly

- A human readable abstraction over machine code

```
48 05 DE C0 37 13 -> add rax, 0x1337c0de
```

- Low Level Languages (e.g. C/C++) are compiled to machine code, which have corresponding assembly instructions

- The processor maintains the Stack, Registers, and Instruction Memory as the main structures to execute programs.

- See Week 5 from FA23 for the first (x86) assembly meeting!

Overview

- **Prerequisite: x86-64 Assembly**
- Background
- ARM Assembly
- Running ARM

Background

What is an ISA?

- Instruction Set Architecture is the specification for what instructions a processor needs to support and how it manipulates memory
- x86/x64, ARM, and RISC-V are example ISAs
- This means you can design your processor however you want, as long as you meet the ISA spec, you've made a(n) [ISA] processor.

RISC vs CISC

- RISC stands for Reduced Instruction Set Computer
- Under most cases in RISC, 1 instruction takes 1 cycle, separate instructions for memory load/store, fixed width instruction length. ARM, MIPS, RISC-V are all RISC ISAs

- CISC is Complex Instruction Set Computer
- Anything *not* RISC. Instructions can take multiple cycles and can directly operate on memory. x86 is a CISC ISA.

What is ARM™

- Founded in 1990 as Advanced RISC Machines
- The most popular RISC format that exists, used in Apple Silicon and Mobile processors, as well as many embedded devices.
- ARM's ISA is licenced, meaning if you want to produce chips, royalties are paid to ARM for the chips produced.
- Heavily uses Performance/Efficiency Core splits in CPUs

ARM vs x86-64

- Fixed Width Instructions
- Explicit Memory Load/Store
- 1 dest, 2 source vs 1 dest & 1 source
- Weak Memory Model (FENCE)
- No segmented memory (canaries easily readable)

What Runs ARM?

- Many IoT devices
- All smartphones
- Apple Macbooks post 2020
- Supercomputers

Versioning

- two sets of version numbers
  - Architecture version (i.e. ARMv7, ARMv8, etc.)
  - Core version (ARM11, Cortex-M3, Cortex-A7)
    - A: application
    - R: realtime
    - M: microcontroller
- AArch64
  - 64-bit register (31 general purpose), 32-bit instructions
- AArch32
  - 32-bit registers (15 general purpose), 32-/16-bit instructions

ARM Assembly

Register Convention

ARM	Description	x86
R0	General Purpose	EAX
R1-R5	General Purpose	EBX, ECX, EDX, ESI, EDI
R6-R10	General Purpose	–
R11 (FP)	Frame Pointer	EBP
R12	Intra Procedural Call	–
R13 (SP)	Stack Pointer	ESP
R14 (LR)	Link Register	–
R15 (PC)	<- Program Counter / Instruction Pointer ->	EIP
CPSR	Current Program State Register/Flags	EFLAGS

Registers	Use	Comment
R0	ARG 1	Used to pass arguments to subroutines. Can use them as scratch registers. Caller saved.
R1	ARG 2	
R2	ARG 3	
R3	ARG 4	
R4	VAR 1	Used as register based variables. Subroutine must preserve their data. Callee saved. Must return intact after call.
R5	VAR 2	
R6	VAR 3	
R7	VAR 4	
R8	VAR 5	
R9	VAR 6	Variable or static base
R10	VAR 7 / SB	Variable or stack limit
R11	VAR 8 / FP	Variable or frame pointer
R12	VAR 9 / IP	Variable or new static base for interlinked calls
R13	SP	Stack pointer
R14	LR	Link back to calling routine.
PC	PC	Program counter

AArch64

- x0, …, x30 64-bit
- w0, …, w30 32-bit

General-Purpose Registers

X0 – X30 64-bit; W0 – W30 32-bit

X0 – **X7** parameter/result registers

X8 Indirect result location register

X9 – **X15** temporary registers

X16 – IP0 intra-procedure-call temp register

X17 – IP1 intra-procedure-call temp register

X18 – PR Platform Register

X19 – **X28** callee-saved registers

X29 – FP Frame Pointer

X30 – LR Link Register

Example

- calling convention and stack frames
- (see next slide)

```
0000000100003f50 <_nest>:
100003f50: d65f03c0      ret

0000000100003f54 <_foo>:
100003f54: d10083ff      sub sp, sp, #32
100003f58: a9017bfd      stp x29, x30, [sp, #16]
100003f5c: 910043fd      add x29, sp, #16
100003f60: b81fc3a0      stur   w0, [x29, #-4]
100003f64: 97fffffb      bl  0x100003f50 <_nest>
100003f68: b85fc3a8      ldur   w8, [x29, #-4]
100003f6c: 11000900      add w0, w8, #2
100003f70: a9417bfd      ldp x29, x30, [sp, #16]
100003f74: 910083ff      add sp, sp, #32
100003f78: d65f03c0      ret

0000000100003f7c <_main>:
100003f7c: d10083ff      sub sp, sp, #32
100003f80: a9017bfd      stp x29, x30, [sp, #16]
100003f84: 910043fd      add x29, sp, #16
100003f88: b81fc3bf      stur   wzr, [x29, #-4]
100003f8c: 52800060      mov w0, #3
100003f90: 97fffff1      bl  0x100003f54 <_foo>
100003f94: a9417bfd      ldp x29, x30, [sp, #16]
100003f98: 910083ff      add sp, sp, #32
100003f9c: d65f03c0      ret
```

Stores pair (fp and lr)

Save argument before call

Compute value

Stack cleanup

```c
void nest() {}

int foo(int bar) {
  nest();
  return bar + 2;
}

int main() {
  return foo(3);
}
```

Addressing

```
ldr r2, [r0]      // r2 = [r0]
str r2, [r1, #2]  // [r1 + 2] = r2 (offset)
str r2, [r1, #4]! // r1 += 4, [r1] = r2, (pre-index)
ldr r3, [r1], #4  // r3 = [r1], r1 += 4 (post index)

str r2, [r1, r3, LSL#2]! // r1 += r3 << 2; [r1] = r2
```

Load/store multiple

```
push {r0, r1}        // <- these are equivalent
stmdb sp!, {r0, r1} // <-
```

- suffixes: `-ia`, `-ib`, `-da`, `-db`
    - determine order (**i**ncrease/**d**ecrease **a**fter/**b**efore)
- with `!`, write result pointer back to register
- `push` = `stmdb`, `pop` = `ldmia`

Conditional Execution

- While you can still branch with ARM like x86 jCC:

```
cmp r1, r2
beq loop
```

- ARM supports conditional execution:

```
cmp r1, r2
addeq r3,r4
movlt r1,r10
```

Thumb Mode

- 16 bit instruction width
- No conditional execution
- Thumb-2 (introduced 2003) mixed 16- and 32-bit instructions
- use BX/BLX and set LSB of addr to 1 to switch to Thumb
- Makes reverse engineering harder

ARM Security Features

- Pointer Authentication (PAC)
  - Pointers don't use the full 64 bits in memory space
  - Use extra space for authentication values to prevent ROP/BOF
  - Low overhead: Call `PACIASP` at the start, `AUTIASP before return`
- Branch Target Identification (BTI)
  - Branching should only lead to specific points in code.
  - Make a NOP instruction: `PACBTI/BXAUT`
  - If a branch does not go here, crash out.

Running ARM

Running ARM with qemu

- You'll need to get qemu and helper packages
  - `qemu-user qemu-user-static gcc-aarch64-linux-gnu binutils-aarch64-linux-gnu binutils-aarch64-linux-gnu-dbg build-essential`
- If you have a STATIC binary:
  - `qemu-aarch64-static ./executable`
- If you have a DYNAMIC binary:
  - `qemu-aarch64 -L /usr/aarch64-linux-gnu ./executable`

- If you want to build a binary:
  - `aarch64-linux-gnu-gcc -o executable file.c`

Debugging ARM with gdb-multiarch

- Install the `gdb-multiarch` package. This should still be compatible with gdb-peda or GEF.
- Run qemu in debug mode:
  - `qemu-aarch64 -g 9001 -L /usr/aarch64-linux-gnu ./exe`
- Run `gdb-multiarch` and connect to the process:
  - `gdb-multiarch ./exe`
  - `gdb> target remote localhost:9001`

Resources

- https://azeria-labs.com/writing-arm-assembly-part-1/
    - 7-part ARM series

Next Meetings

2024-02-11 • This Sunday

- PWN III: Heap Exploitation with Sam
- Learn modern PWN techniques!

2024-02-15 • Next Thursday

- PWN IV: ROP with Akhil
- Learn how to complete PWN exploit chains and achieve RCE!

2024-02-16 • Next Weekend

- LACTF
- UCLA's Major CTF Event! All are welcome!

ctf.sigpwny.com

sigpwny{costs_an_arm_and_a_leg}

Meeting content can be found at sigpwny.com/meetings.