CY-4973/7790

Kernel Security: how2rootkit

What is ARM?

- stands for **Advanced RISC Machine(s)**.
- RISC stands for reduced instruction set computer
- ARM is a company, an ambiguous term for Assembly Languages and architectures

ARM Holdings

- A British(?) semiconductor and software company.
- Specializes in designing RISC architectures.
- Does not manufacture its own chips.
- Licenses its designs to other companies (e.g., Qualcomm, Apple, Samsung).
- ARM chips power billions of devices, from smartphones to IoT devices to ssd controllers to enterprise servers.
- Majority owned by Soft Bank (questionable call by the UK)

RISC vs CISC

Feature	RISC	CISC
Instruction Set	Simple, small	Complex, large
Execution	Single cycle per instruction	Multi-cycle per instruction
Power Efficiency	High	Lower
Performance	Optimized for pipelining	Slower due to complexity
Program Size	Larger	Smaller
Examples	ARM, RISC-V	x86, Intel 8086

(Note x86 is kind of like a sugared RISC at this point)

ArmV8 profiles

1. **A (Application)**:
    - Supports rich operating systems.
    - Focused on performance and complex applications.
    - Examples: smartphones, tablets, servers.
2. **R (Real-Time)**:
    - Predictable and deterministic performance.
    - Common in automotive and industrial systems.
3. **M (Microcontroller)**:
    - Low power, low cost.
    - Designed for IoT and embedded systems.
    - probably in your SSD controller

Microarchitectures

- Specific implementations of the ARM architecture.
- Examples: Cortex-A series, Cortex-R series, Cortex-M series.
- Each microarchitecture optimizes for specific use cases:
    - Performance, power efficiency, or a balance of both.

ARMv8-a

- This class focuses on ARMv8-a
- RISC arch built by ARM holdings
- `aarch64` -> A64: 64 bit execution state
- `aarch32` -> A32: 32 bit execution state

A64 (AArch64 mode) Assembly Crash Course

- `aarch64`: 64 bit execution state for ARMV8-a
- `A64` Assembly language/instruction set for `aarch64`
- 32-bit assembly instructions (4 byte word)
- Uses 31 64 bit general-purpose registers: `x0`–`x30`, plus `sp` (stack pointer) and `pc` (program counter).
  - `w0-w30` are the lower 32 bits $b_0, \ldots, b_{31}$
- Uses 31 128 bit floating point registers `q0-q31`
  - `b,h,w,d,q`: ->
  - `byte`, `hald-word`, `word`, `double-word`,`quad-word`
- Some registers have special usage/convention:
- The goal here is to learn **just enough** to be useful
- technically supports little/big endian but I have only ever seen little endian

Special General Purpose registers (A64)

- Here "special" means there exists a convention
- `x0` often used for function return values form a subroutine.
- `x0`–`x7` typically used for arguments in many Linux ABIs.
- x8 holds syscall number
- `x29`: (sometimes aliased as `fp`) Frame Pointer (optional)
- `x30`: Link Register often holds return address on subroutine calls

Comparison to x86_64

Aarch64 Register	Purpose/Usage	x86_64 Equivalent	Explanation
`x0`	Return values from a subroutine	`rax`	Both are used to store return values for functions.
`x0–x7`	Arguments in many Linux ABIs	`rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`	x86_64 uses a similar approach with a set of registers for function arguments.
`x8`	Syscall number	`rax`	On x86_64, the syscall number is passed through `rax` before invoking syscall.
`x29` (aliased `fp`)	Frame pointer (optional; helps manage stack frames)	`rbp`	Both are used as a frame pointer to access local variables and manage the stack.
`x30`	Link register (holds return address for subroutine calls)	`ret` stack mechanism	In x86_64, the return address is stored on the stack instead of in a dedicated register.

A64 vs x86_64 notes

- Fixed-length 32-bit instructions simplify decoding and pipelining.
- Implements a **load/store architecture**, meaning it cannot perform operations directly on memory. - Aarch64 cannot directly operate on data unless it is stored in a register.
- All data must first be loaded into registers for processing and then stored back into memory.
    - For example, in `memcpy`, data is copied in chunks by loading blocks into registers and then storing them to the destination address.

Other important registers

- `xzr`: zero register. ignores writes.
- `sp`: stack pointer
- `pc`: instruction pointer/program counter. Can't directly read/modify like x_i
  - `bracnh`: used to modify `pc`
- `lr`: alias for `x30`.
  - There is no `call` instruction.
  - There is no `ret` instruction
- various system registers (exception levels, mmu ...etc)
- this is more trivia than anything but "`x31`" is either zero register or stack pointer
  - `sp` and `xzr`/`wzr` are **architecturally distinct**.
  - in *some instruction encodings* the same field in the instruction word can represent either `sp` or `xzr` depending on context.

Refresher: Common Registers (AArch64)

- **x0–x7**:
- **x8**:
- **x9–x15**: temporary registers.
- **x16–x17**: intra-procedure call scratch regs.
- **x19–x28**: callee-saved registers.
- **x29**:
- **x30**:
- **sp**: stack pointer.
- **pc**: program counter (auto-updated cant be directly used).

Refresher: Common Registers (AArch64)

- **x0–x7**:
    - used to pass function arguments, return values.
- **x8**:
- **x9–x15**: temporary registers.
- **x16–x17**: intra-procedure call scratch regs.
- **x19–x28**: callee-saved registers.
- **x29**:
- **x30**:
- **sp**: stack pointer.
- **pc**: program counter (auto-updated cant be directly used).

Refresher: Common Registers (AArch64)

- **x0–x7**:
  - used to pass function arguments, return values.
- **x8**:
  - indirect syscall param or special usage.
- **x9–x15**: temporary registers.
- **x16–x17**: intra-procedure call scratch regs.
- **x19–x28**: callee-saved registers.
- **x29**:
- **x30**:
- **sp**: stack pointer.
- **pc**: program counter (auto-updated cant be directly used).

Refresher: Common Registers (AArch64)

- **x0–x7**:
  - used to pass function arguments, return values.
- **x8**:
  - indirect syscall param or special usage.
- **x9–x15**: temporary registers.
- **x16–x17**: intra-procedure call scratch regs.
- **x19–x28**: callee-saved registers.
- **x29**:
  - optional frame pointer (often used with fp alias)
- **x30**:

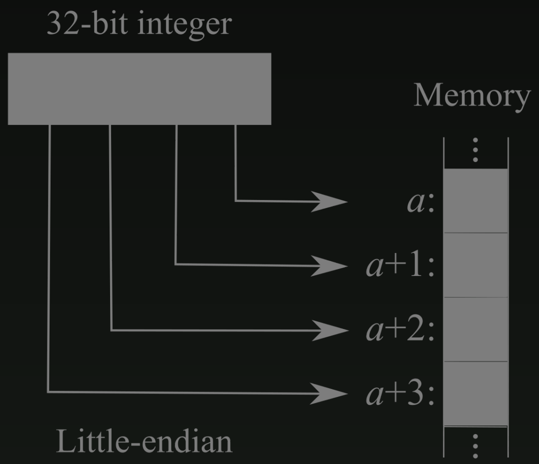
- **sp**: stack pointer.
- **pc**: program counter (auto-updated cant be directly used).

Refresher: Common Registers (AArch64)

- **x0–x7**:
    - used to pass function arguments, return values.
- **x8**:
    - indirect syscall param or special usage.
- **x9–x15**: temporary registers.
- **x16–x17**: intra-procedure call scratch regs.
- **x19–x28**: callee-saved registers.
- **x29**:
    - optional frame pointer (often used with fp alias)
- **x30**:
    - link register (return address).
- **sp**: stack pointer.
- **pc**: program counter (auto-updated cant be directly used).

Operand Sizes/ Types

- Byte: 8 bits
- Halfword: 16 bits
- Word: 32 bits
- Doubleword: 64 bits
- Quadword: 128 bits
- ARMv8-a is (almost always) **little-endian** *.
- In assembly, you'll see mnemonics for loading/storing different widths (e.g. `ldrb`, `ldrh`, `ldr`, `ldur`).



Sanity Check

What does this print? #exercise

```c
#include <stdio.h>

int main() {
  int x = 0xdeadbeef;
  unsigned char *y = (unsigned char *)&x;
  for (int i = 0; i < 4; i++) {
    unsigned char c = y[i] & 0xff;
    printf("%x ", c);
  }
  printf("\n");
}
```

Sanity Check

What does this print? #exercise

```c
#include <stdio.h>

int main() {
  int x = 0xdeadbeef;
  unsigned char *y = (unsigned char *)&x;
  for (int i = 0; i < 4; i++) {
    unsigned char c = y[i] & 0xff;
    printf("%x ", c);
  }
  printf("\n");
}
```

- ef be ad de

Basic Arithmetic (AArch64)

- **ADD** / **SUB**: Integer addition/subtraction
  - Example: `ADD x0, x1, x2` (x0 ← x1 + x2)
  - Immediate form: `ADD w3, w4, #10`
- **ADDS** / **SUBS**: Same as above, but **sets condition flags** (N, Z, C, V)
  - Useful for branching on results
- **MUL**: Multiply the lower 64 bits
  - `MUL x0, x1, x2`
- **SMULH** / **UMULH**: Signed / unsigned high 64-bit multiply
  - If the product exceeds 64 bits, high part stored in the destination

Load/Store Operations

- **LDR** / **STR**: Primary load/store instructions
  - `LDR x0, [x1]` → x0 ← *(uint64_t)*x1
  - `STR x2, [x3, #16]` → *(uint64_t)*(x3+16) ← x2
- **LDRB** / **STRB**: For 8-bit (byte) load/store
  - Similarly `LDRH`, `LDRW` for halfword/word
- **LDP** / **STP**: Load/Store **pairs** of registers
  - Often used to save/restore register pairs in function prolog/epilog
- Offsets can be
  - immediate (`[x1, #4]`)
  - post-/pre-indexed (`[x1], #4`, `[x1, #4]!`)
- READ: https://developer.arm.com/documentation/102374/0102/Loads-and-stores---addressing

Immediate Offset

- The address is computed by adding a constant offset directly to the base register.
- the base register is not modified.

Syntax:

```
ldr w0, [x1, #4]  // Load from address (x1 + 4)
str w0, [x1, #-8] // Store to address (x1 - 8)
```

- address = `x1 + 4 (or x1 - 8)`.
- Perform the memory access.
- Base register x1 remains unchanged.

Post-indexed Offset

- The address is computed using the base register.
- After the access, the offset is added to the base register.

Syntax:

```
ldr w0, [x1], #4   // Load from x1, then x1 += 4
str w0, [x1], #-8  // Store to x1, then x1 -= 8
```

- Use the address in the base register for memory access.
- Update the base register (`x1 += 4 or x1 -= 8`).

Pre-Index Offset

- The offset is added to the base register before the memory access.
- The base register is updated with the new address.

Syntax:

```
ldr w0, [x1, #4]!  // x1 += 4, then load from x1
str w0, [x1, #-8]! // x1 -= 8, then store to x1
```

- Update the base register (x1 += 4 or x1 -= 8).
- Use the updated value of the base register for memory access.

Conditional Flags

- Condition Codes: Special flags set by the processor to indicate the result of an operation.
- Enables conditional execution of instructions based on previous operations.
- N (Negative): Set if the result of an operation is negative.
- Z (Zero): Set if the result is zero.
- C (Carry): Set if an operation results in a carry out or borrow.
- V (Overflow): Set if an operation causes a signed overflow.

Insturctions-> Conditional Flags

- Comparison Instructions:
  - CMP: Compares two values by subtracting one from the other; updates N, Z, C, V flags.
  - CMN: Compares two values by adding them; updates N, Z, C, V flags.
- Logical Instructions:
  - TST: Performs an AND operation; updates N and Z flags.
  - TEQ: Performs an XOR operation; updates N and Z flags.

Basic Control FLow

- **B** : Unconditional branch
- **BL** : Branch and link (subroutine call)
  - Automatically saves the return address in **x30** (Link Register)
- **RET**: Return from subroutine
  - Not a real instruction :)
  - `blr x30`
  - Jumps to the address in x30
- **CBZ** / **CBNZ**: Compare register to zero and branch
  - `CBZ x0, label` → if x0 == 0, branch
- Condition-based branches (after ADDS, SUBS):
  - **B.EQ**, **B.NE**, **B.GT**, **B.LT**, etc.

More Resources

- https://developer.arm.com/documentation/102374/0102
- https://web.archive.org/web/20240829145252/https://modexp.wordpress.com
  assembly/
- https://mariokartwii.com/armv8/

How do we learn A64 in this class?

- `objdump -d a.out`
- Writing little bootstraps

Baby's first A64

```
.section .text
.global _start

_start:
    mov x8, #93           // Syscall number for exit (93)
    mov x0, #0            // Exit code (0)
    svc #0                // Make the syscall
```

Baby's first A64

```
.equ sys_exit, 93


.section .text
.global _start

_start:
    mov x8, sys_exit          // Syscall number for exit (93)
    mov x0, #0            // Exit code (0)
    svc #0                // Make the syscall
```

Baby's first A64

```
// Define sys_exit to be 93
.equ sys_exit, 93


// Macro to perform the exit syscall
// note that code is an arg
.macro exit code
    mov x8, sys_exit     // Load syscall number (94 for exit)
    mov x0, \code        // Load the exit code into x0
    svc #0               // Make the syscall
.endm

.section .text
.global _start


_start:
    exit 0               // Call the macro with exit code 0
```

Baby's first A64

```
// common.S
// Define syscall numbers
.equ sys_exit, 93 // Syscall number for exit

// Macro to perform the exit syscall
.macro exit code
    mov x8, sys_exit     // Load syscall number
    mov x0, \code        // Load the exit code into x0
    svc #0               // Make the syscall
.endm
```

```
.include "common.S" // Include the macros file

.section .text
.global _start


_start:
    exit 0
```

hello world

- write: fd=1, addr_str, str_size

```
.section .rodata
hello:
  .asciz "hello world!\n"

.section .text
.global _start
_start:
  // 1 is stdout
  mov x0, #1
  ldr x1, =hello
  mov x2, #14
  mov x8, #64
  svc 0
  mov x0, 0
  mov x8, #93
  svc 0
```

hello world

```
// Macros
.equ STDOUT_FD, 1
.equ SYS_WRITE, 64
.equ  SYS_EXIT, 93


.section .rodata
hello:
  .asciz "hello world!\n"

.section .text
.global _start
_start:
  // 1 is stdout
  mov x0, STDOUT_FD
  ldr x1, =hello
  mov x2, #14
  mov x8, SYS_WRITE
  svc 0
  mov x0, 0
  mov x8, SYS_EXIT
  svc 0
```

hello world

```
// Macros
.equ STDOUT_FD, 1
.equ SYS_WRITE, 64
.equ  SYS_EXIT, 93

.macro write_stdout message,length
  mov x0, STDOUT_FD
  ldr x1, =\message
  mov x2, \length
  mov x8, SYS_WRITE
  svc 0
.endm

.macro exit_with code
  mov x0, \code
  mov x8, SYS_EXIT
  svc 0
.endm

.section .rodata
hello:
  .asciz "hello world!\n"

.section .text
.global _start
 start:
```

hello world

```
.equ  SYS_EXIT, 93
.section .rodata
fmt: .asciz "Hello %s!\n"
fmt_alt: .asciz "fmt addr: 0x%p\n"
val: .asciz "world"

.section .text
.global _start

_start:
  ldr x0, =fmt
  ldr x1, =val
  bl printf
  ldr x0, =fmt_alt
  ldr x1, =fmt
  bl printf
  mov x8, SYS_EXIT
  svc 0
```

goldbold

- https://godbolt.org/z/Gxjor9Kqj

```c
#include <stdio.h>

unsigned long long  factorial(unsigned long long  n){
    if (n == 0){
        return 1;
    }
    return n * factorial(n-1);
}


int main(int argc, char** argv){
    unsigned long long out = factorial(5);
    printf("Out: %llu\n", out);
return 0;
}
```

goldbolt

```
factorial(unsigned long long):                          // @factorial(unsigned long long)
        cbz     x0, .LBB0_4
        mov     x8, x0
        mov     w0, #1
.LBB0_2:                                // =>This Inner Loop Header: Depth=1
        subs    x9, x8, #1
        mul     x0, x8, x0
        mov     x8, x9
        b.ne    .LBB0_2
        ret
.LBB0_4:
        mov     w0, #1
        ret
main:                                   // @main
        str     x30, [sp, #-16]!                // 8-byte Folded Spill
        adrp    x0, .L.str
        add     x0, x0, :lo12:.L.str
        mov     w1, #120
        bl      printf
        mov     w0, wzr
        ldr     x30, [sp], #16                  // 8-byte Folded Reload
        ret
.L.str:
        .asciz  "Out: %llu\n"
```