CSci 5980/8980
Manual and Automated Binary Reverse Engineering
Slides 4: x86 Functions

Stephen McCamant
University of Minnesota

Outline

x86 functions

Data in functions

Data structures

The stack

- "The" stack is a memory region used for function-related data
  - Growth is stack-structured, but some random access
- Always allocated in multiples of 4 (32-bit) / 8 (64-bit) bytes
- Grows towards numerically lower addresses
- `%rsp` always points at lowest in-use location

Push and pop instructions

- `push` allocates one space and stores a value there
- `pop` loads the top value and moves the stack pointer to deallocate it
- Possible operands:
  - Push and pop of registers has a compact encoding (`0x5[0-f]`)
  - Can also push a constant, or push and pop memory locations
  - Some special registers accessed by push/pop

Offset-based stack accesses

- Can access stack locations as offsets from `%rsp`
  - "Top" is offset 0, older values are larger offsets
  - Offsets always a multiple of 4/8
- Also, allocate with `sub` and deallocate with `add`
- Mixing push/pop and offsets is confusing to people

Argument and return registers

- In 64-bit, first 6 integer/pointer arguments are passed in six registers
  - rdi, rsi, rdx, rcx, r8, r9
  - "Diane's silk dress costs $89"
- Return value is in `eax/rax`
  - `edx/rdx` avalilable for high bits

Sharing registers

- The registers have to be shared by all functions
  - Need a usage convention to avoid conflicts
- Mostly seen so far: scratch registers
  - Includes all the registers on the last slide
- Might be modified by any function call
- Convenient for leaf functions, but not around calls

Preserving registers

- Other convention: preserved registers appear not modified by a function call
  - More convenient for local variables in non-leaf functions
- If all code is in a function, how can preserved registers be used?
- Must save old value before use, and restore later
  - Commonly by push, and pop in reverse order

Which registers are preserved?

- For 64-bit, two part rule:
    - The low registers with `b` in the name (`rbx`, `rbp`)
    - The high registers numbered `r12` and higher
- `esi` and `edi` are preserved in 32-bit code
- `esp` is also preserved, in a sense

Stack frames

- The area of the stack used by a function invocation is one stack frame
- Frames also form a stack at a coarser granularity
- Return addresses mark the boundary between frames
- In 64-bit, frames have 16-byte alignment
    - With return address is at an even multiple of 8

Stack-based argument passing

- Stack locations are used for arguments after the sixth on x86-64
- And for all integer arguments on x86-32
- Just before return address, first argument on top
    - I.e., pushed in reverse order
- At function start, `0(%esp)` is return address, args start at `4(%esp)` (32-bit) or `8(%rsp)` (64-bit)

Variable-argument functions

- The stack argument order is chosen because C has variable-argument functions like `printf`
    - Varargs function implementations use macros `va_start`, `va_arg`, etc.
- First argument determines how many later arguments there are
- In the Windows world, this Linux/x86-32 calling convention is called `cdecl`

Varargs functions on x86-64

- Variable arguments are still passed in registers
- But usually pushed on the stack on the implementation side
    - So they can be referenced by pointers
- Weird quirk: number of arguments in SIMD registers passed in `%al`
    - To avoid saving SIMD registers if not needed

Frame pointers

- A frame pointer is a second stack pointer that stays fixed relative to the stack frame
    - Conventionally `%ebp`/`%rbp`
- Makes it easier to reference arguments and other stack variables when also using push/pop
    - But compilers can just do the math
- Traditionally default on x86-32, now rare except with `alloca`

---

x86-32 frame pointer conventions

- `%ebp` is preserved, so caller's value must be saved
- Conventionally, the first thing saved and last restored
- My `%ebp` points at the caller's saved `%ebp`
    - Return address at `4(%ebp)`, args start at `8(%ebp)`
    - Negative offsets from `%ebp` used for local variables, etc.
- Instructions `enter` and `leave` embody this convention

`alloca`

- The function `alloca` allocates space within the current stack frame
- Automatically freed on exit, like local variables
- Implemented just by changing the stack pointer
    - But requires a frame pointer since the size is dynamic
- Convenient and available on most Unix systems, but never standardized

Stack backtraces

- Can we recover the structure of stack frames at runtime?
  - Used in GDB's `backtrace` and related debugging features
- Traditional implementation followed the chain of frame pointers
- Now, debugging metadata maps from code locations to stack depth
  - More complex, but more efficient at runtime

Outline

x86 functions

Data in functions

Data structures

Local variables

- Local (C `auto`) variables are stored in registers or on the stack
- Stack or preserved registers needed if live across function calls
- The same location might hold different variables at different times
  - As long as their live ranges are disjoint
  - Registers more often reused, since the stack is cheap

Global variables

- Global variables are stored at static memory locations
  - `.data` section, or `.bss` for zero-initialized
- Location is a constant determined after linking
  - In assembly, a label
- Also C function-`static` variables

Position-independent code

- For shared libraries and better ASLR, let code execute at different addresses
  - Runtime relocations (locations fixups) are an alternative
  - But changing code has startup-time and sharing penalties
- For direct jumps, this is automatic from the relative offset encoding
  - Assuming caller and callee compiled together

RIP-relative addressing

- x86-64 mechanism for PIC data accesses: offset from program counter
  - Takes over mod=00, r/m = 5 32-bit displacement
  - Non-RIP mode available via SIB encoding
- Computed by linker once code and data locations determined
- Quite low overhead compared to non-PIC

---

x86-32 PIC

- Older approach: global data pointer in `%ebx`
- Initialize by stub call to get PC, add offset
- Performance hit from losing a register for other purposes
  - And x86-32 has fewer registers to start with
- This cost slowed adoption of full ASLR (PIE)

Runtime relocations

- PIC still needs runtime relocations for, e.g., initialized global function pointers
  - Part of ~100,000 instruction startup cost for glibc
- Windows demonstrates a relocation-only approach is also viable
  - Especially with fewer small programs and multi-process servers

Memory segments

- Primary memory management feature in 16-bit era: separate 64k areas
  - CS, SS and DS are defaults for code, stack, and non-stack data accesses
  - ES, later FS and GS also available via overrides
- Size limits on segments provided isolation
- In 32-bit paging era, mostly unused
  - All default segments set to address same flat memory
- Largely removed in x86-64

Thread-local storage

- Threads share memory but have own registers
- Want some data in multi-threaded programs to be private to each thread
  - Classic example: `errno` “global”
- x86-32 and -64 OSes use FS or GS for this purpose
- Segments set up by kernel, selected in user space

Stack canaries

- Security feature: check if return address has been overwritten
  - `-fstack-protector` in GCC, enabled in many distributions
- Store random value on stack, check if changed
- The per-execution canary value is stored with thread-local data
  - Relatively harder for attacks to access

Outline

x86 functions

Data in functions

Data structures

C pointer arithmetic

- C pointers are pretty much addresses
- Use same registers and operators as integer values
  - Most common operation is pointer + offset = pointer
- Biggest difference: pointer arithmetic unit is object size
  - I.e., integers multiplied by object size
  - This makes pointer types important

Arrays and pointers

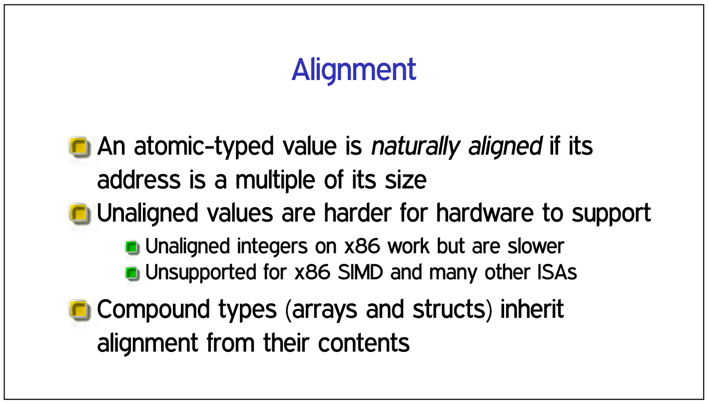
- C arrays are just objects next to each other in memory
  - No other runtime information like size
- Array indexing is just pointer arithmetic
  - Arrays decay to pointer to first element in many places
- Traditional local and global arrays are fixed-sized
- C99 variable-length local arrays are like `alloca`

Multidimensional arrays

- Multidimensional arrays are “rectangular” object layouts
  - C convention is row-major, i.e. contiguous last dimension
- Computing an element location involves multiplication
- Different from multi-level array of array pointers
  - (Despite the same access syntax in C)

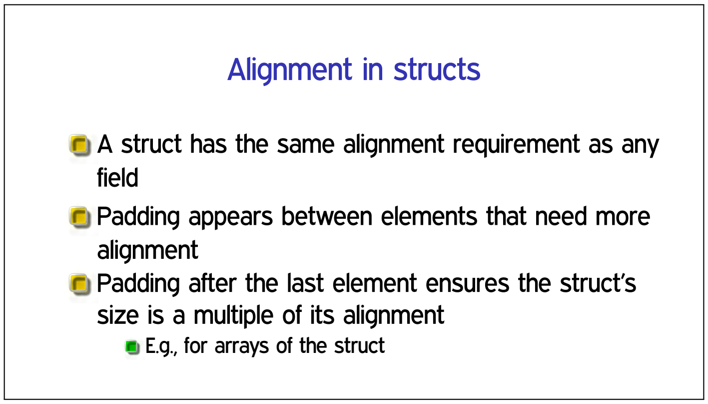
Structs and unions

- Objects of mixed type can be grouped in `struct`s
- Contiguous (except padding)
- Fields identified by byte offsets
- Union is the less-common counterparts where the objects overlap
  - I.e., every offset is 0
  - Only one is usable at once



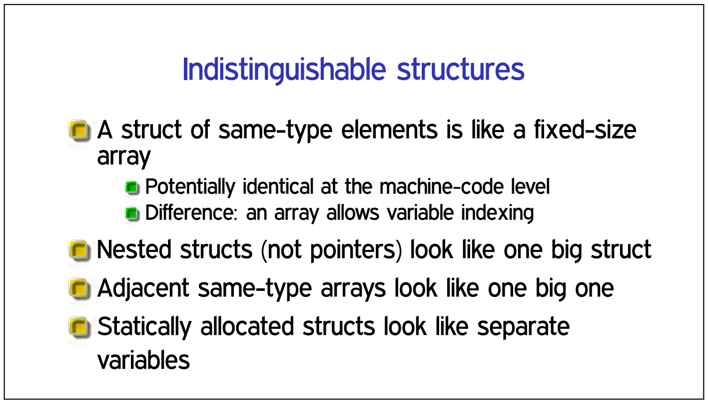
Alignment

- An atomic-typed value is *naturally aligned* if its address is a multiple of its size
- Unaligned values are harder for hardware to support
    - Unaligned integers on x86 work but are slower
    - Unsupported for x86 SIMD and many other ISAs
- Compound types (arrays and structs) inherit alignment from their contents



Alignment in structs

- A struct has the same alignment requirement as any field
- Padding appears between elements that need more alignment
- Padding after the last element ensures the struct's size is a multiple of its alignment
    - E.g., for arrays of the struct



Indistinguishable structures

- A struct of same-type elements is like a fixed-size array
    - Potentially identical at the machine-code level
    - Difference: an array allows variable indexing
- Nested structs (not pointers) look like one big struct
- Adjacent same-type arrays look like one big one
- Statically allocated structs look like separate variables