An Introduction to x86 Assembly Language

Basic Instructions and Shellcode

Most desktop or laptop computers in the world run some variant of the x86 processor. Thus, the most common ISA used in computer security is x86. Knowledge of x86 is necessary for understanding how to both reverse engineer and exploit binaries.

Why Assembly?

- Many computer exploit techniques are fundamentally low level
  - Reverse engineering is done at the assembly level
  - Exploit payloads are (usually) written in assembly
- BLUF: C isn't close enough to the metal to conduct real exploits

x86 ISA Overview

Or, why x86 sucks

- Not easy like MIPS...
- Little Endian (0xdeadbeef is |ef|be|ad|de|)
- CISC Architecture evolving from a 16-bit ISA
  - This is why a 'word' in x86 refers to two bytes
  - Thus, a 32-bit figure is a **dword** (64-bit is a **qword**)
- Many variants (read: *complex*)
- BUT: It's everywhere
  - Business concerns trump technical concerns every time

A Note on Syntax

- There are two syntax styles used in x86:
  - Intel Syntax
  - AT&T Syntax
- We'll be using Intel Syntax
  - I am going to (somewhat arbitrarily) say that it's easier and more intuitive
  - If you see lots of %s and $s, it's probably AT&T
  - Lots of small syntax changes that will trip you up

Brief Note on Segments

Deprecated stuff you can (mostly) ignore

- There are segment registers
- CS, DS, ES, FS, GS, SS
- Pretend they don't exist
- Relic of old 16-bit processors
- After the invention of paging, segments fell out of favor
- Now all they're there for is backwards compatibility

Sections of a Process Image

- .data
  - Initialized Data
- .bss
  - Uninitialized Data (set to 0)
- .text
  - Code
  - Entry Point (_start)
- The Stack
  - Local variables
- The Heap
  - Dynamically allocated memory (malloc/new)

```
section .data:
    message: db 'Hello World!'
    bufsz:   dd 1024
section .bss:
    fname:   resb 255
    num:     resd 1
section .text:
global _start
_start:
    (...)
    call main
    (...)
```

Memory Layout

```
+=============+
|    Stack    | ~0xff8e0000
+-------------+
|   Lots of   |
|    Empty    |
|    Space    |
+-------------+
|    Heap     | ~0x993a0000
+-------------+
|   Lots of   |
|    Empty    |
|    Space    |
+-------------+
|    .bss     |
+-------------+
|   .data     |
+-------------+
|   .text     | ~0x08040000
+-------------+
```

Registers

- General Purpose (eax, ebx, ecx, edx)
  - Leftovers from the 16-bit days
  - ax, bx, cx, and dx refer to low 16 bits
  - ?h refers to the high 8 bits of ?x
  - ?l refers to the low 8 bits of ?x
- Stack Pointer (esp)
- Base Pointer (ebp)
- Index Registers (edi, esi)
  - These are GPRs that also have special instructions

Register naming example:

```
        +-----------------+--------+--------+
    eax |              ax |   ah   |   al   |
        +-----------------+--------+--------+
```

Standard Instructions

The Basics

Note that at most one argument to an instruction may be a memory argument, and at least one argument must be a register (some exceptions).

mov eax, ebx	eax = ebx;
add eax, ebx	eax += ebx;
sub eax, ebx	eax -= ebx;
inc eax	++eax;
dec eax	--eax;
call foo	foo();
ret	return eax;
push 10h	*--esp = 0x10;
pop eax	eax = *esp++;

Memory Addressing

Syntax

- Memory references are always surrounded by brackets, like [esp] (equlvalent to *esp)
- Labels are by default pointers, so references to the value of global variables look like [foo]
- Most instructions can take **at most one** memory reference
- Each memory reference can have **up to** three components:
  - Base Address (Register)
  - Index (Register) * ElemSize (1, 2, 4, or 8)
  - Displacement (Constant)

```
[Base + Index*ElemSize ± Displacement]
```

Memory Addressing

Examples

- [eax] is equivalent to *eax
- [ebp-8] is equivalent to *(ebp-8)
- [esp+eax*4+0x20] is equivalent to ((int*)(esp+0x20))[eax]
- [0xdeadbeef] is equivalent to *((int*)0xdeadbeef)
- [foo] is equivalent to *foo where foo is a global pointer
- Basically: Think [] implies dereference (*)

The LEA instruction

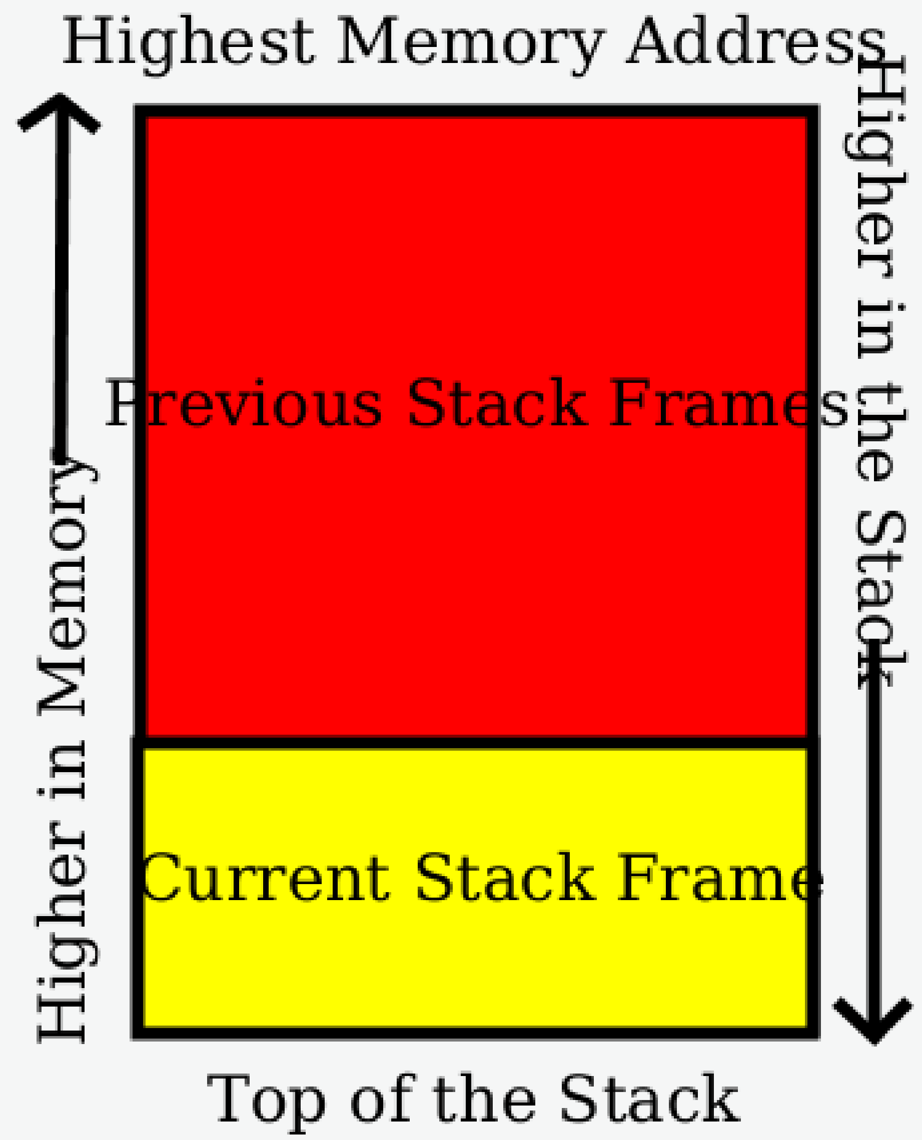
Load Effective Address

- A lot of the time we want to load some address to use later
- We can legally do something like mov eax,[esp+8]
- However, to get the address, mov eax,esp+8 is illegal
- So, we use the LEA instruction: lea eax,[esp+8]
- With LEA we can take the address of a memory reference and load it
- Basically: LEA is always used with [], and it loads the address of its argument instead.

The Stack

Overview

- The stack grows DOWNWARD
  - Top of the stack: lowest memory address
- The esp register points to the top of the stack
  - Adding to esp removes items from the stack
  - Subtracting to esp adds items to the stack



Stack Frames and Calling Conventions

- Caller pushes args on to stack, right to left
- Caller executes call instruction
  - call instruction pushes return address on to the stack
- Callee pushes ebp onto stack, sets ebp to esp
- Callee then allocates space for local variables
- Return value is in eax
- eax, ecx, edx are caller-saved (all others callee-saved)
- After return, caller responsible for cleaning arguments off the stack

Function Example

```c
int identity(int x) {
    return x;
}
```

```nasm
global identity
identity:
    push ebp                ; prologue
    mov ebp, esp            ;
    mov eax, [ebp+8]        ; do actual work
    mov esp, ebp            ; epilogue
    pop ebp                 ;
    ret                     ; return
```

Function Call Example

```
ebx = identity(ebx);
```

```
push ebx            ; push arguments on the stack
call identity       ; call function
add esp, 4          ; clean up passed arguments
mov ebx, eax        ; put return value where we want it
```

A quick note on ebp

What's the frame pointer for

- Constant location (esp changes when you ex. push/pop)
  - I cannot stress enough how much simpler this makes complex code
- Provides a linked list of stack frames (useful for debugging)
- That said, some compilers don't use it
  - GCC has the -fomit-frame-pointer option
  - This breaks some debuggers though
  - Some functions need the frame pointer though:
    - alloca()
    - C99 VLAs

Tips to Success

- DRAW THE STACK OUT
- Update your stack diagram as things are changed in memory
- Keep track of which addresses refer to which variables
- Know what is in all of the registers at all times

A complete program: Hello World

```
[BITS 32]

section .data:
    msg:    db `Hello, World!\n\0`  ; use backticks for the string
                                    ; note that we need to manually add the \0

section .text:
    extern printf           ; have to declare what functions we use
    global main             ; main is a global symbol (accessible from other files)

main:
    push ebp                ; standard prologue
    mov ebp, esp            ;
    push msg                ; push msg onto the stack (to use as an arg)
    call printf             ; printf(msg)
    add esp, 4              ; clean up the arg we pushed
    mov eax, 0              ; put return code in eax
    mov esp, ebp            ; standard epilogue
    pop ebp                 ;
    ret                     ;
```

Another Function Example

```c
void vulnerable() {
    char buf[256];
    gets(buf);
}
```

```asm
global vulnerable
vulnerable:
    push ebp                ; prologue
    mov ebp, esp            ;
    sub esp, 256            ; allocate space on stack for buf
    lea eax, [ebp-256]      ; load address of buf
    push eax                ; push args onto stack
    call gets               ; perform function call
    mov esp, ebp            ; epilogue
    pop ebp                 ;
    ret                     ; return
```

Exploit Techniques

- Return address is on the stack!
- Most common attack: overflow a stack buffer, overwrite return addr
- Vulnerable functions: gets(), scanf("%s"), strcpy()
- Overwrite the return address to run arbitrary
- Lots of techniques, varying degrees of sophistication
- Some defenses to mitigate dangers (more on this later...)

Branching

- Unconditional branch: use the jmp instruction
- Conditional Branching has two steps: check, then jump
- Two different instructions for the check step:
  - test instruction: use to check if something is zero
    - Most commonly: arguments should be the same e.g. test eax, eax
    - Can use the jz (jump if zero) and jnz (jump if not zero) commands after a test
  - cmp instruction: compare two numbers
    - Use like cmp a, b
    - Can use je (==) or jne (!=)
    - Signed arguments: use jl (<), jle (<=), jge (>=), jg (>)
    - Unsigned arguments: use jb (jump if below, <), jbe (<=), jae (>=), ja (jump if above, >)

Multiplication/Division (with bigger numbers)

If you actually care...

- mul reg performs eax*reg and stores the result in edx:eax
- Above notation means that edx stores the overflow (i.e. result == edx*232 + eax)
- imul is the same, but for signed numbers
- div reg divides edx:eax by reg and stores the result in eax, remainder in edx
- If there is overflow (i.e. result cannot fit in eax) the result is undefined/may crash
- idiv is the same again, but for signed numbers

Another Function Example

Another Function Example

```
global foo
foo:
    push ebp
    mov ebp, esp
    mov eax, [ebp+8]
    test eax, eax
    jnz bar
    inc eax
    jmp baz
bar:
    dec eax
    push eax
    call foo
    pop ecx
    inc ecx
    mul ecx
baz:
    mov esp, ebp
    pop ebp
    ret
```

Another Function Example

```
global foo
foo:
    push ebp
    mov ebp, esp
    mov eax, [ebp+8]
    test eax, eax
    jnz bar
    inc eax
    jmp baz
bar:
    dec eax
    push eax
    call foo
    pop ecx
    inc ecx
    mul ecx
baz:
    mov esp, ebp
    pop ebp
    ret
```

```
int fact(int x) {
    if (x == 0) return 1;
    return x * fact(x - 1);
}
```

Is assembly faster than C?

Is assembly faster than C?

- YES, in a quick non-scientific benchmark (of previous slide), speedup = 1.196

Is assembly faster than C?

- YES, in a quick non-scientific benchmark (of previous slide), speedup = 1.196
- BUT compilers have this awesome thing called optimization mode...
  - gcc -O1 is 1.327 times faster than assembly
  - gcc -O4 is 4.565 times faster than assembly

Is assembly faster than C?

- YES, in a quick non-scientific benchmark (of previous slide), speedup = 1.196
- BUT compilers have this awesome thing called optimization mode...
  - gcc -O1 is 1.327 times faster than assembly
  - gcc -O4 is 4.565 times faster than assembly
- Moral of the story is, while assembly is important for RevEng...
  - You probably won't beat a compiler with optimizations. They're **really good** at this shit
  - The best performance: have the compiler optimize C, then tweak assembly as needed

Is assembly faster than C?

- YES, in a quick non-scientific benchmark (of previous slide), speedup = 1.196
- BUT compilers have this awesome thing called optimization mode...
  - gcc -O1 is 1.327 times faster than assembly
  - gcc -O4 is 4.565 times faster than assembly
- Moral of the story is, while assembly is important for RevEng...
  - You probably won't beat a compiler with optimizations. They're **really good** at this shit
  - The best performance: have the compiler optimize C, then tweak assembly as needed
- Rule #1 of performance: **BENCHMARK**. #PrematureOptimizationIsTheRootOfAllEvil

System Calls

- How user processes invoke the kernel
- Activated by triggering interrupt 0x80
- man section 2 covers syscalls (same as in C)
- Separate calling convention though:
    - Syscall # in eax (see <asm/unistd_32.h>)
    - Args (left to right on manpage) in ebx, ecx, edx, esi, edi, ebp
    - Return value is in eax
    - Values in range [-4095, -1] indicate an error

Hello World, with System Calls

Look Mom, no C library!

```
[BITS 32]

section .data:
    hello:      db `Hello, World!\n`  ; this time, don't need \0
    helloLen:   dd $-hello            ; string length

section .text:
    global _start

_start:                         ; not using C, use _start instead of main
    mov eax, 4                  ; write() syscall number
    mov ebx, 1                  ; fd (STDOUT_FILENO)
    mov ecx, hello              ; data (pointer) to write
    mov edx, [helloLen]         ; number of bytes to write
    int 0x80                    ; call kernel
    mov eax, 1                  ; exit() syscall number
    mov ebx, 0                  ; return code (0)
    int 0x80                    ; call kernel
                                ; NOTE: we cannot return from _start, must exit()
```

Shellcode Example

```
[BITS 32]

; Note that we MUST have a valid stack for this to work!

xor ecx, ecx           ; zero ecx
mul ecx                ; edx:eax = eax*ecx, i.e. zeros edx and eax
mov al, 0xb            ; set eax to 0xb, syscall number for execve
push ecx               ; pushes a zero onto the stack (stack is \0\0\0\0)
push '//sh'            ; push '//sh' onto stack (stack is //sh\0\0\0\0)
push '/bin'            ; push '/bin' onto stack (stack is /bin//sh\0\0\0\0)
mov ebx, esp           ; set ebx (arg1: path) to stack pointer (�/bin//sh�)
push ecx               ; push another zero (execve needs a NULL at the end)
push ebx               ; push addr of "/bin//sh"
mov ecx, esp           ; set ecx (arg2: argv) to ["/bin//sh", 0]
                       ; edx (arg3: envp) is already NULL from `mul ecx`
int 80h                ; perform system call
```