Learning x86 and Calling Conventions

Dr. Jared DeMott
CTO AND FOUNDER

@jareddemott www.vdalabs.com

Overview

Review/Solution

Architecture

Compiler tendencies

```
14  int process_input(string mystr) {
15      int x;
16      const char * ptr = mystr.c_str();
17      x = atoi(ptr);
18      return x;
19  }
20
21  int main(int argc, char * argv[]) {
22      string userinput;
23      int result=0;
24
25      while(1) {
26          cout << "Please enter the secret code: " << endl;
27          userinput = get_code();
28
29          cout << "Validating...." << endl;
30          result = process_input(userinput);
31
32          if(result == -10) {
33              cout << "Creditials matched.  Welcome 31337 user." << endl;
34              cout << "Now launching secret application calc.exe" << endl;
35              system("calc");
36              return(0);
37          }
38          else {
39              cout << "Passcode incorrect." << endl << "Try again." << endl;
40              system("pause");
41              system("cls");
42          }
43      }
44  }
```

x86-64 Registers

https://en.wikipedia.org/wiki/X86-64

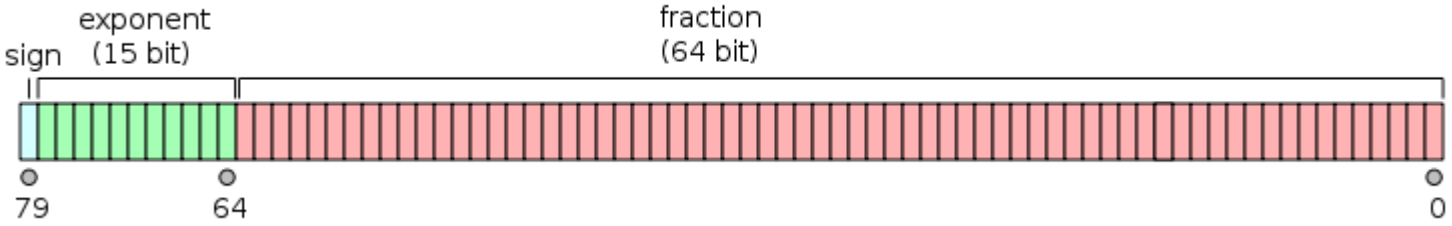
Segment registers
SSE registers
More

rax	eax	ax:ah	al
rbx	ebx	bx:bh	bl
rcx	ecx	cx:ch	cl
rdx	edx	dx:dh	dl
rsp	esp		
rbp	ebp		
rdi	edi		
rsi	esi		
rip	eip	ip	
	eflags	flags	
r8	r8d	r8w	r8b
r15	r15d	r15w	r15b

Floating Point Unit

80486 and subsequent integrated x87 functionality on chip

- Each x87 register, known as ***ST(0) through ST(7), is 80 bits wide*** and stores numbers in the IEEE floating-point standard double extended precision format
- These registers are organized as a stack with ST(0) as the top
- ST0 must always be one of the two operands, regardless of whether the other operand is ST(x) or a memory operand



Common Instructions

mov – Moves source to destination

lea – Loads effective address

jmp – Jump

- jne – Jump if not equal
- jg – Jump if greater than

call – Unconditional function call

ret – Returns from a function to the caller

add – Adds two values

sub – subtracts two values

xor – XORs two values

cmp – Compares two registers via subtraction

test – Logically *and* two operands together

Flags

C – holds a carry or a borrow

P – the parity flag (little use today)

A – auxiliary flag used with DAA and DAS

Z – zero

S – sign

O – Overflow

D – direction, used with string instructions

I – interrupt (interrupt on/off)

T – trace flag (trace on/off)

Points of Interest for “signed bugs”

x86 has a simple way of differentiating signed vs. unsigned comparisons

- above/below indicates an unsigned comparison
  - E.g. JAE(JNB,JNC)/JBE(JNA) means "Jump if above/below or equal" and is unsigned
    - These use the carry flag
- greater/less than indicates a signed comparison
  - E.g. JGE(JNL)/JLE(JNG) is signed
    - These use the sign, zero, or overflow flag

Quiz Time

Memset(buf, 0x00, 100)

```
_asm
    {
        mov ecx, 100
        xor eax, eax
        lea edi, buf
        cld
        rep stosb
    }
```

Memcpy(dst_buf, str_src, strlen(str_src))

```
_asm{
        mov ecx, 100        //not the best strlen but works here
        xor eax, eax
        mov edi, str_src
        repne scasb         //inc'ed di via count
        mov ecx, str_src
        sub edi, ecx        //Calculates strlen
        mov ecx, edi
        mov esi, str_src
        lea edi, dst_buf
        rep movsb           //works the copy a byte at a time
}
```

Calling Conventions

The stack layout and making function calls

Function Calling Conventions

Cdecl
- Most common, especially on UNIX
- Parameters are received in reverse order
- Caller unwinds stack

Stdcall
- Common in Windows
- Parameters are received in reverse order
- Function unwinds stack

Fastcall
- Higher performance
- First two parameters are passed over registers
- No need to unwind stack

Cdecl (Unix)

```
_cdecl int MyFunc1
( int a, int b)
{ return a + b; }
```

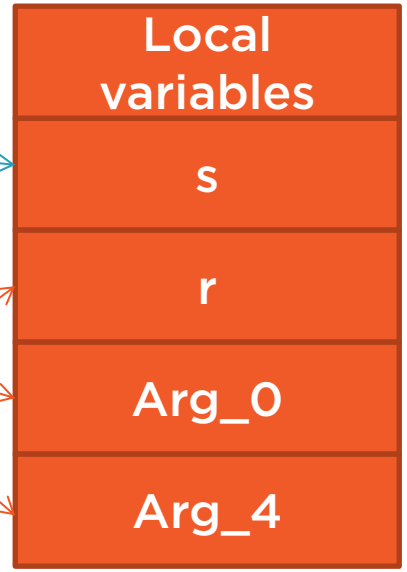
```
x = MyFunc1(2, 3);
```

```
push ebp
mov ebp, esp
```
Enter

```
mov eax, [ebp + 8]
mov edx, [ebp + 12]
add eax, edx
```

```
mov esp, ebp
pop ebp
ret
```
Leave

```
push 3
push 2
call _MyFunction1
add esp, 8
```

Stdcall (Windows)

```
_cdecl int MyFunc1
( int a, int b)
{ return a + b; }
```

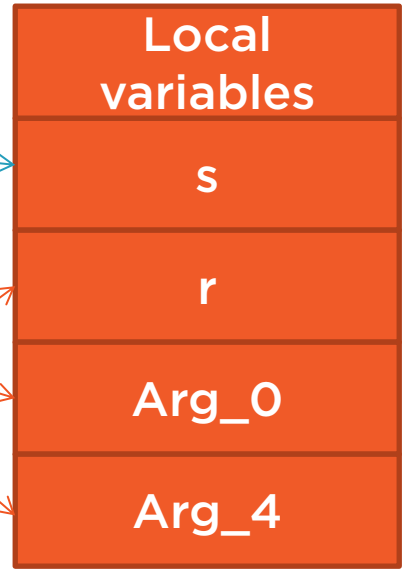
```
x = MyFunc1(2, 3);
```

```
push ebp
mov ebp, esp
```
Enter

```
mov eax, [ebp + 8]
mov edx, [ebp + 12]
add eax, edx
```

```
mov esp, ebp
pop ebp
ret 8
```
Leave

```
push 3
push 2
call _MyFunction1
```



Fastcall

```
_cdecl int MyFunc1
( int a, int b)
{ return a + b; }
```

```
;push ebp
;mov ebp, esp
```
Enter

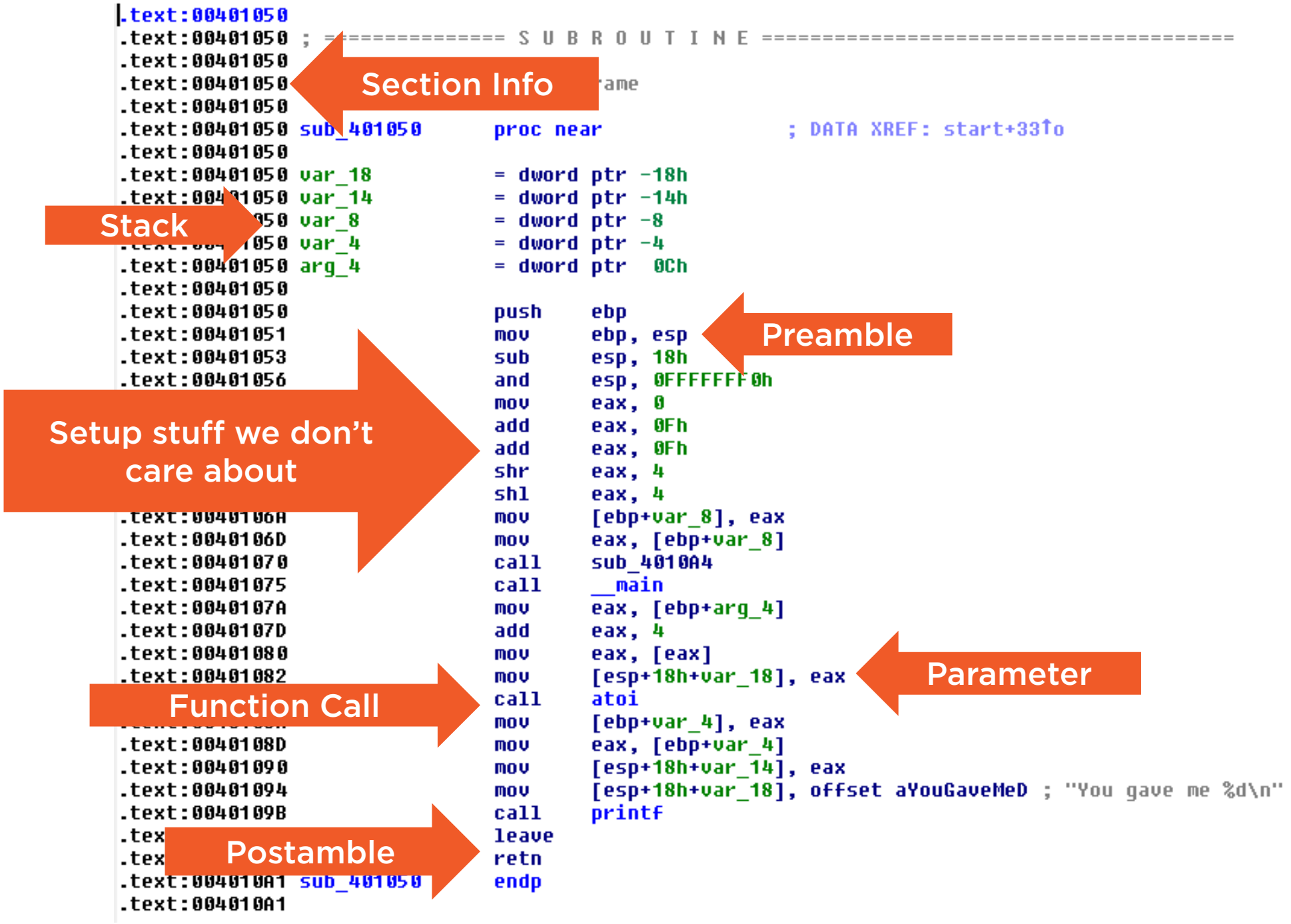
```
add eax, edx
```

```
;mov esp, ebp
;pop ebp
ret
```
Leave

```
x = MyFunc1(2, 3);
```

```
mov eax, 3
mov edx, 2
call _MyFunction1
```

r


.text:00401050
.text:00401050 ; =============== S U B R O U T I N E =======================================
.text:00401050
.text:00401050
.text:00401050
.text:00401050 sub_401050 proc near ; DATA XREF: start+33↑o
.text:00401050
.text:00401050 var_18 = dword ptr -18h
.text:00401050 var_14 = dword ptr -14h
.text:00401050 var_8 = dword ptr -8
.text:00401050 var_4 = dword ptr -4
.text:00401050 arg_4 = dword ptr 0Ch
.text:00401050
.text:00401050 push ebp
.text:00401051 mov ebp, esp
.text:00401053 sub esp, 18h
.text:00401056 and esp, 0FFFFFFF0h
mov eax, 0
add eax, 0Fh
add eax, 0Fh
shr eax, 4
shl eax, 4
mov [ebp+var_8], eax
.text:0040106D mov eax, [ebp+var_8]
.text:00401070 call sub_4010A4
.text:00401075 call __main
.text:0040107A mov eax, [ebp+arg_4]
.text:0040107D add eax, 4
.text:00401080 mov eax, [eax]
.text:00401082 mov [esp+18h+var_18], eax
call atoi
mov [ebp+var_4], eax
.text:0040108D mov eax, [ebp+var_4]
.text:00401090 mov [esp+18h+var_14], eax
.text:00401094 mov [esp+18h+var_18], offset aYouGaveMeD ; "You gave me %d\n"
.text:0040109B call printf
leave
retn
.text:004010A1 sub_401050 endp
.text:004010A1
Section Info
Stack
Preamble
Setup stuff we don't care about
Parameter
Function Call
Postamble

Code + Data == Win

Understand the function call tree and corresponding data

- Btw, Decompilers are nice

Sometimes easy

- Dynamically linked code using well known functions
- E.g. see a socket() call
  - Look at man page to determine types of parameters passed

Sometimes harder

- Statically linked and stripped code
- Intentionally obfuscated code
- Very large or complex code with multiple threads, etc.

A Note on Finding *main()*

Sometimes easy - tool finds it

- Either from symbols or from signature files

Sometimes harder

- Start is first function
- Main is called from start
  - Or from a function that is called within start

Look for common patterns

- Called toward bottom of start
- Often function with 3 args
- Sometimes an argument to _start_main
  - Usually the first arg

Summary

Basic assembly

Calling conventions

Finding main()

Next:

- Common code patterns in assembly