Reversing - Stack and Functions

Stuart Nevans Locke

Overview

- Review
- Intro to the Stack
- Some Instructions
- Functions
- More instructions

Review - Registers

Registers:
GPRs

Instruction Pointer

Review - Instructions

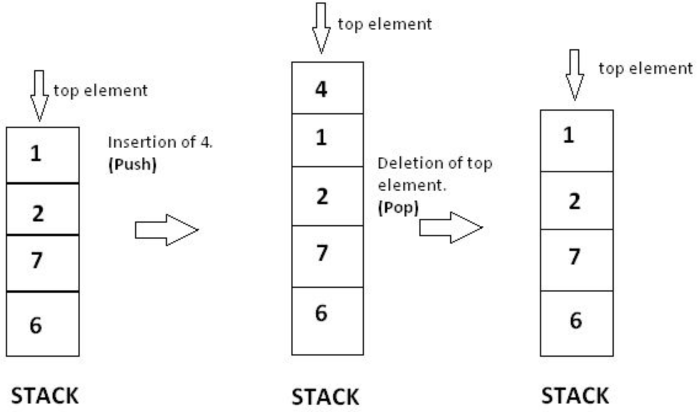
```
mov rax, 0

add rax, 5

mul 10

mov [rbx], rax
```

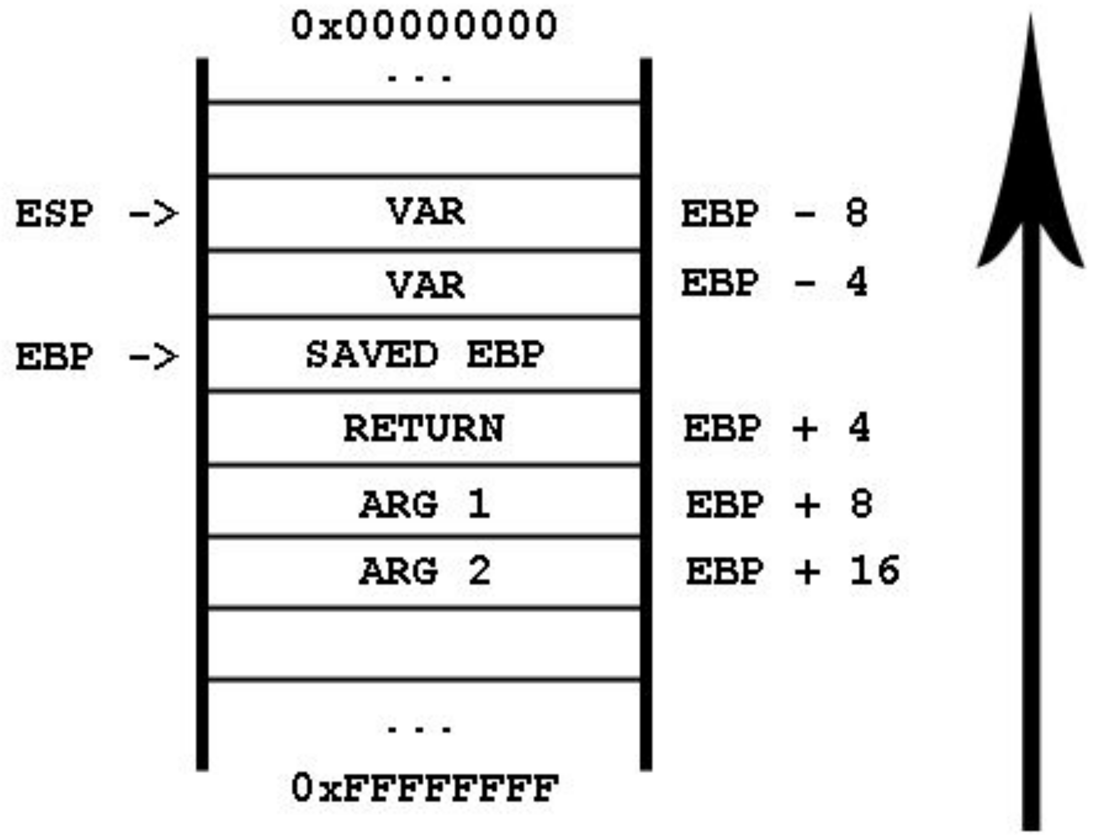
Review - Variables

How are variables stored at a low level?

Introduction to the Stack

- Imagine a stack of books
  - LIFO (Last in - first out)
- Adding to the stack
  - pushing
- Removing from the stack
  - popping





X86 Stack Grows Down
ESP/RSP Keeps track of “top of the stack” -- even though it grows down

Stack Instructions

push SRC

Pushes an argument to the top of the stack

pop DST

Pops from the top of the stack and stores the result in DST

Stack Instructions - Examples

```
mov rax, 8

push rax

push 1

pop rax

pop rbx
```

```
push 1

sub rsp, 8

mov [rsp], 16

pop rax

pop rbx
```

JMP Instruction

Jmp DST

Jump to a given address.

Examples:
jmp rax

jmp 0x100

jmp [rax+4]

Functions

- There’s nothing technically required to have a function in assembly - just instructions at an address
- Still, we have some convention for them

```
main:

call label
```

```
label:

mov rax, 1

ret
```

Function Instructions

call address

Calls a function at a given address. It does this by **pushing** rip/eip/pc to the stack and setting rip equal to the address.

Examples:
call rax

call 0x5151

Function Instructions

ret

Returns from a function. It does this by popping the top item from the stack and setting rip equal to it.

Conceptually the same as “*pop rip*”

Examples:

ret

Function - Calling Convention

- Differs based on architecture
- On 32 bit x86, you push the arguments to the stack before calling
- On x86-64, you set some registers and they are assumed to have the arguments
- Return value is stored in *eax/rax*

Function Examples - C Code

```
int main(){
        int result = add(1, 2);
}
int add(int arg1, int arg2){
        return arg1 + arg2;
}
```

Function Examples

x86:

```
push 1

push 2

call add
```

```
add:

mov rax, [rsp-4]

mov rbx, [rsp-8]

add rax, rbx

ret
```

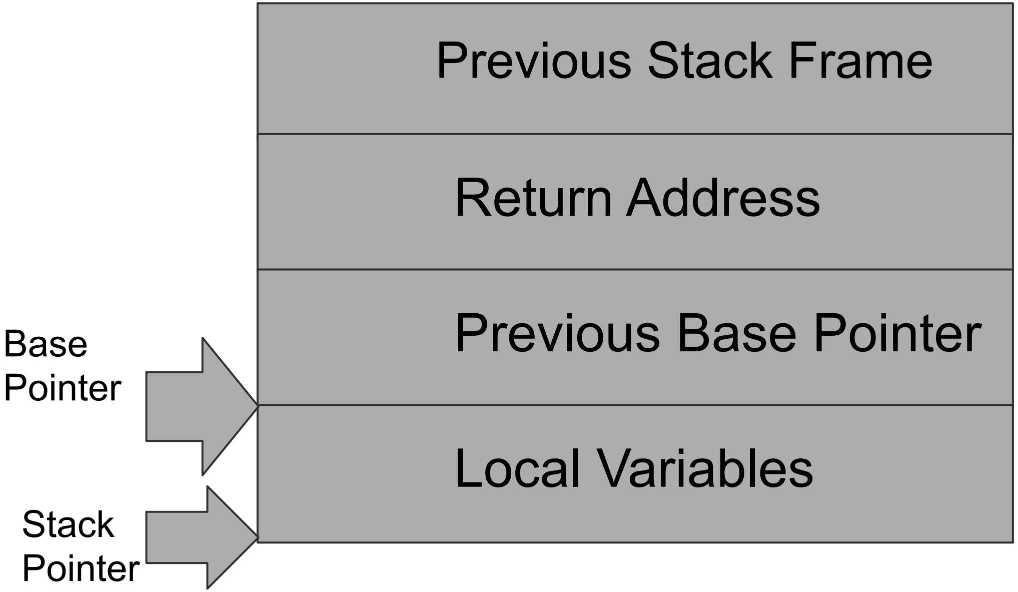
Function Examples

X86-64:

```
mov rdi, 1

mov rsi, 2

call add
```

```
add:

add rdi, rsi

mov rax, rdi

ret
```

Stack Frames



Some functions have local variables, which are stored on the stack

They use the “base pointer to help with this”

Every function saves the calling function’s base pointer, and uses the base pointer to access local variables

Example: C code

```
int main(){
        int result = add(1,2,3);
}
int add(int a, int b, int c){
        int x = a + b;
        return x + c;
}
```

Example x86-64

```
main:
mov rdi, 1
mov rsi, 2
mov rdx, 3
call add
```

```
add:
push rbp
mov rbp, rsp
sub rsp, 0x8
add rdi, rsi
mov [rbp - 0x8], rdi
mov rax, [rbp-0x8]
add rax, rdx
leave
ret
```

Leave

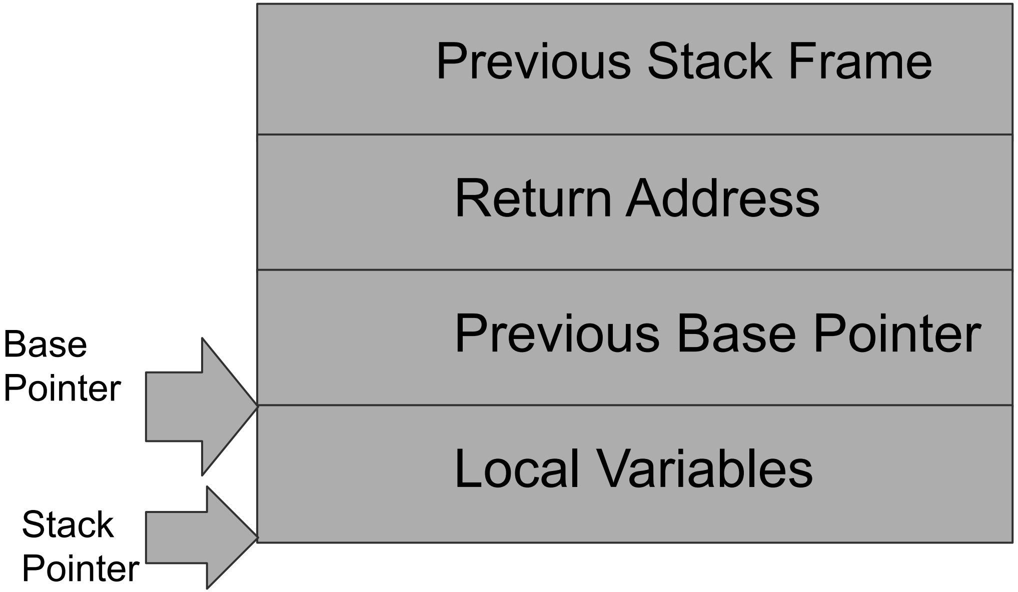
```
leave
```

Cleans up a stack frame.

Leave is the same as:

```
mov rsp, rbp

pop rbp
```

Prologue and Leave



```
push rbp

mov rbp, rsp

sub rsp, SIZE
```

Leave:

```
mov rsp, rbp
pop rbp
```

Instructions - CMP

Cmp arg1, arg2

Compare arg1 and arg2. It essentially does this via subtraction. The result of the comparison is saved in EFLAGS. (**Zero flag**, **Sign flag**, Overflow Flag, Carry Flag)

Examples:

cmp rax, 5

cmp rax, rbx

Instructions - Conditional Jumps

jz/je address-- Jump if Zero. Checks the zero flag. If set, jump to address. Otherwise, just continue executing. (jump zero, jump equal)

jnz/jne address -- Jumps if zero flag is *NOT* set.

jg - Jump if greater than. (Actual flags are more complicated).

jge - Jump if greater than or equal to.

Examples:

```
mov rax, 8

cmp rax, 7

je location1

jg location2
```

Some actual reversing

Download cutter

https://cutter.re/

Open a linux VM or WSL (Windows Subsystem for Linux)