x86 Disassembly/Functions and Stack Frames

Functions and Stack Frames

In the execution environment, functions are frequently set up with a "**stack frame**" to allow access to both function parameters, and automatic function variables. The idea behind a stack frame is that each subroutine can act independently of its location on the stack, and each subroutine can act as if it is the top of the stack.

When a function is called, a new stack frame is created at the current **esp** location. A stack frame acts like a partition on the stack. All items from previous functions are higher up on the stack, and should not be modified. Each current function has access to the remainder of the stack, from the stack frame until the end of the stack page. The current function always has access to the "top" of the stack, and so functions do not need to take account of the memory usage of other functions or programs.

Standard Entry Sequence

For many compilers, the standard function entry sequence is the following piece of code (*X* is the total size, in bytes, of all *automatic* variables used in the function):

```
push ebp
mov ebp, esp
sub esp, X
```

For example, here is a C function code fragment and the resulting assembly instructions (the code generated with ABI standards won't have "sub esp, 12" instruction due to red zones):

```
void MyFunction()
{
  int a, b, c;
  ...
```

```
_MyFunction:
  push ebp     ; save the value of ebp
  mov ebp, esp ; ebp now points to the top of the stack
  sub esp, 12  ; space allocated on the stack for the local variables
```

This means local variables can be accessed by referencing ebp. Consider the following C code fragment and corresponding assembly code:

```
a = 10;
b = 5;
c = 2;
```

```
mov [ebp -  4], 10  ; location of variable a
mov [ebp -  8], 5   ; location of b
mov [ebp - 12], 2   ; location of c
```

This all seems well and good, but what is the purpose of **ebp** in this setup? Why save the old value of ebp and then point ebp to the top of the stack, only to change the value of esp with the next instruction? The answer is *function parameters*.

Consider the following C function declaration:

```
void MyFunction2(int x, int y, int z)
{
  ...
}
```

It produces the following assembly code:

```
_MyFunction2:
  push ebp
  mov ebp, esp
  sub esp, 0     ; no local variables, most compilers will omit this line
```

Which is exactly as one would expect. So, what exactly does **ebp** do, and where are the function parameters stored? The answer is found when we call the function.

Consider the following C function call:

```
MyFunction2(10, 5, 2);
```

This will create the following assembly code (using a Right-to-Left calling convention called CDECL, explained later):

```
push 2
push 5
push 10
call _MyFunction2
```

Note: Remember that the **call** x86 instruction is basically equivalent to

```
push eip + 2 ; return address is current address + size of two instructions
jmp _MyFunction2
```

It turns out that the function arguments are all passed on the stack! Therefore, when we move the current value of the stack pointer (**esp**) into **ebp**, we are pointing ebp directly at the function arguments. As the function code pushes and pops values, ebp is not affected by esp. Remember that pushing basically does this:

```
sub esp, 4   ; "allocate" space for the new stack item
mov [esp], X ; put new stack item value X in
```

This means that first the return address and then the old value of **ebp** are put on the stack. Therefore [ebp] points to the location of the old value of ebp, [ebp + 4] points to the return address, and [ebp + 8] points to the first function argument. Here is a (crude) representation of the stack at this point:

```
:     :
|  2 | [ebp + 16] (3rd function argument)
|  5 | [ebp + 12] (2nd argument)
| 10 | [ebp + 8]  (1st argument)
| RA | [ebp + 4]  (return address)
| FP | [ebp]      (old ebp value)
|    | [ebp - 4]  (1st local variable)
:     :
:     :
|    | [ebp - X]  (esp - the current stack pointer. The use of push / pop is valid now)
```

The stack pointer value may change during the execution of the current function. In particular this happens when:

- parameters are passed to another function;
- the pseudo-function "alloca()" is used.

[FIXME: When parameters are passed into another function the esp changing is not an issue. When that function returns the esp will be back to its old value. So why does ebp help there. This needs better explanation. (The real explanation is here, ESP is not really needed: http://blogs.msdn.com/larryosterman/archive/2007/03/12/fpo.aspx)] This means that the value of **esp** cannot be reliably used to determine (using the appropriate offset) the memory location of a specific local variable. To solve this problem, many compilers access local variables using negative offsets from the **ebp** registers. This allows us to assume that the same offset is always used to access the same variable (or parameter). For this reason, the ebp register is called the **frame pointer**, or FP.

Standard Exit Sequence

The Standard Exit Sequence must undo the things that the Standard Entry Sequence does. To this effect, the Standard Exit Sequence must perform the following tasks, in the following order:

1. Remove space for local variables, by reverting **esp** to its old value.
2. Restore the old value of **ebp** to its old value, which is on top of the stack.
3. Return to the calling function with a ret command.

As an example, the following C code:

```c
void MyFunction3(int x, int y, int z)
{
  int a, b, c;
  ...
  return;
}
```

Will create the following assembly code:

```asm
_MyFunction3:
  push ebp
  mov ebp, esp
  sub esp, 12 ; sizeof(a) + sizeof(b) + sizeof(c)
  ;x = [ebp + 8], y = [ebp + 12], z = [ebp + 16]
  ;a = [ebp - 4] = [esp + 8], b = [ebp - 8] = [esp + 4], c = [ebp - 12] = [esp]
  mov esp, ebp
  pop ebp
  ret 12 ; sizeof(x) + sizeof(y) + sizeof(z)
```

Non-Standard Stack Frames

Frequently, reversers will come across a subroutine that doesn't set up a standard stack frame. Here are some things to consider when looking at a subroutine that does not start with a standard sequence:

Using Uninitialized Registers

When a subroutine starts using data in an *uninitialized* register, that means that the subroutine expects external functions to put data into that register before it gets called. Some calling conventions pass arguments in registers, but sometimes a compiler will not use a standard calling convention.

"static" Functions

In C, functions may optionally be declared with the **static** keyword, as such:

```
static void MyFunction4();
```

The **static** keyword causes a function to have only local scope, meaning it may not be accessed by any external functions (it is strictly internal to the given code file). When an optimizing compiler sees a static function that is only referenced by calls (no references through function pointers), it "knows" that external functions cannot possibly interface with the static function (the compiler controls all access to the function), so the compiler doesn't bother making it standard.

Hot Patch Prologue

Some Windows functions set up a regular stack frame as explained above, but start out with the apparently non-sensical line

```
mov edi, edi;
```

This instruction is assembled into 2 bytes which serve as a placeholder for future function patches. Taken as a whole such a function might look like this:

```
nop              ; each nop is 1 byte long
nop
nop
nop
nop

FUNCTION:        ; <-- This is the function entry point as used by call instructions
mov edi, edi     ; mov edi,edi is 2 bytes long
push ebp         ; regular stack frame setup
mov ebp, esp
```

If such a function needs to be replaced without reloading the application (or restarting the machine in case of kernel patches) it can be achieved by inserting a jump to the replacement function. A short jump instruction (which can jump +/- 127 bytes) requires 2 bytes of storage space - just the amount that the "mov edi,edi" placeholder provides. A jump to any memory location, in this case to our replacement function, requires 5 bytes. These are provided by the 5 no-operation bytes just preceding the function. If a function thus patched gets called it will first jump back by 5 bytes and then do a long jump to the replacement function. After the patch the memory might look like this

```
LABEL:
jmp REPLACEMENT_FUNCTION ; <-- 5 NOPs replaced by jmp

FUNCTION:
jmp short LABEL          ; <-- mov edi has been replaced by short jump backwards
push ebp
mov ebp, esp             ; <-- regular stack frame setup as before
```

The reason for using a 2-byte mov instruction at the beginning instead of putting 5 nops there directly, is to prevent corruption during the patching process. There would be a risk with replacing 5 individual instructions if the instruction pointer is currently pointing at any one of them. Using a single mov instruction as placeholder on the other hand guarantees that the patching can be completed as an atomic transaction.

Local Static Variables

Local static variables cannot be created on the stack, since the value of the variable is preserved across function calls. We'll discuss local static variables and other types of variables in a later chapter.