Memory Layout

The systems memory is divided into several sections. I prefer to imagine the stack with low addresses at the top and high addresses at the bottom. The reason for this is because it's more like a normal numeric list and it's how you'll most often see it being represented. Also, I'm pretty sure that's how your computer sees it. Be warned though, you will see people represent the memory layout as starting from higher addresses. If you don't know what I just said, don't worry about it.

Assembly Segments

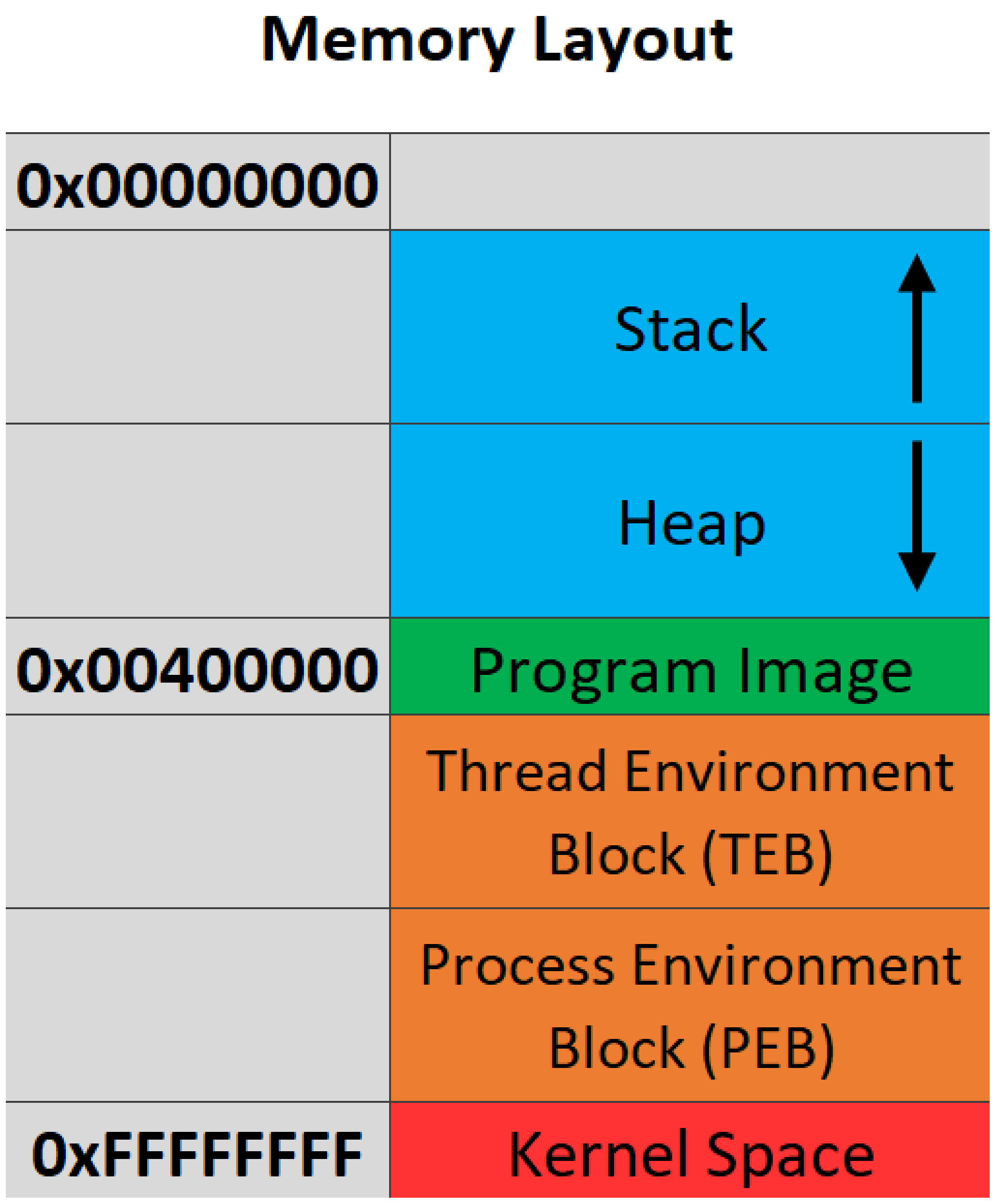
There are different segments/sections in which data or code is stored.

- **.data** - Contains global and static variables. This segment is a fixed size.
- **.bss** - Contains variables that are not explicitly initialized to any value. These variables are often zeroed out when the program starts.
- **.text** - Contains the code of the program (don't blame me for the name, I didn't make it).
  The start of the program code is declared with "global _start".

Overview of Memory Sections

- **Stack** - Area in memory that can be used quickly for static data allocation. Data is read and written as "last-in-first-out" (LIFO). The LIFO structure of the stack is often represented with a stack of plates. You can't simply take out the third plate from the top, you have to take off one plate at a time to get to it. You can only access the piece of data that's on the top of the stack, so to access other data you need to move what's on top out of the way. When I said that the stack holds static data I'm referring to data that has a known length such as an integer. We know that an integer will only be 4 bytes so we can through that on the stack. Unless a maximum length is specified, user input should be stored on the heap because the data has a variable size. When you put data on top of the stack you **push** it onto the stack. When you remove a piece of data off the top of the stack you **pop** it off the stack. There are two registers that are used to keep track of the stack. **When you add data to the stack, the stack "grows" towards lower memory addresses.** The **stack pointer (RSP)** is used to keep track of the top of the stack and the **base pointer (RBP)** is used to keep track of the base/bottom of the stack.
- **Heap** - Similar to the stack but used for dynamic allocation and it's a little slower to access. The heap is typically used for data that is more dynamic (changing or unpredictable). Things such as structures and user input would be stored on the heap. If the size of the data isn't known at compile-time, it's usually stored on the heap. **When you add data to the heap it grows towards higher addresses.**
- **Program Image** - This is the program loaded into memory. On Windows, this is typically a **Portable Executable (PE)**.
- **DLL** - **Dynamic Link Library (DLL)**. Libraries that can be used by programs.
- **TEB** - The **Thread Environment Block (TEB)** stores information about the currently running thread(s).
- **PEB** - The **Process Environment Block (PEB)** stores information about the process and the loaded modules. One piece of information the PEB stores is "BeingDebugged" which can be used to determine if the current process is being debugged.
  MSDN: https://docs.microsoft.com/en-us/windows/desktop/api/winternl/ns-winternl-_peb

Here is a general overview of how memory is laid out in Windows. **This is extremely simplified.**



Stack Frames

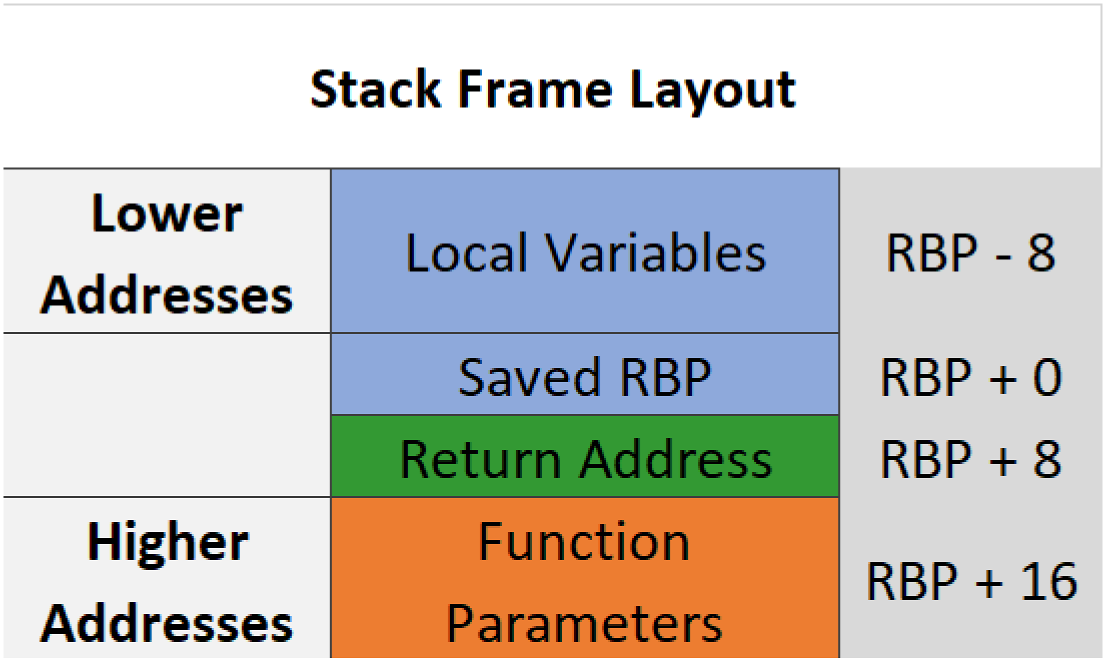
Stack frames are basically chunks of data for functions. This data includes local variables, the saved base pointer, the return address of the caller, and function parameters. Consider the following example:

```
int Square(int x){
    return x*x;
}
int main(){
    int num = 5;
    Square(5);
}
```

In this example, the `main()` function is called first. When `main()` is called, a stack frame is created for it. This stack frame includes the local variable `num`, the saved (previous) base pointer, the return address, and the parameters passed to it (in this case there are no parameters passed to main). Remember, the base pointer points to the base/bottom of the stack. The base pointer is saved so when `Square()` returns it can be restored to what it was before the function call. This ensures that the stack stays organized and doesn't get all wonky. The return address is the address of the instruction after the instruction that called the function. That may sound confusing, hopefully, this can clear it up:

```
mov RAX, 15 ;RAX = 15
call func   ;Instruction that calls func. Same as func();
mov RBX, 23 ;RBX = 23. This is the saved return address for the call on the
previous line.
```

I know that this can be a bit confusing but it really is quite simple in how it works. It just may not be intuitive at first. It's simply telling the computer where to go (what instruction to execute) when the function returns. You don't want it to execute the instruction that called the function because that will cause an infinite loop. This is why the next instruction is used as the return address instead.
Here is the layout of a stack frame:



Note the location of everything. This will be helpful in the future.

> If this section was confusing, read through 0x203-Instructions.md then re-read this section. After you re-read this section you might want to read 0x203-Instructions.md again. I apologize for this but there really isn't a good order to teach this stuff in since it all goes together.

Windows x64 Calling Convention

When a function is called you could, theoretically, pass parameters via registers, the stack, or even on disk. You just need to be sure that the function you are calling knows how you are calling it. This isn't too big of a problem if you are using your own functions, but things would get messy when you start using libraries. To solve this problem we have **calling conventions** that define how parameters are passed to a function, who allocates space for local variables, and who cleans up the stack.

> **Callee** refers to the function being called, and the **caller** is the function making the call.

There are several different calling conventions including cdecl, syscall, stdcall, fastcall, and many more. Because we are going to be reverse engineering on x64 Windows we should only have to focus on x64 fastcall. If you do plan on reversing on other platforms, be sure to learn the calling convention(s).

Fastcall

Fastcall is *the* calling convention for x64 Windows. Windows uses a four-register fastcall calling convention by default. I'm not sure if this calling convention is the only one used on x64, but I hope it is. Differing calling conventions can be extremely annoying to deal with as is the case in x32. When talking about calling conventions you will hear about something called the "Application Binary Interface" (ABI). The ABI defines various rules for programs such as calling conventions, parameter handling, and more.

So how does the x64 Windows calling convention work?

- The first four arguments/parameters are passed in registers. Parameters that are *not* floating point values (floats or doubles) will be passed via RCX, RDX, R8, and R9 (in that order). Non-floating point values include pointers, integers, booleans, chars, etc. Floating point parameters will be passed via XMM0, XMM1, XMM2, and XMM3 (in that order). Floating point values include floats and doubles. If the parameter being passed is too big to fit in a register then it is passed by reference. Parameters are never spread across multiple registers. Any other parameters are put on the stack.

> All parameters, even ones passed through registers, have space reserved on the stack for them. Also, there is always space made for 4 parameters on the stack even if there are no parameters passed. This space isn't completely wasted because the compiler can, and often will, use it.

- A function's return value is passed via RAX if it's an integer, bool, char, etc., or XMM0 if it's a float or double.
- Member functions (functions that are part of a class/struct) have an implicit first parameter for the "this" pointer. Because it's a pointer it will be passed via RCX.[1]
- The *caller* is responsible for allocating space for parameters for the *callee*. The caller must always allocate space for 4 parameters even if no parameters are passed.

That's the x64 Windows fastcall calling convention for you. Learning your first calling convention is like learning your first programming language. It seems complex and daunting at first, but it's really quite simple. It's typically harder to learn you first calling convention than it is your second or third.

If you want to learn more about this calling convention you can here:
https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2019
https://docs.microsoft.com/en-us/cpp/build/x64-software-conventions?view=vs-2019

> If this section was confusing, read through 0x203-Instructions.md then re-read this section. After you re-read this section you might want to read 0x203-Instructions.md again. I apologize for this but there really isn't a good order to teach this stuff in since it all goes together.

Sources

https://docs.microsoft.com/en-us/cpp/build/x64-software-conventions?view=vs-2019
https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2019
https://docs.microsoft.com/en-us/cpp/build/prolog-and-epilog?view=vs-2019
https://www.gamasutra.com/view/news/171088/x64_ABI_Intro_to_the_Windows_x64_calling_convention.php