Development & Security

By Jurriaan Bremer @skier_t

x86 API Hooking Demystified

Posted on **July 2, 2012**

Table of Contents:

Abstract

Today's post presents several ways of API hooking under the x86 instruction set.

Throughout the following paragraphs we will introduce the reader to API hooking, what we can do with it, why API hooking is useful, the most basic form of API hooking. And, after presenting a simple API hooking method, we will cover some lesser known and used (if used at all) methods which might come in handy one day, as well as some other techniques to keep in mind when using any kind of hooking method.

Finally, we refer the reader to production code which will be used on a daily basis to analyze thousands of malware samples.

Introduction

The following fragment is a brief explanation on Hooking, taken from Wikipedia [1].

```
In computer programming, the term hooking covers a range of techniques
used to alter or augment the behavior of an operating system, of
applications, or of other software components by intercepting function
calls or messages or events passed between software components. Code
that handles such intercepted function calls, events or messages is
called a "hook".
```

So, we've established that hooking allows us to alter the behaviour of existing software. Before we go further, following is a brief list of example uses of hooking.

- Profiling: How fast are certain function calls?
- Monitoring: Did we send the correct parameters to function X?
- ..?

A more comprehensive list of why one would want to hook functions can be found here [1] [2].

With this list, it should be clear why API hooking is useful. That being said, it's time to move on to API hooking itself.

A small note to the reader, this post does **not** cover **breakpoints**, **IAT Hooking**, etc. as that is an entire blogpost on its own.

Basic API Hooking

The easiest way of hooking is by inserting a jump instruction. As you may or may not already know, the x86 instruction set has a variable length instruction size (that is, an instruction can have a length between one byte and 16 bytes, at max.) One particular instruction, the unconditional jump, is five bytes in length. One byte represents the opcode, the other four bytes represent a 32bit relative offset. (Note that there is also an unconditional jump instruction which takes an 8bit relative offset, but we will not use that instruction in this example.)

So, if we have two functions, function A and function B, how do we redirect execution from function A to function B? Well, obviously we will be using a jump instruction, so all there's left to do is calculate the correct relative offset.

Assume that function A is located at address *0×401000* and that function B is located at address *0×401800*. What we do next is, we determine the required relative offset. There is a difference of *0×800* bytes between the two functions, and we want to jump from function A to function B, so we don't have to worry about negative offsets yet.

Now comes the tricky part, assume that we have already written our jump instruction at address 0×401000 (function A), and that the instruction is executed. What the CPU will do is the following; first it will add the length of the instruction to the Instruction Pointer [3] (or Program Counter), the length of the jump instruction is five bytes, as we've established earlier. After this, the relative offset is added (the four bytes, or 32bits value, located after the opcode) to the Instruction Pointer. In other words, the CPU calculates the new Instruction Pointer like the following.

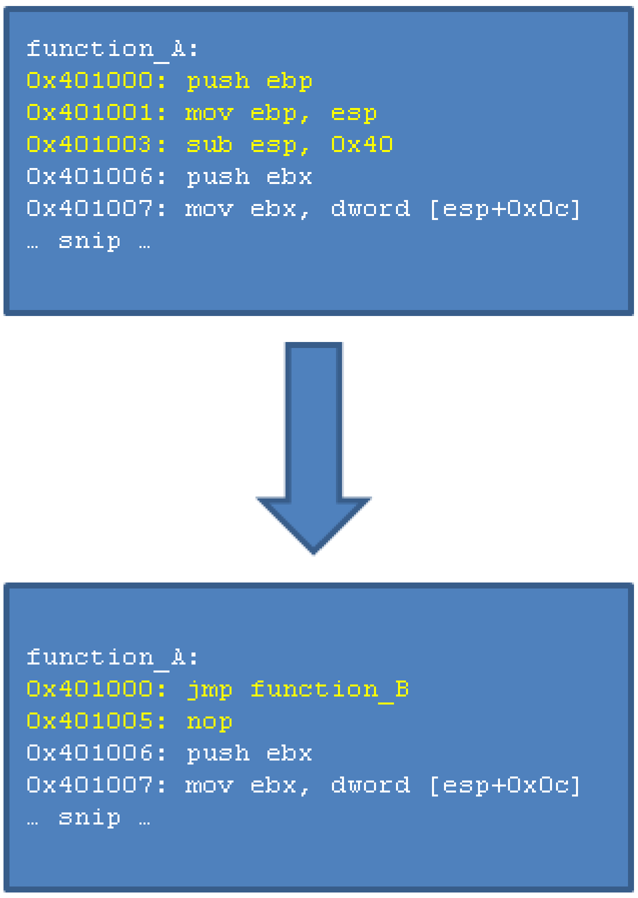
```
instruction_pointer = instruction_pointer + 5 + relative_offset;
```

Therefore, calculating the relative offset requires us to reverse the formula in the following way.

```
relative_offset = function_B - function_A - 5;
```

We subtract five because that's the length of the jump instruction which the CPU adds when executing this instruction, and function_A is subtracted from function_B because it's a *relative* jump; the difference between the addresses of function_B and function_A is 0x800 bytes. (E.g. if we forget to subtract function_A, then the CPU will end up at the address 0x401800 + 0x401000 + 5, which is obviously not desired.)

In assembly, redirecting function A to function B will look roughly like the following.



```
function_A:
0x401000: push ebp
0x401001: mov ebp, esp
0x401003: sub esp, 0x40
0x401006: push ebx
0x401007: mov ebx, dword [esp+0x0c]
… snip …
```

```
function_A:
0x401000: jmp function_B
0x401005: nop
0x401006: push ebx
0x401007: mov ebx, dword [esp+0x0c]
… snip …
```

Before the hook you see a few original instructions, however, after hooking, they are overwritten by our jump instruction. As you can see, the first three original instructions take a length of six bytes in total (i.e. you can see that the *push ebx* instruction is located at address 0x401006.) Our jump instruction uses only five bytes, which leaves us one extra byte, we have overwritten this byte with a *nop* instruction (an instruction that does nothing.)

The instructions that have been overwritten are what we call **stolen bytes**, in the following paragraph we'll go into more detail about them.

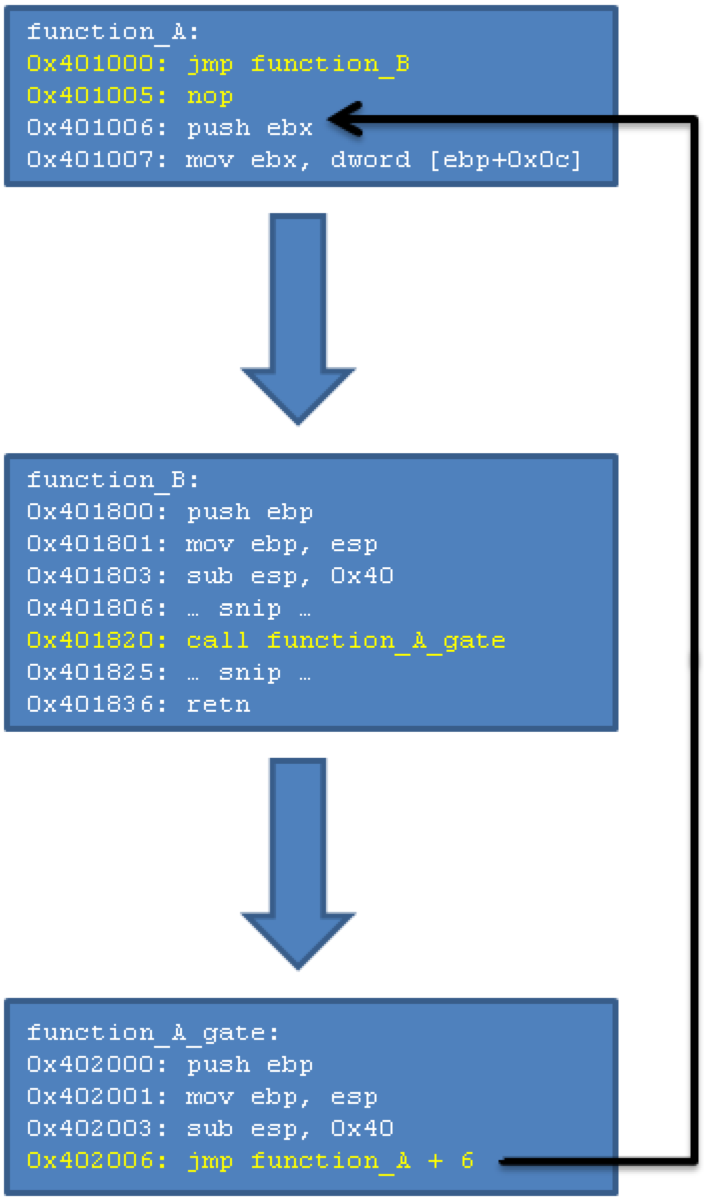
Trampolines

So, we've hooked function A and redirected it to function B. However, what happens when we want to execute the original function A, without executing the hook? In order to do this, we have to make a so-called **trampoline** function.

The following snippet shows a simple example of using a Trampoline to execute the original function from the hooking function, where *function_A_trampoline* denotes the Trampoline to function A (the function which is being hooked.)

```c
// this is the hooked function
void function_A(int value, int value2);

// this is the Trampoline with which we can call
// function_A without executing the hook
void (*function_A_trampoline)(int value, int value2);

// this is the hooking function which is executed
// when function_A is called
void function_B(int value, int value2)
{
    // alter arguments and execute the original function
    function_A_trampoline(value + 1, value2 + 2);
}
```

In the hooking example which we have just discussed, we have overwritten the first five bytes of function A. In order to execute our original function, without the hook, we will have to execute the bytes which we have overwritten when installing the hook, and then jump to the address of function A plus a few bytes (so we will *skip* the jump.) This is exactly what happens in the code snippet above, but you don't see that in the C source, because it's all abstracted away. Anyway, an image says more than a thousand words, so.. here's an image which shows the internal workings of the Trampoline.



```
function_A:
0x401000: jmp function_B
0x401005: nop
0x401006: push ebx
0x401007: mov ebx, dword [ebp+0x0c]
```

```
function_B:
0x401800: push ebp
0x401801: mov ebp, esp
0x401803: sub esp, 0x40
0x401806: … snip …
0x401820: call function_A_gate
0x401825: … snip …
0x401836: retn
```

```
function_A_gate:
0x402000: push ebp
0x402001: mov ebp, esp
0x402003: sub esp, 0x40
0x402006: jmp function_A + 6
```

In the image you see the following execution flow; function A is called, the hook is executed and therefore execution is now in function B. Function B does some stuff, but at address 0×401820 it wants to execute the original function A without the hook, this is where the Trampoline kicks in. Many, many words could be written about the Trampoline, but the image explains it all. The Trampoline consists of exactly two parts; the original instructions and a jump to function A, but past the hook. If you grab the image from the Basic API Hooking section, then you can see that the instructions which were overwritten, are now located in the Trampoline. The jump in the Trampoline is calculated simply using the formula which we discussed earlier, however, in this particular case the addresses and offsets are a bit different, so the exact formula goes as following.

```
relative_offset = (function_A_trampoline + 6) - (function_A + 6) - 5;
```

Note carefully that we jump **from** 0×402006 (function_A_trampoline + 6) **to** 0×401006 (function_A + 6.) These addresses can be verified in the image above. Nothing special, except for the fact that we will end up with a negative relative offset, but that doesn't give us any problems (i.e. the CPU does all the hard work of representing the correct negative relative offset.)

This is actually all there is to know about basic API hooking, so we will now discuss some more advanced methods. In a later chapter, Constructing Trampolines, we will go into more detail regarding how to construct a Trampoline.

Advanced Hooking Methods

We have seen Basic API Hooking and Trampolines. However, because this method of hooking is so simple (inserting a jump instruction), it is also really easy to detect. Therefore we will talk about some more *advanced methods* as well. Besides that, we will also give an introduction to Hooking C++ Class Methods.

Detection of the example in the Basic API Hooking method goes as following.

```
if(*function_A == 0xe9) {
    printf("Hook detected in function A.\n");
}
```

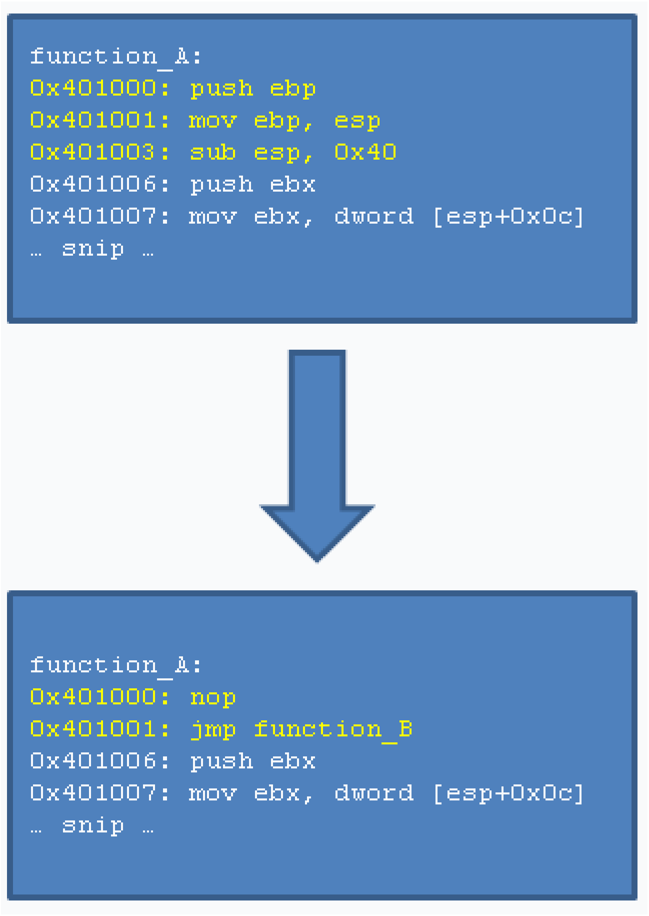
This is because **0xe9** is the opcode for a jump with a 32bit relative offset. Depending on the software which we hook, it may or may not detect such hook. In any case, we will discuss various methods which try to bypass such detection algorithms. (Note that software such as GMER [4] detects all types of hooking methods, because it checks virtual memory against the physical image on the disk.)

Method I: Prepend a Nop

This is a really simple workaround, and works with any of the methods which will follow as well.

Basically, instead of writing for example the jump instruction at function A, we will first write a *nop* instruction (an instruction which does nothing), followed by the jump instruction. When applying this technique, do note that that the jump instruction is now at address 0x401001 (function_A + 1), which alters the relative offset by one.

An image to show this technique follows.



```
function_A:
0x401000: push ebp
0x401001: mov ebp, esp
0x401003: sub esp, 0x40
0x401006: push ebx
0x401007: mov ebx, dword [esp+0x0c]
… snip …
```

```
function_A:
0x401000: nop
0x401001: jmp function_B
0x401006: push ebx
0x401007: mov ebx, dword [esp+0x0c]
… snip …
```

Because the first instruction at function A is now a *nop* instruction, and not a jump instruction, we'd have to rewrite the detection method to something like the following in order to detect it.

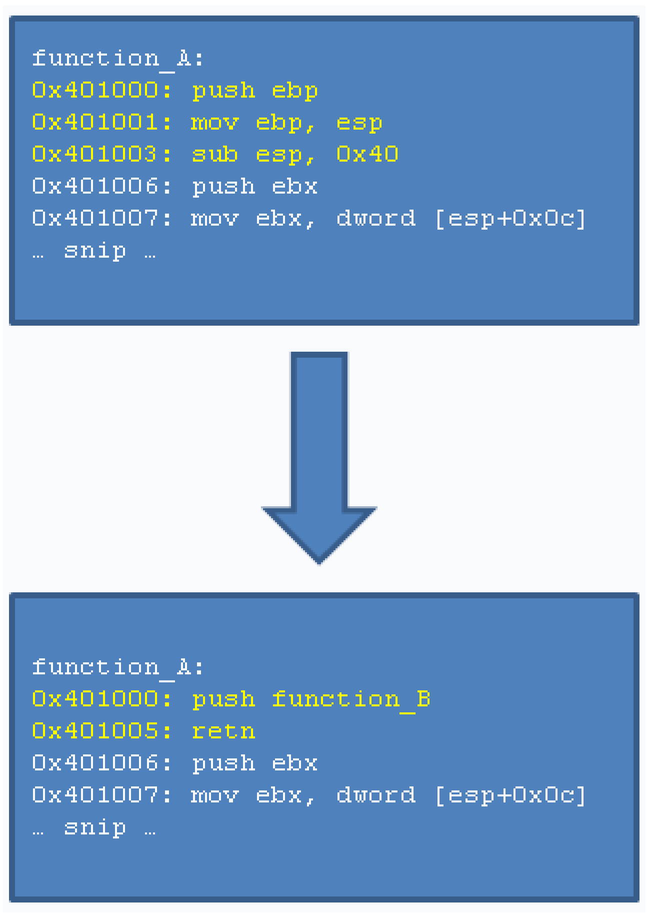
```
unsigned char *addr = function_A;
while (*addr == 0x90) addr++;
if(*addr == 0xe9) {
    printf("Hook detected in function A.\n");
}
```

Basically it skips any nop instructions, which have **0×90** as opcode, and checks for a jump after all the nop instructions.

Method II: Push/Retn

The *push* instruction pushes a 32bit value on the stack, and the *retn* instruction pops a 32bit address off the stack into the Instruction Pointer (in other words, it starts execution at the address which is found at the top of the stack.)

This method is six bytes in total, and looks like the following. Note that the push instruction takes an *absolute* address, not a relative one.



```
function_A:
0x401000: push ebp
0x401001: mov ebp, esp
0x401003: sub esp, 0x40
0x401006: push ebx
0x401007: mov ebx, dword [esp+0x0c]
… snip …
```

```
function_A:
0x401000: push function_B
0x401005: retn
0x401006: push ebx
0x401007: mov ebx, dword [esp+0x0c]
… snip …
```

Detection of this technique might look like the following, although keep in mind that prepending nop instructions, or placing a nop between the push and retn instruction will not be detected by the following code.

```
unsigned char *addr = function_A;
if(*addr == 0x68 && addr[5] == 0xc3) {
    printf("Hook detected in function A.\n");
}
```

As you've hopefully figured out by now, **0×68** is the opcode for the push instruction, and **0xc3** the opcode for the retn instruction.

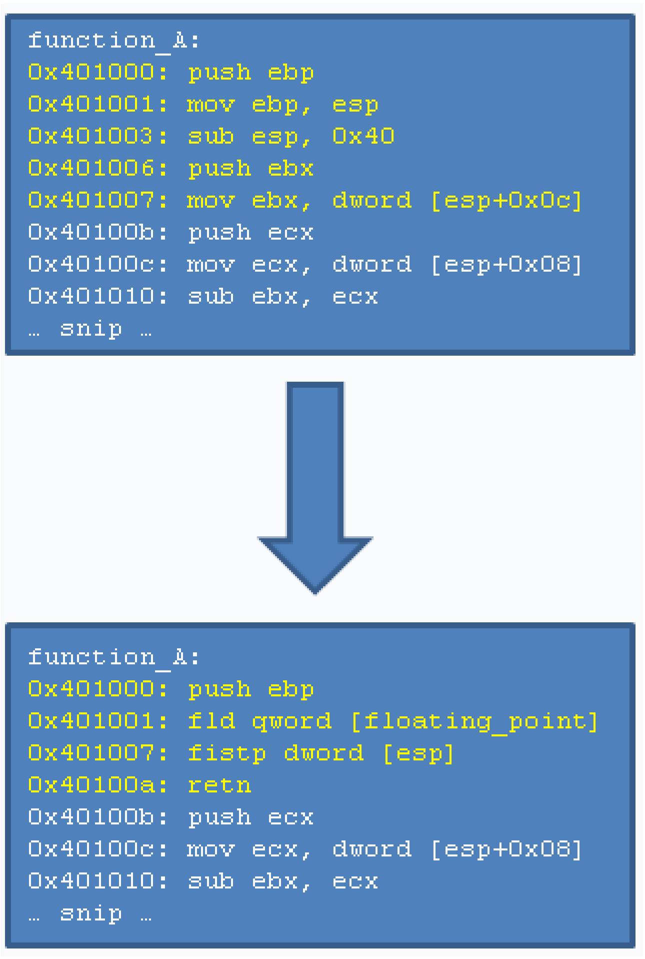
Method III: Floating Points

Whereas the methods discussed so far concentrated on *normal* x86 instructions, there are also some interesting methods involving floating points, and whatnot.

This example is like the push/retn method, we push a dummy value on the stack, we then overwrite this dummy value with the real address, and then we return.

What is interesting about this technique is the following; instead of storing the address to jump to as a 32bit address, we store it as a 64bit floating point. We then read it (using the *fld* instruction), and translate it to a 32bit value (using the *fistp* instruction.)

The following picture demonstrates the technique, the hook uses eleven bytes, so that's a little bit more than the previous methods, but it's still a nice method. Also note that **floating_point** is a pointer to the 64bit floating point value which contains the address of our hook function.



```
function_A:
0x401000: push ebp
0x401001: mov ebp, esp
0x401003: sub esp, 0x40
0x401006: push ebx
0x401007: mov ebx, dword [esp+0x0c]
0x40100b: push ecx
0x40100c: mov ecx, dword [esp+0x08]
0x401010: sub ebx, ecx
… snip …
```

```
function_A:
0x401000: push ebp
0x401001: fld qword [floating_point]
0x401007: fistp dword [esp]
0x40100a: retn
0x40100b: push ecx
0x40100c: mov ecx, dword [esp+0x08]
0x401010: sub ebx, ecx
… snip …
```

Constructing the floating point is fairly easy, and goes as following.

```
double floating_point_value = (double) function_B;
```

Where function B is, as with the previous examples, our hook function.

Calling the original function is done through a Trampoline, just like with the other methods. (Except in this case the Trampoline will have to contain atleast the instructions which are found in the first eleven bytes of the function.)

Obviously this is only the tip of the iceberg, there are many floating point instructions which might be useful for this situation. E.g. you could calculate the hook address by multiplying a value to π (Pi, which can be obtained using the *fldpi* instruction.)

Method IV: MMX/SSE

This technique is similar to that of hooking using Floating Points. However, instead of using Floating Points, we use the MMX/SSE x86 extensions here.

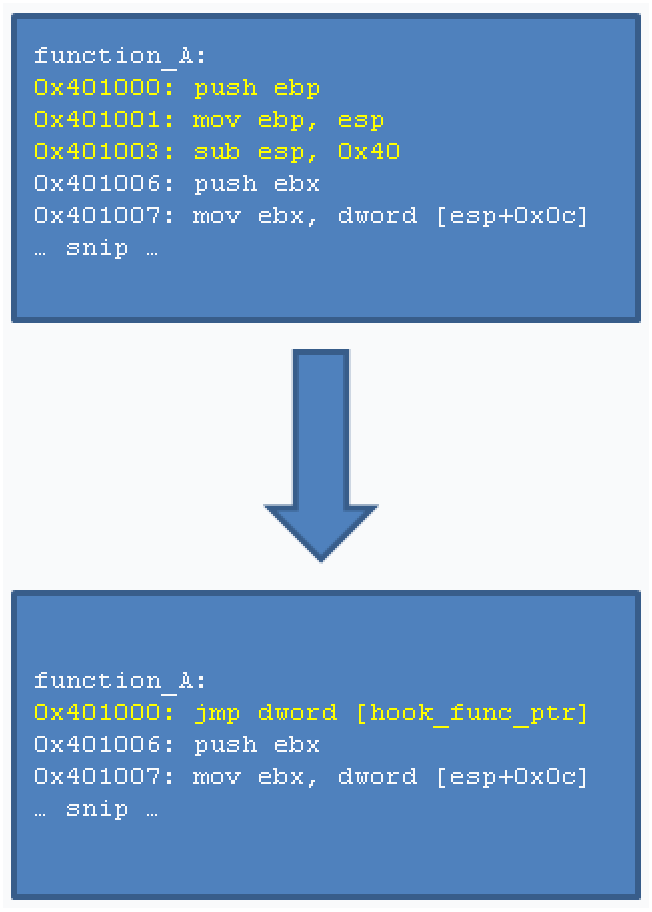
Both techniques use, like the Floating Points technique, the push/retn method. The first method involves the MMX instructions, particularly the **movd** instruction. It allows, just like the *fistp* instruction (a Floating Points instruction) to read a value from memory, and store a value to the stack. The second method, using the SSE instructions, utilizes the same **movd** instruction. The only difference between the two methods is the fact that MMX instructions operate on 64bit registers, whereas the SSE instructions operate on 128bit registers. (Although this is not interesting in our case, because the *movd* instruction allows reading and writing of 32bit values.)

Either way, as this technique is exactly the same as the Floating Points one, except for the instructions that are being used, there is no image (no need to flood this article with images..)

Method V: Indirect Jump

An indirect jump is basically saying, jump to the address which can be found *here*. In the Basic API Hooking section we introduced relative jumps, they are direct jumps. Indirect jumps are more like the Push/Retn method.

An indirect jump is six bytes in length, and an example hook looks like the following.



```
function_A:
0x401000: push ebp
0x401001: mov ebp, esp
0x401003: sub esp, 0x40
0x401006: push ebx
0x401007: mov ebx, dword [esp+0x0c]
… snip …
```

```
function_A:
0x401000: jmp dword [hook_func_ptr]
0x401006: push ebx
0x401007: mov ebx, dword [esp+0x0c]
… snip …
```

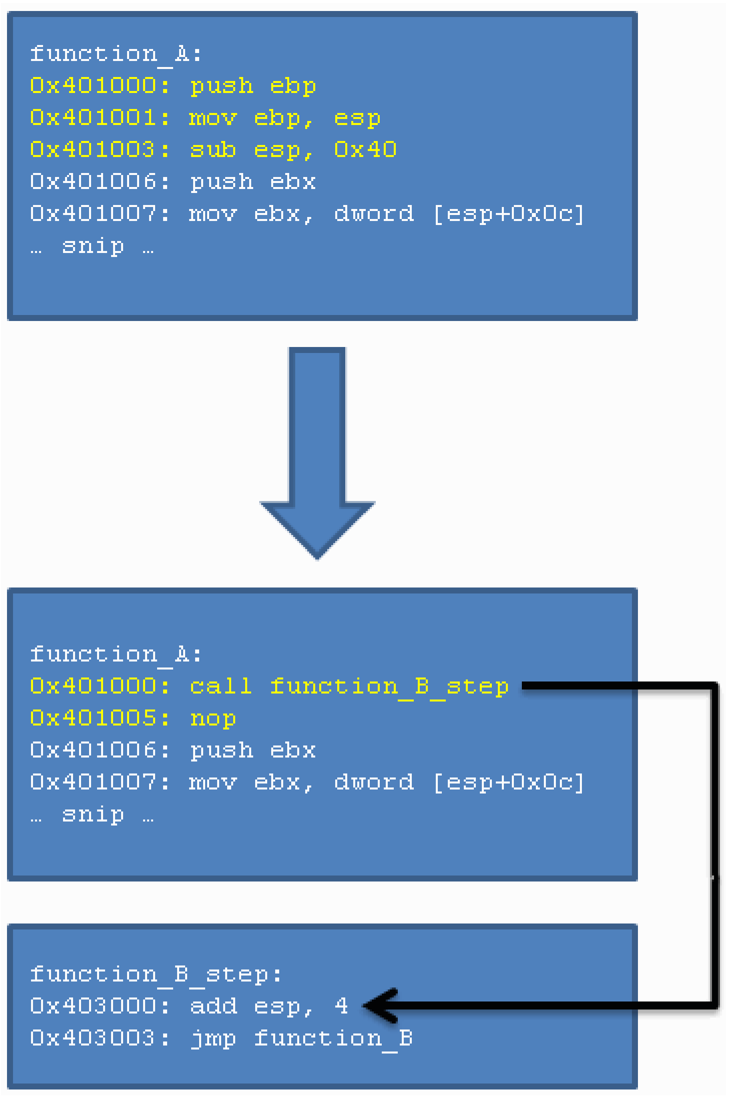
Note that the **hook_func_ptr** denotes an address at which the address of our hooking function (e.g. function B) can be found.

Method VI: Call Instruction

Whereas all of the previous hooking methods jump right into the hooking function itself (e.g. function B), this hooking method needs an additional step. This is because the *call* instruction jumps to a specified address after pushing a return address on the stack (the return address being the current Instruction Pointer plus the length of the instruction.)

Because there is now an additional return address on the stack, we will first have to pop this address off the stack, otherwise our stack will be corrupted. (E.g. the hooking function will read incorrect parameters from the stack, because the stack pointer is incorrect.) We *pop* this address off the stack by adding four to the stack pointer (when the address is *pushed* onto the stack, the stack pointer is first decremented by four, then the address is written to the address pointed to by the updated stack pointer.) After the address has been popped off the stack, we jump to the hooking function.

This technique works for both direct and indirect variants of the call instruction. Following is an imagine which represents the technique.



```
function_A:
0x401000: push ebp
0x401001: mov ebp, esp
0x401003: sub esp, 0x40
0x401006: push ebx
0x401007: mov ebx, dword [esp+0x0c]
… snip …
```

```
function_A:
0x401000: call function_B_step
0x401005: nop
0x401006: push ebx
0x401007: mov ebx, dword [esp+0x0c]
… snip …
```

```
function_B_step:
0x403000: add esp, 4
0x403003: jmp function_B
```

Other Methods

Besides the methods which have been discussed so far, there are also a few other methods, with specific purposes. They might come in handy for some situations, you never know.. 😉

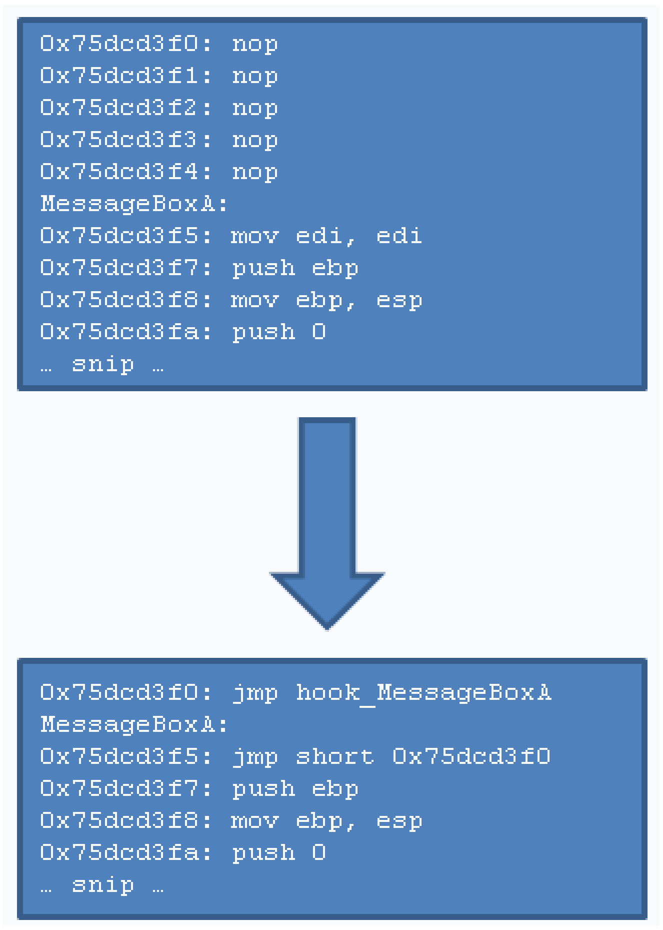
Other Methods I: Hotpatching

This is a method specific for software compiled with the Microsoft Visual C++ Compiler, which have the Hotpatching flag enabled (this is the case for plenty of dlls, such as *user32.dll*.)

If a function accepts a so-called Hotpatch, then it has been prepared in a certain way; the first instruction of the function is a **mov edi, edi** instruction (which is two bytes in length) and there are five **nop** instructions before the function itself. This allows one to place a *short* jump (one that takes an 8bit relative offset, and is two bytes in length) at the function address (to overwrite the *mov edi, edi* instruction) and a normal jump with a 32bit relative offset at the place of the nop instructions.

And yet again, that's all there is to this technique. Note that, instead of Hotpatching such function, it is also possible to hook the function using one of the other methods explained in this article by placing the hook on the address *function+2*, where two denotes the size of the instruction inserted for Hotpatching. (This way one could still apply a Hotpatch, even though we already hook the function with one of our favourite methods.)

An image representing Hotpatching is as follows, with *MessageBoxA* being the function to hook, and *hook_MessageBoxA* the function where execution goes (see it as MessageBoxA = function A, hook_MessageBoxA = function B.)



```
0x75dcd3f0: nop
0x75dcd3f1: nop
0x75dcd3f2: nop
0x75dcd3f3: nop
0x75dcd3f4: nop
MessageBoxA:
0x75dcd3f5: mov edi, edi
0x75dcd3f7: push ebp
0x75dcd3f8: mov ebp, esp
0x75dcd3fa: push 0
… snip …
```

```
0x75dcd3f0: jmp hook_MessageBoxA
MessageBoxA:
0x75dcd3f5: jmp short 0x75dcd3f0
0x75dcd3f7: push ebp
0x75dcd3f8: mov ebp, esp
0x75dcd3fa: push 0
… snip …
```

Other Methods II: C++ Class Methods

This method is regarding the hooking of C++ Class Methods. These functions use the so-called *__thiscall* [5] calling convention (or, atleast they do on windows.)

This particular calling convention stores the object information (which can be referenced through the *this* variable in class methods) in the *ecx* register before calling a class method. In other words, if we want to hook a class method, it needs some special attention.

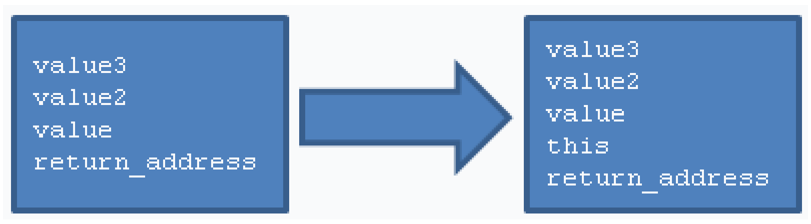
For further explanation we will be using the following code snippet, which defines a function (function_A) which we want to hook.

```cpp
class A {
public:
    void function_A(int value, int value2, int value3) {
        printf("value: %d %d %d\n", value, value2, value3);
    }
};
```

Besides that, because we want to hook C++ functions using regular C functions, we will be interested in having the *this* pointer as first parameter to our hook function. An example of our hooking function looks like the the following, including trampoline (which will be discussed later as well.) Note that we use the variable name *self* instead of *this*, because *this* is a reserved keyword in C++.

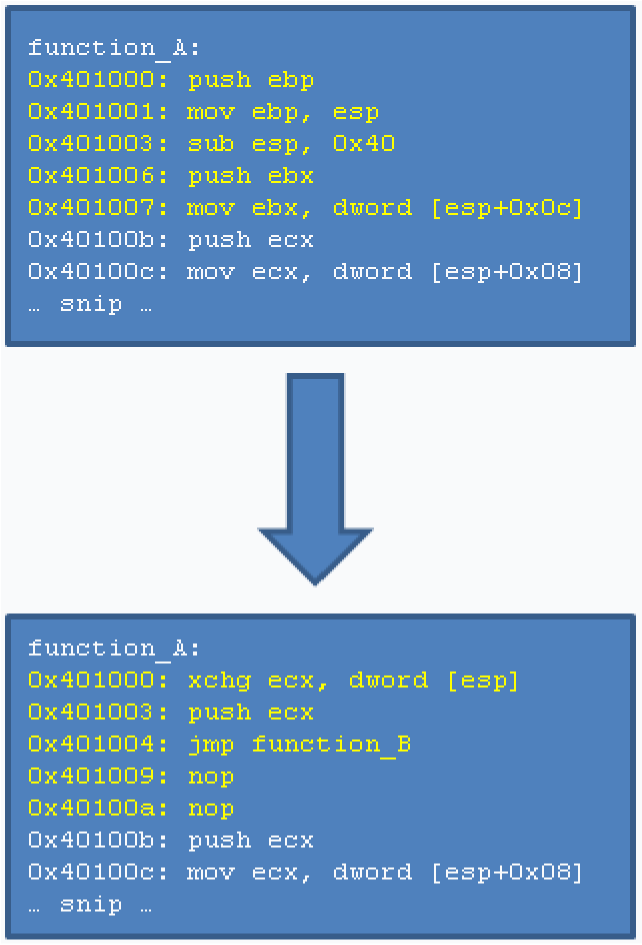
```cpp
void (*function_A_trampoline)(void *self, int value,
    int value2, int value3);

void function_B(void *self, int value,
    int value2, int value3)
{
    return function_A_trampoline(self, value + 1,
        value + 2, value + 3);
}
```

In order to be able to hook C++ Class Methods from a normal C source we will have to alter the stack a bit, because we have to squeeze the *this* pointer into it. The following image represents the stack layout when function_A is called, followed by the layout how we want it when function_B is called (the hooking function.) Note that *ecx* contains the *this* pointer, as outlined before, and the fact that the top of the stack is at the address on which *return_address* is located.



Fortunately for us, we can squeeze the *this* pointer into the stack in exactly two instructions, which is pretty neat. The first step is exchanging the *this* pointer (the *ecx* register) and the top of the stack (the *return_address*.) After this swap we have the *this* pointer at the top of the stack, and the *return_address* in the *ecx* register. From here on we can simply push the *ecx* register onto the stack, and the stack will look exactly like what we wanted (see the last image.)

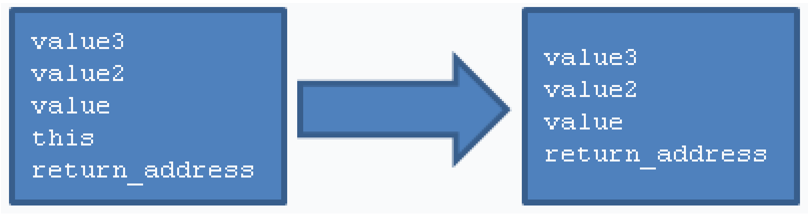
Following is the assembly representation of hooking a C++ class method.



```
function_A:
0x401000: push ebp
0x401001: mov ebp, esp
0x401003: sub esp, 0x40
0x401006: push ebx
0x401007: mov ebx, dword [esp+0x0c]
0x40100b: push ecx
0x40100c: mov ecx, dword [esp+0x08]
… snip …
```

```
function_A:
0x401000: xchg ecx, dword [esp]
0x401003: push ecx
0x401004: jmp function_B
0x401009: nop
0x40100a: nop
0x40100b: push ecx
0x40100c: mov ecx, dword [esp+0x08]
… snip …
```

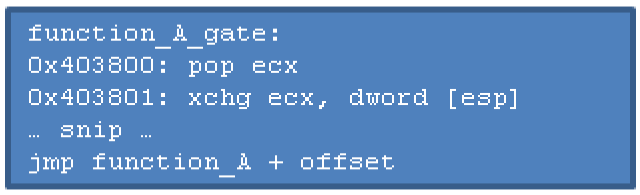
So that's the hooking part. However, we have to adjust our Trampoline as well, because the Trampoline accepts a *this* pointer as first argument. The stack of what we have and what we want is as follows. (The *this* value should obviously be stored into the *ecx* register afterwards.)



```
value3
value2
value
this
return_address
```

```
value3
value2
value
return_address
```

We do exactly the opposite of what we did to hook the function; we first pop the *return_address* off the stack, so the top of the stack points to *this*, and the *ecx* register is loaded with the *return_address*. Now we swap the top of the stack and the *ecx* register, after which the stack looks as we want, and the *ecx* register is loaded with the *this* pointer. The following image shows the Trampoline, although the instructions from function A have been omitted (i.e. this image only shows what's special about C++ Class Method Trampoline, not what we discussed earlier at Trampolines.)

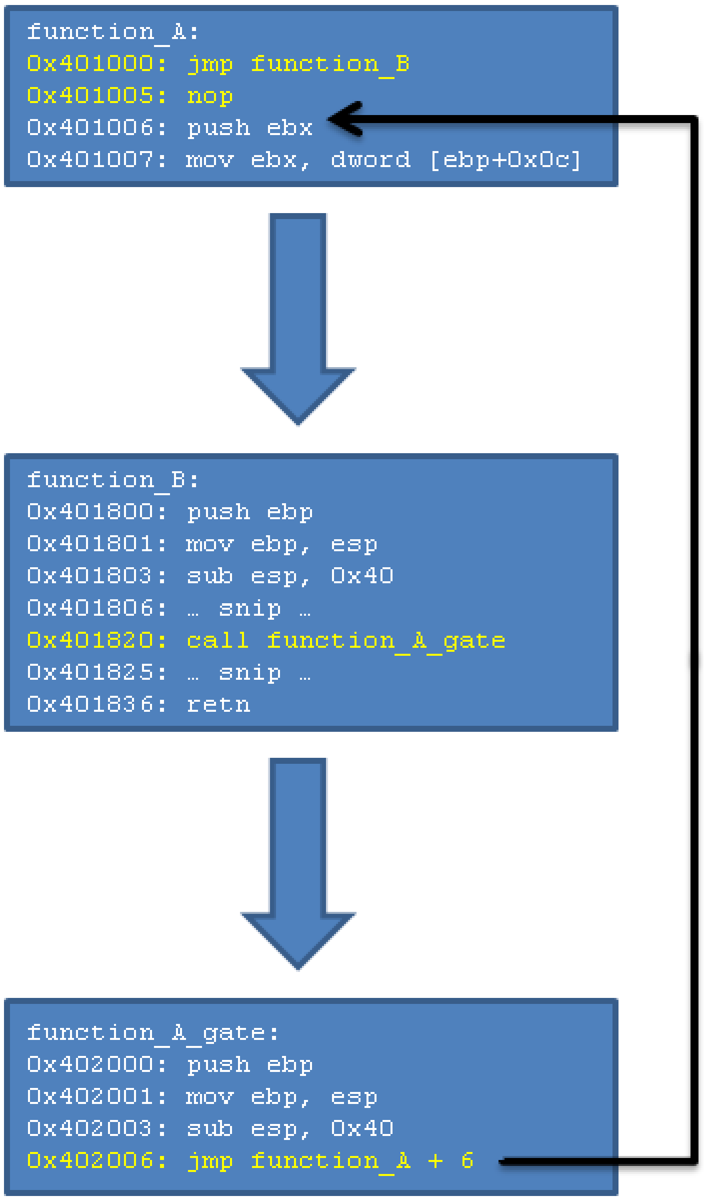


```
function_A_gate:
0x403800: pop ecx
0x403801: xchg ecx, dword [esp]
… snip …
jmp function_A + offset
```

Constructing Trampolines

In an earlier section, Trampolines, we discussed how a Trampoline should be made. In this section we will discuss some edge-cases, which should be taken into account when constructing a Trampoline.

The basics of constructing a Trampoline are as follows. We have a function (function A) which we want to hook, and we have a hooking method. We will use the simplest hooking method now; the direct jump, which is five bytes in size. As we will be overwriting the first five instructions of function A, we will have to take atleast the instructions which are located in the first five bytes of this function. However, it is possible that the last instruction in these five bytes is fairly long, e.g. it spans over byte three upto the sixth byte, which is the case in the example we used earlier (here is the image again, so you can see it again.)



```
function_A:
0x401000: jmp function_B
0x401005: nop
0x401006: push ebx
0x401007: mov ebx, dword [ebp+0x0c]
```

```
function_B:
0x401800: push ebp
0x401801: mov ebp, esp
0x401803: sub esp, 0x40
0x401806: … snip …
0x401820: call function_A_gate
0x401825: … snip …
0x401836: retn
```

```
function_A_gate:
0x402000: push ebp
0x402001: mov ebp, esp
0x402003: sub esp, 0x40
0x402006: jmp function_A + 6
```

As you can see, the third instruction also uses the sixth byte, due to this we cannot simply copy the first five bytes, but instead we have to copy the entire instruction. In order to do this we use a so-called *LDE*, which is short for **Length Disassembler Engine**. An LDE has the ability to calculate the length of an instruction (usually by doing some magic with a pre-defined table containing information about each opcode.)

As an LDE can calculate the length of an instruction for us, we can simply keep getting the length of instructions until we've found enough instructions which account for atleast five bytes. In the example image above it took us three instructions to get to a length of six, after which we have found enough instructions for the Trampoline.

This was the easy part, because the instructions which we found didn't contain any branches. However, any jump, call or return instructions need some special attention as well. To start off, jump and call instructions need to point to the same address in the Trampoline as they did in the original function, although this is not too hard if you use like two formulas (one to obtain the address of a jump or call instruction, and one to calculate the relative offset for the jump or call instruction which will be placed in the Trampoline.) Also note that any jumps that have an 8bit relative offset, should be turned into jump instructions with 32bit relative offsets (it is quite unlikely that the relative offset between the Trampoline and the original function is within the range of an 8bit offset.)

This is brings us to the last point; it is sometimes possible that there is simply **not enough space for our hooking method**. This happens when there is either an unconditional jump instruction, or a return instruction. Such situation occurs when we've simply hit the end of a function, e.g. a totally different function might start at the address after a return instruction, so we can't overwrite such memory. The same goes for an unconditional jump, this function won't continue after an unconditional jump, so overwriting behind this instruction would result in *undefined behaviour*.

Multiple Layers of Hooks

It is perfectly possible to have multiple layers of hooks, i.e. multiple hooks on one function. There is only one condition which must apply; when installing a new hook over an existing hook, then you will have to make sure that the required length of the new hook is equal to, or lower than, the length of the existing hook. For more explanation about this, refer to the Constructing Trampolines section. In any case, a normal direct jump with 32bit relative offset should suffice, as it's (normally) the hook with the shortest length in bytes.

Preventing Hook Recursion

Last, but not least, some thoughts regarding Hooking Recursion. Sometimes it is possible that within a hooking function, function B in our examples, one uses functions such as logging to a file. However, what happens if such function ends up in the same, or another, hook of ours? This might lead to a recursive hook, i.e. a hook might keep hooking itself.. or one hook leads to another hook. This is **not** what we want, besides that it is also very annoying (been there, done that.)

So, basically, what we will want to do to prevent such problem is keeping a *hook count*. Whenever a hook is executed, a check is done against the hook count (and, obviously, the *hook count* is increased when entering a hook.) This way we can tell the hooking engine two things. The first being, once a hook has occurred don't hook any lower hooks (e.g. if we hook the fwrite() function, and we call fwrite() in it's hook, then the hook should not be executed.) The second ability is to give a maximum recursion count, e.g. hook upto three hooks deep. Although this is still not desired functionality, usually.

Also, do note that, such hook count should be thread-specific. To solve this, the production code which will be shown in the Proof of Concept section stores the *hook count* in the segment pointed to by the *fs* register [6] (this section denotes the Thread Information Block, i.e. thread specific data.)

Proof of Concept

As Proof of Concept for today's post we will introduce the reader to Cuckoo Sandbox, a system for automated malware analysis. In Cuckoo's relatively new analysis component, we have deployed techniques from this post. For example, it still uses the direct 32bit jump, but it has a fairly nice engine for constructing Trampolines and it supports Prevention of Hook Recursion.

Current source can be found here, although it will be in here soon enough. For the implementation of methods described in this post, please refer to the hooking.c file.

References

1. Hooking – Wikipedia
2. API Hooking Revealed – CodeProject
3. Instruction Pointer / Program Counter – Wikipedia
4. Rootkit Detector – GMER
5. Thiscall Calling Convention – Nynaeve
6. Win32 Thread Information Block – Wikipedia

This entry was posted in **Uncategorized** by **jbremer**. Bookmark the **permalink [http://jbremer.org/x86-api-hooking-demystified/]** .

15 THOUGHTS ON "X86 API HOOKING DEMYSTIFIED"

Pingback: x86 API Hooking Demystified | ITSecurity | Scoop.it

Pingback: x86 API Hooking Demystified | Development & Security | Frishit Security | Scoop.it

Pingback: Suterusu Rootkit: Inline Kernel Function Hooking on x86 and ARM | Michael Coppola's Blog

Steve Maresca
on **February 15, 2013 at 1:19 pm** said:

Quite surprised that no one has commented yet!
Just wanted to say thanks : this is a truly awesome article.

Daniel
on **March 6, 2013 at 7:29 am** said:

Amazing article!
Thanks for the write up. Help me alot as a student.

amazing
on **March 10, 2013 at 1:21 am** said:

you forgot the hardware breakpoints technique. all of your methods require memory editing and that is detected (crc checks).

> jbremer
> on **March 27, 2013 at 10:48 am** said:
>
> This post was not targeted at Hardware Breakpoints. As you can see, the post is already long enough. A post on Hardware Breakpoints can easily fill pages of information as well, so I decided not to mix the two.

David
on **May 29, 2013 at 9:15 pm** said:

Hey, just a comment. In the first method you are adding a call to your gate function. That is screwing your stack by adding an extra return IP, so your hook function prototype must have an extra dummy parameter at the left (which contains the original return address), right? What I do in my hooks is to insert a jump to my gate function, so it's really transparent. At the end I just jump to the "remainder" which contains the stolen instructions from the prologue and jumps to the appropiate IP after the hook to continue with the original function. So I only use jumps 😉

Mansuro
on **June 10, 2013 at 11:54 am** said:

Hello, I don't understand what the following code does if (*function_A == 0xe9). What does *function_A return? is it the opcode of the function?

> jbremer
> on **November 3, 2013 at 9:47 am** said:
>
> In this context it means the first byte of the first instruction of the function. This byte is (usually) something like "push ebp" (or "\x55".)