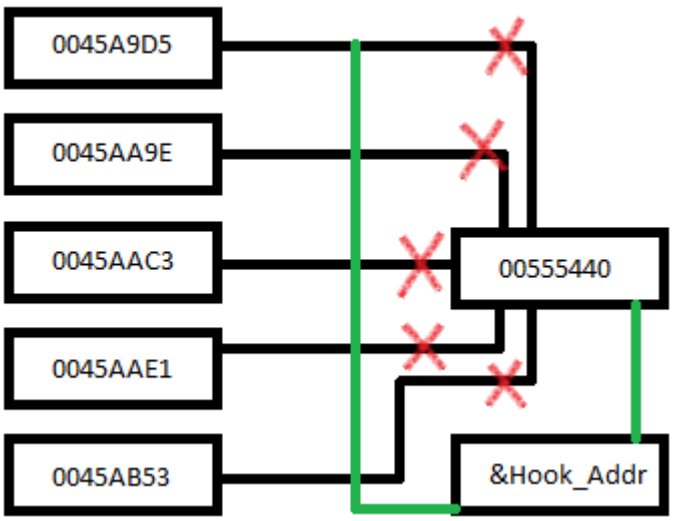
This post will focus on an interesting technique that I thought of this past week. I've noticed that a lot of API hooking libraries, or techniques in general, rely on replacing the prologue to the function with a jump into the hook (e.g. writing in E9 XXXXXXXX and jumping to a trampoline). This technique is quick, effective, and reliable in most scenarios. However, it is also easily detectable since all someone has to do is just check the first five bytes of a function to see whether there is a jump or not. The idea that I thought of this past weekend (although I'm sure I'm not the only one) was to replace the **references** to a function to that of the hook. For example, suppose I have a function at .text:00555440

```
.text:00555440                  push    esi
.text:00555441                  mov     si, [esp+4+arg_0]
.text:00555446                  test    si, si
.text:00555449                  jl      loc_5554FE
.text:0055544F                  mov     eax, [ecx+0A4h]
.text:00555455                  movsx   edx, si
...
```

The actual content is not important. Now again suppose that this function has five places that it's being called from: 0045A9D5, 0045AA9E, 0045AAC3, 0045AAE1, 0045AB53.

```
...
.text:0045AA9C                  mov     ecx, esi
.text:0045AA9E                  call    sub_555440
.text:0045AAA3                  jmp     loc_45AB3A
...
.text:0045AADF                  mov     ecx, esi
.text:0045AAE1                  call    sub_555440
.text:0045AAE6                  jmp     short loc_45AB3A
...
```

and so on. My idea entails replacing all five call sub_555440 instructions to a call to the hooking function. Visually, it is something like this:



Where the green line denotes the new path of the calling functions. While easy to explain pictorially or in words, actually programming it is somewhat tedious. The main issue that is x86 has eight different ways on how a function can be called (reproduced below from here)

Opcode	Mnemonic	Description
E8 cw	CALL rel16	Call near, relative, displacement relative to next instruction
E8 cd	CALL rel32	Call near, relative, displacement relative to next instruction
FF /2	CALL r/m16	Call near, absolute indirect, address given in r/m16
FF /2	CALL r/m32	Call near, absolute indirect, address given in r/m32
9A cd	CALL ptr16:16	Call far, absolute, address given in operand
9A cp	CALL ptr16:32	Call far, absolute, address given in operand
FF /3	CALL m16:16	Call far, absolute indirect, address given in m16:16
FF /3	CALL m16:32	Call far, absolute indirect, address given in m16:32

The ones that I chose to focus on are the two most common ways that 32-bit programs call functions -- E8 cw or FF /2, the 32-bit near relative and near absolute calls. Replacing them involves parsing through the entire file in memory, section by section, and replacing references as they are found. For the actual implementation I cheated a bit and parsed a user-supplied section for the references, but it can be extended to do through all without any problem, just more computation time. The actual implementation looks like this

```
void replace_references(DWORD_PTR image_base, const char* section_name,
LPVOID original_addr, LPVOID hook_addr) {
    PIMAGE_DOS_HEADER dos_header = (PIMAGE_DOS_HEADER)image_base;
    PIMAGE_SECTION_HEADER section_header =
get_section_addr_by_name(image_base, section_name);
    BYTE* section_start = (BYTE*)(image_base + section_header-
>VirtualAddress);
    BYTE* section_end = section_start + section_header->SizeOfRawData;
    BYTE call_near_relative = 0xE8;
    BYTE call_near_absolute[] = {0xFF, 0x15};
    DWORD new_protections = PAGE_EXECUTE_READWRITE;
    DWORD old_protections = 0;
    for(section_start; section_start < section_end; ++section_start) {
        VirtualProtect(section_start, PROTECT_SIZE, new_protections,
&old_protections);
        if(*section_start == call_near_relative) {
            BYTE offset[] = {
                *(section_start + 0x4),
                *(section_start + 0x3),
                *(section_start + 0x2),
                *(section_start + 0x1)
            };
            DWORD_PTR relative_offset = (((offset[0] & 0xFF) << 24) |
((offset[1] & 0xFF) << 16) |
                ((offset[2] & 0xFF) << 8) | offset[3] & 0xFF) +
NEAR_PATCH_SIZE;
            if((section_start + relative_offset) == original_addr) {
                DWORD_PTR hook_offset = (BYTE*)hook_addr - section_start -
NEAR_PATCH_SIZE;
```

```
                patch_memory((section_start + 0x1), &hook_offset);
            }
        }
        else if(memcmp(section_start, call_near_absolute,
sizeof(call_near_absolute)) == 0) {
            BYTE offset[] = {
                *(section_start + 0x5),
                *(section_start + 0x4),
                *(section_start + 0x3),
                *(section_start + 0x2)
            };
            PDWORD_PTR absolute_addr = (PDWORD_PTR)(((offset[0] & 0xFF) <<
24) | ((offset[1] & 0xFF) << 16) |
                ((offset[2] & 0xFF) << 8) | offset[3] & 0xFF);
            __try {
                if(*absolute_addr == (DWORD_PTR)original_addr)
                    patch_memory(absolute_addr, &hook_addr);
            }
            __except(translate_exception(GetExceptionCode(),
GetExceptionInformation())) {
                //Dereferenced bad pointer
            }
        }
        VirtualProtect(section_start, PROTECT_SIZE, old_protections, NULL);
    }
}
```

There is quite a bit to explain here. First off, the function takes four parameters -- the base of the executable (obtained with GetModuleHandle(NULL)), a string to the section name to perform the replacements in, the desired address to be replaced, and lastly, the address of the hooking function that will replace it. The function then goes through the section byte by byte checking for either the E8 or FF 15 opcodes. Once these are found it is time to check and see if the call is to the correct place. For a relative near call this is pretty simple. The offset to the destination is the next four bytes - 0x5. All that needs to be done is to get those four bytes, change their endianness, and see whether the call leads to the address that is to be replaced. For an absolute near call, the process gets a bit tricky. The absolute address of the function to be called is stored in a 32-bit register. This means that the register needs to be read and dereferenced. The contents inside that register then need to be changed to that of the address of the hooking function. For example,

```
004010F1  |. FF15 C0204000  CALL DWORD PTR DS:[<&USER32.MessageBoxA>>;
\MessageBoxA
```

Here [004020C0] will contain 757CFEAE which is the address of USER32.MessageBoxA. To hook the call, [004020C0] should be replaced with the address of the hook. One issue the absolute address that this function finds may not in fact be an absolute address. Since the function simply reads a byte (or two) at a time for a match, the absolute address may not actually be an address. It may be two different instructions that coincidentally got interpreted as one by this function. Short of writing a disassembler, there is not much that can be done about this since instructions are variable length on the x86 architecture. It will simply suffice to catch the

inevitable access violation error that will result from trying to dereference an invalid pointer. However, if a valid address if found (for a relative or absolute call), patching it is very simple

```
void patch_memory(LPVOID patch_addr, LPVOID replacement_addr) {
    DWORD old_protections = 0;
    VirtualProtect(patch_addr, sizeof(DWORD_PTR), PAGE_EXECUTE_READWRITE,
&old_protections);
    memmove(patch_addr, replacement_addr, sizeof(DWORD_PTR));
    VirtualProtect(patch_addr, sizeof(DWORD_PTR), old_protections, NULL);
}
```

That is about it for how it works. The usage of it is pretty straightforward as well. It is easy to hook a function by an absolute address or by its name. Below is a code snippet that hooks MessageBoxA and also hooks a function at 00401000. The hook at 00401000 is more of a specialized exception to how hooking an unknown function by address should be done. Typically it would be done as an offset from an image or section instead of a full absolute address. The absolute version was shown simply to save lines of code and demonstrate the technique.

```
       DWORD_PTR image_base = (DWORD_PTR)GetModuleHandle(NULL);
        //Consider a new thread for actual use
        replace_references(image_base, ".text", &MessageBoxA,
&MessageBoxA_hook);
        if(image_base == 0x00400000)
            replace_references(image_base, ".text", generate_number,
&generate_number_hook);
        else
            MessageBox(NULL, L"Executable did not load at 0x00400000",
L"Error", MB_ICONEXCLAMATION);
```

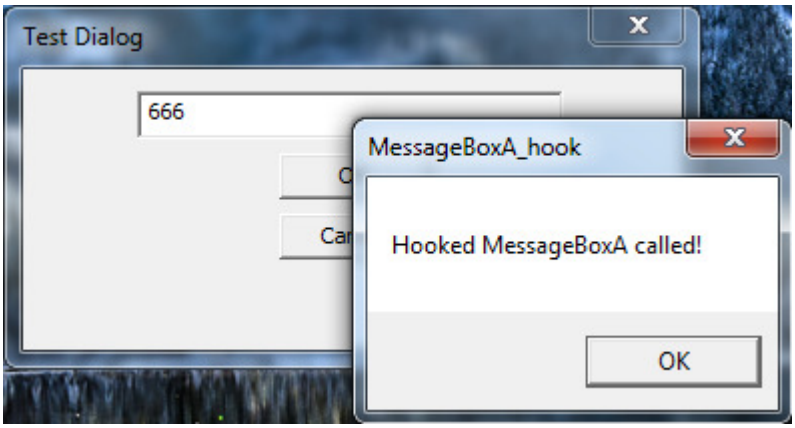
Here the base of the executable is retrieved and MessageBoxA and an "unknown" function in the executable are hooked. The executable is provided in the zip and should always load at 00400000. Again, this was done to simply demonstrate the technique and is not the way it should be done as programs can have randomized base addresses. The two hooking functions look pretty standard

```
typedef int (__cdecl *pgenerate_number)(int a, int b);
pgenerate_number generate_number = (pgenerate_number)(0x00401000); //Example
compiled to load at a fixed base address
//Otherwise an offset from an image/section base would be needed

int __cdecl generate_number_hook(int a, int b) {
    return 666;
}

int WINAPI MessageBoxA_hook(HWND hwnd, LPCSTR lpText, LPCSTR lpCaption, UINT
uType) {
    __asm pushad
    MessageBoxA(hwnd, "Hooked MessageBoxA called!", "MessageBoxA_hook",
uType);
    __asm popad
    return MessageBoxA(hwnd, lpText, lpCaption, uType);
}
```

The hooks are designed for the sample application, but the technique extends to any application with minimal modifications (none in the case of MessageBoxA). Below is a screenshot of them in action:



The hook for MessageBoxA is called and the one for the "unknown" function is called too to always return "666".
It should definitely be noted that this technique does have its downsides. The first one was already mentioned previously -- the fact that you cannot really tell where you are exactly when you match what could be a call. This is a problem that cannot be fixed easily, it really would require writing a large part of a disassembler to fix. The second major problem is that it may be computationally expensive. There may be a lot of sections or the section sizes may be very large. The best fix would be to do a multithreaded replacement on the desired sections. The application will not freeze during the scan, but the functions may be called before the scanner arrives to replace the addresses. The third problem is that modules may be loaded after yours that call the functions, thus they will not be patched. This is more of a simple fix and can be remedied by hooking LoadLibrary/LoadLibraryEx and patching the addresses before returning to the application. The fourth, and what I think is the largest problem, is that this technique is extremely difficult to defeat. Doing something like

```
MOV EAX, [004020C0]
NOP
CALL DWORD PTR:[EAX]
```

completely defeats this method. This can be minimized by heuristically looking at the instructions being scanned, but the obfuscations can get much more complex to the point of it being impractical. Alternatively, any address that is purely calculated at runtime will not be spotted by this. While being an interesting technique, I can understand why I would not find any current existing implementations of it.
In the archive below is the full source code for the hook and sample programs as well as binaries for the DLL and sample EXE that can be hooked.
Download: Archive of source/binaries

A downloadable PDF of this post can be found here.