

0.1 Function hooking

Function hooking, also known as API hooking, method hooking or simply binary hooking, is the process of intercepting function calls and redirecting the execution somewhere else. In most cases the execution is redirected to a function defined by the process intercepting the target function call, but can also be used to redirect execution somewhere else. To understand function hooking, it is important to understand how functions work on the assembly level¹.

x64_86 Call instruction. In x64_84 assembly, the `call` instruction exists for calling functions[3]. Parameters to the function is passed in registers and on the stack according to the calling convention[2]. In x64_86 assembly the parameters are passed according to the rules shown in table 1

Parameter type	Fifth and higher	Fourth	Third	Second	Leftmost
floating-point	stack	XMM3	XMM2	XMM1	XMM0
integer	stack	R9	R8	RDX	RCX
Aggregates (8, 16, 32, or 64 bits) and <code>__m64</code>	stack	R9	R8	RDX	RCX
Other aggregates, as pointers	stack	R9	R8	RDX	RCX
<code>__m128</code> , as a pointer	stack	R9	R8	RDX	RCX

Table 1: x64_86 calling convention in Windows[2]

For everything else than floating-point types, the parameters are passed in the order: `RCX`, `RDX`, `R8`, `R9`, and hereafter the stack as shown on listing 1

```
1 func1(int a, int b, int c, int d, int e, int f);
2 // a in RCX, b in RDX, c in R8, d in R9, f then e pushed on stack
```

Listing 1: x64_86 calling convention demonstrated[2]

Function interception and hooking. To hook a function is to simply redirect the execution somewhere else. Figure 1 shows the logic of redirecting the execution to another function, and hereafter returning to the original function.

¹Note that this project only deals with 64-bit Intel x86 assembly called x64_84

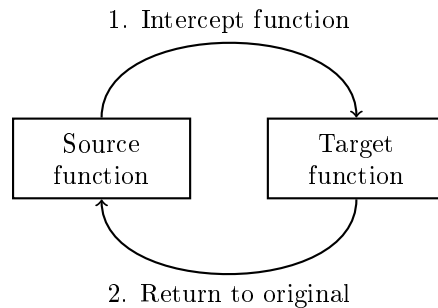


Figure 1: Function hooking

To redirect the execution of the source function to a target function, the instruction `JMP`[1] can be used. This instruction will return to the address located on the stack. The assembly code on listing 2 shows how this is done. The assembly should be written to memory at the beginning of the source function.

```
1 // Push our target function address to the stack
2 PUSH targetFunctionAddress
3 // Return to the location recently pushed on stack
  ↳ (targetFunctionAddress)
4 RET
```

Listing 2: x64_86 assembly code for redirecting execution

Returning to source function. When hooking a function, you often want to return back to the original function. Two considerations are needed when doing so. First off, the target function must restore the parameters originally passed to the source function in order to continue execution. Secondly, the hook itself must be temporarily removed by restoring the replaced bytes, such that calling the source function will not redirect to the target function again. Failing to do so will result in an infinite loop. Listing 3 shows a simplified prototype written in C++

0.1. FUNCTION HOOKING

```
1 FARPROC sourceFunctionAddress = NULL;
2 SIZE_T bytesWritten = 0;
3 char sourceFunctionOriginalBytes[6] = {};
4
5 int __stdcall TargetFunction(int parameter1, int parameter2) {
6
7     WriteProcessMemory(GetCurrentProcess(),
8         ↳ (LPVOID)sourceFunctionAddress, sourceFunctionOriginalBytes,
9         ↳ sizeof(sourceFunctionOriginalBytes), &bytesWritten);
10
11     // call the source function
12     return SourceFunction(parameter1, parameter2);
13 }
14
15 int HookSourceFunction()
16 {
17     HINSTANCE library = LoadLibraryA("sourceLibrary");
18     SIZE_T bytesRead = 0;
19
20     // get address of the source function in memory
21     sourceFunctionAddress = GetProcAddress(library,
22         ↳ "sourceFunction");
23
24     // save the first 6 bytes of the source function - it is needed
25     ↳ for unhooking
26     ReadProcessMemory(GetCurrentProcess(), sourceFunctionAddress,
27         ↳ sourceFunctionOriginalBytes, 6, &bytesRead);
28
29     // Patch the source function
30     void *targetFunctionAddress = &TargetFunction;
31     char patch[6] = { 0 };
32     memcpy_s(patch, 1, "\\x68", 1); // ASM: PUSH
33     memcpy_s(patch + 1, 4, &targetFunctionAddress, 4);
34     memcpy_s(patch + 5, 1, "\\xC3", 1); // ASM: RET
35
36     WriteProcessMemory(GetCurrentProcess(),
37         ↳ (LPVOID)sourceFunctionAddress, patch, sizeof(patch),
38         ↳ &bytesWritten);
39
40     return 0;
41 }
```

Listing 3: Simplified prototype to hook a function using C++

0.1.1 User mode function hooking

0.1.2 Kernel mode hooking