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 level[1].

x64_86 Call instruction. In x64_84 assembly, the `call` instruction exists for calling functions[6]. Parameters to the function is passed in registers and on the stack according to the calling convention[5]. 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 __m64	stack	R9	R8	RDX	RCX
Other aggregates, as pointers	stack	R9	R8	RDX	RCX
__m128, as a pointer	stack	R9	R8	RDX	RCX

Table 1: x64_86 calling convention in Windows[5]

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[5]

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.

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

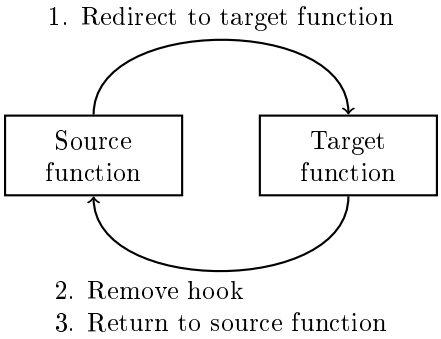

Figure 1: Function hooking

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 it self 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++

```
FARPROC sourceFunctionAddress = NULL;
SIZE_T bytesWritten = 0;
char sourceFunctionOriginalBytes[6] = {};

int __stdcall TargetFunction(int parameter1, int parameter2) {

        WriteProcessMemory(GetCurrentProcess(),
        ↪  (LPVOID)sourceFunctionAddress, sourceFunctionOriginalBytes,
        ↪  sizeof(sourceFunctionOriginalBytes), &bytesWritten);

        // call the source function
        return SourceFunction(parameter1, parameter2);
}

int HookSourceFunction()
{
        HINSTANCE library = LoadLibraryA("sourceLibrary");
        SIZE_T bytesRead = 0;

        // get address of the source function in memory
        sourceFunctionAddress = GetProcAddress(library,
        ↪  "sourceFunction");

        // save the first 6 bytes of the source function - it is needed
        ↪  for unhooking
        ReadProcessMemory(GetCurrentProcess(), sourceFunctionAddress,
        ↪  sourceFunctionOriginalBytes, 6, &bytesRead);

    // Patch the source function
        void *targetFunctionAddress = &TargetFunction;
        char patch[6] = { 0 };
        memcpy_s(patch, 1, "\x68", 1); // ASM: PUSH
        memcpy_s(patch + 1, 4, &targetFunctionAddress, 4);
        memcpy_s(patch + 5, 1, "\xC3", 1); // ASM: RET

        WriteProcessMemory(GetCurrentProcess(),
        ↪  (LPVOID)sourceFunctionAddress, patch, sizeof(patch),
        ↪  &bytesWritten);

        return 0;
}
```

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

In this section we described one method of hooking function in Windows. However, this specific implementation has certain limitations and flaws. One important flaw is the fact that the hook is removed upon trigger and never reinstated making it a one time hook. This could be improved upon, but it is

outside the scope of this section to do so. Hooking used in production system should use more well tested methods such as the trampoline method[2] which is how Microsoft Detours work[4].

0.1.1 Kernel mode function hooking

Up until this point we have only described how user mode function hooking works. To hook a driver such as `tcpip.sys` however, we need to do kernel mode function hooking. This poses an extra challenge as hooking can no longer be done from an user mode process. To hook kernel mode functions the application doing the hooking must also run from the kernel.

Windows security features. Windows has in recent years implemented several security features making it harder to exploit memory corruption vulnerabilities. Among these security features is the Driver Signing Policy which will prohibit any driver from being loaded if it is not signed with a valid EV code signing certificate[3]. If however, you are able to properly sign a driver, security features such as Data Execution Prevention (DEP), Hypervisor-protected code integrity (HVCI) and Control Flow Guard (CFG) make it increasingly harder to modify process memory and redirect execution.

With these security features in mind, it becomes increasingly more difficult to hook functions in kernel mode. Even more so, readily available function hooking libraries such as Detours[4] does not work in kernel mode, forcing one to create their own solution. Being in kernel mode also makes mistakes very costly, as a simple exception or error will not only crash the program but the whole computer resulting in a Blue screen of death (BSoD).