Inline Hooking on x64 - Daniel Ayzenshteyn

Summary

I implemented Inline Hooking on MessageBoxW and OutputDebugStringW.
For this I needed to use DLL injection, IPC via a named pipe, advanced trampoline and a relay function (specific for x64).

Why?

DLL injection for injecting the hook to the target process.
IPC for transferring parameters (filename|-o|-m) and controlling the injected DLL behavior at runtime.
Detour - Advanced Trampoline for returning to the original function and restoring its original flow (Avoid suspicion). Could be expanded to control the return values from the original function as well.
Relay function to implement it in x64. We want to use only 5 bytes for a relative jump in the original function. We put the relay function nearby, jumping to the relay function which is not further than 2GB in memory space from the original function. The relay function then jumps to the Detour function with an absolute jump which takes 12 bytes.

MessageBoxW

General flow of the hook:

Let's start with the hooked function in IDA (user32.dll):

```
                                ; int __stdcall MessageBoxW(HWND hWnd, LPCWSTR lpText, LPCWSTR lpCaption, UINT uType)
                                public MessageBoxW
                                MessageBoxW proc near

                                var_18= word ptr -18h
                                var_10= dword ptr -10h

48 83 EC 38                     sub     rsp, 38h
45 33 DB                        xor     r11d, r11d
44 39 1D 1A F2 03 00            cmp     cs:?gfEMIEnable@@3HA, r11d ; int gfEMIEnable
74 2E                           jz      short loc_18007B07E
```

Here we will patch the first two instructions (48 83 EC 38 + 45 33 DB) to a relative jump:
E9 <4 bytes of relative address>
The relative address will be of the relay function.

The **relay function** will consist of:
0x49, 0xBA, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, //mov r10, addr
0x41, 0xFF, 0xE2 //jmp r10
The addr will be the address of the detour function.

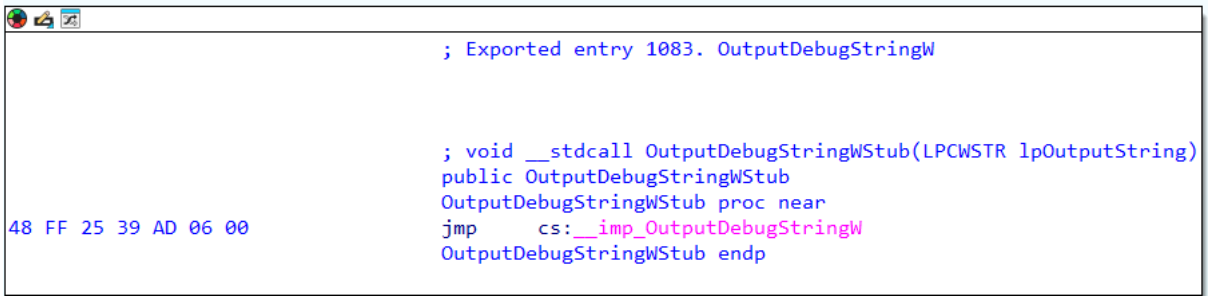
Then in the **detour function** we will write the function parameters into a file (location of which we passed via IPC from the Injector).

The detour function later calls the **trampoline** function which consists of the first two instructions of MessageBoxW and an absolute jump back to the (MessageBoxW + 7) address - The cmp op code and continue regular flow of the command.

At the end the retn op code in the MessageBoxW will return to our detour function and we will be able to alter the returned value before passing it to the caller. (Here we are only interested in writing the input of MessageBoxW to a file so no change of return value will be implemented.)

OutputDebugStringW

Let's look at the function in IDA (kernel32.dll):



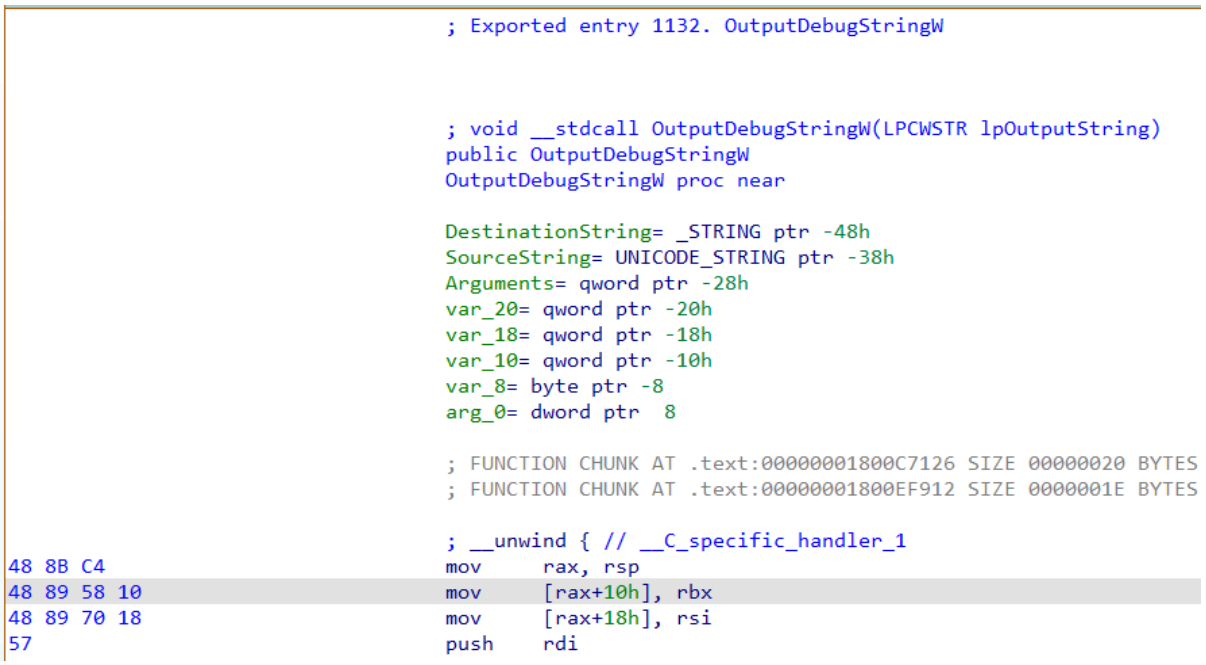
```
                                ; Exported entry 1083. OutputDebugStringW



                                ; void __stdcall OutputDebugStringWStub(LPCWSTR lpOutputString)
                                public OutputDebugStringWStub
                                OutputDebugStringWStub proc near
48 FF 25 39 AD 06 00            jmp     cs:__imp_OutputDebugStringW
                                OutputDebugStringWStub endp
```

I tried to patch it similarly to the MessageBoxW but it didn't work out. appears that because it's a stub and it only jumps to the real function we push the argument twice messing up the stack (probably).

So let's look at the real implementation at KernelBase.dll:



```
                                ; Exported entry 1132. OutputDebugStringW



                                ; void __stdcall OutputDebugStringW(LPCWSTR lpOutputString)
                                public OutputDebugStringW
                                OutputDebugStringW proc near

                                DestinationString= _STRING ptr -48h
                                SourceString= UNICODE_STRING ptr -38h
                                Arguments= qword ptr -28h
                                var_20= qword ptr -20h
                                var_18= qword ptr -18h
                                var_10= qword ptr -10h
                                var_8= byte ptr -8
                                arg_0= dword ptr  8

                                ; FUNCTION CHUNK AT .text:00000001800C7126 SIZE 00000020 BYTES
                                ; FUNCTION CHUNK AT .text:00000001800EF912 SIZE 0000001E BYTES

                                ; __unwind { // __C_specific_handler_1
48 8B C4                        mov     rax, rsp
48 89 58 10                     mov     [rax+10h], rbx
48 89 70 18                     mov     [rax+18h], rsi
57                              push    rdi
```

We see that we have the first two instructions that are (48 8B C4 + 48 89 58 10) that are enough for the relative jump.
We should pay attention to the fact that here we use RAX. (my trampoline jumps with rax so we need to switch the register… we will use r10).

The rest of the patching is similar to the MessageBoxW just in the trampoline I used r10 to jump to the absolute address of the KernelBase.dll original function.