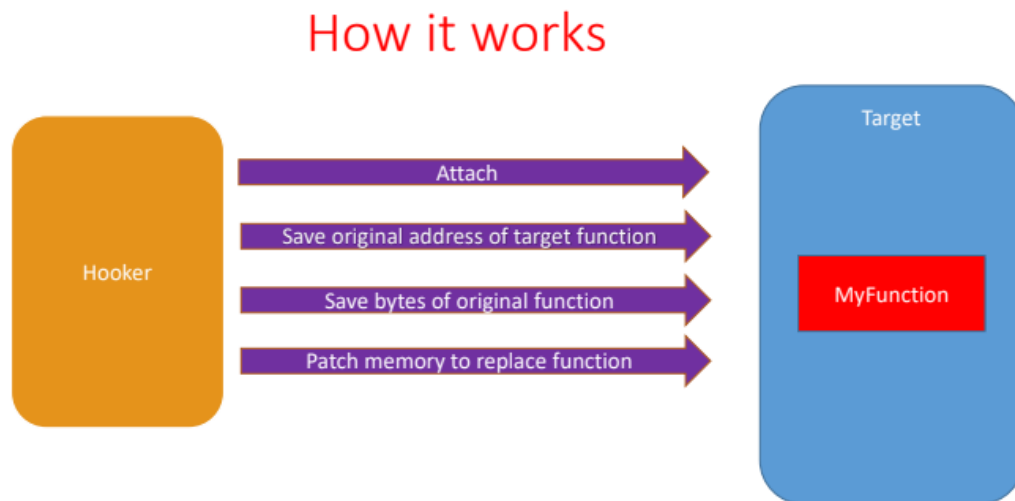


API Hooking using Inline Patch Hooking

In this section, we are going to learn about API Hooking using Inline Patch Hooking. Creating our hooks without using external libraries or frameworks.



Here on the left we have our hooker, and on the right, we have our target process, and inside the target process, it has got an OriginalFunction, which we are going to hook. So, the first step is to somehow attach the hooker to the process, then the hooker will save the original address of this target function(OriginalFunction), it does it to revert the OriginalFunction to its original state if it needs to. It even saves the bytes of the original function for the same reason. Then it will patch the memory to replace this OriginalFunction with the address of our modified MyFunction. So, once the target process executes, and calls the MyFunction function, it will execute our modified function, instead of the original function.

So, that's how inline hooking works.

Now, let's take a look at the code:

We can see here that it is same the code as earlier used.

```

1  #include <windows.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #pragma comment(lib, "user32.lib")
5
6  int main(void){
7
8      printf("Target For Hooker is Starting...\n");
9
10     //-- ref: https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messagebox
11     MessageBox(NULL, "This is the first message from THE FUTURE!", "1st MessageBox", MB_OK | MB_ICONINFORMATION);
12     MessageBox(NULL, "This is the second message from THE FUTURE!", "2nd MessageBox", MB_OK | MB_ICONINFORMATION);
13     MessageBox(NULL, "This is the third message from THE FUTURE!", "3rd MessageBox", MB_OK | MB_ICONINFORMATION);
14
15     printf("Target For Hooker exiting...\n");
16
17     return 0;
18 }

```

Let's take a look at the patchhooker .cpp:

```

1  #include <stdio.h>
2  #include <windows.h>
3  #include <dbghelp.h>
4
5  #pragma comment(lib, "user32.lib")
6  #pragma comment(lib, "dbghelp.lib")
7
8  #define SIZE_OF_ORIGINAL_INSTRUCTION 14
9
10 BOOL HookAndPatch(FARPROC hookingFunc);
11
12 //-- pointer to original MessageBox
13 typedef int (WINAPI * OrigMessageBox_t)(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
14 OrigMessageBox_t pOrigMessageBox = NULL;
15
16 //-- storage for original bytes from MessageBox
17 char OriginalBytes[SIZE_OF_ORIGINAL_INSTRUCTION] = { 0 };
18
19
20 //-- the modified MessageBox function
21 int ModifiedMessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType) {
22     SIZE_T bytesOut = 0;
23
24     printf("ModifiedMessageBox() called. No MessageBox popup on screen!\n");
25
26     //-- restore the original function
27     //WriteProcessMemory(GetCurrentProcess(), (LPVOID)pOrigMessageBox, OriginalBytes, SIZE_OF_ORIGINAL_INSTRUCTION, &bytesOut);
28     //pOrigMessageBox(hWnd, lpText, lpCaption, uType);
29     //HookAndPatch((FARPROC) ModifiedMessageBox);
30
31     return IDOK;
32 }

```

```

35  //-- Set a hook on the MessageBox function by patching code
36  BOOL HookAndPatch(FARPROC hookingFunc) {
37
38      SIZE_T bytesIn = 0;
39      SIZE_T bytesOut = 0;
40
41      //-- save the original address of MessageBoxA
42      pOrigMessageBox = (OrigMessageBox_t) GetProcAddress(GetModuleHandle("user32.dll"), "MessageBoxA");
43
44      //-- copy SIZE_OF_ORIGINAL_INSTRUCTION bytes of original code from MessageBoxA
45      ReadProcessMemory(GetCurrentProcess(), pOrigMessageBox, OriginalBytes, SIZE_OF_ORIGINAL_INSTRUCTION, &bytesIn);
46
47
48      //-- 6 hex bytes for JUMP instruction: \xFF\x25\x00\x00\x00\x00
49      //-- 8 hex bytes for the function address
50      char patch[14] = { 0 };
51      memcpy(patch, "\xFF\x25", 2);
52      memcpy(patch + 6, &hookingFunc, 8);
53
54      // patch the MessageBoxA
55      WriteProcessMemory(GetCurrentProcess(), (LPVOID) pOrigMessageBox, patch, sizeof(patch), &bytesOut);
56
57      printf("MessageBoxA() has been hooked!\n");
58      printf("ModifiedMessageBox is at: %p ; OriginalBytes is at: %p\n", ModifiedMessageBox, OriginalBytes);
59
60      return FALSE;
61  }

```

```

64  BOOL WINAPI DllMain(HINSTANCE hinst, DWORD dwReason, LPVOID reserved) {
65
66      switch (dwReason) {
67          case DLL_PROCESS_ATTACH:
68              HookAndPatch((FARPROC) ModifiedMessageBox);
69              break;
70
71          case DLL_THREAD_ATTACH:
72              break;
73
74          case DLL_THREAD_DETACH:
75              break;
76
77          case DLL_PROCESS_DETACH:
78              break;
79      }
80
81      return TRUE;
82  }

```

In the DllMain function, we can see the case of DLL_PROCESS_ATTACH, here we are calling the HookAndPatch function, it takes only the parameter in, that is the ModifiedMessageBox. Here also we are trying to intercept the API functions and trying to insert our own modified functions. So, here we need to pass the address of the ModifiedMessageBox function.

ModifiedMessageBox function is the same function that he had already used, except for a few lines of code.

So, once we call the HookAndPatch function, then the first thing it will do is save the address of the original function, and then it will copy the original instruction bytes to a safe location in the memory. It does it so that in any case if it wants to revert to the original function, then it will do it so that it can continue back to the normal process, which the target process was supposed to do. To do so we use a new API function ReadProcessMemory.

```
C++

BOOL ReadProcessMemory(
    [in] HANDLE hProcess,
    [in] LPCVOID lpBaseAddress,
    [out] LPVOID lpBuffer,
    [in] SIZE_T nSize,
    [out] SIZE_T *lpNumberOfBytesRead
);
```

And the 1st parameter is GetCurrentProcess, which gets the current Process because it is trying to modify the address of the currently running process.

The 2nd parameter is the address of the original message box API is found

The 3rd parameter is the buffer, where we are going to stall the instructions too.

Then the 4th parameter is the size of the original instruction.

Now the next step is to prepare the bytes that we are going to patch in this running process in the memory. So, we prepared a string of 14 characters bytes long, because the instruction which we are going to use is 14 bytes long. So, we memcopy function to copy the first two bytes "\xFF\x25", which means jump instruction

6 hex bytes for JUMP instruction: \xFF\x25\x00\x00\x00\x00

Then for the 8 remaining bytes, we are going to copy the address of our ModifiedMessageBox function. We are appending the address of our ModifiedMessageBox to the jump instruction.

This hooking function is what is being passed to the HookAndPatch function, and that refers to the ModifiedMessageBox function. So, the ModifiedMessageBox is going to replace the original function.

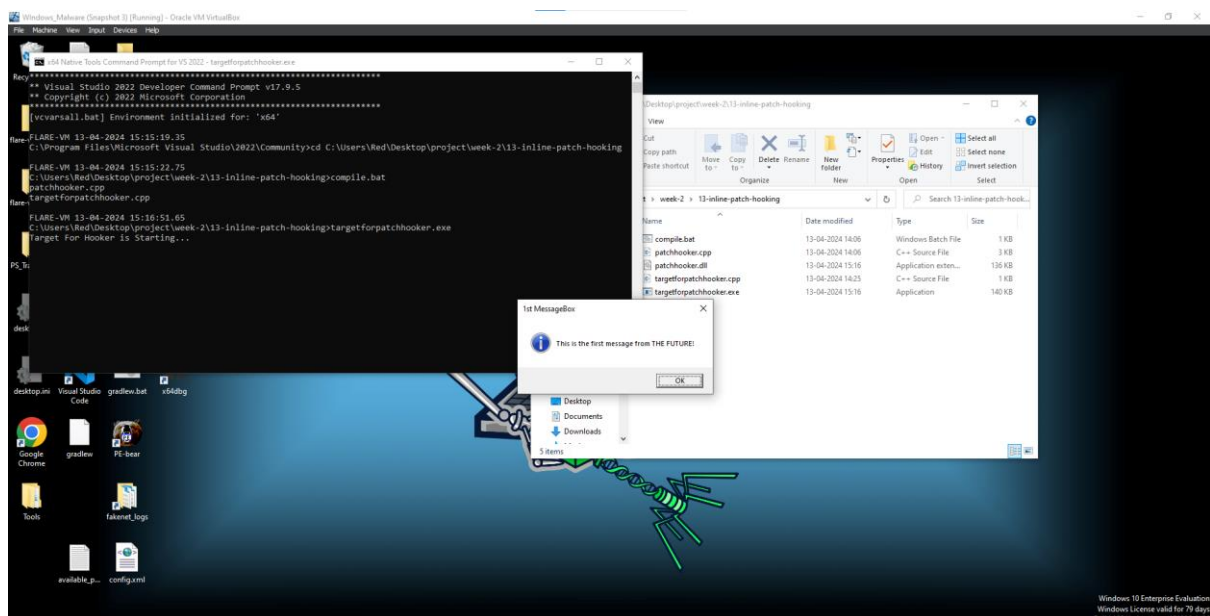
Now, we need to replace the bytes of the original instruction, and it is done by calling the WriteProcessMemory API function.

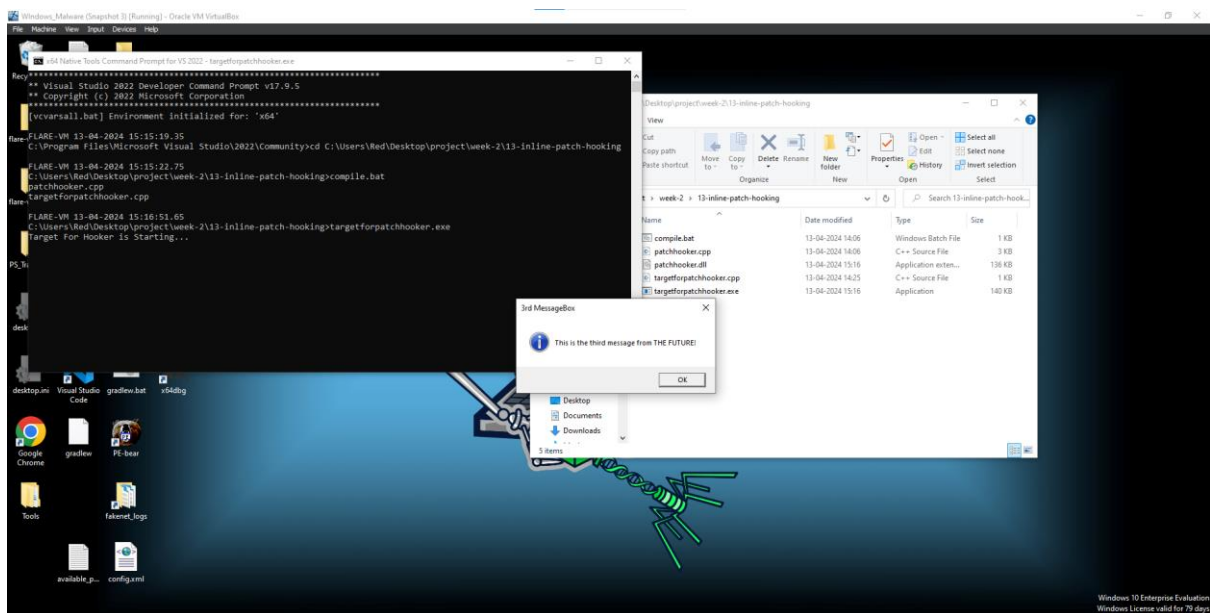
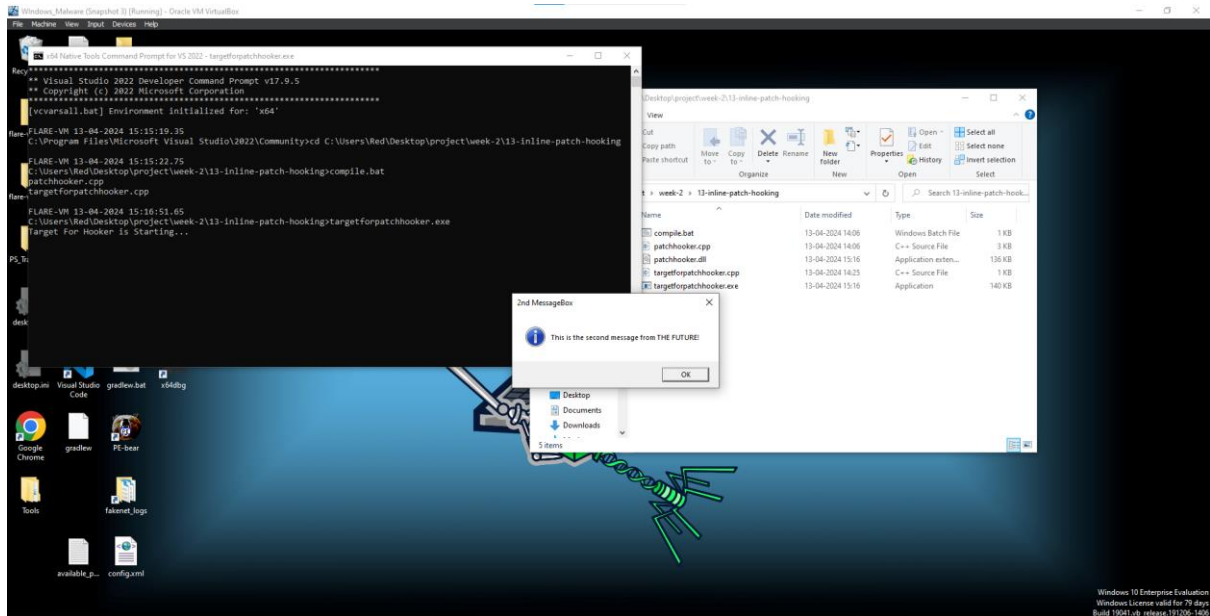
Now once done with the patching we are done with all the steps, now whenever the target process calls the function, it will call my function, because it has been replaced from the original function. So, it is going to run the new replaced function now.

We can even revert the changes by using this code:

```
26 //-- restore the original function
27 //WriteProcessMemory(GetCurrentProcess(), (LPVOID)pOrigMessageBox, OriginalBytes, SIZE_OF_ORIGINAL_INSTRUCTION, &bytesOut);
28 //pOrigMessageBox(hWnd, lpText, lpCaption, uType);
29 //HookAndPatch((FARPROC) ModifiedMessageBox);
```

Now let's compile the files and then execute them:





So, here we can see all of the message boxes, so our target function is working properly, so now let's inject the dll into the process.

```
C:\Users\Red\Desktop\project\week-2\13-inline-patch-hooking>targetforpatchhooker.exe
Target For Hooker is Starting...
MessageBoxA() has been hooked!
ModifiedMessageBox is at: 00007FFF0F411130 ; OriginalBytes is at: 00007FFF0F431AE8
ModifiedMessageBox() called. No MessageBox popup on screen!
ModifiedMessageBox() called. No MessageBox popup on screen!
Target For Hooker exiting...
```

Here we can see that as soon as we inject the dll into the process, it gives us some addresses, as we continue the process, by clicking the ok button, the process exits and then it prints out some statements, because in the code, instead of running a new message box, we are just

So, now let's reverse engineer it:








So, run the .exe file attach the process in xdbg, then in the symbols section go to the user32.dll file, then search for the MessageBoxA, then follow it in the disassembler.

Now, here we can see the instruction is a sub, but now inject the .dll file with the help of a process hacker.

Now, here press on the analysis option, because it will refresh the disassembler, we can see the change here:

We can see that the instruction has been changed to jump because in the patch we had used `"/XFF/X25"` that's the instruction for the jump, and then we copied the address to the `ModifiedMessageBox` function.

We can verify that after the jump instruction, the address is the same as of the `ModifiedMessageBox`. We can even follow it in the dump section:

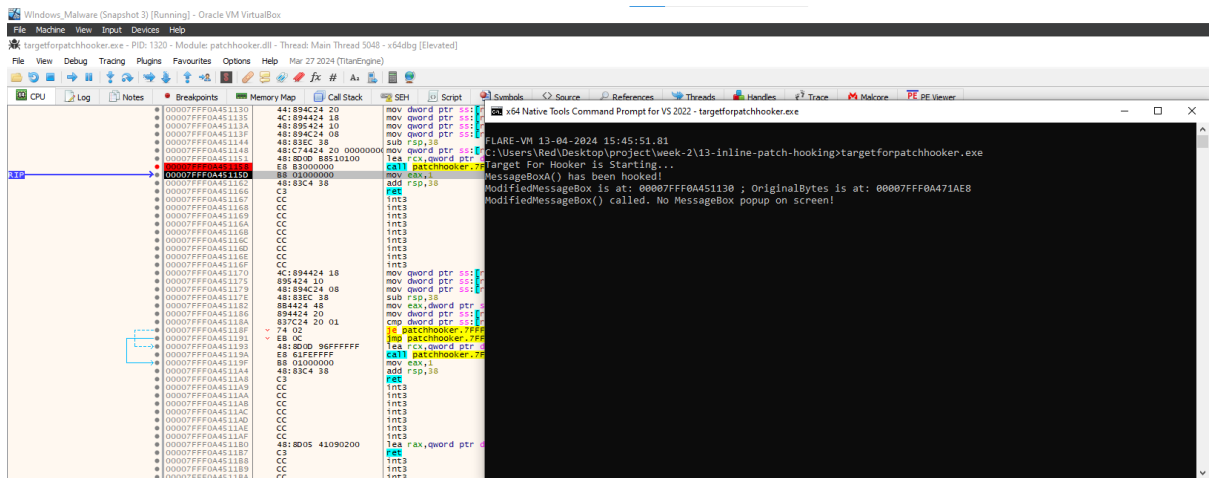
 Dump 1	 Dump 2	 Dump 3	 Dump 4	 Dump 5	 Watch 1	[x=] Locals	 Struct														
Address	Hex																				ASCII
00007FFF31F9AC10	FF	25	00	00	00	00	30	11	45	0A	FF	7F	00	00	74	2E	Y%...O.E.ÿ...t.				
00007FFF31F9AC20	65	48	88	04	25	30	00	00	00	4C	8B	50	48	33	C0	F0	eH.%O...L.PH3A0				
00007FFF31F9AC30	4C	0F	B1	15	C8	91	03	00	4C	88	15	C9	91	03	00	41	L.±.E...L.Ë...A				
00007FFF31F9AC40	8D	43	01	4C	0F	44	D0	4C	89	15	BA	91	03	00	83	4C	.C.L.D0L..°...L				
00007FFF31F9AC50	24	28	FF	66	44	89	5C	24	20	E8	E2	02	00	00	48	83	\$(ÿFD.\\$ eä...H.				
00007FFF31F9AC60	C4	38	C3	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	ASAIiiiiiiiiiiii				
00007FFF31F9AC70	48	83	EC	38	0F	B7	44	24	60	83	4C	24	28	FF	66	89	H.i8..D\$.L\$(ÿf.				
00007FFF31F9AC80	44	24	20	E8	B8	02	00	00	48	83	C4	38	C3	CC	CC	CC	D\$ e...H.ASAIii				
00007FFF31F9AC90	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	iiiiiiiiiiiiiiii				
00007FFF31F9ACA0	48	83	EC	38	0F	B7	44	24	60	83	4C	24	28	FF	66	89	H.i8..D\$.L\$(ÿf.				
00007FFF31F9ACB0	44	24	20	E8	F8	03	00	00	48	83	C4	38	C3	CC	CC	CC	D\$ eo...H.ASAIii				
00007FFF31F9ACC0	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	iiiiiiiiiiiiiiii				
00007FFF31F9ACD0	48	89	5C	24	10	48	89	74	24	18	55	57	41	56	48	8D	H.\\$.H.t\$.UWAVH.				
00007FFF31F9ACE0	6C	24	B9	48	81	EC	F0	00	00	00	48	8B	05	0F	7C	03	1\$'H.i0...H... .				
00007FFF31F9ACF0	00	48	33	C4	48	89	45	37	33	FF	48	8B	D9	33	F6	48	.H3AH.E73YH.Ü30H				
00007FFF31F9AD00	89	7D	87	33	D2	48	89	75	8F	48	8D	4D	E7	44	8D	47	.}.30H.u.H.McD.G				
00007FFF31F9AD10	50	E8	95	C0	FB	FF	0F	10	03	44	8D	77	01	0F	10	53	Pe.Aüy...D.w...S				
00007FFF31F9AD20	10	0F	10	4B	30	0F	29	45	97	0F	10	43	20	0F	29	55	...K0.)E...C.)U				

Here we can see the address, the first 6 bytes are the instructions for jump, and then the next 8 bytes are the addresses of the ModifiedMessageBox function.

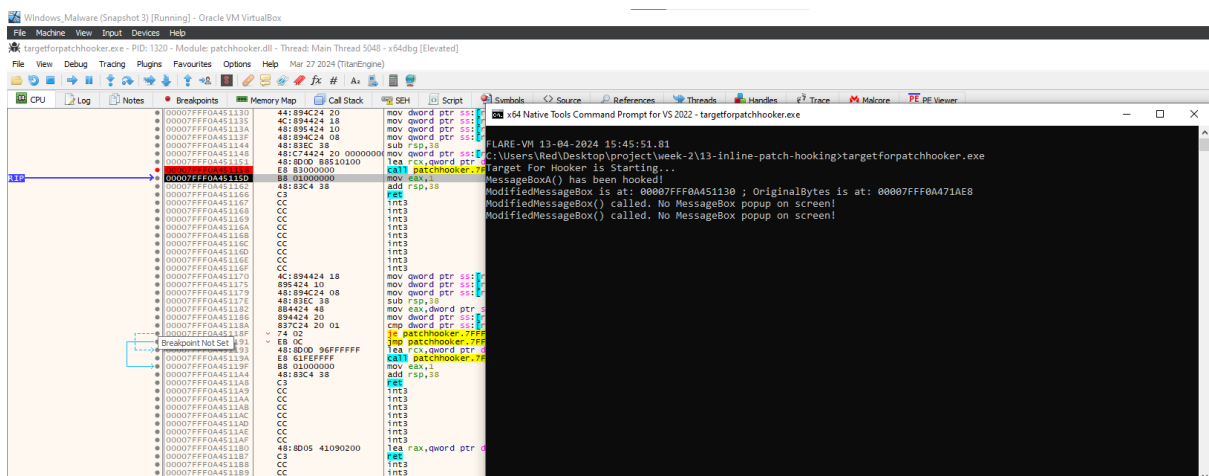
Now set the breakpoint at the MessageBoxA, then run the program, by clicking the ok button, it will hit the breakpoint, the step over, you will go to the MessageBoxA function, then we will another breakpoint at patchhooker because here it will print the message according to the code.

CPU	Log	Notes	Breakpoints	Memory Map	Call Stack	SDH	Script	Symbols	Source	References	Threads	Handles	Trace	Malcore	PE Viewer
00007FFF0A451130			4C:894424 18	00007FFF0A451130			mov qword ptr [?sp-18],r8	[rsp+18]:&ALLUSERSPROFILEC:\ProgramData							
00007FFF0A451131			895424 20	00007FFF0A451131			mov qword ptr [?sp-10],edx	[rsp+10]:&TargetForHooker.exe							
00007FFF0A451132			4E:894C24 08	00007FFF0A451132			mov qword ptr [?sp],rcx	[rsp+08]:&TargetFor Hooker is Starting...\n							
00007FFF0A451133			48:83EC 38	00007FFF0A451133			sub sp,38								
00007FFF0A451134			4EC74424 20 00000000	00007FFF0A451134			mov qword ptr [?sp+20],0								
00007FFF0A451135			48:BD00 B8510200	00007FFF0A451135			lea rcx,qword ptr [?sp+0A466310]	00007FFF0A466310:ModifiedMessageBox() called. No MessageBox popup on screen!\n							
00007FFF0A451136			E8 B300000000	00007FFF0A451136			call patchhooker.7FFF0A451120								
00007FFF0A45113D			B8 01000000	00007FFF0A45113D			mov eax,1								
00007FFF0A451162			48:83C4 38	00007FFF0A451162			add esp,38								
00007FFF0A451166			CC	00007FFF0A451166			ret								
00007FFF0A451167			CC	00007FFF0A451167			int3								
00007FFF0A451168			CC	00007FFF0A451168			int3								
00007FFF0A451169			CC	00007FFF0A451169			int3								
00007FFF0A45116A			CC	00007FFF0A45116A			int3								
00007FFF0A45116B			CC	00007FFF0A45116B			int3								
00007FFF0A45116C			CC	00007FFF0A45116C			int3								
00007FFF0A45116D			CC	00007FFF0A45116D			int3								
00007FFF0A45116E			CC	00007FFF0A45116E			int3								
00007FFF0A45116F			CC	00007FFF0A45116F			int3								
00007FFF0A451170			4C:894424 18	00007FFF0A451170			mov qword ptr [?sp-18],r8	[rsp+18]:&ALLUSERSPROFILEC:\ProgramData							
00007FFF0A451175			895424 20	00007FFF0A451175			mov qword ptr [?sp-10],edx	[rsp+10]:&TargetForHooker.exe							
00007FFF0A451179			4E:894C24 08	00007FFF0A451179			mov qword ptr [?sp],rcx	[rsp+08]:&TargetFor Hooker is Starting...\n							
00007FFF0A45117E			48:83EC 38	00007FFF0A45117E			sub sp,38								
00007FFF0A451182			884424 48	00007FFF0A451182			mov eax,qword ptr [?sp+4]								
00007FFF0A451186			894424 20	00007FFF0A451186			mov qword ptr [?sp+20],eax								
00007FFF0A45118A			83C4 20 01	00007FFF0A45118A			cmp dword ptr [?sp+20],01								
00007FFF0A45118F			74 02	00007FFF0A45118F			jz patchhooker.7FFF0A451199								
00007FFF0A451191			8B 0C	00007FFF0A451191			jmp patchhooker.7FFF0A45119B								
00007FFF0A451193			48:BD00 96FFFFF	00007FFF0A451193			lea rcx,qword ptr [?sp+0A451130]	00007FFF0A451130							
00007FFF0A451194			EB 51FFFFF	00007FFF0A451194			call patchhooker.7FFF0A451100								
00007FFF0A45119F			B8 01000000	00007FFF0A45119F			mov eax,1								
00007FFF0A4511A4			48:83C4 38	00007FFF0A4511A4			add esp,38								
00007FFF0A4511A8			CC	00007FFF0A4511A8			ret								
00007FFF0A4511A9			CC	00007FFF0A4511A9			int3								
00007FFF0A4511AA			CC	00007FFF0A4511AA			int3								
00007FFF0A4511AB			CC	00007FFF0A4511AB			int3								
00007FFF0A4511AC			CC	00007FFF0A4511AC			int3								
00007FFF0A4511AD			CC	00007FFF0A4511AD			int3								
00007FFF0A4511AE			CC	00007FFF0A4511AE			int3								
00007FFF0A4511AF			CC	00007FFF0A4511AF			int3								
00007FFF0A4511B0			48:BD05 41090200	00007FFF0A4511B0			lea rcx,qword ptr [?sp+0A471AF8]								
00007FFF0A4511B7			CC	00007FFF0A4511B7			ret								
00007FFF0A4511B8			CC	00007FFF0A4511B8			int3								
00007FFF0A4511B9			CC	00007FFF0A4511B9			int3								

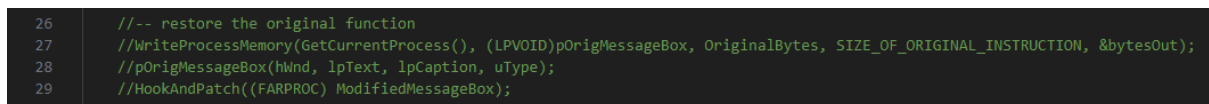
Now after putting the breakpoint, step down to the breakpoint, and then here we can see that it prints the message, as per the code.



Now run the code, then once again it will hit the breakpoint at MessageBoxA, then once again step down, then run the program, you will hit another breakpoint, then once again step down, then we can see that we get another print message in the command prompt.



If we want to print all the message boxes, even though we are injecting the dll file, we can just uncomment these lines:



It will show all of the message boxes, and if we want to rehook the program, we can just uncomment the HookAndPatch line, and it will rehook the program.