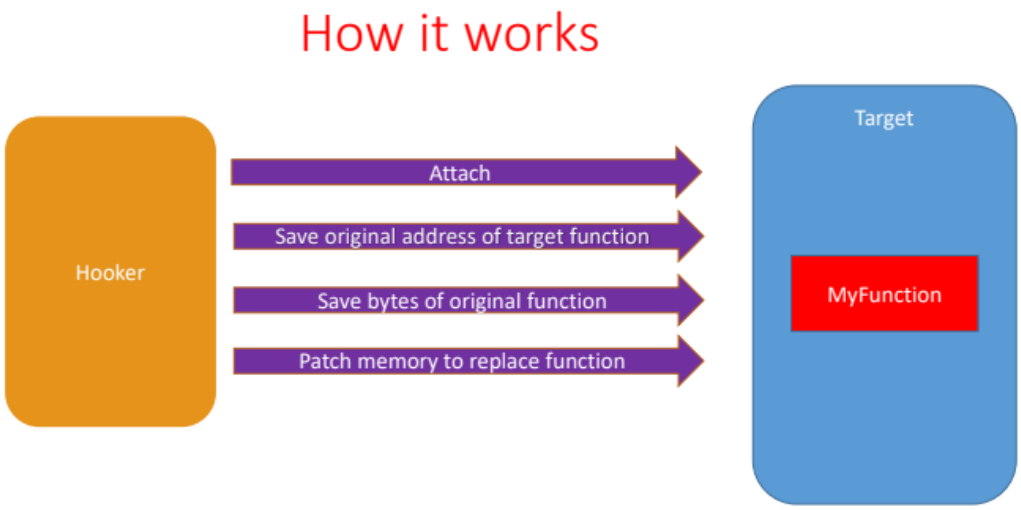
API Hooking using Inline Patch Hooking

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.

```c
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#pragma comment(lib, "user32.lib")

int main(void){

    printf("Target For Hooker is Starting...\n");

    //-- ref:  https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messagebox
    MessageBox(NULL, "This is the first message from THE FUTURE!", "1st MessageBox", MB_OK | MB_ICONINFORMATION);
    MessageBox(NULL, "This is the second message from THE FUTURE!", "2nd MessageBox", MB_OK | MB_ICONINFORMATION);
    MessageBox(NULL, "This is the third message from THE FUTURE!", "3rd MessageBox", MB_OK | MB_ICONINFORMATION);

    printf("Target For Hooker exiting...\n");

    return 0;
}
```

Let’s take a look at the patchooker .cpp:

```c
#include <stdio.h>
#include <windows.h>
#include <dbghelp.h>

#pragma comment(lib, "user32.lib")
#pragma comment (lib, "dbghelp.lib")

#define SIZE_OF_ORIGINAL_INSTRUCTION 14

BOOL HookAndPatch(FARPROC hookingFunc);

//-- pointer to original MessageBox
typedef int (WINAPI * OrigMessageBox_t)(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
OrigMessageBox_t pOrigMessageBox = NULL;

//-- storage for original bytes from MessageBox
char OriginalBytes[SIZE_OF_ORIGINAL_INSTRUCTION] = { 0 };


//-- the modified MessageBox function
int ModifiedMessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType) {
    SIZE_T bytesOut = 0;

    printf("ModifiedMessageBox() called. No MessageBox popup on screen!\n");

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

    return IDOK;
}
```

```
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 memcpy 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.

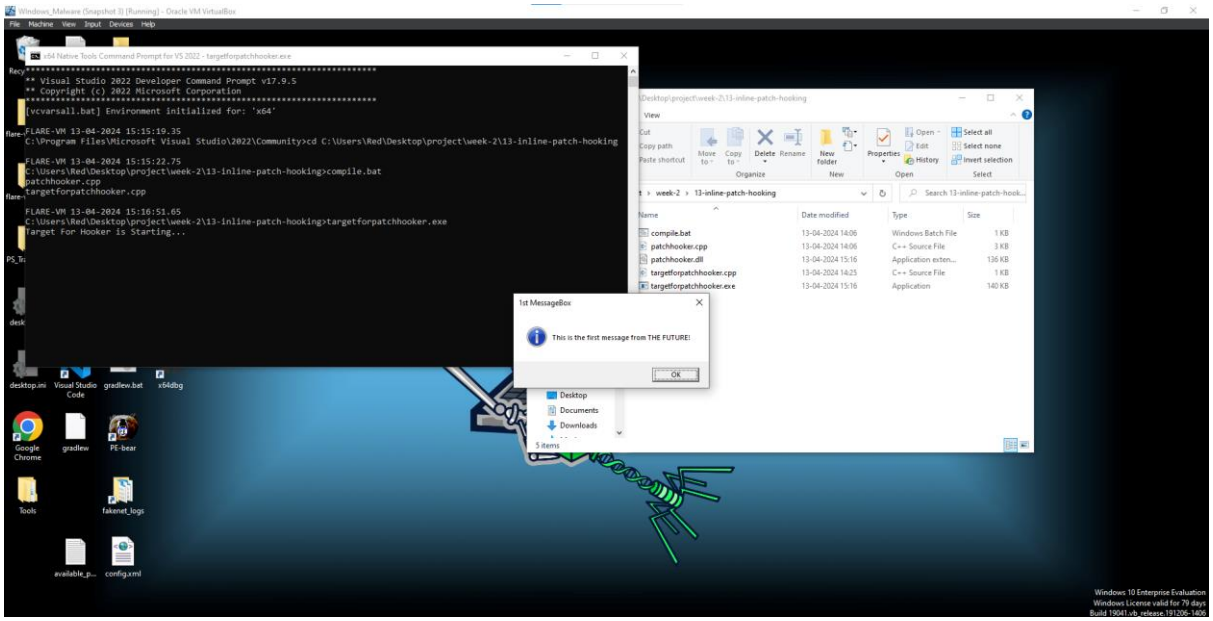
Now, we need to replace the bytes of the original instruction, and it is done by calling the WriteProcessMemory API function.

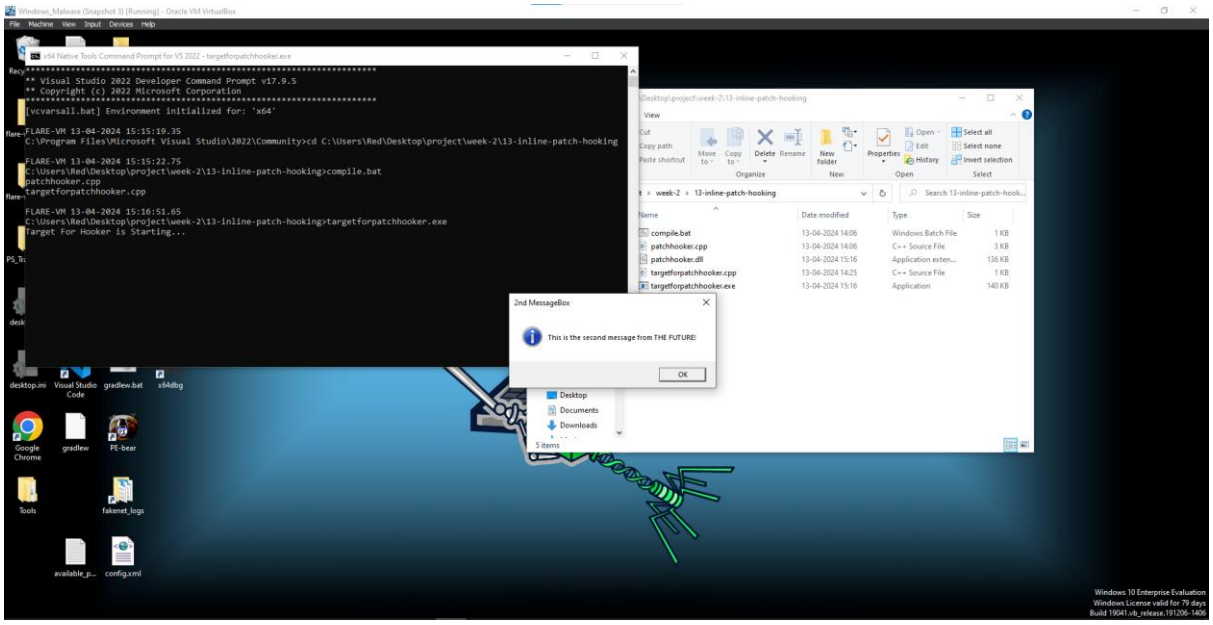
Now once done with the patching we are done with all the steps, now whenever the target process calls the function, it will call my function, because it has been replaced from the original function. So, it is going to run the new replaced function now.

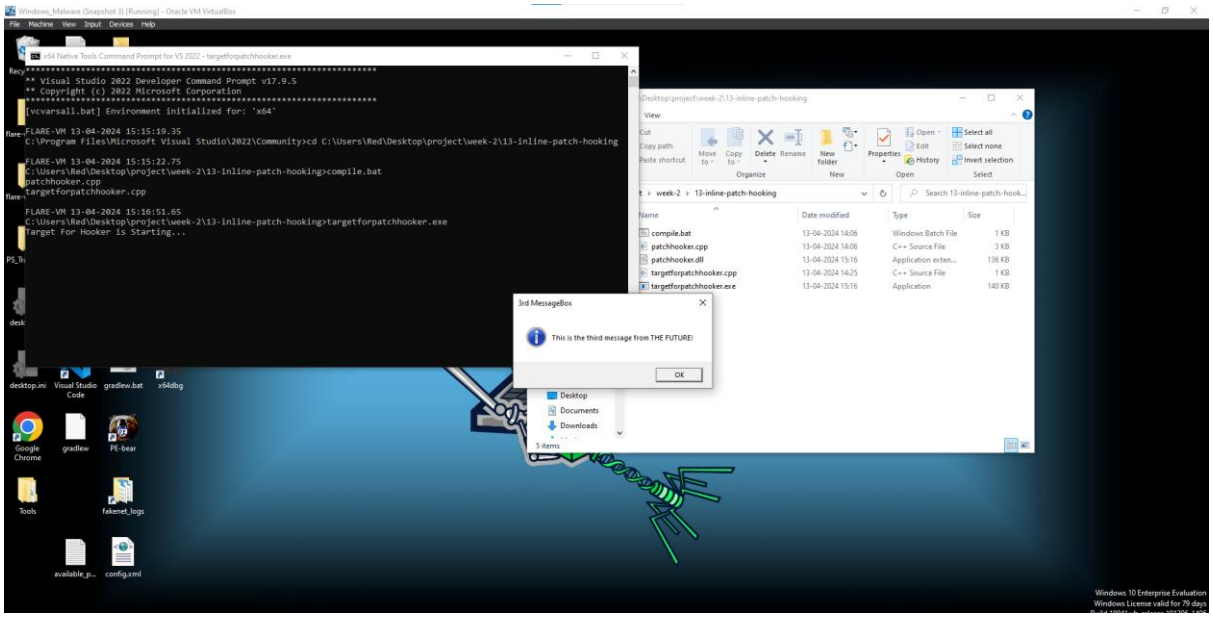
We can even revert the changes by using this code:

```
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...
```

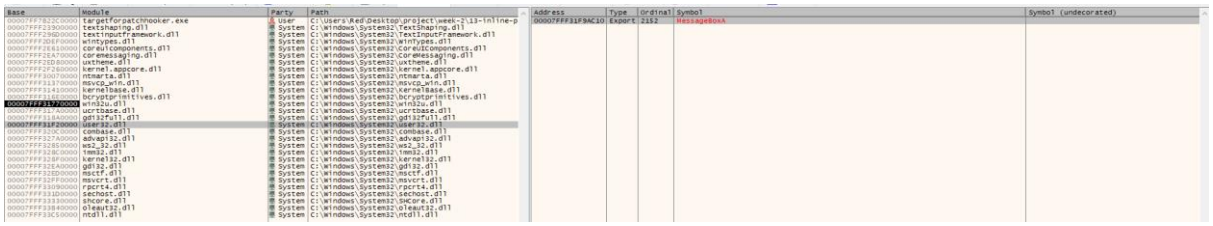
Here we can see that as soon as we inject the dll into the process, it gives us some addresses, as we continue the process, by clicking the ok button, the process exits and then it prints out some statements, because in the code, instead of running a new message box, we are just

printing the line, and here we can see that we didn't get any other popups because our ModifiedMessageBox replaces the original function.

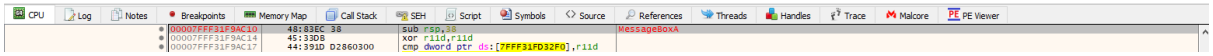
So, now let's reverse engineer it:

We will follow the steps as we had followed in the previous section:

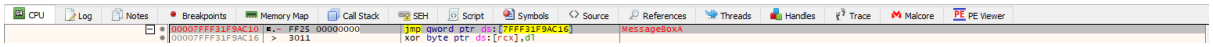
So, run the .exe file attach the process in xdbg, then in the symbols section go to the user32.dll file, then search for the MessageBoxA, then follow it in the disassembler.



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

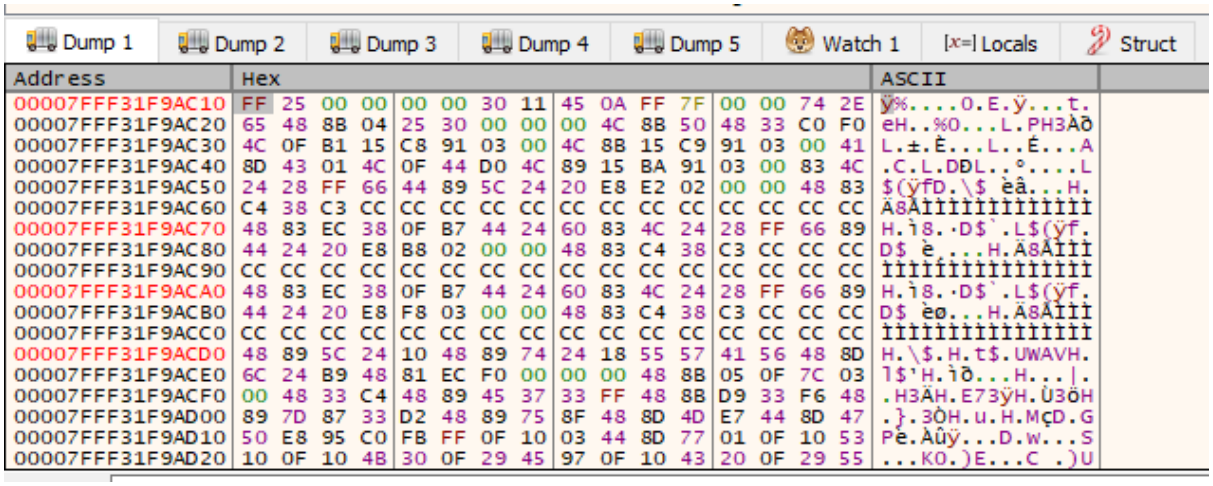


Now, here press on the analysis option, because it will refresh the disassembler, we can see the change here:



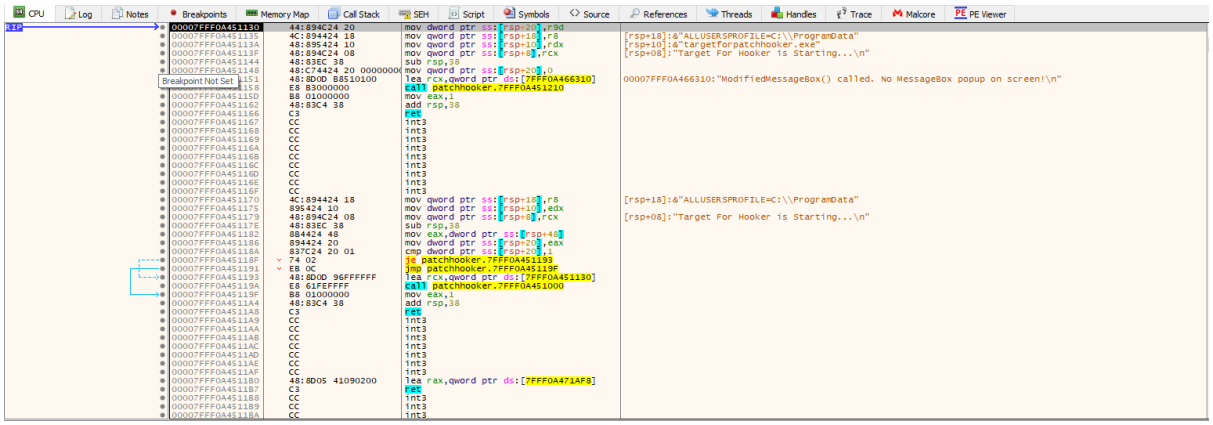
We can see that the instruction has been changed to jump because in the patch we had used "/XFF/X25" that's the instruction for the jump, and then we copied the address to the ModifiedMessageBox function.

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:

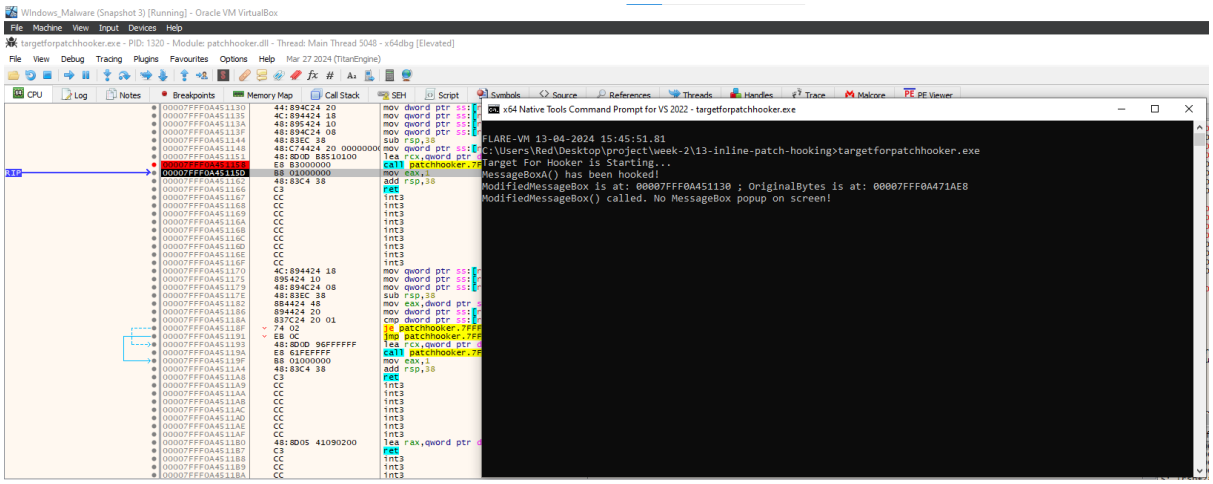


Here we can see the address, the first 6 bytes are the instructions for jump, and then the next 8 bytes are the addresses of the ModifiedMessageBox function.

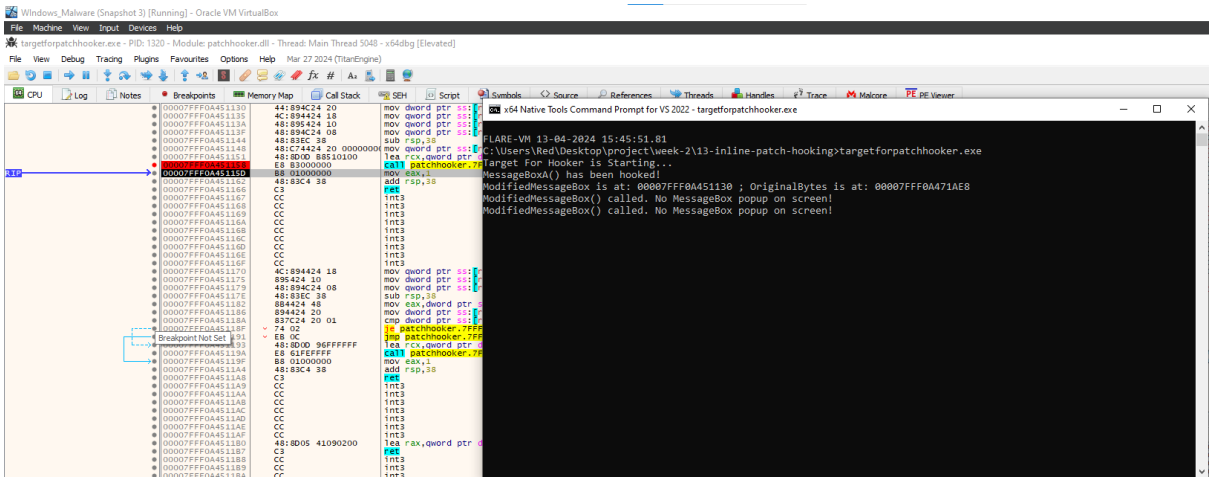
Now set the breakpoint at the MessageBoxA, then run the program, by clicking the ok button, it will hit the breakpoint, the step over, you will go to the MessageBoxA function, then we will another breakpoint at patchooker because here it will print the message according to the code.



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:

```
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);
```

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.