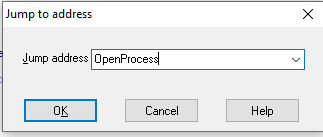
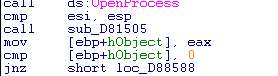
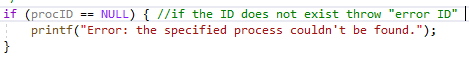
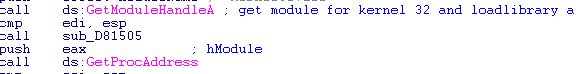
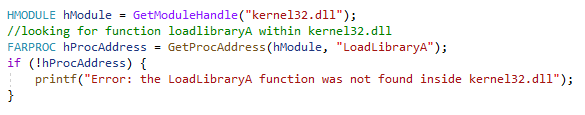
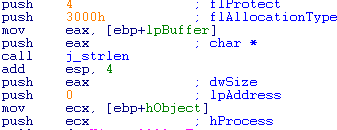
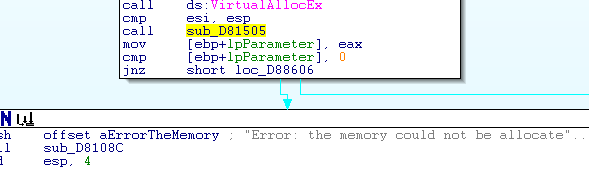
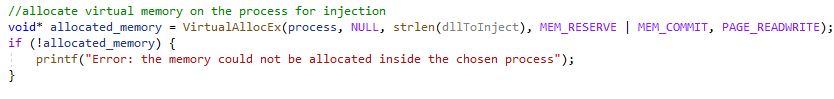
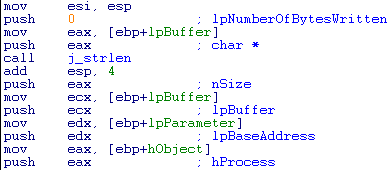
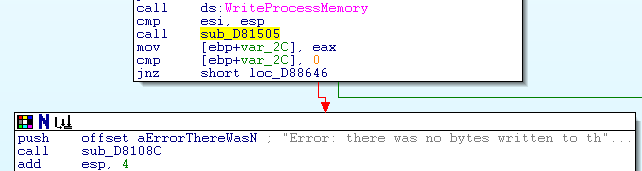
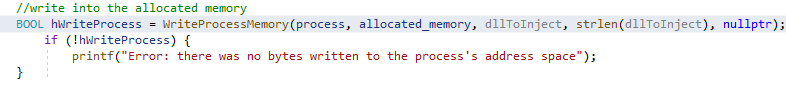
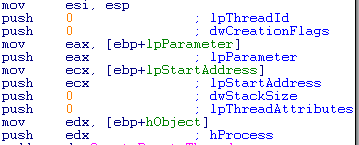
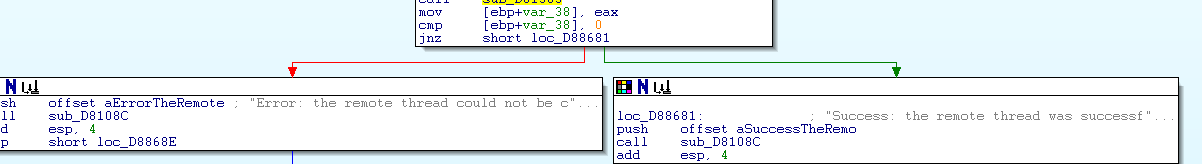
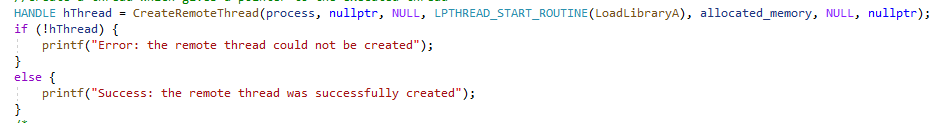
Your task is to finish the partially implemented Injector template by reverse engineering the Injector\_answer.exe you unzipped in step PR III part 1a.

1. After unzipping, opening the source code on Visual Studio, and IDA w/ injector solution
   1. Verify the existence of the process given
      1. To find the needed section of code we can look for a function call, OpenProcess. Press G, and type the function name
      2. Press Ok and click on the subroutine found, which is sub\_D88530
      3. Then we can see a comparison between 0 and what is located within ebp+hObject. hObject being the process name, this comparison is done after the OpenProcesss call so we must be verifying that the returned item exists
      4. If the item does not exist, if it is NULL then we print the following.
         1. This here allows us to complete this portion of the code
   2. Attain Process address for LoadLibraryA within kernel32.dll
      1. We first see two variables that are pushed, ProcName (LibraryLoadA) and ModuleName (kernel32.dll)
      2. The first item kernel32.dll is used in the call GetModuleHandle(), kernel32 is a module. LoadLibraryA is a process/function that is searched for within kernel32 using GetProcAddress()
      3. After attaining the result of ProcAddress() we compare it to 0 or NULL to verify that LibraryLoadA exists within kernel32.dll and that we were able to attain its address
      4. If the comparison is equal to Null then we go to error portion which prints a message to console, otherwise we continue with execution
         1. This section leads to the code:
   3. Allocate virtual memory for process injection
      1. Early in the code we find 5 pushes, these serve as parameters for a future function call. Thanks to IDA each push has a comment that lets us know what parameter identification we are passing
      2. VirtualAllocEx is the function that is used for these parameters, it is found right after the pushes
      3. Going from Bottom Up the VirtualAllocEx is used as: VirtualAllocEx(hProcess, NULL, strlen(dllToInject), MEM\_RESERVE | MEM\_COMMIT, PAGE\_READWRITE). Looking up in VirtualAllocEx, allows uis to translate the hex values into the parameters wanted such as 3000h being MEM\_RESERVE | MEM\_COMMIT
      4. After the call we compare the value to 0 or NULL to verify its existence. If it is false or 0 we print an error, otherwise we continue execution
         1. This section then leads to the code:
   4. Write into the allocated memory
      1. Much like with virtualalloc, we find multiple pushes, this lets us know we will be making a function call soon
      2. Now moving along, we find the call for WriteProccessMemory
      3. Like VirtualAllocEx, we see that the parameters are labeled with comments. Therefore, we can build the call WriteProcessMemory(hProcess, lpBaseAddress, lpBuffer, nsize, lpNumberOfBytesWritten)
      4. By then searching the WriteProcessFunction we can build the high-level code:
      5. Following the call in IDA, we have cmp to 0 or NULL to verify its existence. If it does not exist or is equal to 0 then we print an error to the console, otherwise we continue the execution
         1. This section then leads to
   5. Create the thread on the process
      1. After the success of WriteProcessMemory() we again see multiple pushes within the next subroutine
      2. IDA nicely comments the parameters we are passing and what they are for within the function call, moving further down we find our call is CreateRemoteThread()
      3. With this information we can formulate a call, which would like as follow: CreateRemoteThread(hProccess, 0, 0, lpStartAddress, lpParameter, 0, 0). Now we can look up CreateRemoteThread() to find its parameter types and how to correctly enter them
      4. Soon after the call, we see a compare with its return, cmp 0 or NULL. This is to verify the existence of the thread.
      5. If the result is false or zero we then print an error to the console, otherwise we print a success to console
         1. This section then gives us code: