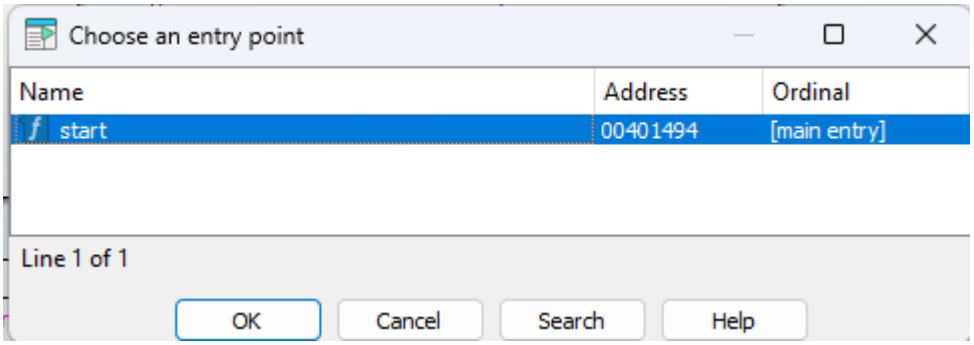
Question 2

1. Sections:

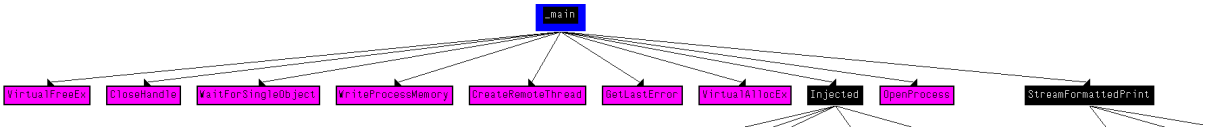
Name	Start	End	R	W	X
.text	00401000	00402000	R	.	X
.idata	00402000	004020E4	R	.	.
.rdata	004020E4	00403000	R	.	.
.data	00403000	00404000	R	W	.

2. Entry Point:



3. Main:

4. The program does not require any arguments as it does not try to access argv at all
5. The main program calls the following functions:



The black functions are the ones written by the user, one of them is injected to a remote process and one is used to print error messages.
The call subroutines which will reversed later.

6. So using the decompiler we can see some C code, in which I renamed the variables and functions. (Image provided below)
   We can see that this code is injecting code to a remote process as learned in class.
   However It does not inject a dllPath name to the remote process with the

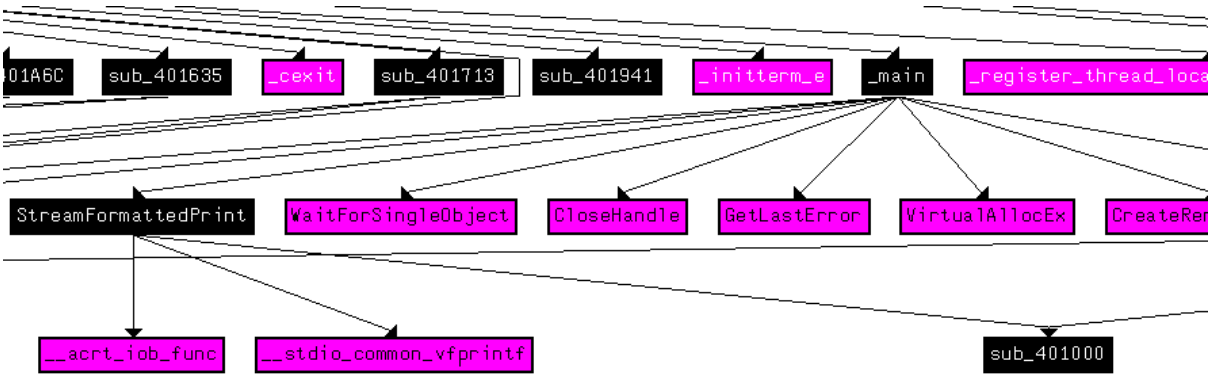
WriteProcessMemory - It writes “Injected” function to the remote process and runs it. It appears that this is some kind of payload which is going to be run on the remote process.
The main functions in usage is pretty standard:
OpenProcess - opens process 0x4D2u (PID = 1234)
VirtualAllocEx - allocates memory in the remote process, 0x1000 = 4096 bytes.
WriteProcessMemory - writes the “Injected” payload to the remote process.
CreateRemoteThread - creates a remote thread in the remote process and runs the remoteBuffer that we allocated and written to.
WaitForSingleObject - waits for the thread to finish running.
CloseHandle - closes the handle to the thread and process
VirtualFreeEx - clears the buffer we created.

The code can be seen below:

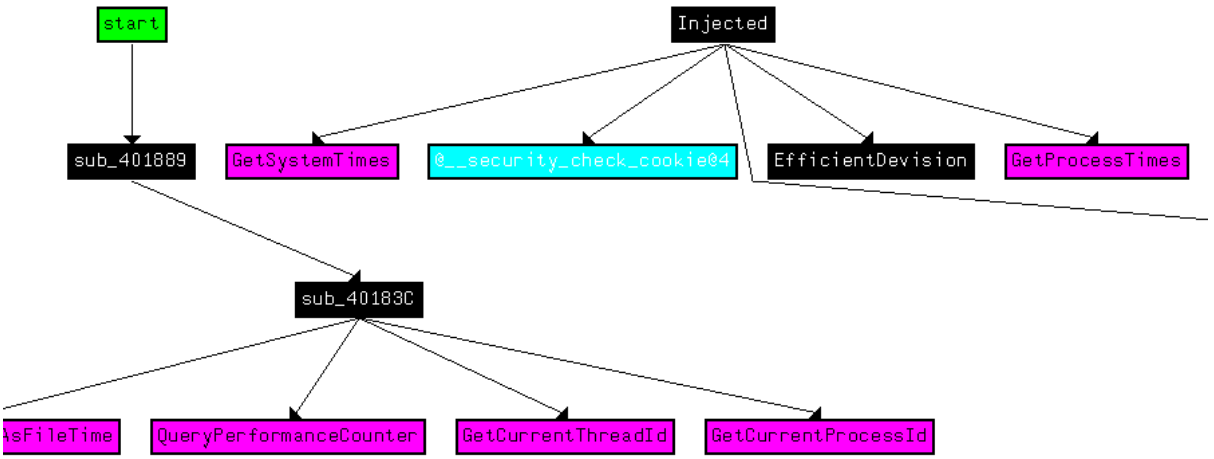
```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  HANDLE hProcess1; // eax
  void *hProcess; // edi
  char v5; // al
  DWORD (__stdcall *remoteBuffer1)(LPVOID); // eax
  DWORD (__stdcall *remoteBuffer)(LPVOID); // ebx
  char v9; // al
  char v10; // al
  HANDLE RemoteThread1; // eax
  void *RemoteThread; // esi
  char LastError; // al

  hProcess1 = OpenProcess(0x1FFFFFu, 0, 0x4D2u);
  hProcess = hProcess1;
  if ( hProcess1 )
  {
    remoteBuffer1 = (DWORD (__stdcall *)(LPVOID))VirtualAllocEx(hProcess1, 0, 0x1000u, 0x1000u, 0x40u);
    remoteBuffer = remoteBuffer1;
    if ( remoteBuffer1 )
    {
      if ( WriteProcessMemory(hProcess, remoteBuffer1, Injected, 0x1000u, 0) )
      {
        RemoteThread1 = CreateRemoteThread(hProcess, 0, 0, remoteBuffer, 0, 0, 0);
        RemoteThread = RemoteThread1;
        if ( RemoteThread1 )
        {
          WaitForSingleObject(RemoteThread1, 0xFFFFFFFF);
          CloseHandle(RemoteThread);
          VirtualFreeEx(hProcess, remoteBuffer, 0, 0x8000u);
          CloseHandle(hProcess);
          return 0;
        }
        else
        {
          LastError = GetLastError();
          StreamFormattedPrint("Failed. GetLastError = %d\n", LastError);
          VirtualFreeEx(hProcess, remoteBuffer, 0, 0x8000u);
          CloseHandle(hProcess);
          return 1;
        }
      }
      else
      {
        v10 = GetLastError();
        StreamFormattedPrint("Failed. GetLastError = %d\n", v10);
        VirtualFreeEx(hProcess, remoteBuffer, 0, 0x8000u);
        CloseHandle(hProcess);
        return 1;
      }
    }
    else
    {
      v9 = GetLastError();
      StreamFormattedPrint("Failed. GetLastError = %d\n", v9);
      CloseHandle(hProcess);
      return 1;
    }
  }
  else
  {
    v5 = GetLastError();
    StreamFormattedPrint("Failed. GetLastError = %d\n", v5);
    return 1;
  }
}
```

The code in the “Injected” function is retrieving some statistics of the injected process like the time it spent in kernel mode and time spent in user mode. Those calculations are than passed to another function which we will call “EfficientDevision” which divides the values provided 64 bits in an efficient way that could not be supported by the hardware.
Those values are than returned from the function.
However the process injection does not retrieve those values so the Idea behind this process injection is still mysterious.
If the values are returned they can be used for process monitoring and system monitoring.

7. The user functions that I suspect were written by a user and not a compiler are:



Main and StreamFormattedPrint



Injected and EfficientDevision

Those functions are user made. there could be more user made functions but they are not called directly from main or the process injection.