D-Dump: Detecting FreeLibrary Evasion Technique.

ElementalX

Contents:

1. FreeLibrary() EDR Evasion Technique.
2. Demo: DLL Loading & Unloading.
3. Offensive use of MiniDumpWriteDump technique.
4. Leverage MiniDumpWriteDump for detection.
5. Resources.

FreeLibrary() EDR Evasion Technique

Abusing FreeLibrary API in windows malware has been one of the old and predominant techniques to tamper logs and tweak EDR bypasses. One of the open-source projects which perform an elegant task of documenting these bypasses can be found here.

The fundamental idea of this technique is to load the malicious DLL inside a foreign process or the process spawned by the executable containing code that involves a call to LoadLibrary API which basically performs the loading.

There are a lot of ways where the EDR can detect the loading of libraries by placing a hook which is a dynamic manner of detecting this technique, whereas static detections and tools which focus on it, blacklist this module. This static technique can cause a huge amount of false positives also, a drawback of the dynamic technique would be something called the Unloading technique where the malicious DLL can be loaded in the running process, and then once the code inside the DLL is executed, prior placing the hook the malware uses FreeLibrary() API to unload the DLL which may look like the DLL was never originally loaded into the memory.

Demo: DLL Loading & Unloading

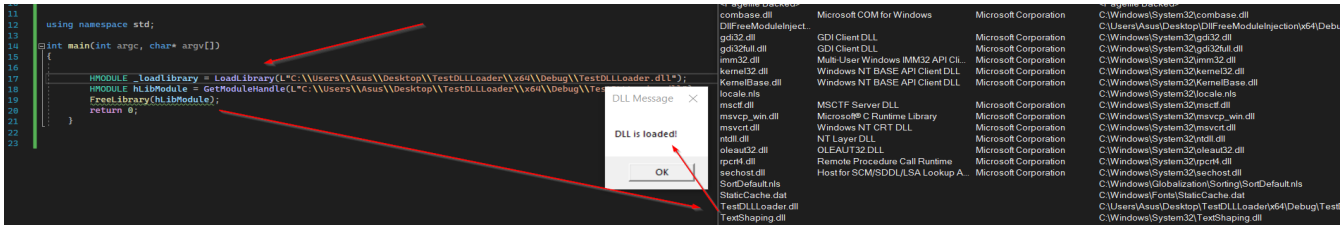
In this section, we will demonstrate the existing technique with some code.

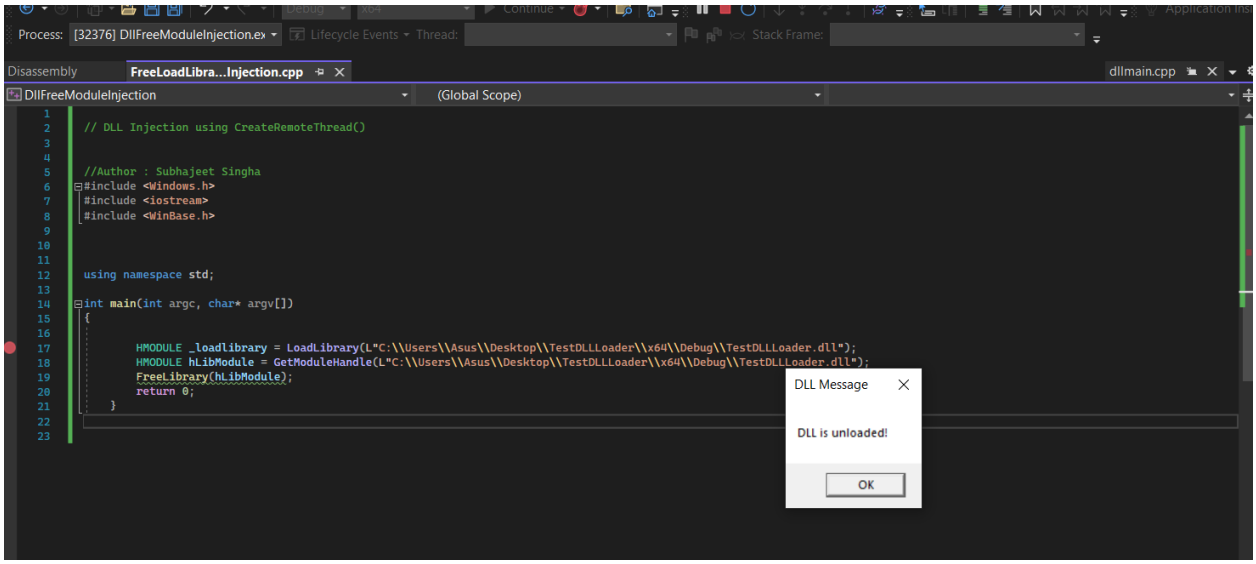
```
#include "pch.h"

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        MessageBox(NULL, L"DLL is loaded!", L"DLL Message", MB_OK);
        break;

    case DLL_PROCESS_DETACH:
        MessageBox(NULL, L"DLL is unloaded!", L"DLL Message", MB_OK);
        break;
    }
    return TRUE;
}
```

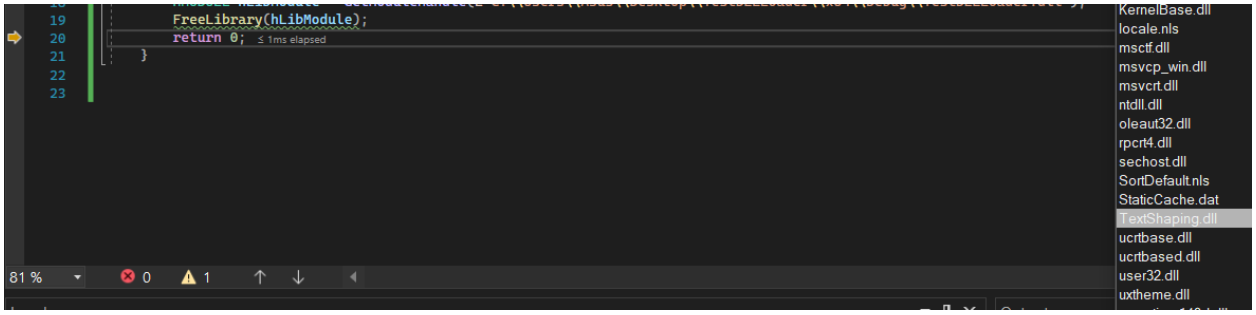
Here is a simple DLL code, which when attached or loaded inside memory will spawn a message box with the message “DLL is loaded” and when it is detached, will print “DLL is unloaded”.Now, let us load this DLL inside a process, and check whether it is properly working or not.

```
//Author : Subhajeet Singha
#include <Windows.h>
#include <iostream>
#include <WinBase.h>



using namespace std;

int main(int argc, char* argv[])
{

        HMODULE _loadlibrary = LoadLibrary(L"C:\\Users\\Asus\\Desktop\\TestDLLLoader\\x64\\Debug\\TestDLLLoader.dll");
        HMODULE hLibModule = GetModuleHandle(L"C:\\Users\\Asus\\Desktop\\TestDLLLoader\\x64\\Debug\\TestDLLLoader.dll");
        FreeLibrary(hLibModule);
        return 0;
    }
```

Here we are using the same APIs as described. Now let us execute this code and check if our DLL was loaded using Process Hacker.



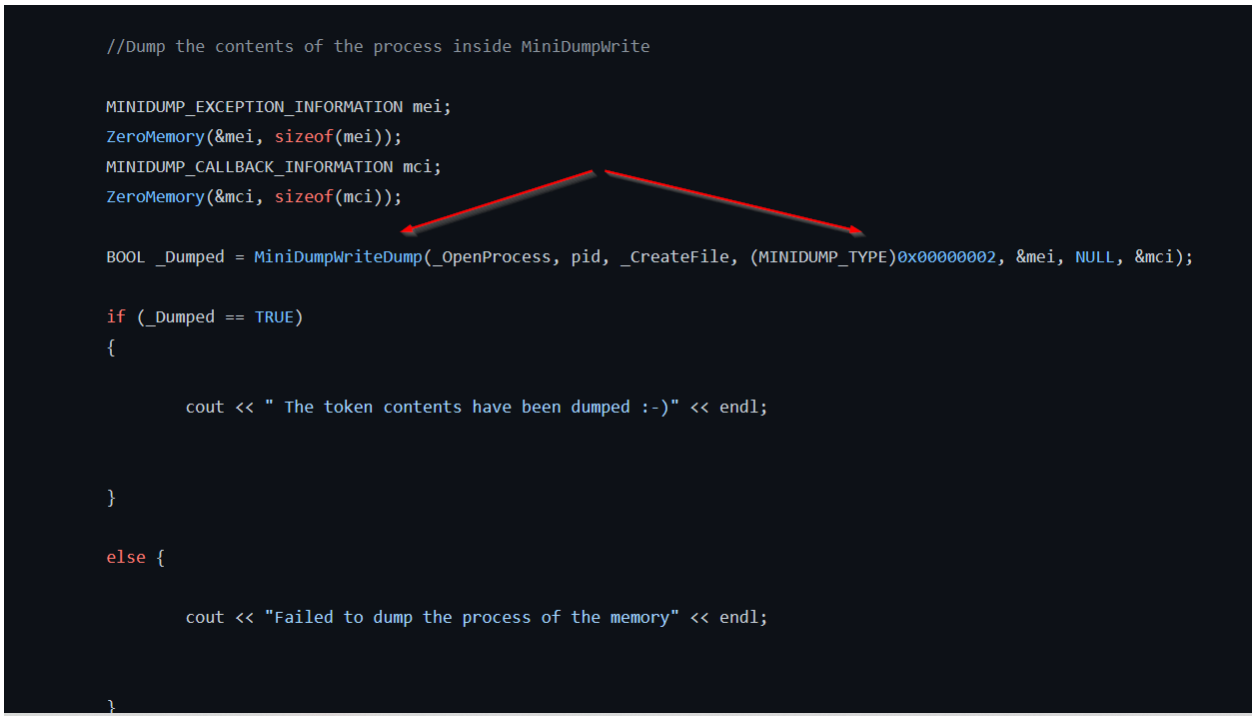




We can clearly see that our DLL was both loaded and unloaded successfully from the process. Once the DLL is unloaded and the program is running, we no more see the DLL inside the loaded module list of the process. This technique sometimes works on bypassing EDRs.

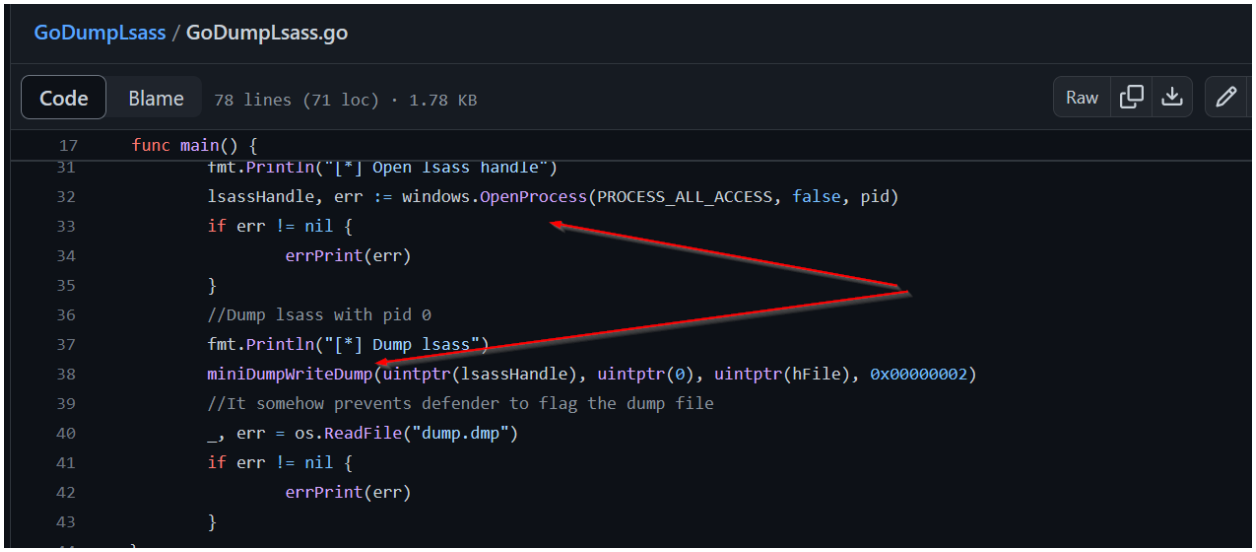
Offensive use of the MiniDumpWriteDump technique

In many cases, we have been seeing the Windows API MiniDumpWriteDump() used for dumping process data like LSASS & Winlogon, which is then exfiltrated and used by the operator for various other reasons in the malware campaigns.



```
//Dump the contents of the process inside MiniDumpWrite

MINIDUMP_EXCEPTION_INFORMATION mei;
ZeroMemory(&mei, sizeof(mei));
MINIDUMP_CALLBACK_INFORMATION mci;
ZeroMemory(&mci, sizeof(mci));

BOOL _Dumped = MiniDumpWriteDump(_OpenProcess, pid, _CreateFile, (MINIDUMP_TYPE)0x00000002, &mei, NULL, &mci);

if (_Dumped == TRUE)
{

        cout << " The token contents have been dumped :-)" << endl;


}

else {

        cout << "Failed to dump the process of the memory" << endl;


}
```

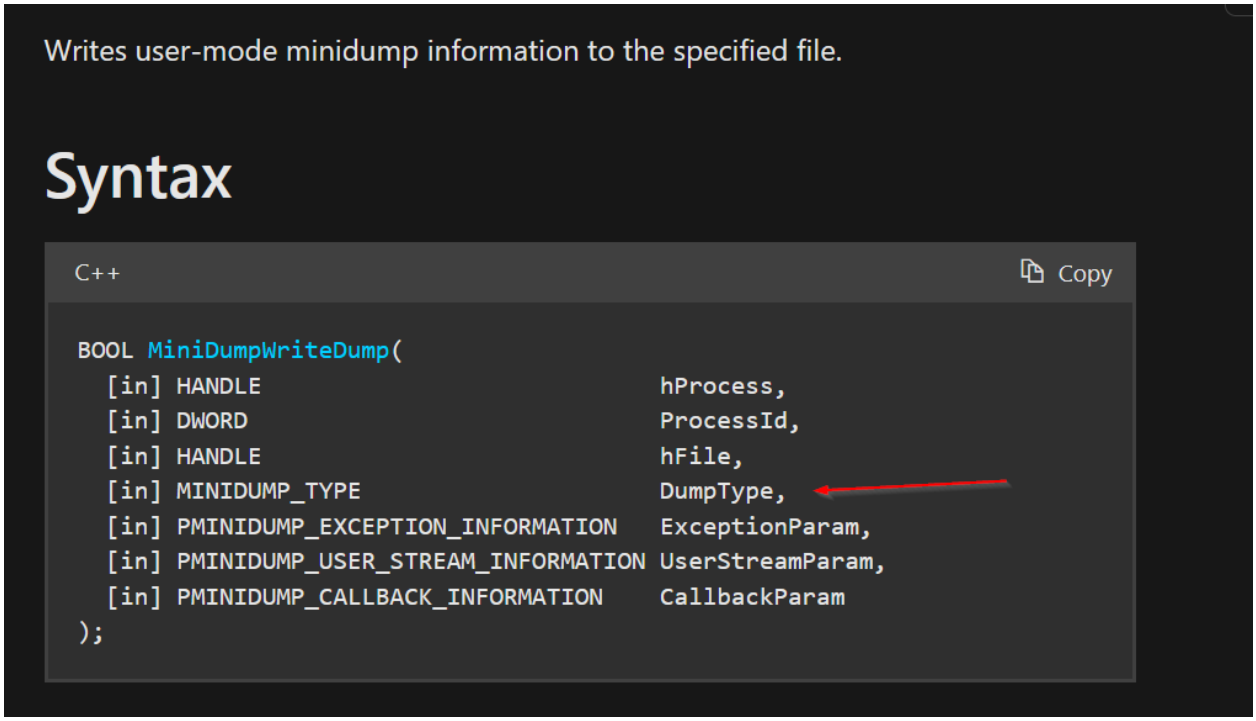
Just for an example, I wrote a small demo which uses MinidumpWriteDump() function to dump the process token and save it inside a file.



Another demo from an open-source tool using MiniDumpWriteDump() to leverage the dumping of processes using this same API known as GoDumpLSASS, and there are other projects which leverage this same API, be it a person who is writing detections, or a trained model writing detections on LSASS dumping a very generic approach would be blocking the execution of MiniDumpWriteDump function.

D-Dump: Leverage MiniDumpWriteDump for detection

This section will introduce the usage of the MiniDumpWriteDump() function for good and from a detection perspective. Before jumping to a conclusion let us explore the function from the MSDN .



Inside the function, there is an input which prompts us to an enumeration named **MINIDUMP_TYPE** which gives us the option to select the kind of information from the selected process, we want to dump inside the file.

```
typedef enum _MINIDUMP_TYPE {
  MiniDumpNormal = 0x00000000,
  MiniDumpWithDataSegs = 0x00000001,
  MiniDumpWithFullMemory = 0x00000002,
  MiniDumpWithHandleData = 0x00000004,
  MiniDumpFilterMemory = 0x00000008,
  MiniDumpScanMemory = 0x00000010,
  MiniDumpWithUnloadedModules = 0x00000020,
  MiniDumpWithIndirectlyReferencedMemory = 0x00000040,
  MiniDumpFilterModulePaths = 0x00000080,
  MiniDumpWithProcessThreadData = 0x00000100,
  MiniDumpWithPrivateReadWriteMemory = 0x00000200,
  MiniDumpWithoutOptionalData = 0x00000400,
  MiniDumpWithFullMemoryInfo = 0x00000800,
  MiniDumpWithThreadInfo = 0x00001000,
  MiniDumpWithCodeSegs = 0x00002000,
  MiniDumpWithoutAuxiliaryState = 0x00004000,
  MiniDumpWithFullAuxiliaryState = 0x00008000,
  MiniDumpWithPrivateWriteCopyMemory = 0x00010000,
  MiniDumpIgnoreInaccessibleMemory = 0x00020000,
  MiniDumpWithTokenInformation = 0x00040000,
  MiniDumpWithModuleHeaders = 0x00080000,
  MiniDumpFilterTriage = 0x00100000,
  MiniDumpWithAvxXStateContext = 0x00200000,
  MiniDumpWithIptTrace = 0x00400000,
  MiniDumpScanInaccessiblePartialPages = 0x00800000,
  MiniDumpFilterWriteCombinedMemory,
  MiniDumpValidTypeFlags = 0x01ffffff
} MINIDUMP_TYPE;
```

Here, we can see that there is a value known as MiniDumpWithUnloadedModules.

```
MiniDumpWithUnloadedModules
Value: 0x00000020
0x00000020. Include information from the list of modules that were recently unloaded, if this
information is maintained
by the operating system.
```

As the documentation says, we can dump the recently unloaded DLLs from the target process into a dump.

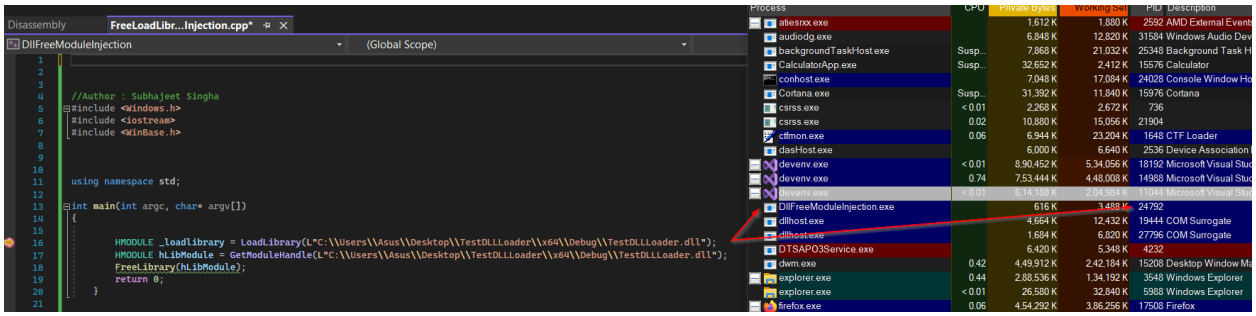
We will leverage this parameter to dump all the loaded modules from a malicious process and check out if our custom DLL which was just loaded gets dumped or not.

```
//Author : Subhajeet Singha

#include <Windows.h>
#include <iostream>
#include <DbgHelp.h>

#pragma comment (lib, "Dbghelp.lib")

using namespace std;


int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{


    wcout << "Welcome to Free-Lin detector : " << endl;
    wcout << "Please enter the specific windows process, you are suspecting :  " << endl;
    DWORD processID ;
    wcin >> processID;
    HANDLE _OpenProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, NULL, processID);
    if (_OpenProcess) {


        cout << "The handle to the suspected malicious process has been acquired ..." << endl;



    }
    else {

        wcout << " Failed to Open the process ...." << endl;
    }

    //Create a file in which the dumped contents are to be written

    HANDLE _CreateFile = CreateFile(L"C:\\Users\\Asus\\Desktop\\process-contents.txt", GENERIC_READ | GENERIC_WRITE, FILE_SHARE_WRITE, NULL, CREA
```

```
}

//Create a file in which the dumped contents are to be written

HANDLE _CreateFile = CreateFile(L"C:\\Users\\Asus\\Desktop\\process-contents.txt", GENERIC_READ | GENERIC_WRITE, FILE_SHARE_WRITE, NULL, CREATE_N


//Using MinidumpWriteDump to dump the unloaded DLLs

MINIDUMP_EXCEPTION_INFORMATION mei;
ZeroMemory(&mei, sizeof(mei));
MINIDUMP_CALLBACK_INFORMATION mci;
ZeroMemory(&mci, sizeof(mci));

BOOL DLL_Dumped = MiniDumpWriteDump(_OpenProcess, processID, _CreateFile, (MINIDUMP_TYPE)0x00000020, &mei, NULL, &mci);

if (DLL_Dumped) {

    cout << " The process's unloaded DLLS are dumped " << endl;

}
else {

    cout << "Dumping Failed " << endl;
}
```

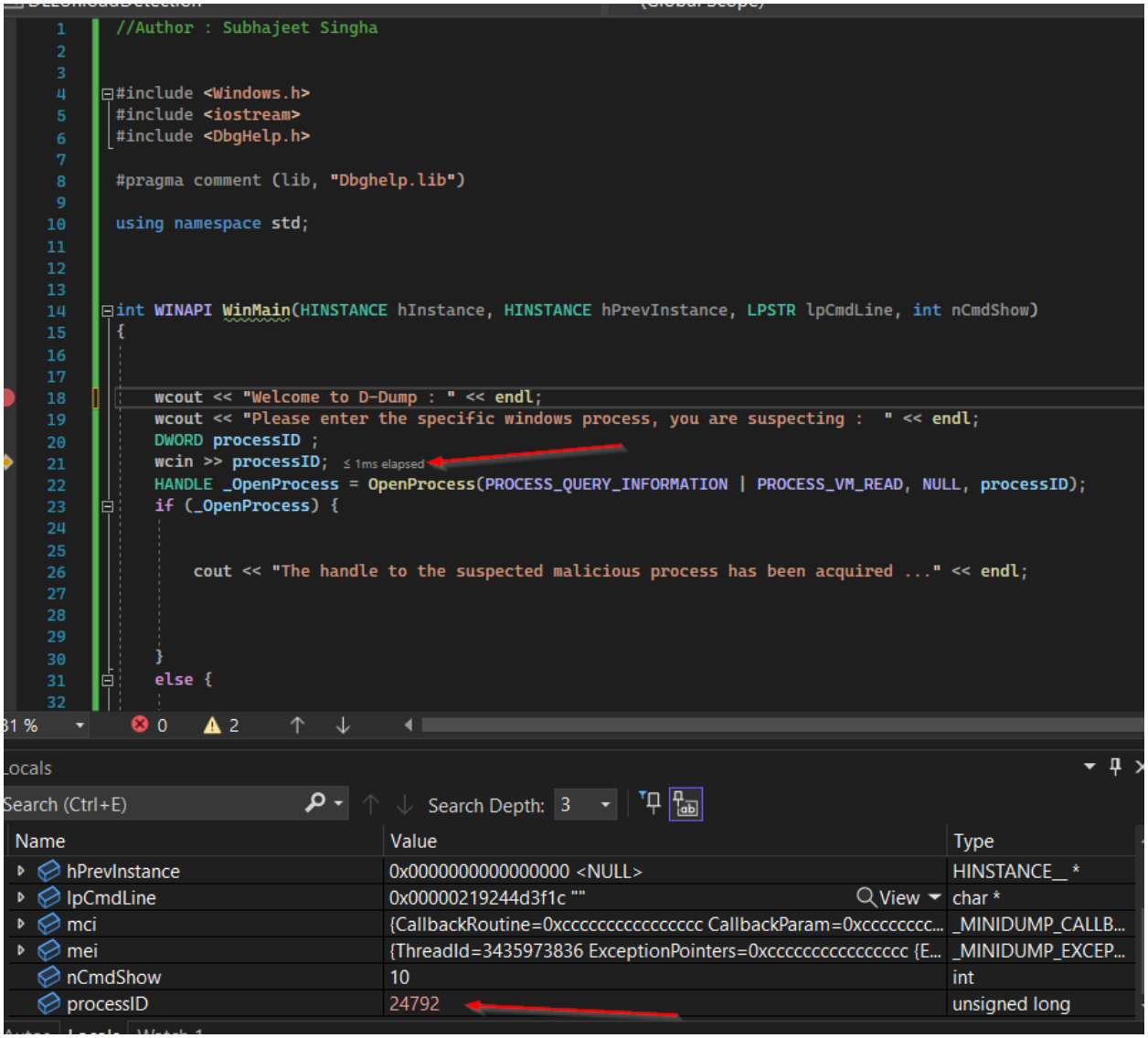
Here in this code, our code is not dumping the LSASS process, but a normal process, which has the FreeLibrary used to unload the DLL during execution.

Let us run our code and check out if our code dumps the list of modules and the target module which was unloaded.



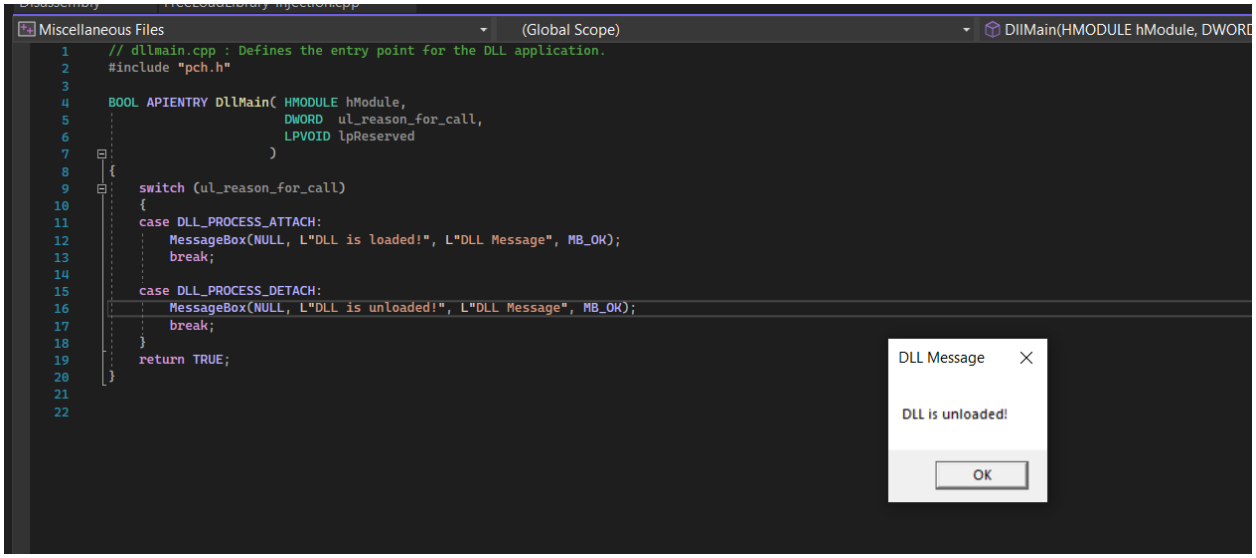
After running the process and setting a breakpoint, we finally load the DLL.

Now, let us try to execute our D-Dump method code, and enter the same process ID which is 24792 as an argument to the OpenProcess.



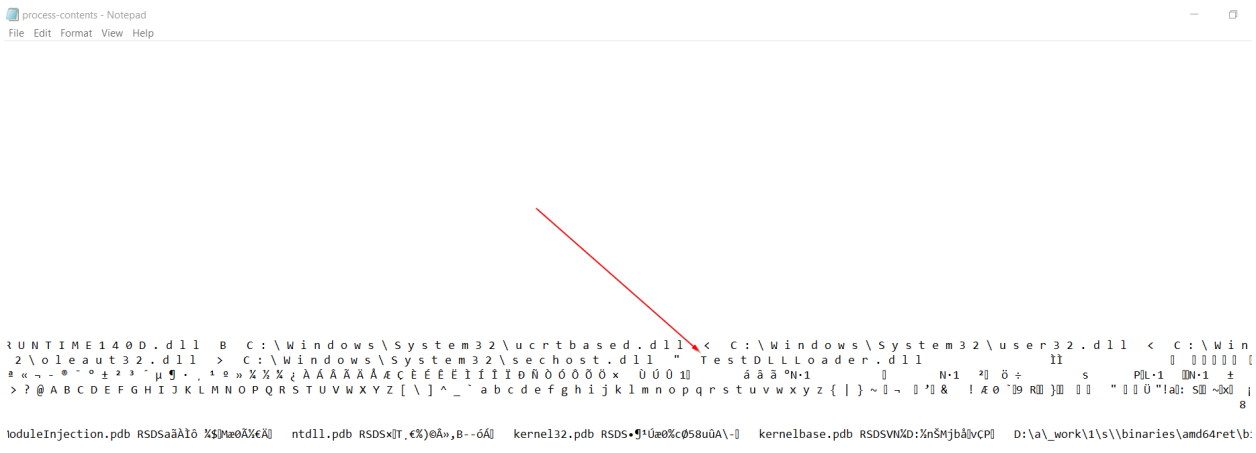
Now, let us step in and go with the flow.

Finally, our DLL has been loaded.



And, then our DLL has been unloaded.

Now, let us check for the file which was dumped and try to find our DLL which was unloaded.



Looks like our DLL which was unloaded using FreeLibrary() has been dumped inside the dump file, making the D-Dump technique a valuable one to detect the unloading of DLLs using FreeLibrary() API.

Resources

- MiniDumpWriteDump from MSDN.
- Unloading Module with FreeLibrary().