**Test**

Injection techniques come in all [shapes and sizes](https://i.blackhat.com/USA-19/Thursday/us-19-Kotler-Process-Injection-Techniques-Gotta-Catch-Them-All-wp.pdf), some are loud, some are quiet, and some even allow to load a whole DLL!

Depending on the situation and our needs, sometimes we need to execute a [shellcode](http://phrack.org/issues/62/7.html#article), and sometimes we need a full blown [PE file](http://phrack.org/issues/62/13.html). Once we do that, [we cannot use traditional methods](https://docs.microsoft.com/en-us/windows/win32/dlls/using-run-time-dynamic-linking), and must get a bit creative with how we achieve even the simplest call to an exported function.

Your mission is to load a simple DLL:

BOOL APIENTRY DllMain(HMODULE hModule,

DWORD ul\_reason\_for\_call,

LPVOID lpReserved

)

{

switch (ul\_reason\_for\_call)

{

case DLL\_PROCESS\_ATTACH:

case DLL\_THREAD\_ATTACH:

case DLL\_THREAD\_DETACH:

case DLL\_PROCESS\_DETACH:

break;

}

return TRUE;

}

extern "C" \_\_declspec(dllexport) void TestMe(LPVOID lpUserdata, DWORD nUserdataLen) {

OutputDebugStringA("Hello there :)");

}

To your main executable and call *TestMe* without using LoadLibrary, GetProcAddress, **and** without the DLL touching the disk. This means you may choose to have the DLL as a resource and read it, or you may host it in some server and fetch it, or even have the DLL on disk and the main executable will read the DLL to a buffer from a file. The point is to have the DLL as a buffer.

From this point, you should document **the method** you chose, **why you chose it**, what kinds of **difficulties and limitations** it has and **how would you possibly solve them** if there are any.

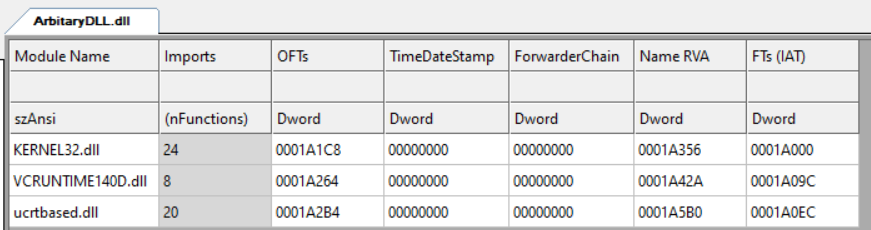
Write a short report on these questions, and your process since we will go over it together.

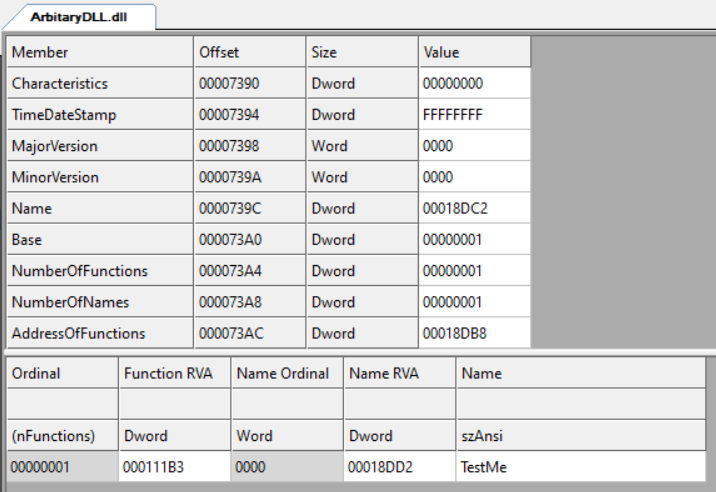
**Note:** You don't have to write it all from scratch! You can use existing implementations, but keep in mind that you **should** know how they work and explain it. Once done we will have a short session talking about your implementation decisions.

**Solution:**

First I need to implement a simple dll as described above.

Quick compile shows that the dll artifact have the following import and export entries.





I do want to remove the CRT dependencies I've got from the default linkage settings. This can be done by link against the lib version of the CRT instead of the dll version.

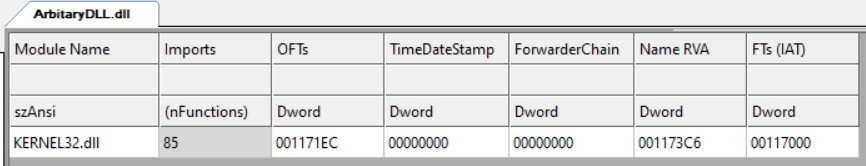
So changing the Runtime Library setting in the C++/Code Generation tab would get the desired result.



\*The equivalent setting for Release builds for Win32 and x64 should have the release lib version

But here I am compiling Win32 Debug version of the dll just to make my life easier.

Now, the import table looks like this:



The next step would be to create a simple DLL loader that would be responsible to map a given unmapped PE structure into memory.

So, in the same solution I would add a new PE loader project.

Warning Level #4 and Enable Warnings as Errors set for the two projects for all configurations.

Moreover, I want to disable all features that potentially modify the final DLL module, So I am going to disable the default (In Debug mode) Incremental Building in the linker settings. This option adds a .textbss section to the PE that which contain some padding that is required for more fast linkage phase. By disable it, I gain more control over my binary and make my life easier when parsing the sections of the PE into memory.

Iam not sure if its possible right now, but I want to link statically against kernel32.lib

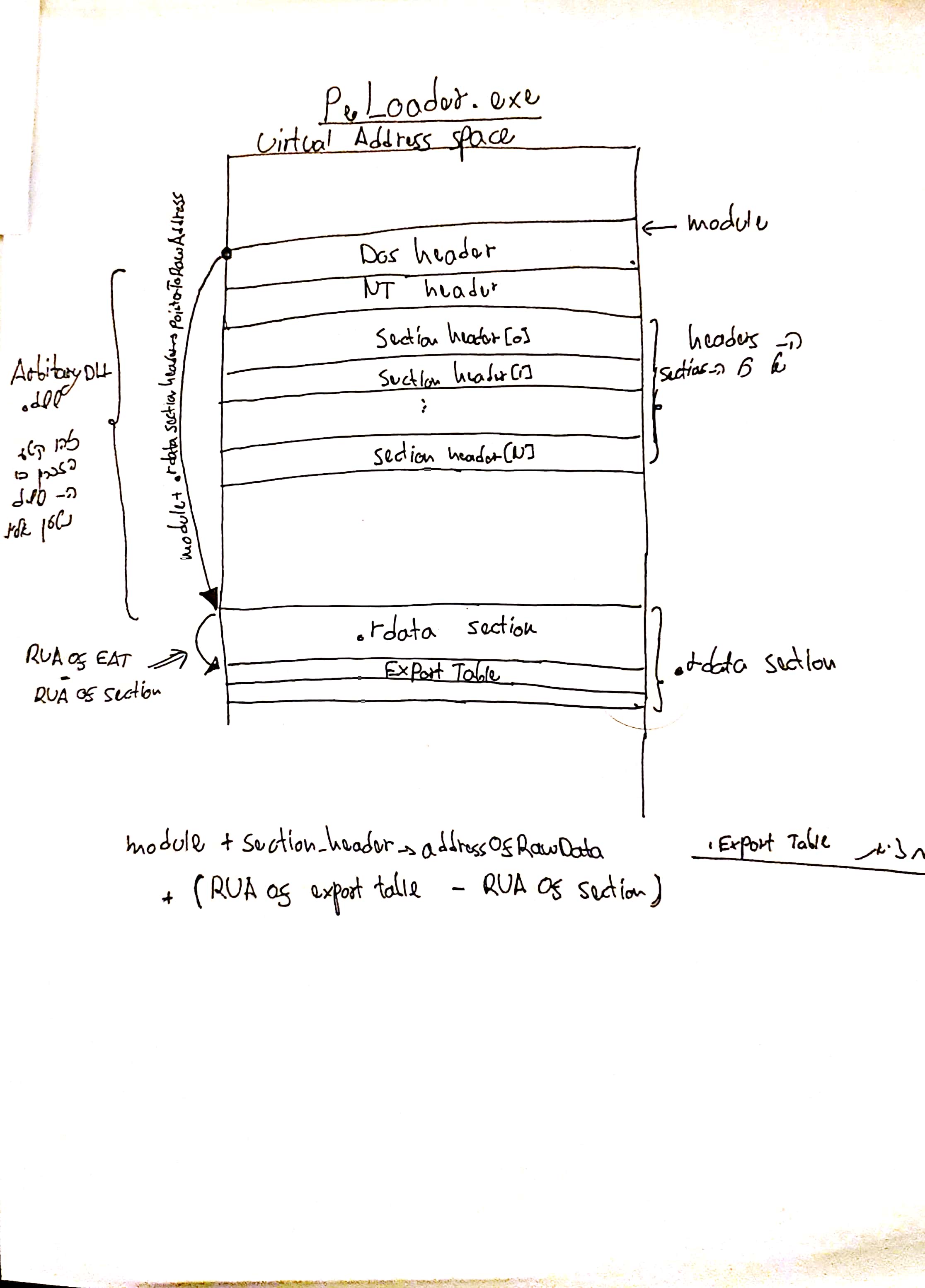
I do succeeded to remove entirely the import descriptor, but than I lost the ability of using any native kernel32 api's.

It took me some time to figure out how to reference the RVA of entities within particular sections.

This can be achieved by using the following formula:

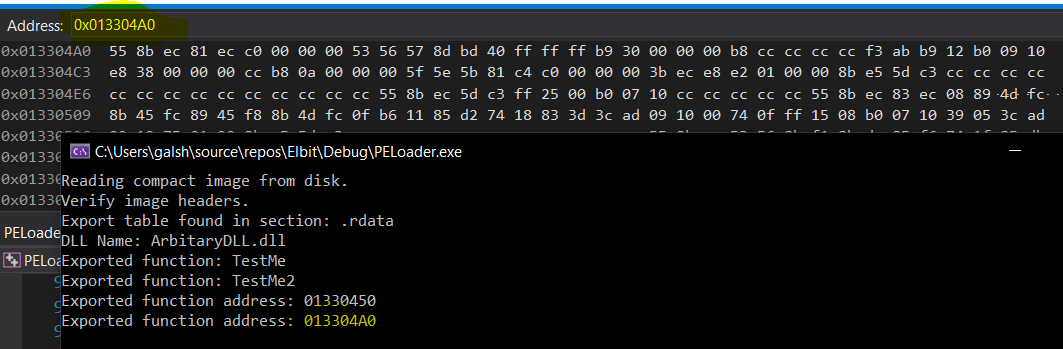
base\_module + section\_header->AddressOfRawData + (RVA of entity – RVA of section of that entity)

It is important to notice that when dealing with RVA, We must check in which section the given RVA exists. For example, the RVA associated with the AddressOfFunctions of the Export Table Directory entity, are RVA's that associated relative to the .text section and not the .rdata section as a user can misleading assume (Just because the parsing was done according to that section earlier).

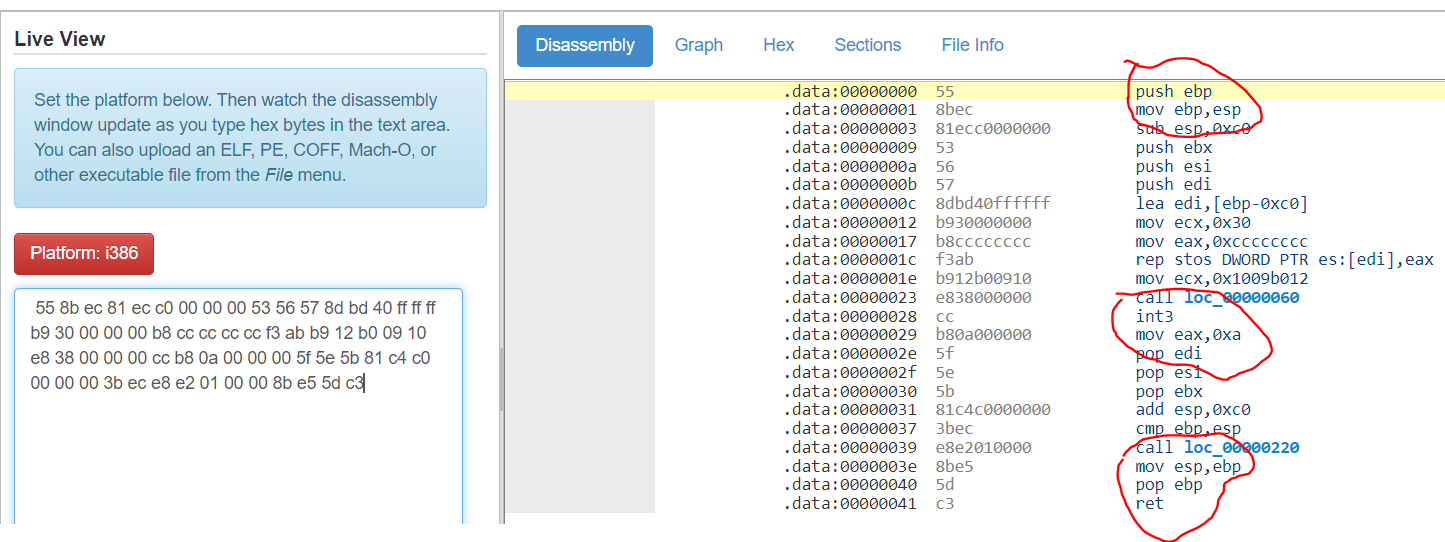


**Iterating the exported functions:**

After assuring the addresses are relative to the .text section, I've grabbed the pointers and examined the raw bytes stream in the PE in order to verify I am getting legit function pointers.



Putting those bytes in a disassembler shows that the given callback is associated rightfully with the second exported TestMe2 method.

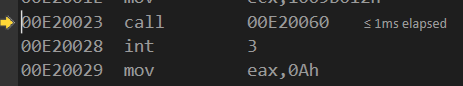


This image above approve that the parsing was right. The top most circled is the function prologue and the stack preparation.

The middle circled is the corresponding bytes for the calls of \_\_debugbreak() and return 10 (0x0a).

The bottom is the stack unwinding the epilogue.

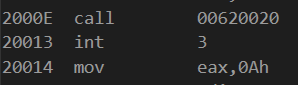
All I need to do right now, is to allocate executable pages for those bytes and make direct call into them.



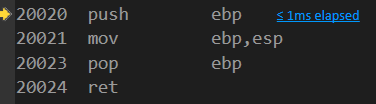
The above call cause an access violation exception, I believe it related to the some enabled mitigations of my DLL.

/GS disabled, so no canary, still got the same exeption.

It seemd like a call made by other component, so will set the No Entry Point settings in the linker Advanced tab + Disable CFG and set Ignore All Default Libraries for the linker as well and rerun the loader.



Call offset changed and it's looks like it does nothing right now after all those changes.





And it's seems to fix my previous access violation and **call my DLL exported function just as expected.**

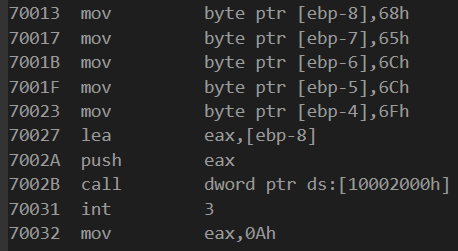
I do have one last thing to handle.

I've noticed that there is a problem when calling api's originated from Kernel32, getting access violation on such scenario, not sure if it's because of the call itself or the arguments such as strings that I have not mapped yet.

In order to eliminate the chance it's related to the string parameter associated with the call, I can try force the compiler to build the string on the stack by using char array instead of putting the string in one of the global pe sections where strings resided (can't remember exactly where it is right now).

So "Hello there 😊" can be converted to char msg[] = {'H', 'e', 'l'. 'l', 'o', ' ', 't', 'h', 'e', 'r', 'e', ' ', ':', ')'};

This would make the string getting generated directly into the stack.



So now, I am building the string right into the stack but iam still getting the exception when getting into the call statement, so I believe the proper way to fix it is so manually map the IAT in the import directory, basically what I mean is that when the PE is on disk, the OriginalFirstThunk array and the Thunk array points to the same address. I can change the IAT to point to the virtual addresses of kernel32 since kernel32 is mapped for the same address for all processes every system reboot.

And this can be achieved using parsing the TEB->PEB->loaded modules list.