Code Generation:: Homework Documentation

Linking and compiling…

The files have been compiled; object files of file1.c, file2.cpp, file3.cpp have been created.

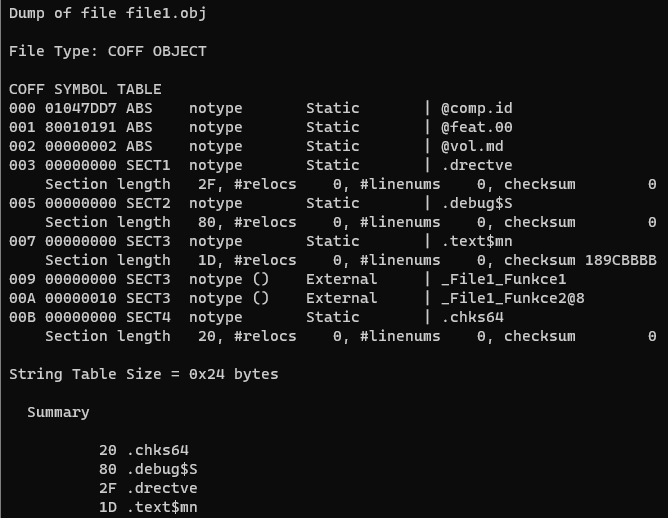
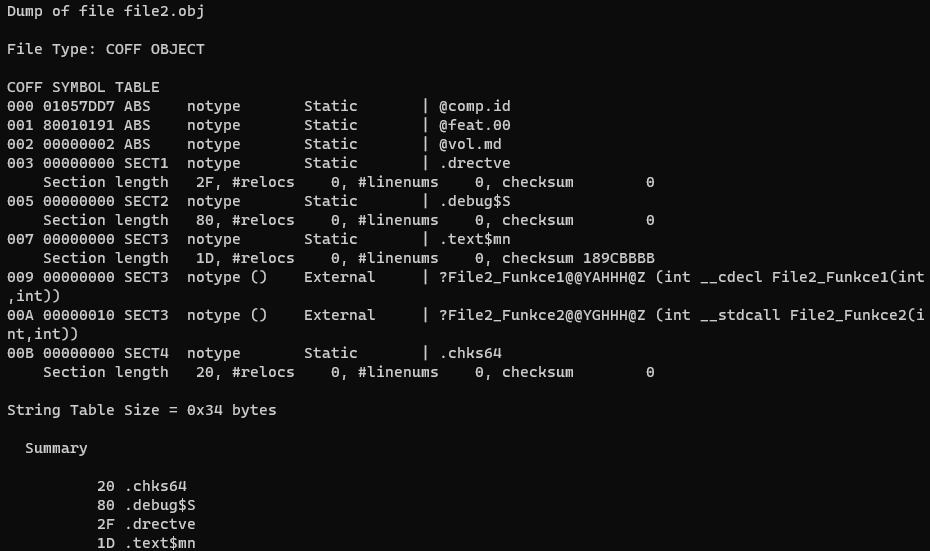
Dumping symbols from object files…

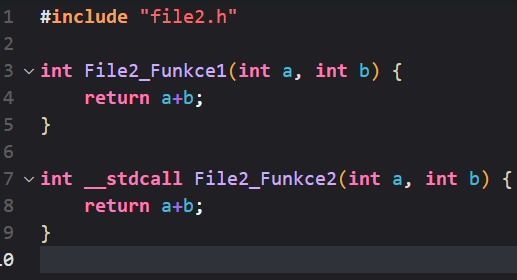
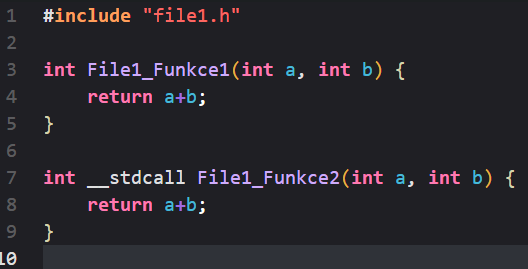
Commands used:

dumpbin /SYMBOLS file1.obj

dumpbin /SYMBOLS file2.obj

The output of command 1:

  
  
The output of command 2:  


When comparing outputs, we see different results of object files, although we are dealing with the (almost) same source code:  
  
file1.c: file2.cpp:  
  
  
The difference is the name of functions in both object files.

file1.obj



file2.obj:  

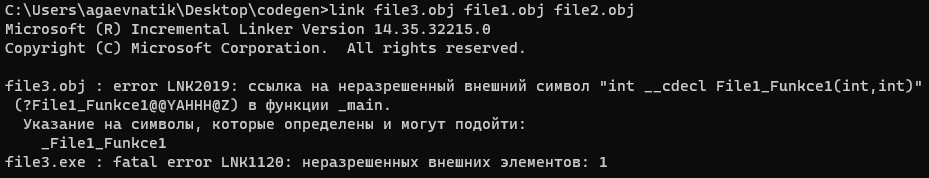

This happens, because of a so-called “name-mangling” (a technique used in programming to change the name of a variable or function in a unique way to avoid naming conflicts). This kind of behavior doesn’t affect that much, but it could affect the compilation process when dealing with multiple other files.

Linking into an executable…

By using this command, we will link the object files into one executable:

link file3.obj file2.obj file1.obj

QUESTION 1 EXPLANATION START

We expect to get an executable file, but we will get the fatal error LNK1120.  


The error occurs because we have faced a function mismatch in object files:

File1.obj:



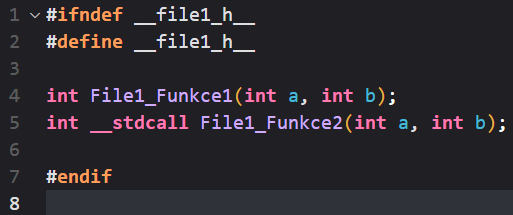
File3.obj:



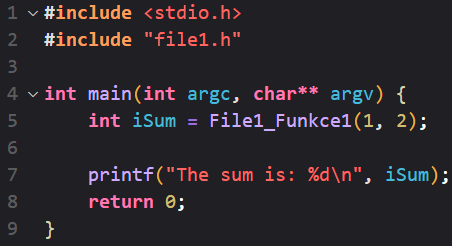
The function naming is totally different. This is the source of mismatch which results in errors in linking.

Now we need to identify the source of the source the error…

Let’s check file1.h

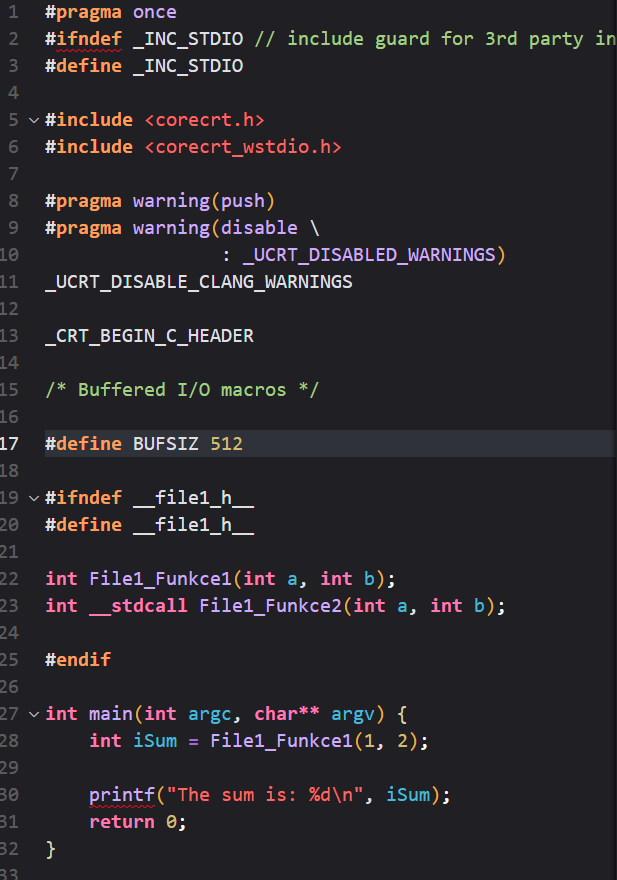


The file3.cpp has the operation: #include, which includes a header file file1.h, that inherits functions from file1.c source file.



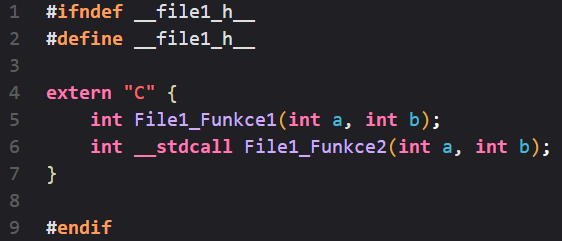
Therefore, whenever we compile the code, the compiler first copies the contents of the header files and pastes them into the main file (file3.cpp in our case). Hence it looks like we are copying the content of the C file into a CPP file (that moment when we tried to link object files together, we linked file1.obj (which came from the C file) and file3.obj (that came from the CPP file)).

Hence file3.cpp looks like this:

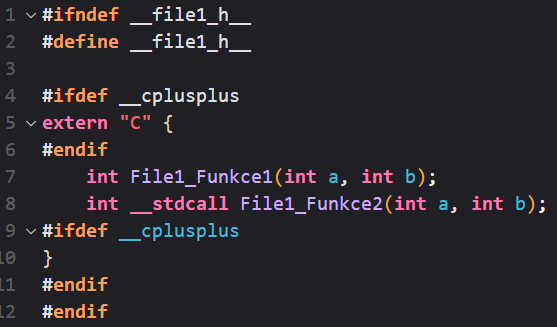


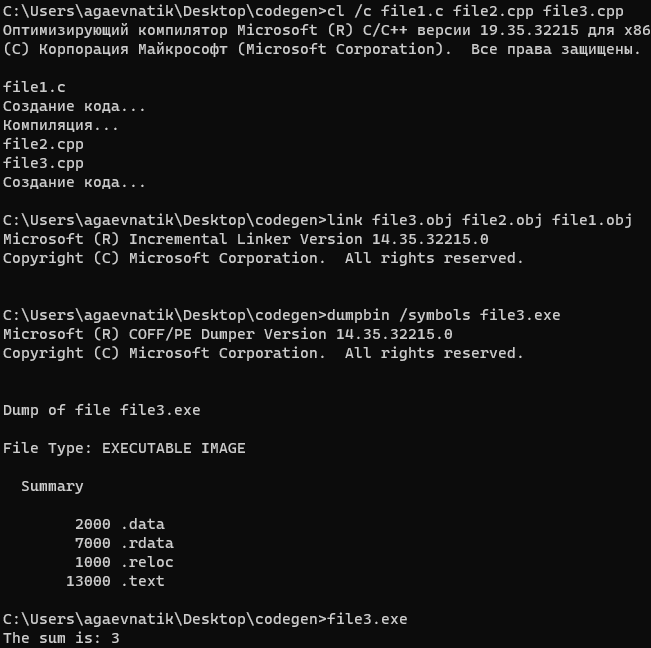
So, to fix this issue we need to modify the header file, so it could be compatible with CPP and C object files.   
\* When using the cl /c command to compile, the system automatically selects a compiler type for a specific source code. Therefore a c file will be compiled via gcc, and cpp will be compiled via g++, resulting in a potential mismatch of object files.

We could add the extern here, but this would cause an error because the C compiler DOES NOT recognize std::string from C++ (there are no string types in C);



This approach might not work (it might link successfully, but if we try to compile the c file again, it will cause an error described earlier). Luckily there is a predefined macro \_\_cplusplus that returns the specific value that tells whether the c++ compiler was used or not. This is called a conditional compilation.

So if we change a little bit, we get:  
  
We check if we use the cpp compiler, then we would extern those functions and they would behave like cpp functions, avoiding naming mismatches, else we will see them as c functions.

Recompile… Relink… Redumping…   


SUCCESS!!

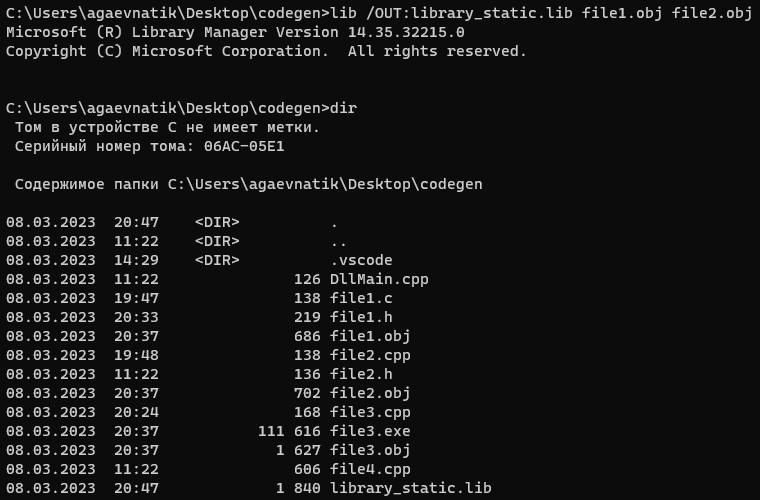
Static library

QUESTION 1 EXPLANATION END

Now we try to combine all the object files into one because it is not convenient to deal with them, especially when there are hundreds of them.

Using the command:

lib /OUT:library\_static.lib file1.obj file2.obj

will generate the static library file.

Now if you want to link something new (something that has been edited and was not part of the library generated), you can just use this command:  
In our case:

link file3.obj library\_static.lib

Compiling with a static library

Just like a regular compilation, but with the library now.  
The command:

cl /o file3.exe file3.cpp library\_static.lib

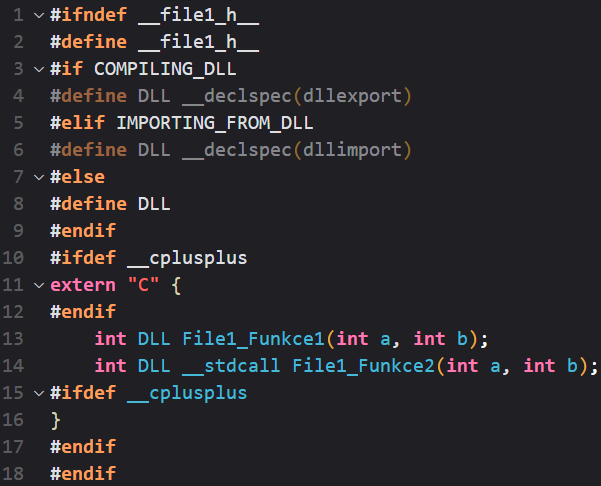
Dynamic library

It is almost like the static one, but it has differences though.

This time we turn the library into a dynamic one. For this purpose, we need to mark elements of the code (functions, vars…) that our library will provide to the export.

For this, we use the \_\_declspec(dllexport) or \_\_declspec(dllimport) keywords.

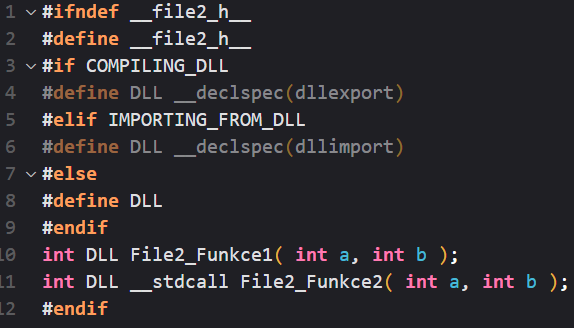
In our case we will modify the C header file like this:



Here we identify whether the functions are to import or export. Using these macros we can define types of functions without of need to re-write the code. Macros do it themselves.

If to import – the DLL will be equal to \_\_declspec(dllimport), else if - \_\_declspec(dllexport) and else nothing.

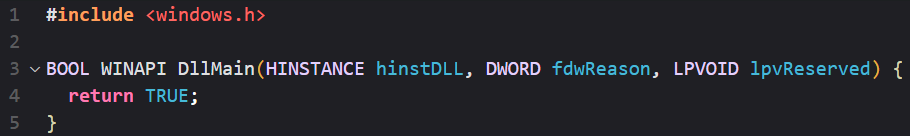
We do that to the file2.h as well…



Now compile everything, but this time we add the flag COMPILING\_DLL = 1 (to export) using this command:

cl /c file1.c file2.cpp file3.cpp /DCOMPILING\_DLL=1

for this time we need to locate DllMain.cpp file. It has a code necessary for dll construction.

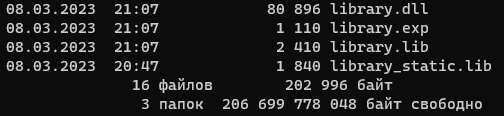


It is like a main function but for dll. It gets called whenever something happens to libraries.

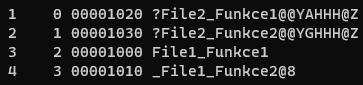
So we add the file to compilation…  
 cl /c file1.c file2.cpp file3.cpp /DCOMPILING\_DLL=1 DllMain.cpp

now we build the DLL using this command:

link /DLL file1.obj file2.obj DllMain.obj /OUT:library.dll /IMPLIB:library.lib



With this, the DLL exports functions. Use the dumpbin /explore command.



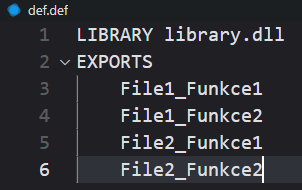
The functions are exported (but they have weird naming problems), because, when compiling, we did set the COMPILING\_DLL=1. Therefore, before compiling the source code looked like this:   
 int \_\_declspec(dllexport) File1\_Funkce1(int a, int b);

As you can see, the function is set to be exported.

QUESTION 2 EXPLANATION START

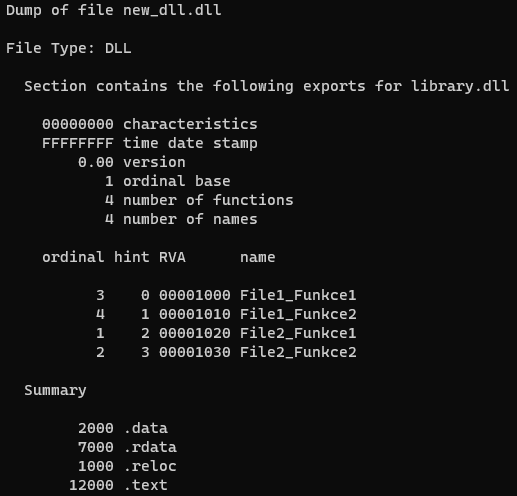
The linker is unaware of the DLL macro. DLL there is nothing, it is just a replacement for \_\_declspec().

Also, to check the export functions we can create a .def file, where we would functions that we think are gonna be exported.



Now we use this command to link files and produce new dll files.

link /DLL /OUT:new\_dll.dll /DEF:def.def file1.obj file2.obj file3.obj DllMain.obj

After that, we get the new\_dll.dll file. When: dumpbin /exports new\_dll.dll, we get:

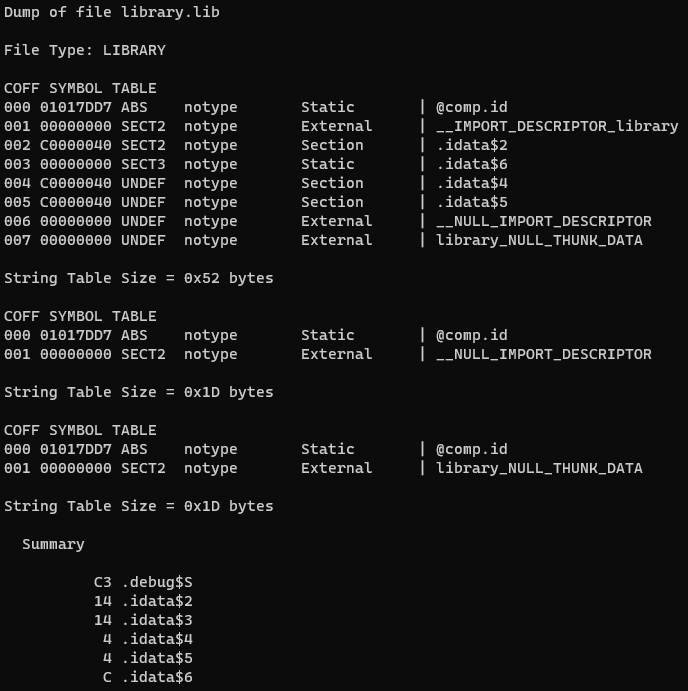
Which shows us that the functions are indeed exported. If we insert a function that does not exist in to .def file, it will cause an error during linking.

QUESTION 2 EXPLANATION END

Okay, so we have now a library.lib and library\_static.lib.   
What is the difference?  
Let’s dumpbin them.

QUESTION 3 EXPLANATION START

dumpbin /symbols library.lib



dumpbin /symbols library\_static.lib



library.lib only exports two kinds of stuff: \_\_IMPORT\_DESCRIPTOR\_library and library\_NULL\_THUNK\_DATA.

On the other hand, library\_static.lib exports four symbols: ?File2\_Funkce1@@YAHHH@Z, ?File2\_Funkce2@@YGHHH@Z, \_File1\_Funkce1, and \_File1\_Funkce2@8.

This means that library\_static.lib has object files that export functions File2\_Funkce1, File2\_Funkce2, \_File1\_Funkce1, and \_File1\_Funkce2. Meanwhile, library.lib does not contain any object files that define and export these functions.

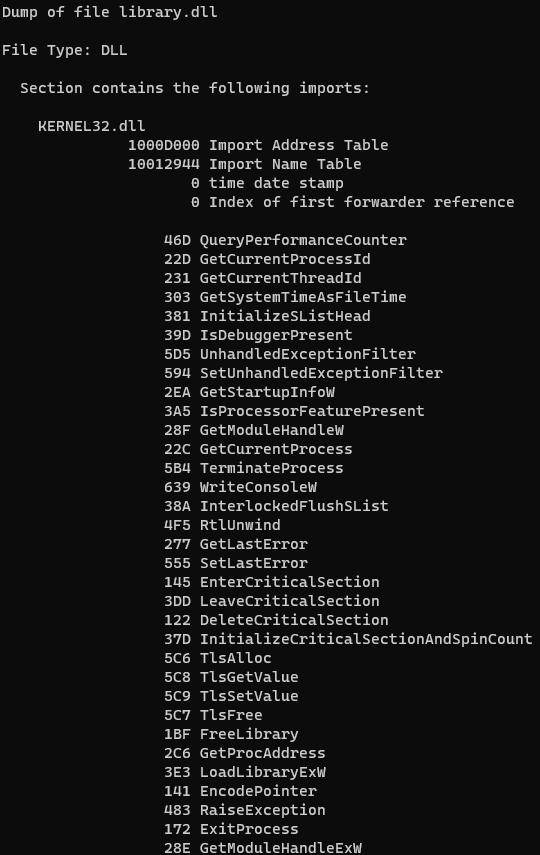
QUESTION 3 EXPLANATION END

QUESTION 4 EXPLANATION START

Using the command

dumpbin /imports library.dll

We get a list of imports from that library.dll contains when we linked with the program.



We can see that the program is importing many functions from kernel32.dll.

QUESTION 4 EXPLANATION END

QUESTION 5 EXPLANATION START

The location of the library is important. If the program can’t find the library to link with, it will cause a problem. Fortunately, OS has some operations to locate the library.

The places where the .dll could be stored are (following an order):

1. DLL Redirection
2. API Sets
3. SxS manifest redirection
4. Loaded-module list
5. Known DLLs
6. (for Windows 11) The package dependency graph of the process. Dependencies are searched in the order they appear in the manifest
7. The folder in which the application loaded.
8. The system folder. (C:\Windows\System32)
9. The 16-bit folder
10. The Windows folder (C:\Windows)
11. The current folder
12. The directories that are listed in the PATH environment table

The OS performs standard search orders for unpacked apps.

Using DLLs allows developers to create an executable that is less in size. In addition, it allows the developers to make changes in dlls without of need to recompile the entire project. It can be also used by several programs at the same time. Updating the app is also relatively easier. When the executable works, it scans for a dll library to get functions that the executable needs. To accomplish this, the OS performs recursive scanning for the library (the directories it scans are described above).

Despite all the benefits, some hackers might use this as an opportunity to attack.

DLL searching is the process of searching for DLL files when a program requests them. It can have security implications such as DLL hijacking and DLL planting, where an attacker replaces or places a malicious DLL in a directory that is searched by the operating system and I’m pretty sure – that doesn’t sound good at all...😰

Speaking of DLL Hijacking…

There are multiple ways to trick a user to download (sometimes accidentally) malicious DLL. The browser download directory is a great example, because first – we all use browsers, and second – some browsers have allowed websites to drop files into the directory by default (without asking permission from the system), because it might run in the PuTTY process.

There are two types of DLL Hijackings – regular and indirect.

Regular DLL hijacking:

Initially, PuTTY was vulnerable to regular DLL hijacking, where all versions up to and including 0.67 would load DLLs from the local directory containing the PuTTY executables. This could be exploited by an attacker who tricked a user into downloading a malicious DLL and running any PuTTY tool directly from their download directory, allowing the attacker's code to run within the PuTTY process.

Mitigation using SetDefaultDllDirectories():

To mitigate this vulnerability, the PuTTY team implemented a fix in the executables by calling the SetDefaultDllDirectories() function for fully-patched Windows Vista and later versions. This reduced the risk of loading DLLs from untrusted locations. However, the mitigation was not available for unpatched or older versions of Windows, which were still advised not to run the PuTTY tools from untrusted locations.

Indirect DLL hijacking:

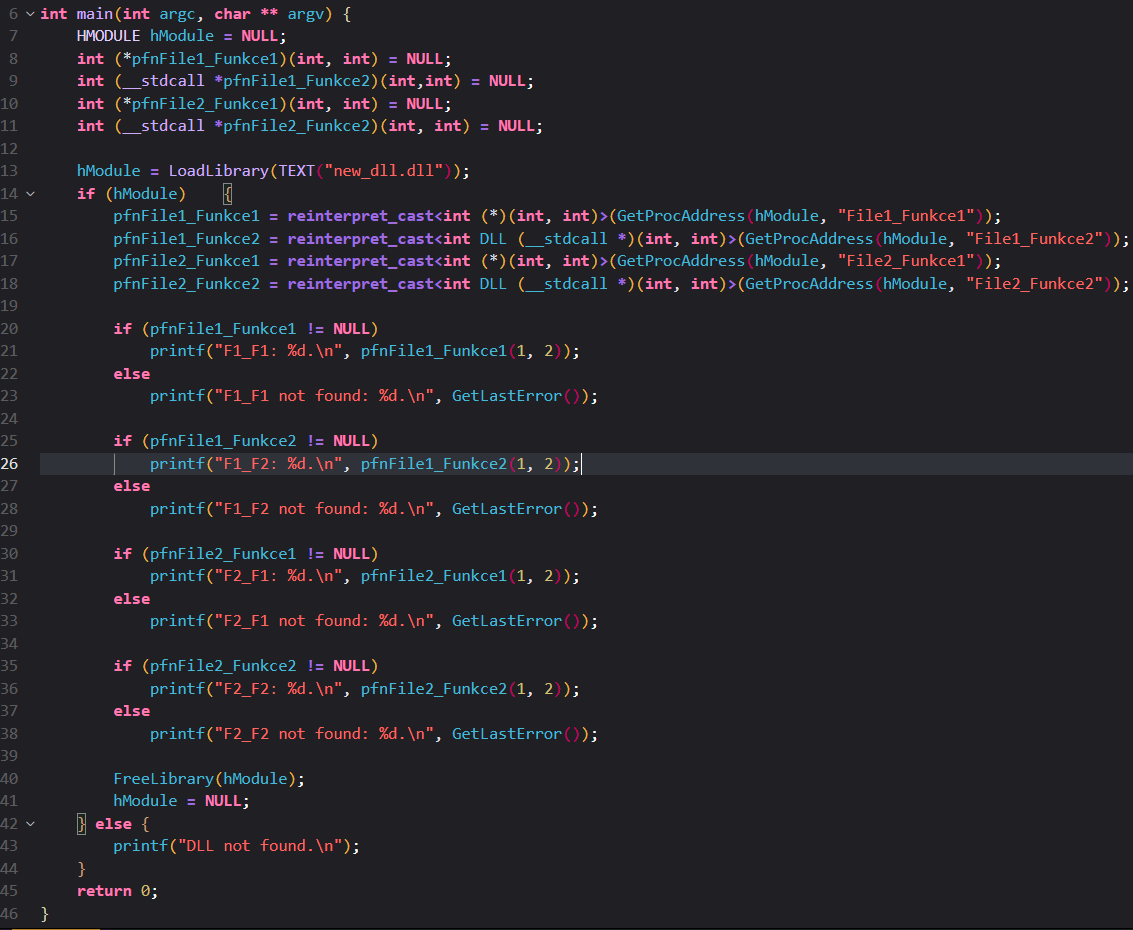
Despite the fix introduced in version 0.68, further problems came to light, indicating that PuTTY was still vulnerable to indirect DLL hijacking. The attacker had modified their technique, and PuTTY 0.68 was found to be vulnerable to loading specific DLL names at startup time, such as WINMM.DLL, WINSPOOL.DRV, BCRYPT.DLL, and SSPICLI.DLL. This vulnerability could still be exploited through a web browser's download directory, similar to the previous regular hijacking scenario.

QUESTION 5 EXPLANATION END

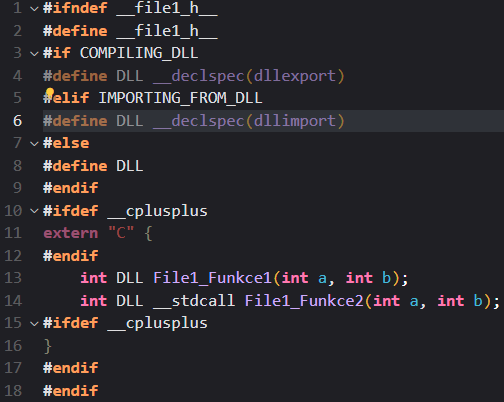
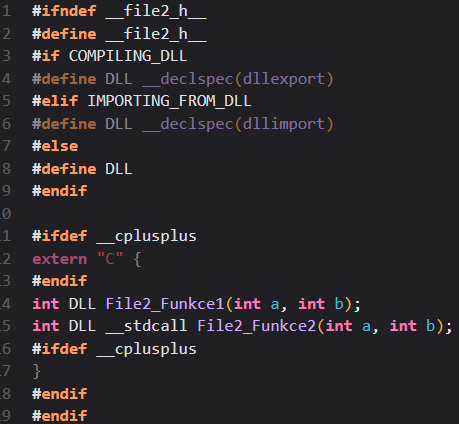
Manual Import

QUESTION 6 EXPLANATION START

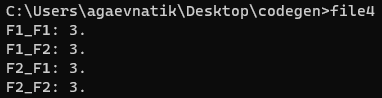
Now let’s modify the file4.cpp so we make a plugin supporting application so that it loads library.dll, imports all 4 symbols, and calls them. For that, we will use the new DLL that we have created because it consists of functions that are exported and will be found in another source file.

Overall the file4.cpp should look like this:  


Meanwhile, let’s change one header file so that it would look the same as another header file.



And when we recompile file4.cpp again and execute it, we get:



Which means – functions have been exported successfully.

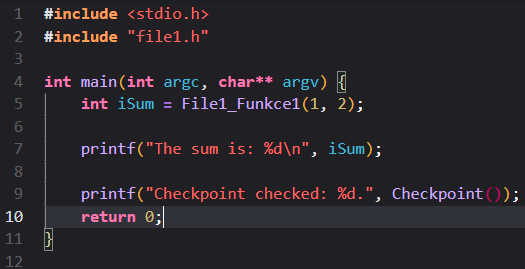
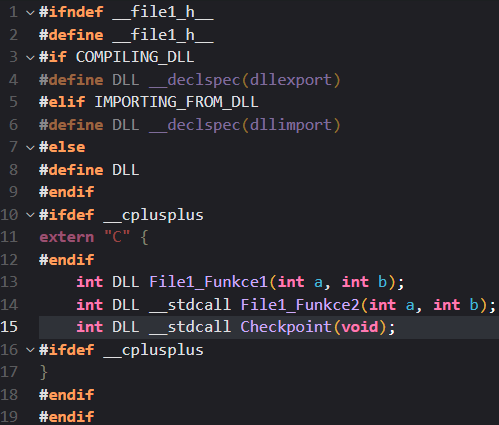
So this is it.

QUESTION 6 EXPLANATION END

Weak link

Let’s try to add new functionality to header file and call it in file3.cpp. Let’s rename library.dll -> library.old.

Add a new function to header files… and call it here…



QUESTION 7 EXPLANATION START

Compile… Link… Compile with the library… Run…

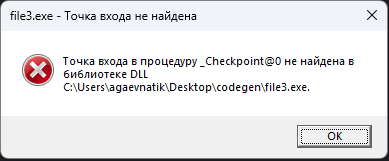
cl /c file1.c file2.cpp /DCOMPILING\_DLL=1 DllMain.cpp

link /DLL file1.obj file2.obj DllMain.obj /OUT:library.dll /IMPLIB:library.lib

cl /o file3.exe file3.cpp library.lib

file3

And the result should be the error message:



This means – the function that we are trying to access – does not exist in the library of dll.

This error message is telling you that a required part of your program is missing and cannot be found. This can happen if the program is looking for a specific version of a file, but that version is not installed or is not compatible. The error message is shown by the operating system, and it can happen at different points in your program, depending on when the missing part is needed. It could be caused by a mistake in how the program was written or if a necessary file was deleted or moved.

In our case – the error occurred at that moment when we called Checkpoint().

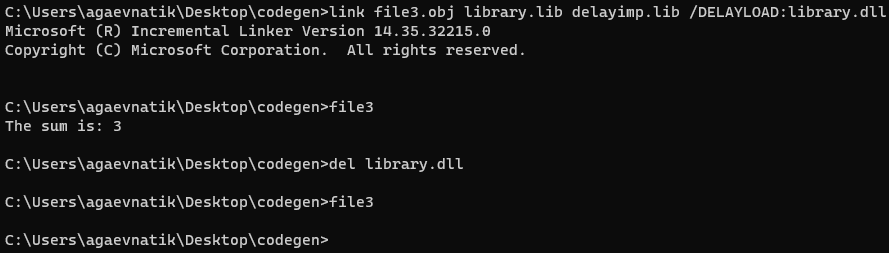
To run our statically-linked program despite the Checkpoint() function missing, it is necessary to use so-called weak linking

Use this command:

Link file3.obj library.lib delayimp.lib /DELAYLOAD:library.dll

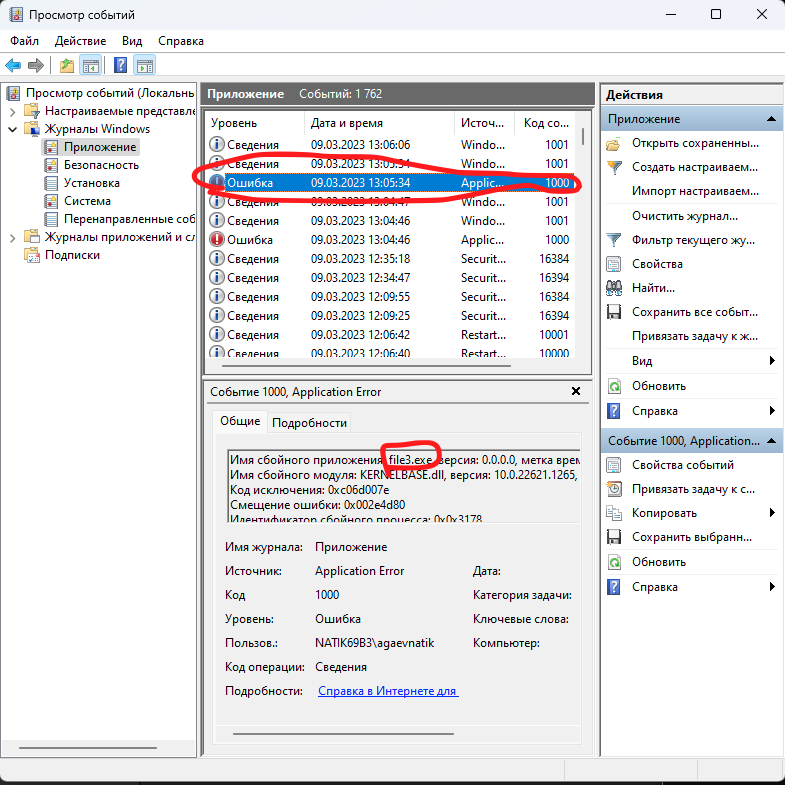
This builds a working app. It will return “The sum is: 3”.

Now remove the library.dll; and run again…. The app will crash certainly.



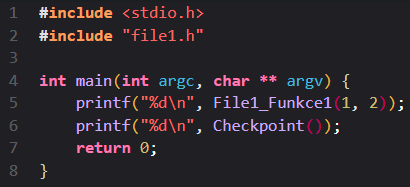
QUESTION 7 EXPLANATION END

If use eventvwr command, you’ll be able to view the moment of app crash.

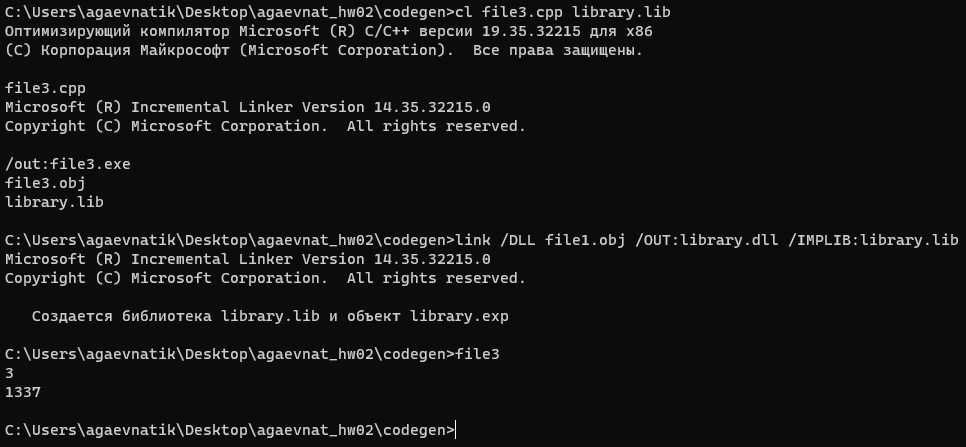


QUESTION 8 EXPLANATION START

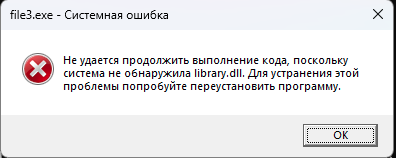
Assume we have this:



Compile, link and run and you get the output.



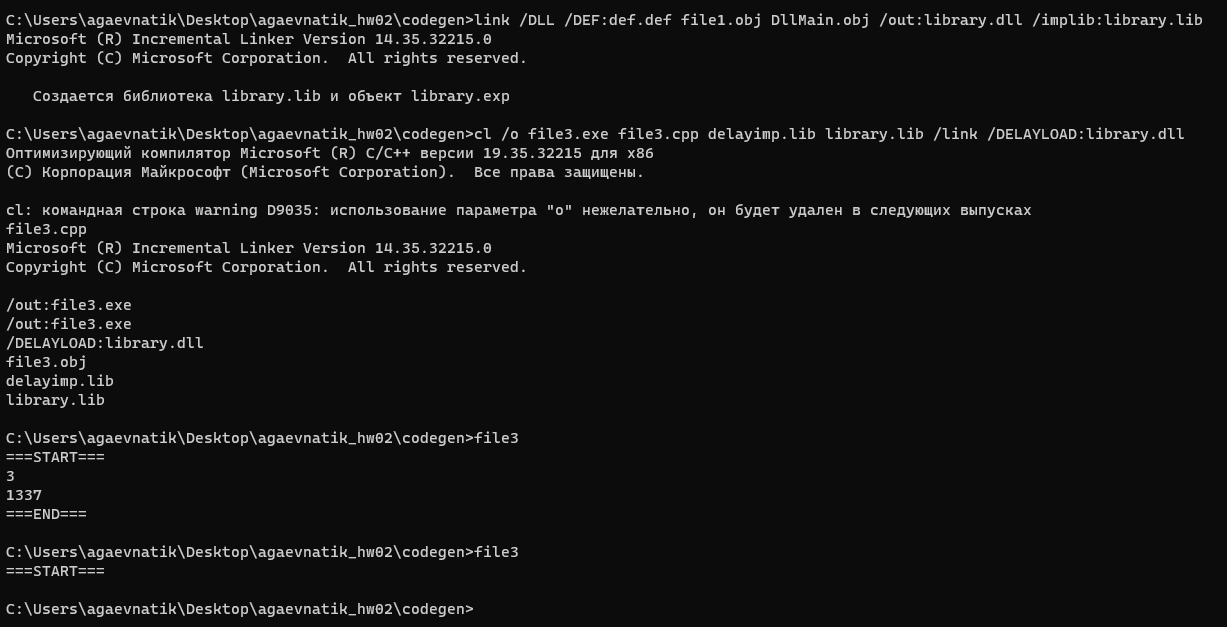
Now let’s remove DLL for some time and execute the file3 again. We would get:



This is a system error, that tells us that it was not possible to continue the code execution, because the system could not find library.dll. The pop-up window suggests trying to reinstall the app.

Now let’s try to make an app NOT crash even if we are trying to access an inaccessible function.

For that we compile source files using this command:



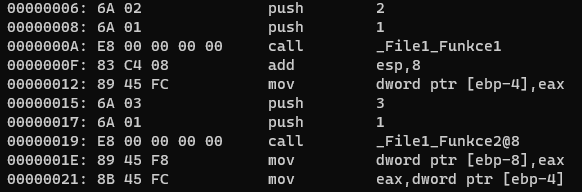
The last command (file3) has been executed after renaming the library.dll (imitation of deletion) and the result -> ===START===. This means – the program has been executed successfully and functions that returns 3 and 1337 have been tried to be called but since the dll does not exist, functions has been ignored. This means – using a weak linking we can try to call a function (that does not exist) and see what is going on at runtime.

CONTINUE! QUESTIONS 8 & 9 TOUCHED!!

**QUESTION 8 EXPLANATION END**

Disassembling the object files

QUESTION 9 EXPLANATION START



There is a difference between the two calls:

The call to File1\_Funkce1 is a standard function call, whereas the call to File1\_Funkce2 has "@8" appended to its name, which suggests that it is a decorated name for a stdcall function. The stdcall calling convention appends the "@" symbol followed by the number of bytes the callee function cleans up from the stack (in this case, 8 bytes).

Explanation of the instructions encountered:

push: Pushes a value onto the stack.

call: Calls a function. The address of the next instruction is pushed onto the stack, and the execution jumps to the specified function address.

add esp,8: Adds 8 to the stack pointer (ESP). This is used to clean up the stack after the function call, as the callee function does not clean up the stack in the cdecl calling convention (used for File1\_Funkce1).

mov dword ptr [ebp-X],eax: Stores the result of the function call (stored in the EAX register) in a local variable located at the base pointer (EBP) minus an offset (X).

QUESTION 9 EXPLANATION END  
QUESTION 10 EXPLANATION START  
The main difference between the \_\_stdcall and \_\_cdecl calling conventions lies in how they handle the stack cleanup after a function call. Both conventions involve passing arguments on the stack, and the return value is stored in the EAX register. However, they differ in the following ways:

Stack cleanup:

\_\_stdcall: The callee function is responsible for cleaning up the stack. This means that the function itself takes care of adjusting the stack pointer (ESP) to remove the arguments from the stack after the function call. The ret instruction in \_\_stdcall functions is followed by the number of bytes to be cleaned up (e.g., ret 8).

\_\_cdecl: The caller function is responsible for cleaning up the stack. After the function call, the caller function needs to adjust the stack pointer (ESP) to remove the arguments from the stack.

Name decoration:

\_\_stdcall: Function names are decorated with the "@" symbol followed by the number of bytes to be cleaned up from the stack (e.g., \_File1\_Funkce2@8).

\_\_cdecl: Function names are not decorated with any additional symbols.

QUESTION 10 EXPLANATION END.

END.