1 První problém

1. otázka

```
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.762 for x86

Copyright (C) Microsoft Corporation.  All rights reserved.

cl : Command line warning D9035 : option 'o' has been deprecated and will be removed in a
future release
file1.c
Generating Code...
Compiling...
file2.cpp
file3.cpp
Generating Code...
Microsoft (R) Incremental Linker Version 8.00.50727.762
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:file1.exe
/out:file3.exe
file1.obj
file2.obj
file3.obj
file3.obj : error LNK2019: unresolved external symbol "int __cdecl File1_Funkce1(int,int)"
(?File1_Funkce1@@YAHHH@Z) referenced in function _main
file3.exe : fatal error LNK1120: 1 unresolved externals
```

Error, který jsme dostali při linkování: `link /out:file3.exe file1.obj file2.obj file3.obj` je způsoben tím, že se `file3.obj` snaží odkazovat na externí symbol `?File1_Funkce1@@YAHHH@Z` pro svoji funkci `_main`, který ovšem nenachází. Když si necháme vypsat symboly z `file1.obj`, zjistíme, že hledaný symbol neexistuje.

```
COFF SYMBOL TABLE
000 010474D8 ABS    notype       Static       | @comp.id
001 80010191 ABS    notype       Static       | @feat.00
002 00000000 SECT1  notype       Static       | .drectve
    Section length   2F, #relocs    0, #linenums    0, checksum        0
004 00000000 SECT2  notype       Static       | .debug$S
    Section length   7C, #relocs    0, #linenums    0, checksum        0
006 00000000 SECT3  notype       Static       | .text$mn
    Section length   1D, #relocs    0, #linenums    0, checksum 189CBBBB
008 00000000 SECT3  notype ()    External     | _File1_Funkce1
009 00000010 SECT3  notype ()    External     | _File1_Funkce2@8
00A 00000000 SECT4  notype       Static       | .chks64
    Section length   20, #relocs    0, #linenums    0, checksum        0
```

Při kompilaci `cl file1.c /c` byl díky typu souboru (.c) spuštěn kompilátor jazyka C. Byla tedy zvolena pojmenovávací konvence symbolů pro jazyk C. Naproti tomu při kompilaci `cl file3.cpp /c` byl kvůli typu souboru (.cpp) spuštěn kompilátor jazyka C++. Ten má ale odlišnou pojmenovávací konvenci symbolů (využívá Name mangling), např. kvůli možnému přetěžování funkcí. `file3.cpp` obsahuje `#include "file1.h"`, ale nástroj `cl` z jeho původní struktury nepozná, že pochází ze zdrojového kódu jazyka C a tedy by při kompilaci `file3.cpp` měl použít konvenci symbolů jazyka C. Dochází tedy ke konfliktu v různém pojmenování ekvivalentních symbolů kvůli použití rozdílných konvencí. Chybu opravíme tím, že přinutíme kompilátor jazyka C++ použít pojmenovávací konvenci jazyka C pomocí makra `__cplusplus` včetně podmíněného překladu a klíčového slova `extern "C"`.

2 Dynamická knihovna

2. otázka

Jak linker z `.obj` souborů poznal, že má funkce exportovat? Pomocí příkazů `dumpbin /directives file1.obj` a `dumpbin /directives file2.obj` si vypíšeme souhrn direktiv příslušných souborů.

```
Microsoft (R) COFF/PE Dumper Version 14.28.29912.0
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file file1.obj

File Type: COFF OBJECT

   Linker Directives
   -----------------
   /DEFAULTLIB:LIBCMT
   /DEFAULTLIB:OLDNAMES
   /EXPORT:_File1_Funkce1
   /EXPORT:_File1_Funkce2@8


Dump of file file2.obj

File Type: COFF OBJECT

   Linker Directives
   -----------------
   /DEFAULTLIB:LIBCMT
   /DEFAULTLIB:OLDNAMES
   /EXPORT:?File2_Funkce1@@YAHHH@Z
   /EXPORT:?File2_Funkce2@@YGHHH@Z

  Summary

          40 .chks64
          F8 .debug$S
          CE .drectve
          3A .text$mn
```

Když kompilátor při kompilaci najde macro`__declspec(dllexport)` u nějaké funkce, emituje na základě této informace linkovací direktivu s příznakem EXPORT, ze které poté linker pozná, které funkce má exportovat. Když zavoláme `link /DLL file1.obj file2.obj DllMain.obj /OUT:knihovna.dll /IMPLIB:knihovna.lib`, exportují se dle předchozího postupu funkce:

```
/EXPORT:_File1_Funkce1
/EXPORT:_File1_Funkce2@8
/EXPORT:?File2_Funkce1@@YAHHH@Z
/EXPORT:?File2_Funkce2@@YGHHH@Z
```

Při linkování tedy již nemusí existovat soubory .cpp, .c a .h, linker je pro tento proces nepotřebuje.

3. otázka

Srovnejte obsah knihoven `knihovna.lib` a `knihovna_static.lib`. Knihovna `knihovna.lib` je importovací a odkazuje na `knihovna.dll`. Statická knihovna je při linkování zahrnuta linkerem přímo do spustileného programu. Když si vypíšeme `dumpbin -all knihovna_static.lib` z výpisu zjistíme, že se nám vypisují pouze informace stejné, jako kdybychom volali pouze `dumpbin -all file1.obj file2.obj`, tedy z objektových souborů, ze kterých byla knihovna složena. Naproti tomu importovací knihovna `knihovna.lib` obsahuje několik symbolů navíc a tabulku se symboly a adresami:

```
2FA     __IMPORT_DESCRIPTOR_knihovna
528     __NULL_IMPORT_DESCRIPTOR
660     knihovna_NULL_THUNK_DATA
8A0     _File1_Funkce1
8A0     __imp__File1_Funkce1
90C     _File1_Funkce2@8
90C     __imp__File1_Funkce2@8
14C     ?File2_Funkce1@@YAHHH@Z
7B4     ?File2_Funkce1@@YAHHH@Z
7B4     __imp_?File2_Funkce1@@YAHHH@Z
14C     ?File2_Funkce2@@YGHHH@Z
82A     ?File2_Funkce2@@YGHHH@Z
430     _File1_Funkce1
430     _File1_Funkce2@8
82A     __imp_?File2_Funkce2@@YGHHH@Z
```

Dále obsahuje odkazy do `knihovna.dll` na adresy s přiřazenými symboly. Tyto symboly, na které se odkazuje do dll lze také vypsat příkazem `dumpbin /exports knihovna.lib`. Statická knihovna `knihovna-static.lib` žádné Export symboly neobsahuje.

```
Exports

   ordinal    name

              ?File2_Funkce1@@YAHHH@Z (int __cdecl File2_Funkce1(int,int))
              ?File2_Funkce2@@YGHHH@Z (int __stdcall File2_Funkce2(int,int))
              _File1_Funkce1
              _File1_Funkce2@8
```

4. otázka

Pro zjištění jaké knihovny DLL bude spustitelný program potřebovat použijeme příkaz `dumpbin /dependents file3.exe`

```
  Image has the following dependencies:

    knihovna.dll
    KERNEL32.dll

  Summary

        2000 .data
        7000 .rdata
        1000 .reloc
       11000 .text
```

a ke zjištění importovaných symbolů vypíšeme `dumpbin /imports file3.exe`.

```
  Section contains the following imports:

    knihovna.dll
                41210C Import Address Table
                418344 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                    2 File1_Funkce1

  ...
```

5. otázka

Kde může být soubor `knihovna.dll` umístěn, aby ho dynamický loader našel při spuštění programu? Když je k dispozici plně definovaná cesta, použije se ta, pokud ne, začíná se hledat. Pořadí hledání DLL závisí na tom, zdali je nebo není povolen `SafeDllSearchMode`. Pokud je povolen, současný adresář uživatele se posouvá dál v pořadí hledání. Ve výchozím nastavení je povolen. Používání DLL pomáhá zejména k modularitě kódu.

SafeDllSearchMode = 1

1. Adresář se spouštěným programem
2. Systémový adresář, který lze zobrazit funkcí `GetSystemDirectory`
3. 16-bit systémový adresář
4. Windows adresář, který lze zobrazit funkcí `GetWindowsDirectory`
5. Současný adresář
6. Adresáře, které jsou v PATH systémové proměnné

SafeDllSearchMode = 0

1. Adresář se spouštěným programem
2. Současný adresář
3. Systémový adresář, který lze zobrazit funkcí `GetSystemDirectory`
4. 16-bit systémový adresář
5. Windows adresář, který lze zobrazit funkcí `GetWindowsDirectory`
6. Adresáře, které jsou v PATH systémové proměnné

Pokud bysme chtěli používat např. knihovnu `knihovna.dll`, která je umístěna v současném adresáři, útočník může umístit školivou knihovnu se setejným jménem do adresáře se spouštěným programem, kde se při jeho spuštění načte škodlivá knihovna, její adresář zde bude mít vyšší prioritu. Podobná hrozba nastala u PuTTY, kde pokud útočník přiměl uživatele stáhnout škodlivé DLL do výchozí složky prohlížeče pro stažené soubory, v případě když by poté byl spuštěn `PuTTY installer .exe` v tomto adresáři, dostane se škodlivý kód do procesu PuTTY před validní knihovnou např. ze systémového adresáře. Takto by vypadal indirect Dll Hijacking. Útočník může také zneužít `SetDefaultDllDirectories` a tím specifikovat, které adresáře se mají prohledávat při procesu načítání. Mohl by pak vložit jeho složku do tohoto procesu a tím by podvrhnul složku plnou knihoven při každém pokusu o načtení knihovny.

3 Manuální import

6. otázka

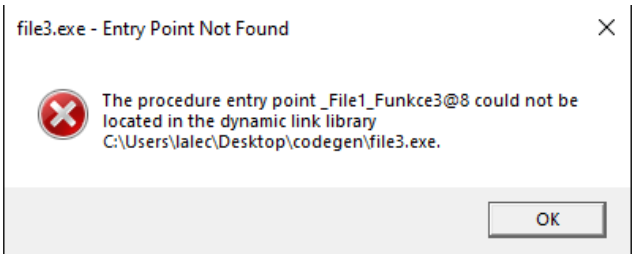
Doplňte soubor file4.cpp tak, aby nahrál ze souboru knihovna.dll všechny 4 symboly a postupně je zavolal. Přiloženo ve zprávě.

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

int main(int argc, char** argv)
{
	HMODULE hModule = NULL;
	int (*pfnFile1_Funkce1)(int, int) = NULL;
	int(__stdcall * pfnFile1_Funkce2)(int, int) = NULL;
	int (*pfnFile2_Funkce1)(int, int) = NULL;
	int(__stdcall * pfnFile2_Funkce2)(int, int) = NULL;

	hModule = LoadLibrary(TEXT("knihovna.dll"));
	if (hModule)
	{
		pfnFile1_Funkce1=(int(*)(int,int))GetProcAddress(hModule,"File1_Funkce1");
		pfnFile1_Funkce2=(int(__stdcall*)(int,int))GetProcAddress(hModule,"_File1_Funkce2@8");
		pfnFile2_Funkce1=(int(*)(int,int))GetProcAddress(hModule,"?File2_Funkce1@@YAHHH@Z");
		pfnFile2_Funkce2=(int(__stdcall*)(int,int))GetProcAddress(hModule,"?File2_Funkce2@@YGH

		if (pfnFile1_Funkce1){
			printf("Soucet: %d.\n", pfnFile1_Funkce1(1, 2));
		}
		else{
			printf("File1_Funkce1: Nenalezena. Chyba %d.\n", GetLastError());
		}

		if (pfnFile1_Funkce2){
			printf("Soucet: %d.\n", pfnFile1_Funkce2(1, 2));
		}
		else{
			printf("File1_Funkce2: Nenalezena. Chyba %d.\n", GetLastError());
		}

		if (pfnFile2_Funkce1){
			printf("Soucet: %d.\n", pfnFile2_Funkce1(1, 2));
		}
		else{
			printf("File2_Funkce1: Nenalezena. Chyba %d.\n", GetLastError());
		}

		if (pfnFile2_Funkce2){
			printf("Soucet: %d.\n", pfnFile2_Funkce2(1, 2));
		}
		else{
			printf("File2_Funkce2: Nenalezena. Chyba %d.\n", GetLastError());
		}

		FreeLibrary(hModule);
		hModule = NULL;
	}

	return 0;
}
```

4 Weak link

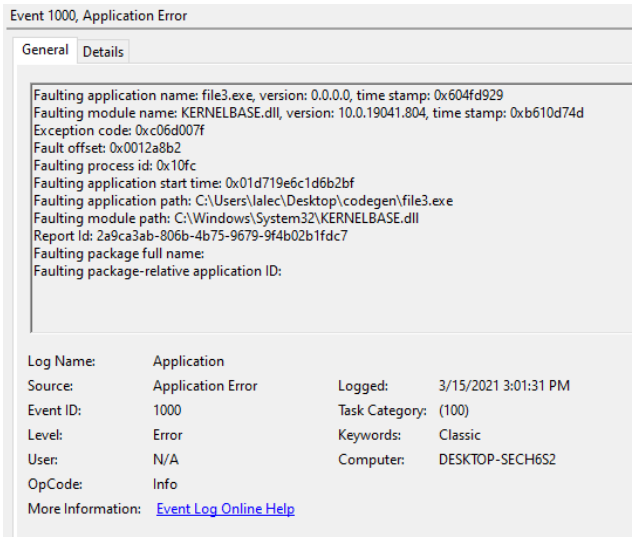
7. otázka

Dostali jsme chybovou hlášku. Program se nespustil, při pokusu o jeho spuštění nebyla nalezena `File1_Funkce3`. `knihovna.lib` má v ten daný moment záznam o symbolu `File1_Funkce3`, který má načítat z knihovny DLL, stará `knihovna.dll` ale tento symbol nemá. Ve fázi, kde se importní knihovna pokouší lokalizovat její známé symboly v `knihovna.dll`, dochází k chybě.



8. otázka

Linkování se dokončí a program bude možné spustit. Program se spustí (narozdíl od předchozí otázky) a spadne s potlačenou chybou (uživateli se nezobrazí chybová hláška), kterou si můžeme zobratzit v `eventvwr.msc`.



Program spadl, protože weak link, který jsme vytvořili, nenalezl `File1_Funkce3` v `knihovna.dll`. Tento problém za běhu programu běžný uživatel nepozná. My to můžeme poznat např. tím, že si necháme vypsat `dumpbin /imports file3.exe` a `dumpbin /exports knihovna.dll`, kde dojdeme k závěru, že `file3.exe` chce používat funkci `File1_Funkce3` z dll, ale `knihovna.dll` takovou funkci nenabízí, tedy dojde k neplatnému čtení z paměti a program padá. Ošetření podobných selhání při používání delay-loaded DLL knihoven má 2 části: obnovení přes hook a oznámení přes výjimku. Můžeme vytvořit hook na pomocnou funkci, která určitým způsobem vyřeší nastalou situaci. Tato rutina by měla být navržena tak, aby její návratová hodnota umožnila pokračovat programu, aby nespadl. Nebo může vracet 0 a tím indikovat vyhození výjimky. Existují notification hooks a failure hooks. Pro oboje může být použita stejná rutina. Failure hooks mohou vypadat např. takto:

```
// This is the failure hook, dliNotify = {dliFailLoadLib|dliFailGetProc}
ExternC
PfnDliHook   __pfnDliFailureHook2;
```

Pokud je notifikace `dliFailLoadLib`, hook může navrátit buď 0, když neumí selhání ošetřit, nebo `HMODULE`, když se problém opravil a knihovna byla načtena. Pokud je `dliFailGetProc` a nelze selhání ošetřit, navrací se opět 0 nebo adresa importované funkce, pokud se podařila načíst. Pro `LoadLibrary` selhání je vyhozena standardní `VcppException(ERROR_SEVERITY_ERROR, ERROR_MOD_NOT_FOUND)`. Pro `GetProcAddress` selhání je vyhozena `VcppException(ERROR_SEVERITY_ERROR, ERROR_PROC_NOT_FOUND)`. (Zdroj)

5 Disassemblování objektových souborů

9. a 10. otázka

Nástroj `dumpbin` lze použít i k disassemblování binárního kódu, a to s přepínačem `/DISASM`.

```
Microsoft (R) COFF/PE Dumper Version 14.28.29912.0
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file file3.obj

File Type: COFF OBJECT

_main:
  00000000: 55                 push        ebp
  00000001: 8B EC              mov         ebp,esp
  00000003: 83 EC 08           sub         esp,8
  00000006: 6A 02              push        2                       <-- push argumentu na zásobník
  00000008: 6A 01              push        1                       <-- push argumentu na zásobník
  0000000A: E8 00 00 00 00     call        _File1_Funkce1          <-- volání File1_Funkce1
  0000000F: 83 C4 08           add         esp,8                   <-- posun esp, uvolnění zásobníku
  00000012: 89 45 FC           mov         dword ptr [ebp-4],eax   <-- uložení výsledku
  00000015: 6A 02              push        2                       <-- push argumentu na zásobník
  00000017: 6A 01              push        1                       <-- push argumentu na zásobník
  00000019: E8 00 00 00 00     call        _File1_Funkce2@8        <-- volání File1_Funkce2
  0000001E: 89 45 F8           mov         dword ptr [ebp-8],eax   <-- uložení výsledku
  00000021: 33 C0              xor         eax,eax
  00000023: 8B E5              mov         esp,ebp
  00000025: 5D                 pop         ebp
  00000026: C3                 ret
```

Můžeme si všimnout, že po volání `File1_Funkce1` probíhá uvolnění zásobníku. `File1_Funkce1` nemá explicitně definovanou volací konvenci, použije se tedy `__cdecl`. `File1_Funkce2` používá konvenci `__stdcall`. V rámci `__cdecl` čistí zásobník volající, oproti tomu u `__stdcall` čistí zásobník volaný (funkce, která se volá). U `__cdecl` RET nic nevrací, u `__stdcall` se vrací počet bytů určených k uvolnění ze zásobníku.

```
_File1_Funkce1:
  00000000: 55                 push        ebp
  00000001: 8B EC              mov         ebp,esp
  00000003: 8B 45 08           mov         eax,dword ptr [ebp+8]
  00000006: 03 45 0C           add         eax,dword ptr [ebp+0Ch]
  00000009: 5D                 pop         ebp
  0000000A: C3                 ret                 <-- RET nic nevrací (__cdecl)
  0000000B: CC                 int         3
  0000000C: CC                 int         3
  0000000D: CC                 int         3
  0000000E: CC                 int         3
  0000000F: CC                 int         3
_File1_Funkce2@8:
  00000010: 55                 push        ebp
  00000011: 8B EC              mov         ebp,esp
  00000013: 8B 45 08           mov         eax,dword ptr [ebp+8]
  00000016: 03 45 0C           add         eax,dword ptr [ebp+0Ch]
  00000019: 5D                 pop         ebp
  0000001A: C2 08 00           ret         8       <-- vrací se počet bytů k vyčištění (__stdcall)
  0000001D: CC                 int         3
  0000001E: CC                 int         3
  0000001F: CC                 int         3
```

6 Opravy a doplnění

V otázce 3 jste správně popsal větší část toho, o co mi šlo, ale jedna věc přesto chybí: Píšete, co má navíc knihovna.lib proti knihovna_static.lib. Existuje i něco, co typicky nalezneme v knihovna_static.lib a nenajdeme to v knihovna.lib?

`knihovna_static.lib` obsahuje všechny objektové soubory sama o sobě, nepotřebuje žádné DLL jako importní `knihovna.lib`.

V otázce 5 píšete o závislosti na stavu nastavení SafeDllSearchMode. Jaký je výchozí stav tohoto nastavení?

Ve výchozím stavu je toto nastavení enabled.

Adresáře, které se prohledávají, jste popsal. Všimněte si však, že jsou i další ohledy, než jen adresáře. Ty jsou přitom dost podstatné, bez nich bychom ani nedokázali zabránit DLL hijackingu. [-1 b.]

Pro lepší ochranu před Dll hijackingem lze využít absolutních cest pro načítání knihoven pomocí LoadLibrary a nejlepé z nějakého bezpečnějšího adresáře, než je současný, např. složku Downloads lze velmi snadno zneužít pro zavedení škodlivé knihovny pouze s právy běžného uživatele. Další bezpečnostní aspekt má register KnownDLLs, který má v sobě list často používaných dynamických knihoven. Pokud aplikace chce využívat nějaké DLL, které je v registeru zapsané, načte se ze System32 adresáře místo použití verze knihovny aplikace. Většina aplikací na Windows dále načítá ty samé systémové knihovny (např. kernel32.dll), proto místo prohledávání adresářů na disku se tyto Dll mohou namapovat do paměti a poté kopírovat do nově vytvořených procesů. Tento mechanismus pomáhá rychlejšímu vytváření nových procesů, ale namapování do paměti a kopírování dále komplikuje snahu načtení škodlivé knihovny.

Nerozumím vašemu popisu indirect DLL hijacking, částečně kvůli překlepu (co je to "vladiní knihovna"?), ale částečně také proto, že jste se vůbec nevyjádřil k otmu, jak se vlastně liší indirect DLL hijacking od normálního DLL hijackingu. [-0,5 b.]

Direct Dll hijacking nastává tehdy, když se útočník snaží zneužít DLL knihovny, která je přímo načítána aplikací. Vývojář aplikace tak může tuto případnou hrozbu snadno ovlivnit. Indirect Dll hijacking naopak využívá zranitelnosti v knihovně DLL, která je načítána aplikací a zároveň knihovna samotná načítá další DLL. Vývojář aplikace zde nemusí mít možnost tuto zranitelnou knihovnu rychle opravit, např. když není jejím tvůrcem apod..