UNIVERZITA PARDUBICE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

DIPLOMOVÁ PRÁCE

2017 Ondřej Kraus

Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Aplikace pro zajištění přenositelnosti libovolné aplikace na platformě Windows

Ondřej Kraus

Diplomová práce

2017

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2016/2017**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Ondřej Kraus**
Osobní číslo:	**I15214**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Aplikace pro zajištění přenositelnosti libovolné aplikace na platformě Windows**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

Cílem diplomové práce je návrh a realizace aplikace, která bude schopna transformovat stávající (zkompilované) aplikace pro platformu Windows na přenositelné.
V teoretické části bude popsána problematika přenositelných aplikací se zaměřením na platformu Windows. Budou zde popsány vlastnosti, které musí splňovat přenositelná aplikace. Dále budou popsány možnosti, jak zajistit, aby existující aplikace (zkompilovaná) byla přenositelná. Vybrané řešení by mělo být schopno zajistit přenositelnost libovolné aplikace v reálném čase. V textu práce bude vybrané řešení podrobně popsáno včetně použitých technik (použité Windows API, function hooking, shellcode, aj.).
V praktické části bude implementováno navržené řešení jako aplikace. Aplikace by měla umožňovat transformovat většinu běžných aplikací na přenositelné. V práci budou popsány podporované vlastnosti aplikace a demonstrováno použití a funkčnost na několika vybraných aplikacích.

Rozsah grafických prací:

Rozsah pracovní zprávy: **50-60 stran**

Forma zpracování diplomové práce: **tištěná**

Seznam odborné literatury:

***PRATA, Stephen. Mistrovství v C++. 4., aktualiz. vyd. Přeložil Boris SOKOL. Brno: Computer Press, 2013. Bestseller (Computer Press). ISBN 978-80-251-3828-1.**
***ERICKSON, Jon. Hacking: umění exploitace. 2., upr. a dopl. vyd. Přeložil Jan POKORNÝ. Brno: Zoner Press, 2009. Encyklopedie Zoner Press. ISBN 978-80-7413-022-9.**

Vedoucí diplomové práce: **Ing. Roman Diviš**
Katedra softwarových technologií

Datum zadání diplomové práce: **31. října 2016**

Termín odevzdání diplomové práce: **17. května 2017**

L.S.

Ing. Zdeněk Němec, Ph.D.
děkan

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2016

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 18. 8. 2017

Ondřej Kraus

Děkuji panu vedoucímu mé diplomové práce panu Ing. Romanu Divišovi za odborné vedení a rady, které mi poskytl v průběhu práce.

ANOTACE

Práce se zaměřuje na využití přesměrování API (Application programming interface) a systémových volání operačních systémů rodiny Microsoft Windows za účelem přenosu dat spolu s aplikacemi tak, aby běh aplikace nenechal na hostitelském systému žádné stopy.

KLÍČOVÁ SLOVA

Windows, přenositelnost, virtualizace aplikací, přesměrování funkcí.

TITLE

Generic application portabiliser for Windows.

ANNOTATION

This thesis is devoted to analysis of API and system call redirection to allow application portability in way that data are transferred with application while leaving no traces of such application on host operating system.

KEYWORDS

Windows, portability, application virtualiztion, function redirection.

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM ZDROJOVÝCH KÓDŮ

SEZNAM ZKRATEK A ZNAČEK

API Application programming interface. 7, 14–17, 24, 25, 27, 29, 33–35, 37, 42, *Terminologie:* aplikační programovací rozhraní

ASLR Address space layout randomization. 36

PE Portable Executable. 14, 24, 27, 39, *Terminologie:* Portable Executable

TERMINOLOGIE

aplikační programovací rozhraní Aplikační programovací rozhraní je rozhraní pro ovládání aplikace z prostředí programovacího jazyka.

cesta ve tvaru 8.3 Cesty kompatibilní s operačním systémem MS-DOS. Jejich délka je omezena na 8 B jména a 3 B koncovky. 17, 40

endianita Endianita určuje pořadí bajtů ve vícebajtových číslech. 14

exportování funkcí Modul může exportovat funkce, které jsou pak přístupné z dalších modulů. 24

handle Identifikátor zdroje v API operačního systému Microsoft Windows. 27, 30

importovací tabulky Datová struktura, která obsahuje všechny importované symboly, jejichž jména v této struktuře linker nahradí skutečnými adresami po načtení zavislostí. 14, 24, 27

importování funkcí Modul může importovat funkce z jiných modulů a používat je stejně, jako kdyby byly jeho součástí. 25, 27

linker modulů Linker projde importovací tabulky souborů PE (Portable Executable) a zapíše skutečné adresy funkcí a symbolů do importovacích tabulek. 14, 24, 25, 27, 37, 39

little endian Architektury používající little endian ukládají bajty od nejméně významného po nejvýznamnější. 20

long mode V architektuře *x86_64* režim s podporou 64 bitových registrů a adres.. 18

PATH Proměnná obsahující seznam cest, ve kterých systém hledá spustitelné soubory a moduly. 27

Portable Executable Formát souborů EXE a DLL v operačních systémech rodiny Microsoft Windows, který specifikuje jak soubor vykonávat a jaké jsou vztahy mezi moduly. 13

přenositelnost V rámci této práce je přenositelností myšlen přenos aplikace i s daty mezi různými instancemi stejného operačního systému, případně v rámci operačních systémů stejné rodiny, pokud není uvedeno jinak. 7

přesměrování funkcí Změna chování funkce. 7, 17, 24, 37, 39, 40, 45

shellcode Kód, který je napsán tak, aby fungoval i po nakopírování na libovolné místo v operační paměti [1]. 35, 37

symbol v modulu Symbol v modulu je název exportované funkce či dat. 14, 24

systémové volání Služba, která je prováděna v režimu jádra operačního systému. 7

virtualizace aplikací Virtualizace zdrojů pro jednu aplikaci naísto pro celý systém. 7, 35

Win32 API API poskytované operačními systémy rodiny Microsoft Windows. 21, 29

ÚVOD

Cílem práce je vytvořit portabilizér aplikací pro operační systémy z rodiny Microsoft Windows. Podporované operační systémy jsou Windows 7 až Windows 10. Portabilizér je softwarový prostředek, který umožňuje přenos aplikace mezi různými instancemi téhož operačního systému, nebo mezi systémy stejné rodiny tak, že s aplikací jsou přenesena všechna data nezbytná pro její fungování i osobní nastavení uživatele této aplikace. Na operačních systémech rodiny Microsoft Windows mohou být předvolby aplikace uložené v souborovém systému a v registrech systému Microsoft Windows.

Mnoho aplikací nedodržuje doporučení pro tvorbu aplikací pro operační systémy rodiny Microsoft Windows a neexistuje tedy univerzální způsob jak vytřídit data konkrétní aplikace. Portabilizér tak musí být plně konfigurovatelný – umožnit výběr adresářů v souborovém systému i klíčů v registrech tak, aby mohl být použit pro většinu aplikací.

K implementaci portabilizéru byla použita metoda přesměrování API. Původní funkce API jsou nahrazeny kódem, který provede přesměrování souborů, adresářů a registrů do struktury, kterou je možné přenášet mezi různými instancemi hostitelského operačního systému. Výsledná struktura může být umístěna v adresáři vedle adresáře s aplikací a portabilizérem.

Portabilizéru k funkci postačují uživatelská oprávnění. Jedním z cílů je možnost snadno aktualizovat hostitelskou aplikaci – bez změn v nastavení portabilizéru, pokud aplikace nezačala používat další adresáře či klíče v registrech.

1 Virtualizace aplikací

Pojem „virtualizace aplikací“ označuje techniku, při které jsou rozhraní poskytovaná operačním systémem nahrazena aplikační virtualizační vrstvou. Virtualizační vrstva zachycuje vybrané operace a provádí činnost, která je po virtualizační vrstvě požadována. Aplikace by neměla mít žádnou možnost poznat, že přistupuje k virtualizovaným prostředkům. Virtualizační vrstva by měla být plně kompatibilní s původní vrstvou, a to i v nedokumentovaných chováních, aby nedošlo k narušení funkce virtualizované aplikace.

Virtualizační vrstva v této práci zachycuje operace se souborovým systémem a registry operačních systémů rodiny Microsoft Windows a přesměrovává všechny soubory i registry do jednoho umístění. Naimplementována je pomocí přesměrovávání funkcí API systému Microsoft Windows.

1.1 Výhody

Virtualizační vrstva, která přesměrovává všechny soubory a registry do jednoho umístění usnadňuje šifrování – poskytuje záruku, že všechna zneužitelná data jsou bezpečně uložitelná do šifrovatelného umístění. V případě ukládání na přenosné médium je pak možné přenášet aplikaci mezi různými systémy s Microsoft Windows a to se zachováním všech potřebných souborů a klíčů v registrech. Díky úplnému oddělení dat aplikace od hostitelského systému je možné provozovat více různých verzí téže aplikace, a to i současně.

Přesměrování cest umožňuje uživatelům s běžnými uživatelskými oprávněními spouštět některé starší aplikace, které kvůli zápisům do systémových adresářů vyžadují vyšší práva. Podobná výhoda platí i pro zápis do registru HKEY_LOCAL_MACHINE.

1.2 Nevýhody

Přesměrování cest vyžaduje zpracování původních cest, hledání vzorů podle konfigurace a vytváření nových cest. U aplikací, které často pracují s různými (tedy používají operace, které vyžadují přesměrování cest) soubory může mít virtualizace souborů negativní vliv na výkon.

Virtualizace některých systémových volání a funkcí API vyžaduje uchovávání informací o stavu, které zvyšuje paměťové nároky a má negativní vliv na výkon.

Implementace slučování adresářů[1] vyžaduje pomocný deskriptor, jelikož je nutné projít jak skutečný adresář, tak virtualizovaný adresář.

Portabilizér nebude podporovat cesty ve tvaru 8.3, z důvodu jejich závislosti na pořadí adresářů a souborů v rámci hierarchie. Implementace by byla možná podle (9.5.4), ale cesty ve tvaru 8.3 jsou používané velice zřídka, a implementace by tak nebyla časově výhodná.

[1] Například pro funkci `ZwQueryDirectoryFile`.

2 Jazyk symbolických adres

2.1 Úvod

Jazyk symbolických adres je jazyk, jehož instrukce odpovídají přímo instrukcím procesoru. Instrukce `jmp` tak může odpovídat kódu `0xE9`[1]. Jazyk symbolických adres tak umožňuje snazší zápis programu než je přímý zápis ve strojovém kódu, ale je se strojovým kódem ekvivalentní. Ukázka složení instrukce je ve výpisu 2.2.

2.2 Registry

Procesorové registry jsou malá datová úložiště, jejichž kapacita je stejná jako velikost slova procesoru. Pomocí instrukcí je možné s registry přímo či nepřímo provádět různé operace – je například možné do nich zapisovat nové hodnoty, nebo je sčítat. Některé registry mají zvláštní význam a přímo ovlivňují vykonávání některých instrukcí – například instrukce `loop` je závislá na hodnotě registru CX, označovaného jako počítadlo[2]. Registry, které začínají na *R* jsou 64 bitové (dostupné pouze v dlouhém režimu), na *E* jsou 32 bitové. Použitelné jsou i registry jako například AX, který je 16 bitový, případně 8 bitové AL a AH, které adresují dolních či horních 8 bitů 16 bitových registrů. Základní registry jsou:

EIP, RIP Instrukční ukazatel. Ukazuje na adresu další instrukce k vykonání [2].

EAX, RAX Akumulátor.

EBX, RBX Báze.

ECX, RCX Počítadlo. Slouží jako počítadlo pro smyčky.

EDX, RDX Rozšíření akumulátoru.

ESP, RSP Adresa vrcholu zásobníku.

EBP, RBP Adresa báze zásobníkového rámce. Ve funkcích usnadňuje přístup k argumentům.

EDI, RDI Adresy pro práci s řetězcovými / skenovacími instrukcemi.

2.3 Syntaxe

Běžně jsou používány dva různé zápisy jazyka symbolický adres[3]: syntaxe od společnosti Intel a syntaxe od AT&T. Ukázka obou syntaxí je v 2.1.

[1] Kód pro *IA-32*.
[2] Písmeno *c* v názvu registru je z Anglického slova *counter* – počítadlo.

```
push   ebx
mov    eax,0x1
mov    ebx,0x2
add    eax,ebx
lea    eax,[eax+4*ebx]
pop    ebx
ret
```

```
push   %ebx
mov    $0x1,%eax
mov    $0x2,%ebx
add    %ebx,%eax
lea    (%eax,%ebx,4),%eax
pop    %ebx
ret
```

Zdrojový kód 2.1: Porovnání sytaxí jazyka symbolický adres

2.3.1 Intel

Syntaxe společnosti Intel je často používána na operačním systému Microsoft Windows. Tuto syntaxi používá většina assemblerů pro platformu *x86*. V této práci je použit assembler *GAS* s Intelovskou syntaxí. Intelovská syntaxe má u operací s více operandy na pozici prvního operandu cíl, na pozici druhého operandu zdroj.

```
operace     operand1,operand2
```

Zdrojový kód 2.2: Skladba instrukce

2.3.2 AT&T

Syntaxe společnosti AT&T je používána na systémech Unixového typu. Nejběžnější assembler *GAS* podporuje jak syntaxi AT&T, tak syntaxi Intel. Syntaxe AT&T má u operací s více operandy na pozici prvního operandu zdroj, na pozici druhého operandu cíl. Instrukce mají přípony, které určují velikost datového typu, se kterým pracují (`movw` pracuje se slovy, `movb` pracuje s bajty). Přípony instrukcí jsou u dnešních assemblerů nepovinné. Operandy předpony, kdy registry mají předponu % a konstanty předponu $ [4].

2.4 Základní instrukce

Instrukce jsou komunikační jednotky pro komunikaci s procesorem. V této sekci je vybráno několik základní, které jsou použité pro cíle této práce.

2.4.1 Přesuny dat

Všechny přesuny dat provádí instrukce `mov`. Umí přenášet data mezi registry, i z a do paměti. Je možné ji použit i k zápisu konstant do registrů i paměti. Pro výpočty adres je používána instrukce pro výpočet efektivní adresy – `lea` – která namísto přesunu dat pouze vypočítá jejich adresu.

2.4.2 Zásobníkové operace

Základní instrukce pro práci se zásobníkem jsou `push` a `pop`. Instrukce `push` vloží svůj operand na zásobník, instrukce `pop` odebere hodnotu ze zásobníku a uloží ji do operandu.

2.4.3 Změny instrukčního ukazatele

Změny instrukčního ukazatele jsou prováděny instrukcemi `jmp`, `call`, `ret` a `retn`. Instrukce `jmp` změn hodnotu instrukčního ukazatele na hodnotu určenou svým operandem. Podle typu operandu je skok buď absolutní, nebo relativní. Instrukce `call` vloží adresu instrukce následující instrukci `call` na zásobník a změní hodnotu instrukčního ukazatele na hodnotu určenou svým operandem. Podle typu operandu je změna buď relativní, nebo absolutní. Instrukce `ret` odebere hodnotu ze zásobníku, na kterou pak nastaví hodnotu instrukčního ukazatele. Jedná se o operaci opačnou k instrukci `call`. Instrukce `retn` funguje stejně, jako instrukce `ret`, ale před odebráním návratové adresy ze zásobníku odebere navíc i počet bajtů daný hodnotou svého operandu.

2.4.4 Práce s řetězci a skenování paměti

Pro tuto práci je důležitá pouze instrukce `repne␣scasw`, která je použita v injektážním kódu pro hledání konců nulou ukončených řetězců. Tato instrukce začne hledání na adrese určené registrem `EDI` či `RDI` a porovnává postupně s registrem `AX`[3]. Pokračuje dokud hodnotu nenajde. S postupem na další hodnotu zvýší hodnotu registru ECX či RCX.

2.5 Kódování instrukcí

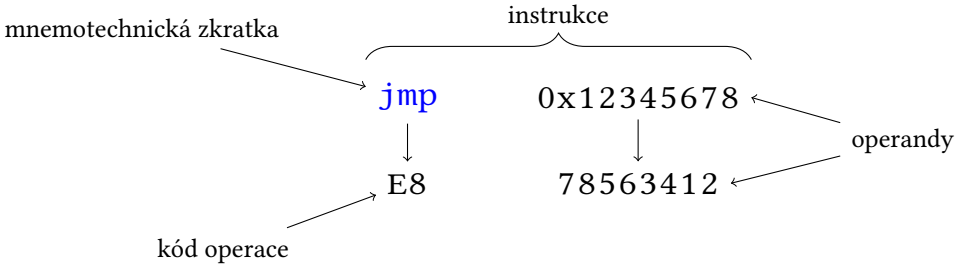
Instrukce jsou složené ze dvou částí: z kódu operace a operandů. Například instrukce nepodmíněného relativního skoku `jmp␣0x12345678` je v šestnáctkové podobě `E9 78563412`, kde `E9` je kód operace a `78563412`[4] je operand. Řetězec `jmp` je mnemotechnická zkratka. Grafické znázornění je v 2.1.

2.6 Funkce a volání

Funkce jsou v programu určeny pouze svou adresou. K zavolání slouží instrukce `call`, která přidá na zásobník adresu pro návrat a změní hodnotu registru `EIP` na hodnotu svého argumentu.

[3] Šíře registru je určena koncvou *w* – pro bajtové porovnání by byla instrukce `repne␣scasb`. Tato práce používá 16 bitové znaky.

[4] Pořadí bajtů je otočené, protože architektura *x86* používá malý endian.



Obrázek 2.1: Skladba instrukce

2.6.1 Volací konvence

Volací konvence určují, v jakém pořadí jsou předávány argumenty a kdo uklízí zásobník. Hlavní rozdělení je pak podle toho, zda zásobník uklízí volající či volaný kód.

Volací konvence *cdecl*

Konvence *cdecl* je běžně používaná kompilátory programovacího jazyka *C*. Zásobník uklízí volající kód a argumenty jsou předávány zprava doleva. Uklízení volající kódem umožňuje předávat jedné funkci různý počet argumentů [2]. Ukázka funkce a volání je ve výpisu 2.3.

Volací konvence *stdcall*

Pro tuto práci je nezbytná volací konvence *stdcall*, kterou používá Win32 API. Jedná se o konvenci, kdy zásobník uklízí volaný kód a argumenty jsou předávány zprava doleva [2]. Ukázka funkce a volání je ve výpisu 2.4.

2.6.2 Volací konvence na platformě *x86_64*

Na platformě *x86_64* existují dvě používané volací konvence: konvence společnosti Microsoft a konvence *System V* [5, 6]. Argumenty jsou předávány zprava doleva.

2.6.3 Další volací konvence

Volací konvence *thiscall*

Volací konvence *thiscall* [7, 8] je používána v jazyce C++ při volání nestatických členských funkcí. Přesné provedení je závislé na implementaci.

```
sum:
    ; uložení původního dna zásobníku
    push    ebp
    ; vytvoření rámce
    mov     ebp, esp
    ; načtení čísla 1
    mov     edx, DWORD PTR [ebp+8]
    ; načtení čísla 2
    mov     eax, DWORD PTR [ebp+12]
    add     edx, eax
    ; načtení čísla 3
    mov     eax, DWORD PTR [ebp+16]
    add     eax, edx
    ; obnovení dna zásobníku
    pop     ebp
    ; návrat
    ret

callsum:
    push    ebp
    mov     ebp, esp
    push    3
    push    2
    push    1
    ; zavolání funkce sum
    call    sum
    ; odebrání argumentů ze zásobníku
    add     esp, 12
    ; obnovení dna zásobníku
    leave
    ret
```

Zdrojový kód 2.3: Ukázka volání funkce s konvencí *cdecl*

```
sum:
    ; uložení původního dna zásobníku
    push    ebp
    ; vytvoření rámce
    mov     ebp, esp
    ; načtení čísla 1
    mov     edx, DWORD PTR [ebp+8]
    ; načtení čísla 2
    mov     eax, DWORD PTR [ebp+12]
    add     edx, eax
    ; načtení čísla 3
    mov     eax, DWORD PTR [ebp+16]
    add     eax, edx
    ; obnovení dna zásobníku
    pop     ebp
    ; odebrání argumentů ze zásobníku + návrat
    ret     12

callsum:
    push    ebp
    mov     ebp, esp
    push    3
    push    2
    push    1
    call    sum ; zavolání funkce sum
    leave       ; obnovení dna zásobníku
    ret
```

Zdrojový kód 2.4: Ukázka volání funkce s konvencí *stdcall*

3 Přesměrování funkcí

Přesměrování funkcí je způsob, jak ovlivnit běh programu. V případě přesměrování funkcí je nahrazena funkce, například nahrazením její úvodní sekvence skokem na náhradní funkci či přepisem její adresy v importovacích tabulkách. Přesměrování netriviálních funkcí má většinou smysl pouze pokud zůstane možnost zavolat původní funkci, i kdyby jen z funkce náhradní.

3.1 Podstrčení modulu

Nejjednodušší způsob jak změnit chování funkcí API je podstrčení modulu. Podstrčený modul exportuje stejné funkce, jako nahrazovaný modul a hostitelský program je pak volá stejně, jako by volal funkce z původního modulu. Nevýhodou této metody je obtížné volání funkcí z původního modulu – je nutné vytvořit kopii původního modulu pod jiným jménem a volat funkce z této kopie. Je nutné přesměrovat všechny symboly, které hostitelský modul používá, jinak dojde při spouštění programu k selhání linkeru. Exportování nenahrazovaných symbolů je možné zautomatizovat. Některé moduly, které jsou součástí distribuce operačního systému jsou přednostně načítány ze systémových umístění. V takovém případě je nutné přepsat název importované knihovny také v hlavičce hostitelského souboru PE. Výhodou metody je nulový dopad na výkon – na volání funkcí náhradního modulu je potřeba stejný počet instrukcí, jako před nahrazením. I volání funkcí z kopie původního modulu je stejně efektivní.

3.2 Přepis v místě volání

Přepis volání je velice jednoduchou metodou, kdy je přepsán cíl instrukce `call`. Tato umožňuje přepsat pouze některá volání a tím poskytuje vyšší granularitu změn chování hostitelského programu. Volání původních funkcí z funkcí náhradních je stejné, jako kdyby k přesměrování nedošlo, protože původní funkce i importovací tabulky zůstavjí beze změny. Nevýhodou je nutnost znát adresy jednotlivých volání. Pro nahrazení všech volání určitých funkcí API je nutné prohledat celý program. V případě nepřímých[1] volání je najití všech volání před spuštěním programu nemožné. Tato metoda je často detekována protipirátskými ochranami jako pokud o prolomení z důvodu změny kódu. Provedení je ve výpisu 3.1.

```
push    1
push    2
call    funkce1
add     esp, 8
```

```
push    1
push    2
call    nahrada1
add     esp, 8
```

Zdrojový kód 3.1: Přepis volání

[1] Například `call eax` pro zavolání funkce, jejíž adresa je v registru EAX.

3.3 Přepis adresy v importovacích tabulkách

Tato metoda spočívá v modifikaci importovacích tabulek hned po skončení práce linkeru. Po skončení běhu linkeru jsou v importovacích tabulkách báze modulů a z nich naimportovaných funkcí. Adresy v tabulce je možné nahradit a případně uchovat pro možnost zavolání původní funkce. Nevýhodou této metody je obtížné zjišťování názvů funkcí v importovacích tabulkách, pokud byl použit ochranný kód, který vkládá mezistupeň [9]. Další nevýhodou je nutnost manipulovat s nedokumentovanými datovými strukturami [10].

3.4 Přepis v cíli volání – trampolíny

Trampolínování [11] je způsob přesměrovaní funkcí, kdy je přepsána volaná funkce, ale výstupem procesu je také funkce, která je svou funkčností totožná s funkcí nahrazovanou. Trampolíny jsou snadné na provedení – nevyžadují znalost interních struktur operačního systému jako je to nutné u přepisu importovacích tabulek a k jejich provedení je třeba pouze alokace a kopírování paměti. Pro obecné trampolínování je třeba dekodér instrukcí pro najití hranic instrukcí, ale v případě přepisování API systému Windows postačuje heuristika. Nevýhodou je nutnost vložit do původní funkce nepodmíněný skok – instrukci `jmp`. Nepodmíněný skok zabírá 5 B a je možné, že v nějaké funkci nebude dost místa. V takových případech je možné použít heuristiku podle konkrétní situace. Pokud by byly dostupné 3 B stačí najít v blízkém okolí volné místo v paměti – například posloupnost instrukcí `nop` – kam bude zapsán 5 B skok, na který se dá dostat krátkým, 3 B skokem. V 64 bitové verzi je navíc nutné zajistit kód trampolíny v dosahu 5 B skoku, protože neexistuje skok s 64 bitovým operandem [12].

3.4.1 Provedení

Při použití trampolíny je úvodní sekvence funkce zkopírována do jiného místa v paměti a za její konec je přidán skok za konec téže sekvence v původním umístění. Původní sekvence je pak nahrazena skokem do náhradní funkce. Náhradní funkce může zavolat původní funkci zavoláním kopie původní sekvence. Ukázka převedení funkce 3.2 pomocí trampolíny je ve výpisu 3.3.

```
funkce1: ; sečte 2 4bytové argumenty
push    ebp
mov     ebp, esp
sub     esp, 12
mov     eax, [ebp + 8]
mov     ebx, [ebp + 12]
add     eax, ebx
mov     esp, ebp
pop     ebp
ret     12
```

Zdrojový kód 3.2: Funkce před přesměrováním

```
funkce1:
jmp     nahrada1
funkce1_original:
mov     eax, [ebp + 8]
mov     ebx, [ebp + 12]
add     eax, ebx
mov     esp, ebp
pop     ebp
ret     12

trampolina:
push    ebp
mov     ebp, esp
sub     esp, 12
jmp     funkce1_original

nahrada1:
push    ebp
mov     ebp, esp
sub     esp, 12
push    1 ; předání falešných argumentů
push    2 ; -||-
call    trampolina
add     esp, 8 ; odebrání argumentů ze zásobníku
mov     esp, ebp
pop     ebp
ret     12
```

Zdrojový kód 3.3: Funkce po přesměrování trampolínou

4 Moduly

Moduly umožňují jsou nosiče kódu, které je možné načíst do běžícího procesu, který po jejich načtení může volat exportované funkce daného modulu. Knihovní moduly sdílí paměťový prostor s hostitelským procesem. Moduly v operačním systému Microsoft Windows mají až na jeden příznak [13] stejný fomát jako spustitelné soubory typu PE. Jejich cílem [14] je umožnit používat některé funkce bez nutnosti začleňovat jejich kód přímo do hlavní aplikace a jejich použití vede také k redukci duplicitního kódu a tím k úspoře paměti a snazším aktualizacím.

4.1 Načtení modulu podle importovacích tabulek

Načtení importovaných modulů provádí linker. Ten projde importovací tabulky procesu a načte požadované moduly. Následně v importovacích tabulkách projde všechny importované funkce a zapíše skoky na správné adresy v cílových modulech.

4.2 Načtení modulu za běhu

Moduly je možné načítat na vyžádání za běhu. K tomuto účelu slouží funkce `LoadLibraryW` nebo funkcí nativního API `LdrLoadDll`. Vysokoúrovňové funkci `LoadLibraryW` stačí předat název modulu, případně celou cestu k modulu, pokud modul není dostupný v proměnné PATH. V případě, že je modul již v hostitelském procesu připojený, funkce pouze zvýší počítadlo použití.

4.2.1 Získání adresy exportované funkce modulu

Po načtení modulu je možné získat adresy exportovaných funkcí pomocí funkce `GetProcAddress` [14] nebo funkcí nativního API `LdrGetProcedureAddress`. Vysokoúrovňové funkci `GetProcAddress` stačí předat handle modulu a název funkce. Funkce v případě úspěchu vrací handle na načtený modul.

4.3 Odpojení modulu načteného za běhu

Modul, který byl načten za běhu je možné i odpojit. K tomu slouží funkce `FreeLibrary` [14] nebo funkce nativního API `LdrUnloadDll`. Vysokoúrovňové funkci stačí předat handle na načtený modul. Modul nemusí být odpojen hned – funkce pouze sníží počítadlo použití a pokud toto počítadlo klesne na nulu, modul je odpojen.

4.4 Hlavní funkce modulu

Každý modul obsahuje funkci `DllMain` [14]. Tato funkce je zavolána při každé z těchto událostí:

- zavedení modulu do procesu,
- odpojení modulu od procesu,
- hostitelský proces vytvořil nové vlákno,
- vlákno hostitelského procesu skončilo.

5 API Windows NT

Využívání služeb operačního systému je možné použitím k tomu určenému rozhraní: API. Poskytované služby a funkce mohou být vykonávány buď v režimu jádra, pak se jedná o systémová volání, nebo v uživatelském režimu.

5.1 Moduly

API operačního systému Microsoft Windows je zpřístupněno prostřednictvím modulů. Moduly jsou rozdělené [15] podle poskytované funkcionality. Pro tuto práci je nejdůležitější modul *ntdll*.

5.1.1 Modul *ntdll*

Modul *ntdll* poskytuje přístup k nativnímu API systému NT. Je to jediný modul, který je zaručeně načtený v každém procesu. Nativní API je rozděleno do několika skupin, které jsou rozlišeny prefixy [16] funkcí. Některé ze skupin jsou

Zw a Nt systémová volání,

Rtl pomocné funkce,

Ldr funkce pro práci se soubory PE.

Tento modul slouží jako základ pro všechny ostatní moduly. Umožňuje práci se soubory i nízkoúrovňový přístup k registrům systému Windows. Protože ostatní moduly nevyhnutelně používají služby modulu *ntdll*, je to modul nejvhodnější pro aplikaci přesměrování funkcí.

5.1.2 Modul *kernel32*

Modul *kernel32* poskytuje většinu funkcí API systému Microsoft Windows. Většina API s viditelnými účinky na systém volá nativní API. Některé funkce nativního API jsou pro jednoduchost zobrazené do několika funkcí základních služeb Win32 API. Například funkce `FindFirstFile` a `FindNextFile` používají pouze funkci `ZwQueryDirectoryFile`.

5.1.3 Modul *user32*

Modul *user32* poskytuje služby spojené s grafickým uživatelským rozhraním. Umožňuje vytvářet okna, ovládací prvky a dialogy.

5.1.4 Modul *advapi32*

Modul *advapi32* poskytuje funkce pro práci s registry systému Windows a službami.

5.2 Důležité funkce

`ZwClose` Uzavírá handle [17].

`ZwCreateFile` Vyváří a otevírá soubor [18].

`ZwOpenFile` Otevírá existující soubor či adresář [18].

`ZwQueryFullAttributesFile` Poskytuje informace o souboru [19].

`ZwQueryAttributesFile` Poskytuje informace o souboru.

`ZwQueryDirectoryFile` Poskytuje informace o souborech v adresáři [20].

`ZwSetInformationFile` Nastavuje informace o souboru [21].

`ZwResumeThread` Umožňuje vláknu pokračovat.

`ZwCreateUserProcess` Vytváří procesy.

`ZwQueryInformationProcess` Poskytuje informaci o procesu [22].

`ZwOpenKey` Otevírá klíč registrů [23].

`ZwCreateKey` Vytváří klíč registrů [24].

6 Možnosti přenášení aplikací

Pokud chce uživatel přenést svou aplikaci i s nastavením a dalšími potřebnými daty na jiný kompatibilní systém, má několik možností, jak cíle dosáhnout. Je důležité kromě samotné aplikace přenést také nastavení a další požadované soubory, aby mohl uživatel aplikaci používat úplně stejně, jako na původním systému. V této kapitole jsou zanalyzovány některé možnosti.

6.1 Přesouvání souborů a kopírování registrů

Kopírování je nejjednodušší možností přenosu aplikací. Je možné jej provést i ručně. Pokud aplikace nevytváří soubory s důvěrnými daty[1], velké množství dat, či dat, u kterých nesmí dojít ke ztrátě, jedná se o vhodné řešení, které vyžaduje pouze základní nástroje poskytované každým běžně používaným operačním systémem.

V případě nesplnění uvedených podmínek hrozí dlouhé časové prodlevy mezi začátkem práce a spuštěním aplikace v případě, že je nutné provést kopírování velkého množství dat. Toto kopírování je pak nutné provést znovu, opačným směrem. V případě selhání systému či například výpadku elektřiny hrozí nechtěné zanechání dat na použitém systému, v některých případech bez možnosti je získat zpět v poslední podobě. Změny v aplikaci mohou vést k zanechávání neočekávaných souborů a adresářů v systému [25].

Rizika je možné omezit nepřepisováním dat, ale kopírováním pod dočasným jménem a až po úspěšném ukončení kopírování původní data nahradit. Toto opatření zvyšuje režii tohoto jednoduchého řešení které pak vyžaduje podporu automatizovaných nástrojů, či alespoň skriptů. Tím se toto řešení stává celkově neefektivním v porovnání s pokročilejšími metodami, protože je nutné pro jedno spuštění aplikace provést celkem 4 kroky: kopírování souborů do systému, kopírování registrů do systému, kopírování souborů ze systému, kopírování registrů ze systému. Celé řešení vyžaduje, aby měl uživatel práva na zápis do všech potřebných adresářů a registrů, což je u netriviálních aplikací výjimečné.

6.1.1 Shrnutí

Řešení je buď snadné, ale nespolehlivé, nebo složitější a spolehlivější. Kopírování může být časově náročné – i webové prohlížeče snadno vygenerují stovky megabajtů dat, jejichž kopírování může trvat desítky sekund. V případě omezených oprávnění může řešení selhávat – uživatel nemusí mít možnost nakopírovat data a registry kam je třeba. V případě, že uživatel nechce riskovat zanechání obnovitelné stopy na cílovém disku, je toto řešení nepoužitelné.

[1]Důvěrnými daty jsou zde myšlena data, která nejsou důvěrná z pohledu zákona, ale pouze data, která za důvěrná považuje uživatel, například i fotografie z dovolené. V případě dat důvěrných ze zákona je nutné dodržovat bezpečnostní politiku, která je u zpracování důvěrných dat nevyhnutelná.

6.2 Symbolické odkazy a kopírování registrů

Oproti řešení v 6.1 je toto řešení spolehlivější a podle výkonu použitého paměťového zařízení i efektivnější. Použití symbolických odkazů vede k odpadnutí nutnosti kopírovat data. Namísto kopie je vytvořen symbolický odkaz, jejichž podpora byla zavedena v operačním systému Windows Vista. Přístupy k datům za symbolickým odkazem pak probíhají transparentně [26], alespoň v případě odkazů na adresář – dle *MSDN* [26] vrací enumerační funkce informace o symbolických odkazech, místo o souborech, na které odkazují. Tato vlastnost způsobuje problémy při práci se soubory, ale ne při přístupu do adresářů.

Některé aplikace v důsledku své činnosti mohou nahradit symbolický odkaz na soubor skutečným souborem. Takový jev způsobuje například dvoufázové ukládání. Nevýhodou je nemožnost vytvářet na systému Windows symbolické odkazy bez běžných uživatelských oprávnění. Z tohoto důvodu doporučuje [27] symbolické odkazy na systému Microsoft Windows vůbec nepoužívat. U této metody zůstává práce s registry stejná, jako v metodě 6.1 se všemi důsledky.

6.2.1 Shrnutí

Nemožnost vytvářet symbolické odkazy bez vyšších oprávnění činí tuto metodu nepoužitelnou pro většinu uživatelů. Menší nevýhodou je pak nekompatibilita s některými aplikacemi. Bez důkladného otestování hrozí náhodné zapsání souboru namísto zápisu do cíle symbolického odkazu – toto riziko je navíc vyšší u aplikací, které používají dvoufázové ukládání, tedy těch, u kterých záleží na správném zápisu dat.

6.3 Podpora v aplikaci

Méně častou možností, jak získat podporu aplikace pro přenositelnost je požádat tvůrce. Tvůrce aplikace ví, jak aplikace funguje a navíc má k dispozici zdrojový kód který mu umožňuje změnit chování aplikace libovolným způsobem. Jednou z možností, jak může tvůrce podporu pro přenositelnost doplnit je parametr příkazového řádku. Parametr může buď určovat cestu k datům, případně může sloužit jako příznak a aplikace v případě jeho použití může ukládat data do vyhrazeného adresáře ve stejné hierarchii jako je aplikace samotná. Další možností je nastavení cesty k datům pomocí proměnné prostředí. Tento způsob používá například databázový systém PostgreSQL [28]. Některé [29] aplikace využívají globální konfigurační soubor v adresáři s aplikací, ve kterém je možné přenosný režim aktivovat. Poté aplikace ukládá data do vyhrazeného adresáře.

Nevýhodou je nutnost počkat na podporu v aplikaci, která nemusí nikdy přijít. Nateré aplikace během práce ve vestavěném přenosném režimu zapisují do uživatelem zvolených umístění pouze data, která tvůrce považuje za důležité, a méně podstatná data nechávají

v systému. Z důvodu podpory ve specifické aplikaci nemá takový přenosný režim vliv na podprocesy, který pak nechávají data přímo v systému.

6.3.1 Shrnutí

Podpora v aplikaci je nejspolehlivější řešení. S výjimkou chyb v aplikaci jako jediné řešení zajišťuje bezproblémovou funkčnost. Nevýhodou je nepříliš rozšířená podpora mezi aplikacemi. V některých případech aplikace ukládá důležitá data kam uživatel požaduje, ale nechává [30] v použitém systému nedůležitá data, jako například pomocné soubory, případně prázdné adresáře. Ponechávání zbytečných stop v systému může být nežádoucí. Řešení funguje pouze pro danou aplikaci, ale nemá vliv na podprocesy.

6.4 Přepis systémových proměnných prostředí

Nepříliš používanou metodou je změna systémových proměnných prostředí. Tato metoda využívá skutečnosti, že některé funkce API vrací cesty na základě hodnot proměnných prostředí [31, 32]. Změnou těchto hodnot v rámci jedné instance příkazového řádku je možné ovlivnit jednotlivé aplikace. Toto řešení je snadné a je možná automatizace dávkovým souborem.

Nevýhodou je nejednotné chování některých aplikací – různé údaje může aplikace získávat různými způsoby což může vést až ke znefunkčnení aplikace, která může část souborů hledat v původních umístěních. Velkou nevýhodou je rozbití dialogu pro výběr souboru v systémech Windows Vista, Windows 7 a Windows 10 pokud není přesně zduplikování struktura uživatelského adresáře. V aplikacích psaných v jazyce *Java* s knihovnou *Swing* či *JavaFX* dochází k rozbití dialogu pro výběr souborů bez známé možnosti nápravy – dialog způsobuje výjimku v nativním kódu.

6.4.1 Shrnutí

Přepis systémových proměnných prostředí funguje bez problémů pouze u jednoduchých aplikací. U aplikací s grafickým uživatelským rozhraním může dojít k omezení funkčnosti čí úplnému znefunkčnění dialogů pro výběr souborů.

6.5 Přesměrování na úrovni API

Toto je nejpokročilejší metoda. Na rozdíl od ostatních je z pohledu aplikace neviditelná a pokud je provedena správně, je také naprosto spolehlivá. Tato metoda nahrazuje funkce poskytované aplikaci operačním systémem a nahrazuje je vlastními, které teprve volají původní funkce operačního systému. V mezikroku je možné s cestami, ke kterým aplikace přistupuje, libovolně manipulovat a nahrazovat tak soubory a adresáře jinými.

Metoda 6.3 spolu s touto jsou jediné, které – s výjimkou chyb v implementaci – zaručují bezpečnost dat. Na rozdíl od metody 6.2 pracuje aplikace s opravdovými soubory a adresáři. Jednotné chování je zaručeno přepisem funkcí, které by jinak mohly vracet různé hodnoty.

Nevýhodou je složitost na provedení. Metoda vyžaduje přímou manipulaci se strojovým kódem, ale tato činnost je zkryta za automatizované nástroje. Další nevýhodou je závislost na konkrétní architektuře a v některých případech interní struktuře dat operačního systému čí chování nedokumentovaných funkcí. Metoda také vyžaduje zvláštní ošetření podprocesů.

6.5.1 Shrnutí

Nejuniverzálnější a nejspolehlivější metoda ze zde zmíněných. Při správném použití může zajistit úplnou izolaci od systému. Ze své podstaty je kompatibilní se všemi běžnými aplikacemi. Nevýhody spojené se závislostí na verzi operačního systému a architektuře procesoru jsou převáženy spolehlivostí a efektivitou a to i za cenu udržování více verzí systému pro přesměrování API.

7 Úvod do praktické části

Cílem je umožnit přenos aplikací na přenosných paměťových médiích. Jedním z možných řešení je přesouvání souborů a registrů z přenosného média do míst, kde je spuštěná aplikace očekává. Také je nutné aktualizovat cesty v konfiguračních souborech. Tato metoda je náchylná na chyby [25] a může způsobit nechtěné zanechání dat v systému, na kterém byla aplikace spuštěna. Aplikace pracující s většími objemy dat mohou vyžadovat i několikaminutové kopírování na cílový systém, a po ukončení zase zpátky. Některé aplikace vyžadují přístup do adresářů, ke kterým běžný uživatel nemusí mít přístup, což vede k nepoužitelnosti aplikace. Aplikace, které používají binární konfigurační soubory jsou s touto metodou nekompatibilní.

Portabilizační řešení z této práce nesmí být postiženo problémy zmíněné v předchozím odstavci a umožní bezpečné přenášení aplikací, při kterém běžnou činností nedojde k zapomenutí dat. Po zhodnocení možností v kapitole 6 byla pro tvorbu portabilizéru vybrána metoda přesměrování API (6.5), která dokáže změnit umístění souborů portabilizované aplikace tak, aby nedošlo k ovlivnění chodu aplikace – celý proces je z pohledu aplikace transparentní. Tomuto procesu se také říká aplikační virtualizace.

Celé řešení je tvořeno dvěma komponentami: loaderem a modulem portabilizéru. Úkolem loaderu je načíst modul portabilizéru do aplikace, a portabilizační modul provede přesměrování API.

Pro vytvoření komponent je použit programovací jazyk C++, z důvodu jeho kompatibility s jazykem C, v němž je popsáno API systému Microsoft Windows. Velkou výhodou použití jazyka C++ je jeho schopnost provádět zásobníkové uvolňování zdrojů které snižuje riziko úniků paměti. Také nevyžaduje žádné běhové prostředí s výjimkou prostředků poskytnutých operačním systémem. Dalším použitým jazykem je jazyk symbolických adres, který umožňuje tvorbu shellcode (podrobněji popsán dále v 9.1) a je použit pro tvorbu injektážního kódu, který je součástí loaderu. Pro kompilaci řešení je použita data nástrojů *GCC*, která obsahuje užitečné nástroje pro převod jazyka symbolických adres do podoby umožňující manipulaci z prostředí jazyka C++.

Loader musí být schopen načíst libovolný modul do libovolné aplikace. Měl by být ovladatelný pomocí parametrů příkazové řádky, aby bylo umožněno využití ze skriptů a případných budoucích podpůrných nástrojů. Také musí být schopen správně předávat parametry spouštěné aplikaci přesně v podobě, v jaké byly předány loaderu aby nedošlo ke znehodnocení parametrů a tím k narušení funkčnosti spouštěné aplikace.

Portabilizační modul by měl podporovat libovolnou běžnou aplikaci pro operační systém Microsoft Windows. Z pohledu uživatele by měl být co nejméně na obtíž. Z tohoto důvodu je jeho jediným projevem chování přepisování cest. Bude konfigurovatelný konfiguračním souborem ve formě prostého textu s jednoduchou strukturou, která by měla umožnit uživatelům snadno přidat cesty, které chtějí přesměrovat.

8 Loader a injektážní kód

Aby bylo možné spouštět cizí kód v paměťovém prostoru nějakého procesu, je nutné kód do jeho paměťového prostoru zapsat. V této práci je použita injektáž modulu. Injektáž libovolného modulu provádí injektážní kód, který je do hostitelského procesu zapsán loaderem.

8.1 Injektážní kód

Injektážní kód se stará o načtení knihovních modulů do hostitelského procesu. Jedná o krátký kód napsaný v jazyce symbolických adres. Není úplně nezávislý, ale vyžaduje přípravu od loaderu.

8.2 Kompilace

Z důvodu spolehlivosti, snadnosti použití pro autora a podpoře [33] standardů je celá soustava sestavovaná sadou *GCC*, dostupnou prostřednictvím balíku nástrojů *MSYS2*.

8.2.1 Kompilace injektážního kódu

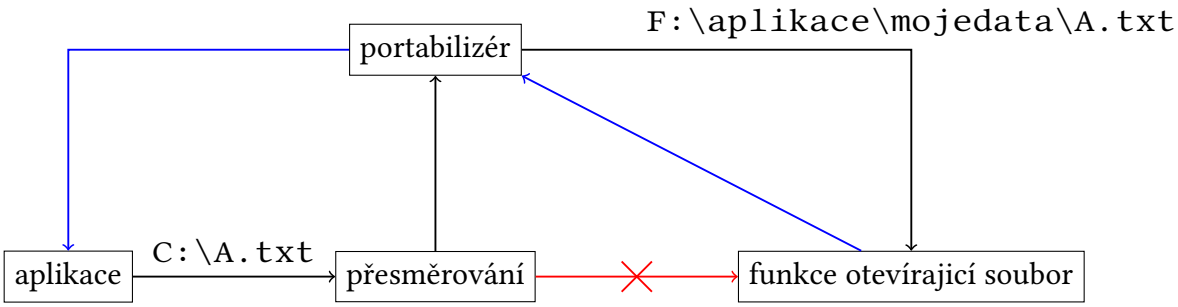
Loader pracuje s injektážním kódem jako s polem znaků. Pole znaků vznikne z injektážního kódu postupným použitím nástrojů *gcc*, *objcopy* a *ld*. Nástroj *gcc* vytvoří objektový soubor ze kterého by šel vytvořit spustitelný soubor. Nástroj *objcopy* vykopíruje pouze vykonavatelný spustitelný kód. Nástroj *ld* vytvoří objektový soubor s definicí pole znaků, které obsahuje původní spustitelný kód. Výsledný objektový soubor je pak připojen k loaderu, který k poli přistupuje jako k externí proměnné.

8.3 Příprava a zápis injektážního kódu

Loader zjistí adresu funkce `LoadLibraryW` v modulu *kernel32*. Její adresa je stejná ve všech procesech, jelikož modul nepodléhá [34] ASLR (Address space layout randomization). Adresu funkce pak loader zapíše na připravené místo v injektážním kódu. Dále loader k injektážnímu kódu připojí cestu k injektovanému modulu. Loader spustí cílový proce. Proces je spuštěn v pozastaveném režimu, aby nedošlo ke kontaminaci hostitelského operačního systému. Po vytvoření procesu injektor zapíše do paměťového prostoru cílového procesu připravený kód, který načte portabilizační modul. Aby proces kód vykonal, změní loader hodnotu registru instrukčního ukazatele na injektážní kód a přidá na zásobník původní hodnotu instrukčního ukazatele. Takto připravenému procesu dovolí loader pokračovat. Portabilizační modul je tak načten hned po *ntdll* a *kernel32*.

9 Portabilizér

Portabilizér svoji funkci provadí přesměrováním některých funkcí a ve vloženém mezikroku změní cesty požadovaným způsobem. Upravené cesty pak předá původní funkci. Výsledkem je přesunutí souborů a registrů. Grafické znázornění je v 9.1.



Obrázek 9.1: Funkce portabilizéru

9.1 Struktura

Portabilizér se skládá ze třech částí: loaderu, injektážního kódu, portabilizačního modulu a konfiguračního souboru. Portabilizér je naimplementovaný jako modul. Aby aplikace načetla modul, který nezná, je nutné ji k tomu přinutit. K tomu slouží loader. Ten spustí aplikaci a zapíše do ní kód, který načte portabilizační modul. Kód, který loader zapisuje musí být nezávislý na pozici v paměti, protože loader nedokáže spolehlivě zařídit, že bude injektážní kód vždy na stejné adrese. Kód také musí být vykonatelný přímo, bez podpůrných prostředků operačního systému, jako je například linker. Z těchto důvodů je kód napsán v jazyce symbolických adres. Informace o modulech, které má načíst jsou pro jednoduchost zapsány přímo za konec kódu. Takovéto kódy, injektované do jiných aplikací jsou nazývány shellcode [1]. Shellcode zavolá z hostitelského procesu prostředky operačního systému pro načítání modulů. Po načtení požadovaných modulů vrátí řízení hostitelskému programu, který může pokračovat v činnosti.

9.2 Implementace portabilizačního modulu

Portabilizér používá přesměrování funkcí z modulu *ntdll* a některých funkcí z modulu *kernel32*. Nahrazení funkcí je provedeno v `DllMain`.

Aby portabilizér fungoval s co největším množstvím aplikací, je nutné přesměrovat API na nejnižší úrovní, tedy na úrovní modulu *ntdll*. Výhodou přesměrování na úrovní modulu *ntdll* je také menší počet přesměrování, než který by byl nutný při přesměrování vysokoúrovňového rozhraní modulu *kernel32*. Přesměrování některých funkcí nativního API je také snazší než přesměrování odpovídacích funkcí z modulu *kernel32*. Například manipulace s argumenty

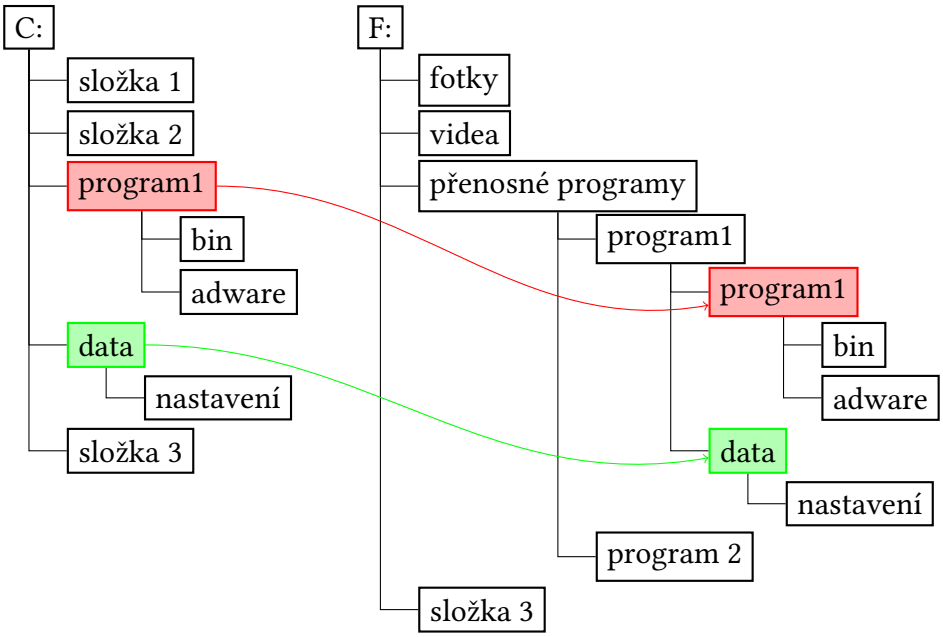
`ZwCreateUserProcess` je snazší než u `CreateProcessW`, jelikož cesta ke spustitelnému souboru i jeho parametry a pracovní adresář jsou již zparsované a rozdělené.

9.3 Struktura virtualizovaných zdrojů

Portabilizér ukládá všechny virtualizované zdroje jako soubory a adresáře v souborovém systému.

9.3.1 Soubory a adresáře

Portabilizér zachovává strukturu souborů a adresářů, pouze celou hierarchii přidává pod nový kořen.



Obrázek 9.2: Zobrazení virtualizovaných zdrojů na skutečné

9.4 Algoritmus přesměrování cest

Cesty v souborovém systému i v registrech jsou přepisovány stejným algoritmem. Algoritmus využívá hledání segmentů cest. V konfiguračním souboru jsou uvedeny očekávané unikatní segmenty, za nimiž jsou data k přesměrování.

9.4.1 Příklad

Pokud aplikace ukládá data do `C:\Users\user\AppData\Roaming\Aplikace1` a do `C:\Users\user\AppData\Local\Aplikace1`, do konfiguračního souboru by

bylo nutné zapsat `\AppData\Roaming\Aplikace1` a `\AppData\Local\Aplikace1`. Použitím středu je obejita nutnost vyplňovat jméno uživatele do konfiguračního souboru. Pokud by z nějakého důvodu nebylo možné uvést pouze částečnou cestu, je lepší problém vyřešit skládáním nástrojů (v tzv. v UNIXovém duchu) a konfigurační soubor generovat z šablony před každým spuštěním portabilizéru. Tento přístup zjednodušuje portabilizační modul, protože není nutné zavádět podporu proměnných a zástupných symbolů v konfiguračním souboru.

9.5 Problémy

Některá systémová volání vyžadují zvláštní zacházení z důvodu jejich složitosti či podstaty jejich funkce. V této části jsou popsána některá tato volání a problémy s nimi spojené. Také je zde popsán problém absolutních cest v konfiguračních souborech.

9.5.1 Cesty v konfiguračních souborech virtualizovaných aplikací

Některé aplikace během svého běhu ukládají absolutní cesty k uživatelským datům do svých konfiguračních souborů. Pokud uživatel uložil svá data přímo do adresáře sousedícího s virtualizovanou aplikací, přičemž nemá tyto cesty v konfiguračním souboru portabilizéru (vedlo by to k nechtěnému přesměrování), po přesunu adresáře s aplikací nebudou absolutní cesty v konfiguračních souborech platné. Z tohoto důvodu si portabilizér ukládá seznam všech cest, ze kterých byl spuštěn a pokud se virtualizovaná aplikace pokusí otevřít soubor či adresář, který by byl potomkem jednoho z dřívějších kořenových adresářů, portabilizér provede přesměrování do nového kořenové adresáře.

9.5.2 Funkce `ZwCreateUserProcess`

Systémové volání `ZwCreateUserProcess` slouží k vytváření procesů a jako základ funkce `CreateProcessW`. Přesměrování volání je nezbytné, aby podprocesy neunikly z portabilizéru. Funkce spouští proces vždy v pozastaveném stavu. Nově vytvořený proces ještě nezpracoval linker a není tak možné provést injektáž portabilizéru. Z toho důvodu portabilizér přepíše adresu vstupního bodu v hlavičce PE, kam se proces dostane po skončení práce linkeru. Do volání je předávána struktura, která obsahuje mimo jiné také cestu k obrazu (spustitelnému souboru) a cestu k pracovnímu adresáři, které portabilizér přepíše. Tento krok je nezbytný, protože není možné použít neexistující obraz a existující není možné spustit v neexistujícím pracovním adresáři.

9.5.3 Stavové automaty se stavem v paměťovém prostoru jádra

Některé funkce, například `ZwQueryDirectoryFile`, používají stavové automaty se stavem v paměťovém prostoru jádra [20]. Po přesměrování musí být stále možné uvolnit paměť stavu v paměťovém prostoru jádra, i paměť pomocných struktur portabilizéru. Přesměrovaná funkce `ZwQueryDirectoryFile` namísto hodnoty HANDLE vrácené původní funkcí `ZwQueryDirectoryFile` vrací vlastní hodnotu HANDLE, která je adresou ukazující na adresu pomocné struktury portabilizéru. Pomocná struktura obsahuje i původní hodnotu HANDLE, která je v náhradní funkci `ZwClose` předána původní funkce `ZwClose`. Náhradní funkce `ZwClose` uvolní i paměť pomocné struktury.

9.5.4 Krátké cesty ve tvaru 8.3

Cesty ve tvaru 8.3 nejsou podporované, přesto by podle autora měla tato práce obsahovat analýzu možného řešení.

Jednou z možností je použít oddělený jmenný prostor, aby nedocházelo ke kolizím mezi nativními krátkými cestami a virtuálními krátkými cestami. Všechny funkce, které vracejí krátké cesty by pak vracely cest z tohoto jmenného prostoru. Nový jmenný prostor může vzniknout úpravou existujícího tvaru krátkých cest, který používají [35] operační systémy rodiny Microsoft Windows: cesta `C:\PROGRA~1\` by v novém jmenném prostoru byla `C:\PROGR~1P\`, kde P[1] je značka jmenného prostoru. Zobrazení z množiny cest z nového jmenného prostoru na množinu dlouhých cest ve virtualizovaném stromu souborů by bylo navždy uloženo v k tomuto účelu vyhrazeném souboru. Tímto způsobem by došlo k oddělení krátkých cest od pořadí v hierarchii, což by umožnilo přenos mezi různými systémy.

9.6 Porovnání s *PortableApps*

Formátu *PortableApps* je nepsaným standardem pro přenosné aplikace. Umožňuje přenášet aplikace na přenosných paměťových médiích, ale často už neumožňuje nic víc, než změnu písmene jednotky [36]. Používá velmi jednoduchou metodu [37], která spočívá v kopírování a přesouvání zdrojů. Při spuštění aplikace jsou uživatelská data přesunuta na hostitelský systém, kde s nimi aplikace pracuje. Po ukončení aplikace jsou data přesunuta zpět do původního umístění.

Portabilizér v této práci nevyžaduje přesouvání ani kopírování zdrojů. Díky přesměrování zdrojů jsou uživatelská data tam, kde je uživatel chce. Nehrozí tak například únik dat z šifrovaného umístění na nešifrovaný disk. V případě výpadku napájení, pádu systému či náhlého odpojení paměťového média nehrozí, že uživatelská data zůstanou kde nemají bez snadné možnosti je vrátit zpět. Odpadá také pomalé kopírování dat tam a zpět, které obzvlášť

[1] Počáteční písmeno slova „portabilizér“.

nepříjemné v případě, kdy jsou uživatelská data velká, ale během jednotlivých užití je potřeba pouze část z nich. Aplikace, které zapisují do umístění, ke kterým je potřeba vyšší oprávnění mohou s portabilizérem z této práce pracovat s oprávněními běžného uživatele, pokud budou přístupy do chráněných adresářů virtualizovány.

10 Ukázka

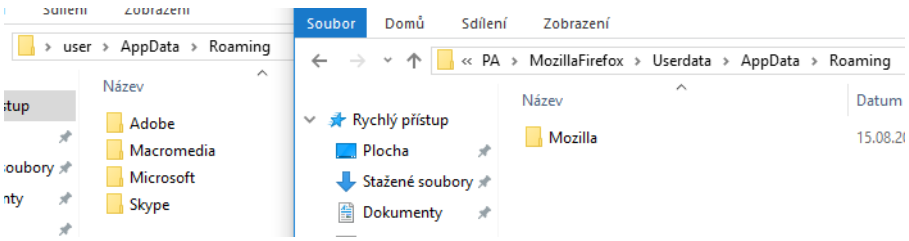
V této kapitole je uvedeno využití portabilizéru na některých aplikacích. Díky přesměrování systémových volání by měl portabilizér fungovat se všemi běžnými aplikacemi pro systém Microsoft Windows. Protože systémová volání nejsou přímo zdokumentovaná, během testování nebyla nalezena žádná aplikace, která by je prováděla přímo pomocí instrukce `syscall`, což je jediný způsob, jak přesměrování funkcí nativního API obejít.

10.1 Mozilla Firefox

Pro prohlížeč Firefox byl použit konfigurační soubor z výpisu 10.1. Ukázka funkčnosti je na obrázku 10.1. Výhodou použití portabilizéru z této práce je bezpečnost: k zapsání dat na disk hostitelského systému vůbec nedochází a je tak možné ukládat data prohlížeče na šifrovaný disk. Díky přesměrování na úrovni systémových volání dojde opravdu ke změně umístění všech souborů, i těch dočasných, které by ale mohly ohrozit soukromí uživatele.

```
\AppData\Roaming\Mozilla
```

Zdrojový kód 10.1: Konfigurační soubor pro Mozilla Firefox



Obrázek 10.1: Ukázka virtulizovaných zdrojů prohlížeče Mozilla Firefox

10.2 NetBeans

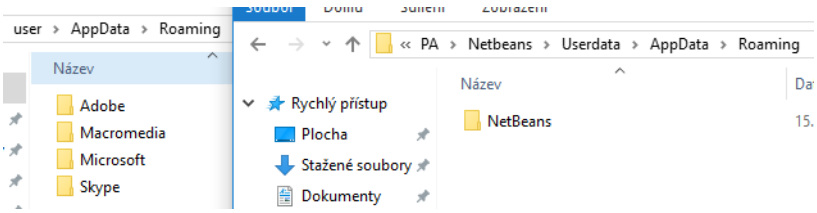
Pro vývojové prostředí NetBeans byl použit konfigurační soubor z výpisu 10.2. Ukázka funkčnosti je na obrázku 10.2. Díky automatické portabilizaci podprocesů dochází i ke správnému přesměrování adresářů využívaných podpůrnými nástroji, jako jsou například *Apache Maven* a *Gradle*. Správně fungují i aplikační servery (autor ověřil funkčnost *Wildfly* a *Payara*) a je tak možné přenášet na přenosném paměťovém médiu kompletní vývojové prostředí pro platformu *Java EE*. Možné využití by mohlo být ve firmách, které by mohly nováčkům distribuovat předpřipravené prostředí, pokud to dovoluje licence použitých nástrojů. Autor práce využíval přenosné prostředí během studií z důvodu chybějících potřebných nástrojů na školních systémech.

Program NetBeans je pro portabilizér důležitý tím, že vytváří rozsáhlé hierarchie procesů. Například spuštění projektu s použitím sestavovacího nástroje *Apache Maven* spustí příkazový řádek, který spustí další příkazový řádek, který spustí *JVM*. Do všech procesů v hierarchii je správně zaveden portabilizační modul a celá soustava se tak chová korektně. Také je zde ověřeno správné přepisování pracovních adresářů a cest ke spustitelným souborům – téměř každý proces spouští své potomky v explicitně nastaveném pracovním adresáři, který je různý od pracovního adresáře vytvářejícího procesu.

Další vlastností NetBeans, která je vhodná pro účely testování portabilizéru je jeho ukládání absolutních cest do konfiguračních souborů. Portabilizér díky 9.5.1 zařídí funkčnost i po přesunutí a přejmenování kořenového adresáře s NetBeans.

```
\AppData\Roaming\NetBeans
\AppData\Local\NetBeans
\NetBeansProjects
\.netbeans-derby
\.m2
\.gradle
```

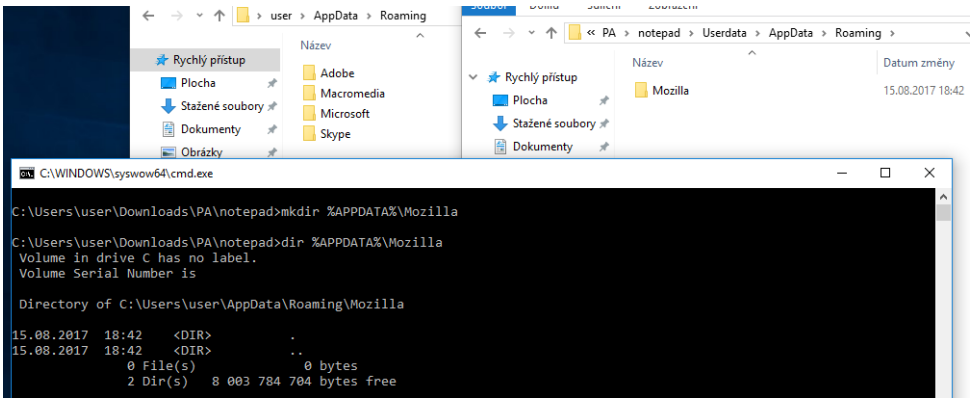
Zdrojový kód 10.2: Konfigurační soubor pro NetBeans



Obrázek 10.2: Ukázka virtulizovaných zdrojů vývojového prostředí NetBeans

10.3 Příkazový řádek systému Windows

Pro ukázku byl použit konfigurační soubor pro prohlížeč Mozilla Firefox (10.1). Ukázka funkčnosti je na obrázku 10.3. Příkazový řádek byl vybrán pro ukázku z důvodu jeho přímého používání nativního API, namísto funkcí z modulu *kernel32*. Je tak vidět, že portabilizér funguje správně i s programy využívající nízkoúrovňové rozhraní.



Obrázek 10.3: Ukázka virtulizovaných zdrojů příkazového řádku systému Windows

11 ZÁVĚR

Podařilo se vytvořit funkční portabilizér pro operační systémy z rodiny Microsoft Windows. Díky portabilizéru je možné přenášet aplikace spolu s daty na externím paměťovém médiu bez nutnosti kopírování na hostitelský systém, což byl cíl práce. Alternativní využití zahrnuje tvorbu archivu s aplikací a portabilizérem, který stačí rozbalit a aplikaci spustit bez nutnosti složité instalace či nutnosti mít vyšší oprávnění. Portabilizér z této práce je efektivnější a spolehlivější než nejrozšířenější existující řešení. Dalším využitím je provozování různých verzí stejné aplikace, a to i současně.

Vedlejším produktem práce je obecně použitelný injektor modulů, který je zároveň snadno ovladatelný pomocí jednoduchých parametrů příkazového řádku. Součástí je také osamostatnitelná knihovna pro přesměrovávání funkcí. Knihovna může být použita k tvorbě specializovaných modulů, které mohou najít uplatnění při testování softwaru či opravování chyb v systémech, ke kterým již nejsou k dispozici zdrojové kódy.

Zlepšení by zahrnovalo takovou úpravu portabilizačního modulu, po které by k portabilizaci stačilo přesměrování funkcí pouze z modulu *ntdll*. Součástí úprav by mohlo být i ruční mapování portabilizačního modulu do paměti hostitelského procesu – portabilizační modul by pak byl pro hostitele neviditelný. Současné řešení je i bez těchto úprav dostačující, jelikož většina procesů používá i modul *kernel32* – v době psaní práce nebyl autorovi znám žádný software, který by nepoužíval modul *kernel32*.

POUŽITÁ LITERATURA

1. *Exploitace #1 - Shellcode - SOOM.cz* [online]. 2007 [cit. 2017-06-23]. Dostupné z: https://www.soom.cz/clanky/400--Exploitace-1-Shellcode.
2. *Steve Friedl's Unixwiz.net Tech Tips*. Intel x86 Function-call Conventions - Assembly View [online] [cit. 2017-07-09]. Dostupné z: http://unixwiz.net/techtips/win32-callconv-asm.html.
3. *Intel and AT&T Syntax* [online] [cit. 2017-07-09]. Dostupné z: http://www.imada.sdu.dk/Courses/DM18/Litteratur/IntelnATT.htm.
4. SHAUGHNESSY, Pat. *Learning to Read x86 Assembly Language - Pat Shaughnessy* [online]. 2016 [cit. 2017-07-08]. Dostupné z: http://patshaughnessy.net/2016/11/26/learning-to-read-x86-assembly-language.
5. *Windows Developer Center: Overview of x64 Calling Conventions* [online]. Redmond: Microsoft Corporation, 2016 [cit. 2017-07-13]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms235286.aspx.
6. *System V ABI AMD64* [online] [cit. 2017-07-23]. Dostupné z: https://c9x.me/compile/doc/abi.html.
7. *Windows Developer Center: __thiscall* [online]. Redmond: Microsoft Corporation, 2017 [cit. 2017-07-13]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/ek8tkfbw.aspx.
8. *Using tha GNU Compiler Collection (GCC): x86 Function Attributes* [online]. Boston: Free Software Foundation, 2017 [cit. 2017-06-10]. Dostupné z: https://gcc.gnu.org/onlinedocs/gcc/x86-Function-Attributes.html.
9. *A Guide to Malware Binary Reconstruction* [online]. 2015 [cit. 2017-07-23]. Dostupné z: https://int0xcc.svbtle.com/a-guide-to-malware-binary-reconstruction.
10. *Перехват вызовов функций из динамических библиотек* [online]. 2008 [cit. 2017-07-27]. Dostupné z: http://www.tdoc.ru/c/delphi-sources/dll/perekhvat-vyzovov-funktsij-iz-dinamitcheskikh-bibliotek.html.
11. *Tadeu Zagallo*. Writing a 'trampoline' in assembly for profiling - Blog by Tadeu Zagallo [online] [cit. 2017-07-22]. Dostupné z: https://tadeuzagallo.com/blog/writing-a-trampoline-in-assembly/.
12. *Trampolines in x64 :: Insanely Low-Level* [online] [cit. 2017-07-23]. Dostupné z: http://www.ragestorm.net/blogs/?p=107.

13. *What is the difference between .dll and .exe ?* [online]. 2007 [cit. 2017-06-11]. Dostupné z: https://social.msdn.microsoft.com/Forums/vstudio/en-US/24c60557-4718-4a9f-bf87-a7d122cc7455/what-is-the-difference-between-dll-and-exe-?forum=csharpgeneral.

14. *Microsoft Support: What is a DLL* [online]. Redmond: Microsoft Corporation, 2017 [cit. 2017-08-08]. Dostupné z: https://support.microsoft.com/en-us/help/815065/what-is-a-dll.

15. *Microsoft Dev Center: Windows API Index* [online]. Redmond: Microsoft Corporation, 2017 [cit. 2017-08-08]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx.

16. *OSR's ntdev List: Meaning of function prefices* [online]. 2005 [cit. 2017-06-10]. Dostupné z: http://www.osronline.com/showThread.cfm?link=72000.

17. *Windows Hardware Dev Center: ZwClose routine (Windows Drivers)* [online]. Redmond: Microsoft Corporation, 2017 [cit. 2017-07-13]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/hardware/ff566417(v=vs.85).aspx.

18. *Windows Hardware Dev Center: ZwCreateFile routine (Windows Drivers)* [online]. Redmond: Microsoft Corporation, 2017 [cit. 2017-07-13]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/hardware/ff566424(v=vs.85).aspx.

19. *Windows Hardware Dev Center: ZwOpenFile routine (Windows Drivers)* [online]. Redmond: Microsoft Corporation, 2017 [cit. 2017-07-13]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/hardware/ff567049(v=vs.85).aspx.

20. *Windows Hardware Dev Center: ZwQueryDirectoryFile routine (Windows Drivers)* [online]. Redmond: Microsoft Corporation, 2017 [cit. 2017-07-13]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/hardware/ff567047(v=vs.85).aspx.

21. *Windows Hardware Dev Center: ZwSetInformationFile routine (Windows Drivers)* [online]. Redmond: Microsoft Corporation, 2017 [cit. 2017-07-13]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/hardware/ff567096(v=vs.85).aspx.

22. *Windows Hardware Dev Center: ZwSetInformationProcess function (Windows)* [online]. Redmond: Microsoft Corporation, 2017 [cit. 2017-07-13]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/desktop/ms687420(v=vs.85).aspx.

23. *Windows Hardware Dev Center: ZwOpenKey routine (Windows Drivers)* [online]. Redmond: Microsoft Corporation, 2017 [cit. 2017-07-13]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/hardware/ff567014(v=vs.85).aspx.

24. *Windows Hardware Dev Center: ZwCloseKey routine (Windows Drivers)* [online]. Redmond: Microsoft Corporation, 2017 [cit. 2017-07-13]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/hardware/ff566425(v=vs.85).aspx.

25. *Thunderbird Portable leaving data on PC?* [online]. 2008 [cit. 2017-06-11]. Dostupné z: https://portableapps.com/node/12743.

26. *Windows Developer Center: Symbolic Link Effects on File Systems Functions* [online]. Redmond: Microsoft Corporation, 2016 [cit. 2017-08-17]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/windows/desktop/aa365682(v=vs.85).aspx.

27. *Don't bother with symlinks in Windows 7* [online]. 2010 [cit. 2017-08-17]. Dostupné z: http://blog.rlucas.net/rants/dont-bother-with-symlinks-in-windows-7/.

28. *PostgreSQL: Documentation: 9.6: postgres* [online]. 2017 [cit. 2017-08-17]. Dostupné z: https://www.postgresql.org/docs/9.6/static/app-postgres.html.

29. *Application directory - Factorio Wiki* [online]. 2017 [cit. 2017-08-17]. Dostupné z: https://wiki.factorio.com/Application_directory.

30. *Portable Mode in paint.net 4.0.17+ | paint.net blog* [online]. 2017 [cit. 2017-08-17]. Dostupné z: https://blog.getpaint.net/2017/07/21/portable-mode-in-paint-net-4-0-17/.

31. *Windows Dev Center: GetUserProfileDirectory function* [online]. Redmond: Microsoft Corporation, 2017 [cit. 2017-08-17]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/desktop/bb762280(v=vs.85).aspx.

32. *Windows Dev Center: SHGetFolderPath function* [online]. Redmond: Microsoft Corporation, 2017 [cit. 2017-08-17]. Dostupné z: https://msdn.microsoft.com/en-us/library/bb762181%28VS.85%29.aspx.

33. *C++ Standards Support in GCC - GNU Project* [online]. Boston: Free Software Foundation, 2017 [cit. 2017-06-10]. Dostupné z: https://gcc.gnu.org/projects/cxx-status.html.

34. *Nynaeve: Adventures in Windows debugging and reverse engineering*. Why are certain DLLs required to be at the same base address system-wide? [online]. 2007 [cit. 2017-06-18]. Dostupné z: http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm.

35. *Windows Dev Center: Naming Files, Paths, and Namespaces (Windows)* [online]. Redmond: Microsoft Corporation, 2017 [cit. 2017-07-13]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/desktop/aa365247(v=vs.85).aspx.

36. *PortableApps.com - Portable Software for USB, portable and cloud drives*. PortableApps.com Format 3.5 (2017-06-01) [online]. 2017 [cit. 2017-07-12]. Dostupné z: https://portableapps.com/development/portableapps.com_format.

37. *PortableApps.com - Portable Software for USB, portable and cloud drives*. Examples - PortableApps.com Launcher 2.2 documentation [online]. 2017 [cit. 2017-07-12]. Dostupné z: https://portableapps.com/manuals/PortableApps.comLauncher/examples/index.html.

38. PRATA, Stephen. *Mistrovství v C++*. 4., aktualiz. vyd. Přel. SOKOL, Boris. Brno: Bestseller (Computer Press), 2013. Bestseller (Computer Press). ISBN 978-80-251-3828-1.

39. ERICKSON, Jon. *Hacking: umění exploitace*. 2., upr. a dopl. vyd. Přel. POKORNÝ, Jan. Brno: Zoner Press, 2009. Encyklopedie Zoner Press. ISBN 978-80-7413-022-9.

40. *Windows Hardware Dev Center: ZwXxx / NtXxx Routines* [online]. Redmond: Microsoft Corporation, 2017 [cit. 2017-07-13]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/hardware/ff567122(v=vs.85).aspx.