FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Portabler: Podpora běhu aplikací pro Windows z přenositelných médií
Student:	Bc. Kamil Kopp
Vedoucí:	Ing. Josef Kokeš, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Počítačová bezpečnost
Katedra:	Katedra informační bezpečnosti
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Smyslem práce je vytvořit nástroj, který umožní spouštět konvenční aplikace, které standardně využívají ryze lokální prostředky (registry, systémový adresář Windows), v přenositelném režimu, například na flashdisku, aniž by byla potřeba jejich instalace nebo získání oprávnění zasahovat do privilegovaných umístění počítače, do kterého je zrovna flashdisk připojen.

1) Nastudujte principy fungování přenositelných aplikací a porovnejte je s aplikacemi tradičními.
2) Seznamte se s reverzně-inženýrskými technikami pro vměšování se do činnosti aplikací: API hooking, debugování, injekce, sandboxing, atd.
3) Navrhněte koncept, jak pomocí těchto technik převést běžnou aplikaci na přenositelnou.
4) Vytvořte prototyp takového konverzního nástroje.
5) Otestujte svůj nástroj s vhodnými programy.
6) Diskutujte své výsledky. Zejména zohledněte omezující podmínky - pro které typy aplikací je váš nástroj vhodný, pro které nikoliv.

Elektronicky schválil/a prof. Ing. Róbert Lórencz, CSc. dne 16. února 2023 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Portabler: Podpora běhu aplikací pro Windows z přenositelných médií

Bc. Kamil Kopp

Katedra informační bezpečnosti
Vedoucí práce: Ing. Josef Kokeš, Ph.D.

4. května 2023

Poděkování

Tímto bych rád poděkoval vedoucímu práce Ing. Josefu Kokešovi, Ph.D. za pomoc, rady i věcné připomínky v průběhu vypracování práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 4. května 2023 . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Kopp, Kamil. *Portabler: Podpora běhu aplikací pro Windows z přenositelných médií.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Abstrakt

V této práci se zabýváme přenositelnými programy. Zjišťujeme, jaké jsou rozdíly mezi přenositelnými a instalovanými programy. Zabýváme se ryze lokálními prostředky Windows registrem a lokálním souborovým systémem. Následně popíšeme metody reverzního inženýrství v softwaru a zjistíme, jaké z těchto metod by mohly mít využití při konverzi instalovaného programu na přenosný. Následně vytvoříme návrh aplikace, která by mohla převádět instalované programy na přenositelné a popíšeme její důležité součásti. Následně implementujeme aplikaci podle návrhu ve Visual C++. Popíšeme detaily implementace včetně simulace registru a přesměrování souborů. Aplikaci následně otestujeme v reálném prostředí. Na závěr provedeme diskuzi nad výsledky testování a dalšími zjištěnými skutečnostmi.

Klíčová slova přenositelnost, přenositelné programy, reverzní inženýrství, Windows, registr, sledování aplikace, injekce kódu, IAT hacking

Abstract

In this work, we are dealing with portable programs. We find out differences between portable and installed programs. We are dealing with purely local resources like Windows registry and local filesystem. After that, we are describing the methods of reverse engineering in software and finding out which of these methods could be useful in converting installed application to portable. Subsequently, we are creating design of application that could convert installed application to portable ones and we are describing its important parts. After that, we are implementing application based on design in Visual C++. We are describing details of implementation including registry simulation and file redirection. We are testing application in real environment. Finally, we are discussing results and other facts found.

Keywords portability, portable programs, reverse engineering, Windows, registry, application tracking, code injection, IAT hacking

Obsah

Seznam obrázků

Seznam zdrojových kódů

Úvod

Většina z nás již někdy na počítač instalovala nějaký program. Uživatel si stáhne instalační soubor, který ho provede nastavením instalace (nejčastěji možnostmi instalace nebo změnou instalační složky) a program se nainstaluje.

Instalované programy na Windows často používají ryze lokální prostředky počítače pro svoji funkčnost a často mají svoje konfigurace na různých místech. To znesnadňuje jejich případný přenos na jiné zařízení, pokud by to uživatel požadoval. I kdyby byl přenos úspěšný, tak aplikace může na cílovém počítači zanechat svoje soubory, které mohou obsahovat i citlivé informace.

Co ale když potřebujeme nějaký konkrétní program mít na více zařízeních najednou? V takovém případě obvykle musíme na každém zařízení program znovu nainstalovat, a tak ho místo jednou máme nainstalovaný vícekrát stejně. Co kdyby ale bylo možné programy přenášet tak, abychom se vyhli opakované instalaci? To nabízejí tzv. přenositelné nebo přenosné programy.

V diplomové práci se budeme zabývat přenositelnými programy, jak se liší od konvenčních programů a možnostmi, jak konvenční programy převádět nebo transformovat na přenositelné. Toho docílíme pomocí metod reverzního inženýrství, se kterými se také seznámíme.

Smyslem práce je vytvořit nástroj, který umožní spouštět konvenční aplikace, které standardně využívají ryze lokální prostředky (registr, systémový adresář Windows) v přenositelném režimu, například na USB flash disku, aniž by byla potřeba jejich instalace nebo získání oprávnění zasahovat do privilegovaných umístění počítače, do kterého je USB flash disk právě připojen.

Cíl práce

Hlavním cílem práce je prozkoumat přenositelné programy, jak fungují a proč se dají přenášet. Na základě toho navrhneme koncept aplikace, která umožní modifikovat programy tak, aby bylo možné je modifikovat na přenosné. Na závěr tuto aplikaci vytvoříme, otestujeme a vyvodíme výsledky.

V rešeršní části se seznámíme s konceptem přenositelných programů a jak se liší od standardně nainstalovaných programů. V další části se seznámíme s reverzním inženýrstvím v softwaru a metodami, které se používají.

Předmětem návrhové části je vytvoření konceptu, kde pomocí získaných znalostí z předchozích kapitol navrhneme aplikaci, která převede nainstalovaný program na přenosný.

Výsledkem implementační části je implementovat prototyp aplikace podle návrhu, ve které jsme aplikovali potřebné metody pro převod programu na přenositelný. Popíšeme zde také důležité části kódu.

Po vytvoření implementace aplikaci otestujeme a vyvodíme výsledky z testování a provedeme závěrečnou diskuzi. Vytvořený program má několik omezení, vyžaduje kontrolu uživatelem a není vhodný na všechny druhy aplikací.

Kapitola 1

Přenositelnost aplikací

V práci se zabýváme přenositelnými aplikacemi z jednoho Windows zařízení na druhé, ne na přenositelnost mezi platformami, jako například z Windows na Linux. V této kapitole popíšeme přenositelné aplikace, co je umožňuje modifikovat na přenositelné a jaké jsou rozdíly mezi přenositelnými a standardními aplikacemi.

1.1 Přenositelnost Windows programů

Přenositelná aplikace je taková aplikace, kterou lze přenést z jednoho zařízení na druhé. Přenosné aplikace jsou většinou open source a bezplatné, nebo s malými příspěvky pro vývojáře. Konkrétní přenosné aplikace jsou poté kompatibilní pro nějaký systém. Přenositelné aplikace se takto dají přenášet z jednoho místa na druhé, například použitím USB flash disku nebo podobného média, ale také třeba prostřednictvím cloudového prostředí. Přenositelná aplikace udržuje všechny soubory aplikace a data dohromady. Z principu si neukládá nic jiného na hostitelské zařízení, a tak běží nezávisle na hostitelském operačním systému [2].

Instalované aplikace vyžadují instalaci, tedy zavedení programu do zařízení. V závislosti na aplikaci, instalace může trvat od několika minut do několik hodin. Aplikaci lze spustit až po tomto počátečním nastavení [1]. Nainstalované aplikace většinou nelze přenášet, protože často používají prostředky mimo svoji instalační složku a při přenosu instalační složky nemusí jít na jiném počítači spustit. Mezi takové typické lokální prostředky patří registr Windows nebo filesystém. Některé aplikace ukládají svoje data i mimo svoji instalační složku.

1.2 Rozdíly mezi přenosnými a nainstalovanými aplikacemi

Pro lepší představu o rozdílech mezi přenositelnými a instalovanými aplikacemi si ukážeme na několika základních vlastnostech, kterými se oba přístupy liší [1].

1.2.1 Umístění souborů programu

Přenositelná aplikace má všechny soubory, které ke svému běhu potřebuje na jednom místě. Nejčastěji jde o instalační složku, ve které je samotný spustitelný soubor. Instalované aplikace ukládají svoje soubory do různých umístění, například do ProgramFiles na disku C, do AppData v uživatelském profilu nebo na jiná umístění [1].

1.2.2 Používání Windows registrů

Přenositelné programy vůbec nemění registr Windows. Instalované aplikace mohou, ale nemusí registry využívat a upravovat. Navíc, když se aplikace odinstalují, ne vždy jsou změny v registru vráceny zpět. To záleží na implementaci aplikace [1].

1.2.3 Změna prostředí a přenos na jiné zařízení

Přenositelné aplikace můžeme přesouvat na jakýkoli disk nebo na jakékoli přenositelné zařízení. Z tohoto důvodu je můžeme přenášet i mezi počítači, například přes USB flash disk a program bude dále fungovat stejně. Přenos instalovaných aplikací není snadný, nejsnažší způsob je aplikaci na cílovém zařízení znovu nainstalovat. Přesunutí souborů jako u přenositelných aplikací často nestačí, aplikace by sice mohla být schopna běhu, ale mohou chybět některé součásti a aplikace by nemusela pracovat správně [1].

1.2.4 Osobní nastavení aplikace

Přenositelné aplikace si typicky nepamatují osobní nastavení, potřebují k tomu navíc konfigurační soubory. Instalované aplikace si může člověk lépe přizpůsobit k obrazu svému tak, jak danému uživateli vyhovuje [1].

1.2.5 Spustitelnost z externích zdrojů

Přenositelné programy lze spustit z externích zdrojů a nebude to ovlivňovat jejich funkčnost. Instalované aplikace také mohou být nainstalovány na přenosné zařízení, ale může být omezena jejich funkčnost nebo nemusí fungovat vůbec. Některé aplikace také vyžadují instalaci na disk s Windows [1].

1.2.6 Vliv na zbytek systému

Protože přenositelné aplikace mají veškeré nastavení na jednom místě, je u nich menší pravděpodobnost, že poškodí některé důležité soubory systému Windows, především proto, že nemodifikují registr. Samozřejmě, pokud se jedná o malware, mohou být škody na systému způsobeny stejně jako u instalované aplikace. U instalovaných programů je toto riziko výrazně vyšší, protože aplikace při instalaci může zasahovat a dělat úpravy jak v registru, tak v některých privilegovaných složkách (například zmíněné Program Files) [1].

1.3 Existující řešení

V této sekci popíšeme existující možnosti a řešení funkčnosti přenositelnosti aplikací v různých prostředích.

1.3.1 Platformy s přenositelnými aplikacemi

V současné době si lze vybrat z několika webových stránek, které nabízejí ke stažení přenositelné verze existujících aplikací. Mezi takové patří například portableapps[1], portablefreeware[2], portapps[3], portable apps od techspot[4] nebo portable software od snapfiles[5]. Všechny weby mají společné to, že nabízejí přenosné verze různých existujících aplikací, ze kterých si uživatel může vybrat a ty si stáhnout.

1.3.2 Sandboxing

Sandboxing je způsob spuštění aplikací v omezeném prostředí. Toto omezené prostředí se nazývá „sandbox“. Sandbox poskytuje bezpečné prostředí, které odděluje uživatele a kritickou část systému od skutečného běhu aplikace, a tak uživatele i systém udržuje v bezpečí beze změny. Sandboxové prostředí je plně pod kontrolou uživatele, které nedovolí aplikaci zasahovat mimo něj [4].

1.3.3 Virtualizace

Virtualizace je proces vytvoření virtuálního prostředí, které dokáže napodobit reálné prostředí. Nejčastěji se virtualizuje operační systém, server nebo síťové zdroje. Využívá software, který dokáže simulovat hardwarové funkce, čímž je dokáže nahradit a hardware už nebude nutně potřeba. To umožňuje například provoz více operačních systémů, více aplikací nebo více serverů na

[1] `https://portableapps.com/`
[2] `https://www.portablefreeware.com/`
[3] `https://portapps.io/`
[4] `https://www.techspot.com/downloads/portable-apps/`
[5] `https://www.snapfiles.com/features/portable-apps.html`

jednom fyzickém počítači. Mezi hlavní výhody virtualizace patří rychlejší nasazení, případné přemisťování, relativně snadná obnova, lepší oddělení různých serverů nebo snadnější management zdrojů spojený s lepší úsporou energie. Virtualizace má i svoje jistá omezení, některé operace nemusí být správně nebo vůbec implementovány, mohou selhat a může dojít k chybě nebo selhání celého systému [3].

1.3.4 Emulace

V oblasti softwaru emulace znamená použití programu nebo zařízení k napodobení chování jiného programu nebo zařízení. Mezi typické příklady emulace patří spouštění OS na hardwaru, pro který původně nebyl navržen, spouštění konzolových her na stolních počítačích nebo spouštění aplikací na systémech, pro které původně nebyly napsány. Typickým příkladem jsou emulátory pro běh Windows aplikací v Linuxovém prostředí [5].

1.4 Registr Windows

Definice 1.4.1 (Windows registry) *The registry is a system-defined database in which applications and system components store and retrieve configuration data. The data stored in the registry varies according to the version of Microsoft Windows. Applications use the registry API to retrieve, modify, or delete registry data [6].*

Definice 1.4.2 (Windows registr) *Registr je systémově definovaná databáze, ve které si aplikace a systémové komponenty ukládají a načítají konfigurační data. Data uložená do registru se liší v závislosti na verzi Microsoft Windows. Aplikace používají API registru pro načtení, změnu nebo odstranění dat registru [6]. (překlad autora)*

1.4.1 Struktura registru Windows

Registr je hierarchická databáze, která obsahuje data, která jsou kriticky důležitá pro fungování systému Windows a aplikací, které ho využívají. Data jsou ukládána ve stromové struktuře. Každý uzel ve stromu se nazývá klíč a může obsahovat hodnoty nebo další podklíče. Jeden klíč může mít v sobě uložen libovolný počet hodnot a hodnoty mohou být v různých formách. Jméno klíče musí obsahovat pouze tisknutelné znaky, musí být dlouhé alespoň 1 znak a nesmí obsahovat znak „\“. Stromová struktura registru může být nejvíce 512 úrovní hluboká [7]. Jméno hodnoty může být dlouhé maximálně 16 383 znaků.

Definované klíče

Než začne aplikace registr používat, musí daný klíč otevřít nebo vytvořit. Pokud chce aplikace otevřít klíč, musí ho otevřít jako podklíč již otevřeného klíče. Aplikace mohou otvírat klíče pomocí volání Windows API `RegOpenKey` nebo vytvářet pomocí `RegCreateKey`. Při vytváření klíčů se může vytvořit pomocí jednoho API volání až 32 úrovní klíče. Klíče se potom uzavírají pomocí funkce `RegCloseKey` [8].

V registru existuje několik základních předdefinovaných klíčů, které jsou vždy otevřené. Jsou to: [9]

- `HKEY_CLASSES_ROOT`, který definuje třídy dokumentů a jejich vlastnosti,
- `HKEY_CURRENT_USER`, který definuje preference uživatele,
- `HKEY_LOCAL_MACHINE`, obsahující informace o fyzickém stavu počítače,
- `HKEY_USERS`, který obsahuje klíče všech uživatelů,
- `HKEY_CURRENT_CONFIG`, který obsahuje informace o aktuálním hardwarovém profilu lokálního počítače.

Datové typy registru

Hodnoty v registru můžeme ukládat v několika formátech, jak číselných, tak řetězcových. Windows definuje následující formáty dat: [10]

- `REG_BINARY`, obsahuje binární data,
- `REG_DWORD`, 32 bitové číslo,
- `REG_DWORD_LITTLE_ENDIAN`, 32 bitové číslo ve formátu little endian,
- `REG_DWORD_BIG_ENDIAN`, 32 bitové číslo ve formátu big endian,
- `REG_EXPAND_SZ`, řetězec zakončený „\0“, který obsahuje nerozbalené odkazy na proměnné prostředí, které se automaticky expandují,
- `REG_LINK`, UNICODE řetězec zakončený „\0“, který obsahuje cílovou cestu symbolického odkazu,
- `REG_MULTI_SZ`, posloupnost řetězců ukončených „\0“, ukončeno sekvencí „\0\0“,
- `REG_NONE`, nedefinovaný datový typ,
- `REG_QWORD`, 64 bitové číslo,
- `REG_QWORD_LITTLE_ENDIAN`, 64 bitové číslo ve formátu little endian,
- `REG_SZ`, řetězec ukončený „\0“.

1.4.2 Hive registru

Hive registru je logická skupina klíčů, podklíčů a jejich hodnot, která má sadu podpůrných souborů načtených do paměti při spuštění systému nebo přihlášení uživatele. Každý uživatel má pro svůj profil vytvořený hive. Ten obsahuje informace například o nastavení aplikací, plochy, prostředí, síťových

připojení nebo tiskáren. Většina hive souborů je umístěna v systémovém adresáři `%SystemRoot%\System32\Config` a aktualizují se při každém přihlášení. Z důvodu zpětné kompatibility mají registry 2 formáty: standardní a novější. Více o hivech, jejich souborech a jak jsou pojmenované lze najít v dokumentaci [11].

1.4.3 Bezpečnost registru

Přístup k registru řídí Windows security model zvaný Access-Control Model [12]. Při volání API funkcí `RegCreateKeyEx` můžeme specifikovat security deskriptor pro daný klíč. Pokud žádný nespecifikujeme, otevře se s defaultním deskriptorem. ACL v defaultním deskriptoru se dědí od rodičovského klíče. Nastavení deskriptoru můžeme provést pomocí volání `RegSetKeySecurity`.

Pro získání informace o deskriptoru slouží volání `RegGetKeySecurity`, alternativně lze využít také `GetNamedSecurityInfo` nebo `GetSecurityInfo`. Při otevírání klíče systém zkontroluje požadovaná práva proti deskriptoru. Pokud by se stalo, že uživatel nemá přístup, operace se nepodaří.

Možnosti úrovně zabezpečení registru a více informací o nich můžeme najít v Microsoft dokumentaci [12].

1.5 Lokální souborový systém Windows

Definice 1.5.1 (Local fileystem) *A file system enables applications to store and retrieve files on storage devices. Files are placed in a hierarchical structure. The file system specifies naming conventions for files and the format for specifying the path to a file in the tree structure [13].*

Definice 1.5.2 (Lokální filesystém) *Souborový systém umožňuje aplikacím ukládat a načítat soubory na úložných zařízeních. Soubory jsou umístěny v hierarchické struktuře. Souborový systém specifikuje jmenné konvence pro soubory a formát pro určení cesty k souboru ve stromové struktuře [13]. (překlad autora)*

Všechny systémy, které Windows podporuje, lze rozdělit na následující součásti úložiště: [13]

- soubor, jako logické seskupení souvisejících dat,
- adresář, jako hierarchická kolekce souborů a jiných adresářů,
- svazek, jako kolekce adresářů a souborů.

1.5.1 Správa souborů

Soubor poskytuje reprezentaci zdroje (buď fyzického zařízení nebo prostředku na fyzickém zařízení), který může být zpracován vstupně-výstupním systémem

(dále I/O systém). Soubory umožňují sdílení dat, mají jméno a podporují synchronizaci. K souborům se dá přistoupit pomocí volání Windows API nebo pomocí aplikací jako Průzkumník nebo Total Commander. Zabezpečení souborů můžeme řídit pomocí tzv. security deskriptorů. Více o správě souborů můžeme najít v Microsoft dokumentaci na rozcestníku [14].

1.5.2 Správa adresářů

Adresář je hierarchická kolekce souborů a jiných adresářů. V závislosti na systému souborů zde může nebo nemusí být omezení na počet souborů, jediné omezení je velikost svazku nebo disku, na kterém je adresář umístěn. Adresář obsahující jeden nebo více dalších adresářů je rodičem každého svého podadresáře a každý podadresář je potomkem svého rodiče. Hierarchická struktura se nazývá adresářový strom. Podobně jako u souborů, k adresářům se dá také přistoupit pomocí Windows API. Více o správě adresářů můžeme najít v Microsoft dokumentaci na rozcestníku [15].

1.5.3 Správa svazků

Svazek je nejvyšší úrovní organizace v souborovém systému. Každý svazek obsahuje alespoň jeden oddíl, tzv. partition, který je logickým rozdělením fyzického disku. Svazek, který má jeden oddíl, se nazývá jednoduchý svazek a svazek, který má více oddílů, se nazývá „multipartition“ svazek. Více o správě svazků můžeme najít v Microsoft dokumentaci na rozcestníku [16].

1.5.4 Správa disků

Pevný disk je fyzický disk uvnitř počítače, který ukládá množství dat a poskytuje k nim co nejrychlejší přístup. Je to nejčastější způsob použití na systémech Windows. Souborový systém poskytuje abstrakci fyzických vlastností zařízení, což umožní aplikacím jednoduše číst a zapisovat do souborů. Více o správě disků můžeme najít v Microsoft dokumentaci na rozcestníku [17].

Kapitola 2

Reverzní inženýrství

V této kapitole si představíme reverzně-inženýrské techniky, které lze použít jak pro zjišťování chování programu, tak pro jeho případnou modifikaci. Také si řekneme něco o reverzním inženýrství obecně a o jeho využití.

2.1 Úvod do reverzního inženýrství

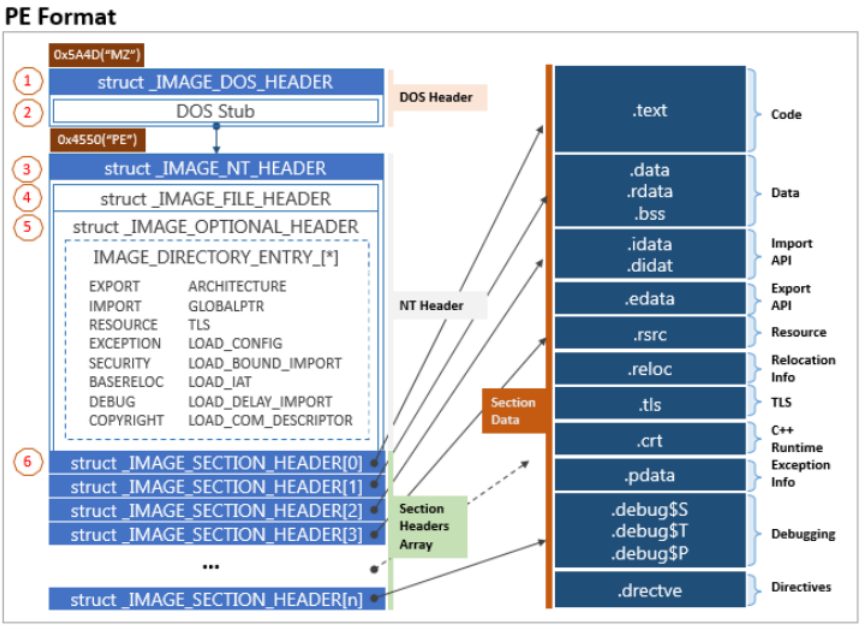
Reverzní inženýrství obecně je akt rozebrání objektu za účelem zjistit, jak uvnitř funguje. Provádí se primárně s cílem analyzovat a získat znalosti o tom, jak objekt funguje. V softwaru jde o zkoumání především binárních nebo spustitelných souborů a zjišťování, jak fungují. Reverzní inženýr dostane spustitelný soubor a snaží se zjistit, jak ve skutečnosti funguje [18].

Reverzní inženýrství u programů je důležité z různých důvodů — provedení bezpečnostní analýzy, získání konkurenční výhody nebo jednoduše k lepšímu pochopení chování programů [18].

Obecně jde o specifický obor, v každém oboru probíhá reverzování trochu jinak. Obecně lze ale popsat 3 kroky, které má reverzní inženýrství společné napříč obory: [18]

- extrakce informací, která spočívá v získání všech možných informací o programu, včetně zdrojového kódu a případné dokumentace, pokud existuje,
- modelování, kdy dochází ke zpracování získaných informací a případné vytvoření nějakého modelu,
- přezkoumání, které zahrnuje testování v různých scénářích pro ověření, zda vytvořený model nebo hypotézy jsou správné.

Existuje několik případů, kdy se reverzní inženýrství používá k opravě nebo rekonstrukci kódu. K tomu může dojít například při ztrátě zdrojového kódu nebo pro přizpůsobení programu pro jinou platformu. Dalším příkladem je oprava chyb, zlepšení výkonu nebo zjištění způsobu, jak program provádí konkrétní operace nebo jaké algoritmy používá.



Obrázek 2.1: Struktura PE souboru. Obrázek převzat z [19].

Další oblastí, ve které se reverzování používá, je oblast malwaru. Analytik se snaží zjistit, jestli je daný spustitelný soubor škodlivý, nebo nikoli. Jako analytici máme k dispozici různé nástroje, které mohou s analýzou pomoci, příklad takových programů a s čím mohou pomoci, můžeme najít níže v sekci 2.5.

2.2 Struktura PE souboru

V této sekci si rozebereme základní strukturu formátu PE. Jde o formát souboru pro spustitelné programy, ale například i dynamicky linkované knihovny v prostředí operačního systému Windows [20]. V hex editoru lze na první pohled poznat PE formát tím, že první dva bajty souboru mají magickou konstantu „MZ“. Struktura PE souboru by se dala rozdělit na dvě hlavní části, jedna obsahuje hlavičky a druhá sekce, jak je možné vidět na obrázku 2.1. Microsoft navíc rozlišuje PE32 pro 32 bitovou architekturu a PE32+ pro 64 bitovou architekturu.

2.2.1 Hlavičky

V PE souboru se objevují tyto druhy hlaviček:

- DOS header, na obrázku 2.1 pod číslem 1,
- NT Header, na obrázku 2.1 pod číslem 3, který obsahuje File Header (na obrázku 2.1 pod číslem 4) a Optional Header (na obrázku 2.1 pod číslem 5),
- Section Header, na obrázku 2.1 pod číslem 6, kterých je v programu více.

Každý PE soubor začíná malým programem pro MS-DOS. Potřeba tohoto kousku kódu je z počátků Windows. Je zde přítomen kvůli zpětné kompatibilitě. Pokud se program spustí na počítači bez Windows, zobrazí se chybová hláška. První bajty PE souboru začínají hlavičkou MS-DOS známou jako `IMAGE_DOS_HEADER`. Hlavička obsahuje dvě důležité hodnoty, a to „e_magic“ a „e_lfanew“. Hodnota „e_magic“ musí být „MZ“, hodnota „e_lfanew“ obsahuje offset další hlavičky [20].

Další hlavičkou je NT Header, která je v programovém prostředí reprezentována strukturou s názvem `IMAGE_NT_HEADER`. Obsahuje tři položky, a to Signature, FileHeader a OptionalHeader. Signatura je pevně daná konstanta ve tvaru „PE\0\0“. [20].

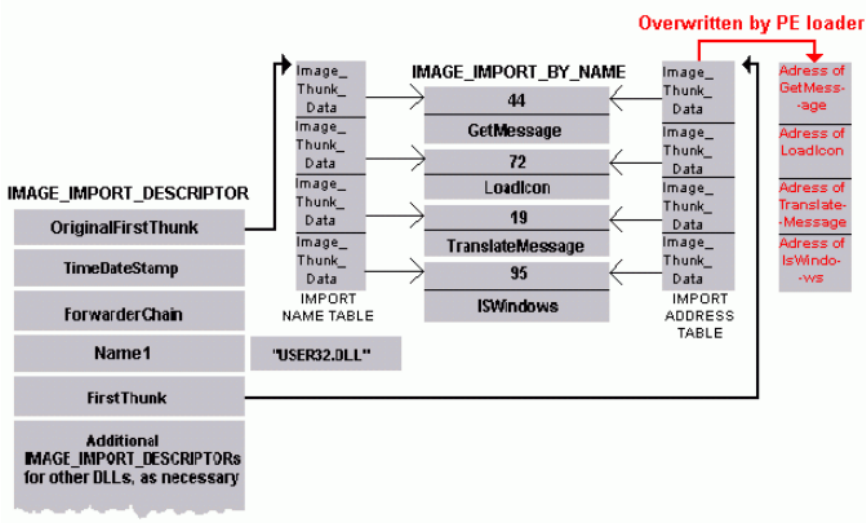
FileHeader je reprezentována strukturou jménem `IMAGE_FILE_HEADER`, kterou lze najít v dokumentaci [21]. Zajímavé jsou například hodnoty „NumberOfSections“, která obsahuje počet hlaviček sekcí, kterých může být maximálně 96, poté „SizeOfOptionalHeader“ nebo „Characteristics“, která indikuje o jaký typ souboru se jedná.

Struktura hlavičky OptionalHeader se liší v závisloti na architektuře, jinak vypadá pro 32 bitovou [6] a jinak pro 64 bitovou [7]. Liší se tím, že ve 32 bitové struktuře je navíc pole „BaseOfData“ odkazující na relativní začátek sekce „.data“. Obsahuje také pole „DataDirectory“, které obsahuje 16 struktur typu `IMAGE_DATA_DIRECTORY`. V práci budeme pracovat s tabulkou Import Address Table (dále IAT), ke které je z této hlavičky přístup. Více o IAT v sekci 2.2.3.

Jednotlivé hlavičky sekcí jsou reprezentovány strukturou pojmenovanou `IMAGE_SECTION_HEADER` [22]. Každá sekce má svoji hlavičku, existuje jich tedy právě tolik, kolik je sekcí. V PE souboru jsou hlavičky sekcí uloženy za sebou. Obsahují informace o sekci, například její adresu nebo velikost. Více informací o sekcích v části 2.2.2.

[6] `https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_optional_header32`

[7] `https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_optional_header64`



Obrázek 2.2: Vztah mezi IAT a importní tabulkou. Obrázek převzat z [25].

2.2.2 Sekce

Rozdělení programu do sekcí poskytuje fyzické oddělení různých částí programu. Pomáhá také při načítání spustitelného souboru do paměti, protože různé sekce usnadňují přístup a načítání stránek [23]. Jak jsme již uvedli, každá sekce má svojí vlastní hlavičku a může jich být maximálně 96. Mezi typické sekce vyskytující se v programu patří:

- .text, obsahující spustitelný kód,
- .data, obsahující inicializovaná data,
- .bss, obsahující neinicializovaná data,
- .rdata, obsahující data pouze pro čtení,
- .rsrc, obsahující zdroje,
- .edata, obsahující informace o exportovaných funkcích,
- .idata, obsahující informace o importovaných funkcích,
- .debug, obsahující informace pro debugování.

2.2.3 Import Address Table

Import Address Table je struktura, která obsahuje informace o jménech importovaných funkcí a jejich adres. Jde o skutečné adresy, na kterých se funkce nacházejí [26]. IAT se nachází v sekci „.idata“ a je defaultně chráněná proti zápisu. Zápis do ní jde ovšem povolit pomocí volání API `VirtualProtect`. To lze například uskutečnit pro změnu adresy volání funkce, tzv. metoda „IAT hooking“, která je podrobněji rozepsaná v sekci 2.4.4. Obrázek 2.2 představuje vztah mezi IAT a importní tabulkou.

2.3 Dynamicky linkované knihovny

Definice 2.3.1 *A dynamic link library (DLL) is a collection of small programs that larger programs can load when needed to complete specific tasks [27].*

Definice 2.3.2 *Dynamicky linkovaná knihovna (DLL) je kolekce malých programů, které mohou větší programy načítat, když jsou potřeba pro dokončení specifického úkolu [27]. (překlad autora)*

Příkladem využití knihovny může být komunikace se zařízeními, například pro tiskárnu nebo skener. Soubory, které podporují specifické operace zařízení, se nazývají ovladače. Podobně jako programy, i DLL mohou obsahovat třídy, proměnné i jiné zdroje [27]. V programu lze DLL načíst například pomocí API funkce `LoadLibrary`. Použití DLL má svoje výhody i nevýhody. Mezi výhody patří: [27]

- Méně chyb vznikajících za běhu, DLL se do paměti nenačítá opakovaně.
- Paměťová efektivita, umožňuje programům běžet rychleji, efektivněji využít paměť a zabírat méně místa na disku, protože DLL se nenačítají společně s hlavním programem. Většina DLL se načte, až když je uživatel potřebuje. Například při tištění wordovského dokumentu z tiskárny se DLL pro práci s tiskárnou načte až při zadání tisku a je načtena pouze jednou.
- Modulární architektura, ta umožňuje dodávat a spravovat program po částech, nemusí být při každé změně přepisována celá aplikace, ale může vyměňovat jen její knihovny nebo komponenty.

Mezi nevýhody použití dynamických knihoven patří: [27]

- Možné chyby při načítání, některé programy vyžadují přítomnost jednoho nebo více konkrétních DLL předtím, než se spustí. V takovém případě například zobrazí chybovou zprávu, že nejde spustit a ukončí se.
- Riziko zneužití, existují útoky jako DLL hijacking nebo DLL injection, které mohou vést ke spuštění cizího kódu, aniž by o tom uživatel věděl. Aplikace tak může načíst útočníkovo DLL, které má sice požadovanou funkčnost, ale dělá i „něco navíc“.
- Rychlost načítání, napojení dynamické knihovny je pomalejší než napojení statické, trvá více cyklů procesoru. Ovšem, operační systém jako takový je obecně rychlejší, protože většinu času využívá méně prostředků.

Jednu z chyb, ke které může dojít při načítání za běhu, označujeme jako „dependency hell“. Jedná se o situaci, kdy aplikace nemůže správně přistoupit k potřebné závislosti. Tato situace může nastat, pokud více aplikaci potřebuje knihovnu se stejným jménem, ale každá s jinou verzí. Může se stát, že různé verze knihovny mají různý obsah a některé aplikace vyžadující jinou verzi knihovny se nemusí správně spustit [28].

2.4 Techniky využívané v RE

V následující sekci popíšeme některé techniky, které využívají reverzní inženýři k získání znalostí o chování programu.

2.4.1 Obfuskace

Obfuskace kódu je proces, kdy se snažíme ztížit analýzu programu tím, že části kódu budou pro analytika obtížně čitelné. Cílem je analytika od zkoumání programu odradit, nebo mu alespoň hodně ztížit práci. Důvodem pro obfuskaci kódu může být například ochrana duševního nebo obchodního vlastnictví části kódu. Obfuskovaný kód ale používá i malware, aby nešlo tak snadno odhalit, jak uvnitř skutečně funguje. Důležité je, aby při vkládání obfuskací nebyla narušena původní funkčnost programu [41].

Jako obfuskace se dá považovat například přidání redundantní logiky, nadbytečného kódu. Takový kód bude pro analytika obtížněji srozumitelný pro nalezení skutečného obsahu a může ho zdržet nebo i odradit. Čtenářem kódu může být jak osoba, která se snaží zjistit chování programu, tak i jiný program, často například jako součást antivirových systémů [41].

Obfuskačních technik je mnoho. Například jiné nebo zavádějící pojmenování metod a proměnných, packování, které zabalí celý program do jakési obálky a znemožňuje jakékoli čtení jeho kódu, vložení nadbytečného kódu, vkládání neprůhledných predikátů, šifrování řetězců, použití antidebuggingových obran, transpozice kódu nebo maskování volání API funkcí [41].

Opačnou metodou je deobfuskace, která se naopak snaží obfuskace z programu odstraňovat a udělat programy lépe čitelnými pro člověka. Automatické deobfuskátory musí často počítat s různými metodami obfuskací najednou. Jednou z metod deobfuskace je tzv. „program slicing“, který se snaží program zúžit na pouze relevantní části kódu. Dalšími metodami jsou například optimalizace kompilátoru nebo metoda zvaná „program synthesis“ [41].

2.4.2 Analýza programů

Analýzu programů můžeme rozdělit do dvou kategorií — statická analýza a dynamická analýza. Obě mají svoje výhody a nevýhody, které rozebereme v následujících podkapitolách.

Statická analýza

Statická analýza programu je proces, kdy se provádí zkoumání kódu bez spuštění programu. Slouží k lepšímu pochopení programu jako celku. Existují automatické nástroje, které s takovou analýzou pomáhají, lze jí dělat ale i ručně [29].

Statickou analýzu může provádět jak vývojář pro vylepšování kódu a opravu chyb, tak reverzní inženýr pro zjišťování, co program dělá. Jelikož ale re-

verzní inženýr má často k dispozici pouze spustitelný soubor, potřebuje nějaký nástroj, který mu ukáže kód, nejčastěji v instrukcích. Pokud není kód výrazně obfuskován, může mu program ukázat i graf toku kódu.

Dynamická analýza

Dynamická analýza kódu je metoda ladění programu, kdy se zkoumá chování programů za jeho běhu. Slouží primárně k diagnostice, k opravě chyb, k řešení problémů s pamětí nebo k předcházení pádu aplikace. K této metodě se nejčastěji používají různé debuggery. Analýza může probíhat jak v „reálném“, tak ve virtuálním prostředí [30].

Na rozdíl od statické analýzy, dynamická analýza umí lépe identifikovat místa, kde dochází například k únikům paměti nebo dereferencí NULL ukazatele, které nemusí být ze statické analýzy zřejmé. Jelikož při dynamické analýze program běží, lze lépe pracovat s problematickými vstupy, požadavky, odpověďmi apod. v reálných situacích. Také pomáhá detekovat neočekávané chyby, které vznikají při interakcích s dalšími aplikačními rozhraními. Ačkoli se to nemusí někdy zdát, dynamická analýza je velmi důležitá nejen při vývoji softwaru [31].

Pro reverzního inženýra je dynamická analýza nesmírně důležitou technikou. Umožňuje mu kontrolovaně spustit a krokovat program, kde může sledovat jeho chování nebo aktuální hodnoty registrů. Jelikož analytici často zkoumají malware, je lepší, když analytik debugguje program v izolovaném prostředí, kdyby došlo k jeho kompromitaci, aby si nekompromitoval reálný systém.

2.4.3 Debugging

„You must think like a machine to understand the machine.“ [35]

Debugování je proces, který zahrnuje identifikaci problému, jeho izolování a následnou opravu. Posledním krokem je otestování opravy nebo náhradního řešení a ujištění, že funguje správně. Debugovat se dá jak software, tak hardware. Při debugování softwaru hledáme a opravujeme chyby v kódu, při vývoji hardwaru opravujeme chybné instalace a konfigurace. Debugger je aplikace, která umožňuje debugování [36].

Debugování lze řídit pomocí bodů přerušení, tzv. breakpointů. Breakpoint je místo, na kterém se debugger zastaví a vývojář si může prohlédnout paměť, vidět proměnné a jejich hodnoty, spouštět program po částech a také změnit hodnotu proměnných nebo obsah paměti [36].

Pro reverzního inženýra je debugování silným nástrojem. Může mu pomoci pro zlepšení pochopení běhu aplikace v určitých místech, k odstranění obfuskací nebo k ověřování hypotéz chování.

Antidebugging

Techniky antidebuggingu jsou metody, jak zjistit, zda program běží pod kontrolou debuggeru. Tyto techniky využívají zejména autoři malwaru, hlavně proto, aby zabránili nebo ztížili práci reverzním inženýrům při zkoumání programu. Techniky mají za cíl jak detekci přítomnosti debuggeru, tak i případnou změnu chování programu při detekci debuggeru. Mezi nejčastější metody antidebuggingu patří volání API metod (například funkce `IsDebuggerPresent`) nebo časování (pokud program stráví v nějakém místě delší než očekávanou dobu, provede akci) [37].

2.4.4 Hooking funkcí

Český překlad „háčkování funkcí“ není moc používaný, častěji se používá anglický pojem „function hooking“ nebo počeštěný výraz „hookování funkcí“, proto budeme používat počeštěný výraz místo českého překladu.

Definice 2.4.1 *The term “hooking a function” is the process of changing the default flow of execution, usually with the intent of either gathering information or changing the result of the hooked function entirely [32].*

Definice 2.4.2 *Termín „hooking funkcí“ je proces změny výchozího toku provedení kódu, obvykle se záměrem sběru informací nebo úplné změny výsledku volání funkce [32]. (překlad autora)*

Pokud chce uživatel hookovat nějakou funkci, potřebuje implementovat novou funkci se stejným rozhraním, která bude původní funkci nahrazovat. Tato nová funkce může mít úplně jinou implementaci nebo může provést nějaký svůj kód a zavolat původní funkci s původními parametry. Ukážeme si dva přístupy, a to hooking pomocí Windows API a hooking pomocí IAT. Oba přístupy jsou si velmi podobné.

Jednou z možností, jak provést hooking funkcí, je pomocí API funkce `GetProcAddress`. Popis ukážeme na příkladu funkce `MessageBoxA`. Nejprve pomocí API `GetProcAddress` nalezneme adresu funkce `MessageBoxA`. Pokud bychom chtěli dělat „unhook“, tedy zpětný proces, můžeme si uložit původní adresu funkce, například pomocí volání `ReadProcessMemory`. Vytvoříme funkci `HookedMessageBoxA` se stejným rozhraním jako původní funkce. Potom pomocí `WriteProcessMemory` zapíšeme adresu `HookedMessageBoxA` na místo, kde byla původně adresa `MessageBoxA`, kterou jsme zjistili dříve pomocí `GetProcAddress`. Pokud teď zavoláme funkci `MessageBoxA`, program si vezme funkci z místa, kde očekává její adresu, avšak tam je adresa naší funkce `HookedMessageBoxA` a zavolá ji. Konkrétní kód v jazyce C++ využívající tuto metodu lze najít zde [33].

Druhá metoda pro změnu adresy funkce využívá IAT. Opět ukážeme na příkladu funkce `MessageBoxA`. Nejprve definujeme funkční prototyp s našimi

argumenty. Poté naimplementujeme `HookedMessageBoxA`, podobně jako jsme jí naimplementovali v předchozí sekci. Následně začneme procházet IAT, dokud v ní nenajdeme `MessageBoxA`. Nalezením funkce v IAT najdeme zároveň i její adresu, kterou změníme na naší `HookedMessageBoxA` a v případě potřeby uložíme starou. Tímto přesměrováním se všechny volání funkce `MessageBoxA` v programu přesměrují na námi implementovanou funkci. Implementaci IAT function hooking v jazyce C++ můžeme najít zde [34].

2.4.5 Injekce kódu

V této sekci se budeme zabývat injekcí kódu do procesu. Nebudeme se zde zabývat útoky typu SQL injection nebo Cross site scripting, které se také snaží injektovat kód.

Definice 2.4.3 (Process injection) *Process injection is a method of executing arbitrary code in the address space of a separate live process [38].*

Definice 2.4.4 (Injekce kódu) *Process injection je metoda spouštění libovolného kódu v adresním prostoru samostatného živého procesu [38].*
(překlad autora)

Pokud se injekce kódu do procesu povede, tak útočník může získat přístup k paměti procesu nebo prostředkům programu. Pokud má injektovaný proces zvýšená práva, potom injektovaný kód bude mít také zvýšená práva. Injektovaný kód se také může vyhnout detekci jiných bezpečnostních programů, zejména antivirů, protože jeho spouštění je „zamaskováno“ pod procesem, do kterého byla injekce provedena. [38]

Kód do procesu lze injektovat více způsoby, některé z nich si představíme blíže. I když je injekce kódu často zneužívána útočníky, kteří mají špatné úmysly, mnohé metody injekce zneužívají legitimní funkce nebo posloupnost legitimních funkcí nebo prostředků [39].

DLL injection

Jedna z nejčastěji používaných technik k injekci kódu je injekce pomocí dynamicky linkované knihovny. Program nejprve zapíše cestu k injektovanému DLL do virtuálního adresního prostoru cílového procesu. Následně zajistí, aby tuto knihovnu vzdálený proces načetl vytvořením vzdáleného vlákna, například voláním `CreateRemoteThread`. Načtením DLL do procesu se spustí funkce DllMain s příznakem `DLL_PROCESS_ATTACH`, kde lze provádět kód [39].

Program potřebuje proces, do kterého se DLL nainjektuje. V případě malwaru jde často o „svchost.exe“. Toho lze docílit například voláním trojice API funkcí `CreateToolhelp32Snapshot`, `Process32First` a `Process32Next` [39].

Pro injekci DLL je třeba zavolat 4 API funkce v následujícím pořadí:

1. `OpenProcess`,
2. `VirtualAllocEx`,
3. `WriteProcessMemory`,
4. `CreateRemoteThread`.

K otevření procesu se použije API funkce `OpenProcess`, která potřebuje PID procesu, do kterého se bude DLL injektovat. Následně se pomocí volání funkce `VirtualAllocEx` v procesu naalokuje místo, do kterého se pomocí další funkce `WriteProcessMemory` uloží cesta k DLL. Poté je třeba použít funkci pro vzdálené spuštění vlákna, například `CreateRemoteThread`. Vlákno se nastartuje na adrese funkce `LoadLibrary` s parametrem, který byl dříve zapsán do procesu pomocí `WriteProcessMemory` [39].

Speciálně funkce `CreateRemoteThread` je často sledována jako potenciální hrozba. Také vyžaduje přítomnost injektovaného DLL, které je možné detekovat [39].

PE injection

V některých ohledech je PE injection podobné DLL injection. Také využívá posloupnost volání API `OpenProcess`, `VirtualAllocEx`, `WriteProcessMemory` a `CreateRemoteThread`. Na rozdíl od ní ale nepotřebuje žádný DLL soubor, protože místo cesty k DLL do procesu rovnou zapisuje svůj kód [39].

Zde se ale objeví problém adres, protože pokud program vloží svůj kód do jiného procesu, bude mít náhodnou adresu, kterou nelze předpovědět. Aby mohl program přepočítat svoje adresy v injektovaném procesu, potřebuje v něm najít relokační tabulky a procházet je, dokud nenajde odpovídající adresu [39].

Process hollowing

Tato metoda se od předešlých liší tím, že místo toho, aby program vytvářel novou sekci a do ní ukládal svůj kód, tak odmapuje z procesu původní legitimní kód a přepíše jeho paměťový prostor svým kódem [39].

Nejprve program vytvoří nový proces v pozastaveném stavu voláním API `OpenProcess` s příznakem `CREATE_SUSPENDED`, proces se tak ihned nespustí. Dalším úkolem programu je odmapovat paměť cílového procesu, to lze udělat například funkcemi `ZwUnmapViewOfSection` nebo `NtUnmapViewOfSection`. Do odmapované paměti potom program zavede svůj kód pomocí `VirtualAllocEx` a `WriteProcessMemory`. Poté zavolá další funkci `SetThreadContext`, pomocí které přesměruje vstupní bod na novou sekci svého kódu, kterou do procesu předtím vložil. Nakonec program zavolá funkci `ResumeThread`, aby obnovil proces z pozastaveného stavu [39].

Thread execution hijacking

Tato metoda je vzdáleně podobná process hollowingu. Program nevytváří žádný nový proces ani vlákno, ale naopak využije již existující vlákno procesu. To se dá provést například pomocí API funkcí `CreateToolhelp32Snapshot`, `Thread32First` a `OpenThread` [39].

Program uspí vlákno pomocí `SuspendThread` a provede injekci pomocí `VirtualAllocEx` a `WriteProcessMemory`. Injektován může být jak kód, tak i třeba cesta k DLL spolu s adresou pro volání `LoadLibrary`. Poté zavolá funkce `SetThreadContext` a `ResumeThread` a dojde k obnovení činnosti vlákna [39].

Problémová může být část, kdy se pozastavuje vlákno. Pokud by došlo k jeho pozastavení uprostřed systémového volání, může dojít k pádu systému. Proto je lepší, když program kontroluje, kdy se spustí. Zvlášť problémové bude, pokud program nebude v dosahu knihovny „NTDLL.dll“ [39].

2.4.6 Sandboxing

Sandboxing je metoda, která umožňuje spouštět programy v programátorem kontrolovaném prostředí. Toto prostředí ovšem musí být pro testovaný program nerozeznatelné od skutečného, aby bylo možné sledovat a zkoumat jeho skutečné chování. V případě, že je detekováno nežádoucí chování programu, je program umístěn do karantény nebo smazán.

Sandbox může být jak lokální (program, který sleduje chování programu a na základě toho vyhodnocuje výsledky, například jako součást antivirových programů), ale může jít i například o virtualizovaný, emulovaný nebo plně implementovaný operační systém. Sandbox vyhodnocuje nejen chování programu, ale i jeho importovaných modulů [40].

2.4.7 Zpětná rekonstrukce kódu

Spustitelné binární soubory typicky obsahují méně informací než samotný zdrojový kód. To je dáno tím, že při transformaci ze zdrojového kódu na binární kód se ztrácí informace, které nejsou pro spustitelný program důležité, například pojmenování funkcí či proměnných.

Proto je zpětné rekonstruování kódu složitý proces, při kterém se nelze dostat k tomu stejnému zdrojovému kódu, ale pouze k hodně podobnému. K tomu se používají dvě metody: disassembling a dekompilace. Obě metody se používají ke snadnější následné statické nebo dynamické analýze kódu.

Disassembly

Disassembling je proces, kdy se binární kód, který je čitelný pro procesor, přeloží do assembly kódu, který je čitelný pro člověka. Popíšeme několik základních přístupů k dissasembly, jaké jsou jejich výhody a nevýhody.

Nejjednodušší metodou je tzv. lineární průchod, neboli „linear sweep“. Disassembly provádí lineárně bajt po bajtu, instrukci po instrukci. Tento algoritmus je rychlý a jednoduchý. Problém nastane, když se assembly nechá zmást, například když jsou data smíchána s kódem, začátkem jinde než na začátku funkce nebo nejednoznačným tokem kódu. V této chvíli začíná být disaasembly nesmyslné [42].

Další metodou je tzv. rekurzivní průchod, neboli „recursive traversal“. Rekurzivní průchod provádí disassembly na základě toku kódu. Začne se na vstupním bodu programu, začne disassemblovat instrukce a označí je jako navštívené. Pokud se jedná o skok, tak si zapamatuje návratovou adresu a pokračuje na adrese skoku. Až dokončí analýzu větve, pokračuje znovu na zapamatované adrese. Tento přístup je pomalejší než lineární. Všechny navštívené adresy označí jako kód, zbytek jako data. [42]

Další metodou, jak provést dissasembly, je tzv. rozšířený lineární průchod. Používá následující postup: [42]

1. Nejprve pro každou sekvenci N adres označme posledních $N - K_{max}$ adres jako data. ($N > K_{max}$, kde K_{max} je maximální počet adres v jedné instrukci, závislé na architektuře, pro x86 platí $K_{max} = 2$).
2. Potom pro každou neoznačenou adresu v kódovém segmentu:
   (a) Použijeme lineární průchod, zastavíme se při dosažení označené adresy.
   (b) Pokud poslední označená instrukce „přetekla“ do již označené adresy, označme ji jako data.
   (c) Prozkoumejme poslední instrukci. Buď m počet adres ve skokové tabulce, $0 < m < K_{max}$, potom je mezi koncem této instrukce $K_{max} - m$ neoznačených adres, každou z nich označme jako data.

Nejsofistikovanější metodou je tzv. hybridní přístup, který se snaží vyvážit výhody a nevýhody předchozích přístupů. Použije rozšířený lineární průchod pro prvotní dissasembly a rekurzivní průchod pro ověření toku kódu. Probíhá následovně:

1. Na každou instrukci funkce použijeme rekurzivní průchod.
2. Ověříme každou instrukci I získanou na adrese a_i, jestli byla získána i prvotním rozšířeným lineárním průchodem. Pokud ne, zahlásíme chybu.
3. Pokud nedošlo k chybě, zahlásíme úspěch.

Dekompilace

Dekompilace je proces, kdy se z instrukcí v assembleru zkonstruuje kód ve vyšším programovacím jazyku. Nenahrazuje disassembly, ale pokud se povede, není disassembly potřeba. Snaží se kód převést do (pro člověka) ještě čitelnější podoby, než je posloupnost instrukcí v assembly. Dekompilace probíhá v těchto krocích: [43]

1. Dekódování formátu binárního souboru.
2. Dekódování strojových instrukcí do assembly kódu pro daný stroj.
3. Provedení sémantické analýzy k obnovení některých základních datových typů. Tím se zlepší dekódování dalších instrukcí.
4. Uložení informací ve vhodné reprezentaci.
5. Provedení analýzy toku dat (data flow), odstranění nízkoúrovňových reprezentací (registry, zásobník).
6. Provedení analýzy řídícího toku (control flow), obnovení řídících struktur (cykly, podmínky).
7. Provedení typové analýzy (type analysis) a obnovení dalších typů ve vysokoúrovňovém jazyku. Je to jedna z nejtěžších částí a často vyžaduje lidský zásah.
8. Vygenerování kódu ve vysokoúrovňovém jazyku.

2.5 Využití pomocných programů

Reverzní inženýři mají k dispozici nástroje, které mohou použít k analýze nebo k ověření toho, jestli jsou jejich výsledky správné. Jejich použití je doporučené, mohou analytikovi usnadnit mnoho práce při analýze programu. Příklady programů, které mohou reverzním inženýrům pomoci:

- IDA Free[8], Ghidra[9], debuggery, například OllyDbg[10] nebo x64dbg[11], Dependency Walker[12], API Monitor[13], hex editory, například HxD[14], Resource Hacker[15], Process Explorer[16], Process Monitor[17], Strings[18].

[8] https://hex-rays.com/ida-free/
[9] https://www.ghidra-sre.org/
[10] http://www.ollydbg.de/
[11] https://x64dbg.com/
[12] https://www.dependencywalker.com/
[13] http://www.rohitab.com/
[14] https://mh-nexus.de/en/hxd/
[15] http://www.angusj.com/resourcehacker/
[16] https://learn.microsoft.com/cs-cz/sysinternals/downloads/process-explorer
[17] https://learn.microsoft.com/cs-cz/sysinternals/downloads/procmon
[18] https://learn.microsoft.com/cs-cz/sysinternals/downloads/strings

Kapitola 3

Návrh aplikace

V této kapitole vytvoříme návrh aplikace Portabler, která umožní spouštět programy v přenositelném módu. Také shrneme, co všechno bude muset aplikace umět.

3.1 Portabler

Hlavním úkolem aplikace Portabler bude umožnit aplikacím běžet v přenosném režimu. Jejím úkolem bude zejména nahradit nebo přesunout prostředky, které se nachází mimo instalační složku programu. Budeme se soustředit na lokální prostředky popsané v první kapitole, tedy na Windows registr a na souborový systém. Aplikace bude dostupná jak ve 32 bitové, tak v 64 bitové verzi.

Souborový systém nebudeme muset nijak speciálně simulovat. Pouze pokud bude program zasahovat mimo instalační složku, tak změníme umístění souboru a danou operaci provedeme nad přemístěným souborem.

U registru nebude stačit měnit cestu ke klíči, to nám nepomůže, ale budeme muset vytvořit obdobu registru pro daný program. To bude vyžadovat přítomnost nové lokální databáze a vytvoření API, které bude s databází pracovat. V databázi tedy budou nasimulovány klíče registru, které bude aplikace využívat.

Cílem takové modifikace není změna chování programu jako takového, ale pouze záměna potřebného rozhraní. Dopad na používání uživatele by měl být minimální, v ideálním případě naprosto žádný. Nesmí být změněna žádná funkčnost programu. Nové rozhraní by také nemělo mít příliš velký vliv na rychlost.

Portabler bude spustitelný program, který umožní jiné aplikaci používat nově naimplementované rozhraní. K tomu bude muset využít prostředky, které se používají v reverzním inženýrství.

3.2 Využití technik RE

V následující sekci popíšeme, jaké techniky reverzního inženýrství v aplikaci použijeme.

3.2.1 Hooking funkcí

V aplikaci bude potřeba implementace metody hookování funkcí, kterou jsme popsali v sekci 2.4.4. Budeme pomocí ní hookovat Windows API pracující se souborovým systémem a API pracující s Windows registrem. U souborových systémů dosáhneme změny pomocí záměny cest k souborům, které se nacházejí mimo instalační složku, u registru API úplně nahradíme a vytvoříme nové API pracující s lokální databází. Nové API musí mít ve výsledku stejné rozhraní jako původní Windows API, aby se dal hooking použít.

3.2.2 Injekce kódu

Aby aplikace používala naše rozhraní místo původního, budeme ho muset do aplikace nainjektovat. K tomu využijeme injekci kódu popsanou v sekci 2.4.5. Vybereme jednu z popsaných technik a s její pomocí zavedeme do programu náš kód, který provede hooking vybraných API funkcí s novým rozhraním.

3.3 Výběr API funkcí

Jedním z nejdůležitějších bodů je vybrat množinu funkcí, kterou budeme chtít nahrazovat. Budeme nahrazovat přímo funkce volané programy. V následujících sekcích popíšeme funkce, které chceme nahrazovat. V přílohách poskytneme plný seznam nahrazovaných funkcí.

3.3.1 API pro práci s registrem

Celé API pro práci s Windows registrem můžeme najít v dokumentaci [44]. Pro napojení na naši databázi budeme potřebovat především tu část API, která přímo pracuje s konkrétními daty aplikace. Jedná se zejména o tyto operace:

- vytvoření klíčů registru,
- otevření klíče registru,
- mazání klíčů,
- mazání hodnot,
- výčet informací o klíči,
- získání dat z registru,
- vložení dat do registru,
- další vybrané funkce.

Drtivá většina z těchto funkcí má dvě verze — jedna používá ANSI řetězce, funkce má příponu „A“, například `RegCreateKeyA`, druhá používá kódování UNICODE a má příponu „W“, například `RegCreateKeyW`.

3.3.2 API pro práci se soubory

API pro práci se soubory můžeme najít v dokumentaci [45]. Je potřeba změnit funkce, které jako argumenty přijímají cestu k souboru. Některé operace nevyžadují v argumentu jméno souboru, ale jeho `HANDLE`. Dá se označit jako tzv. „opaque pointer“ [47], tedy pointer, jehož implementace je programátorovi skrytá. Takové operace řešit nemusíme, protože validní `HANDLE` byl vytvořen předem funkcí, jejíž cestu jsme modifikovali dříve při jeho vytváření. Z funkcí pro práci se soubory jsme vybrali tyto operace, které se změní:

- vytváření souborů,
- mazání souborů,
- kopírování souborů,
- přesunování souborů,
- zjištění informace o souborech,
- další vybrané funkce.

3.4 Databáze

V Portableru budeme potřebovat databázi pro uchování dat naší kopie registru a pro uložení informace o mapování souborů. Budeme potřebovat lehce přenositelnou databázi, která zabere pokud možno co nejméně místa. Nejlepší bude buď nějaká lehká relační databáze s SQL nebo „databáze“ v textovém souboru.

Do databáze budeme ukládat simulovanou část registru, bude to tedy jakási lokální kopie té části registru, kterou bude aplikace používat. Dále budeme chtít přenášet některé složky. Přenesené složky budou mít jiné umístění, ale původní program bude stále počítat se starým původním umístěním, proto budeme potřebovat mapování mezi novým a starým umístěním souborů.

3.5 Logování

Abychom lépe poznali, co se uvnitř aplikace skutečně děje, použijeme k tomu logování. Pokud v naší aplikaci nastanou neočekávané chyby, zprávy z logu nám pomohou lépe pochopit, kde nastala chyba, abychom jí mohli následně odstranit. Také pomůže zaznamenávat, co program přibližně dělá.

3.6 Synchronizace zdrojů

Jelikož Portabler může fungovat i ve vícevláknových aplikacích, bude muset ke kritickým strukturám přistupovat synchronizovaně. Pro přístup k důležitým strukturám bude potřeba tento přístup nějakým způsobem řídit.

3.7 Součásti aplikace

Program bude mít více funkčností — hlavní bude řídící program, konzolová aplikace, která bude řídit zbytek programu. Další důležitou součástí bude kód, který bude dělat hlavní logiku, zejména nové API pro přístup k databázi, která bude nahrazovat registr. Také bude potřeba kód, který bude řídit změnu cesty k přenášeným složkám. Další součástí bude nástroj pro snadnější testování kódu. Bude vhodné, když bude aplikace rozdělena na více logických celků podle funkčnosti.

3.8 Vstup

Aplikace Portabler bude potřebovat vstup, který se skládá ze dvou částí:

- cesty k simulované části registru,
- cesty k simulovaným souborům.

Ačkoli je tento vstup pro aplikaci nutný, je problém ho algoritmicky získat. Neexistuje žádné univerzální místo, ze kterého bychom mohli vyčíst všechny složky, které program používá. Podobná situace platí i pro registr. Programy ale často využívají typická umístění, u souborů se například jedná o složku v `Program Files` nebo v `AppData`. Složka `Program Files` navíc vyžaduje zvýšená oprávnění. Pokud aplikace používá registr, obvykle si vytváří svůj klíč v `HKEY_CURRENT_USER\SOFTWARE` a jeho případné podklíče.

3.8.1 Jak vstup získat

Součástí Portableru bude tzv. „učící“ mód, který by uživateli mohl usnadnit nebo mu nabídl k výběru složky, ke kterým program přistupuje. Uživatel by si před převedením programu na přenositelný mohl vybrat a kontrolovat, které složky chce přemístit, a které klíče registru chce simulovat.

Jeden z možných přístupů je ten, že každý nalezený klíč registru budeme hned při zjištění simulovat do databáze a případné změny v registru provádět „živě“ a každému nalezenému souboru budeme rovnou přidávat novou cestu někam do instalační složky programu. Výhoda tohoto přístupu spočívá v jednoduchosti — registry jsou simulovány a mapování souborů je vytvořeno hned, vše probíhá automaticky. Simulování probíhá v době běhu programu, nepotřebuje tedy mnoho času navíc, pokud vůbec nějaký. Nevýhoda tohoto

přístupu je ta, že jako uživatel nemáme zcela kontrolu nad tím, co se ve skutečnosti provádí. To by bylo v pořádku, kdyby aplikace jako taková vždy zaručovala, že aplikace pokaždé nalezne všechny cesty a všechny klíče registru. Kolik takových vstupů najde, hodně záleží na tom, jaký při učení zvolíme průchod aplikací. Sice zachytíme všechny klíče a soubory pro vybraný průchod, ale nedokážeme zjistit, s jakými zdroji pracují nepoužité funkcionality.

Dalším možným přístupem je takový, že žádné mapování nebude probíhat rovnou, ale učící mód si bude v době běhu aplikace jenom pamatovat, které klíče registru aplikace použila a se kterými soubory pracovala. Předejdeme tím některým duplikacím. Pokud se s klíčem nebo souborem pracovalo vícekrát, nebudeme zde muset řešit úpravu dat v databázi. Nevýhodou tohoto přístupu je, že bude vyžadovat následné zpracování. Zpracování poté může proběhnout automaticky po skončení běhu programu. Aplikace by tak potřebovala čas navíc po dokončení běhu původního programu, ale bude ho méně zatěžovat během něj.

Zůstává ale problém ohledně zjišťování všech klíčů během průchodu aplikací. Představme si situaci, že aplikace nenavštíví klíč, který vyžaduje některá funkcionalita, kterou uživatel ve svém průchodu nevyužil. Pokud by chtěl uživatel do aplikace tento klíč i s hodnotami přidat, musel by přidat klíč a všechny hodnoty do konkrétních tabulek v databázi. Takový postup může být značně komplikovaný a zdlouhavý. Řešením by bylo, kdyby učící mód jako takový vůbec žádnou simulaci nebo mapování neprováděl. To by znamenalo, že učící mód by jen někam jako výstup vypsal, jaké klíče registru a jaké složky navštívil, následné zpracování potom bude muset proběhnout odděleně. Hlavní nevýhoda tohoto přístupu je taková, že uživatel bude muset zpracování sám separátně spustit. Naopak výhoda pozdějšího zpracování je taková, že uživatel v případě chybějících klíčů je může snadno doplnit připsáním nového klíče, podobně u složek může snadno připsat simulovanou složku. Takový přístup sice vyžaduje operaci navíc, ale na druhou stranu má uživatel prakticky plně pod kontrolou, co se simuluje a přenáší. Případné změny nebo přidání nových klíčů proběhnou automaticky bez nutnosti zásahu přímo do nových tabulek.

Poslední z přístupů vyžaduje přidání dalšího kroku, tzv. „konfigurace“, kdy uživatel bude muset pro přenos udělat krok navíc. Na druhou stranu mu kofigurace přinese výhodu, že bude mít větší kontrolu nad tím, jaká data se budou zpracovávat. Například aplikace pracující s filesystémem, při procházení složek, učící mód ukáže navštívené složky a soubory. Přitom ale může jít o běžné soubory, které by bylo nežádoucí mapovat nebo přenášet. Pokud bude chtít uživatel přenést složku, kterou učící mód nenašel, bude jí muset přidat do vstupu konfigurace k výstupu učícího módu.

Aplikace kromě spouštění sice nebude přímo vyžadovat zásah uživatele, ale minimálně kontrola bude velmi doporučena. Bude tedy doporučena větší interakce a spolupráce uživatele za cenu snažšího přidávání potenciálně chybějících nebo odebírání přebývajících cest k souborům i klíčům registru.

3.9 Konfigurace

Aplikace bude mít konfigurační mód, který bude zajišťovat simulaci registru a přenášení složek. Na jeho vstupu bude seznam klíčů registru a seznam souborů nebo složek k namapování. Tento vstup se získá učícím módem a následnou kontrolou uživatelem, jak jsme popsali v sekci 3.8.1. Konfigurace provede zpracování klíčů registru na vstupu tak, že pomocí Windows API pro každý klíč zjistíme všechny jeho hodnoty a uložíme je do databáze. Po zpracování všech klíčů na vstupu budeme v databázi mít lokální kopii těchto klíčů. Konfigurace také provede kontrolu složek pro mapování. Jako vstup bude seznam souborů a složek zjištěný výstupem učícího módu a kontroly uživatele. Po zpracování budou v mapovacích tabulkách ty soubory a složky, které na svých cílových destinacích budou skutečně existovat.

3.10 Jak bude probíhat přenositelnost

Aby program fungoval přenosně, bude vyžadovat simulovanou část registru, který používá a složky, které bude potřebovat přenést. Pokud registr simulován nebude, může se aplikace chovat neočekávaně, jelikož nebude mít žádná data z registru. Pokud nebude žádné mapování složek, žádná složka se nepřenese. Přenositelnost tedy bude mít význam až po provedení konfigurace. Na rozdíl od učení a konfigurace, zde očekáváme, že ke spuštění v přenositelném režimu bude docházet opakovaně.

3.11 Představa o používání

Představa o používání je taková, že nejprve se na aplikaci spustí učící mód, ve kterém uživatel zkusí udělat všechny možné akce, které ho napadnou nebo ty, které bude používat v přenesené verzi. Po skončení učícího módu musí přijít zpracování dat, vytvoření databázové struktury a naplnění dat do databáze. Až po zpracování dat bude žádoucí spouštět aplikaci v přenosném módu.

Průchod učícím módem důrazně doporučujeme, ale není bezpodmínečně nutný. Pokud uživatel dokáže zjistit nebo zná všechny klíče registru a složky, které používá, stačí je jednoduše zapsat na místo výstupu. Podobně jako průchod učením, průchod konfigurací důrazně doporučujeme, ale není nutný. Pokud ale uživatel dokáže data z registru nasimulovat ručně, ověření složek také a nebude to z nějakého důvodu chtít dělat automaticky, může konfiguraci přeskočit.

Program půjde dál normálně spustit tak, jak byl původně nainstalován. Přenositelnost bude do programu přidána další složkou, která bude obsahovat všechny potřebné soubory, které přenositelnost usnadní.

Kapitola 4

Implementace

V této kapitole popíšeme konkrétní implementaci konceptu představeného v předchozí kapitole, rozebereme aplikaci jako celek a také popíšeme implementační detaily.

4.1 Implementační prostředí

Aplikaci jsme napsali v jazyce Visual C++ ve Visual Studio 2019, použili jsme standard C++ 14, operační systém Windows 10 Education. Solution ve Visual Studiu obsahuje 5 projektů, které popíšeme v následující sekci. V aplikaci používáme příkaz `using namespace std;`, abychom nemuseli před všemi strukturami psát `std::`. Aplikaci lze spustit v celkem 5 režimech, které popíšeme níže v sekci 4.2.1.

4.2 Části Portableru

Aplikace se skládá z následujících projektů:

- Portabler, konzolová aplikace,
- StorageLibrary, statická knihovna,
- ScanLibrary, dynamická knihovna,
- PortablerLibrary, dynamická knihovna,
- DllTester, konzolová aplikace,
- SQLite databáze a logovací soubor.

4.2.1 Portabler

Portabler je konzolová aplikace, která řídí, ve kterém režimu se aplikace spustí. Těchto režimů je celkem 5 a jsou následující:

- SCANNEW,
- SCANEXISTING,
- CONFIG,
- PORTABLER,
- PORTABLERCONFIG.

Režim SCANNEW vyžaduje jeden další argument, a to cestu ke spustitelnému souboru, na který se aplikace napojí. K napojení na aplikaci využijeme dynamickou knihovnu ScanLibrary popsanou v sekci 4.2.3.

Dalším režimem je SCANEXISTING, který je podobný režimu SCANNEW, jen s tím rozdílem, že se napojí na již běžící program. Také vyžaduje jeden argument, a to PID procesu, na který se napojí.

Důležitým režimem je CONFIG. Nevyžaduje žádný uživatelský vstup z konzole, ale svůj vstup si bere z databázové tabulky PortablerConfig. Tuto tabulku lze naplnit ručně pomocí návodu nebo pomocí předchozího spuštění SCANNEW nebo SCANEXISTING.

Posledním samostatným režimem je PORTABLER, který spustí aplikaci v přenositelném režimu. Napojí se na nově vytvořené API pro registry a pracuje s naší databází místo s registrem a na API pro soubory, které kontroluje jména souborů a v případě potřeby volání přesměruje. K napojení využijeme knihovnu PortablerLibrary.

Režim PORTABLERCONFIG spojuje dohromady režimy CONFIG a PORTABLER. Nejprve si vezme vstup z tabulky, zpracuje ho a následně spustí aplikaci v přenositelném režimu. Nepotřebuje žádný argument z konzole, jaký program spustit, ví také z tabulky.

4.2.2 StorageLibrary

StorageLibrary je statická knihovna, jejímž účelem je především zamezení duplikací v kódu. Nachází se zde kód, který potřebují alespoň 2 projekty. Mezi součásti StorageLibrary patří dvojice hlavičkový soubor a implementační soubor s názvy:

- DatabaseManager,
- HookingManager,
- Logger,
- Helper,
- Constants.

DatabaseManager, jak název napovídá, zaštiťuje připojení k databázi. Obsahuje potřebné funkce pro práci s databází a jednotlivé dotazy. Jelikož s databází pracují všechny 3 další projekty, je vhodné mít implementaci na jednom místě.

HookingManager zajišťuje hookování funkcí. Funkce se hookují pomocí IAT hooking. Obsahuje také strukturu HookRecord, která uchovává umístění původních funkcí. Toho využívá především API pro správu souborů, které po kontrole cest k souboru volá původní API.

Logger spravuje logování do souboru i do konzole. Uživatel se může podívat, jaké funkce byly nalezeny a sledovány, jaké dotazy na databázi se provádí a v neposlední řadě, jestli došlo k nějakým chybám při běhu.

Další součástí je Helper, ve kterém jsou různé funkce použité napříč projekty, například převody mezi jednotlivými typy nebo wrappery pro použité API funkce.

Poslední částí je samostatný hlavičkový soubor Constants.h, který obsahuje konstanty a případné definice nových typů. Jsou zde definice pointerů na funkce, které jsou využity při hookování. Definuje také nový datový typ `TNUM`, který je typu `DWORD` ve 32 bitovém prostředí nebo typu `__int64` pro 64 bitové prostředí. Jeho účelem je předávání funkcí.

4.2.3 ScanLibrary

ScanLibrary je dynamická knihovna, jejímž účelem je sledování vybraného API. Je hlavní komponentou pro sledování v režimech SCANNEW a SCANEXISTING v aplikaci. Má 2 součásti — ScanLibrary a ScanManager. ScanLibrary obsahuje funkce, které se volají při načtení DLL do procesu, při odpojení DLL od procesu a nové rozhraní funkcí. ScanManager ukládá cesty k souborům, zpracovává je a po odpojení DLL přidá do databáze.

4.2.4 PortablerLibrary

PortablerLibrary je dynamická knihovna, která obsahuje rozhraní pro spuštění aplikace v přenositelném módu. Implementaci jsme rozdělili na 3 části, které jsou PortablerLibrary, RegistryAPI a FilesystemAPI. PortablerLibrary obsahuje funkce volané při načtení DLL, při odpojení DLL a nové rozhraní funkcí. Volání funkcí je přesměrováno právě do dalších 2 implementačních souborů.

4.2.5 DllTester

DllTester je konzolová aplikace, jejímž smyslem je testování PortablerLibrary. Umožňuje ladění přímo v kódu uvnitř DLL. Aplikace byla vytvořena pomocí návodu od Microsoftu [51]. Obsahuje interface pro testy a testy samotné. Více o testování v kapitole 5.

4.3 Schéma aplikace

Ve složce Portabler jsme připravili všechny potřebné soubory a také skripty, ze kterých lze aplikaci spouštět. Aplikace ve Visual Studiu jsme nastavili tak, aby vytvořily schéma podle obrázku 4.1. Na obrázku jsme podrobně ukázali schéma pro 32 bitovou verzi. Schéma pro 64 bitovou verzi je ve složce „portabler64“ stejné jako pro 32 bitovou ve složce „portabler32“ jen s rozdílem, že soubory mají příponu „64“ místo „32“.

- `Portabler` ..... složka Portabler
  - `movedfiles` ..... složka s přenesenými soubory
  - `portabler32` ..... složka se setupem pro 32 bitové aplikace
    - `internals` ..... složka s interními soubory aplikace
      - `Portabler32.exe` ..... spustitelný program Portabler
      - `PortablerLibrary32.dll` ..... knihovna PortablerLibrary
      - `ScanLibrary32.dll` ..... knihovna ScanLibrary
      - `sqlite3.dll` ..... knihovna sqlite3 pro práci s databází
    - `Config32.bat` ..... spuštění Portableru v režimu CONFIG
    - `Portabler32.bat` ..... spuštění Portableru v režimu PORTABLER
    - `PortablerConfig32.bat` ..... spuštění Portableru v režimu PORTABLERCONFIG
    - `ScanExisting32.bat` ..... spuštění Portableru v režimu SCANEXISTING
    - `ScanNew32.bat` ..... spuštění Portableru v režimu SCANNEW
  - `portabler64` ..... složka se setupem pro 64 bitové aplikace
  - `portabler.log` ..... logovací soubor
  - `portabler.db` ..... databázový soubor

Obrázek 4.1: Schéma Portableru

4.4 Databáze

V následující sekci uvedeme, jakou jsme vybrali databázi a popíšeme její strukturu.

4.4.1 Schéma databáze

Jako databázi používáme databázový engine SQLite [48]. SQLite je in-memory databáze, není ji třeba instalovat a nevyžaduje žádný SQL server. Pro nás je důležité, že jí můžeme přenášet. Jak jsme již zmínili, dotazy na databázi jsou implementované ve StorageLibrary, vzniká zde tedy závislost aplikace na knihovně „sqlite3“. Databáze se skládá z 5 samostatných tabulek:

- PortablerConfig,
- RegistryKeysInfo,
- RegistryValuesDetails,
- FolderPrefixMapping,
- FileMapping.

Tabulky pro práci s registrem

Tabulky RegistryKeysInfo a RegistryValuesDetails uchovávají informace o simulovaných registrech. Tabulka RegistryKeysInfo ukládá informace o klíčích, tabulka RegistryValuesDetails o jednotlivých hodnotách klíčů. Tabulka RegistryKeysInfo má následující sloupce:

- `ID INTEGER PRIMARY KEY`, identifikátor, alias pro rowid v tabulce,
- `Hive TEXT NOT NULL`, jeden z předem definovaných klíčů registru,
- `Path TEXT`, cesta k podklíči,
- `ParentID INTEGER`, odkaz na předka.

Tabulka RegistryValuesDetails má následující sloupce:

- `ID INTEGER PRIMARY KEY`, identifikátor, alias pro rowid v tabulce,
- `KeyID INTEGER NOT NULL`, ID klíče, kterému hodnota náleží,
- `RegName TEXT`, jméno hodnoty v klíči,
- `RegType INTEGER`, typ hodnoty,
- `RegValue BLOB`, hodnota,
- `RegValueLength INTEGER`, délka hodnoty.

V tabulce RegistryValuesDetails je jeden cizí klíč v podobě odkazu mezi ID v RegistryKeysInfo a KeyID v RegistryValuesDetails. Jeho předpis je `FOREIGN KEY(KeyID) REFERENCES RegistryKeysInfo(ID)`.

Tabulky pro práci se soubory

Tabulky FolderPrefixMapping a FileMapping uchovávají informace o nových cestách přenesených souborů. Tabulka FolderPrefixMapping má uložené informace o složkách, které jsou nově přemístěné v Portableru. Tabulka FileMapping ukládá mapování jednotlivých souborů, která slouží zejména pro snažší vyhledávání již použitých souborů. Tabulka FolderPrefixMapping má následující sloupce:

- `ID INTEGER PRIMARY KEY`, identifikátor, alias pro rowid v tabulce,
- `OriginalPath TEXT NOT NULL`, originální cesta k adresáři,
- `UpdatedPath TEXT`, nová cesta k adresáři.

Tabulka FileMapping má následující sloupce:

- `ID INTEGER PRIMARY KEY`, identifikátor, alias pro rowid v tabulce,
- `DefaultPath TEXT NOT NULL`, původní cesta k souboru,
- `NewPath`, nová cesta k souboru.

Mezi tabulkami není žádná dodatečná vazba. Ačkoli obě tabulky slouží k ukládání mapování cest, ať už úplných nebo částečných, nemají tabulky mezi sebou žádnou vazbu v podobě cizích klíčů. A to zejména proto, že FileMapping dovoluje mapování samostatných souborů, které nemusí být v žádné z přenášených složek.

Tabulka PortablerConfig

Tabulka PortablerConfig udržuje informace o tom, co všechno je potřeba, aby se nasimulovalo a namapovalo. Obsahuje jak informace o částech registru, které je třeba nasimulovat, tak o souborech a složkách, které budou mít nové umístění. Tabulka funguje jako výstup režimů SCANNEW a SCANEXISTING, kdy se zachycené soubory i registry přidají do této tabulky. Dále je to vstup pro režimy CONFIG a PORTABLERCONFIG, kdy se na základě obsahu nasimulují registry a vytvoří se mapování mezi jednotlivými soubory a složkami. Tabulka PortablerConfig má následující sloupečky:

- `ID INTEGER PRIMARY KEY`, identifikátor, alias pro rowid v tabulce,
- `Mode TEXT`, mód ke zpracování,
- `Value TEXT`, zdrojová hodnota pro zpracování,
- `TargetValue TEXT`, cílová hodnota pro zpracování.

Vysvětlení hodnot a více o tabulce v sekci 4.6.

4.4.2 Rozhraní databáze

Knihovnu pro přístup k databázi „sqlite3.dll“ poskytuje Visual Studio při sestavování aplikace, umístí ji do stejné složky jako výstupní soubory projektu. Knihovna je potřebná pro všechny součásti, které používají hlavičkový soubor `<sqlite3.h>`, a tak využívají přístup k databázi.

Aplikace Portabler používá funkce z této knihovny ve StorageLibrary v „DatabaseManager.h“ a „DatabaseManager.cpp“. Poskytuje potřebné SQL dotazy na databázi. Příkazy, které vyžadují nějaký vstup, jsou implementovány pomocí „prepared statements“. Hlavním smyslem prepared statements je ochrana proti útokům typu SQL injection. V SQLite je řešíme pomocí posloupnosti funkcí:

1. připravení objektu pomocí `sqlite3_prepare_v2()`,
2. svázání hodnot do parametrů nahrazením „?“ přes `sqlite3_bind_*()` podle datového typu,
3. volání SQL dotazu pomocí `sqlite3_step()` jednou nebo vícekrát,
4. destrukce objektu pomocí `sqlite3_finalize()`.

U dotazů, které nevyužívají žádný vstup, nejsou prepared statements implementovány. Takovým příkladem je smazání z tabulky nebo vybrání všech dat z tabulky.

4.4.3 Ochrana databáze

Ochrana databáze jde ruku v ruce s ochranou databázového souboru „portabler.db“. Kdo bude mít přístup k databázovému souboru, bude mít zároveň přístup ke všem datům v simulovaném registru i k mapování mezi soubory a složkami.

4.5 Využití reverzního inženýrství

Z technik reverzního inženýrství jsme využili 2 techniky, a to injekce kódu a hookování funkcí.

4.5.1 Použití injekce kódu

Injekci kódu používáme v aplikaci Portabler v souboru „Injector.cpp“. Injektuje DLL pomocí metody DLL injection popsané v sekci 2.4.5. Kód 4.1 ukazuje konkrétní implementaci, jen v ní chybí deklarace proměnných a kontrolování návratových hodnot. Režimy SCANNEW a SCANEXISTING takto injektují do procesu ScanLibrary, režimy PORTABLER a PORTABLERCONFIG injektují PortablerLibrary. Funkce pro injekci kódu je volaná celkem 2×, nejprve z důvodu závislosti injektujeme knihovnu „sqlite.dll“ a až následně injektujeme požadovanou knihovnu.

Program 4.1: Ukázka DLL injection z kódu, zde bez kontroly validity výsledků funkcí a bez deklarace proměnných.

```
HANDLE hProcess = OpenProcess(PROCESS_CREATE_THREAD | PROCESS_VM_OPERATION |
    PROCESS_VM_WRITE | PROCESS_VM_READ, FALSE, PID);
void* lpDataAddr = VirtualAllocEx(hProcess, NULL, dllPathLen, MEM_RESERVE |
    MEM_COMMIT, PAGE_READWRITE);
SIZE_T szBytesWritten;
BOOL bWritten = WriteProcessMemory(hProcess, lpDataAddr, dllPath, dllPathLen,
    &szBytesWritten);
HANDLE hThread = CreateRemoteThread(hProcess, NULL, NULL, (
    LPTHREAD_START_ROUTINE)(&LoadLibraryA), lpDataAddr, NULL, NULL);
WaitForSingleObject(hThread, INFINITE);
```

4.5.2 Použití hookování funkcí

V aplikaci také používáme hooking funkcí na vybrané Windows API. Hookovat budeme vybrané funkce pracující se souborovým systémem a vybrané funkce z API pro přístup k registru. Implementaci lze najít ve statické knihovně StorageLibrary v souboru „HookingManager.cpp“. Hookování API pro soubory je důležité z toho důvodu, že chceme, aby souborové funkce pracovaly se soubory, které mají ve skutečnosti jiné umístění. Jiné parametry měnit nebudeme, zatímco u API pro přístup k registru bude úplně zcela nové. Kód 4.2 ukazuje část programu, který změní adresu funkce. V ukázce chybí ověřování hodnot a deklarace proměnných. Jelikož hookujeme funkce pouze z knihoven „KERNEL32.dll“ a „”ADVAPI32.dll”“, kontrolujeme jméno knihovny, a pokud není ani jedno z nich, tak ji přeskočíme.

Program 4.2: Ukázka hookování funkcí z kódu, zde bez kontroly validity výsledků funkcí a bez deklarace proměnných.

```
HMODULE hPEFile = GetModuleHandleW(NULL);
PIMAGE_DOS_HEADER pDosHeader = (PIMAGE_DOS_HEADER)hPEFile;
PIMAGE_NT_HEADERS pNTHeaders = (PIMAGE_NT_HEADERS)(((BYTE*)pDosHeader) +
    pDosHeader->e_lfanew);
PIMAGE_IMPORT_DESCRIPTOR pImportDescriptor = (PIMAGE_IMPORT_DESCRIPTOR)(((BYTE
    *)pDosHeader) + pNTHeaders->OptionalHeader.DataDirectory[
    IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress);
PIMAGE_IMPORT_DESCRIPTOR pImportDescriptorSize = (PIMAGE_IMPORT_DESCRIPTOR)(((
    BYTE*)pImportDescriptor) + pNTHeaders>OptionalHeader.DataDirectory[
    IMAGE_DIRECTORY_ENTRY_IMPORT].Size);
for (; pImportDescriptor < pImportDescriptorSize && pImportDescriptor->
    OriginalFirstThunk != NULL; ++pImportDescriptor)
{
    if (!(StringFindCaseInsensitive(nameString, "KERNEL32.dll") != string::
        npos || StringFindCaseInsensitive(nameString, "ADVAPI32.dll") !=
        string::npos)) {
        continue;
    }
    if (pImportDescriptor->Characteristics != NULL) {
        PIMAGE_THUNK_DATA OriginalFirstThunk = (PIMAGE_THUNK_DATA)((char*)
            pDosHeader + pImportDescriptor->OriginalFirstThunk);
        PIMAGE_THUNK_DATA FirstThunk = (PIMAGE_THUNK_DATA)((char*)pDosHeader +
             pImportDescriptor->FirstThunk);
        for (; OriginalFirstThunk->u1.AddressOfData != NULL; ++
            OriginalFirstThunk)
        {
            PIMAGE_IMPORT_BY_NAME NameImg = (PIMAGE_IMPORT_BY_NAME)(((BYTE*)
                pDosHeader) + OriginalFirstThunk->u1.AddressOfData);
            TNUM test = OriginalFirstThunk->u1.Function & IMAGE_ORDINAL_FLAG;
            if (test == 0)
            {
                if (strcmp(functionName.c_str(), (const char*)NameImg->Name)
                    == 0)
                {
                    DWORD defaultAccessProtection;
                    BOOL retval = VirtualProtect(FirstThunk, sizeof(FirstThunk
                        ), PAGE_READWRITE, &defaultAccessProtection);
                    FirstThunk->u1.Function = newAddress;
                    retval = VirtualProtect(FirstThunk, sizeof(FirstThunk),
                        defaultAccessProtection, &junkProtection);
                }
            }
            ++FirstThunk;
        }
    }
}
```

4.6 Vstup v tabulce PortablerConfig

Tabulka PortablerConfig slouží jako vstup pro konfiguraci. Do tabulky PortablerConfig se posílají jak vstupní klíče registru, tak vstupní soubory a složky. Sloupeček „Mode“ nám pomáhá rozlišit, jestli jde o klíč registru, soubor k mapování nebo složka pro prefix. Pro rozlišení, jak se má řádek zpracovat, slouží následující módy:

- FILESYSTEM_SINGLE, mód pro jednotlivé soubory,
- FILESYSTEM_PREFIX, mód pro složky,
- REGISTRY_RECURSIVE, zpracování klíčů registru rekurzivně,
- REGISTRY_SINGLE, zpracování jednoho klíče registru,
- FILESYSTEM_INSTALLFOLDER, instalační složka programu,
- FILESYSTEM_PROGRAMPATH, cesta ke spustitelnému souboru aplikace.

První mód FILESYSTEM_SINGLE řídí přidání souborů do tabulky FileMapping. Do FileMapping lze přidat jak samostatné soubory, které chce uživatel přenášet, tak přidat i soubory v namapovaných složkách pro urychlení hledání souboru. Pokud v posledním sloupečku je validní cesta k souboru a soubor existuje, přidá dvojici stará a nová cesta do FileMapping. Tím je zajištěno, že ve FileMapping budou jen existující soubory.

Druhý mód FILESYSTEM_PREFIX řídí přidání přenášených složek do FolderPrefixMapping. Aby uživatel nemusel ručně přenášet všechny soubory přes FileMapping, pokud sám nechce, může přenést celou složku a předat jí v tomto módu. Pokud cílová složka na daném umístění existuje, přidá se do FolderPrefixMapping. Všechny cesty obsahující tento prefix se pak na jeho základě přemapují a budou odkazovat na přenesenou verzi.

Simulaci registru řídí REGISTRY_SINGLE a REGISTRY_RECURSIVE. Oba módy přidávají data do tabulek RegistryKeysInfo a RegistryValuesDetails. První mód simuluje pouze zadaný klíč a jeho hodnoty, druhý mód bere i rekurzivně všechny podklíče. Rekurzivní možnost je přítomna z důvodu, pokud aplikace má vytvořenou svojí vlastní větev v registru, aby stačilo zadat pouze kořenový klíč a ostatní podklíče se pak zpracují rekurzivně a na žádný se nezapomene.

Mód FILESYSTEM_PROGRAMPATH má jako hodnotu cestu ke spustitelnému souboru aplikace. Aplikace ho využívá ke spuštění aplikace. Mód FILESYSTEM_INSTALLFOLDER určuje instalační složku aplikace. Co se děje v instalační složce, nebude v Portableru simulováno, protože v tomto případě není co přenášet.

4.7 Logování

Logování probíhá do souboru „portabler.log“, jehož přesné umístění v aplikaci lze najít v obrázku 4.1. Logování řídí soubory „Logger.h“ a „Logger.cpp“. Rozhodli jsme se pro vlastní jednoduchou implementaci logu, která vezme zprávu, prioritu a přidá k ní datum a čas. Také můžeme zprávu logovat do souboru nebo do konzole. To řídí vstupní parametry, které jsme ve výchozím voláním nastavili na logování jen do souboru. Definovali jsme celkem 6 logovacích úrovní, a to `LOG_DEBUG_P`, `LOG_INFO_P`, `LOG_WARN_P`, `LOG_ERROR_P`, `LOG_FATAL_P` a `LOG_TEST_P`.

4.8 Synchronizace zdrojů

Protože aplikace může běžet i ve vícevláknových aplikacích, je proto vhodné kritické části zabezpečit kvůli potenciálnímu vstupu více vláken najednou. Mezi takové kritické části patří databáze, IAT a logovací soubor. Všechny struktury jsme zabezpečili strukturou `std::mutex` z knihovny `<mutex>` a poté pomocí funkce `unique_lock<mutex>`. U databázových funkcí každá začíná zamčením mutexu, který se uvolní až na konci funkce. IAT je chráněno vnější funkcí, která zamkne mutex, projde IAT, ve které provede případné změny a až po vrácení z funkce mutex uvolní.

4.9 Sledování aplikace

Pokud uživatel spustí aplikaci v jednom ze sledovacích režimů SCANNEW nebo SCANEXISTING, odpovídající ScanLibrary (ScanLibrary32.dll pro 32 bitové aplikace nebo ScanLibrary64.dll pro 64 bitové aplikace) se pomocí injekce kódu připojí k aplikaci.

Při připojení se DLL napojí na všechny souborové funkce a vybrané registrové funkce. Z registrových funkcí jsou vybrány takové, které identifikují klíč jako dvojici — ukazatel `HKEY` a řetězec pro subklíč. Některé registrové funkce berou jako argument pouze ukazatel `HKEY`, který se předem vytvoří pomocí volání `RegCreateKey`. Struktura `HKEY` se dá také označit jako „opaque pointer“, podobně jako u `HANDLE`, nemůžeme přistoupit k jeho vnitřní implementaci. Jelikož je `RegCreateKey` ve vybraných funkcí, tak tento klíč zachytíme už při volání `RegCreateKey`. Přijít o něj můžeme v případě, že aplikaci spustíme v režimu SCANEXISTING a `RegCreateKey` se zavolá předtím. V takovém případě nebudeme mít o tomto záznamu informaci, protože z existujícího `HKEY` neumíme získat cestu k otevřenému klíči, pokud to není jeden z definovaných klíčů. Celý výpis funkcí, které sledujeme ve ScanLibrary, je v příloze B.

Ve sledovacím módu musí všechny hookované funkce zavolat svoji původní verzi. S jediným rozdílem, že se uloží informace o tom, s jakým souborem, nebo s jakým klíčem registru funkce pracuje. Obecný postup používaný ve ScanLibrary funkcích je následovný:

1. uložení cesty k souboru / podklíče registru,
2. vyzvednutí informace o adrese původní funkce,
3. vytvoření pointeru se stejným rozhraním,
4. zavolání funkce s původními parametry.

Kód 4.3 ukazuje, jak jsou ve ScanLibrary volány původní funkce na příkladu funkce `CreateDirectoryA`.

Program 4.3: Volání původní funkce z hookované.

```
BOOL __stdcall ScanHookedCreateDirectoryA(LPCSTR lpPathName, LPSECURITY_ATTRIBUTES
     lpSecurityAttributes)
{
    SaveFilePath(lpPathName);
    HookRecord record = getHookRecord("CreateDirectoryA");
    oldCreateDirectoryA callFunction = (oldCreateDirectoryA)record.originalAddress
        ;
    return callFunction(lpPathName, lpSecurityAttributes);
}
```

Po zbytku běhu aplikace každé napojené API volání nejprve uloží cestu k souboru nebo registru, pokud je to možné, a potom zavolá původní funkci s původními argumenty. Rozhodli jsme se pro větší množství struktur, aby bylo při následném kontrolování vstupu jasnější, jak se se soubory pracovalo. Cesty ukládáme do připravených runtime struktur, které lze vidět ve výpisu 4.4.

Program 4.4: Runtime struktury ve ScanLibrary.

```
map<string, string> copiedFiles;
map<string, string> movedFiles;
map<string, string> createdDirectoriesEx;
set<string> fileEntries;
set<string> registry;
```

Struktura `copiedFiles` drží informace o souborech, které jsou kopírovány, konkrétně původní soubor a nový soubor. Podobně `movedFiles` drží informace o přesunutých souborech, ukládá původní cestu a novou cestu. Z obou struktur se pak přidají oba soubory do PortablerConfig tabulky s komentářem, že jde o kopírované/přesouvané soubory. I struktura `createdDirectoriesEx` ukládá obě cesty ke složkám do databáze, ale jako prefixy. Všechny cesty se poté zapíší do vstupní tabulky.

Do struktury `fileEntries` se přidávají jak složky, tak soubory, se kterými pracují všechny ostatní funkce, tedy při vytváření, mazání a získávání informace o souborech. Jedná se o množinu (`std::set`), takže pokud je se souborem pracováno opakovaně, v databázi bude právě jednou. Každá nalezená cesta se poté zapíše do vstupní tabulky.

Informace o navštívených klíčích se ukládá do struktury nazvané `registry`. Ukládá se celá jejich cesta od kořenového klíče. Do vstupní tabulky se zapíše

každý záznam, výchozí mód pro zpracování je REGISTRY_SINGLE. Rekurzivní mód se nastaví pouze v případě, že je v cestě přítomno jméno programu. V takovém případě je velmi pravděpodobné, že jde o větev vytvořenou aplikací a je tak žádoucí, aby se zkopírovala celá.

Při odpojení DLL od aplikace nebo po jejím ukončení uložíme data do PortablerConfig tabulky v databázi. Klíče registru uložíme do databáze všechny. Z runtime struktur pro zpracování souborů uložíme pouze soubory a složky, se kterými se pracovalo mimo instalační složku. Se soubory a složkami v instalační složce pracovat nebudeme, protože je s instalační složkou přenášíme automaticky.

4.10 Konfigurace

Konfigurace se spustí v módech CONFIG a PORTABLER_CONFIG. Provádí zpracování vstupu v tabulce PortablerConfig. Pro zpracování je klíčová funkce `ProcessPortablerConfig`, kterou lze najít v Portableru v souboru „Initializer.cpp“. Prochází se záznam po záznamu z tabulky PortablerConfig a na základě kolonky „Mode“ se provedou požadované operace. Před provedením konfigurace se smažou data z tabulek FileMapping, FolderPrefixMapping, RegistryKeysInfo a RegistryValuesDetails.

Soubory a složky se objeví v tabulkách FileMapping a FolderPrefixMapping právě tehdy, pokud je záznam v PortablerConfig a cílová složka nebo cílový soubor existuje. O existenci namapovaných souborů se musí postarat uživatel. Uživatel si může přidat i vlastní složku nebo soubory, které knihovna nenalezla. V takovém případě musí nejen složku nebo soubor nakopírovat, ale také musí přidat mapování do PortablerConfig. Pokud by tedy uživatel chtěl přidat mapování přímo do jedné z nich, musí to udělat až po běhu konfiguračního módu. V tom případě se ale může stát, že program bude hledat soubory v nových adresářích, které nemusí existovat, pokud je uživatel nezadal přesně. Proto doporučujeme přidat před konfigurací do PortablerConfig.

Informace o vstupních registrech se po zpracování objeví v tabulkách RegistryKeysInfo a RegistryValuesDetails. Do databáze se přidají klíče, které jsou vždy definované, jedná se o `HKEY_CURRENT_USER`, `HKEY_LOCAL_MACHINE`, `HKEY_UCERS`, `HKEY_CLASSES_ROOT` a `HKEY_CURRENT_CONFIG`. Pro každý z nalezených klíčů registru se zavolá funkce `ProcessRegistryKey`. Ta rozparsuje vstup a vytvoří klíče, pokud neexistují. Potom otevře klíč pomocí API funkce `RegOpenKeyExW`, získá informace o klíči pomocí `RegQueryInfoKeyW` a získává informace o hodnotách pomocí `RegEnumValueW`, které následně ukládá do databáze. Pokud je nastaven rekurzivní mód, získává dále informace o podklíčích pomocí `RegEnumKeyExW` a rekurzivně volá funkci na každý podklíč. Pokud chce uživatel přidat svoje vlastní registrové klíče na zpracování, musí to udělat před konfigurací. Kostru funkce můžeme vidět v ukázce 4.5.

Program 4.5: Ukázka zpracování klíče registru z kódu, zde bez kontroly validity výsledků funkcí a bez deklarace proměnných.

```
void ProcessRegistryKey(HKEY hkey, const string& path, bool recursive, __int64
    parentId)
{
    __int64 initParentId = GetInitialParentId(scope);
    __int64 id = ConstructKeysInPath(scope, path, "", initParentId, false);
    LSTATUS status = RegOpenKeyExW(hkey, lpSubKeyA, 0, KEY_READ, &openedKey);
    if (status == ERROR_SUCCESS) {
        status = RegQueryInfoKeyW(openedKey, NULL, NULL, NULL, &subkeysCount, &
            maxSubkeySize, NULL,
            &valueCount, &maxNameLength, &maxValueLength, NULL, NULL);
        if (status == ERROR_SUCCESS) {
            for (unsigned int index = 0; index < valueCount; index++) {
                int status2 = RegEnumValueW(openedKey, index, nameBuffer.data(), &
                    charCountValueName,
                    NULL, &type, dataBuffer.data(), &charBytesData);
                if (status2 == ERROR_SUCCESS) {
                    InsertIntoRegistryValuesDetailsTable(id, keyName, type,
                        dataBuffer);
                }
            }
            if (recursive) {
                for (unsigned int i = 0; i < subkeysCount; i++) {
                    status = RegEnumKeyExW(openedKey, i, key, &key_size, NULL,
                        NULL, NULL, NULL);
                    string subkeyToProcess = path + "\\" + key;
                    ProcessRegistryKey(hkey, subkeyToProcess, recursive, id);
                }
            }
        }
    }
    RegCloseKey(openedKey);
}
```

4.11 Simulace registrů

V následující sekci popíšeme, jak jsme nasimulovali API pro práci s registrem.

4.11.1 Reprezentace registru

Data z registru perzistentně ukládáme v databázi. Informace o registrech jsme uložili ve 2 tabulkách — RegistryKeysInfo a RegistryValuesDetails. Tabulka RegistryKeysInfo obsahuje informace o jednotlivých klíčích registru, tabulka RegistryValuesDetails obsahuje informace o konkrétních hodnotách.

Pro identifikaci klíče registru API používá strukturu `HKEY`. `HKEY` je pointer na interní strukturu Windows, ke které ale nemáme přímý přístup. Můžeme o ní říci, že je to také „opaque pointer“ podobně jako `HANDLE`. S tímto pointerem se musíme v aplikaci vypořádat. Jeho definice v hlavičkových souborech je následující:

- `DECLARE_HANDLE(HKEY);`,
- `#define DECLARE_HANDLE(name) struct name##__{int unused;};`,
- `typedef struct name##__ *name`.

Struktura `HKEY` se inicializuje otevřením klíče registru, tedy voláním jedné z funkcí `RegOpenKey` nebo `RegCreateKey`. V kódu se dá `HKEY` také vytvořit pomocí příkazu `HKEY hkey = HKEY(int)`. Takový `HKEY` ale není plně inicializován a ve skutečnosti nereprezentuje žádný klíč registru. Ovšem je to `HKEY`, který můžeme předávat a pomocí kterého můžeme identifikovat klíče. Abychom měli informaci o tom, které klíče aplikace používá, použijeme v programu strukturu `map<HKEY,__int64> registryMapping`, která bude obsahovat informace o našich aktuálně otevřených klíčích. První hodnota je `HKEY`, která bude reprezentovat „otevřený“ klíč registru, druhá hodnota je 64 bitový integer jako ID klíče v databázi. Do této struktury se budou přidávat otevřené klíče a mazat zavírané klíče.

4.11.2 Nalezení klíčů v nové struktuře

Pro hledání klíčů jsme napsali přetíženou funkci `GetRegKeyInfoRecord(hkey)` a `GetRegKeyInfoRecord(hkey,subkey)`. Prvním parametrem je „hkey“ typu `HKEY`, druhý parametr „subkey“ je typu `std::string`. Jednu z těchto dvou verzí funkce využívají všechny další funkce pro nalezení klíče, se kterým poté dále pracujeme.

První verze má k dispozici pouze `HKEY`, který byl již dříve otevřen pomocí `RegCreateKey`. V takovém případě bude klíč v `registryMapping`, který se pokusíme najít. Pokud ho najdeme, podíváme se do databáze, jestli existuje klíč s ID shodným s druhou hodnotou v mapě. Pokud ano, vrátíme informace o tomto klíči. Pokud ne, vrátíme nevalidní hodnotu.

Druhá verze má k dispozici `HKEY` a `string`, který určuje o jaký podklíč klíče se jedná. Informace o klíčích jsou uloženy v databázi, funkce se tedy podívá do databáze, jestli v ní je záznam s touto cestou. Název klíče musí být unikátní, proto v databázi najdeme jeden nebo žádný záznam. Pokud najdeme právě jeden, vrátíme ho. Pokud nenajdeme, je možné, že podklíč nebude žádný a informaci zkusíme najít v `registryMapping` podobně, jako v předchozím případě. Pokud ho zde najdeme, vrátíme ho. Pokud ho nenajdeme, vrátíme nevalidní hodnotu.

Validní hodnoty v názvu klíče jsou pouze tisknutelné znaky, musí být alespoň jeden. Jako nevalidní hodnota tedy postačí taková, která bude mít alespoň jeden netisknutelný znak, který by ale neměl být binární nula. Volané funkce potom vrátí chybu `ERROR_FILE_NOT_FOUND`.

Každou operaci reprezentuje jedna funkce, která může být volána z vícero API funkcí. Tím dojde ke sjednocení minimálně na úrovni dvojice ANSI/UNICODE jedné funkce, v některých případech i více dvojic najednou. Některé funkce nemusí v novém API podporovat všechny parametry, z toho důvodu u každé funkce vypíšeme, které parametry má, a které nahrazuje. Všechny ostatní parametry naše implementace nepodporuje.

4.11.3 Vytvoření klíčů

Nová funkce `CallingRegCreateKeyInternal` provádí vytvoření nových klíčů v nově vytvořeném rozhraní registru. Má následující parametry:

- hkey typu `HKEY`,
- subkey typu `std::string`,
- phkey typu `PHKEY` (ukazatel na `HKEY`).

První argument „hkey“ může být buď předem definovaná hodnota nebo hodnota vrácená dřívějším voláním `RegOpenKey` nebo `RegCreateKey`, druhý argument „subkey“ reprezentuje cestu k podklíči, který se má vytvořit. Třetím parametr „phkey“ funguje jako výstupní parametr a vrátí v něm `HKEY` odkazující na nově vytvořený klíč. Nová operace vytvoření klíčů volající tuto funkci nahrazuje následující funkce:

- `RegCreateKeyA`, `RegCreateKeyW`,
- `RegCreateKeyExA`, `RegCreateKeyExW`,
- `CreateKeyTransactedA`, `CreateKeyTransactedW`.

Podle dokumentace, pokud klíč již existuje, tak se žádný nový nevytvoří, ale otevře se stávající. Nejprve se funkce podívá, jestli klíč v databázi již neexistuje. Pokud ano, vytvoří nový `HKEY`, který uloží do `registryMapping`, vrátí ho v třetím parametru a skončí.

Při vytváření nových klíčů postupně „ukrajuje“ z druhého parametru vždy název dalšího podklíče, který hledá pomocí `string::find_first_of('\')`. Pokud ho najde, přidá k současné nejdelší cestě nový podklíč, uloží ho do databáze a odstraní tuto část i se znakem '\' z cesty. Pokud už nenajde žádný oddělovač '\', jedná se o poslední podklíč. Ten také uloží do databáze, ale zároveň ho otevře a vrátí. V dokumentaci API funkcí se píše, že funkce vytvoří všechny chybějící klíče v dané cestě.

Podle struktury registru [7] můžeme během jednoho volání API vytvořit maximálně 32 úrovní klíčů. Funkce si proto pamatuje, kolik klíčů už vytvořila a skončí, pokud jich bude více jak 32. Vytvoření jednoho klíče v databázi znamená vytvoření nového záznamu v tabulce RegistryKeysInfo. Pokud se jich pokusí vytvořit více, vrátí chybu.

Pokud bude funkce úspěšná, vrací konstantu `ERROR_SUCCESS`. Klasické API funkce vrací v případě neúspěchu definovaný error code, který nesimulujeme. Rozeznáváme chybu `ERROR_FILE_NOT_FOUND`, která indikuje nenalezení klíče. Všechny ostatní selhání funkce indikujeme návratovou hodnotou -1.

4.11.4 Otevření klíče

Otevření klíče registru řeší funkce `CallingRegOpenKeyInternal` s parametry:

- hkey typu `HKEY`,
- subkey typu `std::string`,
- phkey typu `PHKEY` (ukazatel na `HKEY`).

První parametr „hkey“ je buď předem definovaná hodnota nebo hodnota vrácená dřívějším voláním `RegCreateKey` nebo `RegOpenKey`. Druhý parametr „subkey“ je cesta k podklíči, třetím parametrem je výstupní ukazatel, přes který se vrátí ukazatel na nově vytvořenou strukturu. Tato nová funkce nahradí API volání:

- `RegOpenKeyA`, `RegOpenKeyW`,
- `RegOpenKeyExA`, `RegOpenKeyExW`,
- `RegOpenKeyTransactedA`, `RegOpenKeyTransactedW`.

Otevření klíče je poměrně jednoduchá operace. Podíváme se do databáze, jestli daný klíč existuje. Pokud ano, tak vytvoříme nový `HKEY`, uložíme ho do `registryMapping` spolu s ID, vytvořený `HKEY` předáme na výstup a skončíme. V databázi jako takové nenastanou žádné změny. Počet možných otevřených klíčů je limitován rozsahem čísla, kterým inicializujeme `HKEY`. Každý nový klíč se inicializuje číslem, které se před každým novým otevřením inkrementuje. Ve 32 bitové verzi je to 32 bitové číslo, v 64 bitové verzi 64 bitové číslo. Proto může být najednou otevřených maximálně 2^{32}, resp. 2^{64}. Pokud se toto množství přesáhne a klíče se nebudou zavírat, může dojít k nekonzistenci a ztrátě informace o otevřených klíčích.

4.11.5 Mazání klíčů

O mazání klíčů se stará funkce `CallingRegDeleteKeyInternal` s parametry:

- hkey typu `HKEY`,
- subkey typu `std::string`.

První argument „hkey“ je buď předem definovaná hodnota nebo hodnota vrácená dřívějším voláním `RegCreateKey` nebo `RegOpenKey`. Druhý argument „subkey“ reprezentuje cestu k podklíči, který se má smazat. Funkce nahrazuje volání těchto API funkcí:

- `RegDeleteKeyA`, `RegDeleteKeyW`,
- `RegDeleteKeyExA`, `RegDeleteKeyExW`,
- `RegDeleteKeyTransactedA`, `RegDeleteKeyTransactedW`.

Dle dokumentace API funkcí pro všechny platí, že klíč ke smazání nesmí mít žádný podklíč. Nová funkce kontroluje, zda klíč nemá podklíče a dovolí smazání jen v případě, že žádný podklíč nemá. Funkce vrátí `ERROR_SUCCESS`, pokud se klíč podaří smazat. V opačném případě vrátí hodnotu -1. V dokumentaci je také psáno, že klíč se nemusí vždy smazat přímo voláním funkce `RegDeleteKey`, ale až poté, co se uzavřou všechny `HKEY` handle k němu.

Funkce se nejprve podívá, jestli klíč ke smazání vůbec existuje. Pokud ano, podívá se na počet podklíčů. Pokud nemá žádný podklíč, klíč je určen ke smazání. Klíč bude skutečně smazán, až nebude mít otevřený žádný `HKEY` handle. O zavírání `HKEY` se stará funkce `CallingRegCloseKey`, kterou popíšeme v sekci 4.11.14. Funkce se podívá, jestli má v registryMapping nějaký otevřený klíč. Pokud má alespoň jeden, označí se ke smazání přidáním do struktury `set<__int64> idsToDel`. Pokud nemá žádný otevřený `HKEY`, klíč se smaže z databáze.

Smazání klíčů v databázi znamená odstranění klíče podle ID z tabulky RegistryKeysInfo a všech hodnot z tabulky RegistryValuesDetails náležející pod tento klíč, tedy které mají KeyID stejný jako ID klíče.

4.11.6 Mazání hodnot

Nová funkce `CallingRegDeleteValueInternal` reprezentuje smazání hodnoty z klíče. Má tyto argumenty:

- hkey typu `HKEY`,
- subkey typu `std::string`,
- value typu `std::string`.

První parametr „hkey“ může být buď předem definovaná hodnota, nebo hodnota vrácená dřívějším voláním `RegOpenKey` nebo `RegCreateKey`. Druhý parametr „subkey“ uvádí cestu k podklíči otevřeného klíče a třetí parametr „value“ je jméno hodnoty ke smazání. Funkce nahrazuje následující API volání:

- `RegDeleteValueA`, `RegDeleteValueW`,
- `RegDeleteKeyValueA`, `RegDeleteKeyValueW`.

Funkce je velmi jednoduchá, pouze se podívá, zda zadaný klíč existuje. Pokud ano, tak smaže hodnotu. V databázi to znamená odstranění záznamu z RegValuesDetails podle jména. Funkce vrátí `ERROR_SUCCESS`, pokud je klíč v databázi, jinak vrátí -1.

4.11.7 Získání informace o klíči

Získání informace o klíči provádí funkce `CallingRegQueryInfoKey` zastupující API volání `RegQueryInfoKeyA` a `RegQueryInfoKeyW`. Poskytuje tyto informace o klíči, ostatní parametry funkce nepodporuje:

- `lpcSubKeys`, počet podklíčů,
- `lpcbMaxSubKeyLen`, délka nejdelšího jména podklíče,
- `lpcValues`, počet hodnot,
- `lpcbMaxValueNameLen`, délka nejdelšího jména hodnoty,
- `lpcbMaxValueLen`, velikost největší hodnoty.

Výstupní parametry funkce se pak nejčastěji používají pro enumerace klíčů a hodnot. Funkce si z databáze vyzvedne svoje podklíče, hodnoty a podle toho vrací na příslušná pole požadované hodnoty. V databázi neprobíhají žádné změny.

4.11.8 Enumerace klíčů a hodnot

Pro enumeraci klíčů a hodnot jsme připravili 3 nové náhradní funkce:

- `CallingRegEnumKey` pro `RegEnumKeyA` a `RegEnumKeyW`,
- `CallingRegEnumKeyEx` pro `RegEnumValueExA` a `RegEnumValueExW`,
- `CallingRegEnumValue` pro `RegEnumValueA` a `RegEnumValueW`.

Funkce s rozhraním pro ANSI i pro UNICODE vypadají stejně, na základě toho jsme chtěli předejít duplikaci kódu kvůli lepšímu testování a eliminaci chyb, proto nové funkce mají dva vstupní parametry jak typu `LPSTR` pro ANSI, tak typu `LPWSTR` pro UNICODE s tím, že nevyhovující parametr je nastaven na `NULL`.

`CallingRegEnumKey` a `CallingRegEnumKeyEx` najdou daný klíč v pořadí podle indexu a do odpovídajících parametrů nakopírují jméno posledního podklíče. Podobně `CallingRegEnumValue` najde odpovídající klíč a hodnotu na daném indexu, její název i její hodnotu potom nakopíruje do příslušných bufferů a nastaví příslušné délky. Seřazení hodnot tak, aby se při volání v cyklu žádná neopakovala a žádná nechyběla, zaručuje seřazení hodnot vrácených z databáze pomocí klauzule `ORDER BY ID ASC`.

4.11.9 Získání hodnot z klíče

Pro získání informací o klíči a jeho hodnotách jsme definovali 3 funkce, které nahrazují API volání:

- `CallingRegQueryValueInternal`,
- `CallingRegQueryValueExInternal`,
- `CallingRegGetValueInternal`.

Funkce `CallingRegQueryValueInternal` nahrazuje dvojici `RegQueryValueA` a `RegQueryValueW`, `CallingRegQueryValueExInternal` nahrazuje dvojici API `RegQueryValueExA` a `RegQueryValueExW`, `CallingRegGetValueInternal` nahrazuje `RegGetValueA` a `RegGetValueW`.

Funkce `CallingRegQueryValueInternal` slouží k získání defaultní hodnoty klíče registru. Tato hodnota je vždy typu `REG_SZ`. Pokud se aplikace pokusí uložit hodnotu jiného typu, skončí s chybou.

Funkce `CallingRegQueryValueExInternal` má podobný účel, může ale vyzvednout jakoukoli hodnotu jakéhokoli typu. Na vstupu dostane `LPBYTE`, což je ve Windows ukazatel na pole `unsigned char`, do kterého se nakopíruje požadovaná hodnota. Funkce přijímá pouze otevřené klíče. Pokud není buffer dostatečně velký, vrátí chybu `ERROR_MORE_DATA`.

Procedura `CallingRegGetValueInternal` také slouží k získání hodnoty z registru, podobně jako `CallingRegQueryValueExInternal`. Na vstupu dostane buffer, který nemusí být nutně typu `unsigned char*`, ale přijímá `PVOID`, což je ve Windows typ `void*`. V jednom parametru také specifikuje, který datový typ se snaží vyzvednout. Pokud se chtěný a skutečný datový typ liší, funkce skončí s chybou a vrátí -1.

Dle dokumentace, pokud se jedná o řetězcový typ, konkrétně jeden z typů `REG_SZ`, `REG_MULTI_SZ` nebo `REG_EXPAND_SZ`, funkce `RegGetValue` se postará o to, aby byl řetězec korektně ukončen ukončovací nulou nebo nulami. Pokud je řetězec uložen bez nich, funkce jimi řetězec ukončí. V bufferu musí být na případné ukončovací znaky místo. Pokud by nebyl buffer dostatečně velký nebo pokud by v bufferu nebylo místo pro případné ukončovací nuly, vrátí chybu `ERROR_MORE_DATA`.

4.11.10 Nastavení hodnot v klíči

Následující 3 funkce jsme definovali pro nastavování hodnot v klíči, společně s funkcemi, které nahrazují:

- `CallingRegSetKeyValueInternal`,
- `CallingRegSetValueInternal`,
- `CallingRegSetValueExInternal`.

Nová funkce `CallingRegSetKeyValueInternal` nahrazuje `RegSetKeyValueA` a `RegSetKeyValueW`, `CallingRegSetValueInternal` nahrazuje `RegSetValueA` a `RegSetValueW`, funkce `CallingRegSetValueExInternal` místo dvojice API `RegSetValueExA` a `RegSetValueExW`.

Funkce jsou podobné svým „protějškům“ ze sekce 4.11.9, mají podobné rozhraní, akorát místo vyzvedávání hodnoty do ní zapisují. Na základě toho se mění databáze, kdy se modifikují příslušné kolonky v tabulce RegistryValuesDetails.

Dvojice `CallingRegSetValueInternal`, `CallingRegSetValueExInternal` je podobná svým protějškům z „Get“ funkcí, mají stejný účel a podobné rozhraní, jen místo vyzvednutí hodnoty do ní naopak zapisují. Na základě toho se mění databáze, kdy se modifikují příslušné kolonky v tabulce RegistryValuesDetails.

Funkce `CallingRegSetValueInternal` nastavuje defaultní hodnotu klíče, `CallingRegSetValueExInternal` přepíše existující hodnotu nebo vytvoří novou pod zadaným jménem a `CallingRegSetKeyValueInternal` také vytvoří novou nebo přepíše existující. Jediný větší rozdíl mezi druhou a třetí vypsanou funkcí je ten, že první bere vstupní data typu `const BYTE*` a druhá typu `LPCVOID`. Každá zpracuje vstupní data, uloží je do databáze a dojde ke změně tabulky RegistryValuesDetails.

4.11.11 Kopírování stromu

Funkce `CallingRegCopyTree` nahrazuje API `RegCopyTreeA` a `RegCopyTreeW`. Nejprve najde oba podklíče — který bude zkopírovaný a do kterého se bude kopírovat. Pokud alespoň jeden z nich nenajde, vrátí `ERROR_FILE_NOT_FOUND`.

Nejprve zkopíruje hodnoty zdrojového klíče do cílového klíče. Potom rekurzivně, pro všechny poklíče zdrojového cíle, vytvoří nové klíče tím, že nahradí „prefix“ cesty z původního klíče novým podklíčem. Tím vznikne jméno zkopírovaného klíče registru, který se uloží do RegistryKeysInfo. Poté se nakopírují i všechny hodnoty. V databázi tedy přibudou nové klíče do tabulky RegistryKeysInfo i nové hodnoty do RegistryValuesDetails.

4.11.12 Mazání stromu

Dvojici funkcí `RegDeleteTreeA` a `RegDeleteTreeW` nahrazuje nová funkce `CallingRegDeleteTreeInternal`. Najde podklíč, poté rekurzivně hledá další jeho podklíče a od konce je postupně maže. V databázi se postupně mažou všechny podklíče daného klíče. Jelikož v dokumentaci není nic řečeno o nutnosti volání `RegCloseKey` na jednotlivé podklíče, jsou tyto podklíče mazány okamžitě. Kdyby byl klíč určen ke smazání, tedy byl přítomen ve struktuře `idsToDel`, bude z ní také smazán. Databáze smaže všechny klíče v RegistryKeysInfo i odpovídající hodnoty v RegistryValuesDetails.

4.11.13 Přejmenování klíče

Funkce `CallingRegRenameKey` nově reprezentuje původní funkci RegRenameKey. Funguje tak, že si z databáze vyzvedne všechny klíče, které mají tento klíč v „prefixu“, protože se musí změnit nejen vlastní podklíče, ale i podklíče všech podklíčů. Funkce není rekurzivní, z databáze si najednou vybere všechny hodnoty, které potřebuje změnit. V SQLite toho dosáhneme pomocí klauzule `Path LIKE ?`, kde znak „?“ nahradíme cestou ke vstupnímu klíči. Pro

každý nalezený klíč najdeme a nahradíme nové jméno za staré pomocí metod `string::find` a `string::replace`. V databázi proběhne změna jména ve sloupečku Path ve všech klíčích, které mají tento klíč v cestě jako prefix. Žádný klíč nebude přidán ani odebrán.

4.11.14 Uzavření klíče

Uzavření klíče provádí funkce `CallingRegCloseKey`, která nahrazuje původní `RegCloseKey`. Funkce vymaže handle `HKEY` z `registryMapping`, tím ho zavře. Mezitím si ještě ponechá jeho ID, které potom zkontroluje, jestli nebyl klíč označen na smazání přítomností ve struktuře `idsToDel`. Pokud ano, projde celý `registryMapping` a podívá se, jestli na něj ještě nějaký `HKEY` neodkazuje. Pokud již ne, tak z tabulek RegistryKeysInfo a RegistryValueDetails smaže informace o klíči a smaže i záznam v `idsToDel`. V případě, že je ještě nějaký `HKEY` na klíč otevřen, pouze se zavře a vrátí.

Zmíníme ještě funkci `CallingRegFlushKey` nahrazující originální volání `RegFlushKey`, která dostane `HKEY` a aktualizuje jeho data do registru. Jelikož nové rozhraní zapisuje do databáze rovnou bez prodlevy, tak funkce nemá žádnou implementaci a jen vrátí `ERROR_SUCCESS`.

4.11.15 Bezpečnost klíčů

Oprávnění ke klíčím v registru řídíme pomocí API funkcí `RegGetKeySecurity` a `RegSetKeySecurity`. Informace o oprávněních si předávají pomocí struktury `SECURITY_DESCRIPTOR`. V naší nové struktuře nejsou tyto deskriptory řešeny, ke všem klíčům je stejný přístup, lze ale omezit přístup k databázovému souboru „portabler.db“. V kódu existují nové funkce `CallingRegGetKeySecurity` a `CallingRegSetKeySecurity`, které se volají místo původních dvou funkcí, ale obě pouze vrátí úspěch a nic jiného nedělají.

4.11.16 Neimplementované funkce

Některé funkce z dokumentace [44] nejsou hookovány ani implementovány. Pokud je aplikace používá, jejich volání pravděpodobně skončí s chybou neexistujícího klíče nebo mohou mít nedefinované chování. Jde o funkce:

- GetSystemRegistryQuota, RegReplaceKey, RegConnectRegistry, RegDisablePredefinedCache, RegDisablePredefinedCacheEx, RegDisableReflectionKey, RegEnableReflectionKey, RegLoadKey, RegLoadMUIString, RegNotifyChangeKeyValue, RegOpenCurrentUser, RegOpenUserClassesRoot, RegOverridePredefKey, RegQueryMultipleValues, RegQueryReflectionKey, RegRestoreKey, RegSaveKey, RegSaveKeyEx, RegUnLoadKey.

Obecně jde o funkce, které nijak nemodifikují obsah registru jako takový. Například načítají nebo ukládají obsah registru do souboru, pracují s cache, s reflexemi nebo řeší připojení ke vzdálenému registru. Pokud tyto funkce neimplementujeme, není to takový problém, protože nijak nenaruší strukturu databáze a program bude fungovat dál. Případně nemusí některé funkčnosti využívající neimplementované API fungovat správně.

4.12 Simulace souborů

V následující sekci popíšeme, jak zaměňujeme volání Windows API pro práci se soubory.

4.12.1 Změna jména cílových souborů

Na začátek zmíníme, že na rozdíl od API pro registry zde nevytváříme žádné nové API, ale pouze „modifikujeme“ volání stávajícího tak, jak potřebujeme. V každé zachycené funkci se tedy volá i původní verze funkce. Měníme jen vstupní parametry a případně provádíme operace v databázi, pokud původní volání na modifikovaných parametrech skončí úspěšně.

Jména souborů měníme proto, že se snažíme přenášet případné soubory mimo instalační složku, které aplikace potřebuje. Používáme k tomu 2 přístupy, a to záměnu jména souboru jedna k jedné a záměnu „prefixu“ cesty k souboru.

Pracujeme vždy s absolutními cestami. K získání absolutní cesty k souboru jsme dříve použili funkci `GetAbsolutePathForFile`, která je svým způsobem wrapper pro volání API funkce `GetFullPathNameA`. V normálním případě by to nebyl problém, jenže v našem prostředí může být `GetFullPathNameA` zaměněna. Proto jsme napsali novou funkci `GetAbsolutePathForFileWrapper`, která se podívá do záznamu hookovaných funkcí, jestli nebyla funkce zaměněna. Pokud ne, tak zavolá API funkci. Pokud ano, tak zavolá funkci na adrese původní `GetFullPathNameA`, čímž zavolá nezmodifikovanou verzi.

Pro jednodušší práci s novými cestami a databází jsme připravili nové funkce:

- `string CheckFolderPrefixInternal(string)`,
- `string CheckFileMappingInternal(string)`,
- `string CheckFilenameInternal(string)`.

Než se začnou hledat nebo provádět změny v cestách, všechny 3 funkce nejdříve zkontrolují, zda se soubor v instalační složce již nenachází. Níže popsané funkčnosti se aplikují jen tehdy, pokud se soubor nachází mimo instalační složku.

Funkce `CheckFileMappingInternal` kontroluje tabulku FileMapping, zda se v ní nenachází cesta pro soubor zadaný v parametru. Pokud v databázi najde právě jeden záznam odpovídající této cestě (pokud hodnota sloupce

DefaultPath se rovná zadané cestě), vrátí ze stejného záznamu hodnotu NewPath. Pokud se žádný takový soubor nenajde, vrátí se původní cesta. Jedná se o funkci, která přímo přistupuje k datům z tabulky FileMapping a vrací vyhovující záznamy pro danou cestu.

Další funkce `CheckFolderPrefixInternal` porovnává cestu s tabulkou FolderPrefixMapping, kdy si funkce vezme všechny složky z tabulky a porovnává jejich cestu s cestou k souboru. Pokud je složka z tabulky prefixem k cestě a je delší než současně nalezený nejdelší prefix, uloží aktuální záznam jako nejdelší prefix. Pokud funkce nalezla alespoň jeden prefix, tak vezme ten nejdelší a nahradí celou cestu OriginalPath hodnotou UpdatedPath. Jelikož byla při předchozím hledání OriginalPath označena za prefix, tak se záměna povede a vrátí se modifikovaná cesta. Pokud se nepovede najít žádný prefix, vrátí se původní cesta. Jedná se o funkci, která přímo vybírá všechny cesty a hledá nejdelší možný prefix.

Kombinaci obou přístupů umožňuje funkce `CheckFilenameInternal`. Kontroluje jak záznamy z FileMapping, tak FolderPrefixMapping, pokud je třeba. Nejprve proběhne kontrola vůči tabulce FileMapping pomocí dříve popsané funkce `CheckFileMappingInternal`. Pokud byla nalezena nová cesta, vrátí ji. Pokud ne, zkontroluje prefixy z tabulky FolderPrefixMapping pomocí funkce `CheckFolderPrefixInternal`, jejíž výsledek funkce vrátí.

4.12.2 Vytváření souborů

Nové jméno pro nově vytvořené soubory zajišťuje nově vytvořená funkce `CreateFileInternal`. Funkce má jako argument cestu k původnímu souboru typu `string` a vrací novou cestu typu `string`. Pokud je cesta prefix instalační složky, tak se rovnou vrátí. Jinak zavolá `CheckFilenameInternal` a opět zkontroluje prefix s instalační složkou. Pokud změna cesty stále neproběhla, vytvoří novou cestu pomocí nové funkce `ConstructFilenameInternal` a tu vrátí. Zajišťuje tedy, že nová cesta bude uvnitř instalační složky.

Funkce `ConstructFilenameInternal` slouží k vytvoření nové cesty k souboru, pokud je potřeba. Funkce vezme jméno souboru, resp. poslední část jeho cesty. Tu „uřízne“ a složí novou cestu spojením tří částí: instalační složka, cesta k „movedfiles“ a jméno souboru. Kdyby náhodou už zadaný soubor existoval, přidá před jeho příponu, tedy před poslední tečku, znak „1“. To opakuje, dokud soubor s takovým jménem existuje. Funkce je volaná z následujících API volání:

- CreateDirectory, CreateDirectoryA, CreateDirectoryExA, CreateDirectoryExW, CreateDirectoryTransactedA, CreateDirectoryTransactedW, CreateDirectoryW, CreateFile2, CreateFileA, CreateFileTransactedA, CreateFileTransactedW, CreateFileW, LZOpenFileA, LZOpenFileW, OpenFile.

Pokud se operace vytvoření souboru povede, uloží se informace o vytvořeném souboru do databáze. Pokud jde o vytvoření souboru, uloží se do FileMapping. V případě složky se uloží jak do FileMapping, tak do FolderPrefixMapping.

4.12.3 Mazání souborů

Zajištění správné cesty souborů pro smazání má na starosti nová přetížená funkce `DeleteFileInternal` s rozhraním:

- `string DeleteFileInternal(LPCSTR),`
- `wstring DeleteFileInternal(LPCWSTR).`

Obě funkce ve své podstatě „jen“ volají `CheckFilenameInternal`, která pracuje s typem `string`. Proto se pro ni musí připravit parametr (pokud nevyhovuje) a potom vrátí jeho výsledek v požadovaném tvaru. Voláme na následujících API funkcích:

- DeleteFile, DeleteFileA, DeleteFileTransactedA, DeleteFileTransactedW, DeleteFileW, RemoveDirectoryA, RemoveDirectoryTransactedA, RemoveDirectoryTransactedW, RemoveDirectoryW.

Pokud se smazání souboru povede, smaže se v případě souboru záznam o něm z tabulky FileMapping, v případě složky z FileMapping i FolderPrefixMapping.

4.12.4 Kopírování souborů

Přesměrování kopírování souborů zajišťují nové funkce `CopyFileInternal` a `GetNameForCopiedFile` s rozhraním:

- `string CopyFileInternal(LPCSTR),`
- `wstring CopyFileInternal(LPCWSTR),`
- `string GetNameForCopiedFile(string, string).`

Obě verze funkce `CopyFileInternal` se starají o to, jaký soubor bude ve skutečnosti zkopírován. Zavolají `CheckFilenameInternal` a vrátí jeho výstup v požadovaném tvaru.

Druhá funkce `GetNameForCopiedFile` konstruuje jméno zkopírovaného souboru. Nejprve se podívá na nové jméno pomocí `CheckFilenameInternal`. Pokud nová cesta není uvnitř instalační složky, vezme poslední část ze jména souboru a vytvoří ho ve složce „movedfiles“. Potom se podívá, jestli se poslední jména obou souborů ve vstupním parametru rovnají. Pokud ne, nahradí staré poslední jméno novým, zkontroluje duplicity a vrátí novou cestu. Pokud se poslední jména rovnají, tak před koncovku nového jména přidá znak „c“, aby nové jméno odlišil. Poté zkontroluje duplicity a vrátí nové jméno. Funkci voláme v API funkcích:

- CopyFile, CopyFile2, CopyFileA, CopyFileExA, CopyFileExW, CopyFileTransactedA, CopyFileTransactedW, CopyFile.

Pokud se zkopírování souboru provede, tak se do tabulky FileMapping uloží nový záznam mezi původní cestou ke zkopírovanému souboru a novou cestou ke zkopírovanému souboru.

4.12.5 Přesouvání souborů

O konstrukci nových jmen pro přesunuté soubory se stará dvojice nových funkcí `MoveFileInternal` a `GetNewMovedFilename` s rozhraním:

- `string MoveFileInternal(LPCSTR)`,
- `wstring MoveFileInternal(LPCWSTR)`,
- `string GetNewMovedFilenameInternal(string, string)`.

Funkce `MoveFileInternal` pro nalezení jména prvního souboru je stejná jako `CopyFileInternal`. Také funkce `GetNewMovedFilenameInternal` je podobná funkci `GetNameForCopiedFile`. Zavolá `CheckFilenameInternal`, zkontroluje prefix a případně ho přesměruje do „movedfiles“, zkontroluje poslední jména vstupních souborů, a buď jej nahradí nebo přidá koncovku „m“, zkontroluje duplicity a vrátí nové jméno. Funkce se volá v těchto API volání:

- MoveFile, MoveFileA, MoveFileExA, MoveFileExW, MoveFileTransactedA, MoveFileTransactedW, MoveFileW, MoveFileWithProgressA, MoveFileWithProgressW.

Pokud funkce uspěje, vyberou se všechny záznamy o souborech, které mají jako novou cestu tu původně existující a pro každý takový záznam se změní nová cesta na tu přesunutou.

4.12.6 Informace o souborech

Do této skupiny jsme zařadili všechny funkce, které poskytují informace o souboru, například nastavení nebo zjišťování atributů souboru nebo práci s cestou. Mezi takové funkce patří tyto: (Všechny funkce pro ANSI i UNICODE a případně s příponou Ex nebo Transacted, pokud existují.)

- GetCompressedFileSize, GetFileAttributes, GetFullPathName, GetFullPathName, SetFileAttributes.

Pro všechny funkce platí, že před zavoláním jejich verze se zkontroluje, jestli není soubor v mapování pomocí `CheckFilenameInternal`. Původní funkce se zavolá na výstupní cestě k souboru.

4.12.7 FindFirstFile

Vytvořili jsme funkci `FindFileInternal(string)`, která určuje nové jméno, pomocí kterého bude aplikace prohledávat. Zavolá `CheckFilenameInternal` a vrátí jeho výstup. Funkce je volána pro změnu jména v následujících API funkcích:

- FindFirstFileA, FindFirstFileExA, FindFirstFileExW, FindFirstFileTransactedA, FindFirstFileTransactedW, FindFirstFileW.

4.12.8 Neřešené operace se soubory

Jiné operace se soubory, například funkce ReadFile nebo WriteFile, ReadFile nebo LockFile neberou jako argumenty cestu k souboru, ale jejich `HANDLE`. Podobně jako `HKEY` u registru, `HANDLE` je opaque pointer, jehož obsah je schovaný. Více o handlech v Microsoft dokumentaci [46]. Tyto `HANDLE` jsou vytvořeny například pomocí `CreateFile` nebo jí podobné. Když tedy dokážeme zaměnit jméno souboru v CreateFile, tak na něj bude odkazovat i jeho `HANDLE` a nemusíme je dále řešit.

4.13 Postup spuštění aplikace

Aby mohla aplikace běžet v přenositelném módu, je nejdůležitější ji nejprve do přenosného módu dostat. Proto je potřeba setup, který shrneme do 5 níže uvedených kroků, které hned detailněji popíšeme:

1. vybrat aplikaci pro přenesení,
2. spustit aplikaci ve skenovacím režimu,
3. provést konfiguraci,
4. zabalit aplikaci a přenést,
5. spustit aplikaci v přenosném režimu.

Předpokladem je, že aplikace je v počítači již nainstalována. Není to nutná podmínka, skenovací režim Portableru lze spustit dvěma způsoby. Aplikace počítá se složkou Portabler na stejné úrovni jako spustitelný soubor, který bude spouštět. Můžeme spustit novou instanci aplikace pomocí SCANNEW nebo se injektovat do existující instance pomocí SCANEXISTING. K tomu doporučujeme upravit a použít jeden z připravených skriptů „ScanExisting.bat“ nebo „ScanNew.bat“. Modifikace je potřeba kvůli správnému zadání argumentů — „ScanNew.bat“ potřebuje v parametru zadat cestu ke spustitelnému souboru, „ScanExisting.bat“ potřebuje jako parametr PID běžícího procesu.

Výstup skenovacího režimu vytvoří databázi a připraví data ve vstupní tabulce PortablerConfig. Ta se dá vyplnit dvěma způsoby, a to ručně nebo automaticky. Doporučená je kombinace obou přístupů s tím, že nejprve doporučujeme spustit aplikaci ve sledovacím režimu, který tabulku vyprázdní

a uloží do ní nově nalezené hodnoty. Tabulku poté důrazně doporučujeme zkontrolovat a doupravit. K tomu můžeme použít jakýkoli browser pro SQLite, jeden je například k dispozici zde [50]. Pomocí něj lze otevřít soubor „portabler.db“ a provést požadované úpravy v PortablerConfig. Do sloupce TargetValue budeme přidávat relativní cesty vzhledem ke složce „movedfiles“.

Až bude tabulka PortablerConfig připravena dle přání uživatele, bude potřeba provést konfiguraci, tedy zpracování dat ve vstupní tabulce. Konfiguraci provádí režimy CONFIG a PORTABLER_CONFIG. Režim CONFIG provede konfiguraci a skončí, režim PORTABLERCONFIG provede konfiguraci a rovnou spustí aplikaci v přenosném režimu. Konfigurace předem smaže předchozí obsah všech tabulek vyjma PortablerConfig, kterou nechá, a tak lze v případě neúspěchu nebo špatného vyplnění opravit chyby a pustit konfiguraci znovu. Konfiguraci musíme spustit z počítače, ve kterém je aplikace nainstalována. Spuštěním konfiguračního režimu na jiném počítači povede ke smazání dat v ostatních tabulkách. Velmi pravděpodobně se ztratí informace o registrech, kterou již nebudeme mít jak dostat zpátky, dokud aplikaci nepřeneseme zpátky na původní počítač a nespustíme ji tam. Doporučujeme spustit jeden z připravených skriptů „Config.bat“ nebo „PortablerConfig.bat“.

Aplikaci poté můžeme „zabalit“, přenést a spustit v přenosném režimu. K tomu slouží zejména režim PORTABLER. Také lze využít režim PORTABLERCONFIG, který ovšem předtím provede konfiguaci, jak jsme popsali o odstavec výše. Režim PORTABLERCONFIG je určen na jedno prvotní použití, pro opakované spuštění je doporučen režim PORTABLER. Na to doporučujeme spustit odpovídající připravený skript „Portabler.bat“.

4.14 Nutná kontrola uživatelem

Aplikace sice přímo nevyžaduje interakci uživatele (kromě 2× spuštění aplikace), ovšem kontrola uživatele je důrazně doporučena po běhu učících režimů. Jak kontrola části registru, kterou je třeba nasimulovat do dalších tabulek, tak souborových vstupů, zejména kontrola prefixů.

U registru je rizikový zejména režim SCANEXISTING. Jeden z hlavních zdrojů jsou funkce otevírající klíč registru. V případě, že se program napojí až v průběhu, může se stát, že takto přijdeme o některé vstupy do registru. Také se může stát, že při běžném použití aplikace nenavštíví všechny klíče. Další riziko je nesprávné vyhodnocení módu simulace.

Pro souborový systém je důležité zkontrolovat, zda jsou všechny soubory a složky správně překopírovány. Jejich cesta musí odpovídat mapování v databázi, jinak sice složka bude v Portableru, ale nebude využívána. Je možné, že ne všechny soubory a složky bude vhodné mapovat, to se může stát zejména u programů pracujících s filesystémem, příkladem je vstup pro `FindFirstFile`. Ten může používat zástupné znaky „*“ a „?“, v tom případě by se nejednalo o cestu k žádné konkrétní složce nebo souboru, a tak nebude namapován dále.

Kapitola 5

Testování

V této kapitole popíšeme, jak probíhalo testování aplikace.

5.1 Testování DLL

Pro testování a ladění důležitých funkcí PortablerLibrary jsme použili projekt „DllTester“, který umožňuje spouštět (i v debug módu) funkce uvnitř testovaného DLL. Jelikož budeme testovat i volání čistě interních funkcí, museli jsme PortablerLibrary upravit tak, aby umělo exportovat i interní funkce tak, aby se daly přímo volat. Pokud nebudou exportované, tak linker nebude moci najít žádnou konkrétní implementaci, protože ta by jinak zůstala skyta v DLL.

Aplikace slouží k odladění předtím, než budeme testovat funkčnost na skutečných aplikacích a dále k lepšímu porozumění případných problémů. Testovat budeme pouze knihovnu PortablerLibrary. Knihovnu ScanLibrary nebudeme testovat, protože jen sleduje a zpracovává informace z původního volání API.

Správné nastavení projektů umožní debug funkcí v DLL. Stačí mít aktuální verzi DLL v kódu. Potom můžeme v PortablerLibrary přidávat breakpointy, krokovat, sledovat proměnné apod., jako kdybychom debugovali normální program. Návod, jak nastavit debugování DLL můžeme najít v Microsoft dokumentaci [51]. My jsme použili debugování pomocí volací aplikace.

Testování skutečného scénáře znamená, že budeme volat klasické API funkce s tím, že budeme očekávat volání našich upravených funkcí. Budeme tak potřebovat hooknout původní API funkce a nahradit je našimi novými. Jelikož má ale DllTester nastavenou závislost na PortablerLibrary, tak ji automaticky načte a PortablerLibrary tak provede hook všech funkcí při načítání a nemusíme se o něj v kódu jako takovém starat a můžeme hned volat API funkce a sledovat výsledky.

5.2 Průběh testování

Testování knihovny PortablerLibrary jsme provedli ve 3 fázích, které v sekcích níže popíšeme podrobněji:

1. testování interních funkcí,
2. testování v reálném prostředí,
3. testování přenesení.

5.3 Testování interních funkcí

Nejprve jsme testovali a ladili volání interních registrových funkcí. Pro registry jsme připravili soubor pro volání testovaných funkcí „RegistryTest.cpp“, který volá funkce v PortablerLibrary a měří čas, jak dlouho funkce běží v milisekundách. Poté v souboru „InternalRegistryTest.cpp“ provádíme konkrétní testování připravených funkcí z „RegistryTest.cpp“. Největší důraz klademe na testování funkcí zajišťujících vytváření klíčů, ukládání hodnot a vybírání hodnot. Funkce testujeme opakovaným voláním a kontrolováním hodnot.

Poté jsme testovali také vybrané interní souborové funkce v souboru „FilesystemTest.cpp“, který podobně jako u registru, volá interní funkce z PortablerLibrary a měří jim čas běhu v milisekundách. Konkrétní testování těchto funkcí potom probíhá v souboru „InternalFilesystemTest.cpp“. Testujeme jednotlivé operace v různých situacích a sledujeme výsledky.

Po opravě chyb a testování se přesuneme na další fázi, a to testování v reálném prostředí.

5.4 Testování v reálném prostředí

Na rozdíl od testů interních funkcí jsme potřebovali zjistit, jak se implementace bude chovat v „reálném prostředí“. Reálné prostředí bude aplikace, která bude klasicky volat API funkce. Místo API funkcí se ale budou volat naše funkce a můžeme tak postupně sledovat, jestli volání API funkcí v programu nenaruší jeho chování a jestli bude aplikace skutečně pracovat s naší databází. Vytvořili jsme několik scénářů, které volají posloupnost API funkcí a budou sledovat jejich chování.

Při testování máme zároveň otevřenou databázi a kontrolujeme, jestli v databázi nastanou očekávané změny. Aplikace musí mít nejen správný výstup v kódu, ale v závislosti na operaci by se měly provést požadované změny v databázi.

Tento test je důležitý proto, abychom ověřili, jak se bude implementace chovat při „opravdovém“ volání API. Jelikož interní funkce již máme otestované, předpokládáme správné volání interní funkce a kontrolujeme především, zda je interní funkce volána správně a očekávaně, tedy jestli se interní funkce volá se správnými parametry a nedochází k chybám např. při převodu.

5.5 Testování přenesení

Poslední fází testování je samotné testování na aplikacích, které se budeme snažit přenést pomocí přenositelného módu. Aplikace pro přenesení budou nainstalovány na jednom počítači, na kterém se z nich pomocí naší aplikace pokusíme udělat přenositelné programy.

Testovat budeme několik aplikací a budeme sledovat jejich chování. Je pravděpodobné, že minimálně u některých aplikací dojde k potížím a Portabler na ně nebude fungovat. Pokud se nám aplikaci nepodaří spustit v učícím módu s injektovanou ScanLibrary, nebudeme aplikaci dále testovat na přenášení.

Po naučení pomocí ScanLibrary a po provedení konfigurace zkopírujeme instalační složku s programem na přenosný disk a zkusíme aplikaci spustit na jiném počítači.

Pro testování jsme vybrali následující aplikace:

- 7-Zip[19],
- WinRAR[20],
- HxD[21],
- Visual Studio Code[22],
- Mozilla Firefox[23],
- Notepad++[24].

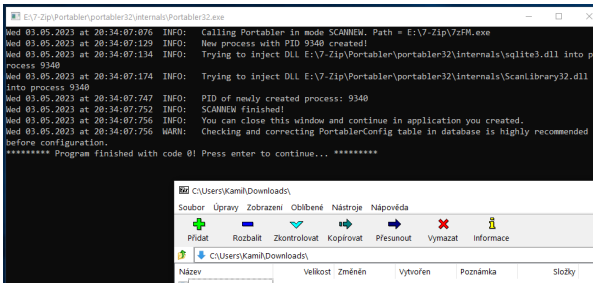
5.5.1 Testování 7-Zip

První testování proběhlo na aplikaci 7-Zip, konkrétně na její 32 bitové verzi. Program byl již na počítači nainstalovaný, tak jsme vzali složku Portabler, kterou jsme nakopírovali do instalační složky se 7-Zip programem. Našli jsme skript „ScanNew32.bat“, který jsme upravili tak, aby měl v parametru cestu ke spustitelnému souboru „7zFM.exe“. Jak spuštění dopadlo, můžeme vidět na obrázku 5.1.

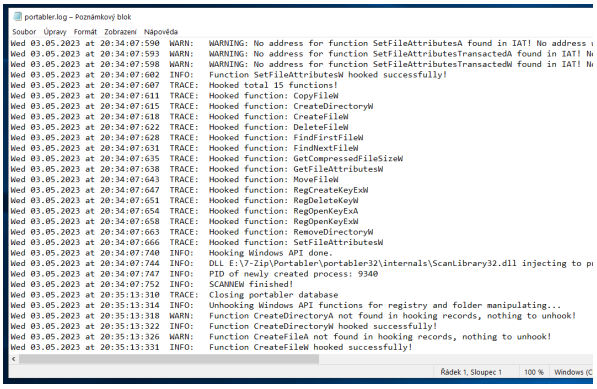
Aplikaci se nám spustit podařilo a zkusili jsme následující průchod: vybrali jsme 2 soubory a přidali je do archivu, kde jsme změnili některá nastavení. Následně jsme využili několik funkčností, nechali jsme vytvořený archiv zkontrolovat a následně jsme ho smazali. Poté jsme aplikaci zavřeli. Obsah tabulky PortablerConfig můžeme vidět na obrázku 5.2, část logu se seznamem hookovaných funkcí na obrázku 5.3.

Otevřeli jsem databázi a našli jsme několik cest, ale pouze ze složky, ve které jsme prováděli změny, nebudeme tedy chtít žádnou mapovat. Klíče registru jsme našli 3. Zastavíme se u klíče `HKEY_CURRENT_USER\Software\7-Zip`,

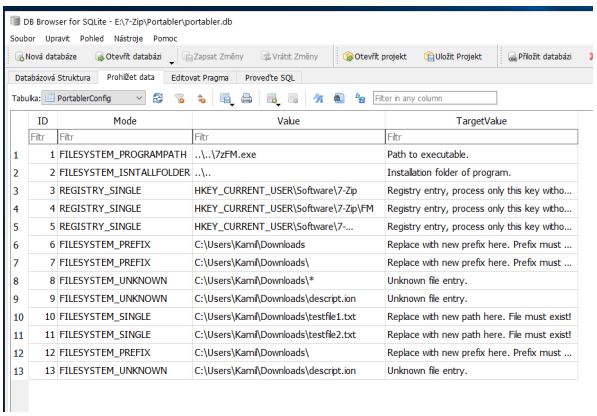
[19] `https://www.7-zip.org/`
[20] `https://www.win-rar.com/start.html?&L=17`
[21] `https://mh-nexus.de/en/hxd/`
[22] `https://code.visualstudio.com/`
[23] `https://www.mozilla.org/cs/firefox/new/`
[24] `https://notepad-plus-plus.org/`



Obrázek 5.1: Spuštění testovaného programu 7-Zip pomocí módu ScanNew.

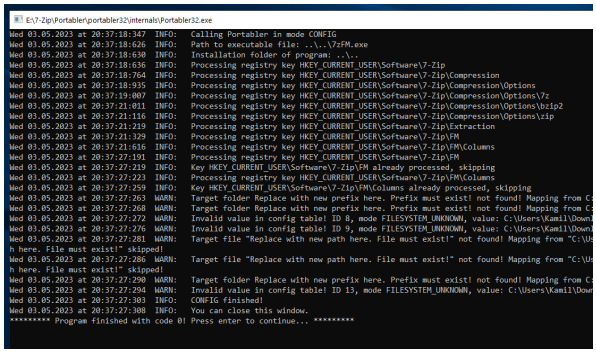


Obrázek 5.2: Obsah tabulky PortablerConfig po vypnutí aplikace 7-Zip.



Obrázek 5.3: Ukázka logu z běhu programu 7-Zip, část s výpisem hookovaných funkcí.

kterému změníme mód zpracování na rekurzivní. Ostatní nalezené klíče jsou jeho podklíče, tím je také rekurzivně zpracujeme a žádný klíč nevynecháme. Následně jsme provedli konfiguraci, která uspěla a nasimulovala klíče registru aplikace. Výstup konfigurace můžeme vidět na obrázku 5.4.



Obrázek 5.4: Ukázka konfigurace programu 7-Zip.

Aplikaci se nám pomocí skriptu „Portabler32.bat“ podařilo spustit i s knihovnou PortablerLibrary, podařilo se nám i zopakovat posloupnost akcí z učícího módu. V logu se můžeme podívat, že je naše API skutečně používáno.

5.5.2 Testování WinRAR

Podobně jako testování na programu 7-Zip proběhlo testování na WinRARu a jeho 64 bitové verzi. Program je také na výchozím počítači nainstalovaný, stačilo tedy, abychom složku s Portablerem vložili do instalační složky se spustitelným souborem WinRARu a můžeme přejít k testování. Spustili jsme „ScanNew64.bat“ s parametrem pro cestu ke spustitelnému souboru.

Jelikož jsou si oba programy funkcionalitami podobné, testovali jsme podobný průchod. Tedy vzali jsme dva testovací soubory, vytvořili nový archiv, nechali si ho zkontrolovat, smazali ho a zavřeli jsme aplikaci.

Po nahlédnutí do databáze jsme našli mnohem více klíčů registru, než kolik jich bylo u 7-Zip. Zajímají nás `HKEY_CURRENT_USER\Software\WinRAR`, `HKEY_CLASSES_ROOT\WinRAR` a `HKEY_LOCAL_MACHINE\Software\WinRAR`. Zde se bude jednat o „kořenové klíče“, které budeme chtít zpracovat rekurzivně. Tyto klíče už program rozpoznal, že má spouštět v rekurzivním módu, tak ho necháme. I ostatní podklíče klíče se pokusí zpracovat rekurzivně, ale jelikož si Portabler pamatuje, které klíče již zpracoval, tak je přeskočí. Kromě navštívených složek a použitých souborů jsme našli použité složky v AppData, rozhodli jsme se tedy vytvořit mapování pro AppData, kde vytvoříme složku AppData v „movedfiles“ a přidáme mapování mezi „movedfiles\AppData“ a původní složkou AppData a označíme ji jako prefix.

Poté spustíme konfiguraci. Konfigurace trvá výrazně déle než u 7-Zipu, protože dochází ke zpracování většího množství dat. Zatímco u 7-Zipu jsme

zpracovali 10 klíčů a asi 30 hodnot, u WinRARu jsme zpracovali 80 klíčů a přes 500 hodnot.

Následné spuštění v přenositelném módu sice proběhlo, aplikace se spustí, ale běží velmi pomalu. Z logu jsme zjistili, že aplikace často pracuje s registrem, jen po pár akcích v aplikaci již aplikace otevřela více jak 600 klíčů. Asi proto je aplikace pod novým API citelně pomalejší. Navíc, když jsme chtěli vytvořit nový archiv, většinou se nepodařilo a skončili jsme s chybou. Také jsme narazili na problém, kdy aplikace vyžadovala přítomnost knihovny `VCRUNTIME140_1.dll`. Tuto knihovnu jsme museli stáhnout a přidat do složky se spustitelnými skripty do „portabler64“. Poté aplikace spustit šla.

5.5.3 Testování HxD

Aplikaci jsme neměli na původním počítači nainstalovanou a chtěli jsme otestovat, jak se bude Portabler, respektive jeho učící mód chovat při instalaci aplikace. Stáhli jsme si tedy instalační soubor a spustili ho pomocí „ScanNew32.bat“. Zjistili jsme, že při instalaci se program podíval do 4 klíčů registru a dočasné soubory si ukládal do Temp složky.

Bohužel se nám do HxD nedaří nainjektovat knihovnu ani pomocí SCANNEW, ani SCANEXISTING, ani PORTABLER. Dostáváme chybu 5, tedy `ERROR_ACCESS_DENIED`, i když proces otevíráme s nízkými požadavky (kombinaci práva pro čtení, zápis, vytváření vláken a alokování místa, ty potřebujeme kvůli funkcím pro DLL injection).

5.5.4 Testování VS Code

Další aplikací, kterou chceme otestovat, je Visual Studio Code. Zkoušíme injektovat rovnou ScanLibrary pomocí SCANNEW a podařilo se nám. Vytvořili jsme novou složku, ve které jsme zkusili vytvořit pár souborů s krátkým obsahem, zkusili si pohrát s nějakými výběry, vyhledáváním a jednoduchým nastavením, poté jsme aplikaci zavřeli.

Pohledem do tabulky PortablerConfig jsme zjistili, že jsme nedokázali zachytit žádný klíč registru, který by VS Code používal. Aplikaci jsme spouštěli pomocí SCANNEW. Pohledem do logu jsme zjistili, že ani nebyly zaměněny žádné registrové funkce, i když se o to aplikace pokoušela. Záznamů o použitých souborech je hodně, pro nás je zajímavá hlavně naše složka se soubory, kterou jsme vytvořili. Také jsme našli mnoho záznamů v AppData.

Abychom mohli naší složku přenést, zkopírovali jsme jí do „movedfiles“ a přidali cestu do PortablerConfig k původní cestě. Podobný postup jsme aplikovali pro AppData. Po provedení konfigurace máme ve FolderPrefixMapping naše 2 záznamy.

V přenositelném módu už se nám aplikaci spustit nepodařilo. Pohledem do logu jsme zjistili, že proces se spustil, PortablerLibrary se injektovalo, ale také se hned odpojilo a vytvořený proces jsme již nenašli ani v Task Manageru.

5.5.5 Testování Firefox

Zkusili jsme testovat aplikaci Mozilla Firefox. U něj jsme narazili hned na několik problémů. Spuštění v módu SCANNEW se povedlo, ale námi vytvoření proces téměř hned skončí. Po otevření task manageru jsme v seznamu procesů našli 8 různých procesů s názvem „firefox.exe“, ale ani jeden z nich neměl stejné PID, jako námi vytvořený proces. PID námi vytvořeného procesu vidíme v konzoli a neshoduje se s žádným PID firefox procesů v Task Manageru. Log nám prozradil, že se povedlo hooknout pouze funkce `CreateFileA`, `CreateFileW` a `DeleteFileW`. Zkoušeli jsme na vytvořené procesy režim SCANEXISTING, ale bez většího úspěchu. Ani zde se nám nepodařilo zjistit dostatečné množství informací pro přenesení, maximálně tak instalační složku v Program Files a AppData, ve kterém se nachází zmíněný soubor. Firefox tedy dále také testovat nebudeme.

5.5.6 Testování Notepad++

Poslední testovanou aplikací je Notepad++. Skenování dopadlo v pořádku, našli jsme soubory v AppData, a žádnou práci s registrem. Po vytvoření nové složky pro AppData a upravení záznamu v PortablerConfig provedeme konfiguraci, po které najdeme ve FolderPrefixMapping naše mapování. Jediný problém, na který jsme narazili, bylo původní mapování AppData. AppData je umístěna v cestě, která zároveň obsahuje jméno profilu uživatele. Notepad++ k přístupu do AppData bere cestu do AppData toho uživatele, který je zrovna přihlášen, a ne uživatele z původního počítače. Museli jsme tedy v mapování změnit v cestě jméno uživatelů, poté aplikace fungovala v přenositelném módu dobře. Podobně jako ostatní 64 bitové aplikace, i tato vyžadovala knihovnu `VCRUNTIME140_1.dll`.

5.6 Výsledky testování

Testování interních funkcí proběhlo v pořádku, v průběhu testování jsme nalezli některé chyby, které jsme odstranili. Podobně testování v „reálném prostředí“ proběhlo v pořádku.

Z výsledků testování interních funkcí vyšlo najevo, že operace vytváření klíčů je v průměrném případě pomalejší oproti ostatním funkcím, řádově ve stovkách milisekund, v málo případech běžela i přes sekundu. Souborové funkce běží v řádu desítek milisekund.

Testování na reálných aplikacích ukázalo, že aplikace má omezené použití. Předpokládá, že do aplikace půjdou injektovat knihovny, a že aplikace používá pro práci se soubory i registrem Windows API. Aplikace, které používají jiná rozhraní, se nemusí chovat očekávaně.

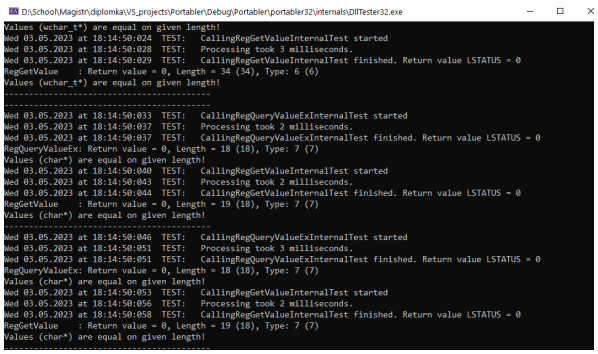
Například Firefox a HxD nám vůbec nedovolili injekci DLL do svých procesů. U HxD se injekce nepovede z důvodu, že ačkoli otevíráme proces s nízkými oprávněními, jsou podle HxD stále příliš velká.

Aplikace WinRAR a Visual Studio Code nám povolily nainjektovat ScanLibrary i PortablerLibrary, ale se spuštěním v přenositelném módu už byl problém. WinRAR byl v přenositelném módu pomalý a operace vytvoření archivu se ne vždy podařila. Visual Studio Code sice spustilo proces, PortablerLibrary se nainjektovalo, ale po velmi krátké době se DLL odpojilo a proces skončil.

Naproti tomu, aplikace 7-Zip a Notepad++ fungovaly i v přenositelném módu docela dobře. Posloupnost akcí se nám podařilo zopakovat. Aplikace sice byly trochu pomalejší než jejich původní verze, ale použít se daly. Zatímco archiv vytvořený v přeneseném 7-Zip byl vytvořen ve složce, ve které jsem jej vytvořil, vytvořený textový soubor v Notepad++ se vždy vytvořil v „movedfiles“. To znamená, že zatímco 7-Zip nepoužil k vytvoření souboru jednu z vybraných API funkcí, tak Notepad++ ano.

Dalším zjištěním pro nás bylo, že testované 64 bitové aplikace vyžadovaly přítomnost knihovny `VCRUNTIME140_1.dll`. Jde o knihovnu, kterou vyžadují aplikace napsané ve Visual C++. Knihovnu jsme museli stáhnout a vložit do složky, ze které jsme spouštěli skripty, aby program mohl fungovat. U 32 bitových aplikací jsme problém s chybějící knihovnou neměli. Proto přidáme toto DLL do přílohy, kdyby na cílovém počítači chybělo.

Na obrázku 5.5 můžeme vidět záznam z testování, konkrétně z testování interních funkcí registru. Záměrem testu bylo vytvořit testovací klíče, vložit data do jednoho z klíčů a následně data vyzvednout.



Obrázek 5.5: Ukázka z DllTesteru při testování interních funkcí registru, konkrétně testování vyzvednutí námi uložených hodnot.

Kapitola 6

Diskuze

V této kapitole zhodnotíme zjištěné výsledky, chování implementace a prodiskutujeme reálnou použitelnost v praxi.

6.1 Výsledky

Z testování vyšlo najevo, že aplikace má v současné podobě omezené využití. Ne všechny testované aplikace se podařilo správně přenést do přenosného módu a ne všechny přenesené aplikace se v něm podařilo následně správně spustit. Ačkoli při testování funkcí jsme dosáhli očekávaných výsledků, při testování reálných aplikací jsme narazili na několik úskalí. Pokud byly ale splněny předpoklady používání Windows API a aplikace dovolila načíst DLL, pak se jí přenést podařilo.

6.2 Použitelnost

Portabler je vhodný pro přenos aplikací, které využívají Windows API a dovolují injekci kódu pomocí DLL. Pokud aplikace splňuje tento předpoklad, lze ji pomocí Portableru přenést z jednoho počítače na druhý.

Použitelnost programu je omezená. Spoléhá na to, že aplikace používají Windows API pro přístup k registru a Windows API pro práci se soubory. Zatímco u registrů se o takový problém nejedná, tak pro práci se soubory existuje v C++ více rozhraní, které mohou pracovat se soubory, a které nejsou zpracovány.

Při testování na reálných aplikacích jsme nezjistili žádný větší rozdíl při využití učícího módu na již nainstalovanou aplikaci oproti prvotnímu instalování aplikace. Aplikace nemusí všechny svoje budoucí zdroje předpřipravovat při instalaci. Vhodnější je proto použít učící mód na již nainstalovanou aplikaci, případně použít kombinaci obou přístupů.

6.3 Omezující podmínky

Jednou z omezujících podmínek je nevyužití některých parametrů API volání, zejména u nového API pro registr. Může tím být omezena některá funkcionalita, kterou mohou programy vyžadovat nebo jí využívat. API je omezené zejména na správu dat.

Dalším možným omezením je i výběr API, které sledujeme a nahrazujeme. Naše řešení spoléhá na to, že aplikace využívá čistě Windows API, které nahrazujeme. Cílová aplikace ale nemusí čistě Windows API používat, může používat i jiné metody přístupu k filesystému. V C++ můžeme do souborů zapisovat pomocí například `std::fstream` nebo `std::filesystem`, se kterými ale naše aplikace nepracuje a může docházet k nekonzistencím nebo nemožnosti konzistentního přenesení naší aplikací.

Dalším jistým omezením je důrazně doporučená interakce uživatele. Ačkoli aplikace poskytuje skenovací mód, který uživateli může pomoci při rozhodování, které všechny klíče registru nebo soubory aplikace používá, neměl by na ní uživatel plně spoléhat bez kontroly.

U registru může být problém zejména, pokud aplikace špatně vyhodnotí rekurzivitu nebo ne. Při testování jsme například u WinRaRu narazili na situaci, kdy přistupoval do jednoho klíče a využíval jen ten jeden klíč, ale ne jeho podklíče, kterých měl ovšem velké množství. Aplikace sice všechny podklíče nasimuluje, to ale může být časově náročné a klíče se nemusí nikdy použít.

U souborového systému může dojít k problému, pokud bude aplikace přenesena a nedostane všechny potřebné soubory. V takovém případě může dojít k neočekávánému chování přenesené aplikace. Tuto chybu lze opravit tak, že aplikaci poskytneme požadovanou složku, která se ale nachází na původním počítači, na kterém byl program původně nainstalován.

Další omezovací podmínka je počet otevřených klíčů registru, kterých může být maximálně 2^{32}, aby byla zaručena správnost. Pokud aplikace přesáhne tuto hodnotu, začnou se do `registryMapping` znovu ukládat `HKEY`, které již byly použity. Pokud klíče nebyly dříve smazány, může dojít k nekonzistenci a chybě.

Skenování nemusí najít úplně všechny hodnoty registru nebo složky, se kterými aplikace pracuje. Důležité je, jaký průchod aplikací uživatel zvolí, když je aplikace skenována. Pokud skenování nenajde nějakou důležitou složku, se kterou program pracuje a nepřidá se manuálně, může se program při jiném průchodu začít chovat neočekávaně.

Některé aplikace se také mohou bránit DLL injection. Nám se například stalo, že některé programy naše DLL buď vůbec nenačetly, nebo je načetly a skoro ihned odpojily, nebo původní aplikace spustila další procesy a skončila.

6.4 Jiné možnosti řešení

Aplikace se dala implementovat s různými přístupy. Náš přístup je zaměřený na to, že se aplikace spustí v jednom z předem připravených módů a na základě něj aplikaci buď sleduje, konfiguruje nebo spouští přenosně. Diskutovali jsme možnosti ukládání a zpracování dat z přenositelného módu, nakonec jsme vybrali možnost s přechodným režimem konfigurace, kdy má uživatel větší kontrolu nad zpracovávanými daty.

6.5 Možnosti rozšiřování

Jako hlavní možnosti rozšíření vidíme možnost implementace dalších funkcí pro nahrazování. Například pro registry lze implementovat API pro práci s registrovými soubory. Pro soubory by mohlo jít o přidání dalších funkcí, které mohou být ovlivněny novou implementací.

U souborů můžeme řešit nahrazování dalších funkcí, které jsme zatím nenahrazovali. Můžeme se například zaměřit na jiná rozhraní pracující se soubory, nejen Windows API.

6.6 Zabezpečení a přístup k datům

Například na implementaci bezpečnosti klíčů pomocí API `RegGetKeySecurity` a `RegSetKeySecurity` jsme nedávali důraz. Ačkoli jde o důležitou část, tak aplikace dovolila uživateli nasimulovat klíče, ke kterým on sám v tu chvíli už měl přístup nebo na ně získal oprávnění. Kdyby se mu nepovedlo ke klíči přistoupit, nemohl by ho simulovat.

Podobně u souborů, pokud uživatel potřeboval předtím zvýšená práva na zkopírování souboru nebo složky, nebude je pravděpodobně v nové implementaci používat. Zabezpečení nových dat přímo souvisí s přístupem k nové struktuře, zejména ke složce Portabler a databázovému souboru „portabler.db“ a ke složce s přesunutými soubory.

Závěr

V rešeršní části jsme se zabývali přenositelnými programy. Zjišťovali jsme, jak se liší od instalovaných programů, co je dělá přenositelnými a jaká jsou existující řešení nebo možnosti přenášení. V další části jsme se zabývali technikami reverzního inženýrství, představili jsme několik technik, které pomáhají analyzovat chování programu nebo zasahovat do jeho činnosti.

Následně jsme vytvořili koncept aplikace, která by mohla instalované aplikace spouštět v přenosném režimu pomocí simulace části registru a přesunu složek a souborů, které aplikace využívá a nejsou v její instalační složce. Diskutovali jsme možnosti, jak co nejvíce efektivně a nejlépe získat pro aplikaci vstup. Nakonec jsme vybrali řešení, které se nám jevilo jako nejlepší kombinace náročnosti na používání a efektivity.

V další části jsme provedli implementaci konceptu v jazyce Visual C++. Napsali jsme aplikaci, která může běžet v několika režimech. Umí se napojit na běžící aplikaci nebo rozběhnout novou instanci pro sledování jejího chování, následně umí upravený vstup zpracovat a spustit aplikaci v přenositelném módu.

Poté jsme provedli testování aplikace. Tu jsme otestovali po částech, nejprve výkonné funkce, poté reálný scénář a na skutečných aplikacích. Testování interních funkcí proběhlo v pořádku, testování v reálném prostředí také. U reálných aplikací jsme při testování narazili na několik omezení.

V závěrečné diskuzi jsme se zamysleli nad možnými důvody, proč má Portabler jen omezené využití na reálných aplikací. Hlavním problémem je spoléhání se na používání výhradně Windows API pro práci s registrem a soubory. Také jsme zmínili možnosti rozšíření aplikace.

V příloze můžeme najít jak zdrojové kódy, tak testované aplikace nebo jejich konfigurační složky. Také tam můžeme najít soubory s vybranými funkcemi, se kterými jsme pracovali a další přílohy.

Literatura

[1] WAHAB, Fatima. What Is The Difference Between Portable & Installable Apps?. *AddictiveTips* [online]. 2015-12-30 [cit. 2023-03-20]. Dostupné z: https://www.addictivetips.com/windows-tips/what-is-the-difference-between-portable-installable-apps/

[2] MCDONALD, Johanna. Portable application. *TechTarget* [online]. TechTarget, ©2010-2023, 2023-02 [cit. 2023-03-20]. Dostupné z: https://www.techtarget.com/searchcloudcomputing/definition/portable-app

[3] BRUSH, Kate a Brian KIRSCH. Virtualization. *TechTarget* [online]. TechTarget, ©2016 - 2023, 2021-10 [cit. 2023-03-20]. Dostupné z: https://www.techtarget.com/searchitoperations/definition/virtualization

[4] AWATI, Rahul a Margaret JONES. Application sandboxing. *TechTarget* [online]. TechTarget, ©2003 - 2023, 2022-02 [cit. 2023-03-20]. Dostupné z: https://www.techtarget.com/searchmobilecomputing/definition/application-sandboxing

[5] Emulation. *TechTarget* [online]. TechTarget, ©1999 - 2023, 2006-09 [cit. 2023-03-20]. Dostupné z: https://www.techtarget.com/whatis/definition/emulation

[6] Registry. *Microsoft Learn* [online]. Microsoft, ©2023 [cit. 2023-03-22]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/sysinfo/registry

[7] Structure of the Registry. *Microsoft Learn* [online]. Microsoft, ©2023 [cit. 2023-03-22]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/sysinfo/structure-of-the-registry

[8] Opening, Creating, and Closing Keys. *Microsoft Learn* [online]. Microsoft, ©2023 [cit. 2023-03-22]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/sysinfo/opening-creating-and-closing-keys

[9] Predefined Keys. *Microsoft Learn* [online]. Microsoft, ©2023 [cit. 2023-03-22]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/sysinfo/predefined-keys

[10] Registry value types. *Microsoft Learn* [online]. Microsoft, ©2023 [cit. 2023-03-22]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/sysinfo/registry-value-types

[11] Registry Hives. *Microsoft Learn* [online]. Microsoft, ©2023 [cit. 2023-03-22]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/sysinfo/registry-hives

[12] Registry Key Security and Access Rights. *Microsoft Learn* [online]. Microsoft, ©2023 [cit. 2023-03-22]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/sysinfo/registry-key-security-and-access-rights

[13] Local File Systems. *Microsoft Learn* [online]. Microsoft, ©2023 [cit. 2023-03-23]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/FileIO/file-systems

[14] File Management (Local File Systems). *Microsoft Learn* [online]. Microsoft, ©2023 [cit. 2023-03-23]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/FileIO/file-management

[15] Directory Management. *Microsoft Learn* [online]. Microsoft, ©2023 [cit. 2023-03-23]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/FileIO/directory-management

[16] Volume Management. *Microsoft Learn* [online]. Microsoft, ©2023 [cit. 2023-03-23]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/FileIO/volume-management

[17] Disk Management. *Microsoft Learn* [online]. Microsoft, ©2023 [cit. 2023-03-23]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/FileIO/disk-management

[18] LUTKEVICH, Ben. Reverse-engineering. *TechTarget* [online]. TechTarget, ©2006 - 2023, 2021-06 [cit. 2023-03-18]. Dostupné z: https://www.techtarget.com/searchsoftwarequality/definition/reverse-engineering

[19] DAULAGUPHU, Satyajit. A Comprehensive Guide To PE Structure. *Tech Zealots* [online]. ©2022, 2022-08-15 [cit. 2023-03-19]. Dostupné z: https://tech-zealots.com/malware-analysis/pe-portable-executable-structure-malware-analysis-part-2/

[20] An In-Depth Look into the Win32 Portable Executable File Format. *Microsoft Learn* [online]. Microsoft, ©2023, 2008-02-26 [cit. 2023-03-19]. Dostupné z: https://learn.microsoft.com/en-us/previous-versions/bb985992(v=msdn.10)

[21] IMAGE_FILE_HEADER structure. *Microsoft Learn* [online]. Microsoft, ©2023, 2021-04-02 [cit. 2023-03-19]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_file_header

[22] IMAGE_SECTION_HEADER structure. *Microsoft Learn* [online]. Microsoft, ©2023 [cit. 2023-03-19]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_section_header

[23] PE File structure: Sections. *Keystrokes* [online]. 2016, 2016-06-03 [cit. 2023-03-19]. Dostupné z: https://keystrokes2016.wordpress.com/2016/06/03/pe-file-structure-sections/

[24] PE Format. Microsoft Learn [online]. Microsoft, ©2023 [cit. 2023-03-19]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format

[25] Statically placing hooks on PE file's IAT. *Reverse Engineering* [online]. Stack Exchange, ©2023 [cit. 2023-03-20]. Dostupné z: https://reverseengineering.stackexchange.com/questions/31233/statically-placing-hooks-on-pe-files-iat

[26] PE Format: Import Address Table. *Microsoft Learn* [online]. Microsoft, ©2023 [cit. 2023-03-20]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format#import-address-table

[27] Dynamic link library (DLL). *TechTarget* [online]. TechTarget, ©2000 - 2023, 2021-11 [cit. 2023-03-20]. Dostupné z: https://www.techtarget.com/searchwindowsserver/definition/dynamic-link-library-DLL

[28] Dependency hell. *TechTarget* [online]. TechTarget, ©2016 - 2023, 2021-10 [cit. 2023-03-20]. Dostupné z: https://www.techtarget.com/searchitoperations/definition/dependency-hell

[29] GILLIS, Alexander S. Static analysis (static code analysis). *TechTarget* [online]. TechTarget, ©1999 - 2023, 2020-07 [cit. 2023-03-17]. Dostupné z: https://www.techtarget.com/whatis/definition/static-analysis-static-code-analysis

[30] What Is Dynamic Analysis?. *TotalView* [online]. Perforce Software, ©2023, 2020-07-10 [cit. 2023-03-17]. Dostupné z: https://totalview.io/blog/what-dynamic-analysis

[31] TAGADE, Kanishk. Dynamic Code Analysis: A Primer. *Security Boulevard* [online]. Techstrong Group, ©2023 [cit. 2023-03-17]. Dostupné z: https://securityboulevard.com/2021/02/dynamic-code-analysis-a-primer/

[32] QUINN, Sam. Function Hooking for Recon and Exploitation. *Musarubra US LLC* [online]. 2022, 2022-04 [cit. 2023-03-17]. Dostupné z: https://www.trellix.com/en-us/assets/docs/atr-library/tr-function-hooking-for-recon-and-exploitation.pdf

[33] Windows API Hooking. *Red Team Notes* [online]. [cit. 2023-03-17]. Dostupné z: https://www.ired.team/offensive-security/code-injection-process-injection/how-to-hook-windows-api-using-c++

[34] Import Adress Table (IAT) Hooking. *Red Team Notes* [online]. [cit. 2023-03-17]. Dostupné z: https://www.ired.team/offensive-security/code-injection-process-injection/import-adress-table-iat-hooking

[35] HENKART, Quest. *Debugging: The Power of Reverse Engineering* [online]. 2015, 2015-04-15 [cit. 2023-03-16]. Dostupné z: https://questhenkart.medium.com/debugging-the-power-of-reverse-engineering-d659214da3a

[36] HEUSSER, Matt. Debugging. *TechTarget* [online]. TechTarget, ©2006 - 2023, 2022-11 [cit. 2023-03-16]. Dostupné z: https://www.techtarget.com/searchsoftwarequality/definition/debugging

[37] AYMAN, David. Anti-Debugging Techniques. *Medium* [online]. 2021, 2021-08-03 [cit. 2023-03-16]. Dostupné z: https://medium.com/@X3non_C0der/anti-debugging-techniques-eda1868e0503

[38] Process Injection. *MITRE ATT&CK* [online]. The MITRE Corporation, ©2015-2023 [cit. 2023-03-14]. Dostupné z: https://attack.mitre.org/techniques/T1055/

[39] HOSSEINI, Ashkan. Ten process injection techniques. *Elastic* [online]. Elasticsearch B.V., ©2023 [cit. 2023-03-14]. Dostupné z: https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process

[40] E LARES S, David. Malware Sandboxing and Reverse engineering overview. *Medium* [online]. 2022, 2022-01-21 [cit. 2023-03-16]. Dostupné z: https://medium.com/nerd-for-tech/malware-sandboxing-and-reverse-engineering-overview-66f6ad84e8c9

[41] LUTKEVICH, Ben. Obfuscation. *TechTarget* [online]. TechTarget, ©2000 - 2023, 2021-04 [cit. 2023-03-16]. Dostupné z: https://www.techtarget.com/searchsecurity/definition/obfuscation

[42] SCHWARZ, Benjamin, Saumya DEBRAY a Gregory ANDREWS. *Disassembly of Executable Code Revisited* [online]. Department of Computer Science, University of Arizona [cit. 2023-03-16]. Dostupné z: https://www2.cs.arizona.edu/~debray/Publications/disasm.pdf

[43] VAN EMMERIK, Mike. The Decompilation Process. *Program-Transformation.Org: The Program Transformation Wiki* [online]. 2005-02-21 [cit. 2023-03-16]. Dostupné z: https://www.program-transformation.org/Transform/DecompilationProcess.html

[44] Registry Functions. *Microsoft Learn* [online]. Microsoft, ©2023, 2022-05-04 [cit. 2023-04-01]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/sysinfo/registry-functions

[45] File Management Functions. *Microsoft Learn* [online]. Microsoft, ©2023, 2022-04-13 [cit. 2023-04-01]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/fileio/file-management-functions

[46] Handles and objects. *Microsoft Learn* [online]. Microsoft, ©2023, 2022-02-08 [cit. 2023-04-01]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/sysinfo/handles-and-objects

[47] Opaque Pointer in C++. *GeeksForGeeks* [online]. GeeksForGeeks [cit. 2023-04-01]. Dostupné z: https://www.geeksforgeeks.org/opaque-pointer-in-cpp/

[48] SQLite Home Page [online]. 2023-03-22 [cit. 2023-04-02]. Dostupné z: https://www.sqlite.org/index.html

[49] Prepared Statement Object. *SQLite* [online]. [cit. 2023-04-02]. Dostupné z: https://www.sqlite.org/c3ref/stmt.html

[50] DB Browser for SQLite [online]. [cit. 2023-04-11]. Dostupné z: https://sqlitebrowser.org/

[51] Debug DLLs in Visual Studio. *Microsoft Learn* [online]. Microsoft, ©2023 [cit. 2023-04-14]. Dostupné z: https://learn.microsoft.com/en-us/visualstudio/debugger/debugging-dll-projects?view=vs-2022

Příloha A

Seznam použitých zkratek

API Application programming interface

ACL Access control lists

ANSI American national standards institute

DLL Dynamic link library

DOS Disk operating system

I/O Input/output

IAT Import address table

ID Identifier

MS-DOS Microsoft disk operating system

NT New technology

OS Operating system

PE Portable executable

PID Process identifier

RE Reverse engineering

SQL Structured query language

USB Universal serial bus

Příloha **B**

Funkce sledované ve ScanLibrary

CopyFile, CopyFile2, CopyFileA, CopyFileExA, CopyFileExW, CopyFileTransactedA, CopyFileTransactedW, CopyFileW, CreateDirectory, CreateDirectoryA, CreateDirectoryExA, CreateDirectoryExW, CreateDirectoryTransactedA, CreateDirectoryTransactedW, CreateDirectoryW, CreateFile2, CreateFileA, CreateFileTransactedA, CreateFileTransactedW, CreateFileW, DeleteFile, DeleteFileA, DeleteFileTransactedA, DeleteFileTransactedW, DeleteFileW, FindFirstFileA, FindFirstFileExA, FindFirstFileExW, FindFirstFileTransactedA, FindFirstFileTransactedW, FindFirstFileW, FindNextFileA, FindNextFileW, GetCompressedFileSizeA, GetCompressedFileSizeTransactedA, GetCompressedFileSizeTransactedW, GetCompressedFileSizeW, GetFileAttributesA, GetFileAttributesExA, GetFileAttributesExW, GetFileAttributesTransactedA, GetFileAttributesTransactedW, GetFileAttributesW, GetFullPathNameA, GetFullPathNameTransactedA, GetFullPathNameTransactedW, GetFullPathNameW, HFILE OpenFile, LZOpenFileA, LZOpenFileW, MoveFile, MoveFileA, MoveFileExA, MoveFileExW, MoveFileTransactedA, MoveFileTransactedW, MoveFileW, MoveFileWithProgressA, MoveFileWithProgressW, RegCopyTreeA, RegCreateKeyA, RegCreateKeyExA, RegCreateKeyExW, RegCreateKeyTransactedA, RegCreateKeyTransactedW, RegCreateKeyW, RegDeleteKeyA, RegDeleteKeyExA, RegDeleteKeyExW, RegDeleteKeyTransactedA, RegDeleteKeyTransactedW, RegDeleteKeyValueA, RegDeleteKeyValueW, RegDeleteKeyW, RegDeleteTreeA, RegDeleteTreeW, RegGetValueA, RegGetValueW, RegOpenKeyA, RegOpenKeyExA, RegOpenKeyExW, RegOpenKeyTransactedA, RegOpenKeyTransactedW, RegOpenKeyW, RegQueryValueA, RegQueryValueW, RegSetKeyValueA, RegSetKeyValueW, RegSetValueA, RegSetValueW, RemoveDirectoryA, RemoveDirectoryTransactedA, RemoveDirectoryTransactedW, RemoveDirectoryW, SetFileAttributesA, SetFileAttributesTransactedA, SetFileAttributesTransactedW, SetFileAttributesW

Příloha C

Funkce simulované v PortablerLibrary

CopyFile, CopyFile2, CopyFileA, CopyFileExA, CopyFileExW, CopyFileTransactedA, CopyFileTransactedW, CopyFileW, CreateDirectory, CreateDirectoryA, CreateDirectoryExA, CreateDirectoryExW, CreateDirectoryTransactedA, CreateDirectoryTransactedW, CreateDirectoryW, CreateFile2, CreateFileA, CreateFileTransactedA, CreateFileTransactedW, CreateFileW, DeleteFile, DeleteFileA, DeleteFileTransactedA, DeleteFileTransactedW, DeleteFileW, FindFirstFileA, FindFirstFileExA, FindFirstFileExW, FindFirstFileTransactedA, FindFirstFileTransactedW, FindFirstFileW, FindNextFileA, FindNextFileW, GetCompressedFileSizeA, GetCompressedFileSizeTransactedA, GetCompressedFileSizeTransactedW, GetCompressedFileSizeW, GetFileAttributesA, GetFileAttributesExA, GetFileAttributesExW, GetFileAttributesTransactedA, GetFileAttributesTransactedW, GetFileAttributesW, GetFullPathNameA, GetFullPathNameTransactedA, GetFullPathNameTransactedW, GetFullPathNameW, HFILE OpenFile, LZOpenFileA, LZOpenFileW, MoveFile, MoveFileA, MoveFileExA, MoveFileExW, MoveFileTransactedA, MoveFileTransactedW, MoveFileW, MoveFileWithProgressA, MoveFileWithProgressW, RegCloseKey, RegCopyTreeA, RegCopyTreeW, RegCreateKeyA, RegCreateKeyExA, RegCreateKeyExW, RegCreateKeyTransactedA, RegCreateKeyTransactedW, RegCreateKeyW, RegDeleteKeyA, RegDeleteKeyExA, RegDeleteKeyExW, RegDeleteKeyTransactedA, RegDeleteKeyTransactedW, RegDeleteKeyValueA, RegDeleteKeyValueW, RegDeleteKeyW, RegDeleteTreeA, RegDeleteTreeW, RegDeleteValueA, RegDeleteValueW, RegEnumKeyA, RegEnumKeyExA, RegEnumKeyExW, RegEnumKeyW, RegEnumValueA, RegEnumValueW, RegFlushKey, RegGetKeySecurity, RegGetValueA, RegGetValueW, RegOpenKeyA, RegOpenKeyExA, RegOpenKeyExW, RegOpenKeyTransactedA, RegOpenKeyTransactedW, RegOpenKeyW, RegQueryInfoKeyA, RegQueryInfoKeyW, RegQueryValueA, RegQueryValueExA, RegQueryValueExW, RegQueryValueW, RegRenameKey, RegSetKey-

Security, RegSetKeyValueA, RegSetKeyValueW, RegSetValueA, RegSetValueExA, RegSetValueExW, RegSetValueW, RemoveDirectoryA, RemoveDirectoryTransactedA, RemoveDirectoryTransactedW, RemoveDirectoryW, SetFileAttributesA, SetFileAttributesTransactedA, SetFileAttributesTransactedW, SetFileAttributesW

Příloha D

Obsah přiloženého média

- `readme.txt` ... stručný popis obsahu média
- `exe` ... adresář s připravenou formou programu
  - `Portabler` ... konfigurační složka připravena k použití
  - `guide.txt` ... průvodce spuštěním
- `src`
  - `codes` ... zdrojové kódy aplikace
  - `hooks` ... seznamy hookovaných funkcí
  - `tests` ... testované aplikace
- `text` ... text práce
  - `thesis.pdf` ... text práce ve formátu PDF
  - `source` ... zdrojová forma práce ve formátu LaTeX