VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

UNIVERZÁLNÍ NÁSTROJ PRO DEKOMPRESI SPUSTITELNÝCH SOUBORŮ

GENERIC UNPACKER OF EXECUTABLE FILES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MAREK MILKOVIČ
AUTHOR

VEDOUCÍ PRÁCE — Ing. PETER MATULA
SUPERVISOR

BRNO 2015

Abstrakt

Komprese spustitelných souborů je proces komprese dat a kódu za účelem zmenšení velikosti nebo ochrany těchto souborů. Chování komprimovaného spustitelného souboru je těžké analyzovat, proto musí být takovýto soubor nejprve dekomprimován. Tato práce pojednává o návrhu a implementaci univerzálního nástroje pro dekompresi spustitelných souborů, neboli generického unpackeru, který je součást dekompilačního procesu v rekonfigurovatelném zpětném překladači společnosti AVG. Cílem této práce je vytvořit generický unpacker, který by byl jednoduše rozšířitelný, platformě a architekturálně nezávislý a jeho výstup by byl dekompilovatelný. V rámci práce jsou navrhnuté a implementované heuristické analýzy pro dosažení dekompilovatelnosti výstupů. Výsledky jsou porovnatelné s unpackery používanými v praxi.

Abstract

Executable files packing is a process used for compression or protection of these files. The behavior of the packed executable file is difficult to analyze, therefore the packed file needs to be unpacked at first. This work deals with the design and implementation of a generic unpacker that is part of the decompilation chain in the AVG's Retargetable Decompiler. The goal of this work is to create the generic unpacker of executable files, which would be easily extensible, platform and architecture independent, and its ouput would be decompilable. The heuristic analyses are proposed and implemented to achieve the decompilability of the ouput. The results are comparable with the other unpackers used in practice.

Klíčová slova

dekompilce, reverzní inženýrství, rekonfigurovatelný zpětný překladač, dekomprese, spustitelný soubor

Keywords

decompilation, reverse engineering, retargetable decompiler, unpacking, executable file

Citace

Marek Milkovič: Univerzální nástroj pro dekompresi spustitelných souborů, bakalářská práce, Brno, FIT VUT v Brně, 2015

Univerzální nástroj pro dekompresi spustitelných souborů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Petra Matuly. Řešení se mnou také konzultovali Ing. Jakub Křoustek a Ing. Petr Zemek. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Marek Milkovič
19. května 2015

Poděkování

Rád bych poděkoval svému vedoucímu Ing. Petru Matulovi za odborné rady, vedení a čas, který mi věnoval. Také bych chtěl poděkovat Ing. Jakubu Křoustkovi a Ing. Petru Zemkovi za konzultace, odborné rady a zpětnou vazbu.

Obsah

Kapitola 1

Úvod

Reverzné alebo spätné inžinierstvo je proces získavania znalostí o zostrojenom objekte pričom je dekonštruovaný natoľko, že je odhalená jeho vnútorná štruktúra, návrh a architektúra [9]. V oblasti informačných technológií sa budeme venovať softwarovému reverznému inžinierstvu, ktorého objektom skúmania je software.

Pri tvorbe softwaru sa využívajú nástroje ako sú prekladače—kompilátory. Ich úlohou je zápis programu v programovacom jazyku previesť na zápis v strojovom kóde, teda postupnosť inštrukcií, ktorú počítač bude vykonávať. Pokiaľ chceme skúmať software, jeho vnútornú štruktúru a správanie, pričom pôvodný zdrojový kód daného softwaru nie je dostupný, znamená to len skúmať jeho preloženú podobu vo forme strojového kódu.

Preto jedným z dôležitých pilierov softwarového reverzného inžinierstva je proces nazývaný dekompilácia—spätný preklad. Je to proces pri ktorom dochádza k spätnému zostrojeniu zápisu programu vo vyššom programovacom jazyku zo strojového kódu. Spätný preklad však nie je vždy možný vykonať tak, aby sme sa dostali k pôvodnému zdrojovému kódu. Autor softwaru mohol pri jeho tvorbe využiť techniky, ktoré tento proces robia zložitejším, ako je napríklad kompresia spustiteľného súboru—packing pomocou nástroju nazývaného packer. Tento postup je veľmi častý u tzv. malware, alebo aj *malicious software*, teda škodlivý software. Uvádza sa, že 80% až 90% malwaru je v dnešnej dobe skomprimovaných [12].

Kompresia spätný preklad takmer absolútne znemožňuje. Pokiaľ chceme takto upravený spustiteľný súbor dekompilovať, musíme najskôr vykonať dekompresiu spustiteľného súboru, nazývanú aj unpacking. Dekompresiu štandardne vykonáva samotný program pri svojom spustení, avšak deje sa buď len v pamäti alebo dočasne na disku. Po vypnutí programu je spustiteľný súbor naďalej skomprimovaný. Existujú však nástroje nazývané unpackery, ktoré dokážu dekomprimovať komprimovaný spustiteľný súbor. Vo väčšine prípadov sa však jedná o jednoúčelové unpackery, ktoré fungujú len na konkrétny druh packeru a konkrétnu verziu. Mnoho z nich taktiež pri dekompresií spúšťa komprimovaný súbor, čo prináša veľa obmedzení a bezpečnostných rizík. Mnoho packerov sa taktiež sústredí len na zachovanie spustiteľnosti dekomprimovaného súboru, nie na maximálnu možnú rekonštrukciu štruktúry súboru. Táto skutočnosť spôsobuje, že aj napriek úspešnej dekompresií nie je niekedy možné dekomprimovaný súbor dekompilovať spätným prekladačom.

Cieľom tejto práce je navrhnúť a vytvoriť generický unpacker. Takýto unpacker by vedel na základe zdetekovaného použitého packeru aplikovať mechanizmy, ktoré by viedli k dekompresií softwaru. Taktiež by bol jednoducho rozšíriteľný pre veľkú škálu packerov. Motiváciou je tiež zaručiť bezpečnosť pred skomprimovaným malwarom počas dekompresie jeho nespúšťaním a umožniť dekompresiu nezávislé na architektúre a platforme, na ktorej generický unpacker bude spustený a pre ktorú bude komprimovaný súbor určený. Generický

unpacker by sa mal tiež sústrediť na maximálnu rekonštrukciu štruktúry dekomprimovaného spustiteľného súboru. Výstup generického unpackeru by mal byť v tom prípade vždy dekompilovateľný.

Práca je členená do viac kapitol, ktoré podrobnejšie rozoberajú problematiku. V kapitole 2 je popísaný všeobecný princíp fungovania spätných prekladačov a podrobnejšie je rozobraný rekonfigurovateľný spätný prekladač spoločnosti AVG. Kapitola 3 je venovaná spustiteľným súborom a ich formátom. Presnejšie sa jedná o formáty PE a ELF, ktoré patria medzi najpoužívanejšie a preto sú terčom packerov. Kapitola 4 popisuje kompresiu a dekompresiu spustiteľných súborov, aké ochrany proti reverznému inžinierstvu sa používajú a aj spomína konkrétne existujúce generické unpackery. Kapitola 5 rozoberá analýzu 2 konkrétnych existujúcich packerov, MPRESS a UPX. Návrh samotného generického unpackeru a taktiež zásuvných modulov pre analyzované packery sa nachádza v kapitole 6. Implementácia generického unpackeru spolu s implementáciou zásuvných modulov je popísaná v kapitole 7. Jednotlivé zásuvné moduly sú testované na sade spustiteľných súborov. Špecifikáciu testov a ich výsledky sa nachádzajú v kapitole 8. Posledná kapitola 9 zhodnocuje výsledky práce a navrhuje možnosti ďalšieho vývoja generického unpackeru.

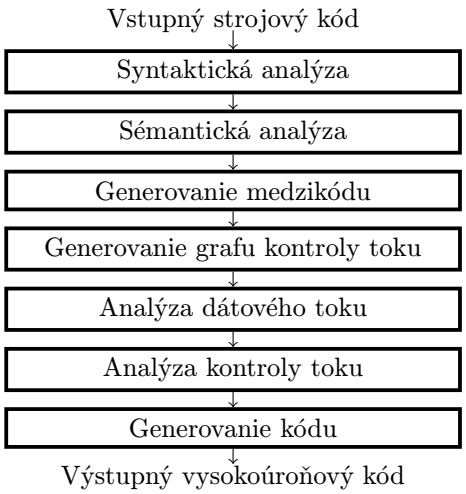
Kapitola 2

Spätný prekladač

Táto kapitola čerpá z [8], [11], [12] a [9].

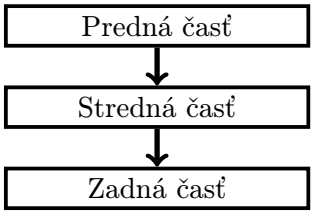
Spätný prekladač (angl. *decompiler*) je nástroj, ktorý číta nízkoúrovňový kód zapísaný v strojovom jazyku (vstupný jazyk) a prekladá ho na ekvivalentný program vo vysokoúrovňovom jazyku (výstupný jazyk) [8]. Tento proces sa nazýva spätný preklad. Spätný sa nazýva z dôvodu, že vykonáva obrátenú činnosť prekladača, ktorého snahou je z vysokoúrovňového jazyku preloziť program do nížkoúrovňovej reprezentácie. Proces spätného prekladu je však náročnejší v porovnaní s obyčajným prekladom. Pri preklade má prekladač dostupných podstatne viac informácií, ktoré sa v priebehu prekladu môžu vynechať, pretože na výslednú činnosť programu nebudú mať žiaden vplyv. Do tejto skupiny patria napríklad komentáre, makrá, direktívy, dátové typy, názvy premenných a funkcií atď. Prekladač taktiež dopĺňa do programu množstvo vlastného kódu. Pri spätnom preklade sa musia rekonštruovať chýbajúce informácie a odfiltrovať nadbytočný kód. Z tohto všetkého vyplýva, že presná replika pôvodného kódu nie je možná, avšak je možné zostaviť kód vykonávajúci ekvivalentnú činnosť.

Princíp fungovania spätného prekladača je založený na tom ako funguje prekladač. Niektoré časti sa však líšia, kedže pri spätnom preklade musí dôjsť k analýze chýbajúcich informácií a ich spätnému zostaveniu. Na obrázku 2.1 je možné vidieť jednotlivé časti všeobecného spätného prekladača.



Obr. 2.1: Schéma všeobecného spätného prekladača. Prevzaté z [8] a upravené.

Často je však možné stretnúť sa so spätnými prekladačmi, ktoré majú jednotlivé časti zoskupené do troch rozdielnych modulov. Schému spätného prekladača s tromi modulmi je možné vidieť na obrázku 2.2.



Obr. 2.2: Schéma všeobecného spätného prekladača s 3 modulmi. Prevzaté z [8] a upravené.

2.1 Použitie spätného prekladača

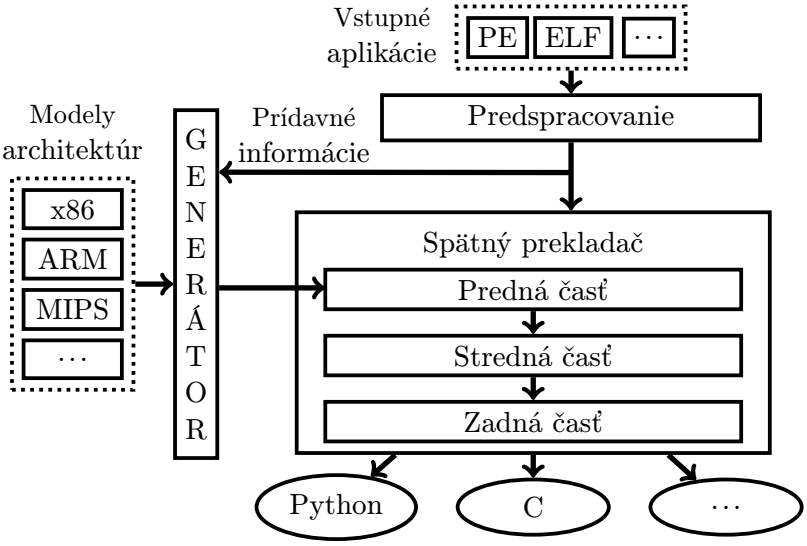
Existuje mnoho dôvodov prečo použiť spätný prekladač, avšak sú dve bežné odvetvia, kde sa spätný preklad využíva. Na nasledujúcich riadkoch sú popísané, spolu s tým na aké účely sa v daných odvetviach používajú.

- **Bezpečnosť**
  - **Analýza malwaru** — Koncom roka 2014 bolo takmer 3 miliardy ľudí pripojených k internetu [10]. To predstavuje 40% svetovej populácie. V týchto podmienkach je šírenie malwaru veľmi jednoduché. Analytici malwaru preto musia skúmať aký je rozsah škôd, ktoré môže napáchať, aké zraniteľnosti systému malware využíva, ale hlavne ako ho odstrániť zo systému a zamedziť ďalšiemu šíreniu.
  - **Detekcia chýb a zraniteľností** — Spätný preklad tiež môže slúžiť ako mechanizmus na hľadanie zraniteľností v software tretej strany, od ktorého nie sú dostupné zdrojové kódy. Na tieto zraniteľnosti môže byť následné autor aplikácie upozornený, čo prispeje k zvýšeniu bezpečnosti.
- **Vývoj softwaru**
  - **Interoperabilita s nedokumentovanými rozhraniami** — Pri práci s nedokumentovanými rozhraniami ako sú knižnice tretích strán, alebo aj niektorými nedokumentovanými aplikačnými rozhraniami systému je nutné pochopiť princíp ako funguje tento systém. Spätný preklad môže podstatne uľahčiť tento proces, nakoľko poskytne reprezentáciu vo vyššom programovacom jazyku.
  - **Obnova stratených zdrojových kódov** — Nie veľmi využívaný dôvod, avšak v prípade možnej straty je spätný preklad spôsob ako sa k pôvodným zdrojovým kódom dostať. Pokiaľ ho vykonáva skutočný autor aplikácie, ktorý pozná jej chovanie, tak môže jednoducho priradiť sémantiku jednotlivým konštrukciám a dostať svoje stratené zdrojové kódy späť. Veľa informácií však bude stratených, takže návrat stratených zdrojových kódov nie je jednoduchý proces.

2.2 Rekonfigurovateľný spätný prekladač spoločnosti AVG

Rekonfigurovateľný spätný prekladač [6] je vyvíjaný spoločnosťou AVG v spolupráci s Fakultou informačných technológií VUT v Brne. Cieľom je vytvoriť spätný prekladač, ktorý je možné konfigurovať pre špecifickú architektúru podľa prekladaného vstupného súboru. Rekonfigurovateľnosť je dosiahnutá zápisom modelu architektúry v jazyku ISAC (*Instruction Set Architecture C*).

Spätný preklad prebieha vo viac fázach v jednotlivých moduloch spätného prekladača. Na obrázku 2.3 je možné vidieť schému spätného prekladača spolu so vstupmi a výstupmi jednotlivých modulov. Jadro rekonfigurovateľného spätného prekladača používa variantu s troma modulmi popísanú na obrázku 2.2.



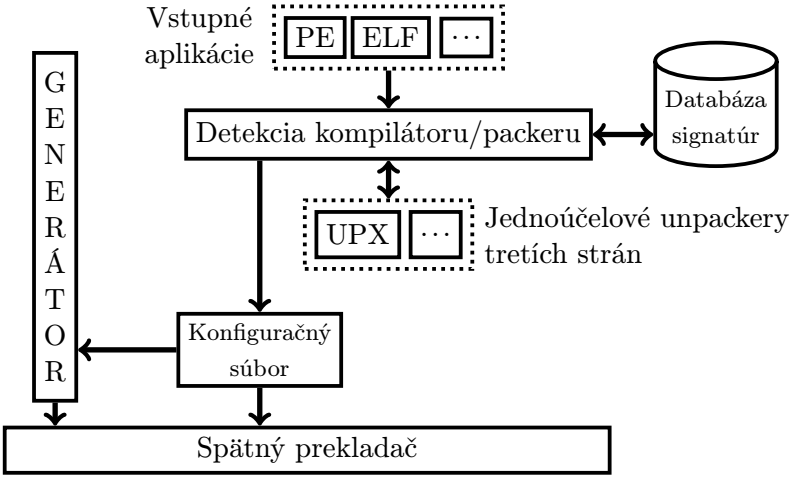
Obr. 2.3: Schéma rekonfigurovateľného spätného prekladača. Prevzaté z [12] a upravené.

2.2.1 Predspracovanie (Preprocessing)

V tejto fáze dochádza k zisteniu informácií o vstupnom súbore a konfigurácií nadväzujúcich častí spätného prekladu. Táto fáza nepatrí priamo k jadru spätného prekladača, ale je súčasťou procesu spätného prekladača. Schému procesu predspracovania je možné vidieť na obrázku 2.4.

Na zistenie informácií slúži nástroj `fileinfo`, ktorý rozozná formát vstupného súboru. Podporované formáty sú PE a ELF. Popis formátov spustiteľných súborov PE a ELF sa nachádza v kapitole 3. Zo vstupného spustiteľného súboru sú následne extrahované informácie ako sú

- Trieda spustiteľného súboru (32 alebo 64 bitová aplikácia)
- Cieľová architektúra
- Typ súboru (spustiteľný súbor, dynamická knižnica, ...)
- Použitý kompilátor a packer



Obr. 2.4: Schéma predspracovania. Prevzaté z [12] a upravené.

Detekcia použitého kompilátoru, alebo packeru je vykonávaná s pomocou databáze signatúr alebo heuristík. Signatúra je postupnosť inštrukcií charakteristická pre určitý kompilátor alebo packer. Zväčša sa jedná o postupnosť inštrukcií nachádzajúcich sa na vstupnom bode programu. Výstupom je konfiguračný súbor vo formáte JSON obsahujúci tieto informácie, ktorý je použitý v ďalších fázach spätného prekladu.

V prípade pozitívneho nálezu použitého packeru, je nutné vstupný súbor unpackovať. Pre príslušný packer a jeho verziu sa nájde odpovedajúci unpacker. V prípade úspešného unpackovania súboru je výstupom nový súbor, ktorý nahradí pôvodný vstupný súbor. Spracovaný vstupný súbor môže následne vstúpiť do ďalšej fázy spätného prekladu.

2.2.2 Predná časť (Front-end)

Jediná fáza jadra spätného prekladača, ktorá je platformovo závislá. Vstupom do tejto fázy je súbor vo formáte PE/ELF a model cieľovej architektúry vstupného súboru. Hlavným nástrojom tejto fázy je nástroj `decfront`. Ten má za úlohu transformovať vstupný súbor v PE/ELF formáte do LLVM IR (*Low Level Virtual Machine Intermediate Representation*) [1]. LLVM IR je nízkoúrovňový programovací jazyk vyvíjaný projektom LLVM, fungujúci na princípe SSA (*Static Single Assignment*). Tento princíp popisuje, že každej premennej je priradená hodnota iba jeden krát a každá premenná je pred svojím použitím definovaná. Tento jazyk je veľmi podobný assembleru a je platformovo nezávislí. Z toho dôvodu príde v tejto fáze k odstráneniu akejkoľvek platformovej závislosti. K tomu je však najskôr potrebné poznať pre akú architektúru je vstupný súbor určený a jej vlastnosti. Jej popis je dostupný z fázy predspracovania, z architektúry popísanej jazykom ISAC. LLVM IR kód tvorí výstup front-end fázy.

Na ukážke kódu 2.1 je jednoduchá funkcia zapísaná v jazyku C. K nej ekvivalentnú funkciu zapísanú v kóde LLVM IR je možné vidieť na ukážke 2.2.

```
int mul_add(int x, int y, int z)
{
    return x * y + z;
}
```

Kód 2.1: Ukážka C kódu. Prevzaté z [2].

```
define i32 @mul_add(i32 %x, i32 %y, i32 %z) {
entry:
    %tmp = mul i32 %x, %y
    %tmp2 = add i32 %tmp, %z ; Tu je možné vidieť SSA princíp. Nepoužije sa
        ↪ %tmp ale vytvorí sa %tmp2
    ret i32 %tmp2
}
```

Kód 2.2: Ekvivalentný LLVM IR kód k ukážke 2.1. Prevzaté z [2].

2.2.3 Stredná časť (Middle-end)

Táto fáza spätného prekladu slúži ako optimalizačná fáza. Dochádza tu k identifikácií inštrukčných idiómov [11] a ich nahradzovaniu sémanticky ekvivalentnému výrazmi. Tie budú následne v ďalších fázach po vygenerovaní vysokoúrovňovej reprezentácie čitateľnejšie a ich sémantika zjavná. Všetky tieto optimalizácie sú vykonávané nad LLVM IR kódom. Výstupom tejto fázy je optimalizovaný LLVM IR kód.

2.2.4 Zadná časť (Back-end)

Jedná sa o poslednú fázu spätného prekladu. Optimalizovaný LLVM IR kód je transformovaný na BIR (*Back-end IR*) kód. Ten už obsahuje konštrukcie vyšších jazykov ako sú cykly alebo podmienky. Nad týmto kódom sa taktiež vykonáva ďalšia sada optimalizácií, ktoré napríklad zjednodušujú výrazy, pomenúvajú premenné, zreťazené podmienky pretvárajú na switch, odstraňujú sa nedosiahnuteľné časti kódu atď. Zároveň sa zostroja aj grafy riadenia toku a volaní.

Zoptimalizovaný BIR kód je pretransformovaný na výstupný kód spätného prekladu. Ten tvorí spolu s vygenerovanými grafmi výstup tejto fázi, ale zároveň aj výstup celého spätného prekladača. Aktuálne sú podporované ako výstupné jazyky C a Python.

2.2.5 Ukážka výstupu

Pre znázornenie funkčnosti rekonfigurovateľného spätného prekladača bol vytvorený jednoduchý program v jazyku C. Tento kód je možné vidieť na ukážke 2.3. Bol preložený bez optimalizácií (`-O0`) a bez akýchkoľvek názvov symbolov (`-s`) prekladačom GCC pre architektúru Intel x86. Formát spustiteľného súboru bol použitý PE. Výstup je možné vidieť na ukážke 2.4.

```c
#include <stdio.h>

int mul_add(int x, int y, int z) {
    return x * y + z;
}

int main() {
    int i, sum = 0;
    for (i = 0; i < 5; ++i)
        sum += mul_add(i, i + 1, i + 2);

    printf("Vysledok:␣%i\n", sum);
    return 0;
}
```

Kód 2.3: Kód programu pre ukážkový spätný preklad.

```c
#include <stdint.h>
#include <stdio.h>

/* -------- Function Prototypes --------- */

int32_t function_401560(int32_t a1, int32_t a2, int32_t a3);

/* ------------- Functions -------------- */

int32_t function_401560(int32_t a1, int32_t a2, int32_t a3) {
    return a2 * a1 + a3;
}

int main(int argc, char ** argv) {
    int32_t v1 = 0;
    int32_t v2;
    for (uint32_t i = 0; i < 5; i++) {
        v2 = function_401560(i, i + 1, i + 2) + v1;
        v1 = v2;
    }
    printf("Vysledok:␣%i\n", v2);
    return 0;
}
```

Kód 2.4: Výstup rekonfigurovateľného spätného prekladača.

Je možné vidieť, že výstup nie je úplná replika pôvodného kódu. Pribudli niektoré premenné, ktoré sa v pôvodnom kóde nenachádzali. Dátový typ premenných bol približne rekonštruovaný, kde u premennej `i` sa zmenila znamienkovosť. Pôvodné názvy symbolov boli úplne stratené. Analýza pôvodného vstupného bodu programu úspešne lokalizovala funkciu `main`.

Kapitola 3

Spustiteľné súbory

Táto kapitola čerpá z [13], [7] a [14]. Jej úlohou je objasniť problematiku fungovania spustiteľných súborov do takej miery, aby bolo v nasledujúcich kapitolách jasné všetko, čo je potrebné vykonať pre dekompresiu takéhoto súboru. Bez tejto znalosti nie je možné zostaviť unpacker.

Spustiteľný súbor je súbor, ktorý obsahuje zápis programu pomocou vykonateľných inštrukcií a informácie o tom akým spôsobom sa dá spustiť. Spustiteľné súbory patria do skupiny objektových súborov, do ktorých môžme zaradiť ešte súbory linkovateľné a knižnice. Formáty pre tieto druhy objektových súborov však bývajú často rovnaké naprieč všetkými typmi s miernymi rozdielmi.

Obsah spustiteľného súboru musí byť zapísaný vo formáte, ktorému cieľový systém rozumie. Všeobecne však obsahuje nasledujúce informácie.

- **Hlavička** — Obsahuje základné informácie o súbore, identifikáciu formátu a informácie o organizácií jednotlivých častí súboru (kód, dáta atď.).
- **Objektový kód** — Binárne inštrukcie programu.
- **Relokačné záznamy** — Záznamy obsahujúce adresy, ktoré je treba v objektovom kóde upraviť v procese linkovania, alebo načítavania programu do pamäte. Táto časť je potrebná hlavne u knižníc a linkovateľných súboroch. Spustiteľné súbory ju používajú len zriedka. Variantou k relokáciam je pozične nezávislí kód (angl. *PIC - Position-Independent Code*).
- **Symboly** — Symboly exportovné von z modulu, symboly, ktoré je potrebné importovať do modulu a lokálne definované symboly.
- **Ladiace informácie** — Nie je povinné, aby objektový súbor obsahoval ladiace informácie. Tie slúžia hlavne pri vývoji a krokovaní programu. Sú to informácie o lokálnych symboloch, spájanie inštrukcií s číslami riadkov v zdrojovom kóde, informácie o štrukturovaných dátových typoch atď.

Proces výroby spustiteľného súboru sa skladá z dvoch krokov. Zdrojový kód je prekladačom preložený na linkovateľný objektový súbor. Ten nemusí byť ešte spustiteľný, nakoľko môže požadovať symboly z iného modulu. Z množiny linkovateľných objektových súborov linker zostaví spustiteľný súbor.

O spustenie sa v operačnom systéme stará zavádzač (angl. *loader*). Pri spustení je procesu pridelený jeho vlastný adresný priestor, mimo ktorý nemôže zasahovať. Do tohto

adresného priestoru sú tiež načítané všetky programom požadované dodatočné moduly. Zvyčajne sa jedná o knižnice.

Existuje mnoho formátov spustiteľných súborov, v tejto kapitole však budú popísané iba dva veľmi rozšírené formáty spustiteľných súborov, a to PE a ELF. Tie sú používané na väčšine dnešných systémov a preto sú aj najčastejšie terčom packerov.

3.1 Sekcie

Spustiteľný súbor je rozdelený na časti nazvané sekcie. Tie obsahujú kód a dáta programu s rôznym významom. Dôvod tohto delenia je, že rôzne časti programu majú rôzne vlastnosti a musí s nimi byť patrične zaobchádzané. Každá sekcia má svoje pomenovanie a atribúty určujúce vlastnosti.

Bežné delenie súboru je na kódovú sekciu (tiež nazývanú aj textová sekcia) obsahujúcu inštrukcie programu a dátovú sekciu obsahujúcu dáta programu. Pri načítavaní programu do pamäte dochádza k nastaveniu prístupových práv na stránky pamäte podľa atribútov sekcie. Kódová sekcia napríklad potrebuje aby bol jej obsah čitateľný a spustiteľný, zápis je zvyčajne v bežných podmienkach nepotrebný. Dátová sekcia zase obsahuje inicializované dáta programu, pre ktoré postačí ak sú iba čitateľné. Zvyčajne sa však v programoch nachádzajú aj sekcie obsahujúce neinicializované dáta, zoznam importovaných alebo exportovaných symbolov, avšak to už záleží od prekladača a aj konkrétneho formátu spustiteľného súboru.

3.2 Importovanie symbolov

Je bežné, že programy nepracujú len čisto s vlastnými funkciami. Využívajú pri svojej činnosti mnoho knižníc, či už systémových alebo užívateľských. Knižnicu je možné počas procesu linkovania tzv. prilinkovať k programu staticky alebo dynamicky. Pri statickom linkovaní je knižnica priamo vložená do súboru programu, čo však vedie k nevýhodám ako nutnosť preložiť program znova pri zmene knižnice, či k zvýšenej veľkosť výsledného súboru.

Dynamické knižnice tieto nedostatky odstraňujú. Pri dynamickom linkovaní je postačujúce, pokiaľ je počas linkovania poskytnutá knižnica, ktorá požadované symboly exportuje. Pri spustení aplikácie je dynamická knižnica opäť lokalizovaná a načítaná do adresného priestoru procesu. Nevýhodou však je, že do programu sa nemôžu doplniť absolútne adresy týchto symbolov, pretože sa nevie na akú adresu loader načíta dynamické knižnice. Musia sa zaviesť mechanizmy, ktoré sa pred spustením programu postarajú o doplnenie týchto adries. Každý formát spustiteľných súborov tento problém rieši rôzne.

3.3 PE

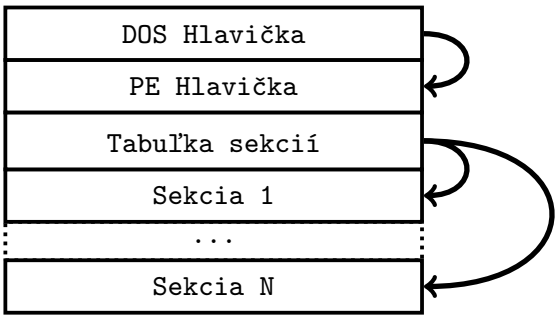
Táto podkapitola čerpá z [16] a [18].

Formát PE (*Portable Executable*) je formát spustiteľných súborov a DLL (*Dynamic-link Library*) knižníc. Je vyvíjaný spoločnosťou Microsoft. Vznikol ako modifikácia formátu COFF používaného v Unixových systémoch, preto sa niekedy označuje aj PE/COFF. Je určený pre operačné systémy Windows rodiny NT.

Formát popisujúci programy určené pre 32 bitové systémy sa nazýva PE32. K nemu bolo vydané rozšírenie PE32+ pre 64 bitové systémy. V tejto kapitole bude popísaný formát PE32

nakoľko sú takmer totožné. Líšia sa len veľkosťami ukazovateľov a niektorými magickými konštantami.

3.3.1 Štruktúra



Obr. 3.1: Štruktúra spustitelného súboru v PE formáte.

3.3.2 DOS Hlavička

Súbory formátu PE začínajú DOS hlavičkou, ktorá je vlastne samostatný MS-DOS program. Nachádza sa tu z historických dôvodov, kedy sa PE formát ešte len zavádzal a väčšina ľudí stále vlastnila MS-DOS. Namiesto toho aby systém upozornil užívateľa, že daný formát nie je spustiteľný, tak sa do DOS hlavičky vložil malý program vypisujúci hlásenie, že na spustenie tohto programu je potrebný Windows. Zvyčajne sa umiestňuje do tejto hlavičky už niekoľko desaťročí stále to isté hlásenie, avšak dnes už prakticky bez využitia.

Štruktúra popisujúca DOS hlavičku je `IMAGE_DOS_HEADER`. Jej deklaráciu je možné vidieť na ukážke kódu v prílohe A.1. Na nasledujúcich riadkoch budú popísané niektoré podstatné členské premenné tejto štruktúry.

- `e_magic` — Nachádza sa na úplnom začiatku súboru. Pre validnú DOS hlavičku je požadované aby tento `WORD` (16 bitov) obsahoval hodnotu `0x5A4D`, ktorá predstavuje ASCII kódy znakov `MZ`, podľa architekta MS-DOS-u Marka Zibowskiho. Z toho dôvodu sa DOS hlavička označuje aj ako MZ hlavička. Pre validnú hodnotu existuje konštanta `IMAGE_DOS_SIGNATURE`.

- `e_lfanew` — Obsahuje offset v rámci súboru na ktorom sa nachádza PE hlavička. Tá sa nenachádza priamo za DOS hlavičkou z dôvodu prítomnosti kódu MS-DOS programu. MS-DOS tento člen nikdy nečíta a je určený čisto len pre PE súbory.

3.3.3 PE Hlavička

Hlavička PE sa v súbore nachádza na offsete špecifikovanom v DOS hlavičke, ako je popísané v kapitole 3.3.2. Jej štruktúru popisuje typ `IMAGE_NT_HEADERS`, ktorého deklarácia sa nachádza v ukážke 3.1.

Skladá sa zo signatúry `Signature` identifikujúcej, že sa jedná o PE hlavičku a ďalších dvoch štruktúr `IMAGE_FILE_HEADER` a `IMAGE_OPTIONAL_HEADER`. Signatúra musí obsahovať hodnotu `0x00004550`, ktorá reprezentuje sekvenciu ASCII kódov znakov `PE\0\0`.

Prvá vnorená hlavička `FileHeader` typu `IMAGE_FILE_HEADER` má predpis podľa ukážky kódu 3.2. Tá obsahuje informácie o PE súbore ako architektúra CPU, pre ktorú je PE súbor

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                 Signature;
  IMAGE_FILE_HEADER     FileHeader;
  IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS;
```

Kód 3.1: Štruktúra IMAGE_NT_HEADERS.

určený (Machine), tiež či je súbor spustiteľný alebo DLL knižnica, či sa v ňom nachádzajú relokácie (Characteristics). NumberOfSections obsahuje počet sekcií v programe. Sekcie sú podrobnejšie popísané ďalej v kapitole 3.3.4.

Druhá vnorená hlavička OptionalHeader je síce pomenovaná ako *optional* (nepovinná), avšak PE súbor bez nej nemôže správne fungovať. Je typu IMAGE_OPTIONAL_HEADER. Obsahuje najviac podstatných informácií pre spustenie. Jej štruktúru je možné vidieť na ukážke kódu v prílohe A.2.

```
typedef struct _IMAGE_FILE_HEADER {
  WORD  Machine;
  WORD  NumberOfSections;
  DWORD TimeDateStamp;
  DWORD PointerToSymbolTable;
  DWORD NumberOfSymbols;
  WORD  SizeOfOptionalHeader;
  WORD  Characteristics;
} IMAGE_FILE_HEADER;
```

Kód 3.2: Štruktúra IMAGE_FILE_HEADER.

Člen Magic môže nadobúdať dve hodnoty a to 0x10B pre 32-bitovú aplikáciu a 0x20B pre 64-bitovú aplikáciu. Jeden z najdôležitejších členov pre spustenie je ImageBase. Určuje adresu, na ktorú je obraz spustiteľného súboru umiestnený do pamäte. Jedná sa len o preferovanú hodnotu programom, loader môže načítať obraz aj na inú adresu. Pre spustiteľné súbory je zvyčajná hodnota 0x00400000 a pre DLL knižnice 0x10000000. Ďalším dôležitým členom je AddressOfEntryPoint, ktorý obsahuje relatívny offset vstupného bodu programu od ImageBase. Hlavička ďalej obsahuje informácie o tom, kde sa nachádzajú začiatky dátových a kódových sekcií (BaseOfCode, BaseOfData) a ich veľkosti (SizeOfCode, SizeOfInitializedData, SizeOfUninitializedData) a iné dodatočné informácie o vlastnostiach programu.

Na konci tejto hlavičky sa nachádza v členskej premennej DataDirectory pole *data directories*. *Data directory* je typu IMAGE_DATA_DIRECTORY, čo je štruktúra obsahujúca relatívnu adresu, kde sa nachádzajú dáta daného *data directory* a jeho veľkosť. Každý *data directory* obsahuje špecifické informácie podľa jeho typu, ktorý určuje ich sémantiku. Každý typ má pridelený pevný index v poli, podľa ktorého sa vyhľadáva konkrétny *data directory*. Používajú sa napríklad pre tabuľku exportovaných symbolov, tabuľku importovaných symbolov alebo tabuľku adries importovaných symbolov. Tieto tabuľky sa vo väčšine prípadov nachádzajú vo vlastných, špecificky pomenovaných, sekciách, avšak to nie je zaručené špecifikáciou. Správne by sa mali tieto tabuľky vyhľadávať v *data directories*. Ich počet určuje NumberOfRvaAndSizes.

3.3.4 Tabuľka sekcií

Tabuľka sekcií sa nachádza bezprostredne za PE hlavičkou. Skladá sa zo záznamov typu `IMAGE_SECTION_HEADER`. Predpis tohto typu je možné vidieť na ukážke 3.3. Každý tento záznam predstavuje hlavičku jednej sekcie. Ten obsahuje informácie ako relatívna adresa začiatku sekcie voči `ImageBase` (`VirtualAddress`) a rôzne iné. V tabuľke sekcií sa nachádza `NumberOfSections` hlavičiek, ako je popísané v kapitole 3.3.3.

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD  NumberOfRelocations;
  WORD  NumberOfLinenumbers;
  DWORD Characteristics;
} IMAGE_SECTION_HEADER;
```

Kód 3.3: Štruktúra `IMAGE_SECTION_HEADER`.

3.3.5 Importovanie symbolov

V súboroch formátu PE sú tabuľky pre exportované a importované symboly uložené v *data directories* na určitých indexoch. Pre importované symboly je to index 1. *Import data directory* obsahuje záznamy pre každú DLL knižnicu z ktorej je potrebné naimportovať aspoň jeden symbol. Tieto záznamy sú typu `IMAGE_IMPORT_DESCRIPTOR`, ktorý je vidno na ukážke kódu 3.4.

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
  _ANONYMOUS_UNION union {
        DWORD Characteristics;
        DWORD OriginalFirstThunk;
  } DUMMYUNIONNAME;
  DWORD TimeDateStamp;
  DWORD ForwarderChain;
  DWORD Name;
  DWORD FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR;
```

Kód 3.4: Štruktúra `IMAGE_IMPORT_DESCRIPTOR`.

V rámci štruktúry sú dôležité členy ako `Name`, `OriginalFirstThunk` a `FirstThunk`. `Name` obsahuje odkaz na názov DLL knižnice, z ktorej je potrebné importovať symboly. `OriginalFirstThunk` obsahuje offset, kde sa nachádza ILT (*Import Lookup Table*). Tá obsahuje zoznam symbolov, ktoré je nutné importovať či už vo forme ich názvov, alebo indexmi do tabuľky exportovaných symbolov danej knižnice, čiže tzv. import ordinálom.

`FirstThunk` zase obsahuje offset do IAT (*Import Address Table*), ktorá obsahuje adresy jednotlivých symbolov v rámci importovaného modulu. Pred spustením programu sa obsah ILT a IAT ničím nelíši, avšak po spustení a načítaní všetkých knižníc do adresného priestoru procesu loader naplní IAT odpovedajúcimi adresami. Program na prístup k importovanému symbolu použije IAT, aby pristúpil na skutočnú adresu symbolu.

Pri adresovaní niektorého z importovaných symbolov sa používajú najčastejšie dve metódy. Buď sa používa priamo ukazovateľ do IAT, alebo v prípade volania importovaných funkcií sa niekedy používajú aj tabuľky skokov. V tabuľke skokov sa nachádzajú nepodmienené skoky na adresy, ktorými je naplnená IAT. Do miesta odkiaľ sa takáto funkcia volá je umiestnené volanie niektorej položky v tabuľke skokov. Tým sa vykoná skok do iného modulu so zachovaním zásobníku.

3.3.6 Thread-Local Storage

Thread-Local Storage—skrátene TLS—je špeciálny druh pamäti, ktorý je unikátny pre každé samostatné vlákno, ktoré v procese beží [16]. Táto pamäť musí byť inicializovaná ešte pred samotným spustením programu, o čo sa postará loader. Formát PE však poskytuje aj možnosť vykonať špecifikované funkcie pri inicializácií TLS. Špecifikovať je ich potrebné v *TLS data directory*, ktoré má štruktúru ako je na ukážke kódu 3.5. Pomocou členu `AddressOfCallBacks` je možné definovať tzv. *TLS callbacky*, ktoré budú vykonané ešte pred štartom programu.

```
typedef struct _IMAGE_TLS_DIRECTORY32 {
  DWORD StartAddressOfRawData;
  DWORD EndAddressOfRawData;
  DWORD AddressOfIndex;
  DWORD AddressOfCallBacks;
  DWORD SizeOfZeroFill;
  DWORD Characteristics;
} IMAGE_TLS_DIRECTORY32;
```

Kód 3.5: Štruktúra `IMAGE_TLS_DIRECTORY`.

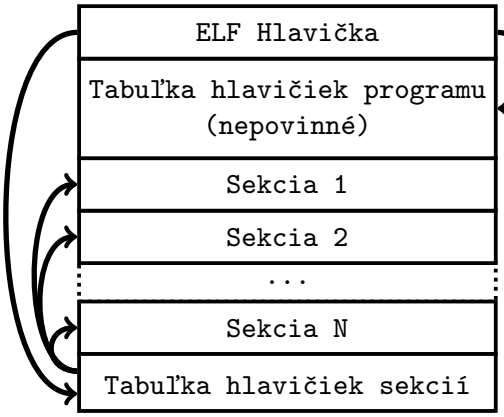
3.4 ELF

Táto podkapitola čerpá z [15].

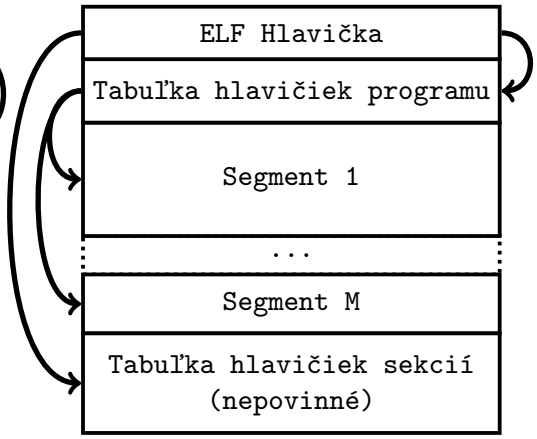
Formát ELF (*Executable and Linkable Format*) je formát pre všetky druhy objektových súborov, či už sa jedná o spustiteľné, linkovateľné alebo knižnice. Bol navrhnutý pri práci na UNIX System V Release 4.0 koncom 90. rokov ako náhrada za staršie formáty `a.out` a COFF. V dnešnej dobe je to štandardný formát všetkých Unixových systémov.

3.4.1 Štruktúra

Na formát ELF je možné sa pozerať z dvoch hľadísk. Na obrázku 3.2 je možné vidieť štruktúru súboru v ELF formáte z linkovateľného hľadiska. Na obrázku 3.3 je naznačený pohľad na súbor zo spustiteľného hľadiska. Ten naznačuje štruktúru, ako by vyzeral program v pamäti. Rozdiel je, že pri spustení sa nepracuje zo sekciami ako pri linkovaní, ale so segmentami. Sekcie sú pritom mapované na segmenty. Jednému segmentu môže odpovedať aj niekoľko sekcií, avšak nemusí ani jedna.



Obr. 3.2: Linkovateľné hľadisko



Obr. 3.3: Spustiteľné hľadisko

3.4.2 ELF Hlavička

ELF hlavička sa nachádza na úplnom začiatku súboru. Obsahuje všeobecné informácie o súbore a tom, kde sa nachádzajú ostatné hlavičky. Jej štruktúru je možné vidieť na ukážke kódu 3.6.

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
  Elf32_Half    e_type;             /* Object file type */
  Elf32_Half    e_machine;          /* Architecture */
  Elf32_Word    e_version;          /* Object file version */
  Elf32_Addr    e_entry;            /* Entry point virtual address */
  Elf32_Off     e_phoff;            /* Program header table file offset */
  Elf32_Off     e_shoff;            /* Section header table file offset */
  Elf32_Word    e_flags;            /* Processor-specific flags */
  Elf32_Half    e_ehsize;           /* ELF header size in bytes */
  Elf32_Half    e_phentsize;        /* Program header table entry size */
  Elf32_Half    e_phnum;            /* Program header table entry count */
  Elf32_Half    e_shentsize;        /* Section header table entry size */
  Elf32_Half    e_shnum;            /* Section header table entry count */
  Elf32_Half    e_shstrndx;         /* Section header string table index */
} Elf32_Ehdr;
```

Kód 3.6: Štruktúra ELF hlavičky.

Význam väčšiny členov je zjavný už len z komentárov pri nich. Na úplnom začiatku sa nachádza identifikácia ELF formátu v poli `e_ident`, ktoré má 16 členov. Aby bolo možné súbor považovať za súbor v ELF formáte, je potrebné aby prvé 4 členy boli nastavení na tzv. magickú hodnotu `'\x7F', 'E', 'L', 'F'`. Zvyšné členy obsahujú informácie ako trieda, verzia ABI (*Application Binary Interface*), verziu formátu súboru a iné.

Hlavička tiež obsahuje odkaz na tabuľku názvov sekcií v `e_shstrndx`,. Sekcie v ELF súbore sú podrobnejšie popísané v 3.4.3.

3.4.3 Tabuľka hlavičiek sekcií

Tabuľka hlavičiek sekcií obsahuje hlavičky jednotlivých sekcií nachádzajúcich sa v súbore. Je umiestnená na konci súboru za telami sekcií. Jednotlivé záznamy sú typu `Elf32_Shdr`, ktorý je možné vidieť na ukážke kódu 3.7.

```
typedef struct
{
  Elf32_Word    sh_name;        /* Section name (string tbl index) */
  Elf32_Word    sh_type;        /* Section type */
  Elf32_Xword   sh_flags;       /* Section flags */
  Elf32_Addr    sh_addr;        /* Section virtual addr at execution */
  Elf32_Off     sh_offset;      /* Section file offset */
  Elf32_Xword   sh_size;        /* Section size in bytes */
  Elf32_Word    sh_link;        /* Link to another section */
  Elf32_Word    sh_info;        /* Additional section information */
  Elf32_Xword   sh_addralign;   /* Section alignment */
  Elf32_Xword   sh_entsize;     /* Entry size if section holds table */
} Elf32_Shdr;
```

Kód 3.7: Štruktúra hlavičky sekcie.

Každá sekcia má svoj názov uložený v tabuľke reťazcov, do ktorej je `sh_name` index. Účel sekcie je definovaný jej typom v `sh_type`. Typ taktiež definuje sémantiku `sh_link` člena, ktorý má pre určité typy špecifický význam a odkazuje na pomocné tabuľky a iné štruktúry.

Ďalšie členy určujú informácie o vlastnostiach sekcie (`sh_flags`), mieste kde začína (`sh_offset`), veľkosti zarovnania (`sh_addralign`) a jej celkovej veľkosti (`sh_size`).

3.4.4 Tabuľka hlavičiek programu

Tabuľka hlavičiek programu obsahuje hlavičky programu (alebo programové hlavičky). Nachádza sa za ELF hlavičkou. Používa sa pri spúšťaní súboru, pretože definuje akým spôsobom sa sekcie budú mapovať na segmenty.

Hlavičky majú typ `Elf32_Phdr` a štruktúru ako je na ukážke kódu 3.8. Obsahuje takmer totožné informácie ako sa nachádzajú v hlavičkách sekcií.

```
typedef struct
{
  Elf32_Word    p_type;         /* Segment type */
  Elf32_Word    p_flags;        /* Segment flags */
  Elf32_Off     p_offset;       /* Segment file offset */
  Elf32_Addr    p_vaddr;        /* Segment virtual address */
  Elf32_Addr    p_paddr;        /* Segment physical address */
  Elf32_Xword   p_filesz;       /* Segment size in file */
  Elf32_Xword   p_memsz;        /* Segment size in memory */
  Elf32_Xword   p_align;        /* Segment alignment */
} Elf32_Phdr;
```

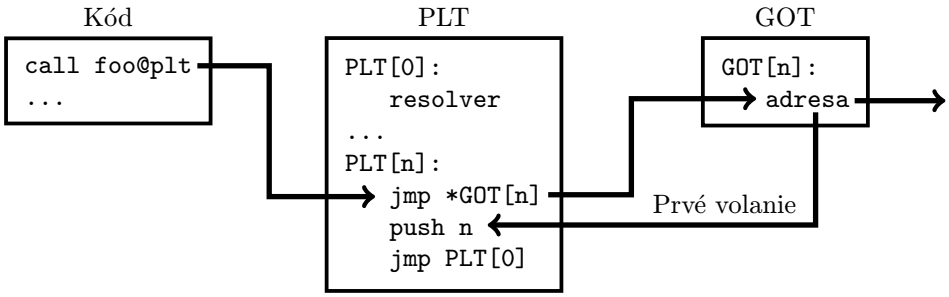
Kód 3.8: Štruktúra hlavičky programu.

3.4.5 Importovanie symbolov

Programy v ELF formáte používajú pri importovaní GOT (*Global Offset Table*) tabuľku. Pri preklade sa vytvorí v špeciálnom segmente GOT, ktorá je na známom statickom offsete. Jednotlivé prvky tejto tabuľky slúžia na uskladnenie skutočných adries symbolov. Pri spúšťaní loader lokalizuje `DYNAMIC` segment pomocou tabuľky hlavičiek programu. Tento segment obsahuje všetky informácie o tom, aké symboly je potrebné načítať z akých knižníc. Následne sú naplnené záznamy v GOT.

Importované funkcie však fungujú odlišným spôsobom. Tie používajú PLT (*Procedure Linkage Table*) pri ich volaní. Táto tabuľka funguje na princípe tzv. *lazy binding*, čo znamená, že k previazaniu zdanlivej a skutočnej adresy funkcie dochádza až v momente, keď je to potrebné, tj. keď je funkcia volaná. Taktiež ako GOT, aj PLT sa nachádza vo vlastnom segmente. Prvý záznam v PLT je vždy *resolver*, ktorý má za úlohu zistiť skutočnú adresu funkcie. Zvyšné záznamy predstavujú jednotlivé importované funkcie. V nich sa nachádza skok do GOT, v ktorej by sa mala nachádzať skutočná adresa, avšak v prípade prvého volanie danej funkcie sa tu nachádza iba skok späť do PLT, čo spôsobí spustenie *resolveru*, ktorý naplní patričný GOT záznam. V prípade ďalšieho volania je už úvodným skokom do GOT zistená skutočná adresa.

Na obrázku 3.4 je naznačená zjednodušená schéma princípu fungovania PLT pri volaní importovanej funkcie `foo`.



Obr. 3.4: Schéma PLT

Kapitola 4

Kompresia spustiteľných súborov

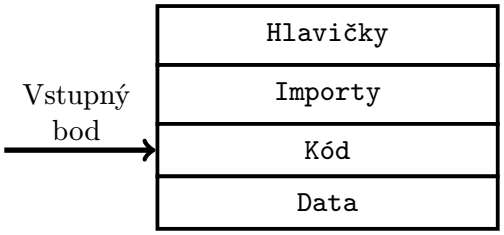
Táto kapitola čerpá z [22].

Kompresia (komprimácia) spustiteľných súborov je proces, počas ktorého dochádza ku kompresií kódovej, prípadne aj dátovej časti spustiteľného súboru. Podstatné je však, že súbor zostáva naďalej spustiteľný a vykonáva svoju pôvodnú činnosť. Nástroj, ktorý kompresiu vykonáva sa nazýva packer.

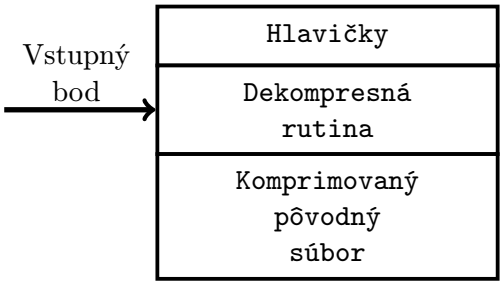
Hlavné dôvody pre kompresiu spustiteľných súborov sú zmenšenie veľkosti a skomplikovanie analýzy. Najčastejšie využíva kompresiu škodlivý software, z dôvodu zamaskovania svojej činnosti pred analýzou antivírusového programu. Takto skomprimovaný súbor nemusí byť identifikovaný ako škodlivý a pri jeho spustení poškodí užívateľa, či jeho systém. Môže sa však jednať len o ochranu pred reverzným inžinierstvom u proprietárneho softwaru.

Napriek tomu, že existuje mnoho packerov, všetky fungujú na podobnom princípe. Vytvoria nový spustiteľný súbor obsahujúci časť pôvodného spustiteľného súboru ako skomprimované dáta a vložia doňho kód, ktorý dokáže tento skomprimovaný obsah dekomprimovať. Veľmi často tiež dochádza k modifikácií zoznamu importovaných symbolov, aby sa zakryla skutočná činnosť programu.

Na obrázku 4.1 je možné vidieť štruktúru spustiteľného súboru pred kompresiou. Stav po kompresií je naznačený na obrázku 4.2.



Obr. 4.1: Pôvodný súbor



Obr. 4.2: Komprimovaný súbor

4.1 Dekompresná rutina

Dekompresná rutina (angl. *unpacking stub*) je časť kódu, ktorú doplní do komprimovaného súboru packer. V komprimovanom súbore je vstupný bod programu nastavený práve do tejto rutiny. Jej úlohu je možné zhrnúť do nasledujúcich bodov.

- Dekompresia obsahu pôvodného spustiteľného súboru do pamäti alebo na disk.
- Oprava poškodených hlavičiek pôvodného súboru (importy, relokácie a iné).
- Presmerovanie riadenia programu na OEP (*original entry point*—pôvodný vstupný bod).

Akým spôsobom jednotlivé kroky rutina vykonáva záleží od konkrétneho packeru. Rôzne packery využívajú rôzne kompresné algoritmy, aj rôzne modifikujú hlavičky pôvodného súboru.

Dekompresná rutina je jediná viditeľná ako kód z pohľadu statickej analýzy. Staticky analyzovať skomprimovaný programu teda znamená analyzovať dekompresnú rutinu, nie skutočný program.

4.1.1 Rekonštrukcia tabuľky importovaných symbolov

Pri spúšťaní programu musí loader operačného systému naimportovať symboly. Akým spôsobom sú uložené tieto informácie v jednotlivých formátoch je popísané v kapitolách 3.3 a 3.4. Nakoľko sú ale tieto informácie skomprimované, tak ich loader nemôže prečítať. Túto činnosť preto musí vykonať dekompresná rutina. Existuje niekoľko spôsobov akými dekompresná rekonštruuje tabuľku importovaných symbolov.

Najčastejšie riešenie je, že sa ponechá len import dôležitých funkcií pre rekonštrukciu zvyšku importov. V prípade Windowsu sú to funkcie `LoadLibrary` a `GetProcAddress` a zase pre Linux `dlopen` a `dlsym`. Môžu to byť aj iné funkcie, napríklad pre nastavenie parametrov stránok v pamäti, alebo ich vytváranie. Po tom ako rutina dekomprimuje informácie o pôvodných importoch, tak použije tieto funkcie pre zostavenie pôvodných importov.

Ďalšie riešenie je ponechať tabuľku importov ako je. Je to najjednoduchší spôsob, pretože dekompresná rutina nemusí vykonávať opravovanie importov. Ponechaná tabuľka importov vie jednoducho prezradiť aké volania komprimovaný program vykonáva. Nejedná sa o optimálne riešenie.

Tretí spôsob je kombinácia predošlých dvoch. Pokiaľ sa ponechá len jeden importovaný symbol z každej použitej knižnice, tak nie je naďalej potrebné aby rutina načítala knižnice do adresného priestoru. Postačí len doimportovať zvyšné symboly z každej knižnice. Jedná sa o bezpečnejší spôsob oproti druhému, avšak stále prezrádza dostatok informácií.

Posledný spôsob je najbezpečnejší zo všetkých, zároveň je však o to potrebné vykonať viac úkonov pri oprave importov. Ide o variantu, kedy packer nezanechá v komprimovanom súbore žiadne importované symboly. Dekompresná rutina sa musí následne sama postarať o to, aby lokalizovala knižnice obsahujúce funkcie popísané v riešení 1. U Windowsu je to knižnica `KERNEL32.DLL` a u Linuxu `libdl.so`. Po lokalizácií potrebných funkcií ich použije na opravenie tabuľky importov.

4.2 Dekompresia

Dekompresia je proces získavania pôvodného spustiteľného súboru z jeho komprimovanej verzie. Na základe toho kto alebo čo vykonáva dekompresiu, ju môžme rozdeliť na dva druhy.

- **Manuálna** — Pri tomto druhu sa využívajú ladiace nástroje ako *debugger* a *disassembler*. Človek, ktorý požaduje dekompresiu musí zanalyzovať dekompresnú rutinu a následne si manuálne dekomprimovať súbor.

- **Automatická** — Využívajú sa pri nej nástroje, ktoré sú priamo určené na dekompresiu istých packerov. Človek nemusí vôbec analyzovať vnútornú štruktúru programu. Špecializovaný nástroj spraví všetko automaticky. Nástroje, ktoré túto činnosť vykonávajú sa nazývajú unpackery.

Ďalšie možné delenie dekompresie je podľa toho akým spôsobom je vykonávaná.

- **Statická** — Program nie je počas statickej dekompresie spúšťaný. Unpacker vykoná rovnakú činnosť ako dekompresná rutina obsiahnutá v komprimovanom súbore. Toto riešenie je bezpečné pokiaľ pôvod súboru nie je známy a mohol by poškodiť systém. Ďalšou výhodou je, že môžme pracovať s ľubovolným formátom spustiteľných súborov, preto môžme dekomprimovať aj súbory určené pre iné systémy a architektúry. K nevýhodám patrí zložitosť zostrojenia takého unpackeru, nakoľko je potrebná starostlivá analýza dekompresnej rutiny, aby ju bolo možné nanovo zostaviť.

- **Dynamická** — Pri tomto druhu dekompresie sa program spúšťa s cieľom nájsť moment v programe, kedy dochádza k predaniu riadenia na OEP. Pokiaľ dekompresná rutina dospeje až do tohto momentu, tak je takmer isté, že sa už pôvodný kód a dáta nachádzajú v pamäti dekomprimované. Všetka práca je teda ponechaná na dekompresnú rutinu samotnú. V tomto momente je možné vytvoriť obraz pamäti procesu pomocou nástrojov ako `OllyDump`[1] zásuvného modulu do `OllyDbg`[2], ktorý z neho zostaví spustiteľný súbor. Nejedná sa však o bezpečný spôsob, pretože sa môže jednať o škodlivý software, ktorý pri spustení poškodí systém.

V praxi sa najčastejšie používajú dynamické unpackery (automatická dynamická dekompresia), aj napriek rizikám, ktoré prinášajú. Jedná sa totiž o najjednoduchšiu variantu.

Veľmi často je však treba siahnuť k manuálnej dekompresií, nakoľko unpackery nie sú úplne spoľahlivé. Tie hľadajú v komprimovanom súbore najčastejšie signatúry daného packeru, ktoré môže autor pozmeniť tak aby tieto detekcie zlyhali. V tomto prípade je potrebné buďto manuálne opraviť pozmenené dáta, tak aby unpacker vyhodnotil súbor ako validný, alebo vykonať úplnú dekompresiu vlastnoručne. Väčšinou potom ide o manuálnu dynamickú analýzu, pretože statický prístup sa bez znalosti dekompresnej rutiny nedá vykonať.

4.3 Ochrana proti reverznému inžinierstvu

Analýza fungovania packeru a jeho dekompresnej rutiny je jednoduchý proces. Postačia nástroje, ktoré sú schopné zobraziť inštrukcie programu v čitateľnej podobe—disassembler a pre prípad dynamickej analýzy debugger. Z toho dôvodu sa snažia autori packerov zaviesť mechanizmy, ktoré by túto analýzu zkomplikovali a oddialili. Nazývame ich taktiež *anti-debugging* alebo *anti-disassembly* mechanizmy, podľa toho, v akej fáze reverzného inžinierstva majú za úlohu skomplikovať analýzu. Následky týchto ochrán môžu byť rôzne. Program môže svoju činnosť ukončiť, prípadne sa znehodnotia niektoré dáta a analytik má pocit, že program funguje správne, avšak príde k úplne odlišným výsledkom.

Je treba poznamenať, že jednotlivé ochrany sú závislé od architektúry a platformy na ktorej spustený program pobeží, ale aj od použitých nástrojov na reverzné inžinierstvo. Na nasledujúcich riadkoch sú popísané niektoré z najpoužívanejších spôsobov ochrany. Príklady sú uvádzané pomocou inštrukcií architektúry x86, avšak všetky spomenuté mechanizmy je možné aplikovať aj na iných architektúrach.

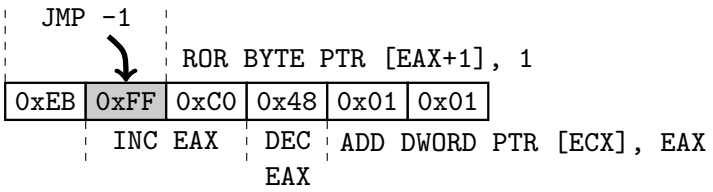
[1] http://www.openrce.org/downloads/details/108/OllyDump
[2] http://www.ollydbg.de/

Vkladanie bajtov medzi inštrukcie

Táto technika sa používa na zmätenie disassemblerov používajúcich lineárny priechod (angl. *linear sweep*) [21], ktoré už v súčastnosti nie sú rozšírené. Tie transformujú postupnosti bajtov na inštrukcie bez ohľadu na význam a obsah inštrukcie. Medzi inštrukcie sú teda vložené buď náhodné, alebo špecificky zvolené postupnosti bajtov, ktoré sú programom preskakované skokovými inštrukciami umiestnenými pred týmito postupnosťami. Disassembler používajúci lineárny priechod však nepozná význam inštrukcie skoku, preto tieto bajty interpretuje ako inštrukcie, čo môže spustiť reťazovú reakciu zle transformovaného obsahu. Riešením je použiť disassembler využívajúci rekurzívny priechod (angl. *recursive traversal*) [21], ktorý sa pri náleze inštrukcií, ktoré presmerovávajú riadenie programu, taktiež presmeruje na miesto, kam sa odkazuje inštrukcia.

Vkladanie bajtov priamo do inštrukcií

Špecifický prípad, kedy sú vložené bajty súčasťou platných inštrukcií. V tomto prípade už ani disassembler využívajúci rekurzívny priechod nedokáže zobraziť správne inštrukcie, bez zásahu človeka. Na obrázku 4.3 je možné vidieť názornú ukážku takto vloženého bajtu vyznačeného šedou farbou. Tento bajt je súčasť inštrukcie skoku, ale sám je operačný kód pre inštrukciu `INC`. Takto umiestený bajt sa nazýva *rogue byte*. Ako je vidieť, skok sa vykoná na druhý bajt inštrukcie `JMP -1` odkiaľ sa začnú vykonávať ďalšie inštrukcie, ktoré pôvodne disassembler neidentifikuje správne. Riešením by bolo napríklad bajt `0xEB` nahradiť za inštrukciu `NOP (0x90)`, ktorá značí inštrukciu nevykonávajúcu nič, len inkrementáciu programového čítača.



Obr. 4.3: *Rogue byte* vložený priamo do platných inštrukcií.

Riadenie toku programu návratovou inštrukciou

Už bolo spomenuté ako funguje disassembler využívajúci rekurzívny priechod. Bežne používané inštrukcie pre riadenie toku ako `JMP` a `CALL` prezradia, kde sa nachádza ďalší kód programu. Musia totiž obsahovať informáciu o tom, kam sa bude riadenie presmerovávať. Autori packerov preto používajú na riadenie toku aj návratovú inštrukciu `RET`. Je to ekvivalent pre dvojicu inštrukcií `POP` a `JMP` na adresu získanú zo zásobníku. Postačí pred vykonaním tejto inštrukcie na zásobník umiestniť adresu, kam sa má riadenie programu presmerovať. Disassemblery často neočakávajú takéto použitie návratovej inštrukcie a neidentifikujú správne odkazované časti kódu.

Výnimky

Pokiaľ je debugger pripojený k procesu, tak výnimky obsluhuje za krokovaný program, s tým že ich môže posunúť ďalej krokovanému programu. Niektoré výnimky, ako *breakpoint* výnimka (`INT 3`) alebo *single-step* výnimka (`INT 1`) sú tie, ktoré používa debugger na svoju

činnosť. Na tieto výnimky preto môže byť zavedená kontrola, kto danú výnimku obsluhuje a skontrolovať tak prítomnosť debuggeru. Poprípade môžu byť vyvolávané rôzne výnimky, ktoré pokiaľ nie sú presmerované späť do programu, môžu znepríjemniť analýzu neustálymi skokmi mimo hlavný tok programu.

Skúmanie stôp po debuggery

Program spustený pod debuggerom je inicializovaný mierne odlišne. Debugger zanechá v riadiacich štruktúrach programu stopy. Kontrola týchto stôp môže jednoducho prezradiť prítomnosť debuggeru. Tento spôsob je vysoko závislí na systéme, pod ktorým spustený program beží. Na operačnom systéme Windows je na to napríklad vyhradená funkcia `IsDebuggerPresent`.

Samomodifikujúci kód

Jedná sa o kód, ktorý sám mení inštrukcie, ktoré sa budú vykonávať alebo vykonávali. Komplikuje to sledovanie toku programu a znemožňuje vytváranie obrazu pamäte.

Časovač

Pri krokovaní programu sa cez inštrukcie treba posúvať ručne, čo trvá niekoľko násobne dlhšie ako keď sa cez inštrukcie posúva sám systém. Vypočítaním času pred a po vykonaní istej časti kódu je možné zistiť, či je program krokovaný. Na výpočet času je napríklad možné použiť inštrukciu `RDTSC` alebo iné systémové funkcie ako `GetTickCount`.

Virtualizácia

Veľmi rozšírený mechanizmus v súčastnosti, nakoľko je ho najťažšie obísť. V skomprimovanom súbore sa istá časť inštrukcií pretransformuje na inštrukcie virtuálneho stroja, ktorý sa spustí pri spustení tohto programu a interpretuje tieto inštrukcie [19]. Jedna interpretovaná inštrukcia vyžaduje vykonať niekoľko stoviek až tisícok inštrukcií skutočného procesoru. Virtualizácia sa často vyskytuje na dvoch miestach. Buďto je časť, prípadne celá dekompresná rutina interpretovaná virtuálnym strojom, alebo sú časti pôvodného spustiteľného súboru prevedené do inštrukcií virtuálneho stroja, ktorý počas behu programu tieto inštrukcie interpretuje. Má to síce vplyv na rýchlosť programu, avšak za cenu zvýšenej bezpečnosti. Odstrániť takúto ochranu je komplikovanejšie ako u predošlých mechanizmov. Skutočné inštrukcie sa nikdy v pamäti nenachádzajú a bez virtuálneho stroja nemajú transformované inštrukcie žiadnu hodnotu. Je nutné zanalyzovať virtuálny stroj, ako funguje, v akom formáte sú jeho inštrukcie a akým spôsobom sa mapujú na inštrukcie reálneho procesoru.

4.4 Existujúce generické unpackery

Generické alebo aj univerzálne unpackery sú taký druh unpackerov, ktoré sú schopné prispôsobiť sa použitému packeru. Nie sú teda určené pre jeden konkrétny packer, ale pre určitú množinu. Nakoľko packery sú odlišné v tom ako docielia kompresiu a skrytie implementácie v spustiteľnom súbore, nie je dosiahnuť univerzálnosti unpackeru jednoduchá úloha. Stále vznikajú nové packery, ktoré fungujú na odlišných princípoch, preto je nutné aj generický unpacker stále vyvíjať.

Spôsob dosiahnutia univerzálnosti unpackeru nie je jednotná, záleží od návrhu a spôsobu implementácie konkrétneho generického unpackeru. Väčšinou sa však používa systém zásuvných modulov—pluginov, pričom jednotlivé zásuvné moduly predstavujú jednoúčelové unpackery.

Na nasledujúcich riadkoch sú opísané niektoré z existujúcich generických unpackerov.

- **FUU** *(Faster Universal Unpacker)*[3] — Open-source aplikácia určená pre Windows. Používá systém zásuvných modulov, ktoré využívajú spoločnú knižnicu TitanEngine pre manipuláciu so spustiteľným súborom. Je napísaný kompletne v MASM vrátane zásuvných modulov. Niekoľko zásuvných modulov je dodávaných priamo so zdrojovými kódmi jadra unpackeru, avšak jedná sa o dynamické unpackery. Posledné zmeny sú z roku 2011, takže už dlhšiu dobu nie je vyvíjaný.

- **PackerBreaker**[4] — Taktiež určený pre Windows, avšak bez dostupných zdrojových kódov. Obsahuje interný systém detekcie a dekompresie, ktorý nie je verejne prístupný. Používa emuláciu na dekompresiu. Aj napriek tomu, že je vyvíjaný interne, tak obsahuje veľkú škálu unpackerov. Sám obsahuje rôzne ochrany proti reverznému inžinierstvu, takže nemôže byť súčasne spustený spolu s diassemblermi, debuggermi a inými nástrojmi na monitorovanie procesov. Toto vyvoláva veľa kontroverzie v komunite, pretože človek nemá istotu, čo všetko sa deje na pozadí. Posledná verzia vyšla v roku 2012.

- **Interné unpackery antivírusových jadier** — Antivírusové spoločnosti vyvíjajú vlastné unpackery používané v antivírusových jadrách na detekciu malwaru. Tieto unpackery nie sú navonok viditeľné a fungujú len na pozadí antivírusových programov. Spoločnosť AVG Technologies má taktiež vlastný interný unpacker, ktorý bol zapožičaný pre účely tejto práce a pre možné vyhodnotenie výsledkov.

[3] https://code.google.com/p/fuu/
[4] http://www.sysreveal.com/tag/packerbreaker/

Kapitola 5

Analýza packerov

Na vytvorenie unpackeru pre istý packer je nutné vedieť ako daný packer funguje. Je potrebné vykonať analýzu samotnej dekompresnej rutiny, aby ju neskôr bolo možné zostaviť v unpackery. V nasledujúcich podkapitolách sú rozoberané niektoré z existujúcich packerov. U týchto packerov je vysvetlené akým spôsobom dosiahnu kompresiu, ako vyzerajú konkrétne dátové štruktúry nachádzajúce sa priamo v skomprimovanom súbore a ako vykonávajú dekompresiu. Na záver je popísaná dekompresná rutina a jej spôsob fungovania.

Všetky packery boli analyzované na architektúre x86. V prípade, že sa píše o formáte PE je analýza vykonávaná na systéme Windows. V prípade formátu ELF sa jedná o systém Linux. Analyzované vzorky boli všetky 32-bitové.

5.1 MPRESS

MPRESS (*Matcode comPRESSor*) [5] je voľne dostupný closed-source packer od spoločnosti MATCODE Software. Podporuje kompresiu formátov PE32, PE32+ a spustiteľné súbory pre platformu .NET. Cieľová architektúra tohto packeru je výhradne x86. Nakoľko zdrojové kódy packeru nie sú dostupné, je nutné všetku analýzu vykonať pomocou reverzného inžinierstva.

5.1.1 Kompresia

Kompresné algoritmy

MPRESS používa dva kompresné algoritmy LZMA [3] a LZMAT [4]. Oba sú z rodiny kompresných algoritmov LZ, ktoré fungujú na báze slovníkovej kompresie dát [20]. Algoritmus LZMAT sa použije v prípade malých súborov (rádovo desiatky kilobajtov), alebo v prípade že je explicitne vyžiadaný. Vo zvyšných prípadoch sa použije LZMA.

Štruktúra súboru

MPRESS komprimuje všetky sekcie okrem pôvodnej PE hlavičky a tzv. *resources*, čo sú zdrojové dáta programu, ktoré nepredstavujú kód, ale napríklad obrázky, fonty, kurzory a iné. V novovzniknutom súbore sa vždy nachádzajú minimálne dve sekcie, a to `.MPRESS1` a `.MPRESS2`. V prípade, že pôvodný súbor obsahoval *resources*, tak sa v ňom nachádza ešte sekcia zvyčajne nazývaná `.rsrc`. Kompresia pôvodného súboru sa vykoná tak, že pôvodný súbor je najskôr namapovaný tak, ako by vyzeral po spustení v pamäti. Tieto namapované dáta sú skomprimované jedným z algoritmov. Tieto dáta sú potom umiestnené do sekcie

.MPRESS1, ktorej je nastavená virtuálna veľkosť (veľkosť v pamäti) dekomprimovaných dát. Sekcia .MPRESS2 obsahuje dekompresnú rutinu, teda je do nej umiestnený vstupný bod programu. Tabuľku hlavičiek sekcií skomprimovaného súboru je možné vidieť na ukážke kódu 5.1. Táto informácia bola zistená pomocou nástroja objdump[1].

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .MPRESS1      00005000  00401000  00401000  00000200  2**2
                  CONTENTS, ALLOC, LOAD, CODE, DATA
  1 .MPRESS2      00000c0c  0042d000  0042d000  00005200  2**2
                  CONTENTS, ALLOC, LOAD, CODE, DATA
  2 .rsrc         000001d8  0042e000  0042e000  00006000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

Kód 5.1: Štruktúra súboru skomprimovaného packerom MPRESS.

Filtrovanie skokov

Pred samotnou kompresiou pôvodných dát súboru sú vykonávané modifikácie, ktoré sú neskôr vyriešené počas dekompresie popísanej ďalej v podkapitole 5.1.2. Jednou z týchto modifikácií je aj filtrovanie skokov. Presnejšie sa jedná o modifikáciu operandov inštrukcií JMP a CALL z dôvodu dosiahnutia lepšej kompresie. Ako príklad si zoberme nasledovný kód na ukážke 5.2. Je na nej možné vidieť volanie rovnakej funkcie na adresách 0x0041A393 a 0x0041A3A6. Nakoľko u architektúry x86 inštrukcie JMP a CALL používajú relatívnu vzdialenosť od adresy kam sa vykonáva skok, tak majú rozdielne operandy. V prípade slovníkovej kompresie je najlepšie, ak komprimované dáta obsahujú čo najdlhšie sekvencie opakujúcich sa bajtov. V tomto prípade by kompresný algoritmus vyhodnotil najdlhšiu spoločnú sekvenciu 8 bajtov 0x6FFFFF83C4040FB6. MPRESS vykoná filtrovanie tak, že od operandov inštrukcií JMP a CALL odpočíta ich vzdialenosť od začiatku sekcie. V prípade ukážky 5.2 sa nachádza začiatok sekcie na adrese 0x410000. Po pripočítaní 0xA394 (prvý bajt zaberá operačný kód inštrukcie CALL) k prvému operandu a 0xA3A7 k druhému operandu dostávame rovnaký výsledok 0x134E. Tento výsledok je zapísaný namiesto operandov a najdlhšia opakujúca sa sekvencia narástla v dĺžke na 10 bajtov, čo zlepšilo vlastnosti dát na ich kompresiu.

```
0041A392    50              push eax
0041A393    E8 BA6FFFFF     call 00411352
0041A398    83C4 04         add esp,4
0041A39B    0FB6C8          movzx ecx,al
0041A39E    85C9            test ecx,ecx
0041A3A0    75 1C           jnz short 0041A3BE
0041A3A2    8B55 E0         mov edx,dword ptr ss:[ebp-20]
0041A3A5    52              push edx
0041A3A6    E8 A76FFFFF     call 00411352
0041A3AB    83C4 04         add esp,4
0041A3AE    0FB6C0          movzx eax,al
```

Kód 5.2: Nefiltrované skoky.

[1] http://www.gnu.org/software/binutils/

Modifikácia tabuľky importovaných symbolov

Tabuľka importovaných symbolov je kompletne zmazaná predtým ako je vytvorený komprimovaný súbor. Namiesto toho, MPRESS si vytvorí vlastnú tabuľku importovaných symbolov v ktorej si nechá naimportovať len dôležité funkcie na opravu importov. Táto metóda bola podrobnejšie popísaná v podkapitole 4.1.1. Pôvodná tabuľka importovaných symbolov je zostrojená pri štarte skomprimovaného programu z pomocných dát, ktoré vloží MPRESS do komprimovaného obsahu spolu s rutinou, ktorá vie tieto pomocné dáta pretransformovať na IAT. Oboje musia byť vložené do komprimovaného obsahu tak, aby nijakým spôsobom nenarušili obsah pôvodného súboru. Výber miesta prebieha tak, že je preskúmaná každá hranica dvoch susedných sekcií. Ak je medzi dvoma susednými sekciami priestor pre pomocné dáta a aj rutinu zároveň, tak sú sem vložené oboje. Ak je priestor len pre jedno z nich, tak sa sem vloží a pre druhé je hľadané ďalšie umiestnenie rovnakým spôsobom. Ak nie je medzi žiadnymi dvoma sekciami dostatok miesta, tak sú tieto dáta vložené na koniec pôvodného obsahu.

Pomocné dáta pre importovanie symbolov pripomínajú svojou štruktúrou ILT, avšak sú mierne odlišné. Štruktúru týchto dát je možné vidieť vyjadrenú pomocou pseudojazyka C na ukážke kódu 5.3. Jedná sa o niekoľko štruktúr `mpressImportLibrary` sekvenčne za sebou, pre každú knižnicu z ktorej sa importuje jedna. Zakončené sú prvkom, ktorý má člen `iatDistance` nastavený na hodnotu `-1`. Tento člen inak predstavuje vzdialenosť od predošlého záznamu v IAT do aktuálneho záznamu v IAT. Pre prvý záznam je to vzdialenosť od rutiny pre opravu importovaných symbolov. Každý záznam typu `mpressImportLibrary` obsahuje v sebe niekoľko záznamov typu `mpressImportSymbol`, ktorý predstavuje jeden importovaný symbol z danej knižnice. Ako bolo vysvetlené v podkapitole 3.3.5, importovať sa môže buď pomocou názvu symbolu, alebo pomocou ordinálu. MPRESS rozlišuje medzi nimi pomocou prvého bajtu, ktorý ak má hodnotu menšiu alebo rovnú ako `0x20` (jedná sa o biele znaky), tak ide o import ordinálom, inak import názvom.

```
union mpressImportSymbol
{
    struct
    {
        uint8_t hint;
        uint16_t ordinal;
    }
    char symbolName[];
}

struct mpressImportLibrary
{
    int32_t iatDistance;
    char libraryName[];
    union mpressImportSymbol importedSymbols[];
}
```

Kód 5.3: Štruktúra pomocných dát pre importovanie symbolov packeru MPRESS v pseudojazyku C.

Použité ochrany proti reverznému inžinierstvu

MPRESS neslúži zrovna ako ochrana spustiteľných súborov, skôr len ako ich kompresia. Používa však samomodifikujúci kód pričom si mení inštrukcie skokov pri vykonávaní rôznych činností. Fakt, že rutina pre rekonštrukciu importovaných symbolov sa nenachádza priamo v sekcií `.MPRESS2` spolu s dekompresnou rutinou môžeme tiež brať ako formu samomodifikujúcehosa kódu.

5.1.2 Dekompresia

Dekompresia je započatá v dekompresnej rutine nachádzajúcej sa v sekcií `.MPRESS2`. Existujú niekoľko možných verzií dekompresnej rutiny naprieč rozličnými verziami packeru MPRESS, avšak postupnosť krokov, ktoré vykonávajú je pre každú rovnaká.

1. Presun dát sekcie `.MPRESS1` zo začiatku na jej koniec, aby sa zamedzilo ich prepísaniu.
2. Dekompresia týchto dát na začiatok sekcie `.MPRESS1`.
3. Presmerovanie riadenia programu do rutiny na rekonštrukciu tabuľky importovaných symbolov v sekcií `.MPRESS1`.
4. Rekonštrukcia tabuľky importovaných symbolov a vyriešenie relokácií ak je to potrebné.
5. Presmerovanie riadenia programu na OEP.

Existujú celkovo tri typy rutín pre rekonštrukciu tabuľky importovaných symbolov naprieč všetkými verziami, avšak líšia sa len použitím iných inštrukcií a poradím, či sa skorej rekonštruujú importované symboly alebo relokácie. Informácie o tom či sa v pôvodnom spustiteľnom súbore nachádzali relokácie, alebo nie je umiestnená spolu s ďalšími informáciami za dekompresnú rutinu. Dáta, ktoré sa tu nachádzajú je možné vyjadriť štruktúrou v pseudojazyku C, ktorú je možné vidieť na ukážke kódu 5.4.

```
struct mpressData
{
    int32_t packedContentOffset;
    int32_t relocationsOffset;
    int32_t sizeOfCompressedData;
    int32_t sizeOfRelocations;
}
```

Kód 5.4: Štruktúra dát pre dekompresiu packeru MPRESS v pseudojazyku C.

Člen `packedContentOffset` obsahuje vzdialenosť od vstupného bodu programu do skomprimovaných dát. Tieto dáta sa však vždy nachádzajú na začiatku sekcie `.MPRESS1`. Pri rekonštrukcií relokácií sa používajú členy `relocationsOffset` a `sizeOfRelocations`. Člen `relocationsOffset` predstavuje vzdialenosť od rutiny pre rekonštrukciu importovaných symbolov do *relocation data directory*. *Data directory* je popísané v podkapitole 3.3.3. Veľkosť komprimovaných dát je zapísaná v `sizeOfCompressedData`.

5.2 UPX

UPX je volne dostupný open-source packer s podporou veľkej škály formátov spustiteľných súborov ako PE, ELF či Mach-O. Podporuje taktiež aj iné architektúry ako x86, napríklad ARM, MIPS, alebo PowerPC. Je veľmi rozšírený medzi malwarmi z dôvodu jeho jednoduchej modifikácie vďaka dostupným zdrojovým kódom. Dodáva sa spolu s unpackerom, ktorý však dokáže dekomprimovať len nemodifikované verzie UPX.

5.2.1 Kompresia

Kompresné algoritmy

UPX používa na kompresiu súborov buďto algoritmus LZMA alebo algoritmy NRV v ich open-source implementácií UCL[2]. Presnejšie sa jedná o algoritmy NRV2B, NRV2D a NRV2E. Štandardne sa používajú algoritmy NRV, ktorých výber záleží na veľkosti komprimovaných dát. Algoritmus LZMA je možné použiť len ak je explicitne vyžiadaný. Aj napriek dostupným zdrojovým kódom je nutné použiť reverzné inžinierstvo na analýzu z dôvodu, že dekompresia štandardným unackerom UPX funguje inak ako dekompresia v dekompresnej rutine.

Štruktúra súboru (PE)

Súbory vo formáte PE komprimuje UPX podobne ako MPRESS. Komprimovaný je všetok obsah, tentokrát však aj spolu s pôvodnou PE hlavičkou. Tá je ale premiestnená zo začiatku na koniec dát. Komprimovaný súbor obsahuje minimálne trie sekcie. Dve z nich sú vždy `UPX0` a `UPX1`. V prípade prítomnosti *resources* v pôvodnom súbore, je tretia sekcia zvyčajne nazývaná `.rsrc`. V ostaných prípadoch sa nazýva `UPX2` a obsahuje importované symboly potrebné dekompresnou rutinou. Sekcia `UPX0` neobsahuje vôbec žiadne dáta a má nastavenú iba virtuálnu veľkosť. Slúži ako sekcia, do ktorej je neskôr vykonaná dekompresia. Komprimovaný obsah pôvodného súboru je umiestnený na začiatok sekcie `UPX1` nasledovaný dekompresnou rutinou. Komprimovaný je obsah ako by vyzeral po namapovaní do pamäte pri spustení pôvodného programu. Sekcie v súbore skomprimovanom s použitím UPX je možné vidieť na ukážke 5.5. Tieto informácie boli zistené pomocou nástroja `objdump`.

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 UPX0          0001b000  00401000  00401000  00000400  2**2
                  CONTENTS, ALLOC, CODE
  1 UPX1          00002000  0041c000  0041c000  00000400  2**2
                  CONTENTS, ALLOC, LOAD, CODE, DATA
  2 .rsrc         00000400  0041e000  0041e000  00002400  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

Kód 5.5: Štruktúra súboru vo formáte PE skomprimovaného packerom UPX.

Štruktúra súboru (ELF)

Pri kompresií súboru vo formáte ELF dochádza ku kompresií kompletne všetkých informácií. Žiadne dáta nie sú z pôvodného súboru zmazané. Segmenty z pôvodného súboru sú

[2] http://www.oberhumer.com/opensource/ucl/

všetky skomprimované, nový súbor obsahuje len 2 segmenty typu `LOAD` ako je možné vidieť na ukážke 5.6. Tento výstup bol získaný pomocou nástroja `objdump`.

```
Program Header:
    LOAD off    0x00000000 vaddr 0x00c01000 paddr 0x00c01000 align 2**12
         filesz 0x0003bef7 memsz 0x0003bef7 flags r-x
    LOAD off    0x00000624 vaddr 0x080f1624 paddr 0x080f1624 align 2**12
         filesz 0x00000000 memsz 0x00000000 flags rw-
```

Kód 5.6: Štruktúra súboru vo formáte ELF skomprimovaného packerom UPX.

Prvý z týchto segmentov obsahuje dekompresnú rutinu. U druhého je možné si všimnúť, že jeho veľkosť v súbore a aj veľkosť v pamäti sú nastavené na hodnotu `0`. Tento segment slúži len ako príprava miesta v pamäti, kam bude dekomprimovaný obsah umiestnený. Priestor tejto časti pamäti bude neskôr navýšený na požadovanú hodnotu.

Súbor vo formáte ELF je komprimovaný do tzv. blokov. Jeden blok je tvorený segmentom typu `LOAD` a nasledujúcimi segmentami až po ďalší `LOAD` segment. Výnimkou je len prvý blok, ktorý obsahuje komprimovanú ELF hlavičku pôvodného súboru. Každý blok nesie informáciu o tom, aký kompresný algoritmus bol použitý v danom bloku, aká je veľkosť komprimovaných a dekomprimovaných dát a ďalšie dodatočné informácie. Blok je možné popísať štruktúrou v pseudojazyku C ako je vidieť na ukážke kódu 5.7.

```c
struct upxBlock
{
    uint32_t sizeOfDecompressedData;
    uint32_t sizeOfCompressedData;
    uint8_t compressionAlgorithm;
    uint8_t jumpFilterIndex;
    uint8_t jumpFilterParameter;
    uint8_t unkAlways0;
    uint8_t compressedData[];
}
```

Kód 5.7: Štruktúra bloku packeru UPX pod formátom ELF.

Bloky sú umiestňované sekvenčne za sebou od konca PE hlavičky až po začiatok dekompresnej rutiny. Je však potrebné, aby sa zachovali aj informácie o sekciách alebo o názvoch sekcií, ktoré priamo nie sú potrebné pre beh programu. Tie sú taktiež skomprimované a umiestnené za dekompresnú rutinu sekvenčne za sebou.

Filtrovanie skokov

UPX taktiež filtruje inštrukcie skokov pre dosiahnutie lepšej kompresie dát rovnakým spôsobom ako MPRESS. Tento mechanizmus je popísaný u packeru MPRESS v podkapitole 5.1. UPX však obsahuje niekoľkonásobne viac filtrov, ktoré sa netýkajú len inštrukcií `JMP` a `CALL`, ale aj inštrukcií podmieneného skoku `Jcc`, do ktorých patria napríklad inštrukcie `JZ`, `JNZ` a veľa iných. Tým, že UPX podporuje aj viac architektúr musí mať aj filtre pre rôzne architektúry. Použitý filter musí byť zaznamenaný v metadátach, ktoré UPX pri kompresií doplní do súboru. U formátu ELF sú v hlavičke bloku.

Modifikácia tabuľky importovaných symbolov (PE)

U súborov vo formáte PE dochádza k ponechaniu jedného importovaného symbolu z každej knižnice, z ktorej sa importuje aspoň 1 symbol. UPX pred komprimovaním obsahu pôvodného súboru doňho vloží pomocné dáta nesúce informácie o importovaných symboloch, ktoré je potrebné rekonštruovať. Tieto dáta sú skomprimované a po dekompresií lokalizované dekompresnou rutinou. Ich štruktúru popísanú pomocou pseudojazyka C je možné vidieť na ukážke kódu 5.8.

```
struct upxImportSymbol
{
    uint8_t hint;
    union
    {
        uint16_t ordinal;
        char symbolName[];
    }
}

struct upxImportLibrary
{
    int32_t libraryNameOffset;
    int32_t iatOffset;
    struct upxImportSymbol importedSymbols[];
}
```

Kód 5.8: Štruktúra pomocných dát pre importovanie symbolov packeru UPX v pseudojazyku C.

Jedná sa o niekoľko štruktúr typu `upxImportLibrary` umiestnených sekvenčne za sebou ukončených prvkom s hodnotou `0` v člene `libraryNameOffset`. Ten inak obsahuje vzdialenosť od začiatku ILT v sekcií `UPX1` po reťazec obsahujúci názov knižnice, z ktorej je potrebné naimportovať symboly. Člen `iatOffset` obsahuje vzdialenosť od začiatku sekcie `UPX0`, kde sa nachádza IAT pre danú knižnicu. Nasleduje zoznam importovaných symbolov, ktoré sú reprezentované prvkami typu `upxImportSymbol`. Import ordinálom alebo názvom je rozoznaný na základe členu `hint`. Ak obsahuje hodnotu menšiu ako `0x80`, tak sa jedná o import názvom.

TLS Callback (PE)

Pre UPX *TLS callback* znamená, že sa musí postarať o to, aby *TLS callbacky* spúšťajúce sa pred štartom pôvodného programu boli spúšťané po ukončení činnosti dekompresnej rutiny a pred predaním riadenia programu na OEP. UPX toto podporuje a rieši to pomocou jedného *TLS callbacku*, ktorý si vloží do skomprimovaného súboru. Ten v sebe obsahuje kód, ktorý siahne do dekomprimovaných dát a postupne preiteruje všetky pôvodné *TLS callbacky* a zavolá ich. Je však potrebné, aby sa táto činnosť vykonala po dekompresií. Preto má na svojom začiatku jednu inštrukciu `JMP`, ktorá preskočí všetok kód *callbacku*. Táto inštrukcia je v dekompresnej rutine modifikovaná, aby neskočila až na koniec, ale iba na nasledujúcu inštrukciu. *TLS callback* je následne explicitne zavolaný, čo spôsobí spustenie všetkých pôvodných *TLS callbackov*.

Použité ochrany proti reverznému inžinierstvu

UPX používa samomodifukujúci kód. U formátu ELF je to taktiež kompresia kompletne všetkých pomocných dekompresných rutín. Na istých miestach je možné nájsť aj vkladanie bajtov priamo do inštrukcií.

5.2.2 Dekompresia

Nakoľko sa dekompresná rutina, a celkovo dekompresia súboru komprimovaného pomocou UPX pod formátmi PE a ELF výrazne líši, preto budú jednotlivé formáty popísané na nasledujúcich riadkoch samostatne.

PE

U formátu PE sa dekompresná rutina skladá zo základu, ktorý predstavuje samotný dekompresný algoritmus a ďalších pomocných dekompresných rutín riešiacich činnosti ako rekonštrukcia importovaných symbolov, relokácie, rekonštrukcia TLS, či odfiltrovanie skokov. Tie sú umiestnené a vykonávané postupne za sebou až pokiaľ nie je predané riadenie programu na OEP. Informácie o tom, aký filter bol použitý, aký kompresný algoritmus, veľkosť komprimovaných a dekomprimovaných dát a iné sú zapísané v UPX metadátach, ktoré nie sú namapované do žiadnej sekcie. Nachádzajú sa len v samotnom súbore bezprostredne za PE hlavičkou. Tieto metadáta nie sú vôbec dekompresnou rutinou čítané, nakoľko už celá dekompresná rutina je prispôsobená, aby dekomprimovala dáta bez týchto informácií. Používané sú len štandardným unpackerom UPX.

ELF

Formát ELF obsahuje v dekompresnej rutine len dekompresný algoritmus. Ten najskôr zoberie prvý blok za dekompresnou rutinou a dekomprimuje ho na prvú dostupnú adresu za dekompresnú rutinu. Prvý blok totiž obsahuje skomprimované ďalšie pomocné dekompresné rutiny, ktoré riešia napríklad odfiltrovanie skokov a pod. Riadenie je presmerované do týchto pomocných rutín, ktoré začnú postupne blok po bloku dekomprimovať ich obsah, pričom využívajú dekompresný algoritmus v hlavnej dekompresnej rutine. Informácie o použitom filteri alebo o kompresnom algoritme sú zapísané v hlavičke bloku. Počas behu programu nedôjde k dekompresií blokov za dekompresnou rutinou. Štandardný unpacker UPX však dekomprimuje aj tento obsah, aby obnovil pôvodnú štruktúru programu.

Kapitola 6

Návrh generického unpackeru

Obsah tejto kapitoly je klasifikovaný ako utajený, viz licenčné ujednanie.

Kapitola 7

Implementácia generického unpackeru

Obsah tejto kapitoly je klasifikovaný ako utajený, viz licenčné ujednanie.

Kapitola 8

Testovanie a výsledky

Táto kapitola rozoberá testovanie a prezentuje výsledky generického unpackeru a jeho zásuvných modulov, ktorého implementácia je popísaná v kapitole 7. Testovanie sa zameriava hlavne na úspešnosť zásuvných modulov. Najskôr sú popísané spôsoby testovania a vyhodnocovanie tohto testovania. Následne sú popísané jednotlivé testy.

8.1 Špecifikácia testov

Testovanie je vykonávané nad sadou spustiteľných súborov vytvorených špeciálne na tieto účely, alebo súbormi tretej strany stiahnutými z databáze portálu VirusTotal[1]. V prípade, že je to možné, sú výsledky generického unpackeru zrovnané s inými generickými unpackermi na rovnakej testovacej sade. Skúmané sú 3 hlavné metriky.

1. **Úspešnosť dekompresie** — Koľko komprimovaných súborov bolo úspešne dekomprimovaných v pomere s počtom všetkých komprimovaných súborov v testovacej sade. Ako úspešná dekompresia je braná taká, pri ktorej nedošlo k žiadnej chybe pri behu unpackeru.

2. **Spustiteľnosť výstupov** — Počet dekomprimovaných súborov, ktoré zostali spustiteľné aj po dekompresií voči počtu úspešne dekomprimovaných súborov.

3. **Dekompilovatelnosť výstupov** — Počet dekompilovateľných súborov voči počtu úspešne dekomprimovaných súborov. Úspešná dekompilácia sa berie taká, pri ktorej nepríde k žiadnej chybe a vo výstupe bude rozpoznateľný pôvodný kód (reťazce, konštanty, volania systémových funkcií atď.)

Výsledky sú vyhodnotené vo forme percent. Testovanie prebieha výlučne na architektúre x86 pod systémami Windows 8.1 64-bit a Linux Fedora 21 Workstation x64.

8.2 Testovanie zásuvných modulov

V tejto časti sú popísané testy a ich výsledky jednotlivých zásuvných modulov popísaných v kapitole **??**.

[1] https://www.virustotal.com/

8.2.1 MPRESS

Testovanie heuristických analýz

Na otestovanie funkčnosti heuristických analýz popísaných v podkapitole **??** bolo vytvorených niekoľko súborov s rôznym usporiadaním kódových a dátových sekcií. Minimalistické vzorky boli vytvorené s použitím prekladača assembleru x86 FASM[2]. Posledná vzorka bola vytvorená pomocou prekladača Microsoft Visual C++. Vzorky boli dekompilované spätným prekladačom pre možné porovnanie výsledkov. Následne boli skomprimované packerom MPRESS vo verzií 2.19. Celkovo sú testované 4 varianty.

1. Kódová sekcia sa nachádza pred dátovými sekciami.
2. Kódová sekcia sa nachádza za dátovými sekciami.
3. Kódová sekcia sa nachádza medzi dátovými sekciami.
4. Sekcie sú vyprodukované prekladačom použitým v praxi a reflektujú možné usporiadanie u vzoriek v praxi.

Na tabuľke 8.1 je možné vidieť sekcie jednotlivých pôvodných súborov pre každú variantu. Najviac pozornosti je kladenej na sekciu `.text`, ktorá obsahuje OEP, ale hlavne celý kód programu. Cieľom tohto testu je overiť kvalitu heuristických analýz pri separácií kódu a dát, čo je veľmi dôležité pre dekompilovateľnosť. Test varianty je označený ako úspešný, pokiaľ sa podarí zrekonštruovať textovú sekciu tak, aby adresa a veľkosť rekonštruovanej sekcie sa rovnali pôvodným hodnotám. Veľkosť rekonštruovanej sekcie môže byť zarovnaná na najbližší násobok zarovnania sekcií určeného PE hlavičkou.

Tab. 8.1: Sekcie pôvodných súborov jednotlivých variánt testovania.

	Varianta 1	Varianta 2	Varianta 3	Varianta 4

Sekcie	`.text`
	`.data`
	`.idata`
		`.data`
		`.idata`
		`.text`
			`.data`
			`.text`
			`.idata`
				`.textbss`
				`.text`
				`.rdata`
				`.data`
				`.idata`
				`.rsrc`
				`.reloc`

Adresa sekcie `.text`	`0x401000`	`0x403000`	`0x402000`	`0x411000`
Veľkosť sekcie `.text`	`0x19`	`0x19`	`0x19`	`0x3BD9`

Jednotlivé súbory boli dekomprimované generickým unpackerom. Štruktúru dekomprimovaných súborov je možné vidieť na tabuľke 8.2.

Je možné si všimnúť, že u minimalistických vzoriek bola heuristická analýza celkom presná, čo sa týka počtu sekcií. U varianty 1 boli síce sekcie `.data` a `.idata` zjednotené do sekcie `.data0`, to však nevadí spätnému prekladača. Varianta 4 však nevyprodukovala

[2] http://flatassembler.net/

Tab. 8.2: Sekcie pôvodných súborov jednotlivých variánt testovania.

	Varianta 1	Varianta 2	Varianta 3	Varianta 4

Sekcie	`.text`
	`.data0`
	`.MPRESS2`
	`.imports`
		`.data0`
		`.data2`
		`.text`
		`.MPRESS2`
		`.imports`
			`.data0`
			`.text`
			`.data3`
			`.MPRESS2`
			`.imports`
				`.data3`
				`.data4`
				`...`
				`.data17`
				`.data18`
				`.text`
				`.data0`
				`.MPRESS2`
				`.rsrc`
				`.imports`

Adresa sekcie `.text`	`0x401000`	`0x403000`	`0x402000`	`0x411000`
Veľkosť sekcie `.text`	`0x1000`	`0x1000`	`0x1000`	`0x4000`

natoľko presnú rekonštrukciu všetkých sekcií. Sekcia `.textbss` bola rozdelená na 16 častí. Naopak, všetky sekcie za pôvodnou sekciou `.text` boli zjednotené do jedinej sekcie `.data0`. Cieľom je však overiť, či je správne ohraničená sekcia s kódom. U všetkých štyroch variant došlo k správnej rekonštrukcií sekcie `.text`.

Testovanie dekompilovateľnosti

V tomto teste sa testovacia sada skladá z 91 reálnych malware vzoriek. Iba 2 používajú kompresiu LZMAT, nakoľko sa jedná o kompresiu pre veľmi malé súbory. Zvyšné používajú kompresiu LZMA. Tieto vzorky sú taktiež dekomprimované interným unpackerom AVG z antivírusového jadra a generickým unpackerom PackerBreaker pre zrovnanie výsledkov. Výsledky je možné vidieť na tabuľke 8.3.

Tab. 8.3: Výsledky testovania zásuvného modulu pre packer MPRESS na reálnych malware vzorkách.

Použitý unpacker	Úspešnosť dekompresie	Spustiteľnosť výstupov	Dekompilovatelnosť výstupov
Generický unpacker	**93,41%**	**100,00%**	**100,00%**
Interný AVG unpacker	93,41%	100,00%	0,00%
PackerBreaker	93,41%	95,29%	0,00%

Všetky 3 unpackery dosiahli rovnakú úspešnosť dekompilácie. Nepodarilo sa im dekomprimovať rovnakých 6 vzoriek, ktoré boli preto analyzované ručne. Bolo zistené, že tieto vzorky síce obsahujú črty packeru MPRESS, ako názvy sekcií, avšak signatúry sa vôbec nezhodovali. Hlbšia analýza ukázala, že vzorky sú skomprimované pomocou packeru MPRESS, ale ešte nad ním je použitá jednoduchá kompresia, alebo obfuskácia. Manuálnym upravením

týchto vzoriek je možná ich dekompresia. Automatizovaná detekcia týchto vzoriek je však náročná a vyžadovala by dynamické emulačné techniky, preto je potrebné aj naďalej tieto vzorky analyzovať manuálne.

V spustiteľnosti výstupov predbehli PackerBreaker zvyšné unpackery. Výstupy PackerBreakeru nezostali všetky spustiteľné, kdežto generický unpacker a aj interný AVG unpacker zrekonštruovali výstupy tak, aby zostali spustiteľné.

U dekompilovateľnosti vedie generický unpacker, ktorého všetky výstupy boli dekompilovateľné. Ostatné unpackery ponechali dekomprimovaný obsah v obrovskej sekcií `.MPRESS1`, čo sa odzrkadlilo na výstupe spätného prekladača.

8.2.2 UPX

Testovanie dekompilovateľnosti

Testovanie UPX prebiehalo na umelo vytvorených spustiteľných súboroch. Na tieto súbory bol použitý štadardný packer UPX vo verzií 3.91. Testovanie jednotlivých formátov PE a ELF prebehlo samostatne kvôli rozdielnemu spôsobou fungovania.

PE

Testovacia sada sa skladá z 30 vzoriek. U 10 vzoriek bol použitý prekladač GCC, u zvyšných Microsoft Visual C++. Presne 15 vzoriek je skomprimovaných algoritmom LZMA a druhých 15 vzoriek niektorým z algoritmov NRV. Výber algoritmu NRV a filtru skokov je ponechaný na packer. Vzorky boli dekomprimované taktiež unpackermi PackerBreaker a interný AVG unpacker z antivírusového jadra. Výsledky testov je možné vidieť na tabuľke 8.4.

Tab. 8.4: Výsledky testovania zásuvného modulu pre packer UPX na vzorkách vo formáte PE.

Použitý unpacker	Úspešnosť dekompresie	Spustiteľnosť výstupov	Dekompilovatelnosť výstupov
Generický unpacker	**93,33%**	**26,67%**	**26,67%**
Interný AVG unpacker	66,67%	50,00%	0,00%
PackerBreaker	100,00%	66,67%	0,00%

Najlepšiu úspešnosť dekompresie dosiahol PackerBreaker, ktorý dekomprimoval všetky vzorky. Nevedel si však poradiť so vzorkami obsahujúcimi TLS, ktoré nezostali spustiteľné. Generický unpacker nevedel dekomprimovať 2 vzorky, ktoré neboli komprimované algoritmom LZMA ani NRV2B. Nízka spustiteľnosť výstupov je spôsobená tým, že nie všetky filtre skokov sú momentálne podporované. Tým pádom zostali poškodené skokové inštrukcie v dekomprimovanom súbore. Všetky súbory, ktoré používali podporovaný filter sa však podarilo dekompilovať. Interný AVG unpacker si nevedel rady so vzorkami, ktoré používali LZMA kompresiu a boli skompilované pomocou prekladača Microsoft Visual C++. Tieto vzorky sa mu nepodarilo ani dekomprimovať. Spustiteľnosť bola zachovaná len u súborov, ktoré vznikli pomocou prekladača GCC. Nakoľko PackerBreaker a ani interný AVG unpacker nerekonštruujú sekcie spustiteľného súboru, tak ich výstupy opäť neboli dekompilovateľné.

ELF

Testovacia sada obsahovala 16 vzoriek vytvorených prekladačmi GCC a Clang. V 10 vzorkách bol použitý algoritmus NRV, u zvyšku LZMA. Výber z druhov algoritmov NRV bol ponechaný na samotný packer. Výsledky unpackovania nemohli byť porovnané so žiadnym iným unpackerom, nakoľko neexistuje žiadny známy unpacker pre formát ELF. Výsledky je možné vidieť na tabuľke 8.5.

Tab. 8.5: Výsledky testovania zásuvného modulu pre packer UPX na vzorkách vo formáte ELF.

Použitý unpacker	Úspešnosť dekompresie	Spustiteľnosť výstupov	Dekompilovatelnosť výstupov
Generický unpacker	**100,00%**	**100,00%**	**100,00%**

Úspešnosť výsledkov je možné pripísať spôsobu, akým kompresia u formátu ELF funguje. Čím viac informácií je skomprimovaných, a naozaj používaných počas dekompresnej z dekompresnej rutiny, tým viac je dostupných informácií pre unpacker. Výsledky sú teda porovnateľné priamo so štandardným unpackerom UPX. V nasledujúcom teste je však ukázané, že fungujú inak.

Testovanie modifikovaných UPX metadát

U packeru UPX je testovaná schopnosť generického unpackeru dekompresie súboru bez prístupu k metadátam, ktoré si packer do súboru doplnil. Testy sú vykonávané nad formátom ELF, ktorý na týchto metadátach nemusí vôbec závisieť. Následne je ukázané, že štandardný unpacker UPX nedokáže takýto súbor dekomprimovať.

V testovacích súboroch bola vykonané manuálne zmeny v UPX metadátach. Tie odstránili reťazce `UPX!`, ktoré si samotné UPX kontroluje. Tieto zmeny je možné vidieť na ukážke 8.2 vo formáte `diff`. Červenou sú naznačené dáta v súbore pred modifikáciou, zelenou dáta po modifikácii. UPX pri kontrole metadát prehlási súbor ako nevalidný a odmietne ho dekomprimovať ako je vidieť na ukážke 8.1.

```
$ upx -d sample
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2013
UPX 3.91        Markus Oberhumer, Laszlo Molnar & John Reiser   Sep 30th
    ↪ 2013

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
upx: sample: NotPackedException: not packed by UPX

Unpacked 0 files.
```

Kód 8.1: Výsledok dekompresie modifikovaného súboru štandardným unpackerom UPX.

```
@@ -5,7 +5,7 @@
 0000040: 0010 c000 8fbf 0300 8fbf 0300 0500 0000  ................
 0000050: 0010 0000 0100 0000 2406 0000 2416 0f08  ........$...$...
 0000060: 2416 0f08 0000 0000 0000 0000 0600 0000  \$...............
-0000070: 0010 0000 ec54 e47e 5550 5821 1012 0d0c  .....T.~UPX!....
+0000070: 0010 0000 ec54 e47e 0000 0000 1012 0d0c  .....T.~........
 0000080: 0000 0000 1261 0b00 1261 0b00 f400 0000  .....a...a......
 0000090: 7c00 0000 0e00 0000 1a03 003f 9145 8468  |..........?.E.h
 00000a0: 3bde e816 f085 b31b 3c5f 93a9 e468 ad94  ;.......<_...h..
@@ -16562,7 +16562,7 @@
 0040b10: 7a85 e86b 77f5 a7b6 49d1 bf0a 75c0 42e0  z..kw...I...u.B.
 0040b20: 238d 0d31 b6d3 ee32 a3f1 13f2 ecff 1af5  #..1...2........
 0040b30: 1858 3c95 0887 8e1f 519b b42e 6388 cab5  .X<.....Q...c...
-0040b40: 783e 67a8 63a3 fc00 0000 0055 5058 2100  x>g.c......UPX!.
-0040b50: 0000 0000 5550 5821 0d0c 0e07 c0e6 0b99  ....UPX!........
+0040b40: 783e 67a8 63a3 fc00 0000 0000 0000 0000  x>g.c...........
+0040b50: 0000 0000 0000 0000 0d0c 0e07 c0e6 0b99  ................
 0040b60: 6cb8 4a62 2af0 0000 8e4b 0000 1261 0b00  l.Jb*....K...a..
 0040b70: 4927 004c 8000 0000                      I'.L....
```

Kód 8.2: Zmeny vykonané v UPX metadátach.

Takto modifikovaný súbor je nemožné dekomprimovať použitím štandardného UPX unpackeru. Generický unpacker však na týchto dátach nezávisí a súbor dekomprimuje.

Kapitola 9

Záver

V tejto práci boli diskutované princípy fungovania spustiteľných súborov, ich kompresia a dekompresia. Bolo vysvetlené, čo sú packery, unpackery a ako fungujú. Taktiež boli zanalyzované dva existujúce packery, MPRESS a UPX. Následne bol navrhnutý univerzálny nástroj na dekompresiu spustiteľných súborov nazývaný generický unpacker, ktorý bol implementovaný v jazyku C++ a integrovaný do rekonfigurovateľného spätného prekladača spoločnosti AVG. Do tohto generického unpackeru bola implementované podpora dekompresie pre analyzované packery. Generický unpacker podstúpil niekoľkým testom na overenie kvality jeho prevedenia. O tejto práci bol taktiež napísaný článok pre študentskú konferenciu Excel@FIT 2015, kde sa dostal medzi prezentované práce [17].

Novovytvorený generický unpacker používa systém zásuvných modulom, čím je umožnená jeho jednoduchá a rýchla rozšíriteľnosť. Jednotlivé zásuvné moduly, predstavujúce unpackery, implementujú statické dekompresné techniky, ktoré nespúšťajú spracovávaný súbor. Tým poskytujú bezpečnosť pred dekompresiou malwaru a taktiež platformovú a architekturálnu nezávislosť. Generický unpacker zároveň vypĺňa prázdne miesto medzi spätnými prekladačmi a unpackermi. Nesústredí sa len na dekompresiu samotnú, ale aj na rekonštrukciu štruktúry spustiteľných súborov do takej miery, aby boli dekompilovateľné, čo ostatné existujúce unpackery neposkytujú.

Dosahované výsledky v úspešnosti dekompresie u podporovaných packerov sú porovnateľné s ostatnými používanými unpackermi v praxi. V prípade packeru MPRESS sú dosiahnuté výsledky dokonca najlepšie. Statické dekompresné techniky dovolili otestovať generický unpacker dokonca na skutočných vzorkách malwaru, čím sa overila jeho schopnosť nasadenia v praxi. U packeru UPX, z dôvodu nekompletnej podpory všetkých kompresných algoritmov, sú výsledky priemerné. Nekompletná podpora nedovolila otestovať unpacker packeru UPX na reálnych vzorkách z praxe, pretože tento packer obsahuje v sebe veľké množstvo parametrov. Bolo by teda možné otestovať len vzorky, ktoré spĺňajú určité parametre a zohnať tieto vzorky nie je jednoduché.

V ďalšom vývoji je potrebné sa sústrediť na zväčšenie miery, do akej sú packery s ich parametrami podporované a aj otestované, pričom je potrebné sa sústrediť hlavne na testovanie na reálnych vzorkách. Postupne tiež budú pribúdať nové zásuvné moduly podporujúce nové packery. Čím náročnejšie packery budú podporované, tým náročnejšie však bude rekonštruovať dekompresnú rutinu a vykonávať len statickú dekompresiu. Jedným z riešení do budúcna môže byť implementácia emulátoru inštrukcií. Tento emulátor by slúžil ako kontrolované prostredie, v ktorom by mohol byť spúšťaný aj škodlivý kód. Skomprimované vzorky by mohli byť emulované a obraz ich pamäte by sa dal použiť na zostrojenie dekomprimovaného súboru. Jednalo by sa teda o dynamickú metódu dekompresie, avšak

v riadenom prostredí. Ďalším možným vylepšením môže byť napríklad zjednotenie internej knižnice `fileformatl` na zisťovanie informácií o spustiteľnom súbore s knižnicou `unpackerl` a vytvorením jeden veľkej knižnice. Tá by umožňovala ako zisťovať informácie, tak aj modifikovať a vytvárať spustiteľné súbory. Aktuálna implementácia sa taktiež sústredí hlavne na architektúru x86, čím nie sú pokryté všetky architektúry podporované rekonfigurovateľným spätným prekladačom. V budúcnosti je plánované rozšíriť podporu aj na zvyšné architektúry ARM, MIPS, PowerPC atď.

V júni 2015 bude vydaná verzia 2.0 rekonfigurovateľného spätného prekladača. Generický unpacker bude jej súčasťou, čím si užívatelia budú môcť sami vyskúšať vlastné skomprimované spustiteľné súbory. To prinesie výhodu aj do vývoja, nakoľko generický unpacker nemá všetky svoje časti otestované na reálnych vzorkách. Spätná väzba od užívateľov vylepší testovanie a zlepší tak celkovú kvalitu unpackeru.

Literatura

[1] LLVM Language Reference Manual - LLVM 3.7 documentation. [cit. 2014-05-14].
URL http://llvm.org/docs/LangRef.html

[2] LLVM Tutorial 1: A First Function. [cit. 2014-12-18].
URL http://llvm.org/releases/2.6/docs/tutorial/JITTutorial1.html

[3] LZMA SDK (Software Development Kit). [cit. 2014-05-14].
URL http://www.7-zip.org/sdk.html

[4] LZMAT - Opensource extremely fast data compression library. [cit. 2014-05-14].
URL http://www.matcode.com/lzmat.htm

[5] MPRESS - Free high-performance executable packer for PE32/PE32+/.NET/MAC-OS-X. [cit. 2014-05-14].
URL http://www.matcode.com/mpress.htm

[6] Retargetable Decompiler. [cit. 2014-05-14].
URL https://retdec.com/

[7] Aho, A. V.; Lam, M. S.; Sethi, R.; aj.: *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006, ISBN 0321486811.

[8] Cifuentes, C.: *Reverse Compilation Techniques*. Dizertační práce, Queensland University of Technology, 1994.

[9] Eilam, E.: *Reversing: Secrets of Reverse Engineering*. Indianapolis, IN: Wiley Publishing, 2005, ISBN 978-0-7645-7481-8.

[10] International Telecommunication Union: The World in 2014: ICT Facts and Figures. [cit. 2014-05-14].
URL http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2014-e.pdf

[11] Křoustek, J.: *Rekonfigurovatelná analýza strojového kódu*. Dizertační práce, Fakulta informačních technologií VUT v Brně, 2014.

[12] Křoustek, J.; Kolář, D.: Preprocessing of Binary Executable Files Towards Retargetable Decompilation. In *8th International Multi-Conference on Computing in the Global Information Technology (ICCGI'13)*, International Academy, Research, and Industry Association, 2013, ISBN 978-1-61208-283-7, s. 259–264.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=10200

[13] Levine, J. R.: *Linkers and Loaders*. San Francisco: Morgan Kaufmann, 2000, ISBN 15-586-0496-0.

[14] Malin, C. H.; Casey, E.; Aquilina, J. M.: *Malware Forensics: Investigating and Analyzing Malicious Code*. Burlington, MA: Syngress, 2008, ISBN 978-1-59749-268-3.

[15] Matz M.; Hubička J.; Jaeger A.; Mitchell M.: *System V Application Binary Interface*. 2013.
URL http://www.x86-64.org/documentation/abi.pdf

[16] Microsoft Corporation: *Microsoft Portable Executable and Common Object File Format Specification*. 2013.
URL http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx

[17] Milkovič, M.: Generic Unpacker of Executable Files. In *Excel@FIT*, 2015.
URL http://excel.fit.vutbr.cz/2015/submissions/030/30.pdf

[18] Pietrek, M.: An In-Depth Look into the Win32 Portable Executable File Format. *MSDN Magazine*, February 2002.
URL http://msdn.microsoft.com/en-us/magazine/cc301805.aspx

[19] Rolles, R.: Unpacking Virtualization Obfuscators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, Berkeley, CA, USA: USENIX Association, 2009.
URL http://static.usenix.org/event/woot09/tech/full_papers/rolles.pdf

[20] Salomon, D.: *Data Compression: The Complete Reference*. Springer Science & Business Media, 2007.

[21] Schwarz, B.; Debray, S.; Andrews, G.: Disassembly of executable code revisited. In *Reverse engineering, 2002. Proceedings. Ninth working conference on*, IEEE, 2002, s. 45–54.

[22] Sikorski, M.; Honig, A.: *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012, ISBN 978-1-59327-290-6.

Príloha A

Deklarácie hlavičiek formátov spusiteľných súborov

```
typedef struct _IMAGE_DOS_HEADER
{
  WORD e_magic;
  WORD e_cblp;
  WORD e_cp;
  WORD e_crlc;
  WORD e_cparhdr;
  WORD e_minalloc;
  WORD e_maxalloc;
  WORD e_ss;
  WORD e_sp;
  WORD e_csum;
  WORD e_ip;
  WORD e_cs;
  WORD e_lfarlc;
  WORD e_ovno;
  WORD e_res[4];
  WORD e_oemid;
  WORD e_oeminfo;
  WORD e_res2[10];
  LONG e_lfanew;
} IMAGE_DOS_HEADER;
```

Kód A.1: Štruktúra IMAGE_DOS_HEADER.

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  WORD                 Magic;
  BYTE                 MajorLinkerVersion;
  BYTE                 MinorLinkerVersion;
  DWORD                SizeOfCode;
  DWORD                SizeOfInitializedData;
  DWORD                SizeOfUninitializedData;
  DWORD                AddressOfEntryPoint;
  DWORD                BaseOfCode;
  DWORD                BaseOfData;
  DWORD                ImageBase;
  DWORD                SectionAlignment;
  DWORD                FileAlignment;
  WORD                 MajorOperatingSystemVersion;
  WORD                 MinorOperatingSystemVersion;
  WORD                 MajorImageVersion;
  WORD                 MinorImageVersion;
  WORD                 MajorSubsystemVersion;
  WORD                 MinorSubsystemVersion;
  DWORD                Win32VersionValue;
  DWORD                SizeOfImage;
  DWORD                SizeOfHeaders;
  DWORD                CheckSum;
  WORD                 Subsystem;
  WORD                 DllCharacteristics;
  DWORD                SizeOfStackReserve;
  DWORD                SizeOfStackCommit;
  DWORD                SizeOfHeapReserve;
  DWORD                SizeOfHeapCommit;
  DWORD                LoaderFlags;
  DWORD                NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER;
```

Kód A.2: Štruktúra IMAGE_OPTIONAL_HEADER.

Příloha B

UML diagramy

Obsah tejto kapitoly je klasifikovaný ako utajený, viz licenčné ujednanie.