VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

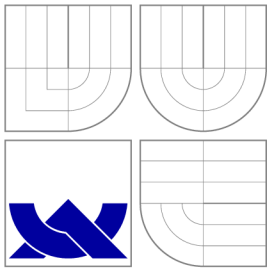
UŽITÍ TECHNIKY LÁMÁNÍ HESEL U KOMPRIMAČNÍCH FORMÁTŮ RAR, ZIP A 7Z A EXTRAKCE HESEL Z SAMOROZBALOVACÍCH ARCHIVŮ

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

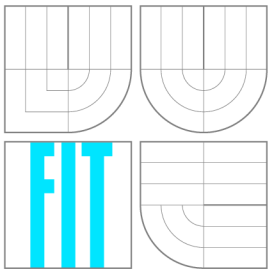
AUTOR PRÁCE
AUTHOR

MILAN PRUSTOMĚRSKÝ

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

UŽITÍ TECHNIKY LÁMÁNÍ HESEL U KOMPRIMAČNÍCH FORMÁTŮ RAR, ZIP A 7Z A EXTRAKCE HESEL Z SAMOROZBALOVACÍCH ARCHIVŮ

ANALYSIS OF THE POSSIBILITY OF PASSWORD BREAK THROUGH FOR RAR, ZIP AND 7Z FORMATS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — MILAN PRUSTOMĚRSKÝ
AUTHOR

VEDOUCÍ PRÁCE — prof. Ing. TOMÁŠ HRUŠKA, CSc.
SUPERVISOR

BRNO 2013

Abstrakt

Práce se zabývá užitím technik lámání hesel u běžných komprimačních formátů a extrakcí hesel ze samorozbalovacích archivů, které se využívají pro potřeby malware. Jsou zde popsány struktury komprimačních formátů, šifry a jejich propojení s komprimačními formáty. Zmíněny jsou i běžné a specializované útoky na archivy a jejich šifry. Následně jsou popsány struktury samorozbalovacích archivů a extrakce hesla, umožňující spuštění samorozbalovacího archivu.

Abstract

This Thesis deals with analysis of the possiblity of password breakthrough for common compression formats and password extraction from self-extraction archives used for malicious software. Structure of compression programs, ciphers and connection between cipher and archives is described. Common and specialized attacks on archives and ciphers are described. Structure of self-extracting archives and password location is used to create extractor of passwords in self-extracting archives..

Klíčová slova

RAR, Zip, 7z, komprimační program, samorozbalovací archiv, SFX, malware, extrakce, heslo, šifra, AES, PKWARE, útok, brute-force, dictionary útok

Keywords

RAR, Zip, 7z, compression program, self-extracting archive, SFX, malware, extraction, password, cipher, AES, PKWARE, attack, brute-force, dictionary attack

Citace

Milan Prustoměrský: Užití techniky lámání hesel u komprimačních formátů RAR, ZIP a 7z a extrakce hesel z samorozbalovacích archivů, diplomová práce, Brno, FIT VUT v Brně, 2013

Užití techniky lámání hesel u komprimačních formátů RAR, ZIP a 7z a extrakce hesel z samorozbalovacích archivů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana prof. Ing. Tomáše Hrušky, CSc.
Další informace a konzultace mi poskytl pan Jiří Kropáč z firmy AVG.

.....
Milan Prustoměrský
22. května 2013

Poděkování

Děkuji panu Tomáši Hruškovi za odborné vedení diplomové práce a panu Jiřímu Kropáčovi za poskytnutí odborné literatury a rad ohledně dané problematiky.

Obsah

Kapitola 1

Úvod

Komprimační formáty hrají v dnešním světě informačních technologií velkou roli. Lze díky nim výrazně zmenšit velikost archivovaných souborů a snížit tak velikost přenášených dat přes počítačové sítě či ušetři místo na pevných discích. Pomocí komprimačních programů je i možné příliš velké soubory rozdělit na několik menších, aniž by byla způsobena ztráta jakékoliv informace z původního souboru.

Přidáním bezpečnostních prvků umožníme přenos důvěryhodných informací. To může být zajištěno přístupem k archivu pouze na základě hesla či přímo zašifrováním celého obsahu archivu. Ovšem cokoliv zašifrované přitahuje pozornost útočníků, kteří chtějí znát původní obsah. Proto se v dnešní době pro ochranu archivů používají silné šifry, které se snaží prolomení bránit či jej alespoň odvrátit.

Útočníci však přišli na to, že zašifrované archivy mohou využít i pro své účely a podnikat pomocí nich různé útoky. Antivirové programy nedokáží rozpoznat signaturu infiltrovaného souboru a určit, zda-li daný soubor je malware[1] či nikoliv. Mohou pouze zjistit, že se jedná o zaheslovaný archiv, avšak nedokáží jej úspěšně otevřít.

Práce se soustředí především na zjištění a extrakci hesel ze samorozbalovacích archivů na platformě Windows, neboť tato platforma a typ archivu bývají nejčastěji používány pro malware. Nejprve jsou však popsány struktury dat nejčastějších komprimačních formátů a programů, které nesou všechny důležité informace o samorozbalovacích archivech. Následují popisy nejpoužívanějších šifer, kterými se šifrují dané formáty, a nejpoužívanější kompresní metody pro kompresi dat. Dále jsou rozebrány obecné i specifické útoky na zaheslované archivy i se simulací útoků na tyto archivy. Hlavní částí této práce je analýza samorozbalovacích archivů, díky které bylo možné z archivů vyextrahovat hesla. V závěru je pak celá práce shrnuta a jsou diskutovány možné budoucí rozšíření.

[1] Škodlivý program určený pro sběr citlivých dat, jejich modifikace či smazání.

Kapitola 2

Programová část

Soubor, představující archiv s komprimovanými daty, se skládá z mnoha částí, které se podílejí na správné reprezentaci obsažených dat. V případě samotného archivu je soubor popsán strukturou, která je pro každý archivační formát jedinečná. V této kapitole budou rozebrány struktury nejčastějších archivních formátů RAR, Zip a 7z. Existují i jiné archivy (ACE, TAR), avšak jejich četnost je výrazně menší než u předešlých tří archivačních formátů.

U samorozbalujících se archivů je situace jiná, takový soubor je složen ze dvou hlavních částí - první část představuje spustitelný kód, který je závislý na platformě, pro kterou je soubor určen. Druhá část pak obsahuje strukturu jako v případě samotného archivu. Bude zde popsána struktura spustitelného souboru na platformě Windows.

Krátce bude probrán i způsob detekce chování spustitelného kódu pomocí diasembleru a dekompilátoru, nemáme-li k dispozici žádnou dokumentaci či zdrojové kódy pro popis činnosti daného spustitelného kódu.

2.1 Zip formát

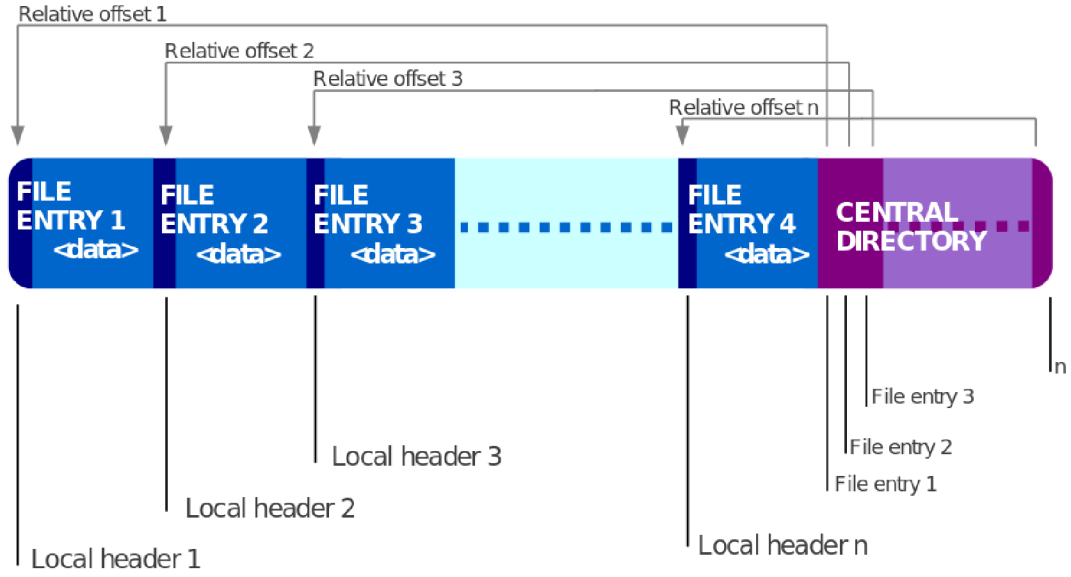
Soubor Zip se skládá ze dvou hlavních částí - seznamu položek a central directory, kde veškerá data ukládá v pořadí Little-Endian[1]

Jednotlivé položky v seznamu obsahují metadata ohledně souboru a samotný soubor, který může, ale nemusí být komprimován. Metadata obsahují např. velikost souboru, název souboru aj. (kompletní formát viz tabulka A.1). *Extra field* jsou určena pro rozšíření samotného Zip formátu. Je možné do něj uložit jakoukoliv upřesňující informaci, kterou autor Zip komprimujícího softwaru potřebuje. Právě zde se ukládají informace ohledně šifrování.

Pokud při kompresi při vytváření *local header* nejsou známy hodnoty CRC–32 a velikosti souboru (z jakéhokoliv důvodu - indikováno prvním bitem v poli *General purpose bit flag)*, jsou příslušná pole v *local header* vyplněny nulou a CRC–32 s velikostí souboru jsou připojena za komprimovaná data ve struktuře viz tabulka A.2.

V *central directory* (tabulka A.3) se nachází redundantní hlavičky jednotlivých položek (*local headers*) spolu s offsetem, kde přesně v souboru se daná položka nachází (*Relative offset*). Díky tomu lze Zip soubor rychle zpracovat a nenačítat tak samotné komprimované soubory. Za poslední hlavičkou se ještě nachází ukončovací hlavička *central directory* (tabulka A.4), která obsahuje informace pro dekompresi.

[1]Pořadí zápisu bytů - nejméně významný byte je vlevo, nejvíce významný byte je vpravo. Např. hodnota 0xA0B0C0 (10531008_{10}) bude mít v Little-Endian podobu 0xC0B0A0.



Obrázek 2.1: Formát .Zip souboru [1]

2.2 RAR formát

Převzato z [27], kde se nacházejí soubory k RAR formátu, přeložitelné na platformě Linux. Formát dat se poté nachází v textovém souboru `technote.txt`.

Datová struktura RAR souboru popsaná v tabulce 2.1 odpovídá verzi 4.20. Archivační formát dat jako takový je stejný od verze 1.50, avšak novější verze mohou (ne)vyžadovat jiné parametry. Taktéž níže popsané struktury nebudou popsány v celé šíři, kvůli veřejné nedostupnosti popisu archivace. RAR soubory používají taktéž pořadí bytů Little-Endian.

Offset	Byte	Popis
0	7	Signature
		(0x52 0x61 0x72 0x21 0x1A 0x07 0x00)
7	m	1. volume header
7+m	n	2. volume header
7+m+n	p	další volume headers
7+m+n+p	q	n. volume header
...	...	...

Tabulka 2.1: Obecná stuktura RAR souboru

Signatura identifikuje RAR soubor a za touto signaturou následují *volume headers* (tabulka A.5), které mají proměnnou délku specifikovanou typem. Ve skutečnosti je signatura také *volume header* s `header_type` `MARK_HEAD` (str. 61).

`header_size` určuje celkovou velikost *volume header* a `header_type` (tabulka A.6) definuje, jak se bude s byty po `header_size` zacházet.

Obsah bloku se u každého `header_type` liší, tj. každý `header_type` má svoji množinu `header_flags` i množinu dalších položek závisejících právě na `header_type`.

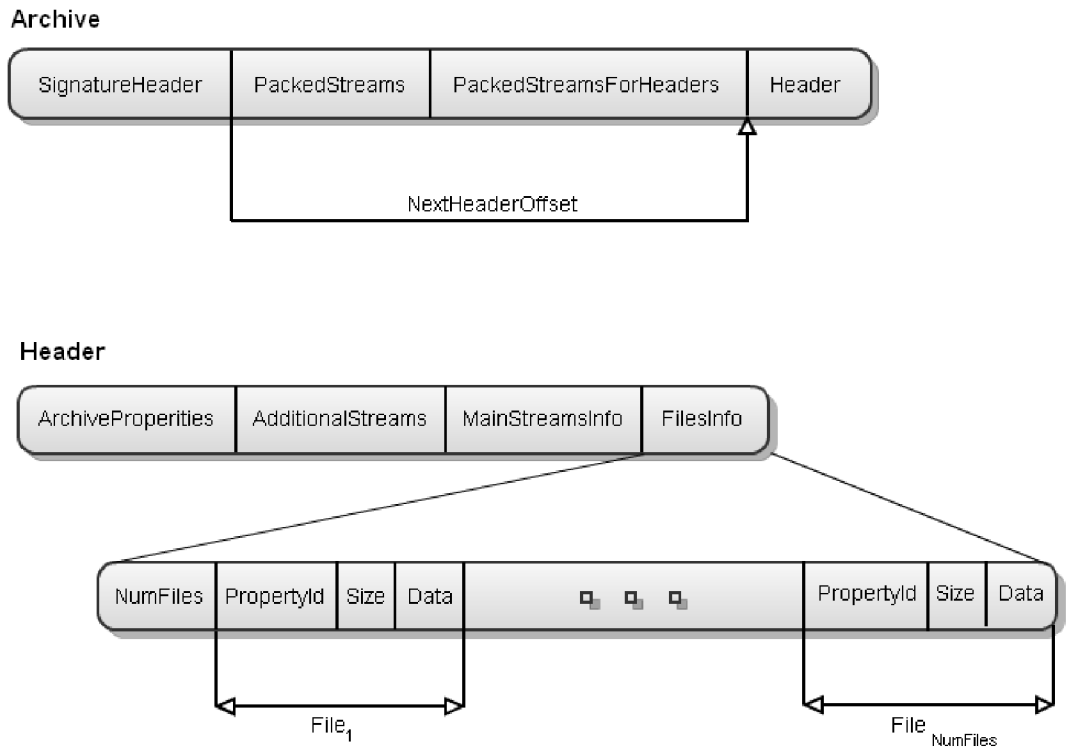
`header_type` `MAIN_HEAD` identifikuje informace o samotném RAR souboru a jeho para-

metrech, které jsou dány množinou příznaků (tabulka A.7).

`FILE_HEAD` slouží pro určení souboru v archivu, kde příznaky (tabulka A.8) a další datové položky (tabulka A.9) upřesňují informace ohledně souboru samotného.

2.3 7z formát

Struktura tohoto formátu je popsána velice abstraktně ve zdrojových souborech, které jsou volně k dispozici [24]. Archiv samotný se skládá z hlavičky signatury, zkomprimovaných souborů a hlaviček, a hlavičky, která poskytuje informace o celém archivu a jeho obsahu (obr. 2.3).



Obrázek 2.2: Struktura .7z archivu

Veškerá data používají pořadí bytů Little-Endian. Hlavička signatury určuje minimální verzi programu potřebnou pro extrakci a hodnotu offsetu hlavní hlavičky.

PackedStreams a *PackedStreamsForHeaders* jsou samotná zkomprimovaná data. Jejich organizaci a jak z nich vyčíst potřebná data, určuje hlavní hlavička, která se při extrakci musí zpracovat dříve než samotná zkomprimovaná data. Informace o jednotlivých komprimovaných souborech je v položce *FilesInfo* a popisuje všechna potřebná data o komprimovaném souboru a způsob, jak jej najít a dekódovat z *PackedStreams*.

2.4 SFX

Self-Extraction Archive (SFX) je samostatně spustitelný soubor, který kromě samotného archivu obsahuje i část, která zajišťuje samotnou extrakci archivu, aniž by na hostitelském počítači byl přítomen software potřebný k extrakci takového archivu. Extrahovací součást

obsahuje pouze ty součásti, které vyžaduje archiv pro své úspěšné extrahování. Pokud je tedy archiv pouze komprimován pomocí některé metody, extrahovací součást obsahuje pouze spustitelný kód, který zajistí dekompresi archivu. Je-li archiv i zapezpečen pomocí některé z šifer, v extrahovací části je i spustitelný kód pro dešifrování takového archivu.

Všechny SFX soubory jsou závislé na platformě, na které byly vytvořeny. Proto není možné úspešně spustit SFX soubor např. na platformě Unix, pokud byl soubor vytvořen na platformě Windows[2].

Jednotný formát pro SFX soubory neexistuje, protože informace pro extrakci SFX souboru si nese každý SFX soubor s sebou a není nutné zajišťovat kompatibilitu mezi jednotlivými výrobci. Proto každý software vytvářející SFX soubory tvoří daný SFX soubor jinak.

2.4.1 SFX a malware

Samorozbalovací archivy jsou z pohledu malware ideálním prostředkem pro úspěšnou infiltraci napadeného počítače. Dle informací od firmy AVG je nejčastější formou malware tohoto typu vnořený samorozbalovací archiv v dalším samorozbalovacím archivu - vnitřní archiv je zaheslovaný a vnější obsahuje konfigurační informace včetně hesla, které umožňují vnitřní archiv otevřít a spustit bez jakékoliv programové výzvy, ať již pro zadání hesla či jiných požadavků.

Toto řešení je pro malware vyhovující, protože antivirové programy bez znalosti hesla nedokáží zjistit obsah vnitřního archivu. Vnější nezaheslovaný archiv antivirové programy otevřít a zkontrolovat umí, avšak při kontrole zaheslovaného archivu jejich činnost končí. Útoky silou na tyto archivy jsou sice možné, ale antivirové programy na klientském počítači nemohou útočit na archivy po neomezený čas a spotřebovávat tak veškeré výpočetní zdroje klientského počítače.

Proto je nutné znát některé slabiny příslušných kontrolovaných souborů, případně i použít základní útoky, které však testují omezenou množinu zadaných hesel. Tato práce se bude věnovat způsobu uložení konfiguračních informací jednotlivých archivačních formátů a následné extrakci hesla, které je potřebné pro úspěšné otevření vnitřního archivu.

2.5 Portable Executable

Zkráceně PE, je souborový formát používaný v operačních systémech typu Windows NT[3] pro různé typy spustitelných souborů, knihoven atd. Tento formát obsahuje a popisuje veškeré informace, které operační systém bude potřebovat pro úspěšné spuštění programového kódu, který je uložen v rámci souboru.

PE se vyvinul z Common Object File Format (COFF), který existoval před vznikem operačního systému Windows NT. COFF byl výsledkem ověřených nástrojů, které se používaly v operačních systémech VAX, VMS a UNIX a které byly inspirací pro operační systém Windows NT[18]. Proto PE struktura i nyní obsahuje položky pro COFF.

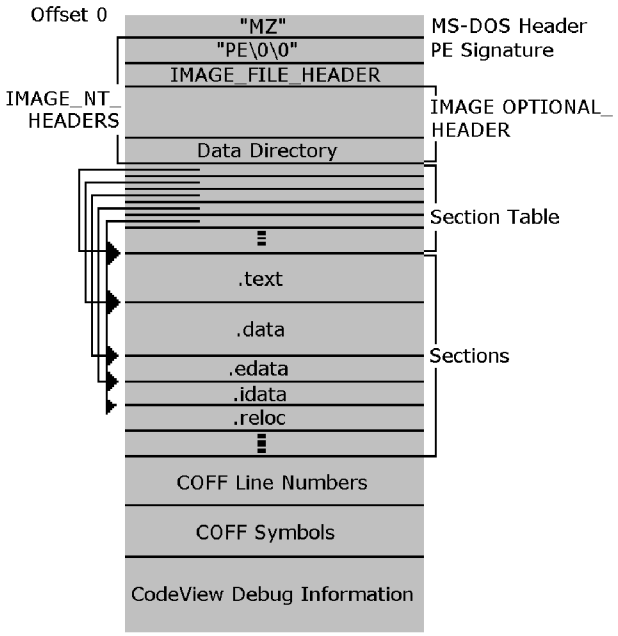
Strukturu PE je nutné znát kvůli způsobu uložení veškerých informací, včetně informací, které přímo nesouvisejí s chodem programu. V případě samorozbalovacích archivů to

[2]Nebereme v potaz programy typu wine[2], které vytvářejí programovou vrstvu, schopnou spouštět Windows aplikace v Unix prostředí.

[3]NT - New Technology. Operační systémy, které začaly využívat 32-bitové procesory. Prvním operačním systémem byl Windows NT 3.1 z roku 1993.

budou námi hledané konfigurační informace, samotný archiv určený k rozbalení a podobně.

Při načítání PE programu do operačního systému jsou příslušná data ze souboru přímo namapována do virtuálního adresového prostoru. Operační systém poté může se souborem pracovat obdobně jako s jakýmkoliv jiným souborem, který je namapován do paměti.



Obrázek 2.3: Souborový formát PE [18]

Na PE struktuře (obrázek 2.3) je vidět, že hlavní část tvoří sekce (Sections). Sekce jsou bloky souvislé paměti, které mohou mít různou velikost a které obsahují buď kód potřebný k běhu programu nebo data, která program využívá. Právě tyto sekce specifikují jak a kam má být program přesně namapován. Nejčastější sekce a jejich význam budou vysvětleny později, prvně je však potřeba popsat úvod PE souboru dle obrázku 2.3.

- „MZ“ - Signatura a krátký MS-DOS program, který se vykoná v případě, kdy je program spuštěn v prostředí, které neumožní správné spuštění celého programu. Činností krátkého MS-DOS programu je pouze tisk informace, že program nelze spustit v MS-DOS módu - „This program cannost be run in MS-DOS mode.“

  Tato část také obsahuje offset na počátek PE hlavičky, jehož hodnota začíná na pozici `0x3C` a má velikost 4 byty v pořadí Little-Endian.

- „PE\0\0“ - Signatura samotné PE struktury. Označuje počátek PE hlavičky.

- Image File Header - hlavička obsahující základní informace o celém souboru, např.:
  - typ procesoru, pro který je program určen
  - počet sekcí
  - čas vytvoření programu v sekundách od 31.prosince 1969, 4:00 PM

  - velikost Image Optional Header

- Image Optional Header - další položky, které jsou pro COFF soubory volitelné, avšak pro PE soubory nikoliv. Patří sem především verze použitých linkerů, minimální verze operačního systému, velikosti a offsety důležitých sekcí a částí kódu. Celou specifikaci lze naleznout na stránkách Microsoftu[18] nebo přímo v souboru `WINNT.h`.
- Section Table - tabulka hlaviček jednotlivých sekcí, kde řádek v tabulce odpovídá jedné hlavičce sekce. Pořadí jednotlivých hlaviček v tabulce je dáno pozicí jednotlivých sekcí v programu, nikoliv abecedně. Každá hlavička sekce obsahuje položky popsané v tabulce 2.2.
- Sections - jednotlivé sekce, nejčastější jsou popsány na sekci 2.5.1.
- COFF Line Numbers - čísla řádků spustitelného kódu, využito pouze při debugování.
- COFF Symbols - COFF tabulka použitých symbolů, využito pouze při debugování.
- CodeView Debug Information - informace pro debubování pro jiný typ debuggeru[4].

Offset	Velikost	Název	Popis
0	8	Name	Jméno sekce v ASCII délky 8 bytů. Obvykle začíná tečkou (".text"), avšak není to nutné.
8	4	VirtualSize	Velikost sekce po načtení do paměti.
12	4	VirtualAddress	Relativní adresa, na kterou se má sekce načíst.
16	4	SizeOfRawData	Velikost sekce v PE programu.
20	4	PointerToRawData	Offset na počátek sekce v PE programu.
24	4	PointerToRelocations	Offset na počátek relokačních záznamů (u PE nevyužito).
28	4	PointerToLinenumbers	Offset na počátek záznámů s čísly řádků (pro debugování).
32	2	NumberOfRelocations	Počet relokačních záznamů (u PE nevyužito).
34	2	NumberOfLinenumbers	Počet záznamů s čísly řádků.
36	4	Characteristics	Příznaky určující atributy sekce (např. obsahuje spustitelný kód, inicializovaná data, sekce pouze pro čtení a jiné).

Tabulka 2.2: PE hlavička sekce

2.5.1 Nejčastější PE sekce

Existuje mnoho druhů sekcí, které se mohou nacházet v PE programu - některé jsou důležité a přítomny v každém programu, některé naopak mají využití téměř nulové. Následuje popis těch důležitých a častých sekcí:

- **.text** - v této sekci se nachází veškerý spustitelný strojový kód na úrovni asembleru, který byl vygenerován kompilátorem

[4]Při použití Windows NT debuggerů NTSD či KD lze využít pouze COFF debugovací informace.

- **.data** - zde jsou všechna inicializovaná data, které představují globální a statické proměnné inicializované v době překladu: hodnoty, řetězce apod.
- **.bss** - obsahuje neinicializované globální a statické proměnné
- **.CRT** - další inicializovaná data, která používá Microsoft C Run Time knihovna. Původ této sekce znají pouze pánové z Redmondu, neboť má stejný význam jako .data sekce[18].
- **.rsrc** - resources neboli ostatní zdroje, které rozšiřují samotný program o nějakou funkcionalitu. Nejčastěji jsou zde ikony, dialogy a dialogová okna, informace o typu písma atd. Typ zdroje však není omezen, je tedy možné zde mít uložena jakákoliv data.
- **.idata** - popisuje seznam funkcí a dat, které jsou do programu importovány z jiných dynamických knihoven DLL.
- **.edata** - taktéž obsahuje seznam funkcí a dat, které ale program samotný exportuje pro jiné programy.
- **.reloc** - tabulka relokačních hodnot, které se využívají pro korekci pozice instrukcí či inicializovaných hodnot pokud je zavadeč operačního systému nedokáže najít na místě, kde by dle linkeru měly být.
- **.rdata** - obsahem jsou debugovací informace pro různé typy debuggerů.

2.6 Diasembler a dekompilátor

Při zjišťování chování určitého programu, jehož činnost není pro veřejnost dostatečně či vůbec zdokumentována, je jedním z možných způsobů, jak chování takového programu zjistit, jeho zpětné sestavení do jazyka, který je pro člověka čitelný. Pro zpětné sestavení programu je však nutné znát způsob tvorby programu.

Ve zjednodušené formě je program vytvořen postupně těmito fázemi:

- zdrojový kód napsaný ve vysokoúrovňovém programovacím jazyce[5] (nejčastěji jazyk C) je pomocí **kompilátoru** přeložen na nízkoúrovňový programovací jazyk[6].
- kód vytvořený kompilátorem je pomocí **asembleru** převeden na strojový kód v binární formě, který je procesor schopen přímo zpracovat.

Zpětné sestavení programu z binární formy je přesně opačné tvorbě programu:

- program v binární formě je pomocí **diasembleru** převeden na nízkoúrovňový programovací jazyk, který je sice pro člověka čitelný, avšak velké programy napsané v tomto jazyce jsou těžké na pochopení.
- z nízkoúrovňového programovacího jazyka **dekompilátor** vytvoří vysokoúrovňový programovací jazyk, v němž je celý program a jeho struktura znatelně čitelnější.

[5] Kód má velkou míru abstrakce, kde konstrukce daného jazyka více odpovídají stylu myšlení člověka.
[6] Konstrukce jazyka se více podobá způsobu práce samotného počítače.

Proces zpětného sestavení zdrojového kódu ve vysokoúrovňovém jazyce, jak je popsáno výše, však není tak jednoduchý a prozatím nikdy nelze z binárního kódu získat diasemblováním a dekompilací zcela totožný zdrojový kód, z něhož byl vytvořen daný binární kód.

Důvodem proč není možné sestavit původní zdrojový kód je fakt, že při kompilaci kódu vysokoúrovňového jazyka kompilátor zahazuje veškeré komentáře a jména proměnných, provádí optimalizaci kódu, která části kódu upraví, zjednoduší, předpočítá nebo i zahodí, pokud se do daných míst program nikdy nedostane. Z takto optimalizovaného kódu se poté vytváří již zmíněný nízkoúrovňový kód, kdy se k příslušným funkcím a konstrukcím hledají příslušné instrukce - každý kompilátor má k dispozici jinou množinu instrukcí, která se může lišit od typu použitého procesoru, na který se program kompiluje. Mohou tak vznikat rozdíly, kdy kompilací jednoho stejného zdrojového kódu vzniknou na jiných počítačích, použitím jiných kompilátorů, zcela jiné kódy v nízkoúrovňovém jazyce.

Proto je v tuto dobu nemožné získat zcela původní zdrojový kód, neboť binární podoba jednoduše neobsahuje všechny potřebné informace k znovusestavení zdrojového kódu. V současnosti je i obtížné vytvořit diasembler, který binární program převede do kódu ve strojovém (nízkoúrovňovém) jazyce takovým způsobem, že získaný kód po překladu asemblerem na binární program bude identický s původním binárním programem.

Existuje mnoho volně dostupných diasemblerů (IDA Pro Freeware[29], Objconv a dalsí) a dekompilátorů (Hex-Rays, Boomerang[26] a jiné), avšak momentálně nelze s žádným nástrojem otevřít binární program, spustit diasembler a dekompilátor a na vygenerovaném kódu v jazyku C ihned pracovat a upravovat jej s vědomím, že bude vzápětí fungovat. Nebude. Především dekompilátor zanese do kódu mnoho zmatku, většinou v oblasti interpretace různých strojových instrukcí a jejich ekvivalentu v jazyce C.

Stále je tedy nutné diasemblovaný strojový kód prozkoumat a porozumět mu a souběžně s tím kontrolovat dekompilátorem vygenerovaný kód, zda-li konstrukce vysokoúrovňového jazyka odpovídají skutečně tomu, co nízkoúrovňový strojový jazyk popisuje.

Kapitola 3

Šifrování a komprese

Pro zabezpečení archivu se dříve používalo heslo, které udávalo přístup do obsahu archivu. Avšak samotný obsah archivu nebyl nijak šifrován, proto bylo velice snadné heslo archivu obejít a dostat se přímo k nešifrovaným souborům.

Dnes již ochrana archivů zahrnuje šifrování samotných souborů i hlaviček jednotlivých formátů archivů. Nejpoužívanější šifrou je AES, která je přítomna ve všech nejčastějších archivačních formátech. Nadále se i používá starší šifra PKAWRE pro své široké použití a podporu mezi mnoha programy. Existují i jiné šifry, některé jsou starší a považovány za slabé (DES [23], 3DES, RC4). Některé jsou naopak nové a bezpečné, některými považovány i za bezpečnější než AES, avšak z neznámého důvodu nejsou hojně využívány (Blowfish [31], Twofish [32]).

Jsou zde rozebrány šifry AES i PKWARE a je zde popsáno, jak dané šifry využívají jednotlivé archivační formáty.

Následně je i popsána komprimační metoda Deflate, která je nejpoužívanější pro kompresi dat, především pak u archivů Zip. Z komprimačních metod je popsána pouze metoda Deflate, neboť činnosti komprimačních metod není zcela nutné znát z hlediska bezpečnosti. Avšak v implemetační části (sekce 5.5) bylo zjištěno použití Deflate pro uložení důležitých dat a pro jejich úspěšnou dekompresi je nutné znát implementační detaily této metody.

3.1 AES

Advanced Encryption Standard (AES) je bloková symetrická šifra. Bloková šifra se aplikuje na celý blok dat určité velikosti, které musí být před šifrováním k dispozici. Symetrická šifra poté používá stejný šifrovací klíč pro šifrovací i dešifrovací proces.

AES používá bloky o velikosti 128 bitů (vstupní, výstupní a „State“ viz 3.1.1). Šifrovací klíč (Cipher Key) může mít různou délku odvíjející se od verze AES. V samotném procesu šifrování se šifrovací klíč nepoužívá v celé své délce, ale je rozdělen na 32-bitové subklíče (Round Key viz 3.1.1). Všechny kombinace verzí AES jsou v tabulce 3.1.

Počet subklíčů - Nk
Nk označuje počet 32-bitových slov (sloupců) v šifrovacím klíči (Cipher Key).

Počet subbloků - Nb
Nb označuje počet 32-bitových slov (sloupců) v matici State (viz 3.1.1).

	Počet subklíčů Nk	Počet subbloků Nb	Počet kol Nr
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Tabulka 3.1: AES verze

Počet kol - Nr
Nr označuje počet kol provedených šifrou AES.

AES provádí s blokem dat kruhovou funkci (round function), která se skládá ze čtyř různých, bytově-orientovaných (byte-oriented) transformací. Tato kruhová funkce se opakuje Nr–krát a je tvořena následujícími transformacemi:

1. Bytová substituce pomocí substituční tabulky (S-Box - str. 15)
2. Posunutí řádků ve State matici o různý offset
3. Smíchání dat s každým sloupcem State matice
4. Přidání Round Key do State matice

Jednotlivé transformace budou popsány níže v sekcích 3.1.2 a 3.1.3.

3.1.1 Pojmy

State matice

Základním stavebním kamenem šifry AES je blok State, což je matice 4×4, kde každá buňka má velikost jednoho byte (8 bitů).

Jednotlivé buňky jsou označeny symbolem s a indexovány číslem řádku r ($0 \leq r < 4$) a číslem sloupce c ($0 \leq c < Nb$). Díky tomu je každá buňka jednoznačně určena pomocí zápisu $s_{r,c}$ nebo $s[r, c]$.

Na začátku (de)šifrování je vstupní blok překopírován do State matice, poté jsou nad State maticí Nr-krát provedeny transformace algoritmu AES a nakonec jsou hodnoty nakopírovány ze State matice do výstupního bloku.

Zkopírování bloků z/do State matice je založeno na schématu:

- vstupní pole `in` zkopírováno do State matice s: $s[r, c] = in[r + 4c]$ pro $0 \leq r < 4$ a $0 \leq c < Nb$
- State matice s zkopírována do výstupního pole `out`: $out[r + 4c] = s[r, c]$ pro $0 \leq r < 4$ a $0 \leq c < Nb$

Graficky poté viz tabulka 3.2.

Vstupní blok					State matice					Výstupní blok			
in_0	in_4	in_8	in_{12}	→	$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$	→	out_0	out_4	out_8	out_{12}
in_1	in_5	in_9	in_{13}		$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$		out_1	out_5	out_9	out_{13}
in_2	in_6	in_{10}	in_{14}		$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$		out_2	out_6	out_{10}	out_{14}
in_3	in_7	in_{11}	in_{15}		$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$		out_3	out_7	out_{11}	out_{15}

Tabulka 3.2: Kopírování z/do State matice [21]

Key Expansion

Proces tvorby subklíčů pro celý algoritmus AES se nazývá Key Expansion. Tato rutina vytvoří celkem $Nb(Nr+1)$ slov[1] - při incializaci algoritmu je potřeba Nb slov, pro každé z Nr kol kruhové funkce je zapotřebí Nb slov.

Key Expansion vezme vstupní klíč (Cipher Key) o velikosti Nk slov a rozšíří jej náhodnými čísly na velikost $Nb(Nr+1)$ slov. Prvních Nk slov rozšířeného klíče odpovídá přesně Nk slovům Cipher Key. Každé další slovo $w[i]$ odpovídá výsledku XOR operace předchozího slova $w[i]$ se slovem o Nk pozic zpět, $w[i-Nk]$.

Slova, jejichž pozice je však násobkem Nk se před XORem s $w[i-Nk]$ transformují: slovo se cyklicky posune o jednu pozici doleva, přetransformuje se pomocí S-Boxu a následně se provede XOR s cyklickou konstantou závisející na pozici slova. Až poté se provede XOR s $w[i-Nk]$. Odpovídající pseudokód lze najít v příloze B.1.

Výsledkem Key Expansion je „rozvrh klíčů“ (**Key Schedule**), což je lineární pole slov, kde každé slovo je značeno $[w_i]$ a kde i je v rozsahu $0 \leq i < Nb(Nr+1)$.

Round Key

Subklíč použitý v průběhu šifrování. Klíč je tvořen Nb slovy z Key Schedule, tj. jeden Round Key má velikost 128 bitů.

3.1.2 Šifrování

Na začátku šifrování je vstupní blok překopírován do State matice podle schématu 3.1.1. Po přidání počátečního Round Key je nad State maticí provedena kruhová funkce s počtem opakování 10×, 12× nebo 14× (odvíjí se od délky klíče). Poslední kolo je poté mírně odlišné od předchozích. Koncová State matice je nakonec zkopírována do výstupního bloku.

Postup transformací State matice

V prvním (nultém kroku) se provede `AddRoundKey()`, poté $Nr-1$ kol transformací `SubBytes()`, `ShiftRows()`, `MixColumns()`, `AddRoundKey()` a v posledním, Nr–tém kole, následuje `SubBytes()`, `ShiftRows()`, `AddRoundKey()`.

[1] 1 slovo (word) = 4×byte = 32 bit

SubBytes()

Nelineární bytová substituce, která pracuje nezávisle na bytech ve State matici a využívá substituční tabulku (S-Box, tab. 3.3). Např. pro buňku $s_{2,1} = \{6e\}$ je její substituční hodnota dána buňkou na řádku „6“ a sloupci „e“. V tomto případě je substituční hodnota $s'_{2,1} = \{9f\}$.

Samotný S-Box je invertovatelný a způsob jeho vytvoření lze najít online [40].

		y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
x	**0**	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

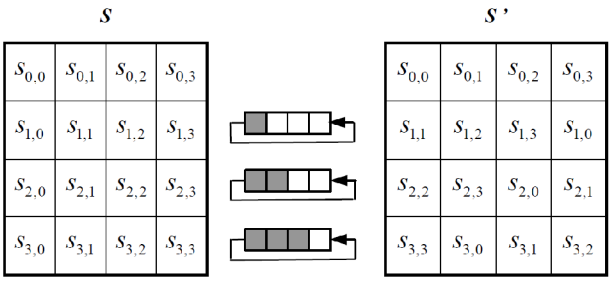
Tabulka 3.3: S-Box, hodnoty v hexadecimálním tvaru xy [21]

ShiftRows()

Byty v posledních třech řádcích State matice jsou cyklicky posunovány o různý počet bytů (offset). První řádek posunován není. Posun lze zapsat:

$$s'_{r,c} = s_{r,(c+shift(r,Nb))\ mod\ Nb} \quad pro \quad 0 < r < 4 \quad a \quad 0 \leq c < Nb$$

kde hodnota `shift(r, Nb)` závisí na číslu řádku r. Tedy, řádek 1 bude posunut o 1 pozici (`shift(1,4) = 1`), řádek 2 o 2 pozice (`shift(2,4) = 2`) atd. Graficky znázorněno na obr. 3.1.



Obrázek 3.1: Schéma ShiftRows() [21]

MixColumns()

Funkce pracuje sloupec po sloupci, kde každý sloupec bere jako čtyř-členný polynom. Sloupce jsou považovány za polynomy tělesa GF(2^8) a jsou násobeny modulo x^4+1 s fixním polynomem $a(x)$:

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$$

Což lze zapsat jako: $s'(x) = a(x) \otimes s(x)$ a rozepsat jako maticové násobení:

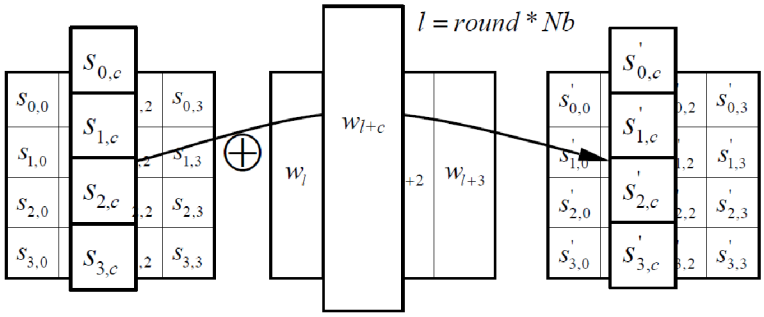
$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad pro \quad 0 \leq c < Nb$$

AddRoundKey()

Nad každým sloupcem State matice se provede XOR operace s Round Key, který obdržíme od Key Schedule (3.1.1).

$$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \oplus [w_{round*Nb+c}] \quad pro \quad 0 \leq c < Nb,$$

kde $[w_i]$ je Round Key a $round$ je hodnota odpovídající aktuálnímu kolu v rozmezí $0 \leq round \leq Nr$. Grafické zobrazení viz obr. 3.2.



Obrázek 3.2: Schéma AddRoundKey() [21]

3.1.3 Dešifrování

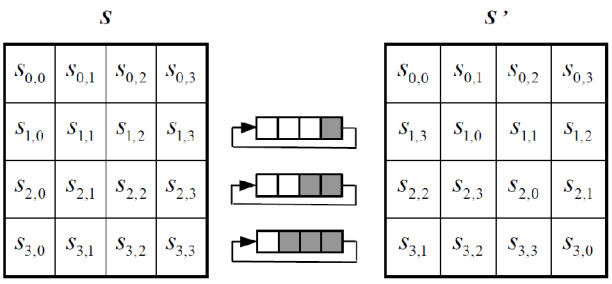
Proces dešifrování je obdobný jako u šifrování, funkce provádějící transformaci State matice zůstavají ve stejném pořadí, avšak jejich struktura je inverzní vůči původním funkcím. Inverzními funkcemi tedy jsou `InvShiftRows()`, `InvSubBytes()` a `InvMixColumns()`. Funkce `AddRoundKey()` je inverzní sama o sobě, protože pouze používá XOR, který je sám sobě inverzní.

InvShiftRows()

Stejně jako u `ShiftRows()`, i zde jsou poslední tří řádky State matice cyklicky posunuty o rozdílný počet bytů (offset) a i zde se první řádek ($r = 0$) neposouvá. Zápis posunu je obdobný:

$$s'_{r,(c+shift(r,Nb))\ mod\ Nb} = s_{r,c} \quad pro \quad 0 < r < 4 \quad a \quad 0 \leq c < Nb$$

kde hodnota `shift(r, Nb)` závisí na číslu řádku r. Grafické zobrazení viz obr. 3.3.



Obrázek 3.3: Schéma InvShiftRows() [21]

InvSubBytes()

Opět nelineární bytová substituce, která se provádí nad invertovanou subtituční tabulkou 3.4.

InvMixColumns()

Obdobně jako `MixMolumns()` se zpracovává sloupec po sloupci, kde každý sloupec bere jako čtyř-členný polynom. Sloupce jsou považovány za polynomy tělesa GF(2^8) a jsou násobeny modulo $x^4 + 1$ s fixním polynomem $a^{-1}(x)$:

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}$$

Což lze zapsat jako: $s'(x) = a^{-1}(x) \otimes s(x)$ a rozepsat jako maticové násobení:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \cdot \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad pro \quad 0 \leq c < Nb$$

		y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
x	**0**	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Tabulka 3.4: Invertovaný S-Box, hodnoty v hexadecimálním tvaru xy [21]

3.2 WinZip a AES

WinZip používá pro šifrování a dešifrování souborů algoritmus AES od verze 9.0 [42]. WinZip využívá dvě šifrovací specifikace: AE-1, AE-2. Jediný funkční rozdíl mezi těmito dvěma specifikacemi je ten, že AE-2 neukládá CRC hodnotu šifrovaného souboru, kdežto AE-1 ano.

3.2.1 Formát dat

Pro začlenění AES algoritmu do stávajícího Zip souboru se struktura Zip hlaviček nijak neupravuje, využívá se *Extra field* (tabulka A.1, str. 59), je zaveden nový kód pro kompresní metodu a hodnota CRC je závislá od použitého šifrovacího formátu (AE-1 či AE-2). Zbylá pole Zip formátu zůstávají stejná a jejich obsah se od nešifrovaného souboru neliší. Veškeré informace o šifrování jsou tedy obsaženy v *Extra field*.

Parametry hlaviček

General purpose bit flag
Nultý bit musí být nastaven na 1 pro každý local header a central directory záznam.

CRC
Pro soubory zašifrované pomocí AE-2 se položka pro CRC nepoužívá[2] a musí mít hodnotu 0. Pro kontrolu, zda-li je soubor poškozen či nikoliv se používá položka *authentication code*, která je součástí dat zašifrovaného souboru (str. 21).

[2]Důvodem tohoto opatření byly útoky na slabiny CRC součtu, kdy AE-1 uchovávala informace o CRC součtu nezašifrovaného souboru [15].

Naopak soubory zašifrované pomocí AE-1 musí mít položku pro CRC standartně vyplněnou.

Extra field
Všechny soubory zašifrované pomocí algortimu AES obsahují *Extra field*, které je součástí jak *local header*, tak *central directory* položky.

Hlavička extra pole pro AES má ID 0x9901. Celá hlavička má v aktuální specifikaci 4.20 velikost 11 bytů. Obsahy položek tabulky 3.5) jsou popsány vzápětí.

Offset	Velikost (byte)	Položka
0	2	Header ID (0x9901)
2	2	Data size
4	2	Vendor version
6	2	Vendor ID
8	1	Encryption strength
9	2	Actual compression method

Tabulka 3.5: Struktura extra data pole AES u WinZip

Data size - počet všech položek zašifrovaného souboru (samotný soubor, data potřebná pro dešifrování atd.) Hodnota je momentálně 7, ale kvůli možnosti změny specifikace v budoucnu je položka větší, aby mohla pojmout případné změny.

Vendor version - integer určující verzi - pro AE-1 hodnota 0x0001, pro AE-2 hodnota 0x0002.

Vendor ID - položka výrobce/prodejce, měla by vždy být nastavena na dvě ASCII hodonoty „AE“.

Encryption strength - integer definující sílu AES algortimu. Specifikuje, jak velký se použije šifrovací klíč a jeho hodnoty mohou být:

- 0x01 pro 128-bitový klíč
- 0x02 pro 192-bitový klíč
- 0x03 pro 256-bitový klíč

Actual compression method - určuje typ kompresní metody, která je použita na zabalený soubor. V normálním případě (kdy není soubor zašifrován) je typ kompresní metody popsán v local a central directory hlavičkách. Avšak v případě AES zašifrování tyto local a central directory hlavičky obsahují hodnotu 99 pro detekci AES šifrování a samotná kompresní metoda je specifikována v AES extra data poli.

Encrypted file storage format
Dodatečná data potřebná pro dešifrování souboru jsou uložena spolu se samotným zašifrovaným souborem, nejsou tedy obsažena v hlavičkách. Struktura daná tabulkou 3.6 odpovídá v *local header* položce *Compressed data*.

Velikost (byte)	Položka
Různá	Salt value
2	Password verification value
Různá	Encrypted file data
10	Authentication code

Tabulka 3.6: Struktura zašifrovaného souboru

Salt value - Salt[3] představuje náhodnou či pseudo-náhodnou sekvenci bytů, která spolu se vstupním šifrovaným heslem vytváří šifrovací a autentizační klíče. Salt hodnota je vygenerována šifrovacím algortimem a je uložena spolu s daty souboru v nezašifrované podobě.

Salt se používá pro znesnadnění slovníkových útoků a útoků hrubou silou na zašifrovaný soubor (více v sekci 4.3).

Key size	Salt size
128 bits	8 bytes
192 bits	12 bytes
256 bits	16 bytes

Tabulka 3.7: Velikost Salt

Password verification value - slouží k rychlému posouzení, zda-li zadané heslo může být správné či nikoliv. Verifikační hodnota je odvozena od vstupního hesla a uložena se zašifrovaným souborem (samotná verifikační hodnota není zašifrována). Při dešifrování je ze zadaného hesla odvozena verifikační hodnota a je porovnána s hodnotou uloženou u souboru.

Verifikační hodnota však nemůže být brána jako indikátor správnosti hesla. Z důvodu uložení na 2-bytech existuje pouze 65536 možností a způsob vytváření verifikační hodnoty nezaručuje, že vytvořená verifikační hodnota bude odpovídat jedinečnému heslu [42].

Encrypted file data - samotné šifrování pomocí AES algortimu se provádí pouze nad obsahem souborů a to až poté, co je provedena komprese dat v souboru. Šifrování probíhá byte po byte, avšak musí být dodržena velikost bloku 16 bytů. Samotná šifra probíhá v CTR režimu (str. 25), kde délka kompresovaných dat a délka kompresovaných a šifrovaných dat je identická [22].

[3] Česky sůl, dalo by se spíš přeložit jako „omáčka“.

Authentication code - autentizace slouží ke kontrole, zda-li zašifrovaný soubor nebyl dodatečně upraven či jakkoliv poškozen. Provádí se pomocí HMAC-SHA1 (sekce 3.2.2), která je provedena po kompresi a dešifrování dat.

Samotný autentizační kód je odvozen od výstupu šifrovacího procesu, je uložen v nezašifrované podobě, je zarovnaný po bytech a následuje okamžitě za posledním bytem zašifrovaných dat.

3.2.2 Generování klíčů

Pro vytvoření klíčů potřebných pro šifrování dat v Zip se využívá implementace Dr. Gladmana [9], který využívá algoritmus PBKDF2, popsaný v RFC 2898 [35].

PBKDF2

Algoritmus PBKDF2 (Password Based Key Derivation Function) lze zapsat jako funkci:

$$DK = \texttt{PBKDF2}(PRF, password, salt, c, dkLen)$$

kde:
DK - výstupní klíč,
PRF - pseudonáhodná funkce s dvěma parametry, délka výstupu $hLen$,
$password$ - vstupní heslo od uživatele,
$salt$ - sůl pro klíče,
c - počet iterací algortimu,
$dkLen$ - velikost výstupního hashe/klíče.

Výstupní klíč se skládá z bloků T_i o velikosti `hLen`, které jsou spolu konkatenovány:

$$DK = T_1 \quad || \quad T_2 \quad || \quad ... \quad || \quad T_{dkLen/hLen}$$

$$T_i = \texttt{F}(password, salt, iterací, i)$$

Funkce `F` je poté XOR c-iterací zřetězených pseudonáhodných funkcí:

$$\texttt{F}(password, salt, iterace, i) = U_1 \oplus U_2 \oplus ... \oplus U_c$$

kde:
$U_1 = \texttt{PRF}(password, salt \ || \ \texttt{INT\_msb(i)})$,
$U_2 = \texttt{PRF}(password, U_1)$,
. . .
$U_c = \texttt{PRF}(password, U_{c-1})$,

a kde `INT_msb(i)` je funkce, která převede parametr `i` na big-endian 32-bit integer. Tento integer je konkatenován se solí a předán spolu s heslem pseudonáhodné funkci.

Pro WinZip a AES-128 bude mít funkce PBKDF2 následující tvar [43]:

$$\texttt{DK} = \texttt{PBKDF2}(HMAC\_SHA1, password, salt(p\text{-}byte), 1000, m{+}n{+}2 \ byte)$$

kde:
m - počet bytů pro AES klíč,
n - počet bytů pro HMAC-SHA1 klíč,
2 - počet bytů pro Password verification value [42].

Hodnoty bytů pro dané klíče se odvíjí od délky hesla (tabulka 3.8).

Délka hesla	Salt	AES	HMAC-SHA1
$8 \leq$ délka < 32	8	16	16
$32 \leq$ délka < 48	12	24	24
$48 \leq$ délka < 64	16	32	32

Tabulka 3.8: Velikosti klíčů v bytech pro PBKFD2 [9]

Algortimus PBDKF2 dle specifikace musí ke své činnosti použít některou ze pseudonáhodných funkcí. Pro verzi WinZip AE-2 se osvědčila funkce HMAC-SHA1.

HMAC-SHA1

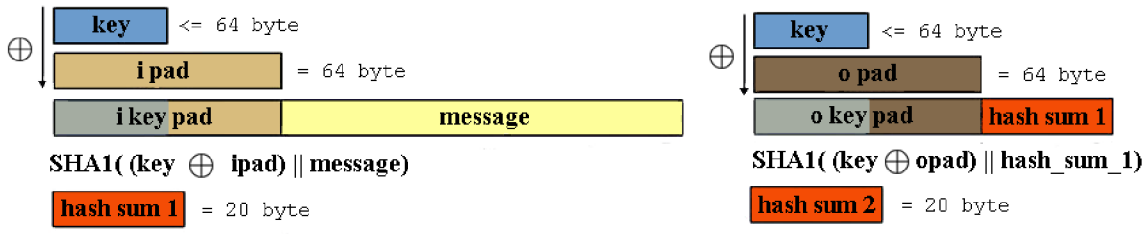
HMAC, neboli Hash-based Message Authentication Code, zajišťuje autentizaci a integritu zprávy pro kontrolu, zda-li zpráva nebyla při přenosu modifikována či jinak poškozena. Ke své činnosti používá hashovací funkce, v tomto případě hashovací funkci SHA1 (str.23).

Definice HMAC vychází z RFC 2104 [36]:

$$\texttt{HMAC}(K, m) = H((K \oplus opad) \quad || \quad H((K \oplus ipad) \quad || \quad m))$$

kde:
H - hashovací funkce,
K - klíč,
m - zpráva, která se má autentizovat,
|| - operace konkatenace,
opad - vnější zarovnání, hexadecimální konstanta 0x5c5c5c…5c5c o velikosti jednoho bloku,
ipad - vnitřní zarovnání, hexadecimální konstanta 0x363636…3636 o velikosti jednoho bloku.

Celou operaci lze nejlépe znázornit na obrázku 3.4.

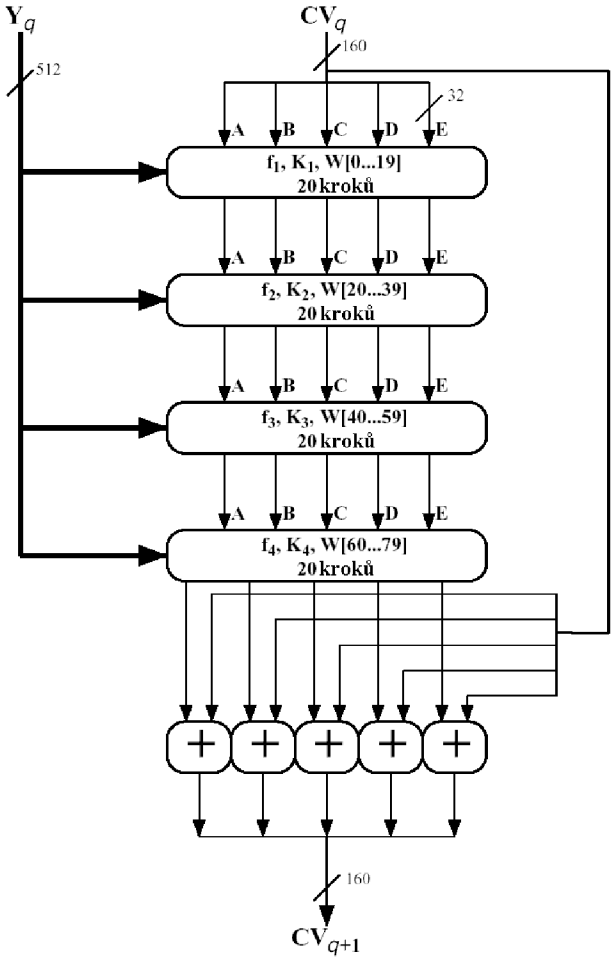


Obrázek 3.4: HMAC-SHA1 [41]

SHA1

Secure Hash Algorithm [19] je hashovací funkce, která pro vstup o maximální délce $2^{64} - 1$ bitů vytvoří hash o délce 160 bitů. Algortimus ke své činnosti využívá blok vstupního textu o velikosti 512 bitů.

Vstupní text y se nejprve zarovná hodnotou x tak, aby výsledný počet bitů (vstupního textu a zarovnání) modulo 512 byl roven hodnotě 448. Např. pro 728 bitů vstupního textu bude zarovnání 232 bitů ($728 + 232 = 960 \ \% \ 512 = 448$). Zarovnání poté bude tvořeno jednou jedničkou a $x - 1$ nulami. V popsaném případě tedy jednou jedničkou a 231 nulami. Hashovací funkce však pracuje s blokem o velikosti 512, proto se na konec dodá 64-bitová hodnota reprezentující původní velikost vstupního textu (proto se počítá modulo 448.). Původní velikost je zleva dorovnána nulami.



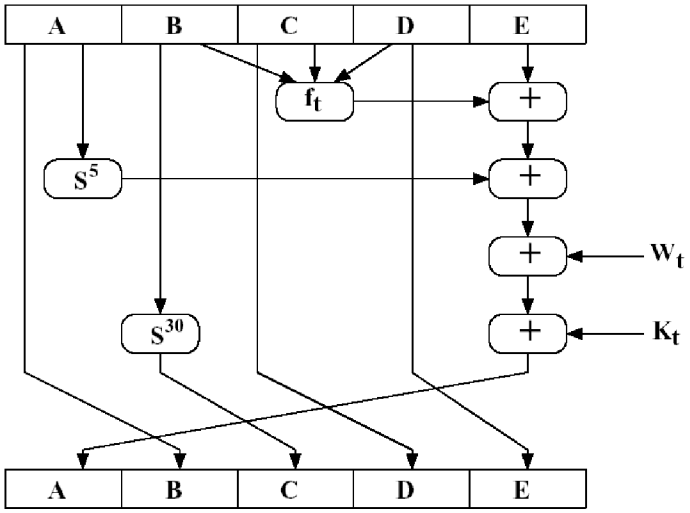
Obrázek 3.5: SHA1 komprese [33]

Jakmile je vstupní text připraven, následuje samotný proces vytvoření hashe - **SHA1 komprese** (obr. 3.5). Využívá 160-bitový buffer, který má 5 registrů o velikosti 32 bitů. Každý z registrů je inicializován na určitou hodnotu [19]. Pro každý blok vstupního textu následuje 4× 20 kroků **SHA1 operací** f_1 až f_4 (obr. 3.6), kde se každá operace mírně liší použitou vnitřní funkcí f_t. Po dokončení všech 80 kroků se sečtou (modulo 2^{32}) výstupy z poslední SHA1 operace s výsledkem předchozího bloku (v případě prvního kroku s inicia-

lizačními hodnotami). Výsledkem je poté 160-bitový blok, který se vyvede na výstup nebo se použije do dalšího kroku, pokud nebyla zpracována celá zpráva.

Funkce f_1 až f_4 použité v SHA1 kompresi lze znázornit pomocí obrázku 3.6. Bloky A až E mají velikost 32 bitů, funkce S^n značí bitový posun doleva o n-pozic. Funkce f_t mají poté následující podobu:
$f_1 = f_t(B, C, D) = (B\ AND\ C)\ OR\ ((NOT\ B)\ AND\ D)$ pro $0 \leq t \leq 19$,
$f_2 = f_t(B, C, D) = B \oplus C \oplus D$ pro $20 \leq t \leq 39$,
$f_3 = f_t(B, C, D) = (B\ AND\ C)\ OR\ (B\ AND\ D)\ OR\ (C\ AND\ D)$ pro $40 \leq t \leq 59$,
$f_4 = f_t(B, C, D) = B \oplus C \oplus D$ pro $60 \leq t \leq 79$.

Dále K_t jsou konstanty [19] o velikosti jednoho slova a W_t jsou slova vytvořená z bloku původní zprávy následovně:
- blok Y_q se rozdělí na 16 slov W_0, W_1, ..., W_{15}, kde W_0 je nejlevější slovo,
- pro každé $t = 16$ až $t = 79$ se provede: $W_t = S^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})$.



Obrázek 3.6: SHA1 operace [33]

3.2.3 Proces šifrování a dešifrování

Každý soubor je komprimován a šifrován nezávisle na ostatních souborech. Použitá kompresní metoda se odvíjí od položky *Actual compression method* a samotná komprese probíhá dle [7].

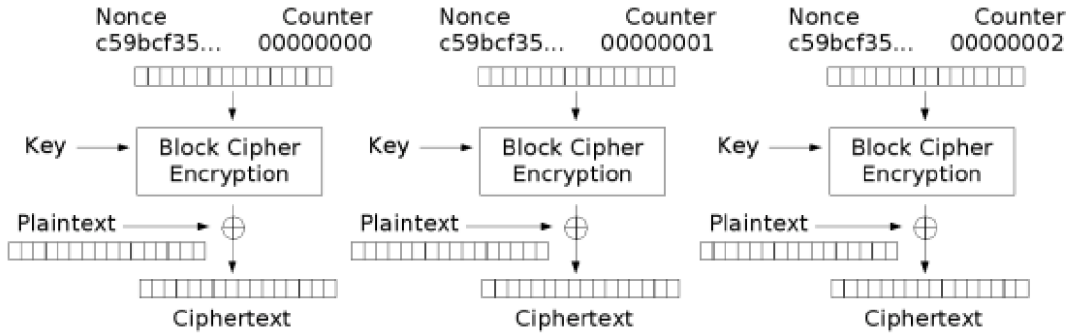
Jakmile je soubor zkomprimován, pomocí PBKFD2 (str. 21), hesla zadaného uživatelem a *Salt value*, vygenerují se potřebné klíče pro šifrování - AES klíč, HMAC-SHA1 klíč a *Password verification value*.

Pomocí AES klíče se nezarovnaná zkomprimovaná data zašifrují pomocí AES šifry v CTR režimu (str. 25) s counterem nastaveným na nuly. Po zašifrování se zpráva autentizuje pomocí HMAC-SHA1, kde se z výstupu 80 bitů použije pro *Authentization code*.

Proces dešifrování postupuje obdobně - vygenerují se potřebné klíče a pomocí nich se nad zašifrovanými daty provede HMAC-SHA1, kde se výstup porovná s *Authentization code* a ověří se tím, zda-li soubor nebyl modifikován[4]. Pokud byla zašifrovaná data úspěšně ověřena, dešifrují se pomocí AES šifry v CTR režimu.

CTR režim

Režim, ve kterém se bloková šifra přetvoří na proudovou šifru (obr. 3.7). Základním kamenem je „counter“, což je jakákoliv funkce, která dokáže vytvořit sekvenci, která se nebude opakovat po velkém počtu provedených kroků. Bloková šifra poté zkombinuje sekvenci z counteru a klíč, provede se operace XOR se vstupním textem a výsledkem je zašifrovaný text. Pro zašifrování i dešifrování se používá šifrující bloková šifra, není tedy nutné pro šifrování a dešifrování v režimu CRT používat dvě různé funkce.



Obrázek 3.7: Counter mode [10]

3.3 WinRAR a AES

Jak bylo již zmíněno, proces komprese ani šifrování není pro RAR formát veřejně dostupný. Avšak proces dešifrování je dostupný skrze zdrojové kódy [27, 39].

Dešifrování je obdobné jako u WinZip - nejprve se zjistí správnost CRC součtu a pokud je součet správný, vygenerují se klíče a inicializační vektor potřebný pro AES a z příslušného pole archivu se zjistí hodnota Salt.

Následně se načte blok vstupních dat (128 bitů). Tento blok se dešifruje a nad zašifrovaným blokem se provede operace XOR s inicializačním vektorem, jehož výstupem bude část původního souboru. Blok dat, který byl načten, ale nedešifrován, poté přepíše inicializační vektor a celý proces od načtení bloku se opakuje pro celý vstupní soubor.

Šifrování bude pravděpodobně postupovat opačně - vstupní soubor načítaný po blocích se vyXORuje s inicializačním vektorem. Tento blok se zašifruje, uloží do archivu a zároveň přepíše inicializační vektor.

[4] Útočník může změnit i hodnotu Authentization code, avšak bez znalosti původního hesla nebude tato hodnota korektní.

3.4 PKWARE

PKWARE je proudová šifra vyvinutá přímo pro Zip formát, kde se každý znak vstupního souboru spojí pomocí operace XOR s pseudonáhodnou posloupností bitů. Tato posloupnost se pro každý znak mění a je vytvořena kombinací hesla a tří klíčů.

V dnešní době je PKWARE šifra považována za slabou [43], jelikož používá tři 32-bitové klíče (celkově tedy 96 bitů). Tato šifra má však stále své uplatnění tam, kde není potřeba silné bezpečnosti. Taktéž je toto zabezpečení podporováno velkým množstvím programů na trhu, které využívají původní Zip formát [43].

3.4.1 Inicializace klíčů

PKWARE používá tři šifrovací klíče, každý o velikosti 32 bitů. Všechny klíče jsou inicializovány na specifickou hodnotu:

```
Key[0] = 305419896
Key[1] = 591751049
Key[2] = 878082192
```

Následně je každý klíč aktualizován na základě jednotlivých znaků hesla:

```
for i = 0 to length(password)-1
    update_keys(password(i))
end for
```

Funkci `update_keys()` lze zapsat:

```
update_keys(char)
    key[0] = crc32( key[0], char )
    key[1] = key[1] + ( key[0] & 000000ffH )
    key[1] = key[1] * 134775813 + 1
    key[2] = crc32( key[2], key[1] >> 24 )
end update_keys
```

kde operátor `&` značí operaci AND a operátor `>>` značí bitový posun doprava o n pozic.

`crc32(old_crc, char)` je poté funkce, která ze starého CRC součtu a předaného znaku vytvoří nový CRC součet na základě [13].

3.4.2 Šifrování

Proces šifrování nejprve začne inicializací klíčů a inicializací pole *encryption header*, což je položka před samotnými šifrovanými daty o velikosti 12 bytů. Tato hlavička by měla zabraňovat plaintext útokům tím, že poslední byte obsahuje CRC součet původního souboru (v příslušném bytovém pořadí). Ostatní položky obsahují náhodná čísla. Zároveň jsou všechny byty postupně zašifrovany pomocí výše popsaných tří klíčů, díky čemuž klíče před samotným šifrováním souboru nezávisí pouze na vstupním heslu. Inicializace hlavičky lze zapsat pomocí následujícího pseudokódu [25]:

```
generate_encryption_header(IsLittleEndian, CRC)
    for i = 0 to 10
        buffer[i] = encrypt_byte(Random());
```

```
    end for

    if IsLittleEndian then
        buffer[11] = encrypt_byte( LittleEndian(CRC) )
    else
        buffer[11] = encrypt_byte( BigEndian(CRC) )
    end if

    write buffer array into encryption_header
end generate_encryption_header
```

Pro zašifrování bytů hlavičky slouží `encrypt_byte()` (operace ^ představuje XOR):

```
byte encrypt_byte(value)
    local unsigned char temp
    temp = decrypt_byte()
    update_keys(value)
    encrypt_byte = temp ^ value
end encrypt_byte
```

Protože je PKWARE proudová šifra, je nutné neustále měnit posloupnost s klíči, což zajišťuje funkce `decrypt_byte()` (operace | představuje OR):

```
unsigned char decrypt_byte()
    local unsigned short temp
    temp = Key(2) | 2
    decrypt_byte = (temp * (temp ^ 1)) >> 8
end decrypt_byte
```

Jakmile je *encryption header* připraven, může nastat samotné zašifrování vstupního souboru pomocí
`encrypt_buffer()`:

```
encrypt_buffer()
    unsigned char temp
    unsigned char result
    while not EOF
        read a character into C
        temp = decrypt_byte()
        update_keys(C)
        result = temp ^ C
        output result
    end while
end encrypt_buffer
```

3.4.3 Dešifrování

Pro dešifrování postupujeme obdobně - inicializujeme klíče a dešifrujeme *encryption header*, kde poslední byte hlavičky může sloužit k tomu, zda-li uživatel zadal správné heslo[5].

[5] špatně zadané heslo způsobí špatnou inicializaci klíčů a výsledkem dešifrování *encryption header* bude nesprávný CRC součet v posledním byte hlavičky.

Proceduru pro dešifrování hlavičky lze zapsat:

```
decrypt_encryption_header()
    read encryption_header into buffer array
    for i = 0 to 11
        C = buffer(i) ^ decrypt_byte()
        update_keys(C)
        buffer(i) = C
    end for
end decrypt_header
```

Pokud vše proběhlo správně, stačí vstupní zašifrovaný soubor dešifrovat pomocí `decrypt_buffer()`:

```
decrypt_buffer()
    while not EOF
        read a character into C
        temp = C ^ decrypt_byte()
        update_keys(temp)
        output temp
    end while
end decrypt_buffer
```

3.5 Deflate

Kompresní metoda Deflate (RFC 1951[37]) vytváří bloky dat, které jsou komprimovány kombinací algoritmu LZ77[44] a Huffmanova kódování[12].

LZ77 pohlíží na vstupní text skrze dva posouvající se buffery - vyhledávací a předvídací. V předvídacím bufferu se metoda snaží naleznout shodu se skupinou znaků, která je ve vyhledávacím bufferu. Není-li shoda nalezena, je první znak v předvídacím bufferu uložen na výstup a oba buffery se posunou o pozici doprava. Je-li shoda nalezena, na výstup se uloží dvojice - offset shodné skupiny ve vyhledávacím bufferu (**vzdálenost**) a počet shodných znaků (**délka**).

Huffmanovo kódování přiděluje vstupním znakům binární kódy, které se odvíjejí od četnosti výskytu znaků ve vstupním textu. Čím častěji se znak v textu vyskytuje, tím kratší bude mít binární kód. Naopak málo se opakující znak bude mít binární kód delší. Pro reprezentaci vzniklých binárních kódů se používá **Huffmanův strom**.

Každý blok Deflate metody se skládá ze tří částí - z hlavičky, z dvojice Huffmanových stromů, popisujících komprimovaná data, a ze samotných komprimovaných dat.

Výstupem metody LZ77 jsou **literály** (znaky u nichž nebyla nalezena shoda ve vyhledávacím bufferu) a dvojice vzdálenost-délka pro zpětné nalezení shodného znaku ve vyhledávacím bufferu. Tyto prvky se převádí do Huffmanova kódování a následně se ukládájí do komprimovaných dat.

První z Huffmanových stromů udržuje informace o literálech a délkách, druhý strom poté o vzdálenostech.

Hlavička bloku se skládá ze tří bitů:

- jeden bit `BFINAL` pro identifikaci posledního bloku dat
- dva bity `BTYPE` pro určení způsobu komprese dat:
  - 00 - bez komprese
  - 01 - komprimováno pomocí fixního Huffmanova kódování
  - 10 - komprimováno pomocí dynamického Huffmanova kódování
  - 11 - rezervováno, označuje chybu

Jediný rozdíl mezi kompresí pomocí fixního či dynamického Huffmanova kódování je ve způsobu uložení Huffmanových stromů pro literály / délky a pro vzdálenosti.

V případě fixního Huffmanova kódování je podoba stromů určena definicí z RFC 1951[37], kdežto u dynamického Huffmanova kódování se stromy vytvářejí při kompresi a každý tento strom je pro každý blok dat různý.

Následuje popis jednotlivých kompresních metod.

Bez komprese

První čtyři byty udávají velikost dat v bloku (parametry `LEN` a `NLEN`), za nimiž následují již samotná nekomprimovaná data, na kterých nebyla použita metoda LZ77 - vstupní data byla přímo zkopírována do bloku.

Tato metoda se využívá především pro rozdělení příliš velkých souborů, které však není nutné komprimovat.

Huffmanovy stromy

Zkomprimovaná data obsahují dva Huffmanovy stromy, jejichž hodnoty jsou dány z definice. První strom obsahuje kódy pro literály a pro délky, kde kódy `0` – `255` představují literály, kód `256` značí konec bloku, a kódy `257` – `285` jsou určeny pro délky (tabulka 3.9). Druhý strom je pro kódy vzdálenosti (tabulka 3.10).

	Extra			Extra			Extra	
Kód	bit	Délka	Kód	bit	Délka	Kód	bit	Délka
257	0	3	267	1	15,16	277	4	67-82
258	0	4	268	1	17,18	278	4	83-98
259	0	5	269	2	19-22	279	4	99-114
260	0	6	270	2	23-26	280	4	115-130
261	0	7	271	2	27-30	281	5	131-162
262	0	8	272	2	31-34	282	5	163-194
263	0	9	273	3	35-42	283	5	195-226
264	0	10	274	3	43-50	284	5	227-257
265	1	11,12	275	3	51-58	285	0	258
266	1	13,14	276	3	59-66			

Tabulka 3.9: Hodnoty Huffmaných kódů pro délky[37]

Kód	Extra bit	Vzdálenost	Kód	Extra bit	Vzdálenost	Kód	Extra bit	Vzdálenost
0	0	1	10	4	33-48	20	9	1025-1536
1	0	2	11	4	49-64	21	9	1537-2048
2	0	3	12	5	65-96	22	10	2049-3072
3	0	4	13	5	97-128	23	10	3073-4096
4	1	5,6	14	6	129-192	24	11	4097-6144
5	1	7,8	15	6	193-256	25	11	6145-8192
6	2	9-12	16	7	257-384	26	12	8193-12288
7	2	13-16	17	7	385-512	27	12	12289-16384
8	3	17-24	18	8	513-768	28	13	16385-24576
9	3	25-32	19	8	769-1024	29	13	24577-32768

Tabulka 3.10: Hodnoty Huffmanových kódů pro vzdálenosti[37]

Fixní Huffmanovo kódování

Pro reprezentaci komprimovaných dat se používají Huffmanovy stromy, které byly popsány výše. Tyto stromy nejsou součástí komprimovaných dat, ale součástí dekompresoru.

Do komprimovaných dat se ale zapisují jiné Huffmanovy kódy, které reprezentují délky z fixní tabulky 3.10. Tyto kódy jsou popsány tabulkou 3.11.

Hodnota literálu	Počet bitů	Kódy
0 – 143	8	00110000 až 10111111
144 – 255	9	110010000 až 111111111
256 – 279	7	0000000 až 0010111
280 – 287	8	11000000 až 11000111

Tabulka 3.11: Kódy délky u fixního Huffmanova kódování[37]

Dynamické Huffmanovo kódování

U dynamického kódování jsou Huffmanovy stromy vytvářeny při kompresi a jejich reprezentace je uložena hned za hlavičkou komprimovanýho bloku.

Tyto stromy vznikají po zpracování vstupního textu pomocí LZ77, kdy se všechny použité symboly převedou do Huffmanova kódování dle četnosti výskytu. Následně se tento Huffmanův strom přeuspořádá, aby splňoval podmínky:

- kratší kódy se nacházejí v levém podstromu, další kódy poté v pravém podstromu
- je-li více kódů se stejnou délkou, tak se lexikograficky nižší kódy umístí na levou stranu

Takovýto strom se pak dá reprezentovat pomocí posloupnosti délek kódů, které se snadněji zkomprimují. Tyto posloupnosti délek kódů jsou však ještě dále zakódovány pomocí Huffmanových kódů z tabulky 3.12.

Po dekódování uvedeného v tabulce 3.12, získáme sekvenci délek kódů, z kterých lze následovně získat již finální Huffmanovy kódy, které jsou použité ve výsledných komprimovaných datech od metody LZ77:

Kód	Popis
0 - 15:	Bitová délka 0 - 15
16:	Zkopíruj předcházející bitovou délku 3- až 6-krát.
	Další dva bity určují počet opakování (0 = 3, … , 3 = 6)
17:	Zopakuj nulu 3- až 10-krát. (3 bity pro počet)
18:	Zopakuj nulu 11- až 138-krát. (7 bity pro počet)

Tabulka 3.12: Kódy kódů délek pro sekvence u dynamického Huffmanova kódování[37]

1. V sekvenci se spočítá počet kódů o stejné délce, kde `bl_count[N]` představuje počet kódů délky `N`

2. Najde se nejmenší numerická hodnota pro každou délku:

```
code = 0;
bl_count[0] = 0;
for (bits = 1; bits <= MAX_BITS; bits++) {
    code = (code + bl_count[bits-1]) << 1;
    next_code[bits] = code;
}
```

   kde `next_code[bits]` představuje nejmenší numerickou hodnotu pro kódy délky `bits`.

3. Vytvoří se všechny kódy délek:

```
for (n = 0;  n <= max_code; n++) {
    len = tree[n].Len;
    if (len != 0) {
        tree[n].Code = next_code[len];
        next_code[len]++;
    }
```

   kde `tree[n].Len` představuje délku aktuálních kódů a `tree[n].Code` představuje výsledný Huffmanův kód pro prvek `n`.

Po definování podmínek pro kódování zbývá popsat strukturu datového bloku za hlavičkou bloku:

- 5 bitů - `HLIT` - počet použitých kódů literálů a délek.
- 5 bitů - `HDIST` - počet použitých kódů vzdáleností.
- 4 bity - `HCLEN` - počet použitých kódů z tabulky 3.12.
- `(HCLEN + 4) * 3` bitů - délky kódů z tabulky. 3.12, které mají fixní podobu: 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15.
- `HLIT + 257` délek kódů Huffmanova stromu s literály a délkami, které jsou kódovány pomocí Huffmanova kódování popsaného výše.

- `HDIST + 1` délek kódů Huffmanova stromu se vzdálenostmi, opět zakódovaných pomocí výše uvedeného Huffmana.
- Samotná zkomprimovaná data.
- Konec bloku označený kódem 256.

Dekomprese
Dekódování všech popsaných kompresních metod lze popsat následovně[37]:

```
do
   read block header from input stream.
   if BTYPE = 00        // no compression
      skip any remaining bits in current partially processed byte
      read LEN and NLEN
      copy LEN bytes of data to output
   else
      if BTYPE = 10     // dynamic Huffman
         read representation of code trees
      loop (until end of block code recognized)
         decode literal/length value from input stream
         if value < 256
            copy value (literal byte) to output stream
         else
            if value = end of block (256)
               break from loop
            else (value = 257..285)
               decode distance from input stream

               move backwards distance bytes in the output
               stream, and copy length bytes from this
               position to the output stream.
      end loop
while not last block
```

Kapitola 4

Útoky

Šifry vždy poutaly pozornost útočníků, kteří se snažili dané šifry prolomit a získat tak zašifrovaný soubor či klíč, díky kterému byl soubor zašifrován. Nejinak je tomu i u šifrovaných archivů. Budou zde popsány jak obecné útoky pro zjištění hesla, tak i specifické útoky pro jednotlivé šifry.

4.1 Brute-force

Brute force, neboli útok silou, popisuje svoji činnost již v názvu. Snažíme se k dané šifře naleznout klíč pomocí vyzkoušení všech možných, za sebou jdoucích, kombinací klíčů. Jednotlivé klíče generujeme v okamžiku použití a nikam je neukládáme. Pokud tedy vygenerovaný klíč není ten správný, zahazujeme jej a generujeme další. V nejhorším případě může tento útok prohledat celý stavový prostor daný kombinací klíčů. Například je-li velikost klíče 10 bitů, prohledávaný prostor má velikost 2^{10} bitů = 1024 možných kombinací.

Velikost prohledávaného prostoru vzrůstá s každým přibývajícím znakem klíče exponenciálně, proto na konvenčních strojích může nalezení klíče trvat od jednotek sekund po desítky let. Na době hledání závisí i množina znaků vyskytujících se v klíči. Klíč složený pouze z písmen malé abecedy bude brute-force útoku odolávat kratší dobu než stejně dlouhý klíč složený z malých i velkých písmen abecedy a numerických znaků. Obecně platí, že čím více použitých znaků se podílí na tvorbě klíče, tím delší dobu bude trvat provedení brute-force útoku.

Příklady časové náročnosti pro různé klíče jsou v tabulce 4.1.

Časová složitost je tedy exponenciální $O(2^N)$ a prostorová složitost je lineání $O(N)$ pro délku klíče N.

Proti brute-force útoku neexistuje žádná ochrana, každý klíč je možné prolomit v konečném čase pomocí tohoto útoku. Avšak je zde důležitá položka „v konečném čase“. Pokud lze klíč prolomit za dva dny a útočník o tom ví (nějakým způsobem zná např. délku klíče), tak tento útok silou provede. Naproti tomu, útok na klíč, který by trval desítky let, útočník ani zkoušet nebude a poohlédne se po jiném způsobu útoku na daný klíč. Z toho vyplývá, že brute-force útok je až poslední variantou zjištění klíče, pokud žádný jiný typ útoku není úspešný.

		Délka hesla						
		2	3	4	5	6	7	8
10	**Variant**	100	1000	10 tis.	100 tis.	1 mil.	10 mil.	100 mil.
zn.	**Čas**	hned	hned	hned	hned	hned	10s	1,5 min
26	**Variant**	676	17576	456876	12 mil.	309 mil.	8 mld.	200 mld.
zn.	**Čas**	hned	hned	hned	12s	5 min	2,25 hod.	2,5 dne
62	**Variant**	3844	238328	15 mil.	916 mil.	57 mld.	3,5 bil.	218 bil.
zn.	**Čas**	hned	hned	15s	15,25 min	16 hod.	41 dní	7 let
96	**Variant**	9216	884736	85 mil.	8 mld.	782 mld.	75 bil.	7,2 bld.
zn.	**Čas**	hned	hned	1,5 min	2,25 hod.	9 dní	2,5 let	229 let

10 znaků	číslice
26 znaků	anglická abeceda (jen malá či jen velká písmena)
62 znaků	anglická abeceda, malá i velká písmena, číslice
96 znaků	anglická abeceda, malá i velká písmena, číslice a běžné znaky NEBO česká abeceda, malá i velká písmena, číslice

Tabulka 4.1: Časové náročnosti Brute-force (1 mil. hesel/s) [17]

4.2 Dictionary útok

Slovníkový útok vychází ze seznamu uložených slov/frází/hashí (slovníku), které mohou být použity jako heslo/klíč. Všechny jednotlivé položky seznamu se poté aplikují na šifru s nadějí, že některé slovo ze slovníku bude to správné. Úspěšnost tohoto útoku vychází z faktu, že většina lidí používá jako heslo jednoduché a běžné fráze: „heslo“, „12345“, „qwertz“ a jiné [6]. Oproti útoku hrubou silou není nutné prohledávat celý stavový prostor tvořený délkou hesla, ale stačí projít slovník. Avšak toto nezaručuje úspěšné nalezení hesla - pokud heslo není přítomno ve slovníku, útok bude neúspěšný.

Časová i prostorová složitost jsou lineární $O(N)$ při použití slovníku o velikost N.

Slovník lze vytvořit z čehokoliv - ze zdrojů z internetu, vyextrahováním slov z knih, vygenerováním hashů atd.

Pokud je však slovník tvořen vygenerovanými hashi, úspěšný útok bude znamenat pouze naleznutí hashe odpovídajícího hesla, ne hesla samotného, z kterého byl vygenerován hash. V některých případech je však toto řešení dostačující.

Proti slovníkovým útokům se dá bránit použitím netradičních hesel, které nejsou tvořeny slovy. Nejideálnější jsou dlouhá hesla tvořená posloupností všech písmen obou velikostí, číslic a speciálních znaků, které dohromady netvoří smysluplné slovo či text. Avšak i obyčejné heslo „`password`“ lze pomocí jednoduché operace přetvořit na „`p0aq1sw2sw2w2o9r4de3`“, kdy písmena hesla doplníme znaky, které se nacházejí nad daným znakem na klávesnici - např. za písmeno `d` doplníme znaky `e,3`. Takovýto postup nesplňuje Kerchhoffův princip [11], avšak každý uživatel si může obdobný postup vymyslet sám a obrana vůči klasickému slovníkovému útoku i brute-force je zajištěna.

4.3 Rainbow tabulky

Tyto tabulky jsou zvláštním případem slovníku, který uchovává heslo a k němu příslušný vygenerovaný hash. Rainbow tabulky jsou navrhnuty tak, že umožňují rychlé prohledání všech hash hodnot [16]. Útok tedy zjišťuje, zda-li se příslušný hash nachází v tabulce a pokud je hash nalezen, je nalezeno i příslušné heslo.

Časová složitost vyhledání v hash tabulce je v tomto případě konstantní $O(1)$, prostorová je lineární $O(N)$ pro tabulku o velikosti N.

Rainbow tabulky je však nutno před použitím vygenerovat či získat z jiných zdrojů. Samotné vygenerování se odvíjí od použité hashovací funkce, maximální velikosti generovaných hesel, množině použitých znaků a typu kódování, v kterém jsou hesla napsána. V případě krátkých hesel a výpočetně málo náročné hashovací funkce (např. MD5) může vygenerování Rainbow tabulky trvat řádově hodiny a výsledný soubor může mít stovky MB. Naopak dlouhá hesla a výpočetně náročné hashovací funkce (např. SHA1 či kombinace více hashovacích funkcí) způsobí generování trvající i týdny s výsledným souborem o velikosti několik GB.

Obecně pro hash funkce je časová i prostorová složitost lineární $O(N)$, záleží tedy na použité hashovací funkci, zda-li časové a prostorové složitosti budou lepší či horší.

Obranou je při hashování hesla používat spolu s heslem pseudonáhodnou sekvenci (salt), která může být volně přístupná. Obrana to není absolutní, avšak při použití dostatečně velké hodnoty salt bude vygenerování Rainbow tabulky trvat výrazně delší dobu než bez jeho použití. Bezpečnost se dá zvýšit i kombinací hashovacích funkcí - `md5(password + salt)` je kryptograficky slabší než `sha1(password + md5(sha1(password + salt)))`. Vytvoření takového hashe vyžaduje delší časový interval, který je pro ověření hesla přípustný, avšak pro tvorbu Rainbow tabulky představuje výpočetní a časový problém (za předpokladu, že útočník vůbec zjistí pořadí a kombinaci hashovacích funkcí).

4.4 PKWARE útoky

4.4.1 Known-plaintext útok

Známým útokem na PKWARE šifru je known-plaintext útok [34, 4], kdy máme k dispozici odpovídající části vstupního a šifrovaného souboru. Pro úspěšný útok je potřeba znát minimálně 13-bytů vstupního souboru a k němu odpovídajícího šifrovaného souboru. S touto znalostí vytvoříme $key3_1$ až $key3_n$, kde n představuje počet bytů vstupního souboru. $key3$ je výstupem funkce `update_keys()` (str. 26), jejíž vstupem byl vstupní soubor. Samotné $key3_i$ se poté vytvoří pomocí XOR: $key3_i = P_i \oplus C_i$, kde P_i je i-byte vstupního souboru a C_i je i-byte šifrovaného souboru. Princip útoku pak spočívá ve vyřešení rovnic v `update_keys()` a zjištění původních inicializačních hodnot klíčů $key[0]$, $key[1]$ a $key[2]$.

Idea útoku je taková, že se postupně vytváří seznamy (listy) všech možných vytipovaných kombinací pro klíče key[2], key[1] a nakonec key[0]. Jakmile jsou tyto seznamy vytvořeny, začneme šifrovat vstupní soubor pomocí klíčů ze seznamu a postupně budeme zahazovat nesprávně otipované klíče, protože budou generovat nesprávný šifrovaný soubor. Tím získáme pouze jednu variatu pro key[0], key[1] a key[2].

Časová složitost tohoto útoku se odvíjí od počtu zjištěných bytů, kde příklady jsou znázorněny v tabulce 4.2. Obecně je časová složitost exponenciální $O(2^N)$ a prostorová $O(N)$ pro klíč o velikosti N.

Známé byty	13	40	110	310	510	1000	4000	10000
Složitost	2^{38}	2^{34}	2^{32}	2^{31}	2^{30}	2^{29}	2^{28}	2^{27}

Tabulka 4.2: Časové složitosti Known-plaintext útoku[4]

4.4.2 Man-in-the-middle útok

Ze sekce 3.2 víme, že WinZip používá pro zašifrování AES v Counter módu a pro autentizaci zašifrovaného textu poté hashovací funkci HMAC-SHA1.

Je však známo [3], že pole pro použitou kompresní metodu a velikost původního souboru se neautentizují pomocí HMAC-SHA1. Tuto skutečnost lze využít k útoku na archiv za použití man-in-the-middle[11, 3, 15]. Příklad útoku může vypadat takto:

Alice a Bob si dohodnou společné tajné heslo, Alice vytvoří ze souboru `F.dat` zašifrovaný soubor `F.zip` a zašle jej přes komunikační kanál Bobovi. Na tomto kanále však naslouchá Mallory a soubor `F.zip` odchytí. Mallory pozmění použitou kompresní metodu a velikost úvodního souboru, protože tyto metadata nebyla autentizována. Takto upravený soubor `F-prime.zip` zašle Bobovi. Ten přijatý soubor úspěšně dešifruje (zná heslo), avšak kvůli špatné kompresní metodě a velikosti původního souboru neuvidí původní soubor `F.dat`, ale pouze „rozsypaný čaj“ `G`. Bob zašle Alici přes komunikační kanál zprávu o „rozsypaném čaji“. Tuto zprávu však odchytí Mallory. Mallory, vydávající se za Alici, přesvědčí Boba o zaslání „rozsypaného čaje“ `G` a z takto získaného `G` poté Mallory zrekonstrujuje původní soubor `F.dat`.

4.5 AES útoky

4.5.1 Kryptoanalýza pomocí úplného bipartitního grafu

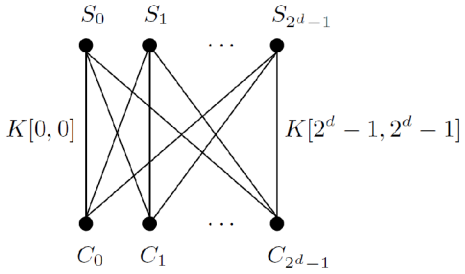
Tento útok popsaný v článku *Biclique Cryptoanalysis of the Full AES*[5] využívá útok meet-in-the-middle se spojením úplného bipartitního grafu. K jeho činnosti je však potřeba mít k dispozici alespoň jeden odpovídající pár vstupního a šifrovaného souboru. Taktéž není o mnoho rychlejší než brute-force útok a jak řekli sami tvůrci tohoto útoku: „Tímto útokem lze učinit z AES-128 za pár minut AES-126.“[38]

Meet-in-the-middle útok probíhá ze dvou stran šifry, jedna část jde od počátku a pomocí vstupního souboru provádí šifrování. Druhá část jde od konce se šifrovaným souborem a snaží se jej dešifrovat. Obě části pracují s množinou klíčů a jakmile se potkají uprostřed šifrovacího procesu, porovnají si množinu klíčů, díky které se dostaly do poloviny šifrovacího procesu. Pokud se podaří naleznout souhlasné klíče, šifra je prolomena. Tato metoda je však náročná na výpočetní i paměťový prostor a pravděpodobnost, že se podaří naleznout souhlasné klíče je nízká. Pravděpodobnost se dá však zvýšit pomocí úplného bipartitního grafu.

Úplný bipartitní graf je dvojice množin uzlů V a hran E: $G = (V, E)$, kde platí: $V = V_1 \cup V_2,\ V_1 \cap V_2 = \emptyset$ a dále: $\forall e = \{u, v\}, e \in E : u \in V_1 \wedge v \in V_2$ a současně $E = V_1 \times V_2$.

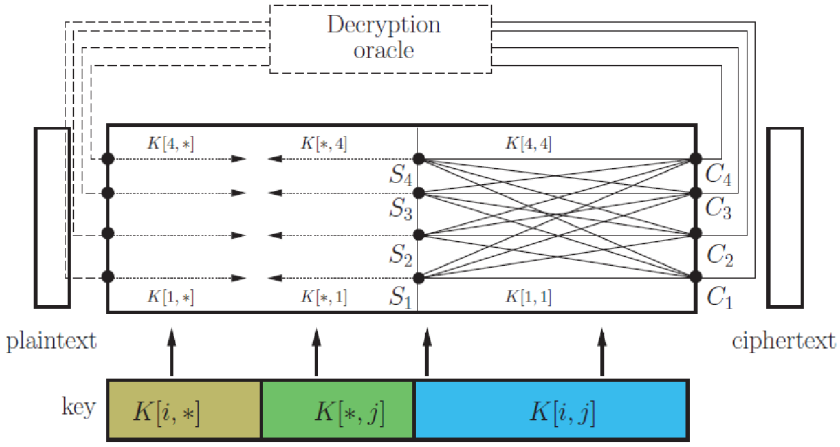
Na obrázku 4.1 odpovídají uzly S_i mezistavům při útoku meet-in-the-middle, uzly C_i představují šifrovaný soubor a hrany $K[i, j]$ reprezentují klíče z uzlu S_i do uzlu C_j.

Kryptoanalýza dle [5] spojuje meet-in-the-middle útok s úplným bipartitním grafem pro



Obrázek 4.1: Úplný bipartitní graf [5]

zvýšení pravděpodobnosti nalezení shodných klíčů. Na obrázku 4.2 komponenta „Decryption oracle“ představuje „věštce“, který generuje takové klíče, u nichž se předpokládá, že budou pro danou dvojici vstupní-šifrovaný soubor správné.



Obrázek 4.2: Princip útoku chosen-plaintext [5]

4.5.2 Chosen-plaintext

Útok popsaný samotným tvůrcem AES Vincentem Rijmenem [28] předpokládá přístup k šifrovacímu systému, kdy útočník může pozastavit proces dešifrování, pozměnit vstup a pokračovat v šifrování. Výstupem pak bude pozměněná část vstupu, pomocí kterého může útočník získat šifrovací klíč[1].

Rijmen vychází z předpokladu, že jsou k dispozici dva Rozdíly δ a ϵ. δ je 16-bytový řetězec, který si útočník může libovolně zvolit, ϵ poté definuje jako $\epsilon = \text{ShiftRows}^{-1}(\text{MixColumns}^{-1}(\delta))$. Poté existuje dvojice $\{p, p^*\}$, kde p je vstup zvolen útočníkem a p^* je vstup odpovídající vztahu:

$$p^* = R_k^{-1}(R_k(p) + \delta),$$

kde $R_k(x)$ představuje jedno kolo transformací AES algoritmu, k představuje klíč a x vstup transformace.

[1]Podstatou chosen-plaintext útoku je možnost zvolit vstup, od šifrovacího systému získat příslušný šifrovaný výstup a poté zjistit použitý šifrovací klíč.

Klíč k lze pak získat z rovnice:

$$\text{SubBytes}(p + k) + \text{SubBytes}(p^* + k) = \epsilon$$

Pokud zvolením správného δ dosáhneme toho, aby všechny byty ϵ byly nenulové, Rijmen udává, že tato rovnice má maximálně 2^{32} řešení.

K tomuto útoku není potřeba znát šifrovaný výstup. Velikost zjišťovaného klíče je tímto útokem snížena z 128-bitů na 32-bitů. Avšak, jak bylo popsáno výše, je potřeba mít k dispozici šifrovací systém, jehož součástí je šifrovací klíč.

Pro úspěšné nalezení klíčů u AES-128 je potřeba $2^{126,1}$ výpočetních operací, pro AES-192 operací $2^{189,7}$ a pro AES-256 $2^{254,4}$. V porovnání s brute-force útokem, který potřebuje v případě AES-128 2^{128} operací, není zrychlení dostatečné.

Časová složitost je tedy také exponenciální $O(2^N)$ a prostorová lineární $O(N)$ pro N představující velikost klíče.

4.6 Simulace útoků

Pro vlastní simulaci útoků na heslem chráněné archivy bylo použito volně dostupné trial verze programu Advanced Archive Password Recovery [8]. Z důvodu toho, že o obsahu chráněného archivu nebudeme mít žádné informace, nezbývá než použít obecné útoky na chráněné archivy - brute-force a dictionary útok. Ostatní útoky popsané v této práci vyžadují ke své činnosti alespoň některé byty vstupního souboru, který ovšem nemáme k dispozici.

Pro testování lámání hesel byl použit jednojádrový procesor o frekvenci 1,6 GHz a veškeré hodnoty se odvíjejí od této konfigurace. Lepších časů by bylo možné dosáhnout pomocí vícejádrových procesorů či víceprocesorových počítačů, nejlépe pak pomocí svazku distribuovaných počítačů, ovšem finanční stránka tohoto řešení je diskutabilní.

4.6.1 Počet hesel za sekundu

Každý formát zajišťuje šifrování jiným způsobem, proto i průměrné časy prohledaných hesel za sekundu se liší.

	Délka hesla				
Formát	2	4	6	8	10
Zip PKWARE	11 mil.	11 mil.	11 mil.	11 mil.	11 mil.
Zip AES 256	3 500	3 500	3 500	3 500	3 500
RAR AES 128	96	75	65	57	50
7z AES 256	96	74	63	56	49

Tabulka 4.3: Průměrné rychlosti počtů hesel za sekundu pro 95 použitých znaků

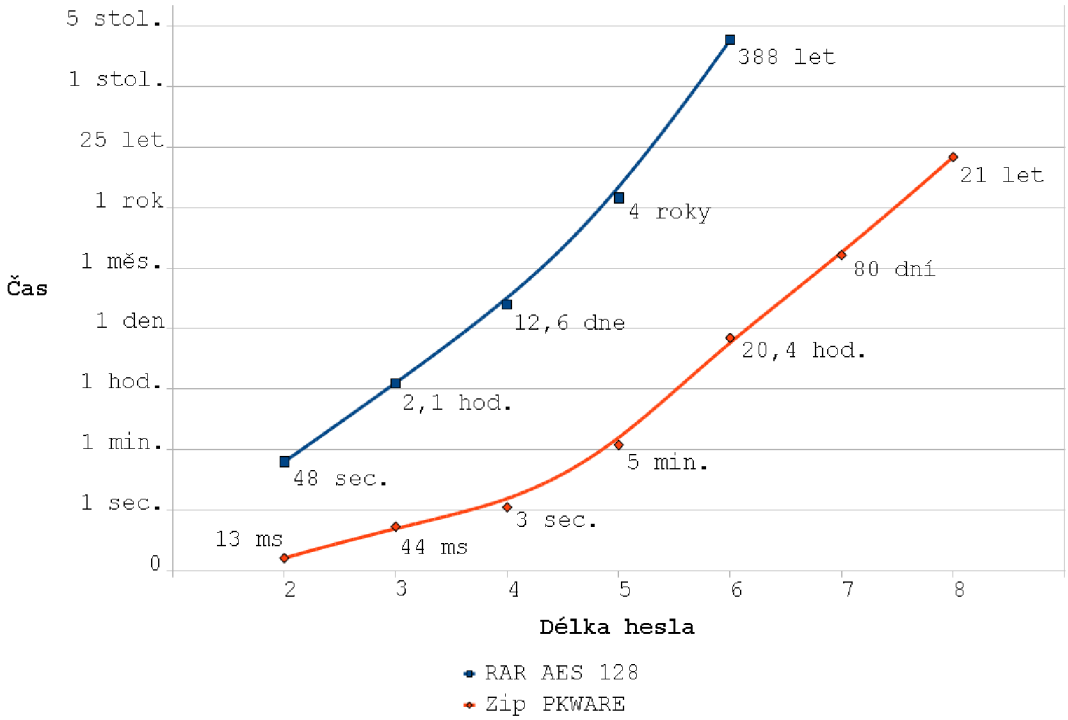
V tabulce 4.3 je vidět, že Zip PKWARE používá slabou šifru, která není výpočetně náročná a lze tak rychle vyzkoušet mnoho hesel. U formátu RAR a 7z, které používají šifru AES jsou rychlosti podobné a s každým dalším znakem hesla se samotných hesel prohledá méně, kvůli náročnosti výpočtu AES (AES 256 se počítá o něco déle než AES 128). Jediná odchylka ohledně AES je u Zip AES, kde je počet hesel za sekundu víceméně konstantní a nezávisí na délce hesla. To je pravděpodobně způsobeno položkou *Password verification value* (str. 20), které poskytuje prvotní kontrolu hesla bez nutnosti dešifrovat AES.

4.6.2 Brute-force

Brute-force útoky byly prováděny na předpřipravené zaheslované archivy, každý útok začínal prohledávat množinu hesel tvořenou 95 znaky od dvou-místných hesel a pokračoval až do doby, než heslo nalezl. Tyto útoky byly prováděny pouze s archivy zaheslovanými nejvýše pěti-místným heslem, delší hesla jsou již vypočítány teoreticky, protože by jejich skutečné nalezení trvalo příliš dlouho. V tabulce 4.4 jsou teoreticky spočítané časy pro delší hesla oddělené čarou.

Formát	Délka hesla	Kombinací	Rychlost h/s	Čas
RAR AES 128	2	4 685	95	48 s
	3	795 386	94	2,1 hod
	4	82 mil.	75	12,6 dnů
	5	7,8 mld.	70	4 roky
Zip PKWARE	2	4 656	11 mil.	13 ms
	3	453 911	11 mil.	44 ms
	4	43 mil.	11 mil.	3 s 7 ms
	5	4 mld.	11 mil.	5 min 51 s
	6	7^{11}	11 mil.	20,4 hod
	7	$6,9^{13}$	11 mil.	80 dní
	8	$6,6^{15}$	11 mil.	21 let

Tabulka 4.4: Získání hesla pomocí Brute-force



Obrázek 4.3: Získání hesla pomocí Brute-force

4.6.3 Dictionary útok

Rychlost prohledávání hesel odpovídá tabulce 4.3, avšak zde se hesla negenerují, ale vybírají z externího souboru. Čas potřebný k případnému nalezení hesla se poté odvíjí jen a pouze od velikosti použitého slovníku. Tento útok nebyl simulován, neboť by bylo nutné nejprve vytvořit slovník, který by v nejhorším případě vypadal stejně jako vygenerovaná hesla brute-force útokem. Časy k nalezení hesla by tedy odpovídaly tabulce 4.4.

4.7 Zhodnocení útoků

Souhrnné zhodnocení popsaných útoků je v tabulce 4.5, kde se nejlepší metodou zdá použití Rainbow tabulek. Pokud tento útok má již k dispozici vygenerovou tabulku, která pokrývá množinu použitých klíčů, tak tento útok je opravdu nejrychlejší. Avšak příprava tohoto útoku a především vygenerování tabulky trvá nějaký čas a není zde jistota, že vygenerovaná množina hashů pokryje množinu klíčů. Je tedy zřejmá existence útoků, které jsou mírně lepší než útok silou, avšak v celkovém důsledku jsou všechny útoky na úrovni právě útoku pomocí hrubé síly.

	Časová složitost	Prostorová složitost
Brute-force	$O(2^N)$	$O(N)$
Slovníkový	$O(N)$	$O(N)$
Rainbow s vytvořenou tabulkou	$O(1)$	$O(N)$
Rainbow bez vytvořené tabulky	$O(N)$	$O(N)$
Known-plaintext	$O(2^N)$	$O(N)$
Chosen-plaintext	$O(2^N)$	$O(N)$

Tabulka 4.5: Časové a prostorové složitosti útoků na archivy

Kapitola 5

Návrh a implementace

Hlavní náplní praktické části této práce je navrhnout a implementovat extrakci hesel ze samorozbalovacích archivů RAR, Zip a 7z na platformě Windows - nejčastějšího cíle malware. Z hlediska čitelnosti by bylo dobré od sebe oddělit návrh řešení problému a jeho následnou implementaci, avšak v tomto případě se tyto dva postupy v mnoha částech prolínají a proto budou popsány v rámci jedné kapitoly.

Pro extrakci hesel ze samorozbalovacích archivů, které se používají pro vnitřní samorozbalovací archiv, je nejprve nutno určit, jak se samotné heslo aplikuje na vnitřní samorozbalovací archiv, na jakém místě v souboru je toto heslo uloženo, v jaké formě, a teprve po zjištění těchto všech informací je možné samotné heslo vyextrahovat.

Průzkum struktur probíhal na samorozbalovacích archivech RAR, 7z a Zip, které byly vytvořeny ve svých oficiálních (a nejpoužívanějších) programech - WinRAR, 7z a WinZip Self-Extractor. Navíc z těchto programů pouze WinRAR umožňuje generovat i samorozbalovací archivu ve formátu Zip. Ostatní zmíněné programy tvoří pouze samorozbalovací archivy svých formátů, ostatní nikoliv (většinou z licenčních důvodů). Existují i jiné programy, které umožňují vytvářet samorozbalovací archivy daných formátů, avšak ty tu nebudou popsány.

Také je důležité správně detekovat typ formátu samorozbalovacího archivu, který se odvíjí od struktury PE programu a samotného archivu.

5.1 Aplikace hesla na archiv

První věcí, kterou je potřeba zjistit, je aplikace samotného hesla na vnitřní zaheslovaný samorozbalovací archiv. Bez této znalosti není možné lokalizovat samotné heslo uložené kdesi v archivu a tudíž jej nelze ani vyextrahovat.

Všechny archivační formáty mají možnost spuštění samorozbalovacího archivu skrze příkazovou řádku. Díky tomu lze na archiv aplikovat spoustu přepínačů a příkazů, které s archivem dokáží provést o mnohem více funkcí, než při spouštění přes grafické uživatelské rozhraní.

Z dokumentace a zkušeností uživatelů byly zjištěny následující přepínače, které umožní zaheslovanému samorozbalovacímu archivu skrze příkazovou žádku zadat heslo:

- RAR - přepínač `-pPassword`, kde *Password* je heslo pro daný archiv
- 7z - taktéž přepínač `-pPassword`
- Zip - neumožňuje zadat heslo přes příkazovou řádku, pouze přes dialogové okno

Po tomto jednoduchém zjištění aplikace hesla na zaheslovaný archiv bylo nutné zjistit, jak vnější samorozbalovací archiv spouští vnitřní zaheslovaný samorozbalovací archiv s příslušnými parametry, včetně hesla. Každý archivační formát používá konfigurační informace, které upřesňují co a jak se má provést, avšak formát zápisu těchto konfiguračních informací se u jednotlivých archivačních formátů liší. Pro jednotlivé formáty bude popsána tvorba vlastních samorozbalovacích archivů a struktura samotných konfiguračních informací.

5.2 Tvorba archivu

Vytvoření samorozbalovacího archivu, který umí rozbalit a spustit zaheslovaný vnitřní archiv, řeší každý archivační formát a program po svém. Někde stačí vše naklikat v oficiálním programu, někde je nutné kombinovat více postupů.

V následující sekci budou pro jednotlivé soubory použity tyto popisy:

- `In.exe` pro vnitřní spustitelný soubor (většinou samotný malware)
- `InSFX.exe` pro vnitřní zaheslovaný samorozbalovací archiv, který spouští soubor `In.exe`
- `OutSFX.exe` pro vnější samorozbalovací archiv, který spouští vnitřní archiv `InSFX.exe`

5.2.1 RAR

Oficiální program WinRAR umožňuje vše vytvořit v rámci svého GUI. Pro tvorbu `InSFX.exe` postačuje při vytváření archivu z `In.exe` zaškrtnout volby „Create SFX archive“ a „Lock archive“ a následně v záložce „Advanced“ specifikovat heslo pro archiv (*Set password...*) a určit jméno programu, který se má po provedení extrakce provést (*SFX Options...* $->$ *Run after extraction* $->$ `In.exe`).

Vytvoření `OutSFX.exe` je stejné jako u vnitřního archivu. Pouze se neprovede uzamčení archivu a v části *Run after extraction* se spolu s jménem spustitelného souboru uvede i přepínač s heslem pro `InSFX.exe`. Tedy: „*Run after extraction:* `InSFX.exe -pHeslo-k-InSFX`“.

5.2.2 7z

Vygenerovat z `In.exe` jen samorozbalovací archiv nebo jen zaheslovaný archiv 7z není problém. Ten nastává při vytvoření zaheslovaného a zároveň samorozbalovacího archivu - vytvoří se sice bez jakékoliv chyby, avšak při pokusu o spuštění takového archivu se zobrazí hláška `Unsupported method`, což je zapříčiněno pravděpodobně špatnou implementací samotného archivačního programu 7z. Jako `InSFX.exe` se tedy použije archiv vytvořený pomocí programu WinRAR.

`InSFX.exe` pomocí GUI verze programu 7z zkomprimujeme na archiv `InSFX.7z`. Dále je nutné mít k dispozici SFX modul `7zsd.sfx` (dostupný z internetu[30]) a konfigurační soubor `config.txt`, který musí být uložen v kódování UTF-8 a jehož tvar je popsán dále. Jsou-li všechny soubory na jednom místě, postačuje je sloučit do jednoho pomocí příkazu:

```
copy /b 7zsd.sfx + config.txt + InSFX.7z OutSFX.exe
```

Tím je tvorba celého samorozbalovacího archivu 7z hotova.

5.2.3 Zip

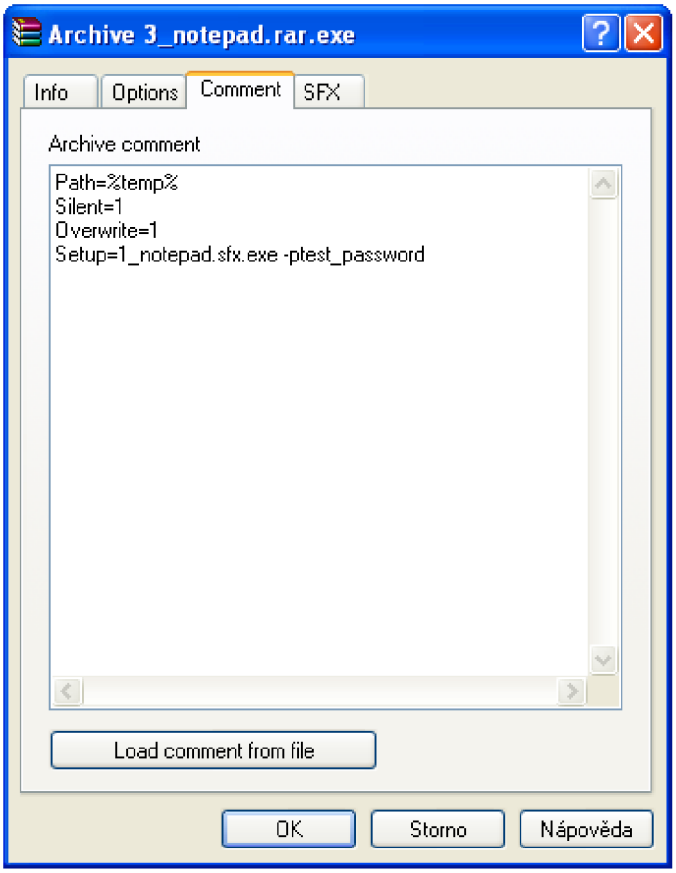
Zaheslované samorozbalovací archivy Zip neumožňují zadat heslo přes příkazovou řádku, proto i zde bude jako vnitřní samorozbalovací archiv použit archiv vygenerovaný pomocí programu WinRAR.

Dostupnými nástroji se `InSFX.exe` zkomprimuje na soubor `InSFX.zip` ve formátu Zip. Tento Zip soubor se pak použije v průvodci programu WinZip Self-Extractor, kde je možno naklikat parametry archivu, včetně příkazu po spuštění ve formě:
„`.\InSFX.exe -pHeslo-k-InSFX`“.

5.3 Tvar konfiguračních informací

5.3.1 RAR

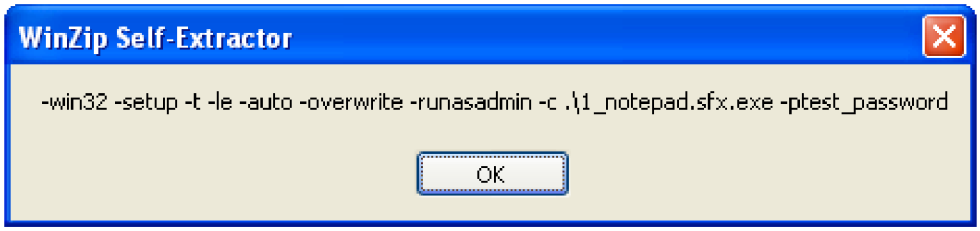
Kromě příkazu pro spuštění s přepínačem pro hesla jde naklikat spoustu parametrů, které se ukládají v textové podobě do části, kterou WinRAR pojmenoval „Komentář“ (Comment). K vizuální podobě Komentáře se lze doklikat v GUI verzi programu a která je znázorněna na obrázku 5.1.



Obrázek 5.1: Konfigurační informace WinRAR

5.3.2 Zip

I zde jsou naklikané parametry uloženy v textové podobě, kterou lze zobrazit pouze v příkazové řádce použitím přepínače `-info` na vytvořeném samorozbalovacím archivu Zip (obrázek 5.2).



Obrázek 5.2: Konfigurační informace Zip

5.3.3 7z

Konfigurační informace jsou uloženy ve speciálním souboru `config.txt`, který má svoji syntaxi a který se přímo vkládá do samorozbalovacího archivu. Konstrukce a možnosti tohoto konfiguračního souboru jsou autorem popsány v dokumentaci[30], je ovšem nutné, aby byl uložen v kódování UTF-8. Nejdůležitejším parametrem je především `RunProgram`, specifikující co se má provést po extrakci archivu.

Zobrazení obsahu konfiguračního souboru však není možné bez externích programů, které jsou na to specializované. Samotný konfigurační soubor `config.txt` však může vypadat následovně:

```
;!@Install@!UTF-8!
Title="Notepad"
RunProgram="1_notepad.sfx.exe -ptest_password"
;!@InstallEnd@!
```

5.3.4 Zip pomocí WinRAR

U samorozbalovacího archivu Zip, který byl vytvořen pomocí jiného programu (WinRAR) je názorně vidět, že každý tvůrce software si řeší strukturu samorozbalovacího archivu po svém. V tomto případě se tvar konfiguračních informací vůbec nepodobá informacím v samorozbalovacích souborech Zip, ale je totožný s RAR jako na obrázku 5.1.

5.4 Uložení konfiguračních informací

Se znalostí tvaru konfiguračních informací je na řadě zjištění, na jakém místě v PE programu (sekce 2.5) se tyto informace nacházejí a v jaké formě.

5.4.1 RAR

RAR jako jediný z formátů poskytuje volně k dispozici knihovnu i samotné zdrojové kódy pro extrakci komprimovaných archivů ve formátu .rar. Proto není nutné zjišťovat již po-

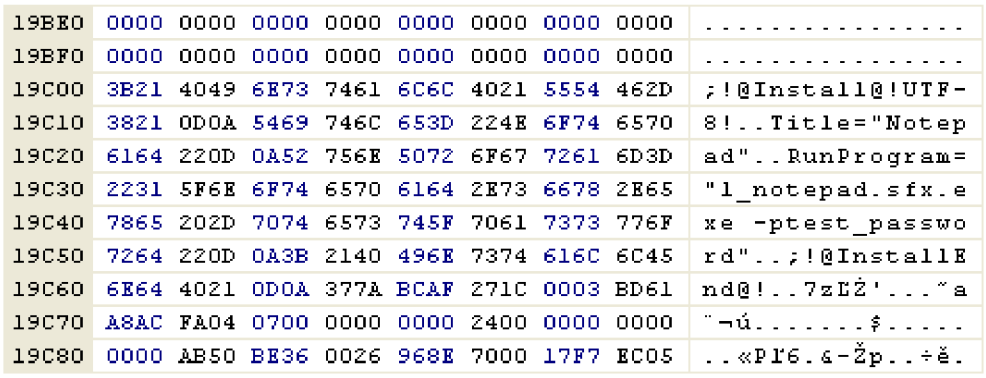
psané věci a stačí využít dostupnou knihovnu nebo se podívat do zdrojových kódů a vyčíst potřebné informace.

Avšak pro doplnění, komentář obsahující konfigurační informace se v souboru nachází na pozici, která je dána součtem velikostí spustitelného kódu, velikostí signatury a velikostí prvního *volume header*. Zároveň je celý komentář zkomprimován jednou z metod z tabulky A.9.

5.4.2 7z

Pro 7z i Zip je už nutné ručně zkontrolovat a projít binární data jednotlivých samorozbalovacích archivů a najít v jejich struktuře hledané konfigurační informace.

V případě archivů 7z je průzkum binárních dat jednoduchý, protože samotný konfigurační soubor se nachází v binárních datech jako textový dokument bez jakékoliv komprese či jiné úpravy. Názorně to je vidět na obrázku 5.3.



Obrázek 5.3: Konfigurační soubor v 7z souboru zobrazený pomocí Hex-ditoru

Pozice konfiguračního souboru pak koresponduje s informacemi v hlavičkách sekcí PE souboru, kdy se počátek samotného konfiguračního souboru nachází v úvodních 16 bytech **.rsrc** sekce PE programu - počátek není vždy na počátku .rsrc sekce, někdy konfiguračnímu souboru předchází pár bytů.

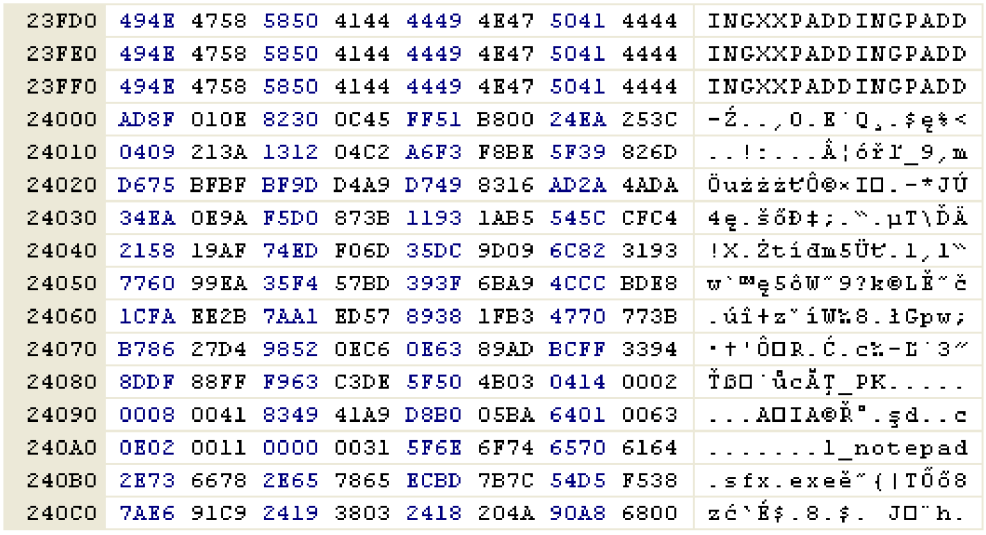
5.4.3 Zip

Oproti 7z je situace u Zip archivů mnohem složitější, neboť i po prozkoumání všech binárních dat nejsou konfigurační informace viditelné v textové podobě.

Bylo tedy nutné lokalizovat alespoň přibližné umístění. K tomu jsem použil metodu porovnání dvou samorozbalovacích archivů, každý se stejným zkomprimovaným souborem, pouze konfigurační informace byly mírně odlišné. Tato metoda pravděpodobné není nejlepší, ale místo uložení konfiguračních informací bylo jednoznačně nalezeno, neboť rozdíly mezi jednotlivými soubory byly pouze v jednom místě.

Jako v případě samorozbalovacích archivů 7z, i u archivů Zip se konfigurační informace nacházejí v úvodu .rsrc sekce PE struktury programu.

Umístění konfiguračních informací je znázorněno na obrázku 5.4, kde samotné konfigurační informace začínají na řádku 24000 a končí na řádku 24080 před signaturou Zip archivu - 0x50 4B 03 04 (tabulka A.1).

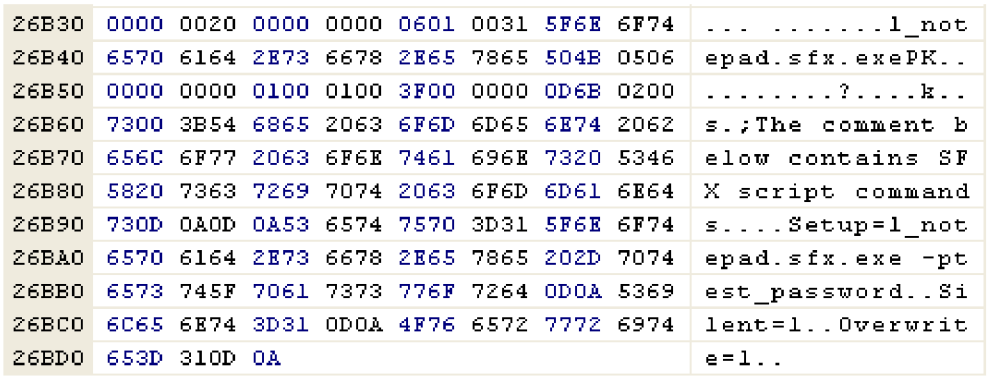


Obrázek 5.4: Konfigurační informace v Zip souboru zobrazené pomocí Hex-ditoru

Ovšem přesný způsob uložení se ze získaných dat nedá určit, i když z podoby dat se dá usuzovat, že samotné konfigurační informace budou pravděpodobně zkomprimovány některou z kompresních metod. Avšak z důvodu toho, že způsoby tvorby vytváření samorozbalovacích archivů jsou zcela v režii autorů, není vyloučené i použití exotických či upravených kompresních metod. Proto bylo nutné způsob uložení konfiguračních informací prozkoumat z bližšího pohledu - až na úrovni strojového kódu.

5.4.4 Zip pomocí WinRAR

Uložení konfiguračních informací je podobné souborům 7z, kdy jsou tyto informace uloženy v podobě čistého textu. Umístění je však netradičně na konci souboru, kdy počátek konfiguračních informací je uvozen řetězcem „*;The comment below contains SFX script commands...*“. Podoba v binární podobě, zobrazená pomocí hexa-editoru, je znázorněna na obrázku 5.5.



Obrázek 5.5: Konfigurační informace Zip souboru vytvořeného pomocí WinRAR a zobrazené pomocí Hex-ditoru

5.5 Extrakce konfiguračních informací

Pro fomáty RAR, 7z a Zip pomocí WinRAR již tedy známe jednoznačné umístění konfiguračních informací v binárních datech nebo máme k dispozici i knihovnu pro extrakci těchto inforamcí, proto extrakce není složitá.

V případě samotného formátu Zip je však potřeba získat strojový kód samorozbalovacího archivu, prozkoumat jej a určit, jak se konfigurační informace extrahují a v jaké formě byly uloženy.

5.5.1 RAR

S využitím knihovny unRAR.dll[27] postačuje příslušný samorozbalovací archiv otevřít pomocí funkce `RAROpenArchiveEx()`, která přímo načte i celý obsah komentáře. Tento komentář poté stačí zkopírovat a zpracovat dle potřeby.

5.5.2 7z

Pro získání pozice konfiguračního souboru vyčteme z hlaviček PE programu offset na počátek .rsrc sekce a velikost sekce, kde součtem těchto dvou položek dostaneme offset na možný počátek dat konfiguračního souboru:

`configurationFileOffset = SizeOfRawData + PointerToRawData` (tabulka 2.2)

Po přesunutí se v otevřeném souboru na možný počátek dat konfiguračního souboru je nutno najít uvozovací a ukončovací řetězce konfiguračního souboru (str. 44) a vše mezi těmito řetězci se musí zkopírovat pro další zpracování.

5.5.3 Zip

Při hledání způsobu, jakým jsou konfigurační informace uloženy, jsem zkoumal mnoho dostupných knihoven pro Zip, zda-li nepopisují způsob extrakce těchto informací. Avšak žádná z nich tuto extrakci neobsahuje. Nicméně bylo zjištěno, že základní kompresní metodou pro soubory uvnitř samorozbalovacího archivu (pokud si uživatel při vytváření archivu nezvolí jinou), je metoda Deflate.

Ze zvědavosti jsem se pokusil binární data konfiguračních informací dekomprimovat metodou Inflate (opak Deflate), která je součástí všech knihoven pro práci se Zip soubory, zda-li konfigurační informace nejsou zkomprimovány právě metodou Deflate. Avšak ani posouváním binárních dat nevracela metoda Inflate čitelné výsledky.

V danou chvíli to tedy vypadalo, že konfigurační informace nejsou komprimovány metodou Deflate.

Poslední možností, která mne napadla, bylo vzít testovací samorozbalovací archiv, diasemblovat jej na strojový kód, postupně projít a identifikovat ty funkce a rutiny, které se podílejí na dekomprimaci konfiguračních informací. Následně pomocí dekompilátoru a ručního převodu ze strojového jazyka do jazyka C dekompresní metodu zprovoznit a pochopit její činnost. Pochopit činnost se dá v mnoha případech i pouze ze strojového jazyka, avšak rozsah takového strojového kódu nesmí být příliš velký, protože běžný člověk, dle mého názoru, není schopen chápat tolik souvislostí.

Diasembler a dekompilátor

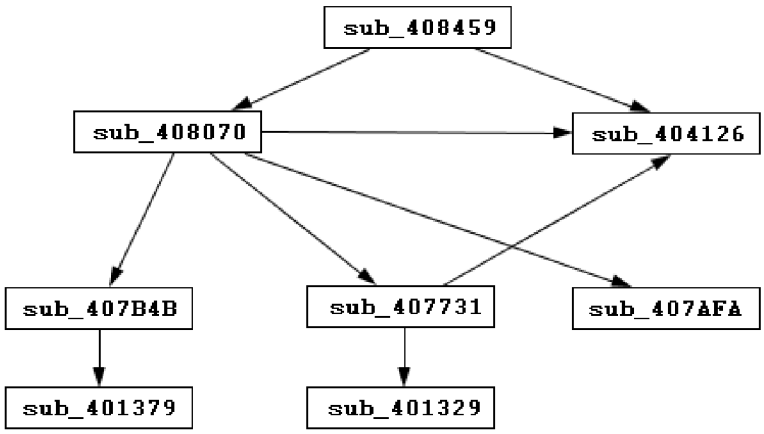
Pro diasemblování testovacího archivu byl využit volně dostupný program IDA Pro Freeware [29], který krom samotného diasemblování umí i zpracováváný soubor debugovat, což je pro souběžnou kontrolu strojového kódu a kódu v jazyce C ideální.

Pro převod ze strojového jazyka do jazyka C byl použit open-source dekompilátor Boomerang[26], kdy výsledný kód v jazyce C dle očekávání (zmíněná v sekci 2.6) nefungoval a nešel ani přeložit kompilátorem.

S předstihem lze napsat, že činnost dekompresoru odpovídá mírně odlišné metodě Inflate. Celá následující část je však psána bez této znalosti i bez znaosti kompresní metody Deflate, proto některé obraty a vysvětlení funkcí neodpovídají skutečné metodě Deflate a představují mé domněnky, které vznikly při prozkoumávání daného kódu.

Studiem a postupným debugováním strojového kódu, který obsahoval přibližně 165 funkcí, procedur či rutin, bylo vyextrahováno pouze 8 funkcí, které mají co dočinění s dekomprimováním konfiguračních informací. Každá funkce volala i jiné funkce, které však nebyly pro činnost dekompresoru důležité a byly tedy ve výsledném C kódu vypuštěny. Proto zůstalo oněch 8 funkcí, které jsou základním kamenem dekompresoru konfiguračních informací.

Graf volání funkcí tohoto dekompresoru je znázorněn na obrázku 5.6, kde obdélníky představují funkce, jejichž jméno bylo diasemblerem vygenerováno na základě řádku, na kterém funkce v binárním kódu začínala. Činnost jednotlivých funkcí byla souběžně kontrolována se strojovým kódem a prioritou bylo tyto funkce zprovoznit, aby je bylo možné zkompilovat a úspěšně spustit. Zjištění smyslu funkcí v globálním měřítku bylo vedlejší a některé zjištěné skutečnosti jsou popsány níže v pořadí dle závislosti jejich složitosti.



Obrázek 5.6: Graf volání funkcí bez znalosti jejich činnosti

sub_404126 Načtení jednoho byte z binárních dat zkomprimovaných konfiguračních informací.

sub_407AFA Uvolnění paměti (`free()`) naalokovaných prostředků, které se vytváří ve funkci `sub_407B4B`.

sub_401329 Zkopírování části řetězce, který již je uložen v jednom z předaných parametrů funkce.

sub_401379 Inicializace části paměti a zkopírování hodnot z parametru do této paměti.

sub_408459 Počáteční inicializace dat, které bude dekompresor potřebovat a samotné spuštění dekompresního procesu.

sub_407B4B Vytváří v paměti jakési pole struktur, kde struktura je tvořena šesticí - první čtyři prvky představují kódy či počty pro další operace, poslední dva prvky nemají z hlediska upraveného dekompresního procesu žádný smysl, neboť obsahují adresu na buffer s výslednými konfiguračními informacemi, který však již není využit (zjišťěno přímo ze strojového kódu).

sub_408070 Tato funkce se stará o celý dekompresní proces, kdy i pomocí dalších funkcí připraví veškeré potřebné proměnné a po dokončení této funkce je k dispozici celé konfigurační informace ve formě prostého textu.

Ač jsou zkomprimované konfigurační informace uloženy jako jednotlivé byty, tato funkce pracuje vždy s určitým počtem bitů a proto je nutné při čtení následujícího bytu (pomocí funkce `sub_404126`) samotný byte upravit a konfrontovat se zbytkem bitů z předchozího bytu.

Na úvodu funkce inicializuje další proměnné a ze vstupu vytvoří 19-ti místné pole, které naplní daty, jež představují nějaký počet výskytů. Toto pole výskytů se následně upraví ve funkci `sub_407B4B`, aby mohlo být použito znovu v druhém zavolání `sub_407B4B`, které vytvoří pole struktur neopakujících se znaků. Třetí zavolání `sub_407B4B` dotvoří pole struktur, avšak opakujících se znaků.

Jakmile jsou tyto dvě pole struktur neopakujících se a opakujících se znaků hotova, funkce `sub_407731` z nich dekomprimuje konfigurační informace a uloží je jako prostý text.

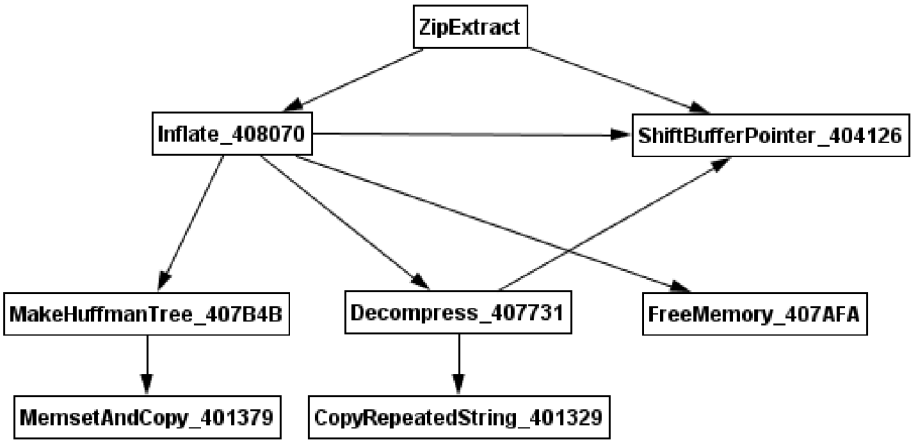
sub_407731 Konečná dekompresní funkce, která z předaných parametrů, které obsahují pole struktur opakujících a neopakujících se znaků, vytvoří finální konfigurační informace.

Jednotlivé bity ze vstupních binárních dat určují, které pole struktur a který znak v tomto poli se použije. V případě neopakujících se znaků se získaný znak uloží do výsledného řetězce konfiguračních informací, v případě opakujících se znaků se část již získaných znaků zkopíruje do výsledného řetězce. Načtením binární hodnoty, která reprezentuje ukončení čtení (hexadecimálně 0x11) dekomprese končí.

S částečnou znalostí činností popsaných funkcí by se dal graf volání funkcí a jejich názvů znázornit pomocí obrázku 5.7.

Exktrakce konfiguračních informací

Po souběžném debugování strojového kódu a kódu v jazyce C byla extrakce konfiguračních informací pro samorozbalovací archivy Zip hotová a plně funkční. Avšak stále nebyla zcela jasná použitá dekompresní metoda. Proto byly části dekompresoru, vzniklého pomocí disembleru a dekompilátoru, porovnány s již existujícími dekompresními metodami, aby se zjistilo, jaký rozdíl je mezi dekompresorem použitým v samorozbalovacích archivech Zip a



Obrázek 5.7: Graf volání funkcí již se znalostí jejich činnosti

dekompresorech běžně používaných v knihovnách pro práci se Zip soubory.

První zkoumanou dekompresní metodou, určenou ke zkoumání, byla metoda Deflate, jelikož soubory v archivu byly komprimovány právě touto metodou. Nicméně dekomprese binárních dat konfiguračních informací pomocí Inflate neposkytovala čitelné výsledky.

Bohužel až v tuto chvíli mne napadlo, proč by samorozbalovací archivy obsahovaly více kompresních metod, když dekompresní metoda použitá pro dekompresi zkomprimovaných souborů může být použita i pro konfigurační informace.

Po nalezení funkčního kódu dekompresní metody Inflate v podobě knihovny `kunzip`[14] bylo možné porovnat mnou vytvořený dekompresor s funkční verzí dekompresoru Inflate.

K mému údivu, počáteční inicializační hodnoty proměnných byly totožné a odpovídaly Inflate dekompresní metodě za použití dynamických Huffmanových kódů. Jeden rozdíl byl u knihovní funkce Inflate, která zcela na počátku kontrolovala první dva byty z binárních zkomprimovaných dat. Tyto dva byty představují použitou kompresní metodu daného Zip souboru a obecné příznaky, avšak tyto dvě položky nejsou v dokumentaci o Zip struktuře popsány.

Po odstranění kontroly prvních dvou bytů v knihovní funkci Inflate byla zkomprimovaná konfigurační data úspěšně a bez problémů dekomprimována na čitelný, prostý text.

Zjistit po měsících práce, že jediným rozdílem mezi Inflate metodou používanou při dekomprimování Zip souborů a mým dekompresorem vzniklým postupným diasemblováním a dekompilací, je pouze v kontrole dvou bytů, nebylo zrovna příjemné.

Se znalostí činnosti Inflate metody bylo alespoň možné porovnat správný kód s popisem mých funkcí `sub_xxxxxx`. Ukázalo se, že mé domnělé činnosti funkcí a dekompresoru byly zhruba správné, jen ony pole struktur (ne)opakujících se znaků byly ve skutečnosti Huffmanovy stromy reprezentující znaky, délky a vzdálenosti v kompresní metodě LZ77. Obecně zjišťovat činnost funkcí z dekompilovaného kódu je dosti obtížné a časové náročné.

5.5.4 Zip pomocí WinRAR

Čistý Zip archiv se zpracovává od konce souboru, na kterém se nachází hlavička ukončující záznamy v *Central directory* (tabulka A.4). Nikde tedy není přítomen offset určující konec souboru, poloha ukončovací hlavičky *Central directory* nebo hodnota velikosti celého archivu. Proto i pro nalezení konfiguračních informací je nutno postupovat od konce souboru. Pro ušetření práce se souborem se načte posledních 0x10000 bytů (maximální velikost komentáře dle RAR specifikace[27]), v kterých se pak pomocí knihovních funkcí jednoduše nalezne odpovídající řetězec „*;The comment below contains SFX script commands...*“ a vše za ním se zkopíruje pro další použití.

5.6 Extrakce hesla

Máme-li k dispozici konfigurační informace uložené v řetězci, postačuje již v řetězci najít úvodní frázi pro spuštění programu (RAR: „`Setup=`“, Zip: „`-c `“, 7z: „`RunProgram="`“) a heslo za příslušným přepínačem zkopírovat.

5.7 Detekce typu archivu

Již víme, odkud a jak úspěšně vyextrahovat heslo k vnitřnímu samorozbalovacímu archivu. Při prvotním seznámení s neznámým souborem pouze zbývá určit, o jaký typ samorozbalovacího archivu se jedná. Po tomto zjištění pak stačí vyhledat a extrahovat heslo.

Pro určení typu archivu budeme vycházet ze signatur jednotlivých archivů, které jednoznačně identifikují soubor (nebo alespoň jeho součást) a ze struktury PE programu. Každá detekce typu archivu začíná nalezením signatury PE programu. Další postup se již pro různé archivy liší.

5.7.1 RAR

Signatura pro RAR je řetězec „Rar!...“, v hexadecimální podobě poté `0x52 0x61 0x72 0x21 0x1A 0x07 0x00`. Tato posloupnost se v binárních datech nachází na počátku **.rsrc** sekce PE programu. Postačuje tedy najít offsety pro .rsrc sekci a zkontrolovat přítomnost RAR signatury.

5.7.2 7z

Archiv 7z je identifikován signaturou „7z....“, hexadecimálně `0x37 0x7A 0xBC 0xAF 0x27 0x1C`. Stejně jako v případě RAR archivů, i tato signatura je v úvodu **.rsrc** sekce. Archivy vytvořené pomocí 7z mohou být ale uvozeny dvě způsoby - signaturou 7z (`0x37 0x7A ...`) nebo počátečním řetězcem konfiguračních informací (`;!@Install@!UTF-8!`).

Je-li na začátku .rsrc sekce přítomna signatura 7z (`0x37 0x7A ...`), celý soubor je sice samorozbalovacím archivem 7z, avšak bez konfiguračních informací, a není nutné tedy extrahovat heslo, neboť tam není. V případě naleznutí řetězce `;!@Install@!UTF-8!` v počátečních datech .rsrc sekce se jedná o samorozbalovací archiv 7z s konfiguračními informacemi, ze kterých může být provedena extrakce hesla.

5.7.3 Zip

Situace u Zip samorozbalovacích archivů je odlišná, neboť archivy vytvořené pomocí WinZip Self-Extractoru nemají žádnou zdokumentovanou signaturu. Proto je nutné opět hledat souvislosti v diasemblovaném kódu.

Důležitá je zde **.data** sekce, obsahující inicializované proměnné pro spustitelný kód samorozbalovacího archivu. Bylo zjištěno, že pro určení místa čtení zkomprimovaných konfiguračních informací v binární podobě, je použita proměnná s názvem `lDistanceToMove`, jejíž hodnota odpovídá hodnotě offsetu počátku .rsrc sekce. Proměnná v .data sekci, uložená před `lDistanceToMove` pak popisuje offset na konec .rsrc sekce.

Detekce Zip samorozbalovacích archivů vychází z obrázku 5.8, který představuje výňatek z kódu, který vygeneroval diasembler. Na obrázku je zobrazen počátek .data sekce, kde řádek `.data:00417054` značí hodnotu pozice začátku .rsrc sekce. Zkoumáním Zip samorozbalovacích archivů s různými soubory a konfiguračními informacemi bylo zjištěno, že počet proměnných před `lDistanceToMove` se různí, avšak řetězec „`NMC sfx`“ a za ním následující proměnné jsou vždy stejné. Samotná detekce vychází právě z tohoto řetězce „`NMC sfx`“.

Určení, zda-li je soubor samorozbalovací archiv typu Zip, začíná nalezením počátku .data sekce. Zde se následně lokalizuje řetězec „`NMC sfx`“, od kterého je offset na proměnnou `lDistanceToMove` u každého Zip samorozbalovacího archivu vždy stejný. V případě, že hodnota `lDistanceToMove` je totožná s hodnotou počátku .rsrc sekce v PE hlavičce, je soubor úspěšně detekován jako samorozbalovací archiv typu Zip.

5.7.4 Zip pomocí WinRAR

Zip archivy obecně mají signaturu „`PK..`“, hexadecimálné poté `0x50 0x4B 0x03 0x04`. Tuto skutečnost dodržuje i WinRAR při vytváření Zip samorozbalovacích archivů, kdy se signatura nachází na počátku **.rsrc** sekce.

Naleznutím Zip signatury na začátku sekce se předpokládá, že daný soubor je samorozbalovacím archivem Zip vytvořeným pomocí WinRAR. Avšak může nastat situace, kdy tento předpoklad bude špatný, neboť jiné programy mohou vytvořit archiv s obdoubnou strukturou, avšak jejich chování může být jiné.

5.8 Zhodnocení

Detekce typu archivu a extrakce hesel ze samorozbalovacích archivů je funkční pro každý takový archiv, který byl vygenerován v jednom z archivačních programů - WinRAR, 7z, WinZip Self-Extractor (oficiální a nejčastěji používané komprimační programy). Pro ostatní a méně používané programy, které taktéž dokáží vytvářet samorozbalovací archivy, by bylo nutné projít strukturu vzniklých archivů. Avšak s již získanou znalostí o způsobech uložení konfiguračních informací by pravděpodobně nebylo obtížné aplikovat tyto znalosti na jiné programy a jejich výstupní samorozbalovací archivy.

Jediný problém nastává u Zip archivů, kde se v konfiguračních informacích vyskytují znaky s diakritikou. Současná verze extraktoru tyto konfigurační informace nedokáže vyextrahovat, neboť způsob uložení takových informací neodpovídá základní kompresní metodě Deflate. Ta spolehlivě a z definice pracuje pouze se základními znaky v ASCII hodnotě od 0 do 255.

```
.data:00417000 ; Segment type: Pure data
.data:00417000 ; Segment permissions: Read/Write
.data:00417000 _data segment para public 'DATA' use32
.data:00417000 assume cs:_data
.data:00417000 ;org 417000h
.data:00417000 dd offset aBadAllocation ; "bad allocation"
.data:00417004 ; HANDLE hObject
.data:00417004 hObject dd 0FFFFFFFFh
.data:00417008 dword_417008 dd 0FFFFFFFFh
.data:0041700C dword_41700C dd 1
.data:00417010 aSfxhead_c db 'sfxhead.c',0
.data:0041701A align 4
.data:0041701C off_41701C dd offset aSfxhead_c ; "sfxhead.c"
.data:00417020 ; SIZE_T dword_417020
.data:00417020 dword_417020 dd 1F4h
.data:00417024 ; SIZE_T dword_417024
.data:00417024 dword_417024 dd 101h
.data:00417028 ; SIZE_T dword_417028
.data:00417028 dword_417028 dd 101h
.data:0041702C ; SIZE_T dword_41702C
.data:0041702C dword_41702C dd 204h
.data:00417030 ; SIZE_T uBytes
.data:00417030 uBytes dd 202h
.data:00417034 ; SIZE_T dwBytes
.data:00417034 dwBytes dd 41h
.data:00417038 ; SIZE_T dword_417038
.data:00417038 dword_417038 dd 200h
.data:0041703C ; WPARAM wParam
.data:0041703C wParam dd 1
.data:00417040 off_417040 dd offset unk_419A60
.data:00417044 unk_417044 db  4Eh ; N
.data:00417045 db  4Dh ; M
.data:00417046 db  43h ; C
.data:00417047 db    1
.data:00417048 db  73h ; s
.data:00417049 db  66h ; f
.data:0041704A db  78h ; x
.data:0041704B db    2
.data:0041704C dword_41704C dd 0B84D27F3h
.data:00417050 ; LONG dword_417050
.data:00417050 dword_417050 dd 2306Bh
.data:00417054 ; LONG lDistanceToMove
.data:00417054 lDistanceToMove dd 23000h
.data:00417058 ; LONG dword_417058
.data:00417058 dword_417058 dd 230EBh
.data:0041705C dword_41705C dd 0
.data:00417060 dword_417060 dd 10A02h
.data:00417064 db    0
.data:00417065 db    0
```

Obrázek 5.8: .data sekce Zip samorozbalovacího archivu vytvořená diasemblerem

Z hlediska bezpečnosti je uložení hesla k vnitřnímu archivu v konfiguračních informacích vnějšího archivu zcela nedostačují, neboť tyto informace nejsou šifrovány. Avšak šifrovány být nemohou, neboť šifrované konfigurační informace by poté nesplňovaly svůj účel - bez jakéhokoliv zásahu uživatele rozbalit obsah archivu.

5.8.1 Porovnání archivů

Zabezpečení běžných archivů (ne-samorozbalujících) se odvíjí především od použité šifry, která je aplikována na obsah archivu, neboť implementace struktur archivů neobsahují bezpečnostní díry (alespoň v tuto chvíli). I když ne vždy toto platí, především pak u Zip archivů využívajících šifru AES, kdy implementace kontroly aplikovaného hesla nevyužívá celou sílu šifry AES (popsáno v sekci 4.6.1).

Všechny archivní formáty dnes umožňují šifrovat soubory pomocí AES (především pomoci ní). Tento fakt má svůj důvod, neboť stále neexistuje spolehlivý útok na tuto šifru, který by byl výrazně rychlejší než klasický útok silou. Z tohoto pohledu jsou nejbezpečnějším archivním formátem formáty RAR a 7z, které využívají šifru AES-256. Formát Zip s šifrou AES-256 se mezi nejbezpečnější rozhodně považovat nemůže, především z důvodu nevyužití plného potencionálu AES při kontrole hesla (sekce 4.6.1).

Ostatní šifry lze považovat za slabší, v případě variant AES však za stále dostačující. Nejhůře je na tom z hlediska bezpečnosti archivní formát Zip za použití šifry PKWARE, která je v dnešní době považována za velmi slabou. Avšak velké rozšíření této kombinace způsobuje, že přežívá dál a hojně se používá.

Jednoduchým porovnáním lze jednotlivé archivní formáty a použité šifry znázornit pomocí tabulky 5.1.

Šifra \ **Formát archivu**	Zip	RAR	7z
AES-256	○	●	●
AES-192	–	–	–
AES-128	○	○	–
PKWARE	×	–	–

- ● neprolomitelné v rozumném čase
- ○ dostačující
- × nedostačující
- – neimplementováno

Tabulka 5.1: Porovnání archivů z hlediska bezpečnosti

Jiné druhy šifrování tyto archivy v současné době oficiálně nepodporují. Existují však různé rozšíření, které umožňují šifrovat dané archivy pomocí jiných šifer, především formát 7z umožňuje aplikaci kterékoliv šifry.

Ovšem jak již bylo zmíněno, za nejbezpečnější šifru je považován AES-256, kterou i americká vládní agentura NSA doporučuje pro šifrování Top Secret informací[20]. Tuto šifru lze tedy bez obav použít na ochranu archivů.

Kapitola 6

Závěr

V této práci byly popsány formáty nejpoužívanějších komprimačních programů, struktura spustitelných programů, dále šifry a kompresní metody, které se v těchto programech používají, a následně vylíčeno vzájemné propojení šifer a archivních formátů, včetně způsobu, jak se dané archivy pomocí šifer zabezpečují. Byly také rozebrány nejpoužívanější i specifické útoky na archivy či samotné šifry. Obecné útoky na archivy byly taktéž odsimulovány pro představu, jak dlouho trvá naleznout heslo k archivu pomocí „nejhorší“ metody.

Všechny zjištěné skutečnosti a mnohé další, které bylo nutné nastudovat a zjistit průběžně při prozkoumávání struktur samorozbalovacích archivů, posloužily k přesné identifikaci místa a způsobu uložení hesel pro nejčastější formáty samorozbalovacích archivů. Na základě veškerých zjištěných informací byl vytvořen extraktor, který pro tyto nejčastější formáty úspěšně hesla extrahoval. Konzultant z firmy AVG, pro kterou byl tento extraktor primárně vytvořen, jej přijal a ohodnotil bez výhrad.

Extraktor však pokrývá pouze nejčastější archivy RAR, Zip, a 7z, struktury méně častých archivů prozkoumány nebyly. U archivů Zip navíc není možné extrahovat hesla a celou konfigurační informaci, pokud obsahují znaky s diakritikou.

Právě extrakce hesel z archivů Zip zabrala nejvíce času a práce, které ovšem mohlo být výrazně méně, neboť zjištěná metoda Deflate pro uložení konfiguračních informací se od řádné metody Deflate liší pouze v kontrole úvodních drobností. Uložení informací pomocí metody Deflate však není nikde zdokumentováno a zůstává tak otázkou, zda-li se uložení informací s kontrolou úvodních drobností, oproti řádné metodě, nedalo zjistit dříve.

Každopádně toto zjištění rozdílu uložení informací pouze v drobnostech nebylo po takové době příjemné, a pokud by se tato skutečnost dala zjištit dříve, mohl být případný ušetřený čas využit například pro popis struktur méně častých archivačních formátů či identifikace způsobu uložení konfiguračních informací se znaky s diakritikou u Zip archivů.

Kromě extraktoru byly také popsány a porovnány jednotlivé komprimační formáty z hlediska bezpečnosti, kde jasnými favority v bezpečnosti jsou formáty RAR a 7z při použití šifry AES-256.

Literatura

[1] Aeschbache, N.: *ZIP-64 Internal Layout* [online]. [cit. 2012-12-30].
URL http://en.wikipedia.org/wiki/File:ZIP-64_Internal_Layout.svg

[2] Amstadt, B.: *wine* [online]. [cit. 2013-01-06].
URL http://www.winehq.org/

[3] Bellare, M.; Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Lecture Notes in Computer Science*, ročník 1976, 2000: s. 531–545.

[4] Biham, E.; Kocher, P. C.: A known plaintext attack on the PKZIP stream cipher. *Lecture Notes in Computer Science*, ročník 1008, 1995: s. 144–153.

[5] Bogdanov, A.; Khovratovich, D.; Rechberger, C.: *Biclique Cryptanalysis of the Full AES* [online]. [cit. 2013-01-05].
URL
http://research.microsoft.com/en-us/projects/cryptanalysis/aesbc.pdf

[6] CBS News: *The 25 most common passwords of 2012* [online]. [cit. 2012-11-08].
URL http://www.cbsnews.com/8301-205_162-57539366/
the-25-most-common-passwords-of-2012/

[7] Drábek, V.: *Kódování a komprese dat: KKO*. Brno: Fakulta informačních technologií, 2008, ISBN 978-0138690175, 236 s.

[8] ElcomSoft: *Advanced Archive Password Recovery* [online]. [cit. 2013-01-06].
URL http://www.elcomsoft.com/archpr.html

[9] Gladman, B.: *A Password Based File Encryption Utility* [online]. [cit. 2012-12-30].
URL http://www.gladman.me.uk/cryptography_technology/fileencrypt/

[10] Gwenda: *CTR encryption* [online]. [cit. 2013-01-01].
URL http://en.wikipedia.org/wiki/File:Ctr_encryption.png

[11] Hanáček, P.; Staudek, J.: *Bezpečnost informačních systémů*. Úřad pro státní informační systém, 2000, ISBN 80-238-5400-3, 127 s.

[12] Huffman, D. A.: A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the I.R.E*, ročník 1952, 1952: s. 1098–1102.

[13] Koch, W.: *crc32.c* [online]. [cit. 2013-01-02].
URL http://fossies.org/linux/misc/old/wedit-1.15.1.tar.gz:
a/wedit-1.15.1/wklib/crc32.c

[14] Kohn, M.: *kunzip library for decompressing Zip archives* [online]. [cit. 2013-05-07].
URL http://downloads.mikekohn.net/programs/kunzip-2006-11-14.tar.gz

[15] Kohno, T.: *Analysis of the WinZip encryption method* [online]. [cit. 2012-12-29].
URL eprint.iacr.org/2004/078.pdf

[16] Kuliukas, K.: *How Rainbow Tables work* [online]. [cit. 2012-11-08].
URL http://kestas.kuliukas.com/RainbowTables/

[17] Lucas, I.: *Password Recovery Speeds* [online]. [cit. 2013-01-04].
URL http://www.lockdown.co.uk/?pg=combi&s=articles

[18] Microsoft: *A Tour of the Win32 Portable Executable File Format* [online]. [cit. 2013-05-07].
URL http://msdn.microsoft.com/en-us/library/ms809762.aspx

[19] National Institute of Standards and Technology: *SECURE HASH STANDARD* [online]. [cit. 2012-12-30].
URL http://www.itl.nist.gov/fipspubs/fip180-1.htm

[20] National Security Agency: *NSA Suite B Cryptography - NSA/CSS* [online]. [cit. 2013-05-13].
URL http://www.nsa.gov/ia/programs/suiteb_cryptography/

[21] NIST: *FIPS 197, Advanced Encryption Standard (AES) - NIST* [online]. [cit. 2012-11-18].
URL http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[22] NIST: *Recommendation for Block Cipher Modes of Operation* [online]. [cit. 2012-11-18].
URL http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

[23] NIST: *DATA ENCRYPTION STANDARD (DES)* [online]. [cit. 2013-01-07].
URL http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf

[24] Pavlov, I.: *7z source codes* [online]. [cit. 2013-01-07].
URL http://downloads.sourceforge.net/sevenzip/7z920.tar.bz2

[25] PKWARE: *PKWARE appnote* [online]. [cit. 2013-01-02].
URL http://www.pkware.com/documents/casestudies/APPNOTE.TXT

[26] Project, T. B. D.: *Boomerang Decompiler* [online]. [cit. 2013-05-06].
URL http://boomerang.sourceforge.net/

[27] RARLAB: *unRAR download* [online]. [cit. 2012-11-03].
URL http://www.rarlab.com/rar/rarlinux-4.2.0.tar.gz

[28] Rijmen, V.: *Practical-Titled Attack on AES-128 Using Chosen-Text Relations* [online]. [cit. 2013-01-05].
URL http://www.cosic.esat.kuleuven.be/publications/article-1475.pdf

[29] SA., H.-R.: *IDA Support: Evaluation Version* [online]. [cit. 2013-05-06].
URL http://www.hex-rays.com/idapro/idadownfreeware.htm

[30] Scherbakov, O.: *General information about the configuration file* [online]. [cit. 2013-05-04].
URL http://7zsfx.info/en/configinfo.html

[31] Schneier, B.: *The Blowfish Encryption Algorithm* [online]. [cit. 2013-01-07].
URL http://www.schneier.com/blowfish.html

[32] Schneier, B.; Kelsey, J.: *Twofish* [online]. [cit. 2013-01-07].
URL http://www.schneier.com/twofish.html

[33] Stallings, W.: *Cryptography and Network Security: Principles and Practice (2nd Edition)*. Prentice Hall, 1998, ISBN 978-0138690175, 569 s.

[34] Stay, M.: *ZIP Attacks with Reduced Known Plaintext* [online]. [cit. 2012-11-07].
URL math.ucr.edu/~mike/zipattacks.pdf

[35] The Internet Society: *RFC 2898* [online]. [cit. 2012-11-18].
URL http://www.faqs.org/rfcs/rfc2898.html

[36] The Internet Society: *RFC 2104* [online]. [cit. 2012-12-29].
URL http://tools.ietf.org/html/rfc2104

[37] The Internet Society: *RFC 1951* [online]. [cit. 2013-05-07].
URL http://www.ietf.org/rfc/rfc1951.txt

[38] TheIACR: *Biclique cryptanalysis of the full AES* [online]. [cit. 2013-01-05].
URL http://www.youtube.com/watch?v=IlfBiNqHIgU

[39] Wagner, E.: *java-unrar* [online]. [cit. 2013-01-02].
URL http://code.google.com/p/java-unrar/source/browse/trunk/src/de/innosystec/

[40] Wagner, N. R.: *The Laws of Cryptography: Advanced Encryption Standard: S-Boxes* [online]. [cit. 2012-11-18].
URL http://www.cs.utsa.edu/~wagner/laws/SBoxes.html

[41] Wikipedia: *HMAC-SHA1 Generation* [online]. [cit. 2012-12-30].
URL http://en.wikipedia.org/wiki/Hash-based_message_authentication_code

[42] WinZip Computing: *WinZip - AES Encryption Information* [online]. [cit. 2012-11-18].
URL http://www.winzip.com/aes_info.htm

[43] WinZip Computing: *About encryption and encryption methods* [online]. [cit. 2012-12-29].
URL http://kb.winzip.com/help/winzip/help_encryption.htm

[44] Ziv, J.; Lempel, A.: *Adaptive data compression system with systolic string matching logic* [online]. [cit. 2013-05-07].
URL http://www.patentstorm.us/patents/5532693/description.html

Příloha A

Formáty archivů

A.1 ZIP

Offset	Byte	Položka	Popis
0	4	Local file header signature = 0x04034b50	Signatura
4	2	Minimum version needed to extract	Verze pro extrakci
6	2	General purpose bit flag	Archiv má normální strukturu
8	2	Compression method	Použitá kompresní metoda
10	2	File last modification time	Čas modifikace
12	2	File last modification date	Datum modifikace
14	4	CRC–32	CRC součet původního souboru
18	4	Compressed size (s)	Velikost komprimovaného souboru
22	4	Uncompressed size	Velikost nekomprimovaného souboru
26	2	File name length (n)	Velikost názvu souboru
28	2	Extra field length (m)	Velikost Extra pole
30	n	File name	Název souboru
30+n	m	Extra field	Extra pole
30+n+m	s	Compressed data	Komprimovaná data

Tabulka A.1: Local header

Offset	Byte	Položka	Popis
0	4	Extended Local file header signature (0x08074b50)	Signatura
4	4	CRC–32	CRC součet původního souboru
8	4	Compressed size	Velikost komprimovaného souboru
12	4	Uncompressed size	Velikost nekomprimovaného souboru

Tabulka A.2: Extended local header

Offset	Byte	Položka	Popis
0	4	Central directory file header signature (0x02014b50)	Signatura
4	2	Version made by	Operačního systém, na kterém byl archiv vytvořen
6	2	Minimum version needed to extract	Verze pro extrakci
8	2	General purpose bit flag	Archiv má normální strukturu
10	2	Compression method	Použitá kompresní metoda
12	2	File last modification time	Čas modifikace
14	2	File last modification date	Datum modifikace
16	4	CRC-32	CRC součet původního souboru
20	4	Compressed size	Velikost komprimovaného souboru
24	4	Uncompressed size	Velikost nekomprimovaného souboru
28	2	File name length (n)	Velikost názvu souboru
30	2	Extra field length (m)	Velikost Extra pole
32	2	File comment length (k)	Velikost komentářů
34	2	Disk number where file starts	Číslo disku se souborem (pro více archivů)
36	2	Internal file attributes	Atributy
38	4	External file attributes	Atributy
42	4	Relative offset	Offset na Local header
46	n	File name	Název souboru
46+n	m	Extra field	Extra pole
46+n+m	k	File comment	Komentář k souboru

Tabulka A.3: Central directory header

Offset	Byte	Položka	Popis
0	4	End of central directory signature (0x06054b50)	Signatura
4	2	Number of this disk	Číslo disku
6	2	Disk where central directory starts	Číslo disku počátku central directory
8	2	Number of central directory records on this disk	Počet záznamů na disku
10	2	Total number of central directory records	Počet záznamů
12	4	Size of central directory (bytes)	Velikost
16	4	Offset of start of central directory relative to start of archive	Offset
20	2	Comment length (n)	Velikost komentáře
22	n	Comment	Komentář

Tabulka A.4: Ukončení Central directory záznamu

A.2 RAR

Offset	Byte	Typ	Popis
0	2	`header_crc`	CRC součet celého bloku
2	1	`header_type`	Typ bloku
3	2	`header_flags`	Příznak bloku
5	2	`header_size`	Velikost bloku (m)
7	m	Další položky závisející na `header_type`	

Tabulka A.5: Volume header

Hodnota	Typ	Popis
0x72	`MARK_HEAD`	Značka identifikující RAR soubor
0x73	`MAIN_HEAD`	Hlavička archivu
0x74	`FILE_HEAD`	Hlavička souboru
0x75	`COMM_HEAD`	Hlavička komentáře (pro starší verze než 3.20)
0x76	`AV_HEAD`	Staré autentizační informace (pro starší verze než 3.20)
0x77	`SUB_HEAD`	Sub–blok (pro starší verze než 3.20)
0x78	`PROTECT_HEAD`	Záznam pro zotavení (pro starší verze než 3.20)
0x79	`SIGN_HEAD`	Autentizační informace (pro starší verze než 4.00)
0x7a	`NEWSUB_HEAD`	Autentizační informace

Tabulka A.6: `header_type`

Hodnota	Popis
0x0001	Atributy svazku
0x0002	Přítomnost komentáře k archivu
0x0004	Atributy uzamčeného archivu
0x0008	Kompresní metoda Solid
0x0010	Název nového svazku („volname.partN.rar“)
0x0020	Přítomnost autentizační informace
0x0040	Přítomnost zotavovacího záznamu
0x0080	Hlavičky bloků jsou šifrované
0x0100	První svazek (3.0 a pozdější)
...	Další příznaky pro interní použití

Tabulka A.7: `header_flags` pro `MAIN_HEAD`

Hodnota	Popis
0x01	Soubor navazuje na soubor z předchozího svazku
0x02	Soubor navazuje na soubor v následujícím svazku
0x04	Soubor zašifrován pomocí hesla
0x08	Přítomnost komentáře k souboru
0x10	Jsou použity informace z předchozích souobrů (Solid komprese**??**)
0x20	Kombinace bitů 0x80 0x40 0x20 udává velikost slovníku:
0x40	0 0 0 - 64 KB, 0 0 1 - 128 KB, 0 1 0 - 256 KB, 0 1 1 - 512 KB
0x80	1 0 0 - 1024 KB, 1 0 1 - 2048 KB, 1 1 0 - 4096 KB, 1 1 1 - soubor je složka
0x100	Přítomnost `HIGH_PACK_SIZE` a `HIGH_UNP_SIZE` polí,
	která jsou použita u souborů větších jak 2 Gb
0x200	`FILE_NAME` obsahuje normální i Unicode jméno souboru
0x400	Hlavička obsahuje 8 bytů navíc pro Salt
0x800	Verze souboru
0x1000	Přítomnost Extended time polí
0x8000	Vždy nastaveno na 1

Tabulka A.8: `header_flags` pro `FILE_HEAD`

Název	Popis
`PACK_SIZE`	Velikost komprimovaného souboru
`UNP_SIZE`	Velikost nekomprimovaného souboru
`HOST_OS`	Použitý operační systém při provádění archivace:
	0 - MS DOS, 1 - OS/2, 2 - Windows
	3 - Unix, 4 - Mac OS, 5 - BeOS
`FILE_CRC`	CRC součet nekomprimovaného souboru
`FTIME`	Datum a čas archivace, standardní MS-DOS formát
`UNP_VER`	Minimální verze RAR potřebná k extrakci:
	10 * Major verze + minor verze
`METHOD`	Použitá kompresní metoda
	0x30 - uložení bez komprese
	0x31 - nejrychlejší komprese
	0x32 - rychlá komprese
	0x33 - normální komprese
	0x34 - dobrá komprese
	0x35 - nejlepší komprese
`NAME_SIZE`	Velikost názvu souboru
`ATTR`	Atributy souboru
`HIGH_PACK_SIZE`	Horní 4 byte z 64 bit velikosti komprimovaného souboru
`HIGH_UNP_SIZE`	Horní 4 byte z 64 bit velikosti nekomprimovaného souboru
`FILE_NAME`	Název souboru o velikosti `NAME_SIZE`
`SALT`	*Salt* pro (de)šifrování
`EXT_TIME`	Extended time

Tabulka A.9: Další položky pro `FILE_HEAD`

Příloha B

Zdrojové a pseudokódy

B.1 AES Key Expansion

Pseodokód převzatý z [21].
`SubWord()` - na vstup aplikuje S-Box (sekce 3.3),
`RotWord()` - byte vstupu posune doleva - z $[a_0, a_1, a_2, a_3]$ bude $[a_1, a_2, a_3, a_0]$,
`Rcon` - kruhová konstanta.

```
KeyExpansion (byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
    word temp

    i = 0

    while (i < Nk)
        w[i] = word (key[4*i], key[4*i + 1], key[4*i + 2], key[4*i + 3]
        i = i + 1
    end while

    i = Nk

    while (i < Nb*(Nr+1))
        temp = w[i-1]
        if (i mod Nk = 0)
            temp = SubWord(RotWord(temp)) XOR Rcon[i/Nk]
        else if (Nk > 6 and i mod Nk = 4)
            temp = SubWord(temp)
        end if
        w[i] = w[i-Nk] XOR temp
        i = i + 1
    end while
end
```