SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
Fakulta elektrotechniky a informatiky

Evidenčné číslo: FEI-12307-30190

Analýza operačnej pamäte s cieľom odhalenia neznámeho kódu

Dizertačná práca

2014 Ing. Štefan Balogh

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE

Fakulta elektrotechniky a informatiky

Evidenčné číslo: FEI-12307-30190

Analýza operačnej pamäte s cieľom odhalenia neznámeho kódu

Dizertačná práca

Študijný program: Aplikovaná informatika
Číslo študijného odboru: 2511
Názov študijného odboru: 9.2.9 aplikovaná informatika
Školiace pracovisko: Ústav informatiky a matematiky
Školiteľ: doc. Ing. Pavol Zajac, PhD.
Konzultant: doc. RNDr. Jaroslav Fogel. PhD.

Bratislava 2014 **Ing. Štefan Balogh**

Poďakovanie

Týmto by som rád poďakoval môjmu vedúcemu práce doc. Ing. Pavlovi Zajacovi, PhD. za cenné rady a poznámky k práci a tiež mojej rodine za jej podporu počas celého obdobia realizácie práce. Zvláštne poďakovanie patrí mojej manželke za trpezlivosť a pomoc počas písania práce a za záverečnú korekciu a kontrolu práce.

Zhrnutie

Práca sa zaoberá možnosťou sledovania obsahu pamäte a využitia získaných informácií k detekcii malware. K splneniu tohto cieľa bolo potrebné vyvinúť vhodnú techniku na čítanie obsahu pamäte a vybrať vhodné informácie z pamäte pre detekciu malware.

V prvej časti práce sme vyvinuli vlastné riešenie pre prístup k obsahu pamäte, ktoré je použiteľné aj pre detekciu malware.

V druhej časti práce sme ako metódu detekcie, pre ktorú dané informácie chceme použiť, vybrali heuristickú detekciu, ktorá využíva metódy strojového učenia a datamining algoritmy. Vhodnosť vybraných informácií pre detekciu sme testovali pomocou techník pre výber optimálnych vlastností pre datamining algoritmy. Výsledky úspešnosti datamining klasifikátorov ukazujú, že vybrané informácie sú pomerne významné pre klasifikáciu a môžu byť využité v automatizovaných detekčných systémoch.

Abstrakt

The aim of this work is to develop new methods for detecting malware from data acquired from the memory of the computer. To meet this objective, it was necessary to develop a suitable technique for reading the memory contents, and to select appropriate data from the memory usable for malware detection.
In the first part, we developed our own custom solution to access the appropriate contents of the memory resistant against rootkit techniques.
In the second part, we focus of using the acquired data for malware detection by heuristics based on machine learning and data mining algorithms. Feature selection was used to test suitability of specific in-memory system data structures to serve as malware indicators. The results of data-mining indicate that analysed features can provide significant detection rates to be used in automatic malware detection software.

Obsah

Slovník pojmov

Command and Control Server - "Command and Control" (C&C) servery sú centralizované počítače schopné posielať príkazy a prijímať spätné dáta z počítačov, ktoré sú infikované bot malware, a tým sú súčasťou botnet siete. Botnet C&C servery obyčajne používajú jednu zo štyroch štruktúr - hviezda, multi server, hierarchická štruktúra a náhodná (pozn. každá má svoje výhody a nevýhody, viac na http://security.radware.com/knowledge-center/DDoSPedia/command-and-control-server/).

DMA - Direct Memory Access je technológia, ktorá zabezpečuje zariadeniam priamy prístup do pamäte bez nutnosti riadenia operácie prenosu dát procesorom. Procesor zabezpečí iba zahájenie a potom ukončenie prenosu.

Hook - presmerovanie volania požadovanej funkcie na útočníkom podsunutú funkciu, ktorá vykoná škodlivú aktivitu a aby sa presmerovanie neprezradilo, zavolá sa pôvodná funkcia. Podľa úrovne kde sa hook vykoná, môžeme mať hooknuté objekty v užívateľskom priestore a v priestore jadra.

MMU - Memory Managment Unit je hardvérová časť, ktorá robí preklad virtuálnej adresy na fyzickú (okrem iného). Využíva ju procesor.

NDIS (The Network Driver Interface Specification) - špecifikácia pre komunikáciu s protokolovými ovládačmi (protocol drivers) a aplikačné programové rozhranie (API) pre sieťové karty (NICs). Bol vyvinutý spoločne firmami Microsoft a 3Com Corporation a používa sa hlavne v operačnom systéme Microsoft Windows

PAE - Physical Address Extension je rozšírenie adresného priestoru pre operačné systémy x86. Keď sa 32-bitov podporujúcich 4GB operačnej pamäte stalo obmedzením, bolo rozšírené stránkovanie adresy na trojúrovňové a podporu tak rozšírilo na 64GB.

PE súbor (portable executable) – formát pre ukladanie binárnych spustiteľných súborov v operačnom systéme MS Windows. Do tejto kategórie spadajú okrem exe súborov aj ovládače a dynamické knižnice DLL.

Proof of concept - krátka alebo nekompletná realizácia určitej metódy, myšlienky, ktorá má demonštrovať, preveriť očakávané hodnoty metódy alebo myšlienky.

Register EIP - register procesora, v ktorom sa nachádza adresa práve vykonávanej inštrukcie procesorom.

Register CR2 (CR3) - kontrolné registre (CR) procesora majú za úlohu meniť alebo kontrolovať základné správanie procesora a ostatných zariadení. Medzi hlavné úlohy patrí kontrola prerušení, prechod z užívateľského adresného módu do módu jadra, kontrola procesu stránkovania pamäte atď. CR2 register obsahuje hodnotu

nazývanú Page Fault Linear Address (PFLA), čo je adresa programu, ktorá vyvolá page fault výnimku. Cr3 sa využíva pri zapnutí stránkovania (paging), to je, keďsa nastaví PG bit v registri CR0. CR3 register v horných 20 bitoch uchováva fyzickú adresu prvého adresára stránok pre daný proces.

Relatívna virtuálna adresa (RVA - relative virtual address) - virtuálna adresa objektu zo súboru, ktorý sa načíta do pamäte, od ktorej je odčítaná základná (base) adresa začiatku súboru. Kedykoľvek sa z nej dá urobiť platný ukazovateľ na pamäťovú lokáciu pripočítaním aktuálnej základnej (ImageBase) adresy a umožňuje flexibilne reagovať na zmeny ImageBase.

SSDT tabuľka (System Service Dispatch Table) - jedna zo základných systémových tabuliek používaná v OS Microsoft Windows. V podstate ide o tabuľku pointerov na API funkcie súvisiace so systémovými volaniami registrovanými operačným systémom.

Tabuľky importov (IAT – Import Address Table) - každý PE súbor obsahuje vlastnú tabuľku importov IAT, kde sú uložené informácie (adresy) API funkcií a funkcií z DLL knižníc, ktoré program pri svojom behu používa.

Tabuľky exportov (EAT- Export Address Table) - každý PE súbor obsahuje aj vlastnú tabuľku exportov EAT, kde sú uložené informácie (adresy) funkcií, ktoré daný súbor, DLL knižnica alebo ovládač exportujú pre ostatné programy. Viac funkcií spravidla exportujú DLL knižnice.

Tabuľka vektorov prerušenia (IDT - Interrupt Descriptor Table) - dátová štruktúra, v ktorej sa na architektúre x86 uchovávajú vektory prerušenia.

Vstupné /výstupné pakety (IRP - I/O request packets) - štruktúry jadra, ktoré sa používajú pri komunikácii medzi modelom Windows ovládačov (WDM - Windows Driver Model) a Windows NT ovládačmi zariadení (Windows NT device drivers) navzájom a s operačným systémom.

Windows loader - systémový program, ktorý načíta exe súbor do pamäte, vykoná potrebné úkony na jeho spustenie a spustí ho.

Záznam adresárovej tabuľky (PDE The page directory entry) - záznam v adresári stránok obsahuje fyzické bázové adresy stránkovacích tabuliek, prístupové práva a iné atribúty.

Záznam stránkovacej tabuľky (PTE The page table entry) - záznam v stránkovacej tabuľke operačného systému, ktorý mapuje virtuálne adresy na fyzické adresy. Záznam tabuľky obsahuje informácie o tom, ktoré rámce (stránky) pamäte sú mapované. Obsahuje tiež pomocné informácie o rámcoch ako present bit, dirty alebo modified bit, adresu miesta alebo ID procesu.

Úvod

V odborných kruhoch zaoberajúcich sa problematikou bezpečnosti je jednou zo stále diskutovaných otázok hľadanie účinných metód pre detekciu škodlivého kódu. Pojmom škodlivý kód (ďalej malware z anglického malicious software) označujeme ľubovoľný program, ktorý je vytvorený so škodlivým zámerom. V minulosti malware vyvíjali autori predovšetkým preto, aby sa zviditeľnili ich programy, ale zlomyseľný zámer neobsahovali. S príchodom internetu sa však zmenila aj podstata malware. Pomocou internet bankingu môžeme vykonávať rôzne finančné transakcie, nakupovať veci v elektronických obchodoch, či podávať rôzne dokumenty na úrady. A to sa snažia autori moderného malware využiť, napríklad na odcudzenie osobných údajov, ktoré na dané úkony využívame. Tento fakt prinútil ľudí hľadať účinné metódy, ako sa proti malware brániť. Vývoj malware však napreduje neuveriteľnou rýchlosťou a útočníci sú často o krok vpred.

V roku 1992 bol počet počítačových vírusov odhadnutý na 1 000 až 2 300. V roku 2002 už bolo 60 000 známych vírusov, trójskych koní, červov a ich variácií (Daoud et al., 2008). Podľa Symantec Global internet Security threat report za rok 2008 bolo vytvorených 1 656 227 nových signatúr pre identifikáciu škodlivého kódu. To je 265% nárast oproti roku 2007, keď bolo vytvorených 624 267 nových signatúr škodlivého kódu. A teda Symantec do konca roka 2008 evidovala celkovo 2 674 171 signatúr a z toho viac než 60% bolo vytvorených v roku 2008 (Symantec, 2009). Ešte alarmujúcejšia je výsledná správa od Symantec za ďalší rok. V roku 2009 bolo vytvorených až 2 895 802 nových signatúr pre identifikáciu škodlivého kódu . Takže počet všetkých evidovaných signatúr do konca roku 2009 vzrástol na 5 724 106. To znamená, že zo všetkých signatúr, ktoré Symantec evidoval za dané obdobie, bolo 51% vytvorených v roku 2009 (Symantec, 2010). Za rok 2010 to už bolo 286 miliónov nových hrozieb (Symantec, 2011) a v roku 2011 bolo vytvorených až 400 miliónov nových variant malware (Symantec, 2012).

Detekcia malware na základe signatúr je v súčasnosti stále jedna z najrozšírenejších metód detekcie. Jej nevýhoda však je, že môže detegovať iba malware, ktoré má v databáze signatúr. Manuálna analýza malware a vytváranie signatúr pri súčasnom trende vývoja a možnostiach modifikácie malware sa však stáva neúnosná. Na to, aby sa predišlo väčším škodám, vznikla potreba odhaliť nové druhy malware v čo najkratšom možnom čase. A práve v tom majú analytikom pomôcť automatizované nástroje, ktoré sa snažia podať čo najpresnejšiu správu o činnosti analyzovaného programu. Všeobecne platí, že tieto prístupy môžeme zadeliť do dvoch hlavných kategórií: statické analýzy a dynamické analýzy. Statická analýza využíva informácie v podozrivých spustiteľných programoch bez toho, aby boli spustené, na základe kontroly binárneho kódu a čítania priamo v assembleri. Tieto prístupy sú sľubné, ale

ukazuje sa, že disassemblovanie spustiteľného kódu môže byť ťažký problém. Približne u 90% vírusov nie je možné plne disassemblovať binárny kód a zachytiť tak ich rutinné správanie (Dai et al., 2009). Dynamická analýza sleduje činnosť programu po načítaní do pamäte. Dynamický prístup sa teda zameriava viac na aktivity, ale je ťažké vopred zabezpečiť, aby sa predišlo poškodeniu, pretože vírus, ktorý testujeme, je "spustený".

Výhodou týchto prístupov oproti bežne zaužívaným metódam je schopnosť odhaľovania vírusov, ktoré sa ešte nenachádzajú vo vírusovej databáze, a teda možnosť odhaliť doposiaľ neidentifikované vírusy a ich derivácie.

Malware sa snaží preto hľadať stále nové prístupy pre schovávanie svojej činnosti a prítomnosti v systéme. Stále viac sofistikované techniky využívajú hlavne rootkity. Rootkity predstavujú samostatnú oblasť na poli malware, ktorých cieľom je schovanie svojej činnosti a činnosti iného malware a ktoré využívajú značne pokročilé techniky.

Jednou z používaných techník je aj aktivovanie, existencia a schovávanie sa škodlivého kódu iba v operačnej pamäti počítača. Čím ďalej, tým viac sa takéto techniky schovávania začínajú používať v rôznych modifikáciach. Výhodou týchto malware, ktoré existujú iba v pamäti počítača, je, že sa veľmi ťažko zisťujú a ich činnosť väčšinou bežnému užívateľovi nie je zviditeľnená. Mnohokrát užívateľ ani netuší o existencii malware.

Malú efektivitu súčasných techník na odhaľovanie malware si uvedomujú aj mnohé výskumné laboratóriá a z tohto dôvodu sa snažia zamerať výskum na hľadanie účinnejších techník. Problematika detekcie škodlivého kódu sa stala jednou z najviac vyvíjajúcich sa oblastí vo svete. Z tohto dôvodu je vlastný výskum v oblasti značne náročný a mnohokrát sa na niektorý problém nájde riešenie vo svete skôr, ako vytvoríme a dostatočne overíme vlastný model riešenia. Z jednej strany je to dobré, že výskum napreduje, z druhej strany vnímať neefektivitu a zdvojovanie výskumu. V mnohých publikovaných prácach sa k detekcii využívajú metódy dataminingu, čo sa na základe prezentovaných výsledkov detekcie javí ako sľubná metóda. Úroveň detekcie malware sa však značne mení, čo je zrejme zapríčinené výberom rôznych vlastností statických aj dynamických, pomocou ktorých identifikujeme rozdiely medzi neškodným programom a malware. Dané práce sa zaoberajú len bežnými formami malware a neriešia otázku malware rezistentných iba v operačnej pamäti.

Z článku, ktorý zahŕňa doterajšie výsledky prác v oblasti využitia datamining techník (Bazrafshan et al., 2013) vyplýva, že výsledky z veľkej miery závisia na nájdení vhodných vlastností, ktoré sa pri klasifikácii použijú. Bolo vidieť, že prístupy sú rôzne, ale všetky platia len pre určitú skupinu malware a nie je možné ich zovšeobecňovať (aj keď niektoré výsledky sú sľubné). Mnohé testy boli úspešné preto, lebo boli navrhnuté na základe známych vlastností vírusov. Ak sa v budúcnosti tvorcovia malware zamerajú na zmenu týchto vlastností, účinnosť

detekcie zrejme môže značne poklesnúť. Ako riešenie by sa javilo nájsť také vlastnosti, ktoré by mohli popisovať čo najpresnejšie rozdiely medzi škodlivým a neškodným programom a zároveň boli ťažko (ak sa má funkcionalita zachovať) zmeniteľné. Medzi také môžeme zaradiť aj vlastnosti popisujúce správanie. Ako zaujímavým riešením z tohto hľadiska by mohol byť návrh vlastností, ktoré môžeme extrahovať priamo z operačnej pamäte systému, a tým zjednodušiť proces získavania a spracovania potrebných údajov pre extrakciu vlastností (čo sa v súčasnosti stáva čoraz zložitejšie z dôvodu existencie rôznych techník ochrany kódu proti disassemblovaniu, šifrovaniu programov a jeho ukrývaniu iba v pamäti OS).

Cieľom tejto práce je hľadanie nových vlastností, ktoré by sa získavali analýzou pamäte OS a dosahovali by čo najlepšiu úroveň detekcie malware. Ďalej otestovať dané vlastnosti pre vybranú metódu strojového učenia. Táto téza zahŕňa nasledovné úlohy:

- návrh a analýza možných vlastností extrahovaných priamo z pamäte operačného systému, ktoré by zabezpečovali čo najlepšiu úroveň detekcie škodlivého kódu prítomného v pamäti OS,
- návrh a implementácia metódy extrakcie vybraných vlastností (pre vlastnosti získavané priamo z pamäte OS),
- návrh modelu pre testovanie a jeho aplikácia,
- testovanie na štatisticky dôveryhodnej vzorke a natrénovanie klasifikátorov na základe testovaných vlastností,
- sledovanie úspešnosti odhalenia škodlivého kódu.

S prvou úlohou hlavnej tézy súvisí nájdenie spôsobu pre efektívny prístup do pamäte a analyzovania získaných dát, ktoré by bolo možné spracovať v reálnom čase.

Celkovo je práca rozvrhnutá nasledujúcim spôsobom. Je členená do jednotlivých kapitol, ktoré tvoria ucelený pohľad na oblasť, ktorej sa v rámci práce venujeme. V prvej kapitole predstavíme základný koncept pamäte. V druhej kapitole sa venujeme možným spôsobom prístupu k obsahu pamäte. Po prehľade existujúcich prístupov diskutujeme ich obmedzenia z pohľadu ich použiteľnosti pre detekciu škodlivého kódu. Zároveň predstavíme v samostatnej sekcii naše vlastné riešenie nástroja pre čitanie obsahu pamäte a možnosti tohto nástroja. V ďalšej časti kapitoly môžeme nájsť stručný popis nástroja pre detekciu prístupu do vybranej oblasti pamäte, čo nám dáva možnosť jeho využitia k zabezpečeniu ochrany nášho nástroja pre čitanie obsahu pamäte a iných dôležitých systémových komponentov, ktoré sa nachádzajú v adresnom priestore pamäte jadra systému. V tretej kapitole popíšeme techniky malware, s ktorými sa v súčasnosti stretávame. Nasledujúca kapitola je

venovaná známym metódam detekcie malware. Po týchto dvoch kapitolách, kde máme možnosť oboznámiť sa so stavom v oblasti tvorby a detekcie malware, prejdeme na popis postupov pre analýzu pamäte. Ide o značne rozsiahlu oblasť, preto sa zameriame iba na základné postupy a viac pojednáme o možnostiach analýzy, ktoré môžu byť prínosom pre detekciu rôznych vektorov útoku. Po piatej kapitole máme potrebné znalosti k splneniu téz dizertačnej práce. V šiestej kapitole teda predstavíme hypotézu a metodiku, vyberieme vlastnosti pre algoritmy strojového učenia a otestujeme vhodnosť vybraných vlastností pre detekciu malware. Zároveň zhrnieme zistené výsledky a diskutujeme o nich. V závere predstavíme výsledok celej práce a zhodnotíme úspešnosť splnenia cieľov práce.

V prílohách práce sa nachádzajú výpisy techník malware (príloha A), výpis názvov mutexov (príloha B) a niektorých dôležitých výsledkov z testovania jednotlivých vlastností (príloha C).

1 Organizácia a činnosť pamäte

Kapitola nám pomôže pochopiť, ako je pamäť organizovaná, ako je zabezpečený jej chod a aké technológie pri jej práci sú používané. Ak chceme definovať pamäť, musíme si uvedomiť, že ide o jednu z najdôležitejších súčastí celého systému. Už v prvotnej koncepcii počítača, ktorý navrhol Von Neumann (Goldstine, 1972) sa s pamäťou počítalo ako so základným komponentom. CPU pri svojej práci načítava všetky informácie z pamäte, ktoré sa tam pred ich použitím procesorom musia nahrať. Tým sa pre nás pamäť stáva veľmi cenným zdrojom informácií. Samozrejme iba v prípade, ak sme schopní tieto informácie získať a analyzovať. K tomu je potrebné hlbšie rozumieť štruktúre a organizácii pamäte. V ďalšej časti nájdeme jej popis so zameraním na časti potrebné pre analýzu a vonkajší prístup k jej obsahu.

1.1 Pamäť počítača

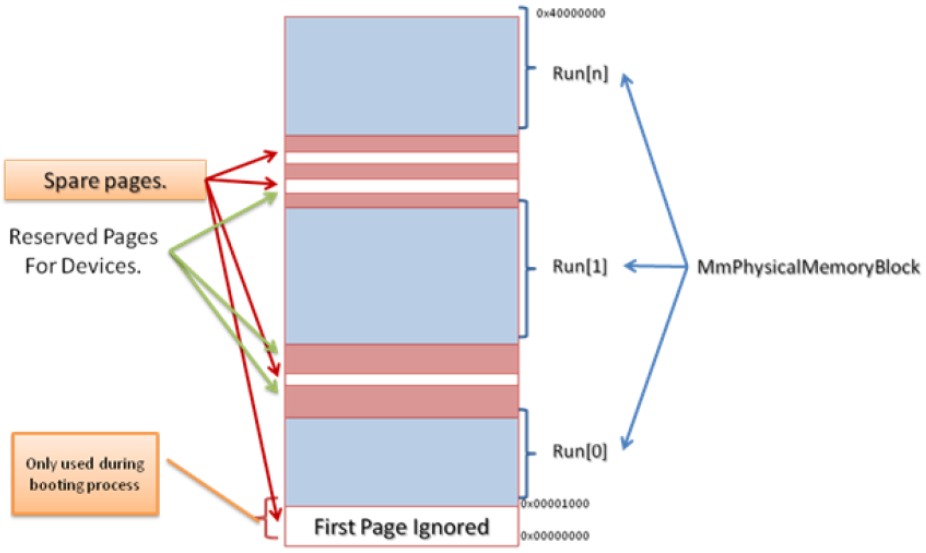
Pamäť s priamym prístupom (RAM Random Access Memory) je typ elektronickej pamäte, ktorá umožňuje prístup k ľubovoľnej oblasti v konštantnom čase bez ohľadu na jej fyzické umiestnenie (na rozdiel od sekvenčných pamätí, ako je napríklad pevný disk). V súčasnosti sa ako RAM používa hlavne pamäť, ktorá po výpadku napájania stratí svoj obsah. Tieto pamäte sa používajú v moderných počítačoch pre svoju rýchlosť, vďaka ktorej dokážu procesory pristupovať k dátam prakticky okamžite.
Komunikácia medzi systémom a pamäťou je pri štarte počítača v tzv. reálnom móde. Reálny mód alebo režim reálnych adries bol základným pracovným režimom procesorov z rodiny x86 . V súčasnosti sa reálny mód používa z dôvodu spätnej kompatibility so staršími systémami.
V režime reálnych adries pracuje v súčasnosti napríklad BIOS a pracovali v ňom operačné systémy typu DOS (MS-DOS, DR-DOS) a tiež prvé verzie Microsoft Windows (Windows 3.0 už pracoval v chránenom režime). Všetky moderné operačné systémy bežia v reálnom móde iba krátky čas pri štarte počítača, kým sa neprepne procesor do chráneného režimu. V reálnom móde je efektívny adresný priestor rovnaký ako lineárny alebo fyzický adresný priestor. Pamäť má nastavený model so segmentáciou[1] a používa 20-bitový adresný priestor (môže adresovať najviac 2^{20} bitov, teda maximálne 1 MByte). Adresa je v reálnom móde určená dvoma registrami, segmentovým a offsetovým. Adresa vznikne ako súčet hodnoty v segmentovom registry vynásobený 16-timi (posun o 4 bity doľava) a hodnoty v offsetovom registry. Dôležité je ešte spomenúť, že v reálnom móde má každý program a aplikácia plný prístup ku všetkým hardvérovým zariadeniam.

[1] Segmentácia je mechanizmus ochrany pamäťového priestoru, ktorý sa už prestal používať. Na architektúre IA32 sa považuje už skôr za dedičstvo, v architektúre x86-64 ho už celkom úplne nahradilo stránkovanie.

V chránenom móde je prístup aplikácií k zariadeniam kontrolovaný operačným systémom. Chránený mód používa model stránkovania pamäte, ktorý bude popísaný v ďalšej časti.
Moderné architektúry taktiež rozlišujú medzi fyzickou a virtuálnou pamäťou. Obyčajne má systém viac virtuálnej pamäte ako fyzickej. Napriek tomu, že fyzická pamäť by bola napr. iba 256 MB veľká, virtuálna pamäť môže byť adresovaná pri 32-bitovej architektúre až do 2^{32} bytov, čo je 4GB. Pri 64-bitovej architektúre môže byť veľkosť virtuálnej pamäte až 2^{64} bytov.

1.1.1 Organizácia pamäte

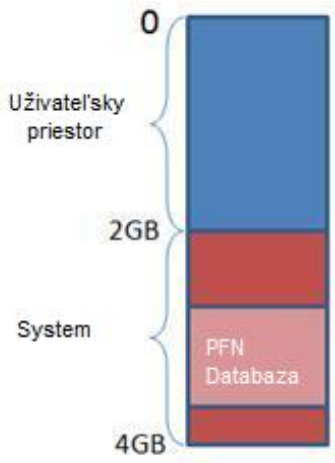
Pri štarte počítača BIOS vykoná alokovanie pamäte počas bootovania systému. Vytvoria sa pamäťové miesta určené pre systém a samostatné časti pamäte určené pre zariadenia. Oblasti vyhradené prevažne zariadeniam môžu byť na rôznych systémoch a rôznych strojoch rozmiestnené rôzne. Spravidla sa však dávajú na koniec adresného priestoru (obr. 1).



Obrázok 1 Rozdelenie oblasti v pamäti v systéme Windows Vista (SUICHE, 2008)

Fyzická pamäť počítača je spravovaná správcom pamäte. Správca pamäte je zodpovedný za obsadenosť pamäte programom a dátami aktívnych procesov, ovládačmi zariadení a samotný operačný systém. Pretože väčšina operačných systémov používa pri svojom behu viac programového kódu a dát, ako sa môže zmestiť do fyzickej pamäte, fyzická pamäť je v podstate len časové okno použité pre kód a dáta. Keď dáta alebo kód nejakého procesu alebo samého operačného systému pri ich potrebe nie sú prítomné v pamäti, správca pamäte ich tam musí z disku nahrať.

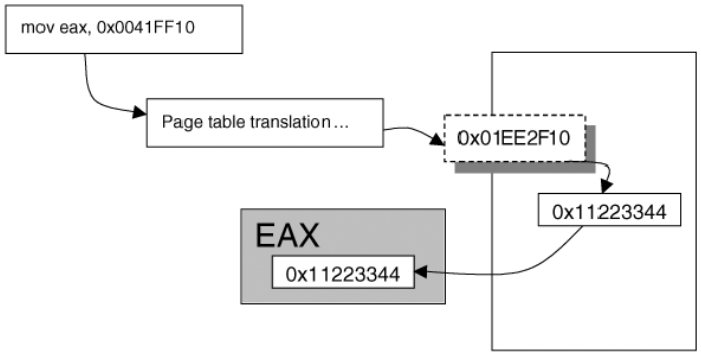
Správca pamäte uchováva zoznam všetkých fyzických rámcov v poli nazvanom PFN databáza. Každý rámec je reprezentovaný 28-bytovou dátovou štruktúrou. V závislosti od veľkosti pamäte, akú systém dokáže obslúžiť, sa mení aj veľkosť databázy. V operačnom systéme sa pre zlepšenie výkonu mapuje celá databáza do virtuálnej pamäte (do oblasti vyhradenej pre systém), čím zmenšuje využiteľný priestor (obr. č. 2).



Obrázok 2 Obsadenosť pamäte

1.1.2 Stránkovanie

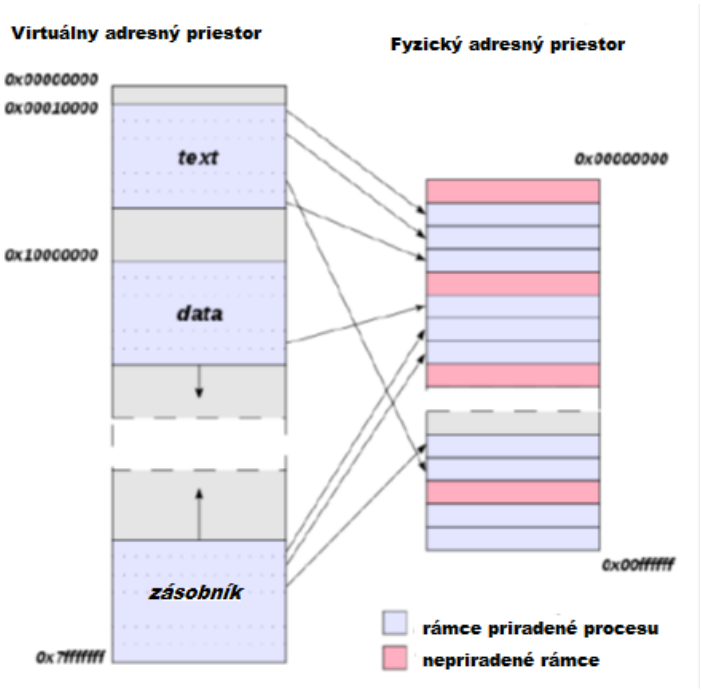
V chránemom režime prechádza počítač do režimu stránkovania pamäte. Základná myšlienka pri stránkovaní je, že virtuálny a fyzický adresný priestor je rozdelený na rovnako veľké bloky. Jednotlivé bloky (stránky) virtuálnej pamäte sú mapované na bloky fyzickej pamäte (rámec). Informácie o mapovaní stránok na rámce sú uchovávané v tabuľkách stránok (PTE) a tabuľkách adresárov pre stránky (PDE). Tieto tabuľky uchovávajú zároveň aj informácie o stave stránok a ich ochranných nastaveniach. Každá stránka má svoje číslo stránky a offset stránky. Ak je zaputé stránkovanie (je nastavený posledný 32. flag PG na 1 v kontrolnom registri CR0), celý proces komunikácie s pamäťou je zabezpečovaný pomocou Memory management unit (MMU). V moderných systémoch je MMU súčasťou architektúry procesora, aby sa minimalizovala časová náročnosť spôsobená prekladom z virtuálej adresy na fyzickú. Keď napríklad EIP register ukazuje na inštrukciu prístupu do pamäte, napr. mov eax, dword ptr [eax], kde v eax je uložená virtuálna adresa v pamäti, procesor požiada MMU o preloženie tejto adresy. MMU vráti procesoru fyzickú adresu. Až tak môže procesor načítať požadovanú hodnotu z pamäte RAM a uložiť ju do registra (obr. č. 3). Do tohto procesu prekladu nemôžeme nijako štandardne zasiahnuť.



Obrázok 3 Preklad adresy z virtuálnej na fyzickú (Hoglund, 2005)

Pri stránkovaní je potrebné si uvedomiť, že linearita virtuálnych adries nezodpovedá linearite fyzických adries v pamäti. Obrázok č. 4 ukazuje, že poradie fyzických adries nezohráva v adresnom priestore nijakú rolu.

.



Obrázok 4 Ilustrácia nelinearity prekladu virtuálnej adresy na fyzickú[2]

Podľa vybranej veľkosti rámca a možného spustenia Physical Address Extension (PAE) režimu, rozdeľujeme stránkovanie na jedno, dvoj alebo trojúrovňové. Najjednoduchšie jednoúrovňové stránkovanie sa vyskytuje v prípade použitia veľkých rámcov 4MB a vypnutého PAE, trojúrovňové sa používa v prípade veľkosti

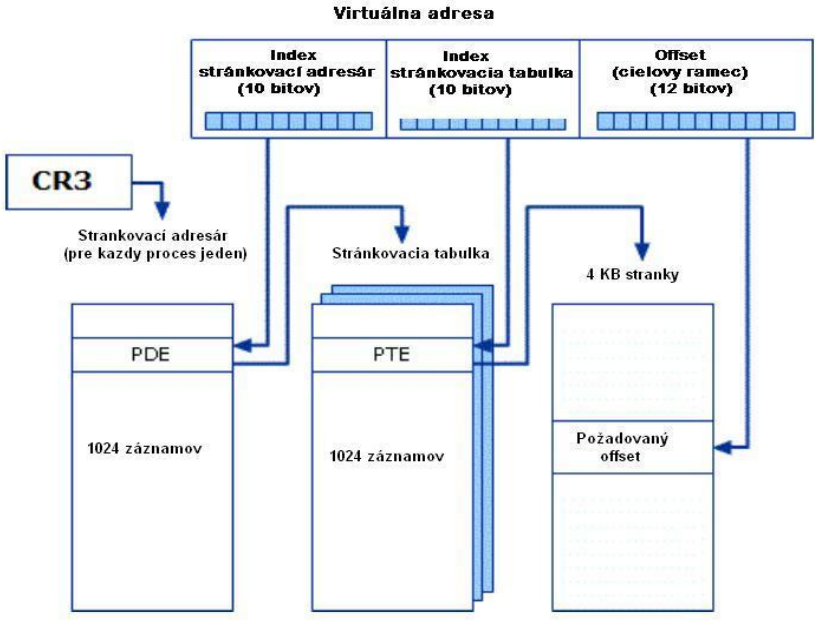
[2] Obrázok vytvorený podľa originálu zo stránky http://en.wikipedia.org/wiki/Virtual_address

rámca 4KB a spusteného PAE (v prípade spusteného PAE sú možné veľkosti rámcov 4KB a 2MB). Na základe úrovní sa virtuálna adresa skladá z rôzneho počtu indexov a offsetu, pričom počet indexov = počtu úrovní.
Pre prípad, keď máme 4KB veľkosť rámcov s vypnutým PAE (to je jeden z najčastejších prípadov), adresa je rozdelená na 3 časti podľa tabuľky č. 1.

Tabuľka 1 Zloženie virtuálnej adresy

31 22	21 12	11 0
Index (stránkovací adresár)	Index (stránkovacia tabuľka)	Offset (cieľový rámec)
0000 0000 01	00 0001 1111	1111 0001 0000
1	1F	F10
0x0041FF10		

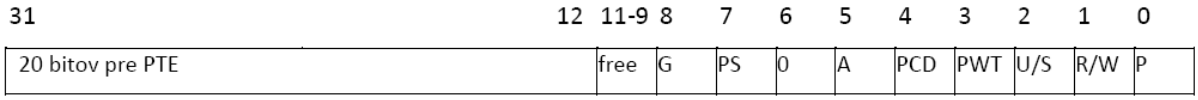
Indexy v adrese určujú pozíciu v tabuľke adresárov a stránok. Preklad začína načítaním hodnoty fyzickej adresy hlavného stránkovacieho adresára (Page Directory), ktorá sa nachádza v registry CR3[3] (pozri obr. č.5). Z prvých 10 bitov virtuálnej adresy (tab.č.1 Index (strankovací adresár)) je určená pozícia v PDE (Page Directory Entry). Záznam na danej pozícii v tabuľke pre stránkovací adresár (PDE) má veľkosť 32 bitov. Prvých 12 bitov je významových a ďalších 20 bitov (doplnených nulami) ukazuje na adresu v stránkovacej tabuľke (Page Table). Nasledovný obrázok názorne prezentuje tento mechanizmus.



Obrázok 5 Postup prekladania virtuálnej adresy na fyzickú pri dvojúrovňovom stránkovaní

[3] Na OS Windows XP sme pri testovaniach zistili, že PDE sa nachádza vždy na tej istej virtuálnej adrese 0xC0300000 a končí na 0xC0300FFF (1024PDE * 4B = 4KB).

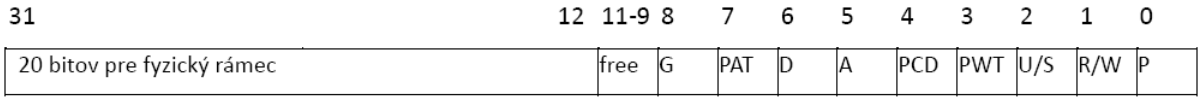
Významové bity slúžia na definovanie pre procesor dôležitých informácií ohľadom pamäťového segmentu, napríklad či bol presunutý z pamäte na disk, veľkosť rámca alebo aké práva pre zápis a čítanie dovoľuje pamäťový segment. Štruktúra záznamu PDE je nasledovná (pozri obr. č. 6):



Obrázok 6 Významové bity v tabuľke stránkovacieho adresára PDE

- Free - nešpecifikované, dostupné pre účely OS,
- G - Global Page (ignorovaný bit),
- PS - Page Size - určuje veľkosť rámca, 0 indikuje 4KB veľkosť rámca,
- A - Accessed bit - indikuje dostupnosť na čítanie alebo zápis,
- PCD - Cache Disabled - určuje, či môže byť stránka/rámec odložený o pagefile, keď je nastavený na 1, stránka nemôže byť odložená na disk,
- PWT - Write-through,
- U/S - User/Supervisor - popisuje, kto má prístup do tejto stránky, 0=kernel,
- R/W - Read/Write, 0=stránka len na čítanie, 1=povolené čítanie + zápis,
- P - Present - ak je nastavený na 1, tak stránkovacia tabuľka je v súčasnosti načítaná v pamäti.

Záznamy v stránkovacej tabuľke (PTE) majú tiež veľkosť 32 bitov (pozri obr. č. 7). Prvých 20 bitov v tomto prípade ukazuje na fyzickú adresu rámca v pamäti. Pretože významové bity v PDE a PTE majú medzi sebou len malé odlišnosti, popíšeme iba rozdiely.



Obrázok 7 Významové bity záznamu stránkovacej tabuľky PTE

- G - Global flag - pomáha ošetriť rámce s častým prístupom, aby neboli odstránené z TLB,
- PAT - Page Attribute Table Index,
- D - Dirty Bit - ak na adresu bol zápis, bude nastavený na 1 (pomáha pri optimalizácii swapovania na disk).

Celý proces prekladu adresy je však zložitejší, ako bol vysvetlený doteraz.
Virtuálny adresný priestor môže byť (a obyčajne aj býva) väčší ako fyzický adresný priestor, a preto môže dôjsť k situácii, keď proces potrebuje väčší priestor v pamäti,

ako je aktuálne fyzicky dostupný. V takom prípade systém musí dočasne uložiť (swapnuť) niektoré dáta z fyzickej pamäte na pevný disk, aby sa uvoľnil priestor v pamäti. Zabezpečí sa to tak, že sa vyberú (na základe algoritmu) niektoré menej aktuálne využívané rámce v pamäti a uložia sa na disk do súboru s názvom „pagefile“. V zázname pre daný rámec v stránkovacej tabuľke sa nastaví nultý (prvý) významový bit ako "not present" (present bit). V prípade požiadavky na prístup k takému „odswapovanému“ rámcu MMU zistí na základe present bitu, že nie je prítomný v pamäti a vygeneruje sa chybové hlasenie „page fault“. Chybové hlásenie je zachytené systémom a vygeneruje sa I/O požiadavka, ktorá zabezpečí načítanie daného rámca do pamäte. V prípade dvojúrovňového stránkovania prístup nejakého procesu do pamäte môže pozostávať z nasledujúcich krokov (pozri obr. č. 5) :

- Pristúpi sa do stránkovacieho adresára (PDE), aby sme overili, či je dotyčná stránkovacia tabuľka prítomná v pamäti (na disk sa môže uložiť aj rámec, ktorý obsahuje stránkovacie tabuľky).
- Ak nie je v pamäti, vygeneruje sa I/O požiadavka, ktorá zabezpečí načítanie stránkovacej tabuľky z disku do pamäte.
- Vyberie z virtuálnej adresy index stránky (PTE).
- Ak je index stránky väčší, ako je počet stránok, vygeneruje chybu „illegall page“.
- Pokúsi sa načítať záznam z požadovanej stránkovacej tabuľky.
- Pristúpi sa do stránkovacej tabuľky, aby sme overili, či je dotyčná stránka prítomná v pamäti.
- Ak nie je v pamäti, vygeneruje sa I/O požiadavka (page fault), ktorá zabezpečí načítanie stránky z disku do pamäte.
- Ak stránka nie je v tabuľke, vygeneruje sa chyba „missing page“.
- Ak je stránka chránená, vygeneruje sa „Access violation“ chyba.
- Nakoniec sa vráti fyzická adresa (číslo rámca * veľkosť ramca + offset).

1.1.3 Pamäť preložených adries (TLB)

Z predchádzajúceho algoritmu pre preklad virtuálnej adresy vyplýva, že v najkomplikovanejšom prípade jednoduchý prístup do pamäte môže vyžadovať tri čítania pamäte a dve I/O požiadavky na disk. Na eliminovanie tohto nadbytočného zaťaženia sa využíva skutočnosť, že veľké množstvo požiadaviek ukazuje na rovnaké miesta v pamäti. Preto procesory obsahujú malú asociatívnu pamäť nazývanú Translation lookaside buffer (TLB).
TLB obsahuje zoznam najpoužívanejších prekladov virtuálnej adresy na fyzickú. K tomu účelu sa využíva L1 alebo L2 cache procesora a obyčajne L1 je najrýchlejšia

pamäť v celom PC. Pri požiadavke na prístupe do pamäte je ako prvé prehľadané TLB, či sa tam nachádza preklad požadovanej cieľovej adresy. Ak sa tam nachádza, návratová hodnota je „hit“, ak nie, tak „miss“. Tým, že prehľadanie TLB je mnohonásobne rýchlejšie ako štandardný preklad virtuálnej adresy, prispieva toto riešenie k zvýšeniu výkonu systému. V súčasnosti sú algoritmy efektivity TLB natoľko úspešné, že TLB hit je viac ako 90% prístupov. V prípade, ak sa preklad adresy v TLB nenájde, vykoná sa štandardný preklad pomocou MMU. TLB je ešte kvôli zlepšeniu efektívnosti rozdelená na ITLB (Instruction TLB), používaná pri prístupe k inštrukciam, a na DTLB (Data TLB), ktorá sa používa v prípade, ak sa pristupuje k dátam.

1.1.4 Stránkované a nestránkované pooly (Paged and Nonpaged Pool)

Stránkované a nestránkované pooly (Paged and nonpaged pools) slúžia ako pamäťové zdroje, ktoré operačný systém a ovládače zariadení využívajú na ukladanie svojich štruktúr. Správca poolov pracuje v móde jadra systému a využíva časti adresného priestoru jadra. Činnosť správcu môžeme prirovnať k C-runtimu alebo k správcovi haldy (heap manager), ktoré bežia v užívateľskom móde. Keďže minimálna veľkosť pamäte, ktorú je možné alokovať, je násobok veľkosti fyzického rámca (4 KB na x86 a x64), správca rozdeľuje určitú alokovanú oblasť na menšie, a tak nedochádza k strate priestoru (ako keby alokoval malý priestor vždy v novom 4 KB rámci). Ak požaduje napríklad aplikácia 512-bytový buffer na uloženie dát, správca vyberie zo svojej alokovanej oblasti potrebnú veľkosť a pridelí ju aplikácii. Ostávajúcu časť priestoru pridá na zoznam pre voľné oblasti. Pri ďalšej požiadavke správca pokračuje alokovaním z voľnej oblasti.

Nestránkovaný pool (Nonpaged Pool)

Jadro alebo ovládače zariadení používajú nestránkovaný pool na uloženie dát, ktoré musia byť dostupné aj vtedy, keď systém nemôže spracovať page fault výnimku (t.j. keď nemôže načítavať stránky z disku do pamäte). Nestránkovaný pool je preto vždy prítomný v pamäti a nedochádza k jeho swapovaniu na disk. Medzi dátové štruktúry, ktoré sa v týchto pooloch uchovávajú, patria objekty reprezentujúce procesy a vlákna, synchronizačné objekty ako mutexy, semafory a udalosti, odkazy na súbory (referencie) reprezentované ako súborové objekty a tiež I/O request pakety (IRPs) reprezentujúce I/O operácie.

Stránkovaný pool (Paged Pool)

Stránkovaný pool umožňuje (už ako napovedá názov) uložené dáta premiestniť na disk (do pagefile). Pri potrebe čítania dát dochádza podobne ako pri užívateľskej

časti pamäte k vyvolaniu page fault výnimky a správca pamäte zabezpečí načítanie dát naspäť do fyzickej pamäte.

1.1.5 Priamy prístup do pamäte (DMA)

Technológia priameho prístupu do pamäte (Direct Memory Access - DMA) je využívaná na prístup k dátam v operačnej pamäti počítača. Používa hardvérový mechanizmus, čím dokážu zariadenia posielať svoje I/O dáta priamo z a do pamäte počítača bez potreby volania inštrukcií procesora. To znamená, že čítanie a zápis dát nie je závislý na operačnom systéme ani type procesora. Procesor iba inicializuje spojenie a tok dát je už ďalej riadený špeciálnym DMA radičom, ktorý je na základnej doske. Tento mechanizmus sa často využíva, lebo nezaťažuje procesor prenosom dát, a tým dokáže výrazne zvýšiť priepustnosť dát (throughput). DMA prenos v súčasnosti používa väčšina hardvérových zariadení pripojených do počítača ako sieťové karty, zvukové karty, radiče pevných diskov a aj grafické karty. Iná možnosť, keď DMA nie je podporované alebo využívané, je použiť programovaný I/O mód, kde musí byť procesor spoluúčastný počas celého prenosu dát. Iba pripomenieme, že väčšina ovládačov alokuje svoje buffre pre DMA na začiatku, čiže pri bootovaní. O alokácii bufferov pre zariadenia sme písali v prvej časti kapitoly. Týmto sa alokované oblasti vyčlenia z oblastí použiteľných pre systém, nie je možné k nim zo systému pristupovať.

Typy DMA

Na základe typu DMA sa mení spôsob správy pamäte. V praxi poznáme 2 základné typy DMA, a to paket-based DMA a common-buffer DMA. Niekedy sa používa aj hybridná kombinácia týchto dvoch typov.

Common-buffer DMA (Wieland, 2006)

Common-buffer DMA je základný a tiež jednoduchší typ. Pri nej sa vytvára oblasť pamäte nazývaná common buffer a tá je prístupná ako pre zariadenie, tak aj ovládaču. Pomocou tohto buffera môžu obidve strany vzájomne komunikovať a posielať dáta.

Tento typ DMA má niekoľko výhod:

- Buffer je uložený ako súvislý blok vo fyzickej pamäti, čo zjednodušuje čítanie a zápis.
- Buffer je vytvorený vo fyzickej oblasti, kde majú zariadenia prístup.
- Zmeny urobené na jednej strane sú okamžite viditeľné na strane druhej.
- Pre funkčnosť nie je potrebný voľný map register.

Hlavnou nevýhodou pri tomto type je, že sa vyžaduje súvislý blok pamäte. Z tohto pohľadu nie je vhodný na veľké prenosy dát. Štandardne nemôžeme v systéme alokovať napr. 1 GB common-buffer, lebo systému sa jednoducho nepodarí nájsť taký veľký nepoužitý blok pamäte.

Paket-based DMA (Wieland, 2006b)

Tento typ DMA rieši problém s alokáciou pri prenose veľkého množstva dát. Každý DMA prenos je spracovaný osobitne a vždy je alokovaná nová oblasť pamäte. Prenos sa vykonáva pomocou paketu dát. Ak chce systém čítať dáta z nejakého zariadenia, poskytne zariadeniu predalokovaný nový buffer, do ktorého zariadenie uloží posielané dáta. Tento typ dokáže lepšie využívať systémové zdroje, a preto je viac rozšírený ako Common buffer DMA.

2 Techniky používané pre prístup k pamäti

V tejto kapitole sa venujeme analýze možností pre prístup k obsahu pamäte. Vo všeobecnosti pamäť počítača nemá priamu podporu pre vonkajší prístup. Údaje z pamäte však možu byť zdrojom cenných informácií pre potreby forenznej analýzy pamäte (Carvey, 2009). Z tohto dôvodu sa začali rozvíjať aj techniky pre prístup a čítanie obsahu pamäte. Tieto techniky sa v nasledujúcej časti pokúsime stručne predstaviť. Autori malware sa však pokúšajú hľadať cesty, ako sa pri forenznej analýze vyhnúť odhaleniu. Z toho dôvodu niektoré techniky vyvinuté pre forenznú analýzu nebudú vhodné pre využitie pri detekcii malware. V kapitole si predstavíme aj návrh techniky pre prístup k obsahu pamäte, ktorý sme samostatne vyvinuli pre účely detekcie malware. Tiež pre zabezpečenie ochrany pred zneužitím techniky sofistikovaným malware sme vyvinuli techniku pre detekciu prístupu k sledovanej oblasti v pamäti.

2.1 Metódy čítania a prístupu do operačnej pamäte

V roku 2006 J. Rutkowska (2006) upozornila na fakt, že nie je možné dovtedy známymi metódami bezpečne čítať (hrozí pád systému) pamäť jadra v systéme Windows. To je spôsobené tým, že jadro nebolo navrhované tak, aby okrem neho mal ešte niekto prístup k daným informáciám. Po analýze rôznych prístupov k čítaniu obsahu pamäte na záver uvádza, že v tejto oblasti bude zrejme potrebná podpora od spoločnosti Microsoft, ktorá by upravila niektoré procesy v jadre tak, aby bolo možné informácie získavať.

Všetky existujúce riešenia, ktoré sa do súčasnosti vyvinuli, sú len náhradou a využívajú techniky, pomocou ktorých sa dá obísť pád systému. Vo všeobecnosti existujú dva možné prístupy, softvérovo a hardvérovo orientované.

2.1.1 Hardvérové metódy pre získanie obsahu pamäte

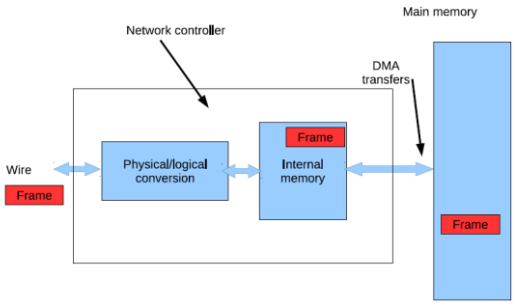
Hlavnou myšlienkou je obídenie operačného systému pomocou fyzického zariadenia. Zariadenie si vytvorí prístup k pamäti pomocou samostatného kanálu, ktorý je nezávislý na operačnom systéme. Veľmi často sa k tomu používa priamy prístup pomocou DMA (Direct memory Access). To umožňuje získanie obrazu pamäte bez nutnosti spustenia ďalšieho procesu (alebo programu na získanie obrazu pamäte) alebo nutnosti použiť potenciálne napadnuté funkcie lokálneho systému. V práci (Carrier and Grand, 2004) bol predstavený koncept hardvérovej PCI karty s názvom TRIBLE, ktorá môže byť použitá na uloženie obsahu operačnej pamäte na externé médium. Daná karta však už musí byť prítomná v systéme v čase, keď potrebujeme získať obsah pamäte. Ďalšou z možností je využitie rozhrania firewire.

Rozhranie firewire bolo vytvorené spoločnosťou Apple a v roku 1995 bolo normalizované ako IEEE 1394. Podľa technickej špecifikácie rozhrania a používaných protokolov firewire rozhranie má plný prístup k operačnej pamäti, čiže z nej môže nielen čítať, ale aj do nej zapisovať. Toto sa deje na hardvérovej úrovni bez vedomia a zásahu operačného systému (Dornseif, 2004). V roku 2006 Adam Boileau vytvoril program schopný pomocou počítača s Linuxom skopírovať obsah operačnej pamäte iného počítača (Boileau, 2006). Nevýhodou je, že pri niektorých konfiguráciach systémov má zbernica firewire problém s hornou oblasťou pamäte (UMA, Upper Memory Area). (Garcia, 2007)

Ďalšou alternatívou spomenutou v tejto sekcii je možnosť získania obrazu pamäte pri virtuálnych operačných systémoch. Virtualizované systémy je možné pri ich behu na čas pozastaviť (zmraziť). Keď je systém pozastavený, obsah pamäte je uložený v súbore. Tým môžeme vytvoriť kópiu daného súboru, a tak získať obsah pamäte.

Viac prác sa venuje návrhu čítania obsahu pamäte s pomocou získania prístupu cez DMA. Prístupy sú takmer všetky zamerané na využitie sieťovej karty. Sieťová karta pri komunikácii s OS používa DMA a zároveň poskytuje možnosť prístupu na sieť a internet, takže obsah pamäte je možné poslať cez sieť priamo na analýzu. Prvýkrát bol tento prístup publikovaný tímom z francúzskeho urádu pre informačnú bezpečnosť (French Office for Information Security alebo Agence Nationale de la Sécurité des Systèmes d'Information, ANSSI)) v roku 2010 (Duflot, 2010). Tím prezentoval zraniteľnosť sieťových kariet typu Broadcom, ktoré mali nesprávne implementovaný protokol ASF (Alert Standard Format) version 2.0 a našli chybu v RISC RX, ktorá je v časti sieťovej karty určenej na odosielanie dát. S využitím tejto chyby boli schopní pristúpiť k obsahu pamäte OS a čítať ju. Postup, ako to však previedli, nie je zverejnený.

Ďalší možný prístup predstavil v roku 2011 Guillaume Delugré (Delugre, 2010). Na rozdiel od predchádzajúceho však nie je založený na zraniteľnosti, ale na modifikovanom firmware sieťových kariet Broadcom. Zmenou firware sa získa možnosť kontrolovať pakety, a tým aj obsah, ktorý je čítaný a zapisovaný cez DMA sieťovej karty. Pre funkčnosť je potrebné meniť BD (deskriptor), ktorý sa používa pri DMA zápise. Delugré našiel spôsob, ako meniť pri DMA zápise pointer (ukazovateľ) na dáta v pamäti, veľkosť dát a flagy. Podrobnejšie je postup popisaný v (Delugre, 2010) a v diplomovej práci (Mydlo, 2012). Nevyhnutnou podmienkou je tu však mať modifikovaný firmware. V sučasnosti modifikovaná verzia firmware však nie je verejne k dispozícii. Taktiež na zabezpečenie posielania dát na iný počítač je potrebné mať ASF podporu vo firmware zapnutú, ale na väčšine sieťových kariet býva z bezpečnostných dôvodov vypnutá.



Obrázok 8 Ilustrácia, ako sa DMA používa pri komunikácii sieťového ovládača so sieťovou kartou (NIC) (Delugre, 2010)

V roku 2011 (Wang et al.) publikovali další na firmware založený prístup pre získanie obsahu pamäte. Kombináciou komerčných sieťových kariet (ktoré sú všeobecne dostupné) s bezpečnostným módom procesorov rady Intel (System Management Mode (SMM)) získali obsah pamäte a obsah CPU registrov (čo je nevyhnutné pre správnu analýzu obsahu pamäte, hlavne CR3 register). Upravili kód, ktorý sa v SMM móde spúšťa, a aby nebolo možné napadnúť ovládač sieťovej karty, naimplementovali ovládač tiež do prostredia SMM. V praxi však je možné upravovať obsah v SMM móde iba pri starších PC, kde SMM mód nie je ešte zablokovaný BIOSom počítača. Nové PC majú prístup do prostredia pre SMM zablokovaný BIOSom. (Wang et al., 2011).

2.1.2 Softvérové metódy pre získanie obsahu pamäte

Tieto nástroje využívajú fakt, že mnoho operačných systémov umožňuje prístup k obsahu pamäte. K tomu slúžia v Unixe zariadenia /dev/mem a /dev/kmem a v systémoch Windows zariadenie \Device\PhysicalMemory. Problém nástrojov využívajúcich dané zariadenia bolo, že obraz vytvárali pomocou prístupu z užívateľského módu. Ale objekty vo fyzickej pamäti sú od verzie Windows Server 2003 SP1 a neskorších verzií vrátane Visty z užívateľského módu nedostupné. Naďalej sú však dostupné z kernel módu. Z tohto dôvodu boli vyvinuté nové metódy, ktoré využívajú vlastný ovládač (driver), pomocou ktorého získajú prístup k pamäti cez mód jadra. (Microsoft Corporation, 2010). Aby bolo možné vytvoriť kópiu pamäte, je nevyhnutné, aby sme nezasahovali do oblastí vyhradených pre DMA bufery ovládačov, a tým predišli nekontrolovanému pádu systému. Pri vytváraní kópie pamäte je potrebné riešiť dve zásadné otázky (Stüttgen, 2013).

1. Výpočet adresného priestoru. Musíme jasne vymedziť, ktorý priestor v pamäti je určený pre systém a ktorý je vyhradený pre DMA bufery ovládačov periférnych zariadení. DMA regiónom sa musíme vyhnúť kvôli možnosti pádu systému.

2. Mapovanie fyzickej pamäte. V systéme môžeme pristupovať k fyzickej pamäti iba jej namapovaním cez virtuálnu pamäť. Mapovanie je potrebné vykonávať po nastavených regiónoch.

Na riešenie týchto otázok je k dispozícii viacero riešení. Ako najvhodnejšie riešenie by sa javilo načítanie štruktúry pamäte priamo z nastavenia BIOSu. Spustenie BIOS funkcie pomocou interrupt Int 15 však nie je možné, keď systém pracuje v chránenom móde po nabootovaní systému. Preto musíme použiť dátové štruktúry alebo API funkcie jadra operačného systému. Tie je ale možné manipulovať, a tým ovplyvniť získaný výsledný obsah pamäte (Stüttgen, 2013).
Pre výpočet adresného priestoru by sme mohli využiť v OS Microsoft Windows symbol MmPfnDatabase, ktorý obsahuje pole s informáciami o každom rámci pamäte a jej využití jadrom systému. Tento symbol však nie je exportovaný a je prístupný iba cez KdDebugBlock štruktúru, ktorá síce tiež nie je exportovaná, ale je využívaná debugerom. Ako vhodnejšia a nástrojmi pre akvizíciu pamäte často využívaná je API MmGetPhysicalMemoryRanges() (Russinovich, 1999). Táto API funkcia nám vráti zoznam fyzických adries oblastí použiteľných pre systém, ktorý bol vytvorený BIOSom. V Linux systémoch symbol iomem_resource naplnený počas bootovania systému obsahuje strom štruktúr, ktoré reprezentujú regióny v pamäti RAM a tiež regióny pridelené DMA. Regióny použiteľné pre systém sú označené ako "System RAM", čo využívajú nástroje pre akvizíciu iba dostupných oblastí. (Sylve, 2012; Cohen, 2011).
Ďalší možný prístup použitý aj v nami vyvinutom riešení (Balogh, 2013) je ošetrenie prípadu, keď sa pomocou funkcie MmMapIoSpace nenamapuje daná časť pamäte. Je to práve vtedy, ak ide o adresný priestor jednotlivých zariadení počítača. MmMapIoSpace vracia hodnotu TRUE, ak sa podarilo pamäť správne namapovať. V opačnom prípade vráti hodnotu FALSE. Prístup spočíva vo vytvorení prázdnych stránok, ktoré sa zapíšu, akonáhle MmMapIoSpace vráti nulovú hodnotu. Prázdna stránka sa zaplní pomocou rutiny ExAllocatePoolWithTag. Rutina slúži na alokovanie pamäte určitého typu. My sme zvolili nestránkovanú, teda NonPagedPool. Rutina nám tiež umožňuje nastaviť veľkosť alokovanej pamäte. Týmto spôsobom zabezpečíme, aby sme nezasahovali do nepovolených regiónov.

Na riešenie druhej otázky existuje tiež viac prístupov. Programy spustené v chránenom móde, nemôžu pristupovať k fyzickej pamäti priamo, ale iba cez mapovanie do virtuálneho adresného priestoru. Mapovanie je realizované pomocou tabuliek PTE, PDE (pozri predošlú kapitolu). Máme niekoľko možností, ako to v OS dosiahnuť. Môžeme využiť funkciu na mapovanie súboru do virtuálneho adresného priestoru API *ZwMapViewOfSection*, lebo všetky operácie pre čítanie z namapovaného regiónu sú sprostredkované akoby zo súboru. Technika vykonáva mapovanie časti objektu *\\.\PhysicalMemory*, ktorý poskytuje operačnému

systému pohľad do fyzickej pamäte. Druhá systémom Windows ponúkaná možnosť je použitie API *MmMapIOSpace(),* ktorá dovoľuje ovládačom mapovať IO regióny alebo priestor fyzickej pamäte do adresného priestoru jadra systému (napr. pre priamy prístup pre PCI konfigurovaný DMA buffer). Existuje aj nedokumentovaná API funcia, napr. *MmMapMemoryDumpMdl(),* ktorá sa štandardne využíva pri dumpe počas pádu systému (Stüttgen, 2012).
Operačný systém Linux používa jadro bez stránkovania, čo umožňuje mať stále lineárne mapovanie medzi virtuálnym adresným priestorom jadra a fyzickým adresným priestorom. Preto je možné celý fyzický adresný priestor vždy mapovať do virtuálneho adresného priestoru s konštantným offsetom. API funkcia *kmap()* na mnohých linux platformách je jednoduché makro, ktoré prekladá číslo fyzického rámca do lineárneho mapovaného virtuálneho priestoru jadra.

2.1.3 Nástroje používané na získanie obrazu pamäte

Takmer všetky softvérové nástroje pre získanie obsahu pamäte využívali zariadenia /dev/mem a /dev/kmem a v systémoch Windows zariadenie \Device\PhysicalMemory. Ako príklad môžeme uviesť nástroj, ktorý pôvodne pochádza z linuxového prostredia, s názvom Data Dumper (DD). V prostredí Linux sa tento nástroj používal už dlho na vytvorenie obrazu disku. Preto mnoho nástrojov na analýzu obrazu podporuje DD formát. Spoločnosť GMG Systems Inc. vytvorila verziu programu, ktorú je možné spustiť v systéme Windows a je schopná vytvoriť obraz obsahu operačnej pamäte. Tento nástroj bol súčasťou Forensic Acquisition Utilities, ale v súčasnosti už nie je dostupný a ani podporovaný jeho vývoj. Ďalší nástroj používajúci podobnú techniku je Nigilant32 od Matt Shannona z Agile Risk Management (Garcia, 2007). Výhodou nástroja je existencia grafického rozhrania. Do tejto skupiny môžeme zaradiť aj ProDiscover IR, ktorý umožňuje vytvorenie zálohy obsahu pamäte operačného systému a taktiež BIOSu.
Problém nástrojov využívajúcich dané zariadenia je, že obraz vytvárajú pomocou prístupu z užívateľského módu. Ale objekty vo fyzickej pamäti sú od verzie Windows Server 2003 SP1 a neskorších verzií vrátane Visty z užívateľského módu nedostupné. Naďalej sú však dostupné z kernel módu. Z tohto dôvodu boli vyvinuté nové metódy, ktoré využívajú vlastný ovládač (driver), pomocou ktorého získajú prístup k pamäti cez mód jadra. George M. z firmy Garner, Jr. (GMG Systems) vyvinul nástroj, ktorý používa tento prístup, pod názvom KnTDD, ktorý je súčasťou KnTTools. Medzi výhody tohto nástroja patrí aj možnosť prevodu získaného obrazu do Windows crash dump formátu. Obraz v tomto formáte je možné analyzovať pomocou Microsoft Windows debuggovacích nástrojov (Carvey, 2009).
Pri používaní týchto nástrojov je potrebné si uvedomiť dva dôležité aspekty. Pri ich spustení dôjde k ich načítaniu do pamäte, a tým sa môžu (v dôsledku stránkovania pamäte) presunúť informácie o niektorom spustenom procese do stránkovacieho

súboru, ktorý je uložený na pevnom disku (page file), o čom hovorí Locardov princíp zmeny. Po druhé pred spustením procesu zálohovania pamäte nedôjde k jeho zmrazeniu a vytvorenie kópie trvá určitý časový úsek. Tým sa obsah pamäte mení aj počas vytvárania obrazu (Libster, 2008).

Alternatívou pre Windows OS je simulácia pádu celého systému (napr. vyvolaním Blue Screen of Death), čím systém zamrzne a automaticky sa začne vytvárať obraz obsahu pamäte (crash dump), ktorý sa uloží na disk. V tomto prípade už systém nebeží, a tak môžeme získať nemenný obraz pamäte. Nevýhodou je, že tým zastavíme systém a je nutný jeho reštart.

Na záver by sme spomenuli ešte niektoré novovzniknuté nástroje z tejto kategórie. V roku 2008 publikoval ManTech International nástroj MDD[4] a Matthieu Suiche svoj vlastný nástroj win32dd[5]. Tieto nástroje spolu s nástrojom winen od Guidance Software sú schopné vytvárať obraz pamäte aj pre novšie verzie Windows. Nástroj win32dd tiež dokáže vytvárať obraz vo formáte Windows crash dump (Carvey, 2009).

Nástroj s názvom Memoryze[6] pri vytváraní obrazu dokáže do procesu zberu zahrnúť aj obsah pamäte, ktorá bola uložená v stránkovacom súbore (page file), čím získavame viac informácií a sú komplexnejšie. Open source projekt Volatility, ktorý sa zaoberá hlavne analýzou obrazu pamäte, vytvoril vlastný nástroj na získanie obrazu pamäte, ktorý bol inšpirovaný nástrojom win32dd pod názvom WinPMEM. Rozhodli sa tak z dôvodu uzatvorenia zdrojových súborov nástroja win32dd. Súčasťou forenzných sád ako Helix a SANS SIFT Workstation je FTK imager od firmy AccessData. Vlastný nástroj pre vytvorenie obrazu pamäte vyvinula aj firma PRoDiscover, ktorá ponúka aj vlastnú komerčnú sadu nástrojov pre forenznú analýzu. Celkový prehľad možností nástrojov a ich porovnanie je vidieť na obr. č. 9. Viac informácií o nástrojoch je možné nájsť v prehľadovom článku (Carvajal, 2013).

[4] Nástroj je možné stiahnúť z odkazu http://sourceforge.net/projects/mdd/

[5] V súčasnosti nahradený MoonSols DumpIt nástrojom a stal sa súčasťou MoonSols Windows Memory Toolkitu viac informácií na stránke http://www.moonsols.com/windows-memory-toolkit/

[6] Podrobnejšie informácie dostupné na stránke www.mandiant.com/software/memoryze.htm

	User Interface	Reporting	Processing Time	Training	Operating System
FTK Imager	GUI	Do not provide	44 seconds	Required minimum training	MS Windows Linux and Mac
Pro Discover	GUI	General Information	110 seconds	Required minimum training	MS Windows
Win32dd	Command line	RAM and additional information details	29 seconds	Required minimum training	MS Windows
Nigilant32	GUI	Live machine snapshot	39 seconds	Required minimum training	MS Windows
Memoryze	Command line	XML output reports about batch results	36 seconds	Required minimum training	MS Windows
Helix3 (dd)	GUI	PDF output report about system information	55 seconds	Required minimum training	MS Windows, Linux

Obrázok 9 Súhrný prehľad nástrojov podľa (Carvajal, 2013)

2.2 Nevýhody a obmedzenia existujúcich metód

V predošlej časti popisované metódy sú schopné zachytiť iba statickú informáciu o činnosti pamäte v jednom konkrétnom okamžiku. Týmito nástrojmi nie sme schopní sledovať dynamický priebeh činností nejakého procesu. Tiež podľa Libstera (2008) doba vytvárania obrazu je dostatočne veľká na to, aby iné procesy menili počas vytvárania kópie dáta v pamäti, a tým sa informácie stávajú nekonzistentné.

2.2.1 Možnosti kompromitácie a anti-forenzné techniky

Ako bolo tiež spomenuté v úvode kapitoly, viac sofistikované rootkity sú schopné obísť detekciu. Rootkity schopné hooknuť Virtual Memory Manager (VMM), ako napr. Shadow Walker rootkit, môžu kontrolovať čítanie pamäte bezpečnostými skenermi (Butler, 2005). Rootkit tak môže presmerovať čítanie na iný blok pamäte alebo zmeniť obsah čítania (viac v časti 3.3 pri popise techniky pre zámenu obsahu pamäte). Tak keď sa skener pokúsi načítať časť pamäte pozmenenú rootkitom, presmeruje sa na pôvodnú nezmenenú kópiu danej časti pamäte. Aby bolo možné detegovať také typy rootkitov, pamäťové skenery nemôžu pracovať s využívaním OS API funkcií alebo procesorom kontrolovaných mechanizmov.
Tento problém sa týka nie iba otázky detekcie malware, ale aj forenznej analýzy vykonávanej pri zapnutom a bežiacom PC (live forensic analysis). Pri forenznej

analýze v zapnutom stave (tiež aj online analýza) môžu byť získavané informácie manipulované rezidentným rootkitom. Analýza pamäte pomocou jej získanej kópie vo vypnutom stave PC je v tomto prípade vhodnejšia a nedáva možnosť manipulovať so získavanými údajmi počas testovania. V tomto prípade rootkit nemôže zasiahnuť do procesu detekcie.
Offline analýza predpokladá vytvorenie kópie pamäte pred vypnutím počítača. Aby nedošlo k modifikácii vytváranej kópie pamäte, je potrebné tiež použiť metódu pre získanie obrazu pamäte, ktorá je proti týmto technikám rootkitov rezistentná. Ďalšie anti-forenzné techniky, ktoré rootkit môže využiť, sú popísané v (Stüttgen, 2013) a ich cieľom je zabrániť funkčnosti nástrojov pre získanie obsahu pamäte. V práci aplikovali nimi predstavené anti-forenzné techniky a testovali odolnosť všetkých známych nástrojov, ktoré vytvárajú kópiu pamäte (sú popísané v časti Nástroje používané na získanie obrazu pamäte). Zistili, že pri aplikácii všetkých navrhnutých techník ani jeden z nástrojov nebol schopný obraz pamäte vytvoriť.
Horeuvedené dôvody a nevhodnosť využitia existujúcich prístupov a metód pre čítanie obsahu pamäte a vytvárania jej kópie pre účely detekcie malware nás viedli k vytvoreniu návrhu vlastnej metódy pre zber informácií a kópie pamäte. Podrobnejší popis navrhnutej metódy a výsledky získané s nástrojom implementujúcim našu metódu popíšeme v nasledujúcej časti práce. Detailný popis je možné nájsť v článku (Balogh, 2013) a v diplomovej práci (Mydlo, 2012).

2.3 Návrh a implemetácia vlastného riešenia pre čítanie obsahu pamäte

Cieľom nášho riešenia bolo vyvinúť nástroj, ktorým by bolo možné čítať obsah operačnej pamäte pre potreby detekcie malware alebo forenznej analýzy.
Ako bolo podrobnejšie popísané v predošlej časti, žiadne z dostupných techník v plnosti nespĺňa požadované podmienky. V tomto smere ideálne riešenie asi neexistuje a neustále sa zlepšujú aj techniky malware na schovávanie pred detekciou. Keď chceme definovať podmienky, ktoré by mal nástroj spĺňať, môžeme ich definovať nasledovne:

- čítanie obsahu pamäte nezávisle a bez možnosti filtrovania obsahu dát procesorom a OS,
- obsah pamäte uložiť na neprepisovateľné médium alebo odosielať na iný počítač v sieti, na ktorý sledovaný systém nemá dosah a nemôže dáta modifikovať,
- nástroj bude možné použiť všeobecne bez nutnosti špeciálneho zásahu do systému alebo špeciálneho hardvéru.

Aby sme mohli splniť požiadavky zo súčasne dostupných možností pre získanie obsahu pamäte, musíme vylúčiť softvérové riešenia, ktoré môžu byť malwarom (rootkitmi) kompromitované. Nám sa ako najvhodnejšie javí riešenie s využitím DMA sieťovej karty, pomocou ktorej budeme čítať obsah pamäte a následne dáta odosielať pomocou sieťovej karty na iný (nami určený) počítač.

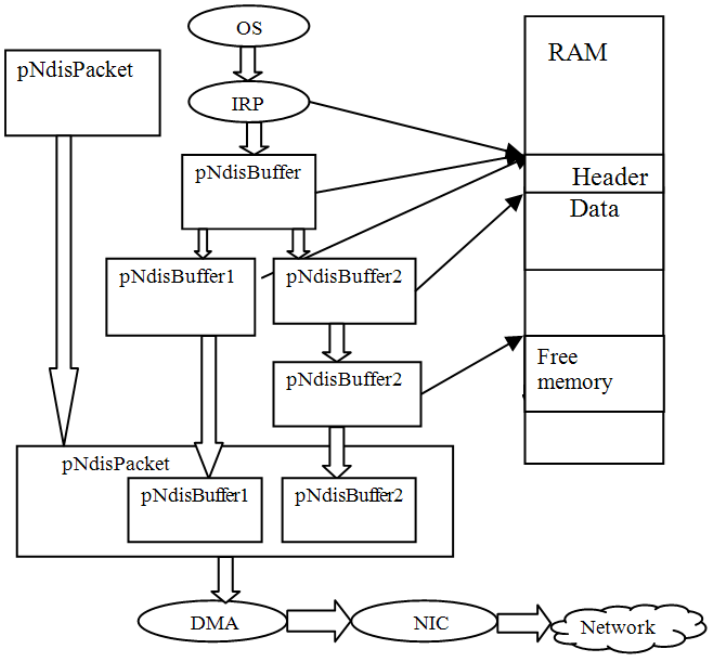
Pri praktickej realizácii riešenia vytvárame sieťový paket, ktorý obsahuje hlavičku a dátovú sekciu. V hlavičke paketu sú informácie na základe typu sieťového paketu, ktorý používame (TCP, UDP). V dátovej sekcii po nami vykonanom presmerovaní ukazovateľa sa nachádza časť obsahu pamäte, ktorú chceme kopírovať. Paket je vytvorený pomocou NDIS ovládača, ktorý pracuje medzi protokolovou vrstvou operačného systému a fyzickou vrstvou (priamo ovládač sieťovej karty). Použitie nízkoúrovňových ovládačov značne obmedzuje (ak nie úplne) možnosť jeho zneužitia malwarom. Informácia o štruktúre a umiestnení paketu sa odošle sieťovej karte. Sieťová karta číta dáta z pamäte pomocou DMA kanálu, čím zabránime možnosti manipulácie s obsahom pamäte, ktorý kopírujeme. Paket je posielaný sieťovou kartou do siete a podľa nastavených údajov v hlavičke paketu (IP adresa, MAC adresa) je prijatý cieľovým počítačom a uložený pomocou riadiaceho programu. Riadiaci program tiež môže komunikovať s NDIS ovládačom, a tak riadiť proces kopírovania dát. Podrobnejší popis riešenia je v časti 2.3.1.

2.3.1 Popísanie funkčnosti programu

Program využíva NDIS protokol ovládač kvôli jeho jednoduchosti vo verzii NDIS 5.1 pre operačné systémy Windows XP. Základnými prvkami v NDIS protokol ovládači sú štruktúry NDIS paket a NDIS buffer. NDIS paket predstavuje štruktúru, v ktorej sú uložené všetky potrebné informácie. V NDIS pakete sa nachádza premenná Head, ktorá je smerník na NDIS buffer. NDIS buffer sa môžu zreťaziť za sebou a v jednom pakete teda môže byť viac týchto bufferov. V NDIS bufferi sa nachádza premenná Next, ktorá obsahuje smerník na tento nasledujúci buffer.

Program pozostáva z dvoch častí, a to je samotný ovládač a riadiaci program. Riadiaci program vyplní hlavičku paketu potrebnými údajmi, a to zdrojovou a cieľovou MAC adresou a typom paketu (ethertype). Taktiež nastaví všetky potrebné parametre, ako je dĺžka paketu a fyzická adresa pamäte, ktorú chceme kopírovať. Všetky tieto údaje sa pošlú ovládaču ako jeden paket. Ovládač NDIS vytvorí dva nové samostatné buffre ndis_buffer1 a ndis_buffer2 pakety. Do prvého uloží dáta z hlavičky paketu a do druhého namapovanú fyzickú adresu na virtuálnu o veľkosti 1500 bajtov. Potom buffer1 a buffer2 znovu zlúčime do jedného paketu, ktorý sa pošle do siete na cieľovú MAC alebo IP adresu z hlavičky (pozri obr. č. 10). Pakety,

ktoré sú poslané do siete a majú nastavenú hodnotu 8999 v poli ethertype, odchytáva riadiaci program a obsah druhej sekcie paketu uloží do súboru.
V prípade, ak chceme iba časť pamäte, riadiaci program pošle iba jeden popr. niekoľko paketov (podľa požadovanej veľkosti) ovládaču s adresou začiatku tejto časti. V prípade, že chceme celý obraz pamäte, riadiaci program v cykle posiela fyzické adresy od 0 až po koniec pamäte. Takto postupne získame obsah celej pamäte.



Obrázok 10 Základný princíp čítania a kopírovania obsahu operačnej pamäte (Balogh, 2013)

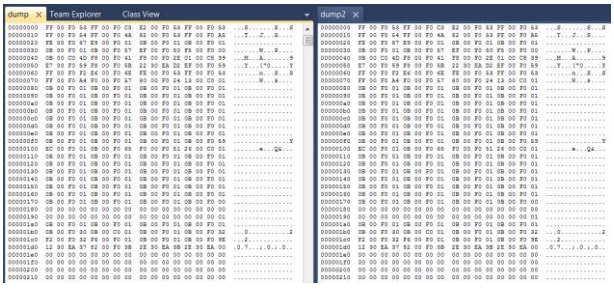
2.3.2 Sledovanie systémovej SSDT

Ako výhoda nášho riešenia je možnosť získavania informácií o štruktúrach a dôležitých objektoch operačného systému pomocou NDIS ovládača, ktorý má systémové práva. Dané informácie môžu byť nápomocné k analýze a získavaniu potrebných dát z pamäte OS. Ako ukážku týchto možností sme NDIS ovládač rozšírili o funkcionalitu získať adresu a veľkosť SSDT tabuľky, na základe čoho vieme získať kópiu oblasti z pamäte, kde sa SSDT tabuľka nachádza. Hookovanie tejto tabuľky začali pre svoje skrývanie využívať rootkity už od roku 2001 (pozri prehľad rootkit techník v časti 3.4). Na detegovanie hooku rootkitom sa zvyčajne sledujú zmeny záznamov zapísaných do SSDT tabuľky. Ak sa udeje zmena v SSDT tabuľke, predpokladá sa, že bola hooknutá, keďže operačný systém obsah tabuľky nemení.

Obsah SSDT tabuľky môžeme pomocou nášho riešenia ukladať do súboru a porovnávaním obsahu v rôznych časových intervaloch môžeme detegovať zmenu (hooknutie) SSDT tabuľky. Podobným spôsobom je možné v prípade potreby doplniť ovládač o ďalšie funkcie. Podrobnejšie informácie o technickej realizácii pre získanie SSDT tabuľky je možné nájsť v práci (Zima, 2014).

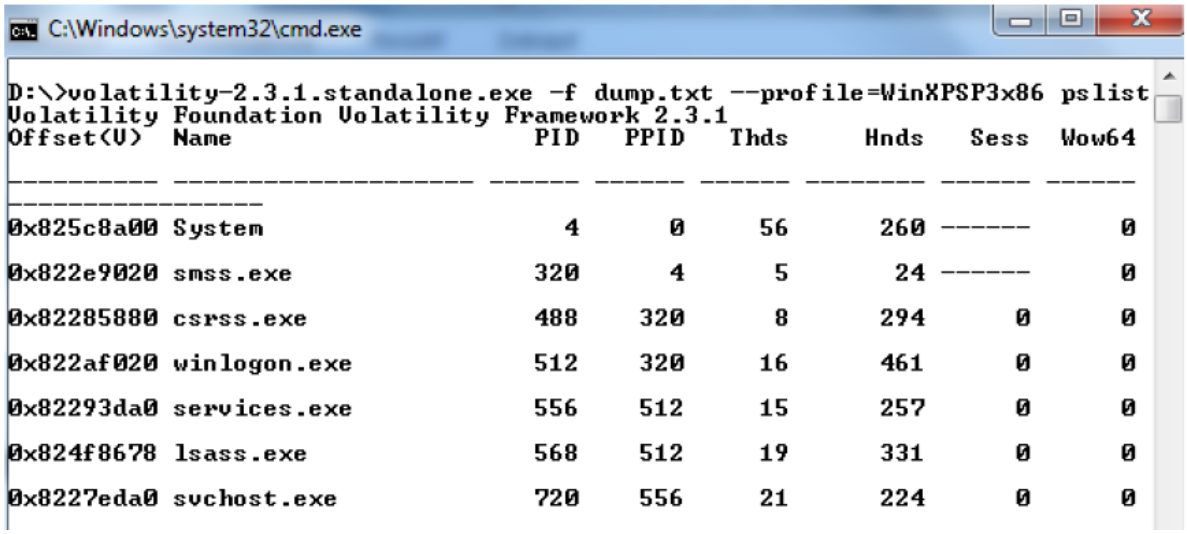
2.3.3 Testovanie

Aby sme mohli overiť schopnosť nášho riešenia čítať obsah celej pamäte alebo vybranej oblasti, použili sme jednoduchú techniku, keď poznáme obsah pamäte vo vybranej oblasti. Pri testovaní sme vybrali adresu pamäte so známym obsahom a následne sme získané dáta porovnali. Obsah pamäte získaný pomocou nášho riešenia zodpovedal obsahu, ktorý sme očakávali. Ako druhý test sme spravili kópiu celej pamäte. Kópiu (dump) sme porovnali s dumpom pamäte vytvoreného pomocou nástroja "MDD tool". MDD je nástroj určený k získaniu obsahu pamäte pre OS Microsoft Windows vyvinutý firmou ManTech International. Získané kópie sa zhodovali (pozri obr. 11).



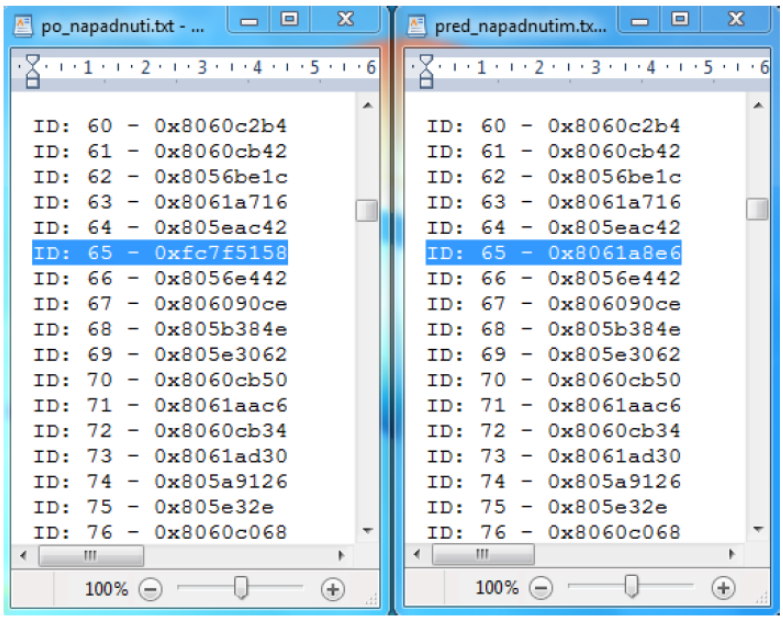
Obrázok 11 Obraz pamäte – porovnanie výsledku z nášho riešenia a MDD nástroja

Po testovaní sme sa pokúsili analyzovať nami vytvorený obraz pamäte v programe Volatility Framework. Volatility framework je projekt otvorený pre verejnosť, zameraný na analýzu obrazu (dumpu) pamäte. Obraz bol frameworkom rozpoznaný a plne funkčný. Týmto sme potvrdili, že obraz vytvorený naším riešením je vhodný aj na digitálnu forenznú analýzu a získavanie dát. Na obrázku č. 12 je vidieť zoznam bežiacich procesov, ktorý sme získali pomocou volatility frameworku z nami vytvoreného obrazu.



Obrázok 12 Zoznam bežiacich procesov z nami vytvoreného obrazu

Pri testovaní správnosti riešenia sme si nechali vypísať aj SSDT tabuľku. Pomocou ovládača SSDTHook.sys vytvoreného na hookovanie SSDT tabuľky sme prepísali niekoľko adries systémových volaní. Následne sme opäť vypísali SSDT tabuľku a obsah tabuliek sme porovnali. Na obrázku č. 13 je vidieť detegované zmeny adries v tabuľke, čo potvrdzuje správnosť detekcie naším riešením a zároveň možnosti jeho využitia.



Obrázok 13 Detekcia zmien v systémovej SSDT

2.4 Aplikácia pre ochranu integrity ovládačov v pamäti

Jednou zo slabých stránok nášho riešenia popísaného v časti 2.3 je možnosť hooknutia NDIS ovládača alebo inej manipulácie sofistikovaným rootkitom priamo v pamäti. K ochrane voči takému útoku je potrebné vyvinúť metódy na ochranu ovládačov vo všeobecnosti, nie len pre naše riešenie. Jedným z prístupov je sledovanie oblasti pamäte (čítanie, zápis), kde sa ovládač nachádza a pri pokuse o zápis do danej oblasti iným programom alebo ovládačom vykonať patričnú reakciu. Návrh takého typu ochrany bol súčasťou nášho výskumu a bolo vytvorené funkčné riešenie. Vyvinutý nástroj dokáže sledovať nízkoúrovňovo každý pokus o zápis a čítanie vo vybranej oblasti pamäte a vykonať reakciu v prípade pokusu o zmenu údajov v sledovanej oblasti pamäte. Spôsob implementácie je značne technicky zložitý, preto v časti 2.4.2 uvádzame zjednodušený popis.

2.4.1 Sledovanie prístupu do pamäte

Pre lepšiu kontrolu prístupu aplikácií do pre nás dôležitých oblastí pamäte boli vytvorené techniky pre sledovanie prístupu do pamäte. V (Miller, 2007) autor uvádza tri metódy, ktoré umožňujú sledovať činnosť aplikácie v pamäti. Tieto metódy môžu byť pre nás inšpiráciou, ako také dynamické sledovanie pamäte realizovať.

Prvá metóda je založená na podsunutí (injection) inštrukčného kódu pred samotný kód aplikácie, ktorá pristupuje do pamäte. Následne za pomoci (podsunutého) inštrukčného kódu získame potrebné informácie o adrese, na ktorú sa pristupovalo. Táto metóda sa nazýva Dynamic Binary Instrumentation (DBI) a je implementovaná napr. vo frameworku DynamoRIO a vo frameworku Valgrind (Nethercote, 2007).

Druhá metóda využíva hardvérové stránkovanie, používanú pri 32-bitových a 64-bitových architektúrach. Pri stránkovaní je implementovaná ochrana, ktorá sleduje, či je stránka prístupná z kernel módu alebo z užívateľského módu. Overovanie sa vykonáva počas prekladu virtuálnej adresy na fyzickú pomocou jednej z kontrolných bitov, ktoré sú uložené na posledných 12 bitoch počas prekladu 32-bitovej adresy. Pomocou tohto bitu sa definuje, či na túto adresu môže pristupovať len proces z kernel módu alebo aj z user módu. Tento bit sa nazýva aj Owner bit. Prepnutím tohto bitu na 0 (0 znamená iba kernel mód, 1 user mód) sa pri pokuse aplikácie o prístup v pamäti vyvolá chyba prístupu Page fault interrupt, nasleduje prerušenie a zavolanie funkcie nt!KiTrap0E. Do tejto funkcie prichádzajú informácie o procese, ktorý pristupoval na dané miesto, ďalej adresa, na ktorú chcel pristupovať, ako aj akcia, či išlo o čítanie alebo zápis. Zaregistrovaním vlastného handlera na zachytenie tejto výnimky môžeme tieto informácie odchytiť a

následne vykonať potrebné kroky, aby aplikácia mohla k danej časti pamäte získať prístup. Nevýhodou týchto dvoch prístupov je, že môžeme pomocou nich sledovať iba oblasť pamäte určenej pre užívateľský mód.
Podobná myšlienka ako pri druhej metóde sa používa aj pri tretej s rozdielom, že vyvolávame výnimku na základe chyby segmentácie pamäte. K tomu sa využívajú DS a ES registre, do ktorých sa nastaví null segment selektor. Tým sa vyvolá chyba general protection fault (pokus o prístup na zakázané miesto). Táto výnimka môže byť zachytená rovnakým spôsobom ako pri druhej metóde.
Inú metódu používajú v rámci projektu Renovo (Kang, 2007), ktorého cieľom je extrakcia ukrytého kódu zo spustiteľných súborov, ktoré využívajú rôzne pakovače. Zápis do pamäte sledujú pomocou tzv. dirty flagu, ktorý je obsiahnutý v PTE (page table entry) pre každú stránku. Ak je nastavené na 1, indikuje, že sa do danej stránky v pamäti zapisovalo.
Ďalšou metódou (v poradí už piatou) je možnosť použitá v známom projekte na ukrývanie rootkitov Shadov walker. Projekt bol predstavený jej autormi Jamie Butler a Sherri Sparksom na konferencii Black Hat v roku 2005 (Butler, 2005). Použitá metóda je podobná tretej, ale nevyužíva sa mechanizmus segmentácie, ale mechanizmus prekladu virtuálnych adries na fyzickú. Keď sa preklad nenachádza v TLB a ani v zázname PTE pre rámec, čiže je označený ako neprítomný (PDE / PTE Present bit = 0, pri bežnej činnosti to znamená, že je odswapovaný na disk), vyvolá sa prerušenie 0x0E a zavolá sa funkcia nt!KiTrap0E. Vedomým odstránením záznamu z TLB a prepísaním hodnoty Present bitu na 0 v zázname PTE môžeme teda vyvolať výnimku. Zachytením tejto výnimky môžeme tak ako pri predchádzajúcich metódach sledovať, ktorá aplikácia sa pokúsila k pamäti pristúpiť. Tento prístup sme využili aj v našej aplikácii pre ochranu integrity ovládačov v pamäti (pozri nasledujúcu časť Aplikácia pre ochranu integrity ovládačov v pamäti).

2.4.2 Popis aplikácie

Jedným z hlavných cieľov aplikácie bola ochrana systémových ovládačov, ktoré sú typu PE (portable executable). Do tejto kategórie spadajú okrem ovládačov aj dynamické knižnice DLL. Každý PE súbor je rozdelený na viac sekcií, ktoré môžeme samostatne chrániť.

Spôsob ochrany

Náš monitor pozostáva z ovládača, ktorý plní funkcie potrebné pre činnosť ochrany a z riadiaceho programu, ktorý zadáva oblasti na sledovanie. Pre sledovanie prístupu do pamäte využíva piatu metódu popísanú v časti 2.4.1. Pri preklade virtuálnej adresy na fyzickú, keď vymažeme záznam o preklade z TLB a v PTE zázname tabuľky označíme sledované rámce pamäte ako neprítomné (PDE / PTE Present

bit = 0), vyvolá sa prerušenie 0x0E a zavolá sa funkcia nt!KiTrap0E. Hooknutie tejto funkcie v IDT a nahradenie vlastnou verziou nám zabezpečí, že sa vykoná naša funkcia namiesto pôvodnej. Samozrejme, že je dôležité zachovať si aj adresu pôvodnej funkcie, ktorá by mala korektne a regulárne riešiť všetky TLB miss, ktoré sú mimo predmetu nášho sledovania. V našej hooknutej IDT funkcii sledujeme, či adresa, na ktorú sa pokúša pristúpiť iná aplikácia, je alebo nie je na zozname chránených. Ak je v zozname, zavoláme funkciu, ktorá nám zistí presne, aký proces chcel pristúpiť a či ide o čítanie alebo o zápis.

Implementácia ochrany

Z hľadiska ochrany sú pre nás najdôležitejšie nasledovné sekcie:

- *.text* - z významového hľadiska má pre nás najväčší význam, lebo obsahuje zdrojový kód celej aplikácie v tvare opcode. Vykonávanie kódu teda bude vždy prebiehať len v tejto sekcii. Cieľom inline hookov je vždy infikovanie sekcie, do ktorej vstupuje EIP, a preto je atraktívna ako cieľ útokov, tak aj ochrany.
- *.data* - obsahuje hodnoty globálnych premenných, ktorých prepísanie by znamenalo, že program by pracoval s nesprávnymi dátami.
- *.rdata* - obsahuje debuggovacie informácie (miesto uloženia .pdb symbolov po kompilácii), ale tiež dôležitú IAT, ktorá býva častým cieľom hookovania. Jej podvrhnutím by útočník mohol zabezpečiť, že chránený ovládač bude importovať falošné funkcie z cudzích zdrojov.

Ochranou jednotlivých sekcií PE súboru chceme zabrániť akýmkoľvek hookom IAT tabuľky programu, kódu procesu alebo jeho premenných. Ochranou hlavičky chceme zabezpečiť akékoľvek prípadné pokusy o narušenie alebo rozšírenie tohto súboru. Pre ochranu jednotlivých sekcií a hlavičky sme si zaviedli konštanty, aby bolo vždy možné definovať, ktoré sekcie budeme chrániť. Aktuálne dostupné sú:

- PROTECT_CODE (0x00000001)
- PROTECT_DATA (0x00000002)
- PROTECT_HEADER (0x00000004)
- PROTECT_RDATA (0x00000008)

Forma volania s maximálnym stupňom ochrany je teda nasledovná:

ProtectDriver("ovladac.sys", PROTECT_CODE | PROTECT_DATA | PROTECT_RDATA | PROTECT_HEADER, status).

Prvým argumentom pre vstup je názov ovládača, ktorý chceme ochrániť, ďalším sú možnosti ochrany spojené logickým súčtom OR a posledným výsledný status, ktorý v prípade úspešného ochránenia zaznamená index do poľa chránených modulov.

Keď sú splnené všetky predpoklady na to, aby sekcie mohli byť sledované, zmenou present bitu ich zneviditeľníme pre systém a následne vymažeme záznam aj z TLB. Úkony sú vykonané vo funkcii HookMemoryPage. Deklarácia funkcie HookMemoryPage má nasledovný tvar:

```
void HookMemoryPage( PVOID pExecutePage, PVOID pReadWritePage, PVOID
pfnCallIntoHookedPage, PVOID pDriverStarts, PVOID pDriverEnds) ,
```

kde
pEcecutePage je adresa na stránku, ktorá bude použitá na naplnenie ITLB,
pReadWritePage je adresa na stránku, ktorá bude použitá na naplnenie DTLB (zhodná s *pEcecutePage*),
pfnCallIntoHookedPage je adresa inštrukcie, ktorá bude zavolaná na naplnenie ITLB,
pDriverStarts je počiatočná adresa chránenej sekcie,
pDriverEnds je konečná adresa chránenej sekcie.

Cieľom funkcie je zaradiť sledovaný rámec do zreťazeného zoznamu sledovaných a spraviť z neho rámec, ktorý bude pre procesor neprítomný v pamäti. Tiež je potrebné zabezpečiť hookovanie IDT vektora 0x0e.

Hookovanie IDT vektora 0x0e

IDT hook musí byť zavedený ešte pred vložením prvého rámca do zoznamu sledovaných a zabezpečuje to funkcia *HookInt*.

```
HookInt( &g_OldInt0EHandler, (unsigned long)NewInt0EHandler, 0x0E )
```

Podrobnosti, ako hookovanie prebieha, z dôvodu obsiahlosti nebudeme uvádzať. Na uchovanie sledovaných rámcov využívame zreťazený zoznam, ktorý má nasledujúcu štruktúru:

```
_HOOKED_LIST_ENTRY.
typedef struct _HOOKED_LIST_ENTRY
{
_HOOKED_LIST_ENTRY* pNextEntry;
PVOID pExecuteView; // pointer used to load the ITLB
PVOID pReadWriteView; // pointer used to load the DTLB
PPTE pExecutePte; // pointer to PTE of pExecuteView
PPTE pReadWritePte; // pointer to PTE of pReadWriteView
PVOID pfnCallIntoHookedPage; // called function to load ITLB
ULONG pDriverStarts; // start of the protected page
ULONG pDriverEnds; // end of the protected page
} HOOKED_LIST_ENTRY, *PHOOKED_LIST_ENTRY.
```

Po naplnení štruktúry dátami odovzdanými z parametrov a dopočítaní pPTE funkciou GetPteAddress môžeme vložiť prvý rámec do zoznamu volaním:

PushPageIntoHookedList(HookedPage).

Po uložení sledovaných rámcov do zoznamu označíme rámce ako neprítomné v pamäti RAM označením present bitu na 0.

```
mov eax, pPte
and dword ptr [eax], 0xFFFFFFFE //mark the page not present
```

Keďže pôvodný preklad je stále v TLB, potrebujeme zaručiť, že všetky prístupy od tohto času pôjdu už len výhradne cez náš exception handler. Odstránime teda preklad z TLB, a tým je schovanie rámca dokončené:

```
invlpg pExecutePage .
```

Riadenie prerušenia 0x0E

Aby sme mohli riadiť činnosť po vyvolaní výnimky 0x0e v nami zmenenom IDT, musíme mať informácie o tom, o aký prístup do sledovanej oblasti pamäte ide a ktorý proces sa o prístup pokúša. Pred tým, než procesor vyvolá prerušenie, do registra CR2 umiestni cieľovú adresu, ktorá vyvolala prerušenie. Súčasne na zásobník (stack) inštrukciou push vloží register EIP, ktorý vykonával operáciu prístupu do cieľovej pamäte a následne aj chybový kód, ktorý nám môže podať detailné informácie o vykonávanej operácii.
Chybový kód pozostáva z troch bitov a je dôležitý pre správne riadenie pôvodnej funkcie *nt!KiTrap0E*. Podľa neho sa vykonávajú operácie vyhľadania rámca na disku, jeho opätovné načítanie do pamäte a naplnenie TLB. Bity majú významy podľa tabuľky č. 2.

Tabuľka 2 Významy jednotlivých bitov chybového kódu exportovaného procesorom pri prerušení 0x0E

Poradie bitu	Význam
Bit 0	(P) Present flag
Bit 1	(R/W) Read/Write flag
Bit 2	(U/S) User/Supervisor flag

Podľa významu bitov prerušenia 0x0E rozlišujeme výslednú operáciu, preto je tento údaj kľúčovým k detekcii útoku. Kombinácia jednotlivých bitov indikuje podľa hodnôt stavy uvedené v tabuľke č. 3.

Tabuľka 3 Dekadické a bitové vyjadrenie jednotlivých možností vyvolania prerušenia

D	U/S	R/W	P	Význam
0	0	0	0	Kernel proces chce čítať rámec, ktorý je mimo pamäť
1	0	0	1	Kernel proces počas čítania rámca vyvolal výnimku
2	0	1	0	Kernel proces chce zapisovať do rámca, ktorý je mimo pamäť
3	0	1	1	Kernel proces počas zapisovania do rámca vyvolal výnimku
4	1	0	0	Užívateľský proces chce čítať rámec, ktorý je mimo pamäť
5	1	0	1	Užívateľský proces počas čítania rámca vyvolal výnimku
6	1	1	0	Užívateľský proces chce zapisovať do rámca, ktorý je mimo pamäť
7	1	1	1	Užívateľský proces počas zapisovania do rámca vyvolal výnimku

Väčšina exception handlerov sa snaží minimalizovať kód, a aj preto sú funkcie typu naked, čo označuje, že funkcia nemá prológ ani epilóg. Funkcia nemá argumenty ani návratovú hodnotu, avšak procesor na stack umiestnil chybový kód spolu s inštrukciou ešte pred zavolaním exception handlera. Volanie samotnej funkcie je v tvare

void __declspec(naked) NewInt0EHandler(void).

Celý obsah funkcie pre spracovanie výnimky (exception handler) je programovaný v assembleri, keďže sme už vo výnimke a snažíme sa o čo najmenší počet volaní externých funkcií a prístupu do pamäte. Súčasne sa snažíme dosiahnuť čo najrýchlejší kód resp. volanie pôvodného handlera v čo najkratšom čase, aby mal tento hook minimálny dopad na rýchlosť systému. Úvodný kód funkcie vyzerá nasledovne:

```
pushad
mov edx, dword ptr [esp+0x20] //PageFault.ErrorCode
test edx, 0x04 //if the processor was in user mode, pass it down
jnz PassDown
mov eax, cr2 //faulting virtual address
cmp eax, HIGHEST_USER_ADDRESS
jbe PassDown //we don't hook user addresses, pass it down .
```

Na začiatku si zálohujeme obsah všetkých registrov inštrukciou pushad, aby sme nenarušili beh jadra za prerušením. Hneď v úvode filtrujeme výnimky, ktoré nie sú cieľom nášho sledovania. Najskôr si podľa chybového kódu prerušenia zistíme, či na adresu pristupoval užívateľský proces alebo jadro. Užívateľské procesy nesledujeme, preto riadenie hneď prepustíme pôvodnému handleru. To isté sa vykoná, aj pokiaľ cieľová adresa nie je z priestoru jadra, teda je nižšia ako 0x7FFF0000 (pod konštantou HIGHEST_USER_ADDRESS). Keďže sa venujeme len

ochrane ovládačov jadra, tieto adresy sú mimo nášho predmetu záujmu. Adresu pôvodného handlera máme uloženú v globálnej premennej, preto ju môžeme volať priamo inštrukciou skoku.

```
PassDown:
popad
jmp g_OldInt0EHandler
```

V tomto kroku už vieme, že jadro pristupovalo k adrese v priestore jadra. Nasleduje teda zistenie, či išlo o sledovaný priestor.

```
push eax
call FindPageInHookedList
mov ebp, eax //pointer to HOOKED_PAGE structure stored in list of hooked pages
cmp ebp, ERROR_PAGE_NOT_IN_LIST
jz PassDown //it's not a hooked page
```

V registri EAX sa stále nachádza hodnota registra CR2, takže na stack ako argument umiestnime adresu požiadavky. Volaním funkcie *FindPageInHookedList* prehľadávame v zozname sledovaných stránok, či sa táto adresa nenachádza medzi sledovanými. Funkcia nám vracia štruktúru PHOOKED_LIST_ENTRY, prípadne chybový kód pod konštantou ERROR_PAGE_NOT_IN_LIST. Pokiaľ sa adresa nenachádzala v zozname sledovaných, znova posunieme riadenie pôvodnému handleru. Ak sa adresa nachádza v rozsahu sledovaných adries, musíme ďalší proces spracovania výnimky riadiť sami. TLB bude napĺňať náš exeption handler. V závislosti od toho, či sa jedná o vykonanie kódu ovládača (execute) alebo ide o čítanie alebo zápis do pamäte, naplníme ITLB alebo DTLB prekladom z virtuálnej adresy na fyzickú. Detailný postup tu zase pre náročnosť postupu nebudeme uvádzať.

ITLB môžeme naplniť len volaním inej inštrukcie v danej sekcii. Pretože je nebezpečné zavolať náhodný kód v ovládači, budeme volať vloženú inštrukciu ret, ktorú sme vložili pri inicializácii na adresu pfnCallIntoHookedPage. Jej zavolanie nijako neovplyvní stav stacku, flagov ani registrov a zaberá len jeden byte, preto je ideálna na náš účel. Keďže stránka musí byť označená ako prítomná v pamäti, volaniu musí predchádzať nastavenie present bitu v PTE na hodnotu 1. Volaním call [ebp].pfnCallIntoHookedPage procesor bude hľadať adresu primárne v ITLB. Keďže ju tam nenájde, začne preklad adresy a pri nájdení PTE ako prítomnej v pamäti uloží adresu rámca do ITLB. Operáciu prístupu zopakuje a po ukončení vráti beh do nášho handlera. Nakoniec si nastavíme v PTE present bit na 0, aby sa pri ďalšom pokuse opäť zachytil tento prístup. Aby sa táto operácia vykonala podľa možností ako atomická, celá táto časť je ohraničená vypnutými maskovateľnými prerušeniami.

Funkcia sa ukončí návratom z prerušenia iretd. ITLB bola naplnená a pôvodný program môže pokračovať ďalej.
V prípade, že prístup bol vyhodnotený ako čítanie/zápis do sledovaného priestoru, potrebujeme ešte odfiltrovať prípady, kedy modul pristupuje k svojim vlastným premenným resp. pokiaľ robí zápis do svojich vlastných častí (čo by i tak bolo veľmi nezvyčajné). Takýto spôsob prístupu by sme nemohli brať ako útok.

```
TestDTLB:
mov edx, [esp+0x24] //eip
cmp edx,[ebp].pDriverStarts
jb AssertUser
cmp edx,[ebp].pDriverEnds
ja AssertUser
```

V štruktúre vrátenej volaním *FindPageInHookedList* sme mali okrem iného aj počiatočnú a konečnú adresu modulu zaokrúhlenú na veľkosť rámca. Tieto využijeme na porovnanie s pointrom inštrukcie na zistenie, či sa jedná o prístup do vlastného priestoru. Pokiaľ sa zistí, že inštrukcia je mimo priestor, prišlo k čítaniu alebo zápisu do chráneného priestoru iným modulom jadra a beh programu prejde do volania AssertUser.

```
AssertUser:
push eax
mov eax,cr2
push eax // CR2
mov eax, [esp+0x2c]
push eax // EIP
mov eax, dword ptr [esp+0x2c]
push eax // PageFault.ErrorCode
mov eax, dword ptr [ebx]
push eax // PTE
call AlertUser
pop eax

jmp LoadDTLB
```

V tomto kroku už predpokladáme, že ide o podozrivú činnosť. Všetky zistené údaje sú uložené na stack a nakoniec sa zavolá naša rozhodovacia funkcia *AlertUser*. Tam už bude porovnaním chybového kódu rozlišovaná operácia čítania a zápisu.
Následne po detekcii prístupu a prípadnej reakcii na zápis do chránenej oblasti môžeme pre pokračovanie naplniť DTLB. Naplnenie DTLB bude podobné ako naplnenie ITLB s tým rozdielom, že namiesto inštrukcie call bude použitá inštrukcia mov. Najskôr si podľa vyššie vysvetleného chybového kódu určíme, či išlo o čítanie alebo zápis do pamäte. Ak išlo o čítanie, tak opäť označíme príslušnému PTE present bit na hodnotu 1 a zavoláme mov eax, dword ptr [eax], čím zabezpečíme, že procesor

nenájde preklad v DTLB (TLB miss), avšak preloženie bude úspešné, a tak do DTLB uloží získaný preklad. Následne obnovíme present bit na 0 a dokončíme beh handlera. V prípade zápisu budeme postupovať podobne s výnimkou, že nebudeme robiť čítanie z pamäte, ale zápis ľubovoľnej hodnoty (v našom vzore 0x01).

Rozhodovacia funkcia AlertUser

V prípade zachytenia čítania alebo zápisu do oblasti sledovanej pamäte je volaná funkcia *AlertUser*. Jej deklarácia je v tvare

void AlertUser(ULONG pte, ULONG errorCode, ULONG attackerAddr, ULONG destinationAddr).

Argumenty funkcie majú nasledovný význam:
pte - PTE rámca, do ktorého smerovala operácia čítanie/zápis,
errorCode - chybový kód exportovaný procesorom,
attackerAddr - adresa inštrukcie, ktorá vykonala čítanie/zápis,
destinationAddr - cieľová adresa, do ktorej prebiehalo čítanie/zápis.

Kľúčovým argumentom pre nás je *errorCode*, podľa ktorého aplikovaním masky 0x2 rozhodneme, či išlo o operáciu čítania alebo zápisu. Operáciu čítania sme sa v našom riešení rozhodli ignorovať, no takýmto spôsobom by sme vedeli zachytiť napríklad antivírusovú kontrolu, ktorá číta obsah pamäte a hľadá podozrivý kód.

Aktuálne už vieme, že spracovávame útok do kritickej sekcie. Dohľadáme informácie o útočníkovi a cieli skrze funkciu *GetModuleNameByAddr*, ktorá podobne ako pri inicializácii v *GetModuleBase* exportuje zoznam modulov volaním *ZwQuerySystem-Information* s parametrom návratovej hodnoty *SystemModuleInformation*.

Nakoniec rozhodovacia funkcia vypíše nasledovné informácie popisujúce útok:
- PTE cieľovej oblasti,
- flagy pre PTE cieľovej oblasti (hexadecimálne a binárne),
- názov modulu, ktorý vykonal útok,
- adresu útočiacej inštrukcie,
- RVA útočiacej inštrukcie,
- názov modulu, ktorý bol cieľom útoku,
- adresu cieľovej pamäte,
- RVA cieľovej pamäte.

Keďže naším cieľom bolo navrhnúť monitor integrity, vykonanie ochrannej reakcie na detegovaný útok je už otázka rozhodnutia. Ak by sme chceli zabrániť dokončeniu zápisu, môžeme vyvolať prerušenie int 0 (delenie nulou) alebo ukončiť program,

ktorý sa o zápis pokúša. Keďže vieme odhaliť tento pokus ešte pred jeho zavedením, môžeme účinne ochrániť sledovaný ovládač, v našom prípade NDIS ovládač používaný pri získavaní obsahu pamäte.

2.5 Sumarizácia

V kapitole sme sa venovali analýze riešenia pre priamy prístup do pamäte a získanie jej obsahu, potrebné k detekcii malware a forenznej analýze. Výsledky analýzy sme sa snažili implementovať v našom riešení pre prístup do pamäte. Za hlavný prínos nášho riešenia považujeme nasledovné:

- Dáta z pamäte sú prenášané pomocou DMA sieťovej karty, čo zabraňuje priamemu prístupu CPU k dátam, a tým aj možnosti manipulácie s nimi.
- Obsah pamäte posiela cez sieť a ukladá mimo sledovaný systém, čo zabraňuje modifikácii dát dodatočne po ich uložení na disk.

Navrhnuté riešenie (pri použití interrmediate alebo miniport drivera) tiež umožňuje riadenie procesu získavania čiastkového alebo úplného obrazu pamäte cez vzdialený počítač. Pomocou navrhnutého protokolu je možné komunikovať s ovládačom pomocou IP paketov, ktoré sú prijaté sieťovou kartou a následne zachytené NDIS ovládačom. Riešenie je možné nasadiť na ľubovoľný systém s OS Windows bez nutnosti inštalácie doplnkových programov (stačí inštalácia NDIS ovládača). Tiež nie je potreba reštartu systému. Tým sme splnili požiadavky, ktoré by mal spĺňať nástroj pre získanie informácií z pamäte pre potreby detekcie malware.

Použitie NDIS ovládača umožňuje tiež získavať fyzické adresy kritických komponentov OS a štruktúr z prostredia operačného systému (IAT / EAT / SSDT / IDT/ IRP tabuľky, funkcie jadra, kľúčové dátové štruktúry). Na základe týchto informácií môžeme analyzovať iba vybranú oblasť v pamäti, kde sa dané objekty a štruktúry nachádzajú.

Sčasti môžeme za nevýhodu daného riešenia považovať nutnosť inštalácie ovládača do OS predtým, ako chceme získať obraz pamäte. K tomu potrebujeme administrátorské práva a prístup k počítaču. Ovládač je možné ale nastaviť tak, aby ho stačilo nainštalovať iba raz (aby bol prítomný aj po reštarte systému). Táto nevýhoda môže byť ale zároveň ochrana, ktorá zabraňuje zneužitiu tohto riešenia. Ako ideálne riešenie by bola implementácia nášho ovládača priamo cez výrobcu do operačného systému, ako navrhujú pre bezpečnostné riešenia autori v (Petroni, 2004).

Slabou stránkou riešenia je možnosť hooknutia NDIS ovládača alebo inej manipulácie sofistikovaným rootkitom priamo v pamäti. Popis nami navrhnutej ochrany pred touto hrozbou aj s analýzou možností sledovania prístupov do pamäte

je v poslednej časti kapitoly (časť 2.4.1). Ide o zjednodušený popis funkčnosti a postupu implementácie ochrany pred hooknutím alebo iným pokusom o zmenu kódu ovládača. Viac informácií k implementácii ochrany je možné nájsť v práci (Ivaniš, 2013). Pri zohľadnení výhod a nevýhod nášho riešenia pre získavanie obrazu pamäte s využitím NDIS ovládača a DMA sieťovej karty na základe doterajších skúseností s jeho používaním môžeme konštatovať, že má potenciál byť úspešne použitý k detekcii sofistikovaného a pokročilé techniky schovávania používaného malware. Riešenie môže byť nasadené aj ako súčasť agenta pre zber dát z operačného systému pri komplexnejších bezpečnostných riešeniach. Všeobecné možnosti využitia pre detekciu malware, ako aj pre forenznú analýzu však musia byť podrobnejšie testované.

Možnosť čítania obsahu pamäte nám umožňuje prístup k ďalším informáciam, ktoré sa v pamäti môžu nachádzať. Môžu tam byť odšifrované časti textov, identifikátory relácií (sessions ID) alebo šifrovacie kľúče. Otázke možností získania šifrovacích kľúčov priamo z pamäte sa venuje veľká pozornosť hlavne v oblasti forenznej analýzy, kde je potrebné získať pri dokazovaní z počítača odšifrované dáta aj v prípade odmietnutia poskytnutia hesla podozrivým. Popisu možnej techniky získania šifrovacieho kľúča priamo z pamäte sa venujeme v článku (Balogh, 2011). Pri tejto problematike narážame na dve protichodné požiadavky. Z jednej strany je tu snaha o vytvorenie čo najbezpečnejšieho kryptografického systému pre účely legitímnej ochrany obchodných, tajných alebo inak citlivých informácií a na druhej strane je potreba dešifrovania digitálnych dôkazov pri podozrení z porušenia zákona. Tejto otázke a celkovému prehľadu o dostupných možnostiach a spôsoboch získania šifrovacích kľúčov je venovaná nami spracovaná kapitola v knihe Multidisciplinary Perspectives in Cryptology and Information Security (Balogh, 2014). V budúcnosti preto považujeme za potrebné riešiť aj otázku, aby nami navrhnuté riešenie pre získanie obsahu pamäte nemohlo byť zneužité na prístup k citlivým informáciam pomocou získania šifrovacieho kľúča neoprávnenou osobou, ale na druhej strane aby bolo nápomocné pri forenznej analýze. Takto sa problematika čítania obsahu pamäte mení z jedného cieľa, získavania informácií pre detekciu malware, na komplexnejší problém, lebo pri súčasnom využívaní počítačových systémov v pamäti sa uchovávajú rôzne typy citlivých informácií. Príkladom môže byť široké spektrum popísaných možností zneužitia tzv. heardbleed bezpečnostnej chyby (označenie chyby CVE-2014-0160) v implementácii OpenSSL knižnice (používanej aj pre zabezpečenie šifrovanej komunikácie pri vytvorení VPN pripojení a internet spojení pomocou protokolu SSL/TLS). Pomocou tejto chyby je možné komukoľvek s internetovým prístupom na server čítať obsah jeho pamäte, čo

umožňuje napr. odcudziť súkromné kľúče a následne dešifrovať citlivé údaje, ako sú heslá, čísla platobných kariet a podobne[7].

Zabrániť zneužitiu však zrejme bude možné, iba ak sa pre uchovanie šifrovacieho kľúča v pamäti nájde bezpečnejšie riešenie a tvorcovia šifrovacích programov ho implementujú vo svojich produktoch. Ak by sme chceli túto otázku riešiť iba z našej strany, dostupnými prostriedkami by bolo riešenie ľahko zkompromitovateľné. Ako teoretické riešenie problému by možno prichádzalo do úvahy vytvorenie špeciálnej oblasti v pamäti, kde by sa kľúče uchovávali. Išlo by o šifrovanú oblasť, ktorá by bola šifrovaná šifrovacími kľúčmi, vytvorenými tvorcami programu a uloženými na bezpečnom tokene alebo USB kľúči. Používateľ by pri šifrovaní musel daný token (alebo USB kľúč) pripojiť k počítaču. Pre účely forenznej analýzy by na základe súdneho rozhodnutia tvorcovia šifrovacích programov poskytli šifrovací kľúč (uložený na tokene) pre odšifrovanie špeciálnej oblasti v pamäti k dispozícii vyšetrovateľom. Ešte vhodnejšie by bolo, keby sa taká oblasť vytvorila priamo pri návrhu operačného systému a všetci výrobcovia šifrovacích programov by využívali túto oblasť na ukladanie šifrovacích kľúčov. Nájdenie vhodného riešenia je otázka, ktorá musí zohľadniť množstvo aspektov, čo si vyžaduje ďalší výskum v danej oblasti.

[7] Viac informácií o tejto „katastrofickej" bezpečnostnej chybe v Open SSL knižnici je možné získať na stránke https://www.schneier.com/blog/archives/2014/04/heartbleed.html alebo tiež na stránke http://heartbleed.com

3 Techniky používané škodlivým kódom

Práca je zameraná na analýzu pamäte. Analýza však má byť vykonávaná s cieľom odhaliť potenciálny škodlivý kód (malware). Malware je v slovníku pojmov[8] definovaný ako „Ľubovoľný softvér navrhnutý tak, aby vykonal niečo, čo si užívateľ nepraje vykonať, nepýta sa, či to môže vykonať a často užívateľ ani nevie, že sa to vykonalo (keď to zistí, je už neskoro)“.
Útočníkmi, často využívanými typmi malware pri kompromitovaní systémov, sú rootkity, backdoory a trójske kone. V našej práci ale nebudeme robiť klasifikáciu škodlivého kódu týmto spôsobom, aj keď sa najčastejšie v literatúre takto delí a pre potreby odporúčame (Szor, 2006). V súčasnosti sa už používané metódy a ciele pri škodlivom kóde značne prelínajú, a preto budeme používať iba pojem škodlivý kód alebo malware. V tejto kapitole si povieme na úvod povieme historické súvislosti vývoja škodlivého kódu a používanej škodlivej činnosti. Následne je pre účely našej analýzy potrebné sa oboznámiť s technikami, ktoré škodlivý kód pri interakcii so systémom a pamäťou najčastejšie využíva. Tieto techniky rozdelíme na:

- základné techniky, ktoré sa pokúšajú ukryť malware pred odhalením užívateľom, expertom alebo detekčným systémom,
- pokročilé techniky rootkitov na ukrývanie pred antvírusovými a anti-rootkit skenermi.

3.1 História vývoja škodlivého kódu

Prvý historicky zaznamenaný vírus bol Creeper, ktorý bol zistený na Arpanet-e (predchodca internetu) začiatkom roka 1970. Creeper bol experimentálne seba-kopírujúci program, ktorý napísal Bob Thomas z BBN v roku 1971. Po získaní prístupu cez ARPANET sa kopíroval na vzdialený systém, kde sa zobrazila správa: "Som Creeper, chyť ma, ak môžeš!" Ale prvý voľne šírený vírus pre osobné počítače IBM PC bol bootovací vírus (napádal boot sektory diskov) nazvaný Brain, ktorý vytvorili v roku 1986 bratia Basit a Amjad Farooq Alvi z Pakistanu údajne na ochranu vlastného programu proti softvérovému pirátstvu.
V tom istom roku boli zaznamenané ešte ďalšie dva vírusy (BURGER a CHARLIE). Od roku 1988 začal byť problém vírusov braný vážne a o dva roky neskôr už existovalo mnoho firiem (v súčasnosti známych), ktoré sa začali zaoberať antivírusovou ochranou. Nárast počtu a technológií je ukázaný na obrázku č. 14. Podľa údajov S&S International (Dr. Solomon’s antivírus) k 1. 2. 1998 bolo už známych 22860, čo bol nárast oproti predošlému roku o 6134.

[8] Slovník pojmov z web http://dictionary.reference.com/browse/malware

V roku 1990 sa tiež objavujú prvé polymorfné vírusy a "prvý vírusový toolkit" (nástroj na masovú tvorbu vírusov). V roku 1991 sa objavil nový boot vírus Michelangelo, ktorý postihol celý svet a vďaka médiam sa o vírusovej problematike dozvedela široká verejnosť (Kovac, 2004).

Obrázok 14 Vývoj počtu a typov vírusov do roku 1995 (Odehnal, 1996)

Postupne sa menil aj cieľ vývoja malware. V počiatočnom období tvorcovia malware implementovali činnosť, ktorou demonštrovali svoju prevahu a schopnosti v oblasti výpočtovej techniky. Išlo o rôzne hlášky pri spustení programov, vymazanie údajov atď. V súčasnosti sa situácia značne zmenila a aká činnosť sa vykonáva, závisí od typu malware, ktorý infikuje systém. Motívom útokov býva hlavne finančný zisk (krádeže prístupov k účtom, čísla kreditných kariet, informácie, ktoré je možné predať, ako mail adresy, levely z hier, atď.). Ďalej býva motívom špionáž (obchodná, tajné služby, ekonomická, priemyselná, zber vedecky zaujímavých informácií a nových objavov), demonštrovania nesúhlasného postoja (DOS útoky, zmeny obsahu stránok, napadnutie a kompromitácia firiem, ministerstiev, úradov...).
Ako príklad pre špionáž alebo zber priemyselných, vedecky zaujímavých informácií môžeme uviesť prípad známej APT skupiny, ktorá podľa správy bezpečnostnej firmy Mandiant[9] odcudzila stovky terabytov dát z minimálne 141 organizácií. K vykonávaniu škodlivej činnosti využíva malware množstvo rôznych techník a postupov. Ich spektrum je široké, dokonca každý malware pre tu istú škodlivú činnosť môže využiť inú techniku. Závisí to od kreativity a schopností autora malware. Z tohto dôvodu nebudeme podrobne rozoberať techniky z tejto oblasti. O zistených najčastejšie používaných technikách a typoch útokov napr. pri APT skupine firma Mandiant vydala správu, kde ich podrobne popisuje[10]. Prehľad najčastejších aktivít vykonávaných na napadnutom systéme je možné nájsť aj

[9] Podrobnejšie informácie sú dostupné na stránke http://intelreport.mandiant.com

[10] Kompletnú správu je možné stiahnúť z pomocou tohto odkazu http://intelreport.mandiant.com/Mandiant_APT1_Report_Appendix.zip

v prílohe práce (Príloha A). Ďalšou z rozšírených činností je zber informácií zameraný na finančné účty. Známy je Zbot malware (patrí medzi bot malware, inak nazývaný aj Zeus), ktorý zhromažďuje údaje o bankových účtoch ako prihlasovacie meno, heslo, jednorázový prihlasovací kód, ktorý sa využíva pri dvoj-faktorovej autentifikácii.
S technikami pre vykonanie škodlivej činnosti malware využíva aj sofistikované techniky pre schovávanie svojej činnosti. Týmto technikám sa budeme venovať v nasledujúcej časti.

3.2 Techniky zabezpečujúce skrývanie prítomnosti a činnosti pred nástrojmi na detekciu a užívateľom

Pre efektívne vykonávanie škodlivej činnosti používa malware množstvo techník, ktoré mu umožňujú svoju činnosť ukrývať. Ako sa postupne vyvíjajú a zlepšujú techniky pre detekciu, tak sa stále vyvíjajú a inovujú aj techniky malware.
Techniky môžeme rozdeliť na:

- techniky malware zamerané na ochranu pred detekciou s využitím statickej analýzy,
- techniky zamerané na ochranu pred detekciou s využitím dynamickej analýzy,
- techniky pre schovávanie pred skenermi a antivírusovými nástrojmi na detekciu.

Na získanie predstavy si môžeme ukázať štandardnú činnosť napr. v predošlej časti spomenutého Zbot malware (Zeus), ktorá vo veľkej miere korešponduje aj s činnosťou ostatných malware typu bot. Typická činnosť bot malware môže byť nasledujúca:

- po stiahnutí a spustení v počítači (býva ako príloha v mailoch alebo ako stiahnuteľný súbor na stránkach v internete) sa postupne rozbalí (po etapách),
- spustí sa v systéme buď pomocou vlastného implementovaného spúšťača (loader) (sofistikovaný malware) alebo štandardného loadera OS,
- nájde vhodnú spustenú aplikáciu alebo systémový proces (digitálne podpísaný a legitímny),
- pomocou techník pre injektovanie kódu sa injektuje do vybranej aplikácie ako DLL knižnica,
- nakoniec vytvorí spojenie (backdoor), pomocou ktorého je možné zadávať botu diaľkovo inštrukcie.

Podrobnejšie o technikách používaných Zeus botom je možné nájsť (Binsalleeh et al., 2010). Ako ukážka ozaj sofistikovaných techník môže poslúžiť analýza Stuxnet malware od firmy Symantec[11] alebo jeho praktická analýza z MNIN Security Blogu[12].

3.2.1 Ochranné techniky pred detekciou pri statickej analýze

Pri statickej analýze ide o detekciu bez spustenia programu s použitím disassemblerov, debugerov a ďalších nástrojov pre analýzu programu. Medzi základné techniky malware na ochranu pred detekciou patrí:

- polymorfizmus,
- metamorfizmus,
- šifrovanie,
- pakery,
- anti-debug a anti-disassemblovacie techniky.

V tejto práci pre rozsiahlosť problematiky popíšeme iba polymorfizmus a metamorfizmus, pre podrobnejšie informácie ohľadom techník na ochranu pred statickou analýzou odkazujeme čitateľov na hodnotné zdroje, kde je problematika dostatočne popísana. Podrobný popis metamorfných a polymorfných techník nájdeme v (Szor, 2006), šifrovaniu a pakerom sa venujú v (Sikorski, 2012; Ferrie, 2008) a podrobný popis anti-debug a anti-disassemblovacích techník je v (Shields, 2008; Zvara, 2014).

Polymorfný (Polymorphic) vírus

Škodlivý kód pri svojej tvorbe postupne začal používať techniky, ktoré mýlili existujúce detektory. Jeden z prvých spôsobov, ktorý sa objavil, boli zašifrované vírusy (Szor, 2006). Tieto typy vírusov obsahujú jednoduchší alebo zložitejší dešifrátor (dekryptor) a ich telo je zašifrované. Pre šifrovanie je možné použiť jednoduchšie algoritmy ako substitúcia, xorovanie, ale niekedy sa používajú aj silné algoritmy napr. IDEA. Detekcia sa pri týchto vírusoch zamerala priamo na detekciu dešifrátora, čím sa odhalilo mnoho šifrovaných vírusov. Preto ako ďalší krok vo vývoji boli oligomorfné vírusy, ktoré obsahovali viac dešifrátorov (aj niekoľko desiatok), z ktorých sa náhodne vždy vybral iba jeden. Ďalší stupeň vo vývoji

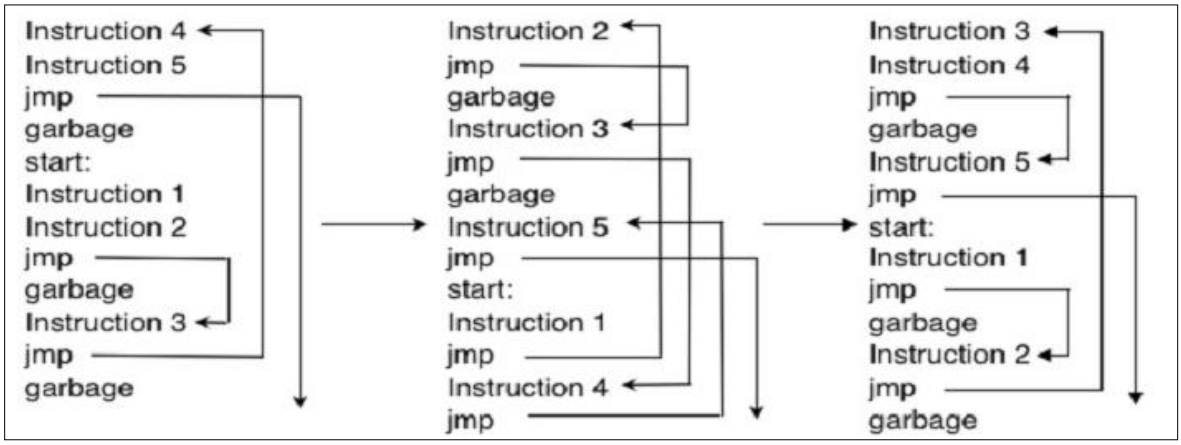
[11] Analýzu je možné stiahnúť pomocou odkazu http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

[12] Michael Ligh spoluautor knihy Malware Analyst's Cookbook (Ligh, 2010) a vývojár volatility frameworku vykonal podrobnú analyzu stuxnet malware ktorá je dostupná na stránke http://mnin.blogspot.sk/2011/06/examining-stuxnets-footprint-in-memory.html

zložitosti predstavujú polymorfné vírusy. Tie dokážu vytvárať mnoho inštancií dešifrátora, ktoré ešte môžu mať veľké množstvo (rádovo aj milióny) rôznych foriem. Aby sa tejto rozmanitosti docielilo, vkladajú sa do binárneho kódu náhodne nič nerobiace inštrukcie, ktoré neovplyvňujú výsledok výpočtu alebo dátové oblasti, ktoré sa mixujú s binárnym kódom. Aj tieto druhy vírusov je však možné napríklad pomocou emulátora odhaliť, keďže ich telo je vždy konštantné. Preto vznikli metamorfné vírusy, pri ktorých nie je potrebné šifrovanie, pretože celé ich telo je polymorfné (odlišná jedna kópia od druhej).

Metamorfný (Metamorphic) vírus

Podstatu metamorfného vírusu vysvetlil Igor Muttik[13] v jednej krátkej vete: „Metamorfizmus je polymorfizmus tela." (Szor, 2006), tzn. že telo škodlivého kódu sa mení pri každej novej kópii. To je docielené jeho možnosťou sám sa preprogramovať. Používa techniku zmätenia inštrukcií v kóde programu, aby zmiatol a znemožnil hlbšiu statickú analýzu a tiež poráža dynamické analyzátory tým, že zmení svoje správanie. Docieli sa to prekladom seba do dočasného súboru, úpravou dočasnej reprezentácie seba, a potom znova zapísaním späť do vlastného kódu (pozri obr. 15). Tento proces vykonáva vírus sám o sebe. Metamorfné vírusy používajú niekoľko rôznych metamorfných transformácií, ako zmena poradia inštrukcie, preusporiadanie dát, inlining a outlining, premenovanie registrov, permutácia kódu, expanzia kódu, zmenšenie veľkosti kódu, premiestňovanie podprogramu a vkladanie zbytočného kódu. Zmenený kód sa potom prekompiluje a vytvorí spustiteľný súbor, ktorý sa zásadne líši od originálu (Daoud, 2008).



Obrázok 15 Príklad vloženia skokov do kódu vírusu (Zperm virus) (Szor, 1998)

[13] Bezpečnostný expert, spoluzakladateľ Anti-Malware Testing Standards Organization (AMTSO) a v súčasnosti vedúcí architekt pre výskumný vo firme McAfee

V ďalšom spomenieme niektoré techniky malware na schovávanie svojej prítomnosti a aktivít pred detekciou s využitím dynamickej analýzy.

3.2.2 Ochranné techniky pred detekciou pri dynamickej analýze

Ako sme spomínali v úvode práce, podľa správy údajov antivírusovej firmy Symantec za rok 2012[14] bolo vytvorených v roku 2011 až 400 miliónov nových variantov malware, to je asi 33 miliónov variant za mesiac a asi milón nových variant za deň. Samozrejme také množstvo malware nie je možné spracovať a otestovať manuálne. Ako jediná možnosť je využitie automatizovaných nástrojov. Automatizované nástroje využívajú k detekcii dynamickú analýzu kódu, kde spustia testovanú vzorku v izolovanom prostredí (sandbox, virtuálne prostredie) a zaznamenávajú aktivity, ktoré vykonáva. Medzi také systémy patria cuckoo[15] a Anubis[16], viac ucelených informácií o dynamickej analýze je možné nájsť v prehľade (Egele, 2012).
Automatizované nástroje teda využívajú sandboxy alebo virtuálne prostredia. Ak sa malware podarí obísť automatizovanú analýzu, aj pri jeho veľkom rozšírení nebude detegovaný antivírusovými nástrojmi. Toto je dosť silný motív pre tvorcov malware, aby vyvíjali techniky, ktoré tieto nástroje pomýlia.
Dlhú dobu malware skúšal zistiť, či nie je spustený vo virtuálnom prostredí a ak áno, zmenil svoje správanie (nerobil nič škodlivé, vypol sa, atď.). Medzi základné postupy pre zistenie prostredia patrili tieto:
- kontrola vybraných registrov,
- kontrola ovládačov pre video a myš,
- kontrola bežiacich systémových služieb (servisov),
- spustenie špeciálneho assemblerovského kódu,
- kontrola konunikačného portu,
- kontrola vybraných názvov bežiacich procesov.

V súčasnosti sa objavujú viac logické prístupy, lebo virtuálne prostredie ako také sa začína používať aj bežnými užívateľmi (nie iba na testovanie), a tak po neutralizácii malwaru pri zistení, že je spustený vo virtuálnom prostredí, by pri súčasnom rozmachu používania virtuálnych platforiem prišiel o množstvo potencionálnych obetí. Ide hlavne o nasledovné oblasti.

[14] Správu je možné stiahnúť zo stránky firmy Symantec http://www.symantec.com/threatreport/

[15] Podrobnejšie informácie o projekte sú dostupné cez odkaz http://www.cuckoosandbox.org/

[16] Informácie o projekte je možné získať na stránke https://anubis.iseclab.org/

Interakcia s užívateľom: Malware pred vykonaním činnosti čaká na prejav ľudskej aktivity. Trojan UpClicker objavený v roku 2012 začal napr. komunikovať s C&C serverom až po tom, ako detegoval ľavé kliknutie myšou. Použitie techniky vidieť v nasledujúcom kóde (pozri obr. č. 16).

```
push    0               ; dwThreadId
push    0               ; lpModuleName
call    ds:GetModuleHandleA
push    eax             ; hmod
push    offset _main_function ; lpfn
push    WH_MOUSE_LL     ; idHook
call    ds:SetWindowsHookExA
```

Obrázok 16 Malware používajúci kontrolu prostredia pomocou myši

API funkcia SetWindowsHookExA s parametrom WH_MOUSE_LL nainštaluje procedúru, ktorá monitoruje nízkoúrovňové vstupy a výstupy od myši. Keď príde malwaru správa od myši, že sa posunula alebo bolo stlačené tlačítko, malware sa spustí. Ľudia väčšinou pri používaní Windows myš používajú, a preto malware pracuje správne. Ale automatizované nástroje myš nepoužívajú, a tak sa malware nespustí a nástroj ho nedetekuje ako škodlivý.

Nastavenie sandboxu: Sandboxy aj keď simulujú reálny počítač a vytvárajú izolované prostredie, sú konfigurované pomocou parametrov. Mnoho analyzátorov monitoruje súbory po spustení iba niekoľko minút. Potom prechádzajú na testovanie ďalšieho súboru. Malware takto stačí „počkať", kým skončí fáza monitorovania.

```
thread_network_tasks proc near          ; DATA XREF: function_Launch+7F↓o
                push    ebx
                mov     ebx, eax
                push    300000          ; dwMilliseconds
                call    Sleep
                mov     eax, ebx
                call    DecryptCode___m

loop:                                   ; CODE XREF: thread_network_tasks+28↓j
                push    1200000         ; dwMilliseconds
                call    Sleep
                call    ModifyRegistry___m
                call    network_main
                jmp     short loop
thread_network_tasks endp
```

Obrázok 17 Malware používajúci volanie Sleep na obídenie dynamickej analýzy

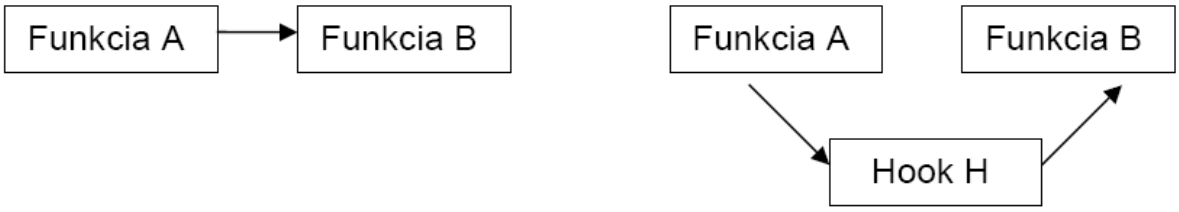
Po spustení malware čaká 300 000 milisekúnd (5 minút), kým sa spustí DecryptCode funkcia (pozri obr. č. 17). Čaká 20 minút, a potom sa spustí funkcia ModifyRegistry, po spustení funkcie Network_main sa čaká dalších 20 minút. Automatizované nástroje bežia iba krátky časový úsek, preto aktivity takého malware nezaregistrujú.

Ďalšiu samostatnú časť tvoria techniky určené na skrývanie pred skenermi a antivírusovými nástrojmi na detekciu.

3.2.3 Techniky pre schovávanie pred skenermi a antivírusovými nástrojmi na detekciu

Tieto techniky sú využívané hlavne rootkitmi, ale v súčasnosti niektoré z nich preberajú aj ďalšie typy malware. Cieľom techník je zabezpečiť neviditeľnosť seba a aktivít, ktoré sa v systéme vykonajú, čo spôsobí, že nástroje na detekciu malware neregistrujú jeho prítomnosť a nevykonajú analýzu. Ak sa analýza nevykoná, nie je možné zistiť, že systém je napadnutý. Medzi základné techniky z tejto oblasti patrí:
- Dll injekcia,
- hook systémových funkcií alebo tabuliek.

Pri hooku sa presmeruje volanie požadovanej funkcie na útočníkom podsunutú funkciu, ktorá vykoná škodlivú aktivitu. Aby sa presmerovanie neprezradilo, zavolá pôvodnú funkciu, avšak môže jej zmeniť vstup (pozri obr. č. 18).



Obrázok 18 Spracovanie normálnej a hookovanej funkcie

Podľa úrovne, kde sa hook vykoná, môžeme mať hooknuté objekty v užívateľskom priestore a v priestore jadra.
Pre úplnosť si ukážeme aj príklad dvoch starších techník, ktoré sa ale už v súčasnosti nepoužívajú pre ich možnosť odhalenia pomocou detektorov testujúcich integritu súborov. Boli však výrazným medzníkom pre techniky schovávania sa v systéme. Ide o knižničné a systémové vytvorenie zadných vrátok (backdoor), ktoré bez spozorovania umožnili prístup do systému.
Podrobnejšie sa ale popisom starších techník používaných rootkitmi nebudeme zaoberať. Popis starších techník je možné nájsť v diplomovej práci (Balogh, 2008).

Knižničné zadné vrátka (Library backdoors)

Takmer každý systém využíva zdieľané knižnice. V nich použité funkcie sú využívané v mnohých programoch, čím sa značne znižuje ich veľkosť. Útočník môže zameniť niektorú z funkcií zdieľanej knižnice za vlastnú (alebo upraviť pôvodnú funkciu v knižnici) , napr. funkcia crypt.c a _crypt.c. Programy, ako napr. login,

používajú práve tieto funkcie na overenie hesla. Ak sa zadá špeciálne heslo, ktoré nastavil útočník vo svojej verzii crypt(), môže byť napr. povolený shell bez overenia skutočného hesla. V takomto prípade administrátor pri vykonávaní kontroly integrity súboru login nič podozrivé nezistí. Až kontrolou zdieľaných knižníc na narušenie integrity bolo možné dané útoky odhaľovať (Balogh, 2008).

Systémové zadné vrátka (Kernel Backdoors)

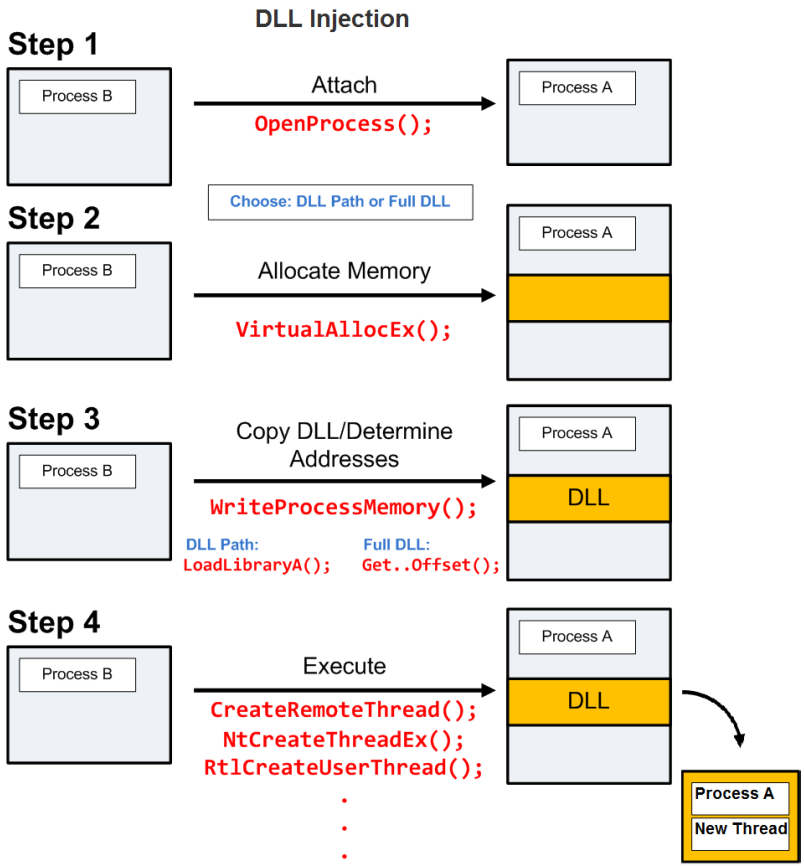
Rovnakú metódu, aká bola popísaná pri knižniciach, je možné aplikovať aj na úrovni jadra, čím sa obídu dokonca aj staticky linkované knižnice. Backdoor na úrovni jadra (ak je dobre spravený) je pre administrátora najťažšie zistiteľný.

DLL injekcia

DLL injekcia (DLL injection) je technika na spustenie kódu v adresnom priestore iného procesu (hosťovského), čím získame práva daného procesu. K tomu potrebujeme do procesu nahrať náš vlastný program, ktorý je vo forme DLL (Dynamic Link Library) dynamickej knižnice. Celý proces môžeme rozdeliť na štyri časti (pozri obr. č. 19):

- napojenie sa na hosťovský proces (attach),
- alokovanie pamäte v adresnom priestore hosťovského procesu (Allocate),
- kopírovanie DLL knižnice (alebo cesty k DLL, v závislosti od použitého postupu) do alokovanej časti pamäte,
- inštruovanie hosťovského procesu, aby spustil funkcie z našej knižnice.

Windows API ponúka mnoho funkcií, ktoré umožňujú napojenie na iný proces za účelom jeho debugovania. Tieto funkcie je možné využiť aj v prípade DLL injekcie. Každá zo štyroch častí môže byť implementovaná rôznymi spôsobmi, pričom každá má svoje výhody a aj nevýhody.
Štandardne je možné nahrať knižnicu pomocou API funkcie CreateRemoteThread> LoadLibrary, čo vykoná nahratie DLL knižnice do hosťovského procesu a spustí ho. Zároveň však DLL knižnicu registruje a začlení do všetkých troch obojsmerne linkovaných zoznamov modulov uložených v PEB (InLoadOrderModuleList, InMemoryOrderModuleList, InInitializationOrderModuleList), čím sa knižnica v systéme stáva viditeľnou pre bežné nástroje a systémové funcie. Tieto typy DLL injekcií sú preto pomerne ľahko detekovateľné antivírusovými nástrojmi, ktoré prechádzajú PEB DLL zoznamy a kontrolujú, či DLL knižnica nie je infikovaná. Preto autory malware hľadajú iné alternatívy. Jednou z nich je použitie knižnice Reflective DLL Injection.



Obrázok 19 Proces vytvorenia DLL injection (Antoniewicz, 2013)

Reflective DLL Injection[17] technika implementuje vlastný jednoduchý PE (Portable Executable) loader súborov a umožňuje, aby sa DLL knižnica načítala sama bez toho, aby použila loader operačného systému alebo LoadLibrary() API funkciu. Takým spôsobom sa minimalizuje interakcia s operačným systémom a injektovaným procesom. Nezanechajú sa v systéme skoro žiadne stopy umožňujúce detekciu prítomnosti DLL knižnice (DLL knižnica nie je registrovaná v žiadnom z troch PEB zoznamov pre moduly).

Inú techniku schovávania používa "user-mode rootkit'' NTIllusion (Kdm, 2004), ktorý injektuje svoju knižnicu do iného procesu, ale potom knižnicu zo zoznamov modulov (uložených v Process Environment Block (PEB) štruktúre) odlinkuje.

Ďalšou z možností je použitie API funkcií SetWindowsHookEx[18] alebo SetWinEventHook[19]. Aj keď sa pomocou týchto techník podarí schovať injektované

17 Zdrojové kódy sú dostupné na https://github.com/stephenfewer/ReflectiveDLLInjection

18 Spôsob implementácie v kóde je možné pozrieť na stránke http://resources.infosecinstitute.com/using-setwindowshookex-for-dll-injection-on-windows/

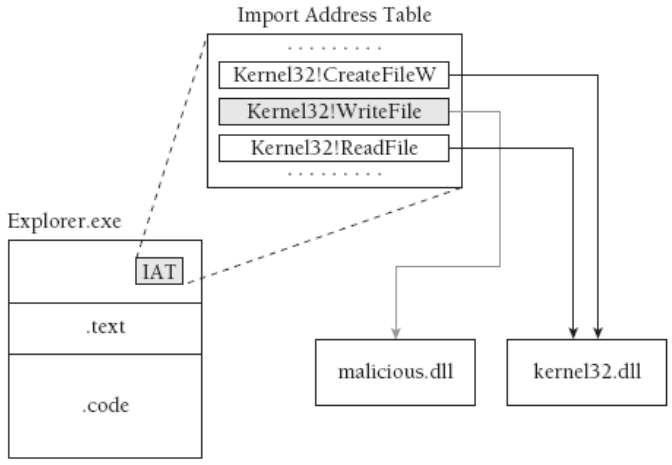
19 Popis funkcie nájdeme na stránke MS Windows http://msdn.microsoft.com/en-us/library/windows/desktop/dd373640(v=vs.85).aspx

DLL knižnice, nie sú úplne neviditeľné. Tieto metódy schovávania neovplyvňujú VAD strom operačného systému a napriek ich neprítomnosti na zoznamoch modulov môžu byť viditeľné ako namapované súbory v adresnom priestore zneužitého cieľového procesu (Dolan-Gavitt, 2007). Pri vytváraní hooku pomocou API funkcií SetWindowsHookEx alebo SetWinEventHook systém navyše cez user32.dll knižnicu volá ovládač win32k.sys, ktorý vytvára záznam v atom tabuľke s plnou cestou o umiestnení k injektovanej DLL knižnici.

IAT hook

Medzi základné hooky, ktoré sa vykonávajú v užívateľskom priestore, patrí hook tabuľky importov (IAT) užívateľského programu. Malware môže prepísať adresu v IAT tabuľke pre ľubovoľnú API alebo knižničnú funkciu na adresu svojej do systému vloženej funkcie (napr. z DLL injektovanej knižnice). Takto malware zabezpečí volanie svojej funkcie (s právami, ktoré má program, ktorý funkciu volá) vždy, keď program zavolá hooknutú funkciu. Z praktického hľadiska si preto malware vyberajú programy, ktoré majú systémové práva a snažia sa hooknuť často volané funkcie.

Na obr. č. 20 môžeme vidieť znázornenie IAT hooku. Ako napadnutý program sa využil explorer.exe, ktorý je vo Windows jedným zo základných a spúšťa sa so systémovými právami hneď po spustení operačného systému. V jeho IAT sú adresy (32-bitové pointre-, ak sme na 32-bitovej architektúre), ktoré ukazujú na ich umiestnenie v pamäti. V našom prípade na obrázku ide o funkcie exportované zo systémovej knižnice kernel32.dll. Adresa pre funkciu Kernel32!WriteFile je presmerovaná a ukazuje na funkciu z malwarom inštalovanej DLL knižnice. Pre detekciu IAT hooku môžeme využiť fakt, že adresa funkcií v IAT by mala byť v rozsahu adresného priestoru knižnice, ktorá danú funkciu exportuje. Ak je adresa mimo tento rozsah, ide zrejme o jej hooknutie a presmerovanie na funkciu z inej knižnice.



Obrázok 20 Zobrazenie princípu IAT hooku (Ligh, 2010)

Postup pre detekciu by mohol byť nasledujúci:

- Prehľadanie všetkých aktívnych procesov (napr. pomocou prehľadania zoznamu EPROCESS štruktúr).
- Prehľadanie všetkých DLL knižníc používaných aktívnymi procesmi a uloženie si ich mena, začiatku adresy v pamäti a veľkosť využívaného adresného priestoru (napr. pomocou načítania PEB alebo VAD informácií).
- Parsovaním PE hlavičky nájsť IAT tabuľku a adresy importovaných funkcií.
- Porovnať, či adresa každej importovanej funkcie je z rozsahu adresného priestoru knižnice, z ktorej je volaná.

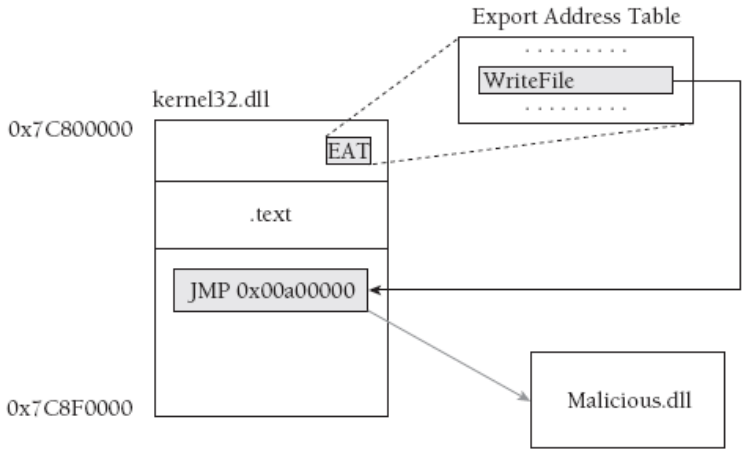
Inline API Hook

Zložitejšie, ale o to efektívnejšie, je inline hook API funkcií. Je to vo svete malware oveľa častejšie používaná technika ako IAT hook. Inline hook sa nazýva aj trampolína alebo detour hook.

V tomto prípade malware neprepisuje iba pointer na funkciu ako v IAT hooku, ale prepisuje niektoré inštrukcie v hooknutej funkcii, čím opäť zabezpečí presmerovanie na svoju funkciu. Na obr. č. 21 je ukázaný príklad inline hooku. Je vidieť, že systémová knižnica kernel32.dll používa rozsah pamäte od 0x7C80000 do 0x7C8F0000. Adresa exportovanej funkcie (WriteFile) v tabuľke exportov (EAT) ukazuje na legitímnu adresu vnútri adresného priestoru knižnice (na začiatok funkcie WriteFile). Ale na začiatku funkcie malware prepísal prológ funkcie (používa ho každá Windows funkcia) na inštrukciu skoku JMP 0x00a00000, čiže na miesto, kde sa nachádza škodlivá knižnica injektovaná do systému malwarom. Táto technika je náročná pre jej nutnosť disassemblovania cieľového programu, ale tým, že existuje dosť príkladov, ako ju implementovať, je veľmi rozšírená (podobnú techniku využíva aj napr. Microsoft Detours[20] alebo knižnica mhook[21]).

[20] Podrobnejšie informácie k projektu a technickej realizácii je možné nájsť na stránke http://research.microsoft.com/en-us/projects/detours/

[21] Zdrojové kódy sú dostupné cez odkaz https://github.com/martona/mhook



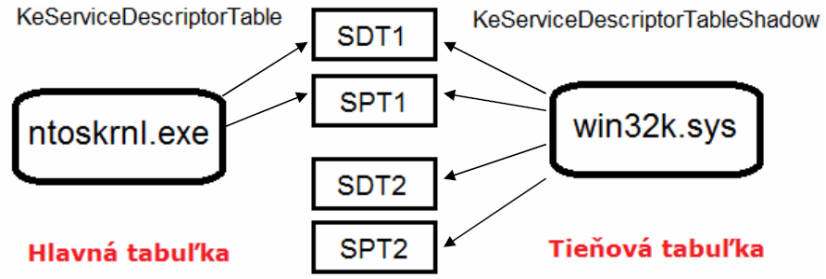
Obrázok 21 Zobrazenie princípu Inline IAT hooku (Ligh, 2010)

Pri detekcii inline hooku môžeme postupovať ako pri detekcii IAT hooku, ale preto, že v tomto prípade adresa hooknutej funkcie ukazuje do adresného priestoru pôvodnej knižnice, využijeme iba prvé tri kroky. Ďalší postup je nasledovný:

- Namiesto parsovania IAT tabuliek procesov budeme parsovať EAT tabuľku každej DLL knižnice, aby sme našli RVA (relative virtual address) exportovaných funkcií. Pripočítaním základnej adresy DLL knižnice k RVA dostaneme VA (virtuálnu adresu) funkcie.
- Presunieme sa na danú adresu v pamäti alebo spravíme dump pamäte so začiatkom na VA funkcie.
- Disassemblujeme prvé inštrukcie exportovanej funkcie. Ak tam nájdeme inštrukcie skoku (JMP) alebo volania funkcie (call), overíme, či adresa skoku je v rozsahu adresného priestoru knižnice.

Hook SSDT

Tabuľka systémových volaní (System Service Descriptor Table (SSDT)) je štruktúra, ktorá obsahuje 2 ďalšie štruktúry (pozri obr. č. 22):
– Service Dispatch Table (SDT – odbavovacia tabuľka),
– Service Parameter Table (SPT – tabuľka parametrov).



Obrázok 22 Náhľad a vzťahy oboch SSDT tabuliek vzhľadom na SDT a SPT

SDT je pole obsahujúce adresy funkcií jadra. SPT pre každú funkciu jadra (rutiny) eviduje počet parametrov.

Z jej obsahu vidieť, aká je SSDT štruktúra dôležitá. Preto je často cieľom rootkitov a malware, ktorý pre zabezpečenie vlastného spustenia presmeruje niektorú zo systémových služieb v SDT na svoj kód. Po spustení môžu vykonať skrytie procesu, súboru, kľúča a pod. Prvýkrát túto techniku použil rootkit NTRootkit-A, čím začal novú éru rootkitov, ktoré s cieľom „schovať sa“ používajú manipuláciu s objektami jadra OS.

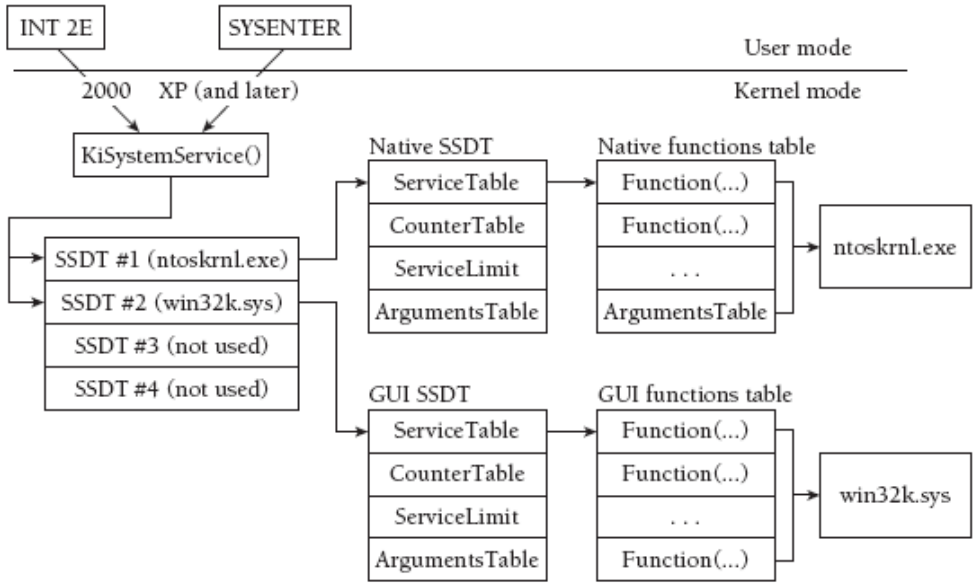
Na hook SSDT je potrebné poznať základnú adresu SSDT tabuľky (kde sa v pamäti nachádza) a index funkcie, ktorý chceme hooknuť. Malware k zisteniu základnej adresy používa rôzne techniky, veľmi často však volá funkciu MmGetSystemRoutineAddress (verzia funkcie GetProcAddress pre jadro) a nájde KeServiceDescriptorTable symbol, ktorý je exportovaný modulom ntoskrnl.exe.

Systém Windows obsahuje štandardne dve SSDT tabuľky. Jedna obsahuje funkcie jadra pre prácu s procesmi, vláknami, súbormi a ďalšími základnými objektmi Windows a nachádza sa v hlavnom module jadra. Druhá SSDT tabuľka (Shadow) slúži pre obsluhu grafického užívateľského rozhrania.

Pre detekciu hooknutia SSDT tabuľky stačí, ak sa zabezpečí pravidelné sledovanie obsahu tabuľky v prípade, ak máme k dispozícii na porovnanie pôvodnú SSDT tabuľku. Druhá možnosť vychádza z predpokladu, že všetky adresy musia ukazovať do adresného priestoru ntoskrnl.exe modulu jadra. Podobne všetky adresy GUI funkcií zo SSDT shadov tabuľky by mali ukazovať do adresného priestoru win32k.sys.

Mnoho bezpečnostých skenerov a anti-rootkit nástrojov podobný prístup využíva. SSDT tabuľka však môže byť spoločná, ale aj individuálna pre každé vlákno procesu. To znamená, že každé vlákno môže pristupovať k inej SSDT tabuľke v závislosti na hodnote, ktorú má nastavenú v premennej štruktúre ETHREAD.Tcb.ServiceTable (Tcb je odkaz na štruktúru KTHREAD). Štandardne táto hodnota ukazuje na SSDT tabuľku vytvorenú systémom, ale malware môže vytvoriť kópiu systémom vytvorenej SSDT tabuľky a zameniť adresu v štruktúre KTHREAD. Vlákno bude takto pristupovať k inej SSDT tabuľke, v ktorej budú hooknuté (presmerované) funkcie a bezpečnostný skener, ktorý sleduje hooky v systémovej SSDT (na ktorú

ukazuje symbol NT!KiServiceTable), nezistí žiadnu anomáliu. Táto technika bola prvýkrát implementovaná rootkitom Rustock v roku 2006 (pozri obrázok č. 23).



Obrázok 23 Štruktúra SSDT tabuľky (Ligh, 2010)

Ako príklad môžeme uviesť rootkit BlackEnergy 2, ktorý hookuje 14 rôznych SSDT funkcií, určených hlavne na kontrolu prístupu do registrov, procesov a virtuálnej pamäte. Rootkit nainštaluje do systému ovládač s názvom 00000B9D.sys, ktorý obsahuje funkcie, na ktoré sa v SSDT tabuľke po hooknutí ukazuje a ktoré budú spúšťané vláknami pred pôvodnou funkciou zo SSDT (pozri výpis č. 1).

```
$ python volatility.py ssdt -f be2.bin | egrep -v '(ntoskrnl|win32k)'
Gathering all referenced SSDTs from KTHREADs...
Finding appropriate address space for tables...
SSDT[0] at 814561b0 with 284 entries
Entry 0x0041: 0x81731487 (NtDeleteValueKey) owned by 00000B9D
Entry 0x0047: 0x8173116b (NtEnumerateKey) owned by 00000B9D
Entry 0x0049: 0x81731267 (NtEnumerateValueKey) owned by 00000B9D
Entry 0x0077: 0x817310c3 (NtOpenKey) owned by 00000B9D
Entry 0x007a: 0x81730e93 (NtOpenProcess) owned by 00000B9D
Entry 0x0080: 0x81730f0b (NtOpenThread) owned by 00000B9D
Entry 0x0089: 0x81731617 (NtProtectVirtualMemory) owned by 00000B9D
Entry 0x00ad: 0x81730da0 (NtQuerySystemInformation) owned by 00000B9D
Entry 0x00ba: 0x8173156b (NtReadVirtualMemory) owned by 00000B9D
Entry 0x00d5: 0x81731070 (NtSetContextThread) owned by 00000B9D
Entry 0x00f7: 0x81731397 (NtSetValueKey) owned by 00000B9D
Entry 0x00fe: 0x8173101d (NtSuspendThread) owned by 00000B9D
Entry 0x0102: 0x81730fca (NtTerminateThread) owned by 00000B9D
Entry 0x0115: 0x817315c1 (NtWriteVirtualMemory) owned by 00000B9D
```

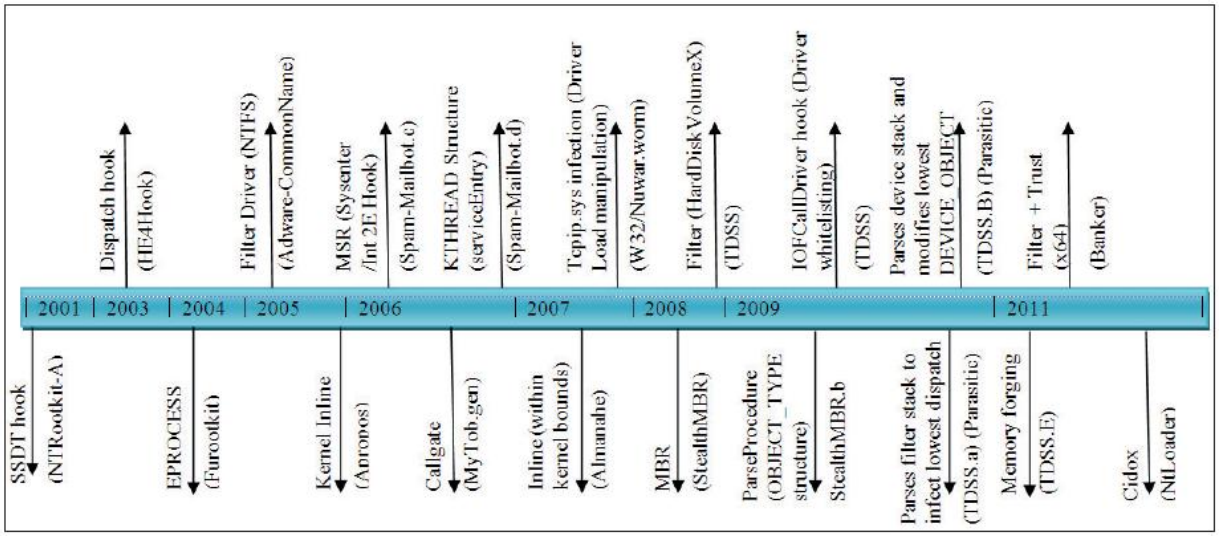
Výpis 1 Výpis hookov SSDT tabuľky pri BlackEnergy 2 rootkite

Riešenie pre detekciu je v tomto prípade podľa (Ligh, 2010) prechádzanie všetkými vláknami v systéme a vytvorenie zoznamu SSDT tabuliek, ktorých adresa bola uložená ako hodnota štruktúry ETHREAD.Tcb.ServiceTable. Následne overenie všetkých zistených SSDT tabuliek.

Rootkit môže podobným spôsobom okrem SDT tabuliek hookovať aj iné systémové tabuľky a objekty. Napr. tabuľku IRP funkcií pre ovládače, IDT tabuľku (tabuľka prerušení) atď. Podrobnejší prehľad o technikách je možné nájsť v (Ligh, 2010; Ivaniš, 2011).

3.3 Pokročilé techniky malware

Tvorcovia malware (hlavne rootkitov) sa snažia vyvíjať stále nové techniky a zabezpečiť sa tak pred odhalením. V tejto časti si povieme o niektorých nových technikách a vývoji v tejto oblasti. Prehľad celkového trendu vo vývoji rootkitov je možné vidieť na obr. č. 24. My si prejdeme iba niektoré techniky pre ich náročnosť, podrobnejšie sa s nimi môžeme oboznámiť z dostupných článkov, napr. (Kapoor, 2011).



Obrázok 24 Prehľad pokročilých rootkit techník (Kapoor, 2011)

Podľa štatistík od McFee, Symantec, Microsoft 10% z celkovo detegovaných malware používa rootkit techniky. Od roku 2001 rootkity začali využívať na schovávanie pred detekciou techniky manipulácie s objektmi a funkciami jadra (Kapoor, 2011). NTRootkit-A hookoval SSDT tabuľku (pozri popis v predchádzajúcej časti), a tak ovplyvňoval výstup, ktorý volané funkcie vracali. Rootkit HE4Hook rozšíril túto ideu a hookoval SDT (dispatch) tabuľku pre NTFS súborový systém, a tak

kontroloval prístupy na ntfs súborové systémy (Davis, 2009). Táto technika sa stala medzi rootkitmi veľmi používaná. Pri schovávaní procesov rootkity začali používať techniku odlinkovania procesu zo zreťazeného zoznamu v štruktúre EPROCESS. Táto technika sa prvýkrát objavila v roku 2004 pri Fu rootkite. V roku 2005 začali rootkity na ochranu svojich súborov využívať NTFS filter ovládač (filter driver), čo značne skomplikovalo situáciu AV spoločnostiam, lebo tento ovládač používali väčšinou aj oni k detekcii a obnove napadnutých súborov.
Pre svoju činnosť si malware často registrujú systémové časovače (timers). Ovládač malware môže vytvoriť timer jednoducho kvôli notifikácii po stanovenom časovom úseku. Timer funguje podobne ako funkcia sleep() s tým rozdielom, že vlákno sa na daný čas neuspí, ale môže vykonávať ďalšiu činnosť. Timer sa navyše po vypršaní času môže resetovať a začať nový cyklus. Tým môže malware zabezpečiť svoju notifikáciu periodicky, nie len jednorázovo. Môže tak napríklad kontrolovať DNS záznamy každých 5 minút alebo meniť hodnotu v registroch každé 2 sekundy. Na také úlohy je timer veľmi vhodný. Pri vytvorení timera môžeme zároveň definovať Deferred procedure calls (DPCs), čiže určitú funkciu (callback), ktorá sa vykoná, keď timer exspiruje. Malware môže takto napr. pravidelne kontrolovať update na C&C servery a ak je zmena, aktualizovať sa alebo v pravidelných intervaloch testovať prítomnosť nových počítačov v sieti.
Ďalším často využívaným komponentom OS je mutex. Mutex môže použiť malware pre štandardnú synchronizáciu vlákien, ale často sa využíva aj k zabezpečeniu spustenia iba jednej kópie, aby sa predišlo opakovanej infekcii už napadnutého systému. Malware môže vytvoriť globálny mutant s nejakým názvom alebo nejakým vybraným reťazcom. Ak sa druhá inštancia aplikácie pokúsi vytvoriť rovnaký mutant, systém mu to nedovolí a aplikácia sa zatvorí. Mutex takto môže byť jednoduchou metódou na overenie, či už v systéme beží inštancia malware. Ako príklad môžeme uviesť, že v (Rieck, 2008) identifikovali pri rodine vírusov Worm.Gobot vytváranie hlavne mutexov s názvom „GhostBOT-0.57a", „GhostBOT-0.57" a „GhostBOT". Pre malware Poison Ivy je napr. typické, že vytvára mutex „)!VoqA.I4"[22].

Z ďalších techník, ktoré rootkity začali používať, by sme spomenuli techniku zámeny obsahu pamäte (Memory forging), ktorú použil v roku 2011 rootkit TDSS.E (Mathur, 2011). Toto bol v podstate prvý reálny rootkit, ktorý použil zámenu obsahu pamäte. Pred ním bolo už popísané proof of concept riešenie (Butler, 2005), ale reálne v praxi použité nebolo. Dôvod využitia techniky je nevyhnutnosť anti-rootkit skenerov pri detekcii rootkitov používať sledovanie obsahu pamäte. Zámena obsahu pamäte ale nie je jednoduchý proces, lebo anti-rootkit skenery môžu k prístupu do pamäte

[22] Podrobnejšiu analýzu je možné nájsť na stránke http://pmelson.blogspot.sk/2012_10_01_archive.html

využiť rôzne metódy (pozri časť 2.1 Metódy čítania a prístupu do operačnej pamäte). Rootkit TDSS.E hookuje dispatch tabuľku, a preto sa snaží pri pokuse o čítanie obsahu tejto tabuľky anti-rootkit skenerom presmerovať požiadavku na oblasť pamäte, kde sa nachádza pôvodná verzia. Takto skener nezistí, že tabuľka bola hooknutá. K tomu rootkit používa hooknutie KiDebugRoutine, ktoré môže sledovať všetky udalosti v systéme (Skape, 2007). Potom nastaví hardvérový breakpoint na monitorovanie prístupu do oblasti pamäte (na miesto, kde sa nachádza dispatch tabuľka). Kedykoľvek príde systému požiadavka na čítanie pamäte na adrese, kde sa nachádza dispatch tabuľka, hardvérový breakpoint sa spustí a vyvolá výnimku, ktorá sa zachytí v KiDebugRoutine. V hooknutej KiDebugRoutine funkcii rootkit overí, či išlo o vyvolanie výnimky jeho vytvoreným breakpointom a aký proces sa pokúša o prístup. Ak ide o bežný proces, výnimku označí za spracovanú a odovzdá riadenie späť procesu (v prípade inej výnimky zavolá pôvodnú funkciu KiDebugRoutine, ktorá výnimku spracuje štandardným spôsobom). Ak ide o prístup anti-rootkit skenerom, rootkit prepíše obsah ESI registra (predpokladá sa, že cieľová adresa požiadavky na prístup do pamäte sa nachádza práve tam). Potom vráti riadenie naspäť systému, ktorý pristúpi na adresu v pamäti zadanú v registry ESI a vráti obsah. Skener ale predpokladá, že dostal obsah pamäte z adresy, kde sa nachádza aktuálna dispatch tabuľka a žiadnu hrozbu nedeteguje.

Proti týmto technikám je možné brániť sa podrobným sledovaním objektov jadra, ak máme možnosť prístupu do jadra, alebo skenovaním pamäte s využitím prístupov do pamäte popísaných v kapitole našej vlastnej implementácie metódy pre prístup do pamäte (časť 2.4).

Možnosť prístupu do jadra po napadnutí rootkitom môže byť problematická. Rootkity k svojej ochrane používajú niekedy agresívnejšie postupy a obmedzujú alebo úplne blokujú činnosť AV detektorov. Rootkit si môže registrovať notifikačnú callback funkciu, ako napr. PsSetLoadImageNotifyRoutine. Pri každom spúšťaní nového procesu je rootkit notifikovaný systémom a kontroluje, či sa spúšťaný proces nenachádza na zozname nepovolených procesov (procesy, ktoré vytvárajú AV nástroje). Ak nie, povolí sa spustenie procesu, ak áno, rootkit zapíše inštrukciu návratu (0xc3) na miesto vstupného bodu (entry pointu) pre daný proces v pamäti. Takto sa proces ihneď po spustení ukončí. Techniku použil prvýkrát rootkit Nuwar back v roku 2007.

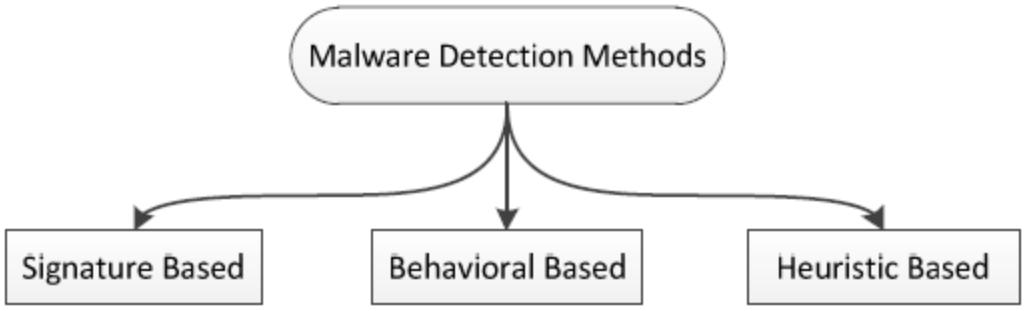
Callback funkciu môže použiť aj na injektovanie vlastnej DLL knižnice do každého nového spúšťajúceho procesu (Mebroot, BlackEnergy, Rustock) alebo na zachytenie a reakciu na rôzne systémové udalosti, ako je pád systému BSOD (Rustoc.C rootkit sa vymaže z pamäte pred vytvorením crach dumpu pamäte systémom, Sinowal rootkit overí, či je MBR sektor na disku stále infikovaný predtým, ako sa systém pri BSOD vypne…).

Ešte prepracovanejšia technika blokovania AV nástrojov[23] sa objavila v roku 2011 v Brazilian Banker rootkite. Technika využíva kombináciu registrácie callback funkcie (psSetLoadImageNotifyRoutine) a boot filter ovládača. Tým kontroluje nahrávanie všetkých ovládačov a komponentov do systému. Pri pokuse o nahratie ovládača alebo iného systémového komponentu kontroluje jeho parametre. Na základe toho rozhodne, či povolí jeho nahratie do systému alebo nie. Ak sa nahratie ovládača nepovolí, zapíše inštrukciu návratu na miesto entry pointu ovládača (alebo aj nejaký error kód). Takýmto spôsobom má rootkit úplnú kontrolu nad tým, čo sa v systéme spúšťa a činnosť AV nástrojov môže byť úplne blokovaná.
K ďalším pokročilým technikám rootkitov ako inštalácia do súborov nachádzajúcich sa mimo dosahu OS, náhodné kopírovanie do systémových ovládačov (techniky na zabezpečenie spúšťania po reštarte) alebo techniky MBR rootkitov sa v tejto práci z dôvodu ich náročnosti a presahovania rámca našej práce venovať nebudeme. Pri záujme o tieto techniky odporúčam literatúru (Kapoor, 2011; Blunden, 2013).

[23] Podrobný popis techniky je popísaný v blogu od V. Zakorzhevskeho s názvom An unlikely couple: 64-bit rootkit and rogue AV for MacOS na stránke http://www.securelist.com/en/blog/473/An_unlikely_couple_64_bit_rootkit_and_rogue_AV_for_MacOS.

4 Metódy detekcie

Metódy pre detekciu malware môžeme kategorizovať rôznym spôsobom v závislosti od uhla skúmania. V práci rozdelíme metódy detekcie na tri kategórie (pozri obr. 25); na metódy založené na porovnávaní vzorov (signature based), na metódy detekcie na základe správania sa a na heuristické metódy detekcie. Pri každej metóde môžeme na získanie potrebných informácií využiť statickú alebo dynamickú analýzu kódu. Statická analýza využíva informácie v podozrivých spustiteľných programoch bez toho, aby boli spustené, na základe kontroly binárneho kódu a čítania priamo v assemblery. Dynamická analýza sleduje aktivity softvéru po načítaní do pamäte. Dynamický prístup venuje viac pozornosti na aktivity, ale je ťažké vopred zabezpečiť, aby sa predišlo poškodeniu, pretože vírus, ktorý testujeme, je "spustený". Preto sa najčastejšie používa spôsob testovania v izolovanom prostredí.



Obrázok 25 Rozdelenie metód detekcie malware (Bazrafshan et al., 2013)

4.1 Metódy detekcie založené na signatúre (Signature-based methods)

V súčasnosti porovnávanie podľa vzorov (pattern matching) je jedna z najbežnejších metód pri detekcii malware a detekcia založená na signatúrach je jedna z najpopulárnejších metód (Gutmann, 2007). Signatúra je jedinečná charakteristika (črta) pre každý súbor, niečo ako jeho odtlačok (fingerprint). Metódy založené na signatúre využívajú jedinečné vzory extrahované z rôznych škodlivých kódov, aby ich dokázali detegovať. Tieto metódy sú viac účinné a rýchlejšie ako ostatné metódy. Extrahované vzory garantujú kvôli zabezpečeniu ich jedinečnosti pri výbere malú chybovosť pri detekcii. Z tohto dôvodu sú ešte stále tieto metódy najčastešie využívané v komerčných antivírusových nástrojoch (Gutmann, 2007.).
Tieto metódy však nie sú schopné detegovať neznáme varianty malware a tiež vyžadujú značné manuálne úsilie a čas pre extrakciu jedinečných signatúr.

Príchodom polymorfných a metamorfných malware sa detekcia pomocou signatúr stala ešte náročnejšia a v mnohých prípadoch neúnosná.

4.2 Metódy detekcie založené na detekcii správania (Behavior-based methods)

Techniky pre detekciu malware na základe správania sledujú správanie programu, na základe čoho sa rozhodujú, či je alebo nie je škodlivý. Tým, že tieto metódy sledujú, čo program robí, neprejavujú sa u nich tie možnosti prekonania detekcie ako u metód založených na signatúrach. Pri týchto metódach môžeme detegovať programy s rovnakým správaním. Už pri detekcii jednoduchého vektoru správania môžeme identifikovať rôzne vzorky malware. Tieto metódy môžu byť značne nápomocné aj pri detekcii metamorfných malware, ktoré mutujú a menia svoj kód pri každej kópii tým, že aj pri zmene naďalej využívajú tie isté systémové zdroje a služby k dosiahnutiu svojej činnosti.

Detektor založený na detekcii správania pozostáva z nasledovných komponentov (Jacob, 2008):

- Zberač dát (senzor) - má za úlohu zber dynamických a statických informácií o spustenom alebo spúšťanom procese.
- Interpreter - konvertuje dáta a informácie zo senzorov do požadovaného formátu.
- Porovnávač - slúži na porovnávanie sformátovaných informácií so známymi (signatúrami) vektormi správania.

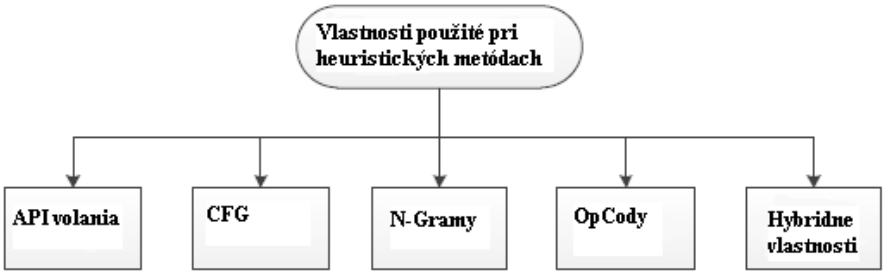
Príkladom pre detekciu založenú na správaní môže byť technika detekcie na základe histogramu patentovaná firmou Symantec[24] .
Medzi obmedzenia týchto metód detekcie patrí dĺžka času potrebná pre skenovanie a vysoká úroveň falošne (chyba typu I) pozitívnych výsledkov (malware vyhodnotí systém ako neškodný program) (Elhadi, 2012).

4.3 Metódy detekcie založené na heuristike (Heuristic-based methods)

Ako bolo spomenuté v predošlej časti, popísané metódy majú svoje nevýhody. Tieto nevýhody chcú eliminovať metódy pre heuristickú detekciu. Pri heuristickej detekcii sa využívajú data mining techniky a metódy strojového učenia na zistenie správania

[24] Technika sa spomína v internetovom príspevku "Introduction to Malware", na stránke "http://securityresearch.in/index.php/projects/malware_lab/introduction-to-malware/8/"

sa malware. Pre správnu klasifikáciu potrebujú vlastnosti, ktoré reprezentujú vstupný objekt a ktoré sú hlavne vhodné pre klasifikáciu. Úspešnosť metód strojového učenia závisí hlavne od výberu vhodných vlastností vstupných objektov. Pre detekciu malware bolo testovaných mnoho vlastností, ktoré boli získavané pomocou statickej alebo dynamickej analýzy. Prehľad vlastností, ktoré si v nasledovnej časti popíšeme, je vidieť na obr. č. 26.



Obrázok 26 Prehľad testovaných vlastností pri metódach heuristickej detekcie

API volania

Monitorovanie API systémových volaní je veľmi často používaná metóda. Mnoho data mining metód ich využíva pri nastavovaní klasifikátorov. Často sa tieto volania používajú ako ukazovatele správania sa programu.

Jednu z najdôležitejších prác v oblasti detekcie malware, ktorá využíva API sekvencie, bola práca (Sung et al., 2004). Vytvorili na signatúrach založený detekčný systém s názvom Static Analyzer of Vicious Executables (SAVE), ktorý porovnáva API sekvencie extrahované z programu so sekvenciami z databázy signatúr. Pre porovnanie použili kosínusovú mieru podobnosti, rozšírenú Jaccard mieru a Pearsonovú koreláciu. Finálny výsledok bol priemer všetkých troch meraní. Použitím týchto štatistických mier pre podobnosť ich program zachytával, aj polymorfné a metamorfné škodlivé kódy, pri ktorých klasické metódy detekcie na základe signatúr boli neúspešné.

V ďalšej práci (Ye et al., 2007) využili API sekvencie pre vytvorenie systému s názvom Intelligent Malware Detection System (IMDS). IMDS bol schopný detegovať všetky testované polymorfné vírusy a 92% neznámych vírusov. Napriek vysokej miere úspešnosti mali problém so spracovaním veľkej sady generovaných pravidiel a nájdením efektívnych pravidiel pre klasifikáciu nových súborov. V roku 2010 však publikovali (Ye et al., 2010) prácu, v ktorej dané problémy riešili pomocou postprocesing techník.

V práci (Jeong, 2008) pomocou API volaní vytvorili graf topológie programu, ktorý nazývame graf kódu (programu). Pri každom novom binárnom programe sa extrahoval graf a následne porovnával s existujúcou databázou grafov. Na základe toho sa rozhodovalo, či ide o malware alebo neškodný program. Aby sa zmenšila

veľkosť grafu, klasifikovali API volania do 128 skupín, čo značne zredukovalo vytváraný graf.

Inštrukcie assemblera - Opcode (Assembly Instructions)

Ako príklad použitia môžeme uviesť prácu (Siddiqui et al., 2008). Po disassemblovaní boli vybrané sekvencie inštrukcií na základe frekvencie výskytu v datasete. Na redukciu vlastností použili Chi-kvadrát test. Použili metódu logistickej regresie, neurónové siete a model rozhodovacích stromov. Pomocou modelu rozhodovacích stromov dosiahli mieru detekcie malware 98,4%.
V roku 2012 (Shabtai et al.) tiež použili inštrukcie assemblera ako vzory pre klasifikáciu pri detekcii malwaru. Vytvorili dataset škodlivých a neškodných spustiteľných súborov. Po ich disassemblovaní vypočítali normalizovanú frekvenciu výskytu slov (TF) a inverznú (TF–IDF) a použili ich ako vlastnosti súboru pre klasifikáciu. Pri testovaní použili niekoľko klasifikačných metód, ako Support Vector Machine (SVM), Logistická regresia (LR), Neuronové siete (ANN).

N-gramy

N-gramy sú jednou z najbežnejších vlastností používaných pri metódach dataminingu. Ide o podreťazce dlžky N z väčšieho reťazca. Reťazec malware by sme na 4-gramy rozdelili nasledovne: „malw“, „alwa“, „lwar“, „ware“.
Už v práci Kepharta a Arnolda z IBM (1994), ktorí vyvinuli štatistickú metódu na automatickú extrakciu signatúry škodlivého kódu, boli signatúry získané na základe pravdepodobnosti nájdenia n-gramu v malware alebo v neinfikovanom súbore.
Vírusy boli spustené v bezpečnom prostredí, aby infikovali nastražené programy. Boli extrahované možné signatúry rôznych dĺžok na základe analýzy tých infikovaných oblastí programov, ktoré boli rovnaké pri všetkých programoch. Signatúry s najmenším odhadom false pozitive (počet zle identifikovaných neinfikovaných programov) pravdepodobnosti boli vybrané ako najlepšie signatúry. Na spracovanie veľkých unikátnych sekvencií si vypožičali 3-gramový prístup z metód pre rozoznávanie hlasu, kde boli veľké sekvencie delené na 3-gramy. Potom bolo možné použiť jednoduchú aproximačnú formulu na odhadnutie pravdepodobnosti výskytu dlhých sekvencií pomocou kombinácie tých nameraných hodnôt frekvencií výskytu pre kratšie sekvencie, ktoré sa vo veľkej sekvencii nachádzajú.
V práci (Tesauro et al., 1996) použili analýzu n-gramov na detekciu malware uloženú v boot sektore pomocou neurónových sietí. N-gramy sa selektovali na základe frekvencie výskytu v škodlivom a neškodnom programe. Vlastnosti boli redukované pomocou generovania 4-násobného krytia, to znamená, že každý vírus

v datasete mal mať najmenej 4 z týchto frekventovaných n–gramov. Ale vzhľadom na obmedzenia implementácie ich klasifikátorov bolo možné analyzovať len malé boot vírusy (vírusy, ktoré sa spúšťajú z navádzacej sekcie disku), čo predstavuje asi len 5% zo škodlivého kódu.

Kolektív (Wang et al., 2005) navrhol spôsob klasifikácie rôznych typov súborov na základe ich odtlačku (fileprints). Bola použitá metóda N-gram analýzy a rozdelenia výskytu n-gramov v súbore bolo použité ako jej odtlačok. Rozdelenie bolo dané frekvenciou rozdelenia bytov a smerodajnou odchýlkou. Tieto odtlačky predstavovali normálny profil súborov a boli porovnávané s odtlačkami získanými neskôr pomocou zjednodušenej Mahalanobisovej vzdialenosti. Veľká vzdialenosť indikovala rozdielne rozdelenie n -gramov, tzn. napadnutie súboru.

Schultz et al. (2001a) použili RIPPER (systém založený na pravidlách) na dataset obsahujúci DLL knižnice. Reťazce použili pre získanie Naive Bayes klasifikátorov a n-gramy boli použité na trénovanie Multi Naive Bays klasifikátorov s hlasovacou stratégiou (voting strategy). Nepoužili žiaden algoritmus na redukciu n–gramov. Namiesto toho použili delenie dát (data set partitioning) a na každej partícii dát bolo natrénovaných šesť Naive-Bayes klasifikátorov. Využili rozdielne vlastnosti pre vytvorenie rôznych klasifikátorov, ktoré ale nie je možné rovnocenne porovnávať.

Rozšírením rovnakej idei v roku 2011 vytvorili MEF (Malicious Email Filte) mail filter na zachytávanie malware (Schultz et al., 2001b). Použili dataset, ktorý obsahoval 3301 škodlivých a 1000 neškodných programov, aby natrénovali a otestovali Naive-Bayes klasifikátory. N-gramy boli získané parsovaním hexadecimálneho výstupu (hexdump output). Kvôli redukcii počtu vlastností bol dataset rozdelený na 16 podmnožín. Každá podmnožina bola trénovaná nezávisle, a tým mala rôzne klasifikátory. Výsledok sa získal potom použitím stratégie hlasovania (voting strategy). Klasifikátory dosiahli 97,7% úroveň detekcie nových malware a 99,8% pre známe. Autori spolu s predchádzajúcou prácou vydláždili cestu pre nepreberné množstvo výskumných prác pre detekciu malware pomocou data mining prístupu.

Z ďalších prác pracujúcich s n-gramami by sme spomenuli už iba dve zaujímavé. Kolter et al. (2004) nastavili rôzne typy klasifikátorov ako Instancebased Learner, TFIDF, Naive-Bayes, Support vector machines, Decision tree, boosted Naive- Bayes, SVMs a boosted decision tree. Ako vlastnosti použili 500 najdôležitejších n-gramov, ktoré získali pomocou informačného zisku. Najefektívnejší sa podľa ich správy javil boosted decision tree J48 algoritmus.

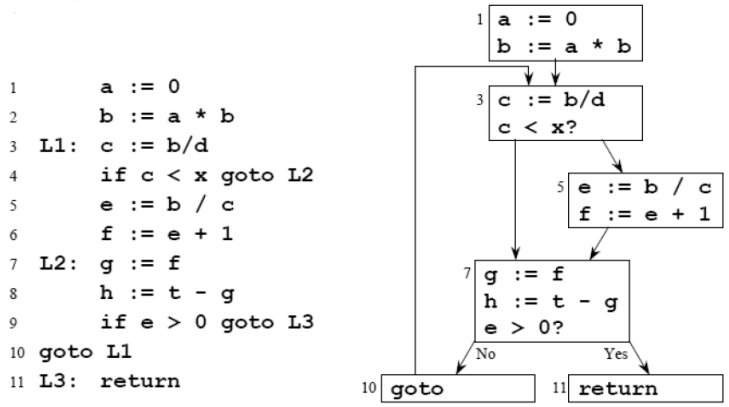
A nakoniec v nadväznosti na svoje predchádzajúce práce (InSeon, 2004) testoval pomocou väčšej zbierky dát (790 infikovaných a 80 neinfikovaných) súborov. N-gramy extrahovali z osembitoveho oktálového výpisu (dumpu) programu oproti obyčajne používaným hexadecimálnym výpisom a následne previedli hodnotu na číslo a použili ako vstup pre SOM. Použitím SOM algoritmu vytvorili

detektor vírusov. Detektor dosahoval účinnosť 84% detekcie s 30% false pozitive (nezachytených malware) hodnotami. Technika je schopná nájsť aj polymorfické a šifrované vírusy (InSeon, 2006).

Grafy toku riadenia (Control flow graph)

Graf toku riadenia je graf, ktorý zachytáva beh programu. CFG je orientovaný graf, v ktorom každý uzol predstavuje blok programu a každá hrana reprezentuje tok kontroly medzi blokmi programu. Na vytvorenie grafu je potrebné najskôr definovať jednotlivé bloky programu a následne ich pospájať (vytvoriť hrany).

Za blok grafu môžeme považovať sekvenciu riadkov kódu, ktorý má vstup len na jej začiatku a výstup len na jej konci (pozri obr. č. 27).



Obrázok 27 Príklad kontrol flow grafu

Detekcii pomocou CFG sa venuje tiež veľké množstvo prác (Jalote, 1997; McCabe, 1976; Tan, 2006). Prehľad techník môžeme nájsť v (Wilhelm, 2008). Pre ilustráciu si popíšeme niektoré viac známe.

V práci (Bruschi, 2006) autori po disassemblovaní spustiteľných súborov vykonali normalizácie, ktoré redukujú efekt mutácie metamorfných vírusov. Potom vytvorili CFG a porovnávali ho s CFG normalizovaného malwaru. Ich cieľom bolo zistiť, či CFG obsahuje podgraf, ktorý je izomorfný k CFG normalizovaného malware. Problém detekcie malwaru sa tak zmenil na problém izomorfizmu podgrafov.

V roku 2011 (Zhao, 2011) publikoval metódu detekcie založenú na vlastnostiach získaných z CFG spustiteľných súborov. Najprv vytvoril CFG z každého PE súboru, a potom z CFG extrahoval vlastnosti, ktoré použil ako trénovacie dáta. Potom na základe natrénovania vykonával klasifikáciu pomocou algoritmu dataminingu. V roku 2013 publikoval ďalší článok pre detekciu malware s využitím vlastností grafov (Zhao, 2013).

Hybridné vlastnosti (Hybrid Features)

Hybridné systémy využívajú viac vlastností ako n-gram, API funkcie alebo inštrukcie assemblera pri detekcii malware. Prvá dôležitá práca, kde boli použité inštrukcie assemblera a v ktorej pracovali s množstvom aj iných vlastností, bola práca (Weber et al., 2002). V práci navrhli PEAT (Portable Executable Analysis Tool), ktorý využíva základný princíp, že dodatočne vložený kód narúša integritu štruktúry programu. Ďalšou zaujímavou prácou využívajúcou hybridné vlastnosti bola práca (Mohammad, 2007).

Na detekciu správania sa zameriava ešte mnoho iných prác, ale správanie vyhodnocujú hlavne na základe volaných API funkcií alebo na základe toku programu. Tieto ukazovatele pri ich aplikovaní do praxe však môžu byť pozmenené a môžu vykazovať klamné informácie, napr. volanie nepotrebných funkcií, napísanie vlastnej funkcie pre danú činnosť. Preto je potrebné nájsť správne ukazovatele správania sa kódu, ktoré na jednej strane vedia odlíšiť regulárny program od škodlivého a na druhej strane sami nemôžu byť zmodifikované.

Podľa nášho predpokladu je metóda detekcie na základe správania sa kódu jediná metóda, ktorá by mohla mať (pri jej správnej implementácii) úspech. Všetky ostatné budú vo väčšej či menšej miere vykazovať chybovosť danú podobnosťou činností s regulárnymi programami. Veľakrát sa vykonávané činnosti môžu podobať (alebo rovnať) a dôležité sú ostatné okolnosti. A samozrejme škodlivý kód bude obsahovať niektoré činnosti, ktoré sa častejšie vyskytujú. Aby sme mohli sledovať správanie sa kódu, potrebujeme systém alebo model, ktorý by bol schopný emulovať činnosť procesora a jeho súčastí. Na základe neho potom vieme testovať jednotlivé programy a identifikovať reálne jeho činnosť. Následne podľa zadaných kritérií, čo je pre nás akceptovateľné správanie (alebo aký sled úkonov je dovolený), vieme zakázať /povoliť/ činnosť daného programu. Bez daného modelu alebo kvalitného emulátora môžeme správanie sa kódu iba odhadovať na základe jeho vonkajších prejavov buď pomocou dynamickej analýzy alebo sledovaním niektorých jeho činností, ktoré môžu možné správanie naznačovať. Samozrejme tieto metódy môžu byť tým pádom aj oklamané a diskriminované pri zistení spôsobu odhadovania správania.

Prehľad vlastností, na ktoré sa doteraz pri výskumoch zamerali, naznačuje veľkú rôznorodosť a oblasť záujmu. Z doterajších výsledkov vyplýva, že výsledky z veľkej miery závisia na nájdení vhodných vlastností, ktoré sa pri klasifikácii použijú. Bolo vidieť, že prístupy sú rôzne, ale všetky platia len pre určitú skupinu malware a nie je možné ich zovšeobecňovať (aj keď niektoré výsledky sú sľubné). Mnohé testy boli úspešné preto, že boli navrhnuté na základe známych vlastností vírusov. Ak sa v budúcnosti tvorcovia malware zamerajú na zmenu týchto vlastností, účinnosť danej metódy zrejme môže značne poklesnúť. Ako riešenie by sa javilo

nájsť také vlastnosti, ktoré by mohli popisovať čo najpresnejšie rozdiely medzi škodlivým a neškodným programom a zároveň boli ťažko (ak sa má funkcionalita zachovať) zmeniteľné. Medzi také môžeme zaradiť aj vlastnosti popisujúce správanie. Zaujímavým riešením z tohto hľadiska by mohol byť návrh vlastností, ktoré môžeme extrahovať priamo z operačnej pamäte systému, a tým zjednodušiť proces získavania a spracovania potrebných údajov (vylúčiť statickú alebo dynamickú analýzu) pre extrakciu vlastností (čo sa v súčasnosti stáva čoraz zložitejšie z dôvodu existencie rôznych techník ochrany kódu proti disassemblovaniu, šifrovaniu programov a jeho ukrývaniu iba v pamäti OS). V súčasnosti nie je známa práca, ktorá by sledovala vlastnosti získané priamo z operačnej pamäte.

V našej práci sa budeme snažiť zistiť, či je možné detegovať malware s využitím vlastností získaných z operačnej pamäte. Detekcia založená na využití vlastností z pamäte zatiaľ nebola publikovaná. Aby sme overili možnosť využitia informácií z pamäte, potrebujeme zistiť, ktoré aktivity a objekty z pamäte sú špecifické pre činnosť malware a môžu byť pre automatickú detekciu použiteľné. K tomu je potrebné vytvoriť testovacie prostredie, kde budeme sledovať rôzne aktivity malware v pamäti OS pomocou jej analýzy.

5 Analýza dát získaných z pamäte

Z nášho pohľadu ide o analýzu zameranú na získanie dôležitých informácií operačnom systéme, spustených procesoch a vláknach, ktoré nám poskytnú dostatočné údaje k identifikácii škodlivých kódov. Tiež nám pomôžu identifikovať nepovolené aktivity, ktoré vedú k narušeniu bezpečnosti (hookovanie systémových funkcií atď.). Analýzou pamäte s podobným cieľom (aj keď zredukované na analýzu obrazu operačnej pamäte) sa v súčasnosti zaoberá odbor forenznej analýzy. V rámci forenzných postupov sa metódam analýzy pamäte v súčasnosti venuje veľké úsilie. Viac sa k danej oblasti vyjadríme v časti 5.1.

V oblasti forenznej analýzy vyšetrovatelia často sledujú obsah pamäte RAM v nádeji, že nájdu informácie podobne, ako nachádzali informácie pomocou analýzy pevného disku (Carvey, 2009). Vychádzajú z myšlienky, že informácie, ako napr. zašifrované heslá, sa v pamäti môžu nachádzať v nezašifrovanej podobe, čo potvrdzuje aj naša práca (Balogh, 2011). Malware stále viac používajú metamorfné transformácie, takže statické offline analýzy sú veľmi ťažké. Tiež na pevnom disku zašifrovaný malware, keď sa spustí, musí byť v pamäti dešifrovaný. Ak bol malware spustený a existuje v pamäti v dešifrovanom stave, uľahčuje sa tým jeho analýza a môžeme zistiť, čo malware vykonáva. Okrem toho rootkity sú schopné skrývať procesy, súbory, kľúče v registroch a dokonca aj sieťové spojenia pred nástrojmi, ktoré sa obvykle používajú na získanie týchto informácií. Práve analýzou obsahu RAM môžeme tieto skryté informácie nájsť. Medzi tradičné techniky forenznej analýzy pamäte patrí hľadanie užitočných textových reťazcov, ako napr. heslá, mail adresy alebo sieťové IP adresy. Medzi nástroje, ktoré toto vykonávajú, patrí napr. String.exe alebo Grep.exe. (Garcia, 2007).

5.1 Metódy analýzy pamäte

V minulosti sa k pamäti pristupovalo iba ako k úložisku s množstvom neštruktúrovaných dát. Ale pamäť je z veľkej časti štruktúrovaná. Ignorovanie tejto skutočnosti je, ako keby ste hľadali obrázok alebo reťazec na pevnom disku bez toho, aby ste najskôr rozdelili dáta na adresáre a súbory (Kaplan, 2007). Základnou úlohou analýzy je teda prevod získaných bajtov z pamäte na štruktúrované informácie. Dátové štruktúry uložené v obraze pamäte sú tak základom pre analýzu dát. Dátová štruktúra obsahuje informácie o dátach a o vzťahoch medzi štruktúrami. Kvôli tomu informácie potrebné pre analýzu pozostávajú z poľa dátovej štruktúry a jej korelácie (vzťahu) s ostatnými štruktúrami (Garcia, 2007).

Mnoho úsilia v rámci forenznej analýzy je venované technikám získania informácií z obrazu pamäte o spustených procesoch. To neznamená, že iné informácie nie sú potrebné, ale výskum a práce v poslednom období boli zamerané práve na tieto

informácie. Pri forenznej analýze tiež vznikajú metódy, pomocou ktorých môžeme nájsť informácie o procesoch, ktoré už boli ukončené (Carvey, 2009).
Štruktúra, ktorá informácie o procesoch v OS Windows uchováva, sa nazýva _EPROCES. Keď sa proces ukončí, EPROCESS štruktúra je odlinkovaná zo zreťazeného zoznamu pre bežiace procesy, ale z pamäte nemusí byť hneď vymazaná. Pomocou prehľadávania pamäte môžeme štruktúru detegovať a získať informácie o ukončenom procese. Malware pri snahe schovať svoju prítomnosť sa môže tiež odlinkovať zo zreťazeného zoznamu procesov bez toho, aby sa ukončil. Tým sa pre nástroje, ktoré k výpisu bežiacich procesov používajú čítanie záznamov zreťazeného zoznamu, stáva neviditeľným.
Skenovanie (prehľadávanie) rôznych štruktúr v pamäti je technika, ktorá takto schovaný malware môže odhaliť. Postup je taký, že testujeme každý Byte v pamäti a hľadáme definované vzory, ktoré nám indikujú, či ide o časť dátovej štruktúry. Definované vzory pozostávajú zo sady podmienok na kontrolu hodnôt štruktúry. Hodnoty v štruktúre, ktoré môžeme použiť pri detekcii EPROCESS štruktúry, sú:

_EPROCESS.Pcb.Header.Type == 0x03 a eprocess.Pcb.Header.Size == 0x1b,
_EPROCESS.Pcb.DirectoryTableBase musí byť nastavený na 0x20,
_EPROCESS zoznam vlákien ukazuje do adresného priestoru jadra (obidve polia Flink a aj Blink z eprocess.ThreadListHead),
_EPROCESS.WorkingSetLock and _EPROCESS.AddressCreationLock sú platné hodnoty.

Najprv hľadáme hodnotu 0x03. Ak ju nájdeme, pozrieme sa, či je size s hodnotou 0x1b a či DirectoryTableBase je nastavé na 0x20. Overíme aj ďalšie dve podmienky a ak sú splnené, ide s veľkou pravdepodobnosťou o štruktúru EPROCESSu. Uložíme si adresu začiatku štruktúry a hľadáme ďalej v pamäti. Tak postupne vieme nájsť rôzne štruktúry jadra. Aby sme identifikovali, že ide o EPROCESS štruktúru, musia byť teda splnené všetky podmienky. Ak nie je splnená hoci iba jedna podmienka, pokračujeme v prehľadávaní bez uloženia adresy na štruktúru. Pri vytváraní pravidiel je podstatné, aby sme testovali hodnoty, ktoré nie je možné v systéme zmeniť bez toho, aby to spôsobilo kolíziu alebo pád systému. Inak malware môže danú (štandardnú) hodnotu v štruktúre prepísať, a tak zabrániť nájdeniu štruktúry pri skenovaní v pamäti (Walters, 2007).
Podobným spôsobom môžeme hľadať v pamäti ďalšie pre nás zaujímavé informácie. Ukážeme si postup pre získanie niektorých informácií, ktoré by nám mohli byť užitočné pri detekcii malware.

Zoznam procesov

Zoznam procesov je možné ľahko nájsť, stačí nám iba prejsť postupne cez obojsmerne-zreťazený zoznam, na ktorý ukazuje hodnota PsActiveProcessHead z _EPROCESS štruktúry. Týmto spôsobom ale nezískame procesy, ktoré sa odlinkovali zo zoznamu alebo už skončili. Ak chceme detegovať aj takéto procesy, môžeme použiť v predošlej časti spomenuté skenovanie celej pamäte. Tak môžeme v pamäti nájsť EPROCESS štruktúry každého procesu a získať tak zoznam všetkých procesov.

Zoznam DLL knižníc

Na získanie zoznamu DLL knižníc nahratých procesom môžeme prechádzať obojsmerne-zreťazený zoznam z LDR_DATA_TABLE_ENTRY štruktúry, na ktorý ukazuje hodnota InLoadOrderModuleList z PEB štruktúry. DLL knižnice sú na tento zoznam automaticky pridávané, keď proces volá LoadLibrary funkciu pre načítanie DLL knižnice. Pri využití iných techník na načítanie DLL knižnice do pamäte, keď sa informácie o nej neuložia do zreťazeného zoznamu z InLoadOrderModuleList, musíme použiť iný postup na detekciu (o spôsobe schovávania DLL knižníc sme písali v kapitole Techniky malware). Aj keď nie je v zozname, informácie o DLL knižnici sú vo VAD (Virtual Address Descriptor). VAD strom používa správca pamäte vo Windows systémoch na uchovanie informácií o oblasti pamäte, ktorú využíva proces. Keď proces alokuje nejakú pamäť, napr. pomocou funkcie VirutalAlloc, správca pamäte vytvorí záznam vo VAD strome. Na čítanie záznamov vo VAD nám stačí čítať obsah štruktúr _MMVAD_SHORT,_MMVAD a MMVAD_LONG, kde sú v stromovej štruktúre uložené informácie o alokovaných častiach pamäte pre proces. Viac o štruktúre VAD je možné nájsť v (Dolan-Gavitt, 2007).

SSDT tabuľka

Je viac spôsobov, ako zistiť umiestnenie SSDT tabuľky (SSDT tabuliek je v systéme viac, ale používajú sa iba dve). Často používaný spôsob je nájdenie exportovaného KeServiceDescriptorTable symbolu v NT module. Druhý spôsob je skenovať v pamäti ETHREAD objekty (podobne ako EPROCESS objekty spomenuté v predošlej časti) a zaznamenať všetky unikátne pointre (ukazujúce na začiatok SSDT) z ETHREAD.Tcb.ServiceTable. Touto metódou môžeme odhaliť aj kópie SSDT tabuliek vytvorené rootkitom.

Sieťové informácie

Aby sme mohli zaznamenať otvorené sokety pre ktorýkoľvek protokol (TCP, UDP, RAW, atď.), musíme prechádzať cez jednosmerne zreťazený zoznam zo štruktúry socket, na ktorý ukazuje ne-exportovaný symbol v tcpip.sys module. Podobne pre získanie otvorených spojení musíme prechádzať cez jednosmerne zreťazený zoznam z connection štruktury, ktorý je tiež z tcpip.sys modulu.

Atomy

Atomy (Atoms) sú reťazce, ktoré môžu byť ľahko zdieľané medzi procesmi v rámci rovnakej relácie (session). Ale miesto toho, aby sa použil priamo reťazec, ponúka atom tabuľka rýchlejšiu implementáciu. Ak nejaký proces pridá atom (reťazec) do atom tabuľky (napr. pomocou volaní AddAtom alebo GlobalAddAtom), daná API funkcia vráti číselný identifikátor, ktorý daný proces alebo iný proces použije, keď chce získať požadovaný reťazec. Na získanie zoznamu atomov v systéme potrebujeme zistiť adresu atom tabuľky v pamäti. Atom tabuľky sú reprezentované ako _RTL_ATOM_TABLE štruktúra. Formát štruktúry je odlišný pre užívateľský mód a pre mód jadra. Ak chceme analyzovať atom tabuľky (win32k!UserAtomTableHandle) a atom tabuľku pre okná (window station) tagWINDOWSTATION.pGlobalAtomTable, použijeme formát pre mód jadra. Atomy relácií sú dostupné pre všetky procesy, ktoré patria do danej relácie (session).
Štruktúra pre atom tabuľku a pre záznam v tabuľke vyzerá nasledovne (pre Windows 7 64-bitový).

```
>>> dt("_RTL_ATOM_TABLE")
' RTL ATOM TABLE' (112 bytes)
0x0   : Signature                     ['unsigned long']
0x8   : CriticalSection               ['_RTL_CRITICAL_SECTION']
0x18  : NumBuckets                    ['unsigned long']
0x20  : Buckets                       ['array', <function <lambda> at 0x10054c9b0>,
['pointer', [' RTL ATOM TABLE ENTRY']]]

>>> dt("_RTL_ATOM_TABLE_ENTRY")
' RTL ATOM TABLE ENTRY' (24 bytes)
0x0   : HashLink                      ['pointer64', ['_RTL_ATOM_TABLE_ENTRY']]
0x8   : HandleIndex                   ['unsigned short']
0xa   : Atom                          ['unsigned short']
0xc   : ReferenceCount                ['unsigned short']
0xe   : Flags                         ['unsigned char']
0xf   : NameLength                    ['unsigned char']
0x10  : Name                          ['String', {'length': <function <lambda> at
0x10428b0c8>, 'encoding': 'utf16'}]
```

Výpis 2 Štruktúra atom tabuľky

- Signature – pre atom tabuľku je hodnota 0x6d6f7441 (Atom) a štruktúra existuje v poole s tagom AtmT. To nám poskytuje dobrý základ pre jej hľadanie v pamäti.
- NumBuckets - definuje počet _RTL_ATOM_TABLE_ENTRY štruktúr v poli Buckets.
- HashLink - jednoznačne určuje každý záznam v buckete.
- Atom - číselný identifikátor pre čítanie atomu z tabuľky (funkcie AddAtom a FindAtom nám vrátia túto hodnotu).
- ReferenceCount – číslo, ktoré sa zväčšuje pri každom novom vložení špecifického reťazca do tabuľky. Keď sa reťazec maže z tabuľky, číslo sa znižuje.
- Name – slovné pomenovanie atomu.

Po získaní adresy atom tabuľky na základe informácií o jeho štruktúre môžeme jednotlivé záznamy jednoducho čítať.

Časovače (Timery)

Objekty typu timer malware často využíva pre synchronizáciu a notifikáciu. Pre detekciu timer objektov potrebujeme získať prístup k štruktúram ETIMER, KTIMER.
Tak ako štruktúry pre proces (EPROCESS) majú vnorenú štruktúru KPROCESS, tak aj každá štruktúra ETIMER má vnorenú štruktúru KTIMER. Záznamy KTIMER.DueTime a KTIMER.Period uchovávajú informáciu o čase exspirácie. KTIMER.Dpc ukazuje na callback funkciu, ktorá sa volá systémom, keď timer exspiruje.
Pri hľadaní týchto štruktúr v pamäti môžeme využiť globálny symbol nt!KiTimerTableListHead z ntoskrnl.exe. Nie je to najoptimálnejšie riešenie, lebo Microsoft pri rôznych verziach používa rôznu štruktúru dát a dokonca symbol pri Windows 7 neexistuje vôbec. Pri Windows 7 budeme postupovať tak, že KTIMERs nájdeme pomocou prechádzania zoznamu, ktorý nájdeme v štruktúre KPCR (KPCR.PrcbData.TimerTable.TimerEntries).

Spätné volania (Callbacks)

Spätné volania alebo callbacks funkcie v systéme môžeme detegovať pomocou prechádzania _DRIVER_OBJECT štruktúry, kde sú registrované, alebo pomocou skenovania poolov v pamäti.

Mutexy

Pri vytváraní mutexu objekt manažér volá BaseNamedObjects štruktúru z ntdll.dll knižnice. Pod touto štruktúrou môžeme nájsť zoznam mutexov, udalostí, semaforov, časovačov (timers) a aj objekty pre danú sekciu.

6 Realizácie práce a výsledky

Medzi nové prístupy v detekcii malware patria, ako už bolo v časti 4.3 spomenuté, aj metódy strojového učenia. Medzi ich hlavné výhody patrí možnosť ich použitia v rámci automatizovaných analytických nástrojov. Výsledky sú však v značnej miere ovplyvnené výberom vlastností pre dané algoritmy. Výber vhodných vlastností pre detekciu malware je netriviálny problém z dôvodu, že tvorcovia malware stále menia techniky a postupy použité pri činnosti malware. Ak pre analýzu využijeme vlastnosti, ktoré je možné jednoducho meniť a modifikovať bez toho, aby to ovplyvnilo funkcionalitu, daný systém detekcie nie je možné v praxi implementovať. Aj malou modifikáciou údajov sa môže malware vyhnúť detekcii pri použití skôr natrénovaných klasifikátorov. Metódy tak ostávajú funkčné len pre typy skupín malware, z ktorých sme mali vhodnú vzorku na trénovanie. Stráca sa možnosť detekcie zero-day útokov a nových skupín malware (samostatnú analýzu by si žiadal aj vplyv metamorfných malware na výsledky detekcie).

Tieto nedostatky by mali riešiť prístupy založené na sledovaní správania sa programu. Aj tu však v značnej miere ovplyvňuje výsledok spôsob, ako správanie v systéme sledujeme (útočník môže využiť iný spôsob volania danej funkcie alebo využiť inú API funkciu, aby dosiahol rovnaké správanie). Ako hypotetické riešenie sa javí nájdenie takých vlastností, ktoré by neboli získavané zo vzoriek malware, ale boli by závislé iba na systéme alebo také, ktoré nie je možné ľahko modifikovať bez narušenia funkčnosti malware. Tieto vlastnosti by mali tiež čo najlepšie vystihovať správanie malware. Medzi také vlastnosti by mohli patriť aj systémové údaje získané z pamäte počítača.

V ďalšom sa pokúsime identifikovať vlastnosti, ktoré by boli systémové (teda nie generované zo vzoriek malware) a zároveň by niesli informáciu o správaní malware. Také vlastnosti by mohli byť značne nápomocné pri detekcii a mohli by byť použité univerzálne pre všetky typy malware, ktoré by dané správanie vytváralo. Pri výbere týchto vlastností potrebujeme nájsť odpoveď na nasledujúce otázky:

- ktoré vlastnosti zo systému vypovedajú o prítomnosti malware (určitým spôsobom) a sú vhodné pre ich detekciu,
- v akej miere neznámy potenciálny škodlivý kód využíva vybrané vlastnosti,
- aká je správnosť detekcie s využitím vybraných vlastností.

Pre zodpovedanie týchto otázok sme vytvorili metodológiu na testovanie. Pri výbere vlastností sme vychádzali z podmienok uvedených vyššie, teda ide o systémové údaje, ktoré majú určitým spôsobom vypovedať o správaní programu. V našom

prípade sme sa zamerali na vlastnosti, ktoré sú dostupné pomocou analýzy pamäte. Či budú ozaj vhodné pre detekciu malware, nám ukáže testovanie. Aby sme vedeli posúdiť dôležitosť (a vhodnosť) vybraných vlastností pre klasifikáciu správania sa malware, po zbere informácií zo systému vytvoríme testovacie súbory vo formáte potrebnom pre algoritmy strojového učenia. Pomocou metód pre výber vhodných atribútov pre datamining algoritmy ohodnotíme využiteľnosť jednotlivých atribútov. Tento údaj je pre nás ukazovateľom vhodnosti použitia sledovaného atribútu pre detekciu malware pomocou strojového učenia a klasifikačných algoritmov.

Metódy pre výber atribútov pre datamining algoritmy

Tieto metódy, častejšie nazývané ako metódy výberu príznakov (feature selection), sú zamerané na nájdenie čo najvhodnejších atribútov s cieľom redukcie veľkosti priestoru atribútov (vlastností) vzhľadom na danú úlohu. Použitie metód výberu atribútov prispieva k zvýšeniu presnosti klasifikácie. Na výpočet najmenej náročné skupiny postupov, nazývané aj ako univariantné metódy (univariate methods) sú tie, pri ktorých sú atribúty považované za navzájom nezávislé (Abbasi et al., 2011). Príkladmi pre také metódy sú informačný zisk, pomerný informačný zisk, chí-kvadrát a vzájomná korelácia. Ako najefektívnejšie sa podľa štúdie (Forman, 2003) ukázali informačný zisk a chí-kvadrát. Informačný zisk sa často používa ako kritérium pre voľbu atribútov a pomerný informační zisk (information gain ratio) berie okrem entropie do úvahy aj počet hodnôt atribútu, preto je niekedy vhodnejší.

Medzi ďalšie metódy vhodné pre výber atribútov patria postupný výber (forward selection), spätná eliminácia (backward elimination) a ďalšie (MRMR, FCBF). V práci sme použili forward selection, pri ktorom sa výber parametrov uskutočňuje postupným pridávaním atribútu s najlepším výsledkom. V každej interácii sa vyberá nový nepoužitý atribút a vkladá sa do testovacej podmnožiny. V rámci interácie sa vykonáva trénovanie a testovanie dát (validácia). Validácia sa vykonáva pomocou krížovej (cross) validácie, ktorá rozdelí dataset na trénovacie a testovacie vzorky. Pomocou trénovacích vzoriek sa natrénuje klasifikátor a pomocou testovacej vyhodnotí úspešnosť klasifikácie. Celý proces sa opakuje n-krát (podľa nastavenia parametra pre krížovú validáciu). Na trénovanie sa používa vybraný klasifikátor (konkrétny datamining algoritmus). Výsledný natrénovaný model sa aplikuje na testovacie dáta a vypočíta sa správnosť (accuracy) a presnosť (precision) klasifikácie. Na základe toho sa potom priradia váhy jednotlivým atribútom. Najvhodnejšie atribúty majú maximálnu váhu (1,0). S použitím najvhodnejších atribútov sa nakoniec natrénuje klasifikátor a vypočíta celková správnosť a presnosť.

Správnosť (accurency) sa počíta podľa nasledujúceho vzťahu:

$$\text{accuracy} = \frac{\text{number of true positives} + \text{number of true negatives}}{\text{number of true positives} + \text{false positives} + \text{false negatives} + \text{true negatives}}$$

A presnosť počítame ako:

$$\text{precision} = \frac{\text{number of true positives}}{\text{number of true positives} + \text{false positives}}$$

Pozitívna trieda pri našich testoch sú čisté programy. Z toho vyplýva význam operandov:

True positives – indikuje sa prítomnosť pozitívnej udalosti, ktorá v skutočnosti ozaj nastala (v našom prípade sa čistý program označí ako čistý).

True negatives – indikuje sa prítomnosť negatívnej udalosti, ktorá v skutočnosti nastala (malware sa označí skutočne ako malware).

False positives – indikuje sa prítomnosť pozitívnej udalosti, ktorá však v skutočnosti nenastala (v našom prípade sa čistý program označí ako malware). Inak to nazývame aj falošný poplach (false alarm).

False negatives – indikuje sa prítomnosť negatívnej udalosti, ktorá ale v skutočnosti nenastala (v našom prípade sa malware označí ako čistý program).

Metodológia testovania

Navrhnutá metodológia pozostávala z nasledujúcich hlavných častí:

- Výber vhodných parametrov systému, ako vlastnosti pre metódy strojového učenia (vlastnosti vhodné pre detekciu je možné sledovať pomocou analýzy pamäte).

- Vytvorenie prostredia, kde je možné spúšťať malware a sledovať vybrané vlastnosti spustením každého malware samostatne. Zachytiť (zaznačiť) všetky sledované vlastnosti pre ďalšie spracovanie a uložiť ich do súborov. Prostredie je potrebné nastaviť tak, aby sa čo najviac podobalo reálnemu systému, aby sme minimalizovali počet malware, ktoré sa kvôli detekcii umelého prostredia neprejavia.

- Spracovanie vygenerovaných výstupov získaných pre každý malware do tvaru potrebného pre vyhodnotenie. V našom prípade sa výstupy ukladajú do formátu json alebo formátovaného txt súboru, aby sme mohli výstupy následne parsovať do databázy.

- Vyhodnotenie používania jednotlivých vlastností pomocou SQL jazyka.

- Vytvorenie datasetov s upravenými dátami z testov vhodnými pre algoritmy strojového učenia.

- Vykonanie testovania pomocou modelov strojoveho učenia a testovanie vhodnosti jednotlivých vlastností pomocou feature selection metódy.

Metodika nám umožňuje zistiť pre nás podstatné informácie o vlastnostiach a dať nám odpoveď na otázky formulované na začiatku sekcie.

6.1 Výber vlastností

V nasledujúcej časti popíšeme nami vybrané vlastnosti so zameraním na ich význam pre detekciu malware.

VAD strom (Detekcia DLL injekcie)

V časti Techniky malware sa spomínali techniky pre injektovanie DLL knižnice do systémového procesu. Pre detekciu týchto útokov môžeme využiť Virtuálny adresový deskriptor (Virtual address descriptor VAD). Vo VAD strome okrem informácií súvisiacimi s mapovaným objektom môžeme nájsť aj informácie o type a úrovni ochrany vyhradenej pamäte. Typ ochrany môže byť nasledovný[25]":

```
#define MM_ZERO_ACCESS        0
#define MM_READONLY           1
#define MM_EXECUTE            2
#define MM_EXECUTE_READ       3
#define MM_READWRITE          4
#define MM_WRITECOPY          5
#define MM_EXECUTE_READWRITE  6
#define MM_EXECUTE_WRITECOPY  7
```

[25] Zobrazenie celej MMVAD štruktúry možeme nájsť na stránke https://www.reactos.org/wiki/Techwiki:Ntoskrnl/MMVAD

Pre testovanie DLL injekcie má zmysel sledovať tie časti pamäte, ktoré majú nasledovnú úroveň ochrany:

```
#define MM_EXECUTE 2
#define MM_EXECUTE_READ 3
#define MM_EXECUTE_READWRITE 6
#define MM_EXECUTE_WRITECOPY 7
```

Teda právo pre spustenie kódu a potom aj čítanie, zápis a kopírovanie. Výhodou tohto prístupu je, že môže detegovať aj injektovaný kód, ktorý nebol súčasťou DLL. Môže nastať scénar, pri ktorom malware použije funkciu WriteProcessMemory() na transfer bloku s kódom do pamäte a následne pomocou funkcie reateRemoteThread() ju spustí. V takom scénari nemáme nikde registrovanú DLL knižnicu.

Ldrmoduly

Ako vhodnú informáciu o injektovaní DLL do systému môžeme využiť v spojitosti so sledovaním VAD štruktúr, ktoré majú práva na spúšťanie aj overenie, či kód alebo DLL, ktorý pomocou VAD nájdeme, existuje aj v regulárnych systémových zoznamoch pre knižnice (či nebol odlinkovaný alebo nahratý nie regulárnym spôsobom). Ak je knižnica nahraná regulárne pomocou API funkcie LoadLibrary, mal by existovať záznam aj v troch zoznamoch, ktoré patria do štruktúry PEB nLoadOrderModuleList, InInitOrderModuleList a InMemoryOrderModuleList.

Ak nenájdeme záznam o DLL, ktorú identifikujeme pomocou VAD štruktúry, môže ísť o chybnú identifikáciu VAD alebo o DLL, ktorá sa pokúša schovať. Odlinkovanie môže byť v jednej z dvoch alebo zo všetkých troch zoznamov v závislosti od úrovne malware (pozri výpis č. 3).

```
Pid      Process              Base       InLoad InInit InMem MappedPath
-------- -------------------- ---------- ------ ------ ----- ----------
     912 services.exe         0x7c900000 True   True   True  \WINDOWS\system32\ntdll.dll
     912 services.exe         0x7c9c0000 False  False  False \WINDOWS\system32\shell32.dll
```

Výpis 3 Ukážka záznamu pri odlinkovaní knižnice zo zoznamov

Mutexy

V jadre systému sa môžeme stretnúť s dvoma druhmi mutexov. Rýchly mutex (fast, executive mutex) sa v jadre využíva veľmi často. My sa však teraz zameriame na mutex objekty (mutanty), ktoré sú prístupné v užívateľskom móde. Program môže vytvoriť globálny mutex s nejakým názvom alebo nejakým vybraným

reťazcom. Ak sa druhá inštancia aplikácie pokúsi vytvoriť rovnaký mutex, systém mu to nedovolí a aplikácia sa zatvorí. Mutex takto môže byť jednoduchou metódou na overenie, či už v systéme beží inštancia nejakého programu. Malware využíva túto techniku často, aby predišiel opakovanej infekcii už napadnutého systému. Ako príklad môžeme uviesť, že v (Rieck et al., 2008) identifikovali pri rodine vírusov Worm.Gobot vytváranie hlavne mutantov s názvom „GhostBOT-0.57a", „GhostBOT-0.57" a „GhostBOT".

Niekedy sa môže stať, že iný malware alebo ochranný program „naočkuje" systém mutexom, aby zabránil jeho reálnej infekcii. Preto prítomnosť mutexu nie je dôkazom reálnej infekcie systému, ale len ukazovateľom možnej infekcie.

V článku (Golomb, 2011) prezentujú zoznam typických mutexov pre malware a aj pre čisté programy. Ich výpis je uvedený v prílohe B. Pri definovaní rozdielností popisujú tieto rozdiely malware mutexov v porovnaní s mutexmi z čistých programov:

1. Rozdiel v dĺžke
2. Rozdiel v „entrópii" samotných reťazcov (názvu mutexu)
3. Rozdiely vo formátovaní reťazcov
4. Rozdiely v používaní špeciálnych znakov a ako sú tie znaky rozmiestnené v reťazci

Samozrejme označenie názvu mutexu závisí výhradne na autorovi, a tak sa na tieto rozdiely nie je možné spoľahnúť, ale pre zabezpečenie unikátnosti názvu mutexu budú autory malware nútení niektorý z popísaných rozdielov zrejme využívať.

Atomy

Atom tabuľky môžu byť zaujímavé z našej perspektívy, pretože mnoho API funkcií vo Windows vytvárajú atomy na pozadí svojej činnosti bez toho, aby to explicitne prezentovali. Malware autori často používajú tieto API funkcie neuvedomujúc si ich vedľajší efekt. Niektoré atomy majú dokonca tendenciu byť prítomné v systéme dlhšie, ako je požadované. Atomy majú napríklad definovanú položku počet referencií, ktorá sa zväčšuje alebo zmenšuje v závislosti, či sa pridáva alebo odoberá daný atom v tabuľke. Podľa popisu z technického manuálu[26] atomy reťazcov ostávajú v tabuľke tak dlho, pokiaľ je počet referencií väčší ako nula, aj keby sa aplikácia, ktorá atom do tabuľky vložila, ukončila.

Keď aplikácia volá funkciu na registrovanie triedy pre Okno RegisterClassEx, ovládač win32k.sys vytvorí atom s hodnotou, akú má parameter

[26] Podrobnejšie informácie sú na stránke http://technet.microsoft.com/en-us/query/ms649053

WNDCLASSEX.lpszClassName. Niektoré škodlivé kódy (napr. Mutihack) nezadajú názov pre triedu Okna alebo použijú nie ascii znaky. Výstup pre atomy potom bude nasledovný[27] (pozri výpis č. 4).

```
AtomOfs(V)       Atom Refs   Pinned Name
---------- ---------- ------ ------ ----
[snip]
0xe179d850     0xc038      1      1 OleMainThreadWndClass
0xe17a7e40     0xc094      2      0 Shell TrayWnd
0xe17c34b8     0xc0c4      2      0 UnityAppbarWindowClass
0xe17c7678     0xc006      1      1 FileName
0xe17d40a0     0xc0ff      2      0
0xe17d4128     0xc027      1      1 SysCH
0xe17e78f0     0xc01c      1      1 ComboBox
0xe17e9070     0xc065     26      0 6.0.2600.6028!Combobox
0xe17ec350     0xc13e      1      0 Xaml
0xe18119c0     0xc08c      5      0 OM_POST_WM_COMMAND
[snip]
```

Výpis 4 Výsledok po skenovaní atomov po infekcii vírusom Mutihack

Malware, ktorý používa SetWindowsHookEx alebo SetWinEventHook na injektovanie DLL do iného procesu, nevedomky vytvára nový atom, ktorý zaznamená plnú cestu ku škodlivej DLL knižnici. API volania pre hookovanie totiž cez user32.dll volajú ovládač win32k.sys, ktorý vytvorí záznam v atom tabuľke o umiestnení knižnice, ktorá sa má injektovať. Príkladom môže byť ukážka Laqma malware:

[27] Výpis je prebratý z blogu http://volatility-labs.blogspot.sk/2012/09/movp-21-atoms-new-mutex-classes-and-dll.html

Obrázok 28 Disassemblovanie časti kódu Laqma malware

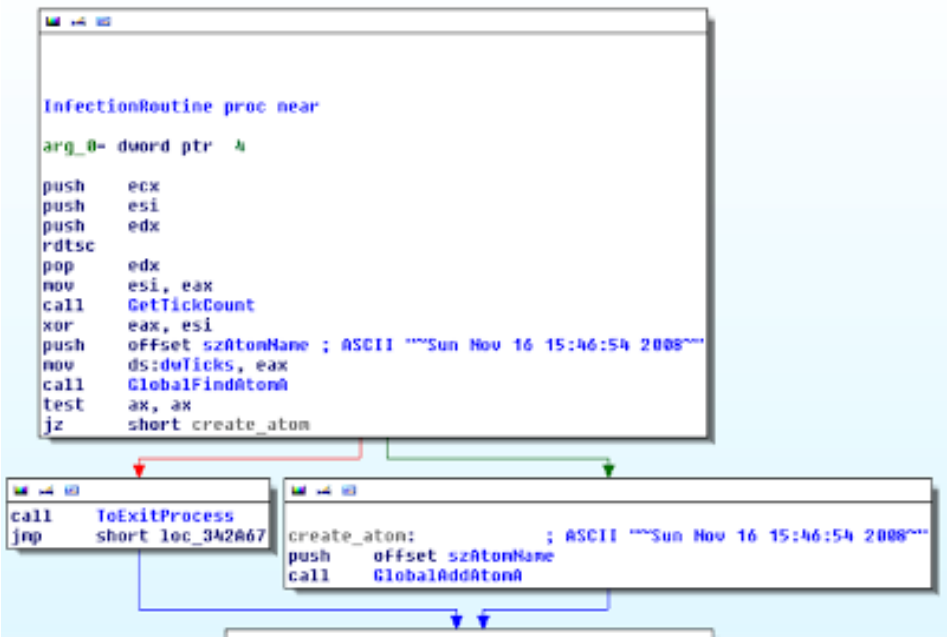
Ako je vidieť na obrázku č. 28 , parameter HINSTANCE pre SetWindowsHookEx ukazuje na knižnicu C:\WINDOWS\system32\Dll.dll. Výstup z atom tabuľky po spustení malware bude (pozri výpis č. 5):

```
AtomOfs(V)        Atom Refs   Pinned Name
---------- ---------- ------ ------ ----
0xe1000d10     0xc001      1      1 USER32
0xe155e958     0xc002      1      1 ObjectLink
0xe100a308     0xc003      1      1 OwnerLink
0xe1518c00     0xc004      1      1 Native
0xe1b2aa88     0xc1b2      2      0   axelsvc
0xe4bcb888     0xc1be      2      0 ShImgVw:CPreview
0xe11b0250     0xc1c1      2      0 __srvmgr32
0xe1f8bc30     0xc1c3      1      0 C:\WINDOWS\system32\psbase.dll
0xe28ed818     0xc1c7      1      0 BCGP_TEXT
0xe2950c98     0xc19f      1      0 ControlOfs01420000000000FC
0xe11d6290     0xc1a0      1      0 C:\WINDOWS\system32\Dll.dll
0xe1106380     0xc1a1      1      0 BCGM ONCHANGE ACTIVE TAB
0xe11a5090     0xc1a2      1      0 ControlOfs01EE0000000003C8
[snip]
```

Výpis 5 Výstup z tabuľky atomov po spustení malware

O využití mutexov škodlivým kódom sme si povedali v predošlej časti (Mutex). Niekedy môžu tvorcovia malware použiť miesto mutexov atomy pre kontrolu existencie inštancie malware v systéme. Malware Tigger je príkladom takého prístupu (pozri obr. č. 29). Najprv volá API volanie GlobalFindAtomA a kontroluje,

či existuje atom s názvom ~Sun Nov 16 15:46:54 2008~. Ak existuje, ukončí sa predpokladajúc, že systém je už infikovaný. Ak nenájde atom, volá funkciu GlobalAddAtomA, aby označil počítač ako infikovaný.



Obrázok 29 Disassemblovanie časti kódu Tigger malware

Na záver môžeme ukázať funkciu, ktorá v cykle hľadá atom s názvom „putas38“ každých 200 milisekúnd (pozri obr. č. 30). Ak by sa atom nevytvoril, cyklus by bežal do nekonečna, preto zrejme iné vlákno má za úlohu vytvoriť atom s daným názvom. Po nájdení atomu je cyklus ukončený a pokračuje sa vo vykonávaní kódu[28].

```
int __stdcall Putas_Global_Atom_Hook_Thread(int a1)
{
  while ( !GlobalFindAtomA("putas38") )
    Sleep(0xC8u);
  if ( Get_Netscape_DLL() )
  {
    Hook_Netscape_PRWrite();
  }
  else
  {
    if ( Null_Function() )
      Hook_Wininet_WSASend();
  }
  Hook_Wininet_Advapi_Functions();
  return 0;
}
```

Obrázok 30 Funcia pre testovanie prítomnosti atomu

[28]Ukážky kódu a podrobnejšie informácie je možné nájsť v blogu Michaela Ligha http://volatility-labs.blogspot.sk/2012/09/movp-21-atoms-new-mutex-classes-and-dll.html

Časovače (Timers)

Timer poskytuje veľké možnosti využitia pre malware (pozri časť 3.4 Pokročilé techniky malware), a preto môže patriť medzi často využívané postupy. Po inštalovaní rootkitu ZeroAccess môžeme pri výpise timerov zistiť, že sa pridal timer, ktorý nie je možné priradiť k žiadnemu ovládaču, preto je označený ako UNKNOWN (pozri výpis č. 6).

.

```
Offset       DueTime               Period(ms) Signaled   Routine      Module
0x805598e0   0x00000084:0xce8b961c 1000       Yes        0x80523dee   ntoskrnl.exe
0x820a1e08   0x00000084:0xdf3c0c1c 30000      Yes        0xb2d2a385   afd.sys
0x81ebf0b8   0x00000084:0xce951f84 0          -          0xf89c23f0   TDI.SYS
[snip]
0x81dbeb78   0x00000131:0x2e896402 0          -          0xf83faf6f   NDIS.sys
0x81e8b4f0   0x00000131:0x2e896402 0          -          0xf83faf6f   NDIS.sys
0x81eb8e28   0x00000084:0xe5855f6a 0          -          0x80534e48   ntoskrnl.exe
0xb20bbbb0   0x00000084:0xd4de72d2 60000      Yes        0xb20b5990   UNKNOWN
0x8210d910   0x80000000:0x0a7efa36 0          -          0x80534e48   ntoskrnl.exe
0x82274190   0x80000000:0x711befba 0          -          0x80534e48   ntoskrnl.exe
0x81de9690   0x80000000:0x0d0c3e8a 0          -          0x80534e48   ntoskrnl.exe
```

Výpis 6 Výpis timerov po spustení ZeroAccess rootkitu

Ďalšia ukážka výpisu (výpis č. 7) je zo systému po nainštalovaní rootkitu Rustock.C variant.

```
Offset       DueTime               Period(ms) Signaled   Routine      Module
0xf730a790   0x00000000:0x6db0f0b4 0          -          0xf72fb385   srv.sys
0x80558a40   0x00000000:0x68f10168 1000       Yes        0x80523026   ntoskrnl.exe
0x80559160   0x00000000:0x695c4b3a 0          -          0x80526bac   ntoskrnl.exe
0x820822e4   0x00000000:0xa2a56bb0 150000     Yes        0x81c1642f   UNKNOWN
0xf842f150   0x00000000:0xb5cb4e80 0          -          0xf841473e   Ntfs.sys
0xf70d00e0   0x00000000:0x81eb644c 0          -          0xf70c18de   HTTP.sys
0xf70cd808   0x00000000:0x81eb644c 60000      Yes        0xf70b6202   HTTP.sys
0x81e57fb0   0x00000000:0x6a4f7b16 30000      Yes        0xf7b62385   afd.sys
0x81f5f8d4   0x00000000:0x6a517bc8 3435       Yes        0x81c1642f   UNKNOWN
0x82055218   0x00000000:0x6cb1d516 10000      Yes        0xf8a126c4   watchdog.sys
0x82022530   0x00000000:0x6cb1d516 10000      Yes        0xf8a126c4   watchdog.sys
0x82007270   0x80000000:0x139ab60a 0          -          0x80534016   ntoskrnl.exe
0x82041b40   0x00000098:0x9f1d5f32 0          -          0xf83fafdf   NDIS.sys
0x8207acc0   0x80000000:0x0f13ff2e 0          -          0x80534016   ntoskrnl.exe
0x81f7eaf4   0x00000000:0x6d0082b0 20000      Yes        0x81c1642f   UNKNOWN
0x82035308   0x00000000:0x74442ce8 60000      Yes        0xf83fb72c   NDIS.sys
0x81f2f158   0x80000000:0x520f0f9e 0          -          0x80534016   ntoskrnl.exe
```

Výpis 7 Výpis timerov po inštalácii rootkitu Rustock.C

Ako vidieť, rootkit Rustock registroval tri rôzne timery, ktoré však ukazujú na rovnaké miesto v pamäti, ale exspirujú v rôznom čase[29].

[29] Vypisy a podrobnejšie informácie sa nachádzajú v blogu Michaela Lighta http://mnin.blogspot.sk/2011/10/aint-nuthin-but-ktimer-thing-baby.html

Sokety

Sokety umožňujú sledovanie a komunikáciu pomocou TCP a UDP spojenia. Malware používajúci socket API však musí pri použití vyšších úrovní protokolov a http manuálne riešiť zachytenie a komunikáciu. Napriek tomu sa pre široké možnosti, ktoré soket ponúka, používa malware pre spojenie s C&C serverom alebo na stiahnutie ďalších škodlivých súborov (trojan-downloader).

```
Offset(V)          PID   Port  Proto Protocol       Address         Create Time
------------------ ------ ------ ------ --------------- ---------------- -----------
0xfffffadfe71bbda0    432   1025      6 TCP          0.0.0.0         2012-01-23 18:20:01
0xfffffadfe7350490    776   1028     17 UDP          0.0.0.0         2012-01-23 18:21:44
0xfffffadfe6281120    804    123     17 UDP          127.0.0.1       2012-06-25 12:40:55
0xfffffadfe7549010    432    500     17 UDP          0.0.0.0         2012-01-23 18:20:09
0xfffffadfe5ee8400      4      0     47 GRE          0.0.0.0         2012-02-24 18:09:07
0xfffffadfe606dc90      4    445      6 TCP          0.0.0.0         2012-01-23 18:19:38
0xfffffadfe6eef770      4    445     17 UDP          0.0.0.0         2012-01-23 18:19:38
0xfffffadfe7055210   2136   1321     17 UDP          127.0.0.1       2012-05-09 02:09:59
0xfffffadfe750c010      4    139      6 TCP          172.16.237.150  2012-06-25 12:40:55
0xfffffadfe745f610      4    138     17 UDP          172.16.237.150  2012-06-25 12:40:55
0xfffffadfe6096560      4    137     17 UDP          172.16.237.150  2012-06-25 12:40:55
0xfffffadfe7236da0    720    135      6 TCP          0.0.0.0         2012-01-23 18:19:51
0xfffffadfe755c5b0   2136   1419      6 TCP          0.0.0.0         2012-06-25 12:42:37
0xfffffadfe6f36510   2136   1418      6 TCP          0.0.0.0         2012-06-25 12:42:37
```

Výpis 8 Vo výpise môžeme vidieť otoverené sokety ktoré boli vytvorené počas behu systému[30]*.*

Spätné volania (Callback)

Autory rootkitov začali namiesto nutnosti aktívne hooknuť komponent operačného systému na zachytenie potrebnej udalosti používať oveľa pre nich bezpečnejšiu registráciu funkcie v systéme ako callback. Takto registrovaná funkcia bude volaná vždy, keď nastane definovaná udalosť. Medzi často využívané udalosti systému, na ktoré sa rootkity registrujú, patria:

- vytvorenie nového procesu/vlákna – používa sa napr. na injektovanie DLL knižnice do práve spúšťaného procesu alebo na overenie, či proces nie je súčasťou AV nástroja a ak áno, ukončí malware daný proces. Tento typ callbacku používajú napr. rootkit Mebroot, BlackEnergy, Rustock, TDL.
- registrácia nového súborového systému v operačnom systéme – používa sa na infekciu MBR sektoru po pripojení nového disku alebo na schovanie svojich

[30] Výpis je prebratý z manuálu k volatility frameworku zo stránky https://code.google.com/p/volatility/wiki/CommandReference21#sockets

súborov pri pripojení externých diskov a USB kľúčov. Callback môžeme vidieť napr. u TDL3, Stuxnet rootkite.

- pád systému (BSOD) – tento typ callbacku môže byť použitý na vymazanie z pamäte kódu rootkitu pred vytvorením crash dumpu, ktorý sa pri BSOD začne automaticky vytvárať, alebo na overenie, či je zabezpečený štart rootkitu pri novom spustení počítača. Používa sa napr. rootkitmi Rustock.C, Sinowal.

6.2 Metóda získavania vlastností z pamäte

Na získanie navrhnutých vlastností z pamäte využijeme upravenú verziu volatility frameworku ktorý má možnosť prehľadávať obraz pamäte a nájsť umiestnenie potrebných štruktúr v pamäti. Upravená verzia, ktorá zabezpečí urýchlenie vykonania testov, pristupuje priamo do pamäte operačného systému pri jeho zapnutí, a tak nie je potrebné vytvárať obraz pamäte na získavanie potrebných vlastností z pamäte. Aj keď je to menej bezpečný spôsob (nepoužiteľný pre detekciu pokročilých rootkitov z dôvodov popísaných v časti 3.4 Pokročilé techniky malware), pre účely našich testov je dostačujúci. Naším cieľom je ušetriť čas pre detekciu a optimalizovať postup získavania údajov zameraný na čo najširšie spektrum malware. Pri skenovaní využívame možnosť vyhľadávania a sledovania štruktúr priamo v pamäti a nie prehľadávania zreťazených zoznamov štruktúr. Tým minimalizujeme možnosti ukrývania aj pre sofistikovanejšie techniky skrývania, ako DKOM manipuláciu (odlinkovanie zo zreťazených zoznamov, hooknutie funkcií pre získavanie systémových informácií, atď.).

6.2.1 Metodika zberu dát

Pre vykonanie testovania potrebujeme získať informácie o vlastnostiach a ich vytvorení testovaným súborom (malware alebo neškodný (čistý) program). Kedže sledované vlastnosti si môže vytvárať aj systém sám alebo aj aplikácie spúšťané po štarte systému (ovládače, systémové aplikácie), je potrebné vykonať zber vlastností pred spustením testovacieho súboru a aj po spustení testovacieho súboru. Potom rozdielom výsledkov získame požadované vlastnosti, ktoré vytvoril v systéme testovaný súbor.

Po spustení operačného systému je teda potrebné vykonať sken čistého systému. Pomocou v predošlej časti popísanej metódy postupne získame z pamäte informácie o testovaných vlastnostiach a ich aktuálnej prítomnosti v systéme. Údaje ukladáme do súborov označených ako čisté. Pri každom novom spustení počítača sa tento súbor vytvorí s označením čísla testu. Následne v systéme spustíme testovací súbor (malware alebo čistú testovanú aplikáciu). Po spustení testovacieho súboru počkáme chvíľu (1000ms), aby súbor mohol vykonať svoje aktivity, a potom ho pozastavíme, aby nedošlo k jeho prípadnému ukončeniu pred skončením testovania. Potom

spustíme opäť proces zachytenia informácií o vlastnostiach z pamäte. Výsledky uložíme do súborov s označením čísla testu a informácie, že ide o výstup po spustení (ukladáme aj názov spusteného testovacieho súboru).
Metodiku by sme mohli zhrnúť nasledovne:

- spustenie OS,
- sken systému a zápis informácií o všetkých vlastnostiach do samostaných súborov,
- spustenie testovaného súboru (malware, čistý program),
- znova sken systému a zápis všetkých vlastností do samostaných súborov,
- rozdielovy výstup medzi prvým a druhým skenom uložiť.

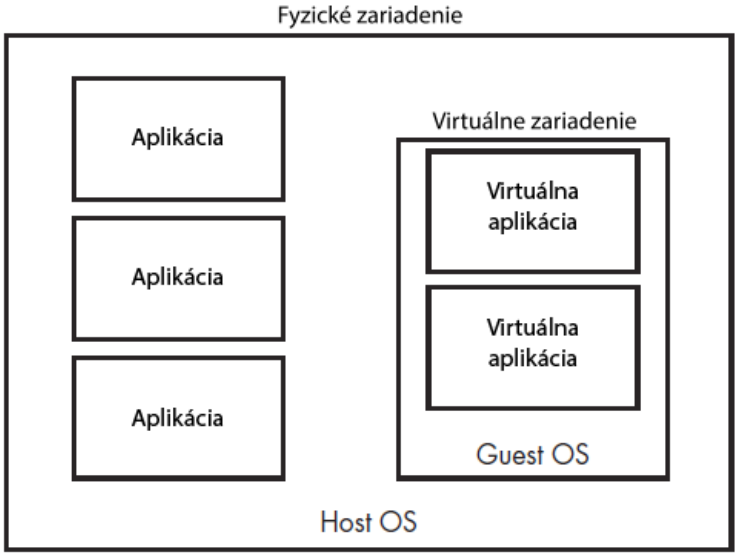
Túto metodiku sme implemetnovali pomocou vytvoreného programu v jazyku C++ a vytvorením testovacieho prostredia. Testovacie prostredie zabezpečilo rovnaké počiatočné podmienky pre každý jeden test a spustenie programu. Program po spustení zabezpečil zber informácií o vlastnostiach z pamäte pomocou volatility frameworku. Potom spustil testovaný súbor (po 1000 ms ho pozastavil) a spustil znova zber informácií. Po ukončení zberu informácií sa prostredie reštartovalo do pôvodného stavu a celý cyklus sa opakoval pre ďalší testovaný súbor.

Program pre testovanie (malwaretest)

Program bol navrhnutý tak, aby prijímal ako vstup adresár s testovacími subormi a adresár, kde má ukladať výsledky zberu informácií z pamäte. Po otvorení vstupného adresára rekurzívne prechádza a číta všetky spustiteľné súbory, ktoré sa v adresári alebo v podadresároch nachádzajú. Pre každý súbor vykoná celú procedúru zberu informácií. Výsledky zberu informácií pre každú vlastnosť (pred spustením testovacieho súboru) ukladá do samostatných adresárov. Potom spúšťa testovací súbor pomocou CreateProcess funkcie systému Windows, čo je približne rovnaký proces ako bežne malware pre spúšťanie používa. Volatility framework (ktorý je pôvodne celý napísaný v pythone) sme po vytvorení spustiteľnej verzie pre Windows spúšťali v programe pomocou systémového volania systém(). Toto volanie spušťa súbor, ktorý dostane ako parameter v cmd konzole. Po vykonaní zberu informácií z pamäte (pre každú vlastnosť samostatne) sa vyvolal reštart počítača. Program si tiež pamätá už spustené súbory a po reštarte a znovu spustení programu pokračuje nasledujúcim súborom v adresári alebo podadresári. Pri danej metodike a potrebe zabezpečiť čisté malwarom neinfikované prostredie bola taká funkcionalita programu potrebná.

Testovacie prostredie

Spustenie programu bolo v prostredí s procesorom AMD s operačným systémom Windows 7 Professional. Samotné testovacie prostredie bolo virtuálne vytvorené pomocou VirtualBox aplikácie. V testovacom prostredí bol nainštalovaný Windows XP Professional so Service pack 3. Táto konfigurácia umožňuje úspešne spustiť aj staršie verzie malware. Pre testovanie bolo použité jedno jadro procesora a pamäť o veľkosti 1GB.



Obrázok 31 Bloková schéma fungovania virtuálneho prostredia

Virtuálne testovacie prostredie zabezpečuje izoláciu od operačného systému, na ktorom je spúšťané, čo je pri testovaní malware nevyhnutná podmienka (pozri obr. č. 31). Tiež môžeme využiť možnosť vytvorenia snapshotov, a tak sa vrátiť do predošlého stavu systému.

Tým, že je dané prostredie virtuálne, máme k dispozícii sadu nástrojov a nastavení, ktoré bežne prostredie neposkytuje. Pre nás potrebnou funkcionalitou bola možnosť prepnúť virtuálny disk, na ktorom bežalo testovacie prostredie, do takzvaného nemenného módu (immutable mode). To nám zabezpečilo možnosť návratu operačného systému do pôvodného stavu po vykonaní testu.

Virtuálne prostredie nám umožňuje vytvoriť zdieľané priečinky medzi virtuálnym prostredím a hostiteľským prostredím. Zdieľané priečinky sme využili ako úložisko pre testovacie súbory a ako miesto na zápis výsledkov.

6.3 Príprava testovacieho prostredia a testovanie

Do virtuálneho prostredia bol nahratý náš program, ktorý zabezpečuje cyklus testovania. Program k svojej činnosi potrebuje mať prístup k spustiteľnému súboru pre volatility framework, preto sme ho nahrali do rovnakého adresára. Ďalším

dôležitým súborom bol konfiguračný súbor, ktorý definoval vlastnosti, o ktorých informácie z pamäte máme záujem. Nakoľko je potrebné reštartovať prostredie a dať do stavu, ktorý bol pred testom (vymazať všetky zmeny v systéme a na disku) po každom teste, aby sa nám výsledky o testovaní nestrácali, tak je potrebné ich zapisovať do zdieľaného adresára. Aby sa zabezpečilo, že program bude znovu po reštarte spustený, bol vytvorený súbor start.bat (výpis č. 9), ktorý sa načíta pri štarte systému:

```
@ECHO OFF
cd [cesta k programu]
start cmd /C " Cesta k programu\malwaretest.exe“  [-parametre]
```

Výpis 9 Výpis obsahu súboru start.bat

Aby sme zabezpečili opätovné spustenie systému po ukončení testovania a jeho vypnutí (reštartovanie nezabezpečovalo obnovenie do pôvodného stavu), vytvorili sme skript v .bat súbore, ktorý sa spustil na hostiteľskom počítači v nekonečnom cykle a pri zistení vypnutia virtuálneho prostredia ho znova spustil (pozri výpis č. 10).

```
@ECHO OFF
SET Wirt=true
cd C:\Program Files\Oracle\VirtualBox\
:check
SET Wirt=true
FOR /F "tokens=*" %%i in ('VBoxManage list runningvms') do SET Wirt=%%i
if "%Wirt%"=="true" ( VBoxManage startvm --type headless f5487518-532g578 1>nul)
goto check
```

Výpis 10 Script na znovu spustenie virtuálneho prostredia po jeho vypnutí

Po zabezpečení opakovaného spúšťania virtuálneho prostredia a samostatného programu pre vykonanie testov môžeme celý proces testovania zautomatizovať.

6.4 Výsledky a vyhodnotenie testov

Priemerná dĺžka testovania jedného súboru bola 8 minút. Z tohto dôvodu bolo celé testovanie časovo dosť náročné. Pri 100 súboroch samotné testovanie trvalo viac

ako 13 hodín. Testovanie prebehlo na 450 vzorkách malware[31] a na vzorke 100 bežných programov. Použili sme štandardné programy operačného systému Windows získané priamo po inštalácii a tiež voľne stiahnuteľné programy zo známych download serverov. Snažili sme sa používať priamo spustiteľné súbory bez nutnosti inštalácie. V súčasnosti už mnoho známych voľne šíriteľných programov má aj portable verziu.

Získané výstupy z prostredia pre testovanie sme pomocou PHP skriptu[32] parsovali a uložili do databázy. Každý typ testu sa ukladal do samostatnej tabuľky. Tu bolo potrebné zabezpečiť správne parsovanie súborov, lebo jednotlivé výstupné súbory mali rôzne formáty uloženia údajov. Tiež aby sme do databázy neukladali pomocné dáta, zároveň pri spracovávaní súborov sme vykonali aj ich selekciu (odstránili sme tie záznamy, ktoré boli v systéme aj pred spustením testovacieho programu), a tak databáza obsahovala iba zmeny vykonané programom v systéme po jeho spustení. Samostatne sme spracovali výsledky z testovania malware a čistých programov.

Po uložení výsledkov do databázy sme pomocou SQL príkazov získali prehľad o používaní jednotlivých sledovaných vlastností malwarom a čistými programami. Zistené výstupy popíšeme a prediskutujeme v nasledujúcej časti.

Atomy

Malware vytvoril pri testovaní celkovo 300 700 jedinečných atomov. Naopak čisté programy vytvorili celkovo iba 2 659 atomov. Z vytvorenia veľkého počtu niektorých špecifických atomov môžeme usudzovať, že ide o výsledok nejakej činnosti, ktorá daný atom vytvárala. Nepredpokladáme, že by toľko rôznych malware vytváralo rovnaký atom. Viditeľné to môže byť pri sledovaní záznamov o plnej ceste ku knižnici (pozri tab. č. 4). Ako sme si povedali pri technikách malware, záznam o plnej ceste sa vytvára systémom, keď sa použije API volanie pre hook DLL knižnice. Medzi rarity a ukážku snahy o použitie jedinečných znakov môžeme považovať nasledujúci zapis v atom tabuľke : „C:\Documents and Settings\lab615\Application Data\捣□□㊁ㄻ攸㊋□.dll“. Jednou z možností použitia takého názvu môže byť aj nemožnosť odstrániť takýto súbor systémom (nepozná názov). V zozname ciest vidieť aj knižnice, ktoré sú štandardne systémové. Nie je preto dôvod, aby sa načítavali do pamäte pomocou API volania a snažili sa schovávať. Logicky nám z toho vyplýva, že knižnica bola hooknutá a aby sa zabránilo identifikácii AV skenermi modifikácie, bola snaha DLL knižnicu schovať. Výsledky jednoznačne potvrdzujú vytváranie údajov v atom tabuľke malwarom, ale

[31] Vzorky boli získane z projektu malicia-project ktoré sú pre účely akademického výskumu dostupné na http://malicia-project.com/dataset.html

[32] skript s názvom run_process je súčasťou balíka processData postupný na CD k práci

kedže ide o náhodné názvy (okrem cesty k súboru, ktorá ale môže byť tiež variabilná), údaje o atomoch nie sú najvhodnejším kandidátom pre algoritmy strojového učenia.

Tabuľka 4 Zoznam atomov s plnou cestou k DLL knižnici

Name	count(*)
C:\WINDOWS\system32\wpabaln.exe	1268
C:\WINDOWS\system32\wuaucpl.cpl	1062
C:\WINDOWS\system32\MSGINA.dll	926
C:\WINDOWS\system32\winsrv.dll	926
C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83\comctl32.dll	925
C:\WINDOWS\system32\stobject.dll	918
C:\WINDOWS\system32\uxtheme.dll	916
C:\WINDOWS\System32\cscui.dll	913
C:\WINDOWS\system32\xpsp2res.dll	909
C:\WINDOWS\system32\VBoxTray.exe	909
C:\WINDOWS\system32\WINMM.dll	908
c:\windows\system32\tapisrv.dll	903
C:\WINDOWS\system32\psbase.dll	120
C:\WINDOWS\system32\riched20.dll	115
C:\WINDOWS\system32\1033\dwintl.dll	113
C:\WINDOWS\system32\shdocvw.dll	11
C:\WINDOWS\system32\DINPUT.DLL	9
C:\WINDOWS\system32\SHELL32.dll	8
C:\WINDOWS\SYSTEM32\hksdll.dll	3
C:\Documents and Settings\lab615\Application Data\捣□□㊐ㄻ攸㊍□.dll	3

Mutexy

Celkovo sme zaznamenali 536 mutexov. Z toho bolo malwarom vytvorených 221 a čistými programami vytvorených 315. Niektoré mutexy sa objavili aj medzi malwarom a aj medzi čistými súbormi, čo môže byť spôsobené volaním systémovej funkcie, ktorá vytvára tiež mutex. Napr. takým mutexom bol WininetConnectionmutex, ktorý sa nachadzal 53-krát v malware a 8-krát v čistých súboroch. Tento mutex vytvára tcpip.sys ovládač pri pokuse o vytvorenie sieťového spojenia . Mutexy, ktoré sa vyskytovali iba pri malware, by sme mohli rozdeliť na tie, čo javili črty jedinečnosti a na tie, ktoré boli vytvorené volaním nejakej funkcionality systému, pri ktorom sa mutex vytvorí. Jedinečných mutexov bolo vytvorených pomerne veľa v malých počtoch, čo hovorí o ich jedinečnosti pre každý variant malware. Z toho môžeme vyvodiť záver podobne ako pri atomoch, že nejde

o vlastnosť, ktorej názov by závisel na tom, či ide alebo nejde o malware. Skôr spomenuté rozdiely v názvoch mena mutexov sa v našom prípade potvrdili, ale tiež sme získali mnoho mutexov z čistých súborov, čo hovorí o rozšírenosti používania mutexov aj v iných aplikáciach Windows. Možná klasifikácia by prichádzala do úvahy iba po ďalšej podrobnej analýze názvov a potvrdenia načrtnutých rozdielov v názvoch medzi mutexami. Nie je však žiadna záruka, že autori malware svoje zvyky pri zadávaní názvov nemôžu porušiť. Jediný argument, ktorý núti pri malware používať iný spôsob vytvárania mena mutexu, je nutnosť zabezpečiť jedinečný názov. To sa ale týka iba mutexov vytvorených s cieľom overenia, či je systém už infikovaný.

VAD

Pri tejto vlastnosti sme získali najviac jedinečných záznamov. Rozdiely v pamäťových oblastiach vytvorených malwarom a čistým kódom je viditeľný. Výrazne to môžeme vidieť pri type oblasti EXECUTE_WRITECOPY, EXECUTE_READ, EXECUTE_READWRITE. Aj keď si väčšinou každý spustený proces v hlavnom vlákne vytvorí takú oblasť, ale ide maximálne o jednu až niekoľko oblastí. Zmysel má pre nás sledovanie týchto oblastí u procesoch, ktoré majú systémové práva alebo sú priamo procesmi operačného systému. Ak si porovnáme niektoré také procesy, vidíme, že malware techniku DLL injekcie používa pri svojej činnosti veľmi často (pozri tab. č. 5).

Tabuľka 5 Prehľad výskytu spustiteľných oblastí v systémových procesoch

Proces	Výskyt malware	Výskyt ne-malware
explorer.exe	187	3
csrss.exe	47	0
alg.exe	71	0
dwwin.exe	1218	0
KERN32.EXE	26	0
services.exe	58	0
lsass.exe	129	2
spoolsv.exe	104	0
svchost.exe	438	38
System	2	0
winlogon.exe	26	0
wuauclt.exe	204	0
wscntfy.exe	34	0
wscript.exe	36	0

Tabuľka č. 5 nám dokazuje časté dodatočné vytvorenie VAD oblastí s nastaveným EXECUTE právom v systémových procesoch. Mnoho z týchto procesov VAD oblasti s právami EXECUTABLE pri svojom bežnom behu vôbec nemá (wuauclt.exe, system, winlogon.exe). Viac systémových procesov bolo aj takých, ktoré boli spustené malwarom a bežne po spustení operačného systému nebežali (spoolsv.exe, wscript.exe, dwwin.exe). Pritom sa rapídne často objavuje proces dwwin.exe, ktorý je súčasťou Microsoft Doctor Watson Windows. V popise programu sa píše, že ide o ladiaci nástroj chýb v programoch, ktorý zhromažďuje informácie o počítačipri výskyte chyby[33]. Z toho vyplýva možnosť jeho širokého využitia. Z výsledkov testu vyplýva, že informácie o VAD štruktúre nám môžu vypovedať veľa o napadnutí a spustení malwaru v systéme.

Časovače (Timers)

Z výsledku testu vidieť použitie timerov, ale v našich testoch sme zistili iba vytvorenie 7 jedinečných timerov malwarom. Timery si vytvárajú niektoré systémové procesy, ako vidieť z tab. Č. 5 (čisté procesy). Z tohto pohľadu pre nás timery nepredstavujú výrazný prínos pre detekcie (zriedkavý výskyt), ale ak sa identifikuje vytvorenie timeru neštandardým systémovým procesom, je veľká pravdepodobnosť, že bola vytvorená malwarom.

Tabuľka 6 Timery pri čistých programoch

Periodms	Module	count(*)
100	tcpip.sys	36
1000	ntoskrnl.exe	22
1000	NDIS.sys	17
60000	ntoskrnl.exe	4

Tabuľka 7 Timery pri malware

Periodms	Module	count(*)
1000	ntoskrnl.exe	46
100	tcpip.sys	39
1000	NDIS.sys	30
10000	watchdog.sys	4
30000	HTTP.sys	2
2141869928	UNKNOWN	1
60000	ntoskrnl.exe	1

[33] Popis programu môžeme nájst na stránke http://support.microsoft.com/kb/308538/sk

Tabuľka 8 Jedinečné timery pri malware

Periodms	Module	count(*)
10000	watchdog.sys	4
30000	HTTP.sys	2
-2141869928	UNKNOWN	1

Spätné volania (Callbacks)

Callback bol v testoch vytvorený iba raz pri spustení malware. Vytvoril sa callback štyroch rôznych typov IoRegisterShutdownNotification, IoRegisterFsRegistration-Change, KeBugCheckCallbackListHead, KeRegisterBugCheckReasonCallback, čo môže veľa vypovedať o dôvode ich vytvorenia (pozri kapitolu pokročilé techniky malware). Z výsledku testu vyplýva, že použitie callback nie je bežné ani pre systémové procesy a ani pre bežné aplikácie. Dokonca ani malware ho v súčasnej dobe vo veľkej miere nevyužíva. Ide zrejme skôr o techniku implemetovanú iba niektorými typmi rootkitov. Z pohľadu využitia ako vlastnosti pre datamining je z dôvodu jeho nízkej frekvencie výskytu pre nás menej prínosná, ale ak sa výskyt objaví, s veľkou pravdepodobnosťou ide o napadnutie malwarom.

Ldrmoduly

Sledovanie DLL knižníc neregistrovaných v zoznamoch modulov jednoznačne ukazuje použitie DLL injekcie. Kým pri malware je výskyt neregistrovania v zoznamoch modulov veľmi častý, pri čistých programoch sa prakticky vôbec nevyskytuje (pri čistých programoch iba pri základnom vlákne chýba registrácia v module Init, čo je prirodzené, v ostatných moduloch je aj zakladné vlákno registrované). Na ukážku si môžeme pozrieť porovnanie v tab. č. 9.

Tabuľka 9 Porovnanie výskytu neregistrovaných DLL knižníc

Proces	Výskyt v malware	InLoad	InInit	InMem	Výskyt v ne-malware
csrss.exe	656	False	False	False	0
dwwin.exe	43	True	False	True	0
dumprep.exe	2	False	False	False	0
explorer.exe	326	True	False	True	0
alg.exe	327	True	False	True	0
csrss.exe	656	False	False	False	0
lsass.exe	328	True	False	True	0
smss.exe	328	True	False	True	0
services.exe	328	True	False	True	0
spoolsv.exe	327	True	False	True	0
ntvdm.exe	150	False	False	False	0
ntvdm.exe	0	True	False	True	6
svchost.exe	1635	True	False	True	0
System	328	False	False	False	0
wuauclt.exe	383	False	False	False	0
winlogon.exe	328	True	False	True	0
wscntfy.exe	318	True	False	True	0
wscript.exe	1	True	False	True	0

Z tabuľky vyplýva, že neprítomnosť knižnice v zoznamoch modulov pri systémových procesoch ukazuje jednoznačne na infekciu malwarom. Pri čistých programoch sa objavil v prípade ntvdm.exe, v 6 prípadoch chýbajuci záznam v init module. Ntvdm.exe proces poskytuje prostredie pre 16-bitové procesy, aby mohli byť spustené na 32-bitovej platforme[34]. Z tohto hľadiska nejde o bežný systémový proces. Z výsledkov vyplýva, že ak sa knižnice načítavajú do systémových procesov v init module, nie sú registrované. Otázka ale je, prečo sa do systémových procesov knižnice načítavajú. Z výsledkov vyplýva, že táto aktivita sa týka výlučne malwaru. Z tohto pohľadu sledovanie tejto vlastnosti môže k detekcii malwaru v systéme byť zjavne prínosom.

[34] Podrobnejší popis ntvdm.exe procesu je na stránke http://www.processlibrary.com/en/directory/files/ntvdm/24761/

Sokety

Výsledky pri sledovaní otvorených soketov pri testoch neposkytujú žiadne vhodné informácie. Aj napriek spusteniu simulácie siete pomocou nástroja FakeNet sme nezachytili žiadne pokusy o vytvorenie sieťových spojení. Dôvodom je zrejme špecifické nastavenie siete vo virtuálnom prostredí, ktoré neumožňovalo otvorenie komunikácie do inej siete. Týmto problémom sa chceme v budúcnosti zaoberať, aby bolo možné v našom testovacom prostredí sledovať aj pokusy o komunikáciu.

6.5 Testovanie vlastností s použitím datamining algoritmov

Pre overenie významu vlastností pre detekciu malware s využitím dataminig algoritmov sme vytvorili testovací dataset, ktorý obsahoval testované vlastnosti a niektoré ich parametre. Na testovanie sme použili IBM bladeCenter HS23 s 8 CPU a 128GB RAM a s inštalovaným OS Debian verzia 7.

Pre testovanie sme vybrali tieto parametre jednotlivých vlastností:

- Mutant - Mutantname,malware,file_name,
- Atomscan – atomname,Refs,malware,file_name,
- Timer-periodms, moduletimer,malware,file_name,
- Vad - typevad,vadprocess,Protect, vadpath,malware,file_name,
- Callbacks - typecallback,modulecallbaks,details,malware,file_name,
- Ldrmodule - ldrprocess,InLoad, InInit,InMem, ldrpath,

kde
Mutantname – názov mutexov vytvorených testovaným súborom,
Malware – označenie, či je záznam z testu malware alebo čistého súboru (y/n),
file_name – názov testovaného súboru,
atomname – názov atomov vytvorených testovaným súborom,
Refs – počet referencií na daný atom,
Periodms – čas v ms zadaný pre časovač (timer),
Moduletimer – modul, ktorý registroval timer,
Typevad – typ VAD oblasti,
Vadprocess – názov procesu, v ktorom bol VAD vytvorený,
Protect – nastavené práva pre VAD oblasť,
Vadpath – cesta k súboru (DLL knižnici), pre ktorú bola VAD oblasť vytvorená,
Typecallback – typ vytvoreného callback záznamu,
Modulecallbaks – modul, ktorý callback registroval,
Details – popis modulu, ktorý callback registroval,
Ldrprocess – názov procesu, v ktorej bola DLL knižnica načítaná,

Inload – prítomnosť v (true/false),
Ininit – prítomnosť v (true/false),
Inmem - prítomnosť v (true/false),
Ldrpath – cesta k DLL knižnici.

Pri testovaní sme vytvorili tri rôzne typy datasetu. Prvý dataset sme vytvorili pomocou spojenia (join) jednotlivých vlastností. Ako ID na prepojenie sme použili názov testovacieho súboru (file_name). Názov súboru však nebol v získaných dátach pre jednotlivé vlastnosti jedinečný a vyskytoval sa viackrát, kedže jeden súbor vytvoril viac záznamov. Tým sme po vykonaní joinu dostali dataset o veľkosti 6GB. Funkcia join spája záznamy pri viacnásobnom výskyte na obidvoch stranách formou každý s každým (karteziánsky súčin). Veľkosť datasetu neumožňovalo reálne testovanie, ani pri použití našej hardvérovej konfigurácie.

Druhý dataset bol vytvorený pomocou spojenia jednotlivých vlastností bez ich opakovania. K tomu sme nemohli použiť funkciu join, ale vytvorili sme si vlastný program v jazyku C++ s názvom „joining tables“[35]. Takto sme dostali dataset bez viacnásobného spojenia údajov v tabuľke. Veľkosť datasetu bola 53 MB.

Pre testovanie významnosti vlastností sme použili metódy pre výber atribútov (forward selection). Ako klasifikátor sme použili rôzne algoritmy strojového učenia a dataminingu, konkrétne algoritmy Naive Bayes, W-JRip, W-Ridor, W-NBTree, W-Id3 a W-J48. K testovaniu bol použitý datamining framework RapidMiner, kde je možné nájsť aj detailný popis jednotlivých algoritmov. Algoritmy W-Jrip a W-Ridor patria medzi pravidlové algoritmy a W-Id3 a W-J48 medzi rozhodovacie stromy.

Pre použitý dataset bolo úspešne realizované testovanie iba pri klasifikátore W-Ridor a Naive Bayes, pričom feature selection pre W-Ridor trval 5 dní. Pri ostatných algoritmoch boli testy neúspešné z dôvodu nedostatku prostriedkov. Pri testovaní algoritmov bola použitá 10-násobná krížová validácia (cross validation). Výsledky je možné vidieť v tabuľke č. 10.

[35] Zdrojové kódy sú dostupné na CD k práci

Tabuľka 10 Výsledky feature selection pre druhý dataset

	Naive Bayes			W-Rider		
Atribúty	WHT	ACC	PRC	WHT	ACC	PRC
PerformanceVector(celkovo)		96.04% +/- 0.03	99.57% +/- 0.32		94.54% +/- 0.07	62.40% +/- 1.99
atomname	0.0	0.947753	0.8986	1.0	0.941298	
ldrprocess	0.0	0.948082	0.9968	1.0	0.941298	
inload	0.0	0.941298		1.0	0.941298	
ininit	0.0	0.941298		1.0	0.941298	
inmem	0.0	0.941298		0.0	0.941298	
ldrpath	0.0	0.941163	0.4415	1.0	0.941298	
protect	0.0	0.941298		1.0	0.941298	
vadpath	0.0	0.941788	0.6130	0.0	0.941298	
vadprocess	1.0	0.959902	1.0	0.0	0.941298	
mutantname	1.0	0.94194	0.9697	1.0	0.941389	0.9490
periodms	0.0	0.941105	0.4292	1.0	0.941276	0.2352
moduletimer	0.0	0.94122	0.3461	1.0	0.941271	0.3777
typecallback	0.0	0.941298		1.0	0.941298	
modulecallbaks	1.0	0.941298		0.0	0.941298	
details	0.0	0.941298		0.0	0.941298	

V tabuľke č. 10 sú zobrazené hodnoty AttributeWeights (WHT), t.j. výsledné váhy priradené jednotlivým atribútom feature selection funkciou a tiež hodnoty pre výslednú správnosť (ACC - accurency) a presnosť (PRC - precision) pri použití týchto váh (PerformanceVector). Váhy sa vyberali na základe najlepších výsledkov pre správnosť a presnosť feature selection funkciou. Uvedené sú aj hodnoty správnosti a presnosti pre konkrétne atribúty samostatne.

Na základe neuspokojivých výsledkov (zlyhanie testov, časová náročnosť) sme vytvorili tretí dataset, kde sme atribúty definovali z údajov testov uložených v databáze. Databáza nám umožňovala vytvoriť súhrné a agregované pohľady. Na základe týchto informácií a poznantkov o malware sme vedeli vybrať nové atribúty, ktoré ukazujú na hlavné rozdiely v hodnotách získanych údajov medzi testovanými vzorkami malware a čistým kódom. Nový dataset obsahoval nasledujúce atribúty reprezentujúce naše testovacie vlastnosti:

Atribúty reprezentujúce atomy

atompathcount – reprezentuje počet záznamov v atom tabuľke s úplnou cestou k súboru alebo DLL knižnici, ale iba tie, pri ktorých existoval rovnaký záznam z ldrmodul pri ldrpath. Plná cesta k súboru sa do atom tabuľky zapíše v prípade, keď sa k injekcii DLL knižnice použije API funkcia SetWindowsHookEx. Hodnota v ldrpath obsahuje cestu k jednotlivým používaným knižniciam procesov, čo nám umožnilo overiť, či nájdená cesta v tabuľke atomov korešponduje reálne s knižnicou. Ak áno, zrejme ide o injektovanú knižnicu a takéto záznamy sa vyskytovali vo veľkej miere pri testovaných malware, čo môže byť dobrým rozlišovacím atribútom.

Atribúty reprezentujúce mutexy

Mutexcount – reprezentuje počet mutexov vytvorených testovaným súborom. Mutexy boli vytvárané aj čistými súbormi, ale je to technika používaná aj malware k zamedzeniu opakovaného spustenia v jednom systéme. Preto výskyt mutexu môže pomôcť k rozhodovaniu, ale zrejme bude mať menšiu váhu, lebo sa vyskytuje aj pri čistých súboroch.

Atribúty reprezentujúce informácie o VAD oblastiach

Výsledky záznamov vo VAD tabuľkách ukazujú na rozdiely vo vytvorených mapovaných oblastiach pre jednotlivé procesy. Môže to byť tiež ukazovateľ injektovania DLL knižnice. Pre nás bolo zaujímavé sledovať vytvorenie nových oblastí po spustení súboru v systéme pri systémových procesoch.Tieto procesy majú väčšinou administrátorské práva a ak sa podarí inému prosesu injektovať svoju knižnicu do takého procesu, može vykonávať svoju činnosť s administrátorskými právami, ktoré má daný proces. Na sledovanie sme vybrali tieto systémové procesy: Explorer.exe, services.exe, smss.exe, csrss.exe ,winlogon.exe,lsass.exe a svchost.exe, ktoré sú dôležité pre funkčnosť operačného systému a spúšťajú sa po štarte systému. Preto bývajú často cieľom pre injektovanie DLL knižnice malwarom.

Význam pre nás mali iba tie vytvorené oblasti, ktoré mali nastavenú úroveň ochrany EXECUTE, EXECUTE_READ, EXECUTE_READWRITE, EXECUTE_WRITECOPY (pozri časť 6.1 Výber vlastností).

Na základe týchto predpokladov sme vybrali nasledujúce atribúty:

- explorervadcount – počet vytvorených VAD oblastí s execute právami v procese Explorer.exe,
- servicesvadcount - počet vytvorených vad oblastí s execute právami v procese services.exe,

- smssvadcount - počet vytvorených vad oblastí s execute právami v procese smss.exe,
- csrssvadcount - počet vytvorených vad oblastí s execute právami v procese csrss.exe,
- winlogonvadcount - počet vytvorených vad oblastí s execute právami v procese winlogon.exe,
- lsassvadcount - počet vytvorených vad oblastí s execute právami v procese lsass.exe,
- svchostvadcount - počet vytvorených vad oblastí s execute právami v procese svchost.exe,
- execute_readvadcount – celkový počet súborom vytvorených oblastí s úrovňou ochrany EXECUTE_READ,
- execute_readwritevadcount - celkový počet súborom vytvorených oblastí s úrovňou ochrany EXECUTE_READWRITE,
- execute_writecopyvadcount - celkový počet súborom vytvorených oblastí s úrovňou ochrany EXECUTE_WRITECOPY.

Atribúty reprezentujúce informácie o DLL knižniciach z ldrmodules sledovanej vlastnosti

Počas testovania sme sledovali aj regisráciu nahraných DLL knižníc v zoznamoch modulov (InLoadOrderModuleList, InMemoryOrderModuleList, InInitialization-OrderModuleList). Pri štandardnom nahrávaní knižnice do procesu dochádza k registrovaniu v týchto zoznamoch. Pri injektovaní DLL knižnice sa však malware často snaží obísť registráciu alebo knižnicu zo zoznamu vymazať, aby tak utajil existenciu knižnice v procese. Podobne ako pri VAD sme sledovali nové načítané DLL knižnice v systémových procesoch a počet neregistrovaných DLL knižníc v zoznamoch. Neregistrované knižnice sa vyskytovali prakticky iba pri malware.

Na základe týchto predpokladov sme vybrali nasledujúce atribúty:

- Explorerldrmodulecount – počet nových DLL knižníc načítaných po spustení súboru v procese Explorer.ese,
- Servicesldrmodulecount - počet nových DLL knižníc načítaných po spustení súboru v procese services.ese,
- Smssldrmodulecount - počet nových DLL knižníc načítaných po spustení súboru v procese smss.ese,
- Csrssldrmodulecount - počet nových DLL knižníc načítaných po spustení súboru v procese csrss.ese,
- Winlogonldrmodulecount - počet nových DLL knižníc načítaných po spustení súboru v procese winlogon.ese,

- Lsassldrmodulecount - počet nových DLL knižníc načítaných po spustení súboru
  v procese lsass.ese,
- Svchostldrmodulecount - počet nových DLL knižníc načítaných po spustení súboru v procese svchost.exe,
- inloadfalseldrmodulecount – počet neregistrovaných knižníc v InLoadOrderModuleList,
- Inmemfalseldrmodulecount - počet neregistrovaných knižníc v InMemoryOrderModuleList.

Atribúty reprezentujúce informácie o timeroch a callback vlastnosti

Pri našich testoch bolo celkovo vytvorených málo záznamov pre timery a callback funkcie. Preto sme ich nezahrnuli do atribútov. Výsledný dataset obsahoval iba číselné hodnoty, čo nám umožňovalo použiť viac metód pre výber atribútov a ohodnotiť významnosť jednotlivých atribútov. Ako prvé sme použili univariantné metódy pre váhovanie atribútov. Boli použité tieto metódy:

- váhovanie na základe informačného zisku (information gain),
- váhovanie na základe pomerného informačného zisku (information gain ratio),
- váhovanie na základe chí-kvadrát metódy (chi squared),
- váhovanie na základe korelácie (correlation).

Výsledky testovania je možné vidieť v tabuľke č. 11.

Tabuľka 11 Výsledky váhovania atribútov vybranými metódami

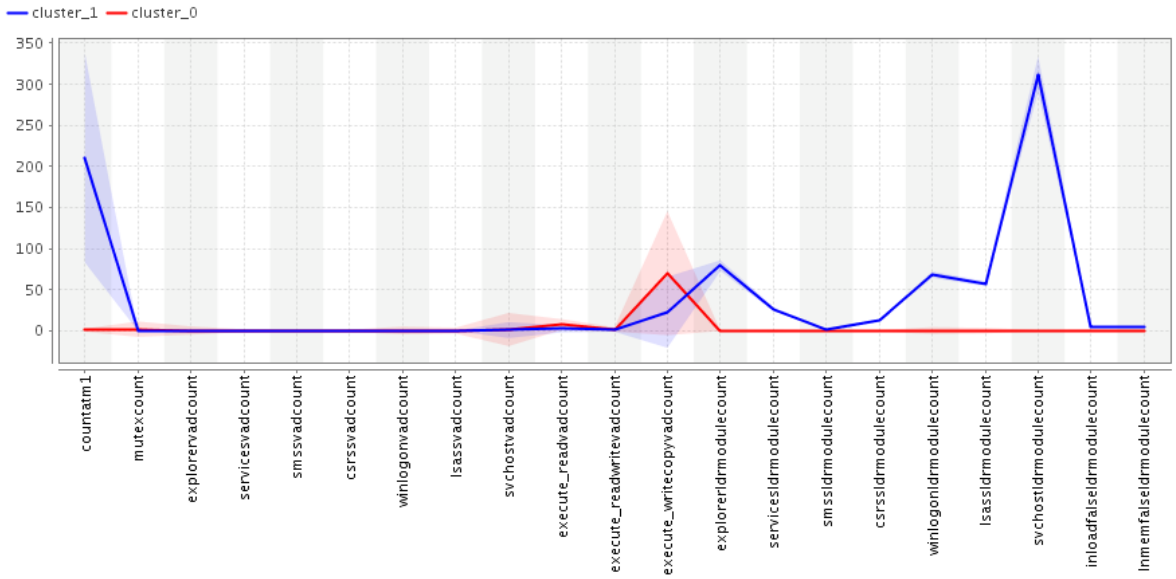
Názov atribútu/ algoritmus	Chí-kvadrát	Informačný zisk	Pomerný informačný zisk	Korelácia
Atompathcount	0,555	0,963	0,803	0,725
mutexcount	0,203	0,305	1	0,309
explorervadcount	0	0,016	0	0,004
servicesvadcount	0	0	0	0,001
Smssvadcount	0	0	0	0,001
csrssvadcount	0,001	0,006	0,024	0,01
winlogonvadcount	0	0,012	0,037	0,005
Lsassvadcount	0	0	0	0
svchostvadcount	0,005	0,123	0,268	0,012
execute_readvadcount	0,673	0,804	0,659	0,676
execute_readwritevadcount	0,044	0,79	0,644	0,046
execute_writecopyvadcount	0,185	0,771	0,625	0,667
explorerldrmodulecount	0,872	0,944	0,783	0,979
servicesldrmodulecount	0,884	0,953	0,793	0,979
Smssldrmodulecount	0,884	0,953	0,793	0,987
csrssldrmodulecount	0,884	0,953	0,793	0,985
winlogonldrmodulecount	0,884	0,973	0,813	0,987
Lsassldrmodulecount	0,884	0,958	0,798	0,987
svchostldrmodulecount	0,878	0,949	0,788	0,98
inloadfalseldrmodulecount	1	1	0,851	1
Inmemfalseldrmodulecount	1	1	0,851	1

Ako ďalšiu metódu pre výber atribútov sme použili forward selection tým istým spôsobom, ako pri druhom datasete. V tomto prípade bolo možné použiť viac klasifikátorov (DTNB, J48, W-NBTree, ID3 num) a neobjavila sa ani časová náročnosť. Dĺžka testovania bola niekoľko minút pri danej hardvérovej konfigurácii. V datasete bolo celkovo 597 záznamov, z toho 501 záznamov bolo pre malware a 96 pre čisté programy. Aby sme vylúčili možnosť skreslenia z dôvodu nevyváženia počtov vzoriek, vykonali sme testy pri použití všetkých záznamov z malware a tiež testy pri použití sample funkcie, pomocou ktorej sme vyberali pre validáciu z malware iba 100 náhodných záznamov a všetky záznamy z čistých programov. Porovnanie rozdielov vo výsledkoch je v tabuľke č. 12. Podrobné výsledky aj s hodnotami váh a správnosti pre jednotlivé atribúty je možné vidieť v prílohe C práce (kvôli jej veľkosti). Výsledky v prílohe C sú zobrazené pre prípad výberu 100 záznamov z malware (použitie samplovania).

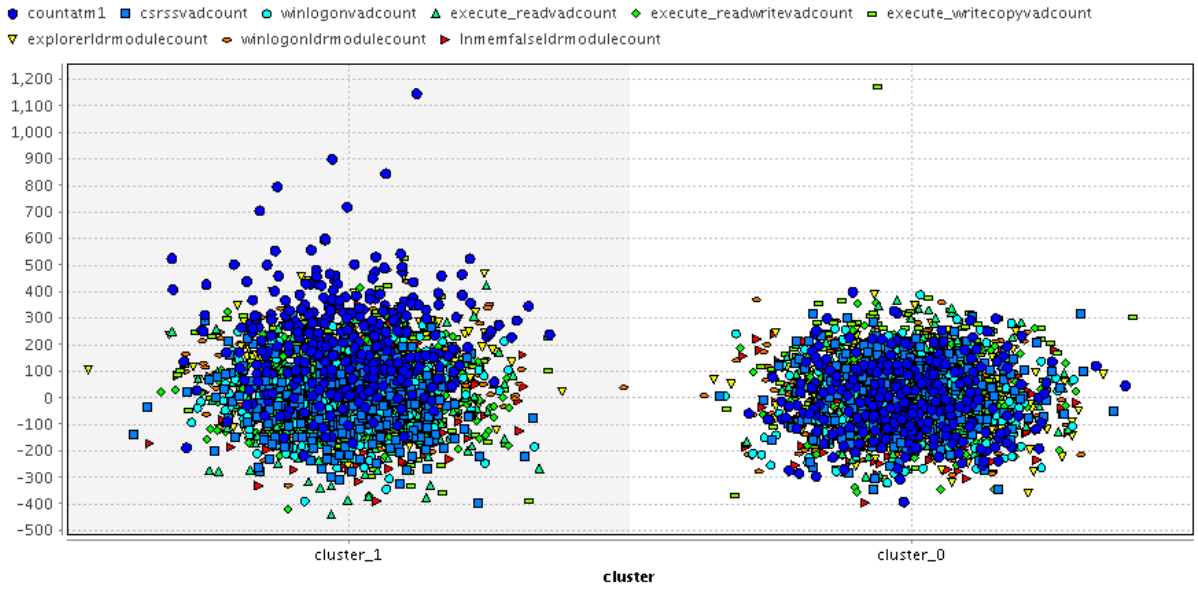
Tabuľka 12 Výsledky forward selection na treťom datasete pri použití všetkých záznamov a iba vybraných 100 záznamov pre malware

	Všetky prvky		Výber 100 prvkov z malware	
	accuracy	Precision	Accuracy	Precision
DTBN	90.78% +/- 2.85	66.44% +/- 7.03	95.02% +/- 3.78	94.27% +/- 6.48
ID3 num	95.31% +/- 1.79	87.05% +/- 2.71	94.57% +/- 3.97	94.16% +/- 6.56
J48	93.81% +/- 1.05	80.34% +/- 6.10	94.15% +/- 4.47	93.11% +/- 5.92
Jrip	94.31% +/- 3.10	83.14% +/- 9.89	95.47% +/- 2.88	95.07% +/- 4.96
Ridor	94.14% +/- 2.91	83.58% +/- 13.87	95.02% +/- 3.19	94.31% +/- 6.00
NBTree	92.80% +/- 3.8	81.08% +/- 16.50	95.02% +/- 3.78	94.27% +/- 6.48

Pri tomto datasete sme mohli použiť aj algoritmus pre klastrovanie K-means. Výsledok je možné vidieť na obr. č. 32 a 33.



Obrázok 32 Diagram štandardnej odchýlky pri zobrazení výsledku klastrovania tretieho datasetu pomocou K-means



Obrázok 33 Bodový graf (korelačný diagram) pre zobrazenie výsledku klastrovania pomocou K-means algoritmu pri treťom datasete

6.6 Vyhodnotenie výsledkov datamining testov

Celkovo sme testovali tri datasety s rôznou štruktúrou a obsahom dát. Výsledky mali niektoré spoločné črty, ale v mnohých parametroch je vidieť rozdiely. Prejdeme si postupne jednotlivé datasety.

Prvý dataset

Výsledky nebolo možné získať z dôvodu značnej výpočtovej náročnosti. Z tohto môžeme vyvodiť fakt, že priame výstupy zo sledovania vlastností v systéme nie je možné priamo spojiť a klasifikovať datamining algoritmami. Je potrebné predtým získané údaje pre datamining klasifikátory predspracovať do vhodnejšej formy vstupných dát.
Samostatné klasifikovanie vlastností a potom spoločné vyhodnotenie výsledku klasifikovania jednotlivých testovaných vlastností nebolo tiež vhodným riešením, lebo pre jednotlivé vlastnosti sme mali málo vhodných atribútov pre klasifikáciu. Použiteľný je pre nás iba dataset, ktorý spája vhodné atribúty zo všetkých vlastností do jedného datasetu.

Druhý dataset

Druhý dataset obsahoval výsledky aktivít testovaného programu v systéme pre každú sledovanú vlastnosť. Tým sa v datasete pre jeden testovaný súbor nachádzalo viac riadkov záznamov. To zrejme vytváralo pri klasifikátoroch založených na rozhodovacích stromoch a aj niektorých pravidlovo založených veľké

pamäťové nároky a viedli k zložitosti klasifikácie. Klasifikátory založené na rozhodovacích stromoch (J48, ID3, DTNB) skončili s chybou dôsledkom nedostatočnej pamäte, napriek nadštandartnej konfigurácii[36]. Podobne skončili aj pravidlovo založené klasifikátory (JRip), okrem klasifikátora W-Ridor pri ktorom testovanie trvalo 5 dní. V prijateľnom časovom okne sa vykonala iba klasifikácia pomocou Naive Bayes klasifikátora. Naive Bayes klasifikátore pridelila metóda forward selection váhu iba trom atribútom ako dostatočným pre klasifikáciu(pozri tab. 10). Výsledná hodnota správnosti bola 0,96 a presnost 0,9957, čo je nad očakávanie dobrý výsledok. Z troch vybraných atribútov mal vadprocess správnosť 0,95 a presnosť 1,0, čo bolo zrejme dôvodom, prečo algoritmus forward selection ostatné atribúty označil ako nevýznamné. Ďalšie dva atribúty iba mierne zlepšovali výsledok pre správnosť a presnosť. V druhom prípade pri klasifikátore W-Ridor, ktorý patrí medzi pravidlové (rule) klasifikátory, vylúčil iba 5 atribútov ako nevýznamné pre klasifikáciu. Vysledná správnosť bola 0,9455, ale presnosť bola iba 0,624. Na základe tohto môžeme konštatovať, že sme našli vhodné atribúty (ktoré samozrejme patria niektorej vlastnosti) na natrénovanie klasifikátorov. V praxi by ale trénovanie pomocou tohto datasetu zrejme nebolo možné alebo by si vyžadovalo značné výpočtové nároky. Preto sa s podrobnou analýzou výsledkov ďalej nebudeme zaoberať. Pri Naive Bayes klasifikátore ide o klasifikátor založený na pravdepodobnosti. Tu veľkú úlohu zohráva aj frekvencia výskytu jednotlivých hodnôt atribútov. Čím viackrát sa vyskytuje, tým sa zväčšuje pravdepodobnosť. Z tohto dôvodu použitie klasifikátora Naive Bayes budeme ešte podrobnejšie testovať.

Tretí dataset

Tretí dataset bol vytvorený na základe priebežných štatistických poznatkov o rozdielnych hodnotách pre jednotlivé vlastnosti a poznatkov o činnosti malware. Výsledky pri treťom datasete ukazujú, že testované vlastnosti poskytujú potrebné informácie pre klasifikáciu a detekciu malware. Pri váhovaní pomocou univariantných metód (pozri tab. 11) sa pod hranicou 0,5 vyskytli atribúty mutexcount, explorervadcount, servicesvadcount, smssvadcount, csrssvadcount, winlogonvadcount, lsassvadcount a svchostvadcount, ktoré prislúchajú sledovaným vlastnostiam mutex a VAD. Avšak atribút execute_readvadcount, prislúchajúci VAD vlastnosti, mal už váhu nad 0,6. Váhy pre ostatné atribúty boli pomerne vysoké, dokonca inloadfalseldrmodulecount a inmemfalseldrmodulecount atribút dosiahli hodnotu 1,0. Podobné výsledky boli dosiahnuté aj pomocou forward selection metódy testovania atribútov. Z výsledkov pri jednotlivých klasifikátoroch (pozri tab. v prílohe C) môžeme hodnotiť väščinu atribútov ako významné pre klasifikovanie.

[36] Pri testoch sme pridelili systému pokusne až 120 GB pamäte

Iba atribúty servicevadcount, svchostvadcount, serviceldrmodulecount a smssldrmodulecount neboli vybrané pri testovaní metódou forward selection pri žiadnom klasifikátore. Ostatné atribúty sa vyskytujú minimálne raz medzi navrhnutými atribútmi, pomocou ktorých sa dosiahli najlepšie výsledky pre správnosť a presnosť pri krížovej validácii (váha atribútu je 1,0). Pri porovnaní týchto výsledkov s výsledkami získanými pri váhovaní vidíme podobnosť v pridelení významnosti atribútom. Vadcount atribúty sú pre klasifikáciu málo významné. Je to vidieť aj pri klastrovaní (pozri obr. 32), kde vidíme vzájomné prelínanie medzi čistými programami a malware pri vadcount atribútoch. To je zrejme dôvod ich malej významnosti pre klasifikáciu. Pre ostatné atribúty je vidieť pri každom klasifikátore mierne odlišnosti vo výslednej hodnote správnosti, presnosti a váhy pri váhovaní. V každom prípade sa dosiahli dostatočné hodnoty, ktoré hovoria o veľkej miere významnosti atribútov a sledovaných vlastností pre správnu detekciu a klasifikáciu malware.

Významnosť práve týchto atribútov ukazuje na použitie DLL injekcie vo väčšine testovaných vzorkách malware. Z toho usudzujeme, že táto technika je v súčasnosti jedna z najviac používaných, alebo testovací malware dataset, ktorý sme mali k dispozícii, obsahoval vzorky malware, ktoré využívali pri svojej činnosti práve DLL injection techniku. V budúcnosti preto plánujeme vykonať ďalšie testy pri použití iných typov malware, aby sme vedeli vyhodnotiť závislosť významnosti atribútov na typoch malware.

Záver

Bezpečnosť a ochrana súkromia, a samozrejme aj majetku, boli vždy dôležitou oblasťou, do ktorej zabezpečenia sa investovalo veľa prostriedkov a ľudského potenciálu. V súčasnosti škodlivý kód (malware) sa stáva stále viac rozšíreným a hlavne používaným kriminálnou časťou našej spoločnosti k narušeniu súkromia, špionáži a krádeži financií a iných aktivít spoločnostiam a súkromným osobám. V tejto dobe informatizácie spoločnosti sa malware stal prakticky ich hlavnou zbraňou. Aktuálne štatistiky bezpečnostných firiem sú alarmujúce. Iba za jeden rok sa vytvorí niekoľko stoviek miliónov nových vzoriek malware. Z tohto dôvodu vzniká naliehavá potreba hľadať riešenie pre účinnú detekciu a zapojenie do výskumu aj akademické a vedecké inštitúcie. Do budúcnosti účinný spôsob detekcie môžu zabezpečiť iba automatizované systémy schopné samostatne identifikovať škodlivú činnosť. Prezentovaná práca má za cieľ zhodnotiť možnosti sledovania informácií z pamäte operačného systému pre detekciu malware a byť nápomocná práve k vytvoreniu takého automatizovaného systému detekcie.

Celková realizácia práce bola rozdelená do dvoch samostatných častí. Prvá časť sa venuje problematike prístupu k obsahu pamäte pre potreby detekcie malware a druhá časť hľadaniu vhodných informácií z pamäte pre identifikáciu útoku alebo prítomnosti malware.

V pamäti počítača sa nachádzajú systémové a užívateľské informácie. Prístup k týmto informáciam má hlavne procesor. Návrh súčasných systémov nepočíta s podporou plného prístupu k pamäti pre iné zariadenia, preto sa pri snahe čítania obsahu pamäte používa väčšinou softvérový prístup (využívajúci procesor). Malware však v súčasnosti môže operovať na rovnakej úrovni ako operačný systém. Preto s využitím pokročilých techník môže ovplyvniť výsledok čítania obsahu pamäte. Viac sofistikované rootkity sú schopné hooknuť Virtual Memory Manager (VMM), ako napr. Shadow Walker môže presmerovať čítanie na iný blok pamäte alebo zmeniť obsah čítania (viac v časti 3.3 pri popise techniky pre zámenu obsahu pamäte). Po našej analýze dostupných riešení pre čítanie obsahu sme zistili, že žiadne riešenie nevyhovuje požiadavkám potrebným pre využitie k detekcii malware. Preto bolo potrebné navrhnúť a vytvoriť vlastné riešenie, ktoré by sme mohli využiť aj pre detekciu malware. Aby sme splnili požiadavky zo súčasne dostupných možností pre získanie obsahu pamäte, museli sme vylúčiť softvérové riešenia, ktoré môžu byť malwarom operujúcim na úrovni operačného systému kompromitované. Ako najvhodnejšie sa javilo riešenie s využitím DMA sieťovej karty, pomocou ktorej môžeme čítať obsah pamäte a následne dáta odosielať cez sieťovú kartu na iný (nami určený) počítač. DMA prenos nie je riadený procesorom, preto prenášané dáta nemôžu byť modifikované malware (rootkitom), ktorý získal systémové práva a má možnosť využívať plnú funkcionalitu procesora.

Aby sme overili správnosť nášho riešenia čítať obsah celej pamäte alebo vybranej oblasti, vykonali sme niekoľko testov. Jedným z nich bolo vytvorenie kópie celej pamäte (dump) s použitím nášho riešenia a porovnanie s dumpom pamäte vytvoreného pomocou nástroja "MDD tool". Získané kópie boli zhodné. Po testovaní sme tiež analyzovali nami vytvorený obraz pamäte v programe Volatility Framework. Obraz bol Volatility Frameworkom rozpoznaný a plne funkčný. Tým sme overili, že naše riešenie je použiteľné aj pre digitálnu forenznú analýzu a uložené dáta sú v tvare rozpoznateľnom nástrojmi pre analýzu obsahu pamäte.

Ako nevýhodu daného riešenia môžeme považovať potrebu inštalácie ovládača do OS pred tým, ako chceme získať obraz pamäte. To vyžaduje získanie administrátorských práv a prístupu k počítaču, čo ale pri legitímnom využívaní riešenia nie je obmedzujúce. Preto táto potreba môže byť zároveň ochrana, ktorá zabraňuje zneužitiu riešenia. Ovládač je možné nastaviť tak, aby ho stačilo nainštalovať iba raz (a aby sa automaticky spustil aj po reštarte systému). Ako ideálne riešenie by bola implementácia nášho ovládača priamo cez výrobcu do operačného systému. Ďalšou potenciálnou nevýhodou nášho riešenia je možnosť hooknutia NDIS ovládača alebo inej manipulácie sofistikovaným rootkitom priamo v pamäti. Preto sme pre zabezpečenie ochrany ovládača vyvinuli nástroj, ktorý dokáže sledovať nízkoúrovňovo každý pokus o zápis a čítanie v sledovanej oblasti pamäte a vykonať reakciu v prípade pokusu o zmenu údajov.

Použitie systémového ovládača prináša niektoré výhody pre naše riešenie. Popri jeho využití na riadenie procesu získania obsahu pamäte sme zároveň schopní z pamäte získať adresy dôležitých systémových štruktúr a tabuliek. Po získaní počiatočnej adresy a jej veľkosti dokážeme danú oblasť pamäte odoslať na vzdialený počítač pre podrobnú analýzu. Takým spôsobom môžeme testovať integritu rôznych štruktúr a systémových tabuliek (napr. SSDT). Samozrejme týmto možnosti nástroja nekončia. Možnosť sledovania rôznych systémových štruktúr a objektov alebo obsahu celej pamäte nám dáva možnosť veľkej variability využitia. Pri zohľadnení výsledkov testov a možností využitia môžeme považovať naše riešenie za potenciálne vhodné pre potreby detekcie aj sofistikovaného a pokročilé techniky schovávania používaného malware. Jednou z variant nasadenia môže byť aj integrácia riešenia do agenta pre zber dát z operačného systému pri komplexnejších bezpečnostných riešeniach.

Z dôvodu prítomnosti aj rôzných typov iných informácií v pamäti v budúcnosti považujeme za potrebné riešiť aj otázku zneužitia riešenia pre prístup k citlivým informáciam. ale na druhej strane aby bolo nápomocné pri forenznej analýze a získavaní digitálnych dôkazov. Tým sa problematika získania obsahu pamäte mení na komplexnejší problém. Napríklad získanie šifrovacieho kľúča priamo z pamäte neoprávnenou osobou môže viesť k narušeniu súkromia a bezpečnosti, ale v prípade jej získania pri forenznej analýze môže byť nápomocné k odhaleniu kriminálnych

aktivít. Problematike získavania šifrovacích kľúčov priamo z pamäte sme sa venovali v inej publikovanej a v súčasnosti už aj viackrát citovanej práci (Balogh, 2011) a tiež v rámci celej kapitoly v knihe Multidisciplinary Perspectives in Cryptology and Information Security (Balogh, 2014).

V druhej časti práce sme sa zamerali na hľadanie informácií, ktoré môžu byť vhodné atribúty (vlastnosti) pre algoritmy strojového učenia. Automatická heuristická detekcia využíva hlavne algoritmy strojového učenia a datamining klasifikátory ku klasifikácii novej vzorky. Podľa nás v súčasnom stave nárastu počtov malware je automatická detekcia jediná zmysluplná cesta pre hľadanie riešenia pre detekciu malware. V mnohých prácach sa zaoberali možnosťami využitia algoritmov strojového učenia na klasifikáciu malware. Výsledky sú však v značnej miere závislé na výbere vlastností pre dané algoritmy. Výber vhodných vlastností pre detekciu malware sa ukazuje ako netriviálny problém z dôvodu, že tvorcovia malware môžu podľa potreby meniť techniky a postupy použité pri činnosti malware. Ak pre analýzu vyberieme vlastnosti, ktoré je možné jednoducho meniť alebo činnosť zameniť za inú bez toho, aby to ovplyvnilo funkcionalitu, daný systém je málo účinný. Modifikáciou sledovaných vlastností môže malware zabrániť správnej klasifikácii. Metódy tak ostávajú funkčné len pre malware, z ktorých sme mali vhodnú vzorku na trénovanie.

Ako hypotetické riešenie sme navrhli hľadanie takých vlastností, ktoré nie sú priamo získavané zo vzoriek malware, sú závislé na systéme a nie je možné ich ľahko modifikovať bez narušenia funkčnosti malware. Medzi také vlastnosti môžu patriť aj niektoré systémové informácie získané z pamäte počítača, čo nás motivovalo k hľadaniu takých informácií (vlastností) a zároveň k vytvoreniu metodiky a testovacieho prostredia na overenie významu nájdených informácií pre správnosť klasifikácie pomocou algoritmov strojového učenia. Na základe poznatkov o v súčasnosti používaných technikách malware sme ako informácie pre sledovanie v pamäti navrhli tieto systémové objekty: VAD štruktúry, atomy, mutexy, záznamy v zozname modulov, spätné volania (callbacks), timery a sokety.

Podľa vytvorenej metodiky sme vykonali testovanie pomocou testovacieho prostredia, ktoré bolo potrebné zabezpečiť tak, aby spĺňalo požiadavky na bezpečnosť a automatické vykonávanie testov. V prvej časti testov sme vykonali zber dát (nami vybraných informácií) z pamäte počítača pri spustení testovacích súborov.

V druhej časti testov sme sledovali ich významnosť pre správnosť a presnosť klasifikácie pre vybrané algoritmy strojového učenia a datamining klasifikátory. Pre overenie významnosti vlastností pre detekciu malware sme vytvorili testovací dataset, ktorý obsahoval testované vlastnosti a niektoré ich parametre. Výpočet významnosti bol realizovaný aplikovaním forward selection metódy na výber vhodných vlastností pre datamining klasifikátory a s využitím výsledkov váhovania

pomocou univariantných metód. Na testovanie sme použili IBM bladeCenter HS23 s 8 CPU a 128GB RAM a s inštalovaným OS Debian verzia 7.

Hodnotenie významnosti atribútov na základe váhovania a forward selection metód ukazuje na významnosť atribútov, ktoré ukazujú na použitie DLL injekcie vo väčšine testovaných vzorkách malware. Z toho usudzujeme, že táto technika je v súčasnosti jedna z najviac používaných, alebo že testovací malware dataset, ktorý sme mali k dispozícii obsahoval vzorky malware, ktoré využívali pri svojej činnosti práve DLL injekciu. To nás priviedlo k myšlienke vytvorenia v ďalšom budúcom výskume testy, ktoré by zisťovali úroveň závislosti významnosti atribútov na typoch malware.

Pomocou vykonaných testov sme vybrali vhodné atribúty (podľa významnosti) pre datamining klasifikátory na základe výsledkov validácie a hodnôt správnosti a presnosti. Pre nás sú tieto výsledky ukazovateľom, že testované vlastnosti poskytujú potrebné informácie pre klasifikáciu a detekciu malware. Nehovoria však o správnosti detekcie a klasifikácie na základe nich naučeného systému, preto nevieme porovnať naše riešenie v tomto štádiu s inými výsledkami detekčných nástrojov. K tomu by sme potrebovali vytvoriť komplexné riešenie pre zber dát a ich klasifikáciu, čo nebolo cieľom tejto práce, a potom testovať úspešnosť detekcie s použitím navrhnutých atribútov. Predtým, ako také riešenie vytvoríme, ale chceme testovat závislosť významnosti atribútov pre jednotlivé typy mlaware, aby sme vedeli vybrať atribúty, ktoré pokrývajú čo najväčšie spektrum typov malware. Výsledky práce však môžu tvoriť základ pre vytvorenie takého riešenia.

Z dostupných prác nie je známe získavanie a využitie nami sledovaných atribútov (informácií) získavaných priamo z pamäte alebo podobných informácií, ako vstup pre datamining algoritmy alebo automatické systémy pre detekciu malware. V tomto je naša práca inovatívna. Dôvodom bola možno aj zložitosť sledovania vlastností v pamäti pri reálnom nasadení. V práci sa nám podarilo zlepšiť tieto možnosti pre reálne nasadenia.

Výsledok práce tým môže prispieť k vývoju účinných senzorov na detekciu rôznych vektorov útoku, na systém a jadro sytému (zmena integrity systémových tabuliek, zmeny v dôležitých štruktúrach systému, zmena integrity ovládačov atď.). Navrhnuté vlastnosti a výsledky testovania ukazujú tiež na možnosť tieto vlastnosti monitorovať a úspešne využiť pre detekciu napadnutia systému. Tým môžeme pokryť sledovanie väčšej škály správania sa typického pre malware, a tak zlepšiť úspešnosť jeho detekcie. Ako bolo už viackrát uvedené, obrovský nárast počtu malware núti použiť ako jediný možný prístup automatizované nástroje pre detekciu. Táto práca môže byť prínosom v tomto úsilí.

Literatúra

Abbasi, A. et al. 2011. *Selecting Attributes for Sentiment Classification Using Feature Relation Networks*. In *Ieee Transactions on Knowledge and Data Engineering*. 2011, vol. 23, p. 447-462.

Antoniewicz, B. 2013. *Windows DLL Injection Basics, Open Security Research, Foundstone* [online]. 2013 [cit. 2014-09-20]. Dostupné na internete: <http://blog.opensecurityresearch.com/2013/01/windows-dll-injection-basics.html>.

Balogh, Š. 2008. *Forenzné nástroje na zbieranie digitálnych dôkazov* : diplomová práca. Bratislava : FEI STU, 2008. 70 s.

Balogh, S., Pondelik, M. 2011. *Capturing encryption keys for digital analysis*. In *Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2011 IEEE 6th International Conference on* (Vol. 2, pp. 759-763). IEEE.

Balogh, Š., Mydlo, M. 2013. *New Possibilities for Memory Acquisition by Enabling DMA Using Network Card*. In *IDAACS 2013: 7th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems*. Berlin (Germany), September 12-14, 2013. Piscataway : IEEE, 2013. ISBN 978-1-4799-1426-5. Vol. 2, p. 635-639.

Balogh, Š. 2014. *Forensic Analysis, Cryptosystem Implementation and Cryptology: Methods and Techniques for Extracting Encryption Keys from Volatile Memory*. In: *Multidisciplinary Perspectives in Cryptology and Information Security. - Hershey : IGI Global*, 2014. - ISBN 978-1-4666-5808-0. - S. 381-396

Bazrafshan, H., Hashemi, H., Mehdi, S. 2013. *A Survey on Heuristic Malware Detection Techniques*. 5th Conference on Information and Knowledge Technology. 2013

Binsalleeh, Hamad et al. 2010. *On the analysis of the zeus botnet crimeware toolkit*. Privacy Security and Trust (PST), 2010 Eighth Annual International Conference on. IEEE, 2010.

Boileau, A. 2006. *Hit By A Bus: Physical Access Attacks with Firewire Security*. Assessment.com, Ruxcon. 2006. [cit. 2014-07-25]. Dostupné na internete: <http://www.storm.net.nz/static/files/ab_firewire_rux2k6-final.pdf>.

Blunden, B. 2013. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Jones & Bartlett Publishers.

Bruschi, D., Martignoni, L., Monga, M. 2006. *Detecting self-mutating malware using control-flow graph matching*. In *Detection of Intrusions and Malware & Vulnerability Assessment*. p. 129-143. Springer Berlin Heidelberg.

Butler, J., Sparks, S. 2005. *Shadow walker raising the bar for rootkit detection. Black Hat Briefings and trainning*. Japan, 2005.

Carrier, B., Grand, J. 2004. *A Hardware - Based Memory Acquisition Procedure for Digital Investigations*. In *Digital Investigation Journal*. February 2004.

Carvajal, L., Varol, C., Chen, L. 2013. May. *Tools for collecting volatile data: A survey study*. In *Technological Advances in Electrical, Electronics and Computer Engineering (TAEECE)*. 2013. International Conference. p. 318-322. IEEE.

Carvey, H. 2009. *Windows Forensic Analysis DVD Toolkit*. 2. Edition. Syngress. June 11, 2009. ISBN-13: 978-1597494229, ASIN: 1597494224.

Cohen, M. 2011. *PMEM – physical memory driver*. Dostupné na internete: <http://code.google.com/p/volatility/source/browse/branches/scudette/tools/linux>. 2011

Dai, J., Guha, R., Lee, J. 2009. *Efficient Virus Detection Using Dynamic Instruction Sequences*. In *Journal of Computers* [online]. May 2009, vol. 4, no. 5, p. 405-414. Dostupné na internete: <doi:10.4304/jcp.4.5.405-414>.

Daoud, E. A., Jebril, I. H., Zaqaibeh, B. 2008. *Computer Virus Strategies and Detection Methods*. In *Open Problems Compt. Math*. Vol. 1, no. 2, September 2008.

Davis, M.A., Bodmer, S. LeMaster, A. 2009. *Kernel Mode Rootkits. Hacking Exposed Malware & Rootkits: Malware & Rootkits Secrets & Solutions*. McGraw-Hill Osborne, 2009. ISBN-13: 978-0071591188.

Delugré, Guillaume. 2010. *Closer to metal: Reverse engineering the Broadcom NetExtreme's firmware*. In *Sogeti ESEC Lab* [online]. 2010 [cit. 2014-07-16]. Dostupné na internete: <http://esec-lab.sogeti. com/dotclear/public/publications/10-hack.lu-nicreverse_slides.pdf>.

Dolan-Gavitt, B. 2007. *The VAD tree: A process-eye view of physical memory.* In *Digital investigation* [online]. vol. 4, p. 62-64.
Dornseif, M. 2004. *Owned by an iPod Laboratory for Dependable Distributed Systems*. PacSec 2004. [cit. 2014-05-30]. Dostupné na internete: <http://md.hudora.de/presentations/firewire/PacSec2004.pdf.>.

Duflot, L., Perez, Y., Valadon, G. a Levillain, O. 2010. *Can you still trust your network card?*. In *Agence nationale de la sécurité des systèmes d'information* [online]. 2010 [cit. 2014-05-16]. Dostupné ne internete: <http://www.ssi.gouv.fr/IMG/pdf/csw-trustnetworkcard.pdf>.

Elhadi, A. A., Maarof, M. A., Osman, A. H. 2012. *Malware detection based on hybrid signature behaviour application programming interface call graph*. In *American Journal of Applied Sciences*. 9(3), p.283.

Ferrie, P. 2008. *Anti-unpacker tricks–part one*. In *Virus Bulletin*. 4.

Garcia, G. 2007. Forensic physical memory analysis : *Overview of tools and techniques, Telecommunications Software and Multimedia Laboratory* [online]. Helsinki (Finland) : Helsinki University of Technology, 2007 [cit. 2014-09-15]. Dostupné na internete: <www.tml.tkk.fi/Publications/C/25/papers/Limongarcia final.pdf .>.

Goldstine, H. H. 1972. *The computer from Pascal to von Neumann*. Princeton University Press.

Golomb, G. 2011. *Mutexes, part one: The Canary in the Coal Mine and Discovering New Families of Malware* [online]. 2011 [cit. 2014-09-15]. Dostupné na internete: <http://resources.infosecinstitute.com/mutexes-analysis-part-one/>.

Gutmann, P. 2007. *The commercial malware industry*. In *DEFCON conference*. August 2007.

Hoglund, G., Butler, J. 2005. *Rootkits: Subvering the Windows Kernel*. USA : Addison-Wesley Longman, 2005. 352 p. ISBN 0-321-29431-9.

Ivaniš, D. 2013. *Monitor integrity jadra OS systému*: diplomová práca. FEI STU. 2013, e.č. FEI-5384-50542.

Ivaniš, D. 2011. *Detekcia škodlivého kódu pomocou zachytenia hook mechanizmu*: bakalárska práca. FEI STU. 2011, e.č. FEI-5382-50542.

InSeon, Yoo. 2004. *Visualizing windows executable viruses using self-organizing maps.* In *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, 2004, p. 82–89.

Jacob, G., Debar, H., Filiol, E. 2008. *Behavioral detection of malware: from a survey towards an established taxonomy*. In *Journal in computer Virology*. 4(3), p.251-266.

Jalote, P. 1997. *An integrated approach to software engineering*. Springer.

Kang, M. G., Poosankam, P., Yin, H. 2007. *Renovo: A Hidden Code Extractor for Packed Executables*. Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM). October 2007.

Kaplan, B. 2007. *RAM is Key, Extracting Disk Encryption Keys From Volatile Memory*. Advisor: Matthew Geiger, Carnegie Mellon University. May 2007.

Kapoor, A., Mathur, R. 2011. *Predicting the future of stealth attacks*. In *Virus Bulletin conference*. October 2011.

Kdm..2004. NTIllusion: a portable Win32 userland rootkit. Phrack July 2004;11(62)

Kephart, J. O., Arnold, B. 1994. *Automatic extraction of computer virus signatures*. In Proceedings of the 4th Virus Bulletin Internation Conference. 1994, p. 178–184.

Kolter, J. Z., Maloof, M. A. 2004. *Learning to detect malicious executables in the wild*. In Proceedings of the 2004 ACM SIGKDD. International Conference on Knowledge Discovery and Data Mining, 2004.

Kováč, M., 2004. *História počítačových vírusov a červov*. In *PC Revue*. 9/2004.

Libster, E., Kornblum, J.D. 2008. *A proposal for an integrated memory acquisition mechanism*. In *ACM SIGOPS Operating Systems Review*. Vol. 42. Issue 3, April 2008.

Ligh, M., Adair, S., Hartstein, B., Richard, M. 2010. *Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code.* Wiley Publishing. ISBN-13: 978-0470613030.

Egele, M., Scholte, T., Kirda, E., Kruegel, C. 2012. *A survey on automated dynamic malware-analysis techniques and tools.* In *ACM Computing Surveys (CSUR)*. 44(2), 6.

Forman, G. 2003. *An extensive empirical study of feature selection metrics for text*

Classification. In *J. Mach. Learn. Res*. 2003, vol. 3, p. 1289-1305.

Mathur, R. 2011. *Memory Forging Attempt by a Rootkit.* In *McAfee Labs Blog* [online]. 21 April 2011. Dostupné na internete: <http://blogs.mcafee.com/mcafee-labs/memoryforging-attempt-by-a-rootkit>.

McCabe, T. J. 1976. *A complexity measure.* In *Software Engineering*. IEEE Transactions on, (4), p. 308-320.

Microsoft Corporation, 2010. *Microsoft Corporation. Device physicalmemory object*. [online]. Dostupné na internete: <http://technet.microsoft.com/de-de/library/cc78756528WS.1029.aspx>.

Mydlo, M. 2012. *Priamy prístup do pamäte* : diplomová práca. FEI STU.

Mohammad, L. K., Masud, M., Thuraisingham, B. 2007. *A scalable multi-level feature extraction technique to detect malicious executables*. In *Information Systems Frontiers*. 2007.

Rieck, K., Holz, T., Willems, C. et al. 2008. *Learning and classification of malware behavior*. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Berlin Heidelberg : Springer, 2008. p. 108-125.

Russinovich, M. 1999. *New Win2K RC Kernel APIs*. In *The Systems Internals Newsletter*. 1999, 1:5.

Rutkowska, J. 2006. *Rootkits vs. Stealth by Design Malware*. Amsterdam : Black Hat Europe, March 2006, 2.

Odehnal, P., Zahradníček, P. 1996. *Praktická sebeobrana proti virům*. Praha : Grada Publishing, spol. s r. o., 1996.

Petroni, N. L. Jr., Fraser, T., Molina, J. et al. 2004. *Copilot- a coprocessor-based kernel runtime integrity monitor*. In *SSYM'04*. Proceedings of the 13th conference on USENIX Security Symposium. Berkeley, CA (USA) : USENIX Association, 2004, p. 13.

Skape, S. 2007. *KiDebugRoutine.* In *Uninformed* [online].

September 2007. Dostupné na internete: <http://uninformed.org/index.cgi?v=8&a=2&p=16>.

Siddiqui, M., Wang, M. C., Lee, J. 2008. *A Survey of Data Mining Techniques for Malware Detection*. Proceedings of the 46th Annual Southeast Regional Conference, ACM. 2008, p. 509-510.

Sikorski, M., Honig, A. 2012. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012. 800 p. ISBN-10: 1593272901.

Shabtai, A., Moskovitch, R., Feher, C., et al. 2012. *Detecting unknown malicious code by applying classification techniques on OpCode patterns*. In *Security Informatics*. 1(1), p.1-22.

Shields, T. 2008. *Anti-Debugging Series - Part I. Veracode* [online]. [cit. 2014-02-27]. Dostupné na internete: <http://www.veracode.com/blog/2008/12/anti-debugging-series-part-i/>.

Schultz M. G., Eskin, E., Zadok, E. et al. 2001a. *Data Mining Methods for Detection of New Malicious Executables*. In Proceedings of the IEEE Symposium on Security and Privacy, 2001, p. 38–49.

Schultz M. G., Eskin, E., Zadok, E. et al. 2001b. *MEF: Malicious Email Filter: A UNIX Mail Filter That Detects Malicious Windows Executables*. 2001, p. 245–252.

Suiche, Matthieu. *2008. Microsoft crash dump analysis weaknesses*. In *Matthieu Suiche's blog* [online]. 2008 [cit. 2014-05-16]. Dostupné na internete: <http://www.msuiche.net/2008/10/16/microsoft-crash-dump-analysis-weaknesses/>.

Sung A. H., Xu, J., Chavez, P. et al. 2004. *Static Analyzer of Vicious Executables*. In 20th Annual Computer Security Applications Conference. 2004, p. 326–334.

Stüttgen, J., Cohen, M. 2013. *Anti-forensic resilient memory acquisition Digital Investigation*. Elsevier, 2013.

Sylve J. LiME. 2012. *Linux memory extractor*. In *ShmooCon'*. 12/2012. Vidstrom A. Forensic memory dumping intricacies PhysicalMemory, DD,and caching issues. Dostupné na internete: http://ntsecurity.nu/onmymind/2006/2006-06-01.html; 2006.;

Symantec. Executive Summary [online]. April 2009, vol. XIV, [cit. 2014-09-05]. Dostupné na internete: <http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xiv_04-2009.en-us.pdf.>.

Symantec. Executive Summary [online]. April 2010, vol. XV, [cit. 2014-09-05]. Dostupné na internete: <http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xv_04-2010.en-us.pdf.>.

Symantec. Executive Summary [online]. April 2011, vol. XIV, [cit. 2014-09-05]. Dostupné na internete: <http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xiv_04-2009.en-us.pdf.>.

Symantec. Executive Summary [online]. April 2012, vol. XV, [cit. 2014-09-05]. Dostupné na internete: <http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xv_04-2010.en-us.pdf.>.

Szor, P. 2006. *Počítačové viry, analýza útoku a obrana*. Brno : Zoner Press, 2006.

Szor, P., Ferrie, P. 1998. *Hunting for metamorphic*. In Virus Bulletin Conference. Prague (Czech Republic) : Vecna, 1998. Miss Lexotan, 8, 29A E-Zine (2001)

Tesauro, G., Kephart, J. O., Sorkin, G. B. 1996. *Neural Networks for Computer Virus Recognition*. IEEE Expert, 11(4):5-6. IEEE Computer Society, August, 1996.

Ye, Y. et al. 2007. *Intelligent malware detection system*. In *KDD '07*. Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2007, p. 1043–1047.

Ye, Y., Li, T., Jiang, Q. et al. 2010. *CIMDS: adapting postprocessing techniques of associative classification for malware detection*. IEEE Trans. Syst., Man, Cybern. C, 2010, vol. 40, no. 3, p. 298-307.

Jeong, K., Lee, H. 2008. *Code graph for malware detection*. In *Information Networking*. ICOIN 2008. International Conference on p. 1-5. IEEE.

Tan, L. 2006. *The worst case execution time tool challenge 2006: The external test*. In *Leveraging Applications of Formal Methods, Verification and Validation*. November 2006. ISoLA 2006. Second International Symposium on, p. 241-248. IEEE.

Zhao, Z., Wang, J., Wang, C. 2013. *An unknown malware detection scheme based on the features of graph*. In *Security and Communication Networks*. 6(2), p. 239-246.

Zhao, Z. 2011. *A virus detection scheme based on features of Control Flow Graph*. In *Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC*). 2011, 2nd International Conference on, p. 943-947. IEEE.

Zima, M. 2014. *Priamy prístup do pamäte OS* : bakalárska práca. FEI STU. 2014, ev. č. FEI-5382-6328.

Zvara, P. 2014. *Detekcia techník používaných škodlivým kódom na ochranu pred detekciou* : diplomová práca. FEI STU. 2014, ev.č. FEI-5384-35645.

Walters, A., Petroni, N. 2007. *Volatools: Integrating volatile memory forensics into the digital investigation process* [online]. Dostupné na internete: <https://www.blackhat.com/presentations/bh-dc-07/Walters/Presentation/bh-dc-07-Walters-up.pdf>.

Wang K. W. Li, Stolfo, S., Herzog, B. 2005. *Fileprints: Identifying File Types by n-gram Analysis.* In 6th IEEE Information Assurance Workshop. 2005.

Wang, J., Zhang, F., Sun, K. et al. 2011. *Firmware-assisted Memory Acquisition and Analysis tools for Digital Forensics (SADFE)*. 2011, IEEE Sixth International Workshop.

Weber, M., Schmid, M., Schatz, M. et al. 2002. *A toolkit for detecting and analyzing malicious software*. In Computer Security Applications Conference, 2002. Proceedings. 18th Annual, p. 423-431. IEEE.

Wieland, Peter. 2006a. *What is DMA: Common Buffer*. In *MSDN* [online]. 2006 [cit. 2014-05-16]. Dostupné na internete: <http://blogs.msdn.com/b/peterwie/archive/2006/03/09/546961.aspx>.

Wieland, Peter. 2006b *What is DMA: Paket Based DMA*. In: *MSDN* [online]. 2006 [cit. 2014-05-16]. Dostupné na internete: <http://blogs.msdn.com/b/peterwie/archive/2006/03/20/555597.aspx>.

Wilhelm, R., Engblom, J., Ermedahl, A. et al. 2008. *The worst-case execution-time problem – overview of methods and survey of tools*. ACM Transactions on Embedded Computing Systems (TECS), 7(3), 36.

Publikačná činnosť doktoranta

ABC Kapitoly vo vedeckých monografiách vydané v zahraničných vydavateľstvách
Počet záznamov: 1
ABC1 Balogh, Štefan: Forensic Analysis, Cryptosystem Implementation and Cryptology: Methods and Techniques for Extracting Encryption Keys from Volatile Memory. In: Multidisciplinary Perspectives in Cryptology and Information Security. - Hershey : IGI Global, 2014. - ISBN 978-1-4666-5808-0. - S. 381-396, [1.46 AH]

ADF Vedecké práce v domácich nekarentovaných časopisoch
Počet záznamov: 1
ADF1 Balogh, Štefan: Metódy detekcie vírusov a škodlivého kódu.
In: EE časopis pre elektrotechniku, elektroenergetiku, informačné a komunikačné technológie. - ISSN 1335-2547. - Roč. 16, mimoriadne č. : ELOSYS. Trenčín, 5.-8.10.2010 (2010), s. 11-16

AFA Publikované pozvané príspevky na zahraničných vedeckých konferenciách
Počet záznamov: 1
AFA1 Lehocki, Fedor - Balogh, Štefan - Kováč, Miroslav - Žákovičová, Eva - de Witte, Bart: Innovative Telemedicine Solutions for Diabetic Patients.
In: 2012 IEEE-EMBS Conference on Biomedical Engineering (IECBES 2012) : Langkawi, Malaysia, 17-19th December 2012. - Piscataway : IEEE, 2013. - ISBN 978-1-4673-1666-8. - S. 203-208

AFC Publikované príspevky na zahraničných vedeckých konferenciách
Počet záznamov: 4
AFC1 Balogh, Štefan - Pondelík, Matej: Capturing Encryption Keys for Digital Analysis.
In: IDAACS 2011 : 6th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications. Prague, Czech Republic, 15 - 17 September 2011. - Piscataway : IEEE, 2011. - ISBN 978-1-4577-1425-2. - S. 759-763

AFC2 Balogh, Štefan - Lehocki, Fedor - Ivaniš, Daniel - Kučera, Erik - Lajtman, Miloš - Miňo, Igor: Data Processing from mHealth Patient Data Acquisition Related to Extracting Structured Data from EH Records.
In: Wireless Mobile Communication and Healthcare : 3rd International Conference MobiHealth 2012. Paris, France, November 21-23. 2012. - Berlin Heidelberg : Springer, 2013. - ISBN 978-3-642-37892-8. - S. 255-262

AFC3 Balogh, Štefan - Mydlo, Miroslav: New Possibilities for Memory Acquisition by Enabling DMA Using Network Card.

In: IDAACS 2013 : 7th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems. Berlin, Germany, September 12-14, 2013. - Piscataway : IEEE, 2013. - ISBN 978-1-4799-1426-5. - Vol. 2, p. 635-639

AFC4 Balogh, Štefan - Mydlo, Miroslav: New Possibilities for Reading Memory Contents by Enabling DMA Using NDIS Driver.
In: Memics 2012. 8th Doctoral Workshop on Matematical and Engineering Methods in Computer Science. Znojmo, October 25-28, 2012. - Brno : Novopress, 2012. - ISBN 978-80-87342-15-2. - S. 12 s

AFD Publikované príspevky na domácich vedeckých konferenciách
Počet záznamov: 5
AFD1 Balogh, Štefan - Skokan, Lukáš: Návrh architektúry pre dynamickú analýzu kódu.
In: ELOSYS. Elektrotechnika, informatika a telekomunikácie 2012 [elektronický zdroj] : Trenčín, 9.-12.10.2012. - Bratislava : FEI STU, 2012. - ISSN 1335-2547. - S. 126-130

AFD2 Balogh, Štefan - Chovančák, Dušan: Použitie dataminig metód pri odhaľovaní nových typov malware a pri sieťových útokoch.
In: Znalosti 2011 : 10. ročník konference, Stará Lesná, Vysoké Tatry, 31. ledna - 2.února 2011. Sborník příspěvků. - Ostrava : Fakulta elektrotechniky a informatiky, VŠB - Technical University of Ostrava, 2011. - ISBN 978-80-248-2369-0. - S. 255-258

AFD3 Balogh, Štefan - Varga, Juraj: Threats in Mobile Security.
In: ELITECH´12 [elektronický zdroj] : 14th Conference of Doctoral Students. Bratislava, Slovak Republic, 22 May 2012. - Bratislava : Nakladateľstvo STU, 2012. - ISBN 978-80-227-3705-0. - CD-ROM, [6] s.

AFD4 Balogh, Štefan: Using Memory Analysis for Malware Detection.
In: ELITECH´13 [elektronický zdroj] : 15th Conference of Doctoral Students; Bratislava, Slovakia, 5 June 2013. - Bratislava : Nakladateľstvo STU, 2013. - ISBN 978-80-227-3947-4. - CD-ROM, [6] s.

AFD5 Pondelík, Matej - Balogh, Štefan: Capture of Encryption Keys by Analyzing Memory.
In: ELITECH´10 : 12th Conference of Doctoral Students. Bratislava, Slovak Republic, 26.5.2010. - Bratislava : STU v Bratislave, 2010. - ISBN 978-80-227-3303-8. - CD-Rom

ADE Vedecké práce v zahraničných nekarentovaných časopisoch
Počet záznamov: 1

ADE1 Balogh, Štefan:Memory Acquisition by Using Network Card.
In: Journal of Cyber Security and Mobility (pages 65–76), Vol: 3 Issue: 1, January 2014, ISSN: 2245-1439 (Print Version), ISSN: 2245-4578 (Online Version), doi: 10.13052/jcsm2245-1439.314

BED Odborné práce v domácich recenzovaných zborníkoch (konferenčných aj nekonferenčných)
Počet záznamov: 2
BED1 Kubačka, Slavomír - Balogh, Štefan: Electronic Health Records.
In: Meditech - Proceedings of the ESF Project Conference : Innovative Program of Modern Biomedical Technologies. Project No. SORO/JPD-26/2005. Bratislava, Slovakia, 26.5.2008. - Bratislava : STU v Bratislave, 2008. - ISBN 978-80-227-2881-2. - CD-Rom

BED2 Pavlovič, Andrej - Lehocki, Fedor - Balogh, Štefan: Security of Electronic Health Records.
In: Meditech - Proceedings of the ESF Project Conference : Innovative Program of Modern Biomedical Technologies. Project No. SORO/JPD-26/2005. Bratislava, Slovakia, 26.5.2008. - Bratislava : STU v Bratislave, 2008. - ISBN 978-80-227-2881-2. - CD-Rom

BEF Odborné práce v domácich nerecenzovaných zborníkoch (konferenčných aj nekonferenčných)
Počet záznamov: 2
BEF1 Balogh, Štefan - Lehocki, Fedor - Juhás, Gabriel: Ehealth na Slovensku a jeho implementácia.
In: SMART S2AI : Workshop SMART systémov a služieb v oblasti aplikovanej informatiky. Bratislava, 18. apríla 2011. - Bratislava : STU v Bratislave FEI, 2011. - ISBN 978-80-227-3513-1. - S. 11-13

BEF2 Lehocki, Fedor - Oravec, Miloš - Balogh, Štefan - Juhásová, Ana: Vybrané modely diagnostických systémov.
In: SMART S2AI : Workshop SMART systémov a služieb v oblasti aplikovanej informatiky. Bratislava, 18. apríla 2011. - Bratislava : STU v Bratislave FEI, 2011. - ISBN 978-80-227-3513-1. - S. 1-5

Citácie doktoranta

K článku:

Balogh, S., & Pondelik, M. (2011, September). Capturing encryption keys for digital analysis. In *Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2011 IEEE 6th International Conference on* (Vol. 2, pp. 759-763). IEEE.

1. Bashir, M. S., & Khan, M. N. A. (2013). Triage in Live Digital Forensic Analysis. *International journal of Forensic Computer Science*, *1*, 35-44.

2. Youn, J. H., & Ji, Y. K. (2013). Secure USB using Reconstructed File Stricture. In *Proceedings of ICCA* (pp. 27-29).

3. Abbas, A., Rathje, C. A., Wienbrandt, L., & Schimmler, M. (2012, December). Dictionary Attack on TRUECRYPT with RIVYERA S3-5000. In *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems* (pp. 93-100). IEEE Computer Society.

4. Youn, J. H., & Ji, Y. K. (2014). Secure USB Design Using Unallocated Arbitrary Area. *International Journal of Security & Its Applications*, *8*(2).

Priloha A

Kategórie funkcií zistených pre malware vykonávajúce APT útoky firmou Mandiant podľa Appendix C: The Malware Arsenal
http://intelreport.mandiant.com/Mandiant_APT1_Report_Appendix.zip

Function category	Description
Capture keystrokes	Record what the user types
Capture mouse movement	Record how the user is moving the mouse
Change directories	Change the current working directory
Close connection	Close a network connection
Create processes	Run programs
Create/modify files	Create, modify or delete files or directories
Download and execute file [from specified URL]	Download and execute a file from a specified address
Download file [from specified URL]	Download a file from a specified address
Enumerate files	Gather information about files or directories
Enumerate systems	Gather information about other systems in the network
Enumerate users	Gather information about user accounts
Establish connection	Create a network connection to another system
Exit	Stop the backdoor from running
File upload	Transfer a file from the victim system to the C2 server
File download	Transfer a file to the victim system from the C2 server
Gather system information	Gather information about the victim system (usually includes details like hostname, IP address, operating system)
Harvest passwords	Take actions to collect user account passwords or password hashes
Hide Connections	Hide the fact that the system has certain open network connections
Hide Processes	Hide the fact that the system is running certain programs
Interactive command shell	Allow the attacker to type commands that are executed locally on the victim system. This most often involves passing the attacker a Windows command shell, allowing the attacker to issue any command that the Windows command shell can process (e.g. "dir", "cd", "type")
Kill processes	Stop currently running programs
List processes	List the currently running programs
Log off the current user	Cause the currently logged-in user to log off
Modify the registry	Make configuration changes to the system by altering the registry
Open listening port	Listen for incoming communication on a specific port
Process injection	Modify an already-running program to execute attacker-specified code
Read files	Open and review file or directory contents
Remote desktop interface	Give the attacker a graphical user interface to the system
Route network traffic	Direct network traffic from one address to another
Set file attributes	Modify the metadata that describes a file, e.g. file creation times
Set sleep interval	Specify the amount of time the backdoor should go inactive
Shutdown the system	Shutdown the system
Sleep	Go inactive (that is, do not communicate with the C2 server)
Take screenshots	Display images to the attacker that show what a user sitting in front of the system would see on the screen
Uninstall	Uninstall the backdoor
Update C2 config	Begin communicating with the C2 server at a new address

Príloha B

Krátky zoznam mutexov z obyčajných programov (nie s malware generovaných)

```
$ IDA registry mutex $
.NET CLR Data Perf Library Lock PID 1bc
C::Users:student:AppData:Local:Microsoft:Windows:Explorer:thumbcache idx.db
!rwWriterMutex
C::Users:student:AppData:Local:Microsoft:Windows:Explorer:thumbcache idx.db
!ThumbnailCacheInit
C::Users:student:AppData:Local:Microsoft:Windows:Explorer:thumbcache_sr.db!
dfMaintainer
ContentIndex_Perf_Library_Lock_PID_b58
CtfmonInstMutexDefaultS-1-5-21-1960408961-725345543-1149019770-500
E7F93AFA-D81E-4635-8A95-8458F48D369D
ESENT_Perf_Library_Lock_PID_1bc
ISAPISearch_Perf_Library_Lock_PID_3d8
iTunes-{95FD314A-DC69-4e30-AFD8-3D2E80612344}
Lsa_Perf_Library_Lock_PID_8f0
McAfee.VS.OAS.1807E9718F2D4BB3B8CCDEA17EA6A8A4
McAfeeEngineReloadMutex
McAfeeVscanBofUpdateMutex
McSvHost_CAD0E02E86CD4436B6318C111B9092AC
mcupdmgr_CAD0E02E86CD4436B6318C111B9092AC
MSCTF.GCompartListMUTEX.DefaultS-1-5-21-1960408961-725345543-1149019770-500
MSCTF.Shared.MUTEX.ACI
MsoSqmMutex_S-1-5-21-2594740-624851219-1919609084-1001
Network Inspection System_Perf_Library_Lock_PID_c64
RemoteAccess_Perf_Library_Lock_PID_1bc
Sandboxie_SingleInstanceMutex_Control
ServiceModelService 3.0.0.0_Perf_Library_Lock_PID_c64
usbhub_Perf_Library_Lock_PID_1bc
VMware_Perf_Library_Lock_PID_c64
W3SVC_Perf_Library_Lock_PID_c28
Windows Workflow Foundation 4.0.0.0_Perf_Library_Lock_PID_c64
WindowsSearchService_EfsRegKeysMutex
WSearchIdxPi_Perf_Library_Lock_PID_c28
ZonesLockedCacheCounterMutex
{bed077ff-1e5e-4efe-8e0b-739b8cb2d9bd}:sqlce_se_lck:5
{CCEDA70F-1260-4eb5-A6FD-47B501DFBF5D}
{D41B4D6A-6FC4-4bf5-9611-49CA6A58D9C1}_Mutex
```

Krátky zoznam mutexov generovaných malwarom

```
3f4bb7fc
6d5ac198
746bbf3569adEncrypt
__DDrawExclMode__
_SHuassist.mtx
AMResourceMutex2
ASPLOG
d242A3A092D94E907320A17430
DDrawDriverObjectListMutex
DDrawWindowListMutex
DENEK
DirectSound DllMain mutex (0x0000009C)
DirectSound DllMain mutex (0x000000C8)
DirectSound DllMain mutex (0x00000168)
DirectSound DllMain mutex (0x00000594)
DirectSound DllMain mutex (0x000005CC)
DirectSound DllMain mutex (0x000005DC)
DPSoundStreamMutex
HPWuSchedv.exeDm28sf0V@XK$NX8hOu
iemate_Crash
iemate_history
iemate_stock
oleacc-msaa-loaded
RDPCLIP is already running
RegWizResource_1298345_ForRegistration
VideoRenderer
WBEMPROVIDERSTATICMUTEX
Xp
c:!documents and settings!administrator!local settings!history!history.ie5!
WininetConnectionMutex
c:!documents and settings!administrator!cookies!
c:!documents and settings!administrator!local settings!temporary internet
files!content.ie5!
```

Príloha C

Výsledky testov

Tabuľka mutexov, ktoré boli zistené len u malware

Name	count(*)
MidiMapper_modLongMessage_RefCnt	4
c:!documents and settings!lab615!local settings!history!history.ie5!	4
ZonesCounterMutex	4
c:!documents and settings!localservice!cookies!	4
c:!documents and settings!localservice!local settings!history!history.ie5!	4
userenv: User Registry policy mutex	4
RAS_MO_02	3
wscntfy_mtx	3
VboxTray	3
RAS_MO_01	3
238FAD3109D3473aB4764B20B3731840	3
4FCC0DEFE22C4f138FB9D5AF25FD9398	3
0CADFD67AF62496dB34264F000F5624A	3
!MSFTHISTORY!	3
ExplorerIsShellMutex	3
ZonesCacheCounterMutex	3
ZonesLockedCacheCounterMutex	3
{A3BD3259-3E4F-428a-84C8-F0463A9D3EB5}	3
WPA_LICSTORE_MUTEX	3
SHIMLIB_LOG_MUTEX	3
WPA_RT_MUTEX	3
userenv: user policy mutex	3
_twelve_twelve_1212_87139	3
SRDataStore	3
c:!documents and settings!lab615!local settings!temporary internet files!content.ie5!	3
VboxService	3
WPA_HWID_MUTEX	3
RasPbFile	3
SingleSesMutex	3
XTCUpdate	3
fsdhqherwqi2001	3
WinExt	3
□?타虾ance0: ESENT Performance Data Schema Version 40	2
WininetStartupMutex	2
PnP_Init_Mutex	2

Instance0: ESENT Performance Data Schema Version 40	2
WPA_PR_MUTEX	2
_SHuassist.mtx	2
userenv: machine policy mutex	2
MidiMapper_Configure	2
winlogon: Logon UserProfileMapping Mutex	2
ShimCacheMutex	2
c:!documents and settings!lab615!cookies!	2
WPA_LT_MUTEX	2
WindowsUpdateTracingMutex	2
c:!documents and settings!localservice!local settings!temporary internet files!content.ie5!	2
746bbf3569adEncrypt	2
WininetProxyRegistryMutex	2
userenv: Machine Registry policy mutex	2
EnErGy	2
퓙□ance0: ESENT Performance Data Schema Version 40	1
29A	1

Tabuľka vysledkov pri forward selection metóde výberu atribútov pre tretí dataset

	DTBN			ID3 num			J48			Jrip			Ridor			NBTree		
Atributy	WHT	ACC	PRC	WHT	ACC	PRC	WHT	ACC	PRC	WHT	ACC	PRC	WHT	ACC	PRC	WHT	ACC	PRC
Performance Vector(celkovo)		95.02	94.3		94.57	94.1		94.15	93.1		95.47	95.0		95.02	94.3		95.02	94.3
atompathcount	0.0	0.7964	0.69	0.0	0.7918	0.69	1.0	0.8009	0.69	0.0	0.8009	0.69	0.0	0.7873	0.68	0.0	0.7964	0.69
mutexcount	0.0	0.6422	0.75	1.0	0.6877	0.82	1.0	0.6922	0.81	0.0	0.6922	0.81	0.0	0.6786	0.81	0.0	0.6422	0.75
explorervadcount	1.0	0.5656		0.0	0.5610	0.0	1.0	0.5656		1.0	0.5656		1.0	0.5656		0.0	0.5656	
Servicesvadcount	0.0	0.5656		0.0	0.5656		0.0	0.5656		0.0	0.5656		0.0	0.5656		0.0	0.5656	
smssvadcount	0.0	0.5656		0.00	0.5656		0.0	0.5656		0.0	0.5656		0.0	0.5656		1.0	0.5656	
csrssvadcount	1.0	0.5656		0.0	0.5656		1.0	0.5656		0.0	0.5656		1.0	0.5656		1.0	0.5656	
winlogonvadcount	1.0	0.5656		0.0	0.5656		0.0	0.5656		1.0	0.5656		1.0	0.5656		0.0	0.5656	
lsassvadcount	0.0	0.5656		0.0	0.5656		0.0	0.5656		1.0	0.5656		0.0	0.5656		0.0	0.5656	
svchostvadcount	0.0	0.6243	0.80	0.0	0.6243	0.80	0.0	0.6243	0.80	0.0	0.6243	0.80	0.0	0.6199	0.8	0.0	0.6243	0.80
execute_readvadcount	0.0	0.7915	0.69	1.0	0.7869	0.69	1.0	0.8096	0.71	0.0	0.7960	0.69	0.0	0.7915	0.74	1.0	0.7915	0.69
execute_readwritevadcount	1.0	0.7824	0.70	0.0	0.8142	0.74	0.0	0.8142	0.75	1.0	0.7869	0.71	1.0	0.7869	0.71	0.0	0.7869	0.70
execute_writecopyvadcount	0.0	0.8146	0.72	0.0	0.7600	0.76	1.0	0.8055	0.71	0.0	0.8191	0.73	0.0	0.7918	0.75	1.0	0.8146	0.72
explorerldrmodulecount	1.0	0.7786	0.67	0.0	0.7786	0.67	0.0	0.7786	0.67	1.0	0.7786	0.67	1.0	0.7786	0.67	0.0	0.7786	0.67

servicesldrmodulecount	0.0	0.7918	0.67	0.0	0.7918	0.67	0.0	0.7918	0.67	0.0	0.7918	0.67	0.0	0.7918	0.67	0.0	0.7918	0.67
smssldrmodulecount	0.0	0.7875	0.68	0.0	0.7875	0.68	0.0	0.7875	0.68	0.0	0.7875	0.68	0.0	0.7875	0.68	0.0	0.7875	0.68
csrssldrmodulecount	0.0	0.8280	0.73	1.0	0.8280	0.73	1.0	0.8280	0.73	0.0	0.8280	0.73	0.0	0.8280	0.73	1.0	0.8280	0.73
winlogonldrmodulecount	0.0	0.8055	0.69	1.0	0.8055	0.69	1.0	0.8100	0.70	0.0	0.8100	0.70	0.0	0.8055	0.69	0.0	0.8055	0.69
lsassldrmodulecount	0.0	0.7918	0.68	1.0	0.7918	0.68	1.0	0.7918	0.68	0.0	0.7918	0.68	0.0	0.7918	0.68	0.0	0.7918	0.68
svchostldrmodulecount	1.0	0.7966	0.68	1.0	0.7966	0.68	1.0	0.7920	0.68	1.0	0.7920	0.68	1.0	0.7966	0.68	1.0	0.7966	0.68
inloadfalseldrmodulecount	1.0	0.7652	0.65	1.0	0.7652	0.65	1.0	0.7697	0.65	1.0	0.7697	0.65	1.0	0.7652	0.65	1.0	0.7652	0.65
Inmemfalseldrmodulecount	1.0	0.8326	0.72	1.0	0.8326	0.72	1.0	0.8326	0.72	1.0	0.8326	0.72	1.0	0.8326	0.72	1.0	0.8326	0.72

Priloha D

Obsah CD

datamining adresár – obsahuje datasety, schémy a výsledky z časti
joining tables-program adresár – obsahuje program na vytvorenie datasetu 2
malwaretest_program – obsahuje program pre testovanie v testovacom prostredí
processData – obsahuje PHP skripty pomocou ktorých sme spracovávali textové súbory ukladali do databáze
vysledky_testov – obsahuje výsledky zberu dát v testovacom prostredí
dizertačná práca – pdf formát dizertačnej práce