Tomi Karppinen

Haittaohjelmat ja niiden analyysi

Tietotekniikan pro gradu -tutkielma

16. toukokuuta 2014

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Tomi Karppinen

Yhteystiedot: Ag C225.1, tomi.karppinen@jyu.fi

Ohjaaja: Timo Hämäläinen

Työn nimi: Haittaohjelmat ja niiden analyysi

Title in English: Malware and their analysis

Työ: Pro gradu -tutkielma

Suuntautumisvaihtoehto: Ohjelmisto- ja tietoliikennetekniikka

Sivumäärä: 112+35

Tiivistelmä: Tässä pro gradu -tutkielmassa tarkastellaan haittaohjelmia sekä menetelmiä ja työkaluja niiden analyysiin 32-bittisessä Windows-ympäristössä. Menetelmät vaihtelevat pintapuolisesta tarkastelusta syvemmälle ohjelman assembly-kieliseen käännökseen. Empiirisessä osassa työkaluja ja menetelmiä sovelletaan todellisen haittaohjelmanäytteeseen.

Avainsanat: haittaohjelma, haittaohjelmien analyysi, staattinen analyysi, dynaaminen analyysi, assembly, Windows, reverse engineering

Abstract: This master's thesis is a study of malware and methods and tools used in their analysis in a 32-bit Windows environment. The methods vary from simple observations to deeper static and dynamic analysis in assembly language. In the practical we use these tools and methods to dissect a real world malware sample.

Keywords: malware, malware analysis, static analysis, dynamic analysis, assembly, Windows, reverse engineering

Esipuhe

Tämä tutkielma on noin vuoden mittaisen prosessin tulos, jonka varrella on ehtinyt sattua monenlaista niin tutkielmaan liittyen kuin sen ulkopuolellakin. Haluan kiittää professori Timo Hämäläistä tämän tutkielman ohjauksesta ja Tietoturva-kurssin järjestämisestä, joka sai minut kiinnostumaan aiheesta. Haluan myös kiittää Vesa Lappalaista, Antti-Jussi Lakasta ja muita Nuorten peliohjelmointikurssin projektitiimiä mielenkiintoisesta työstä, joka on tehnyt tämänkin tutkielman kirjoittamisen mahdolliseksi. Lopuksi haluan kiittää perhettäni ja MLP:FiM-yhteisöä kaikesta tuesta, ja prinsessa Celestiaa auringon nostamisesta joka aamu.

Jyväskylässä 16. toukokuuta 2014

Tomi Karppinen

Termiluettelo

Assembler	Ohjelma joka ottaa syötteenään assembly-kielisen ohjelman ja tekee siitä konekielisen.
Assembly-kieli	Konekielen esitys symbolisessa, ihmisten luettavaksi tarkoitetussa muodossa.
API	Application Programming Interface, ohjelmoijalle julkistettu rajapinta jonka kautta järjestelmää tai sen komponenttia voidaan käyttää.
Disassembler	Ohjelma joka ottaa syötteenään suoritettavan ohjelman ja pyrkii luomaan siitä mahdollisimman tarkan assembly-kielisen esityksen.
Export-taulu	Taulu, joka sisältää ohjelmamoduulin julkaisemat funktiot ja niiden virtuaaliset osoitteet.
Import-taulu	Taulu, joka sisältää ohjelmamoduulin käyttämät funktiot toisista moduuleista ja niiden virtuaaliset osoitteet.
Konekieli	Ohjelmakoodi prosessorin ymmärtämässä muodossa.
Käskykanta	Kaikki kone- tai assembly-kieliset käskyt tietylle prosessoriarkkitehtuurille.
PE	Portable Executable, Windowsin käyttämä tiedostomuoto ohjelmamoduuleille.
Reverse engineering	Prosessi jossa lähdetään valmiista ohjelmasta tai tuotteesta ja pyritään selvittämään mahdollisimman tarkasti miten se on tehty.
Suhteellinen virtuaaliosoite, RVA	Virtuaaliosoite, josta on vähennetty ohjelman ensimmäisen tavun osoite muistissa (image base).
Virtuaaliosoite	Osoite prosessin muistiavaruudessa.

Kuviot

Kuvio 1. FBI MoneyPak -kiristysohjelma uhkaa käyttäjää pidätyksellä jos sakoksi naamioituja lunnaita ei makseta. Lähde: Turner et al. 2013 7
Kuvio 2. CreateFile-funktion kutsu C-kielellä. 25
Kuvio 3. Hello World -ohjelma assembly-kielellä. 30
Kuvio 4. Hello World -ohjelma konekielellä ja assembly-muodossa. xx, yy ja zz ovat symbolien muistiosoitteita, rivinvaihdot ovat vain käskyjen rajan havainnollistamiseksi. Todellisuudessa käskyn ensimmäiset tavut määräävät sen pituuden. 31
Kuvio 5. Funktiokutsu assembly-kielellä C-kutsukäytännön mukaisesti. 32
Kuvio 6. Assembly-funktion prologi C-kutsukäytännön mukaisesti. 32
Kuvio 7. Kuva pinosta funktion suorituksen aikana. Kuva piirretty lähteen “x86 Assembly Guide” 2013 perusteella. 33
Kuvio 8. Assembly-funktion epilogi C-kutsukäytännön mukaisesti. 34
Kuvio 9. Komentoriviohjelma C-kielellä. 37
Kuvio 10. Ikkunoidun Windows-ohjelman alussa suoritettava WinMain-funktio. 38
Kuvio 11. Ikkunan viestit käsittelevä ikkunaproseduuri. 39
Kuvio 12. Windows-ohjelman viestisilmukka. 40
Kuvio 13. Dynaamisesti linkitettävään kirjastoon kuuluva DllMain-funktio. 42
Kuvio 14. SeDebugPrivilege-käyttöoikeuden asettaminen. Lähde: Sikorski, Honig ja Lawler 2012, s. 247 54
Kuvio 15. Ruutukaappaus IDA Pro -disassemblerista. Pelkkien assembly-rivien sijaan ohjelman kulku on havainnollistettu ns. kontrollivuokaavion avulla. 58
Kuvio 16. Datajonon väärä tulkinta lineaarisessa disassemblerissa. Tavu 8B on todellisuudessa dataa, mutta disassembler tulkitsee sen käskyn ”mov” ensimmäiseksi tavuksi. 66
Kuvio 17. IDA Python -skripti käskyjonon korvaamiseen mitään tekemättömillä nop-käskyillä. Lähde: Sikorski, Honig ja Lawler 2012, s. 340 67
Kuvio 18. Strukturoidun poikkeuksenkäsittelyn asentaminen pinoon. Lähde: Sikorski, Honig ja Lawler 2012, s. 354 69
Kuvio 19. Strukturoidun poikkeuksenkäsittelijän poistaminen poikkeuksen jälkeen. Lähde: Sikorski, Honig ja Lawler 2012, s. 354 70
Kuvio 20. Debuggerin tunnistus kolmella tavalla prosessiympäristölohkoa käyttäen. Lähde: Sikorski, Honig ja Lawler 2012, s. 354–355 71
Kuvio 21. Rakennekuva pakatusta haittaohjelmasta ja sen sektioista. 76
Kuvio 22. Kuvakaappaus OllyDbg-ohjelmasta: PUSHAD-käsky ohjelman alusta. 76
Kuvio 23. Windowsin palomuurivaroitus haittaohjelman ajon yhteydessä. Kuvakaappaus. 79
Kuvio 24. Kuvakaappaus Process Hacker -ohjelmasta, jossa näkyy kaksi haittaohjelmaprosessia. 79
Kuvio 25. Koodianalyysin perusteella rakennettu vuokaavio MyDoom-haittaohjelman pääsäikeen toiminnasta. 84
Kuvio 26. Säikeen 1 pääohjelma Thread1Start (sub_50311C) ja sen kutsuma aliohjelma. Kuvakaappaus IDA Pro -disassemblerista. 85

Kuvio 27. Koodianalyysin perusteella rakennettu vuokaavio MyDoom-haittaohjelman säikeen 2 toiminnasta. Kysymysmerkeillä merkityt ehdot ovat globaaleista muuttujista ja GetTickCount-kutsujen tuloksista laskettuja kompleksisia ehtoja. 87
Kuvio 28. Pinoanalyysin epäonnistuminen funktiossa MakeDnsQuery. Kuvakaappaus IDA Pro -disassemblerista. 88
Kuvio 29. Pinon rakenne MakeDnsQuery-funktiossa. Perinteisen pinokehyksen sijaan pinoa käytetään funktiossa suoraan pino-osoittimeen lisäämällä. 89
Kuvio 30. Koodianalyysin perusteella rakennettu vuokaavio MyDoom-haittaohjelman säikeen 3 toiminnasta. 91
Kuvio 31. Koodianalyysin perusteella rakennettu vuokaavio MyDoom-haittaohjelman säikeen 4 toiminnasta. 93
Kuvio 32. Vuokaavio koodianalyysin kulusta säie kerrallaan. 95
Kuvio 33. Microsoftin cl-kääntäjällä assembly-kielelle käännetty C-kielinen Hello World -ohjelma. 123
Kuvio 34. Esimerkki import-osoitetaulukoukun asentavasta aliohjelmasta. Vapaasti mukaeltu lähteestä Richter 1999, luku 22 124
Kuvio 35. Kuvakaappaus järjestelmänvalvontakonsolista, jossa on näkyvissä Windowsin palveluita. 125
Kuvio 36. Kuvakaappaus ScoopyNG-ohjelmasta, joka käyttää tunnetuimpia menetelmiä virtuaalikoneen tunnistamiseen. 125
Kuvio 37. Kuvakaappaus VirusTotal-sivuston tunnistamasta MyDoom-haittaohjelmasta. . 126
Kuvio 38. Kuvakaappaus PEid-ohjelmasta, joka näyttää, että haittaohjelma on pakattu UPX-pakkauksella. 127
Kuvio 39. Kuvakaappaus PEview-ohjelmasta: IMAGE_FILE_HEADER-sektio. 127
Kuvio 40. Kuvakaappaus PEview-ohjelmasta: pakatun ohjelman UPX0-sektio. 127
Kuvio 41. Kuvakaappaus PEview-ohjelmasta: pakatun ohjelman UPX1-sektio. 128
Kuvio 42. Kuvakaappaus OllyDbg-ohjelmasta: suorituksen pysäyttäminen POPAD-käskyyn. 128
Kuvio 43. Kuvakaappaus PEView-ohjelmasta: puretun ohjelman rekonstruoitu import-hakemisto. 129
Kuvio 44. Kuvakaappaus Windowsin Muistiosta, jossa näkyy puretusta haittaohjelmasta kaapattuja merkkijonoja. 130
Kuvio 45. Kuvakaappaus Fakenet-verkkosimulaattorista haittaohjelman ajon jälkeen. 131
Kuvio 46. MyDoom-haittaohjelman aloituskohta. Kuvakaappaus IDA Pro -disassemblerista.131
Kuvio 47. MyDoom-haittaohjelman funktion sub_5031E4 alku. Kuvakaappaus IDA Pro -disassemblerista. 132
Kuvio 48. Jatkoa MyDoom-haittaohjelman funktiolle sub_5031E4. Osa funktiokutsuista on nimetty uudelleen niiden toiminnallisuuden analyysin perusteella. Kuvakaappaus IDA Pro -disassemblerista. 132
Kuvio 49. MyDoom-haittaohjelman funktio sub_50565B, jota pääsäie kutsuu ikuisessa silmukassa sekunnin välein. Kuvakaappaus IDA Pro -disassemblerista. 133
Kuvio 50. Verkkoyhteyden tarkastus MyDoom-haittaohjelman säikeessä. Jos verkkoyhteyttä ei tunnisteta, ohitetaan huomattava määrä ohjelmakoodia. Kuvakaappaus IDA Pro -disassemblerista. 133

Kuvio 51. Heksadesimaalinäkymä hyppykäskystä. Tilarivillä näkyvä osoite 00004B0D on fyysinen osoite tiedostossa. Korostetut tavut nollakäskyillä 90 korvaamalla hyppykäsky saadaan poistettua. Kuvakaappaus IDA Pro -disassemblerista. 134
Kuvio 52. Säikeen 2 aliohjelmaa Thread2Main, jossa käynnistetään säikeitä 3. Kuvakaappaus IDA Pro -disassemblerista. 134
Kuvio 53. Säikeen 3 pääohjelma Thread3Start, jossa kasvatetaan laskuria Addend säikeiden lukumäärän kasvaessa ja kutsutaan aliohjelmaa Thread3Main. Kuvakaappaus IDA Pro -disassemblerista. 135
Kuvio 54. MyDoom-haittaohjelman funktioriippuvuuskaavio. Kuvakaappaus IDA Pro -disassemblerista. 135
Kuvio 55. MyDoom-haittaohjelman funktio sub_506C46. Kuvakaappaus IDA Pro -disassemblerista. 136
Kuvio 56. Tiedoston tunnisteen vertailu merkki kerrallaan MyDoom-haittaohjelman tiedostojen prosessointifunktiosta. Vertailut on siirretty käsin kolmeen tasoon kunkin kirjaimen mukaan. Kuvakaappaus IDA Pro -disassemblerista. 136
Kuvio 57. Keskeytyskohta, joka ei pysäytä suoritusta, mutta laskee kuinka monta kertaa siitä on menty läpi. Kuvakaappaus IDA Pro -disassemblerista. 137
Kuvio 58. Aliohjelman sub_504971 palauttamia SMTP-otsikkotietoja. Kuvakaappaus IDA Pro -disassemblerista. 137

Taulukot

Taulukko 1. Tärkeimmät PE-tiedostosta löytyvät sektiot. Lähteet: Sikorski, Honig ja Lawler 2012, s. 22 ja “Microsoft PE and COFF Specification” 2013, s. 66-68. 18
Taulukko 2. Sektion otsikkotiedot PE-tiedostossa. Lähde: “Microsoft PE and COFF Specification” 2013, s. 24. 19
Taulukko 3. Microsoft Windowsin ydinkirjastoja. Lähteet: Sikorski, Honig ja Lawler 2012, s. 17, 159, Petzold 1998, luku 5 46
Taulukko 4. sockaddr_in -tietotyypin kentät. Lähde: Hart 2010, s. 416 55
Taulukko 5. Hyppykäskyjen operaatiokoodit x86-arkkitehtuurissa. Lähde: “Intel 64 and IA-32 Architectures Software Developers Manual” 2013, Vol. 2, Chapter 3. 118
Taulukko 6. x86-prosessorin rekisterit. Lähde: “Intel 64 and IA-32 Architectures Software Developers Manual” 2013, Vol. 1, chapter 3.4 119
Taulukko 7. x86-prosessorin tärkeimmät liput debuggauksen kannalta. Lähde: Sikorski, Honig ja Lawler 2012, s. 72 120
Taulukko 8. Puretun MyDoom-ohjelmatiedoston merkkijonoista löydettyjä kirjastofunktioita. 122

Sisältö

1 JOHDANTO 1
1.1 Tutkielman taustaa 1
1.2 Tutkimusongelma 2
1.3 Tutkimuksen rakenne ja menetelmät 2

2 HAITTAOHJELMAT 3
2.1 Tyypillisiä piirteitä 3
2.2 Luokittelu 4
2.3 Haittaohjelmien tarkoitus 6
2.4 Haittaohjelmien tartuntavektorit 8
2.5 Suojautuminen haittaohjelmilta virustorjuntaohjelmilla 9
2.6 Haittaohjelmahyökkäykseen varautuminen suunnittelemalla 11
2.6.1 Valmistautuminen 12
2.6.2 Uhkan tunnistaminen 12
2.6.3 Vahinkojen minimointi 13
2.6.4 Uhkan poistaminen 13
2.6.5 Tilanteen palauttaminen 13
2.6.6 Oppiminen tapahtuneesta 14

3 HAITTAOHJELMAN ANALYYSI 15
3.1 Staattinen analyysi 15
3.1.1 Tiedoston visuaalinen tarkastelu 15
3.1.2 Merkkijonot 16
3.1.3 Dynaamisesti linkitetyt funktiot 17
3.1.4 PE-tiedostomuoto 18
3.1.5 Ohjelmaan käännetyt resurssit 19
3.1.6 Staattista analyysia vaikeuttavia tekniikoita 20
3.2 Dynaaminen analyysi 21
3.2.1 Koeympäristö 21
3.2.2 Muutosten seuranta järjestelmässä 23
3.2.3 Muutokset tiedostoihin ja rekisteriin 24
3.2.4 Kahvat 25
3.2.5 Verkkoliikenne 26

4 ASSEMBLY-KIELI JA 32-BITTINEN WINDOWS 28
4.1 Lyhyt johdatus assembly-kieleen x86-arkkitehtuurissa 28
4.1.1 Assembly-kielen määritelmä 28
4.1.2 Rekisterit ja muistiavaruus 29
4.1.3 Hello World -esimerkkiohjelma ja yhteys konekieleen 30
4.1.4 Funktiokutsut ja C-kutsukäytäntö 31
4.1.5 Keskeytykset ja poikkeukset 34
4.2 32-bittiset Windows-ohjelmat 36
4.2.1 Standardi komentoriviohjelma 37

4.2.2 Standardi ikkunoitu ohjelma 37
4.2.3 Ei-standardi aloituskohta 40
4.2.4 Dynaamisesti linkitettävät kirjastot eli DLL:t 40
4.2.5 Kernel-tilan ohjelmointi ja ajurit 42
4.2.6 Palvelut (Windows Services) 44
4.3 Tärkeitä Windows API -kutsuja ja menetelmiä 45
4.3.1 Tiedostot ja IO 46
4.3.2 Rekisterioperaatiot 47
4.3.3 Haitallisen ohjelmakoodin ajaminen 48
4.3.4 Pysyvyyden varmistaminen eli persistenssi 49
4.3.5 Käyttäjältä piiloutuminen ja käyttäjän toiminnan vakoilu 51
4.3.6 Käyttöoikeuksien korottaminen 53
4.3.7 Verkkoliikenne 54

5 HAITTAOHJELMIEN SYVÄLLISEMPI ANALYYSI 56
5.1 Staattinen lähestymistapa 57
5.2 Dynaaminen lähestymistapa 59

6 ANALYYSIA VAIKEUTTAVIA TEKNIIKOITA 61
6.1 Pakatut haittaohjelmat 61
6.1.1 Pakatun haittaohjelman rakenne 61
6.1.2 Pakatun haittaohjelman tunnistaminen 62
6.1.3 Pakkauksen purkaminen 63
6.1.4 Poly- ja metamorfismi 64
6.2 Assembly-koodin obfuskaatio staattista analyysiä vastaan 65
6.2.1 Disassemblerin toimintaperiaate 65
6.2.2 Ohjelmakoodin muuttaminen käsin 66
6.2.3 Ehdottomat ehdolliset hypyt 67
6.2.4 Hyppykäskyt 68
6.3 Debuggerin ja virtuaalikoneen tunnistus 70

7 HAITTAOHJELMAN ANALYYSI KÄYTÄNNÖSSÄ 74
7.1 Tutkittava haittaohjelma 74
7.2 Tutkimusympäristö 75
7.3 Pakkauksen purku 75
7.4 Puretun tiedoston staattinen tarkastelu 77
7.5 Puretun tiedoston dynaaminen tarkastelu 78
7.6 Syvempi staattinen ja dynaaminen analyysi 80
7.6.1 MyDoom-unpacked.exe 80
7.6.2 MyDoom-unpacked.exe: säie 1 (sub_50311C) 85
7.6.3 MyDoom-unpacked.exe: säie 2 (sub_504C1B) 85
7.6.4 MyDoom-unpacked.exe: säie 3 (sub_504A37) 87
7.6.5 MyDoom-unpacked.exe: säie 4 (sub_50477F) 92
7.6.6 Tutkimuksen tulokset ja päätelmät 93

8 YHTEENVETO 96

LÄHTEET 97

LIITTEET 103
A Työkaluja haittaohjelmien analysointiin 103
A.1 Pakkauksen tunnistaminen 103
A.2 MD5-tarkistussummalaskurit 104
A.3 Dynaaminen analyysi 104
A.4 Ohjelman otsikkotietojen ja sektioiden tarkasteluun 106
A.5 Ohjelman resurssien tarkasteluun ja muokkaamiseen 107
A.6 Disassemblerit ja debuggerit 108
A.7 Pakkaus- ja kryptausohjelmat 109
A.8 Muita ohjelmia 110
B Tärkeimmät x86-assemblykäskyt 110
B.1 Datan siirto ja muistin osoitus 110
B.2 Kokonaislukuaritmetiikka 112
B.3 Boolean-aritmetiikka 114
B.4 Pino 115
B.5 Suorituksen haarauttaminen 116
B.6 Vertailu ja ehdolliset hypyt 117
C Taulukot 119
D Kuvat 123
D.1 Luku 4 123
D.2 Luku 5 125
D.3 Luku 6 126

1 Johdanto

Tämä tutkielma on kirjoitettu lukijalle, joka on kiinnostunut haittaohjelmien analyysistä tieteenalana tai haluaa oppia analysoimaan haittaohjelmia itsenäisesti esimerkiksi kotonaan tai työpaikallaan. Aiheesta kirjoitetut tieteelliset artikkelit ovat pääosin syventäviä, perehdyttävää kirjallisuutta on vähän ja tämän tutkielman tarkoituksena onkin koota niistä oleellisin ymmärrys yksiin kansiin.

Tutkielman painopisteessä ovat menetelmät ja työkalut joilla löydettyä, ennestään tuntematonta haittaohjelmaa voidaan oppia ymmärtämään. Menetelmät esitellään järjestyksessä vähemmän esitietoja vaativista ”yksinkertaisista” menetelmistä vähitellen monimutkaisuutta nostaen menetelmiin, jotka vaativat syvällistä tietämystä suoritusympäristöstä. Tarvittavat käsitteet ja muut esitiedot pyritään selittämään sitä mukaa kuin niitä tarvitaan sitä mukaa kuin aiheen laajuus antaa myöten.

1.1 Tutkielman taustaa

Haittaohjelmat ovat olleet tietojärjestelmien käyttäjien ja ylläpitäjien riesana jo tietotekniikan alkuvaiheista asti, ja joka vuosi niitä syntyy uusia. Tietoturvayritykset tekevät kaikkensa pysyäkseen tilanteen tasalla, mutta uusimmilta ja etenkin kohdennetuilta uhkilta heidänkään tuotteensa eivät pysty suojaamaan. Suojautumisen lisäksi usein halutaan tietää miten haittaohjelma on päässyt järjestelmään, kuka sen on laatinut ja miten vastaavilta tapauksilta voidaan välttyä tulevaisuudessa.

Haittaohjelmat, etenkin virukset, ovat saaneet monien käyttäjien ja jopa IT-alan ammattilaisten silmissä suorastaan myyttisiä ulottuvuuksia, ja siten myös niiden analysointi käsin voidaan nähdä taitona, joka vain suurten virustorjuntayritysten leivissä työskentelevät korkeamman tason asiantuntijat voivat hallita. Uutiset miljoonien eurojen menetyksiin johtaneista verkkohyökkäyksistä ja Stuxnetin kaltaisista ”superhaittaohjelmista” kasvattavat entisestään näitä luuloja. Eräs motiivi työn takana onkin antaa lukijalle selkeä, tieteellinen näkökulma haittaohjelmiin ja niiden tutkimukseen.

1.2 Tutkimusongelma

Haittaohjelmien analyysistä on kirjoitettu hyvin vähän perehdyttävää kirjallisuutta. Merkittävimpänä poikkeuksena tähän on vuonna 2012 julkaistu 800-sivuinen perusteos *"Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software"* (Sikorski, Honig ja Lawler 2012), johon myös tämä tutkielma suurin osin perustuu. Toinen erityisesti haittaohjelmiin keskittyvä teos on *"Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code"* (Ligh et al. 2010), menetelmäkeskeisempi teos joka vaatii lukijaltaan jo hieman esitietoja aiheesta. Muu aiheeseen perehdyttävä materiaali on esimerkiksi SANS-instituutin julkaisemaa opetusmateriaalia ja muuta ei-akateemista kirjallisuutta.

1.3 Tutkimuksen rakenne ja menetelmät

Tutkielma koostuu teoriaosasta, jossa analyysimenetelmät ja niille tarvittavat esitiedot käydään läpi, ja empiirisestä osasta, jossa menetelmiä sovelletaan käytännössä olemassa olevaan haittaohjelmaan.

Luku kaksi sisältää yleistä tietoa haittaohjelmista: mikä tekee ohjelmasta haittaohjelman, miten haittaohjelmia luokitellaan ja miten niiltä voidaan suojautua. Kolmannessa luvussa esitellään staattinen ja dynaaminen analyysitapa, ja keinoja tutkia haittaohjelmia menemättä vielä pintaa syvemmälle. Luku 4 antaa tarvittavat perustiedot x86-arkkitehtuurista ja sen assembly-kielestä, sekä 32-bittisestä Windowsista ja sen tärkeimmistä API-kutsuista. Luvussa 5 esitellään disassembleri ja debuggeri, ja kuinka niitä voidaan käyttää haittaohjelmakoodin tarkempaan tutkimukseen. Luku 6 kokoaa menetelmiä, joilla haittaohjelmien kirjoittajat voivat vaikeuttaa analyysiä, ja tarjoaa keinoja niiden kiertämiseen. Luvussa 7 sovelletaan opittuja menetelmiä käytäntöön MyDoom-madon F-variantin parissa. Luku 8, summaa yhteen tärkeimmät johtopäätökset ja esittää mahdollisia jatkotutkimuskohteita.

Tutkielmassa käytetyt työkalut linkkeineen ja kuvauksineen löytyvät liitteestä A. Liitteeseen B on koottu tiivistelmä tärkeimmistä assembly-käskyistä esimerkkien kanssa. Esimerkit sisältävät myös konekieliset vastineet käskyille. Liitteisiin C ja D on koottu pidemmät kuvat ja taulukot, jotka eivät sopineet varsinaisen tekstin joukkoon.

2 Haittaohjelmat

Haittaohjelmalle löytyy kirjallisuudesta erilaisia määritelmiä:

- ” ’Malware’ is short for malicious software and is typically used as a catch-all term to refer to any software designed to cause damage to a single computer, server, or computer network, whether it’s a virus, spyware, et al.” (Moir 2003)
- ”Malware: Any piece of code that has malicious intentions and/or performs a function that the user was not aware that it was going to do” (Hutcheson 2006, s. 7)
- ”Malware is a general term for a variety of harmful software specifically designed to attack computer systems, networks, or data.” (Buehler et al. 2009, s. 121)
- ”Malicious code (or malware) is defined as software that fulfills the deliberately harmful intent of an attacker.” (Moser, Kruegel ja Kirda 2007)

Laveimpana määritelmänä haittaohjelma on siis mikä tahansa ohjelma, joka asentaa itsensä johonkin elektroniseen järjestelmään tehden siellä käyttäjän tietämättä jotain, mikä voidaan olettaa haitalliseksi käyttäjälle tai järjestelmälle. Tämä määritelmä on niin laaja, että sen alle mahtuu lukuisia alaluokituksia: virukset, madot ja botnetit vain joitain mainitakseni. On myös todennäköistä, että tulevaisuudessa syntyy uusia haittaohjelmatyyppejä ja entisiä poistuu käytöstä kuten on tapahtunut levykkeistä Internetiin siirryttäessä. Tässä luvussa tarkastellaan lyhyesti erilaisia haittaohjelmia ja niiden tekijöiden motivaatioita ennen varsinaiseen analyysiin siirtymistä.

2.1 Tyypillisiä piirteitä

Haittaohjelmia on ollut niin kauan kuin tietokoneitakin ja niiden tekninen toteutus on muuttunut aikojen ja mahdollisuuksien mukaan, mutta tietyt piirteet ovat pysyneet samoina (Zeltser 2010).

Leviäminen järjestelmästä toiseen tavalla tai toisella (levykkeet, verkko, sähköposti) on ollut aina haittaohjelmille ominainen piirre. Biologiset virukset jakavat tämän piirteen haittaohjelmien kanssa, mistä nimitys ”tietokonevirus” on saanut alkunsa.

Puhutaan myös, että virus, tai haittaohjelma, ”tartuttaa” tai ”saastuttaa” tietokoneen tai tiedoston. Tällä salaperäiseltä kuulostavalla tapahtumalla tarkoitetaan tapaa jolla haittaohjelma varmistaa toimintansa jatkumisen, yleensä asettamalla itsensä käynnistymään laitteen tai käyttöjärjestelmän käynnistymisen yhteydessä. Haittaohjelma voi myös kiinnittää itsensä toiseen ohjelmaan niin että se käynnistyy kun toinen ohjelma käynnistetään. Tällaista ”saastunutta” ohjelmaa kutsutaan Troijan hevoseksi kreikkalaisten Troijan sodassa käyttämän onton puuhevosen mukaan.

Ohjelman jatkuvan toiminnan edellytyksenä on myös, ettei haittaohjelmaa tunnisteta ja poisteta. Ohjelma saattaa muuttaa itseään tunnistamisen vaikeuttamiseksi, tai piilottaa itsensä käyttöjärjestelmän aktiivisten prosessien listasta. Haittaohjelma saattaa myös sisältää vastatoimia analyysin vaikeuttamiseksi.

Lopuksi haittaohjelmalla on yleensä jokin tarkoitus jota varten se on tehty ja jota muu sen toiminnallisuus palvelee. Tarkoitus voi olla mitä tahansa käytännön pilasta vakoiluun tai järjestelmälliseen verkkohyökkäykseen osallistumiseen.

2.2 Luokittelu

Virustorjuntayritysten tietokannoissa ja muissa järjestelmällisissä haittaohjelmaluetteloissa haittaohjelmat on yleensä luokiteltu niiden toiminnallisuuden mukaan. Seuraavat luokitukset ovat peräisin lähteistä Sikorski, Honig ja Lawler 2012, s. 3 ja Buehler et al. 2009, s. 122.

Virukset ja madot ovat järjestelmästä toiseen tavalla tai toisella leviäviä haittaohjelmia. Kaikissa lähteissä selkeää eroa ei ole tehty näiden kahden välillä, mutta yleisimpänä erotuksena mato käyttää hyväkseen tietoturva-aukkoja leviämiseen kun taas virus vaatii yhteistyötä käyttäjältä esimerkiksi sähköpostin liitetiedoston avaamisen muodossa.

Takaovi (backdoor) päästää hyökkääjän yhdistämään järjestelmään ja suorittamaan siinä haluamiaan komentoja.

Bottiverkko eli zombie-verkko (botnet) on erikoistapaus takaovesta, jossa käskyt tulevat keskuspalvelimelta joko suoraan tai toisten verkon jäsenien eli *bottien* kautta.

Latausohjelma (downloader) on ohjelma joka lataa järjestelmään lisää haittaohjelmia. Latausohjelma on yleensä kooltaan varsinaista haittaohjelmaa pienempi, mikä tekee siitä soveltuvamman Troijan hevosen tai tietoturva-aukkoa hyödyntävän hyökkäyksen yhteyteen.

Vakoiluohjelma (spyware) kerää tietoa saastuttamastaan järjestelmästä ja sen käyttäjistä. Kerättävä tieto voi olla esimerkiksi käyttäjän web-selailuhistoriaa, näppäimien painalluksia tai käyttäjätunnuksia.

Keylogger eli näppäimistön kuuntelija on vakoiluohjelma joka kuuntelee näppäimenpainalluksia.

Mainosohjelmat (adware) näyttävät käyttäjälle mainoksia, mahdollisesti käyttäjän selailuhistorian perusteella. Erilaiset selainten työkalurivit ja selaimen kotisivua muuttavat ohjelmat ovat yleisimpiä mainosohjelmia. Osa mainosohjelmista kerää tietokantaa käyttäjistään vakoiluohjelmien tapaan, mahdollisesti käyttäjien tietämättä. Mainosohjelmat asentuvat yleensä huomaamatta ilmaisohjelmien mukana ja oikeuttavat olemassaolonsa ohjelman käyttöoikeussopimuksessa.

Käynnistäjä (launcher) käynnistää muita ohjelmia, yleensä jollain epätavallisella tavalla joka piilottaa käynnistettävän ohjelman tai antaa sille ylimääräisiä käyttöoikeuksia.

Rootkit kätkee itsensä ja mahdollisesti muita sovelluksia käyttäjältä ja järjestelmältä. Rootkitiä käytetään yleensä muiden haittaohjelmien kuten takaovien yhteydessä vaikeuttamaan tunnistamista.

Pelotteluohjelman (scareware) tarkoituksena on säikäyttää käyttäjä ostamaan jotain, yleensä antivirusohjelma pelotteluohjelman tekijältä.

Roskapostimato tai -virus tartuttaa käyttäjän järjestelmän ja käyttää sitä lähettämään roskapostia.

Yleensä haittaohjelmat eivät sovi siististi mihinkään tiettyyn kategoriaan, vaan sisältävät toiminnallisuutta useammasta. Esimerkiksi Symantecin (Liu 2004) madoksi luokittelema MyDoom.F -haittaohjelma sisältää latausohjelman ja takaoven sekä suorittaa ajoittaisia DDoS-hyökkäyksiä päätarkoituksensa, roskapostin lähettämisen lisäksi.

2.3 Haittaohjelmien tarkoitus

Haittaohjelmien alkuaikoina motivaatio niiden tekemiselle oli enimmäkseen osoittaa joko omaa teknistä osaamistaan tai haavoittuvuuksia kohdejärjestelmästä. Vuoden 2004 ”matosota” Bagle-, Netsky- ja Mydoom -matojen välillä myös osoittaa joidenkin tekijöiden kilpailumielen – madot sisältävät herjaavia tekstinpätkiä toisistaan ja poistavat ”kilpailijansa” tietokoneelta sellaisen kohdatessaan. Kilpailu herätti myös tutkijoiden mielenkiinnon ja esimerkiksi Tanachaiwiwat ja Helmy 2006 tarkasteli tapahtunutta biologiasta tutun saalistaja–saalis-mallin avulla ja nosti esiin mahdollisuuden tehdä ”hyödyllinen” haittaohjelma, jonka ainoa tarkoitus olisi poistaa muita haittaohjelmia. Menetelmää on kuitenkin pidetty sen verran kyseenalaisena ettei sitä ole otettu käyttöön laajemmin. (Egele et al. 2012; “PC World: Virus Writers Wage Worm War” 2013)

Nykypäivän haittaohjelmien tekijöiden mielessä on entistä enemmän raha, minkä paljastaa kiristysohjelmien (ransomware), vakoiluohjelmien ja bottiverkkojen kasvava lukumäärä. Roska- ja huijaussähköpostit (phishing) ovat myös olleet joitakin vuosia suosittuja ansaintakeinoja verkkohuijareille ja niiden lähettämiseen on käytetty myös haittaohjelmia ja niiden avulla rakennettuja bottiverkkoja. Sähköpostin osuus haittaliikenteestä on kuitenkin vähenemään päin, mutta fokus on siirtymässä enemmän sosiaaliseen mediaan. (Turner et al. 2013).

Kiristysohjelmat lukitsevat käyttäjän tietokoneen tai puhelimen kunnes tämä suostuu maksamaan kiristäjälleen. Esimerkiksi FBI MoneyPak-kiristysohjelma näyttää käyttäjälle varoitusruudun kuten kuvassa 1, jolla käyttäjä yritetään saada uskomaan että uhkaus tulee viralliselta taholta. Haittaohjelma voi käyttää uhkaustansa tehostamaan myös kryptausta, jonka se lupaa avaavansa kun lunnaat on maksettu. Kuten perinteisessäkin kiristyksessä, käyttäjällä ei ole mitään takuuta siitä että kiristäjä pitää sanansa, mutta etenkin kryptaavan haittaohjelman tapauksessa maksaminen voi olla vaihtoehdoista helpoin jos tiedostoista ei ole varmuuskopioita. (Bridges 2008)

Botti on ohjelma joka asentaa itsensä käyttäjän järjestelmään ja pitää yhteyttä hallintapalvelimeen (command & control server). Hallintapalvelimen kautta bottiverkon ylläpitäjä (botmaster) voi käskyttää botteja esimerkiksi etsimään uusia verkkoon liitettäviä tietokoneita,

Kuvio 1. FBI MoneyPak -kiristysohjelma uhkaa käyttäjää pidätyksellä jos sakoksi naamioituja lunnaita ei makseta. Lähde: Turner et al. 2013

suorittamaan palvelunestohyökkäyksiä, lähettämään roskapostia tai keräämään tietoa käyttäjien järjestelmästä. Bottiverkkoja käytetään myös rikollisen toiminnan anonymisointiin. Niiden koko vaihtelee kymmenistä tuhansiin botteihin. Symantecin vuoden 2012 arvio bottien määrästä koko maailmassa on 3,4 miljoonaa. (Abu Rajab et al. 2006; Turner et al. 2013)

Vakoiluohjelmille arvokkaimpia tietoja ovat pankki- ja luottokorttitunnukset, mutta myös käyttäjätunnukset erilaisiin verkkopalveluihin ovat identiteettivarkaille rahanarvoista tietoa. Massiivimoninpelien, kuten World of Warcraftin käyttäjätunnukset ovat myös houkuttelevia kohteita, sillä pelimaailman rahaa ja harvinaisia esineitä voidaan huutokaupata oikeaa rahaa vastaan. Haittaohjelmien tekijöitä ja muita verkkorikollisia on usein vaikea saada kiinni, sillä heidän toimintansa on kansainvälistä ja jää helposti paikallisen tuomiovallan ulkopuolelle (Jakobsson 2012; Bridges 2008).

Suurin osa haittaohjelmista on tarkoitettu tarttumaan niin moneen tietokoneeseen kuin mah-

dollista. Nämä ovat yleensä niitä ohjelmia joihin virustorjuntayritykset kiinnittävät ensimmäisenä huomionsa, ja joiden ei tarvitse olla niin huolella tehtyjä. Pääasia ohjelman tekijän kannalta on että tarpeeksi moni saa tartunnan, ei ole syytä kiinnittää huomiota kuin korkeintaan käytetyimpiin vastatoimiin.

On myös olemassa kohdennettuja haittaohjelmia, jotka on varta vasten räätälöity tarkoitukseensa, esimerkiksi varastamaan liikesalaisuuksia kilpailevalta yritykseltä. Nämä haittaohjelmat ovat yleensä huomattavasti huolellisemmin tehtyjä ja vaikeampia tunnistaa jo siksi että niitä ei päästetä leviämään ennen niiden varsinaista käyttöä, jolloin virustorjuntaohjelmat eivät voi niitä tunnistaa. Analyysitekniikat joita tässäkin tutkielmassa käsitellään ovatkin ainoita tapoja tunnistaa kohdennettu haittaohjelma, selvittää sen tarkoitus ja alkuperä sekä tietenkin minimoida sen aiheuttamat vahingot. (Sikorski, Honig ja Lawler 2012, s. 4)

2.4 Haittaohjelmien tartuntavektorit

Haittaohjelmien luokitukset antavat joitain vihjeitä niiden tartuntatavoista ja -lähteistä. Eräs yksinkertaisimpia tapoja haittaohjelman levittämiseen on sähköposti. Sähköpostin välityksellä leviävät haittaohjelmat kuitenkin yleensä vaativat käyttäjän aktiivisuutta toimiakseen, mikä vähentää lähestymistavan tehokkuutta. Mahdollisesti käyttäjien tietoturvatietoisuuden parantumisen vuoksi sähköpostin välityksellä leviävien haittaohjelmien suosio on laskenut viime vuosina (Turner et al. 2013, s. 46).

Haittaohjelma voi ladata ja käynnistää itsensä myös ilman käyttäjän apua. Tähän tarvitaan yleensä haavoittuvuus jossain kohtaa verkkoliikenteen käsittelyssä. Aikaisemmin suosittuja kohteita ovat olleet avoimet palvelut (portit) käyttäjän tietokoneella, mutta palomuurien ja NAT-reitittimien yleistyttyä tämä lähestymistapa on menettänyt tehoaan. (Mavrommatis ja Monrose 2008). Sen sijaan hyökkääjät ovat kääntäneet katseensa sovellusprotokollien haavoittuvuuksiin esimerkiksi web-selaimissa, johon palomuuri tai reititin ei voi vaikuttaa, koska se toimii alemmalla verkkoprotokollakerroksella. Haavoittuvuuksien korjaaminen ohjelmistopäivityksillä jää ainoaksi "varmaksi"tavaksi tällaisten hyökkäysten ennaltaehkäisyyn. Web-selaimen tapauksessa tällaisesta hyökkäyksestä käytetään nimitystä "drive-by-download". (Narvaez et al. 2010)

Drive-by-download -hyökkäyksessä hyökkääjä käyttää haavoittuvuutta esimerkiksi JavaScriptissä tai muussa käyttäjän selaimessa suoritettavassa ohjelmakoodissa. Hyökkäys tapahtuu yleensä kolmessa osassa. Ensin käyttäjä vierailee sivulla, jolle hyökkääjä on jättänyt näkymättömän HTML-komponentin, kuten nollan pikselin kokoisen `<iframe>`-kehyksen. Kehykseen ladataan hyökkääjän sivustolta skripti, joka hyödyntää haavoittuvuutta käyttäjän selaimessa tai jossain sen lisäosassa. Jos haavoittvuuden hyödyntäminen onnistuu, skripti lataa varsinaisen haittaohjelman hyökkääjän sivustolta (joka voi olla myös eri sivusto kuin millä skripti on) ja käynnistää sen. Useammasta komponentista koostuvan haittaohjelman tapauksessa tämä haittaohjelma voi olla myös ns. latausohjelma, joka lataa kaikki haittaohjelmiston tarvitsemat komponentit hyökkääjän määrittelemältä sivustolta. (Mavrommatis ja Monrose 2008)

Mavrommatisin ym. tutkimuksen perusteella todennäköisimpiä hyökkäyksen sisältäviä sivustoja ovat aikuisviihdesivustot, mutta vaara on olemassa myös muilla sivustoilla. Uhka voi tulla joko sivustolta itseltään jos palvelin- tai esimerkiksi keskustelupalstaohjelmisto sisältää haavoittuvuuksia, tai sivuston ulkopuolelta esimerkiksi mainoksien mukana. Haittakoodia sisältävän mainoksen elinikä on yleensä lyhyt, mutta koska useat sivustot käyttävät samoja mainosverkostoja, ne voivat ehtiä saavuttaa jopa miljoonia käyttäjiä. (Mavrommatis ja Monrose 2008)

2.5 Suojautuminen haittaohjelmilta virustorjuntaohjelmilla

Virustorjuntaohjelmat perustuvat sääntöihin, joilla haittaohjelma voidaan tunnistaa muista tiedostoista. Nämä säännöt ovat luonteeltaan syntaktisia, eli ne tunnistavat ohjelmasta bittijonoja jotka ovat tietyssä järjestyksessä. Jokaista uutta haittaohjelmaa tai varianttia varten ohjelman ylläpitäjien tulee lisätä sen tietokantaan sääntö sen tunnistamiseen, ja ohjelman käyttäjät lataavat nämä uudet tunnistussäännöt päivitysten muodossa. (Bruschi, Martignoni ja Monga 2006)

Syntaktisen tunnistuksen heikkous on kuitenkin siinä, että jos haittaohjelma muuttuu esitykseltään (ei välttämättä toiminnallisuudeltaan), sitä ei enää tunnisteta säännön perusteella. Tästä on seurannut kilpajuoksu haittaohjelmien kehittäjien ja torjujien välillä, missä kumpi-

kin osapuolista on kehittänyt uusia tapoja olla ovelampi kuin vastustajansa.

Kryptografisilla algoritmeilla salatut haittaohjelmat ovat nykyään arkipäivää, mutta ensimmäisen kerran menetelmää käytettiin jo vuonna 1986. Cascade-virus käytti loogiseen XOR-operaatioon perustuvaa salausta mikä itsessään on heikko varsinkin lyhyemmillä avaimilla, mutta sopi hyvin tarkoitukseensa. Vaikka haittakoodi itsessään on salattua, sen purkava ohjelman osa on oltava selväkielisenä, ja niin vähitellen salauksen tunnistamisesta tuli osa virustorjuntaohjelmien perustoiminnallisuutta. (Schiffman 2014a)

Seuraava haaste haittaohjelmien tekijöille oli salauksen purkavan ohjelmakoodin kätkeminen virustorjuntaohjelmilta. Näin kehittyivät 90-luvulla ns. oligomorfiset eli useita salausalgoritmeja vaihdellen käyttävät, ja myöhemmin polymorfiset eli ohjelmakoodiaan muuntelevat haittaohjelmat. Polymorfismi on mahdollista saavuttaa menetelmillä, jotka muuttavat ohjelman rakennetta muuttamatta sen toiminnallisuutta, esimerkiksi lisäämällä joukkoon käskyjä jotka eivät tee mitään. Vuonna 1992 kehitettiin myös nimeä ”The Mutation Engine” kantava ohjelma, jolla kuka tahansa pystyi tekemään omasta haittaohjelmastaan polymorfisen. Samantyyppisiä ohjelmia, kuten DAME ja TPE, on julkaistu ja jatkokehitelty myös myöhemmin. (Schiffman 2014a)

Polymorfiset haittaohjelmat ovat yhä ongelma virustorjuntaohjelmille ja syntaktisen tunnistuksen rajat ovat tulleet jo vastaan. Osa virustorjuntaohjelmista käyttää ns. hiekkalaatikkoa eli emulaattoria epäilyttävien ohjelmien ajamiseen. Emulaatiolla on kaksi tarkoitusta, se auttaa tunnistamaan ajon aikana epäilyttävän ohjelmakoodin ennen sen suoritusta ja jos se suoritetaan, vaikutukset rajoittuvat hiekkalaatikon sisälle. Emulaatio on kuitenkin aina hidasta eikä kaikkia ohjelmia ei kannata ajaa hiekkalaatikossa, ja osa haittaohjelmista osaakin hyödyntää tätä valikointia. (Schiffman 2014a)

Polymorfisista haittaohjelmista seuraava askel ovat metamorfiset eli itseään muuntelevat haittaohjelmat. Metamorfiset haittaohjelmat luokitellaan kommunikaatiotyypiltään avoimiksi (*open world*) tai suljetuiksi (*closed world*), riippuen siitä, käyttävätkö ne muuntelussaan apuna viestintää. Avoimesti metamorfiset haittaohjelmat, kuten Conficker (2008) ovat yleensä modulaarisia ja lataavat itselleen lisäominaisuuksia liitännäisinä verkon kautta. Metamorfismi voi perustua muunnoksiin joko suoraan binäärimuodossa, tai ohjelma voi sisältää oman

lähdekoodinsa jollain korkeamman tason kielellä. Esimerkiksi Win32.Apparition -virus sisältää oman lähdekoodinsa, josta se kääntää itsensä uudelleen muunnosten jälkeen, jos se löytää kohdejärjestelmästä sopivan kääntäjän. (O'Kane, Sezer ja McLaughlin 2011)

Vaihtoehtona syntaktiselle tunnistukselle on semanttinen tunnistus, jossa tunnistettavia yksiköitä ovat bittien sijasta ohjelman suorittamat operaatiot. Samaan tapaan kuin staattisessa analyysissä, ohjelma tai koodilohko käännetään assembly-kielelle ja siitä muodostetaan kontrollivuograafi. Graafi kertoo käskyjen suoritusjärjestyksen, jota voidaan verrata valmiisiin malleihin tunnetuista haittaohjelmista. Etuna syntaktiseen tunnistukseen vakioiden ja rekisterien nimet on abstrahoitu pois mallista mikä tekee siitä vastustuskykyisen polymorfisia menetelmiä vastaan. Menetelmää voidaan vielä parantaa käyttämällä assembly-kielen sijaan välikieltä, jossa samaa tarkoittavat käskyt kuten `inc x` ja `add x, 1` kuvautuvat samaksi käskyksi. Kirjoitushetkellä semanttisia menetelmiä käyttäviä virustorjuntaohjelmia ei vielä ole kaupallisessa käytössä, mahdollisesti menetelmän hitauden vuoksi. (Christodorescu et al. 2005)

Metamorfiset haittaohjelmat voivat sisältää hyvin oivaltavia keinoja tunnistuksen välttämiseen. Esimerkiksi venäläisen W95.Zmist (2000) -viruksen erikoisuus on sen kyky purkaa mikä tahansa ohjelma komponenteiksi, integroida viruskoodi komponenttien lomaan ja koota niistä uusi, täysin toimiva saastutettu ohjelma. Integraatioaste on sen verran korkea, että viruskoodia on hyvin vaikea tunnistaa alkuperäisesta ohjelmakoodista. (Schiffman 2014b)

2.6 Haittaohjelmahyökkäykseen varautuminen suunnittelemalla

Varsinkin yrityksille on tärkeää, että mahdollisesta verkkohyökkäyksestä selviydytään mahdollisimman nopeasti ja vähillä vahingoilla, ja minimoidaan tulevien hyökkäysten mahdollisuus. Kuten tulipalojen ja muiden perinteisempien katastrofien tapauksessa, myös tietomurtoja ja haittaohjelmien hyökkäyksiä varten voidaan suunnitella etukäteen. Tällaista suunnitelmaa kutsutaan verkkohyökkäyksiin varautumissuunnitelmaksi (eng. incident response plan)

Tietoturvauhkiin erikoistuneen SANS-instituutin (Distler 2007) mukaan suunnitelman tulisi koostua kuudesta peräkkäisestä askeleesta, jotka ovat valmistautuminen, uhkan tunnistami-

nen, vahinkojen minimointi, uhkan poistaminen, tilanteen palauttaminen ja oppiminen tapahtuneesta. Suunnitelman rakenne ja järjestys pysyvät aina samana, mutta jokainen askel on räätälöitävä erikseen yrityksen tarpeisiin.

2.6.1 Valmistautuminen

Valmistautuminen on se suunnitelman askel, joka on syytä ottaa ennen hyökkäystä. Tärkeä osa sitä on uhkaprofiilin rakentaminen eli tunnistaminen, mitä uhkia on olemassa ja kuinka todennäköisiä mitkäkin niistä ovat. Haittaohjelmien tapauksessa esimerkiksi suurimmassa vaarassa ovat julkiseen verkkoon kytketyt järjestelmät.

Jotta hyökkäykseen vastaaminen olisi nopeaa ja tehokasta, on tärkeää kiinnittää mukana olevat henkilöt ja heidän tehtävänsä. Teknisen henkilöstön lisäksi mukana on hyvä olla ihmisiä yrityksen laki-, viestintä- ja henkilöstöosastoilta. Myös yhteistoiminnasta poliisin kanssa on viisasta sopia etukäteen, kuten myös julkinen tiedottaminen.

Myöhemmissä vaiheissa tarvittavat työkalut on myös syytä laittaa valmiiksi, ja pitää päivitettynä. Erityisesti haittaohjelmien analyysissa tarvitaan runsaasti erilaisia työkaluja verkkoliikenteen ja ohjelmien seurantaan ja tutkimiseen, joista osa esitellään tässä tutkielmassa sitä mukaa kun niitä tarvitaan. Lista työkaluista on myös nähtävissä liitteessä A.

Viimeinen vaihe valmistautumisessa on harjoittelu. Kuten paloharjoituksissakin, tarkoituksena on saada toimintatavat ja operaation kulku mahdollisimman tutuksi. On kuitenkin pidettävä mielessä, että jokainen hyökkäys on yksilöllinen, ja siksi myös harjoituksiin on hyvä lisätä hieman variaatiota.

2.6.2 Uhkan tunnistaminen

Kun hyökkäys tapahtuu, on ensimmäinen askel tunnistaa mitä tarkalleen tapahtui tai on tapahtumassa. Ensimmäinen havainto voi olla esimerkiksi työntekijän tietokoneen hidastuminen, tai outo merkintä palomuurin lokissa. Siksi kaikkien tulisi oppia näkemään merkit mahdollisesta verkkohyökkäyksestä ja tietää, kenelle epäilyksestä voi ilmoittaa.

Uhkan tunnistamisessa on tärkeää pitää kiinni faktoista ja johdonmukaisesta ajattelusta. On

helppoa lähteä huhujen ja spekulaatioiden tielle tai joutua paniikkiin, mutta hyvällä valmistelulla ja harjoitellulla rationaalisuudella sitä voidaan ennaltaehkäistä.

Haittaohjelmien löytäminen ja analysointi kuuluvat tähän vaiheeseen. Täydellinen analyysi ei ole aina välttämätön eikä siihen saata olla aikaa, mutta mitä paremmin haittaohjelmien toiminnasta ollaan perillä, sitä paremmin voidaan varmistua myöhempien vaiheiden onnistumisesta. (Zeltser 2010)

2.6.3 Vahinkojen minimointi

Kun saastuneet tietokoneet on tunnistettu, ne kannattaa kytkeä irti verkosta joko fyysisesti tai palomuurisäännöillä jotta ne eivät pääse saastuttamaan muuta verkkoa tai suorittamaan hyökkääjän komentoja. Syyllisten selvittäminen astuu yleensä kuvaan tässä vaiheessa prosessia, joten tietokoneet tulisi jättää päälle alkuperäistilaansa ja haittaohjelmat ottaa dokumentoidusti talteen todistusaineistoksi ennen seuraavaa vaihetta, uhkan poistamista.

2.6.4 Uhkan poistaminen

Kun tapaus on tutkittu niin teknisesti kuin juridisestikin, voi järjestelmiä alkaa palauttaa takaisin tuotantokäyttöön. Ensimmäinen osa siitä on uhkien, eli tässä tapauksessa haittaohjelmien poistaminen.

Varmin tapa haittaohjelman hävittämiseen on aina järjestelmän uudelleenasennus puhtaalta asennusmedialta. Varmuuskopioiden kanssa on oltava varovainen, sillä nekin voivat olla saastuneita, sitä suuremmalla todennäköisyydellä mitä viimemmin ne on otettu.

Jos ja vain jos haittaohjelma on analysoitu tarkasti, se voidaan poistaa ilman järjestelmän uudelleenasennusta. Tämä on kuitenkin aina riski, sillä siitä on voinut jäädä komponentteja joita analyysi ei ole löytänyt.

2.6.5 Tilanteen palauttaminen

Palautusvaiheessa kun uhka on poistettu, puhdistetut järjestelmät voidaan kytkeä takaisin verkkoon. Järjestelmiä ei kuitenkaan kannata ottaa heti takaisin tuotantokäyttöön, vaan niitä

tulisi seurata uusien hyökkäyksien varalta sopivaksi määritellyn ajan. Hyviä välineitä seurantaan ovat palomuurilokit ja niiden analysointiohjelmat, sekä tunkeutumisenestosovellukset (IDS, Intrusion Detection System). Seurantaa voidaan jatkaa vielä senkin jälkeen kun järjestelmä on takaisin käytössä, mutta varsinkin jos kyseessä on työasematietokone, on seurannasta ja sen tarkoituksesta syytä ilmoittaa käyttäjille.

2.6.6 Oppiminen tapahtuneesta

Viimeinen vaihe alkaa tapahtuneen dokumentoinnista ja sen esittämisestä yrityksen tai organisaation johdolle. Sen sijaan että dokumentaatio päätyisi hyllyyn pölyttymään, siitä voidaan johtaa uusia toimintamalleja tulevien uhkien estämiseen ja pohtia niiden toteutusmahdollisuuksia.

Tietoturvahenkilöstölle oppiminen tapahtuneesta tarkoittaa järjestelmien immunisointia tulevia hyökkäyksiä vastaan. Analyysissa opitut yksityiskohdat tulevat tässä tarpeeseen. Tässä joitakin ehdotuksia niiden hyödyntämiseen:

Verkkoliikenne: Yleensä haittaohjelmat kommunikoivat joko keskuspalvelimen tai tekijänsä kanssa, tai esimerkiksi lähettävät roskapostia. Liikenne itsessään voi olla salattua, mutta vähintäänkin kohdeosoite ja portti ovat aina selväkielisiä, ja ne voidaan lisätä sekä reitittimien että järjestelmien palomuuri- ja IDS-sääntöihin. Säännöt kannattaa konfiguroida myös hälyttämään estämisen lisäksi, jolloin voidaan seurata hyökkäyksen etenemistä.

Tiedostot: Tiedostojen tunnistaminen on ongelmallista uusien varianttien ja myöhemmin esiteltyjen polymorfisten menetelmien takia, mutta se on silti mahdollista. Jos kyseessä on uusi haittaohjelma jota virustorjunta ei tunnista, se kannattaa lähettää virustorjuntayritykselle analysoitavaksi tai harkita isäntäpohjaisen tunkeutumisenestojärjestelmän (HIDS) käyttöä. Jotkut virustorjuntaohjelmat kuten ClamAV antavat käyttäjän myös lisätä omia tunnistussääntöjä (Ligh et al. 2010, s. 54)

3 Haittaohjelman analyysi

Kun epäilyttävä ja potentiaalisesti haitallinen ohjelma on löydetty, halutaan siitä tietää yleensä lisää. Kendall ja McMillan 2007 esittää sarjan käytännön kysymyksiä, jotka helposti käyvät ohjelman löytäneen mielessä:

- Mitä varten ohjelma on laadittu?
- Miten sen on päässyt järjestelmään?
- Kuka sen on lähettänyt ja kuinka taitavia he ovat?
- Miten siitä pääsee eroon?
- Onko se kenties varastanut meiltä jotain?
- Kuinka pitkään se on ollut täällä?
- Leviääkö se itsestään?
- Miten pystymme löytämään sen muista järjestelmistä?
- Miten voin estää vastaavanlaiset tapaukset tulevaisuudessa?

Osa näistä ja muista mieltä askarruttavista kysymyksistä selviää tutkimalla haittaohjelman rakennetta ja käytöstä tarkemmin. Kahtiajako rakenteeseen ja käytökseen on oleellinen, sillä se määrittelee analyysin kaksi muotoa: staattisen ja dynaamisen.

3.1 Staattinen analyysi

Staattisessa analyysissa tarkastellaan ohjelmaa suorittamatta sitä. Jo ilman assembly-kielen tuntemusta ohjelmatiedostosta voidaan selvittää esimerkiksi mitä kirjastokutsuja se mahdollisesti tekee ja mitä merkkijonoja se sisältää. Pienemmilläkin yksityiskohdilla kuten tiedoston koolla ja muutospäivämäärillä voi olla merkitystä kokonaisuuden hahmottamisen kannalta. Tässä luvussa tutustutaan näihin tekniikoihin.

3.1.1 Tiedoston visuaalinen tarkastelu

Tiedosto voi antaa jo päällepäin vihjeitä itsestään ja sen tekijästä. Aivan ensimmäisenä kannattaa varmistaa että tiedostoselain näyttää tiedostojen tarkenteet. Tiedostoilla voi olla kak-

sinkertaisia tarkenteita kuten .txt.vbs tai .jpg.exe, jotka yhdessä harhaanjohtavan kuvakkeen ja tarkenteiden piilotuksen kanssa antavat käyttäjälle väärän turvallisuuden tunteen. Esimerkiksi vuonna 2000 laajasti levinnyt VBS.LoveLetter-mato käytti hyväkseen Windowsissa oletuksena päällä olevaa tarkenteiden piilotusta. (Chien 2002)

Tiedoston aikaleimat voivat antaa kiinnostavaa tietoa siitä, milloin haittaohjelma on päässyt järjestelmään, milloin sitä on viimeksi käytetty ja mahdollisesti jopa milloin se on kirjoitettu. Jos tiedostoja on useampia ja osalla niistä on uudempi aikaleima, se voi kertoa itseään päivittävästä haittaohjelmasta. Aikaleimojen tarkastelua kutsutaan tiedostojen temporaaliseksi analyysiksi (Chow et al. 2007) ja sitä käytetään rikosteknisessä tutkimuksessa. Aikaleimat ovat väärennettävissä joten ne eivät sellaisenaan kelpaa todistusaineistoksi, mutta etenkin ketjuina ne voivat tukea olemassaolevaa todistusaineistoa.

Muutokset tiedoston sisältöön voi aikaleimaa varmemmin nähdä siitä lasketusta tarkistussummasta (eng. hash, checksum). Tarkistussumma on merkkijono, joka on laskettu koko tiedoston sisällöstä, ja siten kun sisältö muuttuu, myös tarkistussumma muuttuu. Kirjoitushetkellä selkeästi suosituin algoritmi tarkistussummien laskemiseen on MD5. [1]

3.1.2 Merkkijonot

Useimmat tiedostot, myös ajettavat ohjelmat, sisältävät selväkielisiä merkkijonoja. Ohjelmien tapauksessa merkkijonot voivat olla esimerkiksi ikkunoissa näkyviä tekstejä, IP- tai DNS-osoitteita, dynaamisesti ladattavien funktioiden nimiä tai virheilmoituksia. Jos haittaohjelma sisältää vaikkapa sähköpostin lähettämiseen käytettyjä SMTP-komentoja, on todennäköistä että kyseessä on roskapostimato. (Sikorski, Honig ja Lawler 2012, s. 11-13)

Merkkijonot on tyypillisesti koodattu joko ASCII- tai Unicode-notaation mukaisesti. ASCII-koodauksessa yhtä merkkiä vastaa yksi tavu, ja jokainen merkkijono päättyy tavuun 00. Unicodessa vastaavat merkit ovat samoja kuin ASCII:ssa mutta merkkiä varten on varattu kaksi tavua. Loppumerkki eli nollaterminaattori on vastaavasti kaksi nollatavua.

Merkkijonoja voi yrittää metsästää tiedostosta heksaeditorilla tai jopa tekstinkäsittelyohjel-

1. Valmiita MD5-laskureita löydät liitteestä A.

malla, mutta se ei ole kovin tehokasta. Sen sijaan kannattaa käyttää apuohjelmaa; eräs hyvin suosittu työkalu tarkoitukseen on komentoriviohjelma strings. Strings etsii tiedostosta tai muusta syötteestä kaikki mahdolliset kolmen tai useamman kirjoitusmerkin mittaiset merkkijonot ja näyttää ne listana.

3.1.3 Dynaamisesti linkitetyt funktiot

Käytännössä kaikki ohjelmat käyttävät apunaan käyttöjärjestelmän tai muun kolmannen osapuolen tarjoamia valmiita funktioita. Nämä funktiot on yleensä koottu dynaamisesti linkitettäviin kirjastoihin (DLL), jotka rakenteeltaan muistuttavat ajettavia ohjelmatiedostoja (EXE) mutta eivät yleensä ole itsessään käynnistettäviä. Dynaamisten kirjastojen käyttö paitsi helpottaa paitsi ohjelmoijan työtä, myös käyttöjärjestelmän muistinkäyttöä: jos kirjasto on jo ladattu muistiin, sitä ei tarvitse ladata enää toista kertaa. (Eilam 2011, s. 96)

Kun kirjasto tai sen sisältämä funktio otetaan käyttöön ohjelmassa, puhutaan linkityksestä. Linkitys voidaan tehdä kolmella eri tavalla: staattisesti, dynaamisesti tai ajonaikaisesti. Staattisessa linkityksessä ohjelmakoodi kopioidaan suoraan ohjelmatiedostoon ja se kasvattaa ohjelmatiedoston kokoa oman kokonsa verran. Staattisesti linkitettyä funktiota on vaikea erottaa ohjelman omista funktioista muuten kuin sen sisällön perusteella. Dynaaminen linkitys tarkoittaa sitä, että funktiot ladataan kirjastosta ohjelman käynnistyksen yhteydessä. Ajonaikainen linkitys vastaa muuten dynaamista linkitystä paitsi että funktiot ladataan vasta sitten kun niitä tarvitaan. (Sikorski, Honig ja Lawler 2012, s. 15-17)

Etenkin Windows-ympäristössä dynaaminen linkitys on yleisin tapa käyttää ulkopuolisia funktioita. Dynaamisesti linkitetyt kirjastot ja funktiot on merkitty ohjelman niin sanottuun import-tauluun, josta ne ohjelman käynnistyksessä ladataan muistiin. Import-taulun tutkimista varten on olemassa valmiita työkaluja [2].

Haittaohjelmien analyysin kannalta dynaamisesti linkitetyistä kirjastokutsuista voidaan päätellä, millainen haittaohjelma on kyseessä. Esimerkiksi funktiota ”CreateSocket” kutsuva haittaohjelma käyttää verkkoyhteyksiä johonkin tarkoitukseen ja ”RegisterHotkey” voi viitata näppäimistön kuunteluun. Luvussa 5 käydään läpi tarkemmin kiinnostavia API-funktio-

2. esim. Dependency Walker, ks. liite A

kutsuja ja mitä niistä voidaan päätellä.

Ajonaikaista linkitystä käytetään paljon haittaohjelmissa, koska sillä tavalla ladatut funktiot ja kirjastot eivät näy import-taulussa. Niiden löytämiseen vaaditaan yleensä joko ohjelman purkamista assembly-kielelle (ks. luku 5) tai dynaamista analyysiä, jossa käynnistetään ohjelma ja katsotaan mitä moduuleita se lataa.

3.1.4 PE-tiedostomuoto

PE eli Portable Executable -tiedostomuoto on 32- ja 64-bittisen Windowsin käyttämä tallennusmuoto ajettavalle ohjelmakoodille, eli EXE- ja DLL-tiedostoille. PE-ohjelmatiedostot koostuvat otsikkotiedoista ja joukosta nimettyjä sektioita. Sektioiden nimet alkavat yleensä pisteellä ja ne voivat sisältää joko koodia, dataa tai sekä että (“Microsoft PE and COFF Specification” 2013). Taulukossa 1 on esitelty analyysin kannalta tärkeimmät PE-tiedostosta löytyvät sektiot.

Sektion nimi	Sisältö
.text	Suoritettava ohjelmakoodi
.data	Ohjelmalle globaali data
.rdata	Ohjelmalle globaali, vain luettavissa oleva data
.bss	Alustamattomalle datalle varattu muistitila
.idata	Import-taulu eli käytetyt funktiot (ks. edellinen luku)
.edata	Export-taulu eli muiden käyttöön julkaistavat funktiot
.rsrc	Resurssit (ks. seuraava luku)

Taulukko 1. Tärkeimmät PE-tiedostosta löytyvät sektiot. Lähteet: Sikorski, Honig ja Lawler 2012, s. 22 ja “Microsoft PE and COFF Specification” 2013, s. 66-68.

Käytännössä mitkään näistä sektioista eivät ole pakollisia (“Microsoft PE and COFF Specification” 2013) ja joskus jos haittaohjelman kirjoittaja on ollut ovela, ne voi jopa olla nimetty harhaanjohtavasti. Sektioiden otsikkotiedoista voidaan kuitenkin päätellä, mihin sitä käytetään. Niitä voidaan tarkastella esimerkiksi PEView-ohjelmalla. Taulukossa 2 on lueteltu kullekin sektiolle määritellyt otsikkotiedot. (Sikorski, Honig ja Lawler 2012, s. 22).

Kentän nimi	Suomennos	Selitys
VirtualSize	Virtuaalinen koko	Sektiota varten varattavan muistin määrä.
VirtualAddress	Suhteellinen virtuaalio-soite (RVA)	Sektion ensimmäisen tavun osoite suhteessa ohjelman alkuun muistis-sa.
SizeOfRawData	Fyysinen koko	Sektion koko tiedostossa.
PointerToRawData	Osoitin raakadataan	Tiedosto-osoitin ensimmäiseen sektion sivuun muistissa.
Characteristics	Ominaisuudet	Tietoja siitä miten muistialuetta voidaan käyttää (esim. luku, kirjoi-tus, suoritus)

Taulukko 2. Sektion otsikkotiedot PE-tiedostossa. Lähde: "Microsoft PE and COFF Specification" 2013, s. 24.

Ohjelmakoodin (.text-sektio) koko muistissa ja levyllä on yleensä sama tai lähes sama, mutta jos se on merkittävästi suurempi muistissa, se voi olla merkki pakkauksesta (Sikorski, Honig ja Lawler 2012, s. 23). Pakattu ohjelma sisältää vain sen verran näkyvää koodia, että se pystyy purkamaan varsinaisen suoritettavan ohjelmakoodin toisesta, dataksi merkitystä sektiosta ajon aikana [3].

3.1.5 Ohjelmaan käännetyt resurssit

Ohjelman graafiset elementit kuten kuvakkeet, valikot ja ikkunat, sekä myös lokalisoidut merkkijonot, on yleensä tallennettu resursseina. Resurssit löytyvät ohjelman .rsrc-sektiosta omassa käännetyssä skriptimuodossaan. Niiden tarkasteluun ja muokkaamiseen on olemassa valmiita työkaluja kuten Resource Hacker [4].

Haittaohjelman resurssien tarkastelu voi paljastaa ohjelmasta esimerkiksi piilotettuja, ylläpitäjälle tarkoitettuja hallintaikkunoita. Resurssit eivät ole myöskään rajoitettuja GUI-ele-

3. ks. luvut 4.1.6. ja 6.1.2.
4. ks. liite A

mentteihin, vaan niistä voi löytyä myös ohjelmatiedostoja tai muuta ajettavaa koodia kuten skriptejä.

3.1.6 Staattista analyysia vaikeuttavia tekniikoita

Haittaohjelmat ovat yleensä pieniä ja kevyitä, mutta vain kymmentenkin kilotavujen kokoinen ohjelma voi paljastaa yllättävää monimutkaisuutta. Siksi kannattaakin jättää yksityiskohdat viimeiseksi ja keskittyä ohjelman yleisen rakenteen ja perustoiminnallisuuksien ymmärtämiseen. (Sikorski, Honig ja Lawler 2012, s. 5)

Joskus haittaohjelma voi näyttää siltä ettei siitä saa mitään tietoa: automaattiset analyysityökalut näyttävät enimmäkseen tyhjiä kenttiä ja assembly-käskyjen sijaan ohjelma näyttää sisältävän pelkkää dataa. Tällöin kyseessä on obfuskoitu ja/tai pakattu haittaohjelma. Pakatun ohjelman tunnistaa myös siitä, ettei se sisällä juuri mitään tunnistettavia merkkijonoja. Obfuskaatio tai pakkaus on ensin purettava ennen kuin ohjelmaan pääsee käsiksi.

Pakattu ohjelma koostuu kahdesta osasta, pakatusta datasta ja sen purkavasta kääreestä (eng. wrapper). Pakattu data sisältää alkuperäisen ohjelman jollain pakkausalgoritmilla tiivistetyssä muodossa. Käynnistettäessä kääreohjelma varaa alkuperäisen, pakkaamattoman ohjelman kokoisen alueen muistista, purkaa sinne alkuperäisen ohjelman ja siirtää sille suorituksen (Sikorski, Honig ja Lawler 2012, 2. 13-14). Haittaohjelmien tekijöille pakkauksesta on se hyöty, että ohjelma näyttää staattisessa tarkastelussa aivan erilaiselta. Pelkkä pakkausalgoritmin vaihto voi tehdä ohjelmasta tunnistamattoman virustorjuntaohjelmille ja antaa sille lisäaikaa.

Tarkastelemme pakkausta tarkemmin luvussa 6. Valmiista pakkausohjelmista erityismainitsemisen ansaitsee UPX [5], erittäin suosittu open source -pakkausohjelma (Dinaburg et al. 2008, s. 59) joka osaa myös purkaa pakkauksen. UPX-pakattujen haittaohjelmien määrä on vähentynyt, koska suurin osa virustorjuntaohjelmista ja disassemblereistakin (esim. IDA Pro, ks. liite A) purkaa sen jo läpinäkyvästi.

Tunnettujen pakkausohjelmien tunnistamiseen on myös valmiita työkaluja, kuten PEiD (ks.

5. ks. liite A

liite A) jonka kehitys on lopetettu, mutta se on silti käyttökelpoinen ohjelma. PEiD antaa ohjelmasta myös muita tietoja jotka voivat auttaa analyysissa. (Sikorski, Honig ja Lawler 2012, s. 14).

3.2 Dynaaminen analyysi

Dynaamisessa analyysissä käynnistetään haittaohjelma ja seurataan sen käytöstä, eli miten se vaikuttaa ympäristöönsä. Haittaohjelmaa varten rakennetaan oma koeympäristö eli ”hiekkalaatikko” joka on eristetty muusta verkosta. Koeympäristö voi olla joko reaalinen tietokone tai tietokoneverkko, tai nykyään yleisemmin virtuaalinen. Dynaamiseen analyysiin siirrytään yleensä vasta staattisen analyysin jälkeen, sillä staattisessa analyysissä voi selvitä, mitä kannattaa seurata ja millaisia varotoimenpiteitä on syytä käyttää. (Sikorski, Honig ja Lawler 2012, s. 2-3)

Siinä missä staattinen analyysi keskittyy ohjelmatiedostoon, dynaamisesa analyysissä ollaan kiinnostuneita prosessista. Ohjelmasta tulee prosessi kun käyttöjärjestelmän lataaja on varannut sille oman muistialueensa, kopioinut ohjelmakoodin ja datan sinne massamuistilta (mukaanlukien tarvittavat moduulit) ja siirtänyt sille suorituksen. Suoritus voidaan pysäyttää debuggerilla ja tarkastella sitä käsky kerrallaan jos halutaan tarkka kuva siitä mitä se tekee, mutta myös sen vaikutuksia ympäristöön tarkastelemalla saadaan arvokasta tietoa haittaohjelman toiminnasta. Ympäristöllä tässä tapauksessa tarkoitetaan kaikkea mihin ohjelma voi vaikuttaa, esimerkiksi tiedostoja, muita prosesseja ja rekisteriä. Dynaaminen analyysi on yleensä helpompi suorittaa kuin staattinen, mutta haittapuolena siinä missä staattinen analyysi antaa kokonaiskuvan koko ohjelmasta, dynaamisessa analyysissä voidaan tutkia vain yhtä suorituspolkua kerrallaan. (Willems, Holz ja Freiling 2007, s. 33)

3.2.1 Koeympäristö

Ensimmäinen kysymys koeympäristöä laadittaessa on halutaanko siitä virtuaalinen vai fyysinen. Virtuaalisen ympäristön etuna on sen helppo käyttöönotto ja hallinta - yhdellä fyysisellä tietokoneella voidaan emuloida kokonaista verkkoa. Uudemmat virtualisaatioympäristöt

[6] tarjoavat myös mahdollisuuden rakentaa aikajanan järjestelmän tiloista. Edellisiin tiloihin on mahdollista palata ja luoda niistä vaihtoehtoisia aikajanoja muodostaen tilojen historiasta puumaisen rakenteen. (Kendall ja McMillan 2007)

Virtuaalikonetta rakentaessa on tärkeää muistaa konfiguroida sen verkkoasetukset sellaisiksi, ettei haittaohjelma saa yhteyttä internetiin (ellei se nimenomaan kuulu kokeen luonteeseen) eikä isäntäkoneen palveluihin. Isäntäkone eli se tietokone jolla virtuaaliympäristöä ajetaan, onkin syytä pitää palomuurilla suojattuna ja sen ohjelmistopäivitykset ajan tasalla. On pidettävä myös mielessä se (joskin harvinainen) tapaus, että haavoittuvuus virtuaaliympäristössä itsessään voi päästää haittaohjelman suorittamaan ohjelmakoodia isäntäkoneella (Raffetseder, Krügel ja Kirda 2007, s. 2).

Virtualisointi ei aina myöskään ole täydellistä – eroja saattaa löytyä esimerkiksi prosessorimallikohtaisista bugeista ja sivuvaikutuksista tai käskyjen välisistä ajoituksista (Raffetseder, Krügel ja Kirda 2007, s. 6). Myös emuloiduilla oheislaitteilla voi olla oma sormenjälkensä. Tällaiset erot voivat paljastaa haittaohjelmalle että sitä ajetaan virtuaalikoneessa ja saada sen käyttäytymään eri tavalla kuin normaaliolosuhteissa. Fyysisessä ympäristössä testaamisen lisäksi mahdollinen keino kiertää virtuaalikoneen tunnistus on poistaa se ohjelmasta, mikä onnistuu debuggerin tai disassemblerin avulla. Tarkempaa tietoa tästä löytyy luvusta 6.

Virtualisoinnille on myös vaihtoehtona emulaatio. Emulaation tekninen ero virtualisaatioon on se, että siinä missä virtuaalikone suorittaa annetut käskyt (ainakin pääosin) sellaisenaan isäntäkoneella, emulaattori emuloi koko laitteistoa ja tulkitsee käskyt suoritettavaksi emuloidulla laitteistolla. Tämä tekee emulaattoreista huomattavasti virtuaalikoneita hitaampia, mikä ei sinällään ole ongelma analyysiä ajatellen, mutta poikkeava ajoitus voi antaa emulaation tunnistavalle haittaohjelmalle ilmi että sitä ajetaan emulaattorissa. Emulaation suurimpia etuja on se, että emuloitavan laitteistoarkkitehtuurin ei tarvitse olla saman kuin isäntäkoneen laitteistoarkkitehtuurin. Lisäksi assembly-käskyjen tulkinta antaa mahdollisuuden puuttua käskyjen suoritukseen, mikä avaa mahdollisuuden esimerkiksi debuggerin tunnistuksen ja muiden analyysiä vaikeuttavien tekniikoiden ajonaikaiseen kiertämiseen. (Raffetseder, Krügel ja Kirda 2007, s. 4-5)

6. esim. VMWare, Oracle VirtualBox

Eräs käytetyimmistä emulaattoreista on Qemu (Bellard 2013), joka tukee x86:n lisäksi myös monia muita kohdearkkitehtuureja. Qemu on avointa lähdekoodia, ja sen pohjalle on kehitetty useita analyysiympäristöjä ja -palveluita erityisesti haittaohjelmien automaattiseen dynaamiseen analyysiin, esimerkkeinä Anubis ("Anubis: Analyzing Unknown Binaries" 2013; Bayer et al. 2009) ja TTAnalyze (Bayer, Kruegel ja Kirda 2006). Samantapaisia, muihin ympäristöihin perustuvia työkaluja ovat CWSandbox ("CWSandbox: Behavior-based Malware Analysis" 2013; Willems, Holz ja Freiling 2007) ja Ether ("Ether: Malware Analysis via Hardware Virtualization Extensions" 2013; Dinaburg et al. 2008), joista ensimmäinen on Anubiksen tapaan verkossa toimiva analyysipalvelu, ja jälkimmäinen laitteistopohjaiseen virtualisaatioon perustuva, lokaalisti ajettava ratkaisu. Analyysin automatisoinnin syvällisempi tarkastelu jää tämän tutkielman ulkopuolelle, mutta analyysiympäristöihin palataan myöhemmin empiirisessä osassa sekä luvussa 6.

3.2.2 Muutosten seuranta järjestelmässä

Ohjelman järjestelmään tekemien muutosten seuraamiseen on kolme eri tapaa: API-koukut, tilamuutosten seuranta ja järjestelmänotifikaatiot. Tilamuutosten seuranta vaatii hieman erilaisia työkaluja, esimerkkinä Winanalysis ja Regshot, jotka on ajettava ennen ja jälkeen haittaohjelman suoritusta mutta kaksi muuta toimivat taustalla seuraamassa tapahtumia reaaliaikaisesti. Työkalujen toimintaperiaatetta ei tarvitse välttämättä tietää osatakseen käyttää niitä, mutta siitä voi olla hyötyä esimerkiksi jos haittaohjelma yrittää syöttää niille väärää tietoa.

API-koukku toimii kääreenä API-funktiolle johon se liitetään – se saa kohdefunktiolle annetut parametrit, kutsuu kohdefunktiota tai toteuttaa sen omalla tavallaan ja lopuksi palauttaa sen palautusarvon. API-koukut antavat yksityiskohtaista tietoa järjestelmäkutsusta ja siksi se on käytetty tapa myös haittaohjelmissa, erityisesti vakoiluohjelmissa. (Willems, Holz ja Freiling 2007, s. 33)

Tilamuutosten seuranta tarkoittaa järjestelmän tilan tai sen osatilojen tallentamista ennen tuntemattoman ohjelmakoodin suoritusta ja sen jälkeen, ja näiden tilojen vertailua. Sen varjopuolena on sokeus nopeille muutoksille kuten väliaikaistiedostojen luomiselle ja tapahtumien järjestyksen puuttuminen. Myöskään esimerkiksi oikeuksien puuttumisen takia epä-

onnistuneita yrityksiä tämä menetelmä ei tunnista. Vahvuutena tilamuutosten määrällä tai nopeudella ei ole vaikutusta.

Järjestelmänotifikaatiot muistuttavat periaatteeltaan API-koukkuja, mutta funktiokutsujen sijaan kohteena ovat operaatiot kuten tiedoston tai rekisteriavaimen luominen. Toisin kuin koukut, ne ovat yhdensuuntaisia eli ne eivät voi esimerkiksi estää tiedoston luomista. Koukkuja ja notifikaatioita käytetään usein yhdessä prosessinseurantaohjelmien yhteydessä. (Ligh et al. 2010, s. 284)

3.2.3 Muutokset tiedostoihin ja rekisteriin

Yleensä haittaohjelma asentaa itsensä ensimmäisellä ajokerralla joko kopioimalla suoritettavan tiedoston, purkamalla esimerkiksi sen resursseihin tallennettuja tiedostoja tai lataamalla verkosta uusia. Jotta haittaohjelma käynnistyisi automaattisesti myös seuraavalla uudelleenkäynnistyksellä, sen täytyy joko kirjoittaa rekisteriin tai kiinnittää itsensä tavalla tai toisella johonkin jo valmiiksi käynnistyvään ohjelmaan. Haittaohjelma voi pitää yllä myös erilaisia lokitiedostoja esimerkiksi käyttäjän näppäimienpainalluksista tai pikaviesteistä. Suurin osa näistä operaatioista voidaan havaita tarkkailemalla muutoksia levyllä ja rekisterissä.

Tiedostoihin ja rekisteriin kohdistuvia muutoksia voidaan seurata kaikilla edellisissä luvuissa mainituilla menetelmillä, mutta pääasiassa työkalut jakaantuvat reaaliaikaisiin ja tilamuutoksiin perustuviin. Reaaliaikaisista työkaluista käytetyimpiä on Sysinternals Softwaren (nyk. osa Microsoftia) kehittämä Process Monitor, joka käyttää API-koukkuja ja järjestelmänotifikaatioita. Process Monitor tuottaa valtavan määrän tietoa paitsi kaikista järjestelmässä tapahtuvista tiedosto- ja rekisterimuutoksista, myös uusien prosessien ja säikeiden luomisesta sekä verkkoliikenteestä. API-koukkujen ansiosta myös epäonnistuneet yritykset kirjataan. Process Monitor sisältää sääntöpohjaisen suodatusjärjestelmän, jolla suuresta tietomäärästä löytää helpommin etsimänsä. Ohjelma on ladattavissa vapaasti Microsoftin verkkosivulta. (Sikorski, Honig ja Lawler 2012, s. 47) (Ligh et al. 2010, s. 286)

Tilamuutoksiin perustuvista työkaluista suosituimpia on Regshot, jolla koko rekisterin voi vedostaa tiedostoon tai muistiin ja vertailla näitä vedoksia keskenään. Nimestään huolimatta sillä voi vertailla myös tiedostoja. Regshot on avointa lähdekoodia ja on saatavissa Source-

forge.net -sivustolta. (Ligh et al. 2010, s. 288)

3.2.4 Kahvat

Yksi hyödyllinen mutta helposti ylenkatsottu tilatieto Windowsissa on prosessien varaamat kahvat. Kun mikä tahansa olio kuten tiedosto, rekisteriavain tai prosessi avataan, Windows antaa kahvan jolla siihen voi viitata. Kuva 2 antaa esimerkin tiedoston avaamisesta API-funktiolla CreateFile.

```
HFILE handle = CreateFile(
    _T("tiedoston nimi"),
    FILE_READ_DATA, FILE_SHARE_READ,
    NULL, OPEN_ALWAYS, 0, NULL);
```

Kuvio 2. CreateFile-funktion kutsu C-kielellä.

Kun tiedosto-operaatiot on saatu päätökseen, voidaan kahva sulkea CloseHandle-funktiolla. Siitä eteenpäin kahvaa ei voi enää käyttää. Kahvoilla Windows pitää kirjaa siitä mitkä prosessit käyttävät mitäkin resursseja, ja niitä seuraamalla voidaan myös selvittää paitsi mitä resursseja haittaohjelman itsensä prosessi käyttää, myös mitä vaikutuksia sillä on muihin prosesseihin. Näin saadaan kiinni koodia toisiin prosessiin injektoivat haittaohjelmat kuten Zeus (Ligh et al. 2010, s.). Prosessin avaamien kahvojen tarkasteluun on erilaisia ohjelmia, kuten Sysinternalsin suoraviivainen komentorivityökalu Handle.exe ja Wen Jia Liun monipuolinen open source -tehtävänhallintaohjelma Process Hacker.

Prosessikahvojen tapauksessa eräs mielenkiintoinen tarkastelun kohde on csrss.exe (Client / Server Runtime Subsystem). Se sisältää listan kaikista käyttäjän oikeuksilla käynnistetyistä prosesseista, myös niistä joita ei tehtävänhallintaikkunan kautta näe. Prosessi voi yrittää piilottaa itsensä myös CSRSS:n prosessitaulusta, mutta se vaatii kahvan avaamista CSRSS-prosessiin, mikä puolestaan jättää oman jälkensä. (Ligh et al. 2010, s. 297)

Koodi-injektiohyökkäysten ja vastaavan toisten prosessien manipuloinnin tunnistusta varten voidaan vertailla prosessikohtaista listaa kaikista järjestelmän kahvoista ennen ja jälkeen haittaohjelman käynnistyksen. Lähteen Ligh et al. 2010 mukana tulevalta DVD-levyltä löytyy siihen soveltuva ohjelma handlediff, joka on saatavilla lähdekoodeineen kirjan verkko-

sivuilta [7]. Vaihtoehtoinen tapa on käyttää Sysinternalsin Handle-työkalua ilman parametreja ja ohjata tulostus tiedostoihin joita voi myöhemmin vertailla esimerkiksi diff-ohjelmalla.

3.2.5 Verkkoliikenne

Verkkoliikenne on yksi tärkeimmistä seurantakohteista nykypäivän haittaohjelmissa. Verkon kautta haittaohjelma voi esimerkiksi yrittää levittää itseänsä eteenpäin, ottaa yhteyttä komentopalvelimeen tai ladata ohjelmistopäivityksiä tai lisäosia. Jotkut haittaohjelmat myös suorittavat DDoS-hyökkäyksiä tai lähettävät suuria määriä roskapostia. Nämä ovat jo hyviä syitä miksi haittaohjelmaa ei kannata päästää verkkoon vaikka se vaikuttaisikin helpolta tavalta saada tietoa sen käytöksestä. Riskinä voi olla myös haittaohjelman kehittäjän puuttuminen peliin ja analyysin muuttuminen reaaliaikaiseksi verkkosodaksi. Siksi, jos vain mahdollista, kannattaa käyttää simuloituja verkkopalveluita todellisten sijasta. (Kendall ja McMillan 2007, s. 3)

Nopeammin käyttöönotettavana vaihtoehtona palvelinohjelmistojen erilliselle asennukselle ovat valmiit verkkosimulaattorit. Eräs ilmainen haittaohjelmien analysointiin tarkoitettu ja siten hyvin konfiguroitavissa oleva vaihtoehto on INetSim [8]. INetSim simuloi kaikkia tunnetuimpia protokollia kuten DNS, HTTP ja IRC, ja on konfiguroitavissa myös generoimaan erilaisia tiedostotyyppejä kuten JPEG-kuvia vastauksena palvelupyyntöihin. INetSim kirjaa myös kaiken käsittelemänsä verkkoliikenteen lokiin. (Sikorski, Honig ja Lawler 2012, s. 55)

Verkkoliikenteen tarkkailuun voidaan käyttää myös paketinkaappausohjelmaa, kuten Wiresharkia. Simulaattorin tapauksessa paketinkaappausohjelmaa ei välttämättä tarvita, mutta se on silti hyvä lisä sellaisten protokollien tapauksessa, joita simulaattori ei tunnista. Paketinkaappausohjelmat käyttävät ajuria, joka kaappaa kaiken tulevan ja lähtevän verkkoliikenteen määrätyltä verkkosovittimelta. Paketit näkyvät listassa, josta niiden sisällön voi purkaa kehys kerrallaan fyysiseltä kerrokselta aina sovellusprotokollakerrokselle saakka. Pakettien tarkastelu paljastaa paitsi mihin osoitteisiin ja portteihin haittaohjelma yhdistää, myös mitä dataa se yrittää siirtää. Näiden tietojen perusteella voidaan hioa protokollan simulaatiota entistä tarkemmaksi. On kuitenkin pidettävä mielessä että kaappausohjelmaa ajetaan samas-

7. http://www.malwarecookbook.com
8. http://www.inetsim.org

sa ympäristössä kuin haittaohjelmaa, ja siten analyysille erityisen herkkä haittaohjelma voi tunnistaa sen ja kehittää vastatoimia. (Sikorski, Honig ja Lawler 2012, s. 53)

4 Assembly-kieli ja 32-bittinen Windows

Analyysin jatkaminen edellisen luvun tekniikoista eteenpäin alkaa vaatia yhä enemmän ja yksityiskohtaisempaa tietoa suoritusympäristöstä (laitteisto + käyttöjärjestelmä) ja ajettavan ohjelman rakenteesta käskytasolla. Tämän luvun tarkoituksena on antaa nämä tarvittavat esitiedot lukijalle, jolle ne eivät ole entuudestaan tuttuja.

4.1 Lyhyt johdatus assembly-kieleen x86-arkkitehtuurissa

Kirjoitushetkellä yleisin PC-tietokoneissa käytetty laitteistoarkkitehtuuri on Intel x86 [1] tai sen 64-bittinen laajennus x86-64 [2]. Suurin osa haittaohjelmista on käännetty 32-bittiselle laitteistoarkkitehtuurille, joten siihen keskittyminen on mielekästä. Laitearkkitehtuuri määrää käytössä olevan käskykannan ja siten käytettävän assembly-kielen, itse tekniikat kuten debuggaus ja kontrollivuokaavioiden luominen ovat arkkitehtuurista riippumattomia. (Sikorski, Honig ja Lawler 2012, s. 68)

4.1.1 Assembly-kielen määritelmä

Assembly-kieli on symbolinen esitys konekielestä. Niinpä ei ole vain yhtä assembly-kieltä, vaan jokaiselle prosessoriarkkitehtuurille on omansa. Käskyt koostuvat operaatiosta ja nollasta tai useammasta operandista. Assembly-kielellä operaatio esitetään sitä lyhyesti kuvaavalla sanalla (1 – 4 kirjainta), esimerkiksi MOV tai PUSH. Operandit tulevat järjestyksessä operaation jälkeen, yleensä kohdeoperandi ensimmäisenä. Esimerkiksi x86-assemblykäsky MOV EAX, 3 siirtää luvun 3 rekisteriin EAX. Tärkeimmät käskyt esimerkkeineen löytyvät liitteestä B ja täydellinen käskykanta konekielisine vastineineen on saatavilla Intelin arkkitehtuurikäsikirjan “Intel 64 and IA-32 Architectures Software Developers Manual” 2013 osasta 2. (Eilam 2011, s. 11)

Assembly-kielisen tekstitiedoston kääntäminen ajettavaksi ohjelmaksi onnistuu assemble-

1. x86 tunnetaan myös nimellä IA-32
2. Intel kutsuu omaa x86-64 -arkkitehtuuriaan nimellä Intel 64 ja AMD:n vastaava on AMD64, mutta kyse on silti samasta asiasta

riksi kutsutun ohjelman avulla. Assembler kääntää tekstimuotoiset operaatiot ja operandit prosessorin ymmärtämään binäärimuotoon. Vastaava käänteinen toimenpide voidaan tehdä disassemblerilla, joka onkin binäärianalyysissä (mukaanluettuna haittaohjelmien analyysi) tärkeimpiä työkaluja. Disassemblerit eivät kuitenkaan ole täydellisiä ja niitä voidaan harhauttaa erilaisilla tekniikoilla (obfuskaatio), minkä takia on hyvä tuntea myös jonkin verran konekieltä näiden tekniikoiden kiertämiseksi. (Eilam 2011, s. 11)

4.1.2 Rekisterit ja muistiavaruus

x86-arkkitehtuurissa matalalla tasolla työskenneltäessä tärkeimpiä tietovarastoja ovat prosessorin rekisterit ja muistiavaruus. Koska x86 on 32-bittinen prosessoriarkkitehtuuri, kaikki rekisterit ovat myös 32-bittisiä, eli toisin sanoen niihin mahtuu yksi etumerkitön kokonaisluku väliltä $[0, 2^{32}[$ tai etumerkillinen kokonaisluku väliltä $[-2^{31}, 2^{31}[$. Rekistereitä voidaan käyttää suoraan joko kokonaislukujen laskutoimituksiin, tai osoittimina osoittamaan esimerkiksi pinomuistissa olevaan merkkijonoon. Rekisterit ja niiden tyypillisimmät käyttötarkoitukset on listattu taulukossa 6. Muistin osoitukseen tarkoitetut segmenttiregisterit, joita on kuusi kappaletta (CS, SS, DS, ES, FS, GS), on jätetty pois taulukosta koska niille ei ole juurikaan käyttöä Windows-ympäristössä mutta niiden olemassaolo on silti hyvä tiedostaa (Eilam 2011, s. 149).

Useimpien rekistereiden kuuttatoista alinta bittiä voidaan käsitellä omana rekisterinään pudottamalla siitä ensimmäinen E-kirjain pois, jolloin esimerkiksi EAX:stä tulee AX. Vastaavasti esimerkiksi AX:rekisterin korkeampaa ja matalampaa tavua voidaan osoittaa symbolien AH ja AL avulla.

EFLAGS eli lippurekisteri sisältää bittimuotoista tietoa prosessorin tilasta. Lippujen tilaa voi muuttaa niille tarkoitetuilla käskyillä tai sijoittamalla EFLAGS-rekisteriin. Monet etenkin laskentaan ja vertailuun käytetyt käskyt muuttavat lippuja. Täydellinen lista lipuista ja käskyjen vaikutuksista niihin löytyy Intelin arkkitehtuurikäsikirjasta “Intel 64 and IA-32 Architectures Software Developers Manual” 2013. Debuggauksen (dynaaminen analyysi) kannalta oleellisimmat liput on lueteltu taulukossa 7.

Muistiavaruus jakautuu pino- ja kekomuistiin sekä ohjelman koodi- ja data-alueisiin. Pinoa

käytetään yleensä parametrien välittämiseen funktiokutsujen yhteydessä ja lokaalien muuttujien tallentamiseen. Keko taas puolestaan toimii dynaamisena muistina jota ohjelma voi varata ja vapauttaa tarpeen mukaan. Esimerkiksi oliokielissä kuten C++ ei-lokaalit oliot luodaan yleensä kekomuistiin. Koodi- ja data-alueet vastaavat exe-tiedoston vastaavia sektioita .code (suoritettava ohjelmakoodi) ja .data (globaalit muuttujat). (Sikorski, Honig ja Lawler 2012, s. 69)

4.1.3 Hello World -esimerkkiohjelma ja yhteys konekieleen

Kuva 3 esittelee lyhyen Hello World -ohjelman assembly-kielellä.

```
lea   eax, HelloWorld
mov   ecx, 0
push  ecx
push  eax
push  eax
push  ecx
call  MessageBoxA
push  ecx
call  ExitProcess
```

Kuvio 3. Hello World -ohjelma assembly-kielellä.

Symbolien HelloWorld (merkkijono), MessageBoxA ja ExitProcess (funktioita) määrittelyt vaihtelevat kääntäjästä riippuen, mutta binääritiedostojen analyysin kannalta ne eivät ole oleellisia. Assembler muuttaa ne osoittimiksi, funktioiden tapauksessa import-tauluun, ja datan tapauksessa datasektioon, jonka lataaja kopioi exe-tiedostosta vastaavalle alueelle prosessin muistissa.

Assemblerilla kääntämisen jälkeen myös käskyt muuttavat muotoaan konekielisiksi. Konekielellä operaatiot on esitetty operaatiokoodien (eng. opcode) avulla, jotka kukin ovat pituudeltaan yhdestä kolmeen tavua. Operaatiokoodin jälkeen tulevat sille operaatiolle yksilölliset operandit koodattuina erityisellä tavalla joka ottaa huomioon sen, osoitetaanko data muistista, rekisteristä vai annetaanko se parametrina. Yksityiskohdat tästä jätetään myöhempiin tarkasteluihin. (“Intel 64 and IA-32 Architectures Software Developers Manual” 2013, Vol. 1, chapter 2.1)

Kuvassa 4 on esitetty ylläoleva Hello World -ohjelma konekielelle käännettynä.

```
8D 05 xx xx xx xx        lea   eax, HelloWorld
B9 00 00 00 00           mov   ecx, 0
51                       push  ecx
50                       push  eax
50                       push  eax
51                       push  ecx
E8 yy yy yy yy           call  MessageBoxA
51                       push  ecx
FF 25 zz zz zz zz        call  ExitProcess
```

Kuvio 4. Hello World -ohjelma konekielellä ja assembly-muodossa. xx, yy ja zz ovat symbolien muistiosoitteita, rivinvaihdot ovat vain käskyjen rajan havainnollistamiseksi. Todellisuudessa käskyn ensimmäiset tavut määräävät sen pituuden.

Operaatiokoodit ja niiden tarvitsemat operandit löytyvät Intelin arkkitehtuurikäsikirjasta ("Intel 64 and IA-32 Architectures Software Developers Manual" 2013, Vol. 2, chapter 2.1.). Ne voivat tulla tarpeeseen jos ohjelma on obfuskoitu tai sen toiminnallisuutta halutaan muuttaa jälkikäteen heksaeditorilla.

4.1.4 Funktiokutsut ja C-kutsukäytäntö

x86 tarjoaa käskyn **CALL** aliohjelmakutsuja varten. CALL ottaa parametrikseen muistiosoitteen johon siirrytään. Paluu takaisin kutsuvaan ohjelman osaan tapahtuu käskyllä **RET**. Todellisuudessa mitä tapahtuu, call-käsky laittaa pinoon seuraavan suoritettavan käskyn osoitteen rekisteristä EIP ja vaihtaa sen uudeksi arvoksi parametrina annetun osoitteen. Vastaavasti RET-käsky ottaa paluuosoitteen pinosta ja asettaa sen seuraavaksi käskyksi. ("x86 Assembly Guide" 2013)

Puhtaassa assembly-ohjelmoinnissa parametrien ja paluuarvojen välitys on jätetty täysin ohjelmoijan harteille. Eri ohjelmointikielillä kirjoitettujen kirjastojen yhteensopivuuden takaamiseksi on kehitetty erilaisia kutsukäytäntöjä, joista käytetyimpiä on ns. C-käytäntö (cdecl). C-käytännössä jokaista funktiokutsua varten luodaan ns. pinokehys (eng. stack frame) joka sisältää parametrit, paluuosoitteen ja lokaalit muuttujat. ("x86 Assembly Guide" 2013; Sikorski, Honig ja Lawler 2012)

Funktion kutsun yhteydessä kutsuja asettaa parametrit pinoon käänteisessä järjestyksessä (ks. kuvio 5). Funktion itse tulee panna entinen EBP-rekisterin arvo pinoon ja asettaa sen uu-

```
push param3
push param2
push param1
call funktio
```

Kuvio 5. Funktiokutsu assembly-kielellä C-kutsukäytännön mukaisesti.

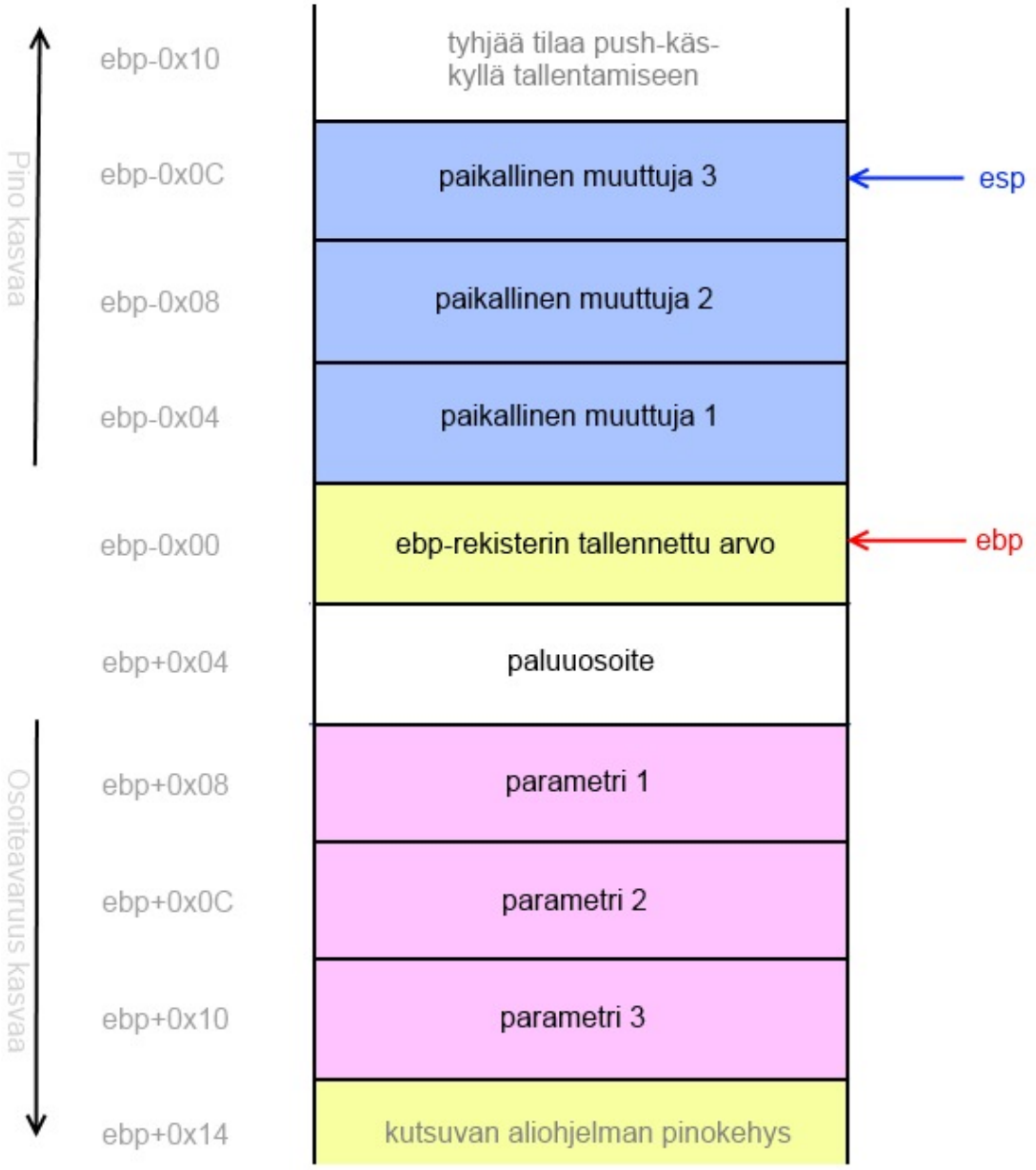
deksi arvoksi ESP-rekisterin arvo. Tätä sanotaan uuden pinokehyksen luomiseksi, tai funktion prologiksi. Esimerkki funktion prologista on esitetty kuviossa 6. Pinokehyksen hyötynä funktion kirjoittajan ei tarvitse tietää mitään pinon tilasta ennen funktion kutsua. Kun pinokehys on alustettu, EBP-osoittimeen lisäämällä voidaan viitata parametreihin ja siitä vähentämällä viitata lokaaleihin muuttujiin. (Sikorski, Honig ja Lawler 2012, s. 77)

```
funktio:
  push ebp
  mov  ebp, esp
  sub  esp, 16
  ...
```

Kuvio 6. Assembly-funktion prologi C-kutsukäytännön mukaisesti.

Näiden peräkkäisten käskyjen näkeminen disassembly-koodissa on lähes varma merkki C- tai stdcall-käytännön mukaisesta funktiosta. Kontrollivuota analysoivat disassemblerit kuten IDA Pro tunnistavat nämä automaattisesti ja osaavat luoda kutsugraafin niiden perusteella. (Sikorski, Honig ja Lawler 2012, s. 97)

Funktion sisällä pinokehys näyttää kuvan 7 mukaiselta.



Kuvio 7. Kuva pinosta funktion suorituksen aikana. Kuva piirretty lähteen “x86 Assembly Guide” 2013 perusteella.

Vastaavasti funktiosta poistuttaessa pino palautetaan alkuperäiseen tilaansa, eli lokaalit muuttujat poistetaan ja palautetaan EBP-osoittimen arvo. Tätä nimitetään funktion epilogiksi (ks. kuvio 8) (Sikorski, Honig ja Lawler 2012, s. 77).

C-kutsukäytäntö määrittelee myös joitain sääntöjä rekisterien käytölle. Funktio voi vapaasti

```
funktio:
  ...
  mov esp, ebp
  pop ebp
  ret
```

Kuvio 8. Assembly-funktion epilogi C-kutsukäytännön mukaisesti.

muuttaa rekisterien ECX ja EDX arvoja, joten niiden arvojen tallentaminen jää kutsujan vastuulle. Sen sijaan jos funktio muuttaa rekisterien EBX, ESI tai EDI arvoja, ne täytyy palauttaa alkuperäiseen arvoonsa. Rekisteri EAX on varattu funktion paluuarvolle. (“x86 Assembly Guide” 2013)

Muita suosittuja kutsukäytäntöjä ovat stdcall ja fastcall. Stdcall eroaa C-käytännöstä siten, että parametrien poistaminen pinosta jää kutsuttavan funktion vastuulle. Windowsin API-funktiot käyttävät kaikki stdcall-käytäntöä, ja siten niitä kutsuttaessa ei tarvitse poistaa parametreja pinosta. Fastcall on nimensä mukaisesti nopeampi tapa sillä se käyttää pinon sijaan rekistereitä (ECX, EDX) ensimmäisten parametrien välitykseen. (Sikorski, Honig ja Lawler 2012, s. 120)

On huomattava, että nämä kutsukäytännöt ovat oleellisimpia 32-bittistä koodia tarkasteltaessa. 64-bittisessä arkkitehtuurissa rekistereitä on enemmän ja niitä käytetään myös parametrien välitykseen hieman fastcall-käytännön tapaan. Microsoftin 64-bittinen poikkeustenhallinta myös vaatii että 64-bittinen funktio ei saa muuttaa pinon kokoa kesken sen suorituksen. 64-bittiset haittaohjelmat jäävät kuitenkin tämän tutkielman ulkopuolelle. (Sikorski, Honig ja Lawler 2012, s. 444)

4.1.5 Keskeytykset ja poikkeukset

x86-prosessori suorittaa käskyjä lineaarisesti, mutta keskeytys voi siirtää suorituksen toisaalle joko väliaikaisesti tai pysyvästi. Keskeytykset jakaantuvat karkeasti kolmeen tyyppiin: poikkeuksiin, laitekeskeytyksiin ja ohjelmallisiin keskeytyksiin. (“OSDev.org Wiki: Interrupts” 2014)

Kuten korkeamman tason kielissäkin, poikkeus tapahtuu kun suoritettava operaatio epäon-

nistuu jollain tapaa. Syy voi olla esimerkiksi jako nollalla tai luku virheellisestä muistiosoitteesta. Poikkeuksia voi tapahtua myös normaalitilanteessa, kuten muistin sivutuksen yhteydessä. Ne ovat oleellisia myös debuggerin toiminnan kannalta – poikkeuksilla 1 (suoritus käsky kerrallaan) ja 3 (keskeytyskohta) debuggeri voi kontrolloida ohjelman suoritusta. (Sikorski, Honig ja Lawler 2012, s. 175–176)

Laitekeskeytyksiä käytetään asynkroniseen kommunikaatioon oheislaitteiden kanssa. Kun esimerkiksi näppäimistön näppäintä painetaan, tapahtuu keskeytys, jonka keskeytyskäsittelijä voi ”ottaa kiinni” ja esimerkiksi tulostaa näppäintä vastaavan merkin ruudulle. (“OSDev.org Wiki: Interrupts” 2014)

Käyttöjärjestelmät käyttävät yleensä ohjelmallisia keskeytyksiä tarjoamaan järjestelmäkutsuja, eli ytimen (kernel) palveluita käyttäjätilan ohjelmille. Näihin kuuluu esimerkiksi prosessien hallinta sekä tiedosto- ja laite-IO. Windowsissa keskeytykset on kapseloitu DLL-kirjastokutsujen sisälle (Dang, Gazet ja Bachaalany 2014, s. 97), mutta esimerkiksi Linuxissa ja muissa Unix-pohjaisissa käyttöjärjestelmissä niitä käytetään usein myös suoraan. (“OSDev.org Wiki: Interrupts” 2014)

Keskeytykset sijaitsevat keskeytysvektoritaulukossa, jonka osoite on tallennettu erityiseen IDTR-rekisteriin. Kukin taulukon alkio sisältää muistiosoitteen vastaavan keskeytyksen käsittelevälle funktiolle. Keskeytyksiä on kaikkiaan 256 kappaletta, joista 32 ensimmäistä on varattu prosessorin käyttöön, ja loput ovat vapaasti asetettavissa. (Dang, Gazet ja Bachaalany 2014, s. 95)

Koska debuggeri on nimensä mukaisesti tarkoitettu virheiden jäljitykseen ohjelmista, se on suunniteltu ottamaan kiinni poikkeukset ennen ohjelman potentiaalista kaatumista. Haittaohjelmien tutkimuksessa tätä ei yleensä haluta (debuggerin käyttämiä poikkeuksia 1 ja 3 lukuunottamatta), ja toiminnallisuus kannattaakin kytkeä pois päältä debuggerin asetuksista. Jotkut haittaohjelmat jopa käyttävät poikkeuksia tunnistamaan debuggerin – jos poikkeusta ei tapahdukaan ohjelmassa, se tarkoittaa että debuggeri on käsitellyt sen. (Sikorski, Honig ja Lawler 2012, s. 175–176)

Windows-ohjelmissa poikkeusten käsittelyyn käytetään ns. rakenteista poikkeuskäsittelyä (SEH, Structured Exception Handling), jossa kullekin säikeelle on oma poikkeuksenkäsit-

telijälistansa. Lista sijaitsee fs-rekisterin osoittamassa muistialueessa (Thread Information Block), joka on yksilöllinen kullekin ohjelman säikeelle. Ohjelma voi joko lisätä oman poikkeuksenkäsittelijänsä suoraan listaan, tai kutsua Windows API:n RaiseException-funktiota. Haittaohjelmat voivat hyödyntää tätä esimerkiksi lisäämällä oman poikkeuskäsittelijänsä olemassaolevaan prosessiin tai sekoittamalla suoritusjärjestystä analyysiä vaikeuttamaan. (Eilam 2011, s. 105–107)

4.2 32-bittiset Windows-ohjelmat

Assembly-kielen lisäksi on oleellista tuntea myös käyttöjärjestelmä, jolle tutkittava haittaohjelma on kirjoitettu. Käyttöjärjestelmä tarjoaa valmiita kirjastokutsuja korkean tason objektien kuten ikkunoiden ja verkkoyhteyksien käsittelyyn. Näistä kirjastokutsuista ja niiden määrittelemästä rajapinnasta käytetään nimitystä API, joka tulee sanoista Application Programming Interface, ohjelmistokehitysrajapinta. Windowsin API-dokumentaatio löytyy Microsoftin MSDN-sivustolta [3], kuten myös Platform SDK, joka sisältää oleellista tietoa ja työkaluja matalan tason Windows-ohjelmointiin. Windows sisältää myös lukuisia dokumentoimattomia API-funktioita joita haittaohjelmat voivat käyttää, mutta myös osalle niistä löytyy toisen käden dokumentaatiota [4].

Windowsissa ohjelmia on käytännössä kahta päätyyppiä, komentoriviltä ajettavia ja ikkunoituja. Komentoriviltä ajettavat ohjelmat ovat rakenteeltaan lineaarisempia, kun taas ikkunoidut ohjelmat perustuvat jatkuvaan tapahtumien kuunteluun silmukassa. Kolmanneksi alatyypikseen voitaisiin laskea dynaamisesti linkitettävät kirjastot, DLL:t.

Edelleen ohjelmat voidaan jaotella käyttäjä- ja kernel-ympäristössä toimiviin. Kernel on käyttöjärjestelmän ydin, ja sen kanssa samalla tasolla toimivat lähinnä laitteistoajurit, mutta myös monet palomuuri- ja virustorjuntaohjelmat. Kernel-ohjelmointi ei kosketa suurinta osaa ohjelmoijista, mutta haittaohjelmista erityisesti rootkitit toimivat useimmiten kernel-tilassa. Kernel-tilassa toimiva haittaohjelma voi vaikuttaa suoraan käyttöjärjestelmän ja ajureiden toimintaan, ja siten esimerkiksi piilottaa itsensä tiedostojärjestelmästä tai ohittaa pa-

3. Microsoft Developer Network, http://msdn.microsoft.com
4. esimerkiksi sivustolta http://undocumented.ntinternals.net/

lomuurin. Kernel-ohjelmien tekeminen on kuitenkin paljon työläämpää kuin käyttäjätason ohjelmien, sillä yksikin bugi voi kaataa koko käyttöjärjestelmän. (Sikorski, Honig ja Lawler 2012, s. 158)

4.2.1 Standardi komentoriviohjelma

Komentoriviohjelma Windows-ympäristössä ei juurikaan eroa Linux- tai DOS-vastineestaan (ks. kuvio 9). Suoritus alkaa main-metodin alusta ja päättyy sen loppuun. Main-metodi saa parametreinaan ohjelmalle komentoriviltä annettujen argumenttien määrän (argc) ja niiden sisällön (argv). (Sikorski, Honig ja Lawler 2012, s. 83)

```
int main ( int argc, char** argv )
{
   // ...
   return 0;
}
```

Kuvio 9. Komentoriviohjelma C-kielellä.

Ohjelman voi kääntää esimerkiksi Microsoftin cl-kääntäjällä assembly-kieliseksi. Tässä tapauksessa syntyy vain viisi riviä varsinaisia assembly-käskyjä, loput rivit liittyvät ympäristön ja main-metodin määrittelyyn (ks. kuva 33).

Toisin kuin disassemblerilla saadussa ohjelmalistauksessa, tässä symbolit ovat vielä nähtävissä ja lisäksi kääntäjä on kommentoinut mistä C-lähdekoodin riveistä mikäkin joukko assembly-käskyjä on syntynyt. Koska monet haittaohjelmat on kirjoitettu C-kielellä (Sikorski, Honig ja Lawler 2012, s. 83), on tarpeellista tietää miltä C-kielen ohjaus- ja tietorakenteet näyttävät assembly-muodossa. Näitä rakenteita on käsitelty esimerkiksi lähteessä Sikorski, Honig ja Lawler 2012, mutta laajemman kuvan saamiseksi omatoiminen kokeilu on suositeltavaa. Tähän tarkoitukseen yksinkertaiset komentoriviohjelmat ovat omiaan.

4.2.2 Standardi ikkunoitu ohjelma

Ohjelmat, jotka näyttävät ruudulla ikkunoita eivät yleensä sisällä main-funktiota, vaan vastaavan WinMain -funktion (Petzold 1998, osa 1, luku 1). Syynä tähän on se, että todelli-

suudessa ohjelman aloituskohta (entry point) on ohjelman ulkopuolella, käyttöjärjestelmän ytimessä ja ajonaikaisessa standardissa C-kirjastossa (MSVCRT) josta myös funktiot kuten printf ovat peräisin. Myös komentoriviohjelmien main-funktiota kutsutaan samaan tapaan kun prosessi ladataan muistiin. (Eilam 2011, s. 87, Richter 1999, Writing your First Windows Application)

Analyysissä voidaan yleensä kuitenkin aloittaa WinMain-funktiosta, jos sellainen löydetään. Se näyttää kuvan 10 mukaiselta.

```
int WINAPI WinMain(
   HINSTANCE   hInstance,
   HINSTANCE   hPrevInstance,
   LPSTR       lpCmdLine,
   int         nShowCmd)
{
   // ...
   return 0;
}
```

Kuvio 10. Ikkunoidun Windows-ohjelman alussa suoritettava WinMain-funktio.

`WINAPI` on makro, joka on määritelty Windowsin WINDEF.H -otsikkotiedostossa arvolla `__stdcall`. Kuten aiemmassa luvussa mainittiin, stdcall on kutsukäytäntö, jossa kutsuttava aliohjelma poistaa omat parametrinsa pinosta. Parametreista `hInstance` sisältää kahvan ohjelmainstanssiin, jota käytetään osassa API-kutsuja. `lpCmdLine` on 32-bittinen osoitin (LP = long pointer) merkkijonoon, joka sisältää komentorivin argumentteineen jolla ohjelma käynnistettiin. nShowCmd on vakio, joka kertoo ohjelmalle millaisena sen ikkuna halutaan ruudulle. `hPrevInstance` -parametri on jäänne 16-bittisestä Windowsista eikä sitä suositella käytettäväksi. (Petzold 1998, osa 1, luku 1, Richter 1999, CreateProcess function)

Ikkunan luomiseksi Windows-ohjelmassa luodaan ja rekisteröidään ikkunaluokka (window class) `RegisterClass`-funktiolla ja itse ikkuna `CreateWindow` -funktioilla, ja näytetään se ruudulla `ShowWindow`-funktiolla. Ikkunaluokalle määritellään sitä luotaessa ns. ikkunaproseduuri, jonka tehtävänä on käsitellä ikkunaa koskevat viestit. Esimerkki ikkunaproseduurista on esitelty kuvassa 11. (Petzold 1998)

Ikkunaproseduuri (yllä) suoritetaan kun ikkuna jolle se on rekisteröity saa viestin. Ikkuna

```
LRESULT CALLBACK MainWndProc(
    HWND hwnd,        // ikkunan kahva
    UINT uMsg,        // viestin tunniste
    WPARAM wParam,    // viestin 1. parametri
    LPARAM lParam)    // viestin 2. parametri
{
   switch (uMsg)
   {
      case WM_CREATE:   // Ikkunan alustukset
         return 0;

      case WM_PAINT:    // Ikkunan piirto
         return 0;

      case WM_SIZE:     // Tapahtuu kun ikkunan kokoa on muutettu
         return 0;

      case WM_DESTROY:  // Tapahtuu kun ikkuna tuhotaan
         return 0;

      // Muiden viestien käsittely samaan tapaan

      default:
         // Käytetään oletusviestinkäsittelijää
         return DefWindowProc(hwnd, uMsg, wParam, lParam);
   }
   return 0;
}
```

Kuvio 11. Ikkunan viestit käsittelevä ikkunaproseduuri.

saa viestin kaikesta vuorovaikutuksesta käyttäjän kanssa sen koon muuttamisesta näppäimien painalluksiin. Ikkunaproseduurin parametreista `hwnd` on kahva ikkunaan ja `message` kertoo viestin tyypin. `wParam` ja `lParam` ovat viestin parametreja, jotka vaihtelevat viestityypistä toiseen. Täydellinen lista viestityypeistä ja niiden parametreista löytyy Microsoftin MSDN-sivustolta [5].

Ikkunaproseduurien kutsu ja viestien vastaanotto tapahtuvat niinsanotussa viestisilmukassa (eng. message loop), joka käy läpi viestisilmukkaa (eng. message queue) sitä mukaa kun uusia viestejä saapuu. Viestisilmukka kirjoitetaan yleensä aivan WinMain-metodin loppuun, tai ainakin ennen ohjelman viimeistelyrutiineita. Silmukka näyttää kuvan 12 mukaiselta.

5. http://msdn.microsoft.com/en-us/library/windows/desktop/ms632586(v=vs.85).aspx

```
while ( GetMessage(&msg, NULL, 0, 0) )
{
   TranslateMessage( &msg );
   DispatchMessage( &msg );
}
```

Kuvio 12. Windows-ohjelman viestisilmukka.

Parametri `msg` on määritelty funktion alussa, ja on tietorakenne (struct) tyyppiä `MSG`.

4.2.3 Ei-standardi aloituskohta

Ohjelman aloituskohta voidaan määritellä myös suoraan ohjelman sisälle, ohittaen perinteiset kernel32- ja crt-kirjastojen aloitusrutiinit. Tämä tekniikka vähentää linkitettävien kirjastojen määrää ja nopeuttaa hieman ohjelman lataamista (“Stack Overflow: Compile a C++ program with only dependency on kernel32.dll and user32.dll” 2014). Ohjelma voidaan kääntää joko komentorivi- tai ikkunoiduksi ohjelmaksi. Luonnollisesti se vähentää myös toiminnallisuutta kuten komentoriviparametrien välitystä ja automaattista C-olioiden rakentajien kutsumista (“Stack Overflow: What functions does WinMainCRTStartup perform” 2014), mutta matalan tason ohjelmoinnissa (assembly, C) se ei välttämättä haittaa merkittävästi. Haittaohjelmien kirjoittajille pakollisten linkitettävien kirjastojen vähentäminen tarkoittaa pienempää, ja siten huomaamattomampaa ja helpommin siirrettävää ohjelmatiedostoa. Tarvittavat kirjastot on joka tapauksessa mahdollista ladata myös ajonaikaisesti, minkä lisäetuna on niiden näkymättömyys import-tauluissa (ks. luku 4.1.3.)

4.2.4 Dynaamisesti linkitettävät kirjastot eli DLL:t

Dynaamisesti linkitettävät kirjastot tulevat vastaan monessa muodossa haittaohjelmia tarkasteltaessa. Sen lisäksi että haittaohjelma käyttää funktiokutsuja, joita Windows API ja muut kirjastot tarjoavat, DLL-tiedoston injektio ajossa olevaan prosessiin antaa haittaohjelmalle mahdollisuuden ajaa ohjelmakoodia tämän prosessin sisällä. Tätä tekniikkaa käytetään ohjelmakoodin kätkemiseen ja käyttöoikeuksien korottamiseen. (Sikorski, Honig ja Lawler 2012, s. 254, s. 278)

DLL-tiedostot muistuttavat muodoltaan ja rakenteeltaan EXE-tiedostoja: niillä on tyypillisten koodi- ja datasektioiden lisäksi import-sektio (muista kirjastoista tuodut funktiot) ja resurssisektio (kuville ja muulle datalle). Lisäksi, jos DLL:n ainoa tarkoitus ei ole sisältää resursseja, se sisältää ns. export-sektion. Siinä missä import-sektio kuvaa funktiot jotka ladataan muista DLL-moduuleista, export-sektio listaa muiden moduulien käyttöön julkaistavat funktiot ja niiden osoitteet. Funktio voidaan julkaista joko nimellä tai järjestysnumerolla (ordinal). Järjestysnumeroa voidaan käyttää jos ei haluta ulkopuolisten käyttävän kirjastoa, mutta nimen käyttäminen on huomattavasti yleisempää. (“Microsoft PE and COFF Specification” 2013; Richter 1999; Sikorski, Honig ja Lawler 2012)

Alustuksia ja viimeistelyitä varten DLL voi sisältää erityisen funktion `DllMain`, jota myös aloituskohdaksi (entry point) kutsutaan. Ohjelman aloituskohdasta poiketen DllMain-funktiota kutsutaan sen alustuksen lisäksi myös sen lopetusvaiheessa, toisin sanoen kun DLL ladataan muistiin tai se vapautetaan. Funktiota kutsutaan myös uusien säikeiden syntyessä ja tuhoutuessa kun DLL on ladattuna. Sen toteutus C-kielellä näyttää kuvan 13 mukaiselta. (Richter 1999, luku 20)

```
BOOL WINAPI DllMain(
   HINSTANCE   hinstDll,
   DWORD       fdwReason,
   PVOID       fImpLoad)
{
   switch (fdwReason)
   {
      case DLL_PROCESS_ATTACH:
         // DLL ladataan prosessin muistitilaan
         break;

      case DLL_THREAD_ATTACH:
         // Uusi säie luodaan, DLL ladattuna
         break;

      case DLL_THREAD_DETACH:
         // Säie lopettaa toimintansa, DLL ladattuna
         break;

      case DLL_PROCESS_DETACH:
         // DLL poistetaan prosessin muistitilasta
         break;
   }

   return TRUE;
}
```

Kuvio 13. Dynaamisesti linkitettävään kirjastoon kuuluva DllMain-funktio.

Kuten ylläolevasta ohjelmakoodista nähdään, `fdwReason` on parametri joka kertoo syyn funktion kutsulle. `hinstDll` vastaa `WinMain`-funktion `hInstance`-parametria ja `fImpLoad` kertoo, onko kirjasto ladattu implisiittisesti (import-taulua käyttäen) vai eksplisiittisesti (ajon aikana). Kirjaston alustuksen yhteydessä funktion palautusarvo TRUE kertoo että alustus onnistui, muissa tapauksessa palautusarvoa ei käytetä. (Richter 1999, luku 20)

4.2.5 Kernel-tilan ohjelmointi ja ajurit

Suurin osa Windowsissa ajettavista ohjelmista, Windowsin omat komponentit mukaanlukien, toimivat käyttäjätilassa (user mode, nonprivileged mode). Ydin- eli kernel-tilassa toimivat vain Windowsin kriittisimmät toiminnot sekä laiteajurit. Kernel-tilan ohjelmat voivat

käsitellä kaikkea muistia ja laitteita suoraan, mikä tekee niistä houkuttelevia kohteita haittaohjelmien kirjoittajille. Virustorjuntaohjelmat ja palomuurit sisältävät myös kernel-tilassa toimivia komponentteja, joten kernel-tilasta myös niiden toiminnan häiritseminen onnistuu paremmin. (Sikorski, Honig ja Lawler 2012, s. 158)

Kernel-tilan ohjelmointi eroaa huomattavasti käyttäjätilan ohjelmoinnista. Ehkä tärkeimpänä erona Windows API:n päällimmäisten kerrosten funktiot kirjastoista, kuten kernel32.dll ja user32.dll, eivät ole käytettävissä. Sen sijaan käytetään kernel-tilan kirjastoista, kuten ntdll.dll, löytyviä funktioita [6]. Myöskään kahvoille, jotka todellisuudessa ovat tunnistenumeroita kernel-tilan olioille (Hart 2010, s. 7), ei ole tarvetta, vaan vastaavia olioita käytetään suoraan ohjelmakoodista (Reeves 2010, s. 20).

Miten kernel-tilaan sitten päästään? Yksinkertaisin tapa siihen on oman laiteajurin kirjoittaminen. Nimestään huolimatta laiteajurin ei tarvitse liittyä lainkaan fyysiseen laitteeseen. Kernel-tilan ajurit kirjoitetaan tyypillisesti C-kielellä Microsoftin tarkoitukseen kehittämää WDF-ohjelmistorajapintaa käyttäen. Tässä tutkielmassa käsittelemme aihetta vain lyhyesti ja olemassaolevien ajurien analyysin kannalta, oman ajurin kirjoittamisesta kiinnostuneille erinomainen lähde on esimerkiksi Reeves 2010.

Ajurin alustava funktio on yleensä nimeltään DriverEntry, ja se ottaa parametrikseen osoittimet ajurin kernel-olioon ja sen polkuun rekisterissä (Dang, Gazet ja Bachaalany 2014, s. 147). Kernel-oliota käytetään rekisteröimään käsittelijöitä erilaisille I/O-kutsuille (Dang, Gazet ja Bachaalany 2014, s. 148). Yleisin haittaohjelmien käyttämä I/O-kutsu on `DeviceIoControl`, jota voidaan käyttää yleiseen tiedonvälitykseen ajurin ja käyttäjätilan ohjelman välillä (Sikorski, Honig ja Lawler 2012, s. 206). DriverEntry-kutsun jälkeen ajuri pysyy ladattuna muistissa ja vastaa I/O-kutsuihin, jotka sille on rekisteröity.

Ajurin rajapintana käyttäjätilaan toimivat laiteobjektit, joita ajuri voi luoda haluamansa määrän `IoCreateDevice`-kutsuilla (Dang, Gazet ja Bachaalany 2014, s. 149). Nämä objektit näkyvät käyttäjätilassa tiedostoina, joita voidaan käsitellä normaalien tiedostojen tapaan API-funktioiden CreateFile, ReadFile ja WriteFile avulla (Sikorski, Honig ja Lawler 2012, s. 206). Tämä on kuitenkin vain yksi tapa viestintään ajurin ja ohjelman välillä, muita vaihtoeh-

6. Lisätietoja ntdll-kirjaston sisältämistä funktioista löytyy sivustolta http://undocumented.ntinternals.net/

toja ovat esimerkiksi jaetut muistialueet ja keskeytykset (Dang, Gazet ja Bachaalany 2014, s. 150–151).

Staattinen analyysi ajurille ei juurikaan eroa muusta staattisesta analyysistä, mutta dynaamiseen analyysiin joutuu näkemään enemmän vaivaa. Koska ajuri on ladattu käyttöjärjestelmän ytimeen, myös debuggerin on toimittava siellä. Ja koska debuggauksen kohde on käyttöjärjestelmä itse, se tarkoittaa että koko käyttöjärjestelmän tila on pysäytettävä debuggauksen ajaksi. Tämä johtaa siihen, että on käytettävä erillistä kernel-debuggeria, jonka interaktiivista osaa ajetaan toiselta tietokoneelta. WinDbg on tähän käyttöön käytetyimpiä ohjelmia, ja esimerkiksi Sikorski, Honig ja Lawler 2012 opastaa sen käyttöönotossa. Nykyisin debugattavaa käyttöjärjestelmää ajetaan yleensä virtuaalikoneessa ja debuggeria virtuaalikoneen ulkopuolella, ja kommunikaatio niiden välillä tapahtuu virtuaalisen sarjaportin kautta. (Sikorski, Honig ja Lawler 2012, s. 207)

4.2.6 Palvelut (Windows Services)

Windows NT toi mukanaan palvelut, jotka ovat Windowsin palvelunhallinnan (SCM, Service Contol Manager) alaisuudessa toimivia taustaprosesseja. Palvelimet ja vastaavat ohjelmat, jotka eivät vaadi käyttäjän syötettä on usein toteutettu palveluina. Haittaohjelman toteuttaminen palveluna antaa sen kirjoittajalle mahdollisuuden paitsi käynnistää se huomaamatta järjestelmän käynnistyksen yhteydessä, mutta myös käynnistää se korkeammilla järjestelmän (SYSTEM) oikeuksilla. SYSTEM-oikeudet antavat haittaohjelmaprosessille joissain tapauksissa jopa korkeammat valtuudet kuin järjestelmänvalvojakäyttäjälle. (Sikorski, Honig ja Lawler 2012, s. 152)

Lista Windowsin palveluista on nähtävillä järjestelmänvalvontakonsolin (MMC) moduulista services.msc 35, johon pääsee joko suoraan kirjoittamalla moduulin nimen komentoriville tai Oman tietokoneen (My Computer) kuvakkeen kontekstivalikon kohdasta Hallitse (Manage). Lista ei kuitenkaan ole täydellinen, vaan osa etenkin ajureista ja kriittisemmistä palveluista on piilotettu, ja usein myös haittaohjelmat piilottavat itsensä tästä listasta. Suoremmin palveluihin pääsee käsiksi komentorivityökalun sc.exe avulla tai suoraan rekisteristä HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services -avaimen kautta. Kak-

si viimeistä tapaa näyttävät myös palvelun tyypin ja tiedoston, ja antavat mahdollisuuden poistaa ja luoda uusia palveluita.

Palvelut voivat olla joko tavallisia exe-tiedostoja (tyyppiä WIN32_OWN_PROCESS), tai erityisesti palveluiksi kirjoitettuja dll-tiedostoja (tyyppiä WIN32_SHARE_PROCESS), jotka suoritetaan svchost-prosessin alla. Jälkimmäinen sopii haittaohjelmien kirjoittajille, sillä se piilottaa samalla prosessin tehtävänhallinnasta. Tyyppiä WIN32_KERNEL_DRIVER olevat palvelut ovat laiteajureita, joista on kerrottu tarkemmin edellisessä alaluvussa. (Sikorski, Honig ja Lawler 2012, s. 153)

Palveluksi kirjoitettu dll-tiedosto sisältää main-funktion sijasta ServiceMain-funktion, joka on muuten main-funktion kaltainen (parametrit argc ja argv) mutta se ei palauta mitään ja se noudattaa stdcall-kutsukäytäntöä (Hart 2010, s. 455). Palvelu sisältää myös ServiceCtrlHandler-funktion, jota kutsutaan sopivalla arvolla kun palvelu halutaan käynnistää tai pysäyttää (Hart 2010, s. 460).

Uuden palvelun rekisteröinti tapahtuu Advapi-kirjaston CreateService-funktiolla. Haittaohjelmien tapauksessa suoraan rekisteriin kirjoittaminen on myös vaihtoehto, jos viitettä CreateService-funktioon halutaan välttää. Palvelu saadaan käyntiin StartService-funktiolla. Funktioiden kutsuminen vaatii järjestelmänvalvojan oikeudet. (hart Sikorski, Honig ja Lawler 2012, s. 468)

4.3 Tärkeitä Windows API -kutsuja ja menetelmiä

Windows API tarjoaa lukemattoman määrän valmiita funktioita kaikkiin yleisemmin käytettyihin operaatioihin, kuten tiedostojen käsittelyyn ja verkkoyhteyksien hallintaan. Tässä luvussa on esitelty tärkeimpiä näistä funktioista ja haittaohjelmissa usein käytettyjä tekniikoita, jotka hyödyntävät niitä.

API-funktiot on ryhmitelty useisiin DLL-kirjastoihin, joista useimmat sijaitsevat Windowsin system32-hakemistossa. Taulukossa 3 on lueteltu tärkeimpiä näistä kirjastoista.

Kirjasto	Sisältää
Advapi32.dll	Rekisterioperaatiot, palvelut (Windows Services)
Gdi32.dll	Grafiikkakirjasto pikseleiden, kaarien ja alueiden piirtämiseen
Kernel32.dll	Muisti-, tiedosto- ja laite-IO-funktioita
User32.dll	Käyttöliittymäkomponentit
Ntdll.dll	Käyttäjätilassa toimiva kirjasto joka välittää tietoa kernel32.dll:n ja ntoskrnl.exen (kernel-tilassa) välillä. Tarjoaa (enimmäkseen dokumentoimattomia) operaatioita esimerkiksi prosessien ja tiedostojen suorempaan käsittelyyn.
Wininet.dll	Sovelluskerroksen verkkoprotokollafunktioita (FTP, HTTP, NTP)
WSock32.dll, Ws2_32.dll	Verkko- ja kuljetuskerroksen protokollafunktioita (Windows Sockets)

Taulukko 3. Microsoft Windowsin ydinkirjastoja. Lähteet: Sikorski, Honig ja Lawler 2012, s. 17, 159, Petzold 1998, luku 5

4.3.1 Tiedostot ja IO

Käyttäjätilassa kernel32.dll tarjoaa tärkeimmät kutsut tiedostojen ja hakemistojen käsittelyyn. Kernel-tilassa vastaavat operaatiot on toteutettu ntdll-kirjastossa ”Nt”-etuliitteellä, lisää tietoa niistä on saatavilla sivustolta http://undocumented.ntinternals.net/.

Tiedostojen avaamiseen ja luomiseen käytetään useimmiten funktiota `CreateFile` ja niiden lukuun ja kirjoitukseen funktioita `ReadFile` ja `WriteFile`. Toinen vaihtoehtoinen tapa on käyttää funktioita `CreateFileMapping` ja `MapViewOfFile` lataamaan tiedostot muistiin niin, että varattuun muistialueeseen kirjoittaminen vastaa suoraan tiedostoon kirjoittamista. Kaikki nämä funktiot löytyvät kernel32-kirjastosta. (Sikorski, Honig ja Lawler 2012, s. 137)

Tavallisten tiedostojen lisäksi IO-funktioita voidaan käyttää käyttöjärjestelmän luomien erikoistiedostojen käsittelyyn. Merkeillä `\..\` ja `\Device` alkavat tiedostot viittaavat laiteolioihin, joiden kautta päästään esimerkiksi lukemaan ja kirjoittamaan suoraan muistiin (\Device\PhysicalMemory) tai levylle (\Device \PhysicalDiskX). Vastaavasti muotoa `\\Tie-`

tokone\Kansio\Tiedosto olevat tiedostot ovat lähiverkossa olevia jaettuja tiedostoja. (Sikorski, Honig ja Lawler 2012, s. 138)

NTFS-tiedostojärjestelmä tarjoaa myös tavan tallentaa tiedostoon näkymättömiä ns. vaihtoehtoisia tietovirtoja (ADS, Alternate Data Streams). Tekniikan alkuperäinen tarkoitus lienee ollut lisätä tiedostoon metadataa, mutta haittaohjelmien kirjoittajille se on tehokas tapa datan piilottamiseen käyttäjältä. Vaihtoehtoinen virta tiedostoon voidaan avata lisäämällä sen nimen perään :Stream:$DATA, missä ”Stream” on virran nimi. (Sikorski, Honig ja Lawler 2012, s. 139)

Tiedostojen etsimiseen ja listaukseen voidaan käyttää funktioita FindFirstFile ja FindNextFile, tai niiden uudempia Ex-päätteisiä versioita. FindFirstFile ottaa parametreina hakumerkkijonon (esimerkiksi ”%USERPROFILE%\Documents*.doc”) ja osoittimen WIN32_FIND_DATA-tyyppiseen struktuuriin, johon löydetyn tiedoston tarkemmat tiedot kirjoitetaan. Funktio palauttaa kahvan, joka voidaan antaa FindNextFile-funktiolle etsimään lisää tiedostoja samoilla kriteereillä. Haun lopuksi kahva suljetaan FindClose-funktiolla. Esimerkiksi vakoiluohjelma saattaa käyttää näitä funktioita etsimään mielenkiintoisia tiedostoja. (“Microsoft Developer Network: Windows Dev Center” 2014)

4.3.2 Rekisterioperaatiot

Rekisterin käsittelyyn tarvittavat funktiot löytyvät Advapi-kirjastosta Reg-etuliitteellä. Rekisteriavain avataan RegOpenKey- tai RegOpenKeyEx-funktiolla, jonka jälkeen avatusta avaimesta voidaan lukea arvoja RegGetValue-funktiolla ja asettaa niitä RegSetValue- ja RegSetValueEx-funktioilla (Sikorski, Honig ja Lawler 2012, s. 141) RegCreateKey(Ex) luo uuden rekisteriavaimen ja funktioita RegEnumKey(Ex) ja RegEnumValue(Ex) voidaan käyttää luettelemaan kaikki avaimet tai arvot tietystä rekisteripolusta (Hart 2010, s. 89).

Rekisteri on jaoteltu juuriavaimiin (root key) ja niiden alta löytyviin avaimiin. Yleisimmin käytettyjä avaimia ovat HKEY_LOCAL_MACHINE ja HKEY_CURRENT_USER, joista ensimmäinen sisältää koko järjestelmää ja jälkimmäinen kirjautunutta käyttäjää koskevia tietoja. Juuriavain HKEY_CURRENT_CONFIG sisältää järjestelmän konfiguraatiotieto-

ja ja HKEY_USERS kaikkien järjestelmän käyttäjien HKEY_CURRENT_USER -avaimet. HKEY_CLASSES_ROOT sisältää tiedostojen ja niitä käsittelevien ohjelmien välisiin kytköksiin liittyviä tietoja. Rekisterifunktiot, kuten RegOpenKeyEx, ottavat juuriavaimen parametrina HKEY-tyyppisenä enumeraatioarvona. HKEY-arvot ja niiden numeeriset arvot on nähtävissä sekä MSDN-sivustolta verkosta että suoraan header-tiedostosta WinReg.h. (Hart 2010, s. 88)

4.3.3 Haitallisen ohjelmakoodin ajaminen

Yksinkertaisin tapa käynnistää ohjelma Windowsissa on käyttää funktioita `ShellExecute` (shell32.dll), `ShellExecuteEx` (shell32.dll) tai `CreateProcess` (kernel32.dll). Niiden lisäksi haittaohjelmat voivat käyttää myös injektiota, jolla olemassa oleviin prosesseihin syötetään haitallista ohjelmakoodia. Näin haittaohjelmaa voidaan ajaa huomaamattomasti, ja mahdollisesti myös korkeammilla käyttöoikeuksilla.

Aivan ensimmäisenä haittaohjelma tarvitsee kahvan kohdeprosessiin. Kaikki järjestelmän prosessit voidaan listata joko kernel32-kirjaston `CreateToolhelp32Snapshot`-funktiolla tai funktioilla `Process32First` ja `Process32Next` (Sikorski, Honig ja Lawler 2012). Nämä funktiot antavat kullekin prosessille tunnisteen (PID), joka voidaan muuttaa kahvaksi avaamalla prosessi `OpenProcess`-funktiolla. Tunniste saadaan myös kiertotietä esimerkiksi etsimällä ikkunan kahva `FindWindow`- tai `EnumWindows`-funktiolla, ja käyttämällä funktioita `GetWindowThreadProcessId`.

Prosessin kahvaa käyttämällä voidaan varata lisää muistia sen osoiteavaruudesta `VirtualAllocEx`-funktiolla. Varattuun muistialueeseen saadaan kirjoitettua uutta ohjelmakoodia `WriteProcessMemory`-funktiolla ja se saadaan suoritettua `CreateRemoteThread`-kutsulla. Yleensä suoritettava ohjelmakoodi on `LoadLibrary`-funktiokutsu haittaohjelman dll-tiedostoon, jonka DllMain-funktiosta varsinaisen haittaohjelmakoodin suoritus alkaa. Jos haittaohjelma on kirjoitettu erityisesti tätä tarkoitusta varten assembly-kielellä (hyvin harvinaista), DLL-injektiovaihe voidaan jopa kokonaan ohittaa ja kirjoittaa haittaohjelmakoodi suoraan prosessin muistiin ja ajaa sitä sen säikeenä. (Sikorski, Honig ja Lawler 2012, s. 257)

DLL-injektioon on myös muita tapoja. Yksinkertainen, mutta toimiva tapa on korvata olemassaoleva dll-tiedosto omalla vastaavalla, haittaohjelmakoodia sisältävällä kirjastolla. Tätä tapaa voidaan käyttää myös kirjastokutsujen vakoiluun ja prosessin piilottamiseen käyttäjältä tai sen lopettamisen estämiseen. Toinen tapa on kirjoittaa rekisteriavaimeen `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs`, joka sisältää listan kirjastoista, jotka ladataan jokaisen prosessin muistiavaruuteen. Tämä on myös pätevä tapa varmistaa, että haittaohjelma käynnistyy myös seuraavalla tietokoneen käynnistyskerralla. (Richter 1999, luku 22)

Vaihtoehtoinen tapa naamioida prosessi toiseksi on käynnistää toinen ohjelma ja kirjoittaa haittaohjelma sen päälle. Prosessi, joksi haittaohjelma halutaan naamioida, käynnistetään normaalisti `CreateProcess`-kutsulla, mutta sille annetaan parametri `CREATE_SUSPEND` kertomaan, että se käynnistetään pysäytettynä. Tyypillisesti tämän jälkeen vapautetaan prosessin ohjelmakoodin sisältävä muistialue `ZwUnmapViewOfSection`-funktiokutsulla ja varataan uusi muistialue `VirtualAllocEx`-funktiolla ja kirjoitetaan haittaohjelmakoodi siihen `WriteProcessMemory`-funktiolla. Oikea aloituskohta saadaan asetettua `SetThreadContext`-kutsulla ja lopuksi kutsutaan `ResumeThread`-funktiota ohjelman käynnistämiseksi. Prosessi näkyy nyt tehtävänhallinnassa valenimellä, ja jopa sen käynnistyspolku Process Hackerin kaltaisissa ohjelmissa näkyy vääränä ja kuten injektionkin kanssa, prosessi pääsee myös verkkoon palomuurin estämättä. (Sikorski, Honig ja Lawler 2012, 257–259)

4.3.4 Pysyvyyden varmistaminen eli persistenssi

Oleellinen osa mitä tahansa haittaohjelmaa on varmistaa, että se pysyy käynnissä vielä seuraavallakin järjestelmän käynnistyskerralla. Kuten muissakin tämän luvun tekniikoissa, menetelmät vaihtelevat triviaalista hyvinkin mielenkiintoisiin ja mielikuvituksellisiin.

Yleisin tapa käynnistää ohjelma järjestelmän käynnistyksen yhteydessä on lisätä sille rekisteriarvo avaimen `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run` alle. Myös juuriavain HKEY_CURRENT_USER sisältää vastaavan avaimen käyttäjäkohtaisena. Muita vastaavia rekisteriavaimia ovat (lähde: “Understand and Cont-

rol Startup Apps with the System Configuration Utility" 2009):

- HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Load
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\RunServices
- HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\RunServices
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Policies\Explorer\Run
- HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Policies\Explorer\Run
- HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager

Näiden lisäksi ohjelma voi tehdä itsestään palvelun luvun 4.2.6. mukaisesti joko sellaisenaan tai erityisenä DLL:nä, joka ajetaan svchost.exe -järjestelmäprosessin sisällä. DLL-muotoinen haittaohjelma voidaan myös lisätä rekisteriavaimen HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows alta löytyvään AppInit_DLLs-arvoon, jolloin se ladataan jokaiseen käynnistyvään prosessiin. Tätä käynnistystapaa käyttävät haittaohjelmat usein tarkistavat DllMain-funktiossaan, mihin prosessiin kirjasto on liitetty ja muuttavat toimintaansa sen mukaan. (Sikorski, Honig ja Lawler 2012, s. 241–242)

Haittaohjelma voi myös hyödyntää dll-tiedostojen latausjärjestystä. Kun ohjelma tarvitsee tietynnimistä kirjastoa, se etsii sitä seuraavassa järjestyksessä (lähde: Sikorski, Honig ja Law-

ler 2012, s. 245):

1. KnownDLLs-polku (Windows XP ja uudemmat): turvamekanismi kriittisimmille Windowsin kirjastoille
2. Ohjelman käynnistyshakemisto
3. Nykyinen työhakemisto
4. Järjestelmähakemisto (Windows\System32)
5. 16-bittinen järjestelmähakemisto (Windows\System)
6. Windows-hakemisto
7. Polussa (PATH-ympäristömuuttuja) luetellut hakemistot

KnownDLLs on Windows XP:n mukanaan tuoma mekanismi kriittisimmille käyttöjärjestelmän kirjastoille, joka nopeuttaa niiden lataamista ja estää haittaohjelmia asettamasta itseään korkeammalle latausjärjestyksessä. Kirjastot, jotka eivät sijaitse System32-hakemistossa ja joita ei ole merkitty KnownDLLs-listaan voidaan kuitenkin ohittaa laittamalla oma vastaava kirjasto System32-hakemistoon. Korvaavan kirjaston tulee toteuttaa kaikki alkuperäisen funktion tarjoamat funktiot samoilla parametreilla, palautusarvolla ja kutsukäytännöllä. (Sikorski, Honig ja Lawler 2012, s. 244-245)

Haittaohjelma saattaa myös kirjoittaa omaa koodiaan suoraan järjestelmän DLL- tai EXE-tiedostoon. Tavallinen tapa on korvata ohjelman ensimmäinen käsky hyppykäskyllä tai kutsulla troijalaisaliohjelmaan, ladata siellä dll-tiedosto `LoadLibrary`-kutsulla ja palata alkuperäiseen aloituskohtaan. Rekisterien sisältö tallennetaan yleensä ennen kutsua pusha- tai pushad-käskyllä ja palautetaan käskyllä popa tai popad ennen kuin suoritus siirretään takaisin alkuperäiseen aloituskohtaan. (Sikorski, Honig ja Lawler 2012, s. 243-244)

4.3.5 Käyttäjältä piiloutuminen ja käyttäjän toiminnan vakoilu

Haittaohjelmia, jotka piiloutuvat käyttäjältä ja/tai käyttöjärjestelmältä, sanotaan rootkiteiksi. Haittaohjelma voi sisältää rootkit-toiminnallisuutta, tai modulaarisen haittaohjelman tapauksessa se voi olla oma moduulinsa. Tehokkaimmat rootkitit toimivat omina ajureinaan kernel-tilassa, mutta myös käyttäjätilassa haittaohjelma voi piilottaa toiminnallisuuttaan tai estää sen poistamista.

Piilottaakseen tiedostoja tai prosesseja, tai estääkseen niiden poistamista tai lopettamista, haittaohjelman on päästävä kiinni käyttöjärjestelmän API-kutsuihin. Tämän luvun persistenssitekniikoissa esitelty ohjelmakoodin korvaaminen suoraan dll-tiedostoon on toimiva tapa, mutta yleensä haittaohjelmat käyttävät tähän tarkoitukseen API-koukkuja.

API-koukku voidaan asentaa joko kohteena olevan prosessin import-funktiotauluun (IAT hooking) tai suoraan sen lataamaan kirjastoon (inline hooking). Import-tauluun asetettava ”koukku” korvaa kutsuttavan funktion (esimerkiksi `TerminateProcess`) osoitteen haittaohjelmamoduulissa sijaitsevan funktion osoitteella, joka kutsuu alkuperäistä funktiota, mutta esimerkiksi sillä ehdolla, että lopetettava prosessi ei ole haittaohjelman prosessi. Esimerkki IAT-koukun asentamisesta on nähtävillä kuvassa 34. (Sikorski, Honig ja Lawler 2012, s. 248)

Import-osoitekoukun sijasta voidaan käyttää myös vaikeammin tunnistettavaa inline-koukkua, jonka kohteena on muistiin ladattu DLL-funktio. Koukun kohteena on DLL-tiedosto, jonka muistiin lataamista haittaohjelman tulee odottaa. Kun kohteena oleva kirjasto on ladattu muistiin, haittaohjelma etsii siitä kohteena olevan funktion osoitteen ja kirjoittaa siihen hypyn korvaavan rootkit-funktion alkuun. Rootkit-funktio voidaan injektoida kirjastoon varaamalla sille ensin tilaa `GlobalAllocEx`-funktiolla ja kirjoittamalla se `WriteProcessMemory`-funktiolla. Jos alkuperäinen funktio päätetään suorittaa, rootkit-funktio voi suorittaa hyppykäskyn korvaamat käskyt ja hypätä takaisin alkuperäiseen funktioon. (Sikorski, Honig ja Lawler 2012, s. 248)

Käyttäjän vakoiluun käytetyt tekniikat usein vaativat myös pääsyä käyttäjän ja käyttöjärjestelmän välillä kulkevaan tietoon, joten siihen käytetyt tekniikat ovat pitkälti samoja. Näppäimistön kuuntelijat eli keyloggerit ovat perinteisimpiä vakoiluohjelmia ja niitä käytetään yhä käyttäjätunnusten, henkilötietojen ja pankkitunnusten kaappaamiseen. Kuuntelija voidaan asentaa joko kernel-tilaan laiteajurina tai käyttäjätilaan tavallisena ohjelmana. (Sikorski, Honig ja Lawler 2012, s. 238)

Keylogger-haittaohjelma (tai mikä tahansa vakoiluohjelma) voidaan toteuttaa joko tapahtumapohjaisena koukkujen avulla tai pollaamalla eli kyselemällä käyttöjärjestelmältä näppäinten tilaa tietyin aikavälein. Koukku voi olla joko API-koukku edellä esitellyillä menetelmillä,

tai SetWindowsHookEx-funktiolla asetettu ns. Windows-koukku. Funktio ottaa parametrikseen tapahtumatyypin (näppäimistötapahtumille WH_KEYBOARD tai WH_KEYBOARD_LL) ja osoittimen funktioon, jota kutsutaan tapahtuman yhteydessä. Windows-koukkujen käyttö rajoittuu ikkunoihin ja käyttöliittymään, joten toisin kuin API-koukkuja, niitä ei voida käyttää esimerkiksi prosessien tai tiedostojen piilottamiseen. (Sikorski, Honig ja Lawler 2012, s. 239, 260)

Näppäimistön tilaa pollaavat haittaohjelmat käyttävät tyypillisesti GetAsyncKeyState-funktiota kyselemään näppäinten tilaa yksitellen. Ohjelman täytyy myös tarkastaa erikoisnäppäinten kuten shift, caps lock ja num lock tila muuntaessaan näppäimiä merkeiksi, ja näiden merkkijonojen löytyminen ohjelmasta onkin todennäköinen merkki keylogger-toiminnallisuudesta. Usein ohjelma käyttää myös funktiota GetForegroundWindow aktiivisen ikkunan tunnistamiseen. (Sikorski, Honig ja Lawler 2012, s. 239)

4.3.6 Käyttöoikeuksien korottaminen

Toisin kuin Unix-pohjaisissa järjestelmissä, Windowsissa on ollut tavallista pitää järjestelmänvalvojan oikeuksia tavallisella käyttäjällä ilman erillisiä kyselyitä. Tämä on helpottanut haittaohjelmien kehittäjien työtä, sillä erillistä käyttöoikeuksien korottamista (privilege escalation) ei ole tarvittu osana hyökkäystä. Jotkut operaatiot, kuten järjestelmäprosesseihin vaikuttaminen, vaativat kuitenkin vielä korkeammat järjestelmän (SYSTEM) valtuudet, jotka on mahdollista saada esimerkiksi käynnistämällä haitaohjelma palveluna (ks. luku 4.2.6) tai antamalla prosessille SeDebugPrivilege-käyttöoikeus. (Sikorski, Honig ja Lawler 2012, s. 246)

Windowsin käyttäjätilan objekteilla, kuten prosesseilla, on kullakin tunnisteolio (access token), joka kertoo sen käyttöoikeudet. Järjestelmänvalvoja voi avata olion OpenProcessToken-funktiolla ja muuttaa sen oikeuksia AdjustTokenPrivileges-funktiolla. AdjustTokenPrivileges-funktion tarvitsema lokaali arvo käyttöoikeudelle (kuten tässä tapauksessa SeDebugPrivilege) saadaan funktiolla LookupPrivilegeValue. Assembly-kielinen toteutus tästä tekniikasta on esitetty kuvassa 14 . (Sikorski, Honig ja Lawler 2012, s. 246–247)

```
00401003 lea eax, [esp+1Ch+TokenHandle]
00401006 push eax ; TokenHandle
00401007 push (TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY)  ; DesiredAccess
00401009 call ds:GetCurrentProcess
0040100F push eax ; ProcessHandle
00401010 call ds:OpenProcessToken
00401016 test eax, eax
00401018 jz short loc_401080
0040101A lea ecx, [esp+1Ch+Luid]
0040101E push ecx ; lpLuid
0040101F push offset Name ; "SeDebugPrivilege"
00401024 push 0 ; lpSystemName
00401026 call ds:LookupPrivilegeValueA
0040102C test eax, eax
0040102E jnz short loc_40103E
...
0040103E mov eax, [esp+1Ch+Luid.LowPart]
00401042 mov ecx, [esp+1Ch+Luid.HighPart]
00401046 push 0 ; ReturnLength
00401048 push 0 ; PreviousState
0040104A push 10h ; BufferLength
0040104C lea edx, [esp+28h+NewState]
00401050 push edx ; NewState
00401051 mov [esp+2Ch+NewState.Privileges.Luid.LowPt], eax
00401055 mov eax, [esp+2Ch+TokenHandle]
00401059 push 0 ; DisableAllPrivileges
0040105B push eax ; TokenHandle
0040105C mov [esp+34h+NewState.PrivilegeCount], 1
00401064 mov [esp+34h+NewState.Privileges.Luid.HighPt], ecx
00401068 mov [esp+34h+NewState.Privileges.Attributes], SE_PRIVILEGE_ENABLED
00401070 call ds:AdjustTokenPrivileges
```

Kuvio 14. SeDebugPrivilege-käyttöoikeuden asettaminen. Lähde: Sikorski, Honig ja Lawler 2012, s. 247

4.3.7 Verkkoliikenne

Matalan tason verkkokommunikaatio on toteutettu Windowsissa Berkeley-yhteensopivilla soketeilla, joiden käyttö vastaa pitkälti Unix-pohjaisten järjestelmien vastaavia kirjastoja. Soketteja voidaan käyttää lähettämään ja vastaanottamaan vapaamuotoisia TCP- ja UDP-paketteja. Sokettifunktiot löytyvät kirjastosta `ws2_32.dll` (Sikorski, Honig ja Lawler 2012, s. 143).

Sokettien alustamiseen käytetään ws2_32-kirjastosta löytyvää `WSAStartup`-funktiota. Kun rajapinta on alustettu, voidaan käyttää funktiota `socket` alustamaan soketti. Riippuen siitä, halutaanko toimia asiakkaana vai palvelimena, kutsutaan funktioita `connect` asiakkaalle, tai `bind`, `listen` ja `accept` palvelimelle. Datan lähetys ja vastaanotto tapahtuu funktioil-

la `send` ja `recv`, mutta sokettia voidaan käsitellä myös tiedostokahvana jos niin halutaan. Palvelimen `accept`-funktiolla hyväksymät yhteydet suljetaan `shutdown`-kutsulla ja lopuksi kun sokettia ei enää tarvita, se voidaan sulkea `closesocket`-funktiolla. (Hart 2010, s. 412–421)

Verkko-osoitteen esittämiseen käytetään `sockaddr`-rakennetyyppiä, jolle TCP/IP -protokollan tapauksessa käytetään alityyppiä `sockaddr_in`, jonka kentät on avattu taulukossa 4. `in_addr`-tyyppinen IP-osoite voidaan rakentaa merkkijonosta funktiota `inet_addr` käyttämällä. (Hart 2010, s. 415–416)

Kenttä	Tietotyyppi	Selitys
sin_family	short	Protokolla. Asetettava arvoksi AF_INET.
sin_port	u_short	Portti.
sin_addr	in_addr	IP-osoite.
sin_zero	char [8]	Ei käytetä.

Taulukko 4. sockaddr_in -tietotyypin kentät. Lähde: Hart 2010, s. 416

Sokettien lisäksi Windows Internet API (WinInet) tarjoaa valmiita funktioita HTTP- ja FTP-protokollille (Sikorski, Honig ja Lawler 2012, s. 145). Yhteys avataan `InternetOpen`-funktiolla, jonka jälkeen voidaan kutsua funktioita, kuten `InternetConnect`, `InternetOpenURL` ja `InternetReadFile`. Funktiota `InternetGetConnectedState` voidaan käyttää tarkastamaan verkkoyhteyden tila. Täydellinen lista API-funktioista on saatavilla Microsoftin MSDN-sivustolta [7]. Muita kiinnostavia Windowsin verkkokirjastoja ovat iphlpapi.dll, joka tarjoaa apufunktioita reititykseen sekä dnsapi.dll ja etenkin sen `DnsQuery`-funktio, jota voidaan käyttää yhdistämään verkkonimiä ip-osoitteisiin tai päinvastoin. (“Microsoft Developer Network: Windows Dev Center” 2014)

7. http://msdn.microsoft.com/en-us/library/windows/desktop/aa385483(v=vs.85).aspx

5 Haittaohjelmien syvällisempi analyysi

Tässä luvussa tutustumme tekniikoihin, jotka vaativat syvällisempää tuntemusta tietokonearkkitehtuurista, käyttöjärjestelmästä ja assembly-kielestä. Englanninkielisessä kirjallisuudessa näille tekniikoille on vakiintunut nimitys "reverse engineering"tai epämuodollisemmin "reversing", millä tarkoitetaan minkä tahansa valmiin tuotteen purkamista osiin ja sen toimintaperiaatteen perinpohjaista selvittämistä. Termille ei ole vakiintunutta suomenkielistä vastinetta. Reverse engineering on perinteisesti yhdistetty tuotteiden ja ohjelmistojen laittomaan kopiointiin eli piratismiin, mutta sille on myös lukuisia laillisia käyttökohteita tilanteissa joissa lähdekoodia tai alkuperäisiä suunnitteludokumentteja ei ole saatavilla. (Eilam 2011)

Laitteistoarkkitehtuuriksi on valittu tässä Intel x86 ja käyttöjärjestelmäksi Microsoft Windows niiden yleisyyden vuoksi (Sikorski, Honig ja Lawler 2012, s. 67), ja analyysin kannalta tärkeimmät perusteet niistä on esitelty edellisessä luvussa. Luvussa esiteltyjä tietoja ja tekniikoita voidaan soveltaa myös muissa ympäristöissä pienillä muutoksilla.

Sekä staattisessa että dynaamisessa analyysissä tällä tasolla assembly-kielen tuntemus on tärkeää, sillä sitä käytetään kuvaamaan ohjelman suoritusta ihmiselle epäintuitiivisen binäärimuodon sijaan. Yksittäiset assembly-käskyt keskeytyksiä ja kirjastokutsuja lukuunottamatta tekevät hyvin yksinkertaisia operaatioita kuten peruslaskutoimituksia ja siirtoja muistiosoitteista toiseen, mikä tekee ohjelmalistauksista hyvin pitkiä. Käsky kerrallaan lukemisen sijasta tärkeämpää on tutkia kirjastokutsuja ja suorituksen siirtymistä ohjelman osasta toiseen.

Staattisessa analyysissä ohjelma avataan disassembleriksi kutsuttuun ohjelmaan, joka näyttää ohjelman esityksen assembly-kielellä. Modernit disassemblerit kuten IDA Pro auttavat koodin tarkastelussa tunnistamalla siitä mahdollisia funktioita, merkkijonoja ja muita symboleita. Automatiikasta huolimatta haittaohjelman staattinen analyysi vaatii analysoijalta runsaasti aikaa ja vaivaa, minkä vuoksi se jää usein vähemmälle huomiolle, varsinkin jos sama tieto on saatavissa helpommin dynaamisella analyysillä.

Dynaaminen analyysi jatkaa siitä mihin staattinen päättyy: kun ohjelman rakenne on selvillä, se voidaan käynnistää (yleensä emulaattorissa tai virtuaalikoneessa) ja tutkia ohjelman

ja prosessorin tilaa suorituksen eri kohdissa. Tämä on erityisen hyödyllistä ohjelman kohdissa, joissa sen suorituksen haarautuminen riippuu ohjelman tilasta ja sen päätteleminen ohjelmakoodista vaatii työtä. Myös erilaisia pakkausmenetelmiä voidaan kiertää dynaamista koodianalyysiä käyttämällä. On kuitenkin pidettävä mielessä, että dynaamisessa analyysissä tutkitaan aina vain yhtä suorituskertaa ja eri suorituskertojen välillä ohjelman kulku voi vaihdella merkittävästikin. (Bergeron et al. 2001)

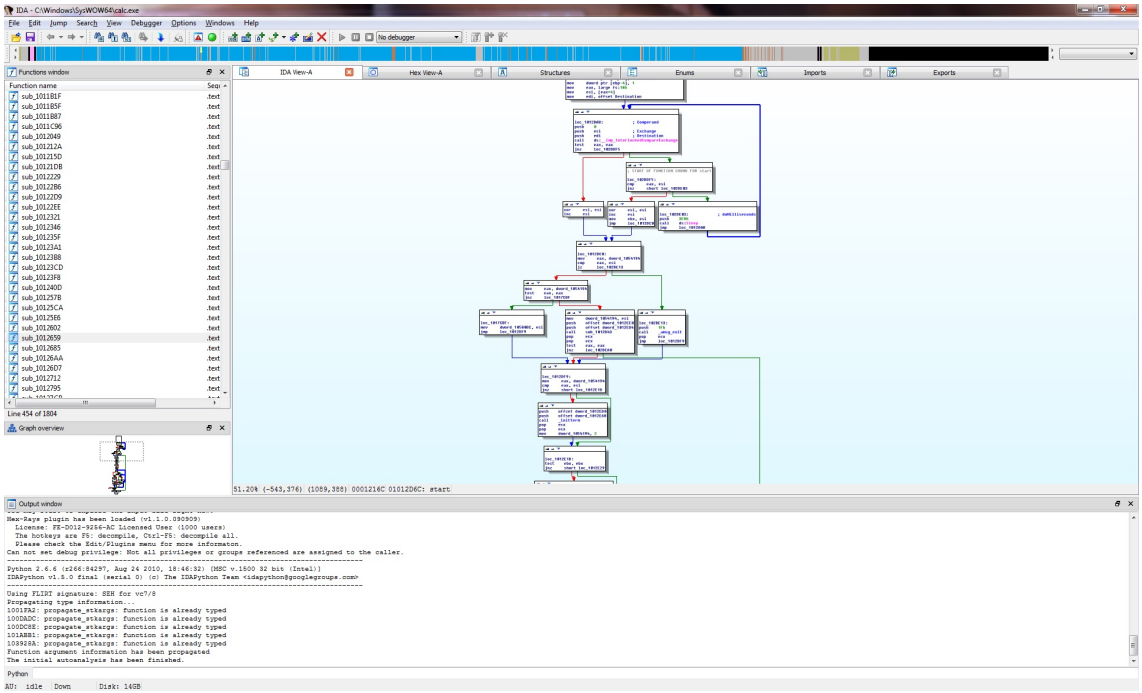
5.1 Staattinen lähestymistapa

Ohjelman staattinen analyysi kooditasolla alkaa sen konekielisen esityksen muuntamisesta ihmiselle luettavampaan muotoon, assembly-kielelle (Eilam 2011, s. 15). Tämä onnistuu disassembleriksi kutsutun ohjelman avulla. Yksinkertaisimmillaan disassembler voi olla komentoriviohjelma, joka ottaa parametrinaan ohjelmatiedoston ja tulostaa sille assembly-kielisen vastineen, mutta modernimmat ns. interaktiiviset disassemblerit kuten IDA Pro (kuva 15) tarjoavat analysoijalle lisäksi automaattisia työkaluja jotka auttavat koodin tutkimisessa ja ymmärtämisessä.

Ohjelma assembly-kielellä esitettynä voi viedä helposti kymmeniä tuhansia rivejä. Ideaalisinta olisi jos ohjelman voisi kääntää takaisin lähdekielelleen, mutta se on yleensä mahdollista vain välikielelle käännettyjen ohjelmien, kuten Java- ja .NET-ohjelmien tapauksessa. Konekielelle tai assembly-kielelle käännettynä informaatiota häviää niin paljon, että täydellinen käännös ei ole mahdollinen. Niinsanottuja decompiler-ohjelmia on kehitteillä, mutta mikään niistä ei ole vielä onnistunut tuottamaan analyysikelpoista koodia. (Eilam 2011, s. 129)

Assembly-kielistä ohjelmaa voidaan lähteä pilkkomaan jakamalla se ensin funktioihin (ks. luku 4.1.4) ja funktiot peruslohkoihin. Peruslohkoksi sanotaan jonoa peräkkäisiä käskyjä, jonka sisällä suoritus ei haaraudu (Allen 1970). Näin saadaan ohjelman tarkasteluun kaksi abstraktiotasoa, funktioiden välinen eli interproseduraalinen ja funktioiden sisäinen eli intraproseduraalinen (Bergeron et al. 2001).

Funktion sisällä peruslohkoista voidaan muodostaa ns. kontrollivuograafi, jossa suoritusta seurataan kaarilla peruslohkosta toiseen (Allen 1970; Bergeron et al. 2001). Näin funktiosta



Kuvio 15. Ruutukaappaus IDA Pro -disassemblerista. Pelkkien assembly-rivien sijaan ohjelman kulku on havainnollistettu ns. kontrollivuokaavion avulla.

voidaan jo silmämääräisesti tunnistaa esimerkiksi silmukoita ja muita rakenteita. Myös dataa voidaan abstrahoida tunnistamalla funktiosta pinomuuttujat (lokaalit muuttujat ja parametrit) ja esittämällä ne symbolisesti. IDA Pro on tällä hetkellä ainoa disassembleri, joka osaa tehdä kaiken tämän automaattisesti, mikä selittää sen suosion reverse engineering -piireissä (Sikorski, Honig ja Lawler 2012, s. 120).

Tyypillinen haittaohjelma voi sisältää helposti satoja funktioita. Osa niistä on pieniä apufunktioita, joita kutsutaan ympäri ohjelmaa, ja osaa niistä ei kutsuta kuin kerran tietystä kohtaa. Koko ohjelman funktioiden välisten suhteiden kuvaaminen graafisesti ei ole järkevää (kuten kuvasta 54 voi huomata), joten tällä tasolla analyysi täytyy tehdä käsin. Ohjelman aloituskohta on hyvä lähtöpiste analyysille, mistä voidaan rekursiivisesti seurata funktiokutsuja joko leveys- tai syvyyssuunnassa tai näitä vapaasti yhdistellen. Analyysiä voi lähestyä myös ”alhaalta ylös”, lähtien liikkeelle funktioista aloituskohdan sijaan. Vaikka kriittisimmät funktiot on yleensä toteutettu haittaohjelmissa ajonaikaisella linkityksellä, import-taulu voi silti paljastaa mielenkiintoisia funktioita, kuten ”RegOpenKeyEx” tai ”socket”. Funktiot ja niiden parametrit kannattaa nimetä sitä mukaa kun niiden merkitys selviää, sillä se hel-

pottaa niiden käytön hahmottamista myöhemmin. Kirjallisuudesta löytyy tähän vaiheeseen hyvin vähän apuja, jokainen löytää kokemuksen myötä oman tyylinsä.

Staattinen analyysi on aikaavievää, ja usein eteen voi tulla kuvan 55 kaltaisia funktioita. Käsky kerrallaan analysoimisen sijaan voidaan ottaa abstraktimpi näkökulma. Mitä parametreja funktio ottaa, ja mitä se palauttaa? Entä mitä funktioita se kutsuu ja missä järjestyksessä, ja millaisia suorituspolkuja näiden funktioiden läpi syntyy? Tarvittaessa, ja jos tilanne sen sallii, voidaan käyttää myös debuggausta apuna, ja kävellä funktio läpi askel askeleelta. Näin nähdään myös parametrien ja lokaalien muuttujien arvot pinosta. Kannattaa pitää mielessä, että kaikki haittaohjelmat, etenkään assembly-kielellä kirjoitetut, eivät aina noudata standardeja kutsukäytäntöjä, mutta vähintäänkin ulkoisten kirjastokutsujen kanssa niitä on käytettävä. Näihin tärkeimpinä kuuluu funktion palautusarvon palauttaminen eax-rekisterissä, parametrien välitys pinon kautta (pinokehykset) ja tiettyjen rekisterien (muut kuin eax, ecx ja edx) arvon säilyminen funktiokutsun läpi.

5.2 Dynaaminen lähestymistapa

Missä staattisessa analyysissä työkaluna käytettiin disassembleria, dynaamisessa analyysissä tärkein työkalu on virheenjäljitysohjelma eli debuggeri. Debuggeri voi olla joko disassemblerin yhteydessä (IDA Pro, w32dasm) tai erillisenä ohjelmana (OllyDbg, Immunity Debugger). Debuggerin kantavana ajatuksena on ohjelman suorituksen pysäyttäminen ja sen sisäisen tilan näyttäminen sen käyttäjälle. Ohjelmistokehityksessä käytetyt ns. korkean tason debuggerit käyttävät tilan kuvaamiseen lähdekoodia, mitä haittaohjelmien tapauksessa harvemmin on saatavilla. Valmiin ohjelmabinäärin debuggaukseen voidaan käyttää matalan tason debuggeria, joka näyttää suorituksessa olevan ohjelmakoodin assembly-kielellä ja ohjelman käsittelemän datan suoraan muistista ja prosessorin rekistereistä. Ohjelman tilaa voidaan myös muuttaa tarvittaessa ja saada se suorittamaan toimintoja joita se ei normaalitilanteessa suorittaisi. (Sikorski, Honig ja Lawler 2012, s. 167 – 168)

Kun debuggeri käynnistetään ja sille annetaan ohjelma, se lataa ohjelman muistiin ja käynnistää sen. Sen sijaan että ohjelma käynnistyisi normaalisti ja esimerkiksi näyttäisi ikkunan, suoritus pysähtyy ohjelman ensimmäiselle riville odottamaan käskyä debuggerilta. Tässä ti-

lassa ohjelman tila ei muutu, ja sitä voidaan tarkastella (ja muuttaa) debuggerissa.

Kun ohjelma on pysähdyksissä, sitä voidaan edistää joko käsky kerrallaan tai antamalla suorituksen jatkua vapaasti. Jälkimmäisessä tapauksessa voidaan asettaa keskeytyskohtia (breakpoint) jotka palauttavat ohjelman pysähtyneeseen tilaan, odottamaan käskyä debuggerilta. Matalan tason debuggauksessa keskeytyskohtia on kahdenlaisia: ohjelmallisia ja laitekeskeytyskohtia (software / hardware breakpoint). (Sikorski, Honig ja Lawler 2012, s. 171 – 173)

Ohjelmalliset keskeytyskohdat pysäyttävät suorituksen tietylle riville samaan tapaan kuin lähdekooditasolla debugattaessa. Debuggeri tekee tämän rekisteröimällä itselleen laitteistokeskeytyksen 3, joka on erityisesti tarkoitettu debuggaukseen. Jokaista keskeytyskohtaa varten debuggeri ylikirjoittaa siinä olevan käskyn keskeytyksen 3 laukaisevalla käskyllä INT 3 (operaatiokoodi 0x55). Näin suoritus siirtyy keskeytyskohtaan tultaessa debuggerille, joka lukee ohjelman tilan, kirjoittaa alkuperäisen käskyn takaisin ohjelmaan ja jää odottamaan käyttäjän syötettä. (Sikorski, Honig ja Lawler 2012, s. 173 – 174)

Laitekeskeytyskohdat puolestaan seuraavat suoraan muistiosoitteita laitetasolla. Tämän seurauksena suoritus pysähtyy myös dataoperaatioihin jotka kohdistuvat osoitteeseen. Prosessori pitää kirjaa keskeytyskohdista rekistereissä DR0-DR7, jotka sisältävät niiden osoitteet ja ehdon kullekin keskeytykselle. Ehto voi olla luku, kirjoitus tai suoritus. Toisin kuin ohjelmistokeskeytyskohdilla, rekisterien määrä rajoittaa laitekeskeytyskohtien määrän kuuteen kerrallaan. (Sikorski, Honig ja Lawler 2012, s. 174 – 175)

Jotkut debuggerit tukevat myös ehdollisia keskeytyskohtia. Ehtona voi olla esimerkiksi tietty arvo tietyssä rekisterissä tai muistiosoitteessa, keskeytyskohdan ohittaminen tietyn monta kertaa tai tietty käsky. Ehdolliset keskeytyskohdat toimivat sisäisesti kuten ehdottomatkin, mutta debuggeri ohittaa ne ilmoittamatta käyttäjälle, jos ehto ei täyty. Jatkuvat keskeytykset kuitenkin hidastavat suoritusta huomattavasti, joten niitä kannattaa välttää usein suoritettavissa funktioissa. (Sikorski, Honig ja Lawler 2012, s. 175)

6 Analyysia vaikeuttavia tekniikoita

Jotta haittaohjelma pysyisi mahdollisimman kauan liikenteessä, se on syytä suojata vastatoimilla analyysiä ja tunnistusta vastaan. Virustorjuntaohjelmien yleistyttyä pakkauksesta ja erilaisista polymorfisista tekniikoista on tullut haittaohjelmalle lähes välttämätön suojauskeino. Staattista analyysiä vaikeuttamaan on keksitty keinoja harhauttaa disassembleria monitulkintaisilla käskyillä, ja dynaamista analyysiä vastaan on kehitelty keinoja debuggerien ja virtuaalikoneiden tunnistamiseen ja kaatamiseen. Haittaohjelman analyysissä ensimmäinen vaihe onkin usein näiden keinojen kiertäminen, johon tämä luku pyrkii antamaan tarvittavat tiedot ja välineet.

6.1 Pakatut haittaohjelmat

Suurin osa moderneista haittaohjelmista on jollain tavalla pakattuja tai salattuja. Haitallisen ohjelmakoodin pakkaus paitsi pienentää sen kokoa, tekee myös sen sisältämistä merkkijonoista ja muista tuntomerkeistä tunnistamattomia pintapuoliselle tarkastelulle. Ovelammat ns. polymorfiset haittaohjelmat voivat myös harhauttaa virustorjuntaohjelmia pakkaamalla itsensä uudelleen, jolloin pakattu osa ohjelmaa muuttuu täysin. Vastatoimena useimmat virustorjuntaohjelmat osaavatkin purkaa yleisimmät pakkaukset, kuten UPX-pakkauksen läpinäkyvästi. Osa virustorjuntaohjelmista sisältää myös ns. hiekkalaatikon, jossa se käynnistää epäilyttävät, pakatulta näyttävät ohjelmat ja pyrkii vedostamaan muistista ohjelman puretun version sen käynnistyksen jälkeen. (Yan, Zhang ja Ansari 2008)

6.1.1 Pakatun haittaohjelman rakenne

Pakattu haittaohjelma voi sisältää myös muita analyysiä vaikeuttavia tekniikoita kuten obfuskaatiota, mutta sen perusrakenne ja -toiminnallisuus ovat samoja pakkausalgoritmista ja suoritusympäristöstä riippumatta. Suojattu osa ohjelmasta on tallennettu datana sen datasektioon, jonka ohjelman käynnistyvä purkajaosa purkaa ns. purkusilmukassa ja siirtää suorituksen puretulle ohjelmalle. Virustorjuntaohjelmien kehittäjät tuntevat myös hyvin tämän rakenteen, minkä vuoksi siihen on yleensä lisätty variaatiota vaikeuttamaan tunnistusta. Esi-

merkiksi hyppy purettuun ohjelman osaan on usein toteutettu jollain tavallista `jmp`-käskyä luovemmalla tavalla. (Yan, Zhang ja Ansari 2008)

Pakattu PE-ohjelma tarvitsee minimissään kolme sektiota:

- sektion, josta suoritus alkaa ja joka sisältää purkukoodin. Tämän sektion on oltava luettavissa ja suoritettavissa.
- sektion, joka sisältää pakatun ohjelmakoodin. Tämän sektion on oltava luettavissa.
- sektion, johon purettu ohjelmakoodi muistissa sijoitetaan. Tämän sektion on oltava kirjoitettavissa ja suoritettavissa.

Nämä sektiot voivat olla kaikki erillisiä, kaikki yhdessä tai sopiva kombinaatio jossa käyttöoikeudet osuvat yksiin. Yleensä ohjelmakoodin sisältävän `.text`-sektion ei tarvitse olla kirjoitettavissa, joten jos se on, se voi olla merkki pakkauksesta tai jonkinlaisesta polymorfismista.

6.1.2 Pakatun haittaohjelman tunnistaminen

Sektioiden lisäksi on myös muita tapoja tunnistaa pakattu ohjelma. Pakkaus itsessään ei ole merkki haittaohjelmasta – esimerkiksi asennusohjelmat ja pelit usein pakkaavat datansa mahtuakseen pienempään tilaan. Osa kaupallisista ohjelmistoista käyttää myös pakkausta ja muita analyysiä vaikeuttavia menetelmiä älyllisen omaisuutensa suojaamiseen (Yan, Zhang ja Ansari 2008). Silti pakkauksen automaattinen tunnistaminen auttaa antivirusohjelmaa seulomaan, mitkä ohjelmat se suorittaa hiekkalaatikkoympäristössä. Kaikkien ohjelmien suoritus hiekkalaatikossa ei kannata, sillä purku tai sen yritys voi kestää kymmenistä sekunneista minuutteihin (Perdisci, Lanzi ja Lee 2008). Käsin tehtävässä analyysissä pakkaus paljastuu nopeasti, mutta sen tunnistamiseen on myös systemaattisempia menetelmiä.

Staattinen analyysi paljastaa, että pakatut ohjelmatiedostot sisältävät yleensä kokoonsa nähden vähän merkkijonoja ja viitteitä ulkoisiin kirjastoihin. Koska purkaja lukee pakatun datan omasta muistiavaruudestaan, purkaa sen aritmeettis-loogisilla operaatioilla ja kirjoittaa edelleen omaan muistiavaruuteensa, se ei ole riippuvainen ulkoisista kirjastofunktioista. (Perdisci, Lanzi ja Lee 2008)

Eräs käytetyimpiä tapoja pakkauksen tunnistamiseen on tiedoston entropian mittaaminen. Entropia eli informaation tiheys on informaatioteoreettinen suure, joka on sitä pienempi, mitä säännönmukaisempaa mitattava data on. Näin esimerkiksi tekstillä tai jonolla konekielisiä käskyjä on huomattavasti pienempi entropia kuin (pseudo)satunnaisella datalla. (Lyda ja Hamrock 2007)

Koska pakkaus perustuu toistuvien bittijonojen tunnistamiseen ja koodaamiseen pienempään tilaan, se kasvattaa tiedoston entropiaa. PE-tiedostojen tapauksessa entropia voidaan laskea joko koko tiedostosta, tai erikseen jokaisesta sektiosta jolloin voidaan päätellä, mikä sektio sisältää pakatun datan. Tutkimusartikkelia Lyda ja Hamrock 2007 varten kehitetty Bintropy-ohjelma tekee juuri tämän, ja lisäksi laskee minimi-, maksimi- ja keskiarvot jokaisesta 512 tavun lohkosta. Ohjelmaa ei ole julkisesti saatavilla, mutta esimerkiksi John Walkerin Ent-työkalu [1] osaa myös laskea tiedoston entropian.

Muita ilmaisohjelmia pakkauksen tunnistamiseen ovat PEiD ja Mandiant Redline (ent. Red Curtain). PEiD käyttää sääntöpohjaista pakkauksen tunnistusta ja Redline entropiaan perustuvaa tunnistusta. Pakkauksen tunnistuksesta on tehty myös tutkimusta, joka on tuottanut työkaluja kuten PHAD (Choi et al. 2008), PE-Probe (Shafiq, Tabish ja Farooq 2009) ja Reminder (Han, Lee ja Lee 2009), jotka eivät kuitenkaan ole vapaasti saatavilla. Kaikki kolme työkalua käyttävät entropiaa ja sektioiden kirjoitusoikeuksia pakkauksen tunnistukseen. (O'Kane, Sezer ja McLaughlin 2011)

6.1.3 Pakkauksen purkaminen

Pakatussa muodossaan haittaohjelman staattinen analyysi on mahdotonta, minkä vuoksi pakkauksesta yleensä halutaan päästä eroon. Pakkauksen purkamiseen on kolme vaihtoehtoista tapaa. Jos pakkausalgoritmi on jokin ennestään tunnettu, se voidaan purkaa erityisesti sille tarkoitetulla purkuohjelmalla. Jos algoritmia ei tiedetä tai sille ei ole purkuohjelmaa, voidaan käyttää ns. geneeristä eli universaalia purkajaa. Geneerinen purkaja käynnistää ohjelman joko suoraan tai emuloidusti ja odottaa että ohjelma purkaa itsensä muistiin, jonka jälkeen se vedostaa puretun datan (oletetusti purettu haittaohjelma) levylle. Useat debuggerit, ku-

1. Ent: http://www.fourmilab.ch/random/

ten IDA Pro ja OllyDbg, sisältävät geneerisen purkukomponentin osana toiminnallisuuttaan. (Yan, Zhang ja Ansari 2008)

Jos ohjelma vastustaa geneeristä purkua, jää vaihtoehdoksi purkaminen käsin. Purkua voi lähestyä joko staattisesta tai dynaamisesta näkökulmasta – staattisessa vaihtoehdossa analysoidaan pakkausalgoritmi sen assembly-esityksen perusteella ja dynaamisessa ajetaan ohjelmakoodi ja annetaan sen purkaa itsensä samaan tapaan kuin geneerinen purkaja tekee. Jälkimmäinen tapa on yleensä vähemmän aikaa työtä vaativa, mutta ohjelma voi sisältää debuggausta vastustavia tekniikoita jotka on ensin poistettava staattisella analyysillä. Erityisen vahvasti suojatut haittaohjelmat voivat vaatia pakkauksen purkamista osissa ja kaikkia sen osia ei välttämättä saakaan purettua esimerkiksi ajan puutteen vuoksi, mutta pahimmassa tapauksessa myös osittain purettua ohjelmatiedostoa voidaan tutkia staattisesti esimerkiksi IDA Pro:lla vaikka sitä ei saisikaan ajettua. (Sikorski, Honig ja Lawler 2012, s. 416–401)

6.1.4 Poly- ja metamorfismi

Polymorfismi (kreik. *poly* + *morphē* + ismi, monimuotoisuus) on suojautumiskeino, jolla haittaohjelma muuttaa omaa ohjelmatiedostoaan välttääkseen tunnistuksen ja vaikeuttaakseen analyysiä. Kun polymorfinen haittaohjelma suoritetaan tai se kopioi itsensä eteenpäin, se generoi itselleen uuden, alkuperäisestä poikkeavan ohjelmatiedoston. Polymorfiset haittaohjelmat voivat käyttää joko pakkausta tai salausta, joista jälkimmäinen on helpompi toteuttaa sillä salausavaimen vaihtaminen riittää muuttamaan salattu ohjelman osa tunnistamattomaksi. (You ja Yim 2010)

Pakkaus tai salaus eivät sinällään täysin suojaa tunnistukselta, sillä ohjelman purkuosa jää edelleen samanlaiseksi ja sen perusteella voidaan tehdä tunnistussääntö virustorjuntaohjelmaa varten. Tätä vastaan kamppaillakseen haittaohjelmien kehittäjät muuttavat myös pakkauksen purkavaa osaa osana muunnosta. Tällöin haittaohjelmasta tulee metamorfinen, itseään muunteleva. Toisin kuin polymorfismi, metamorfismi ei välttämättä vaadi pakkausta tai salausta tuekseen. (You ja Yim 2010)

Tapoja muuttaa ohjelmakoodin esitystä muuttamatta sen toiminnallisuutta on monia: koodin joukkoon voidaan lisätä tarpeettomia käskyjä jotka eivät tee mitään, käytettäviä rekistereitä

voidaan vaihtaa keskenään, toisistaan riippumattomia käskyjä suorittaa eri järjestyksessä ja aliohjelmien paikkaa tiedoston sisällä muuttaa (You ja Yim 2010). Jotkut metamorfiset haittaohjelmat voivat myös sisältää oman lähdekoodinsa, josta ne kääntävät muuntavalla kääntäjällä itselleen uuden ohjelmatiedoston (esim. Win32/Apparition, O'Kane, Sezer ja McLaughlin 2011). Näin on mahdollista saada aikaan tuhansia täysin erilaisia variantteja samasta ohjelmasta, mikä tekee perinteisen syntaktisen tunnistuksen käytännössä mahdottomaksi.

6.2 Assembly-koodin obfuskaatio staattista analyysiä vastaan

Koska x86-arkkitehtuurissa konekieliset käskyt vaihtelevat pituudeltaan ja käskyjen joukossa voi olla myös dataa, muunnos assembly-kielelle ei ole triviaali operaatio. Poikkeukselliset käsky-data -yhdistelmät voivat sekoittaa disassemblerin synkronoinnin ja saada sen tuottamaan virheellisiä käskyjä. Haittaohjelmien tekijät käyttävät tarkoituksella tähän tarkoitukseen kehitettyjä tavujonoja vaikeuttamaan analyysiä, ja staattista analyysiä tehdessään analysoijan on osattava kiertää ne päästäkseen näkemään ohjelman oikean assembly-listauksen. Tässä alaluvussa käydään läpi disassemblerin toimintaperiaate ja tapoja harhauttaa disassembleria tuottamaan virheellistä koodia.

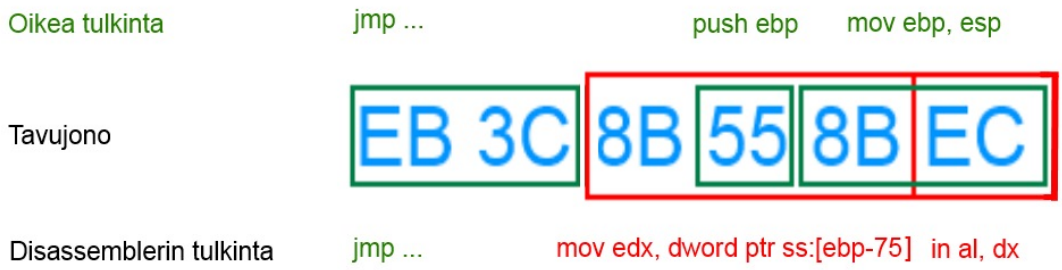
6.2.1 Disassemblerin toimintaperiaate

Toimintaperiaatteeltaan disassemblerit jakaantuvat kolmeen pääryhmään: lineaarisiin, rekursiivisiin ja hybridimenetelmiä käyttäviin. Lineaarinen disassembler, kuten GNU objdump [2] lukee ohjelman "sokeasti"alusta loppuun käsky kerrallaan. Rekursiivinen disassembler aloittaa lukemisen ohjelman suorituskohdasta (entry point) suorituksen päättymiseen. Hypyt ja kutsut käsitellään omina haaroinaan vastaavaan tapaan. Kaikki ohjelman osat joita ei ole käyty läpi rekursiolla oletetaan dataksi. Hybridimenetelmiä ovat menetelmät jotka käyttävät sekä lineaarista että rekursiivista menetelmää. Disassemblerit voivat myös käyttää heuristisia menetelmiä (koodirakenteiden tunnistus) parantamaan käännöksen tarkkuutta jos ohjelma on käännetty tietyllä kääntäjällä. (Schwarz, Debray ja Andrews 2002)

Lineaarisen menetelmän suurin vahvuus on sen yksinkertaisuus, suurin heikkous sen kykene-

2. osa GNU Binutils-työkalupakettia, http://www.gnu.org/software/binutils/

mättömyys erottaa käskyjä datasta. Yksikin datatavu käskyvirrassa voi saada disassemblerin kadottamaan synkronointinsa ja tuottamaan kelvotonta koodia, kuten kuva 16 osoittaa. Rekursiivinen menetelmä on immuuni tälle ongelmalle, minkä vuoksi sitä käytetään suurimmassa osassa moderneita disassemblereita (Eilam 2011, s. 337). Sen kompastuskivenä ovat kuitenkin epäsuorat hypyt. Hyppykäskyn (tai kutsun) kohdeosoitteena voi olla suoran osoitteen sijasta rekisterin tai muistipaikan arvosta laskettu osoite, jota voi olla mahdotonta selvittää ajamatta ohjelmaa. Osoitteiden selvittämiseen on menetelmiä (datavuoanalyysi, program slicing, constant propagation), mutta usein ne on helpompi selvittää dynaamisesti debuggerilla. (Schwarz, Debray ja Andrews 2002)



Kuvio 16. Datajonon väärä tulkinta lineaarisessa disassemblerissa. Tavu 8B on todellisuudessa dataa, mutta disassembler tulkitsee sen käskyn ”mov” ensimmäiseksi tavuksi.

Jalostuneemmat interaktiiviset disassemblerit kuten IDA Pro antavat analysoijan merkitä käsin alueita alkuperäisestä ohjelmatiedostosta tulkittavaksi eksplisiittisesti koodiksi tai dataksi. Näin harjaantunut analysoija voi tunnistaa kuvan 16 kaltaiset tilanteet ja palauttaa disassemblerin synkronisaation. (Sikorski, Honig ja Lawler 2012, s. 334)

6.2.2 Ohjelmakoodin muuttaminen käsin

Osa disassemblereista tarjoaa mahdollisuuden muuttaa tutkittavaa ohjelmaa korvaamalla käskyjä toisilla käskyillä tai ”kommentoimalla” niitä pois nop (no-operation) -käskyillä. Tämä tulee usein tarpeeseen obfuskoituja ohjelmatiedostoja tutkittaessa. Sikorski, Honig ja Lawler 2012 esittää helpon Python-makron valittujen tavujen korvaamiseen nop-käskyillä (kuva 16).

Vaihtoehtoinen, disassemblerista riippumaton tapa muuttaa ohjelmakoodia on muokata oh-

```
import idaapi

idaapi.CompileLine('static n_key() { RunPythonStatement("nopIt()"); }')

AddHotkey("Alt-N", "n_key")

def nopIt():

      start = ScreenEA()
      end = NextHead(start)
      for ea in range(start, end):
            PatchByte(ea, 0x90)
      Jump(end)
      Refresh()
```

Kuvio 17. IDA Python -skripti käskyjonon korvaamiseen mitään tekemättömillä nop-käskyillä. Lähde: Sikorski, Honig ja Lawler 2012, s. 340

jelmatiedostoa suoraan heksaeditorilla. Yleensä disassemblerit kertovat kullekin käskylle niiden paikan (offset) ohjelmatiedostossa. Tämä tapa vaatii tiedon haluttujen assembly-käskyjen konekielisistä vastineista, mutta sen voi löytää esimerkiksi Intelin arkkitehtuurikäsikirjasta (“Intel 64 and IA-32 Architectures Software Developers Manual” 2013). Tärkeimpiä käskyjä ja niiden konekielisiä vastineita on käsitelty myös liitteessä 2.

6.2.3 Ehdottomat ehdolliset hypyt

Ehdolliset hyppykäskyt tarjoavat lukuisia mahdollisuuksia rekursiivisen disassemblerin harhaanjohtamiseen. Rekursiivisessa disassembly-algoritmissa käydään joka kerta läpi ehdollisen hypyn molemmat haarat, eli hypyn kohde ja sitä seuraava käsky johon siirrytään jos ehto ei toteudukaan. Yleensä disassemblerit käsittelevät käskyn jälkeen tulevan haaran ensin (Sikorski, Honig ja Lawler 2012, s. 336) ja siten konfliktitilanteessa luottavat siihen enemmän kuin hyppykäskyn takana olevaan haaraan. Tämä tekee ehdottomien hyppyjen korvaamisesta ehdollisilla pätevän keinon disassemblerin harhauttamiseen.

Eräs käytetyimmistä obfuskaatiotavoista on korvata ehdottomia hyppyjä kahdella ehdollisella hypyllä, joiden ehto on sama mutta vastakkainen (Sikorski, Honig ja Lawler 2012, s. 334–335). Näin hyppy säilyy edelleen ehdottomana, mutta jos disassembler ei ole ottanut tätä erikoistapausta huomioon, se luulee jälkimmäisen hypyn jälkeen alkavaa käskyä mahdolliseksi suorituspoluksi. Käskyjen jälkeen seuraa sopivan mittainen lohko dataa, jolloin

disassembler menettää synkronointinsa. Hyppyjen väliin voidaan myös lisätä käskyjä jotka eivät tee mitään (esimerkiksi kasvattavat ja vähentävät rekisterin arvoa samalla luvulla) analysoijan ja disassemblerin harhauttamiseksi edelleen.

Toinen samantyylinen, suosittu keino on hyppy, jossa toiseen haaraan ei koskaan siirrytä. Koska disassembly-algoritmit suosivat hyppykäskyn jälkeistä käskyä, seuraava todellinen käsky löytyy yleensä ehdollisen hypyn kohteesta. (Sikorski, Honig ja Lawler 2012, s. 336)

Näille ja muille ehdollisia hyppyjä käyttäville obfuskaatiotekniikoille on ainakin kaksi vaihtoehdoista ratkaisua. Ohjelmatiedostoa muuttamattomana ratkaisuna disassemblerin ohjelmakoodiksi tulkitsemat data-alueet voi merkitä dataksi, jos disassembler tarjoaa siihen mahdollisuuden. Toisena vaihtoehtona ehdollisilta näyttävät hypyt voi palauttaa takaisin ehdottomiksi hypyiksi joko disassemblerissa tai esimerkiksi heksaeditorilla. (Sikorski, Honig ja Lawler 2012, s. 336)

6.2.4 Hyppykäskyt

Ehdolliset ja ehdottomat hyppykäskyt muuttavat rekisteriä EIP, joka sisältää seuraavan suoritettavan käskyn muistiosoitteen. Rekisteriin ei voida sijoittaa suoraan, mutta siihen voidaan vaikuttaa muilla tavoilla. Syynä varsinaisten hyppykäskyjen välttämiseen voi olla esimerkiksi alkuperäisen ohjelmnan aloituskohdan löytämisen vaikeuttaminen pakatussa haittaohjelmassa, tai kontrollivuon hämärtäminen analysoijalta ja disassemblerilta.

Epäsuorat hypyt, eli sellaiset joiden kohdeosoite lasketaan suorituksen aikana, ovat staattisen analyysin perusongelma. Epäsuora hyppy voidaan toteuttaa joko käsin assembly-kielellä, tai korkeamman tason kielellä käyttämällä funktio-osoittimia. Disassemblerille kaikki paitsi triviaali tapaus, osoitteen lataaminen sellaisenaan muistiin tai rekisteriin, on vielä selvittämätön ongelma. Hypyn osoite täytyykin selvittää joko staattisesti tarkastelemalla ohjelmakoodia tai dynaamisesti tutkimalla ohjelman tilaa debuggerissa. IDA Pro ei anna lisätä funktioviitteitä käsin suoraan käyttöliittymästä, mutta `AddCodeXref`-skriptifunktiota käyttämällä se on mahdollista. (Sikorski, Honig ja Lawler 2012, s. 340–342)

Jos hyppykäskyjä halutaan välttää kokonaan, voidaan käyttää käskyä call tai ret. Tavallisesti käskyjä käytetään funktiokutsuihin ja niistä palaamiseen, mutta toteutukseltaan ne ovat

vain hyppykäskyjä pino-operaatiolla varustettuina. Call-käskyn jälkeen paluuosoitteen pinosta poistamalla tai ret-käskyä ennen osoitteen lisäämällä pinoon käskystä saadaan yksisuuntainen. Lisähyötynä (tai -haittana analyysin kannalta) call-käskyn tapauksessa disassembleri olettaaa, että kutsusta palataan, ja tulkitsee sen jälkeen tulevat tavut käskyinä. Tällainen hyppy harhauttaa myös funktiograafeja rakentavia disassemblereita, ja saattaa jopa kaataa sen jos tilannetta ei ole otettu mitenkään huomioon. (Sikorski, Honig ja Lawler 2012, s. 342–343)

Hyppyjen ja aliohjelmakutsujen lisäksi suorituksen voi katkaista myös poikkeus. Windowsissa poikkeusten käsittelystä vastaa rakenteinen poikkeustenkäsittely (SEH, structured exception handling). Poikkeukset korkeamman tason ohjelmointikielissä on tavallisesti toteutettu rakenteisen poikkeustenkäsittelyn avulla, mutta luovasti käytettynä sitä voidaan käyttää myös hyppykäskynä, joka hämmentää staattista analyysiä ja voi saada debuggerin kaatumaan. (Sikorski, Honig ja Lawler 2012, s. 344)

SEH-poikkeuksenkäsittelijät sijaitsevat linkitettynä listana FS-rekisterin ensimmäisen DWORD-arvon osoittamassa muistipaikassa. Kukin listan solmuista koostuu kahdesta 32-bittisestä osoitteesta, joista ensimmäinen osoittaa edelliseen solmuun ja toinen poikkeuksen käsittelevään funktioon. Hyppyä varten väliaikainen poikkeuksenkäsittelijä voidaan tehdä pinoon kuvan 18 mukaisella ohjelmakoodilla. (Sikorski, Honig ja Lawler 2012, s. 344–345)

```
push ExceptionHandler
push fs:[0]
mov fs:[0], esp
```

Kuvio 18. Strukturoidun poikkeuksenkäsittelyn asentaminen pinoon. Lähde: Sikorski, Honig ja Lawler 2012, s. 354

Itse poikkeus voidaan aiheuttaa esimerkiksi jakamalla nollalla, osoittamalla muistia kielletyltä alueelta tai kutsumalla `RaiseException`-funktiota. Poikkeus saadaan yksisuuntaiseksi poistamalla asetettu poikkeuksenkäsittelijä sekä toinen poikkeuksenkäsittelijä, jonka Windows on poikkeuksen tapahtuessa pinoon lisännyt (ks. kuva 19). (Sikorski, Honig ja Lawler 2012, s. 344–345)

```
mov esp, [esp+8]
mov eax, fs:[0]
mov eax, [eax]
mov eax, [eax]
mov fs:[0], eax
add esp, 8
```

Kuvio 19. Strukturoidun poikkeuksenkäsittelijän poistaminen poikkeuksen jälkeen. Lähde: Sikorski, Honig ja Lawler 2012, s. 354

6.3 Debuggerin ja virtuaalikoneen tunnistus

Sekä debuggeri että virtuaalikone ovat tärkeä osa dynaamista analyysiä. Yhteisenä tekijänä kummallekin ohjelman suoritus niiden sisällä voi poiketa normaaliympäristöstä, ja tällaisen tilanteen tunnistettuaan haittaohjelma voi vastatoimena esimerkiksi lopettaa itsensä, mennä ikuiseen silmukkaan tai jopa kaataa suoritusympäristönsä. Helpoin ja varmin tapa näiden menetelmien kiertämiseen on poistaa tarkistukset kokonaan ohjelmasta staattisessa analyysissä. Debuggereille on myös saatavilla lisäosia tunnetuimpia antidebuggauskeinoja vastaan. (Sikorski, Honig ja Lawler 2012, s. 348)

Huomattavin ero debuggerissa tai virtuaalikoneessa ajetussa ohjelmakoodissa on sen hitaampi suoritus. Suoritusnopeus itsessään ei poikkea niin paljoa natiivista, että sitä voitaisiin käyttää tunnistukseen, mutta debuggeri voidaan tunnistaa, jos se pysäyttää ohjelman kahden mittauspisteen väliin. Virtuaalikoneissa ja emulaattoreissa taas käskyjen suhteellinen suoritusaika keskenään voi poiketa natiivina suoritettuihin mittausarvoihin (Raffetseder, Krügel ja Kirda 2007, s. 6–7). Tyypillisesti ajoitusten mittaamiseen käytetään assembly-käskyä `rdtsc` tai API-funktioita `QueryPerformanceCounter` ja `GetTickCount`.

Windows API tarjoaa valmiita funktioita debuggerien tunnistukseen, kuten `IsDebuggerPresent`, mutta koska niiden käyttö on niin ilmeistä, haittaohjelmat harvemmin käyttävät niitä. Sen sijaan ne käyttävät usein Windowsin tietorakenteita suoraan. Rekisteri FS osoittaa muistialueeseen, josta käytetään nimitystä Thread Information Block (TIB), ja joka sisältää aiemmin esiteltyjen poikkeuskäsittelijöiden lisäksi myös muuta prosessi- ja säiekohtaista tietoa. (Sikorski, Honig ja Lawler 2012, s. 352–353)

TIB-lohkon tavut 0x30–0x33 (`fs:[30h]`) sisältävät osoittimen prosessiympäristölohkoon

(PEB, process environment block), johon on tallennettu tietoja prosessin suoritusympäristöstä. Prosessitietostruktuurista [3] voidaan päätellä debuggerin käyttö ainakin kolmella eri tavalla:

- Tavun 0x03 (`BeingDebugged`) arvosta 1
- Tavujen 0x10–0x13 (`Reserved4`) osoittaman `ProcessHeap`-rakenteen lippumuuttujasta `ForceFlags`. Muuttujan paikka rakenteessa on joko 0x44 (Windows XP) tai 0x40 (Windows 7) sen alusta laskettuna. Lipun nollasta poikkeava arvo paljastaa, että ohjelman pino on luotu debuggerissa.
- Tavun 0x68 (`NtGlobalFlag`) arvo 0x70 = `FLG_HEAP_ENABLE_TAIL_CHECK | FLG_HEAP_ENABLE_FREE_CHECK | FLG_HEAP_VALIDATE_PARAMETERS`, joka myös kertoo, että pino on luotu debuggerissa.

Kuvassa 20 on esitelty kaikki kolme tunnistustapaa yhdistettynä. (Sikorski, Honig ja Lawler 2012, s. 353–355)

```
mov eax, dword ptr fs:[30h]

mov ebx, byte ptr [eax+2]
test ebx, ebx
jne DebuggeriTunnistettu

mov ebx, dword ptr [eax+18h]
cmp dword ptr [ebx+10h], 0
jne DebuggeriTunnistettu

cmp dword ptr [eax+68h], 70h
je DebuggeriTunnistettu

; Ei debuggeria
```

Kuvio 20. Debuggerin tunnistus kolmella tavalla prosessiympäristölohkoa käyttäen. Lähde: Sikorski, Honig ja Lawler 2012, s. 354–355

Debuggauksessa ohjelmaan asetettavat `int 3` -käskyt, eli ohjelmalliset keskeytyskohdat, ovat myös potentiaalinen tapa debuggerin tunnistukseen. Tunnistus voidaan toteuttaa joko etsimällä tavua `0xCC` koodisektiostaan tai vieläkin tehokkaammin laskemalla sektiosta tai sen osasta tarkistussumma. Tarkistussumman laskeminen estää myös tehokkaasti muutosten

3. PEB-struktuuri: http://msdn.microsoft.com/en-us/library/windows/desktop/aa813706(v=vs.85).aspx

tekemisen koodiin tarkistusta poistamatta. Tarkistus ei kuitenkaan tunnista laitteistokeskeytyskohtia, sillä ne eivät muuta ohjelmakoodia. (Sikorski, Honig ja Lawler 2012, s. 357)

Sekä debuggeri että virtuaalikone jättävät myös muita jälkiä järjestelmään. Haittaohjelma voi tutkia rekisteriä tai Program Files -kansiota etsien jälkiä ohjelmista, kuten IDA Pro, OllyDbg tai VMWare Tools. Myös ikkunoita niiden otsikolla etsivää `FindWindow` -funktiota voidaan käyttää tähän tunnistukseen. Kaikkia jälkiä ei ole aina mahdollista tai käytännöllistä poistaa, mutta ilmeisimmät ja tarpeettomimmat, kuten virtuaalikoneisiin asennettavat lisätyökalut kannattaa jättää asentamatta. Jos ohjelmatiedosto sisältää merkkijonoja kuten ”VMWare” tai ”OllyDbg”, helppo tapa tunnistuksen sekoittamiseen on myös korvata nämä merkkijonot jollain muulla. (Sikorski, Honig ja Lawler 2012, s. 356, 370–373)

Tähän mennessä esitellyt tunnistuskeinot ovat perustuneet pitkälti ohjelmistopohjaisiin eroihin. Virtuaalikoneiden tunnistuksessa hyödynnetään paljon myös laitteistopohjaisia eroja, kuten dokumentoimattomien assembly-käskyjen tuottamia tuloksia, prosessorin laitekohtaisia rekistereitä sekä oheislaitteiden, kuten verkkokorttien ja näytönohjainten antamia tunnistetietoja. (Raffetseder, Krügel ja Kirda 2007, s. 6)

Virtuaalikoneen ja emulaattorin ero on siinä, että siinä missä emulaattori tulkitsee jokaisen käskyn, virtuaalikoneessa ne ajetaan normaalisti prosessorilla. Etuoikeutetut käskyt (privileged instructions), joita voidaan suorittaa Windowsissa vain kernel-tilassa, hallitaan käyttäjätilassa käsittelemällä niiden aiheuttama yleinen suojausvirhe (general protection fault) ja simuloimalla niiden toimintaa virtuaalikoneen sisällä. Tällaisia käskyjä ovat tyypillisesti kaikki suoraan laitteiston kanssa kommunikoivat käskyt. x86-käskykanta sisältää kuitenkin 17 laitteiston kanssa keskustelevaa käskyä, jotka eivät ole etuoikeutettuja, ja siten eivät aiheuta poikkeusta. Näitä käskyjä sanotaan virtualisoimattomiksi, sillä niitä ei voida simuloida oikein ilman täyttä järjestelmäemulaatiota. (Raffetseder, Krügel ja Kirda 2007, s. 4–5)

Ehkä tunnetuin käyttäjätilan laitteistokäskyjä hyödyntävistä tunnistuskeinoista on Joanna Rutkowskan kehittämä ja nimeämä Red Pill -tekniikka, joka hyödyntää virtualisoimatonta käskyä `sidt`. Sidt-käsky tallentaa prosessorin IDTR-rekisterin arvon, joka sisältää osoittimen järjestelmän keskeytystenkäsittelytauluun (interrupt descriptor table). Virtuaalikoneelle tämä taulu on eri paikassa kuin käyttöjärjestelmälle, esimerkiksi VMWarella osoite on

0xFFxxxxxx ja Virtual PC:llä 0xE8xxxxxx, joista virtuaalikone voidaan tunnistaa (Rutkowska 2004). Red Pill kehitettiin vuonna 2004, jolloin suurin osa prosessoreista koti- ja yrityskäytössä oli yksiytimisiä. Nykyään testi on vähemmän luotettava, sillä jokaisella prosessoriytimellä on oma IDT-taulunsa, joista osa voi tunnistua virtuaalisiksi natiiviympäristössäkin. (Quist ja Smith 2006s)

Vastaava No Pill -niminen tekniikka hyödyntää käskyä `sldt`. Windows ei käytä LDT (local data table) -rakennetta lainkaan, mutta esimerkiksi VMWare alustaa sen ja asettaa sille osoittimen. Näin jos sldt-käsky palauttaa muun tuloksen kuin 0, suoritusympäristö on todennäköisesti virtuaalinen. Toisin kuin Red Pill, No Pill -tunnistus toimii luotettavasti myös moniydinprosessoreilla (Quist ja Smith 2006s). VMWaressa LDT-taulun alustus voidaan kuitenkin poistaa käytöstä kytkemällä prosessorin kiihdytys pois päältä asetuksista. (Sikorski, Honig ja Lawler 2012, s. 356, 375)

Käskyjen sidt ja sldt lisäksi myös ainakin käskyjä sgdt, smsw, str ja in voidaan käyttää virtuaalikoneen tunnistukseen (Sikorski, Honig ja Lawler 2012, s. 377). Näitä käskyjä ei käytetä normaalisti mihinkään muuhun haittaohjelmissa, joten ne on helppo löytää assemblylistauksesta, jos epäilyttää että ohjelma ei toimi oikein. Käskyt ja niiden tarkistukset voi poistaa ohjelmakoodista tässä luvussa aiemmin esitellyllä IDAPython-skriptillä tai heksaeditoria käyttämällä. Näitä käskyjä ja muutamia VMWare-virtuaalikoneelle tehtyjä tunnistusmenetelmiä voi kokeilla ScoopyNG-työkalulla, joka on saatavilla osoitteesta `http://www.trapkit.de` (ks. kuva 36).

7 Haittaohjelman analyysi käytännössä

Tässä tutkielman empiirisessä osassa kokeilemme aiemmissa luvuissa käsittelemiämme menetelmiä todellisen haittaohjelman analyysiin. Tutkimus hyödyntää sekä staattista että dynaamista analyysiä edeten ensimmäisten lukujen pinnallisesta tarkastelusta syvälle koodianalyysiin disassemblerin ja debuggerin avulla. Haittaohjelmassa on käytetty myös joitain analyysin vastaisia keinoja, joiden toiminnan pyrimme selvittämään ja ohittamaan sellaisen kohdatessamme.

7.1 Tutkittava haittaohjelma

Valitsin tutkittavaksi haittaohjelmaksi MyDoom-madon oletetun M-variantin kahdesta syystä. Tärkeimpänä syynä se sopi parhaiten työn laajuuteen – esimerkiksi uudemmat Conficker- ja Zeus -haittaohjelmat ovat korkeasti metamorfisia ja koostuvat useista komponenteista. Toiseksi minulla oli haittaohjelmasta binääritiedosto valmiina aiemmalta tietoturvakurssilta. Analysoin haittaohjelmaa myös kurssilla, mutta huomattavasti suppeammin.

MyDoom on vuodelta 2004 peräisin oleva laajalti levinnyt haittaohjelma, joka saastutti miljoonia tietokoneita ympäri maailmaa. Sen leviämiskanavia olivat sähköposti ja p2p-verkot. Alkuperäisen haittaohjelman tärkein funktio oli kaataa utahilaisen SCO-yrityksen verkkosivut denial of service -hyökkäyksellä, todennäköisesti protestina SCO:n aiempiin yrityksiin häiritä Linux-käyttöjärjestelmän kehitystä ohjelmistopatentteihinsa nojaten. MyDoom sisälsi myös muuta toiminnallisuutta variantista riippuen, mukaanlukien takaoven ja näppäimistön kuuntelijan. (“Security firm: MyDoom worm fastest yet” 2014)

Tutkittavan haittaohjelmanäytteen tiedostonnimi on ”Email-Worm-Win32.Mydoom.m” ja sen MD5-summa on d8d9ebce2ff9f94ee0855c0d3e756049. Sen koko on 28 864 tavua ja kuvake exe-tiedostoksi nimettynä kirjekuori. Muutospäivämäärä tiedoston ominaisuuksista katsottuna on 18.7.2004. VirusTotal.com -sivusto tunnistaa tiedoston MyDoom-madoksi, joskin sen variantin nimeäminen vaihtelee virusskannerista riippuen 37. Vain yksi antivirusohjelma ei tunnistanut näytettä.

PEiD-analyysiohjelma tunnistaa, että haittaohjelma on pakattu UPX-pakkausohjelmalla 38. Jotta ohjelmaa voitaisiin tutkia tarkemmin, pakkaus täytyy ensin purkaa.

7.2 Tutkimusympäristö

Valitsin tutkimusympäristöksi Oraclen VirtualBoxin [1] lähinnä tottumuksesta sen käyttöön. Sen ominaisuuksiin kuuluu myös virtuaalikoneen tilan tallentaminen ja automaattinen aikajanan rakentaminen tallennetuista tiloista, mikä antaa vapaammat kädet yrityksille ja erehdyksille.

Käyttöjärjestelmäksi virtuaalikoneeseen asensin Windows XP:n 32-bittisen version kahdesta syystä: toisaalta se oli yleisin käyttöjärjestelmä MyDoom-viruksen julkaisuaikaan, ja toisaalta IDA Pro Free 5.0 -disassembleri toimii siinä paremmin kuin uudemmilla käyttöjärjestelmillä. Virtuaalikoneen asetuksista tärkeimpinä rajoitin virtuaalikoneen pääsyn vain sen sisäiseen verkkoon ja kytkin jaetut kansiot ja drag-and-drop -tiedostonjakamisen pois päältä. Nämä ovat aiheellisia turvatoimia, jotta haittaohjelma ei pääsisi leviämään testiympäristön ulkopuolelle.

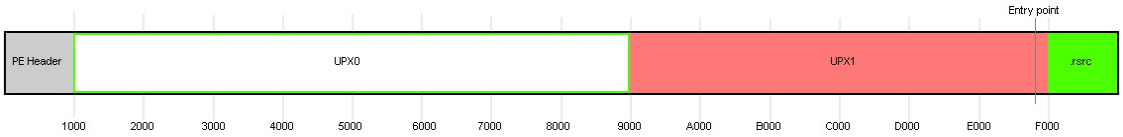
Työkalut, joita tässä tutkimuksessa käytän on ”poltettu” virtuaaliselle cd-levylle eli ISO-tiedostoon, jotta haittaohjelma ei pääsisi muuttamaan niitä. Suurin osa virtuaalikoneista ymmärtää ISO-tiedostomuotoa ja virtuaaliselle käyttöjärjestelmälle se näyttää aivan tavalliselta CD- tai DVD-levyltä.

7.3 Pakkauksen purku

UPX-pakkaus voidaan purkaa pakkausohjelmalla itsellään, mutta puramme sen tässä käsin selvittääksemme miten pakkaus toimii. PEview-ohjelma paljastaa, että ohjelmatiedostossa on kolme sektiota: ”UPX0”, ”UPX1” ja ”.rsrc”. `IMAGE_FILE_HEADER` -otsikkosektiosta löytyisi myös aikaleima, joka kertoisi milloin tiedosto on pakattu, mutta se on poistettu. 39

Sektioiden alkuosoitteita ja kokoja tarkastelemalla voimme piirtää rakennekuvan tiedostosta 21. Sektio UPX0 on fyysiseltä kooltaan (Size of Raw Data) nolla, mikä tarkoittaa että

1. https://www.virtualbox.org/



Kuvio 21. Rakennekuva pakatusta haittaohjelmasta ja sen sektioista.

ohjelman käynnistyksen yhteydessä se on tyhjä. Sen virtuaalinen koko (Virtual Size) 8000 on maksimikoko datalle, joka tähän sektioon voidaan sijoittaa ohjelman suorituksen aikana. Huomaa, että nämä arvot ovat heksadesimaalilukuja. Tämä on hyvin todennäköisesti sektio, johon purettu ohjelma kirjoitetaan. 40

Sektio UPX1 sisältää kokonaisuudessaan alustettua dataa, sillä sekä sen fyysinen että virtuaalinen koko ovat samat. Sektio sisältää myös oikeudet `READ` ja `EXECUTE`, ja `IMAGE_OPTIONAL_HEADER`-otsikkosektiosta nähdään, että ohjelman aloituskohta osoittaa tähän sektioon. Tämä sektio sisältää siten sekä purkukoodin että pakatun alkuperäisen ohjelmakoodin. 41

Pakkauksen purkamiseen helpoin tapa on antaa ohjelman purkaa itse itsensä, pysäyttää sen suoritus ja tallentaa se levylle. OllyDbg-debuggeri ja sen OllyDump -lisäosa sopivat hyvin tähän tarkoitukseen. Ladattaessa ohjelmaa debuggeriin huomataan, että ensimmäinen käsky on PUSHAD, joka tallentaa kaikkien rekistereiden sisällön pinoon 22. Tämä on tyypillistä pakatuissa ohjelmissa. Kun pakkaus on purettu, rekisterien sisältö palautetaan POPAD-käskyllä ennen hyppyä purettuun koodiin. Purkusilmukka voidaan tunnistaa näiden käskyjen välistä, mutta yleensä sen analyysille ei ole tarvetta, jos ei nimenomaan haluta tehdä omaa purkuohjelmaa.

```
0050ED00  60              PUSHAD
0050ED01  BE 00905000     MOV ESI,Email-Wo.00509000
0050ED06  8DBE 0080FFFF   LEA EDI,DWORD PTR DS:[ESI+FFFF8000]
0050ED0C  57              PUSH EDI
0050ED0D  83CD FF         OR EBP,FFFFFFFF
0050ED10  EB 10           JMP SHORT Email-Wo.0050ED22
```

Kuvio 22. Kuvakaappaus OllyDbg-ohjelmasta: PUSHAD-käsky ohjelman alusta.

OllyDbg antaa pysäyttää suorituksen myös tiettyihin käskyihin 42. Kun pysäytysehdoksi annetaan käsky POPAD ja päästetään ohjelma ajoon (huom. trace into, ei run), se purkaa itsensä

ja pysähtyy juuri ennen rekistereiden palauttamista 42. Tästä seuraava hyppy onkin hyppy puretun ohjelman alkuun eli alkuperäiseen käynnistyskohtaan (original entry point, OEP). Nyt täysin purettu ohjelma voidaan vedostaa levylle OllyDump-lisäosalla. PE-sektioiden ja niiden otsikkotietojen vedostamisen lisäksi täytyy vielä myös muuttaa käynnistyskohta ja rakentaa ohjelmalle import-taulu. OllyDump osaa joissain tapauksissa rekonstruoida puretun ohjelman import-taulun, mutta esimerkiksi tässä tapauksessa se ei onnistunut suoraan. Tällöin täytyy käyttää erillistä ImpREC-ohjelmaa. ImpREC liittää itsensä pakattuun prosessiin (debuggerissa käynnistetty prosessi käy tähän hyvin), ja kun sille annetaan alkuperäisen käynnistyskohdan osoite, se rakentaa siitä import-taulun. Taulu voidaan kirjoittaa OllyDumpilla vedostettuun exe-tiedostoon, jolloin siitä tulee täysin toimiva.

Purettua ohjelmaa PEView-ohjelmalla tarkasteltaessa huomataan ImpREC-ohjelman lisäämä ”.mackt”-sektio ja import-osoitetaulu ohjelmakoodin sisältävän sektion alla 43. Nimesimme UPX0-sektion uudelleen ”.text”-sektioksi OllyDumpin asetuksissa, sillä osa työkaluista odottaa sitä nimeä koodisektiolta, ja merkitsimme sen sisältämään alustettua dataa. UPX1-sektio, joka aiemmin sisälsi vain pakatun datan ja sen purkualgoritmin, on myös muuttunut ja todennäköisesti sisältää ohjelmalle tarpeellista dataa, joten sitä ei kannata poistaa.

7.4 Puretun tiedoston staattinen tarkastelu

Nyt kun pakkaus on saatu purettua, tiedostoa voidaan tutkia tarkemmin. Strings-ohjelma, joka näyttää tiedostossa puhtaana tekstinä olevat merkkijonot, löytää siitä 1640 merkkijonoa. Osa niistä on pelkkää roskaa (dataa tulkittuna merkkijonoiksi), mutta osa niistä antaa yllättävänkin syvällistä tietoa ohjelman toiminnallisuudesta.

Merkkijonoista näkee jo selvästi, että kyseessä on sähköpostin välityksellä leviävä haittaohjelma. SMTP-käskyjen (`mail`, `rcpt`) lisäksi siitä löytyy myös kokonaisia muotoilumerkkijonoja sähköpostiviestin sisällöksi 44. Lisäksi merkkijonot, kuten ”abuse”, ”bugs” ja ”submit”, viittaavat tyypillisiin oletussähköpostilaatikoihin, ja linkitys ”ws2_32.dll”-kirjastoon on lähes varma merkki jonkinlaisesta verkkotoiminnallisuudesta.

Myös import-taulun sisältö eli eri kirjastoista dynaamisesti linkitetyt funktiot ovat näkyvis-

sä merkkijonoissa, ja jopa ajonaikaisesti linkitettävät kirjastot ja niistä ladatut funktiot, jos niiden nimet on tallennettu selväkielisenä. Import-taulu PEView-ohjelmasta tarkasteltuna paljastaa kirjastot ”kernel32.dll”, ”user32.dll”, ”msvcrt.dll”, ”advapi32.dll” ja ”ws2_32.dll”. Niiden lisäksi strings-ohjelma paljastaa kirjastot ”dnsapi.dll”, ”iphlpapi.dll”, ”urlmon.dll” ja ”wininet.dll”. Taulukkoon 8 on koottu merkkijonoista löydettyjä mielenkiintoisia funktioita, joihin kannattaa myöhemmin kiinnittää huomiota.

Ohjelma sisältää myös muita mielenkiintoisia merkkijonoja, joiden tarkoitusta ei voi vielä päätellä, mutta jotka tulevat varmasti vastaan myöhemmässä analyysissä.

- *sf.net, sourceforge, google, yahoo, microsoft* ym: suosittuja web-sivustoja
- *hello, error, status, test, report*: mahdollisia komentoja esim. takaovea varten?
- *zincite*: mahdollisesti tekijän pseudonyymi tai käyttäjätunnus
- hakumerkkijonoja Lycos, Altavista, Yahoo, Google -hakukoneille

Resource Hacker ei paljasta ohjelmasta muita resursseja kuin kuvakkeen (kirjekuori). Tarkempaa staattista tarkastelua varten on siirryttävä disassemblerin käyttöön.

7.5 Puretun tiedoston dynaaminen tarkastelu

Ennen haittaohjelman ajamista virtuaalikoneen tila kannattaa tallentaa snapshot-toiminnolla, niin haittaohjelmasta voi tehdä uuden puhtaan asennuksen joutumatta rakentamaan kokonaan uutta virtuaaliympäristöä. Näin testiajoja voi tehdä useampia ja verrata eri ohjelmilla saatuja tuloksia toisiinsa. Varotoimena kannattaa vielä tarkistaa, ettei virtuaalikoneesta ole verkkoyhteyttä ulkomaailmaan.

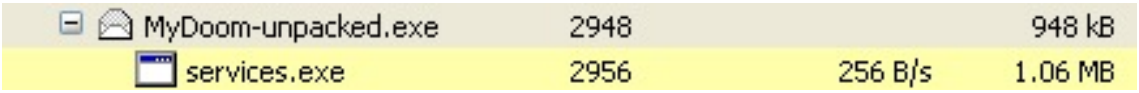
Ensimmäistä testiajoa varten käynnistämme Fakenet-verkkosimulaattorin, Process Hacker -tehtävänthallinnan ja Process Monitor -seurantaohjelman rekisteri- ja tiedostomuutosten tarkkailuun. Process Monitor näyttää kaiken järjestelmässä tapahtuvan toiminnan, joten sen tulostusta on syytä suodattaa. Haittaohjelmat nimeävät usein itsensä joksikin järjestelmäprosessiksi, joten suodatus kannattaa tehdä mieluummin prosessinumeron kuin tiedoston nimen perusteella.

Haittaohjelman käynnistyksen yhteydessä Windowsin palomuuri varoittaa ”services”-nimisestä

ohjelmasta, joka yrittää päästä verkkoon 23. Process Hacker varmistaa, että sellainen prosessi todella löytyy, ja haittaohjelmaprosessi on käynnistänyt sen Windows-hakemistosta. Todellinen ”services.exe”, joka ylläpitää Windowsin palveluita, sijaitsee system32-hakemistossa, joten ”C:\Windows\services.exe” on haittaohjelman osa, jonka se on ajon aikana purkanut Windows-hakemistoon. Käyttäjän lisäksi nimi hämää myös käyttöjärjestelmää niin, että se ei anna käyttäjän lopettaa prosessia tehtävänhallinnasta tai taskkill-työkalulla. Esimerkiksi Process Hacker -työkalulla prosessin lopettaminen kuitenkin onnistuu.

Kuvio 23. Windowsin palomuurivaroitus haittaohjelman ajon yhteydessä. Kuvakaappaus.

Process Hacker näyttää käynnissä olevat haittaohjelmaprosessit ”MyDoom-unpacked.exe” ja ”services.exe”, josta jälkimmäinen on käynnistetty ensimmäisen aliprosessina. Ensimmäisellä niistä on kaksi säiettä alkaen virtuaaliosoitteesta 0x3280 (ohjelman tulokohta kuten purettaessa todettiin) ja 0x4c1b. Services.exe -prosessi ajaa kolmea säiettä osoitteista 0x5770, 0x1c36 ja wshelp32.dll -kirjaston WahQueueUserApc -funktiosta osoitteesta 0x5c. Näistä osoitteista voi olla apua myöhemmin disassemblerin kanssa.



Kuvio 24. Kuvakaappaus Process Hacker -ohjelmasta, jossa näkyy kaksi haittaohjelmaprosessia.

Kolmesta ohjelmasta eniten tietoa antaa Process Monitor, joka paljastaa services.exen todellakin MyDoom-prosessin luomaksi tiedostoksi. Pääprosessi luo myös tiedoston ”zincite.log”

käyttäjän temp-hakemistoon. Tiedosto on 64 tavua pitkä ja sisältää satunnaisen näköisiä tavuja. Services.exe lukee tätä tiedostoa, ja myös kirjoittaa samaan hakemistoon omaa 32 tavun mittaista ”wwivvt.log”-tiedostoaan, jonka sisältö paljastaa itsestään yhtä vähän.

Kummatkin prosessit luovat omat käynnistysarvonsa HKEY_LOCAL_MACHINE\Microsoft\Windows\CurrentVersion\Run -rekisteriavaimen alle seuraavaa uudelleenkäynnistystä varten. Services.exe -prosessi myös kirjoittaa omaa arvoansa vähän väliä varmistaakseen, ettei sitä poisteta. Pääprosessi ei käynnistä itseään samasta paikasta kuin käyttäjä on sen tehnyt, vaan kopioi ohjelmatiedoston Windows-hakemistoon java.exe -nimellä. Ohjelmatiedostojen md5-summa on sama, joten polymorfismia ei ole tapahtunut ainakaan tässä vaiheessa.

Process Monitor antaa myös vihjeitä kryptografisesta toiminnallisuudesta (crypt32.dll) sekä Internet Explorer- selaimen sivuhistorian ja evästeiden lukemisesta. Myös TCP\IP -ja sokettiaiheisia rekisteriavaimia luetaan, mutta Process Monitor, kuten myöskään Fakenet-simulaattori ei näytä juurikaan mielenkiintoista verkkoliikennettä 45. Tämä voi olla merkki siitä, että haittaohjelma on tunnistanut olevansa laboratorioympäristössä. Syvällisempi analyysi on tarpeen hypoteesin varmistamiseksi ja lisätietojen saamiseksi.

7.6 Syvempi staattinen ja dynaaminen analyysi

Tässä vaiheessa on saatu kerättyä kaikki tieto mitä pintapuolisella tarkastelulla voidaan löytää ja on aika siirtyä varsinaisen koodianalyysin pariin. Analyysi tapahtuu IDA Pro -disassemblerilla ja yhdistelee staattista ja dynaamista lähestymistapaa tarpeen mukaan.

7.6.1 MyDoom-unpacked.exe

Pääohjelmatiedoston MyDoom-unpacked.exe (tai java.exe) aloituskohta (start IDA:n nimeämänä) näyttää kuvion 46 mukaiselta. Ohjelma kutsuu ensin WinSock-kirjaston alustavaa WSAStartup-funktiota ja sen jälkeen kahta aliohjelmaa itse ohjelmatiedostosta. Niistä ensimmäinen kutsuu GetTickCount-kirjastofunktiota ja lataa sen tuloksen muistiosoitteeseen ds:50B168. GetTickCount palauttaa järjestelmän käynnissäoloajan millisekunteina (“Microsoft Developer Network: Windows Dev Center” 2014).

Aliohjelmakutsujen jälkeen pääohjelma lopettaa itsensä ExitProcess-kutsulla, jonka jälkeen seuraa keskeytys 3. Debuggeri käyttää keskeytystä 3 suorituksen pysäyttämiseen, ja sen kirjoittaminen suoraan ohjelmakoodiin on suosittu debuggauksen vastainen keino. IDA Pro kuitenkin pitää kirjaa omista keskeytyskohdistaan eikä hämäänny näistä kutsuista, mutta muita herkempiä debuggereita varten käskyt voi olla syytä poistaa ohjelmatiedostosta ennen debuggausta.

Pääohjelma jatkuu aliohjelmaan sub_5031E4 (ks. kuva 47), jolle annamme nimen MainProc myöhempää viittausta varten. Se ottaa parametrinään säikeen tunnistenumeron pääohjelmalta ja siirtää sen osoitteen rekisteriin esi. Aliohjelmakutsun jälkeen (aliohjelma säilyttää rekisterin esi arvon) sitä verrataan lukuun 0 joka vastaa ohjelman aloittanutta säiettä. Jos säie on ohjelman pääsäie, kutsutaan toista aliohjelmaa sub_502D8E, jonka tuloksen perusteella joko palataan takaisin pääohjelmaan (jossa on jäljellä enää lopetuskoodi) tai jatketaan eteenpäin osoitteeseen 503204, kuten myös muun säikeen tapauksessa tehdään.

MainProc-funktion aliohjelma sub_502C90 sisältää kutsuja rekisterifunktioihin, mutta rekisteriavaimet eivät ole näkyvillä selväkielisenä. Sen sijaan pino sisältää runsaasti tavumuuttujia, joihin rekisteriavaimen muodostavat merkit siirretään yksitellen. IDA näyttää oletuksena tavut heksadesimaalilukuina, mutta ne voidaan näyttää myös merkkeinä. Näin tekemällä rekisteriavaimeksi paljastuu ”Software\Microsoft\Daemon”, joka luodaan sekä juuriavaimiin HKEY_LOCAL_MACHINE että HKEY_CURRENT_USER, ellei vähintään toista niistä ole olemassa. Juuriavaimet eivät ole suoraan nähtävissä koodista, mutta debuggaus auttaa niiden selvittämisessä. Rekisterin juuriavaimien ja muiden vastaavien symbolisten vakioiden numeeriset arvot ovat nähtävillä Windows Platform SDK:n tai Visual Studion mukana tulevista header-tiedostoista, ellei IDA osaa niitä siinä tilanteessa suoraan näyttää.

Toinen MainProc-funktion aliohjelma sub_502D8E luo mutex-objektin, jonka nimen se rakentaa tietokoneen verkkonimestä ja sanasta ”root”. Jos objekti on olemassa, suoritus siirtyy takaisin pääohjelmaan, jossa on jäljellä enää ExitProcess-funktiokutsu. Näin haittaohjelma varmistaa, että siitä on vain yksi instanssi kerrallaan käynnissä.

MainProc osoitteesta 503204 eteenpäin sisältää neljä aliohjelmakutsua ja jatkuu kolmeen seuraavan lohkoon, joissa kahdessa ensimmäisessä luodaan uusi säie. Kolmas lohko on ikui-

nen silmukka, joka kutsuu sekunnin välein aliohjelmaa osoitteessa 50565B. Tutkitaan ensin neljää oletettavasti ohjelman alustavaa aliohjelmaa (kuva 48).

Ensimmäinen aliohjelmista sisältää peräkkäiset kutsut GetWindowsDirectory / GetTempPath, CreateFile ja CreateProcess. Lisäksi siinä on viitteitä merkkijonoihin ”services” ja ”exe”. Debuggerilla nähdään, että aliohjelma todellakin luo ”services.exe”-tiedoston ja käynnistää sen. Tiedosto luodaan Windows-hakemistoon, tai jos se epäonnistuu, käyttäjän temphakemistoon. Myös ”zincite.log”-tiedosto luodaan aliohjelmassa sub_50746B, jota tämä aliohjelma kutsuu.

Itse ”services.exe”-tiedoston sisällön kirjoittaminen tapahtuu omassa funktiossaan osoitteessa 50737C. Koodia silmäilemällä nähdään, että tiedoston sisältö sijaitsee muistiosoitteessa 509168 ja sen koko on 8192 tavua, mikä on sama kuin levylle kirjoitetun tiedoston koko. Pintapuolisen tarkastelun harhauttamiseksi tiedosto on ”kryptattu” kääntämällä sen jokaisen tavun ensimmäinen ja viimeinen bitti xor-operaatiolla binääriarvoa 10000001 vastaan.

Toinen kuvan 48 aliohjelmista MainProc-funktiossa kopioi ohjelmatiedoston Windows-hakemistoon nimellä ”java.exe”. Tiedoston nimi on myös tässä aliohjelmassa yksittäisinä merkkeinä strings-työkalun ja muun pintapuolisen tarkastelun harhauttamiseksi. Peräkkäiset funktiokutsut GetModuleFileName, GetWindowsDirectory / GetTempPath ja CopyFile kuitenkin paljastavat nopeasti, mitä aliohjelma tekee.

Kolmas aliohjelma sisältää myös runsaasti yksittäisiä tavuja, ja kutsut RegOpenKeyEx, RegSetValueEx ja RegCloseKey. Edelleen tavut merkeiksi muuttamalla nähdään rekisteriarvo ”Software\Microsoft\Windows\Run\JavaVM”, joka luodaan HKEY_LOCAL_MACHINE -juuriavaimen alle. Juuriavain nähdään IDA:n kommentoiman ”hKey”-muuttujan arvosta, kun se muutetaan symboliseksi vakioksi. Java.exe-tiedoston sijainti on annettu parametriksi tälle aliohjelmalle.

Neljäs aliohjelma kutsuu järjestelmäfunktiota RegisterServiceProcess, jolla se rekisteröi itsensä prosessiksi, jonka suoritus jatkuu vaikka käyttäjä kirjautuu ulos. Kutsu on toteutettu ajonaikaisella linkityksellä GetModuleHandle- ja GetProcAddress-funktioiden avulla ja funktion nimi kopioidaan pinoon merkki kerrallaan, jotta se ei näkyisi import-taulussa tai merkkijonoissa.

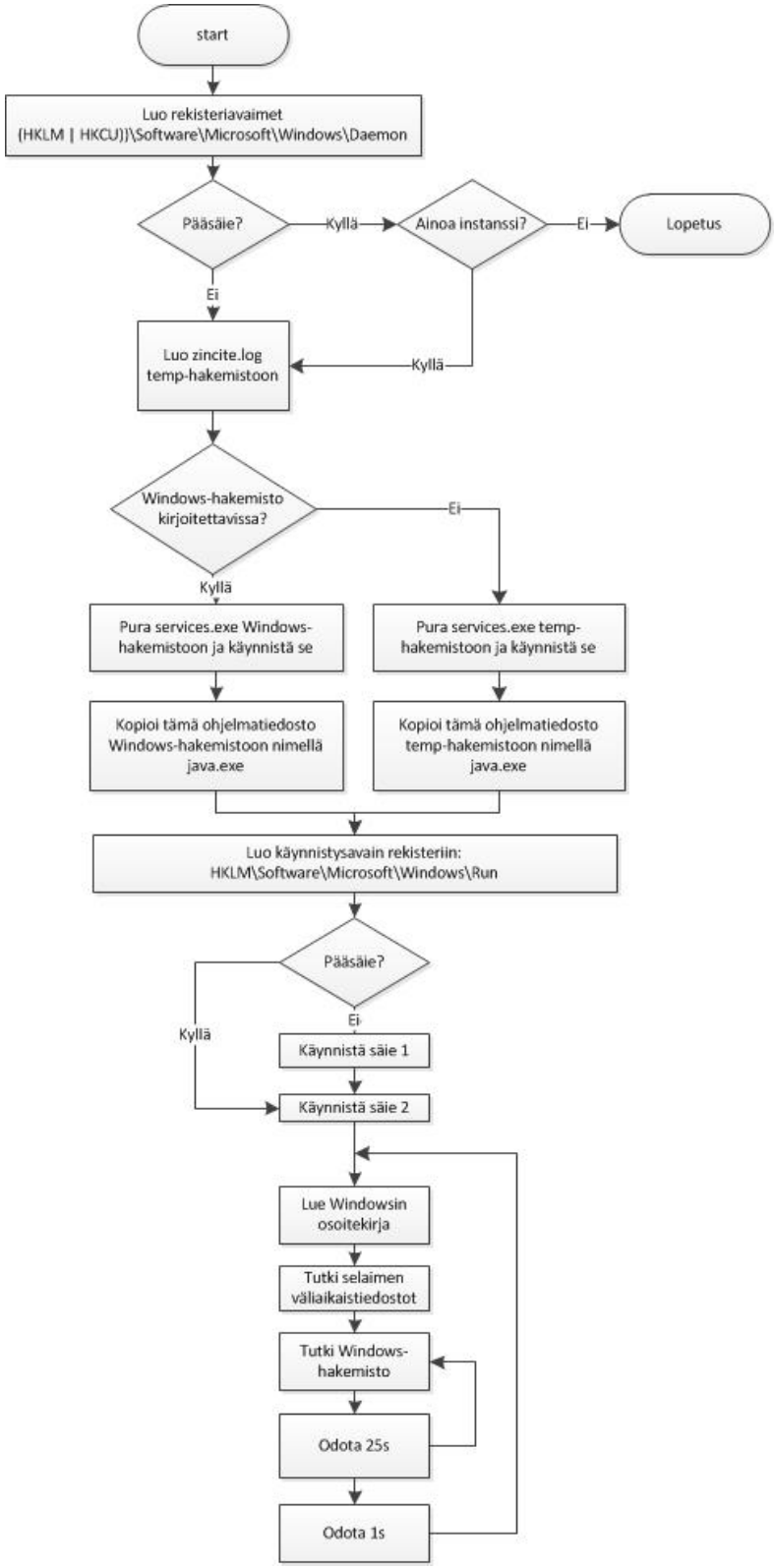
Alustusten jälkeen luodaan kaksi säiettä, joista ensimmäinen vain siinä tapauksessa että nykyinen säie ei ole ohjelman pääsäie. IDA on nimennyt ensimmäisen säikeen pääohjelman nimellä StartAddress ja toisen nimellä sub_504C1B sen ensimmäisen käskyn muistiosoitteen mukaan. Nimeämme aliohjelmat nimellä Thread1Start ja Thread2Start, ja palaamme niihin myöhemmin.

MainProc päättyy silmukkaan, joka jää suorittamaan aliohjelmaa sub_50565B sekunnin välein. Nimeämme tämän aliohjelman nimellä MainLoop. Tämä aliohjelma on vain 11 käskyn mittainen, mutta sisältää paljon funktiokutsuja 49. Se asettaa ensin nykyisen säikeen prioriteetin normaalia alemmaksi [2], kutsuu kahta aliohjelmaa ja siirtyy ikuiseen silmukkaan, jossa se kutsuu edelleen uutta aliohjelmaa 25 sekunnin välein ja kasvattaa globaalia muuttujaa.

Ensimmäinen MainLoop-funktion aliohjelmista ennen silmukkaa lukee rekisteristä arvon ”HKEY_CURRENT_USER\Software\Microsoft\WAB\WAB4\Wab\File Name” ja lukee siitä löytämänsä tiedoston. Google-haku paljastaa avaimen liittyvän Windowsin osoitekirjaan. Jälkimmäinen aliohjelma käy läpi käyttäjän Temporary Internet Files -kansion tiedostoja rekursiivisesti `FindFirstFile`- ja `FindNextFile` -kutsuilla, ja silmukan sisällä oleva aliohjelma tekee saman Windows-hakemistolle. Löydetyt tiedostot prosessoidaan osoitteesta 505131 löytyvällä aliohjelmalla, joka vertaa tiedoston tarkennetta merkki kerrallaan ja kutsuu edelleen aliohjelmia sub_504E00 ja sub_504EEA. Aliohjelma näyttää monimutkaiselta, mutta järjestelemällä lohkoja uudelleen ruudulla (ks. 56) nähdään, että haittaohjelma on kiinnostunut ainakin TXT-, TBB- ja WAB-päätteisistä tiedostoista. Kaksi jälkimmäistä tiedostoa ovat osoitekirjatiedostoja, TBB The Bat! -ohjelmalle ja WAB Windows Outlookille.

Ohjelman pääsäikeen toiminnallisuus on koottu vuokaavioon 25. Tarkastellaan seuraavaksi kahta MainProc-aliohjelman käynnistämää alisäiettä.

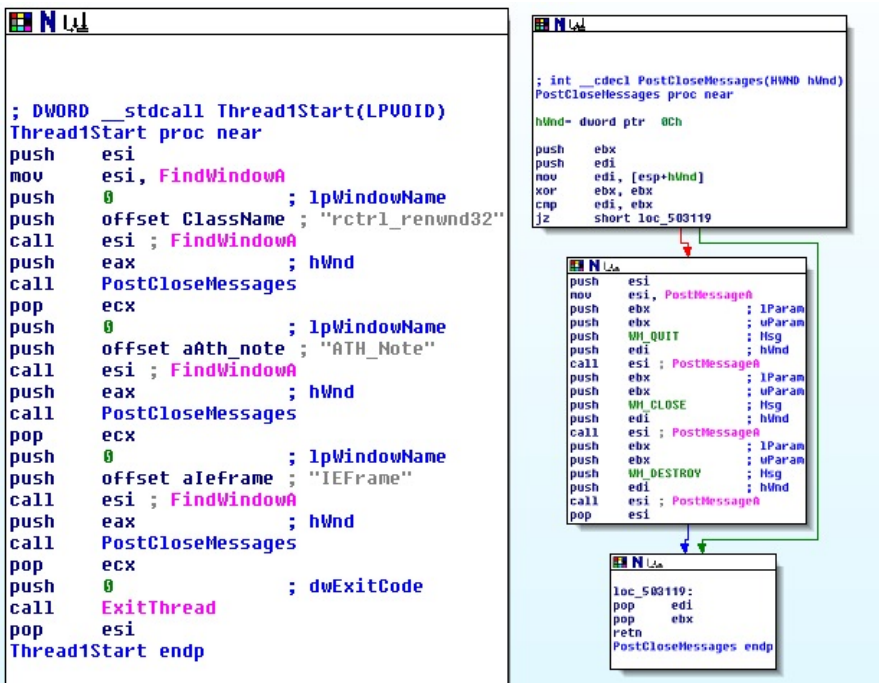
2. IDA ehdottaa ensin lukuarvoa 0FFFFFFFFh, mutta kontekstivalikon ja funktion dokumentaation kautta arvolle löytyy oikea symbolinen vakio.



Kuvio 25. Koodianalyysin perusteella rakennettu vuokaavio MyDoom-haittaohjelman pääsäikeen toiminnasta.

7.6.2 MyDoom-unpacked.exe: säie 1 (sub_50311C)

Säie 1 on hyvin yksinkertainen, kuten kuvasta 26 voi nähdä. Se etsii ikkunoita tietyillä ikkunaluokilla, jotka se yrittää sulkea lähettämällä niille viestit `WM_QUIT`, `WM_CLOSE` ja `WM_DESTROY`. Ikkunaluokat ovat ”rctrl_renwnd32”, ”ATH_Note” ja ”IEFrame”, joiden verkkohaku paljastaa kuuluvan Microsoft Outlookille, Outlook Expressille ja Internet Explorerille. Funktion tarkoitus jää vielä tässä vaiheessa hämärän peittoon, kuten myös sen käynnistys – debuggerilla ajettuna säiettä ei koskaan käynnistetä, ja myös sen aloitusehto (haittaohjelman pääohjelma `start` käynnistetään säikeestä joka ei ole ohjelman pääsäie) tuntuu oudolta, sillä ohjelma ei tee `CreateThread`-kutsuja aliohjelma `start` parametrina. Tästä kaikesta voidaan päätellä, että säie on todennäköisesti mukana ohjelmassa joko hämäyksenä tai jäänteenä haittaohjelman toisesta variantista.



Kuvio 26. Säikeen 1 pääohjelma Thread1Start (sub_50311C) ja sen kutsuma aliohjelma. Kuvakaappaus IDA Pro -disassemblerista.

7.6.3 MyDoom-unpacked.exe: säie 2 (sub_504C1B)

Osoitteesta 504C1B alkava säie sisältää aliohjelmakutsun, säikeen lopetuskutsun ja ansan debuggerille keskeytyksellä 3. Aliohjelman alusta paljastuu ajonaikaisesti linkitetty kutsu `In-`

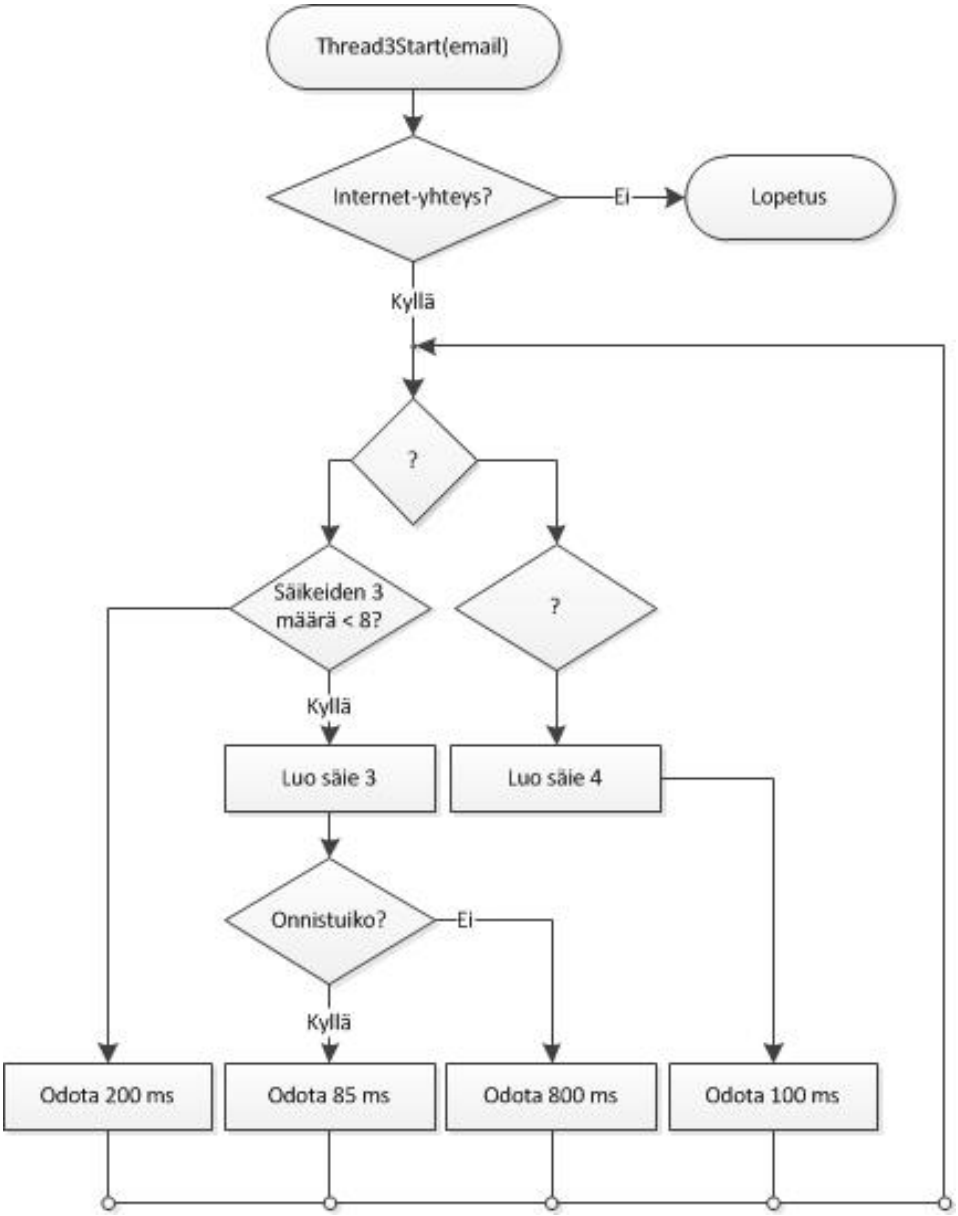
ternetGetConnectedState-funktioon kirjastoon wininet.dll. Debuggaus paljastaa, että koska verkkoyhteyttä ei tunnisteta, ohitetaan suuri määrä muuten suoritettavaa toiminnallisuutta 50. Tämän vuoksi verkkosimulaattorissa ei näkynyt mitään mielenkiintoista. Hyppykäskyn ”jz short loc_504AF2” tavut korvaamalla operaatiokoodeilla 90 (no operation) voidaan efektiivisesti poistaa hyppy ja saada ohjelma käyttäytymään niin kuin se olisi tunnistanut verkkoyhteyden 51. Debuggerissa voidaan myös tehdä sama väliaikaisesti kääntämällä vertailun jälkeen prosessorin Z- eli nollalippu arvoon 0.

Tämä säie sisältää runsaasti kutsuja GetTickCount-aliohjelmaan ja globaaleihin muuttujiin, mikä viittaa ajastettuun ja / tai satunnaiseen toiminnallisuuteen. Kutsuja ei esiinny pareittain, joten niitä ei luultavasti ole tarkoitettu debuggausta vastaan. Säikeen pääfunktio Thread2Main päättyy Sleep-kutsuun, joka jää odottamaan 1–800 millisekunnin ajaksi ja palaa takaisin funktion alkuun.

Thread2Main-funktio luo edelleen kahden eri tyyppisiä säikeitä CreateThread-kutsuilla, joista toinen nähdään kuvassa 52 keskellä ohjelmakoodia (olkoon tämä säie 3), ja toinen (säie 4) oikeassa alakulmassa olevan CreateThread4-aliohjelman sisällä. Säie 3 luodaan, jos globaali muuttuja Addend on alle 8. Säikeen pääohjelmassa (Thread3Start, ks. kuva 53) nähdään, että muuttujaa kasvatetaan InterlockedIncrement-kutsulla sen alussa ja vähennetään InterlockedDecrement-kutsulla. Voidaan päätellä, että säikeet ovat jonkinlaisia työsäikeitä, joita on käynnissä enintään seitsemän kerrallaan.

CreateThread-kutsua tarkastelemalla nähdään myös, että sille annetaan rekisterin esi arvo parametrinä. Debuggeri paljastaa rekisterin sisällöksi sähköpostiosoitteen. Vastaavasti säie 4 ottaa parametrikseen (mahdollisesti säikeen 3 sähköpostiosoitteesta erotteleman) domainnimen. Säikeen 4 luontia joutuu odottelemaan debuggerissa minuutista kolmeen, mutta sen jälkeen niitä luodaan nopeaan tahtiin. Säikeet ovat lyhytikäisiä, eikä niillä ole laskuria kuten säikeellä 3.

Keskeytyskohdilla (ks. kuva 57) tehty mittaus paljastaa säikeitä 4 luotavan 360 kertaa minuutin aikana, eli 6 kertaa sekunnissa. Säikeitä 3 luodaan kaksi ohjelman käynnistyessä ja lisää kun säikeiden 4 luonti alkaa. Säikeiden 3 elinikä vaihtelee hyvin lyhyestä hyvin pitkään. Säikeen 2 toiminnallisuus on koottu vuokaavioon 27.



Kuvio 27. Koodianalyysin perusteella rakennettu vuokaavio MyDoom-haittaohjelman säikeen 2 toiminnasta. Kysymysmerkeillä merkityt ehdot ovat globaaleista muuttujista ja GetTickCount-kutsujen tuloksista laskettuja kompleksisia ehtoja.

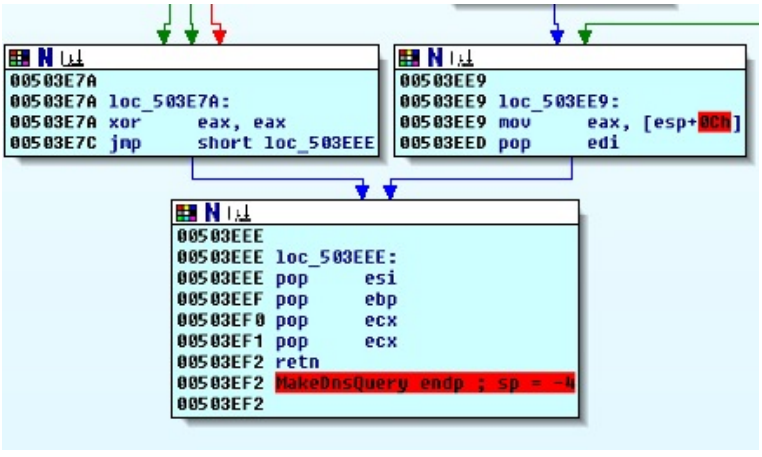
7.6.4 MyDoom-unpacked.exe: säie 3 (sub_504A37)

Säikeen 3 pääohjelma `Thread3Main` alkaa silmukalla, jossa erotellaan domain-nimi parametrina annetusta sähköpostiosoitteesta. Tämän jälkeen kutsutaan aliohjelmaa `sub_504971` domain-nimi parametrina, jonka tuloksen perusteella kutsutaan edelleen kahta muuta

aliohjelmaa. Debuggaamalla nähdään, että toinen aliohjelma ottaa parametrikseen sähköpostiosoitteen ja palauttaa alkuosan sähköpostiviestistä SMTP-protokollan mukaisena (kuva 58). Viesti sisältää otsikkotiedot, mutta ei vielä varsinaista tekstiä.

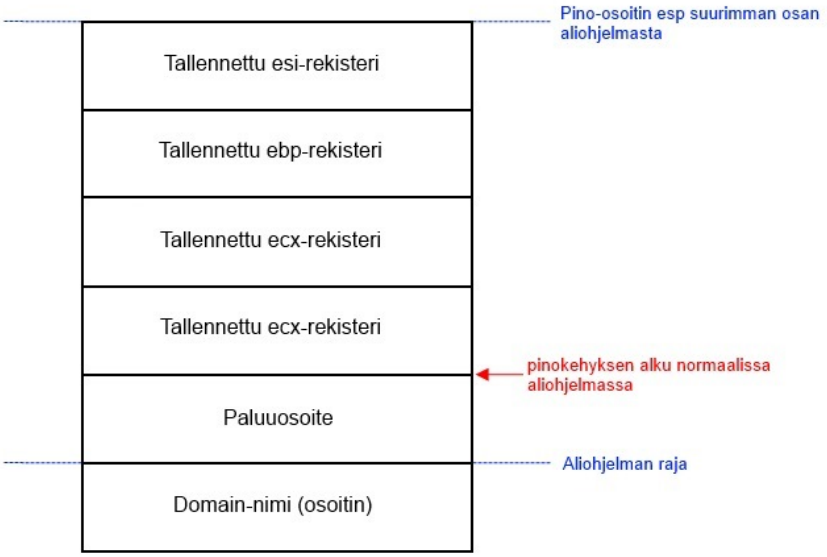
Kolmas aliohjelma ottaa parametrikseen domain-nimen ja osittaiset otsikkotiedot. Fakenet-verkkosimulaattorista nähdään, että aliohjelma on koonnut viestin ja lähettänyt sen. Simulaattori tallentaa kaappaamaansa liikennettä pcap-tiedostoon, joka voidaan avata esimerkiksi Wiresharkilla. Sähköpostin vastaanottajat ovat osoitteita, joita haittaohjelma on poiminut osoitekirjasta ja levyltä löytyvistä txt- ja html-tiedostoista. Sähköposti sisältää liitetiedoston base64-koodattuna, joka purettuna paljastuu samankokoiseksi kuin haittaohjelma (MyDoom-unpacked.exe). MD5-summa on kuitenkin eri, mikä viittaa polymorfismiin. Sähköpostin otsikko ja liitetiedoston nimi vaihtelevat jokaisen viestin välillä.

Aliohjelmista ensimmäinen, `sub_504971`, tekee DNS-kyselyn sille annettuun domain-nimeen. Kyselyn tekevässä `MakeDnsQuery`-funktiossa nähdään ensimmäisen kerran IDA:n automaattisen pinoanalyysin epäonnistuminen. Pino näyttäisi sisältävän kolme paikallista muuttujaa, mutta niitä ei koskaan käytetä funktiossa. Lisäksi IDA on merkinnyt punaisella viittauksia, joissa näytettäisiin mentävän pinokehyksen ulkopuolelle, ja funktion viimeisen rivin kommentilla sp = -4, joka viittaisi siihen, että pinosta poistetaan 4 tavua enemmän kuin pitäisi (ks. kuva 28). Debuggerilla nähdään, että näin ei kuitenkaan todellisuudessa käy.



Kuvio 28. Pinoanalyysin epäonnistuminen funktiossa MakeDnsQuery. Kuvakaappaus IDA Pro -disassemblerista.

Aliohjelmaa tarkastelemalla huomataan, se ei tee normaalia pinokehystä, vaan se viittaa parametriin ja tallennettuihin rekisterien arvoihin suoraan pino-osoittimeen lisäämällä. Funktion pino näyttää kuvan 29 mukaiselta. Rekisteriä ebp käytetään kuten mitä tahansa datarekisteriä. Myös funktion lokaalit muuttujat on tunnistettu virheellisesti, ohjelmassa niihin ei viitata lainkaan. Ne voidaan poistaa käsin funktionmuokkausikkunasta (Edit function) asettamalla lokaalien muuttujien alueen kooksi nolla. Funktiolle voitaisiin tehdä myös muita muokkauksia, kuten rakentaa sille käsin perinteinen pinokehys kirjoittamalla sen assemblykoodi uudestaan, mutta tämän analyysin kannalta se ei ole tarpeellista.



Kuvio 29. Pinon rakenne MakeDnsQuery-funktiossa. Perinteisen pinokehyksen sijaan pinoa käytetään funktiossa suoraan pino-osoittimeen lisäämällä.

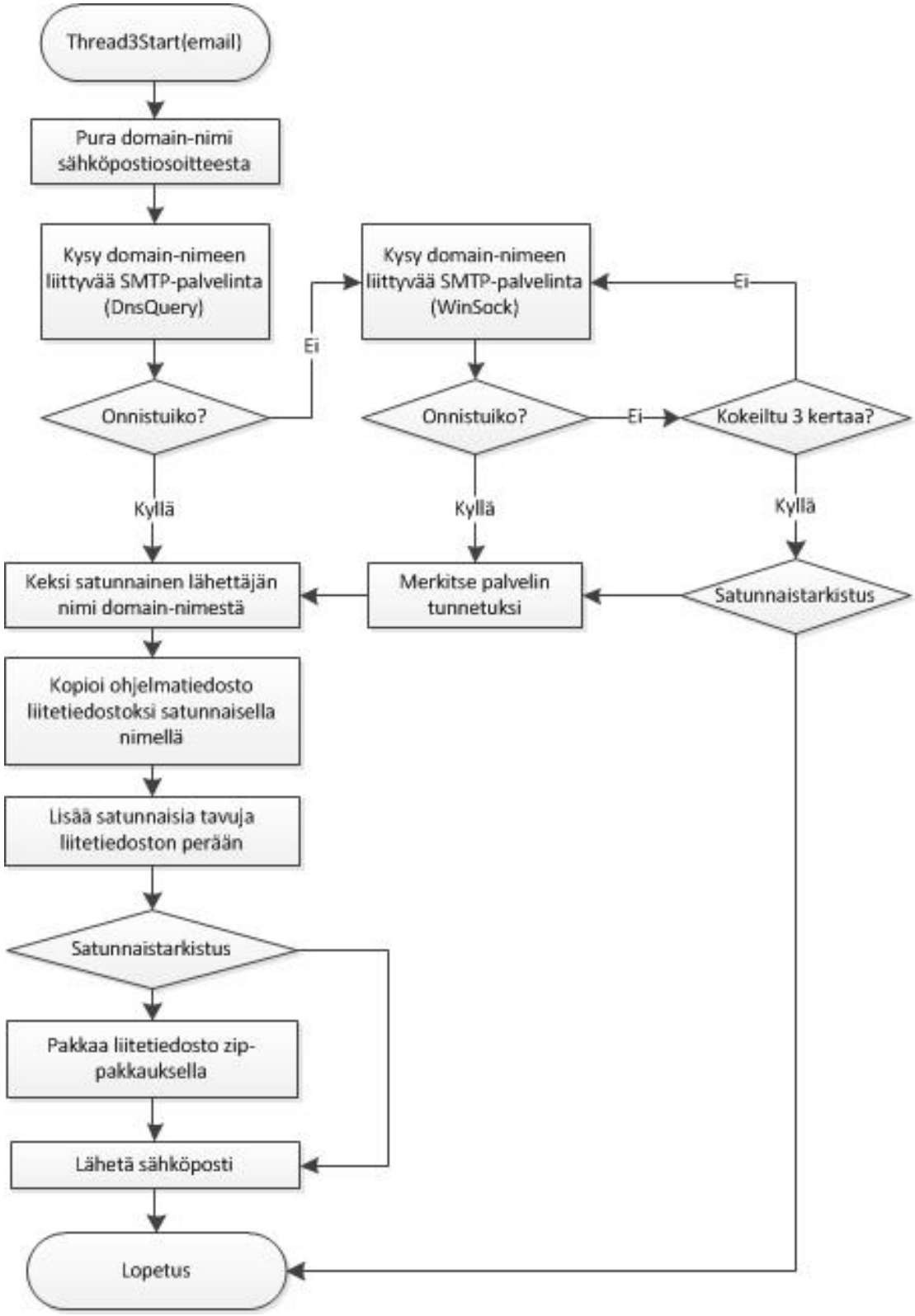
Jos ensimmäinen DNS-kysely dnsapi-kirjaston `DnsQuery`-funktiolla ei onnistu, jatketaan hyppykäskyllä aliohjelmaan `sub_503EF3`. Koska molemmilla aliohjelmilla on samat parametrit ja sama kutsukäytäntö, kutsu hypyllä on laillinen. Tämä aliohjelma kokeilee vielä DNS-kyselyä sokettirajapinnan kautta. Pakettianalyysillä nähdään, että kaikki haittaohjelman tekemät DNS-kyselyt ovat MX-tyyppisiä, eli kyselyitä domain-nimeen liittyvästä SMTP-palvelimesta. Kun se löytää toimivan SMTP-palvelimen, se tallentaa sen globaaliin muuttujaan eikä tee enää uusia kyselyitä. Fakenetin simuloima SMTP-palvelin ei läpäise kaikkia tarkistuksia, mutta siitä huolimatta se saatetaan hyväksyä satunnaisesti, tarkalleen

jos järjestelmän käynnissäoloajan vähiten merkitsevä tavu on 3. Analyysin helpottamiseksi tämän osoitteessa 50499C olevan hyppykäskyn (tai sitä edellisen) voi muuttaa ehdottomaksi tai kääntää nollalipun debuggerissa, jolloin palvelin hyväksytään joka kerta.

Tarkastellaan seuraavaksi sähköpostin luovaa aliohjelmaa. Se kutsuu lukuisia funktioita, joilla se rakentaa sähköpostiviestin. API-funktiokutsuja silmäilemällä nähdään, että liitetiedoston lisääminen tapahtuu funktiossa `sub_505FAF`. Funktiossa kopioidaan ensin moduulin tiedosto käyttäjän temp-hakemistoon. Sille arvotaan satunnainen tiedostonnimi kahdella tarkenteella, joista sisempi on html, htm, txt tai doc, ja ulompi pif, scr, exe tai com. Näin tiedosto saadaan näyttämään vaarattomalta dokumentilta jos tarkenteiden näyttö on pois päältä. Tiedosto saatetaan satunnaisesti myös pakata zip-pakkauksella.

”Polymorfismi” on toteutettu funktiossa `sub_605EE4` yksinkertaisesti avaamalla tiedosto `zincite.log` ja kirjoittamalla sen tavut satunnaisessa järjestyksessä ohjelman perään. Sähköpostin lähettävässä aliohjelmassa ei ole juurikaan erityistä mainittavaa. Se kokeilee smtp-palvelimelle ”mail”-, ”smtp”- ja ”mx”-alidomainnimiä, yhdistää siihen sokettirajapinnan kautta ja lähettää viestin.

Säikeen 3 toiminnallisuus on koottu vuokaavioon 30.



Kuvio 30. Koodianalyysin perusteella rakennettu vuokaavio MyDoom-haittaohjelman säikeen 3 toiminnasta.

7.6.5 MyDoom-unpacked.exe: säie 4 (sub_50477F)

Säikeen 4 käynnistyminen riipppuu useiden globaalien muuttujien arvoista ja satunnaisuudesta, joka on johdettu ajasta joka on kulunut tietokoneen käynnistyksestä. Eräänä tekijänä näyttäisi olevan myös päivämäärä, jonka apuprosessi ”services.exe” palauttaa alkuperäiseen arvoonsa, jos sitä yritetään muuttaa. Suorituksen ohjaaminen säikeessä 2 debuggerissa lippurekisteriä muuttamalla aiheuttaa ohjelman kaatumisen virheelliseen muistinosoitukseen, mutta rekisterin `ebx` muuttaminen arvosta 2 arvoon 3 päästää tarkistuksesta läpi aiheuttamatta vastaavaa ongelmaa.

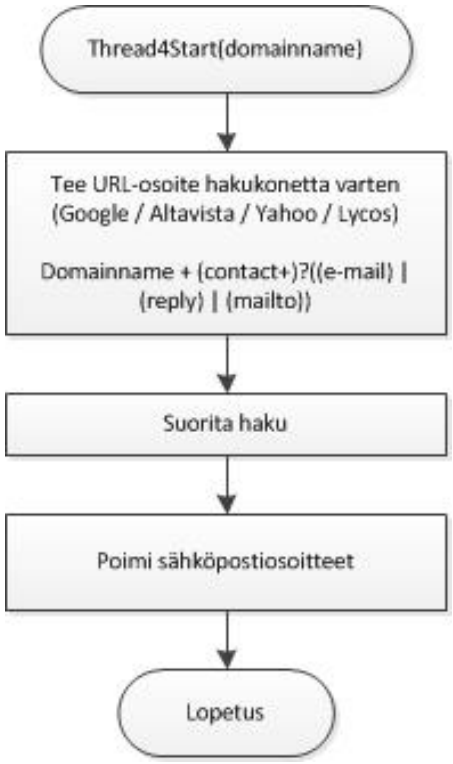
Säie 4 saa parametrikseen domain-nimen ja sisältää kolme aliohjelmakutsua. Ensimmäinen niistä rakentaa domain-nimestä ja satunnaisista hakusanoista URL-osoitteen Google-, Altavista-, Yahoo-, tai Lycos-hakua varten. Hakusanat valitaan satunnaisesti sanoista ”e-mail”, ”reply” ja ”mailto”, jonka lisäksi merkkijonossa voi esiintyä sana ”contact”. Palautettavien tulosten määrä on rajoitettu sataan.

Toinen aliohjelmista kutsuu API-funktiota `URLDownloadToCacheFile` suorittamaan ensimmäisen aliohjelman rakentaman hakumerkkijonon. Funktio tallentaa tiedoston selaimen väliaikaistiedostoihin. Lopuksi kolmas aliohjelma avaa tiedoston, lukee sen muistiin ja käsittelee sen kahden aliohjelma `sub_504C29` ja `sub_504D0C` avulla. Debuggerilla muistialuetta tarkastelemalla nähdään, että ensimmäinen niistä poistaa siitä rivinvaihdot ja muut ohjausmerkit. Jälkimmäinen aliohjelma sisältää paljon sisäkkäisiä silmukoita, joista ensimmäisessä etsitään merkkiä ’@’. Tämän jälkeen merkkejä ei verratakaan enää suoraan asciiarvoina. Funktiota on helpompi analysoida dynaamisesti, mutta se vaatii hieman ylimääräistä työtä.

Fakenet-simulaattori antaa oletuksena sen *defaultFiles*-hakemistossaan olevan *FakeNet.html* -tiedoston vastauksena kaikkiin http-kyselyihin. Haittaohjelma olettaa saavansa sivullisen hakutuloksia, jotka voimme sille antaa tekemällä ensin haun isäntäkoneella, ja tallentamalla saadun html-sivun Fakenetin *defaultFiles*-hakemistoon. Näin tehtyämme voimme asettaa keskeytyskohdan silmukan keskellä olevaan aliohjelmakutsuun, ja kuten voimme olettaa, funktion parametrinä on sivulta luettu sähköpostiosoite. Toinen parametri on 0, jonka voimme olettaa tarkoittavan osoitemerkkijonon ensimmäisen merkin indeksiä. IDA:n *Xrefs to* -

toimintoa funktioon käyttämällä huomaamme myös, että funktio on sama, jota esimerkiksi ohjelman pääsäie käyttää lukiessaan tiedostoja, eli osoite lisätään samaan listaan.

Säikeen 4 toiminnallisuus yksinkertaistuu varsin suoraviivaiseen muotoon, mutta se on silti esitetty täydellisyyden vuoksi vuokaaviona kuvassa 31.



Kuvio 31. Koodianalyysin perusteella rakennettu vuokaavio MyDoom-haittaohjelman säikeen 4 toiminnasta.

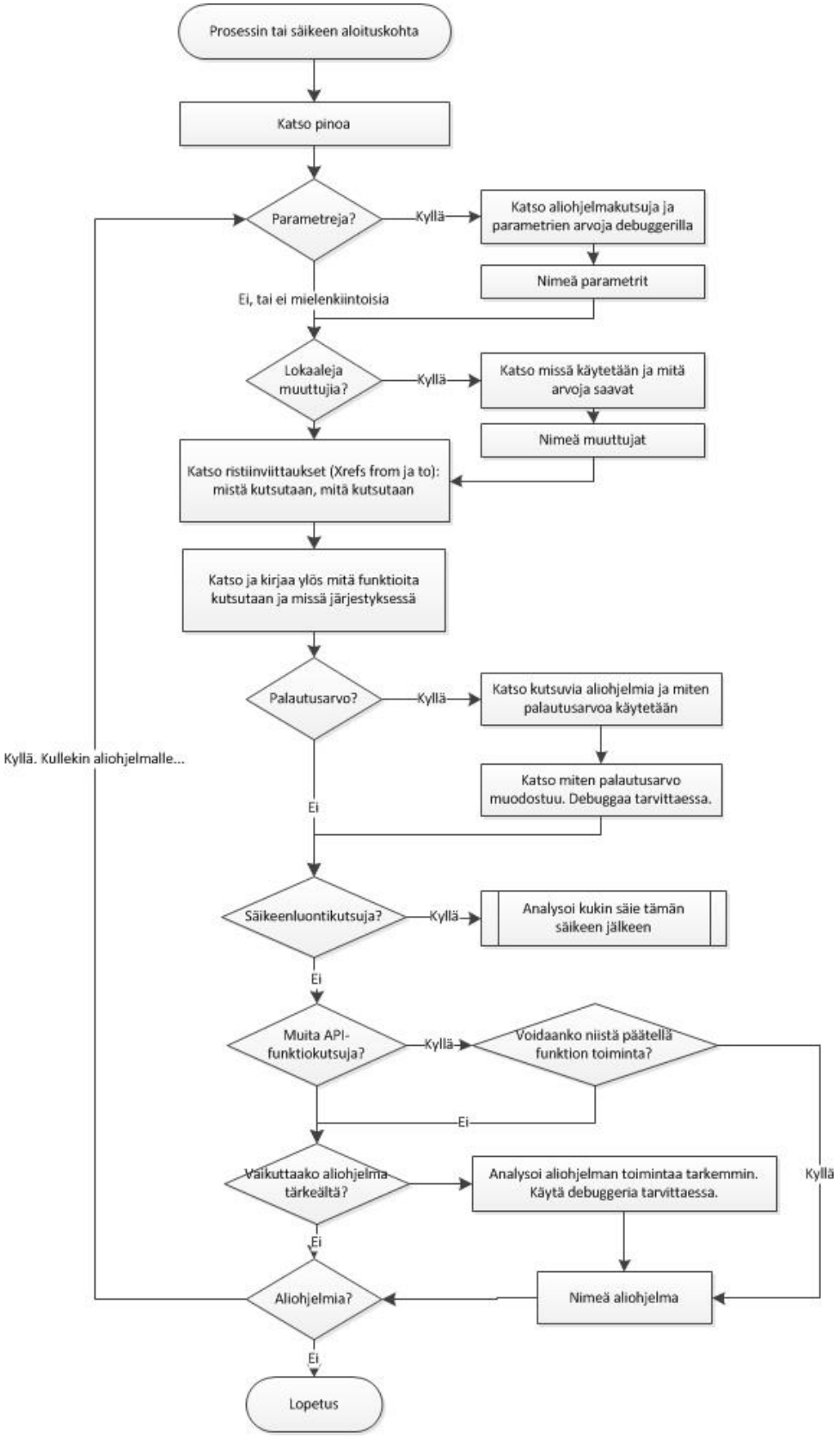
7.6.6 Tutkimuksen tulokset ja päätelmät

Haittaohjelman analyysi etenkin koodianalyysivaiheessa on aikaavievää, mutta tarpeeksi tutkimalla disassemblerilla ja debuggerilla saadaan selvitettyä ohjelman toiminta täydellisesti. Vastatoimet, kuten obfuskaatio ja erikoinen pakkausmenetelmä lisäävät huomattavasti analyysin vaikeusastetta, ja tässä työssä käytetty haittaohjelma olikin valittu niin, että se sisälsi niitä sopivan määrän tutkielman laajuuteen nähden. Haittaohjelmasta jäi vielä tutkimatta komponentti ”services.exe”, mikä jäi pois ajan puutteen ja tutkimuksen laajuuden vuoksi. Toisaalta menetelmät sen tutkimiseen ovat samoja, joten se ei olisi tuonut juurikaan uusia

näkökulmia.

Tärkeimpinä tuloksina tästä tutkimuksesta saatiin vuokaaviot ohjelman säikeistä, jotka ovat nähtävissä kunkin alaluvun lopussa. Vuokaaviot näyttävät varsin yksinkertaisilta, mutta niiden rakentamiseksi on täytynyt käydä läpi kymmeniä aliohjelmia disassemblerissa ja debuggerissa. Vuokaavioihin voitaisiin lisätä vielä kuva 32, joka kuvaa itse koodianalyysiprosessin, tai millaiseksi se omalla kohdallani jalostui tutkimuksen myötä.

IDA Pro Free 5.0 -ohjelmasta ja sen automaattisesta funktioanalyysistä oli suuri apu tutkimuksessa. Automatiikan lisäksi sen interaktiivisuus tuli myös tarpeeseen – erityisesti funktioiden ja muuttujien nimeäminen, koodirivien kommentointi ja symbolisten vakioiden asettaminen lukuarvoille lisäsivät ohjelman luettavuutta merkittävästi.



Kuvio 32. Vuokaavio koodianalyysin kulusta säie kerrallaan.

8 Yhteenveto

Reverse engineering on laaja ja hyvin tekninen aihealue, ja siitä on vaikeaa kirjoittaa lyhyesti. Mielestäni onnistuin kuitenkin kiteyttämään tähän tutkielmaan oleellisimman osan sen soveltamiseen haittaohjelmien analyysiin. Sikorskin ym. kirja ”Practical Malware Analysis” (Sikorski, Honig ja Lawler 2012) ja Eilamin ”Secrets of Reverse Engineering” (Eilam 2011) olivat tutkimuksessa korvaamattomia lähteitä ja voin suositella niitä kaikille aiheesta kiinnostuneille.

Haittaohjelmien analyysi on aikaavievää varsinkin koodianalyysivaiheessa, ja monesti täydellinen analyysi ei ole tarpeen ellei sitä nimenomaan vaadita. Osittainenkin analyysi voi kuitenkin kertoa paljon ohjelman toiminnasta ja alkuperästä, ja voi olla hyökkäyksen tapahtuessa ainoa varteenotettava vaihtoehto. Siksi ainakin perustaidot binäärianalyysistä tulisi kuulua jokaisen tietoturva-asiantuntijan työkalupakkiin.

Tutkimuksen empiirinen osa paljastui yllättävän aikaavieväksi, vaikka haittaohjelma oli jo hieman ennestään tuttu eikä sisältänyt merkittäviä vastatoimia verrattuna moniin nykypäivän merkittävimpiin haittaohjelmiin. Siitä huolimatta, tai siitä johtuen, se antoi paljon uutta käytännön tietoa ja kokemusta haittaohjelman tutkimisesta, x86-arkkitehtuurista ja binäärianalyysistä yleensä.

Valmiit ohjelmat, kuten IDA Pro ja verkkosimulaattori Fakenet helpottavat ja nopeuttavat analyysiä huomattavasti, mutta kehitykselle on vieläkin tilaa. Eniten jäin kaipaamaan IDA:lta funktiograafia, joka kuvaisi myös funktioiden suoritusjärjestyksen. Funktiokutsujen yhteydessä myös parametrien ja paluuarvojen välityksen voisi kuvata selkeämmin. IDA on kuitenkin jatkuvasti kehittyvä tuote, joka on myös laajennettavissa lisäosilla, joten tulevaisuudessa luultavasti myös nämä puutteet tulevat korjaantumaan.

Lähteet

Abu Rajab, Moheeb, Jay Zarfoss, Fabian Monrose ja Andreas Terzis. 2006. “A multifaceted approach to understanding the botnet phenomenon”. Teoksessa *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement,* 41–52. ACM.

Allen, Frances E. 1970. “Control flow analysis”. Teoksessa *ACM Sigplan Notices,* 5:1–19. 7. ACM.

“Anubis: Analyzing Unknown Binaries”. 2013. Viitattu 18. marraskuuta. `http://anubis.iseclab.org`.

Bayer, Ulrich, Imam Habibi, Davide Balzarotti, Engin Kirda ja Christopher Kruegel. 2009. “A view on current malware behaviors”. Teoksessa *USENIX workshop on large-scale exploits and emergent threats (LEET).*

Bayer, Ulrich, Christopher Kruegel ja Engin Kirda. 2006. “TTAnalyze: A tool for analyzing malware”. Teoksessa *15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference.*

Bellard, Fabrice. 2013. “Qemu - open source processor emulator”. Viitattu 18. marraskuuta. `http://wiki.qemu.org`.

Bergeron, Jean, Mourad Debbabi, Jules Desharnais, Mourad M Erhioui, Yvan Lavoie ja Nadia Tawbi. 2001. “Static detection of malicious code in executable programs”. *Int. J. of Req. Eng* 2001:184–189.

Bridges, Lloyd. 2008. “The changing face of malware”. *Network Security* 2008 (1): 17–20.

Bruschi, Danilo, Lorenzo Martignoni ja Mattia Monga. 2006. “Detecting self-mutating malware using control-flow graph matching”. Teoksessa *Detection of Intrusions and Malware & Vulnerability Assessment,* 129–143. Springer.

Buehler, Marianne, Kelly Socia, Kevin McCarthy, Michael Kozak, Eric Walter, Paul Lepkowski, Cator Daniel, James Lippard, Samuel McQuade, Neel Sampat et al. 2009. “Encyclopedia of Cybercrime”.

Chien, Eric. 2002. “Symantec Security Response: VBS.LoveLetter.Var”. Viitattu 18. marraskuuta 2013. http://www.symantec.com/security_response/writeup.jsp?docid=2000-121815-2258-99.

Choi, Yang-seo, Ik-kyun Kim, Jin-tae Oh ja Jae-cheol Ryou. 2008. “Pe file header analysis-based packed pe file detection technique (phad)”. Teoksessa *Computer Science and its Applications, 2008. CSA'08. International Symposium on,* 28–31. IEEE.

Chow, Kam-Pui, Frank YW Law, Michael YK Kwan ja Pierre KY Lai. 2007. “The rules of time on NTFS file system”. Teoksessa *Systematic Approaches to Digital Forensic Engineering, 2007. SADFE 2007. Second International Workshop on,* 71–85. IEEE.

Christodorescu, Mihai, Somesh Jha, Sanjit A Seshia, Dawn Song ja Randal E Bryant. 2005. “Semantics-aware malware detection”. Teoksessa *Security and Privacy, 2005 IEEE Symposium on,* 32–46. IEEE.

“CWSandbox: Behavior-based Malware Analysis”. 2013. Viitattu 19. marraskuuta. mwanalysis.org.

Dang, Bruce, Alexandre Gazet ja Elias Bachaalany. 2014. *Practical Reverse Engineering.* John Wiley & Sons Inc.

Dinaburg, Artem, Paul Royal, Monirul Sharif ja Wenke Lee. 2008. “Ether: malware analysis via hardware virtualization extensions”. Teoksessa *Proceedings of the 15th ACM conference on Computer and communications security,* 51–62. ACM.

Distler, Dennis. 2007. *Malware Analysis: An Introduction.* SANS Institute. http://www.sans.org/reading-room/whitepapers/malicious/malware-analysis-introduction-2103.

Egele, Manuel, Theodoor Scholte, Engin Kirda ja Christopher Kruegel. 2012. “A survey on automated dynamic malware-analysis techniques and tools”. *ACM Computing Surveys (CSUR)* 44 (2): 6.

Eilam, Eldad. 2011. *Reversing: Secrets of reverse engineering.* Wiley. com.

“Ether: Malware Analysis via Hardware Virtualization Extensions”. 2013. Viitattu 19. marraskuuta. http://ether.gtisc.gatech.edu.

Han, Seungwon, Keungi Lee ja Sangjin Lee. 2009. “Packed PE File Detection for Malware Forensics”. Teoksessa *Computer Science and its Applications, 2009. CSA’09. 2nd International Conference on,* 1–7. IEEE.

Hart, Johnson M. 2010. *Windows system programming*. 4. painos. Pearson Education.

Hutcheson, Lorna. 2006. “Malware Analysis The Basics”. *CACI International y Inc.*

“Intel 64 and IA-32 Architectures Software Developers Manual”. 2013. Viitattu 1. joulukuuta. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

Jakobsson, Markus. 2012. *The Death of the Internet.* Wiley Publishing.

Kendall, Kris, ja Chad McMillan. 2007. “Practical malware analysis”. Teoksessa *Black Hat Conference, USA.*

Ligh, Michael, Steven Adair, Blake Hartstein ja Matthew Richard. 2010. *Malware Analyst’s Cookbook and DVD: Tools and Techniques for Fighting Malicious Code.* Wiley Publishing.

Liu, Yana. 2004. “Symantec Security Response: W32.MyDoom.F@mm”. Viitattu 18. marraskuuta 2013. `http://www.symantec.com/security_response/writeup.jsp?docid=2004-022011-2447-99`.

Lyda, Robert, ja James Hamrock. 2007. “Using entropy analysis to find encrypted and packed malware”. *IEEE Security & Privacy* 5 (2): 40–45.

Mavrommatis, Niels Provos Panayiotis, ja Moheeb Abu Rajab Fabian Monrose. 2008. “All your iframes point to us”.

“Microsoft Developer Network: Windows Dev Center”. 2014. Viitattu 9. toukokuuta 2014. `http://msdn.microsoft.com/en-us/windows/`.

“Microsoft PE and COFF Specification”. 2013. Viitattu 18. marraskuuta 2013. `http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx`.

Moir, Robert. 2003. “Defining Malware: FAQ”. Viitattu 18. marraskuuta 2013. `http://technet.microsoft.com/en-us/library/dd632948.aspx`.

Moser, Andreas, Christopher Kruegel ja Engin Kirda. 2007. “Exploring multiple execution paths for malware analysis”. Teoksessa *Security and Privacy, 2007. SP'07. IEEE Symposium on,* 231–245. IEEE.

Narvaez, Julia, Barbara Endicott-Popovsky, Christian Seifert, Chiraag Aval ja Deborah A Frincke. 2010. “Drive-by-downloads”. Teoksessa *System Sciences (HICSS), 2010 43rd Hawaii International Conference on,* 1–10. IEEE.

O'Kane, Philip, Sakir Sezer ja Kieran McLaughlin. 2011. “Obfuscation: The hidden malware”. *Security & Privacy, IEEE* 9 (5): 41–47.

“OSDev.org Wiki: Interrupts”. 2014. Viitattu 2. toukokuuta. `http://wiki.osdev.org/Interrupts`.

“PC World: Virus Writers Wage Worm War”. 2013. Viitattu 16. joulukuuta. `http://www.pcworld.com/article/115076/article.html`.

Perdisci, Roberto, Andrea Lanzi ja Wenke Lee. 2008. “Classification of packed executables for accurate computer virus detection”. *Pattern Recognition Letters* 29 (14): 1941–1946.

Petzold, Charles. 1998. *Programming Windows®*. 5. painos. O'Reilly.

Quist, Danny, ja Val Smith. 2006s. “Detecting the Presence of Virtual Machines Using the Local Data Table”. *Offensive Computing*. Viitattu 12. toukokuuta 2014. `http://www.offensivecomputing.net/files/active/0/vm.pdf`.

Raffetseder, Thomas, Christopher Krügel ja Engin Kirda. 2007. “Detecting system emulators”. Teoksessa *Information Security,* 1–18. Springer.

Reeves, Ronald D. 2010. *Windows 7 Device Driver*. Pearson Education.

Richter, Jeffrey M. 1999. *Programming Applications for Microsoft Windows*. 4. painos. Microsoft Press, Redmond, WA.

Rutkowska, Joanna. 2004. “Red pill... or how to detect VMM using (almost) one CPU instruction”. Viitattu 12. toukokuuta 2014. `http://www.hackerzvoice.net/ouah/Red_%20Pill.html`.

Schiffman, Mike. 2014a. “A Brief History of Malware Obfuscation: Part 1 of 2”. Viitattu 3. maaliskuuta. http://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_1_of_2/.

———. 2014b. “A Brief History of Malware Obfuscation: Part 2 of 2”. Viitattu 3. maaliskuuta. http://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_2_of_2/.

Schwarz, Benjamin, Saumya Debray ja Gregory Andrews. 2002. “Disassembly of executable code revisited”. Teoksessa *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on,* 45–54. IEEE.

“Security firm: MyDoom worm fastest yet”. 2014. Viitattu 13. huhtikuuta. http://edition.cnn.com/2004/TECH/internet/01/28/mydoom.spreadwed/.

Shafiq, M Zubair, S Tabish ja Muddassar Farooq. 2009. “PE-probe: leveraging packer detection and structural information to detect malicious portable executables”. Teoksessa *Proceedings of the Virus Bulletin Conference (VB),* 29–33.

Sikorski, Michael, Andrew Honig ja Stephen Lawler. 2012. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software.* No Starch Press.

“Stack Overflow: Compile a C++ program with only dependency on kernel32.dll and user32.dll”. 2014. Viitattu 31. tammikuuta. http://stackoverflow.com/questions/4786236/compile-a-c-program-with-only-dependency-on-kernel32-dll-and-user32-dll.

“Stack Overflow: What functions does WinMainCRTStartup perform”. 2014. Viitattu 31. tammikuuta. http://stackoverflow.com/questions/1583193/what-functions-does-winmaincrtstartup-perform.

Tanachaiwiwat, Sapon, ja Ahmed Helmy. 2006. “VACCINE: War of the worms in wired and wireless networks”. Teoksessa *IEEE INFOCOM,* 05–859.

Turner, Dean, Stephen Entwisle, O Friedrichs, D Hanson, M Fossi, D Ahmad, S Gordon, P Szor ja E Chien. 2013. “Symantec Internet security threat report”. *2012 Trends* 18.

"Understand and Control Startup Apps with the System Configuration Utility". 2009. Viitattu 10. toukokuuta 2014. http://technet.microsoft.com/en-us/magazine/ee851671.aspx.

Willems, Carsten, Thorsten Holz ja Felix Freiling. 2007. "Toward automated dynamic malware analysis using CWSandbox". *Security & Privacy, IEEE* 5 (2): 32–39.

"x86 Assembly Guide". 2013. Viitattu 10. joulukuuta. http://www.cs.virginia.edu/~evans/cs216/guides/x86.html.

Yan, Wei, Zheng Zhang ja Nirwan Ansari. 2008. "Revealing packed malware". *Security & Privacy, IEEE* 6 (5): 65–69.

You, Ilsun, ja Kangbin Yim. 2010. "Malware Obfuscation Techniques: A Brief Survey." Teoksessa *BWCCA,* 297–300.

Zeltser, Lenny. 2010. "Analyzing Malicious Software". Teoksessa *CyberForensics,* 59–83. Springer.

Liitteet

A Työkaluja haittaohjelmien analysointiin

Tässä liitteessä on esitelty tutkielmassa käytettyjä ja muuten analyysin kannalta hyödyllisiä työkaluja. Kaikki työkalut ovat vapaasti saatavilla ilman lisenssimaksua ellei toisin mainita.

A.1 Pakkauksen tunnistaminen

Nimi	PEiD
Tekijä	snaker
Käyttöjärjestelmä	Windows
Käyttöliittymä	GUI
URL	http://www.aldeid.com/wiki/PEiD
Kuvaus	Tunnistaa yli 160 erilaista pakkausta. Kehitys lopetettu, mutta silti de facto -työkalu pakkauksen tunnistukseen.

Nimi	binwalk
Tekijä	devttys0
Käyttöjärjestelmä	Linux, Mac OS X
Käyttöliittymä	Komentorivi
URL	http://binwalk.org/
Kuvaus	Tiedoston entropian analyysiin. Osaa myös visualisoida entropiaa ja tunnistaa eri laitearkkitehtuurien assembly-käskyjä.

Nimi	ent
Tekijä	John Walker
Käyttöjärjestelmä	Windows
Käyttöliittymä	Komentorivi
URL	http://www.fourmilab.ch/random/
Kuvaus	Laskee tiedoston entropian.

A.2 MD5-tarkistussummalaskurit

Nimi	MD5sum
Tekijä	GNU coreutils
Käyttöjärjestelmä	Linux, BSD, Windows (Cygwin)
Käyttöliittymä	Komentorivi
URL	https://www.gnu.org/software/coreutils/

Nimi	MD5sums
Tekijä	Jem Berkes
Käyttöjärjestelmä	Windows
Käyttöliittymä	Komentorivi
URL	http://www.pc-tools.net/win32/md5sums/

Nimi	WinMD5
Tekijä	Edwin Olson
Käyttöjärjestelmä	Windows
Käyttöliittymä	GUI
URL	http://www.blisstonia.com/software/WinMD5/

A.3 Dynaaminen analyysi

Nimi	CaptureBAT
Tekijä	New Zealand Honeynet Alliance
Käyttöjärjestelmä	Windows
Käyttöliittymä	Komentorivi-ikkuna
URL	http://www.honeynet.org/node/315
Kuvaus	All-in-one -ratkaisu dynaamiseen analyysiin. Seuraa muutoksia tiedostoissa, rekisterissä ja prosesseissa kernel-tasolla.

Nimi	Fakenet
Tekijä	Andrew Honig ja Michael Sikorski
Käyttöjärjestelmä	Windows
Käyttöliittymä	Komentorivi-ikkuna
URL	http://practicalmalwareanalysis.com/fakenet/
Kuvaus	Practical Malware Analysis -kirjan tekijöiden kirjoittama verkkosimulaattori, joka simuloi kaikkia yleisimpiä protokollia. Kaappaa liikenteestä Wireshark-yhteensopivia pcap-tiedostoja. Laajennettavissa lisäosilla.

Nimi	Process Hacker
Tekijä	Wen Jia Liu
Käyttöjärjestelmä	Windows
Käyttöliittymä	GUI
URL	http://processhacker.sourceforge.net/
Kuvaus	Monipuolisempi vaihtoehto Windowsin tehtävänhallinnalle. Näyttää tarkkaan prosessin käyttämät resurssit, moduulit ja tiedostot, ja antaa myös lopettaa prosesseja jota Windowsin tehtävänhallinta ei anna.

Nimi	Process Monitor
Tekijä	Sysinternals
Käyttöjärjestelmä	Windows
Käyttöliittymä	GUI
URL	http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx
Kuvaus	Seuraa tiedosto-, prosessi- ja rekisterioperaatioita käyttäjätasolla ja sisältää monipuoliset suodatusominaisuudet. Yhdistää Sysinternalsin aiempien Filemon- ja Regmon-työkalujen toiminnallisuudet.

Nimi	Wireshark
Tekijä	Gerald Combs ym.
Käyttöjärjestelmä	Windows
Käyttöliittymä	GUI
URL	http://www.wireshark.org/
Kuvaus	Pakettienkaappausohjelma verkkoliikenteen seurantaan. Tulkitsee kaikkia tunnettuja protokollia.

A.4 Ohjelman otsikkotietojen ja sektioiden tarkasteluun

Nimi	Dependency Walker
Tekijä	Steve Miller
Käyttöjärjestelmä	Windows
Käyttöliittymä	GUI
URL	http://www.dependencywalker.com/
Kuvaus	Näyttää ohjelman import- ja export-taulut ja kuvaa moduulien väliset riippuvuussuhteet.

Nimi	PEView
Tekijä	Wayne Radburn
Käyttöjärjestelmä	Windows
Käyttöliittymä	GUI
URL	http://wjradburn.com/software/
Kuvaus	Näyttää ohjelman sektiot ja otsikkotiedot.

A.5 Ohjelman resurssien tarkasteluun ja muokkaamiseen

Nimi	Resource Hacker
Tekijä	Angus Johnson
Käyttöjärjestelmä	Windows
Käyttöliittymä	GUI
URL	http://www.angusj.com/resourcehacker/
Kuvaus	Ensimmäisiä PE-resurssieditoreita. Kehitys lopetettu 2011. Lisää, poistaa, purkaa ja muokkaa resursseja EXE- ja DLL-tiedostojen sisältä.

Nimi	Resource Hacker
Tekijä	Angus Johnson
Käyttöjärjestelmä	Windows
Käyttöliittymä	GUI
URL	http://www.angusj.com/resourcehacker/
Kuvaus	Ensimmäisiä PE-resurssieditoreita. Kehitys lopetettu 2011. Lisää, poistaa, purkaa ja muokkaa resursseja EXE- ja DLL-tiedostojen sisältä.

Nimi	XN Resource Editor
Tekijä	Colin Wilson
Käyttöjärjestelmä	Windows
Käyttöliittymä	GUI
URL	http://www.wilsonc.demon.co.uk/d10resourceeditor.htm
Kuvaus	Avoimen lähdekoodin resurssieditori. Erityistuki Delphi-ohjelmille

A.6 Disassemblerit ja debuggerit

Nimi	IDA Pro
Tekijä	Hex-Rays
Käyttöjärjestelmä	Windows, Linux
Käyttöliittymä	GUI
URL	http://www.hex-rays.com
Kuvaus	Interaktiivinen disassembleri ja debuggeri runsailla ominaisuuksilla. Ilmainen versio IDA Pro Free 5.0 sisältää osan ominaisuuksista ja sillä pääsee alkuun haittaohjelmienkin analyysissä, mutta siitä puuttuu tuki uudemmille Windowsin versioille ja 64-bittisille ohjelmille.

Nimi	Immunity Debugger
Tekijä	Immunity Inc.
Käyttöjärjestelmä	Windows
Käyttöliittymä	GUI ja komentorivi
URL	https://www.immunityinc.com/products-immdbg.shtml
Kuvaus	Python-skripteillä laajennettava kevyt debuggeri.

Nimi	OllyDbg
Tekijä	Oleh Yuschuk
Käyttöjärjestelmä	Windows
Käyttöliittymä	GUI
URL	http://http://www.ollydbg.de/
Kuvaus	Suosittu ilmainen debuggeri, johon on saatavilla paljon lisäosia esimerkiksi anti-debuggaustekniikoiden kiertämiseen ja pakkauksen purkamiseen.

Nimi	PE Explorer
Tekijä	Heaventools Inc.
Käyttöjärjestelmä	Windows
Käyttöliittymä	GUI
URL	http://www.heaventools.com/
Kuvaus	Disassembler, resurssieditori ja sektioeditori yhdessä ohjelmassa. Maksullinen.

Nimi	WinDbg
Tekijä	Microsoft
Käyttöjärjestelmä	Windows
Käyttöliittymä	GUI
URL	http://msdn.microsoft.com/en-us/windows/hardware/hh852365.aspx
Kuvaus	Kernel-debuggeri Windowsille.

A.7 Pakkaus- ja kryptausohjelmat

Nimi	Armadillo / Software Passport
Tekijä	Silicon Realms
Käyttöjärjestelmä	Windows
Käyttöliittymä	GUI
URL	http://www.siliconrealms.com
Kuvaus	Erittäin suosittu kryptausohjelma joka sisältää anti-debugging- ja anti-disassembly-tekniikoita.

Nimi	UPX
Tekijä	John F. Reiser
Käyttöjärjestelmä	Kaikki
Käyttöliittymä	Komentorivi
URL	http://upx.sourceforge.net/
Kuvaus	Avoimen lähdekoodin pakkausohjelma joka tukee useita ohjelmamuotoja ja osaa myös purkaa pakkauksen. Erittäin käytetty varsinkin vanhemmissa haittaohjelmissa.

A.8 Muita ohjelmia

Nimi	HxD
Tekijä	Maël Hörz
Käyttöjärjestelmä	Windows
Käyttöliittymä	GUI
URL	http://http://mh-nexus.de/en/hxd/
Kuvaus	Monipuolinen ja kompakti heksaeditori.

Nimi	strings
Tekijä	GNU binutils. Windows-versio: Mark Russinovich
Käyttöjärjestelmä	Kaikki
Käyttöliittymä	Komentorivi
URL (Windows)	http://technet.microsoft.com/en-us/sysinternals/bb897439.aspx
Kuvaus	Etsii merkkijonoja tiedostosta.

B Tärkeimmät x86-assemblykäskyt

Tässä liitteessä on lueteltuna analyysin kannalta tärkeimmät x86-assemblykäskyt. Täydellinen käskykanta on saatavissa Intelin käsikirjan “Intel 64 and IA-32 Architectures Software Developers Manual” 2013 osasta 2.

B.1 Datan siirto ja muistin osoitus

MOV toimii sijoitusoperaattorin tavoin, eli se kopioi toisesta sille annetusta operandista arvon ensimmäiseen. MOV ei muuta lippuja. Esimerkkejä:

Assembly	Konekieli	Selitys
MOV eax, ebx	8B C3	Kopioi rekisterin ebx sisällön rekisteriin eax.
MOV eax, 6	B8 06 00 00 00	Kopioi kokonaisluvun 6 rekisteriin eax.
MOV eax, [ebx]	8B 03	Kopioi 4 tavua (dword) rekisterin ebx osoittamasta muistiosoitteesta rekisteriin eax.
MOV al, byte ptr [ebx]	8A 03	Kopioi yhden tavun (byte) rekisterin ebx osoittamasta muistipaikasta rekisteriin al.
MOV dword ptr [eax], ebx	89 18	Kopioi 4 tavua (dword) rekisteristä ebx rekisterin eax osoittamaan muistiosoitteeseen.
MOV dword ptr [eax], 2	C7 00 02 00 00 00	Kopioi kokonaisluvun 2 rekisterin eax osoittamaan muistiosoitteeseen.

Hakasulkuoperaattoria **[]** voidaan käyttää selvittämään hakasulkujen sisällä olevassa muistiosoitteessa oleva arvo. Muistiosoite on muotoa a*r+b miaaä a on kerroin, joka voi olla 2, 4 tai 8, r yleiskäyttöinen rekisteri ja b joko kokonaisluku tai rekisteri. Määreitä "dword ptr", "word ptr"ja "byte ptr"voidaan käyttää kertomaan osoitettavan muistialueen koko jos assembler ei sitä pysty automaattisesti päättelemään, esimerkiksi tapauksessa mov [ebx], 2. (“x86 Assembly Guide” 2013)

LEA (load effective address) toimii päinvastoin kuin **[]**-operaattori, eli se palauttaa muistiosoitteen sille parametrina annetulle arvolle. Assembly-koodia kirjoitettaessa sitä voidaan käyttää yhdessä muuttujien symbolisten nimien kanssa, mutta konekielisenä sitä käytetään enemmän yhdessä hakasulkuoperaattorin kanssa laskemaan kerto- ja yhteenlaskun yhdellä käskyllä. Kohdeoperandi on aina rekisteri ja lähdeoperandi muistiosoite (tai sen symbolinen esitys).

Assembly	Konekieli	Selitys
LEA eax, HelloWorld	8D 05 xx xx xx xx	Lataa symbolin HelloWorld muistiosoitteen rekisteriin eax.
LEA eax, [4*ebx+1]	8D 04 9D 01 00 00 00	Laskee laskutoimituksen 4*ebx+1 ja sijoittaa sen tuloksen rekisteriin eax.
LEA ecx, [2*eax+ebx]	8D 0C 43	Laskee laskutoimituksen 2*eax+ebx ja sijoittaa sen tuloksen rekisteriin ecx.

B.2 Kokonaislukuaritmetiikka

ADD laskee yhteen kaksi sille annettua operandia ja sijoittaa tuloksen ensimmäiseen niistä. **SUB** suorittaa vastaavan vähennyslaskun.

Assembly	Konekieli	Selitys
ADD eax, ebx	03 C3	Laskee yhteen eax- ja ebx-rekisterien sisällön ja sijoittaa tuloksen rekisteriin eax.
SUB eax, ebx	2B C3	Vähentää rekisterin ebx sisällön rekisterin eax sisällöstä ja sijoittaa tuloksen rekisteriin eax.
ADD eax, dword ptr [ebx]	03 03	Laskee yhteen eax-rekisterin arvon ja ebx-rekisterin osoittamasta muistiosoitteesta löytyvän neljän tavun mittaisen arvon ja sijoittaa tuloksen rekisteriin eax.
ADD dword ptr [eax], ebx	01 18	Laskee yhteen eax-rekisterin osoittaman dword-arvon ja ebx-rekisterin sisällön ja sijoittaa tuloksen eax-rekisterin osoittamaan muistiosoitteeseen.
ADD eax, 1	83 C0 01	Lisää arvon 1 rekisteriin eax.
ADD [eax], ebx	01 18	Lisää rekisterin ebx arvon rekisterin eax osoittamaan muistiosoitteeseen.

INC kasvattaa ja **DEC** vähentää sille annettua operandia yhdellä. Operandi voi olla joko rekisteri tai muistiosoite.

Assembly	Konekieli	Selitys
INC eax	40	Kasvattaa rekisterin eax arvoa yhdellä.
INC ebx	43	Kasvattaa rekisterin eax arvoa yhdellä.
DEC eax	48	Vähentää rekisterin ebx arvoa yhdellä.
INC dword ptr [eax]	FF 00	Kasvattaa rekisterin eax osoittamaa arvoa yhdellä.
DEC dword ptr [eax]	FF 08	Vähentää rekisterin eax osoittamaa arvoa yhdellä.

MUL ja **IMUL** kertovat kaksi lukua keskenään. IMUL ottaa myös operandien etumerkin huomioon. MUL ottaa vain yhden operandin, jolla se kertoo EAX-rekisterin arvon. IMUL-käskyä käytetään yleensä kahdella operandilla, jolloin kertolaskun tulos sijoitetaan rekistereihin EAX (vähemmän merkitsevä osa) ja EDX (enemmän merkitsevä osa). Käskystä on myös kolmen operandin versio, jossa kahden viimeisen operandin tulo sijoitetaan ensimmäiseen, mutta viimeisen operandin on oltava vakio.

Assembly	Konekieli	Selitys
IMUL ebx, ecx	0F AF D9	Kertoo rekisterien ebx ja ecx sisältämät etumerkilliset luvut keskenään ja sijoittaa tuloksen rekistereihin eax ja edx.
IMUL cx, 5	66 6B C9 05	Kertoo rekisterin cx sisältämän etumerkillisen luvun vakiolla 5 ja sijoittaa tuloksen rekistereihin ax ja dx.
IMUL ebx, ecx, 5	6B D9 05	Kertoo rekisterin ecx sisältämän etumerkillisen luvun vakiolla 5 ja sijoittaa tuloksen rekisteriin ebx.
IMUL byte ptr [ebx]	F6 2B	Kertoo rekisterin ebx osoittaman tavun rekisterin al sisältämän tavun kanssa ja sijoittaa tuloksen rekisteriin ax.
MUL ecx	F7 E1	Kertoo rekisterin ecx arvon rekisterin eax sisältämän arvon kanssa ja sijoittaa tuloksen rekisteriin rekistereihin eax ja edx.

DIV ja **IDIV** jakavat rekistereiden EAX (vähemmän merkitsevä) ja EDX (enemmän merkitsevä) sisältämän arvon sille annetulla arvolla. IDIV ottaa myös operandien etumerkin huomioon. Jos jakaja on 0, tapahtuu poikkeus 0, ja suoritus siirtyy sille asetettuun poikkeuksenkäsittelijään. Huomaa, että (I)DIV-käsky ei tue vakiotyyppistä operandia jakajana, vaan sen on oltava aina joko rekisteri tai muistiosoite. Osamäärä sijoitetaan rekisteriin eax ja jakojäännös rekisteriin edx.

Assembly	Konekieli	Selitys
IDIV ecx	F7 E9	Jakaa rekisterien edx:eax sisältämän arvon rekisterin ecx arvolla ja sijoittaa osamäärän rekisteriin eax ja jakojäännöksen rekisteriin edx. Ottaa huomioon etumerkin.
DIV word ptr [ecx]	66 F7 31	Jakaa rekisterien dx:ax sisältämän arvon rekisterin ecx osoittamalla sanalla (2 tavua) ja sijoittaa osamäärän rekisteriin ax ja jakojäännöksen rekisteriin dx. Ei ota huomioon etumerkkiä.

B.3 Boolean-aritmetiikka

Boolean-operaatiot **AND**, **OR** ja **XOR** ottavat kukin kaksi operandia, ja sijoittavat operaation tuloksen niistä ensimmäiseen. Operaatiot päivittävät myös lippurekisteriä, joten esimerkiksi lippua Z tarkastelemalla nähdään, onko operaation tulos nolla.

Assembly	Konekieli	Selitys
AND eax, ebx	23 C3	Suorittaa loogisen ja-operaation rekisterien eax ja ebx arvoille, ja sijoittaa tuloksen rekisteriin eax.
OR ax, bx	66 0B C3	Suorittaa loogisen tai-operaation rekisterien ax ja bx arvoille, ja sijoittaa tuloksen rekisteriin ax.
XOR eax, dword ptr [ebx]	33 03	Suorittaa loogisen poissulkeva tai -operaation rekisterin eax sisällölle ja rekisterin ebx osoittamalle tuplasanalle, ja sijoittaa tuloksen rekisteriin eax.

Operaatiot **NOT** (yhden komplementti) ja **NEG** (kahden komplementti) ottavat vain yhden operandin, jota ne muuttavat. Operandi voi olla joko rekisteri tai muistiosoite.

Assembly	Konekieli	Selitys
NOT eax	F7 D0	Lukee rekisterin eax arvon, kääntää sen bitit ja kirjoittaa takaisin rekisteriin eax.
NEG eax	F7 D8	Lukee rekisterin eax arvon, kääntää sen etumerkin ja kirjoittaa takaisin rekisteriin eax.

B.4 Pino

x86-arkkitehtuurissa pino kasvaa alaspäin, eli siitä varataan muistia vähentämällä pino-osoitinta ESP. Vastaavasti varattu muisti vähennetään kasvattamalla pino-osoitinta vastaavalla määrällä tavuja.

PUSH varaa pinosta automaattisesti oikean kokoisen muistialkion ja siirtää siihen sille parametrina annetun arvon. Esimerkiksi

```
PUSH eax    ; konekielellä: 50
```

vastaa käskyjä

```
SUB ESP, 4      ; konekielellä: 83 EC 04
MOV [ESP], EAX ; konekielellä: 89 04 24
```

POP vastaavasti poistaa muistialkion pinon huipulta.

```
POP EBX   ; konekielellä: 5B
```

Huomaa, että alkion kokoa ei määrää sen alkuperäinen koko, vaan rekisterin koko johon se siirretään. Varomaton käyttö voi aiheuttaa dataesityksen vääristymiseen pinossa, mutta sitä voidaan käyttää myös hyödyksi pilkkomaan tai yhdistämään useammasta tavusta koostuvia dataesityksiä:

```
PUSH EAX ; konekielellä: 50
POP BX    ; konekielellä: 66 5B
POP CX    ; konekielellä: 66 59
```

B.5 Suorituksen haarauttaminen

JMP hyppää sille annettuun muistiosoitteeseen korkeamman tason kielien **goto**-lauseen tapaan.

```
0x00401000  JMP 0x00401050
0x00401006  (ei suoriteta)
...
0x00401050  (suoritus jatkuu)
```

CALL hyppää sille annettuun osoitteeseen samaan tapaan kuin JMP, mutta se tallentaa paluuosoitteen (seuraavan käskyn osoite) pinoon. Kutsusta voidaan palata **RET**-käskyllä. Alla olevassa esimerkissä suoritusjärjestystä on havainnollistettu suluissa olevien numeroiden avulla.

```
0x00401000  (1) CALL 0x00401050
0x00401006  (4) ...
...
```

```
0x00401050    (2)  ...
0x00401100    (3)  RET
```

Käskyjen JMP ja CALL osoitteet ovat konekielellä seuraavan käskyn osoitteeseen suhteellisia siirtymiä. Siirtymä voi olla annettu joko suoraan (etumerkillisenä) tai epäsuorasti (rekisterissä tai muistiosoitteessa). Poikkeuksena tähän ovat ns. kauas tehtävät hypyt ja kutsut (far calls / jumps) joissa päivitetään myös segmenttirekisteriä CS käskyosoittimen EIP lisäksi. Näissä hypyissä kohde on ilmoitettu absoluuttisena osoitteena. ("Intel 64 and IA-32 Architectures Software Developers Manual" 2013, Vol. 2, Chapter 3)

Ehdollisten ja ehdottomien hyppyjen operaatiokoodit on esitelty taulukossa 5.

B.6 Vertailu ja ehdolliset hypyt

CMP vertaa kahden sille annetun operandin yhtäsuuruutta vähentämällä jälkimmäisen operandin ensimmäisestä. Vertailu muuttaa lippuja CF, OF, SF, ZF, AF ja PF mutta niitä harvemmin luetaan suoraan, sen sijaan käytetään ehdollisia hyppykäskyjä. Vähennyslaskun tulosta ei tallenneta.

Assembly	Konekieli	Selitys
CMP eax, ebx	3B C3	Vertaa rekisterin eax sisältöä rekisterin ebx sisältöön.
CMP eax, [ebx]	3B 03	Vertaa rekisterin eax sisältöä rekisterin ebx sisältämän muistiosoitteen sisältöön.
CMP [eax], ebx	39 18	Vertaa rekisterin eax sisältämän muistiosoitteen sisältöä rekisterin ebx sisältöön.
CMP eax, 0	83 F8 00	Vertaa rekisterin eax sisältöä kokonaislukuvakioon 0.

TEST vertaa kahta operandia loogisella ja-operaatiolla ja asettaa liput SF, ZF ja PF. Operaation tulosta ei tallenneta. Yleisin käyttötapa tälle käskylle on antaa sama rekisteri kummaksikin operandiksi jolloin ZF-lippu kertoo (nopeammin kuin vastaava CMP-operaatio) onko rekisterin arvo nolla.

Hyppykäskyistä **JE** (jump if equal) ja **JZ** (jump if zero) suorittavat hypyn kun ZF-lippu on asetettu eli edellisen laskutoimituksen arvo on nolla. Vastaavasti **JNE** (jump if equal) ja **JNZ** hyppäävät kohdeosoitteeseen kun päinvastainen ehto toteutuu.

CMP-käskyn yhteydessä voidaan käyttää myös suurempi- ja pienempi kuin -vertailuja. Seuraavassa taulukossa on esitetty kaikki relaatiot ja niitä vastaavat käskyt.

Käsky	**Operaattori**	**Relaatio**
JE, JZ	==	yhtäsuuruus
JNE, JNZ	!=	erisuuruus
JL	<	pienempi kuin
JLE	<=	pienempi tai yhtä suuri kuin
JG	>	suurempi kuin
JGE	>=	suurempi tai yhtä suuri kuin

Ehdollisia hyppyjä on myös muita, esimerkiksi **JC** hyppää jos carry-lippu on asetettuna. Epäsuoria hyppyjä ja hyppyjä kauas absoluuttisilla osoitteilla ei tueta ehdollisina x86-arkkitehtuurissa (“Intel 64 and IA-32 Architectures Software Developers Manual” 2013, Vol. 2, Chapter 3). Ehdollisten ja ehdottomien hyppyjen operaatiokoodit on esitelty taulukossa 5.

Operandi	JMP	JE	JNE	JL	JLE	JG	JGE
Tavu	EB	74	75	7C	7E	7F	7D
Sana	E9	0F84	0F85	0F8C	0F8E	0F8F	0F8E
Tuplasana	E9	0F84	0F85	0F8C	0F8E	0F8F	0F8E
Rekisteri / muistiosoite	FF						
Osoite suoraan (hyppy kauas)	EA						
Osoite muistista (hyppy kauas)	FF						

Taulukko 5. Hyppykäskyjen operaatiokoodit x86-arkkitehtuurissa. Lähde: “Intel 64 and IA-32 Architectures Software Developers Manual” 2013, Vol. 2, Chapter 3.

C Taulukot

Rekisteri	Nimi englanniksi	Käyttötarkoitus
EIP	Extended Instruction Pointer	Osoittaa seuraavaksi suoritettavan käskyn.
EAX	Extended Accumulator Register	Yleiskäyttöinen rekisteri.
EBX	Extended Base Register	Yleiskäyttöinen rekisteri.
ECX	Extended Counter Register	Yleiskäyttöinen rekisteri. Käytetään laskurina merkkijono-operaatiossa ja silmukoissa.
EDX	Extended Data Register	Yleiskäyttöinen rekisteri.
ESI	Extended Source Index	Yleiskäyttöinen rekisteri, käytetään yleisimmin merkkijono-operaatioissa lähdeindeksinä.
EDI	Extended Destination Index	Yleiskäyttöinen rekisteri, käytetään yleisimmin merkkijono-operaatioissa kohdeindeksinä.
EBP	Extended Base Pointer	Osoittaa pinon pohjan. Aloittaa ns. pinokehyksen joita käytetään funktiokutsujen yhteydessä (ks. seuraava alaluku).
ESP	Extended Stack Pointer	Osoittaa pinon huipun. Muuttuu joka kerta kun pinoon lisätään (PUSH) tai siitä poistetaan (POP). Voidaan muuttaa myös käsin.
EFLAGS	Extended flags	Lippu- eli boolean-muuttujia prosessorin tilasta ja edellisen käskyn tuloksesta.

Taulukko 6. x86-prosessorin rekisterit. Lähde: “Intel 64 and IA-32 Architectures Software Developers Manual” 2013, Vol. 1, chapter 3.4

Lippu	Nimi englanniksi	Käyttötarkoitus
ZF	Zero Flag	1 jos edellisen laskutoimituksen tulos on 0.
CF	Carry Flag	1 jos edellisen laskutoimituksen tulos ei mahtunut kohdeoperandiin.
SF	Sign Flag	1 jos edellisen laskutoimituksen tulos on negatiivinen, 0 jos positiivinen. Etumerkittömillä luvuilla laskettaessa kertoo tuloksen merkittävimmän bitin.
TF	Trap Flag	Jos asetettu (1) niin prosessori laukaisee jokaisen käskyn suorituksen jälkeen keskeytyksen 1. Näin debuggerilla voidaan suorittaa käskyjä yksi kerrallaan. (Eilam 2011, s. 332)

Taulukko 7. x86-prosessorin tärkeimmät liput debuggauksen kannalta. Lähde: Sikorski, Honig ja Lawler 2012, s. 72

Kirjasto	**Funktio**	**Käyttötarkoitus**
advapi32.dll	RegSetValueExA	Kirjoittaa arvon rekisteriin.
	RegQueryValueExA	Lukee arvon rekisteristä.
	RegCreateValueExA	Luo uuden rekisteriavaimen.
kernel32.dll	GetFileSize	Palauttaa tiedoston koon.
	FindFirstFile	Etsii tiedostoja tiedostojärjestelmästä.
	FindNextFile	
	FindClose	
	GetTempPathA	Palauttaa käyttäjän hakemiston väliaikaisille tiedostoille.
	GetTempFileNameA	Palauttaa nimen uudelle väliaikaistiedostolle.
	GetWindowsDirectoryA	Palauttaa Windows-hakemiston polun.
	GetEnvironmentVariable	Lukee ympäristömuuttujan arvon.
	GetFileAttributes	Lukee tiedoston attribuutit.
	SetFileAttributes	Asettaa attribuutit tiedostolle.
	LoadLibraryA	Ottaa DLL-kirjaston käyttöön ajonaikaisesti.

Kirjasto	Funktio	Käyttötarkoitus
	GetProcAddress	Etsii annetun funktion osoitteen funktiokutsua varten.
	CreateProcessA	Käynnistää uuden prosessin (ohjelman).
	CreateThread	Luo uuden säikeen prosessille.
	CreateMutexA	Luo mutex-objektin (käytetään säikeiden väliseen kommunikointiin).
	ExitProcess	Lopettaa nykyisen prosessin.
	ExitThread	Lopettaa nykyisen säikeen.
	GetModuleFileNameA	Palauttaa tiedoston nimen polkuineen nimetylle prosessille.
	GetModuleHandleA	Palauttaa kahvan nimetylle prosessille.
	InterlockedIncrement	Kasvattaa säikeiden välillä jaetun muuttujan arvoa.
	InterlockedDecrement	
	CopyFileA	Kopioi tiedoston.
	DeleteFileA	Poistaa tiedoston.
	CreateFileA	Avaa uuden tai olemassa olevan tiedoston.
	ReadFile	Lukee tiedostosta.
	WriteFile	Kirjoittaa tiedostoon.
	CreateFileMapping	Avaa muistinäkymän tiedostoon.
	MapViewOfFile	Avaa osan tiedostoa muistiin lukua tai kirjoitusta varten.
	UnmapViewOfFile	
	GetSystemTime	Palauttaa järjestelmän päivämäärän ja ajan UTC-ajassa.
	GetLocalTime	Palauttaa järjestelmän päivämäärän ja ajan paikallisessa ajassa.
	GetTimeZoneInformation	Palauttaa tietorakenteen järjestelmän aikavyöhyketiedoista.

Kirjasto	Funktio	Käyttötarkoitus
	GetTickCount	Palauttaa järjestelmän päivämäärän ja ajan.
	Sleep	Odottaa annetun ajanjakson.
user32.dll	FindWindowA	Etsii ikkunan tietyllä tyypillä ja/tai otsikolla ja palauttaa kahvan siihen.
	PostMessageA	Lähettää viestin ikkunalle.
ws2_32.dll	connect	Yhdistää soketin annettuun palvelimeen ja porttiin.
	send	Lähettää dataa yhdistettyyn sokettiin.
	recv	Vastaanottaa dataa yhdistetystä soketista.
	closesocket	Katkaisee yhteyden palvelimeen.
dnsapi.dll	DnsQuery_A	Tekee dns-nimikyselyn.
iphlpapi.dll	GetNetworkParams	Hakee tietoja verkkoyhteydestä ja verkkoadaptereista.
urlmon.dll	URLDownloadToCacheFile	Lataa tiedoston http- tai ftp-osoitteesta selaimen välimuistiin ja palauttaa sen tiedostonnimen.
wininet.dll	InternetGetConnectedState	Palauttaa internetyhteyden tilan.

Taulukko 8: Puretun MyDoom-ohjelmatiedoston merkkijonoista löydettyjä kirjastofunktioita.

D Kuvat

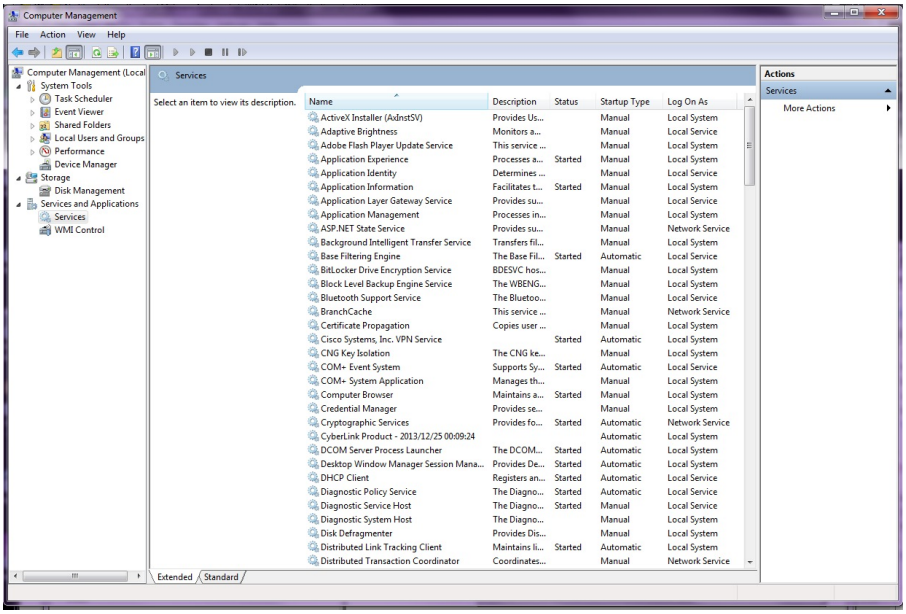
D.1 Luku 4

```
; Listing generated by Microsoft (R) Optimizing \
;Compiler Version 16.00.40219.01

TITLE   C:\\polku\\simple.c
.686P
.XMM
include listing.inc
.model  flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

PUBLIC  _main
; Function compile flags: /Odtp
_TEXT   SEGMENT
_argc$ = 8      ; size = 4
_argv$ = 12     ; size = 4
_main   PROC
; File c:\\polku\\simple.c
; Line 2
        push    ebp
        mov     ebp, esp
; Line 3
        xor     eax, eax
; Line 4
        pop     ebp
        ret     0
_main   ENDP
_TEXT   ENDS
END
```

Kuvio 33. Microsoftin cl-kääntäjällä assembly-kielelle käännetty C-kielinen Hello World -ohjelma.

```
BOOL ReplaceIATEntryInOneMod(PCSTR pszCalleeModName,
   PROC pfnCurrent, PROC pfnNew, HMODULE hmodCaller)
{
   ULONG ulSize;
   PIMAGE_IMPORT_DESCRIPTOR pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR)
      ImageDirectoryEntryToData(hmodCaller, TRUE,
      IMAGE_DIRECTORY_ENTRY_IMPORT, &ulSize);

   if (pImportDesc == NULL)
      return;  // Moduulilla ei ole import-sektiota

   // Etsitään import descriptor -lista, joka sisältää viitteet
   // annetusta kirjastosta ladattuihin funktioihin
   for (; pImportDesc->Name; pImportDesc++) {
      PSTR pszModName = (PSTR)
         ((PBYTE) hmodCaller + pImportDesc->Name);
      if (lstrcmpiA(pszModName, pszCalleeModName) == 0)
         break;
   }

   if (pImportDesc->Name == 0)
      // Moduuli ei lataa funktioita annetusta kirjastosta
      return;

   // Lasketaan viite import-osoitetauluun
   PIMAGE_THUNK_DATA pThunk = (PIMAGE_THUNK_DATA)
      ((PBYTE) hmodCaller + pImportDesc->FirstThunk);

   // Käydään läpi funktiolistaa
   for (; pThunk->u1.Function; pThunk++)
   {
      // Osoitin funktion osoitteeseen
      PROC* ppfn = (PROC*) &pThunk->u1.Function;

      // Vastaako osoite etsittyä?
      if ( *ppfn == pfnCurrent ) {
         // Asennetaan koukku
         WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,
            sizeof(pfnNew), NULL);
         return TRUE;
      }
   }

   return FALSE;
}
```

Kuvio 34. Esimerkki import-osoitetaulukoukun asentavasta aliohjelmasta. Vapaasti mukaeltu lähteestä Richter 1999, luku 22



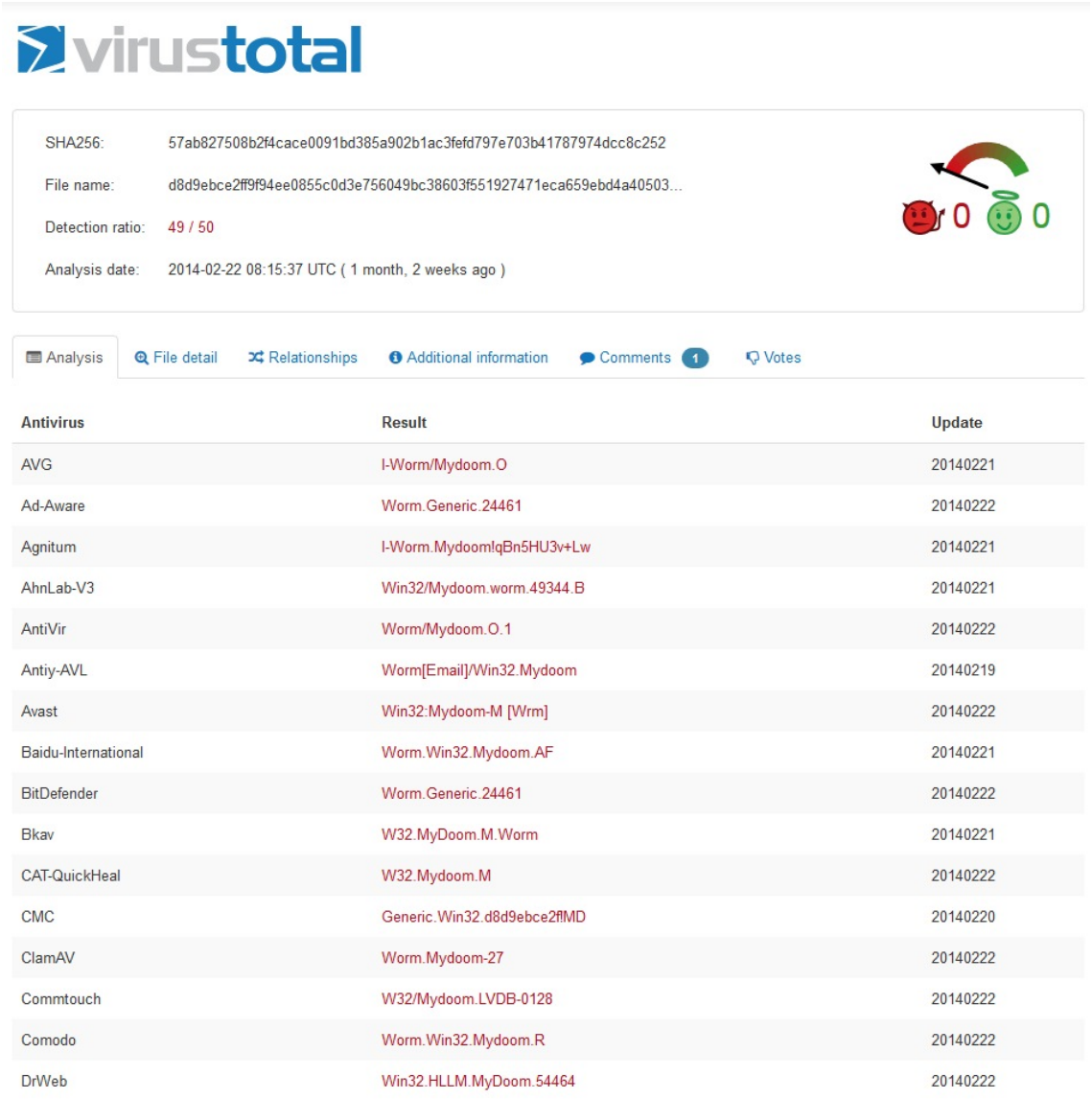
Kuvio 35. Kuvakaappaus järjestelmänvalvontakonsolista, jossa on näkyvissä Windowsin palveluita.

D.2 Luku 5

```
############################################################
::       ScoopyNG - The VMware Detection Tool     ::
::              Windows version v1.0              ::

[+] Test 1: IDT
IDT base: 0x8003f400
Result  : Native OS


[+] Test 2: LDT
LDT base: 0xdead0000
Result  : Native OS


[+] Test 3: GDT
GDT base: 0x8003f000
Result  : Native OS


[+] Test 4: STR
STR base: 0x28000000
Result  : Native OS


[+] Test 5: VMware "get version" command
Result  : Native OS


[+] Test 6: VMware "get memory size" command
Result  : Native OS


[+] Test 7: VMware emulation mode
Result  : Native OS or VMware without emulation mode
          (enabled acceleration)

::                   tk,  2008                    ::
::                [ www.trapkit.de ]              ::
############################################################
```

Kuvio 36. Kuvakaappaus ScoopyNG-ohjelmasta, joka käyttää tunnetuimpia menetelmiä virtuaalikoneen tunnistamiseen.

D.3 Luku 6



virustotal

SHA256: 57ab827508b2f4cace0091bd385a902b1ac3fefd797e703b41787974dcc8c252

File name: d8d9ebce2ff9f94ee0855c0d3e756049bc38603f551927471eca659ebd4a40503...

Detection ratio: 49 / 50

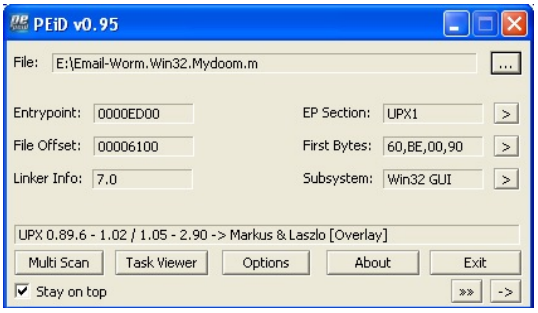
Analysis date: 2014-02-22 08:15:37 UTC (1 month, 2 weeks ago)

0 0

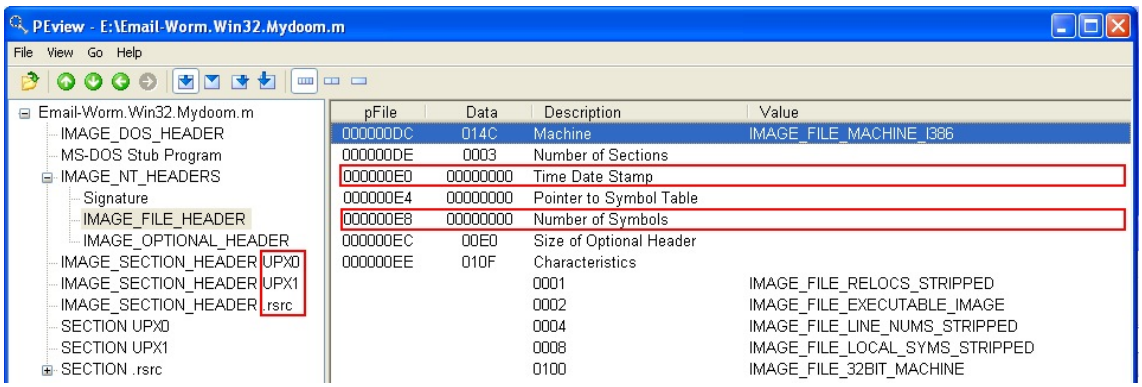
Analysis | File detail | Relationships | Additional information | Comments 1 | Votes

Antivirus	Result	Update
AVG	I-Worm/Mydoom.O	20140221
Ad-Aware	Worm.Generic.24461	20140222
Agnitum	I-Worm.Mydoom!qBn5HU3v+Lw	20140221
AhnLab-V3	Win32/Mydoom.worm.49344.B	20140221
AntiVir	Worm/Mydoom.O.1	20140222
Antiy-AVL	Worm[Email]/Win32.Mydoom	20140219
Avast	Win32:Mydoom-M [Wrm]	20140222
Baidu-International	Worm.Win32.Mydoom.AF	20140221
BitDefender	Worm.Generic.24461	20140222
Bkav	W32.MyDoom.M.Worm	20140221
CAT-QuickHeal	W32.Mydoom.M	20140222
CMC	Generic.Win32.d8d9ebce2ffMD	20140220
ClamAV	Worm.Mydoom-27	20140222
Commtouch	W32/Mydoom.LVDB-0128	20140222
Comodo	Worm.Win32.Mydoom.R	20140222
DrWeb	Win32.HLLM.MyDoom.54464	20140222

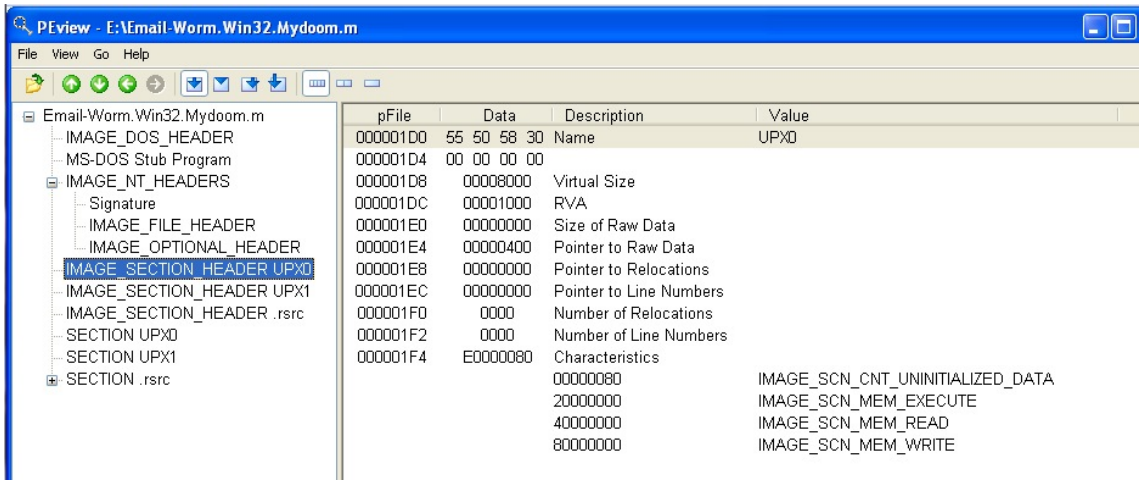
Kuvio 37. Kuvakaappaus VirusTotal-sivuston tunnistamasta MyDoom-haittaohjelmasta.



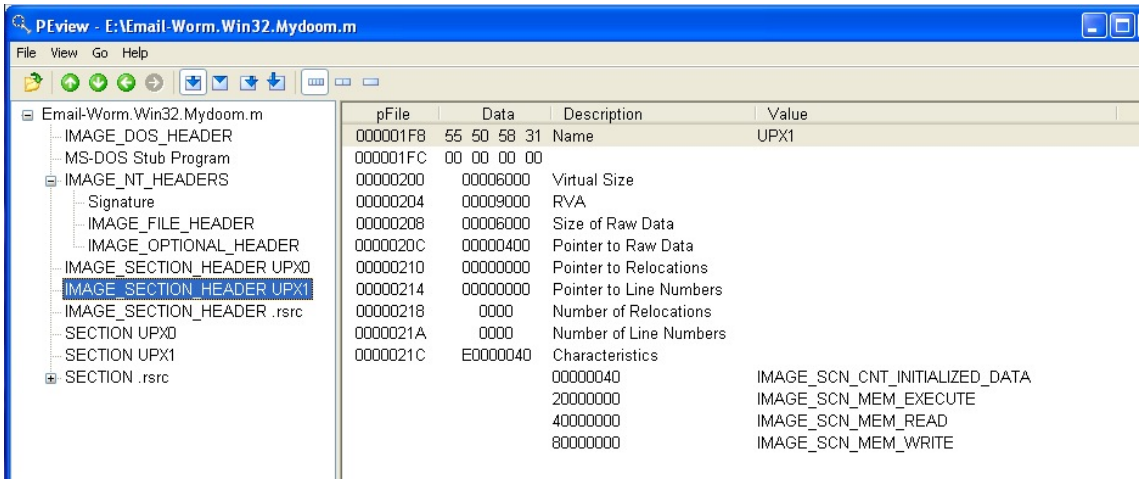
Kuvio 38. Kuvakaappaus PEid-ohjelmasta, joka näyttää, että haittaohjelma on pakattu UPX-pakkauksella.



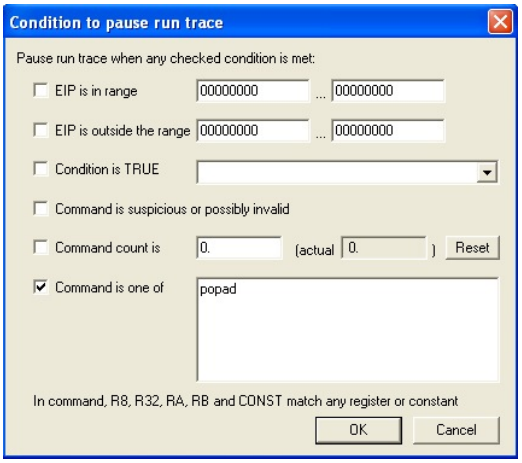
Kuvio 39. Kuvakaappaus PEview-ohjelmasta: IMAGE_FILE_HEADER-sektio.



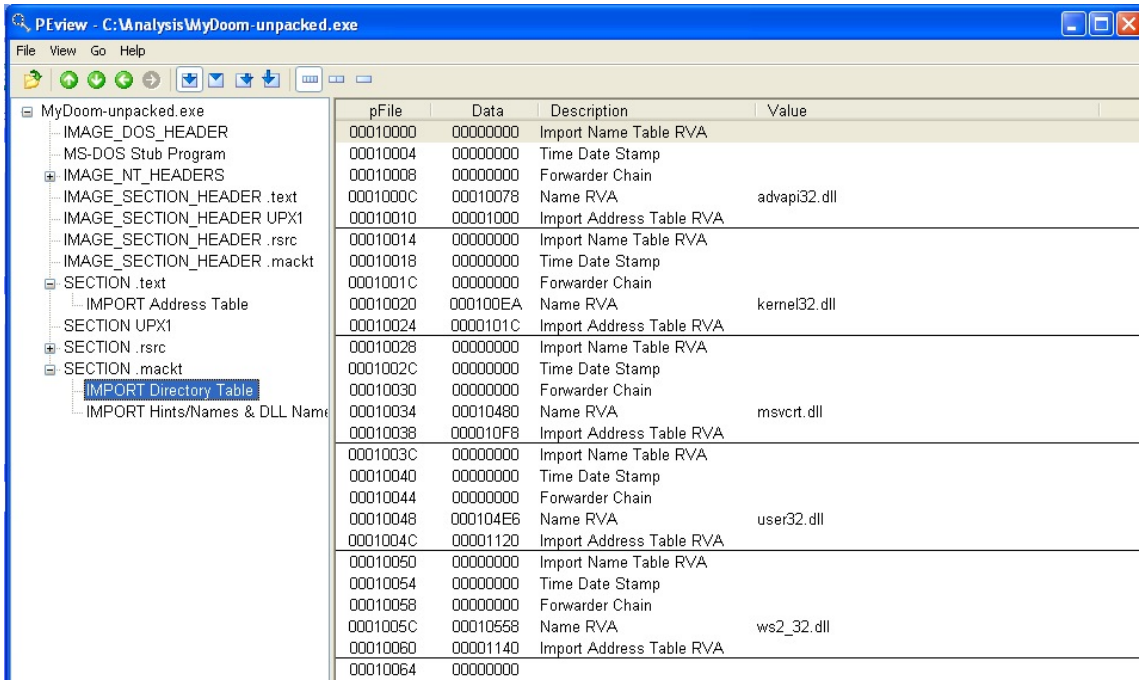
Kuvio 40. Kuvakaappaus PEview-ohjelmasta: pakatun ohjelman UPX0-sektio.



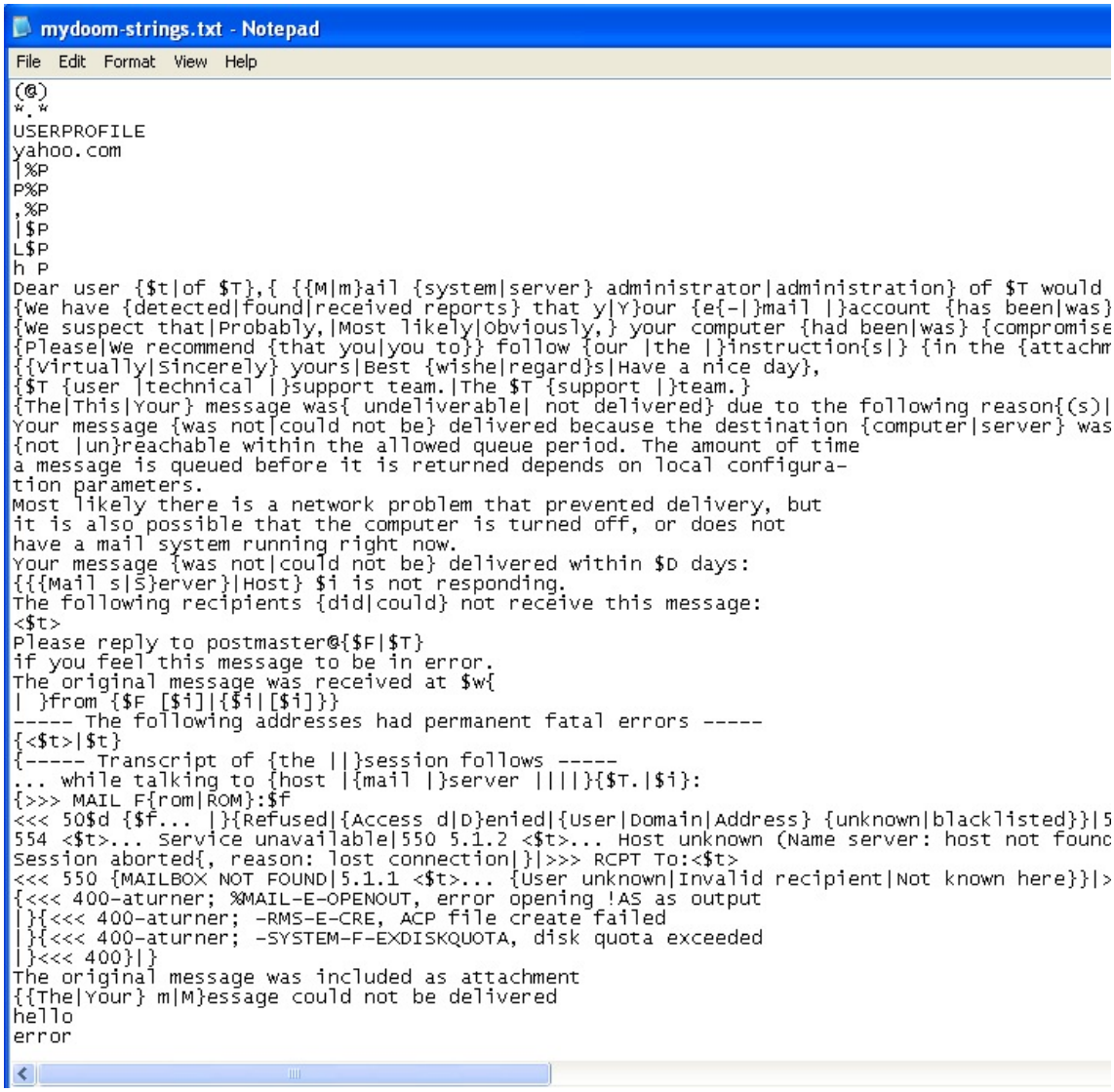
Kuvio 41. Kuvakaappaus PEview-ohjelmasta: pakatun ohjelman UPX1-sektio.



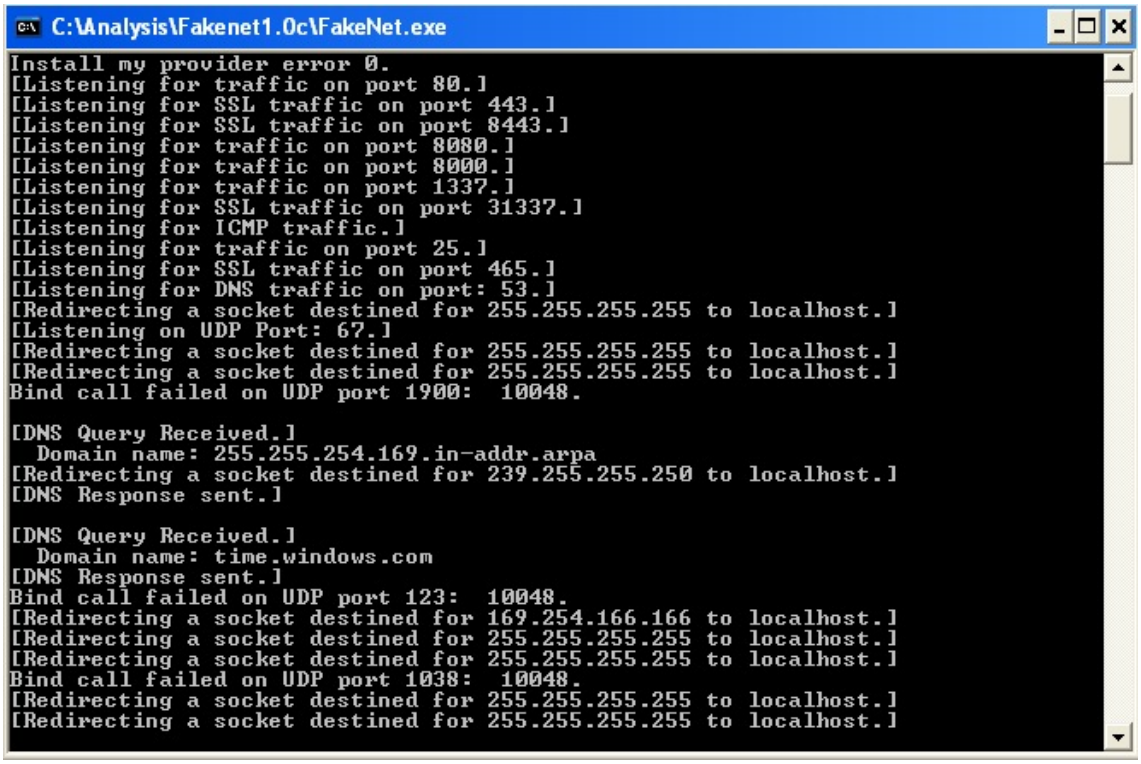
Kuvio 42. Kuvakaappaus OllyDbg-ohjelmasta: suorituksen pysäyttäminen POPAD-käskyyn.



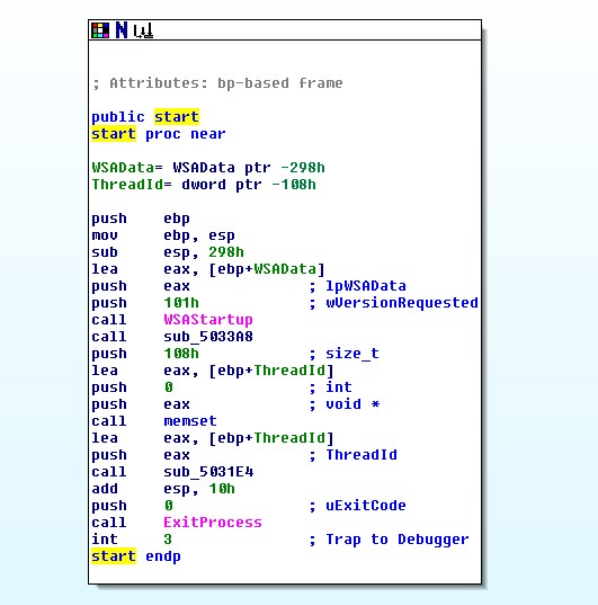
Kuvio 43. Kuvakaappaus PEView-ohjelmasta: puretun ohjelman rekonstruoitu import-hakemisto.



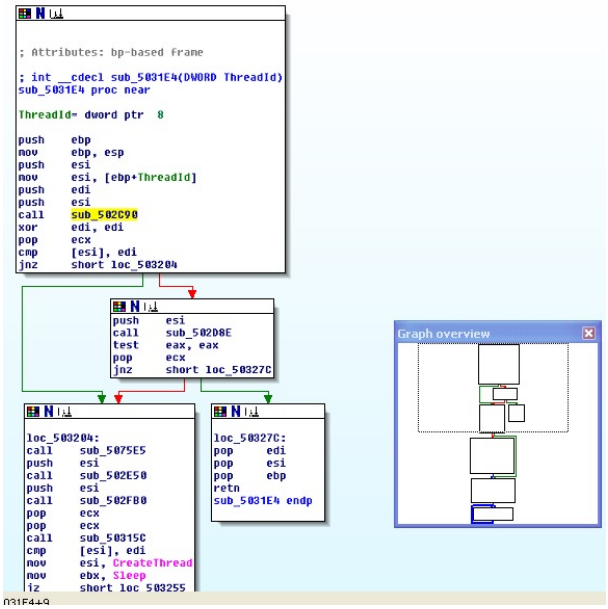
Kuvio 44. Kuvakaappaus Windowsin Muistiosta, jossa näkyy puretusta haittaohjelmasta kaapattuja merkkijonoja.



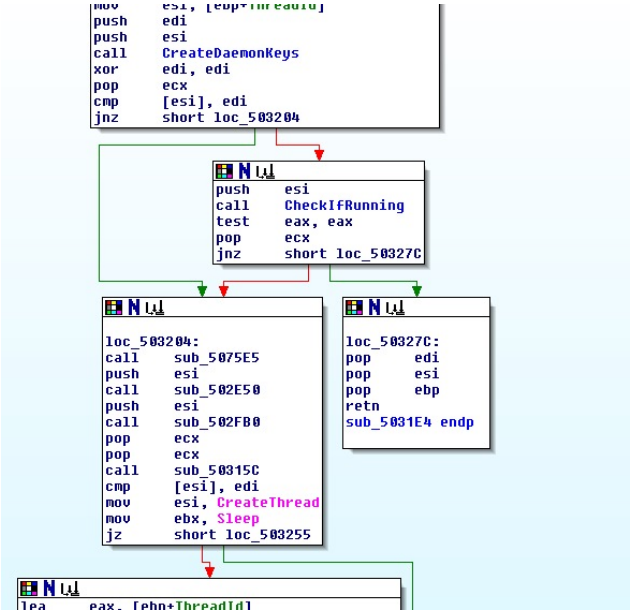
Kuvio 45. Kuvakaappaus Fakenet-verkkosimulaattorista haittaohjelman ajon jälkeen.



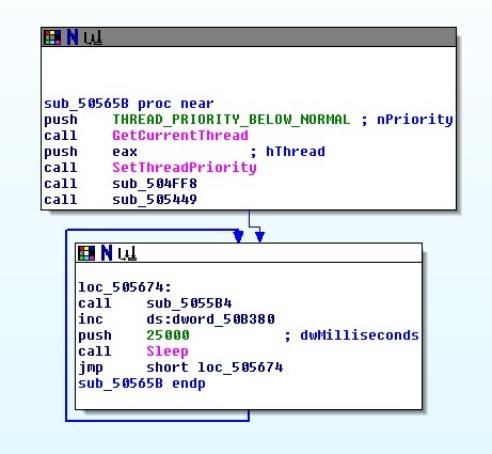
Kuvio 46. MyDoom-haittaohjelman aloituskohta. Kuvakaappaus IDA Pro -disassemblerista.



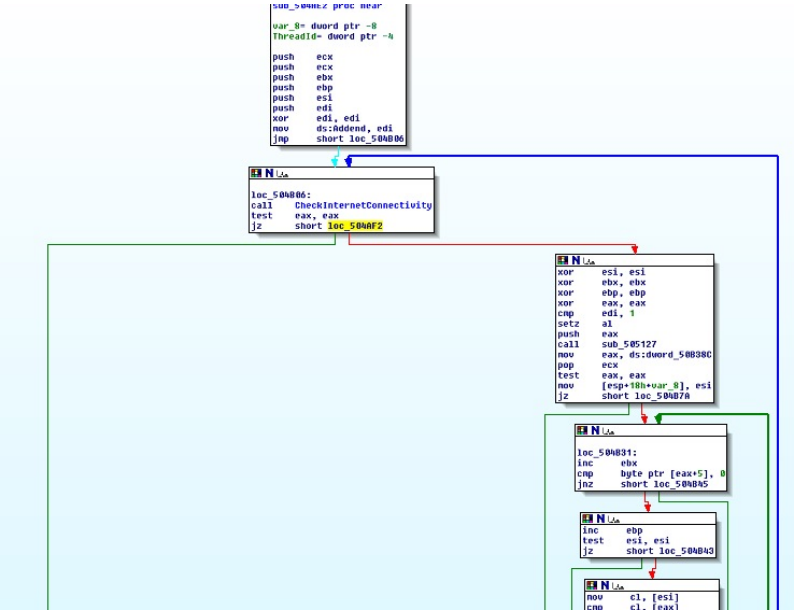
Kuvio 47. MyDoom-haittaohjelman funktion sub_5031E4 alku. Kuvakaappaus IDA Pro -disassemblerista.



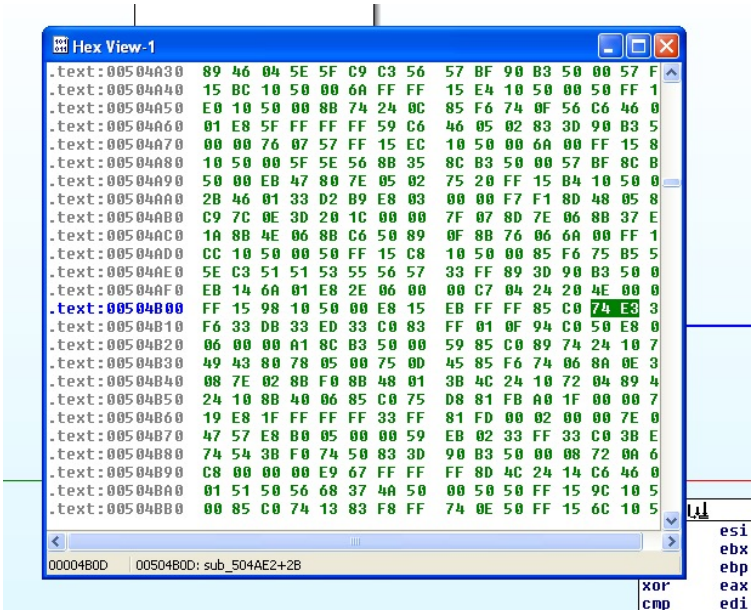
Kuvio 48. Jatkoa MyDoom-haittaohjelman funktiolle sub_5031E4. Osa funktiokutsuista on nimetty uudelleen niiden toiminnallisuuden analyysin perusteella. Kuvakaappaus IDA Pro -disassemblerista.



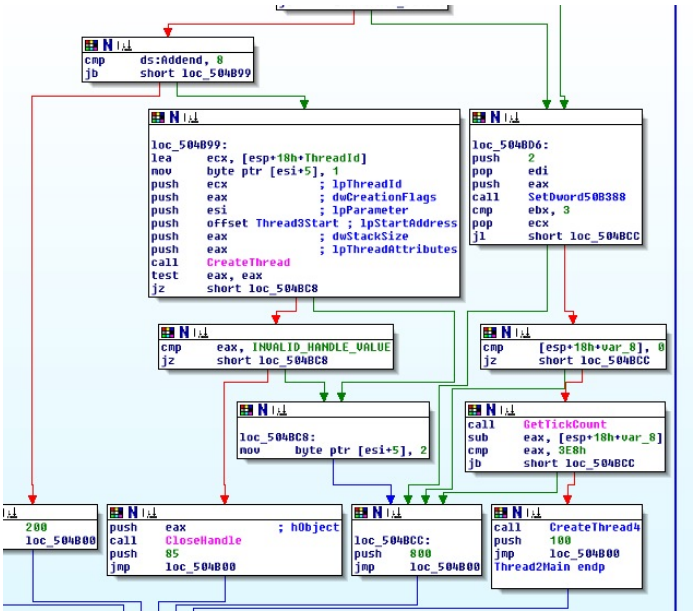
Kuvio 49. MyDoom-haittaohjelman funktio sub_50565B, jota pääsäie kutsuu ikuisessa silmukassa sekunnin välein. Kuvakaappaus IDA Pro -disassemblerista.



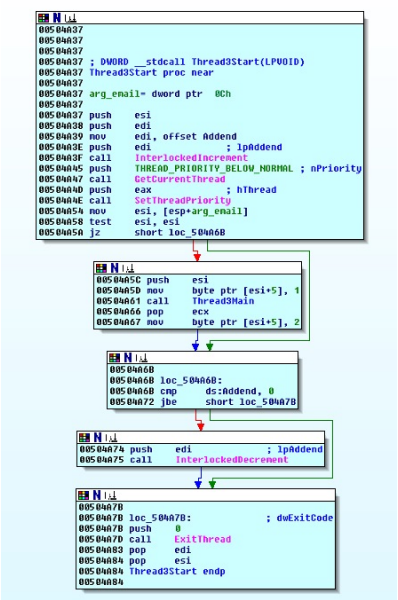
Kuvio 50. Verkkoyhteyden tarkastus MyDoom-haittaohjelman säikeessä. Jos verkkoyhteyttä ei tunnisteta, ohitetaan huomattava määrä ohjelmakoodia. Kuvakaappaus IDA Pro -disassemblerista.



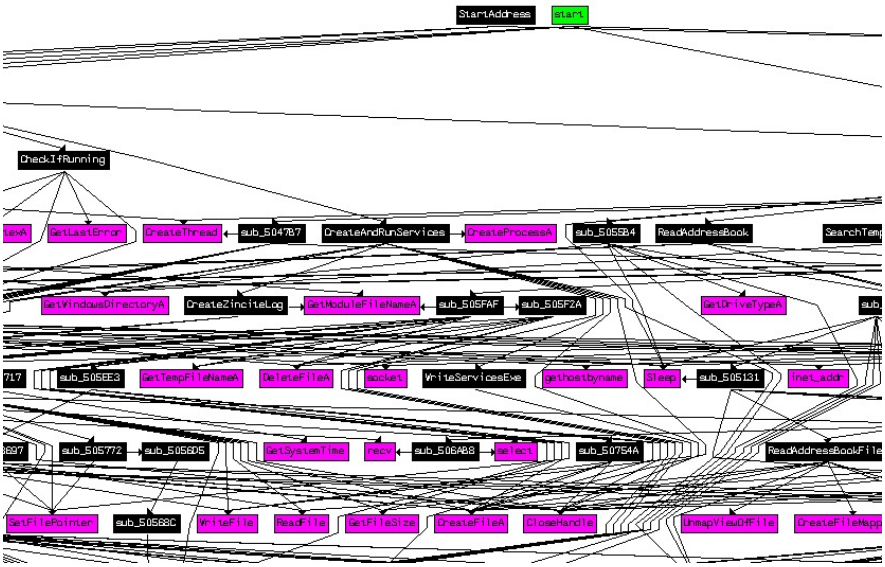
Kuvio 51. Heksadesimaalinäkymä hyppykäskystä. Tilarivillä näkyvä osoite 00004B0D on fyysinen osoite tiedostossa. Korostetut tavut nollakäskyillä 90 korvaamalla hyppykäsky saadaan poistettua. Kuvakaappaus IDA Pro -disassemblerista.



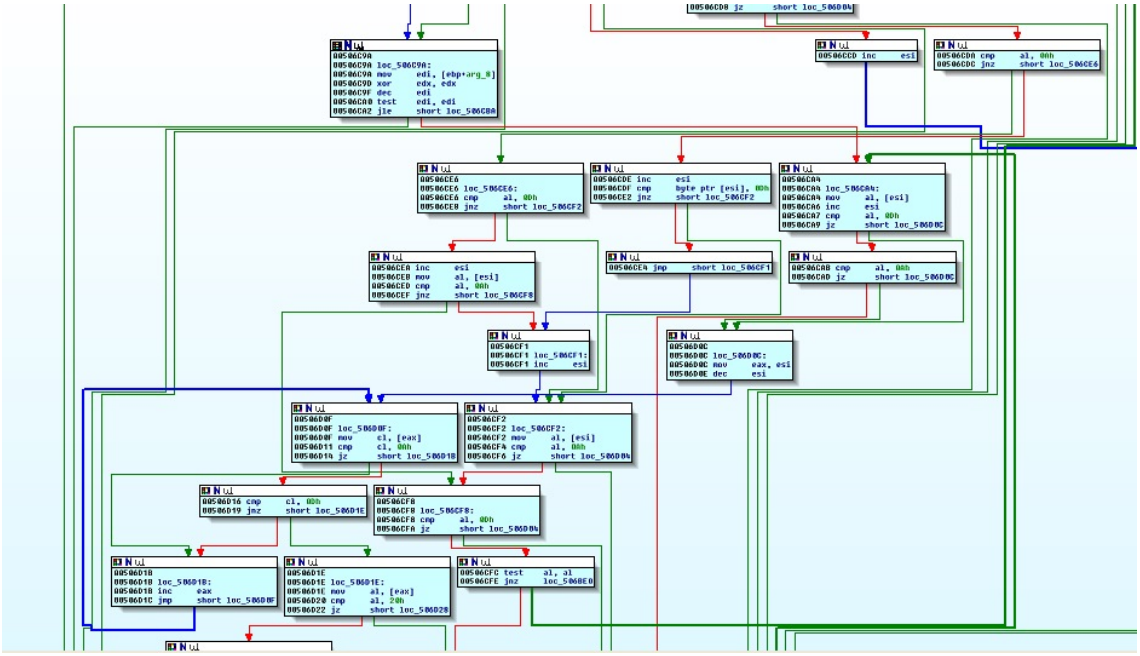
Kuvio 52. Säikeen 2 aliohjelmaa Thread2Main, jossa käynnistetään säikeitä 3. Kuvakaappaus IDA Pro -disassemblerista.



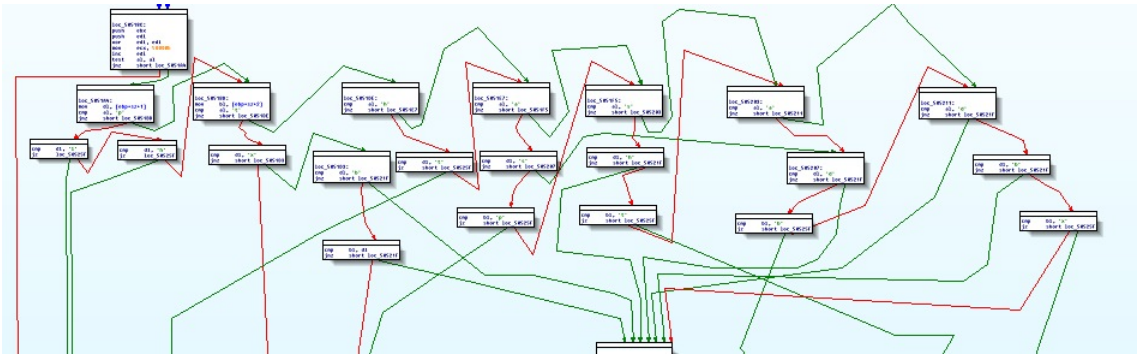
Kuvio 53. Säikeen 3 pääohjelma Thread3Start, jossa kasvatetaan laskuria Addend säikeiden lukumäärän kasvaessa ja kutsutaan aliohjelmaa Thread3Main. Kuvakaappaus IDA Pro -disassemblerista.



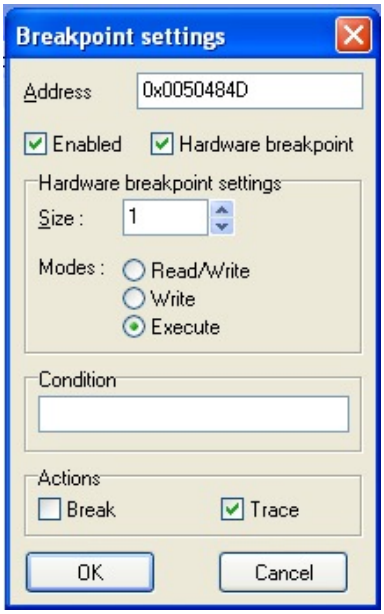
Kuvio 54. MyDoom-haittaohjelman funktioriippuvuuskaavio. Kuvakaappaus IDA Pro -disassemblerista.



Kuvio 55. MyDoom-haittaohjelman funktio sub_506C46. Kuvakaappaus IDA Pro -disassemblerista.



Kuvio 56. Tiedoston tunnisteen vertailu merkki kerrallaan MyDoom-haittaohjelman tiedostojen prosessointifunktiosta. Vertailut on siirretty käsin kolmeen tasoon kunkin kirjaimen mukaan. Kuvakaappaus IDA Pro -disassemblerista.



Kuvio 57. Keskeytyskohta, joka ei pysäytä suoritusta, mutta laskee kuinka monta kertaa siitä on menty läpi. Kuvakaappaus IDA Pro -disassemblerista.

```
00178A37 db    0
00178A38 aFromBlog@practicalmalwar db 'From: blog@practicalmalwareanalysis.com',0Dh,0Ah
00178A38 db 'To: blog@practicalmalwareanalysis.com',0Dh,0Ah
00178A38 db 'Subject: Delivery reports about your e-mail',0Dh,0Ah
00178A38 db 'Date: Thu, 15 May 2014 11:45:16 +0300',0Dh,0Ah
00178A38 db 'MIME-Version: 1.0',0Dh,0Ah
00178A38 db 'Content-Type: multipart/mixed;',0Dh,0Ah
00178A38 db 9,'boundary="----=_NextPart_000_0010_4412F3D9.496EA41E"',0Dh,0Ah
00178A38 db 'X-Priority: 3',0Dh,0Ah
00178A38 db 'X-MSMail-Priority: Normal',0Dh,0Ah
00178A38 db 'X-Mailer: Microsoft Outlook Express 6.00.2600.0000',0Dh,0Ah
00178A38 db 'X-MimeOLE: Produced By Microsoft MimeOLE V6.00.2600.0000',0Dh,0Ah
00178A38 db 0Dh,0Ah
00178A38 db 'This is a multi-part message in MIME format.',0Dh,0Ah
00178A38 db 0Dh,0Ah
00178A38 db '------=_NextPart_000_0010_4412F3D9.496EA41E',0Dh,0Ah
00178A38 db 'Content-Type: text/plain;',0Dh,0Ah
00178A38 db 9,'charset=us-ascii',0Dh,0Ah
00178A38 db 'Content-Transfer-Encoding: 7bit',0Dh,0Ah
00178A38 db 0Dh,0Ah,'4'
00178C8F db 0FAh ; ·
00178C90 db  85h ; à
```

Kuvio 58. Aliohjelman sub_504971 palauttamia SMTP-otsikkotietoja. Kuvakaappaus IDA Pro -disassemblerista.