Win32-tiedostovirukset ja niiden havaitseminen – staattisen rakenteellisen heuristiikan rajoitukset ja mahdollisuudet

TURUN YLIOPISTO
Informaatioteknologian laitos
Tietojenkäsittelytieteet
Pro gradu -tutkielma
Mikko Suominen
Helmikuu 2007

Tarkastajat:
Seppo Virtanen
Jukka Teuhola

TURUN YLIOPISTO
Informaatioteknologian laitos

SUOMINEN, MIKKO: Win32-tiedostovirukset ja niiden havaitseminen – staattisen rakenteellisen heuristiikan rajoitukset ja mahdollisuudet

Pro gradu -tutkielma, 90 s. + 1 liites.
Tietojenkäsittelytiede
Helmikuu 2007

Tietokoneille on 1980-luvulta alkaen kehitetty monentyyppisiä haittaohjelmia. Yksi haittaohjelmatyyppi ovat tiedostovirukset. Ne ovat itseään monistavia ohjelmia, jotka käyttävät muita ohjelmatiedostoja leviämiseensä liittämällä niihin kopion itsestään ja huolehtimalla siitä, että kopio tulee suoritetuksi isäntäohjelmansa mukana. Tiedostovirukset pyrkivät vaikeuttamaan löytämistään muuttamalla itseään salaamalla, poly- tai metamorfismilla ja piilottamalla sijaintinsa EPO-menetelmillä (Entry Point Obscuring).

Kaikki Win32-pohjaiset käyttöjärjestelmät käyttävät samaa PE-tiedostomuotoa (Portable Executable), jonka rakenne on tarkkaan määritelty. Koska rakenne on tarkkaan määritelty, voidaan loisvirustartunta pyrkiä havaitsemaan ohjelmatiedoston poikkeuksellisten ominaisuuksien tai epäilyttävien rakenteiden perusteella. Tutkielmassa esitetty staattinen rakenteellinen heuristiikka koostuu yksittäisistä testeistä, jotka on yhdistetty virustartunnat tunnistaviksi sääntöjoukoiksi. Yksittäiset testit liittyvät tulokohdan sijaintiin ja sisältöön, osioiden ominaisuuksiin, osioiden kokoon liittyvien kenttien tarkistukseen, ohjelman hyödyntämien Windows-funktioiden analysointiin ja otsakkeiden tarkistukseen tartuntamerkkien varalta. Yksittäisistä testeistä luotiin sääntöjoukkojen perusta 16:a tiedostovirusta analysoimalla. Tämän jälkeen sääntöjoukkoja parannettiin 5334:n tiedostoviruksilla saastuneen tiedoston satunnaisotoksella.

Testeissä heuristiikka pystyi löytämään 62.4 prosenttia tiedostoviruksista. Vääriä positiivisia tuloksia syntyi kahdessa eri testiympäristössä 0.28 prosenttia ja 0.09 prosenttia. Itsekoodauksella tai poly- ja metamorfismilla ei ole suoranaista vaikutusta staattisen rakenteellisen heuristiikan toimintaan. Staattinen rakenteellinen heuristiikka löytää luotettavasti isäntäohjelman loppuun itsensä lisäävät loisvirukset, jotka muuttavat tulokohdan osoittamaan itseensä, otsakevirukset ja uuden PE-otsakkeen lisäävät virukset. Muuhun kuin isäntäohjelman viimeiseen osioon itsensä lisäävät loisvirukset jäävät heuristiikalta löytämättä, tosin tulokohdassa mahdollisesti oleva hyppykäsky saattaa riittää paljastamaan kyseiset virukset. Isäntäohjelman alkuun itsensä lisäävien virusten löytäminen ilman runsasta määrää vääriä positiivisia on heuristiikalla ongelmallista ja vaatisi vähintään asennus- ja poisto-ohjelmien tunnistamista ja erityiskohtelua. Joka tapauksessa osa isäntäohjelman alkuun tarttuvista loisviruksista on mahdollista löytää ilman suuria määriä vääriä positiivisia tuloksia. IAT:n (Import Address Table) korvaavien EPO-virusten löytämiselle tutkielmassa esitetty heuristiikka tarjoaa erittäin nopean menetelmän. Muita EPO-menetelmiä käyttäviä loisviruksia ei ole mahdollista löytää staattisella rakenteellisella heuristiikalla.

Asiasanat: tietokonevirukset, virustentorjuntaohjelmat, heuristiikka

UNIVERSITY OF TURKU
Department of Information Technology

SUOMINEN, MIKKO: Searching for Win32 File Infectors – The Possibilities and Limitations of Static Structural Heuristics
Master's Thesis, 90 p. + 1 app. p.
Computer Science
February 2007

Since the 1980s many different types of malicious computer software have been developed. File infectors are one such program type. File infectors are self-replicating programs, that use other program files as a host when spreading. File infectors add a functional copy of themselves to host files and make sure the copy is executed together with the host program. File infectors try to hide by transforming themselves from one infection to another with self-encryption, poly- and metamorphism and by hiding their location by using EPO-methods (Entry Point Obscuring).

All Win32-based operating systems utilise the same PE file format (Portable Executable) for their program files. Because the structure of PE files is exactly defined, it can be used to detect parasitic virus infections by looking for suspicious properties and structures inside files. The static structural heuristics developed in this thesis consist of individual tests, which are combined into rule sets that detect virus-infected files. The individual tests examine the content and location of the entry point, section properties, size-related fields of headers, Windows functions used by the program and search the headers for infection markers. The basis for the rule sets was developed by analysing 16 parasitic Win32 viruses. The rule sets were then improved by analysing a randomly selected sample of 5334 virus-infected program files.

Based on testing, the heuristic is able to find 62.4 percent of parasitic Win32 viruses. The amount of false positives was in two different systems 0.28 percent and 0.09 percent. Self-encryption or poly- or metamorphism has no direct effect on the effectivness of a static structural heuristic. The heuristics are able to reliably find appending viruses which replace the address of the entry point with their own starting address, header viruses and viruses that add a new PE header. If the virus adds itself to some other section than the last one, the heuristic is possibly able to find the virus only if the program has a jump command at its entry point. Finding prepending viruses without a considerable amount of false positives is difficult and would require identifying and separately handling installer and uninstaller programs. In any case it is possible to find only a part of prepending viruses without too many false positive results. The heuristic developed in this thesis offers a fast way of finding EPO viruses which replace the IAT (Import Address Table). Viruses that use other EPO methods can not be found with a static structural heuristic.

Keywords: computer virus, anti-virus software, heuristics

SISÄLLYS

1 JOHDANTO 3
  1.1 Tietokonevirukset ja niiden eri tyypit 3
  1.2 Tietokonevirusten aiheuttamat vahingot 6
  1.3 Tietokonevirusten määrä ja nimeäminen 7
  1.4 Tutkielman rajaus ja tavoitteet 9
2 TIEDOSTOVIRUKSET 12
  2.1 Johdanto 12
  2.2 Tiedostovirustyypit 12
    2.2.1 Päällekirjoittavat virukset 12
    2.2.2 Loisvirukset 13
    2.2.3 Onkalovirukset 14
  2.3 Edistyneet tartuntamenetelmät 15
    2.3.1 Pakkaus 15
    2.3.2 Itsekoodaus 16
    2.3.3 Polymorfismi 17
    2.3.4 Metamorfismi 23
    2.3.5 Tulokohdan kätkeminen (EPO) 26
  2.4 Yhteenveto tiedostoviruksista 28
3 WIN32-TIEDOSTOVIRUKSET 30
  3.1 Johdanto 30
  3.2 Win32-käyttöympäristö 30
  3.3 Portable Executable -tiedostomuoto 31
    3.3.1 Yleistä PE-tiedostojen rakenteesta 31
    3.3.2 Otsakkeet 32
    3.3.3 Osiotaulu 35
    3.3.4 Tuontiosoitetaulu 36
  3.4 Win32-tiedostovirusten tartuntamenetelmät 38
  3.5 Win32-tiedostovirusten EPO-menetelmät 40
  3.6 Yhteenveto Win32-tiedostoviruksista 42
4 TIEDOSTOVIRUSTEN ETSIMINEN 43
  4.1 Johdanto 43
  4.2 Merkkijonohaut 43
    4.2.1 Yksinkertaiset merkkijonohaut 43
    4.2.2 Tunnistustarkkuuden parantaminen 47
  4.3 Algoritmiset menetelmät 48
    4.3.1 Yleistä algoritmisista menetelmistä 48
    4.3.2 Suodatus 49
  4.4 X-RAY-menetelmät 50
  4.5 Koodin emulointi 55
  4.6 Heuristiset menetelmät 57
  4.7 Yhteenveto tiedostovirusten etsimisestä 62
5 STAATTISET RAKENTEELLISET HEURISTIIKAT WIN32-TIEDOSTOVIRUSTEN ETSIMISEEN 64
  5.1 Kehitettävien heuristiikkojen määrittely ja tavoitteet 64
  5.2 Heuristiikka EPO-menetelmiä käyttämättömille viruksille 65
    5.2.1 Epäilyttävien tulokohtien etsiminen 65

5.2.2 Epäilyttävät osioiden ominaisuudet 68
5.2.3 Otsakkeissa olevien kokoon liittyvien kenttien tarkistaminen....................... 69
5.2.4 Käyttöjärjestelmän muistialueelle ladattavat osiot 71
5.2.5 Viruksiin viittaava APIen käyttö.. 71
5.2.6 Otsakkeiden tarkistus tartuntamerkkien varalta 72
5.2.7 Väärä tarkistussumma 73
5.2.8 Heuristiikan sääntöjoukot 74
5.3 Heuristiikka EPO-viruksien löytämiseksi 75
5.3.1 Vertailu tavalliset tiedostovirukset löytävään heuristiikkaan 75
5.3.2 IAT:n korvaavat EPO-virukset 76
5.3.3 Heuristiikan sääntöjoukot 78
5.4 Heuristiikan testaus ja täydentäminen suurella virusjoukolla 78
5.5 Eri tartuntatekniikoiden vaikutus heuristiikan toimintaan 80
6 YHTEENVETO 84
LÄHTEET.. 87
Liite 1: Heuristiikkojen pohjana käytetyt virukset... 91

1 JOHDANTO

1.1 Tietokonevirukset ja niiden eri tyypit

Tietokonevirukset ovat toimintaperiaatteeltaan pitkälti samanlaisia kuin biologiset vastineensa: ne tarttuvat isäntiin, joita virukset käyttävät välikappaleina elämälleen. Virusten elämäntarkoituksena on leviäminen uusiin isäntiin elämänsä jakamiseksi. Biologiset virukset käyttävät isäntinään eliöitä, tietokonevirukset taas yleisimmin tiedostoja.

Ensimmäisen määritelmän tietokoneviruksille antoi Cohen vuonna 1984: ”A virus is a program that is able to infect other programs by modifying them to include a possibly evolved copy of itself.” (Cohen 1984) Määritelmä perustuu Cohenin luomaan tietokoneviruksen matemaattiseen malliin, eikä sanallisessa muodossaan ole täydellinen, koska se ei kata esimerkiksi kumppaniviruksia. Kumppanivirukset eivät tartu tiedostoihin, vaan käyttöympäristön ominaisuuksia hyödyntämällä saavat ajettua virusohjelman käyttäjän tarkoittaman ohjelman sijaan. Myöskään virusten leviämisen kannalta oleellista rekursiota ei Cohenin määritelmässä kerrota eksplisiittisesti. Edellisten huomioiden pohjalta Szor on tarjonnut oman määritelmänsä: ”A computer virus is a program that recursively and explicitly copies a possibly evolved version of itself.” (Szor 2005 s. 19) Vielä yksityiskohtaisempi määritelmä on Bontchevin Seborgilta lainaama:"We define a computer 'virus' as a self-replicating program that can 'infect' other programs by modifying them or their environment such that a call to an 'infected' program implies a call to a possibly evolved, and in most cases, functionally similar copy of the 'virus'." (Bontchev 1994)

Szorin ja Seborgin määritelmiin pohjautuen voidaan tiedostovirukset tämän tutkielman tarpeisiin määritellä seuraavasti: tiedostovirus on ohjelma, joka rekursiivisesti ja eksplisiittisesti kopioi mahdollisesti kehittyneen version itsestään toisen ohjelman (isäntäohjelma) osaksi niin, että viruksen tartuttaman isäntäohjelman suoritus johtaa viruksen suoritukseen.

Virukset voivat tarttua ohjelmatiedostojen lisäksi tiettyihin datatiedostoihin (makrovirukset) tai levyjen käynnistyslohkoihin. Viruksien sisällä madot lasketaan usein omaksi virustyypikseen, kuten myös troijalaiset.

Makroviruksen tekevät mahdolliseksi sovellukset, joilla myös tavallisiin datatiedostoihin (esim. tekstidokumentteihin) voidaan sisällyttää toiminnallisuutta. Makrovirukset ovat suhteellisen uusi virustyyppi, ensimmäinen makroviruksen mahdolliseksi osoittanut virus WM/Concept ilmestyi 1995. Makrovirukset yleistyivät niiden tekemisen helppouden vuoksi nopeasti.

Käynnistyslohkovirukset tarttuvat nimensä mukaisesti tiedostojen sijaan levyjen alussa olevaan käynnistyslohkoon. Käynnistyslohkossa on käyttöjärjestelmän sisältävän levyosion löytämiseen ja sen latauksen valmisteluun tarvittava lyhyt ohjelma, joka suoritetaan aina tietokoneen käynnistyessä. Käynnistyslohkovirus korvaa käyttöjärjestelmän lataajan (boot-strap loader) ja pääsee näin ollen kopioimaan itsensä muistiin, kun tietokone käynnistetään. Käynnistyslohkovirukset tallentavat yleensä alkuperäisen käynnistyslohkon jonnekin muualle levyllä, ja suorittavat sen käyttöjärjestelmän käynnistämiseksi keskusmuistiin kopioiduttuaan. Käynnistyslohkovirukset pysyvät muistissa myös käyttöjärjestelmän latauduttua ja leviävät tartuttamalla saastuneessa koneessa käytettävät muut levyt. Ensimmäinen käynnistyslohkovirus oli 1982 tehty Elk Cloner, joka on samalla myös ensimmäinen tunnettu levinnyt PC-tietokonevirus. Ensimmäinen IBM PC -tietokoneisiin tarttunut virus, Brain, oli myös käynnistyslohkovirus, ja lähti leviämään vuonna 1986. Vaikka käynnistyslohkovirukset olivat tietokonevirusten alkuaikoina hyvin yleisiä, ei niitä enää vuoden 2001 jälkeen ole käytännössä tavattu (Bridwell 2005, s. 15). Käynnistyslohkovirusten katoamiseen on syynä niiden leviämiseen vaadittavien levykkeiden käytön raju väheneminen. Virukset voivat olla samanaikaisesti sekä tiedosto- että käynnistyslohkoviruksia. Tällaisten moniosiovirusten (multipartite) poistaminen on yleensä yhdessä tartuntamuodossa pitäytyviä viruksia hankalampaa. Ensimmäinen moniosiovirus oli 1989 tehty Ghostball.

Nopeimmin leviävä virustyyppi ovat madot. Madon määritteleminen ei ole aivan yksiselitteistä, mutta madoilla on muutamia niille tyypillisiä ominaisuuksia, jotka erottavat madot tavallisista viruksista. Tärkein ero on leviämistavassa: madot leviävät itsenäisesti tietoverkkoja pitkin toisiin tietokoneisiin, kun tavalliset virukset taas leviävät tiedostosta toiseen tietokoneen sisällä. Lisäksi madot eivät välttämättä tartu ohjelmatiedostoihin, vaan mato-ohjelma saattaa olla oma itsenäinen ohjelmatiedostonsa tai levitä jopa täysin muistinvaraisena. Ensimmäiseksi todelliseksi madoksi katsotaan yleensä vuonna 1988 liikkeelle luikerrellut Morris-mato. Jotkin madot myös hyväksikäyttävät ohjelmistoissa tai käyttöjärjestelmissä olevia haavoittuvuuksia, joiden avulla ne voivat levitä ilman ihmisen myötävaikutusta, toisin kuin tavalliset virukset, jotka levitäkseen tarvitsevat ihmistä käynnistämään saastuneen ohjelman. Nykyään madoille tyypillinen leviämistapa on sähköpostin liitetiedostona leviäminen. Vuonna 1999 maailmalle ilmaantunut W97M/Melissa oli ensimmäinen massapostittajamato, se levisi massapostittajamadoille tyypillisesti hyvin nopeasti kolmessa päivässä 100000 kohteeseen (Chen 2003).

Älypuhelimet mahdollistavat matojen toteuttamisen myös mobiililaitteissa. Vuoden 2004 kesäkuussa ilmestyikin ensimmäinen mobiililaitteille tehty mato, SymbOS/Cabir (Ferrie ym. 2004). SymbOS/Cabir tarttuu Symbian Series 60 -käyttöjärjestelmillä varustettuihin mobiililaitteisiin ja leviää bluetooth-yhteyden välityksellä. Bluetoothin lisäksi mobiilimadot leviävät nykyisin myös mms-viestien välityksillä. Toistaiseksi mobiilimadot ovat suhteellisen harmittomia, esimerkiksi SymbOS/Cabirin ainoa haittavaikutus on akun nopeahko tyhjeneminen bluetooth-yhteyden aktiivisen käytön myötä. Mahdollisuus rahastustarkoituksessa tehdyille madoille (esimerkiksi kalliita mms-viestejä lähettämällä) saattaa tulevaisuudessa tehdä myös mobiililaitteiden virussuojasta välttämättömyyden.

Troijalaiset ovat ohjelmia, jotka päällepäin vaikuttavat harmittomilta. Käynnistyessään ne kuitenkin suorittavat lisäksi jonkin käyttäjänsä kannalta salatun, haitallisen toiminnon. Usein tuo salattu toiminto on takaportin asentaminen koneeseen, jolloin troijalaisen tekijä saa vapaan pääsyn uhrien tietokonejärjestelmiin. Takaportti on usein yhdistetty näppäinlokiin, joka tallentaa käyttäjän kirjoittamat salasanat pahantekijän

luettaviksi. Troijalaiset eivät itsenäisesti monistu, vaan antavat uhrien itse hoitaa troijalaisen levityksen houkutuslintuna toimivan hyötyohjelman asentamalla. Koska troijalaiset eivät monistu itsenäisesti, eivät ne varsinaisesti ole viruksia. Niille tyypillisiä piirteitä on kuitenkin usein yhdistetty varsinaisiin viruksiin, tai virus tai mato saattaa tarttuessaan jättää jälkeensä troijalaisen. Muutenkin virukset saattavat sisältää useiden eri virustyyppien piirteitä: virus saattaa levitä madon lailla tietoverkkoja pitkin, mutta samalla levitä myös tietokoneen sisällä tiedostosta toiseen.

1.2 Tietokonevirusten aiheuttamat vahingot

Vaikka virukset päätyvät tiedotusvälineisiin aiheuttamiensa vahinkojen vuoksi, ei suurimman osan viruksista ole tarkoitus aiheuttaa vahinkoa, vaan niiden ainoana tehtävänä on leviäminen. Niissä viruksissa, joilla on muukin tarkoitus kuin leviäminen, ei lataus (payload) useimmiten aiheuta pysyvää vahinkoa. Tyypillisin lataus vain tulostaa jonkin tekstin tai kuvan näytölle. Virusten aiheuttamat vahingot ovat useimmiten poistettuja tai korruptoituneita tiedostoja tai kokonaisia levyosioita. Jotkut virukset korruptoivat levyllä olevaa dataa hitaasti pitkän ajan kuluessa, jolloin ne yleensä säilyvät kauemmin havaitsemattomana ja ehtivät paremmin levitä. Tällaisten hitaasti tuhojaan tekevien virusten seurauksena myös varmuuskopiot saattavat olla hyödyttömiä ellei niitä ole olemassa pidemmältä ajalta. Latauksen sisältävillä viruksilla on erityinen liipaisin, joka määrittelee milloin lataus laukeaa. Liipaisin on tyypillisesti jokin tietty päivämäärä tai kellonaika. Liipaisin on välttämätön viruksen leviämisen kannalta – mikäli virus suorittaisi latauksensa heti tartuttuaan, se havaittaisiin nopeasti eikä se ehtisi leviämään. Vaikkei valtaosaa viruksista ole tarkoitettu vahingollisiksi, ei se valitettavasti tarkoita, etteivätkö ne silti poikkeuksetta olisi haitallisia. Vaikkei viruksen tekijä alun perin olisi tarkoittanut virustaan tuhoisaksi saattaa se ohjelmointivirheen takia silti aiheuttaa datan menetystä ja heikentää järjestelmän vakautta. Virukset myös kuluttavat aina tietokoneen resursseja ja heikentävät siksi järjestelmän suorituskykyä. Erityisesti nykyään valokeilassa viihtyvät sähköpostitse leviävät virukset kuormittavat yritysten sähköpostipalvelimia ja tietoverkkoja sekä aiheuttavat suuria menetyksiä tuottavuudessa. Erityisen suuria taloudellisia tappioita ovat viime vuosina aiheuttaneet joko sähköpostitse tai erilaisten ohjelmistoissa olevien haavoittuvuuksien avulla levinneet madot. Virusten aiheuttamia tarkkoja taloudellisia vahinkoja on vaikea arvioida, mitä kuvaavat hyvin VBS/LoveLetter@mm:n

vahinkoarviot, jotka vaihtelevat 100 miljoonasta aina 10 miljardiin dollariin (Hinde 2000). Joka tapauksessa virusten aiheuttamat vahingot ovat nykyisessä pitkälti tietokoneistetussa ja tietoverkkoihin nojautuvassa yhteiskunnassa mittavia.

1.3 Tietokonevirusten määrä ja nimeäminen

Virusten alkuaikoina uusia viruksia ilmestyi niin paljon, että erilaisten virusten määrä kaksinkertaistui noin joka yhdeksäs kuukausi. Uusien virusten ilmestymisnopeus alkoi kuitenkin lähestyä lineaarista 90-luvun puoliväliin mennessä (Bontchev 1997). Vaikkei uusien virusten ilmestymisnopeuden kasvu ole enää aivan alkuvuosien huippulukemissaan, tehdään uusia viruksia silti vuosi vuodelta entistä enemmän. Uusien virusten tulva aiheuttaa virustentorjuntaohjelmistojen kehittäjille suuria ongelmia, koska ohjelmistot täytyisi saada nopeasti päivitettyä löytämään kaikki uudet virukset ja suuret määrät jatkuvasti analysoinnin tarpeessa olevia uusia viruksia hidastavat reaktionopeutta uusiin uhkiin.

Vuonna 2004 erilaisia viruksia oli arviolta yli 70000 (Bridwell 2004 s. 7). Virusten tarkkaa lukumäärää on mahdotonta sanoa ja sen määräävät pitkälti se kuinka pienet, mahdollisesti jopa viruksen toiminnallisuuteen vaikuttamattomat, erot katsotaan riittäviksi virusten yksilöintiin. Toisena virusmäärään vaikuttavana tekijänä on se, mitkä kaikki haittaohjelmat lasketaan viruksiksi. Joka tapauksessa viruksia on tähän päivään mennessä ehditty tekemään kymmeniä tuhansia. Valtaosaa viruksista ei enää tavata tavallisilla käyttäjillä, vaan ne ovat kadonneet toiminta-alustojensa mukana, tai eivät alun perinkään levinneet merkittävästi. Leviämisen estää usein viruksen toimintaa rajoittava tai kokonaan estävä ohjelmointivirhe. Vapaana olevia (tavallisilla tietokoneen käyttäjillä normaalikäytössä tavattuja, eikä ainoastaan tutkijoiden kokoelmissa löytyviä) viruksia oli helmikuussa 2006 731 erilaista (WildList Organization International 2006). Luku perustuu anti-virustutkijoiden tietoon tulleisiin varmoihin virushavaintoihin, joissa sama virus on tavattu vähintään kahdessa eri kohteessa yhden kuukauden sisällä. WOIn luvut eivät ole tarkkoja järjestelmässä olevien epäkohtien vuoksi (Bontchev 1999), mutta antavat silti ainakin jonkinlaisen kuvan siitä kuinka liikkeellä olevien virusten määrä suhteutuu koko virusmäärään.

Virusten ja muiden haittaohjelmien nimeämiseen ei ainakaan vielä ole yhteistä standardia. Virustentorjuntaohjelmiston valmistajien ja tutkijoiden nimeämiskäytännöissä on kuitenkin paljon yhtäläisyyksiä. Yleensä nimi koostuu viruksen vaatimasta alustasta (useimmiten käyttöjärjestelmä tai sovellusohjelma), selkokielisestä nimestä ja mahdollisesti variantista. Lisäksi nimeen saattaa kuulua viruksen koko, tyyppi (virus, mato, troijalainen jne.) ja leviämistapa (sähköposti, vertaisverkot, irc jne.). Eri yritysten käyttämät kentät ovat pitkälti samat ja muistuttavat usein CARO:n (Computer Antivirus Research Organisation) yritystä luoda epävirallinen standardi haittaohjelmien nimeämiseen. Tarkemman selostuksen CARO:n nimeämiskäytännöstä ja sen nykytilasta on antanut Bontchev (2005). Vaikka eri tahojen käyttämät tiedot nimissä ovatkin pitkälti samat, on ongelmana eri yritysten samalle virukselle antamien nimien koordinointi. Yritykset pyrkivät sopimaan yhteisestä nimestä tai variantista ennen löydöksen lisäämistä ohjelmistoihin. Varsinkin vaarallisimmista viruksista antivirusyritykset jakavat nopeasti näytteet kilpailijoilleen leviämisen hillitsemiseksi, ja samalla alkuperäisen löytäjän virukselle antama nimi tulee useimmiten yleiseen käyttöön (Bontchev 2004). Uusia kriittisiä uhkia ilmenee kuitenkin usein, ja koska antivirusohjelmistot täytyy päivittää nopeasti, ei aikaa nimestä sopimiseen välttämättä ole. Taulukossa 1 on esimerkkinä eräästä Sober-madon variantista käytettyjä nimiä. Käyttöjärjestelmä (Win32 tai lyhyemmin W32) ja viruksen tarkentaminen madoksi tai tarkemmin massapostittajaksi (@mm, mass-mailer) ovat yleisesti mukana viruksen nimessä. Varianttien nimeämisen koordinoinnin vaikeudesta taulukko antaa hyvän kuvan – kymmenen eri yritystä käyttää samasta variantista yhteensä seitsemää eri kirjaintunnistetta.

Common Malware Enumeration (CME) on vuonna 2004 alkunsa saanut yritys luoda järjestelmä yritysten viruksille antamien nimien koordinoinnille. CME ei pyri korvaamaan yritysten viruksista käyttämiä nimiä, vaan se antaa viruksille yritysten käyttämien nimien lisäksi yhteisen tunnisteen. Tunniste koostuu merkkijonosta CME, jota seuraa väliviivan jälkeen kokonaisluku (esimerkiksi CME-681). Muutamat merkittävät yritykset ovat jo lisänneet CME-tunnisteet viruksista käyttämiinsä nimiin tai muuhun viruksiin liittyvään materiaaliin kuten virustiedotteisiin. Symantec ja McAfee ovat käyttäneet CME-tunnisteita huhtikuusta 2005, F-Secure lokakuusta 2005 ja

Microsoft helmikuusta 2006. CME-tunniste on kuitenkin olemassa vain erittäin pienelle osalle viruksia (6. huhtikuuta 2006 tunniste oli ainoastaan 32 virukselle tai variantille).

Taulukko 1. Yhdestä Sober-madon variantista eri yritysten käyttämiä nimiä (Common Malware Enumeration 2006).

CA	Win32.Sober.W
F-Secure	Sober.Y
Kaspersky	Email-Worm.Win32.Sober.y
McAfee	W32/Sober@MM!M681
Microsoft	Win32/Sober.Z@mm!CME-681
Norman	W32/Sober.AA@mm
Panda	W32/Sober.AH.worm
Sophos	W32/Sober-Z
Symantec	W32.Sober.X@mm
Trend Micro	WORM_SOBER.AG

1.4 Tutkielman rajaus ja tavoitteet

Käyttöjärjestelmämarkkinoilla vallinnut Microsoftin monopoliasemaa lähennellyt tilanne on tarjonnut viruksille hyvän mahdollisuuden levitä. Kun ylivoimainen enemmistö tietokoneista käyttää pääasiassa yhden valmistajan käyttöjärjestelmiä, on viruksille tarjolla runsaasti tartuntakelpoisia kohteita. Viruksia on toki tehty useimmille kehitetyille laitealustoille, mutta niille tehtyjen virusten määrä on vain murto-osa PC-tietokoneille ja Microsoftin käyttöjärjestelmille tehdyistä viruksista. Microsoftin kaikki nykyiset käyttöjärjestelmät kattavan Win32-alustan ollessa selkeästi yleisin virusten tartunta-alusta, rajataan virusten tarkempi käsittely tässä tutkielmassa juuri Win32-ohjelmatiedostoihin tarttuviin loisviruksiin. Win32-alustalle olemassa olevia datatiedostoihin tarttuvia makroviruksia, matoja tai muita haittaohjelmia ei tutkielmassa jatkossa käsitellä.

Virusten tiedostoista etsimisessä ongelmana on useamman toisilleen vastakkaisen tavoitteen yhteensovittaminen. Viruksista on löydettävä mahdollisimman suuri osa, mutta samalla väärien hälytysten lukumäärän täytyy olla mahdollisimman lähellä nollaa.

Väärät hälytykset on pyrittävä minimoimaan, koska ne heikentävät käyttäjän uskoa virustentorjuntaohjelmaan ja saattavat jopa johtaa siihen, että käyttäjä jättää huomioimatta myös oikeat viruslöydökset. Tarkkuus yksinään ei olisi toivottoman vaikeaa saavuttaa, mutta lisäksi etsinnän täytyy olla myös nopeaa.

Uusien Win32-virusten nykyisen nopean ilmestymisvauhdin hallitsemiseen ovat yhtenä mahdollisuutena heuristiset menetelmät, jotka mahdollistavat uusien virusten havaitsemisen ilman virustentorjuntaohjelmiston päivitystä. Perinteisesti heuristiset ratkaisut ovat pyrkineet löytämään virukset etsimällä niille tyypillistä toimintaa joko staattisesti ohjelmatiedostoista tai dynaamisesti suorituksen aikana. Tässä työssä kehitettävät heuristiikat tarkastelevat toiminnallisuuden sijaan tiedostojen rakennetta. Rakennetta tarkasteltaessa tiedostoista tarvitsee käydä läpi huomattavasti pienempi osa verrattuna ohjelman toiminnan tutkimiseen. Levyltä tapahtuvien lukujen vähentäminen nopeuttaa heuristiikan toimintaa tuntuvasti, koska levyoperaatiot ovat useimmiten tietojenkäsittelyssä pullonkaulana. Tiedoston rakenteeseen perustuvan heuristiikan periaatteesta on kirjoittanut Szor (Szor 2005 s. 467). Siitä minkälaisia tiedostoviruksia staattisella rakenteellisella heuristiikalla on mahdollista löytää, ei ole kirjoitettu. Tämän tutkielman tavoitteena on selvittää, minkä tyyppiset Win32-tiedostovirukset on mahdollista löytää staattisella rakenteellisella heuristiikalla nopeasti, runsaita määriä vääriä positiivisia tuloksia aiheuttamatta (väärät positiiviset tulokset ovat heuristisille menetelmille tyypillinen ongelma). Eri virustyypeistä käsitellään sekä luvussa 2 esitettävät yleisen tason tartuntamenetelmät että luvussa 5 esiteltävät tiedostojen rakenteeseen liittyvät tartuntatavat. Staattisen rakenteellisen heuristiikan mahdollisuuksien ja rajoitusten selvittämiseksi kehitetään tässä tutkielmassa kaksi uutta heuristiikkaa Win32-tiedostovirusten havaitsemiseksi. Heuristiikan lopullisen suorituskyvyn testaamiseen tarjoutui tilaisuus F-Securen tiloissa Helsingissä, jolloin testijoukoksi saatiin kattava 5334:n loisviruksella saastuneen tiedoston joukko.

Heuristiikkoja kehitetään kaksi kappaletta, jaoteltuna ennakkoon arvioituna heuristiikoille haastavimman tiedostovirusten käyttämän tartuntamenetelmän mukaan. Heuristiikoista ensimmäinen on tarkoitettu löytämään virukset, jotka eivät käytä löytämistä vaikeuttavia EPO-menetelmiä (Entry Point Obscuring). Toisen heuristiikan

tarkoituksena on selvittää voidaanko staattisella heuristiikalla luotettavasti löytää EPO-virukset, joita on yleensä vaikea heuristisilla tai muillakaan yleisillä menetelmillä löytää uhraamatta yhden tiedoston läpikäymiseen liikaa aikaa.

Tiedostovirusten tarkastelu aloitetaan luvussa 2 käymällä läpi yleisimpiä tapoja, joilla tiedostovirukset liittävät itsensä ohjelmatiedostoihin. Sen jälkeen luvussa 2 tarkastellaan edistyneempiä tartuntamenetelmiä, joiden tarkoituksena on viruksen havaitsemisen ja analysoinnin vaikeuttaminen. Läpi käydään sisältönsä piilottamaan pyrkivät pakkaavat, itsekoodaavat sekä poly- ja metamorfiset virukset. Viimeisenä käsittelyyn otetaan sijaintinsa piilottamaan pyrkivät EPO-virukset. Staattiset rakenteelliset heuristiikat perustuvat Win32-alustan käyttöjärjestelmien ohjelmatiedostojen PE-tiedostomuodon (Portable Executable) rakenteen tarkasteluun. Tiedostomuodon ymmärtäminen on välttämätöntä heuristiikkoja käsiteltäessä, joten PE-tiedostomuoto esitellään tutkielmassa virusten kannalta olennaisilta osiltaan tarkasti luvussa 3, joka käsittelee Win32-käyttöympäristöä myös pelkkää tiedostomuotoa laajemmin. Luvussa 4 siirrytään virusten käyttäytymisestä tarkastelemaan niiden havaitsemista. Tarkasteltavat menetelmät rajataan ohjelmatiedostoihin tarttuvien tiedostovirusten havaitsemisessa käytettäviin. Tutkielman ulkopuolelle jätetään virukset suorituksen aikana käytöksen perusteella tunnistavat ja haitallisen toiminnan estävät järjestelmät (behaviour blocker) sekä tiedostojen muuttamisen huomaavaan pyrkivät ohjelmat (integrity scanner). Käsiteltäviä menetelmiä ovat perinteiset merkkijonohaut, haastavampien virusten löytämiseen tarkoitetut algoritmiset menetelmät sekä X-RAY-menetelmät, koodin emulointi ja heuristiset ratkaisut. Luvussa 5 esitetään edellisissä luvuissa käsiteltyjen asioiden pohjalta heuristiikan pohjana olevat yksittäiset testit, sekä niistä kootut sääntöjoukot kahdentyyppisten Win32-tiedostovirusten löytämiseen (ohjelmatiedostoihin tarttuvien tavallisten virusten, sekä ohjelmatiedostoihin tarttuvien EPO-virusten).

2 TIEDOSTOVIRUKSET

2.1 Johdanto

Tiedostoviruksen täytyy suoritetuksi tullakseen liittää itsensä ohjelmatiedoston toimivaksi osaksi. Luvussa 2.2 käsitelläänkin ensin yleisimpiä virustyyppejä, jaoteltuina niiden tiedostoon liittymiseen käyttämän periaatteen mukaan. On syytä pitää mielessä, että tiedostoja käsitellään tässä luvussa loogisella tasolla – levyllä tiedostot eivät välttämättä ole yhtenäisenä kokonaisuutena, vaan saattavat sijaita levyllä useampana erillisenä osana. Virustyyppien esittelyn jälkeen käydään luvussa 2.3 läpi edistyneempiä tartuntamenetelmiä, joiden tarkoituksena on virusten löytämisen ja analysoinnin vaikeuttaminen. Edistyneet tartuntamenetelmät eivät ole luvun 2.2 virustyyppejä vastaavia, vaan ne ovat ikään kuin virustyyppien alityyppejä. Esimerkiksi luvun 2.2.2 loisvirukset voivat käyttää mitä tahansa edistynyttä tartuntamenetelmää.

2.2 Tiedostovirustyypit

2.2.1 Päällekirjoittavat virukset

Virukselle helpoin tapa tarttua tiedostoihin on yksinkertaisesti korvata tartunnan kohteena oleva ohjelmatiedosto itsensä kopiolla. Hieman parempi vaihtoehto on päällekirjoittaa jokin alkuperäisen tiedoston osa viruksella jolloin tiedoston koko säilyy muuttumattomana, minkä seurauksena käyttäjä ei havaitse virustartuntaa aivan yhtä helposti. Tyypillisesti virus päällekirjoittaa ohjelmatiedoston alkuosan taatakseen suoritetuksi tulemisensa. Jotkut harvat virukset päällekirjoittavat satunnaisen osan tiedostosta. Virusskannerit etsivät viruksia yleensä tietyistä viruksille tyypillisistä kohdista (alusta tai lopusta), joten satunnaisessa kohdassa sijainti saattaa auttaa viruksen piiloutumisessa. Satunnaisen kohdan päällekirjoittavan viruksen suoritus ei kuitenkaan ole taattua, koska isäntäohjelman suoritus ei välttämättä koskaan etene viruksen sijaintikohtaan saakka. Päällekirjoittavien virusten suurin ongelma on niiden havaitsemisen helppous – niiden tartuttamat isäntäohjelmat ovat tartunnan jälkeen vain harvoin toimintakykyisiä, koska osa alkuperäisestä ohjelmasta joutuu viruksen ylikirjoittamaksi.

Vaikkeivät yksinkertaiset ohjelmatiedostoja päällekirjoittavat virukset ole helpon havaittavuutensa vuoksi erityisemmin onnistuneet leviämään, ovat eräät viime vuosien laajalle levinneet sähköpostimadot käyttäneet päällekirjoitusta. VBS/LoveLetter@mm korvasi ohjelmatiedostojen sijaan datatiedostoja, kuten kuva-, teksti- ja äänitiedostoja kopiolla itsestään. Koska kaikkien päällekirjoitettujen tiedostojen alkuperäinen data on pysyvästi menetetty, ovat päällekirjoittavat virukset huomattavan tuhoisia.

Jotkin virukset käyttävät päällekirjoitusta hieman poikkeavalla tavalla korvatessaan käyttöjärjestelmätiedostoja muutetuilla versioilla. Esimerkiksi W95/MTX@m korvaa Wsock32.dll-tiedoston omalla versiollaan, jonka avulla se lähettää käyttäjän lähettämistä sähköposteista myös kopiot, joiden liitetiedostoiksi virus lisää itsensä.

2.2.2 Loisvirukset

Loisvirukset tarttuvat ohjelmiin lisäämällä itsensä niiden osaksi, eivätkä päällekirjoittamalla ohjelmatiedoston osia. Näin ne ainakin yrittävät säilyttää isäntäohjelman toiminnan ulkoisesti normaalina pysyäkseen paremmin havaitsemattomina. Käytännössä ohjelmointivirheet aiheuttavat melko usein outoa käytöstä ainakin, jos virus erehtyy tarttumaan tiedostoon, jonka rakenne poikkeaa tavallisesta.

Yksinkertaisin tapa tarttua tiedostoon on lisätä virus ohjelmatiedoston loppuun (appending virus). Esimerkiksi vanhoja MS-DOSin COM-ohjelmatiedostoja tartutettiin yleisesti lisäämällä virus isäntäohjelman päätteeksi. Lisäämällä ohjelman alkuun hyppykäsky, joka siirsi suorituksen viruksen alkuun, saatiin virus suoritettua ennen isäntäohjelmaa. Hyppykäskyn korvaamat tavut kopioitiin viruksen loppuun, josta ne viruksen suorituksen päätyttyä kopioitiin keskusmuistissa alkuperäiselle paikalleen ohjelman alkuun. Levyllä tiedoston alussa säilytettiin virukselle kontrollin siirtävä hyppykäsky. Kun isäntäohjelma oli keskusmuistissa palautettu alkuperäiseksi, siirsi virus hyppykäskyllä suorituksen sen alkuun. Modernien tiedostomuotojen rakenne on huomattavasti vanhaa COM-muotoa monimutkaisempi, mutta samaa periaatetta voidaan käyttää myös niiden tartuttamiseen.

Virus voi lisätä itsensä myös ohjelmatiedoston alkuun (prepending virus), lisäämiseen on kaksi yleisesti käytettyä menetelmää. Virus voi kopioida ensin isäntäohjelman keskusmuistiin itsensä jatkeeksi ja kirjoittaa sen jälkeen saadun yhdistelmän levylle alkuperäisen ohjelman tilalle. Toinen vaihtoehto on siirtää viruksen kokoinen osa alkuperäistä ohjelmaa tiedoston loppuun, ja kopioida virus tiedoston alkuun syntyneeseen tyhjään tilaan. Jotkut harvat virukset tekevät korvatusta osasta oman tiedostonsa, jonka ne pyrkivät piilottamaan käyttäjältä. Isäntäohjelman suorittaminen ei kummassakaan tapauksessa ole yhtä yksinkertaista kuin viruksen tiedoston loppuun lisäävien virusten tapauksessa, koska isäntäohjelman sisältämät absoluuttiset muistiosoitteet osoittavat ohjelman siirron jälkeen vääriin osoitteisiin. Useimmiten virukset hoitavat isäntäohjelman suorituksen luomalla ja suorittamalla tilapäisen ohjelmatiedoston, jonka sisältönä on alkuperäinen isäntäohjelma. Tilapäistiedosto poistetaan, kun isäntäohjelman suoritus on päättynyt.

2.2.3 Onkalovirukset

Onkalovirukset ovat kehittyneempi versio päällekirjoittavista viruksista. Onkalovirukset eivät päällekirjoita mitä tahansa kohtaa tartutettavasta ohjelmasta, vaan etsivät ohjelmatiedostoista käyttämättömiä kohtia, jotka koostuvat usein nollista tai välilyönneistä. Kääntäjän ja linkkerin jäljiltä saattaa ohjelmissa olla myös muunlaista täytettä. Koska onkalovirukset päällekirjoittavat vain isäntäohjelman toimintaan vaikuttamattomia kohtia, säilyy alkuperäisen ohjelman toiminnallisuus ennallaan. Onkalovirukset ovat usein hitaita leviämään, koska vain ohjelmatiedostot, jotka sisältävät tarpeeksi päällekirjoitettavaksi soveliasta dataa voidaan tartuttaa.

Uudet monimutkaisemmat tiedostomuodot mahdollistavat uudentyyppiset osioidut onkalovirukset (fractionated cavity virus). Esimerkiksi Windows-ohjelmien tiedostot koostuvat useammasta osiosta, joiden väliin jää käyttämätöntä tilaa. Osioidut onkalovirukset päällekirjoittavat tarvitsemansa määrän osioiden välisiä täytejonoja tartuttamansa tiedoston sisällä. Ensimmäinen osista (ns. head-osa) sisältää muiden osien osoitteet tiedoston sisällä ja kopioi viruksen suorituksen alkaessa kaikki osat toimivaksi kokonaisuudeksi keskusmuistiin. Virus muuttaa otsaketiedoissa olevan ohjelman ensimmäiseen komentoon osoittavan tulokohdan (entry point) osoittamaan head-osansa alkuun, ja saa niin ajettua itsensä ennen isäntäohjelmaa. Viruksen lopussa on

tallennettuna alkuperäinen tulokohta, jolla virus siirtää toimintansa lopuksi kontrollin isäntäohjelmalle. Isäntäohjelman suorittaminen ei vaadi virukselta muita toimenpiteitä, koska sitä ei muuteta tartunnan yhteydessä. Rajoituksena osioitujen onkalovirusten leviämiselle on isäntäohjelmalta vaadittavien onkaloiden koko. Kaikkien tarjolla olevien onkaloiden yhteiskoon täytyy olla virukselle riittävä. Lisäksi head-osan täytyy viruksen toimimiseksi olla yhdessä osassa, joten virus voi tarttua vain tiedostoihin, joissa on head-osalle tarpeeksi suuri yksittäinen onkalo.

2.3 Edistyneet tartuntamenetelmät

2.3.1 Pakkaus

Pakkaavat virukset pakkaavat tartuttamansa ohjelmatiedostot ja liittävät niihin (mahdollisesti pakatun) kopion itsestään. Viruksessa on aina pakkaamattomana muiden osien purkuun tarvittava koodi, joka täytyy ajaa ennen pakatun koodin suorittamista, koska pakatussa muodossa olevia ohjelmia ei voida sellaisenaan suorittaa. Pakatut ohjelmat puretaan usein erilliseen tilapäistiedostoon suorittamista varten. Ohjelmatiedostojen pakkaamiseen on olemassa runsaasti ohjelmia, joista virusten tekijät voivat valita haluamansa. Virusten usein käyttämiä pakkausalgoritmeja ovat esimerkiksi PKLITE ja LZEXE. Jotkin ajonaikaiset pakkaajat kuten UPX ja ASPack voivat purkaa ja suorittaa ohjelmia ilman erillistä väliaikaistiedostoa.

Motivaationa pakkauksen käyttöön viruksissa on niiden havaitsemisen ja analysoinnin vaikeuttaminen. Pakkaamalla isäntäohjelma pienempään kokoon voidaan virus tartuttaa siihen niin, että tartunnan saaneen ohjelman koko pysyy samana kuin ennen tartuntaa, eikä herätä käyttäjän huomiota. Eräät madot hyödyntävät pakkausta vähentääkseen aiheuttamansa verkkoliikenteen määrää. Viruksen analysointia varten täytyy virus ensin purkaa, ja myöskään perinteistä virustorjuntaohjelmien virusten löytämiseen käyttämää merkkijonohakua ei voida käyttää ennen kuin virus on purettu. Nykyiset virustentorjuntaohjelmat tukevat kaikkia yleisimpiä pakkausmenetelmiä, joten pakkaavat virukset eivät ole erityisen suuri ongelma.

Pakatut virukset ovat kuitenkin saaneet aikaan sen, että virustentorjuntaohjelmat saattavat antaa vääriä hälytyksiä joistakin laillisista ohjelmista, jotka käyttävät tiettyjä

viruksille ominaisia pakkausmenetelmiä. Useat virustentorjuntaohjelmat tulkitsivat esimerkiksi WinRar-ohjelman virheellisesti virukseksi sen käyttämän UPX-pakkauksen seurauksena (Marx 2005).

2.3.2 Itsekoodaus

Yksi edellisessä luvussa esiteltyjen pakkaavien virusten tarkoituksista on viruksen koodin sekoittaminen sen analysoinnin ja havaitsemisen vaikeuttamiseksi. Askeleen pidemmälle saman idean vievät itsensä salakirjoittavat virukset. Tässä vaiheessa on ehkä syytä huomauttaa, että tässä luvussa käsitellään nimenomaan itsensä salakirjoittavia viruksia eikä viruksia, jotka salakirjoittavat käyttäjän dataa vahingoittamistarkoituksessa. Itsekoodaava virus vaihtaa käyttämäänsä salausavainta tarttuessaan uuteen tiedostoon, ja tekee näin jokaisesta kopiostaan uniikin. Uniikkien tartuntojen tarkoituksena on estää virustentorjuntaohjelmia etsimästä virusta sille ominaisen merkkijonon perusteella.

Koska salattua koodia ei voi sellaisenaan suorittaa täytyy se ensin purkaa ja purkamisen suorittavan viruksen osan täytyy aina olla salaamattomassa muodossa. Salauksen avaava koodi on verrattain lyhyt ohjelmanpätkä. Se koostuu yleensä kuitenkin vähintään 10–15:sta tavusta, mikä on tarpeeksi paljon virustentorjuntaohjelmille viruksen tunnistamiseksi (Nachenberg 1997). Purkuohjelmista muodostettavista viruksen tunnistamiseen käytettävistä allekirjoituksista (signature) tulee kuitenkin lyhyitä. Käytettäessä lyhyitä allekirjoituksia kasvaa todennäköisyys sille, että jokin toinen, etsittävään virukseen millään lailla liittymätön ohjelma sisältää saman allekirjoituksen. Lisäksi usean eri viruksen erottelu toisistaan saattaa niiden purkuohjelmien samanlaisuuden vuoksi olla vaikeaa. Salakirjoitettujen virusten varmempaan löytämiseen ja tunnistamiseen voidaan käyttää X-RAY-menetelmää (luku 4.4) tai koodin emulointia (luku 4.5).

Virus tarvitsee aina itseään kopioidessaan uuden salausavaimen tehdäkseen kopiosta erilaisen kuin edellisestä. Avaimen luontiin tarvittavana satunnaisuuden lähteenä virukset usein käyttävät tietokoneen kelloa. Purun vaikeuttamiseksi salauksessa voidaan käyttää useampaa kuin yhtä avainta, ja avaimien arvoja voidaan muuttaa salauksen edetessä. Salaamiseen virukset käyttävät sen yksinkertaisuuden vuoksi usein XOR-

operaatiota salattavan osan ja avaimen välillä avaimen mittaisissa osissa. XOR on symmetrinen operaatio (teksti XOR avain = salaus, salaus XOR avain = teksti), joten sekä salaus että salauksen purku voidaan toteuttaa samalla algoritmilla, mikä sopii pienuuteen pyrkiville viruksille hyvin. Virustentorjuntaohjelmien tehtävän hankaloittamiseksi virus voi salata itsensä useampaan kertaan ja käyttää jokaisen salauksen purkuun omaa purkuohjelmaa, myös salaussilmukan suuntaa voidaan vaihdella (ts. virus voidaan salata lopusta alkuun, tai alusta loppuun päin). Erikoinen tapaus ovat RDA-virukset (Random Decryption Algorithm). Ne eivät tallenna salauksessa käyttämäänsä avainta minnekään, vaan purkavat salauksensa väsytyshyökkäyksellä (brute force attack). Oikean avaimen virukset tunnistavat esimerkiksi laskemalla purun tuloksena saadulle viruksen rungolle tarkistussumman, ja vertaamalla sitä purkuohjelmaan tallennettuun oikeaan tarkistussummaan. Tällainen salauksen purku vie huomattavasti avaimella suoritettua purkua enemmän aikaa ja aiheuttaa nopeaan virusten etsimiseen pyrkiville virustorjuntaohjelmille ylimääräisiä ongelmia.

Salauksen purku tapahtuu useimmiten suoraan muistissa, päällekirjoittamalla salattua ohjelmakoodia puretulla sitä mukaan kuin purku etenee. Mikäli käyttöjärjestelmä ei mahdollista suoraan muistissa päällekirjoittamista, voidaan purku suorittaa lisäämällä purettu koodi pinomuistiin. Huonoin vaihtoehto on erillisen muistialueen varaaminen puretulle koodille. Muistin varaamisen suorittavan ohjelman täytyy olla salaamatonta ja sijaita ennen salauksen purun suorittavaa koodia, mikä kasvattaa virustentorjuntaohjelmille viruksen löytämiseen ja tunnistamiseen tarjolla olevaa salaamatonta materiaalia. (Szor 2005 s. 258)

2.3.3 Polymorfismi

Itsensä salakirjoittavien virusten purkuohjelma riittää useimmissa tapauksissa virusten löytämiseen, joten viruksille ja virustentorjuntaohjelmille tyypillinen kilpajuoksu jatkui virusten tekijöiden seuraavalla askeleella: purkuohjelman muuttamisella jokaisessa tartunnassa. Polymorfiset virukset salaavat ohjelmakoodinsa kuten tavalliset itsekoodavat virukset, mutta lisäksi niissä on erillinen mutaatiokone (mutation engine), joka luo jokaiselle virustartunnalle ulkoisesti erilaisen, mutta toiminnallisesti samanlaisen purkuohjelman. Kun polymorfisen viruksen tartuttama ohjelma ajetaan,

hoitaa purkuohjelma ensin viruksen rungon ja sen mukana mutaatiokoneen salauksen purun. Tämän jälkeen viruksen runko saa kontrollin ja etsii seuraavan tartutettavan ohjelman. Virus kopioi itsensä purkuohjelmaa lukuunottamatta keskusmuistiin ja salaa itsensä uudella avaimella luodakseen uniikin kopion. Mutaatiokone luo uuden purkuohjelman, jonka virus liittää keskusmuistiin luotuun viruksen salakirjoitettuun runkoon. Lopuksi purkuohjelma–runko-yhdistelmä liitetään tartutettavaan tiedostoon.

Mutaatiokoneen luoma mutaatio ei ole samanlaista kuin biologinen, satunnainen mutaatio. Mikäli purkukoodi mutatoituisi täysin satunnaisesti, tulisi lähes kaikista purkuohjelmista toimimattomia. Mutaatiokoneen aikaansaama mutaatio on hallittua, ja tuloksena on aina (olettaen että mutaatiokoneessa ei ole ohjelmointi- tai suunnitteluvirheitä) toimiva, ja toiminnallisesti edellisiä purkuohjelmia vastaava ohjelma. Mutaatiokoneet luovat erilaisia purkuohjelmia yksinkertaisimmillaan lisäämällä varsinaisen purun hoitavien komentojen sekaan toimintaan vaikuttamattomia komentoja. Kuvassa 1 on viisi kuvitteellisella symbolisella konekielellä kirjoitettua lyhyttä ohjelmaa, jotka huomattavasta erilaisuudestaan huolimatta suorittavat saman laskutoimituksen.

MOV A,R1	MOV A,R1	MOV A,R1	MOV A,R1	MOV A,R1
ADD B,R1	NOP	ADD #0,R1	OR R1,R1	TST R1
ADD C,R1	ADD B,R1	ADD B,R1	ADD B,R1	ADD C,R1
SUB #4,R1	NOP	OR R1,R1	MOV R1,R5	MOV R1,R5
MOV R1,X	ADD C,R1	ADD C,R1	ADD C,R1	ADD B,R1
	NOP	SHL #0,R1	SHL R1,0	CMP R2,R5
	SUB #4,R1	SUB #4,R1	SUB #4,R1	SUB #4,R1
	NOP	JMP .+1	ADD R5,R5	JMP .+1
	MOV R1,X	MOV R1,X	MOV R1,X	MOV R1,X
			MOV R5,Y	MOV R5,Y
(a)	(b)	(c)	(d)	(e)

Kuva 1. Ohjelmia, jotka eroistaan huolimatta suorittavat saman tehtävän (Tanenbaum 2001 s. 632)

Kuvan 1(a) ohjelmassa ei ole merkityksettömiä komentoja, vaan se suorittaa ilman ylimääräisiä mutkia laskutoimituksen X=(A+B+C-4), käyttäen rekisteriä R1 välitulosten tallentamiseen. Selvin vaihtoehto ylimääräiseksi käskyksi on NOP (No OPeration), jonka tavatessaan prosessori siirtyy suoraan seuraavan käskyn suorittamiseen (Kuva 1(b)). Muita käytännössä vaikutuksettomia käskyjä ovat nollan lisääminen rekisterin arvoon, muuttujan OR-operaatio itsensä kanssa, rekisterin siirtäminen vasemmalle nolla

bittiä tai seuraavaan käskyyn hyppy (Kuva 1(c)). Kehittyneemmät virustentorjuntaohjelmat osaavat kuitenkin polymorfisia viruksia jahdatessaan jättää turhat käskyt huomioimatta, joten hyvän mutaatiokoneen tulisi luoda käskyjä, jotka ovat turhia, mutta eivät näytä siltä. Kuvassa 1(d) on käyttöön otettu hämäykseksi ylimääräinen rekisteri R5, jota käytetään näennäisesti hyödyksi. Joitain käskyjä voidaan korvata saman vaikutuksen omaavalla toisella käskyllä. Esimerkiksi XOR-operaatio rekisterin itsensä kanssa, ja rekisterin arvon vähentäminen omasta arvostaan nollaavat molemmat kyseisen rekisterin arvon. Ylimääräisten käskyjen käyttämisen lisäksi voi mutaatiokone myös vaihtaa välttämättömien käskyjen suoritusjärjestystä silloin, kun käskyjen keskinäisellä järjestyksellä ei ole merkitystä (Kuva 1(e)). Kuten kuvan 1 ohjelmien radikaaleista eroista selvästi huomaa, ei yksinkertaisella merkkijonohaulla ole mahdollista löytää polymorfista virusta sen purkuohjelman perusteella.

Ensimmäinen polymorfinen virus oli vuonna 1990 MS-DOSille tehty 1260. 1260 käyttää salaukseensa kahta eri avainta, joista toista se muuttaa salauksen edetessä. Purkurutiininsa muokkaukseen se käyttää edellä kuvattuja menetelmiä. Ohjelmalistauksessa 1 on 1260:n purkuohjelma perusmuodossaan ilman ylimääräisiä käskyjä.

Ohjelmalistaus 1. 1260:n purkurutiini perusmuodossaan (Skulason 1990)

```
          mov  ax, encryption_key_1
          mov  cx, encryption_key_2
          mov  di, 0147h
label:    xor  [di], cx
          xor  [di], ax
          inc  di
          inc  ax
          loop label
```

Tarttuessaan 1260 lisää purkuohjelman käskyjen sekaan satunnaisen määrän turhia käskyjä (NOP, CLC (Clear Carry Flag), XOR-operaatioita ja merkityksettömien rekisterien kasvattamista sekä aritmeettisia operaatioita). Lisäksi purkurutiini on jaettu kolmeen loogiseen osaan, joiden sisällä olevien käskyjen suoritusjärjestystä 1260 voi muuttaa toiminnallisuuteen vaikuttamatta. Kolme ensimmäistä komentoa hakevat rekistereihin molemmat käytetyt salausavaimet, sekä purettavan osan alkuosoitteen. Näiden kolmen komennon keskinäisellä suoritusjärjestyksellä ei ole merkitystä, ja uutta

purkurutiinia luodessaan 1260 laittaa ne satunnaiseen permutaatioon. XOR-vertailut suorittavat salauksen purun, ja niiden suoritusjärjestyksen 1260 valitsee satunnaisesti, samoin kuin avainta kasvattavan sekä purettavan sanan osoitetta muuttavien käskyjen keskinäisen järjestyksen.

Mutaatiokoneet ovat viruksiin verrattuna monimutkaisia tehdä, hienostuneet mutaatiokoneet erittäinkin monimutkaisia. Niiden kehittämisen vaikeus rajoitti suuresti vaarallisten polymorfisten virusten määrää. Asiaan tuli kuitenkin muutos vuonna 1991, kun Dark Avenger -aliaksella tunnettu virusten kirjoittaja julkaisi oman MtE:ksi nimeämänsä mutaatiokoneen. MtE:n taustalla oli ajatus antaa kenelle tahansa mahdollisuus luoda tehokas polymorfinen virus liittämällä MtE viruksensa osaksi. MtE ei käytä yhtään hyödytöntä käskyä purkuohjelman satunnaistamiseen. Sen käyttämät käskyt ovat kaikki välttämättömiä, mutta sellaisia, että lopullinen tulos saadaan ikään kuin kiertoteitse usean turhan mutkan kautta. Suoranaisia turhia käskyjä välttämällä MtE estää virustentorjuntaohjelmia löytämästä purkuohjelmia optimoimalla ne perusmuotoonsa turhat käskyt poistamalla. Intel 80x86 arkkitehtuuria käyttävien tietokoneiden ohjelmissa rekistereitä voidaan ohjelmoijan niin halutessa hyödyntää hyvin vaihtelevasti. Esimerkiksi operaatioita suoritettaessa dataa voidaan hakea keskusmuistista mihin tahansa neljästä ohjelmoijan haluamasta rekisteristä. Eri rekistereitä käsittelevät toiminnot suoritetaan konekielen eri komennoilla, joten MtE voi näin muuttaa purkuohjelmaa toiminnallisuuteen vaikuttamatta. Lisäksi MtE voi hyödyntää rekisterien kasvattamisessa eri komentoja. Arvoja voidaan kasvattaa joko käskyllä INC, joka lisää rekisterin arvoa yhdellä tai ADD, joka lisää rekisteriin halutun arvon. Sama lopputulos saadaan aikaan myös kasvattamalla rekisterien arvoa ensin tarvittua enemmän, ja sen jälkeen vähentämällä arvo halutuksi. (Yetiser 1992)

Myöhemmin ilmestyi muitakin valmiita vaihtelevan tasoisia mutaatiokoneita virusten tekijöiden hyödynnettäviksi. Vaikka valmiiden mutaatiokoneiden käyttö antoi helpon tavan luoda polymorfinen virus, saattoi se olla myös viruksen tekijälle haitaksi. Mikäli mutaatiokoneen luomissa purkuohjelmissa oli tarpeeksi yhtäläisyyksiä, pystyivät virustentorjuntaohjelmat havaitsemaan kaikki tiettyä mutaatiokonetta hyödyntäneet uudet virukset purkuohjelman perusteella ilman ohjelmiston päivittämistä.

Kuten aiemmin todettiin polymorfiset virukset vaihtavat purkuohjelmansa käskyjen suoritusjärjestystä, kun järjestyksellä ei ole merkitystä lopputulokseen. Vuonna 1996 tehty Ply vei idean askeleen pidemmälle ja vaihteli vapaasti käskyjen järjestystä purkuohjelman koodissa. Toiminnallisuuden Ply säilytti lisäämällä purkuohjelmaan hyppykäskyjä, jotka pitivät purkuohjelman käskyjen suoritusjärjestyksen tarvittavana huolimatta käskyjen järjestyksestä itse ohjelmakoodissa.

Uusiin käyttöjärjestelmiin siirryttäessä virusten tekijöillä menee oma aikansa uuden alustan vaatiman tietouden hankkimisessa. Ensimmäinen Windows 95 -virus, joka muutti purkuohjelmaansa, ilmestyikin vasta vuonna 1997. W95/Memorial ei ollut kuitenkaan varsinainen polymorfinen virus, koska sen luomien erilaisten purkuohjelmien määrä oli rajallinen. W95/Memorialin purkuohjelma koostuu 11 osasta, jotka virus voi järjestellä 96:lla eri tavalla (Szor 1997). Rajallisen määrän eri purkuohjelmia luovia viruksia kutsutaan oligomorfisiksi. Koska purkuohjelmia on rajallinen määrä, voidaan niistä jokaisesta etsiä virustutkaa varten allekirjoitus. Oligomorfisia viruksia voidaan siis virustorjuntaohjelmien näkökulmasta käsitellä samoin kuin tavallisia viruksia.

Ensimmäinen todellinen 32-bittinen polymorfinen Windows-virus tehtiin vasta vuonna 1998. W95/Marburg on hidas polymorfinen virus. Hitaalla polymorfisella viruksella tarkoitetaan virusta, joka muuttaa purkuohjelmaansa vain vähän tartunnasta toiseen, tai käyttää samaa purkuohjelmaa useammassa peräkkäisessä tartunnassa. Hitaan polymorfisen viruksen etuna on sen analysoinnin vaikeus. Tavallisesta polymorfisesta viruksesta saadaan nopeasti luotua tuhansia erilaisia tartuntoja, joita analysoimalla saadaan yleiskuva viruksen luomista purkuohjelmista ja niiden mahdollisista yhtäläisyyksistä. Hitaasta polymorfisesta viruksesta saattaa joitain harvinaisempia erikoistapauksia jäädä havaitsematta, mikäli analysointiin ei uhrata huomattavasti aikaa. W95/Marburg jakoi purkuohjelmansa lyhyisiin aliohjelmiin, joiden järjestystä ohjelmakoodissa virus vaihtelee. Aliohjelmien järjestyksellä koodissa ei ole merkitystä viruksen toiminnan kannalta, koska purkusilmukka kutsuu niitä joka tapauksessa aina samassa järjestyksessä. Aliohjelmien järjestyksen vaihtelemisen lisäksi virus lisää niiden väliin turhia käskyjä. Purkuohjelman ulkonäön vaihtamisen lisäksi myös salaus-

ja purkuohjelman toiminnassa on suurta vaihtelua - W95/Marburg vaihtelee salauksessa käytettävää operaatiota ja avaimen pituutta. Ohjelmalistauksessa 2 on esimerkki W95/Marburgin aliohjelmilla suorittamasta purkuohjelman sekoittamisesta. Listauksesta on poistettu viruksen purkuohjelman sekaan normaalisti lisäämät ylimääräiset käskyt, ensimmäiseksi suoritettava aliohjelma on Decryptor_Start.

Ohjelmalistaus 2. Yksinkertaistettu esimerkki W95/Marburgin purkuohjelmasta (Szor 2005 s. 265).

```
Start:
     ; Encrypted/Decrypted Virus body is placed here

Routine-6:
dec     esi                 ; decrement loop counter
ret

Routine-3:
mov     esi,439FE661h       ; set loop counter in ESI
ret

Routine-4:
xor     byte ptr [edi],6F   ; decrypt with a constant byte
ret

Routine-5:
add     edi,0001h           ; point to next byte to decrypt
ret

Decryptor_Start:
call    Routine-1           ; set EDI to "Start"
call    Routine-3           ; set loop counter

Decrypt:
call    Routine-4           ; decrypt
call    Routine-5           ; get next
call    Routine-6           ; decrement loop register
cmp     esi,439FD271h       ; is everything decrypted?
jnz     Decrypt             ; not yet, continue to decrypt
jmp     Start               ; jump to decrypted start

Routine-1:
call    Routine-2           ; Call to POP trick!

Routine-2:
pop     edi
sub     edi,143Ah           ; EDI points to "Start"
ret
```

W95/Marburg oli niin valitettavan onnistunut virus, että virustentorjuntaohjelmat vaativat muutoksia hakutekniikkaansa viruksen löytääkseen. Moiset muutokset vievät enemmän aikaa kuin pelkän tunnistukseen vaadittavan merkkijonon lisääminen tietokantaan. Tämän vuoksi W95/Marburg levisi laajalle muiden muassa useampien tietokonelehtien kylkiäisenä levitettävien cd-rom-levyjen mukana. (Szor 1998c)

2.3.4 Metamorfismi

Polymorfiset virukset aiheuttavat vaikeuksia vain, jos niiden salaus kestää. Nykyiset virustentorjuntaohjelmat pystyvät purkamaan viruksen salauksen, minkä jälkeen on sisältä löytyvä virus helppo tunnistaa yksinkertaisella merkkijonohaulla. Metamorfisten virusten ideana on koodin muuntelu kuten polymorfisissa viruksissa, mutta laajemmin käytettynä. Metamorfiset virukset eivät rajaa muutoksia purkuohjelmaan, vaan muuttavat satunnaisesti koko virusta, säilyttäen kuitenkin viruksen toiminnan ennallaan. Koko viruksen muuttuessa tartunnasta toiseen ei virustorjuntaohjelmille ole tarjolla merkkijonoa, josta tunnistaa virus. Siksi metamorfisten virusten ei välttämättä tarvitse salakirjoittaa itseään lainkaan.

Metamorfisilla viruksilla on mutaatiokone, jonka tehtävänä on ohjelmien tartuttamista varten luoda viruksesta kopio, joka on edeltäjästään ulkoisesti erilainen, mutta toiminnallisesti samanlainen. Metamorfisten virusten mutaatiokoneissa on kolme osaa: takaisinkääntäjä (disassembler), toiminnallisuuden analysoija (reverse engineer) ja muunnin (transformer). Osat suorittavat tehtävänsä peräkkäin, edellisen osan tuloksen toimiessa seuraavan osan syötteenä. Takaisinkääntäjä muokkaa virusta ensimmäisenä, se muuttaa konekielisen ohjelman takaisin symboliseksi konekieleksi. Takaisinkääntäjän tärkein tehtävä on ohjelmakoodin ja datan erottaminen toisistaan, koska dataa ei viruksen toiminnan säilyttämiseksi saa muuttaa. Toiminnallisuuden analysoijan tehtävä määräytyy pitkälti muuntimen toimintaperiaatteen mukaan. Mikäli muunnin on hyvin yksinkertainen ja muokkaa vain yksittäisiä käskyjä, ei toiminnallisuuden analysoijaa tarvita lainkaan. Koodin muokkauksen tapahtuessa lohkoissa, toiminnallisuuden analysoijan tehtävänä on viruksen jako niihin. Jokaisessa lohkossa täytyy olla tasan yksi tulo- ja lähtömahdollisuus, ts. lohkon keskellä ei saa olla hyppykäskyjä lohkon ulkopuolelle, eikä kontrolli saa siirtyä lohkon ulkopuolelta muuhun kuin lohkon ensimmäiseen käskyyn. Toiminnallisuuden analysoija voi jaotella

koodia myös jollain muulla muuntimen kaipaamalla tavalla, mutta sen tarkoitus on aina jakaa virus sellaisiksi paloiksi, joiden sisäistä toteutusta muunnin voi vapaasti muokata vaarantamatta viruksen toimintaa. (Lakhotia ym. 2004)

Ensimmäinen yritys tehdä 32-bittinen metamorfinen virus oli vuonna 2000 tehty W32/Apparition. Poikkeuksellista tässä viruksessa oli sen tapa muuntua omaa lähdekoodiaan muuttamalla. Lähdekoodiaan virus muutti lisäämällä ja poistamalla satunnaisesti ylimääräisiä käskyjä. Tarttuessaan viruksen piti kääntää oma muutettu lähdekoodinsa, mikä rajoitti sen leviämistä suuresti, koska suurimmassa osassa tietokoneista ei ollut sen vaatimaa kääntäjää asennettuna. Ohjelmien muuttaminen on lähdekoodimuodossa helpompaa kuin jo konekielelle käännettynä, joten mikäli tulevaisuudessa käyttöjärjestelmien mukana tulee automaattisesti kääntäjä, tullaan lähdekoodiaan mukanaan kuljettavia viruksia varmasti näkemään enemmänkin.

Yksinkertaiset metamorfiset virukset hyödyntävät samoja tekniikoita kuin polymorfiset virukset. Vuonna 1998 tehty W95/RegSwap oli ensimmäinen Windowsille tehty yksinkertainen metamorfinen virus. Se teki tartunnoistaan uniikkeja vaihtamalla operaatioissa käyttämiään rekistereitä. Yksinkertaisten muutosten vuoksi virusten eri tartunnoissa on runsaasti yhtäläisyyksiä, joita voidaan etsiä merkkijonohauilla. Eri tartunnoille yhteisten tavujen etsintää voidaan täydentää myös osittaisia tavuja hakemalla. Eri rekisterejä käyttävien käskyjen operaatiokoodit (käskyjen heksamuotoinen esitys) ovat usein lähellä toisiaan, joten etsintää voidaan täydentää myös osittaisia tavuja hakemalla. Esimerkiksi kahdesta eri W95/RegSwapin tartunnasta löytyvät jonot 5ABFh ja 58BBh löydetään haulla 5?B?h. Toinen polymorfisista viruksista tuttu tekniikka on esimerkiksi Ply:n ja W95/Marburgin käyttämä ohjelmakoodin permutaatio. Permutaatioon turvauduttaessa ohjelmakoodin järjestystä tiedostossa muutetaan, mutta suoritusjärjestys säilytetään joko hyppykäskyillä tai aliohjelmien suoritusjärjestys säilyttämällä.

Esimerkki kehittyneemmästä metamorfisesta viruksesta on W32/Metaphor. W32/Metaphorin takaisinkääntäjä ja toiminnallisuuden analysoija muuttavat viruksen muuntimen ymmärtämään pseudokoodi-muotoon. Pseudokoodi kuvaa ainoastaan

viruksen toimintaa, eikä sitä miten se halutun toimenpiteen suorittaa. Kun muunnin saa mutaatiokoneen aikaisemmilta komponenteilta tiedoksi toimenpiteisiin jaetun ohjelman, se voi vapaasti valita satunnaisen, edellisestä virussukupolvesta riippumattoman implementaation jokaiselle erilliselle toimenpiteelle. Ohjelmalistauksessa 3 on eräs mahdollinen toteutus ohjelman osasta, jolla W32/Metaphor hakee kernel32.dll-moduulin osoitteen. (Jordan 2002)

Ohjelmalistaus 3. Osa W32/Metaphor viruksesta. Oikealla saman toiminnallisuuden omaava lyhyt ohjelma (Jordan 2002).

```
mov dword_1, 0h
mov edx, dword_1
mov dword_2, edx
mov ebp, dword_2
mov edi, 32336C65h
lea eax, [edi]
mov esi, 0A624548h
or esi, 4670214Bh
lea edi, [eax]
mov dword_4, edi
mov edx, ebp
mov dword_5, edx
mov dword_3, esi
mov edx, offset dword_3
push edx
mov dword_6, offset GetModuleHandleA
push dword_6
pop dword_7
mov edx, dword_7
call dword ptr ds:0[edx]
```

```
mov dword_3, 6E72654Bh
mov dword_4, 32336C65h
mov dword_5, 0h
push offset dword_3
call
ds:[GetModuleHandleA]
```

Vastaavasti pitkiä käskysarjoja toimintojen suorittamiseen generoimalla metamorfinen virus voi luoda itsestään loputtomasti toisistaan niin radikaalisti poikkeavia versioita, ettei niissä ole mitään yhteistä merkkijonohaun kohteeksi kelpaavaa koodia.

Poly- ja metamorfisia viruksia käsitelleissä luvuissa on esitelty useita niiden koodin muuttamisessa käyttämiä tekniikoita, joista seuraavaksi kertauksena tärkeimmät:

- turhien, toiminnan kannalta merkityksettömien käskyjen käyttö,
- operaatioissa käytettävien rekisterien vaihtelu,
- toisistaan riippumattomien käskyjen uudelleenjärjestely,
- minkä tahansa käskyjen uudelleenjärjestely ja suoritusjärjestyksen säilyttäminen hyppykäskyillä tai aliohjelmakutsuilla,
- käskyn tai käskyjoukkojen korvaaminen toisilla sama toiminnallisuus säilyttäen.

Vaikka metamorfiset virukset käyttävät periaatteessa samoja tekniikoita kuin polymorfiset, on metamorfisten virusten havaitseminen huomattavasti vaikeampaa. Polymorfiset virukset ovat haavoittuvaisia, koska niiden salauksen sisältä löytyvä ohjelma ei muutu tartunnasta toiseen. Kunhan salaus saadaan purettua, on polymorfinen virus helppo löytää sen allekirjoituksen perusteella. Metamorfiset virukset aiheuttavat huomattavasti suuremman haasteen, koska niillä ei ole pysyvää allekirjoitusta jota etsiä.

2.3.5 Tulokohdan kätkeminen (EPO)

Tulokohdaksi (entry point) kutsutaan ohjelmatiedoston ensimmäistä suoritettavaa käskyä. Vanhoissa yksinkertaisissa tiedostomuodoissa tulokohta sijaitsi aina samassa kohtaa ohjelmatiedostoa. Uudemmissa tiedostomuodoissa tulokohta määritellään erikseen ohjelmatiedoston otsakekentissä.

Tiedostovirukselle ei riitä pelkästään se, että virus kopioi itsensä ohjelmatiedostojen yhteyteen. Suoritetuksi tullakseen viruksen täytyy jotenkin siirtää kontrolli itselleen, kun sen isäntäohjelmaa suoritetaan. Virus saa itsensä suoritettua joko kopioimalla itsensä ohjelmatiedoston tulokohtaan tai muuttamalla tulokohdassa olevan käskyn hyppykäskyksi omaan aloituskohtaansa. Uudemmissa tiedostomuodoissa virus voi myös muuttaa otsaketiedoissa olevan tulokohdan osoittamaan itseensä. Virustutkat voivat nopeuttaa toimintaansa hyödyntämällä virusten tarvetta kopioida itsensä ohjelman tulokohtaan. Sen sijaan, että virustutka kävisi läpi koko tiedoston virusten varalta, voi se rajoittua etsimään virusten allekirjoituksia ohjelman tulokohtaa seuraavista muutamasta kilotavusta. Vastatoimena virus voi pitäytyä kopioimasta itseään suoraan tulokohtaan tai muuttamasta otsakkeessa olevaa tulokohdan osoitetta. Sen sijaan virus korvaa tulokohdassa olevan komennon virukseen johtavalla hyppykäskyllä (obfuscated tricky jump). Virustutkien on kuitenkin helppo tarkistaa ohjelman tulokohta hyppykäskyn varalta, ja sellaisen löytäessään tarkistaa hyppykäskyn päätepisteestä jälleen muutama kilotavu virusten allekirjoitusten varalta.

Tulokohtansa kätkevät virukset (Entry Point Obscuring, EPO) eivät vaihda isäntäohjelman tulokohtaa, eivätkä tulokohdassa sijaitsevaa koodia. Peittääkseen jalanjälkensä virustutkilta EPO-virukset siirtävät suorituksen omaan koodiinsa satunnaisessa kohdassa isäntäohjelmaa. Eräänlainen EPO-virus on luvussa 2.2.1

mainittu satunnaisen kohdan isäntäohjelmasta päällekirjoittava virus. Sen ongelmina ovat kuitenkin isäntäohjelman toimimattomuudesta seuraava viruksen helppo havaitseminen, sekä viruksen suorittamisen epävarmuus. Täysin satunnaiseen kohtaan virus kopioitaessa ei ole takeita, että isäntäohjelman suoritus etenee komentojen haarautumisen vuoksi aina virukseen saakka. Todellisen EPO-viruksen täytyy säilyttää isäntäohjelman toimivuus ja suorittaa tartunta niin, että viruksen suoritus on mahdollisimman varmaa jokaisella isäntäohjelman suorituskerralla.

EPO-virukset yleistyivät 90-luvun puolivälissä MS-DOSille. Yhdessä polymorfisuuden kanssa niiden tarkoituksena oli vaikeuttaa viruksen löytämistä ohjelmaa emuloimalla. Emuloinnin ongelmana on sen hitaus – normaaleja tulokohtia käyttäviä viruksia etsittäessä tiedetään mistä kohtaa isäntäohjelmaa emulointi kannattaa aloittaa, ja näin voidaan vähentää emuloinnin määrää. EPO-virukset pakottavat emuloimaan pitempiä aikoja, ja emulointi mahdollisesti lopetetaan ennen kuin se ehtii virukseen asti. MS-DOSille tehdyt EPO-virukset tyypillisesti korvasivat tiedoston satunnaisesta kohdasta käskyjä hyppykäskyllä viruksen alkuun. Itsekoodavat tai polymorfiset virukset saattoivat kopioida hieman suuremmankin osan keskelle tiedostoa valmistelemaan viruksen rungon suoritusta. Se, kuinka usein virus tulisi suoritetuksi, jäi lähinnä sattuman varaan.

Vuonna 1996 tehty Nexiv_Der oli omalaatuinen siksi, että se pyrki varmistamaan omaan koodiinsa suorituksen siirtävän käskyn hyvän sijoittamisen seuraamalla isäntäohjelman suoritusta. Nexiv_Der tarttui tiedostoista ainoastaan COM-tyyppisiin ja etsi isäntäohjelmasta jonkin seuraavista kolme tavua vaativista käskyistä:

- CALL xxxx
- JMP xxxx
- ADD <8-bittinen rekisteri>, xx
- OR <8-bittinen rekisteri>, xx

Käskyn löydettyään virus tallensi sen isäntäohjelman jatkeeksi lisättävän viruksen loppuun, ja korvasi käskyn alkuperäisessä sijainnissaan CALL-komennolla, joka siirsi suorituksen viruksen alkuun. Isäntäohjelman suoritusta seuraamalla virus paransi suoritetuksi tulemisensa todennäköisyyttä verrattuna satunnaisessa kohdassa

tulokohtaansa siirtymiseen, mutta varmaa viruksen suoritus tulevilla isäntäohjelman ajokerroilla ei ollut. Usein ohjelman suorituksen kulkuun vaikuttavat ohjelman käynnistysparametrit tai käyttäjän suorituksenaikaiset valinnat. Vaikka Nexiv_Der siis tarttuessaan otti huomioon isäntäohjelman suoritettavan osan, ei se taannut viruksen suoritusta jokaisella isäntäohjelman suorituskerralla. Erityisesti käyttöympäristön tai käyttäjän vaihto saattavat muuttaa isäntäohjelmasta suoritettavaksi tulevia osia, ja nämä vaihdot puolestaan ovat viruksen leviämisen vuoksi väistämättömiä. (Szor 1996)

Nexiv_Der-viruksen ideoimaa isäntäohjelman suorituksen seurantaa on mahdollista käyttää myös Windows-ympäristössä. W32/Perenast hyödyntää Windowsin ohjelmien testaukseen tarkoitettua debug-APIa tiedostoihin tarttuessaan. Virus luo debug-APIn avulla Windowsin prosessilistassa näkymättömän prosessin, ja etsii isäntäohjelmasta soveliaan 450 tavun mittaisen osuuden, jonka se korvaa purkuohjelmallaan (W32/Perenast on polymorfinen virus). Isäntäohjelman korvatun osuuden virus tallentaa oman koodinsa osaksi. (Marinescu 2003)

Vaativin EPO-tartuntatapa on koodin integrointi. Tätä tekniikkaa käyttävät virukset lisäävät itsensä isäntäohjelman joukkoon. Esimerkiksi W95/Zmist ei normaalin loisviruksen tapaan liitä itseään yhtenäisenä ohjelmana tartunnan yhteydessä, vaan sen takaisinkääntäjä lisää isäntäohjelmaan satunnaisiin paikkoihin tyhjää tilaa viruksen käskyille. Tekemiinsä tyhjiin tiloihin virus lisää oman ohjelmakoodinsa toisistaan erillään olevina käskysaarekkeina. Jokaisen käskysaarekkeen lopussa on hyppykäsky seuraavan saarekkeen alkuun, jotta virus tulee suoritetuksi kokonaisuudessaan. Tällainen raju koodin muuttaminen vaatii myös isäntäohjelman paikkausta (lähinnä muistiosoitteiden päivittämistä) sen pitämiseksi toimivana tartunnan jälkeen. W95/Zmistin kaltaisen viruksen tekeminen on erittäin vaativaa ja aikaavievää. Myös itse viruksen toiminta on takaisinkääntämisen vuoksi hidasta.

2.4 Yhteenveto tiedostoviruksista

Tiedostovirukset pyrkivät lisäämään itsensä ohjelmatiedostojen osaksi säilyttäen isäntäohjelman toiminnallisuuden samalla ennallaan. Tiedostovirukset voidaan luokitella sen mukaan, mihin kohtaan isäntäohjelmaa ne itsensä liittävät. Esitellyistä

virustyypeistä jatkossa keskitytään loisviruksiin, jotka muodostavat valtaosan tehdyistä Win32-viruksista.

Ajan mittaan virusten kirjoittajat ovat kehittäneet edistyneitä tartuntamenetelmiä vastustaakseen parantuneita virustentorjuntaohjelmia ja vaikeuttamaan virusten analysointia. Salauksen käyttö on pohjana itsekoodaville viruksille, salauksen ja tartunnasta toiseen muuttuvan koodin käyttö puolestaan polymorfisille viruksille. Metamorfiset virukset laajentavat muutettavan koodin aluetta koko viruksen kattavaksi ja poistavat näin mahdollisuuden etsiä viruksen salaamatonta muotoa. EPO-virukset pyrkivät piilottamaan itsensä virustentorjuntaohjelmilta salaamalla tulokohtansa. EPO-virukset, metamorfismi ja EPO-menetelmiä ja polymorfismia yhdistelevät virukset saattavat olla hyvinkin vaikeita tapauksia virustentorjuntaohjelmille löydettäviksi siedettävän lyhyessä ajassa.

3 WIN32-TIEDOSTOVIRUKSET

3.1 Johdanto

Tässä luvussa tarkastellaan lähemmin, miten edellisessä luvussa melko yleisellä tasolla esitetyt tartuntamenetelmät voidaan toteuttaa Win32-käyttöympäristössä. Win32-virusten toiminnan ymmärtämiseksi esitellään Win32-käyttöympäristö ja käydään läpi Portable Executable -tiedostomuoto, jonka ymmärtäminen on virusten toiminnan hahmottamiseksi välttämätöntä. Myös DLL-tiedostot (Dynamic Link Library) ja niiden hyödyntäminen ohjelmatiedostoista käsin käydään tarkasti läpi, koska ne tarjoavat mahdollisuuksia vaikeasti havaittavien EPO-menetelmien toteutukselle. Käyttöympäristön ja tiedostomuodon läpikäynnin jälkeen voidaan esittää kuinka edellisen luvun virustyypit voidaan toteuttaa Win32-tiedostoviruksissa.

3.2 Win32-käyttöympäristö

Win32-tiedostoviruksista puhuttaessa on ensin syytä määritellä mitä Win32:lla tarkoitetaan. Win32 on nimitys käyttöjärjestelmän sovelluksille tarjoamalle rajapinnalle (Application Program Interface, API). Rajapinta määrittelee rutiinit, jotka ovat sovelluksien hyödynnettävissä DLL-tiedostojen kautta. DLL-kirjastotiedostot ovat Windowsin tapa mahdollistaa samojen rutiinien käyttö useammasta ohjelmasta ilman, että jokaisen ohjelmista täytyy sisältää kyseiset rutiinit. Käyttöjärjestelmän kirjastotiedostot sisältävät yleensä satoja rutiineja. Win32-rajapinta määrittelee kaikki palvelut, jotka käyttöjärjestelmä tarjoaa sovellusohjelmille. Käyttöjärjestelmän ydintoiminnot sisältävä kernel32.dll ja muutamat muut DLL-tiedostot ovat usein Win32-virusten toiminnassaan hyödyntämiä. Win32-rajapinta sisältyy kaikkiin Microsoftin käyttöjärjestelmiin Windows 95:stä alkaen, mukaan lukien myös lähinnä kämmentietokoneissa käytetty Windows CE. Windows CE:n rajapinta on tosin käyttötarkoituksensa vuoksi huomattavasti rajatumpi kuin muissa Windows-versioissa. Muidenkaan Win32-pohjaisten käyttöjärjestelmien rajapinnat eivät ole keskenään aivan samanlaisia. Uusia käyttöjärjestelmiä kehittäessään Microsoft laajentaa rajapintaa uusilla rutiineilla, joita ei luonnollisesti vanhoihin käyttöjärjestelmiin jälkikäteen lisätä. Rajapinnan ydin on kaikissa Win32-käyttöjärjestelmissä sama, ja muut erot huomioiden on mahdollista kirjoittaa virus, joka toimii kaikissa Win32-käyttöjärjestelmissä.

Saman rajapinnan lisäksi Microsoftin käyttöjärjestelmiä yhdistää sama ohjelmatiedostoissa käytetty tiedostomuoto – Portable Executable (PE). PE-tiedostomuoto on pitkälti samanlainen kuin UNIXin COFF-tiedostomuoto (Common Object File Format) ja sitä käyttävät kaikki Win32-käyttöjärjestelmät. Ohjelmatiedostojen lisäksi PE-muotoisia ovat muiden muassa aiemmin mainitut DLL-tiedostot sekä laiteajurit. Nimensä mukaisesti PE-tiedostomuoto on siirrettävä (portable) ja samaa tiedostomuotoa käytetään useilla laitealustoilla. Prosessorin käyttämät operaatiokoodit ovat toki erilaiset eri alustoilla, joten sellaisenaan ei sama ohjelma toimi monilla alustoilla, vaikka sama tiedostomuoto onkin käytössä.

Sama Win32-rajapinta ja saman tiedostomuodon käyttäminen tarjoavat virusten kirjoittajille oivan mahdollisuuden tuotoksille, joille ovat haavoittuvia useat eri käyttöjärjestelmät. Kun vielä Microsoftin käyttöjärjestelmät ovat ylivoimaisesti yleisin PC-tietokoneissa oleva käyttöjärjestelmä, on Win32-viruksen tekeminen houkutteleva vaihtoehto laajaa vahinkoa haluaville virus-nikkareille. Win32-viruksia ilmestyi käyttöympäristön alkuvuosina melko harvakseltaan ennen kuin virusten tekijät oppivat Win32-ympäristön toimintaperiaatteet ja niiden soveltamisen virusten tekemiseksi. 2000-luvulla uusien Win32-virusten ilmestymistahti on moninkertaistunut: vuonna 2001 löydettiin yhteensä 741 uutta virusvarianttia, vuonna 2003 jo 5500 ja pelkästään vuoden 2004 alkupuoliskolla ilmestyi 4500 uutta Win32-virusvarianttia (Szor 2004). Aluksi tekeleet olivat usein riippuvaisia käyttöjärjestelmän versiosta, esimerkiksi ensimmäinen Win32-virus W95/Boza toimi ainoastaan Windows 95:n eräissä englanninkielisissä versioissa (Szor 2005 s. 172–173).

3.3 Portable Executable -tiedostomuoto

3.3.1 Yleistä PE-tiedostojen rakenteesta

PE-tiedostot koostuvat otsakkeista, osiotaulusta ja varsinaisista ohjelmakoodin ja datan sisältävistä osioista. PE-tiedostojen rakenne on turhan monimutkainen tässä tutkielmassa kokonaan käsiteltäväksi, joten jatkossa keskitytään lähinnä virustartuntojen ja niiden havaitsemisen kannalta oleellisiin kohtiin PE-tiedostoissa. PE-tiedostojen tarkasta rakenteesta ei ole juurikaan kirjoitettu. Tässä tutkielmassa PE-tiedostoista esitetyt tiedot perustuvat Microsoftin määritelmään (Microsoft 1999), jota on

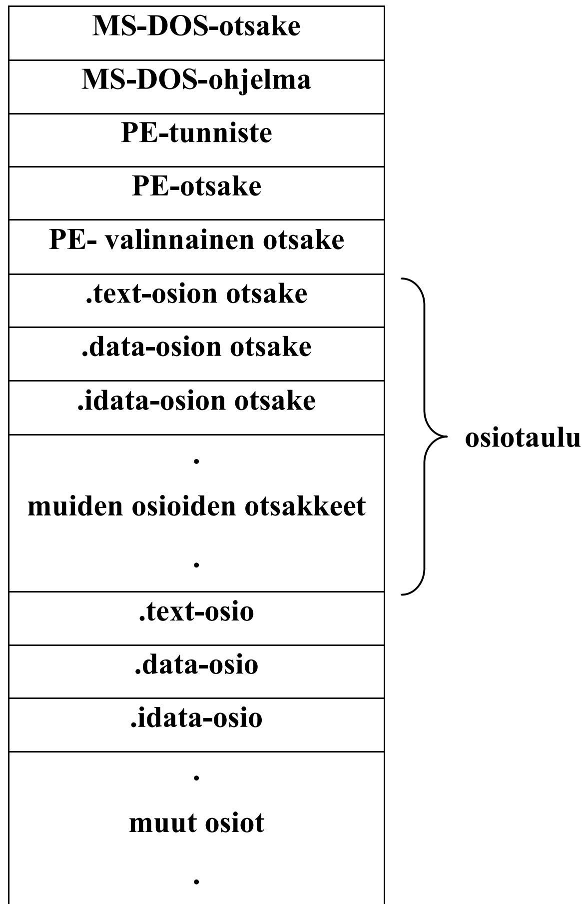
täydennetty itse PE-tiedostoja tutkimalla. PE-tiedostojen analysoinnissa hyödynnettiin PEBrowse Professional -ohjelmaa (Smidgeonsoft 2006). Erityisesti luvussa 3.3.4 käsiteltävän tuontitaulun tarkasta toteutuksesta ei ole kirjoitettu tämän tutkielman vaatimalla tarkkuudella, joten se miten ohjelman ulkopuolisten DLL-tiedostojen tuonti ohjelman käytettäväksi on toteutettu, selvitettiin PE-tiedostoja analysoimalla.

PE-tiedostoissa olevat muistiosoitteet ovat yleensä RVA-muodossa (Relative Virtual Address). Todellinen osoite (Virtual Address, VA) saadaan lisäämällä RVA:han ohjelman alkusoite muistissa. Useimmiten ohjelmat ladataan alkamaan keskusmuistin osoitteesta 00400000h. Esimerkiksi RVA 0200h löytyisi siten muistiosoitteesta 00400200h. Muistiosoitteita käsiteltäessä on syytä huomioida, ettei kaikkea PE-tiedoston sisältämää dataa ladata muistiin ja osioiden koot muistissa ja levyllä poikkeavat tavallisesti toisistaan, joten RVA ei yleensä ilmoita osoitetta tiedoston sisällä, vaan vasta keskusmuistiin ladatun ohjelman sisällä.

Kokonaisluvut ovat käsiteltävän laitealustan prosessoreissa little endian -muodossa. Niissä vähiten merkitsevä tavupari on tiedostossa ensimmäisenä. Otsakkeiden kokonaislukuja sisältävien kenttien tavut ovat siis tiedostoissa käänteisessä järjestyksessä. Esimerkiksi 00135911h on tiedostossa tavujonona 11 59 13 00.

3.3.2 Otsakkeet

PE-tiedoston aivan alussa on MS-DOS-otsake ja minimaalinen MS-DOS-ohjelma, joka antaa virheilmoituksen, jos ohjelmaa yritetään suorittaa MS-DOS-ympäristössä. MS-DOS-otsakkeen – ja samalla PE-tiedoston – ensimmäiset merkit ovat MZ, joita virukset voivat käyttää ohjelmatiedostojen tunnistamiseen. Ainoa nykyisten käyttöjärjestelmien hyödyntämä kenttä ensimmäisessä otsakkeessa on otsakkeen viimeisenä kenttänä osoitteessa 3Ch oleva osoite PE-tunnisteeseen. Tätä e_lfanew-kenttää muuttavat eräät virukset tartunnan yhteydessä. PE-tunniste koostuu merkeistä P ja E sekä kahdesta nolla-tavusta (PE\0\0), ja siitä tunnistetaan PE-muodossa olevat tiedostot. Myös virukset voivat PE-tunnisteen etsimällä varmistaa tiedoston kelpoisuuden isäntäohjelmaksi. Kuvassa 2 on esitetty pääpiirteissään malli PE-tiedoston rakenteesta.



Kuva 2. PE-tiedoston rakenne pääpiirteissään.

PE-tunnistetta seuraa välittömästi 20 tavun mittainen PE-otsake. Otsakkeen tärkein kenttä viruksille on NumberOfSections, joka määrittelee, kuinka monta osiota ohjelmassa on, ja samalla otsakkeita seuraavan osiotaulun (section table) koon. Useat virukset lisäävät itsensä PE-tiedostoon omana osionaan, ja niiden täytyy siis kasvattaa NumberOfSections-kentän arvoa yhdellä. TimeDateStamp-kenttä sisältää tiedoston luontiajankohdan, ja jotkin virukset käyttävät kentässä tiettyä arvoa tartuntamerkkinä välttääkseen tarttumasta samaan ohjelmaan useampaan kertaan. Yleensäkin virukset lisäävät otsakkeiden normaalisti käyttämättömiin kohtiin tartuntamerkkejä. PE-otsakkeen viimeisenä kenttänä on Characteristics, joka koostuu 15 binäärisestä muuttujasta, jotka määrittelevät ohjelman ominaisuuksia. Ominaisuuksista viruksille olennaisin on IMAGE_FILE_DLL, jonka tarkistamalla virukset erottavat varsinaiset ohjelmatiedostot DLL-tiedostoista.

PE-otsaketta seuraa vielä yksi otsake. Viimeistä otsaketta kutsutaan valinnaiseksi otsakkeeksi (optional header), koska sitä ei tarvita kaikissa PE-tiedostoissa, vaan ainoastaan EXE- ja DLL-tiedostoissa. Valinnainen otsake on jaettu kolmeen osaan: vakio-kenttiin, Windows-kenttiin ja data-hakemistoihin. Vakio-kentistä ensimmäinen on ohjelman tyypin ilmoittava luku (magic number). Yleisin arvo kentälle on 010Bh, joka tarkoittaa normaalia 32-bittistä ohjelmaa. Tulevaisuudessa tullaan sen sijaan näkemään 020Bh, joka merkitsee 64-bittistä ohjelmaa. 64-bittisten sovellusten käyttämästä tiedostomuodosta käytetään nimitystä PE32+. Tiedostomuodon otsakkeet ovat lähes samat kuin 32-bittisten käyttämät, joillekin kentille on varattu pitempien muistiosoitteiden vuoksi enemmän tilaa. Lisäksi valinnaisen otsakkeen vakio-kentistä viimeinen (BaseOfData) on poistettu. Virukset voivat käyttää ensimmäistä kenttää varmistaakseen kyseessä olevan normaalin ohjelmatiedoston. Jatkossa kenttä on oletettavasti viruksille tärkeämpi, niiden erotellessa 64-bittiset ohjelmat 32-bittisistä. SizeOfCode on ohjelman kaikkien suoritettavien koodi-osioiden kokonaispituus. Jos virus ei muuta kenttää lisättyään oman koodinsa ohjelmaan, jää kentän arvo epäyhtenäiseksi tiedoston sisällön kanssa. Lopuista seitsemästä (64-bittisen ohjelman tapauksessa kuudesta) kentästä ainoa virusten hyödyntämä on AddressOfEntryPoint, joka määrittelee ohjelman tulokohdan. Viruksille helpoin tapa siirtää suoritus itseensä on vaihtaa juuri AddressOfEntryPoint-kentän arvo osoittamaan oman koodinsa alkuun.

Windows-kenttiä on kaikkiaan 21, ja ne sisältävät lataajan sovelluksen muistiin lataamiseen tarvitsemaa tietoa. Lataajan lisäksi myös virukset ammentavat kentistä toiminnalleen hyödyllistä tietoa. Ensimmäinen Windows-kenttä on ImageBase, joka määrittelee ohjelman alkamisosoitteen keskusmuistissa, olettaen että kyseinen osoite on vapaana. ImageBasen määrittelemä osoite on ikään kuin oletusosoite, johon ohjelma ladattaessa ei lataajan tarvitse laskea ohjelman mahdollisesti sisältämiä absoluuttisia osoitteita uudelleen latauksen yhteydessä. Virukset laskevat ImageBasen avulla eräiden asioiden osoitteita muistissa. Ohjelman eri osiot eivät muistissa ala satunnaisista kohdista, vaan SectionAlignment määrittelee luvun, jolla jaollisia kaikkien osioiden alkamisosoitteet ovat. Virukset käyttävät tätä kenttää laskiessaan viruksen rungolle käypää osoitetta. FileAlignment on SectionAlignmentin vastine osioiden sijainnille tiedostoissa. FileAlignmentin on oltava jokin kahden potenssi 512:ta ja 65536:n välillä.

Kenttä on viruksille tärkeä, mikäli ne lisäävät isäntäohjelmaan uuden osion koodiaan varten. SizeOfImage on lataajan keskusmuistiin lataamien tavujen määrä. Viruksen täytyy kasvattaa lukua saadakseen myös oman tiedoston loppuun lisäämänsä koodin ladatuksi isäntäohjelman mukana. CheckSum-kentässä on tiedoston tarkistussumma, joka on pakollinen ajureille ja DLL-tiedostoille. DLL-tiedostoihin tarttuvien virusten on tartunta kätkeäkseen syytä laskea tarkistussumma uudelleen tiedostoon tartuttuaan. Tarkistussumman laskemiseen on rutiini imagehlp.dll tiedostossa.

Valinnaisen otsakkeen viimeisenä oleva data-hakemisto sisältää tiedoston sisäisten tärkeiden tietorakenteiden osoitteet ja koot niiden nopeaa löytämistä varten. Esimerkiksi tuonti- ja vientitaulujen alkamisosoite ja koko löytyvät datahakemistosta.

3.3.3 Osiotaulu

Osiotaulu seuraa otsakkeita, ja määrittelee kaikki tiedostossa olevat koodi- ja dataosiot. Osiotaulu on sarja 40 tavun mittaisia osioiden otsakkeita. Osion otsake koostuu muun muassa osion nimestä, osion koosta keskusmuistissa (VirtualSize) ja levyllä (SizeOfRawData) sekä alkamisosoitteista keskusmuistissa (VirtualAddress) ja levyllä (PointerToRawData). Muita virusten yleisesti käsittelemiä kenttiä ovat osioiden ominaisuuksia määrittelevät liput. Virusten kannalta oleellisia lippuja ovat osion luettavaksi, kirjoitettavaksi tai suoritettavaksi mahdollistavat – samalla osiolla voi olla kaikki kolme ominaisuutta.

Osiotaulua seuraavat varsinaiset osiot, jotka alkavat valinnaisen otsakkeen FileAlignment-kentän sisällöllä jaollisista osoitteista. Tämän seurauksena osiotaulua seuraa useimmiten ylimääräisiä täytetavuja, jotta ensimmäinen osio alkaa oikeasta kohdasta.

PE-ohjelmatiedostojen osiot voidaan jakaa koodi- ja dataosioihin. Koodiosiot sisältävät suoritettavaa ohjelmakoodia ja dataosiot suorituksessa hyödynnettävää tai käyttöjärjestelmän tarvitsemaa dataa. Koodiosion nimi on useimmiten .text, mutta CODE on myös yleisessä käytössä. Tyypillisiä dataosioita ovat ohjelman muuttujat sisältävä tavallinen, usein .data-niminen osio, ainoastaan luettavissa oleva dataa sisältävä .rdata, tuontitaulun sisältävä .idata, vientitaulun sisältävä .edata ja muita

ohjelman resursseja (esim. kuvia) sisältävä .rsrc. Lataajan tarvitsemat sijoitustiedot (relocation data) ovat .reloc-osiossa. Sijoitustietoja lataaja hyödyntää siinä tapauksessa, jos ohjelma joudutaan lataamaan muuhun kuin ImageBasen ilmoittamaan oletusosoitteeseen, ja ohjelma sisältää absoluuttisia muistiosoitteita. Ohjelman testaukseen ja muuhun virhetilanteiden jäljittämiseen on .debug-osio. Osioista käytetyt nimet ovat merkityksettömiä käyttöjärjestelmälle, nimet toimivat vain tunnisteina ohjelmoijia varten. Viruksille osioista keskeisin on tuontitaulun sisältävä osio, koska virukset tarvitsevat toimiakseen käyttöjärjestelmän tarjoamia palveluita, joita käytetään tuontitaulun kautta.

Datan ja ohjelmakoodin ollessa eroteltuna omiin osioihinsa voidaan kuhunkin osioon antaa vain sen tarvitsemat käyttöoikeudet: koodiosiosta tehdään suoritettava ja dataosiosta kirjoitettava. Virusten tapauksessa koodia ja dataa ei ole eritelty eri osioihin, joten niiden täytyy valmiiseen koodiosioon itsensä lisätessään muuttaa osio kirjoitettavaksi voidakseen käyttää muuttujiaan.

3.3.4 Tuontiosoitetaulu

Sovellusohjelmat eivät kutsu DLL-tiedostojen rutiineja suoraan koodiosiostaan, vaan kierrättävät kutsut tuontiosoitetaulun (Import Address Table, IAT) kautta. IAT on osa DLL-tiedostojen käyttöön tarvittavia tietorakenteita, jotka sijaitsevat usein omassa .idata-osiossaan, mutta saattavat sijaita myös jonkin toisen osion osana (esimerkiksi vain luettavaa dataa sisältävässä osiossa tai koodiosiossa). Tuontiosion alussa on hakemisto (Import Directory Table, IDT), jossa on 20 tavun alkio jokaiselle sovelluksen käyttämälle ulkoiselle DLL-tiedostolle. Tuontiosio on löydettävissä helposti valinnaisen otsakkeen lopussa sijaitsevan hakemiston perusteella. Hakemiston toinen alkio (IMAGE_DIRECTORY_ENTRY_IMPORT) ilmoittaa IDT:n koon ja alkuosoitteen muistissa RVA-muodossa. IDT:n alkioiden oleellisimmat kentät ovat osoite DLL-tiedoston nimen merkkijonoon (Name RVA) sekä osoitteet DLL-tiedoston IAT:hen ja ILT:hen (Import Lookup Table). ILT on taulukko 4 tavun alkioita, jotka sisältävät RVA-muodossa olevat osoitteet kaikkiin sovelluksen kyseisestä DLL-tiedostosta tarvitsemien rutiinien nimiin ja hakua nopeuttaviin järjestysnumeroihin DLL-tiedoston sisällä. Sekä IDT:n että ILT:n loppu määritellään alkiolla, jonka kaikkien kenttien arvot ovat nollia. Varsinainen API-kutsujen lopullisten osoitteiden lähde on IAT, joka saattaa

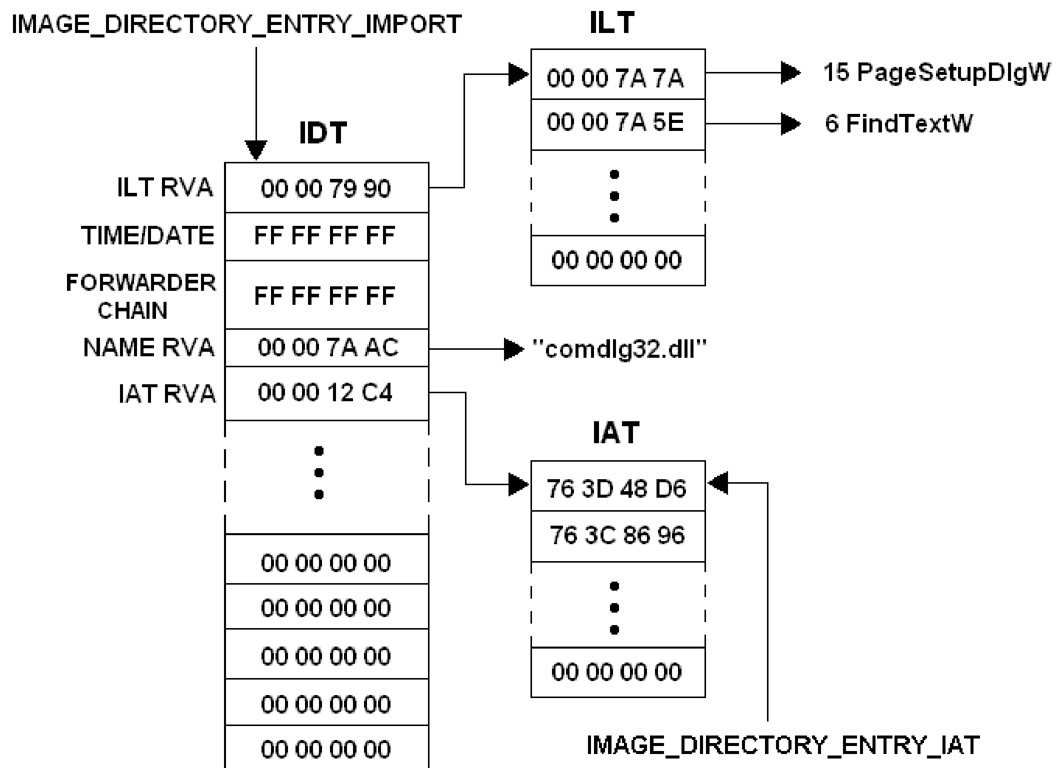
ennen ohjelman latausta olla kopio ILT:stä. IAT:n kentät korvataan lataajan toimesta DLL-rutiinien varsinaisilla osoitteilla. PE-tiedostojen käyttämä menetelmä API-kutsuille nopeuttaa lataajan toimintaa, koska sen tarvitsee korvata vain IAT:ssa sijaitsevat osoitteet oikeilla sovelluksen jok'ikisen DLL-kutsun osoitteiden sijaan. IAT:n nopean löytämisen mahdollistamiseksi sen alkuosoite on valinnaisen otsakkeen lopussa olevassa hakemistossa 13. alkiona (IMAGE_DIRECTORY_ENTRY_IAT). Kuvassa 3 on esitetty APIen tuontiin liittyvät tietorakenteet käyttäen esimerkkinä Windows XP:n notepad-ohjelmaa, jossa IAT:n sisältönä ovat rutiinien todelliset osoitteet. Tilan säästämiseksi on esitetty vain kaksi rutiinia yhdestä DLL-tiedostosta, tavallisesti DLL-tiedostoja on useita ja ulkopuolisia rutiineja on yhteensä vähintään kymmeniä.

DLL-tiedostojen rutiineja kutsutaan koodiosiosta CALL-käskyllä. CALL-käsky kutsuu koodiosiossa sijaitsevaa JMP-käskyä, joka hoitaa varsinaisen API-kutsun. JMP-käskyn osoite on osoitin IAT:hen, jossa API-käskyn osoite sijaitsee. Ohjelmalistauksessa 4 on esimerkki API-kutsujen toteutuksesta, osoite 00405030h sijaitsee IAT:ssä ja sisältää halutun APIn todellisen muistiosoitteen. (Pietrek 2002a)

Ohjelmalistaus 4. API-kutsujen toteutus Win32-ohjelmissa (Pietrek 2002a).

```
CALL 0x0040100C
•••
0x0040100C:
JMP       DWORD PTR [0x00405030]
```

Virukset saavat useiden tarvitsemiensa API-kutsujen osoitteet kätevästi suoraan isäntäohjelmansa tuontiosoitetaulusta. Nykyiset virukset päivittävät lisäksi taulukkoon muut käyttämänsä API-kutsut, jotka eivät siellä valmiiksi ole. Lisäksi virukset voivat hyödyntää tuontiosoitetaulua EPO-menetelmiä käyttäessään, Win32-virusten EPO-menetelmiä käsitellään luvussa 3.5.



Kuva 3. PE-tiedostomuodon ulkopuolisten rutiinien sovellukseen tuonnissa käyttämät tietorakenteet.

3.4 Win32-tiedostovirusten tartuntamenetelmät

Win32-tiedostovirukset tarttuvat tiedostoihin samoilla periaatteilla kuin on esitetty luvussa 2. Tässä luvussa käsitellään tartuntamenetelmien tarkempaa toteutusta PE-muotoisissa tiedostoissa.

Päällekirjoittavat virukset tekevät isäntäohjelmastaan toimintakelvottoman ja ovat siksi erittäin helposti havaittavia. Tämän vuoksi yhtään ohjelmatiedostoja päällekirjoittavaa Windows 95 -virusta ei tiettävästi tehty (Szor 2005 s. 174). Myöhemmille Win32-käyttöjärjestelmille on kuitenkin olemassa yksinkertaisia päällekirjoittavia viruksia. Päällekirjoittavat virukset on usein tehty korkean tason ohjelmointikielellä (esim. W32/HLLO.Marion joka on ohjelmoitu Visual Basicillä), jolloin viruksen tekeminen on yksinkertaiseen toimintaperiaatteeseen yhdistettynä mahdollisimman helppoa.

Helpoin tartuntatapa, joka pitää isäntäohjelman toimivana on viruksen isäntäohjelman alkuun lisäävä loistartunta. Tällöin virus siirtää kokoisensa palan isäntäohjelman alusta isäntäohjelman loppuun, ja lisää itsensä syntyneeseen tilaan. Viruksen ei näin toimiessaan tarvitse kajota PE-tiedoston rakenteeseen otsakkeita tai muita osioita muuttaen. Isäntäohjelman ajaakseen virus kopioi sen alkuperäiseen muotoonsa tilapäistiedostoon, jonka virus ajaa oman suorituksensa jälkeen.

Isäntäohjelman loppuun viruksen lisäävät loistartunnat voidaan toteuttaa kolmella tavalla: lisäämällä virus omana osionaan, lisäämällä virus isäntäohjelman jonkin olemassa olevan osion (useimmiten viimeisen) jatkeeksi tai lisäämällä virus omine otsakkeineen tiedoston loppuun.

Itsensä isäntäohjelmiin uutena osiona lisäävien virusten täytyy lisätä osiotauluun kokonaan uusi otsake omalle osiolleen. Virus voi lisätä oman otsakkeensa osiotauluun vain, jos osiotaulun päätteenä olevia täytetavuja on otsakkeen vaatima määrä. Osiotaulun päivittämisen lisäksi viruksen täytyy kasvattaa PE-otsakkeen NumberOfSections-kentän arvoa yhdellä. Huomaamattomampi tapa virukselle on sisällyttää itsensä johonkin valmiista osioista kokonaan uuden osion sijaan. Tällöin tartunta onnistuu aina, eikä ainoastaan silloin, jos uuden osion otsakkeelle on tarpeeksi tilaa. Isäntäohjelman viimeiseen osioon itsensä kopioidessaan täytyy viruksen kasvattaa viimeisen osion kokoa osiotaulussa (VirtualSize- ja SizeOfRawData-kentät). Johonkin aiempaan osioon kopioituessaan täytyy viruksen lisäksi siirtää kaikkia myöhempiä osioita eteenpäin sekä levyllä että muistissa. Mikäli osion alkuperäiset ominaisuudet eivät riitä viruksen toiminnalle, täytyy viruksen mahdollisesti muuttaa tartuttamansa osio kirjoitettavaksi tai suoritettavaksi. Molemmissa jo esitetyissä loppuun lisäävissä tartuntamenetelmissä täytyy viruksen kasvattaa myös valinnaisen otsakkeen SizeOfImage-kentän arvoa tarvittavalla määrällä sekä muuttaa AddressOfEntryPoint-kentän arvo osoittamaan omaan tulokohtaansa. Kolmas loppuun lisäävä loisvirustyyppi perustuu MS-DOS-otsakkeen e_lfanew-kentän muuttamiseen. Virus lisää itsensä tartutettavan ohjelmatiedoston loppuun kokonaisena itsenäisesti toimivana ohjelmana, jolla on omat otsakkeensa ja tarvitsemiensa API-rutiinien tuontitiedot omassa osiossaan. Isäntäohjelmastaan virus muuttaa ainoastaan MS-DOS-otsakkeen e_lfanew-kentän

osoittamaan isäntäohjelman PE-tunnisteen sijaan omaan PE-tunnisteeseensa. Tartutettu isäntäohjelma käynnistettäessä kontrolli siirtyy suoraan virukselle. Isäntäohjelman virus saa suoritettua luomalla tilapäistiedoston, jonka sisältönä on isäntäohjelma alkuperäisellä e_lfanew-kentällä varustettuna. Tällainen tarttumistapa on loisviruksille yksinkertaisin ja siksi yleisessä käytössä (Szor 2005 s. 178).

Osiotaulun jälkeiseen mahdolliseen tyhjään tilaan voi virus uuden otsakkeen sijaan lisätä koko viruksen. Tällaisten otsakevirusten täytyy olla hyvin pieniä, koska osiotaulun ja ensimmäisen osion väliin ei jää juurikaan tilaa. Tyhjän tilan koon ratkaisevat ohjelmassa olevien osioiden määrä (jokaisen osion otsake vie 40 tavua osiotaulussa) ja FileAlignment-kentän arvo (osiot alkavat kentän arvolla jaollisista osoitteista). Otsakevirukselle jäävä tila on korkeintaan muutamia satoja tavuja, ensimmäinen otsakevirus W95/Murkry olikin siksi vain alle 400 tavun mittainen (www.viruslist.com 2000). PE-otsakkeen kentistä otsakeviruksen täytyy muuttaa ainoastaan valinnaisen otsakkeen AddressOfEntryPoint osoittamaan oman koodinsa alkuun. Koska otsakevirukset lisäävät itsensä käyttämättömään tilaan isäntäohjelmassa, voitaneen niitä pitää onkaloviruksina. PE-tiedostomuoto tarjoaa mahdollisuuden myös osioitujen onkalovirusten tekemiseen. Osioiden täytyy alkaa valinnaisen otsakkeen FileAlignment-arvolla jaollisista osoitteista. Osioiden päätteeksi linkkeri lisää täytettä (yleensä 00h- tai CCh-tavuja), jotta seuraava osio alkaa laillisesta osoitteesta. Virukset voivat korvata täytetavut omalla koodillaan vaikuttamatta isäntäohjelman suoritukseen.

3.5 Win32-tiedostovirusten EPO-menetelmät

AddressOfEntryPoint-kentän muuttamisen sijaan virus voi muuttaa isäntäohjelman tulokohdassa olevat tavut hyppykäskyksi omaan koodiinsa. Kyseinen menetelmä on eräänlainen EPO-menetelmien esiaste. PE-tiedostojen tapauksessa pelkkä tavujen korvaaminen ja tallentaminen isäntäohjelman myöhempää suorittamista varten ei kuitenkaan riitä. Kaikilla ohjelmilla on oletusosoite, johon ohjelma ladattaessa kaikki ohjelman sisältämät muistiosoitteet ovat ohjelman toiminnan kannalta oikeita. Mikäli ohjelma ladataan alkamaan jostakin muusta muistiosoitteesta, täytyy lataajan mahdollisesti muuttaa joitakin muistiosoitteita huomioimaan oletusosoitteen ja varsinaisen osoitteen erotuksen. Näiden siirtojen tekoon lataaja käyttää .reloc-osiota, joka sisältää tiedon siitä mitkä ohjelman kohdat ovat riippuvaisia ohjelman

aloitusosoitteesta keskusmuistissa. EXE-tiedostojen tapauksessa siirtojen tarpeellisuus on harvinaista, koska ohjelmat ladataan ensimmäisenä ja ne pääsevät näin lähes poikkeuksetta alkamaan oletusosoitteestaan, DLL-tiedostojen tapauksessa latausjärjestys vaihtelee ja lataaja joutuu mahdollisesti turvautumaan siirtoihin (Pietrek 2002b). Joka tapauksessa ohjelmien käyttämät muistiosoitteet ovat valtaosaltaan ohjelman alkamisosoitteesta riippumattomassa RVA-muodossa. Korvatessaan isäntäohjelman alun hyppykäskyllä itseensä täytyy viruksen kuitenkin huolehtia siitä, ettei .reloc-osio sisällä osoitteita korvattavaan ohjelmakohtaan. Muuten lataaja saattaa tehdä hyppykäskyyn muutoksia ja tehdä viruksesta ja samalla isäntäohjelmasta toimintakyvyttömän.

Varsinaiset EPO-virukset käyttävät usein Win32-ympäristölle oleellisia API-kutsuja siirtääkseen suorituksen itseensä. EPO-viruksen on suhteellisen yksinkertaista tunnistaa API-kutsut sen perusteella osoittavatko ne tuontitauluun, ja korvata jokin isäntäohjelman tekemistä API-kutsuista kutsulla omaan koodiinsa. Virus tallentaa alkuperäisen kutsun oman koodinsa loppuun, ja ajaa sen suorituksensa jälkeen pitääkseen isäntäohjelman toiminnallisena. Valitsemalla sopivan API-kutsun korvattavaksi, voi virus välttää tilanteen, jossa se tulee suoritetuksi vain harvoin. Viruksen kannalta hyvä valinta on esimerkiksi ExitProcess, jota kutsutaan aina ohjelman suoritus lopetettaessa. Tällöin virus tosin tulee suoritetuksi vasta isäntäohjelman suorituksen päättyessä, eikä ennen isäntäohjelmaa kuten yleensä – viruksen toimintaa tämä ei mitenkään estä. Kaikki virukset eivät välitä suorituksensa varmistamisesta, vaan korvaavat jonkin satunnaisen API-kutsun. (Szor 2005 s. 150–151)

Sovelluksen tekemien API-kutsujen sijaan virus voi muuttaa IAT:n sisältöä. Korvaamalla IAT:n sisältämien API-kutsujen osoitteet oman tulokohtansa osoitteella takaa EPO-virus suorituksensa jokaisella isäntäohjelman ajokerralla. Alkuperäiset API-kutsujen osoitteet virus kopioi oman koodinsa jatkeeksi, jotta se voi luoda keskusmuistiin isäntäohjelman tarvitseman alkuperäisen IAT:n. (Szor 2005 s. 153)

IAT:n sisältöä muuttava EPO-virus on esimerkiksi W32/Idele. W32/Idele.2108 korvaa isäntäohjelman IAT:stä kaikki kernel32.dll:n funktioiden muistisoitteet oman tulokohtansa osoitteella.

API-kutsujen kohteiden vaihtamisen sijaan tulokohtansa piilottava Win32-virus voi korvata myös isäntäohjelman sisällä tapahtuvan aliohjelmakutsun. Aliohjelmakutsua ei voi tunnistaa pelkän CALL-käskyn tavujonon perusteella, koska samat tavut voivat olla myös osa muuta ohjelmakoodia. Varmistaakseen löytäneensä juuri CALL-komennon, virus tarkistaa oletetun CALL-komennon kohteesta löytyvät tavut. Aliohjelmat suorittavat aina ensimmäisinä komentoinaan kutsuvan aliohjelman pinokehysosoittimen (base pointer) tallentamisen pinoon, ja uuden pinokehysosoittimen muodostamisen. Symbolisena konekielenä komennot ovat `push ebp` ja `mov ebp, esp`. Operaatiokoodit, joilla halutut komennot aikaansaadaan, vaihtelevat ohjelmoinnissa käytetyn kääntäjän mukaan ja ovat joko 55h 89h E5h tai 55h 8Bh ECh. Näitä tavujonoja etsimällä virus voi varmistaa löytäneensä CALL-käskyn, jonka se voi korvata kutsulla itseensä. (Szor 2005 s. 151–152)

3.6 Yhteenveto Win32-tiedostoviruksista

Win32-käyttöympäristö tarjoaa viruksille useamman eri käyttöjärjestelmän kattavan alustan jolla toimia. Win32-alustan käyttämän tiedostomuodon rakenne on tarkasti määritelty ja sisältää useita kenttiä, joita virusten on pakko muuttaa saadakseen lisättyä itsensä isäntäohjelman toimivaksi osaksi. PE-tiedostomuoto tarjoaa Win32-tiedostoviruksille eri tapoja toteuttaa luvussa 2 esiteltyjä tartuntamenetelmiä muuttamalla otsakkeiden ja muiden tietorakenteiden arvoja. Samoja otsakkeita ja tietorakenteita analysoimalla myös virustentorjuntaohjelmat voivat pyrkiä löytämään virustartuntaan viittaavia seikkoja. Luvussa 5 esitettävät heuristiikat perustuvat juuri tässä luvussa esitettyihin virusten toimintatapoihin ja PE-tiedoston ominaisuuksiin.

4 TIEDOSTOVIRUSTEN ETSIMINEN

4.1 Johdanto

Tiedostovirusten etsimiseen on vuosien mittaan kehitetty entistä monimutkaisempia menetelmiä uudentyyppisten virusten aiheuttamien uusien haasteiden vuoksi. Tässä luvussa käydään läpi nykyisin käytössä olevia menetelmiä alkaen perinteisistä merkkijonohauista ja niiden parannetuista versioista. Edistyneempien virusten löytämiseen esitetään algoritmiset ja X-RAY-menetelmät sekä koodin emulointi. Läpi käydään myös ennalta määrättyjen virusten löytämisen sijaan virusohjelmien yleiseen tunnistukseen pyrkivät heuristiset menetelmät.

4.2 Merkkijonohaut

4.2.1 Yksinkertaiset merkkijonohaut

Vanhin ja yksinkertaisin tapa virusten etsimiseen on merkkijonohaku. Ohjelmatiedostot ovat kääntäjän jäljiltä prosessorin ymmärtämässä konekielisessä muodossa, jossa komennot ovat heksamuodossa olevia operaatiokoodeja. Operaatiokoodeista muodostuu merkkijonoja, joita virustutkat voivat käyttää viruksen etsimiseen. Tunnistamiseen käytettävän merkkijonon täytyy olla tarpeeksi pitkä, jotta todennäköisyys sen esiintymiseen sattumalta jossain muussa yhteydessä kuin osana virusta on pieni. 16-bittisten alustojen viruksista on yleensä käytetty 16-tavuisia allekirjoituksia (Szor 2005 s. 430). Laitearkkitehtuurin bittimäärän kasvaessa allekirjoitusta on syytä pidentää. Allekirjoituksen valinta suoritetaan siten, että allekirjoitukseen tulevat tavut ovat viruksen toiminnan kannalta mahdollisimman keskeisiä, eivätkä esimerkiksi viruksen tekijän koodin sekaan lisäämää tekstiä. Kun allekirjoitus kattaa viruksen keskeisen osan, löytää sama allekirjoitus mahdollisimman hyvin myös samasta viruksesta myöhemmin tehtävät variantit. Ohjelmalistauksessa 5 on esimerkkinä Stoned-viruksen se osa, josta allekirjoitus on muodostettu (0400 B801 020E 07BB 0002 33C9 8BD1 419C).

Ohjelmalistaus 5. Stoned-viruksen allekirjoituksen muodostava osa.

```
BE0400         MOV     SI,0004
B80102         MOV     AX,0201
0E             PUSH    CS
07             POP     ES
BB0002         MOV     BX,0200
33C9           XOR     CX,CX
8BD1           MOV     DX,CX
41             INC     CX
9C             PUSHF
```

Merkkijonohakuja voidaan laajentaa jokerihauiksi (wildcard) sallimalla yhden tai useamman tavun arvon olla minkä tahansa. Toinen vaihtoehto on IBM:n kehittämä mismatch-menetelmä. Sitä käytettäessä haettavassa tavujonossa sallitaan olevan tietty määrä vääriä tavuja eikä väärien tavujen sijaintia ole rajattu toisin kuin jokerihauissa. Variaatioita merkkijonoihin sallimalla voidaan paremmin löytää toisiinsa pohjautuvia viruksia, mutta haittapuolena haut hidastuvat. Hakuja voidaan nopeuttaa hash-funktioilla, joiden käyttö voidaan rajata ensimmäiseen tavuun jokerihakujen mahdollistamiseksi. Hash-funktiota käytettäessäkin hitaus on merkkijonohakujen suurin ongelma.

Jokerihaut mahdollistavat itsekoodaavien virusten löytämisen merkkijonohauilla. Kuvassa 4 on W95/Madin purkuohjelma ja kuvassa 5 purkuohjelmasta muodostettu merkkijono (alkaen osoitteesta 0040421Fh), jolla virusta voidaan etsiä. Viruksen rungon alkamisosoite ja salausavain täytyy jättää merkkijonosta pois, koska ne muuttuvat tartunnan mukaan. Lisäksi viruksen rungon pituus voidaan jättää haun ulkopuolelle, jotta tulevaisuudessa mahdollisesti tehtävät viruksen rungoltaan erilaiset variantit löydetään samalla haulla.

Itsekoodaavien virusten purkuohjelmien lisäksi joidenkin heikkojen polymorfisten virusten purkuohjelmia voidaan hakea samantyyppisillä jokerihauilla. Hakukriteerien laventaminen johtaa kuitenkin myös väärien hälytysten lisääntymiseen ja monimutkaisten virusten etsimiseen on olemassa parempiakin menetelmiä, joita esitellään luvusta 4.3 alkaen.

```
00404200                        public start
00404200                     start:
00404200 E8 00 00 00 00         call $+5
00404205 5F                     pop  edi
00404206 8B C7                  mov  eax, edi
00404208 2D 05 22 00 00         sub  eax, 2205h       ; variable intruction operand!
0040420D
0040420D                     init:                    ; CODE XREF: .reloc:00404283↓j
0040420D 81 EF 05 30 40 00      sub  edi, 403005h
00404213 89 87 3C 30 40 00      mov  [edi+40303Ch], eax ; entry of host
00404219 89 AF 40 30 40 00      mov  [edi+403040h], ebp ; saved for later use
0040421F 8B EF                  mov  ebp, edi
00404221 33 C0                  xor  eax, eax
00404223 BF 45 30 40 00         mov  edi, 403045h
00404228 03 FD                  add  edi, ebp         ; adjust EDI to "decrypted"
0040422A B9 6B 0A 00 00         mov  ecx, 0A6Bh       ; number of bytes to decrypt
0040422F 8A 85 44 30 40 00      mov  al, [ebp+403044h] ; pick key (constant byte)
00404235
00404235                     decrypt:                 ; CODE XREF: .reloc:00404238↓j
00404235 30 07                  xor  [edi], al        ; decrypt current byte
00404237 47                     inc  edi              ; position to next byte
00404238 E2 FB                  loop decrypt
0040423A EB 09                  jmp  short near ptr decrypted
0040423A                     ; ---------------------------------------------------
0040423C 00 10 40 00            dd 401000h            ; stored host EP
00404240 78 FF 63 00            dd 63FF78h            ; stored EBP
00404244 7B                  key db  7Bh ; {
00404245 BD                  decrypted db 0BDh ; +    ; CODE XREF: .reloc:0040423A↑j
00404246 FE                     db 0FEh ; ¦
00404247 28                     db  28h ; (
00404248 42                     db  42h ; B
00404249 3B                     db  3Bh ; ;
0040424A 7B                     db  7Bh ; {
0040424B 7B                     db  7Bh ; {
```

Kuva 4. W95/Madin purkuohjelma (Szor 2005 s. 445).

8BEF 33C0 BF?? ???? ??03 FDB9 ??0A 0000 8A85 ???? ???? 3007 47E2 FBEB

viruksen rungon alkamisosoite — salausavaimen valinta

viruksen rungon pituus

Kuva 5. Tunnisteeksi kelpaava merkkijono W95/Madin purkuohjelman hakemiseen (Szor 2005 s. 445).

Nykyisin tietojenkäsittelyssä levyoperaatiot ovat useimmiten pullonkaulana prosessorinopeuden sijaan. Kun vielä levyjen tallennuskapasiteetit ovat parantuneet hakuaikoja nopeammin, on merkkijonohakujen nopeuttamiseen ollut pakko löytää keinoja. Perinteisesti tiedostovirukset ovat pieniä, keskimäärin 4 kilotavun kokoisia (Nachenberg 1997), ja liittävät itsensä ohjelman alkuun tai loppuun. Rajaamalla merkkijonohaut tiedoston alku- ja loppuosiin voidaan virusten etsimistä nopeuttaa huomattavasti. Sen sijaan, että jokaista virustutkan tietokannassa olevaa allekirjoitusta verrattaisiin tiedoston alusta saakka alkukohtaa aina tavu eteenpäin siirtäen, voidaan allekirjoituksia verrata vain tiedoston ensimmäisiin ja viimeisiin muutamiin kilotavuihin. Tällä top–tail-menetelmällä merkkijonovertailu saadaan nopeutettua

murto-osaan koko tiedostoon tapahtuvan vertailun viemästä ajasta. Voitetun ajan määrää ohjelmatiedostojen keskimääräinen koko: mitä suurempia ohjelmatiedostot ovat, sitä enemmän aikaa top–tail-menetelmällä voitetaan. Hakua voidaan entisestään nopeuttaa ottamalla tyypilliset tulokohdat huomioon ja aloittaa virusten etsintä isäntäohjelman tulokohdasta. Jotkin virukset siirtävät suorituksen itseensä hyppykäskyllä isäntäohjelman tulokohdasta tai lisäävät tulokohtaan roskakäskyjä ennen omaa koodiaan. Siksi tulokohdan komentoja voidaan seurata hetken ja tarkistaa mahdollisten hyppykäskyjen päätepiste virusten varalta. Tulokohtiin keskittyvä tarkistus nopeuttaa merkkijonohakua vielä huomattavasti tiedoston ääripäihin rajoittuvaa tarkastusta enemmän. Tulokohtia tarkastettaessa merkkijonoa tarvitsee verrata ainoastaan yhteen kohtaan ohjelmatiedostoa. Verrattaessa esimerkiksi 16 tavun merkkijonoa 2048 tavun osiin tiedoston alusta ja lopusta täytyy merkkijonoa verrata 2*(2048-16) kertaa. Tulokohtien tarkastuksen variaationa voidaan viruksia etsiä niistä kullekin tyypillisestä kohdasta. Virustutkan tietokantaan tallennetaan jokaisen allekirjoituksen yhteyteen osoite, johon allekirjoitusta verrataan jokaisessa tiedostossa. Vertausosoitteen lisäksi voidaan määritellä etäisyys, jonne asti allekirjoitusta alkuosoitteesta etsitään. Etsintäalueen rajaaminen tai tulokohtiin keskittyminen nopeuttavat etsintää huomattavasti, mutta esimerkiksi EPO-virukset jäävät löytämättä.

Kuten polymorfisia viruksia käsitelleessä luvussa mainittiin, käyttävät ne turhia roskakäskyjä muuttaakseen purkuohjelmaansa. Toinen roskakäskyjä käyttävä viruslähde ovat eräät automaattiset virusrakennussarjat (virus generation kit). Nämä ohjelmat luovat käyttäjän valintojen perusteella aina edellisestä poikkeavan viruksen. Käyttäjän valittavissa ovat yleensä esimerkiksi viruksen aiheuttama vahinko, sen käyttämä salaus ja leviämisnopeus. Valinnanvara vaihtelee rakennussarjan ominaisuuksien mukaan. Älykäs etsintä (smart scanning) on menetelmä, jossa roskakäskyt jätetään pois allekirjoituksesta ja jätetään huomioimatta ohjelmakoodissa. Älykästä etsintää käytettäessä allekirjoitukseen ei voida sisällyttää dataa sisältäviä tavuja, koska roskakäskyt siirtävät datan osoitteita ohjelmatiedostossa ja siten muuttaisivat allekirjoitusta tartunnasta toiseen.

4.2.2 Tunnistustarkkuuden parantaminen

Saman viruksen varianttien tunnistamisen lisäksi allekirjoitus saattaa kattaa myös joitain täysin toisia viruksia. Tämä ei välttämättä ole pelkästään hyvä asia. Mikäli virustutka tunnistaa viruksen vaarattomaksi yksilöksi, mutta todellisuudessa virus on tuhoa aiheuttava, saattaa käyttäjän reaktio osoittautua riittämättömäksi. Lisäksi eri virukset vaativat ohjelmia puhdistettaessa eri toimenpiteitä, joten väärästä tunnistuksesta saattaa isäntäohjelmaa puhdistettaessa olla seurauksena toimimaton ohjelma.

Virusten tunnistuksen tarkkuutta voidaan parantaa käyttämällä tunnistamisessa allekirjoitusten lisäksi kirjanmerkkejä (bookmark). Kirjanmerkit ovat viruksen allekirjoituksen ulkopuolisia tavuja, joilla saman allekirjoituksen omaavat virukset voidaan erottaa toisistaan. Kirjanmerkkien sijainti lasketaan yleensä suhteessa viruksen alkuun eikä koko ohjelman alkuun. Siksi viruksen allekirjoituksen ja viruksen ensimmäisen tavun (nolla-tavu, zero byte) etäisyys tallennetaan virustentorjuntaohjelman tietokantaan yhdessä viruksen allekirjoituksen kanssa. (Szor 2005 s. 433)

Yksittäisten ylimääräisten tarkistustavujen lisäksi voidaan käyttöön ottaa toinen tunnistuksessa käytettävä merkkijono. Tiedoston puhdistaminen viruksesta suoritetaan vain mikäli molemmat merkkijonot täsmäävät. Merkkijonojen sijaan voidaan hyödyntää tarkistussummia. Tarkistussumma otetaan osuudesta, joka erottaa erityisen hyvin toisistaan viruksen variantit, jotka vaativat erilaiset puhdistustoimenpiteet. Tarkistussummilla päästään merkkijonoja parempaan tarkkuuteen, koska tarkistussumma voidaan laskea pidemmästä tavujonosta ilman, että se vaatii enemmän tilaa virustutkan tietokannasta. Pidempiä merkkijonoja käytettäessä tietokanta turpoaa, ja lisäksi pitkien merkkijonojen hakeminen tiedostoista on useimmiten entistäkin hitaampaa. Tarkistussummia voidaan käyttää pelkästään tarkkaan tunnistamiseen merkkijonohaun tukena, tai merkkijonohaut voidaan korvata kokonaan tarkistussummilla. Kahdella merkkijonolla tai tarkistussummalla saadaan aikaan lähes täsmällinen tunnistus (nearly exact identification). Laajentamalla tarkistussumman kattamaa aluetta koko viruksen peittäväksi, saadaan suoritettua täsmällinen tunnistus (exact identification). Mikäli viruksessa on muuttujia, joiden arvo vaihtelee tartunnasta

tai käyttöympäristöstä toiseen, täytyy ne rajata tarkistussumman ulkopuolelle. Viruksen tarkistussumma lasketaan tällöin useammasta erillisestä alueesta. Virustutkan tietokantaan tallennetaan jokaisen viruksen yhteyteen alueet joista sen tarkistussumma lasketaan, sekä itse tarkistussumma. Varsinainen virusten etsintä hoidetaan nopean haun saavuttamiseksi jollain muulla menetelmällä, ja koko viruksen kattavaa tarkistussummaa käytetään vain tunnistamisessa.

4.3 Algoritmiset menetelmät

4.3.1 Yleistä algoritmisista menetelmistä

Algoritminen menetelmä on nimenä melko epämääräinen, koska algoritmejahan edellisen luvun merkkijonohautkin ovat. Tässä yhteydessä algoritmisilla menetelmillä kuitenkin tarkoitetaan jonkin yksittäisen viruksen saalistamiseen varta vasten kehitettyä menetelmää. Alun perin erilliset etsintämenetelmät yhdistettiin osaksi virustutkan runkoa. Tämän seurauksena ohjelmasta tuli helposti epävakaa ja vaikeasti ylläpidettävä. Erityisen ongelmallista on virustentorjunta-alalle tyypillinen kiire, jonka seurauksena suoraan ohjelman ytimeen osaksi asennettavat yksittäiset lisät saattoivat sisältää koko ohjelman kaataneita virheitä. Myös yksittäisten virusten hakumenetelmien siirrosta eri laitealustoille tuli työlästä, koska kaikki algoritmit piti toteuttaa uudelleen. (Szor 2005 s. 441–442)

Algoritmien yhtenäistämiseksi niitä varten voidaan kehittää oma hakukielensä. Hakukieli kattaa virusten tiedostoista etsimiseksi tarvittavat operaatiot, kuten esimerkiksi tiedoston tietystä kohdasta luvun ja hyppykäskyjen tai aliohjelmakutsujen kohteeseen siirtymisen. Siirrettävyys-ongelma voidaan ratkaista toteuttamalla algoritmit siirrettävänä tavukoodina, jota prosessorin sijaan suorittaa virtuaalikone. Virtuaalikone tulkkaa siirrettävää koodia prosessorin ymmärtämäksi konekieleksi. Samaa siirrettävällä koodilla toteutettua algoritmia voidaan silloin käyttää millä tahansa laitealustalla, kunhan virtuaalikone on alustalle kertaalleen toteutettu. Tällaisen tulkkauksen avulla toteutetun järjestelmän haittapuolena on sen huomattava hitaus verrattuna suoraan konekielen suorittamiseen. Hitaudesta päästään oletettavasti tulevaisuudessa eroon ajonaikaisella kääntämisellä (just-in-time compilation, JIT). JITiä käytettäessä virtuaalikone ei suorita tavukoodia, vaan tavukoodi käännetään konekieleksi ohjelmaa

suoritettaessa. Kääntäminen voidaan suorittaa joko kerralla koko ohjelmalle, tiedosto kerrallaan tai vielä pienemmissä osissa sitä mukaa, kun ohjelman eri osia tarvitaan. Ajonaikaisella kääntämisellä saavutetaan siis tavukoodin tuoma siirrettävyys ilman tulkkaamisen aiheuttamaa heikkoa suorituskykyä. (Szor 2005 s. 442)

Algoritmisia menetelmiä käytetään tyypillisesti löytämään viruksia, joiden etsiminen yleisemmillä menetelmillä on vaikeaa. Metamorfiset tai vaikeat polymorfiset ja EPO-menetelmiä käyttävät virukset on usein mahdollista löytää luotettavasti vain virukselle varta vasten kehitetyllä algoritmisella menetelmällä.

4.3.2 Suodatus

Jokaisen kohteen tarkistaminen kaikkien tunnettujen virusten varalta on turhaa, koska yksikään virus ei tartu kuin pieneen joukkoon kaikista mahdollisista kohteista. Virusten etsintää voidaan nopeuttaa tallentamalla virustutkan tietokantaan tiedot kunkin viruksen tarttumiskohteista. Erityisen tärkeää suodatus on algoritmisia menetelmiä käytettäessä, koska ne ovat merkkijonohakuja monimutkaisempia ja siksi hitaampia.

Karkein suodatin on virustyyppi, esimerkiksi käynnistyslohkosta ei ole tarvetta hakea muita kuin käynnistyslohkoviruksia. Tiedostotyyppi on toinen yksinkertainen suodatin, EXE-tiedostoihin tarttuvia viruksia on turha etsiä muunlaisista tiedostoista ja makroviruksia puolestaan vain niiden potentiaalisista tarttumiskohteista. Tarkempia viruskohtaisia suodattimia voidaan tehdä joko viruksen tarttumisen yhteydessä isäntäohjelmaan tekemistä muutoksista tai viruksen tarttumiseensa asettamista rajoituksista. Virukset tekevät tartunnan yhteydessä usein pieniä muutoksia, joilla ne tunnistavat jo tartuttamansa tiedostot ja välttävät näin tarttumasta samaan tiedostoon useaan kertaan. Samoja muutoksia voivat virustutkat käyttää etsiessään viruksen mahdollisesti tartuttamia tiedostoja lähempään tarkasteluun. Tyypillisiä merkkejä viruksesta ovat muutoin mahdoton päivämäärä tai kellonaika (esim. sekuntien arvona 62), otsakekentissä olevat ylimääräiset merkinnät tai jonkin viruksen aina käyttämä ohjelma-osion nimi. Viruksen tartunta rajautuu usein tiedostotyypin lisäksi tarkemmin ohjelmatiedostotyyppiin. Nykyisin PE-ohjelmatiedostot ovat lähes aina tartunnan kohteina, mutta PE-tyypeistä virukset tarttuvat useimmiten ainoastaan aitoihin ohjelmatiedostoihin, eivätkä DLL-tiedostoihin, jotka myös ovat PE-muodossa. Muita

virusten usein tekemiä tarkistuksia ennen tartuntaa ovat kohdeohjelman minimi- ja maksimikoko sekä tiettyjä merkkijonoja sisältävien tiedostonnimien rajaaminen tartunnan ulkopuolelle.

Esimerkiksi W95/Zmistin etsimisessä suodatus auttaa erittäin paljon. W95/Zmistin algoritminen etsiminen on viruksen poikkeuksellisen EPO-tekniikan vuoksi erittäin hidasta (ks. s. 28), mutta useimmat tiedostot voidaan kuitenkin nopeasti rajata epäiltyjen listalta. W95/Zmist nimittäin tarttuu tiedostoon vain, jos

- tiedoston ensimmäiset merkit ovat MZ (kyseessä on ohjelmatiedosto),
- tiedosto on PE-muotoinen ja
- ohjelmatiedoston koko on vähemmän kuin 448 kilotavua.

Lisäksi W95/Zmist tarttuessaan lisää tiedoston ensimmäisen otsakkeen osoitteeseen 1Ch Z-kirjaimen. (Ferrie ym. 2001)

4.4 X-RAY-menetelmät

Kuten luvussa 4.2.1 mainittiin, itsekoodaavia ja joitakin polymorfisia viruksia voidaan etsiä jokerihauilla niiden purkuohjelman perusteella. Verrattain lyhyistä purkuohjelmista haussa käytettäviä merkkijonoja muodostettaessa väärien hälytysten todennäköisyys valitettavasti kasvaa ja jokerihakujen käyttö heikentää tilannetta entisestään. Lisäksi viruksen tarkemman tunnistamisen ja turvallisen poistamisen saavuttamiseksi täytyy viruksen salaus pystyä purkamaan. Merkkijonohakuja parempi ase itsekoodaavia ja polymorfisia viruksia vastaan ovat X-RAY-menetelmät. Lääketieteessä röntgeniä käytetään kudosten läpi näkemiseen, virustorjunnassa virusten salauksen läpi näkemiseen. Kaikki X-RAY-menetelmät perustuvat tunnetun selkotekstin pohjalta tehtyyn salauksen purkuun (known-plaintext attack). Tällainen hyökkäys salausta vastaan on mahdollista, koska viruksen salaamaton runko (selkoteksti) tunnetaan viruksen analysoinnin pohjalta.

X-RAY on menetelmänä jo vanha keksintö ja on ollut käytössä lähes yhtä kauan kuin itsekoodaavia viruksia on ollut olemassa. Koodin emulointi on nykyisin monissa tapauksissa X-RAYtä parempi vaihtoehto, ja tarjoaa mahdollisuuden löytää laajemman joukon eri virustyyppejä. Joissain tapauksissa X-RAY on kuitenkin edelleen koodin emulointia parempi vaihtoehto. Emulaattori voi käyttää yhden ohjelman emulointiin

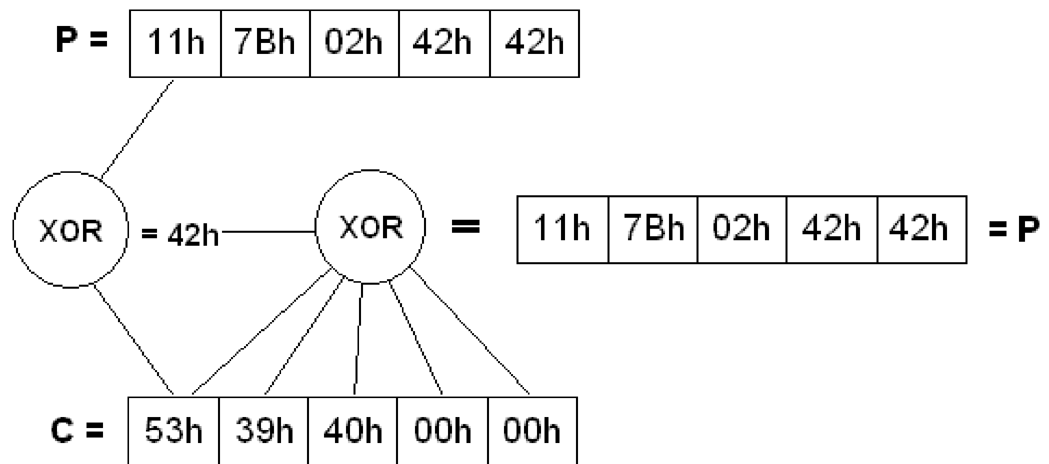
vain rajallisen määrän aikaa, mikä on useimmiten taustalla X-RAYn ollessa parempi vaihtoehto. EPO-virusten tapauksessa viruksen suoritus saattaa alkaa niin pitkän ajan kuluttua isäntäohjelman suorituksen alkamisesta, että kyseisen ohjelman emulointi lopetetaan ennen kuin se ehtii virukseen saakka. Joidenkin virusten purkuohjelmat suorittavat jopa miljoonia turhia käskyjä ennen kuin alkavat purkaa viruksen rungon salausta, näissäkin tapauksissa emulaattori saattaa lopettaa ohjelman emuloinnin ennen viruksen paljastumista. Sama kesto-ongelma emulaattoreilla on RDA-viruksissa, jotka murtavat oman salauksensa väsytyshyökkäyksellä. Jotkin polymorfiset virukset eivät pura itseään jokaisella isäntäohjelman suorituskerralla. Tällaisten virusten purkuohjelman alussa on ehtolause, joka tarkistaa jonkin usein vaihtelevan arvon, ja sitä kautta jatkaa viruksen suoritusta näennäisen satunnaisesti. Mikäli virus ei aina pura salaustaan, ei sitä voida täydellä varmuudella löytää koodia emuloimalla. Erikoistapaus ovat viruksen ohjelmointivirheiden seurauksena syntyvät vialliset purkuohjelmat – mikäli purkuohjelma ei toimi, ei emulointiakaan voida suorittaa viruksen rungon löytämiseen asti. X-RAY jättää purkuohjelmat huomiotta, eikä siksi hyydy viallisiin purkuohjelmiin. Itse asiassa purkuohjelmien ohittamisen seurauksena polymorfisuus itsessään ei vaikeuta X-RAY-menetelmien toimintaa mitenkään. X-RAY-menetelmien heikkoutena ovat useampaan kertaan salatut virukset. Sellaisten virusten löytäminen onnistuu X-RAY-menetelmillä vain joissain erikoistapauksissa – usean salauskerroksen läpi murtautuminen on lähes aina liian hidasta. (Ferrie ym. 2004)

Ferrie ym. jakavat X-RAY-menetelmät kolmeen eri luokkaan toimintaperiaatteensa mukaan (Ferrie ym. 2004 s. 53):

1. avaimen palautus (key recovery),
2. avaimen varmistus (key validation),
3. invariantti-muunnos ja saadun pienennöksen etsintä (invariant scanning).

Avaimen palautus on yksinkertaisesti potentiaalisen avaimen arvaus selkotekstin pohjalta ja soveltaminen salattuun tekstiin kunnes lopputuloksena on selkoteksti, jolloin arvattu avain on osoitettu oikeaksi. Avaimen palautus on kolmesta X-RAY-toimintaperiaatteesta tarkin, mutta samalla myös hitain. Mikäli koko viruksen salaus halutaan viruksen turvallisen poistamisen varmistamiseksi purkaa, on avaimen palautus

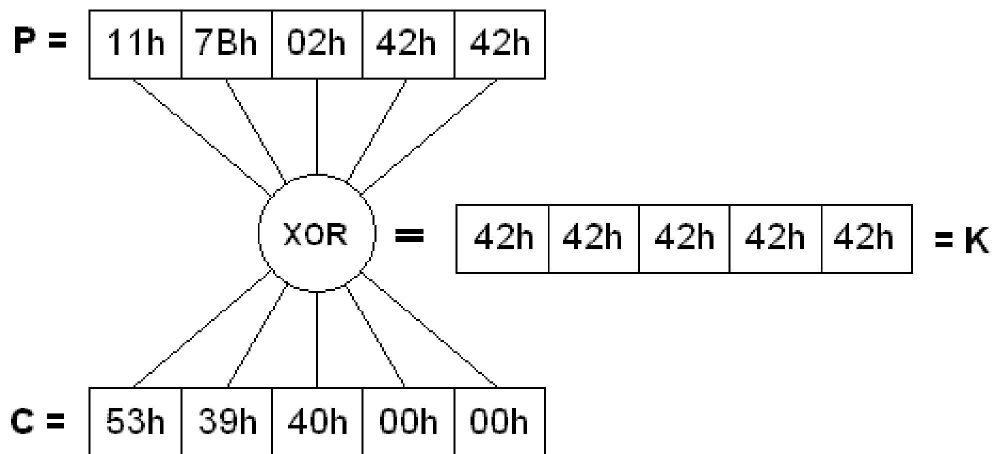
ainoa mahdollisuus. Avaimesta puhuttaessa on syytä huomioida ero perinteisen kryptografian ja X-RAY-menetelmien välillä. X-RAY:n näkökulmasta avaimessa on kaksi osaa: perinteisen avaimen (data-avaimen) lisäksi avaimessa on myös salauksessa käytetty operaatio (koodiavain, esim. XOR, ADD). Data-avainta käytetään koodiavaimen argumenttina. Sekä operaatio että varsinainen avain täytyy selvittää viruksen salauksen purkamiseksi. Otetaan esimerkkinä W95/Madin käyttämän salauksen murtaminen. Virus käyttää salauksessa aina samaa XOR-operaatiota eikä muuta käyttämäänsä lyhyttä avainta salauksen aikana. Lisäksi viruksen analysoinnin perusteella (esim. emulointia hyödyntämällä, jolloin virus purkaa itse salauksensa) myös salaamaton runko tunnetaan, joten avain on helppo löytää XOR-operaation symmetrisyyden vuoksi. Kuvassa 6 P on osa W95/Madin salaamatonta runkoa ja C sama kohta salattuna. XOR-operaation symmetrisyyden vuoksi avain saadaan salaamattomasta ja salatusta tavusta: 53h XOR 11h = 42h. Kun kaikille salatuille tavuille suoritetaan XOR-operaatio saadun avaimen kanssa, saadaan tuloksena sama merkkijono kuin ennalta tunnetussa salaamattomassa viruksen osassa, joten avain on oikea.



Kuva 6. Avaimen palautus W95/Madin kaltaisissa viruksissa.

Valitettavasti itsekoodaavat virukset eivät aina ole yhtä helppoja tapauksia kuin W95/Mad. On olemassa viruksia jotka vaihtelevat käyttämäänsä salaustapaa, käyttävät useampaa pitkää salausavainta, muuttavat avaimia salauksen aikana, salaavat itsensä useampaan kertaan tai muuten vaikeuttavat salauksensa purkua.

Avaimen varmistaminen on toinen salausavaimiin suoraan perustuva X-RAY-menetelmä. Avaimen varmistamisessa etsitään ensin useampi avain pohjaten etsintä eri osiin viruksen koodia. Avaimien löytämisen jälkeen niitä verrataan toisiinsa ja mikäli kaikki avaimet ovat samoja, on löydetty oikea avain viruksen salauksen purkuun. Löydetty virus on siis sama, jonka salaamatonta versiota käytettiin avaimen etsinnässä. Kuvassa 7 on jälleen esimerkkinä W95/Mad, jonka salaus puretaan avain varmistamalla. Virukselle etsitään viisi avainta XOR-operaatioilla, joiden argumentteina ovat yksi salaamaton tavu ja vastaavassa kohdassa oleva salattu tavu. Kaikki viisi tuloksena saatua avainta ovat samoja, joten virus varmistuu W95/Madiksi.



Kuva 7. Avaimen varmistus -menetelmän soveltaminen W95/Mad-virukseen.

Kolmas X-RAY-menetelmä on alun perin IBM:n kehittämä invariantti-muunnos (invariant transformation). Menetelmä koostuu viidestä eri vaiheesta (Arnold ym. 1995).

1. Hahmoille (virusten salaamattomille muodoille) tehdään invariantti-muunnos, jonka tuloksena saadaan hahmojen pienennökset (reduction).
2. Salatulle datalle (mahdollisesti viruksia sisältäville tiedostoille) tehdään sama muunnos, jonka seurauksena saadaan datan pienennös.
3. Hahmojen ja datan pienennöksiä verrataan merkkijonohaulla hahmojen löytämiseksi datasta. Mikäli hahmojen pienennöksiä löydetään, on hahmo todennäköisesti salatussa muodossa datan seassa.
4. Löydösten varmistetaan todella olevan etsittyjä hahmoja jollain erikoistuneemmalla menetelmällä. Tämä vaihe voidaan haluttaessa jättää väliin, mikäli väärät hälytykset ovat käytetyn salausmenetelmän vuoksi

epätodennäköisiä tai haluttu varmuustaso ei ole erityisen korkea. Avaimen palautus on varmistamiseen käypä menetelmä.

5. Täsmäävistä löydöistä ilmoitetaan, mitkä hahmot on löydetty ja hahmojen tarkat sijainnit tiedostoissa.

Invariantti-muunnos perustuu monilla yksinkertaisilla salausmenetelmillä olevaan ominaisuuteen: on olemassa operaatio, jolla kaikki samalla menetelmällä salatut merkkijonot käytetystä data-avaimesta huolimatta voidaan muuntaa yhdeksi ja samaksi merkkijonoksi. Aina samalla menetelmällä salatusta viruksesta muodostuu siis invariantti-muunnoksen jälkeen sama merkkijono, vaikka salauksessa käytettäisiin mitä tahansa data-avainta. Esimerkiksi yhden tavun mittaisella avaimella tavu kerrallaan suoritettavan XOR-salauksen invariantti-muunnos on merkkijonon XOR-operaatio yhden tavun oikealle siirretyn kopionsa kanssa (Arnold ym. 1995). Kuvassa 8 on W95/Madin salaamattomalle (P) ja salatulle versiolle (C) suoritettu invariantti-muunnos.

P =	11h	7Bh	02h	42h	42h	
XOR		11h	7Bh	02h	42h	42h
		6Ah	79h	40h	00h	

C =	53h	39h	40h	00h	00h	
XOR		53h	39h	40h	00h	00h
		6Ah	79h	40h	00h	

Kuva 8. Esimerkki XOR-operaation invariantti-muunnoksesta.

Invariantti-muunnosten tuloksena saadut pienennökset ovat samat, joten virus tunnistetaan W95/Madiksi, vaikkei salausta virusta etsittäessä pureta. Käytännön sovelluksessa salaamattomille viruksille invariantti-muunnos tehdään vain kerran, saadut hahmojen pienennykset tallennetaan virustutkan tietokantaan yhdessä kunkin viruksen käyttämien salausoperaatioiden kanssa. Tiedostoille suoritetaan kaikki virustutkan kattamat invariantti-muunnokset, ja jokaisen muunnoksen tuloksista etsitään kyseistä salausoperaatioita käyttävien virusten pienennöksiä.

X-RAY-menetelmät vaativat hitautensa vuoksi suodatuksen, joka poistaa mahdollisimman suuren osan tiedostoista epäiltyjen listalta ilman varsinaisten X-RAY-menetelmien käyttöä. Luvussa 4.3.2 mainittujen suodattimien lisäksi on tiedoston osien merkkijakauma salausta käyttäville viruksille ominainen suodatin. Salauksen

seurauksena eri merkkejä pitäisi itsekoodavan viruksen sisältävässä osassa olla melko tasaisesti, joten X-RAY-menetelmien käyttö voidaan rajata tiedoston niihin osiin, joissa merkkijakauma täyttää halutut ehdot. Samaten jos nollien osuus kaikista tavuista on tiedoston tietyssä osassa liian suuri, ei juuri siinä osassa ole virusta. (Ferrie ym. 2004 s. 54–55)

4.5 Koodin emulointi

Koodin emulointiin perustuvat menetelmät kehitettiin 90-luvun alussa reaktiona tehokkaisiin polymorfisiin viruksiin. Emuloinnissa tietokoneen keskusmuistiin luodaan virtuaalikone, joka on toiminnaltaan kopio fyysisestä tietokoneesta. Virtuaalikone saa syötteenä ohjelmatiedoston, jota se alkaa suorittaa. Virtuaalikone tarjoaa hallitun ympäristön, jossa viruksen sisältäviä ohjelmia voidaan suorittaa niiden aikaansaamatta vahinkoa. Ajatus emulaattorin käyttämisestä polymorfisten virusten löytämiseen perustuu kahteen polymorfisten virusten ominaisuuteen: niiden täytyy purkaa runkonsa salaus ennen kuin sen suoritus voidaan aloittaa ja viruksen salaamaton runko pysyy tartunnasta toiseen samana. Ajamalla virus virtuaalikoneen sisällä purkaa se itse salauksensa, mutta ei voi levitä tai aiheuttaa muuta vahinkoa. Virustutka tarkkailee virtuaalikoneen muistialuetta ja etsii sieltä polymorfisia tai itsekoodaavia viruksia niiden salaamattomasta rungosta muodostettujen normaalien allekirjoitusten perusteella. Emulaattorin avulla suoritettua itsensä salakirjoittavien virusten löytämistä kutsutaan geneeriseksi koodin avaamiseksi (generic decryption).

Emulaattorin osista keskeisin on prosessori-emulaattori. Sen täytyy luoda ja ylläpitää keskusmuistissa kaikkia emuloimansa aidon prosessorin sisältämiä rekistereitä ja lippuja (flag). Prosessori-emulaattori suorittaa virtuaalikoneen syötteenä saamaa ohjelmaa käsky kerrallaan tulkitsemalla ohjelman sisältämiä operaatiokoodeja kuten fyysinen prosessori. Jokaiselle operaatiokoodille on emulaattorissa oma rutiininsa, joka päivittää rekisterien ja lippujen virtuaalivastineita kunkin konekielisen käskyn vaatimalla tavalla. Esimerkiksi IP-rekisterin (Instruction Pointer) virtuaalivastineessa ylläpidetään seuraavaksi suoritettavan käskyn osoitetta.

Emuloitavien ohjelmien ei voida antaa suoraan käyttää keskusmuistia, vaan muisti-emulaattori hoitaa emuloitavan ohjelman muistioperaatiot. Muisti-emulaattorin on myös

syytä pitää kirjaa eri muistialueilla tapahtuvista aktiviteeteista, erityisesti siitä mille muistisivuille suoritetaan kirjoitusoperaatioita. Virusten täytyy salaustaan purkaessaan suorittaa runsaasti keskusmuistiin kirjoituksia, joten ne muistiosoitteet joihin on emuloinnin aikana kirjoitettu, ovat luonnollinen etsintäkohde.

Tietokoneen täydelliseksi emuloimiseksi myös käyttöjärjestelmän toiminnallisuus täytyy sisällyttää virtuaalikoneeseen. Windows-ympäristössä tämä tarkoittaa ainakin tärkeimpien käyttöjärjestelmän tarjoamien APIen toteutusta sekä tiedostojärjestelmän emulointia.

Geneerisessä koodin avaamisessa (generic decryption) on varsinaisen emuloinnin hoitavien komponenttien lisäksi oltava päätöksentekomekanismi (ECM, Emulator Control Module), joka ohjaa emulointia. Päätöksentekomekanismi ratkaisee, mistä kohdista tiedostoja emulointi aloitetaan ja päättää myös, milloin emulointi lopetetaan. Molemmissa tapauksissa taustalla on virustorjuntaohjelmien ainainen tasapainoilu nopeuden ja löytämisvarmuuden välillä. Virusten nopeaksi löytämiseksi on emulointi syytä pyrkiä aloittamaan läheltä viruksen purkuohjelmaa. Tätä varten ohjelmia emuloidaan niiden kaikista tunnetuista tulokohdista. Koodin emulointi on huomattavasti konekielisen ohjelman suoraa ajamista hitaampaa ja siksi jokaista ohjelmaa ei voida emuloida loputtomiin. Vaikein päätöksentekomekanismin tekemä ratkaisu on, milloin ohjelman emulointi voidaan lopettaa jättämättä merkittävää mahdollisuutta sille, että ohjelmassa on virus, joka ei ole ehtinyt vielä purkaa salaustaan. Yksinkertaisin ratkaisu on emuloida kaikkia ohjelmia tietty määrä käskyjä ja tarkistaa tasaisin väliajoin muisti-emulaattorin ohjelman suorituksen aikana muutetuiksi merkitsemät muistialueet virusten allekirjoitusten varalta. Tällainen kankea järjestelmä emuloi auttamatta joko viruksen sisältämättömiä ohjelmia liian kauan tai jättää merkittävän mahdollisuuden emuloinnin lopettamiselle liian aikaisin. Szor esittää kolme vaihtoehtoa emuloinnin paremmaksi kontrolloinniksi: aktiivisten käskyjen seuraaminen, purkuohjelmien seuraaminen profiilien avulla ja ennalta määrätyt katkokohdat (Szor 2005 s. 454).

Aktiivisilla käskyillä tarkoitetaan käskyjä, jotka muuttavat muistissa olevaa dataa kahdessa vierekkäisessä kohdassa. Tällainen vierekkäisiin keskusmuistin kohtiin

kirjoitus on tyypillistä monille purkuohjelmille. Emulaattori suorittaa ohjelmaa esimerkiksi 250000–1000000 käskyä, kunhan koodissa esiintyy aktiivisia käskyjä. Mikäli aktiivisia käskyjä ei esiinny, päättyy emulointi aikaisemmin.

Purkuohjelmat koostuvat yleensä vain pienestä määrästä eri käskyjä, joita purkuohjelma suorittaa silmukassa viruksen runkoa purkaessaan. Purkuohjelmille voidaan luoda profiilit näiden purkuohjelmissa käytettyjen käskyjen mukaan. Ensimmäisen profiiliin kuulumattoman käskyn tullessa suoritetuksi on viruksen rungon salaus oletettavasti purettu, joten emulointi voidaan lopettaa ja suorituksen aikana muutetut muistin osat tarkistaa viruksen rungon varalta.

Emulointi voidaan lopettaa myös asettamalla sille ennalta määrättyjä katkokohtia ehdoilla, jotka tavallisesti täyttyvät vasta purkuohjelmien suorituksen päätyttyä. Jokaisesta polymorfisesta viruksesta voidaan esimerkiksi valita yksittäinen käsky tai muutaman käskyn tiiviste (hash), jonka suoritusvuoroon tuleminen merkitsee viruksen rungon suorituksen alkamista. Muita mahdollisuuksia ovat ensimmäinen ohjelman aiheuttama keskeytys tai ohjelman suorittama API-kutsu, joita kumpiakaan polymorfiset virukset eivät yleensä käytä purkuohjelmissaan.

Emuloinnin jatkamisesta päättämiseen voidaan käyttää myös heuristisia menetelmiä, joita käsitellään seuraavassa luvussa.

4.6 Heuristiset menetelmät

Virustentorjunnasta puhuttaessa heuristiikalla tarkoitetaan joukkoa sääntöjä, joita ohjelmaan soveltamalla pyritään päättelemään, onko kyseisessä ohjelmassa virustartunta (Gryaznov 1995). Tällainen itse päätelmiä tekevä menetelmä herättää houkuttelevan ajatuksen ohjelmasta joka automaattisesti tunnistaa kaikki nykyiset ja tulevat virukset. Valitettavasti virukset muista ohjelmista 100-prosenttisen varmasti tunnistavan ohjelman tekeminen ei kuitenkaan ole mahdollista (Cohen 1984).

Viruksia etsivien heuristiikkojen säännöt perustuvat ohjelman ominaisuuksiin tai toimintoihin, joiden tiedetään olevan ominaisia viruksille, mutta harvinaisia muille ohjelmille. Sääntöihin liittyy aina tietty epävarmuustekijä, koska aivan täyttä varmuutta

ei koskaan ole siitä, ettei tavallinenkin ohjelma voisi sisältää ainakin jotakin virukseksi laskettua ominaisuutta. Onnistuneen heuristiikan kehittämiseksi täytyy löytää tasapaino väärien hälytysten ja löytämistarkkuuden välillä – heuristiikan täytyy löytää mahdollisimman suuri osan viruksista mahdollisimman vähillä väärillä hälytyksillä. Heuristiikka joka löytää kyllä kaikki virukset, mutta siinä sivussa merkitsee suurimman osan puhtaista ohjelmista virustartunnan saaneiksi, ei ole alkuunkaan hyödyllinen. Heurististen menetelmien etuna on niiden mahdollisuus havaita ennestään tuntemattomia viruksia ilman ohjelmiston tai tietokannan päivitystä. Muut menetelmät vaativat vähintään tietokantansa päivityksen löytääkseen uusia uhkia (poikkeuksena jokerihakujen mahdollisesti löytämät vanhojen tuttujen virusten uudet variantit).

Heuristiikat voidaan päätöksentekotapansa mukaan jakaa kahteen luokkaan: painoarvoja käyttäviin (weighted) ja sääntöjoukkopohjaisiin (rule based). Painoarvoja käyttävissä heuristiikoissa jokaiselle tiedostoista etsittävälle virukseen viittaavalle ominaisuudelle on oma painoarvonsa. Heuristisen analyysin edetessä kaikkien löydettyjen ominaisuuksien painoarvot lasketaan yhteen ja mikäli ennalta määrätty raja-arvo ylittyy, tulkitaan ohjelman sisältävän viruksen. Positiivisten painoarvojen lisäksi voidaan käyttää myös negatiivisia painoarvoja. Negatiiviset painoarvot annetaan ominaisuuksille, jotka viittaavat siihen ettei ohjelmassa ole virusta. Negatiivisia painoarvoja käyttämällä saadaan vääriä hälytyksiä vähennettyä ja ohjelman analysointi voidaan lopettaa aikaisemmin, jos näyttää vahvasti siltä ettei virustartuntaa ole.

Esimerkkinä painoarvoja käyttävästä heuristiikasta on seuraavassa geneeriseen koodin avaamiseen liittyvä esimerkki (Nachenberg 1996 s. 9). Geneerisessä koodin avaamisessa vaikein ratkaisu on se kuinka kauan ohjelman emulointia on syytä jatkaa. Lopettamispäätös voidaan tehdä heuristisesti antamalla purkuohjelmille tyypillisille käskyille positiivinen painoarvo ja tavallisiin ohjelmiin viittaaville tapahtumille negatiivinen painoarvo.

- Jokainen NOP-käsky kasvattaa todennäköisyyttä 0.5 prosenttiyksikköä.
- Jonkin rekisterin sisällön tuhoaminen ennen sen hyödyntämistä kasvattaa viruksen todennäköisyyttä 1.2 prosenttiyksikköä. Molemmat todennäköisyyttä

lisäävät tapaukset ovat yksinkertaisten polymorfisten virusten purkuohjelmille tyypillisiä roskakäskyjä.

- Jos ohjelma aiheuttaa keskeytyksen, vähennetään todennäköisyyttä 15 prosenttiyksikköä.
- Jos ohjelma ei kirjoita muistiin kertaakaan sadan perättäisen käskyn aikana, vähennetään viruksen todennäköisyyttä 5 prosenttiyksikköä.

Jokaisella ohjelmalla katsotaan suorituksen alkaessa olevan 10 prosentin todennäköisyys sisältää polymorfinen virus. Emulointia jatketaan kunnes viruksen todennäköisyys laskee nollaan. Käytännön heuristiikassa sääntöjä täytyy olla enemmän, vain kahta positiivista sääntöä käyttävä heuristiikka ei huomaisi tarpeeksi suurta joukkoa polymorfisista viruksista ja lopettaisi emuloinnin ennen aikojaan.

Painoarvojen sijaan voidaan heuristiikka perustaa eri sääntöjoukkojen etsimiseen ohjelmista. Eri viruksille ja yleisemmin virustyypeille kehitetään oma joukkonsa sääntöjä, jotka määrittelevät juuri kyseisen uhan. Symantecin Striker-järjestelmä käyttää geneerisen koodin avaamisen ohjaamiseen sääntöjoukkoihin perustuvaa heuristiikkaa. Jokaiselle polymorfiselle virukselle ja mutaatiokoneelle on oma profiilinsa, joka sisältää kaikki kyseiseen uhkaan viittaavat säännöt. Sääntöinä ovat esimerkiksi matemaattisten laskutoimitusten suorittaminen ja saatujen tulosten hyödyntämättä jättäminen, tiettyjen satunnaisten roskakäskyjen käyttäminen tai käyttöjärjestelmän kutsuminen salauksen purun edetessä. Kaikkiaan erilaisia sääntöjä on yli 500. Otetaan kuvitteellisena esimerkkinä säännöt A, B, C, D ja E, jotka muodostavat päätöksentekomekanismin heuristiikan. Virus 1:n purkuohjelma täyttää säännöt A, B ja C, virus 2 säännöt A, B ja D sekä virus 3 säännöt B, D ja E. Jos koodia emuloitaessa heuristiikka huomaa säännön A täyttyvän, voi se sulkea viruksen 3 pois epäiltyjen listalta. Oletetaan seuraavaksi säännön C toteutuvan, nyt myös virus 2 on suljettavissa mahdollisuuksien ulkopuolelle. Striker-järjestelmän kaltaista ratkaisua käytettäessä emulointi on nopeampaa, koska tarvitsee etsiä vain niitä sääntöjä, jotka ovat jossakin jäljellä olevien virusten profiileissa. Päätöksentekojärjestelmä lopettaa ohjelman emuloinnin, kun löydetyt säännöt eivät ole yhdenkään viruksen profiilin osajoukko, eikä ohjelmassa näin ollen voi olla virusta. (Nachenberg 1996)

Heuristiikat voidaan jakaa dynaamisiin ja staattisiin sen mukaan miten ne ohjelmia analysoivat. Dynaamiset heuristiikat perustuvat ohjelmien emulointiin ja niiden ajonaikaisen käytöksen analysointiin. Staattiset heuristiikat analysoivat suoraan tiedostojen ominaisuuksia ja sisältöä ohjelmaa emuloimatta. Staattiset heuristiikat etsivät ohjelmatiedostoista viruksille tyypillisien toimintojen käskyjonoja, kuten tiedoston avaamista kirjoitusta varten tai kiintolevyn alustamista. Esimerkkinä seuraavat COM-tiedostoihin loppuun itsensä lisäävän loisviruksen löytävät karkeat säännöt (Gryaznov 1995).

1. Ohjelma siirtää kontrollin itsensä loppuosaan heti suorituksen alkaessa.
2. Ohjelma muuttaa keskusmuistissa muutaman ensimmäisen tavunsa arvoja.
3. Ohjelma etsii muita ohjelmatiedostoja.
4. Ohjelma avaa löytämänsä ohjelmatiedoston.
5. Ohjelma lukee avaamaansa tiedostoa.
6. Ohjelma kirjoittaa avaamansa tiedoston loppuun.

Ohjelmalistauksessa 6 on edelliset toiminnot sisältävä ohjelma symbolisena konekielenä.

Ohjelmalistaus 6. COM-tiedoston loppuun itsensä lisäävä ohjelma ja siihen liittyvät heuristiset säännöt (Gryaznov 1995).

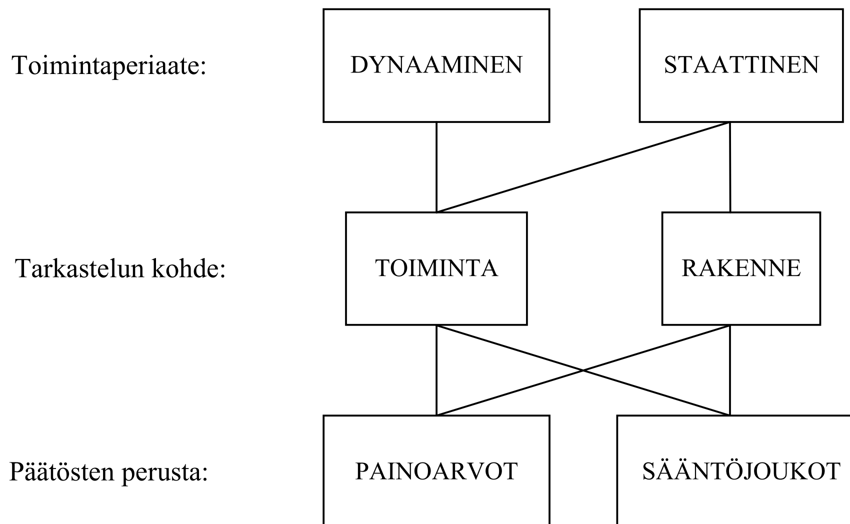
```
START:              ; Start of the infected program
     JMP  VIRUSCODE ; Rule 1: the control is passed
                    ;        to the virus body
                    ;

   <victim's code>

VIRUS:              ; Virus body starts here

SAVED:              ; Saved original bytes of the victim's code

MASK: DB `*.COM',0  ; Search mask

VIRUSCODE:          ; Start of the virus code
      MOV  DI,OFFSET START  ; Rule 2: the virus restores
      MOV  SI,OFFSET SAVED  ; victim's code
      MOVSW                 ; in memory
      MOVSB                 ;

      MOV  DX,OFFSET MASK   ; Rule 3: the virus
      MOV  AH,4EH           ; looks for other
```

```
INT  21H              ; programs to infect

MOV  AX,3D02H         ; Rule 4: the virus opens a file
INT  21H              ;

MOV  DX,OFFSET SAVED  ; Rule 5: first bytes of a file
MOV  AH,3FH           ; are read to the virus
INT  21H              ; body

MOV  DX,OFFSET VIRUS  ; Rule 6: the virus writes itself
MOV  AH,40H           ; to the file
INT  21H              ;
```

Toimintojen löytäminen tiedostoista ei ole aivan yksinkertaista, koska samat toiminnot voidaan aikaansaada erilaisilla käskysarjoilla. Tehokkaan toimintaa tarkastelevan staattisen heuristiikan tulisi pystyä tunnistamaan epäilyttävien käskyjen ja käskysarjojen kaikki erilaiset muodot. Dynaaminen heuristiikka on siinä mielessä parempi, että sille riittää huomata, että epäilyttävä toiminto suoritetaan, eikä tarkalla suoritustavalla silloin ole merkitystä. Dynaamisten heuristiikkojen ongelma on hitaus, kuten yleensä koodin emulointiin perustuvilla menetelmillä. Staattiset heuristiikatkaan eivät ole immuuneja hitaus-ongelmille, vaan niiden täytyy keskittyä analysoimaan viruksille tyypillisiä sijainteja tiedostojen sisällä.

Monimutkaisemmat, mutta tarkemmin rakenteeltaan määritellyt tiedostomuodot ovat mahdollistaneet uudentyyppisen staattisen heuristiikan. Ohjelman käyttäytymisen sijaan staattinen heuristiikka voi tarkastella ohjelman rakennetta, etsien ohjelmatiedostoista sellaisia rakenteellisia epäsäännöllisyyksiä, joiden aiheuttajana voi todennäköisesti olla vain virus.

Tässä luvussa esitetyt heuristiikat voidaan karkeasti jakaa niiden toimintaperiaatteen, tarkastelun kohteen ja päätöksenteon perustan mukaan. Kuvassa 9 on esitetty yhteenvetona mahdolliset yhdistelmät. Dynaaminen heuristiikka perustuu aina ohjelman toiminnan analysointiin, kun taas staattinen heuristiikka voi tutkia myös ohjelmatiedoston rakennetta. Päätöksenteon perustaksi voi aina ottaa joko painoarvot tai sääntöjoukot.



Kuva 9. Heurististen menetelmien jaottelu niiden ominaisuuksien mukaan.

4.7 Yhteenveto tiedostovirusten etsimisestä

Ensimmäinen virusten löytämiseksi kehitetty menetelmä eli merkkijonohaku, on edelleen käyttökelpoinen yksinkertaisten virusten löytämiseksi. Sen yhtenä etuna on virusten tarkka tunnistaminen. Edistyneempiä tartuntamenetelmiä (itsekoodausta tai polymorfismia) hyödyntävien virusten luotettava löytäminen vaatii muita menetelmiä. Viruksen ominaisuuksien mukaan määräytyen vaihtoehtoina on silloin joko jokin X-RAY-menetelmistä tai geneerinen koodin avaus. Koodin emulointi on yleisempi menetelmä ja tarjoaa yksinkertaisten virusten löytämisen lisäksi mahdollisuuksia myös monien vaikeidenkin virusten löytämiseen. Emulaattori onkin hitaudestaan huolimatta tärkeä osa nykyaikaista virustentorjuntaohjelmaa. Metamorfisten ja vaikeiden EPO-virusten löytäminen saattaa vaatia varta vasten kehitettyä algoritmista metodia, siksi virustentorjuntaohjelman olisi hyvä tukea uusien algoritmisten menetelmien lisäystä itsensä osaksi. Heuristiset menetelmät tarjoavat mahdollisuuden löytää ennalta määriteltyjen virusten sijaan yleisesti virukset niille tyypillisten ominaisuuksien perusteella. Heuristiikat voidaan yhdistää myös koodin emulointiin. Heuristiikkojen heikkoutena ovat niiden tyypillisesti muita menetelmiä runsaslukuisemmat väärät hälytykset.

Kuten esitettyjen menetelmien määrästä voi päätellä, ei virusten löytämiseen ole olemassa yhtä ylivertaista menetelmää, vaan kaikkien virusten löytämiseksi on

hyödynnettävä kuhunkin tapaukseen parhaiten soveltuvaa metodia ja tasapainoiltava samalla löytämistarkkuuden ja nopeuden välillä. Monet menetelmistä vaativatkin hitautensa vuoksi läpikäytävien tiedostojen määrän vähentämistä suodattamalla.

5 STAATTISET RAKENTEELLISET HEURISTIIKAT WIN32-TIEDOSTOVIRUSTEN ETSIMISEEN

5.1 Kehitettävien heuristiikkojen määrittely ja tavoitteet

PE-tiedostoista viruksia voidaan etsiä pyrkimällä tunnistamaan tiedostoista epäilyttäviä rakenteita. Mahdollisuuden tähän tarjoaa otsakkeiden ja ohjelman osioihin jaon tuoma säännönmukaisuus, jossa tapahtuvista poikkeamista voidaan tunnistaa virusten todennäköisesti aiheuttamat epäsäännöllisyydet. Tiedostoista ei siis etsitä yksittäisten virusten tunnisteina käytettäviä merkkijonoja, mikä olisi virustorjuntaohjelmien perinteisin tapa toimia, tai viruksiin viittaavia käskyjonoja, mikä olisi perinteisen toimintaa tarkastelevan staattisen heuristiikan metodi.

Tässä tutkielmassa kehitetään kaksi erillistä staattista rakenteellista heuristiikkaa. Päätökset heuristiikat tekevät sääntöjoukkoihin eivätkä painoarvoihin perustuen. Ensimmäinen heuristiikka pyrkii löytämään EPO-menetelmiä käyttämättömiä Win32-tiedostoviruksia ja toinen heuristiikka Win32-tiedostoviruksia, jotka käyttävät EPO-menetelmiä. Molempien heuristiikkojen tavoitteena voidaan luonnollisesti pitää mahdollisimman suuren osan viruksista löytämistä mahdollisimman pienellä määrällä vääriä hälytyksiä, tosin heuristiset menetelmät lähes väistämättä päättelyyn perustuvan luonteensa vuoksi aiheuttavat niitä. Toisena tavoitteena heuristiikoille on nopeus. Suurimpana pullonkaulana tiedostoviruksia etsittäessä ovat levyoperaatiot, joten nopeuden parantamiseksi etsintöjä ei suoriteta koko tiedostosta, vaan ainoastaan otsakkeista ja tuontitaulusta sekä isäntäohjelman tulokohdasta. Läpikäytävän datan määrä saadaan näin useimmissa tapauksissa rajattua alle kilotavuun per tiedosto. Ratkaisevimpana tekijänä tiedostosta luettavan datan määrässä on se, täytyykö tiedoston tarkistussumma laskea, jolloin koko tiedosto käydään läpi alusta loppuun. Tarkistussumma on kuitenkin tavallisissa ohjelmatiedostoissa vain harvoin käytössä. Nopeuden tarkoituksena on mahdollistaa heuristiikkojen käyttö suodattimina muille tarkemmille, mutta huomattavasti hitaammille menetelmille. Heuristiikat etsivät viruksia ainoastaan PE-muotoisista varsinaisista EXE-ohjelmatiedostoista, eivät DLL-, ajuri- tai objektitiedostoista tai muista PE-muodossa olevista tiedostoista. Heuristiikkojen kohdelaitearkkitehtuuri on nykyisissä henkilökohtaisissa tietokoneissa

yleisimmin käytössä oleva IA-32. Heuristiikat rajataan 32-bittisiin ohjelmatiedostoihin, 64-bittisiä ohjelmatiedostoja ei käsitellä.

Vain pieni osa yksittäisistä tarkistuksista löytää viruksia aiheuttamatta suuria määriä vääriä positiivisia, joten heuristiikat perustuvat yksittäisistä testeistä muodostettuihin joukkoihin. Sääntöjoukot muodostettiin ajamalla yksittäiset testit 16:lla eri loisviruksella tartutetuille ohjelmatiedostoille. Sääntöjoukkojen pohjana toimineet virukset on lueteltu liitteessä 1. Virukset pyrittiin valitsemaan niin, että ne edustaisivat monipuolisesti erilaisia tartuntatekniikoita, jotta alkuperäisistä sääntöjoukoista tulisi mahdollisimman kattavia pienestä virusjoukosta huolimatta. Alkuperäisiä sääntöjoukkoja täydennettiin testaamalla heuristiikkaa F-Securen tiloissa Helsingissä suurella virusjoukolla.

5.2 Heuristiikka EPO-menetelmiä käyttämättömille viruksille

5.2.1 Epäilyttävien tulokohtien etsiminen

Hyvä tapa löytää tavalliset loisvirukset on tarkastella ohjelman tulokohtaa. EPO-menetelmiä käyttämättömät virukset muuttavat joko isäntäohjelman tulokohtaa tai alkuperäisessä tulokohdassa olevaa koodia. Epäilyttävä tulokohta on selkeä merkki mahdollisesta virustartunnasta.

MS-DOS-otsakkeen e_lfanew-kentän arvoa muuttavat virukset on yksinkertaista havaita varmistamalla kentän arvon olevan normaalin. Tiedoston tavussa 3Ch täytyy olla suhteellisen pieni arvo, jotta voidaan varmistua siitä, ettei tiedostoon ole tarttunut uuden PE-otsakkeen lisäävää virusta. PE-otsakkeen sijaitessa ohjelman jälkimmäisessä puoliskossa on virustartunta todennäköinen (Szor 2005 s. 469).

Tavallisesti loisvirukset lisäävät isäntäohjelmaan kokonaan uuden osion tai liittävät itsensä viimeisen jo olemassa olevan osion jatkeeksi. Virukset siirtävät suorituksen itseensä muuttamalla valinnaisen otsakkeen AddressOfEntryPoint-kentän arvon osoittamaan itseensä. Kääntäjät tekevät tavallisesti vain yhden ohjelmakoodia sisältävän osion, jonka nimi on usein .text. Tulokohdan sijaitseminen .text-osion ulkopuolella on hyvin epäilyttävää (Szor 2005 s. 468). Tulokohdan osuminen .text-osion sisälle voidaan

tarkistaa vertaamalla AddressOfEntryPoint-kentän arvoa .text-osion alku- ja päätepisteisiin, jotka saadaan osiotaulusta. AddressOfEntryPoint-kentän arvo on RVA-muodossa, joten osiotaulusta käytetään vastaavassa muodossa olevia VirtualAddress- ja VirtualSize-kenttiä, eikä osion sijaintia tiedostossa ilmaisevia SizeOfRawData- ja PointerToRawData-kenttiä. Tulokohta sijaitsee koodi-osiossa, mikäli seuraavat kaksi ehtoa toteutuvat:

- AddressOfEntryPoint >= VirtualAddress,
- AddressOfEntryPoint < VirtualAddress + VirtualSize - 1.

Vaihtoehtoisesti voidaan tulokohdan .text-osiossa olemisen sijaan tarkistaa onko tulokohta ohjelman viimeisessä osiossa, mikä on sekin virukseen viittaava ominaisuus. Tulokohdan etsiminen viimeisestä osiosta on .text-osiosta etsimistä toimivampi ratkaisu, koska koodi-osion nimi saattaa .text:in sijaan olla mikä hyvänsä, eikä kääntäjän luoman koodi-osion luotettava tunnistaminen osion nimen perusteella siksi ole mahdollista. Lisäksi, vaikka useamman kuin yhden koodi-osion sisältävät ohjelmat ovat selkeä vähemmistö, eivät ne kuitenkaan ole aivan tavattoman harvinaisia. Tällaisten useamman koodi-osion sisältävien ohjelmien tapauksessa tulokohta voi sijaita .text-osion ulkopuolella, mutta silti laillisessa koodi-osiossa. Näiden seikkojen vuoksi heuristiikka tarkistaa tulokohdan mahdollisen sijainnin viimeisessä osiossa, eikä tulokohdan sijaintia ennalta määrättyjen nimisten osioiden ulkopuolella.

Tulokohdan sijaintia analysoimalla voidaan löytää myös otsakevirukset. Osiotaulun ja ensimmäisen osion väliin itsensä lisäävät otsakevirukset paljastuvat tarkistamalla sijaitseeko tulokohta ennen ensimmäistä osiota. Tarkistus voidaan suorittaa vertaamalla AddressOfEntryPoint-kentän arvoa ensimmäisen osion PointerToRawData-kenttään tai valinnaisen otsakkeen SizeOfHeaders-kenttään. SizeOfHeaders kertoo kaikkien otsakkeiden (mukaan lukien osiotaulun) yhteiskoon – AddressOfEntryPoint-kentän arvon ollessa otsakkeiden kokoa tai ensimmäisen osion alkuosoitetta pienempi, on tiedostossa todennäköisesti otsakevirus. PointerToRawData-kentän arvo ei ole RVA-muodossa toisin kuin tulokohdan kertova kenttä, mutta tässä tapauksessa sillä ei ole merkitystä. Tiedoston otsakkeet nimittäin ladataan levyltä keskusmuistiin sellaisenaan, jolloin RVA-muodossa oleva tulokohta ja osoitteen suhteessa tiedoston alkuun kertova PointerToRawData ovat verrattavissa toisiinsa.

Hieman vaikeammin itsensä havaittaviksi tekevät virukset eivät muuta isäntäohjelman tulokohdan määräävää kenttää osoittamaan itseensä, vaan muuttavat alkuperäisessä tulokohdassa olevaa koodia niin, että suoritus siirtyy virukseen (obfuscated tricky jump, esimerkiksi W32/Cabanas). Kuvassa 10 ovat nykyisten PC-tietokoneiden käyttämän IA-32-arkkitehtuurin kaikki mahdolliset hyppykäskyt, joilla virus voi siirtää kontrollin itselleen. Erityisen epäilyttäviä ovat tapaukset, joissa hyppy siirtää ohjelman suorituksen johonkin toiseen osioon. Ohjelma, jonka tulokohdassa on jokin kuvan 10 operaatiokoodeista, on mahdollisesti saanut virustartunnan. Lisäksi havaittiin Win32/Oporto.3076:n siirtävän suorituksen itseensä poikkeuksellisesti operaatiokoodeilla 68 XX XX XX XX C3 (PUSH muistiosoite RET). Kyseistä operaatiokoodiparia käytetään normaalisti ennen aliohjelmakutsua tallentamaan paluuosoite pinoon, ja aliohjelman lopuksi siirtämään suoritus aliohjelmasta takaisin pinoon tallennettuun muistiosoitteeseen.

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
EB *cb*	JMP *rel8*	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits
E9 *cw*	JMP *rel16*	N.S.	Valid	Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.
E9 *cd*	JMP *rel32*	Valid	Valid	Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits
FF /4	JMP *r/m16*	N.S.	Valid	Jump near, absolute indirect, address = sign-extended *r/m16*. Not supported in 64-bit mode.
FF /4	JMP *r/m32*	N.S.	Valid	Jump near, absolute indirect, address = sign-extended *r/m32*. Not supported in 64-bit mode.
FF /4	JMP *r/m64*	Valid	N.E.	Jump near, absolute indirect, RIP = 64-Bit offset from register or memory
EA *cd*	JMP *ptr16:16*	Inv.	Valid	Jump far, absolute, address given in operand
EA *cp*	JMP *ptr16:32*	Inv.	Valid	Jump far, absolute, address given in operand
FF /5	JMP *m16:16*	Valid	Valid	Jump far, absolute indirect, address given in *m16:16*
FF /5	JMP *m16:32*	Valid	Valid	Jump far, absolute indirect, address given in *m16:32*.
REX.W + FF /5	JMP *m16:64*	Valid	N.E.	Jump far, absolute indirect, address given in *m16:64*.

Kuva 10. IA-32-arkkitehtuurin hyppykäskyjen operaatiokoodit (Intel 2006 s. 497).

5.2.2 Epäilyttävät osioiden ominaisuudet

Osiotaulussa jokaisen osion otsakkeessa määritellään kyseisen osion sisältö suoritettavaa, kirjoitettavaa, luettavaa tai jokin niiden yhdistelmä. Normaalisti ohjelman koodi-osio on luettavissa ja suoritettavissa. Data-osiot puolestaan on jaettu luettaviin ja kirjoitettaviin sekä ainoastaan luettavissa oleviin. Osion ominaisuudet sisältävä Characteristics-kenttä on osion otsakkeen viimeisenä kenttänä. Kenttä on neljän tavun mittainen, ja kentän toiseksi merkitsevin heksamuotoinen numero määrittelee edellä mainitut ominaisuudet. Osio on suoritettavissa, mikäli numero on 2h, luettavissa numeron ollessa 4h ja kirjoitettavissa numeron ollessa 8h. Useampia käyttömahdollisuuksia osiolle voidaan antaa laskemalla summia, koodi-osiolle tyypilliset luku- ja suoritusoikeudet annetaan siis numerolla 6h (2h+4h). Sekä luku- että kirjoitusoikeudet annetaan numerolla Ch (4h+8h). Viruksille tyypillisiä osioita ovat vain kirjoitettavissa oleva osio tai sekä kirjoitettavissa että suoritettavissa oleva osio (Szor 2005 s. 468). Sekä kirjoitettavissa että suoritettavissa olevat osiot viittaavat vahvasti virukseen, koska tavallisissa ohjelmissa data ja koodi ovat erotettuna omiin osioihinsa.

Epäilyttävien käyttöoikeusyhdistelmien lisäksi voitaisiin tarkistaa, ettei tiettyihin osioihin ole niille normaalisti kuulumattomia oikeuksia. Yleisesti tiedoston viimeisenä oleviin osioihin, kuten .debug-, .reloc- tai .rsrc-osioihin, on vain lukuoikeus. Mikäli kyseisiin osioihin on joitakin muitakin oikeuksia on jokin loisvirus mahdollisesti lisännyt itsensä osion jatkeeksi.

Väärien positiivisten vähentämiseksi osion ominaisuuksien tarkastelu rajataan kehitettävässä heuristiikassa tulokohdan sisältävään osioon: kirjoitettavissa olevaa tulokohdan sisältävää osiota voidaan pitää erityisen epäilyttävänä.

Testauksessa havaittiin myös, että tulokohdan sijaintiin liittyvän testin aiheuttamia vääriä positiivisia voidaan joidenkin sääntöjoukkojen tapauksissa vähentää tarkistamalla osiotaulusta sisältääkö tulokohdan sisältävä osio koodia (osion sisällön tyypin kertoo Characteristics-kentän toiseksi vähiten merkitsevä numero). Heuristiikka hyödyntää siis myös sääntöä, joka määrittelee epäilyttäväksi tiedoston, jossa tulokohdan sisältävä osio sisältää koodia, mutta on kirjoitettavissa.

5.2.3 Otsakkeissa olevien kokoon liittyvien kenttien tarkistaminen

Valinnaisessa otsakkeessa on kentät ohjelman koolle keskusmuistissa (SizeOfImage) ja tiedostossa olevien koodiosioiden yhteiskoolle (SizeOfCode). Uuden osion ohjelmaan lisäävien tai itsensä johonkin valmiiseen osioon lisäävien virusten löytämistä voidaan edesauttaa vertaamalla kenttien arvoja tiedoston todellisiin arvoihin, koska kaikki virukset eivät päivitä otsakkeiden kenttiä tai laskevat niiden arvot väärin. Koodiosioiden todellinen yhteiskoko saadaan osiotaulun kaikkien suorituskelpoisiksi merkittyjen osioiden SizeOfRawData-kenttien summasta. Osio on suoritettavissa, jos sen otsakkeen Characteristics-kentän toiseksi merkitsevin numero on 2h, 6h, Ah tai Eh. Ohjelmatiedostoja tutkittaessa huomattiin, että SizeOfCode-kentän arvo ei viruksia sisältämättömissäkään ohjelmissa aina ole oikea. Joissain tapauksissa syynä tähän on se, että kentän arvo on virheellisesti laskettu osioiden vaatimasta tilasta keskusmuistissa, eikä niiden levyltä viemästä tilasta. Tämän vuoksi heuristiikka tarkistaa SizeOfCode-kentän arvon myös verrattuna koodia sisältävien osioiden VirtualSize-kenttien arvojen summaan.

Loisvirustartunnan seurauksena ohjelman koodia sisältävien osioiden yhteiskoko kasvaa, joten yksinkertaisimmillaan epäilyttävänä voidaan pitää ainoastaan tapauksia, joissa koodia sisältävien osioiden yhteiskoko on otsakkeessa ilmoitettua suurempi.

SizeOfImage-kentän arvon muodostumisesta ei ollut saatavilla tarpeeksi yksityiskohtaista tietoa, joten tarkka metodi varmistettiin PE-tiedostoja tutkimalla. SizeOfImage-kentän arvo koostuu tiedoston kaikkien osioiden sekä otsakkeiden yhteiskoosta ja määrittelee ohjelman keskusmuistissa viemän tilan. Otsakkeiden koko saadaan valinnaisen otsakkeen SizeOfHeaders-kentästä ja osioiden yhteiskoon laskemiseen käytetään niiden VirtualSize-kenttiä, jotka kertovat kunkin osion keskusmuistissa vaatiman tilan. Osiot alkavat keskusmuistissa SectionAlignment-kentän arvolla jaollisista osoitteista, joten kokoja laskettaessa VirtualSize- ja SizeOfHeaders-kenttien arvot täytyy pyöristää ylöspäin lähimpään SectionAlignment-kentän arvolla jaolliseen lukuun. Taulukossa 2 on esimerkkinä Windows XP:n notepad-ohjelman SizeOfImage-kentän arvon määräytyminen. Kolmesta osiosta ja otsakkeista koostuva ohjelma vaatii keskusmuistista yhteensä 14000h tavua tilaa.

Taulukko 2. Esimerkki SizeOfImage-kentän arvon muodostumisesta SectionAlignment-kentän arvon ollessa 1000h.

	VirtualSize	koko SectionAlignment huomioitaessa
.text	7748h	8000h
.data	1BA8h	2000h
.rsrc	8958h	9000h
SizeOfHeaders	400h	1000h
SizeOfImage		14000h

Windows NT -pohjaisissa käyttöjärjestelmissä (2000, XP jne.) lataaja tarkistaa SizeOfImage-arvon oikeellisuuden eikä suorita ohjelmaa, jos arvo ei ole yhteneväinen ohjelman todellisen koon kanssa (Szor 2005 s. 165). Tämän perusteella SizeOfImage-kenttään pohjautuvaa heuristiikkaa voitaisiin siis käyttää vain vanhojen Windows 95/98/Me -käyttöjärjestelmiin tehtyjen virusten löytämiseen. Vaikka lataaja tarkistaa kentän arvon, kuten Szor toteaa, ei tarkistus kuitenkaan ole aivan aukoton. Heuristiikkaa toteutettaessa havaittiin, että väärän SizeOfImage-arvon omaavat ohjelmat latautuivat tietyissä tapauksissa myös uusia Windows-versioita käytettäessä. Esimerkiksi Notepad-ohjelma latautui SizeOfImage-arvoilla 13001h–14000h. Mahdollisesti SizeOfImage-kentän arvo voi olla oikeaa arvoa pienempi määrällä SectionAlignment-1. SizeOfImage-kenttään perustuva heuristiikka ei siis ole täysin hyödytön nykyisten Windows-virusten löytämisessä, vaan saattaa olla apuna pienten loisvirusten löytämisessä. Joka tapauksessa väärän SizeOfImage-arvon mahdollisuus on syytä huomioida väärien hälytyksien vähentämiseksi. Syystä tai toisesta eräissä virusta sisältämättömissä ohjelmissa ei SizeOfImage-arvoa muodostettaessa ole huomioitu viimeisen osion koon pyöristämistä ylöspäin SectionAlignment-kentän arvolla jaolliseen lukuun. Tämän vuoksi heuristiikka katsoo oikeiksi myös kyseiset SizeOfImage-arvot.

SizeOfCode-kentälle eivät uudetkaan Windowsin versiot suorita mitään tarkastuksia, joten sen arvoa voidaan käyttää laajemmin myös nykyisten virusten löytämiseen.

5.2.4 Käyttöjärjestelmän muistialueelle ladattavat osiot

Windows 9x/Me -käyttöjärjestelmissä käyttöjärjestelmälle itselleen on varattu muistiosoitteet C0000000h–FFFFFFFFh. Kyseistä muistialuetta ei ole suojattu lukemiselta tai kirjoittamiselta, vaan mikä tahansa prosessi voi käsitellä järjestelmän muistialuetta haluamallaan tavalla. Jonkinlaisena suojana muistialueeseen vahingossa tapahtuvia kirjoituksia vastaan alkaa sen ensimmäinen komponentti VMM (Virtual Machine Manager), joka hoitaa esimerkiksi muistin- ja prosessien hallinnan, vasta muistiosoitteesta C0001000h (Szor 1998a s. 82). Näin ollen aivan muistialueen alkuun tapahtuvat kirjoitukset eivät vaikuta järjestelmän toimintaan ja siten aiheuta virheellistä toimintaa. VMM:ää edeltävät vapaat 4096 tavua ovat riittävästi virukselle, jonka kannalta järjestelmän alueella sijaitseminen onkin houkutteleva vaihtoehto. Käyttöjärjestelmälle kuuluvalla muistialueella toimiessaan virus saa automaattisesti samat käyttöoikeudet kuin käyttöjärjestelmä, ja saa siten mahdollisimman vapaat kädet toiminnalleen. Virus saadaan sijoitettua käyttöjärjestelmän alueelle lisäämällä se ohjelmaan omana osionaan, jonka aloitusosoitteeksi (VirtualAddress) osiotaulussa annetaan C0000000h. Ohjelmat joiden jokin osio alkaa kyseisestä kohdasta sisältävät todennäköisesti viruksen (Szor 2005 s. 471).

5.2.5 Viruksiin viittaava APIen käyttö

Win32-virukset tarvitsevat toimintaansa runsaasti käyttöjärjestelmän APIensa kautta käytettäviksi tarjoamia palveluita. Tiedostovirukset tarvitsevat tyypillisesti joukkoa tiedostojen käsittelyyn käytettäviä rutiineja. Isäntäohjelman alkuun itsensä lisääville loisviruksille tyypillinen joukko kernel32.dll-tiedostossa sijaitsevia rutiineja on Szorin mukaan GetModuleHandle, Sleep, FindFirstFile, FindNextFile, MoveFile, GetWindowsDirectory, WinExec, DeleteFile, WriteFile, CreateFile, MoveFile ja CreateProcess (Szor 2005 s. 470). Szorin esittämää rutiinijoukkoa muokattiin heuristiikan pohjana olevan virusjoukon perusteella. Rutiinijoukko muodostettiin heuristiikan pohjana olevan virusjoukon sisältämistä neljästä isäntäohjelman alkuun itsensä lisäävästä loisviruksesta (Win32/HLLP.Sypon.a, Win32/HLLP.Shodi.c, Win32/Hidrag.a, Win32/Slow.8192.a). Heuristiikkaan sisällytettiin kaikki neljälle viruksille yhteiset ja tiedostovirusten toiminnalle ominaiset funktiot FindFirstFile, FindNextFile, GetWindowsDirectory, ReadFile, WriteFile, CreateFile, SetFilePointer sekä ainakin toinen funktioista CreateProcess ja CreateThread.

Eri Windows-versioissa on funktioista esitelty uusia versioita, kuten FindFirstFileW ja GetModuleHandleA. Näiden uusien versioiden katsotaan vastaavan vanhoja APIen käyttöä tarkastettaessa. Virustartunta voidaan pyrkiä tunnistamaan etsimällä ohjelmasta tällaista tiedostoviruksille tyypillistä joukkoa API-kutsuja. Ohjelman käyttämät APIt löytyvät ILT:stä ja mahdollisesti myös IAT:stä, jos sen sisältöä ei ole korvattu APIen osoitteilla.

5.2.6 Otsakkeiden tarkistus tartuntamerkkien varalta

Monet virukset lisäävät otsakkeeseen oman tartuntamerkkinsä välttääkseen tartuttamasta samaa tiedostoa useampaan kertaan. Tartuntamerkki voi sijaita joko normaalisti käyttämättömässä kohdassa otsaketta, jossain harvoin hyödynnetyssä kentässä tai laittomana arvona jossakin tavallisestikin käytetyssä kentässä. Suurimmalla osalla otsakkeiden kentistä on oleellinen osuus ohjelman toiminnassa eivätkä virukset siksi voi muuttaa niiden arvoa tartuntamerkikseen estämättä ohjelman toimintaa. Otsakkeissa on kuitenkin myös ohjelman toiminnan kannalta merkityksettömiä tai harvoin käytettyjä osia, joita virukset voivat seuraamuksitta muuttaa. Viruksia voidaan PE-tiedostoista pyrkiä löytämään etsimällä otsakkeiden tietyistä kentistä niille epätyypillisiä arvoja. Heuristisen ratkaisun on tarkoitus kattaa mahdollisimman suuri joukko viruksia ilman päivityksiä yksittäisten virusten löytämiseksi, joten otsakkeista ei etsitä yksittäisten virusten tartuntamerkkejä, vaan kentistä etsitään niille epätyypillisiä arvoja.

PE-tiedoston aloittavassa MS-DOS-otsakkeessa on runsaasti käyttämättömiä tavuja. Osoitteissa 1Ch–23h ja 28h–3Bh olevien tavujen arvot ovat normaalisti nollia, eikä niiden sisältöä hyödynnetä ohjelman lataamisessa tai toiminnassa millään tavalla. Kyseiset tavut ovat siten vapaasti virusten muutettavissa ja esimerkiksi W32/Zmist tallentaa tartuntamerkkinsä osoitteeseen 1Ch (Ferrie ym. 2001) ja W32/Tenga osoitteeseen 32h (Neitzel 2005).

PE-otsakkeessa on kolme kenttää joita virukset voivat käyttää tartuntamerkkiensä tallentamiseen: TimeDateStamp, PointerToSymbolTable ja NumberOfSymbols. Aika ja päivämäärä ilmoitetaan PE-tiedostossa neljällä tavulla, joiden arvo ilmoittaa sekunneissa kuluneen ajan vuoden 1970 tammikuun 1. päivän keskiyöstä. Heuristinen

tarkistus varmistaa, ettei päivämäärä sijaitse tulevaisuudessa eikä epäilyttävän kaukana menneisyydessä. Testauksen aikana vanhin yleisesti tavattu päivämäärä oli kesäkuun 20. 1992 kello 01:22:17, joten sitä aikaisempia arvoja heuristiikka pitää epäilyttävinä. Myös pelkkiä nollia sisältävä kenttä on hyväksyttävä, koska kentän arvo ei ole pakollinen.

PointerToSymbolTable- ja NumberOfSymbols-kentät liittyvät ohjelman käyttämiin muuttujiin. Kentät ovat käytössä lähinnä objekti-tiedostoissa, jotka toimivat syötteenä linkkerille ja ovat myös PE-muodossa. Kenttien arvojen ollessa nollia, ne voidaan korvata millä tahansa arvoilla ohjelman toiminnan siitä kärsimättä. Nollasta poikkeava arvo on sellaisenaan epäluotettava heuristiikka, koska kentät ovat joskus käytössä myös ohjelmatiedostoissa. Luotettavampi tulos saadaan aikaan tarkistamalla poikkeaako vain toisen PointerToSymbolTable- ja NumberOfSymbols-kentistä arvo nollasta, koska virukset kuten W95/Lorez (Szor 1998b), W95/Invirsible (Szor 2000) ja W32/Magistr (Ferrie 2001), jotka tallentavat kyseisiin kenttiin tartuntamerkkinsä, muuttavat kentistä vain toista. Kenttien ollessa normaalissa käytössä, on niissä molemmissa nollasta poikkeava arvo. Heuristiikkaa testattaessa ilmeni, että kenttiä tartuntamerkkeihin hyödyntävien virusten määrä on käytännössä liian pieni syntyvien väärien positiivisten tulosten määrään verrattuna. Symbolitauluihin liittyvien kenttien tarkistaminen olisi hyödyllistä lähinnä jonkin hitaan algoritmisen menetelmän vaatimassa tiedostojen suodatuksessa.

5.2.7 Väärä tarkistussumma

Valinnaisen otsakkeen Checksum-kenttä sisältää neljän tavun tarkistussumman laskettuna koko tiedostosta. Tarkistussumman avulla käyttöjärjestelmä voi ennen ohjelman latausta varmistaa, ettei tiedoston sisältö ole tavalla tai toisella muuttunut. Tarkistussumma on kuitenkin pakollinen vain ajuri- ja DLL-tiedostoissa. Tavallisissa ohjelmatiedostoissa tarkistussumma voi olla mikä hyvänsä käyttöjärjestelmän siitä välittämättä. Useimmiten kentän arvo on tavallisissa ohjelmatiedostoissa pelkkiä nollia. Monet tiedostovirukset eivät päivitä isäntäohjelmansa tarkistussummaa, tai käyttävät kenttää jopa tartuntamerkkinsä sijaintina. Tarkistussumma voidaan laskea imagehlp.dll:n tarjoamalla rutiinilla, ja verrata Checksum-kenttään – väärä tarkistussumma saattaa olla seurausta virustartunnasta. Heuristiikan nopeuttamiseksi

tarkistussumman laskeminen on syytä rajata vain niihin tapauksiin, joissa Checksum-kentän arvo poikkeaa nollasta.

5.2.8 Heuristiikan sääntöjoukot

Heuristiikan pohjan muodostamiseksi pieni otos Win32-loisviruksia tartutettiin tavallisiin ohjelmatiedostoihin. Heuristiikan perustana toimivat internet-lähteestä hankitut 16 loisvirusta, jotka on lueteltu liitteessä 1 yhdessä niiden löytämiseen tarvittavien testien kanssa. Virusten tartuttamisprosessia nopeutettiin hyödyntämällä VMware Workstation -ohjelmistoa, jonka luomassa virtuaalikoneessa ajettiin Windows XP -käyttöjärjestelmää. Virtuaalikonetta hyödyntämällä virukset saatiin tartutettua ohjelmatiedostoihin kontrolloidusti, vaarantamatta tietokoneen varsinaista käyttöjärjestelmää. Lisäksi virtuaalikone saatiin varmuuskopiosta palautettua nopeasti puhtaaseen perustilaan kunkin viruksen tartuttamisen päätyttyä.

Virustartuntojen analysoinnin pohjalta havaittiin tulokohdan sijainnin viimeisessä osiossa, tulokohdan sisältävän osion ominaisuuksien ja tulokohdassa sijaitsevan hyppykäskyn toimivan erityisen luotettavina merkkeinä virustartunnasta. Mikään edellä mainituista ominaisuuksista ei kuitenkaan yksinään riitä tunnistamaan virustartuntaa ilman merkittävää määrää vääriä positiivisia tuloksia. Vain kahden testin yhdistäminen riitti kuitenkin vähentämään väärien positiivisten tulosten määrän hyvin pieneksi.

Isäntäohjelman loppuun uutena osiona itsensä lisäävät virukset heuristiikka etsii sääntöparilla

- tulokohta on viimeisessä osiossa ja
- tulokohdan sisältävä osio on kirjoitettavissa.

Tulokohdassa olevalla hyppykäskyllä suorituksen itseensä siirtäviä viruksia heuristiikka etsii sääntöparilla

- tulokohdassa on hyppykäsky ja
- otsakkeessa on tartuntamerkki tai tulokohta on kirjoitettavissa olevassa koodiosiossa.

Jos virus lisää itsensä johonkin muualle kuin viimeiseen osioon tai kokonaan uudeksi osioksi ohjelman loppuun, voidaan se löytää sääntöparilla

- tulokohdan sisältävä osio sisältää koodia, mutta on kirjoitettavissa ja
- otsakkeessa on tartuntamerkki.

Yksi 16:a viruksen joukossa olleista neljästä isäntäohjelman alkuun itsensä liittävästä loisviruksesta löydettiin ehdoilla

- viruksiin viittaavia API-kutsuja ja
- otsakkeen tiedoston luontiaikaa ilmoittavassa kentässä oleva tartuntamerkki.

Ohjelman koko muistissa ja erityisesti koodiosioiden väärä yhteiskoko auttavat useiden virusten löytämisessä, mutta toimivat lähinnä varmentavassa roolissa, jos etsinnän tuloksissa haluttaisiin käyttää eri varmuusasteita. Tarkistussumman ongelmana on sen isäntäohjelmasta riippuvuus, jos virus tarttuu tiedostoon, jolla ei ole tarkistussummaa jää virus löytämättä tarkistussummaan vahvasti perustuvalla heuristiikalla.

Yksittäisistä tarkistuksista PE-otsakkeen sijainti tiedoston jälkimmäisessä puoliskossa tai tulokohdan sijainti ennen ensimmäistä osiota riittävät yksinään virustartunnan tunnistamiseen.

Käyttöjärjestelmän muistialueelle ladattavan osion perusteella tehtävää tunnistamista ei voitu testata Windows XP -käyttöjärjestelmän alla. Kyseinen testi riittäisi oletettavasti yksinään tunnistamaan virukset suhteellisen luotettavasti, koska kyseessä on niin poikkeuksellinen ohjelman ominaisuus, eikä tarkistus aiheuttanut testeissä vääriä positiivisia tuloksia.

5.3 Heuristiikka EPO-viruksien löytämiseksi

5.3.1 Vertailu tavalliset tiedostovirukset löytävään heuristiikkaan

EPO-virusten ideana oleva tulokohdan piilottaminen tekee kaikista tulokohtaan suoraan liittyvistä tarkistuksista hyödyttömiä. EPO-virukset eivät koskaan muuta PE-otsakkeen AddressOfEntryPoint-kenttää, eivätkä isäntäohjelman tulokohdassa sijaitsevaa koodia, joten niihin liittyvät tarkistukset eivät auta EPO-virusten löytämisessä. Myös MS-DOS-

otsakkeen e_lfanew-kentän tarkistaminen on turhaa, koska EPO-virukset säilyttävät sen ennallaan.

EPO-virustenkin täytyy luonnollisesti jotenkin lisätä koodinsa isäntäohjelmaan. EPO-viruksista, jotka lisäävät koodinsa isäntäohjelman jonkin osion jatkeeksi, voidaan saada viitteitä osioiden ominaisuuksia tarkastelemalla, kuten tavallisista loisviruksista. Kenttien arvot ovat vääriä lähinnä vanhoissa, hieman kehittymättömissä Win32-viruksissa ja Win32-EPO-virukset puolestaan ovat suhteellisen uusia ja hienostuneita, joten ne useimmiten päivittävät kenttien arvot oikeiksi tartunnan yhteydessä.

Otsakkeiden tarkistaminen tartuntamerkkejä etsien voidaan suorittaa aivan samoin kuin tavallisille loisviruksille, koska tartuntamerkkien käyttö on täysin virustyypistä riippumatonta.

Yhteenvetona voidaan todeta, että EPO-virusten löytämiseen käyvät seuraavat tavallistenkin virusten löytämiseen käytetyt tunnisteet:

- epäilyttävät osioiden ominaisuudet,
- otsakkeissa olevat tartuntamerkit,
- väärä tarkistussumma.

Periaatteessa toimivia, mutta käytännössä lähinnä vanhoja EPO-menetelmiä käyttämättömiä viruksia löytäviä tunnisteita ovat

- kokoon liittyvien kenttien tarkistus,
- käyttöjärjestelmän muistialueelle tapahtuva osion lataus.

Täysin hyödyttömiä ovat kaikki tulokohtiin liittyvien kenttien tai itse tulokohdan tarkastukset.

5.3.2 IAT:n korvaavat EPO-virukset

EPO-virukset siirtävät suorituksen itseensä useimmiten korvaamalla jonkin funktio- tai API-kutsun kutsulla oman koodinsa alkuun. Tällaisten suorituksen siirtojen löytäminen vaatisi koko ohjelman koodin läpikäynnin, mikä hidastaisi nopeaan analysointiin tarkoitettua heuristiikkaa liikaa. Yleensäkin EPO-virukset ovat vaikeita tapauksia

staattisille heuristiikoille, koska tulokohdan analysointi on yksi luotettavimmista viruksien löytämiseksi tarjolla olevista heuristiikoista. Dynaamisetkaan heuristiikat eivät välttämättä auta, koska ohjelman suoritus ei etene virukseen asti tarpeeksi nopeasti.

Yhdenlaiset EPO-virukset on kaikesta huolimatta mahdollista löytää rakenteellisella staattisella heuristiikalla – nimittäin IAT:n korvaavat EPO-virukset kuten W32/Idele. Dynaamisella heuristiikalla on näidenkin virusten tapauksessa sama ongelma kuin kaikissa EPO-viruksissa: emulointi saatetaan lopettaa ennen viruksen suoritukseen asti pääsemistä.

IAT:n korvaavat virukset eivät muuta isäntäohjelman varsinaista ohjelmakoodia siirtämään ohjelman suoritusta virukseen, vaan korvaavat tuontitaulun sisältämiä APIen osoitteita omalla tulokohdallaan. Todelliset APIen osoitteet virus tallentaa itsensä osaksi ja mahdollistaa sitä kautta alkuperäisen IAT:n palauttamisen ja isäntäohjelman toiminnan. Ohjelmaan on tarttunut W32/Idelen kaltainen EPO-virus, jos IAT sisältää muistiosoitteen, joka osoittaa ohjelman omaan muistialueeseen eikä johonkin ulkopuoliseen moduliin. IAT:ssä muistiosoitteet ovat poikkeuksellisesti absoluuttisia eivätkä suhteellisia, ne ovat siis muotoa imageBase + RVA (imageBase määritellään valinnaisessa otsakkeessa ja se määrää ohjelman alkupisteen keskusmuistiin ladattuna). Jos IAT:ssa oleva muistiosoite on suurempi kuin imageBase, mutta pienempi kuin imageBase + sizeOfImage – 1, osoittaa se ohjelman omaan muistialueeseen, mikä tarkoittaa suurella todennäköisyydellä EPO-virustartuntaa.

Käytännön testeissä vääriä positiivisia aiheutui 1584 ohjelmatiedoston joukosta noin prosentti. Vaikka osuus onkin pieni, eliminoitiin väärät positiiviset kokonaan lisäämällä testiin ehto, joka vaatii kahden peräkkäisen IAT:ssa olevan muistiosoitteen olevan samoja ja osoittavan moduulin omaan muistialueeseen. Haittapuolena heuristiikka ei lisäyksen seurauksena enää löydä EPO-viruksia, jotka korvaavat IAT:sta ainoastaan yhden muistisoitteen omalla tulokohdallaan.

Ohjelman IAT voi sisältää myös täysin laillisia osoitteita omaan muistialueeseensa. Jos IAT:n sisältöä ei ole vielä korvattu APIen todellisilla osoitteilla, on IAT:n sisältönä osoitteet dll-tiedostoista tarvittavien funktioiden nimiin. Nämä osoitteet ovat kuitenkin RVA-muodossa, joten ne ovat aina imageBase-kentän arvoa pienempiä ja siksi helppo erottaa EPO-viruksen lisäämistä muistiosoitteista.

5.3.3 Heuristiikan sääntöjoukot

EPO-menetelmiä käyttämättömien virusten löytämisessä olivat avainasemassa tulokohdan sijainti, tulokohdan sisältämän osion ominaisuudet ja tulokohdan mahdollisesti sisältämä hyppykäsky. Yksikään näistä tarkistuksista ei auta EPO-virusten löytämisessä, mikä antaa hyvän kuvan EPO-virusten löytämisen vaikeudesta.

IAT:hen oman tulokohtansa lisäävät EPO-virukset on mahdollista löytää luotettavasti luvussa 5.3.2 kuvatulla tarkastuksella. Heuristiselle menetelmälle poikkeuksellisesti ei tarkastus testiajoissa aiheuttanut yhtään väärää positiivista tulosta vaikkei siihen yhdistetty muita tarkistuksia. Kaikki Win32/Idele-tartunnan saaneet tiedostot puolestaan löytyivät samalla yksittäisellä tarkistuksella.

Yleisistä tarkistuksista voitaisiin yhdistellä geneerinen sääntöjoukko, jolla olisi mahdollista löytää joitain EPO-viruksia:

- koodiosioiden väärä yhteiskoko
- epäilyttävät osion ominaisuudet
- tartuntamerkki otsakkeessa tai väärä tarkistussumma

Sääntöjoukko aiheuttaisi kuitenkin epämääräisyytensä vuoksi käytännössä liikaa vääriä positiivisia havaintoja.

5.4 Heuristiikan testaus ja täydentäminen suurella virusjoukolla

Alkuperäistä pieneen virusjoukkoon perustunutta heuristiikkaa testattiin ja parannettiin suurella satunnaisotoksella Win32-loisviruksia F-Securen Helsingin toimitiloissa. Kaikkiaan satunnaisotos koostui 5334 ohjelmatiedostosta, joista tietyllä viruksella saastuneiden tiedostojen määrä vaihteli yhdestä muutamaan kymmeneen. Ohjelmatiedostoista osa oli tartunnan jäljiltä niin korruptoituneita, ettei heuristiikka

niiden rakennetta pystynyt tarkastelemaan, joten käytännössä testauksessa käytetty joukko koostui 4510 saastuneesta ohjelmatiedostosta.

Alkuperäinen 16 virukseen perustunut heuristiikka tunnisti 2368 saastunutta tiedostoa 4510 mahdollisesta (52.5 prosenttia). Suhteellisen korkea osuus osoittaa alkuperäisen virusjoukon edustaneen yleisimpiä virustyyppejä. Testitulosten perusteella heuristiikan sääntöjoukkoihin lisättiin 4 alkuperäisiä tarkempia, useammasta kuin kahdesta yksittäisestä testistä koostuvaa sääntöjoukkoa, jotka on esitetty seuraavaksi.

Sääntö 1:

- ohjelman koko muistissa on eri kuin otsakkeessa ilmoitettu
- tulokohdassa on hyppykäsky
- virukseen viittaavia API-kutsuja

Sääntö 2:

- tulokohdan sisältävä osio sisältää koodia, mutta on kirjoitettavissa ja
- koodiosioiden yhteiskoko on väärä
- ohjelman luontiajan sisältävässä kentässä on tartuntamerkki tai tiedoston tarkistussumma on väärä

Sääntö 3:

- koodiosioiden yhteiskoko on väärä
- tulokohdan sisältävä osio on kirjoitettavissa
- ohjelman luontiajan sisältävässä kentässä tai MS-DOS-otsakkeessa on tartuntamerkki

Sääntö 4:

- ohjelman koko muistissa on eri kuin otsakkeessa ilmoitettu
- tulokohta on viimeisessä osiossa
- koodiosioiden yhteiskoko on väärä
- MS-DOS-otsakkeessa on tartuntamerkki

Sääntöjoukkojen lisäämisen jälkeen heuristiikka havaitsi 2895 saastunutta tiedostoa 4510 mahdollisesta (62.4 prosenttia). Osuutta olisi edelleen mahdollista nostaa uusia sääntöjoukkoja lisäämällä, mutta korkeintaan muutamia prosenttiyksikköjä, koska

yleisimmät virustyypit, jotka staattisella rakenteellisella heuristiikalla on ylipäänsä mahdollista löytää, ovat jo heuristiikassa mukana.

Väärien positiivisten tulosten osuus oli kahdessa eri testausympäristössä 0.28 prosenttia (5 väärää positiivista 1775 ohjelmatiedostosta) ja 0.09 prosenttia (2 väärää positiivista 2181 ohjelmatiedostosta). Yksinkertaisin tapa päästä eroon vääristä positiivisista olisi integrity checker -tyyppinen ratkaisu. Väärän positiivisen tuloksen aiheuttavista tiedostoista kootaan heuristiikkaan poikkeuslista, johon lisätään tiedoston nimi ja jonkinlainen tarkistussumma jolla tiedostoon tehdyt muutokset voidaan havaita. Heuristiikan tavatessa tiedoston, jonka nimi löytyy poikkeuslistalta, verrataan sen tarkistussummaa (ei otsakkeessa olevaa, vaan erikseen määriteltyä) poikkeuslistalta löytyvään, jolloin mahdollinen virustartunta voidaan havaita. Parempi ratkaisu olisi lisätä heuristiikkaan negatiivisia sääntöjä, joilla vääriä positiivisia aiheuttavat tiedostotyypit (erityisesti asennus- ja poisto-ohjelmat, jotka muistuttavat APIen käytöltään prepending-tyyppisiä viruksia) pyrittäisiin tunnistamaan ja välttämään niiden määrittely saastuneiksi vaikka jokin heuristiikan sääntöjoukko toisin väittäisi. Heuristisen menetelmän ollessa kyseessä on vääristä positiivisista kokonaan eroon pääseminen kuitenkin hyvin vaikeaa.

Yritykset nostaa löydettyjen tiedostojen osuutta merkittävästi reilusta 60 prosentista kasvattavat myös väärien positiivisten määrää huomattavasti. Seuraavalla sääntöparilla voitaisiin löydettyjen saastuneiden tiedostojen osuutta nostaa huomattavasti noin 75 prosenttiin:

- tulokohdan sisältävä osio sisältää koodia, mutta on kirjoitettavissa ja
- koodiosioiden yhteiskoko on väärä

Seurauksena väärien positiivisten määrä kuitenkin kasvoi kahdessa käytetyssä testiympäristössä 40 (5 kappaleesta 7:ään) tai peräti 350 prosenttia (2 kappaleesta 9:ään).

5.5 Eri tartuntatekniikoiden vaikutus heuristiikan toimintaan

Tartuntatekniikoista käydään ensin läpi luvussa 2.3 käsiteltyjen edistyneiden tartuntamenetelmien, ja sen jälkeen luvuissa 3.4 sekä 3.5 käsiteltyjen tiedostoihin liittymiseen käytettävien tekniikoiden vaikutus heuristiikan toimintaan.

Heuristiikka ei tarkastele varsinaista viruskoodia, joten viruksen mahdollisesti suorittama pakkaus tai itsekoodaus itsessään ei vaikeuta heuristiikan toimintaa. Pakkauksen sivuvaikutuksena loisviruksen havaitseminen saattaa kuitenkin vaikeutua. Esimerkkinä isäntäohjelman pakkaava virus, joka sijoittaa pakatun isäntäohjelman tiedostossa viruskoodin jatkeeksi. Tulokohta sijaitsee tällöin viruksen sijainnin vuoksi ensimmäisessä osiossa, jolloin heuristiikan on huomattavasti vaikeampaa tunnistaa virustartuntaa.

Polymorfismikaan ei yksinään auta virusta piiloutumaan esitetyltä heuristiikalta. Polymorfismiin on kuitenkin pakkausta tai itsekoodausta hyödyntäviä viruksia useammin yhdistetty muita viruksen havaitsemista vaikeuttavia menetelmiä. Esimerkiksi polymorfinen Win32/Parvo-virus jää heuristiikalta havaitsematta sen ennen itseensä suorituksen siirtävää hyppykäskyä lisäämien roskakäskyjen, ei polymorfismin vuoksi. Vahvoista polymorfisista viruksista heuristiikka löysi otoksessa olleet Win32/DeadCode.a:n tartuttamat tiedostot ja osan Win32/DeadCode.b:n tartuttamista tiedostoista, Win32/Tuareg jäi heuristiikalta löytämättä.

Ajatellaan kuvitteellista metamorfista virusta, joka muuttaa itsensä tartuntojen välillä niin täydellisesti, että kyseessä on jokaisen tartunnan jälkeen käytännössä eri virus. Heuristiikka löytää metamorfisen viruksen eri muodot samalla tavalla kuin se löytäisi täysin eri virukset, koska se ei tarkastele viruksen sisältöä. Käytännössä metamorfisen viruksen ohjelmointi on niin haastavaa, että siihen pystyvä henkilö tekee viruksesta myös muuten vaikeasti havaittavan (ääriesimerkkinä W95/ZMist). Metamorfismi itsessään ei siis aiheuta heuristiikalle ongelmia, mutta käytännössä metamorfiset virukset ovat yleensä staattiselle rakenteelliselle heuristiikalle muista syistä mahdottomia löydettäviä. Testeissä metamorfisista viruksista W95/ZMist jäi odotetusti löytämättä, mutta esimerkiksi W95/ZPerm.a löytyi.

Kuten ennakkoon arveltiin, osoittautuivat EPO-menetelmät staattiselle rakenteelliselle heuristiikalle vaikeimmaksi esteeksi. Tulokohdan sijainti ja sisältö ovat tärkeä osa loisvirusten luotettavaa löytämistä staattista heuristiikkaa käytettäessä, joten EPO-menetelmien käyttö poistaa tärkeimmät heuristiikan käytössä olevat keinot.

Poikkeuksena ovat IAT:ta muuttavat EPO-virukset, joiden löytämiseen tutkielmassa kehitetty heuristiikka tarjoaa poikkeuksellisen yksinkertaisen ja nopean, mutta samalla luotettavan menetelmän. API- tai aliohjelmakutsun korvaamalla suorituksen itseensä siirtäviä EPO-viruksia ei staattisella rakenteellisella heuristiikalla ole mahdollista löytää.

Yleisistä tartuntamenetelmistä siis ainoastaan EPO-tekniikoilla on suoranainen vaikutus staattisen rakenteellisen heuristiikan toimintaan. Huomattavasti suurempi vaikutus on tiedostoon liittymisessä käytetyllä tekniikalla.

Heuristiikalle helpoimpia tapauksia löytää ovat isäntäohjelman loppuun itsensä lisäävät loisvirukset. Heuristiikka löytää luotettavasti isäntäohjelman loppuun uuden osion lisäävät tai isäntäohjelman viimeisen osion jatkeeksi itsensä lisäävät loisvirukset, jotka vaihtavat otsakkeessa olevan tulokohdan osoittamaan itseensä. Myös uuden PE-otsakkeen isäntäohjelman loppuun lisäävät loisvirukset ja otsakevirukset löytyvät luotettavasti.

Obfuscated tricky jump -menetelmää käyttävistä tai isäntäohjelman alkuun itsensä lisäävistä loisviruksista heuristiikka löytää osan. Kummassakin tapauksessa viruksen löytäminen vaatii otsakkeessa olevan tartuntamerkin. Jotkut hyppykäskyllä alkuperäisestä tulokohdasta suorituksen itseensä siirtävät virukset lisäävät varsinaisen hyppykäskyn eteen roskakäskyjä. Toteutettu heuristiikka ei tällaisia viruksia löydä, mutta heuristiikkaa voitaisiin parantaa lisäämällä siihen mahdollisia roskakäskyjä läpikäyvä ja ohittava selaaja, jolloin hieman myöhemmin sijaitseva hyppykäsky löydettäisiin. Epäilyttävään APIen käyttöön perustuvan alkuun lisäävien loisvirusten löytämisen ongelmana ovat asennettujen ohjelmien poistamisen suorittavat ohjelmat. Nämä uninstall-ohjelmat käyttävät samoja API-funktioita kuin virukset, mikä aiheuttaa pelkkiin API-kutsuihin perustuvassa virusten löytämisessä liikaa vääriä positiivisia tuloksia. Tämän vuoksi tartuntamerkin löytäminen täytyy lisätä ehdoksi virustartunnan tunnistamiselle, mikä valitettavasti lisää väärien negatiivisten määrää, koska läheskään kaikki virukset eivät käytä tartuntamerkkejä. Yksi mahdollisuus parantaa tilannetta olisi

poisto-ohjelmien tunnistaminen ja tarkastamatta jättäminen, mutta tällöin ongelmaksi jäisi edelleen erotella virustartunnan saaneet poisto-ohjelmat puhtaista.

Staattisella rakenteellisella heuristiikalla ei ole lainkaan mahdollista löytää loisviruksia, jotka lisäävät itsensä johonkin muuhun kuin viimeiseen osioon, eivätkä lisää tulokohtaan hyppykäskyä.

6 YHTEENVETO

Tiedostovirusten löytämisen vaikeuttamiseksi on vuosien mittaan kehitetty yhä uusia menetelmiä. Viruksen sisältö on pyritty salaamaan tai muuttamaan itsekoodauksella tai poly- ja metamorfismilla ja sijainti piilottamaan EPO-menetelmillä. Virustyyppien monimuotoisuuden seurauksena ei loisvirusten löytämiseen ole mitään yksittäistä kaikkivoipaa menetelmää, vaan tehokkaan virustutkan täytyy yhdistellä useita eri menetelmiä toimivaksi kokonaisuudeksi kattavan virussuojan saavuttamiseksi. Virusten etsiminen merkkijonohauilla on vanhin ja edelleen käyttökelpoinen tapa etsiä yksinkertaisia loisviruksia. Koodin emuloinnin yhdistäminen virustutkaan on käytännössä edellytys monimutkaisempien virusten löytämiseen ja tunnistamiseen.

Win32-käyttöjärjestelmien ohjelmatiedostoissa käytetty PE-tiedostomuoto tarjoaa tarkkaan määritellyn rakenteensa johdosta yksinkertaisen tavan löytää loisviruksia. Tutkielman tavoitteena oli selvittää tällaisen ohjelmatiedostojen rakenteen tarkasteluun perustuvan staattisen heuristiikan toimintamahdollisuuksia kehittämällä heuristiikat sekä EPO-menetelmiä käyttämättömille tiedostoviruksille että EPO-viruksille.

Staattinen rakenteellinen heuristiikka tarjoaa nopean menetelmän loisvirusten geneeriseen löytämiseen. Kuten mikään muukaan menetelmä se ei yksinään riitä loisvirusten kattavaan löytämiseen, vaan kattaa testauksessa käytetyn satunnaisotoksen perusteella loisviruksista noin kaksi kolmasosaa. Kaikkiaan heuristiikka havaitsi 2895 saastunutta tiedostoa 4510 mahdollisesta (62.4 prosenttia). Osuutta voitaisiin nostaa uusia sääntöjoukkoja lisäämällä, mutta korkeintaan muutamia prosenttiyksikköjä, koska yleisimmät virustyypit, jotka staattisella rakenteellisella heuristiikalla on ylipäänsä mahdollista löytää, ovat jo heuristiikassa mukana. Väärien positiivisten tulosten osuus oli pelkistä ohjelmatiedostoista laskettuna kahdessa eri testausympäristössä 0.28 prosenttia (5 väärää positiivista 1775 ohjelmatiedostosta) ja 0.09 prosenttia (2 väärää positiivista 2181 ohjelmatiedostosta). Kehitetty heuristiikka tarjoaa mahdollisuuden löytää uusia, ennen tuntemattomia, loisviruksia ilman ohjelmistoon tehtäviä muutoksia (vrt. esimerkiksi merkkijonohakuihin perustuvien menetelmien tietokantapäivitykset). Virustentorjuntaohjelmistossa tutkielmassa kehitetyt heuristiikat sopisivatkin oletettavasti parhaiten ennestään tuntemattomien virusten etsimiseen. Heuristiikkojen

lopullisen soveltuvuuden kyseiseen tehtävään ratkaisee se, kuinka hyvin ne löytävät tällä hetkellä ilmestyviä viruksia. Testeissä viruksia ei eroteltu niiden tekemisajankohdan mukaan, joten tässä tutkielmassa ei voida antaa vastausta virusten valmistusajankohdan mahdollisesta vaikutuksesta staattisen rakenteellisen heuristiikan toimintaan.

Erityisen helppoja tapauksia staattiselle rakenteelliselle heuristiikalle ovat isäntäohjelman loppuun itsensä liittävät loisvirukset, jotka vaihtavat tiedoston otsakkeessa olevan tulokohdan osoittamaan itseensä. Lisäksi heuristiikan mielenkiintoinen ominaisuus on sen immuunius virusten käyttämille itsekoodaukselle sekä poly- ja metamorfismille. IAT:n korvaavien EPO-virusten löytämiselle tutkielmassa esitetty heuristiikka tarjoaa erittäin nopean menetelmän. Vastapainona staattinen rakenteellinen heuristiikka on muita EPO-menetelmiä käyttävien virusten edessä täysin voimaton. Toisena selkeänä heikkoutena ovat itsensä muuhun kuin isäntäohjelman viimeiseen osioon lisäävät loisvirukset, tosin tulokohdassa mahdollisesti oleva hyppykäsky saattaa riittää paljastamaan kyseiset virukset. Myös isäntäohjelman alkuun itsensä lisäävien virusten löytäminen ilman runsasta määrää vääriä positiivisia on heuristiikalla ongelmallista ja vaatisi vähintään asennus- ja poisto-ohjelmien tunnistamista ja erityiskohtelua. Heuristiikan jatkokehityksessä oleellisinta olisi väärien positiivisten tulosten vähentäminen entisestään mahdollisesti negatiivisia sääntöjoukkoja käyttämällä. Negatiivisten sääntöjoukkojen tarkoituksena olisi tunnistaa saastuneiden tiedostojen sijaan puhtaita ohjelmatiedostoja.

Sääntöjoukkojen luontiin olisi mahdollisesti hyödyllistä kehittää jonkinlainen itseoppiva järjestelmä, joka analysoisi erittäin suuria määriä saastuneita ja puhtaita tiedostoja ja loisi niiden perusteella sekä positiivisia että negatiivisia sääntöjoukkoja. Seurauksena syntyvät sääntöjoukot olisivat manuaalisella luonnilla saatuja tarkempia. Ei kuitenkaan ole varmuutta siitä, olisiko parannus merkittävä verrattuna tutkielmassa kehitettyihin sääntöjoukkoihin.

Jonkinlaisena ongelmana voidaan pitää staattisen rakenteellisen heuristiikan kykenemättömyyttä virheellisten, tartunnan yhteydessä pahasti korruptoituneiden

ohjelmatiedostojen tunnistamiseen saastuneiksi. Tiedoston rakenteen analysointiin perustuva heuristiikka ei pysty otsakkeistaan korruptoituneita tiedostoja analysoimaan.

Heuristiikan pohjana olevia testejä voitaisiin hyödyntää myös virusten analysoinnin apuvälineenä, esimerkiksi yhdistettynä itseään analysoivaan syöttitiedostoon. Loisvirus voidaan tartuttaa syöttitiedostoon, joka suoritettaessa ilmoittaa rakenteessaan tapahtuneista muutoksista. Seurauksena saadaan suuresta osasta tiedostoviruksia vaivatta selville mihin kohtaan isäntäohjelmaa se itsensä lisää, ja miten se siirtää suorituksen itseensä.

LÄHTEET

Arnold, W., Chess, D., Kephart, J., Sorkin, G. & White, S. 1995. *Searching for Patterns in Encrypted Data*. United States Patent 5442699.

Bontchev, V. 1994. *Are 'Good' Computer Viruses Still a Bad Idea?* Teoksessa Proceedings of the EICAR 1994 Conference. s. 25–47. St Albans, Iso-Britannia.

Bontchev, V. 1997. *Future Trends in Virus Writing*. International Review of Law, Computers & Technology, 11(1):129–146.

Bontchev, V. 1999. *The WildList - Still Useful?*. Teoksessa Proceedings of the 9th International Virus Bulletin Conference. s. 281–287. Vancouver, Kanada.

Bontchev, V. 2004. *Anti-Virus Spamming and the Virus Naming Mess - Part 2*. Virus Bulletin July 2004. s. 13–15. Virus Bulletin Ltd.

Bontchev, V. 2005. *Current Status of the CARO Malware Naming Scheme*. http://www.people.frisk-software.com/~bontchev/papers/pdfs/caroname.pdf viitattu 26.2.2006.

Bridwell, L. M. 2005. *ICSA Labs 10th Annual Computer Virus Prevalence Survey 2004* http://newlabs.icsalabs.com/icsa/docs/html/library/whitepapers/VPS2004.pdf viitattu 24.2.2006.

Chen, T. M. 2003. *Trends in Viruses and Worms*. The Internet Protocol Journal, 6:23–33.

Cohen, F., 1984. *Computer Viruses - Theory and Experiments*. Teoksessa 7th Security Conference, DOD/NBS. s. 143–158. Gaithersburg, Maryland, USA.

Common Malware Enumeration. 2006. *CME List*. http://cme.mitre.org/data/list.html viitattu 6.4.2006

Ferrie, P. 2001. *Magisterium Abraxas*. Virus Bulletin May 2001. s. 6–7. Virus Bulletin Ltd.

Ferrie, P. & Perriot, F. 2004. *Principles and Practice of X-raying*. Teoksessa Proceedings of the 14th International Virus Bulletin Conference. s. 51–66. Chicago, Illinois, USA.

Ferrie, P. & Szor, P. 2001. *Zmist Opportunities*. Virus Bulletin, March 2001. s. 6–7. Virus Bulletin Ltd.

Ferrie, P. & Szor, P. 2004. *Cabirn Fever*. Virus Bulletin August 2004. s. 4–5. Virus Bulletin Ltd.

Gryaznov, D. 1995. *Scanners of The Year 2000: Heuristics*. Teoksessa Proceedings of the 5th International Virus Bulletin Conference. s. 225–234. Boston, Massachusetts, USA.

Hinde, S. 2000. *Love Conquers All?* Computers & Security 19(5): 408–420.

Intel. 2006. *IA-32 Intel® Architecture Software Developer's Manual - Volume 2A: Instruction Set Reference, A-M.* Intel Corporation.

Jordan, M. 2002. *Dealing with Metamorphism*. Virus Bulletin October 2002. s. 4–6. Virus Bulletin Ltd.

Kapoor, A., Kumar, E. U. & Lakhotia, A. 2004. *Are Metamorphic Viruses Really Invincible?* Virus Bulletin December 2004. s. 5–7. Virus Bulletin Ltd.

Marinescu, A. 2003. *Russian Doll*. Virus Bulletin August 2003. s. 7–9. Virus Bulletin Ltd.

Marx, A. 2005. *The false positive disaster: Anti-Virus vs Winrar & Co.* Virus Bulletin November 2005 s. 11–13. Virus Bulletin Ltd.

Microsoft. 1999. *Microsoft Portable Executable and Common Object File Format Specification, Revision 6.0*. Microsoft Corporation.

Nachenberg, C. 1996. *Understanding and Managing Polymorphic Viruses.* Symantec Enterprise Papers Volume XXX.

Nachenberg, C. 1997. *Computer Virus-Anti Virus Coevolution.* Communications of the ACM 40(1): 46–51.

Nachenberg, C. 1999. *Computer Parasitology*. Teoksessa Proceedings of the 9th International Virus Bulletin Conference. s. 1–26. Vancouver, Kanada.

Neitzel, M. 2005. *Win32/Tenga.A*. http://www.nod32.bg/main.php?id=38&virusID=77 viitattu 19.4.2006.

Pietrek, M. 2002a. *Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format, Part 1*. MSDN Magazine February 2002. http://msdn.microsoft.com/msdnmag/issues/02/02/PE/ viitattu 21.4.2006

Pietrek, M. 2002b. *Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format, Part 2.* MSDN Magazine March 2002. http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/ viitattu 23.4.2006

Skulason, F. 1990. *1260 – Variable Virus.* Virus Bulletin March 1990, s. 12. Virus Bulletin Ltd.

Smidgeonsoft. 2006. *PEBrowse Professional Windows Disassembler.* http://www.smidgeonsoft.prohosting.com/pebrowse-pro-file-viewer.html viitattu 25.4.2006

Szor, P. 1996. *Nexiv_Der: Tracing the Vixen.* Virus Bulletin April 1996. s. 11–12. Virus Bulletin Ltd.

Szor, P. 1997. *Junkie Memorial.* Virus Bulletin September 1997. s. 6–8. Virus Bulletin Ltd.

Szor, P. 1998a. *Attacks on Win32.* Teoksessa Proceedings of the 8th International Virus Bulletin Conference. s. 57–84. München, Saksa.

Szor, P. 1998b. *Breaking the Lorez*. Virus Bulletin October 1998. s. 11–13. Virus Bulletin Ltd.

Szor, P. 1998c. *The Marburg Situation.* Virus Bulletin November 1998. s. 8–10. Virus Bulletin Ltd.

Szor, P. 2000. *The Invirsible Man*. Virus Bulletin May 2000. s. 8–9. Virus Bulletin Ltd.

Szor, P. 2004. *EPOCalypse Now!* Virus Bulletin July 2004. s. 2. Virus Bulletin Ltd.

Szor, P. 2005. *The Art of Computer Virus Research and Defense*. Addison-Wesley, Upper Saddle River, New Jersey.

Tanenbaum A., S. 2001. *Modern Operating Systems. 2nd Edition*. Prentice Hall, Upper Saddle River, New Jersey.

WildList Organization International. 2006. *PC Viruses In-the-Wild - February 2006.* http://www.wildlist.org/WildList/200602.htm viitattu 1.4.2006.

Www.viruslist.com. 2000. *Virus.Win9x.Murkry.399*.
http://www.viruslist.com/en/viruses/encyclopedia?virusid=19962 viitattu 29.3.2006

Yetiser, T. 1992. *Mutation Engine Report.* VIRUS-L Digest 5(122).
http://securitydigest.org/virus/mirror/www.phreak.org-virus_l/1992/vlnl05.122 viitattu 18.1.2007

LIITE 1: HEURISTIIKKOJEN POHJANA KÄYTETYT VIRUKSET

Nimet ovat Kaspersky Labin viruksista käyttämiä. Useimmat viruksista aiheuttivat myös muita positiivisia testituloksia, mutta vain sääntöjoukon muodostavat on mainittu.

Win32.Bika.1906, Win32.Kaze.2056, Win32.Ramdile, Win32.Xorala, Win32.Tosep.1419

- tulokohta on viimeisessä osiossa
- tulokohdan sisältävä osio on kirjoitettavissa

Win32.Apathy.5378, Win32.Cabanas.release, Win32.Oporto.3076

- tulokohdassa on hyppykäsky
- otsakkeessa on tartuntamerkki tai tulokohta on kirjoitettavissa olevassa koodiosiossa

Win32.Glyn

- tulokohdan sisältävä osio sisältää koodia, mutta on kirjoitettavissa
- otsakkeessa on tartuntamerkki

Win32.Slow.8192.a

- viruksiin viittaavia API-kutsuja
- otsakkeen tiedoston luontiaikaa ilmoittavassa kentässä oleva tartuntamerkki

Win32.Idele.2108

- IAT:ssä kaksi peräkkäistä samaa osoitetta moduulin omaan muistialueeseen

Win32.IKX (lisää itsensä ensimmäisen osion loppuun ja siirtää muita osioita eteenpäin)
Win32.Parvo (obfuscated tricky jump jota edeltää roskakäskyjä, lisää viimeiseen osioon)
Win32.Hidrag.a (isäntäohjelman alkuun itsensä lisäävä)
Win32.HLLP.Shodi.c (isäntäohjelman alkuun itsensä lisäävä)
Win32.HLLP.Sypon.a (isäntäohjelman alkuun itsensä lisäävä)