jamk

Haittaohjelmien välttelytekniikat ja torjuntastrategiat

Thomas Semenius

Opinnäytetyö, AMK
Kesäkuu 2025
Tieto- ja viestintätekniikan tutkinto-ohjelma

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Kuvailulehti

Semenius, Thomas

Haittaohjelmien välttelytekniikat ja torjuntastrategiat

Jyväskylä: Jyväskylän ammattikorkeakoulu. Kesäkuu 2025, 40 sivua

Tieto- ja viestintätekniikan tutkinto-ohjelma. Opinnäytetyö AMK.

Julkaisun kieli: suomi

Julkaisulupa avoimessa verkossa: kyllä

Tiivistelmä

Kyberturvallisuuden kontekstissa haittaohjelmat muodostivat merkittävän uhan tietojärjestelmille erityisesti kehittyneiden välttelytekniikoiden vuoksi, jotka vaikeuttivat haitallisen toiminnan tuottamista ja analysointia. Jatkuvasti kehittyvien uhkakuvien ja kasvavan haittaohjelmatuotannon myötä oli tarpeellista selvittää, mitkä välttelytekniikat esiintyivät nykyaikaisessa haittaohjelmistossa ja millä keinoilla niiden tehokas torjunta voitiin toteuttaa. Työn tavoitteena oli tunnistaa ajantasaiset tekniikat, joilla haittaohjelmat pystyivät välttämään havaitsemista niin staattisen kuin dynaamisen analyysin aikana ja arvioida, miten niitä vastaan voitiin kehittää suojelutoimenpiteitä.

Tutkimuksessa sovellettiin kirjallisuuskatsausta sekä kokeellista menetelmä, jossa kehitettiin tarkoitukseen rakennettu haitallinen ohjelmaesimerkki, joka sisälsi dokumentoidut välttelymekanismit kuten anti-debugging, anti-disassembler, obfuskaation sekä sandbox evasion. Luodussa haittaohjelman läpikäynnissä on tarkoituksena oppia käyttämään ohjelmakoodin purkua ja analyysiä käyttäen nykyaikaisia työkaluja, joita ovat kuten x32/64dbg, IDA 9 sekä FlareVM. Näillä työkaluilla tutkittiin haittaohjelman toimintaa sekä analyysin välttelyä.

Tuloksissa ilmeni, että yhdistämällä useita yksinkertaisia tekniikoita – kuten PEB- ja API-pohjaisia debuggereiden tunnistuksia, virtuaalikoneprosessien tarkkailu voitiin luoda havaintoa välttelevä ohjelma, joka onnistui välttämään perustason hiekkalaatikkoympäristöjä. Tiedonkeruun perusteella havaittiin myös, että dynaamiset tarkistusmekanismit, kuten exception-pohjaiset ohjausvirran manipuloinnit ja ajastukseen perustuvat viivästyvät payloadit aiheuttavat lisähaasteita automatisoiduille analyysityökaluille.

Johtopäätöksenä todettiin, että haittaohjelmien välttelytekniikat perustuivat usein yksinkertaisiin, mutta tehokkaasti yhdistettyihin menetelmiin, joiden torjuminen edellytti laaja-alaista havaintomekanismien päivittämistä sekä analyysityökalujen kehittämistä. Etenkin aikapohjaiset sekä interaktiota vaativat tekniikat osoittautuivat tehokkaiksi perinteisiä dynaamisen analyysin ympäristöjä vastaan. Lisäksi ilmeni, että dokumentoitujen tekniikoiden läpinäkyvyys helpotti vastatoimien suunnittelua, mikä kannustaa jatkuvaa tutkimustyötä ja testausympäristöjen kehittämistä kyberturvallisuuden alalla.

Avainsanat (asiasanat)

Kyberturvallisuus, haittaohjelmat, tietoturva, välttelytekniikat

Muut tiedot (salassa pidettävät liitteet)

jamk

Description

Semenius, Thomas

Malware Evasion Techniques and Counteracting Strategies

Jyväskylä: JAMK University of Applied Sciences, June 2025, 40 pages

Degree Programme in Information and Communication Technology. Bachelor´s thesis.

Permission for open access publication: Yes

Language of publication: Finnish

Abstract

In the context of cybersecurity malware posed a significant threat to information systems due to advanced evasion techniques that made it difficult to produce and analyze malicious activity. With constantly evolving threat scenarios and increasing malware production there was need to identify which evasion techniques were present in modern malware and how they could be effectively countered. The aim of this work was to identify the state-of-the-art techniques that enabled malware to evade detection during both static and dynamic analysis and to assess how protection measures could be developed against them.

The study applied experimental methodology to develop a purpose-built malware example that included documented evasion techniques such as anti-debugging, anti-disassembler, obfuscation and sandbox evasion. The purpose of the created malware walkthrough is to learn how to use program code decomposition and analysis using modern tools such as x32/64dbg, IDA 9 and FlareVM. These were the tools used to study the malware's behavior and analysis evasion. The methodology was supported by recent research literature and technical sources, which were used to provide context for the implementation and observations.

The results showed that by combining several simple techniques - such as PEB- and API-based debugger detection, monitoring of virtual machine processes - a detection-avoiding program could be created that managed to avoid basic sandbox environments. Data collection also revealed that dynamic checking mechanisms such as exception-based control flow manipulations and timing-based delayed payloads create additional challenges for automated analysis tools.

In conclusion, it was found that malware evasion techniques were often based on simple but effectively combined methods, which required extensive updating of detection mechanisms and development of analysis tools. Time-based and interaction techniques proved effective against traditional dynamic analysis environments. It was also found that the transparency of the documented techniques facilitated the design of countermeasures, encouraging continued research and development of testing environments in cybersecurity.

Keywords/tags (subjects)

Cybersecurity, malware, data security, evasion techniques

Miscellaneous (Confidential information)

Sisältö

Kuviot

1 Johdanto

Haittaohjelmilla (englanniksi malware) tarkoitetaan tietojärjestelmiin kohdistuvia ohjelmistoja tai koodin osia, joiden tarkoituksena on aiheuttaa vahinkoa, häiriötä tai luvattomia toimenpiteitä ilman käyttäjän suostumusta. Haittaohjelmat ovat jatkuvasti kehittyvä uhka, jotka voivat aiheuttaa laajoja rahallisia sekä mainehaittoja niin yhteiskunnalle, yrityksille ja yksilöille. Tämä tekee haittaohjelmien torjuntakeinojen tuntemuksesta olennaisen osan ammattimaista kyberturvallisuusosaamista. Haittaohjelmien torjunta vaatii jatkuvaa kehitystä torjuntastrategioiden kanssa vastaamaan haittaohjelmien jatkuvaa muutosta sekä monimuotoisuutta.

Opinnäytetyön lähtökohtana on tarve ymmärtää, millä tavoin haittaohjelmat pyrkivät välttämään havaitsemista ja miten niitä voidaan tehokkaasti analysoida ja torjua. Työssä perehdytään erityisesti haittaohjelmien käyttämiin välttelytekniikoihin, joita hyödynnetään tilanteissa, joissa ohjelma toimii valvotussa ympäristössä, kuten debuggausalustalla, virtuaalikoneessa tai automaattisen analyysin kohteena sandboxissa. Tavoitteena on yhdistää teoreettinen tarkastelu ja käytännön toteutus siten, että lopputuloksena syntyy sekä kattava yleiskuva haittaohjelmien välttelystrategioista että konkreettinen esimerkki siitä, miten tällaiset tekniikat voivat ilmetä analyysissä.

Opinnäytetyössä tuotetaan tekninen Proof-of-Concept-ohjelma, joka simuloi tunnettuja välttelytekniikoita ja tarjoaa mahdollisuuden havainnoida niiden toimintaa suorituksen aikana. Toteutus palvelee samalla koulutuksellista tarkoitusta, sillä sitä voidaan käyttää opetuksessa esimerkkinä haittaohjelmien toiminnasta ja niihin kohdistuvista analyysi- ja torjuntamenetelmistä. Työssä selvitetään, mitkä tekniikat ovat yleisimpiä nykyaikaisessa haittaohjelmistossa, miten näitä voidaan teknisesti havaita ja mitkä keinot tarjoavat suojaa tai mahdollistavat analyysin etenemisen. Tutkimuksen aineisto koostuu ajankohtaisista tutkimuksista, artikkeleista, yritysblogeista ja teknisistä lähteistä, joiden pohjalta toteutus ja analyysi rakennetaan. Lisäksi työn aikana pyritään tunnistamaan kehityskohteita ja jatkotutkimuksen mahdollisuuksia, jotka voivat parantaa haittaohjelmien torjuntaa sekä syventää alan ammatillista osaamista.

2 Haittaohjelmat

2.1 Mitä ovat haittaohjelmat

Haittaohjelmat eli malware, lyhenne sanoista malicious software, ovat omalaatuinen lisäys ohjelmistojen maailmaan. Yleistäen ohjelmien tarkoitus on helpottaa ja tuottaa hyötyä käyttäjille, kun taas haittaohjelmien ensisijainen tarkoitus on aiheuttaa vahinkoa jossain muodossa. Teknisesti haittaohjelma suorittaa käyttäjän tai järjestelmän kannalta haitallisia toimintoja ilman käyttäjän suostumusta taikka tietämystä. Haittaohjelmien toiminta perustuu ohjelmoinnin neutraaliin luonteeseen. Mikäli koodi täyttää ohjelmoinnin puolesta logiikan ja muut ehdot, voidaan kyseinen koodi ajaa koneessa riippumatta sen eettisestä tarkoituksesta. Haittaohjelmat ovat kuitenkin laaja sateenvarjotermi, jonka alle ne voidaan jakaa toiminnallisuuden mukaan alakategorioihin, joita esitellään tarkemmin luvussa 2.2. Nykyaikaiset haittaohjelmat ovat kuitenkin yhä useammin teknisempiä hybridejä ja voivat täyttää useamman luokituksen samaan aikaan. Esimerkkinä tästä on Heinemeyerin (2019) esitys, että Emotet-haittaohjelma täyttää samaan aikaan troijalaisen ja madon kategorioita.

Nykyisessä digitaalisessa ekosysteemissä kyberturvallisuusuhkien määrä ja monimuotoisuus ovat kasvaneet räjähdysmäisesti. Microsoftin Digital Defense Report 2024 mukaan sen asiakkaita kohdeltiin keskimäärin 600 miljoonalla haittaohjelma- ja muiden kyberhyökkäysten yrityksellä päivittäin ajanjaksolla heinäkuu 2023 – heinäkuu 2024 (Microsoft Digital Defense). Vastaavasti Check Point Research raportoi 30 % kasvun viikoittaisten hyökkäysten määrässä Q2 2024 verrattuna Q2 2023:een, mikä havainnollistaa uhkien jatkuvaa kiihtymistä ja uhka-aktiivisuuden kausivaihteluita. (Check Point Research Reports Highest Increase of Global Cyber Attacks seen in last two years – a 30% Increase in Q2 2024 Global Cyber Attacks 2024.)

Euroopan kyberturvallisuusvirasto ENISA listasi Threat Landscape 2024 -raportissaan seitsemän pääuhkaa ajanjaksolla kesäkuu 2023 – heinäkuu 2024. Näistä merkittävimpiä olivat palvelunestohyökkäykset (availability), kiristysohjelmat (ransomware) ja tietohyökkäykset (data attacks), jotka yhdessä muodostavat enemmistön vakavista ja julkisuudessakin noteeratuista hyökkäyksistä. (ENISA Threat Landscape 2024.) Voidaan sitten todeta, että haittaohjelmat ovat vieläkin akuuttiosa kyberturvallisuutta.

2.2 Erilaiset haittaohjelmat/luokitukset

2.2.1 Ransomware

Ensimmäisenä tarkastellaan ransomware-haittaohjelmia, jotka ovat yksi näkyvimmistä ja tuhoisimmista haittaohjelmatyypeistä nykypäivänä. Berardi ja muut (2023) määrittävät lyhyesti ransomwaren yleistermiksi, jota käytetään kuvaamaan haittaohjelmaluokitusta, mitä käytetään digitaalisesti kiristämään uhreja maksamaan tietyn summan. Tarkemmin Ransomwaren ainoa tehtävä on evätä pääsy tietokonejärjestelmään tai salaamaan sen sisältämät tiedostot, kunnes kiristyslunnaat ovat maksettu. Tämän haittaohjelman alagenreistä suuren suosion on saavuttanut niin sanottu kryptoalaluokka, joka salaa uhrin tiedostot, kunnes uhri siirtää kryptovaluuttaa kyberuhkatoimijan lompakkoon saadakseen salausavaimen tiedostojen avaamiseen. Nämä hyökkäykset ovat yleistyneet ajan kanssa samalla kuin hyökkäyksistä on tullut kehittyneempiä. Tämä haittaohjelma voidaan kohdistaa yksilöihin, yrityksiin sekä laajimmillaan valtioiden instituutioihin globaalisti. Yleisiä levitystapoja ovat erilaiset ”kalastussähköpostit”, haitalliset lataukset sekä muiden exploittien hyväksikäyttö. (Berardi ym. 2023.)

2.2.2 Tiedostoton haittaohjelma

Tiedostottomat haittaohjelmat poikkeavat muista haittaohjelmaluokituksista toimivat ilman suoritettavia tiedostoja, mikä vaikeuttaa sen havaitsemista sekä poistamista. Suoritettavan tiedoston puuttumisen takia haittaohjelma sijaitsee järjestelmän keskusmuistissa, josta se käyttää järjestelmän omia sallittuja prosesseja sekä työkaluja omien pahantahtoisten toimintojen ajamiseen. Tästä syystä tiedostomattomien haittaohjelmien havaitseminen allekirjoitus pohjaisilla tunnistusjärjestelmillä on hankalaa. (Sudhakar & Kumar 2020.)

Sudhakar esittää, että tiedostottomat haittaohjelmat voidaan luokitella kolmeen eri kategoriaan. Ensimmäinen luokituksista on muistia käyttävät haittaohjelmat, jotka käyttävät vain keskusmuistia säilyttääkseen itseään aktivointiin asti, jolloin ne käyttävät Windowsin omia tiedostoja suorittamaan tarkoituksensa. Toinen ehdotetuista luokituksista on Windows-rekisteriä käyttävät haittaohjelmat. Windows-rekisteri on tietokanta Windowsista käyttöjärjestelmänä sekä joistakin kriittisistä sovelluksista. Tämän alaluokituksen haittaohjelmat kirjoittavat lähdekoodinsa salattuna rekisteriin, jossa se selviytyy käyttämällä järjestelmän välimuistia. Aktivoinnin jälkeen ne poistavat itsensä rekisteristä sekä pyyhkivät jälkensä ympäristöstä ja lokitiedostoista. Ehdotuksen viimeinen variaatio

on tiedostoton rootkit-haittaohjelma, jossa kyberuhkatoimija voi asentaa käyttöjärjestelmän kernel-tasolle saavutettuaan järjestelmänvalvojan oikeudet muuta kautta. Tutkielmassa kuitenkin mainitaan, että kyseinen tapahtuma ei ole sata prosenttisesti tiedostoton tapahtuma. (Sanjay ym. 2018.) Tämä huomioiden se voidaan silti katsoa kuuluvan tiedostottomien haittaohjelmien alaluokkaan.

2.2.3 Spyware ja Adware

Kolmantena haittaohjelmatyyppinä käsitellään spyware- ja adware-ohjelmia. Spyware eli vakoiluohjelmat ja Adware eli mainosohjelmat ovat haittaohjelmia, jotka ovat kehitelty keräämään tietoja salaa käyttäjästä sekä seurata uhrin toimintoja verkossa ilman tämän huomaamista (Adware vs Spyware n.d.). D'Andrea (2014) esittää, että adware kuuluisi osaksi spywarea, mutta monet lähteet kuten Ciscon artikkelissa Adware vs Spyware (n.d.) erittävät nämä eri alakategoriaan. Adware vs Spyware esittää adwaren tuottavan kehittäjälleen rahaa näyttämällä käyttäjälle mainoksia käyttöjärjestelmässä tai selaimessa. Adwaret ei välttämättä ole haitallisia, ne voi ohjata turvattomille sivuille tai sivuladata vakavampia haittaohjelmia. Spyware on aina haitallinen ja sen tarkoitus on kerätä informaatiota uhrista tai uhrin organisaatiosta sekä välittää nämä tiedot haittaohjelman käyttäjälle. Spywarea käytetään usein varastamaan henkilökohtaisia- ja taloudellisia tunnuksia. Tämä luokittelun epätäsmällisyys mahdollistaa sen, että spywaren alle voidaan luokitella monia muitakin haittaohjelma luokituksia. (Adware vs Spyware n.d.) D'Andrea (2014) ehdottaa seitsemää eri luokkaa, mikä koostuisi Adware, Stalkerware, valtiotason spyware, troijalaiset, rootkit, keyloggers ja browser hijackers. Näistä yleisesti myös troijalaiset, rootkit sekä keyloggerit kuten adware pidetään omana alakategoriana. (D'Andrea 2014.)

Stalkerwarella tarkoitetaan ohjelmia, jolla sen asentaja voi salaa seurata laitteen sijaintia sekä toimintaa ilman uhrin huomaamista. Näin asentaja voi valvoa lähes kaikkea laitteen toiminnallisuuksista kuten viestejä, sosiaalisenmedianprofiileja, valokuvia, puhelu- ja selainhistoriaan (What is Stalkerware n.d.). Osa spywaresta on alun perin luotu taistelemaan rikollisuutta vastaan, mutta niitäkin on käytetty vakoilemaan kaikki, joilla on luottamuksellisia tietoja. Esimerkkinä tästä on Pegasus. Pegasus on israelilaisten kehittämä spyware-puhelinten salakuunteluun sekä tietojen keräämiseen. Sen alkuperäinen tarkoitus oli valtion turvallisuuden ylläpitäminen ja lainviranomaisten käyttöön erinäisten rikollisten torjuntaan. Pegasus toimii useammissa puhelinkäyttöjärjestelmissä kuten Android-, iOS-, BlackBerry-, Windows-Phone- ja Symbian-käyttöjärjestelmissä. (Greengard

2025.) Browser highjacker eli selaimen kaappaaja voi ohjata uhrin väärille verkkosivuille. Sitä voidaan käyttää myös saadakseen uhrin henkilökohtaisia tietoja sekä seuratakseen selaustoimintaa. Samalla se voi muuttaa selaimen muita asetuksia kuten selaimen suojausasetuksia ja hakukonetta. (D'Andrea 2014.)

2.2.4 Troijalainen

Troijalaiset, kuten muinaiskreikasta johdettu nimi, ovat haittaohjelmia, jotka ovat naamioitu laillisiksi ohjelmiksi. Toisin kuin monet muut haittaohjelmat, troijalaiset eivät monistu itsestään vaan tarvitsevat käyttäjiä aloittamaan niiden lataukset. Järjestelmään päästyään troijalaiset ovat tikittävä aikapommi, joka odottaa sopivaa hetkeä sisältönsä toteuttamiseksi. Tämä ”payload” voi vaihdella käyttöjärjestelmää lamauttavasta ja häiritsevät toiminnasta käyttäjien vakoiluun sekä tietojen keräämiseen. Troijalaiset ovat yleensä osa monimutkaisempia hyökkäyksiä, joissa ne toimivat välineinä kyberturvauhkatekijöiden käytössä. Kun troijalainen on saavuttanut valmiutensa, uhkatekijät saavat hallinnan, jonka avulla he voivat suunnitella ja toteuttaa koordinoituja hyökkäyksiä. Tämä voi johtaa monenlaisiin vaaroihin, kuten laajoihin tietomurtoihin, DDoS-hyökkäyksiin ja muiden haittaohjelmien levittämiseen. Esimerkkinä tällaisen hyökkäyksen aloittamisesta on ”Downloader Trojans”. Toinen hyvin tunnettu troijalaisten alakategoria on Remote Access Trojan, monelle lempinimellä rotta, johtuen lyhenteestä R.A.T. Näiden rottien avulla uhkatoimijat voivat ottaa uhrin koneet etähallintaansa ja näin saavuttaen järjestelmänvalvojan oikeudet. Kolmas yleinen troijalaisen luokituksia on pankkitroijalaiset. Pankkitroijalaiset keskittyvät usein verkkoselaimiin käyttäen man-in-the-middle- hyökkäyksiä kerätäkseen käyttäjätunnuksia, salasanoja sekä luottokorttitietoja.

2.2.5 Madot

Madot ovat haittaohjelmaluokka, jonka ensisijainen tehtävä monistuttaa itseään sekä tartuttaa muita tietokoneita pysyessään aktiivisena. Yleensä käyttäjä reagoi madon olemassaoloon, vasta kun sen hallitsematon monistuminen kuluttaa järjestelmän resursseja ja täten alkaa hidastamaan tai pysäyttämään muita tehtäviä. Matojen leviäminen perustuu usein File Transfer Protocollan haavoittuvuuksiin. Esimerkkeinä madoista on ensimmäisenä *Stuxnet*, Yhdysvaltoihin ja Israeliin yhdistetty mato levitettiin Iranissa USB-asemien kautta ja toisena *WannaCry*, joka on yhdistetty Pohjois-Korealaisen Lazarus Groupiin. WannaCry toimi ransomwarena salaten koneen kaikki tiedot ja

käytti leviämiseen haavoittuvuutta Windows Server Message Blockin ensimmäisessä versiossa, joka on resurssienjakoprotokolla. Saastutettuaan tietokoneen se alkoi etsiä verkosta uusia mahdollisia uhreja, jotka vastasivat madon tekemiin pyyntöihin. Näin WannaCry vaikutti satoihin tuhansiin tietokoneisiin jopa 150 maassa ympäri maailmaa. (Bedell 2022)

2.2.6 Virus

Virukset ovat haittaohjelmia, jotka on suunniteltu tartuttamaan isäntäsovelluksia, kopioimaan koodinsa ja leviämään muihin tiedostoihin tai dokumentteihin. Latto (2022) määrittelee viruksen ohjelmaksi, joka kiinnittyy muuhun suoritettavaan koodiin, voi itse monistua ja levitä tietokoneisiin. On kuitenkin hyviä jakaa virukset kahteen kategoriaan: heti aktiivisiin ja odottaviin. (Latto 2022) Kuitenkin Vijayan ja Lawton (2024) esittävät, että klassisten virusten uhka ei ole enää suuri, vaikka ne voivat olla ärsyttäviä. Modernimmat hyökkäykset kuten esimerkiksi ransomware muodostavat suurimman osan hyökkäyksistä. Tästä syystä on organisaatioiden ylläpidettävä virustorjunta sekä haittaohjelmaominaisuuksia. (Vijayan & Lawton 2024)

2.2.7 Rootkit

Rootkit on haittaohjelma, joka luo jatkuvan etuoikeutetun pääsyn tietokoneeseen ja piilottaa läsnäolonsa sulautumalla syvälle käyttöjärjestelmän sisälle. Muiden kategorioiden haittaohjelmat saattavat huomauttaa käyttäjää läsnäolostaan hidastamalla tietokoneen toimintaa tai tekemällä muita havaittavia muutoksia, kun rootkitit on suunniteltu niin, ettei niitä havaita. Tästä syystä ne pysyvät usein salassa pitkään, mikä nostaa niiden uhkaavuutta huomattavasti. Rootkit sai alkunsa Unix-järjestelmässä osana järjestelmänvalvojien työkalupakkia. Ajan kanssa näistä työkaluista tuli kehittyneempiä versioita, josta syystä kyberturvauhkatekijät valitsivat ne myös osaksi omia työkalujaan. 2000-luvun alusta lähtien rootkitit ovat olleet erittäin kehittyneitä ja ne pystyivät hyödyttämään kernel-tason oikeuksia saadakseen syvän hallinnan järjestelmästä. Tämä on myös yksi rootkitien yleisimmistä tekniikoista ja samalla myös vaarallisin johtuen niiden syvästä integroitumisesta järjestelmään. Toinen näistä tekniikoista on muihin prosesseihin itsensä injektointi. Ne pystyvät ohittamaan monet tietoturvatohjelmat olemalla osana luotettuja ohjelmia. Kolmantena vaihtoehtona on tiedostojärjestelmän manipulointi. Manipuloimalla tiedostorakennetta voi haittaohjelma piilottaa itsensä sekä käyttäjältä että virusohjelmalta. Vastaavasti neljäs tekniikka piilottaa itsensä käynnistyksen yhteyteen. Näin se varmistaa oman käynnistyksen ennen käyttöjärjestelmää. Tämä

myös tarkoittaa, että kyseinen variaatio selviää käyttöjärjestelmän uudelleen asennukselta. Viimeinen näistä yleisistä tekniikoista on verkkoliikenteen uudelleenohjaus. Uhkatekijä voi tämän uudelleenohjauksen kautta suorittaa muuta toimintaa, kuten tietojenkalastelua tai sisällyttää omaa haittakuormaa liikenteeseen.

2.2.8 Keyloggerit

Keyloggerit ovat eräänlainen variaatio rootkit-haittaohjelmista, jotka on suunniteltu erityisesti kaappaamaan näppäimistön näppäinpainallukset ja tallentamaan omiin lokitiedostoihin. Näin kyberuhkatoimijat voivat anastaa arkaluonteisia tietoja, kuten käyttäjätunnuksia, salasanoja ja PIN-koodeja ilman huomiota käyttäjältä. Keyloggerit voidaan alaluokitella kahteen eri versioon: laitteisto- ja ohjelmistopohjaisiin-keyloggereihin. Laitteisto-keyloggerit edellyttävät fyysistä pääsyä kohdejärjestelmiin, kun taas ohjelmistopohjaiset-keyloggerit voidaan asentaa etänä kohdejärjestelmiin ja ne ovat näin yleisempiä. Nämä muodostavat merkittävän uhan sähköiselle kaupankäynnille, verkkopankille, sähköpostiviestinnälle ja tietokantajärjestelmille. Keyloggerien havaitseminen on haastavaa, koska ne pystyvät toimimaan salaa ja välttävät usein perinteiset virustorjuntaohjelmat. (Solairaj ym. 2016.)

2.2.9 Botnet

Weyns kertoo, että botnetit ovat kokoelmia useista saastuneista laitteista, joilla on yhteys internettiin. Kyseinen haittaohjelma on suunniteltu tarttumaan ja leviämään verkkoon liitettyihin laitteisiin liittääkseen lisää laitteita haltuunsa. Tämän jälkeen, kun haittaohjelma on onnistunut saastuttamaan laitteen, on sille annettava ohjeet, mikä sen tehtävä on ja kenen kanssa se kommunikoi. On olemassa erilaisia rakenteita miten botnet toimii, mutta niistä yleisin on keskitetty ”command and control” eli komento- ja hallintapalvelin. Tästä käytetään usein nimitystä C2-palvelin. Tässä rakenteessa on olemassa yksi ylläpitävä palvelin, jota kutsutaan ”botmasteriksi”. Botmaster hallinnoi tartunnan saaneita laitteita, joita kutsutaan joko zombeiksi tai boteiksi. Botnetillä voidaan tehdä useita askareita, joista esimerkkeinä annetaan kryptovaluutan louhintaa, palvelunestohyökkäyksiä sekä muista haittaohjelmista tuttuja salasanojen sekä pankkitietojen varastamisia. Tiettyjä botnetejä kutsutaan perheiksi, mikäli niissä on tunnistettavia ominaisuuksia. Näitä ominaisuuksia ovat esimerkiksi millaisia hyökkäyksiä ne tekevät ja mihin laitteisiin ne keskittyvät. Eri perheet perustuvat pitkälti toisiinsa ja jakavat suuren osan koodistansa. Yksi tuhoisimmista botnet-perheistä

on Mirai-niminen haittaohjelma, joka ilmestyi 2016. Se yllätti internetin täysin ja on vielä 2025 yksi suurimmista mitatuista palvelunestohyökkäyksistä. Mirai ei kuitenkaan ollut ensimmäinen sen tyyppinen botnet, mutta sen leviämisnopeutta sekä laajuutta ei ollut aiemmin ennen nähty. Tutkijat arvoivat, että se oli levinnyt noin 65 000 laitteeseen ensimmäisen kahdenkymmenen tunnin aikana. Mirai on myös edelleen vuonna 2025 aktiivinen haittaohjelma. (Weyns 2025)

2.2.10 Mobiililaitteiden haittaohjelma

Mobiililaitteiden haittaohjelmat ovat haittaohjelmia, mutta ne ovat erityisesti suunniteltu mobiililaitteisiin kuten älypuhelimiin, tabletteihin sekä älykelloihin. Tämän takia ne ovat keskittyneet Androidiin ja iOSin kaltaisiin käyttöjärjestelmiin sekä niiden sovellusten ja verkkoyhteyksien ominaisuuksiin sekä haavoittuvuuksiin. Mobiilihaittaohjelmien yleistyminen voidaan rinnastaa täysin samaan suuntaan kuin mobiililaitteiden räjähdysmäinen käyttö. Kyberuhkatekijät voivat levittää monin erimenetelmin, mutta toisin kuin työpöytäjärjestelmissä, niissä usein ei ole yhtä vankkaa tietoturvaohjelmistoa. Näin käyttäjät eivät välttämättä tunnista riskejä samalla tavalla epäilyttävän linkin painamisesta tai kauppapaikkojen ulkopuolisten sovellusten asentamisesta. Mobiililaitteisin kohdistuvat haittaohjelmat ovat erityisen vaarallisia, koska ne pystyvät hyödyntämään saastuneen laitteen kannettavuutta. Kyseiset haittaohjelmat voivat napata kaksivaiheisen tunnistautumisen koodit lennosta, jolloin kyberuhkatekijät voivat päästä käsiksi pankkisovelluksiin tai sähköposteihin. Jotkin variaatiot voivat myös kytkeä mikrofonin tai kameran päälle ilman uhrin tietämystä. Androidit ovat yleensä haavoittuvampia kuin iOSit johtuen niiden hajanaisesta päivitysjärjestelmästä sekä avoimesta lähdekoodista. (Mobile Malware n.d.)

2.2.11 Wiper Malware

Wiperit, tai kuten suomeksi kääntyisi pyyhkimet, ovat haittaohjelman alaluokitus, jolla on vain yksi tarkoitus: poistaa käyttäjän tiedot palautettavuuden rajalle. Wipereita käytetään yleensä tuhoamaan tietokoneverkkoja julkisissa ja yksityisissä yrityksissä teollisuudesta viihdesektorille. Uhkatoimijat käyttävät myös wipereita pyyhkimään tunkeutumisesta jälkeen jääneet jäljet, mikä heikentää uhrien reaktiokykyä. Ransomwarella sekä wipereilla on joitakin yhteisiä tekijöitä. Molemmat käyvät massamuistit läpi etsien tiedostoja muutettavaksi tai vahingoitettavaksi sekä molemmat pystyvät tekemään tietojenpalauttamisen mahdottomaksi uhrille. Näillä kahdella on kuitenkin yksi

iso ero: ransomwaret yleensä mahdollistavat tiedostojen palauttamisen uhreille lunnaiden yhteydessä, kun taas wiperit yrittävät suoraan tuhota ne. Tästä syystä wiperit usein toimivat nopeammin ja vaativat vähemmän resursseja, koska niiden ei tarvitse yrittää lukea tiedostoja ollenkaan. Wipereiden tekniikat voidaan jakaa kolmeen periaatteeseen: tiedostojen ylikirjoittamiseen, tiedostojen sisällön ylikirjoittamiseen tai suoraan massamuistin tuhoamiseen. Tiedostojen ylikirjoittaminen ja sisällön ylikirjoittaminen kuulostavat samalta, mutta niiden erot teoriassa ovat suuret. Esimerkiksi ylikirjoittamisessa saatetaan poistaa tiedostojärjestelmän viitteet, jolloin todisteet tiedoston olemassaolostakin poistuu samalla. Tämän palautettavuus on vaikeaa, mutta sisällön ylikirjoittamisen palauttaminen on äärimmäisen vaikeaa, mikäli ylikirjoituksessa on käytetty satunnaisuutta. Jotkin wiperit menevät tämä pidemmälle ja yrittävät tuhota itse levyn eivätkä vain sen tietoja. Tämä tarjoaa uhkatekijälle useita etuja, kuten toiminnan nopeuttamisen ja palauttamisen vaikeuttamisen mahdollisesti tehden siitä mahdottoman. (Iacob & Ionita 2022)

3 Välttelytekniikat

3.1 Syyt välttelytekniikoille

Haittaohjelmat käyttävät välttelytekniikoita ensisijaisesti ohittaakseen tietoturvajärjestelmien havaitsemisen ja analysoinnin. Yksi monista välttelytekniikoiden hyödyistä on sen, että haittaohjelmat toimivat järjestelmässä huomaamatta, mikä pidentää niiden eliniän odotetta ja mahdollista vaikutuksen laajuutta. Tämä salamyhkäisyys on tärkeää haittaohjelmille, joiden tarkoituksena on siirtää arkaluonteisia tietoja tai suorittaa muita vahingollisia toimintoja ilman, että käyttäjä tai turvajärjestelmät saavat siitä hälytystä. Toiseksi nykyaikaiset tietoturvajärjestelmät, kuten virustorjuntaohjelmat, palomuurit ja muut toimet, käyttävät dynaamista analyysiä havaitsemaan ja ymmärtämään haittaohjelmia. Haittaohjelmat käyttävät eri tekniikoita välttääkseen nämä analyysit, jolloin tietoturvajärjestelmien on vaikeampi tunnistaa sekä estää uhkat. Kolmanneksi välttelytekniikat monimutkaistavat haittaohjelmia itsessään, mikä tekevät haittaohjelmien torjunnasta haastavaa, mikä saattaa hidastaa reagointiaikaa ja aiheuttaa suurempia vahinkoja. Tämä myös vaikeuttaa myös tietoturvan-asiantuntijoiden haittaohjelmien takaisinmallinnusta ja analysointia. Kyseiset kyberturvallisuuden ammattilaiset kuitenkin kehittävät jatkuvasti puolustustoimenpiteitä, joka tarkoittaa, että vastauksena näille kyberuhkatoimijoiden on pakko jatkuvasti kehittää uusia ja parantaa nykyisiä välttelytekniikoita sekä strategioitaan. Loppujen lopuksi välttelytekniikat ovat tärkeä

osa nykyajan haittaohjelmia, jotka auttavat niistä ohittamaan turvatoimia ja toteuttaa tavoitteensa. (Afianian ym. 2019.)

3.2 Välttelytekniikoita määriteltynä

3.2.1 Anti-debugging

Anti-debugging-tekniikat viittaavat erinäköisiin ohjelmointitekniikoihin, joiden tarkoituksena on estää tai vaikeuttaa ohjelman analysointia niin sanotussa virheenkorjausympäristössä. Virheenkorjausympäristö on keskeinen osa ohjelmistokehitystä ja Havaitessaan kyseisen ympäristön, ohjelma reagoi siihen muuttamalla käytäntöjään tai suoraan sulkemalla itsensä esimerkkeinä. Näitä tekniikoita käytetään niin myös kyberturvauhkavaikuttajien haittaohjelmissa kuin myös standardeissa, laillisissa sovelluksissa. Hyvänä esimerkkinä tästä ovat erilaiset lisenssisuojatut ohjelmistot sekä pelit, jotka käyttävät näitä ominaisuuksia suojaamaan koodiansa väärinkäytöltä.

Yksi näistä tekniikoista on käyttää Windowsin tarjoamia API-kutsuja debug-tilan tarkistamiseen kuten *IsDebuggerPresent*. Käytännössä tämä funktio tarkistaa Process Enviroment Block -rakenteesta *Being Debugged* -lipun arvon ja näin päättelee, onko se näin debuggauksen alla. Tätä voidaan käyttää myös suoraan rekisteripohjasta ilman tarvetta käyttää API-kutsuja. Kolmas tyyppi on Heap-flagit, jotka ovat yleinen debugging-tekniikka. Vastaavasti neljäs keskeinen lippu *NtGlobalFlag* paljastaa debuggaustilassa asetetut erityiset muistin ja suorituksen tarkkailuliput, kuten aiemmat Heap-flagit. (Anti-Debug: Debug Flags 2022)

Vastaavasti Linuxin puolelta löytyy vastaavia tekniikoita kuin Debug Flagsissä on esitetty Windowsissa. Yksi yleisimmistä näistä on *ptrace()*-kutsun käyttö. Komento *ptrace(PTRACE_TRACEME, 0)* palauttaa negatiivisen arvon jos prosessia jo tarkkaillaan. Koska *ptrace()*-kutsut ovat helposti havaittavissa, minkä takia niiden naamiointi dynaamisella kirjastonlatauksella on yleinen suojakeino. Tällöin itse *ptrace()*-kutsu ei näy ohjelmakoodissa ennen suoritusta. Signaalipohjaiset menetelmät, kuten SIGILL, SIGSEGV ja SIGTRAP perustuvat virheellisiin käskyihin tai muistiviitteisiin. Ohjelma voi esimerkiksi suorittaa laittoman käskyn (kuten *udf*) ja tarkistaa, onko sen oma signaalinkäsittelijä aktivoitu. Jos debugger sieppaa signaalin ennen kuin haittaohjelman signaalinkäsittelijä toimii, tilanne paljastaa debuggerin olevan aktiivinen. Samalla tavalla SIGSEGV:n (muistirikko) ja SIGTRAPin

(debuggerin ansa) käyttäytymisestä voidaan päätellä, onko tilanteessa ulkopuolista tarkkailijaa. (Park ym. 2024)

Prosessimuistin tarkastelu tarjoaa keinon havaita debuggerin manipuloimaa koodia. Haittaohjelma voi esimerkiksi lukea omaa muistiansa ja etsiä siitä INT3-käskyjä, joita debuggerit käyttävät breakpointien asettamiseen. Jos näitä esiintyy paikoissa, joissa niitä ei normaalisti olisi, ohjelma voi reagoida esimerkiksi lopettamalla suorituksensa. (Anti-Debug: Process Memory 2022.) Poikkeuspohjaiset tekniikat sisällyttävät itseensä erillisen ja monipuolisen anti-debugging-kategorian. Haittaohjelma voi aiheuttaa tarkoituksella poikkeuksia kuten INT 3, DIV 0 tai käyttää RaiseException-funktiota, josta se jälkikäteen tarkistaa poikkeuskäsittelijän käyttäytymisen. Mikäli virhettä ei käsitellä ohjelman oman logiikan mukaisesti niin voidaan päätellä, että prosessia tarkkaillaan debuggerin puolesta. (Anti-Debug: Exceptions 2022)

Assemblyä käyttävät menetelmät perustuvat arkkitehtuurikohtaisten konekäskyjen väärinkäyttöön. Haittakoodi voi käyttää laittomia käskyjä kuten UD2 tai rakentaa monimutkaisia hyppyjä, jotka vaikeuttavat disassemblereiden avulla tehtävää analyysiä. (Anti-Debug: Process Assembly 2022.) Viimeinen näistä tekniikoista tarkkailee käyttäjän aktiivisuutta tai sen puuttumista. Jos ohjelma odottaa käyttäjän syötettä eikä sitä kuulu määräajassa, se voi päätellä olevansa virtualisoidussa ympäristössä. Joissain variaatioissa ohjelma voi vaatii esimerkiksi tiettyjä näppäimistö- tai hiiritapahtumia ennen kuin se aloittaa tai jatkaa toimintaansa. (Anti-debug: Direct debugger Interactive 2022)

3.2.2 Anti-Disassembly

Haittaohjelmienkehittäjät pyrkivät aktiivisesti estämään reverse-engineeringin kautta tehtävää analyysiä käyttämällä erinäisiä tekniikoita. Näiden tekniikoiden tarkoitus on vaikeuttaa ja mahdollisesti estää disassembler-ohjelmia kuten IDA Prota ja Ghidraa toimimasta. Yksi yleisimmistä menetelmistä on ohjelmavirran harhautus, *control flow obfuscation*, missä haittaohjelman koodiin sisällytetään ylimääräisiä hyppyjä, looppirakenteita ja dynaamisesti laskettuja hyppykohteita. Nämä estävät dissambleria muodostamasta suoritettavan koodin oikeaa rakennetta. Maininnan arvoisia esimerkkejä tästä on niin kutsut läpinäkymättömät ehdot eli *opaque predicates* ja ohjausvirran tasoittaminen, englanniksi *control-flow flattening*. Läpinäkymättömissä ehdoissa koodi täytetään eh-

tolauseilla, joiden ehto on todellisuudessa aina tosi tai epätosi. Tämä luo ylimääräisiä haaroja tutkittavaksi, jotka ovat ylimääräistä koodia kokonaisuudessaan. Control-flow flatteningissä vastaavasti koodi kehitetään suureksi ympyräksi, josta koodin suoritusjärjestys aukeaa vasta ajon aikaisesti. (Haruyama 2020)

3.2.3 Anti-Forensics

Kyberrikolliset käyttävät erilaisia anti-forensiikkatekniikoita peittääkseen jälkensä ja välttääkseen havaitsemisen. Yksi yleisimmistä on aikamanipulaatio, erityisesti timestomping, jossa haitallisten tiedostojen luontipäivämääriä ja -aikoja muutetaan. Tämä vaikeuttaa tutkijoiden työtä koota yhteen tapahtumien aikajanaa järjestelmän Master File Tablen metadatan perusteella. Lisäksi pimeäverkossa saatavilla olevat Cybercrime-as-a-Service-palvelut tarjoavat työkaluja rikollisten käyttöön. Esimerkiksi Crypting-as-a-Service uudelleenkonfiguroi tunnettuja haittaohjelmia muuttamalla niiden digitaalista allekirjoitusta, jotta ne ohittavat perinteiset virustorjunnat. Pilvipalveluiden yleistyminen on tarjonnut hyökkääjille uuden piilopaikan. Piilottamalla toimintansa pilveen he tekevät verkkoliikenteen seurannasta ja analysoinnista haastavaa. Samoin VPN:ien käyttö vaikeuttaa kyberturvauhkatekijöiden sijainnin ja identiteetin selvittämistä. Uusimmat anti-forensiikkatekniikat hyödyntävät tekoälyä. Tekoäly voi luoda erittäin hienostunutta koodin peittämistä, joka ohittaa edistyneet turvallisuustyökalut kuten EDR-järjestelmät. Lisäksi suuret LLM:t, kuten ChatGPT ja Google Gemini, voivat auttaa kehittämään menetelmiä, jotka sulauttavat haittaohjelmaliikenteen normaaliin verkkoliikenteeseen, tehden poikkeamien havaitsemisesta entistä vaikeampaa tutkijoille. Cortado (2024)

3.2.4 Antivirus & Endpoint Detection Response Evasion

Antivirus ja Endpoint Detection Response, EDR eli suomeksi päätelaitteiden havaitsemis- ja reagointiratkaisut. Nämä järjestelmät vertaavat tiedostoja tunnettuja haittaohjelmia edustaviin binäärisiin "sormenjälkiin", eli allekirjoituksiin. Haittaohjelman tunnistus perustuu siten siihen, löytyykö sen koodista rakenteellisia tai sisällöllisiä yhtäläisyyksiä aikaisemmin tunnistettuihin haittaohjelmanäytteisiin. Kuitenkin kyberturvauhkatekijät kehittävät jatkuvasti uusia tekniikoita näiden suojausten ohittamiseksi. Antiviruksen ja EDR:n toimintamallit voidaan jakaa karkeasti neljään osaan. (Anderson ym. 2018)

Ensimmäinen näistä on Signature-based Evasion Techniques eli suomeksi allekirjoituksiin perustuvat välttelytekniikat pohjautuvat pitkälti vain antiviruksen pohjalle. Allekirjoituksiin perustuvat välttelytekniikat ovat haittaohjelmien käyttämää taktiikkaa, jonka tavoitteena on estää virustorjuntaohjelmia ja muita haittaohjelman tunnistusjärjestelmiä havaitsemasta haitallista koodia. Tämä strategia keskittyy koodin muuttamiseen siten, että se ei enää vastaa aiemmin tunnettujen haittaohjelmien allekirjoituksiin, joita virustorjuntaohjelmat käyttävät tunnistaakseen haittaohjelmia. Virustorjuntaohjelmat luottavat usein allekirjoituksiin perustuvaan tunnistukseen, jossa ohjelma vertaa koodin osia ennalta tunnettuun haittaohjelma-aineistoon. Jos koodi vastaa kyseistä allekirjoitusta, ohjelma merkitsee sen haitalliseksi. Allekirjoituksiin perustuvan tunnistuksen perusperiaate toimii hyvin yksinkertaisissa skenaarioissa, joissa haittaohjelmat ovat staattisia ja ennakoitavissa. Kuitenkin nykypäivän haittaohjelmat hyödyntävät yhä enemmän tekniikoita, jotka estävät allekirjoitusten käytön tehokkuuden. Näihin tekniikoihin kuuluvat muun muassa koodin muokkaaminen, pakkaaminen ja koodin salaaminen. (Anderson ym. 2018)

Haittaohjelmat hyödyntävät usein koodin pakkausta tai salausta (packer ja crypter) piilottaakseen varsinaisen haittakoodinsa Antivirus-järjestelmiltä. Packer-tekniikassa haittakoodi pakataan tai salataan uudeksi suoritettavaksi tiedostoksi, johon sisältyy pieni purkuohjelma. Tämä purkaa alkuperäisen koodin ajonaikaisesti niin, että staattisessa analyysissa nähdään vain pakattu versio. Vastaavasti crypter-tekniikassa haittakoodi salataan ja ohjelmaan lisätään dekryptaus-rutiini, joka purkaa koodin suorituksen yhteydessä. Tuloksena haittaohjelmasta muodostuu jatkuvasti muuntuvia versioita, mikä vaikeuttaa staattista tunnistusta ja pakottaa puolustusjärjestelmät turvautumaan käyttäytymiseen perustuvaan heuristiikkaa, mikä heikentää pelkästään signatuurien pohjautuvia menetelmiä. Haittaohjelmat käyttävät myös koodin obfuskointia eli koodin sisällön ja rakenteen tarkoituksellista mutkistamista. Obfuskointi voi tarkoittaa esimerkiksi lohkojen järjestyksen muuttamista, turhien komentojen lisäämistä, muuttujien nimeämistä sekavaksi tai koodin lohkojen salaamista. Joissakin obfuskointi-tekniikoissa käytetään koodin virtualisointia, jolloin alkuperäinen ohjelma käännetään toteutettavaksi omassa virtuaalikoneessaan. Virtuaalitekniikat tekevät koodin muokkaamattomasta tarkastelusta käytännössä mahdotonta, sillä käännetyssä koodissa ei ole suoraa yhteyttä alkuperäisiin käskyihin. (Malware Detection: Evasion Techniques. 2023.)

Antivirus ja EDR toimivat vastaan prosessin injektioita vastaan. Prosessin injektiotekniikat ovat strategioita, joita haittaohjelmat käyttävät lisätäkseen haitallista koodia laillisiin prosesseihin. Näin

haittaohjelmat voivat välttää havaitsemisen, hyödyntää kohteena olevien prosessien etuoikeuksia ja suorittaa haitallisia toimintojaan luotettavan sovelluksen varjolla. Näistä esimerkkeinä DLL-injektio haittaohjelmissa on prosessin injektiotekniikka, jossa haitallinen koodi injektoi oman Dynamic Link Library (DLL) -kirjastonsa oikeaan, muistissa käynnissä olevaan prosessiin. Tämä mahdollistaa prosessin käyttäytymisen manipuloinnin ja mahdollisesti haitallisten toimintojen suorittamisen herättämättä epäilystä. Tämä tekniikka hyödyntää kohdeprosessin muistitilaa haitallisen DLL-tiedoston lataamiseen, jolloin haittaohjelma saa hallintaansa prosessin toiminnot, tiedot ja resurssit. (Malware Detection: Evasion Techniques. 2023.)

Toinen variaatio injektiosta on koodin injektio. Tässä haitallinen koodi lisätään oikean prosessin muistitilaan muuttaen sen käyttäytymistä haitallisten toimintojen suorittamiseksi. Tämä menetelmä hyödyntää kohdeprosessin muistissa olevia haavoittuvuuksia haitallisen koodin injektoimiseksi. Hyödyntämällä luotetun prosessin kontekstia injektoitu koodi voi kiertää tunnistusmekanismeja, jotka perustuvat allekirjoituspohjaiseen tunnistukseen, mikä tekee tietoturvaratkaisuille vaikeaksi erottaa laillinen prosessi injektoidusta haitallisesta käyttäytymisestä. Seuraava injektiotyyppi on muistin varaaminen. Siinä haittaohjelma lisää haitallisen koodinsa oikean prosessin muistitilaan. Nämä tekniikat sisältävät sen, että haittaohjelma varaa dynaamisesti muistia kohdeprosessin osoiteavaruudesta ja kirjoittaa sitten hyötykuormansa kyseiseen varattuun muistiin. Manipuloimalla luotetun prosessin muistia haittaohjelma voi suorittaa koodiaan laukaisematta perinteisiä allekirjoituspohjaisia tunnistusmenetelmiä. (Malware Detection: Evasion Techniques 2023)

Viimeinen injektion tyyppi on Reflective DLL Injection. Sen avulla DLL injektoidaan kohdeprosessiin suoraan muistista. Toisin kuin perinteiset DLL-injektiomenetelmät, jotka lataavat DLL:n levyltä, reflektiivinen DLL-injektio hyödyntää reflektiivistä ohjelmointia kirjaston lataamiseen muistista. Tästä syystä arvostetaan erityisesti sen kyvystä ohittaa monia perinteisiä tietoturvamekanismeja, mikä tekee siitä edistyksellisten kyberturvauhkatekijöiden suosiman valinnan. (What Is Reflective DLL Injection? How It Works & Examples 2024)

Muut välttelytekniikat, joihin antivirus sekä EDR yrittävät reagoida, on zero-day-haavoittuvuudet. Prakashin (2023) mukaan zero-day-haavoittuvuudet ovat kyberturvallisuuden merkittävä ja monimutkainen uhka. Ne ovat käytännössä tietoturva-aukkoja ohjelmistoissa tai laitteistoissa, jotka

ovat täysin tuntemattomia sekä järjestelmien kehittäjille että tietoturvavalmistajille siinä vaiheessa. Nimi "zero-day" viittaa siihen, että kehittäjillä ei ole ollut aikaa julkaista korjausta tai päivitystä haavoittuvuuden julkitulohetkellä. Zero-day-hyökkäykset hyödyntävät näitä tuntemattomia aukkoja usein korkean profiilin tapauksissa aiheuttaen merkittäviä seurauksia niin yksityiselle kuin julkisellekin sektorille. Niiden seurauksena voi olla tietomurtoja, tietojen varastamista, järjestelmien lamauttamista sekä tuntuvaa taloudellista ja maineellista vahinkoa. Perinteiset, allekirjoituksiin perustuvat tietoturvamenetelmät ovat usein tehottomia zero-day-hyökkäyksiä vastaan, sillä niille ei ole olemassa tunnettuja tunnistusmerkkejä. Tämän vuoksi zero-day-uhkien torjunta edellyttää kehittyneitä strategioita, kuten käyttäytymisperusteista havaitsemista, koneoppimisen hyödyntämistä ja ennakoivia turvallisuuskäytäntöjä. (Prakash 2023)

3.2.5 Network Evasion

Network evasion eli verkon välttelytekniikat ovat kehittyneitä menetelmiä, joita haittaohjelmat ja hyökkääjät käyttävät ohittaakseen verkon suojausmekanismeja, erityisesti tunkeutumisen havaitsemisjärjestelmiä (IDS) ja tunkeutumisen estojärjestelmiä (IPS). Turukmane ja muut (vuosi) esittävät näiden tekniikoiden tarkoituksena on estää haitallisen verkkoliikenteen tunnistaminen ja hälytysten laukaiseminen, tai hämätä turvajärjestelmiä käsittelemään liikenteen sisältöä väärin tai epäjohdonmukaisesti. Yksi sen keskeisistä tekniikoista on Denial-of-Service-hyökkäykset analyysimoottoreita vastaan. Näiden hyökkäysten tavoitteen on pyrkiä ylikuormittamaan IPS:n muistia tai prosessointikykyä. Esimerkiksi Snortin kaltaisten järjestelmien sääntöjen takaisinkelausalgoritmin hidastaminen manipuloimalla syötettyä verkkoliikennettä voi johtaa tunnistamisen epäonnistumiseen, vaikka käytetty kaistanleveys olisi pieni, jopa vain noin 4 kbps. (Turukmane ym. 2024)

Toinen näistä tekniikoista on pakettien pilkkominen, jossa hyökkääjä jakaa haitallisen koodin tai allekirjoituksen pieniin, ei-päällekkäisiin IP-datagrammeihin tai TCP-segmentteihin. Jos IDS/IPS ei kykene täysin kokoamaan näitä fragmentteja tai segmenttejä takaisin alkuperäiseksi sovellussisällöksi, se ei pysty havaitsemaan haitallista sisältöä. Kolmas esimerkki tästä on duplicate insertion eli kaksoiskappaleiden lisäys. Tässä tekniikassa hyökkääjä lisää useita päällekkäisiä tai limittäisiä IP-fragmentteja tai TCP-segmenttejä, jotka hämmentävät IPS:ää. Koska IPS:ltä puuttuu tietoa verkon topologiasta ja uhrin käyttöjärjestelmästä, se ei välttämättä käsittele duplikaatteja samalla tavalla kuin kohdejärjestelmä. Hyökkääjä voi esimerkiksi lisätä segmenttejä pienillä Time-To-Live-arvoilla,

jotta ne poistetaan ennen kuin ne saavuttavat kohteen, mikä estää IPS:ää rekonstruoimasta segmenttejä luotettavasti ja näkemästä samaa materiaalia kuin uhri. (Turukmane ym. 2024)

Myös payload mutation, suomeksi hyötykuorman muuntelu, on tekniikka, missä haitallinen sisältö muokataan semanttisesti samanlaiseksi, mutta syntaktisesti erilaiseksi, jotta se ei vastaa IPS:n allekirjoituksia. Esimerkiksi Microsoft IIS -palvelin ja Apache-verkkopalvelin voivat tulkita kenoviivoja eri tavalla URL-poluissa, mikä antaa hyökkääjälle mahdollisuuden ohittaa tunnistuksen hyödyntämällä näitä eroja. Viimeisenä on Shellcode mutation eli haavoittuvuutta hyödyntävä koodinpätkä, jonka avulla salataan, pakataan tai obfuskoidaan polymorfisiin muotoihin. Tämä tekee siitä vaikeasti tunnistettavan IPS-järjestelmille, jotka perustuvat allekirjoituksiin, ilman purkua tai simuloitua suoritusta. Esimerkiksi "no operation" (nop) -komentojen lisäys tai erilaiset purku- ja dekoodausrutiinit voivat piilottaa koodin todellisen toiminnan. (Turukmane ym. 2024)

3.2.6 Sandbox Evasion

VMRay 2014 esittää, että sandboxien kiertoon on pääsääntöisesti kolme metodia: Sandboxin havaitseminen, sandbox-aukkojen sekä ekosysteemien aukkojen käyttäminen sekä Kontekstipohjaiset välttelytekniikat. (Malware Sandbox Evasion Techniques: All You Need to Know 2024)

Näistä ensimmäinen lähestymistapa tutkii ajoympäristöään mahdollisia eroja sandbox-ympäristön merkkejä. Vanhin näistä tekniikoista on etsiä koneesta virtuaalisation tai hypervisor-jäännöksiä. Nykypäivänä tämä ei ole niin hyödyllinen johtuen vähentyneistä jäännöksistä. Myös tuotannonympäristöt alkavat olemaan virtualisoituja eikä vain tutkijoiden ja haittaohjelma-analyytikkojen käytössä. Toinen tapa on etsiä muita yleisiä nimiä, joita on yhdistetty virtuaalikoneisiin. Nämä voivat sisällyttää tiettyjä tiedostoja, prosesseja, ajureita, järjestelmäntiedostojen rakennetta, Windows ID:n sekä käyttäjänimeä. Myös tarkempi tutkiminen ympäristöön voi paljastaa, onko se analyytikkojen ansa vai oikea kohde. Näitä on monenlaisia, mutta Vmray esittää seuraavia neljää tarkistettavaksi. Ensimmäinen on tarkistaa suoraa laitteiston ominaisuuksia kuten näytön resoluutiota, onko millaisia USB-portteja, 3D-renderöinnin mahdollisuuksia, kuinka monta prosessorin ydintä on käytössä, muistin sekä RAMin koko. Myös pikaviestintäohjelmien sekä sähköpostin puute voivat kertoa, ettei kyseessä oli normaali ympäristö. Viimeisenä käyttöjärjestelmän ja käyttäjän asetukset. Mikäli kone on juuri käynnistetty tai ollut lukuisia päiviä päällä ilman verkkoliikennettä on yksi

merkeistä. Käyttäjän puolesta on puhdas työpöytä, ei ylimääräisiä tiedostoja, ei evästesteitä tai viimeksi avattuja tiedostoja. (Malware Sandbox Evasion Techniques: All You Need to Know 2024)

Sandbox-tekniikan heikkouksia voidaan hyödyntää edistyneemmin haittaohjelmien toiminnassa siten, että ne voivat toimia huomaamatta. Yksi tällainen lähestymistapa on monitoroinnin sokaistaminen, jossa hiekkalaatikon valvontaa heikennetään esimerkiksi poistamalla tai kiertämällä koukkuja. Koukut tarkoittavat valvontapisteitä tai rajapintoja, joiden kautta virtuaaliympäristössä voidaan havainnoida tai muokata koodin suoritusta analyytikkojen haluamissa kohdissa. Koukut voivat olla haasteellisia, sillä haittaohjelmat voivat käyttää esimerkiksi laittomia API-rajapintoja ohittaakseen nämä keinot. Haittaohjelmat voivat hyödyntää näitä haavoittuvuuksia suojatussa hiekkalaatikossa suorittaakseen toimintoja ilman havaitsemista. Tämä lähestymistapa luo uusia haasteita havaita ja torjua haittaohjelmia hiekkalaatikkoympäristöissä. (Malware Sandbox Evasion Techniques: All You Need to Know 2024.)

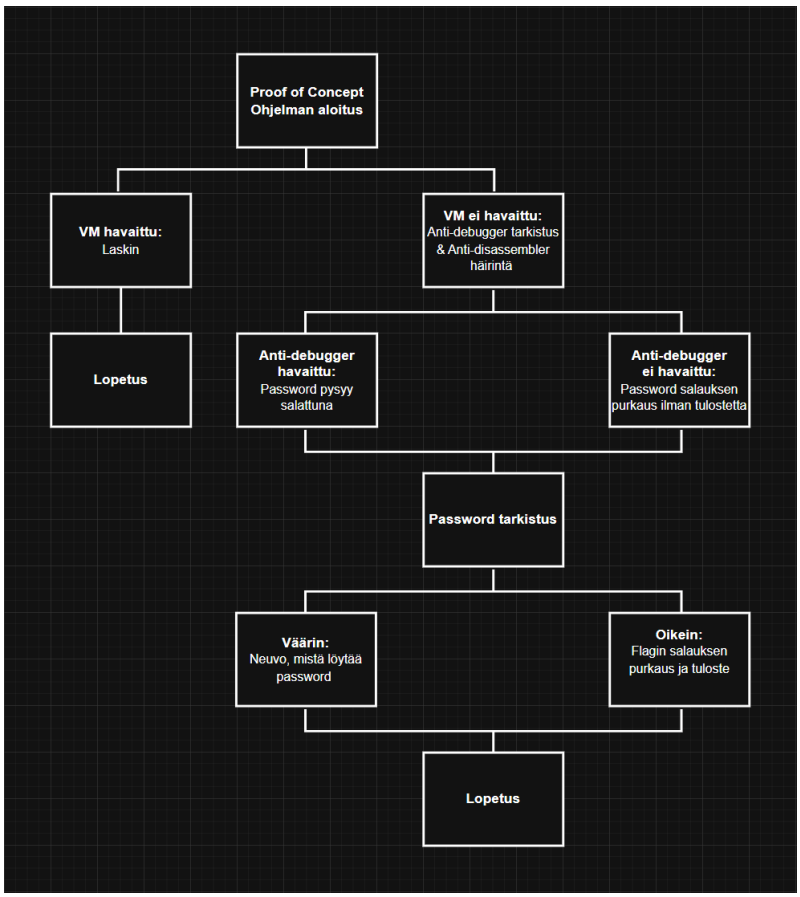
Viimeinen kolmesta yläkategoriasta on kontekstiin perustavat välttelytekniikat. Ne eivät tutki ympäristö tai hae sen heikkouksia, vaan ne ovat suunniteltu viivästyttämään tai lykkäämään haittaohjelman hyökkäyksen suorittamista, kunnes tietty laukaiseva tapahtuma tai olosuhteet täyttyvät. Yksi yleisimmistä tekniikoista on niin kutsutut aikapommit, jotka viivyttävät haittaohjelman ajoa tietyn aikaa, sillä hiekkalaatikot suorittavat näytteitä yleensä vain rajoitetun ajan. Näin haittaohjelma voi nukkua seurannan ohi ja aktivoitua vasta oikeassa ympäristössä. Toinen strategioista on seurata järjestelmätapahtumia, jotka aktivoivat haittaohjelman esimerkiksi tietokoneen sammutus, uudelleenkäynnistyminen tai käyttäjän kirjautumisen yhteydessä. Tätä tekniikkaa käyttävät haittaohjelmat voivat myös vasta aktivoitua huomattuaan käyttäjän vuorovaikutusta. Nämä toimet voivat olla hiiren liikkeitä, näppäimistöllä kirjoittelua tai tiettyjen ohjelmien kuten selaimen, sähköpostin tai Office-dokumenttien availua.

4 Käytännöntoteutus

4.1 Proof-of-Concept-haittaohjelma

4.1.1 Suunnitelma

Keskeisenä osana opinnäytetyötä luotiin Proof-of-Concept-haittaohjelma, joka toimii opettavana esimerkkinä haittaohjelman välttelytekniikoista, ilman haittaohjelmien vaarallisuutta. Sen tarkoituksena on esittää käytännönläheisesti, miten modernit haittaohjelmat voivat tunnistaa analyysiympäristöjä sekä estää käänteisanalyysiä. Kehitysprosessi noudatti vuokaavion mukaista haarautuvaa loogista rakennetta, jossa eri tunnistus- ja suojamekanismit vaikuttavat ohjelman suorituksen kulkuun (kuvio 1). Työn teknisenosuuden tilauksessa käytiin lävitse, että vähintään seuraavat tekniikat ovat sisällytettävä toteutukseen: *”Anti-debugger, Anti-disassembler, Anti-sandbox (Sandbox evasion)*”. Näiden lisäksi työssä on tarkoitus lisätä kevyttä obsfuskaatiota suojaamaan sen sisäistä pelillistämistä.



Kuvio 1. Konseptintodennuksen alkuperäinen suunnitelma

Konseptin tarkoitus on suorittaa ensimmäiset tarkistukset ennekuin käyttäjävuorovaikutusta. Ensimmäinen tarkasteltava elementti on virtuaaliympäristön tunnistus. Mikäli virtuaaliympäristöä tunnistetaan, lopetetaan haittaohjelman puolen ajo ja pyöritetään oma terminaalipohjainen laskin tai Windowsin oma *calc.exe*. Toisen haaran alussa suoritetaan anti-debugging-tarkistusta. Mikäli debuggausta havaitaan, ohjelma pitää salasanan kryptattuna estäen ohjelmallisen salaisuuden vuotamisen dynaamiselle disassembler-tarkastelulle. Staattista analyysiä vastaan on käytetty rakenteellisia anti-disassembler tekniikoita.

Mikäli mitään tarkistusta ei havaita, ohjelman on tarkoitus purkaa kryptattu salasana sisäisesti ilman tulostusta terminaaliin. Tämä tarjoaa aiemman vaiheen välttelytekniikoiden paikkaamisen suorittaneille salasanan käytettäväksi seuraavaan kohtaan, jossa kysytään salasanaa. Salasanan syöttämisen jälkeen ohjelman looginen polku jakautuu viimeisen kerran kahteen. Jos salasana on kirjoitettu väärin tai ei ole annettu ollenkaan, sen on tarkoitus antaa vihje salasanan sijainnista. Oikealla salasanalla ohjelma tulostaa palkinnoksi lipun, millä usein vastaavanlaisia pelillistettyjä tehtäviä merkataan voitetuiksi.

4.1.2 Viimeistelty Proof-of-Concept-ohjelma

Kehitetty *ConceptOfProof.exe*-ohjelma noudattaa yleisesti ottaen kuvio 1 osoittamaa vuokaaviota. Esitetty laskimen käyttö korvattiin täysin vain tulosteella, että virtualisointia havaittu ja suoraan ohjelman sammutus. Kuvion 2 mukainen koodi perustuu Unprotectin Detecting Virtual Environment Process (n.d.) esittämään esimerkkiin, jossa luodaan prosessien tilannekuva ja tarkistetaan virtuaaliympäristölle ominaiset prosessit. Jos prosessilista sisältää jonkin näistä, ohjelma päätetään ennen haitallisten tai hyödyllisten toimintojen suorittamista. Tämä simuloi käytännön tilanteita, missä aktuaalinen haittaohjelma pyrkii estämään ajonsa analyysityökaluilla varustetussa hiekkalaatikoissa.

```cpp
// === VM Detection With Process Name Return ===
bool DetectVirtualEnvironmentProcess(std::wstring& detectedProcess) {
    std::vector<std::wstring> virtualProcesses = {
        L"VMwareService.exe", L"VMwareTray.exe", L"TPAutoConnSvc.exe",
        L"VMtoolsd.exe", L"VMwareuser.exe", L"VBoxService.exe", L"VBoxTray.exe"
    };

    HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (hProcessSnap == INVALID_HANDLE_VALUE) return false;

    PROCESSENTRY32W pe32;
    pe32.dwSize = sizeof(PROCESSENTRY32W);

    if (!Process32FirstW(hProcessSnap, &pe32)) {
        CloseHandle(hProcessSnap);
        return false;
    }

    do {
        for (const auto& proc : virtualProcesses) {
            if (_wcsicmp(pe32.szExeFile, proc.c_str()) == 0) {
                detectedProcess = pe32.szExeFile;
                CloseHandle(hProcessSnap);
                return true;
            }
        }
    } while (Process32NextW(hProcessSnap, &pe32));

    CloseHandle(hProcessSnap);
    return false;
}
```

Kuvio 2. Sandbox evasion-koodi (Detecting Virtual Environment Process n.d., muokattu)

Ohjelman toisessa vaiheessa toteutetaan kolmen eri kohdan anti-debugging-tarkistusta. Ensimmäisenä käytetään *IsDebuggerPresent*-funktiota, jota yleisesti käytetään Windows API-tasoisten debuggerien havaitsemiseen (Interrupts n.d.). Toisena vaiheena on Process Environment Block-rakenteen tunnistaminen lukemalla *BeingDebugged*-kentän rekisteristä, jolla voidaan välttää API-tason suojaukset. Kolmas tarkistus suoritetaan *NtQueryInformationProcess*-funktiolla, joka selvittää, onko debugging-portti avoinna prosessille. Nämä kolme tarkistusmenetelmää yhdessä parantavat ohjelman kykyä havaita debugger-ympäristöjä, edustaen realistisempaa lähestymistapaa haittaohjelman näkökulmasta. Kyseisiä tekniikoita voi miettiä varasuunnitelmana sekä varmistuksena analyysityökaluilla varustettuja sandboxeja vastaan, mikäli ensimmäisen vaiheen ympäristön tutkiminen on onnistettu ohittamaan (kuvio 3).

```
// === Anti-Debugging Techniques ===
bool Detect_IsDebuggerPresent() {
    return IsDebuggerPresent();  // Checks for debugger presence using Windows API
}

bool Detect_PEBBeingDebugged() {
#ifdef _M_IX86
    return ((*(PBYTE)(__readfsdword(0x30) + 2)) != 0);  // Accesses BeingDebugged flag from PEB (x86)
#elif _M_X64
    return ((*(PBYTE)(__readgsqword(0x60) + 2)) != 0);  // Accesses BeingDebugged flag from PEB (x64)
#else
    return false;
#endif
}

typedef NTSTATUS(WINAPI* pNtQueryInformationProcess)(
    HANDLE, ULONG, PVOID, ULONG, PULONG
);

bool Detect_DebugPort() {
    DWORD debugPort = 0;
    pNtQueryInformationProcess NtQueryInfo =
        (pNtQueryInformationProcess)GetProcAddress(GetModuleHandleW(L"ntdll.dll"), "NtQueryInformationProcess");

    if (NtQueryInfo) {
        NtQueryInfo(GetCurrentProcess(), 7, &debugPort, sizeof(DWORD), NULL); // 7 = DebugPort
    }
    return debugPort != 0;
}
```

Kuvio 3. Anti-Debugging-tekniikat

Kolmannessa vaiheessa konseptia aktivoidaan anti-disassembler-tekniikat. Näistä käytössä on kolme eri mekanismia: *INT3-ansa*, joka aiheuttaa keskeytyksen, mikäli ne suoritetaan. *UD2-ansa*, joka on tarkoituksellisesti epätoimiva konekomento ja aiheuttaa virheen, mikäli disassembler-ohjelma yrittää tulkita sen suoritettavaksi. *FakeControlFlow* lisää staattisen analyysin työmäärää luomalla ylimääräisen polun tulkittavaksi, mitä ei todellisuudessa voi suorittaa. Nämä tekniikat tukevat erityisesti staattista suojausta ennen kuin ohjelman toiminta etenee seuraavaan vaiheeseen (kuvio 4).

```
// === Anti-disassembler Technique 1: INT3 Trap (not called)
void DecoyINT3Trap() {
    __asm {
        int 3       // 0xCC – breakpoint instruction
        int 3       // multiple to confuse linear sweep
    }
}

// === Anti-disassembler Technique 2: UD2 illegal instruction (x86/x64)
void UD2Trap() {
#if defined(_MSC_VER)
    __asm {
        _emit 0x0F  // UD2 = 0F 0B
        _emit 0x0B
    }
#endif
}

// === Anti-disassembler Technique 3: Fake jump chain
void FakeControlFlow() {
    int trigger = GetTickCount();
    if (trigger == 0x31337) {
        DecoyINT3Trap(); // Dead path to fake breakpoint trap
        UD2Trap();       // Dead path to illegal instruction
    }
}
```

Kuvio 4. Anti-Disassembler-tekniikat

Ohjelman neljännessä vaiheessa paljastuu salattu teksti, mikäli debuggeria ei ole havaittu. Salattu teksti on niin kutsutusti ”kova koodattu” teksti, joka on pyöritetty XOR-salauksen läpi. Kyseinen salaus puretaan ohjelman ajon aikana, mikäli aiempi debuggerien tarkistus ei löytänyt mitään. Tämä esimerkin käytös muistuttaa oikeaa haittaohjelmaa, joka piilottaa toiminallisuuksiaan. Ensimmäinen salattu teksti antaa salasanan syötettäväksi ohjelman seuraavassa osuudessa. Tämä tarkoittaa, että konseptin ajavan analysoijan on pystyttävä ajamaan ohjelma ilman välttelytekniikoiden aktivoitumista. Vastaavanlainen runtime-suojausmekanismi on tyypillinen nykyaikaisissa haittaohjelmissa, jotka pyrkivät mukautumaan toimintaansa ajoympäristön perusteella (kuvio 5).

```
// === Hidden password hint only for static analysis ===
__declspec(noinline) void RevealPasswordHint_DebugOnly() {
    // This function is never called. Exists only for reverse engineering.
    char encHint[] = {
        'T' ^ 0x3C, 'h' ^ 0x3C, 'a' ^ 0x3C, 't' ^ 0x3C,
        's' ^ 0x3C, 'L' ^ 0x3C, 'a' ^ 0x3C, 'm' ^ 0x3C,
        'e' ^ 0x3C, 0
    };

    for (int i = 0; encHint[i]; ++i)
        encHint[i] ^= 0x3C;

    std::cout << "[Hint] Password: " << encHint << "\n";
}
```

Kuvio 5. XOR-salakoodattu salasana, mikäli debuggeria ei ole havaittu

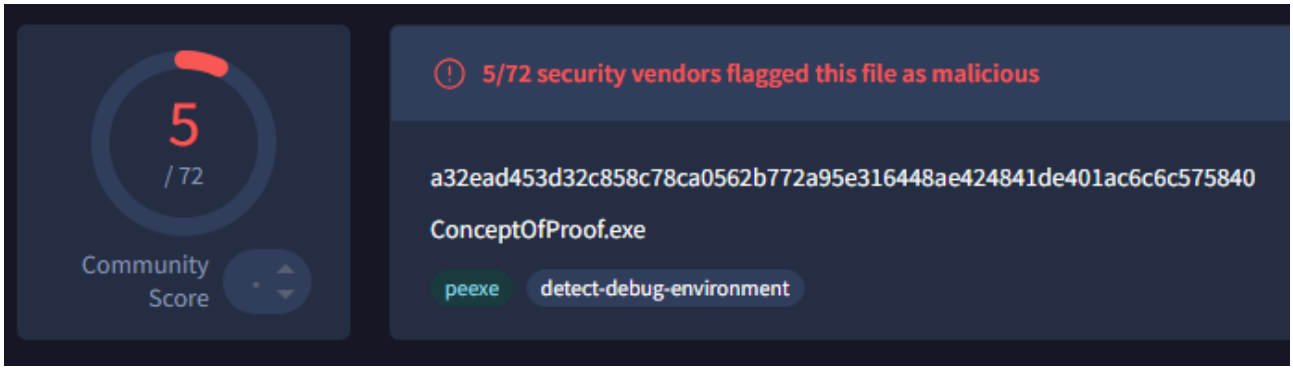
Proof of Concept on myös pelillistetty. Näin PoC:in voi ottaa käyttöön koulutustarkoitukseen haittaohjelmien välttelytekniikoiden tutkimisessa ja analyysityönharjoittelussa. PoC ”voittamiseen” on suoritettava välttelytekniikoiden ohittamiset ja näin paljastaen salaisen tulosteen. Tämä tuloste on suojattu samantyyppisellä obfuskoinnilla, kuten aiempi salasanavihje, mutta siihen on lisätty käänteinen aakkosjärjestys (kuvio 6).

```
// === XOR + Substitution Flag Decryption ===
char substitute(char c) {
    return (c >= 'a' && c <= 'z') ? ('z' - (c - 'a')) : c;  // Reverse alphabet substitution
}

void RevealFlag() {
    char encoded[] = {
        substitute('f') ^ 0x2A, substitute('l') ^ 0x2A, substitute('a') ^ 0x2A, substitute('g') ^ 0x2A,
        '{' ^ 0x2A,
        'G' ^ 0x2A, substitute('o') ^ 0x2A, substitute('o') ^ 0x2A, substitute('d') ^ 0x2A,
        substitute('b') ^ 0x2A, substitute('y') ^ 0x2A, substitute('e') ^ 0x2A,
        'F' ^ 0x2A, substitute('r') ^ 0x2A, substitute('i') ^ 0x2A, substitute('e') ^ 0x2A,
        substitute('n') ^ 0x2A, substitute('d') ^ 0x2A,
        '}' ^ 0x2A,
        '\0'
    };

    for (int i = 0; encoded[i] != '\0'; ++i) {
        encoded[i] ^= 0x2A;
        encoded[i] = substitute(encoded[i]);
    }

    std::cout << "FLAG: " << encoded << "\n";
}
```

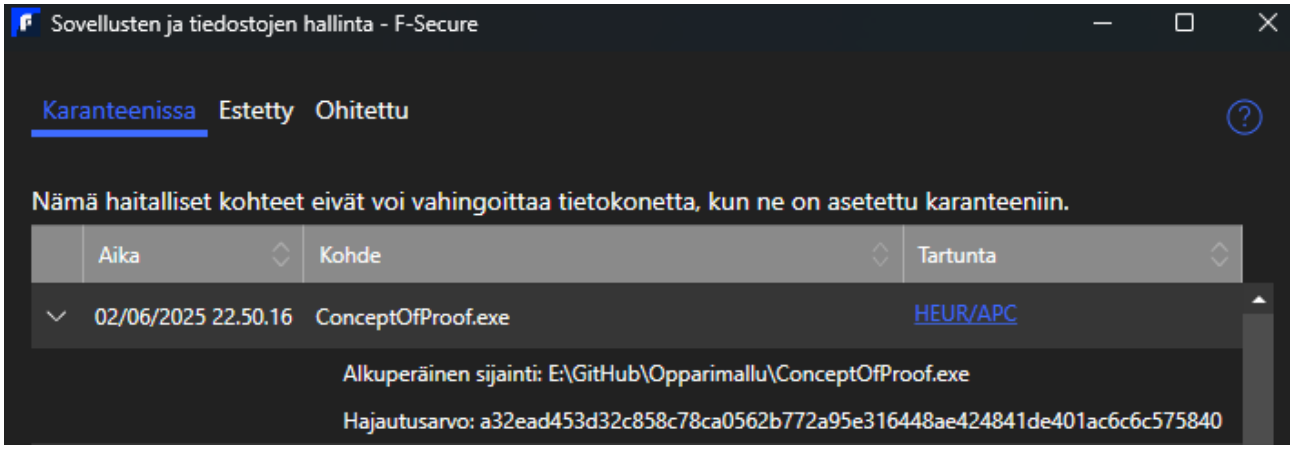
Kuvio 6. XOR ja aakkostonkorvaus suojauksessa onnistuneen puolustuksen lipulle

4.1.3 Ulkopuolinen analyysi

ConceptOfProof.exe-ohjelmaa analysoitiin muutamalla eri turvallisuusalustalla. Näiden tulokset osoittavat, että vaikkei kyseessä ole varsinainen haittaohjelma, sen käyttäytyminen jäljittelee monia oikeiden käyttämiä välttelytekniikoita. Kuten VirusTotal-raportista (kuvio 7) ilmenee, ohjelma tunnistettiin mahdollisesti haitalliseksi viiden virustorjuntamoottorin toimesta. Tämä johtui sen epätavallisista binäärirakenteista ja analyysin välttämiseen viittaavista käyttäytymismalleista. Tämän lisäksi myös F-Securen tietoturvaratkaisu reagoi ohjelman olemassa asettamalla sen karanteeniin (kuvio 8). F-Securen ilmoituksen mukaan heidän heuristinen tunnistuksensa osoittaa ohjelman sisältävän ohjeita haitalliselle toiminnalle tai toimivan samalla tavalla kuin ennestään tunnetut haittaohjelmat (Heuristic n.d.).



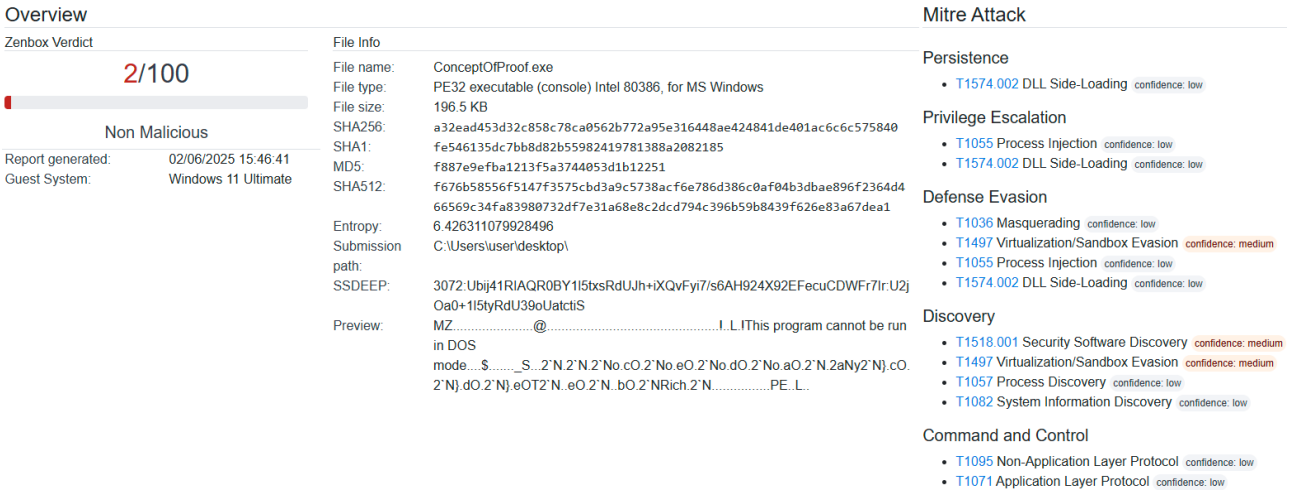
Kuvio 7. VirusTotal tiivistelmä (VirusTotal 2025)



Kuvio 8. F-Securen tarkistus fyysisellä raudalla

VirusTotalin käyttämään Zenboxin raporttiin (kuvio 9) perustuen ohjelma arvioitiin ei-haitalliseksi, mutta sen sisältöä pidettiin silti huomionarvoisena. Ohjelman entropia-arvo, jolla mitataan tiedoston satunnaisuutta on 6,4. Yocamin (2024) mukaan, tämä arvo kuuluu ”Medium Entropy:n” osaan, mikä menee asteikolla 5.0 ja 6.5 välissä (Yocam 2024). Korkeammat arvot viittaavat siihen, että tiedosto on joko pakattu tai sisältää obfuskointia ja encryptausta. Tätä arvoa myös nostaa binäärirakenteesta löytyvät debug-symbolien polut sekä ei-standardin mukaisia Portable Executable sektioita, joiden tarkoitus on vaikeuttaa analyysiä. MITRE ATT&CK on tietoturvayhteisön käyttämä viitekehys, jolla voidaan kuva kyberturvauhkatekijöiden taktiikoita, tekniikoita ja muita menetelmiä. Kuviossa 10 käyttäytymisen yhteydessä ilmoitetaan tekninen tunniste, lyhyt kuvaus ja varmuus siitä, kuinka luotettavasti kyseinen toiminto on analyysissä tunnistettu. Kaikista tekniikoista kaksi oli tunnistettu keskikorkealla-tasolla. Nämä ovat *T1497 Virtualization/Sandbox Evasion* ja

T1518.001 Security Software Discovery. T1497 Virtualization/Sandbox Evasion on mainittu kahdesti, johtuen sen kuuluvan havaitsemisen välttämiseen sekä ympäristön tiedonkeruuseen.



Kuvio 9. Zenboxin yleisarvio sekä Mitre ATT&CKin mukaiset analyysit listattuna

4.2 Ajoympäristö

Opinnäytetyön konseptin todentamisessa on käytetty FlareVM-virtuaaliympäristöä. FlareVM on erityisesti haittaohjelmien analysointiin suunniteltu Windows-virtuaaliympäristö, joka sisältää valmiiksi esiasennetun kokoelman haittaohjelmien tutkimiseen ja analysointiin tarkoitettuja työkaluja. FlareVM:n idea on näin tarjota valmis virtuaaliympäristö, joka on valmiina käyttöön. FlareVM sisältää laajan kokoelman tunnettuja ohjelmia kuten IDA, Ghidra, Wireshark, OllyDbg, Volatility ja Yara. Tämä mahdollistaa haittaohjelmien reverse engineeringiä, verkkoliikenteen tarkastelua, muistin ja prosessien forensiikka-analyysiä sekä haitallisen toiminnan tunnistamista.

Proof-of-Concept suoritettiin FlareVM:ssä, jonka verkkoasetukset oli konfiguroitu ”host-only”-tilassa. Tämä rajaa kaiken ulkoisen verkkoliikenteen estäen mahdollisen vahingollisen koodin tai sen komentokanavayhteyksien vuodon ulkoverkkoon. Tämä on yliampuva varotoimenpide suoritettavaan ohjelmaan nähden, mutta se on hyvä rutiini opetella oikeita haittaohjelmia varten. Käytännössä ConceptOfProof.exe suoritettiin useampaan otteeseen käyttäen FlareVM-ympäristössä terminaalia, x32dbg-debuggeria sekä IDA 9-disassembleria.

Näin voitiin todentaa, miten ohjelma käyttäytyi eri ympäristöissä sekä testata totuttujen anti-debugging-, anti-disassembler- ja sandbox-evasion-tekniikoiden käyttö. Samalla tämä mahdollisti binäärin turvallisen purkamisen ja muokkaamisen, joka mahdollistaa ohjelman sisäisten suojaustekniikoiden ohituksen. Näin myös pystyttiin varmistamaan ohjelman toiminen virtualisoidussa ympäristössä ilman pelkoa isäntäkoneen saastumisesta, jos kyseessä olisi oikea haittaohjelma.

4.3 Suojautuminen

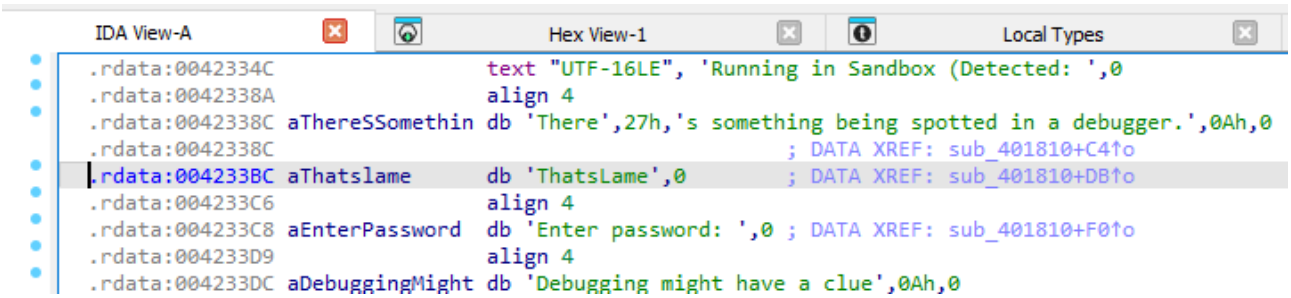
4.3.1 Teoreettinen malliesimerkki

Kun PoC-ohjelma suoritetaan virtualisoidussa ympäristössä, se havaitsee olevansa virtuaalikoneessa tunnistamalla tietyn ohjelman esimerkiksi VMtoolsin ja sulkee itsensä tämän perusteella. Tämä toimintalogiikka voidaan kiertää avaamalla binäärianalyysityökaluun, kuten IDA:n tai x32dbg:en, ja muokaamalla tunnistusmekanismia manuaalisesti. Vaihtoehtoinen suoritusreitti on muokata tunnistettavan ohjelman nimeä siten, että se ei enää vastaa PoC:n käyttämää tunnistuskriteeriä. Esimerkiksi ohjelman nimen muuttaminen muodosta *X.exe* muotoon *Not-X.exe* estää ympäristön tunnistamisen virtuaaliseksi ja näin PoC suorittaa toimintansa normaalisti ilman itsesuojelun aktivoitumista. Seuraavana on debuggerin tunnistuksen ohittaminen. Tässä haetaan, että PoC:n tekijä kävisi patchäämässä kaikki kolme tarkistusta ja palauttamassa arvoksi false. Vaihtoehtoisesti voidaan käyttää x32/64dbg:n sisäänrakennettua ScyllaHide-työkalua, joka piilottaa debuggerin läsnäolon ohjelmalta. Kolmannessa vaiheessa analyysin jatkaminen edellyttää IDA:n käyttöä, sillä ohjelman salasana täytyy etsiä Functions-listasta ja purkaa sieltä manuaalisesti. Neljännessä vaiheessa funktio *FakeControlFlow()* tarkistaa, onko *GetTickCount()*-funktion palauttama arvo täsmälleen 0x31337, mikä ei ole normaalisti mahdollista. Tämä antaa analyytikolle kaksi vaihtoehtoista lähestymistapaa: joko ohjelman binäärikoodia muokataan siten, että vertailukutsu ohitetaan, tai asetetaan keskeytyskohta funktion alkuun ja määritetään rekisteriin haluttu arvo ennen ehtolauseen tarkastamista. Näin päästään kohtaan, missä ohjelma kysyy salasanaa. Tämä pitäisi löytyä IDA:n avulla kaivettuna, joten saadaan ansaittu flag lopputuloksena.

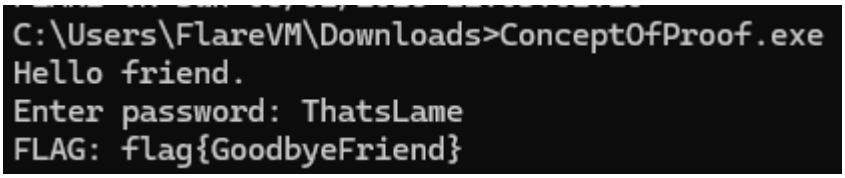
4.3.2 Käytännön toteutus

Teoriatasolla Sandbox evasion -tehtävän pitäisi olla toimiva, mutta käytännössä PoC:n ei tunnista FlareVM-ympäristöä ilman, että siihen asennettaan lisäohjelmia kuten vmtools.exe. Tästä pääsem-

mekin x32dbg:n ja IDAn käyttöön. Tarkemmin IDAta tarkistaen voi löytyä suoraan salasana selkokielellä Enter Passwordin yläpuolella (kuvio 10). Samalla samaisen salasanan löytää myös tarkistamalla ohjelman Strings-osuutta IDAssa. Sieltä se löytyy helpommin noin sadan stringin keskeltä paikalta kuusi. Näillä tiedoilla voidaan ajaa ConceptOfProof.exe terminaalissa, josta saa oikein kirjoittamalla flagin pihalle (kuvio 11).



Kuvio 10. Selkokielinen salasana



Kuvio 11. Pelillistäminen voittava lippu

5 Johtopäätökset

Haittaohjelmien määrä on ollut kasvussa 2020-luvun alusta lähtien vuoteen 2024 verrattuna sekä maailmanlaajuisesti että Suomessa, eikä ole syytä olettaa sen myöskään vähenevän tai pysyvän samassa lukemissa 2020-luvun loppuun mennessä. Osana tätä yleisesti pidetään olevan tekoälyn sekä automaation jatkuva lisääntyminen.

Kyberturvauhkatoimijoiden ja tietoturva-asiantuntijoiden välinen kissa ja hiiri -leikki on jatkuvaa, dynaamista kamppailua, jossa toisen osapuolen teknologiset edistysaskeleet pakottavat vastapuolen mukautumaan päivittämällä olemassa olevia keinoja ja kehittämällä uusia torjuntatekniikoita. Tietoa ja konkreettisia oppeja on kuitenkin paikoitellen vaikea löytää. Tähän vaikuttavat muun mu-

assa eettiset näkökohdat sekä pyrkimys olla paljastamatta puolustustekniikoiden tarkkaa toimintakykyä ja mahdollisia haavoittuvuuksia. Lisäksi tutkimuksen aikana on havaittu, että aihepiirin terminologia ei ole kaikilta osin vakiintunutta, vaan se sisältää tulkinnanvaraisia ja päällekkäisiä käsitteitä, mikä tekee kokonaiskuvan hahmottamisesta entistä haastavampaa. Yhtenä esimerkkinä alkuperäisestä suunnitelmasta voidaan mainita, että *Defense evasion* oli tarkoitus käsitellä omana kokonaisuutenaan. Kuitenkin termin määrittely osoittautui vaihtelevaksi, sillä sen sisältö riippui huomattavasti käytetystä lähteestä. Osa lähteistä halusi käyttää defense evasiota yläkategoriana, mihin olisi kuulunut Antivirus, EDR, Antimonitoring sekä Network Evasion. Toinen lähde taas ehdotti, että Defense Evasion olisi EDR:n alakategoria. Tämä lähteiden ristiriitaisuus johti termin tiputtamiseen työstä. Kuitenkin tähän löytyy alustoja, kuten Mitre Att&ck, Malware Information Sharing Platform, AlienVault OTX, Ciscon Talos Intelligence Center, Unprotect Project ja OWASP.

Työn tavoitteena oli tutkia haittaohjelmia, niiden välttelytekniikoita sekä sitä kautta myös harjaantua välttelytekniikoiden ohittamiseen. Työn aihetta käsitteleviä opinnäytetöitä on saatavilla vain rajallisesti. Ne, joita on löydetty, ovat tyypillisesti hyvin spesifisiä ja keskittyvät kapeasti yhteen osa-alueeseen hyödyntäen usein tekoälyä tai koneoppimismenetelmiä. Tämän työn tavoitteena oli kehittää oma haittaohjelmaa jäljittelevä konsepti, jonka avulla voidaan harjoitella ja testata menetelmiä haittaohjelmien välttelytekniikoiden kiertämiseksi.

Mikäli haittaohjelma otetaan koulutuskäyttöön, on hyvä vaihtaa salasana sekä voittava lippu. On hyvä idea korvata ne satunnaisnimetyillä salasanoilla ja lipuilla henkilökohtaisessa ympäristössä. Näin koko koulutettavalta ryhmältä voidaan todistaa sen suorittaminen sekä estää muiden töiden plagiointi.

Tiedonhaun ongelmana on myös paikoitellen haittaohjelmien ja niiden tekniikoiden monimutkaisuus. Haittaohjelmien luokittelu on haastavaa, koska haittaohjelmien alakategorisointi on usein veteen piirrettyjä viivoja ja koska modernit haittaohjelmat hyväksikäyttävät ja sekoittavat useamman kategorian ominaisuuksia keskenään. Asia pätee myös välttelytekniikoiden puolella, missä ohjelmat usein käyttävät useita eri tekniikoita suojautuakseen havaitsemiselta sekä mahdolliselta analysoinnilta.

6 Pohdinta

Alun perin asetetut tutkimuskysymykset muodostivat työn rungon, ja niiden ydin pysyi samana läpi projektin. Kuitenkin käytännön toteutuksen haasteet ja käytettävissä olleiden resurssien rajoitukset vaikuttivat siihen, miten syvällisesti ja laajasti jokaiseen kysymykseen pystyttiin vastaamaan. Työn tavoitteiden saavuttaminen osoittautui osittain haasteelliseksi suhteutettuna käytettävissä olleeseen aikaan ja tekijän senhetkisiin taitoihin. Erityisesti Proof-of-Concept-toteutuksen kehitysvaiheessa ja testaamisessa ilmeni odottamattomia vaikeuksia, jotka vaikuttivat kokonaistuloksiin. Esimerkiksi C++ -ohjelmointikielellä työskentely oli alkuvaiheessa hidasta, vaikka käytössä oli tekoälyavusteinen ohjelmointituki, kuten Visual Studio Codeen saatavilla oleva Copilot-laajennus. Lisäksi disassembler-ohjelmien, kuten IDA Pron ja x32dbg:n tehokas hyödyntäminen osoittautui puutteelliseksi tässä vaiheessa. Kuitenkin työn edetessä tiedonhaun ja käytännön kokeilujen kautta nämä taidot kehittyivät merkittävästi. Projektin laajuus pyrki laajenemaan mielenkiintoisten teknisten ideoiden ja ratkaisujen kehittyessä koodaustyön edetessä, mikä johti lopulta siihen, että osa toimivista ominaisuuksista karsittiin rajauksen vuoksi, koska haluttiin pitää konseptitoteutus yksinkertaisena.

Teknisen toteutuksen suunnittelu ja kehitystyö osoittautuivat työn antoisimmiksi vaiheiksi. Vaikka alkuperäisestä suunnitelmasta jouduttiin karsimaan osia toteutuksen vuoksi, itse kehitysvaiheessa päästiin soveltamaan ongelmanratkaisukykyä ja luovuutta. Työ osoitti valmiutta rakentaa rajoitetustikin toimiva Proof-of-Concept-toteutus, joka simuloi haittaohjelmien välttelytekniikoita ja tarjoaa pohjan jatkokehitykselle. Teknisestä näkökulmasta voidaan pitää onnistumisena sitä, että työ eteni ongelmista huolimatta: C++-kielen hitaus, työkalujen käyttöhaasteet ja virtualisoitujen ympäristöjen rajoitteet hidastivat prosessia, mutta niitä opittiin tunnistamaan ja kiertämään.

Vaikka lopputulos ei täysin vastannut alkuperäistä suunnitelmaa, prosessin aikana opitut asiat muodostavat arvokasta osaamispohjaa tulevaisuutta varten. Esimerkiksi lähteiden merkitsemisen ja dokumentoinnin systemaattinen ylläpito osoittautui yllättävän vaativaksi, ja sen merkitys opinnäytetyön hallittavuudelle konkretisoitui vasta työn edetessä. Vastaavasti kehitysympäristön, kuten virtualisoinnin, järjestelmällisempi valmistelu olisi voinut ehkäistä joitakin toteutusvaiheessa esiintyneitä haasteita.

Työnrajausta olisi pitänyt tehdä tarkemmin heti alusta lähtien, kuten samoin aikataulutusta. Aikaa kului liian suuresti tietoperustan ja muun keräämisen, vaikkei siitä loppujen lopuksi suurinta osaa tullutkaan käytettyä. Tekstissä tämä varmasti näkyy lähteiden viitteiden niukkuutena. Työn aikana havaittiin myös, että tuotetun tekstin säilyttäminen lähteiden kanssa ja niiden jalostaminen olisi voinut tehostaa kirjoitusprosessia ehkäisten tarvetta kirjoittaa osioita toistuvasti uudelleen tyhjästä. Työnrajauksen takia myös oli haasteita saada tarkennettua tutkimuskysymyksiä.

Menetelmälliset rajoitteet olivat merkittävässä roolissa opinnäytetyön toteutuksessa. PoC-toteutuksen tekninen kompleksisuus suhteessa käytettävissä olevaan aikaan ja tekijän aiempaan osaamistasoon aiheutti haasteita. C++-kielen käyttö tarjosi mahdollisuuden todellisten välttelytekniikoiden lisäämisen suoraan toteutukseen, mutta sen vieraus hidasti työskentelyä verrattuna korkeamman tason kieliin. Lisäksi työkalujen, kuten IDA Pron ja x64dbg:n käyttöön liittyi opettelukynnys, joka vaikutti analyysin syvyyteen erityisesti projektin loppuvaiheessa. Työskentelyprosessin aikana ilmeni myös odottamattomia teknisiä häiriöitä, jotka vaikuttivat työn etenemiseen. Muutamissa tapauksissa virtualisoidun ympäristön ja ohjelmointieditorin yhteistoiminnassa havaittiin toimintahäiriöitä, jotka keskeyttivät kehitystyötä. Esimerkiksi varmuuskopion hajoutuminen kertaalleen saattoi olla seurausta virtualisointialustalla suoritetuista välttelytekniikkakokeiluista. Vaikka näiden ongelmien tarkkoja syitä ei pystytty kattavasti selvittämään, ne korostivat vakaan ja luotettavan kehitysympäristön merkitystä projekteissa.

Luotettavuuden näkökulmasta voidaan todeta, että työssä tuotetut tulokset ovat todennettavissa, mutta niiden yleistettävyys laajempiin haittaohjelmatapauksiin on rajallinen. PoC keskittyi rajattuun joukkoon välttelytekniikoita, eikä se sisältänyt kehittyneitä välttelytekniikoita. Näin ollen tuloksia voidaan pitää käyttökelpoisena demonstraationa, mutta ei kattavana esityksenä nykyaikaisista uhkista.

Työn tuloksia voidaan hyödyntää erityisesti haittaohjelmien analysointiin keskittyvässä opetuksessa ja harjoittelussa, jossa tarvitaan konkreettisia, helposti purettavia esimerkkejä haitallisista toiminnoista. Toteutettu Proof-of-Concept toimii esimerkkiaineistona, jonka avulla voidaan testata ja harjoitella yleisten välttelytekniikoiden, kuten anti-debugging- ja virtualisointitunnistuksen tunnistamista sekä ohittamista. Tämä tekee siitä käyttökelpoisen materiaalin esimerkiksi tietoturvan koulutusohjelmiin tai analyytikoiden perehdyttämiseen. Työn aikana syntyneet ideat ja havainnot,

kuten CPUID-manipulaation sekä näytön resoluution tunnistukset tarjoavat myös jatkokehityksen kannalta käyttökelpoisia suuntia. Näiden ideoiden ympärille voidaan rakentaa laajempia teknisiä testikokonaisuuksia, jotka syventävät ymmärrystä haittaohjelmien käyttäytymislogiikasta. Näin työ ei ainoastaan palvele nykytilanteen havainnollistamista, vaan toimii myös pohjana tulevalle tutkimukselle ja menetelmien kehitykselle.

Lähteet

Advanced Techniques of Malware Evasion and Bypass in the Age of Antivirus. N.d. International Journal of Engineering and Computer Science. Viitattu 28.5.2025. https://ijeci.lgu.edu.pk/index.php/ijeci/article/view/205

Adware vs Spyware: What is the Difference?. N.d. Cisco. Viitattu 24.5.2025. https://www.cisco.com/site/us/en/learn/topics/security/adware-vs-spyware.html

Afianian, A., Baptiste, D., Niksefat, S., & Sadeghiyan, B.. 2019. Malware Dynamic Analysis Evasion Techniques. ACM Computing Surveys, 52,6, 1–28. Viitattu 30.5.2025. https://arxiv.org/abs/2101.08429

Anderson, S., Harkar, A., Filar, B., Evans, D., & Roth, P. 2018. Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning. ArXiv. Viitattu 16.5.2025. https://arxivorg/abs/1801.08917

Anti-Debug: Assembly. N.d. Checkpointin tietokanta antidebuggerista. Viitattu 2.6.2025. https://anti-debug.checkpoint.com/techniques/assembly.html

Anti-Debug: Debug Flag. 2022. Checkpointin tietokanta antidebuggerista. Viitattu 2.6.2025. https://anti-debug.checkpoint.com/techniques/debug-flags.html

Anti-Debug: Direct debugger interaction. 2022. Checkpointin tietokanta antidebuggerista. Viitattu 2.6.2025. https://anti-debug.checkpoint.com/techniques/interactive.html

Anti-Debug: Exceptions. 2022. Checkpointin tietokanta antidebuggerista. Viitattu 2.6.2025. https://anti-debug.checkpoint.com/techniques/exceptions.html

Anti-Debug: Object Handles. 2022. Checkpointin tietokanta antidebuggerista. Viitattu 2.6.2025. https://anti-debug.checkpoint.com/techniques/object-handles.html

Anti-Debug: Process Memory. 2022. Checkpointin tietokanta antidebuggerista. Viitattu 2.6.2025. https://anti-debug.checkpoint.com/techniques/process-memory.html

Anti-Debug: Timing. 2022. Checkpointin tietokanta antidebuggerista. Viitattu 2.6.2025. https://anti-debug.checkpoint.com/techniques/timing.html

Baker, K. N.d. The 12 Most Common Types of Malware. Crowdstrike. Viitattu 21.5.2025. https://www.crowdstrike.com/en-us/cybersecurity-101/malware/types-of-malware/

Bedell, C. 2022. computer worm. TechTarget. Viitattu 19.5.2025. https://www.techtarget.com/searchsecurity/definition/worm

Berardi, D., Giallorenzo, S., Melis, A., Melloni, S., Onori, L. & Prandini, M. 2023. Data Flooding against Ransomware: Concepts and Implementations. Computers & Security, 131, 103295. Viitattu 30.5.2025. https://hal.science/hal-04316302v1/document

Buxton, O. N.d. 15 types of malware: real-world examples and protection tips. Norton. Viitattu 25.5.2025. https://us.norton.com/blog/malware/types-of-malware

Check Point Research Reports Highest Increase of Global Cyber Attacks seen in last two years – a 30% Increase in Q2 2024 Global Cyber Attacks. 2024. Checkpointin blogi. Viitattu 18.5.2025. https://blog.checkpoint.com/research/check-point-research-reports-highest-increase-of-global-cyber-attacks-seen-in-last-two-years-a-30-increase-in-q2-2024-global-cyber-attacks/

Chua, M., & Balachandran, V. 2018. Effectiveness of Android Obfuscation on Evading Anti-malware. ACM Digital Library. Viitattu 22.5.2025. https://doi.org/10.1145/3176258.3176942

Cortado, J. 2024. Anti-Forensics: An Overview of Evasion Tactics. University of Hawai'i – West O'ahu Cyber. Viitattu 4.6.2025. https://westoahu.hawaii.edu/cyber/forensics-weekly-executive-summmaries/anti-forensics-an-overview-of-evasion-tactics/

D'Andrea, A. 2024. Types of Spyware. Keeper Security. Viitattu 17.5.2025. https://www.keepersecurity.com/blog/2024/08/28/types-of-spyware/

Detecting Virtual Environment process. N.d. Unprotect. Viitattu 4.5.2025. https://unprotect.it/technique/detecting-virtual-environment-process/

ENISA Threat Landscape 2024 identifies availability, ransomware, data attacks as key cybersecurity threats. 2024. Industrial Cyber. Viitattu 4.5.2025. https://industrialcyber.co/reports/enisa-threat-landscape-2024-identifies-availability-ransomware-data-attacks-as-key-cybersecurity-threats/

ENISA Threat Landscape 2024. 2024. ENISA. Viitattu 9.5.2025. https://www.enisa.europa.eu/publications/enisa-threat-landscape-2024

Greengard, S. 2025. Pegasus (spyware). Britannica. Viitattu 29.5.2025. https://www.britannica.com/topic/Pegasus-spyware

Haruyama, T. 2020. VB2019 paper: Defeating APT10 compiler-level obfuscations. Virus Bulletin. Viitattu 27.5.2025. https://www.virusbulletin.com/virusbulletin/2020/03/vb2019-paper-defeating-apt10-compiler-level-obfuscations/

Heinemeyer, M. 2019. Insider Analysis of Emotet Malware. Darktrace blog. Viitattu 13.5.2025. https://www.darktrace.com/blog/glimpsing-inside-the-trojan-horse-an-insider-analysis-of-emotet

Heuristic. N.d. F-Secure. Viitattu 2.5.2025. https://www.f-secure.com/v-descs/heuristic.shtml

Iacob, I. & Ionita, M. 2022. The Anatomy of Wiper Malware, Part 1: Common Techniques. Crowdstrike. Viitattu 19.5.2025. https://www.crowdstrike.com/en-us/blog/the-anatomy-of-wiper-malware-part-1

Interrupts. N.d. Unprotect. Viitattu 5.5.2025. https://unprotect.it/snippet/interrupts/128/

Latto, N. 2022. What Is a Computer Virus and How Does It Work?. Avast Academy. Viitattu 24.5.2025. https://www.avast.com/c-computer-virus

Lim, M. & Porter, R. 2022. There Is More Than One Way to Sleep: Dive Deep Into the Implementations of API Hammering by Various Malware Families. Palo Alto Networks. Viitattu 19.5.2025. https://unit42.paloaltonetworks.com/api-hammering-malware-families/

Luoma-aho, M. 2023. Analysis of Modern Malware: Obfuscation Techniques. Opinnäytetyö. YAMK. Jyväskylän Ammattikorkeakoulu, Master's Degree Programme in Information Technology, Cyber Security. Viitattu 19.6.2025. https://www.theseus.fi/handle/10024/798038

Malware Detection and Analysis: Challenges and Research Opportunities. 2021. arXiv. Viitattu 20.5.2025. https://arxiv.org/abs/1811.01190

Malware Detection: Evasion Techniques. 2023. Cyfirma. Viitattu 2.6.2025. https://www.cyfirma.com/research/malware-detection-evasion-techniques/

Malware Sandbox Evasion Techniques: All You Need to Know. 2024. VMRay blogi. Viitattu 14.5.2025. https://www.vmray.com/sandbox-evasion-techniques

Microsoft Digital Defense Report: 600 million cyberattacks per day around the globe. 2024. Microsoft News. Viitattu 3.5.2025. https://news.microsoft.com/en-cee/2024/11/29/microsoft-digital-defense-report-600-million-cyberattacks-per-day-around-the-globe/

Mobile Malware. N.d. Xcitium. Viitattu 29.5.2025. https://www.xcitium.com/knowledge-base/mobile-malware/

Nieminen, P. & Penttilä, T. 2023. Haittaohjelmien analysointi automaattisilla sekä manuaalisilla työkaluilla. Opinnäytetyö, AMK. Jyväskylän Ammattikorkeakoulu, tekniikanala, tieto- ja viestintätekniikantutkinto-ohjelma. Viitattu 13.5.2020. https://www.theseus.fi/bitstream/handle/10024/881034/Nieminen_Penttila.pdf

Park, Y., Choi, S., Choi, Y., Jin, H., Narzia, N. & Park, Y. 2024. A practical approach for finding anti-debugging routines in the Arm-Linux using hardware tracing. Scientific Reports (Nature). Viitattu 22.5.2025. https://www.nature.com/articles/s41598-024-65374-w

Prakash, D. S. 2023. Zero-day vulnerabilities: An in-depth analysis. Bournemouth University / ResearchGate. Viitattu 4.6.2025. https://www.researchgate.net/publication/376517268_Zero-day_Vulnerabilities_An_In-depth_analysis

Rootkits: Definition, Types, Detection, and Protection. 2025. SentinelOne Cybersecurity 101. Viitattu 24.5.2025. https://www.sentinelone.com/cybersecurity-101/cybersecurity/rootkits

Sanjay, B. N., Akash, R. B., Rakshith, D. C., & Hegde, Dr. V. V. 2018. An Approach to Detect File-less Malware and Defend its Evasive mechanisms. ResearchGate. Viitattu 27.5.2025. https://www.researchgate.net/publication/334703011_An_Approach_to_Detect_Fileless_Malware_and_Defend_its_Evasive_mechanisms

Solairaj, A., Prabanand, S., Prathap, C., Vignesh, L. & Mathalairaj, J.. 2016. Keyloggers software detection techniques. ResearchGate. Viitattu 14.5.2025. https://www.researchgate.net/publication/354558970_Keylogger_Detection_and_Prevention

Sudhakar, S. & Kumar, S. 2020. An emerging threat Fileless malware: a survey and research challenges. Cybersecurity, 3,1. Viitattu 19.5.2025. https://www.researchgate.net/publication/338576309_An_emerging_threat_Fileless_malware_a_survey_and_research_challenges

Trojan. N.d. ReversingLabs. Viitattu 28.5.2025. https://www.reversinglabs.com/glossary/trojan

Turukmane, A. V., Khekare, G., Shelke, N., Sakarkar, G. & Buchade, S.. 2024. Evasion Techniques in Cybersecurity: An In-Depth Analysis. ResearchGate. Viitattu 4.6.2025. https://www.researchgate.net/publication/389232618_Evasion_Techniques_in_Cybersecurity_An_In-Depth_Analysis

Veerappan, C. S., Keong, P. L. K., Tang, Z., & Tan, F. 2018. Taxonomy on malware evasion countermeasures techniques. IEEE WF-IoT. Viitattu 15.5.2025. https://doi.org/10.1109/wf-iot.2018.8355202

Vijayan, J. & Lawton, S. 2024. 13 essential enterprise security tools — and 10 nice-to-haves. CSO Online. Viitattu 18.5.2025. https://www.csoonline.com/article/566389/10-essential-enterprise-security-tools-and-11-nice-to-haves.html

VirusTotal: ConceptOfProof.exe Analyysi tulokset. 2025. VirusTotal. Viitattu 1.6.2025. https://www.virustotal.com/gui/file/a32ead453d32c858c78ca0562b772a95e316448ae424841de401ac6c6c575840/detection

Weyns, M. 2025. Exploring the Gorillas in the Malware Jungle. TU Delft Repository. Viitattu 17.5.2025. https://repository.tudelft.6d60-d4a9-4db1-8f90-5a7082f75e98

What Is Reflective DLL Injection? How It Works & Examples. 2024. Twingate. Viitattu 3.5.2025. https://www.twingate.com/blog/glossary/reflective

What is Stalkerware. N.d. Kaspersky. Viitattu 23.5.2025. https://www.kaspersky.com/resource-center/definitions/what-is-stalkerware

Yocam, E. 2024. Entropy and Packing Analysis. GitHub. Viitattu 2.6.2025. https://github.com/ericyoc/win_entropy_packing_poc

Liitteet

Liite 1. ConceptOfProof.cpp

```
#include <windows.h>
#include <winternl.h>
#include <tlhelp32.h>
#include <iostream>
#include <vector>
#include <string>
#include <ctime>
#include <cstdlib>
#include <cctype>

// === Anti-Debugging Techniques ===
bool Detect_IsDebuggerPresent() {
    return IsDebuggerPresent();
}

bool Detect_PEBBeingDebugged() {
#ifdef _M_IX86
    return ((*(PBYTE)(__readfsdword(0x30) + 2)) != 0);
#elif _M_X64
    return ((*(PBYTE)(__readgsqword(0x60) + 2)) != 0);
#else
    return false;
#endif
}

typedef NTSTATUS(WINAPI* pNtQueryInformationProcess)(
    HANDLE, ULONG, PVOID, ULONG, PULONG
);

bool Detect_DebugPort() {
    DWORD debugPort = 0;
    pNtQueryInformationProcess NtQueryInfo =
        (pNtQueryInformationProcess)GetProcAddress(GetModuleHan-
dleW(L"ntdll.dll"), "NtQueryInformationProcess");

    if (NtQueryInfo) {
        NtQueryInfo(GetCurrentProcess(), 7, &debugPort, sizeof(DWORD), NULL);
    }
    return debugPort != 0;
}

// === VM Detection With Process Name Return ===
bool DetectVirtualEnvironmentProcess(std::wstring& detectedProcess) {
    std::vector<std::wstring> virtualProcesses = {
        L"VMwareService.exe", L"VMwareTray.exe", L"TPAutoConnSvc.exe",
        L"VMtoolsd.exe", L"VMwareuser.exe", L"VBoxService.exe", L"VBoxTray.exe"
    };

    HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (hProcessSnap == INVALID_HANDLE_VALUE) return false;

    PROCESSENTRY32W pe32;
    pe32.dwSize = sizeof(PROCESSENTRY32W);

    if (!Process32FirstW(hProcessSnap, &pe32)) {
        CloseHandle(hProcessSnap);
        return false;
```

```
    }

    do {
        for (const auto& proc : virtualProcesses) {
            if (_wcsicmp(pe32.szExeFile, proc.c_str()) == 0) {
                detectedProcess = pe32.szExeFile;
                CloseHandle(hProcessSnap);
                return true;
            }
        }
    } while (Process32NextW(hProcessSnap, &pe32));

    CloseHandle(hProcessSnap);
    return false;
}

// === Anti-disassembler Technique 1: INT3 Trap
void DecoyINT3Trap() {
    __asm {
        int 3
        int 3
    }
}

// === Anti-disassembler Technique 2: UD2 illegal instruction
void UD2Trap() {
#if defined(_MSC_VER)
    __asm {
        _emit 0x0F
        _emit 0x0B
    }
#endif
}

// === Anti-disassembler Technique 3: Fake jump chain
void FakeControlFlow() {
    int trigger = GetTickCount();
    if (trigger == 0x31337) {
        DecoyINT3Trap();
        UD2Trap();
    }
}

// === XOR + Substitution Flag Decryption ===
char substitute(char c) {
    return (c >= 'a' && c <= 'z') ? ('z' - (c - 'a')) : c;
}

void RevealFlag() {
    char encoded[] = {
        substitute('f') ^ 0x2A, substitute('l') ^ 0x2A, substitute('a') ^ 0x2A,
substitute('g') ^ 0x2A,
        '{' ^ 0x2A,
        'G' ^ 0x2A, substitute('o') ^ 0x2A, substitute('o') ^ 0x2A, substi-
tute('d') ^ 0x2A,
        substitute('b') ^ 0x2A, substitute('y') ^ 0x2A, substitute('e') ^ 0x2A,
        'F' ^ 0x2A, substitute('r') ^ 0x2A, substitute('i') ^ 0x2A, substi-
tute('e') ^ 0x2A,
        substitute('n') ^ 0x2A, substitute('d') ^ 0x2A,
        '}' ^ 0x2A,
        '\0'
    };
```

```
    for (int i = 0; encoded[i] != '\0'; ++i) {
        encoded[i] ^= 0x2A;
        encoded[i] = substitute(encoded[i]);
    }

    std::cout << "FLAG: " << encoded << "\n";
}

// === Hidden password hint only for static analysis ===
__declspec(noinline) void RevealPasswordHint_DebugOnly() {
    // This function is never called. Exists only for reverse engineering.
    char encHint[] = {
        'T' ^ 0x3C, 'h' ^ 0x3C, 'a' ^ 0x3C, 't' ^ 0x3C,
        's' ^ 0x3C, 'L' ^ 0x3C, 'a' ^ 0x3C, 'm' ^ 0x3C,
        'e' ^ 0x3C, 0
    };

    for (int i = 0; encHint[i]; ++i)
        encHint[i] ^= 0x3C;

    std::cout << "[Hint] Password: " << encHint << "\n";
}

// === Main Program ===
int main() {
    std::cout << "Hello friend.\n";

    std::wstring vmProcess;
    if (DetectVirtualEnvironmentProcess(vmProcess)) {
        std::wcout << L"Running in Sandbox (Detected: " << vmProcess << L")\n";
        return 1;
    }

    bool debuggerDetected =
        Detect_IsDebuggerPresent() ||
        Detect_PEBBeingDebugged() ||
        Detect_DebugPort();

    if (debuggerDetected) {
        std::cout << "There's something being spotted in a debugger.\n";
    }

    FakeControlFlow();

    std::string correctPassword = "ThatsLame";
    std::string input;
    std::cout << "Enter password: ";
    std::getline(std::cin, input);

    if (input == correctPassword) {
        RevealFlag();
    } else {
        std::cout << "Debugging might have a clue\n";
    }

    return 0;
}
```