Kasperi Kuusipuro

Verkkopelien suojaaminen huijaamiselta

Tradenomi

Tietojenkäsittely

Kevät 2020

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä: Kuusipuro Kasperi

Työn nimi: Verkkopelien suojaaminen huijaamiselta

Tutkintonimike: Tradenomi, tietojenkäsittely

Asiasanat: verkkopelit, peleissä huijaaminen, huijauksilta suojautuminen

Verkkopeleissä huijaaminen on iso ongelma, ja se voi pahimmassa tapauksessa johtaa menetettyihin tuloihin pelinkehittäjille. Mitä suositummaksi peli kasvaa, sitä todennäköisemmin pelissä tullaan huijaamaan. Parhaan pelikokemuksen takaamiseksi on tärkeää yrittää suojata peli mahdollisimman hyvin huijaamiselta. Tässä opinnäytetyössä esitellään yleisiä huijaustekniikoita sekä suojauskeinoja, joilla huijaamista on mahdollista yrittää hankaloittaa tai jopa estää. Valitettavasti huijaamista ei ole mahdollista täysin estää, mutta siitä kannattaa tehdä mahdollisimman vaikeaa.

Jotta peli voidaan suojata mahdollisimman kattavasti, on hyvä ymmärtää eri huijaustekniikoita. Pelin muistia lukemalla ja manipuloimalla huijarin on mahdollista etsiä pelille tärkeitä arvoja ja muokata niitä. Huijari voi esimerkiksi muokata pelaajahahmon elämäpisteitä tehden hahmosta kuolemattoman. Simuloimalla käyttäjän syötettä huijarin on mahdollista luoda botteja pelaamaan peliä huijarin puolesta. Takaisinmallintamisella tarkoitetaan jonkin tuotteen toimintaperiaatteiden selvittämistä. Koska monet huijaukset vaativat hyvää ymmärrystä pelin toiminnasta, takaisinmallintaminen on tärkeä osa huijauksien kehittämisprosessia. Kun huijarilla on hyvä ymmärrys pelin toiminnasta, on hänen mahdollista suorittaa huijauskoodia pelissä esimerkiksi injektoimalla huijauskoodia suoraan pelin suoritettavaksi tai ottamalla pelin käyttämät funktiot omaan käyttöön hookkaamalla nämä. Koska verkkopelien tila täytyy jollain tavalla kommunikoida verkon yli, huijaaminen on myös mahdollista vaikuttamalla pelin verkkoliikenteeseen, esimerkiksi muokkaamalla lähetettäviä verkkopaketteja.

Pelien suojaamiseen on monia eri tekniikoita, mutta näistä ehkä yksi tärkeimmistä on määräävä palvelin eli *authoritative server*. Määräävän palvelimen tapauksessa koko pelisimulaatio suoritetaan palvelimella, jolloin huijarit eivät voi vaikuttaa pelin kulkuun. Valitettavasti määräävä palvelin voi aiheuttaa viivettä peliin, joten se ei välttämättä sovi kaikkiin peleihin. Tärkeitä arvoja, kuten pelaajan elämäpisteitä, on mahdollista myös yrittää salata pelin muistissa, jolloin niiden löytäminen vaikeutuu. Myös takaisinmallintamista voidaan yrittää hankaloittaa, esimerkiksi obfuskoimalla pelin ohjelmakoodia tai estämällä debuggerin toimintaa käyttämällä erilaisia anti-debug-tekniikoita. Pelin tiedostot kannattaa myös tarkastaa muokkaamisen varalta, esimerkiksi laskemalla tiedostoista tiiviste, joka voidaan tarkastaa palvelimella. Pelin verkkoliikenne voidaan salata esimerkiksi käyttämällä RSA- ja AES-salausalgoritmeja, jolloin sen analysoiminen on huomattavasti vaikeampaa. Koska huijareiden on mahdollista muokata peliä omalla tietokoneellaan ja siten ohittaa huijaamista estäviä mekanismeja, kannattaa myös palvelimelle lisätä huijauksien tunnistamista. Palvelimella voidaan esimerkiksi analysoida pelaajien liikkeitä tai toimintoja ja löytää niistä poikkeamia, jotka voivat olla merkkejä bottien käytöstä. Monet kaupalliset huijauksenesto-ohjelmistot lisäksi monitoroivat käyttöjärjestelmän toimintaa esimerkiksi analysoimalla auki olevia prosesseja ja etsimällä huijausohjelmia niiden muistista.

Abstract

Author: Kuusipuro Kasperi

Title of the Publication: Protecting Online Games Against Cheating

Degree Title: Bachelor of Business Information Technology

Keywords: online games, cheating in online games, protecting games against cheating

Cheating in online games is a big problem, and in the worst case it can lead to lost revenue for the game developers. Cheating attempts will only grow the more popular the game becomes. To guarantee the best possible experience for the players, it is necessary to protect the game as well as possible. This Bachelor's thesis will introduce the most common cheating techniques and ways to protect games form cheating. Unfortunately cheating can never be fully prevented, but it can still be made as hard as possible.

To be able to fully protect the game, it is good to have some familiarity of all the different cheating techniques and programs cheaters utilize. By manipulating the game's memory, cheaters can find important values and change them. Cheaters can, for example, find and change the player character's hit points and make their character invincible. By simulating user input, cheaters can create bots to play the game for them. Reverse engineering means taking something apart and figuring how it works. Because many cheats require in-depth knowledge of the game's inner workings, reverse engineering is a big part of developing new cheats. Having good knowledge of the game will enable cheaters to take control of it by either injecting their own code into the process or by hooking existing functions. Cheating is also possible by targeting the game's network protocol because online games must somehow communicate their state over the internet.

There are many ways to protect online games, but perhaps the most important one is the authoritative server model. Authoritative server means doing the whole game simulation on the server side, so there is no possibility of anyone manipulating important values. Unfortunately, the authoritative server can cause some extra latency, so it might not be a good fit for all games. Important values can still be protected, for example by encrypting them in the memory so they cannot be found so easily. Reverse engineering can also be made more difficult, for example by obfuscating code or by preventing debuggers using anti-debug techniques. The game's files can be checked for modifications, for example, by calculating a hash that can be checked on the server. Network traffic can also be encrypted, for example, by using RSA and AES encryption, so analyzing it will be considerably harder. Because cheaters can modify the game as much as they want on their own computer, all client-side cheating prevention mechanisms can be bypassed. Adding some cheat detection on the server-side is also a good idea. The player's movements and actions can be analyzed on the server to find patterns, which can be an indication of bot usage. Many commercial anti-cheat systems also monitor the operating system by scanning open processes for known cheats.

Sisällys

Termiluettelo

AES	*Advanced Encryption Standard* on lohkosalausmenetelmä, jota voidaan käyttää esimerkiksi verkkoliikenteen salaamiseen.
Assembly	Konekielen symbolinen esitystapa. Jokaisella prosessoriarkkitehtuurilla on oma assembly-kielensä.
AutoIt	Skriptikieli, jota voidaan käyttää Windowsin käyttöliittymän automatisointiin. AutoItin avulla voidaan esimerkiksi luoda botteja.
Cheat Engine	Avoimen lähdekoodin muistiskanneri, joka on suunniteltu etsimään arvoja pelien muistista.
Decompiler	Työkalu, jolla voidaan kääntää ohjelma takaisin lähdekoodiksi. Decompilerin käyttö on usein mahdollista vain tavukoodiksi kääntyvissä ohjelmointikielissä.
Disassembler	Työkalu, joka esittää konekielen assembly muodossa. Käytetään ohjelmien analysointiin, jos alkuperäistä lähdekoodia ei ole saatavilla.
ESP hack	ESP- eli *Extra Sensory Perception* huijaus paljastaa huijarille pelistä tietoa, jota ei normaalisti olisi saatavilla, kuten esimerkiksi piilossa olevia vihollisia. Esimerkiksi *wallhack* ja *map-hack* ovat ESP-huijauksia.
Hookkaus	Eng. *Hooking* on tekniikka, jolla ohjelman käyttämien funktioiden suoritus voidaan kaapata, jolloin huijarin on mahdollista esimerkiksi suorittaa omaa koodia ohjelman sisällä.
IAT	*Import Address Table* on Windows-ohjelman osa, jossa on ohjelman käyttämien ulkopuolisten funktioiden osoitteita. IAT:n sisältämiä funktioiden osoitteita voidaan päällekirjoittaa, jolloin funktiokutsut voidaan uudelleenohjata.
IDAPro	Erittäin voimakas disassembler-työkalu, jota voidaan käyttää ohjelmien takaisinmallintamiseen.

ILSpy	Decompiler työkalu, jolla voidaan kääntää .NET-ympäristön ohjelmia takaisin lähdekoodiksi.
.NET	Microsoftin kehittämä ohjelmien kehitykseen tarkoitettu ympäristö. C#-kieli on osa .NET-ympäristöä.
Obfuskaatio	Koodin tarkoitusperän piilottaminen siten, että ohjelman suoritus pysyy ennallaan. Vaikeuttaa ohjelman analysointia.
RSA	Julkisen salausavaimen salausmenetelmä, jota voidaan käyttää esimerkiksi verkkoliikenteen salaamiseen.
SDB	*Signature Based Detection* on usein virustorjuntaohjelmien käyttämä menetelmä, jossa ohjelmien muistista etsitään tunnisteita haittaohjelmista. Voidaan kuitenkin käyttää myös huijausohjelmien tunnistamiseen.
Tavukoodi	Jotkin ohjelmointikielet kääntyvät konekielen sijaan tavukoodiksi, jota usein suoritetaan erillisellä virtuaalikoneella. Tavukoodissa on usein konekieltä enemmän informaatiota, joten tavukoodiksi kääntyviä ohjelmia voi olla yksinkertaisempaa analysoida.
Vtable	Virtuaaliset luokat C++-ohjelmointikielessä sisältävät vtablen, jossa on listattu kaikki luokan käyttämien virtuaalisten funktioiden osoitteet. Vtablen sisältämiä osoitteita on mahdollista päällekirjoittaa, jolloin funktiokutsut on mahdollista uudelleenohjata.
Wallhack	Huijaus, jossa vihollisia on mahdollista nähdä seinien läpi. Yleinen esimerkiksi ensimmäisen persoonan ammuntapeleissä.
Wireshark	Avoimen lähdekoodin ohjelma verkkopakettien kaappaamiseen ja analysointiin.
x64dbg	Avoimen lähdekoodin assemblytason debuggeri Windowsille, joka mahdollistaa sekä 64- että 32-bittisten ohjelmien analysoinnin
XOR	XOR eli ”eksklusiivinen tai” on matemaattinen operaatio, jota voidaan hyödyntää yksinkertaisessa salauksessa.

1 Johdanto

Huijaaminen verkkopeleissä on iso ongelma. Irdeto-tietoturvafirman vuonna 2018 toteuttamassa kyselyssä, jossa haastateltiin lähes kymmentätuhatta pelaajaa, yli 60 % vastanneista ilmoitti huijaamisen johtaneen negatiiviseen pelikokemukseen. Pahimmillaan tämä voi johtaa menetettyihin tuloihin pelinkehittäjille. Kyselyyn vastanneista 77 % ilmoitti mahdollisesti lopettavan pelin pelaamisen, jos he kokevat muiden pelaajien huijaavan. Lisäksi 48 % vastanneista ilmoitti myös ostavansa vähemmän pelinsisäisiä ostoksia huijaamisen seurauksena. [1.]

Tässä opinnäytetyössä käydään läpi erilaisia huijaustekniikoita verkkopeleissä ja kuinka niiltä voi suojautua. Luvussa 2 esitellään erilaisia huijaustyyppejä ja niiden luokittelua. Koska peleissä voi huijata eri tavoin, pelin suojaaminen voi olla vaikeaa, jos ei tunne kaikkia huijaustekniikoita. Luvussa 3 käydään läpi yleisimpiä huijaustekniikoita sekä tutustutaan muutamiin yleisiin ohjelmiin, joita usein käytetään huijaamisen yhteydessä. Lopuksi luvussa 4 esitellään keinoja, joilla pelinkehittäjät voivat hankaloittaa, tai jopa estää, peleissä huijaamista. Työssä keskitytään lähinnä Windows-ympäristöön, sillä suurin osa peleistä toimii siinä, mutta lähes kaikkia tekniikoita voidaan soveltaa muissakin ympäristöissä.

Halusin tutkia peleissä huijaamista, sillä vaikka internet on täynnä esimerkkejä, kuinka peleissä voi huijata, tietoa pelien suojaamisesta on vaikea löytää. Lisäksi monilla keskustelupalstoilla tuntuu vallitsevan mentaliteetti, että huijaamisen estäminen on täysin turhaa, sillä joku kuitenkin onnistuu huijaamaan pelissä. Halusinkin siksi kerätä mahdollisimman kattavasti eri huijauskeinoja ja suojausmenetelmiä yhteen, jotta esimerkiksi aloittelevien pelinkehittäjien olisi helpompi tutustua pelien suojaamiseen. Pelinkehittäjäguru Matthew Pritchard on luonut verkkopeleissä huijaamiselle yhdeksän sääntöä, joista kaksi ensimmäistä ovat: Mitä tahansa rakennatkaan, joku tulee huijaamaan siinä, sekä huijausyritykset tulevat lisääntymään, mitä suositummaksi peli kasvaa [2]. Huijaamisen estäminen kannattaakin ottaa tosissaan, ja vaikka peleissä huijaamista ei voikaan ikinä estää kokonaan, ei se tarkoita, että siitä ei voisi tehdä mahdollisimman hankalaa.

Huijaaminen on usein pelien loppukäyttäjän lisenssisopimuksen (EULA) vastaista ja hyvin todennäköisesti johtaa pelioikeuden menettämiseen. Muutamat peliyhtiöt ovat myös haastaneet huijauksien kehittäjiä oikeuteen [3]. Tässä opinnäytetyössä ei esitellä valmiita huijauksia yhtäkään oikeaa peliä vastaan, ja vaikka opinnäytetyö esittelee monia huijaustekniikoita, ei se kannusta yrittämään kyseisiä tekniikoita yhteenkään peliin.

2 Peleissä huijaaminen

Peleissä huijaamiseen on luotu useita eri määritelmiä. Yan & Randell määrittelevät huijaamisen olevan mitä tahansa, mitä pelaaja tekee saadakseen edun kanssapelaajiinsa nähden, tai saavuttaakseen päämäärän, jos kyseinen etu tai päämäärä on pelin sääntöjen vastainen tai pelioperaattorin harkinnan mukaan sellainen, jota ei ole tarkoitus saavuttaa [4]. Huijaaminen voi keskittyä eri osiin peleissä, joten myös huijauksien luokitteluun on monia eri tapoja. Webb ja Soh jakavat huijaukset neljään tasoon: pelitasoon (*game level*), ohjelmatasoon (*application level*), protokollatasoon (*protocol level*) sekä infrastruktuuritasoon (*infrastructure level*) [5].

Pelitason huijaukset tapahtuvat täysin pelin sisällä, ja ne koostuvat pelinsisäisistä bugeista sekä oikean rahan transaktioista. Pelinsisäiset bugit ovat virheitä pelin koodissa tai suunnittelussa, eivätkä ne siksi vaadi minkäänlaisia huijaustekniikoita toimiakseen. Pelaaja voi esimerkiksi tehdä tietyn sarjan toimintoja ja siten monistaa pelinsisäisiä esineitä. Pelinsisäisten bugien estämiseen ei ole mitään erityisiä tekniikoita. Kattava testausprosessi auttaa löytämään bugit ennen pelin julkaisua, ja uudet bugit voi helposti korjata pelin päivityksissä. Oikean rahan transaktioissa pelaaja voi ostaa esimerkiksi pelinsisäisiä esineitä tai valuuttaa kolmannelta osapuolelta. Oikean rahan transaktioihin liittyy läheisesti ns. *gold-farming*, jossa usein halpatyövoimaa käytetään pelinsisäisen valuutan hankkimiseen myytäväksi pelaajille. [5.]

Ohjelmatason huijaukset keskittyvät pelin muokkaamiseen, joko suoraan tai välillisesti huijausohjelman avulla. Niihin kuuluvat esimerkiksi informaation paljastuminen sekä botit ja refleksienparantajat. Informaation paljastumiseen luokitellaan huijaukset, joissa huijari saa käsiinsä tietoa, johon normaalisti hänellä ei ole pääsyä. Esimerkkinä tästä on ns. ESP- eli *Extra Sensory Perception* huijaukset, jotka manipuloivat peliä näyttämään piilotettua dataa. ESP-huijauksiin kuuluvat esimerkiksi strategiapeleissä *map-hackit*, jotka paljastavat normaalisti piilossa olevia vihollisen joukkoja. Botit ovat ohjelmia, jotka pelaavat peliä pelaajan puolesta ja ovat yleisiä etenkin MMO-peleissä. Botit ovat yleensä suunniteltu automatisoimaan jokin tietty osa pelistä, kuten vihollisten toistuva tappaminen pelaajahahmon kehittämiseksi. Refleksienparantajiin kuuluvat esimerkiksi ns. *aimbotit*, jotka parantavat pelaajan tähtäämistä ammuntapelissä, usein yli-inhimilliselle tasolle. Myös ns. *speedhack* voidaan luokitella ohjelmatason huijaukseksi. *Speedhack* mahdollistaa pelin suorittamisnopeuden muuttamisen, mikä voi johtaa siihen, että huijausta käyttävä pelaaja on esimerkiksi muita pelaajia huomattavasti nopeampi. [6, s. 189.] [5.]

Protokollatason huijaukset keskittyvät verkkoliikenteen manipulointiin, esimerkiksi muokkaamalla verkkopakettien sisältöä tai lisäämällä tai poistamalla lähetettäviä paketteja [5]. Esimerkiksi *Ultima Online* -pelissä oli mahdollista purkaa toisten pelaajien rakennuksia, koska pelipalvelin ei tarkistanut, keneltä purkukomento tuli. Huijarin oli mahdollista lähettää muokattu verkkopaketti ja purkaa mikä tahansa pelin rakennus. Koska Ultima Onlinessa oli rajoitettu mahdollisten rakennusten määrää, rakennuksia pystyi myös myymään oikeaa rahaa vastaan. [7.]

Infrastruktuuritason huijaukset kohdistuvat pelin ulkopuolelle esimerkiksi muokkaamalla ajureita tai verkkoinfrastruktuuria. Grafiikka-ajureita muokkaamalla huijari voi esimerkiksi tehdä seinistä läpinäkyviä, jolloin on mahdollista nähdä vihollisia niiden läpi. Tämä on ns. *wallhack*, joka kuuluu ESP-huijauksiin ja on yleinen esimerkiksi ensimmäisen persoonan ammuntapeleissä. Huijari voi myös luoda välityspalvelimen pelin ja pelipalvelimen välille muokkaamaan tai lukemaan verkkopakettien sisältöä. Tämä mahdollistaa esimerkiksi refleksienparantajien luomisen pelin ulkopuolelle, missä pelaajan ampuessa välityspalvelin lisää muutaman verkkopaketin, jotka kääntävät pelaajan sopivasti kohti vastustajaa ennen ampumiskomentoa. [5.]

3 Huijaustekniikat

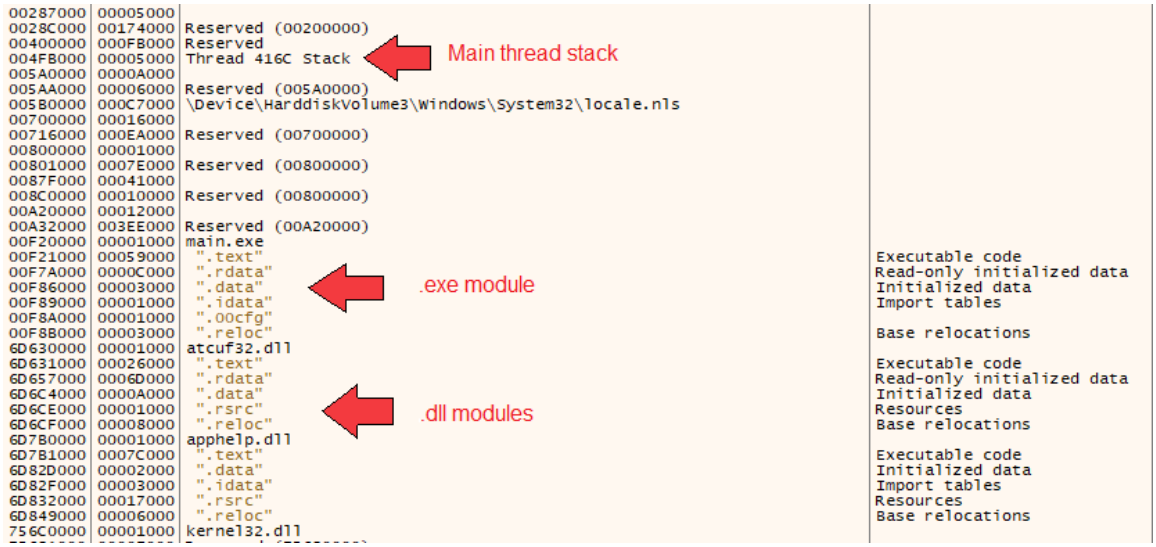
3.1 Muistin luku ja manipulointi

Muistin luku ja manipulointi on yksinkertainen, mutta todella voimakas, tekniikka peleissä huijaamiseen. Muistin luku ja manipulointi mahdollistaa koko pelin tilan tarkastelun ja tarvittaessa myös sen muokkaamisen. Huijari voi esimerkiksi löytää pelaajan elämäpisteitä vastaavan arvon muistista ja muokata sen haluamakseen, tehden pelaajahahmostaan kuolemattoman.

3.1.1 Windows-ohjelman rakenne muistissa

Kun ohjelma käynnistetään, Windows varaa prosessille oman virtuaalisen muistiblokin. Tämä muisti on yhtenäistä, ja jokaisella tavulla on oma muistiosoitteensa, joka yleensä ilmoitetaan heksadesimaalina, esimerkiksi *0xDEADBEEF*. Vaikka virtuaalinen muisti on yhtenäinen, alla oleva fyysinen muisti ei yleensä ole. 32-bittisillä laitteilla prosessin virtuaalisen muistin teoreettinen koko on 4 GB (2^{32} tavua), josta 2 GB on varattu Windowsille itselleen. 64-bittisillä virtuaalisen muistin teoreettinen koko on 16 EB (2^{64} tavua), mutta maksimi on käytännössä 128 TB. [8.]

Ohjelman muisti jakautuu eri moduuleihin, joita ovat ohjelman oma moduuli (.exe-moduuli) sekä ohjelman käyttämät dynaamiset kirjastot (.dll-moduulit). Moduulit itsessään jakautuvat eri segmentteihin, joista tärkeimpiä ovat text- tai code-segmentti, datasegmentti ja bss-segmentti. Text-segmentti sisältää moduulin suoritettavan ohjelmakoodin. Moduulin globaalit ja staattiset muuttujat sijaitsevat yleensä data- tai bss-segmenteissä. Datasegmentissä ovat alustetut muuttujat ja bss-segmentissä taas alustamattomat muuttujat. Muita tärkeitä muistin osia ovat stackit ja heapit. Jokaisella ohjelman säikeellä on oma stack, joka sisältää säikeen funktioiden kutsujärjestyksen, parametrit ja kaikki lokaalit muuttujat. Heapit sisältävät dynaamista muistia, jota esimerkiksi C++-kielen *new*-sanalla luodut objektit käyttävät. Heapeja voi olla useampia eri puolilla ohjelman muistia, ja prosessi voi luoda ja tuhota niitä ajon aikana. Kuvassa 1 on esimerkki, josta nähdään, kuinka eri osat sijoittuvat erään prosessin muistiin. [9, s. 99–102.]



Kuva 1. Esimerkki prosessin muistin rakenteesta

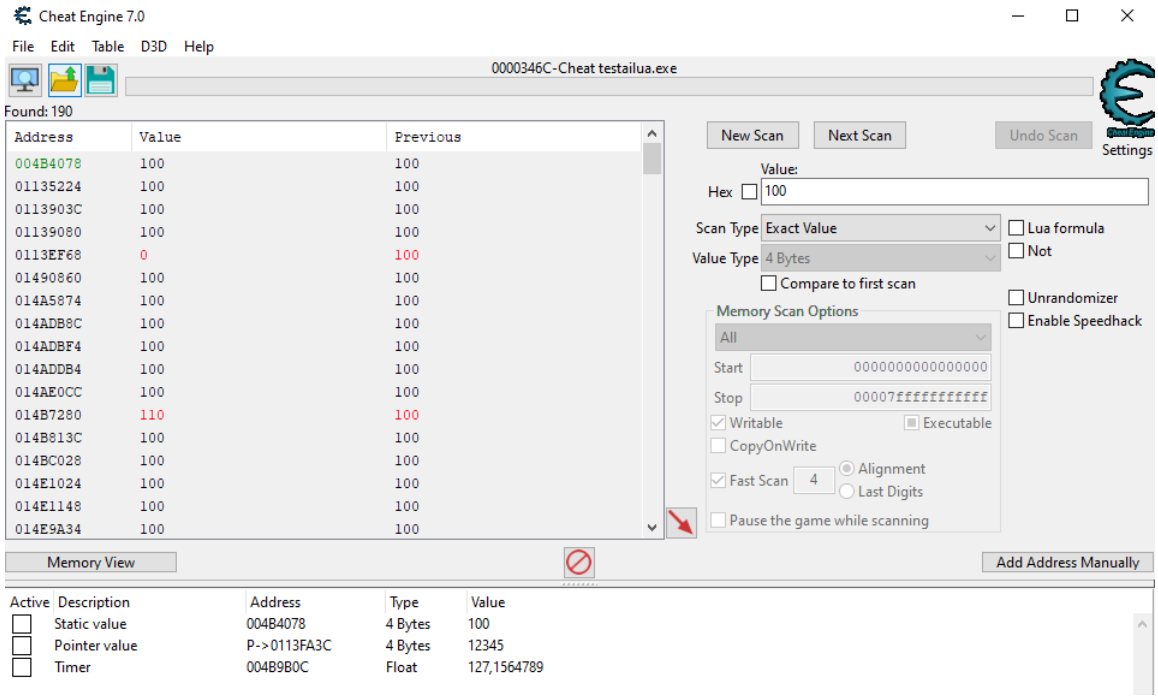
Moderneissa käyttöjärjestelmissä, kuten uusissa Windowseissa, on usein käytössä ASLR eli *Address Space Layout Randomization*, joka lataa ohjelman moduulit satunnaisiin muistiosoitteisiin. Tämä vaikeuttaa huomattavasti pelien muistin lukemista, sillä aikaisemmin löydetty arvo muistissa voi olla täysin eri osoitteessa pelin seuraavalla käynnistyskerralla. Muistiosoitteet eivät kuitenkaan välttämättä muutu yksittäisten moduulien sisällä, joten pysyvä muistiosoite on silti mahdollista löytää. [6, s. 128.]

3.1.2 Cheat Engine

Cheat Engine on avoimen lähdekoodin muistiskanneri, joka on tarkoitettu erityisesti pelien muistin manipulointiin. Muistiskannerin lisäksi Cheat Engine tarjoaa debuggaus- ja takaisinmallinnustyökaluja, speedhackin sekä LUA-skriptaustuen. Cheat Engine mahdollistaa myös ns. trainerien luomisen. Trainereissa voidaan yhdistää muistista löydettyihin arvoihin näppäinpainalluksia, joilla voi esimerkiksi jäädyttää kyseisen arvon. Cheat Enginen luomat trainerit voi myös eriyttää kokonaan itsenäisiksi ohjelmiksi. [10.]

Cheat Enginen muistiskanneri mahdollistaa useiden erilaisten arvojen, esimerkiksi kokonaislukujen, liukulukujen tai yksittäisten tavujen hakemisen muistista. Skannaus toimii valitsemalla halutut parametrit ja painamalla ”Next Scan”-painiketta. Kuvassa 2 on esimerkki hausta, jossa etsittiin arvoa 100 ohjelman muistista. Koska pelit ovat todella isoja ohjelmia ja niiden muistissa on todella

paljon eri arvoja, ensimmäinen skannaus tuottaa usein huomattavan määrän tuloksia. Cheat Engine voi tämän jälkeen tehdä useampia uusia skannauksia ja verrata niiden tuloksia aikaisempiin tuloksiin. Esimerkiksi etsittäessä pelaajan elämäpisteitä voidaan aluksi etsiä tiettyä arvoa pelin muistista. Kun ensimmäinen skannaus on valmis, elämäpisteitä voidaan pudottaa pelissä ja tehdä seuraava skannaus, jossa haetaan joko uutta elämäpisteiden arvoa tai esimerkiksi pienentynyttä arvoa. Skannausta voidaan toistaa niin pitkään, kunnes tuloslistassa on vain muutama arvo, joista on mahdollista päätellä etsityn arvon muistiosoite. [6, s. 5–7.]



Kuva 2. Cheat Engine-käyttöliittymä

Muistiskannauksen tuloslistassa olevat arvot voivat olla joko staattisia (vihreät arvot) tai dynaamisia (mustat arvot). Staattisten muistosoitteiden löytäminen on tärkeää, koska ne pysyvät samana, vaikka peli käynnistettäisiin uudelleen. Dynaamiset muistiosoitteet taas yleensä muuttuvat pelin käynnistyskertojen välillä. Valitettavasti aina ei ole mahdollista löytää staattista osoitetta haettavalle arvolle, mutta Cheat Engine tarjoaa tähän *pointer scan* -vaihtoehdon, jolla on mahdollista löytää staattinen osoite, joka osoittaa haluttuun dynaamiseen arvoon. *Pointer scan* toimii etsimällä pointer- eli osoitinketjuja, jotka alkavat staattisesta osoitteesta ja päättyvät haluttuun muistiosoitteeseen. *Pointer scan* perustuu siihen, että käyttääkseen dynaamista muistia ohjelman on viitattava siihen jossain staattisessa muistissa, joten löydetty osoitinketju käyttäytyy kuin staattinen muistiosoite ja säilyy pelin käynnistyessä uudelleen. [6, s. 11–12.]

Kun staattinen muistiosoite tai osoitinketju on löydetty, Cheat Engine mahdollistaa niiden tallentamisen seuraavaa kertaa varten. Kuvan 2 alalaidassa on esimerkiksi tallennettu kolme eri muistiosoitetta, joista yksi on osoitinketju. Talteen otettuja muistiosoitteita voidaan esimerkiksi jäädyttää, jolloin muistiosoitteessa oleva arvo pysyy samana, tai arvoa voi muokata haluamakseen.

3.1.3 Muistin luku ja muokkaus käyttäen Windowsin ohjelmointirajapintaa

Jos kyseessä on isompi huijausohjelma, kuten esimerkiksi itse tehty botti, muistin manipulaatio voidaan tehdä käyttämällä Windowsin ohjelmointirajapinnan tarjoamia funktioita. Yksinkertaisimmat muistin lukuun ja kirjoittamiseen tarkoitetut funktiot ovat *ReadProcessMemory* ja *WriteProcessMemory*. Molemmat funktiot vaativat parametrikseen *handlen* kohdeprosessiin sekä halutun muistiosoitteen. *Handle* on mahdollista saada käyttämällä *OpenProcess*-funktiota. Jotta muistinmanipulointifunktiot toimisivat oikein, tulee prosessia avatessa pyytää *PROCESS_VM_READ* ja *PROCESS_VM_WRITE*-oikeuksia, jolloin huijausohjelma voi vapaasti manipuloida kohdeprosessin muistia. [6, s. 120–123.]

Kuvassa 3 on esimerkki näiden funktioiden käytöstä. Esimerkkiohjelma lukee joka sekunti elämäpisteiden arvon muistista. Jos tämä arvo putoaa alle 90:n, kirjoittaa ohjelma arvon 100 vanhan arvon tilalle. Kohteena oleva muistiosoite on esimerkin tapauksessa etsitty etukäteen käyttämällä Cheat Engine ohjelmaa. On kuitenkin täysin mahdollista luoda kokonaan oma muistiskanneri käyttämällä *ReadProcessMemory*- ja *WriteProcessMemory*-funktioita, jolloin ei olisi tarvetta ulkopuoliselle muistiskannerille.

```
while (true)
{
    DWORD health;
    ReadProcessMemory(processHandle, (LPCVOID)memoryAddress, &health, sizeof(DWORD), NULL);

    if (health < 90)
    {
        health = 100;
        WriteProcessMemory(processHandle, (LPVOID)memoryAddress, &health, sizeof(DWORD), NULL);
    }

    Sleep(1000);
}
```

Kuva 3. Esimerkki muistin manipuloinnista käyttäen Windowsin ohjelmointirajapintaa

Joskus *ReadProcessMemory*- ja *WriteProcessMemory*-funktioiden käyttö voi epäonnistua, jos kohteena oleva muistiosoite sijaitsee suojatussa muistissa. Tietyt alueet muistissa voivat olla esimerkiksi vain luettavissa, jolloin niihin kirjoittaminen ei ole mahdollista. Muistin suojausta voi kuitenkin muuttaa käyttämällä Windowsin ohjelmointirajapinnan *VirtualProtectEx*-funktiota. Manipuloinnin kohteena olevan muistiosoitteelle voi esimerkiksi asettaa *PAGE_EXECUTE_READWRITE*-suojauksen, jolloin sieltä lukeminen ja sinne kirjoittaminen on mahdollista. Kun muistin manipulointi on suoritettu, olisi hyvä myös palauttaa muistin alkuperäinen suojaustaso, jolloin voidaan välttyä virheiltä kohteena olevan ohjelman suorituksessa. [6, s. 124–127.]

3.2 Syötteen simulointi

Käyttäjän syötettä simuloimalla huijari voi väärentää näppäinpainalluksia ja hiiren liikkeitä, mikä mahdollistaa pelien hallitsemisen. Syötteen simulointi voi yksinkertaisimmillaan olla pieniä makroja, jotka simuloivat muutamia näppäinpainalluksia, tai laajimmillaan kokonaisia botteja, jotka voivat pelata peliä huijarin puolesta. Syötteen simulointiin voi käyttää yksinkertaisia ohjelmia, mutta Windowsin ohjelmointirajapinta tarjoaa tähän myös muutamia funktioita.

3.2.1 Syötteen simulointi AutoIt-skriptikielellä

AutoIt on yksinkertainen, mutta tehokas skriptikieli, joka mahdollistaa Windows-käyttöliittymän automatisoinnin. AutoItin avulla on mahdollista simuloida näppäinpainalluksia ja hiiren liikkeitä sekä manipuloida ikkunoita. Skriptit on myös mahdollista kääntää omiksi ohjelmikseen. Kuvassa 4 on esimerkkinä yksinkertainen skripti, joka lähettää ”abc”-merkkijonon notepad-tekstieditoriin. [11.]

```
1
2   $hwnd = WinGetHandle("[CLASS:Notepad]")
3
4   WinActivate($hwnd)
5
6   Send('abc')
7
```

Kuva 4. Esimerkki yksinkertaisesta AutoIt skriptistä

AutoItin kanssa on myös mahdollista käyttää muiden tekemiä lisäosia. Peleissä huijaamiseen hyödyllisiä ovat esimerkiksi erilaiset kuvantunnistuskirjastot, joiden avulla voi esimerkiksi tunnistaa

pelin käyttöliittymän osia. Kuvantunnistuksella botit voivat lisäksi tunnistaa haluttuja kohteita, mikä helpottaa etenkin, jos kohteet ovat liikkeessä. [9, s. 18.]

3.2.2 Syötteen simulointi Windowsin ohjelmointirajapinnan kautta

Yksinkertaisin Windowsin ohjelmointirajapinnan funktio syötteen simulointiin on *SendInput*-funktio, jolla on mahdollista simuloida sekä näppäinpainalluksia että hiiren liikkeitä. Funktio ottaa parametreikseen syötteiden määrän, osoittimen syötteisiin ja syötteen koon. *SendInput*-funktio lähettää injektoidut näppäinpainallukset ja hiiren liikkeet suoraan senhetkiseen aktiiviseen ikkunaan, mikä tarkoittaa, että kohteena olevan pelin tulee aina olla aktiivisena. [12.]

Kuvassa 5 on esimerkki funktiosta, joka lähettää näppäinpainalluksen käyttämällä *SendInput*-funktiota. Itse syöte määritellään *INPUT*-rakenteessa, joka voi olla tyyppiä *INPUT_MOUSE*, jos kyseessä on hiiren syöte, tai *INPUT_KEYBOARD,* kun kyseessä on näppäimistön syöte kuten esimerkissä. Simuloidessa näppäimistön syötettä tarvitaan virtuaalinen näppäinkoodi, joka on muotoa *VK_SPACE*, sekä erityinen *scan code*, joka riippuu näppäimistöstä. Näppäinkoodia vastaava *scan code* on mahdollista hakea käyttämällä *MapVirtualKey*-funktiota. Näppäinpainalluksen tyyppi määritetään binäärimuuttujassa *dwFlags*. Kun mitään erityistä ei ole määritelty, syöte vastaa alas painettua näppäintä. Jotta koko näppäinpainallus voidaan simuloida, on myös lähetettävä toinen syöte, jossa näppäinpainalluksen tyypiksi on määritelty *KEYEVENTF_KEYUP*, joka vastaa ylös päästettyä näppäintä. [13.]

```
void SendKeypressViaSendInput(WORD key)
{
    WORD scanCode = MapVirtualKeyA(key, MAPVK_VK_TO_VSC_EX);

    INPUT input = {0};
    input.type = INPUT_KEYBOARD;
    input.ki.wVk = key;
    input.ki.wScan = scanCode;
    input.ki.dwFlags = 0;

    if ((scanCode >> 8) == 0xE0) // Extended key
        input.ki.dwFlags |= KEYEVENTF_EXTENDEDKEY;

    SendInput(1, &input, sizeof(input));

    Sleep(100);

    input.ki.dwFlags |= KEYEVENTF_KEYUP;
    SendInput(1, &input, sizeof(input));
}
```

Kuva 5. Esimerkki näppäinpainalluksen lähettämisestä *SendInput*-funktiolla

Toinen syötteen manipuloinnin mahdollistava funktio on *SendMessage*, jonka avulla voi lähettää viestejä toisiin ohjelmiin. Se eroaa *SendInput*-funktiosta lähettämällä syötteen suoraan kohdeikkunalle. Tämä on etenkin hyödyllistä peleissä huijaamisessa, sillä se mahdollistaa pelin suorittamisen taustalla. *SendMessage* ottaa parametreikseen kohdeikkunan, lähetettävän viestin tyypin sekä kaksi viestille spesifistä parametria. [14.]

Kuvassa 6 on esimerkki *SendMessage*-funktion käytöstä näppäinpainalluksen lähettämiseen kohdeikkunaan. Kokonaisen näppäinpainalluksen lähettäminen vaatii kaksi tai kolme kutsua *SendMessage*-funktioon. Kutsujen määrä riippuu siitä, syöttääkö näppäin tekstiä, kuten esimerkiksi A-näppäin. Painettaessa näppäin alas lähetetään *WM_KEYDOWN*-viesti, jonka viestikohtaiset parametrit ovat virtuaalinen näppäinkoodi ja binäärimuuttuja, joka määrittää näppäintä vastaavan *scan coden* sekä muita näppäinpainallusta määrittäviä asioita. Kun näppäin nostetaan ylös, lähetetään *WM_KEYUP*-viesti, joka ottaa samat parametrit kuin näppäintä alas painettaessa, mutta binäärimuuttuja eroaa hieman. Jos näppäin syöttää tekstiä, näiden viestien välissä tulee lähettää myös *WM_CHAR*-viesti, jonka ensimmäisenä viestikohtaisena parametrina on virtuaalista näppäinkoodia vastaava merkki. [6, s. 213–214.]

```
void SendKeypressViaSendMessage(HWND window, WORD key)
{
    WORD scanCode = MapVirtualKeyA(key, MAPVK_VK_TO_VSC_EX);
    char character = MapVirtualKeyA(key, MAPVK_VK_TO_CHAR);

    LPARAM flags = 0;
    flags |= ((scanCode & 0xFF) << 16);

    if ((scanCode >> 8) == 0xE0) // Extended key
        flags |= 0x01000000;

    SendMessageA(window, WM_KEYDOWN, key, flags);

    if (character != 0)
    {
        flags |= 0x40000000;
        SendMessageA(window, WM_CHAR, character, flags);
    }

    flags |= 0xC0000000;
    SendMessageA(window, WM_KEYUP, key, flags);
}
```

Kuva 6. Esimerkki näppäinpainalluksen lähettämisestä käyttäen *SendMessage*-funktiota

3.3 Pelien takaisinmallinnus

Takaisinmallintamisella tarkoitetaan yleensä jonkin tuotteen toimintaperiaatteen selvittämistä. Pelien tapauksessa tämä käytännössä tarkoittaa ohjelman analysoimista erilaisten takaisinmallinnukseen tarkoitettujen työkalujen avulla, sillä usein pelin alkuperäinen lähdekoodi ei ole huijarin saatavilla. Ohjelmien takaisinmallinnus jaetaan usein staattiseen analyysiin, jossa ohjelmaa tutkitaan ilman sen suorittamista, sekä dynaamiseen analyysiin, jossa ohjelmaa analysoidaan sen ajon aikana. Takaisinmallintaminen ei kuitenkaan itsessään ole huijaamista, mutta se on tärkeä osa huijauksien kehitysprosessia. Usein huijaukset vaativat hyvää ymmärrystä pelin toiminnasta toimiakseen oikein, minkä takia iso osa huijausten kehitykseen käytetystä ajasta on takaisinmallintamista. [6.] [15, s. 112.]

3.3.1 Ohjelman rakenne

Pelien ohjelmoinnissa käytetään yleensä jotakin korkean tason ohjelmointikieltä. Tietokone ymmärtää kuitenkin vain konekieltä, joten pelit on käännettävä konekielelle, jotta niiden suorittaminen olisi mahdollista. Konekieli koostuu sarjoista komentoja, jotka ovat käytännössä pelkästään nollia ja ykkösiä. Lisäksi jokaisella prosessoriarkkitehtuurilla on vielä oma konekielensä. Jotta konekieltä olisi helpompi ymmärtää, esitetään se usein assembly-kielenä. Assembly-kieli, jota kutsutaan myös symboliseksi konekieleksi, on vain konekielen tekstimuotoinen esitystapa, jossa jokaista komentoa vastaa tietty merkkijono. [16, s. 10–11.]

Eri ohjelmointikielillä on erilaisia ratkaisuja siihen, kuinka kääntäminen konekieleksi tapahtuu. C++ on todella yleinen kieli pelinkehityksessä, ja se kääntyy suoraan konekielelle. Tähän käytetään erillistä kääntäjäohjelmaa, joka ottaa kirjoitetun lähdekoodin, tarkastaa sen virheiden varalta ja kääntää sen kohdealustan konekielelle. C# on toinen yleinen ohjelmointikieli, jota esimerkiksi Unity-pelimoottori käyttää skriptaukseen. C# on osa Microsoftin .NET-ympäristöä, joten se ei käänny suoraan konekieleksi. Sen sijaan C# kääntyy ensiksi tavukoodiksi (*CIL, eli Common Intermediate Language*), jota sitten erillinen virtuaalikone suorittaa. Tämä tekee C#-ohjelmakoodista hieman hitaampaa suorittaa kuin suoraan konekielelle käännettävät kielet, mutta CIL-tavukoodi on paljon helppolukuisempaa, koska se sisältää esimerkiksi muuttujien ja tyyppien nimiä. Siksi se on myös mahdollista kääntää takaisin ohjelmakoodiksi, mikä ei ole täysin mahdollista suoraan konekielelle kääntyvissä kielissä. [16, s. 34–37.]

3.3.2 Staattinen analysointi

Staattinen analysointi, tai offline-analysointi, tarkoittaa ohjelman analysoimista ilman sen suoritusta, käyttämällä erilaisia decompiler- tai disassembler-työkaluja. Decompiler on työkalu, joka kääntää ohjelman takaisin lähdekoodiksi. Tämä on usein mahdollista vain tavukoodiksi kääntyvillä kielillä, kuten C#-kielellä, mutta se tekee ohjelman toimintaperiaatteiden selvittämisestä todella helppoa, sillä takaisinmallintajan on mahdollista lukea selkeää ohjelmakoodia. Usein kuitenkin ohjelma on tehty käyttäen suoraan konekielelle kääntyvää ohjelmointikieltä, kuten C++-kieltä. Tällöin on turvauduttava disassembler-työkaluun, joka kääntää konekielen assembly-koodiksi. Koska assembly-kielet ovat todella alhaisen tason kieliä, vie niiden analysointi huomattavasti enemmän aikaa kuin tavallisen ohjelmakoodin. [16.]

ILSpy on decompiler-työkalu, joka kääntää .NET-ohjelmatiedostoja takaisin C# lähdekoodiksi [17]. Esimerkiksi Unity-pelimoottorissa on mahdollista käyttää C#-kieltä pelilogiikan ohjelmointiin, joten käyttämällä ILSpy-työkalua on siis mahdollista analysoida joidenkin Unity-pelien lähdekoodia. Unity-pelien skriptit löytyvät yleensä *Assembly-CSharp.dll*-kirjastosta. Kuvassa 7 vasemmalla on ILSpyn tuottama käännös eräästä skriptistä sekä oikealla alkuperäinen lähdekoodi. Käännös vastaa toiminnallisuudeltaan täysin alkuperäistä lähdekoodia, vaikka näyttääkin hieman erilaiselta.

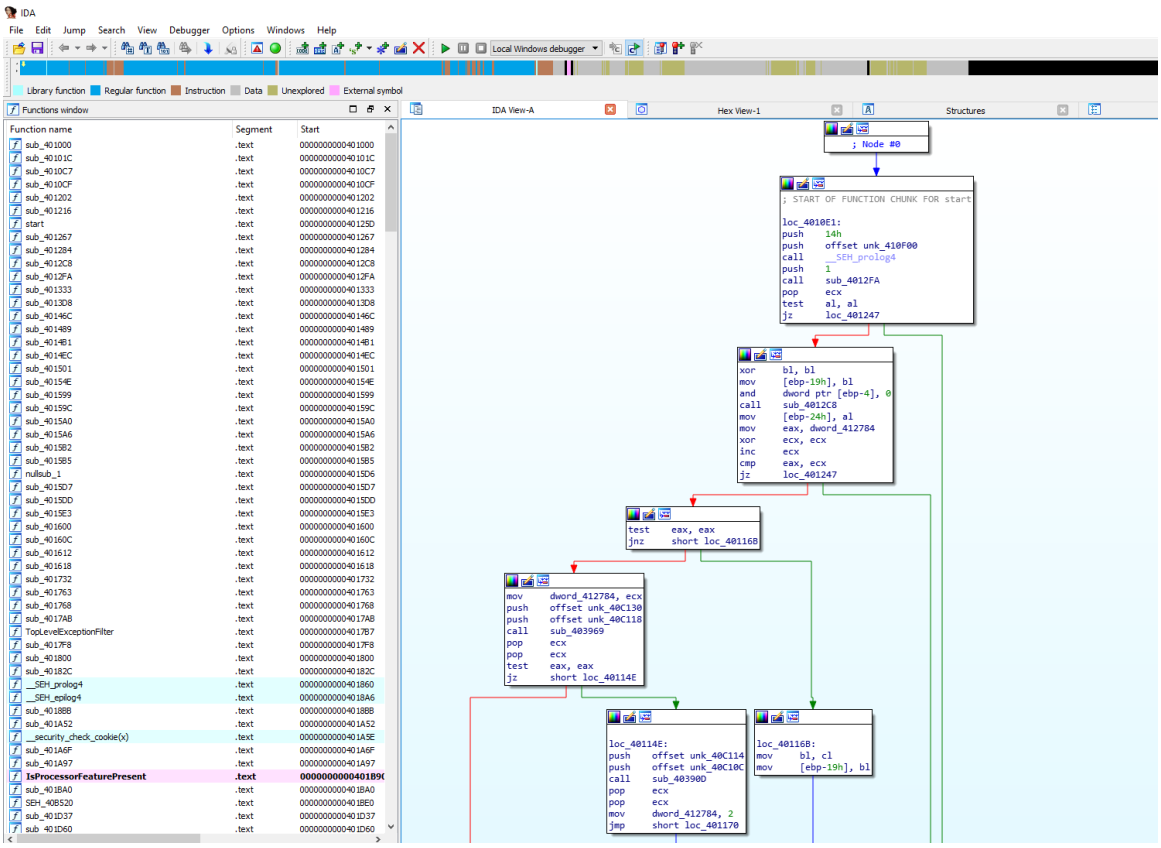
```
using ...

public class ExampleScript : MonoBehaviour
{
    [SerializeField]
    private int counter;

    private void Update()
    {
        this.counter = this.ExampleMethod(this.counter, 1);
        Debug.LogFormat("Counter: {0}", new object[]
        {
            this.counter
        });
    }

    private int ExampleMethod(int param1, int param2)
    {
        return param1 + param2;
    }
}
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ExampleScript : MonoBehaviour
{
    [SerializeField] private int counter = 0;

    // Update is called once per frame
    private void Update()
    {
        counter = ExampleMethod(counter, 1);

        Debug.LogFormat("Counter: {0}", counter);
    }

    private int ExampleMethod(int param1, int param2)
    {
        int sum = param1 + param2;
        return sum;
    }
}
```

Kuva 7. Esimerkki ILSpy-ohjelman tuottamasta käännöksestä

IDA (*Interactive Disassembler*) on erittäin tehokas ja suosittu disassembler-työkalu, joka tukee useita eri prosessoriarkkitehtuureja sekä tiedostotyyppejä. IDA tarjoaa myös debuggerin, joka mahdollistaa ohjelman analysoinnin ajon aikana. IDA on melko kallis ohjelma, mutta se tarjoaa

myös rajoitetumman ilmaisversion ei-kaupalliseen käyttöön. Yksi IDA:n parhaimmista ominaisuuksista on assembly-koodin esittäminen kulkukaaviona, joka helpottaa huomattavasti koodin jäsentämisessä. Kuvassa 8 on esimerkki IDA:n käyttöliittymästä, jossa myös näkyy esimerkki kaavionäkymästä. [18.]



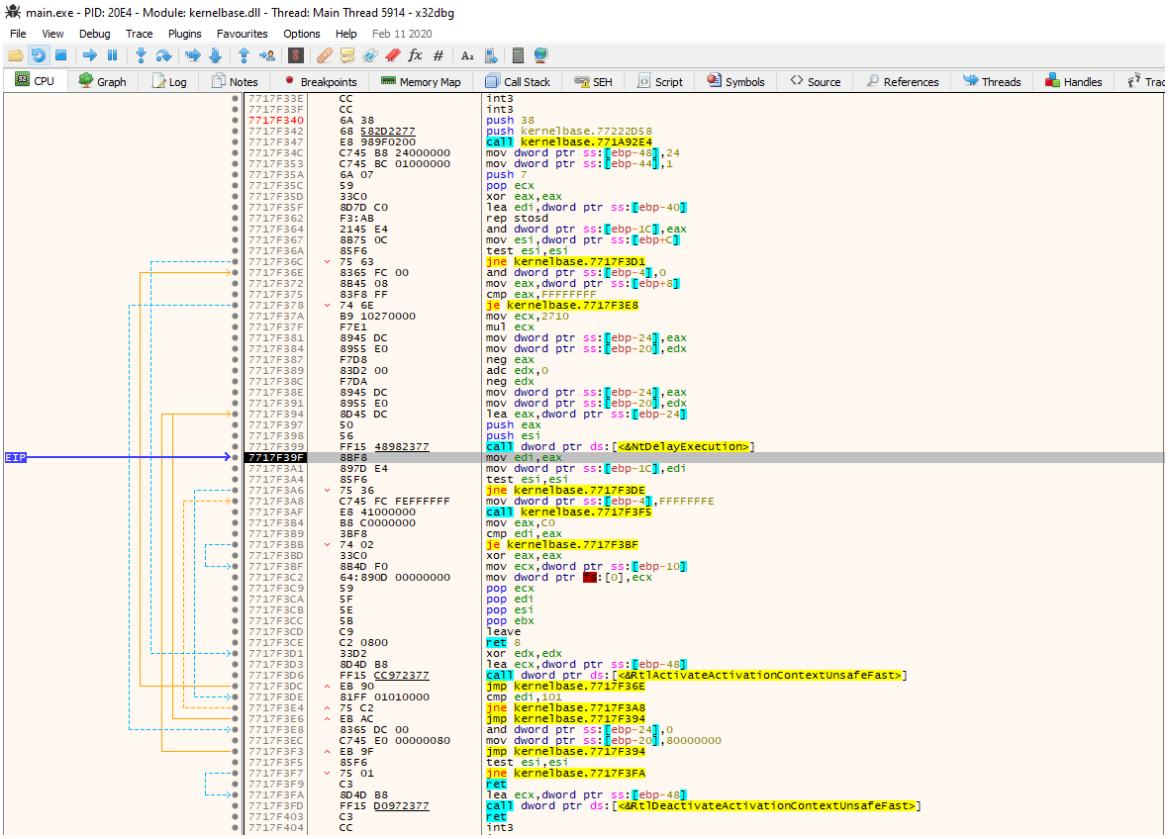
Kuva 8. IDA-ohjelman käyttöliittymä

3.3.3 Dynaaminen analysointi

Kuten staattisessa analysoinnissa, dynaamisessa analysoinnissa pyritään kääntämään ohjelma paremmin ymmärrettävään muotoon. Dynaamisessa analysoinnissa ohjelmaa kuitenkin analysoidaan sen ajon aikana, mikä tarjoaa huomattavasti enemmän tietoa ohjelman tilasta kullakin hetkellä. Ajon aikainen analysointi tapahtuu usein käyttämällä virheenkorjaajaa eli debuggeria, joka mahdollistaa ohjelman pysäyttämisen sekä suorittamisen käsky kerrallaan. Debuggeria yleensä käytetään ohjelmistokehityksessä virheiden ja bugien etsimiseen, jolloin debuggeri toimii usein lähdekooditasolla. Mikään ei kuitenkaan estä käyttämästä debuggeria muiden tuottamien ohjelmien, kuten pelien, analysointiin. Tällöin tarvitaan kuitenkin usein assembly-tason debuggeri,

jossa on disassembler-ominaisuus sisäänrakennettuna. Tällaisella debuggerilla on mahdollista asettaa pysäytyspisteitä eri kohtiin ohjelman assembly-koodia tai johonkin tiettyyn muistiosoitteeseen. Kun ohjelman suoritus osuu pysäytyspisteeseen, debuggeri pysäyttää ohjelman automaattisesti, jolloin on mahdollista tutkia ohjelman senhetkistä tilaa. [16, s. 116–118.]

x64dbg on avoimen lähdekoodin debuggeri, joka mahdollistaa sekä 64- että 32-bittisten ohjelmien debuggauksen Windows-ympäristössä. Se tarjoaa normaalien debuggaustyökalujen lisäksi mahdollisuuden tarkastella ohjelman muistin rakennetta, skriptaustuen sekä mahdollisuuden asentaa muiden tuottamia lisäosia. Kuvassa 9 on osa x64dbg:n käyttöliittymän päänäkymää debuggauksen aikana. [19.]



Kuva 9. x64dbg-käyttöliittymä

Peleissä huijaamisen kannalta ehkä yksi x64dbg:n hyödyllisimmistä ominaisuuksista on mahdollisuus paikata ohjelman koodia. Tämä mahdollistaa pelin koodin muokkaamisen esimerkiksi poistamalla tiettyjä osia koodista korvaamalla assembly-käskyjä käskyillä, jotka eivät tee mitään. Pelissä voi esimerkiksi olla kartta, johon viholliset piirretään vain, kun ne ovat pelaajan näköpiirissä. Tämä tarkastus on mahdollista päällekirjoittaa, jolloin viholliset piirtyvät aina kartalle. x64dbg

mahdollistaa näiden paikkausten tallentamisen erilliseksi tiedostoksi, joka on helppo ladata ohjelman uudelleen käynnistyessä, tai vaihtoehtoisesti paikkaukset voi kirjoittaa myös suoraan ohjelmatiedostoon.

3.4 Oman koodin suorittaminen

Ehkä kaikista voimakkain keino, mikä huijareilla on käytössä, on saada peli suorittamaan huijarin tekemää koodia. Suorittamalla omaa koodia pelin sisällä huijarin on mahdollista tehdä pelin sisällä käytännössä mitä tahansa. Onnistuakseen tämä yleensä vaatii hyvää ymmärrystä pelin toiminnasta, jonka takia se usein vaatii pelin takaisinmallintamista.

3.4.1 Koodin injektointi

Jos kyseessä on konekielelle käännetty ohjelma, omaa koodia on mahdollista lisätä käyttämällä esimerkiksi x64dbg-debuggerin paikkaustoimintoa. Tällöin koodia on kuitenkin kirjoitettava alkuperäisen ohjelman päälle, mikä kuitenkin rajoittaa uuden koodin lisäämistä, sillä korvattava osa koodia tulee olla aina samanpituinen alkuperäisten käskyjen kanssa. Siksi koodin paikkaus sopii lähinnä yksinkertaisille muokkauksille, kuten tiettyjen osien poistamiselle. [6, s. 31–32.]

Jos peliin halutaan lisätä uutta koodia, Windowsin ohjelmointirajapinta tarjoaa muutamia funktioita, joiden avulla on mahdollista injektoida omaa koodia käynnissä olevaan prosessiin. *VirtualAllocEx*-funktiota käyttämällä on mahdollista varata muistia toisesta prosessista, johon on mahdollista kirjoittaa haluttua assembly-koodia käyttämällä esimerkiksi *WriteProcessMemory*-funktiota. Tämän etuna on, että injektoitu koodi ei päällekirjoita mitään olemassa olevaa koodia. [6, s. 134–142.]

Pelkästään koodin kirjoittaminen prosessin muistiin ei kuitenkaan takaa, että ohjelma suorittaisi sitä. Peli on siis jotenkin huijattava suorittamaan tätä koodia. Tämä on mahdollista esimerkiksi käyttämällä *CreateRemoteThread*-funktiota, joka aloittaa uuden säikeen halutussa prosessissa. Yksi *CreateRemoteThread*-funktion parametreista on säikeen suorituksen aloituspiste, joka voidaan asettaa muistiin injektoituun koodipätkään. Vaihtoehtoisesti koodin suorittaminen on myös mahdollista kaappaamalla jonkin olemassa olevan säikeen suoritus, jolloin voidaan olla varmoja, että kyseinen säie ei suorita mitään, joka haittaisi injektoitua koodia. [6, s. 134–142.]

Koodin injektointi on erittäin voimakas keino saada peli suorittamaan huijauskoodia, mutta sen haittapuolena on, että injektoitu koodi tulee olla assembly-kielellä. Tämä tekee monimutkaisempien huijauksien kehittämisestä paljon vaikeampaa. Tähän on ratkaisuna DLL-kirjaston injektoiminen prosessiin, milloin on mahdollista kirjoittaa kaikki huijauskoodi käyttämällä esimerkiksi C++-kieltä. Prosessi on mahdollista huijata lataamaan tämä huijauskoodia sisältävä kirjasto sen muistiin, kuten esimerkiksi kuvassa 10 on tehty. Esimerkissä ensiksi kirjoitetaan DLL-kirjaston nimi kohdeprosessin muistiin. Tämän jälkeen haetaan *LoadLibrary*-funktion osoite, jota käytetään uuden säikeen käynnistämiseen kohdeprosessissa. Tämä säie sitten lataa halutun DLL-kirjaston, joka voi sen jälkeen aloittaa huijauskoodin suorittamisen pelissä. Kun kirjasto on ladattu pelin muistiin, se käyttäytyy aivan kuten mikä tahansa muu osa pelin koodia, joka helpottaa huomattavasti prosessin haltuun ottamista. [6, s. 142–145.]

```
void LoadDll(HANDLE process, const char* dllPath)
{
    int namelen = strlen(dllPath) + 1;
    LPVOID remoteString = VirtualAllocEx(process, NULL, namelen, MEM_COMMIT, PAGE_EXECUTE);
    WriteProcessMemory(process, remoteString, dllPath, namelen, NULL);

    HMODULE k32 = GetModuleHandleA("kernel32.dll");
    LPVOID funcAddr = GetProcAddress(k32, "LoadLibraryA");

    HANDLE hThread = CreateRemoteThread(process, NULL, NULL,
                        (LPTHREAD_START_ROUTINE)funcAddr, remoteString, NULL, NULL);

    WaitForSingleObject(hThread, INFINITE);
    CloseHandle(hThread);
}
```

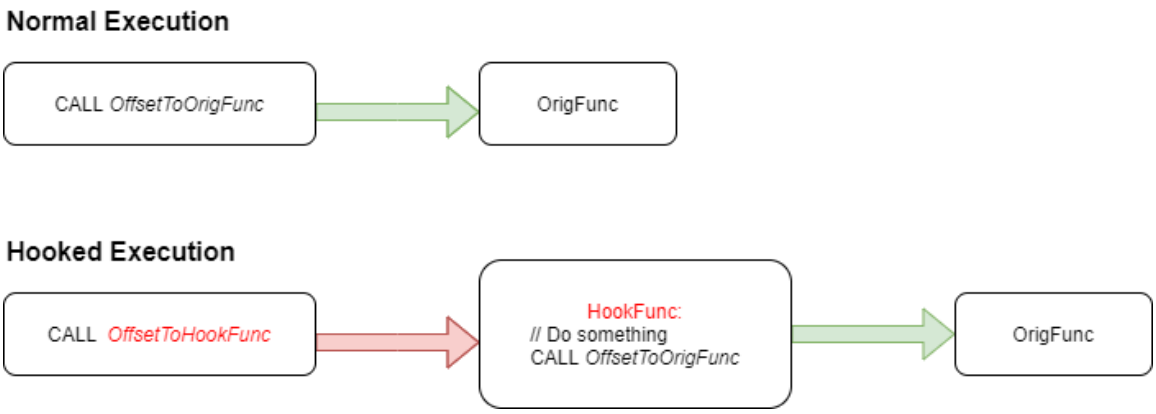
Kuva 10. DLL-kirjaston lataaminen toiseen prosessiin

Jos ohjelma on toteutettu C#-kielellä, myös silloin on mahdollista injektoida koodia, mutta oman koodin suorittamiseen on myös helpompi vaihtoehto. Koska C#-ohjelmat on usein mahdollista kääntää takaisin lähdekoodiksi, on niitä myös mahdollista muokata. Tämä toimii esimerkiksi Unity-pelimoottorilla tehdyissä peleissä, joissa on käytetty C#-kieltä pelilogiikan toteuttamiseen. DnSpy on työkalu, jolla on mahdollista muokata .NET-ohjelmien ja -kirjastojen sisältämää koodia lähdekooditasolla. DnSpy-työkalulla voi muokata mitä tahansa skriptiä tai metodia haluamakseen, tai vaikka lisätä kokonaan uusia skriptejä peliin. [20.]

3.4.2 Funktioiden hookkaus

Injektoimalla omaa koodia prosessiin pelin voi saada tekemään mitä vain. Usein kuitenkin voi olla helpompaa käyttää pelin omaa koodia hyödyksi ottamalla pelin omat funktiot parempaan käyttöön. Tätä prosessia kutsutaan funktioiden hookkaukseksi (eng. *hooking*). Funktioiden hookkaus mahdollistaa funktiokutsujen kaappaamisen, jolloin voidaan esimerkiksi suorittaa omaa koodia ennen tai jälkeen alkuperäistä funktiota, muokata funktion parametreja, tai koko funktiokutsun voi vaikka estää. Koska funktioita voidaan kutsua muutamalla eri tapaa, myös niiden hookkaus voidaan toteuttaa eri tavoin. [6, s. 153.]

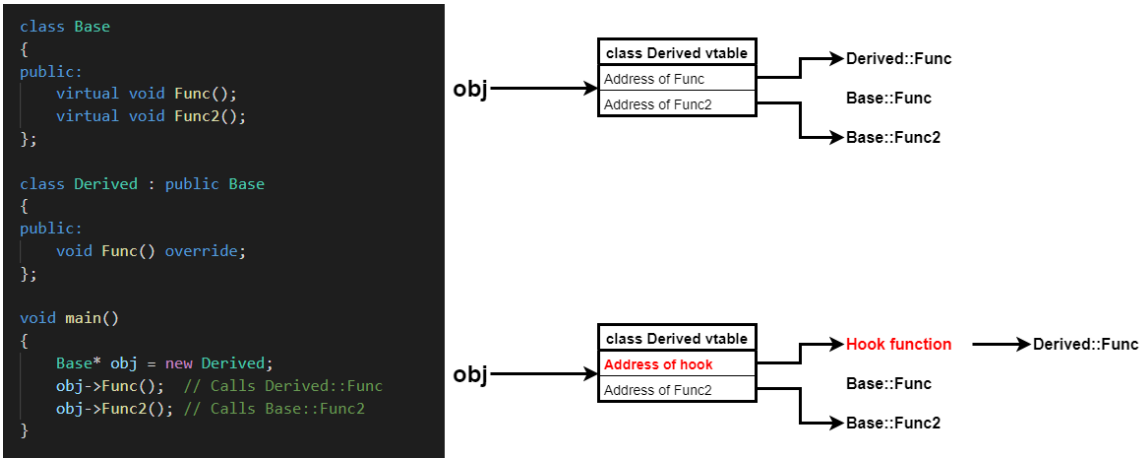
Yksinkertaisin tapa kutsua funktiota assembly-koodissa on CALL-käsky, jota seuraa etäisyys kutsuttavan funktion osoitteeseen. Tällaisen funktiokutsun hookkaaminen onnistuu helposti korvaamalla CALL-käskyä seuraava osoite osoittamaan omaan funktioon, mikä onnistuu esimerkiksi aiemmin esitellyillä muistinmanipulointifunktioilla. Hookatessa funktiota voidaan myös ottaa alkuperäisen funktion osoite talteen, jolloin sitä voidaan esimerkiksi kutsua huijauskoodin jälkeen. Kuvassa 11 on esitelty, kuinka tämä käytännössä toimii. Kutsuttavan oman funktion täytyy myös olla rakenteeltaan samanlainen alkuperäisen funktion kanssa, eli palautettavan arvon sekä parametrien on oltava identtisiä. Lisäksi sen on noudatettava samaa kutsumistapaa kuin alkuperäinen funktio, jotta vältyttäisiin virheiltä. [6, s. 153–155.]



Kuva 11. Esimerkki CALL-hookin toiminnasta

CALL-käskyn muokkaaminen vaikuttaa kuitenkin vain yhteen funktiokutsuun. Jos samaa funktiota kutsutaan useammasta paikasta, täytyisi jokainen kutsu hookata erikseen. CALL-hookkaukselle on kuitenkin muita vaihtoehtoja, jotka vaikuttavat jokaiseen funktiokutsuun. Tällaisia ovat esimerkiksi luokan virtuaalitaulun, eli vtablen, hookkaus ja IAT-taulun hookkaus.

C++-kielessä periytyvien luokkien on mahdollista tehdä omia versioita perittävän luokan funktioista eli metodeista. Näitä metodeja kutsutaan yleensä virtuaalisiksi metodeiksi. Jos ajon aikana ei ole tiedossa, mikä luokan alkuperäinen tyyppi on, virtuaalista metodia kutsutaan virtuaalitaulun eli vtablen kautta, joka on uniikki jokaiselle tyypille. Vtable on yksinkertainen taulukko, jossa on listattu kaikkien virtuaalisten metodien sijainnit. Nämä osoitteet ovat mahdollista päällekirjoittaa uusien funktioiden osoitteilla, jolloin jokaisen kyseisen luokan instanssin virtuaaliset metodikutsut ohjataan uusiin funktioihin. Kuvassa 12 oikealla on esimerkki yksinkertaisesta koodipätkästä, jossa kutsutaan virtuaalista metodia. Kuvassa vasemmalla yllä on esitetty, kuinka metodien kutsuminen normaalisti tapahtuisi, ja sen alla esimerkki, jossa *Func*-metodi on hookattu korvaamalla sen osoite vtableen. [6, s. 156–159.]



Kuva 12. Esimerkki vtable-hookin toiminnasta

Vtablen hookkaus on mahdollista kuitenkin vain, jos kyseessä on luokkaan kuuluva metodi, ja sitä kutsuvan objektin tyyppi ei ole tiedossa. Jos kyseessä onkin funktio, jota kutsutaan ohjelman ulkopuolelta erillisestä kirjastosta (dll), kuten monet Windowsin ohjelmointirajapinnan funktiot, joudutaan hyödyntämään muita keinoja. Koska ulkopuolisten kirjastojen funktioiden osoitteet ovat tiedossa vasta ohjelman käynnistyksen yhteydessä, Windows luo ohjelman muistiin vtablea muistuttavan taulukon näiden funktioiden osoitteista. Tätä taulukkoa kutsutaan IAT:ksi (*Import Address Table*), ja se sijaitsee moduulin PE-tunnisteessa. Aivan kuten vtable hookissa, funktioiden osoitteet on mahdollista korvata IAT-taulukkoon, jolloin jokainen kutsu näihin funktioihin ohjataankin huijarin funktioiden kautta. [6, s. 160–163.]

Kuvassa 13 on esimerkki, jossa hookataan *Sleep*-funktio IAT-taulukon kautta. Kun kohdeohjelma yrittää kutsua *Sleep*-funktiota, kutsuukin se ensin *SleepHook*-funktiota, joka esimerkissä vain ilmoittaa funktiokutsusta ja kutsuu sitten alkuperäistä funktiota. Itse hookkaus tapahtuu *HookIAT*-

funktiossa, joka etsii ja korvaa alkuperäisen *Sleep*-funktion osoitteen IAT-taulukkoon. *HookIAT*-funktion koodi on hieman monimutkainen, joten sitä ei ole esitelty tässä. HookIAT-funktio palauttaa alkuperäisen funktion osoitteen, jolloin korvaavan funktion on mahdollista kutsua sitä. Jotta esimerkkiä voitaisiin käyttää toiseen prosessiin, on siitä luotava dynaaminen kirjasto, joka voidaan injektoida kohdeprosessiin.

```
// Create function pointer for the original function
typedef void (WINAPI *_OrigSleep)(DWORD ms);
_OrigSleep origSleep;

void WINAPI SleepHook(DWORD ms)
{
    std::cout << "Sleep called for " << ms << "ms" << std::endl;
    // Call original function
    origSleep(ms);
}

void InjectEntry()
{
    // Perform hooking and assign original function
    origSleep = (_OrigSleep)Hooking::HookIAT("Sleep", (DWORD)&SleepHook);
    if (origSleep == NULL)
        std::cout << "Hooking Sleep FAILED!" << std::endl;
    else
        std::cout << "Hooking Sleep SUCCESS!" << std::endl;
}
```

Kuva 13. IAT-hook-esimerkki

Luvussa esiteltyjen hookkaustekniikoiden lisäksi on olemassa monia muita eri tapoja vaikuttaa koodin suorittamisjärjestykseen. Yksi mahdollinen tapa on jump-hookkaus, jossa funktion alkuun kirjoitetaan assembly-koodia ohjaamaan suoritus toisaalle. Jump-hookin toteutus on kuitenkin hieman monimutkaista, mutta funktiokutsujen hookkaukseen on olemassa useita kirjastoja, jotka tarjoavat toteutukset moniin eri hookkausmenetelmiin. PolyHook2 on monipuolinen kirjasto, joka tarjoaa luvussa esiteltyjen tekniikoiden lisäksi toteutukset monista muista hookkausmenetelmistä [21].

3.5 Pelin verkkoliikenteen analysointi ja manipulointi

Yksi parhaista keinoista saada tietoa pelin tilasta on seurata pelin verkkoliikennettä. Kaikki informaatio verkkopelin tilasta kommunikoidaan jollain tavalla verkon yli, joten verkkoliikennettä seuraamalla on mahdollista päästä käsiksi koko pelin tilaan. Verkkoliikenteen seuraaminen voi esi-

merkiksi johtaa informaation paljastumishuijauksiin, jos peli ottaa vastaan piilossa olevien vihollisten sijainteja. Verkkoliikennettä manipuloimalla voi myös toteuttaa refleksienparantajia lähettämällä verkkopaketteja, joissa pelaaja kääntyy automaattisesti kohti vastustajia sopivasti ennen ampumiskomentoa. [6, s. 206.] [2.]

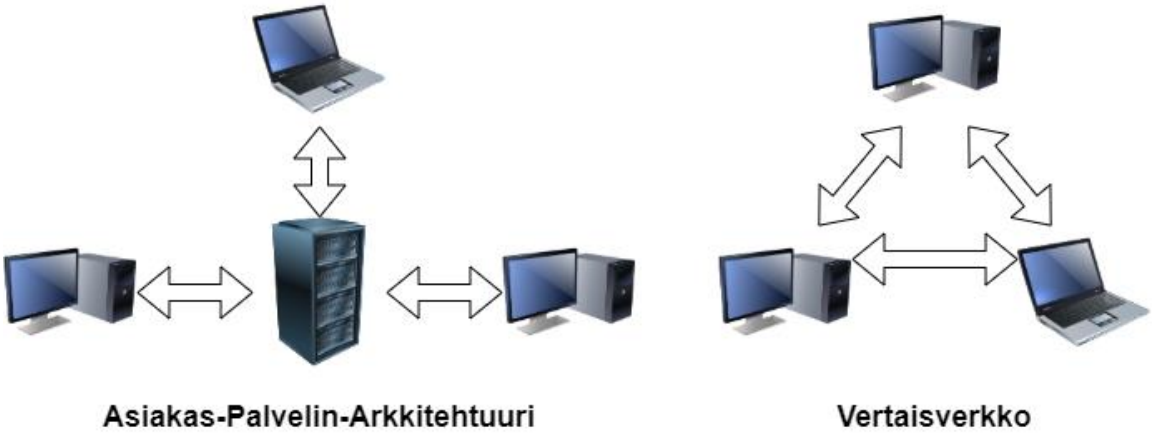
3.5.1 Pelien verkkoliikenne

Kaikki verkkoliikenne toimii käyttämällä TCP/IP-protokollaperhettä, joka on yhdistelmä eri kerroksiin jakautuneita verkkoprotokollia. Kaiken pohjana on IP-protokolla, jossa lähetettävä data pilkotaan IP-paketteihin. Jotta nämä paketit löytäisivät oikeaan ositteeseen, jokaiselle internetiin liitetylle koneelle on annettu oma IP-osoite, joka on esimerkiksi IPv4-protokollan tapauksessa muotoa 192.168.1.2. Jotta samalla tietokoneella voisi olla monta verkon yli kommunikoivaa prosessia, datan lähettämiseen käytetään yleensä TCP- tai UDP-protokollia. TCP ja UDP ovat kuljetuskerroksen protokollia, jotka toimivat IP-protokollan päällä. Ne lisäävät IP-osoitteeseen portit, jolloin verkkopaketit löytävät tiensä oikealle prosessille. Esimerkiksi portti 80 on varattu http verkkoliikenteelle. [22.]

Pelien verkkoliikenne on yleensä toteutettu joko TCP:llä tai UDP:llä. TCP muodostaa yhteyden kahden koneen välille ja takaa, että kaikki verkkopaketit toimitetaan perille oikeassa järjestyksessä. Varmennukseen TCP hyödyntää erilaisia algoritmeja, jotka myös saattavat aiheuttaa verkkoliikenteen paikoittaista hidastumista. Koska viive ei ole ideaalia peleille, jotka usein vaativat nopeita reaktioaikoja, pelit usein päätyvät käyttämään UDP:tä. UDP on yhteydetön protokolla, joka ei anna mitään takeita, että paketit toimitettaisiin perille asti. Paketit voivat myös saapua perille eri järjestyksessä, kuin missä ne lähetettiin. Tämä johtaa siihen, että pelien on itse huolehdittava kaiken tarvittavan datan saapumisesta perille. UDP:n etuna on kuitenkin sen nopeus verrattuna TCP:hen. [22.]

Ohjelmallisesti verkkopakettien lähettäminen ja vastaanottaminen useimmiten toteutetaan käyttämällä *Berkeley Sockets* -ohjelmointirajapintaa, joka sisältää toteutukset sekä TCP- että UDP-kommunikaatiolle. Windows-ympäristössä tämä on toteutettu Winsock-kirjastossa. Hyvä aloituspiste pelien verkkokommunikaation tutkimiselle voikin olla tarkastelemalla, kuinka peli hyödyntää Winsock-kirjaston funktioita. Tärkeimpiä Winsock-funktioita ovat *send* ja *recv*, joita käytetään datan lähettämiseen ja vastaanottamiseen. [22.]

Korkeammalla tasolla verkkopelien kommunikaatio voidaan useimmiten toteuttaa kahdella eri tapaa. Asiakas-palvelin-arkkitehtuurissa on yksi tai useampi palvelin, jonka kanssa kaikki asiakkaat kommunikoivat. Tällöin asiakkaat eivät ole tietoisia toisistaan. Vastakohtana asiakas-palvelin-arkkitehtuurille on vertaisverkko, jossa kaikki asiakkaat kommunikoivat keskenään, eikä palvelinta silloin tarvita. Kuvassa 14 on esitelty, miltä molemmat mallit käytännössä näyttävät. Näistä kahdesta mallista asiakas-palvelin-arkkitehtuuri on hieman yksinkertaisempi toteuttaa ja lisäksi helpompi suojata huijaamiselta. Asiakas-palvelin on myös suositumpi malli, joten tässä opinnäytetyössä keskitytään pääasiassa siihen. Monet tekniikat toimivat kuitenkin myös vertaisverkon tapauksessa. [22.]



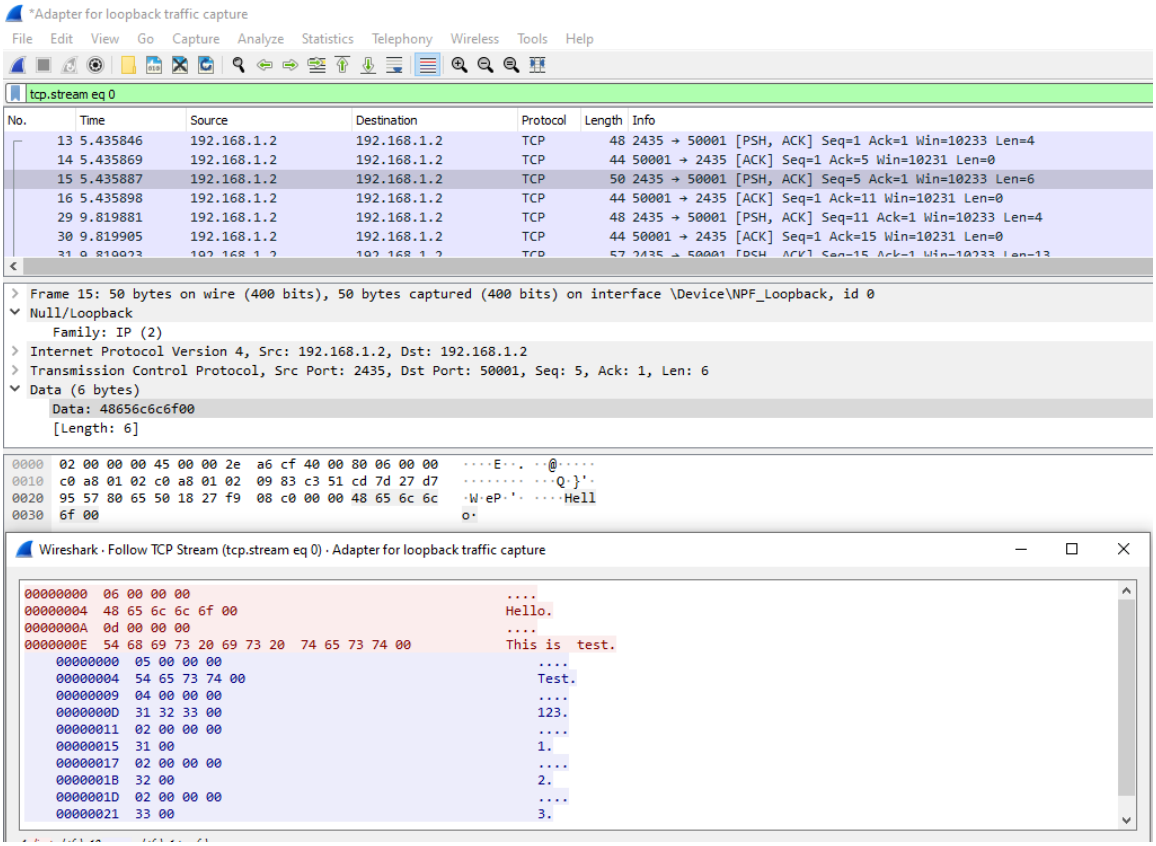
Kuva 14. Esimerkit erityyppisistä verkkoarkkitehtuureista peleissä

3.5.2 Verkkoliikenteen kaappaaminen ja manipulointi

Verkkoliikenteen kaappaaminen voidaan jakaa kahteen eri tapaan: passiiviseen ja aktiiviseen kaappaamiseen. Passiivisessa kaappaamisessa ei suoraan vaikuteta verkkoliikenteen kulkuun, vaan liikennettä ikään kuin salakuunnellaan. Aktiivinen kaappaaminen taas perustuu verkkoliikenteen udelleenohjaamiseen esimerkiksi erillisen välityspalvelimen kautta. [15.]

Wireshark on passiiviseen verkkoliikenteen kaappaamiseen ja analysointiin tarkoitettu suosittu avoimen lähdekoodin työkalu [23]. Kuvassa 15 on esimerkki, jossa Wiresharkia on käytetty yksinkertaisen viestiohjelman verkkoliikenteen kaappaamiseen. Kuvassa yllä näkyy kaikki verkkopaketit, jotka ohjelma on lähettänyt ja vastaanottanut. Paketteja voi myös tarkastella yksitellen, jolloin voidaan myös analysoida tarkemmin niiden ylätunnisteita ja sisältöä. Koska yksittäisistä pake-

teista voi olla hankala hahmottaa ohjelman verkkoprotokollaa, tarjoaa Wireshark mahdollisuuden tarkastella pakettivirtaa kokonaisuutena, kuten kuvan 15 alaosassa on tehty. Tämä antaa huomattavasti selkeämmän kuvan ohjelman verkkoliikenteestä kokonaisuutena, mistä voi olla apua sen analysoimisessa. Kuvan alalaidasta onkin mahdollista nähdä, että ohjelma lähettää ensin viestin pituuden kokonaislukuna, jonka jälkeen lähetetään vasta itse viesti.



Kuva 15. Verkkoliikenteen kaappaamista Wireshark-ohjelmalla

Wireshark on erinomainen työkalu verkkoliikenteen kaappaamiseen ja analysointiin, mutta sillä ei kuitenkaan ole mahdollista vaikuttaa verkkoliikenteeseen. Jos verkkopakettien sisältöä halutaan muokata, voidaan esimerkiksi luoda välityspalvelin pelin ja palvelimen välille. Välityspalvelimen voi esimerkiksi toteuttaa hookkaamalla verkkopakettien lähettämisestä ja vastaanottamisesta huolehtivat Winsock-kirjaston funktiot *send* ja *recv*. Näin paketteihin pääsee käsiksi ennen niiden lähettämistä tai vastaanottoa. Toinen vaihtoehto on luoda välityspalvelin reitittämällä verkkoliikenne uudelleen esimerkiksi toisen tietokoneen kautta. Tällaisen välityspalvelimen etuna on, että pelin on vaikeampi tunnistaa sen olemassaolo, sillä välityspalvelin toimii täysin erillään pelistä. Välityspalvelin voi siten tarkastella sen läpi kulkevaa liikennettä sekä tarvittaessa muokata pakettien sisältöä, injektoida uusia paketteja, tai täysin estää tiettyjen pakettien kulku. [24, s. 25–27.]

4 Huijauksilta suojautuminen

Tässä luvussa esitellään yleisimpiä suojauskeinoja, joita pelinkehittäjillä on käytössään. Tarpeeksi taitava ja motivoitunut huijari voi kuitenkin kiertää melkeinpä kaikki luvussa esiteltävät suojauskeinot, mutta mitä paremmin peli on suojattu, sitä vähemmän osaavia huijareita tulee olemaan. Huijauksien tunnistamisessa kannattaa kuitenkin olla tarkkana, jotta oikeita pelaajia ei vahingossa rangaista. Vuonna 2010 Valven huijauksenestojärjestelmä VAC antoi vahingossa porttikiellon 12 000 pelaajalle bugin seurauksena [25].

4.1 Määräävä palvelin

Jos kyseessä on peli, joka käyttää asiakas-palvelin-arkkitehtuuria, monet huijaukset toimivat, koska pelipalvelin luottaa sokeasti dataan, jota asiakkaat sille lähettävät. Matt Pritchardin peleissä huijaamisen neljäs sääntö kuuluu: mikään huijarin tietokoneella ei ole suojassa [2]. Huijareilla on pääsy kaikkiin pelin tiedostoihin ja pelin muistiin, ja kaikkea niitä on mahdollista muokata. Siksi yksi parhaimmista suojauskeinoista onkin olla luottamatta asiakkaaseen lainkaan. *Authoritative server*- eli määräävä palvelin -mallissa pelisimulaatio suoritetaan täysin palvelimella. Käytännössä pelaajat lähettävät vain omat näppäinpainalluksensa palvelimelle ja palvelin lähettää pelin senhetkisen tilan takaisin. Tällöin huijarit eivät pysty muokkaamaan pelille tärkeitä arvoja, kuten pelaajan elämäpisteitä, koska kaikki arvot ovat aina tallessa palvelimella, josta ne lähetetään pelaajille. [26.]

Määräävä palvelin vaatii kuitenkin huomattavasti enemmän suorituskykyä palvelimelta sekä se aiheuttaa hieman viivettä peliin, sillä pelaajien liikkeet täytyy ensin lähettää palvelimelle, jossa ne validoidaan ja lähetetään takaisin. Tämä ei välttämättä ole ongelma kaikissa pelityypeissä, mutta esimerkiksi nopeatempoisissa peleissä tai virtuaalitodellisuuden peleissä viive voi pilata pelikokemuksen. Viivettä on kuitenkin mahdollista peittää erilaisin ekstrapolaatio- ja interpolaatiotekniikoiden avulla. Näin pelaajan näkökulmasta viive voi olla lähes olematon, vaikka oikea pelisimulaatio suoritetaan vieläkin palvelimella. [26.]

4.2 Tärkeiden arvojen suojaaminen muistissa

Jos määräävää palvelinta syystä tai toisesta ei ole mahdollista toteuttaa, muistin manipulointia voi silti yrittää hankaloittaa. Yksinkertainen keino vaikeuttaa tärkeiden arvojen muokkaamista muistiskannereilla on salata nämä arvot muistissa. Salauksen voi esimerkiksi toteuttaa yksinkertaisella XOR-salauksella. XOR-salauksessa tehdään XOR-operaatio salattavan arvon ja salausavaimen välillä, mikä tuottaa salatun arvon. Salaus on helppo purkaa toteuttamalla XOR-operaatio salausavaimen kanssa uudelleen. Normaalisti XOR-salausta tulisi välttää sen yksinkertaisuuden takia, mutta yksinkertaisten arvojen piilottamiseen se on todennäköisesti riittävä. Kuvassa 16 on esimerkki hyvin yksinkertaisesta XOR-salauksen toteutuksesta kokonaisluvun suojaamiseksi. Esimerkissä salausavain luodaan satunnaisesti jokaiselle salattavalle kokonaisluvulle. Vaikka toteutus onkin todella yksinkertainen, eikä sitä kannata hyödyntää sellaisenaan oikeissa projekteissa, esimerkiksi Cheat Enginellä oli vaikeuksia löytää arvoa testiohjelman muistista. Jos XOR-salaus ei tarjoa riittävää salausta, arvojen salaamiseen on myös mahdollista käyttää monimutkaisempia salausalgoritmeja, kuten esimerkiksi AES-salausalgoritmia. [9, s. 197–204.]

```
class SecureInt
{
public:
    SecureInt(int i)
    {
        // Generate random encryption key, rand initialized elsewhere
        encryptionKey = std::rand();

        // XOR original value with encryption key
        value = i ^ encryptionKey;
    }

    int  GetValue() { return value ^ encryptionKey; }
    void SetValue(int newVal) { value = newVal ^ encryptionKey; }

private:
    int value;
    int encryptionKey;
};
```

Kuva 16. Yksinkertaistettu toteutus kokonaislukujen suojaamiseen muistissa

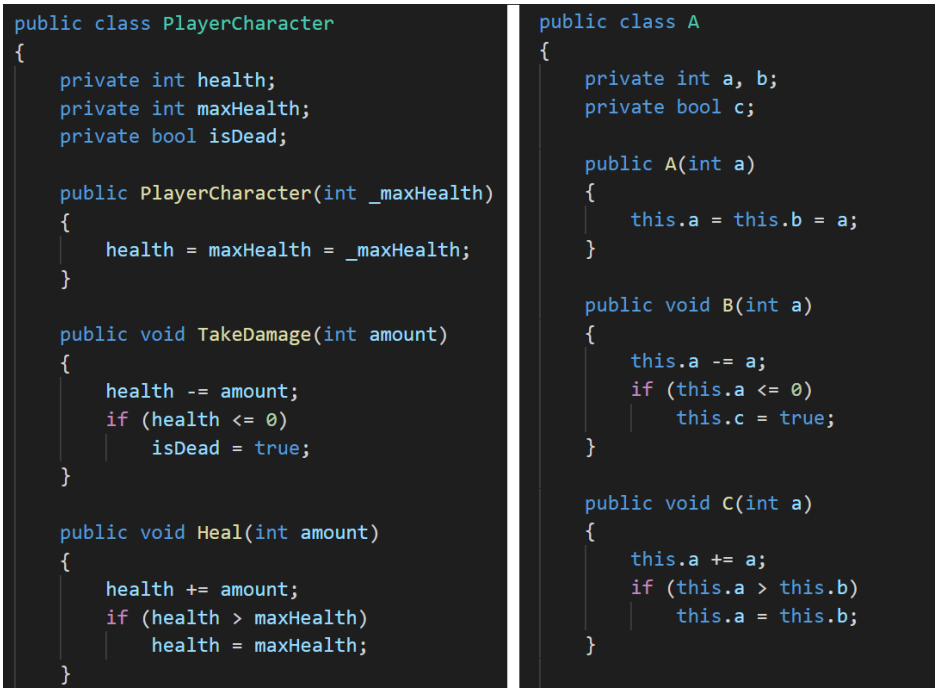
Arvojen muokkaamista voi myös yrittää estää luomalla niistä tiivisteitä, joihin alkuperäisiä arvoja voidaan verrata. Tiiviste voidaan luoda käyttämällä esimerkiksi jotain tiivistefunktiota, kuten

MD5:tä. Kun peli päivittää arvoja, lasketaan niistä samalla tiiviste, jota voidaan silloin tällöin verrata alkuperäiseen arvoon. Jos tiiviste eroaa alkuperäisestä arvosta, alkuperäistä arvoa on yritetty muokata pelin ulkopuolelta. Tämän voi viedä vielä pidemmälle luomalla muistiin valearvoja, joita ei oikeasti käytetä pelisimulaatiossa. Valearvoja voidaan sitten verrata oikeaan salattuun arvoon ja jos valearvo eroaa salatusta arvosta, on selvää, että joku on yrittänyt muokata sitä. [9, s. 204–206.]

4.3 Obfuskaatio

Koska monet huijaustekniikat vaativat ymmärrystä pelin toimintaperiaatteista, takaisinmallintaminen on iso osa huijausten kehitysprosessia. Pelinkehittäjät voivat yrittää hankaloittaa takaisinmallintamista käyttämällä erilaisia obfuskaatiotekniikoita. Obfuskaatiolla tarkoitetaan koodin rakenteen muokkaamista siten että sen luettavuutta hankaloitetaan, mutta toiminta pysyy ennallaan.

Obfuskaatio on tärkeää etenkin tavukoodiksi kääntyvillä ohjelmointikielillä, kuten esimerkiksi C#:ssa tai Javassa, sillä tavukoodi usein sisältää koodissa käytettyjen luokkien ja muuttujien nimiä. Yksi tapa obfuskoida tavukoodia on käyttää ohjelmaa, joka nimeää valmiissa koodissa esiintyvät luokat, metodit ja muuttujat uudelleen siten, että niiden nimet eivät merkitse enää mitään. Kuvassa 17 on esimerkki C#-koodista, jossa yksinkertainen luokka on obfuskoitu korvaamalla kaikki nimet yksittäisillä kirjaimilla. Luokan toiminta pysyy täysin samana, mutta koodia on huomattavasti vaikeampi lukea. [16, s. 329.]



```
public class PlayerCharacter
{
    private int health;
    private int maxHealth;
    private bool isDead;

    public PlayerCharacter(int _maxHealth)
    {
        health = maxHealth = _maxHealth;
    }

    public void TakeDamage(int amount)
    {
        health -= amount;
        if (health <= 0)
            isDead = true;
    }

    public void Heal(int amount)
    {
        health += amount;
        if (health > maxHealth)
            health = maxHealth;
    }
```

```
public class A
{
    private int a, b;
    private bool c;

    public A(int a)
    {
        this.a = this.b = a;
    }

    public void B(int a)
    {
        this.a -= a;
        if (this.a <= 0)
            this.c = true;
    }

    public void C(int a)
    {
        this.a += a;
        if (this.a > this.b)
            this.a = this.b;
    }
```

Kuva 17. Esimerkki kuinka obfuskaatio voisi toimia C#-kielellä

Koska esimerkiksi Unity-pelimoottori käyttää C#-kieltä pelilogiikan toteuttamiseen, Unityllä tehtyjen pelien lähdekoodia on mahdollista tarkastella lähes alkuperäisessä muodossaan. Valitettavasti Unityn tapa käsitellä C#-koodia ei mahdollista kaikkien skriptien täyttä obfuskoimista. Esimerkiksi peliobjekteihin liitettyjen skriptien nimiä ei ole mahdollista muokata. Lisäksi erityisten metodien, kuten esimerkiksi *Start* ja *Update*, nimiä ei voi muuttaa, koska Unity ei silloin osaa kutsua niitä. Unityn rajoituksia voi yrittää kiertää rakentamalla systeemin, joka ei luota kaikkiin pelimoottorin tarjoamiin ominaisuuksiin, mutta Unityssä on myös hieman parempi vaihtoehto koodin obfuskointiin. Sen sijaan, että C#-skriptit käännettäisiin pelkästään tavukoodiksi, on mahdollista käyttää Unityn IL2CPP-kääntäjää, joka kääntää C#-tavukoodin C++-kielelle, joka sitten kääntyy suoraan konekielelle. Vaikka tämä ei ole täysin oikea obfuskaatiotekniikka, konekieltä on huomattavasti vaikeampaa analysoida kuin C#-tavukoodia, eikä peliä voida enää kääntää takaisin lähdekoodiksi. [27.] [28.]

Vaikka konekieli onkin melko vaikealukuista sellaisenaankin, myös sitä on mahdollista obfuskoida. Koska takaisinmallintaminen suoritetaan lähes aina jonkin työkalun avustuksella, monet obfuskaatiotekniikat keskittyvät näiden työkalujen käytön hankaloittamiseen. Yksi tapa obfuskoida konekieltä on lisätä ylimääräisiä turhia käskyjä oikean koodin joukkoon. Turhia käskyjä ei ikinä suoriteta, mutta takaisinmallintamiseen käytettävät työkalut saattavat luulla niiden olevan osa ohjelmaa ja esittää ne oikeina käskyinä muun koodin seassa. Myös koodin suoritusjärjestykseen voi-

daan vaikuttaa esimerkiksi lisäämällä turhia hyppyjä koodin sekaan, mikä voi hankaloittaa kulkukaavion luomista koodista. Nämä obfuskaatiotekniikat voivat kuitenkin hidastaa ohjelman suorittamista, etenkin jos niitä käytetään usein kutsuttavissa funktioissa. Tämä voi olla pelien kannalta ei-toivottua, sillä pelit usein vaativat paljon suorituskykyä, ja pienetkin hidastukset voivat suoraan vaikuttaa pelikokemukseen. [16, s. 336–356.]

Koko ohjelma on myös mahdollista salata käyttämällä packer-työkalua. Kun salattu ohjelma käynnistyy, puretaan salaus ja alkuperäinen ohjelma suoritetaan. Pakatun ohjelman staattinen analysointi on miltei mahdotonta, jos salausavain ei ole tiedossa. Packer-ohjelmat ovat kuitenkin kierrettävissä, sillä ohjelmaa voi tarkastella sen suorituksen aikana. Lisäksi monille yleisille packer-työkaluille on vastaava unpacker-työkalu, joka osaa purkaa ohjelman salauksen. [16, s. 330.]

Obfuskaatio ei kuitenkaan ole tekniikka, joka suojaisi pelin kaikilta huijauksilta. Matt Pritchardin viides sääntö peleissä huijaamisessa on: epämääräisyys ei ole turvallisuutta (*Obscurity is not security*) [2]. Takaisinmallintamista ei voi ikinä täysin estää, ja osaavalle huijarille obfuskaatiotekniikat ovat vain pieni rasite. Vaikka obfuskaatiotekniikat voitaisiin ohittaa, ei se kuitenkaan tarkoita, ettei niitä kannattaisi käyttää pelien suojaamiseen. Suojaamattoman pelin takaisinmallintamiseen voi mennä vain hetki, mutta obfuskoidun pelin takaisinmallintaminen voi kestää jopa viikkoja. Tänä aikana on mahdollisesti julkaistu peliin jo uusi päivitys, jossa koodia on esimerkiksi obfuskoitu hieman eri tavoin, ja huijarin tulee aloittaa takaisinmallinnusprosessi alusta. [29.]

4.4 Anti-debug-tekniikat

Debuggerin tunnistaminen on tärkeää pelin suojaamiseksi, sillä monet huijaustekniikat vaativat ymmärrystä pelin toiminnasta. Jos debuggerin toimintaa voi hankaloittaa, huijauksien kehittämisestä tulee huomattavasti hankalampaa. Debuggerin tunnistaminen on mahdollista käyttämällä erilaisia anti-debug-tekniikoita. Osa näistä tekniikoista on kuitenkin helpostikin kierrettävissä, mutta ne voivat silti toimia tehokkaina keinoina estämään aloittelevia huijareita. [9, s. 178.] [30.]

Windowsin ohjelmointirajapinta tarjoaa kaksi funktiota debuggerin tunnistamiseen. Nämä ovat *IsDebuggerPresent* ja *CheckRemoteDebuggerPresent*. Kuvassa 18 on esimerkit näiden funktioiden käytöstä. Koska nämä funktiokutsut voi olla mahdollista huomata pelin koodia analysoidessa, kuvassa on myös esimerkki, kuinka *IsDebuggerPresent* voidaan toteuttaa assembly-kielellä, jolloin funktio on huomattavasti vaikeampi tunnistaa. Kaikki esimerkkifunktiot on lisäksi määritelty

__forceinline-avainsanalla, joka pakottaa kääntäjän lisäämään funktion koodin suoraan funktiokutsun paikalle. Tällöin funktiokutsuja ei voi helposti estää poistamalla funktion koodi, sillä se löytyy jokaisesta paikasta, josta funktiota on kutsuttu. [9, s. 178–186.] [6, s. 251.]

```
__forceinline bool CheckForDebugger()
{
    return IsDebuggerPresent();
}

_forceinline bool CheckForDebuggerRemote()
{
    BOOL beingDebugged = FALSE;
    CheckRemoteDebuggerPresent(GetCurrentProcess(), &beingDebugged);
    return beingDebugged;
}

__forceinline bool CheckForDebuggerFlagAsm()
{
    DWORD res = 0;
    __asm
    {
        mov     eax, fs:[30h]
        movzx   eax, byte ptr [eax + 02h]
        mov     res, eax
    }
    return res == 1;
}
```

Kuva 18. Eri funktioita debuggerin tunnistamiseen

Kun prosessia debuggataan, koodia usein analysoidaan suorittamalla sitä yksi käsky kerrallaan. Tätä on mahdollista hyödyntää debuggerin tunnistuksessa mittaamalla tiettyjen funktioiden suorittamiseen kulunutta aikaa. Jos funktion suoritukseen kuluu yllättäen normaalia enemmän aikaa, on mahdollista, että prosessia yritetään debuggata. Tämän tekniikan kanssa kannattaa kuitenkin olla hieman varovainen, sillä eri tietokoneilla voi kulua suoritukseen eri määrä aikaa. Tämän takia suoritusajan mittausta ei kannata käyttää pääasiallisena debuggerintunnistustekniikkana, mutta se voi toimia muiden tekniikoiden apuna huijausten tunnistamiseen. [30.]

Koska Windows antaa vain yhden prosessin debuggata kohdeprosessia samaan aikaan, debuggaus on mahdollista estää luomalla debuggeri itse. Itseänsä debuggaava ohjelma voi käynnistyessään luoda kopion itsestään, jota alkuperäinen prosessi voi sitten debuggata. Prosessit voivat myös kommunikoida keskenään, jolloin debuggaava prosessi voi vaikuttaa debuggattavan prosessin suoritukseen esimerkiksi muokkaamalla sen koodia. Itseänsä debuggaava prosessi voi huomattavasti vaikeuttaa pelin toiminnan analysointia, mutta se on todella vaikea toteuttaa. [30.]

4.5 Pelin tiedostojen tarkastaminen

Pelin tiedostot on hyvä validoida käynnistettäessä, jolloin voidaan varmistua siitä, että kukaan ei ole päässyt muokkaamaan tiedostoja. Muokkaamalla pelin tiedostoja huijari voi esimerkiksi lisätä omaa koodia peliin tai muokata pelin käyttämiä assetteja, kuten tekstuureja tai 3D-malleja, mikä voi esimerkiksi johtaa seinien poistamiseen pelistä. Validointi voidaan toteuttaa esimerkiksi käyttämällä jotakin tiivistefunktiota, kuten MD5 tai SHA-perheen funktioita. Tiivistefunktiolla voidaan laskea jokaisesta pelin tiedostosta tiiviste, joka voidaan sitten lähettää tarkastettavaksi palvelimelle, jossa niitä voidaan verrata hyväksyttäviin tiivisteisiin. Koska pienikin muutos tiedostossa tuottaa täysin erilaisen tiivisteen, vertaamalla tiivisteitä voidaan varmistua, että tiedostoja ei ole muokattu. Pelin tiedostoja voi myös tarkastaa pelin suorituksen aikana vertaamalla muistissa olevaa koodia levyllä olevaan. Näin on esimerkiksi mahdollista tunnistaa, jos koodista on yritetty poistaa joitain osia tai koodin suorittamisjärjestykseen yritetään vaikuttaa. [6, s. 247–248.]

Ajon aikana voidaan myös tarkastaa prosessin lataamat moduulit. Kun DLL-kirjasto injektoidaan prosessiin, se tulee näkyviin prosessin moduulilistaan. Windowsin ohjelmointirajapinta tarjoaa *EnumProcessModules*-funktion, joka mahdollistaa prosessin lataamien moduulien tarkastelun. Kuvassa 19 on esimerkki C++-koodista, joka tulostaa prosessin senhetkiset moduulit. Käyttämällä esimerkissä olevaa koodia prosessin lataamat moduulit voidaan esimerkiksi lähettää palvelimelle tarkastettavaksi, jolloin tuntemattomat moduulit voidaan tunnistaa. [31.]

```
void PrintProcessModules(HANDLE hProcess)
{
    HMODULE hMods[1024];
    DWORD totalSize;

    std::cout << "Modules for process: " << std::endl;

    if (EnumProcessModules(hProcess, hMods, sizeof(hMods), &totalSize))
    {
        int moduleCount = totalSize / sizeof(HMODULE);
        for (int i = 0; i < moduleCount; ++i)
        {
            char modName[MAX_PATH];

            if (GetModuleBaseNameA(hProcess, hMods[i], modName, sizeof(modName) / sizeof(char)))
            {
                std::cout << modName << std::endl;
            }
        }
    }
}
```

Kuva 19. Prosessin moduulien listaus

4.6 Verkkoliikenteen suojaaminen

Koska huijareiden on mahdollista tarkastella pelin verkkoliikennettä esimerkiksi käyttämällä Wireshark-työkalua, on heillä pääsy kaikkeen pelin verkkoliikenteeseen. Salaamalla verkkoliikenne tätä tarkastelua voidaan vaikeuttaa huomattavasti. Jos peli prosessoi arkaluontoisia tietoja, verkkoliikenteen salaus on suositeltavaa, jotta kolmas osapuoli ei pääse käsiksi esimerkiksi pelaajien käyttäjätietoihin. Joskus verkkoliikenteen salaus voi olla myös vaadittu laissa, esimerkiksi jos peli prosessoi maksukorttien tietoja. [22.]

Verkkoliikenteen salaukseen kannattaa käyttää yleisesti hyväksyttyjä salausalgoritmeja, kuten RSA- ja AES-algoritmeja. RSA on julkisen avaimen salausalgoritmi, jossa molemmilla osapuolilla on kaksi salausavainta, julkinen ja yksityinen. Datan salaus toimii salaamalla se käyttämällä toisen osapuolen julkista avainta. Salattu data on mahdollista lukea vain käyttämällä julkista avainta vastaavaa yksityistä avainta, joka on tiedossa vain datan vastaanottajalla. RSA on kuitenkin melko hidas, eikä se sovellu kovin isojen datamäärien salaamiseen. Siksi usein RSA:n rinnalla käytetään jotain symmetrisen avaimen salausalgoritmia, kuten AES-algoritmia. AES on lohkosalausalgoritmi, jossa on käytössä vain yksi salausavain, jota käytetään sekä datan salaamiseen että sen lukemiseen. Yleisesti käytössä oleva ratkaisu on käyttää RSA:ta salaamaan salausavain, jota on käytetty muun viestin salaamiseen AES-algoritmilla. [15.] [22.]

Verkkoliikenteen salaaminen ei kuitenkaan täysin estä sen tarkastelua, sillä tarpeeksi motivoituneen huijarin on aina mahdollista löytää käytetyt salausavaimet pelin muistista. Lähetettäviä verkkopaketteja on myös mahdollista tarkastella ennen niiden salausta, jos huijari löytää ne pelin muistista. Verkkoliikenteen tarkastelun tarkoituksena on usein löytää dataa, joka ei ole pelaajalle suoraan näkyvissä, esimerkiksi seinien takana olevien vihollisten sijainteja. Verkkoliikenteen salaaminen ei tule poistamaan mahdollisuutta lukea tätä dataa. Parempi ratkaisu on lähettää asiakkaille vain tarpeellinen data, jolloin vaikka verkkopaketteja pystyisi tarkastelemaan, eivät ne sisältäisi tietoa, jota pelaaja ei näkisi ruudullaan. Esimerkiksi pelin kartan toisessa päässä olevia vihollisia ei tarvitse lähettää pelaajalle, jos pelaaja ei heitä näkisi. [22.]

Pelin verkkoprotokollaa voi myös päivittää usein, jolloin huijareiden tulee analysoida verkkoprotokollaa uudestaan jokaisen päivityksen jälkeen. Jos esimerkiksi peli erottelee eri pakettityypit tietyillä tunnisteilla, voidaan näitä vaihdella joka päivityksen yhteydessä. Myös pakettien sisäistä rakennetta voidaan muokata, jolloin paketteja on vaikeampi tunnistaa niiden rakenteen perus-

teella. Jos verkkoprotokollaa päivitetään usein, on myös mahdollista tunnistaa huijauksia käyttäviä pelaajia, jos kyseiset pelaajat yrittävät lähettää verkkopaketteja, jotka eivät enää vastaakaan päivitettyä verkkoprotokollaa. [22.]

4.7 Palvelinpuolen analyysi

Koska huijarilla on mahdollisuus hallita peliä täysin hänen omalla koneellaan, monet asiakaspuolen huijauksenestomekaniikat on mahdollista kiertää. Siksi myös palvelimen puolelle olisi hyvä lisätä huijauksien tunnistamista. Palvelinpuolen huijaustentunnistus soveltuu etenkin bottien tunnistamiseen, sillä palvelimella on mahdollista analysoida kaikilta pelaajilta tulevaa dataa. Pelaajien dataa analysoidessa voidaan siten tunnistaa poikkeamia, jotka voivat olla merkkejä botin käytöstä. Esimerkiksi harva pelaaja pystyy pelaamaan peliä kymmeniä tunteja putkeen, mutta boteilta tämä onnistuu helposti. Pelaajien datan analysointiin on myös mahdollista käyttää koneoppimisen algoritmeja. Koska huijarit voivat kuitenkin muistuttaa käyttäytymisellään parhaimpia pelaajia, tulee pelaajien datan analysoimisessa olla erittäin tarkkana, jottei oikeita pelaajia merkitä huijareiksi. Vahingossa pelin parhaan pelaajan estäminen voi aiheuttaa paljon huonoa julkisuutta pelille. [32.]

Koska monet botit ovat melko yksinkertaisia, ne usein toistavat tiettyä kaavaa, joka on tunnistettavissa. Palvelimella voidaan esimerkiksi analysoida pelaajien lähettämien komentojen toistuvuutta ja säännöllisyyttä. Esimerkiksi yksinkertaiset botit saattavat lähettää komentoja aina tietyin aikavälein, mikä ei ole ihmispelaajalta mahdollista. Etenkin MMO-peleissä botit saattavat myös usein seurata tiettyä reittiä esimerkiksi tappamassa vihollisia. Pelaajien liikkeitä analysoimalla voidaan tunnistaa, jos joku pelaaja käy todella usein tietyissä paikoissa, mikä voi olla merkki botin käyttämisestä. [9, s. 72.] [33.]

4.8 Käyttöjärjestelmän monitorointi

Käyttöjärjestelmän monitorointi on tekniikka, jota isot ja tunnetut huijauksenesto-ohjelmistot käyttävät osana huijauksien estämistä. Koska käyttöjärjestelmän monitorointiin käytettävät tekniikat ovat melko monimutkaisia, eivät ne välttämättä ole pienien pelinkehitystiimien toteutettavissa. Lisäksi osa näistä tekniikoista muistuttaa toiminnaltaan vakoiluohjelmia, joten ne voidaan kokea myös pelaajien yksityisyyttä loukkaavina.

Huijaustekniikoita käsiteltäessä esiteltiin, kuinka huijaukset hyödyntävät funktioiden hookkausta pelin kulun muuttamiseen. Hookkausta voi kuitenkin käyttää myös pelin suojaamiseen, ja monet isot huijauksenesto-ohjelmistot hyödyntävätkin tätä. Hookkaamalla tiettyjä käyttöjärjestelmän funktioita, kuten esimerkiksi *OpenProcess*-funktion, huijauksenesto-ohjelmistot voivat monitoroida näiden funktioiden käyttöä, ja tarvittaessa myös estää niiden suorituksen kokonaan. Esimerkiksi kutsu *OpenProcess*-funktioon, joka antaa pääsyn toiseen prosessiin, voidaan estää, jos sillä yritetään avata suojattavaa peliä. Huijauksenesto-ohjelmisto voi myös siirtyä kernel-tilaan, jossa se voi hookata ja monitoroida myös kernel-tason funktiokutsuja, jolloin käyttäjätilassa toimivat huijausohjelmat eivät voi yrittää kiertää sitä. [6, s. 248–249.]

Huijausohjelmia voi myös tunnistaa hyödyntämällä Signature Based Detectionia, eli SDB:tä. SDB on esimerkiksi virustorjuntaohjelmistojen käyttämä tekniikka, joka analysoi tietokoneen muistissa olevia ohjelmia ja luo niistä tunnisteen, jota voidaan verrata palvelimella oleviin, jo tiedossa oleviin, huijausohjelmien tunnisteisiin. SDB ei voi kuitenkaan tunnistaa uusia huijausohjelmia, mutta se on erittäin tehokas tapa tunnistaa yleisiä huijausohjelmia, jotka ovat jo kehittäjien tiedossa. Monet tunnetut huijauksenesto-ohjelmistot hyödyntävät SDB:tä, kuten Valven VAC, Blizzardin Warden ja Punkbuster. [6, s. 246–250.]

Useita huijauksenesto-ohjelmistoja on kuitenkin kritisoitu ratkaisuista, joita pelaajat ovat kokeneet liian tunkeileviksi. Esimerkiksi monet tietoturvatutkijat ovat arvostelleet Blizzardin Warden-huijauksenesto-ohjelman käyttämiä tekniikoita, sillä se muistuttaa toiminnaltaan läheisesti vakoiluohjelmaa. Warden voi lukea kaikkien aukiolevien ikkunoiden otsikot ja lähettää ne analysoitavaksi Blizzardille. Lisäksi Warden skannaa aukiolevien prosessien muistia etsiäkseen sieltä huijausohjelmia. Vaikka Blizzard käyttäisi Wardenin antamaa tietoa vain huijauksien estämiseen, on muiden prosessien analysointi vähintäänkin kyseenalaista. Pelaajilla ei usein kuitenkaan ole vaihtoehtoja yksityisyyttä loukkaavien huijauksenestojärjestelmien kanssa, sillä pelatakseen peliä heidän on hyväksyttävä sen toiminta pelin palveluehdoissa. [24, s. 49–53.]

5 Yhteenveto

Peleissä huijaaminen on iso ongelma, ja pahimmassa tapauksessa se voi myös johtaa menetettyihin tuloihin pelienkehittäjille. Huijaaminen voi kohdistua eri osiin peliä, joten ne voidaan jakaa esimerkiksi neljään eri tasoon: pelitasoon, ohjelmatasoon, protokollatasoon ja infrastruktuuritasoon. Pelitason huijaukset ovat lähinnä pelinsisäisiä bugeja, joten ne ovat hyvin pelikohtaisia. Tässä opinnäytetyössä keskityttiin lähinnä kolmen jälkimmäisen huijauskategorian tekniikoihin ja niiltä suojautumiseen. Ohjelmatason huijaukset toimivat muokkaamalla pelin toimintaa esimerkiksi jonkin huijausohjelman avulla. Protokollatasolla huijaaminen tapahtuu vaikuttamalla pelin verkkoprotokollaan esimerkiksi manipuloimalla lähetettäviä verkkopaketteja. Infrastruktuuritason huijaukset toimivat täysin pelin ulkopuolella esimerkiksi muokkaamalla grafiikka-ajureita.

Jotta pelit voisi suojata mahdollisimman hyvin, kannattaa tuntea yleisimmät huijaustekniikat ja -ohjelmat. Luvussa 3 esiteltiin pelin muistin lukemista ja manipulointia, syötteen simulointia, takaisinmallintamista, tekniikoita huijauskoodin suorittamiseen sekä pelin verkkoliikenteen analysoimista ja manipulointia. Muistin manipulointi on ehkä yksinkertaisin huijaustekniikka, ja se mahdollistaa lähes minkä tahansa arvon muokkaamisen pelin muistissa. Muistin manipulointiin voidaan käyttää esimerkiksi Cheat Engine -ohjelmaa, mutta myös oman muistiskannerin toteutus on mahdollista Windowsin ohjelmarajapinnan funktioilla. Syötettä simuloimalla huijari voi lähettää näppäinpainalluksia ja hiiren liikkeitä pelille ja esimerkiksi luoda botteja pelaamaan peliä huijarin puolesta. Syötteen simulointiin voidaan esimerkiksi käyttää AutoIt-skriptikieltä.

Takaisinmallintamalla huijarit voivat analysoida pelin toimintaa ja sitä kautta esimerkiksi löytää uusia keinoja huijata. Takaisinmallintaminen voi olla joko staattista, jolloin peliä analysoidaan ilman sen suorittamista, tai dynaamista, jolloin peliä analysoidaan sen suorittamisen aikana. Staattisessa analysoinnissa käytetään yleensä erilaisia decompiler- ja disassembler-työkaluja, kuten esimerkiksi ILSpy- tai IDAPro-ohjelmia. Dynaamisessa analysoinnissa hyödynnetään usein assemblytason debuggeria, kuten esimerkiksi x64dbg-debuggeria. Kun huijarilla on tarpeeksi hyvä ymmärrys pelin toiminnasta, pelin voi saada suorittamaan huijarin toteuttamaa koodia. Oman koodin suorittaminen on mahdollista esimerkiksi injektoimalla joko assemblytason koodia tai kokonaisia DLL-kirjastoja prosessiin. Usein on myös mahdollista ottaa pelin käyttämät funktiot omaan käyttöön hookkaamalla funktiokutsuja. Hookkaamalla huijarin on mahdollista suorittaa omaa koodia ennen alkuperäisen funktion suorittamista, jolloin alkuperäinen funktiokutsu voidaan jopa ohittaa. Huijarin on myös mahdollista analysoida ja manipuloida pelin verkkoliikennettä, mikä voi

mahdollistaa esimerkiksi verkkopakettien muokkaamisen. Suosittu työkalu verkkoliikenteen analysoimiseen on Wireshark, mutta jos verkkoliikennettä halutaan manipuloida, on huijarin usein toteutettava välityspalvelin pelin ja palvelimen välille.

Luvussa 4 esiteltiin keinoja, joilla huijaamista on mahdollista vaikeuttaa tai jopa estää. Tärkein näistä on *authoritative server* eli määräävä palvelin. Määräävän palvelimen tapauksessa koko pelisimulaatio suoritetaan palvelimella, jolloin esimerkiksi huijarin ei ole mahdollista muokata muistia, sillä kaikki arvot sijaitsevat palvelimella. Jos määräävä palvelin ei jostain syystä ole mahdollista toteuttaa, tärkeät arvot olisi hyvä suojata pelin muistissa. Suojaaminen onnistuu esimerkiksi käyttämällä yksinkertaista XOR-salausta. Myös takaisinmallintamista olisi hyvä hankaloittaa, sillä monet monimutkaisemmat huijaukset vaativat ymmärrystä pelin toiminnasta. Obfuskaatiolla tarkoitetaan koodin luettavuuden hankaloittamista, ja se on tärkeää etenkin tavukoodiksi kääntyvillä ohjelmointikielillä, joissa on mahdollista palauttaa alkuperäinen lähdekoodi. Myös debuggerin toimintaa voidaan vaikeuttaa käyttämällä erilaisia anti-debug-tekniikoita. Pelin tiedostot olisi hyvä myös tarkastaa pelin käynnistyksen yhteydessä, jolloin voidaan huomata, jos peliä on yritetty muokata. Verkkoliikenteen analysointia kannattaa vaikeuttaa käyttämällä salausta, jolloin esimerkiksi verkkopaketteja ei voida enää analysoida Wireshark-työkalulla. Salaus voidaan toteuttaa esimerkiksi RSA- ja AES-salausalgoritmeja käyttämällä. Koska huijarin on kuitenkin mahdollista ohittaa miltei kaikki huijauksenestot omalla tietokoneellaan, huijauksien tunnistamista kannattaa toteuttaa myös palvelimelle. Palvelimella voidaan esimerkiksi analysoida pelaajien toimintoja ja tunnistaa toistuvia kuvioita, jotka voivat olla merkkejä bottien käytöstä. Monet kaupalliset huijauksenesto-ohjelmistot lisäksi monitoroivat käyttöjärjestelmän toimintaa pelaajien tietokoneella. On esimerkiksi mahdollista analysoida prosessien muistia ja etsiä sieltä tunnettujen huijausohjelmien tunnisteita. Käyttöjärjestelmän monitorointi voi kuitenkin joissain tapauksissa loukata pelaajien yksityisyyttä.

Huijaukset kehittyvät koko ajan, ja monet monimutkaisemmat huijaukset käyttävät edistyneempiä versioita tässä opinnäytetyössä esitellyistä tekniikoista. Monimutkaisimmat huijaukset saattavat esimerkiksi toimia käyttöjärjestelmän kernel-tilassa, jolloin ne on mahdollista tunnistaa vain siirtämällä huijausten tunnistaminen kernel-tilaan. Täysin huijaamiselta suojattua peliä ei valitettavasti ole olemassa, ja huijarit tulevat löytämään uusia keinoja ohittaa peleihin tehdyt suojaukset. Vaikka kaikkia huijauksia ei voida estää, ei se kuitenkaan tarkoita, että pelit pitäisi jättää täysin suojaamattomiksi. Peleissä huijaaminen on ikuinen taisto pelinkehittäjien ja huijareiden välillä, ja täydellisen huijauksenestoon sijaan pelinkehittäjien kannattaa keskittyä tekemään huijaamisesta mahdollisimman hankalaa ja aikaa vievää.

Lähteet

1. Irdeto Global Gaming Survey. 2018. Viitattu 30.4.2020. https://resources.irdeto.com/irdeto-global-gaming-survey

2. Pritchard, M. 2000. How to Hurt the Hackers: The Scoop on Internet Cheating and How You Can Combat It. Viitattu 12.1.2020. https://www.gamasutra.com/view/feature/131557/how_to_hurt_the_hackers_the_scoop_.php

3. Carpenter, N. 2019. Teenage Fortnite player continued to cheat, even after being sued by Epic Games. Polygon. Viitattu 30.4.2020. https://www.polygon.com/2019/10/14/20913710/epic-games-fortnite-cheating-lawsuit-continues

4. Yan, J. & Randell, B. 2005. A systematic classification of cheating in online games. 1–9.

5. Webb, SD. & Soh, S. 2007. Cheating in Networked Computer Games: A Review. 105–112.

6. Cano, N. 2016. Game hacking: developing autonomous bots for online games. San Francisco, USA: No Starch Press.

7. Rhysider, J. 2017. Darknet Diaries EP 7: Manfred (Part 1). Podcast. Viitattu 30.4.2020. https://darknetdiaries.com/episode/7/

8. Yosifovich, P., Russinovich, ME., Solomon, DA. & Ionescu, A. 2017. Windows Internals Seventh Edition Part 1: System architecture, processes, threads, memory management, and more. Redmond, Washington, USA: Microsoft Press.

9. Shpigor, I. 2018. Practical video game bots: automating game processes using C++, Python, and Autoit. New York, NY, USA: Springer Science+Business Media.

10. Cheat Engine. Viitattu 20.1.2020. https://www.cheatengine.org/

11. AutoIt. Viitattu 20.1.2020. https://www.autoitscript.com/site/

12. Microsoft dokumentaatio. 2018. SendInput function. Viitattu 30.4.2020. https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-sendinput

13. Microsoft dokumentaatio. 2018. KEYBDINPUT structure. Viitattu 30.4.2020. https://docs.microsoft.com/en-us/windows/win32/api/winuser/ns-winuser-keybdinput

14. Microsoft dokumentaatio. 2018. SendMessage function. Viitattu 30.4.2020. https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-sendmessage

15. Forshaw, J. 2018. Attacking network protocols: a hacker's guide to capture, analysis, and exploitation. San Francisco, USA: No Starch Press.

16. Eilam, E. 2005. Reversing: secrets of reverse engineering. Indianapolis, IN, USA: Wiley.

17. ILSpy. Viitattu 20.1.2020. https://github.com/icsharpcode/ILSpy

18. IDA Pro. Viitattu 20.1.2020. https://www.hex-rays.com/products/ida/

19. x64dbg. Viitattu 20.1.2020. https://x64dbg.com/

20. dnSPy. Viitattu 30.4.2020. https://github.com/0xd4d/dnSpy

21. PolyHook 2.0. Viitattu 30.4.2020. https://github.com/stevemk14ebr/PolyHook_2_0

22. Glazer, JL. & Madhav, S. 2016. Multiplayer game programming: architecting networked games. New York, NY, USA: Addison-Wesley.

23. Wireshark. Viitattu 20.1.2020. https://www.wireshark.org/

24. Hoglund, G. & McGraw, G. 2008. Exploiting Online Games: Cheating Massively Distributed Systems. Upper Saddle River, NJ, USA: Addison-Wesley.

25. Webster, A. 2010. Valve responds to MW2 Steam glitch with free copies of L4D2. ArsTechnica. Viitattu 30.4.2020. https://arstechnica.com/gaming/2010/07/valve-responds-to-mw2-steam-glitch-with-free-copies-of-l4d2/

26. Gambetta, G. Fast-Paced Multiplayer. Viitattu 30.4.2020. https://www.gabrielgambetta.com/client-server-game-architecture.html

27. Karlsson, K. 2016. Obfuscate your C# code in Unity3d. Viitattu 30.4.2020. https://medium.com/@utterbbq/in-this-article-im-writing-about-how-you-can-obfuscate-your-c-code-in-unity3d-e829e7b881c

28. Unity dokumentaatio. 2018. IL2CPP. Viitattu 30.4.2020. https://docs.unity3d.com/Manual/IL2CPP.html

29. De Witte, P. 2016. Game Obfuscation and Security: Why You Should Definitely Obfuscate Your Game. Viitattu 30.4.2020. https://pimdewitte.me/2016/11/06/game-obfuscation-and-security-why-you-should-definitely-obfuscate-your-game/

30. Tully, J. 2008. An Anti-Reverse Engineering Guide. Viitattu 30.4.2020. https://www.codeproject.com/Articles/30815/An-Anti-Reverse-Engineering-Guide

31. Microsoft dokumentaatio. 2018. Enumerating All Modules For a Process. Viitattu 30.4.2020. https://docs.microsoft.com/en-us/windows/win32/psapi/enumerating-all-modules-for-a-process

32. Kang, A., Jeong, SH., Mohaisen, A. & Kim, HK. 2016. Multimodal Game Bot Detection using User Behavioral Characteristics. Viitattu 30.4.2020. https://arxiv.org/pdf/1606.01426.pdf

33. Mitterhofer, S., Krügel, C., Kirda, E. & Platzer, C. 2009. Server-Side Bot Detection in Massively Multiplayer Online Games. IEEE Security & Privacy. 7:29–36.